Introduction to Neural Networks
U. Minn. Psy 5038

Disriminant functions
Gradient descent
Regression, least squares, and Widrow-Hoff

Introduction

Last time

- **Turning linear networks into classifiers with a hard threshold**
- **Perceptrons, perceptron learning rule**

Today

- **Discriminant functions: Linear part of a Perceptron-unit is a linear discriminant**
- **Linear regression & brain-style learning**

Discriminant functions

Let's build our geometric intuitions of what a simple perceptron unit does by viewing it from a more formal point of view. Perceptron learning is an example of nonparametric statistical learning, because it doesn't require knowledge of the underlying probability distributions generating the data (such distributions are characterized by a relatively small number of

"parameters", such as the mean and variance of a Gaussian distribution). Of course, how well it does will depend on the generative structure of the data. Much of the material below is covered in Duda and Hart (1978).

Linear discriminant functions: Two category case

A discriminant function, g(x) divides input space into two category regions depending on whether g(x)>0 or g(x)<0. (We've switched notation, x=**f**). The linear case corresponds to the simple perceptron unit we studied earlier:

$$g(\mathbf{x}) = \mathbf{w}.\mathbf{x} + w_0 \tag{1}$$

where w is the weight vector and w_0 is the threshold (sometimes called bias, although this "bias" has nothing to do with statistical "bias").

Discriminant functions can be generalized, for example to quadratic decision surfaces:

$$g(\mathbf{x}) = w_0 + \sum_{i=1} w_i x_i + \sum_{i=1}\sum_{j=1} w_{ij} x_i x_j \tag{2}$$

We've seen how g(x)=0 defines a decision surface which in the linear case is a hyperplane. Suppose x_1 and x_2 are points sitting on the hyperplane, then their difference is a vector lying in the hyperplane

$$\begin{aligned} \mathbf{w}.\mathbf{x}_1 + w_0 &= \mathbf{w}.\mathbf{x}_2 + w_0 \\ \mathbf{w}.(\mathbf{x}_1 - \mathbf{x}_2) &= 0 \end{aligned} \tag{3}$$

so the weight vector w is normal to any vector lying in the hyperplane. Thus w determines how the plane is oriented. The normal vector w points into the region for which g(x)>0, and -w points into the region for which g(x)<0.

Let x be a point on the hyperplane. If we project x onto the normalized weight vector x.w/|w|, we have the normal distance of the hyperplane from the origin equal to:

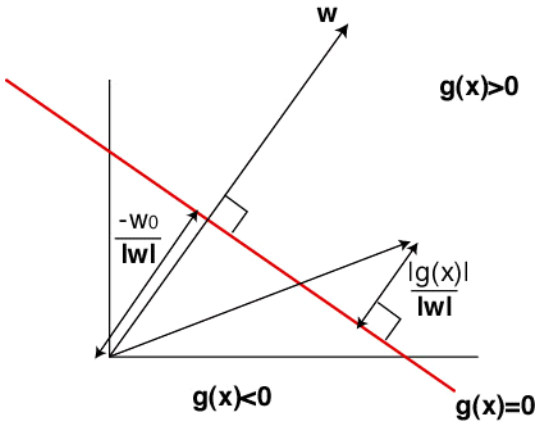
$$\mathbf{w}.\mathbf{x} / |\mathbf{w}| = -w_0 / |\mathbf{w}| \tag{4}$$

Thus, the threshold determines the position of the hyperplane.

One can also show that the normal distance of x to the hyperplane is given by:

$$g(\mathbf{x}) / |\mathbf{w}| \tag{5}$$

So we've seen that: 1) disriminant function divides the input space by a hyperplane decision surface; 2) The orientation of the surface is determined by the weight vector w; 3) the location is determined by the threshold w_0 ; 4) the discriminant function gives a measure of how far in input vector is from the hyperplane.

