The Linear Associator

We want to remember the pattern $\mathbf{g}^1$ when we are shown the pattern $\mathbf{f}^1$. We call this "associating" $\mathbf{g}^1$ with $\mathbf{f}^1$.This means that when we impose the input vector $\mathbf{f}^1$on the input neurons, we want the output of neuron I_1 to be $g_1{}^1$, and so on for all the output neurons. If we had learned pattern $\mathbf{f}^1$ using the summed vector rule Eq 4, we would have obtained the output value 1 for $g_1{}^1$ (the assumed length of the input vector). But if instead we had used the Hebb rule during learning, with the output neuron's activity being set at the value $g_1{}^1$ (i.e. the vector of weight changes for I_1 would be $g_1{}^1\mathbf{f}^1$ instead of just $\mathbf{f}^1$) we would recall $g_1{}^1$ as the output. But this is exactly what we want! So using the Hebb rule during learning, with input and output vectors set equal to the desired association, results in remembering the correct association. Note that during learning the activity of the output neurons is determined by the appropriate pattern to be learned, NOT by calculating the dot product of the input and weight vectors. The dot product is used only during the second, recall or retrieval, phase, when the association is being remembered. (In the case of the vector sum memory we considered before, we did not need to distinguish learning and retrieval, because learning did not depend on postsynaptic activity). This procedure will clearly allow learning and recall of all the elements of **f** and **g**, since our choice of output neuron I_1 was arbitrary. Furthermore, if we assume that a set of patterns $\mathbf{f}^1$, $\mathbf{f}^2$….. etc (we could call this set of patterns {**f**}), are all orthogonal, we can learn all the arbitrary associations $\mathbf{f}^1$-$\mathbf{g}^1$, $\mathbf{f}^2$-$\mathbf{g}^2$ etc (e.g. the faces and names of all our friends, or the capitals of different countries).

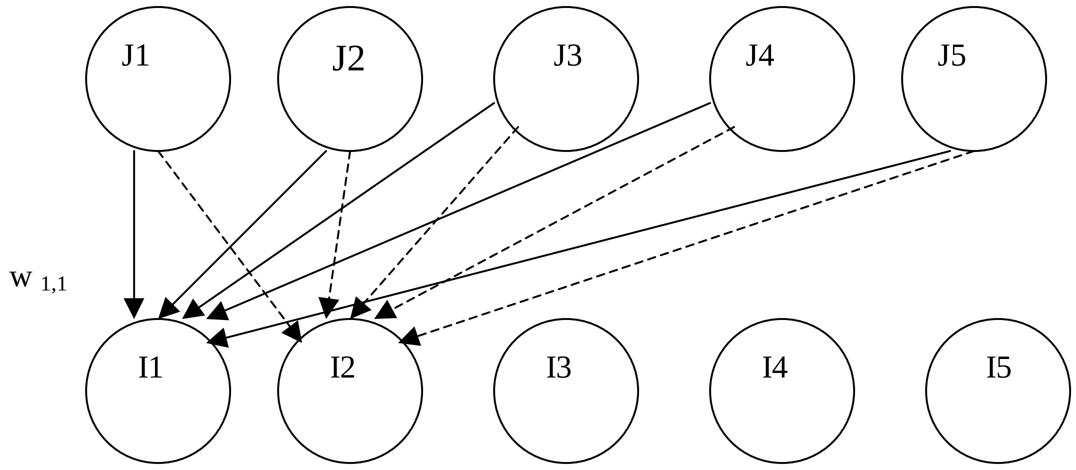

Fig. 1. The linear associator. Input neurons J are all linked by synaptic weights w to all the output neurons I. The weight between J_1 and I_1 is $w_{1,1}$, and generally the weight from the jth cell to the ith cell is $w_{i,j}$. The weights form a matrix **W** whose rows are the values of the weights from the J cells onto a particular I cell (i.e. the rows are the different weight vectors $\mathbf{w}_i$, and whose columns are the weights made by a particular J cell). Only

weights from the first 2 J-cells are shown, but all the other J cells make weights on all the I cells, and all the I cells receive weights from all the J cells. The output of I_i (i.e. g_i) is given by $\Sigma\, w_{i,j} f_j$ (f_j is the firing rate of the jth J cell; the sum is over all j values). The output vector g is thus given by $\Sigma\Sigma\, w_{i,j} f_j$ (the sums are first over the j index, and then over the i index), which can be written as **Wf** (see text).

