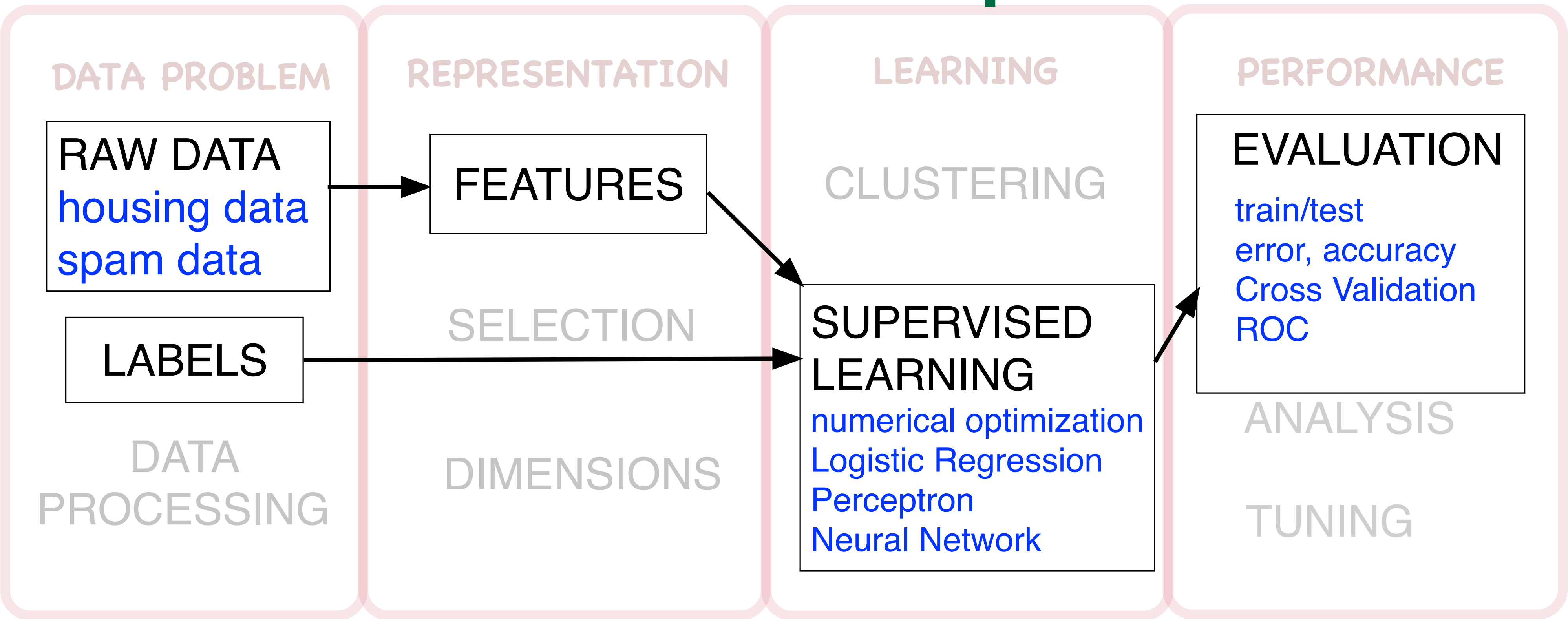
Neural Networks

Module2 : learning with Gradient Descent

module 2: numerical optimization

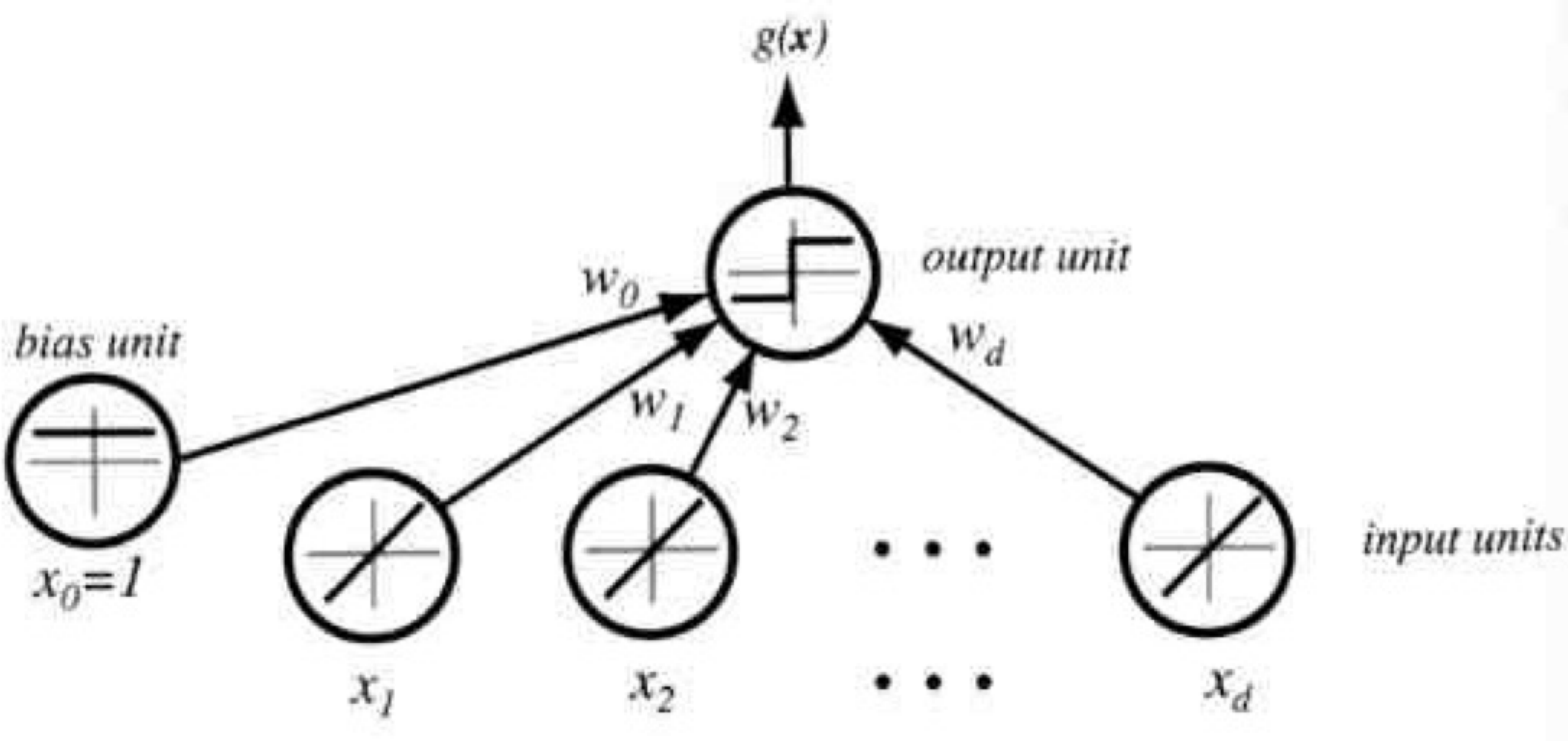


- formulate problem by model/parameters
- formulate error as mathematical objective
- optimize numerically the parameters for the given objective
- usually algebraic setup
  - involves matrices and calculus
- probabilistic setup (likelihoods) next module

