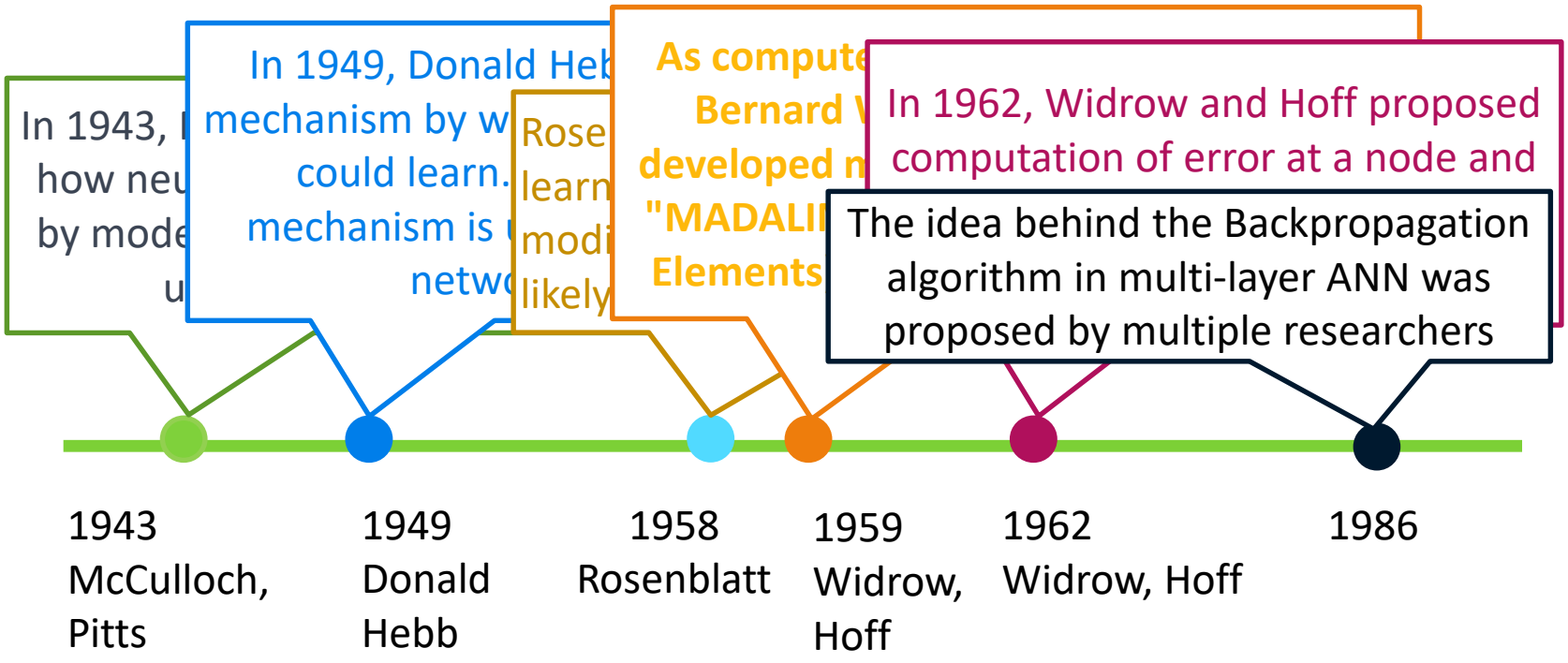
CISC452/CMPE452/COGS 400
Perceptron

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.

History & Evolution of ANN Models

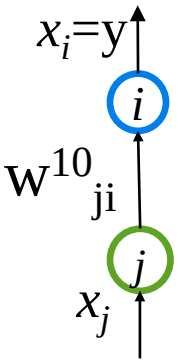


Introduce Learning

- 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_{ji} 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.,

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

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.
  - 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 back-propagation algorithms were proposed for training MULTI-LAYER networks.

