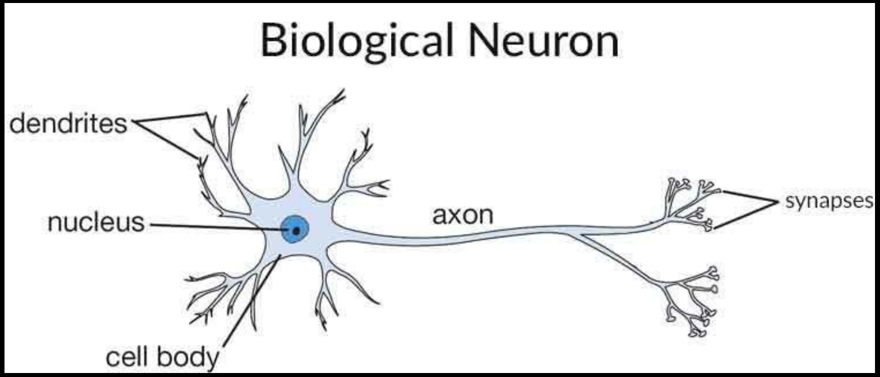
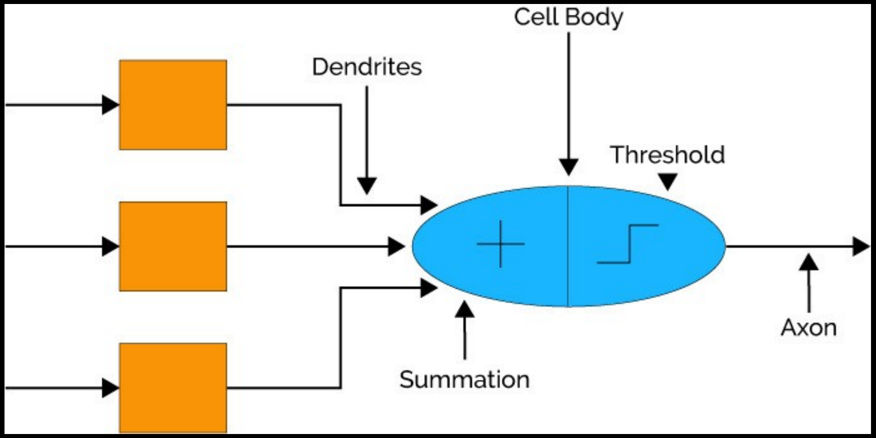
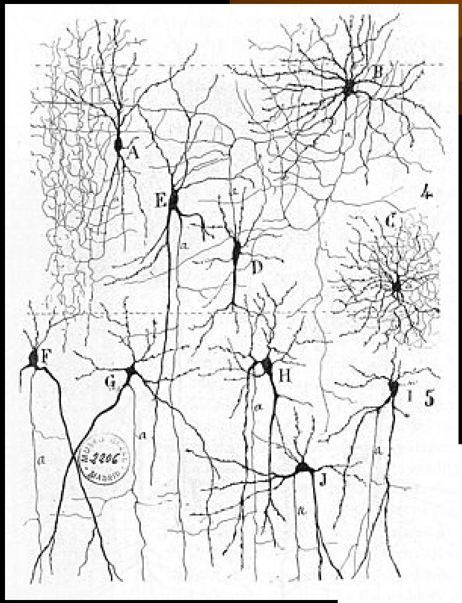
Artificial Neural Networks

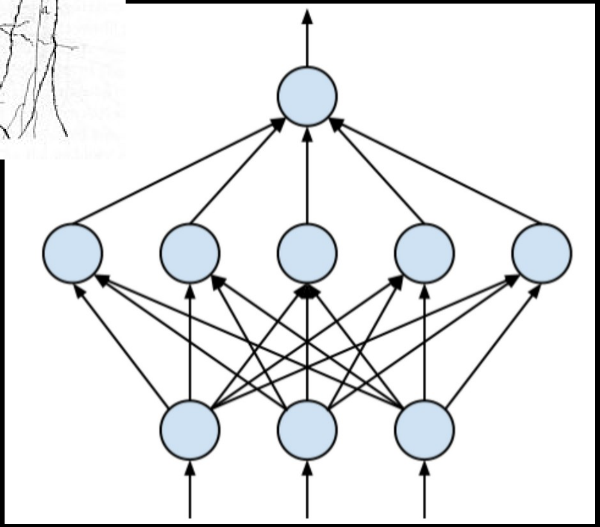
- Robust approach to approximating real and discrete-valued target functions
- Biological Motivations
  - Using ANNs to model and study biological learning processes
  - Obtaining highly effective Machine Learning algorithms by mirroring biological processes

