Lecture 11

Supervised Learning
Artificial Neural Networks

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

Slides by Stuart Russell and Peter Norvig

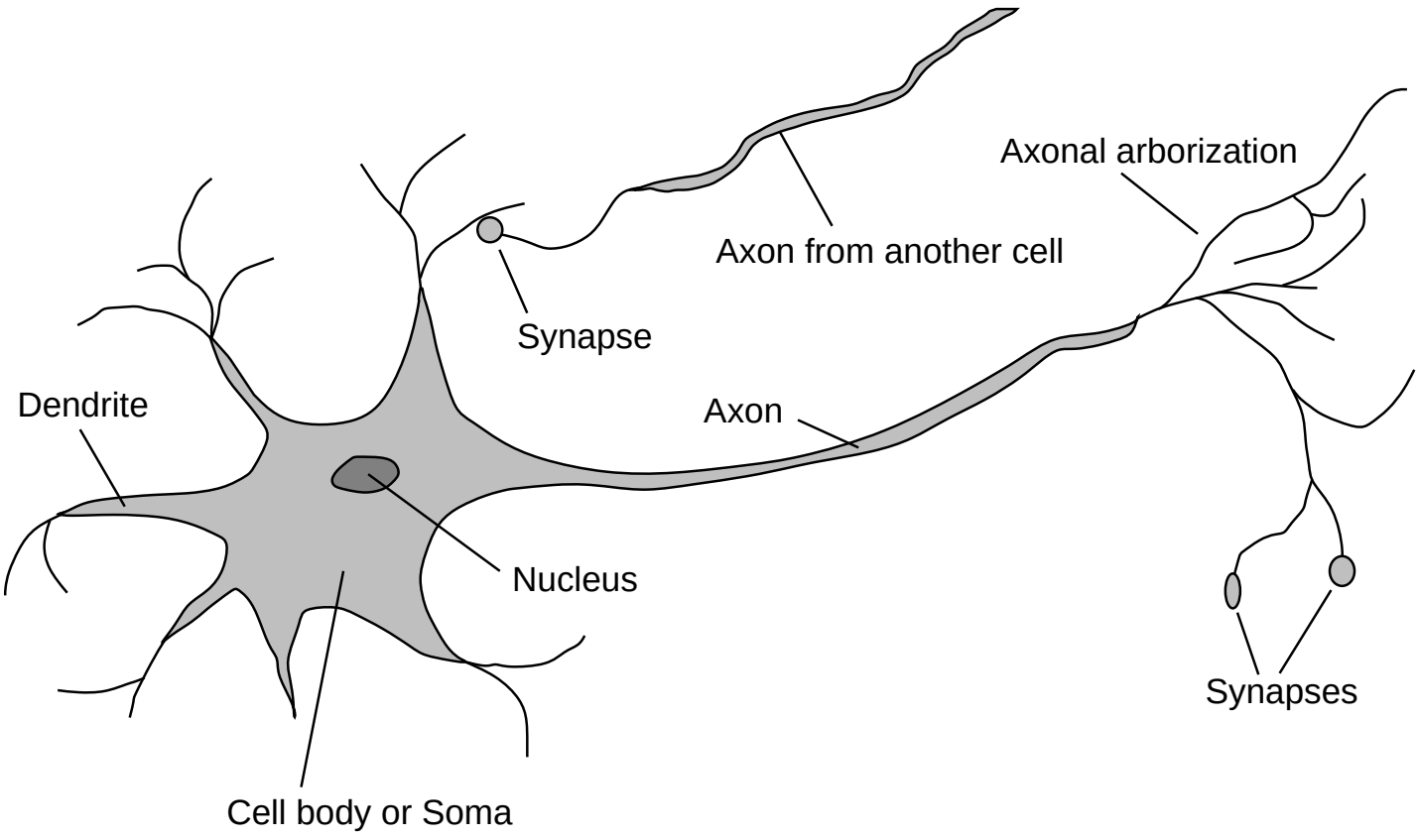
Course Overview

- ✔ Introduction
  - ✔ Artificial Intelligence
  - ✔ Intelligent Agents
- ✔ Search
  - ✔ Uninformed Search
  - ✔ Heuristic Search
- ✔ Uncertain knowledge and Reasoning
  - ✔ Probability and Bayesian approach
  - ✔ Bayesian Networks
  - ✔ Hidden Markov Chains
  - ✔ Kalman Filters

- Learning
  - Supervised
    Decision Trees, Neural Networks
    Learning Bayesian Networks
  - Unsupervised
    EM Algorithm
- Reinforcement Learning
- Games and Adversarial Search
  - Minimax search and Alpha-beta pruning
  - Multiagent search
- Knowledge representation and Reasoning
  - Propositional logic
  - First order logic
  - Inference
  - Plannning

Outline

A neuron in a living biological system



Signals are noisy "spike trains" of electrical potential

In the brain: > 20 types of neurons with 10^{14} synapses

	Brain	Computer
No. of processing units	$\approx 10^{11}$	$\approx 10^{9}$
Type of processing units	Neurons	Transistors
Type of calculation	massively parallel	usually serial
Data storage	associative	address-based
Switching time	$\approx 10^{-3}$s	$\approx 10^{-9}$s
Possible switching operations	$\approx 10^{13}\frac{1}{s}$	$\approx 10^{18}\frac{1}{s}$
Actual switching operations	$\approx 10^{12}\frac{1}{s}$	$\approx 10^{10}\frac{1}{s}$

(compare with world population $= 7 \times 10^9$)
Additionally, brain is parallel and reorganizing while computers are serial and static
Brain is fault tolerant: neurons can be destroyed.

Observations of neuroscience

- Neuroscientists: view brains as a web of clues to the biological mechanisms of cognition.
- Engineers: The brain is an example solution to the problem of cognitive computing

Applications

- supervised learning: regression and classification
- associative memory
- optimization:
- grammatical induction, (aka, grammatical inference)
  e.g. in natural language processing
- noise filtering
- simulation of biological brains

Artificial Neural Networks

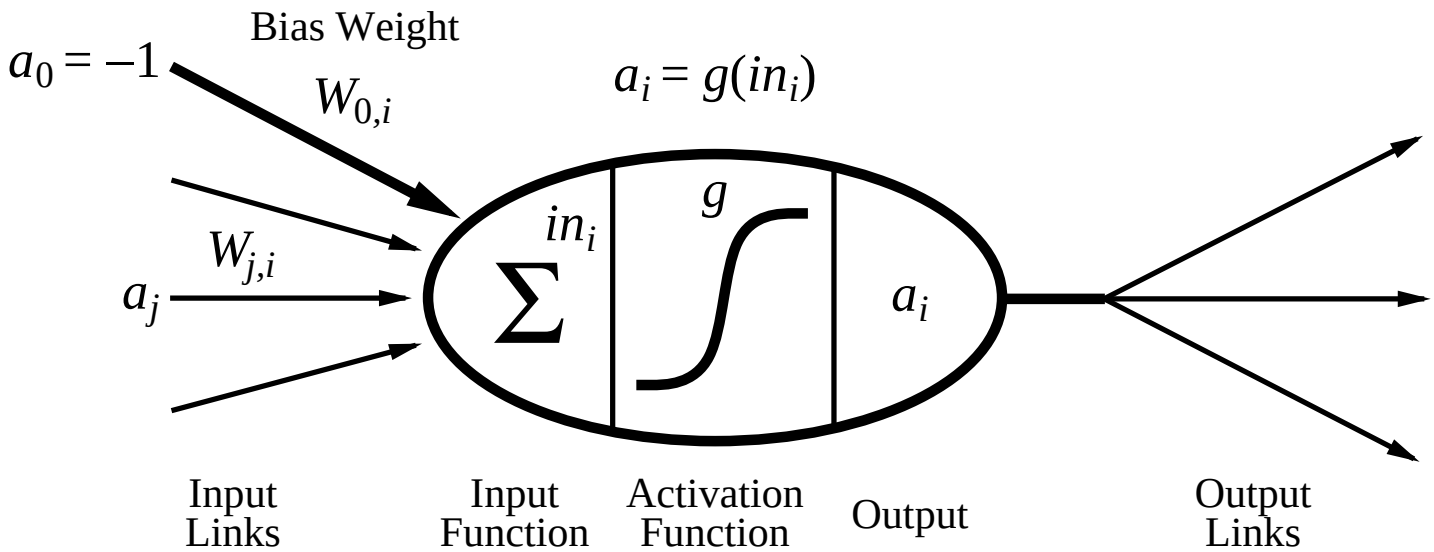
$\rightsquigarrow$ "**The** neural network" does not exist. There are different paradigms for neural networks, how they are trained and where they are used.

