Introduction to Neural Networks

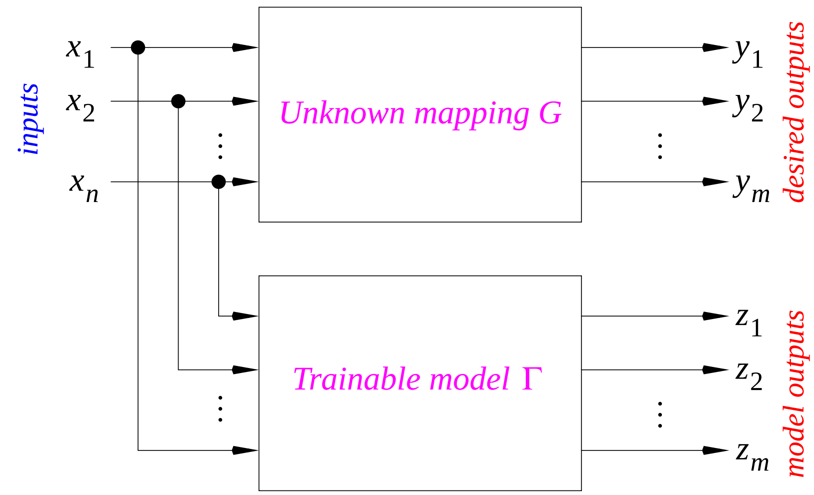
What are neural networks?

- Nonlinear function approximators

How do they relate to pattern recognition/classification?

- Nonlinear discriminant functions
- More complex decision boundaries than linear discriminant functions (e.g. Fisher, Gaussians with equal covariances)

Learning framework for NNs



Inputs/outputs

Definitions:

$\mathbf{y} = G(\mathbf{x})$ *(e.g discriminant function we want to learn)*

$\mathbf{x} = [x_1\ x_2\ \dots\ x_n]^T$ *(n inputs)*

$\mathbf{y} = [y_1\ y_2\ \dots\ y_m]^T$ *(m process outputs)*

Trainable model:

$\mathbf{z} = \Gamma(\mathbf{x}, \underset{\sim}{\mathbf{w}})$ *($\mathbf{w}$ = adjustable parameters)*

$\mathbf{z} = [z_1\ z_2\ \dots\ z_m]^T$ *(m model outputs)*

Learning goal:

Find $\mathbf{w}^*$ such that:

$$E(\mathbf{w}^*) \le E(\mathbf{w}),\ \forall \mathbf{w},$$

where $E(\mathbf{w})$ = error between G and Γ.

What should $E(\mathbf{w})$ be?

Error function (ideal)

Ideally,

$$E(\mathbf{w}) = \int_{\mathbf{x}} \|\mathbf{y} - \mathbf{z}\|^2 p(\mathbf{x}) d\mathbf{x}$$

How to compute?

Error function (practical)

Input/output data: p **input-output training patterns**

$$\begin{bmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \dots & \mathbf{x}_p \\ \mathbf{y}_1 & \mathbf{y}_2 & \dots & \mathbf{y}_p \end{bmatrix}$$

$$\mathbf{x}_i = \begin{bmatrix} x_{i1} & x_{i2} & \dots & x_{in} \end{bmatrix}^T$$

$$\mathbf{y}_i = \begin{bmatrix} y_{i1} & y_{i2} & \dots & y_{im} \end{bmatrix}^T, \ \mathbf{z}_i \equiv \Gamma(\mathbf{x}_i, \mathbf{w}).$$

$$E(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{p} \|\mathbf{y}_i - \mathbf{z}_i\|^2 = \frac{1}{2}\sum_{i=1}^{p}\sum_{j=1}^{m} (y_{ij} - z_{ij})^2$$

Artificial neural networks (NNs)

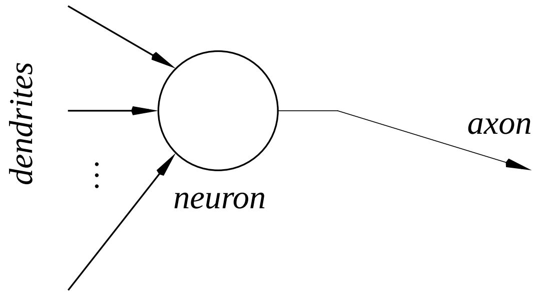
Neural networks are one type of parametric model Γ **.**

- Nonlinear function approximators
- Adjustable (trainable) parameters $\mathbf{w}$ (weights)
- Map inputs to outputs

Why "Neural Network?"

Biological inspiration

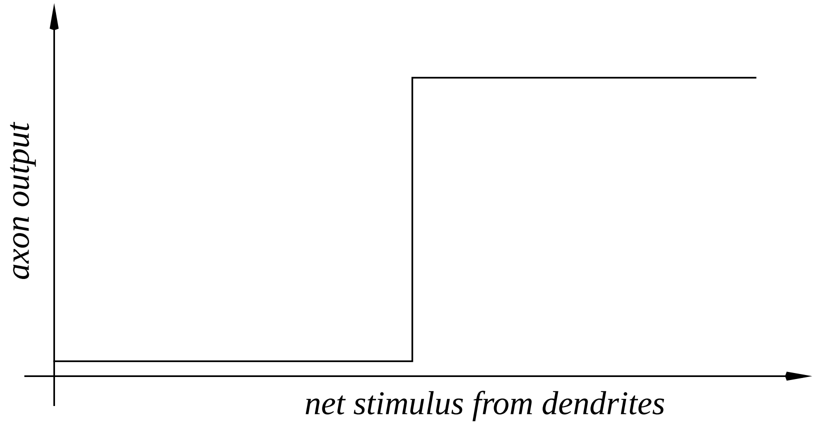
- Structure and function loosely based on *biological* neural networks (e.g. brain).
- Relatively simple building blocks connected together in massive and parallel network.



