CISC452/CMPE452/COGS 400
Perceptron

Farhana Zulkernine

McCulloch and Pitts's Neurons

- McCulloch and Pitts (1943) gave the first mathematical model of a single neuron.
- Early models of ANNs did not demonstrate learning.
- Weights were static and so were the connections.
- Had single layer that could not implement XOR.

Introduce Learning

- Hebb's learning rule (1949): For each input pattern presentation, increase connection weight between nodes i and j if both nodes are simultaneously ON or OFF.
- Activation of j always causes an activation of i where w_{ij} is the weight associated with connection from j to i and x_i and x_j are inputs to i and j respectively.
- The strength of connections between neurons eventually comes to represent the correlations between their outputs, e.g.,

$$\Delta w_{ij} = c \,.\, x_i x_j$$

where *c* is a some small constant.

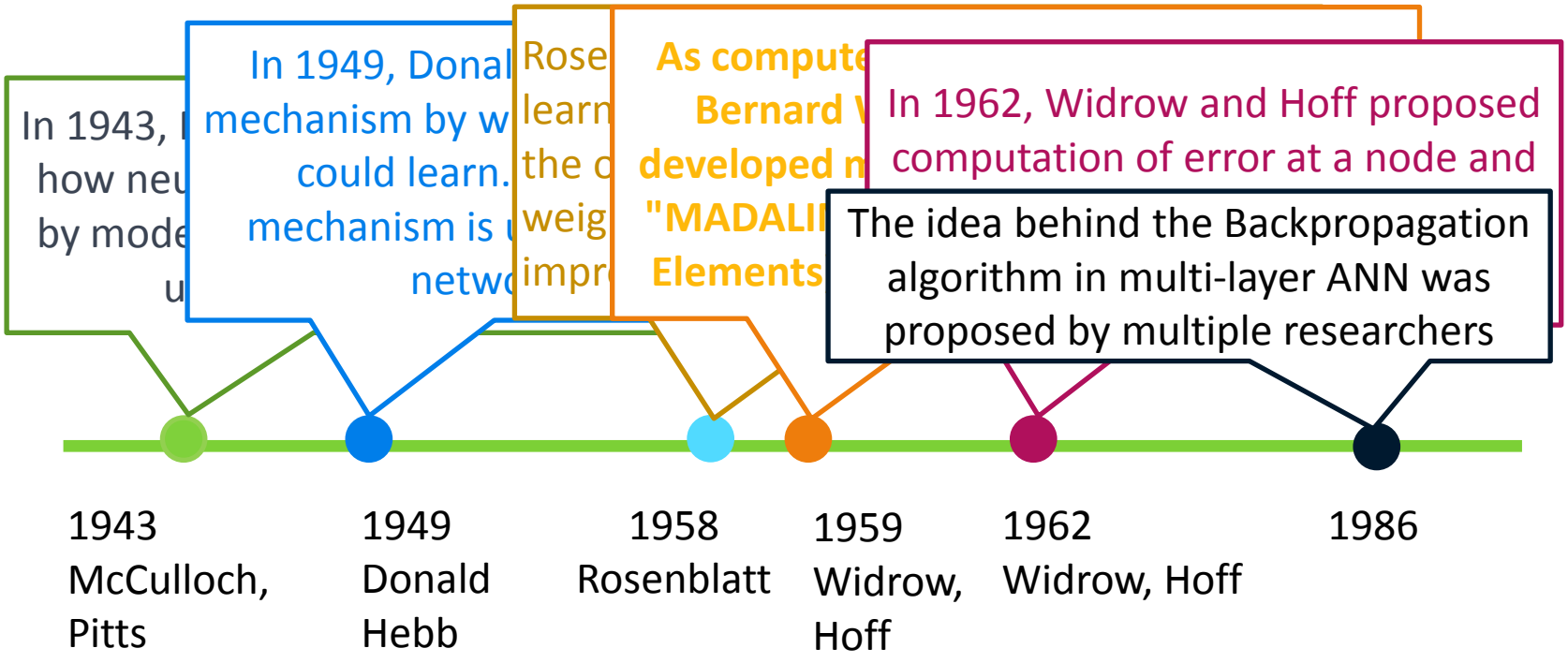
Perceptrons

- Rosenblatt's "perceptrons" (1958) used the following learning rule
  - If the output is unsatisfactory, modify each weight by a quantity that is likely to improve network performance.
- Also introduced the idea of supervised learning.
  - As if a supervisor gave feedback about whether the generated output was satisfactory or unsatisfactory.
  - Correct output was known and was used to modify weights to generate better output, and thereby, TRAIN the network.

