THE COOPER UNION

ALBERT NERKEN SCHOOL OF ENGINEERING

AN EXPLORATION AND DEVELOPMENT OF CURRENT ARTIFICIAL NEURAL NETWORK THEORY AND APPLICATIONS WITH EMPHASIS ON ARTIFICIAL LIFE

by

David J. Cavuto

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Engineering

May 6, 1997

THE COOPER UNION FOR THE ADVANCEMENT OF SCIENCE AND ART

THE COOPER UNION FOR THE ADVANCEMENT OF SCIENCE AND ART

ALBERT NERKEN SCHOOL OF ENGINEERING

This thesis was prepared under the direction of the Candidate's Thesis Advisor and has received approval. It was submitted to the Dean of the School of Engineering and the full Faculty, and was approved as partial fulfillment of the requirements for the degree of Master of Engineering.

_____

Dean, School of Engineering - Date

_____

Prof. Simon Ben-Avi - Date
Candidate's Thesis Advisor

Acknowledgments

I would like to take this opportunity to thank, first and foremost, my thesis advisor, **Dr. Simon Ben-Avi**. His advice, both as a professor and as a friend, were and always will be invaluable. Moreover, I would like to thank the entire EE department faculty and staff for all the support and encouragement (and toleration) they have shown me throughout the years.

I am deeply indebted to my friend and Big Brother **Yashodhan Chandrahas Risbud** (Yash!). Without the occasional smack in the head he needed to give me, I might not have made it through at all. Thanks for putting up with me.

Kappa Phi – Zeta Psi. My brothers supported me in the hard times and cheered me in the good times. Can anyone ask for more?

Special thanks to **The Leib**, **Seamous**, and of course, my **Muffin**.

My utmost appreciation and thanks to my parents, **George and Doris Cavuto**. What can I say? Thanks for everything. (Especially all that money!)

And finally, a big old **THANKS!** to **Peter Cooper** for giving me a place to work, learn, and grow for the last six years. *Anywhere else would have been just a school. The Cooper Union has been my home.*

– DJC

Disclaimer: this thesis is entirely a product of my imagination. Any resemblance to actual work is purely coincidental.

1. Abstract

The purpose of this study is to explore the possibilities offered by current Artificial Neural Net (ANN) structures and topologies and determine their strengths and weaknesses. The biological inspiration behind ANN structure is reviewed, and compared and contrasted with existing models. Traditional experiments are performed with these existing structures to verify theory and investigate more possibilities. This study is conducted to the end of examining the possibility of using ANNs to create "artificial life," which is defined here as a structure or algorithm which displays characteristics typically only attributed to biological organisms, usually nonrepeating, nonrandom processes. Although some ANN topology is shown to be highly similar to that of biological systems, existing ANN algorithms are determined be insufficient to generate the desired type of behavior. A new ANN structure, termed a "Temperon", is designed, which encompasses more properties in common with biological neurons than did its predecessors. A virtual environment based on turtle graphics is used as a testbed for a neural net built with the new type of neuron. Experiments performed with the Temperon seem to confirm its ability to learn in an unassisted fashion.

Table of Contents

Table of Figures

2. Background

2.1 Biological Nature of Neural Cells

Since a good deal of this thesis centers around an mathematical construct known as an *artificial neural net* (ANN), it will be useful to expand upon the nature of these constructs, as well as the physiological inspiration for their design. As such, we look into the structure of actual nerve cells as found in many living creatures, including man. It is important to point out that although many of the following explanations are generally accepted as fact in the field of neuroscience, the ultimate function of each of the component parts of the nervous system is not precisely defined, and many are not even known for certain. However, some such conclusions are repeated here for the purpose of providing insight into the development of artificial neural structures. More discussion regarding ANNs can be found in a later chapter.

2.1.1 Physical Structure of biological neuron

For simplicity and familiarity, the following discussion will center around human neural structures. However, the facts and conclusions presented herein do generalize to most of the animal world. In fact, we will later look at one specific example of these neural structures in a snail known as *Aplysia*. Also, please note that the following text is in no way a complete description of neural cell structure and behavior. It merely attempts to familiarize the reader to the specific structures and processes which inspire the work in later chapters.

2.1.1.1 Body, Axon, Dendrites, Synapse

Nerve cells are thought to be the main processing element in our central nervous system. Humans generally have about 100 billion nerve cells in the entire nervous system.

The nerve cell, or *neuron*, has four general regions, each defined by its physical position in the cell as well as its function. The cell body, or *soma*, provides the basic foundation on which the other parts of the cell can grow. It also provides the basic life-supporting functions characteristic of any biological cell -- nourishment, replenishment, reproduction, etc.

The cell body has two types of interconnection structures which emerge from it: *dendrites* and the *axon*. Each neuron generally has only one axon, but typically has many dendrites. The axon carries the nerve signal away from the cell body to other neurons. Dendrites carry signals in towards the cell body from the axons of other neurons. As such, the basic nerve cell can be

thought of as a “black box” which has a number of inputs (dendrites) and only one output (the axon). This analogy, though not technically complete, forms the basis neural network theory as explained later.

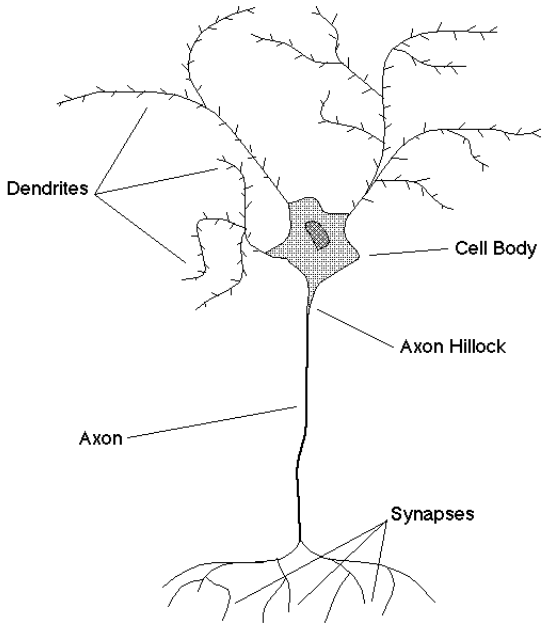


Figure 2–1. A typical nerve cell [Guyton, 4].

The point at which the axon of one cell interconnects with a dendrite of another cell is known as a *synapse*. At the point of interconnection, the axon forms a node called a *presynaptic terminal*. This terminal lies on the surface of a dendrite of another nerve cell. When the axon is stimulated, the presynaptic terminal releases a substance known as *neurotransmitter*, which flows from the terminal to the adjacent dendrite. This chemical causes the dendrite to become stimulated which thereby stimulates the second neuron. For clarity, we will refer to the neuron which initiates a signal and releases neurotransmitter the *transmitting* neuron, and we will call the neuron which reacts to the neurotransmitter the *receiving* neuron. Although strictly speaking, this terminology is not technically accurate, from a signal processing and information theoretic point of view it will provide a more clear analogy when looking at its ANN motivation.

2.1.1.2 Neurotransmitter

A substance called a *neurotransmitter* flows across the gap from one neuron to another, thereby acting as a chemical “bridge” for the neural signal. This gap is consequently known as

a *chemical synapse*. It is important to note that in a chemical synapse, the signal always flows one way; that is, from the presynaptic terminal of one neuron to a dendrite of the postsynaptic neuron. This phenomenon helps us isolate the exact nature of the signal transmitted.

There does exist a structure known as an *electrical synapse*. However, there are very few of these relative to chemical synapses in the central nervous system, and their significance is not known.

When a neural signal reaches the presynaptic terminal, it causes a depolarization in the terminal voltage-gated calcium channels. The terminal contains a large number of these channels, which subsequently release a large number of calcium ions into the terminal. The transmitter vesicles, which synthesize the neurotransmitter, then bind with the neuron's cell membrane and finally spill their contents to the exterior of the cell — a process known as *exocytosis*.

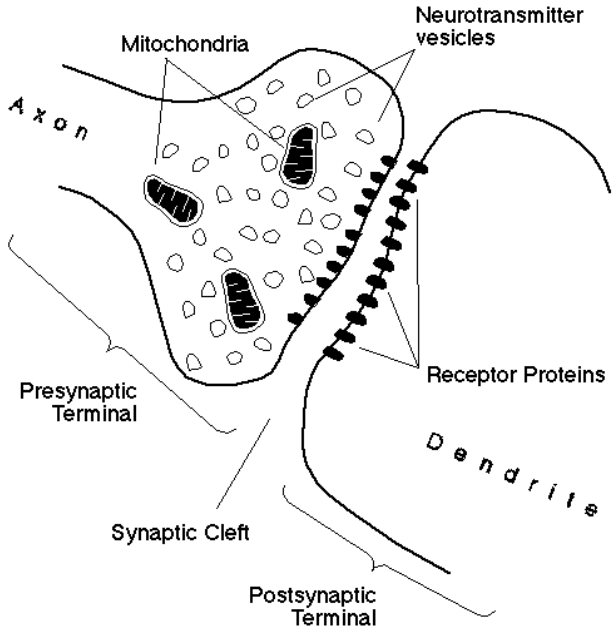


Figure 2–2. A magnification of the structures present in the synapse [Guyton, 126].

In the intercellular gap, the neurotransmitter travels across to the membrane of the postsynaptic, or receiving, neuron. The gap is typically on the order of two to three hundred Angstroms wide. Once across the gap, the chemical interacts with a number of receptor proteins located on the surface of the receptor neuron. These receptor proteins protrude through the cell membrane and provide a binding terminal for the neurotransmitter. The

component of the receptor proteins located inside the cell is known as the *ionophore*. The ionophore relays the signal received from the neurotransmitter to its own nerve cell.

One type of ionophore is a chemically activated ion-channel. This conducting channel passes the signal along from the transmitting neuron to the receiving one. However, it comes in three varieties: (1) sodium channels, which allow sodium ions to pass, (2) potassium channels, which allow potassium ions to pass, and (3) chloride channels, which allow mainly chloride ions to pass. As it turns out, sodium ions excite the receiving neuron, but both potassium and chloride inhibit the receiving neuron. The type and number of ion-channels in a dendrite varies with each dendrite in a neuron, and with every neuron in the body. However, as we look at how nerve impulses are conducted, we will see that the processes of *inhibition* and *excitation* are of pivotal importance when trying to model the behavior of the neuron.

2.1.1.3 Sodium/Potassium Pump

Nerve impulses, though conducted across intercellular gaps by way of chemical neurotransmitter, are primarily electrical in nature. The nature of these signals and their generation is the focus of the next portion of our discussion.

The fluids contained inside the nerve cells, as well as the fluids in which the cells are suspended, are both highly electrolytic solutions, containing a fair concentration of both positive and negative free ions, in approximately equal proportion. However, in the neuron rest state, a very small excess of negative ions accumulates along the neuron’s inner membrane, and an equal number of positive ions accumulate along its outer membrane. This slight concentration difference creates a voltage difference from the inside to the outside of the cell, specifically across its membrane, known as the *membrane potential*.

It is taken as convention to measure the voltage relative to the extracellular fluid, which is designated zero volts. When the neuron is at rest, typically the membrane potential is on the order of -65 mV, although it can range from -40 to -85 or -90 mV in different nerve cells. The rest state of a neuron is defined as the state when it is neither inhibited nor excited by the neurons connected to it.

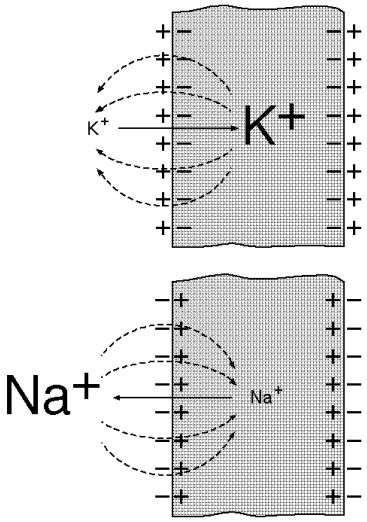


Figure 2–3. Diffusion of ions due to concentration gradients and varying permeability of membrane results in a *membrane potential* [Guyton, 64].

A nerve signal propagates from the cell body, along its axon, and ends in the presynaptic terminals at the end of the axon. The signal takes the form of a change in the polarity of the rest membrane potential. This polarity change is a result of a change in the permeability of the membrane to potassium and sodium.

Normally, the cell membrane is selectively permeable only to positive potassium ions, which flows out from the cell to the extracellular fluid, where it exists at a lower concentration. Similarly, positive sodium ions flow into the cell when the membrane is permeable to sodium.

However, when the neural signal is initiated, it causes a depolarization in the cell membrane which propagates the length of the axon by means of both the selectively permeable membrane and the sodium-potassium pump. This process derives its name from the action of voltage-gated ion channels along the length of the axon which pump sodium and potassium against their diffusion gradients out of and into the cell, respectively.

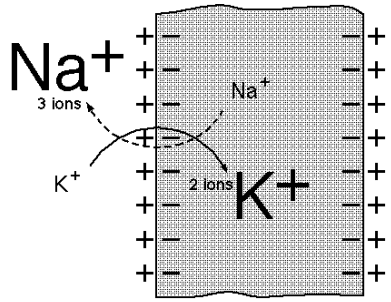


Figure 2–4. Sodium-potassium pump moves ions against their gradients to create a depolarization of the normal rest membrane potential [Guyton, 64].

The ion channels make use of internal energy in the cell to perform this "pumping" action. The process converts one molecule of ATP into ADP and uses the energy gained to push three sodium ions out of the cell, and pull two potassium ions into it. The change of concentrations of these two ions results in a drastic change in potential across the membrane, usually causing a resulting voltage increase from its rest state of -65 mV to -25 or -30 mV, but sometimes can even cause the potential to become slightly positive.

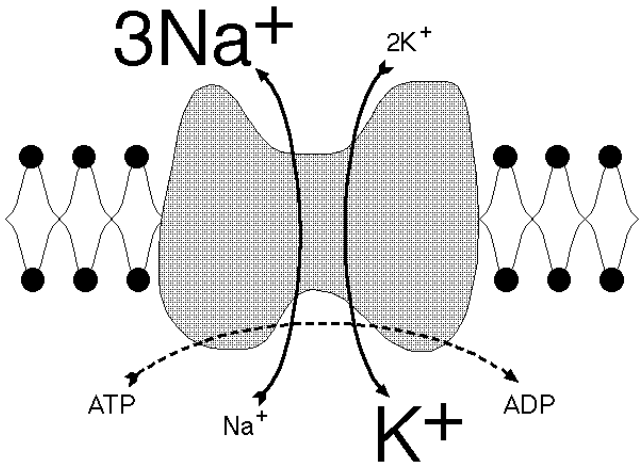


Figure 2–5. Conversion of ATP to ADP in action of ion pump to exchange three sodium anions for two potassium ones [Guyton, 68].

2.1.1.4 Ionized pulse

The depolarized region occurs over a small portion of the cell membrane, usually along the axon. However, this drastic change in polarization, known as the *action potential*, across the cell membrane affects adjacent portions of the membrane. The voltage-gated ion channels in

local portions of the cell membrane become active, performing their ionizing pumping along their length of membrane. This propagation continues along the entire length of the axon.

Each ion channel can continue its pumping action for a very short period of time, after which the cell diffusion processes take over to return the cell to its rest state. The membrane then becomes highly permeable to sodium ions, which rush in and hyperpolarize the membrane, making the potential across it more negative. This sudden influx causes the membrane to actually overshoot its rest potential, and fall to -90 or -95 mV.

As suddenly as this happens, it stops. The membrane becomes permeable to potassium ions again, which rush out to even the concentration differences create by the "pump." Gradually, the section of axon returns to its rest potential.

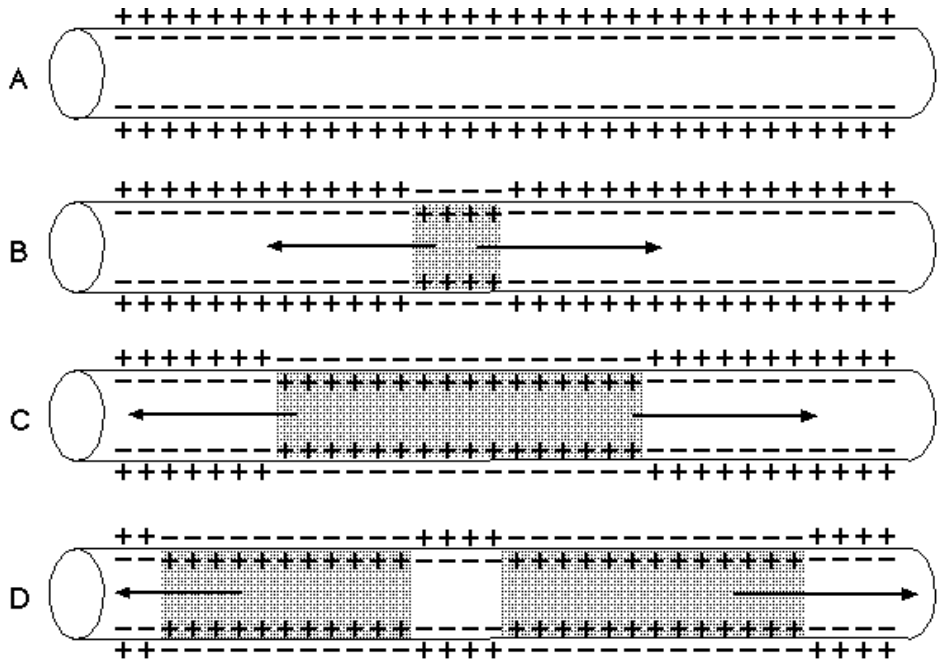


Figure 2–6. Depiction of depolarized zone propagating along a nerve fiber. Note the propagation here is in both direction because the stimulus is initiated in the middle of the fiber. Normally, the propagation is in one direction (from the soma and down the axon) only [Guyton, 74].

This process spreads along the length of the axon. The action potential propagates down the axon like an electrical pulse radiating out from its source. Since the depolarization can only last a very short period of time in any one region, due to the need for energy from ATP to power the pump, each depolarized region quickly returns to its rest potential. Hence, the net effect is a "pulse" of depolarized region propagating down the length of the axon. Actually, the

pulse propagates outwards from the source of causation, but the source is usually immediately adjacent to the cell body, which does not depolarize, so the pulse only ends up flowing the other direction.

2.1.1.5 All-or-Nothing Causation

The action potential is initiated in the dendrites and the soma. We recall from our discussion of the processes in the post-synaptic terminal in the dendrites, the presence of neurotransmitter in the interneuron gap causes the receptor ionophores to activate, each of which can be either excitatory, or inhibitory.

The excitatory ionophores allow the cell body membrane to become more positive, by causing its membrane to become more permeable to sodium. The inhibitory ones allow the membrane to pass more potassium, making the membrane potential more negative.

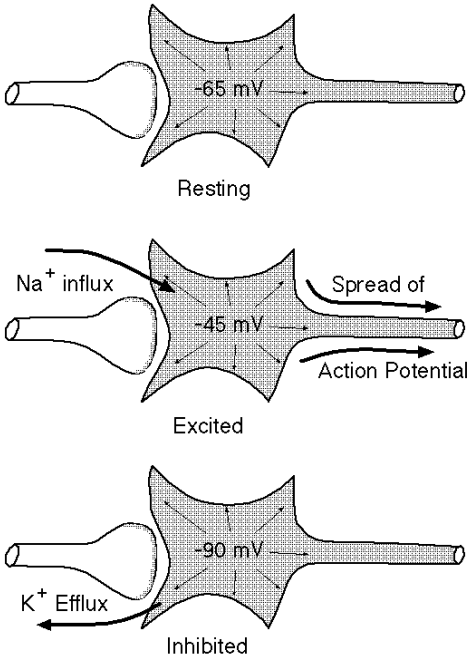


Figure 2–7. Excitatory and inhibitory stimuli at the synapse [Guyton, 131].

At any given time, a number of these different reactions may be occurring simultaneously at different locations around the soma and dendrites. However, the net effect on the soma membrane is an overall change in potential relative to its environment.

Furthermore, the extent to which a certain dendrite affects the overall potential is directly proportional to its proximity to the cell body. The further away from the cell body along the dendrite the stimulus occurs, the less effect it has on the cell. As such, the cell body can be thought of as acting as a *summing amplifier*, which multiplies each incoming signal by an attenuation factor, and then sums the resulting values. This analogy is discussed in depth in the next section.

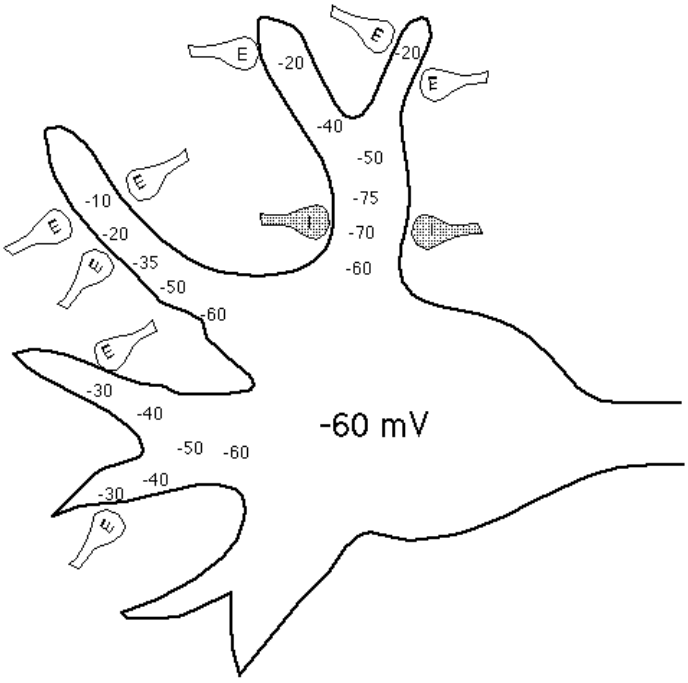


Figure 2–8. Summing action of soma presented with both excitatory (E) and inhibitory (I) stimuli [Guyton, 134].

It is this net change which creates the action potential. When the cell body potential rises a certain amount (usually about 20 mV) it triggers the set of sodium-potassium pumps in the axon closest to the cell body. This amount is known as the *threshold* for an action potential. The depolarized pulse then propagates along the axon as previously described, and the cell is said to have *fired*.

It is important to emphasize that this occurrence either happens or it doesn't. Once the first set of pumps start working, the signal *will* propagate along the axon. There is no magnitude or other information contained in the action potential pulse. It simply exists or not. This binary type of operation, coupled with the summing amplifier-type of behavior forms the basis for the neural net theory in the next chapter.

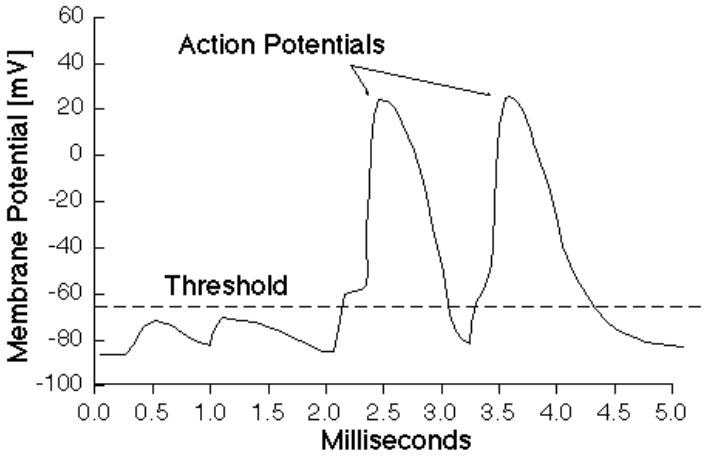


Figure 2–9. Graph showing the all-or-nothing response of the action potential [Guyton, 79].

2.1.2 Mathematical Representation of Nerve Cell processes

It is important to reinforce the correlation of the physical processes and structures found in biological neurons with mathematical constructs. It is these correlations which directly result in the ANN models discussed in later chapters. However, it is equally important to point out that the neural net models DO NOT try to capture the microscopic workings of each neuron exactly as occurs in biological systems. Rather, they try to *model* the critical processes in an attempt to capture the macroscopic behavior of the system.

2.1.2.1 Mathematical correlation to the physical interconnections

Going back to our discussion of the physical topology of a nerve cell, we recall that it is convenient to look at the neuron as a "black box" with a number of inputs, some transfer function, and one output. If we further consider the fact that the action potential propagates along the axon in an "all or none" fashion, we can easily say that the output is binary.

So, we can model the neuron as some function of its inputs. Let's designate each input x_i, where i ranges from 1 to n, the total number of inputs.

Again, recalling from the discussion of the postsynaptic receptors in the dendrites and on the cell body, we note that each input is attenuated proportional to its distance from the cell body. Since this value is related to a physical property of the neuron, it can be considered a constant with respect to the time parameter of the system. Each input has a weight constant associated with it, designated w_i.

Figure 2–10. Each input X is attenuated (or amplified) by a weight constant W, which relates to the physical attenuation imposed at the synapse.

Also note that the ionophores or attenuation in the postsynaptic receptors can stimulate either a sodium or a potassium ion channel. These can either excite the cell or inhibit the cell. This correlates to the sense of the weight. A positive weight is a excitatory constant, and a negative weight is an inhibitory one.

Putting all this together, we can see that each input results in a weighted signal of $x_i w_i$.

2.1.2.2 Linear combination of inputs

Once each of these signals reaches the cell body, they combine additively. Since some signals are positive and some are negative, the net result can be either. Physically, the change is electrochemical, resulting in a linear combination of ions. Therefore, the net result is a linear combination of each of the weighted input vectors, i.e. $\Sigma\, x_i w_i$

Note the linearity of the system in this stage. This is important later when we analyze the system using this model.

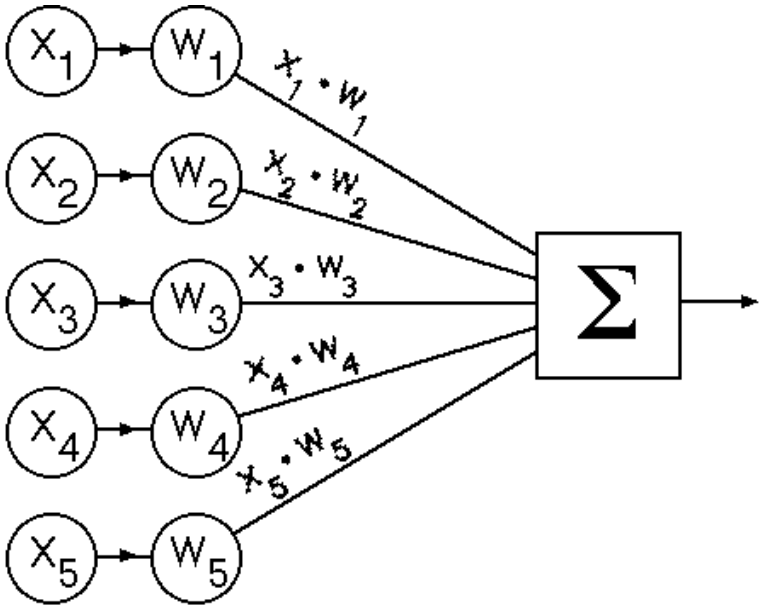


Figure 2–11. Summing amplifier effect at soma can be modeled as a weighted sum of inputs.

