Introduction to Neural Networks
U. Minn. Psy 5038
Spring, 1999

Gradient descent
Regression, least squares, and Widrow-Hoff

Introduction

In this notebook, we will introduce gradient descent learning in the simple context of finding the weights of a linear matrix transformation. This will lead us to the Widrow-Hoff learning rule. Which, in turn, will provide the introduction to the generalization of this rule to multiple layer networks with smooth non-linear squashing functions--the error back-propagation algorithm.

By treating the linear case first, we will be able to see how the Widrow-Hoff learning rule relates to classic problems of statistical regression. This will provide a useful point of view and intuitions to understanding the more complicated non-linear feedforward networks.

Regression, PseudoInverse, and Widrow-Hoff learning

■ Introduction

We have been studying a linear matrix model of memory based on the storage of connection weights that follow a particular Hebbian rule. We have studied the "psychology" of some operations of linear algebra and have seen some interesting parallels to human memory, such as interference and pattern reconstruction.

In this section, we continue the study of linear models of memory. However, we are going to view memory from the point of view of statistical regression. The idea is to treat memory as an attempt to fit past input/output associations into a model that can be used both for recall and for generalization. For the problem of regression in statistics, given a set of vector inputs $\{\mathbf{x}_i\}$, and a set of corresponding vector outputs $\{\mathbf{y}_i\}$, one tries to find a transformation **W** that will map **x->y** as closely as possible over the data available. For this we need a model for **W**, and a measure of goodness of fit. We will assume below a linear model for **W**, so for the discrete case, **W** is a matrix. Our measure of goodness of fit will be the sum of the squared differences between predicted output and actual output. In this linear case, this is least squares regression.

■ Least squares regression - linear models

The idea behind least squares regression is given a set of N training pairs {$\mathbf{x}_i$,$\mathbf{y}_i$}, where i runs from 1 to N, we would like to find a function that given an input **x**, the function approximates well the output **y**. If the function reproduces the association between input and output that it has seen before, this is like "remembering". But in addition, regression generalizes. So novel input values get mapped to predicted outputs based on past "experience". We've already seen how linear heteroassociative learning does this.

Recall that in the last lecture, we distinguished between *interpolation* and *approximation*. Interpolation finds a fit to the data points in such a way that the data are matched perfectly. An example would be drawing lines connecting data points on a graph. Approximation doesn't necessarily exactly fit the data points, but should just come close. We are going to study approximation.

An example of linear regression

A fundamental assumption behind any learning system is that there is an underlying structure to the data--the relationship between associative pairs is not arbitrary. When trying to understand how a relationship can be learned between a set of two patterns, it is essential to have some understanding of the structure of the relationship. For example, one shouldn't try to fit a straight line to data when there is evidence to indicate that the underlying process is quadratic. We will study this more when we learn about the "bias/variance dilemma". So let us assume that the data have an underlying structure that we are going to try to discover or approximate using **W**. We will study a simple "toy" problem that has the following very specific structure. The inputs are randomly located points on a 2D plane, and the outputs are heights above these points. The outputs lie approximately on a planar surface that runs through the origin and whose orientation is characterized by two parameters, **a** and **b**.

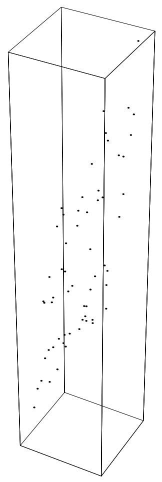
It may seem like overkill, but we are going to estimate **W** for the same set of data in 4 (yes, 4!) different ways. The first two are drawn from standard linear algebra (a least squares solution using transpose and inverse, and the second using the pseudoinverse). The third introduces the method of "gradient descent" which we will use later in a number of contexts. The fourth method is the most relevant to neural network theory--we estimate **W** using a biologically plausible learning rule (the Widrow-Hoff rule).