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.
- Input/Desired-output examples – supervised learning: $\{(X_1, D_1), \ldots, (X_p, D_p)\}$

  Vector $X_i = (x_{i1}, x_{i2}, \ldots, x_{in})$, $D_j = (d_{j1}, d_{j2}, \ldots, d_{jm})$

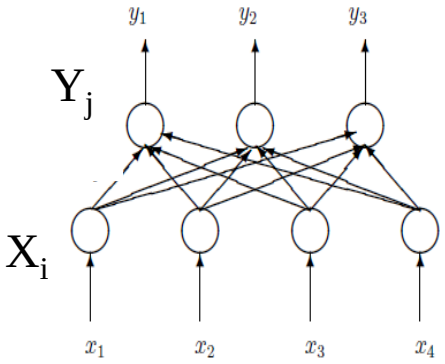
  $x_i \in \{-1,+1\}^n$ or $[0,1]^n$ or R^n

  $d \in \{-1,+1\}^m$ or $d_i \in \{0,1\}^m$

- $y_{jl} = f(\text{net}) = f(\sum_{k=1..n} x_{ik} w^{ij}_{kl})$

  if net >= θ, 0 otherwise

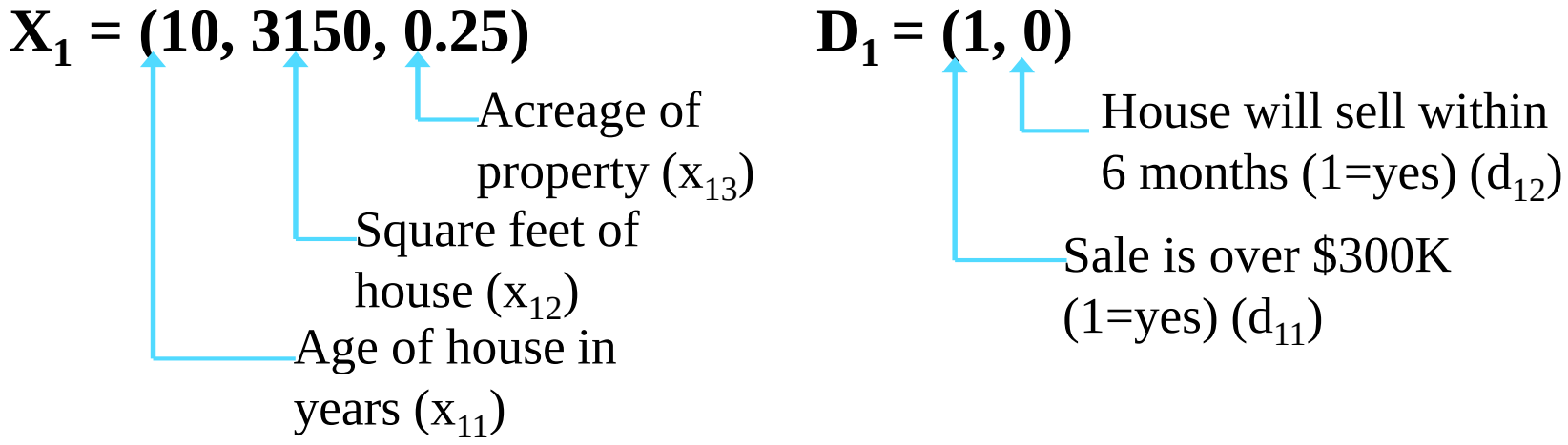
  $w_{ij} \in R$



- $(D_j - Y_j)$ is the error, θ is threshold or bias

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 known.



Features and Functionality

- Two layer network
- Applies **feedforward processing – all connections go to the next layer.**
- **Initially w_i are assigned random values** which results in poor initial performance (high error).
- To improve performance, network is **trained to adjust the weight values** → network **learns**.
  - A **Learning Rule** is a strategy by which input/output pairs are used to *incrementally change the weights to gradually improve the performance* of the network.