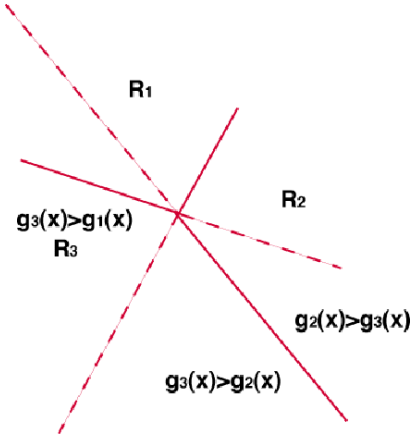


Multiple classes

Suppose there are c classes. There are a number of ways to define multiple class discriminant rules. One way that avoids undefined regions is:

$$g_i(x) = w_i.x + w_{i0},\ i = 1, \ldots, c \qquad (6)$$

Assign x to the ith class if : $g_i(x) > g_j(x)$ for all $j \neq i$. (7)



It can be shown that this classifier partitions the input space into simply connected convex regions. This means that if you connect any two feature vectors belonging to the same class by a line, all points on the line are in the same class. Thus this linear classifier won't be able to handle problems for which there are disconnected clusters of features that all belong to the same class. Also, from a probabilistic perspective, if the underlying generative probability model for a given class has multiple modes, this linear classifier won't do a good job either.

Preview: Fisher's linear "discriminant"

Later, when we consider the problem of dimensionality reduction, we will take another look at hyperplanes. But here the idea will be to find hyperplanes onto which we can project our input data, and from there divide up the hyperplane into decision regions. The idea is that the original input space may be impractically huge, but if we can find a subspace (hyperplane) that preserves the distinctions between categories as well as possible, we can make our decisions in smaller space. We will derive the Fisher linear "discriminant".

This is closely related to the psychology idea of finding "distinctive" features. E.g. consider bird identification. If I want to discriminate cardinals from other birds in my backyard, I can make use of the fact that (males) cardinals may be the only birds that are red. So even tho' the image of a bird can have lots of dimensions, if I project the image on to the "red" axis, I can do fairly well with just one number. How about male vs. female human faces?

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.

We've learned that linear networks can be configured to be supervised or unsupervised learning devices. But linear mappings are severely limited in what they can compute. The simple perceptron unit added a threshold. The network then makes discrete decisions. The kinds of classifications, however, are still limited by the linearity of the decision surface, i.e. by the fact that it is a hyperplane. More complex networks can be built by adding layers, but then we have another problem: how can we learn the weights in this more complicated setting?

In this section, we return to 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.

We will return to strictly linear networks (no threshold) in order to introduce techniques (gradient descent learning in the context of finding the weights of a linear matrix transformation) and concepts that will generalize to non-linear networks. Gradient descent will lead us to the Widrow-Hoff learning rule. 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, 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.

More terminology

Consider s**upervised learning**. We have a "training set" $\{\mathbf{f}_i,\mathbf{t}_i\}$ representing inputs $\mathbf{f}_i$, and target outputs $\mathbf{t}_i$. The training set in some sense "samples" the larger space of possible input/output pairs $\{\mathbf{f},\mathbf{t}\}$. We would like to learn a general mapping: T: **f** -> **g** in such a way that T is a good fit to the training data (i.e. **g** is close to **t**), and generalizes well to novel inputs. The set of target data is the feedback for the"teacher". The feedback can say whether the mapping is correct or not. Or the feedback can provide information as to how far off the map T's prediction of f (i.e. T[**f**]) is from **t**. After training, one can require that T always maps members of the training set to exactly the target members, and generalizes appropriately for other inputs. This means that the learning should be *consistent*. ***Interpolation*** is between data points on a graph is an example of consistent learning. An example would be drawing lines connecting data points on a graph. Or, we may require that the T maps the original members of the training set to outputs **g**, that are close to the original targets **t**. Linear regression is a an example of ***approximation*** learning. The linear associator was our first example of supervised learning. Approximation doesn't necessarily exactly fit the data points, but should just come close. We are going to study approximation.

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.

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 important 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 generative 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).