- Artificial Neuron
  - Each input is multiplied by a weighting factor.
  - Output is 1 if sum of weighted inputs exceeds the threshold value; 0 otherwise.
- Network is programmed by adjusting weights using feedback from examples.

McCulloch–Pitts "unit" (1943)

Output is a function of weighted inputs:

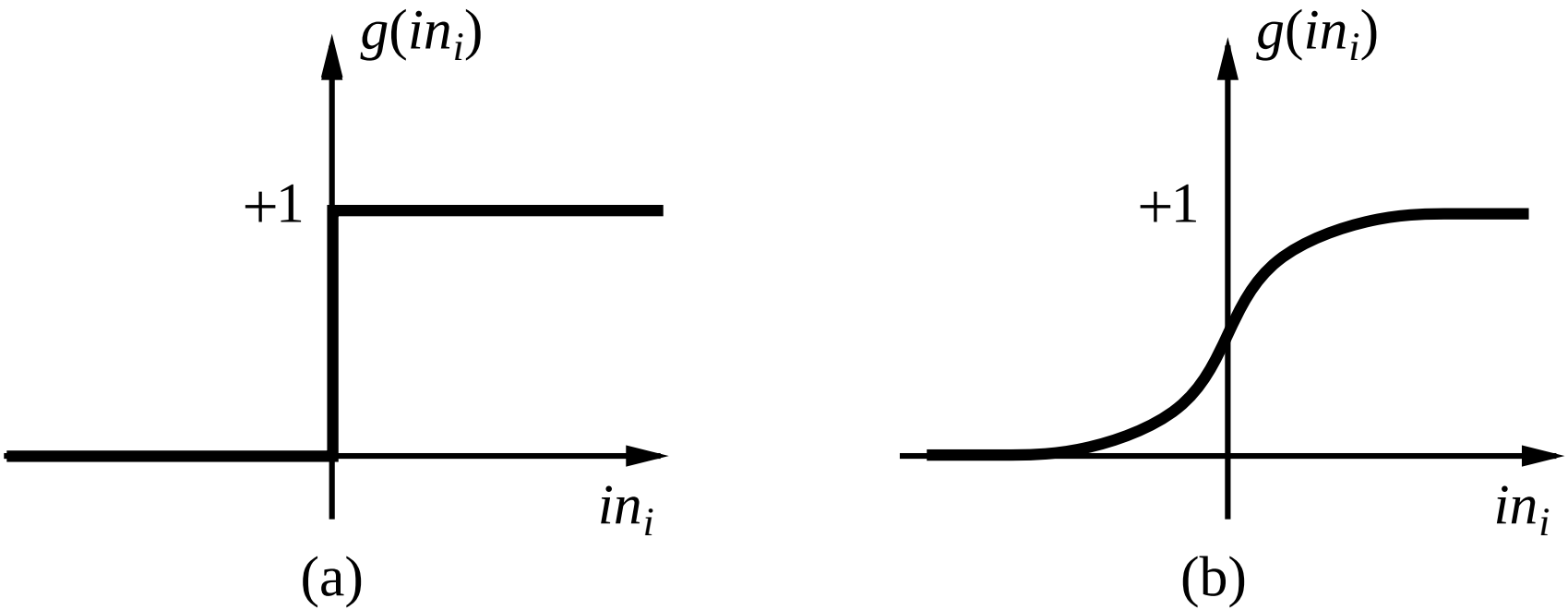
$$a_i = g(in_i) = g\left(\sum_j W_{j,i} a_j\right)$$



A gross oversimplification of real neurons, but its purpose is to develop understanding of what networks of simple units can do

Activation functions

Non linear activation functions

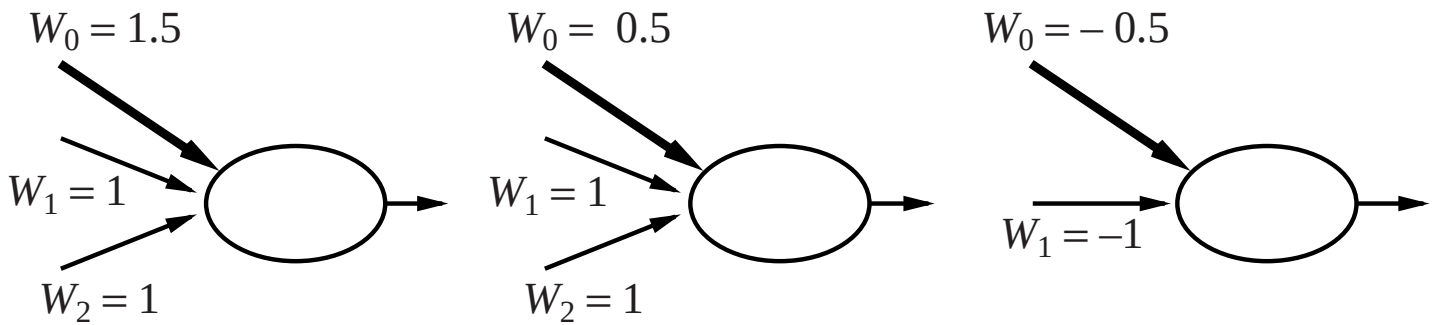


(a) is a step function or threshold function
(mostly used in theoretical studies)

(b) is a continuous activation function, e.g., sigmoid function $1/(1+e^{-x})$
(mostly used in practical applications)

Changing the bias weight $W_{0,i}$ moves the threshold location

Implementing logical functions



McCulloch and Pitts: every (basic) Boolean function can be implemented (eventually by connecting a large number of units in networks, possibly recurrent, of arbitrary depth)

Network structures

Architecture: definition of number of nodes and interconnection structures and activation functions g but not weights.

- Feed-forward networks:
  no cycles in the connection graph
  - single-layer perceptrons (no hidden layers)
  - multi-layer perceptrons (one or more hidden layers)

  Feed-forward networks implement functions, have no internal state

- Recurrent networks:
  - Hopfield networks have symmetric weights ($W_{i,j} = W_{j,i}$)
    $g(x) = \text{sign}(x)$, $a_i = \{1, 0\}$; **associative memory**
  - recurrent neural nets have directed cycles with delays
    $\Longrightarrow$ have internal state (like flip-flops), can oscillate etc.

