Module -III & IV
Neural Network, Fuzzzy basic, EM, K-NNB, HMM

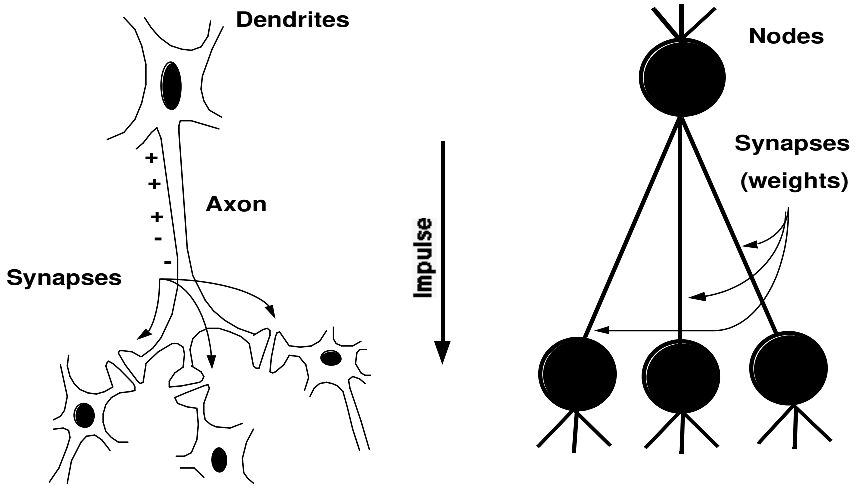
Syllabus

- Neural network-
  - A. Simon Haykin – Chapter 1(op),2(op), 3(imp),4(imp)
  - **B. Soft Computing – S.N Deepa – Chapter 2, 3**

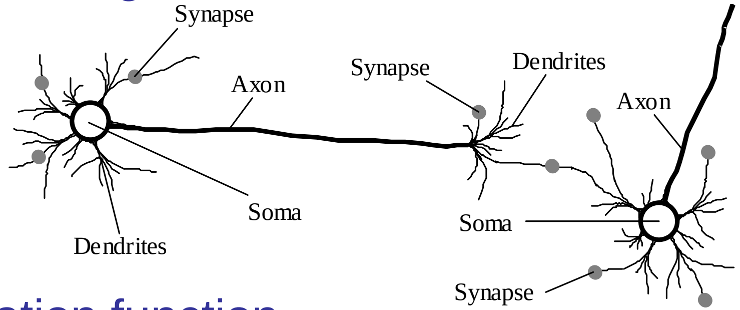
Fuzyy Basic -

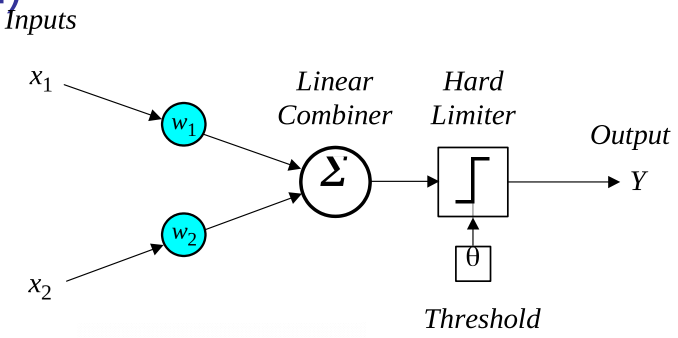
Neural Network

- Consider humans:
- Neuron switching time
  - ~ 0.001 second
- Number of neurons
  - ~ 10^{10}
- Connections per neuron
  - ~ 10^{4-5}
- Scene recognition time
  - ~ 0.1 second
- 100 inference steps doesn't seem like enough
  - much parallel computation
- Properties of artificial neural nets (ANN)
- Many neuron-like threshold switching units
- Many weighted interconnections among units
- Highly parallel, distributed processes



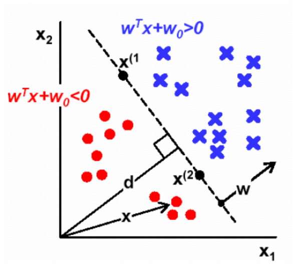
- From biological neuron to artificial neuron (perceptron)