2.1.2.3 Thresholding resulting in binary or near-binary outputs

Now that the total inputs are combined in a linear fashion, we need to model the "all or none" response of the neuron. Recall that each neuron has slightly different rest membrane potentials. Additionally, every neuron will "fire", or elicit an action potential, at a slightly different depolarization value. This can be thought of as a *threshold* above which action occurs.

Our model uses some sort of nonlinearity to capture this thresholding value. Usually, the hard-limit function (sometimes referred to as the Heaviside function) is used. This function is +1 when its input is greater than zero and 0 when its input is less than or equal to zero.

However, since we need to encompass a definite threshold value, rather than simply above or below zero, we can define a *bias b* which we add to the function inside the hard-limit to create its discontinuity anywhere along the real axis.

So putting it all together, we get a formula for the output of this model neuron:

$$O = f_{nl}(\Sigma x_i w_i + b)$$

where f_{nl} is a non-linear transfer function, e.g. the hard-limit function.

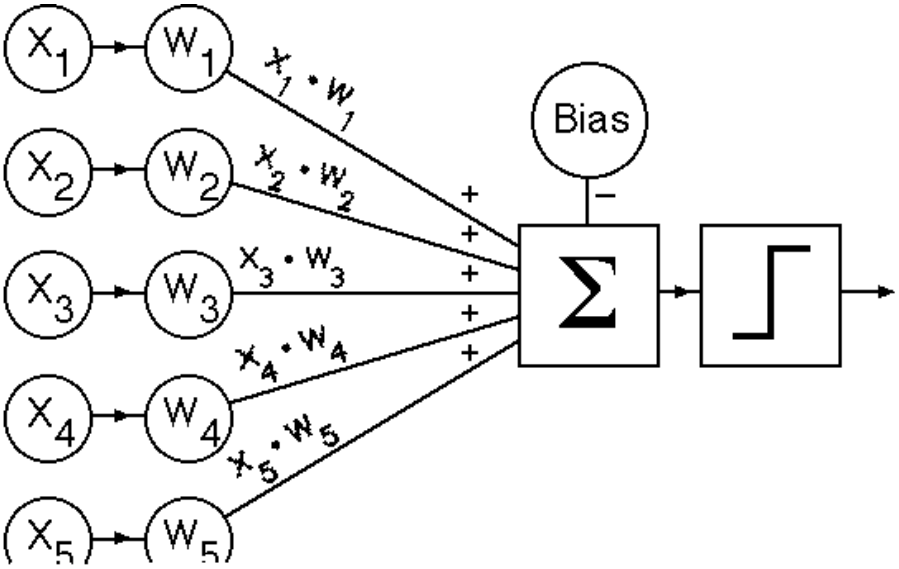


Figure 2–12. Complete block diagram of Neural model, with bias and nonlinear thresholding function.

Note that even though the action of a human cell is strictly “all or none”, the resulting mathematics are sometimes problematic when using a function which is discontinuous at a point and thereby only piecewise differentiable. Therefore it is common to substitute an analytic function such as the TANSIG or LOGSIG sigmoid functions, which allow differentiation through their nonlinearities. The use of these functions will be explored in greater depth in the next chapter.

2.2 Artificial Neural Nets and their Applications

2.2.1 General Theory

2.2.1.1 Purpose

Artificial neural nets, or ANNs, were designed as a simplified model of the biological neurons previously discussed. In an attempt to capture "intelligence," it was theorized that since the human brain was constructed of a number of similarly constructed neural cells, a simulation constructed using these neural models should have similar capabilities. Using the mathematical model previously described, a "neural network" can be designed by putting a number of these mathematical "neurons" together in various configurations.

It is important to point out that generally speaking, a neural net is *not* a set of physical interconnections between finite physical elements. Most of the work done in the ANN field is performed using computer simulations of the algorithms described herein.

2.2.1.2 Structure

The neurons are generally arranged in parallel to form "layers." Although many combinations are possible, configuration usually follows a similar pattern. Each neuron in a layer has the same number of inputs, which is equal to the total number of inputs to that layer. Therefore, every layer of p neurons have a total of n inputs and a total of p outputs (each neuron has exactly one output). This implies that each neuron has n inputs as well. In addition, all the neurons in a layer have the same nonlinear transfer function.

Again, recalling the discussion regarding the structure of each neuron, each one has a number of weights, each corresponding to an input, along with a threshold or bias value. The list of weights can be thought of as a vector of weights arranged from 1 to n, the number of inputs for that neuron.

Since each neuron in a layer has the same number of inputs, but a different set of weights, the weight vectors from p neurons can be appended to form a weight matrix W of size (n, p). Similarly, the single bias constant from each neuron are appended into a vector of length p.

Note that *every* neuron in the net is identical in structure. This fact suggests that the neural net is inherently massively parallel.

2.2.1.3 Weight Updating

Since each of the neurons are identical in structure, the nontrivial information in the neural layer must be stored in the weight matrix and the bias vector. Looking at the model used, we see that these values are the only factors which can change from neuron to neuron; otherwise, each element in a net is identical.

Usually the weights and biases are initialized randomly, so as not to impose a prejudice on the network. Therefore, we must have some facility for changing the weights and biases in a systematic fashion. This is referred to as a *learning rule*, because the process of updating the weights and biases is thought of as *training* the network.

Just as there are many ways to arrange a neural net, there are many learning rules available. Following, we explore a few of the popular ones.

2.2.2 Perceptrons - Classification

Each neuron arranged in a layer in this fashion are referred to as a *perceptron*. Inputs to the layer are applied in parallel to all neural inputs simultaneously. For each neuron k in [1, p], each input x_i is multiplied by the weight w_i, for i in [1, n], and then summed. Each sum is then added to its bias b_k and passed through the nonlinear function. At the conclusion of this process, the layer outputs appear in parallel.

Looking at this process mathematically, we can see that if the inputs are presented as a row vector, the layer outputs can be found by the matrix product $\mathbf{X}\ \mathbf{W} + \mathbf{B}$, which results in a ($p$ x 1) column vector. If we apply the nonlinear transfer function to each of the elements, we get the outputs to the neural layer.

Perceptrons are trained in a *supervised* learning fashion. This means that one tries to train the net to perform a specific, known functions. We have a target test set where the outputs are known, which is used to train the net.

Also note that perceptrons are generally arranged in a strictly feedforward fashion. There are usually no closed loops, or cycles, either.

2.2.2.1 Single Layer

The general procedure for working with a perceptron layer is to first initialize the network as described with random weights and biases. Usually the random numbers are kept small, and symmetrical about zero. Then an input vector is applied to the net, which generates an output.

Since the net has just been initialized, the output is generally "incorrect," that is to say, not equal to the training target vector.

The learning rule is then applied to the layer. A simple learning rule which is widely used is called the Widrow-Hoff rule:

$$\Delta = d(t) - y(t)$$
$$w_i(t+1) = w_i(t) + h\Delta x_i(t)$$
$$d(t) = \begin{cases} +1, & \text{if input from class A} \\ 0, & \text{if input from class B} \end{cases}$$

where $0 \le h \le 1$, a positive gain function. Class A is defined to be when the output is 0 and should be 1; Class B is defined to be when the output is 1 and should be 0. [Beale, 50]

This rule specifies a simple method of updating each weight. It tries to minimize the error between the target output and the experimentally obtained output for each neuron by calculating that error and calling it a "delta." Each weight is then adjusted by adding to it its delta multiplied by some attenuation constant, usually by about 10%. This process is then iterated until the net error falls below some threshold.

By adding its specific error to each of the weights, we are ensured that the network is being moved towards the position of minimum error. And by using an attenuation constant, rather than the full value of the error, we move it *slowly* towards this position of minimum error.

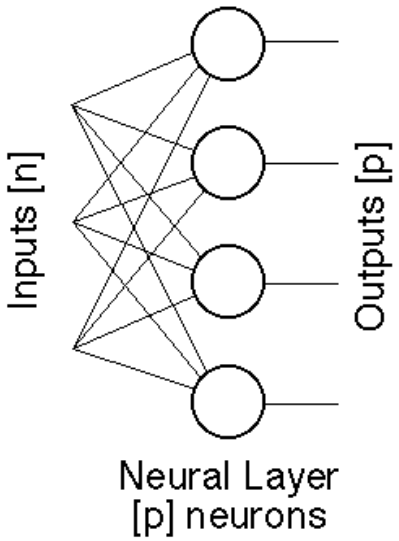


Figure 2–13. Single neural layer. Each circle represents an entire Neural model, each with *n* inputs, and one output.

If we look at the vector of *p* outputs as defining an p-dimensional energy space, we can think of this process of minimizing error as propagating the state of the network to a configuration of lowest energy. The current state of the network is represented as a point on the energy landscape, and the next position of the net should be at a "lower" energy point. This can be visualized in three dimensions as a ball placed on uneven terrain, driven by gravity to find a configuration of lowest energy.

When correctly trained, the perceptron exhibits some highly promising behavior. **A single perceptron is able to learn to classify objects according to their position in n-dimensional hyperspace defined by the *n* inputs.** For example:

A single perceptron is trained with a set of vectors which represent, points in a three-dimensional space. These training vectors could take the form of a (*y* x 3) matrix of *y* ordered triples in a Cartesian space. The goal here is to ascertain whether a given point is or is not a member of some arbitrarily-defined region in space. The corresponding target vector would be single elements, consisting of either "1" or "0," representing if or if not the point is within the region. As each training point and its target output is presented to the neuron, it learns more about the arbitrarily shaped region in 3-space. Soon, it is able to correctly classify points it has never seen before!

This behavior suggests a general task for what the perceptron does, and what it is good for: it classifies. **A layer of *p* perceptrons each with *n* inputs can theoretically learn to classify *p* regions in n-space.**

When perceptrons were first introduced, they seemed revolutionary. Here was a mathematical model of a structure which could be taught to classify points in hyperspace, *not according to rules, but by being shown which points belonged in which* sets. This freed humans from the necessity of determining rules by which the points should be classified — a tricky business in few dimensions, and an impossibility in higher ones. It was thought that they could be trained to solve any problem which could be set up as a classification problem in hyperspace.

Unfortunately, there are some difficulties with this model. The first can be seen readily from the ball analogy. Just like a ball, when released on an uneven surface can come to rest in a local depression and not find a deep, but distant, hole, the net can be trained according to the Widrow-Hoff algorithm and still not find the position of lowest energy. It can "get stuck" in a local minimum. There are, however, more advanced training mechanisms which have been developed to minimize this problem.

Secondly, and more importantly, a careful analysis of the mathematics shows a critical flaw in the perceptron layer: since each element is simply a linear combination of its elements, *each perceptron can only classify linearly separable regions*. This revelation was regarded as the fatal blow to supporters of the perceptron as a panacea.

In fact, the problem became known as the XOR problem, since the exclusive-or logic gate is not a linearly separable algorithm, and probably the simplest one to implement: Look at a 2-D plane, each axis only having two coordinates, zero and one. The axes represent inputs to the XOR, and points placed on the grid represent outputs to the XOR. Place an “X” on the grid locations where the output of the XOR is a one ([1,0] and [0,1]), and place an “O” on the grid where the output is a zero ([0,0] and [1,1]).

Now try to draw exactly one straight line which divides all the Xs from the Os. Impossible.

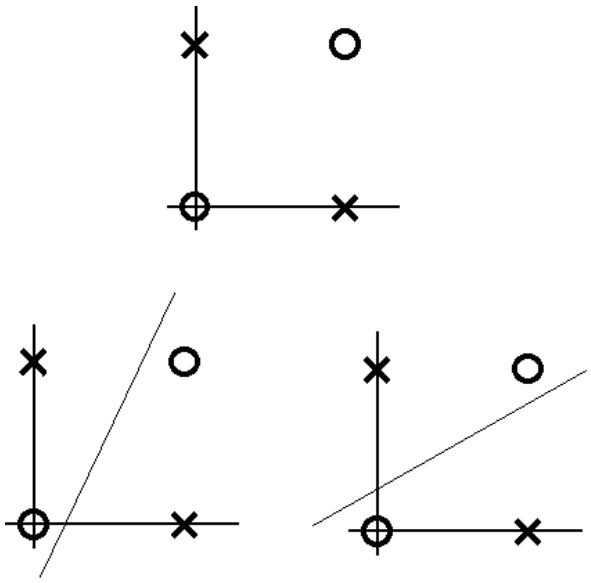


Figure 2–14. Graphical depiction of the XOR problem. The two sets (Xs and Os) are linearly inseparable and therefore cannot be partitioned by a single perceptron layer.

2.2.2.2 MLP - Feedforward

With the discovery that the perceptron was unable to deal with linearly inseparable problems, work on neural nets all but ceased. Finally, in 1986, its was shown that by cascading neural layers, the resulting neural network can be trained to solve arbitrary regions in n-space, not just linearly separable one. An informal proof of this assertion is as follows:

Each neural layer (and indeed, each perceptron) is able to divide a space linearly. Picturing this process in two dimensions would be drawing a straight line across the Cartesian grid, and in three dimensions, like slicing a cube in two pieces along any arbitrary plane. Higher dimensions can be partitioned similarly, but are impossible to visualize.

If, however, we cascade a number of such processes, each succeeding layer can perform another linear partitioning, *but it may be along a different (hyper)plane*. Drawing one set of lines on the grid gives you binary partitioning: A-or-not-A. But if we take this already-partitioned space and further partition it, we can obtain further refinement of the specification region. Then if we take *that* region and partition it once again, we get an even more refined region. And so forth.

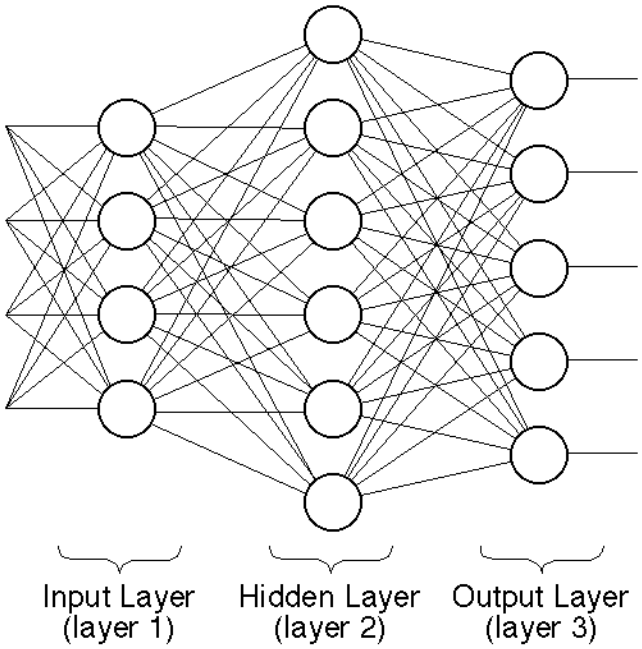


Figure 2–15. Three-layer MLP Neural Network.

In fact, we can show, just as informally, that *only three layers* are necessary to define *any* region in n-space. The first layer allows the drawing of “lines”. The second layer allows the combining of these lines into “convex hulls.” The third layer allows the convex hulls to be combined into arbitrary regions. So we construct our neural network of three layers of cascaded perceptrons. We call this the Multi-layer Perceptron or **MLP**. That’s all there is to it.

Well, maybe not. The question arises: how do we train this three-layer neural network? We obviously cannot use the Widrow-Hoff learning rule. However, the spirit of that learning rule

does in fact hold over to these nets. Specifically, we want to minimize our total error by reinforcing the weights which produce the correct outputs and suppressing those weights which produce the incorrect ones.

Note here that the use of the terms “correct” and “incorrect” imply a similar type of “supervised” learning as discussed before. Therefore, we must have a test set, consisting of input vectors and associated target outputs.

Examining the Widrow-Hoff rule further, we run into a problem: it relies on the fact that a given neuron needs to be aware of the state of all inputs to the net, as well as all the outputs of the net, in order to determine if a weight should be reinforced or suppressed. However, in the three-layer arrangement described, *no* layer has both pieces of information; the first layer has the input information, but no output information — the opposite is true for the output layer. And the middle layer has no information whatsoever.

This dilemma finds its roots in the fact that the hard-limit transfer function provides no information to the output of a neuron regarding its inputs. As such, it is useless for our purposes. To get around this problem, we make use of other nonlinear functions, as mentioned previously, which have transfer functions similar to the hard-limit function, but not its problem-causing discontinuity.

By using these other functions, we are able to obtain information about the state of the inputs in layer layers. Therefore, we are able to correct the weights in the final layer, because we now have information about both the outputs and the inputs.

Once we update the final layer, we can backpropagate the error at third layer to the previous layer. Then we are able to update the weights at the previous layer, and compute its error. That error is then propagated back to the layer before it. This process can continue for as many layers as necessary.

The learning rule described here is known as the *generalized delta rule*, because the error is a “delta” which is propagated back to previous layers.

The algorithm is as follows. Starting from the output and working backwards, calculate the following:

$$w_{ij}(t+1) = w_{ij}(t) + hd_{pj}o_{pj}$$

where $w_{ij}(t)$ represents the weights from node *i* to node *j* at time *t*, *h* is a gain term, and d_{pj} is an error term for pattern *p* on node *j*.

For output units

$$d_{pj} = ko_{pj}(1-o_{pj})(t_{pj}-o_{pj})$$

For hidden units

$$d_{pj} = ko_{pj}(1-o_{pj})\sum_{k} d_{pk}w_{jk}$$

where the sum is over the *k* nodes in the layer above node *j*. [Beale, 74]

2.2.3 Hopfield Net - Pattern Recognition

A Hopfield net is constructed with the same type of neuron model as the perceptron nets; however, the Hopfield net is constructed using a completely different topology. MLP nets are always strictly feedforward. Hopfield nets generally involve feedback, and are usually fully-connected, i.e. every neuron connects to every other neuron.

Imagine, then, a Hopfield network consisting of six neurons. Each neuron should have six inputs, one for each output of the other five neurons, plus one for its own output. This type of topology generates a more active network, which works quite differently from the other nets we have seen.

MLP networks function by presenting them with an input and then generating an output. Hopfield networks have no explicit input or output nodes; rather, any node can function as any other one. In addition, once started, Hopfield networks continue to oscillate for an indefinite period of time, due to the many cycles — closed loops — inherent in their construction topology.

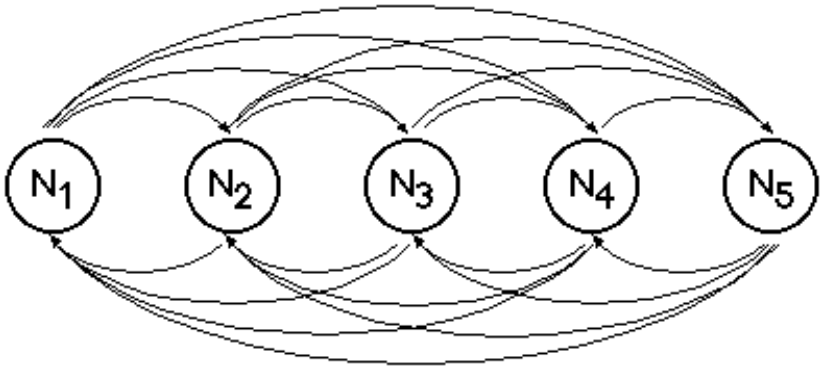


Figure 2–16. Five neurons in a fully-connected Hopfield network. All neuron outputs feed to all other neurons.

To use the net, all outputs are "set" to a given input state simultaneously. The inputs then propagate around the net, generating new outputs. These outputs are fed back into the net and generate new outputs, etc. What will occur is that the net will either settle down to a steady-state, or it will oscillate between a fixed number of states.

If we consider the action of the net to be moving in pattern space, it follows an energy-descent mechanism along the energy gradient. It ends up in a local minimum, which is either unique, or shared by a finite number of states.

This network is trained with a set of inputs which it learns to recognize. Then, when presented with some input, it will oscillate to a steady state of one or a number of its trained states. This process is less like the classification behavior exhibited by the MLP net, and more like a pattern matching-type behavior. In fact, when the net is trained with certain fixed patterns, and it is then presented with the same patterns plus noise, it can iterate to the noise-free patterns. In this capability, it is extremely useful and can be trained to pick out patterns even corrupted with a large amount of noise.

Obviously, the generalized delta rule is not applicable for this type of network; there are no explicit "layers" here. Therefore a different learning rule must be used for the Hopfield net.

2.2.4 Generalizations

It is of utmost importance to note here that there is one powerful similarity among these types of neural nets — that is the necessity of some external learning rule applied macroscopically to the entire system to make its behavior *more correct*. Even so-called "unsupervised" networks make use of some controlling rule which governs the network's behavior. We will see in the next section that there is no parallel to this in biological systems. Therefore, it may be

necessary to evolve a new type of neural model which does not depend on this type of system-wide learning rule.

2.3 Our friend Aplysia

Even though ANNs are demonstrably powerful, we can readily see some of their limitations. Primarily, they are limited by their topology and learning rule. However, since a goal of this thesis is to explore implementations of artificial life, we are specifically interested in neural behavior which comes close to that of living organisms. Therefore, it is useful to explore one exemplary organism which has been used for such a purpose already.

The large marine snail *Aplysia californica* has been studied extensively regarding its neural topology, as well as its learning abilities. Since it has a relatively small number of neurons (about 20,000 as compared to the human twenty billion) it is much less complex, and therefore simpler to study than higher life forms. Other similar invertebrates have also been used for such purposes, with similar results; so, the following discussion will concentrate on the research done with *Aplysia*.

2.3.1 General Observations

One major point observed by the researchers was that in different specimens of *Aplysia* similar neural structures could be found. In fact, with regards to the topology of the nervous system, the specimens were virtually identical. For example, certain sensory neurons were always found to connect to certain motor neurons. Other such similarities were found extensively through a large number of sample specimens. This led researchers to believe that the topology, or interconnections, of neurons were governed largely by genetic construction, and not a function of the experiences or "knowledge" of the specific sample.

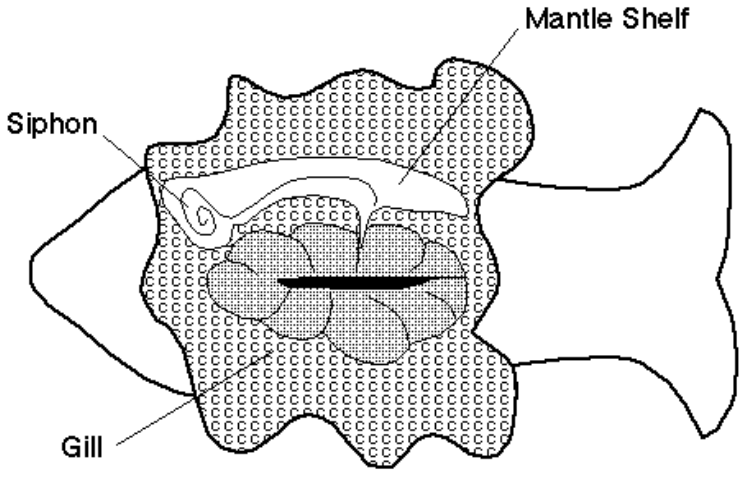


Figure 2–17. Bottom view of *Aplysia Californica*.

Also, they found a number of preexisting behavioral responses, or *reflexes*, which proved useful in later experimentation. One reflex which was extensively used was the "gill-withdrawal" reflex. When the snail was gently poked on its siphon or mantle, it would move its gill to a withdrawn position. This is similar to the human response which quickly moves one's hand away from a hot object.

2.3.2 Summary of Relevant Experiments

2.3.2.1 Habituation

It was observed that repeated stimulations of this gill-withdrawal reflex resulted in differing responses by the snail. At first, multiple stimulations resulted in an equal number of responses. However, as the frequency of stimulations and training sessions increased, *Aplysia's* response gradually decreased, and eventually vanished. This type of decrease in the strength of a behavioral response to repeated stimulation is known as *habituation*.

Several observations should be made regarding habituation in *Aplysia*. Firstly, the snail was shown to exhibit both short-term and long-term habituation responses. When it was stimulated ten to fifteen times in one session, it recovered from its habituation partially in about an hour and almost completely in a day. In this context, recovery from the habituation process is equivalent to "forgetting." However; when stimulated ten times in each of four training sessions, with breaks in between, it remained habituated for weeks afterwards.

It is important to observe changes incurred by habituation on the cellular level. From the brief discussion of the nervous system, we recall that when an action potential reaches the pre-synaptic node at the end of a neuron's axon, the node releases a number of packets of neurotransmitter. After release, the neurotransmitter is regenerated in the node by many neurotransmitter vesicles. This process takes a finite, though small, amount of time. A series of frequent excitations depletes the node's ability to generate and release neurotransmitter. Each subsequent stimulus results in a smaller and smaller number of neurotransmitter vesicles to spill their contents.

Recall now that we mentioned that the action potential is thought of as binary. When the postsynaptic terminals of corresponding motor neurons were examined, it was determined that the corresponding postsynaptic potential of these frequent stimulations *did not change*. Even with smaller amounts of neurotransmitter in the interneuron gap, the potentials generated by the postsynaptic terminal were identical as those generated with larger amounts of neurotransmitter. Therefore the resulting action potential of the receiver neuron remains unchanged for these depressed concentrations of neurotransmitter.

However, it was discovered that this type of long-term repeated stimulation actually leads to a decrease in the number of interconnections between the affected neurons, contradicting the earlier hypothesis that the neural structure is static. Exactly how this process actually takes place is still relatively unknown, but its existence is well documented. It is thought that the decrease in the amount of neurotransmitter signals a secondary control mechanism in the cells which trigger the physical changes.

2.3.2.2 Sensitization

Using the same techniques described above, a similar experiment was conducted; however, at the same time as the mantle was touched, an electrical shock was applied to the body, usually the tail or the head. Adding this type of "noxious" stimulus proved to dramatically alter the type of response elicited from *Aplysia*. Whereas in habituation, the response gradually diminished, here, we see the response greatly enhanced. The gill contracted much more readily and responsively then otherwise.

As with habituation, a number of training sessions had a lasting affect on the subject. Afterwards, the ordinarily mild stimulus of touching the mantle alone resulted in a drastic contraction of the gill, as when the electric shock was applied.

On a cellular level, we see a similar effect to long-term sensitization: more connections from between affected neurons, as well as reinforcement of existing connections. Again, the process which causes these changes is not well understood.

2.3.3 Relevance and relation to neural nets

In *Aplysia* we observe behavior not seen at all in the ANN models we have looked at: **the ability of a neuron or neural circuit to change its own connections or connection weights**. There has been some evidence to suggest that the mechanisms used in the snail which are general neural behavior in animals, and not specific to *Aplysia*. It may then be necessary to evolve a new neural model with the ability to change itself if we want to capture this type of living behavior in a model.

3. Approaches

3.1 General Methods and Tools used

Since the neural network techniques discussed generally function as a simulation, it was necessary to choose a platform and environment in which to design. Two such environments were chosen. The MATLAB suite, by Mathworks, has an optional Neural Network toolbox which includes, among other things, predefined routines for training feedforward 3-layer perceptron networks as well as Hopfield networks. The system, while efficient for working with preexisting network topologies, is difficult to customize.

