Introduction to Artificial Intelligence

LECTURE 6:

K-NN Classification & Neural Networks

Overview

- K-NN Classification
- Neural Networks

K-NN Classification

- Simple classification algorithm
- Idea:
  - Look around you to see how your neighbors classify data
  - Classify a new data-point according to a majority vote of your k nearest neighbors

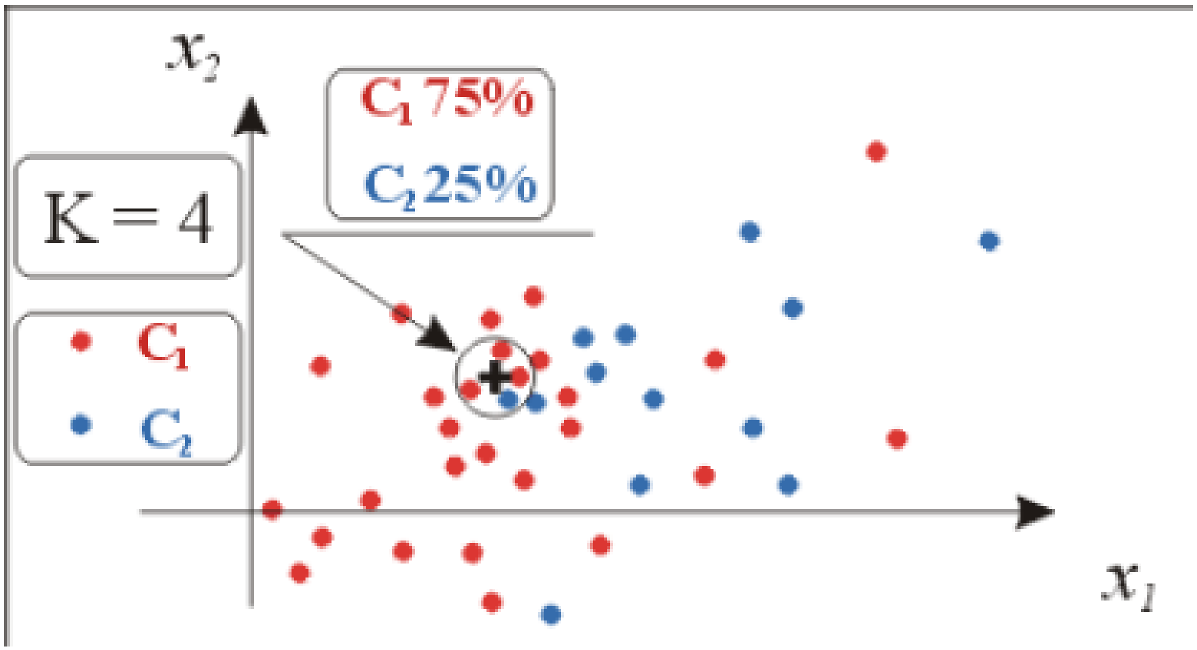
Distance Metric

- How do we measure what it means to be a neighbor (what is “close”)?
- Appropriate distance metric depends on the problem
- Examples:
  - x discrete (e.g., strings): Hamming distance
    $d(x_1, x_2)$ = number of features on which x_1 and x_2 differ
  - x continuous (e.g., vectors over reals): Euclidean distance
    $d(x_1, x_2) = || x_1 - x_2 ||$ = square root of sum of squared differences between corresponding elements of data vectors

Example

Input Data: 2-D points (x_1, x_2)

