Linear Models

In this section, we review some basics of modeling via linear algebra: finding a line of best fit, Hebbian learning, pattern classification.

Best Fitting Line

In this section, we examine the simplest case of fitting data to a function. We are given n ordered pairs of data:

$$X = \{x_1, x_2, \ldots, x_n\} \quad Y = \{y_1, y_2, \ldots, y_n\}$$

We wish to find the best linear relationship between X and Y. But what is "best"? It depends on how you look at the data, as described in the next three sections.

Sum of Absolute Values

Let y be a function of x. Then we are trying to find b_0 and b_1 so that

$$y = b_0 x + b_1$$

best describes the data. If the data were perfectly linear, then this would mean that:

$$\begin{array}{rcl} y_1 &=& b_0 x_1 + b_1 \\ y_2 &=& b_0 x_2 + b_1 \\ & \vdots & \\ y_n &=& b_0 x_n + b_1 \end{array} \quad \Rightarrow \quad \mathbf{y} = \begin{bmatrix} & 1 \\ \mathbf{x} & 1 \\ & \vdots \\ & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \end{bmatrix} \quad \Rightarrow \quad A\mathbf{b} = \mathbf{y}$$

However, most of the time the data is not actually, *exactly* linear, so that the values of y don't match the line: $b_0 x + b_1$. There are many ways of expressing the error- Below, we look at three ways.

The first way is to define the error as the following:

$$E_1 = \sum_{k=1}^{n} |y_k - (b_0 x_k + b_1)|$$

Exercises

1. Let the data be given by $(0, 1), (1, 3), (2, 5)$. Use Maple to plot the error function. To make it easier to plot, you might re-write the unknowns so the function is:

   $$E = |1 - b| + |3 - (m + b)| + |5 - (2m + b)|$$

   Note that you must plot this in three dimensions.

2. E_1 is a function of two variables, b_0 and b_1. What is the difficulty in determining the minimum[1] error using this error function?

[1] We can find the minimum using Linear Programming; something we'll discuss later.

The Usual Method: Least Squares

The usual method of defining the error is to sum the squared errors up:

$$E_{\text{se}} = \sum_{k=1}^{n} (y_k - (b_0 x_k + b_1))^2$$

Why is it appropriate to use this error instead of the other error?

Exercises

1. E_{se} is a function of b_0 and b_1, so the minimum value occurs where

   $$\frac{\partial E_{\text{se}}}{\partial b_0} = 0 \quad \frac{\partial E_{\text{se}}}{\partial b_1} = 0$$

   This leads to the system of equations: (the summation index is 1 to n)

   $$\begin{aligned} b_0 \sum x_k^2 + b_1 \sum x_k &= \sum x_k y_k \\ b_0 \sum x_k + b_1 n &= \sum y_k \end{aligned}$$

2. Show that this is the same set of equations you get by solving the normal equations, $A^T A\mathbf{b} = A^T\mathbf{y}$, assuming A is full rank.

3. Write a Matlab routine that will take a $2 \times n$ matrix of data, and output the values of $b_0 = m$ and $b_1 = b$ found above. The first line of code should be:

   ```
   function [m,b]=Line1(X)
   ```

   and save as `Line1.m`.

Treat the variables independently

The last case is where we treat x and y independently, so that we don't assume that one is a function of the other.

1. Show that, if $ax + by + c = 0$ is the equation of the line, then the distance from (x_1, y_1) to the line is

   $$\frac{|ax_1 + by_1 + c|}{\sqrt{a^2 + b^2}}$$

   which is the size of the orthogonal projection of the point to the line. This is actually problem 53, section 11.3 of Stewart's Calculus text, if you'd like more information.

   (HINT: The vector $[a, b]^T$ is orthogonal to the line $ax + by + c = 0$. Take an arbitrary point P on the line, and project an appropriate vector to $[a, b]^T$.)

   Conclude that the error function is:

   $$E = \sum_{k=1}^{n} \frac{(ax_k + by_k + c)^2}{a^2 + b^2}$$

2. Draw a picture of the error in this case, and compare it graphically to the error in the previous 2 exercises.

3. The optimimum value of E occurs where $\frac{\partial E}{\partial c} = 0$. Show that if we mean subtract X and Y, then we can take $c = 0$. This leaves only two variables.

4. Now our error function is:
$$E = \sum_{k=1}^{n} \frac{(ax_k + by_k)^2}{a^2 + b^2}$$
Show that we can transform this function (with appropriate assumptions) to:
$$E = \sum_{k=1}^{n} \frac{(x_k + \mu y_k)^2}{1 + \mu^2}$$
(for some μ), and conclude that E is a function of one variable.

5. Now the minimum occurs where $\frac{dE}{d\mu} = 0$. Compute this quantity to get:
$$\mu^2 A + \mu B + C = 0$$
where A, B, C are expressions in $\sum x_k y_k$, $\sum x_k^2$, $\sum y_k^2$. This is a quadratic expression in μ, which we can solve. Why are there (possibly) 2 real solutions?