Use

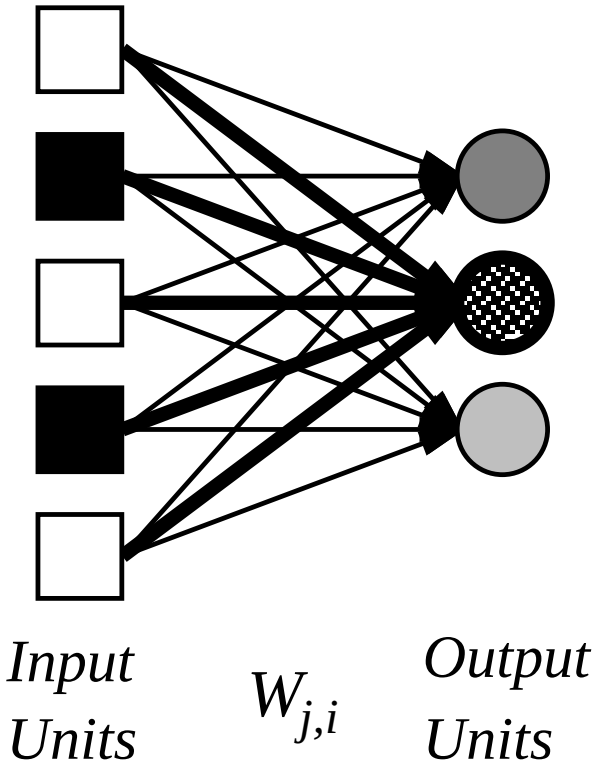
Neural Networks are used in classification and regression

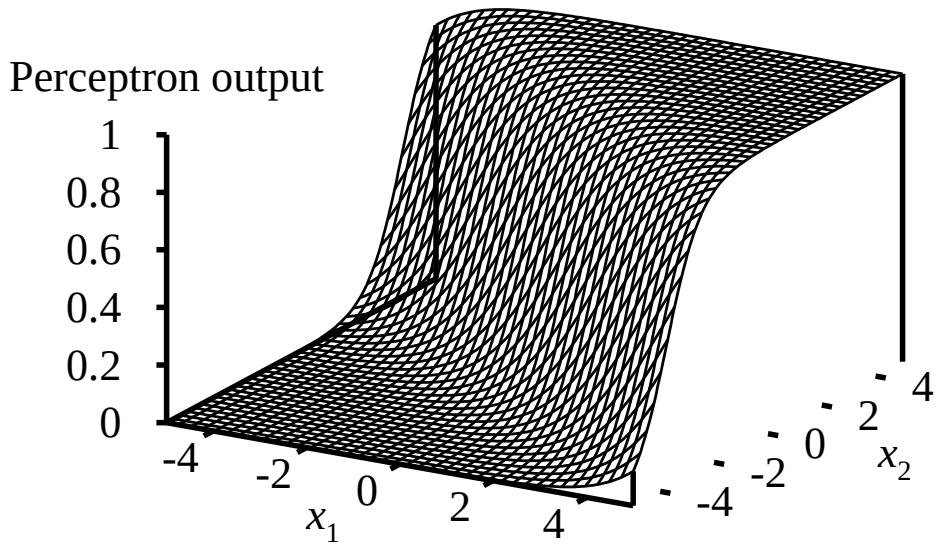
- Boolean classification:
  - value over 0.5 one class
  - value below 0.5 other class

- k-way classification
  - divide single output into k portions
  - k separate output unit

- continuous output
  - identity activation function in output unit

Single-layer NN (perceptrons)





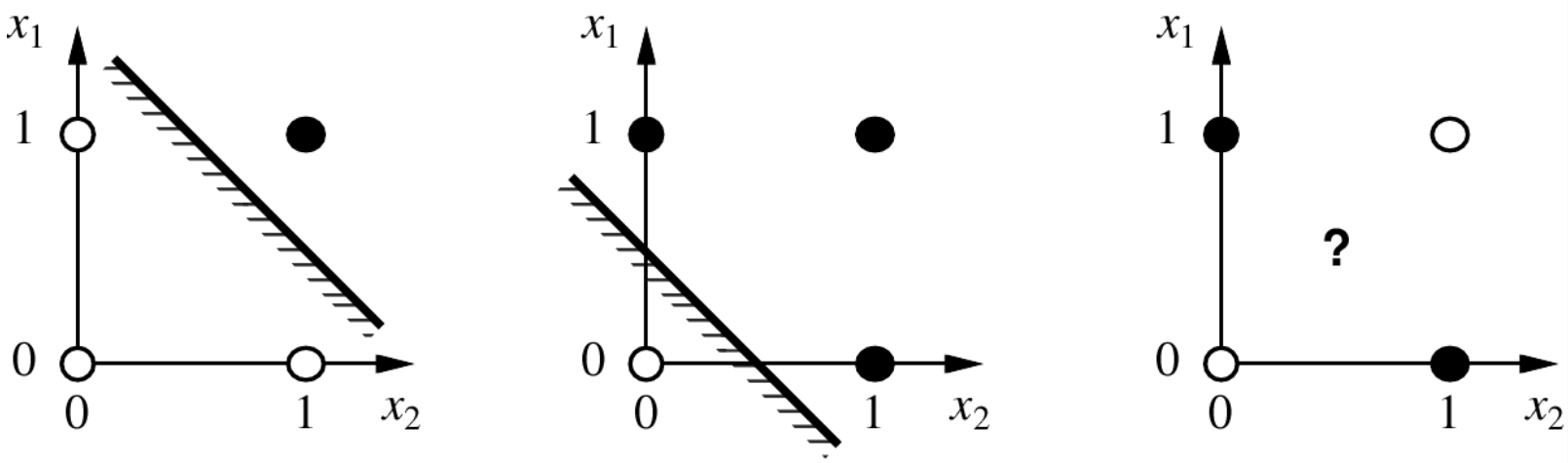
Output units all operate separately—no shared weights
Adjusting weights moves the location, orientation, and steepness of cliff

Expressiveness of perceptrons

Consider a perceptron with g = step function (Rosenblatt, 1957, 1960)
The output is 1 when:

$$\sum_j W_j x_j > 0 \quad \text{or} \quad \mathbf{W} \cdot \mathbf{x} > 0$$

Hence, it represents a linear separator in input space:
- hyperplane in multidimensional space
- line in 2 dimensions



Minsky & Papert (1969) pricked the neural network balloon

Perceptron learning

Learn by adjusting weights to reduce error on training set
The squared error for an example with input $\mathbf{x}$ and true output y is

$$E = \frac{1}{2} Err^2 \equiv \frac{1}{2}(y - h_{\mathbf{W}}(\mathbf{x}))^2 \ ,$$

Find local optima for the minimization of the function $E(\mathbf{W})$ in the vector of variables $\mathbf{W}$ by gradient methods.

Note, the function E depends on constant values $\mathbf{x}$ that are the inputs to the perceptron.