Biological motivations

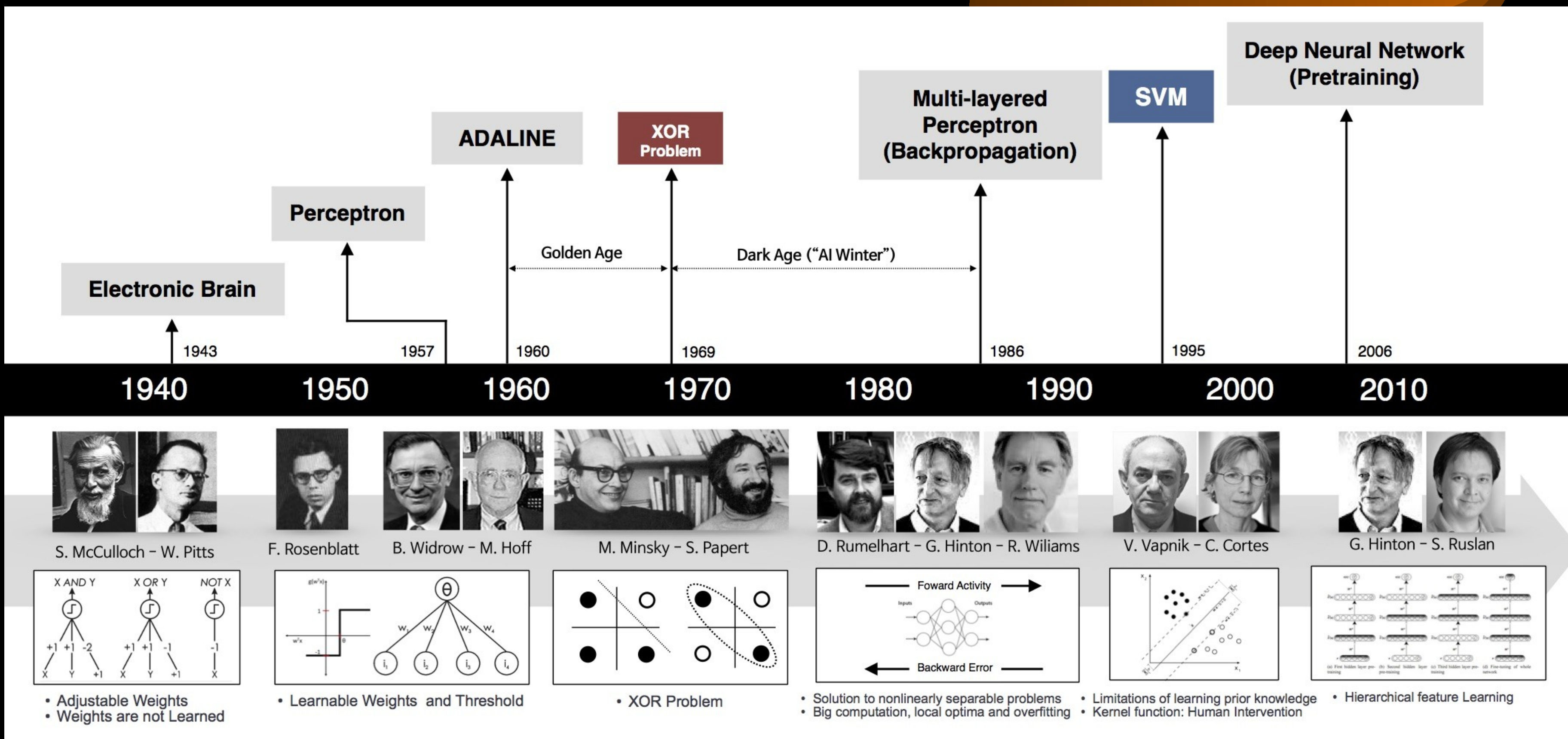








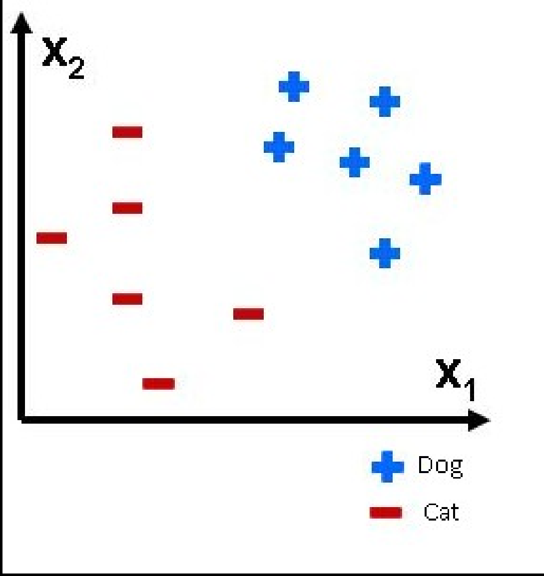
Timeline



2018: Turing Award to developers of Deep Neural Networks

Appropriate Problems for ANNs

- Regression and Multiclass classification
- High dimension inputs
- Training examples may contain errors
- Long training times are acceptable
- Fast evaluation of the learned target function may be required
- Ability to understand the target function not important



Perceptrons

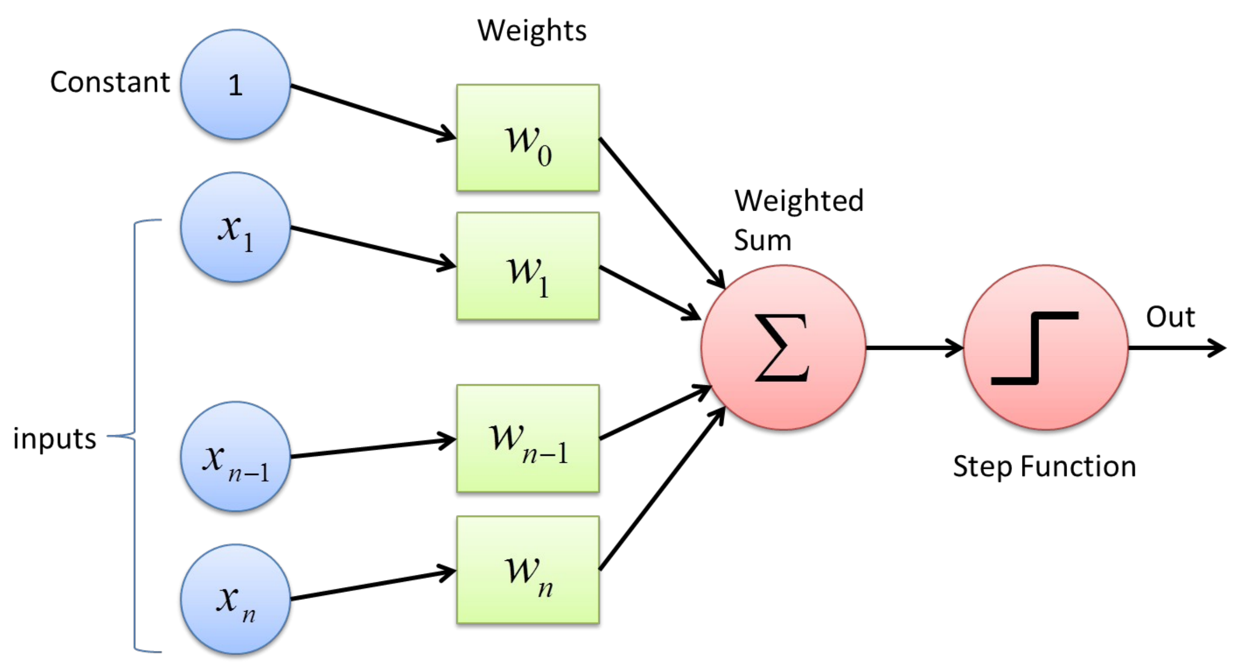
$$o(x_1, x_2 \ldots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \ldots + w_n x_n > 0 \\ -1 & \text{otherwise} \end{cases}$$

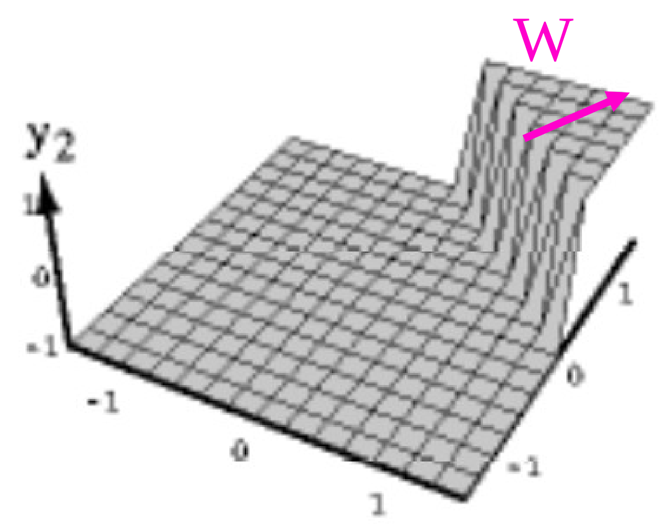
$$o(\mathbf{x}) = \text{sgn}(\mathbf{w.x}) \qquad (x_0 = 1)$$

Hypothesis Space:

All Hyperplanes $\qquad H = \{\mathbf{w} \mid \mathbf{w} \in \Re^{n+1}\}$

Perceptrons





The Perceptron Training Rule

$w_i \leftarrow w_i + \Delta w_i$ $\Delta w_i = \eta\,(t - o)\,x_i$

t: target output for the current training example

o: output generated by the perceptron

η: learning rate

Gradient Descent

Unthresholded perceptrons: $o(\mathbf{x}) = \mathbf{w.x}$

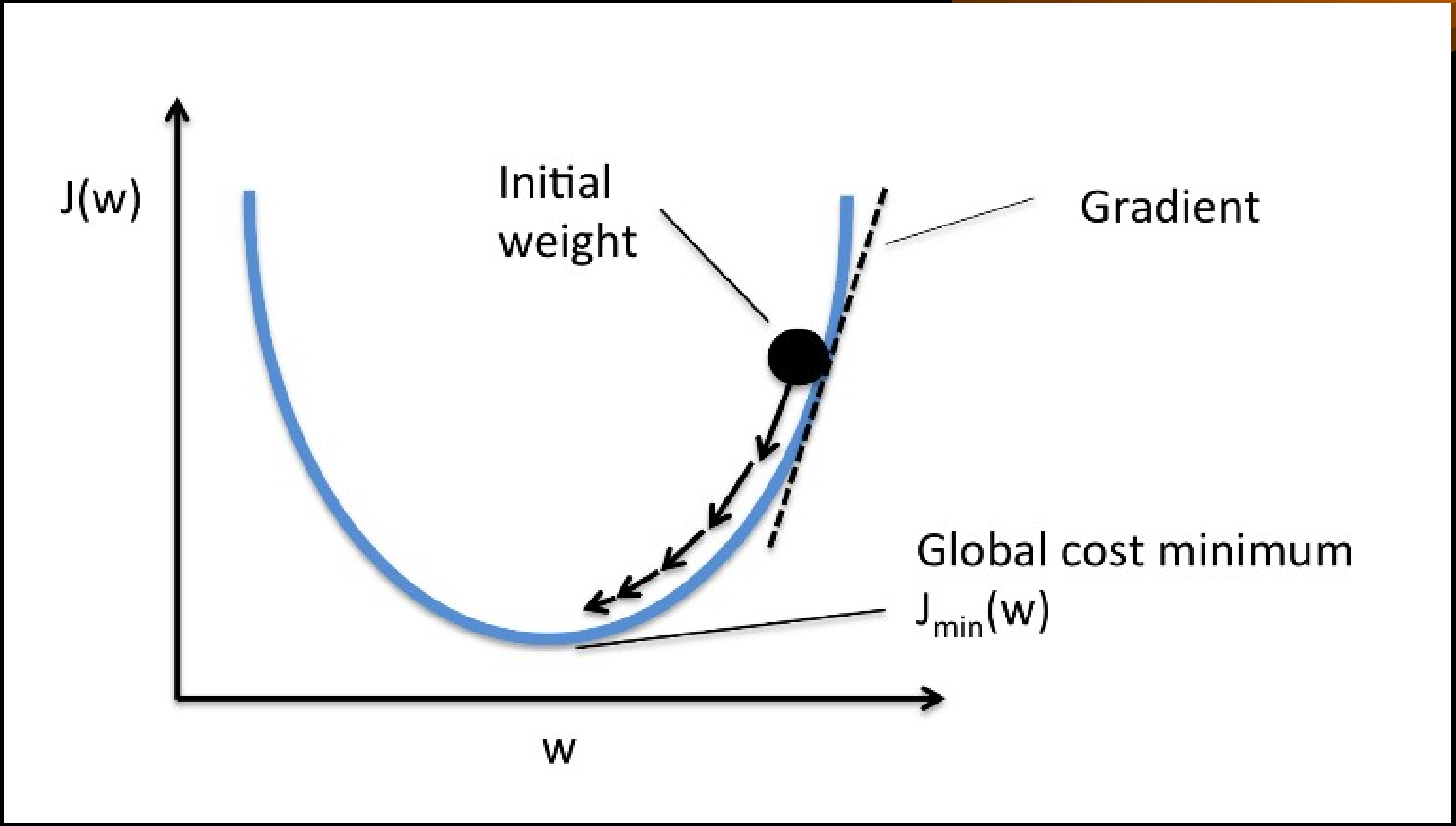
Another strategy:

