



# Machine Learning for Bioinformatics & Systems Biology

## 5. Selected topics

Marcel Reinders

*Delft University of Technology*

Perry Moerland

*Amsterdam UMC, University of Amsterdam*

Lodewyk Wessels

*Netherlands Cancer Institute*

*Some material courtesy of Robert Duin, David Tax, & Dick de Ridder*

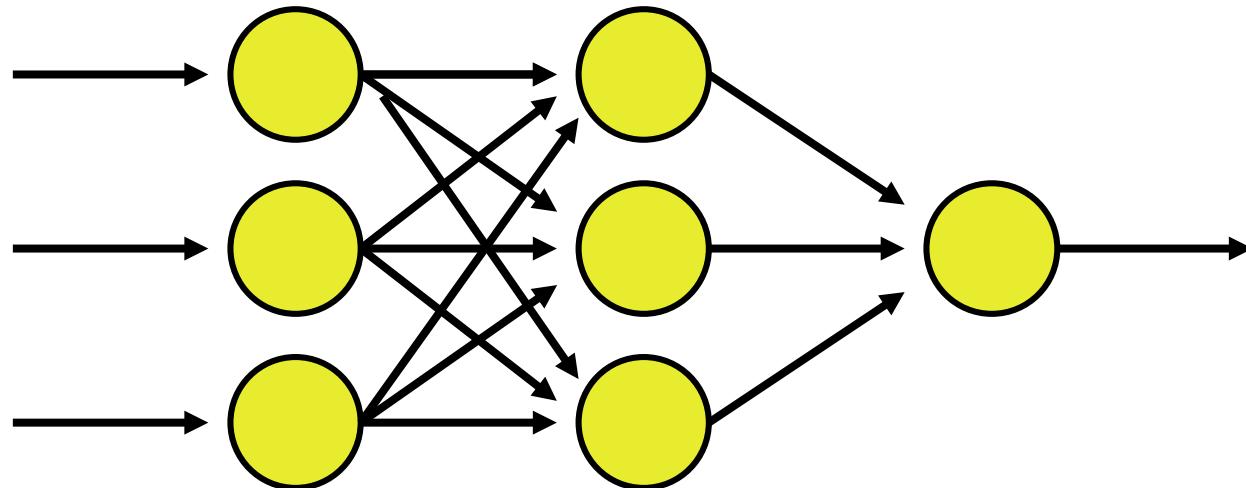
# Selected topics

- Famous classifiers
  - Artificial neural networks
  - Support vector classifiers
  - Classifier combination
- The fundamental pattern recognition trade-off
  - Complexity
- Recent developments

# Artificial neural networks

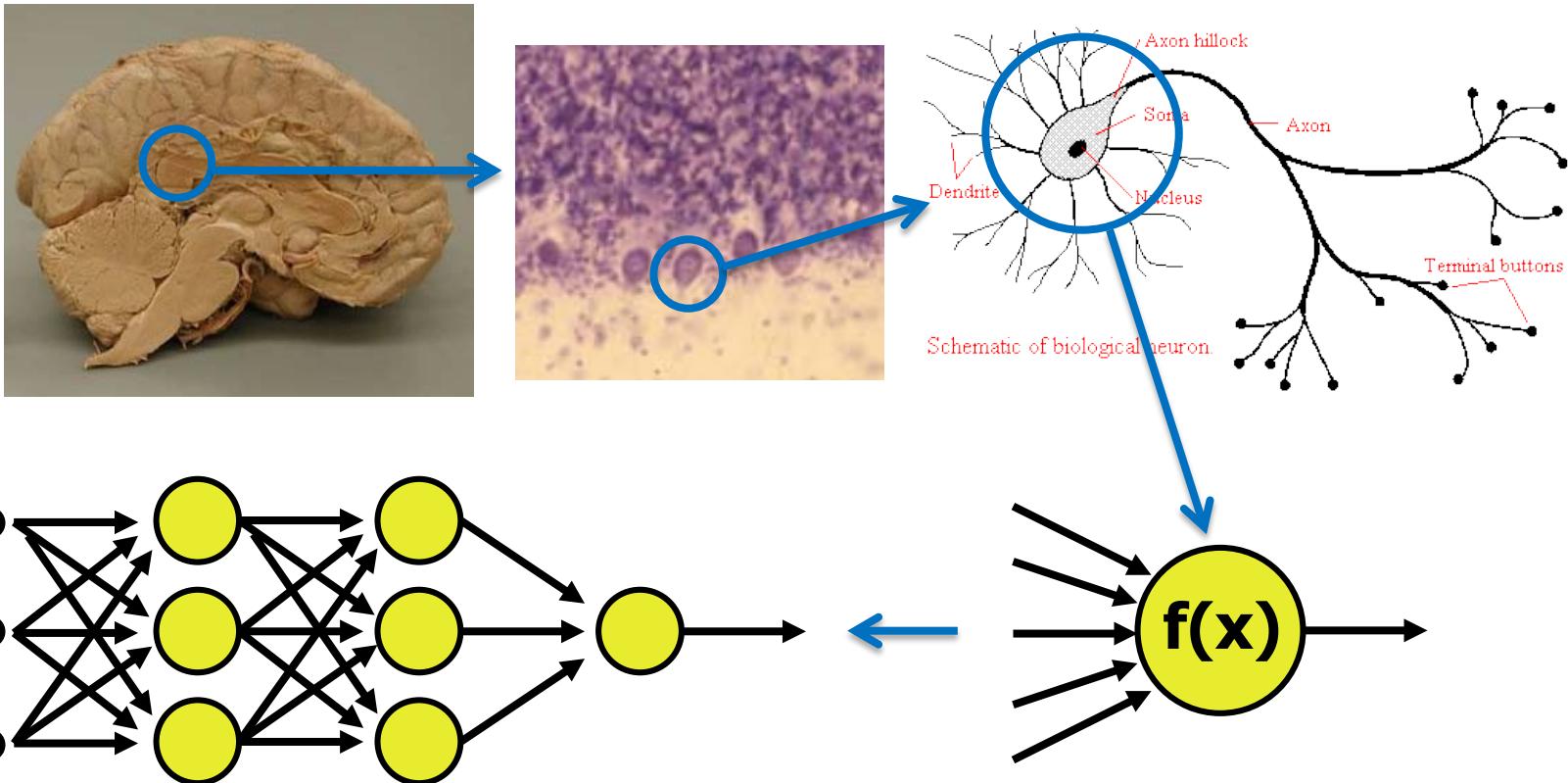
# Artificial neural networks (2)

- Large, densely interconnected networks of simple processing units



# Artificial neural networks (3)

- Inspired by the brain



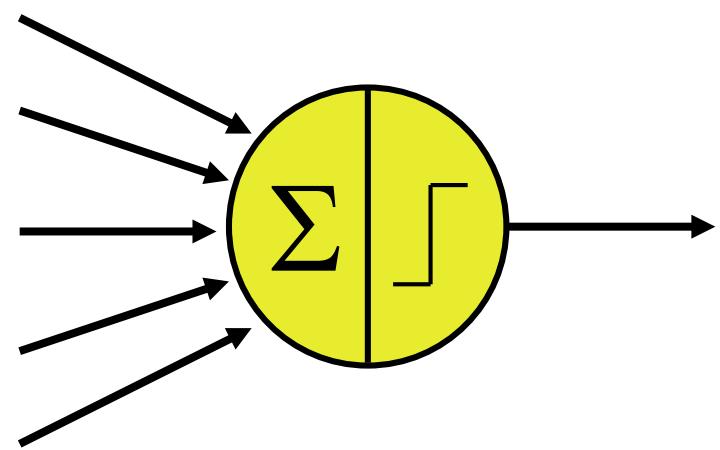
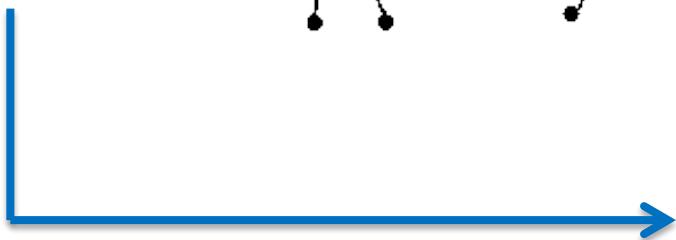
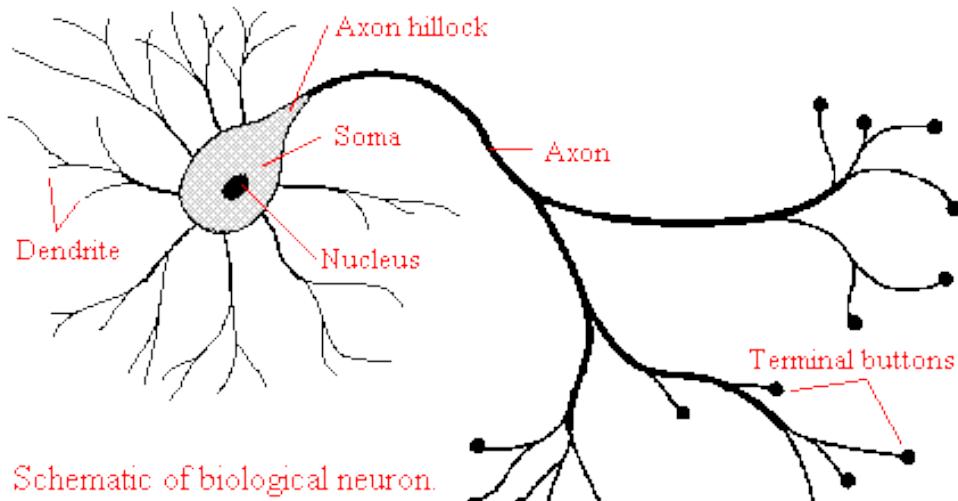
# Artificial neural networks (4)

- Research started in the 1950s
  - Took off after 1986 – big hype for about 10-15 years
    - brought together psychologists, neurologists, philosophers, machine learners, statisticians...
    - helped thinking about, among others, pattern recognition
    - resulted in a *lot* of grant money
  - From 2005/2009 – renewed interest
    - Extension to deep learning (deep nets)
    - Advances in hardware (GPUs) made it possible to learn these networks
    - Major steps in performance improvement (10%)
    - Development of several toolboxes Keras/Tensorflow/Theano/...
    - World attention, also from outside Machine Learning field
- made people realize they were doing pattern recognition  
Let PR researchers do stuff they never did before (speech recognition)

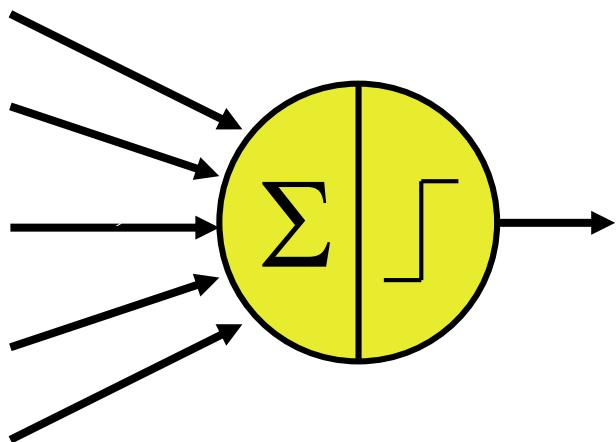
# History

- 1943 : McCulloch and Pitts: model of neuron
- **1958** : Rosenblatt: perceptron
- 1960s : Rosenblatt, Nilsson work on perceptrons
- 1968 : Minsky and Papert point out limitations:  
perceptrons are linear
- 1982 : Hopfield network (associative memory),  
Kohonen's self-organising map (clustering),  
Fukushima's Neocognitron (vision)
- **1986** : Rumelhart, Hinton and Williams:  
training of nonlinear networks
- 1997 : Hochreiter and Schmidhuber introduce Long Short-term memory (LSTM), recurrent neural net
- 2006 : Hinton showed effective training one-layer at a time
- 2009 : Nvidia involved in “big bang” of “deep learning”, 100x time improvement

# McCulloch-Pitts model (1943)



## McCulloch-Pitts model (2)



*weights inputs*  
output  $o_i = \phi\left(\sum_j w_{ij}x_j - b_i\right)$   
*threshold or bias*

$$\phi(a) = \begin{cases} 1 & a \geq 0 \\ 0 & a < 0 \end{cases}$$

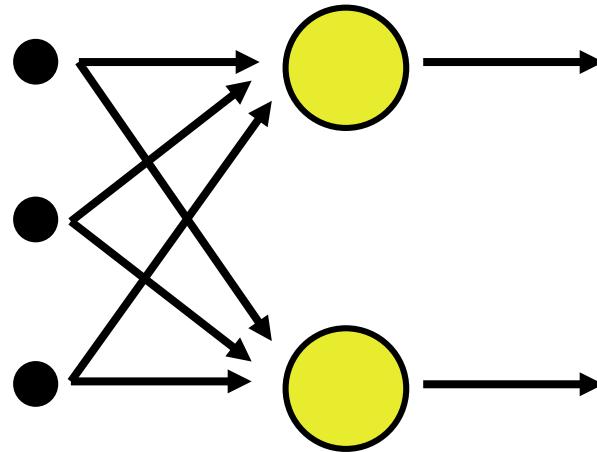
$$\phi(a) = \frac{1}{1 + \exp(-a)}$$

↙  
*transfer function*  
or  
*activation function*

“Fire” if total input exceeds a threshold

# Perceptron

- Networks of McCulloch-Pitts models can perform *universal computation*, given the right weights  $w$ : it can do anything a binary computer can do
- ...but how can we find the right weights  $w$  ?
- Rosenblatt (1958): possible for single layer networks, *perceptrons*

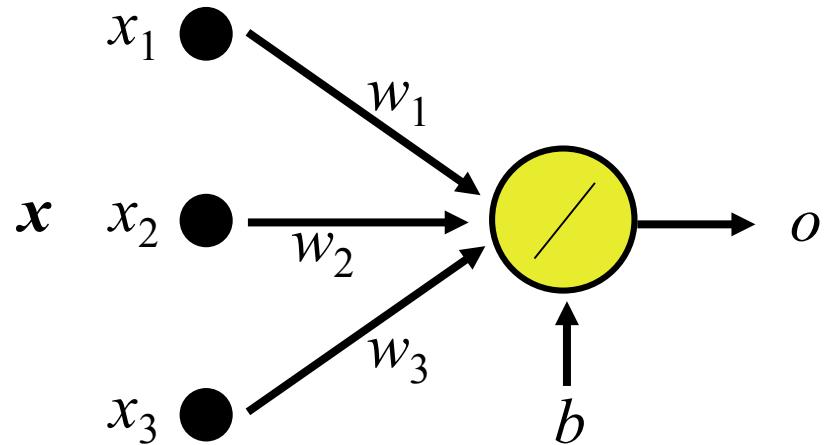


# Perceptron (2)

- Goal:

$$o(\mathbf{x}) = \phi(\mathbf{w}^T \mathbf{x} + b)$$

$$\begin{cases} > 0 & \mathbf{x} \in \omega_1 \\ < 0 & \mathbf{x} \in \omega_2 \end{cases}$$



- Trick #1: add bias as weight with constant input

$$\left. \begin{aligned} \mathbf{z} &= \begin{bmatrix} 1 \\ \mathbf{x} \end{bmatrix}, \mathbf{v} = \begin{bmatrix} b \\ \mathbf{w} \end{bmatrix} \\ \phi(a) &= a \end{aligned} \right\} \Rightarrow o(\mathbf{z}) = \mathbf{v}^T \mathbf{z}$$

# Perceptron (3)

- For classification, set targets  $q$  for every input vector  $z$ :

$$z \in \omega_1 : q = 1$$

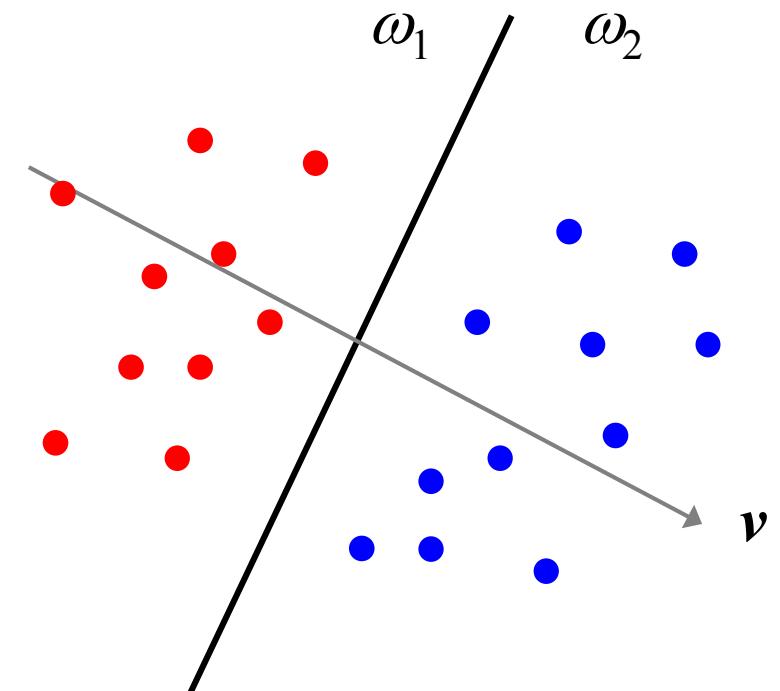
$$z \in \omega_2 : q = -1$$

- Trick #2: use targets to obtain single criterion

$$o(z) = \nu^T z \begin{cases} > 0 & z \in \omega_1 \\ < 0 & z \in \omega_2 \end{cases}$$

$$\Rightarrow \nu^T z \cdot q > 0$$

$$\Rightarrow \nu^T y > 0, \quad y = z \cdot q$$

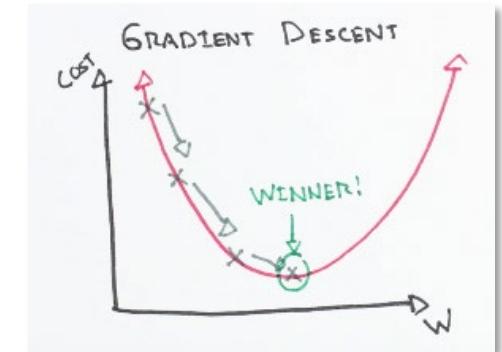


# Perceptron (4)

- Goal: zero misclassifications, i.e.  $\boldsymbol{v}^T \mathbf{y}_i > 0 \quad \forall i$

- Criterion to minimize:  $J(\boldsymbol{v}) = \sum_{\mathbf{y}_i \in \mathcal{Y}} (-\boldsymbol{v}^T \mathbf{y}_i)$

where  $\mathcal{Y}$  is the set of misclassified samples



- Can use gradient descent:  $\partial J(\boldsymbol{v}) / \partial \boldsymbol{v} = \sum_{\mathbf{y}_i \in \mathcal{Y}} (-\mathbf{y}_i)$

$$\boldsymbol{v}^{k+1} = \boldsymbol{v}^k - \rho \frac{J(\boldsymbol{v})}{d\boldsymbol{v}} = \begin{cases} \boldsymbol{v}^k + \rho \sum_{\mathbf{y}_i \in \mathcal{Y}} \mathbf{y}_i & \text{batch update} \\ \boldsymbol{v}^k + \rho \mathbf{y}_i, \quad \mathbf{y}_i \in \mathcal{Y} & \text{single update} \end{cases}$$

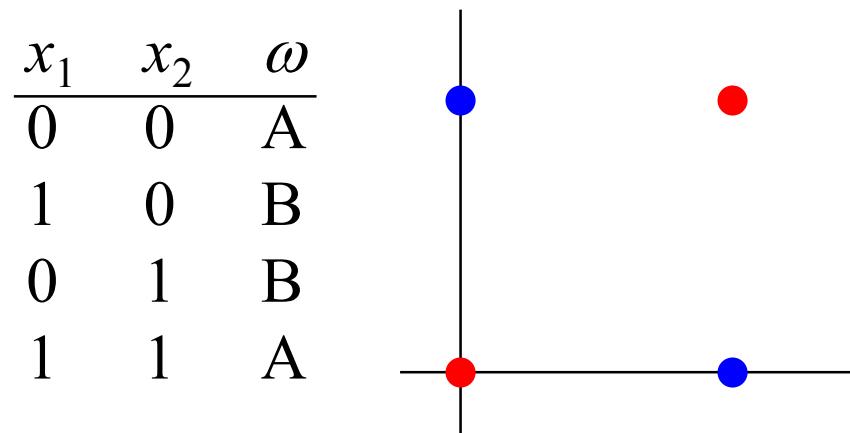
Criterion is somewhat arbitrary, could also count misclassifications

# Perceptron (7)

- Perceptron is a trainable two-class linear discriminant (extendable to multiple classes)
- Training algorithm can be proven to converge to correct solution for separable classes
- When classes are not linearly separable:
  - indefinite training, weights will blow up
  - solution: decrease  $\rho$  during training,  $\rho(k)$ , or early stopping

# Perceptron (8)

- Minsky & Papert (1969): perceptrons are limited



*The XOR problem cannot be solved by a linear discriminant such as the perceptron*

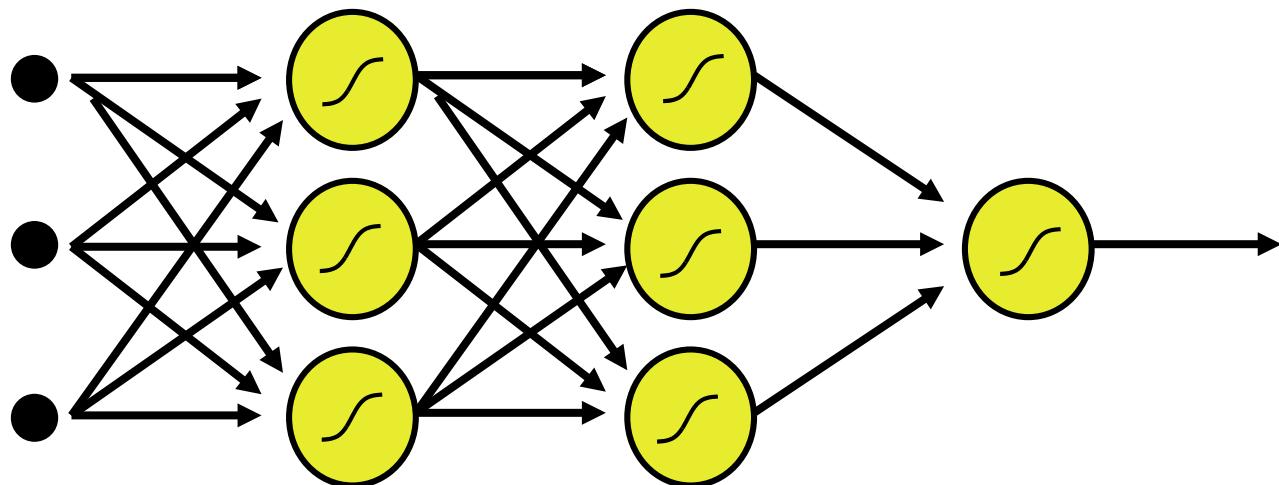
- When classes are nonlinearly separable:
  - nonlinear transfer functions
  - multilayer perceptron – but how to find weights...?
  - Rumelhart et al. (1986): use the chain rule!

*This did in fact take twenty years...*

# Multilayer perceptron (MLP)

- Stacked perceptrons: *feedforward networks*
- Each unit has a nonlinear *transfer function*,

e.g. the sigmoid or logistic function  $\phi(a) = \frac{1}{1 + \exp(-a)}$



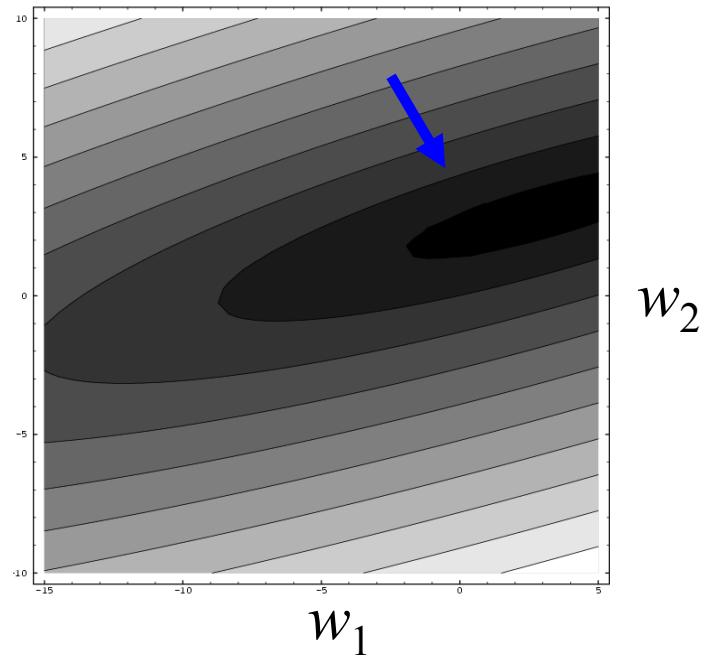
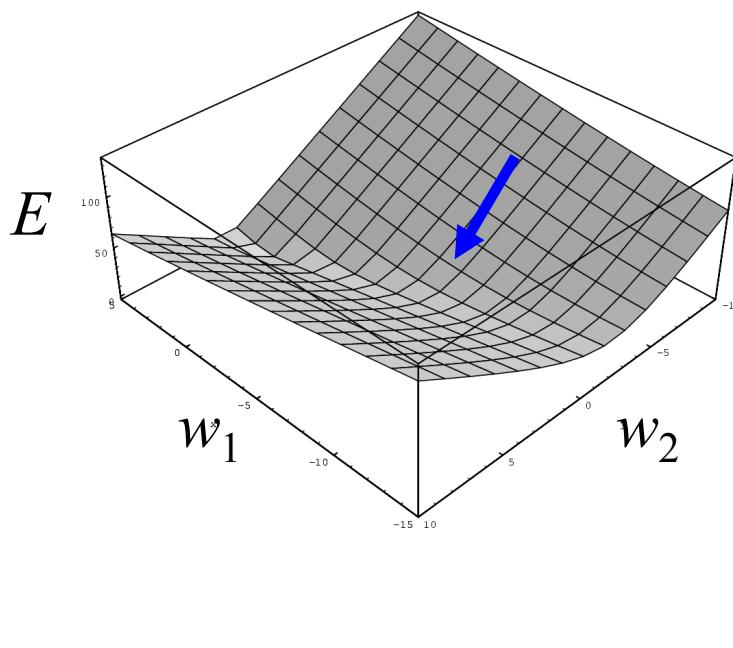
# Backpropagation training

- Method to distribute weight updates through the network
- Criterion: error  $E$ , difference between network output and targets (mean square error between output and target  $\sum(e_i - o_i)^2$ )
- Initialize weights  $w$  to small random values
- While not converged, e.g. while  $|E^{old} - E|/E > E_{thr} = 10^{-6}$ , or while error on test set decreases:
  - select a training sample  $x_i$
  - for each weight  $w$ 
    - calculate  $\partial E / \partial w$
    - set  $w' = w - \rho \partial E / \partial w$   
(with  $\rho$  a learning rate, e.g. 0.01)
    - or use a momentum term,  
 $w' = w - \rho \partial E / \partial w - \alpha [\partial E / \partial w]^{prev}$

$\alpha >> \rho$ : keep moving in previous direction  
 $\rho >> \alpha$ : adapt to new direction

