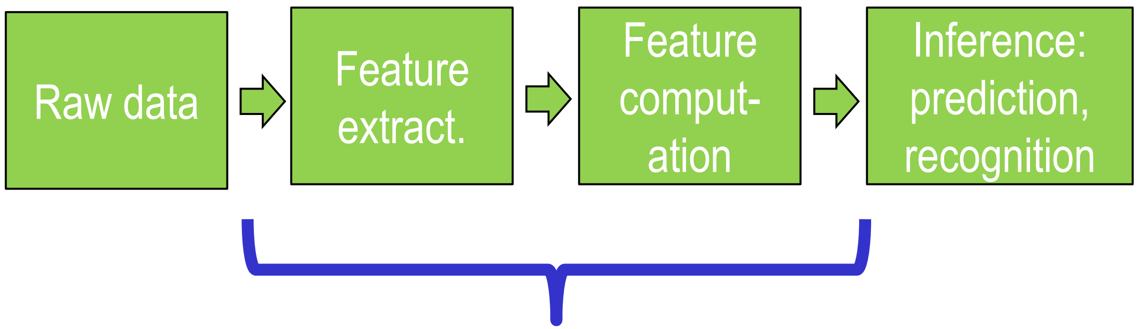
Neural Networks and Deep Learning

Example Learning Problem

Example Learning Problem

Celebrity Faces in the Wild

Machine Learning Pipeline

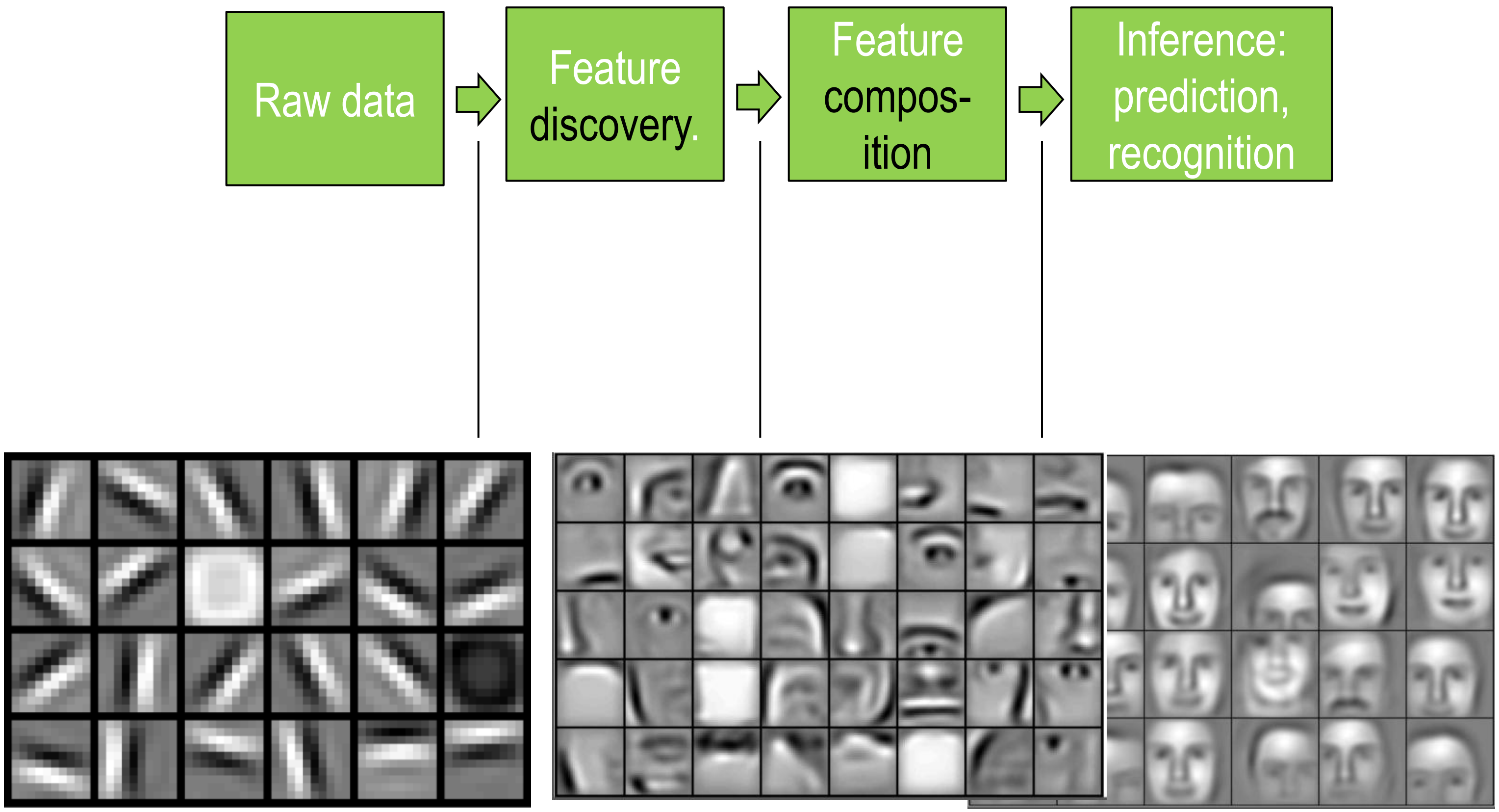


Features

- critical for accuracy
- traditionally hand-crafted

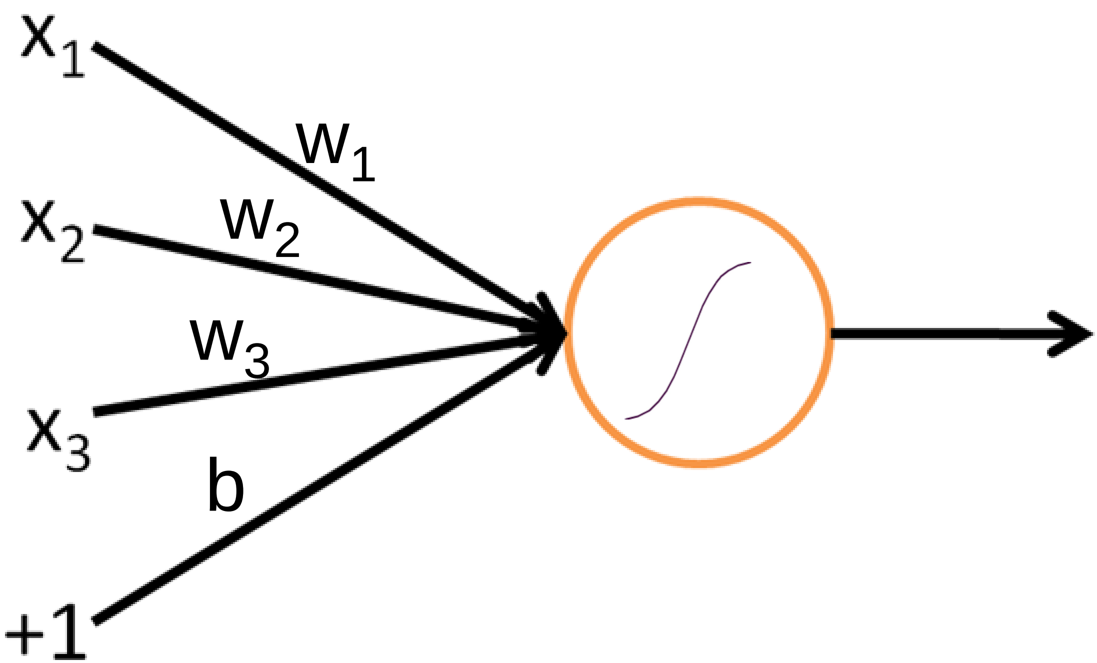
Instead of designing features, try to design feature detectors

Machine Learning Pipeline



Low level features Mid-level features Face-like features

Logistic Regression unit

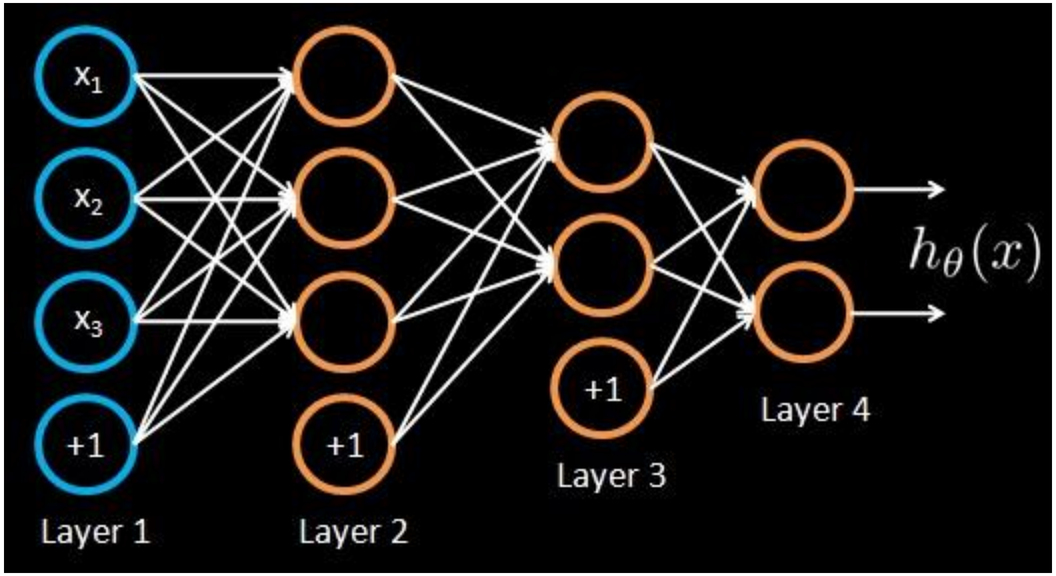


$$h(\mathbf{x};\, \mathbf{w}, b)$$

$$= 1/\, e^{-(\mathbf{w}^T\mathbf{x} + b)}$$

Objective:
determine parameters ***w***,*b*

Training a neural network



Given training set (x_1, t_1), (x_2, t_2), (x_3, t_3), ….