-define a cost function

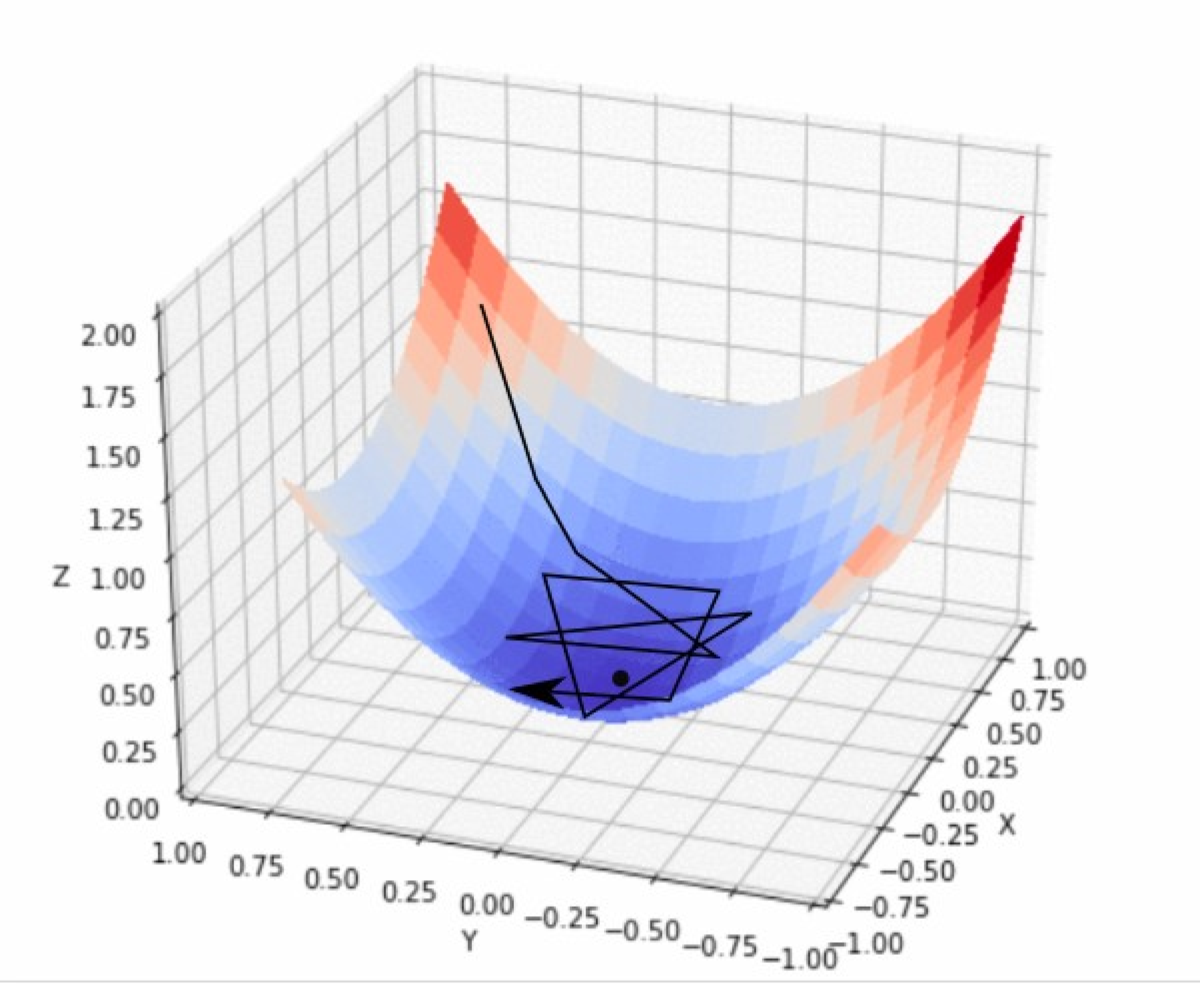
-minimize it

$$E(\mathbf{w}) = \frac{1}{2} \sum_D (t-o)^2$$

Gradient Descent



Gradient Descent



Gradient Descent

Unthresholded perceptrons: $o(\mathbf{x}) = \mathbf{w.x}$

$w_i \leftarrow w_i + \Delta w_i$ $\quad\quad$ $\Delta w_i = - \eta\ \partial E / \partial w_i$

$E(\mathbf{w}) = \frac{1}{2} \sum_D (t-o)^2$ $\quad\quad$ $\partial E / \partial w_i = \sum_D (t-o)\ x_i$

Gradient Descent

GRADIENT-DESCENT($training\_examples, \eta$)

Each training example is a pair of the form $\langle \vec{x}, t \rangle$, where $\vec{x}$ is the vector of input values, and t is the target output value. η is the learning rate (e.g., .05).

- Initialize each w_i to some small random value
- Until the termination condition is met, Do
  - Initialize each Δw_i to zero.
  - For each $\langle \vec{x}, t \rangle$ in $training\_examples$, Do
    - Input the instance $\vec{x}$ to the unit and compute the output o
    - For each linear unit weight w_i, Do

$$\Delta w_i \leftarrow \Delta w_i + \eta (t - o) x_i \tag{T4.1}$$

  - For each linear unit weight w_i, Do

$$w_i \leftarrow w_i + \Delta w_i \tag{T4.2}$$

Gradient Descent and Delta Rule

- For each $\langle \vec{x}, t \rangle$ in *training_examples*, Do
  - Input the instance $\vec{x}$ to the unit and compute the output o
  - For each linear unit weight w_i, Do

$$\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i \qquad \text{(T4.1)}$$

- For each linear unit weight w_i, Do

$$w_i \leftarrow w_i + \Delta w_i$$

Move the update into the inner loop

Delta (Adaline, Widrow-Hoff, LMS) Rule:

$w_i \leftarrow w_i + \Delta w_i$ $\Delta w_i = \eta\ (t\text{-}o)\ x_i$

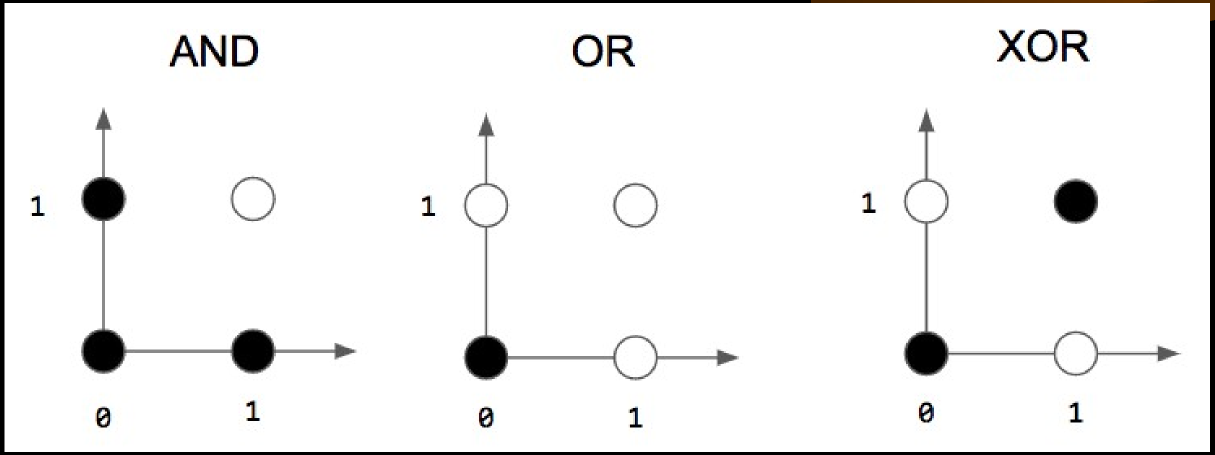
Remarks

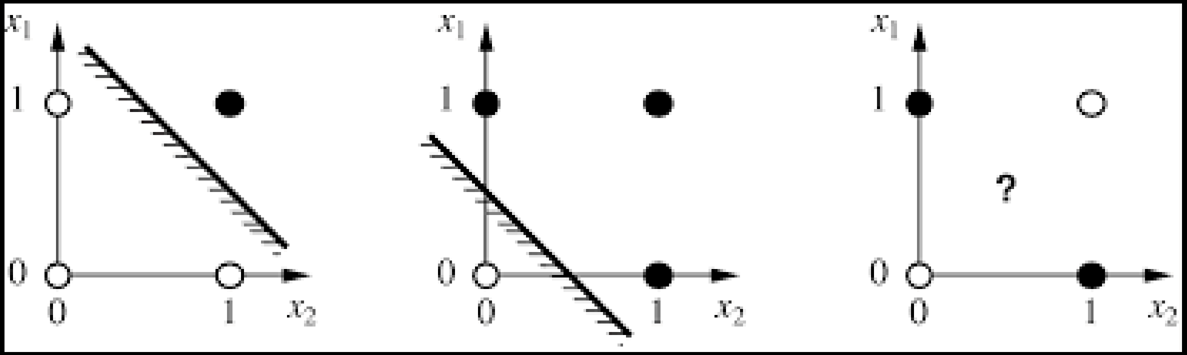
- The perceptron training rule converges after a finite number of iterations to a hypothesis that perfectly classifies the data, provided the examples are linearly separable
- The delta rule converges only asymptotically toward the minimum error hypothesis, but regardless of the data linear separability (general approach)

Representational Power

- Perceptrons can represent all the primitive Boolean functions AND, OR, NAND (¬AND) and NOR (¬OR)
- They cannot represent all Boolean functions (for example, XOR)
- Every Boolean function can be represented by some network of perceptrons two levels deep

The XOR problem





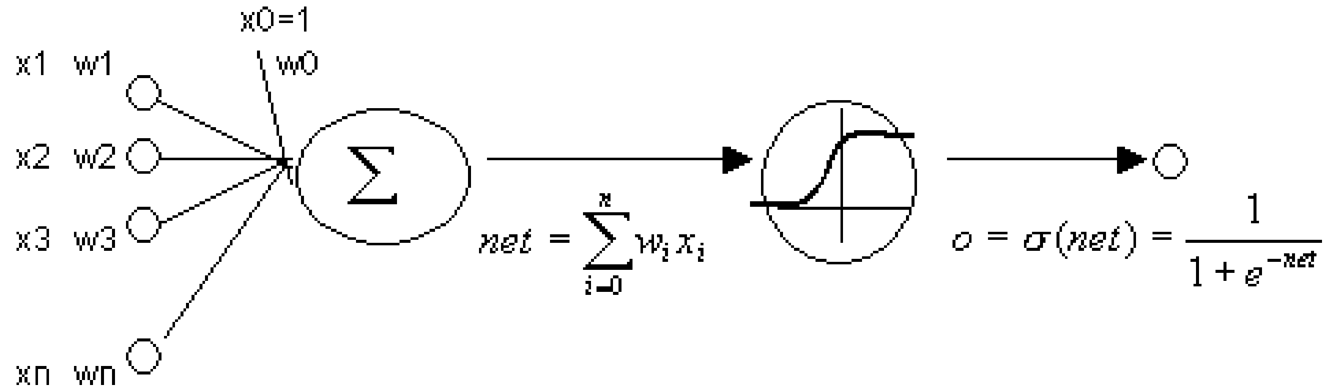
Multilayer Networks

ANNs with two or more layers are able to represent complex nonlinear decision surfaces.

We need nonlinear unit in the hidden layers.

Differentiable units can help learning.

Multilayer Networks



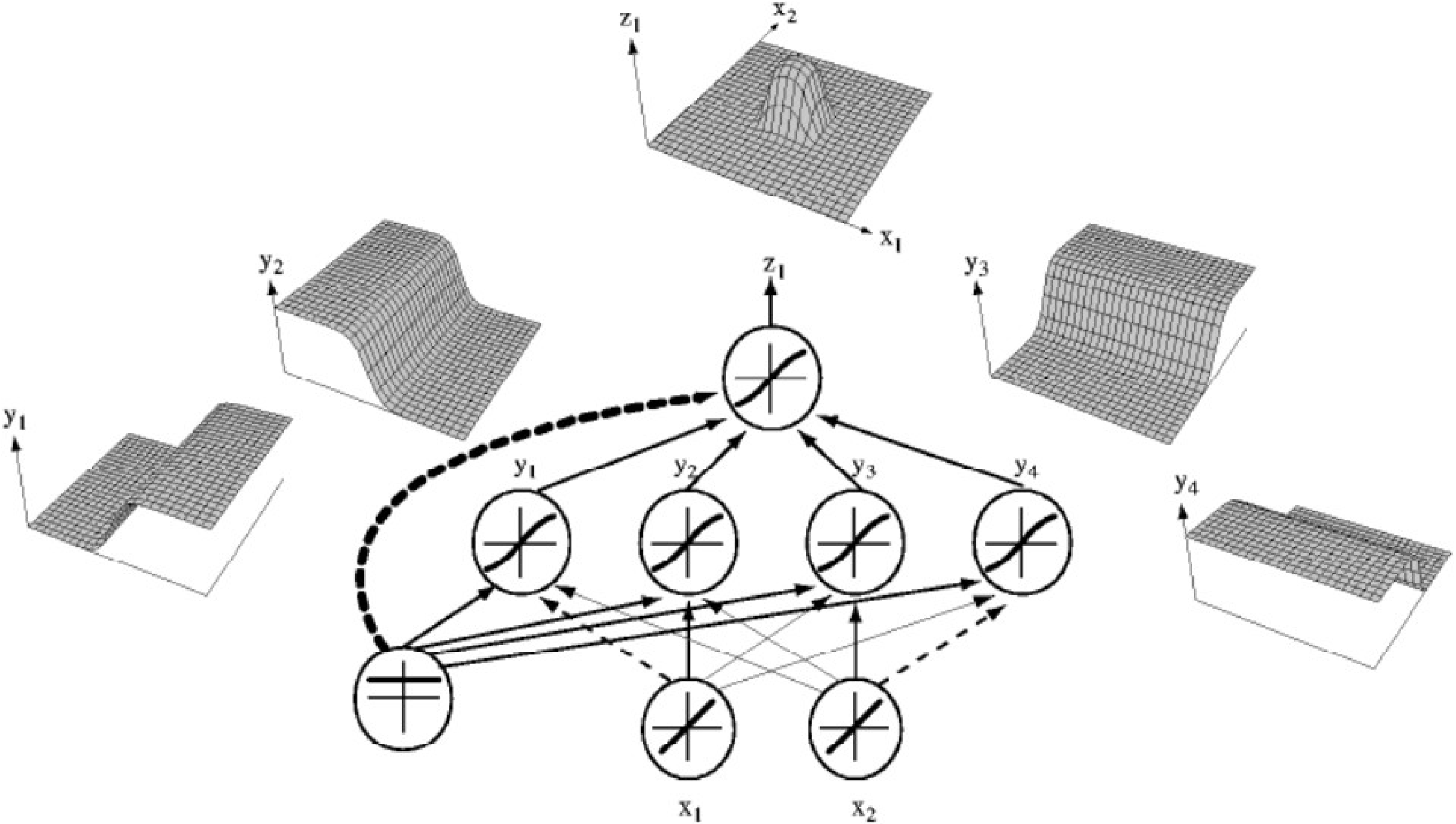
[The sigmoid threshold unit]