Two classes: C_1 and C_2. New Data Point **+**



K = 4: Look at 4 nearest neighbors of **+**

3 are in C_1, so classify **+** as in C_1

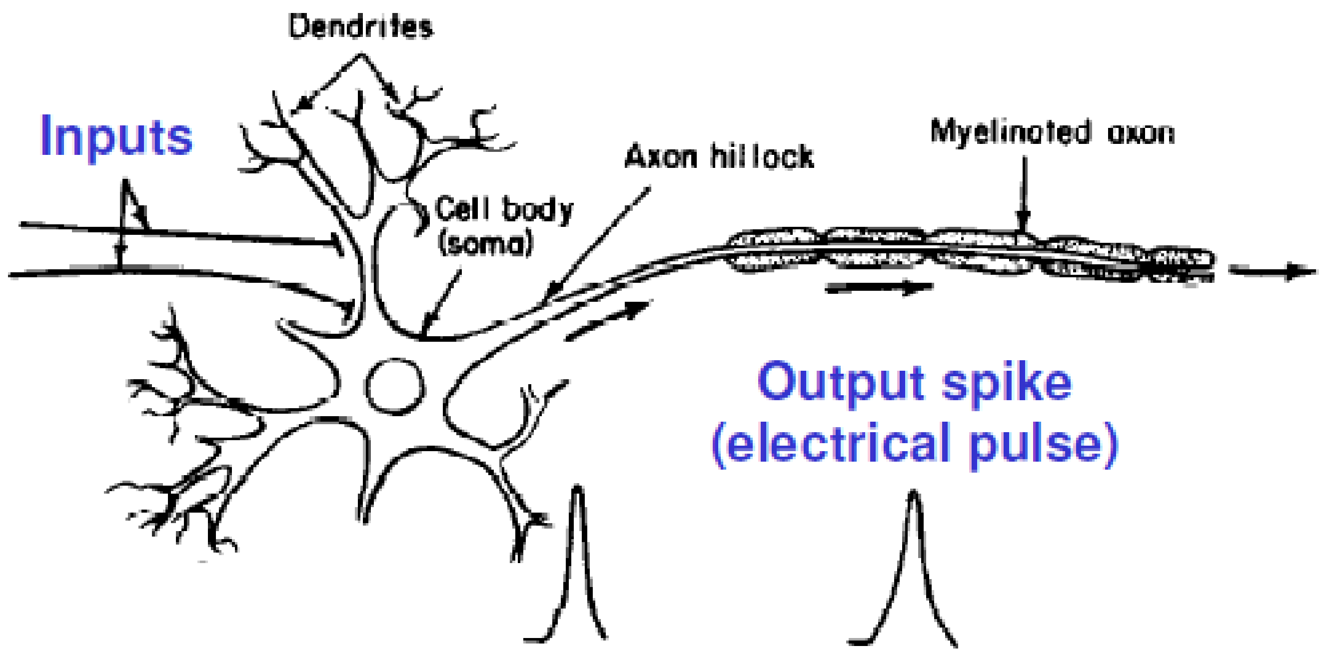
Practical 3

- Image-based Object Classification
  - Training Dataset


  - Testing Dataset


Emulating the Brain

10^{11} neurons of more than 20 types, 10^{14} synapses, 1 ms-10 ms cycle time



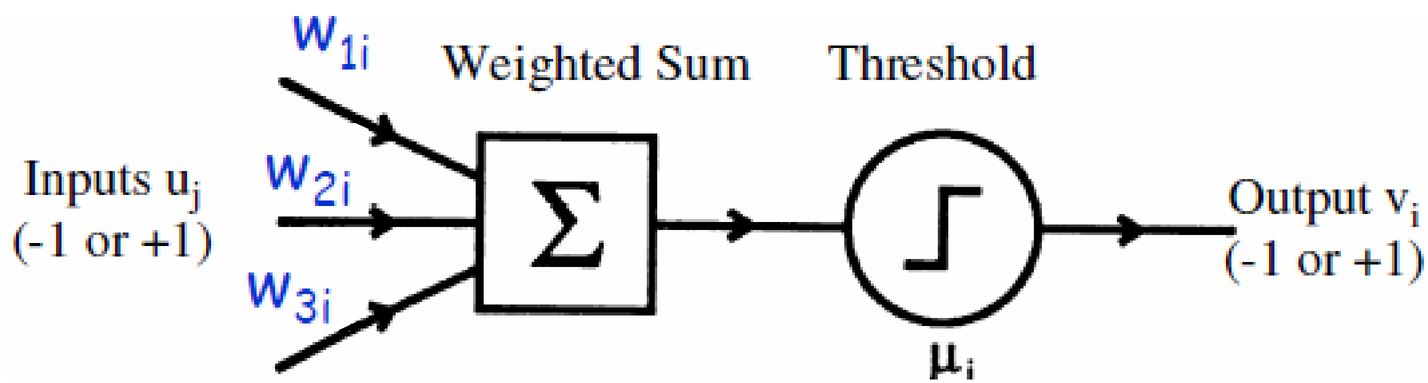
Output spike roughly dependent on whether sum of all inputs reaches a threshold

Neurons as "Threshold Units"

- Artificial neuron
  - Binary inputs (-1 or 1) and 1 output (-1 or 1)
  - Synaptic weights w_{ji}
  - Threshold μ_i

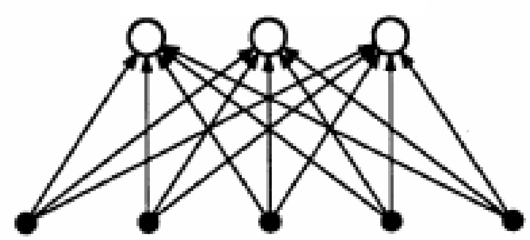
$$v_i = \Theta\left(\sum_j w_{ji} u_j - \mu_i\right)$$

$$\Theta(x) = 1 \text{ if } x > 0 \text{ and } -1 \text{ if } x \le 0$$



"Perceptrons" for Classification

- Uses artificial neurons ("units") with binary inputs and outputs



- Weighted sum forms a linear hyperplane

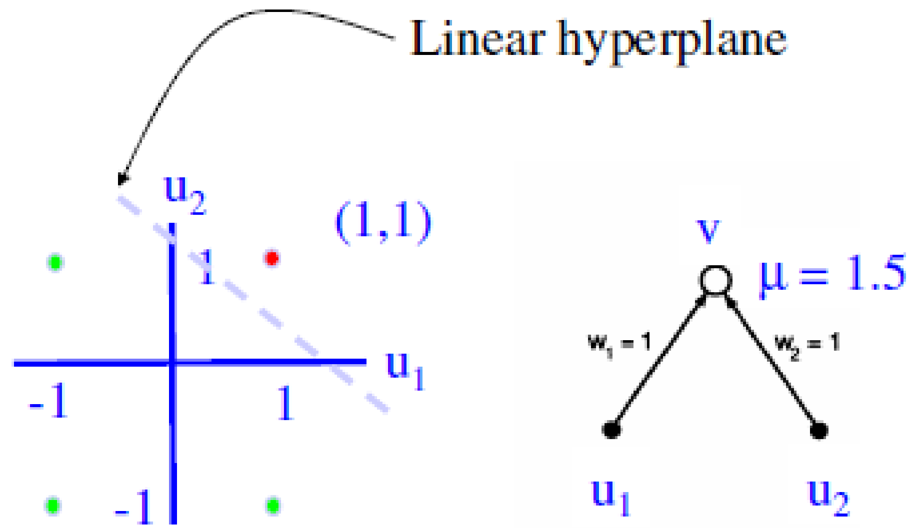
$$\sum_j w_{ji} u_j - \mu_i = 0$$

- Everything on one side of this hyperplane is in class 1 (output = +1) and everything on other side is in class 2 (output = -1)
- Any function that is linearly separable can be computed by a perceptron