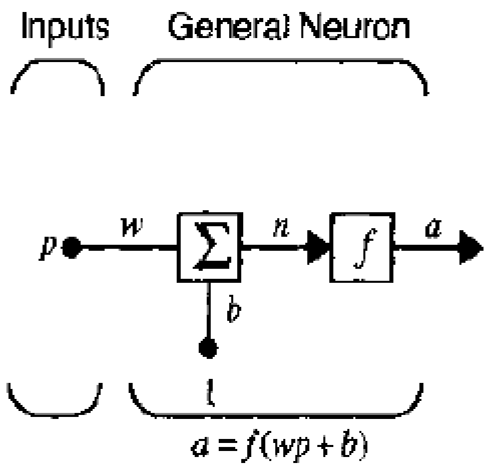
- Activation function

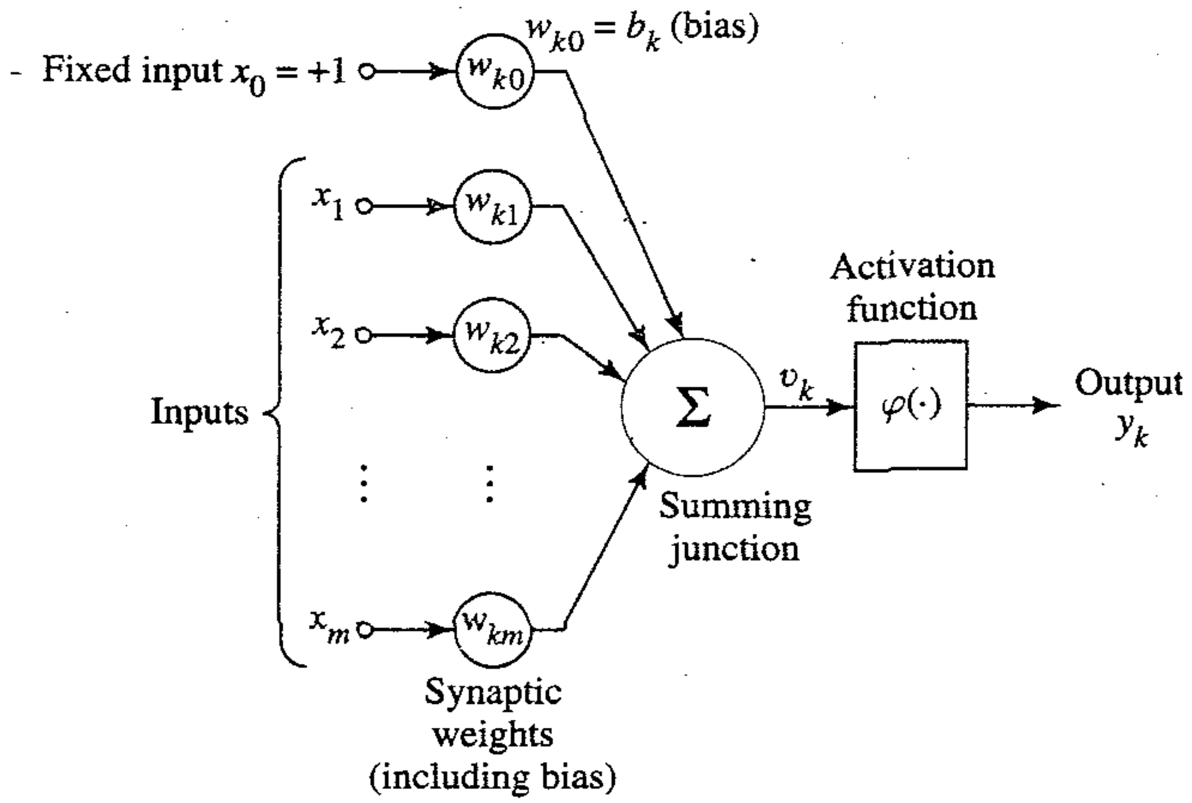
$$X = \sum_{i=1}^{n} x_i w_i \qquad \mathcal{Y} = \begin{cases} +1, & \text{if } X \geq \omega_0 \\ -1, & \text{if } X < \omega_0 \end{cases}$$



- Artificial neuron networks
  - supervised learning
  - gradient descent

Single-layer Neuron





Example

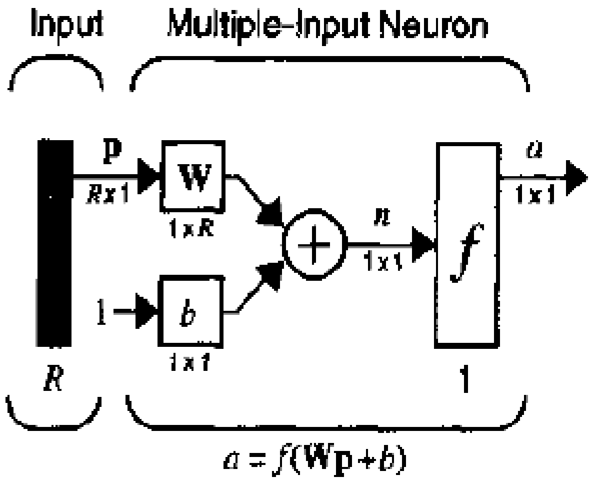
The input to a single-input neuron is 2.0, its weight is 2.3 and its bias is −3.

- **i. What is the net input to the transfer function?**
- **ii. What is the neuron output?**

i. The net input is given by:

$$n = wp + b = (2.3)(2) + (-3) = 1.6$$

ii. The output cannot be determined because the transfer function is not specified.

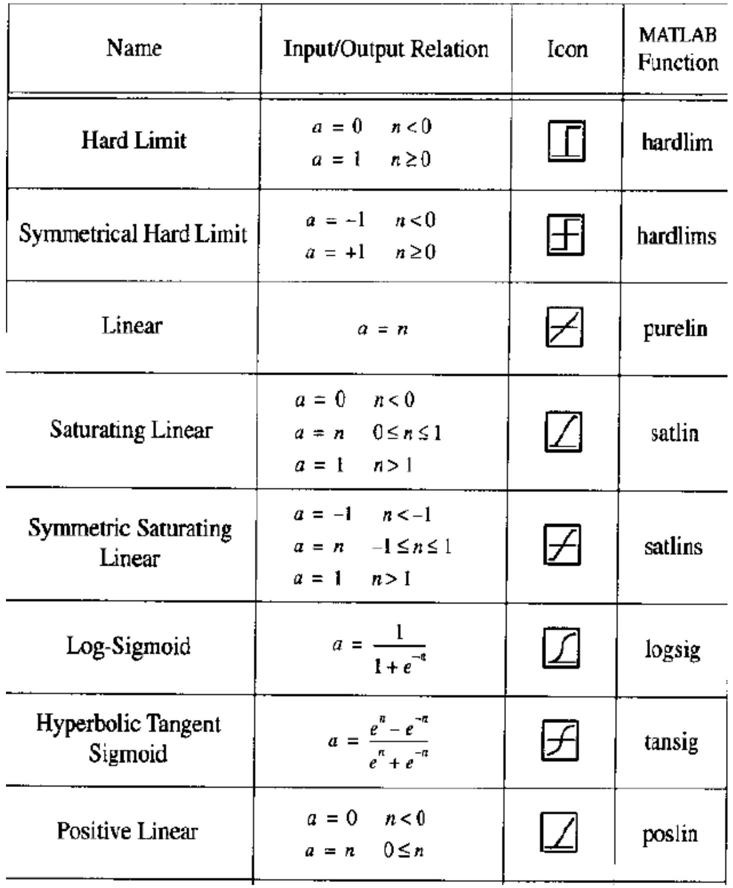


1st itr
Net = wP+b =(2 x 2.3) +(-3) = 1.6
Hard-limit:
A = transfer-function(net)= hard-limit(net) = hard-limit(1.6)=1
Symetrical Hard-limit:
A=transfer-function(net) = hardlims(1.6) =1
Linear: (purelin)
A = F(net) = F(1.6) = 1.6

