Homework 3 (20 points total)
Error Correction Techniques

Psych 186B

January 27, 2014
Due 1159pm, February 2, 2014

Let us look at an important extension of the linear associator: an error correcting technique called the *Widrow-Hoff* algorithm, a well known adaptive learning technique. This algorithm is also called the *LMS Algorithm*, where LMS stands for "Least Mean Squares." It was first described around 1960 by Bernard Widrow and Ted Hoff as a neural network learning algorithm. It is used now as an adaptive filter for high speed modems and filters and also has a number of other important communications applications. The original meeting paper that announced the technique was quite clear and readable. The theory behind the technique is described in Chapter 9 of the text.

There are several reasons why we do this assignment now.

First, the properties of the linear associator should be fresh in your mind. In particular, you should have observed that the linear associator degrades badly as more and more information is stored.

Second, a simple change in your program will allow us to implement a system that will `correct' errors and which will converge to almost perfect answers even when a number of associations are to be learned. This `teachable' and higher performance system will be useful by itself and also serves as the basis for a more complex learning algorithm for multi-layer systems called *back-propagation*. Back propagation is presently the most commonly used neural network algorithm.

In the linear association system, when we want to associate state vectors **f** and **g**, we construct a matrix according to the Hebbian (outer product) learning rule

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

This algorithm does not care whether or not the association will be correct, that is, there is no feedback as to final system behavior although both the input and output are specified during learning. As you were able to demonstrate, this technique comes close to the right answer if few associations are stored, but it is never completely correct and it tends to get worse with more associations because other learned patterns interfere with correctly retrieving the right association.

Suppose we ask what would be required to obtain perfect association: that is, given an **f**, is there a matrix **A** that gives perfect or nearly perfect association, so that

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

As long as we don't learn too many associations there is likely to be such a matrix. The problem is to find it. The general problem as to whether such a matrix exists at all for these system and what its mathematical form is, is discussed at length by Teuvo Kohonen in a valuable book, *Associative Memory: A system theoretic approach*, Berlin: Springer, 1977, 2nd Edition, 1987.

The most popular way to solve this problem is to use a *supervised* algorithm. In a supervised algorithm, the system is tested with an input and told when and how much it is in error. Then the system attempts to reduce its error.

Let us implement a simple form of error correction?

Suppose we want to learn the association

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

Because other information has been stored in the system, we have a matrix **A** which instead gives rise to

$$\mathbf{Af} = \mathbf{g}'.$$

where **g'** is not the correct association, that is, **g** does not equal **g'**.

The simplest way to correct **g'** so it equals **g** is to assume we have available both what the association should be and what it actually is, that is, we have access to both **g** (the association we want to form) and **g'** (what we actually got). This information can be obtained through appropriate teaching techniques, neural mechanisms, or maybe such a mechanism doesn't exist at all in the

real brain! Algorithms where the correct answer is assumed to be known to a teacher so the correct answer and the answer produced by the network can be compared are called ***supervised learning algorithms***. Algorithms where the correct answers are not known to the system are called ***unsupervised algorithms***. The learning problems involved are quite different in the two cases. Generally, supervised algorithms are much quicker and more efficient since they have access to more pertinent information and can decide if an answer is correct or not and how big the errors are. Many unsupervised algorithms exist. They are often more difficult to implement than supervised ones, since the problem is less well defined though they are also often much more interesting in their behavior.

The way to implement the Widrow-Hoff algorithm is to learn the difference between what we got as output and what it should have been, that is, the vector difference between the two vectors, an error vector, **g-g′**. The theoretical foundation for this technique has been discussed in Chapter 9. This learning rule implements one form of a ***gradient descent*** or ***steepest descent*** algorithm, that is, the system will evolve so as to constantly reduce the error term.

We will call the old matrix **A**, and the matrix we are going to add to **A** to improve its behavior, **ΔA**. Perceptive readers will detect here the germ of a differential equation, a point that will not be pursued further. Now, suppose we form

$$\mathbf{\Delta A} = \kappa(\mathbf{g} - \mathbf{g}')\mathbf{f}^{T}$$

and add **ΔA** to the original matrix, **A**, so that **A** now becomes

$$\mathbf{A} \leftarrow \mathbf{A} + \mathbf{\Delta A}.$$

First, notice that if the matrix **A** is initially zero, **g'** will be zero and the learning rule is the same as the simple outer-product Hebbian rule.