The function E depends on h which is non-convex, hence the optimization problem cannot be solved just by solving $\nabla E(\mathbf{W}) = 0$

Digression: Gradient methods

Gradient methods are iterative approaches:

- find a descent direction with respect to the objective function E
- move $\mathbf{W}$ in that direction by a step size

The descent direction can be computed by various methods, such as gradient descent, Newton-Raphson method and others. The step size can be computed either exactly or loosely by solving a line search problem.

Example: gradient descent

1. Set iteration counter $t = 0$, and make an initial guess $\mathbf{W}_0$ for the minimum
2. Repeat:
3. Compute a descent direction $\mathbf{p}_t = \nabla(E(\mathbf{W}_t))$
4. Choose α_t to minimize $f(\alpha) = E(\mathbf{W}_t - \alpha \mathbf{p}_t)$ over $\alpha \in \mathbb{R}_+$
5. Update $\mathbf{W}_{t+1} = \mathbf{W}_t - \alpha_t \mathbf{p}_t$, and $t = t + 1$
6. Until $\|\nabla f(\mathbf{W}_k)\| < \mathit{tolerance}$

Step 3 can be solved 'loosely' by taking a fixed small enough value $\alpha > 0$

Perceptron learning

In the specific case of the perceptron, the descent direction is computed by the gradient:

$$\begin{aligned}\frac{\partial E}{\partial W_j} &= Err \cdot \frac{\partial Err}{\partial W_j} = Err \cdot \frac{\partial}{\partial W_j}\left(y - g(\sum_{j=0}^{n} W_j x_j)\right) \\ &= -Err \cdot g'(in) \cdot x_j\end{aligned}$$

and the weight update rule (perceptron learning rule) in step 5 becomes:

$$W_j^{t+1} = W_j^t + \alpha \cdot Err \cdot g'(in) \cdot x_j$$

For threshold perceptron, $g'(in)$ is undefined: Original perceptron learning rule (Rosenblatt, 1957) simply omits $g'(in)$

Perceptron learning contd.

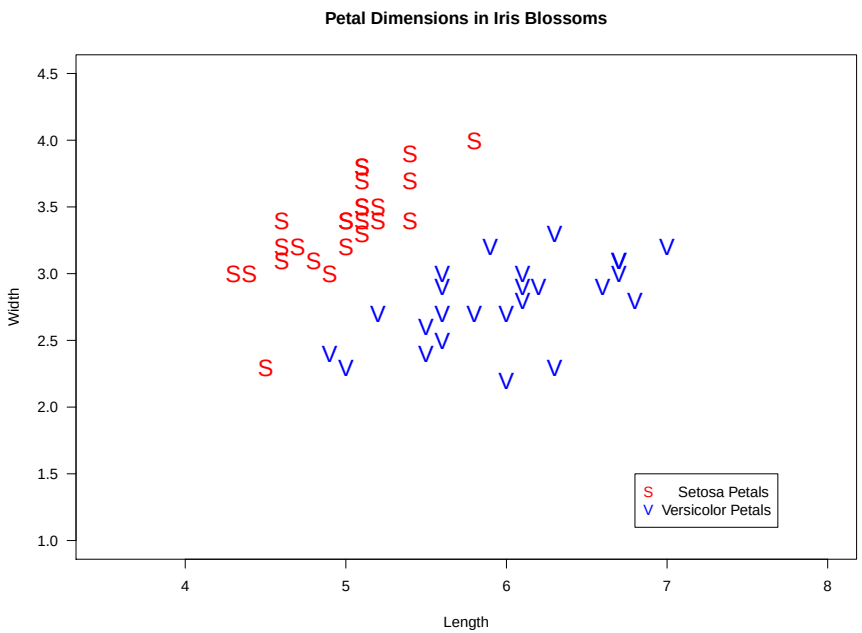
function Perceptron-Learning(*examples,network*) **returns** *perceptron weights*
inputs: *examples*, a set of examples, each with input
$\mathbf{x} = x_1, x_2, \ldots, x_n$ and output y
inputs: *network*, a perceptron with weights $W_j,\ j = 0, \ldots, n$ and
activation function g

repeat
for each e **in** *examples* **do**
$in \leftarrow \sum_{j=0}^{n} W_j x_j[e]$
$Err \leftarrow y[e] - g(in)$
$W_j \leftarrow W_j + \alpha \cdot Err \cdot g'(in) \cdot x_j[e]$
end
until all examples correctly predicted or stopping criterion is reached
return *network*

Perceptron learning rule converges to a consistent function
for any linearly separable data set

Numerical Example

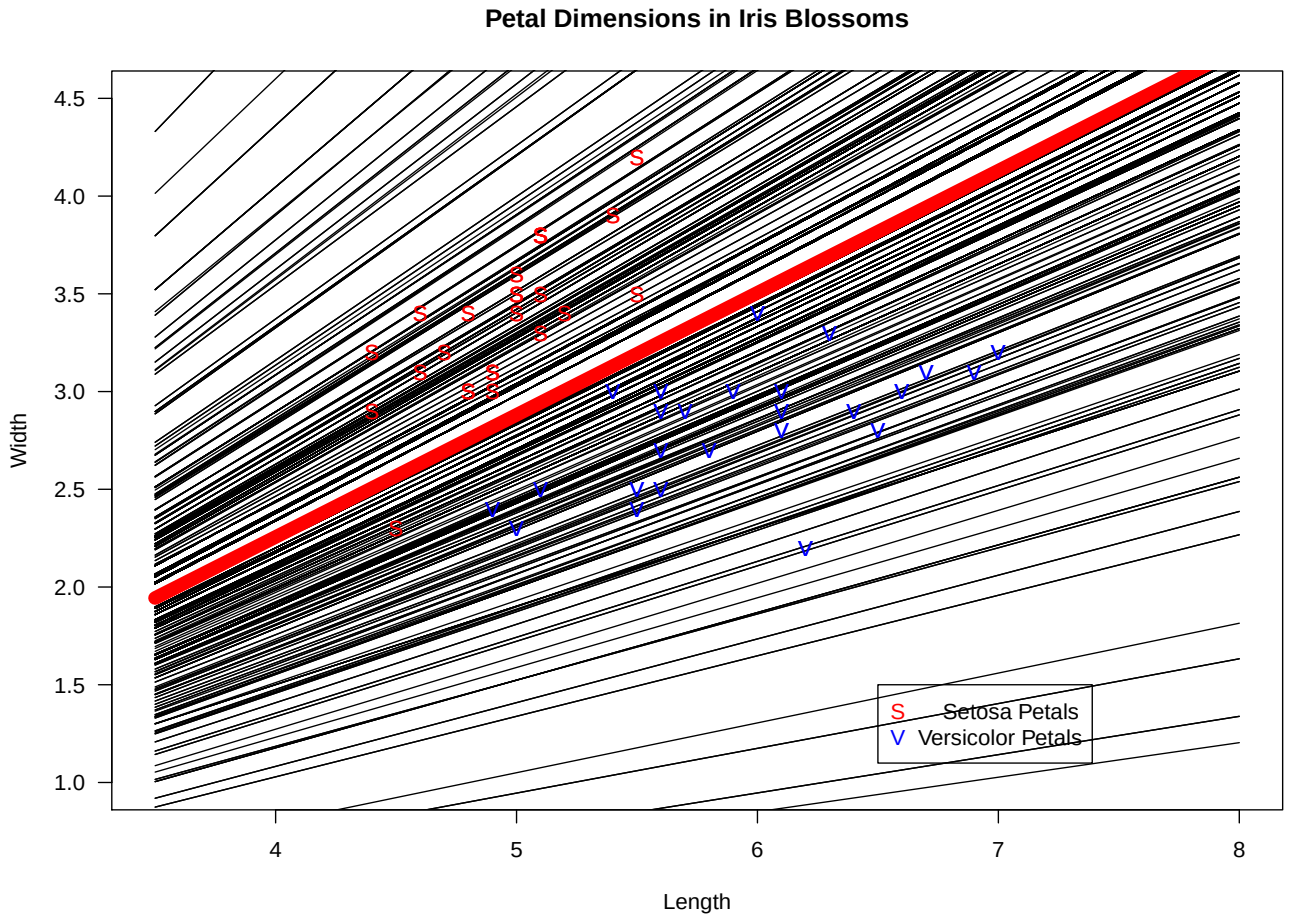
The (Fisher's or Anderson's) iris data set gives the measurements in centimeters of the variables petal length and width, respectively, for 50 flowers from each of 2 species of iris. The species are "Iris setosa", and "versicolor".



