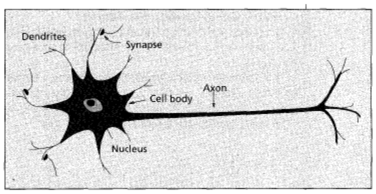
Building Classifiers: Part 3

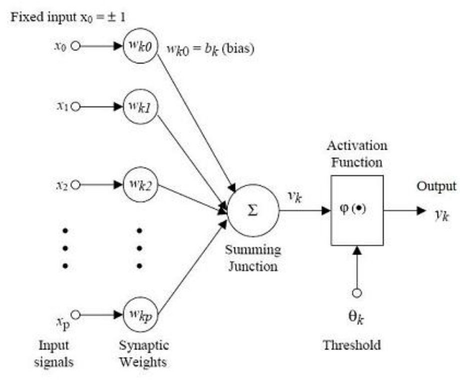
Ishwar K Sethi

Neural Network Methods

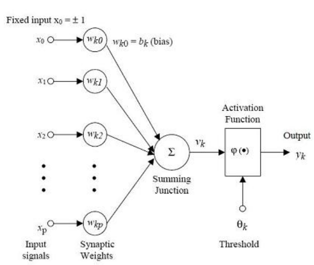
- Inspired by biological neurons
- *Non-algorithmic* or *model-free* approach to solve complex tasks
- Train a large number of interconnected elementary processing elements, called *neurons,* to solve the task at hand

Biological & Artificial Neurons





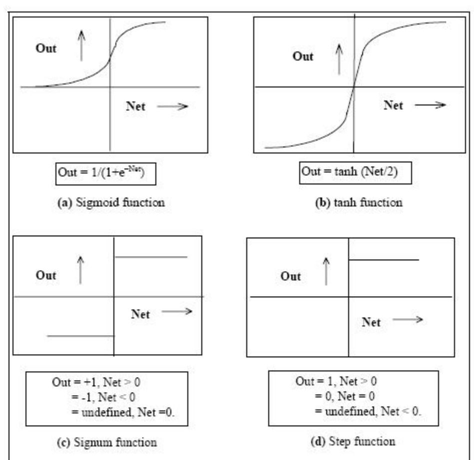
Activation Functions



Net

$$v_k = \sum_{j=1}^{p} w_{kj} x_j$$

$$\varphi(v) = \tanh\left(\frac{v}{2}\right) = \frac{1-\exp(-v)}{1+\exp(-v)}$$



Common non-linear functions used for synaptic inhibition. Soft non-linearity: (a) Sigmoid and (b) tanh; Hard non-linearity: (c) Signum and (d) Step.

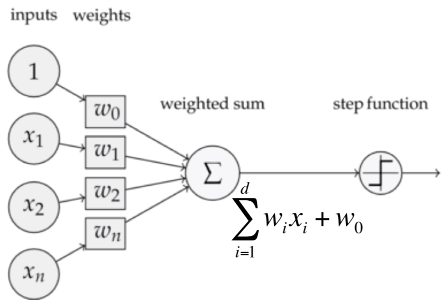
Learning Modes

- Supervised Learning
- Self-Organizing or Unsupervised Learning
- Reinforcement Learning

Single-Layer Perceptron Network Model

- Perceptron learning rule
  - Iterative error-correction procedure
  - Learning may never cease for some problems
  - Pocket algorithm
- Delta rule
  - Error minimization procedure
  - Guaranteed convergence
  - Linear models only
  - May produce locally minimum solutions

Perceptron Classifier



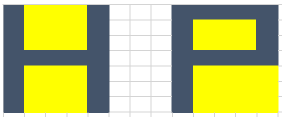
The perceptron classifier sums the weighted input and generates an output of +1 if the weighted sum is positive; otherwise an output of -1 is generated. Designing/modeling a perceptron classifier involves finding weights given a collection of training examples from two classes.

Perceptron Learning Rule

- Uses training data from two classes
- Starts with a random weight vector
- Compute the perceptron output for each training example one by one
- If the output matches with the desired output, leave the weight vector unchanged and move to the next example
- If the output is not correct, modify the weight vector either by adding to it or by subtracting from it the current training vector, and move to the next example
- Continue repeating until perceptron produces correct output for all examples.

