Pre-course Work for CS648 — Neural Networks

Jonathan Shapiro

1 Introduction

The goal of the pre-course work is to get you thinking about neural network models and to get you familar with the laboratory software. To this end, you are to do a lab on one of the simplest neural network models: the Hopfield model.

2 Introduction to the Standard Models

Most neural network models are based around a standard idealization of neurons. Read about this. My summary is Appendix A. Descriptions can also be found in early chapters of many textbooks on neural networks. Recommended textbooks are given below.

There are some on-line neural network introductions, which you could look at as alternatives. They are at

- `http://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/cs11/report.html#Contents`; and
- `http://www.shef.ac.uk/psychology/gurney/notes/contents.html`.

You may read one of these instead.

3 Introduction to the Laboratory Software — Matlab

The labs will use a software package called Matlab. If you do not already know Matlab, gain some familiarity with it. Look at Appendix B to get started.

4 The Hopfield Model Lab

The Hopfield model lab follows this document. I want you to do as much of that lab as you can. You will require a laboratory notebook in which to record your results of all of your labs. You need not write up the results of this lab, but on the first day you will need to show in your lab notebook that you have done it.

You may work in groups, but each person must keep their own record of what was done and what the results were. The last part (on image restoration) is the one part which you may have trouble seeing how to do. Be sure to do the other parts, and do your best on the last part. It is most important that you get some familiarity with Matlab, so you can do the practical work which is integrated into the course.

4.1 Textbooks

There are zillions of books on neural networks, and I am sure that most of them are pretty much the same. These are ones whose content I know.

Elementary Treatments of Neural Networks

- Neural Computing, Beale and Jackson, 1990

  Perhaps the simplest book on neural networks which covers the standard models. No applications and not much detail, but very easy to read.

- Artificial Neural Networks, D. W. Patterson, 1996.

  A good overview of artificial neural network models, mostly as they relate to statistical pattern recognition applications.

- The Essence of Neural Networks, Robert Callan, 1999.

  As the title suggests, this is a primer on neural networks.

More Advanced Treatments of Neural Networks

- Introduction to the Theory of Neural Computation, J. Hertz, A. Krogh, R. Palmer, 1991.

  Until recently, the best book on the theory of neural networks. The treatment is at an advanced level. Dated, but still excellent. Covers the theory of models which will be covered in this course.

- Neural Networks, A comprehensive Foundation, 2nd Edition, Simon Haykin, 1998.

  A very comprehensive and modern book on neural networks. I find it highly abstract.

- Neural Networks for Pattern Recognition, Christopher Bishop, 1996

  A excellent book on the modern applied aspects of neural networks in a statistical and Bayesian context. Advanced in treatment. However, only covers supervised learning neural networks.

- A Guide to Neural Computing Applications, L. Tarassenko, 1998.

  Works through the application of MLPs and similar approaches in some detail.

A Appendix A — Connectionist Models

Connection models are based on a standard set of idealizations of neural networks. These are described here.

Neurons

Neurons are the important functional cells as far as mental processes are concern. Neurons have special properties which differentiate them from other cells. First, they spike. When one measures the potential of the cell, one finds peaks of about 1 millisecond. These can come in rapid succession, separated only by a few 10s of milliseconds, or can be few and far between. These are electrochemical in nature, involving the exchange of ions. Figure 1 shows idealized picture of the neural spikes.