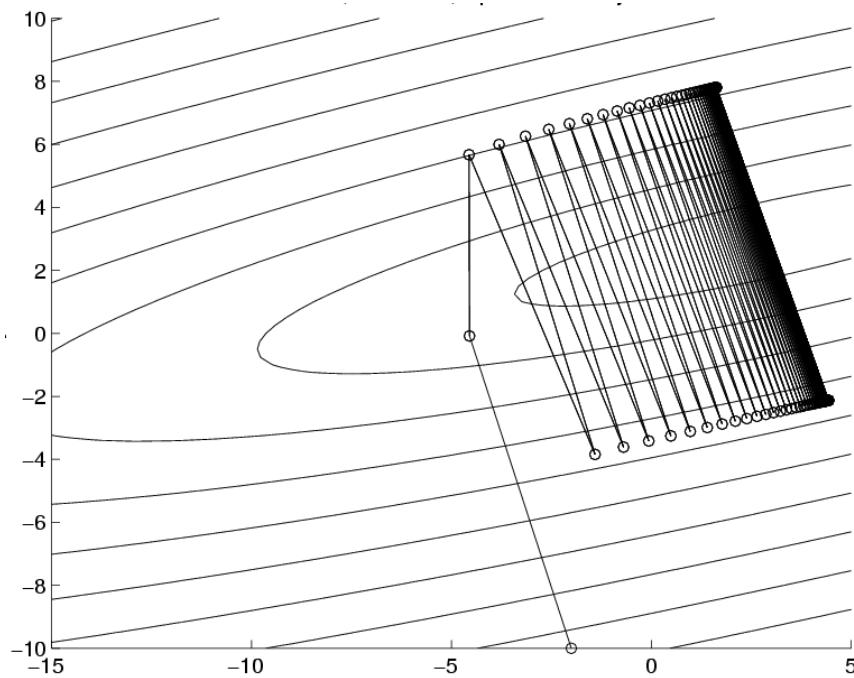
# Backpropagation training (8)

- Example: two weights

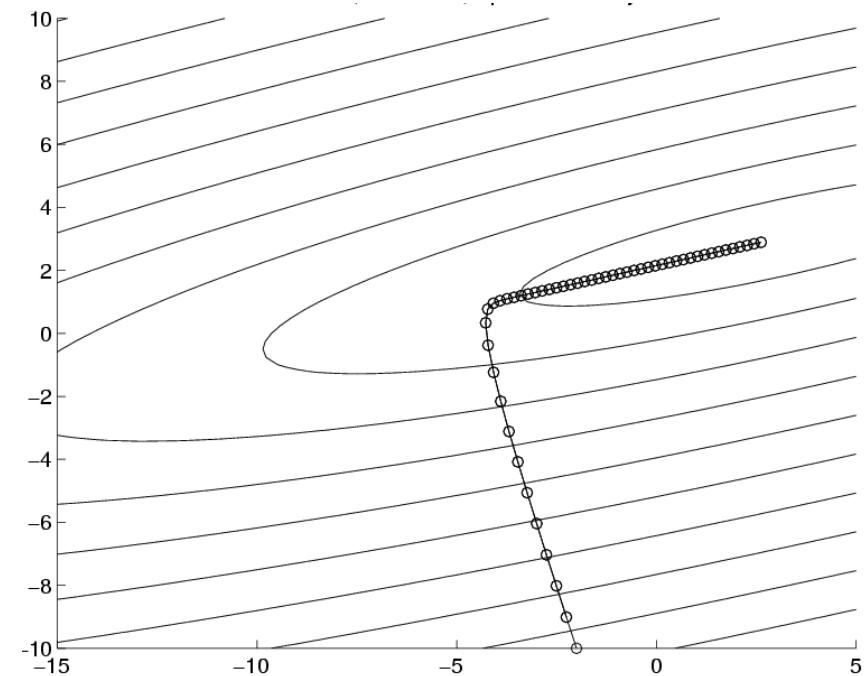


# Backpropagation training (9)

- Learning rate controls oscillation and speed



$\rho = 1$ : >100 iterations

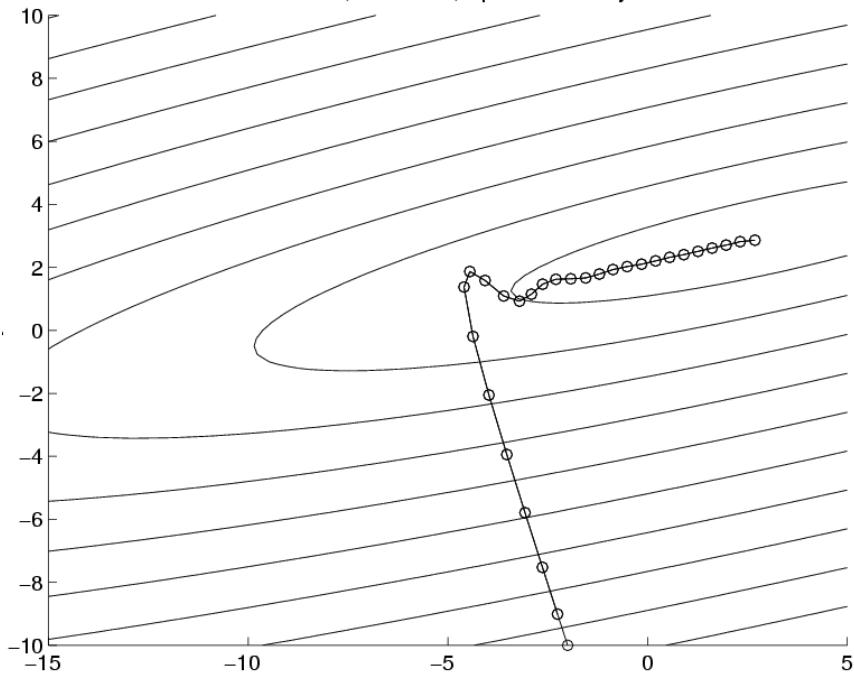


$\rho = 0.1$ : 52 iterations

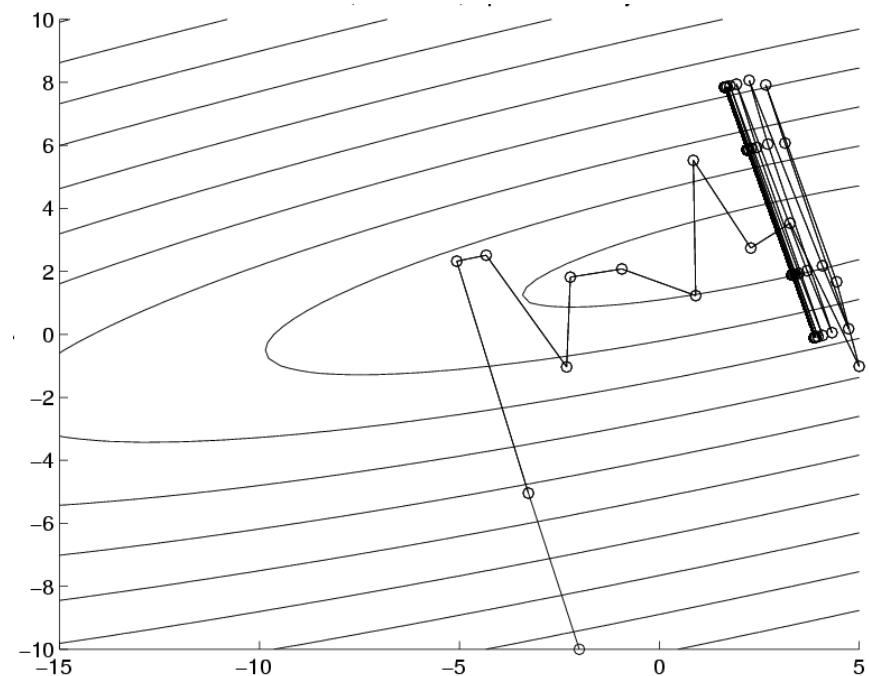
*In practice, not easy  
(imagine doing this for thousands of weights)*

# Backpropagation training (10)

- Momentum uses a bit of the previous step



$\rho = 0.1, \alpha = 0.5$ : 29 iterations

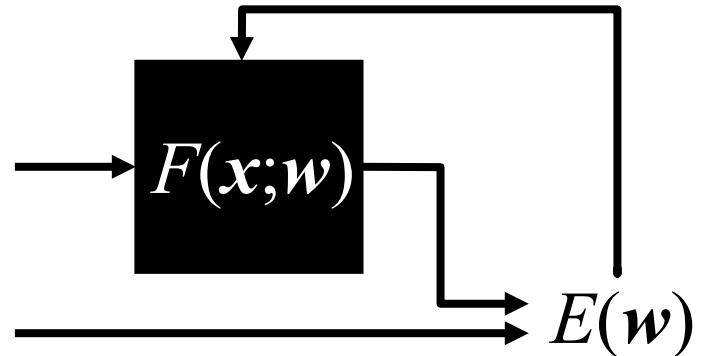


$\rho = 0.5, \alpha = 0.5$ : >100 iterations

*Right: learning rate too large , so oscillations start occurring again ...  
Also option to make learning rate dependent on time :  $\rho(t)$*

# Other training algorithms

- Backpropagation training is simple gradient descent, but implemented in a useful way: all updates can be calculated locally (in parallel)
- Other view: simply optimize MSE  $E$  w.r.t. weight vector  $w$  using any optimization routine, e.g.
  - second order (Newton, pseudo-Newton)
  - conjugate gradient descent
  - Broyden-Fletcher-Goldfarb-Shanno (BFGS)
  - Levenberg-Marquardt (LM, in **PRTtools**)



# Multilayer perceptrons (2)

- Choices:
  - targets  $(0/1, 0.1/0.9, 0.2/0.8) t$
  - **number of hidden layers**
  - **number of units per hidden layer**  $n_i$
  - transfer functions  $\phi(a)$
  - initialisation  $w^{(0)}$
  - training algorithm
  - parameters (learning rate  $\rho$  etc.)
  - convergence decision  $E_{thr}$  or test set selection
  - ...
- All of these influence results!

*“Training ANNs is more of an **ART** than a science”*

# Multilayer perceptrons (3)

- Number of weights = number of parameters =  $\sum_{l=1}^{o-1} (n_l + 1)n_{l+1}$   
e.g. for  $p = 10$ ,  $C = 2$ , 2 20-unit hidden layers:  
 $(10 + 1) \cdot 20 + (20 + 1) \cdot 20 + (20 + 1) \cdot 2 = 682$  parameters  

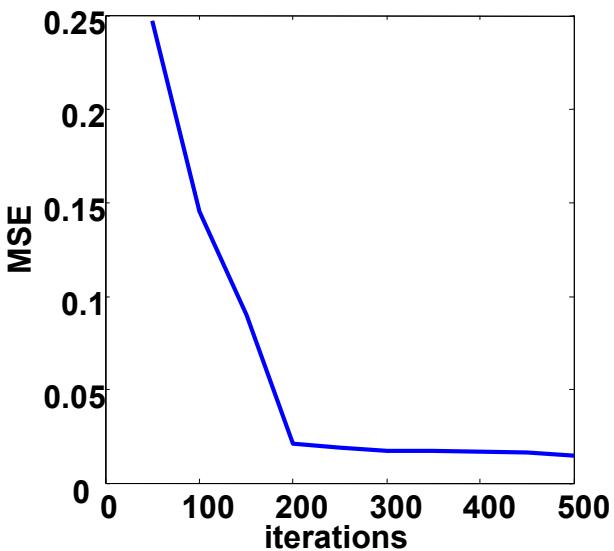
*Per node: #parents+bias node ( $n_l + 1$ )*
- Danger of overtraining!
- Prevention:
  - use small networks
  - regularize: minimize  $E(\mathbf{w}) + \lambda \|\mathbf{w}\|$
  - small  $w$ 's: low complexity, training slowly increases  $w$ 's;  
so when stopping in time: automatic regularization!
- Regularization is a form of complexity control (discussed later)

# Multilayer perceptrons (4)

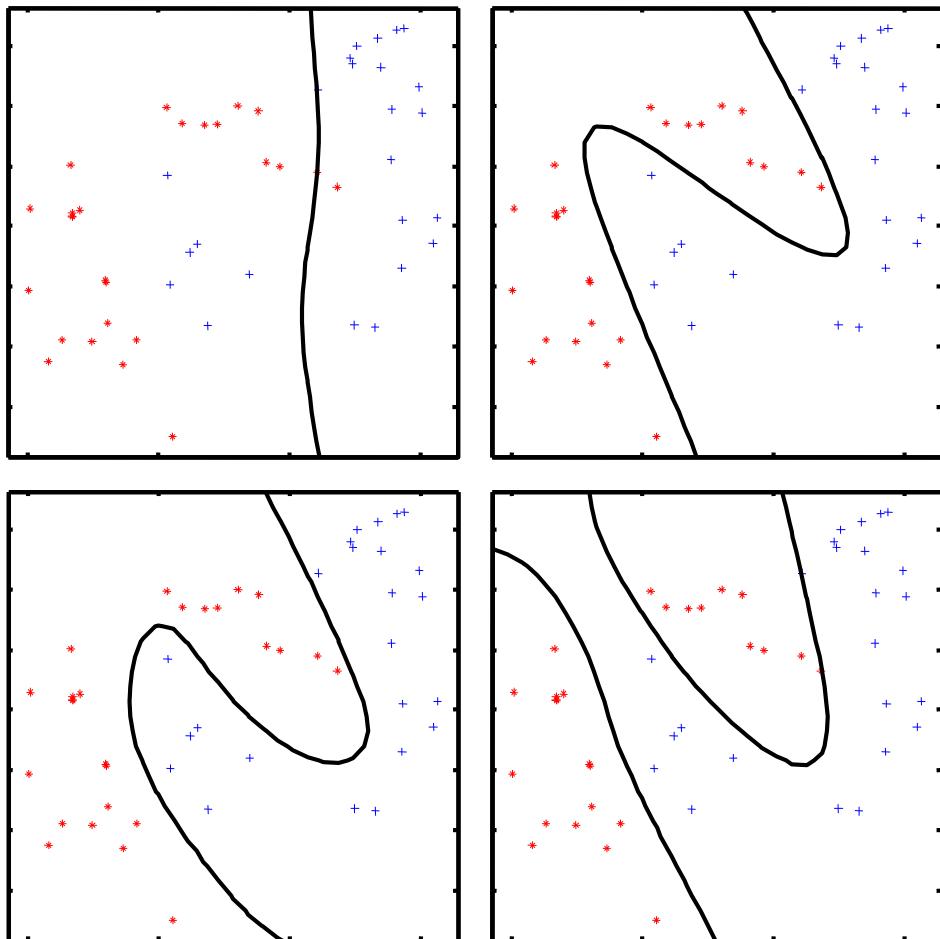
*Initialization still important*

- Examples:

1 hidden layer  
of 3 units,  
2 initialisations



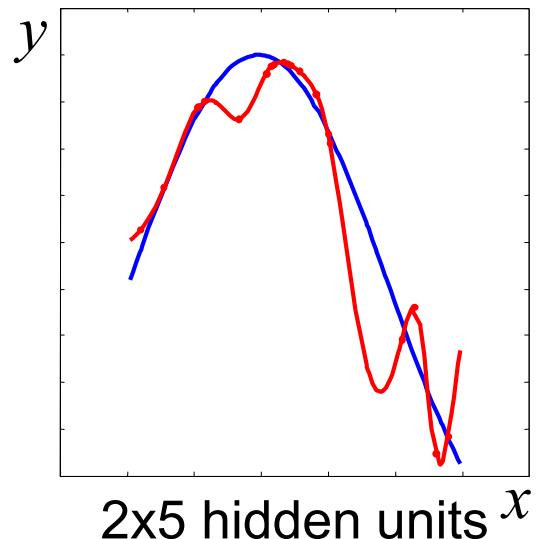
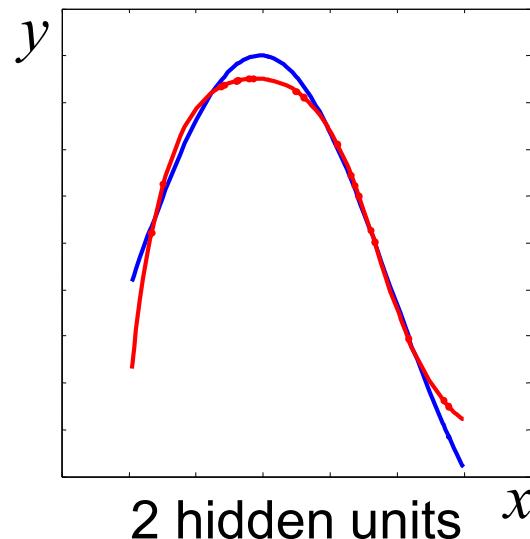
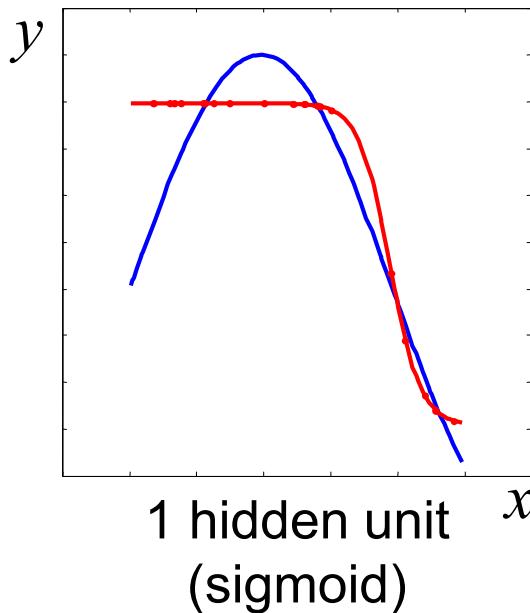
2 hidden layers  
of 5 units each,  
2 initialisations



*Next to making ANN simpler, another solution is to train for ever, because one can still make small improvements ....(maybe not with gradient descent ...) This is also one of the success of deep learning : more optimization power*

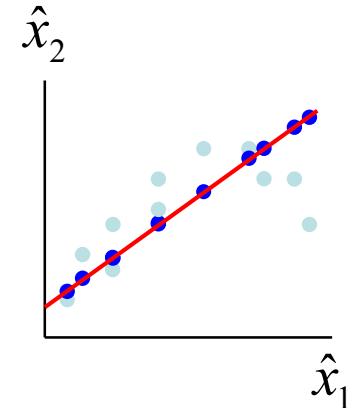
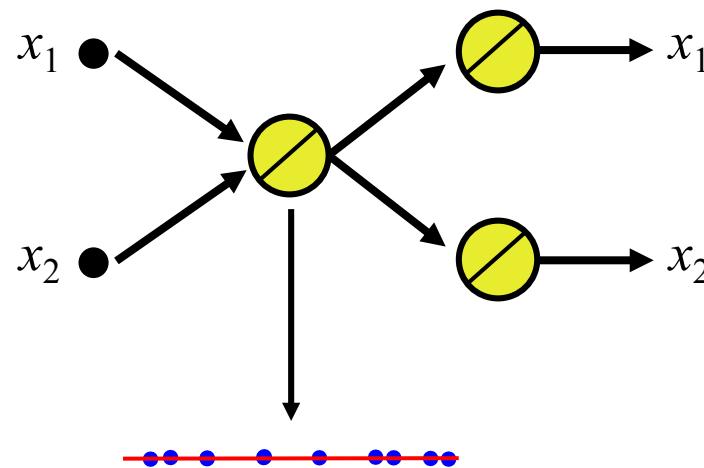
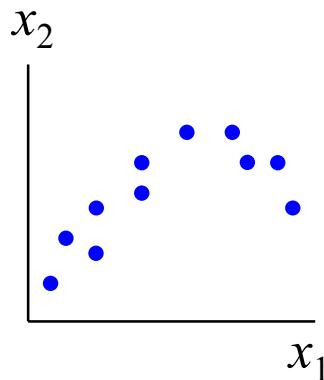
# ANNs for regression

- Feedforward ANNs are *universal approximators*
  - Classification: input  $x$ , targets  $y = 0/1, 0.1/0.9$
  - Regression: input  $x$ , output  $y$
- Examples:



# Autoregressive ANNs / Autoencoder

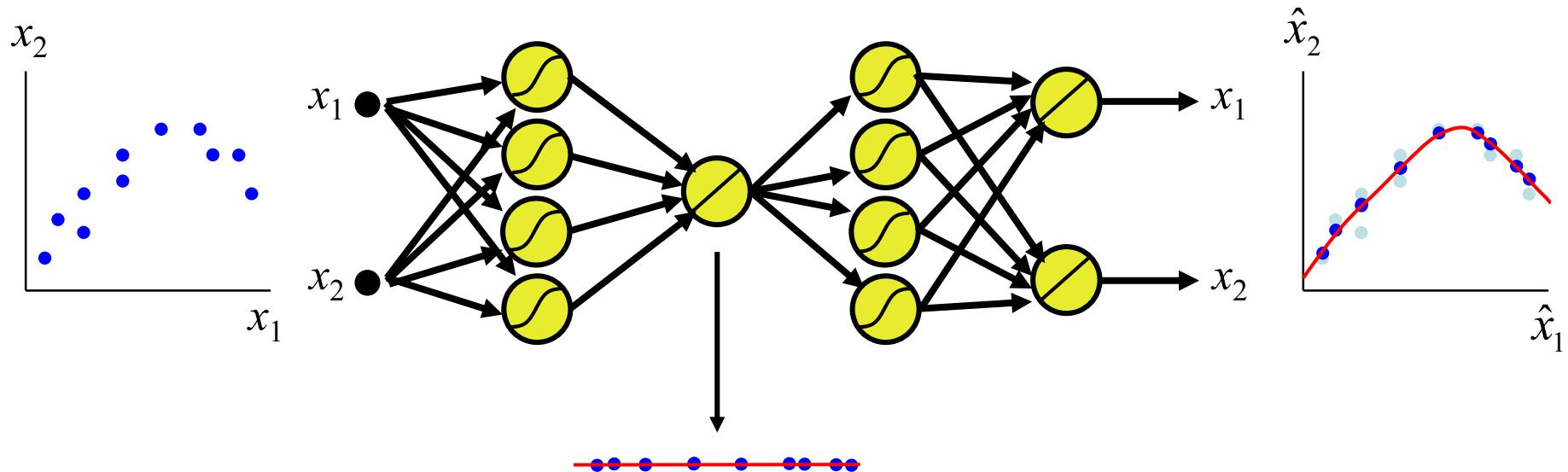
- Feedforward ANNs that predict their input
- Bottleneck layer: feature extraction



*If linear (as in this example) : then we are performing PCA !!!*

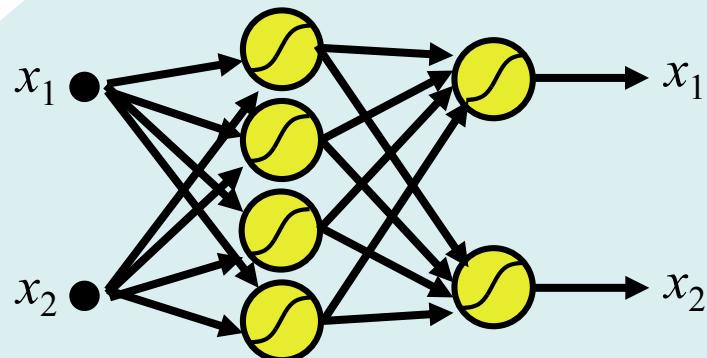
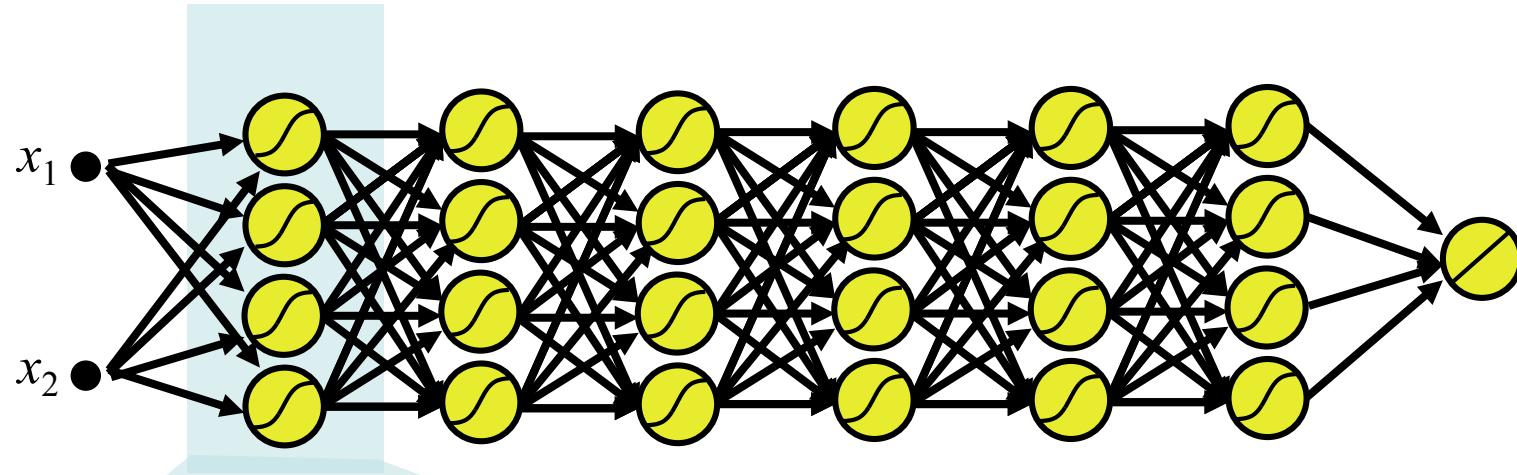
# Autoregressive ANNs / Autoencoder (2)

- With multiple hidden layers:  
nonlinear feature extraction



# Deep learning

## Many hidden layers, learn by auto-encoding

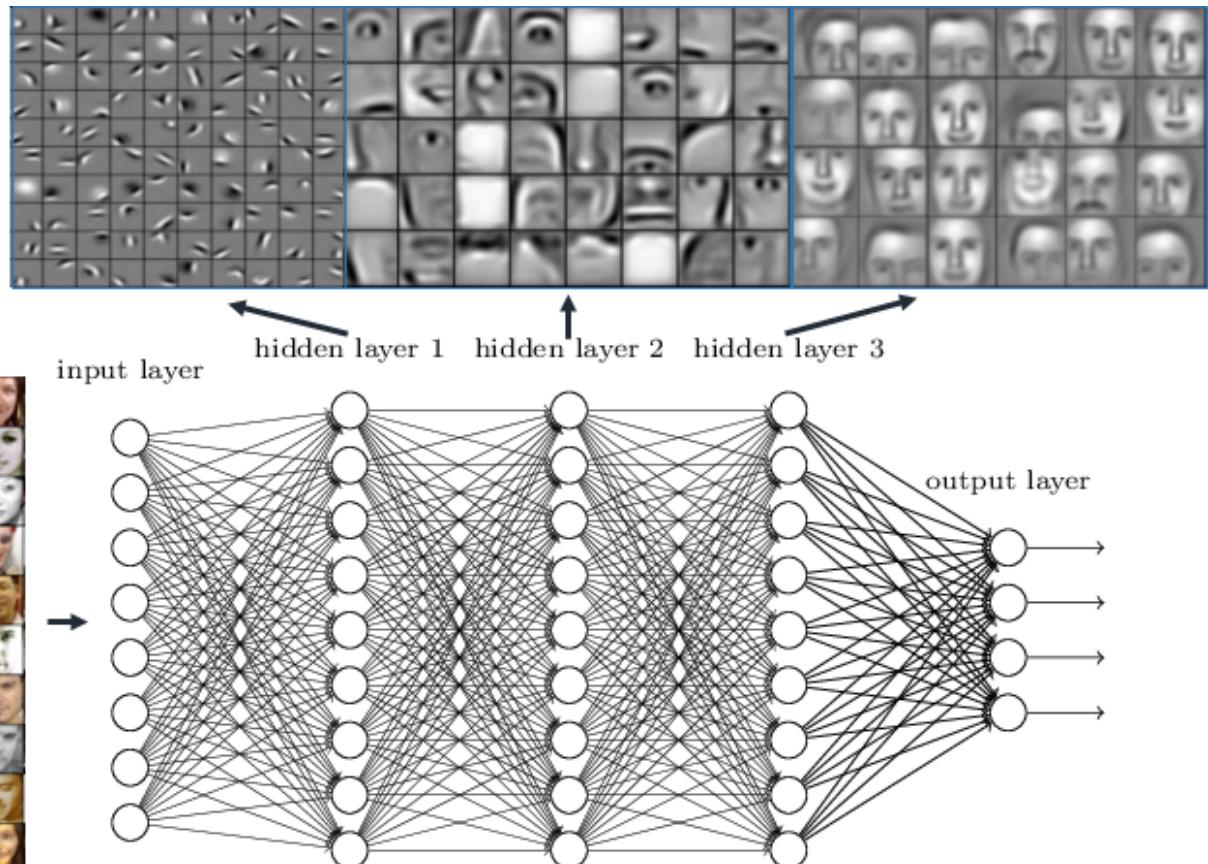


NOW not necessary anymore to learn by autoencoders  
With GPUs you can use Backpropagation again (fast enough)