Second, if the matrix is constructed so that it gives perfect responses, that is,

$$\mathbf{g}' = \mathbf{g}$$

there will be no learning at all since **ΔA** will be zero. If the entire set of inputs gives the correct associated outputs, then learning ceases. This answers an important criticism of simple

Hebbian systems where connections are constantly modified with learning and potentially can increase in magnitude without bound.

When error correcting techniques are implemented, there is a scalar, κ, multiplying the error signal, that is,

$$\mathbf{\Delta A} = \kappa(\mathbf{g} - \mathbf{g}')\mathbf{f}^T$$

If κ gradually drops toward zero as learning progresses, eventually the matrix **A** will cease to change, even if errors are not completely corrected. If κ drops very slowly, it can be shown that the final matrix is the best matrix in the sense of linear regression, that is, the resulting matrix will minimize the mean square error between the actual and desired outputs over the input set. This is why the Widrow-Hoff algorithm is also called the LMS (Least Mean Squares) algorithm.

If κ remains constant, the matrix **A** will generally change a little from learning trial to learning trial since it is constantly correcting the error of the last pair of associations it has seen.

The exact way k drops is not particularly important as long as it is relatively slow. One possibility is to let κ be proportional to *1/n* where *n* is the number of learning trials. Use of a decreasing function for k is sometimes called *tapering*. Don't use tapering at first -- the system will work fine without using it as long as you are not storing very many associations.

We can get some idea of a reasonable initial value of κ by the following calculation. If $\kappa = 1/[\mathbf{f},\mathbf{f}] = 1/\mathbf{f}^T\mathbf{f}$, the reciprocal of the square of the length of **f**, the system will respond perfectly to the immediately preceding association. Suppose we are working with associated pair of vectors, $\mathbf{f} \rightarrow \mathbf{g}$ and association is not quite perfect, so that

$$\mathbf{Af} = \mathbf{g}'.$$

We increment the matrix **A** by the error matrix $\mathbf{\Delta A}$ so that the new matrix, **A'** is given by

$$\mathbf{A}' = \mathbf{A} + \mathbf{\Delta A} = \mathbf{A} + \kappa(\mathbf{g} - \mathbf{g}')\mathbf{f}^T$$

If the just learned vector **f** is the input to the system, and $\kappa = 1/\mathbf{f}^T\mathbf{f}$ then

$$\mathbf{A}'\mathbf{f} = \mathbf{g}' + \kappa(\mathbf{g} - \mathbf{g}')\mathbf{f}^T\mathbf{f} = \mathbf{g}.$$

Some care has to be taken with the value of κ. If κ is too large, the system will oscillate and will blow up numerically. (Try it and see.) In this case, learning overcorrects, and increases the error rather than decreasing it. If you draw a picture of the error vector, it is obvious why this is so -- the overcorrection will continues to grow and eventually will cause a floating point error in your computer. Generally, choose an initial value for κ somewhat less than $1/\mathbf{f}^T\mathbf{f}$. If κ is too small the system may take a long time to converge to an answer, however, it will get there eventually.

If κ is a constant $\mathbf{A}$ may never converge to an unchanging value but will oscillate around the best matrix in the sense of linear regression. This observation suggests that a kind of short term memory exists in most error correcting systems, including this one. Suppose we have a set of associations $\{\mathbf{f} \rightarrow \mathbf{g}\}$. Suppose we present associations from this set to the learning system, that is, choose $\mathbf{f}_i \rightarrow \mathbf{g}_i$, at random. If κ is relatively large, that is, on the order of $1/\mathbf{f}^T\mathbf{f}$ the association will be perfect or nearly so as shown above for association learned in the immediate past. Association will be less accurate if a pair has not been presented recently, because the corrections to later pairs may have corrupted the association between earlier pairs.

This error correction system (and most error correcting systems) displays a few of the properties of a short term memory, in that it gives very good responses to the immediate past associations, and then gets progressively worse and worse for temporally distant associations. This might make a research project for someone to investigate. It perhaps also suggests why constant practice is necessary.