Linear Separability

- Illustration: AND is linearly separable

u_1	u_2	AND
-1	-1	-1
1	-1	-1
-1	1	-1
1	1	1



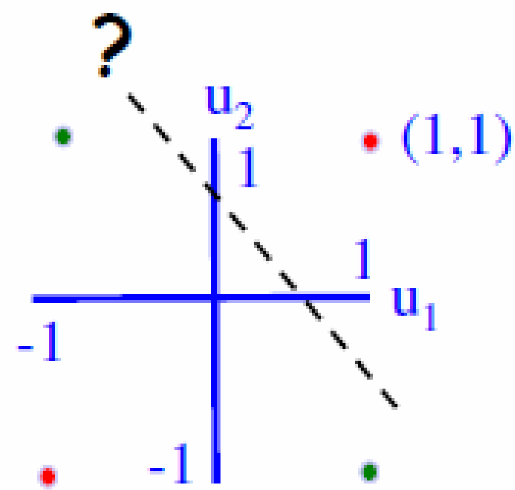
$v = 1$ iff $u_1 + u_2 - 1.5 > 0$

- Similarly for OR, NOT

Linear Separability?

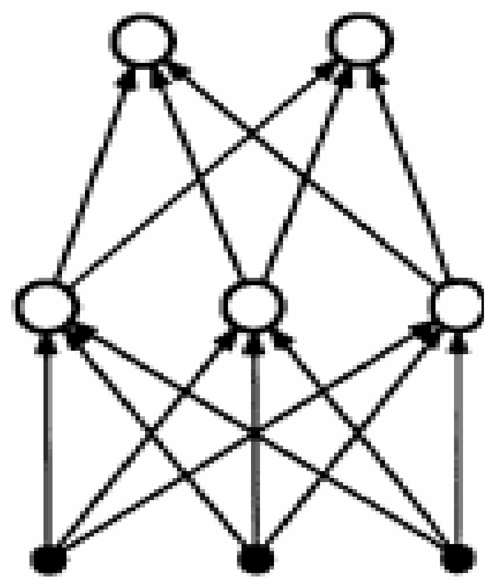
- Illustration: XOR function
  - Can a straight line separate the +1 outputs from the -1 outputs?

u_1	u_2	XOR
-1	-1	1
1	-1	-1
-1	1	-1
1	1	1



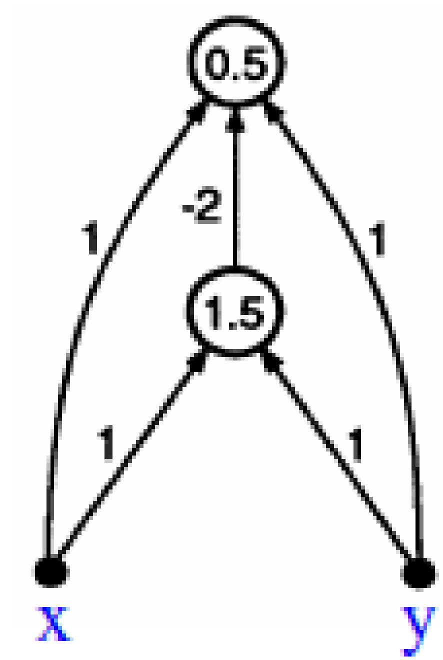
Linear Inseparability

- Single-layer perceptron with threshold units fails if classification task is not linearly separable
  - Example: XOR where no single line can separate the “yes” (+1) outputs from the “no” (-1) outputs!
- Dealing with linear inseparability: multilayer perceptrons



Multilayer Perceptrons

- Removes limitations of single-layer networks
- Example: Two-layer perceptron that computes XOR



Output is +1 if and only if $x + y - 2\Theta(x + y - 1.5) - 0.5 > 0$

Weighting

- How do we learn the appropriate weights given only examples of (input, output)?
- Idea: Change the weights to decrease the error in output

Perceptron Learning Rule

Given input pair (u, vd) where vd ϵ {+1,-1} is the desired output, adjust w and μ as follows:

1. Calculate current output v of neuron

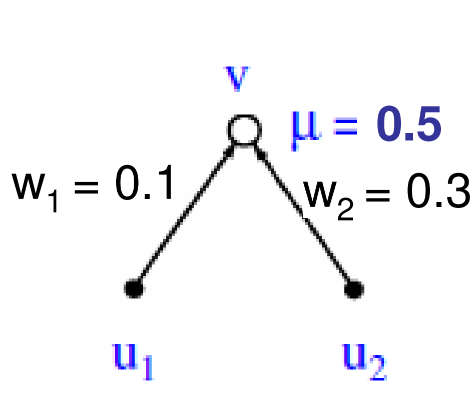
$$v = \Theta\left(\sum_j w_j u_j - \mu\right) = \Theta(\mathbf{w}^T \mathbf{u} - \mu)$$