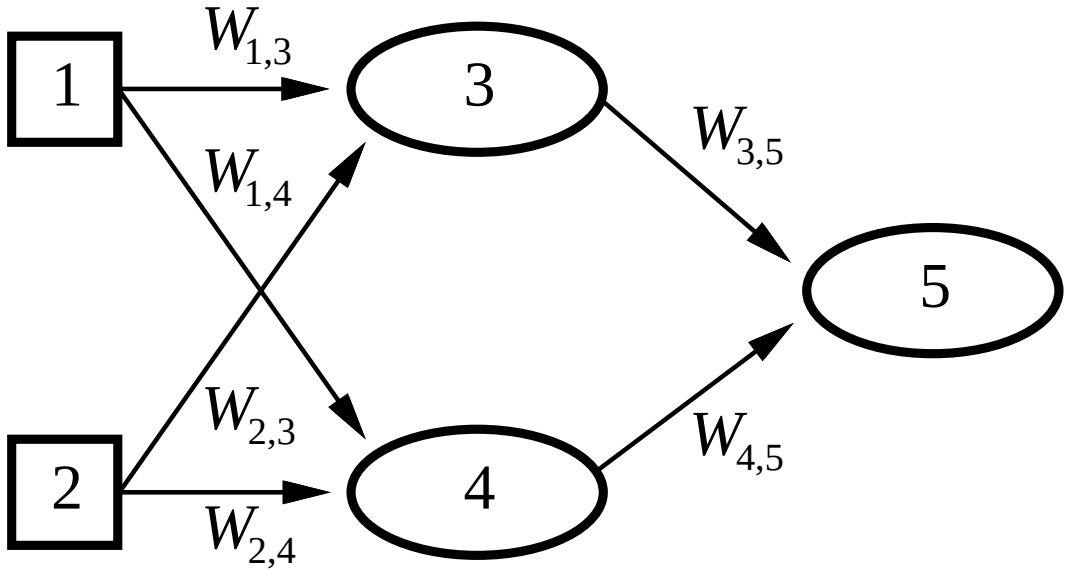

```
> head(iris.data)

   Sepal.Length Sepal.Width    Species id
6           5.4         3.9     setosa -1
4           4.6         3.1     setosa -1
84          6.0         2.7 versicolor  1
31          4.8         3.1     setosa -1
77          6.8         2.8 versicolor  1
15          5.8         4.0     setosa -1
```

```
> sigma <- function(w, point) {
+     x <- c(point, 1)
+     sign(w %*% x)
+ }
> w.0 <- c(runif(1), runif(1), runif(1))
> w.t <- w.0
> for (j in 1:1000) {
+     i <- (j - 1)%%50 + 1
+     diff <- iris.data[i, 4] - sigma(w.t, c(iris.data[i, 1], iris.data[i, 2]))
+     w.t <- w.t + 0.2 * diff * c(iris.data[i, 1], iris.data[i, 2], 1)
+ }
```



Multilayer Feed-forward

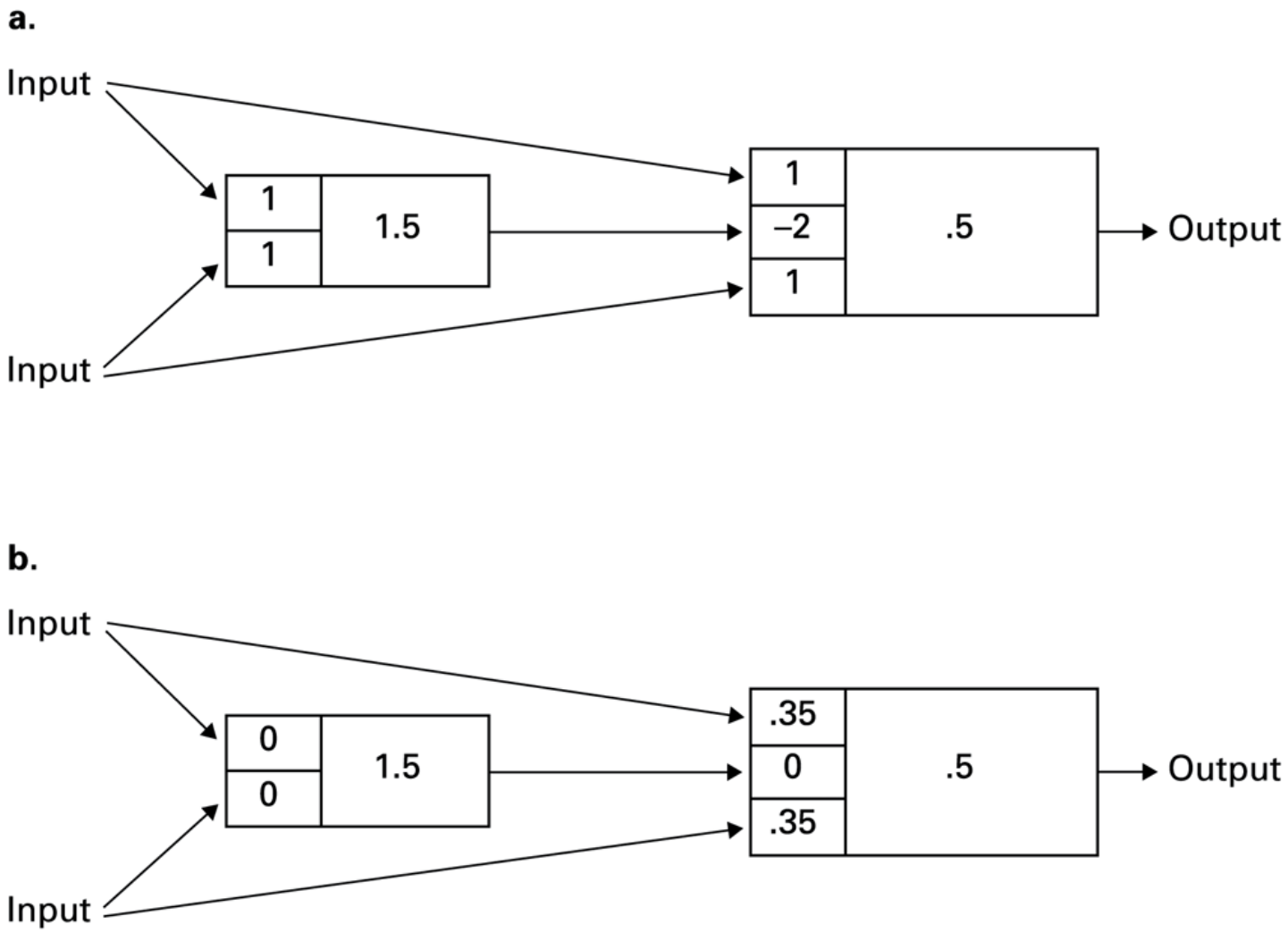


Feed-forward network = a parametrized family of nonlinear functions:

$$\begin{aligned} a_5 &= g(W_{3,5} \cdot a_3 + W_{4,5} \cdot a_4) \\ &= g(W_{3,5} \cdot g(W_{1,3} \cdot a_1 + W_{2,3} \cdot a_2) + W_{4,5} \cdot g(W_{1,4} \cdot a_1 + W_{2,4} \cdot a_2)) \end{aligned}$$