Assignment

Since you have just finished the linear associator assignment, Assignment 2, your task is to repeat some of the studies you performed for the linear association system, using the Widrow-Hoff procedure. The only essential difference is that you will use more computer time! Specifically, please repeat homework 2.2 (linear associator), the third problem. Please try to use a system of 100 dimensions, with stored pairs of vectors in the number of 20, 40, 60, and 80.

A good way to proceed is to set up the set of associations to be in a pair of arrays, the F_set and the G_set. Then pick an associated pair from the arrays at random and associate them with the error correction procedure.

Add the resulting matrix to the developing **A** matrix. Keep going until done, that is, until some accuracy criterion has been met. Part of the lore of error correcting models is that the pairs to be learned must be presented in random order. Otherwise sequential dependencies may be built into the matrix. In fact, this problem rarely occurs.

Some typical parameters might be useful to check program performance. Suppose we are working in a 200 dimensional system with perhaps 25 associations to learn. The system will learn essentially perfectly, that is, with the cosine of the angle between **g** and **g'** for every association over 0.9999 with about 25 presentations of each vector for a total of about 600 random presentations. The learning constant κ was slightly less than $1/\mathbf{f}^T\mathbf{f}$.

Since we present each pair multiple times, this simulation will obviously take many times the computer time of a system where each association is only learned once. However if the simulation starts to take minutes something is wrong with the program. (Or you are using an extremely slow computer.)

Things to look for that are special to this error correcting technique include:

(a) (5 points). The oscillations mentioned above. (Oscillation here means a variable, e.g., error vector length, never converges but keeps increasing and decreasing continuously)

(b) (5 points). How long it takes to converge. (Try to come up with a reasonable criterion for convergence, e.g., error decrease smaller than 1%)

(c) (5 points). How many associations can be stored before the system starts to break down. (If you think about it, a system breaks down when it associates f and g vectors no better than chance, right?)

(d) (5 points). What would happen if we presented the associations in sequence rather than randomly. What happens in the linear associator if we present associations to be learned in different sequences?

For your convenience, I have included a program fragment to show you how simple it is to implement this technique. You have already run across variants of the local PROCEDUREs.

The most common error in implementing Widrow-Hoff is to get the sign of the correction wrong. Then you have constructed an *error-enhancing* algorithm, an algorithm in limited demand. The next most common error is getting the learning constant too large so the state vector undergoes wild oscillations in length and angle from step to step and eventually blows up. Another common error is getting the learning constant too small so huge amounts of computer time are used. Bizarre behavior of your programs indicates a bug, perhaps one of these. Widrow-Hoff, when properly implemented, is impressively robust and effective. Every year at least one student emphatically insists (a) the method does not work or (b) there is a bug in the compiler because his simulations keep blowing up. It does and there isn't.

```
PROCEDURE Widrow_Hoff (Correct_g, Actual_g, F: Vector;
                       VAR Delta_A: Matrix);

  VAR Difference_vector,
      Weighted_vector  : Vector;
      Learning_constant: REAL;      {Chose appropriate value.}

  PROCEDURE Scalar_times_vector (S: REAL; A: Vector; VAR B: Vector);
    {Multiplies A times S, Product is B.}

    VAR I: INTEGER;
    BEGIN
    FOR I:= 1 TO Dimensionality DO B [I]:= S * A [I];
    END;

  PROCEDURE Subtract_vectors (A, B: Vector; VAR Difference: Vector);
    {Subtracts B from A. The Difference Vector is Difference.}

    VAR I: INTEGER;
    BEGIN
    FOR I:= 1 TO Dimensionality DO Difference [I]:= A [I] - B [I];
    END;

  PROCEDURE Outer_product (A, B: Vector; VAR C: Matrix);
    VAR I,J: INTEGER;
    { Note:  We want the outerproduct procedure call to be in the form
      gf transpose format.  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;
```

```
BEGIN {Procedure Widrow_Hoff.}
Subtract_vectors (Correct_g, Actual_g, Difference_vector);
Scalar_times_vector (Learning_constant,Difference_vector,Weighted_vector);
Outer_product (Weighted_vector, F, Delta_A);
END;  {Procedure Widrow_Hoff.}
```