

# ANN History: First steps

## ❑ **Pitts & McCulloch (1943)**

- ❑ First mathematical model of biological neurons
- ❑ All Boolean operations can be implemented by these neuron-like nodes (with different threshold and excitatory/inhibitory connections).
- ❑ Competitor to Von Neumann model for general purpose computing device
- ❑ Origin of automata theory.

## ❑ **Hebb (1949)**

- ❑ Hebbian rule of learning: increase the connection strength between neurons  $i$  and  $j$  whenever both  $i$  and  $j$  are activated.
- ❑ Or increase the connection strength between nodes  $i$  and  $j$  whenever both nodes are simultaneously ON or OFF.

# ANN History: Perceptron

## ❑ Early booming (50's – early 60's)

- Rosenblatt (1958)
  - Perceptron: network of threshold nodes for pattern classification. Perceptron learning rule – first learning algorithm
  - Perceptron convergence theorem: everything that can be represented by a perceptron can be learned
- Widrow and Hoff (1960, 1962)
  - Learning rule that is based on minimization methods
- Minsky's attempt to build a general purpose machine with Pitts/McCulloch units

# ANN History: Setback ...

- **The setback (mid 60's – late 70's)**
  - Minsky and Papert publish a book "Perceptrons" (1969):
    - Single layer perceptrons **cannot represent** (learn) simple functions such as **XOR**
    - Multi-layer of non-linear units may have greater power but there was no learning algorithm for such nets
    - Scaling problem: connection weights may grow infinitely
  - ***US Defense/Government stop funding research on ANN***

# ANN History: Backpropagation

- **Renewed enthusiasm and progress (80's – 90's)**
  - New techniques
    - **Backpropagation** learning for multi-layer feed forward nets (with non-linear, differentiable node functions)
    - Physics inspired models (Hopfield net, Boltzmann machine, etc.)
    - Unsupervised learning (LVQ nets, Kohonen nets)
  - Impressive applications (character recognition, speech recognition, text-to-speech transformation, process control, associative memory, etc.)

## **But:**

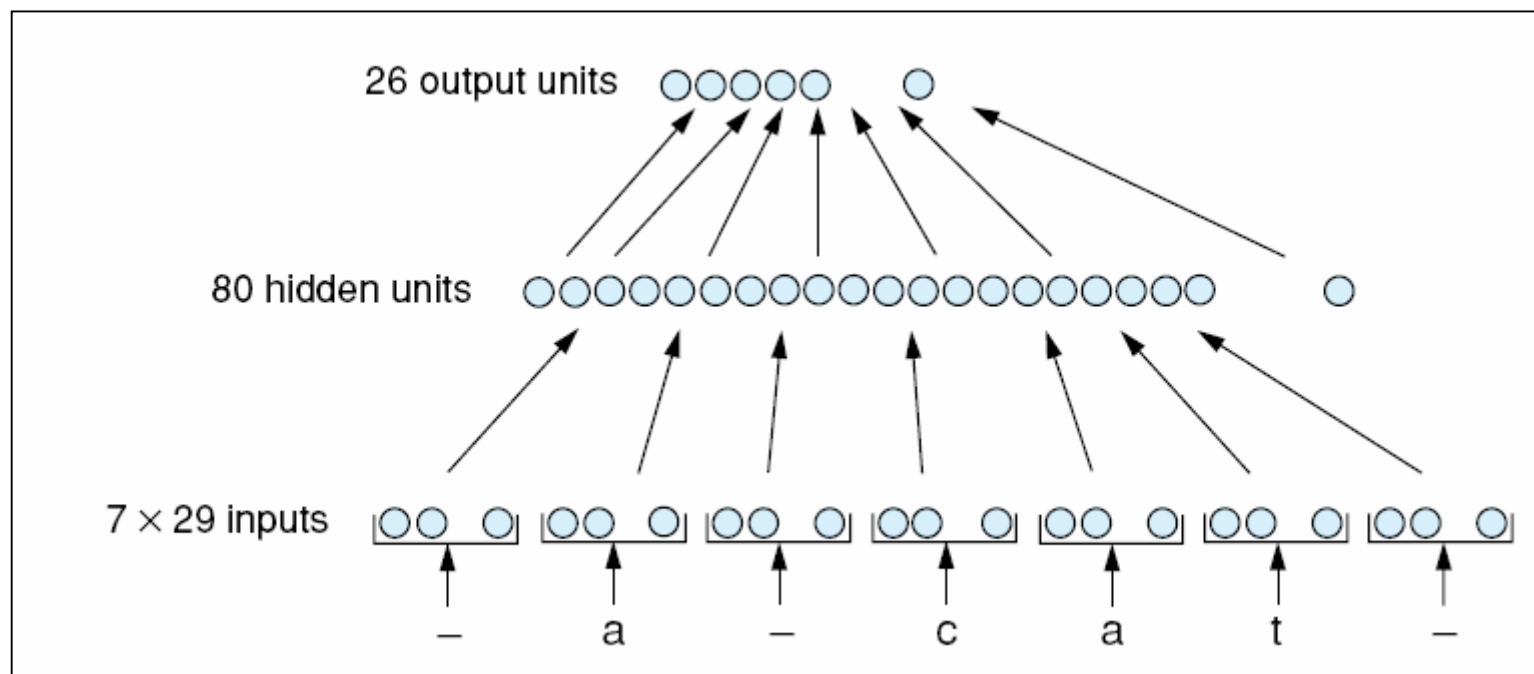
- Criticism from Statisticians, Neurologists, Biologists, Ordinary Users, ...
- Lots of ad-hoc solutions, "wild creativity"
- A lot of rubbish is produced ...

# NETtalk (Sejnowski & Rosenberg, 1987)

---

- The task is to **learn to pronounce English text** from examples (text-to-speech)
- Training data: a list of **<phrase, phonetic representation>**
- **Input:** 7 consecutive characters from written text presented in a moving window that scans text
- **Output:** phoneme code giving the pronunciation of the letter at the center of the input window
- **Network topology:** 7x29 binary inputs (26 chars + punctuation marks), 80 hidden units and 26 output units (phoneme code). Sigmoid units in hidden and output layer

# NETtalk



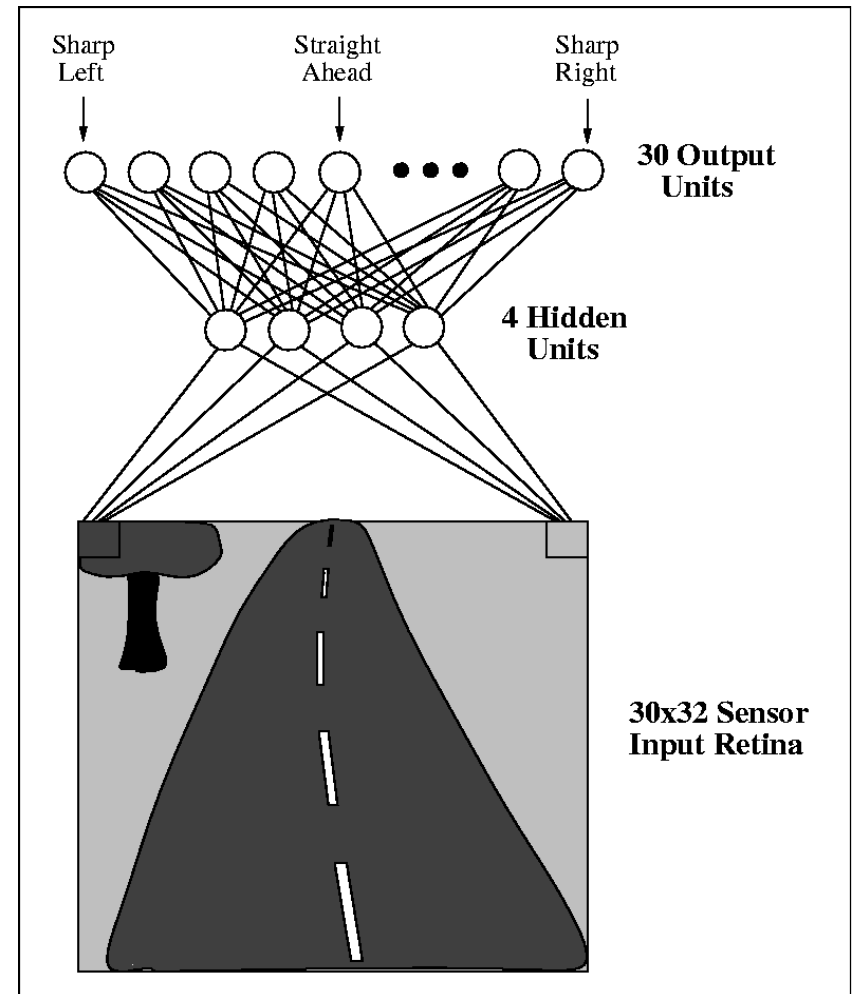
## NETtalk (contd.)

---

- Training protocol: 95% accuracy on training set after 50 epochs of training by full gradient descent.  
78% accuracy on a test set
- DEC-talk: a rule-based system crafted by experts (a decade of efforts by many linguists)
- Functionality/Accuracy almost the same
- Try: <http://cnl.salk.edu/Media/nettalk.mp3>

# ALVINN (1989)

D.A. Pomerleau, Autonomous Land Vehicle In a Neural Network (NIPS '89)



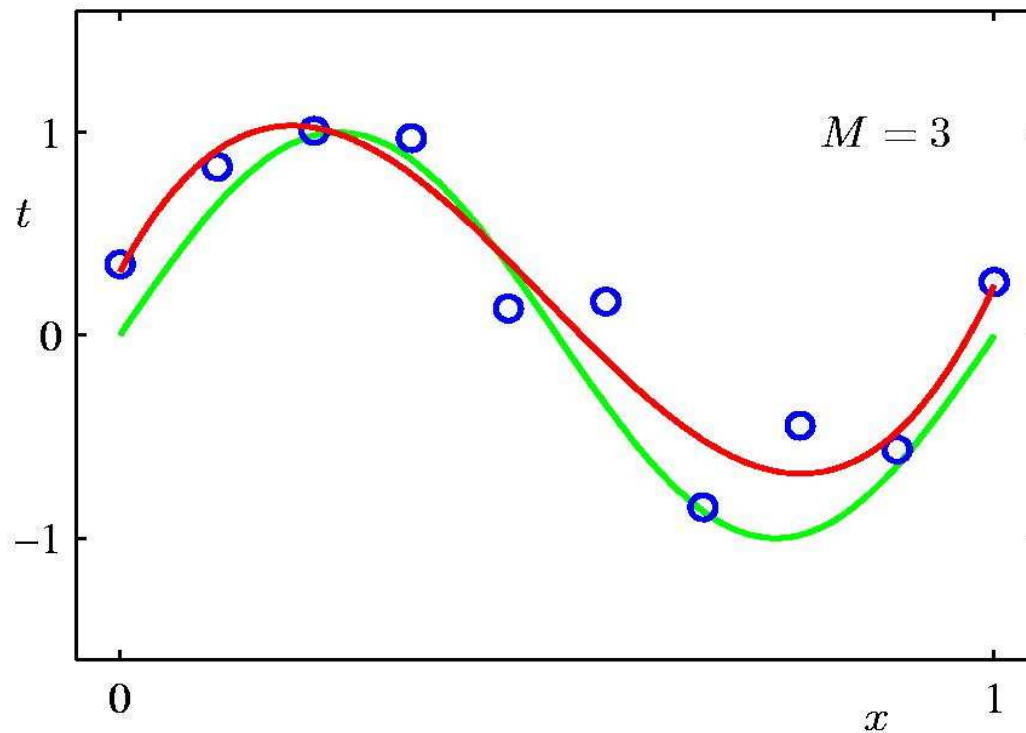
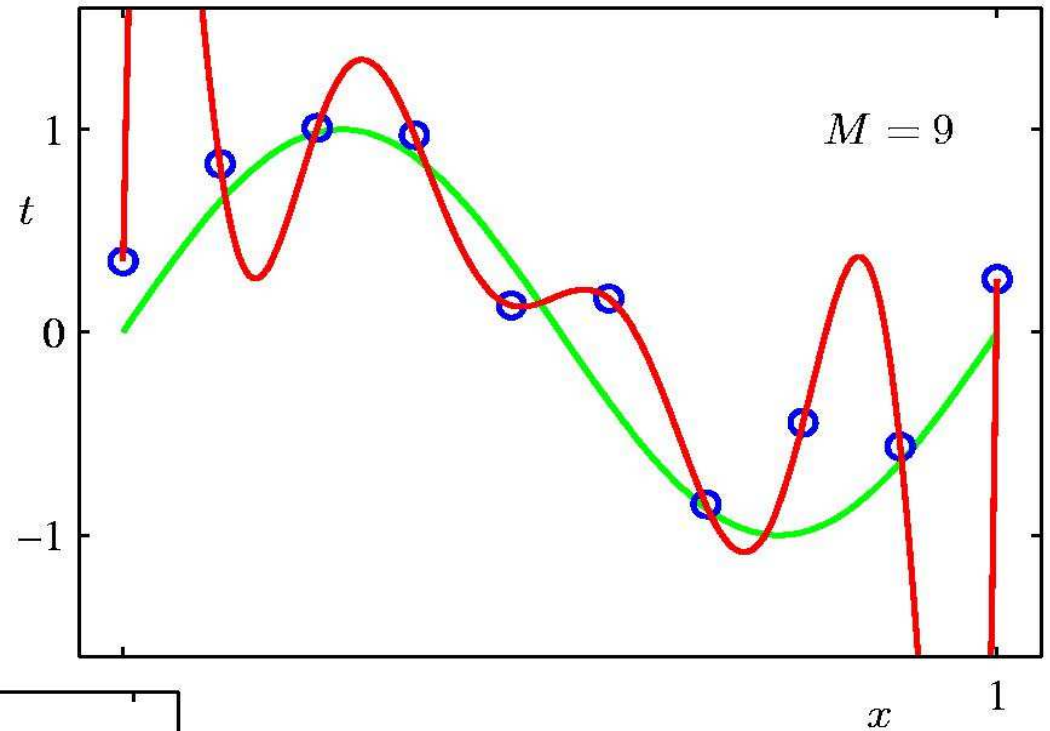


# Why only “shallow networks” ?

---

- In practice, "classical" multi-layer networks work fine only for a very small number of hidden layers (typically 1 or 2) - this is an empirical fact ...
- Adding layers is harmful because:
  - the increased number of weights quickly leads to data overfitting (lack of generalization)
  - huge number of (bad) local minima trap the gradient descent algorithm
  - vanishing or exploding gradients (the update rule involves products of many numbers) cause additional problems

**Overfitting:**  
which polynomial  
better approximates  
the green line?



The more parameters  
the higher the risk  
of overfitting !

# Why do we need many layers?



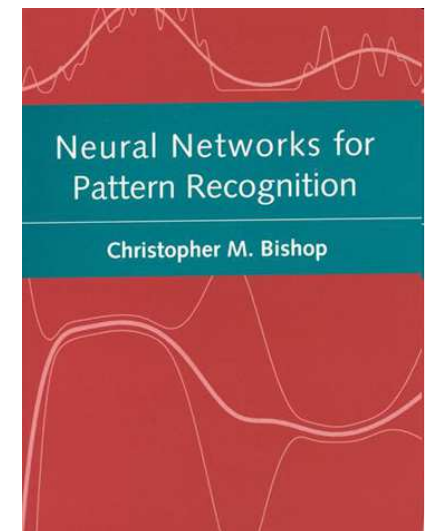
- In theory, one hidden layer is sufficient to model any function with an arbitrary accuracy; however, the number of required nodes and weights grows exponentially fast
- The deeper the network the less nodes are required to model "complicated" functions
- Consecutive layers "learn" features of the training patterns; from simplest (lower layers) to more complicated (top layers)
- Visual cortex consists of about 10 layers

# An end of hype?

---

## Dominance of new techniques (90's - 2005):

- Support Vector Machines (Vapnik)
- Kernel Methods (Vapnik, Scholkopf, ...)
- Ensemble methods (Breiman, Hasti, Tibshirani, Friedman,...)
- Random Forests
- *Neural Networks lose their prominent role in the fields of Pattern Recognition and Machine Learning*
- *since 1996 no new textbooks on Neural Networks !*



# Deep Learning Revolution



**1989 : LeCun et al: A Convolutional Network for OCR (AT&T)**

**~ 2006 : G. Hinton's group: new ideas, inventions, applications**

- Restricted Boltzmann Machine as building block for DNN
- Contrastive Divergence algorithm
- Combine supervised with unsupervised learning
- Deep Belief Networks
- Stacked Auto-Encoders
- Deep Recurrent Networks

**Many spectacular applications beating existing approaches**

# Enabling Factors



- **Availability of “Big Data”:** the more data we have the better we can train a network
- **Powerful Hardware (GPU’s):** speedup of the training process by 100-1000 times reduces the training time from years to hours
- **New algorithms and architectures:** leaving the MLP standard behind ...
- ***Many spectacular successes!***

# Statistical Pattern Recognition



1. What is Pattern Recognition?
2. The “a or b?” example
3. Bayes Formula
4. Optimal Decision Boundary
5. Misclassification Costs and Minimizing Risk
6. Classification and Regression
7. Model Complexity and Overfitting; Regularization

# What is pattern recognition?

“The assignment of a physical object or event to one of several prespecified categories” -- Duda & Hart

- A **pattern** is an object, process or event that can be given a name.
- A **pattern class** (or category) is a set of patterns sharing common attributes and usually originating from the same source.
- During **recognition** (or **classification**) given objects are assigned to prescribed classes.
- A **classifier** is a machine which performs classification.



# Examples of applications

- **Optical Character**

**Recognition (OCR)**

- **Biometrics**

- **Diagnostic systems**

- **Military applications**

Neural Networks

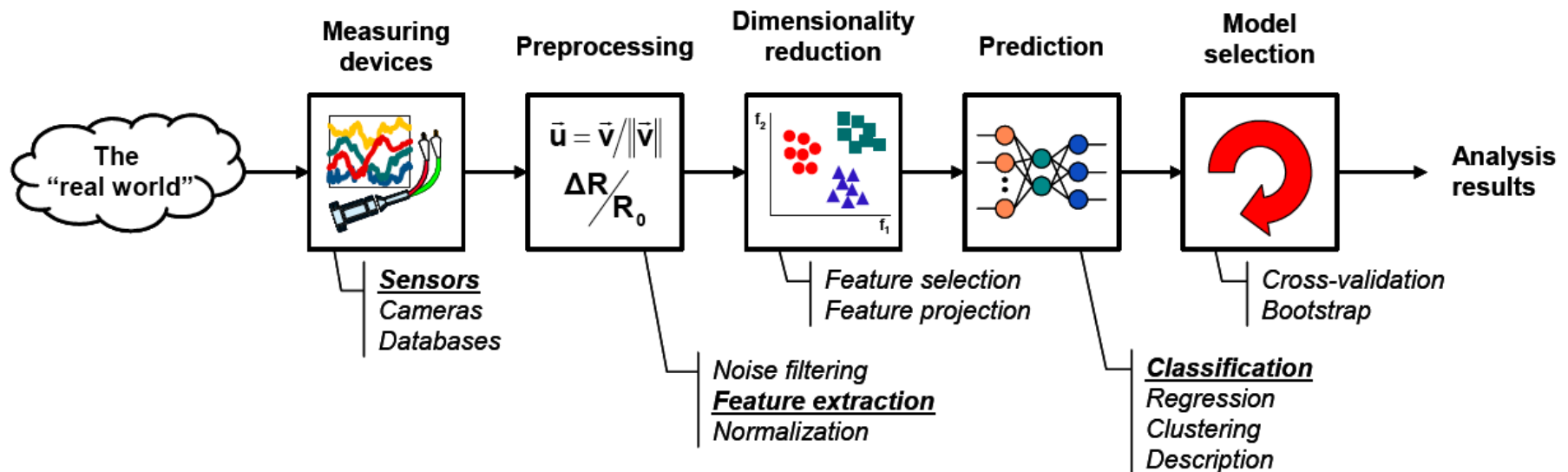
- Handwritten: sorting letters by postal code, input device for PDA's.
- Printed texts: reading machines for blind people, digitalization of text documents.

- Face recognition, verification, retrieval.
- Finger prints recognition.
- Speech recognition.

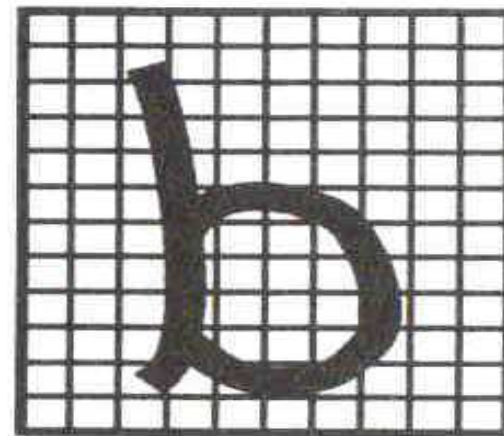
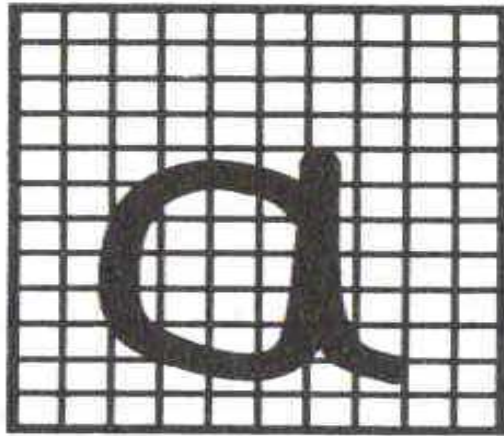
- Medical diagnosis: X-Ray, EKG analysis.
- Machine diagnostics, waster detection.

- Automated Target Recognition (ATR).
- Image segmentation and analysis (recognition from aerial or satelite photographs).

# Components of a PR System



# Example: “a” or “b” ?



9665407401 9665407401 9665407401  
3134727121 3134727121 3134727121  
1742351244 1742351244 1742351244

# Task

- Classify images of handwritten **a**'s ( $C_1$ ) and **b**'s ( $C_2$ )

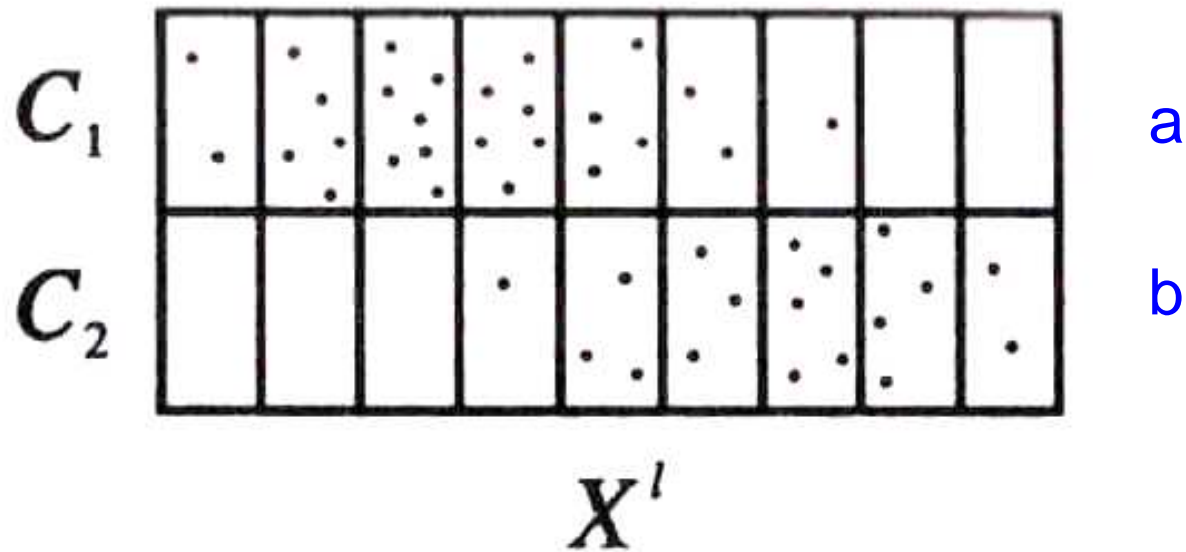
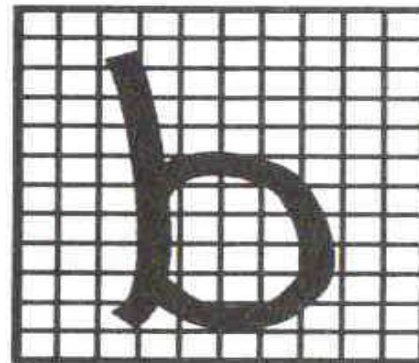
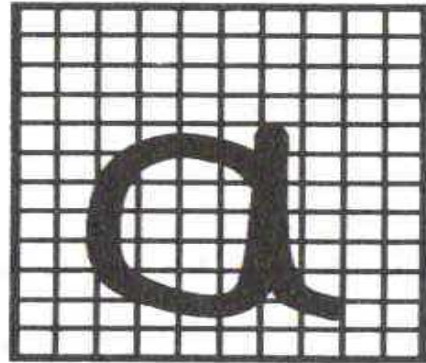
Let us consider one feature only:

$X = \text{height/width}$

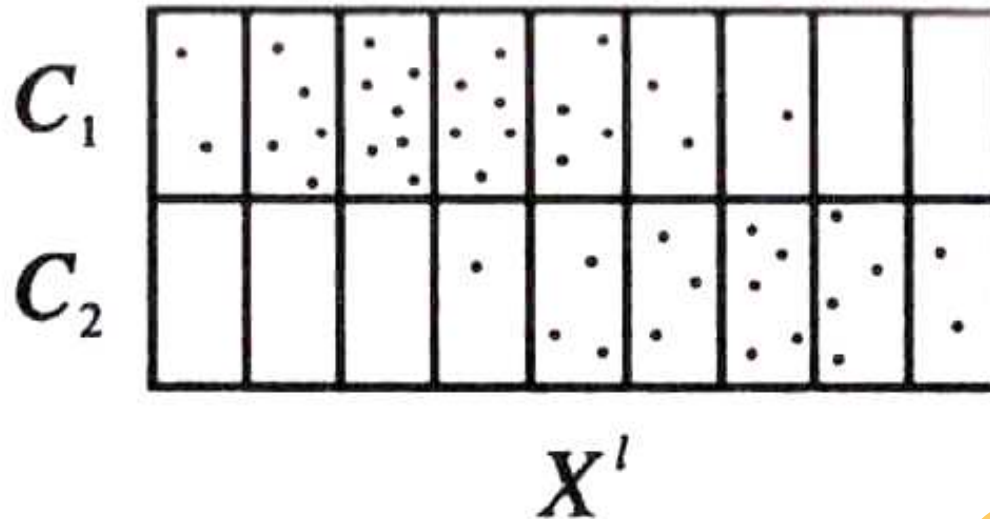
- 1) How would you decide if you don't know the value of  $X$ ?
  - Choose  $C_1$  if  $P(C_1) > P(C_2)$  :prior probabilities
  - Choose  $C_2$  otherwise
- 2) How about if you know the value of  $X$ ?
  - Estimate probabilities  $P(C_1|X)$  and  $P(C_2|X)$ , and take bigger one!

How could we estimate  $P(C_1|X)$  and  $P(C_2|X)$  from data?

# Example



# Bayes' Theorem



multiply {

$$P(C_1, X=x) = \frac{\text{num. samples in corresponding box}}{\text{num. all samples}}$$

//joint probability of  $C_1$  and  $X$

$$P(X=x|C_1) = \frac{\text{num. samples in corresponding box}}{\text{num. of samples in } C_1\text{-row}}$$

//class-conditional probability of  $X$

$$P(C_1) = \frac{\text{num. of samples in } C_1\text{-row}}{\text{num. all samples}}$$

//prior probability of  $C_1$

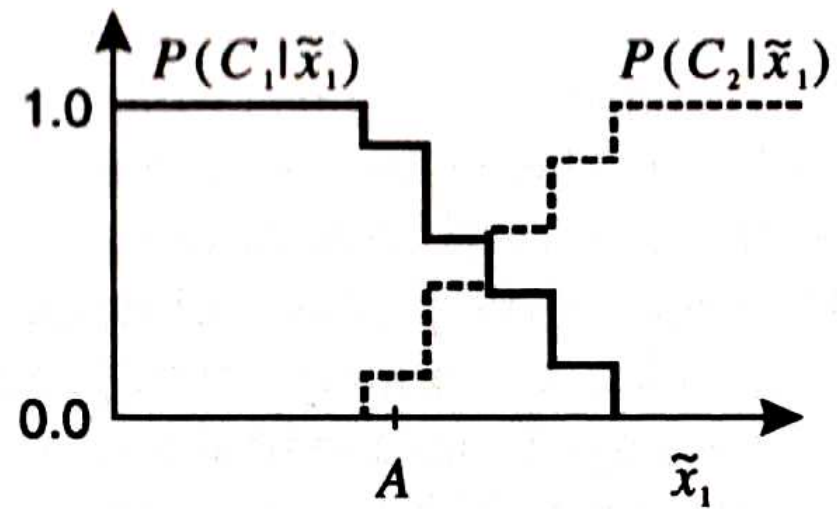
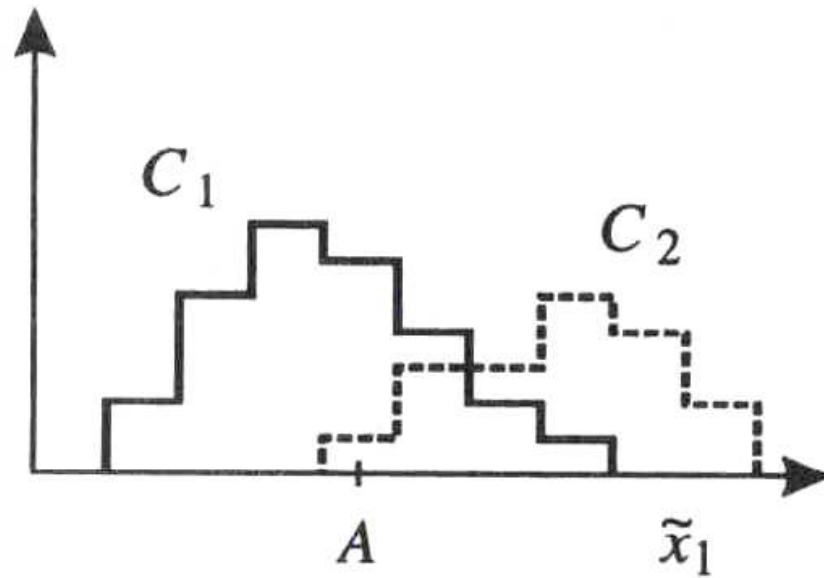
$$P(C1, X=x) = P(X=x|C_1) P(C_1)$$

$$P(C1, X=x) = P(C_1|X=x) P(X=x)$$

$$P(C1|X=x) = P(X=x|C_1) P(C_1) / P(X=x)$$

$$\text{posterior} = \frac{\text{likelihood} \times \text{prior}}{\text{normalization factor}}$$

# Histograms



# Bayes Theorem

$$P(C_k | X) = \frac{P(X | C_k)P(C_k)}{P(X)}$$

scaling factor :

$$P(X) = P(X | C_1)P(C_1) + P(X | C_2)P(C_2)$$

priors :  $P(C_k)$

class - conditional :  $P(X | C_k)$

posteriors :  $P(C_k | X)$



# Posterior Probability Distribution

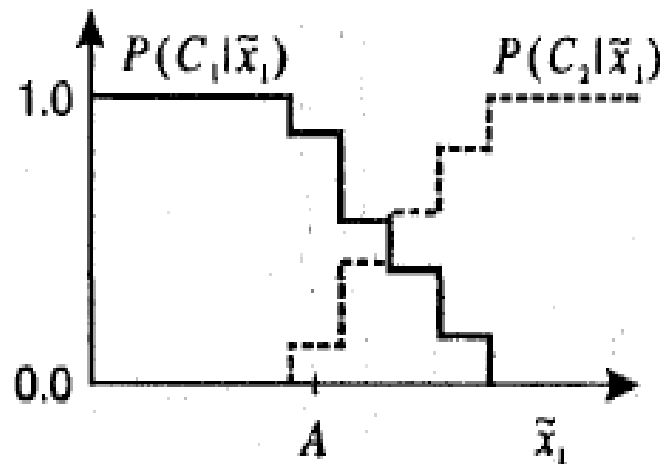


Figure 1.14. Histogram plot of posterior probabilities, corresponding to the histogram of observations in Figure 1.2, for prior probabilities  $P(C_1) = 0.6$  and  $P(C_2) = 0.4$ .

# Optimal Decision Boundary

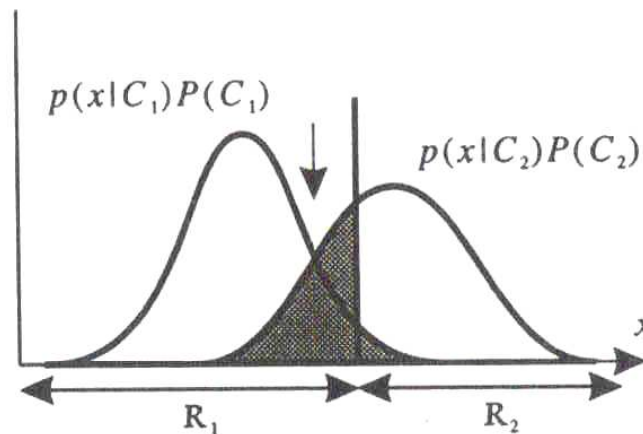


Figure 1.15. Schematic illustration of the joint probability densities, given by  $p(x, C_k) = p(x|C_k)P(C_k)$ , as a function of a feature value  $x$ , for two classes  $C_1$  and  $C_2$ . If the vertical line is used as the decision boundary then the classification errors arise from the shaded region. By placing the decision boundary at the point where the two probability density curves cross (shown by the arrow), the probability of misclassification is minimized.

# Optimal Decision Boundaries

- In general, assign a pattern  $x$  to  $C_k$  for  $k$  making  $P(C_k|x)$  biggest
- In other words, assign a pattern  $x$  to  $C_k$  if:

$$P(C_k|x) > P(C_j|x) \quad \text{for all } j \neq k.$$

- This generates  $c$  decision regions  $R_1 \dots R_c$  such that a point falling in region  $R_k$  is assigned to class  $C_k$ .
- Note that each of these regions need not be contiguous, but may itself be divided into several disjoint regions all of which are associated with the same class. The boundaries between these regions are known as *decision surfaces* or *decision boundaries*.

# Discriminant Functions

- Instead of *posterior probabilities* we could use other *discriminant functions*  $y_1(x), \dots, y_c(x)$  such that an input vector  $\mathbf{x}$  is assigned to class  $C_k$  if:

$$y_k(x) > y_j(x) \text{ for all } j \neq k$$

- Examples:
  - $y_k(x) = P(C_k | x)$  (posterior probability)
  - $y_k(x) = P(x | C_k) * P(C_k)$  (normalization not needed)
  - $y_k(x) = \ln(P(x | C_k) + \ln P(C_k))$  (monotonic transformation doesn't affect final decisions!!!)
- When there are only two classes ( $C_1$  and  $C_2$ ) we often use  $y(x) = y_1(x) - y_2(x)$  as a discriminant; then the sign of  $y(x)$  decides on class:
  - if  $y(x) > 0$  then  $x$  in  $C_1$  else  $x$  in  $C_2$

# Loss Matrix and Risk

- In some situations misclassifying “a” as “b” may **cost** much more than misclassifying “b” as “a” (think about “fraud” and “non-fraud” or “cancer” and “no-cancer” problems...).
- Usually we use then a **Loss Matrix** (or Cost Matrix) that quantify these costs. For example:

$$L_{12}=2; L_{21}=1; L_{11}=L_{22}=0,$$

where  $L_{ij}$  denotes the cost of “classifying” an object from class  $i$  as object from class  $j$  (correctly or incorrectly).

- Then, instead of minimizing the number of misclassified cases, we are interested in minimizing the **EXPECTED COSTS** of misclassified cases.