■ Generative model: Synthetic training pairs

Let x1 and x2 be the (scalar) inputs, and z be the (scalar) output:

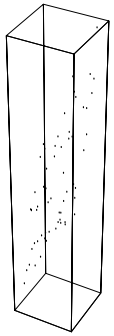
```
{x1, x2} → z,

and assuming the mapping is a plane through the origin and additive noise        (8)

z = a x1 + b x2 + noise
```

We will want to learn a and b from the training pairs: {{x1,x2},z}.

```
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**).

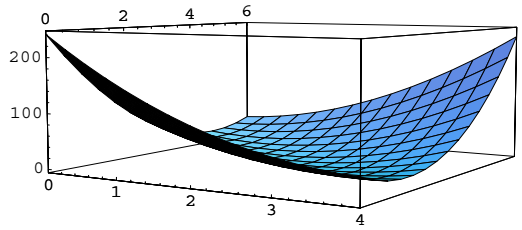
$$e(\mathbf{W}) = \sum_{i=1}^{N} \left| \mathbf{y}_i - \mathbf{W}\mathbf{x}_i \right|^2$$

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]}]
```



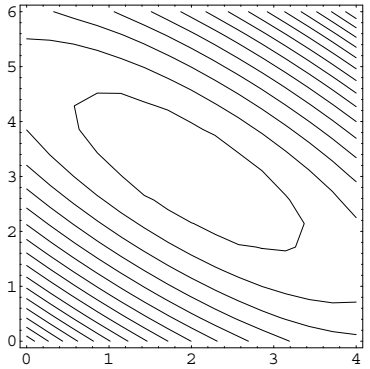
```
g=Plot3D[eW[w1,w2], {w1,0,4},{w2,0,6},
        ViewPoint->{1.780, -2.861, 0.312}];
```



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:

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

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

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

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

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**:

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

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

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

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 $\mathbf{X}\mathbf{X}^{-1}$ 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

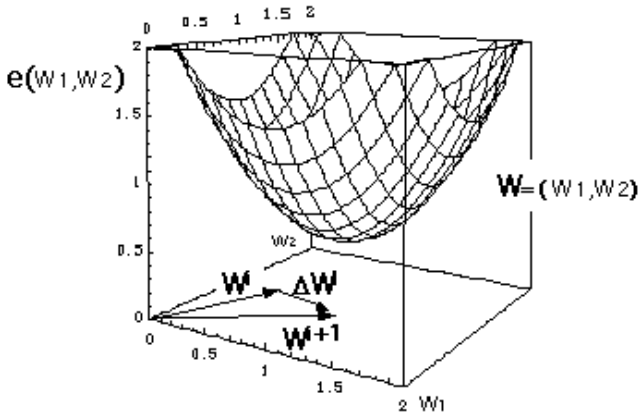
Let's go back to the original global error function that we used above for standard linear least squares regression. There we found the minimum by setting the error to zero, and then solving for the weights.

There is another way of finding the minimum of the error function that is more general in the sense that it 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 an expression for **ΔW**, with η taking the place of Δt:

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

```
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, suppose 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 learn the regression map without having to store all the training pairs with the accompanying computation of a global error term. Instead, we'd like to compute an error term incrementally, trial-by-trial.

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`
```

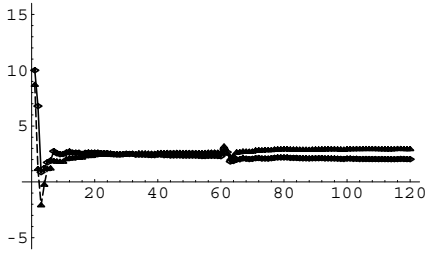
```
ww1 = {{0,0}}; ww1list = {}; ww2list = {};
```

```
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

```
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]
```

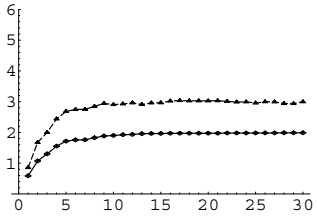
```
{2, 1}
```



```
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.}
```

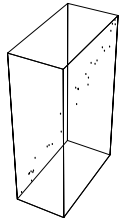
```
{1.8, 2.5}
```

```
{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 generative model for our synthetic data:

x1 -> {x2,z}, and where assuming the form is a straightline through the origin with some additive noise, the output {x2,z}is given by:

{x2, z} = {a x1, b x1 + noise}

We learned the parameters {a,b} from training pairs of inputs and outputs: {x1, {{x2, z}}}

```
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.

References

Bishop, C. M. (1995). Neural Networks for Pattern Recognition. Oxford: Oxford Univeristy Press.

Duda, R. O., & Hart, P. E. (1973). Pattern classification and scene analysis. New York.: John Wiley & Sons.

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