Module 2 Objectives / Neural Networks

- perceptron rules
- neural network idea, philosophy, construction
- NN weights
- Backpropagation : training NN using gradient descent
- NN modes, autoencoders
- run NN-autoencoder on a simple problem

The perceptron



$$h_{\mathbf{w}}(\mathbf{x}) = \mathbf{x}\mathbf{w} = \sum_{d=0}^{D} x^d w^d$$

The perceptron

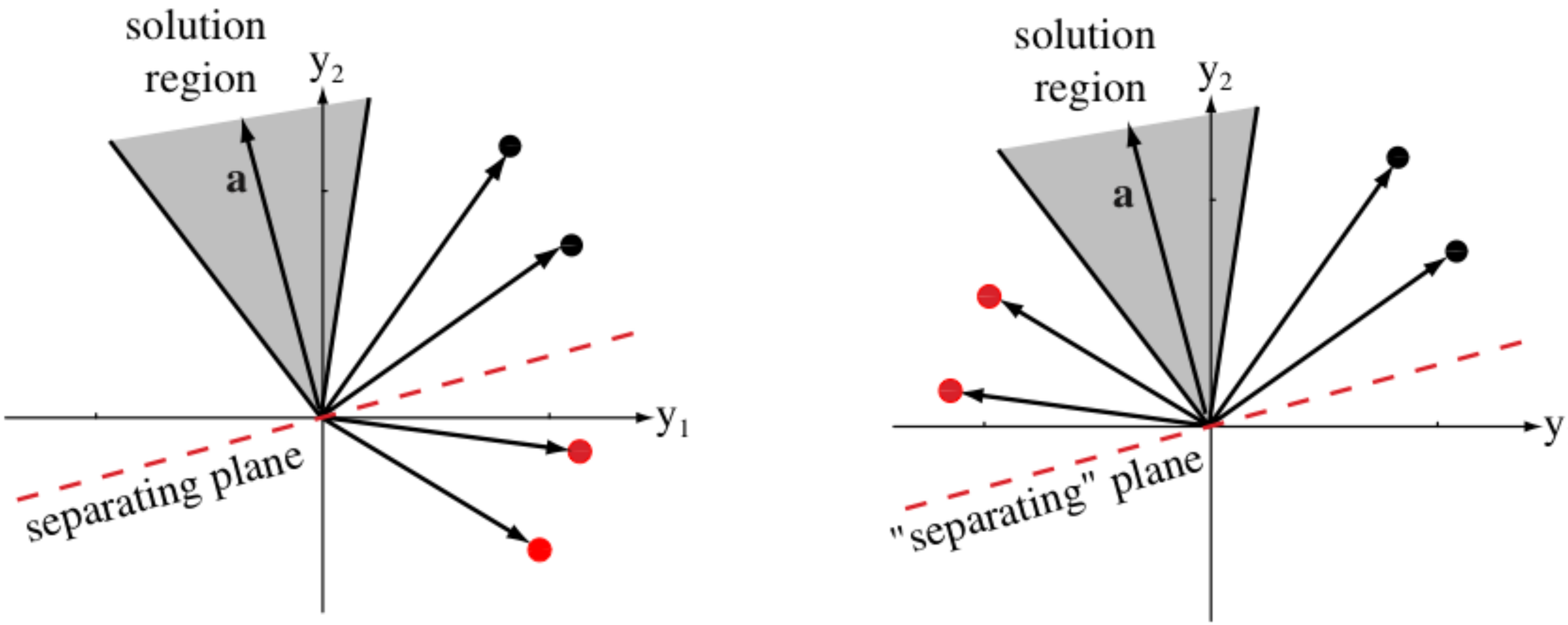
- (like with regression) we are looking for a linear classifier