# Deep learning

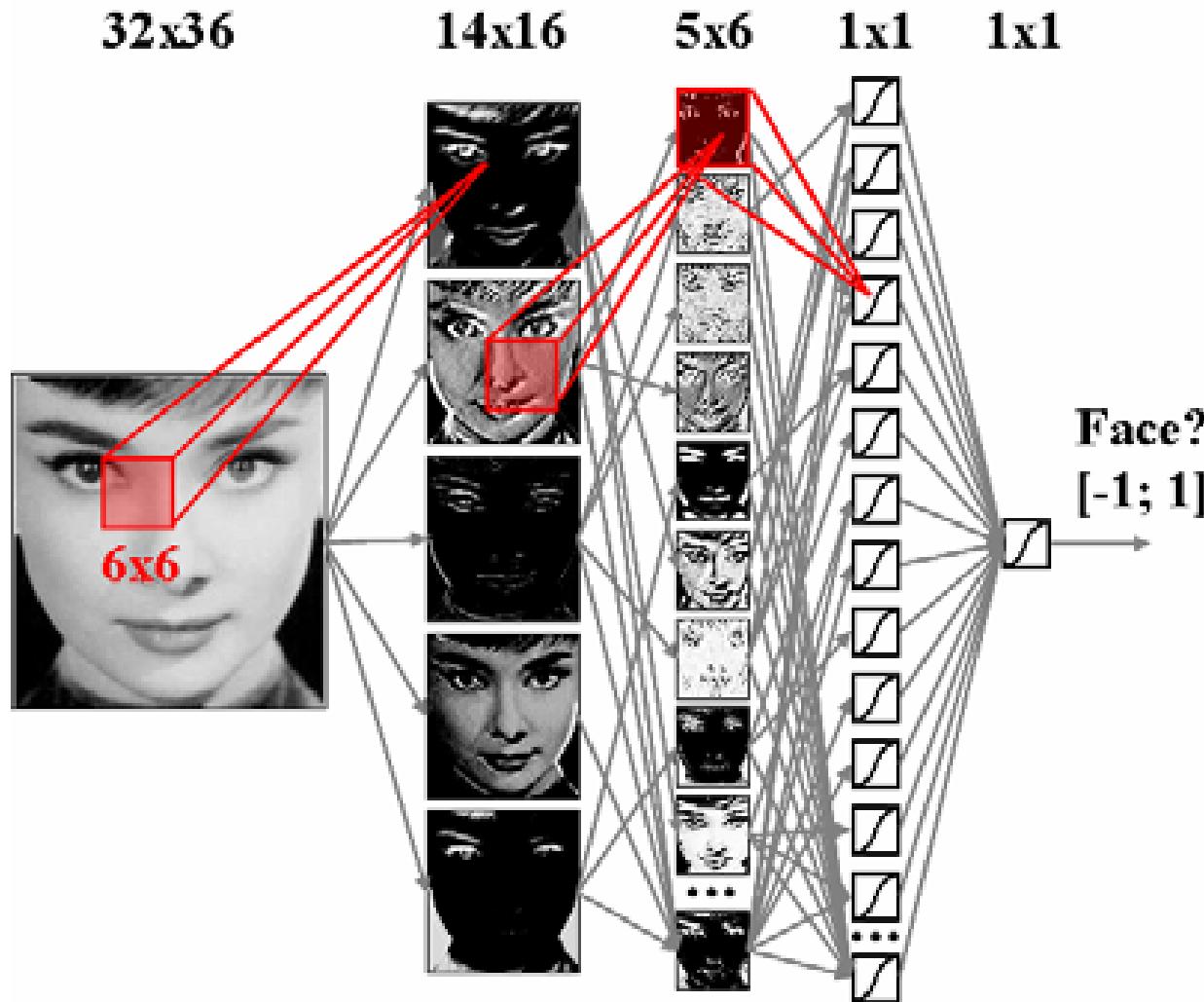
## Learning features

Deep neural networks learn hierarchical feature representations



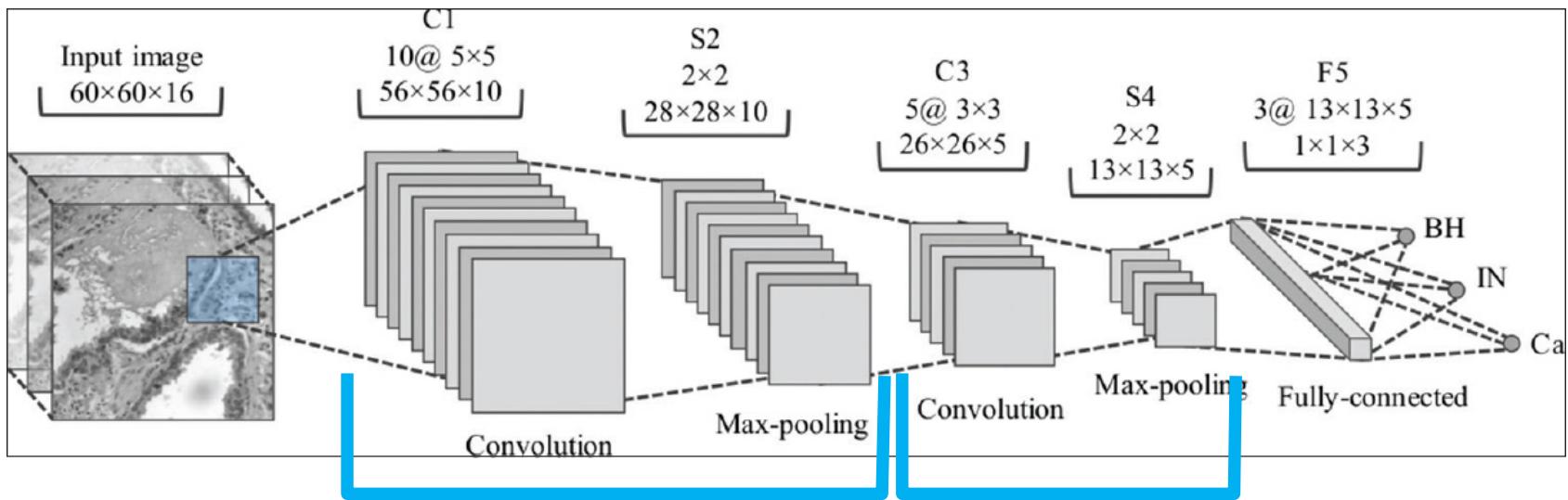
# Deep learning

## Convolutional Neural Networks (1)



# Deep learning

## Convolutional Neural Networks (2)



- amount of layers
- use of pre-trained networks (on another problem)

12	20	30	0
8	12	2	0
34	70	37	4
112	100	25	12

$\xrightarrow{2 \times 2 \text{ Max-Pool}}$

20	30
112	37

# Deep learning

## Convolutional Neural Networks (3)

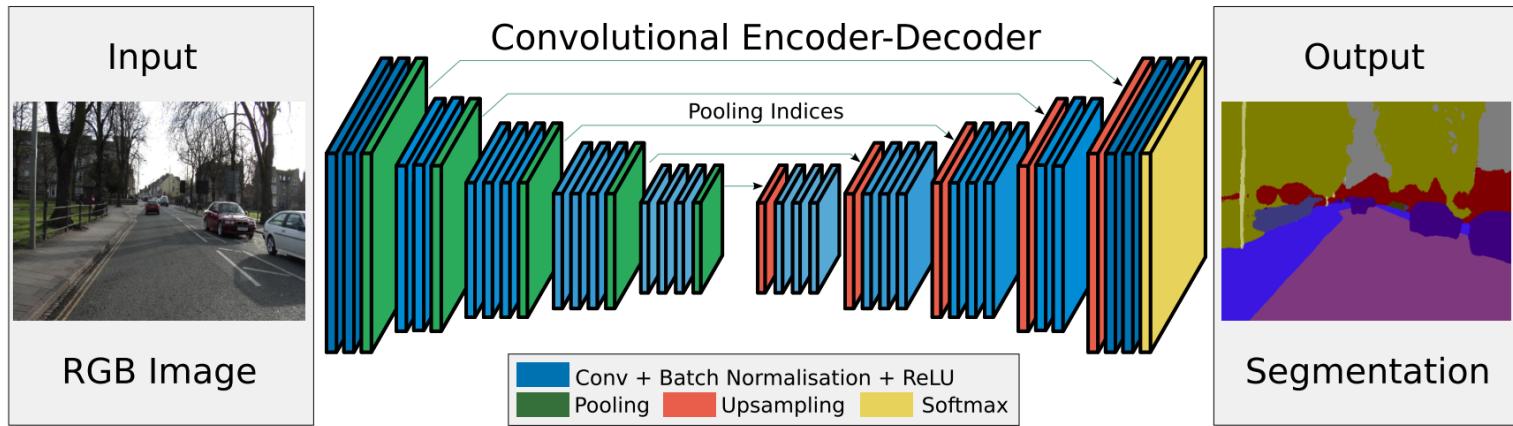
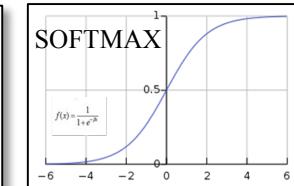
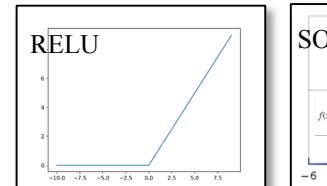
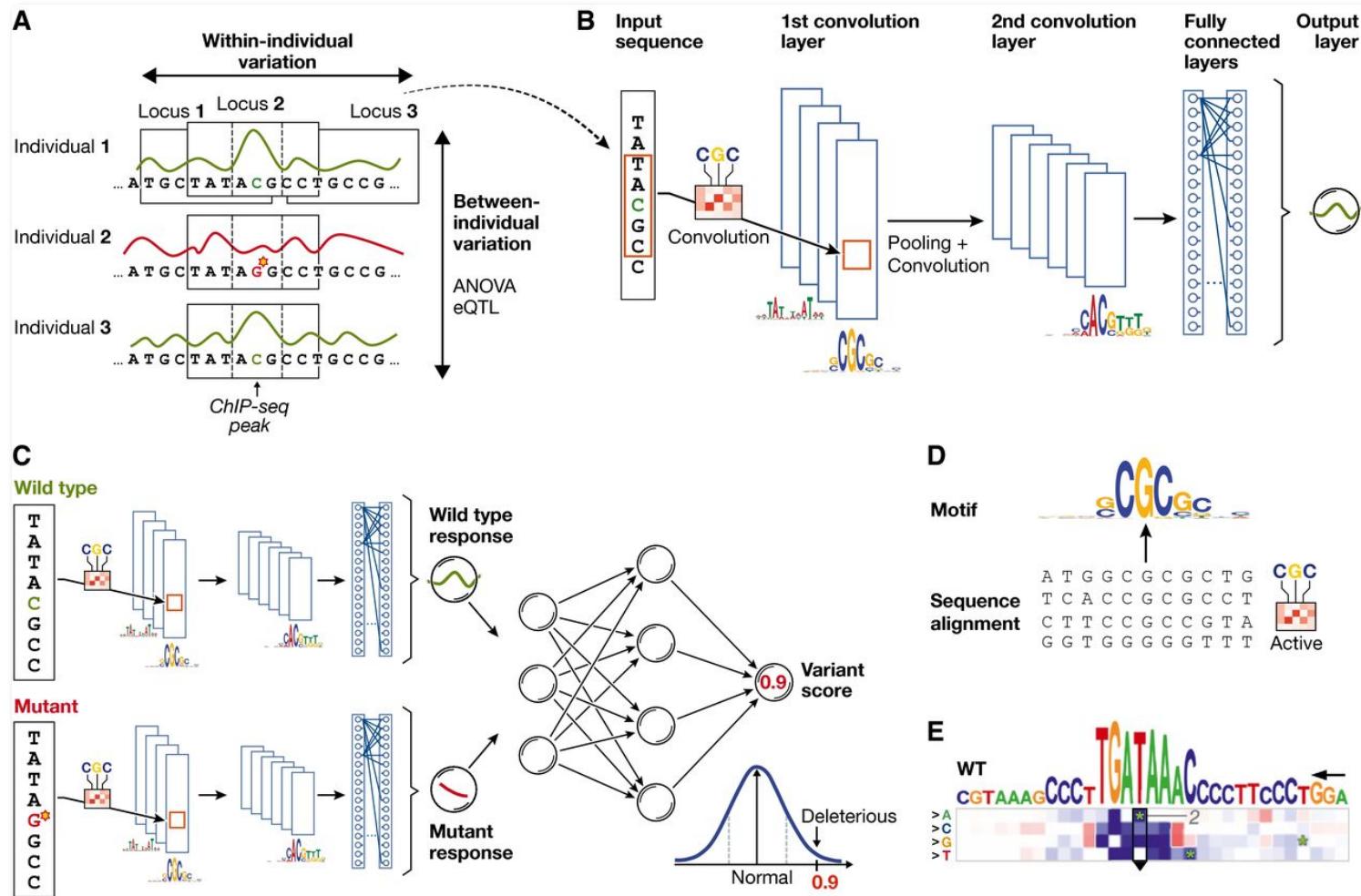


Fig. 2. An illustration of the SegNet architecture. There are no fully connected layers and hence it is only convolutional. A decoder upsamples its input using the transferred pool indices from its encoder to produce a sparse feature map(s). It then performs convolution with a trainable filter bank to densify the feature map. The final decoder output feature maps are fed to a soft-max classifier for pixel-wise classification.

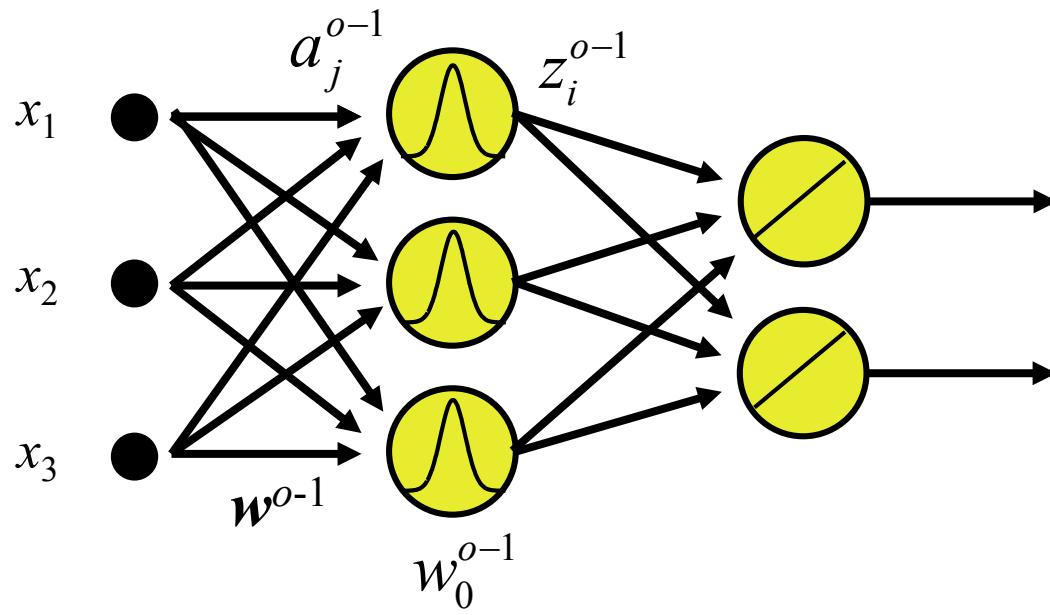


# Deep learning Convolutional Neural Networks (4)



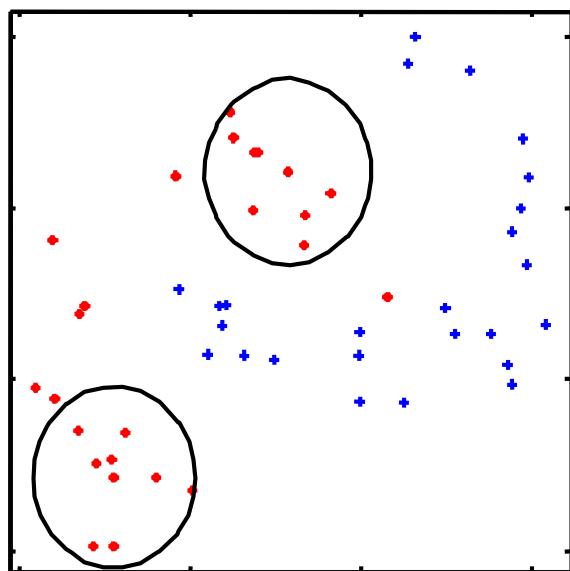
# Radial basis function ANNs

- Feed-forward ANNs with
  - Squared distance activation functions  $a_j^{o-1} = \|x - w^{o-1}\|^2$
  - Gaussian transfer functions  $z_j^{o-1} = N(\mu = a_j^{o-1}, \sigma^2 = w_0^{o-1})$

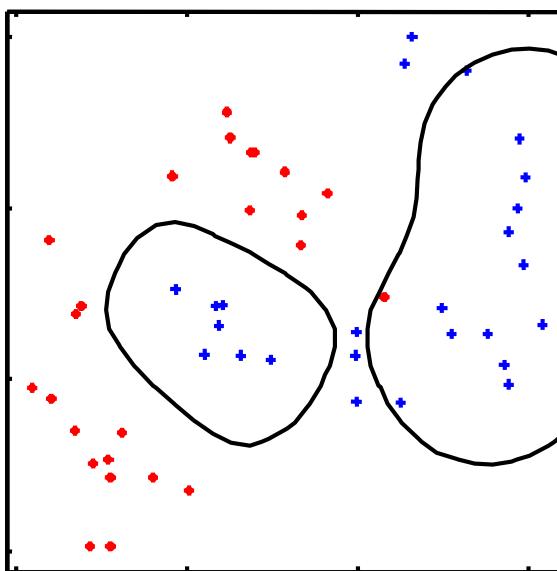


# Radial basis function ANNs (3)

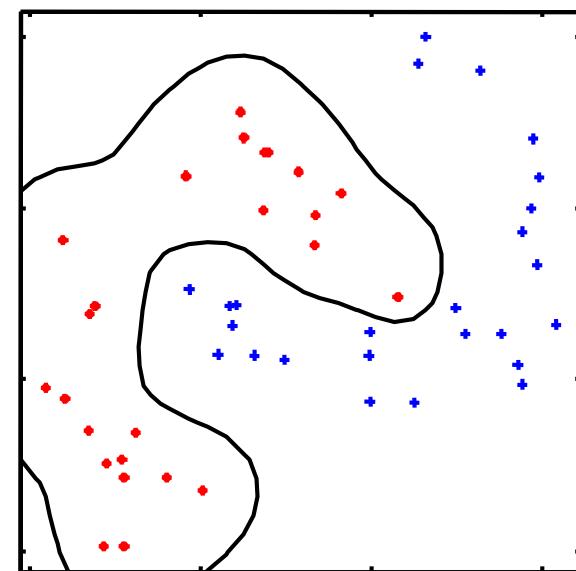
- Example: classification



2 hidden units



5 hidden units



10 hidden units

# Other types of ANN

- Large number of feedforward variants
  - cascading correlation (self-constructing)
  - Neocognitron (for vision)
  - time-delay (for speech and image analysis)
  - ...
- Self-organising maps and GTMs:
  - feature extraction, clustering
- Hopfield networks:
  - associative memories, optimisation
- Boltzmann machines, Bayesian networks:
  - conditional probability models

# Recapitulation

- *Perceptrons* are “neuron-inspired” linear discriminants
- *Multilayer perceptrons* and *radial basis function* feedforward ANNs are trainable, nonlinear discriminants
- Feed-forward ANNs in general can be used for classification, regression and feature extraction
- There is a large body of alternative ANNs
- Key problems in the application of ANNs are choosing the right *architecture* and good *training parameters*



10 min break

# Support vector classifiers

# Vapnik

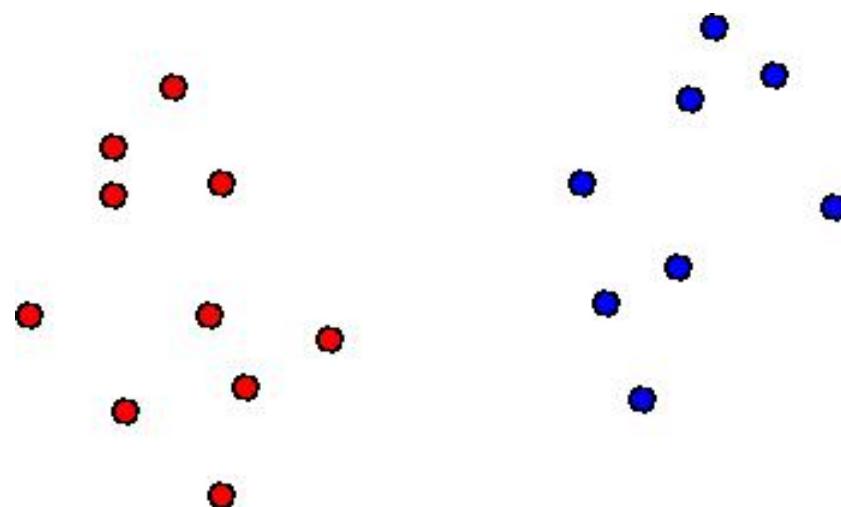
- Performed foundational work in pattern recognition with Chervonenkis in Russia from the 1960s
- Motto:

When you have limited training data,  
and you want to solve a classification problem,  
avoid solving a more complicated intermediate problem
- Translation to classification:

when you want to find a discriminant, avoid estimating densities

# Maximum margin classifier

- Simple problem: 2 linearly separable classes
  - What is a good linear classifier?
  - What is the best linear classifier?

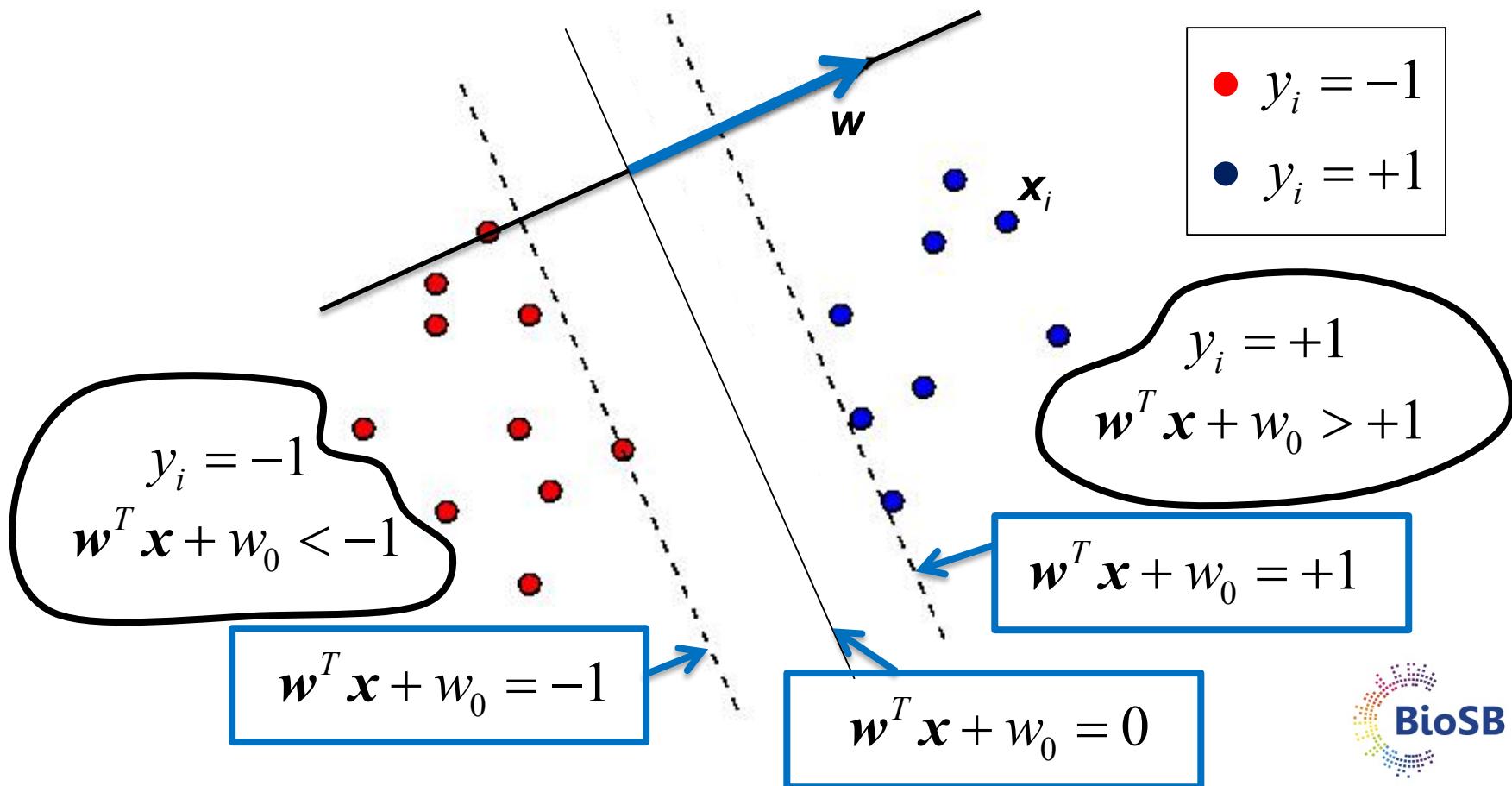


●	$y_i = -1$
●	$y_i = 1$

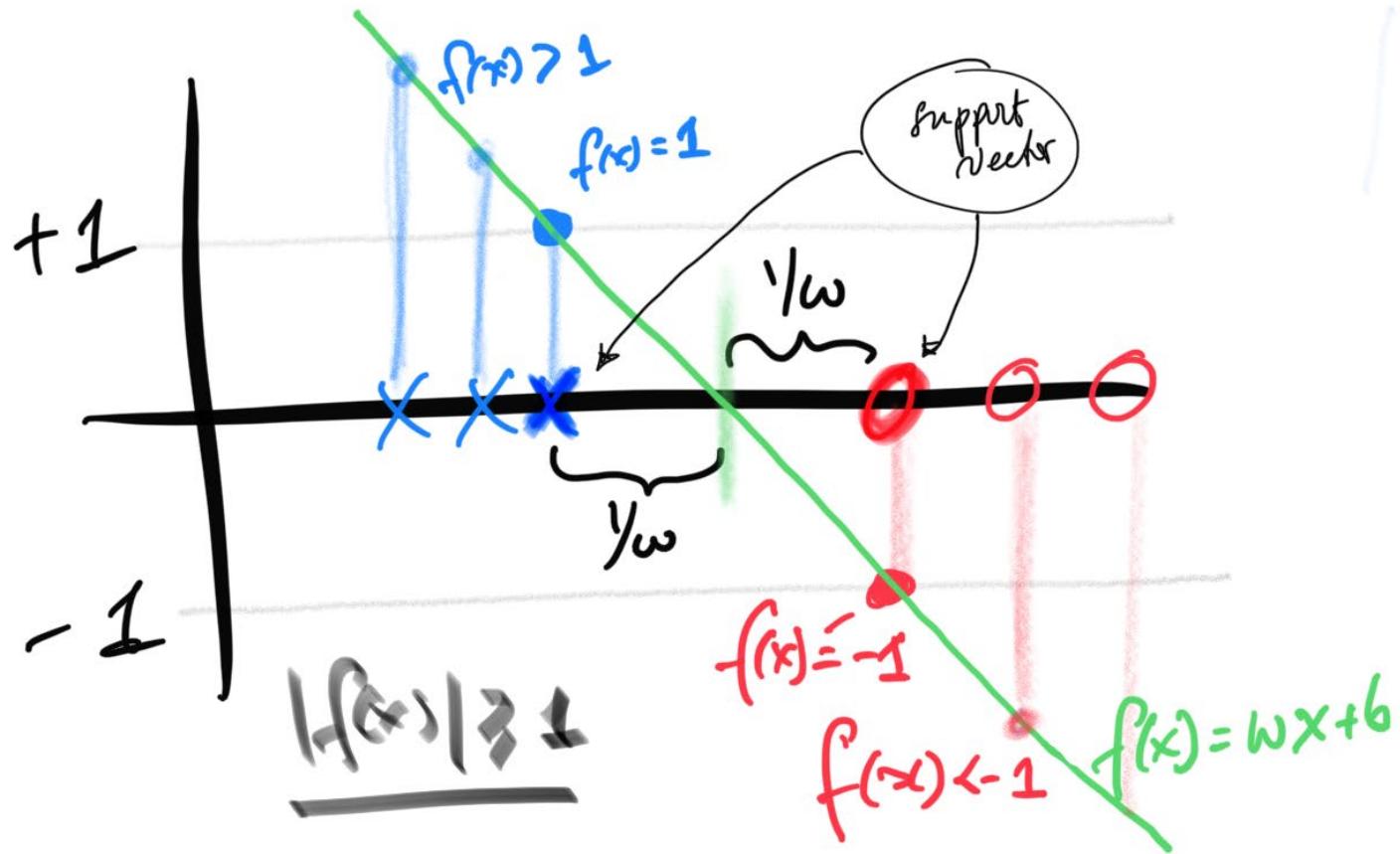
# Maximum margin classifier (2)