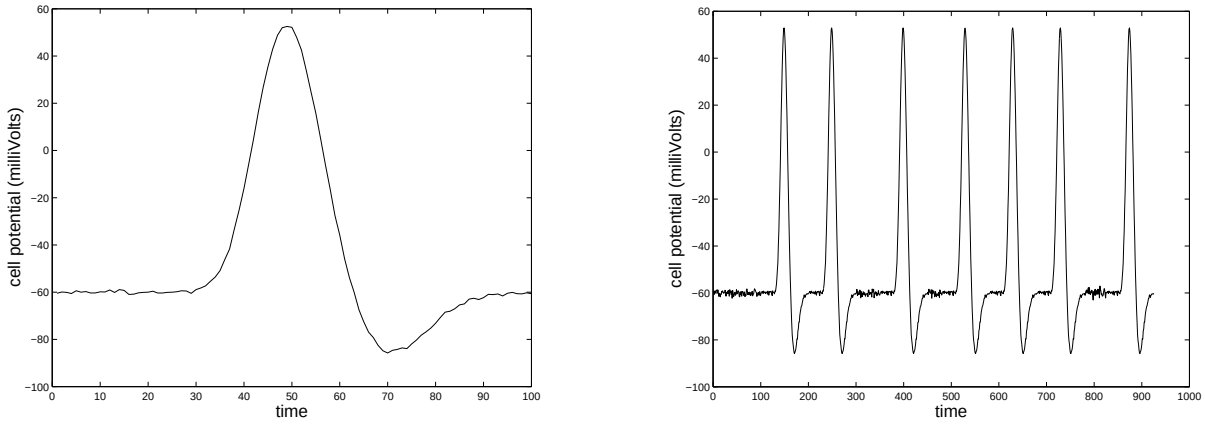

Figure 1: Left) An idealized spike. Right) A sequence of spikes.

Neurons have special structures which allow them to interact with other neurons. The axon is a elongation through which the spike can travel towards other neurons. At the ends of the axon are synapses, which are points of close proximity with other neurons. Spikes travel down the axon to the synapses, where they result in the release of neurotransmitters which are picked up by other neurons. Dendrites are tree-like structures which receive the neurotransmitters from spiking neurons. Figure 2 shows the parts of a generic neuron.

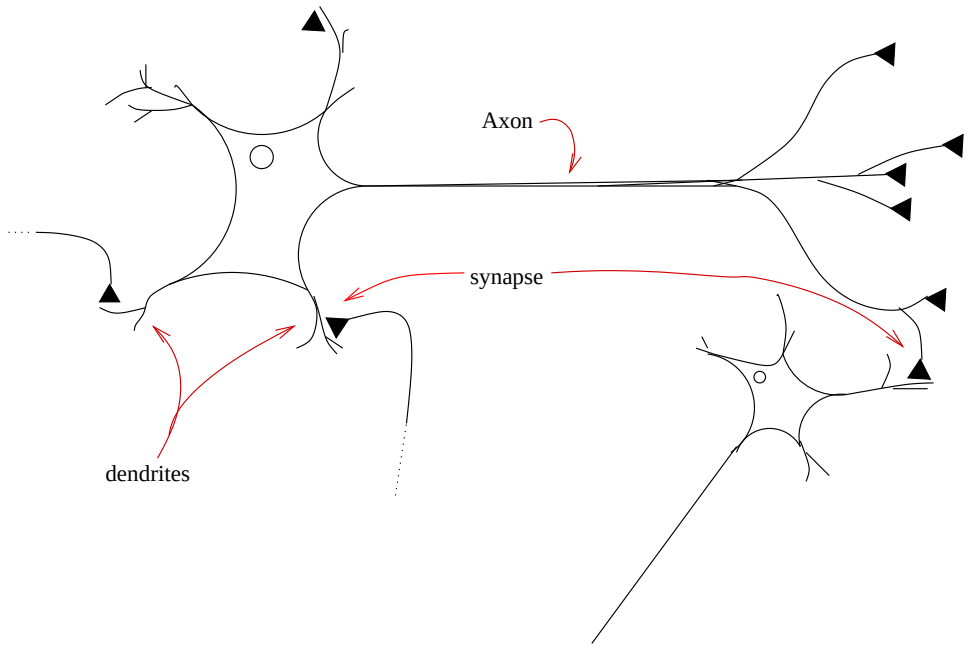

Figure 2: An idealized neuron

The spiking of a neuron is influenced by the spiking of other neurons which synapse with it. A particular neuron may, through its spiking, inhibit the spiking of the other neurons which it is connected to, or it may excite them. Usually, a particular type of neuron will either always be excitatory or always inhibitory. The strength of this interaction between neurons can vary from synapse to synapse. The strength of the effect of one neuron on another can also change over time, perhaps by the firing of the presynaptic neuron or by coincidental firings of pre and postsynaptic neurons.