What does a neuron do?

Neuron transfer function

Rough approximation: threshold function



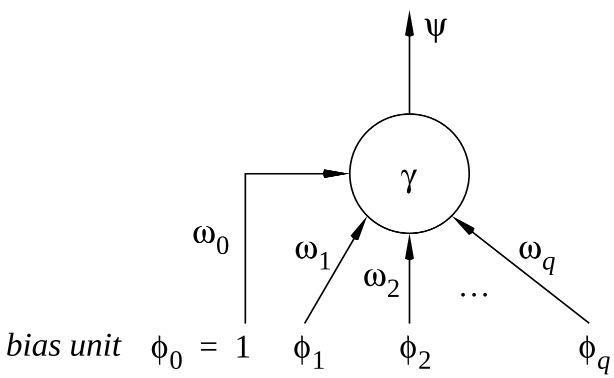
Neural networks: crude emulation of biology

- Simple basic building blocks.
- Individual units are connected massively and in parallel.
- Individual units have threshold-type activation functions.
- Learning through adjustment of the strength of connection (weights) between individual units

Caveat**: Artificial neural networks are much, much, much simpler than biological systems. Example: Human brain:**

- 10^{10} neurons
- 10^{12} connections

Basic building blocks of neural networks



Basic building block: the unit

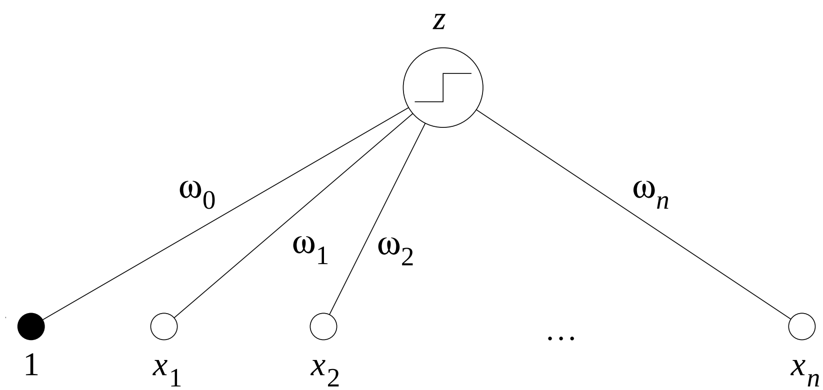
$\underset{\sim}{\phi} \equiv [\phi_0 \ \phi_1 \ \dots \ \phi_q]^T$ (scalar inputs)

$\mathbf{w} \equiv [\omega_0 \ \omega_1 \ \dots \ \omega_q]^T$ (weights)

γ = nonlinear activation function

$\psi \equiv \gamma(\mathbf{w} \cdot \underset{\sim}{\phi}) = \gamma\left(\sum_{i=0}^{q} \omega_i \phi_i\right)$ (output)

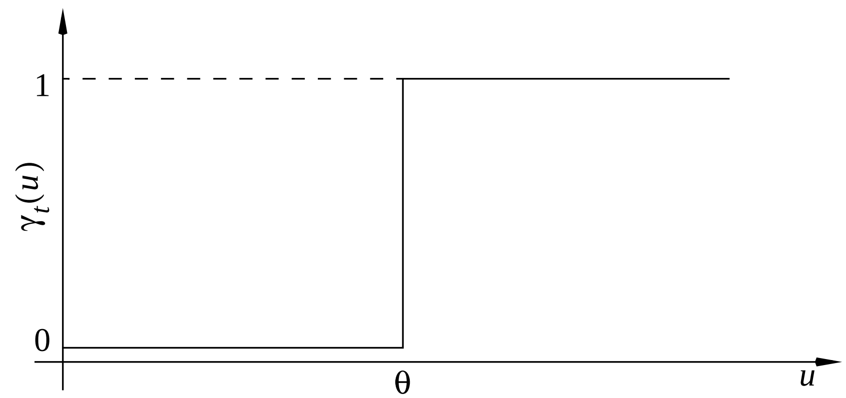
Perceptrons: the simplest "neural network"



What is this?

Threshold activation function

$$\gamma_t(u) = \begin{cases} 1 & u \geq \theta \\ 0 & u < \theta \end{cases}$$



Perceptron output

Perceptron mapping:

$$z = \begin{cases} 1 & \mathbf{w}^t\mathbf{x} \geq 0 \\ 0 & \mathbf{w}^t\mathbf{x} < 0 \end{cases}$$

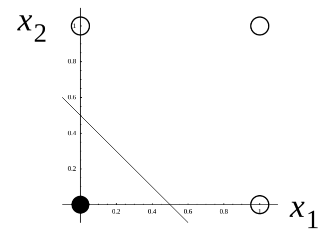
where,

$$\mathbf{x} = \begin{bmatrix} 1 & x_1 & \dots & x_n \end{bmatrix}^T$$

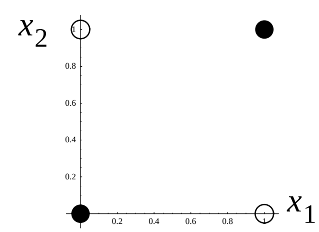
$$\mathbf{w} = \begin{bmatrix} \omega_0 & \omega_1 & \dots & \omega_n \end{bmatrix}^T$$