$$h_{\mathbf{w}}(\mathbf{x}) = \mathbf{x}\mathbf{w} = \sum_{d=0}^{D} x^d w^d$$

- error different than regression: weighted sum over misclassified points set M

$$J(\mathbf{w}) = \sum_{\mathbf{x} \in M} -h_{\mathbf{w}}(\mathbf{x}) = \sum_{\mathbf{x} \in M} -\mathbf{x}\mathbf{w}$$

Perceptron – geometry



- perceptron is a linear (hyperplane) separator
- for simplicity, will transform data points with y=-1 (left) to y=1 (right) by reversing the sign

The perceptron

- To optimize for perceptron error, use gradient descent

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \sum_{\mathbf{x} \in M} -\mathbf{x}^T$$

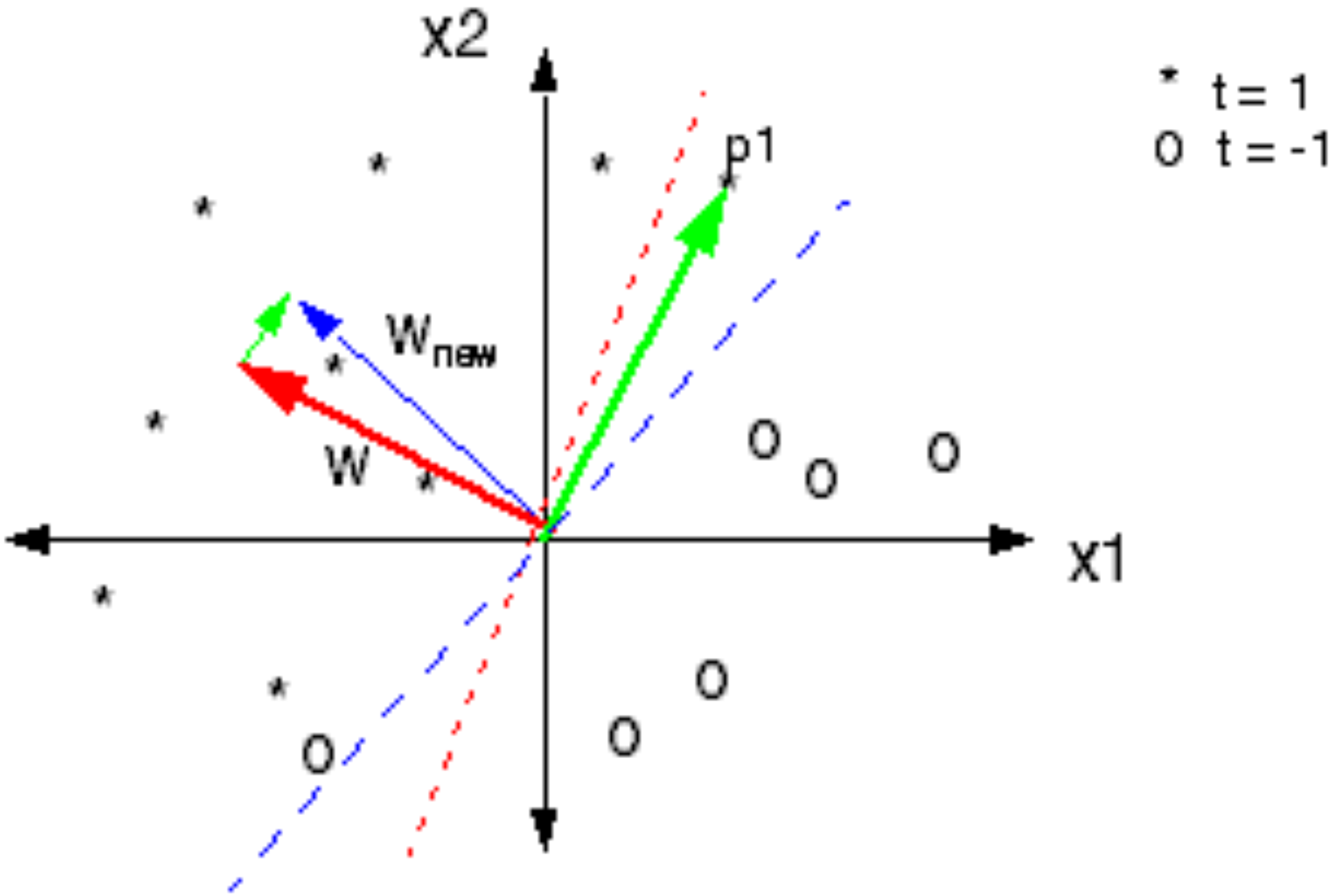
- with update rule

$$\mathbf{w} := \mathbf{w} + \lambda \sum_{\mathbf{x} \in M} \mathbf{x}^T$$

- batch update:

1. init $\mathbf{w}$
2. LOOP
3. get M = set of missclassified data points
4. $\mathbf{w} = \mathbf{w} + \lambda \sum_{\mathbf{x} \in M} \mathbf{x}^T$
5. UNTIL $|\lambda \sum_{\mathbf{x} \in M} \mathbf{x}| < \epsilon$

perceptron update – intuition



- perceptron update: the plane (dotted red) normal w (red arrow) moves in the direction of misclassified p1 until p1 is on the correct side.

Perceptron proof of convergence

- if data is indeed linearly separable, the perceptron will find the separator line.

Proof of perceptron convergence Assuming data is linearly separable , or there is a solution $\bar{\mathbf{w}}$ such that $\mathbf{x}\bar{\mathbf{w}} > 0$ for all $\mathbf{x}$.
Lets call $\mathbf{w}_k$ the $\mathbf{w}$ obtained at the k-th iteration (update). Fix an $\alpha > 0$. Then

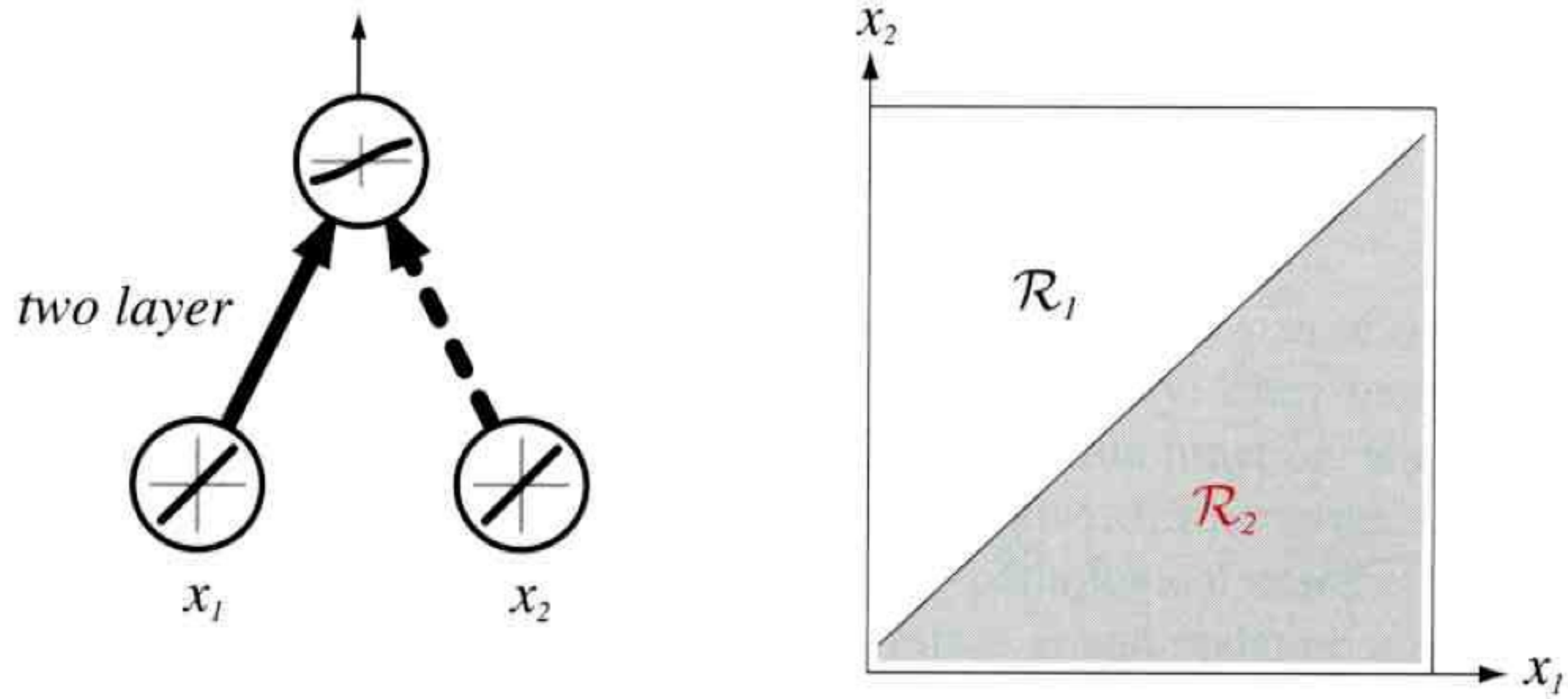
$$\mathbf{w}_{k+1} - \alpha\bar{\mathbf{w}} = (\mathbf{w}_k - \alpha\bar{\mathbf{w}}) + \mathbf{x}_k^T$$

