Chapter 1

What is Hopfield?

1.1 Introduction

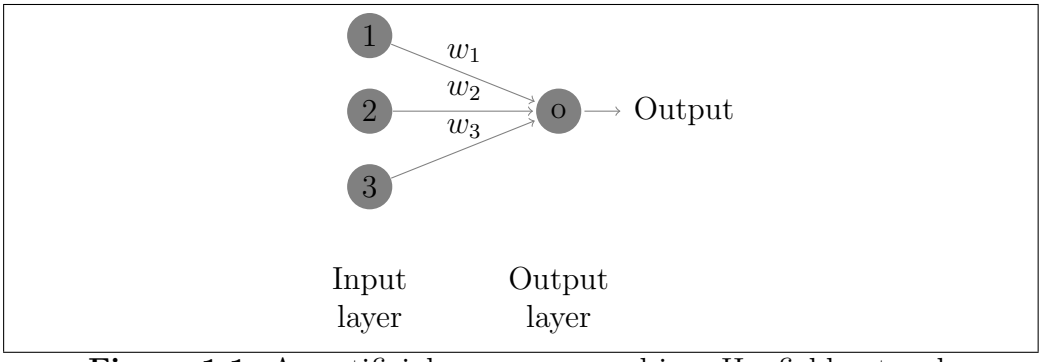

Figure 1.1: An artificial neuron as used in a Hopfield network

Hopfield networks are constructed from artificial neurons Figure 1.1. These artificial neurons have N inputs. With each input i there is a weight w_i associated. They also have an output. The state of the output is maintained, until the neuron is updated. Updating the neuron entails the following operations:

- The value of each input, x_i is determined and the weighted sum of all inputs, $\sum_{i=1}^{n} w_i x_i$ is calculated.

- The output state of the neuron is set to $+1$ if the weighted input sum is larger or equal to 0. It is set to -1 if the weighted input sum is smaller than 0.
- A neuron retains its output state until it is updated again.

Written as formula:

$$o = \begin{cases} 1: & \sum_i w_i x_i >= 0 \\ -1: & \sum_i w_i x_i < 0 \end{cases}$$

A Hopfield network is a network of N such artificial neurons, which are fully connected. The connection weight from neuron j to neuron i is given by a number w_{ij}. The collection of all such numbers is represented by the weight matrix W, whose components are w_{ij}.

Now given the weight matrix and the updating rule for neurons the dynamics of the network is defined if we tell in which order we update the neurons. There are two ways of updating them:

- **Asynchronous**: one picks one neuron, calculates the weighted input sum and updates immediately. This can be done in a fixed order, or neurons can be picked at random, which is called asynchronous random updating.
- **Synchronous**: the weighted input sums of all neurons are calculated without updating the neurons. Then all neurons are set to their new value, according to the value of their weighted input sum. The lecture slides contain an explicit example of synchronous updating.

1.2 Use of the Hopfield network

The way in which the Hopfield network is used is as follows. A pattern is entered in the network by setting all nodes to a specific value, or by setting only part of the nodes. The network is then subject to a number of iterations using asynchronous or synchronous updating. This is stopped after a while. The network neurons are then read out to see which pattern is in the network. The idea behind the Hopfield network is that patterns are stored in the weight matrix. The input must contain part of these patterns. The dynamics of the network then retrieve the patterns stored in the weight

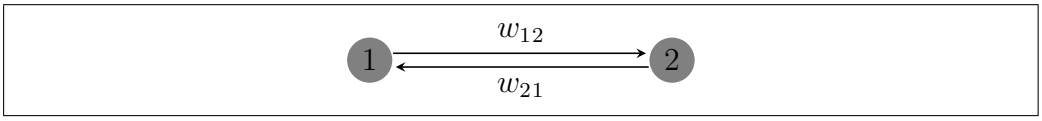

Figure 1.3: There are just two options for the weight matrix: $w_{i,j} = 1$ or $w_{i,j}$ = -1. If the weight is 1, there are two stable states under synchronous updating {+1, +1}, or {-1, -1}. For a weight of -1, the stable states will be {-1, +1} or {+1, -1} depending on initial conditions. Under synchrononous updating the states are oscillatory.