What happens if we try to store additional associations, for example $\mathbf{f}^2 – \mathbf{g}^2$, in the same set of synaptic connections? If the patterns are orthogonal, i.e. very unlike each other, the above argument still holds. This can be expressed a little more formally by invoking the following (associative) rule for vector multiplication: **a. (b.c)** = **(a.b).c** which is the vector analog of the ordinary rule a(bc) = (ab)c. (The operation in the parenthesis is performed first). In the Hebb rule, the elements $w_{i,j}$ of the matrix **W** are formed by multiplying the individual row and column vector elements g_i and f_j. We will use the notation **g** for the row vector and $\mathbf{f^T}$ for the column vector (T stands for "Transpose", the operation of making a row into a column, or vice versa), and we will use the notation $\mathbf{f^T g}$ to represent the dot product **f.g**, and the notation $\mathbf{g f^T}$ (called the outer product of **g** and **f**) to represent the matrix **W**.

W is formed by adding together all the weight changes produced by the set of associations {**f-g**} in a learning phase. Thus $\mathbf{W} = \Sigma \mathbf{W}^k$ where $\mathbf{W}^k$ is the set of weight changes produced by the kth association. Suppose we apply the input vector $\mathbf{f}^1$ to the network in the retrieval phase. As usual, the output, which we will call **g**"is given by $\mathbf{W f}^1$. i.e.

$$\mathbf{g}'' = \sum_{\text{all } k} \mathbf{W}^k \mathbf{f}^1 \ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \text{Eq } 1$$

But $\mathbf{\Sigma W^k}$ is composed of the sum of the weight changes, in the learning phase, due to $\mathbf{f}^1$ plus all the other patterns where k was *not* 1. So

$$\mathbf{g}'' = \mathbf{W}^1\mathbf{f}^1 + \sum_{k \text{ not } 1} \mathbf{W}^k \mathbf{f}^1 \ldots\ldots\ldots\ldots\ldots\ldots\ldots \text{Eq } 2$$

We can write the second term on the right as

$$\sum_{k \text{ not } 1} (\mathbf{g}\, \mathbf{f}^{\mathbf{k,T}})\, \mathbf{f}^1 = \sum_{k \text{ not } 1} \mathbf{g}\, (\mathbf{f}^{\mathbf{k,T}}\, \mathbf{f}^1) = 0 \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots \text{Eq } 3$$

since the dot product of $\mathbf{f}^1$ with any member of the set {**f**} other than $\mathbf{f}^1$ is zero if these vectors are orthogonal. The term $\mathbf{f}^{\mathbf{k,T}}{}_{(k \text{ not } 1)}$ refers to the (column) input vectors other than $\mathbf{f}^1$.

It therefore follows that

$\mathbf{g}'' = \mathbf{W}^1\mathbf{f}^1$ ………………Eq 4

Now $\mathbf{W}^1$, the weight change due to pattern 1, is given by

$\mathbf{W}^1 = \mathbf{g}^1\, \mathbf{f}^{1\mathbf{T}}$ ……………Eq 5

So $\mathbf{g}'' = (\mathbf{g}^1\, \mathbf{f}^{1,\mathbf{T}})\, \mathbf{f}^1 = \mathbf{g}^1\, (\mathbf{f}^{1,\mathbf{T}}\, \mathbf{f}^1) = \mathbf{g}^1$…………….Eq 6

since, as before, we assume that all vectors have unit length. So we have proved that, providing we learn orthogonal vectors of unit length, what we retrieve on inputting a given vector is the learned associate of that vector. The argument is just a more elaborate case of the vector sum memory argument we used before, but this time what results is not a scalar estimate of familiarity, but the complete desired association. (If the vectors are not of unit length, then the retrieved vector will not have the correct length, but it will still point in the correct direction – so the pattern is remembered). Note that it is only possible to store perfectly n pairs of associations (n is the number of input neurons), because this is the maximum number of vectors that can be orthogonal to each other.

What happens if we input a pattern that the network has NOT learned? If the new pattern is orthogonal to all the learned patterns, Eq 6 tells us that there will be zero output. However, if the new pattern is not orthogonal to the learned patterns, it will generate an output that is similar to the learned output corresponding to the input pattern which it most closely resembles. However, as we discussed for the summed vector memory, provided that the dimensionality is high enough, it works quite well for random patterns (like phone numbers, and to a lesser extent names, addresses, and even faces). There is evidence that some neural circuitry exists to make input patterns that are rather similar more dissimilar, by removing “redundancy” – we will return to this concept. Such preprocessing would help memorization.