2. Compute **error signal** e = (vd – v)
3. Multiply the error signal by the learning rate ϵ (small positive number). Add this correction to any weight for which the input is non-zero:

   w ← w + ϵ (vd – v) u

4. If the network outputs the correct result for all of the training set examples, conclude.

Exercise: learning the OR function



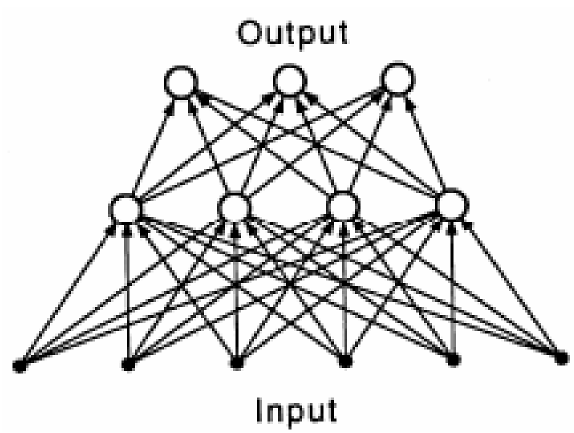
- Apply the algorithm for learning the OR function. Use learning rate = 0.2. Weights have initial values $w_1 = 0.1$, $w_2 = 0.3$.

Exercise: learning the OR function

u_1	u_2	w_1	w_2	v_d	v	e	w_1	w_2
0	0	0.1	0.3	0	(-0.5) 0	0	0.1	0.3
0	1	0.1	0.3	1	(-0.2) 0	1	0.1	0.5
1	0	0.1	0.5	1	(-0.4) 0	1	0.3	0.5
1	1	0.3	0.5	1	(0.3) 1	0	0.3	0.5
0	0	0.3	0.5	0	(-0.5) 0	0	0.3	0.5
0	1	0.3	0.5	1	(0) 0	1	0.3	0.7
1	0	0.3	0.7	1	(-0.2) 0	1	0.5	0.7
1	1	0.5	0.7	1	(0.7) 1	0	0.5	0.7
0	0	0.5	0.7	0	(-0.5) 0	0	0.5	0.7
0	1	0.5	0.7	1	(0.2) 1	0	0.5	0.7
1	0	0.5	0.7	1	(0) 0	1	0.7	0.7
1	1	0.7	0.7	1	(0.9) 1	0	0.7	0.7
0	0	0.7	0.7	0	(-0.5) 0	0	0.7	0.7
0	1	0.7	0.7	1	(0.2) 1	0	0.7	0.7
1	0	0.7	0.7	1	(0.2) 1	0	0.7	0.7
1	1	0.7	0.7	1	(0.9) 1	0	0.7	0.7

Function Approximation

- We want networks that can learn a function
  - Network maps real-valued inputs to real-valued output
  - Idea: Given data, minimize errors between network's output and desired output by changing weights

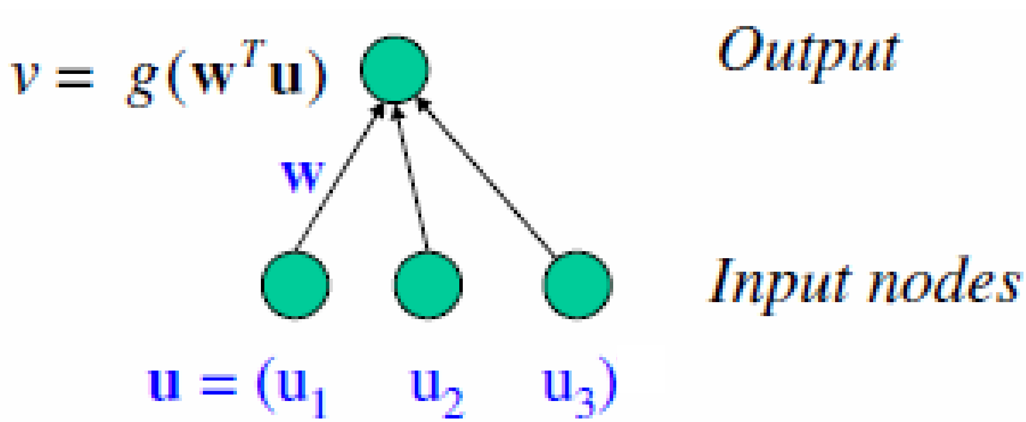


Continuous output values: binary threshold units cannot be used anymore

To minimize error, a *differentiable* output function is desirable

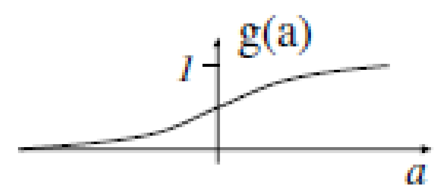
Sigmoidal Networks

- Most common activation function: Sigmoid function
  - Non-linear “squashing” function
  - Squashes input to be between 0 and 1. The parameter β controls the slope.