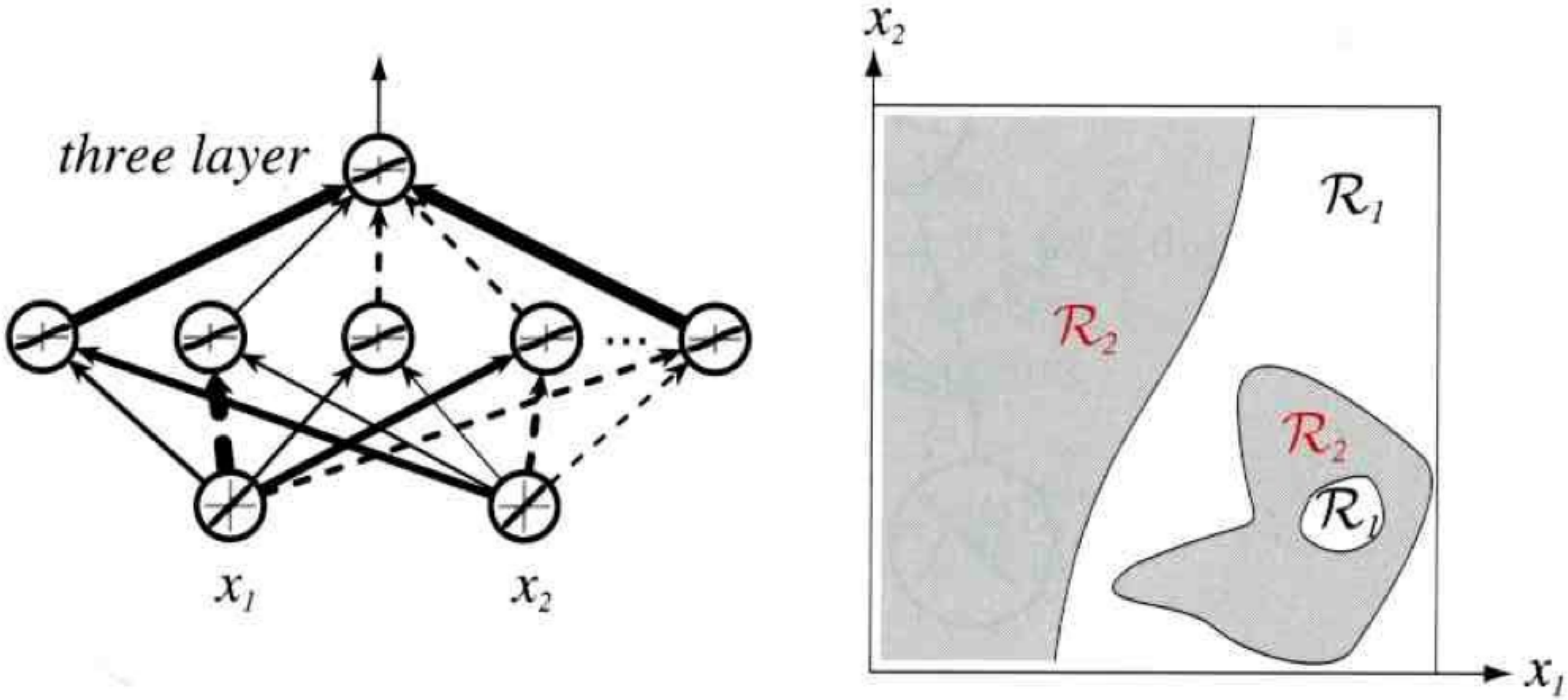
where $\mathbf{x}_k$ is the datapoint that updated $\mathbf{w}$ at iteration k. Then

$$||\mathbf{w}_{k+1} - \alpha\bar{\mathbf{w}}||^2 = ||\mathbf{w}_k - \alpha\bar{\mathbf{w}}||^2 + 2\mathbf{x}_k(\mathbf{w}_k - \alpha\bar{\mathbf{w}}) + ||\mathbf{x}_k||^2 \leq ||\mathbf{w}_k - \alpha\bar{\mathbf{w}}||^2 - 2\mathbf{x}_k\alpha\bar{\mathbf{w}} + ||\mathbf{x}_k||^2$$

Since $\mathbf{x}_k\bar{\mathbf{w}} > 0$ all we need is an α sufficiently large to show that this update process cannot go on forever. When it stops, all datapoints must be classified correctly.