The portable language Java, developed by Sun Microsystems, was also used as an alternative platform for development. Since Java is a fully functional development language, like C, it allows for fully customized routines to be written. In addition, since Java is oriented to GUI (Graphical User Interface) programming, it allowed for a complex visual representation of the internal state of the network to be designed. It is, admittedly, much slower in execution than MATLAB, an environment optimized for matrix manipulations.

3.1.1 MATLAB ANN toolbox

The Neural Network toolbox has a number of routines which were utilized to explore the response of Perceptron-based networks, whether strictly feedforward, or feedback. MATLAB was used to investigate the various performance parameters of feedforward networks. This toolbox is available directly from Mathworks.

As was previously mentioned, these feedforward networks can be modeled in a set of three matrices representing the weights of each layer and three vectors representing the corresponding biases. MATLAB provides a simple procedure to initialize those matrices:

```
> W1 = RANDS(h, w);
```

initializes a matrix of size (h x w) with small random numbers in the range [-1 1].

Once those three matrices were established, initialized, and trained, the network could be easily simulated with one command:

```
> OUT=SIMUFF(IN,W1,B1,type1,W2,B2,type2,W3,B3,type3);
```

simulates a feedforward network when presented with an input vector IN. The weights and biases are input to the function as the *W* and *B* variables. The *type* variables are strings which designate which type of nonlinear transfer function desired for each layer e.g. `type1 =`

`"LOGSIG";` for a logarithmic-sigmoidal function. Note that the hard-limit function cannot be used here for the reasons discussed in the neural network theory section.

Training a feedforward network with the generalized delta rule and backpropagation is performed by another function:

```
> [W1,B1,W2,B2,W3,B3]=TRAINBP(IN,TARGETS,W1,B1,type1,
  W2,B2,type2,W3,B3,type3,ERR);
```

where TARGETS is the list of target outputs associated with the list of inputs in IN. Note that IN and TARGETS can be matrices which represent a set of different inputs each with their corresponding target output. Here, ERR is the sum-squared error between all the simulated outputs and their corresponding target outputs.

It should be noted that the TRAINBP function has a tendency to get "stuck" in a local minimum and not meet its ERR condition. Reinitialization of the net and retraining will usually help the net converge. However, creating the network with too small dimensions will prevent the net from converging no matter how many different iterations of training sessions are attempted.

In addition, the TRAINBPX function trains the network to similar conditions, but with an accelerated algorithm using momentum and small random perturbations to help the net converge quickly. This function avoids many of the local minima which create difficulties with the TRAINBP function. Otherwise, the TRAINBP and TRAINBPX functions perform identically.

All the above MATLAB functions are included in the Neural Network toolbox provided by Mathworks.

Finally, a new custom function was developed by the experimenter to dump the output of a certain type of network. This function, called "TestNet" produces an intensity matrix which represents the output of one output neuron when varying two inputs to the net, represented by the Cartesian coordinates of the corresponding grid square. This matrix is then displayed as a gray-scale grid. For example, if a net is presented with the input vector [2,1] and generates an output of 0.4. Then the corresponding TestNet gray-scale plot would have a 40% gray square at location (2,1). The TestNet function has a resolution parameter which allows the step size to be varied, thereby displaying the net at greater detail. Unfortunately, the time required to generate the full test is proportional to the square of the resolution parameter; therefore, it is useful to keep it low unless looking for a specific network detail.

3.1.2 Java

Java is an extremely object-oriented programming language designed to be portable over many of the platforms in use today. Its class-based structure makes it an excellent choice for implementing the neural structure discussed because the structure of neuron behavior can be inherited from one model to another. This allows a general neuron structure to be specified and then specific types of behavior to be added simply, with minimal redundancy.

3.1.2.1 Neuron Package

A `neuron` package was written in Java. The base class, the `Neuron` class, is an abstract class which specifies general neuron behavior. It specifies an `output()` method which is called and returns a real number which is the output of the neuron. It also specifies an `addInput()` method which is used to connect the output of other neurons to the input of this neuron. Also, the neuron class contains a number of private instance variables. There is a Vector which holds the weights and a real variable which holds the bias, in addition to the Vector of references to the neurons connected to it. With this arrangement, when a neuron's output is requested, it can then query the outputs of each neuron connected to it.

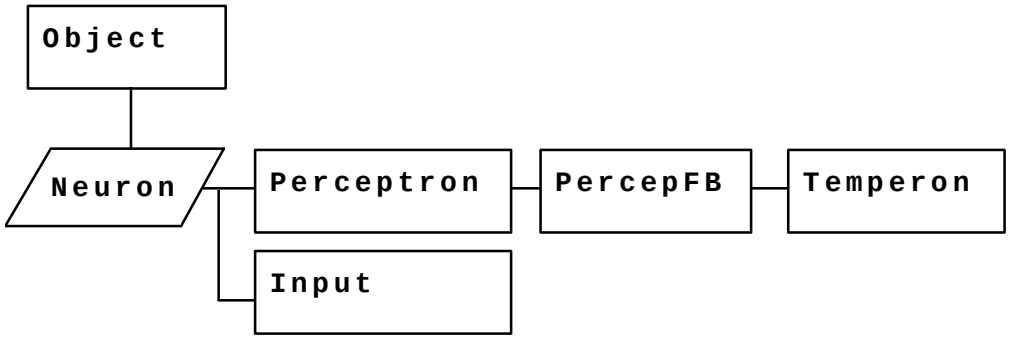


Figure 3–1. Class hierarchy for the Neuron package. All classes inherit from the Neuron abstract class.

To actually instantiate the `Neuron`, it was subclassed into specific types of neurons. A `Perceptron` class was written which implemented the specific behavior necessary to sum all the inputs, pass them through a nonlinearity. This class can be instantiated to create an actual neural net. Building the network is as simple as instantiating the correct number of

Perceptron objects, then connecting them to each other. An `Input` class was also written, to provide a dumb input layer for feedforward networks.

This structure technically allows the programmer to connect a perceptron to itself, by calling the `addInput`() method with `"this"` as the parameter. Unfortunately, a subsequent call to the `output`() method will cause the program to hang, because of the infinite loop incurred by an object's method calling itself.

To get around this limitation, a `PercepFB` class was written to extend the `Perceptron` class. This class functions similarly to the `Perceptron` class, but it allows feedback-type connections.

In addition, a `Temperon` class was written to subclass the `PercepFB` class. More discussion about the `Temperon` class can be found later.

3.1.2.2 TurtleMouse Environment

Since we are interested in using a neural net as the "brain" of some artificially intelligent device, we needed a device to put the brain in. At first, it was thought that an actual physical robot was an appropriate testbed to use as the end construct. However, after a series of explorations, it was abandoned as being unnecessarily complicated, in addition to the fact that it is much more difficult to control the environment in which a robot operates.

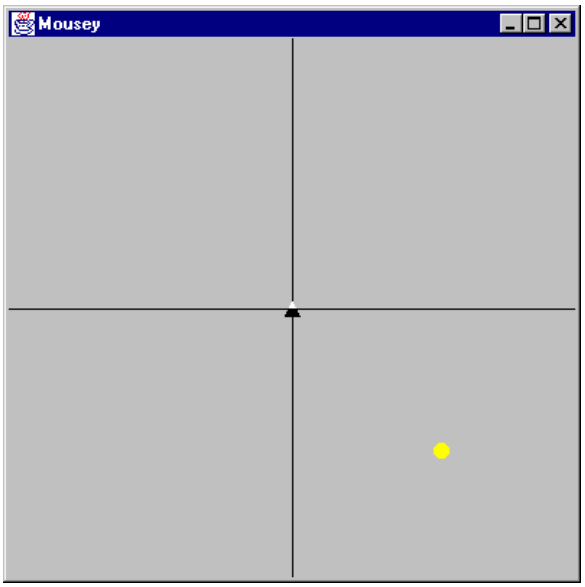


Figure 3–2. TurtleMouse virtual environment window. Turtle (triangle in the middle) moves around window, leaving a trail behind. Window edges loop around.

Therefore, it was decided that a virtual environment could be designed, in which a virtual object could interact. This environment, written in Java, is essentially a port of the turtle-based environment LOGO. The environment was encapsulated in an object which can be instantiated in another environment.

It involves a turtle or mouse-like object shaped like a triangle with the capability to move about its environment, which is essentially a resizable window. The TurtleMouse moves about according to simple commands. It can move forward or backward a certain number of pixels, or rotate about its center a certain number of degrees. In the environment, the TurtleMouse wraps around the borders of the window. The commands are public instance methods which can be called from another object. This allowed the TurtleMouse environment to be used as a "black box" which can be controlled from another Java object.

In addition, the environment has the ability to add an "object", which is essentially a colored dot placed somewhere in the environment. The turtle interacts with the dot through another set of boolean methods which react to the position of the dot relative to the mouse. For example, one function returns true if the object is in the 180° half-plane in front of the mouse. Another returns true if the mouse is within a certain distance from the object. This ability gives the mouse "senses" which can be used as inputs to the neural network.

3.2 Perceptron exploration

In an effort to better understand the workings in the neural net structure, a number of explorations of simple perceptron networks were performed. These investigations were performed to get a sense of the capabilities of perceptron-based networks, and to see if perceptrons can possibly be used in the context of artificial life.

3.2.1 Explorations

A set of MATLAB scripts were written to demonstrate the three-layer network's abilities. As an example task to train the net, a 2-dimensional classification technique was chosen. The task can be described as follows: given an arbitrary coordinate pair in 2 dimensions, choose whether that point is or is not in some arbitrary set A. As more points are presented, set A becomes more and more specified. Some time later, if an unknown point X is presented to the net, the net should be able to classify point X as being in or not in set A as appropriate.

This technique was applied to the XOR problem previously described. A network of size [3,1,1] was able to classify the two selection sets relatively easily.

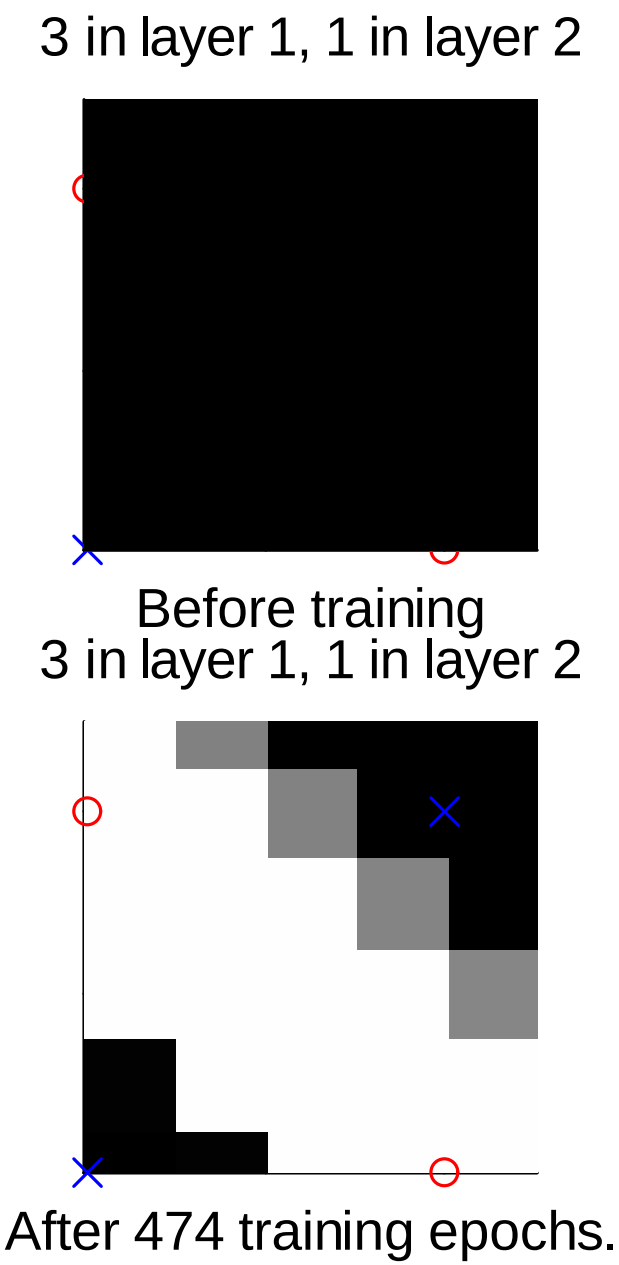


Figure 3–3. The XOR problem (above) and its solution (below) illustrated graphically.

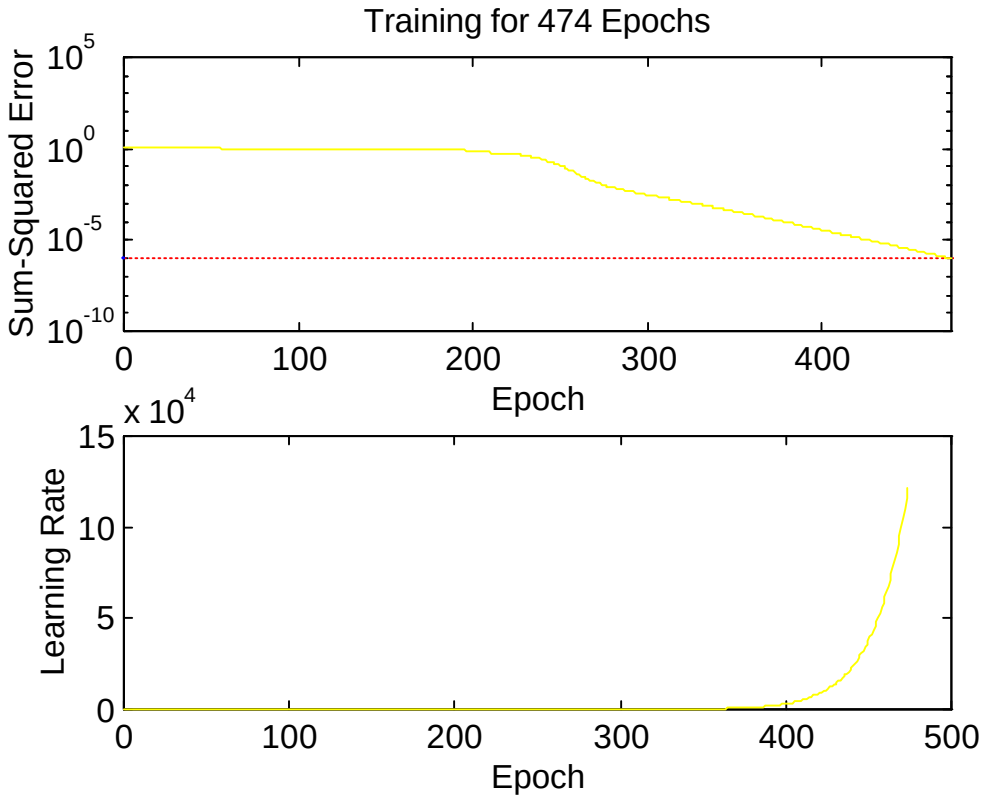


Figure 3–4. Training statistics for the XOR solution (with random initialization).

We can similarly examine more arbitrary arrangements of points in the inclusion and exclusion sets. For example: say that set A describes points in a square-shaped region between (2,2) and (6,6) in the Cartesian plane. The net is trained with points (2,2), (2,6), (6,6), and (6,2) as being in set A, and points (0,0), (8,8), (0,8) and (8,0) as not being in set A. When presented with the point (4,4), the net sho

uld respond that the point is IN set A. If not, it requires more examples or training points to further specify the region.

This type of task requires a certain fixed network structure: two inputs are necessary— one for each coordinate. Also, exactly one output is necessary. This allows the net to have a binary response; that is, in or not in the specified set. The latter constraint specifies that layer three have only one neuron in it. However, layer one and layer two may vary in size. These sizes were varied as part of the exploration.

Also, as is evidenced by the trivial example, the classification learned by the network is relative to the number of points in the test set. Various sets of test points were trained while keeping

other network parameters constant. This allowed us to determine the sensitivity of the shape of selection region to the number and location of points.

For example, an arbitrary selection region was chosen. Then, points both inside and outside the region were used as training data. The resulting network was then probed to determine the classification region learned by the specific training session. This procedure was used to experimentally verify the mathematical proof that a three-layer network can determine an arbitrarily shaped region in space.

3.2.1.1 Test Set 1 - Network Size Limits

The first test set was chosen to test the sensitivity of the network's ability to converge while varying the number of neurons in the critical layers of the network. This test set involved ten test points, five in the selection region and five not in it. The five points in the selection region were arranged in a roughly kidney-shaped pattern. The size of the net was then varied to see at what point it would fail to converge. Recall that the variables in the size of the network were the size of layer one and layer two. Layer two was reduced first, then layer one was reduced, until the net failed to converge. After the net failed to converge, the other layer was increased in size to see if it could make the net converge.

After a number of test simulations, it was found that the minimum specification for the five points in the inclusion set was four neurons in layer one, and one neuron in layer two. Once this set was trained, it was viewed using the `NETDUMP` function. This revealed that the resulting selection region was specified by four linear regions. These regions seem to correlate to the four neurons in layer one. The one neuron in layer two and one in layer three do not provide any additional specification.

In addition, it was found that when the size of layer one was reduced to below four neurons, absolutely no increase in the size of subsequent layers had any effect on the convergence of the network. Even increasing layer two to 500 neurons did not allow the net to converge.

3.2.1.2 Test Set 2 - Disjoint Set A

The second set was a test to see if an arbitrary arrangement of points would converge. The arrangement of points was chosen such that only a disjoint selection region could encompass all points in the set and none of the points not in the set. In fact, a ring-shaped region — with an open center — was desired. Again, the number of points in the network was kept constant, while the size of layer one and layer two were varied.

After increasing both the number of neurons in layer one, then in layer two, it was found that none of the test selection combinations would converge at all. Even when the network was increased in size to 50 neurons in both layer one and layer two, the net still failed to converge.

It did, however, generate true disjoint selection regions. This demonstrated that the three-layer network could in fact combine the linear divisions in an arbitrary manner.

3.2.1.3 Test Set 3 - Enclosed Region

Since all of the previous tests resulted in “unbounded” selection regions, it was decided to temporarily abandon the ring-shaped set in favor of a more simple disk-shaped set.

Here, the number of neurons was chosen to be a more reasonable size as more points in the selection region are added. The net was tested with 5 neurons in layer one and 15 in layer two.

After a number of iterations, the net did converge; however, the TestNet function revealed that the resulting region was triangularly shaped, and not disk-shaped. Also, the region still seemed unbounded at some points. More points were added to the not-A region in an attempt to “fence it in.”

After almost triple the number of points were added to the not-A set, the net refused to converge. The network was then expanded to 25 neurons in the second layer and retrained. This resulting net converged and generated the desired disk-shaped selection region. That’s not to say that the region was perfectly circular; rather, it was a simple closed convex hull with finite bounds.

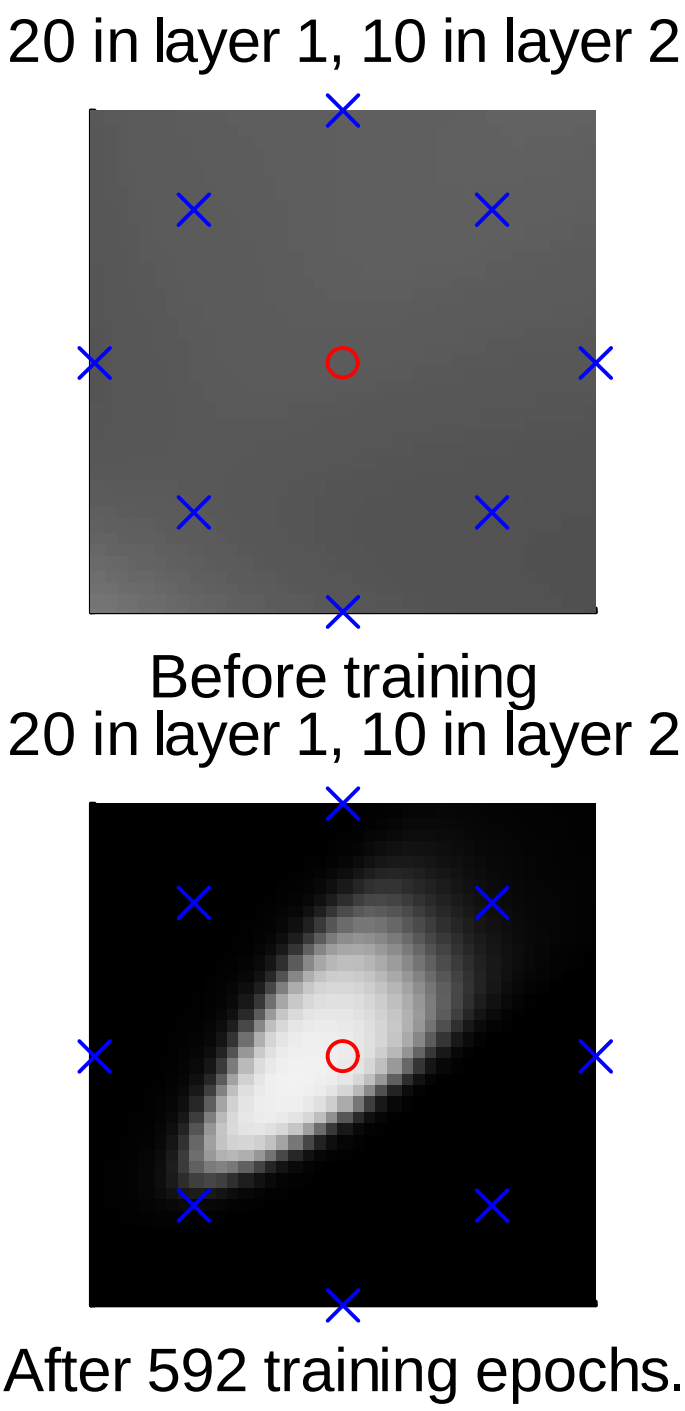


Figure 3–5. (Roughly) circular selection region solution.

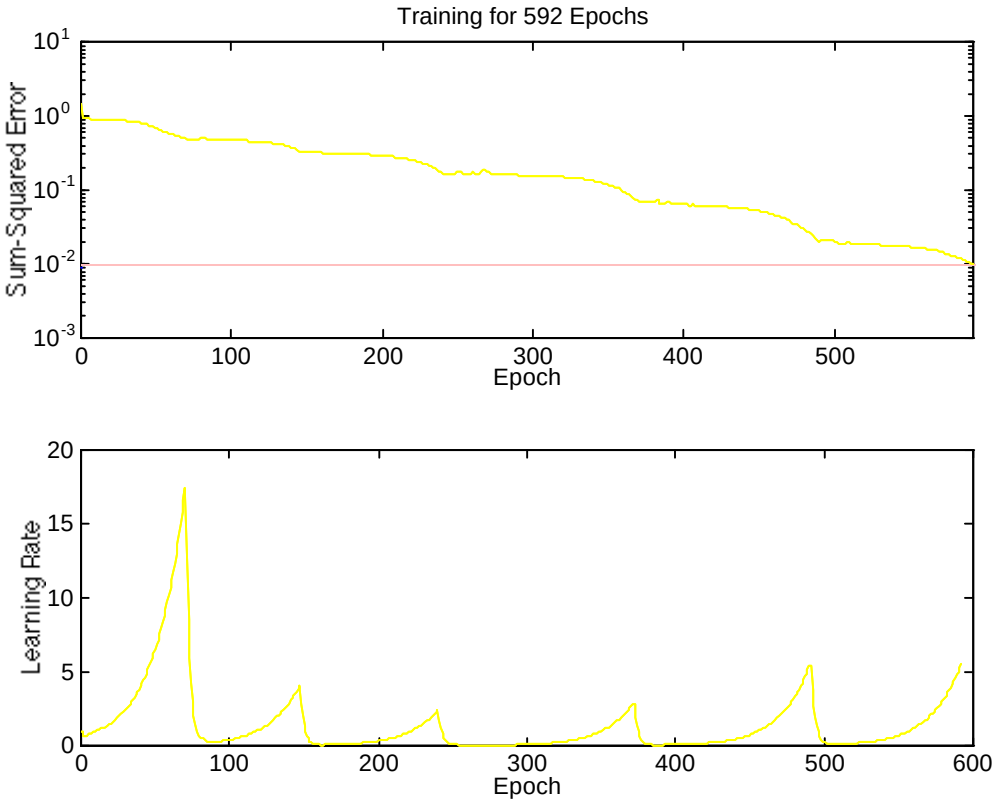


Figure 3–6. Training statistics for the circular selection region solution.

3.2.1.4 Test Set 4 - Disjoint Set B

The final test shows the ultimate ability of the network to in fact classify completely arbitrary regions, here in 2-space, but in general in N-space, even disjoint regions or regions with "holes". Now that we have shown that the net can in fact learn to classify a finite disk-shaped region, we return to the disk-shaped selection region. This time, we make use of the observations from the previous test sets. A region is defined by a square of points in set A (four points), with a larger concentric square in set not-A. This set does converge.

Next, a point is added to the not-A set, at the center of the set-A square. This immediately causes difficulty for the perceptron training algorithm. The network does converge, but the selection region is highly unusually shaped. In the quest to achieve the desired selection region, more points were added to the not-A set. As was observed above, this caused the net to refuse

to converge. So more neurons were gradually added to both layers as more points were added to the region.

