Homework 3: The Linear Associator (20 points total)
Psych 186B
Due: 11:59pm, Sunday, Jan. 26, 2014

For our first real experiment in neural modeling let us investigate the linear associator model for associative memory. This model displays many of the virtues of distributed neural models and is easy to implement. It is described in detail in Chapters 6 and 7 of the book.

Write and test your programs carefully, so a simple change in the modification rule can be tested in the same way as the simple linear associator described here. The next assignment will incorporate an error correction rule into this model, and assignment after that will build in a non-linearity. Allow for modifications to your code. Also, be careful in your write-ups of this set of assignments since many of the effects you see are typical of more complex network models.

You will make use of the programs you developed in the first homework. Your basic strategy will be to write a program realizing the linear associator and then study it just as you would study any new system. I will make some suggestions for things to look at, but please feel encouraged to try your own ideas. There are a lot of aspects of this model. Many of the qualitative properties displayed by the linear associator are also shown by more complex neural models. Therefore, it is worth taking some care with the writeup and experimentation for this assignment.

Theory. The essence of this assignment is to show that a generalized Hebb synapse realizes a powerful associative system. In fact, almost all variants of Hebb synapses realize associative systems, one of the things that initially caused most people to believe that they must exist in some form or other in the nervous system given the associative nature of human memory. There is now very good biological evidence for Hebb synapses, probably of several kinds, in the brain.

Given two sets of neurons, one projecting to the other, and connected by a matrix of synaptic weights **A**, we wish to associate two activity patterns **f** and **g**. We assume **A** is composed of a set of modifiable synapses.

We make two fundamental assumptions. First, the neuron acts to a first approximation like a linear summer of its inputs. That is,

the *ith* neuron in the second set of neurons will display activity *g(i)* when a pattern **f** is presented to the first set of neurons according to the rule,

$$g(i) = \sum_j a(i,j) f(j).$$

Justification for this first order assumption was discussed in the text. This model is basically a generalization of the Limulus eye, discussed in Chapter 4 of the text, but allowing positive as well as negative weights. We will introduce some simple non-linearities in a later assignment.

Considering the patterns shown by the entire system, we can write the set of output activities as the simple matrix multiplication

$$\mathbf{g} = \mathbf{A}\mathbf{f}.$$

Our second fundamental assumption involves the construction of the matrix **A**. We assume that these matrix elements (connectivities) are modifiable according to a generalized Hebb rule,

$$\Delta A[i,j] \propto f[j]g[i].$$
$$= \eta f[j]g[i]$$

The constant η is sometimes called the *learning constant*. Suppose that the matrix **A** starts from zero, and a single association of **f** and **g** is formed. Then **A** is given by

$$\mathbf{A} = \eta \begin{vmatrix} f[1]g[1] & f[2]g[1] & f[3]g[1] & \ldots & f[N]g[1] \\ f[1]g[2] & f[2]g[2] & f[3]g[2] & \ldots & f[N]g[2] \\ f[1]g[3] & f[2]g[3] & f[3]g[3] & \ldots & f[N]g[3] \\ \ldots & & & & \\ f[1]g[N] & f[2]g[N] & f[3]g[N] & \ldots & f[N]g[N] \end{vmatrix}.$$

This is a confusing way to represent a matrix with such simple structure. Note that the rows of the matrix are composed of the vector **f** and the columns contain the vector **g**, and the elements are given by the product of the appropriate row and column elements of **f** and **g**. This kind of matrix is called an *outer product* matrix.

One notation for representing this kind of matrix compactly (there are several others) is used here.

The t*ranspose* operation in linear algebra interchanges rows and columns of a matrix, **A**, that is, if A*[i,j]* is an element of **A**, the transpose matrix, traditionally represented by superscript '^{T}', as in $\mathbf{A}^{\mathbf{T}}$ is given by,

$$\mathbf{A}^{\mathbf{T}}[i,j] = \mathbf{A}[j,i].$$

If we have a column vector **f**, our usual convention, then the transpose of **f** is a row vector. Then we can write our matrix **A** as

$$\mathbf{A} = \mathbf{g}\mathbf{f}^{\mathbf{T}}$$

We make one other observation. One vector times the transpose of another is a matrix. A transposed vector times an untransposed vector is (according to the rules of matrix algebra) an inner product, for example,

$$[\mathbf{f},\mathbf{g}] = \mathbf{f}^{\mathbf{T}}\mathbf{g}$$

Note that it matters a lot whether the transpose operation occurs first or second in the equation.