6. Write a Matlab routine `[a,b,c]=Line2(X)` that will input a $2 \times n$ matrix, and output the right values of a, b, and c.

7. Try the 2 different approaches on the following data set, which represents heights (in inches) and weight (in lbs.) of 10 teenage boys. (Available in `HgtWgt.mat`)

X	69	65	71	73	68	63	70	67	69	70
Y	138	127	178	185	141	122	158	135	145	162

Plot the data with the 3 lines. What do the 3 approaches predict for the weight of someone that is 72 inches tall?

8. Do the same as the last exercise, but now add the data point $(62, 250)$. Compare the new lines with the old. Did things change much?

The Median-Median Line:

The median of data is sometimes preferable to the mean, especially if there exists a few data points that are far different than "most" data.

1. **Definition:** The *median* of a data set is the value so that exactly half of the data is above that point, and half is below. If you have an odd number of points, the median is the "middle" point. If you have an even number, the median is the average of the two "middle" points. Matlab uses the `median` command.

2. **Exercise:** Compute (by hand, then check with Matlab) the medians of the following data: $\{1, 3, 5, 7, 9, 11\}$

   The motivation for the median-median line is to have a procedure for line fitting that is not as sensitive to "outliers" as the 3 methods in the previous section.

Median-Median Line Algorithm

- Separate the data into 3 equal groups (or as equal as possible). Use the $x-$axis to sort the data.
- Compute the median of each group (first, middle, last).
- Compute the equation of the line through the first and last median points.
- Find the vertical distance between the middle median point and the line.
- Slide the line 1/3 of the distance to the middle median point.

3. **Exercise:** The hardest part of the Matlab code will be to partition the data into three parts. Here are some hints:

   - If we divide a number by three, we have three possible remainders: $0, 1, 2$. What is the most natural way of seperating data in these three cases (i.e., if we had 27, 28 or 29 data points)?
   - Look at the Matlab command `rem`. Notice that:

     ```
     rem(27,3)=0   rem(28,3)=1   rem(29,3)=2
     ```

   - The command to sort: `[s,index]=sort(x)`. For example,

     ```
     >> a=[3 2 4 1];
     >> [a2,idx]=sort(a)
     a2 =
          1     2     3     4
     idx =
          4     2     1     3
     ```

     Notice that $a_2(j) = a(idx(j))$. We can therefore sort x first, then sort y according to the index for x.

4. **Exercise:** Try this algorithm on the last data set, then add the new data point. Did your lines change as much?

5. **Exercise:** Consider the following data set [1] which relates the index of exposure to radioactive contamination from Hanford to the number of cancer deaths per $100,000$ residents. We would like to get a relationship between these data. Use the four techniques above, and compare your answers. Compute the actual errors for the first three

types of fits and compare the numbers.

County/City	Index	Deaths
Umatilla	2.5	147
Morrow	2.6	130
Gilliam	3.4	130
Sherman	1.3	114
Wasco	1.6	138
Hood River	3.8	162
Portland	11.6	208
Columbia	6.4	178
Clatsop	8.3	210

Hebbian Learning

D.O. Hebb (1904-1985) was a physiological psychologist at McGill University. In Hebb's view, learning could be described physiologically. That is, there is a physical change in the nervous system to accommodate learning, and that change is summarized by what we now call Hebb's postulate (from his 1949 book):

> When an axon of cell A is near enough to excite a cell B and repeatedly takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased.

As with many named theorems and postulates, this was not an idea that was completely new, but he does give the postulate in a form that can be used as a basis for machine learning.

Next, we look at a mathematical model of Hebb's postulate.

Linear Neurons and Hebbian Learning

Let us first build a simple model for a neuron. A neuron has three basic parts- The dendrites, which carry information to the cell body, the cell body, and the axon, which carries information away from the cell body.

Multiple signals come in to the cell body from the dendrites. Mathematically, we will assume they all arrive at the same time, and the action of the dendrites (or the arrival site of the cell body) is that each signal is changed by the physiology of the cell. That is, if x_i is "information" along dendrite i, arrival at the cell body changes it to $w_i x_i$, where w_i is some real scalar. For example, $w_i > 1$ is an amplification of the signal, $0 < w_i < 1$ is an inhibition of the signal, and negative values mean re-polarization.

Next, the cell body collates this information by summing these signals together. This action is easily represented by the inner product of the vector of w's (the *weights*) to the signal x. For the purposes of this section, we will assume no further processing. Thus, for one neuron with n incoming signals, the input-output relationship is:

$$\mathbf{x} \mapsto \mathbf{w} \cdot \mathbf{x}$$

If the signal is passed to a *cell assembly*, or group of neurons, then each neuron has its own set of weights, and the mapping becomes:

$$\mathbf{x} \to W\mathbf{x} = \mathbf{y}$$

If we have k neurons and $\mathbf{x}$ is a vector in $\mathbb{R}^n$, then W is a $k \times n$ matrix, and each row corresponds to a signal neuron's weights (that is, W_{ij} refers to the weight taking x_j to neuron i).