Multilayer perceptrons





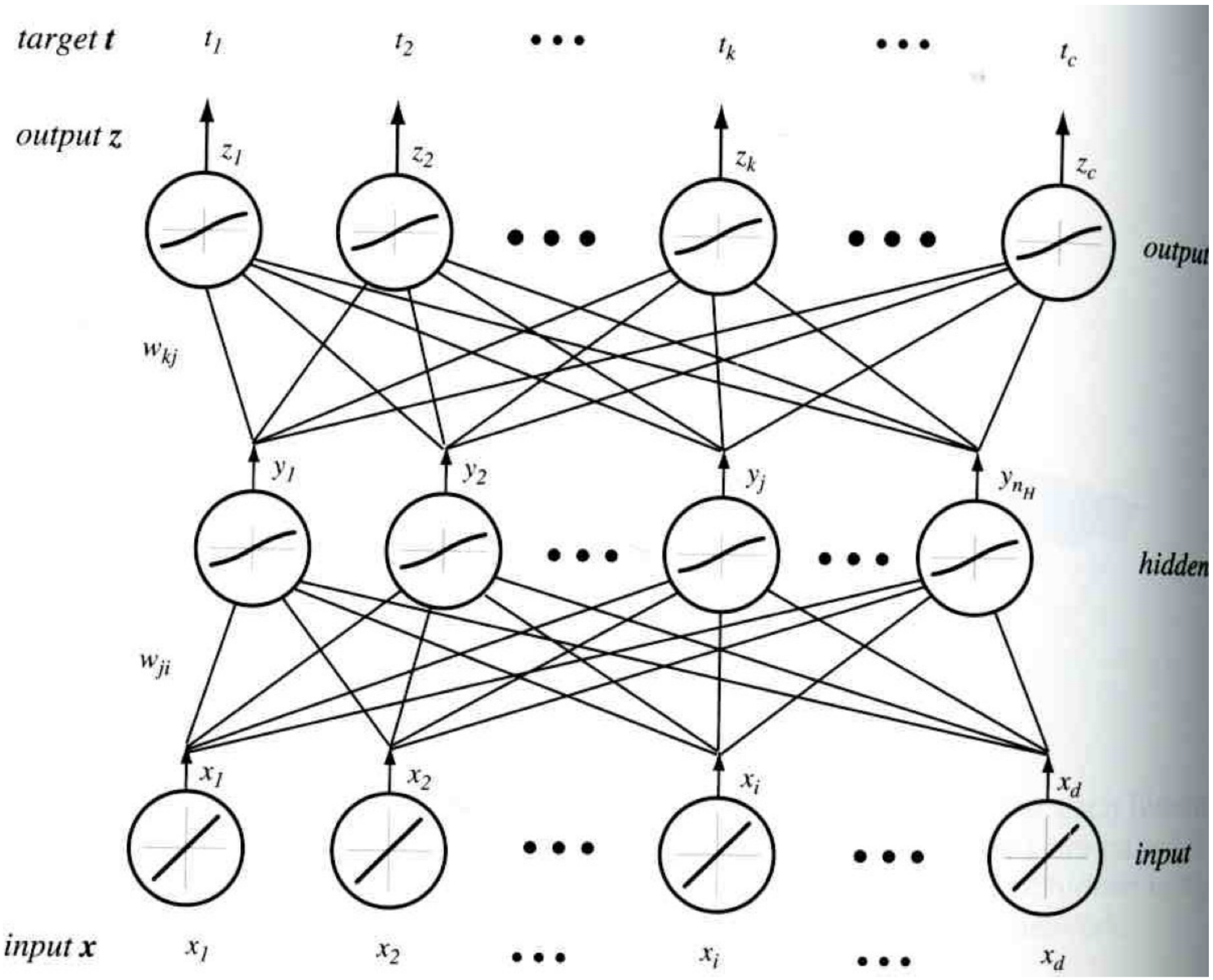
Checkpoint: XOR perceptron

- build/explain a 3-layer perceptron that give the same classification as the logical XOR function