The number of neurons in the human brain is often stated as $10^{10} - 10^{12}$, i.e. between 10 and 1000 billion. The number of neurons which a typical neuron is connected to is said to be 1000 – 10000, although in fact this number varies considerably in different regions of the brain. This level of connectivity is much greater than that which can be achieved in electronic systems, yet neurons are very slow devices when compared to transistors. A typical refractory period (the minimum time between spikes) for a neuron is around 10 milliseconds. Thus, a complex image recognition task which we can do in 0.1 of a second may have taken only 100 levels of processing. This numerology has been taken as evidence that massively parallel processing must be important in the brain.

A.1 A Standard Model

Assuming that neurons exchange information, a crucial question is how is that information encoded. An assumption which is almost universally made is that the message is encoded in the firing rate. Thus, the message that a neuron sends may be taken as binary, the two states being: "firing at the highest rate" and "not firing" (i.e. firing at the background rate). Or, the message may be taken as an analog signal representing the value of the firing rate.

The interaction between two neurons is represented as a weight value which is a real number. We will denote the effect on neuron i due to neuron j as w_{ij} The sign of this weight is positive if the effect is excitatory, and negative if it is inhibitory; the magnitude of the weight represents the strength of the interaction. In what follows, we consider a single neuron with a set of weights w_i.

In the standard connectionist model, all of the temporal aspects of the flow of information within a neuron is ignored. It is assumed that all of the signals from the incoming neurons (the input) arrive simultaneously. The neuron produces an output which is a function of the weighted sum of the inputs from the incoming neurons,

$$y = f\left(\sum_i w_i x_i\right). \tag{1}$$

Here y is the output of the neuron (representing its rate). The inputs to the neuron are denoted x_i; these could be outputs of the neurons feeding into this neuron or could come from sensory cells. The weights, w_i are the interaction strengths. The output of the neuron is a function of the weighted sum of its inputs. A connectionist neuron is depicted as shown in figure 3.

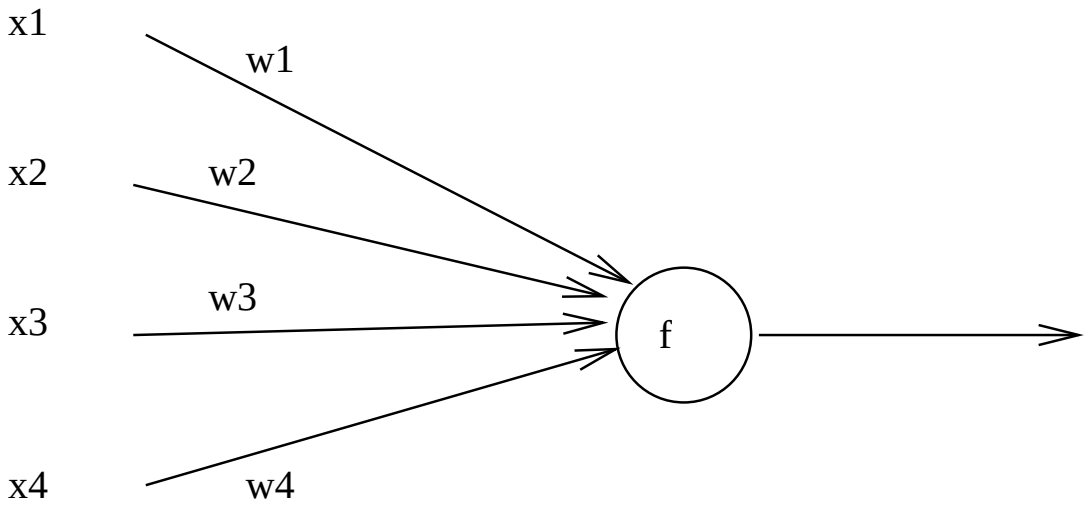

Figure 3: The standard connectionist model of a neuron. The connection strengths between input x_i and the node is determined by a weight w_i.

The choice of the "transfer function" $f(\cdot)$ varies in different models. For the binary representation, a common

choice is to state that the output is 1 if the weighted sum of input exceeds some threshold, and is 0 otherwise,

$$y = \begin{cases} 1; & \text{if } \sum_i w_i x_i > \Theta \\ 0; & \text{otherwise} \end{cases} \tag{2}$$

Transfer functions of this form are called *linear threshold functions* or hard limiting functions (hardlim in Matlab).

