

Chapter 10

Linear Neural Networks

In this chapter, we introduce the concept of the linear neural network. As we will see, a neural network is a biologically inspired algorithm that allows us to build a functional representation from data. In statistical terms, a neural network generally represents nonlinear regression. In this chapter, however, we start with a linear network and examine its properties and shortcomings.

10.1 A Model of Learning

D.O. Hebb (1904-1985) was a physiological psychologist at McGill University. In Hebb's view, learning could be described physiologically: There is some 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. Here are some questions you might think about:

1. Hebb's postulate describes a strengthening or weakening of connections. How is this biologically or chemically done? What exactly is that "physical change"?
2. If this is the basis for learning, is it also the basis of addiction? Friendship? We can't answer these questions, but they are interesting to consider philosophically.
3. The postulate does not give any consideration to *feedback*. If the action causes pain, will the neuron connections still be strengthened? How does emotion in general mitigate or assist in these constructions? Again, we can't give authoritative answers to these questions.

We'll leave these questions for you to think about, and move into something we can answer. How do you mathematically model Hebb's postulate?

10.2 Linear Neural Nets

We will go into the formal details later for defining *neural nets*, but this is a good place to get a feel for what they're all about.

Let us first build a simple model for a neuron. A neuron has three body 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 value. Next, the cell body collates this information by summing these signals together. So far, then, this action is simply a dot product of the vector \mathbf{w} (the *weights*) to the signal \mathbf{x} . An additional value is added to the result, which we can think of as the resting state of the cell (or in statistical terms, the bias). In Figure 10.1, we graphically depict the flow of information from the input layer to the output layer.

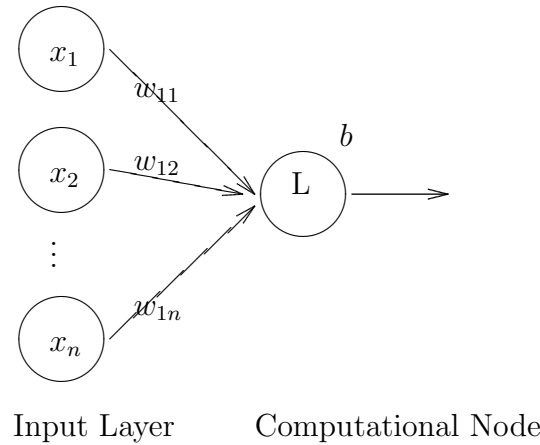


Figure 10.1: The Linear Node. Information travels from left to right.

Some vocabulary that we'll use:

- x_1, \dots, x_n are presented to the “input layer”. Some researchers call this an actual layer, some do not (which makes some counts of the network different).
- The w 's are called the “weights”, and we will also denote the edge by the same notation. When a signal passes through an edge, the result is that the signal is multiplied by the weight.
- At node L, the sum of the incoming signals is taken, and added to a value, b . We think of b as the “resting state” of the cell, which is also called the bias term.

We see that mathematically, this single node of a linear network is an affine function from \mathbb{R}^n to \mathbb{R} :

$$\mathbf{x} \mapsto (w_{11}, w_{12}, \dots, w_{1n}) \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} + b = w_{11}x_1 + w_{12}x_2 + \dots + w_{1n}x_n + b = \mathbf{w} \cdot \mathbf{x} + b$$

If we have m neurons and \mathbf{x} is a vector in \mathbb{R}^n , then W is a $m \times n$ 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). Graphically, we see this in Figure 10.2. The full mapping is now formally an affine from \mathbb{R}^n to \mathbb{R}^m :

$$\mathbf{x} \rightarrow W\mathbf{x} + \mathbf{b}$$

As we know, problems that are *linear* are usually easier to work with, so we can use a “trick” from computer graphics to convert our affine map to a linear map by going up one dimension. First an example of how this will be done:

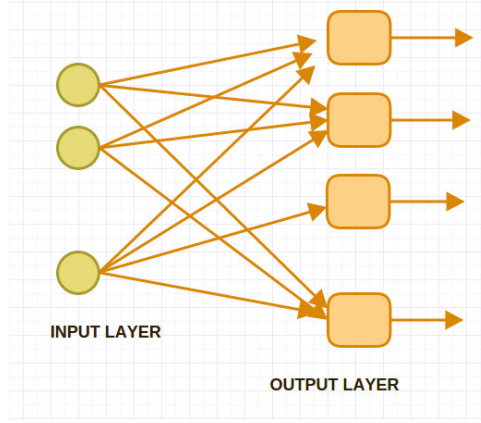


Figure 10.2: The Linear Neural Network is an affine mapping from \mathbb{R}^n to \mathbb{R}^m