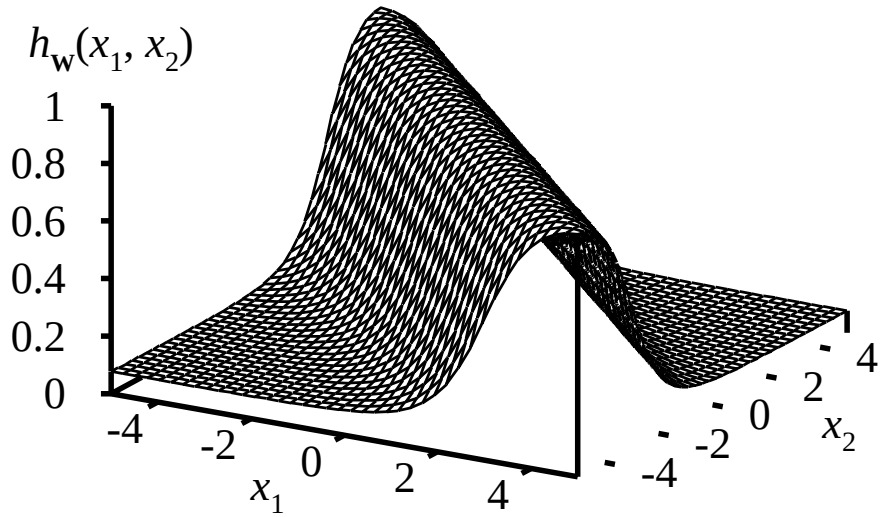
Adjusting weights changes the function: do learning this way!

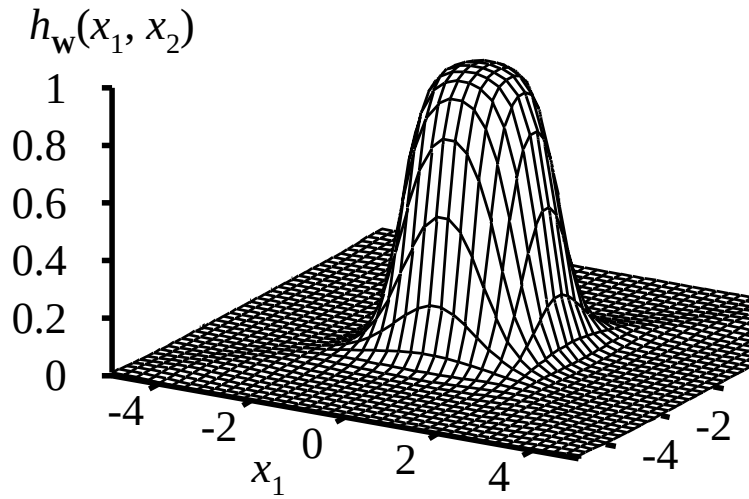
Neural Network with two layers



Expressiveness of MLPs

All continuous functions with 2 layers, all functions with 3 layers





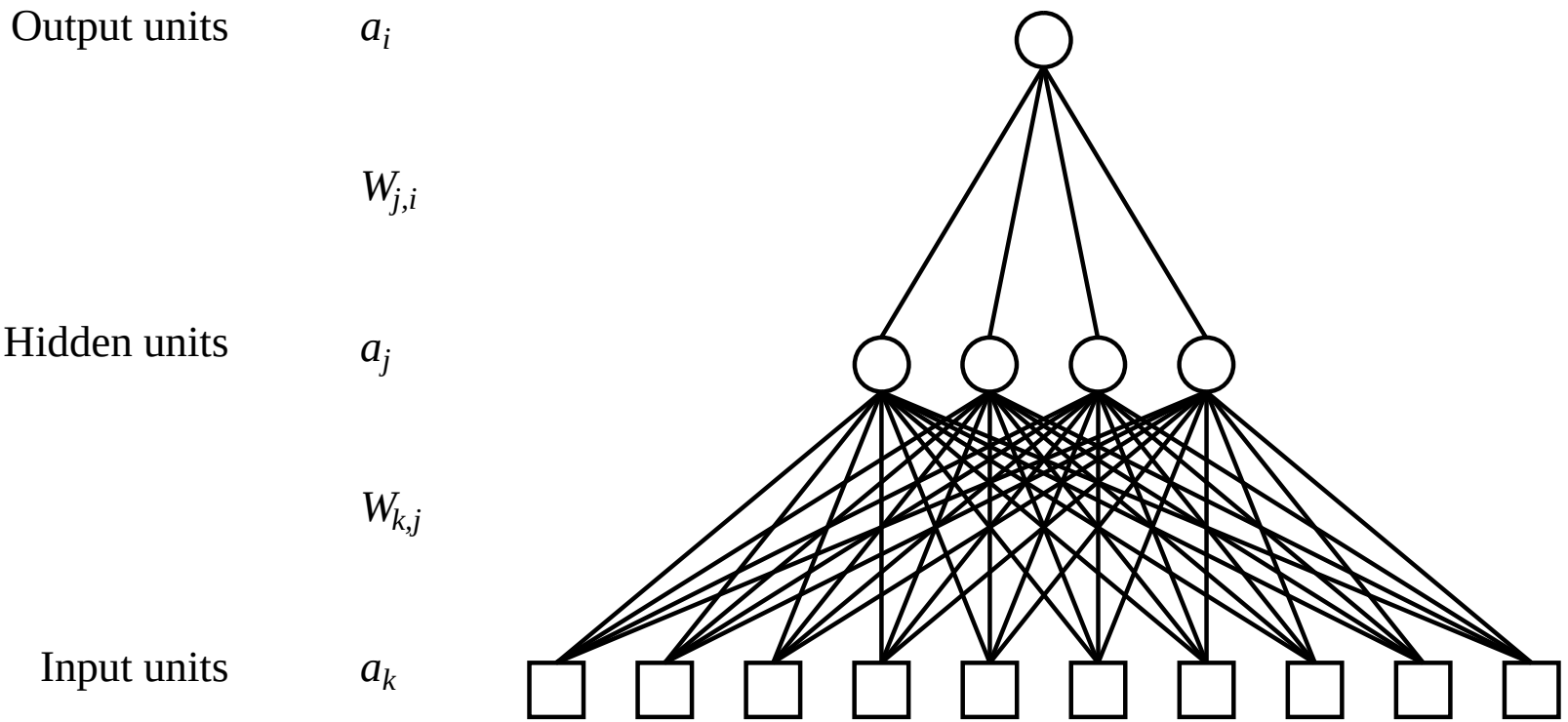
Combine two opposite-facing threshold functions to make a ridge
Combine two perpendicular ridges to make a bump
Add bumps of various sizes and locations to fit any surface
Proof requires exponentially many hidden units