# Minimizing Risk in our Example

Loss Matrix:  $L_{ab}=2$ ;  $L_{ba}=1$ ;  $L_{aa}=L_{bb}=0$ ,

Consider a pattern  $x$ . How should we label it, given  $P(C1|x)$  and  $P(C2|x)$ , to minimize expected loss on  $x$ ?

Each  $x$  is either:

“a” with probability  $P(C1|x)$ , or

“b” with probability  $P(C2|x)$ , thus:

if we say “a” then the expected loss is:  $P(C1|x)*L_{aa} + P(C2|x)*L_{ba}$

if we say “b” then the expected loss is:  $P(C1|x)*L_{ab} + P(C2|x)*L_{bb}$

**TAKE THE CHEAPER ALTERNATIVE!**

# Continuous valued attributes

$$P(x \in [a, b]) = \int_a^b p(x) dx.$$

We use an upper-case  $P$  for *probabilities* and a lower-case  $p$  for *probability densities*

For continuous variables, the class-conditional probabilities introduced earlier become class-conditional probability density functions (pdf's), which we write in the form  $p(x|C_k)$ .

# Multiple attributes

- If there are  $d$  variables/attributes  $x_1, \dots, x_d$ , we may group them into a vector  $\mathbf{x} = [x_1, \dots, x_d]^T$  corresponding to a point in a  $d$ -dimensional space.
- The distribution of values of  $\mathbf{x}$  can be described by probability density function  $p(\mathbf{x})$ , such that the probability of  $\mathbf{x}$  lying in a region  $R$  of the  $\mathbf{x}$ -space is given by

$$P(\mathbf{x} \in \mathcal{R}) = \int_{\mathcal{R}} p(\mathbf{x}) d\mathbf{x}.$$



# Bayes Theorem

- For continuous variables, the prior probabilities can be combined with the class conditional **densities** to give the posterior probabilities  $P(C_k|x)$  using Bayes theorem:

$$P(C_k|x) = \frac{p(x|C_k)P(C_k)}{p(x)}.$$

- Note that you can show (and generalize to k classes):

$$p(x) = p(x|C_1)P(C_1) + p(x|C_2)P(C_2).$$

# Minimizing Risk in general case

- This risk is minimized if each pattern  $\mathbf{x}$  is assigned to a class  $j$  with the smallest expected loss, i.e., for each another class  $i$ , the corresponding expected risk is bigger:

$$\sum_{k=1}^c L_{kj} p(\mathbf{x}|\mathcal{C}_k) P(\mathcal{C}_k) < \sum_{k=1}^c L_{ki} p(\mathbf{x}|\mathcal{C}_k) P(\mathcal{C}_k) \quad \text{for all } i \neq j$$

## THE OPTIMALITY OF BAESIAN CLASSIFIER:

Once we know prior & class conditional probability distributions we can assign class labels in an **optimal** way, i.e., minimizing “classification error” or “loss”.

*Proof not required (check the book).*

# Key Points

---

1. Pattern, feature, class, classifier, decision boundary, discriminant function, loss matrix, risk (expected misclassification cost), error (expected misclassification rate), train and test sets, over-fitting, regularization
2. Bayes Formula, prior, posterior, conditional probabilities, likelihood,  $P$  (probability) and  $p$  (probability density)
3. Optimal Decision Rules; Expected Risk;

If we could only estimate probabilities we would be done ...

Estimating probabilities => next lecture

If we can't estimate probs => just find decision boundaries ...

# REGULARIZATION

Regularization is a powerful technique of limiting overfitting.

The key idea: somehow enforce the absolute values of model parameters to be relatively small (in our case: coefficients of the polynomial).

For example: add to the error function an extra term: “the sum of squared coefficients of your model”:

$$\lambda \sum_{i=0}^M w_i^2$$

where  $\lambda$  a tunable parameter that controls the size “punishment” for too big values of coefficients.

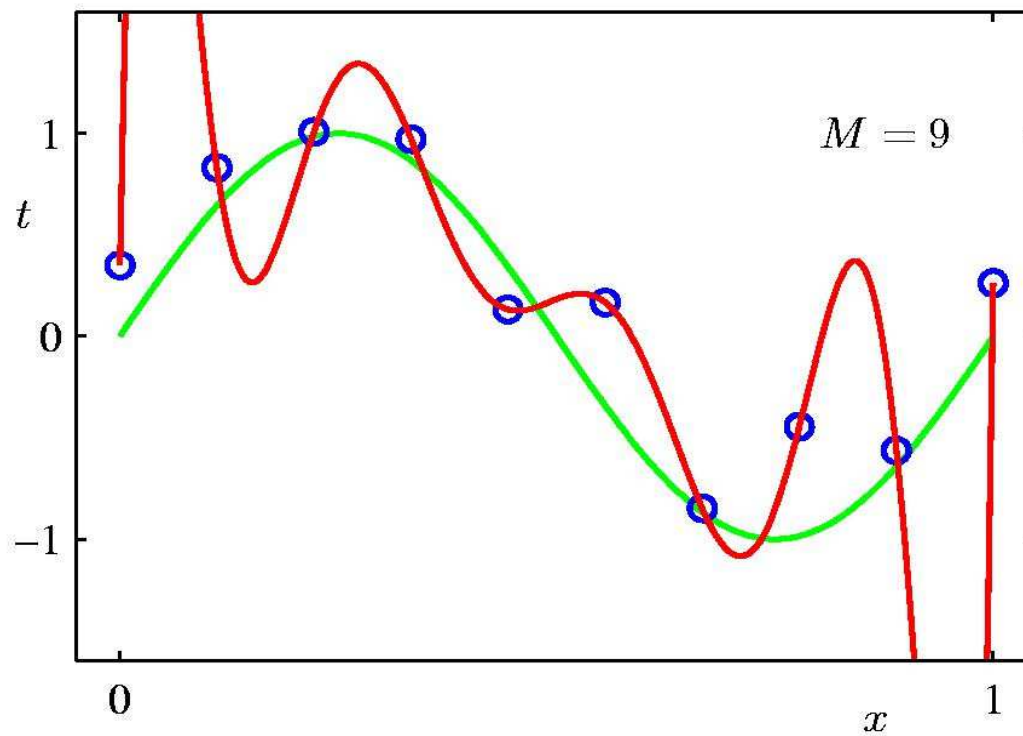
# Regularization

- **Penalize large coefficient values**

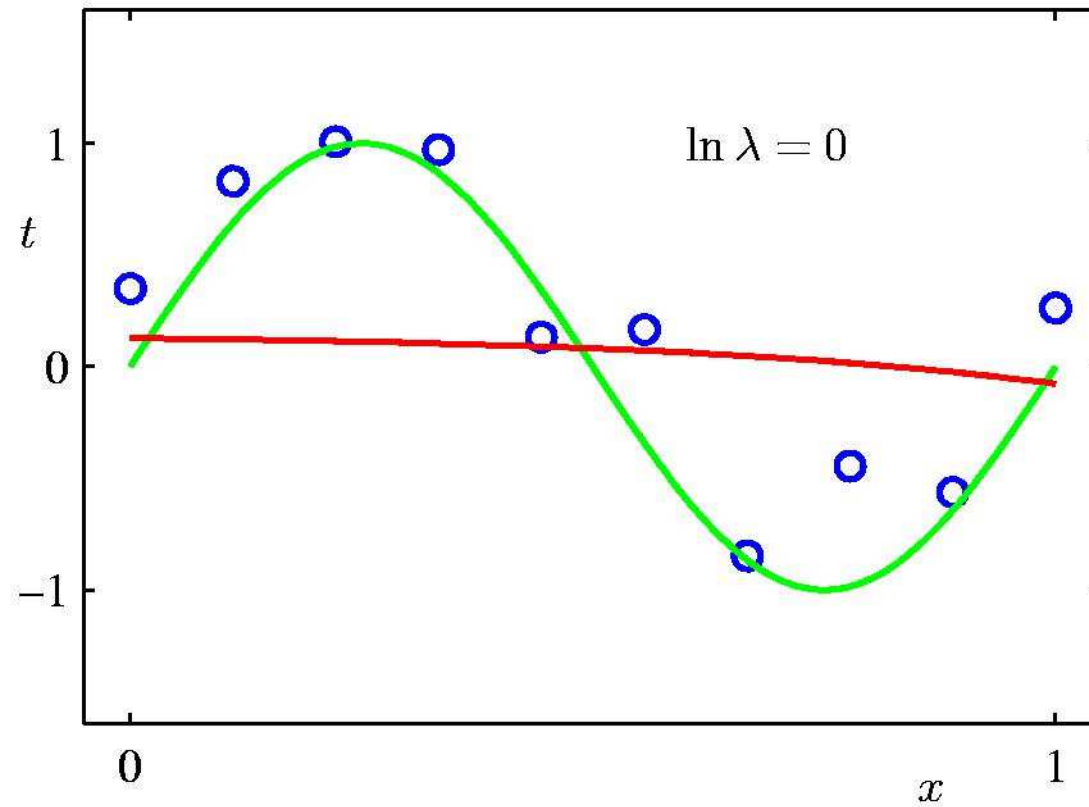
$$\tilde{E}(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^N \{y(x_n, \mathbf{w}) - t_n\}^2 + \frac{\lambda}{2} \|\mathbf{w}\|^2$$

- **Minimize training error while keeping the weights small. This is known as:**
  - shrinkage,
  - ridge regression,
  - weight decay (neural networks)

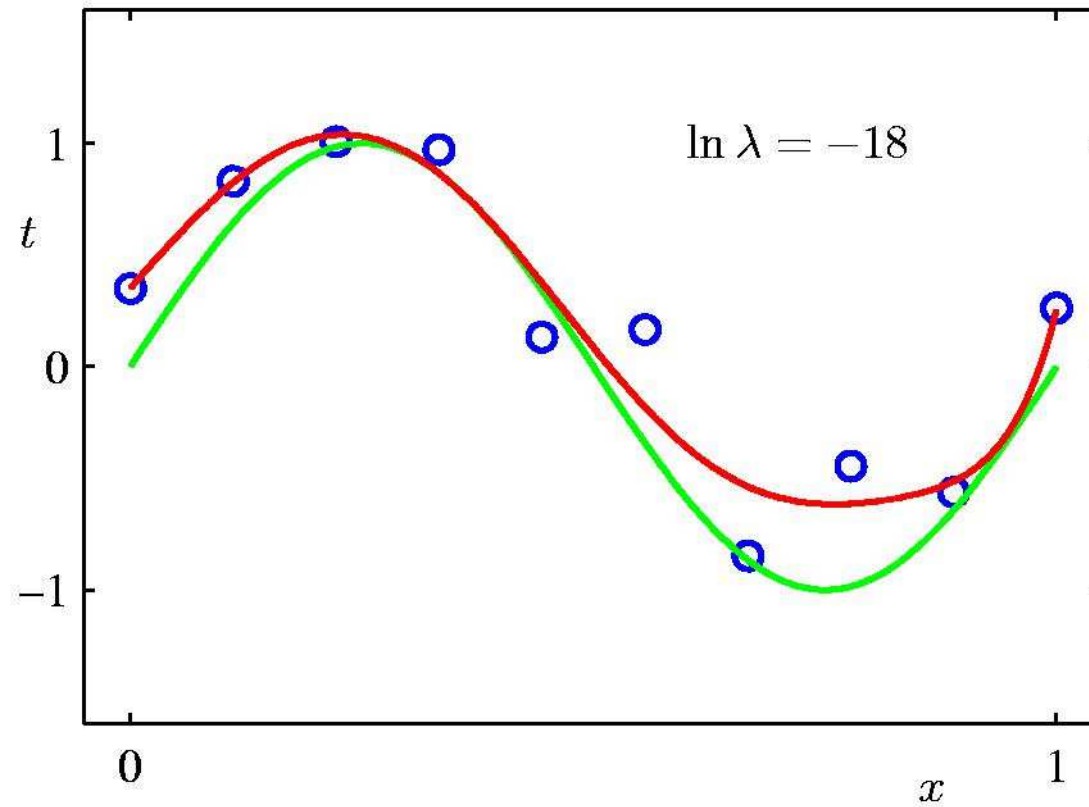
# Regularization ( $M=9$ ; $\lambda=0$ )



# Regularization (d=9; $\lambda=1$ )

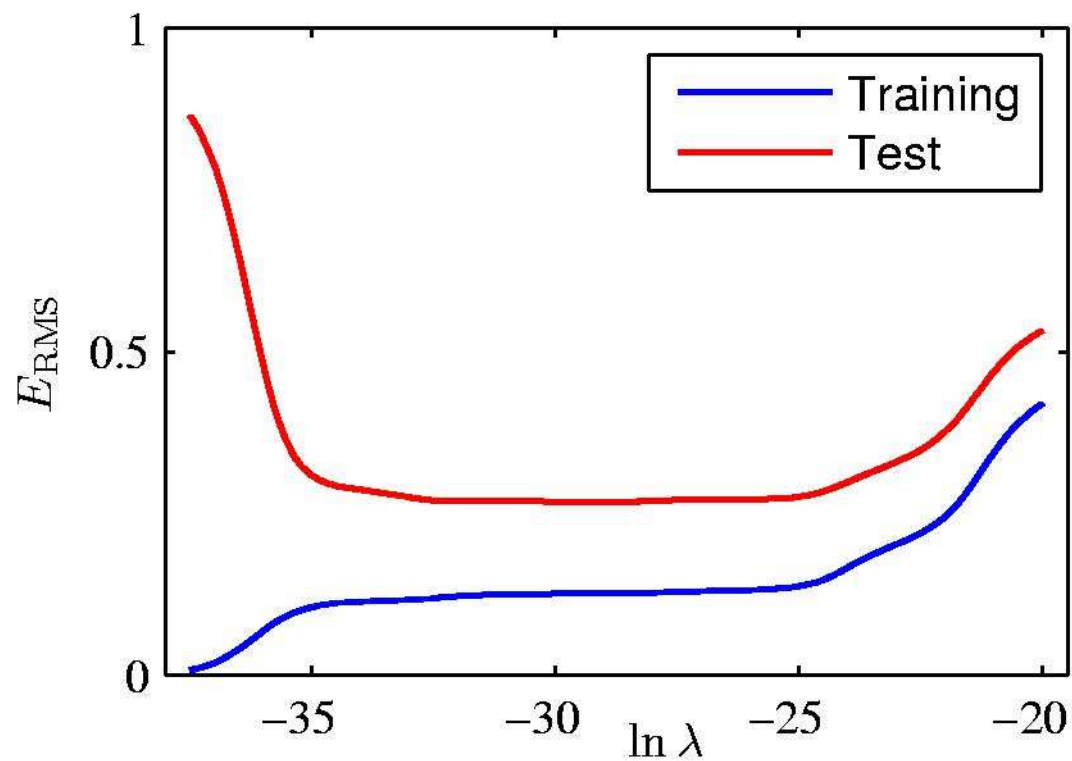


# Regularization (M=9; $\lambda=1.5230\text{e-}08$ )





# Regularization: error vs. lambda



# Probability Density Estimation

---

*Traditional approach to pattern recognition:*

1) estimate (from data)  $P(C_i|x)$  (inference)

2) make optimal decisions

*In practice we estimate  $p(x|C_i)$  rather than  $P(C_i|x)$  !*

- Parametric methods and ML estimates
- Semi-parametric methods: Mixture Models and EM
- Non-parametric methods:
  - histograms
  - kernel methods
  - k-nearest neighbors

# Parametric density estimation

---

## Key idea:

- Assume that  $p(x)$  can be expressed by a formula that involves some parameters  $\Theta$ ,  $p(x, \Theta)$  (e.g.,  $\text{Normal}(\mu, \sigma)$ ,  $\text{Poisson}(\lambda)$ , etc.)
- Find (using your data) "optimal" values of these parameters
- "OPTIMAL" = Maximum Likelihood Principle:  
"optimal values are those that maximize the "likelihood" of observing data":

$$L(\Theta) = \prod_{x \in X} p(x | \Theta) \quad (\text{Likelihood (maximize!)})$$

$$-\ln L(\Theta) = -\sum_{x \in X} \ln p(x | \Theta) \quad (\text{Negative Log Likelihood (minimize!)})$$

- For most known distributions formulas for optimal values are known

# Example: Normal Distribution

---

Consider a set of 10 numbers:

0.8165 0.7627 0.7075 0.7352 0.6303 0.8696 0.7059 0.8797 0.7264 0.7872

Assuming that they come from a normal distribution with unknown parameters  $\mu$  and  $\sigma$ , how can we find “most likely” values of these parameters?

For  $\mu=0.5$  and  $\sigma=0.1$  we get `[>> pdfs=normpdf(x,0.5,0.1)]`:

0.0267 0.1266 0.4633 0.2512 1.7059 0.0043 0.4789 0.0030 0.3075 0.0646

Their product is 8.1093e-011 (very unlikely!)

For  $\mu=0.6$  and  $\sigma=0.2$  we get `[>> pdfs=normpdf(x,0.6,0.2)]`:

1.1103 1.4329 1.7264 1.5875 1.9719 0.8040 1.7338 0.7502 1.6336 1.2874

Their product is 18.9067 (more likely!)

# Example: Normal Distribution

Trying  $\mu=0.1:0.1:1$  and  $\sigma=0.1:0.1:1$  we get the matrix  
 $\log(\text{prod}(\text{normpdf}(x,\mu,\sigma)))$ :

-208.0754	-48.5730	-21.8065	-13.8960	-11.1344	-10.2453	-10.1514	-10.4253	-10.8754	-11.4085
-146.8654	-33.2705	-15.0054	-10.0703	-8.6860	-8.5451	-8.9023	-9.4689	-10.1198	-10.7964
-95.6554	-20.4680	-9.3154	-6.8697	-6.6376	-7.1226	-7.8572	-8.6688	-9.4875	-10.2843
-54.4454	-10.1655	-4.7365	-4.2941	-4.9892	-5.9778	-7.0161	-8.0249	-8.9788	-9.8722
-23.2354	-2.3630	-1.2688	-2.3435	-3.7408	-5.1109	-6.3792	-7.5372	-8.5935	-9.5601
-2.0254	2.9395	1.0879	-1.0178	-2.8924	-4.5217	-5.9463	-7.2058	-8.3316	-9.3480
9.1845	5.7420	2.3335	-0.3172	-2.4440	-4.2103	-5.7176	-7.0306	-8.1932	-9.2359
10.3945	6.0445	2.4679	-0.2416	-2.3956	-4.1767	-5.6929	-7.0117	-8.1783	-9.2238
1.6045	3.8470	1.4912	-0.7910	-2.7472	-4.4209	-5.8723	-7.1491	-8.2868	-9.3117
-17.1855	-0.8505	-0.5965	-1.9654	-3.4988	-4.9429	-6.2557	-7.4427	-8.5188	-9.4996

Thus  $\mu$  around 0.7-0.8 and  $\sigma$  around 0.1 are good guesses...

In general we have to find an optimum of a function of two variables:  $\mu$ ,  $\sigma$ .

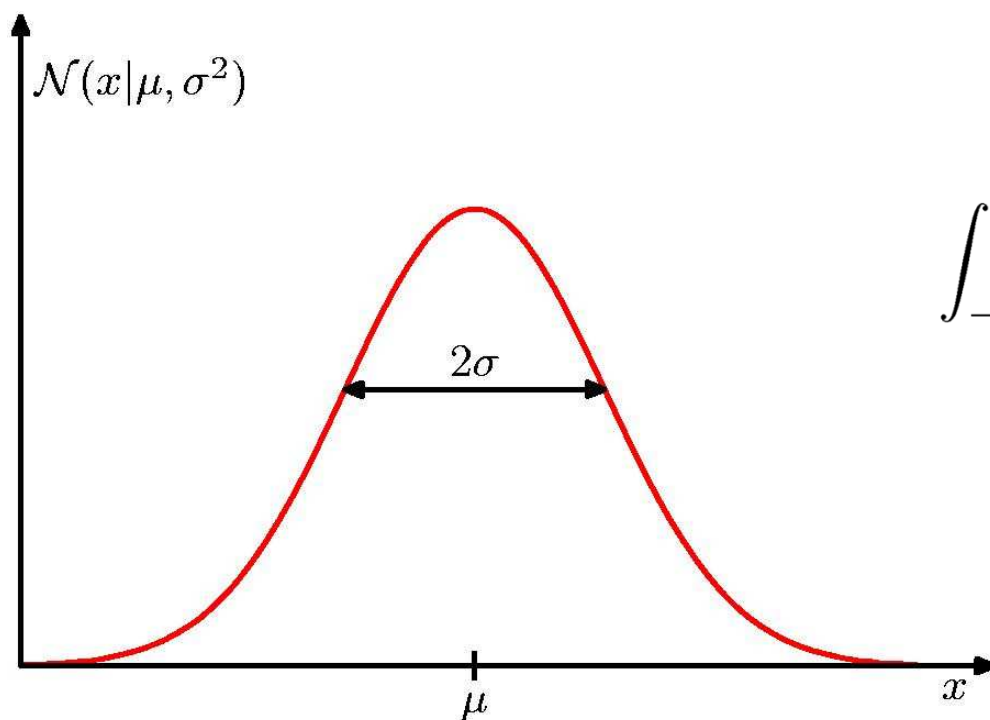
**Accidentally (!?),** it happens that this optimum is reached at:

$\mu = \text{mean}(x)$ ;  $\sigma = \text{std}(x)$

[In our case:  $\mu = 0.7621$ ;  $\sigma = 0.0778$ ]

# The Gaussian Distribution

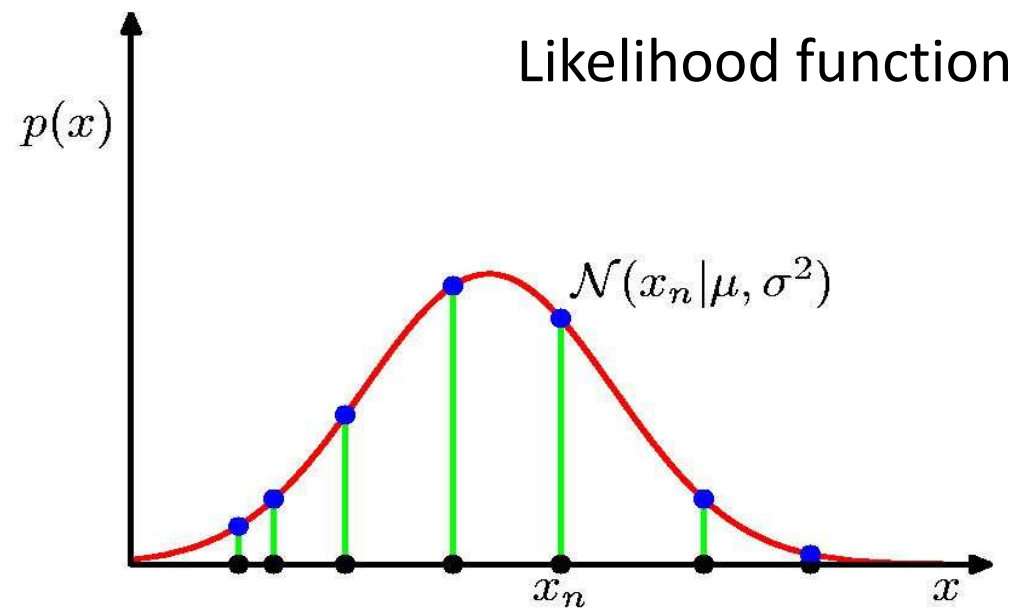
$$\mathcal{N}(x|\mu, \sigma^2) = \frac{1}{(2\pi\sigma^2)^{1/2}} \exp \left\{ -\frac{1}{2\sigma^2} (x - \mu)^2 \right\}$$



$$\mathcal{N}(x|\mu, \sigma^2) > 0$$

$$\int_{-\infty}^{\infty} \mathcal{N}(x|\mu, \sigma^2) dx = 1$$

# Gaussian Parameter Estimation



$$p(\mathbf{x} | \mu, \sigma^2) = \prod_{n=1}^N \mathcal{N}(x_n | \mu, \sigma^2)$$

# Maximum (Log) Likelihood

$$\mathcal{N}(x|\mu, \sigma^2) = \frac{1}{(2\pi\sigma^2)^{1/2}} \exp \left\{ -\frac{1}{2\sigma^2}(x - \mu)^2 \right\}$$

$$\ln p(\mathbf{x}|\mu, \sigma^2) = -\frac{1}{2\sigma^2} \sum_{n=1}^N (x_n - \mu)^2 - \frac{N}{2} \ln \sigma^2 - \frac{N}{2} \ln(2\pi)$$

$$\mu_{\text{ML}} = \frac{1}{N} \sum_{n=1}^N x_n \qquad \sigma_{\text{ML}}^2 = \frac{1}{N} \sum_{n=1}^N (x_n - \mu_{\text{ML}})^2$$

The likelihood function is a function of mu and sigma, so its minimum can be found analytically (***try it!!!***)



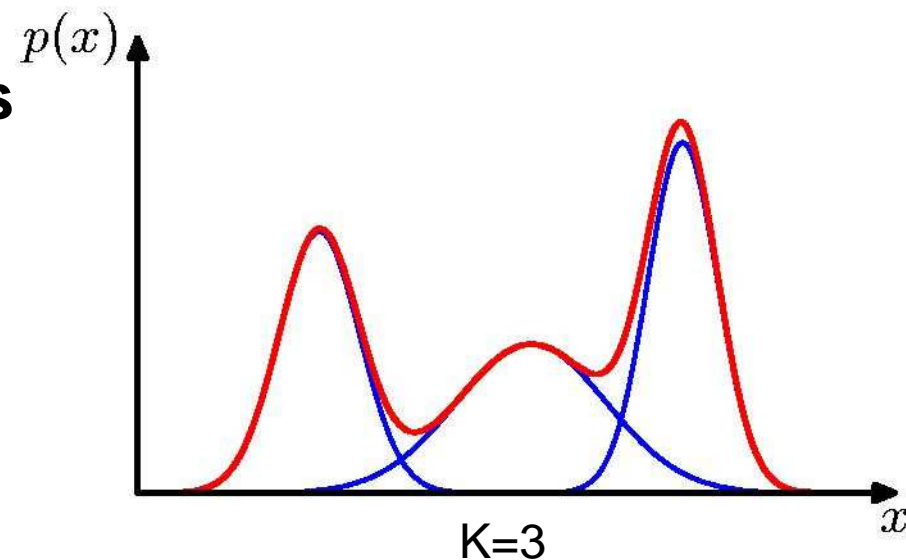
# Mixtures of Gaussians

**Combine simple models  
into a complex model:**

$$p(\mathbf{x}) = \sum_{k=1}^K \pi_k \underbrace{\mathcal{N}(\mathbf{x} | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}_{\text{Component}}$$

Mixing coefficient

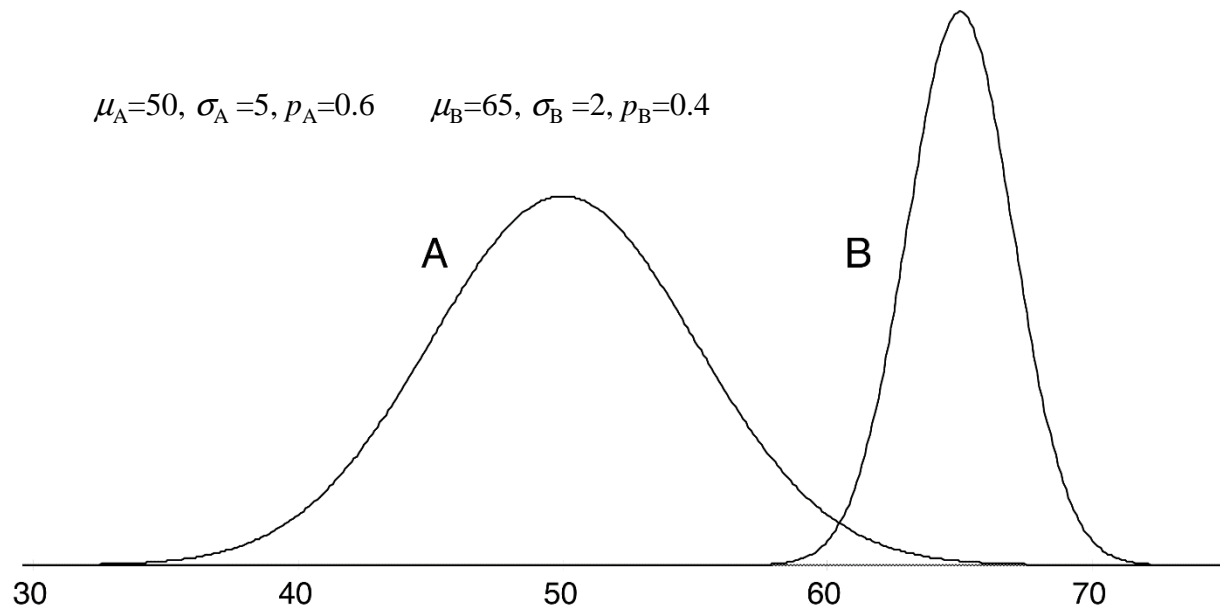
$$\forall k : \pi_k \geq 0 \quad \sum_{k=1}^K \pi_k = 1$$



**Expectation Maximization (EM):** the main algorithm  
for estimating parameters of mixtures

# An example of a mixture models

data					
A	51	B	62	B	64
A	43	A	47	A	51
B	62	A	52	A	52
B	64	B	64	B	62
A	45	A	51	A	49
A	42	B	65	A	48
A	46	A	48	B	62
A	45	A	49	A	43
A	45	A	46	A	40
model					



# Problem formulation

## given:

- data:  $x_1, x_2, \dots$  (measurements)
- “meta-knowledge”: the data is a mixture of 2 normal distributions  $N(m_A, s_A), N(m_B, s_B)$

## problem:

find values of  $p_A, m_A, s_A$  and  $p_B, m_B, s_B$   
(without knowing the labels!!!)

## How???

=> Expectation Maximization Algorithm

# Expectation Maximization Algorithm (EM)

## Initialization:

start with *some* values of all parameters

## Iteration:

- E-step:

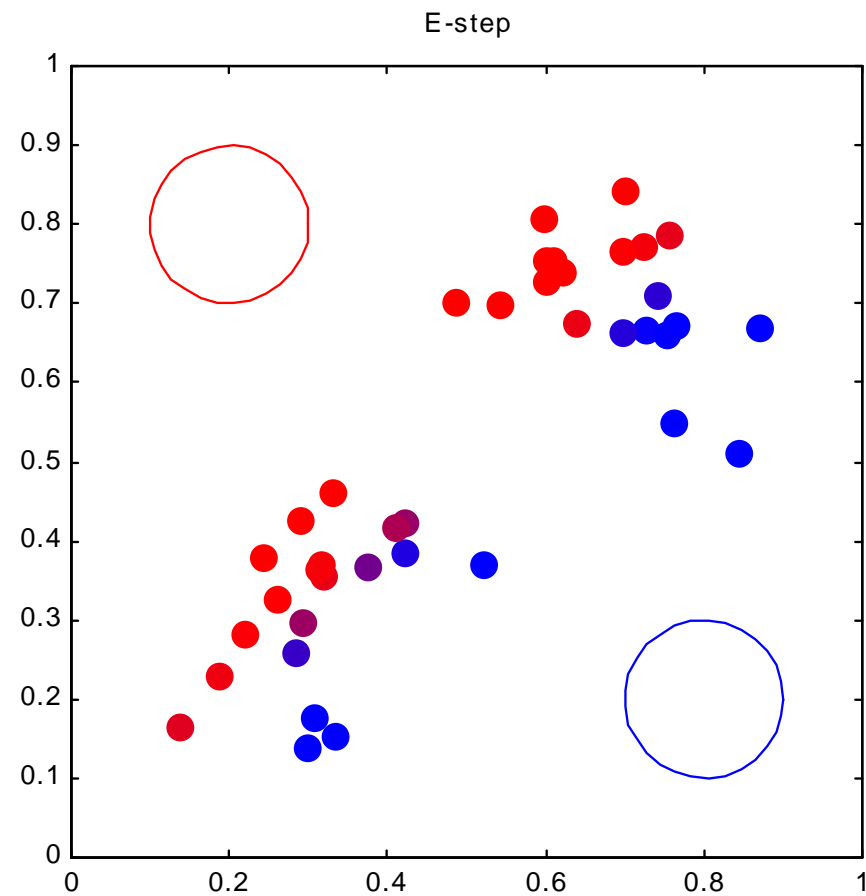
given parameter values calculate “probabilistic labels”:

for each observation  $x$  find  $P(A|x)$  and  $P(B|x)$

- M-step:

given “probabilistic labels” re-compute values of all parameters (mixing coefficients, means, std's)

## Demo EM in Matlab (demgmm1.m)



# Expectation Maximization Algorithm (EM)

## Initialization:

start with *some* values of  $p_A, m_A, s_A$  and  $p_B, m_B, s_B$   
(just guess some values!)

## Iteration:

- E-step:

calculate  $P(\text{class}|x)$ , where class is A or B

$P(\text{class}|x) = p(x|\text{class}) * P(\text{class}) / p(x)$ ; i.e.:

$$P(A|x) = p_A * p(x|A) = p_A * N(x; m_A, s_A)$$

$$P(B|x) = p_B * p(x|B) = p_B * N(x; m_B, s_B)$$

Normalize both terms to make sure that they sum up to 1!