matrix. This is called **Content Addressable Memory (CAM)**. The network can also be used for auto-association. The patterns that are stored in the network are divided in two parts: **cue** and **association**. By entering the cue into the network, the entire pattern, which is stored in the weight matrix, is retrieved. In this way the network restores the association that belongs to a given cue. The stage is now almost set for the Hopfield network, we must only decide how we determine the weight matrix. We will do that in the next section, but in general we always impose two conditions on the weight matrix:

- Symmetry : $w_{ij} = w_{ij}$
- No self connections: $w_{ii} = 0$

It turns out that we can guarantee that the network converges under asynchronous updating when we use these conditions

1.3 Training the network

1.3.1 A Simple Example

Consider the two nodes in Fig 1.3 Depending on which values they contain initially they will reach end states +1, +1 or -1,-1 under asynchronous updating.
This simple example illustrates two important aspects of the Hopfield network: the steady final state is determined by the value of the weight and the network is sign-blind; it depends on the initial conditions which final state will be reached, {+1,+1} or {-1, -1} Figure 1.3.

1.4 Setting the weight matrix

1.4.1 A single pattern

Consider two neurons. If the weight between them is positive, then they will tend to drive each other in the same direction: this is clear from the two-neuron network in the example. It is also true in larger networks: suppose we have neuron i connected to neuron j with a weight of $+1$, then the contribution of neuron i to the weighted input sum of neuron j is **positive** if $x_i = 1$ and negative if $x = -1$. In other words neuron, i tries to drive neuron j to the same value as it has currently. If the connection weights between them is negative, however, neuron i will try to drive neuron j to the opposite value! This inspires the choice of the Hebb rule: given a network of N nodes and faced with a pattern $x = (x_1, ..., x_N)$ that we want to store in the network, we chose the values of the weight matrix as follows:

$$w_{i,j} = x_i x_j$$

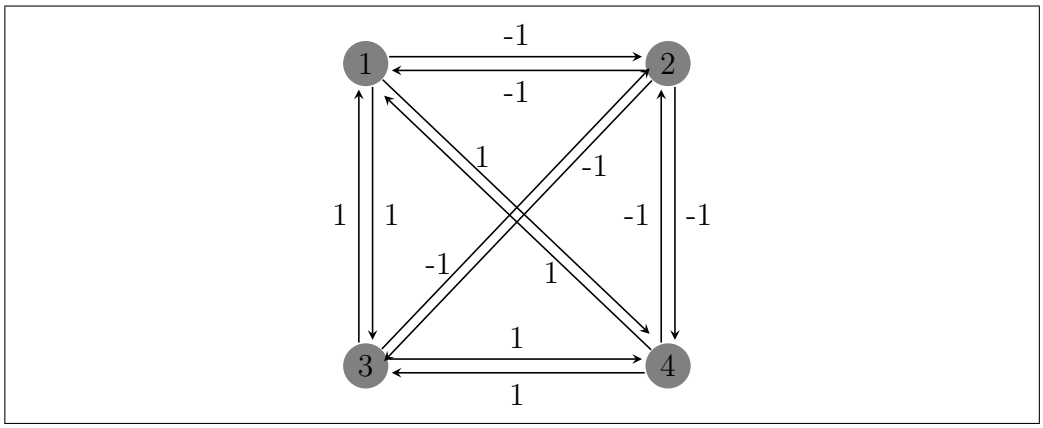

Figure 1.4: Single Layer Hopfield network

To see how this works out in a network of four nodes, consider Figure 1.3. Note that weights are optimal for this example in the sense that if the pattern (1, -1, 1, 1) is present in the network, each input node receives the maximal or minimal (in the case of neuron 2) input some possible! We can actually prove this: suppose the pattern $\vec{x}$, which has been used to train the network, is present in the network and we update neuron i, what will happen? The weighted input sum of a node is often denoted by

$h_i = \sum_j w_{i,j} x_j$, which is called the local field. If w_{ij} was chosen to store this very same pattern, we can calculate h_i:

$$h_i = \sum_j w_{i,j} x_i x_j = \sum_{j=1}^{3} = x_i x_j x_j = x_i + x_i + x_i = 3x_i$$

This is true for all nodes i. So all other three nodes (self connections are forbidden!) in the network give a summed contribution of $3x_i$. This means that x_i will never change value, the pattern is stable!

$$h_i = \sum_j w_{i,j} v_j = -\sum_j w_{i,j} x_j = -\sum_j x_i x_j x_j = -\sum_{j=1}^{3} x_i = 3x_i = 3v_i$$

Again, this pattern is maximally stable: the Hopfield network is sign-blind.