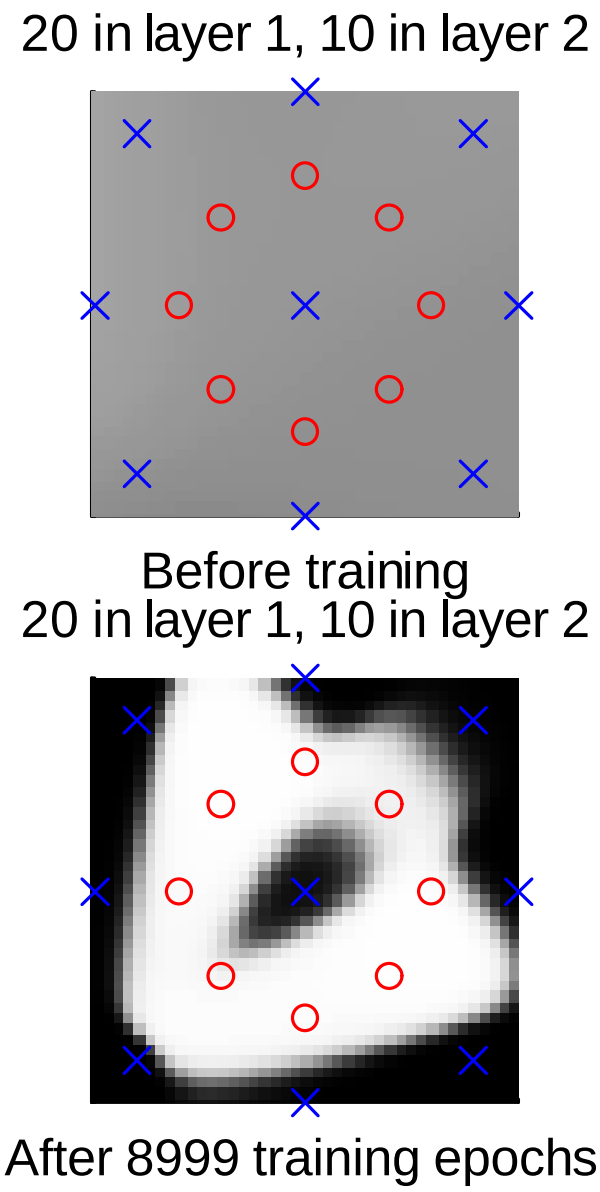
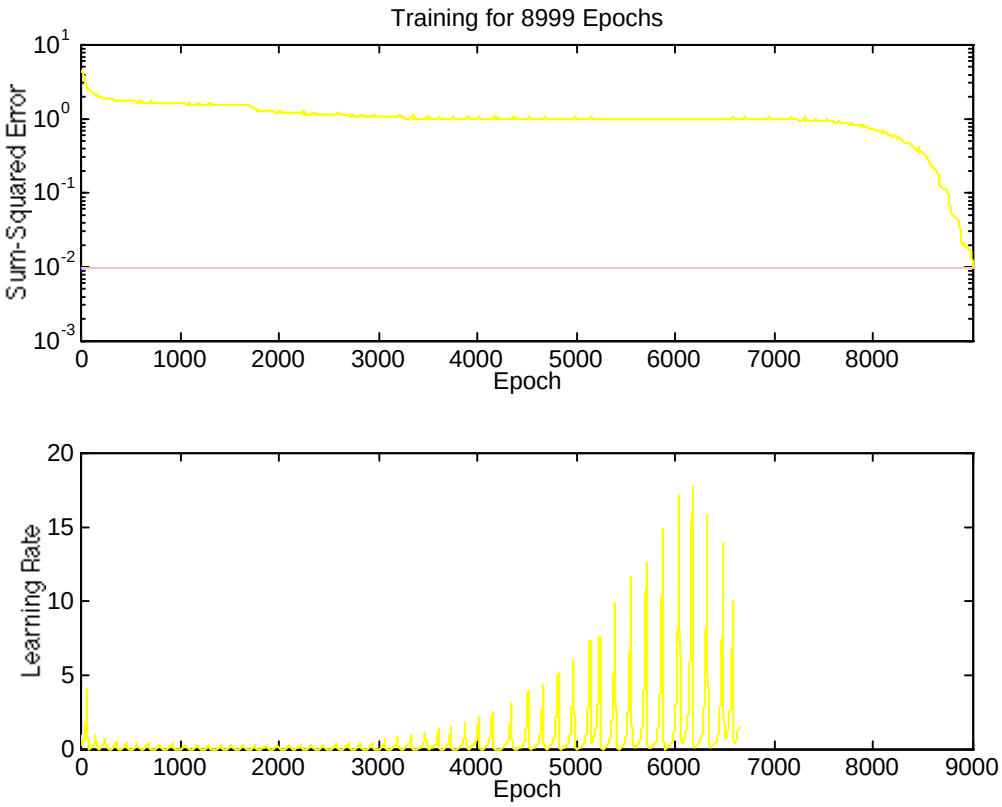


Figure 3–7. Set with a “hole” successfully classified by the 3-layer MLP.

It was observed that as the network grew, the learned region grew more and more irregular, but stubbornly refused to create the desired shape. Finally, at 20 neurons in both layers, and 10 points in the not-A region, it generated a region with a "hole" in it.



3.2.2 Conclusions

Since there are no hard and fast rules to use when designing a net, it is helpful to have some "rules of thumb", especially for those inexperienced at working with perceptron networks. It is important to understand their abilities and limits; certain applications tend to lend themselves to the use of this type of network. In general, any application which can be modeled as partitioning some hyperspace into a number of selection regions can usually be applied to feedforward networks.

3.2.2.1 Partitioning of n-space

If we look at the number of inputs to the entire network as the number of dimensions in a space, we can think about the network as partitioning the space into k regions, where k is the number of neurons in the output layer. Then, we can look at the training of the net as slicing the hyperspace into *k* selection regions. Note that the *k* regions need not be disjoint.

A consequence of this model is that the number of neurons in each layer is directly related to the complexity of the selection region in the hyperspace. Since each additional neuron in a layer adds a simultaneous linear partition, the number of neurons required in each layer is directly proportional to the number of hyperplanes required to partition the space. Therefore, the number of neurons in each layer can be thought of as the *resolution* of the layer. Fewer neurons in a layer gives the layer a rougher shape; more neurons give it a finer one.

Furthermore, there is a finite limit on the ability of the net to learn differentiation between points in disjoint selection regions, given that the points are a certain distance apart. As the Euclidean distance *x* decreases, the training algorithm is limited by the slope of the nonlinearity used in each neuron. In other words, if training points in different regions are too close together, the net cannot possibly converge because the nonlinearity is not sharp enough. This difficulty can be thought of as trying to make a cut between two points on a piece of wood with a blade that is wider than the distance between the two points. Fortunately, the nonlinearity is arbitrarily adjustable to fit the criterion required; however, this quality must be taken into account if the network refuses to converge.

3.2.2.2 Sensitivity of n^{th} layer to $n\text{-}1^{th}$ layer underspecification

Another conclusion was that due to the feedforward nature of the interconnections involved in the network, problems created by not having enough neurons in a given layer *cannot* be rectified by adding more neurons in a later layer. Each layer adds increasing detail to the selection regions. Lower layers are analogous to coarse detail, and upper layers to fine detail. This analogue leads to one important conclusion. As we saw in test set 1, if lower layers are

underspecified, increasing specification in higher layers will have no effect on the net. Therefore, we have to ensure that the lower layers have sufficient neurons to learn to create regions which will contain the correct points.

A reasonable method to use when training a net is therefore to make all the layers the same size, and increase until the net does converge. Then, the number of neurons in each layer can be decreased starting with the lower layers, until there are just enough neurons for the net to converge. This process can then be repeated for each layer.

When working with dynamic training data, however, this process is overly time consuming. It is simpler, and usually as effective to perform this process once or twice to get a general idea of the size needed to make the net converge, then increasing each layer by a few neurons, to allow for differences from data set to set.

3.2.2.3 Tendency to find ‘simplest’ solution results in sometimes non-useful heuristics

Looking at the results from test set 4, we can see that in forcing the network to converge to a specific shape, generally it is difficult to produce a desired shape. Since the network algorithm learns by specifying linear regions, it has a distinct bias towards creating selection regions which are simple linear regions. To create more complex regions, the algorithm has to combine two or more layers.

As a result, the target set must be thorough enough to ensure that the correct heuristic is learned. Otherwise, the network will learn only to classify the specific test set given, and not the desired heuristic. This problem can be avoided by providing a training set which is a representative cross section of the points the network will have to classify in action.

Neural networks are good at interpolation, but bad at extrapolation; this fact suggests that we should provide extrema in the target set, plus points to suggest the general heuristic to the net. It should then be able to develop its own heuristic for other input patterns.

3.3 Speaker Differentiation

3.3.1 General Idea

Since the perceptron exploration led to the conclusion that the primary strength of the perceptron network is classification, it was decided to implement a nontrivial task to test the limits of the heuristic-forming ability of the net. The task chosen was *speaker differentiation*. This task is quite different from speech recognition. Speech recognition deals with the translation of speech into text or recognizable commands; speaker differentiation deals with the identification of one particular voice out of many samples. Or, given one sample, identify the speaker associated with it.

This task has definite practical uses. It can serve as the basis for an authentication system which is almost foolproof. It can be trained to respond to the user's voice and no other. As such, it can serve as, say, an electronic locking mechanism, authentication for critical transactions, or as a key for encryption of sensitive data.

Since the feedforward ANN was shown to excel at classification, it was hoped that it would prove ready to the task of classifying the speech patterns it was presented with.

3.3.2 Approaches

We have shown that to train this type of network correctly requires a suitable training set. For this application, its was theorized that to model the human voice correctly and sufficiently, as full a sample of each voice as possible would be desirable. However, since the neural structure places restrictions on the type of input presented, it became obvious that some standard input format was needed.

To accomplish this goal, a brief foray into phonetics was necessary. North American English speech was chosen as the target language, but the system should work for any language with similar sounds. English, as a spoken language, has about 46 phonemes. Any word in English can be broken into some combination of these 46 sounds. To give input samples some consistency, and to provide some measure of control over the process, it was decided that an exemplar sentence could be constructed which contained a fair percentage of these 46 sounds. Then, the training data could consist of a number of samples of different people speaking the same sentence aloud.

The exemplar sentence decided upon was: "Clairvoyant pedestrians ambulate knowingly." It was chosen because it contained 23 phonemes, about half the number in the entire language. Also, since the sentence did make sense, in a manner of speaking, it could be spoken in a normal speaking voice, with less stress or unusual inflection than could a random arrangement

of unrelated words. This would help ensure that future repetitions of the same phrase would be similar in tone and volume.

More than thirty samples of this phrase were taken, from fourteen different people. The samples were taken at 44.1 kHz at 16 bits per sample, then resampled down to 8 bits per sample at the same frequency. This allowed for easier manipulation of the data with MATLAB, and retained much of the higher quality of the 16 bit sampling than would simple 8 bit sampling. Each sample was roughly 100 kBytes in size, or about a hundred thousand samples each. Since it is less than convenient to have a hundred thousand inputs for the neural net, besides the need for a consistent size input vector, it was decided that the time series of samples should be preprocessed before being presented to the net.

The preprocessing involved a three step process. First, the Fast Fourier Transform was applied to the time series. This removed the dependence of speech speed on the system. If a time-based input was applied to the network, it may be affected by how fast or slow the speaker spoke the exemplar sentence. Since the FFT generates complex numbers which are difficult for the ANN to deal with, the next step involved taking the power spectral density, or PSD. The PSD generates a series the same size as the FFT, but as entirely real values. However, we still have a signal which is roughly a hundred thousands samples in size; therefore, the last step involved a substantial downsampling of the signal. This downsampling preserved the general shape of the PSD curve while providing a reasonably-sized input vector for the ANN.

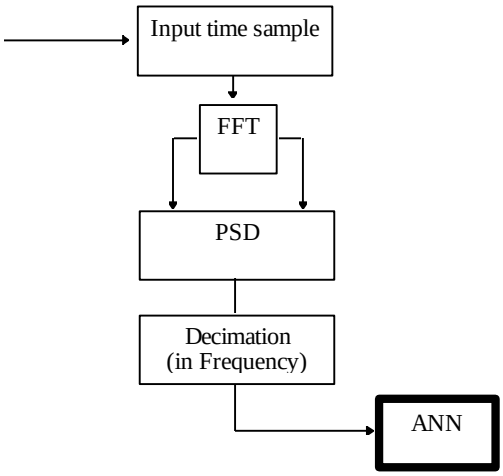


Figure 3–9. Block diagram of speaker differentiation preprocessing to generate input vectors for the ANN processor.

Once the preprocessing step was completed, the input samples were converted into “signature” target vectors, each representing one speaker sample. The target vectors were all resampled to the same number of points, usually either 24 or 96 (differing values were tested), with the most common being 24 points. When these 24-point vectors were displayed, they displayed striking individual characteristics; signatures from the same person were remarkably similar, and those from different speakers were visibly dissimilar, even to the naked eye. The trick was then to train the ANN to recognize the differences.

The main test involved creating a three-layer network with one neuron at the output, much like the networks discussed in the perceptron exploration. The training set was then designed with the aforementioned signature vectors, using all the samples from one specific speaker as those in “set A” or the desired selection region, and all the others in the not-A set. The net was then trained to mastery on those samples. Then, a number of sample test vectors *not* in the training set were presented to the trained network, and the network attempted to determine if the speaker of each test vector was the same as the speaker which the network was trained to identify.

A second test was conducted which involved the same procedure, but with a slightly different method of constructing the signature vectors. Here, a periodogram approach to the input samples was used by chopping the input into a number of vectors, then applying the FFT-PSD-Downsampling procedure on them, then appending each of the resultant vectors to get back a single signature vector. Varying numbers of divisions were tested, as well as varying the downsampling rate. The best results seemed to be accomplished using 8 periods of the original wave, then downsampling each to 16 points. This process resulted in a 96-point input vector.

3.3.3 Conclusions

Recalling from the discussion of the perceptron exploration, we determined that the primary ability of feedforward ANNs was to partition n-space with p partitions, where *n* was determined by the size of the input vector and *p* was determined by the number of neurons in the output layer. Here we are attempting to divide (at least) 24-dimensional space with one partition into exactly 2 regions — those determined by the signatures of one particular speaker and those determined by the signatures of any other speaker. Since it is obviously impossible to visualize 24-dimensional space, we must rely on the techniques developed for the two- and three-dimensional cases for model verification.

Using the methods described, varying degrees of success was achieved; however, none of the trials resulted in a foolproof network. The main training procedure was repeated a number of times, and would converge only occasionally. Due to the incredible amount of processing time

required for each trial run (up to 20 hours on an SGI R4400 Indy) it was not possible to test every combination of periodogram size and input vector downsampling.

It is possible to make generalizations regarding the performance of the networks which did successfully converge. In general, they would correctly identify about half of the “chosen” speaker as being correct, but would *never* misidentify an “incorrect” speaker as the chosen. This behavior is reasonably acceptable from a verification and authentication system. An incorrect match should never be allowed, but a correct match may be denied and prompted for a second or third trial. This implementation is typical of current authentication systems in place today e.g. ATM PIN code verifications and UNIX and other computer system login prompts.

A practical implementation of this system can be designed as follows. Take samples from all persons who require access. Then create one custom neural net for each such person, using that person as the “chosen” one for his/her specific net, and all the other samples as the rest of the training set. Once these networks are designed, the resulting authentication set will be a collection of neural nets, each consisting of three real-valued matrices, and three real-valued vectors. Each set of network data is exactly the same size, in terms of storage space.

Application at the point of authentication would involve a three-step procedure. First, the user is prompted for his/her user name. Once this name is presented, the system retrieves the correct neural net parameters from persistent storage, and the net is generated. The user then speaks the exemplar sentence into a microphone, where it is then sampled and preprocessed, in hardware or software. The resulting signature is then presented to the neural net, which decides if the signature matches the training data.

Updating the system for new users can take the form of either one of two options. The simpler option is for a new user create his/her own network using the all other input samples as “not-A” targets. This however does not ensure that another person in the user set will not be able to be authenticated by the new user, acting maliciously. A more secure, but admittedly much more time-consuming method would be to regenerate *all* the users’ network parameters while adding the new user samples to the training set.

3.4 Temperon

While the previous work performed was both informative as well as practical, it provided little insight into the field of artificial life. It was, however, determined that perceptrons and perceptron-based networks were an insufficient model on which to base a system which was to learn in an unsupervised manner, with no specific guiding learning rules. Therefore, it was decided that a new neuron model would have to be developed — one which learned in an entirely different way.

Think of the type of learning which occurs in a newborn animal, or even a newly-conceived animal. Mammals begin as a collection of cells as the fertilized zygote begins to undergo mitosis. Specialized systems of cells begin to form the basis for the nervous system. The brain begins to form from a collection of interconnected nerve cells. Once those cells are in place and functioning, they begin to process input. Granted, there is a limited amount of input in the mother's womb, but there are some sensations: light, heat, sound, and even touch. These basic senses provide input to the infant brain. The infant must then learn to distinguish those senses from one another and learn to process and understand the baffling array of inputs it receives.

The infant does have basic instincts which are genetically programmed into it, but no specific rules which it follows. Gradually, it learns about its environment from interacting with it, and receiving input from it. Anyone who has ever spent time with a small child has seen evidence of this. Children learn about new things by attempting to sense them with all their senses, in an effort to learn its properties. Is the object hard or soft? Rough or smooth? Does it smell? How does it taste? Is it brightly colored? Does it make a noise? Each and every object the infant interacts with provides it input which shapes its thought processes and memories.

Here we attempt to design an artificial lifeform using ANN structures as its processing center, or "brain", which moves about a virtual space, and interacts with an object in that space. It is hoped that it will display the type of behavior normally attributed to infant biological lifeforms; that is, exploration, followed by discovery, followed by a more focused type of exploration, etc. It is of *utmost* importance that the reader understand that this behavior *is NOT* programmed into the simulation anywhere. It would, of course, be possible to design a system which explores its environment and learns from it. Please note that the system described herein *is explicitly NOT designed to perform any such function.* Rather, it is given the capacity to learn, and the capacity to sense, and the capacity to act, and then just let loose. What happens next is entirely determined by its initial conditions and the inputs it receives from its environment.

3.4.1 Evolution of Model inspired by Aplysia

Most ANNs rely on their learning rules as macroscopic constructs to guide their development, whether they are self-organizing or target-set-based. This structure is like the "hand of God", providing an omniscient holistic perspective on the state of the network and the direction in which it will proceed. If we recall from our discussion of biological neuron structures, there is no corresponding macroscopic system for updating the synapse potentials to be found anywhere in biological organism. This updating takes place directly on a cellular level. In fact, each nerve cell is responsible for updating its own synapse connections. To create a neural model more appropriate for the type of learning which a biological organism experiences, each neuron, or at least each neural circuit, must be responsible for its own learning.

If we look at the structures found in *Aplysia* as described above, we can see similarities to those found in the neuron model. However, there is one variable that *Aplysia* responds to which is not taken into account in the model: time. We saw from the habituation and sensitization experiments performed on the snail that there was a direct correlation between the frequency of stimulation and the learning mechanism observed.

Therefore, a new model was developed which encompassed the same performance parameters as the perceptron, but with a temporal-based weight updating mechanism. This model was therefore named a *Temperon*.

It is the hope of the author, that the microstructure of the Temperon does in fact cause the macroscopic behavior of a Temperon-based system to exhibit such behavior. The difference between the system described herein and a hypothetical system explicitly designed to explore and learn from its environment is that the Temperon system *learns to learn*. As the simulated lifeform moves around its environment, in receives input from a number of senses. The design of the Temperon is suited to processing these inputs and reacting to them, as well as learning from them. However, the actions taken by the network in response to a series of inputs are not prespecified; it "decides" what action it wants to take by trying different actions and observing the results — a system we know as "trial and error".

3.4.2 Description of Model

This Temperon model is in many major ways similar to the perceptron model. It has a number of inputs, weights associated with each input, a bias, and a nonlinear transfer function. However, this model keeps track of the system time associated with each stimulation it receives. (A stimulation is defined as a non-zero input.) Therefore, it can compute the inter-stimulation time for each input. When any input receives a certain number of inputs within a

specific time period, the input is marked for weight update. The exact approach to weight updating is subject to debate.

This mechanism, although not exactly identical to that used in *Aplysia*, does provide the neurons a greater proximity to that of biological neurons. Recalling from the discussions of sensitization, a “noxious” stimulus was applied to create the enhancement of a neural connection, where habituation merely resulted from an innocuous stimulus. There is no way to easily simulate the difference between a noxious or innocuous stimulus. Therefore, a slightly different mechanism was applied.

When an input was marked for weight update as described above, the overall output was considered and the connection was either reinforced or suppressed, depending on whether the output was “on” or “off”.

The actual algorithm involved in the interstimulation time is related to a value called the “likelihood of change”. Each input has a likelihood variable associated with it. When the neuron is created, all the likelihood variables are initialized. As the system time progresses, the variables decay exponentially. When a specific input is stimulated, its current likelihood variable is added to its initial value, and this resulting value immediately begins to decay. If a particular input is stimulated often enough, its likelihood variable will exceed a threshold value. When this occurs, it is marked for update as described above.

This Temperon model is purposely vague. It encompasses the ability to update its own weights based on the frequency of stimulation at each input. The exact implementation of the algorithm is left unspecified for future experimentation.

3.4.3 Description of Testbed

As mentioned above, the ANN implementation consisted of a set of hierarchical classes, which culminated in the `Temperon` class. The TurtleMouse environment was used as a “real-time” interactive experience in which the Temperon-based net could be tested. The procedure used to develop the system is described as follows:

The testbed was written entirely in Java as an Applet. As such, it runs inside a “viewer” application. This application creates one main window. The main program instantiates a variable number of Temperons, then fully connects them. The actual number of neurons used here is an HTML parameter, so it is variable at each execution. This format is similar to the Hopfield networks described above. A new window was displayed on the screen, which depicts the internal state of the network. Each Temperon’s weights were displayed as a vertical series

of editable text fields. The current output of each Temperon was displayed above the weight vector.

The network window also contained control buttons to perform various functions on the network, such as: randomize the weights, manually edit any weight, change the current output of any neuron, and start/stop the system clock. Due to the feedback nature of this type of network, it was desirable to have a discrete clock, rather than make use of the CPU internal clock. This allowed for much finer control over testing the feedback connections. Also, a series of commands were added to allow for the loading and saving the weight data, as well as the time series output of the network. This allowed both repeatability of the network initialization, as well as provided a facility to analyze the data generated over time.

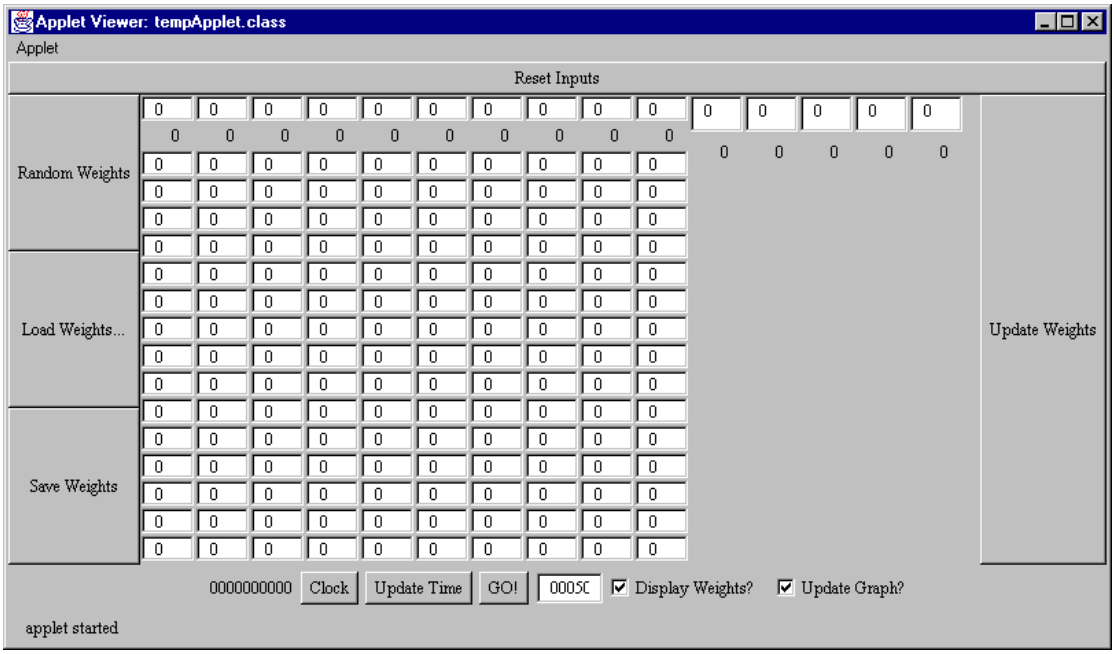


Figure 3–10. Main controlling window in the Temperon testbed applet. Window size (i.e. number of neurons in matrix) is controlled by HTML parameter.

In addition to the series of Temperon outputs, the program also created a series of Input neurons, which were added to the inputs of each of the other neurons in the net. Recall that the `Input` class of neurons has no weights or biases, but are used as a "dumb" input layer. These neurons are used to report the "senses" of the TurtleMouse back to the network. A biological interpretation would associate these neurons with sensory or *afferent* neurons. They are inputs only, and perform no other processing.

Correspondingly, the network required some form of motor neuron connectivity. This was accomplished by taking the first two Temperons created and making special note of their outputs. The following control mechanism was created: since the TurtleMouse has essentially four actions — that is, move forward, turn left, turn right, and stop — a binary code generated by the outputs of two neurons can specify four combinations. When the outputs of those two neurons were [1,1], the TurtleMouse was given the "move forward" command. The outputs of [1,0] and [0,1] generated the "turn left" and "turn right" commands, and the [0,0] output combination stops the mouse from moving.

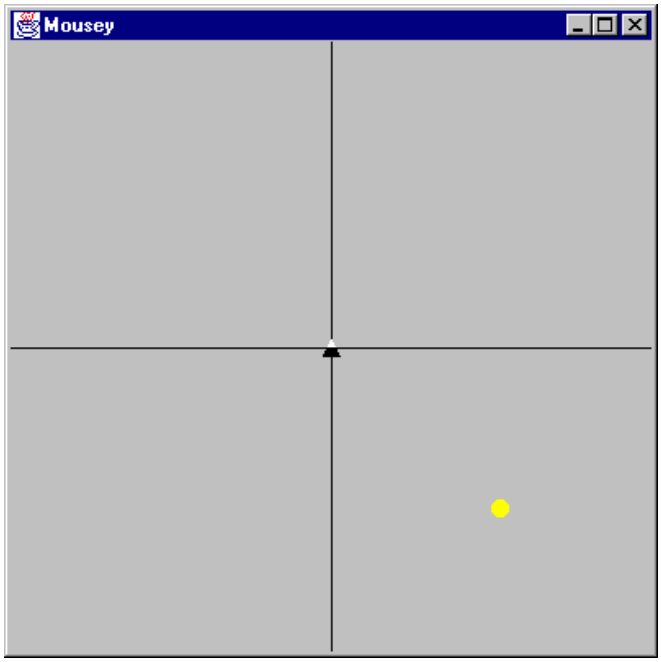


Figure 3–11. TurtleMouse virtual environment. Note triangular "turtle" in center and circular "target" in lower-right quadrant. Turtle has senses which respond to the target.

Once initial parameters such as possible input vectors, obstacle position, and weight initialization were set, the network was generally "started" and left to run indefinitely. The clock repeat rate could be set, and the clock could be started and stopped at any point. As the network oscillates through its various states, the TurtleMouse moves about its environment, controlled by the outputs of the first two neurons; correspondingly, as the mouse moves about, the states of the "sensory" neurons change, feeding back information about the environment into the net.

3.4.4 Various test sets — Overview

A number of different variations on this basic theme were explored as follows: a number of slightly different learning rules were implemented; various constraints were placed on the network parameters; randomness was added to the weight adjusting algorithm; the number and types of senses were varied; the position of the obstacle was varied; the number of Temperons in the net was adjusted. Each time one of these variables were adjusted, the simulation was reinitialized and run again. However, due to the large number of variables involved, permuting all possible combinations proved to be impractical.

Recalling from the discussion of the Temperon learning rule, the specification was left quite vague. The reason for this was that different implementations produced widely varying results, and there were no combinations which performed outstandingly "correctly". Under the specifications of this set of tests, *any* actions performed by the TurtleMouse which are not stochastic are "interesting" and a desirable characteristic. Each time the network was initialized with different random values and simulated, considerably different results were obtained, even if the rest of the model remained completely unaltered.