An alternative way of writing equation 2 is found by moving the threshold to the other side of the equation, whence it is called a *bias*. I.e. we could write the equation as

$$y = \begin{cases} 1; & \text{if } \sum_i w_i x_i + b > 0 \\ 0; & \text{otherwise} \end{cases} \tag{3}$$

where b is the bias (so $b = -\Theta$, obviously). This is just a notational convenience. Expressed this way, the bias can be treated as another weight to an input which always has the value 1.

Another common choice for the transfer function is a so-called sigmoid function. This is a smooth function which interpolates between the two values of the function above. This is called "logsig" in Matlab. A sigmoid function and a hardlimiting function are shown in figure 4.

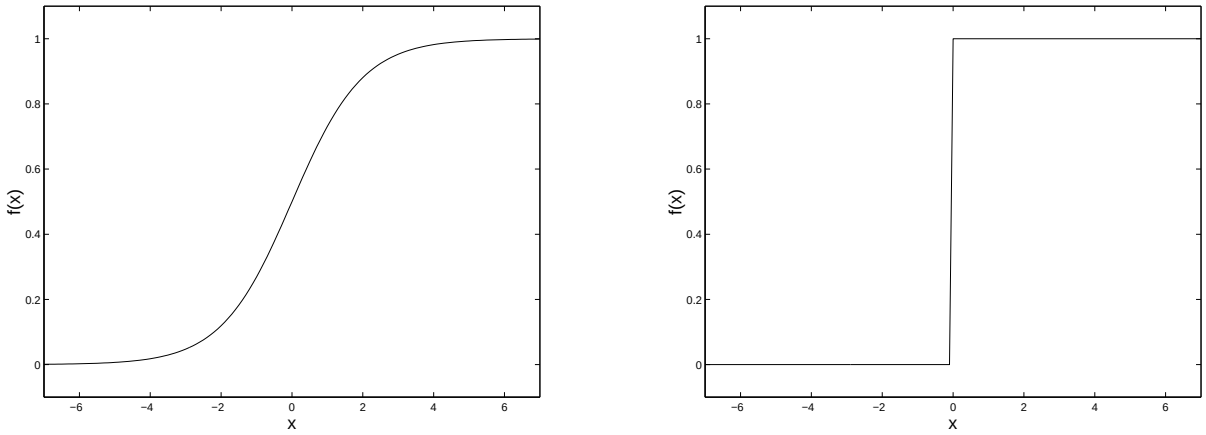

Figure 4: Left) A sigmoid function. Right) A hard-limiting function.

Computation and problem-solving is carried out in a neural network by the interaction between neurons and the sequence of outputs produced. Thus, the computation which a neural network carries out depends upon the values of the connection strengths. We might say that knowledge is stored in the weights. We next must consider how the weight values are determined.

One way that the weight values get set is through learning. An important simple learning rule comes from the principle of Hebb, that when two neurons are simultaneously active, the synaptic weight between them should be increased. Within this simple model, this is usually represented like this:

$$w_{ij} \to w_{ij} + \eta x_j y_i. \tag{4}$$

Here η is a learning rate parameter. So, if neurons x_j and y_i are simultaneously active, the weight between them will increase.

Notice that using this rule, weights only increase unless some other mechanism is introduced, such as weight decay over time. It is also possible to interpret Hebb to mean that if two neurons have the same activity, the weight between them should increase, whereas if they have a negative correlation it should decrease. This can be implemented by using the same rule, equation 8, but using as the two values for the neurons 1 and -1 (instead of 0). This is called a symmetric representation (as opposed to a 0-1 representation). The transfer function in Matlab which implements this is called "hardlims (the final "s" stands for symmetric).

Hebbian learning can be supervised (which means learning with a teacher). In this case, there is an additional signal which forces the output to be the desired output during the learning phase. The learning rule is

$$w_{ij} \to w_{ij} + \eta x_i d_j. \tag{5}$$

Here, d_j is the desired output (which we could imagine that the output is forced to be during the learning phase). During recall, when there is no teaching signal, the output simply follows the rule in equation 1.