The perceptron output is calculated by computing the weighted sum of the input signals and comparing the result with a threshold value. If the net input is less than the threshold, the perceptron output is –1. But if the net input is greater than or equal to the threshold, the perceptron becomes activated and its output attains a value +1.

Perceptron Demo



1	0	0	0	1	1	0	0	0	1	1	0	0	0	1	1	1	1	1	1	1	0	0	0	1	1	0	0	0	1	1	0	0	0	1	
1	1	1	1	1	1	0	0	0	1	1	0	0	0	1	1	1	1	1	1	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	
1	1	0	0	1	1	0	0	0	1	1	0	1	0	0	0	1	1	0	0	0	1	1	0	1	0	0	0	1	1	0	0	0	1	1	Initi
10																																			
8																																			P m
0	0	-1	-1	0	0	0	0	0	0	0	0	1	0	-1	-1	0	0	-1	-1	-1	1	1	0	1	-1	0	0	1	1	-1	0	0	1	1	
-4																																			H m
1	0	-1	-1	1	1	0	0	0	1	1	0	1	0	0	0	1	1	0	0	0	1	1	0	2	0	0	0	1	2	0	0	0	1	2	
5																																			P m
0	-1	-2	-2	0	0	0	0	0	0	0	0	1	0	-1	-1	0	0	-1	-1	-1	1	1	0	2	-1	0	0	1	2	-1	0	0	1	2	
-1																																			H m
1	-1	-2	-2	1	1	0	0	0	1	1	0	1	0	0	0	1	1	0	0	0	1	1	0	3	0	0	0	1	3	0	0	0	1	3	
2																																			P m
0	-2	-3	-3	0	0	0	0	0	0	0	0	1	0	-1	-1	0	0	-1	-1	-1	1	1	0	3	-1	0	0	1	3	-1	0	0	1	3	
2	OK																												Solution Vector						
-15	OK																																		

Augmented Vectors

$$\sum_{i=1}^{d} w_i x_i + w_0 \geq 0 \text{ Then Output} = 1$$

$$\sum_{i=1}^{d} w_i x_i + w_0 < 0 \text{ Then Output} = \text{-}1$$

Define $\quad \mathbf{y}^t = [\mathbf{x}^t 1] = [x_1 x_2 \ldots x_d 1]$

$\quad \mathbf{a}^t = [\mathbf{w}^t 1] = [w_1 w_2 \ldots w_d 1]$

Augmented pattern and weight vectors. Augmentation makes equations simpler.

Then

$$\mathbf{a}^t \mathbf{y} \geq 0 \text{ Then Output} = 1$$

$$\mathbf{a}^t \mathbf{y} < 0 \text{ Then Output} = \text{-}1$$

Finding the Weight Vector Using Training Data

Linearly Separable Case

- Let $\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots, \boldsymbol{y}_n$ be a set of n examples in augmented feature space, which are linearly separable.
- We need to find a weight vector **a** such that
  - $\mathbf{a}^t\mathbf{y} > 0$ for examples from the positive class.
  - $\mathbf{a}^t\mathbf{y} < 0$ for examples from the negative class.
- Normalizing the input examples by multiplying them with their class label (replace all samples from class 2 by their negatives), find a weight vector **a** such that
  - $\mathbf{a}^t\mathbf{y} > 0$ for all the examples (here **y** is multiplied with class label)
- Such a weight vector is called a *separating vector* or a *solution vector*

Perceptron Criterion Function

- Goal: Find a weight vector **a** such that $\mathbf{a}^t\mathbf{y} > 0$ for all the examples (assuming it exists).
- Mathematically, this can be expressed as finding an **a** that minimizes the number of samples misclassified
  - Function is piecewise constant (discontinuous, and hence non-differentiable) and is difficult to optimize
- Perceptron Criterion Function:

Find an **a** that minimizes this criterion.

$$J_p(\mathbf{a}) = \sum_{\boldsymbol{y} \in \mathcal{Y}} (-\mathbf{a}^t\mathbf{y})$$

The criterion is proportional to the sum of distances from the misclassified samples to the decision boundary

Minimization is mathematically tractable, and hence it is a better criterion function than number of misclassifications.