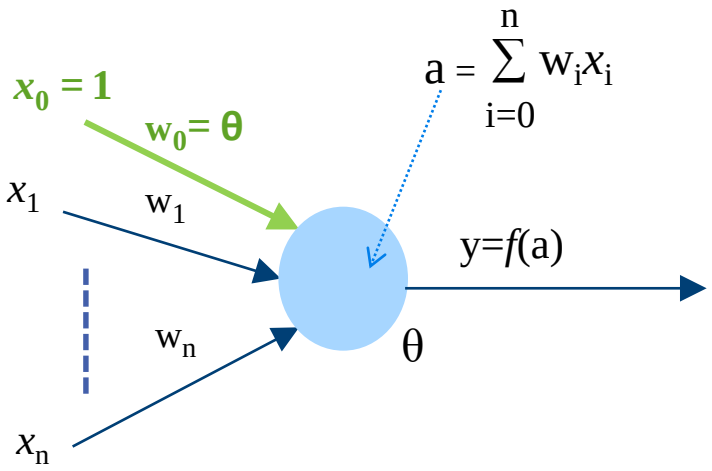
Adjusting both weight and bias

$$\sum_{i=1}^{n} x_i w_i - \theta = 0 \quad \Rightarrow \quad \sum_{i=1}^{n} x_i w_i - x_0 w_0 = 0, \quad \text{with } x_0 = 1,\ w_0 = -\theta$$

Now weight $w_0 = -\theta$ can be learned like the other weights

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

Allows each neuron to set its own threshold θ.



Plotting the line

- For 2-D space, a neuron will represent a straight line

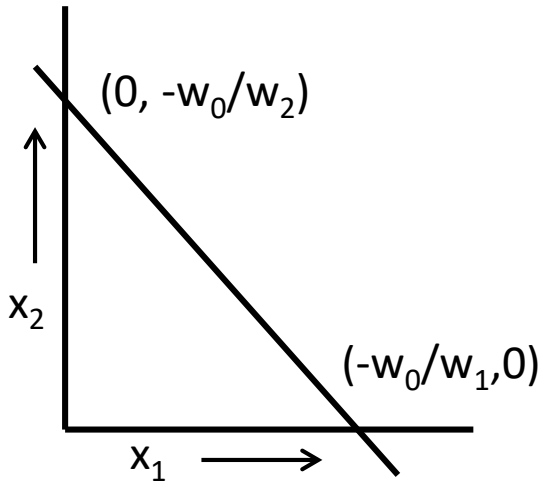
$$w_0 + w_1 x_1 + w_2 x_2 = 0$$

- Representing it as $y = mx + c$, (and $y = x_2$)

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

Slope Intercept

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



Perceptron Learning

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

     Uses the correct/incorrect feedback and info about (y>=d) or (y<d) to change weights.
  2. **Error Correction Learning**

     Uses an error measure to adapt the weight vector.

Simple Feedback Learning

If y=1 and d=0 (y > d):

$$w_{ji} \leftarrow w_{ji} - cx_i$$

Use input value in calculation because if input value is high, error will be high and vice versa)

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

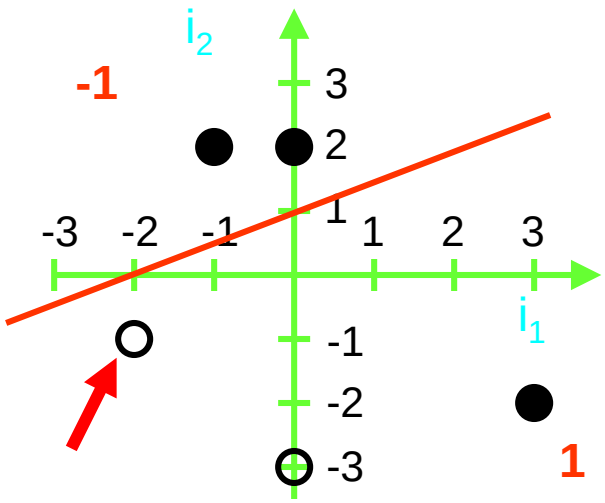
If y=0 and d=1 (y < d):

$$w_{ji} \leftarrow w_{ji} + cx_i$$

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

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 $(0, -w_0/w_2)]$

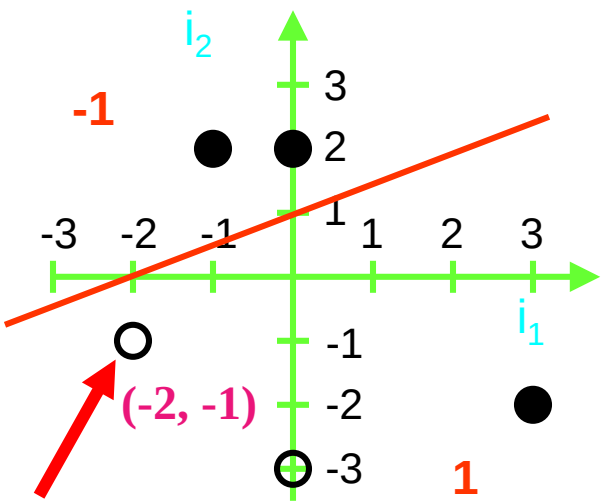
which are (-2, 0) and (0, 1).