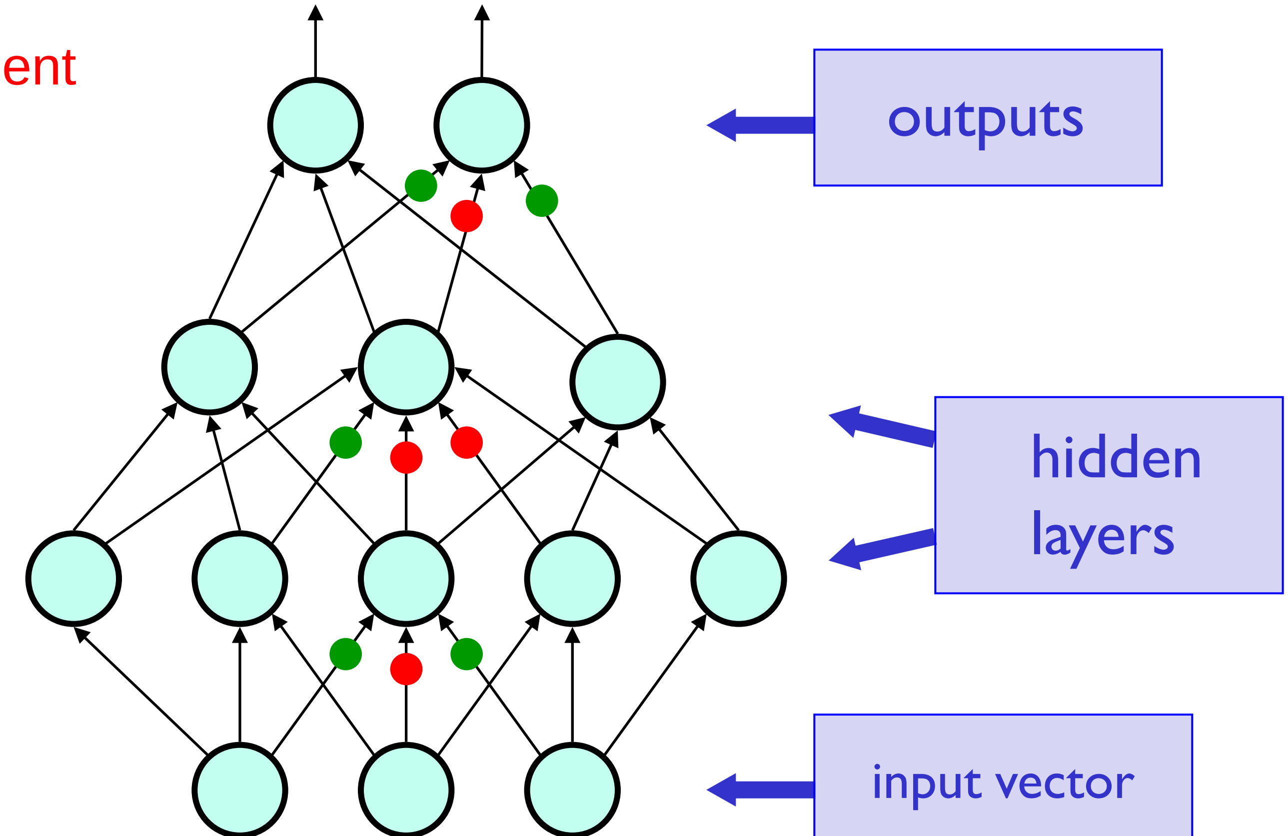
minimize error = $h(x_i; \mathbf{w}, b) - t_i$ by adjusting parameters ($\mathbf{w}$,b) over all nodes

Use gradient descent : “Backpropagation”: → local optima

MLP with Back-propagation

Compare outputs with correct answer to get **error** vector

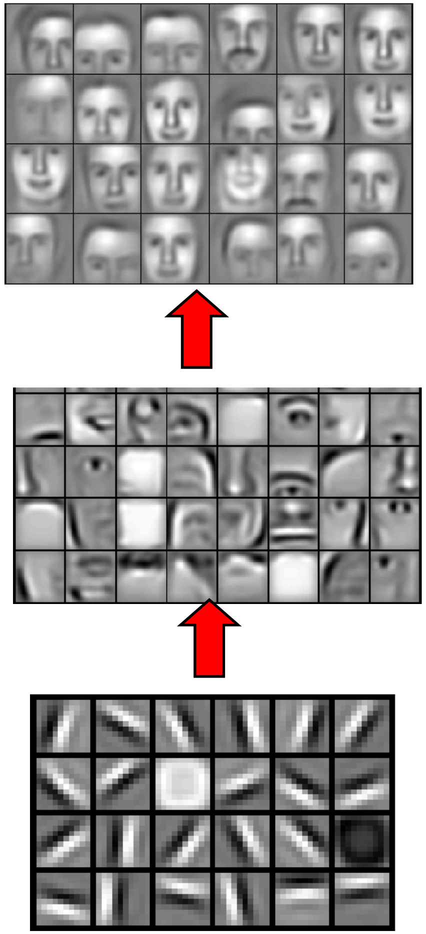
Back-propagate error to get gradient for parameter updating



[Hinton 09] DBN tutorial

Why “Deep”?

- Brains are very deep
- Humans organize their ideas hierarchically, through composition of simpler ideas
- Insufficiently deep architectures can be exponentially inefficient
  - functions computable with a polynomial-size circuit of depth *k* may require exponential size at depth *k-1* [Hastad 86].
- Deep architectures help share features



Why learn features??

- In the brain, very few filters are hard-coded
- irreversible damage produced in kittens by early visual deprivation [Hubel Wiesel 63]
- Avoids different feature extraction schemes for different kinds of input data
- Hypothesis :

Good Reconstruction ➔ Good Recognition

Drawbacks of Back-propagation

- Purely discriminative

  Get all the information from the labels

  And the labels don't give so much of information

  Need a substantial amount of labeled data

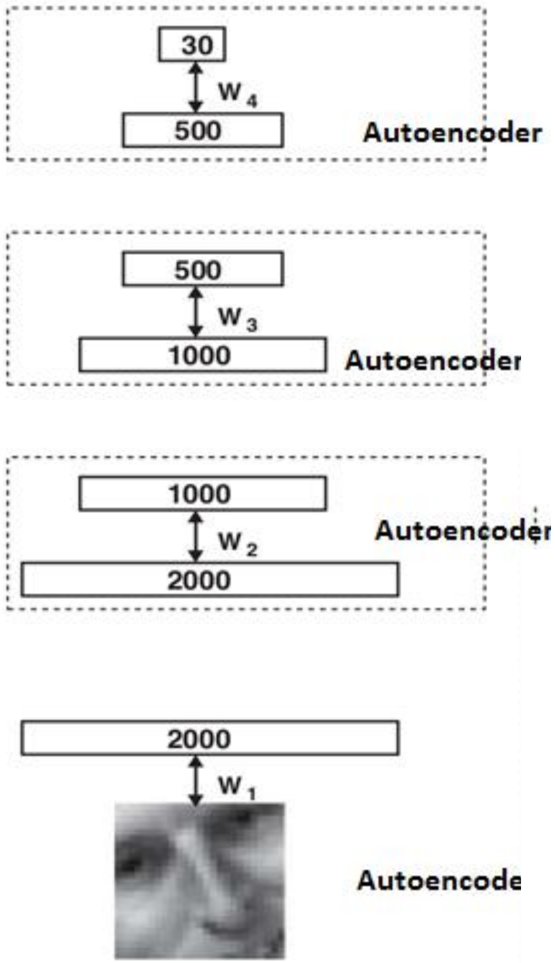
- Gradient descent with random initialization leads to poor local minima

Deep Belief Networks

- Pre-train network from input-data alone (generative step)
- Use weights of pre-trained network as the initial point for traditional back-propagation
  - Leads to quicker convergence
- Pre-training is fast; fine-tuning can be slow