Limited mapping capability

$$\omega_0 = -0.5$$
$$\omega_1 = 1$$
$$\omega_2 = 1$$



OR function

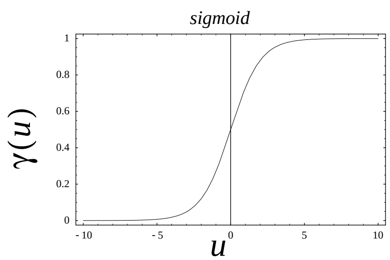


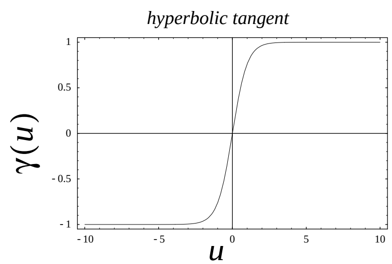
XOR function

More general networks: activation function

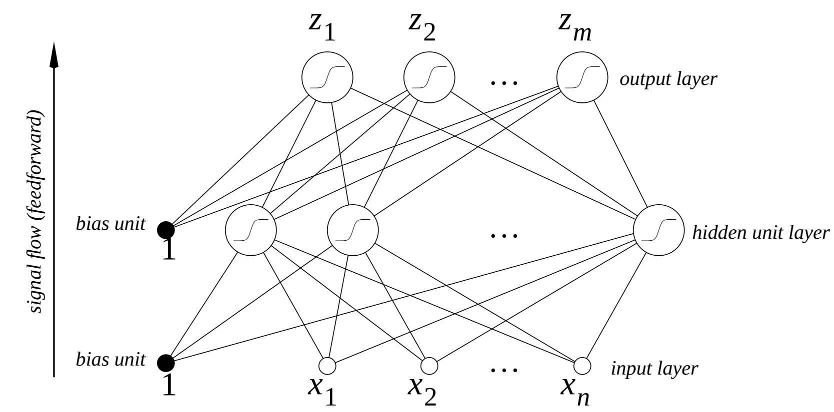
$$\gamma(u) = \frac{1}{1+e^{-u}} \quad \text{(sigmoid)}$$

$$\gamma(u) = \frac{e^{u}-e^{-u}}{e^{u}+e^{-u}} \quad \text{(hyperbolic tangent)}$$

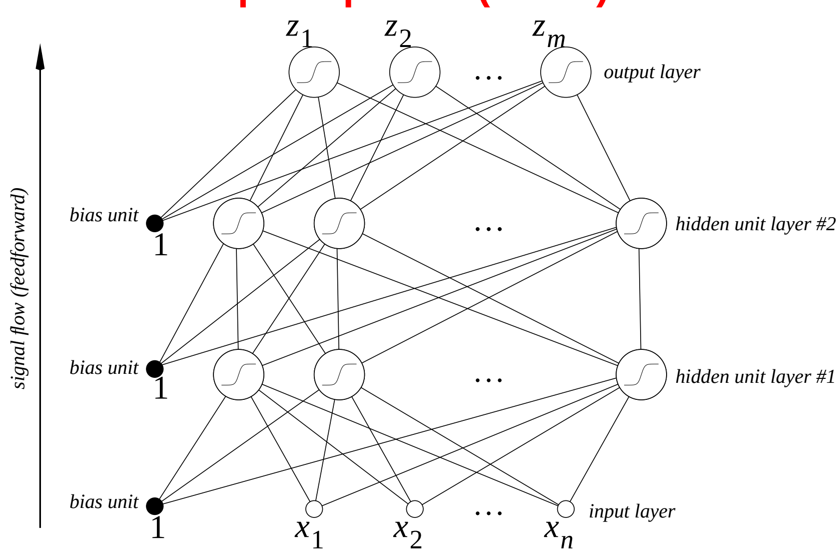




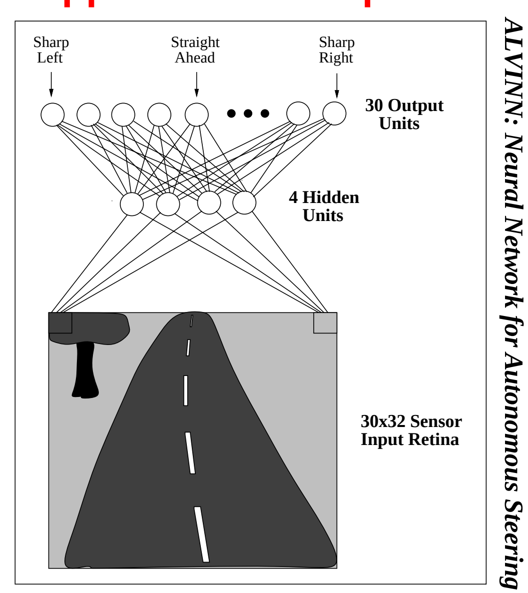
More general networks: multilayer perceptrons (MLPs)



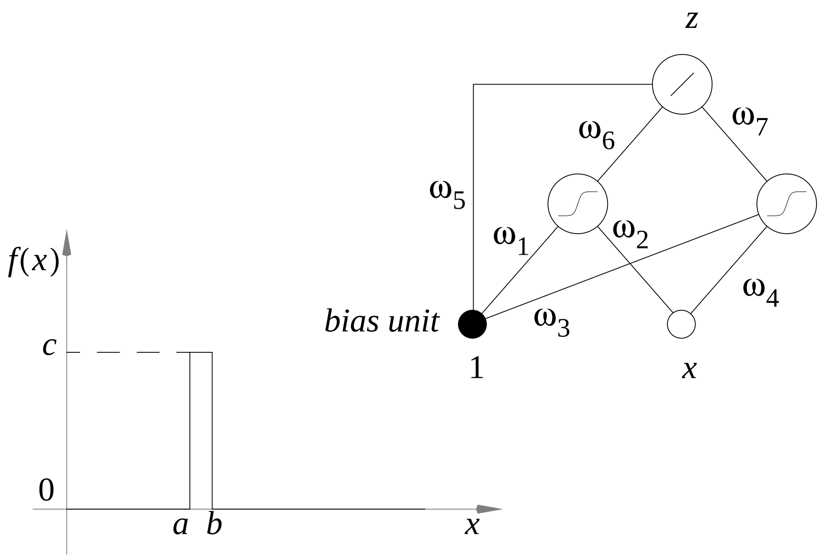
More general networks: multilayer perceptrons (MLPs)