Weight adaptation for learning:

$w_i \leftarrow w_i \pm cx_i$

Example(cont…)

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



○ class -1
● class 1

Considering **learning rate c=1,** $x_0 = 1$

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

Since $y=1$, $d= -1$,

decrease the weight $\Delta w = -c\mathbf{x}$

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

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

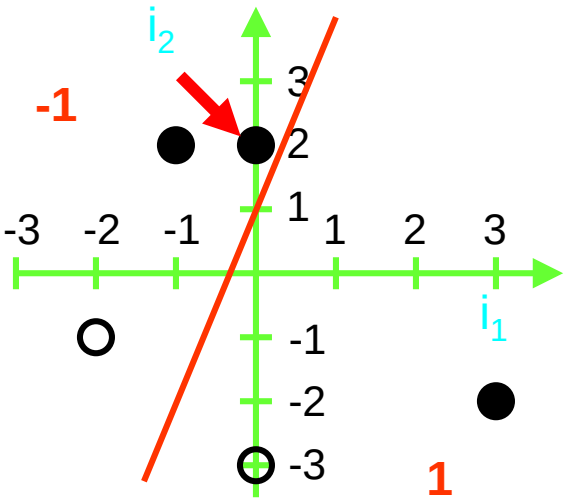
$\mathbf{w^1} = \mathbf{w^0} + \Delta w$

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

Example (cont…)

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

The new dividing line intersects the axes 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 $x_0 = 1$)

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

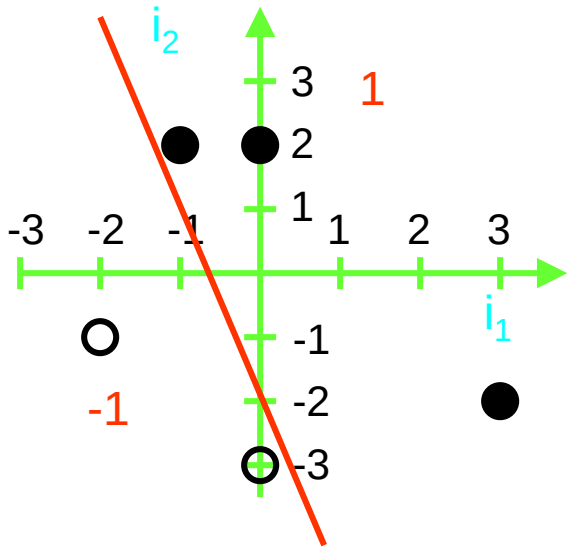
$\mathbf{w^2} = (1, 3, -1) + \Delta w = (2, 3, 1)$

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.

How do you know the algorithm works?

- Activation $a = w.x$
- If $y=1$ and $d=0$, then $(w - \Delta w).x < w.x$
- Considering learning rate $c=1$, $\Delta w = cx = x$
- Therefore, $(w - \Delta w).x \Rightarrow w.x - x.x$
- But $x.x > 0$ and so, $(w.x - x.x)$ must be $< w.x$ which implies that the weight adjustment will eventually lead to a weight value that will correctly classify the input data.
- Same justification can be used for $y=0$ and $d=1$.

Perceptron Convergence Theorem

- It can be guaranteed that the Perceptron training algorithm will classify all the data correctly **when they are linearly separable and *c* is sufficiently small**.
- ***See proof in the book or the slides posted on OnQ.

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.

Choice of c

- If *c* is too small, the algorithm will make very small changes to the weights each time → very long training time
- If *c* is too big, the weight changes will be too much and the data that were previously correctly classified may be misclassified again.
  - The separator line will fluctuate its slope too much and never reach the correct slope.