■ Synthetic training pairs

```
rsurface[a_,b_] :=
N[Table[{x1=1 Random[],x2= 1 Random[],
             a x1 + b x2 + .5 Random[] - 0.25},{60}],2];
```

```
data = rsurface[2,3];
```

```
Show[Graphics3D[Map[Point,data]]];
```



```
Outdata = Table[{data[[i,3]]},{i,1,Length[data]}];
Indata = Table[{data[[i,1]],data[[i,2]]},{i,1,Length[data]}];
```

■ Least squares regression to find W

So let's assume we want to find a matrix **W** that will come close to reproducing values **y**, given inputs **x**. Of course, because we generated the data, we know the underlying structure and what the matrix W should be. It should be a 1x2 matrix = **{{2,3}}**. But let's assume we are ignorant, and want to discover the weights from **Outdata** and **Indata**.

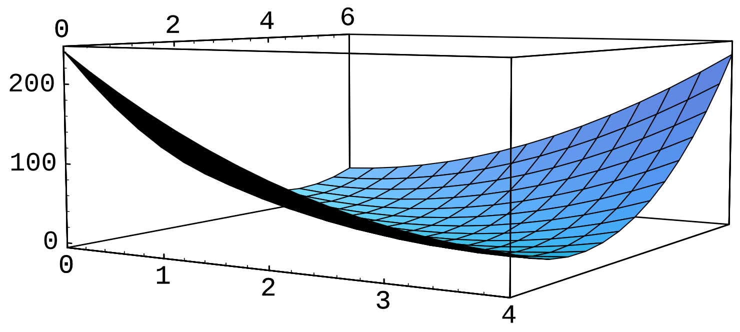
In least squares regression, we try to find the values of the matrix that will minimize e(**W**).

$$e(\mathbf{W}) = \sum_{i=1}^{N} \left| \mathbf{y}_i - \mathbf{W}\mathbf{x}_i \right|^2$$

Most of the time our error function will be over a very high dimensional space. However, it is useful to get an intuition for the problem in a low dimensional space. We can actually get a picture of the total error, **e(W)**, for our synthetic data as a function of the weight parameters **w1** and **w2**:

```
eW[w1_,w2_] :=
Sum[(
Outdata[[i]]-{w1,w2}.Indata[[i]]).(Outdata[[i]]-{w1,w2}.Indata[[i]]),
        {i,Length[Indata]}]
```

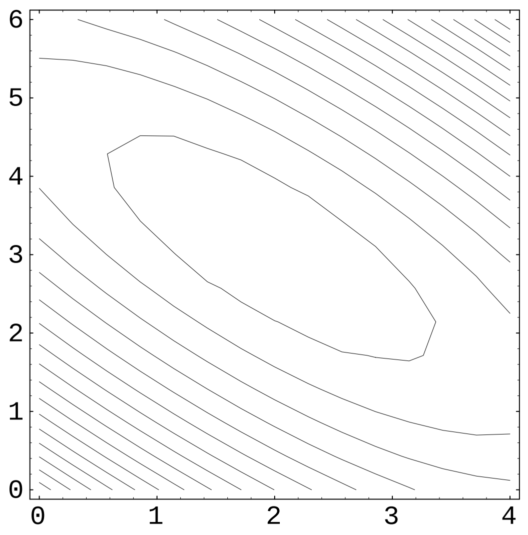
```
g=Plot3D[eW[w1,w2], {w1,0,4},{w2,0,6},
        ViewPoint->{1.780, -2.861, 0.312}];
```



Note that because our input data are 2-element vectors, and our output or target values are 1-D, the matrix W is not square--it has 1 row and 2 columns. So we represent **W** above as a vector.

The minimum appears to be near **{2,3}**. It is easier to see whether there is a minimum or not using **ContourPlot[]**.

```
ContourPlot[eW[w1,w2], {w1,0,4},{w2,0,6},Contours->16,
            ContourShading->False];
```



Note: For*Mathematica*, on a Macintosh, you can get coordinates from a plot by: 1) Selecting the plot; 2) Hold down the Apple key () while moving the mouse over the plot. The lower left corner shows the coordinates. You can even (temporarily) mark points on the graph. If you go to the Edit menu, and Copy. These coordinates are copied to the Clipboard. Then go to a cell in your notebook and Paste. Here is a guess for the minimum:

```
{1.999798, 3.100128}
```