Example (Convert Affine to Linear)

Suppose that we have the 2×2 affine problem:

$$\begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$$

If we put the vector \mathbf{b} as the last column of the matrix, we just need to add a dimension to the vector \mathbf{x} by putting a 1 in that position. That is, you should verify that our affine map is equivalent to the following linear map:

$$\begin{pmatrix} a_1 & a_2 & b_1 \\ a_3 & a_4 & b_2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$$

As a summary, we have the following conversion: Given the affine problem

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

define $\hat{A} = [A \quad \mathbf{b}]$ and $\hat{\mathbf{x}} = [\mathbf{x}, 1]^T$. Then the affine map is equivalent to the linear map

$$\hat{A}\hat{\mathbf{x}} = \mathbf{y}.$$

10.3 Training a Network

The output of a linear neural net can be modeled by an affine map. Typically, the data pairs, or desired associations, are given to you and interpreted as ordered pairs:

$$(\mathbf{x}^{(i)}, t^{(i)})$$

and the problem is to determine the weights W and biases \mathbf{b} so that

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

The process by which we determine W, \mathbf{b} is called **training** the network- sometimes the process is also called **learning**, as in “the network is learning the association”. There are many ways one may perform the training, but there is one way of classification that is helpful: On-line or Batch.

- On-line training (or on-line learning) is an adaptive approach where the weights and biases are modified after looking at a single data pair $(\mathbf{x}^{(i)}, \mathbf{t}^{(i)})$.
- Batch training is a one-step training method where the weights and biases are determined after the entire data set has been analyzed.

As it turns out, both training processes are designed to minimize an error function, and the most common way to construct an error function is to take the “sum of squares error”. If we assume that we have p data pair, then the sum of squares error is defined as the following. Note that some practitioners will multiply by $1/2$ (so the derivative doesn’t have a “2” in it), or will take the average error (so multiply by $1/p$).

$$E(W, \mathbf{b}) = \sum_{j=1}^p \|\mathbf{t}^{(j)} - \mathbf{y}\|^2 = \sum_{j=1}^p \|\mathbf{t}^{(j)} - (W\mathbf{x} - \mathbf{b})\|^2$$

If we multiply E by a constant, the values of W and \mathbf{b} that give us the optimal value will be the same, so we may do so if it makes our computations easier.

In the next section, we’ll talk about adapting our linear net one data point at a time- This is the on-line training. Later, we’ll discuss batch training.

10.4 Hebbian Learning (On-line training)

We’ll recall that the linear network inputs a pattern, $\mathbf{x} \in \mathbb{R}^n$, and it outputs a pattern, $\mathbf{y} \in \mathbb{R}^m$. In terms of the individual weights,

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, and if there is a mismatch in sign, W_{ij} gets smaller. This is the *unsupervised Hebbian rule*. There are some difficulties with this- in particular, if α stays fixed, then the update rule will “blow up” on us- We need to incorporate the targets.

Now, assume we have known input-output pairs, $(\mathbf{x}^{(i)}, \mathbf{t}^{(i)})$. Keeping our definition of \mathbf{y} as the output of the network (or the predicted value of \mathbf{t}):

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

We want to define the update rule in such a way as to go to zero as the output \mathbf{y} gets closer to the target \mathbf{t} . Here is one way to accomplish this, and it is called the Widrow-Hoff learning rule¹:

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

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

$$W_{\text{new}} = W_{\text{old}} + \alpha(\mathbf{t} - \mathbf{y})\mathbf{x}^T \quad (10.1)$$

where (\mathbf{x}, \mathbf{t}) is a desired input-output relation, and $\mathbf{y} = W\mathbf{x} + \mathbf{b}$, and the update rule for the bias vector is similar:

$$\mathbf{b}_{\text{new}} = \mathbf{b}_{\text{old}} + \alpha(\mathbf{t} - \mathbf{y}) \quad (10.2)$$

A couple of notes about these formulas:

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

- The notation $(\mathbf{t} - \mathbf{y})\mathbf{x}^T$ is called an **outer product** of vectors. Think about the dimensions of the vectors involved: \mathbf{t}, \mathbf{y} are both $m \times 1$, and \mathbf{x} is $n \times 1$. Therefore, the multiplication will yield a matrix with the following dimensions:

$$(m \times 1) \times (1 \times n) \Rightarrow (m \times n)$$

We recall the inner product will output a scalar, and now we see that the outer product will produce a matrix.