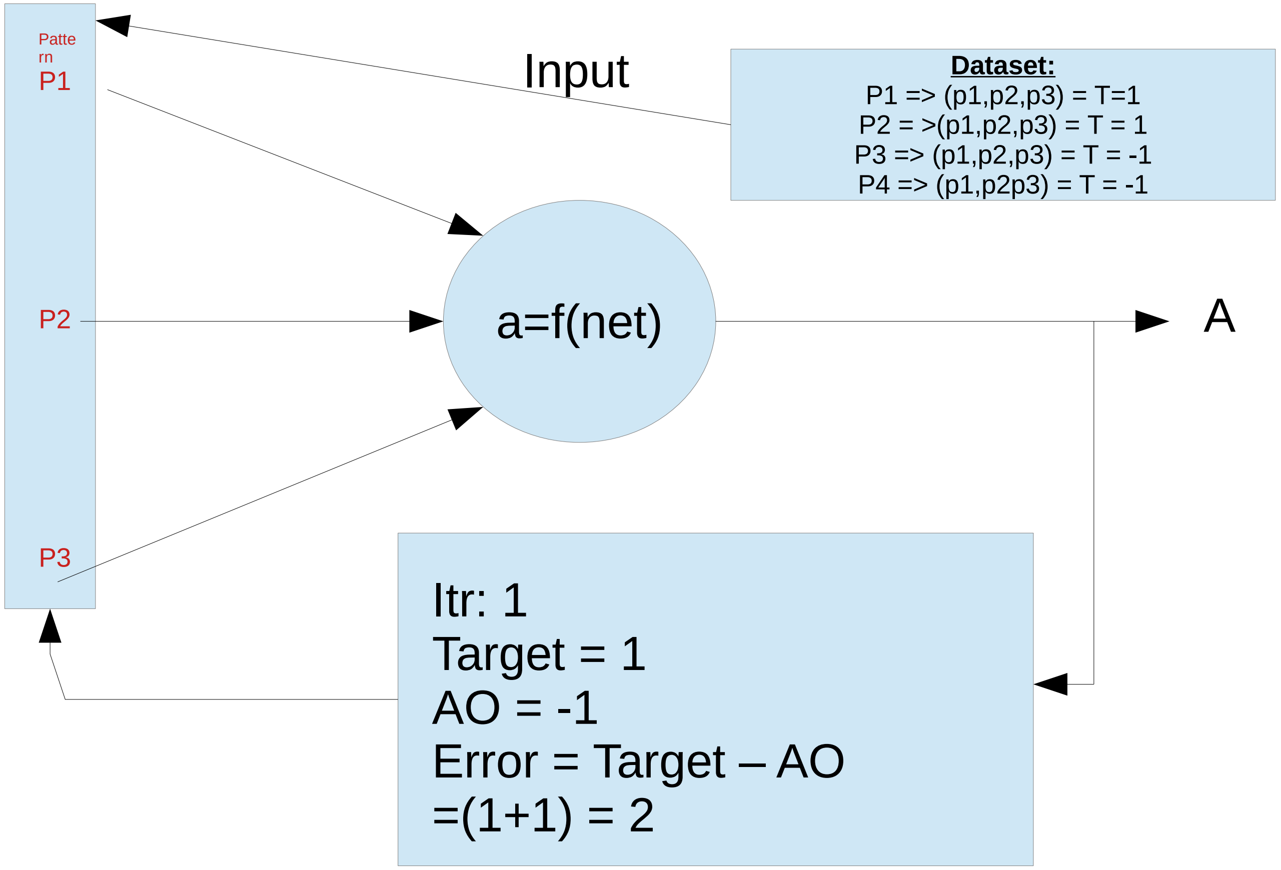
Transfer Function

- The transfer function may be linear or non-linear function of n or net.
- A particular transfer function is chosen to satisfy some specificatio problem that the neuron is attempting to solve.
- A transfer function , denoted by f(net) , defines the output of a neuron in terms of the local induced field net.

Transfer function



Name	Input/Output Relation	Icon	MATLAB Function
Hard Limit	$a = 0 \quad n < 0$ $a = 1 \quad n \geq 0$		hardlim
Symmetrical Hard Limit	$a = -1 \quad n < 0$ $a = +1 \quad n \geq 0$		hardlims
Linear	$a = n$		purelin
Saturating Linear	$a = 0 \quad n < 0$ $a = n \quad 0 \leq n \leq 1$ $a = 1 \quad n > 1$		satlin
Symmetric Saturating Linear	$a = -1 \quad n < -1$ $a = n \quad -1 \leq n \leq 1$ $a = 1 \quad n > 1$		satlins
Log-Sigmoid	$a = \frac{1}{1+e^{-n}}$		logsig
Hyperbolic Tangent Sigmoid	$a = \frac{e^{n}-e^{-n}}{e^{n}+e^{-n}}$		tansig
Positive Linear	$a = 0 \quad n < 0$ $a = n \quad 0 \leq n$		poslin



Dataset:
P1 => (p1,p2,p3) = T=1
P2 = >(p1,p2,p3) = T = 1
P3 => (p1,p2,p3) = T = -1
P4 => (p1,p2p3) = T = -1

Itr: 1
Target = 1
AO = -1
Error = Target – AO
=(1+1) = 2

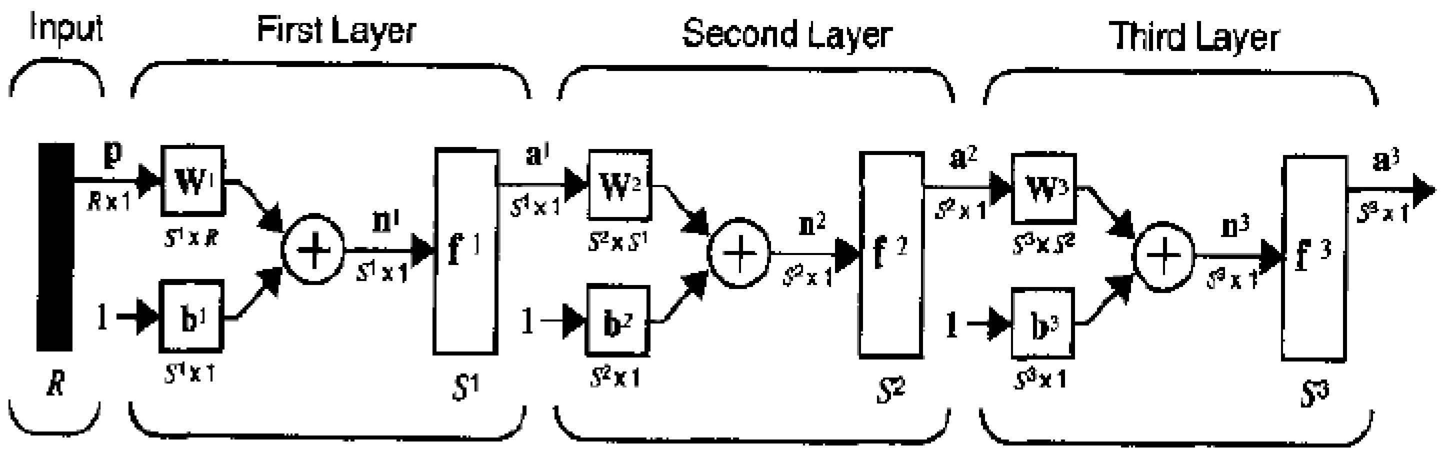
<u>Error Cal.:</u>
$SSE = Er .Er^T$
a=f(net)= weight*pattern+bia
Weight =bias=pattern => matrix