Next, we model Hebb's postulate. We suppose that we present the network with a pattern, $\mathbf{x} \in \mathbb{R}^n$, and it outputs a pattern, $\mathbf{y} \in \mathbb{R}^k$. In terms of each weight, we see that

W_{ij} connects the j^{th} value of the input to the i^{th} value of the output.

Thus we might take the following as Hebb's Rule. The change in the weight connecting the j^{th} input to the i^{th} cell is given by:

$$\Delta W_{ij} = \alpha y_i x_j$$

where α is called **the learning rate**. If both x_j and y_i match in sign, then W_{ij} becomes larger. If there is a mismatch in sign, W_{ij} gets smaller. This is the *unsupervised Hebbian rule.* We now should formulate this using matrix algebra.

As before, assume we have n inputs to the network, and m outputs. Then W is $m \times n$, with $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{y} \in \mathbb{R}^m$. If we compute the outer product, $\mathbf{y}\mathbf{x}^T$, we get:

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} [x_1, x_2, \ldots, x_n] = \begin{bmatrix} y_1x_1 & y_1x_2 & \ldots & y_1x_n \\ y_2x_1 & y_2x_2 & \ldots & y_2x_n \\ \vdots & & & \vdots \\ y_mx_1 & y_mx_2 & \ldots & y_mx_n \end{bmatrix}$$

You should verify that in this case, we can compactly write Hebb's rule as:

$$W_{\text{new}} = W + \alpha \mathbf{y}\mathbf{x}^T$$

and that this change is valid for a single $\mathbf{x}, \mathbf{y}$ stimulus-response pair. If we define W_0 to be the initial weight matrix (we could initialize it randomly, for example), then the update rule becomes:

$$W_1 = W_0 + \alpha \mathbf{y}\mathbf{x}^T = W_0 + \alpha W_0 \mathbf{x}\mathbf{x}^T = W_0 \left(I_{n\times n} + \alpha \mathbf{x}\mathbf{x}^T \right)$$

EXERCISES:

1. Write W_n in terms of W_0 using the previous formula as a starting point.
2. Show that, if λ_i, $\mathbf{v}_i$ is the eigenvalue and eigenvector of a matrix A, then $(1 + \beta\lambda_i)$ and $\mathbf{v}_i$ is an eigenvalue and eigenvector of $(I + \beta A)$.
3. Let the matrix $A = \mathbf{x}\mathbf{x}^T$, where $\mathbf{x} \in \mathbb{R}^n$, and $\mathbf{x} \neq 0$. If $\mathbf{v} \in \mathbb{R}^n$, show that $A\mathbf{v}$ is a scalar multiple of x. (This shows that the dimension of the columnspace of $\mathbf{x}\mathbf{x}^T$ is 1)
4. Same matrix A as in the previous exercise. Show that one eigenvalue is $\|x\|^2$ (Hint: The eigenvector is $\mathbf{x}$).
5. We'll state the following without proof for now: If A is $n \times n$ and symmetric, and the columnspace of A has dimension 1, then there is exactly one nonzero eigenvalue. Use this, together with the previous exercises to compute the eigenvalues of $I + \alpha\mathbf{x}\mathbf{x}^T$.
6. There is a theorem that says that if the eigenvalues satisfy $|\lambda_i| \leq 1$, then the elements of A^n will converge (otherwise, the elements of A^n will diverge).

   Given our previous computation, will Hebb's rule give convergence?

Hebb's Rule with Feedback

Somehow, we want to take feedback into account so that we can use Hebb's rule in supervised learning problems.

Let $\mathbf{t}$ be the target (or desired) value for the input $\mathbf{x}$. That is, we would be given pairs of vectors,

$$(\mathbf{x}_i, \mathbf{t}_i)$$

and we want to build an affine function using matrix W and vector $\mathbf{b}$ so that

$$W\mathbf{x}_i + \mathbf{b} = \mathbf{t}_i$$

In the supervised Hebbian rule, we need to update the weights based on what we want the network to do, rather than what the network is already doing:

$$\Delta W_{ij} = \alpha t_j x_i$$

There is still something unsatisfying here- When should we stop the training? It seems like the weights could diverge as training progresses, and furthermore, we're not taking the actual outputs of the network into account. Heuristically, we would like for the update to go to zero as the target values approach the network outputs. This leads us to our final modification of our basic Hebb rule, and is called the Widrow-Hoff learning rule[2]:

$$\Delta W_{ij} = \alpha(t_j - y_j)x_i$$

If we put this in matrix form, the learning rule becomes:

$$W^{\text{new}} = W^{\text{old}} + \alpha\left(\mathbf{t} - \mathbf{y}\right)\mathbf{x}^T$$

where $(\mathbf{x}, \mathbf{t})$ is a desired input-output relation, and $\mathbf{y} = W\mathbf{x}$.