More Learning Algorithms…

- Widrow and Hoff's learning rule (1960, 1962) was also based on *gradient descent*.
- Then error correction or back-propagation algorithms were proposed for training MULTI-LAYER networks.

History & Evolution of ANN Models

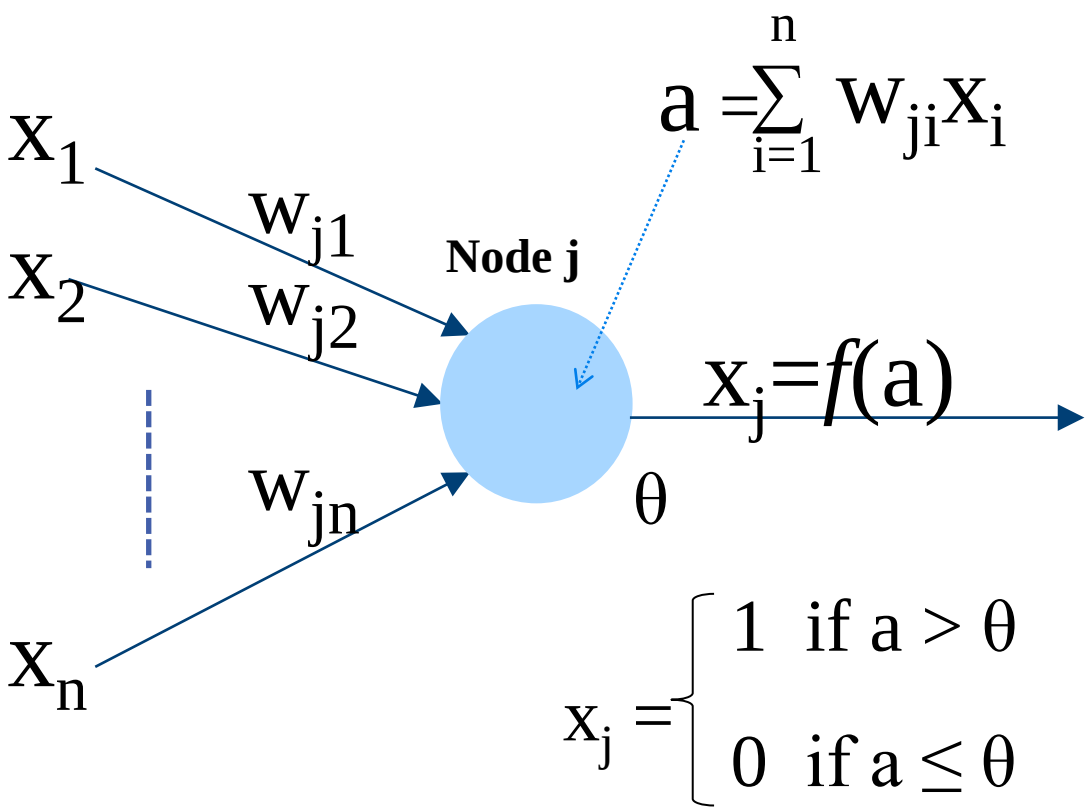


Perceptron

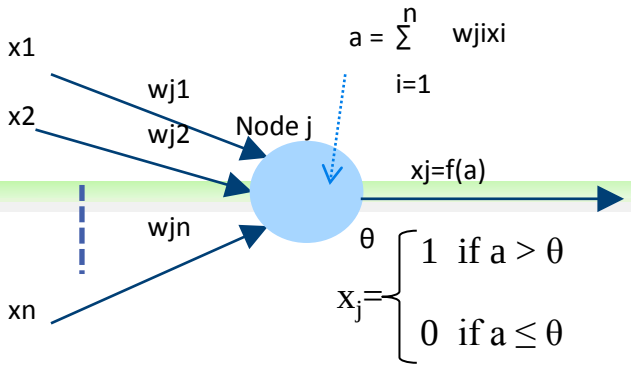
- Frank Rosenblatt proposed the perceptron learning rule in 1950's based on the idea that the operation of a neuron and its learning could be modeled mathematically, and used as a form of computation.