We can look at the linear associator in action by converting readily-identifiable words and phrases to vectors using a simple look-up table. In principle this involves writing A = 1, B= 2, C =3 etc. If we learn just the pair “Laurel - Hardy”, then we generate the response “Hardy” to the input “Laurel”. Since the desired and actual outputs are identical, the cosine of the angle between the corresponding vectors is 1.0. Unfortunately, in response to a variety of novel inputs, we still get the output “Hardy” rather than “don’t know”. It repeats the one thing it knows, like a demented parrot. But this is not a fair test – after all, it was trained to remember not to recognize. But suppose it also learns several other pairs, such as Beethoven-Mozart, Paris-France and “Bacon-Eggs”, we find it gives the response “Mozart” to the prompt “Beethoven”, the response “France” to “Paris” and “Egds” to “Bacon”. It makes mistakes, but performs reasonably (and the mistakes reflect more inadequacies in the letter-number coding scheme rather than problems with the memory device; the cosine of the angle between “Egds” and “Eggs” is still quite large.). In response to the unlearned “Frank” it outputs “Parid” rather than the correct “Sinatra”;

it has not learned "Sinatra" so it gives roughly the output to the item it learned that most resembled "Frank".

So far we have considered the linear associator as a "heteroassociator" – learning associations between different vectors **f** and **g**. The linear associator can also act as an "autoassociator", if it is taught the association "**f** –**f**". This can be done by imposing the same vector on both the input neurons and the output neurons during the learning phase. At first glance this appears a bit pointless : why learn to respond "Hardy" to the prompt "Hardy" ? However, because the linear associator comes up with the response that most closely resembles a stored memory ("Parid" in the example above), it can retrieve whole memories from fragments (the taste of the madeleine in Proust recalls the whole of Combray). We can formalize this as follows. Consider a vector **f** composed of 2 orthogonal parts **f**' and **f**" (i.e. **f** = **f**' + **f**"). Consider a matrix **W** storing the autoassociation **f**-**f**. Thus

$$\mathbf{W} = \mathbf{f}\mathbf{f}^{\mathbf{T}} = (\mathbf{f}' + \mathbf{f}'')(\mathbf{f}' + \mathbf{f}'')^{\mathbf{T}}$$

If the fragment **f**' is applied to the input (in a retrieval phase after learning") the output **g** will be given by

$$\mathbf{g} = \mathbf{W}\,\mathbf{f}' = (\mathbf{f}'\mathbf{f}'^{\mathbf{T}} + \mathbf{f}'\mathbf{f}''^{\mathbf{T}} + \mathbf{f}''\mathbf{f}'^{\mathbf{T}} + \mathbf{f}''\mathbf{f}''^{\mathbf{T}})\,\mathbf{f}' = \mathbf{f}'\{\mathbf{f}'^{\mathbf{T}}\,\mathbf{f}'\} + \mathbf{f}'\{\mathbf{f}''^{\mathbf{T}}\,\mathbf{f}'\} + \mathbf{f}''\{\mathbf{f}'^{\mathbf{T}}\,\mathbf{f}'\} + \mathbf{f}''\{\mathbf{f}''^{\mathbf{T}}\,\mathbf{f}'\}$$

where we grouped together terms that form inner products using brackets {}. The second and fourth terms above contain inner products of orthogonal vectors, which are zero. So the equation simplifies to

$$\mathbf{g} = (\mathbf{f}'+\mathbf{f}'')\,(\mathbf{f}'^{\mathbf{T}}\mathbf{f}') = a\,\mathbf{f}$$

Since $(\mathbf{f}'^{\mathbf{T}}\mathbf{f}')$ is an inner product (i.e. just a number, a), we have shown that the retrieved response **g** to the input fragment **f**' lies in the same direction as the original memory **f**.

We can test this using phrases as random vectors. For example, the linear associator can learn, autoassociatively, the phrases "London England", "Paris France", "Berlin Germany" "Rome Italy" etc. When given the prompts "England", "France", "Germany" and "Italy" it might respond "Longix Hnlans", "Parid Fganle" etc. The reason it makes so many errors are (1) the vectors are not perfectly orthogonal" and (2) there are problems with the numerical encoding scheme. Nevertheless it is not completely wrong – it gets a C not an F. There are at least three obvious ways to improve the performance of this autoassociator.

(1) When it makes mistakes, use the mistakes to improve the memories (to correct some of the weights). This is known as supervised learning.