# Expectation Maximization Algorithm (EM)

M-step:

given “probabilistic labels” find new values of  
parameters:  $p_A$ ,  $m_A$ ,  $s_A$  and  $p_B$ ,  $m_B$ ,  $s_B$

$$p_A = \text{sum}(P(A|x))/N; \quad p_B = \text{sum}(P(B|x))/N;$$

$$m_A = \text{sum}(P(A|x)*x)/\text{sum}(P(A|x));$$

$$m_B = \text{sum}(P(B|x)*x)/\text{sum}(P(B|x));$$

$$s_A = \text{sum}(P(A|x)*(x - m_A)^2)/\text{sum}(P(A|x)); \quad s_A = \text{sqrt}(s_A);$$

$$s_B = \text{sum}(P(B|x)*(x - m_B)^2)/\text{sum}(P(B|x)); \quad s_B = \text{sqrt}(s_B);$$

# EM Algorithm: summary

Initialization: set parameters at random

Repeat:

E-step:

calculate  $P(\text{class}|\mathbf{x})$ , where class is A or B  
(probabilities of being in A or B)

M-step: re-estimate model parameters

given “probabilistic labels” find new values of  
parameters:  $p_A, m_A, s_A$  and  $p_B, m_B, s_B$

till convergence



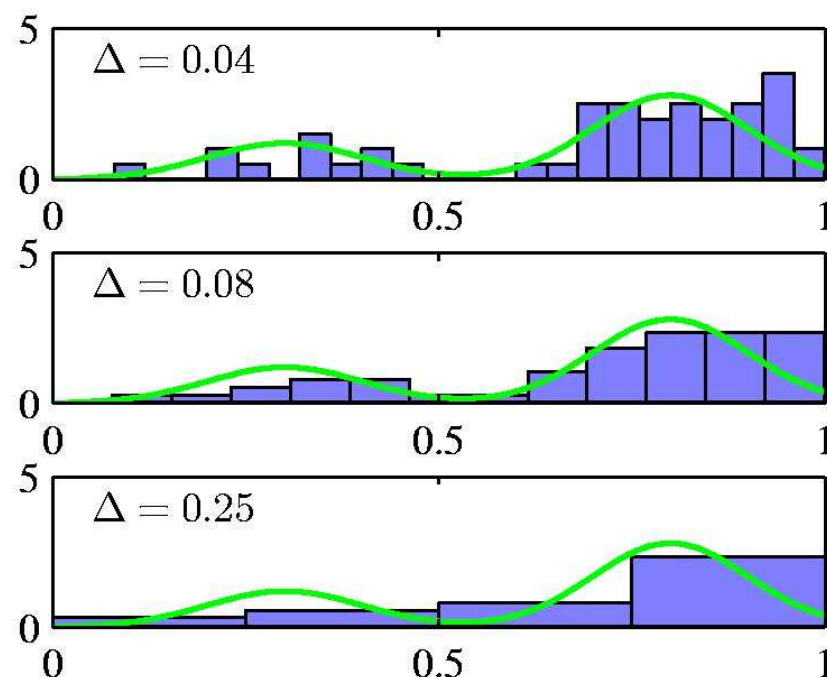
# Histograms

- Histogram methods partition the data space into distinct bins with widths  $\Delta_i$  and count the number of observations,  $n_i$ , in each bin.

$$p_i = \frac{n_i}{N \Delta_i}$$

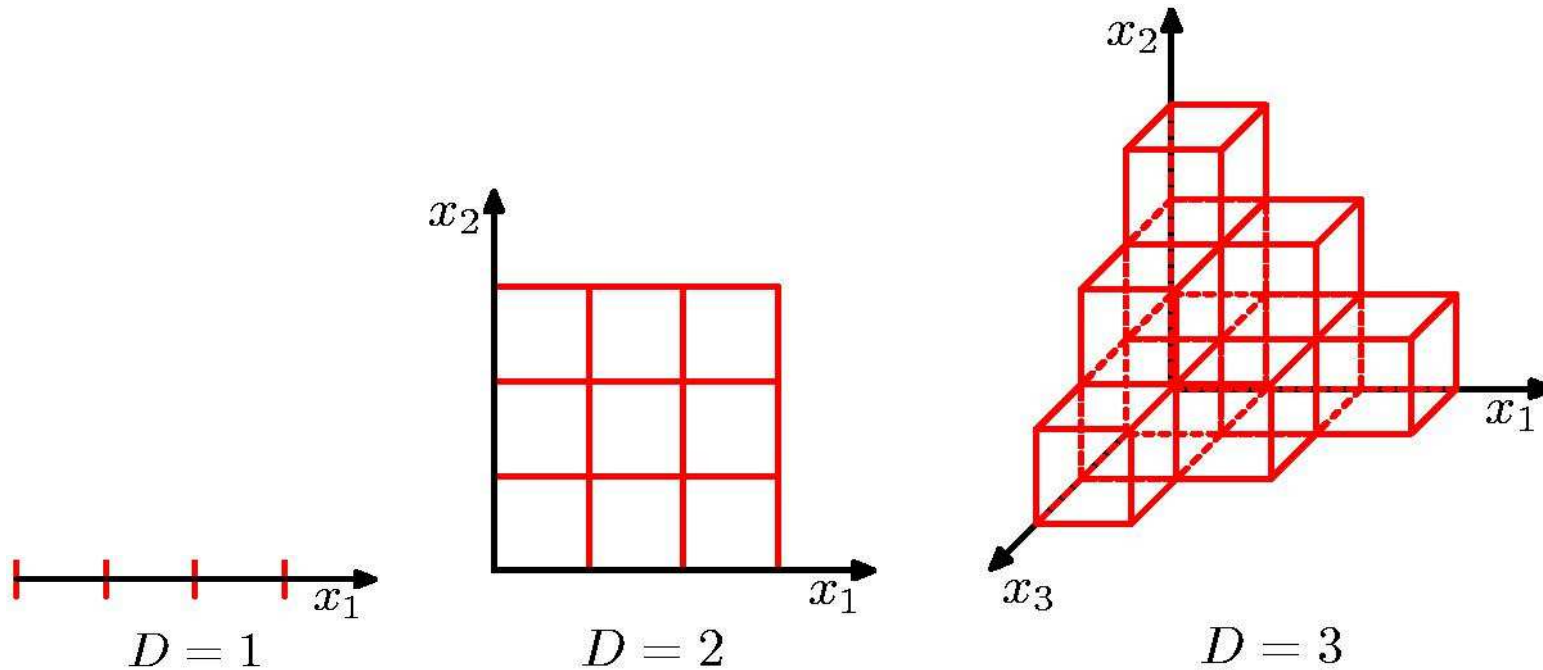
- Often, the same width is used for all bins,  $\Delta_i = \Delta$ .
- $\Delta$  acts as a smoothing parameter (*how to choose it?*)
- Doesn't work for highly dimensional data!!!

Neural Networks



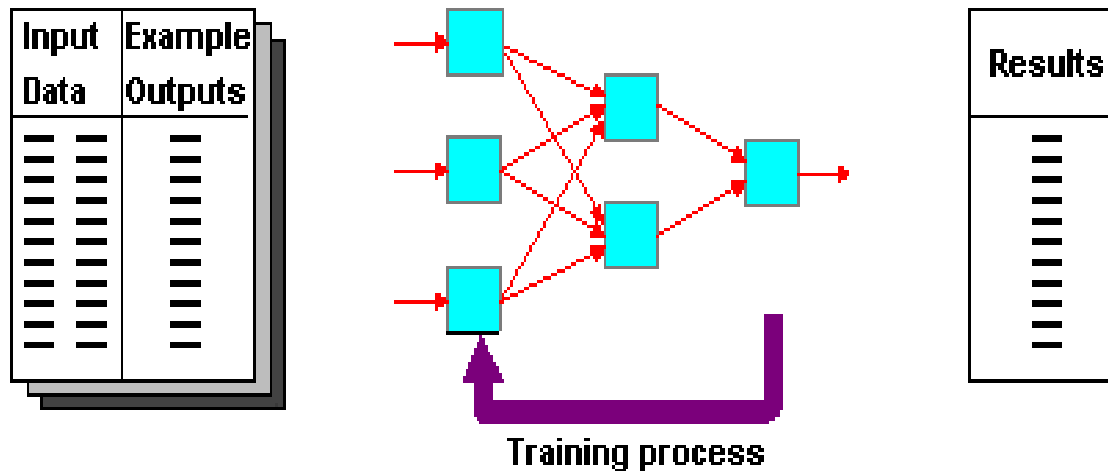
- In a D-dimensional space, using M bins in each dimension will require  $M^D$  bins!

# Curse of Dimensionality



# Linear Models

- Training and test data sets
- Training set: input & target
- Test set: only input



# Single Layer Networks (Linear Models)

---

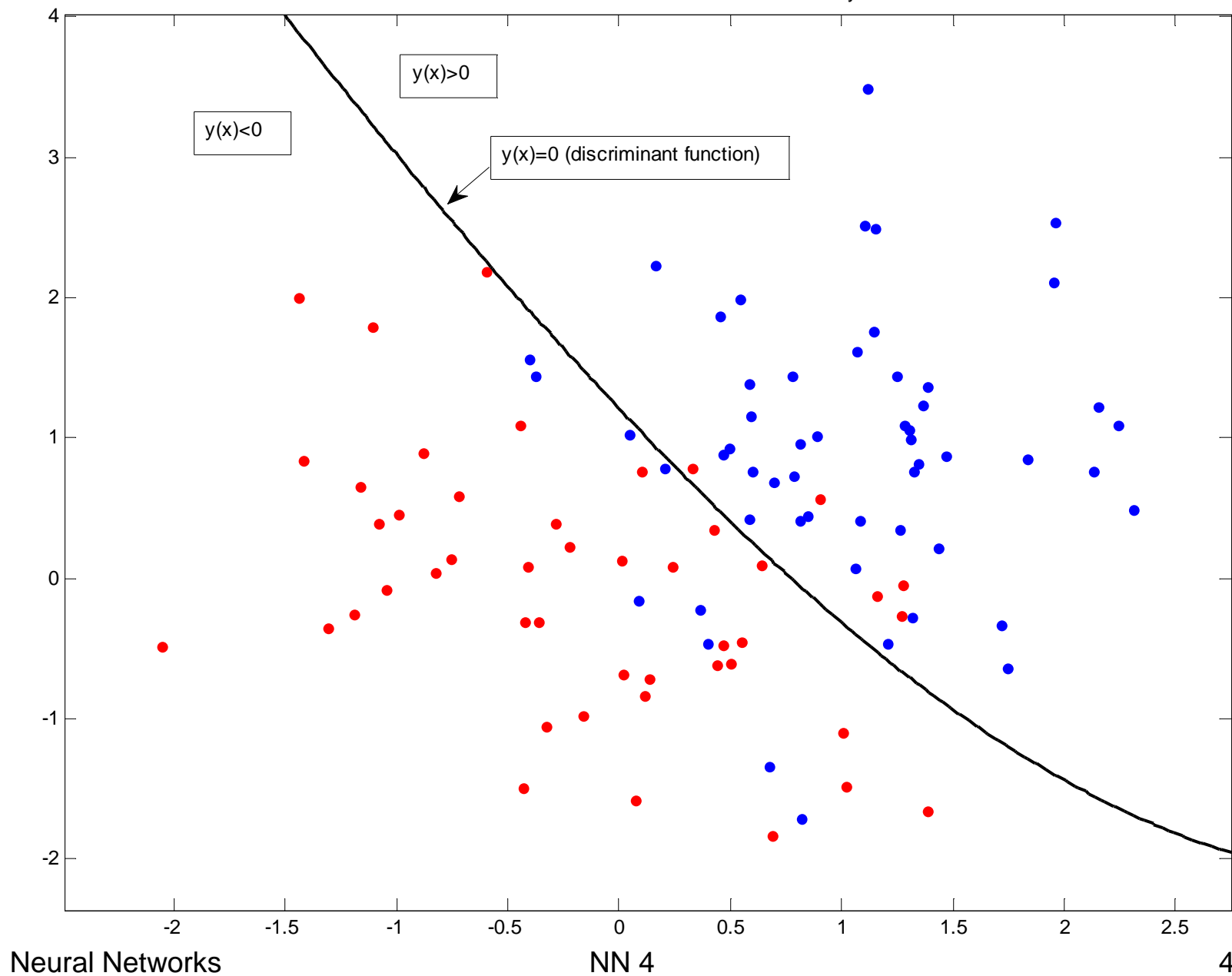
- Linear Discriminants
- Single Layer Perceptron
- Linear Separability and Cover's Theorem
- Learning Algorithms:
  - Perceptron Rule and Convergence Theorem
  - Pocket Algorithm
  - Least-Squares Method (Adaline)
  - Gradient Descent and Logistic Regression
  - Support Vector Machines
- Generalized Linear Discriminants
- Multi-class perceptron learning algorithm

# Motivation



- Sometimes modeling  $P(C_i|X)$  is impossible (very few data points, very high dimensionality of data, lack of background knowledge, etc.)
- Then one can try to find class boundaries directly, assuming certain form of discriminant functions and optimizing a suitable error measure
- *The simplest case of a boundary: a line (a hyperplane)  
single layer networks (Perceptron; Adaline; GLM; SVM)*
- General case: multi-layer networks  
(Multi-layer Perceptron; RBF-networks)

A Classification Problem and a Class Boundary



# Discriminant Functions

- A *discriminant function for classes  $C_1$  and  $C_2$*  is any function  $y(\mathbf{x})$  such that an input vector  $\mathbf{x}$  is assigned to class  $C_1$  if  $y(\mathbf{x}) > 0$  (and to  $C_2$  if  $y(\mathbf{x}) < 0$ ) [what to do with ties?]
- In case of  $N$  classes, we need  $N$  discriminant functions  $y_1(\mathbf{x}), y_2(\mathbf{x}), \dots, y_N(\mathbf{x})$ , such the  $k$ -th function  $y_k(\mathbf{x})$  discriminates class  $C_k$  from all other classes, for  $k=1, \dots, N$ , i.e.,

$\mathbf{x}$  is assigned to class  $C_k$  iff  $y_k(\mathbf{x}) > y_n(\mathbf{x})$  for all  $k \neq n$

- Examples:
  - $y_k(\mathbf{x}) = P(C_k|\mathbf{x})$  (posterior probability)
  - $y_k(\mathbf{x}) = P(\mathbf{x}|C_k) * P(C_k)$  (normalization not needed)
  - $y_k(\mathbf{x}) = \ln(P(\mathbf{x}|C_k) + \ln P(C_k))$  (monotonic transformation doesn't affect final decisions!!!)
- In case of two classes ( $C_1$  and  $C_2$ ) we often use  $y(\mathbf{x}) = y_1(\mathbf{x}) - y_2(\mathbf{x})$  as a discriminant; then the sign of  $y(\mathbf{x})$  decides on the class:  
if  $y(\mathbf{x}) > 0$  then  $\mathbf{x}$  in  $C_1$  else  $\mathbf{x}$  in  $C_2$

# Linear Discriminant Functions

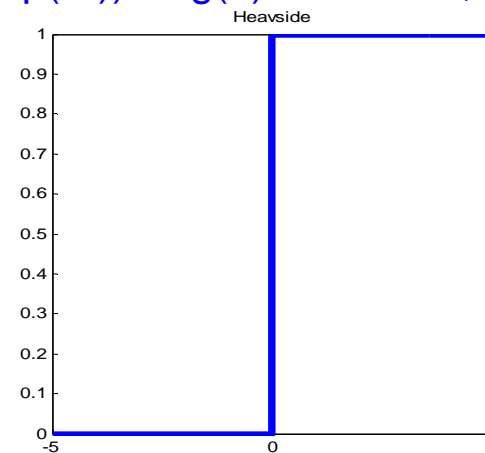
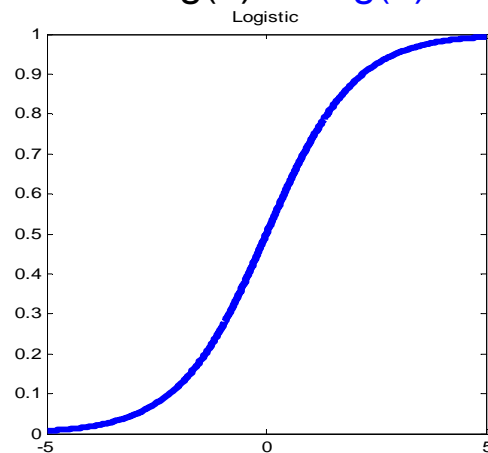
A *linear discriminant function in  $n$ -dimensional space* is a function  $y(\mathbf{x})$  in the form:

$$y(\mathbf{x}) = w_0 + w_1x_1 + \dots + w_nx_n = w_0 + \mathbf{w}^T \mathbf{x}$$

It defines a **point** (in  $\mathbb{R}^1$ ), a **line** (in  $\mathbb{R}^2$ ), a **plane** (in  $\mathbb{R}^3$ ), a **hyperplane** (in  $\mathbb{R}^n$ ), that splits the space into “positive” and “negative” half-spaces.

A composition of a linear discriminant  $y(\mathbf{x})$  with a **monotonic function**  $g(a)$  also defines a linear boundary:  $g(y(\mathbf{x})) > 0$  iff  $y(\mathbf{x}) > g^{-1}(0)$ .

Common choices of  $g(a)$  are:  $g(a) = 1/(1 + \exp(-a))$  or  $g(a) = 0$  for  $a < 0$ ; 1 otherwise

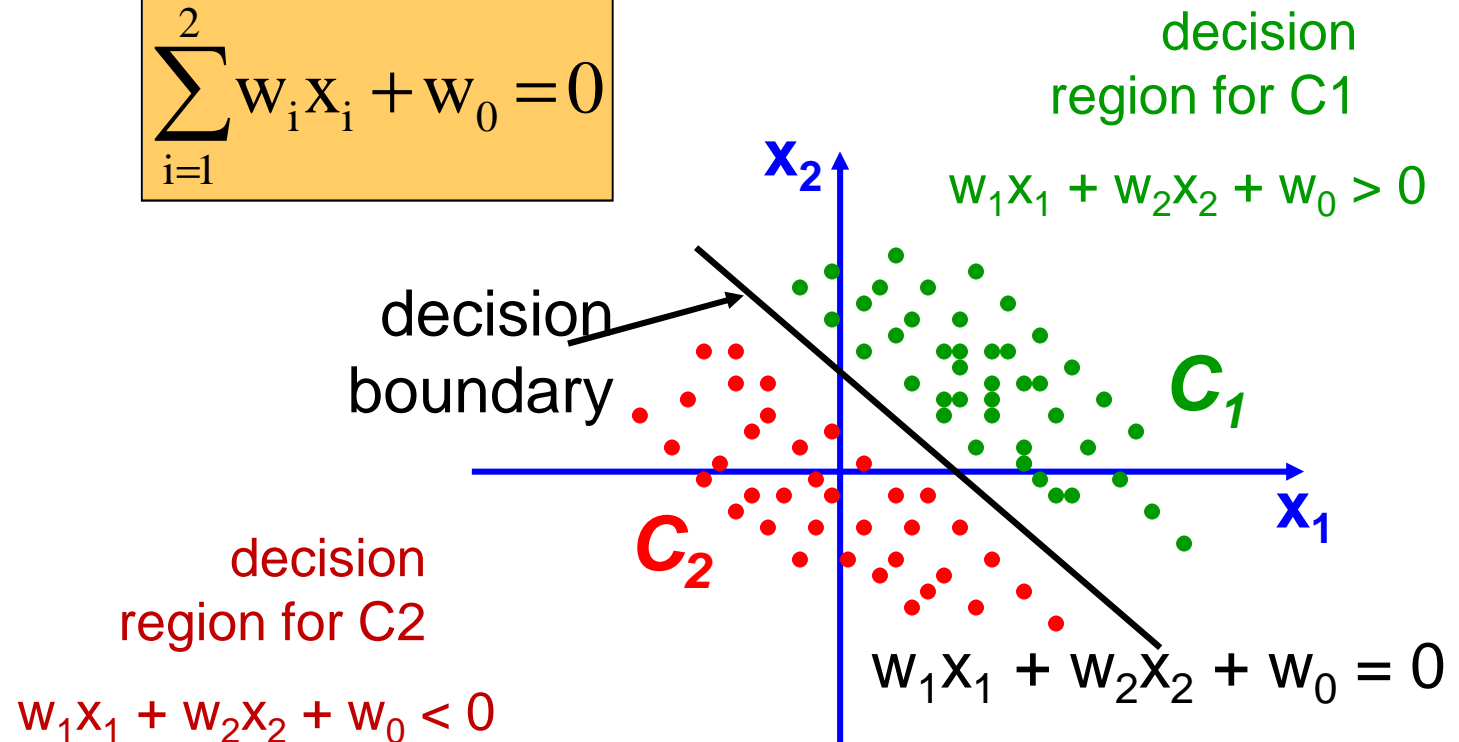




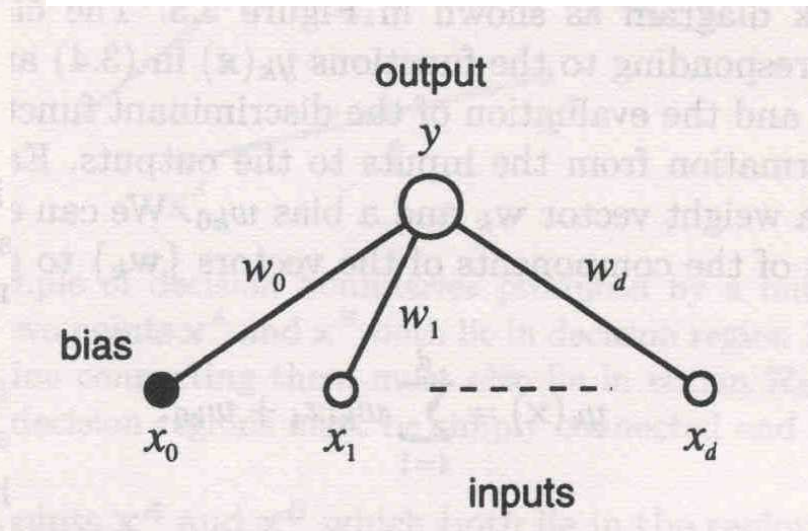
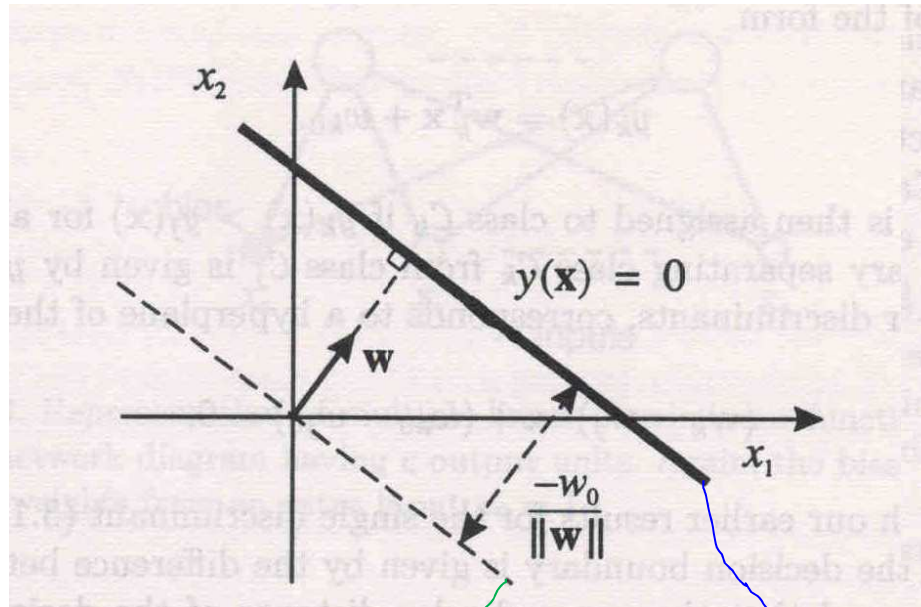
# Geometric View in 2D

The equation below describes a (hyper-)plane in the input space consisting of real valued 2D vectors. The plane splits the input space into two regions, each of them describing one class.

$$\sum_{i=1}^2 w_i x_i + w_0 = 0$$



# Linear Discriminant and Perceptron



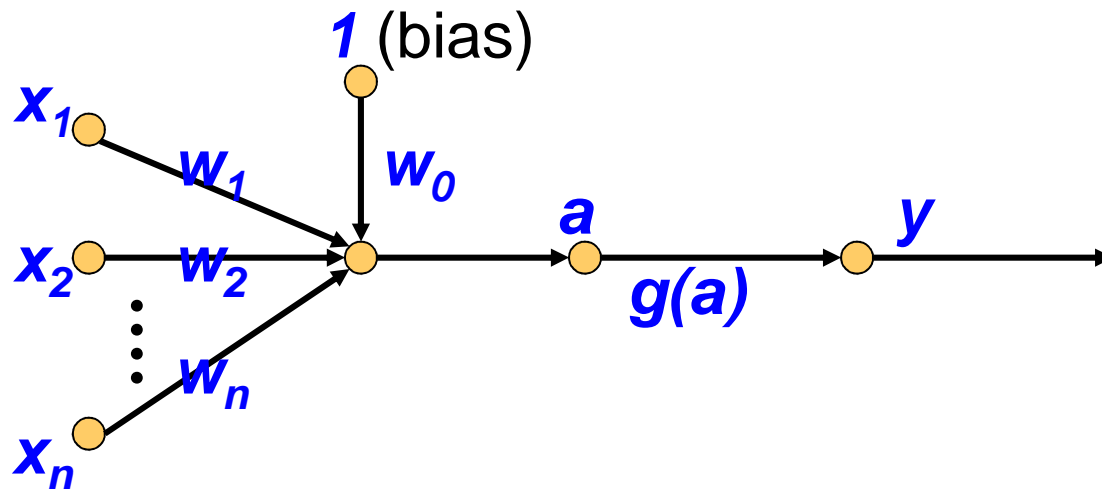
$w_1x_1 + w_2x_2 = 0$   
 $\langle w_1, w_2 \rangle$  is *perpendicular*  
to the boundary!

$$w_1x_1 + w_2x_2 + w_0 = 0$$

# Perceptron: McCulloch-Pitts Model

The (McCulloch-Pitts) **perceptron** is a single node NN (or a single layer NN) with a non-linear function  **$g(a)$** :

$$a = w_0 + \sum w_i x_i; \quad g(a) = \begin{cases} +1 & \text{if } a \geq 0 \\ -1 & \text{if } a < 0 \end{cases}$$



# Perceptron Training



- How can we train a perceptron for a classification task?
- We try to find suitable values for the **weights** in such a way that the training examples are correctly classified.
- Geometrically, we try to find a **hyper-plane** that separates the examples of the two classes.
- Two classes  $C_1$  and  $C_2$  are *linearly separable* if there exists a hyperplane that separates them.

# Perceptron learning algorithm

```
initialize w randomly;  
while (there are misclassified training examples)  
    Select a misclassified example (x, d)  
     $\mathbf{w}_{\text{new}} = \mathbf{w}_{\text{old}} + \eta d \mathbf{x};$   
end-while;  
 $\eta > 0$  is a learning rate parameter (step size);
```

## Motivation:

If **x** missclassified and  $d=1 \Rightarrow \mathbf{w}\mathbf{x}$  should be bigger,

If **x** missclassified and  $d=-1 \Rightarrow \mathbf{w}\mathbf{x}$  should be small

$$(\mathbf{w} + \eta d \mathbf{x})\mathbf{x} = \mathbf{w}\mathbf{x} + \eta d \mathbf{x}^2$$

This rule does exactly what we want ( $\mathbf{x}^2$  is  $>0$ ) !!:

## Conventions:

- **w** is a row vector;
- **x** is a column vector
- $\mathbf{x}^2$  is  $\mathbf{x}^T * \mathbf{x}$

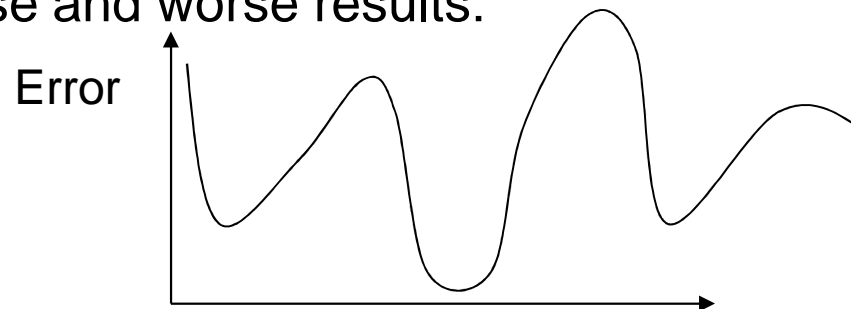
# Main properties

## 1) Perceptron Convergence Theorem:

If the classes  $C_1$ ,  $C_2$  are linearly separable (that is, there exists a hyper-plane that separates them) then the perceptron algorithm applied to  $C_1 \cup C_2$  terminates successfully after a finite number of iterations. [*the value of the learning rate  $\eta$  is not essential*]

## 2) Bad Behavior:

If the classes  $C_1$ ,  $C_2$  are **not** linearly separable then the perceptron algorithm may produce worse and worse results:



# Improvement: (Naive) Pocket Algorithm

0. Start with a random set of weights;  
put them in a 'pocket'
1. Select at random a pattern
2. If it is misclassified then
  - 2A. apply the perceptron update rule
  - 2B. check whether new weights are better than those kept in the pocket; if so put new weights to the pocket (and remove the old ones)*
3. goto 1.

Testing if new weights are better (less errors) is very expensive !!!

# Gallant's Pocket Algorithm

**main idea:** measure the quality of weights by counting the number of consecutive correct classifications ('current\_run')

0. Start with a random set of weights; put them in the 'pocket';  
*best\_run := 0; current\_run := 0;*
1. Select a pattern at random
2. If it is misclassified then  
    apply the update rule; *current\_run := 0;*  
    else  
        *current\_run++;*  
        if *current\_run > best\_run* then  
            put new weights to the pocket; *best\_run := current\_run*
3. Goto 1



# Gallant's Pocket Algorithm with ratchet

0. Start with a random setting of weights; put it in the 'pocket'

1. *best\_run:=0; current\_run:=0;*

2. Select at random a pattern

3. **IF** it is correctly classified **THEN**

*current\_run:=current\_run+1*

**IF** *best\_run < current\_run* **AND** current weights  
are better than those kept in the pocket **THEN**

*pocket:=current weights; best\_run=current\_run;*

**ELSE**

*current\_run=0;*

update weights



# Pocket convergence theorem

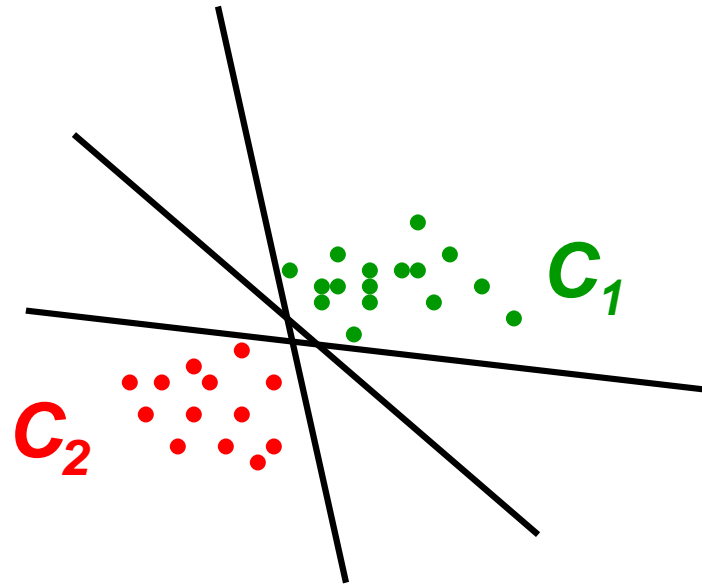
## **Pocket Convergence Theorem:**

The pocket algorithm converges with probability 1 to optimal weights (even if sets are not separable!)

## **Practice:**

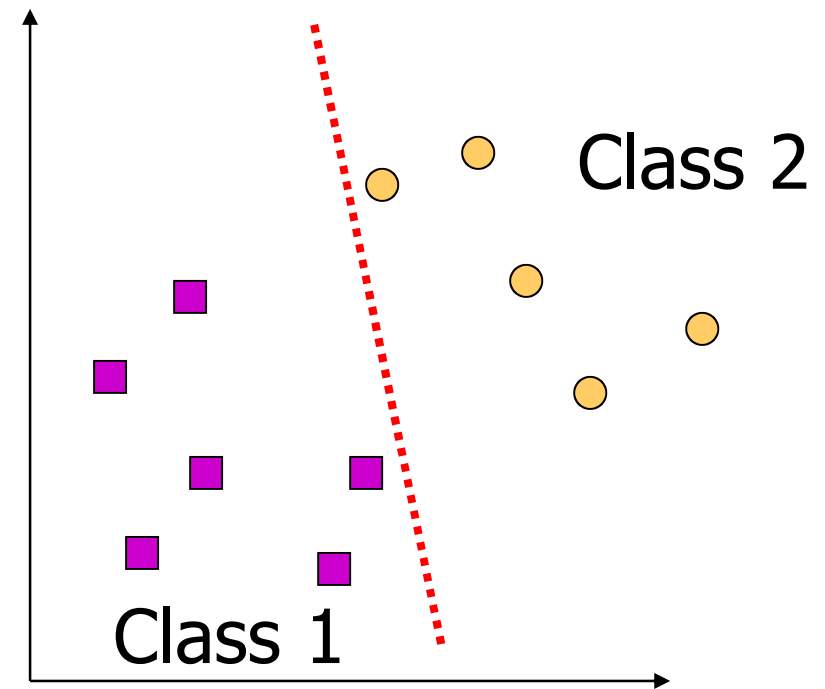
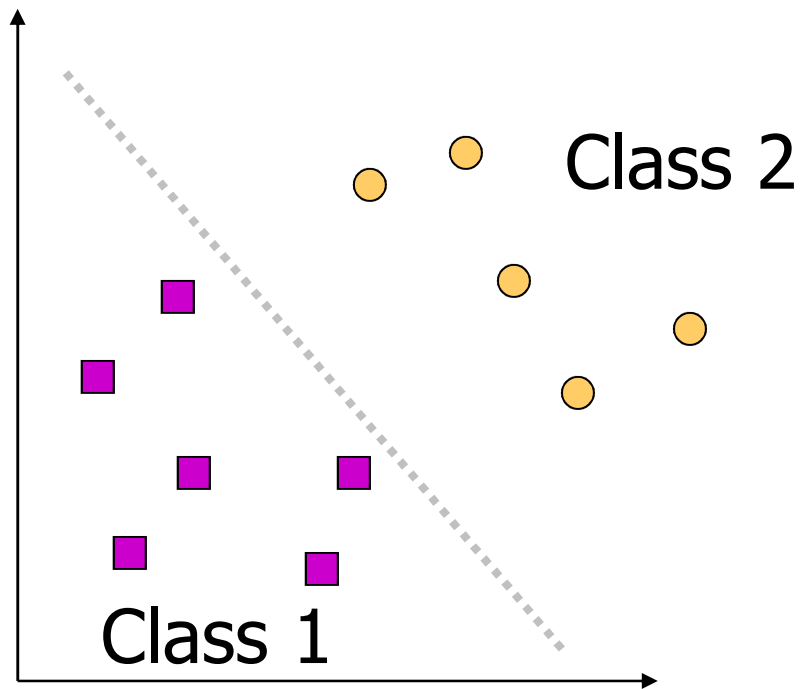
- 1) The convergence rate is quite high
- 2) Pocket algorithm is better than the Perceptron algorithm
- 3) Both algorithms have very limited use
- 4) There may be many separating hyperplanes ...

Which separating hyperplane is best?



We are interested in accuracy on the test set!

# Examples of Bad Decision Boundaries



**Idea:**

define a “continuous error measure” and try to minimize it !

## Cover's Theorem (1965):

What is the chance that a **randomly labeled set of  $N$  points** (in general position) **in  $d$ -dimensional space, is linearly separable?**

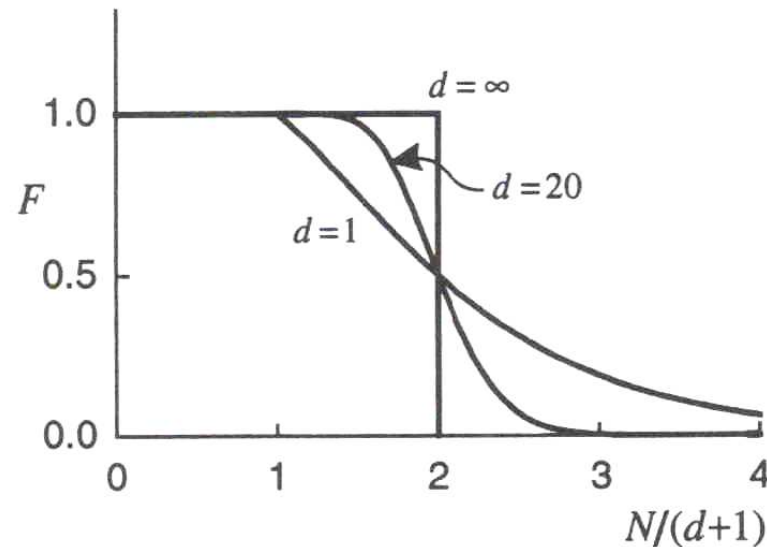


Figure 3.7. Plot of the fraction  $F(N, d)$  of the dichotomies of  $N$  data points in  $d$  dimensions which are linearly separable, as a function of  $N/(d+1)$ , for various values of  $d$ .

$$F(N, d) = \begin{cases} 1 & \text{when } N \leq d+1 \\ \frac{1}{2^{N-1}} \sum_{i=0}^d \binom{N-1}{i} & \text{when } N \geq d+1 \end{cases} \quad (3.30)$$

## Cover's Theorem in highly dimensional spaces:

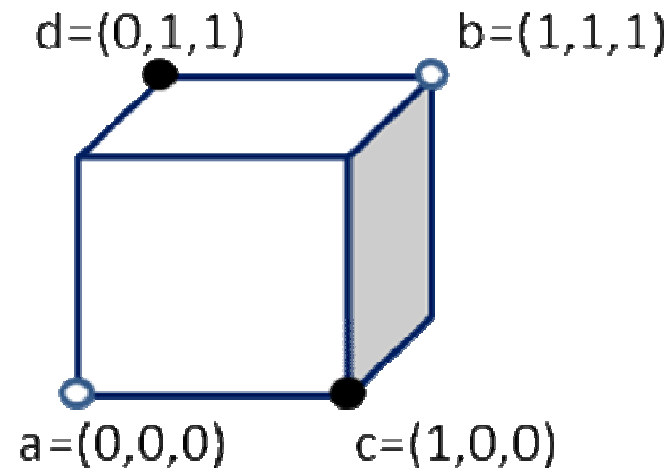
1) if the number of points in  $d$ -dimensional space is smaller than  $2^d$  then they are almost always linearly separable

2) if the number of points in  $d$ -dimensional space is bigger than  $2^d$  then they are almost always linearly non-separable

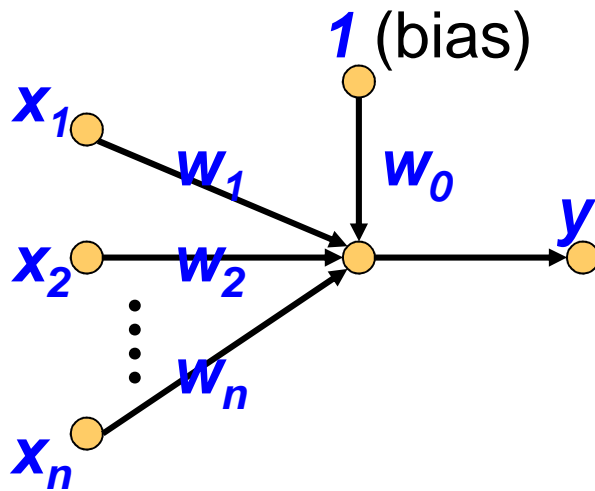
A quick check:

*Are points  $\{a, b\}$  linearly separable from  $\{c, d\}$ ?*

*How does it relate to the Cover's theorem?*



# Adaline: Adaptive Linear Element



$$y = w_0 + \sum w_i x_i$$

- Activation function  $g(x)=x$  (identity)
- The desired outputs are -1's or 1's, the actual outputs (y's) are "real numbers"
- Main idea: minimize the squared error:

$$Error = \sum_{Examples} (y_i - d_i)^2$$

# Error function

on pattern  $i$ :  $E(i)=(d_i-y_i)^2$

on all patterns:  $E= \sum (d_i-y_i)^2$

But

$$y_i = w_0 + w_1 x_1 + \dots + w_k x_k,$$

so

$$E = \sum (d_i - (w_0 + w_1 x_1 + \dots + w_k x_k))^2$$

thus

$E$  is a function of  $w_0, w_1, \dots, w_k$ .