Sigmoid function:

$$g(a) = \frac{1}{1 + e^{-\beta a}}$$



Gradient Descent Learning

- Given training examples (u_m, vd_m) (m = 1,…, N), define an error (also cost/energy) function

$$\textit{Error}(\mathbf{w}) = \frac{1}{2}\sum_m \mathbf{E_m}^2$$

$$\textit{Error}(\mathbf{w}) = \frac{1}{2}\sum_m (vd_m - v_m)^2$$

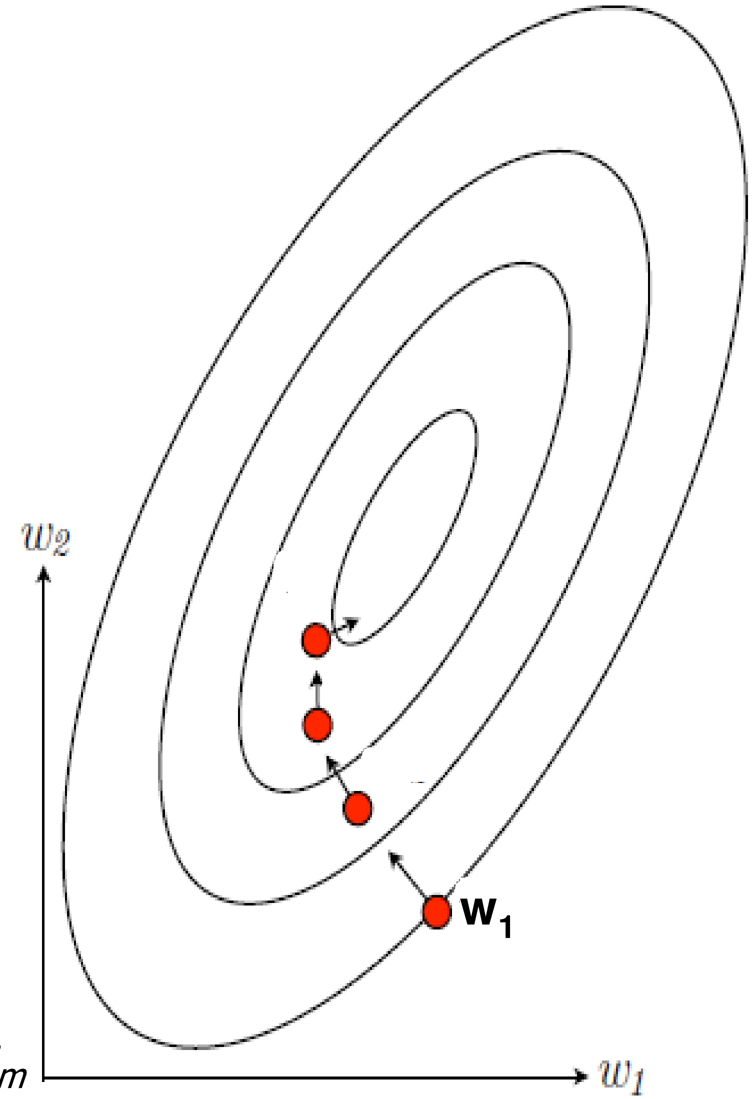
where $v_m = g(\mathbf{w}^T u_m)$

- Change w so that E(w) is minimized: Gradient Descent

$$\mathbf{w} \leftarrow \mathbf{w} - \varepsilon \frac{dE}{d\mathbf{w}}$$

$$\frac{dE}{d\mathbf{w}} = -\sum_m (vd_m - v_m)\frac{dv_m}{d\mathbf{w}} =$$

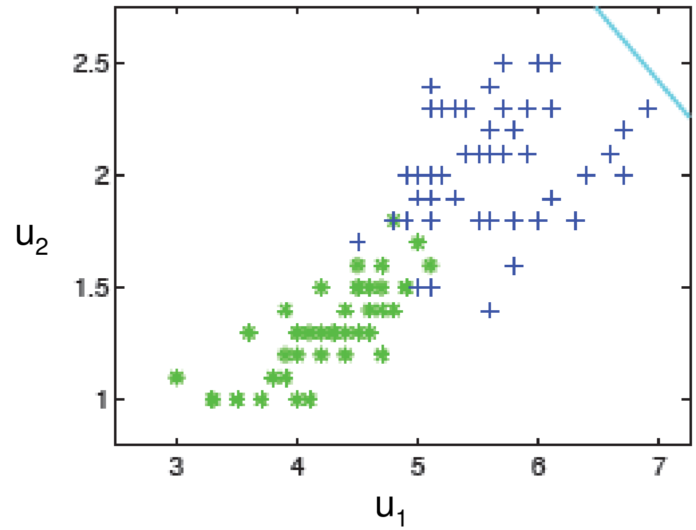
$$-\sum_m (vd_m - v_m)\, g'(\mathbf{w}^T u_m)\, u_m$$