It is possible to allow the bias to learn as well. The usual way to do this is to view the bias as a weight to an input which is always 1. The learning rule is, therefore,

$$b \to b + \eta y. \tag{6}$$

This makes it easier for the node to produce an output of 1 every time it learns an output of 1 (and if using the symmetric representation, harder to produce an output of 1 every time it learns an output of -1).

In general, a neural network is a network, or graph, of interconnected nodes. The structure of the network is called the *architecture* of the network. There are two commonly used classes of network architectures. Feedforward networks contain no feedback; if node A influences the output of node B, node B cannot influence the output of node A. Layered feedforward networks are very commonly used. A layered feedforward network is shown below. It is obviously easy to compute the output of a feedforward network. Start at the input end and compute the nodes layer by layer. Figure 5 shows a feedforward network.

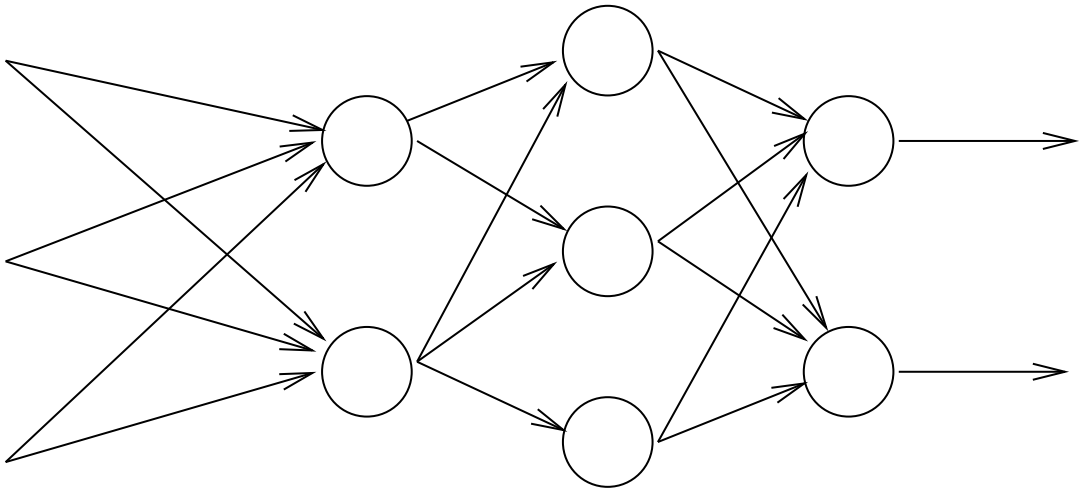

Figure 5: A feedforward network.

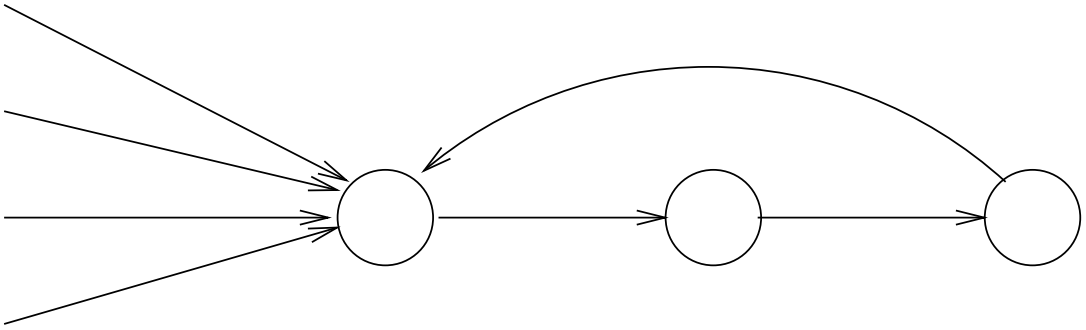

Figure 6: A recurrent network

Networks with feedback are called *recurrent networks*. The figure below shows a simple recurrent network. Computing the states of recurrent networks is complex, because they can undergo very complicated dynamics. Figure 6 shows a simple recurrent network.