MLP application example: ALVINN



A simple example



Derivation of function *f*(x)

$$f(x) = c[\gamma_t(x-a) - \gamma_t(x-b)]$$

$$f(x) = c\gamma_t(x-a) - c\gamma_t(x-b)$$

$$\gamma_t(u) \to \gamma(ku) \text{ as } k \to \infty$$

$$f(x) \approx c\gamma[k(x-a)] - c\gamma[k(x-b)] \text{ for large } k.$$

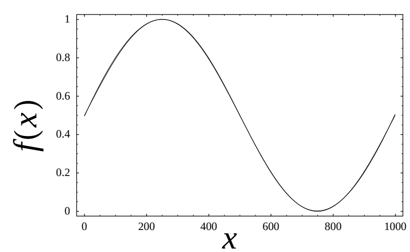
$$z = \omega_5 + \omega_6\gamma(\omega_1 + \omega_2 x) + \omega_7\gamma(\omega_3 + \omega_4 x)$$

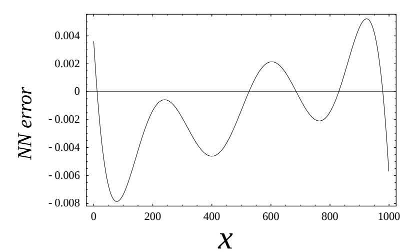
Weight values for simple example

	ω_1	ω_2	ω_3	ω_4	ω_5	ω_6	ω_7
set #1	$-kb$	k	$-ka$	k	0	$-c$	c
set #2	$-ka$	k	$-kb$	k	0	c	$-c$

Some theoretical properties of NNs

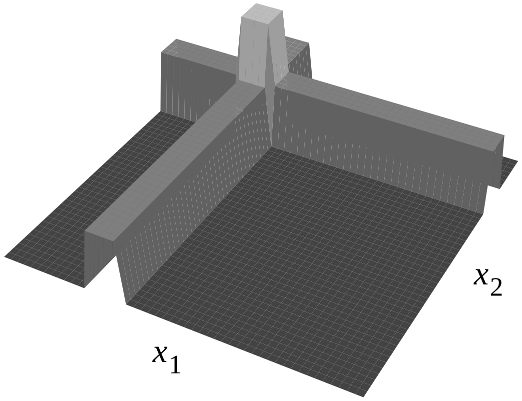
Single-input functions: what does the previous example say about single-input functions?





Multi-input functions: universal function approximator?

Does the single-input example hold in general?



Neural networks in practice: 3 basic steps

1. **Collect input/output training data.**
2. **Select an appropriate neural network architecture:**

- Number of hidden layers
- Number of hidden units in each layer.

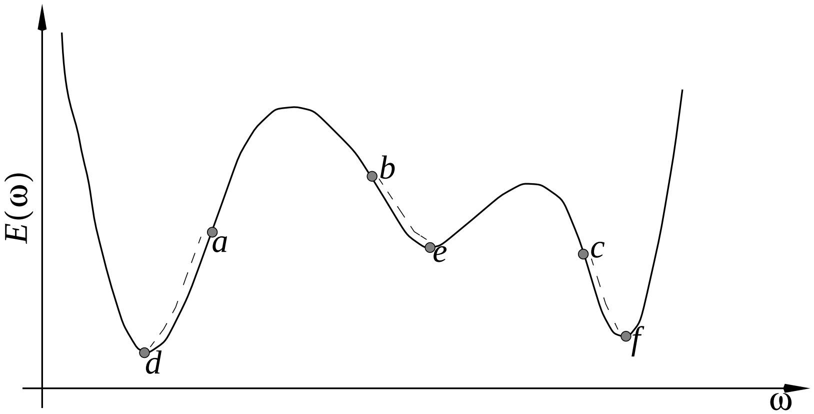
3. **Train (adjust) the weights of the neural network to minimize the error measure,**

$$E = \frac{1}{2}\sum_{i=1}^{p}\|\mathbf{y}_i - \mathbf{z}_i\|^2$$

Neural network training

Key problem: How to adjust w to minimize E?

Answer: use derivative information on error surface.



Gradient descent (one parameter)

1. **Initialize ω to some random initial value.**
2. **Change ω iteratively at step t according to:**

$$\omega(t+1) = \omega(t) - \eta\frac{dE}{d\omega(t)}$$

Implies local, not global minimum...

General gradient descent

1. Initialize $\mathbf{w}$ to some random initial value.

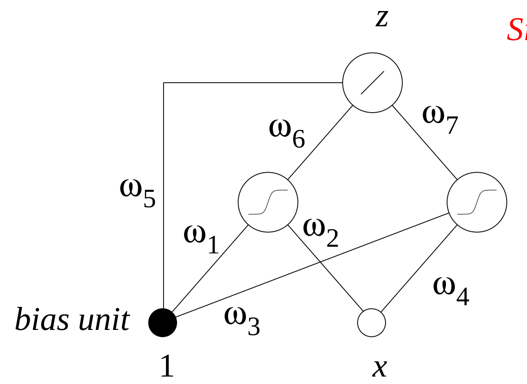
2. Change $\mathbf{w}$ iteratively at step t according to:

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \eta \nabla E[\mathbf{w}(t)]$$

$$\nabla E[\mathbf{w}(t)] = \left[\frac{\partial E}{\partial \omega_1(t)} \ \frac{\partial E}{\partial \omega_2(t)} \ \cdots \ \frac{\partial E}{\partial \omega_q(t)}\right]^T$$

Simple example of gradient computation

Compute $\dfrac{\partial E}{\partial \omega_4}$ for the neural network below:



Single training pattern

$$\langle x, y \rangle$$

$$E = \frac{1}{2}(y-z)^2$$

Derivation

Generalization to multiple training patterns:

$$\frac{\partial E}{\partial \omega_j} = \frac{\partial}{\partial \omega_j}\left[\frac{1}{2}\sum_{i=1}^{p}(y_i - z_i)^2\right] = \sum_{i=1}^{p}\frac{\partial}{\partial \omega_j}\left[\frac{1}{2}(y_i - z_i)^2\right].$$

Derivation

$$net_1 \equiv \omega_1 + \omega_2 x$$

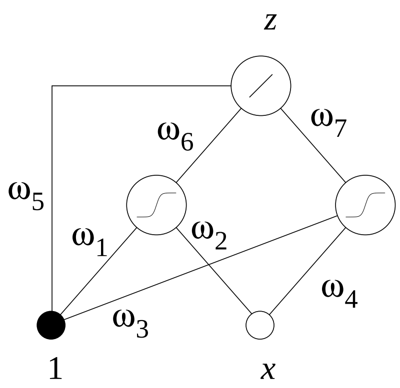
$$net_2 \equiv \omega_3 + \omega_4 x$$

$$h_1 \equiv \gamma(net_1)$$

$$h_2 \equiv \gamma(net_2)$$

$$z = \omega_5 + \omega_6 h_1 + \omega_7 h_2$$

$$\frac{\partial E}{\partial \omega_4} = -(y-z)\frac{\partial z}{\partial \omega_4}$$



$$\frac{\partial E}{\partial \omega_4} = (z-y)\left(\frac{\partial z}{\partial h_2}\right)\left(\frac{\partial h_2}{\partial net_2}\right)\left(\frac{\partial net_2}{\partial \omega_4}\right)$$

Derivation

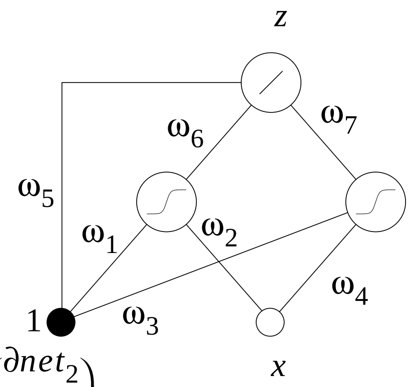
$net_1 \equiv \omega_1 + \omega_2 x$

$net_2 \equiv \omega_3 + \omega_4 x$

$h_1 \equiv \gamma(net_1)$

$h_2 \equiv \gamma(net_2)$

$z = \omega_5 + \omega_6 h_1 + \omega_7 h_2$



$$\frac{\partial E}{\partial \omega_4} = (z-y)\left(\frac{\partial z}{\partial h_2}\right)\left(\frac{\partial h_2}{\partial net_2}\right)\left(\frac{\partial net_2}{\partial \omega_4}\right)$$

$$\frac{\partial E}{\partial \omega_4} = (z-y)\omega_7\gamma'(net_2)x$$

Generalization: Backpropagation

Key problem: Generalize specific result to compute derivatives in more general manner.

Answer: *Backpropagation algorithm* [Rumelhart and McClelland,1986].

- Efficient, algorithmic formulation for computing error derivatives
- Gradient computation without hardcoding derivatives (allows on-the-fly adjustment of NN architectures).

Backpropagation derivation

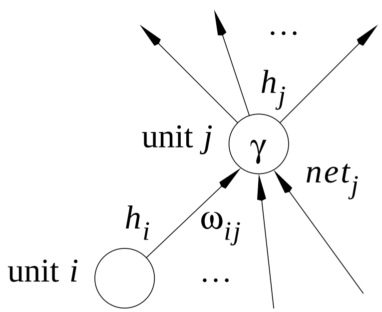
$h_j \equiv \gamma(net_j)$

$net_j \equiv \sum_k h_k \omega_{kj}$

$$\frac{\partial E}{\partial \omega_{ij}} = \left(\frac{\partial E}{\partial net_j}\right)\left(\frac{\partial net_j}{\partial \omega_{ij}}\right)$$

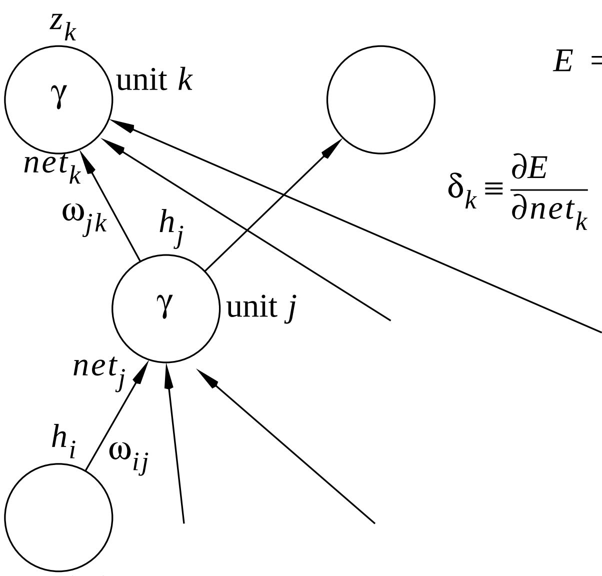
$$\frac{\partial net_j}{\partial \omega_{ij}} = h_i$$