We can find the exact location of the minimum by finding **W** for which the gradient of **e** is zero. Although there are a lot of indices to worry about, the result can be written very concisely in terms of vector outerproducts, inversion, and matrix multiplication:

$$e(\mathbf{W}) = \sum_{i=1}^{N} \left| \mathbf{y}_i - \mathbf{W}\mathbf{x}_i \right|^2$$

$$\frac{\partial e}{\partial \mathbf{W}} = -2 \sum_{i=1}^{N} (\mathbf{y}_i - \mathbf{W}\mathbf{x}_i)\mathbf{x}_i^T = 0$$

$$\sum_{i=1}^{N} \mathbf{y}_i \mathbf{x}_i^T - \mathbf{W} \sum_{i=1}^{N} \mathbf{x}_i \mathbf{x}_i^T$$

$$W = \sum_{i=1}^{N} \mathbf{y}_i \mathbf{x}_i^T \left(\sum_{i=1}^{N} \mathbf{x}_i \mathbf{x}_i^T \right)^{-1}$$

Let's try it on our data

```
sumX = Sum[Outer[Times,Indata[[i]],Indata[[i]]],
            {i,Length[Indata]}];
sumYX = Sum[Outer[Times,Outdata[[i]], Indata[[i]]],
            {i,Length[Indata]}];
W = sumYX.Inverse[sumX]
```

```
{{1.93658, 3.04565}}
```

The values for **W** come close to what we would expect from the structure of our data, a plane with parameters **{2,3}**.

■ Pseudoinverse to find W

There is another way of solving the linear least squares regression that uses the pseudoinverse of matrix.

Define $\mathbf{X}^*$ to be the *pseudoinverse* (sometimes called the *generalized inverse*) of the matrix **X**:

$$\mathbf{X}^* = \left(\mathbf{X}^T \mathbf{X}\right)^{-1} \mathbf{X}^T$$

or as:

$$\mathbf{X}^* = \mathbf{X}^T \left(\mathbf{X}\mathbf{X}^T\right)^{-1}$$

Let's take all of the input vectors **x**, and arrange them as columns in a matrix **X**:

$$\mathbf{X} = \begin{pmatrix} x_1^1 & x_1^2 & \ldots & x_1^N \\ x_2^1 & x_2^2 & & x_2^N \end{pmatrix}$$

Now do the same for the **y**'s:

$$\mathbf{Y} = \begin{pmatrix} y_1 & y_2 & y_3 \ldots y_N \end{pmatrix}$$

And what is the matrix that maps the x's to the y's with least squared error? It is $\mathbf{X}^* \mathbf{Y}$:

```
Inverse[Transpose[Indata].Indata].Transpose[Indata].Outdata
```

```
{{1.8492}, {3.0852}}
```

For a square matrix **X**, the inverse of **X** is chosen so that $\mathbf{XX}^{-1}$ is equal to the identity matrix, **I**. The pseudoinverse, **X***, of a rectangular matrix **X** is chosen so that **XX*** is close to the identity matrix in the sense that the sum of the squares of all of the entries of **XX*-I** is least. The **PseudoInverse[]** function is built into *Mathematica*.

```
PseudoInverse[Indata].Outdata
```

```
{{1.93658}, {3.04565}}
```

It can be shown that **PseudoInverse[X].X** is the identity matrix. We won't prove it here, but we can verify that it is the case with our data:

```
Chop[PseudoInverse[Indata].Indata]//MatrixForm
```

```
1.   0
0    1.
```

Question: What are the dimensions of the above PseudoInverse?

■ Gradient descent

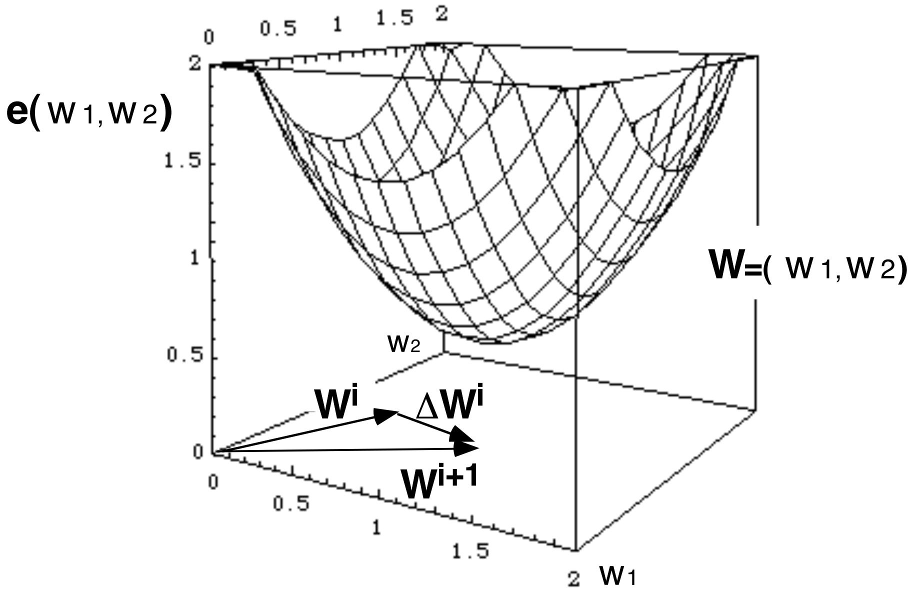
Let's go back to the original global error function that we used above for standard linear least squares regression. There we found the minimum by setting the error to zero, and then solving for the weights.