1.4.2 Weight Determination

How to determine weight matrix for multiple patters? You can go about this in the following way.

1.4.2.1 Binary to Bipolar Mapping

By replacing each 0 in a binary string with a -1, you get the corresponding bipolar string.

$$f(x) = 2x - 1$$

For inverse mapping, which turns a bipolar string into a binary string, you use the following function:

$$f(x) = (x + 1)/2$$

1.4.2.2 Patterns Contribution to Weight

We work with the bipolar versions of the input patterns. We take each pattern to be recalled, one at a time, and determine its contribution to the weight matrix of the network. The contribution of each pattern is itself a matrix. The size of such a matrix is the same as the weight matrix of the network. Then add these contributions, in the way matrices are added, and

you end up with the weight matrix for the network, which is also referred to as the correlation matrix. Let us find the contribution of the pattern $A = (1, 0, 1, 0)$: First, we notice that the binary to bipolar mapping of $A = (1, 0, 1, 0)$ gives the vector $(1, 1, 1, 1)$. Then we take the transpose, and multiply, the way matrices are multiplied, and we see the following:

$$\begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix} * \begin{bmatrix} 1 & -1 & 1 & -1 \end{bmatrix} = \begin{bmatrix} 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 \\ 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 \end{bmatrix}$$

Now subtract 1 from each element in the main diagonal (that runs from top left to bottom right). This operation gives the same result as subtracting the identity matrix from the given matrix, obtaining 0s in the main diagonal. The resulting matrix, which is given next, is the contribution of the pattern $(1, 0, 1, 0)$ to the weight matrix.

$$W_1 = \begin{bmatrix} 0 & -1 & 1 & -1 \\ -1 & 0 & -1 & 1 \\ 1 & -1 & 0 & -1 \\ -1 & 1 & -1 & 0 \end{bmatrix}$$

Similarly, we can calculate the contribution from the pattern $B = (0, 1, 0, 1)$ its bipolar version $B = (-1, 1, -1, 1)$ by verifying that pattern B's contribution is the same matrix as pattern A's contribution.

$$\begin{bmatrix} -1 \\ 1 \\ -1 \\ 1 \end{bmatrix} * \begin{bmatrix} -1 & 1 & -1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 \\ 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 \end{bmatrix}$$

$$W_2 = \begin{bmatrix} 0 & -1 & 1 & -1 \\ -1 & 0 & -1 & 1 \\ 1 & -1 & 0 & -1 \\ -1 & 1 & -1 & 0 \end{bmatrix}$$

$$W_1 + W_2 = \begin{bmatrix} 0 & -2 & 2 & -2 \\ -2 & 0 & -2 & 2 \\ 1 & -2 & 0 & -2 \\ -2 & 1 & -2 & 0 \end{bmatrix}$$

We can now optionally apply an arbitrary scalar multiplier to all the entries of the matrix if you wish. This is shown in the 2.0.1.

1.4.2.3 Autoassociative Network

The Hopfield network just shown has the feature that the network associates an input pattern with itself in recall. This makes the network an **autoassociative** network. The patterns used for determining the proper weight matrix are also the ones that are **autoassociatively** recalled. These patterns are called the exemplars. A pattern other than an exemplar may or may not be recalled by the network. Of course, when you present the pattern 0000, it is stable, even though it is not an exemplar pattern.