Perceptron Learning algorithms:

$W(new)= W(old)^i+\alpha*(Er)*pattern^i$
$B(new) = B(old)+\alpha*(Er)$

$\alpha \in (0,1)$

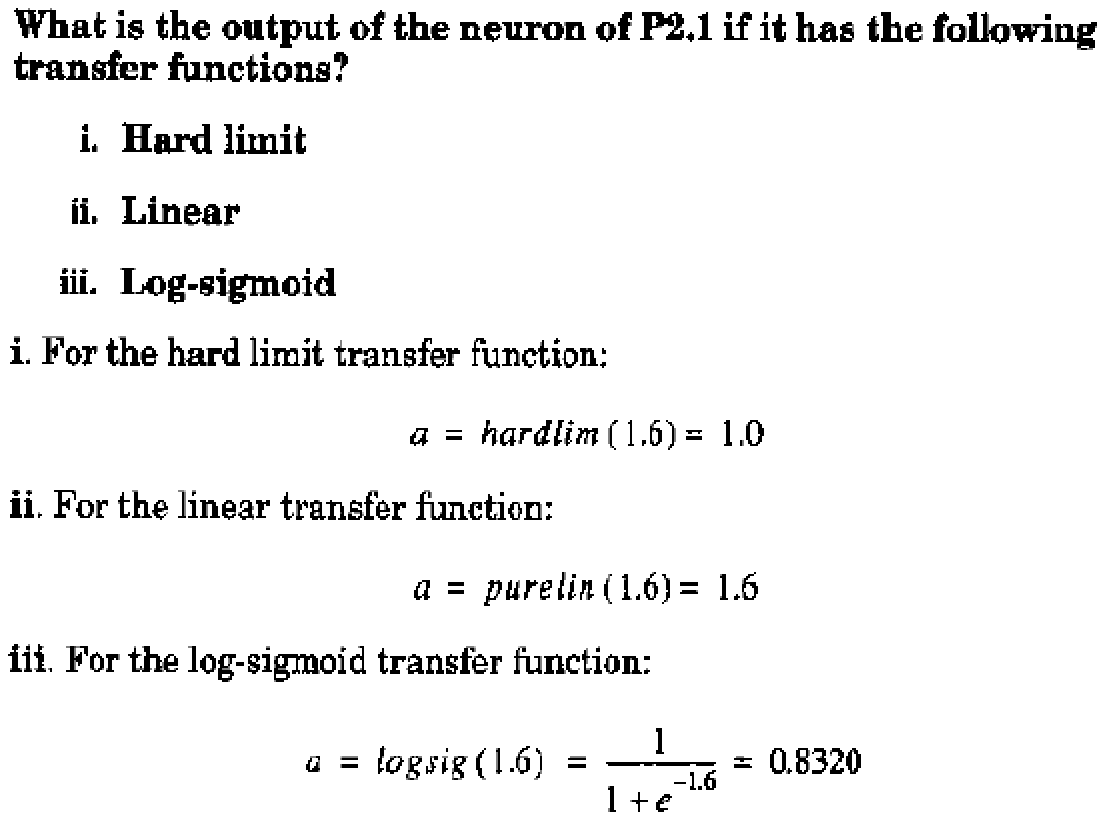
Multi-layer Neural Network



$$a^1 = f^1(W^1 p + b^1) \qquad a^2 = f^2(W^2 a^1 + b^2) \qquad a^3 = f^3(W^3 a^2 + b^3)$$

$$a^3 = f^3(W^3 f^2(W^2 f^1(W^1 p + b^1) + b^2) + b^3)$$

Example



What is the output of the neuron of P2.1 if it has the following transfer functions?

i. **Hard limit**

ii. **Linear**

iii. **Log-sigmoid**

i. For the hard limit transfer function:

$$a = hardlim(1.6) = 1.0$$

ii. For the linear transfer function:

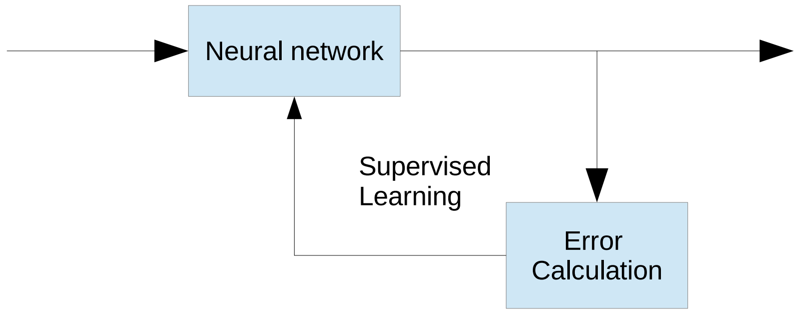
$$a = purelin(1.6) = 1.6$$

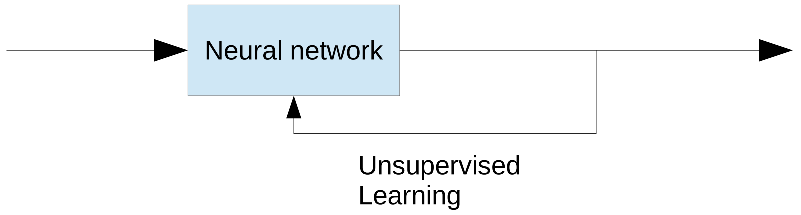
iii. For the log-sigmoid transfer function:

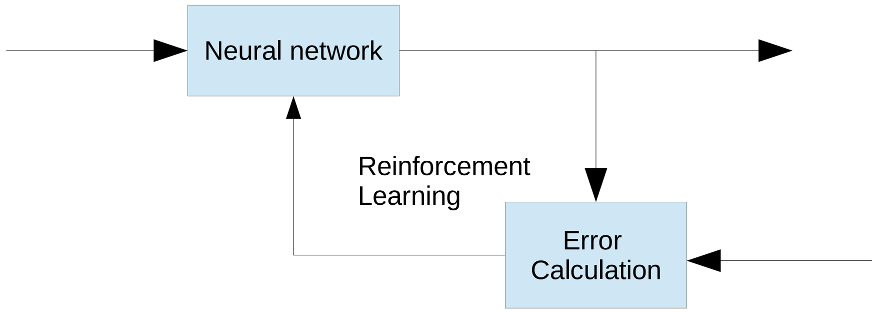
$$a = logsig(1.6) = \frac{1}{1+e^{-1.6}} = 0.8320$$