Given the above notation, we have the outer product **A**,

$$\mathbf{A} = \mathbf{g}\mathbf{f}^{\mathbf{T}}$$

Suppose after this **A** is formed, the pattern **f** is input to the system. Then since the matrix **A** has non zero elements, a pattern **g'** will be generated as the output to the system according to the simple matrix multiplication rule discussed before. This output, **g'**, can be computed as

$$\begin{aligned} \mathbf{g'} &= \mathbf{Af}, \\ &= \eta \mathbf{g} \mathbf{f}^T \mathbf{f}, \\ &\propto \mathbf{g} \end{aligned}$$

since the square of the length of **f**, that is, $\mathbf{f}^T\mathbf{f}$, is simply a constant. Thus subject to a multiplicative constant, we have generated a vector in the same direction as **g**. We can change the length (but not the direction) of **g** either by adjusting the learning constant η or by normalizing the lengths of the input vectors. In either case, the cosine of the angle between **g** and **g'** will be 1, since they are pointed in the same direction.

In the case where more than one set of associations is formed and stored in the same matrix, the potential exists for interference between the different associations. In fact, this observation leads to many of the immediately testable predictions of the models.

One special case deserves mention. In homework 1 you convinced yourselves that the inner product of two random vectors in a high dimensionality space is close to zero. Suppose we store a group of associations.

We will assume that the set of input vectors, $\{\mathbf{f}_i\}$, are normalized, so we can assume the inner product $[\mathbf{f}_i, \mathbf{f}_i] = 1$. Suppose the set $\{\mathbf{f}_i\}$ is composed of orthogonal vectors, an idealization of random vectors in a space whose dimensionality is high compared to the number in the set.

For convenience, we will also assume that the learning constant $\eta = 1$.

Suppose each pair of associations

$$\mathbf{f}_i \rightarrow \mathbf{g}_i$$

generates an outer product associative matrix

$$\mathbf{A}_i = \mathbf{g}_i \mathbf{f}_i^T$$

and the overall connectivity matrix **A** is the sum of all the matrices of the individual associations, that is,

$$\mathbf{A} = \sum_i \mathbf{A}_i$$

It is then easy to show that the associator works perfectly. Suppose a member of the set **{f}**, say $\mathbf{f}_i$ with paired association $\mathbf{g}_i$, and with the outer product matrix $\mathbf{A}_i = \mathbf{g}_i\mathbf{f}_i^T$, part of the sum forming **A**, is the input to the system. The output is then given by $\mathbf{Af}_i$, or

$$\mathbf{Af}_i = \sum_j \mathbf{A}_j\mathbf{f}_i$$

$$= \sum_{j\neq i} \mathbf{A}_j\mathbf{f}_i + \mathbf{A}_i\mathbf{f}_i$$

$$= \sum_{j\neq i} \mathbf{g}_j\mathbf{f}_j^T\mathbf{f}_i + \mathbf{g}_i$$

By assumption (or approximation, remember homework 1!) the set **{f}** is orthogonal, that is, the inner product of $\mathbf{f}_i$ with any different $\mathbf{f}_j$ in the set is zero. Then all the terms in the sum are zero and we have now shown that for this specific situation,

$$\mathbf{Af}_i = \mathbf{g}_i$$

Therefore, if the pairs $\mathbf{f}_i \rightarrow \mathbf{g}_i$ whose inputs are orthogonal are stored, the association is perfect and the system reconstructs the output pattern perfectly. There is a capacity limitation on such a system because there cannot be more than *n* orthogonal vector pairs stored, where *n* is the dimensionality of the vectors.

Of course, we are all suspicious of mathematical results (aren't we?). Let us try to study how well the system works with more reasonable assumptions, using the computer.

The Simulation. Your task in this simulation will have several parts.

You will generate mean zero normalized random vectors **f** and **g** as test vectors and then use them to construct an outer product matrix **A.** Mean zero in this context means that the average vector element has zero mean. For example, if you are generating

uniformly distributed random vectors between 0 and 1, the mean will be 0.5. Therefore you will have to subtract 0.5 from each element to ensure mean vector values for the entire vector of about zero.