Perceptron

- A Perceptron Network is designed to learn the relationship between an input and output data.



Perceptron



- Input/Output examples:
  - $\{(x_1, d_1), \ldots, (x_p, d_p)\}$
  - Where (x_1, d_1) is an input/output pair of **vectors**
  - Vector $x_i = (x_{i1}, x_{i2}, \ldots, x_{in})$ has *n* input components
  - Vector $d_i = (d_{i1}, d_{i2}, \ldots, d_{im})$ has *m* output components
  - Weights are real values, $w_i \in \mathbb{R}$
  - x_j is the **output**
  - *f*() is the **output or transfer** function
  - θ is the threshold value

Perceptron for Prediction

- Train the perceptron using **input** and **desired output** vectors.
- Example: Given x_1, we like the perceptron to produce d_1 for output where d_1 is a known fact.

x = (10, 3150, 0.25)

Age of house in years

Square feet of house

Acreage of property

d= (1, 0)

Sale is over $300K (1=yes)

House will sell within 6 months (1=yes)

Perceptron for Prediction (cont…)

- We would like to give our ANN a newly listed house, and have it predict the outcome before the actual sale.
- For perceptrons the outputs d_i are binary:

  $d_i \in \{0,1\}^m$ or sometimes bipolar:

  $d \in \{-1,+1\}^m$
- The inputs could be in any reasonable range $\{0,1\}^n$, $\{-1,+1\}^n$, $[0,1]^n$, $\mathbb{R}^n$, etc.

Features and Functionality

- Two layer network (is it really a network?)
- Applies **feedforward processing** where data only flows towards output *y* which is computed from the input, along with the weights w_i.
- **Initially w_i are assigned random values** which results in poor initial performance.
- To improve performance **a procedure is needed to adjust the weight values**.

Learning

- A **Learning Rule** is a strategy by which example input/output pairs can be used to *incrementally change the weights in a way that gradually improves the performance* of the network.
- The perceptron learning rule involves the example input **x**, the computed output **y**, and the desired output **d** as y=*f*(a) where

$$a = \sum_{i=1}^{n} w_{ji} x_i$$

Perceptron Learning

- Two types of learning:
  1. **Simple Feedback learning**

     Uses the input value as feedback for changing weights.
  2. **Error Correction Learning**

     Uses an error measure to compute and adapt the weight vector.

Simple Feedback Learning

If y=1 and d=0: (because if input magnitude is high the error may be high and vice versa)

$$w_{ji} \leftarrow w_{ji} - \eta x_i$$

where (i = 1,..,n) and η is a small learning rate

If y=0 and d=1:

$$w_{ji} \leftarrow w_{ji} + \eta x_i$$

where (i = 1,..,n) and η is a small learning rate

Bias

- The action potential requires that

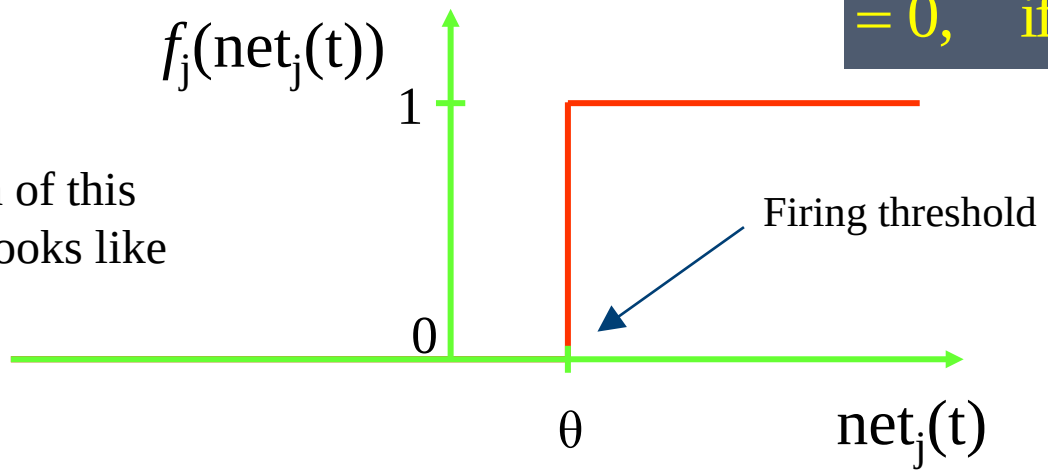
$f_j(net_j(t)) = 1$ if $(net_j(t) >= \theta)$ or $(net_j(t) - \theta) >= 0$

$f_j(net_j(t)) = 0$ if $(net_j(t) \leq \theta)$ or $(net_j(t) - \theta) \leq 0$

$$f_j(net_j(t)) = 1, \quad \text{if} \quad net_j(t) \geq \theta$$

$$= 0, \quad \text{if} \quad net_i(t) < \theta$$

The graph of this function looks like this.



Bias (cont…)

- So, considering $(net_j(t) - \theta) = 0$ is the line of separation of the output classes,

$$\sum_{i=1}^{n} x_i w_{ji} - \theta = 0$$

- Defining $x_0=1$, a fixed extra input called **bias**, and weight $w_0 = -\theta$ that is adjusted during learning, we can write

$$\sum_{i=0}^{n} x_i w_{ji} = 0$$

Bias (cont…)

- Therefore, $net_j(t)$ can be defined as: $\sum_{i=0}^{n} x_i w_{ji}$ which
  - Includes the bias and
  - Allows each processor (neuron node) to set its own threshold θ.
- So we are now able to adjust the weights and the threshold to train the network.

Linear Separability

- We have $a = w_0+w_1x_1+w_2x_2$ (ignoring j when thinking of only a single output node network) which is a linear equation considering $x_0=1$ and

$$0 = \sum_{i=0}^{n} x_i w_i$$

So, on the two sides of the above line,

$y = 1$ when $a > 0$ or $y=0$ when $a \leq 0$
(depends on your output function)

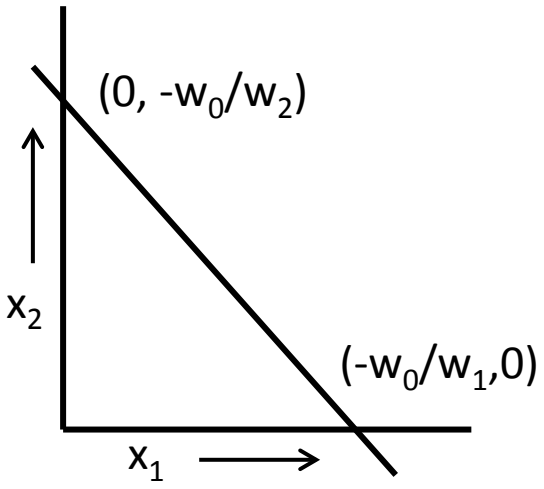
Linear Separability (cont…)