Deep Autoencoder

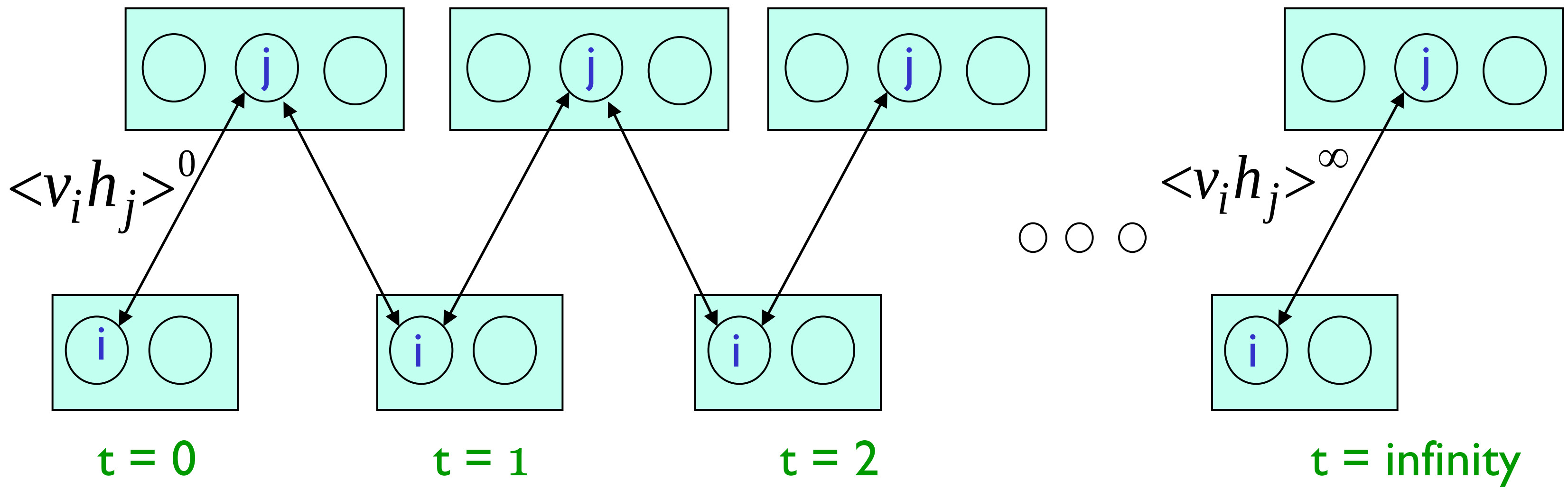
2-layer Auto- Encoders (coarse)



Pretraining

Auto-Encoders → stack

Pre-Training: Maximum likelihood learning



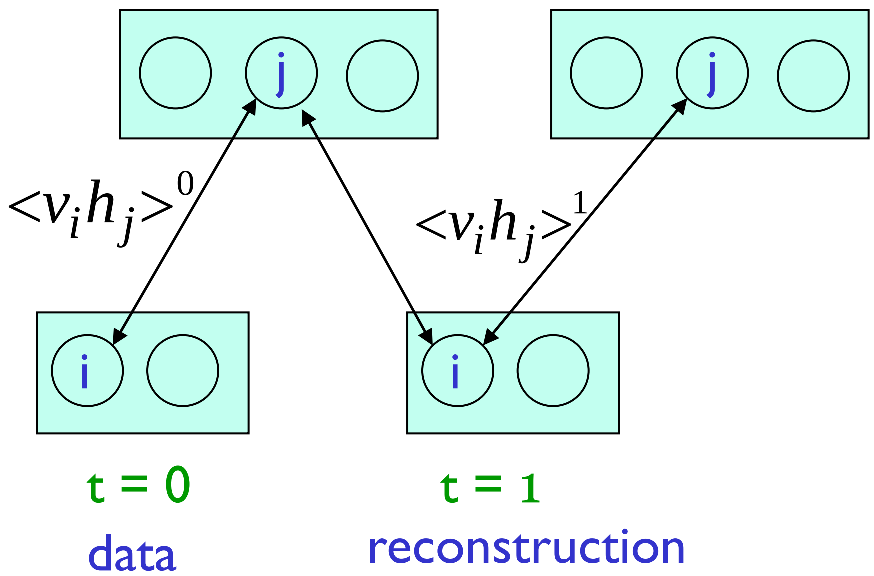
Start with a training vector on the visible units.

Then alternate between updating all the hidden units in parallel and updating all the visible units in parallel.

$$\frac{\partial \log p(v)}{\partial w_{ij}} = <v_i h_j>^0 - <v_i h_j>^\infty$$

[Hinton 09] DBN tutorial

Pre-Training: Maximum likelihood learning



For each training vector

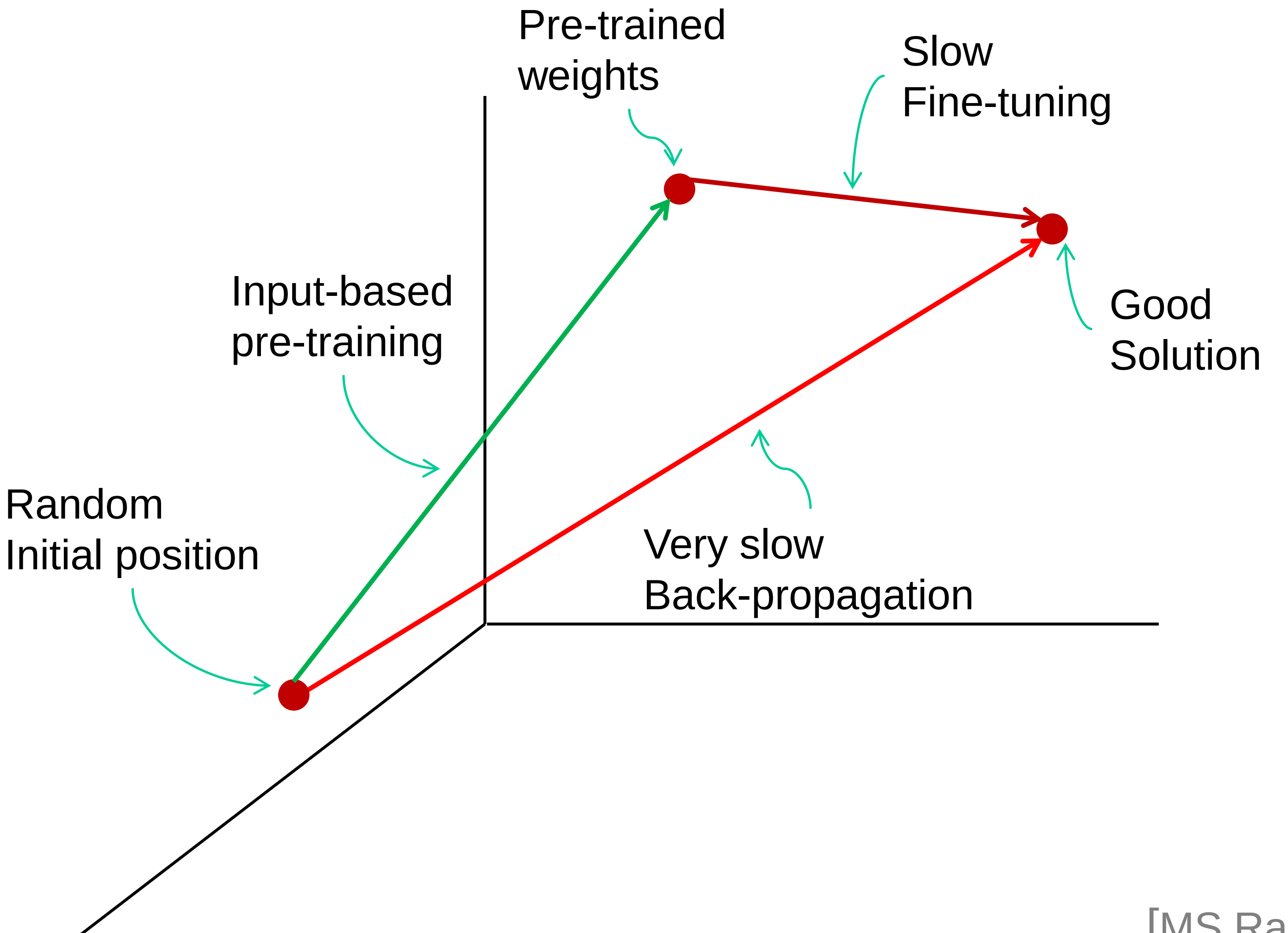
Update all hidden units in parallel

Update the all the visible units in parallel to get a "reconstruction".