$$\delta_j \equiv \frac{\partial E}{\partial net_j}$$



$$\frac{\partial E}{\partial \omega_{ij}} = \delta_j h_i$$

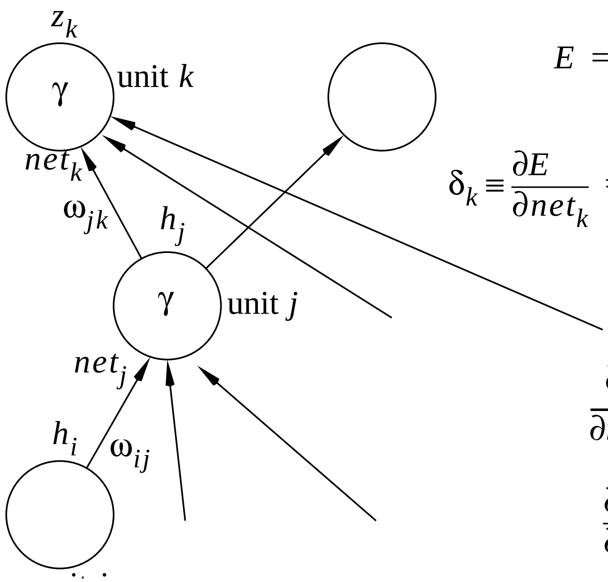
Backpropagation derivation: output units



$$E = \frac{1}{2}\sum_{l=1}^{m}(y_l - z_l)^2$$

$$\delta_k \equiv \frac{\partial E}{\partial net_k} = \left(\frac{\partial E}{\partial z_k}\right)\left(\frac{\partial z_k}{\partial net_k}\right)$$

Backpropagation derivation: output units



$$E = \frac{1}{2}\sum_{l=1}^{m}(y_l - z_l)^2$$

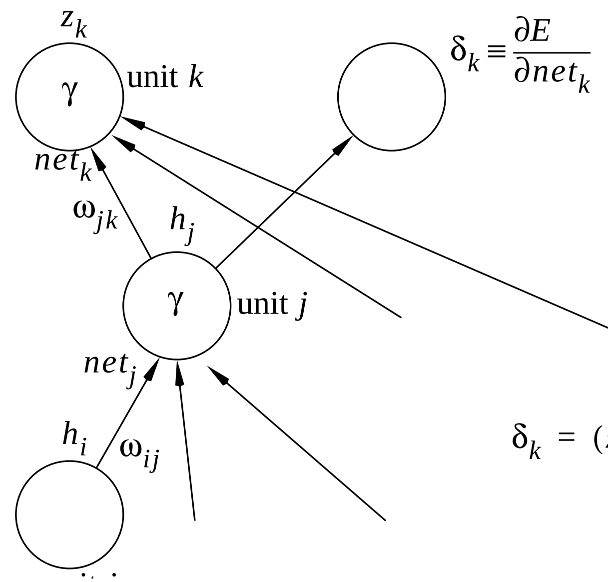
$$\delta_k \equiv \frac{\partial E}{\partial net_k} = \left(\frac{\partial E}{\partial z_k}\right)\left(\frac{\partial z_k}{\partial net_k}\right)$$

$$z_k = \gamma(net_k)$$

$$\frac{\partial z_k}{\partial net_k} = \gamma'(net_k)$$

$$\frac{\partial E}{\partial z_k} = (z_k - y_k)$$

Backpropagation derivation:output units

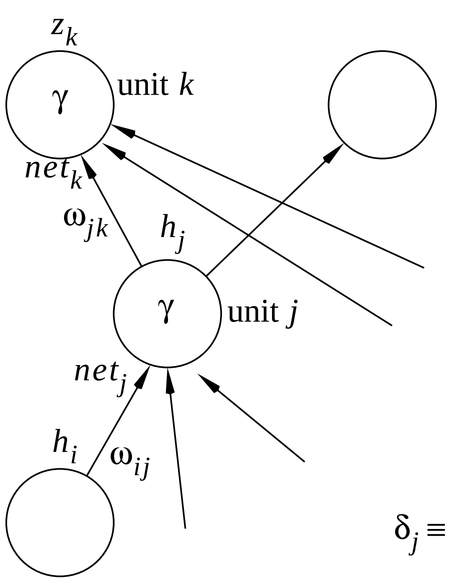


$$\delta_k \equiv \frac{\partial E}{\partial net_k} = \left(\frac{\partial E}{\partial z_k}\right)\left(\frac{\partial z_k}{\partial net_k}\right)$$

$$\delta_k = (z_k - y_k)\gamma'(net_k)$$

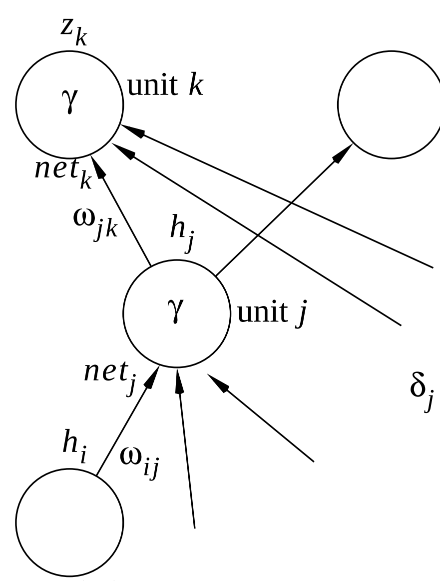
$$\frac{\partial E}{\partial \omega_{jk}} = \delta_k h_j$$

Backpropagation derivation: hidden units



$$\delta_j \equiv \frac{\partial E}{\partial net_j} = \sum_l \left(\frac{\partial E}{\partial net_l}\right)\left(\frac{\partial net_l}{\partial net_j}\right)$$

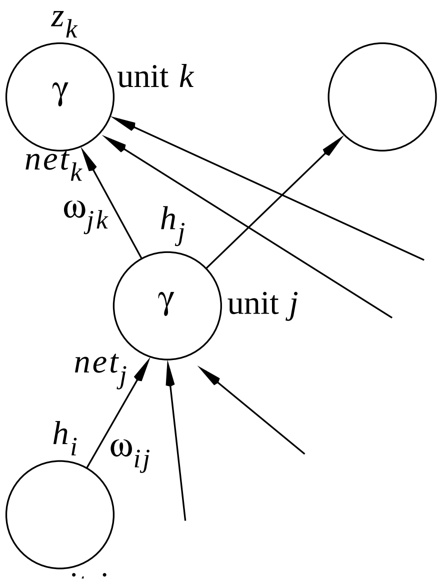
Backpropagation derivation: hidden units



$$\delta_j \equiv \frac{\partial E}{\partial net_j} = \sum_l \left(\frac{\partial E}{\partial net_l}\right)\left(\frac{\partial net_l}{\partial net_j}\right)$$

$$\delta_j = \sum_l \delta_l \left(\frac{\partial net_l}{\partial net_j}\right)$$