$$XOR(x, y) = OR(x, y) \; AND \; (NOT(AND(x, y)))$$

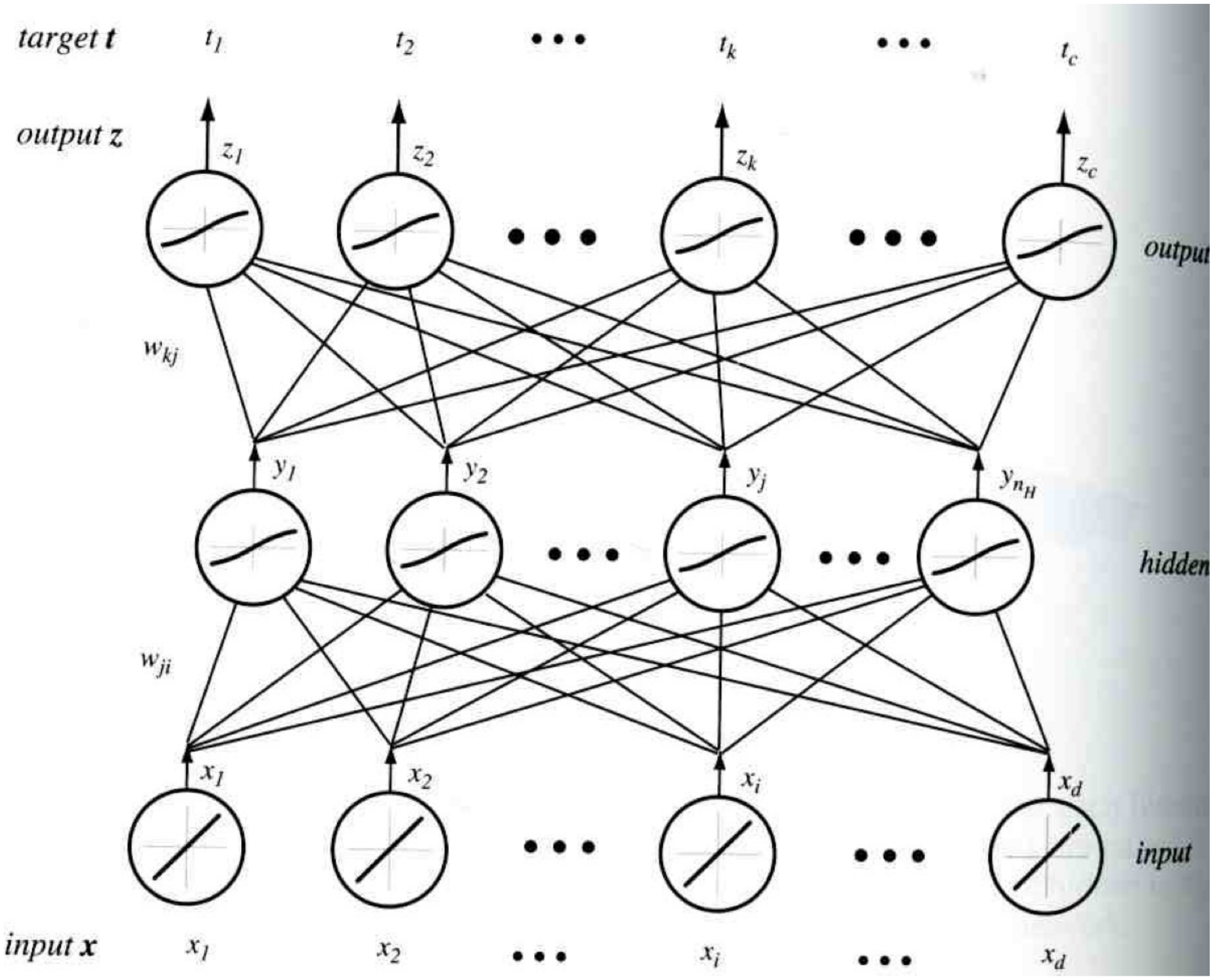
- your answer is required! Submit via dropbox.

Neural Networks



- NN is a stack of connected perceptrons
- bottom up:
  - input layer
  - hidden layer
  - output layer
- multilayer NN very very powerful in that they can approximate almost any function
  - with enough training data

Neural Networks



- Each unit performs first a linear combination of inputs

$$net_j = \sum_{i=1}^{d} x_i w_{ji} + w_{j0} = \sum_{i=0}^{d} x_i w_{ji} = \mathbf{w}_j^t \mathbf{x}$$

- Then applies a nonlinear (ex. logistic) function "f" before outputting a value

$$y_j = f(net_j)$$

- Three layer NN output can be expressed mathematically as

$$g_k(\mathbf{x}) = z_k = f\left(\sum_j w_{kj} f\left(\sum_i w_{ij} x_i + w_{j0}\right) + w_{k0}\right)$$

Training the NN weights (w)

- one datapoint

$$J(w) = \frac{1}{2}\sum_k (t_k - z_k)^2$$

$$\Delta w_{pq} = -\lambda \frac{\partial J}{\partial w_{pq}},$$

- set of weights up (close to output):

$$\frac{\partial J}{\partial w_{kj}} = \frac{\partial J}{\partial net_k}\frac{\partial net_k}{\partial w_{kj}} = -\delta_k \frac{\partial net_k}{\partial w_{kj}}$$

$$\delta_k = -\frac{\partial J}{\partial net_k} = -\frac{\partial J}{\partial z_k}\frac{\partial z_k}{\partial net_k} = (t_k - z_k) f'(net_k)$$

$$\frac{\partial net_k}{\partial w_{kj}} = y_j$$

- we obtain the hidden-output weight update rule

$$w_{kj} = w_{kj} + \lambda (t_k - z_k) f'(net_k) y_j$$

Training the NN weights (w)

- weight first set of weights (close to input)

$$\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial y_j}\frac{\partial y_j}{\partial net_j}\frac{\partial net_j}{\partial w_{ji}}$$

$$\frac{\partial J}{\partial y_j} = \frac{\partial[\frac{1}{2}\sum_k (t_k - z_k)^2]}{\partial y_j}$$

$$= -\sum_k (t_k - z_k)\frac{\partial z_k}{\partial net_k}\frac{\partial net_k}{\partial y_j}$$

$$\frac{\partial h_j}{\partial net_j} = f'(net_j)$$

$$\frac{\partial net_j}{\partial w_{ji}} = x_i$$

$$w_{ji} \leftarrow w_{ji} - \lambda[\sum_k (t_k - z_k) f'(net_k) w_{kj}] f'(net_j) x_i$$