Update the hidden units again.

$$\Delta w_{ij} = \varepsilon \left(<v_i h_j>^0 - <v_i h_j>^1 \right)$$

Deep Belief Networks



[MS Ram]

Deep Belief Networks

- Pre-train network from input-data alone (generative step)
- Use weights of pre-trained network as the initial point for traditional back-propagation
  - Leads to quicker convergence
- Pre-training is fast; fine-tuning can be slow

Searching in parameter space

One layer :1000 input + 1000 hidden

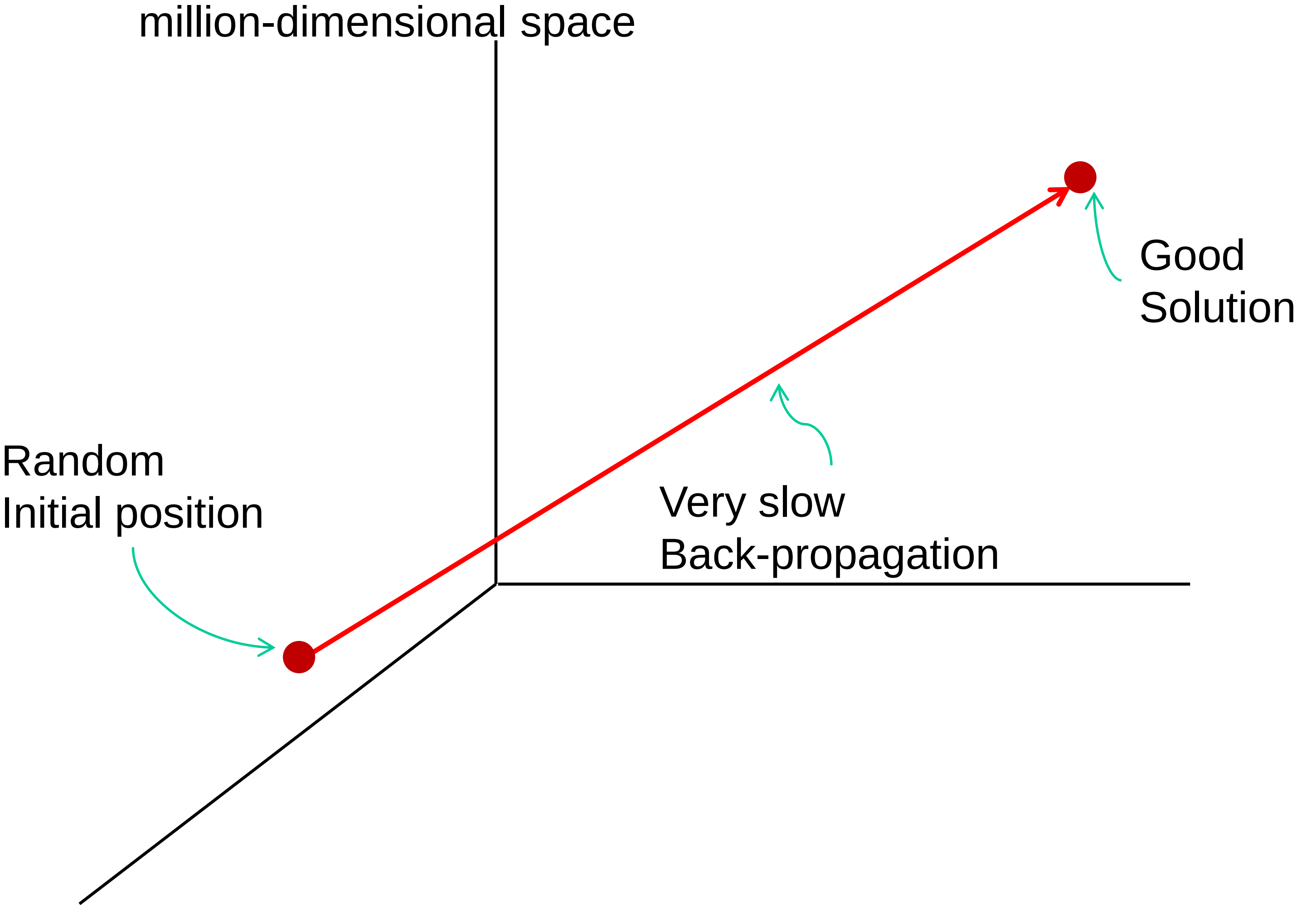
≈ 1 **million weights**

→ million-dimensional optimization

need to find global (or at least good) optimum from random initialization)

Impossibly slow for Gradient descent

Deep Belief Networks



Searching in parameter space

One layer :1000 input + 1000 hidden

$\approx$ 1 **million weights**

→ million-dimensional optimization

Impossibly slow for Gradient descent to find global optimum from random initialization

- Added complications:
  - gradient magnitude vanishingly small in lower parts of network
  - deep networks tend to have more local minima than shallow networks

In practice : MLP vs DBN (MNIST)

MLP (1 Hidden Layer)

- 1 hour: 2.18%
- 14 hours: 1.65%

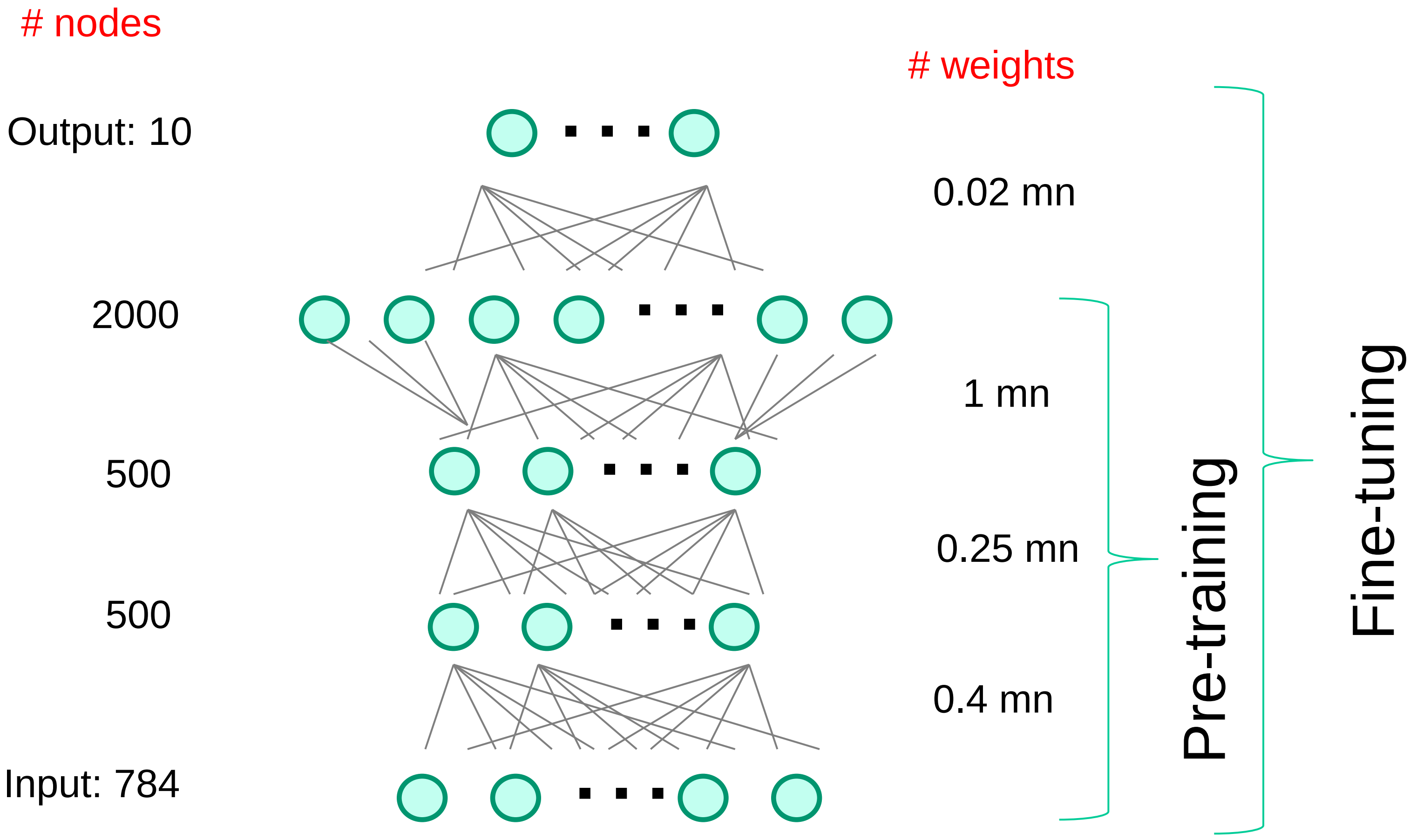
DBN

- 1 hour: 1.65%
- 14 hours: 1.10%
- 21 hours: 0.97%

Intel QuadCore 2.83GHz, 4GB RAM

[MS Ram]

Deep network architecture (MNIST)