Additionally, sometimes it is appropriate to add a bias term so that the network has the output:

$$\mathbf{y} = W\mathbf{x} + \mathbf{b}$$

The bias update is similar to the previous update:

$$\mathbf{b}^{\text{new}} = \mathbf{b}^{\text{old}} + \alpha\left(\mathbf{t} - \mathbf{y}\right)$$

This modification is called the *Widrow-Hoff* update rule (See a later section for a little more on why). For now, let's get to the Matlab code so we can try some examples.

Widrow-Hoff in Matlab

To be consistent with Matlab's built in routines, we'll call $2\alpha = \text{lr}$ for *learning rate*. Then, the function to train the linear neural network will be the following:

[2]Also goes by the names Least Mean Squares rule, and the delta rule.

```
function [W,b,err]=wid_hoff1(X,Y,lr,iters)
%FUNCTION [W,b,err]=wid_hoff1(X,Y,lr,iters)
%This function trains a linear neural network
%using the Widrow-Hoff training algorithm.  This
%is a steepest descent method, and so will need
%a learning rate, lr (for example, lr=0.1)
%
%        Input:  Data sets X, Y (for input, output)
%                Dimensions:  number of points x dimension
%                lr:     Learning rate
%                iters:  Number of times to run through
%                        the data
%        Output: Weight matrix W and bias vector b so
%                that Wx+b approximates y.
%                err:  Training record for the error

%It's convenient to work with X and Y as dimension
%by number of points
X=X';
Y=Y';
[m1,m2]=size(X);
[n1,n2]=size(Y);

%Initialize W and b to zero
W=zeros(n1,m1);
b=zeros(n1,1);

for i=1:iters              %Number of times through data
  for j=1:m2               %Go through every data point
    e=(Y(:,j)-(W*X(:,j)+b));  %Target - Network Output
    dW=lr*e*X(:,j)';
    W=W+dW;
    b=b+lr*e;
    err(i,j)=norm(e);       %Store error for later
  end
end
```

Example: Associative Memory

Here we will reproduce an experiment by Widrow and Hoff[3] who built an actual machine to do this (we'll do a computer simulation).

We'll have three letters as input, T, G and F. We'll associate these letters to the numbers $-60, 0, 60$ respectively. We want our network to perform the association using the Widrow-

[3]See "Adaptive Switching Circuits" by B. Widrow and M.E. Hoff, in 1960 IRE WESCON Convention Record, New York: IRE, Part 4, p. 96-104. You might find reprints also on the internet.

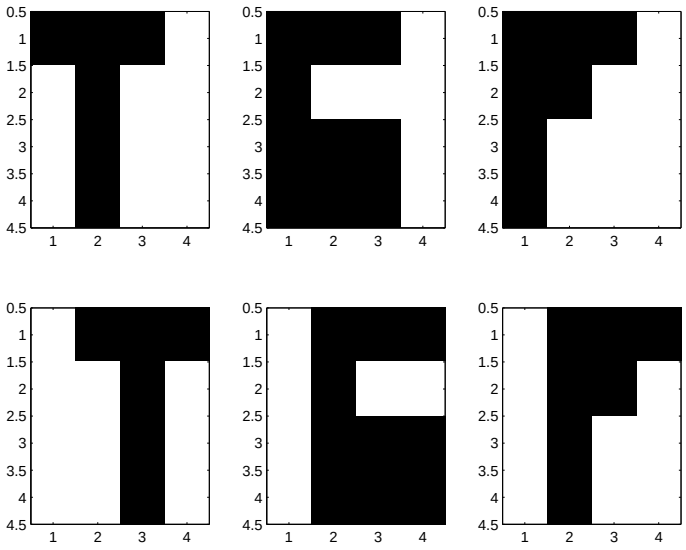

Figure 1: The inputs to our linear associative memory model: Three letters, T, G, H, where we have two samples of each letter, and each letter is defined by a 4×4 grid of numbers. We'll be associating T with -60, G with 0, and H with 60.

Hoff learning rule.

The letters will be defined by 4×4 arrays of numbers, where 1 corresponds to the color black, and -1 corresponds to the color white. In this example, we'll have two samples of each letter, as shown in Figure 1.

Implementation:

- First, we process the input data. Rather than working with 4×4 grids, we concatenate the columns to work with vectors in $\mathbb{R}^{16}$. Thus, we have 6 domain data points in $\mathbb{R}^{16}$, two samples of each letter. Construct range points so that they correspond with the letters.
- We'll use an $\alpha = 0.03$.
- We'll take several passes through all the data points.
- To measure the error, after each pass through the data, we'll put each letter through the function to get an output value. We'll take the square of the difference between that and the desired value. We'll take the sum of the errors squared for those six samples to be the measure of the error for that pass.