Learning

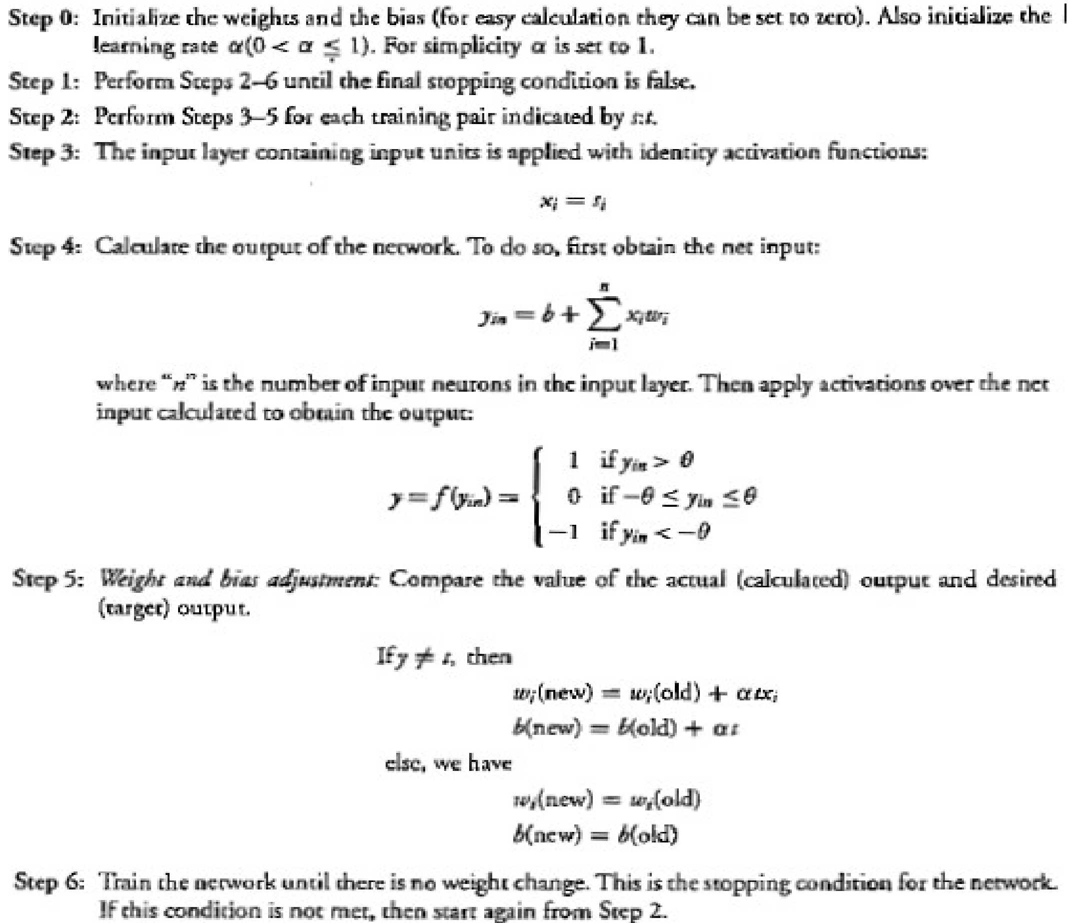
- Supervised learning
- Unsupervised learning
- Re-inforcement learning







Perceptron Training algo



Step 0: Initialize the weights and the bias (for easy calculation they can be set to zero). Also initialize the learning rate $\alpha (0 < \alpha \leq 1)$. For simplicity α is set to 1.

Step 1: Perform Steps 2–6 until the final stopping condition is false.

Step 2: Perform Steps 3–5 for each training pair indicated by $s:t$.

Step 3: The input layer containing input units is applied with identity activation functions:

$$x_i = s_i$$

Step 4: Calculate the output of the network. To do so, first obtain the net input:

$$y_{in} = b + \sum_{i=1}^{n} x_i w_i$$

where "n" is the number of input neurons in the input layer. Then apply activations over the net input calculated to obtain the output:

$$y = f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} > \theta \\ 0 & \text{if } -\theta \leq y_{in} \leq \theta \\ -1 & \text{if } y_{in} < -\theta \end{cases}$$

Step 5: *Weight and bias adjustment:* Compare the value of the actual (calculated) output and desired (target) output.

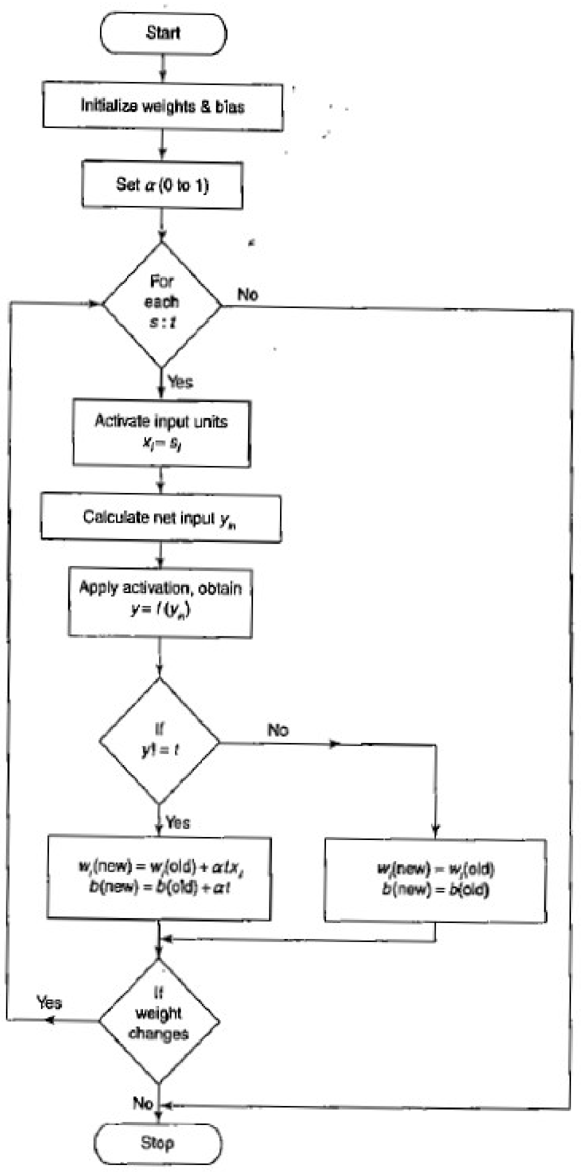
If $y \neq t$, then

$$w_i(\text{new}) = w_i(\text{old}) + \alpha t x_i$$
$$b(\text{new}) = b(\text{old}) + \alpha t$$

else, we have

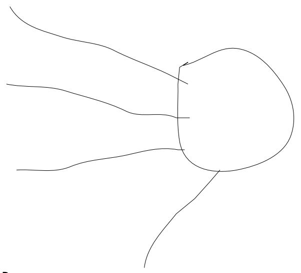
$$w_i(\text{new}) = w_i(\text{old})$$
$$b(\text{new}) = b(\text{old})$$

Step 6: Train the network until there is no weight change. This is the stopping condition for the network. If this condition is not met, then start again from Step 2.