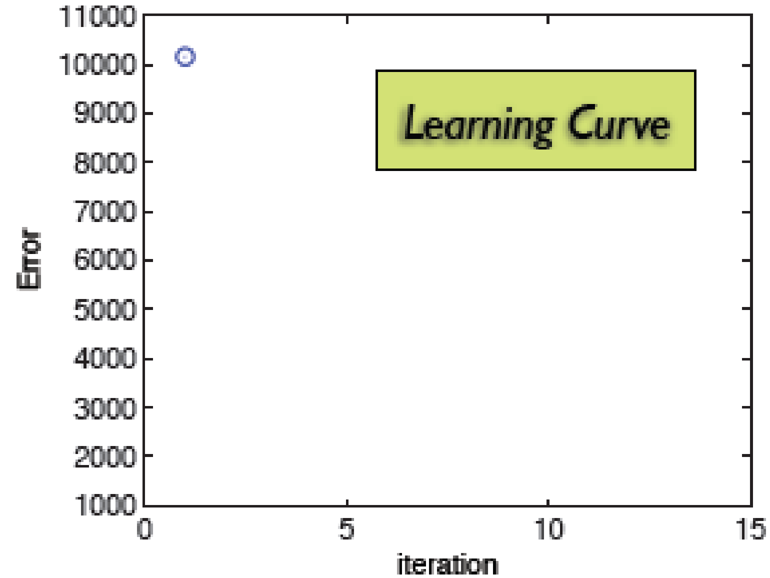


Learning the Decision Boundary

- Each iteration updates the gradient

- Value of ϵ
  - Small: O.1/N
  - Too large: learning diverges
  - Too small: slow convergence

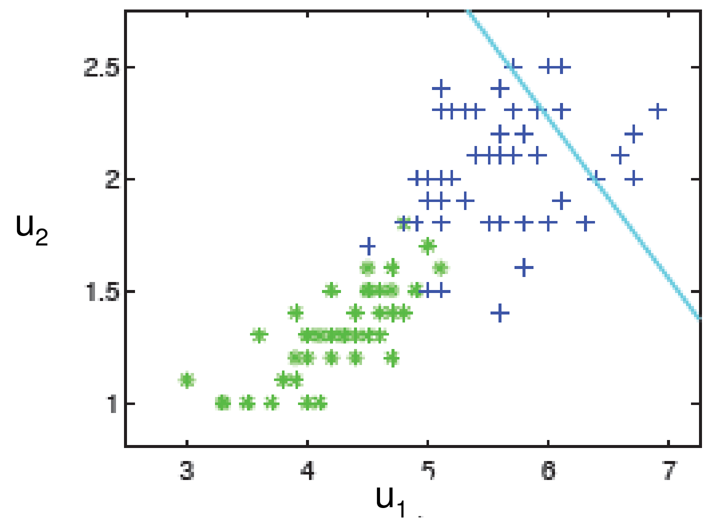


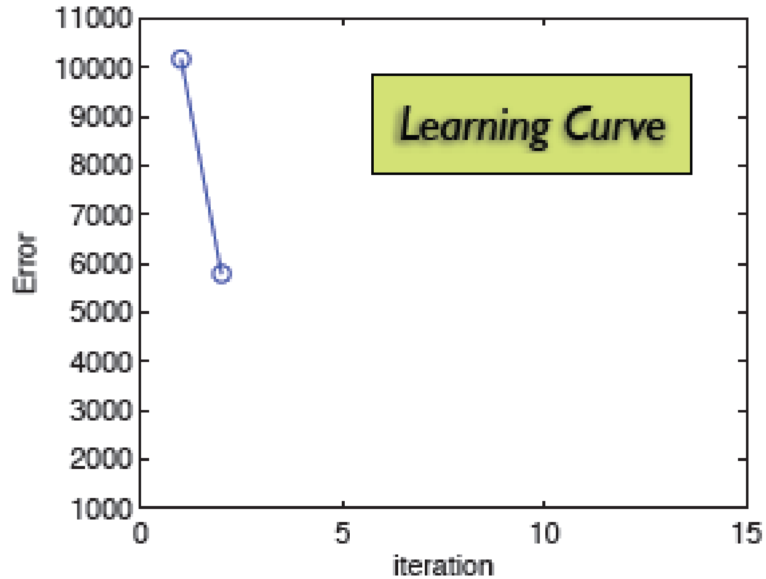


Learning the Decision Boundary

- Each iteration updates the gradient

- Value of ϵ
  - Small: O.1/N
  - Too large: learning diverges
  - Too small: slow convergence