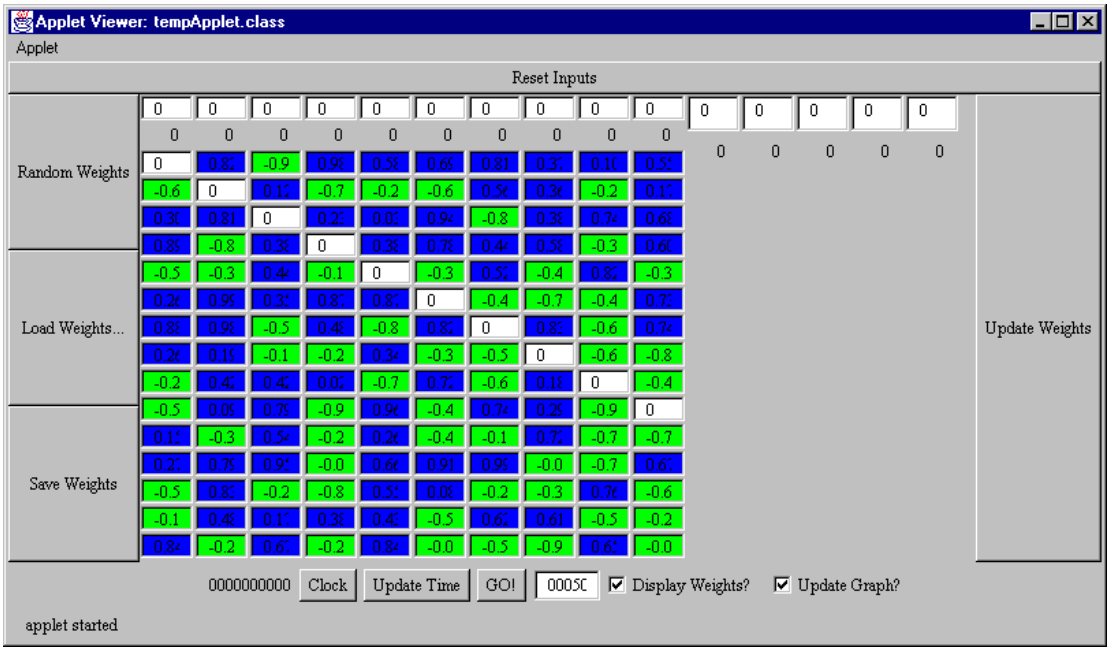


Figure 3–12. Weight set one. Used as starting point for most test sets, as a control. Shading in cells depicts positive or negative changes from previous value. Note zero main diagonal.

3.4.4.1 Test Set 1: Learning rule adjustments

The main factor affecting the behavior of the Temperon is its learning algorithm. Much experimentation was performed with different combinations of learning algorithm parameters.

It was mentioned that a critical aspect of the learning rule is the Temperon's ability to update its own weights. This ability relied on the "likelihood of change" variable discussed above. The algorithm specifies that once an input passes the likelihood threshold, it was marked for updating. Then, at the end of a time cycle, it was updated.

3.4.4.1.1 Reinforce/Suppress pos/neg

The sense of the updating had to do with the current output state of the neuron. The rule specifies that an active output (i.e. a '1') should cause the weight to be reinforced and an inactive output (i.e. '0') should cause the weight to be suppressed. The thought was that if an input was *on* and the consequent output was *on* then the input was contributing to the output and should therefore be encouraged to do so. Contrastingly, if the output was *off*, then the input was actively contributing *against* the state of the neuron and thereby should be suppressed. (Recall that there are no provisions for inactive inputs because the learning rule only comes into play when an input is active; no updating is done for inactive inputs.)

This reasoning, however contrived it may sound, actually proved to work quite well. Unfortunately, when we arbitrarily reverse the logic and reinforce where we would have suppressed and vice versa, the system *still* behaves in an "interesting" fashion. Perhaps a more critical issue is the next one discussed, namely the definition of "suppress" and "reinforce".

3.4.4.1.2 Push up/down vs. towards/away

Recalling that each weight variable is a real number, the definitions of the terms suppress and reinforce become imprecise. The options are easiest thought of on the real number line. A weight falls somewhere on the real line, either positive or negative; what does it mean to reinforce its value? What does it mean to suppress it? We have two options: (1) add/subtract a positive value, thereby pushing the point up or down the line, regardless of its initial position; or (2) add/subtract a value of the same sign as the weight, thereby pushing the point towards or away from zero.

Again, the "correctness" of either procedure is only verified by experimentation. Either can be supported by discussion, depending on if the supporter takes the position that either zero or negative infinity is the "least reinforced" value.

3.4.4.1.3 Max/Min limits on weights

Since the Temperon algorithm calls for the constant shifting of weights, it may occur to the reader that some weights could grow immense in magnitude. Therefore, it was proposed that an arbitrary limit be imposed on weights — a bounds outside of which the weights were not allowed to grow. At some point, the variables involved would overflow, causing the network to fail.

This position may seem at first glance more like implementational laziness (rather than reflecting some quality inherent to the model), but it can actually be justified biologically. Remembering that the weights actually represent attenuation values of the action potential imposed on the postsynaptic terminal, then the further away on the dendrite from the cell body the synapse occurs, the more attenuation exists, and consequently the smaller the weight.

Keeping this process in mind, one can see that there is a minimum constant which the attenuation must adhere to: if the synapse is located *on* the cell body, there is some small amount of attenuation, and no more. It is impossible to be "closer". In fact, the attenuation can be thought of as ranging from unity to zero, where unity represents the optimum connection and zero signifies one which is attenuated to nothingness.

However, when we impose this restriction on the network, we see that often, weights will rise to this bound and remained pinned there, creating a singular or oscillatory condition. Neither solution seems to be a better one.

There is, however, a third possibility, which was conceived very late in the testing stages. Simply put, when any weight exceeds some threshold, it is mathematically reasonable to simply *divide* all the weights by some positive rescaling constant. This has absolutely no effect on the output of the Temperon, but allows the inputs to "grow" indefinitely.

3.4.4.1.4 Random increment

The discussion of adjusting the weights begs the question: what value do we add/subtract to the weights. If we use some small positive constant, we observe an interesting phenomenon: some weights in a row remain matched as they get the exact same value added to them over and over.

We can also reasonably add/subtract a small *random* value to the weight. There is little biological justification for either position. However, it should be noted that any steady introduction of random values into a system makes it nonrepeatable.

3.4.4.1.5 Zero main diagonal

If we look at the fully-connected network of n neurons as a n x n square matrix, the [i,j]th value represents the connection weight from neuron i to neuron j. Therefore, the main diagonal represents the feedback from a neuron to itself. Typically in a Hopfield net, the diagonal is zeroed out, in effect, disconnecting its input from its own output.

Experiments were performed with this constraint in place.

3.4.4.1.6 Symmetric Matrix

Again, drawing from Hopfield net technology, typically a fully-connected network is symmetric about the main diagonal. In other words, the connection weight from neuron i to neuron j is the same as the weight from neuron j to neuron i. No experiments were performed with this constraint; it is simply mentioned for completeness.

3.4.4.1.7 Forced Oscillations

Another characteristic of a biological neural cell is the fact that it has a repolarization period after it fires. In other words, it cannot re-fire immediately after it has just fired. This property means that it is impossible for a neuron to remain "on". As a result, information is carried in the *frequency* of action potentials.

A Hopfield-style network, on the other hand, *can* contain neurons whose output is always "1". It was thought that this dissimilarity would bias the network. Therefore, some experiments included a Temperon with an arbitrary condition which forced the output to zero if it had just been one, regardless of the output computed by its transfer function. This property forces a given neuron to oscillate, instead of being "stuck" at "1".

3.4.4.2 Test Set 2: Number of neurons

Another variable which was adjusted widely was the number of Temperons in the network. A larger number of neurons in a fully-connected network will result in a larger number of states through which the network can cycle. Theoretically, the increasing complexity in the network should allow it to learn more "intelligent" behavior.

3.4.4.3 Test Set 3: Number/Types of senses

Since the sense neurons are arbitrarily defined algorithms, it is simple to add any sense to the TurtleMouse. The original set consisted of three senses: the "180° in front" sense, and the

“180° to the left side” sense, and the “proximity to the dot” sense. It was thought that these three senses should give the net sufficient information to “locate” the dot if necessary.

After a number of experiments with the three senses, it was decided to add a few more. A corresponding “180° to the right side” sense was added. Also, a “30° in the front” sense was added. This sense gives the TurtleMouse the ability to “home in” on the dot.

3.4.4.4 Test Set 4: Obstacle position

The obstacle position was varied while testing. This was a critical test, because it would show if the TurtleMouse was actually responding to the presence of the dot, or if its actions were completely random. Keeping all other parameters constant and moving the position of the dot should result in an entirely different time response, if the turtle is truly reacting to the dot in its environment.

3.4.4.5 Test Set 5: SDIC

In fact, it was hypothesized that since the position of the object was the only input to which the network can respond, even a very small change in its position will result in a different time response. To test this possibility, the network was initialized, and then saved before having been run. It was then run once, and the output recorded. Then a new network was started and the initial parameters used in the first run were loaded. The position of the dot was changed very slightly — on the order of 0.001% — and the network was re-run. The time series outputs were then compared.

3.4.5 Conclusions

3.4.5.1 Overall Behavior

Again, since the output we are looking for is a very subjective type of behavior, it is difficult to quantitatively verify the Temperon model. However, even the simplest test run in the environment can show that the system is performing with some degree of learned behavior.

A typical test run can be characterized in terms of its general motion. Usually, the TurtleMouse will move about randomly, either rotating or shooting straight up, until it “senses” the obstacle. Then its behavior becomes entirely unpredictable. However, it has a tendency to have a short-term transient, when it first senses the dot. It will either make a beeline directly towards the dot, or begin to circle the dot, or be repelled by the dot. This is generally followed by a long-term transient, where it learns a more “advanced” type of behavior. It will then follow that behavior for a longer period of time, usually from five to ten

times as long as the short-term transient. These general transient behavioral characteristics remain relatively consistent regardless of the variables described above.

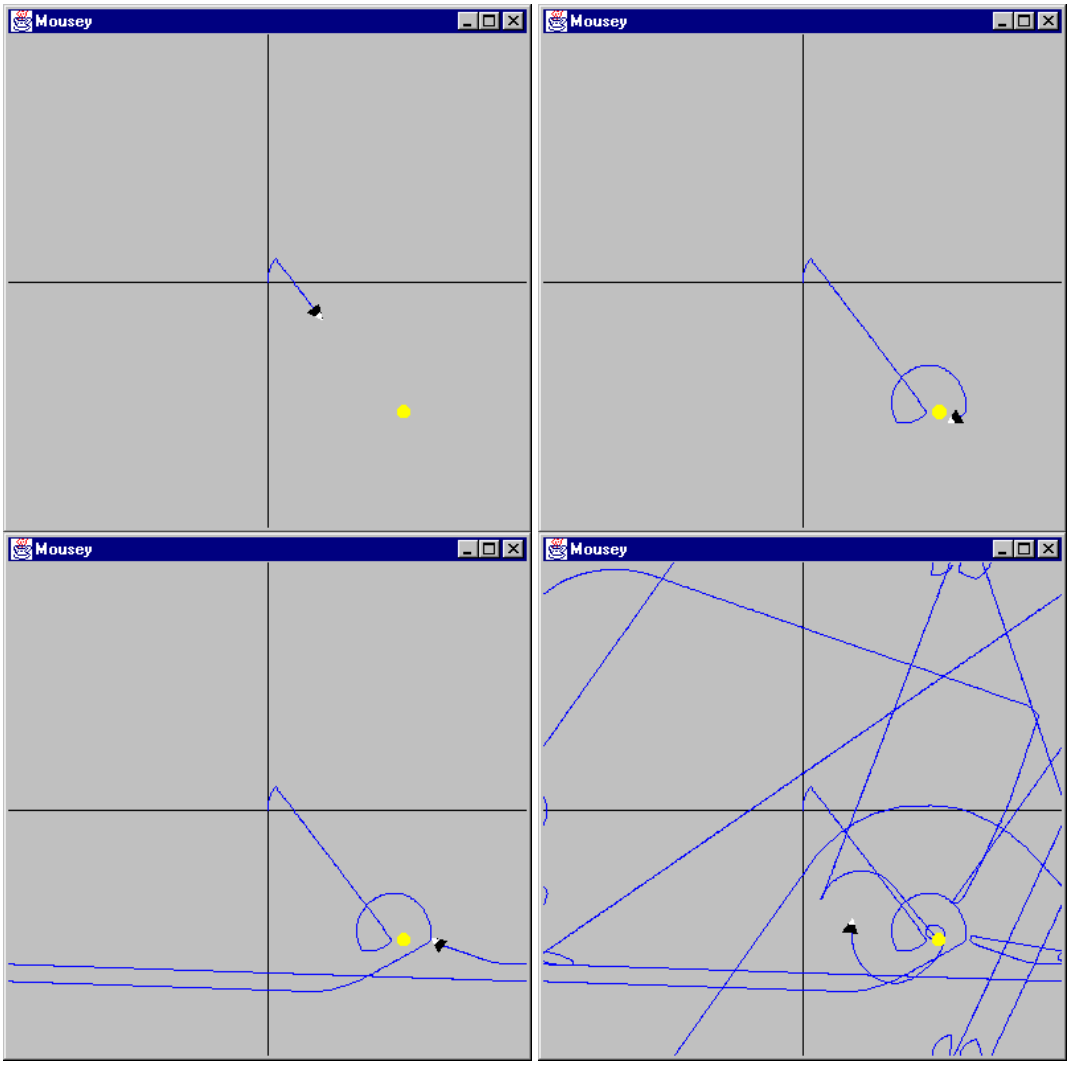


Figure 3–13. Basic execution of TurtleMouse over time [l-r, t-b], using above initialization weights. All options are set to initialization states. "Learning" is characterized by periods of oscillatory states (long, straight lines) separated by "interesting behavior" where the turtle responds to the position and presence of the target..

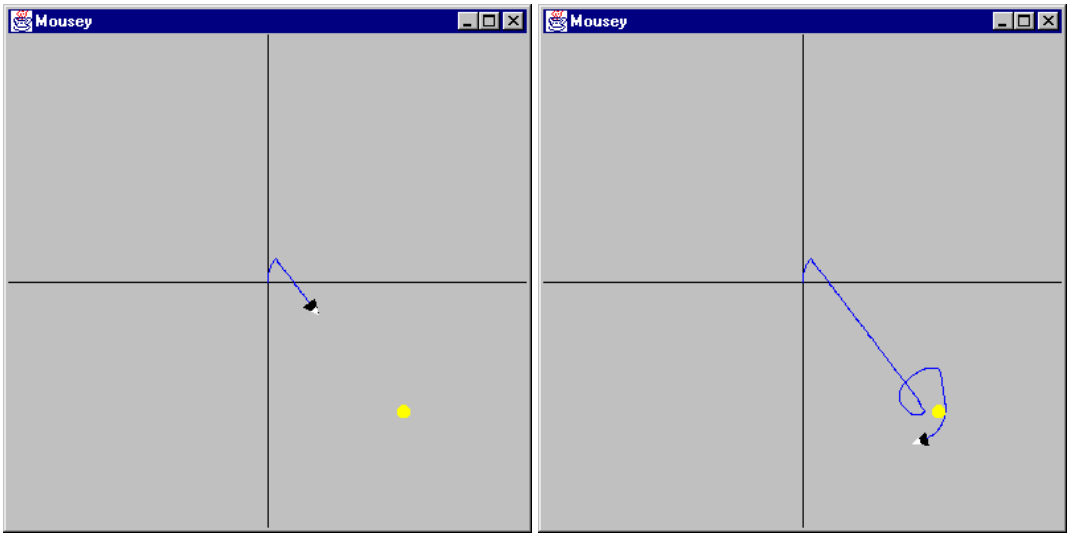
However, once the long-term transient period is over, the different variables have a substantial effect on the performance of the net. Sometimes, the net falls into a steady-state condition where it oscillates between the "all-zeros" and the "all-ones" states.

3.4.5.2 Learning rule changes

Looking at the learning rule variables, it was found that the primary catalyst came from changing the push up/down rule to push-to-zero/away-from-zero. Once this rule was changed, the network started to exhibit the emergent behavior discussed above. However, this behavior was very short-lived.

The weights in the network tended to grow out of hand reasonably quickly. The main diagonal usually grew the fastest. This tended to cause the network to become stuck in the oscillatory condition. These characteristics led to more changes in the learning rule. The main diagonal was then zeroed, and the max/min conditions were added.

These had little effect on the short-term behavior of the TurtleMouse. The max/min condition imposed tended to result in the network growing in size to hit the max (all the weights equal to the max or the min), where it stayed for an indefinite period of time. This has the effect of holding the TurtleMouse in a static repeating pattern. After some time, though, at least a few of the neurons moved away from the max value. When this happened, the mouse broke out of its pattern and went through the short-term/long-term transient behavior described above.



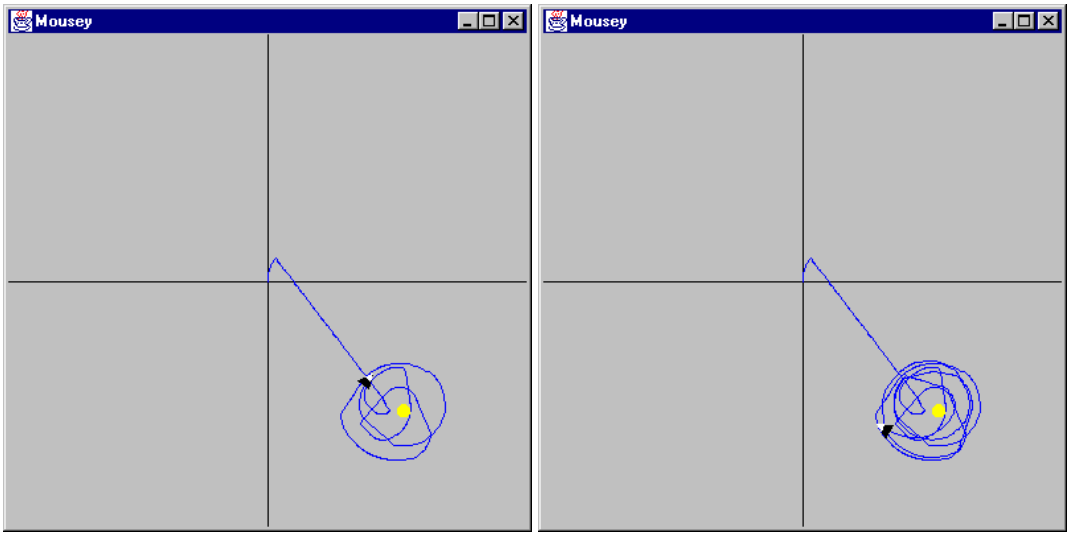


Figure 3–14. Time series of same network as previous example (with identical initial values), with max/min weight condition added. Here we are using a value of |3| as the absolute max for the value of each weight. Note that the early states are similar to the previous execution; however, this set stays very close to the target and does not exhibit the alternating transient behavior seen before.

Adding randomness to the weight updating procedure seemed to have the effect of preventing the net from reaching the oscillatory state for a longer time, if at all. However, it does have the obvious effect of making a test run non-repeatable.

The "forced oscillations" condition seemed to have little effect on the overall performance of the network. However, since the sensory neurons do *not* oscillate, they receive at least a two-to-one bias in terms of their neural weight updating properties.

3.4.5.3 Responses to test sets

The number of neurons in the network seemed to have little effect on its "intelligence". The mouse learned similar behavior regardless of the number of neurons added to the net. However, they were necessary to process additional information, as seen below.

The number of senses had a great deal of influence on both the types of behavior learned, as well as the speed at which it learned them. When the TurtleMouse had only three senses, it learned basic repulsion/attraction characteristics. Some test runs, it would be drawn to the object, making loops around it, and some times, it would turn away from the object, each time it passed in front of it.

However, when two more senses were added, the mouse was capable of more complex movements, such as actually pinpointing the object and stopping on it, or circling it at a fixed

radius. Incidentally, **when more senses were added, in order to obtain this complex behavior, more neurons were required.**

The position of obstacle, although generally set at some arbitrary position in the virtual environment, proved to be a valuable tool in determining if the network was acting randomly or otherwise. When the dot was removed from the environment, the mouse performed as expected: that is, randomly. It moved about in a random fashion, then oscillated to a steady state in a reasonable short period of time. When it was put back, but in a different position, the behavior changed according to the position of the dot. This verified that the TurtleMouse was actually responding to the conditions of its virtual environment.

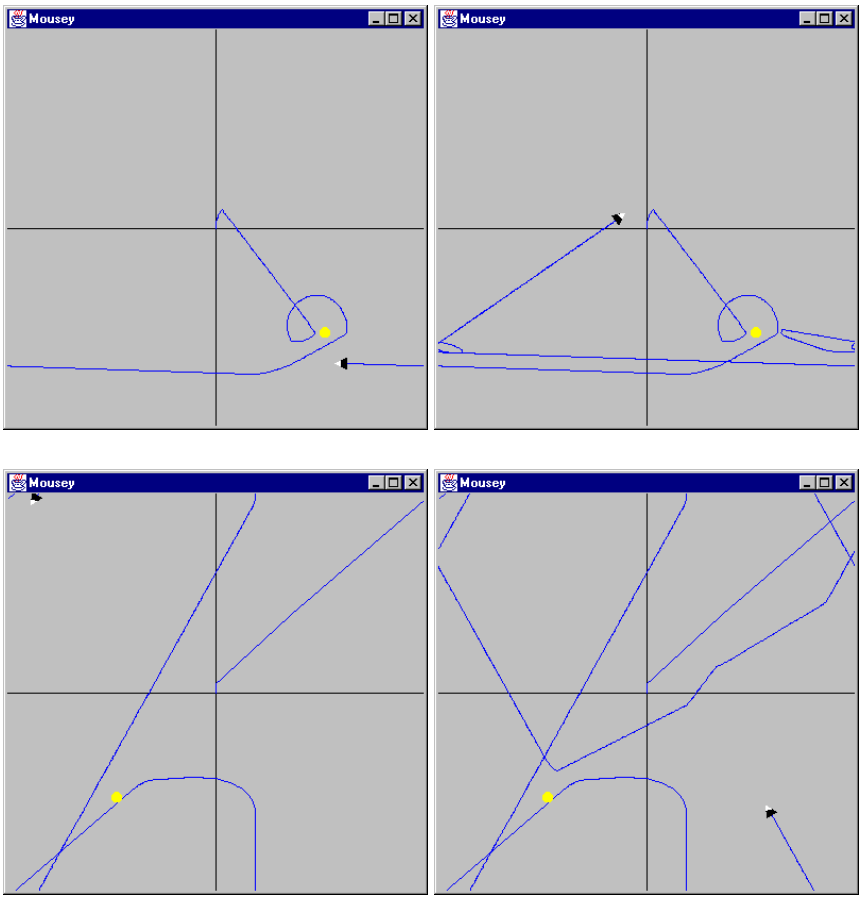


Figure 3–15. Contrast top two pictures with bottom two pictures. Each set represents the state of the network at 1000 and 2000 iterations, from left to right. Both sets were initialized *identically*. Note the difference in the position of the dot.

In fact, when the position of the dot was changed a very small amount, the network responded exactly as it originally did, for a number of time steps. Then, over a very few time steps, it diverged relatively quickly from its previous course, and subsequent actions were entirely different.

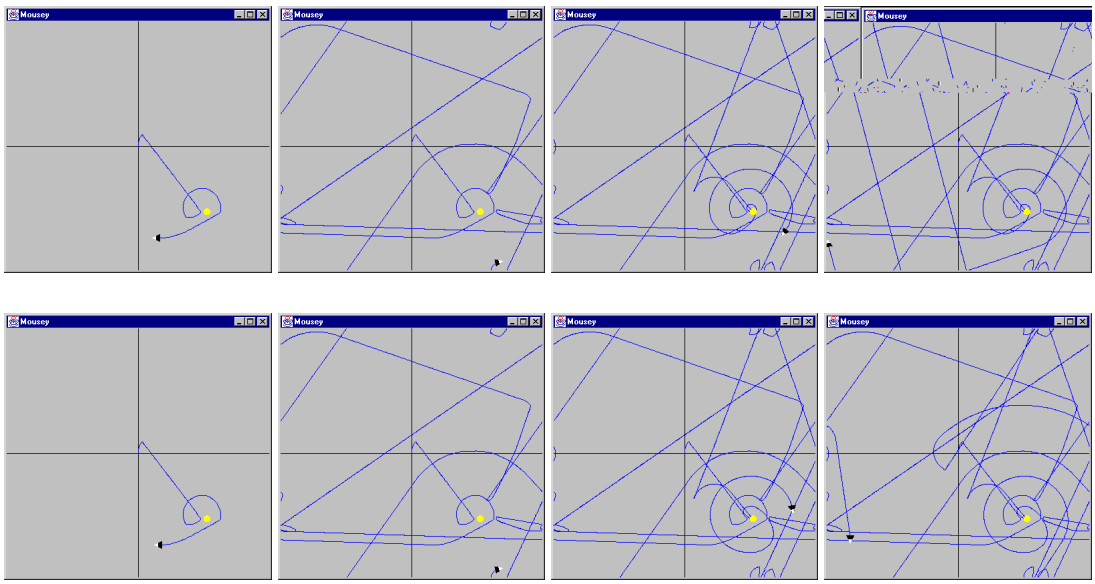


Figure 3–16. Illustration of sensitive dependence on initial conditions. Here the position of the object was changed by 0.001%. Note the variations begin to occur in frame three, which represents 6000 iterations of the network. The last frame (7000 iterations) shows the states to be completely divergent.

It should be mentioned that the **single biggest factor which determined the long-term behavior of the TurtleMouse was the random initialization**. With no other variables being changed, each time the network was reinitialized and re-run, it behaved entirely differently. This raises serious doubts as to whether the random initialization is a reasonable procedure to use when designing new networks.

3.4.5.4 General Conclusions

The Temperon model seems to have captured at least some of the capabilities of its biological inspiration. As the TurtleMouse moves through its virtual space, it evokes feelings of infancy, then childhood as it learns. It is entirely different from other types of Artificial Neural Net structures used to date in that its output is based upon stimulations over time.

It does, however, need much further study. It is beyond the scope of this thesis to provide a finalized version of the Temperon learning rule. It is suggested that although the learning rule

is not firmly established, the neural structure of the Temperon has been validated. It has been shown that the self-updating microstructure of the Temperon can, in fact, learn with no controlling overall learning rule, as well as no inputs or targets. This was a primary goal of the research.

It is also suggested that, in the search for creating an artificially-sentient structure, that is, a machine which can think, the Temperon can be of use. Although it was beyond the scope of the available facilities, it is hypothesized that a large number of these Temperon structures, when provided with enough input and the ability to interact with its environment, will, in fact, produce artificial life.

4. Conclusions

A number of classical ANN tests were attempted and successfully completed using traditional perceptron networks. The XOR problem was solved, and the limits as to the minimum size of the network necessary were tested as well.

In addition, three geometries were also tested successfully: a closed arbitrary area, a disjoint area, and an area with a “hole”. Again, limits were tested as to the size of network required for each of these geometries. The classical perceptron network lived up to theory and was able to classify all three types of geometries. This can be extrapolated to imply that a three-layer MLP can be taught to classify *any* arbitrary region in N-space.

One type of application for the MLP network was designed and tested with limited success. The Speaker Identification system was able to successfully distinguish a number of speakers as being “incorrect” i.e. not the chosen speaker. However, it did only report the “correct” speaker 3 times out of 7. This could be attributed to the fact that the limited computing resources could not train the network to perfection in a reasonable simulation time. For this type of authentication system, this type of performance, though sub-optimal, is acceptable.