Start
Initialize weights & bias
Set α (0 to 1)
For each $s:t$
No
Yes
Activate input units $x_i = s_i$
Calculate net input y_{in}
Apply activation, obtain $y = f(y_{in})$
If $y != t$
No
Yes
$w_i(\text{new}) = w_i(\text{old}) + \alpha t x_i$
$b(\text{new}) = b(\text{old}) + \alpha t$
$w_i(\text{new}) = w_i(\text{old})$
$b(\text{new}) = b(\text{old})$
If weight changes
Yes
No
Stop

Example:



- NO of layer = 1; pattern = {P1->[1,1,1,1],p2->[-1,1,-1,1], p3->[1,1,1,-1], p4->[1,-1,-1,1]};target = {1,1,-1,-1}
- P1 =>t1=1; P2=>t2=1;P3=>t3=-1;P4=>t4=-1
- Weight=[0,0,0,0], transfer function=Symetrical hard limit
- Learning rate = alpha = 1

  1st iteration E-array, Er = [0,0,-2,-2] //// Sum of square Error = matrix(Er.)*matrix(Er).T
- A1 = f(w1*p1) = f[[0,0,0,0]*[1,1,1,1].T] = f(0)= 1
- E1 = t1-a1 =1 -1 = 0 ---> no change in weight
- A2 = f(w1*p2) = f[[0,0,0,0]*[-1,1,-1,1].T] = f(0)= 1
- E2 = t2- A2 =1 -1 = 0 ---> no change in weight
- A3 = f(w1*p3) = f[[0,0,0,0]*[1,1,1,-1].T] = f(0)= 1
- E3 = t3- A3 =-1 -1 = -2 ---> w2 = w1- (alpha . P3) = [0,0,0,0] – [1,1,1,-1] = [-1, -1 ,-1, 1]
- A4= f(w2*p4) = f[[-1, -1 ,-1, 1]*[1,-1,-1,1].T] = f(2)= 1
- E4 = t4- A4 =-1 -1 = -2 ---> w3 = w2- (alpha . P4) = [-1, -1 ,-1, 1] - [1,-1,-1,1] = [-2, 0 ,0, 0]

- NO of layer = 1; pattern = {P1->[1,1,1,1],p2->[-1,1,-1,1], p3->[1,1,1,-1], p4->[1,-1,-1,1]};target = {1,1,-1,-1}
- P1 =>t1=1; P2=>t2=1;P3=>t3=-1;P4=>t4=-1
- Weight=[-2, 0 ,0, 0], transfer function=Symetrical hard limit
- Learning rate = alpha = 1
- 2nd iteration E-array = [2,], w3 = [-2, 0 ,0, 0]
- A1 = f(w3*p1) = f[[-2, 0 ,0, 0]*[1,1,1,1].T] = f(-2)= -1
- E1 = t1-a1 =1 +1 = 2 ---> w4 = w3+ (alpha . P1) = [-1 , 1,1,1]

Adaline

- Adaptive linear neuron

1. The units with linear activation function

are called linear units.

2. A network with a single linear unit

is called an Adaline (adaptive linear neuron).

3. The Adaline nerwork maybe trained using delta rule.

The delta rule may afso be called as least mean square (LMS)

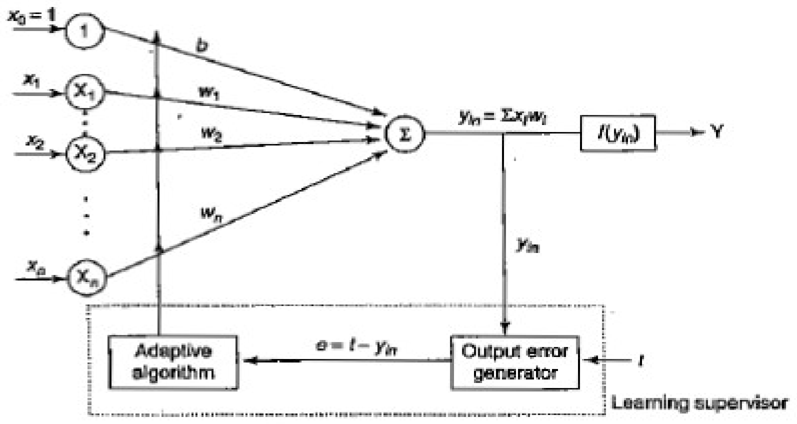
rule or Widrow-Hoff Rule.

4. This learning rule is found to minimize the mean-squared-error between the activation and the target

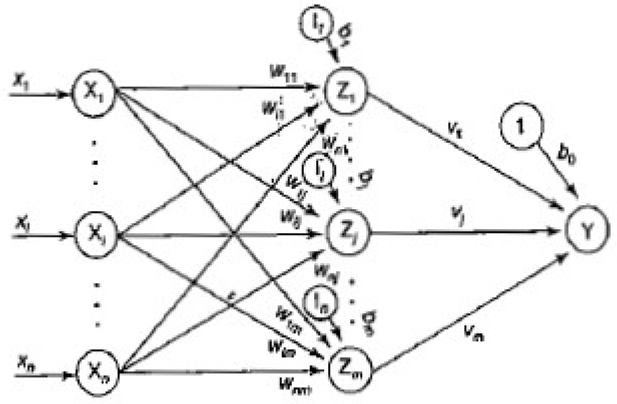
Value.