Backpropagation Algorithm

- Supervised learning method to train multilayer feedforward NNs with diffrerentiable transfer functions.
- Adjust weights along the negative of the gradient of performance function.
- Forward-Backward pass.
- Sequential or batch mode
- Convergence time vary exponentially with number of inputs
- Avoid local minima by simulated annealing and other metaheuristics

Multilayer perceptrons

Layers are usually fully connected;
numbers of hidden units typically chosen by hand



Back-propagation learning

Output layer: same as for single-layer perceptron,

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$$

where $\Delta_i = Err_i \times g'(in_i)$.
Note: the general case has multiple output units hence: $\mathbf{Err} = (\mathbf{y} - \mathbf{h}_w(x))$

Hidden layer: **back-propagate** the error from the output layer:

$$\Delta_j = g'(in_j) \sum_i W_{j,i} \Delta_i \qquad \text{(sum over the multiple output units)}$$

Update rule for weights in hidden layer:

$$W_{k,j} \leftarrow W_{k,j} + \alpha \times a_k \times \Delta_j \ .$$

(Most neuroscientists deny that back-propagation occurs in the brain)

Back-propagation derivation

The squared error on a single example is defined as

$$E = \frac{1}{2}\sum_i (y_i - a_i)^2 \ ,$$

where the sum is over the nodes in the output layer.

$$\begin{aligned}
\frac{\partial E}{\partial W_{j,i}} &= -(y_i - a_i)\frac{\partial a_i}{\partial W_{j,i}} = -(y_i - a_i)\frac{\partial g(in_i)}{\partial W_{j,i}} \\
&= -(y_i - a_i)g'(in_i)\frac{\partial in_i}{\partial W_{j,i}} = -(y_i - a_i)g'(in_i)\frac{\partial}{\partial W_{j,i}}\left(\sum_j W_{j,i}a_j\right) \\
&= -(y_i - a_i)g'(in_i)a_j = -a_j\Delta_i
\end{aligned}$$

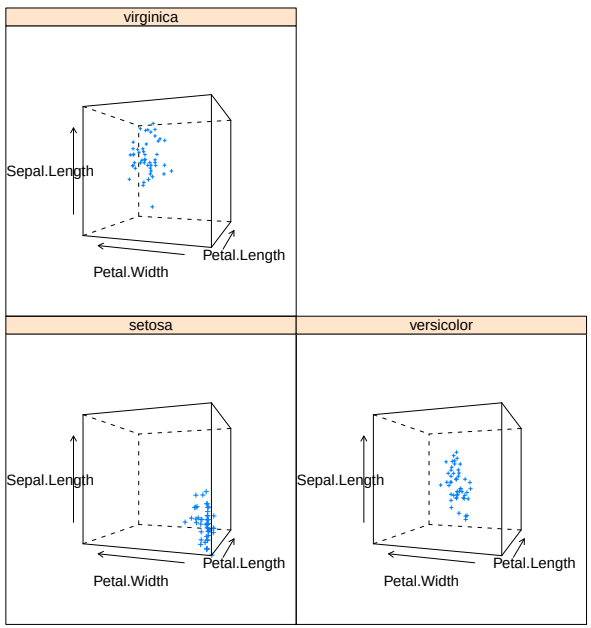
Back-propagation derivation contd.

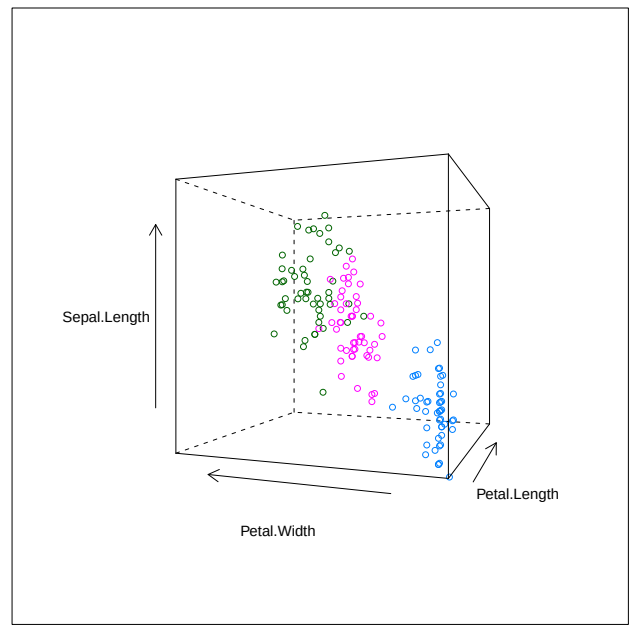
For the hidden layer:

$$\begin{aligned}
\frac{\partial E}{\partial W_{k,j}} &= -\sum_i (y_i - a_i)\frac{\partial a_i}{\partial W_{k,j}} = -\sum_i (y_i - a_i)\frac{\partial g(in_i)}{\partial W_{k,j}} \\
&= -\sum_i (y_i - a_i) g'(in_i)\frac{\partial in_i}{\partial W_{k,j}} = -\sum_i \Delta_i \frac{\partial}{\partial W_{k,j}}\left(\sum_j W_{j,i} a_j\right) \\
&= -\sum_i \Delta_i W_{j,i}\frac{\partial a_j}{\partial W_{k,j}} = -\sum_i \Delta_i W_{j,i}\frac{\partial g(in_j)}{\partial W_{k,j}} \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j)\frac{\partial in_j}{\partial W_{k,j}} \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j)\frac{\partial}{\partial W_{k,j}}\left(\sum_k W_{k,j} a_k\right) \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j) a_k = -a_k \Delta_j
\end{aligned}$$

Numerical Example

The (Fisher's or Anderson's) iris data set gives the measurements in centimeters of the variables petal length and width, respectively, for 50 flowers from each of 2 species of iris. The species are "Iris setosa", and "versicolor".





Numerical Example


```
> samp <- c(sample(1:50, 25), sample(51:100, 25), sample(101:150, 25))
> Target <- class.ind(iris$Species)
> ir.nn <- nnet(Target ~ Sepal.Length * Petal.Length * Petal.Width, data = iris, subset = sam
+     size = 2, rang = 0.1, decay = 5e-04, maxit = 200, trace = FALSE)
> test.cl <- function(true, pred) {
+     true <- max.col(true)
+     cres <- max.col(pred)
+     table(true, cres)
+ }
> test.cl(Target[-samp, ], predict(ir.nn, iris[-samp, c(1, 3, 4)]))
```