Backpropagation derivation: hidden units



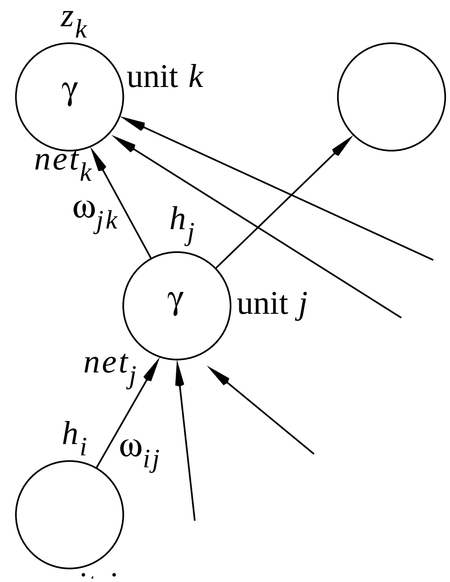
$$\delta_j = \sum_l \delta_l \left(\frac{\partial net_l}{\partial net_j} \right)$$

$$net_l = \sum_s \omega_{sl} \gamma(net_s)$$

$$\frac{\partial net_l}{\partial net_j} = \omega_{jl} \gamma'(net_j)$$

$$\delta_j = \sum_l \delta_l \omega_{jl} \gamma'(net_j)$$

Backpropagation derivation: hidden units



$$\delta_j = \left(\sum_l \delta_l \omega_{jl} \right) \gamma'(net_j)$$

$$\frac{\partial E}{\partial \omega_{ij}} = \delta_j h_i$$

Backpropagation summary

Output units:

$$\frac{\partial E}{\partial \omega_{jk}} = \delta_k h_j$$

$$\delta_k = (z_k - y_k) \gamma'(net_k)$$

Hidden units:

$$\frac{\partial E}{\partial \omega_{ij}} = \delta_j h_i$$

$$\delta_j = \left(\sum_l \delta_l \omega_{jl} \right) \gamma'(net_j)$$

Basic steps in using neural networks

1. **Collect training data**
2. **Preprocess training data**
3. **Select neural network architecture**
4. **Select learning algorithm**
5. **Weight initialization**

6. **Forward pass**
7. **Backward pass**
8. **Repeat steps 6 and 7 until satisfactory model is reached.**

The Forward Pass

1. Apply an input vector $\mathbf{x}_i$ to network.
2. Compute the net input to each hidden unit (net_j).
3. Compute the hidden-unit outputs (h_j),
4. Compute the neural network outputs (z_k).

The Backward Pass

1. Evaluate δ_k at the outputs, where,

   $$\delta_k = \partial E / \partial net_k$$

   for each output unit k.
2. Backpropagate the δ values from the outputs backwards through the neural network.
3. Compute $\partial E / \partial \omega_i$.
4. Update weights based on the computed gradient,

   $$\mathbf{w}(t+1) = \mathbf{w}(t) - \eta \nabla E[\mathbf{w}(t)].$$

Practical issues

1. What should your training data be?

- Sufficient training data?
- Biased training data?
- Deterministic/stochastic task?
- Stationary/non-stationary?

2. What should your neural network architecture be?
3. Preprocessing of data.
4. Weight initialization — why small, random values?

Practical issues (continued)

5. Selecting the learning parameter

In gradient descent:

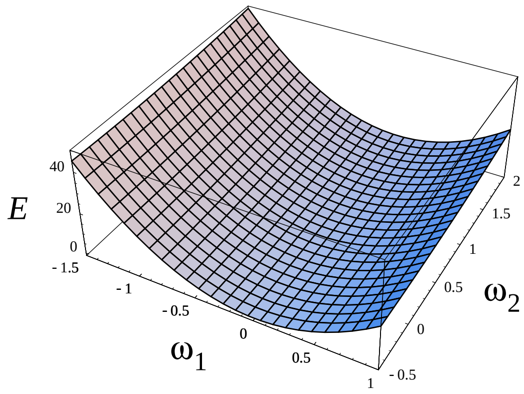
$$\mathbf{w}(t+1) = \mathbf{w}(t) - \eta \nabla E[\mathbf{w}(t)]$$

what should η be?

Difficult question to answer...

Selecting the learning parameter: an example

Sample error surface: $E = 20\omega_1^2 + \omega_2^2$ **(realistic?)**



Selecting the learning parameter: an example

Where is the minimum of this "error surface?"

$$E = 20\omega_1^2 + \omega_2^2$$

How many steps to convergence? ($\sqrt{E} < 10^{-6}$)

- Different initial weights
- Different learning rates

Deriving the gradient descent equations

$$E = 20\omega_1^2 + \omega_2^2$$

Gradient?

$$\frac{\partial E}{\partial \omega_1} = 40\omega_1 \qquad \frac{\partial E}{\partial \omega_2} = 2\omega_2$$

Gradient descent?