Choice of c (cont…)

- A common choice is $c = 1$.
- To ensure that the sample x is correctly classified following the weight change
- $(w \pm \Delta w).x$ must be of the opposite sign of $w.x$
  $\Rightarrow |\Delta w \cdot x| > |w \cdot x|$

$\leftrightarrow c\,|x \cdot x| > |w \cdot x|$ since $\Delta w = cx$

$$\leftrightarrow c > \frac{|w \cdot x|}{|x \cdot x|}$$

Terminating Condition

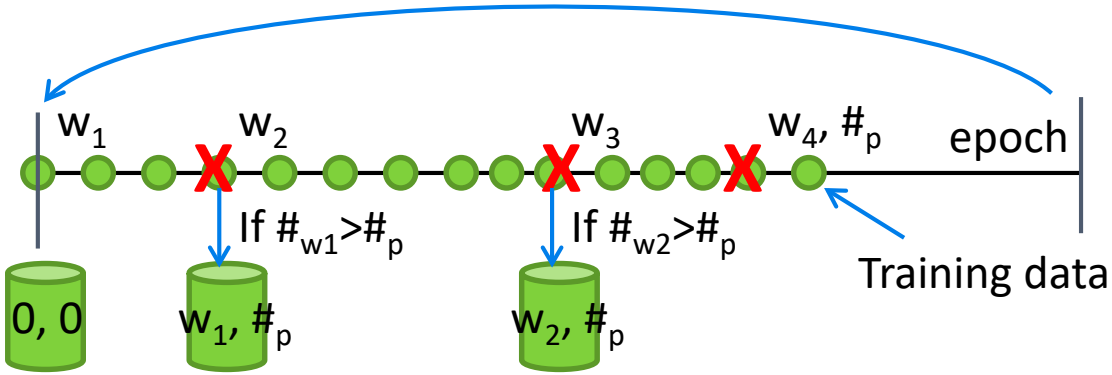
- Until all data are correctly classified
  - Problem ***
    - Data may not be linearly separable → results in infinite loop
      - Add a maximum number of iteration
    - The value of *c* may be too high and the weight vector fluctuates too much.
      - Try using a lower value.
- Until a fixed number of iteration has been run
  - One iteration = Running with $X_1..X_p$ once
- Until acceptable error level is reached
  - Error = (misclassified data points / total data points) <= threshold
  - When data is not linearly separable – How do we know that?
- Combine multiple of the above conditions

Not Linearly Separable – Algorithms

- 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.
  - Identifies the weight vector ($w_1,\ldots,w_n$) with the **longest unchanged run** (# of points correctly classified: $\#_p$) as the best solution so far.
  - Stores *the best weight vector* in a "pocket" as well as *the best length of the run* associated with it.
  - Pocket contents are replaced with a new weight vector when 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"
  - 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.
  - Expensive computation.

Results

- The Pocket algorithm gives good results, although there is no guarantee of reaching the optimal weight vector in a reasonable number of iterations due to distribution of data points.
- The best solution may never get saved in the pocket and hence ratchet version in that case will also not be able to find that solution.
  - If the best solution ever gets selected to be put into the Pocket then ratchet will test it for all solutions and keep it in the pocket.

Learning by Error Correction

- Compute the error (d-y), difference between desired and actual output
- Adjust the weights to reduce error

$$w^{t+1}_{ji} = w^{t}_{ji} + \Delta w = w^{t}_{ji} + (d-y)*cx_i$$

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 ϵ {-1, +1}, (d-y) ϵ {-2, 0, +2}
  - c * (d-y) will be high.
  - so *c* should be ½ of what it would be for y ϵ {0, 1}

Categorical Inputs

- What if input values are categorical?
  - E.g. color ϵ {red, blue, green, yellow}
- The simplest alternative:
  - Generate four new dimensions: red, blue, green and yellow
  - Recode categorical value as a binary vector [red, blue, green, yellow] = $\{0,1\}^4$.
    - For example, if red=0, blue=0, green=1, yellow=0 then "green" can be represented as (0,0,1,0).
- You can use 2 digits to represent the 4 possible values [(0,0),(0,1),(1,0),(1,1)] – Problems
  - Mapping of layers get complicated
  - Results in longer training time and need more neurons in hidden layers.
- In the general case, if an attribute (e.g. color) can take one of *n* different values, then *n* new dimensions are obtained.

Perceptron and Linear Separability

- In general, networks of perceptron-like processors can solve most non-linearly separable tasks by using more than one layer of processors.
- Rosenblatt (and others) realized this in the 1960's, but did not have a learning rule that would work effectively with more than one level (it wasn't invented until the mid 1970's).