Here is the code we used for this example. Again, be sure to read and understand what the code is doing. A lot of the initial part of the code is just there to get the data read in and plotted.

```
%% Script file:  Associations with Widrow-Hoff
%% First, the data (and plots)

(Deleted to save space- See the actual program online)
```

```
%% Next, the actual training and results:
X=[T1(:) T2(:) G1(:) G2(:) F1(:) F2(:)];  %Each of these was a 4 x 4 matrix of numbers
X=X';  %Data should be number of pts (6) by dimension (16)
T=[60 60 0 0 -60 -60];
T=T';  %Targets should be a column.
lr=0.06;   % alpha is 0.03
iters=60;  %60 times through the data

[W,b,EpochErr]=wid_hoff1(X,T,lr,iters);

figure(2)
plot(EpochErr);

%% Example of using the network to classify the points.
Ans1=W*X(3,:)'+b  %Should be zero for G
Ans2=W*X(6,:)'+b  %Should be about -60 for F
```

Exercise: Show that the system of equations $A\mathbf{x} + \mathbf{b}$ can be written as a linear system:

$$\hat{A}\hat{\mathbf{x}}$$

for an appropriate matrix $\hat{A}$ and $\hat{\mathbf{x}}$. This means that Hebb's rule could be performed directly without training $\mathbf{b}$ separately.

Vocabulary of Learning

"To Train" a linear network is to determine weights and biases that best (in the sense of some error) match a given input-output set. There are two distinct types of training: Training when all data is available, and on-line training.

If all of the data is available, we have **batch training**, and this means that we need to solve some system of equations (least squares) for the weights and biases. Finding a line of best fit is **batch training**.

On-line training is a training algorithm that partially updates the weights and biases at each data point, and we slowly evolve the network to best match the data. It is in the latter sense that we can describe a linear network as "adaptive", and Hebb's rule was **on-line**.

Example

Find the linear neural network for the mapping from X to Y (data is ordered) given below. Use batch training instead of Hebb's rule.

$$X = \left\{ \begin{pmatrix} 2 \\ 2 \end{pmatrix}, \begin{pmatrix} 1 \\ -2 \end{pmatrix}, \begin{pmatrix} -2 \\ 2 \end{pmatrix}, \begin{pmatrix} -1 \\ 1 \end{pmatrix} \right\} \quad Y = \{-1, 1, -1, 1\}$$

SOLUTION: We'll build the system of equations as:

$$WX = Y$$

where

$$X = \begin{bmatrix} 2 & 1 & -2 & -1 \\ 2 & -2 & 2 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \qquad Y = [-1,\ 1,\ -1,\ 1]$$

In Matlab, the weight matrix W is found as: `Y/X`, as we see below. We will also illustrate the solution by plotting it. For your reference, the weight matrix was:

$$W = [-0.1523,\ -0.5076,\ 0.3807]$$

And WX gave:

```
-0.9391    1.2437   -0.3299    0.0254
```

Side remark: Did you notice that we are solving a *matrix equation* $WX = Y$ and NOT a matrix-vector equation $A\mathbf{x} = \mathbf{y}$? In this case, the matrix W was a matrix with three unknowns and we had plenty of data so that the system was overdetermined (so we could find the least squares solution). Here is the Matlab code:

```
X=[2 1 -2 -1;2 -2 2 1];
X(3,:)=ones(1,4);
Y=[-1 1 -1 1];
W=Y/X;
plot(X(1,[1,3]),X(2,[1,3]),'^',X(1,[2,4]),X(2,[2,4]),'x');
x1=-3;x2=3;y1=-3;y2=3;
t=linspace(x1,x2);
%Seperating line is ax+by+c=0
y=-(W(1)/W(2))*t-(W(3)/W(2));
hold on
plot(X(1,[1,3]),X(2,[1,3]),'o',X(1,[2,4]),X(2,[2,4]),'o');
plot(t,y);
axis([x1 x2 y1 y2]);
```

Pattern Classification

In the general pattern classification problem, we are given samples of data that represent each class. These samples are translated into vectors $\mathbf{x}_i \in \mathbb{R}^n$. Each sample is identified with a class label. Our goal is to build a function that will input a sample and output the class to which the sample belongs.

In fact, we have seen that a set of linear neurons may be able to build such a classifier. Before we continue, you may note that the class labels do not really have any intrinsic numerical value- They simply refer to which class the sample belongs, and we have indexed the classes.

For example, if we have classes $1, 2, 3, 4, 5$, does that necessarily mean that class 1 is "closer" to class 2 than class 5? Does an output of 3.5 mean that the desired target is between classes 3 and 4? You see, these numbers have no meaning.