A.2 The Hopfield Model

The Hopfield model (in its simplest manifestation) is a completely connected recurrent network of $+1, -1$ binary neurons. Hopfield's ideas were important for several reasons. First, he showed how a simple model could act as a content-addressable memory and show tolerance to noise in the input. Second, he showed how the dynamics of interacting neurons could be understood by defining a Lyapunov or energy function (see below). Finally, he showed how to build actual devices which could solve hard problems via analog collective computation.

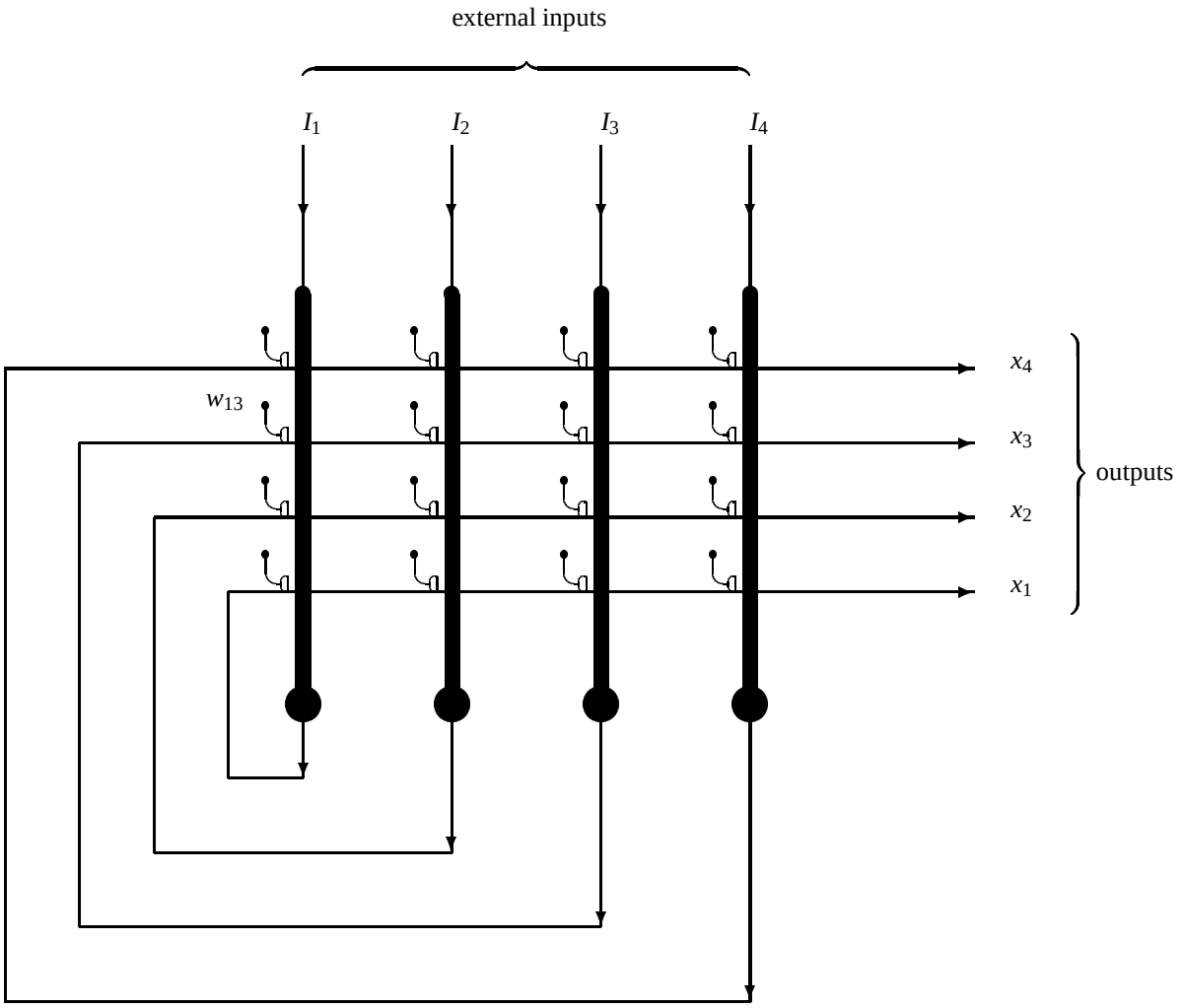

Figure 7: The Architecture of the Hopfield Model. This shows the external inputs, I_i, the outputs of the nodes, x_i, and that each node is connected to all others by a weight w_{ij} which connects node j to node i

The architecture of the Hopfield network is shown in figure 7. The network is completely connected; every node is connected to all other nodes by a set of weights w_{ij}. The state of each node can be one of two values, -1 and 1.