Gradient Descent

A very simple idea:
1. Pick a starting point
2. Calculate gradient
3. Update using the equation shown
4. Stop updating under certain conditions

$$\underset{x}{\text{minimize}}\ \ f(x)$$

$$x_{t+1} = x_t - \eta_t \nabla f(x_t)$$

Directional Derivatives : Along the Axes…

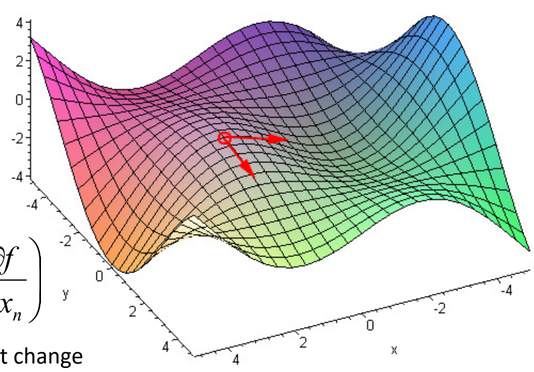
$$\frac{\partial f(x,y)}{\partial y}$$

$$\frac{\partial f(x,y)}{\partial x}$$

$$\nabla f(x,y) := \left(\frac{\partial f}{\partial x} \quad \frac{\partial f}{\partial y} \right)$$

$$\nabla f(x_1,\ldots,x_n) := \left(\frac{\partial f}{\partial x_1}, \ldots, \quad \frac{\partial f}{\partial x_n} \right)$$

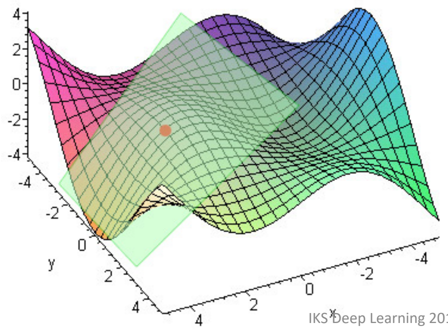
$$f := (x,y) \to \cos\left(\frac{1}{2}x\right)\cos\left(\frac{1}{2}y\right)x$$



The gradient points at the greatest change direction

The Gradient Properties

- The gradient defines (hyper) plane approximating the function infinitesimally



$$\Delta z = \frac{\partial f}{\partial x} \cdot \Delta x + \frac{\partial f}{\partial y} \cdot \Delta y$$

Minimization Example

Consider the problem:

Minimize $f(x_1, x_2, x_3) = (x_1)^2 + x_1(1 - x_2) + (x_2)^2 - x_2x_3 + (x_3)^2 + x_3$

First, we find the gradient with respect to x_i:

$$\nabla f = \begin{bmatrix} 2x_1 + 1 - x_2 \\ -x_1 + 2x_2 - x_3 \\ -x_2 + 2x_3 + 1 \end{bmatrix}$$

Minimization Example Contd.

Next, we set the gradient equal to zero:

$$\nabla f = 0 \qquad \Rightarrow \qquad \begin{bmatrix} 2x_1 + 1 - x_2 \\ -x_1 + 2_2 - x_3 \\ -x_2 + 2x_3 + 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

So, we have a system of 3 equations and 3 unknowns. When we solve, we get:

First we solve it using analytical approach

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix}$$

Soln.

Gradient Descent Approach

Minimize $f(x_1, x_2, x_3) = (x_1)^2 + x_1(1 - x_2) + (x_2)^2 - x_2x_3 + (x_3)^2 + x_3$

Let's pick

$$\mathbf{x}^0 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Gradient Descent Example

$$\nabla f(\mathbf{x}) = [2x_1 + (1 - x_2) \quad -x_1 + 2x_2 - x_3 \quad -x_2 + 2x_3 + 1]$$