Therefore, we should translate the classes into vectors for which we can assign some meaning. One method for labeling that may be preferable is to set the k^{th} pattern label to e_k. In

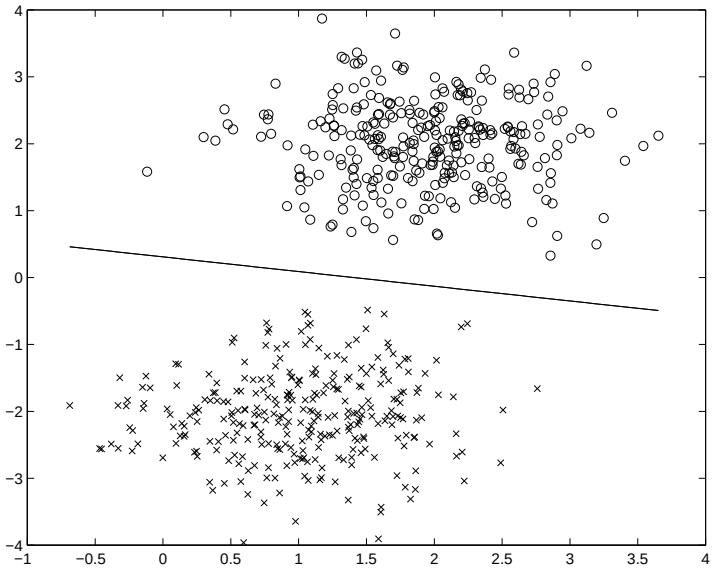

Figure 2: The Two Pattern Classification Problem. The line is the preimage of $(0.5, 0.5)^T$.

the two label problem, the output for data in pattern 1 would be set to $(1, 0)^T$ and for pattern 2 would be $(0, 1)^T$.

This has an added benefit: we can interpret $(a, b) \to (\frac{a}{a+b}, \frac{b}{a+b})^T$ as a probability. That is, $\boldsymbol{x}$ has probability $\frac{a}{a+b}$ of being in pattern 1, and probability $\frac{b}{a+b}$ of being in pattern 2.

Example:

The first set of commands creates the data that we will classify. This script file will reproduce (some of the data is random) the image in Figure 2.

```
1  X1=0.6*randn(2,300)+repmat([2;2],1,300);
2  X2=0.6*randn(2,300)+repmat([1;-2],1,300);
3  X=[X1 X2];
4  X(3,:)=ones(1,600);
5
6  Y=[repmat([1;0],1,300) repmat([0;1],1,300)];
7  C=Y/X;
8
9  %Plotting routines:
10 plot(X1(1,:),X1(2,:),'o',X2(1,:),X2(2,:),'x');
11 hold on
12 n1=min(X(1,:));n2=max(X(1,:));
14 t=linspace(n1,n2);
15 L1=(-C(1,1)*t+(-C(1,3)+0.5))./C(1,2);
16 L2=(-C(2,1)*t+(-C(2,3)+0.5))./C(2,2);
17 plot(t,L1,t,L2);
```

- Lines 1-3 set up the data set X. We will take the first 300 points ($X1$) as pattern 1, and the last 300 points as pattern 2.
- Line 4 sets up the augmented matrix for the bias.
- Line 6 sets up the targets.
- Line 7 is the training. The weights and biases are in the 2×3 matrix C.
- Line 10: Plot the patterns
- Line 12-17: Compute the seperating lines.
- Line 18: Plot the seperating lines. (They are identical in theory, in practice they are very, very close).

Exercise

Let's put all of this together to solve another pattern classification problem using Hebb's rule. Suppose we are given the following associations:

Point	Class
$(1, 1)$	1
$(1, 2)$	1
$(2, -1)$	2
$(2, 0)$	2
$(-1, 2)$	3
$(-2, 1)$	3
$(-1, -1)$	4
$(-2, -2)$	4

Graphically, we can see the classes in the plane in Figure 3. In this example, take Class 1 to be the vector $[-1, -1]^T$, Class 2 as vector $[-1, 1]^T$, Class 3 as $[1, -1]^T$, and Class 4 as $[1, 1]^T$- this puts the 4 classes are on the vertices of a square.

Now for the details of the program. First write the inputs as an 2×8 matrix, with a corresponding output matrix that is also 2×8. Parameters that can be placed first will be the maximum number of times through the data N and the learning rate, a, which we will set to 0.04. We can also set an error bound so that we might stop early. Set the initial weights to the 2×2 identity, and the bias vector b to $[1, 1]^T$.

Be sure you have trained long enough to get a good error, and plot the decision boundaries as well.

Derivation of the Widrow Hoff Algorithm

The purpose of this section is to show that the Widrow-Hoff change to the standard Hebbian unsupervised learning algorithm actually converges to the least squares solution given by a batch algorithm. To proceed, we have to assume that all data is known in advance (so the model does not change over time).

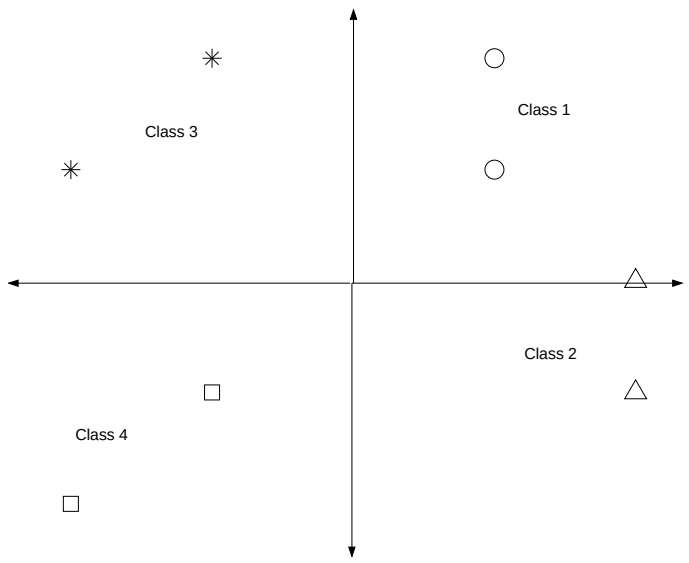

Figure 3: Pattern Classification Problem. Each point is a sample of one of the four classes.