- We could have converted the affine map into a linear map and use one update rule as well (in that case, only the first rule).

10.4.1 Derivation of Widrow-Hoff (Exercises)

In this series of exercises, you'll see that the Widrow-Hoff update is really an approximation to gradient descent on our error function. First, we'll look at one-dimensional output, and then extend that to multidimensional output.

For notation, let $k = 1, 2, \dots, p$ index the data. Let $(\mathbf{x}^{(k)}, t^k)$ denote the input, target pairs for the linear network, and

$$\mathbf{w}^T \mathbf{x}^{(k)} + b = y^{(k)} \approx t^{(k)}$$

Exercises:

1. Find an expression for $\partial E / \partial w_j$ (be sure to substitute the function in for y), where E is our sum of squares error.
2. Find an expression for $\partial E / \partial b$.
3. Using the previous two answers, what would our update rule look like if we performed gradient descent on the error function?
4. Instead of using the full error function, we will estimate the full error by using only one data point. That is, for the k^{th} data point,

$$E(\mathbf{w}, b) \approx (t^{(k)} - y^{(k)})^2$$

Show that using the approximation gives us the Widrow-Hoff rule:

$$\mathbf{w}_{\text{new}} = \mathbf{w}_{\text{old}} + \beta(t_k - y_k)\mathbf{x}^{(k)}$$

and

$$b_{\text{new}} = b_{\text{old}} + \beta(t_k - y_k)$$

for some constant β .

5. Show that the multidimensional extension leads us to Equation 10.1 and 10.2.

10.4.2 Matlab Function: WidHoff.m

Hebbian learning using the Widrow-Hoff update can now be summarized by the following Matlab function. We input the input-output pairs in the arrays X and T , give the value of α and the number of times through the data, `NumEpochs`, and what comes out is the weight matrix and bias vector, with a measure of error.

```
function [W,b,EpochErr]=WidHoff(X,T,alpha,NumEpochs)
% function [W,b,EpochErr]=WidHoff(X,T,alpha,NumEpochs)
% Data in X should be dimension x num of points
% Target data should be dimension x num of points
% OUTPUT: Weight matrix W is dim(T) x dim(X), and b is dim(T)
```

```

% Some error checks to be sure data is input correctly and initializations:
[rX,cX]=size(X);
[rT,cT]=size(T);
if cX~=cT
    error('Error in inputs: Number of points do not match.\n');
end

NumPoints=cX;
W=randn(rT,rX);
b=randn(rT,1);
EpochErr=zeros(NumEpochs,1);

% Main Code:

for k=1:NumEpochs
    idx=randperm(NumPoints);

    for j=1:NumPoints
        ThisOut=W*X(:,idx(j))+b;
        ThisErr=T(:,idx(j))-ThisOut;
        %Update the weights and biases using Widrow-Hoff:
        W=W+alpha*ThisErr*X(:,idx(j))';
        b=b+alpha*ThisErr;
    end
    EpochErr(k)=norm((W*X+b*ones(1,NumPoints))-T);
end

```

10.4.3 Example: Associative Memory

Here we will reproduce an experiment by Widrow and Hoff² 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 10.3.

Implementation and problem specification:

- 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} . Therefore, we have 6 data points (or vectors) in \mathbb{R}^{16} . We'll assume they are in order: T's, then G's, then F's. The target vector T will be 6 dimensional.
- We'll use an $\alpha = 0.03$ (rather arbitrary at this point, we find good values of α using some experimentation).
- We'll take 60 passes through the data (60 training epochs).

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. To save space, I have grouped up commands where you don't need to change them.