$$\nabla f(X_0) = [2(0) + 1 - 0 - 0 + 0 - 0 - 0 + 0 + 1]^t$$

$$= [101]^t$$

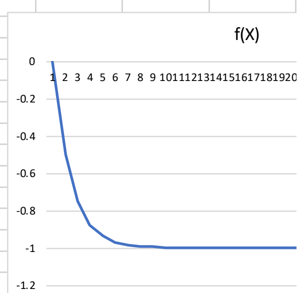
$$X_1 = X_0 - \alpha * \nabla f(X_0)$$

Let α be 0.5. Then

X_1 = [-0.5 0 -0.5]^t

IKS Deep Learning 2017

X1	X2	X3	f(X)	delx1	delx2	delx3	alpha
0	0	0	0	1	0	1	0.5
-0.5	0	-0.5	-0.5	0	1	0	
-0.5	-0.5	-0.5	-0.75	0.5	0	0.5	
-0.75	-0.5	-0.75	-0.875	0	0.5	0	
-0.75	-0.75	-0.75	-0.9375	0.25	0	0.25	
-0.875	-0.75	-0.875	-0.96875	0	0.25	0	
-0.875	-0.875	-0.875	-0.984375	0.125	0	0.125	
-0.9375	-0.875	-0.9375	-0.9921875	0	0.125	0	
-0.9375	-0.9375	-0.9375	-0.9960938	0.0625	0	0.0625	
-0.96875	-0.9375	-0.96875	-0.9980469	0	0.0625	0	
-0.96875	-0.96875	-0.96875	-0.9990234	0.03125	0	0.03125	
-0.984375	-0.96875	-0.984375	-0.9995117	0	0.03125	0	
-0.984375	-0.984375	-0.984375	-0.9997559	0.015625	0	0.015625	
-0.9921875	-0.984375	-0.9921875	-0.9998779	0	0.015625	0	
-0.9921875	-0.9921875	-0.9921875	-0.999939	0.0078125	0	0.0078125	
-0.9960938	-0.9921875	-0.9960938	-0.9999695	0	0.0078125	0	
-0.9960938	-0.9960938	-0.9960938	-0.9999847	0.00390625	0	0.00390625	
-0.9980469	-0.9960938	-0.9980469	-0.9999924	0	0.00390625	0	
-0.9980469	-0.9980469	-0.9980469	-0.9999962	0.00195313	0	0.00195313	
-0.9990234	-0.9980469	-0.9990234	-0.9999981	0	0.00195313	0	
-0.9990234	-0.9990234	-0.9990234	-0.999999	0.00097656	0	0.00097656	
-0.9995117	-0.9990234	-0.9995117	-0.9999995	0	0.00097656	0	
-0.9995117	-0.9995117	-0.9995117	-0.9999998	0.00048828	0	0.00048828	
-0.9997559	-0.9995117	-0.9997559	-0.9999999	0	0.00048828	0	
-0.9997559	-0.9997559	-0.9997559	-0.9999999	0.00024414	0	0.00024414	
-0.9998779	-0.9997559	-0.9998779	-1	0	0.00024414	0	



Iterative Optimization

- Do the following:
  1. Choose a search direction $\mathbf{d}^k$
  2. Minimize along that direction to find a new point:

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \alpha^k \mathbf{d}^k$$

where k is the current iteration number and α^k is a positive scalar called the step size. In practice α^k is taken a small number less than 1. Often it remains constant for all *k*.

Steepest Descent/ Gradient Search Method

- This method is very simple – it uses the gradient (for maximization) or the negative gradient (for minimization) as the search direction:

$$\mathbf{d}^k = \begin{Bmatrix} + \\ - \end{Bmatrix} \nabla f(\mathbf{x}^k) \text{ for } \begin{Bmatrix} \max \\ \min \end{Bmatrix}$$

So, $$\mathbf{x}^{k+1} = \mathbf{x}^k \begin{Bmatrix} + \\ - \end{Bmatrix} \alpha^k \nabla f(\mathbf{x}^k)$$

Gradient/Steepest Descent Method Steps

So the steps of the Steepest Descent Method are:

1. Choose an initial point $\mathbf{x}^0$
2. Calculate the gradient $\nabla f(\mathbf{x}^k)$ where k is the iteration number
3. Calculate the search vector:
4. Calculate the next $\mathbf{x}$:
   Use a single-variable optimization method to determine α^k for optimal step size.