Designing DBNs

Relative importance of

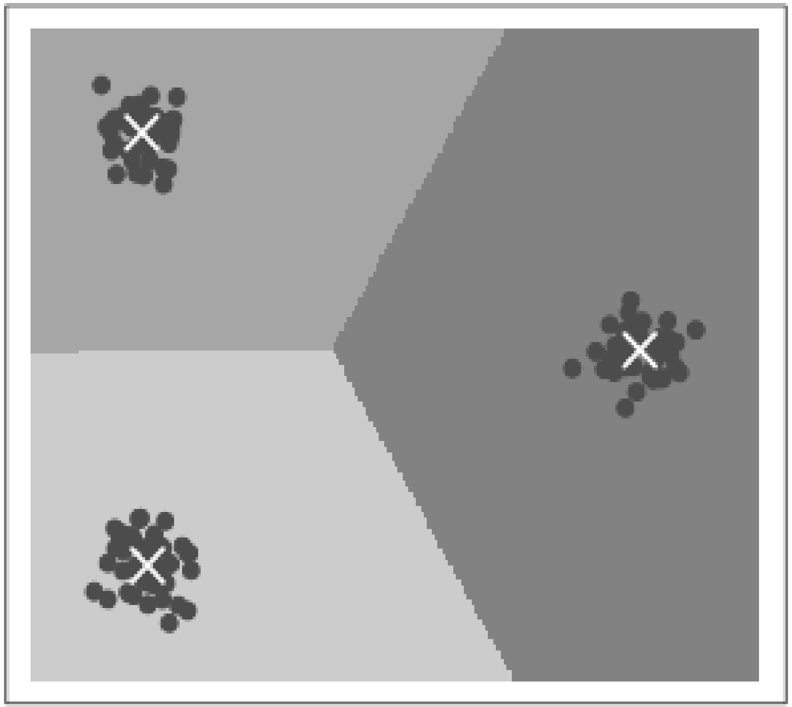
- Depth of network :
  - Seems quite important
- Layer-wise pre-training
  - Counter-example: MNIST: 6-layer MLP: 784-2500-2000-1500-1000-500-10 (on GPU, w elastic distortions)
    - → Error Rate: 0.35% [Ciresan et al 2010]
  - *"No fashionable unsupervised pre-training is necessary! "*
    - Jürgen Schmidhuber
- Amount of labeled training data
  - Affine and Elastic distortions

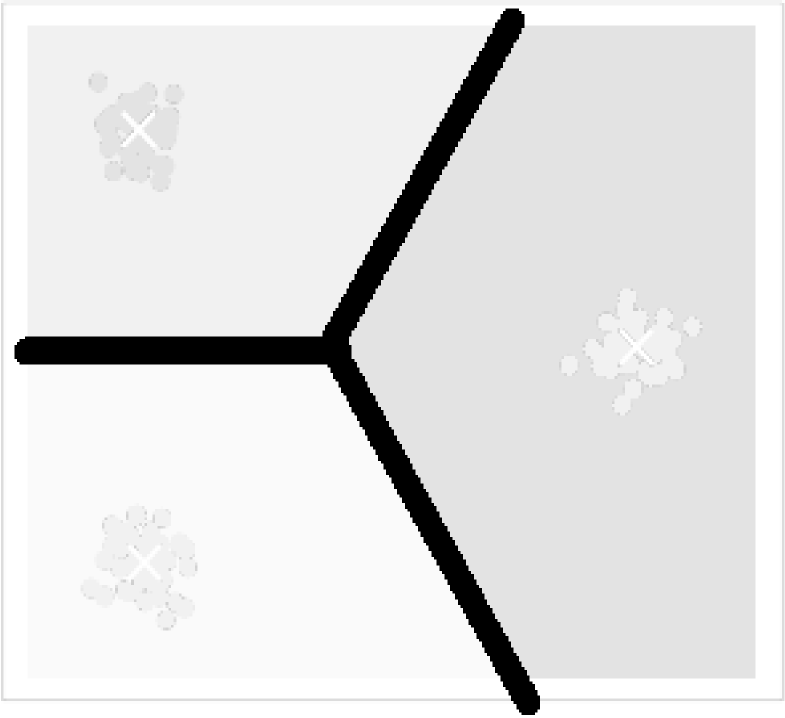
Main benefit: DBNs work w less training data

Kernel Methods

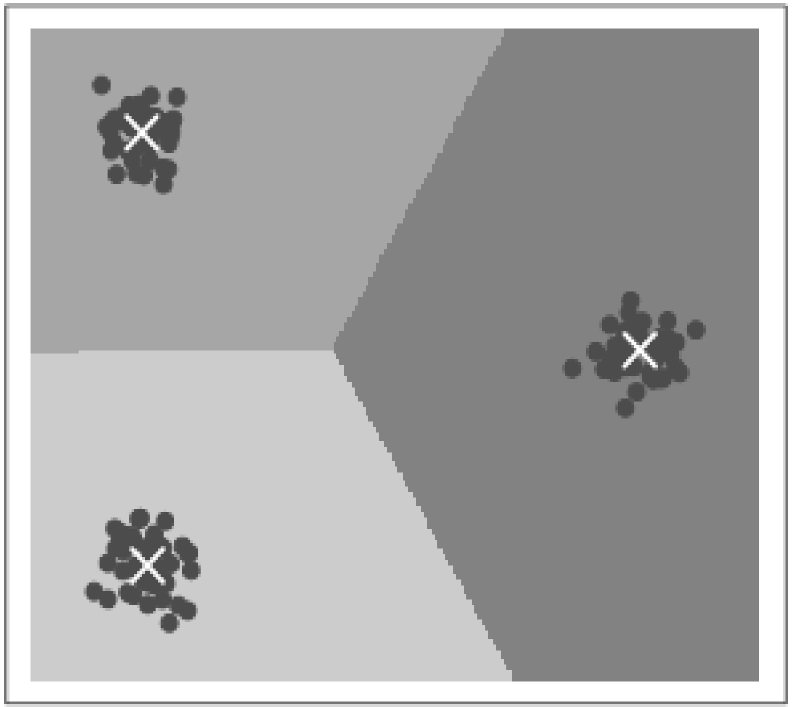
Bishop, Ch.6
R & N ch 18.6

Parametric: Discriminative :





Parametric: Generative



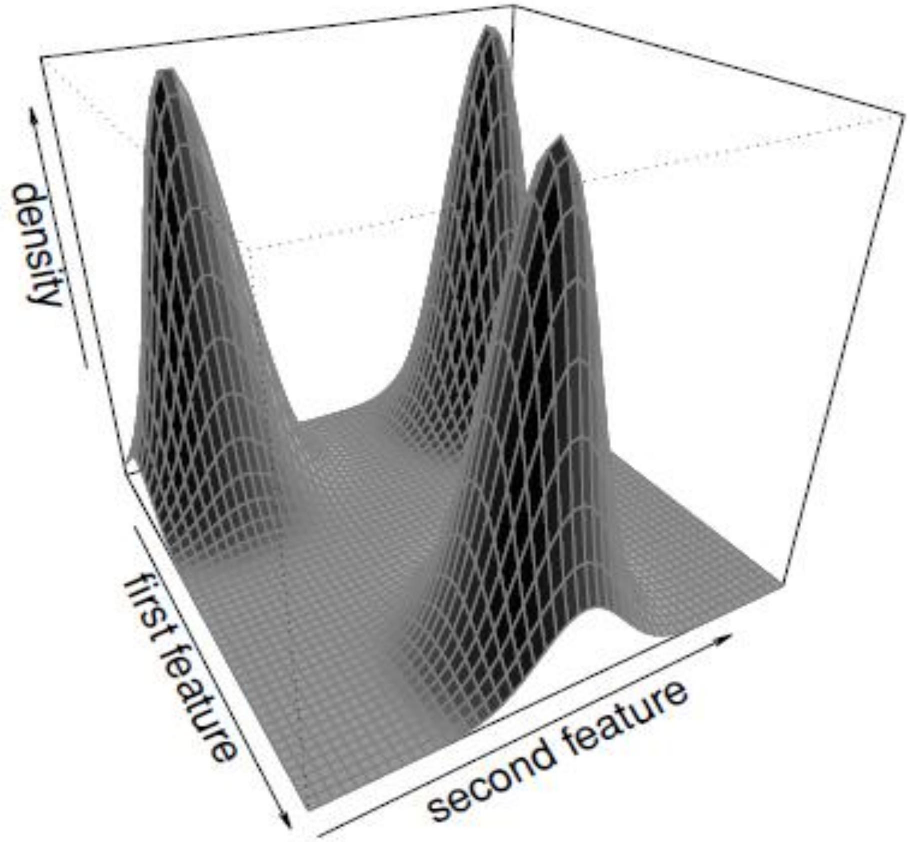
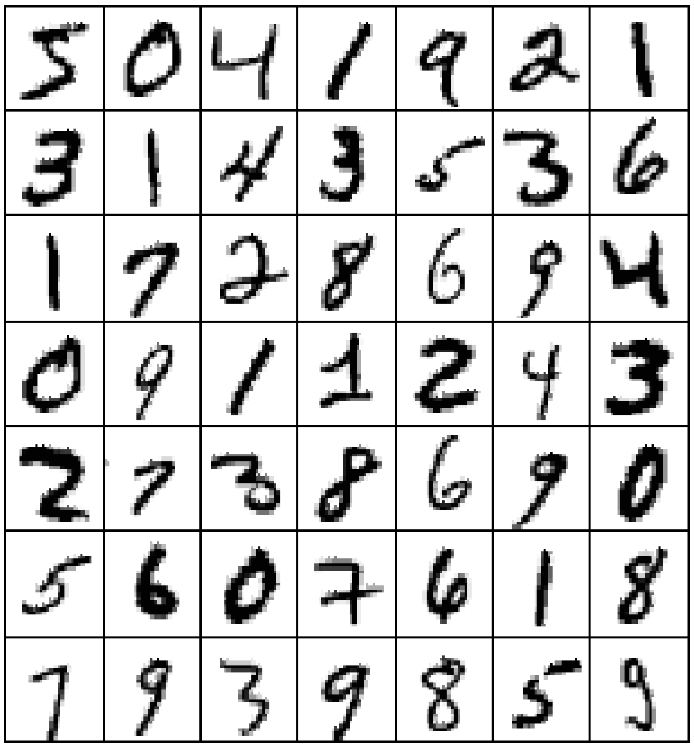


image from [Herbrich 2002]