How can we find the minimum of  $E(w_0, w_1, \dots, w_k)$  ?

***By the gradient descent algorithm ...***



# Gradient Descent Algorithm

How to find a minimum of a function  $f(x,y)$  ?

1. Start with an arbitrary point  $(x_0, y_0)$
2. Find a direction in which  $f$  is decreasing most rapidly

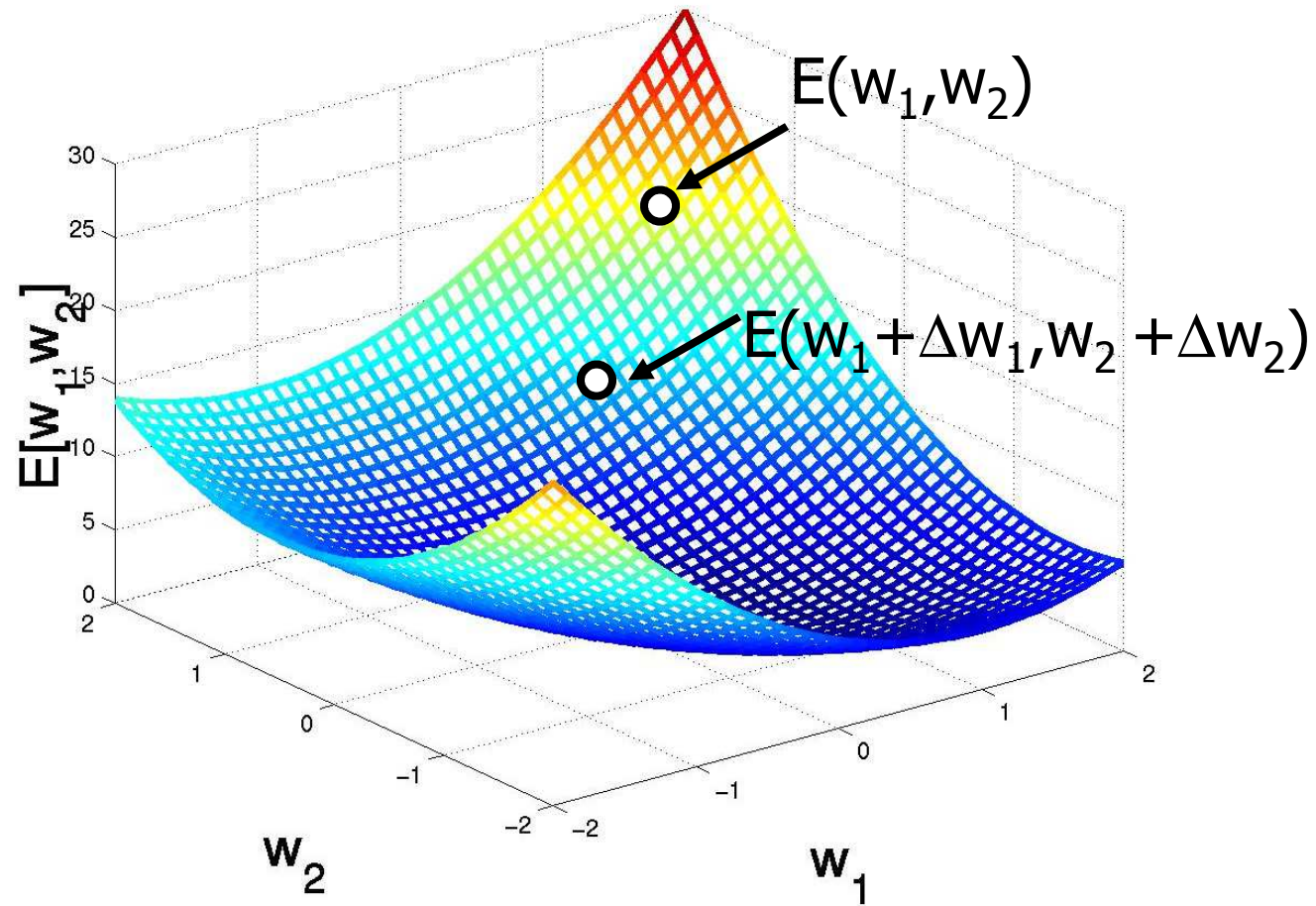
$$-\left[ \frac{\partial f(x_0, y_0)}{\partial x}, \frac{\partial f(x_0, y_0)}{\partial y} \right]$$

3. Make a small step in this direction

$$(x_0, y_0) = (x_0, y_0) - \eta \left[ \frac{\partial f(x_0, y_0)}{\partial x}, \frac{\partial f(x_0, y_0)}{\partial y} \right]$$

4. Repeat the whole process

# Gradient Descent



# Adaline: Gradient Descent

- Find  $w_i$ 's that minimize the squared error

$$E(w_0, \dots, w_m) = \sum (d-y)^2$$

- Gradient:

$$\nabla E[w] = [\partial E / \partial w_0, \dots, \partial E / \partial w_m] ; \Delta w = -\eta \nabla E[w]$$

$$\partial E / \partial w_i = \partial / \partial w_i \sum (d-y)^2 = \partial / \partial w_i \sum (d - \sum_j w_j x_j)^2 = 2 \sum (d-y)(-x_i)$$

(summation over all examples)

- The weights should be updated by:  $w_i = w_i + \eta \sum (d-y)x_i$
- Summation over all cases? Split the summation by examples!  
I.e., for each training example,  $w_i = w_i + \eta (d-y)x_i$

# Incremental versus Batch Learning

- Batch mode : gradient descent  
 $w = w - \eta \nabla E_D[w]$  over the entire data D  
 $E_D[w] = \sum_d (t_d - o_d)^2$
- Incremental mode: gradient descent  
 $w = w - \eta \nabla E_d[w]$  over individual training examples d:  $E_d[w] = (t_d - o_d)^2$
- Incremental Gradient Descent can approximate Batch Gradient Descent arbitrarily closely if  $\eta$  is small enough

# Adaline Training Algorithm

1. start with a random set of weights  $w$
2. select a pattern  $x$  (e.g., at random)
3. update weights:

$$w := w + \eta x(d - y), \text{ (Adaline rule)}$$

where  $d$  - desired output on input  $x$  and  $y = wx$

4. goto 2

**The step size  $\eta$  should be relatively small**

# Adaline and Linear Regression

- Perceptron can be used for classification problems only
- Adaline doesn't require that outputs are -1's or 1's - arbitrary values are allowed
- Therefore Adaline can be used for solving Linear Regression Problems
- There is a direct (fast) algorithm for Linear Regression!
- **But:** Adaline requires no memory: it "learns on-the-fly"; it's biologically justified (?)

# Linear Regression

---

- Assume

$$y = W_0 + W_1 X_1 + W_2 X_2 + \dots + W_n X_n$$

- Find values for  $W_0, \dots, W_n$  for which the squared error:

$$error = (d_1 - y_1)^2 + (d_2 - y_2)^2 + \dots + (d_k - y_k)^2$$

is minimized

$d_1, \dots, d_k$  are desired values,

$y_1, \dots, y_k$  are predictions

# Finding Linear Regression Model: Least Squares Method

---

- Error function is a quadratic function of  $n+1$  variables:

$$y = w_0 + w_1x_1 + w_2x_2 + \dots + w_nx_n$$

$$\text{error} = (d_1 - y_1)^2 + (d_2 - y_2)^2 + \dots + (d_k - y_k)^2$$

$$\text{error}(\mathbf{w}) = (d_1 - y_1(\mathbf{w}))^2 + (d_2 - y_2(\mathbf{w}))^2 + \dots + (d_k - y_k(\mathbf{w}))^2$$

- all partial derivatives = 0  $\implies$  error is minimal
- partial derivatives are linear functions of  **$\mathbf{W}$**



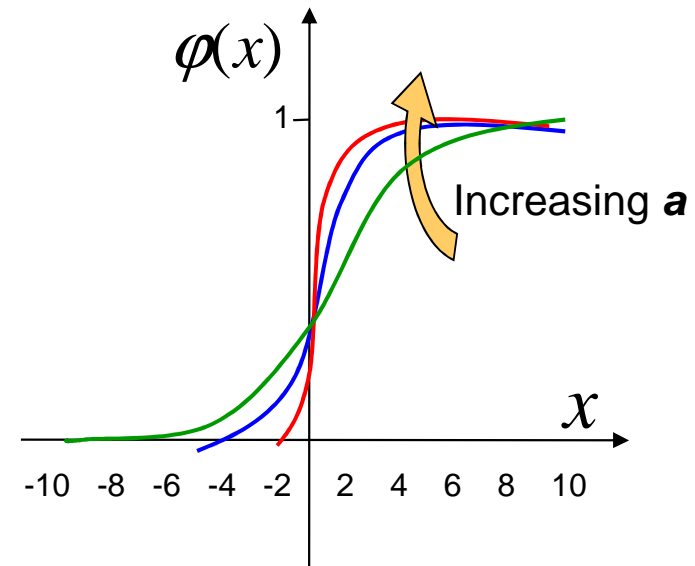
finding optimal  **$\mathbf{W}$**  reduces to the problem of solving  
a system of  **$(n+1)$**  linear equations with  **$(n+1)$**  variables  
(no matter how big the training set)



# “Smooth Perceptron” => Logistic Regression

- Main Idea:  
Replace the sign function by its “smooth approximation” and use the steepest descent algorithm to find weights that minimize the error (as with ADALINE)

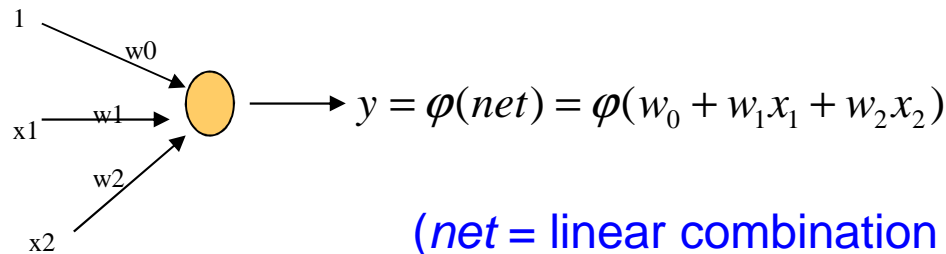
$$\varphi(x) = \frac{1}{1+e^{-ax}} \text{ with } a > 0$$



- The function to be optimized is a bit more complicated than in ADALINE case
- FORTUNATELY: the update rules are simple !

# Derivation of update rules for simple net

- Derive the Delta rule for the following network



(*net* = linear combination of inputs)

$$E(w_0, w_1, w_2) = \frac{1}{2}(y - d)^2 = \frac{1}{2}(\varphi(w_0 + w_1x_1 + w_2x_2) - d)^2$$

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} \text{ for } i \text{ in } \{0,1,2\}$$

- We need to find  $\frac{\partial E}{\partial w_0}$ ,  $\frac{\partial E}{\partial w_1}$ ,  $\frac{\partial E}{\partial w_2}$

## Derivation of Delta Rule

$$\begin{aligned}\frac{\partial E(w_0, w_1, w_2)}{\partial w_1} &= \frac{1}{2} \frac{\partial (\varphi(w_0 + w_1 x_1 + w_2 x_2) - d)^2}{\partial w_1} \\&= \frac{1}{2} 2(\varphi(w_0 + w_1 x_1 + w_2 x_2) - d) \frac{\partial (\varphi(w_0 + w_1 x_1 + w_2 x_2) - d)}{\partial w_1} \\&= (\varphi(net) - d) \varphi'(net) \frac{\partial (w_0 + w_1 x_1 + w_2 x_2)}{\partial w_1} \\&= (\varphi(net) - d) \varphi'(net) x_1\end{aligned}$$

- From similar calculations we get:

$$\frac{\partial E(w_0, w_1, w_2)}{\partial w_2} = (\varphi(net) - d) \varphi'(net) x_2$$

- and  $\frac{\partial E(w_0, w_1, w_2)}{\partial w_0} = (\varphi(net) - d) \varphi'(net)$

## Concluding:

---

$$\Delta w_0 = \eta (d - \varphi(net)) \varphi'(net)$$

$$\Delta w_1 = \eta (d - \varphi(net)) \varphi'(net) x_1$$

$$\Delta w_2 = \eta (d - \varphi(net)) \varphi'(net) x_2$$

$$\Delta \mathbf{w} = \eta (d - \varphi(net)) \varphi'(net) \mathbf{x}$$

- It's good to know that for the logistic sigmoid function:  
 $\varphi(x) = 1/(1+\exp(-x))$  we have:  $\varphi'(x) = \varphi(x)(1-\varphi(x)) = output(1-output)$
- *Adaline: Linear Regression*
- *“A Neuron”: “Logistic Regression” (what is the error function?)*

# Logistic Regression

---

- A single neuron is equivalent to logistic regression:  $y=g(\mathbf{w}\mathbf{x}+w_0)$ , where  $g(a)=1/(1+\exp(-x))$
- In case  $P(x|C_1)$  and  $P(x|C_2)$  can be modeled by Gaussians with the same matrix  $\Sigma$ , the logistic regression models the posterior probability:

$$P(C_1 | x) = \frac{p(x | C_1)P(C_1)}{p(x | C_1)P(C_1) + p(x | C_2)P(C_2)} =$$
$$= \frac{1}{1 + \exp(-a)} = g(a), \text{ where } a = \ln \frac{p(x | C_1)P(C_1)}{p(x | C_2)P(C_2)}$$

# Summary of learning rules:

Perceptron learning rule:

$$\Delta \mathbf{w} = \eta * \mathbf{x} * (\mathbf{d} - \text{out}) \text{ (for misclassified)}$$

$\mathbf{x}$  = input vector  
 $\text{out}$  = output of the network;  
 $\text{out} = f(\text{net})$ , where:  
 $\text{net}$  = linear comb. of inputs

Adaline learning rule:

$$\Delta \mathbf{w} = \eta * \mathbf{x} * (\mathbf{d} - \text{out})$$

Perceptron:  $f(\text{net}) = \text{sign}(\text{net})$

Adaline :  $f(\text{net}) = \text{net}$

“A Neuron” learning rule:

“Neuron”:  $f(\text{net}) = 1/(1 + \exp(-\text{net}))$

$$\Delta \mathbf{w} = \eta * \mathbf{x} * (\mathbf{d} - \text{out}) * \text{out}'(\mathbf{x})$$

# Perceptron for multi-class problems

---

- So far, we were considering binary classification problems: how to separate two sets of points with a (linear) model? So we were looking for a single (linear) discriminant function...
- Multi-class classification problems: we want to separate  $c > 2$  sets of points

## **Linear Separability for multi-class problems:**

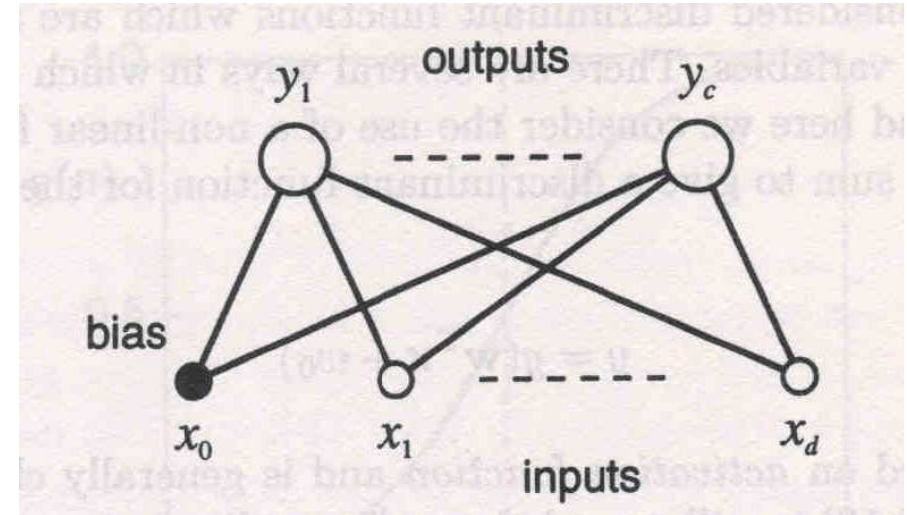
There exist  **$c$  linear discriminant functions**  $y_1(x), \dots, y_c(x)$  such that each  $\mathbf{x}$  is assigned to class  $C_k$  if and only if  $y_k(x) > y_j(x)$  for all  $j \neq k$

All algorithms discussed so far can be generalized to handle multi-class classification problems. In some cases the generalization is easy, in others not.

## **Generalized Perceptron convergence theorem:**

*If the  $c$  sets of points are linearly separable then the generalized perceptron algorithm terminates after a finite number of iterations, separating all classes.*

# Generalized Perceptron Algorithm (Duda et al.)



initialize weights  $\mathbf{w}$  at random

**while** (there are misclassified training examples)

    Select a misclassified example  $(\mathbf{x}, \mathbf{c}_i)$

    Then *some nodes* are activated more than *the node  $\mathbf{c}_i$*

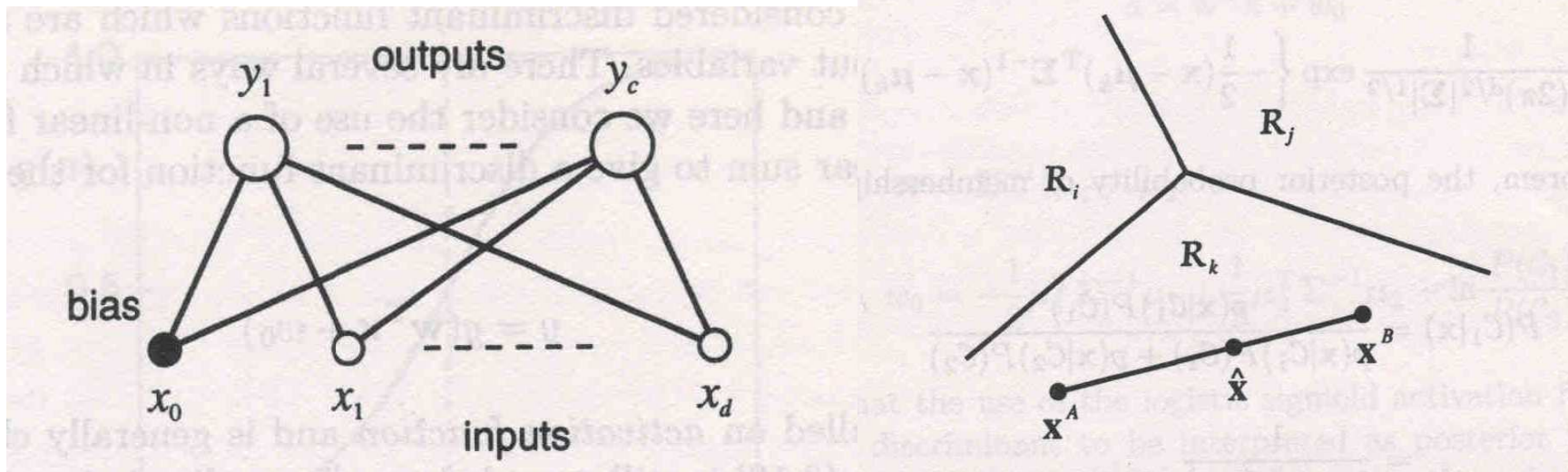
- 1) update weights of *these nodes* by  $-\mathbf{x}$ :  $\mathbf{w} = \mathbf{w} - \mathbf{x}$ ;
- 2) update weights of the node  $\mathbf{c}_i$  by  $\mathbf{x}$ :  $\mathbf{w} = \mathbf{w} + \mathbf{x}$ ;
- 3) leave weights of all other nodes unchanged

**end-while;**



# Perceptron for multi-class problems

A network of  $c$  perceptrons that share the same input vector represent  $c$  linear discriminant functions and can be used for solving multi-class classification problems.



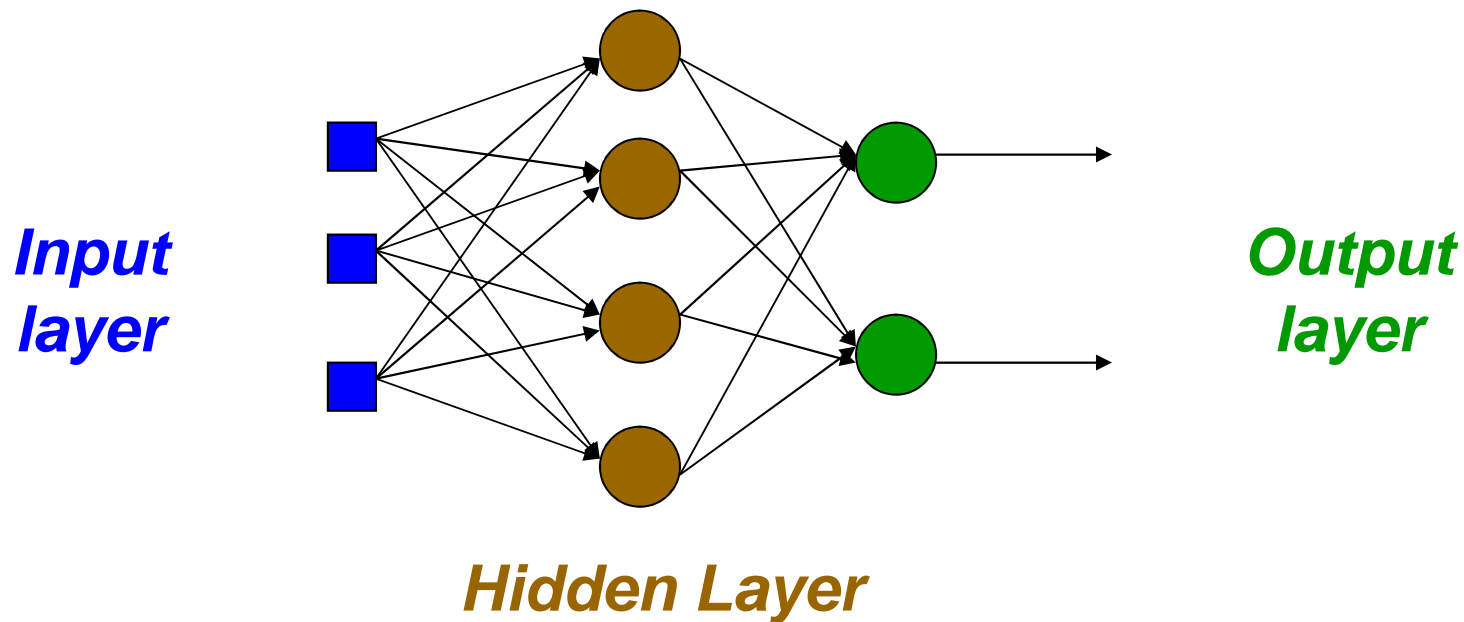
Note that  $y_i - y_j$  is a linear function which separates class  $i$  from  $j$ , for all  $i, j$ .  
So decision regions are intersections of half-spaces!

Decision regions are always **convex**: for any two points  $x^A$  and  $x^B$  from the same region, the whole line interval between  $x^A$  and  $x^B$  is also in this region.

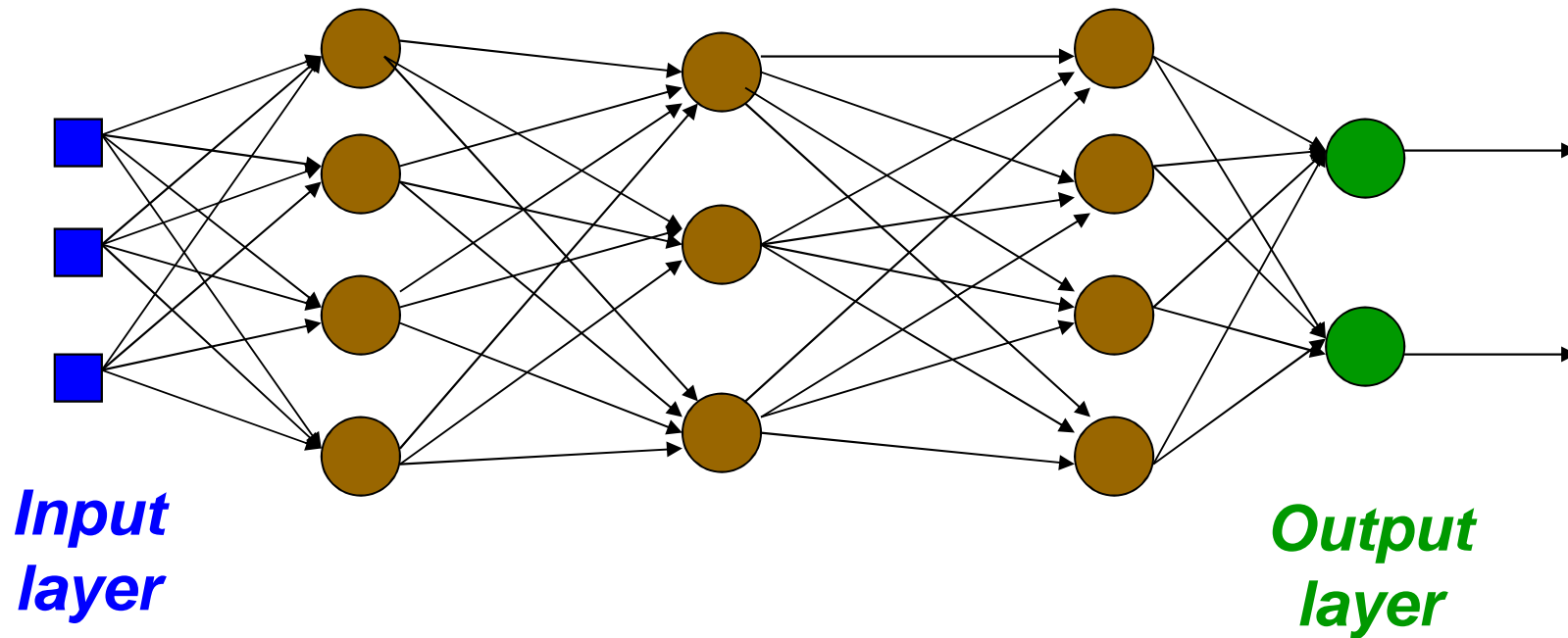
# Multi-layer Perceptron (MLP) and Backpropagation

---

- Single perceptrons = linear decision boundary
- The XOR problem cannot be solved by a single perceptron (*proof?*)
- MLPs overcome the limitation of single-layer perceptrons
- How to train them ?



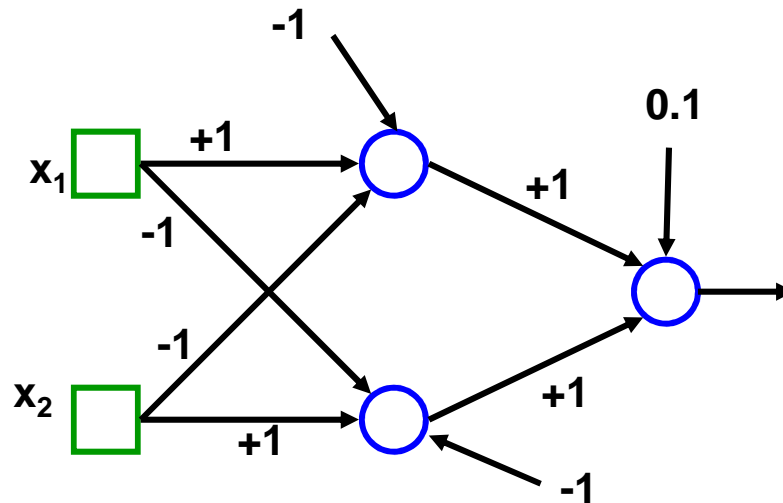
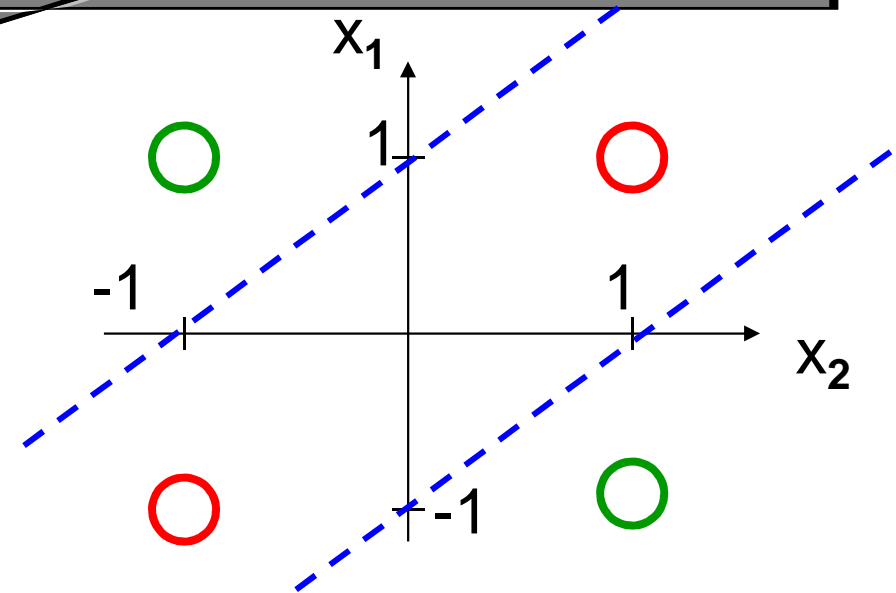
# A 3:4:3:4:2 network



*Hidden Layer 1   Hidden Layer 2   Hidden Layer 3*

# A solution for the XOR problem

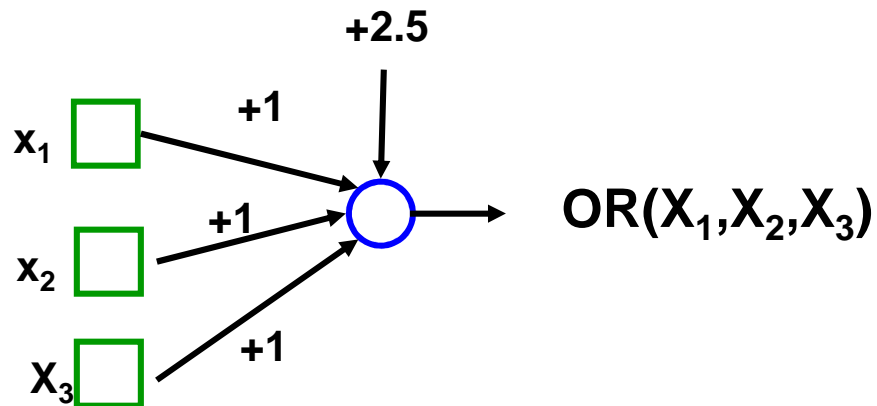
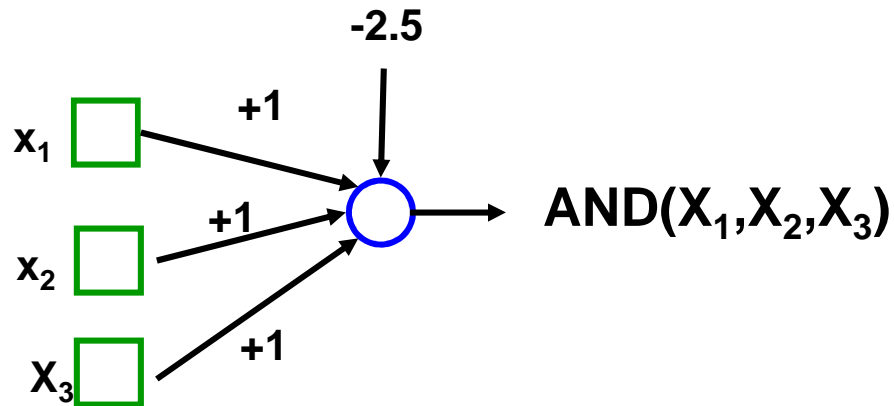
$x_1$	$x_2$	$x_1 \text{ xor } x_2$
-1	-1	-1
-1	1	1
1	-1	1
1	1	-1



$$\phi(v) = \begin{cases} 1 & \text{if } v > 0 \\ -1 & \text{if } v \leq 0 \end{cases}$$

$\phi$  is the sign function.