- Canonical hyperplane:

any plane of the form  $f(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + w_0$   
for which  $\min_i |f(\mathbf{x}_i)| = 1$



# Maximum margin for 1D data



# Maximum margin classifier (3)

- The distance between an object  $x_i$  and the hyperplane is

$$d(x_i, \text{decision boundary}) = \frac{\mathbf{w}^T \mathbf{x}_i + w_0}{\|\mathbf{w}\|}$$

- The maximum margin classifier is a canonical hyperplane s.t. the distance between the object closest to the hyperplane on one side,

$$\arg \min_i (\mathbf{w}^T \mathbf{x}_i + w_0) \quad | \quad y_i = +1$$

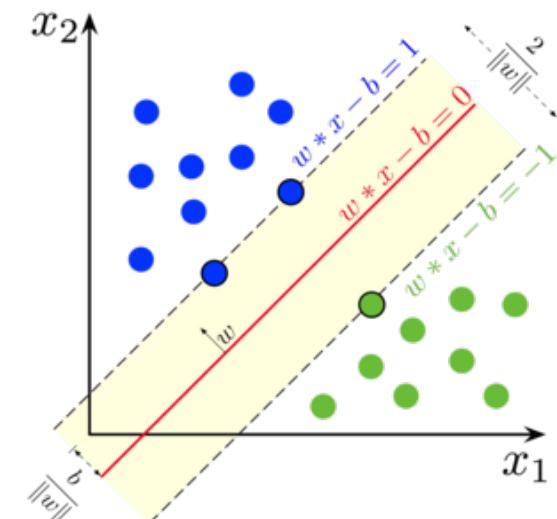
and the object closest on the other side,

$$\arg \max_i (\mathbf{w}^T \mathbf{x}_i + w_0) \quad | \quad y_i = -1$$

is maximal

- This distance is called the margin:  $\rho = \frac{2}{\|\mathbf{w}\|}$

(it will become apparent later why this is a Good Thing)



# Support vector classifier

- Maximizing the margin  $\rho = \frac{2}{\|\mathbf{w}\|}$

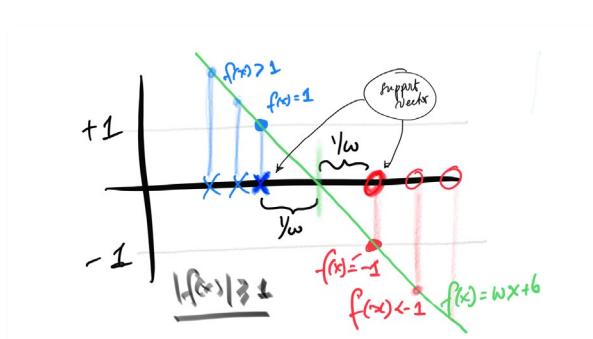
under the constraint that all training samples are classified correctly, leads to the optimization problem:

$$\min \frac{1}{2} \|\mathbf{w}\|^2 \text{ such that}$$

$$\mathbf{w}^T \mathbf{x}_i + w_0 \leq -1 \mid y_i = -1$$

$$\mathbf{w}^T \mathbf{x}_i + w_0 \geq +1 \mid y_i = +1$$

- The constraints can be written as  $y_i(\mathbf{w}^T \mathbf{x}_i + w_0) > 1$
- This is called the *support vector classifier*, or *support vector machine* (SVM)



# Support vector classifier (2)

- It is possible to incorporate the constraints into the optimization itself, using Lagrange multipliers (basic calculus):

$$\max_{\alpha} \min_{\mathbf{w}, w_0} \frac{1}{2} \|\mathbf{w}\|^2 - \sum_{i=1}^n \alpha_i (y_i (\mathbf{w}^T \mathbf{x}_i + w_0) - 1)$$

with  $\alpha_i > 0 \quad \forall i$

- Each constraint corresponds to a single object  $\mathbf{x}_i$
- Each constraint has a Lagrange multiplier  $\alpha_i$
- So each object corresponds to a Lagrange multiplier

# Support vector classifier (3)

- To solve the optimization, take the derivative and set to 0
  - Differentiate with respect to  $\mathbf{w}, w_0$  :

$$\sum_{i=1}^n \alpha_i y_i = 0 \quad (w_0)$$

$$\mathbf{w} = \sum_{i=1}^n \alpha_i y_i \mathbf{x}_i \quad (\mathbf{w})$$

- Re-substituting gives:

$$\max \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j$$

$$\text{with } \alpha_i > 0 \quad \forall i \quad \text{and} \quad \sum_{i=1}^n \alpha_i y_i = 0$$

Max over  $\alpha$ , derivatives wrt  $\alpha$

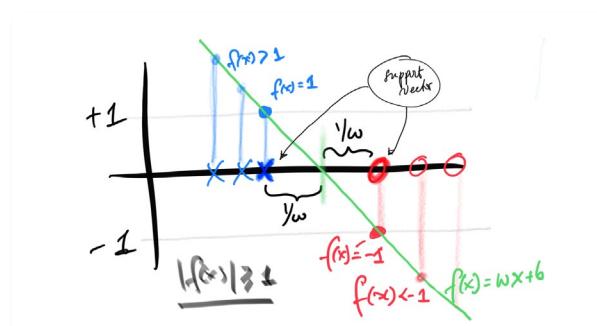
$$\max_{\alpha} \min_{\mathbf{w}, w_0} \frac{1}{2} \|\mathbf{w}\|^2 - \sum_{i=1}^n \alpha_i (y_i (\mathbf{w}^T \mathbf{x}_i + w_0) - 1), \quad \alpha_i > 0$$

# Support vectors

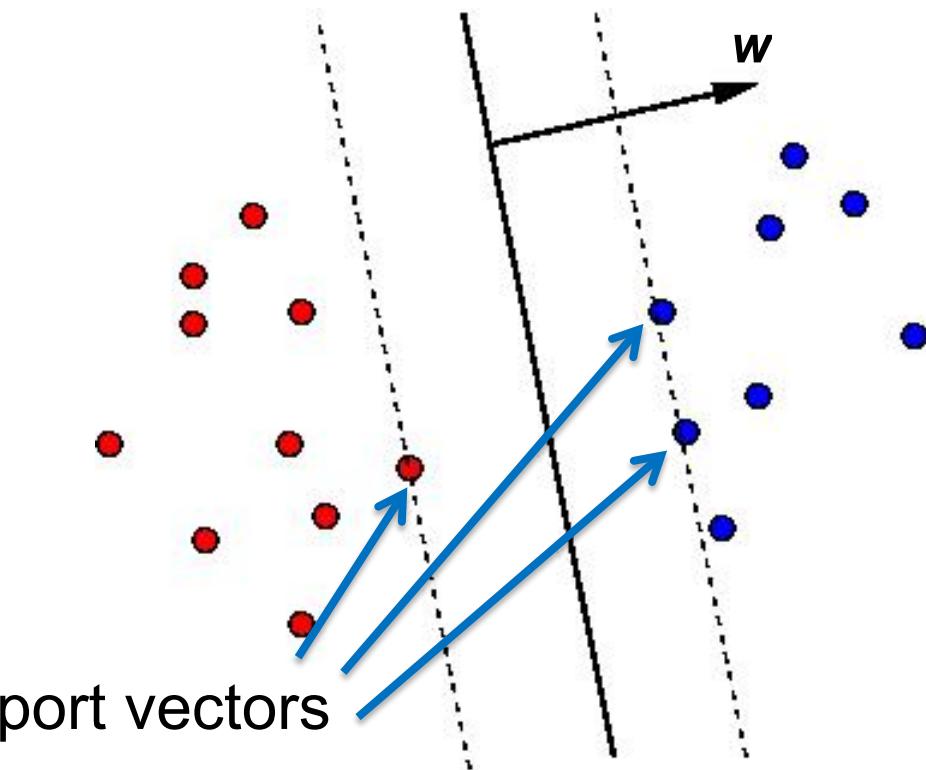
- The classifier is a linear combination of objects:

$$\mathbf{w} = \sum_{i=1}^n \alpha_i y_i \mathbf{x}_i$$

- Many Lagrange multipliers become equal to 0, so in fact the classifier is a *sparse* linear combination of objects
- Objects for which the Lagrange multiplier  $> 0$  are called *support vectors*



# Support vectors (2)

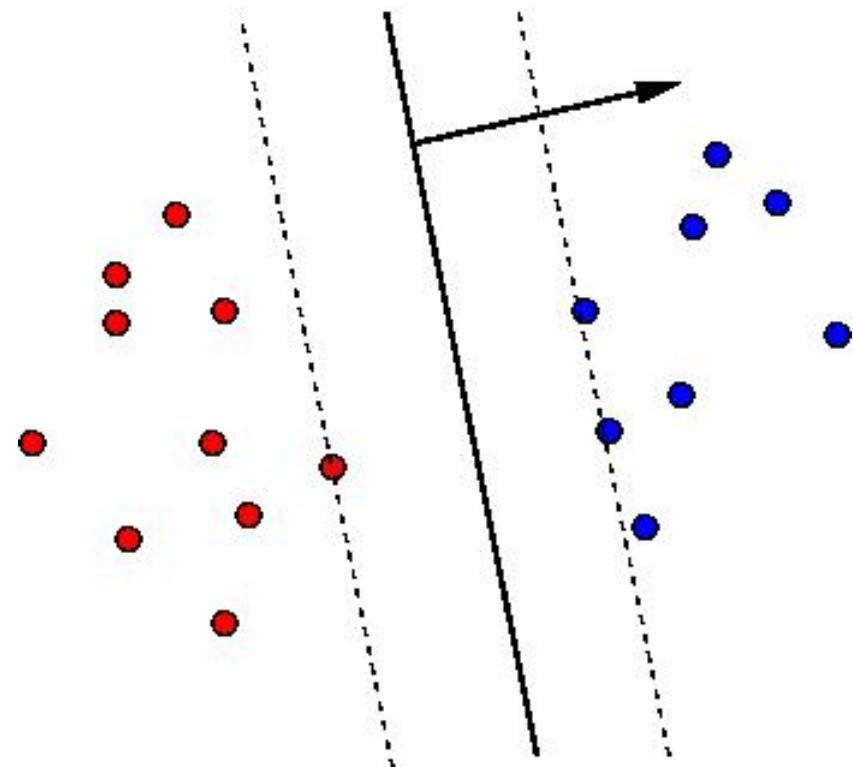


support vectors

# Support vectors (3)

- If non-support vectors are left out and training is repeated, the resulting classifier is identical
- The number of support vectors gives a bound on the *leave-one-out error estimate*:

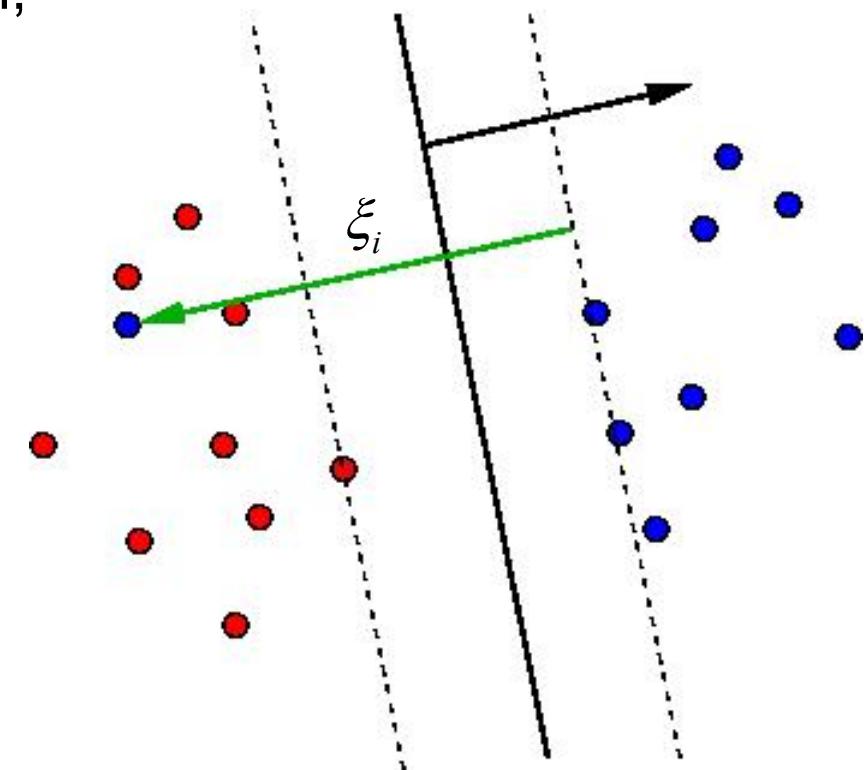
$$\hat{e}_{loo} \leq \frac{\# \text{ support vectors}}{n}$$



# Class overlap

- When there is overlap between the classes, the canonical hyperplane is not defined
- To be able to still find a solution, apply a trick:  
soften the constraints  
that each object is on  
the correct side of the  
decision boundary
- For the blue object on the  
incorrect side of the boundary:

$$y_i(\mathbf{w}^T \mathbf{x}_i + w_0) \geq 1 - \xi_i$$



- The variable  $\xi_i$  is called a *slack variable*

# Class overlap (2)

- In the ideal (non-overlapping) case, all slack variables are 0
- To force slack variables to be small,  
we add them to the margin to be minimized:

$$\min \frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{i=1}^n \xi_i \text{ such that}$$

$$\mathbf{w}^T \mathbf{x}_i + w_0 \leq -(1 - \xi_i) \mid y_i = -1$$

$$\mathbf{w}^T \mathbf{x}_i + w_0 \geq +(1 - \xi_i) \mid y_i = +1$$

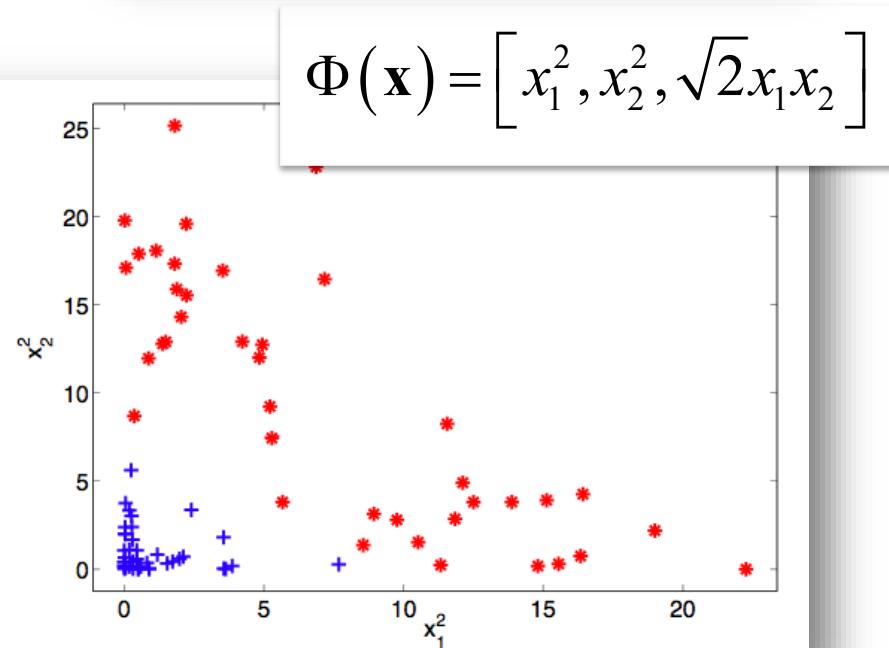
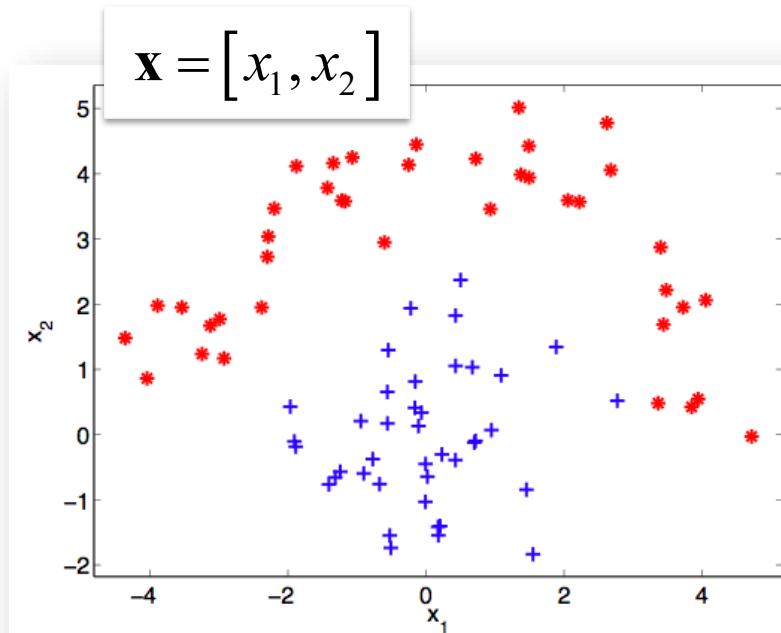
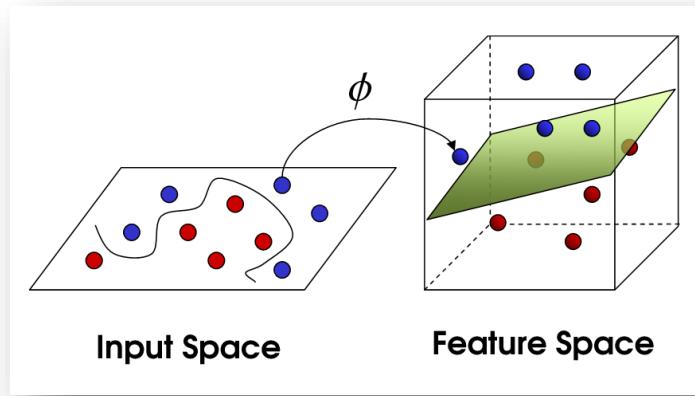
- We can rewrite that in almost the same way we did before:

$$\max \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j$$

$$\text{with } 0 \leq \alpha_i \leq C \quad \forall i \quad \text{and} \quad \sum_{i=1}^n \alpha_i y_i = 0$$

# The kernel trick

- Function  $\Phi$  maps data into a space in which classification may be easier

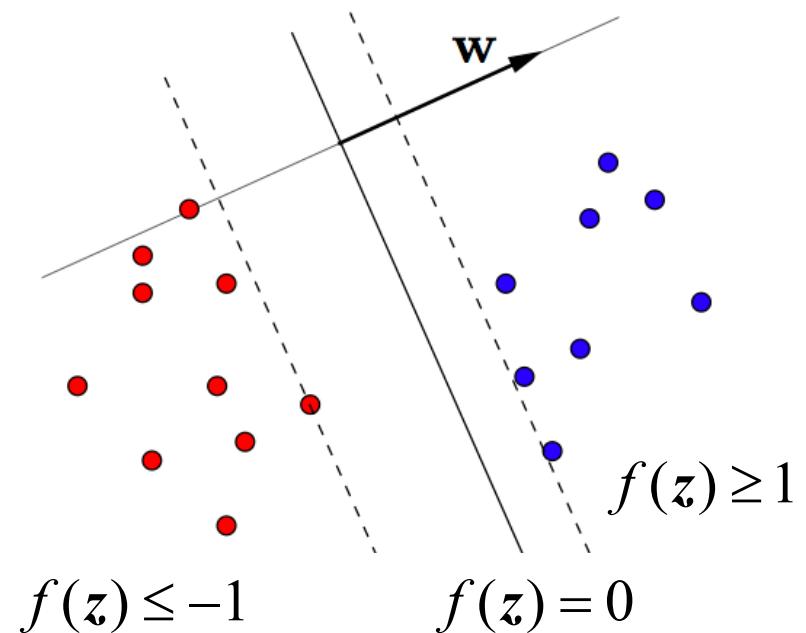


# The kernel trick (2)

- Classifier:

$$f(\mathbf{z}) = \mathbf{w}^T \mathbf{z} + w_0$$

$$= \sum_{i=1}^n \alpha_i y_i \mathbf{x}_i^T \mathbf{z} + w_0$$



- Optimization problem:

$$\max_{\alpha} \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n y_i y_j \alpha_i \alpha_j \mathbf{x}_i^T \mathbf{x}_j$$

$$\alpha_i \geq 0, \quad \forall i$$

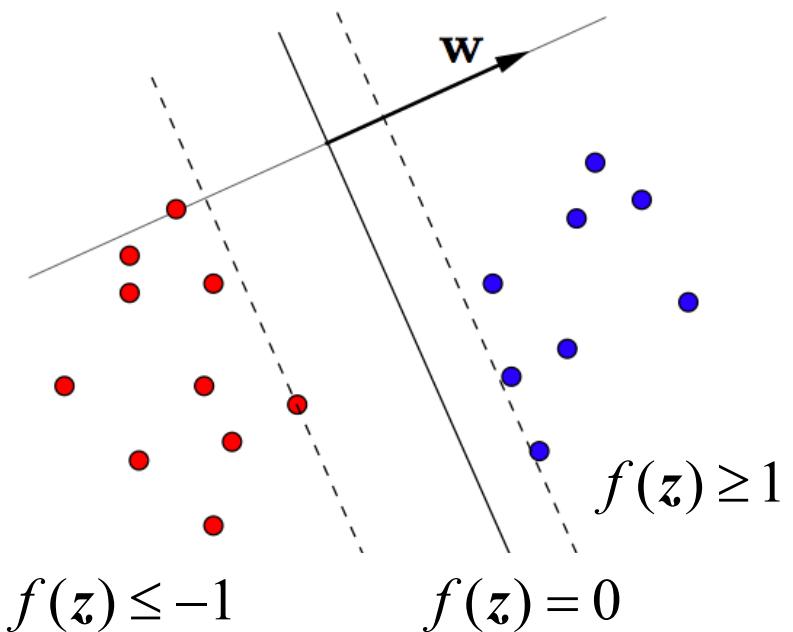
$$\sum_{i=1}^n \alpha_i y_i = 0$$

# The kernel trick (3)

- Classifier can be rewritten as:

$$f(\mathbf{z}) = \mathbf{w}^T \Phi(\mathbf{z}) + w_0$$

$$= \sum_{i=1}^n \alpha_i y_i \Phi(\mathbf{x}_i)^T \Phi(\mathbf{z}) + w_0$$



$$f(\mathbf{z}) \leq -1$$

$$f(\mathbf{z}) \geq 1$$

- Optimization problem can be rewritten as:

$$\max_{\boldsymbol{\alpha}} \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n y_i y_j \alpha_i \alpha_j \Phi(\mathbf{x}_i)^T \Phi(\mathbf{x}_j)$$

$$\alpha_i \geq 0, \quad \forall i$$

$$\sum_{i=1}^n \alpha_i y_i = 0$$

- Only need to specify **kernel** (inner product of transformed points):

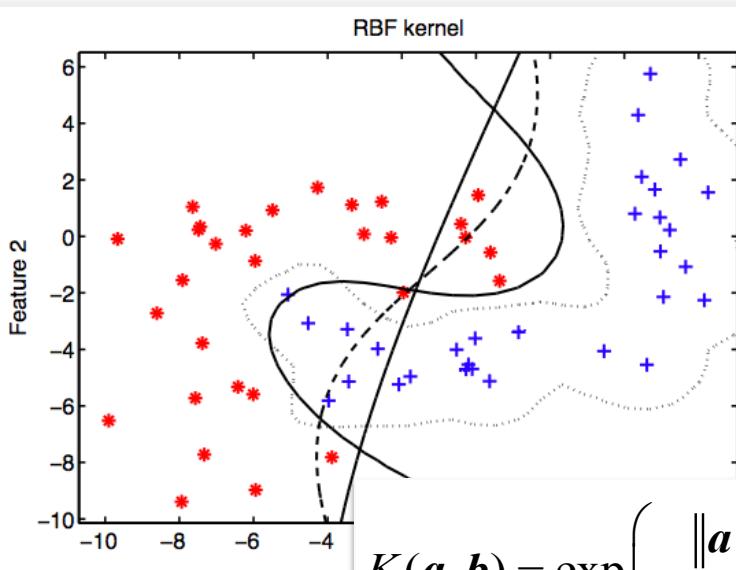
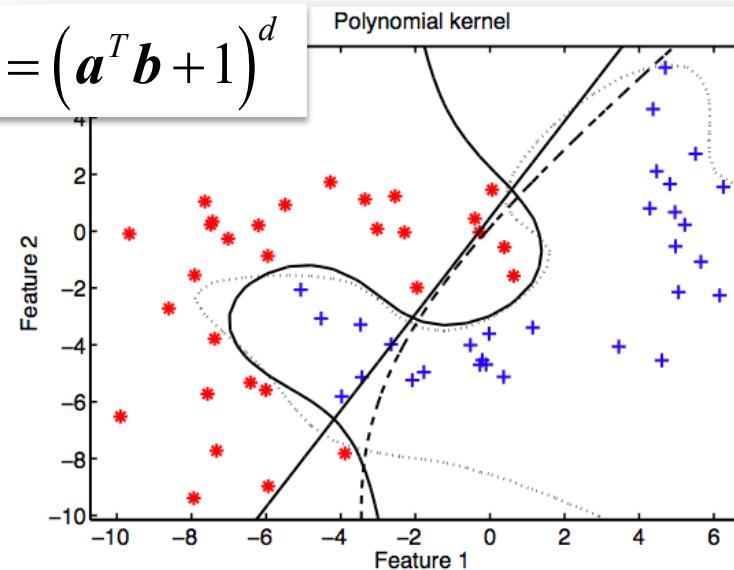
$$K(\mathbf{a}, \mathbf{b}) = \Phi(\mathbf{a})^T \Phi(\mathbf{b})$$

(inner product = distance)

# Kernels

- Kernels  $K(\mathbf{a}, \mathbf{b}) = \Phi(\mathbf{a})^T \Phi(\mathbf{b})$ : nonlinear classifier in original space
- Not necessary to actually know  $\Phi(\cdot)$ ,  
as long as  $K(\mathbf{a}, \mathbf{b})$  fulfills some conditions (!) (positive semi-definite)

$$K(\mathbf{a}, \mathbf{b}) = (\mathbf{a}^T \mathbf{b} + 1)^d$$



$$K(\mathbf{a}, \mathbf{b}) = \exp\left(-\frac{\|\mathbf{a} - \mathbf{b}\|^2}{\sigma^2}\right)$$

- Also kernel versions of  
PCA, ICA, LDA, CCA, ...