²See "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'll find reprints on the internet.

```

%% Script file: Online Training (Hebbian Learning)
% Example 1: T, G and F.

%% Load the Data and Graph the Results.
T1=[1 1 1 -1;-1 1 -1 -1;-1 1 -1 -1;-1 1 -1 -1];
T2=[-1 1 1 1;-1 -1 1 -1;-1 -1 1 -1;-1 -1 1 -1];
G1=[1 1 1 -1;1 -1 -1 -1; 1 1 1 -1 ; 1 1 1 -1];
G2=[-1 1 1 1;-1 1 -1 -1; -1 1 1 1; -1 1 1 1];
F1=[1 1 1 -1;1 1 -1 -1;1 -1 -1 -1;1 -1 -1 -1];
F2=[-1 1 1 1;-1 1 1 -1;-1 1 -1 -1;-1 1 -1 -1];

gg=colormap(gray); gg=gg(end:-1:1,:);

subplot(2,3,1); imagesc(T1); colormap(gg)
subplot(2,3,2); imagesc(G1);
subplot(2,3,3); imagesc(F1);
subplot(2,3,4); imagesc(T2);
subplot(2,3,5); imagesc(G2);
subplot(2,3,6); imagesc(F2);

%% Main code start

X=[T1(:) T2(:) G1(:) G2(:) F1(:) F2(:)]; %X is 16 x 6
T=[60 60 0 0 -60 -60];
alpha=0.03;

NumPoints=6; %Number of training points
NumEpochs=60; %An epoch is one pass through the data.

[W,b,EpochErr]=WidHoff(X,T,alpha,NumEpochs);

%% Output results
W*X+b*ones(1,NumPoints)
figure(2)
plot(EpochErr);

```

The plot of the error is shown in Figure 10.4. The horizontal axis counts the number of passes through the data, and the vertical axis gives the sum of the squared errors. Note that after 60 passes, we get very good classification of the letters!

10.4.4 Exercise: Pattern Classification

Let's put all of this together to solve a pattern classification problem. 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 10.5. There are several ways of performing the desired mapping- for example, the outputs could be 1, 2, 3, 4. But this may have unintended consequences. In this case, the metric between outputs would imply that Class 4 is much farther away from Class 1 than Class 3. A better method may be to take Class 1 to be the vector $[-1, -1]^T$, Class 2 is the vector $[-1, 1]^T$, Class 3 is $[1, -1]^T$, and Class 4 is $[1, 1]^T$. Now 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 .

Set the learning to 0.04. Set the initial weights to the 2×2 identity, and the initial bias vector \mathbf{b} to $[1, 1]^T$.

Let the program run until you think it has converged (this is your choice). We can also set an error bound so that we might stop early. In our code, we would add an “if-statement” that breaks off the computation if our error is small enough. In the code (like after assigning a value to α , you would set a value for the error tolerance, `ErrTol` in this case. Then we would add:

```
if EpochErr(k)<ErrTol
    break;
end
```

See if you can determine where this code fragment should be added.

Finally, plot the points $\mathbf{x} = [x, y]^T$ so that $W\mathbf{x} + \mathbf{b} = \mathbf{0}$, which will be the two lines:

$$W_{11}x + W_{12}y + b_1 = 0, \quad W_{21}x + W_{22}y + b_2 = 0$$

These lines form what is called the *decision boundary*. Here is one way to do that:

```
tt=linspace( min(X(1,:)), max(X(2,:)) );
yy1=(W(1,1)/W(1,2))*tt+(b(1)/W(1,2));
yy2=(W(2,1)/W(2,2))*tt+(b(2)/W(2,2));
plot(tt,yy1,'r',tt,yy2,'b');
hold on
plot(X(1,:),X(2,:), 'k*');
hold off
```

10.5 Batch Training

In batch training, all of the data is available to us at the start. In this case, we will assume that that we are working with the linearized version of the neural net, where the matrix of weights is appended with the bias vector at the end, and the input vectors \mathbf{x} have an extra “1” appended to them.