# Generalized AND and OR

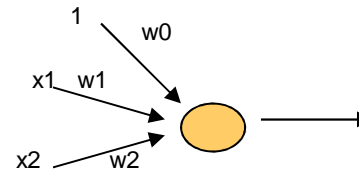


**Generalized AND (OR) can be computed by a single perceptron!**

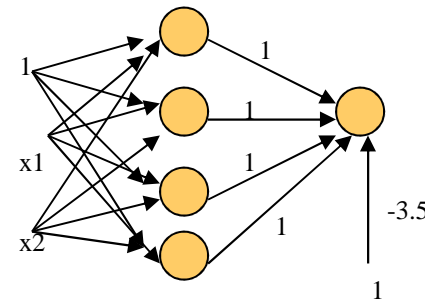
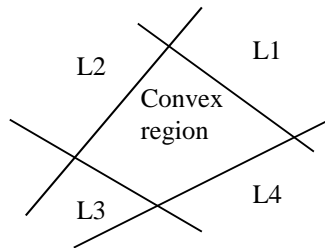
# Types of decision regions

$$w_0 + w_1x_1 + w_2x_2 > 0$$

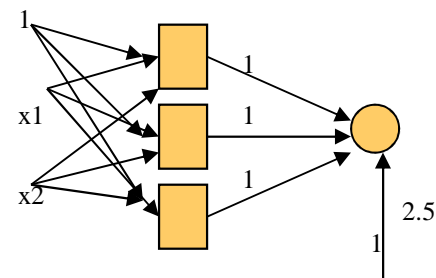
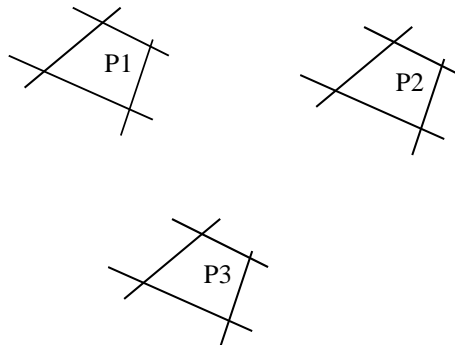
$$w_0 + w_1x_1 + w_2x_2 < 0$$



Network with a single node

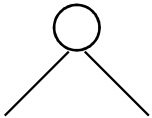
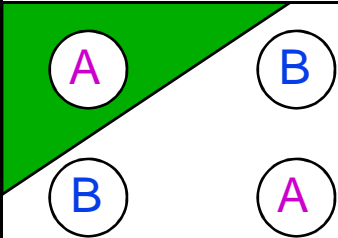
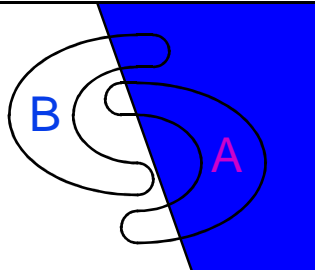
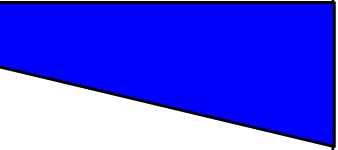
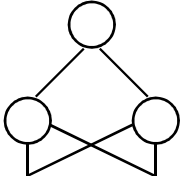
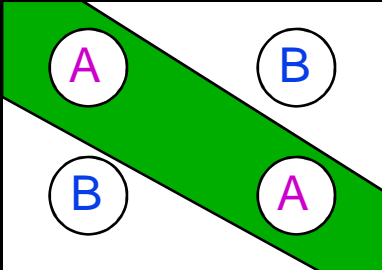
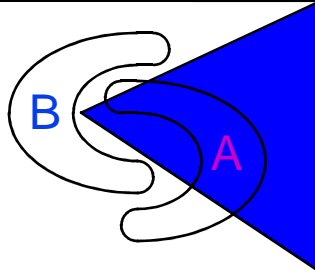
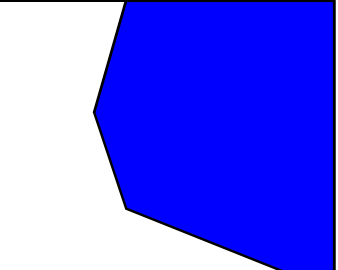
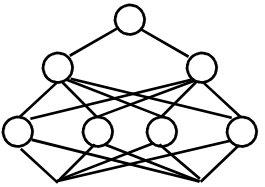
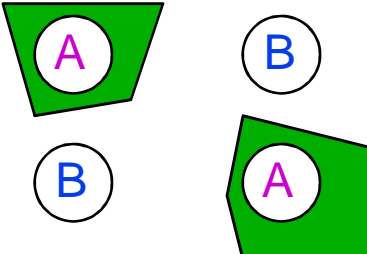
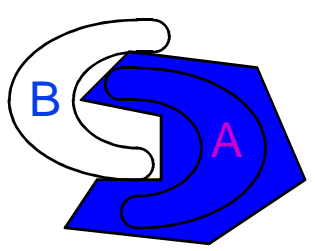
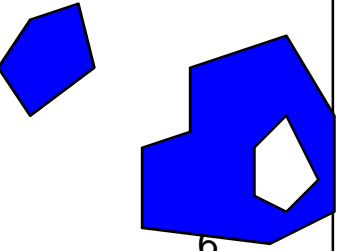


One-hidden layer network that realizes the convex region: each hidden node realizes one of the lines bounding the convex region



two-hidden layer network that realizes the union of three convex regions: each box represents a one hidden layer network realizing one convex region

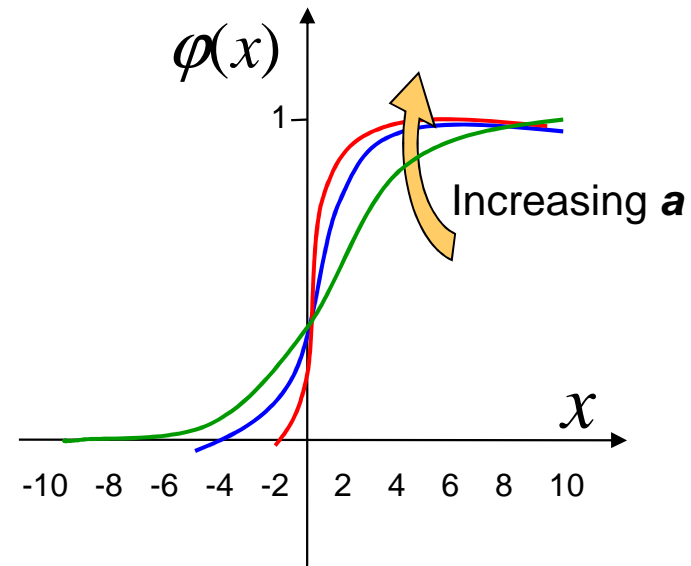
# Different Non-Linearly Separable Problems

Structure	Types of Decision Regions	Exclusive-OR Problem	Classes with Meshed regions	Most General Region Shapes
Single-Layer 	Half Plane Bounded By Hyperplane			
Two-Layer 	Convex Open Or Closed Regions			
Three-Layer 	Arbitrary (Complexity Limited by No. of Nodes)			

# How to train multi-layer networks?

- Main Idea:  
Replace the sign function by its “smooth approximation” and use the gradient descent algorithm to find weights that minimize the error (as with ADALINE)

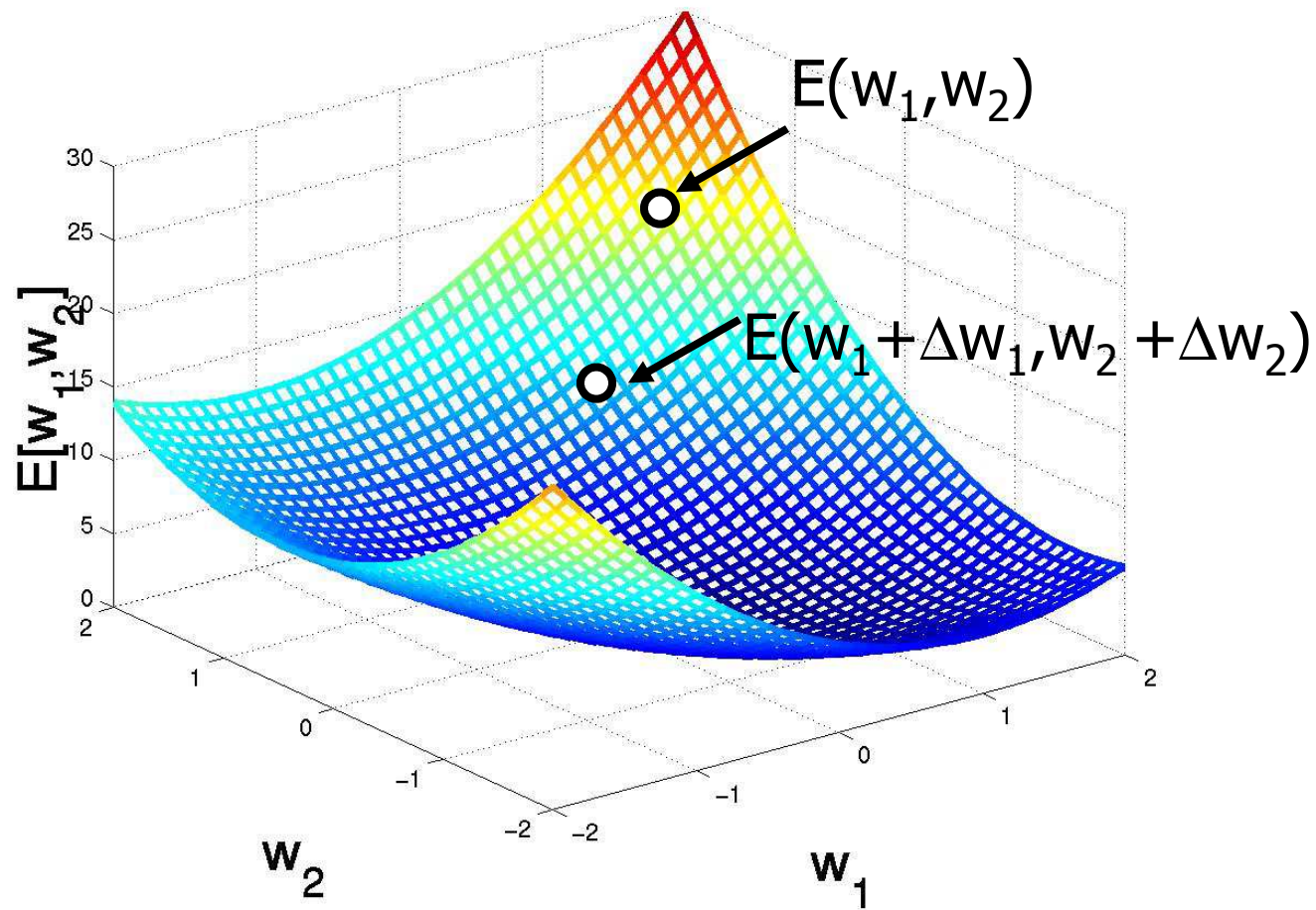
$$\varphi(x) = \frac{1}{1+e^{-ax}} \text{ with } a > 0$$



- The function to be optimized is a complicated one, with many unknowns and nestings ...
- FORTUNATELY: the final update rules are simple !!!



# Gradient Descent



# Weight Update Rule

---

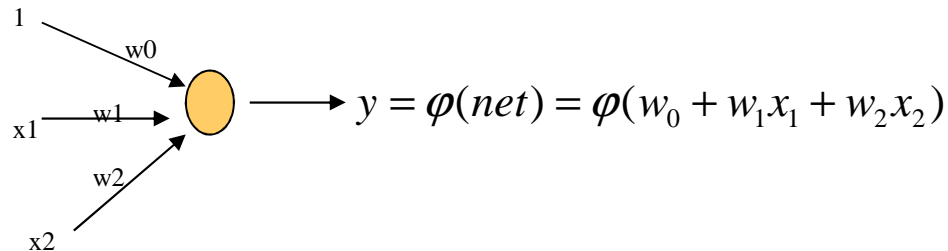
**gradient descent method:** “walk” in the direction yielding the maximum decrease of the network error  $E$ . This direction is the opposite of the gradient of  $E$ .

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

$$\Delta w_{ji} = -\eta \frac{\partial E}{\partial w_{ji}}$$

# Delta Rule: one node

- Derive the Delta rule for the following network



$$E(w_0, w_1, w_2) = \frac{1}{2}(y - d)^2 = \frac{1}{2}(\varphi(w_0 + w_1x_1 + w_2x_2) - d)^2$$

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} \text{ for } i \text{ in } \{0,1,2\}$$

- We need to find  $\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}$

## Delta Rule: one node

$$\begin{aligned}\frac{\partial E(w_0, w_1, w_2)}{\partial w_1} &= \frac{1}{2} \frac{\partial (\varphi(w_0 + w_1 x_1 + w_2 x_2) - d)^2}{\partial w_1} \\&= \frac{1}{2} 2(\varphi(w_0 + w_1 x_1 + w_2 x_2) - d) \frac{\partial (\varphi(w_0 + w_1 x_1 + w_2 x_2) - d)}{\partial w_1} \\&= (\varphi(net) - d) \varphi'(net) \frac{\partial (w_0 + w_1 x_1 + w_2 x_2)}{\partial w_1} \\&= (\varphi(net) - d) \varphi'(net) x_1\end{aligned}$$

- From similar calculations we get:

$$\frac{\partial E(w_0, w_1, w_2)}{\partial w_2} = (\varphi(net) - d) \varphi'(net) x_2$$

- and  $\frac{\partial E(w_0, w_1, w_2)}{\partial w_0} = (\varphi(net) - d) \varphi'(net)$

# Delta Rule: one node

---

Thus the update rules for the weights are:

$$\Delta w_0 = \eta (d - \varphi(net)) \varphi'(net) \cdot 1$$

$$\Delta w_1 = \eta (d - \varphi(net)) \varphi'(net) x_1$$

$$\Delta w_2 = \underbrace{\eta (d - \varphi(net))}_{\text{delta}} \underbrace{\varphi'(net) x_2}_{\text{input}}$$

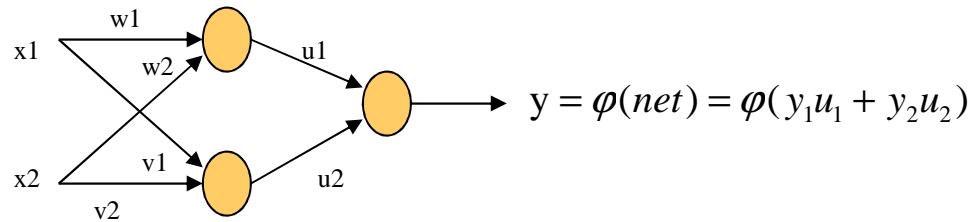
It's handy to know that for the logistic sigmoid function:

$\varphi(x) = 1/(1+\exp(-x))$  we have:  $\varphi'(x) = \varphi(x)(1-\varphi(x)) = \text{output}(1-\text{output})$ ,

so once we know  $\varphi(x)$  we also know  $\varphi'(x)$  !

# Delta Rule: XOR-network

Consider the network (no biases):



$$\text{net}_1 = w_1 x_1 + w_2 x_2, \quad y_1 = \varphi(\text{net}_1)$$

$$\text{net}_2 = v_1 x_1 + v_2 x_2, \quad y_2 = \varphi(\text{net}_2)$$

$$\text{net} = y_1 u_1 + y_2 u_2, \quad y = \varphi(\text{net})$$

$$= \varphi(y_1 u_1 + y_2 u_2) =$$

$$= \varphi(\varphi(\text{net}_1) u_1 + \varphi(\text{net}_2) u_2) =$$

$$= \varphi(\varphi(w_1 x_1 + w_2 x_2) u_1 + \varphi(v_1 x_1 + v_2 x_2) u_2)$$

# Derivation of the Delta Rule

---

$$\begin{aligned}\frac{\partial E}{\partial u_1} &= \frac{1}{2} \frac{\partial (\varphi(y_1 u_1 + y_2 u_2) - d)^2}{\partial u_1} = \frac{1}{2} 2(\varphi(y_1 u_1 + y_2 u_2) - d) \frac{\partial (\varphi(y_1 u_1 + y_2 u_2) - d)}{\partial u_1} \\ &= (y - d) \varphi'(net) \frac{\partial (y_1 u_1 + y_2 u_2)}{\partial u_1} = (y - d) \varphi'(net) y_1\end{aligned}$$

From similar calculations we get:  $\frac{\partial E}{\partial u_2} = (y - d) \varphi'(net) y_2$

$$\begin{aligned}\frac{\partial E}{\partial w_1} &= \frac{1}{2} \frac{\partial (\varphi(y_1 u_1 + y_2 u_2) - d)^2}{\partial w_1} = \frac{1}{2} 2(\varphi(y_1 u_1 + y_2 u_2) - d) \frac{\partial (\varphi(y_1 u_1 + y_2 u_2) - d)}{\partial w_1} \\ &= (y - d) \varphi'(net) \frac{\partial (y_1 u_1 + y_2 u_2)}{\partial w_1}\end{aligned}$$

# Derivation of the Delta Rule

---

$$\begin{aligned}\frac{\partial(y_1 u_1 + y_2 u_2)}{\partial w_1} &= \frac{\partial(y_1 u_1)}{\partial w_1} = u_1 \frac{\partial y_1}{\partial w_1} = u_1 \frac{\partial(\varphi(w_1 x_1 + w_2 x_2))}{\partial w_1} \\ &= u_1 \varphi'(net_1) \frac{\partial(w_1 x_1)}{\partial w_1} = u_1 \varphi'(net_1) x_1\end{aligned}$$

So we obtain:

$$\begin{aligned}\frac{\partial E}{\partial w_1} &= (y - d) \varphi'(net) \frac{\partial(\varphi(y_1 u_1 + y_2 u_2))}{\partial w_1} \\ &= (y - d) \varphi'(net) u_1 \varphi'(net_1) x_1\end{aligned}$$

From similar calculations we get:

$$\begin{aligned}\frac{\partial E}{\partial w_2} &= (y - d) \varphi'(net) u_1 \varphi'(net_1) x_2 \\ \frac{\partial E}{\partial v_1} &= (y - d) \varphi'(net) u_2 \varphi'(net_2) x_1, \quad \frac{\partial E}{\partial v_2} = (y - d) \varphi'(net) u_2 \varphi'(net_2) x_2\end{aligned}$$

*delta*



# General case: Generalized Delta Rule

---

$\Delta w_{ji} = \eta \delta_j y_i$  , where:

$$\delta_j = \begin{cases} \phi'(v_j)(d_j - y_j) & \text{IF } j \text{ output node} \\ \phi'(v_j) \sum_{\text{k of nextlayer}} \delta_k w_{kj} & \text{IF } j \text{ hidden node} \end{cases}$$

$w_{ji}$  : weight from node  $j$  to node  $i$  (moving from output to input!)

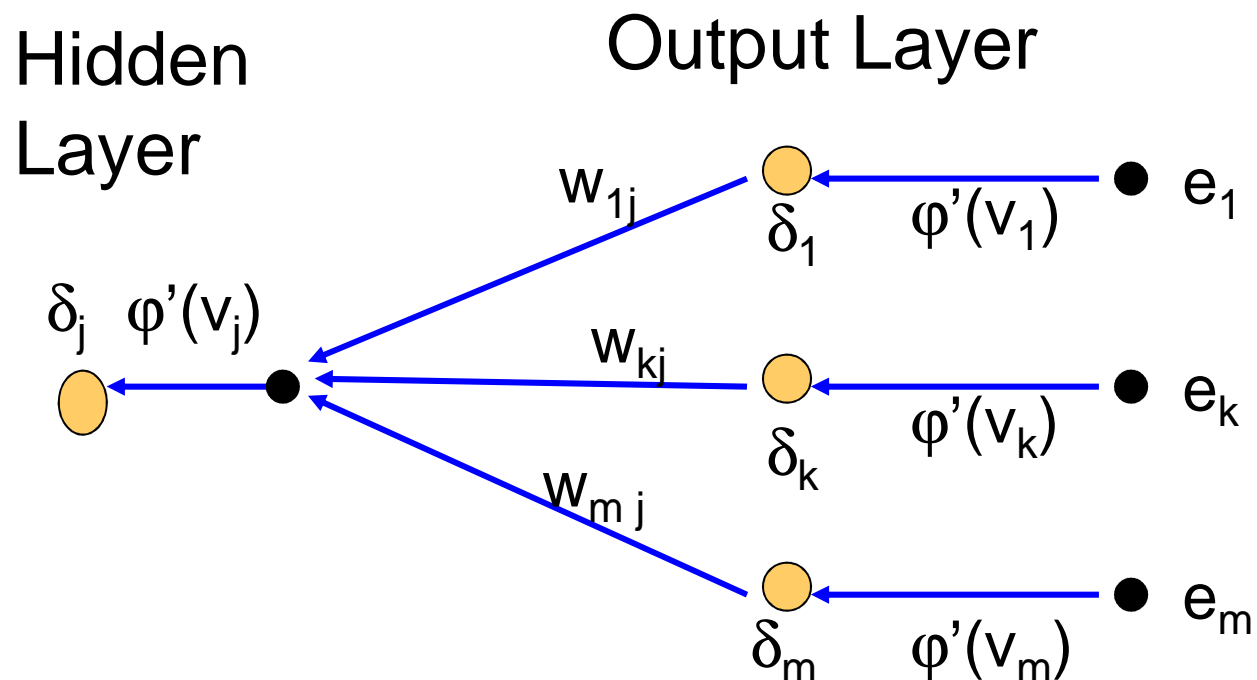
$\eta$  : learning rate (constant)

$y_j$  : output of node  $j$

$v_j$  : activation of node  $j$

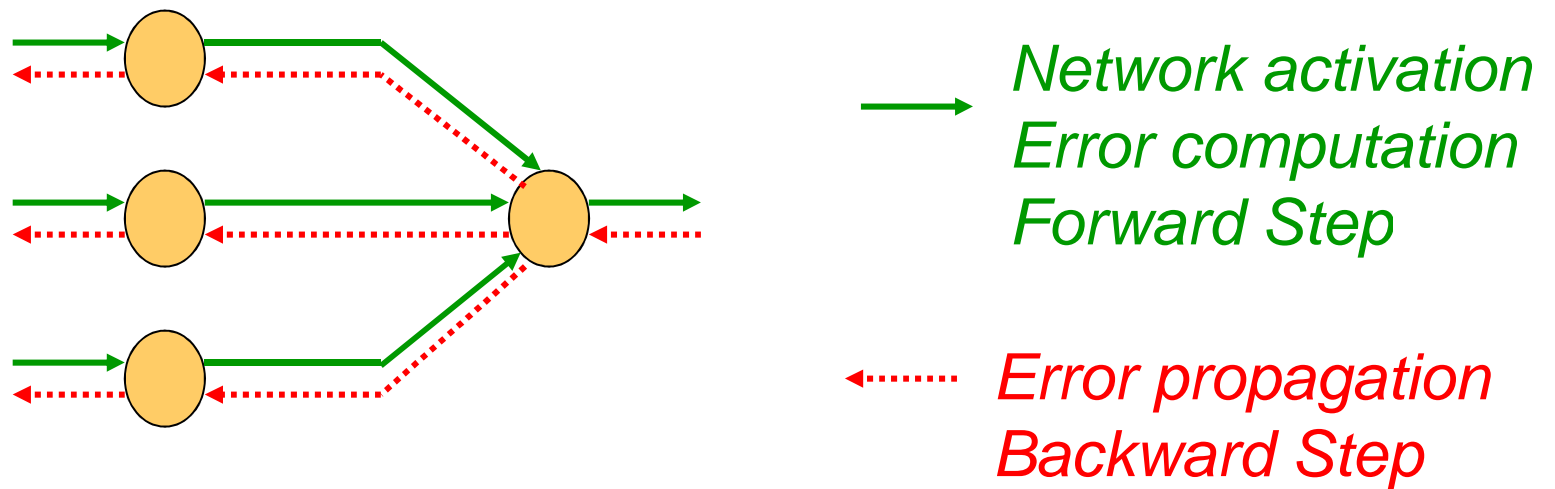
# Error backpropagation

The flow-graph below illustrates how errors are back-propagated from  $m$  output nodes to the hidden neuron  $j$



# Backpropagation: two phases

---



# Backpropagation algorithm

---

- The Backprop algorithm searches for weight values that **minimize the total error of the network**
- Backprop consists of the **repeated** application of the following two passes:
  - **Forward pass**: in this step the network is activated on one example and the **error** of each neuron of the output layer is **computed**.
  - **Backward pass**: in this step the network error is used for updating the weights. Starting at the output layer, **the error is propagated backwards through the network, layer by layer** with help of the generalized delta rule. Finally, all weights are updated.
- No guarantees of convergence  
(when learning rate too big or too small)
- In case of convergence: local (or global) minimum
- In practice: try several starting configurations and learning rates.

# Network training:

## Two types of network training:

- **Incremental mode** (on-line, stochastic, or per-pattern)  
Weights updated after each pattern is presented
- **Batch mode** (off-line or per -epoch)  
Weights updated after all the patterns are presented

# Advantages and disadvantages of different modes

## Sequential mode:

- Less storage for each weighted connection
- Random order of presentation and updating per pattern means search of weight space is stochastic--reducing risk of local minima able to take advantage of any redundancy in training set (i.e.. same pattern occurs more than once in training set, esp. for large training sets)
- Simpler to implement

## Batch mode:

- Faster learning than sequential mode

# Stopping criteria

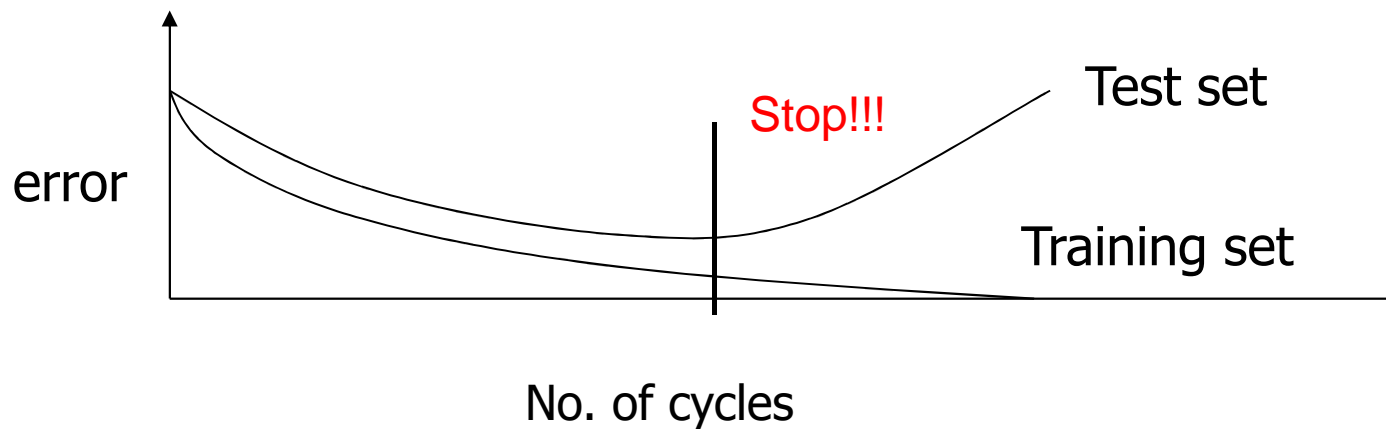
---

- **total mean squared error change**: Backprop is considered to have converged when the absolute rate of change in the average squared error per epoch is sufficiently small
- **generalization based criterion**: After each epoch the NN is tested for generalization using a different set of examples (test set). If the generalization performance is adequate then stop. (**Early Stopping**)

# Early Stopping

Training network for too many cycles may lead to **data overfitting** (or “overtraining”)

**Early Stopping:**  
**stop training as soon as the error on the test set increases**





# Early stopping and 3 sets:

---

- **Training set** – used for training
- **Test set** – used to decide when to stop training by monitoring the error on it
- **Validation set** – used for final estimation of the performance of the trained neural network  
(using the test set for it is methodologically not correct)

In practice, the use of validation set is not so important!

# Model Selection by Cross-validation

---

- **Too few hidden units** may prevent the network from learning adequately the data and learning the concept.
- **Too many hidden units** leads to overfitting.
- A **cross-validation scheme** can be used to determine an appropriate number of hidden units by using the optimal test error to select the model with optimal number of hidden layers and nodes.
- N-fold cross-validation:
  1. split your data into N parts (equal size);
  2. develop N networks on all combinations of (N-1) parts;
  3. test each network on the remaining parts (test sets);
  4. average the error over these test sets.

# Expressive Power of MLP

---

## Boolean functions:

- **Every boolean function** can be described by a network with a **single hidden layer**
- but it might require **exponential** (in the number of inputs) hidden **neurons**.

## Continuous functions:

- **Any bounded continuous function** can be approximated with arbitrarily small error by a network with **two hidden layers**.
- **Stronger: Any bounded continuous function** can be approximated with arbitrarily small error by a network with **one hidden layer**.

# Complexity of Learning

---

Blum, Rivest (1993):

“Training a 3-node neural network is NP-complete”

- “training” = “optimal training”
- 3-nodes, but N k-dimensional input patterns
- “NP-complete” =  
“no polynomial-time algorithm exists”
- “in practice, the problem is not solvable”

# Optimization Methods (Ch. 7)

---

- Variable Learning Rate algorithms
  - Generalized Delta Rule (with momentum)
  - Adaptive Gradient Descent
- Numerical Optimization Algorithms
  - Quickpropagation Algorithm
  - Line Search Method (Brent's Algorithm)
  - Conjugate Gradient Algorithms
- Applications

# Generalized delta rule

---

- In classical backpropagation weights are updated in cycles which may lead to some “chaotic” directions of updates (from cycle to cycle or from pattern to pattern).
- Therefore it make sense to keep an extra term that takes care of the “general direction of changes”.
- This is achieved by introducing **a momentum term  $\alpha$**  in the delta rule that takes into account previous updates:

$$\Delta w_{ji}^{\text{new}} = \alpha \Delta w_{ji}^{\text{previous}} + \eta \delta_j y_i$$

# Adaptive Gradient Descent

---

- "classical" backproagation is very slow !!!
  - if learning rate too small  $\Rightarrow$  convergence too slow
  - if learning rate too big  $\Rightarrow$  no convergence
- Idea: ***dynamically tune the value of the learning rate !***
- **Adaptive Gradient Descent (a bold-driver strategy):**

if new_error < old_error	$\Rightarrow$	increase learning rate $lr = 1.05 * lr;$
if new_error > 1.04 * old.error	$\Rightarrow$	ignore weight changes; decrease learning rate $lr = 0.7 * lr;$

# Working with Neural Networks

---

## Process:

- problem
- data
- data preprocessing***
- network development***
- solution

## Network Development:

- Data representation
- Network Topology
- error function; activation function***
- Network Parameters
- Training
- Validation



# Error Functions (Ch. 6, Bishop's book)

---

- **Binary classification**: two classes A, B; is  $x$  in A or B?
- **Multi-class classification**: eg., 10 classes (digits)
- **Regression**: predict the value of  $f(x)$
- Why Sum-Squared-Error? Are there better alternatives?
- Statisticians: **under some assumptions** about the data there are some “best” error functions for each problem!
- Keep in mind that **in practice** the assumptions might not be satisfied and **another error functions** (like SSE) **might work best!**

# Binary Classification

---

If the output of the network, *output*, is interpreted as probability then for each input pattern  $\langle \mathbf{x}, t \rangle$ , where the class label  $t = 0$  or  $1$ , it should satisfy:

- $0 < \text{output} < 1$
- $p(t/x) = \text{output}^t (1 - \text{output})^{1-t}$

It turns out that ML estimates of weights of a MLP with a sigmoid output unit minimize the cross-entropy error function:

$$E = -\sum \text{target}_k \cdot \log(\text{output}_k) + ((1 - \text{target}_k) \cdot \log(1 - \text{output}_k))$$

**Main Conclusion :**

**For binary classification problems use logistic output unit and minimize the cross-entropy function!**

# Multi-class Classification

---

In case of  $c$  classes and 1-of- $c$  coding of the outputs, the error function that should be optimized is also the cross-entropy:

$$E = -\sum_k \sum_c \text{target}_{k^*} \log(\text{output}_k)$$

But the output layer is activated by a softmax activation function:

$$y_k = \exp(a_k) / \sum_{k'} (\exp(a_{k'}))$$

where  $a_k$  denotes the input to the  $k$ -th output node.

**Main Conclusion :**

**For multi-class classification problems use softmax activation function and minimize the cross-entropy function!**

# Regression

---

Assume that the training set  $(x, t)$  is given by:

$$f(x) + \text{norm}(0, \sigma)$$

( $f(x)$  is a “deterministic” function).

Then one can show that the ML estimates of weights of a MLP that model the training set correspond to the result of training MLP with the linear output unit with Sum-Squared-Error measure.

Moreover, for each  $x$ , the output of the network approximates  $\langle t | x \rangle$  (the assumption about normality of noise can be skipped).

**Main Conclusion :**

**For regression problems use linear outputs and the Sum-Squared-Error function**

# Error functions in Netlab/other packages

---

The three error functions are implemented in Netlab and should be specified when providing the type of the output nodes:

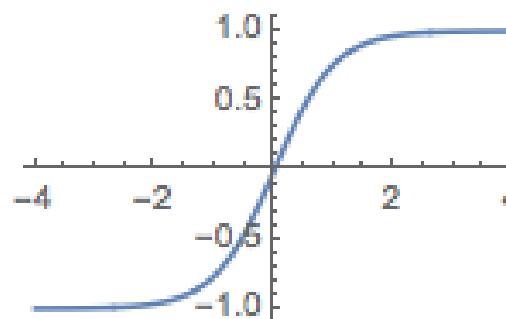
'linear' => SSE

'logistic' => cross-entropy

'softmax' => cross-entropy + softmax

One can also add his own error function definitions (and the corresponding gradients).

[Note: in Netlab all hidden nodes use the tanh activation function ]



# Data Preparation: example

---

fuel\_consumption = f(  
    **cylinders:**          multi-valued discrete  
    **displacement:**     continuous  
    **horsepower:**       continuous  
    **weight:**           continuous (?)  
    **acceleration:**     continuous  
    **model year:**       multi-valued discrete (?)  
    **model type:**       multi-valued discrete  
    **model color:**      multi-valued discrete (?)  
    **maker:**           multi-valued discrete  
)

# Input Representation



- Be creative !!!
  - use a single node to represent a single number
  - use several input nodes to represent a single number
  - scale your data
  - ignore some variables
  - introduce new variables
  - use polar system of coordinates
  - .....

# Reasons...



- Data representation **depends on the problem**.
- In general NNs work with **continuous (real valued) inputs**. Therefore symbolic attributes are encoded by numbers.
- Attributes of different types may have different ranges of values which affect the training process.
- **Normalization** may be used, like the following one which scales each attribute to assume values between 0 and 1 (or [-1 1]).

$$x = \frac{x - \min}{\max - \min}$$



# Alternatives

---

- Standardization: “center and normalize”:

$x := (x - \text{mean}(x)) / \text{std}(x)$  (new x has mean=0 and std=1!)

- Use a “thermometer representation”, e.g.:  
any x from [0 10] can be represented by 10 inputs:

$2.7 \rightarrow [1, 1, 0.7, 0, 0, 0, 0, 0, 0, 0]$

- “squeeze” outliers (set lower and upper bounds):

$x = \min(x, \text{upperbound}); x = \max(x, \text{lowerbound})$

- Replace  $x$  by  $\text{rank}(x)$  (the position in sorted version of x)
- Etc., etc.

# Discrete Values

---

- N discrete values can be represented by:
  - Vectors of length n:  
 $1 \Rightarrow [1, 0, \dots], 2 \Rightarrow [0, 1, 0, \dots]$
  - Vectors of length  $\log(n)$  (binary representations)  
 $1 \Rightarrow [1, 0, 0], 2 \Rightarrow [0, 1, 0], 3 \Rightarrow [1, 1, 0], \dots$
  - A single number, e.g, the relative frequency of observing the given value in the data
  - Colors could be represented by their 3 RGB components
  - use 2 nodes to represent days of a week:  
 $(\sin(k \cdot 2 \cdot \pi / 7), \cos(k \cdot 2 \cdot \pi / 7)), k=0, 1, \dots, 6$

# Output representation

---

- Different strategies for different problems:
  - **binary classification**
  - **multiclass classification**
  - **function approximation**
- Binary classification problems:
  - Two ways of representing outputs:
    - a single output node
    - two output nodes
- **Remark:** sigmoid function never (?) returns 0 or 1  
=> replace 0 by 0.1 and 1 by 0.9

# Interpreting the output

---

A) **if** output  $> 0.5$  **then** 1 **else** 0

B) **if** output  $> 0.7$  **then** 1  
    **elseif** output  $< 0.3$  **then** 0  
    **else** “unclassified”

C) **if** output  $>$  Threshold **then** 1 **else** 0  
(Threshold chosen in such a way that the ratio  
POS:NEG is consistent with the training set)

# Multi-class classification



Cases should be classified into  $K > 2$  classes

1. K binary classification problems (K nets)
2. K output nodes (one net)
  - 'Winner-Takes-All' rule (max node decides)
  - 'bit-by-bit' match
3.  $\log_2(K)$  output nodes (binary coding)

# Function Approximation (regression)

---

- target values rescaled to:  
[0, 1] or [0.1, 0.9] if logistic sigmoid is used  
[-1, 1] or [-0.9, 0.9] if hyperbolic tangent is used
- output unit could use a linear activation function
- What about unbounded functions  
(e.g.  $f(x)=1/x$ )?  
=> Use, e.g.,  $y \Rightarrow 1/(1+y)$  transformation

**In all cases remember to convert  
outputs to the original scale !!!**

# Network Topology



- The number of layers and of neurons **depend on the specific task**. In practice this issue is solved by trial and error.
- Two types of heuristics can be used:
  - start from a large network and successively remove some neurons and links until network performance degrades.
  - **RECOMMENDED:** begin with a small network and introduce new neurons until performance is satisfactory.