- To visualize a straight line, we determine the points where $w_0+w_1x_1+w_2x_2= 0$ crosses the coordinate axes.
- Representing it as y=mx+c, (and y => x_2)

$$x_2 = (-w_1/w_2)x_1 - w_0/w_2$$

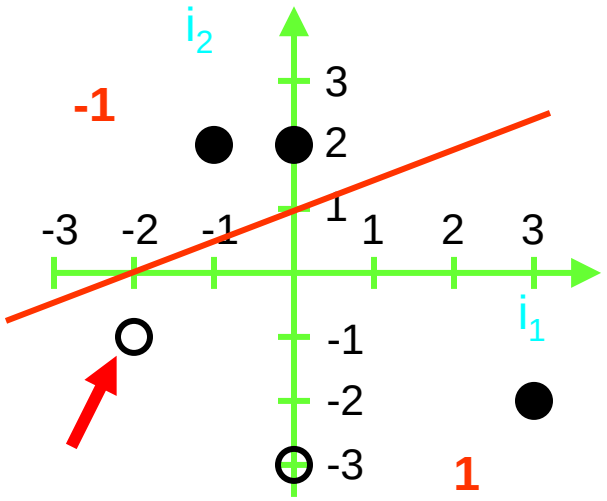
Slope Constant

- On x_1, $x_2 = 0$ and $x_1 = -w_0/w_1$
- On x_2, $x_1=0$ and $x_2 = -w_0/w_2$



Perceptron Learning Example

We would like our perceptron to correctly classify the five 2-dimensional data points below.



○ class -1
● class 1

Let the random initial weight vector $\mathbf{w^0} = (w_0, w_1, w_2) = (2, 1, -2)$.

So, the dividing line crosses the axes at

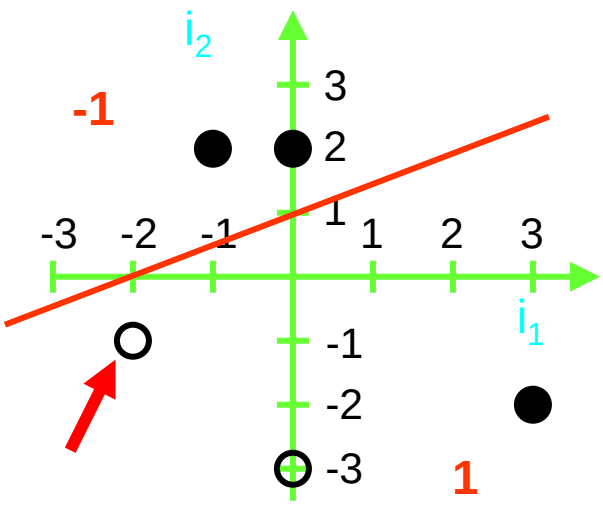
$[(-w_0/w_1, 0)$ and $(0, -w_0/w_2)]$

which are (-2, 0) and (0, 1).

Weight adaptation for learning:

$w_i \leftarrow w_i \pm cx_i$

Example(cont…)



○ class -1
● class 1

Let us pick the misclassified point $(x_1 , x_2) = (-2, -1)$ for learning.

Considering learning rate c=1:

$\mathbf{x} = (x_0, x_1, x_2) = (1, -2, -1)$
considering $x_0 = 1$

Since y=1, should be -1 (desired) decrease the weight

So, $\Delta w = -c\mathbf{x}$

$\Delta w = (-1) \cdot (1, -2, -1)$

$\Delta w = (-1, 2, 1)$

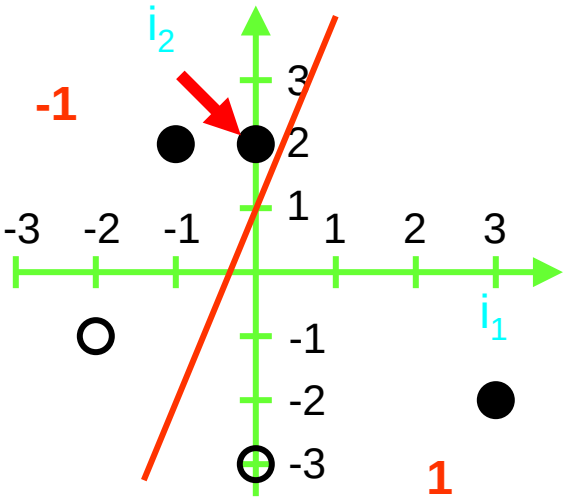
Example (cont…)

$\mathbf{w^1} = \mathbf{w^0} + \Delta w$

$\mathbf{w^1} = (2, 1, -2) + (-1, 2, 1) = (1, 3, -1)$

[at $(-w_0/w_1, 0)$ and $(0, -w_0/w_2)$]

The new dividing line crosses at (-1/3, 0) and (0, 1).