In 1960 Bernard Widrow and his graduate student Marcian Hoff developed a new neural network and a new learning rule which they called the LMS (Least Mean Square) Algorithm- which is an online algorithm designed to minimize the mean square error (MSE). That is, given n stimulus-response pairs, $(\mathbf{x}_i, \mathbf{t}_i)$, and given a matrix W and vector $\mathbf{b}$ so that

$$\mathbf{y}_i \doteq W\mathbf{x}_i + \mathbf{b}$$

the MSE is given by

$$E = \frac{1}{n}\sum_{i=1}^{n} \|\mathbf{y}_i - \mathbf{t}_i\|^2$$

The (online) training rule proposed by Widrow and Hoff is the same as our modification to Hebb's rule:

$$W^{(new)} = W^{(old)} + \alpha\,(\mathbf{t}_i - \mathbf{y}_i)\,\mathbf{x}_i^T$$

Now, we know from calculus that the direction of largest decrease of a function $f : \mathbb{R}^n \to \mathbb{R}$ is in the direction of the negative of the gradient of f. If we view f as our error function E, performing gradient descent will ideally result in minimizing the error.

To simplify the derivation, we'll assume that the network has a one-dimensional output so that

$$y_i = \mathbf{w}^T\mathbf{x}_i + b$$

Then we wish to minimize the mean square error with respect to the weights and biases:

$$E_{\text{mse}} = \frac{1}{p}\sum_{k=1}^{p}(e(k))^2 \doteq \frac{1}{p}\sum_{k=1}^{p}(t_k - y_k)^2$$

NOTE: In full generality, the error function is the sum of the squares of the *norm* of the error vector:

$$E_{\text{mse}} = \frac{1}{p}\sum_{k=1}^{p}\|\mathbf{e}(k)\|^2 \doteq \frac{1}{p}\sum_{k=1}^{p}\|\mathbf{t}_k - \mathbf{y}_k\|^2$$

1. **Exercise:** Verify that:
$$\frac{\partial e^2(k)}{\partial w_j} = 2e(k)\frac{\partial e(k)}{\partial w_j}$$
and
$$\frac{\partial e^2(k)}{\partial b} = 2e(k)\frac{\partial e(k)}{\partial b}$$

2. **Exercise:** Show that:
$$\frac{\partial e(k)}{\partial w_j} = -x_j(k) \text{ and } \frac{\partial e(k)}{\partial b} = -1$$
where $x_j(k)$ refers to the j^{th} coordinate of the k^{th} data point.

   The standard way of performing gradient descent would mean that we adjust the vector $\boldsymbol{w}$ and scalar b by:
$$\text{New } w_j = \text{Old } w_j + \alpha\frac{\partial E_{\text{mse}}}{\partial w_j} \text{ and New } b = \text{Old } b + \alpha\frac{\partial E_{\text{mse}}}{\partial b}$$
where α is our learning rate. We will *estimate* E_{mse} at data point k:
$$w_j(k+1) = w_j(k) + \alpha\frac{\partial e^2(k)}{\partial w_j} \text{ and } b(k+1) = b(k) + \alpha\frac{\partial e^2(k)}{\partial b}$$

3. **Exercise:** Show that the Widrow-Hoff rule is given by:
$$\boldsymbol{w}(k+1) = \boldsymbol{w} + 2\alpha(t_k - y_k)\boldsymbol{x}^{(k)}$$
$$b(k+1) = b(k) + 2\alpha(t_k - y_k)$$

4. **Extensions:** For multidimensional output, this update extends to:
$$W(k+1) = W(k) + 2\alpha\boldsymbol{e}(k)\left(\boldsymbol{x}^{(k)}\right)^T$$
$$\boldsymbol{b}(k+1) = \boldsymbol{b}(k) + 2\alpha\boldsymbol{e}(k)$$

5. We could also add a "momentum" term to try to speed up the gradient descent. A useful example of how the learning rate and momentum effect the convergence can be found in Matlab: `nnd10nc`, which we will also look at in the next section. The general form of gradient descent with a momentum term μ and learning rate α is given by:
$$\Delta\boldsymbol{x} = \mu\Delta\boldsymbol{x} + (1-\mu)\alpha\nabla f(\boldsymbol{x}) \quad (1)$$
$$\boldsymbol{x} = \boldsymbol{x} + \Delta\boldsymbol{x} \quad (2)$$
   From these equations, we see that if the momentum term is set so that $\mu = 0$, we have standard gradient descent (which may be too fast), and if we set $\mu = 1$, then since $\Delta\boldsymbol{x}$ is usually set to 0 to start, then $\Delta\boldsymbol{x}$ will be zero for all iterations.