# Kernels (2)

- Vector kernels:

- Linear

$$K(\mathbf{a}, \mathbf{b}) = \mathbf{a}^T \mathbf{b}$$

- Polynomial

$$K(\mathbf{a}, \mathbf{b}) = (\mathbf{a}^T \mathbf{b} + 1)^d$$

- Radial basis function

$$K(\mathbf{a}, \mathbf{b}) = \exp\left(-\frac{\|\mathbf{a} - \mathbf{b}\|^2}{\sigma^2}\right)$$

# Kernels (3)

- For other data types: empirical kernel map
  - If we have a distance measure (not *per se* positive definite), then for each object we can construct a vector with distances to a number of other objects
  - This vector can then be used in a vector kernel
- Example: BLAST kernel
  - BLAST a set of sequences w.r.t. each other
  - Represent each sequence by a vector of  $-\log(E)$ -values
  - Use linear kernels on these vectors

# Kernels (4)

- Spectrum kernel:
  - Construct a dictionary of all  $k$ -mers
  - Construct vector with #occurrences of each  $k$ -mer
  - Use this in a linear kernel
  - Need for smart data structures (trie)
  - Versions with gaps, substitutions, wildcards...

• Example:

$a = \text{aabbbababa}$

$b = \text{abbaabbab}$

$\xrightarrow{\quad}$

aabb	abba	bbab	baba	abab	bbaa	baab
1	1	1	2	1	0	0
1	2	1	0	0	1	1

$\xrightarrow{\quad} K(a,b) = 4$

# Kernels (10)

- Convolution kernel:
  - When kernels operate on subparts, but it is not clear which subparts
  - Try all possible decompositions into subparts:

$$K_1 \otimes K_2 \otimes \dots \otimes K_n(\mathbf{a}, \mathbf{b}) = \sum_{\substack{\mathbf{a} = \mathbf{a}_1 \mathbf{a}_2 \dots \mathbf{a}_n \\ \mathbf{b} = \mathbf{b}_1 \mathbf{b}_2 \dots \mathbf{b}_n}} K_1(\mathbf{a}_1, \mathbf{b}_1) K_2(\mathbf{a}_2, \mathbf{b}_2) \dots K_n(\mathbf{a}_n, \mathbf{b}_n) s$$

# Kernels (11)

- Local alignment kernel:

- Trivial kernel:  $K_t(\mathbf{a}, \mathbf{b}) = 1$

- Letter alignment kernel: 
$$K_a(\mathbf{a}, \mathbf{b}) = \begin{cases} 0 & |\mathbf{a}| > 1 \vee |\mathbf{b}| > 1 \\ \exp(\beta S(\mathbf{a}, \mathbf{b})) & \text{otherwise} \end{cases}$$

with  $S$  the substitution cost

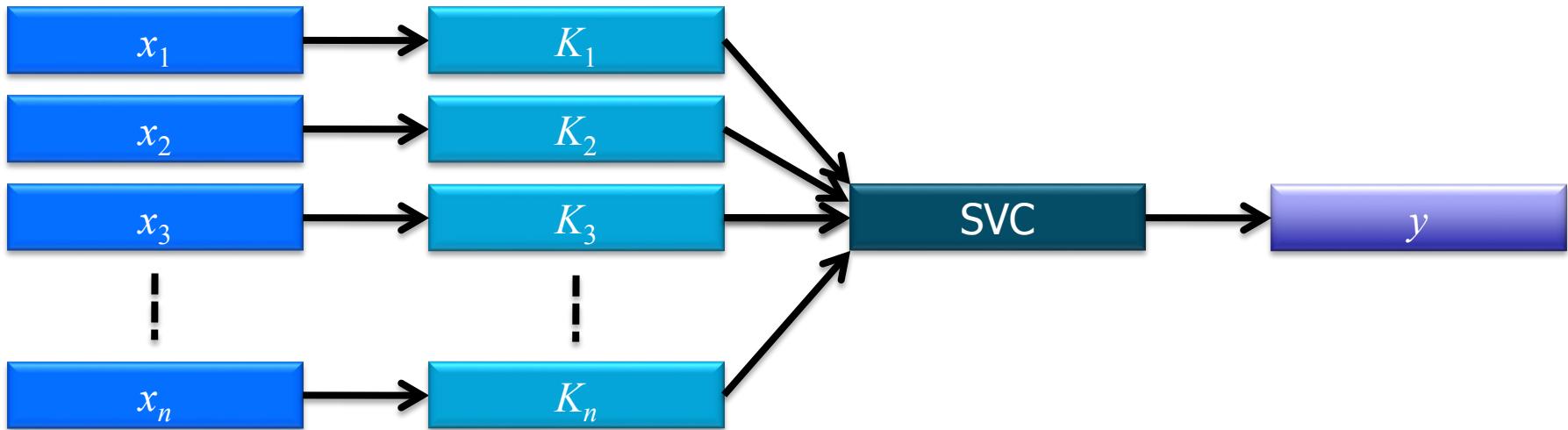
- Gap kernel:  $K_g(\mathbf{a}, \mathbf{b}) = \exp(\beta(|\mathbf{a}| + |\mathbf{b}|))$
- Local alignment kernel of length  $n$ :

$$K_{la(n)}(\mathbf{a}, \mathbf{b}) = K_t \otimes \left( K_a \otimes K_g \right)^{(n-1)} \otimes K_a \otimes K_t(\mathbf{a}, \mathbf{b})$$

- Local alignment kernel:

$$K_{la}(\mathbf{a}, \mathbf{b}) = \sum_{n=0}^{\infty} K_{la(n)}(\mathbf{a}, \mathbf{b})$$

# Kernel combination



- Combination: weighted sum of normalized kernel matrices

$$K'_i(a, b) = \frac{K_i(a, b)}{\sqrt{K_i(a, a)K_i(b, b)}}$$

$$K_{combined}(a, b) = \sum_{i=1}^n w_i K'_i(a, b)$$

*powerful: can apply optimal kernel to each data type*

# Recapitulation

- The *support vector classifier* is based on a well-founded theoretical basis (discussed later)
- The original support vector classifier is limited to problems with two non-overlapping classes, but:
  - can be extended to overlapping classes using *slack variables*
  - can be extended to nonlinear decision boundaries using *kernels*
  - can be extended to multiple classes by combining multiple 2-class classifiers
- A large number of specific kernels for biological data are available
- A support vector regressor is available (not discussed)

# Recapitulation (2)

- Classification performance is often very good
- In particular suited for problems with high-dimensional datasets, for which classes are often separable (and hence there is no need for estimating densities)
- The optimization problem is formulated in terms of the training objects, not the features: slow training for large datasets
- The value for the slack variable trade-off  $C$  and kernel-specific parameters  $d, \sigma$  etc. have to be set

*Kernels need to be chosen, also an ART!*

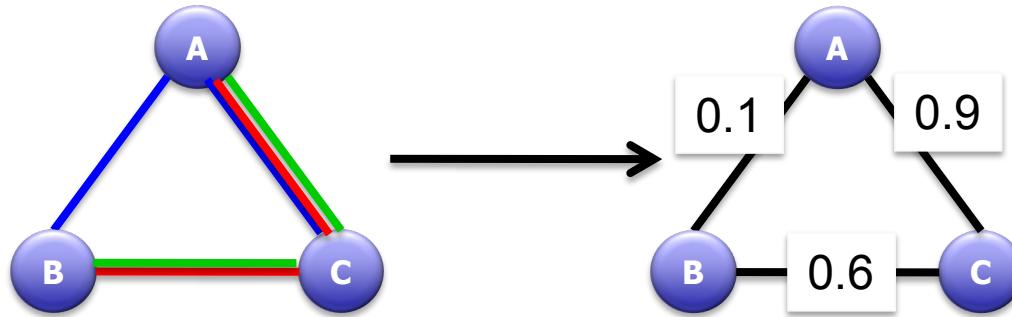


10 min break

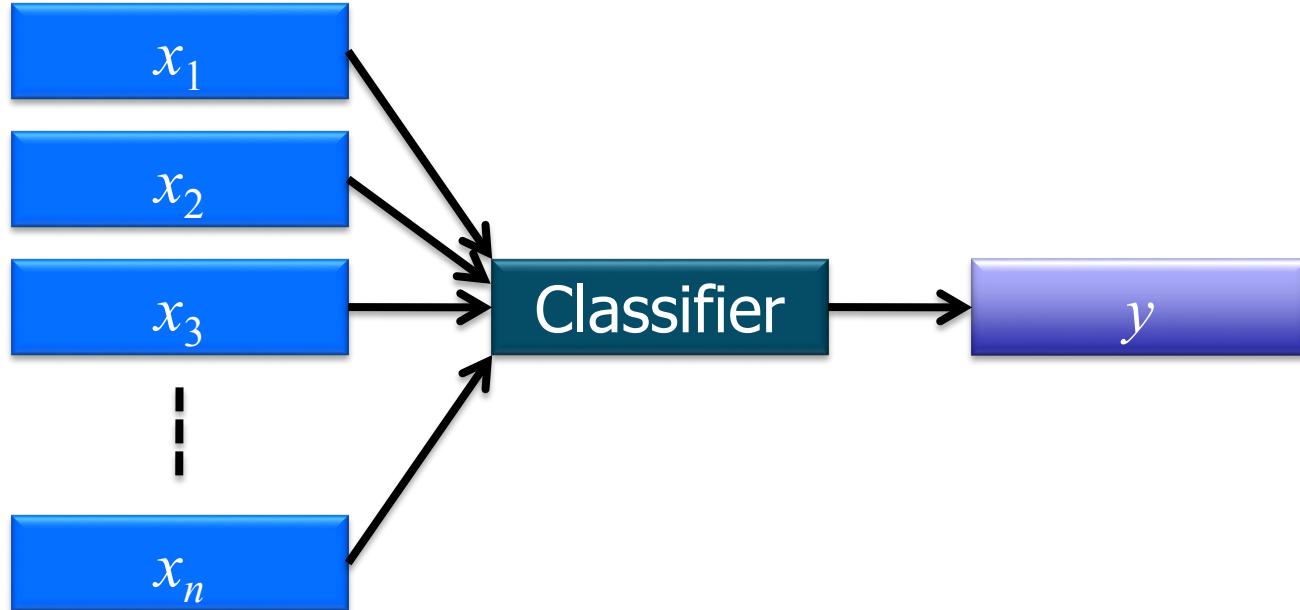
# Classifier combination

# Data integration

- Often required in bioinformatics, e.g. in interaction prediction

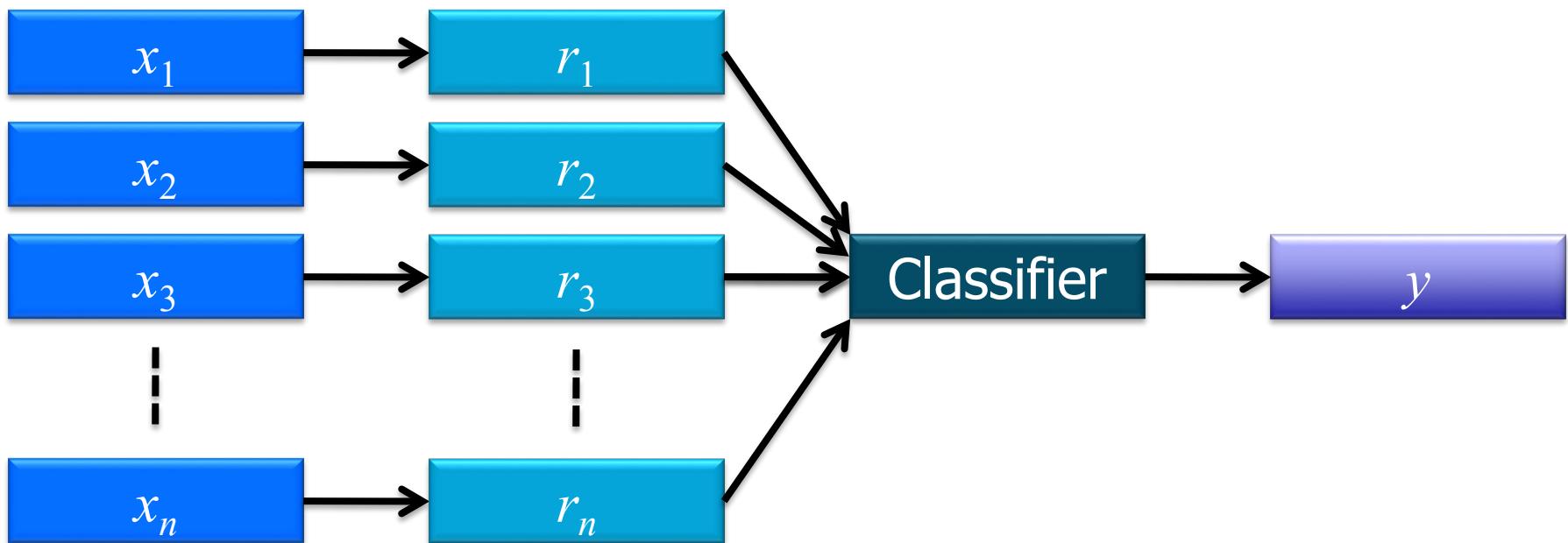


- Early integration: feature fusion



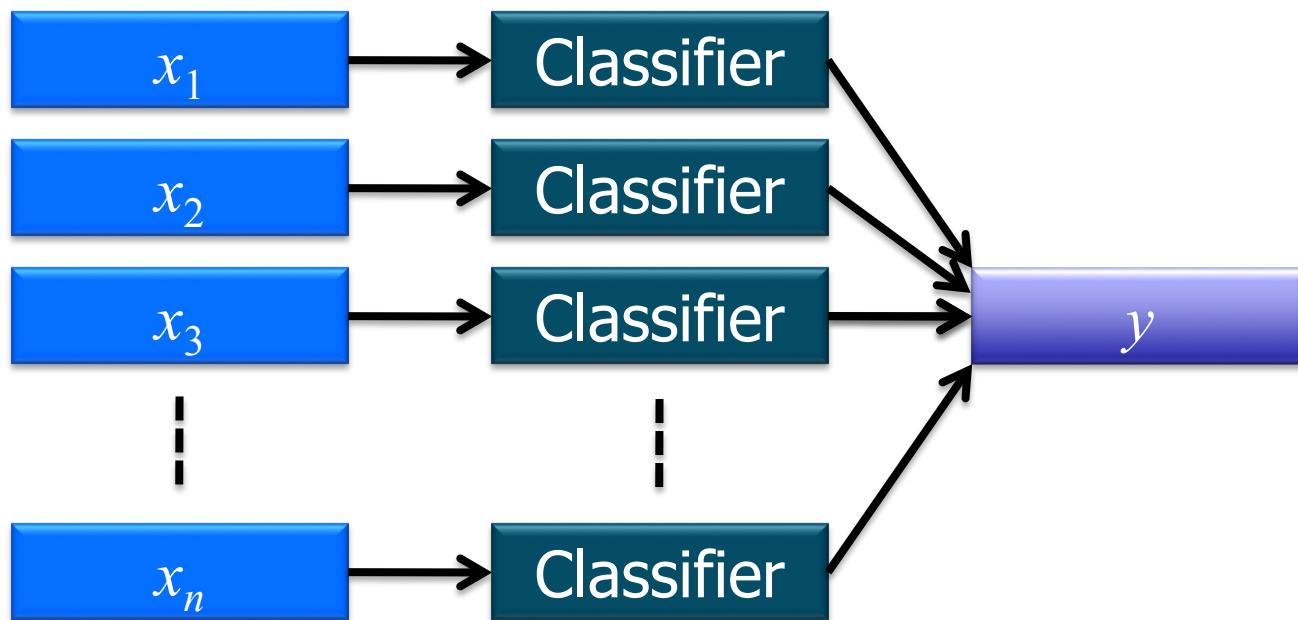
# Data integration (2)

- Intermediate integration: common representation (e.g. kernels or probability distributions)



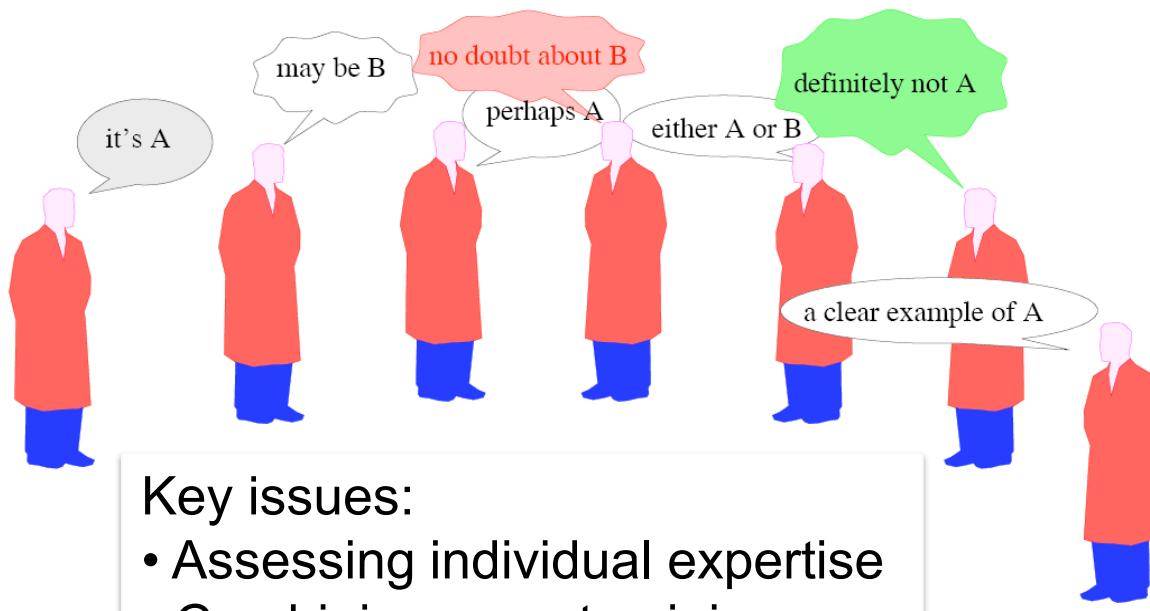
# Data integration (3)

- Late integration: classifier combination

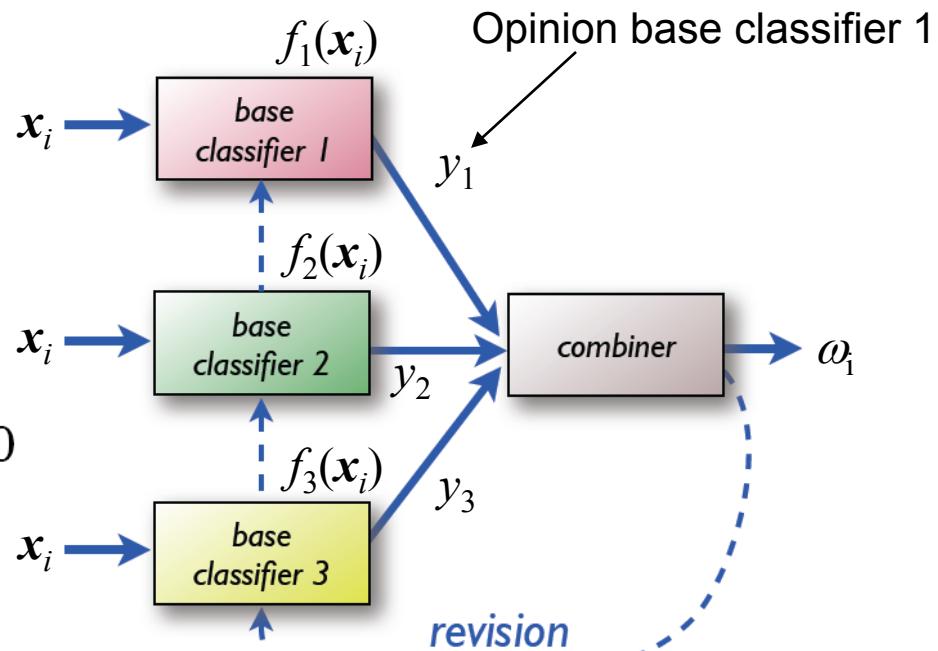
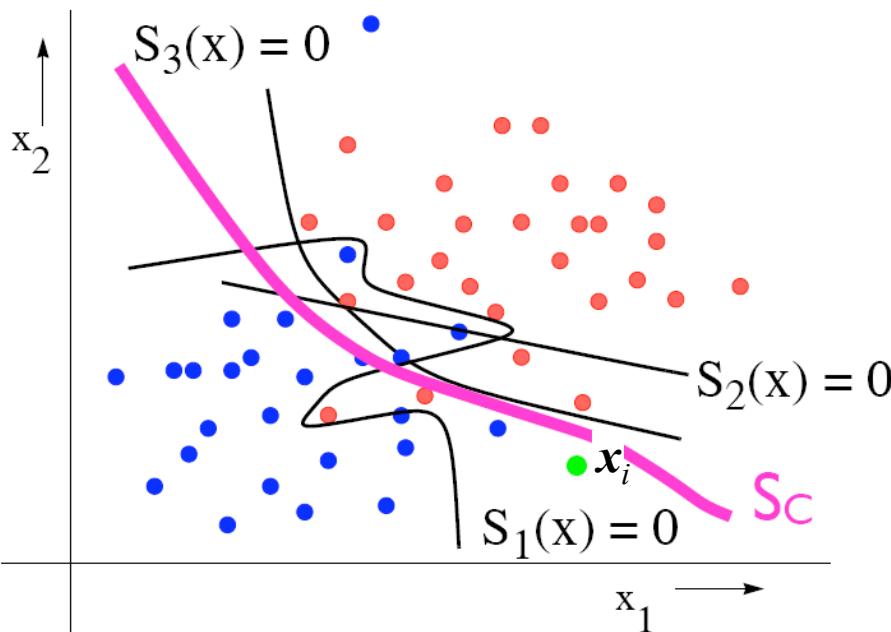


# Classifier combination

- Design choices:
  - Base classifier: Identical or different?  
Base classifiers, feature spaces, training sets, initialisations, etc.
  - Combination by a fixed rule or by another classifier?
- Related to work on committees-of-experts



# Fixed combination



- Classifiers: individual opinion = **posterior probabilities or labels**
- Combination by **fixed rule**, e.g.:

$$\omega_i = \arg \max_c (\text{combination-rule}(y_{j,c} = f_{j,c}(x_i)))$$

i.e. assign label  $\omega_i = c$  to object  $x_i$  if the combination of outputs  $y_{j,c}$  for class  $c$  over all classifiers  $f_j(x_i)$  is maximum

*Combination rule might be maximum over all classifiers  $j$ , or votes by all classifiers for that class*

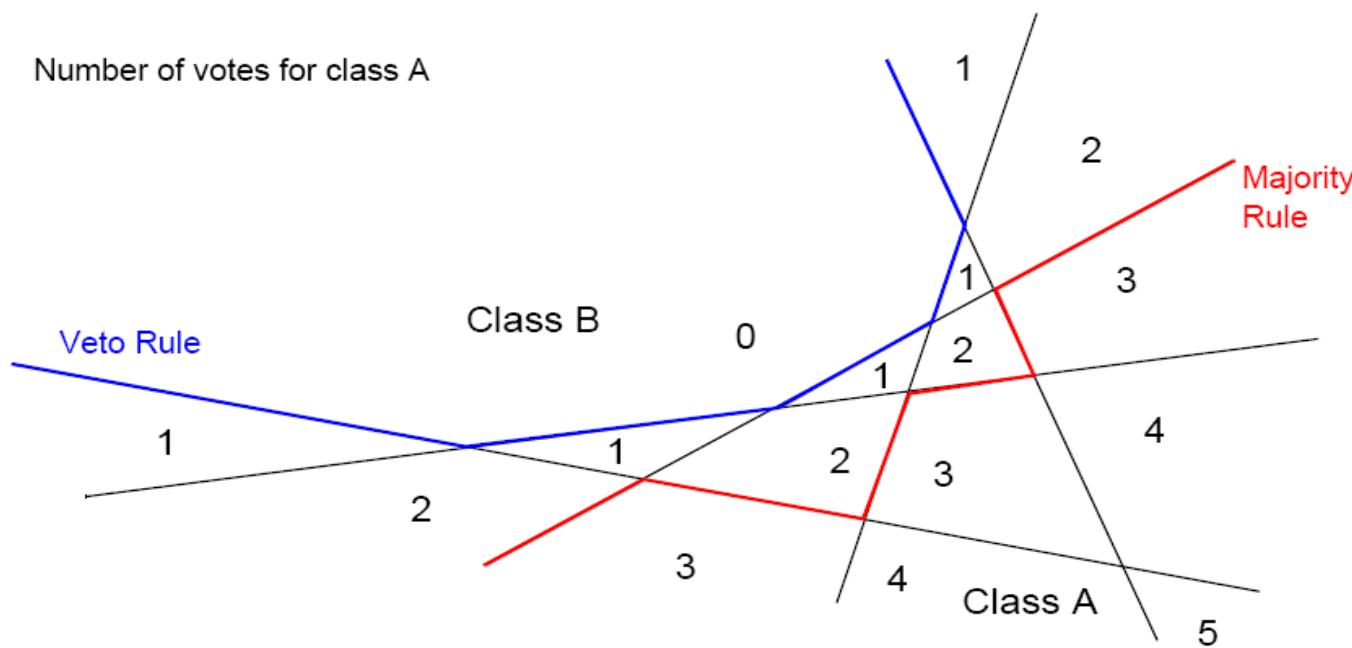


# Fixed combination (2)

- Combination rules on posterior probabilities  $y_{j,c} = p(\omega_i=c|x_i)$ :
  - Generally applicable:
    - Maximum, to select “most confident” classifier  
(assumes good estimates of posteriors)
    - Preferable for classifiers trained in different feature spaces:
      - Product, justified if feature spaces independent
      - Minimum, to select “least objecting” classifier  
(assumes good estimates of posteriors)
    - Preferable for comparable classifiers trained on the same features:
      - Sum/median, to (robustly) improve estimates of posteriors