○ class -1
● class 1

Let us pick the next misclassified point (0, 2) for learning:

$\mathbf{x} = (1, 0, 2)$ (include bias $x_0 = 1$)

$\Delta w = (1).\ (1, 0, 2)$ (y=-1, should be 1)

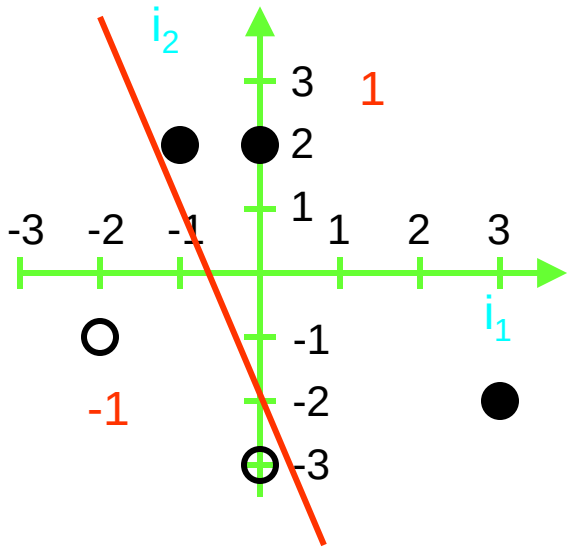
$\mathbf{w^2} = (1, 3, -1) + \Delta w = (2, 3, 1)$

Why do you think we pick the closest misclassified point?

Example (cont…)

$\mathbf{w^2} = (2, 3, 1)$ [at $(-w_0/w_1, 0)$ and $(0, -w_0/w_2)$]

Now the line crosses at (-2/3, 0) and (0, -2).



○ class -1
● class 1

With this weight vector, the perceptron achieves perfect classification!

The learning process terminates.

In most cases, many more iterations are necessary than in this example.

Justification

Let w be the weight vector (including the threshold) and i be the input vector (including the dummy input constant $i_0 = 1$) for a given iteration of the while loop. If i belongs to class for which the desired node output is -1 but $w \cdot i > 0$, then the weight vector needs to be modified to $w + \Delta w$ so that $(w + \Delta w) \cdot i < w \cdot i$; so that i would have a better chance of correct classification in the flowing iteration. Choose

$$\Delta w = -\eta i$$

where $\eta > 0$ is a constant. this will work since

$$(w + \Delta w) \cdot i = (w - \eta i) \cdot i = w \cdot i - \eta i \cdot i < w \cdot i$$

because $i \cdot i > 0$

Justification (cont…)

Similarly, if $w \cdot i < 0$ when the desired output value is 1, the weight vector needs to be modified so that $(w + \Delta w) \cdot i > w \cdot i$, which can be done by choosing Δw to be ηi for $\eta > 0$, since $(w + \eta i) \cdot i = w \cdot i + \eta i \cdot i > w \cdot i$

Some important questions;

1. How long should we execute this procedure, i.e., what is the termination criterion if the given samples are not linearly separable?
2. What is the appropriate choice for the learning rate η ?
3. How can the perceptron training algorithm be applied to problems in which the inputs are non-numeric values (e.g., values of the attributes "color" instead of numbers ?

Termination and Linear Separability

4. Is there a guarantee that the training algorithm will always succeed whenever the samples are linearly separable?
5. Are there useful modifications of the perceptron training algorithms that work reasonably well when samples are not linearly separable?

Termination Criteria

"Halt when the goal is achieved." The goal is the correct classification of all samples, assuming these are linearly separable. So allow to run until all sample are correctly classified.

Termination is assured if η is sufficiently small, and samples are linearly separable. (Perceptrons Convergence Theorem)

Termination Criteria (cont…)

The procedure will run indefinitely if the given samples are not linearly separate or if the choice of η is inappropriate. So, if the number of misclassifications has not changed in a large number of steps, the samples may not be linearly separable.