– Differentiable Threshold (Sigmoid) Units

$o = \sigma(\mathbf{w.x})$ $\sigma(y) = 1/(1+e^{-y})$

$\partial\sigma/\partial y = \sigma(y)\,[1-\sigma(y)]$

Example



The BackPropagation Algorithm

x_{ji}: i-th input to unit j

w_{ji}: weight associated with the i-th input to unit j

$net_j = \sum_i w_{ji} x_{ji}$: weighted sum of inputs for unit j

o_j : output computed by unit j

t_j : target output for unit j

DS(j) : DownStream(j), set of units whose inputs include the output of unit j

The BackPropagation Algorithm

$o=\sigma(\mathbf{w.x})$ $\sigma(y)=1/(1+e^{-y})$ $\partial\sigma/\partial y= \sigma(y).[1-\sigma(y)]$

$$E(\boldsymbol{w}) = ½ \sum_D \sum_{k\in \text{outputs}} (t_k-o_k)^2 = \sum_D E_d$$

$$\partial E_d/\partial w_{ji} = \partial E_d/\partial net_j \, . \, x_{ji}$$

The BackPropagation Algorithm

– Case 1: Output Units k

$$\partial E_d/\partial net_k = \partial E_d/\partial o_k \times \partial o_k/\partial net_k \equiv -\delta_k$$

$$\partial E_d/\partial o_k = -(t_k - o_k) \quad \partial o_k/\partial net_k = o_k(1 - o_k)$$

$$\Rightarrow \qquad \Delta w_{kj} = -\eta\, \partial E_d/\partial w_{kj} = \eta\, (t_k - o_k)\, o_k(1 - o_k)\, x_{kj}$$

The BackPropagation Algorithm

– Case 2: Hidden Units j

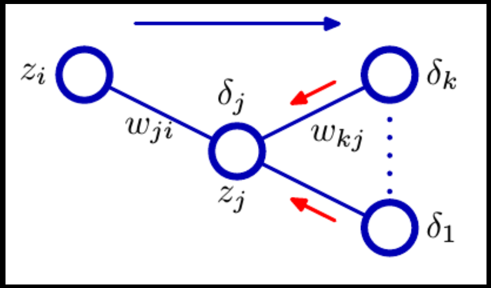
$$\partial E_d/\partial net_j = \Sigma_{k\in DS(j)}\ \partial E_d/\partial net_k \times \partial net_k/\partial net_j$$

$$= -\ \Sigma_{k\in DS(j)}\ \delta_k \times \partial net_k/\partial o_j \times \partial o_j/\partial net_j$$

$$= -\ \Sigma_{k\in DS(j)}\ \delta_k\, w_{kj}\, o_j\,(1-o_j)$$

$$\Rightarrow \qquad \delta_j = -\ o_j\,(1-\ o_j)\ \Sigma_{k\in DS(j)}\ \delta_k\, w_{kj}$$

$$\Delta w_{j\,i} = -\ \eta\ \partial E_d/\partial w_{ji} = \eta\ \delta_j\, x_{ji}$$