NN training

STOCHASTIC TRAINING

Select x_t (randomly chosen)

$$w_{ij} = w_{ij} + \lambda \delta_j x_i$$

$$w_{jk} = w_{jk} + \lambda \delta_k y_j$$

until $| \bigtriangledown_w J| < \epsilon$

BATCH TRAINING

for each iteration:

for each x_t

$$\delta w_{ij} = \delta w_{ij} + \lambda \delta_j x_i$$

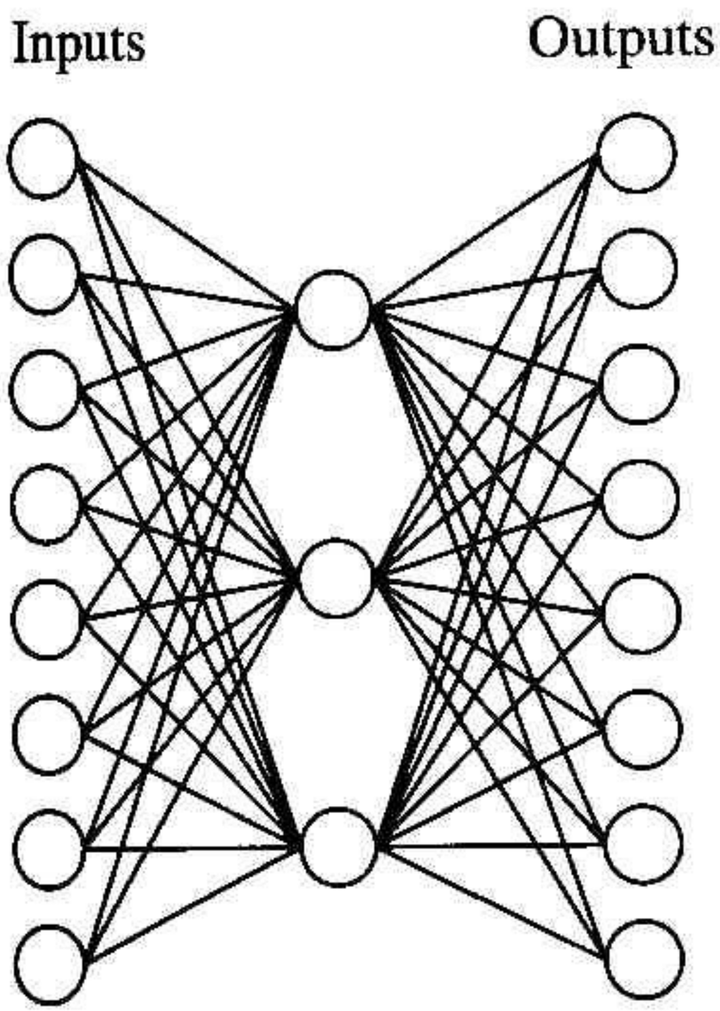
$$\delta w_{jk} = \delta w_{jk} + \lambda \delta_k y_j$$

$$w_{ij} \leftarrow w_{ij} + \delta w_{ij}$$

$$w_{jk} \leftarrow w_{jk} + \delta w_{jk}$$

until $|| \bigtriangledown_w J|| < \epsilon$

Autoencoders



Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.15	.99	.99	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.01	.11	.88	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

- network is “rotated”
  - from left to right: input-hidden-ouput
- input and output are the same values
  - hidden layer encodes the input and decodes back to itself

BackPropagation (Tom Mitchell book)

BACKPROPAGATION($training\_examples, \eta, n_{in}, n_{out}, n_{hidden}$)

Each training example is a pair of the form $\langle \vec{x}, \vec{t} \rangle$, where $\vec{x}$ is the vector of network input values, and $\vec{t}$ is the vector of target network output values.

η is the learning rate (e.g., .05). n_{in} is the number of network inputs, n_{hidden} the number of units in the hidden layer, and n_{out} the number of output units.

The input from unit i into unit j is denoted x_{ji}, and the weight from unit i to unit j is denoted w_{ji}.

- Create a feed-forward network with n_{in} inputs, n_{hidden} hidden units, and n_{out} output units.
- Initialize all network weights to small random numbers (e.g., between −.05 and .05).
- Until the termination condition is met, Do
  - For each $\langle \vec{x}, \vec{t} \rangle$ in $training\_examples$, Do

    Propagate the input forward through the network:

    1. Input the instance $\vec{x}$ to the network and compute the output o_u of every unit u in the network.

    Propagate the errors backward through the network:

    2. For each network output unit k, calculate its error term δ_k

    $$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k) \quad \text{(T4.3)}$$

    3. For each hidden unit h, calculate its error term δ_h

    $$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh}\delta_k \quad \text{(T4.4)}$$

    4. Update each network weight w_{ji}

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

    where

    $$\Delta w_{ji} = \eta \, \delta_j \, x_{ji} \quad \text{(T4.5)}$$