There is another way of finding the minimum of the error function that is more general in the sense that it will can be used when the error function is much more complicated, and there is no linear solution.

The idea is to start off at some location, W_0 in weight space (which is just a guess), and iteratively move towards the minimum by always taking a step downhill. The downhill direction is given by the gradient of the error function.

$$\frac{d\mathbf{W}}{dt} = -\frac{\partial e}{\partial \mathbf{W}} = -\nabla e$$

$$\mathbf{W}^{i+1} = \mathbf{W}^{i} - \eta \left.\frac{\partial e}{\partial \mathbf{W}}\right|_{\mathbf{W}^{i}}$$



From the expression for the gradient which we wrote earlier in terms of outer products, we can obtain:

$$\frac{\partial e}{\partial \mathbf{W}} = -2\left(\sum_{i=1}^{N} \mathbf{y}_i \mathbf{x}_i^T - W \sum_{i=1}^{N} \mathbf{x}_i \mathbf{x}_i^T\right)$$

```
eWgradient[wg_] :=
             Sum[Outer[Times,Outdata[[i]], Indata[[i]]], {i,Length[
Indata]}]-
             wg.Sum[Outer[Times,Indata[[i]],Indata[[i]]],
                 {i,Length[Indata]}];
```

```
i=0; wg={{0,0}}; eta = .05; wglist = {};
T[wg_] := wg + eta eWgradient[wg];
```

```
w1 = Nest[T,wg,20]
```

```
{{1.93909, 3.04261}}
```

■ "Brain-style" learning: Iterative Widrow-Hoff learning to estimate W

So far so good. But there are a couple of problems. First, this is a course in Neural Networks, and we are interested in brain-style computation. Based on what we think we know about neurons, how could the brain compute transposes, do matrix inversion and multiplication? Further we don't seem to gather information on a whole set of training pairs, and then suddenly build a memory matrix. We learn incrementally, trial by trial. The second problem is purely computational. What if the dimensionality of the vectors is really big? It is computationally expensive to invert large matrices. The above gradient descent procedure avoids the problem of inverting large matrices, but it involved computing a global error term over all the training pairs. We would like a method which would learning the regression map without having to store all the training pairs, and then compute this global error term.

Can we find **W** in such a way so as to be biologically plausible, *and* avoid having to invert a large matrix? The basic idea behind Widrow-Hoff learning is to update **W** iteratively with each new training pair. Let's start off with an arbitrary **W**, find out which direction we would have to go in weight space to reduce the discrepancy between what **W** tells us **x** should map to and what it actually is, namely **y**.

$$e(\mathbf{W}) = \left|\mathbf{y}_i - \mathbf{W}\ \mathbf{x}_i\right|^2$$

$$\frac{\partial e}{\partial \mathbf{W}} = -2\left(\mathbf{y}_i - \mathbf{W}\ \mathbf{x}_i\right)\mathbf{x}_i^T$$

$$\frac{\partial e}{\partial \mathbf{W}} = -\frac{d\mathbf{W}}{dt}$$

$$\mathbf{W}^{i+1} = \mathbf{W}^i + \eta_i\left(\mathbf{y}_i - \mathbf{W}^i\mathbf{x}_i\right)\mathbf{x}_i^T$$