To be specific about dimensions, suppose we have p data pairs, where the domain is in \mathbb{R}^n and the range is in \mathbb{R}^m . Then for each $\mathbf{x}^{(i)} \in \mathbb{R}^n$, the linear network output is $\mathbf{y}^{(i)}$ that hopefully approximates our target $\mathbf{t}^{(i)}$. This implies that the weight matrix W is $m \times n$ and $\mathbf{b} \in \mathbb{R}^m$:

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

We can put this in matrix form:

$$W \begin{bmatrix} \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \dots & \mathbf{x}^{(p)} \end{bmatrix} + [\mathbf{b}, \mathbf{b}, \dots, \mathbf{b}] = \begin{bmatrix} \mathbf{y}^{(1)} & \mathbf{y}^{(2)} & \dots & \mathbf{y}^{(p)} \end{bmatrix}$$

We transform this into a linear problem by appending \mathbf{b} to the last column of W and we put a row of 1’s as the bottom row in the data:

$$\hat{W} = [W \quad \mathbf{b}], \quad \hat{X} = \begin{bmatrix} \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \dots & \mathbf{x}^{(p)} \\ 1 & 1 & \dots & 1 \end{bmatrix}$$

Now the output of the linear network is simply

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

and we solve the following system of equations for \hat{W} :

$$\hat{W} \hat{X} = T$$

(Be sure you can give the dimensions of each of those three arrays!)

In this form, we are solving the following for X : $XA = B$. In Matlab, we will use the slash command to solve this. If the matrix A is invertible, the slash will compute the inverse, and if the matrix is not invertible, the slash command will find the least squares solution. In either case, in Matlab, our matrix is easy to find:

```
hatW=T/hatX
```

Example: Associate Memory, Part II

Going back over the associate memory model, we'll try to solve the problem using batch training. The first part of the code is the same until after we set up the vector T . Use Matlab's matrix division to solve it. Then we have:

```
%% Main code start

X=[T1(:) T2(:) G1(:) G2(:) F1(:) F2(:)]; %X is 16 x 6
T=[60 60 0 0 -60 -60];
NumPoints=6;

hatX=[X;ones(1,NumPoints)]; %Puts a row of 1's in the bottom

hatW=T/hatX;

W=hatW(1:16);
b=hatW(17);

%% Output Results
Y=W*X+b*ones(1,NumPoints)
```

Exercise:

Find the linear neural network using batch training for the mapping from X to T (data is ordered) given below.

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

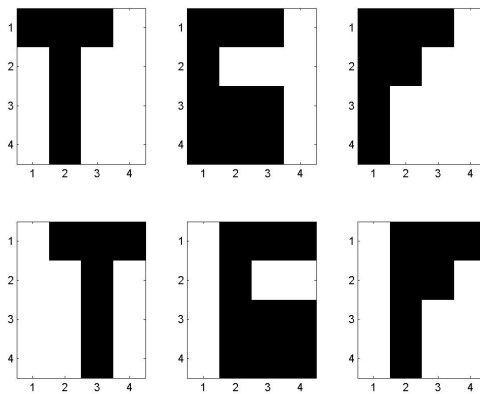


Figure 10.3: The inputs to our linear associative memory model: Three letters, T, G, H , where we have two samples of each letter, and each letter is defined by a 4×4 grid of numbers. We'll be associating T with -60 , G with 0 , and H with 60 .

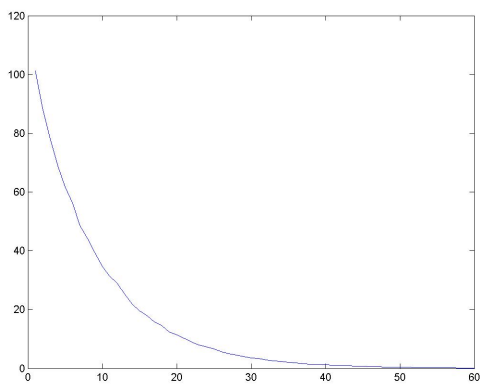


Figure 10.4: The errors for the Widrow-Hoff rule applied to letter recognition (or associative memory. After 60 passes through the data, the associations are very good.

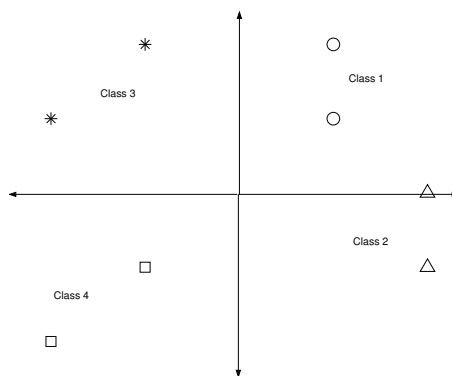


Figure 10.5: Pattern Classification Problem. Each point is a sample of one of the four classes.