# Network parameters



- How are the **weights** initialized?
- How is the **learning rate** chosen?
- How many **hidden layers** and how many **neurons**?
- How many examples in the **training set**?



# Initialization of weights

---

- In general, initial weights are **randomly chosen**, with typical values between -1.0 and 1.0 or -0.5 and 0.5.
- **If some inputs are much larger than others**, random initialization may bias the network to give much more importance to larger inputs. In such a case, weights can be initialized as follows:

$$w_{ji} = \pm \frac{1}{2N} \sum_{i=1, \dots, N} \frac{1}{|x_i|}$$

For weights from the input to the first layer

$$w_{kj} = \pm \frac{1}{2N} \sum_{i=1, \dots, N} \frac{1}{\varphi(\sum w_{ji} x_i)}$$

For weights from the first to the second layer

# Choice of learning rate

---

- The right value of  $\eta$  depends on the application. Values between 0.1 and 0.9 have been used in many applications.
- Trial-and-error is the best heuristic, e.g.,:  
0.01, 0.03, 0.06, 0.1, 0.3, 0.6, ...  
0.01, 0.006, 0.003, 0.001, ...
- Use faster algorithms (whenever possible)!

# Size of Training set

---

- Rule of thumb:
  - the number of training examples should be at least five to ten times the number of weights of the network.
- Other rule:

$$N > \frac{|W|}{(1 - a)}$$

|W|= number of weights  
a=expected accuracy

# Applications of MLP

---

## Classification, pattern recognition:

- MLP can be applied to non-linearly separable learning tasks:
  - Recognizing printed or handwritten characters
  - Face recognition
  - Scoring loan applications
  - Analysis of sonar data recognize mines

## Regression and forecasting:

- MLP can be applied to learn non-linear functions (regression) and in particular functions whose inputs is a sequence of measurements over time (time series).

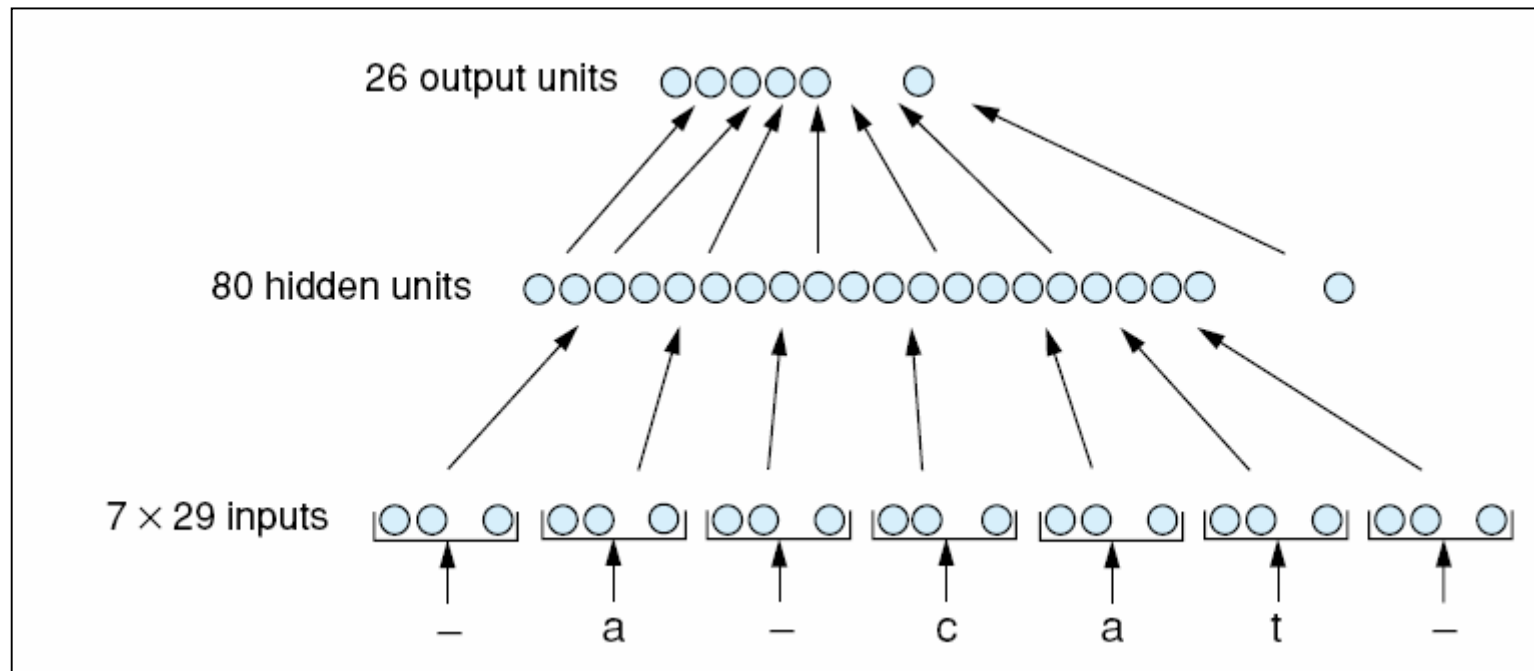
## Data Compression and Dimensionality Reduction

# NETtalk

(Sejnowski & Rosenberg, 1987)

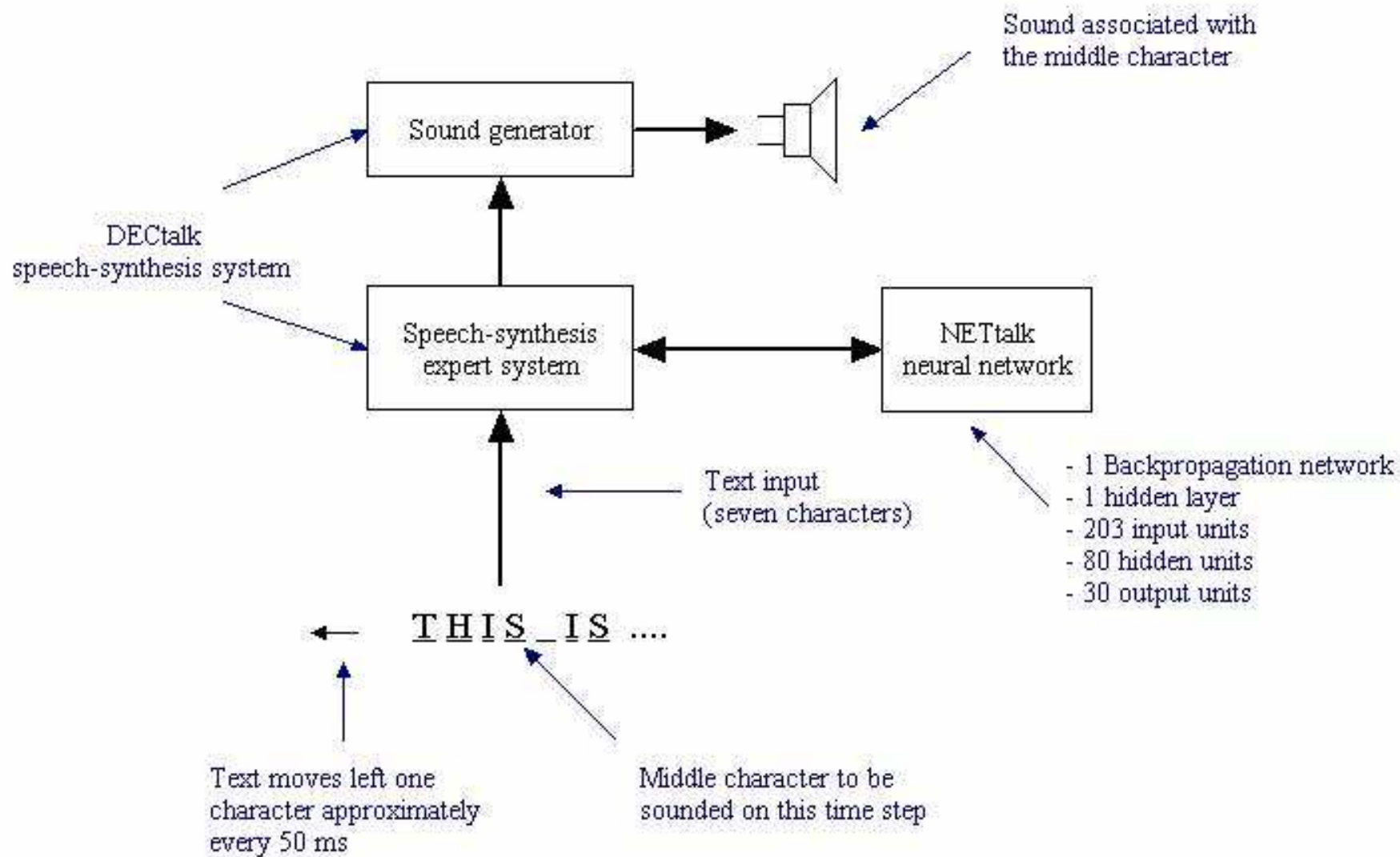
- The task is to **learn to pronounce English text** from examples (text-to-speech)
- Training data is a list of words from a side-by-side English text -phoneme source
- Input: 7 consecutive characters from written text presented in a moving window that scans text
- Output: phoneme code giving the pronunciation of the letter at the center of the input window
- Network topology: 7x29 binary inputs (26 chars + punctuation marks), 80 hidden units and 26 output units (phoneme code). Sigmoid units in hidden and output layer

# NETtalk



## NETtalk (contd.)

- Training protocol: 95% accuracy on training set after 50 epochs of training by full gradient descent.  
78% accuracy on a test set
- DEC-talk: a rule-based system crafted by experts (a decade of efforts by many linguists)
- Functionality/Accuracy almost the same
- Try: <http://cnl.salk.edu/Media/nettalk.mp3>





# ALVINN: Autonomous Land Vehicle In a Neural Network

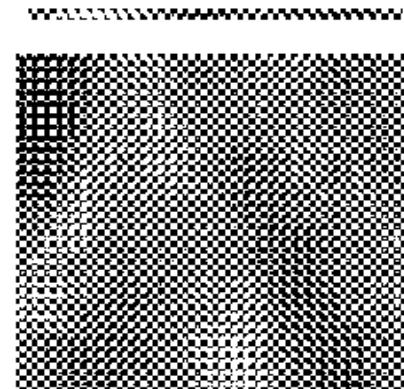
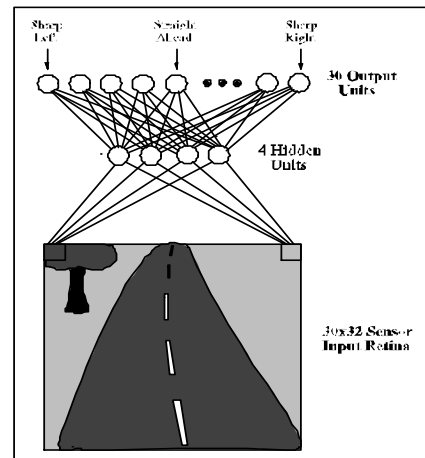
[http://www.ri.cmu.edu/projects/project\\_160.html](http://www.ri.cmu.edu/projects/project_160.html)



30 outputs  
for steering

4 hidden  
units

30x32 pixels  
as inputs



30x32 weights  
into one out of  
4 hidden units

# Fraud Detection with Credit Cards

<http://www.fico.com/en/Products/DMAApps/Pages/FICO-Falcon-Fraud-Manager.aspx>

Right now, leading institutions around the world trust Fair Isaac's Falcon™ Fraud Manager to protect more than **450 million active credit and debit cards**. Why? They know they'll receive regular technological advances which will allow them to stay a step ahead of new and emerging fraud types. **Protecting 65% of the world's credit card transactions, Falcon detects fraud** with pinpoint accuracy via **proven neural network models** and other proprietary predictive technologies. Debit, credit, oil and retail card issuers in numerous marketplaces rely on Falcon to detect and stop fraudulent transactions - and combat identity theft - in real time.

**The network:** *several perceptrons trained with Gallant's pocket algorithm with ratchet*

# Convolutional Neural Networks



Dr. Wojtek Kowalczyk

[wojtek@liacs.nl](mailto:wojtek@liacs.nl)

# Why do we need many layers?



- In theory, one hidden layer is sufficient to model any function with arbitrary accuracy; however, the number of required nodes and weights grows exponentially fast
- The deeper the network the less nodes are required to model "complicated" functions
- Consecutive layers "learn" features of the training patterns; from simplest (lower layers) to more complicated (top layers)
- Visual cortex consists of about 10 layers

# Why can't we just add more layers?

---

- In practice, "classical" multi-layer networks work fine only for a very small number of hidden layers (typically 1 or 2) - this is an empirical fact ...
- Adding layers is harmful because:
  - the increased number of weights quickly leads to data overfitting (lack of generalization)
  - huge number of (bad) local minima trap the gradient descent algorithm
  - vanishing or exploding gradients (the update rule involves products of many numbers) cause additional problems

# A disturbing observation

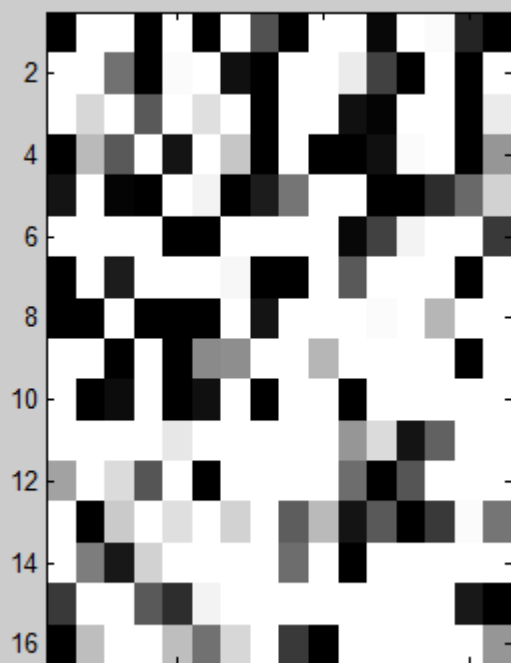
---

- Consider the *digit recognition problem* (16x16)
- Let us modify the images by *randomly permuting all pixels*: take a random permutation  $p$  and change every image  $x[0:16 \times 16]$  into  $x[p(0:16 \times 16)]$
- What accuracy can be achieved by a single layer perceptron on such a “randomly permuted” data?

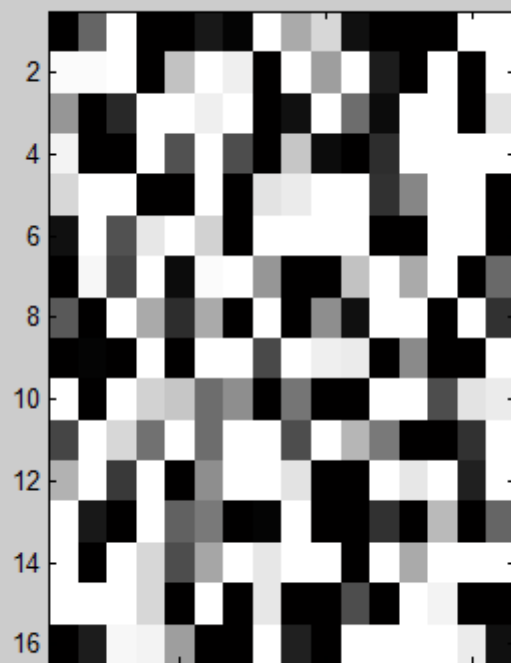
**THE SAME AS ON THE ORIGINAL DATA!**

*(the same holds for a multi-layer perceptron)*

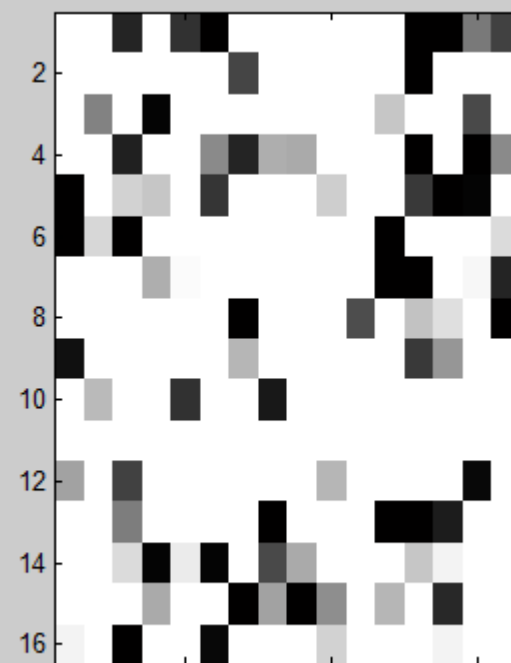
Digit: 6



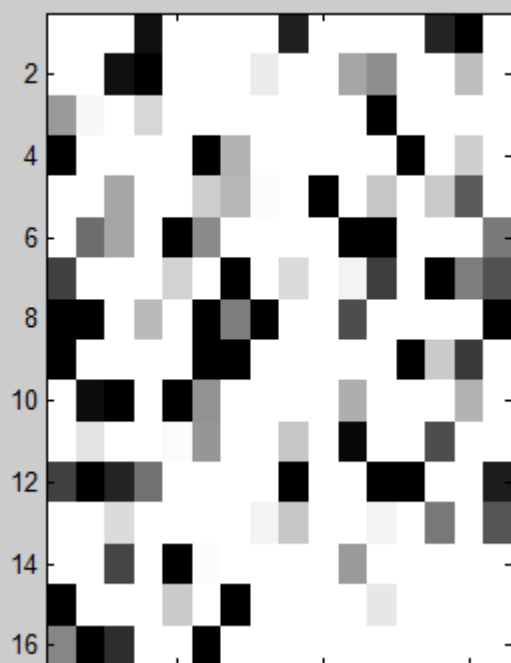
Digit: 5



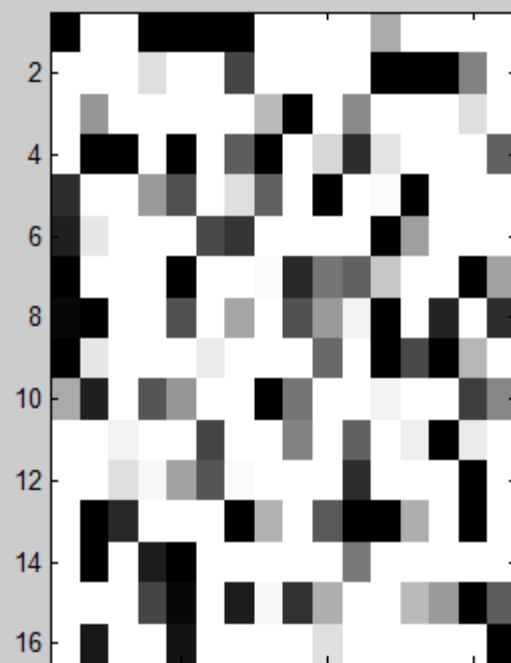
Digit: 7



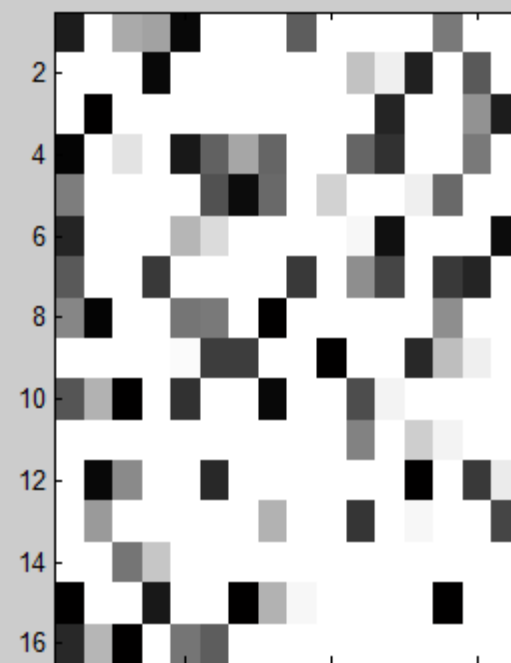
Digit: 4



Digit: 3



Digit: 6



# MNIST data set

---



training:  
60.000 images

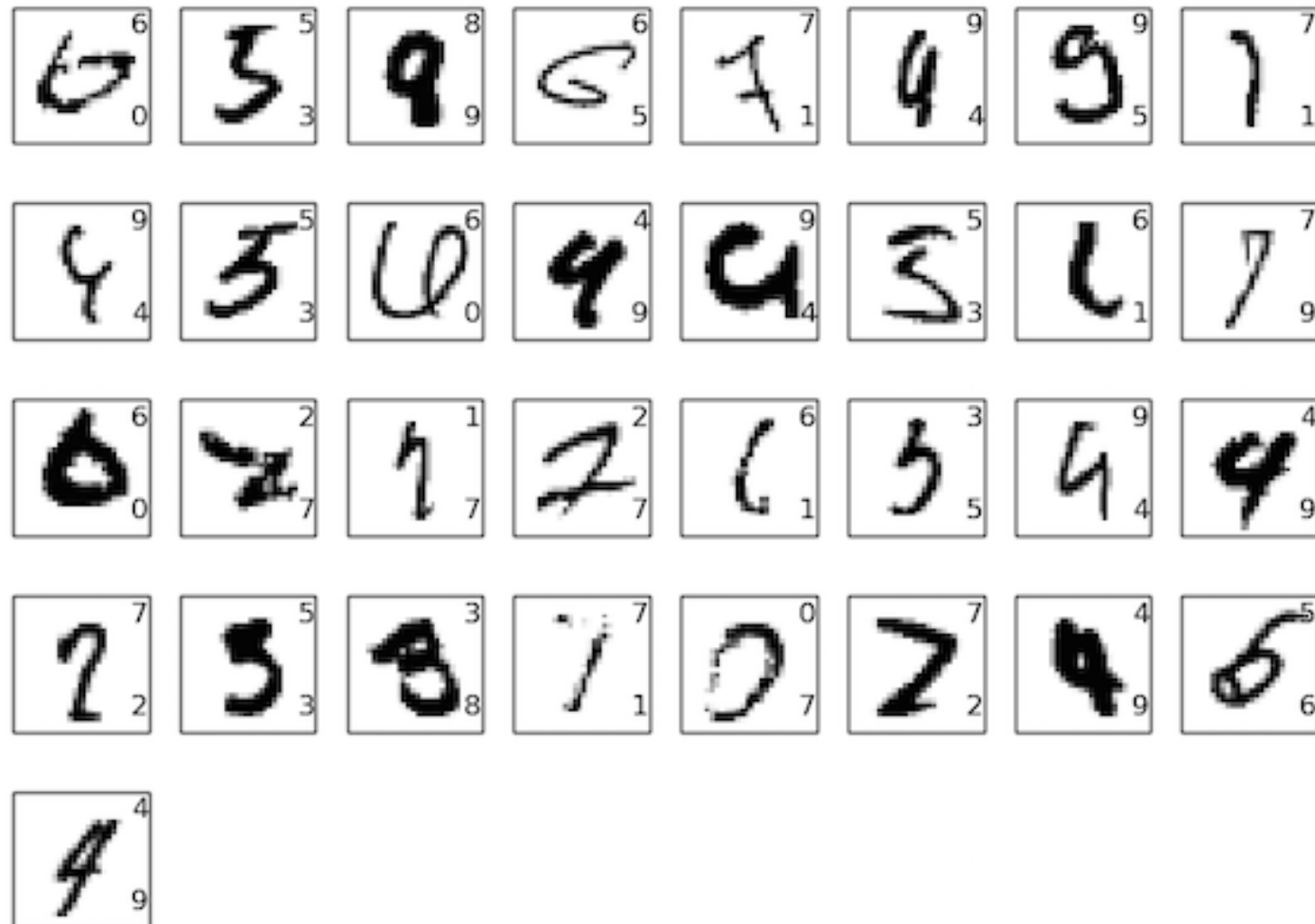
testing:  
10.000 images

each image:  
32x32 pixels

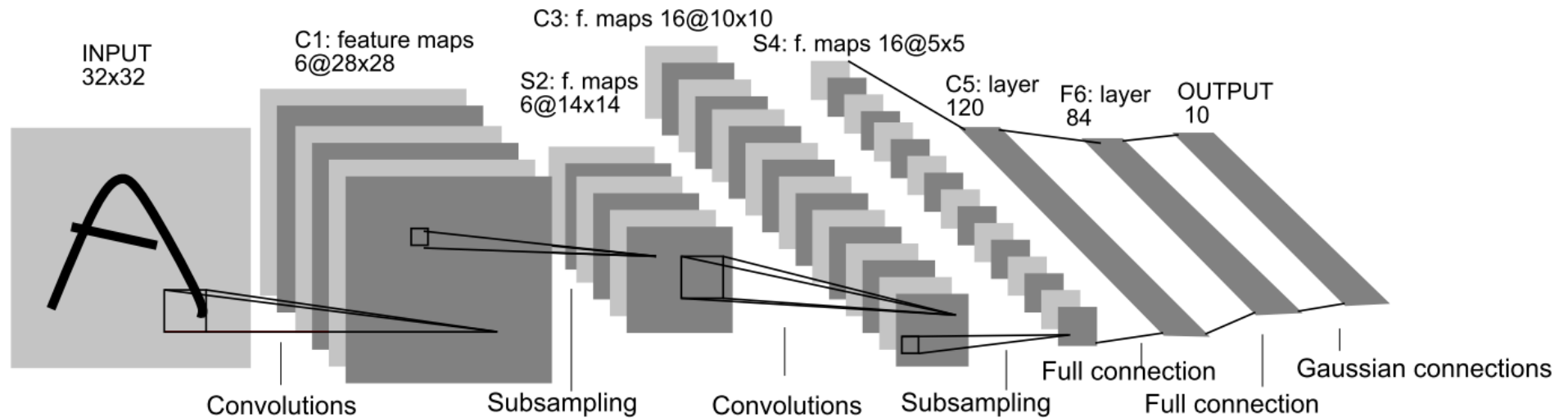
accuracy: 99.7%  
(on the test set)



# All 33 misclassified digits



# LeNet5



- Input: 32x32 pixel image
- Cx: Convolutional layer
- Sx: Subsample layer  
(reduces image size by averaging 2x2 patches)
- Fx: Fully connected layer

# A Convolutional Filter

Let us suppose that in an input image we want to find locations that look like a 3x3 cross. Define a matrix **F** (called a **kernel**, **receptive field** or **convolutional filter**) and "convolve it" with all possible locations in the image. We will get another (smaller) matrix with "degrees of overlap":

$$F = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

*"multiply and add"*

1 <sub>x1</sub>	1 <sub>x0</sub>	1 <sub>x1</sub>	0	0
0 <sub>x0</sub>	1 <sub>x1</sub>	1 <sub>x0</sub>	1	0
0 <sub>x1</sub>	0 <sub>x0</sub>	1 <sub>x1</sub>	1	1
0	0	1	1	0
0	1	1	0	0

Image

4		

Convolved  
Feature

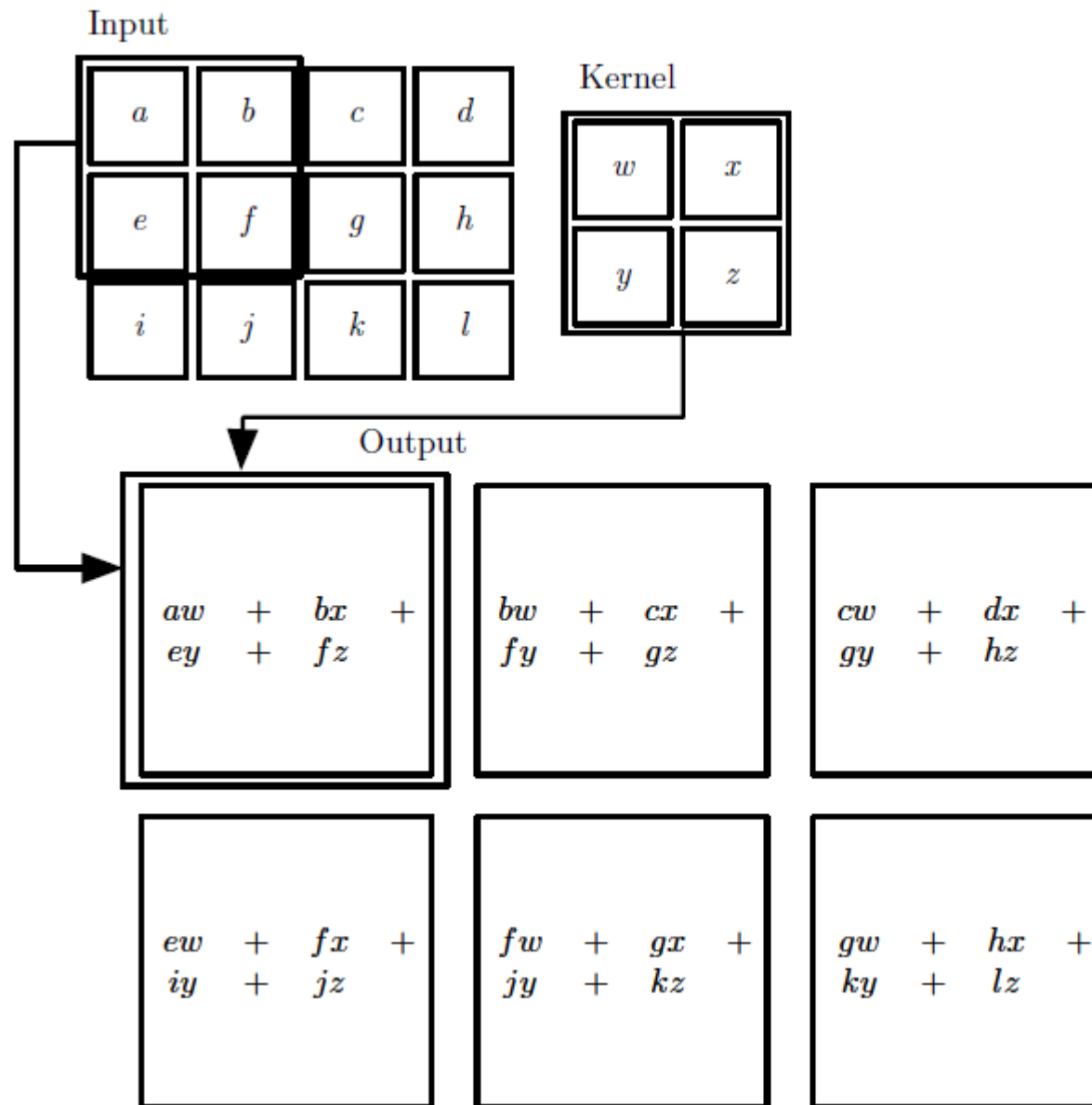


Figure 9.1 (the “Deep Learning” textbook)

# Motivation

---

A filter: a “feature detector” – returns high values when the corresponding patch is similar to the filter matrix

Think about all pixels being **-1** (black) or **+1** (white) and filter parameters also restricted to **-1** and **1**

Example: what features are “detected” by:

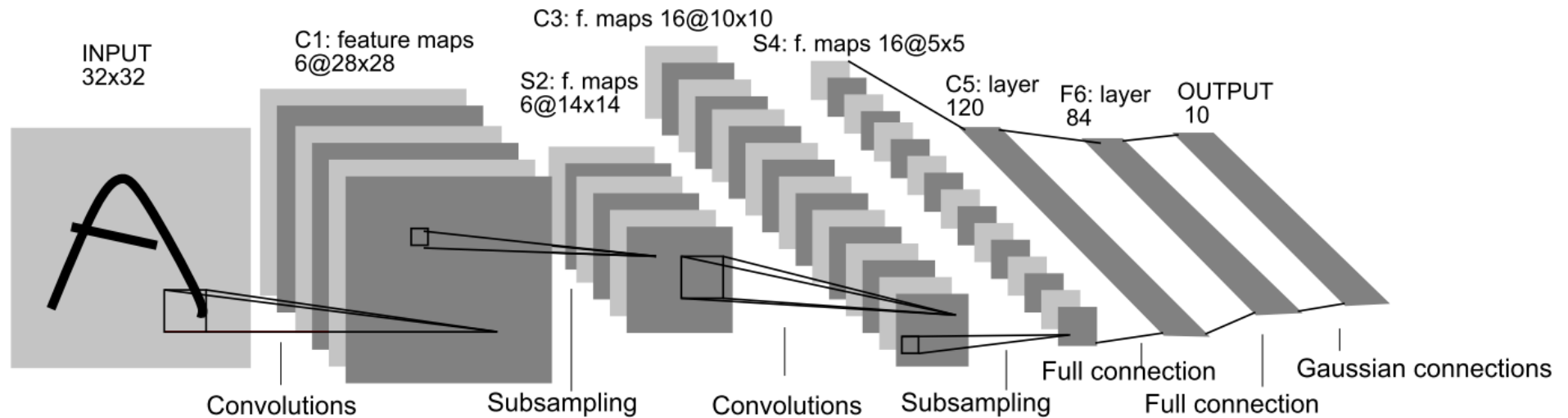
+	-	-
-	+	-
-	-	+

+	-	+
-	+	-
+	-	+

-	+	-
-	+	-
-	+	-

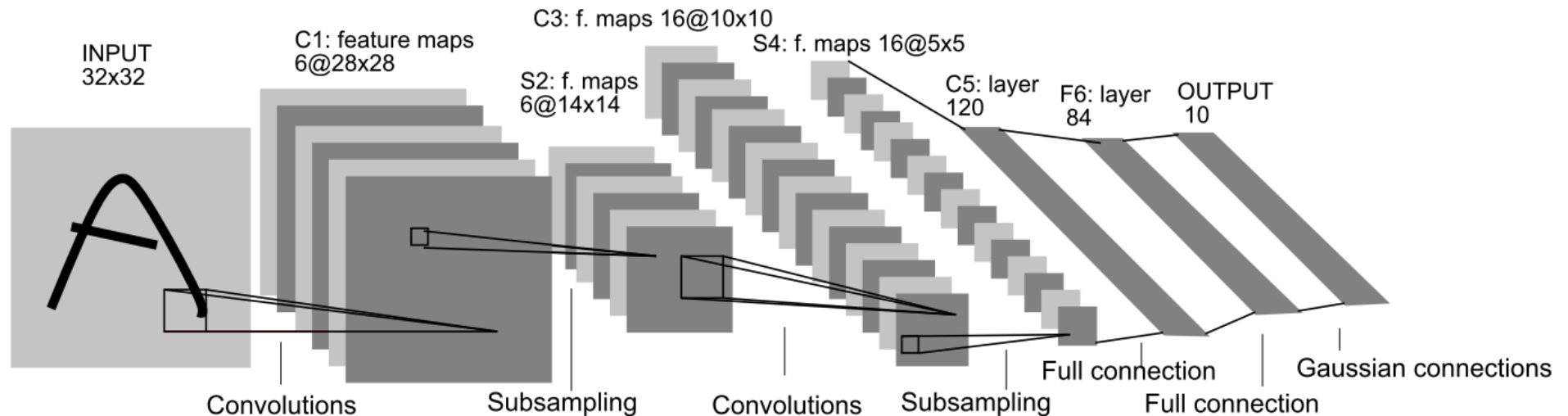
-	-	-
-	+	-
-	-	-

# LeNet5



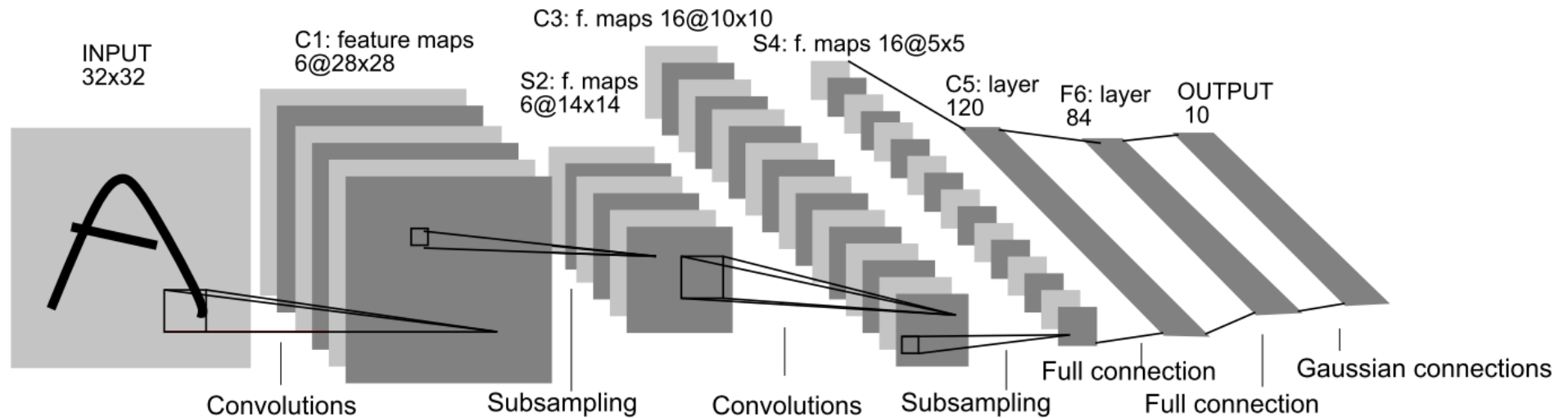
- Input: 32x32 pixel image
- Cx: Convolutional layer
- Sx: Subsample layer  
(reduces image size by averaging 2x2 patches)
- Fx: Fully connected layer