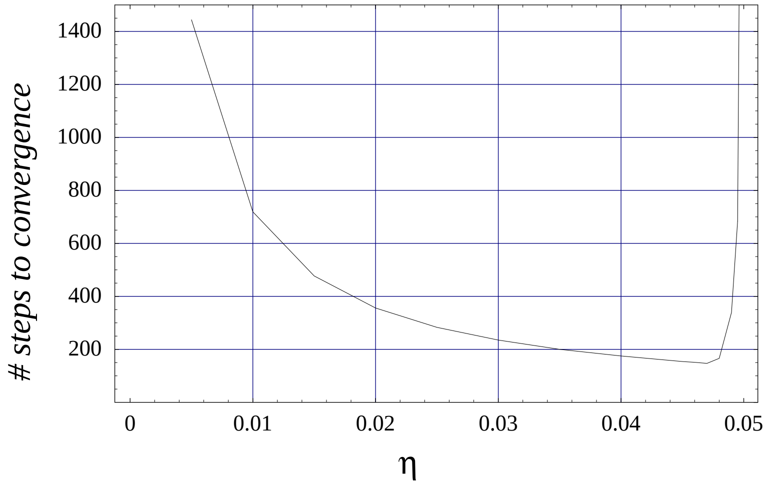
$$\omega_1(t+1) = \omega_1(t) - \eta\frac{\partial E}{\partial \omega_1(t)}$$

$$\omega_1(t+1) = \omega_1(t)(1-40\eta)$$

$$\omega_2(t+1) = \omega_2(t)(1-2\eta)$$

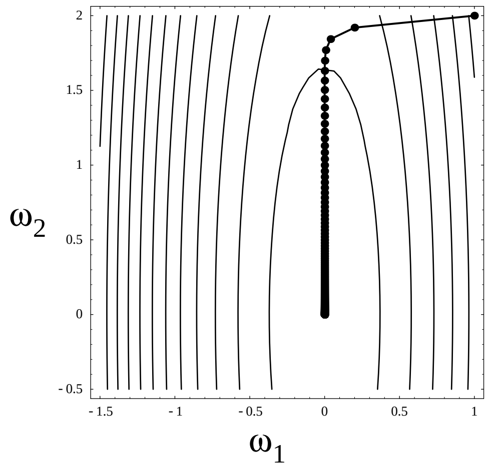
Convergence experiments

Initial weights: $(\omega_1, \omega_2) = (1, 2)$



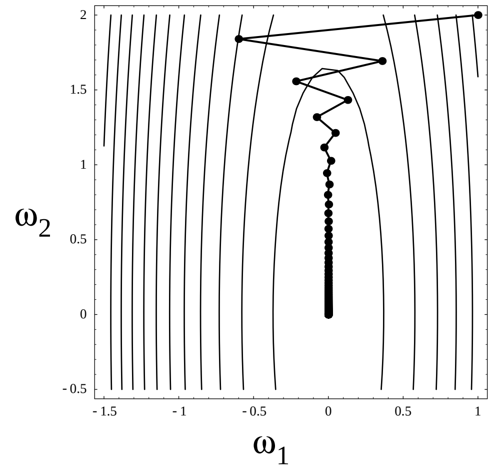
A closer look

$\eta = 0.02$



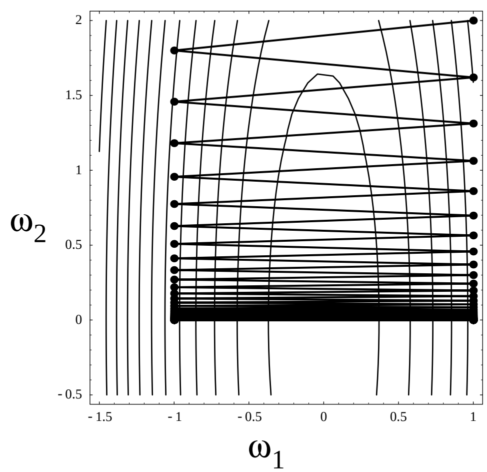
A closer look

$\eta = 0.04$



A closer look

$\eta = 0.05$



What happens at $\eta > 0.5$?

Gradient descent equations:

$$\omega_1(t+1) = \omega_1(t)(1 - 40\eta)$$

$$\omega_2(t+1) = \omega_2(t)(1 - 2\eta)$$

Similar to fixed-point iteration:

$$\omega(t+1) = c\omega(t)$$

- diverges for $\|c\| > 1$, $\omega(0) \neq 0$
- converges for $\|c\| < 1$.

Convergence of gradient descent equations

$$\omega_1(t+1) = \omega_1(t)(1-40\eta)$$

$$\omega_2(t+1) = \omega_2(t)(1-2\eta)$$

require that:

$\|1-40\eta\| < 1$

$-1 < 1-40\eta < 1$

$0 < \eta < 0.05$

Why not $\|1-2\eta\| < 1$ **?**

Learning rate discussion

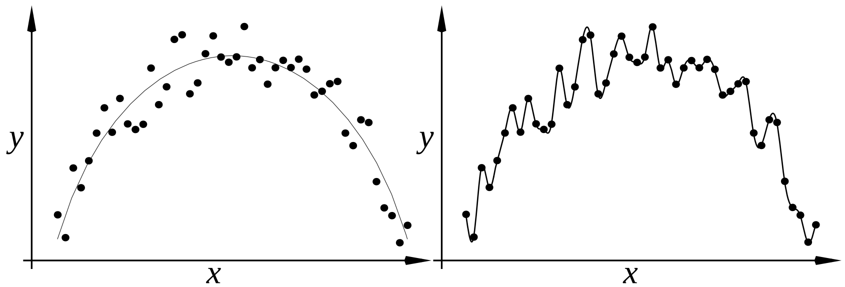
- Problematic error surfaces: "long, steep-sided valleys"
- If learning rate is too small, slow convergence. If learning rate is too large, possible divergence.
- Theoretical bounds not possible in general case (only for specific, trivial example).

Motivation for looking at more advanced training algorithms — doing more with the gradient information. Any thoughts?

Practical issues (continued)

6. Pattern vs. batch training

7. Good generalization



- Sufficiently constrained neural network architecture.
- Cross validation.

Good generalization: Two data sets