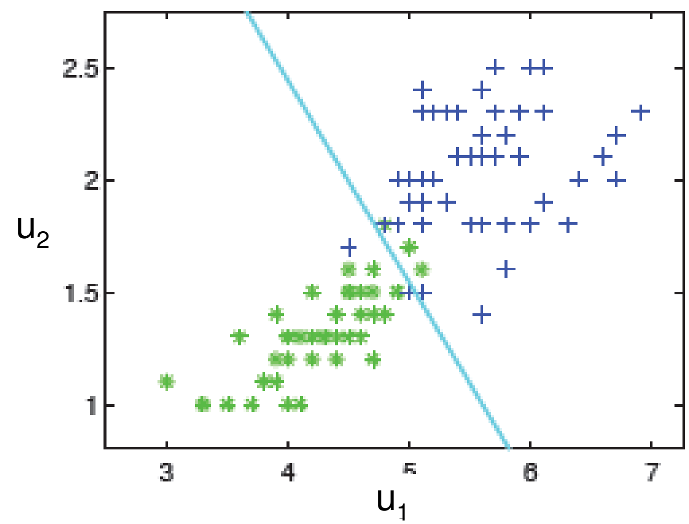


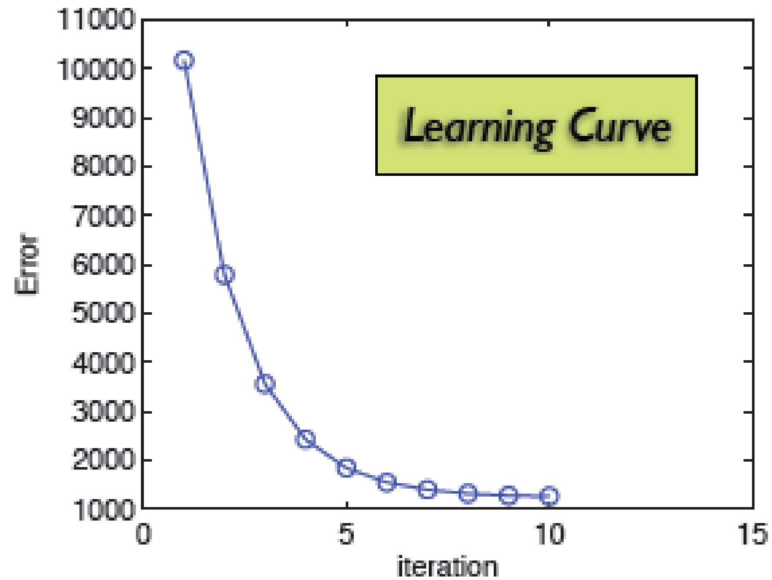


Learning the Decision Boundary

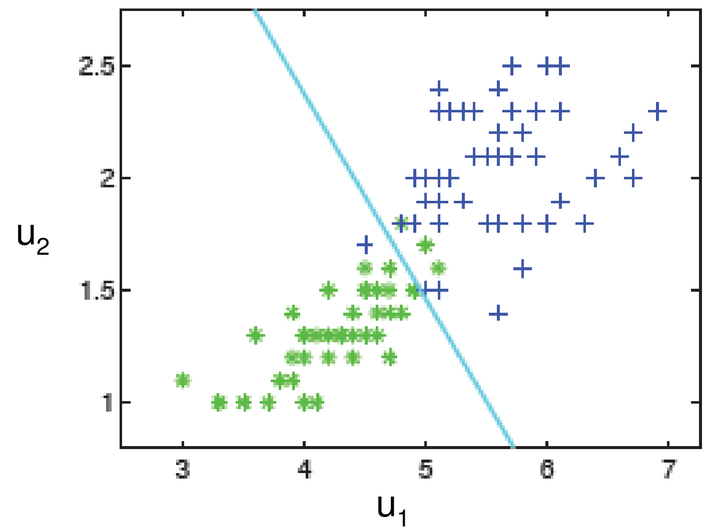
- Each iteration updates the gradient

- Value of ϵ
  – Small: O.1/N
  – Too large: learning diverges
  – Too small: slow convergence