Parametric vs Memory models

- **Parametric models:**
  - learn model for data: parameter vector **w** */* posterior distribution p(**w** | $t_1..t_N$)
  - discard training set **t**
  - e.g. linear classifiers (perceptron)
- **Non-Parametric :**
  - models on data e.g. k-NN
  - memory-based: some or all of the training data is saved
  - SVM: save a set of “support vectors”

MNIST dataset



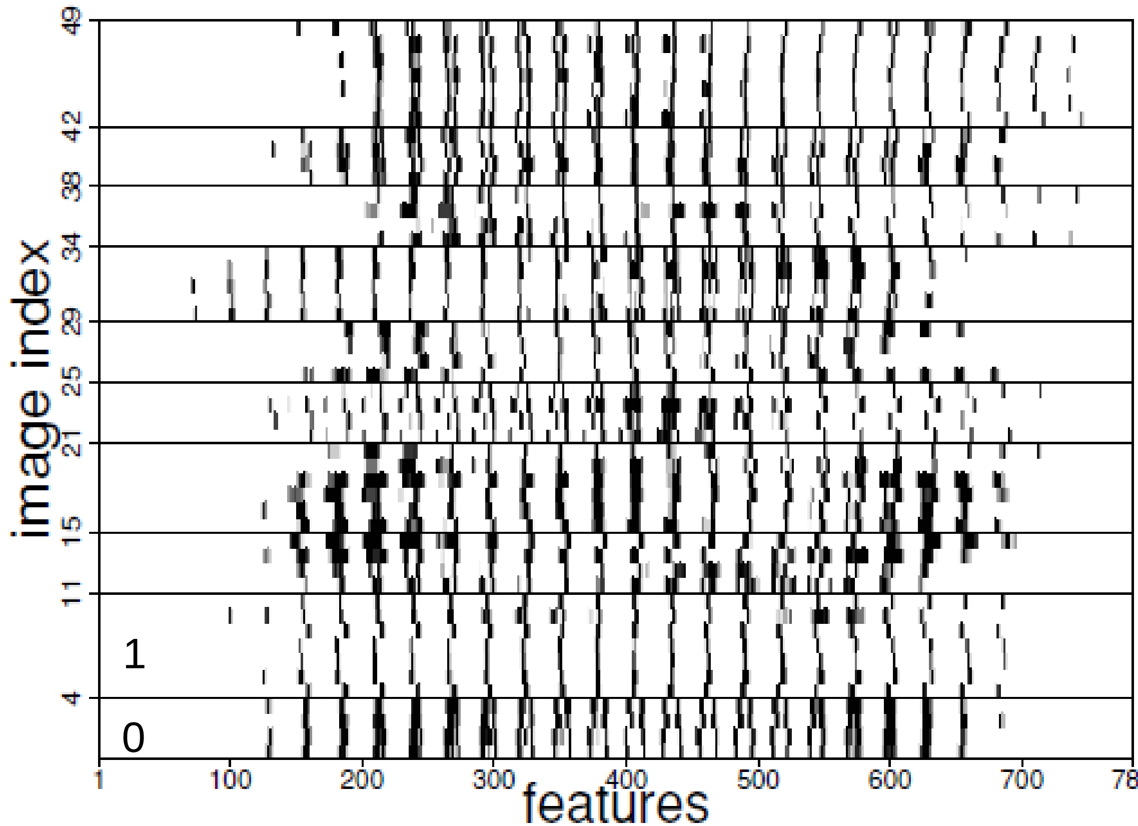
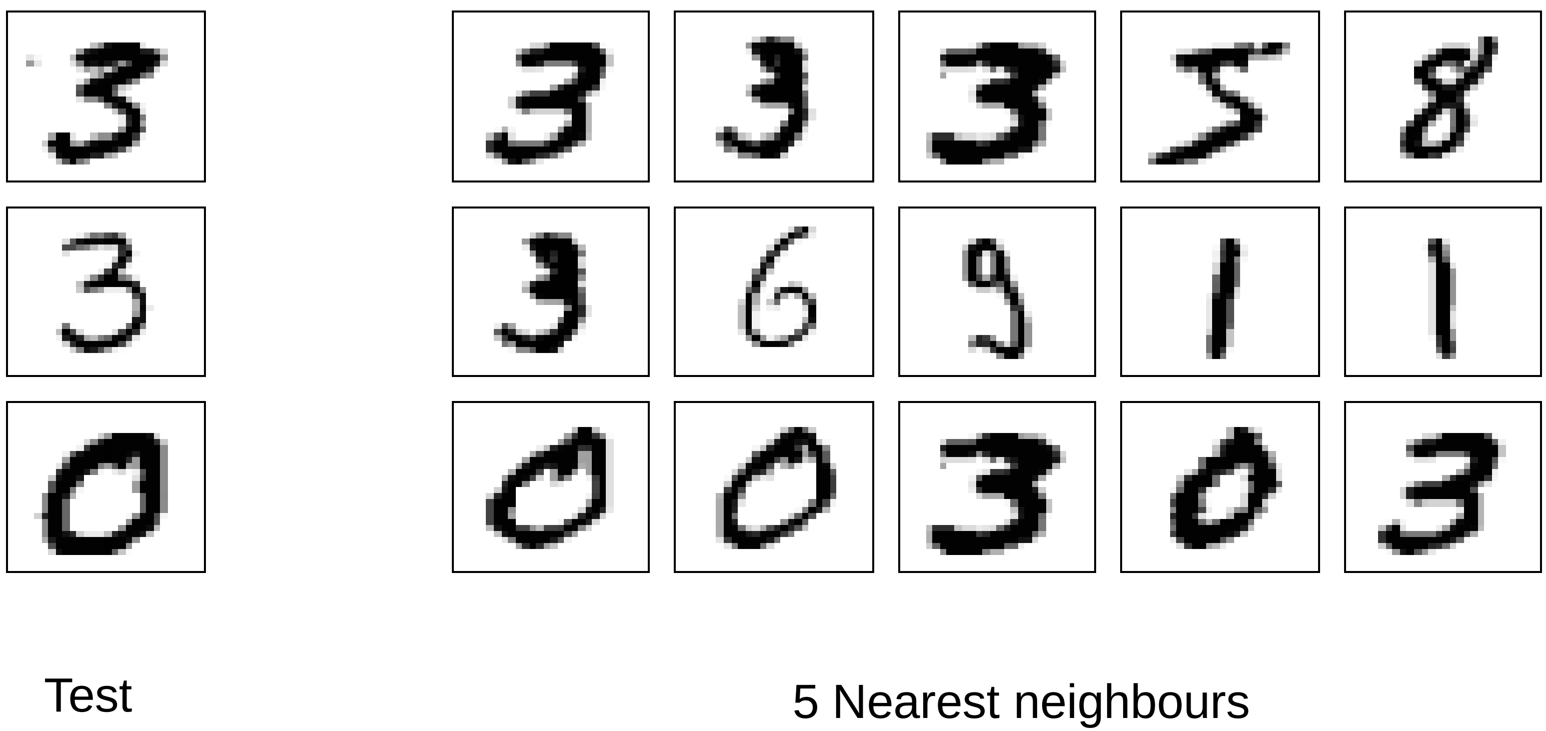


image from [Herbrich 2002]

k-NN



Kernel methods

- feature space mapping $\varphi(x)$:

$$k(x, x') = \varphi(x)^T \varphi(x')$$

- symmetric: $k(x, x') = k(x', x)$

- linear kernel: $\varphi(x) = x$
- stationary kernel: $k(x, x') = k(x - x')$ [stationary under translation]
- homogeneous kernel:
  $k(x, x') = k(|x - x'|)$ (e.g. RBF)

Support Vector Machines

- Main idea:
  - linear classifier, but in φ(x) kernel space.
  - criterion for decision: max-margin
- Algorithm
  - user specifies kernel function
  - learn weights for instances
  - no actual computation in high-dim space
- Classification
  - average of the instance labels, weighted by a) proximity b) instance weight.