A good dimensionality to use at first would be 100. The computer you are using may allow you to go to higher dimensions but older personal computer compilers may start running into array size limitations with large matrices. For your convenience, here is a Pascal **PROCEDURE** to generate the outer product matrix:

```
PROCEDURE Outer_product (A, B: Vector; VAR C: Matrix);
    VAR I,J: INTEGER;

    { Note:  We want the outer_product procedure call to be
      in the form gf transpose.  Then C = AB transpose. }
    BEGIN
    FOR I:=1 TO Dimensionality DO
        FOR J:= 1 TO Dimensionality DO C[I,J]:= B[J] * A[I];
    END;
```

Conversion of this **PROCEDURE** to other languages should be simple.

You should know how to generate random normalized mean zero vectors from homework 1. An interesting simulation question is whether it makes a difference if you use mean zero elements with element values taken from normally distributed, uniformly distributed, or a random binary distribution, that is, +1 or −1. In fact it makes almost no difference. You might want to show this, however.

A more difficult question asks, "If element values can be -1, 0, or +1 does the fraction of zeros make a difference?" The answer is yes but this question touches on hard questions about data representation.

First (2 points), see if the system can associate a single pair of vectors.

(1) Generate two vectors **f**, **g**.

(2) Set the mean of the vectors to zero.

(3) Normalize them, i.e. set the lengths to equal one.

(4) Compute **A**.

(5) Show that **Af** gives an output **g′** which is in the same direction as **g**.

With a normalized input and $\eta = 1$, vectors, **g** and **g′** will be equal. Compute the cosine between **g** and **g′** to demonstrate that they are in the same direction. It should be one. Compute the length of **g′**. It should be also one.

Second (2 points), see if the system can discriminate between new and old input vectors.

(1) Generate a new normalized random vector, **f'**.

(2) Check to see if it is more or less orthogonal to **f** by looking at the cosine of the angle between **f** and **f'**.

(3) Compute **Af'** and look at the length of this vector.

What do you think it should be? What is it?

Third (10 points total), store many vector associations (50 of them), $\mathbf{f}_i \rightarrow \mathbf{g}_i$ and see what happens.

(1)(1 point) Generate many pairs of normalized random vectors, $\mathbf{f}_i$ and $\mathbf{g}_i$.

(2)(1 point) compute the outer product matrices, $\mathbf{A}_i = \mathbf{g}_i\mathbf{f}_i^{\mathbf{T}}$.

(3)(1 point) form the overall connectivity matrix, **A**, as the sum of the individual outer product matrices, that is,

$$\mathbf{A} = \sum_i \mathbf{A}_i$$

A will change as it accumulates more and more pairs. Constructing truly orthogonal sets of vectors can be done but is not easy except for one trivial case. (Do you know what it is?). So trust to high dimensionality to let you approximate real orthogonality. This approximation is a more realistic and interesting case in any event.

(4)(3 points) Test the resulting matrix as follows:

(a) Compute the output for each stored input, $\mathbf{f}_i$.

(b) Compare it with what it should be (i.e., $\mathbf{g}_i$) by computing the cosine between the actual output, **g'**, and the true association, $\mathbf{g}_i$.

(c) Compute the length of the output vector, **g'**.

(d) Test the selectivity of the system by generating a new set of random vectors (50 of them) and computing the matrix product between each of these vectors and the matrix, **A**, you constructed. If the system develops any selectivity, we hope that the output of the system to a new random vector would be small. You can also compute the length of the outputs for the new random vectors and compare the lengths to the vectors that were stored. Compute the average lengths. (Optional: If you feel ambitious, you might plot the length data collected here in the form of a histogram as you did before in homework 1. You might want to generate many more random vectors (say 500) so that the histogram is reliable.)

(5)(4 points) Repeat for different numbers of pairs of stored vectors, and observe how the behavior of the system deteriorates as more and more vectors are stored. Of your 100 dimensional system, interesting numbers of stored vectors to look at would range from 1 pair (which should give perfect results!) to 100 (should be poor) with a few intermediate values: 20, 40, 60, and 80 pairs.

Fourth (6 points), perform a simple computational experiment using the system. **You only need to do one of the below**, or, best of all, think up your own. By now, you should know how to reasonably select your parameters, e.g., the dimensionality of your system, the number of samples you have run to make your case (neither these numbers should be small). So we will not specify numbers in the following. Please note however that you will earn only partial credits if your numbers are too small.