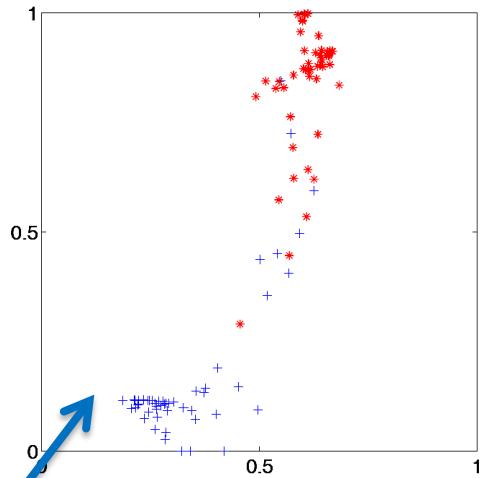
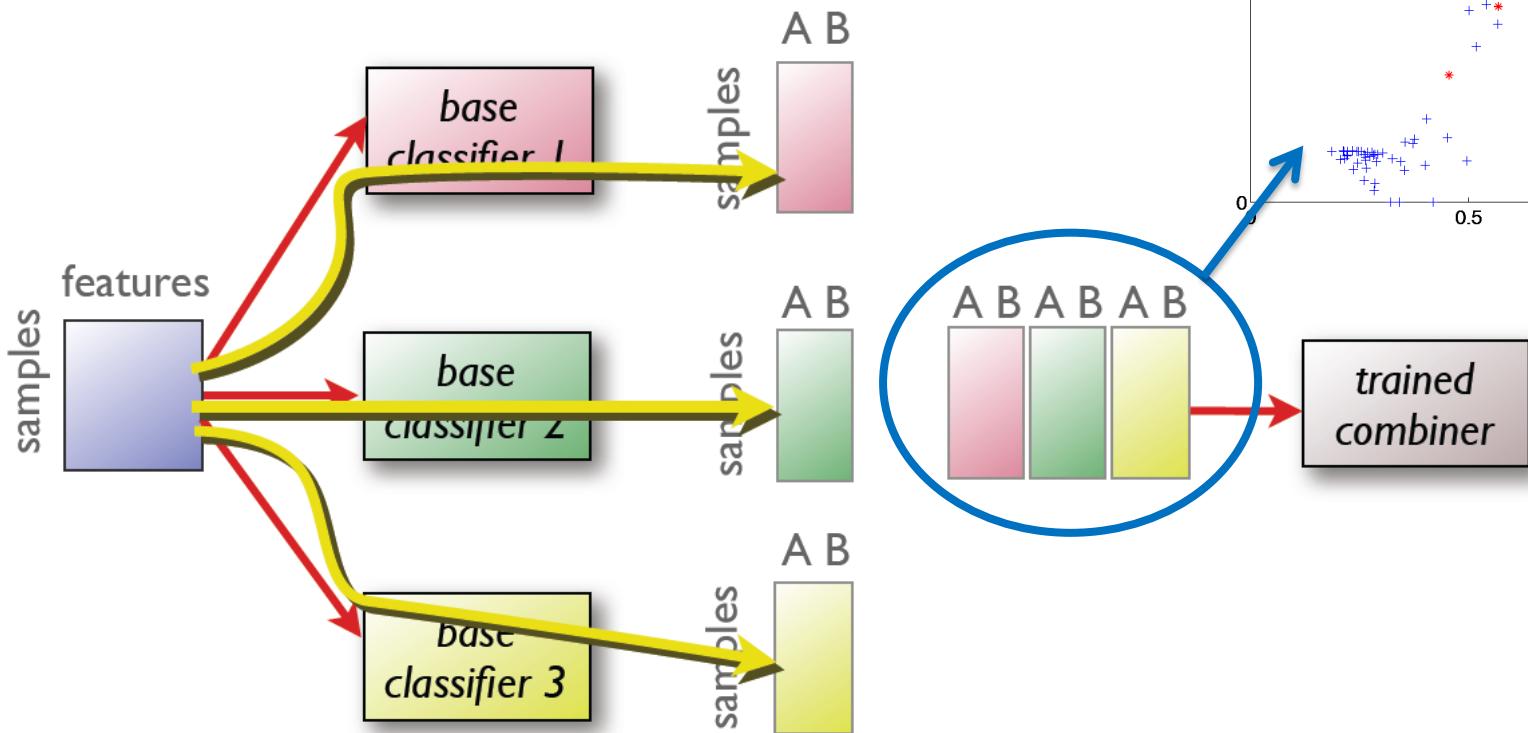
# Fixed combination (4)

- Alternatively, combine labels assigned by classifiers:
  - Veto (like minimum, but needs reject)
  - Majority vote (like sum/median)



# Trained combination

- Treat base classifier outputs as new dataset



- In principle, possible to use any classifier
- Danger of overtraining when using full training set for both stages: use cross-validation

# Base classifier generation

*Let's not combine some classifiers,  
but set out to generate MANY*

- Bagging: bootstrapping and aggregating
  - For  $B$  repetitions
    - Sample a subset of size  $n' < n$  using bootstrapping
    - Train classifier on this subset (e.g. linear or decision tree)
  - Combine  $B$  classifier outputs (e.g. sum or vote)
- Boosting:
  - Initialize all objects with equal weight
  - As often as necessary
    - Sample a subset of size  $n' < n$  according to object weights
    - Train a *weak classifier* on this subset
    - Increase weights of incorrectly classified objects
  - Combine classifier outputs



*Use weak classifiers: only sensible to average over things that differ*

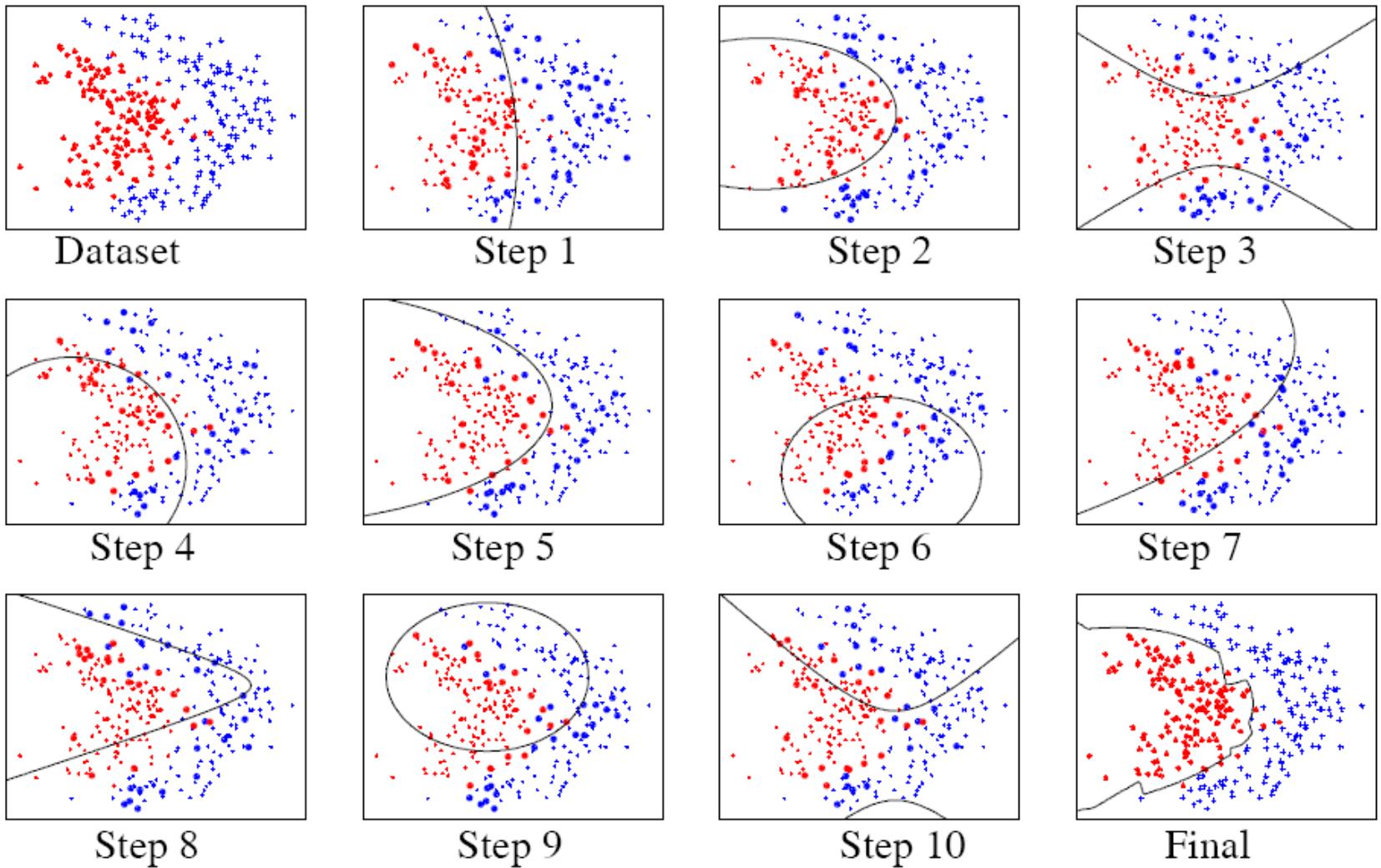
# Base classifier generation (2)

- Adaboost:
  - Initialize all objects with equal weight
  - As often as necessary
    - Select a train set size  $n' < n$  according to object weights
    - Train a weak classifier  $j$
    - Classify entire data set and calculate classifier error  $e_j$
    - Calculate classifier weight  $\alpha_j = 0.5 \log((1-e_j)/e_j)$
    - Multiply weights of incorrectly classified objects with  $\exp(\alpha_j)$ , multiply weights of correctly classified objects with  $\exp(-\alpha_j)$
  - Combine weak classifiers by weighted voting, using  $\alpha_j$

*Boosting: weight objects with #errors*

*Adaboost: weight objects with classifier error*

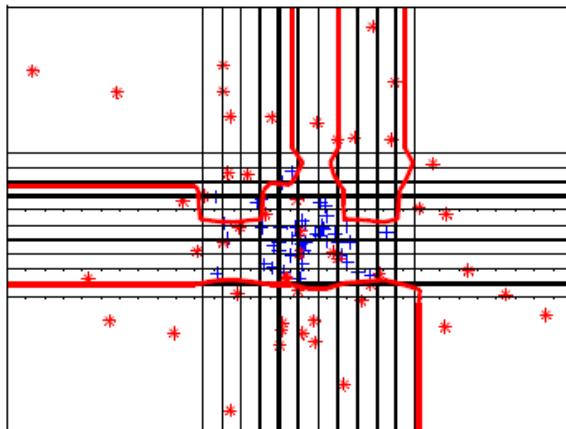
# Base classifier generation (3)



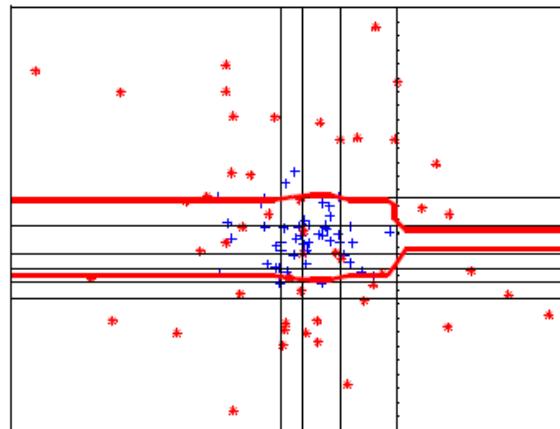
- Adaboost example

# Base classifier generation (4)

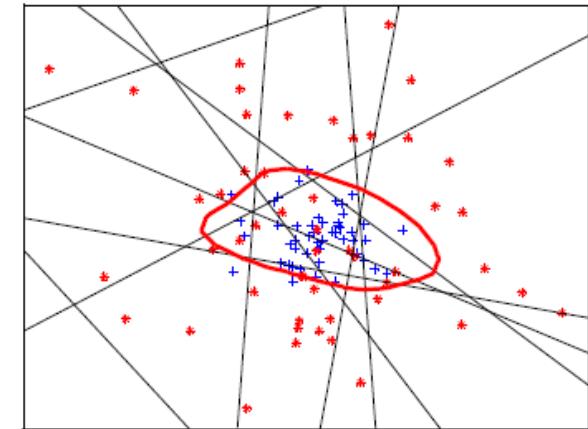
- For all combination methods: base classifier should be fast and weak, i.e. have large bias and small variance
  - Decision stumps: short decision trees
  - Linear classifiers: nearest mean, LDA



100 decision stumps,  
combined by Adaboost



10 decision stumps,  
combined by LDA



10 LDAs,  
combined by LDA

# Recapitulation

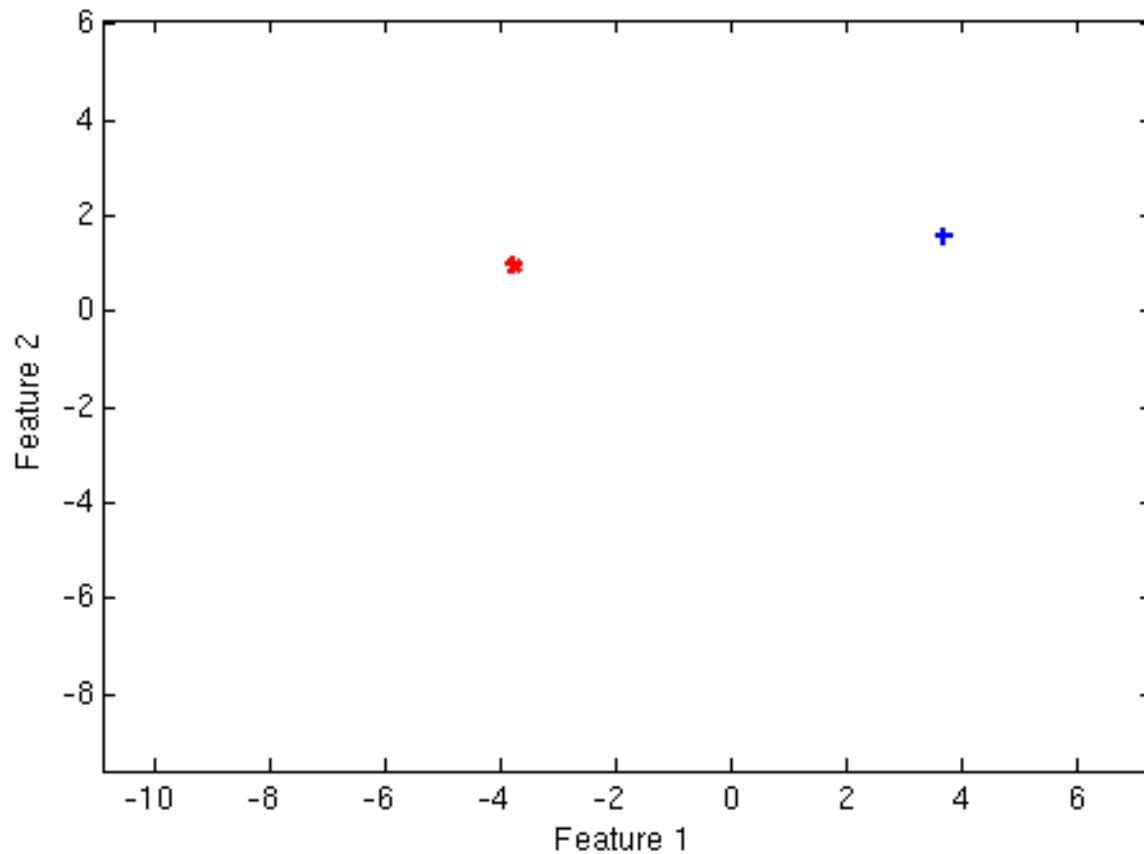
- Combining classifiers can help, but is no panacea
  - *Fixed* combination:
    - Usually sub-optimal
  - *Trained* combination:
    - Use cross-validation to prevent overtraining
- Use *weak* classifiers: fast, large bias, small variance
- Combination requires *variation* between classifiers:
  - Train different classifiers on the same features
  - Train classifiers on different feature spaces (sample features!)
  - Subsample the train set (*bagging, boosting*)



10 min break

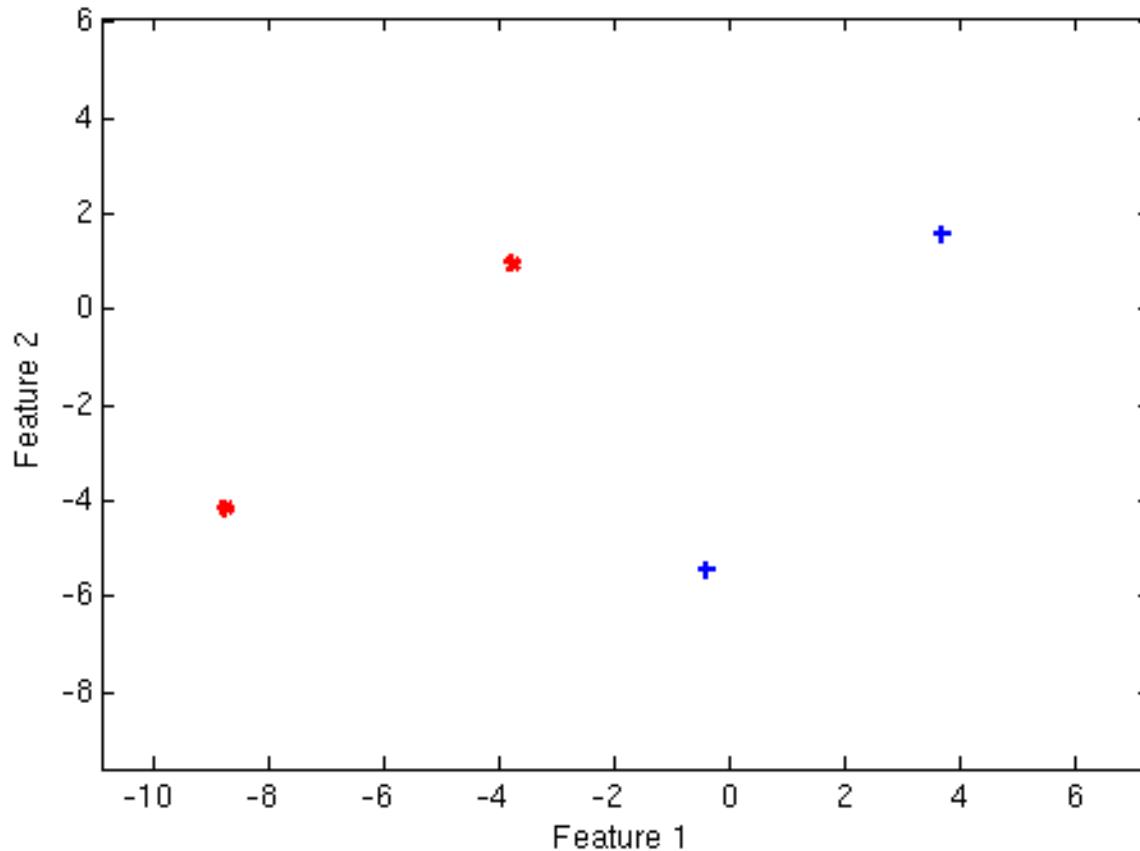
# Complexity

# Sample size



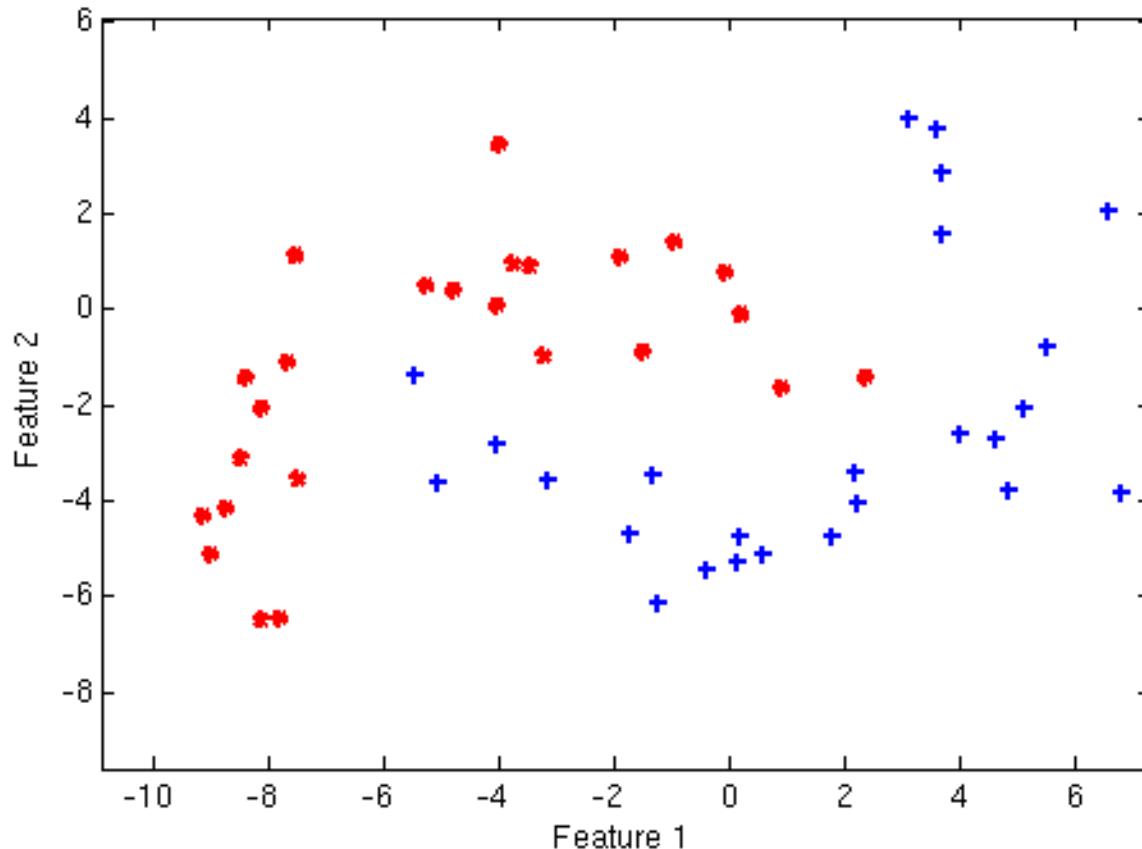
*What is a good classifier?*

# Sample size (2)



*What is a good classifier?  
And now?*

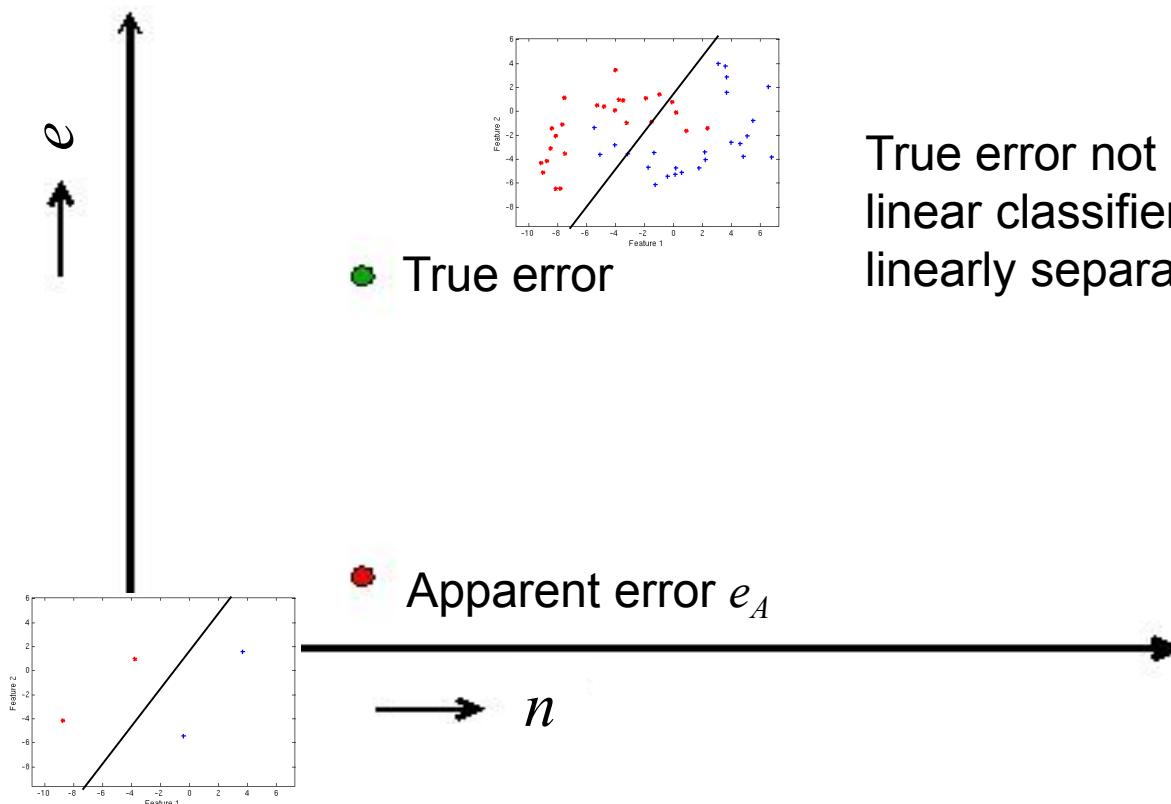
# Sample size (3)



*What is a good classifier?  
And now? Training size matters! But how?*

# Learning curves

- How does the error change with varying sample size (number of objects in the train set)?

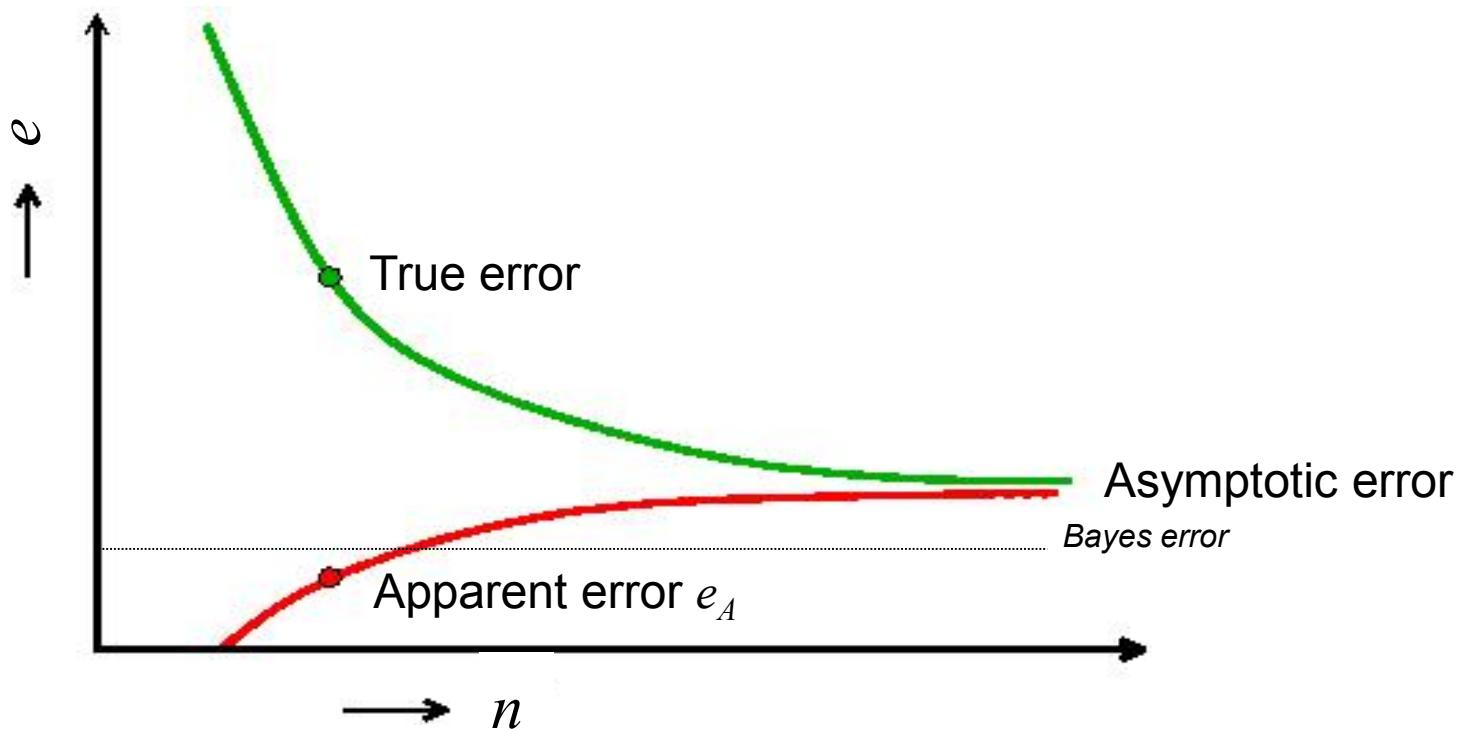


True error not small because of linear classifier and data is not linearly separable

True error: error on infinite test data  
Apparent error: error on training data

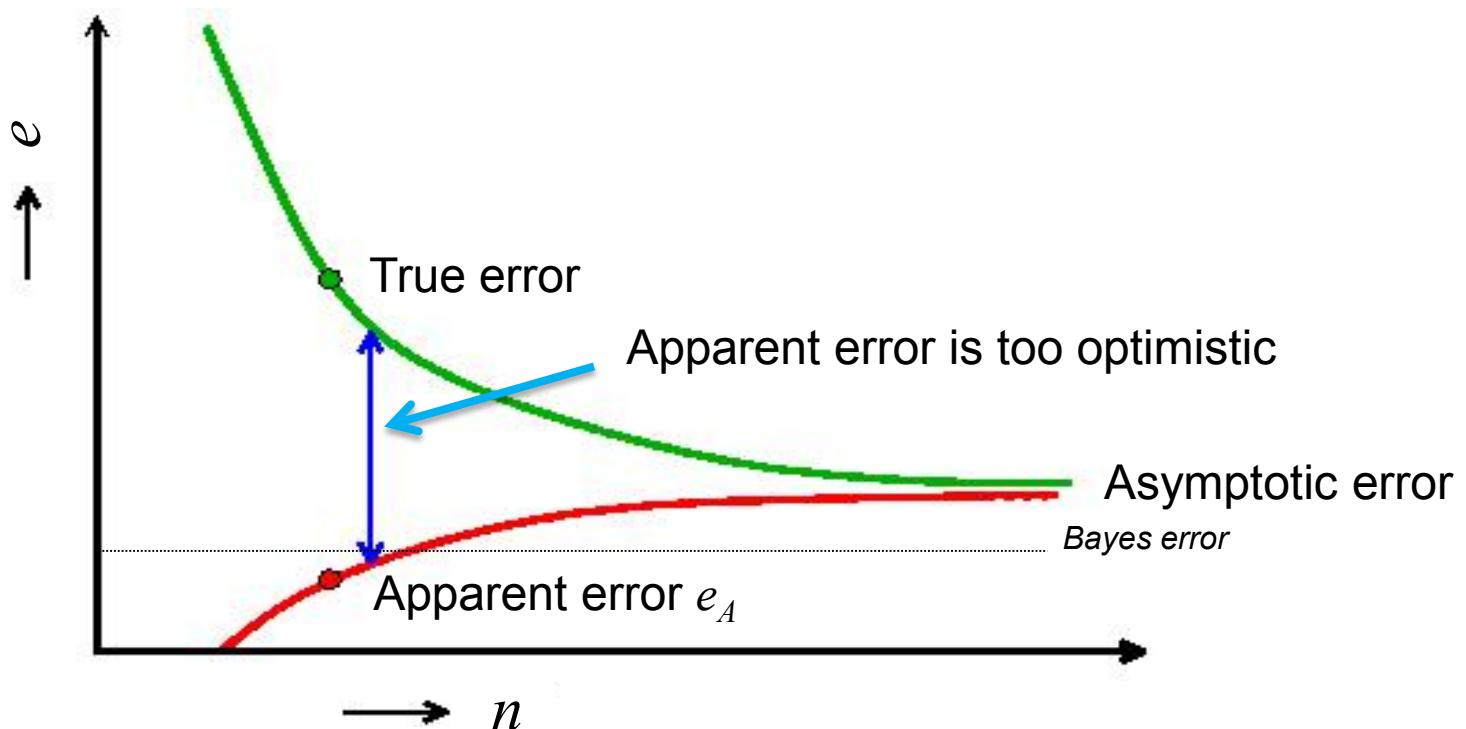
# Learning curves (2)

- How does the error change with varying sample size (number of objects in the train set)?