# LeNet 5: Layer C1



- C1: Convolutional layer with 6 feature maps of size 28x28.
- Each unit of C1 has a 5x5 receptive field in the input layer.
- Shared weights  $(5*5+1)*6=156$  parameters to learn  
Connections:  $28*28*(5*5+1)*6=122304$
- If it was fully connected we had:  
 $(32*32+1)*(28*28)*6 = 4.821.600$  parameters

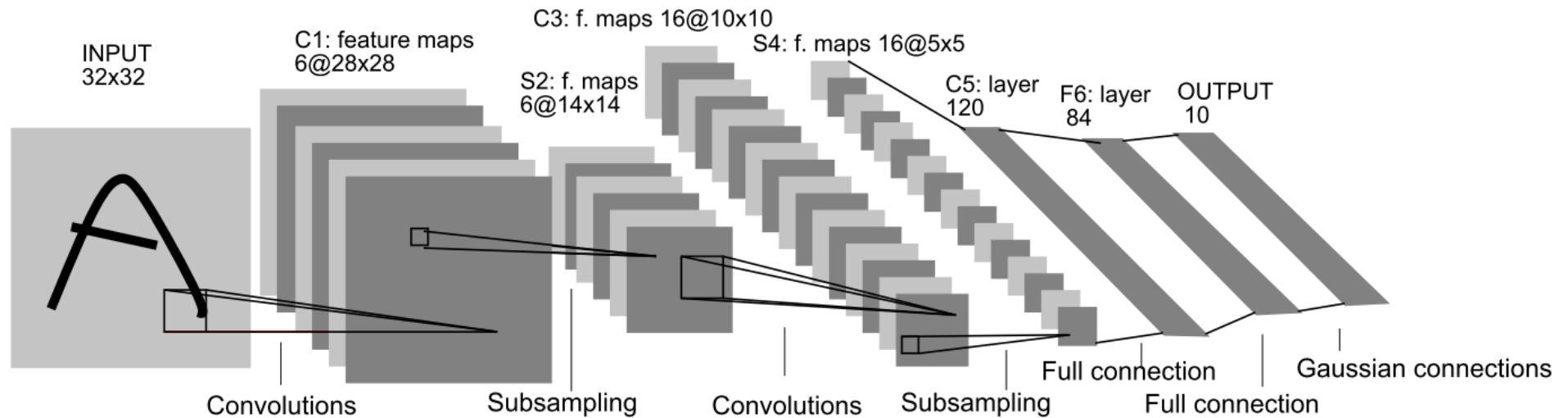
# LeNet 5: Layer S2



- S2: Subsampling layer with 6 feature maps of size 14x14 2x2 nonoverlapping receptive fields in C1
- Layer S2:  $6 \times 2 = 12$  trainable parameters.
- Connections:  $14 \times 14 \times (2 \times 2 + 1) \times 6 = 5880$



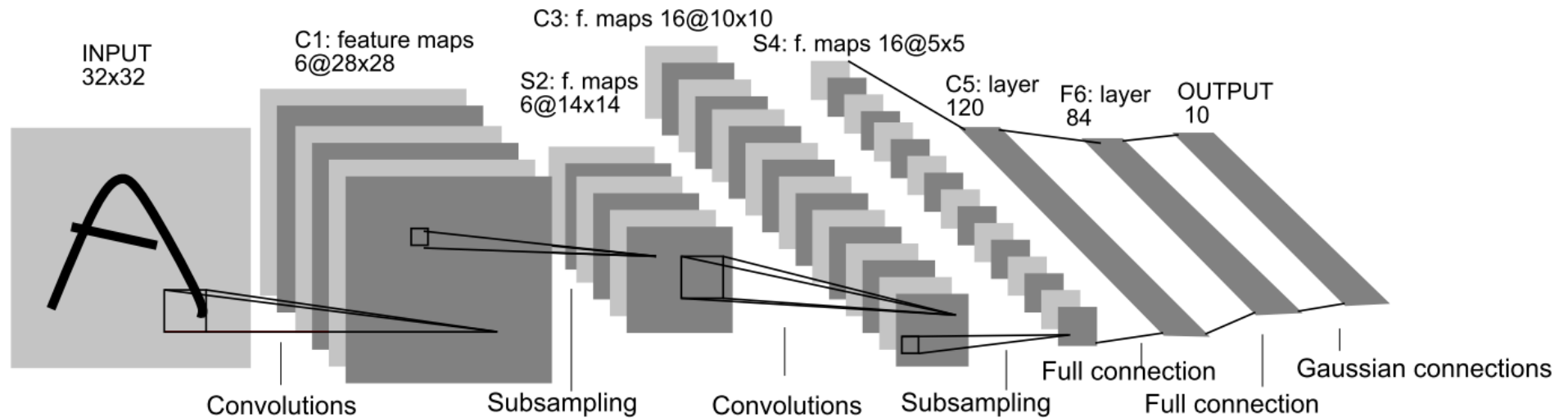
... and so on ...



Study slides 11-27 of [DeepLearning.pdf](#)

Read the original paper [lecun-01a.pdf](#)

# LeNet 5: totals



- The whole network has:
  - 1256 nodes
  - 64.660 connections
  - **9.760 trainable parameters (and not millions!)**
  - **trained with the Backpropagation algorithm!**

# Misclassified cases

4 4→6	3 3→5	8 8→2	1 2→1	3 5→3	4 4→8	2 2→8	3 3→5	6 6→5	7 7→3
4 9→4	8 8→0	7 7→8	5 5→3	8 8→7	6 0→6	3 3→7	2 2→7	8 8→3	4 9→4
8 8→2	3 5→3	4 4→8	3 3→9	6 6→0	9 9→8	4 4→9	6 6→1	9 9→4	9 9→1
9 9→4	2 2→0	6 6→1	3 3→5	3 3→2	9 9→5	6 6→0	6 6→0	6 6→0	6 6→8
4 4→6	7 7→3	9 9→4	4 4→6	2 2→7	9 9→7	4 4→3	9 9→4	9 9→4	9 9→4
2 8→7	4 4→2	8 8→4	3 3→5	8 8→4	6 6→5	8 8→5	3 3→8	3 3→8	9 9→8
1 1→5	9 9→8	6 6→3	0 0→2	6 6→5	9 9→5	0 0→7	1 1→6	4 4→9	2 2→1
2 2→8	8 8→5	4 4→9	7 7→2	7 7→2	6 6→5	9 9→7	6 6→1	5 5→6	5 5→0
4 4→9	2 2→8								

# From LeNet5 to ImageNet (2010/2012)

---

## ImageNet

- 15M images
- 22K categories
- Images collected from Web
- RGB Images
- Variable-resolution
- Human labelers (Amazon's Mechanical Turk crowd-sourcing)

## ImageNet Large Scale Visual Recognition Challenge (ILSVRC-2010)

- 1K categories
- 1.2M training images (~1000 per category)
- 50,000 validation images
- 150,000 testing images

# ImageNet (study slides 28-40)

---

- ILSVRC-2010 test set

Model	Top-1	Top-5
<i>Sparse coding</i> [2]	47.1%	28.2%
<i>SIFT + FVs</i> [24]	45.7%	25.7%
CNN	<b>37.5%</b>	<b>17.0%</b>

- ILSVRC-2012 test set

Model	Top-1 (val)	Top-5 (val)	Top-5 (test)
<i>SIFT + FVs</i> [7]	—	—	26.2%
1 CNN	40.7%	18.2%	—
5 CNNs	38.1%	16.4%	<b>16.4%</b>
1 CNN*	39.0%	16.6%	—
7 CNNs*	<b>36.7%</b>	15.4%	<b>15.3%</b>

# Key Points



- convolutions, feature maps, kernels, ...
- subsampling/pooling
- weights sharing
- ReLU (Rectified Linear Unit)
- Data Augmentation
- Dropout

# Homework

- Study slides 11-40 of DeepLearning.pdf
- Study Chapter 6 of Nielsen's book:  
<http://neuralnetworksanddeeplearning.com/chap6.html>
- Read (but don't get intimidated!) Chapter 9 of the "Deep Learning" textbook
- How a random permutation of input pixels (in the training and testing sets) would affect the accuracy of CNNs?

# Nesterov Momentum

“On the importance of initialization and momentum in deep learning”  
(Sutskever et al., ICML 2013)

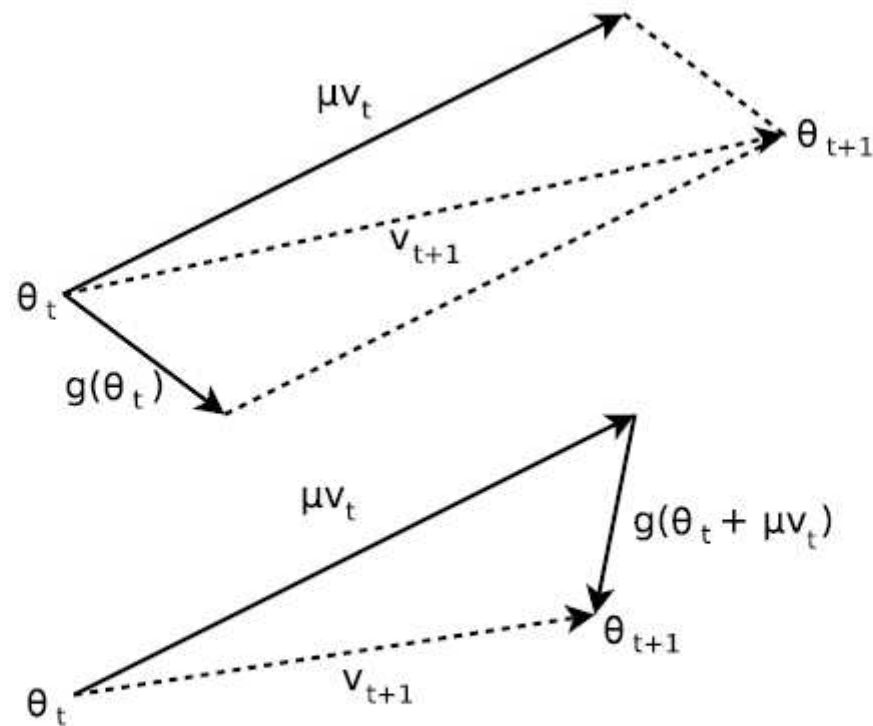


Figure 1. **(Top)** Classical Momentum **(Bottom)** Nesterov Accelerated Gradient



# SGD



---

**Algorithm 8.1** Stochastic gradient descent (SGD) update at training iteration  $k$

---

**Require:** Learning rate  $\epsilon_k$ .

**Require:** Initial parameter  $\theta$

**while** stopping criterion not met **do**

    Sample a minibatch of  $m$  examples from the training set  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$  with corresponding targets  $\mathbf{y}^{(i)}$ .

    Compute gradient estimate:  $\hat{\mathbf{g}} \leftarrow +\frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

    Apply update:  $\theta \leftarrow \theta - \epsilon \hat{\mathbf{g}}$

**end while**

---

# SGD with Momentum



---

**Algorithm 8.2** Stochastic gradient descent (SGD) with momentum

---

**Require:** Learning rate  $\epsilon$ , momentum parameter  $\alpha$ .

**Require:** Initial parameter  $\theta$ , initial velocity  $v$ .

**while** stopping criterion not met **do**

    Sample a minibatch of  $m$  examples from the training set  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$  with corresponding targets  $\mathbf{y}^{(i)}$ .

    Compute gradient estimate:  $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

    Compute velocity update:  $\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \mathbf{g}$

    Apply update:  $\theta \leftarrow \theta + \mathbf{v}$

**end while**

---

# SGD with Nesterov Momentum

---

---

**Algorithm 8.3** Stochastic gradient descent (SGD) with Nesterov momentum

---

**Require:** Learning rate  $\epsilon$ , momentum parameter  $\alpha$ .

**Require:** Initial parameter  $\theta$ , initial velocity  $v$ .

**while** stopping criterion not met **do**

    Sample a minibatch of  $m$  examples from the training set  $\{x^{(1)}, \dots, x^{(m)}\}$  with corresponding labels  $y^{(i)}$ .

    Apply interim update:  $\tilde{\theta} \leftarrow \theta + \alpha v$

    Compute gradient (at interim point):  $g \leftarrow \frac{1}{m} \nabla_{\tilde{\theta}} \sum_i L(f(x^{(i)}; \tilde{\theta}), y^{(i)})$

    Compute velocity update:  $v \leftarrow \alpha v - \epsilon g$

    Apply update:  $\theta \leftarrow \theta + v$

**end while**

---

# ImageNet Classification with Deep Convolutional Neural Networks

Alex Krizhevsky, Ilya Sutskever, Geoffrey Hinton,  
Advances in Neural Information Processing Systems 2012

# ImageNet

- ❑ 15M images
- ❑ 22K categories
- ❑ Images collected from Web
- ❑ Human labelers (Amazon's Mechanical Turk crowd-sourcing)
- ❑ ImageNet Large Scale Visual Recognition Challenge (ILSVRC-2010)
  - 1K categories
  - 1.2M training images (~1000 per category)
  - 50,000 validation images
  - 150,000 testing images
- ❑ RGB images
- ❑ Variable-resolution, but this architecture scales them to 256x256 size



# ImageNet

## Classification goals:

- ❑ Make 1 guess about the label (Top-1 error)
- ❑ make 5 guesses about the label (Top-5 error)



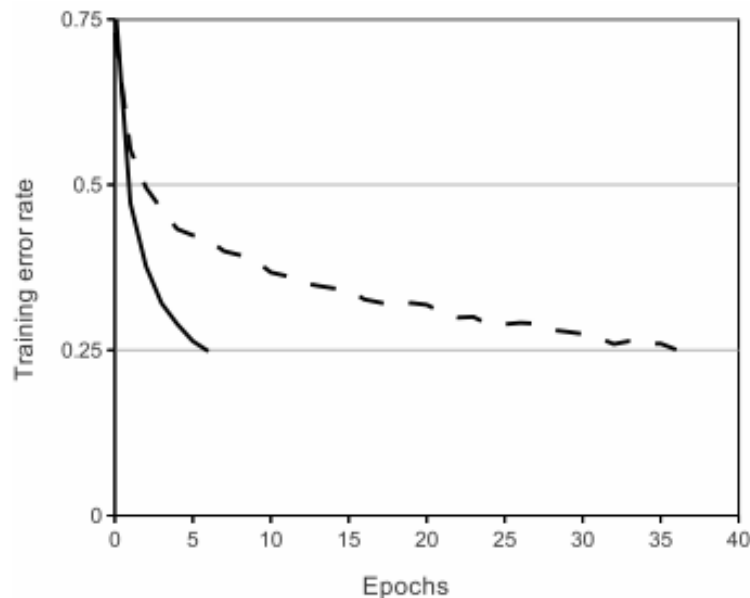
# The Architecture

Typical nonlinearities:  $f(x) = \tanh(x)$

$$f(x) = (1 + e^{-x})^{-1}$$

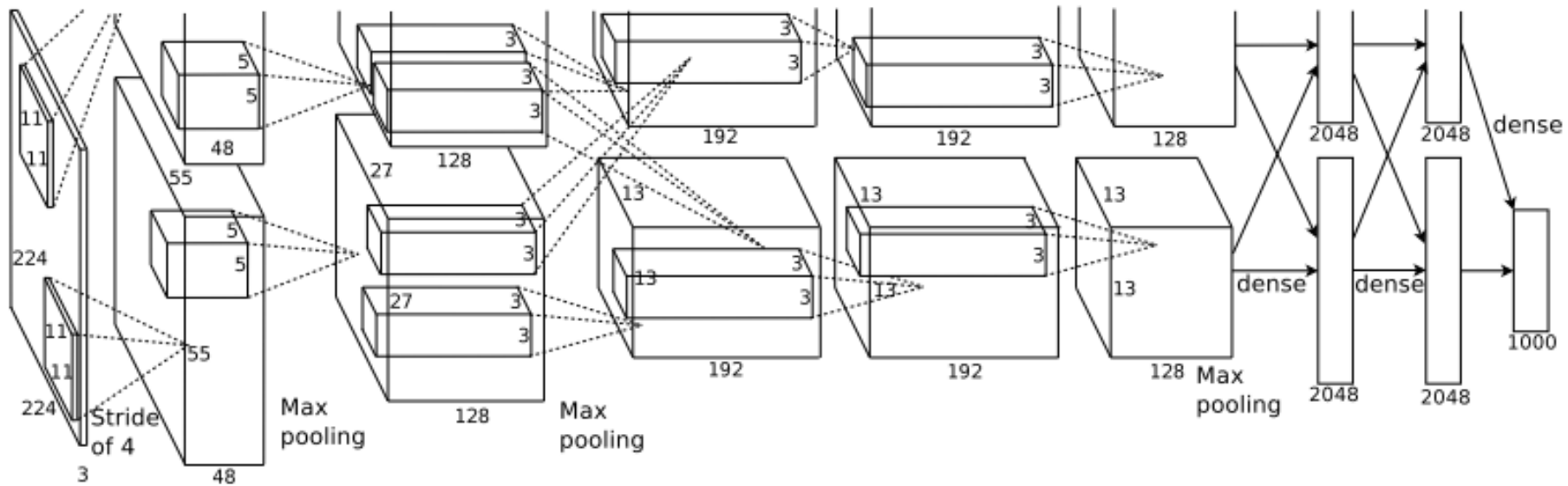
Here, however, Rectified Linear Units (ReLU) are used:  $f(x) = \max(0, x)$

**Empirical observation:** Deep convolutional neural networks with ReLUs train several times faster than their equivalents with tanh units



A four-layer convolutional neural network with ReLUs (solid line) reaches a 25% training error rate on CIFAR-10 six times faster than an equivalent network with tanh neurons (dashed line)

# The Architecture



**The first convolutional layer** filters the  $224 \times 224 \times 3$  input image with 96 kernels of size  $11 \times 11 \times 3$  with a stride of 4 pixels (this is the distance between the receptive field centers of neighboring neurons in the kernel map.  $224/4=56$

**The pooling layer:** form of non-linear down-sampling. Max-pooling partitions the input image into a set of rectangles and, for each such sub-region, outputs the maximum value



# The Architecture

- Trained with stochastic gradient descent
  - on two NVIDIA GTX 580 3GB GPUs
  - for about a week
- 
- ❑ 650,000 neurons
  - ❑ 60,000,000 parameters
  - ❑ 630,000,000 connections
  - ❑ 5 convolutional layer, 3 fully connected layer
  - ❑ Final feature layer: 4096-dimensional

# Data Augmentation

The easiest and most common method to **reduce overfitting** on image data is to artificially **enlarge the dataset** using label-preserving transformations.

We employ two distinct forms of data augmentation:

- image translation
- horizontal reflections
- changing RGB intensities

# Dropout

- ❑ We know that combining different models can be very useful (Mixture of experts, majority voting, boosting, etc)
- ❑ Training many different models, however, is very time consuming.

## **The solution:**

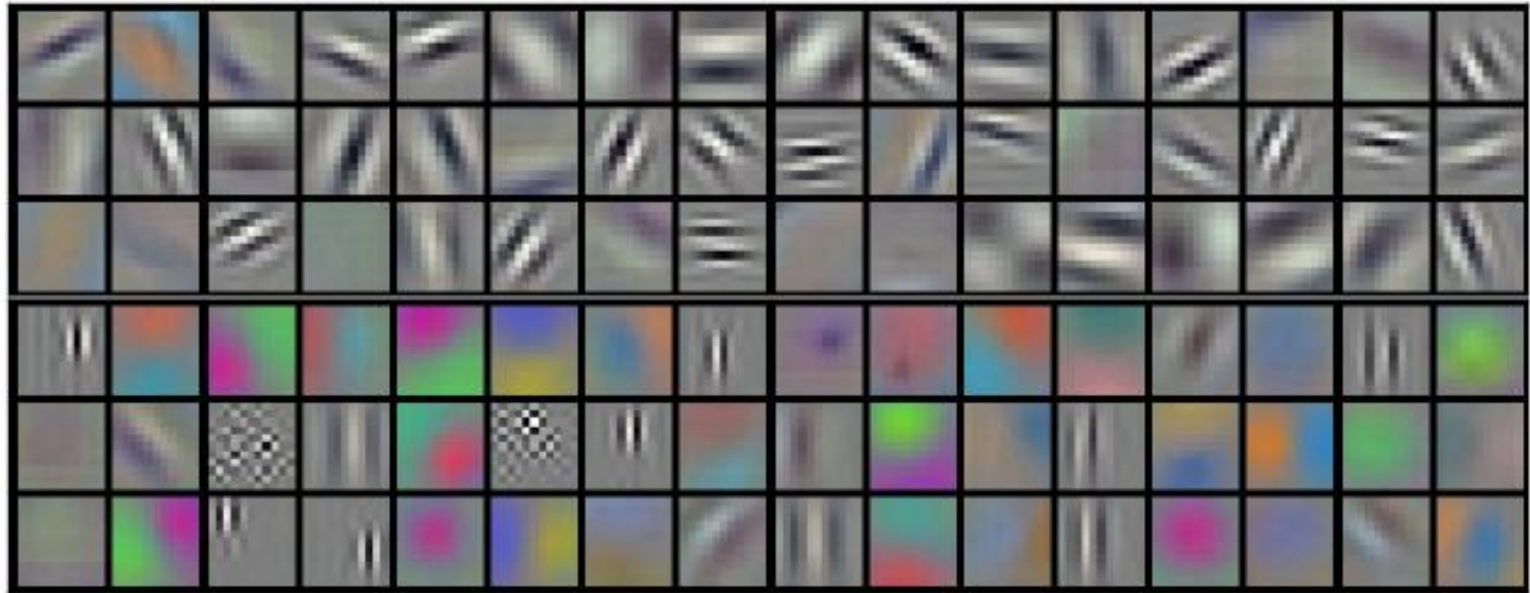
*Dropout*: set the output of each hidden neuron to zero w.p. 0.5.

# Dropout

**Dropout:** set the output of each hidden neuron to zero w.p. 0.5.

- The neurons which are “dropped out” in this way do not contribute to the forward pass and do not participate in backpropagation.
- So every time an input is presented, the neural network samples a different architecture, but all these architectures share weights.
- This technique reduces complex co-adaptations of neurons, since a neuron cannot rely on the presence of particular other neurons.
- It is, therefore, forced to learn more robust features that are useful in conjunction with many different random subsets of the other neurons.
- Without dropout, our network exhibits substantial overfitting.
- Dropout roughly doubles the number of iterations required to converge.

# The first convolutional layer



96 convolutional kernels of size  $11 \times 11 \times 3$  learned by the first convolutional layer on the  $224 \times 224 \times 3$  input images.

The top 48 kernels were learned on GPU1 while the bottom 48 kernels were learned on GPU2

Looks like Gabor wavelets, ICA filters...

# Results

## **Results on the test data:**

top-1 error rate: 37.5%

top-5 error rate: 17.0%

## **ILSVRC-2012 competition:**

15.3% accuracy

2<sup>nd</sup> best team: 26.2% accuracy

# Results



**mite**

**container ship**

**motor scooter**

**leopard**

mite black widow cockroach tick starfish	container ship lifeboat amphibian fireboat drilling platform	motor scooter go-kart moped bumper car golfcart	leopard jaguar cheetah snow leopard Egyptian cat



**grille**

**mushroom**

**cherry**

**Madagascar cat**

convertible grille pickup beach wagon fire engine	agaric mushroom jelly fungus gill fungus dead-man's-fingers	dalmatian grape elderberry ffordshire bullterrier currant	squirrel monkey spider monkey titi indri howler monkey



# Results: Image similarity



Test column

six training images that produce feature vectors in the last hidden layer with the smallest Euclidean distance from the feature vector for the test image.



# RBM, DBN, AutoEncoders

---

- Restricted Boltzmann Machines (RBMs)
  - energy model of probability distribution
  - training: Contrastive Divergence
  - why does it work?
- Deep Belief Networks
- Auto Encoders
- Denoising Auto Encoders
- Stacked Auto Encoders

# References

---

- R.B. Palm, Prediction as a candidate for learning deep hierarchical models of data Deep Belief Networks (MSc. Thesis, 2012)  
[github.com/rasmusbergpalm/DeepLearnToolbox](https://github.com/rasmusbergpalm/DeepLearnToolbox)

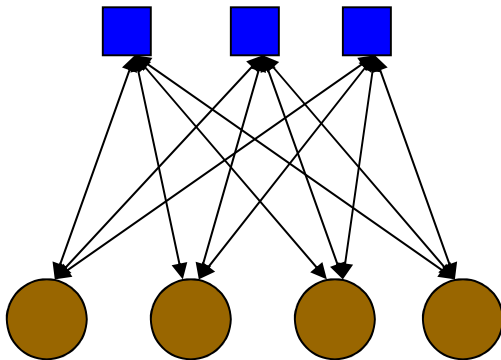
***Sections 1.2 (Methods) and 1.3 (Results)  
required for exam! (pages 9-27)***

- Hinton's tutorial on using RBMs  
[www.cs.toronto.edu/~hinton/absps/guideTR.pdf](http://www.cs.toronto.edu/~hinton/absps/guideTR.pdf)
- Kevin Duh course on Deep Learning (optional)  
<http://cl.naist.jp/~kevinduh/a/deep2014/>

# Restricted Boltzmann Machine

---

*Hidden  
layer ( $h$ )*



*Visible  
layer ( $x$ )*

- 2-layered network; **unsupervised!**
- $W=(w_{ij})$  : matrix of weights  
 $b = (b_i)$  : biases of visible layer  
 $c = (c_j)$  : biases of hidden layer
- nodes take values 0 or 1
- **model of probability distr. of  $P(x, h)$**
- non-deterministic behavior:

$$P(h_i = 1|x) = \text{sigm}(c_i + W_i x)$$

$$P(x_i = 1|h) = \text{sigm}(b_i + W_i h)$$

# RBM as a model of $P(x, h)$

---

$P(x, y)$  is defined as:

$$P(x, h) = \frac{e^{-E(x, h)}}{Z}$$

where **energy**  $E$  and **normalization**  $Z$  are given by:

$$E(x, h) = -b'x - c'h - h'Wx$$

$$Z(x, h) = \sum_{x, h} e^{-E(x, h)}$$

# Conditional Probabilities

---

$$P(h|x) = ???$$

$$P(x, h) = P(h|x)P(x), \text{ so } P(h|x) = P(x, h)/P(x)$$

Moreover,  $P(x) = \sum_h (P(x, h))$ , so:

$$P(h|x) = \frac{\exp(b'x + c'h + h'Wx)}{\sum_h \exp(b'x + c'h + h'Wx)}$$

$$P(h|x) = \frac{\prod_i \exp(c_i h_i + h_i W_i x)}{\prod_i \sum_h \exp(c_i h_i + h_i W_i x)}$$

$$P(h|x) = \prod_i \frac{\exp(h_i (c_i + W_i x))}{\sum_h \exp(h_i (c_i + W_i x))}$$

$$P(h|x) = \prod_i P(h_i|x)$$

$$P(x, h) = \frac{e^{-E(x, h)}}{Z}$$

# Conditional Probabilities (cont.)

---

Thus:

$$P(h_i = 1|x) = \frac{\exp(c_i + W_i x)}{1 + \exp(c_i + W_i x)}$$

$$P(h_i = 1|x) = \text{sigm}(c_i + W_i x)$$

and, by symmetry:

$$P(x_i = 1|h) = \text{sigm}(b_i + W_i h)$$

(Note, that

$$1/(1+\exp(-t)) = \exp(t)/(1+\exp(t))$$

so the “old” definition is equivalent to the new one)

# Training of RBM

---

- Purpose of training: for a given training set  $X$  find weights that maximize the log likelihood of  $X$ :

$$L(X) = \sum_{\{x \text{ in } X\}} \log(P(x))$$

- Use the gradient descend algorithm!
- Derivation of the gradient of  $L$  (see page 11) leads to a formula that involves two terms:
  - expected value of  $x_i h_j$  for known  $x$
  - expected value of  $x_i h_j$  over all vectors  $x$  (infeasible!)
- Hinton's trick: approximate the second term by iterating a random process for a few times (1 or 2, perhaps 3 times)  
=> Contrastive Divergence

# Derivation of gradients of L (page 11)

$$\frac{\partial}{\partial \theta} (-\log P(x)) = \frac{\partial}{\partial \theta} \left( -\log \sum_h P(x, h) \right) \quad \text{by definition}$$

$$= \frac{\partial}{\partial \theta} \left( -\log \sum_h \frac{\exp(-E(x, h))}{Z} \right) \quad \text{by definition of } P(x, h)$$

$$= -\frac{Z}{\sum_h \exp(-E(x, h))} \left( \sum_h \frac{1}{Z} \frac{\partial \exp(-E(x, h))}{\partial \theta} - \sum_h \frac{\exp(-E(x, h))}{Z^2} \frac{\partial Z}{\partial \theta} \right) \quad \begin{array}{l} \log(x)' = 1/x; \text{ chain rule,} \\ (f \cdot g)' = f' \cdot g + f \cdot g' \end{array}$$

$$= \sum_h \left( \frac{\exp(-E(x, h))}{\sum_h \exp(-E(x, h))} \frac{\partial E(x, h)}{\partial \theta} \right) + \frac{1}{Z} \frac{\partial Z}{\partial \theta}$$

$$= \sum_h P(h|x) \frac{\partial E(x, h)}{\partial \theta} - \frac{1}{Z} \sum_{x, h} \exp(-E(x, h)) \frac{\partial E(x, h)}{\partial \theta}$$

$$= \sum_h P(h|x) \frac{\partial E(x, h)}{\partial \theta} - \sum_{x, h} P(x, h) \frac{\partial E(x, h)}{\partial \theta}$$

$$= \mu_1 \left[ \frac{\partial E(x, h)}{\partial \theta} \Big| x \right] - \mu_1 \left[ \frac{\partial E(x, h)}{\partial \theta} \right]$$

$$\frac{\partial}{\partial W} (-\log P(x)) = \mu_1 [-h'x | x] - \mu_1 [-h'x]$$

$$\frac{\partial}{\partial b} (-\log P(x)) = \mu_1 [-x | x] - \mu_1 [-x]$$

$$\frac{\partial}{\partial c} (-\log P(x)) = \mu_1 [-h | x] - \mu_1 [-h]$$

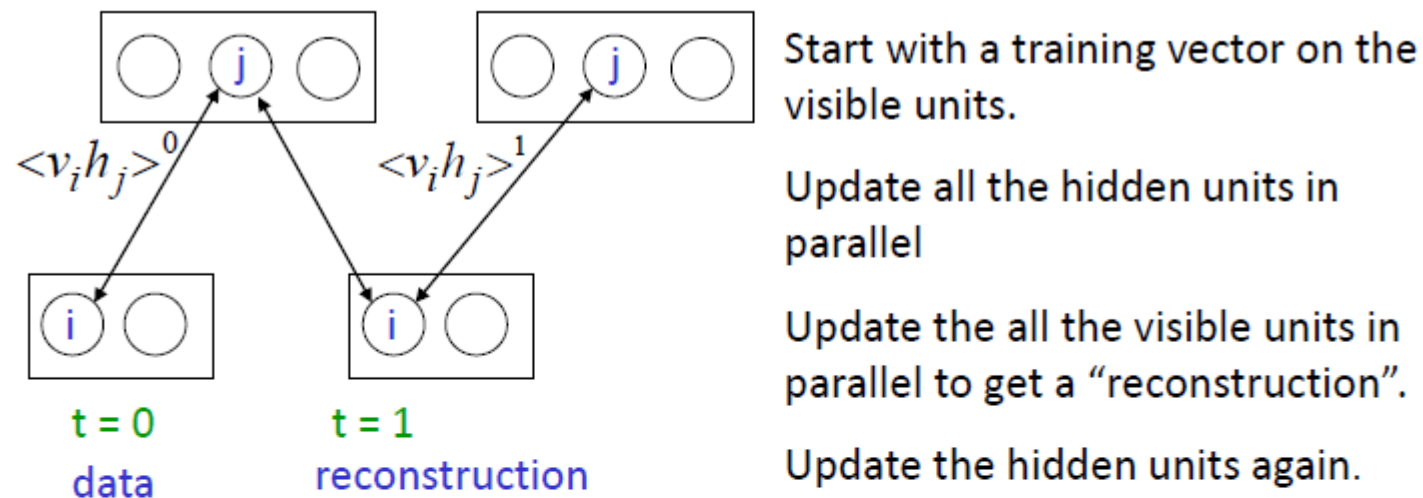
$$E(x, h) = -b'x - c'h - h'Wx$$

$$Z(x, h) = \sum_{x, h} e^{-E(x, h)}$$

$$P(x, h) = \frac{e^{-E(x, h)}}{Z}$$



# Approximation of gradients



$$\Delta w_{ij} = \varepsilon ( \langle v_i h_j \rangle^0 - \langle v_i h_j \rangle^1 )$$

**This is not following the gradient of the log likelihood.** But it works well. It is approximately following the gradient of another objective function (Carreira-Perpinan & Hinton, 2005).

# Motivation: page 12

---

- At each iteration the entire layer is updated. To get unbiased samples, we should initialize the model at some arbitrary state, and sample  $n$  times,  $n$  being a large number.
- To make this efficient, we'll do something slightly different. We'll initialize the model at a training sample, iterate one step, and use this as our negative sample. This is the contrastive divergence algorithm as introduced by Hinton [HOT06] with one step (CD-1).
- The logic is that, as the model distribution approaches the training data distribution, initializing the model to a training sample approximates letting the model converge.