(2) Since the responses are more similar to the originals than are the prompting fragments, use the rough first answers as new prompting fragments. The second answers should be even more similar to the originals, and hopefully by repeating

this many times, one might converge to exactly the right answer. This involves feedback.

(3) We could try to make it easier for the Linear Associator by imposing the condition that input and output values can only take certain discrete values, for example 1 or –1. We would then use a "majority" rule to convert the weighted sums at each output neuron to either 1 or –1, as appropriate. Discretisation is a general method for dealing with noise and error, because we know that nondiscrete (continuous or intermediate-level) signals have to rounded up or down.

The Hopfield network we will consider next combines both of the latter strategies.

It is worth comparing this memory system to that of a computer. In the network, all n memories are stored simultaneously in all the n^2 memory locations (the connections). They are smeared out over the whole system, and a particular item of information (a nose, or a letter of a name) is not found in a particular location. If some of these locations are damaged, it will degrade all the memories to a similar extent, but if the damage is slight the degradation will be slight. In the computer, each memory is located in a unique set of labeled "locations". For example, we could store each input-output pair at separate locations, labeled 1-in, 1-out, 2-in, 2-out etc. Then given an input pattern the computer could check through all the stored input patterns and determine the label, and then read off the output pattern with the same label. Just like the associator the computer can store n memories using n^2 storage locations. The computer is not restricted to orthogonal patterns, but it does require someone to write and store a program, and it is not automatically autoassociative. If the computer memory is partially corrupted, it may retrieve totally the wrong memory – it does not degrade "gracefully". The most significant difference is in style, not performance. The computer treats everything serially – vectors are broken down to numbers, which are further broken down to binary digits. This serial style means that all operations have to be extremely fast but exact, containing eseentially no hardware errors. The network handles the whole vector at once, in a parallel or distributed fashion, so no error can influence other simultaneous computations. Individual computations (eg multiplying vector elements) can be slow and sloppy. This makes it very fault tolerant. In a nutshell, computers use relatively small numbers of precise components, and brains use vast numbers of imprecise components. Will a computer ever exceed the performance of the human brain? For specialized tasks (tictactoe, checkers and most recently chess) this has already happened (though the computer programs that play these games were developed by humans; the brain is selfprogramming). But across a range of tasks this still seems far off. The real problem is that computers already operate close to limits set by physics (speed of light, Planck's constant, Avogadro's number). Further large increases in chip densities will probably make transistors work less precisely, forcing computers to become less serial and more parallel.

It could be argued that in real learning situations, such as a child learning the names of objects, the different names are not random, but follow clear rules – the rules of english pronunciation (i.e. phonetics). However, although the sounds follow regularities (which

differ in different languages), the child is not actually associating sound with objects, but sequences of *phonemes* with objects. The sequences of phonemes within words are fairly random (for example, of the 25 possible words formed by placing one of the first 5 consonants on either side of the first vowel – bab, bac, bad etc – 14 are actual words, including verbs, nouns and adjectives. Furthermore real words represent only a small subset of possible words. The child also has to learn how to categorise sounds as phonemes, but this requires a rather different approach than simple association. The brain uses a variety of learning devices, including brute force association of random facts in the way discussed in this lecture.
In a later lecture we will discuss the central problem of language: how can different humans come to agree to use the same phoneme sequence – object associations, given that the sounds that humans exchange do not infallibly identify phonemes?

A Note on Orthogonality of Binary Vectors.

In a binary vector each element can take only one of 2 values. If we take these values as 1 and –1, then the length of a vector is simply and conveniently the dimensionality of the vector (i.e. the number of elements, n). In principle a polynucleotide string is a binary vector (which we could represent 1,0,1,1, etc or equivalently 1,-1,1,1). A frequent question in molecular biology is: how similar are 2 polynucleotide sequences? A convenient way to measure the similarity is in terms of Hamming distance: the number of basepairs that differ (call it d). The relationship between the Hamming distance between 2 binary vectors **f** and **g** and their dot product is $d = (n - \mathbf{f}.\mathbf{g})/2$. Suppose that we compare 2 random strings. We would expect that if the strings are long enough, they would (by chance) agree in half the positions, so the Hamming distance would be n/2. Now as random vectors become infinitely long, they become increasingly orthogonal, with a cosine between them of 0. This agrees with the given formula. Also, if the 2 random vectors are identical d = 0 and by the formula $d = (n - n \cos\theta)/2$
$= 0$ (since the lengths of the vectors are root n). If the 2 polynucleotides differ in *every* position, they are complementary, and represent binary vectors that lie in the same orientation.