# Learning curves (3)

- How does the error change with varying sample size (number of objects in the train set)?

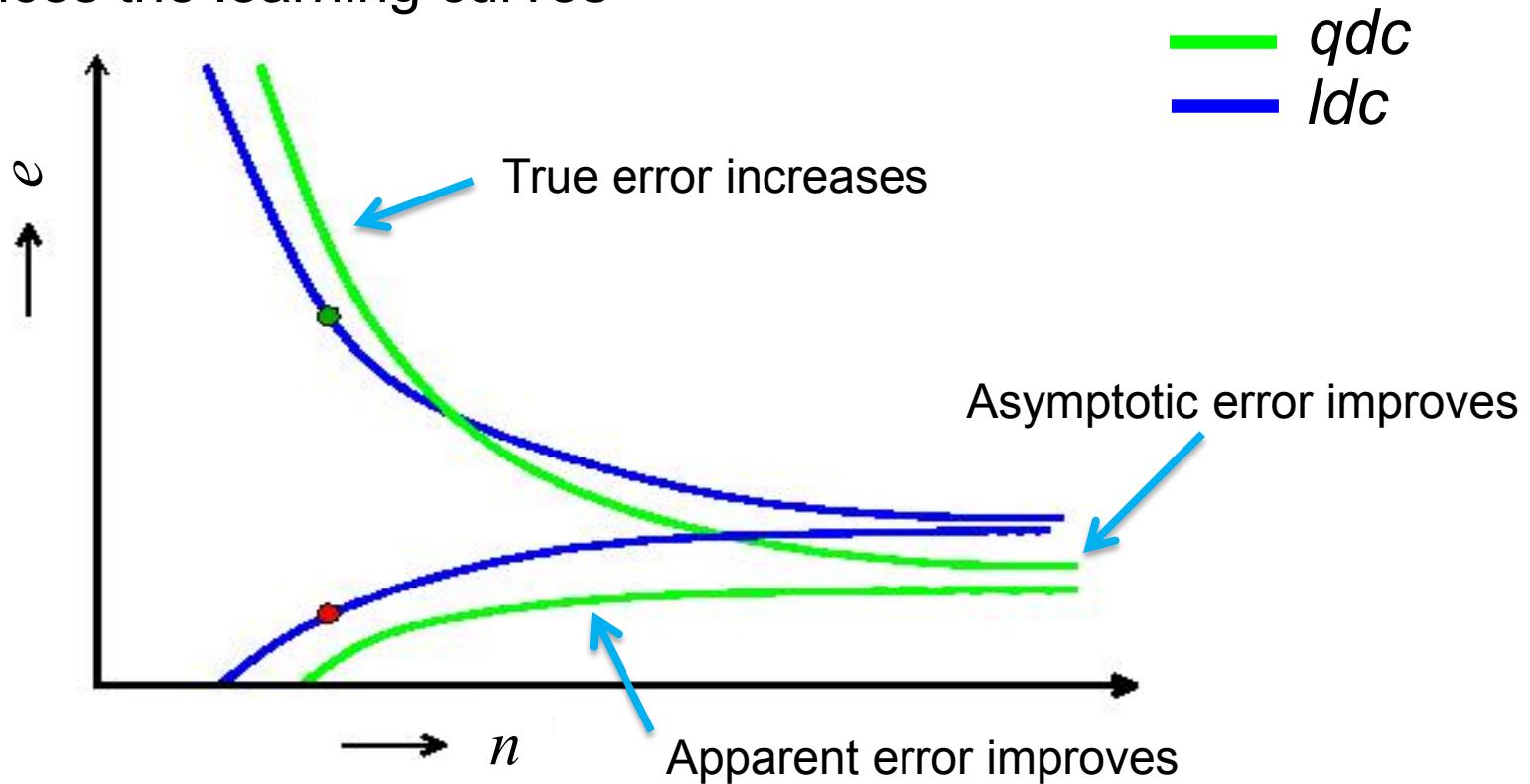


# Learning curves (4)

- What happens when you take another classifier?  
(say, use a `qdc` instead of an `lrc`)
- More flexible:
  - Better performance on the training set
  - Worse performance on the test set
  - Will perform best in the limit of many training objects
- Less flexible:
  - Less adapted to the training set
  - Better performance on the test set
  - Will not perform best in the limit of many training objects

# Learning curves (5)

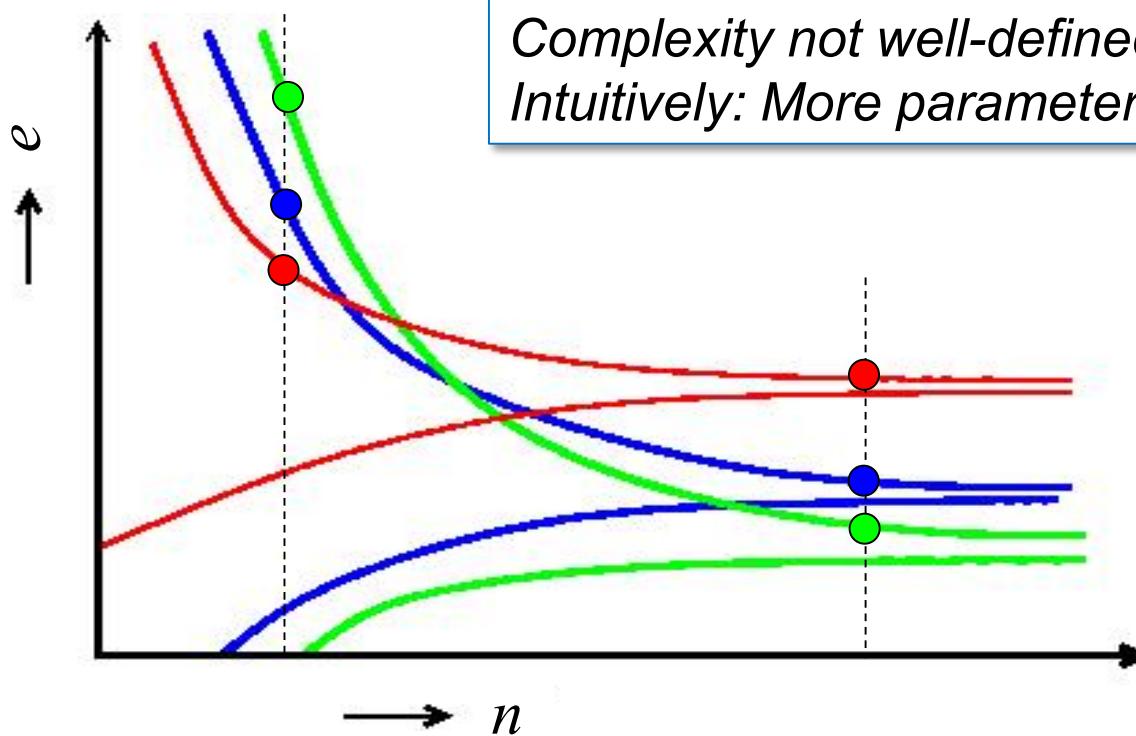
- Switching to a more complex classifier influences the learning curves



- So why not always use complex classifiers?

# Classifier complexity

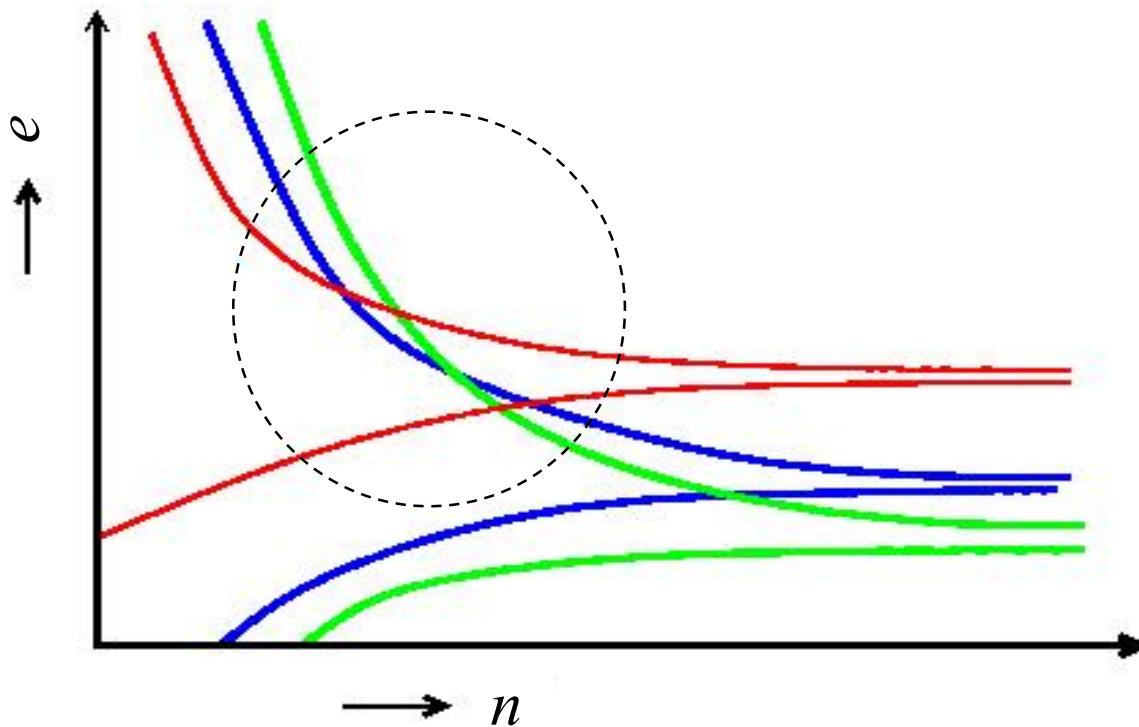
- Optimal complexity depends on sample size



- Small: use a simple classifier
- Large: can use a complex classifier

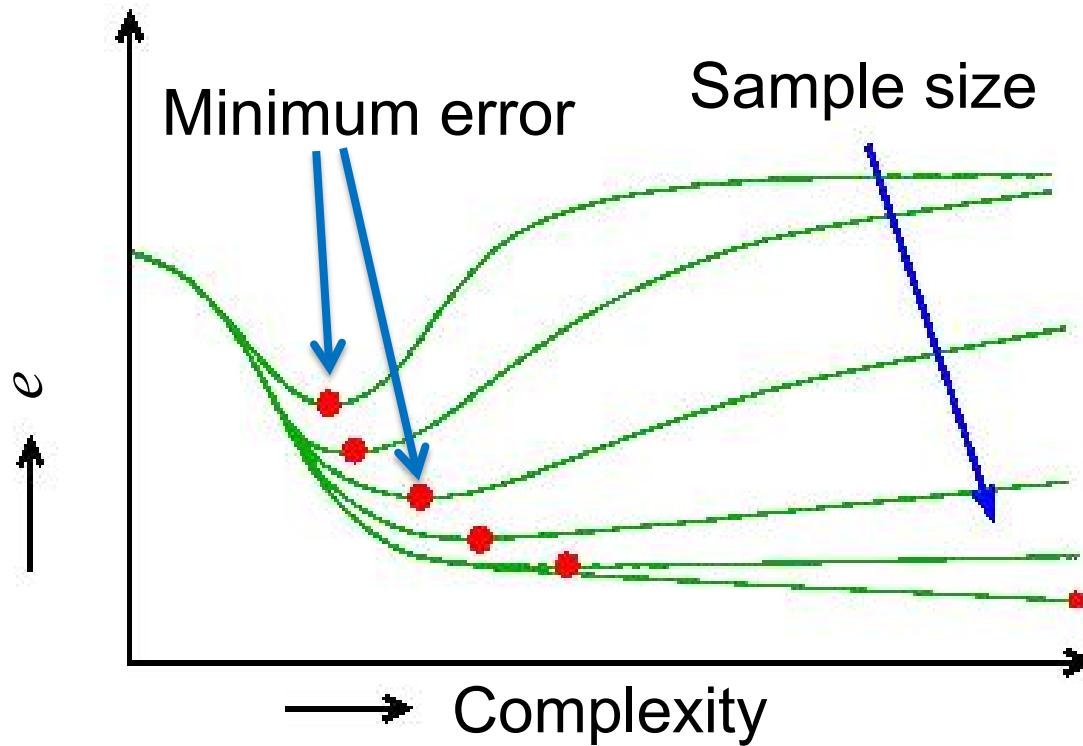
# Classifier complexity (2)

- There is a tradeoff between complexity and training size



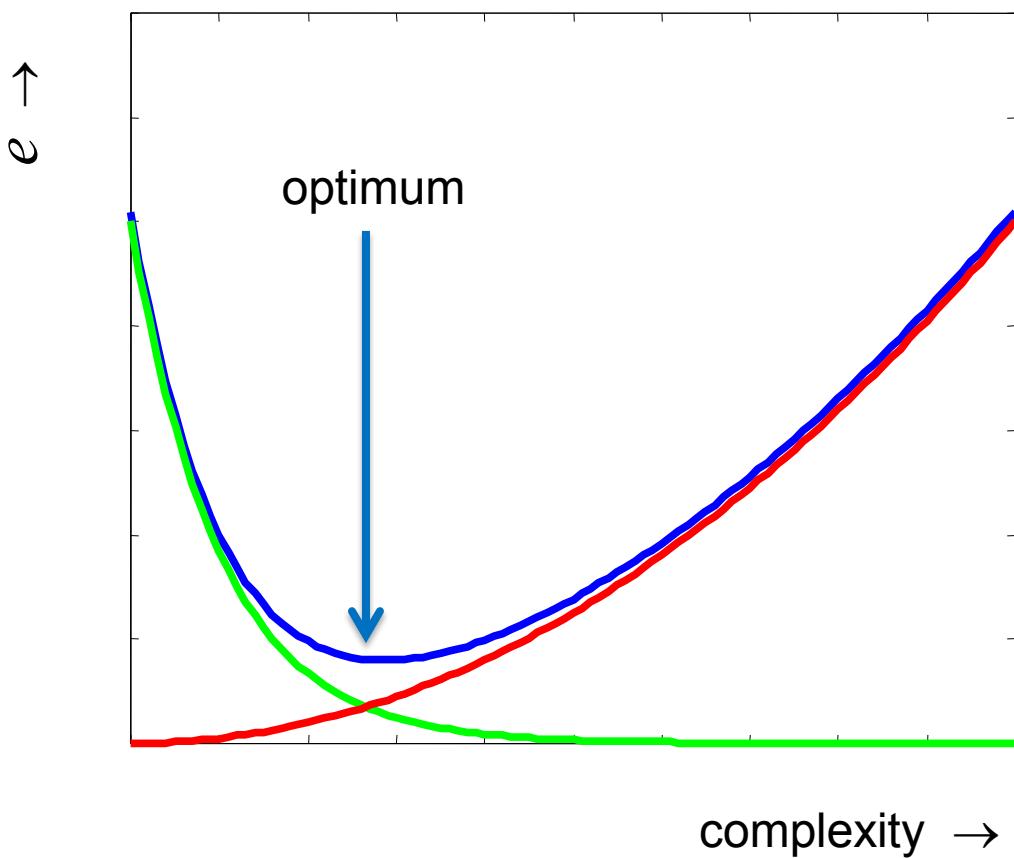
# Classifier complexity (3)

- Remember the curse of dimensionality: for fixed sample size, error increases if classifier complexity increases



# Bias/variance

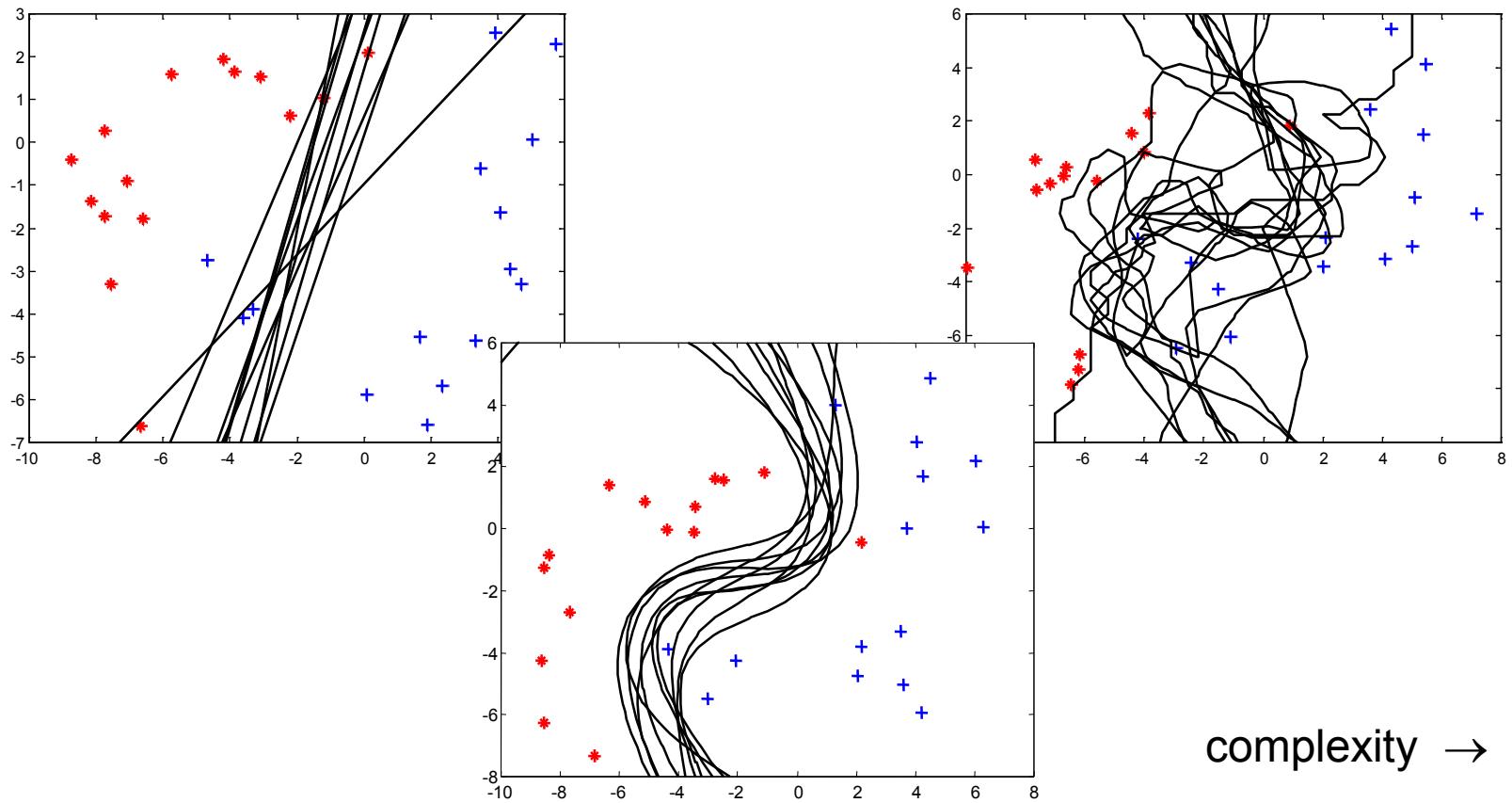
- Total error is combination of bias and variance:
  - Bias
  - Variance
  - Total error



- With increasing sample size, variance component decreases, bias stays the same

# Bias/variance (2)

- In classification:



# Classifier complexity (6)

- How to find the best complexity for a given problem?
- Standard approach:
  - Define a large set of classifiers
  - Use cross-validation, and repeatedly
    - Train all the classifiers on the training set
    - Test all the classifiers on the test set
  - Find the best classifier
- This is a lot of work....

# Regularization

- For many classifiers, it is possible to reduce the complexity of a classifier by adding constraints on the parameters  $\theta$
- Often a term is added to the cost function:

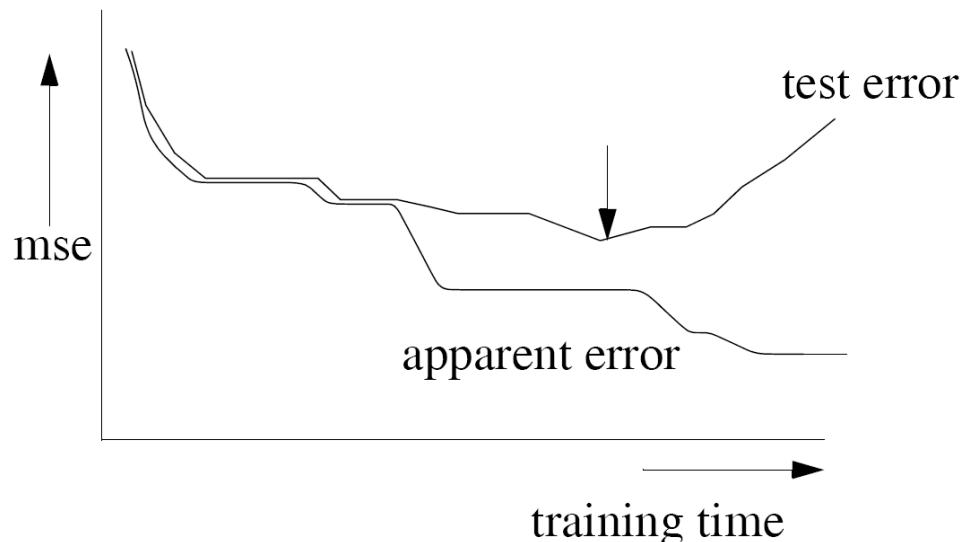
$$E = e_A + \lambda f_{reg}(\theta)$$

- For example:

- Multilayer perceptron:  $E = \sum_{k=1}^n |t_k - g(x_k)|^2 + \lambda \sum_i w_i^2$
- Support vector classifier:  $E = \frac{1}{2} \|w\|^2 + C \sum_{i=1}^n \xi_i$

# Regularization (2)

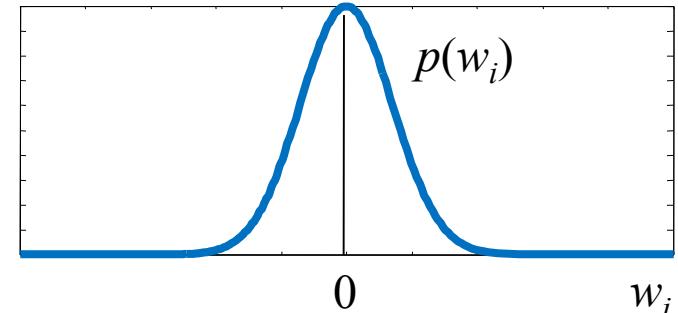
- Another form of regularization:  
starting with small initial weights in training multilayer perceptrons
- Effective complexity of MLPs increases during training



# Regularization (3)

- Intuitively:
  - Regularization is often a quadratic penalty on weight values
  - Small weights correspond to simple classifier, large weights to complex classifiers
  - This boils down to a *prior* on weights
  - For example:

$$E = \sum_{k=1}^n |t_k - g(x_k)|^2 + \lambda \sum_i w_i^2$$



- Regularization is like Bayesian estimation *on parameters*
- Bayesian model selection: apply Bayesian estimation to entire *models* (classifiers/regressors)

# Bayesian model selection

- The *evidence* for model  $M$  is the probability of data  $X = \{x\}$  given model  $M$
- Found by integrating over *all possible values* of parameters  $\theta$ :

$$p(X | M) = \int p(X | M, \theta) p(\theta | M) d\theta$$

- If multiple alternative models are available, use the Bayes factor:

$$\frac{p(X | M_1)}{p(X | M_2)} > 1 \Rightarrow M_1$$

- We can even take priors on models into account:

$$\frac{p(X | M_1)}{p(X | M_2)} \frac{p(M_1)}{p(M_2)} > 1 \Rightarrow M_1$$

# Bayesian model selection (2)

- Integrating over all possible values of  $\theta$  is very hard in practice

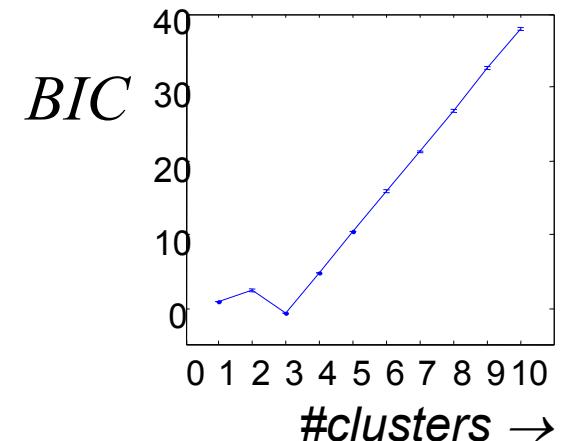
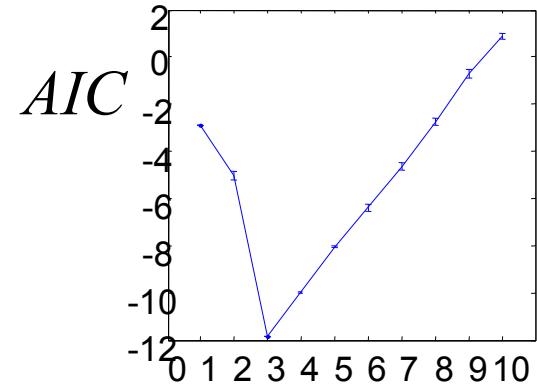
- Use Monte Carlo methods
- Use approximations:
  - Akaike Information Criterion:

$$AIC = 2k - 2 \log[p(X | M, \theta_{opt})]$$

- Bayesian Information Criterion:

$$BIC = k \log(n) - 2 \log[p(X | M, \theta_{opt})]$$

- $k$  = number of parameters
- $n$  = number of training objects
- $\theta_{opt}$  = parameters optimizing likelihood