(1) Destroy parts of the matrix at random and see how it distorts the output, that is, do a numerical *ablation* experiment. One of the virtues of matrix models is generally felt to be their resistance to noise and damage. Although this statement is often made, it is seldom actually demonstrated. Convince yourself that this biologically important property is true. You can use the cosine between pre- and post- ablated outputs as a measure of damage. There are other measures, but this is a reasonable one and it is easy to compute.

Question: Would you expect the response of the system to damage to depend on the representation of data in the network. If so, why? Can you give reasons?

(2) Before learning of each input-output pair takes place, randomly set some matrix elements to be zero (i.e. some `neurons' are not connected to other `neurons'.) (You may want to try different proportions of elements being set to zero, e.g., 25%, 50%, 75%.) See how this changes the properties of the system. Can you store as many vectors before the system becomes unusable? This system is also interesting when compared with the error correction model to be described in the next assignment.

(3) Make chains of associations. That is, store vectors **f→g**, **g→h**, **h→i**, **i→j**, etc. Observe how noise increases if we start off with **f** and cycle output back through the system. This particular demonstration is dramatic when compared with same thing done using the error correcting learning algorithm we shall study in the next assignment.

(4) If the input and output vectors are identical, that is, **f** = **g**, the association of a vector with itself is referred to by Kohonen as an `autoassociative' system. (Advanced comment: One way to view a simple autoassociative system is that it is forcing the system to develop a particular set of eigenvectors.)

Suppose we are interested in looking at autoassociative systems,

$$A = \eta \mathbf{f}\mathbf{f}^{\mathbf{T}}$$

where η is the learning constant.

We can use feedback to reconstruct the missing part of an input state vector. To show this, suppose we have a normalized state vector f, which is composed of two parts, say **f′** and **f″** so that

f = f′ + f″.

Suppose we want **f′** and **f″** to be orthogonal. One way to accomplish this would be to have **f'** and **f″** be subvectors that occupy different sets of elements -- say **f′** is non-zero only for elements [1..*m*] and **f″** is non-zero only for elements [(*m*+1)..*n*] where *n* is the dimensionality.

Then consider a matrix **A** storing only the autoassociation of **f** that is

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

We will assume the learning constant η=1 as before.

The matrix is now formed. Suppose at some future time a sadly truncated version of **f** with many missing elements, say **f′** is presented at the input to the system.

The output is then given by

$$\text{(output)} = \mathbf{A}\mathbf{f}'$$

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

Since **f′** and **f″** are orthogonal,

$$\text{(output)} = (\mathbf{f}' + \mathbf{f}'')\ [\mathbf{f}',\mathbf{f}'].$$

$$= c\mathbf{f}$$

where c is some constant since the inner product [**f′**,**f′**] is simply a number. Therefore the autoassociator has regenerated the missing part of the state vector by using the information in the connection matrix. Of course if a number of items are stored, the problem becomes more complex, but with similar qualitative properties.

Try to study this system. This technique has the valuable property that if only part of **f** is put into the system it will regenerate the missing part. This is what is called the *reconstructive* property of autoassociative memories. You might try to think of some way to demonstrate this property. Looking at what happened to the missing (i.e. zero) parts of the vector is a good technique.

Autoassociative models are very general and many non-linear neural network models are based on them and their generalizations. There is now considerable evidence that autoassociative network memories may actually exist in the hippocampus, a cortical structure known to be involved in memory.

(5) Many variants of the simple outer-product learning rule are possible. There is some suggestion from biology that small changes in activities are not learned, that is the learning rule has a "hole" in it when values are small. This idea could be implemented quite nicely by simply putting in a modification threshold, that is, there is no modification unless some parameter

is above threshold. Possible obvious parameters would be pre-synaptic activity, post-synaptic activity, or their product. You might try a product threshold as the most obvious candidate, and the most in harmony with the outer product rule.

This rule would also be easy to implement since it would simply involve checking the outer product matrix for values below threshold and then setting them to zero. Obviously you will have to check to see how many values are set to zero. If there are too many or too few obvious things will happen. Check to see what difference setting small values to zero makes in the associative properties of the system. A rule of thumb I have heard occasionally says something like "the largest 25 percent of the connection strengths do almost all the work." Is this true? Many more complex possibilities for modifying the learning constant are possible and have been investigated including some temporally based ones we will discuss in the future.