Let's try out this update rule on our synthetic training pairs.

```
<<Graphics`MultipleListPlot`
```

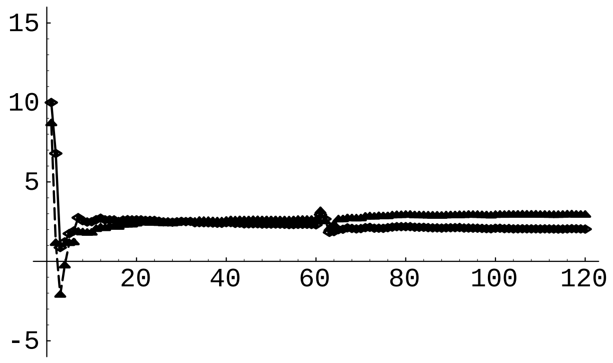
```
ww1 = {{0,0}}; ww1list = {}; ww2list = {};
```

```
i=0;eta = 5;
While[i<Length[data],
    ++i;
    in = {data[[i,1]],data[[i,2]]} ; out = {data[[i,3]]};
    ww1 = ww1 + (eta/i) Outer[Times,(out - ww1.in),in];
    ww1list = Append[ww1list,ww1[[1,1]]];
    ww2list = Append[ww2list,ww1[[1,2]]];
];
ww1
```

```
{{2.02737, 2.92272}}
```

You may have to run through the above loop several times before reaching stable convergence. We can plot up the two weights as a function of the iteration number to see how the Widrow-Hoff rule for weight modification eventually leads to two stable weights:

```
MultipleListPlot[ww1list, ww2list, PlotRange->{-6,16},
            AxesOrigin->{0,0},PlotJoined->True];
```



■ Memory recall

We've seen several ways of finding the weights of a matrix that will approximately reproduce an output, given an input it has seen before. Let's try it out.

So in order to "recall" a response, from an input **Indata[[6]]**, we run it through the "network" memory matrix **w1**:

```
w1.Indata[[22]]
```

```
{3.27156}
```

And we can check to see how well it recalls:

```
Outdata[[22]]
```

```
{3.5}
```

One property of the regression model of memory is that it also generalizes. For example, **{11,15}** wasn't in the training set, but the expected output is:

```
w1.{11,15}
```

```
{65.5427}
```

The network has "learned" a surface: **z = 1.9x + 3.07y** through the points specified in the training set. The main point is that the linear associator will try to fit a plane (or hyperplane) through the data. If the data do not fit that model, then the memory and generalization will not be good.

An obvious generalization is to fit hypersurfaces, rather than hyperplanes. And that is the direction we will head. But first let us look at linear regression from a point of view that you may not have thought of before.

Underconstrained problems and redundancy

This example is very similar to the preceding example, except that we are going to try to learn 2D responses from 1D stimuli. At first this doesn't seem to make sense, because the mapping from 1D to 2D in general would be expected to be underconstrained by the data. But let's try it with some synthetic data whose generation process we'll keep hidden for now.

■ Synthetic data -- don't look

```
r3Dline[a_,b_] := N[Table[{x1=1 Random[], a x1,
            b x1 + .5 Random[] - 0.25},{30}],2];
```

```
data = r3Dline[2,3];
```

```
Indata = Table[{data[[i,1]]},{i,1,Length[data]}];
Outdata = Table[{data[[i,2]],data[[i,3]]},{i,1,Length[data]}];
```

So the input data is a list of 1D scalars, and the output data a list of 2D vectors. Let's apply the Widrow-Hoff algorithm to learn the relationship between the input stimuli, **Indata**, and the output responses, **Outdata**.

■ Iterative Widrow-Hoff learning

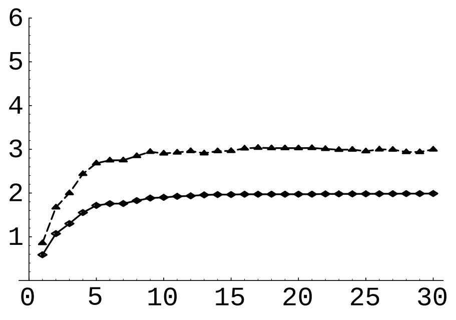
```
i=0; w1 = {{0},{0}}; w1list = {}; w2list = {}; eta = 0.4;
Dimensions[w1]
```

```
{2, 1}
```

```
While[i<Length[data],
    ++i;
    out = {data[[i,2]],data[[i,3]]} ; in = {data[[i,1]]};
    w1 = w1 + eta Outer[Times,(out - w1.in),in];
    w1list = Append[w1list,w1[[1]][[1]]];
    w2list = Append[w2list,w1[[2]][[1]]];
];
w1
```

```
{{1.98848}, {2.99641}}
```

```
MultipleListPlot[w1list, w2list, PlotRange->{0,6},
            AxesOrigin->{0,0},PlotJoined->True];
```