Else, if the problem is with the choice of η, then experimenting with a different choice of η may yield improvement.

Choice of the Learning Rate

If η is too large then the components of $\Delta w = \pm \eta x$ can have very large magnitudes, (assuming components of x are not infinitesimally small). Consequently, each weight update swings perceptron outputs completely in one direction, so that the perceptron now considered all samples to be in the same class as the most recent one. This effect is reversed when a new sample of the other class is presented.

If $\eta \approx 0$, the change in weights in each step is going to be infinitesimally small, assuming components of x are not very large in magnitude. So an extremely large number of such (infinitesimally small) changes to weights are needed.

Choice of Learning Rate

- A common choice is $\eta = 1$. To ensure that the sample x is correctly classified following the weight change (say from previous class of -1 to +1), we need $(w + \Delta w).x$ to be of the opposite sign of w.x or

  $|\Delta w \,.\, x| > |w \,.\, x|$

$\leftrightarrow \eta\,|x \,.\, x| > |w \,.\, x|$

$$\leftrightarrow \eta > \frac{|w \,.\, x|}{|x \,.\, x|}$$

Categorical Inputs

- What if input values are categorical?
  - For example, color ϵ {red, blue, green, yellow}
- The simplest alternative is:
  - Generate four new dimensions ("red“, "blue", "green" and "yellow“) and
  - Replace each original attribute-value pair by a binary vector with one component corresponding to each color.
- For example, if red=0, blue=0, green=1, yellow=0 then "green" can be represented as (0,0,1,0).
- In the general case, if an attribute can take one of *n* different values, then *n* new dimensions are obtained.

Perceptron Convergence Theorem

Theorem 1.1

Given training samples from two linearly separable classes, the perceptron training algorithm terminates after a finite number of steps, and correctly classifies all elements of the training set., irrespective of the initial random non-zero weight vector w_0.

See proof in the book (optional).

Not Linearly Separable – Algorithms

- Nothing useful can be assumed about the behavior of the Perceptron training algorithm when applied to classification problems in which the classes are not linearly separable.
- The "Pocket" and "Least Mean Squares" (LMS) algorithms attempt to achieve robust classification when the two classes are not linearly separable.

The Pocket Algorithm

- The pocket algorithm, is a useful modification of the perceptron training algorithm
  - Weight change mechanism is the same as that of the perceptron.
- In addition, the pocket algorithm identifies the weight vector with the **longest unchanged run** as the best solution among those examined so far.
  - Separately stores (in a "pocket") *the best solution* explored so far, as well as *the length of the run* associated with it.
  - The contents of the pocket are replaced whenever a new weight vector with a longer successful run is found.

Pocket Algorithm with Ratchet

- A lucky run of several successes may allow a poor solution to replace a better solution in the pocket.
- To avoid this, the Pocket algorithm with ratchet ensures that the pocket weights always
  - "ratchet up": a set of weights w^1 in the pocket is replaced by w^2 that has longer successful run **only after testing on all training samples** whether w^2 does correctly classify a greater number of samples than w^1.

Results

- The Pocket algorithm gives good results, although there is no guarantee of reaching the optimal weight vector in a reasonable number of iterations.

Learning – Error Correction

- Compute the error
- Adjust the weights to reduce error
- Weight adjustments are made based on the difference between desired and actual output

  i.e., (d-y)

Error Correction (d-y)

- Considering $x \in \{-1, +1\}^n$, for $y = 1$, $d = 0$,

  if $x_i > 0$, w_{ji} must be decreased

  if $x_i < 0$, w_{ji} must be increased and

- As $y \in \{0, 1\}$ or $y \in \{-1, +1\}$,

  $w_{ji} \leftarrow w_{ji} + (d-y)\, c x_i$

  where $(i = 1,..,n)$ and c is learning rate

- Range of the input and output values should be taken into account
  - Note that for both d and $y \in \{-1, +1\}$, $(d-y) \in \{-2, 0, +2\}$, so c should be ½ of what it would be for $y \in \{0, 1\}$