$$\mathbf{d}^k = \pm\nabla f(\mathbf{x}^k)$$

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \alpha^k \mathbf{d}^k$$

Steepest Descent Method Steps

5. To determine convergence, either use some given tolerance ε_1 and evaluate:

   $$\left| f(\mathbf{x}^{k+1}) - f(\mathbf{x}^k) \right| < \varepsilon_1$$

   for convergence

Or, use another tolerance ε_2 and evaluate:

$$\left\| \nabla f(\mathbf{x}^k) \right\| < \varepsilon_2$$

for convergence

Perceptron Learning

- Perceptron criterion function J(**a)** is a scalar function of vector variables. This can be minimized using gradient descent.
- Gradient Descent Procedure:
  - Start with an arbitrarily chosen weight vector **a**(1)
  - Compute the gradient vector $\nabla J(\mathrm{a}(1))$
  - The next value **a**(2) is obtained by moving in the direction of the steepest descent, i.e. along the negative of the gradient.
  - In general, the *k+1*-th solution is obtained by

$$\mathbf{a}(k+1) = \mathbf{a}(k) - \eta(k)\boldsymbol{\nabla} J(\mathbf{a}(k))$$

Perceptron Learning

- Use gradient descent to find **a**
  - Move in the negative direction of the gradient iteratively to reach the minima.
- The gradient vector is given by,

$$\boldsymbol{\nabla} J_p = \sum_{\mathbf{y}\in\mathcal{Y}} (-\mathbf{y})$$

- Starting from **a = 0**, update **a** at each iteration *k* as follows:

$$\underbrace{\mathbf{a}(k+1)}_{\text{Updated } \mathbf{a}} = \underbrace{\mathbf{a}(k)}_{\text{Current } \mathbf{a}} + \eta(k) \underbrace{\sum_{\mathbf{y}\in\mathcal{Y}_k} \mathbf{y}}_{\text{Direction of update}}$$

Learning rate/ Stepsize: Magnitude of update

Perceptron Learning: Incremental Rule

- Computes the gradient using a single sample
  - Also called perceptron learning in an online setting
- For large datasets, this is much more efficient compared to batch descent (O(n) vs O(1) gradient computation in batch vs single-sample)

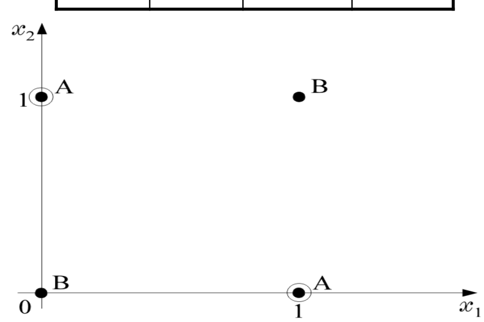
Algorithm 4 (Fixed-increment single-sample Perceptron)

```
begin initialize  a, k = 0
          do  k ← (k + 1)mod n
                 if  y_k is misclassified by a  then  a ← a + y_k
          until all patterns properly classified
     return a
end
```

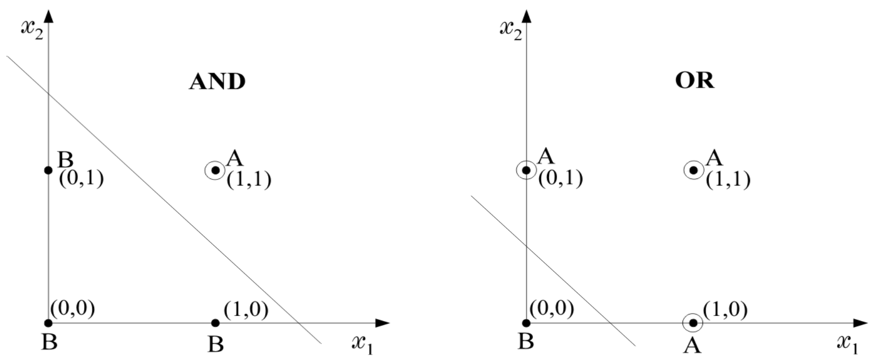
Iterative update step.
Notice the gradient.