Example: XOR

* X-OR problem

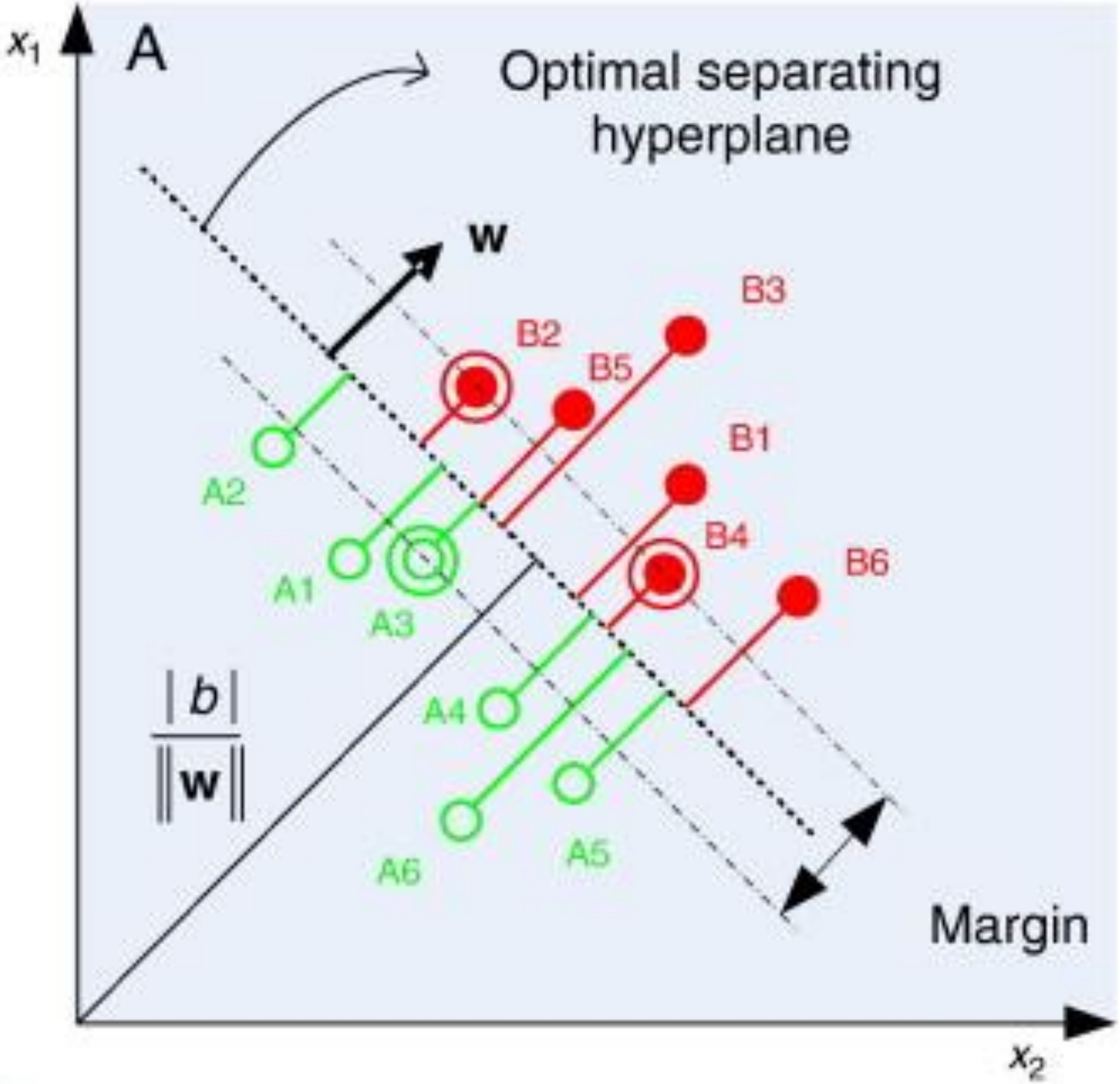
x_2 \ x_1	0	1
0	0	1
1	1	0

$$\varphi(x_1,x_2) = \{x_1,\ x_2,\ x_1x_2\}$$

Better:

$$\varphi(x_1,x_2) = \{1,\ \sqrt{2}x_1,\ \sqrt{2}x_2,\ x_1^2,\ \sqrt{2}\,x_1x_2,\ x_2^2\}$$

Margin maximization



Decision hyperplane:

$$\mathbf{w}^T \mathbf{x} + b = 0$$

If $t_i = \{+1,-1\}$, margin m=

$$1/||\mathbf{w}|| \ \min_i t_i (\mathbf{w}^T \mathbf{x}_i + b)$$

w must satisfy the constraint that for all data $(\mathbf{x}_i, t_i)$:

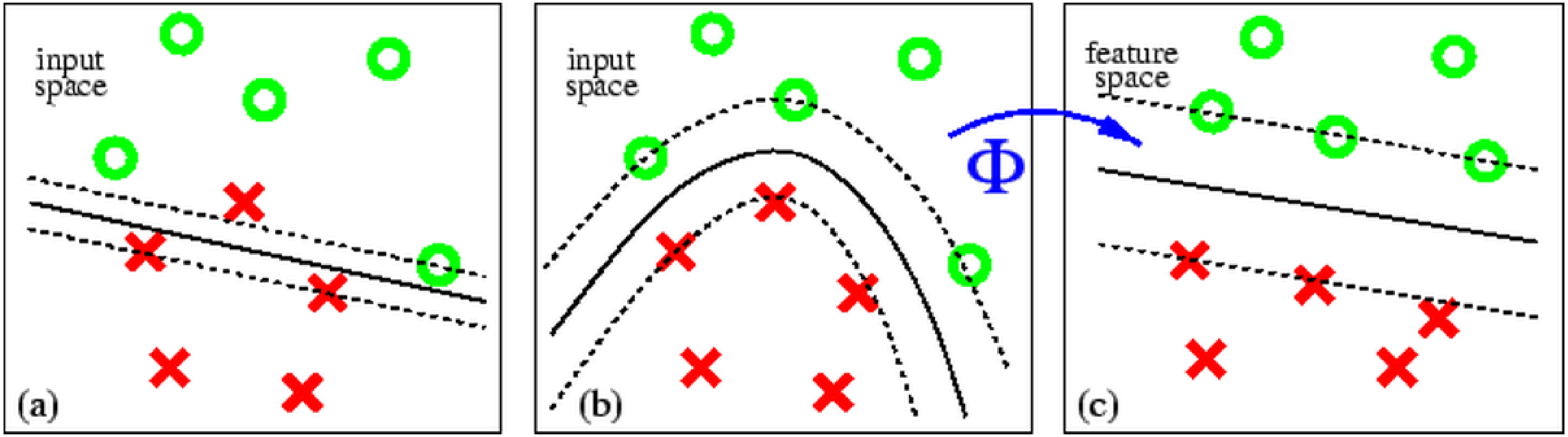
$$t_i (\mathbf{w}^T \mathbf{x}_i \ + b) > m$$

Margin is maximized when $1/||\mathbf{w}||$ is maximum, ie. minimize $||\mathbf{w}||$

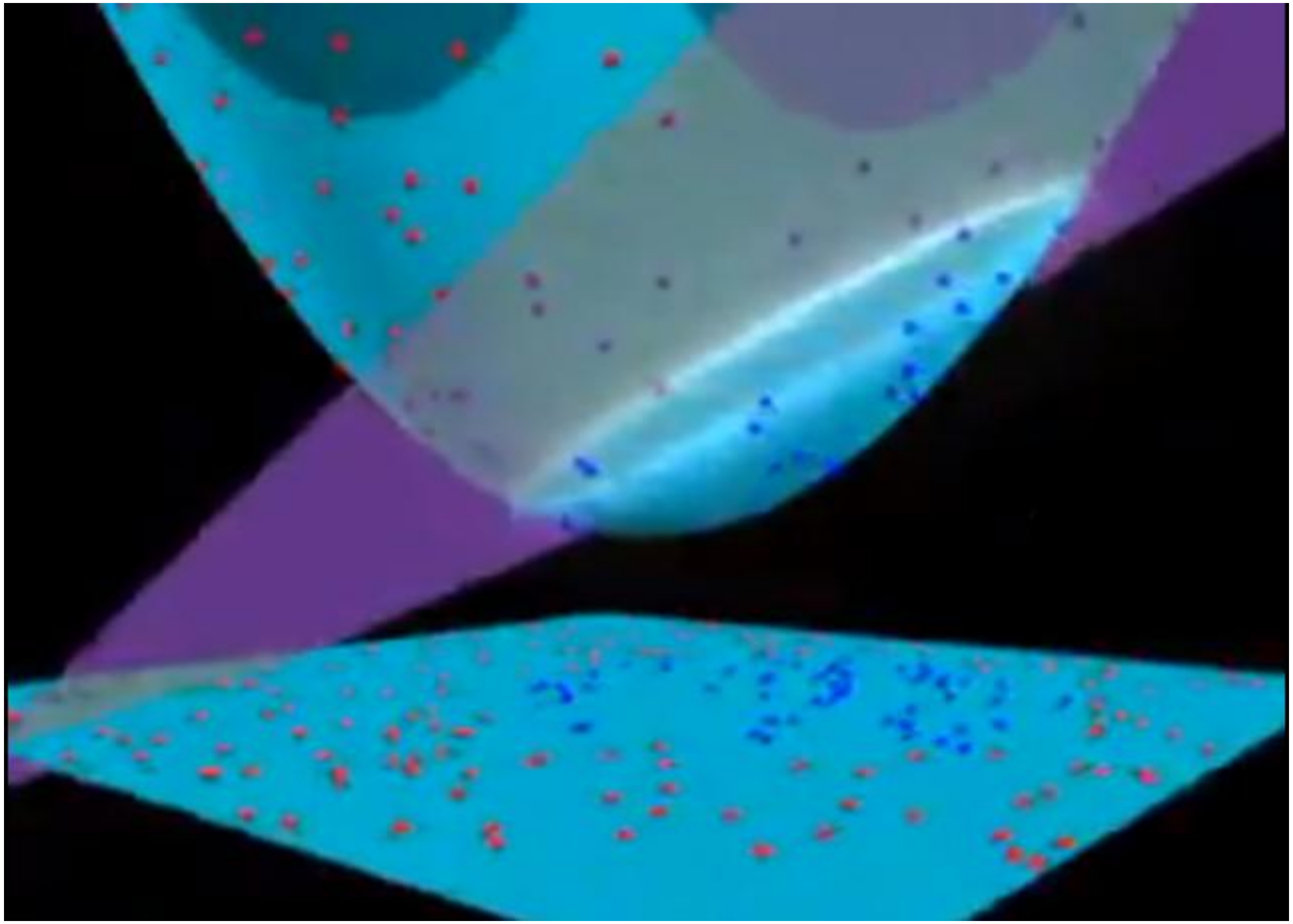
Support Vector Machines

- Main idea:
  - linear classifier, but in φ(x) kernel space.
  - criterion for decision: max-margin
- Algorithm
  - user specifies kernel function
  - learn weights for instances
  - no actual computation in high-dim space
- Classification
  - average of the instance labels, weighted by a) proximity b) instance weight.