1.4.2.4 Orthogonal Bit Patterns

You may be wondering how many patterns the network with four nodes is able to recall. Let us first consider how many different bit patterns are orthogonal to a given bit pattern. This question really refers to bit patterns in which at least one bit is equal to 1. A little reflection tells us that if two bit patterns are to be orthogonal, they cannot both have 1's in the same position, since the dot product would need to be 0. In other words, a bitwise logical AND operation of the two bit patterns has to result in a 0. This suggests the following. If a pattern P has k, less than 4, bit positions with 0 (and so $4-k$ bit positions with 1), and if pattern Q is to be orthogonal to P, then Q can have 0 or 1 in those k positions, but it must have only 0 in the rest $4-k$ positions. Since there are two choices for each of the k positions, there are 2^k possible patterns orthogonal to P. This number 2^k of patterns includes the pattern with all zeroes. So there really are $2^k - 1$ non-zero patterns orthogonal to P. Some of these $2^k - 1$ patterns are not orthogonal to each other. As an example, P can be the pattern (0 1 0 0), which has $k = 3$ positions with 0. There are $2^3 - 1 = 7$ nonzero patterns orthogonal to (0 1 0 0). Among these are patterns (1 0 1 0) and (1 0 0 1), which are not orthogonal to each other, since their dot product is 1 and not 0.

1.4.2.5 Network Nodes and Input Patterns

Since our network has four neurons in it, it also has four nodes in the directed graph that represents the network. These are laterally connected

because connections are established from node to node. They are lateral because the nodes are all in the same layer. We started with the patterns A = (1, 0, 1, 0) and B = (0, 1, 0, 1) as the exemplars. If we take any other nonzero pattern that is orthogonal to A, it will have a 1 in a position where B also has a 1. So the new pattern will not be orthogonal to B. Therefore, the orthogonal set of patterns that contains A and B can have only those two as its elements. If you remove B from the set, you can get (at most) two others to join A to form an orthogonal set. They are the patterns (0, 1, 0, 0) and (0, 0, 0, 1). If you follow the procedure described earlier to get the correlation matrix, you will get the following weight matrix:

$$W = \begin{bmatrix} 0 & -1 & 3 & -1 \\ -1 & 0 & -1 & -1 \\ 3 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 \end{bmatrix}$$

With this matrix, pattern A is recalled, but the zero pattern (0, 0, 0, 0) is obtained for the two patterns (0, 1, 0, 0) and (0, 0, 0, 1). Once the zero pattern is obtained, its own recall will be stable.

1.5 Energy

We can define an energy for each node:

$$E_i = -\frac{1}{2} h_i x_i$$

Note that the energy is positive if the sign of h_i and x_i is different! An update of node i will result in a sign change in this case because the local field h_i has a different sign. The updating will change the energy form a positive number to a negative number. This corresponds to the intuitive idea that stable states have a low energy. We can define an energy for the entire network:

$$E(\vec{x}) = \sum_i E_i = -\sum_{i,j} \frac{1}{2} h_i x_i = -\frac{1}{2} \sum_{i,j} w_{i,j} x_i x_j$$

Again, it is clear that for the pattern that has been used to train the weight matrix the energy is minimal. In this case:

$$E = -\frac{1}{2}\sum_{i,j} w_{i,j}x_i x_j = -\frac{1}{2}\sum_{i,j} x_i x_j x_j = -\frac{1}{2}\sum_{i,j} 1 = -\frac{1}{2}(N-1)^2$$

1.6 Stability of a single pattern

So far we have looked at the situation where the same patterns was in the network that was used to train the network. Now let us assume that another pattern is in the network. It is pattern $\vec{y}$ which is the same as pattern $\vec{x}$ except for three nodes. The network, as usual, has N nodes. No we are updating a node i of the network. What is the local field? It is given by:

$$h_i = \sum_j w_{i,j}y_j = \sum_j x_i x_j y_j$$

We can split this sum in 3 nodes, which have an opposite value of $\vec{x}$ and $N3$ nodes which have the same value:

$$h_i = \sum_{j=1}^{N-3} x_i x_j x_j + \sum_{j=1}^{3} x_i x_j - x_j = (N-3)x_i - 3x_i = (N-6)x_i$$

For $N > 6$ all the local fields point the direction of x_i, the pattern that was used to define the weight matrix!. So updating will result in no change (if the value of node i is equal to x_i) or an update (if the value of node i is equal to $-x_i$). So, the pattern that is stored into the network is stable: up to half of the nodes can be inverted and the network will still reproduce $\vec{x}$.