The input to a node i is the sum of an external input and the inputs from the other nodes,

$$\text{total input on node i} \equiv h_i = \sum_j w_{ij} x_j + I_i.$$

The nodes are updated one at a time (asynchronously) in arbitrary or random order. When the node i is updated, it is changed according to the rule,

$$x_i = \begin{cases} 1; & \text{if } \sum_j w_{ij} x_j + I_i > \Theta_i \\ -1; & \text{if } \sum_j w_{ij} x_j + I_i \leq \Theta_i \end{cases} \tag{7}$$

Here Θ_i is the threshold for node i. Note that this system has feedback — a change in the output of node A can cause node B to change which can cause node A to change again at some later time. Because of this, the dynamics of the network can be complicated. It can cycle forever, for example.

A.3 Dynamics of the Hopfield Model

The dynamics of the network is the repeated application of equation (7), one node at a time with the weights fixed. For arbitrary weights, this system could have complicated dynamics and might never settle down to stable states. Hopfield had the insight that if the following conditions were met,

1. $w_{ii} = 0$,
2. $w_{ij} = w_{ji}$, (Unlikely to hold in real brains).
3. Only one node updated at a time,

the network would always settle down to a fixed point.

The first application of the model is as an auto-associative memory. The purpose of such a model is to store a set of memories such that if one inputs a corrupted version of one of the memories, or a partial version of one of the memories, the whole or uncorrupted memory is produced. This is an example of a content-addressible memory, because part of the memory itself is used to access the memory.

Thus a method is required to find a set of weights which makes the stored patterns stable. That is, if the memory is set in one of the stored patterns it will stay in that state. If the input is a distorted version of one of the stored patterns, the memory will evolve from that state to the stored pattern.

We will consider only a Hebbian learning rule. To store patterns $\{\vec{s}_1, \vec{s}_2, ..., \vec{s}_p\}$ set,

$$w_{ij} = \sum_{k=1}^{p} (\vec{s}_k)_i (\vec{s}_k)_j \tag{8}$$

or in matrix notation for weight matrix $\mathbf{w}$,

$$\mathbf{w} = \sum_k \vec{s}_k \vec{s}_k^T. \tag{9}$$

The above equations do not apply to w_{ii} which must be set to zero for all i.

Retrieval occurs by starting the network in some state $\vec{X}$, say. This is done by producing a strong external input $\vec{I} = \vec{X}$. Then the external input is removed or reduced and the network dynamics, equation (7) is allowed to run.

B Appendix B: Getting Started With Matlab

Matlab is a software package which was developed for numerical analysis involving matrices ("Matlab" as in **mat**rix **lab**oratory). It is an interpreted language, which means that commands are run as they are entered in, without being compiled first. It also means that most of the commands, except the most basic ones, are defined in files which are written in this interpreted language. So, you can read these files, and add to the language by making similar files. It is also possible to buy a student version if you want one on your home computer.

Matlab will runs on the PCs under Linux and NT. The description below about how to start up Matlab is for Linux.

1. Copy /opt/info/courses/NeuralNets/startup.m to your working directory (which ever directory you are going to do your lab work from). This file tells Matlab where to look for the commands which I have devised for this course. In case you don't know UNIX, the command

   ```
   cp /opt/info/courses/NeuralNets/startup.m startup.m
   ```

2. From your working directory, start Matlab simply by typing

   ```
   /opt/matlab6/bin/matlab
   ```

   Of course you can add `/opt/matlab6/bin` to your search path; then Matlab would be invoked by simply typing

   ```
   matlab
   ```

There are on-line help menus and demos. There is a particularly useful introductory document on line. Click on "Help" and follow the links to the "Getting Started" document.