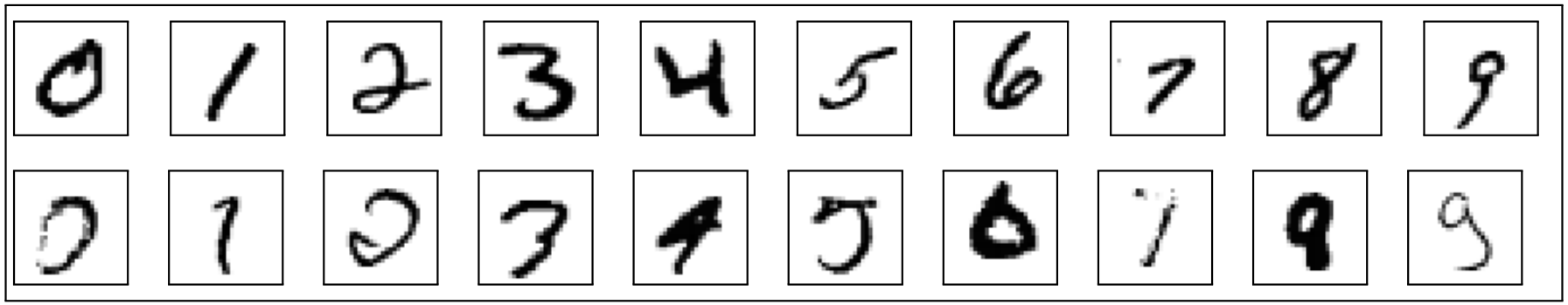
```
    cres
true  1  2  3
   1 25  0  0
   2  0 22  3
   3  0  2 23
```

Learning structures

Beside weights also structure can be learned:

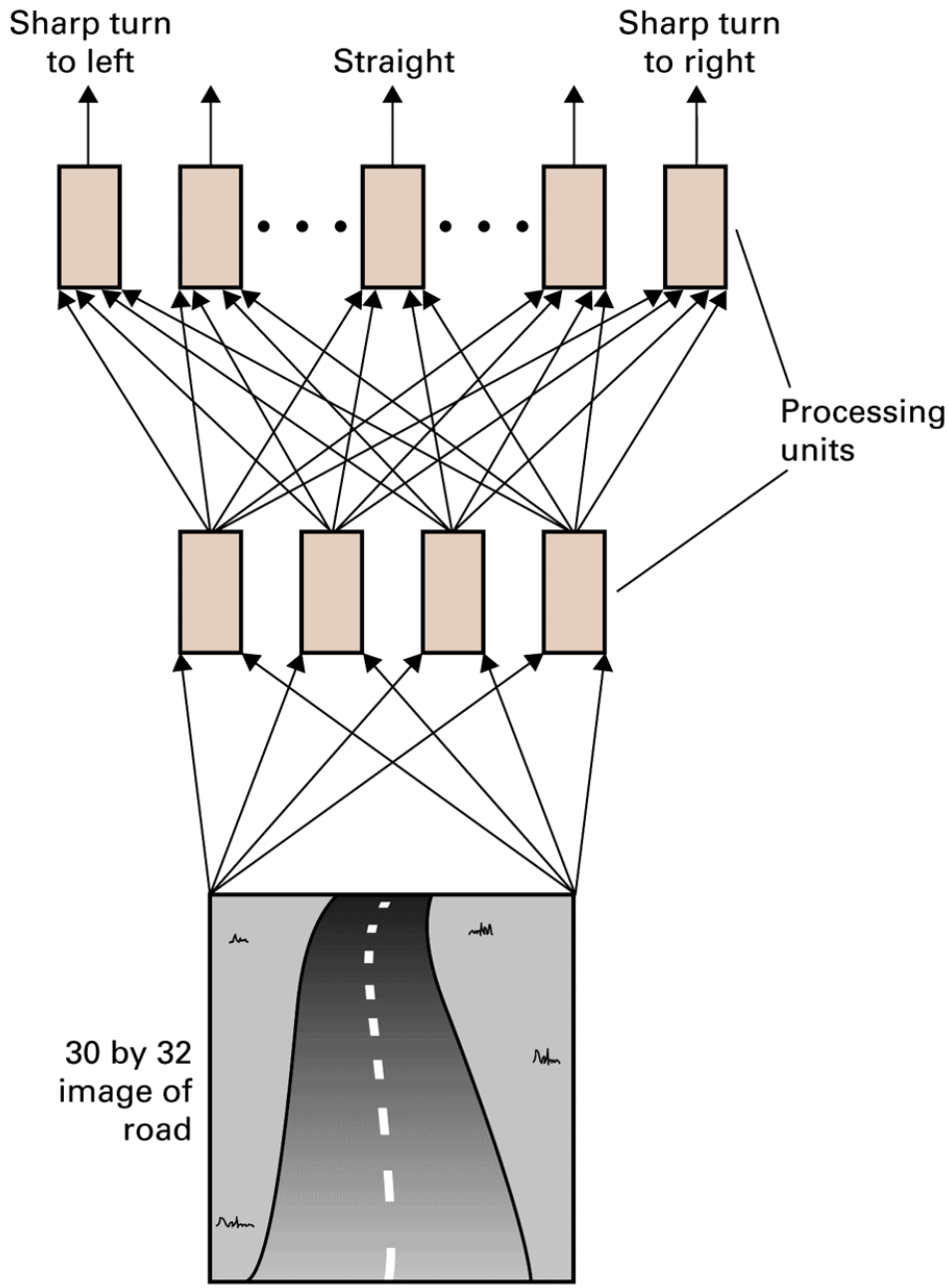
- Optimal brain damage: iteratively remove single edegs or units if performance does not worsen after weights are re-learned
- Tiling: iteralively add units and links and re-learn weights

Handwritten digit recognition



- 400–300–10 unit MLP = 1.6% error
- LeNet: 768–192–30–10 unit MLP = 0.9% error
  http://yann.lecun.com/exdb/lenet/
- Current best (kernel machines, vision algorithms) $\approx$ 0.6% error
- Humans are at 0.2% – 2.5 % error

Another Practical Example



Directions of research in ANN

- Representational capability assuming unlimited number of neurons (no training)
- Numerical analysis or approximation theoretic: how many hidden units are necessary to achieve a certain approximation error? (no training) Results for single hidden layer and multiple hidden layers
- Sample complexity: how many samples are needed to characterize a certain unknown mapping.
- Efficient learning: backpropagation has the curse of dimensionality problem

Approximation properties

NNs with 2 hidden layers and arbitrarily many nodes can approximate any real-valued function up to any desired accuracy, using continuous activation functions

> *E.g.: required number of hidden units grows exponentially with number of inputs.*
> $2^n/n$ *hidden units needed to encode all Boolean functions of* n *inputs*

However profs are not constructive.

More interest in efficiency issues: NNs with small size and depth

Size-depth trade off: more layers $\rightsquigarrow$ more costly to simulate

Outline

Training and Assessment

Use different data for different tasks:

- Training and Test data: holdout cross validation
- If little data: k-fold cross validation

Avoid peeking:

- Weights learned on training data.
- Parameters such as learning rate α and net topology compared on validation data
- Final assessment on test data

Ensemble Methods

- Use majority rule to predict among K hypothesis learned.
- If the hypothesis are independent this yields a considerable reduction of misclassification
- Boosting: weight adaptively the examples

Learning Theory

- Probably approximately correct (PAC) learning
- Vapnik-Chervonenkis (VC) dimensions provide information-theoretic bounds to sample complexities in continuous function classes

Summary

- Supervised learning
- Decision trees
- Linear models
- Neural Networks
  - Perceptron learning rule: an algorithm for learning weights in single layered networks.
  - Perceptrons: linear separators, insufficiently expressive
  - Multi-layer networks are sufficiently expressive
  - Many applications: speech, driving, handwriting, fraud detection, etc.
- k nearest neighbor, non-parametric regression