Chapter 2

C++ Code

Code : aja/example/ann/hopfield_network.cpp

2.0.1 Example A Simple Hopfield Network

A simple single layer Hopfield network is created with four neurons. We place, in this layer, four neurons, each connected to the rest, as shown in Figure 2.1. Some of the connections have a positive weight, and the rest have a negative weight. The network will be presented with two input patterns, one at a time, and it is supposed to recall them. The inputs would be binary patterns having in each component a 0 or 1. If two patterns of equal length are given and are treated as vectors, their dot product is obtained by first multiplying corresponding components together and then adding these products. Two vectors are said to be orthogonal, if their dot product is 0. The mathematics involved in computations done for neural networks include matrix multiplication, transpose of a matrix, and transpose of a vector. The inputs (which are stable, stored patterns) to be given should be orthogonal to one another.

The two patterns we want the network to recall are $A = (1, 0, 1, 0)$ and $B = (0, 1, 0, 1)$, which you can verify to be orthogonal. Recall that two vectors A and B are orthogonal if their dot product is equal to zero. This is true in this case since

$A_1B_1 + A_2B_2 + A_3B_3 + A_4B_4 = (1 \times 0 + 0 \times 1 + 1 \times 0 + 0 \times 1) = 0$

The following matrix W gives the weights on the connections in the network.

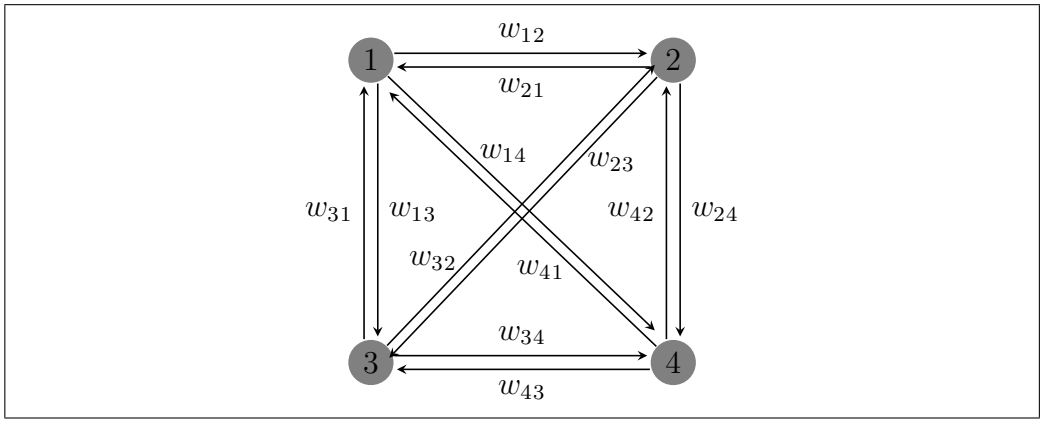

Figure 2.1: Single Layer Hopfield network

$$W = \begin{bmatrix} 0 & -3 & 3 & -3 \\ -3 & 0 & -3 & 3 \\ 3 & -3 & 0 & -3 \\ -3 & 3 & -3 & 0 \end{bmatrix}$$

We need a threshold function also, and we define it as follows. The threshold value θ is 0.

$$f(t) = \begin{cases} 1 & \text{if } t \geq \theta \\ 0 & \text{if } t < \theta \end{cases}$$

We have four neurons in the only layer in this network. We need to compute the activation of each neuron as the weighted sum of its inputs. The activation at the first node is the dot product of the input vector and the first column of the weight matrix (0 -3 3 -3). We get the activation at the other nodes similarly. The output of a neuron is then calculated by evaluating the threshold function at the activation of the neuron. So if we present the input vector A, the dot product works out to 3 and f(3) = 1. Similarly, we get the dot products of the second, third, and fourth nodes to be -6, 3, and -6, respectively. The corresponding outputs therefore are 0, 1, and 0. This means that the output of the network is the vector (1, 0, 1, 0), same as the input pattern. The network has recalled the pattern as presented, or we can say that pattern A is stable, since the output is equal to the input. When B is presented, the dot product obtained at the first node is -6 and the output is 0. The outputs for the rest of the nodes taken

together with the output of the first node gives (0, 1, 0, 1), which means that the network has stable recall for B also.