5. Weight modification:

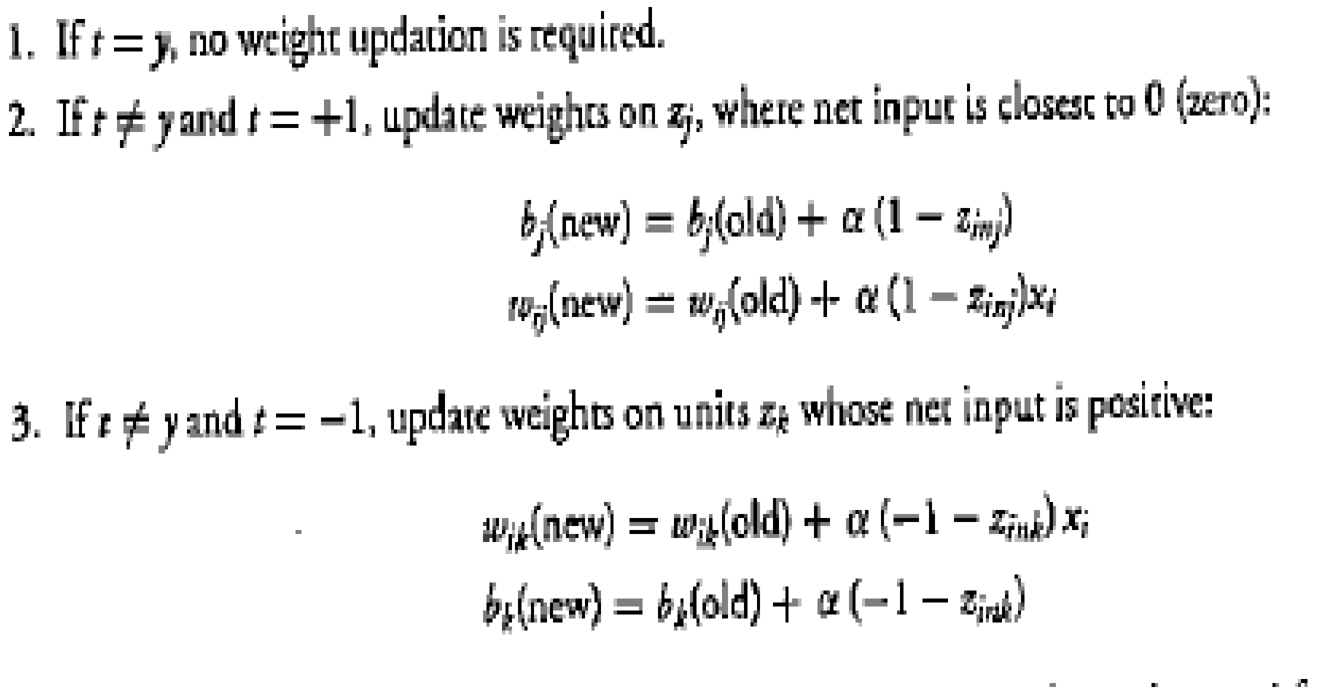
$$w_i(\text{new}) = w_i(\text{old}) + \alpha\,(t - y_{in})\,x_i$$
$$b(\text{new}) = b\,(\text{old}) + \alpha\,(t - y_{in})$$



Multiple Adaptive Linear Neurons



Weight and Bias update



1. If $t = y$, no weight updation is required.
2. If $t \neq y$ and $t = +1$, update weights on z_j, where net input is closest to 0 (zero):

$$b_j(\text{new}) = b_j(\text{old}) + \alpha\,(1 - z_{inj})$$

$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + \alpha\,(1 - z_{inj})x_i$$

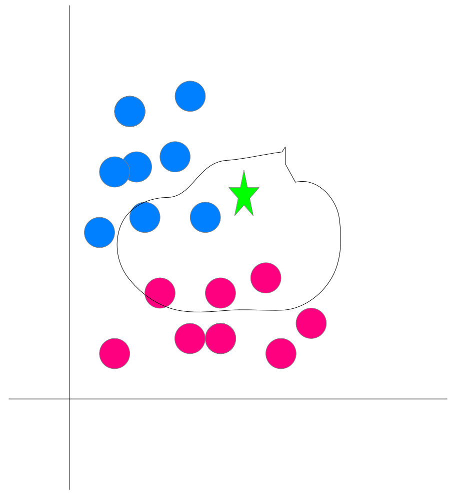
3. If $t \neq y$ and $t = -1$, update weights on units z_k whose net input is positive:

$$w_{ik}(\text{new}) = w_{ik}(\text{old}) + \alpha\,(-1 - z_{ink})\,x_i$$

$$b_k(\text{new}) = b_k(\text{old}) + \alpha\,(-1 - z_{ink})$$

K-NNB – Instance based learning

- Idea:
  - Similar examples have similar label.
  - Classify new examples like similar training examples.
- Algorithm:
  - Given some new example x for which we need to predict its class y
  - Find most similar training examples
  - Classify *x* “like” these most similar examples
- Questions:
  - How to determine similarity?
  - How many similar training examples to consider?
  - How to resolve inconsistencies among the training examples?