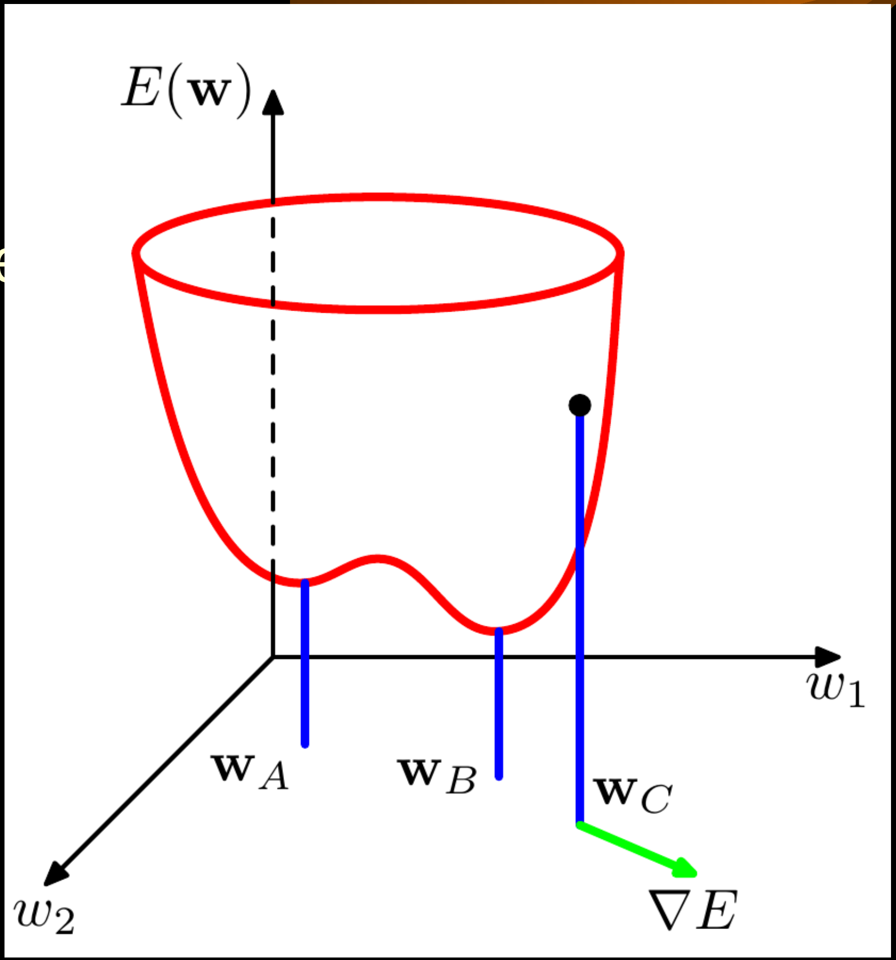


Remarks on the BP Algorithm

– Implements a gradient descend search

Remarks on the BP Algorithm

– Implements a gradie



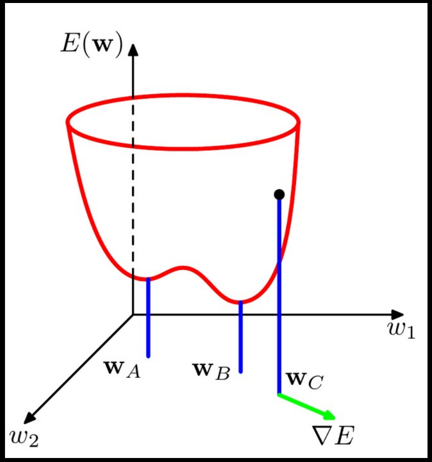
Remarks on the BP Algorithm

- Implements a gradient descend search
- Heuristics

  Momentum term

  Stochastic gradient descent

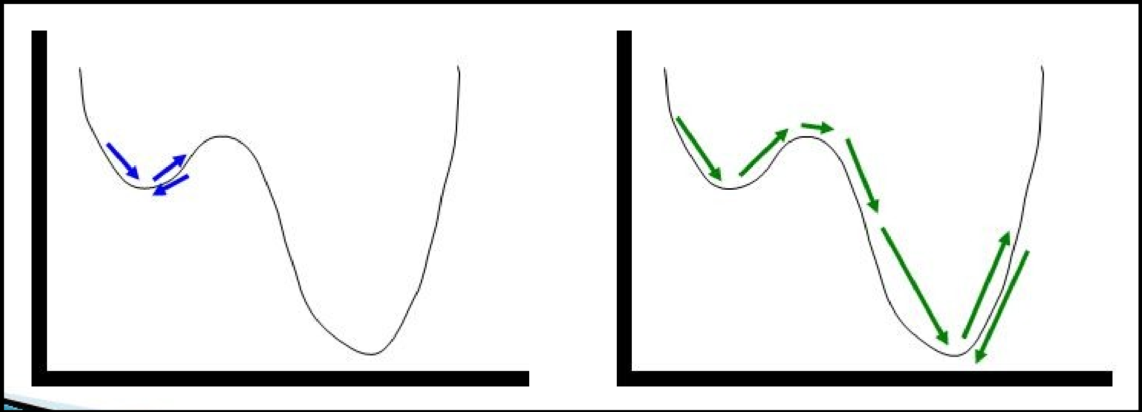
  Training multiple networks

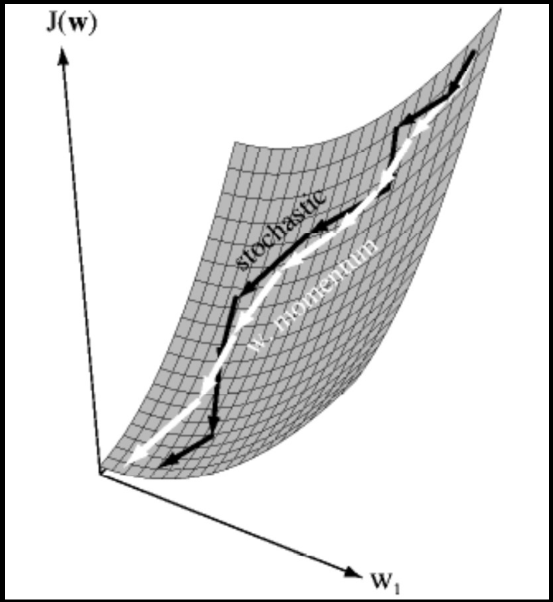


Momentum term

– Adds “momentum” to gradient descent

$$\Delta w_{t+1} = - \eta \, \partial E / \partial w_t + \alpha \, \Delta w_t$$





Stochastic gradient descent

GRADIENT-DESCENT(*training_examples*, η)

Each training example is a pair of the form $\langle \vec{x}, t \rangle$, where $\vec{x}$ is the vector of input values, and t is the target output value. η is the learning rate (e.g., .05).

- Initialize each w_i to some small random value
- Until the termination condition is met, Do
  - Initialize each Δw_i to zero.
  - For each $\langle \vec{x}, t \rangle$ in *training_examples*, Do
    - Input the instance $\vec{x}$ to the unit and compute the output o
    - For each linear unit weight w_i, Do

$$\Delta w_i \leftarrow \Delta w_i + \eta (t - o) x_i \quad \text{(T4.1)}$$

  - For each linear unit weight w_i, Do

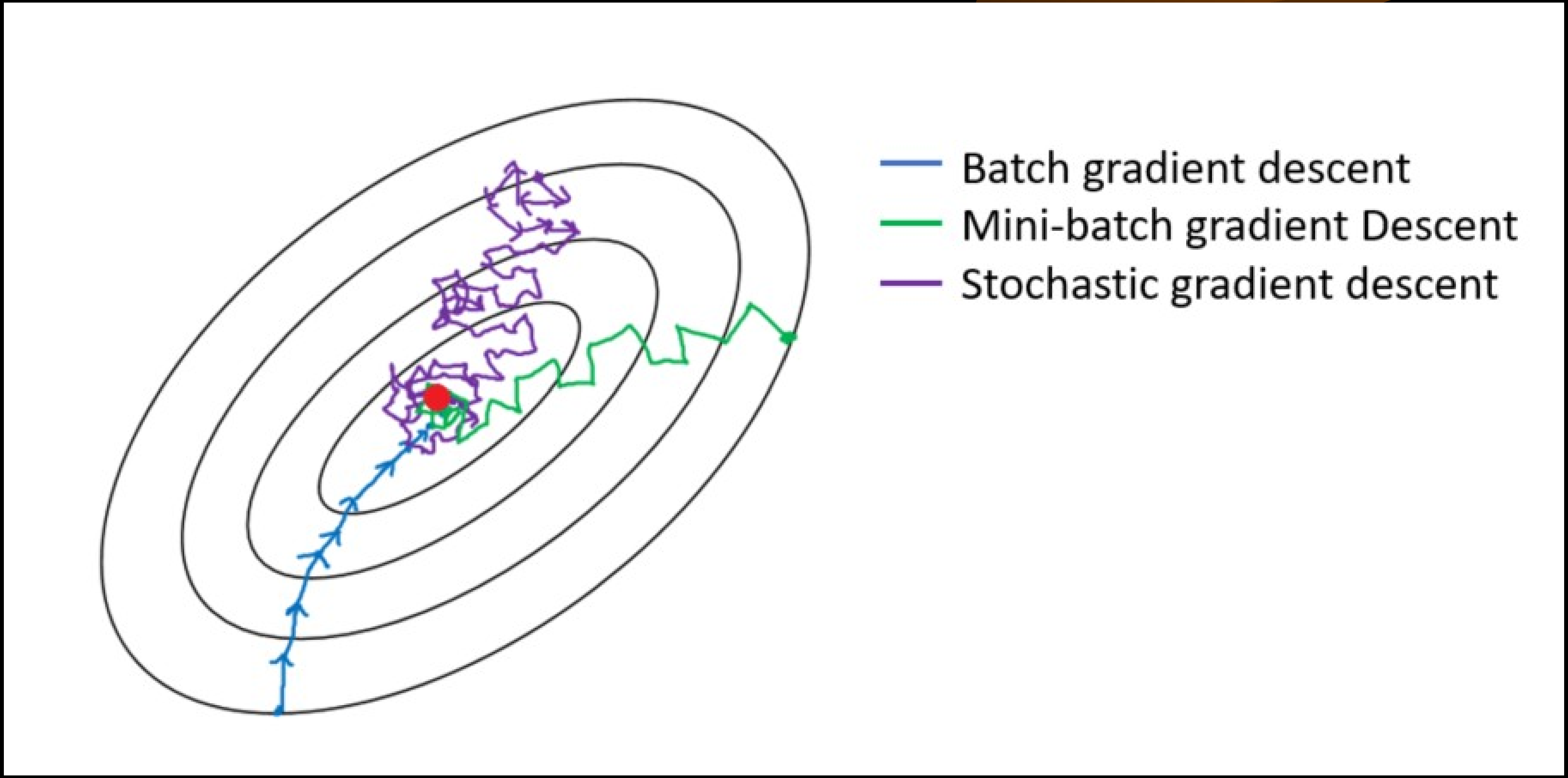
$$w_i \leftarrow w_i + \Delta w_i$$

Move the update into the inner loop

Stochastic gradient descent

- Batch: compute all deltas, then update weights
- Stochastic: for each pattern, compute delta and update weights
- Minibatch: select a small subset of patterns, compute their deltas, then update weights

Stochastic gradient descent



Training multiple networks

- Training is partially random (initial weights, stochastic descent)
- Train multiple networks, keep the best

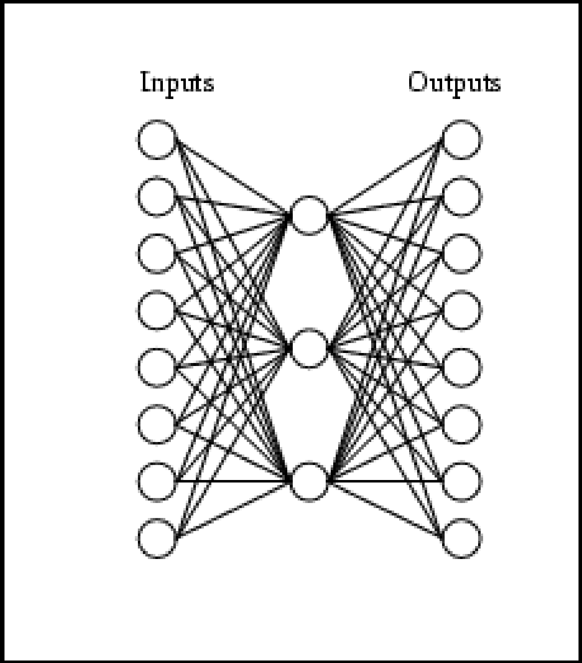
Representational Power of FeedForward ANNs