So far we have presented easy cases to the networkvectors that the Hopfield network was specifically designed (through the choice of the weight matrix) to recall. What will the network give as output if we present a pattern different from both A and B? Let C = (0, 1, 0, 0) be presented to the network. The activations would be -3, 0, -3, 3, making the outputs 0, 1, 0, 1, which means that B achieves stable recall. This is quite interesting. Suppose we did intend to input B and we made a slight error and ended up presenting C, instead. The network did what we wanted and recalled B. But why not A? To answer this, let us ask is C closer to A or B? How do we compare? We use the distance formula for two four-dimensional points. If (a, b, c, d) and (e, f, g, h) are two four-dimensional points, the distance between them is:

$$\sqrt{(ae)^2 + (bf)^2 + (cg)^2 + (dh)^2}$$

The distance between A and C is $\sqrt{3}$, whereas the distance between B and C is just 1. So since B is closer in this sense, B was recalled rather than A. You may verify that if we do the same exercise with D = (0, 0, 1, 0), we will see that the network recalls A, which is closer than B to D.

2.1 Asynchronous Update

The Hopfield network is a recurrent network. This means that outputs from the network are fed back as inputs. This is not apparent from Figure2.2.

The Hopfield network always stabilizes to a fixed point. There is a very important detail regarding the Hopfield network to achieve this stability. In the examples thus far, we have not had a problem getting a stable output from the network, so we have not presented this detail of network operation. This detail is the need to update the network asynchronously. This means that changes do not occur simultaneously to outputs that are fed back as inputs, but rather occur for one vector component at a time.

The true operation of the Hopfield network follows the procedure below for input vector **Invec** and output vector **Outvec**:

- Apply an input, Invec, to the network, and initialize Outvec = Invec

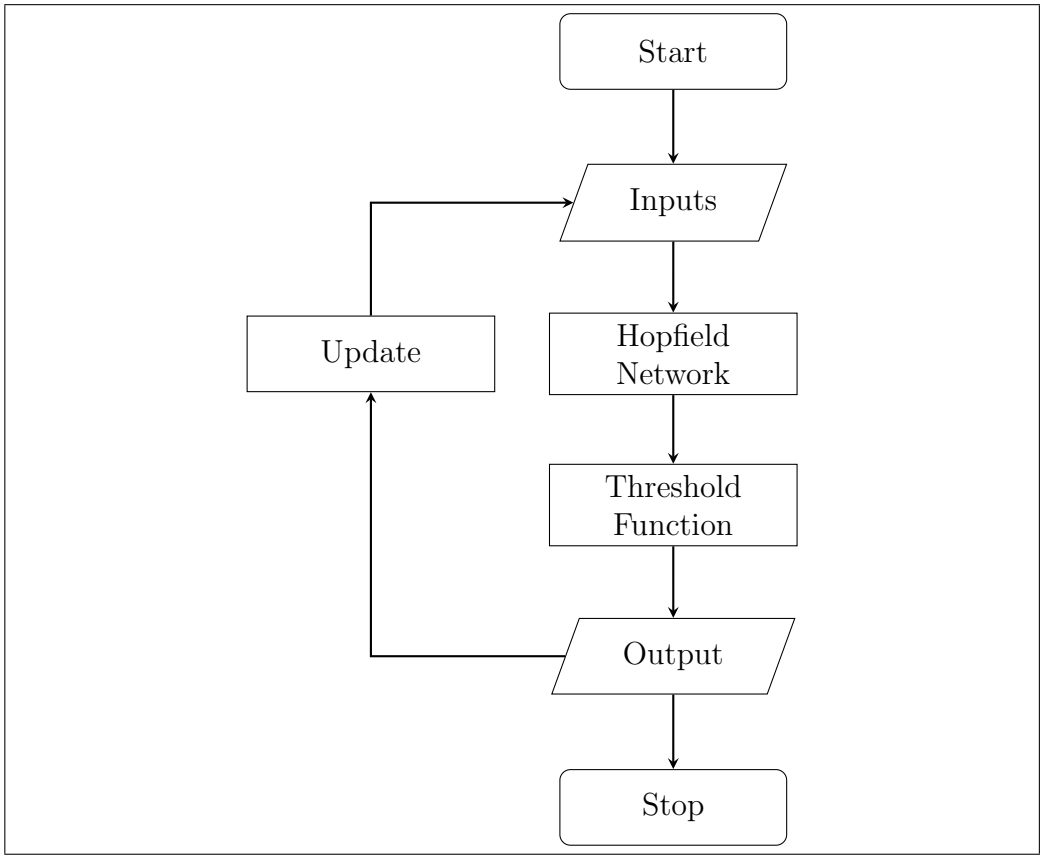

Figure 2.2: Asynchonous Update Flow

- Start with i = 1
- Calculate Value i = DotProduct (Invec i, Column i of Weight matrix)4.
- Calculate Outvec i = f(Value i) where f is the threshold function discussed previously
- Update the input to the network with component Outvec i
- Increment i, and repeat steps 3, 4, 5, and 6 until Invec = Outvec(note that when i reaches its maximum value, it is then next reset to 1 for the cycle to continue)

Table 2.1: Example of Asynchronous Update for the Hopfield Network

Step	i	Invec	Column of weight vectors	Value	Outvec	notes
0		1001			1001	initialization : set Outvec = Invec =Input pattern
1	1	1001	0 -3 3 -3	-3	0001	column 1 of Outvec changed to 0
2	2	0001	-3 0 -3 3	3	0101	column 2 of Outvec changed to 1
3	3	0101	3 -3 0 -3	-6	0101	column 3 of Outvec stays as 0
4	4	0101	-3 3 -3 0	3	0101	column 4 of Outvec stays as 1
5	1	0101	0 -3 3 -3	-6	0101	column 1 stable as 0
6	2	0101	-3 0 -3 3	3	0101	column 2 stable as 1
7	3	0101	3 -3 0 -3	-6	0101	column 3 stable as 0
8	4	0101	-3 3 -3 0	3	0101	column 4 stable as 1; stable recalled pattern = 0101

Now lets see how to apply this procedure. Building on the last example, we now input E = (1, 0, 0, 1), which is at an equal distance from A and B. Without applying the asynchronous procedure above, but instead using the shortcut procedure weve been using so far, you would get an output F = (0, 1, 1, 0). This vector, F, as subsequent input would result in E as the output. This is incorrect since the network oscillates between two states. We have updated the entire input vector synchronously. Now lets apply asynchronous update. For input E, (1,0,0,1) we arrive at the following results detailed for each update step, in Table 2.1

2.2 Classes in C++ Implementation

In our C++ implementation of this network, there are the following classes: a **network class**, and a **neuron class**. In our implementation, we create the network with four neurons, and these four neurons are all connected to one another. A neuron is not self-connected, though. That is, there is no edge in the directed graph representing the network, where the edge is from one node to itself. But for simplicity, we could pretend that such a connection exists carrying a weight of 0, so that the weight matrix has 0s in its principal diagonal.

The functions that determine the neuron activations and the network output are declared public. Therefore they are visible and accessible without restriction. The activations of the neurons are calculated with functions defined in theneuron class. When there are more than one layer in a neural network, the outputs of neurons in one layer become the inputs for neurons in the next layer. In order to facilitate passing the outputs from one layer as inputs to another layer, our C++ implementations compute the neuron outputs in the network class. For this reason the threshold function is made a member of the network class. We do this for the Hopfield network as well. To see if the network has achieved correct recall, you make comparisons between the presented pattern and the network output, component by component.

2.3 A New Weight Matrix to Recall More Patterns

Lets continue to discuss this example. Suppose we are interested in having the patterns $E = (1, 0, 0, 1)$ and $F = (0, 1, 1, 0)$ also recalled correctly, in addition to the patterns $\vec{A}$ and $\vec{B}$. In this case we would need to train the network and come up with a learning algorithm, which we will discuss in more detail later in the book. We come up with the matrix W_1, which follows...

$$W_1 = \begin{bmatrix} 0 & -5 & 4 & 4 \\ -5 & 0 & 4 & 4 \\ 4 & 4 & 0 & -5 \\ 4 & 4 & -5 & 0 \end{bmatrix}$$

Try to use this modification of the weight matrix in the source program, and then compile and run the program to see that the network successfully recalls all four patterns A, B, E, and F.