XOR Problem

x_1	x_2	XOR	Class
0	0	0	B
0	1	1	A
1	0	1	A
1	1	0	B

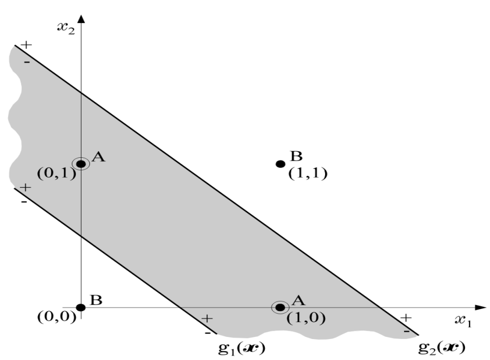


There is no single line (hyperplane) that separates class A from class B. On the contrary, AND and OR operations are linearly separable problems



Two Layer Perceptron

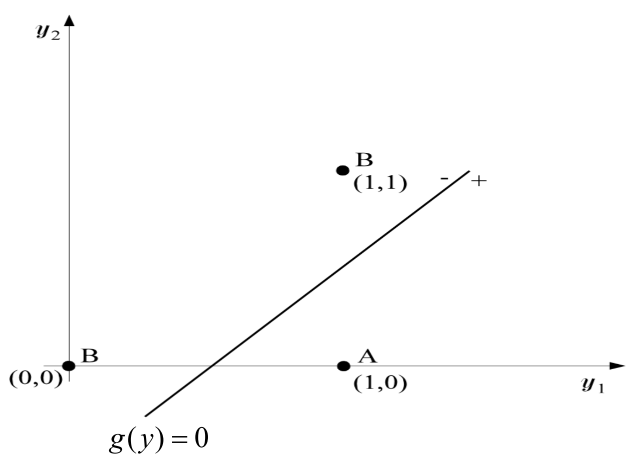
- Lets draw two lines, instead of one



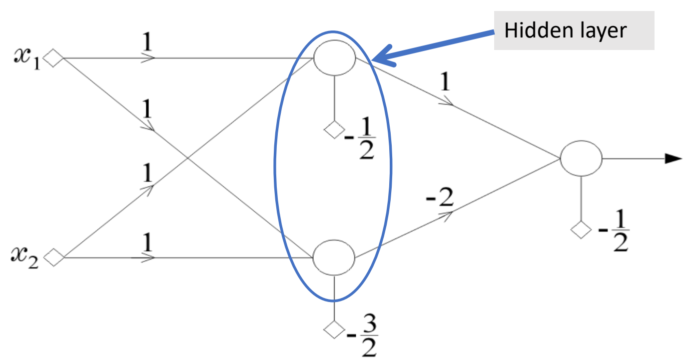
- Then class B is located outside the shaded area and class A inside. One can look at this arrangement in two steps:
  - Step 1: The first stage perceptrons perform a mapping of input producing output depending on input's position
  - Step 2: Use the values of y_1, y_2 to do classification

1st Stage				2nd Stage
x_1	x_2	y_1	y_2	
0	0	0(-)	0(-)	B(0)
0	1	1(+)	0(-)	A(1)
1	0	1(+)	0(-)	A(1)
1	1	1(+)	1(+)	B(0)

The decision is now performed on the transformed/mapped data using another perceptron



- Computations of the first phase perform a mapping that transforms the nonlinearly separable problem to a linearly separable one.
- The architecture



- The idea can be extended by having multiple layers

Linearly Non-separable Case

- There is no way of knowing before training whether a given collection of training examples is linearly separable or not
- Perceptron learning procedure may not converge. Several heuristics are used.
  - Decreasing the step size as training progresses
  - Averaging of weight vectors over training
  - Pocket algorithm (Gallant)
  - Multiple layers of perceptrons as shown by the XOR example

Minimum Squared Error (MSE) Criterion

- Perceptron criterion function focused only on errors. Mean-squared error (MSE) procedures involve all the samples.
- Using matrix notation for convenience

Weight vector.