Time Series and Linear Networks

We've already seen one application of linear networks: If data is linearly seperable, then a linear network can do pattern classification. Furthermore, if the input-output relationship is linear, then a linear net can approximate that relationship. Here, we will see that a linear neural network can be used in signal processing.

1. **Definition:** A time series is a sequence of real (or complex) numbers. We denote a time series in the usual way:
$$X = \{x(1), x(2), x(3), \ldots, x(t), \ldots\}$$

2. **Definition:** A tapped delay line with k taps is constructed from a time series:
$$\hat{x}_1 = \begin{pmatrix} x(k) \\ x(k-1) \\ \vdots \\ x(1) \end{pmatrix}, \quad \hat{x}_2 = \begin{pmatrix} x(k+1) \\ x(k) \\ \vdots \\ x(2) \end{pmatrix}, \ \ldots$$

3. **Remark:** This is also called a time series with *lag* k.

4. **Remark:** This is also called an *embedding* of the time series to $\mathbb{R}^k$.

5. **Remark:** In Matlab, we can do the embedding in the following way. Here, we embed to $\mathbb{R}^5$, columnwise:

```
Q=length(X);
P=zeros(5,Q);
P(1,2:Q)=X(1:(Q-1));
P(2,3:Q)=X(1:(Q-2));
P(3,4:Q)=X(1:(Q-3));
P(4,5:Q)=X(1:(Q-4));
P(5,6:Q)=X(1:(Q-5));
```

   Look these lines over carefully so that you see what is being done- we're doing some padding with zeros so that we don't decrease the number of data points being used.

6. **Exercise:** Use the last remark as the basis for a function you write, call `lag`, whose input is a time series (length n), and input the number of taps, k. Output the $k \times n$ matrix of embedded points.

7. **Definition:** A *filter* with $k-$taps is a function on the time series so that:
$$x_i = f(x_{i-1}, x_{i-2}, \ldots, x_{i-k})$$

8. **Remark:** We can think of a filter as performing a prediction on x_i using the past k time series values.

9. **Definition:** A *linear filter* will have the form:
$$\boldsymbol{w}^T \boldsymbol{x} + b = x_i$$
   where $\boldsymbol{w}$ are the weights, b is the bias, and $\boldsymbol{x} = (x_{i-1}, x_{i-2}, \ldots, x_{i-k})^T$.

Application: Novelty Detection

Given a time series, $\{x_i\}$, one critical application is to determine if a signal is operating "normally", or if some property of the data is changing over time. The linear network can be used in some instances to perform novelty detection.

The main idea is that we will assume that the current real value, x_i is a function of some fixed number of past values, $x_{i-1}, \ldots, x_{i-k}$. Another way to say this is that we assume that the following model holds:

$$x_i = w_1 x_{i-1} + w_2 x_{i-2} + \cdots + w_k x_{i-k} + b$$

for each i. If the model is accurate, then we should be able to find values of the weights and bias.

Once trained, *if the underlying process ever changes*, then the model will change to give us a large error (which would provide a flag for novelty). Here's an example using 5 past values to predict the 6th. Notice that at "4 seconds" the underlying model changes. Try running this code and report on the results.

```
%  First, build the time vectors:

time1 = 0:0.05:4; % from 0 to 4 seconds
time2 = 4.05:0.024:6; % from 4 to 6 seconds
time = [time1 time2]; % from 0 to 6 seconds

%%  The data
T = [sin(time1*4*pi) sin(time2*8*pi)];
lr = 0.2;

% The domain will be vectors in R^5 found by lagging the vector T:
X=lag(T,4,1);  %X is 5 x 159.  That is, T(1:5) is the first column of X,
               % and is used to predict T(6).
X(:,159)=[];   %Last column not used.

% Construct the desired targets:
Targets=T(6:158+5);

%Online training.  In this case, track the error point-by-point:
[W,B,err]=wid_hoff1(X',Targets',0.01,120);

%The error is an array, 80 x 158 (80=number of epochs, 158=number of
%  points)
plot(mean(err))

%Plot the predicted values and the actual values:
Yout=W*X+B;
tt=time(6:158+5);
figure
plot(tt,Targets-Yout)
```

Summary

A linear neural network is an affine mapping. The network can be trained two ways: Least squares (using the pseudoinverse) if we have all data available, or adaptively, using the Widrow-Hoff algorithm, which is an approximate gradient descent on the least squares error.

Applications to time series include novelty detection and prediction, where we first embed the time series to $\mathbb{R}^k$. These applications are widely used in the signals processing community.

Bibliography

References

[1] Fadeley, R. Oregon malignancy pattern physiographically related to Hanford, Washington, Radioisotope Storage. Journal of Environmental Health, 27(6), p. 883–897, June 1965.