The Temperon structure developed was shown to have emergent behavioral characteristics consistent with those attributed to biological systems. A number of simple tests proved that the virtual mouse, using the Temperon network as its processor, was in fact able to learn to adapt and react to its environment. Moreover, it did so with absolutely no overriding control program, or specific instructions to do so.

After extensive testing of perceptron-based neural networks, it was determined that a more flexible type of neuron was needed in order to implement some type of artificial life. The current ANN algorithms all imposed a “big picture” type of processing environment where the learning rule controlled the overall direction in which the network converged.

The aforementioned Temperon algorithm takes an evolutionary approach to the system development. Each neuron in the system is responsible for its own learning, according to very simple rules. Then, as is typical of massively parallel systems, the entire system exhibits complex behavior. Unlike the “expert system” approach, the Temperon-based network is not given a task to perform or a puzzle to solve; it is simply allowed to exist. This approach helps avoid imposing any external prejudices on the system.

We have shown that the Temperon network does in fact learn to respond to its environment. The simulations described utilize the Temperon net as its “brain” while it has certain predefined “senses” and certain predefined “motor functions”. Note that the fact that these functions are predefined does not impose any bias on the system; in fact, the system learns to adapt even when its senses or motor functions are altered.

It is difficult to speculate as to the possible uses of the Temperon technology, other than as an end unto itself. It does, however, have possible applications both in technical and biological fields.

In a technical capacity, a Temperon net can be utilized as the “brain” or central processing unit of an entirely new type of computer. It represents a massively parallelizable architecture which has the ability to learn to perform various tasks assigned to it. It can also be used as a functional unit in traditional processing systems to perform evaluation and judgment-oriented tasks once it has been trained to do so.

In addition, studying the Temperon network may help give us additional insight about the biological learning process. Since the Temperon algorithm attempts to mimic the nature of actual biological systems, we may be able to learn from observing its behavior in certain situations. A network complex enough to encapsulate human behaviors could help psychiatrists and neurobiologists diagnose human patients. But all this relies on much more development of a technology which is in its infancy.

The field of Artificial Intelligence, being a relatively young and unexplored field of exploration, is a ripe breeding ground for speculation as to the direction future development should face. Many researchers currently are utilizing A/I technology and techniques to provide a functional block in an otherwise traditional system. This type of “black-box” integration virtually dominates development resources because it can provide results either difficult or impossible to achieve with traditional processing units. However, it is difficult to believe that any of these systems, although perhaps useful, embody any actual “intelligence”.

Even the connotation of the term “intelligence” is hotly debated. What does it mean for a system to be intelligent, or to display intelligent behavior? For years, the defacto standard of intelligence was derived from Noam Chomsky’s work in language development as the ability to communicate using a natural language. However, recent advances have seen systems with exactly that ability — to communicate in and react to a natural language — which are nothing more than complex processing algorithms, really just large computer programs.

Terry Winograd's "SHRDLU" is one example. His program can both react to commands by moving groups of virtual blocks around in a space and respond to queries regarding the relative positions and attributes of blocks. However, it is generally stipulated that such programs are *not* intelligent, but rather an intricate combination of algorithms designed to respond to a infinitesimal subset of language statements. So a different test of intelligence, one not based on language manipulation, is needed.

Or is it needed at all? Only a very small subset of development is being performed with the end of developing a system which exhibits characteristics similar to those of living organisms. The reason such a small amount of work is being done in this area is most likely because there are no obvious profits involved; since most research is driven by funds, the lack of available funds tends to curtail exploration.

Another possible detractor may be an innate fear of what such development would imply. The entertainment industry, in particular science-fiction, has continuously thrived on plots involving so-called "self-aware" machines. They are now so ubiquitous in movies and books that generally no surprise is elicited at the fiction of the presence of machines which interact with humans as peers. However, equally omnipresent are plots which involve the failure of these machines and their descent into psychosis and even homicide or genocide. Witness one of the earliest and arguably the classic of sci-fi thrillers, "2001, A Space Oddessy" which sees a spaceship's main computer, HAL, go completely mad. More recently, the plot of the "Terminator" series begins when the defense system becomes "self-aware" and decides the human race should be exterminated.

This type of media bombardment, while it creates fantastic box-office returns, cannot help but create a feeling of disquiet in the general populous when they are presented with a system with the ability to learn for itself and make decisions based on its own learned knowledge. Even the thought of a very primitive system, such as insect-colony behavior, tends to create uneasiness in observers.

However, within the small community of researchers working on artificial life, opinion is divided into two main camps with regards to the development of a true "artificial life". The first group advocate so-called "expert systems" i.e. systems whose "understanding" of and reactions to their world is based on a set of rules or relationship pre-entered into their database. They then rely on these rules to acquire information and develop more rules and so on.

The second camp of researchers is dedicated to the notion that any initial rules imposed on the system would bias it Their approach is to simply provide a learning mechanism and allow the

system to acquire information. Then, the test is to see if the system organizes itself and produces a nonrandom output. This is the approach described in this study.

The Temperon represents a small shift in the traditional neural network paradigm. However, it is thought that this development brings us one step closer to the goal of creating a truly intelligent lifeform.

5. Future Considerations

This research has demonstrated to us a principle we see time and time again, perhaps unknowingly: the combination of many simple systems can produce very complex and unpredictable behavior. Take, for instance, the flocking of birds. They coordinate their changes in direction with no apparent leader. Schools of fish also demonstrate this behavior. Why, then, do groups of humans have such a hard time getting along with one another? Perhaps we are too complicated.

The research presented here seems to suggest the success of the Temperon-based system is due to the overwhelming simplicity of the individual elements. Neurobiologists have concluded that most cortical (brain) neurons are reasonably simplistic, yet essentially identical in function. Development in high-level artificial intelligence has failed to produce a system which can be shown to demonstrate emergent behavioral traits. These two facts taken together may be indicative that the direction new development must follow is in massively parallel systems of highly simple devices, rather than a small number of overwhelming complex devices.

It has been hypothesized by classical A/I researchers that in order for a computing device to approach human intelligence, the number of interconnections in the device must approach the order of magnitude of the number of interconnections in the human brain. Moore's law, which states that the density of elements on integrated circuits doubles every eighteen months, already predicts the not-too-distant future when ICs will be able to meet and surpass that classical boundary, at least in sheer quantity of elements, anyway. However, we suggest here that no matter how massive the system becomes, it is always at the mercy of the its controlling process. Similarly, a group of workers under the strict control of their supervisor can only perform at most to the level of the supervisor's imagination, and not exceed it. It is only when one of the workers is able to step outside the controlling envelope of the foreman's direction that change and growth are truly able to occur.

We therefore postulate two additional constraints on the above statement: that the individual elements must (1) each have the ability to perform some amount of processing on its own, no matter how simple; and (2) that there exists *no* overriding control program which governs the actions of individual elements.This being said, it remains to future researchers to realize the potential of the Temperc

system, and more importantly, the characteristics noted in the systems described herein. Due to limited computing resources, we were able to simulate systems of less than thirty fully-

connected neurons. Also, it should be reemphasized that the neural behavior here was *synchronous,* i.e. the output of each neuron was held in a buffer until requested, rather than propagating in a data-flow model, which is more accurate compared to biological systems.

A more powerful computing resource should be able to take the simple structure of the Temperon algorithm and replicate it thousands or millions of times, allowing for the massive amount of parallelism demanded from these systems. Also, event-driven programming would allow the firing of each neuron to automatically stimulate its succeeding neuron. It is suggested that a hardware-description language like VHDL could be utilized to produce the desired results, because it has both the installed base of hardware necessary to simulate such a large system, and because it is designed to support processes which are triggered by certain events.

It is important to note here that one of the critical steps in demonstrating the functionality of the Temperon system was in defining its “senses” and “motions” i.e. its input and output processes. It is much more crucial in a larger system to provide many means of expression as well as many sensing processes. The TurtleMouse simulation demonstrated quite clearly that it only began to learn when it encountered the stimulation of an external object. Then, it learned by moving around and observing the changes it could “see”. As we have already mentioned, this is not unlike the way any living system, including a human child, interacts with the world.

A simulation can only go so far to illustrate the viability of this type of system, no matter how powerful the computing resources, simply due to the nature of the modern computer OS. In fact, the term “operating system” is exactly antithetical to the parameters the Temperon system is based on, namely a centralized overwhelmingly controlling process. In order for these types of systems to progress, they need autonomy.

Therefore, the second and most obvious path down which this research must proceed is purely hardware-oriented. The challenge is to design a circuit which can display small-scale behavior similar to the functional characteristics of the Temperon, and provide for some way to dynamically create and destroy interconnections among large numbers of those functional units.

This system could then be connected to a number of input and output devices, which it should have individual control over, rather than defining discrete control systems to perform specific tasks. In other words, each individual component should be attached to the Temperon net.

However, even when the system is designed and fabricated, and the input systems and output systems are connected, and even when it is initialized and started up—even then—do not expect any intelligent action from the system. We have shown here that it takes a certain

amount of learning time before a system does anything interesting. We have also shown that that amount of time increases with the number of neurons in the system. Therefore, given a system on the order of complexity of the human brain (billions of neurons each with thousands of interconnections) expect a significant period of seeming dormancy, then a long period of apparent random behavior before anything even remotely resembling intelligent behavior begins to emerge. Remember, even a human infant does little more than cry and wave its limbs during its first few weeks of life. And it had the advantage of nine months in which to acquire data about itself and its environment during which all its needs were accommodated.

This research takes the attitude that true life, exists as a combination of both features — the predestined (or inherited) characteristics, working in conjunction with the learned (or acquired) ones. We can see that in the Temperon network, the initial conditions imposed on the network proved to be the major factor influencing the path of development as the system progressed. It was then hypothesized that another possible avenue of development is to progress in the direction of *network growth* i.e. starting out with a small, minimally initialized network which has the ability to add neurons dynamically to its net.

Network growth represents the next phase in ANN and Temperon development. Look to the biological systems these simulations are based on. They all start from the fertilized zygote which develops into a functioning organism, while learning all the while. Nowhere in nature do we see a system simply pop into existence, fully functional physically but with zero mental capacity.

Of course, the next logical argument is that the DNA structure found in most terrestrial life directs an organism's growth and learning. The position taken by this researcher is that the pattern described in DNA is merely a blueprint as to how to create a structure and that the structure does not totally determine an organism's behavior. This attitude is a reflection of the ANN ideology that the topology of the network does not uniquely determine its behavior, but does influence its ability to learn.

So we can see that there are two variables to take into account: the weights and biases found in the network (along with their initial values), as well as the actual interconnection topology. In this study, we focused primarily on using a fully-connected network of varying size, while designing the neurons to adjust their interconnection weights.

6. Appendices

6.1 Appendix A: MATLAB code

6.1.1 Perceptron exploration

6.1.1.1 HINTONEM.M

```
figure

subplot(3,1,1)
hintonwb(w1,b1)


subplot(3,1,2)
hintonwb(w2,b2)
subplot(3,1,3)
hintonwb(w3,b3)

orient tall
```

6.1.1.2 PLOTEM.M

```
axis([0 10 0 10])
axis(axis)
hold on

imshow([0,10],[0 10],outmat,256);

plot(inA(:,1),inA(:,2),'ro')
plot(notinA(:,1),notinA(:,2),'bx')


%plot(0,0,'yo')
%plot(10,10,'yo')
```

6.1.1.3 SET1.M

```
% matlab script file for setting up variables
% for a classification test.

% we are attempting to make matlab
% correctly classify points in 2-d
% according to a binary (A-or-not-A)
% set.

% The experiment is to test the sensitivity
% of an ANN to various numbers of neurons in
% each of the three layers

% why 3 layers?
% First layer creates lines -> linearly separable sets only
% second layer creates convex hulls -> will not deal with disjoint or
concave sets
% Third layer creates arbitraty shapes -> can classify according to general
shapes
```

```
% here, we create the test set explicitly.
% we are working within a cartesian grid x=[0,10];y=[0,10]
% Let's create some points, the first half of which
% are 'in' A and the second half aren't;

clear

inA = [
2,1
3,4
8,4
6,2
7,9
];

notinA = [
4,8
2,6
7,1
9,9
5,5
];

A = [inA;notinA]';

targets = [ones(size(inA,1),1);zeros(size(notinA,1),1)]';

% let's look at these puppies...
%clg
%axis([0 10 0 10])
%hold on

%plot(inA(:,1),inA(:,2),'yo')
%plot(notinA(:,1),notinA(:,2),'bx')

% set up the net...
%initialize 1st and 2nd layer weights

inputs = 2;% this is fixed -- 2 coordinates
size1 = 3;
size2 = 50;
size3 = 1; % this is fixed, because we only want a binary output

w1 = rands(size1,inputs);

b1 = rands(size1,1);

w2 = rands(size2,size1);

b2 = rands(size2,1);

w3 = rands(size3,size2);

b3 = rands(size3,1);

figure(1);clg;
subplot(2,1,1);
outmat=testNet(w1,b1,w2,b2,w3,b3,.25);
plotem;
str = sprintf('%d in layer 1, %d in layer 2',size1,size2);
title(str);
```

```
xlabel('Before training')
%pause
figure(2);
[w1,b1,w2,b2,w3,b3,TE] =
trainbpx(w1,b1,'logsig',w2,b2,'logsig',w3,b3,'logsig',A,targets,[100,50000,
1e-2,.9])

[o1,o2,o3] = simuff(A,w1,b1,'logsig',w2,b2,'logsig',w3,b3,'logsig');

A

o3

outmat=testNet(w1,b1,w2,b2,w3,b3,.25);

figure(1);
subplot(2,1,2);
plotem;
title(str);
lbl = sprintf('After %d training epochs.',TE);
xlabel(lbl);
orient('tall')
```

6.1.1.4 SET6.M

```
% matlab script file for setting up variables
% for a classification test.

% OK. now you've done it. Here is a 16-point 1D dct.
% so there.


% the inputs are going to come from a random set of 1D DCTs.
% there are numSets different sets of input/target combos

numSets = 100;

npoints = 16;

A=10*rands(npoints,numSets);

targets = dct(A);


% set up the net...
if exist('init'),
   if (init==-1),
      %initialize 1st and 2nd layer weights
      clear init;
      inputs = npoints;% this is fixed
      size1 = 25;
      size2 = 25;
      size3 = npoints; % this is fixed

      w1 = rands(size1,inputs);

      b1 = rands(size1,1);

      w2 = rands(size2,size1);
```

```
       b2 = rands(size2,1);
       w3 = rands(size3,size2);

       b3 = rands(size3,1);
    end
end

%figure(1);clg;
%subplot(2,1,1);
%outmat=testNet(w1,b1,w2,b2,w3,b3,.25);
%plotem;
%str = sprintf('%d in layer 1, %d in layer 2',size1,size2);
%title(str);
%xlabel('Before training')
%pause
%figure(2);
[w1,b1,w2,b2,w3,b3,TE] =
trainbpx(w1,b1,'logsig',w2,b2,'logsig',w3,b3,'purelin',A,targets,[100,10000
,1e-2]);

%[o1,o2,o3] = simuff(A,w1,b1,'logsig',w2,b2,'logsig',w3,b3,'purelin');


%outmat=testNet(w1,b1,w2,b2,w3,b3,.25);

%figure(1);
%subplot(2,1,2);
%plotem;
%title(str);
%lbl = sprintf('After %d training epochs.',TE);
%xlabel(lbl);
%orient('tall')

% if it sufficiently trained, let's test it.
clear testcol;
testcol = 10*rands(npoints,1);

[o1,o2,o3] = simuff(testcol,w1,b1,'logsig',w2,b2,'logsig',w3,b3,'purelin');

testcol
[testcol o3 (testcol-o3)]
```

6.1.1.5 TESTNET.M

```
% function outmat=testNet(increment);
%
% this file generates a full test of the 10x10 space and
% plots the outputs as a greyscale image
% it should help visualize exactly what region the
% net has solved for


% generate the test vector.
% essentially, we need all 2-d coordinates within the space
% with a certain increment.

function outmat=testNet(w1,b1,w2,b2,w3,b3,increment);

if (~exist('increment')),
```

```
   increment = 1;
end

A = [0,0];

for outloop = 0:increment:10,
   for inloop = 0:increment:10,
      A = [A;outloop,inloop];
   end
end

A=A';

% remove the pesky 1st element;



[o1,o2,o3] = simuff(A,w1,b1,'logsig',w2,b2,'logsig',w3,b3,'logsig');

len = length(o3)-1;

% get rid of the 1st element
o3 = o3(2:len+1);

side = sqrt(len);

outmat = reshape(o3,side,side);
% the flip corrects for the display order
```

6.1.2 Speaker Differentiation

6.1.2.1 HAMDIST.M

```
% [diff] = hamDist(v1,v2);
%
% Returns a vector which represents the bitwise Hamming distance
% of each of the elements in v1 to the corresponding element in v2
% If v1 and v2 are not integral, it rounds them towards zero.
%
% i.e. hamDist(4,5) = 1,
% because 4 = '100' and 5 = '101' (in binary)
% which differ in 1 position (the 2^0 place)
%
% Note: hamming distance is commutative, i.e. hamDist(A,B) = hamDist(B,A)


function [diff] = hamDist(v1,v2);

% fix the vectors so we have integral column vectors.
v1 = fix(v1(:));
v2 = fix(v2(:));

if (length(v1) > length(v2))
   minlen = length(v2);
else
   minlen = length(v1);
end

v1 = v1(1:minlen);
v2 = v2(1:minlen);

highpow = nextpow2(max(max(v1,v2)));

% now that that's done, we can loop thru each of the vectors
diff = zeros(size(v1)); % initialize the output vector (just in case)

for count = 1:minlen,
   count;
   t1 = v1(count);
   t2 = v2(count);
   for bits = highpow:-1:0,
      m = 2^bits;
      logical = ((t1>=m)&(t2<m)) | ((t1<m)&(t2>=m));
      if (logical)
         diff(count) = diff(count) + 1;
      end
      if(t1>=m) t1 = t1 - m; end
      if(t2>=m) t2 = t2 - m; end
   end
end
```

6.1.2.2 READDATA.M

```
% this function allows MATLAB to read in a data file
% written by the java tempApplet program
% for the Temperon experiment
```

```
% it assumes the datafile root path is c:\java\neuron\
%
% function [points,count] = readData(filePath);

function [points,count] = readData(filePath);

basepath = 'c:\java\neuron\';

fp = [basepath,filePath];

fid = fopen(fp,'r','b');
% The B is for big-endian (which is how java writes)

if (fid == -1)
   error('Problem opening file');
end;

[points,count] = fread(fid,'long');

fclose(fid);
```

6.1.3 Temperon

6.1.3.1 LCTEST.M

```
% this is a test of the Likelihood of Change (L.C.) function2
% function [lc,time] = lctest(stimuli)
% stimuli is a vector of times (in ms) at which the stimuli occur...
% it should be monotonically increasing

function [lc,time] = lctest(stimuli)

stimuli = stimuli(:); % make sure its a column vector

% check stimuli for monotonicity (increasing)
if min(diff(stimuli)) <= 0
   error('Stimuli must be monotonically increasing');
end

if ~(stimuli(1) == 0)
   stimuli = [0; stimuli]; % prepend zero if the first element is not zero
end

iv = .1; % Initial value of L.C.

level = iv;

spac = 5; % how far apart to space the time vectors (resolution)

for count = 2:length(stimuli),   % the time vector in millisecs
   diffr = stimuli(count) - stimuli(count-1); % this is the difference
parameter
   points = stimuli(count-1):spac:stimuli(count); % this is for plotting
purposes...

   points = points - stimuli(count-1) ; % correct for shift

% ***************** The hoopdie-big equation
   scale = 1/100; %useful for playing with the slope of the curve
%  newlevel = (level - log(points/50) + iv); % this does sort of the right
thing
   newlevel = ( (level * exp(- scale * points)) + iv); % but this is more
correct

% ***************** Yup, this is it...

   lc = [lc, newlevel(1:length(newlevel)-1)]; % append all but last points
to the vector we are working on
   level = newlevel(length(newlevel)); % use the LAST point in the test
vector for next iteration
end

% get that pesky last point after the iteration is complete
lc = [lc, level];

time = 0:spac:stimuli(length(stimuli));
```

6.2 Appendix B: JAVA code

6.2.1 NEURON Package

6.2.1.1 Neuron.java

```
package neuron;
import java.util.Vector;

/* Neuron.java -Abstract class establishing general neuron behavior
/
/  Written David J. Cavuto
/  on 10/19/1996
/  Under the auspices of Dr. Simon Ben-Avi
/  at The Cooper Union for the Advancement of Science and Art
/
/  Created in partial fulfillment of the requirements for the
/  Master of Electrical Engineering degree
/  at the Albert Nerkin School of Engineering
/
/  Copyright (c) 1996 David J. Cavuto and The Cooper Union
/  All rights reserved.
/
*/

/**@version 1.0 10/19/1996 */
/**@author David J. Cavuto */


public abstract class Neuron
{

   protected int ninputs=0;      // number of inputs the neuron has
   private Vector inputs;    // pointers to the other neurons

   protected String transFun; // transfer Function as a String
   private Vector  weights;  // vector containing the weights
   private double  bias;     // the bias or threshold level

   public void Neuron()
   {
      ninputs = 0;
      inputs = new Vector();
      transFun = new String("unnamed");
      weights = new Vector();
      bias = 0;
System.out.println("Neuron: constructor");
   };

   abstract double output();

   public void addInput(Neuron newNeuron, double weight)
   {
      ninputs++; // increment the number of inputs the neuron has
      inputs.addElement(newNeuron);
      weights.addElement(new Double(weight));
   }
```

```
   public void removeInputAt(int position)
   {
      if (ninputs>0)
         try
         {
            ninputs--;
            inputs.removeElementAt(position);
            inputs.removeElementAt(position);
         }
         catch (ArrayIndexOutOfBoundsException e)
         {
            System.out.println("neuron.Neuron.removeInputAt: invalid
position");
         }
      else
         System.out.println("neuron.Neuron.removeInputAt: no inputs to
remove");
   }

   public double getWeightAt(int position)
   {
      try
      {
         return ( ((Double)weights.elementAt(position)).doubleValue() );
      }
      catch (ArrayIndexOutOfBoundsException e)
      {
         System.out.println("neuron.Neuron.getWeightAt: invalid
position");
         return 0;
      }
   }

   public void setWeightAt(int position, double val)
   {
      try
      {
         weights.setElementAt(new Double(val),position);
      }
      catch (ArrayIndexOutOfBoundsException e)
      {
         System.out.println("neuron.Neuron.setWeightAt: invalid
position");
      }
   }

   public Neuron getNeuronAt(int position)
   {
      try
      {
         return ( ((Neuron)inputs.elementAt(position)));
      }
      catch (ArrayIndexOutOfBoundsException e)
      {
         System.out.println("neuron.Neuron.getNeuronAt: invalid
position");
         return null;
      }
   }
```

```
    public double getBias()
    {
        return (bias);
    }

    public void setWeights(Vector wts)
    {
        weights = (Vector)wts.clone();
    }

    public void setBias(double bs)
    {
        bias = bs;
    }


}
```

6.2.1.2 Perceptron.java

```
package neuron;
import neuron.Neuron;

/* Perceptron.java - Implements perceptron
/
/  Written David J. Cavuto
/  on 10/19/1996
/  Under the auspices of Dr. Simon Ben-Avi
/  at The Cooper Union for the Advancement of Science and Art
/
/  Created in partial fulfillment of the requirements for the
/  Master of Electrical Engineering degree
/  at the Albert Nerkin School of Engineering
/
/  Copyright (c) 1996 David J. Cavuto and The Cooper Union
/  All rights reserved.
/
*/

/**@version 1.0 10/19/1996 */
/**@author David J. Cavuto */


public class Perceptron extends Neuron
{
    public Perceptron()
    {
        Neuron();
        System.out.println("Perceptron: Constructor");
    }

    public Perceptron(String str)
    {
        Neuron();
        System.out.println("Perceptron: Constructor");
        transFun = new String(str);
    }


    public double output()
```

```
   {
      if (ninputs>0)
      {
         double acc=0; // accumulator
         for(int count=0;count<ninputs;count++)
            acc+= getWeightAt(count) * getNeuronAt(count).output();
         return feval(transFun,acc);
      }
      else
         return 0d;
   }

   protected double feval(String str, double num)
   {
//System.out.println("Perceptron:feval:str >>"+str+"<<");
      if (str.equals(new String("HardLim")))// hard limit xfer function
      {
         num -= getBias(); // adjust for the bias...

         if (num>=0)
            return 1d;
         else
            return 0d;
      }
      else if (str.equals(new String("HardLimNeg")))// hard limit xfer
function
      {
         num -= getBias(); // adjust for the bias...

         if (num>0)
            return 1d;
         else
            return 0d;
      }     else if(str == "unnamed")
      {
         System.out.println("Perceptron:feval: Unspecified transfer
function");
         return 0d;
      }
      else
      {
         System.out.println("Perceptron:feval: Unknown transfer
function");
         return 0d;
      }
   }


}
```

6.2.1.3 PercepFB.java

```
package neuron;
import neuron.Neuron;
import neuron.Perceptron;

/* PercepFB.java - Implements perceptron with feedback and control signals
/
/  Written David J. Cavuto
```

```
/  on 10/19/1996
/  Under the auspices of Dr. Simon Ben-Avi
/  at The Cooper Union for the Advancement of Science and Art
/
/  Created in partial fulfillment of the requirements for the
/  Master of Electrical Engineering degree
/  at the Albert Nerkin School of Engineering
/
/  Copyright (c) 1996 David J. Cavuto and The Cooper Union
/  All rights reserved.
/
*/

/**@version 1.0 10/20/1996 */
/**@author David J. Cavuto */


public class PercepFB extends Perceptron
{
   protected double lastOutput=0;
   private double thisOutput=0;

   public PercepFB()
   {
      super();
      System.out.println("PercepFB: Constructor");
   }

   public PercepFB(String str)
   {
      super();
      System.out.println("PercepFB: Constructor");
      transFun = new String(str);
   }


   public double output()
   {
      return lastOutput;
   }

   public void compute()
   {
      thisOutput = super.output();
   }

   public void updateOutput()
   {
      lastOutput = thisOutput;
   }

   public void setOutput(double out)
   {
      lastOutput = out;
   }


}
```

6.2.1.4 Input.java

```
package neuron;
import neuron.Neuron;

/* Input.java - Implements dumb input layer
/
/  Written David J. Cavuto
/  on 10/19/1996
/  Under the auspices of Dr. Simon Ben-Avi
/  at The Cooper Union for the Advancement of Science and Art
/
/  Created in partial fulfillment of the requirements for the
/  Master of Electrical Engineering degree
/  at the Albert Nerkin School of Engineering
/
/  Copyright (c) 1996 David J. Cavuto and The Cooper Union
/  All rights reserved.
/
*/

/**@version 1.0 10/19/1996 */
/**@author David J. Cavuto */


public class Input extends neuron.Neuron
{
   public Input()
   {
      Neuron();
System.out.println("Input: Constructor");
   }


   private double out = 0;

   public double output()
   {
      return out;
   }

   public void setOutput(double o)
   {
      out = o;
   }
}
```

