Linear Models

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

Generally speaking, when we have data, we will have a set for the input (or domain), which will typically be an array X or individually marked $\mathbf{x}^{(i)}$ as vectors. The desired output set, or *target* set, we will put in an array T or $\mathbf{t}^{(i)}$. We typically also have some function we're building. The output of the function we will call the array Y. There is no widespread standard for this- Statistics will often use y and $\hat{y}$ for the target set and model set, for example.

Now, we take a look at constructing linear models.

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:

$$(x^1, t^1), (x^2, t^2), \cdots, (x^n, t^n)$$

which we can concatenate in the arrays X and T:

$$X = \left[x^1, x^2, \ldots, x^n\right] \quad T = \left[t^1, t^2, \ldots, t^n\right]$$

These are indexed with a superscript because they may be vectors.

We wish to find the best linear relationship between X and T. 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, and let x, y be real numbers. Then we are trying to find m and b so that

$$y = mx + b \quad \text{and} \quad y \approx t$$

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

$$\begin{array}{rcl} t_1 &=& mx_1 + b \\ t_2 &=& mx_2 + b \\ &\vdots& \\ t_n &=& mx_n + b \end{array} \quad \Rightarrow \quad \mathbf{t} = \begin{bmatrix} & 1 \\ \mathbf{x} & 1 \\ & \vdots \\ & 1 \end{bmatrix} \begin{bmatrix} m \\ b \end{bmatrix} \quad \Rightarrow \quad A\mathbf{c} = \mathbf{t}$$

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

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

$$E_1 = \sum_{k=1}^{n} |t_k - y_k| = \sum_{k=1}^{n} |t_k - (mx_k + b)|$$

This is the 1-norm error, and we saw earlier that the error can be minimized using Linear Programming.

The Usual Method: Least Squares

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

$$E_{\text{se}} = \sum_{k=1}^{n} (t_k - y_k)^2 = \sum_{k=1}^{n} (t_k - (mx_k + b))^2$$

This is a nice error, since the partial derivatives of E are continuous.

Exercises

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

$$\frac{\partial E_{\text{se}}}{\partial m} = 0 \quad \frac{\partial E_{\text{se}}}{\partial b} = 0$$

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

$$\begin{aligned} m \sum x_k^2 + b \sum x_k &= \sum x_k t_k \\ m \sum x_k + bn &= \sum t_k \end{aligned}$$

2. Show that this is the same set of equations you get by solving the normal equations, $A^T A\mathbf{c} = A^T \mathbf{t}$,

Background to Linear Nets

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:

$$\mathbf{x} \in \mathbb{R}^n \to W\mathbf{x} = \mathbf{y} \in \mathbb{R}^k$$

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} connects the i^{th} dimension of the output to the j^{th} dimension of the input.

Thus we might take the following as Hebb's Rule. Given a vector $\mathbf{x}$ and output vector $W\mathbf{x} = \mathbf{y}$, 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*.

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 $W\mathbf{x} = \mathbf{y} \in \mathbb{R}^m$. 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 $\mathbf{x}$. 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

$$\mathbf{y}^i = W\mathbf{x}^i + \mathbf{b} \approx \mathbf{t}^i$$

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. For output dimension j, input dimension k (and for the i^{th} data point),

$$\Delta W_{jk} = \alpha t_j^i x_k^i$$

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[1]:

$$\Delta W_{jk}^{(i+1)} = \alpha(t_j^i - y_j^i)x_k^i$$

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:

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

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

The bias update is similar to the previous update:

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

This modification is called the *Widrow-Hoff* update rule, which we show is related to gradient descent using a certain error.

Derivation of the Widrow Hoff Algorithm

The purpose of this section is to show that the Widrow-Hoff algorithm 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).

As a little background, 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

$$E(W, b) = \frac{1}{n}\sum_{i=1}^{n} \|\mathbf{t}^i - \mathbf{y}^i\|^2$$

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

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

We'll show the connection to minimizing the error below.