(fit MoG, 3 Gaussians)

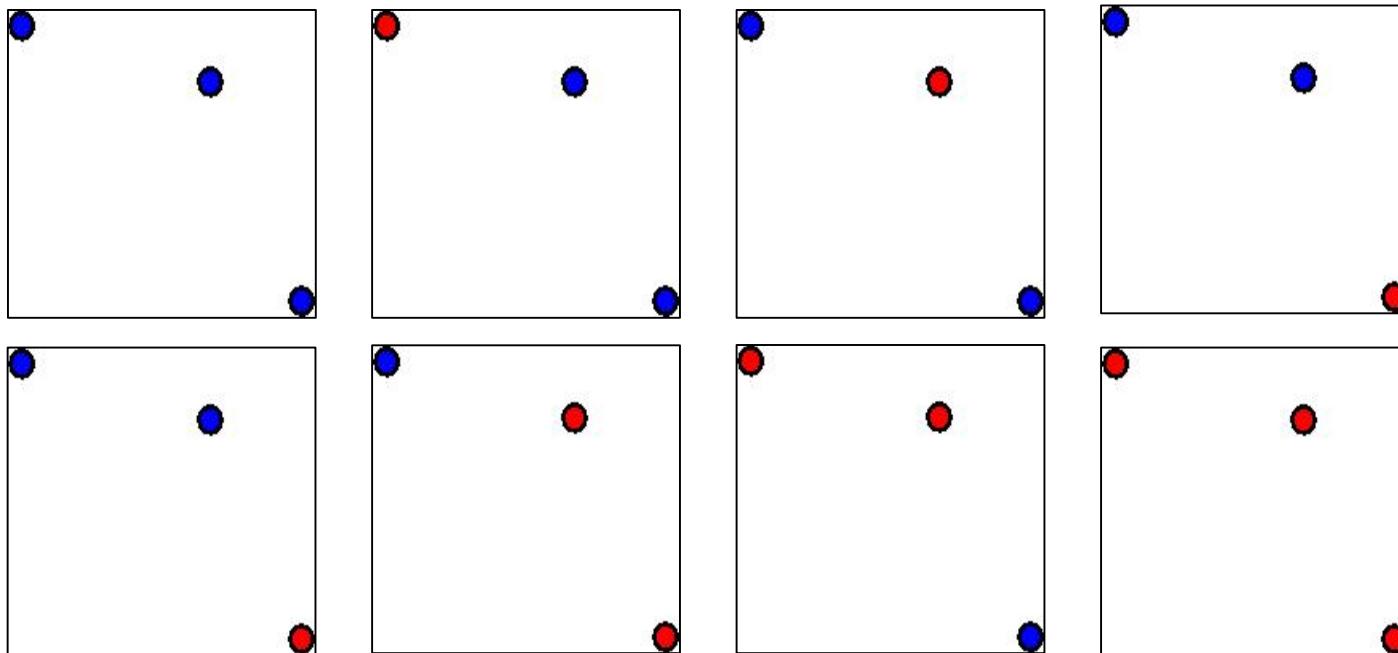




10 min break

# VC dimension

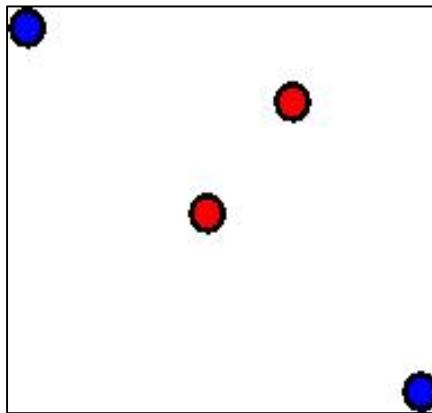
- Complexity measure underlying support vector classifier
- Vapnik-Chervonenkis dimension  $h$  of a two-class classifier:
  - the largest number of objects in “general position” that can be separated in *all possible*  $2^h$  ways



*All possible labelings of  $N = 3$  objects in 2D*

# VC dimension (2)

- For  $N = 3$  objects in 2D we can always find a linear separation
- For  $N = 4$  objects in 2D this is *not* always possible:



- Conclusion: for a linear classifier,  
 $h = p + 1$  (where  $p$  is the dimensionality)
- For (almost) all other classifiers it is *not known* (though some upper bounds exist for neural networks)

# VC dimension (3)

- With probability at least  $1 - \delta$  this inequality holds:

$$e \leq e_A + \frac{1}{2} E(n) \left( 1 + \sqrt{1 + \frac{e_A}{E(n)}} \right)$$

- where

$$E(n) = 4 \frac{h \left( \log \left( \frac{2n}{h} \right) + 1 \right) - \log \left( \frac{\delta}{4} \right)}{n}$$

- When  $h$  is small, the apparent error  $e_A$  is close to the true error  $e$

V.Vapnik, Statistical learning theory, 1998

- An optimal classifier:

- has small apparent error  $e_A$  (i.e. is well-trained)
- has small VC dimension  $h$  (i.e. is simple)

# VC dimension (4)

- Construct a linear classifier with apparent error  $e_A = 0$

$$\mathbf{w}^T \mathbf{x}_i + w_0 \leq -1 \mid y_i = -1$$

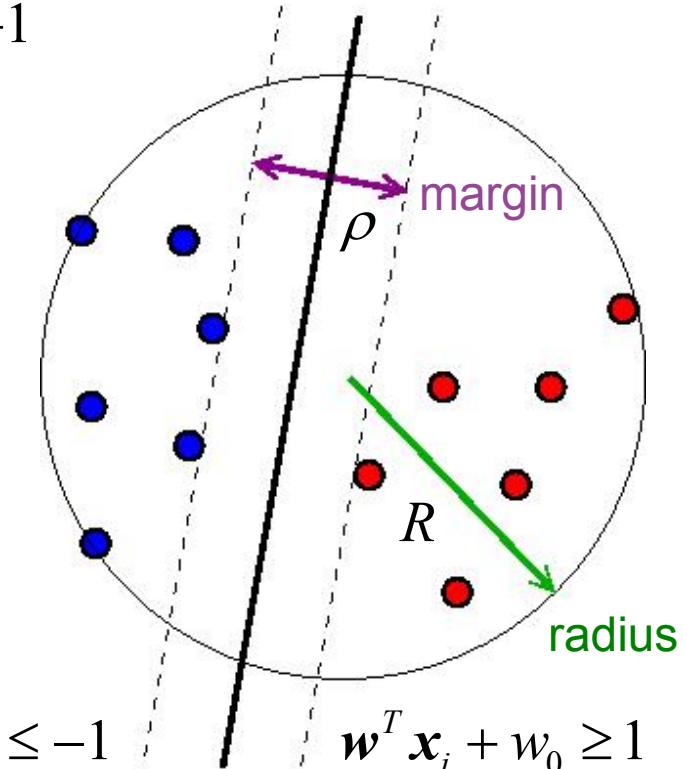
$$\mathbf{w}^T \mathbf{x}_i + w_0 \geq +1 \mid y_i = +1$$

and then minimize VC dimension  $h$

- It can be proven that  $h = \min\left(\frac{R^2}{\rho^2}, p+1\right)$

- $R$  is radius of smallest enclosing ball:  
minimize, but it is fixed

- $\rho = \frac{1}{\|\mathbf{w}\|^2}$  is the margin: maximize,  
 $\mathbf{w}^T \mathbf{x}_i + w_0 \leq -1$   
so minimize  $\|\mathbf{w}\|^2$



- This is the maximum margin classifier!

# Recapitulation

- A fundamental trade-off in pattern recognition is between *model descriptiveness* (e.g. classification error) and *model complexity*
- Optimal complexity depends on the problem and sample size, and can be assessed/controlled through:
  - *Cross-validation and learning curves*
  - *Regularization*
  - *Bayesian information criteria*
- More fundamental approaches are:
  - *Bayesian model selection*
  - *Minimum description length*
  - *VC dimension*

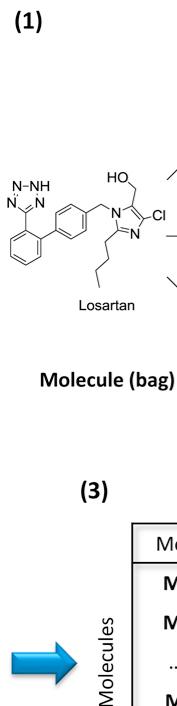
Only the latter leads to a practical solution,  
the support vector classifier

# Recent developments

- Recent developments focus not so much on developing new methods, but tackling new types of problems
  - multiple instance learning
  - structured learning
  - semi-supervised learning
  - active learning
  - and more deep learners ....

# Multiple instance learning

- Uses *bag-of-instances representations* of objects, usually labeling a bag positive if *at least one* instance is labeled positive
- Applications:
  - drug discovery
  - predicting activity of molecules
  - predicting protein binding sites



(2)

Pharmacophore Fingerprint						
Mol.	Conf.	P <sub>1</sub>	P <sub>2</sub>	...	P <sub>k</sub>	...
$M_1$	$C_{11}$	1	0	...	1	...
	$C_{12}$	0	1	...	0	...
$M_2$	$C_{21}$	1	1	...	0	...
	$C_{22}$	1	0	...	0	...
...						
$M_l$	$C_{l1}$	0	1	...	0	...
	$C_{l2}$	1	1	...	1	...
...						

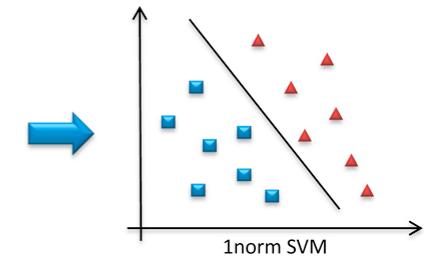
(3)

Mol.	C <sup>1</sup>	C <sup>2</sup>	...	C <sup>n</sup>
$M_1$	$D(M_1, C^1)$	$D(M_1, C^2)$	...	$D(M_1, C^n)$
$M_2$	$D(M_2, C^1)$	$D(M_2, C^2)$	...	$D(M_2, C^n)$
...	...	...	...	...
$M_i$	$D(M_i, C^1)$	$D(M_i, C^2)$	...	$D(M_i, C^n)$
...	...	...	...	...
$M_l$	$D(M_l, C^1)$	$D(M_l, C^2)$	...	$D(M_l, C^n)$
...	...	...	...	...



Molecules

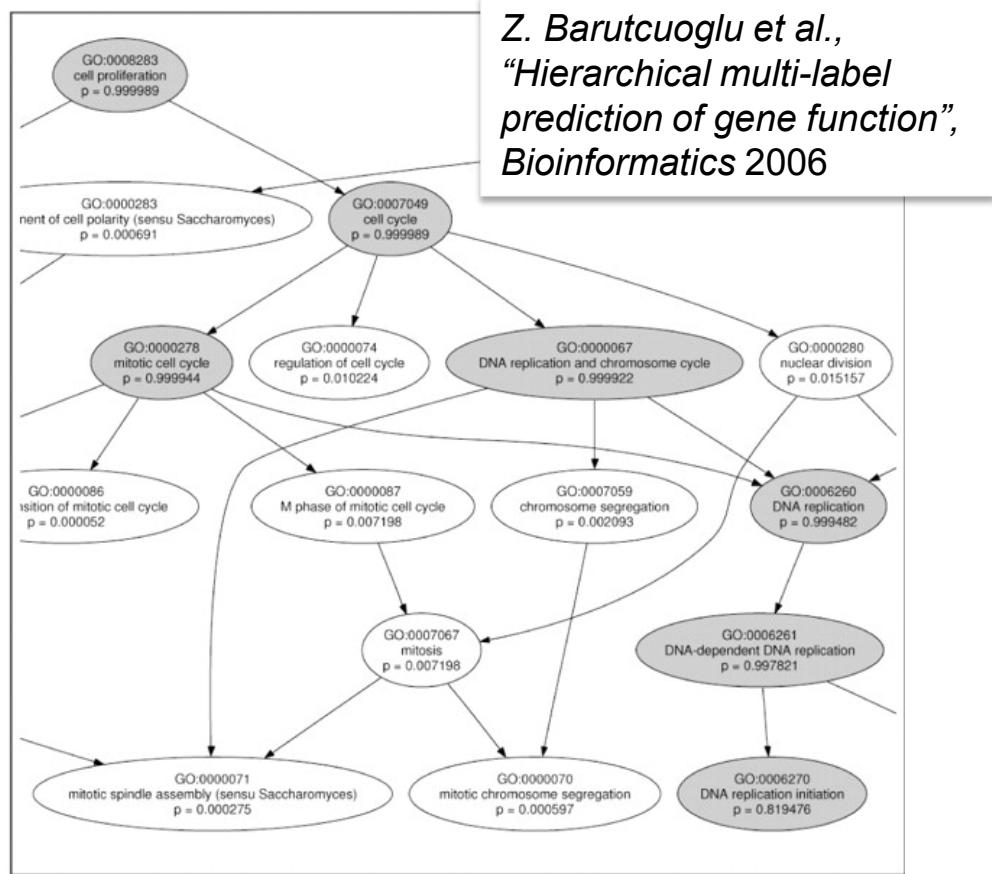
(4)



G. Fu et al, "Implementation of multiple-instance learning in drug activity prediction", BMC Bioinformatics 2012

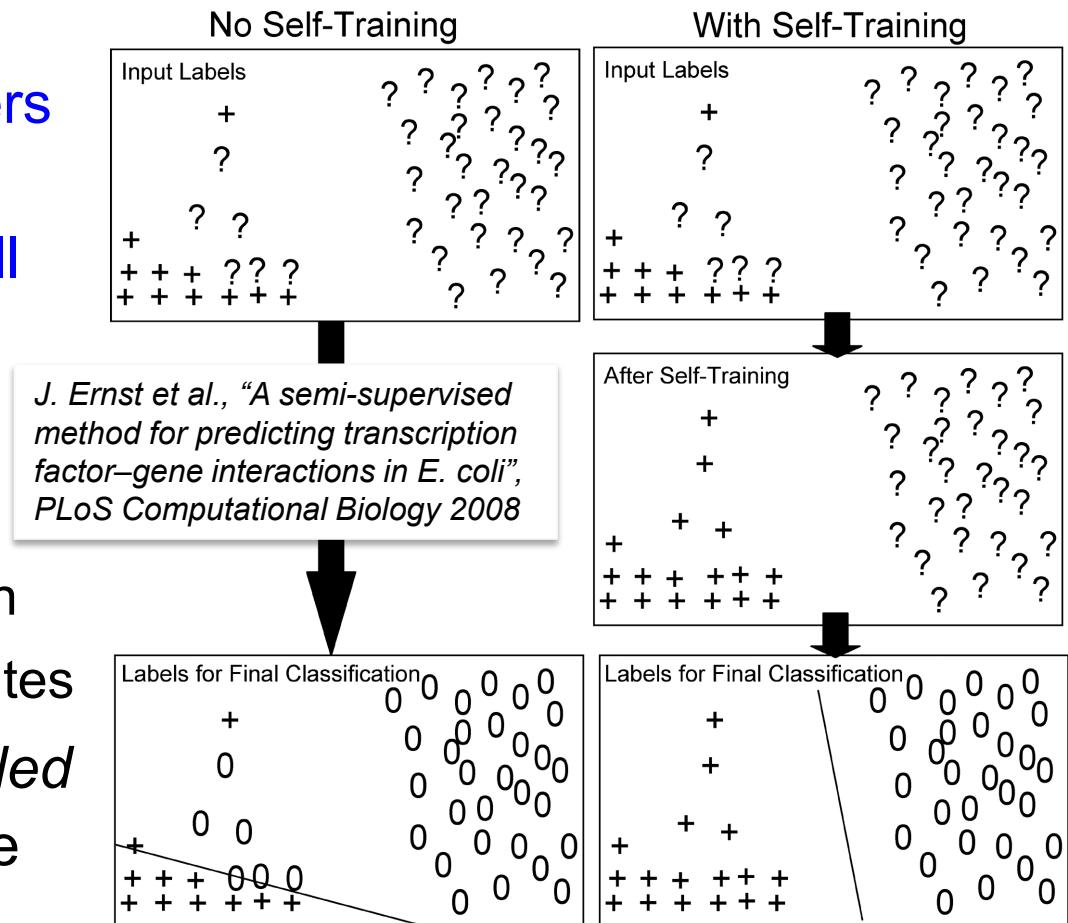
# Structured learning

- Predicting arbitrarily shaped output rather than a single label
- Applications in predicting:
  - gene structure
  - secondary protein structure
  - drug activity
  - metabolic reaction
- Special case:  
*multi-label learning*,  
outputting several related labels, for example gene ontology (GO) annotations



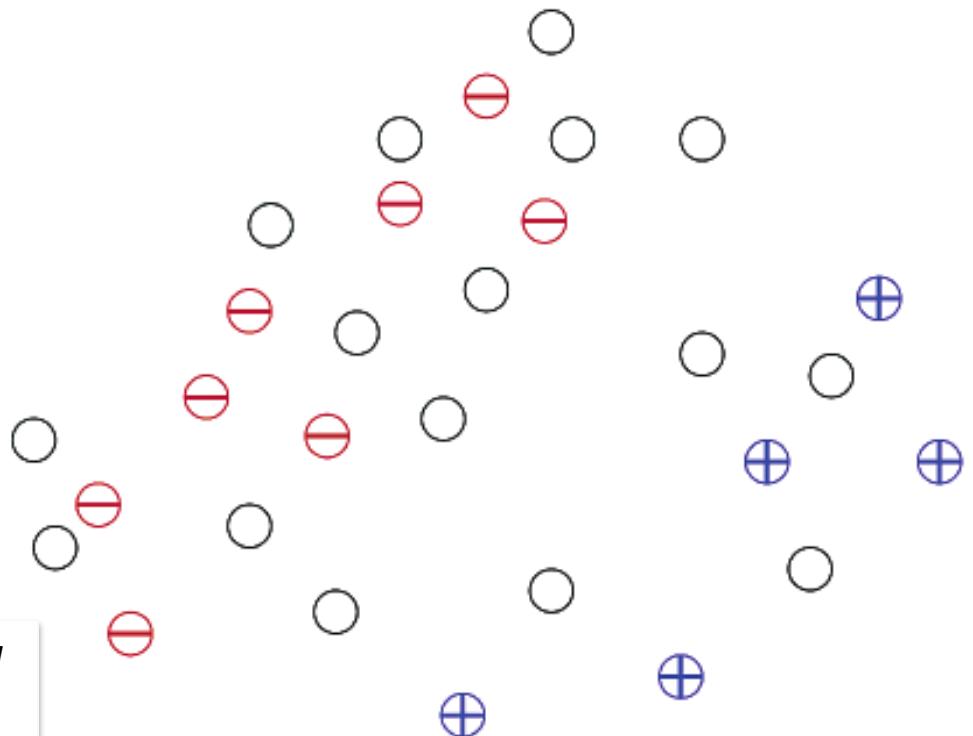
# Semi-supervised learning

- Used when large numbers of unlabeled objects are available besides a small set of labeled objects
- Applications in
  - clustering expression
  - predicting gene function
  - predicting TF binding sites
- Related: *positive unlabeled learning*, assuming some objects have a (single, positive) label and the remainder is unlabeled, e.g. for protein-protein and genetic interaction data



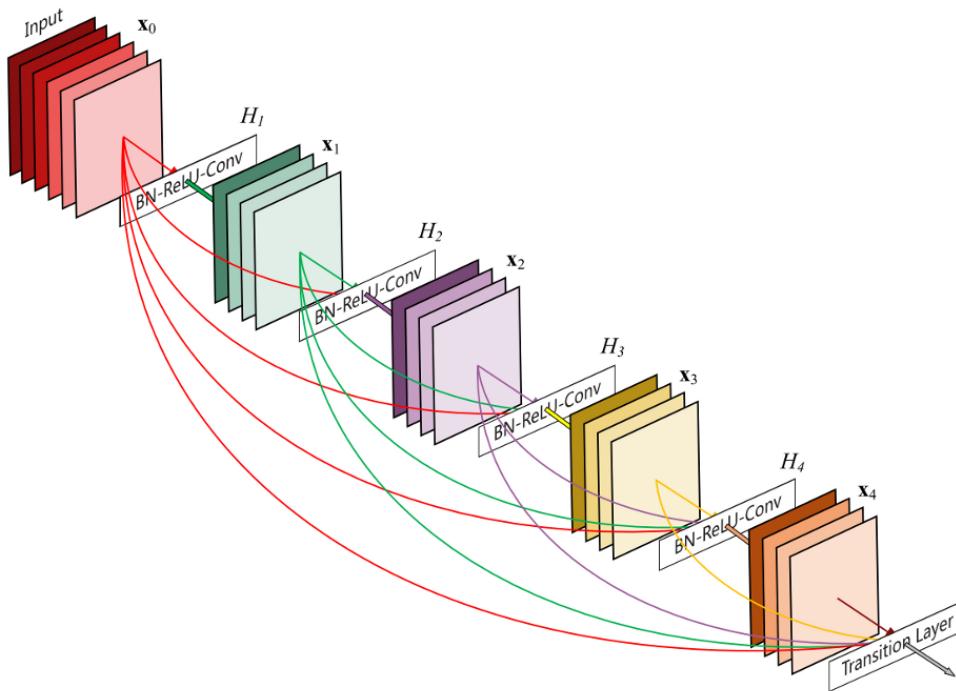
# Active learning

- Using a classifier to decide which unlabeled object should be labeled next to best improve that classifier
- Applications:
  - diagnosis
  - drug discovery
  - predicting protein interactions, transmembrane helices

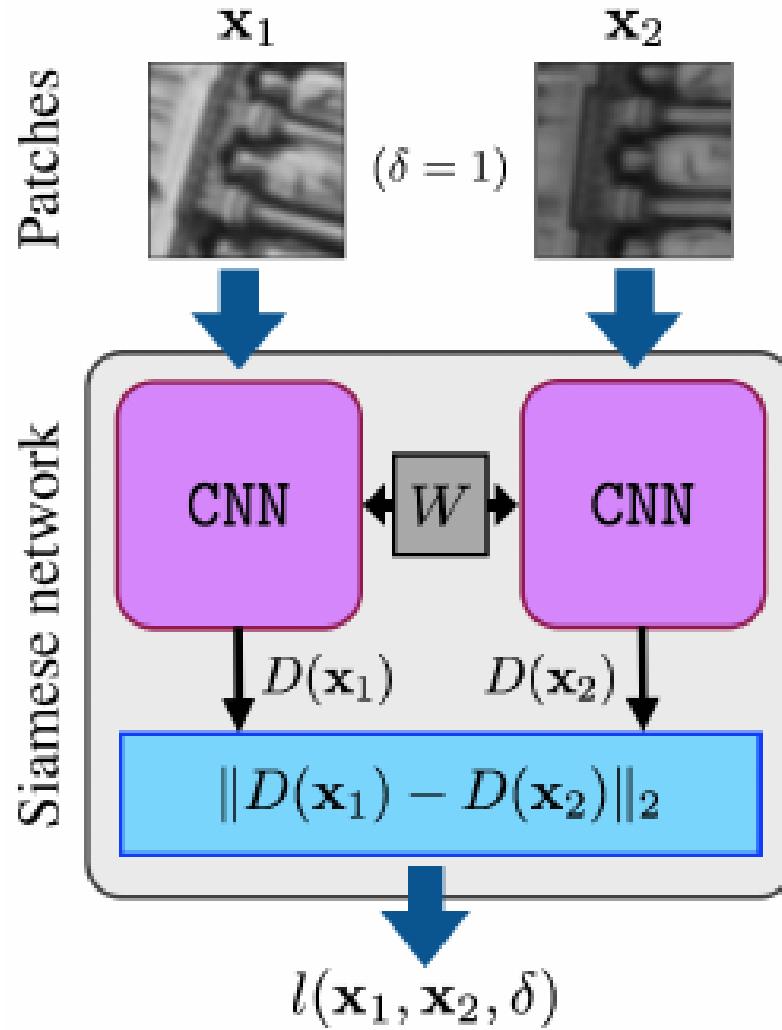


M.K. Warmuth et al., "Active learning with support vector machines in the drug discovery process", *Journal of Chemical Information and Computer Sciences* 2003

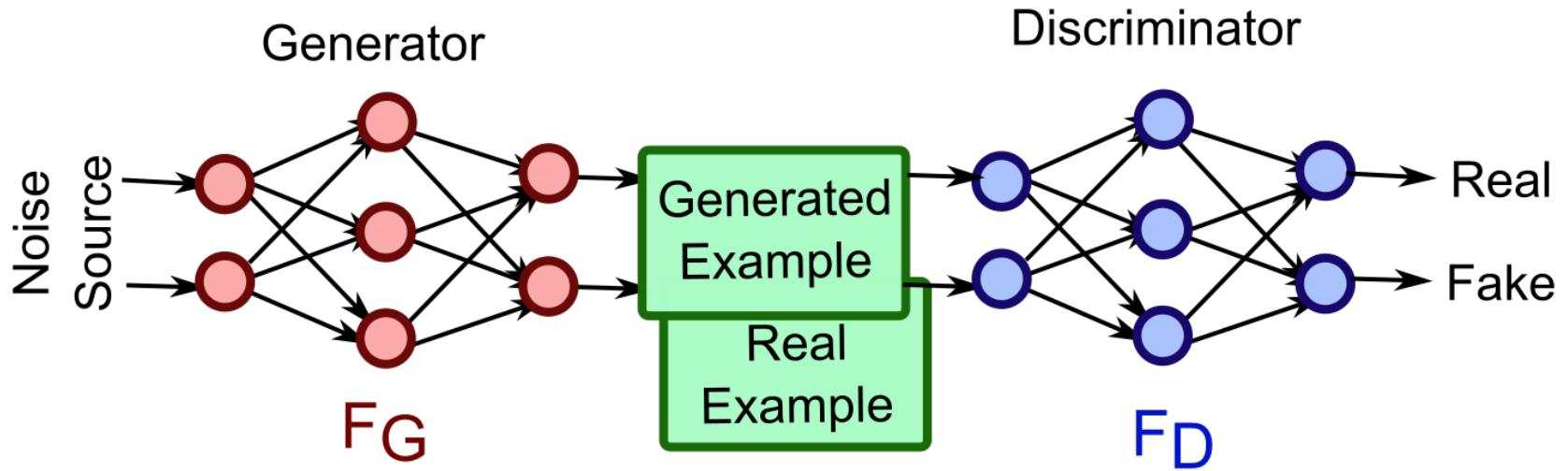
# and of course deep nets Residual Networks



# and of course deep nets Siamese Networks



# and of course deep nets generative adversarial network (GAN)



A mostly complete chart of  
**Neural Networks**

©2016 Fjodor van Veen - asimovinstitute.org

- Backfed Input Cell
- Input Cell
- △ Noisy Input Cell
- Hidden Cell
- Probabilistic Hidden Cell
- △ Spiking Hidden Cell
- Output Cell
- Match Input Output Cell
- Recurrent Cell
- Memory Cell
- △ Different Memory Cell
- Kernel
- Convolution or Pool