# Training of RBM

---

---

## Algorithm 1 Contrastive Divergence 1

---

**for all** training samples as  $t$  **do**

$$x^{(0)} \leftarrow t$$

$$h^{(0)} \leftarrow \text{sigm}(x^{(0)}W + c) > \text{rand}()$$

$$x^{(1)} \leftarrow \text{sigm}(h^{(0)}W^T + b) > \text{rand}()$$

$$h^{(1)} \leftarrow \text{sigm}(x^{(1)}W + c) > \text{rand}()$$

$$W \leftarrow W + \alpha(x^{(0)}h^{(0)} - x^{(1)}h^{(1)})$$

$$b \leftarrow b + \alpha(x^{(0)} - x^{(1)})$$

$$c \leftarrow c + \alpha(h^{(0)} - h^{(1)})$$

**end for**

---

# More details

---

- To increase the accuracy (esp. in the final stage of the training process) iterate “up”-“down” updates several times
- In some cases, instead of using of  $x_i h_j$  it is better to use of  $x_i p(h_j|x)$
- Use mini-batches: process your data in small batches, updating weights after the whole mini-batch is processed:
  - “controlling randomness”: speeding up convergence
  - distributed implementations (GPU’s)
- more tips: A practical guide to training RBMs (Hinton):  
[www.cs.toronto.edu/~hinton/absps/guideTR.pdf](http://www.cs.toronto.edu/~hinton/absps/guideTR.pdf)

# Deep Belief Networks

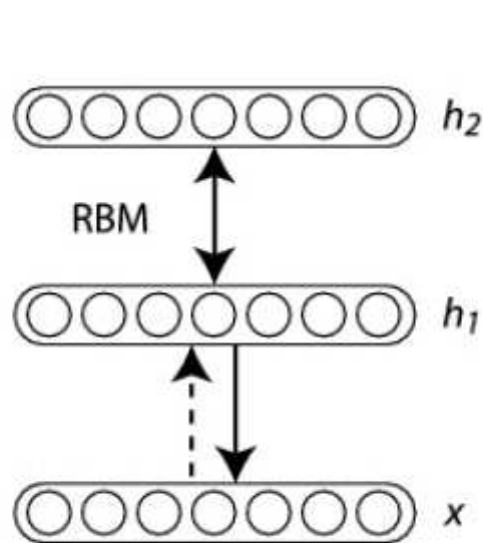
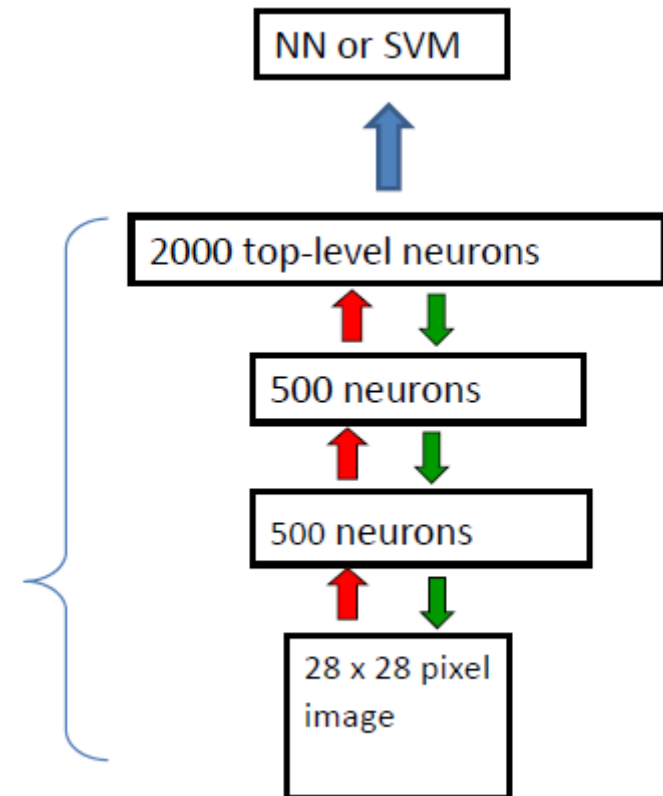
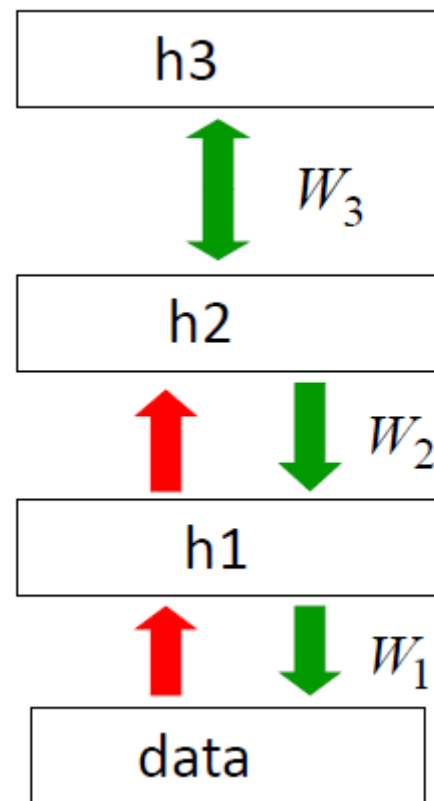


Figure 1.5: Deep Belief Network. Taken from



# Applications: A model of digit recognition

This is work from Hinton et al., 2006

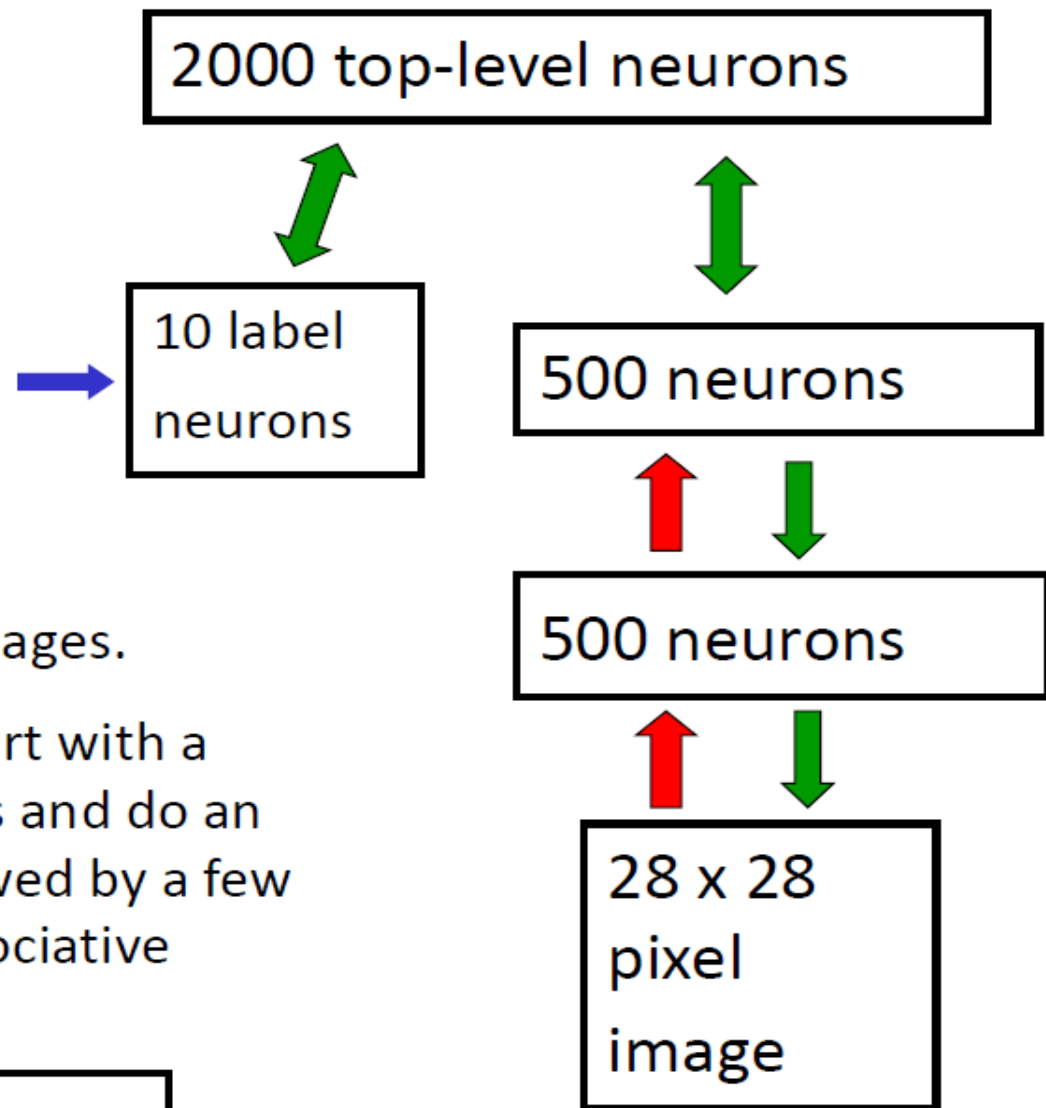
The top two layers form an associative memory whose energy landscape models the low dimensional manifolds of the digits.

The energy valleys have names

The model learns to generate combinations of labels and images.

To perform recognition we start with a neutral state of the label units and do an up-pass from the image followed by a few iterations of the top-level associative memory.

Matlab/Octave code available at <http://www.cs.utoronto.ca/~hinton/>



Slide modified from Hinton, 2007

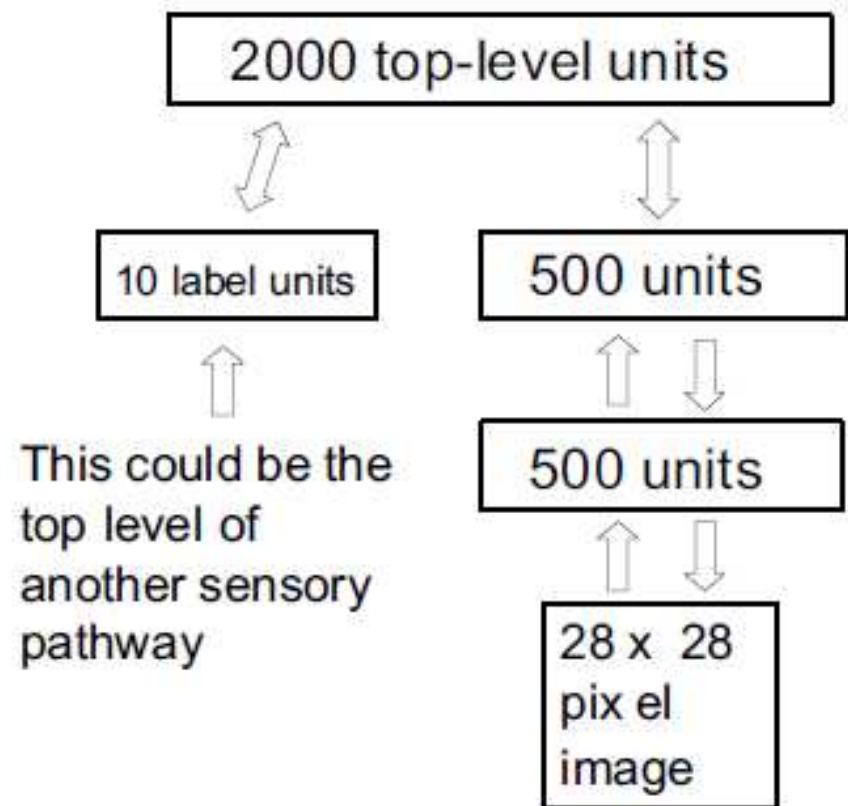
# Hinton's demo

<http://www.cs.toronto.edu/~hinton/adi/index.htm>

<http://www.cs.toronto.edu/~hinton/absps/fastnc.pdf>

DNN as a classifier

DNN as a generator



0	1	2	3	4
5	6	7	8	9

0	0	0	0	0	0
1	1	1	1	1	1
2	2	2	2	2	2
3	3	3	3	3	3
4	4	4	4	4	4
5	5	5	5	5	5
6	6	6	6	6	6
7	7	7	7	7	7
8	8	8	8	8	8
9	9	9	9	9	9



INCREASE SPEED

DETAILED VIEW





# More...

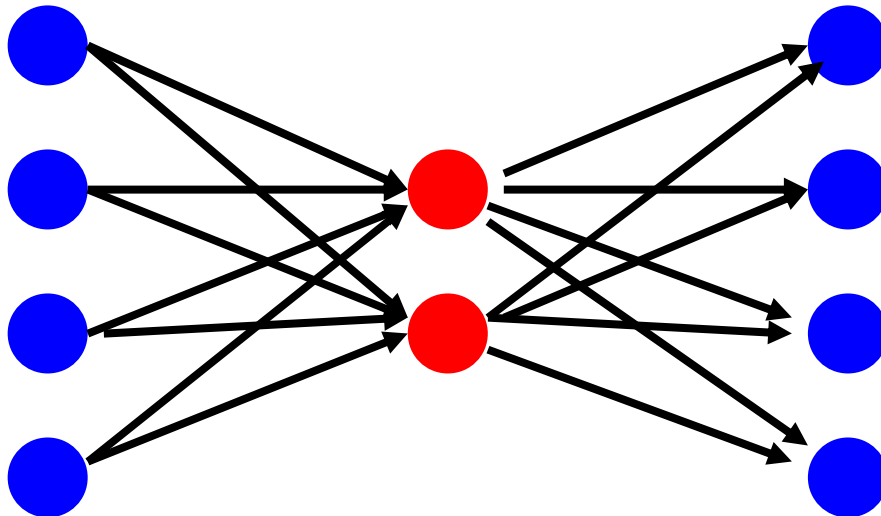


- Why DBN is better than MLP?
  - in MLP “delta’s” vanish (or explode) from layer to layer  
=> very few layers
  - RBMs “discover” intrinsic features of data  
=> a hierarchy of features
  - RBMs “pre-train” MLP to avoid local minima!
  - RBM is trained on unlabeled data  
it’s easy to get unlabeled data; labeled data is difficult to get

# Encoders, PCA, Compression

---

- Consider a 4:2:4 MLP (ENCODER):



- Inputs:  $(0\ 0\ 0\ 1)$ ,  $(0\ 0\ 1\ 0)$ ,  $(0\ 1\ 0\ 0)$ ,  $(1\ 0\ 0\ 0)$
- Outputs:  $(0\ 0\ 0\ 1)$ ,  $(0\ 0\ 1\ 0)$ ,  $(0\ 1\ 0\ 0)$ ,  $(1\ 0\ 0\ 0)$

# Encoders



- Encoder network "learns" the concept of binary representation of integers!
- General case:  $2^n:n:2^n$  networks
- Somehow the network is able to "extract" the most concise representation of the input!
- The same trick works for "real numbers": PCA networks
- It can be used for dimensionality reduction, data compression, and visualization.

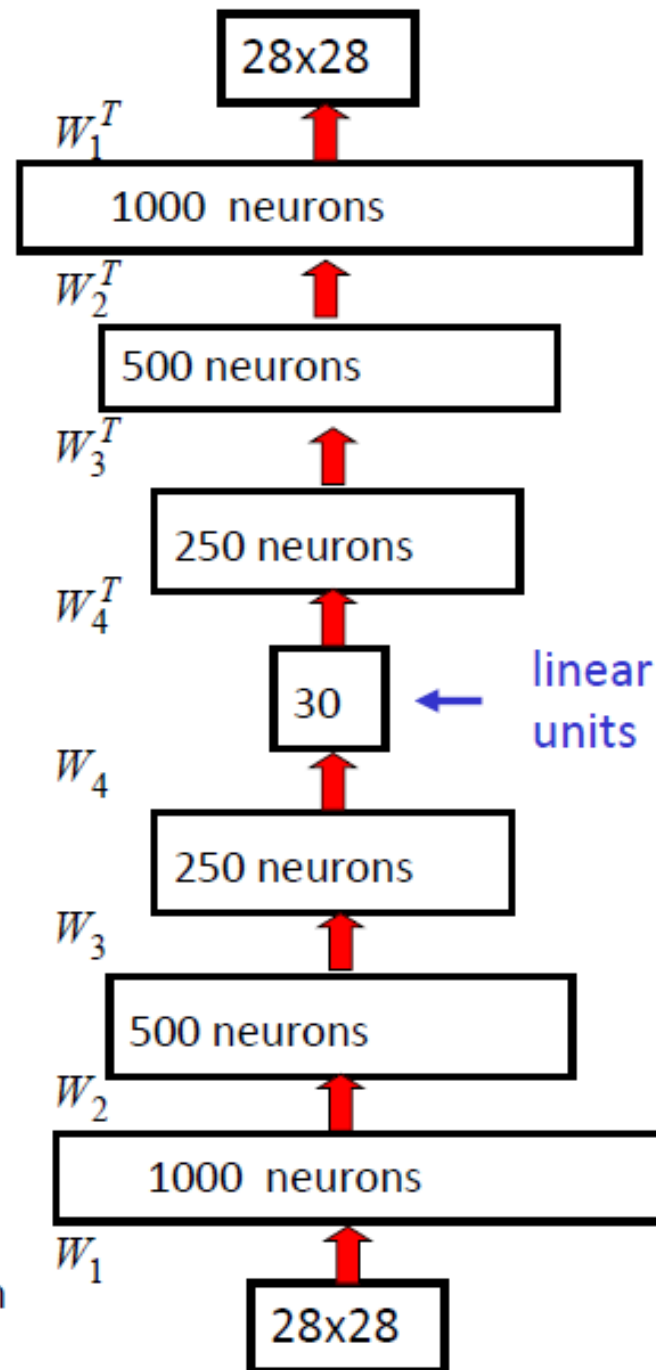
# Training Encoders: variants

---

- Backpropagation (good for shallow architectures)
- Enforced "symmetry of weights"  
(upper part = mirror of lower part)
- **Denoising Autoencoders**: the input layers gets a "noisy version of true  $x$ "; the target contains the original  $x$  ("noise": e.g., 10% of randomly selected pixels set to 0)
- **Deep Autoencoders**: the first "half" is trained unsupervised as a DBN (a stack of RBMs); the upper part is the "mirror" of the lower part; final phase trained by backpropagation

# Deep Autoencoders

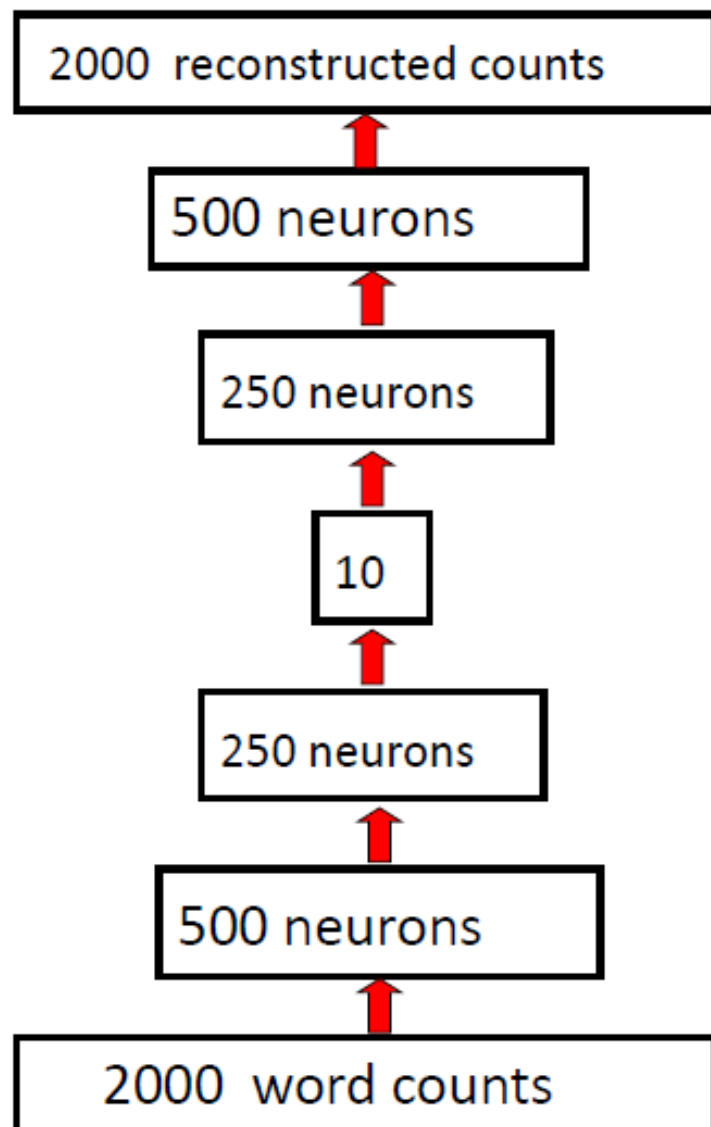
- They always looked like a really nice way to do non-linear dimensionality reduction:
  - But it is **very** difficult to optimize deep autoencoders using backpropagation.
- We now have a much better way to optimize them:
  - First train a stack of 4 RBM's
  - Then “unroll” them.
  - Then fine-tune with backprop.



## Applications: Classifying text documents

- A document can be characterized by the frequency of words that appear (ie, word counts for some dictionary become feature vector)
- Goals...
  1. Group/cluster similar documents
  2. Find similar documents

# How to compress the count vector

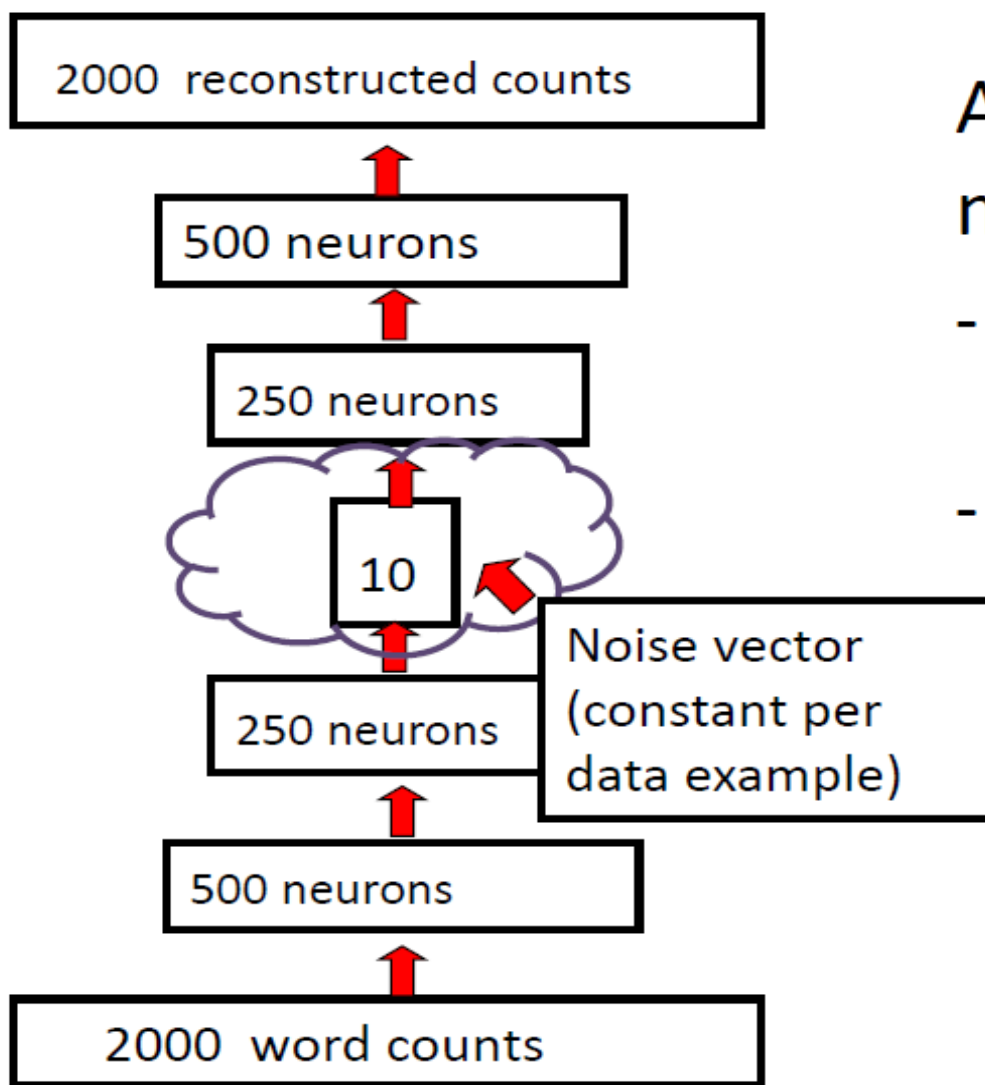


## Multi-layer auto-encoder

- Train a model to reproduce its input vector as its output
- This setup forces as much information as possible be compressed and passed thru the 10 numbers in the central bottleneck.
- These 10 numbers are then a good way to compare documents.

Slide modified from Hinton, 2007

# Search



Add noise to input to middle layer

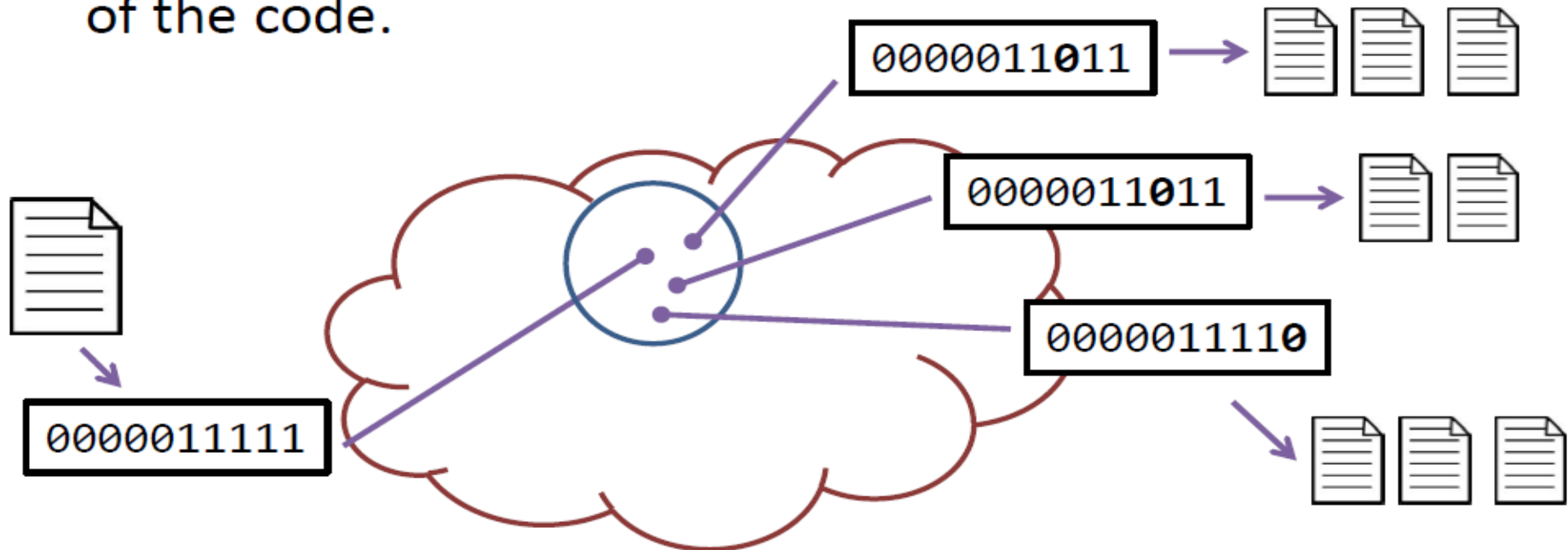
- Forces output to become bimodal
- Round values to 0 or 1 to form a binary vector (ie, code)



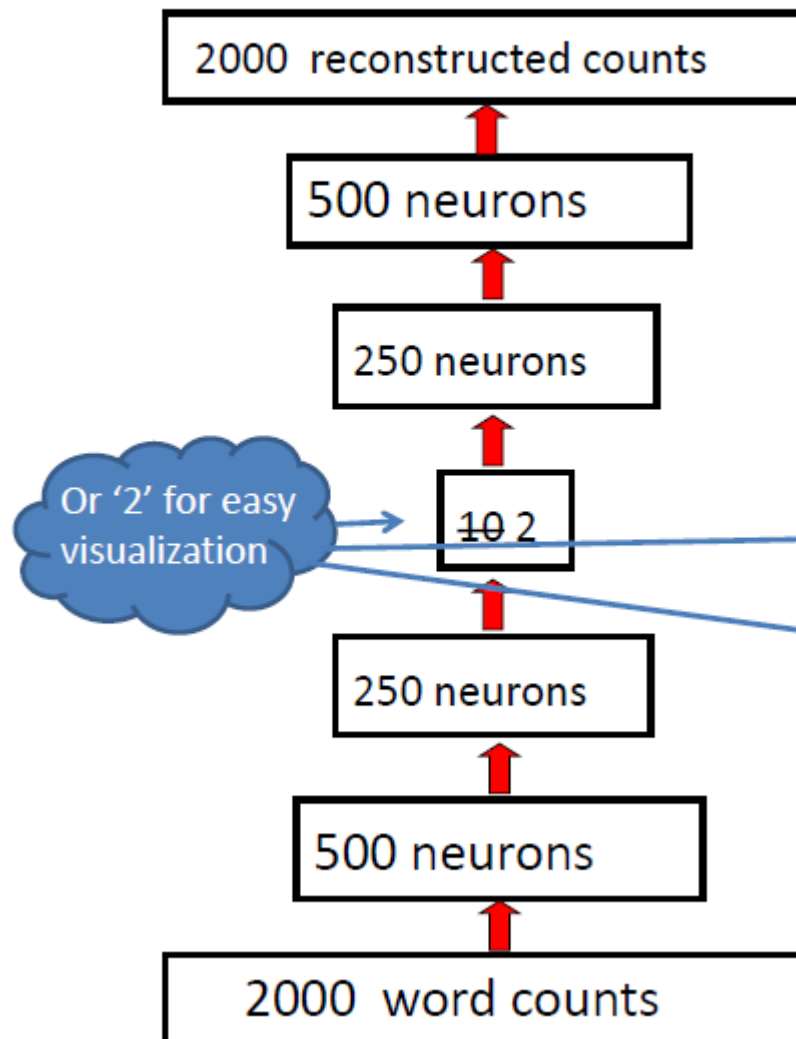
# Search

Use the binary codes as a key/hash documents

To find a similar document, calculate binary code and then retrieve documents that correspond to small deviations of the code.



# How to compress the count vector



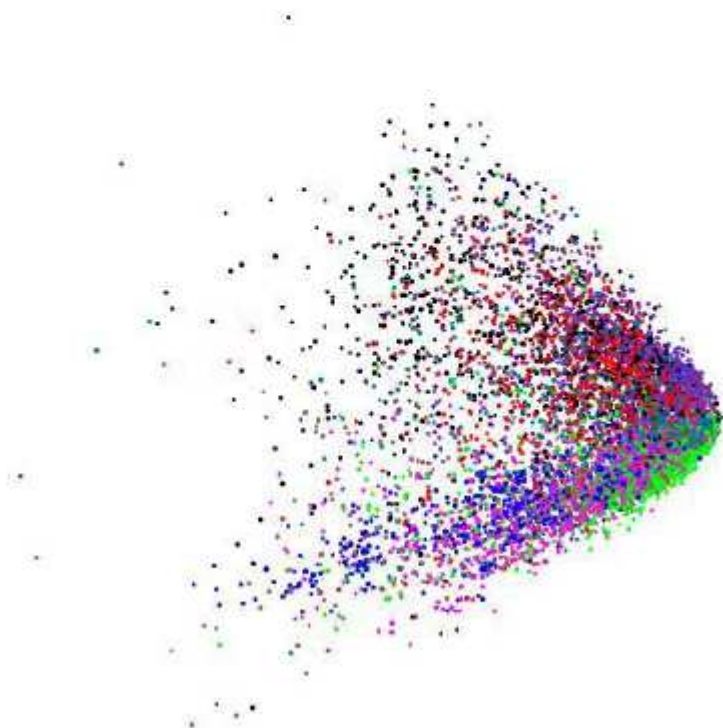
## Multi-layer auto-encoder

- Train a model to reproduce its input vector as its output
- This setup forces as much information as possible be compressed and passed thru the 10 2 numbers in the central bottleneck.
- These 10 2 numbers are then a good way to compare documents.

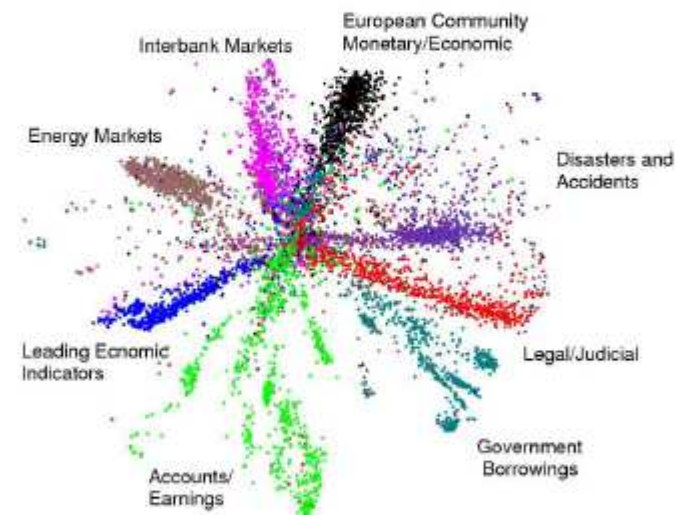
Slide modified from Hinton, 2007

# Clusters

LSA 2-D Topic Space

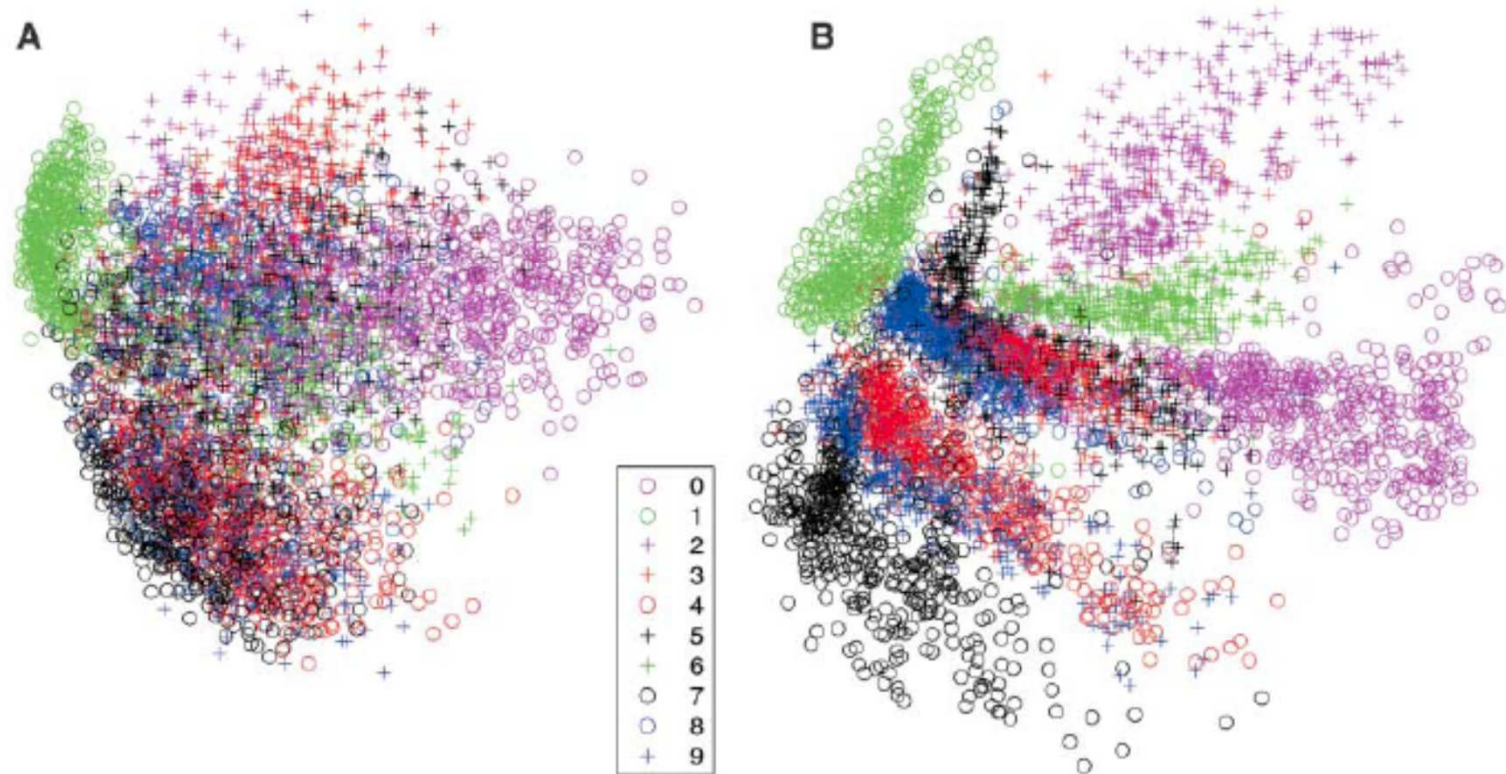


Autoencoder 2-D Topic Space



Images from Hinton, 2007

**Fig. 3.** (A) The two-dimensional codes for 500 digits of each class produced by taking the first two principal components of all 60,000 training images. (B) The two-dimensional codes found by a 784-1000-500-250-2 autoencoder. For an alternative visualization, see (8).



The same trick applied to "digits"