One Dimensional Output

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

$$y^i = \mathbf{w}^T\mathbf{x}^i + b = w_1x_1^i + w_2x_2^i + \cdots + w_nx_n^i + b$$

for $i = 1, 2, \cdots, p$ (we've got p data pairs). Further, suppose we look at the error for a single data pair (so below, i is a fixed integer representing which data point we're looking at):

$$E^i(\mathbf{w}, b) = (t^i - y^i)^2 \quad \Rightarrow \quad \frac{\partial E^i}{\partial w_j} = 2(t^i - y^i)\left(-x_j^i\right), \quad \frac{\partial E^i}{\partial b} = 2(t^i - y^i)(-1)$$

or, the gradient of E^i would be:

$$-\nabla E^i = 2(t^i - y^i)(\mathbf{x}^i)^T \qquad -\frac{\partial E^i}{\partial b} = 2(t^i - y^i)$$

Which gives us a scalar multiple of our update rule (using a single point i).

Multidimensional Output

If we have multidimensional output, then $\mathbf{y}^i = W\mathbf{x}^i + \mathbf{b}$ and the j^{th} coordinate of $\mathbf{y}^i$ can be written out using the j^{th} row of W, $W(j,:)$, which is:

$$y_j^i = W(j,:)\mathbf{x}^i + b_j$$

and the error using the i^{th} data point is:

$$E^i(W,\mathbf{b}) = \|\mathbf{t}^i - \mathbf{y}^i\|^2 = (t_1^i - y_1^i)^2 + \cdots + (t_m^i - y_m^i)^2$$

$$= (t_1^i - (W(1,:)\mathbf{x}^i + b_1))^2 + \cdots + (t_m^i - (W(m,:)\mathbf{x}^i + b_m))^2$$

Therefore,

$$\frac{\partial E^i}{\partial W_{jk}} = 2(t_j^i - y_j^i)\left(-\frac{\partial y_j^i}{\partial W_{jk}}\right) = 2(t_j^i - y_j^i)(-x_k^i)$$

The main idea now is that we *estimate* the overall average error E by using the error at a single data point, E^i. Then, the update rule for gradient descent gives:

$$W^{(i+1)} = W^i + 2\alpha(\mathbf{t}^i - \mathbf{y}^i)\left(\boldsymbol{x}^i\right)^T$$

$$\boldsymbol{b}^{(i+1)} = \boldsymbol{b}^i + 2\alpha(\mathbf{t}^i - \mathbf{y}^i)$$

Vocabulary

"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.

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**.

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. We describe this process below.

An Alternative to Online Learning: Batch Training

If we know all of the data, we do not need to use the on-line learning algorithm. Rather, we can construct and solve a linear algebra problem.

Converting affine to linear

Before going further, we note that it is somewhat simpler theory to deal with solving the linear mapping $\mathbf{x} \to A\mathbf{x}$ than it is the affine mapping where we add $\mathbf{b}$. However, using a change of coordinates, we can convert the affine map to a linear map.

Given the $m \times n$ matrix A and the input data $\mathbf{x}^i$, we define the new $m \times (n+1)$ matrix $\hat{A}$ and new $(n+1)-$dimensional vector $\hat{\mathbf{x}}$:

$$\hat{A} = [A|\mathbf{b}] \qquad \hat{x} = \begin{bmatrix} \mathbf{x} \\ \hline 1 \end{bmatrix}$$

You should verify that $\hat{A}\hat{\mathbf{x}}$ is the same as $A\mathbf{x} + \mathbf{b}$.