- Boolean functions: exactly with two layers and enough hidden neurons
- Continuous functions: bounded functions can be approximated with arbitrarily small error with two layers (sigmoid hidden units and linear output units)
- Arbitrary functions: can be approximated to arbitrary accuracy with three layers (two hidden layers with sigmoid units plus linear output units)

Representational Power of FeedForward ANNs

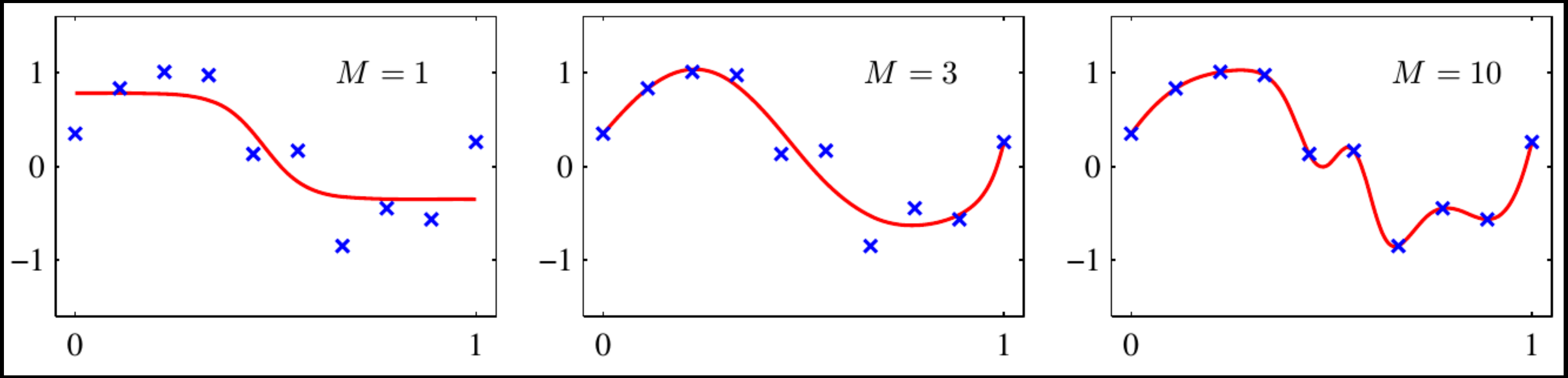
- Hypothesis Space search and inductive Bias
  - Hypothesis Space: *n*-dimensional Euclidean space of network weights
  - Inductive Bias: Smooth interpolation between data points
- Hidden Layer Representation
  - Encoding of information
  - Discovering of new features not explicit in the input representation

Hidden representation



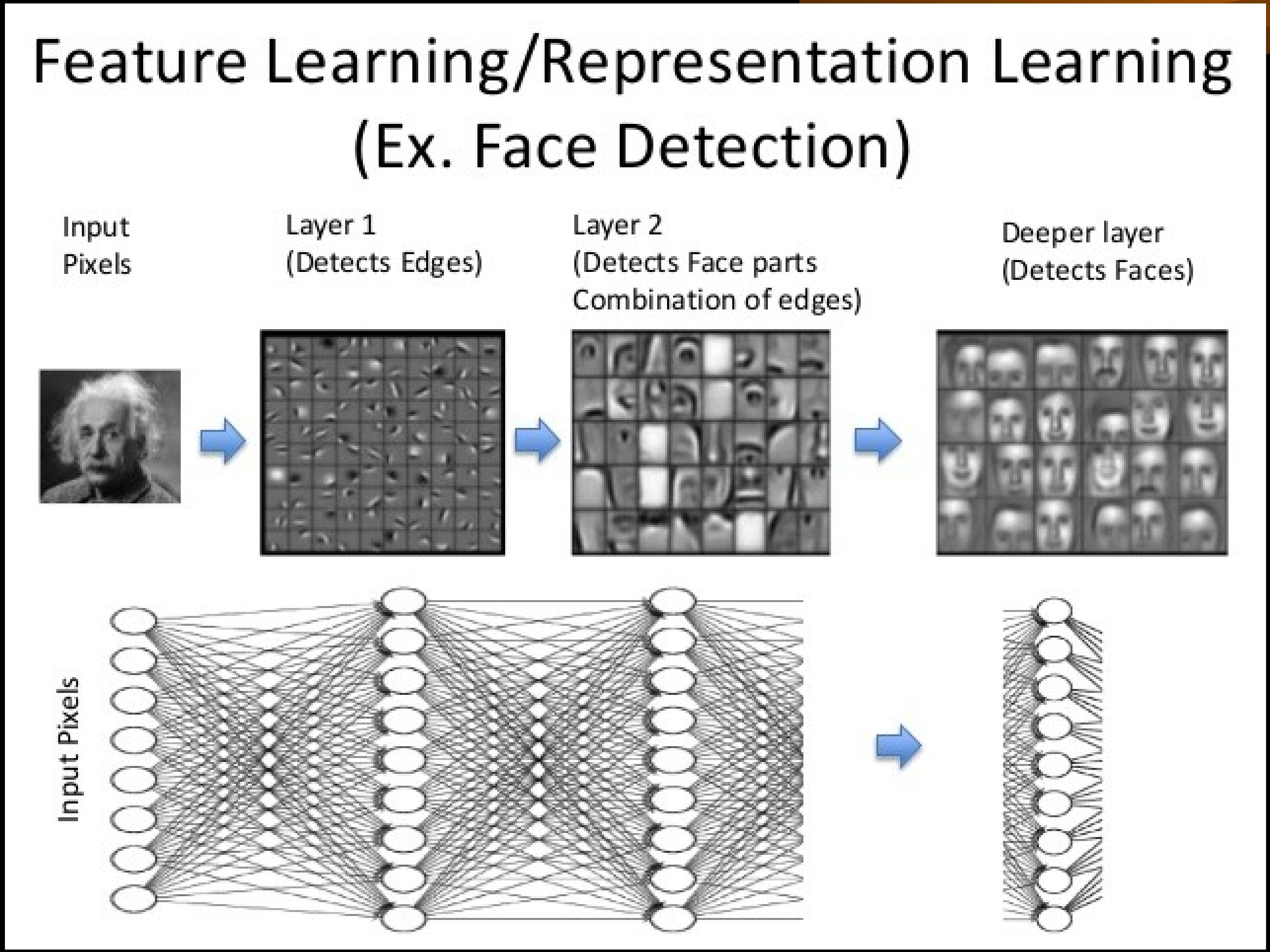
Input		Output
10000000	→	10000000
01000000	→	01000000
00100000	→	00100000
00010000	→	00010000
00001000	→	00001000
00000100	→	00000100
00000010	→	00000010
00000001	→	00000001

Hidden representation



M = # of hidden units

Hidden representation



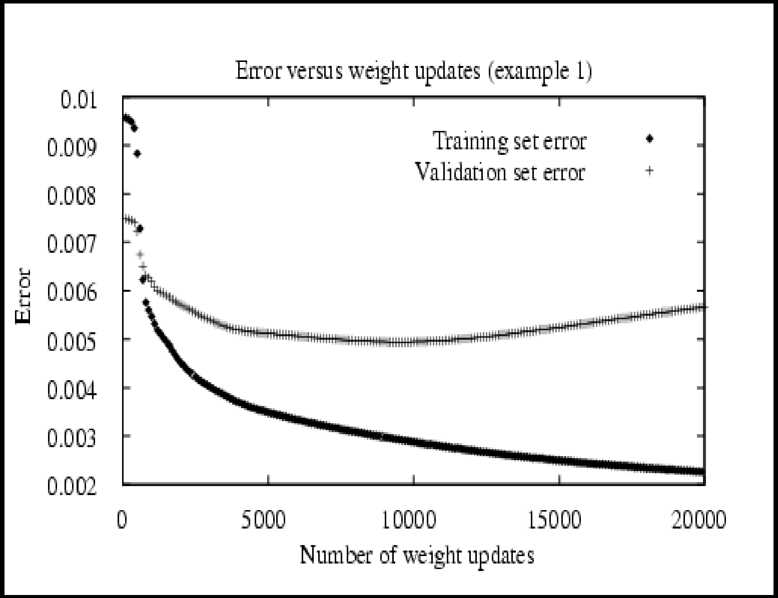
Generalization, Overfitting and Stopping Criterion