1-Nearest Neighbor

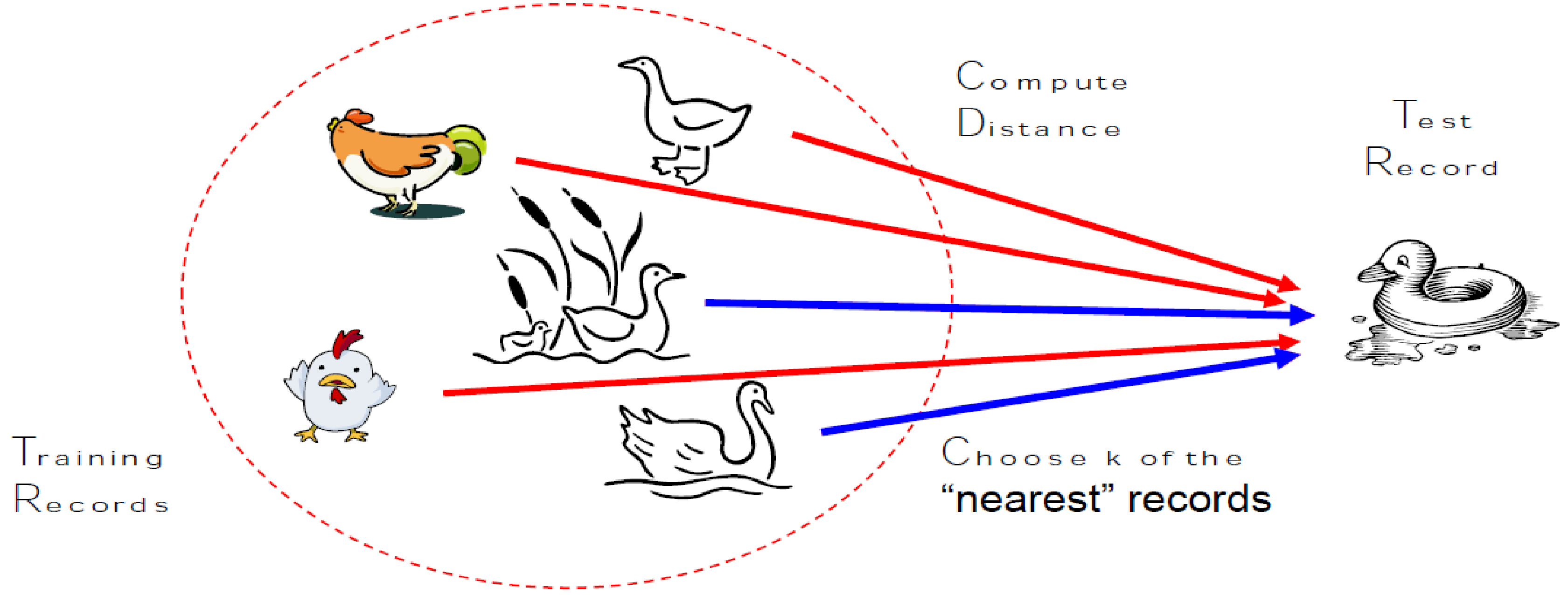
- One of the simplest of all machine learning classifiers
- Simple idea: label a new point the same as the closest known point
- A distance metric
- Euclidean
  - When different units are used for each dimension
    - → normalize each dimension by standard deviation
  - For discrete data, can use hamming distance
    - → D(x1,x2) =number of features on which x1 and x2 differ
  - Others (e.g., normal, cosine)

- How many nearby neighbors to look at?
  - One
- How to fit with the local points?
  - Just predict the same output as the nearest neighbor.

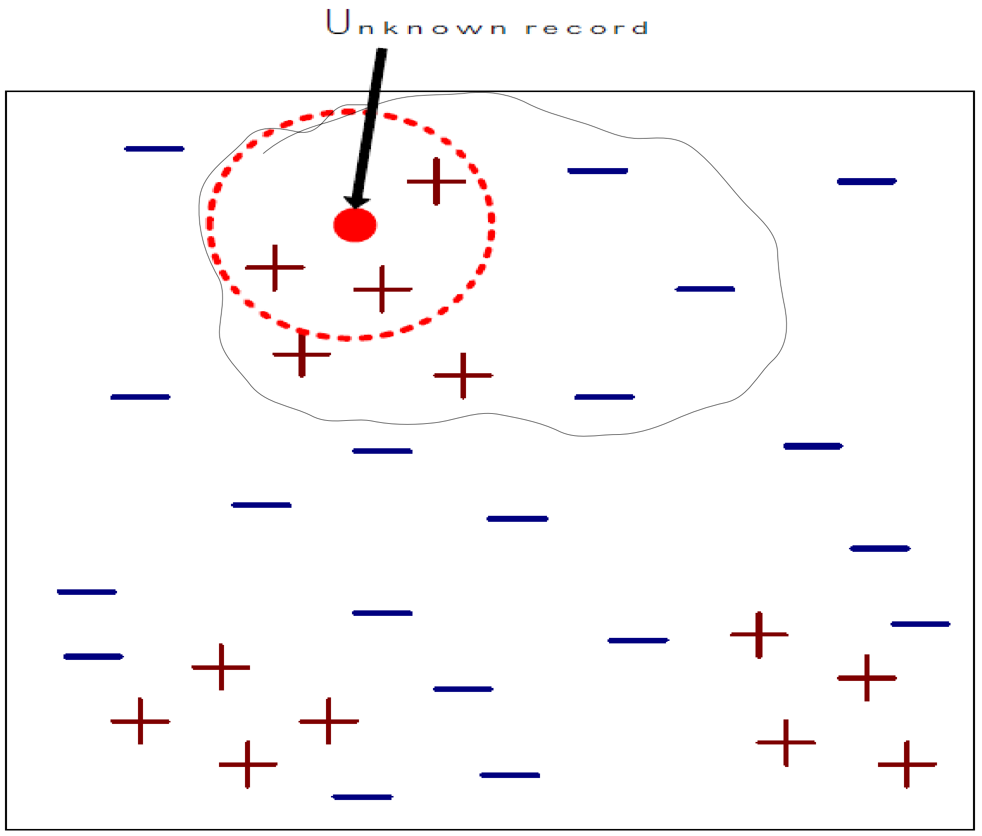
K-NNB Classifier

Basic idea:

– If it walks like a duck, quacks like a duck, then it's probably a duck



K-NNB Classifier



- Requires three things
  - The set of stored records
  - Distance Metric to compute distance between records
  - The value of *k*, the number of nearest neighbors to retrieve
- To classify an unknown record:
  - Compute distance to other training records
  - Identify *k* nearest neighbors
  - Use class labels of nearest neighbors to determine the class label of unknown record (e.g., by taking majority vote)

Measure Distance

Compute distance between two points:

- Euclidean distance

$$d(p,q) = \sqrt{\sum_i (p_i - q_i)^2}$$

- Manhatten distance

$$d(p,q) = \sum_i |p_i - q_i|$$

- q norm distance

$$d(p,q) = (\textstyle\sum_i |p_i - q_i|^q)^{1/q}$$

Determine the class

Determine the class from nearest neighbor list

– take the majority vote of class labels among the k-nearest neighbors

$$y' = \underset{v}{\operatorname{argmax}} \sum_{(x_i, y_i) \in D_z} I(v = y_i)$$

where D_z is the set of k closest training examples to z.

– Weigh the vote according to distance

$$y' = \underset{v}{\operatorname{argmax}} \sum_{(x_i, y_i) \in D_z} w_i \times I(v = y_i)$$

- weight factor, $w = 1/d^2$

K-NNB Algorithm

Let k be the number of nearest neighbors and D be the set of training examples.

```
1. for each test example z = (x′,y′) do
2.     Compute d(x′,x), the distance between z and every example, (x,y) ∈ D
3.     Select $D_z \subseteq D$, the set of k closest training examples to z.
4.     $y' = \underset{v}{\operatorname{argmax}} \sum_{(x_i,y_i) \in D_z} I(v = y_i)$
5. end for
```

k-NN classifiers are lazy learners

- It does not build models explicitly
- Unlike eager learners such as decision tree induction and rule-based systems
- Classifying unknown records are relatively expensive