If we have p data points, we can line them up column-wise in an $(n+1) \times p$ matrix X:

$$X = \begin{bmatrix} \hat{\mathbf{x}}^1 & \hat{\mathbf{x}}^2 & \cdots \hat{\mathbf{x}}^p \end{bmatrix}$$

Similarly, the desired targets can be lined up column-wise in an $m \times p$ matrix T, so that

$$\hat{A}X = Y \quad \text{ so that } \quad Y \approx T$$

While $\hat{A}$ is not invertible (it is not even square), there is a matrix called a *pseudo-inverse* of $\hat{A}$, that will solve our problem. To understand the computation of the pseudoinverse, we need the Singular Value Decomposition of the matrix A, which will get us too far afield for now. For us, we will use Matlab's `pinv` command to compute the pseudoinverse.

Pattern Classification

A common problem we are asked to solve is the pattern classification problem. 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.

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 $1, 2, 3, 4, 5$, that does not necessarily mean that class 1 is "closer" to class 2 than class 5.

Therefore, we should translate the classes into vectors for which we can assign some meaning. In the first example below, one class is assigned the value -1 and the other is assigned the class 1.

A second method for labeling is used in the second example below. In this case, we assign the k^{th} pattern label to e_k. In 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 1:

Problem: Find the linear neural pattern classifier for the mapping from X to Y (data is ordered) given below. Use batch training.

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

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

$$\hat{A}X = Y$$

where X has the last row of 1's, as we discussed earlier:

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

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

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

which translates to

$$\mathbf{y} = A\mathbf{x} + \mathbf{b} = [-0.1523,\ -0.5076]\mathbf{x} + 0.3807$$

And WX gave:

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

which we compare to $[-1, 1, -1, 1]$. In practice, we might go ahead and define any output that is negative as class 1, any output that is positive is class 2. Any output that is zero would not be classified- This is the boundary that separates class 1 and 2. This line is given by:

$$[-0.1523,\ -0.5076]\mathbf{x} + 0.3807 = 0$$

which is plotted in the Matlab code as well. Here is the Matlab code and plot:

```
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');
%Window Size for graphics
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]);
```

Example 2: A Similar Problem

The first set of commands creates the data that we will classify. The main idea here is how we will set up the targets- In this case, we use $[1, 0]^T$ and $[0, 1]^T$ as the desired targets in place of using ± 1. In this case, the line that separates the classes

This script file will reproduce (some of the data is random) the image in Figure 1.

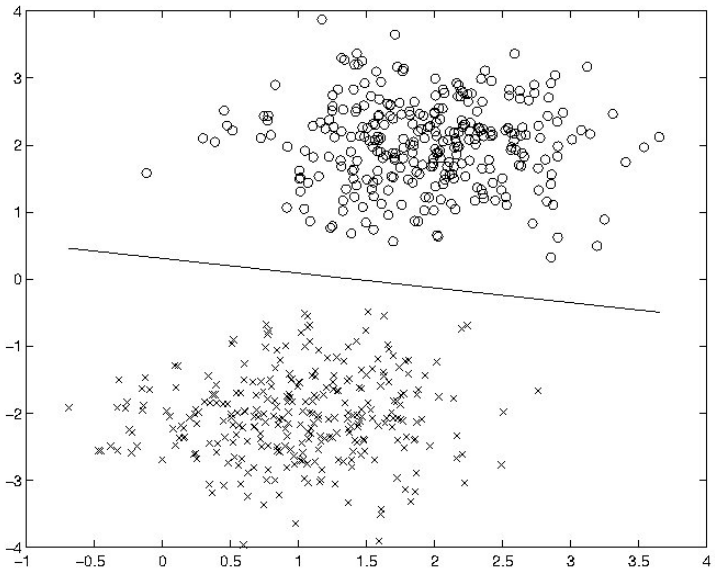

Figure 1: The Two Pattern Classification Problem. The line is the preimage of $(0.5, 0.5)^T$.

```
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).

In either of our two examples, once the matrix W (or A) and vector $\mathbf{b}$ is set, they are not changed. To classify new points, we would compute $A\mathbf{x} + \mathbf{b}$ to determine the class of the unknown points.

Alternatively, if we do not know all of the data at once, or if we suspect that the data may change over time, we can use online learning using the Widrow-Hoff update. Below, we write up the Matlab code for it and look at some examples.

Pattern Classification: Online Learning

In this section, we look at pattern classification using online learning and the Widrow-Hoff update.

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:

```
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
```

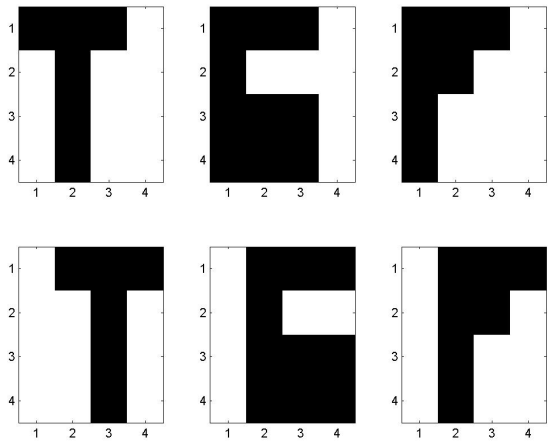

Figure 2: The inputs to our linear associative memory model: Three letters, T, G, H, where we have two samples of each letter, and each letter is defined by a 4×4 grid of numbers. We'll be associating T with -60, G with 0, and H with 60.

```
  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
```

You should copy this file into Matlab, and save it as `wid_hoff1.m` We'll need this function when running the scripts below (these files should also be available on our class website).

Example: Associative Memory

Here we will reproduce an experiment by Widrow and Hoff[2] 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-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 2.

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 $\mathbb{R}^{16}$,

[2]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.

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
```

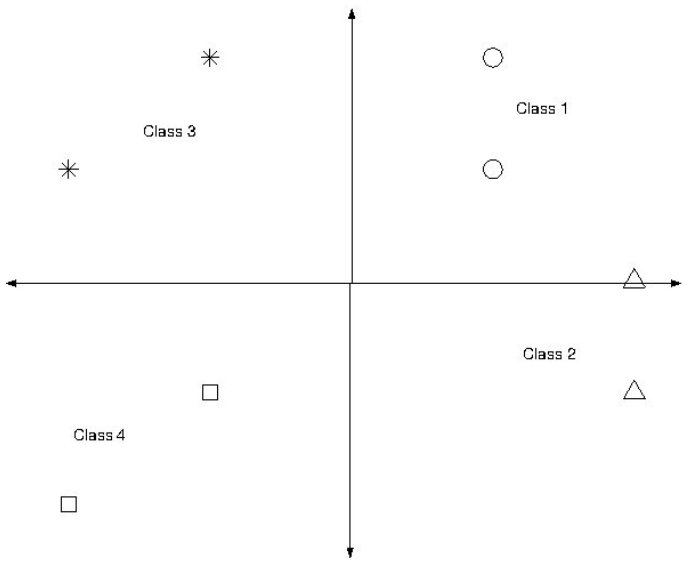

Figure 3: Pattern Classification Problem. Each point is a sample of one of the four classes.

Exercises

1. 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.

2. **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.

```
%% Construct the time
k=5;  %Use this many points to predict the next point.

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;
N=length(T);

%% Construct the domain:

% First we'll compute how many vectors we can form, then we'll form them.
m=N-k-1;
for j=1:k
    temp=T(j:m+j);
    X(j,:)=temp';
end
Targets=T(k+1:k+m+1);

%Online training.  In this case, track the error point-by-point:
[W,B,err]=wid_hoff1(X',Targets',0.01,240);

plot(mean(err))
title('Mean of the error, 240 repetitions')
%Plot the predicted values and the actual values:
Yout=W*X+B;
tt=time(6:158+5);
figure
```

```
plot(tt,Targets,'ko',tt,Yout,'bx')
title('Targets and outputs');
```