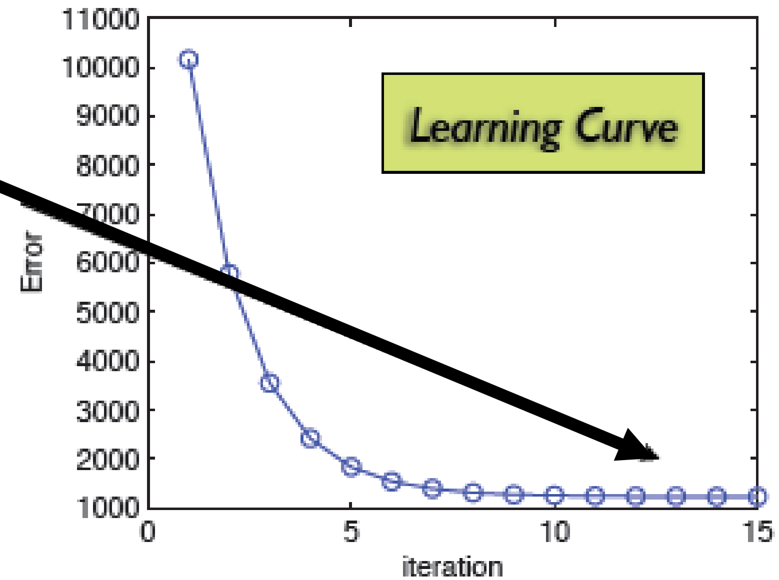




Learning the Decision Boundary

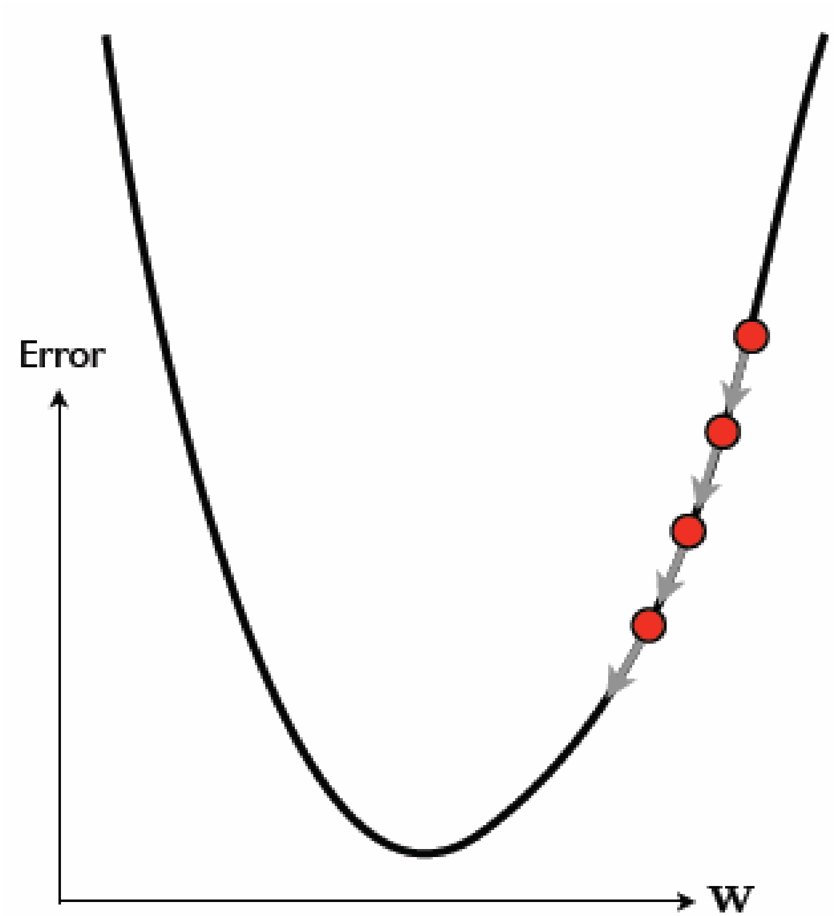


- Learning converges onto the solution that minimizes the error
- For linear networks, this is guaranteed to converge to the minimum

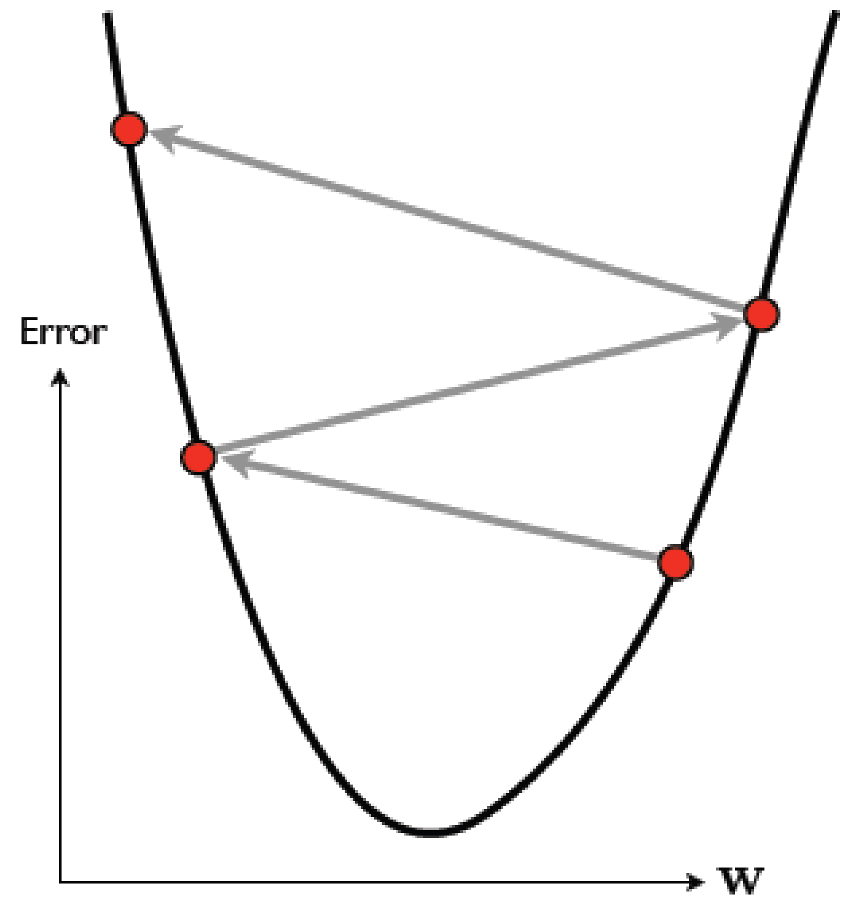


Influence of ϵ on learning

Learning is slow when ϵ is too small



Learning can oscillate between different sides of the minimum if ϵ is too large



Exercise: learning the OR function

- Error function in terms of vd_m, w_1, w_2, x_{m1}, x_{m2}
- Gradient of the error function with respect to w_1 and w_2

  $dE/dw_1 =$

  $dE/dw_2 =$

- Write the gradient update rule

  $w_1 \leftarrow$

  $w_2 \leftarrow$

Exercise: learning the OR function

- Error function to be minimized

  $E(w) = 0.5\ \Sigma_m (vd_m - (-0.5 + w_1x_{m1} + w_2x_{m2}))^2$
- Gradient of the error function with respect to w_1 and w_2

$dE/dw_1 = (vd_m - (-0.5 + w_1x_{m1} + w_2x_{m2}))(-x_{m1}) = -E_mx_{m1}$

$dE/dw_2 = (vd_m - (-0.5 + w_1x_{m1} + w_2x_{m2}))(-x_{m2}) = -E_mx_{m2}$

- Considering the gradient update rule

  $w_1 \leftarrow w_1 + \epsilon\ E_mx_{m1}$

  $w_2 \leftarrow w_2 + \epsilon\ E_mx_{m2}$