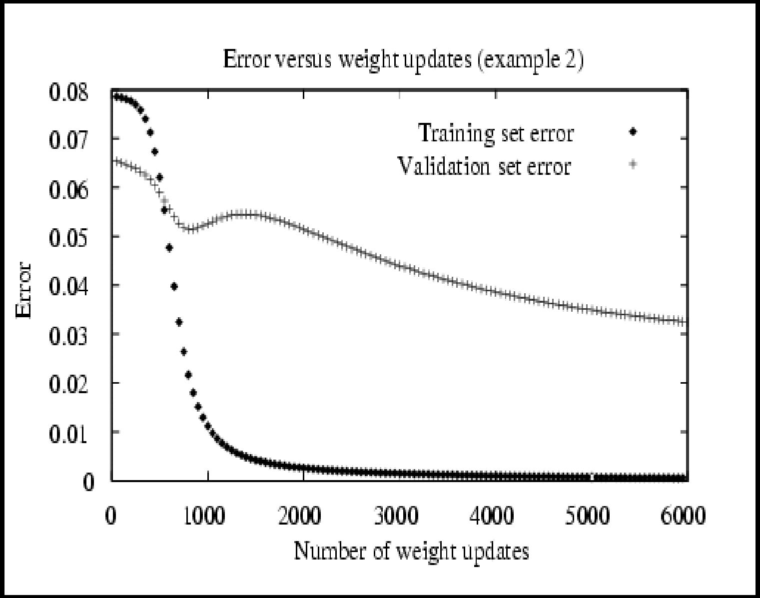
What is an appropriate condition for terminating the weight update loop?

– Hold-out Validation

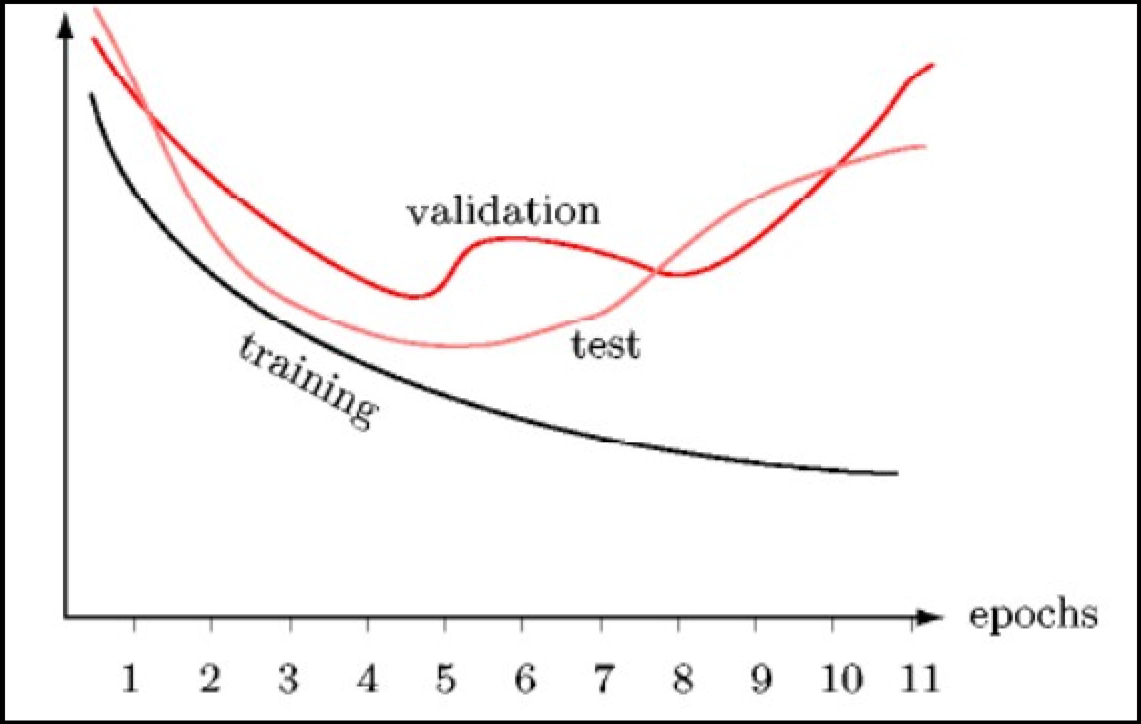
– *k*-fold Cross Validation

Overfitting in ANNs





Overfitting in ANNs

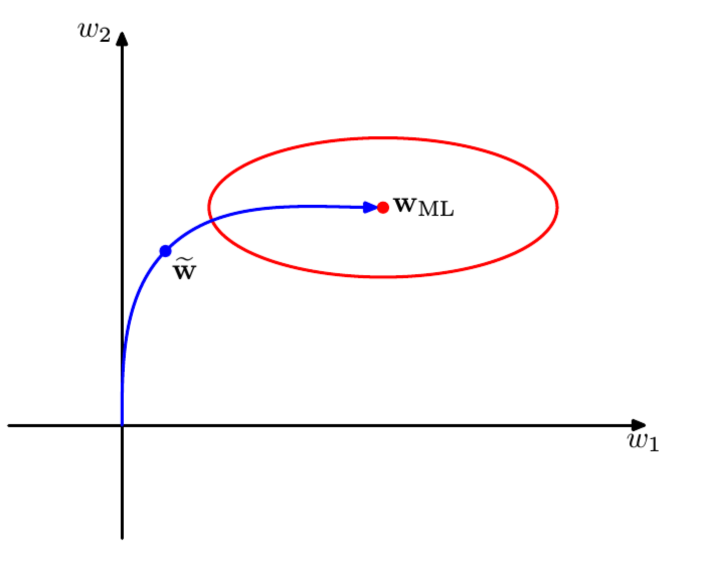


Regularization

E = cost + complexity

Weight Decay:

$$E(\boldsymbol{w}) = \frac{1}{2} \sum_D \sum_{k \in \text{outputs}} (t_k - o_k)^2 + \gamma \sum_{ij} (w_{ji})^2$$



Example: Face Recognition

– The Task

- Classifying camera images of faces of 20 different people, including 32 images per person, varying the person’s expression (happy, sad, angry, neutral), the direction in which they are looking (left, right, straight ahead, up), and whether or not they are wearing sunglasses
- There are also variation in the background behind the person, the clothing worn by the person and the position of the face within the image

Example: Face Recognition

- Each image has a 120x128 resolution, with pixels in a greyscale intensity from 0 (black) to 255 (white)

Task: Learning the direction in which the person is facing

– Design Choices
  - Input encoding: 30x32 coarse intensity values
  - Output encoding: 4 distinct output units

Example: Face Recognition

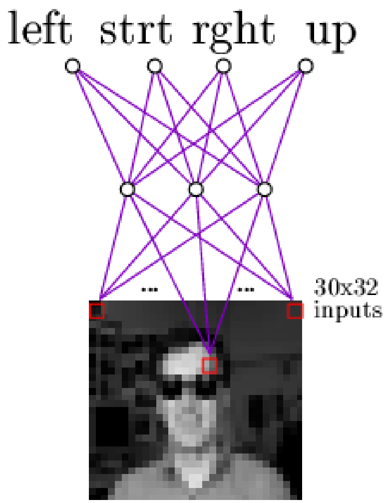
- Network structure: i : h: o

  i = 30x32 h = 3 to 30 o = 4

- Learning parameters:

  learning rate = 0.3 momentum = 0.9

Example: Face Recognition



Typical input images

Example: Face Recognition