Pattern matrix

$$\begin{pmatrix} Y_{10} & Y_{11} & \cdots & Y_{1d} \\ Y_{20} & Y_{21} & \cdots & Y_{2d} \\ \vdots & \vdots & & \vdots \\ \vdots & \vdots & & \vdots \\ \vdots & \vdots & & \vdots \\ Y_{n0} & Y_{n1} & \cdots & Y_{nd} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_d \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ \vdots \\ \vdots \\ b_n \end{pmatrix}$$

Desired Targets

$$\text{or} \qquad \mathbf{Ya} = \mathbf{b}.$$

- MSE Criterion: Minimize the sum of squared differences between Ya and b:

$$J_s(\mathbf{a}) = \|\mathbf{Ya} - \mathbf{b}\|^2 = \sum_{i=1}^{n} (\mathbf{a}^t\mathbf{y}_i - b_i)^2.$$

Minimum Squared Error (MSE) Criterion

- While a gradient search can be used to solve MSE, a closed form solution can be obtained in many cases.
- Computing the gradient gives:

$$\nabla J_s = \sum_{i=1}^{n} 2(\mathbf{a}^t\mathbf{y}_i - b_i)\mathbf{y}_i = 2\mathbf{Y}^t(\mathbf{Ya} - \mathbf{b})$$

- Setting the gradient to zero,

$$\mathbf{Y}^t\mathbf{Ya} = \mathbf{Y}^t\mathbf{b}.$$

- The solution for **a** can be obtained uniquely if **Y**^{t}**Y** is non-singular.

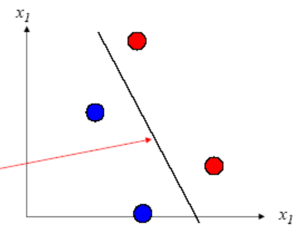
$$\begin{aligned} \mathbf{a} &= (\mathbf{Y}^t\mathbf{Y})^{-1}\mathbf{Y}^t\mathbf{b} \\ &= \mathbf{Y}^\dagger\mathbf{b}, \end{aligned}$$

Pseudo-inverse of the rectangular pattern matrix Y.

$$\mathbf{Y}^\dagger \equiv (\mathbf{Y}^t\mathbf{Y})^{-1}\mathbf{Y}^t$$

Example of Linear Classifier by Pseudoinverse

- ω_1: $(1,2)^t$ and $(2,0)^t$
- ω_2: $(3,1)^t$ and $(2,3)^t$



$$a^t \begin{pmatrix} 1 \\ x_1 \\ x_2 \end{pmatrix} = 0$$

Sample Matrix (d = 1+2, n = 4)

$$Y = \begin{bmatrix} 1 & 1 & 2 \\ 1 & 2 & 0 \\ -1 & -3 & -1 \\ -1 & -2 & -3 \end{bmatrix}$$

Pseudo-inverse

$$Y^{*} = (Y^tY)^{-1}Y^t = \begin{bmatrix} 5/4 & 13/12 & 3/4 & 7/12 \\ -1/2 & -1/6 & -1/2 & -1/6 \\ 0 & -1/3 & 0 & -1/3 \end{bmatrix}$$

Assuming $b = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$

our solution is $a = Y^{*}b = \begin{bmatrix} 11/3 \\ -4/3 \\ -2/3 \end{bmatrix}$

Minimum Squared Error (MSE) Criterion

MSE Solution using Gradient Descent:

- Criterion function $J_s(a) = ||Ya-b||^2$ could be minimized by gradient descent
- Advantage over pseudo-inverse:
  - Problem when Y^tY is singular
  - Avoids need for working with large matrices
  - Computation involved is a feedback scheme that copes with roundoff or truncation

Minimum Squared Error (MSE) Criterion

Widrow-Hoff or Least Mean Squared (LMS) Rule

Since $\Delta J_s = 2\mathbf{Y}^t(\mathbf{Ya} - \mathrm{b})$

The obvious update rule is

a(1) arbitrary

$\mathrm{a}(k+1) = \mathrm{a}(k) + \eta(k)\mathbf{Y}^t(\mathbf{Ya}(k)-\mathrm{b})$

Also known as LMS rule/ADALINE/ Delta rule

Can be reduced for storage requirement to the rule where samples are considered sequentially:

a(1) arbitrary

$\mathrm{a}(k+1) = \mathbf{a}(k) + \eta(k)(\mathrm{b}(k) - \mathbf{a}^t(k)\mathbf{y}^k)\mathbf{y}^k$

LMS Example

Feature1	Feature2	Class	
2	2	1	0.1
3	1	1	
1	-1	2	
2	-3	2	
Augmented and Normalized			
2	2	1	
3	1	1	
-1	1	-1	
-2	3	-1	
Start Training			0.7
-0.3			0.1