6.2.1.5 Temperon.java

```
package neuron;
import neuron.Neuron;
import neuron.Perceptron;
import neuron.PercepFB;
import java.util.*;

/* Temperon.java - Implements custom perceptron with inherent learning and
temporal awareness
/
/  Written David J. Cavuto
/  on 11/12/1996
```

```
/  Under the auspices of Dr. Simon Ben-Avi
/  at The Cooper Union for the Advancement of Science and Art
/
/  Created in partial fulfillment of the requirements for the
/  Master of Electrical Engineering degree
/  at the Albert Nerkin School of Engineering
/
/  Copyright (c) 1996 David J. Cavuto and The Cooper Union
/  All rights reserved.
/
*/

/**@version 1.7 12/16/1996 */
/**@author David J. Cavuto */

/* Revision History

   1.7 12/16/1996
      added constructor option to NOT have weight max/min
      added rescaleWeights method

   1.6 12/13/1996
      added revision history
      changed back from random updates of weights to a fixed step size

*/

public class Temperon extends PercepFB
{
   protected Vector lastUpdateTime;
   protected Vector lastUpdateValue;
   final static double INPUT_THRESHOLD=0.5d;
   final static double LEVEL_THRESHOLD=0.5d;
   final static double CHANGE_VALUE=0.2d;
   final static double INITIAL_VALUE=0.1d;
   final static long REPOLARIZATION_TIME = 60;
   final static double WEIGHT_MAX = 3.0d;
   final static double WEIGHT_MIN = -3.0d;
   final static double RESCALE_MAX = 20d;
   final static double SCALE = 1d/100d;
   private static long localtime=0;
   private long lastFire=0;
   private boolean fire_flag = false;
   private boolean  maxWeights = true;
   protected boolean oscillate_flag = true;


   public Temperon()
   {
      super();
      System.out.println("Temperon: Constructor");
      init("HardLim");
   }

   public Temperon(String str)
   {
      super();
      System.out.println("Temperon: Constructor");
      init(str);
   }
```

```
   public Temperon(String str, boolean m)
   {
      super();
      System.out.println("Temperon: Constructor");
      maxWeights = m;
      init(str);
   }


   private void init(String str)
   {
      transFun = new String(str);
      lastUpdateTime = new Vector();
      lastUpdateValue = new Vector();
   }

   public void compute()
   {
      super.compute();
      updateWeights();

   }

   public void setOscillateFlag(boolean f)
   {
      oscillate_flag = f;
   }

   public void updateOutput()
   {
      super.updateOutput();
      if(fire_flag && oscillate_flag) // it fired last time
      {
         super.setOutput(0d); // make it go off this time
         fire_flag = false;
      }
      else if (super.output() > INPUT_THRESHOLD) // if this output fires
this time, set a flag
         fire_flag = true;

//     long tempTime = getTime();
//     if((tempTime-lastFire) < REPOLARIZATION_TIME)
//     {
//        super.setOutput(0d);
//     }
//     lastFire = getTime();

   }

   public void setWeightAt(int position, double val)
   {
      if (getNeuronAt(position)==this) val=0d; //check for self-connect
      super.setWeightAt(position,val);
   }

   public void addInput(Neuron newNeuron, double weight) // overrides
Neuron.addInput()
   {
      if (newNeuron == this) weight = 0; // don't connect 'em to itself
      super.addInput(newNeuron, weight);
      lastUpdateTime.addElement(new Long(getTime()));
```

```
      lastUpdateValue.addElement(new Double(INITIAL_VALUE));
   }

   public void removeInputAt(int position) // overrides
Neuron.removeInputAt()
   {
      super.removeInputAt(position);
      lastUpdateTime.removeElementAt(position);
      lastUpdateValue.removeElementAt(position);
   }

   protected void updateWeights()
   {
      boolean rescaleFlag = false;
      if (ninputs>0)
      {
         double acc=0; // accumulator
         for(int count=0;count<ninputs;count++)
         {
            if (getNeuronAt(count).output() >= INPUT_THRESHOLD)
            {  // input is 'on' and we should update its likelihood of
change
               long now=getTime(); // get the new time
               long
then=((Long)lastUpdateTime.elementAt(count)).longValue(); // get the last
time from the vector
               lastUpdateTime.setElementAt(new Long(now),count); // set
last time <= NOW
               double diff = (double)(now-then); // the difference in
time

               // now we have to compute the new LC and see if it exceeds
the threshold
               double
level=((Double)lastUpdateValue.elementAt(count)).doubleValue();

//             double newLevel = level - (diff/50d)*Math.exp(-diff /
50000d) + INITIAL_VALUE; // compute the new level
               double newLevel = level * Math.exp(-diff * SCALE) +
INITIAL_VALUE;
               lastUpdateValue.setElementAt(new Double(newLevel),count);
// update the Vector

//           System.out.println("updateWeights: input "+count+",
oldlevel="+level+", newLevel="+newLevel);
//           System.out.println("  oldtime="+then+", newTime="+now);
               if (newLevel >= LEVEL_THRESHOLD) // we have to update the
weight
               {
                  double wt = getWeightAt(count);
                  double sign = wt / Math.abs(wt) ; // returns either a
+1 or -1 depending on the sign of wt
//                sign = 1; // ignore the man behind the green curtain...

//           System.out.println("updateWeights: Updating Weights for input
"+count+" with sign: "+sign);
                  double newwt=0;
                  if (lastOutput >= INPUT_THRESHOLD) // if this neuron's
output is 'on'
//                   newwt = +(CHANGE_VALUE*Math.random()*sign); //
reinforce
```

```
                        newwt = +(CHANGE_VALUE*0.5*sign); // reinforce
                    else  // otherwise
//                      newwt = -(CHANGE_VALUE*Math.random()*sign); //
suppress
                        newwt = -(CHANGE_VALUE*0.5*sign); // suppress

                    if (getNeuronAt(count) == this) newwt = 0; // if it is
itself...

                    newwt = wt+newwt;

                    // check the limits
                    if(maxWeights)
                    {
                        if(newwt>WEIGHT_MAX) newwt = WEIGHT_MAX;
                        else if (newwt<WEIGHT_MIN) newwt = WEIGHT_MIN;
                    }

                    // actually do the updating.
                    setWeightAt(count,newwt);

                    // and let's reset the level
                    lastUpdateValue.setElementAt(new
Double(INITIAL_VALUE),count); // update the Vector
                    if(Math.abs(newwt) > RESCALE_MAX) rescaleFlag = true;

                } // if
            } // if
          }// for loop
          if(rescaleFlag) rescaleWeights(RESCALE_MAX);

       }// if
    } // updateWeights

    public static long getTime()
    {
        return localtime;
//      return System.currentTimeMillis();
    }

    protected void rescaleWeights(double rf)
    {
        System.out.println("Temperon: rescaling weights");
        for(int count=0;count<ninputs;count++)
        {
            setWeightAt(count,(getWeightAt(count)/rf));
        }
    }

    public static void updateTime()
    {
        localtime+=10;
        if (localtime>1000000)localtime=0;
    }
}
```

6.2.2 Test programs

6.2.2.1 MultiMouseApplet.java

```
import java.applet.*;
import java.awt.*;
import java.util.*;

/**    multiMouseApplet.java -
*
*   Written David J. Cavuto<BR>
*   on 11/14/1996<BR>
*   Under the auspices of Dr. Simon Ben-Avi<BR>
*   at The Cooper Union for the Advancement of Science and Art<BR>
*   <BR>
*   Created in partial fulfillment of the requirements for the<BR>
*   Master of Electrical Engineering degree<BR>
*   at the Albert Nerkin School of Engineering<BR>
*   <BR>
*   Copyright (c) 1996 David J. Cavuto and The Cooper Union<BR>
*   All rights reserved.<BR>
*
*   @version 1.3 12/13/1996
*   @author David J. Cavuto, The Cooper Union
*/

/*  Revision History

    1.3 12/13/1996
        started keeping history
        fixed cross-screen display problems
        added moveObstacle method

*/

public class multiMouseApplet extends NoFlickerApplet
{
    int height=0;
    int width=0;

    double turtle_x=0;
    double turtle_y=0;
    double turtle_angle=0;
    boolean turtle_pendown=false;
    Color turtle_color = Color.black;

    Vector turtleVector = new Vector();


    Image turtle_image=null;
    Graphics turtle_graphics=null;
    Dimension turtle_size=null;

    double old_x=0;
    double old_y=0;

    double obj_x=0;
    double obj_y=0;
    boolean obj_flag = false;
```

```
   Label tempLabel = new Label("false");
   Label closeLabel = new Label("false");
   Label angLabel = new Label("00000000");
   Label dirLabel = new Label("00000000");

   Choice moveChoice;

   /**
   * This is a test - init()
   *
   */
   public void init()
   {
      height=this.size().height;
      width=this.size().height;
      moveChoice = new Choice();
      moveChoice.addItem("fd(1)");
      moveChoice.addItem("bd(1)");
      moveChoice.addItem("fd(10)");
      moveChoice.addItem("bd(10)");
      moveChoice.addItem("rl(5)");
      moveChoice.addItem("rr(5)");
      moveChoice.addItem("rl(30)");
      moveChoice.addItem("rr(30)");
      moveChoice.addItem("pu()");
      moveChoice.addItem("pd()");
//    add(moveChoice);
//    addObstacle(.5,.5);
//    add(tempLabel);
//    add(closeLabel);
//    add(angLabel);
//    add(dirLabel);
   }

   public void addTurtle(int ID)
   {
      turtleVector.addElement(new turtle(ID));
      repaint();
   }

   public boolean removeTurtle(int ID)
   {
      boolean retvalue = false;
      int pos = getTurtleID(ID);
      if (pos!= -1)
      {
         turtleVector.removeElementAt(pos);
         retvalue = true;
      }
      repaint();
      return retvalue;
   }


   /**
   * We're going to work with the area scaled in the range (-1,-1) to
(1,1)
   * hence, we need to be able to convert the scale we are working in
   * to actual pixels relative to the size of the window we are working
with
```

```
    *
    * @param pos takes a position in the scale -1 to +1 and rescales it to
fit the current window
    * @return the positions rescaled as integers relative to the size of
the current window
    */
    public int rescale(double pos, int scale_factor)
    {
        return( (int)Math.round(((pos+1)/2)*(scale_factor)) );
    }

    public int rX(double pos)
    {
        return(rescale(pos,width));
    }

    public int rY(double pos)
    {
        return(rescale(pos,height));
    }

    public double scale(int pos, int scale_factor)
    {
        return( ((double)pos/(double)scale_factor) * 2d - 1 );
    }

    public double sX(int pos)
    {
        return(scale(pos,width));
    }

    public double sY(int pos)
    {
        return(scale(pos,height));
    }

    public void rangeCheck()
    {
        if(turtle_x < -1d)
            turtle_x = 1d;
        else if(turtle_x > 1d)
            turtle_x = -1d;
        if(turtle_y < -1d)
            turtle_y = 1d;
        else if(turtle_y > 1d)
            turtle_y = -1d;

    }

    public void otherRangeCheck(int ID)
    {
//System.err.println("otherRangeCheck: "+ID);
        double loc_turtle_x = getTurtleX(ID);
        double loc_turtle_y = getTurtleY(ID);
        double loc_turtle_angle = getTurtleAngle(ID);

        if(loc_turtle_x < -1d)
            setTurtleX(ID,1d);
        else if(loc_turtle_x > 1d)
            setTurtleX(ID,-1d);
        if(loc_turtle_y < -1d)
```

```
            setTurtleY(ID,1d);
         else if(loc_turtle_y > 1d)
            setTurtleY(ID,-1d);
      }


      protected void updateTurtleGraphics()
      {
         Dimension d = this.size();
         if((turtle_image==null) || (d.width!=turtle_size.width) ||
            (d.height!=turtle_size.height)) // have to make a new one
         {
            turtle_image = createImage(d.width, d.height);
            turtle_size = d;
            turtle_graphics = turtle_image.getGraphics();
            turtle_graphics.setColor(this.getBackground());
            turtle_graphics.fillRect(0,0,d.width, d.height);

            //let's warm up by drawing an axis
            turtle_graphics.setColor(Color.darkGray);
            turtle_graphics.drawLine(rX(-1),rY(0),rX(1),rY(0)); // horizontal
axis
            turtle_graphics.drawLine(rX(0),rY(-1),rX(0),rY(1)); // vertical
axis
         }
      }

      private void moveHandler() //called on any forward or backward movement
      {
         if(moveCheck() && turtle_pendown) doLine(turtle_graphics);

         rangeCheck();
      }

      private void netMoveHandler()
      {
         for(int count=0;count<turtleVector.size();count++)
         {
            int ID = (((turtle)turtleVector.elementAt(count)).getID());
            otherRangeCheck(ID);
            if(moveCheck(ID) && getPenDown(ID)) doLine(ID,turtle_graphics);
         }
      }

      public void paint(Graphics g)
      {
         height=this.size().height;
         width=this.size().width;

         updateTurtleGraphics();

         g.drawImage(turtle_image,0,0,null); // update the background z-
buffer

         if(obj_flag)
         {
            drawObstacle(g,obj_x,obj_y);
//          tempLabel.setText(String.valueOf(facingObstacle()));
//          closeLabel.setText(String.valueOf(closeToObstacle()));
//          angLabel.setText(String.valueOf(turtle_angle));
//          dirLabel.setText(String.valueOf(angleToObstacle()));
```

```
//        angLabel.setText(String.valueOf(rightHandObstacle()));
//        dirLabel.setText(String.valueOf(pointingAtObstacle()));


      }

      old_x = turtle_x; //update the old positions (local)
      old_y = turtle_y;

      drawTurtle(g, turtle_x,turtle_y,turtle_angle,turtle_color); // draw
(local)
turtle


      drawAllTurtles(g); // do all the damn other ones

   }

   protected void drawAllTurtles(Graphics g)
   {
      for(int count=0;count<turtleVector.size();count++)
      {
         drawTurtle(g,((turtle)turtleVector.elementAt(count)).turtle_x,
                ((turtle)turtleVector.elementAt(count)).turtle_y,
                ((turtle)turtleVector.elementAt(count)).turtle_angle,
                ((turtle)turtleVector.elementAt(count)).turtle_color
                );
      }
   }

   protected boolean moveCheck()
   {
      if(turtle_x == old_x && turtle_y == old_y)
         return false;
      else return true;
   }

   protected boolean moveCheck(int ID)
   {  double loc_turtle_x = getTurtleX(ID);
      double loc_turtle_y = getTurtleY(ID);
      double loc_turtle_angle = getTurtleAngle(ID);

      if(loc_turtle_x == old_x && loc_turtle_y == old_y)
         return false;
      else return true;
   }

   protected void doLine(int ID, Graphics _g)
   {
      double loc_turtle_x = getTurtleX(ID);
      double loc_turtle_y = getTurtleY(ID);
      double loc_old_x = getOldX(ID);
      double loc_old_y = getOldY(ID);
      if(checkDistance(loc_turtle_x,loc_old_x) &&
checkDistance(loc_turtle_y,loc_old_y) )
      {
         _g.setColor(Color.blue);
         _g.drawLine(rX(loc_old_x), rY(loc_old_y), rX(loc_turtle_x),
rY(loc_turtle_y));
      }
   }
```

```
   private boolean checkDistance(double x1, double x2) // if the distances
cross a screen boundry, return false
   {
      if( Math.abs(x2-x1) > 1.95 )return false;
      else return true;
   }

   protected void doLine(Graphics _g)
   {
      if(checkDistance(turtle_x,old_x) && checkDistance(turtle_y,old_y) )
      {
         _g.setColor(Color.blue);
         _g.drawLine(rX(old_x), rY(old_y), rX(turtle_x), rY(turtle_y));
      }
   }

   /* ***************** Vector stuff ************* */


   public int getTurtleID(int ID)
   {
      int pos=-1;
      for(int count=0;count<turtleVector.size();count++)
      {
         if( ((turtle)turtleVector.elementAt(count)).getID() == ID)
            pos = count;
      }
      return pos;
   }

   protected Color getTurtleColor(int ID)
   {
      int pos = getTurtleID(ID);
      return ((turtle)turtleVector.elementAt(pos)).turtle_color;
   }

   protected double getTurtleX(int ID)
   {
      int pos = getTurtleID(ID);
      return ((turtle)turtleVector.elementAt(pos)).turtle_x;
   }

   protected double getTurtleY(int ID)
   {
      int pos = getTurtleID(ID);
      return ((turtle)turtleVector.elementAt(pos)).turtle_y;
   }

   protected double getOldY(int ID)
   {
      int pos = getTurtleID(ID);
      return ((turtle)turtleVector.elementAt(pos)).old_y;
   }

   protected double getOldX(int ID)
   {
      int pos = getTurtleID(ID);
      return ((turtle)turtleVector.elementAt(pos)).old_x;
   }
```

```
   protected double getTurtleAngle(int ID)
   {
      int pos = getTurtleID(ID);
      return ((turtle)turtleVector.elementAt(pos)).turtle_angle;
   }

   protected boolean getPenDown(int ID)
   {
      int pos = getTurtleID(ID);
      return ((turtle)turtleVector.elementAt(pos)).turtle_pendown;
   }
   protected void setTurtleX(int ID, double val)
   {
      int pos = getTurtleID(ID);
      ((turtle)turtleVector.elementAt(pos)).old_x =
((turtle)turtleVector.elementAt(pos)).turtle_x;
      ((turtle)turtleVector.elementAt(pos)).turtle_x = val;
   }

   protected void setTurtleY(int ID, double val)
   {
      int pos = getTurtleID(ID);
      ((turtle)turtleVector.elementAt(pos)).old_y =
((turtle)turtleVector.elementAt(pos)).turtle_y;
      ((turtle)turtleVector.elementAt(pos)).turtle_y = val;
   }

   protected void setTurtleAngle(int ID, double val)
   {
      int pos = getTurtleID(ID);
      ((turtle)turtleVector.elementAt(pos)).turtle_angle = val;
   }

   protected void setTurtlePenUp(int ID)
   {
      int pos = getTurtleID(ID);
      ((turtle)turtleVector.elementAt(pos)).turtle_pendown = false;
   }

   protected void setTurtlePenDown(int ID)
   {
      int pos = getTurtleID(ID);
      ((turtle)turtleVector.elementAt(pos)).turtle_pendown = true;
   }

   protected void setTurtleColor(int ID, Color c)
   {
      int pos = getTurtleID(ID);
      ((turtle)turtleVector.elementAt(pos)).turtle_color = c;
   }

   public void pu(int ID)
   {
      setTurtlePenUp(ID);
   }

   public void pd(int ID)
   {
      setTurtlePenDown(ID);
   }
```

```
   /* ****************************************** */


   public void pu()
   {
      turtle_pendown = false;
   }

   public void pd()
   {
      turtle_pendown = true;
   }


   /**
   * Moves the Turtle forward
   * note that "forward" is relative to whichever direction it is pointing
   *
   * @param how_far distance in pixels
   */
   public void fd(int how_far) // moves it in whichever direction it is
pointing
   {
      double[] t_x = {turtle_x};
      double[] t_y = {sY(rY(turtle_y) - how_far)};
      rotateDoublePoints(t_x,t_y,turtle_x,turtle_y,turtle_angle);
//    System.out.println("It was at: x="+turtle_x+", y="+turtle_y);
      turtle_x = t_x[0];
      turtle_y = t_y[0];
//    System.out.println("It's now at: x="+turtle_x+", y="+turtle_y);
      moveHandler();
      repaint();
   }

   /**
   * Moves the Turtle forward -- PARAMETERIZED
   * note that "forward" is relative to whichever direction it is pointing
   *
   * @param how_far distance in pixels
   */
   public void fd(int ID, int how_far) // moves it in whichever direction
it is pointing
   {
      double loc_turtle_x = getTurtleX(ID);
      double loc_turtle_y = getTurtleY(ID);
      double loc_turtle_angle = getTurtleAngle(ID);

      double[] t_x = {loc_turtle_x};
      double[] t_y = {sY(rY(loc_turtle_y) - how_far)};

   rotateDoublePoints(t_x,t_y,loc_turtle_x,loc_turtle_y,loc_turtle_angle);

      setTurtleX(ID,t_x[0]);
      setTurtleY(ID,t_y[0]);
      netMoveHandler();
      repaint();
   }

   /**
   * Moves the Turtle backward
```

```
    * note that "backward" is relative to whichever direction it is
pointing
    *
    * @param how_far distance in pixels
    */
    public void bd(int how_far) // moves it backward
    {
       fd(-how_far);
    }

    /**
    * Moves the Turtle backward -- PARAMETERIZED
    * note that "backward" is relative to whichever direction it is
pointing
    *
    * @param how_far distance in pixels
    */
    public void bd(int ID,int how_far) // moves it backward
    {
       fd(ID, -how_far);
    }


    /**
    * Rotates the turtle left (clockwise)
    *
    * @param how_far clockwise angle in degrees
    */
    public void rl(double how_far) //how_far is in degrees
    {
       turtle_angle -= (how_far * Math.PI/180d);
       turtle_angle = fixAngle(turtle_angle);
       repaint();
    }

    /**
    * Rotates the turtle left (clockwise) -- PARAMETERIZED
    *
    * @param how_far clockwise angle in degrees
    */
    public void rl(int ID, double how_far) //how_far is in degrees
    {
       double loc_turtle_angle = getTurtleAngle(ID);

       loc_turtle_angle -= (how_far * Math.PI/180d);
       loc_turtle_angle = fixAngle(loc_turtle_angle);

       setTurtleAngle(ID,loc_turtle_angle);

       repaint();
    }

    /**
    * Rotates the turtle right (counterclockwise)
    *
    * @param how_far ccw angle in degrees
    */
    public void rr(double how_far)
    {
       rl(-how_far);
    }
```

```
   /**
   * Rotates the turtle right (counterclockwise) -- PARAMETERIZED
   *
   * @param how_far ccw angle in degrees
   */
   public void rr(int ID,double how_far)
   {
      rl(ID, -how_far);
   }



   public double fixAngle(double ang) // recursively puts an angle in the
range 0 - 2*pi
   {
      if (ang>2*Math.PI) return fixAngle(ang-2*Math.PI); // if its greater
than 360, subtract 360 from it
         else if (ang<0) return fixAngle(2*Math.PI+ang); // it its less
than 360, add 360 to it
         else return ang; // otherwise just return it
   }

   public boolean action(Event e, Object o)
   {
      boolean ret_value=false;

      if(e.target instanceof Choice)
      {
         doit(moveChoice.getSelectedIndex());
         ret_value=true;
      }

      return ret_value;
   }

   public void doit(int which)
   {
      switch (which)
      {
         case 0: fd(1); break;
         case 1: bd(1); break;
         case 2: fd(10); break;
         case 3: bd(10); break;
         case 4: rl(5); break;
         case 5: rr(5); break;
         case 6: rl(30); break;
         case 7: rr(30); break;
         case 8: pu(); break;
         case 9: pd(); break;
      }
      repaint();
   }

   public boolean handleEvent(Event e)
   {
/*    if(e.id==Event.MOUSE_MOVE)
      {
         double tx = sX(e.x) - turtle_x;
         double ty = sY(e.y) - turtle_y;
         turtle_angle = Math.atan2(ty,tx) + Math.PI/2d;
```

```
            repaint();
        }
        if(e.id==Event.MOUSE_DOWN)
        {
            fd(10);
//          turtle_x = sX(e.x);
//          turtle_y = sY(e.y);
            repaint();
        }
        if(e.id==Event.MOUSE_DRAG)
        {
//          turtle_x = sX(e.x);
//          turtle_y = sY(e.y);
            repaint();
        }
*/
        if(e.id==Event.ACTION_EVENT)
            return action(e,e.arg);

        return(false); // so something else will handle it.
    }

    public void drawTurtle(Graphics g, double _x,double _y, double rot)
    {
        drawTurtle(g,_x,_y,rot,Color.black);
    }

    public void drawTurtle(Graphics g, double _x,double _y, double rot,
Color col)
    {
        // for now, rot is an angle in radians

        int size=6; // Make the little mousie
        int x_o = rX(_x); // x_origin
        int y_o = rY(_y); // y_origin
        int x[] = {x_o,x_o+size,x_o-size,x_o};
        int y[] = {y_o-size,y_o+size,y_o+size,y_o-size};

        int size2 = size/2; // Make his cute little snout
        int y_o2 =y_o-size2;
        int x1[] = {x_o,x_o+size2,x_o-size2,x_o};
        int y1[] = {y_o2-size2,y_o2+size2,y_o2+size2,y_o2-size2};

        // now, the x,y and x1,y1 arrays hold the points for the
        // turtle body and its head in an upright position
        // we have to rotate the damn points
        rotatePoints(x,y,x_o,y_o,rot);
        rotatePoints(x1,y1,x_o,y_o,rot);

        g.setColor(col);
        g.fillPolygon(x,y,4);
        g.setColor(Color.white);
        g.fillPolygon(x1,y1,4);

    }

    public void addObstacle(double _x, double _y)
    {
        obj_x = _x;
        obj_y = _y;
        obj_flag = true;
```

```
   }

   public void drawObstacle(Graphics g, double _x, double _y)
   {
      g.setColor(Color.yellow);
      g.fillOval(rX(_x),rY(_y),10,10);
   }

   public void drawTurtle(Graphics g, double _x,double _y)
   {
      drawTurtle(g,_x,_y,0);
   }

   public double angleToObstacle()
   {
      return fixAngle(Math.atan2(turtle_y-obj_y,turtle_x-obj_x)-
Math.PI/2d);
   }

   public boolean facingObstacle()
   {
      double ao = fixAngle(Math.abs(angleToObstacle() - turtle_angle));

      if ( (ao>0 && ao < Math.PI/2d) || (ao>3d*(Math.PI/2d)) )
         return true;

      else return false;
   }

   public boolean rightHandObstacle()
   {
      double ao = fixAngle(Math.abs(angleToObstacle() - turtle_angle-
Math.PI/2d));

      if ( (ao>0 && ao < Math.PI/2d) || (ao>3d*(Math.PI/2d)) )
         return true;
      else return false;
   }

   public boolean pointingAtObstacle()
   {
      double ao = fixAngle(Math.abs(angleToObstacle() - turtle_angle));
      if ( (ao>0 && ao < Math.PI/8d) || (ao> (2d*Math.PI - Math.PI/8d)) )
         return true;
      else return false;
   }

   public boolean closeToObstacle()
   {
      double dx = obj_x-turtle_x;
      double dy = obj_y-turtle_y;
      double dist = Math.sqrt((dx*dx)+(dy*dy)); // Euclidean distance -
sqrt of sum of squares
      if (dist<.2) return true;
      else return false;
   }

   public boolean proximity(int ID,double rad)
   {
      if(getTurtleID(ID)!=-1)
```

```
      {
         double loc_turtle_x = getTurtleX(ID);
         double loc_turtle_y = getTurtleY(ID);
         double dx = loc_turtle_x-turtle_x;
         double dy = loc_turtle_y-turtle_y;
         double dist = Math.sqrt((dx*dx)+(dy*dy)); // Euclidean distance -
sqrt of sum of squares
System.err.println("mmA: proximity: dist = "+dist);
         if (dist<rad) return true;
         else return false;
      }
      else return false; // if the other turtle doesn't exist return false
   }

   /**
   * Positions the obstacle where you want it.
   *
   */
   public void moveObstacle(double _x, double _y)
   {
      obj_x = _x;
      obj_y = _y;
   }



/*  public boolean pointingAtObstacle()
   {
      double EPSILON = 0.001;
      double ao = angleToObstacle() - turtle_angle + Math.PI/2d;
   }
*/

   /**
   * Rotates a set of points (doubles) about an origin a specified number
of radians
   * @param x An array of x-values of points
   * @param y An array of y-values of points
   * @param x_o the x-origin
   * @param y_o the y-origin
   * @param rot Rotation angle in radians
   */
   public void rotateDoublePoints(double[] x, double[] y, double x_o,
double y_o, double rot)
   {
      if(rot != 0d)
      {
         double xt=0;
         double yt=0;
         double c = Math.cos(rot);
         double s = Math.sin(rot);
         if(x.length!=y.length)
            System.out.println("rotatePoints: x and y must have the name
number of elements");
         int npoints = x.length;
         for (int count=0;count<npoints;count++)
         {
            xt = x[count]; // temp values for the x and y;
            yt = y[count];
            xt -= x_o; // move the point so it is being rotated around
the origin
```

```
            yt -= y_o;
//          System.out.println("Old point x= "+x[count]+", y="+y[count]);

            x[count] = (c*xt - s*yt) + x_o; // compute rotation and
correct for origin
            y[count] = (s*xt + c*yt) + y_o;
//          System.out.println("New point x= "+x[count]+", y="+y[count]);
         }
      }
   }

   /**
   * Rotates a set of points (integers) about an origin a specified number
of radians
   * @param x An array of x-values of points
   * @param y An array of y-values of points
   * @param x_o the x-origin
   * @param y_o the y-origin
   * @param rot Rotation angle in radians
   */
   public void rotatePoints(int[] x, int[] y, int x_o, int y_o, double
rot)
   {
      if(rot != 0d)
      {
         int xt=0;
         int yt=0;
         double c = Math.cos(rot);
         double s = Math.sin(rot);
         if(x.length!=y.length)
            System.out.println("rotatePoints: x and y must have the name
number of elements");
         int npoints = x.length;
         for (int count=0;count<npoints;count++)
         {
            xt = x[count]; // temp values for the x and y;
            yt = y[count];
            xt -= x_o; // move the point so it is being rotated around
the origin
            yt -= y_o;
//          System.out.println("Old point x= "+x[count]+", y="+y[count]);

            x[count] = (int)Math.round(c*xt - s*yt) + x_o; // compute
rotation and correct for origin
            y[count] = (int)Math.round(s*xt + c*yt) + y_o;
//          System.out.println("New point x= "+x[count]+", y="+y[count]);
         }
      }
   }


}

class turtle
{
   public double turtle_x = 0d;
   public double turtle_y = 0d;
   public double turtle_angle = 0d;
   public boolean turtle_pendown = false;
   protected int unique_ID;
   public double old_x = 0d;
```

```
    public double old_y = 0d;
    public Color turtle_color = Color.red;

    turtle(int ID) // constructor
    {
       unique_ID = ID;
    }

    public int getID()
    {
       return(unique_ID);
    }

}
```