■ Pseudoinverse solution

We can calculate the memory matrix using the **PseudoInverse** in this case too:

```
matmem = Transpose[PseudoInverse[Indata].Outdata]
```

```
{{2.}, {3.05258}}
```

■ Recall

Let's check out a few values to see how well the memory matrix can recall a 2 dimensional output, given a one dimensional input.

```
matmem.Indata[[5]]
matmem.Indata[[13]]
matmem.Indata[[28]]
```

```
{1.91066, 2.91622}
```

```
{1.81768, 2.7743}
```

```
{1.40065, 2.13779}
```

```
Outdata[[5]]
Outdata[[13]]
Outdata[[28]]
```

```
{1.9, 3.}
```

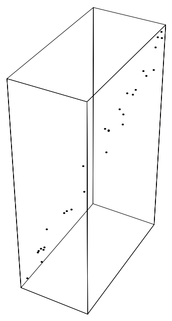
```
{1.8, 2.5}
```

```
{1.4, 1.9}
```

Revealing the underlying structure

So why does it work to learn to predict a 2D value from a 1D input? The reason, of course, is that the underlying structure of the output data is very redundant or highly constrained, and in fact lies close to a straight line in 3-space.

```
Show[Graphics3D[Map[Point,data]]];
```



Here is the code that generated our synthetic data:

```
r3Dline[a_,b_] := N[Table[{x1=1 Random[], a x1,
            b x1 + .5 Random[] - 0.25},{30}],2];
```

```
data = r3Dline[2,3];
```

It produced the coordinates of a noisy line in 3-space.

The bias/variance dilemma

The problem in general is how to choose the function that both remembers the relationship between **x** and **y**, and generalizes with new values of **x**. The function has to be parameterized in some way that allows easy computation. What form should the function take? At first one might think that it should be as general as possible to allow all kinds of maps. For example, if one is fitting a curve, you might wish to use a very high-order polynomial. There is a draw back to flexibility. We can get drastically different fits for different sets of data that are randomly drawn from the same underlying process. On the other hand, if the function is restrictive, (e.g. straight lines through the origin), then we will get similar fits for different data sets, because all we have to adjust is one parameter--the slope. The problem here, is that the fit is only good if the underlying process is in fact a straight line through the origin. Statisticians refer to this problem as the *bias/variance* dilemma. To sum up, lots of parameter flexibility has the benefit of fitting anything, but at the cost of sensitivity to variability in the data--there is *variance* in the fits found over multiple training sets of a fixed size. A fit with very few parameters is not sensitive to the inevitable variability in the training set, but can give large constant errors or *bias* if the data do not match the underyling model. There is no general way of getting around this problem, and neural networks are no exception. Later we will generalize linear regression to non-linear fits when the error back-propagation method is introduced. Because back-propagation models can have lots of hidden layers with many units and weights, they form a class of very flexible approximators. These models can show high variability in their fits from one data set to the next, even when the data comes from the same underlying process. The linear associator is very restrictive, so would show high bias. One has to study the problem one is trying to model (independently of the neural network parameters if possible) and choose the appropriate network model to do the fit. Because the straight line, or in higher dimensions, a linear model is, in a mathematical sense, the "simplest", it does make good sense to try it first. Later on, one can complicate matters if it turns out that the data one is trying to fit is non-linear.

Much current research in data modeling and pattern learning is devoted to the problem of how to achieve good generalization after learning from finite examaples. Good models combine some faithfulness to the data seen, while minimizing the complexity of the model used to fit the data.

References

Geman, S., Bienenstock, E., & Doursat, R. (1992). Neural networks and the bias/variance dilemma. Neural Computation, 4(1), 1-58.