This document is not about neural networks at all. It is simply about the general aspects of Matlab. Read as much or little as you find useful. I would particularly recommend the sections on "Matrices and Magic Squares" and "Working with Matrices" to learn how to to represent and create matrices; the sections on "Expressions", "Linear Algebra", and "Arrays" to learn how arithmetic works in Matlab; and the section on the "Command Window" to learn how to enter and edit within the Matlab window. You can certainly skip the stuff about graphics; look at it later if you want to make a graph. Likewise, be aware that there are if statements and looping commands described in the Flow Control section, which you can look at later if you need them. Alternatively, you can look at the matlab tutorial I devised for undergraduates, at `http://www.cs.man.ac.uk/~jls/CS2411/matlab_tutorial.pdf`

The actual neural network commands are described in the Matlab Neural Networks Toolbox User's Guide. This runs to 800 pages (give or take) in the current manifestation, it is also available as an On-line Manual from the Helpdesk. you started.

B.0.1 Representing a single neuron

Here is how the standard model is simulated in the Matlab Neural Network Toolbox[1] As an example, consider a neuron with three inputs (x_1, x_2, and x_3), weights denoted w_1, w_2, and w_3, a bias b and output y.

Input patterns: An input pattern is represented as a column vector. Thus, if the neuron had three inputs and we wanted them to be 1 1 and 0 respectively, this would be represented as

```
p = [1;
     1;
     0]
```

(the carriage returns are optional).

Connection Weights: With a single node, the weights are a row vector. For example, a three input neuron with weights 0, 1, 5 would be represented as

[1] A warning for students who took the Machine Learning course. The Matlab Neural Network toolbox uses a different representation from Netlab.

```
w=[0 1 5]
```

If there were many neurons receiving the same inputs (a layer of neurons), the weights would be expressed as an *m* row and *n* column matrix, where *m* is the number of neurons and *n* is the number of inputs. I.e. w_{ij} is viewed as the components of a matrix.

Weighted sum of inputs: The weighted sum of inputs is simply matrix multiplication. In Matlab "*" means matrix multiplication. In the example defined above,

```
w*p
```

is $\sum w_i x_i$. (Note: in Matlab all operators take their matrix meaning. So "*" means matrix multiplication. If you put a dot in front of the operator, it means the ordinary form of multiplication acting element by element on the two operands. See the subsection on "Arrays" under the section on "More about Matrices and Arrays" in the Introductory Document on Matlab.)

Bias: For a single neuron the bias is simply a number.

```
b = -2.5
```

If there were many neurons represented by a *m* by *n* weight matrix, bias would be a *m* by 1 matrix, i.e. a column vector.

Transfer function: Matlab has a number of built-in transfer functions. Some graphs are shown in the figure taken from the manual. We will consider the 0-1 hard-limiting function called "hardlim". It can be used like this,

```
y = hardlim(w*p,b)
```

which will produce a 1 if the weighted sum of input plus the bias exceeds 0, and will return 0 otherwise. This can also be expressed as

```
y = hardlim(w*p+b)
```

but this second form will not work if a batch of patterns are used.

Batching: Often we wish to show the neuron (or any neural network) not one pattern, but several. This can be done by creating a matrix; each column of the matrix is a different pattern. For example, suppose we want to know the output of a three input neuron with weights $(0,1,5)$ and bias -2.5, for all input patterns of 0 and 1. Define,

```
p = [0 1 0 0 1 1 0 1;
     0 0 1 0 1 0 1 1;
     0 0 0 1 0 1 1 1]
```

There are eight patterns here. To get the output for the batch of patterns, enter,

```
y = hardlim(w*p,b)
```

the output will be

```
y = 0 0 0 1 0 1 1 1
```

which is the output for each of the patterns.

Hebbian Learning: Hebbian learning is carried out with a command called "learnh". The form is

```
w = w + learnh(p,a,1)
```

This command produces the change in weights due to Hebbian learning, hence the equation adds it to the existing weights. Here p is a pattern or batch of patterns, a is the desired output or row vector of outputs, one for each pattern in the batch, and the final argument is the learning rate, denoted as η in the equation 8.

Matlab commands specific to the Hopfield Model are found in the document describing the lab.