6.2.2.2 TempApplet.java

```
import neuron.*;

import java.applet.*;
import java.awt.*;
import java.io.*;
import graph.*;

/* tempApplet.java - Test applet for the neuron.Neuron.Temperon class
/  This is an attempte to implement a Hopfield network.
/  NOT part of the neuron.* package
/  Here, we want to see if the neurons will actually update themselves...
/
/  Written David J. Cavuto
/  on 11/13/1996
/  Under the auspices of Dr. Simon Ben-Avi
/  at The Cooper Union for the Advancement of Science and Art
/
/  Created in partial fulfillment of the requirements for the
/  Master of Electrical Engineering degree
/  at the Albert Nerkin School of Engineering
/
/  Copyright (c) 1996 David J. Cavuto and The Cooper Union
/  All rights reserved.
/
*/

/**@version 1.6 12/13/1996 */
/**@author David J. Cavuto */

/* Revision history

   1.8 12/16/1996
      Changed setupNeuralElement to reflect new constructor in Temperon

   1.7 12/14/1996
      Discovered the problem... the convert outputs method uses the
displayed outputs,
      not the actual ones, so if the display isn't updated fast enough,
they are wrong.
      Added netOutput[] array to handle this.

   1.6 12/13/1996
      started keeping history
```

```
       now uses multiMouseApplet (designed for Tag program), with better
display
       obstacle can be controlled locally
       added weight IO
*/

public class tempApplet extends java.applet.Applet implements Runnable
{
   String WEIGHT_FILE_NAME = "weights";
   int fileCounter = 0;
   int numDataPoints=0;
   int numOutputPoints=0;
   int numNeurons;
   int numInputs = 5;
   int delay=50;
   Panel neuronPanel, ioPanel;
   Temperon[] fbNeuron;
   Input[] inputNeuron;
   Panel[] fbPanel;
   Panel buttonPanel;
   TextField[][] weightMatrix;
   TextField[]  inputArray;
   TextField delayField;
   Label timeField;
   Button clockButton,resetButton,updateButton,
      timeButton,randomButton,goButton,saveButton,loadButton;
   Label[] outLabel;
   boolean weightUpdateFlag = false;
   Thread goThread=null;
   Image osi = null;
   Graphics osg = null;
   double[] netOutput;

   int MAX_GRAPH_POINTS = 500;

   Graph2D outputGraph;
   DataSet outputData;
   Axis outputAxis;
   Frame outFrame, mouseFrame;
   int iwidth=0;
   int iheight=0;
   double obj_x=0.5;
   double obj_y=0.5;

   Checkbox weightUpdateBox, graphUpdateBox;

   static int MOVE_MASK = 3; // 0b000000011 masks out the two LSBs


   multiMouseApplet mouseApp = new multiMouseApplet();


   DataOutputStream outputDataFile;
   BufferedOutputStream outputBuffer;
   FileOutputStream outputFile;
   final int BUFFER_SIZE = 1024*10;
   int FLUSH_EVERY; // flush the buffer thru to a file once in a while

   public void init()
   {
      String s = getParameter("numNeurons");
```

```
//System.out.println("s = >>"+s+"<<");
       if (s == "null")
          s = "3"; // in case there is no numNeurons parameter
       numNeurons = (Integer.valueOf(s)).intValue();

       netOutput = new double[numNeurons];

       s = getParameter("flush_every");
//System.out.println("s = >>"+s+"<<");
       if (s == "null")
          s = "500"; // in case there is no numNeurons parameter
       FLUSH_EVERY = (Integer.valueOf(s)).intValue();

       s = getParameter("obj_x");
//System.out.println("s = >>"+s+"<<");
       if (s == "null")
          s = String.valueOf(obj_x); // in case
       obj_x = (Double.valueOf(s)).doubleValue();

       s = getParameter("obj_y");
//System.out.println("s = >>"+s+"<<");
       if (s == "null")
          s = String.valueOf(obj_y); // in case
       obj_y = (Double.valueOf(s)).doubleValue();


       /* Deal with File output Stuff *********************/
       File theFile = checkFileName("netdata");
       System.out.println("Creating datafile: >>"+theFile.getPath()+"<<");
       try{outputFile = new FileOutputStream(theFile);}catch(IOException
e){System.out.println(e);};
       outputBuffer = new BufferedOutputStream(outputFile,BUFFER_SIZE);
       outputDataFile = new DataOutputStream(outputBuffer); // the is the
one we actually write to

       /* Deal with Layout stuff **************************/
       setLayout(new BorderLayout());
       neuronPanel = new Panel();
       add("Center",neuronPanel);
       neuronPanel.setLayout(new GridLayout(1,numNeurons+numInputs));
       buttonPanel = new Panel();
       add("South",buttonPanel);
       ioPanel = new Panel();
       add("West",ioPanel);
       ioPanel.setLayout(new GridLayout(3,1));

       timeField = new Label("0000000000");
       buttonPanel.add(timeField);
       clockButton = new Button("Clock");
       buttonPanel.add(clockButton);
       timeButton = new Button("Update Time");
       buttonPanel.add(timeButton);
       goButton = new Button("GO!");
       buttonPanel.add(goButton);
       delayField = new TextField("000"+String.valueOf(delay));
       buttonPanel.add(delayField);
       weightUpdateBox = new Checkbox("Display Weights?",null,true);
       buttonPanel.add(weightUpdateBox);
       graphUpdateBox = new Checkbox("Update Graph?",null,true);
       buttonPanel.add(graphUpdateBox);
```

```
      resetButton = new Button("Reset Inputs");
      add("North",resetButton);
      updateButton = new Button("Update Weights");
      add("East",updateButton);
      randomButton = new Button("Random Weights");
      ioPanel.add(randomButton);
      loadButton = new Button("Load Weights...");
      ioPanel.add(loadButton);
      saveButton = new Button("Save Weights");
      ioPanel.add(saveButton);

      /* Initialize array of Neurons *********************/
      fbNeuron = new Temperon[numNeurons]; // init array
      inputNeuron = new Input[numInputs];
      fbPanel = new Panel[numNeurons+numInputs]; // give each Neuron a
panel in the grid to sit in.
      weightMatrix = new TextField[numNeurons][numNeurons+numInputs]; //
set up matrix of weights
      inputArray = new TextField[numNeurons+numInputs]; // set up array of
inputs
      outLabel = new Label[numNeurons+numInputs]; // set up array of
labels
      for(int count=0;count<(numNeurons);count++)
      {
         setupNeuralElement(count);
      } // layout and initialize each one
      for(int count=0;count<(numInputs);count++)
      {
         setupInputElement(numNeurons+count);
      } // layout and initialize each one

      /* ************* Position Obstacle *************** */
      mouseApp.addObstacle(obj_x,obj_y);
      mouseApp.pd(); // set the pen down so we draw a line...

      repaint();

      fullyConnectNeurons();
      updateAllNeurons();

      outFrame = new Frame("ANN Data Output");
      outFrame.setLayout(new BorderLayout());

      /* ****** Setup graph */
      outputGraph = new Graph2D();
      outputGraph.zerocolor = new Color(0,255,0);
      outputGraph.borderTop = 0;
      outputGraph.borderBottom = 0;
      outputGraph.setDataBackground(Color.black);
      outFrame.add("Center",outputGraph);
      outFrame.resize(600,300);


      /* setup dataset */
      outputData = new DataSet();
      outputData.linecolor = new Color(255,0,0);
      outputData.marker = 1;
      outputData.markercolor = new Color(100,100,255);


      outputAxis = outputGraph.createAxis(Axis.LEFT);
```

```
      outputAxis.attachDataSet(outputData);
      outputGraph.attachDataSet(outputData);
      outputData.yaxis.maximum = (int)Math.pow(2,numNeurons);
      outputData.yaxis.minimum = 0;
      outFrame.show();
      outFrame.pack();

      /* Make the mousey applet in a frame */
      mouseFrame = new Frame("Mousey");
      mouseApp.init();
      mouseApp.start();
      mouseFrame.add("Center",mouseApp);
      mouseFrame.resize(400,400);
      mouseFrame.show();
      initOutputs();

   }

   public void destroy()
   {
    /* deal with the file closure and stuff *************/
      System.out.println("Wrote "+outputDataFile.size()+" bytes to data
stream");
      System.out.println("for a total of "+numOutputPoints+" data
points");
      flushAll();
      try
      {
         outputDataFile.close();
         outputBuffer.close();
         outputFile.close();
      }
      catch (IOException e)
      {
         System.out.println(e);
      }
      mouseFrame.dispose();
      outFrame.dispose();
   }

        public void flushAll()
        {
      System.out.println("Flushing: up to -- "+numOutputPoints+" data
points");
      try
      {
         outputDataFile.flush();
         outputBuffer.flush();
         outputFile.flush();
      }
      catch (IOException e)
      {
         System.out.println(e);
      }
       }

   protected File checkFileName(String s, int _count)
   {
      File theFile = null;
      int count=_count;
```

```
      boolean fileExists = true;
      while(fileExists)
      {
         String fname = new String(s+count+".dat");
         try
         {
            System.out.println("Looking for file: >>"+fname+"<<");
            theFile = new File(fname);
            fileExists = theFile.exists();
         }
         catch (NullPointerException e)
         {
            System.out.println(e);
         }
         count++; // update the number for the loop
      }
      if (fileCounter == 0)fileCounter = count-1;
      return theFile;
   }

   protected File checkFileName(String s)
   {
      return checkFileName(s,1);
   }

   private void setupNeuralElement(int index)
   {
      fbNeuron[index] = new Temperon("HardLimNeg",true); // make a new
Neuron
      fbNeuron[index].setOscillateFlag(true); // make it (not) oscillate

      fbPanel[index] = new Panel();  // make a new Panel to stick it in
      neuronPanel.add(fbPanel[index],index); // add the Panel to the main
layout at the correct grid position

      /* Here we add all the fun stuff to the panel */
      /* namely, the initial input, the output, and all the weights */
      fbPanel[index].setLayout(new GridLayout(numNeurons+numInputs+2,1));

      // First do the input
      inputArray[index] = new TextField(String.valueOf(0d));
      fbPanel[index].add(inputArray[index],0); // add the input to the top
of the panel

      // do the Output next
      outLabel[index] = new Label("0",Label.CENTER);
      fbPanel[index].add(outLabel[index],1); // add the output Label to
the next position

      // now do all the weights;
      for(int count=0;count<numNeurons+numInputs;count++) // for each of
the weights
      {
         weightMatrix[index][count] = new TextField(String.valueOf(0d));
// make a new one
         fbPanel[index].add(weightMatrix[index][count],count+2); // and
add it to the panel
      }
//    weightMatrix[index][index].setBackground(Color.blue);
//    weightMatrix[index][index].setForeground(Color.white);
   }
```

```
   private void setupInputElement(int index)
   {
      inputNeuron[index-numNeurons] = new Input(); // make a new Neuron
      fbPanel[index] = new Panel();  // make a new Panel to stick it in
      neuronPanel.add(fbPanel[index],index); // add the Panel to the main
layout at the correct grid position

      /* Here we add all the fun stuff to the panel */
      /* namely, the initial input, and the output*/
      fbPanel[index].setLayout(new GridLayout(numNeurons+2,1));

      // First do the input
      inputArray[index] = new TextField(String.valueOf(0d));
      fbPanel[index].add(inputArray[index],0); // add the input to the top
of the panel

      // do the Output next
      outLabel[index] = new Label("0",Label.CENTER);
      fbPanel[index].add(outLabel[index],1); // add the output Label to
the next position

   }


   public void fullyConnectNeurons()
   {
      for(int outNeuron=0;outNeuron<numNeurons;outNeuron++)
      {
         for(int inNeuron=0;inNeuron<numNeurons;inNeuron++)
               fbNeuron[outNeuron].addInput(fbNeuron[inNeuron],0d);
         for(int inNeuron=0;inNeuron<numInputs;inNeuron++)
               fbNeuron[outNeuron].addInput(inputNeuron[inNeuron],0d);
      }
   }


   public void paint(Graphics g)
   {
// System.out.println("Paint!");
      redrawOutputs();
   }

   private void redrawOutputs()
   {
//    for(int count=0;count<(numNeurons);count++)
//
   outLabel[count].setText(String.valueOf(fbNeuron[count].output()));
      for(int count=0;count<(numNeurons);count++)
         outLabel[count].setText(String.valueOf(netOutput[count]));
           for(int count=0;count<numInputs;count++)

   outLabel[count+numNeurons].setText(String.valueOf(inputNeuron[count].ou
tput()));
   }

   public void updateAllNeurons()
   {
      for(int count=0;count<(numNeurons);count++)
         fbNeuron[count].compute();// make them all recompute
      for(int count=0;count<(numNeurons);count++)
```

```
          fbNeuron[count].updateOutput();     // make them all update their
outputs
   }

   public void collectOutputs()
   {
      int count=0;
      for(count=0;count<numNeurons;count++)
         netOutput[count]=fbNeuron[count].output();
   }

   public void initOutputs()
   {
      int count=0;
      for(count=0;count<numNeurons;count++)
         netOutput[count]=0;
   }

   public void clockHandler()
   {
      Temperon.updateTime();
//    System.out.println("System time is: "+Temperon.getTime());
      timeField.setText(String.valueOf(Temperon.getTime()));

      collectOutputs(); // stick 'em in an array

      if(weightUpdateFlag) // if they need to be updated...
      {
         updateWeights(); // update 'em
         weightUpdateFlag = false; // and don't update 'em next time
      }

      inputHandler();
      updateAllNeurons();
      if (weightUpdateBox.getState())
      {
         repaintWeights();
      }
      if (graphUpdateBox.getState())
      {
         graphHandler();
      }

      // Here's the file output stuff
      try
      {
         outputDataFile.writeInt((int)convertOutputs());
         numOutputPoints++;
      }
      catch (IOException e)
      {
         System.out.println(e);
      }

//    System.out.println(">> "+convertOutputs());

      if (numOutputPoints % FLUSH_EVERY == 0)
         flushAll();

      mouseHandler();
```

```
//     redrawOutputs();
       repaint();
   }

   public void inputHandler()
   {
      // we are going to tie the first input to the 'facingObstacle'
method
      if(mouseApp.facingObstacle()) inputNeuron[0].setOutput(1);
         else inputNeuron[0].setOutput(0);
      // the second to the 'rightHandObstacle' method
      if(mouseApp.rightHandObstacle()) inputNeuron[1].setOutput(1);
         else inputNeuron[1].setOutput(0);
      // the third to the 'pointingAt' method
      if(mouseApp.pointingAtObstacle()) inputNeuron[2].setOutput(1);
         else inputNeuron[2].setOutput(0);
      // and the fourth to the 'closeTo' method
      if(mouseApp.closeToObstacle()) inputNeuron[3].setOutput(1);
         else inputNeuron[3].setOutput(0);
      // add a fifth which is opposite of the righthand obstacle
      if(mouseApp.rightHandObstacle()) inputNeuron[4].setOutput(0);
         else inputNeuron[4].setOutput(1);
   }

   public void mouseHandler()
   {
      int check = (int)convertOutputs() & MOVE_MASK;
      switch(check)
      {
         case 3: mouseApp.fd(1);break;
         case 2: mouseApp.rl(2);break;
         case 1: mouseApp.rr(2);break;
      }
   }

   public void graphHandler()
   {
      double data[]=new double[2];
      Graphics g;

      // add the new data points to the graph...
      data[1] = (double)convertOutputs();
      data[0] = numDataPoints++;

      if(numDataPoints >= MAX_GRAPH_POINTS)
      {
         outputData.delete(0,0);
//       System.out.println("Deleting a point from the graph...");
      }

      try
      {
         outputData.append(data,1);
      }
      catch (Exception e)
      {
         System.out.println("clockHandler: error appending data");
      }

               g = outputGraph.getGraphics();
         if( osi == null || iwidth != outputGraph.size().width
```

```
                                    || iheight != outputGraph.size().height
)
         {
                        iwidth = outputGraph.size().width;
                        iheight = outputGraph.size().height;
                        osi = outputGraph.createImage(iwidth,iheight);
                        osg = osi.getGraphics();
                    }

                    osg.setColor(outFrame.getBackground());
                    osg.fillRect(0,0,iwidth,iheight);
                    osg.setColor(g.getColor());

                    osg.clipRect(0,0,iwidth,iheight);

                    outputGraph.update(osg);

                    g.drawImage(osi,0,0,outputGraph);

       outFrame.repaint();

   }

   public void goHandler()
   {
       if (goThread==null)
       {

          delay = Integer.parseInt(delayField.getText());
//System.out.println("got here");
          goThread = new Thread(this);
          goThread.start();
       }
       else
          goThread = null;

   }

   public void run()
   {
       long time = System.currentTimeMillis();
       while (goThread!=null)
       {
          try
          {
             time += delay; // miliseconds between clocks
             Thread.sleep(Math.max(0,time - System.currentTimeMillis()));
          }
          catch (InterruptedException e){};

          clockHandler();
       }
   }

   public boolean action(Event e, Object o)
   {
       if (e.target instanceof Button) // the Clock button was pushed so
update outputs...
       {
          if(o=="Clock")
          {
```

```
            clockHandler();
            return true;
         }
         else if (o=="Reset Inputs")
         {
            for(int count=0;count<(numNeurons);count++)

   fbNeuron[count].setOutput(Double.valueOf(inputArray[count].getText()).d
oubleValue());
            for(int count=0;count<numInputs;count++)

   inputNeuron[count].setOutput(Double.valueOf(inputArray[numNeurons+count
].getText()).doubleValue());
            repaint();
            return true;
         }
         else if(o=="Update Weights")
         {
            weightUpdateFlag=true;
            repaint();
            return true;
         }
         else if(o=="Random Weights")
         {
            randomizeWeights();
            weightUpdateFlag = true;
         }
         else if(o=="Update Time")
         {
            Temperon.updateTime();
//          System.out.println("System time is: "+Temperon.getTime());
            timeField.setText(String.valueOf(Temperon.getTime()));
         }
         else if(o=="GO!")
         {
            goHandler();
                              flushAll();
         }
         else if(o=="Load Weights...")
         {
            loadHandler();
         }
         else if(o=="Save Weights")
         {
            saveHandler();
         }
      }
      else if (e.target instanceof Checkbox)
      {
         return false;
         // do nothing
      }
      else // someone updated a weight or something
      {
         weightUpdateFlag = true; // they now need to be updated...
         return true;
      }
      return false;
        }

   private void loadHandler()
```

```
   {
      DataInputStream inst = new DataInputStream(System.in);
      System.out.println("Which weight data table? ");
      String s = null;
      try{s = inst.readLine();}catch(IOException e){}
      File fn = null;
      try{fn = new File(WEIGHT_FILE_NAME+s+".dat");}
      catch(NullPointerException e){System.err.println("loadHandler:
"+e);}
      getWeights(fn);

   }

   private void saveHandler()
   {
      dumpWeights(checkFileName(WEIGHT_FILE_NAME,fileCounter));
   }

   public void updateWeights()
   {
      System.out.println("Updating weights");
      for(int outNeuron=0;outNeuron<numNeurons;outNeuron++)

            for(int inNeuron=0;inNeuron<numNeurons+numInputs;inNeuron++)

   fbNeuron[outNeuron].setWeightAt(inNeuron,Double.valueOf(weightMatrix[ou
tNeuron][inNeuron].getText()).doubleValue());

   }

   public void randomizeWeights()
   {
      System.out.println("Randomizing weights");
      for(int outNeuron=0;outNeuron<numNeurons;outNeuron++)

         for(int inNeuron=0;inNeuron<numNeurons+numInputs;inNeuron++)

   weightMatrix[outNeuron][inNeuron].setText(String.valueOf(2d*Math.random
()-1d));
   }

   public void dumpWeights(File fn)
   {
      if(weightUpdateFlag) updateWeights();

      FileOutputStream fileFile;
      DataOutputStream weightFile;
      top: {
      try
      {
         fileFile=new FileOutputStream(fn);
      }
      catch (IOException e)
      {
         System.err.println("Error opening data file: "+fn+" for write:
"+e);
         break top;
      }
      weightFile = new DataOutputStream(fileFile);

      int count=0;
```

```
      for(int outNeuron=0;outNeuron<numNeurons;outNeuron++)

         for(int inNeuron=0;inNeuron<numNeurons+numInputs;inNeuron++)
         {
            double current = fbNeuron[outNeuron].getWeightAt(inNeuron);
            try
            {
               weightFile.writeDouble(current);
               count++;
            }
            catch(IOException e)
            {
               System.err.println("Error dumping weights!: "+e);
               break top;
            }
         }
      try{
      weightFile.flush();weightFile.close();
      fileFile.flush();fileFile.close();
      }
      catch(IOException e){}
      System.err.println("Dumped "+count+" weights to file: "+fn);
      }
   }

   public void repaintWeights()
   {
      double epsilon=0.00001;
      double previous=0;
      double current=0;
//    System.out.println("Repainting weights");
      for(int outNeuron=0;outNeuron<numNeurons;outNeuron++)

         for(int inNeuron=0;inNeuron<numNeurons+numInputs;inNeuron++)
         {
            previous =
Double.valueOf(weightMatrix[outNeuron][inNeuron].getText()).doubleValue();
            current = fbNeuron[outNeuron].getWeightAt(inNeuron);

   weightMatrix[outNeuron][inNeuron].setText(String.valueOf(current));
            if (previous>current+epsilon)

   weightMatrix[outNeuron][inNeuron].setBackground(Color.green);
            else if (previous<current-epsilon)

   weightMatrix[outNeuron][inNeuron].setBackground(Color.blue);
            else

   weightMatrix[outNeuron][inNeuron].setBackground(Color.white);
         }

   }

   public void getWeights(File fn)
   {
      FileInputStream fileFile;
      DataInputStream weightFile;
      double current =0;
      top: {
      try
```

```
       {
          fileFile=new FileInputStream(fn);
       }
       catch (IOException e)
       {
          System.err.println("Error opening data file: "+fn+" for write:
"+e);
          break top;
       }
       weightFile = new DataInputStream(fileFile);

       int count=0;

       for(int outNeuron=0;outNeuron<numNeurons;outNeuron++)

          for(int inNeuron=0;inNeuron<numNeurons+numInputs;inNeuron++)
          {
              try
              {

                 current = weightFile.readDouble();
                 System.err.println("Read: "+current);
                 count++;
              }
              catch(IOException e)
              {
                 System.err.println("Error reading weights!: "+e);
                 break top;
              }
              fbNeuron[outNeuron].setWeightAt(inNeuron,current);
          }
       try{
       weightFile.close();
       fileFile.close();
       }
       catch(IOException e){}
       System.err.println("Read "+count+" weights from file: "+fn);
       }
       repaintWeights();
   }


   public long convertOutputs() // converts all the outputs from binary to
decimal
   { // note: works only as long as all the outputs are binary
       long temp=0;
       for(int count=0;count<numNeurons;count++)
       {
          temp+= (long)Math.pow(2,count) * (int)netOutput[count]; //
updated
       }
       return temp;
   }

A.     }
```

7. Bibliography

1. Beale, R and Jackson, T. “Neural Computing: An Introduction”. Department of Computer Science, University of York. IOP Publishing: 1990. Briston, England. ISBN 0-85274-263-2

2. Chester, M. “Neural Networks. A Tutorial” PTR Prentice-Hall. New Jersey: 1993. ISBN 0-13-368903-4

3. Guyton, Arthur C. “Basic Neuroscience” W.B. Saunders Co. Philadelphia: 1997. ISBN 0-7216-2061-2

4. Kandel, E., Schwartz, J., Jessel, T. “Principles of Neural Science, 3ed” Center for Neurobiology and Behavior. Columbia University. Appleton & Lange. Connecticut: 1991. ISBN 0-8385-8034-3

5. Kung, S.Y. “Digital Neural Networks” Department of EE. Princeton University. PRT Prentice-Hall. New Jersey: 1993. ISBN 0-13-612326-0

6. Llinas, R. ed. “The Biology of the Brain: From Neurons to Networks” Readings from Scientific American magazine. W.H. Freeman and Co. New York: 1988. ISBN 0-7167-2037-X