Kernel trick



Demo



Demo by Udi Aharoni http://www.youtube.com/watch?v=3liCbRZPrZA

Support Vector Machines

- Main idea:
  - linear classifier, but in φ(x) kernel space.
  - criterion for decision: max-margin
- Algorithm
  - user specifies kernel function
  - learn weights for instances via convex optimization
  - no actual computation in high-dim space
- Classification
  - average of the instance labels, weighted by a) proximity b) instance weight.

Kernel trick

- Linear classifier is in in high-dimensional $\phi(x)$ space
- However, no computation directly on $\phi(x)$; compute only kernel = $\phi(x)^T\phi(x)$

  e.g. for

$$\phi(x_1,x_2) = \{1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, \sqrt{2}\, x_1 x_2, x_2^2\}$$

$$k(x,x') = \phi(x)^T\phi(\underline{x}')$$

$$= ((1, x_1, x_2)(1, x'_1, x'_2)^T)^2 = \langle \mathbf{x},\mathbf{x}' \rangle^2$$

- Efficient only if scalar product can be efficiently computed. Holds for:
- k(x,x') : continuous, symmetric and positive definite

Dual Representations

- Scalar product representations arise naturally in many classes of problems
- e.g. Linear regression

$$J(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\left\{\mathbf{w}^{\mathrm{T}}\phi(\mathbf{x}_n) - t_n\right\}^2 + \frac{\lambda}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w}$$

- Setting gradient to zero:

$$\mathbf{w} = -\frac{1}{\lambda}\sum_{n=1}^{N}\left\{\mathbf{w}^{\mathrm{T}}\phi(\mathbf{x}_n) - t_n\right\}\phi(\mathbf{x}_n) = \sum_{n=1}^{N} a_n\phi(\mathbf{x}_n) = \mathbf{\Phi}^{\mathrm{T}}\mathbf{a}$$

where Φ^T = matrix of $\phi(x_n)$, and

$$a_n = -\frac{1}{\lambda}\left\{\mathbf{w}^{\mathrm{T}}\phi(\mathbf{x}_n) - t_n\right\}$$

Dual Representations

- Instead of parameter space **w**, use parameter space **a**
- Writing w = $\mathbf{\Phi}^{\mathrm{T}}$a into *J(w)*:

$$J(\mathrm{a}) = \frac{1}{2}\mathrm{a}^{\mathrm{T}}\mathbf{\Phi}\mathbf{\Phi}^{\mathrm{T}}\mathbf{\Phi}\mathbf{\Phi}^{\mathrm{T}}\mathrm{a} - \mathrm{a}^{\mathrm{T}}\mathbf{\Phi}\mathbf{\Phi}^{\mathrm{T}}\mathbf{t} + \frac{1}{2}\mathbf{t}^{\mathrm{T}}\mathbf{t} + \frac{\lambda}{2}\mathrm{a}^{\mathrm{T}}\mathbf{\Phi}\mathbf{\Phi}^{\mathrm{T}}\mathrm{a}$$
$$= \frac{1}{2}\mathrm{a}^{\mathrm{T}}\mathbf{K}\mathbf{K}\mathrm{a} - \mathrm{a}^{\mathrm{T}}\mathbf{K}\mathbf{t} + \frac{1}{2}\mathbf{t}^{\mathrm{T}}\mathbf{t} + \frac{\lambda}{2}\mathrm{a}^{\mathrm{T}}\mathbf{K}\mathrm{a}.$$

where gram matrix **K** = $\mathbf{\Phi}\mathbf{\Phi}^{\mathrm{T}}$, with $K_{nm} = \phi(x_n)^T\phi(x_m)$

solving for ***a*** by setting *dJ(**a**)/d**a** to zero:*

$$\mathbf{a} = (\mathbf{K} + \lambda\mathbf{I}_N)^{-1}\,\mathbf{t}.$$

Dual Representations

- Substituting back into original regression model:

$$y(\mathbf{x}) = \mathbf{w}^{\mathrm{T}}\phi(\mathbf{x}) = \mathbf{a}^{\mathrm{T}}\boldsymbol{\Phi}\phi(\mathbf{x}) = \mathbf{k}(\mathbf{x})^{\mathrm{T}}\left(\mathbf{K} + \lambda\mathbf{I}_N\right)^{-1}\mathbf{t}$$

- Thus, all computations are performed purely with the kernel

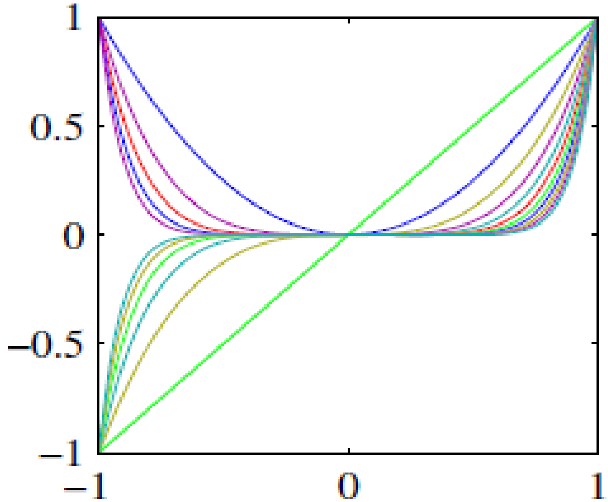
SVM

- Why so popular:
  - Very good classification performance, compares w best
  - Fast (convex) and scaleable learning
  - Fast inference (but slower training)
- Difficulties:
  - No model (discriminative; black-box)
  - Not as useful for discrete inputs.
  - Art: how to specify kernel function
  - Difficulties with multiple classes

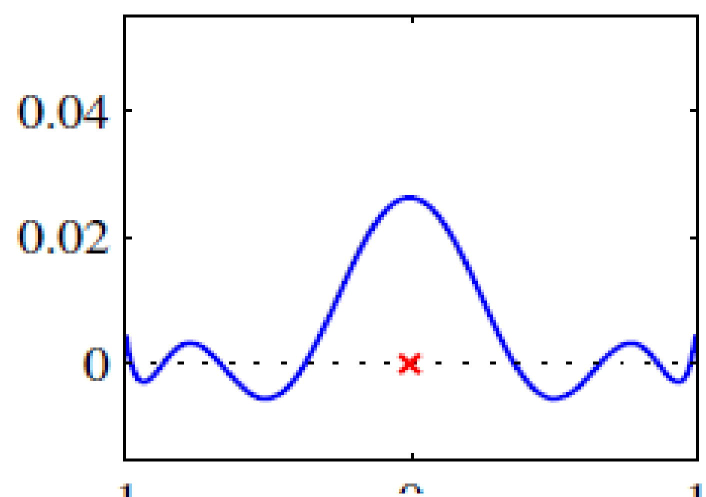
Choosing Kernels

Popular kernels that satisfy the Gram matrix positive-definiteness criterion inlcude:

- Linear kernels: k(**x,x'**) = <**x,x'**>
- Polynomials: k(**x,x'**) = <**x,x'**>^{n}
  - for n=2 (quadratic): $\phi(\mathbf{x}) = (x_1^2, \sqrt{2}x_1x_2, x_2^2)^{\mathrm{T}}$



Polynomial basis functions



k(x,x') = $\sum_{i=1}^{M} \phi_i(x)\phi_i(x')$ for x' = 0

Choosing Kernels

– Gaussian: $k(\mathbf{x}, \mathbf{x}') = \exp\left(-\|\mathbf{x} - \mathbf{x}'\|^2 / 2\sigma^2\right)$

– Radial basis functions

$$f(\mathbf{x}) = \sum_{n=1}^{N} w_n h(\|\mathbf{x} - \mathbf{x}_n\|)$$

– Sigmoid (logistic) function

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

Gaussian / Sigmoid Bases

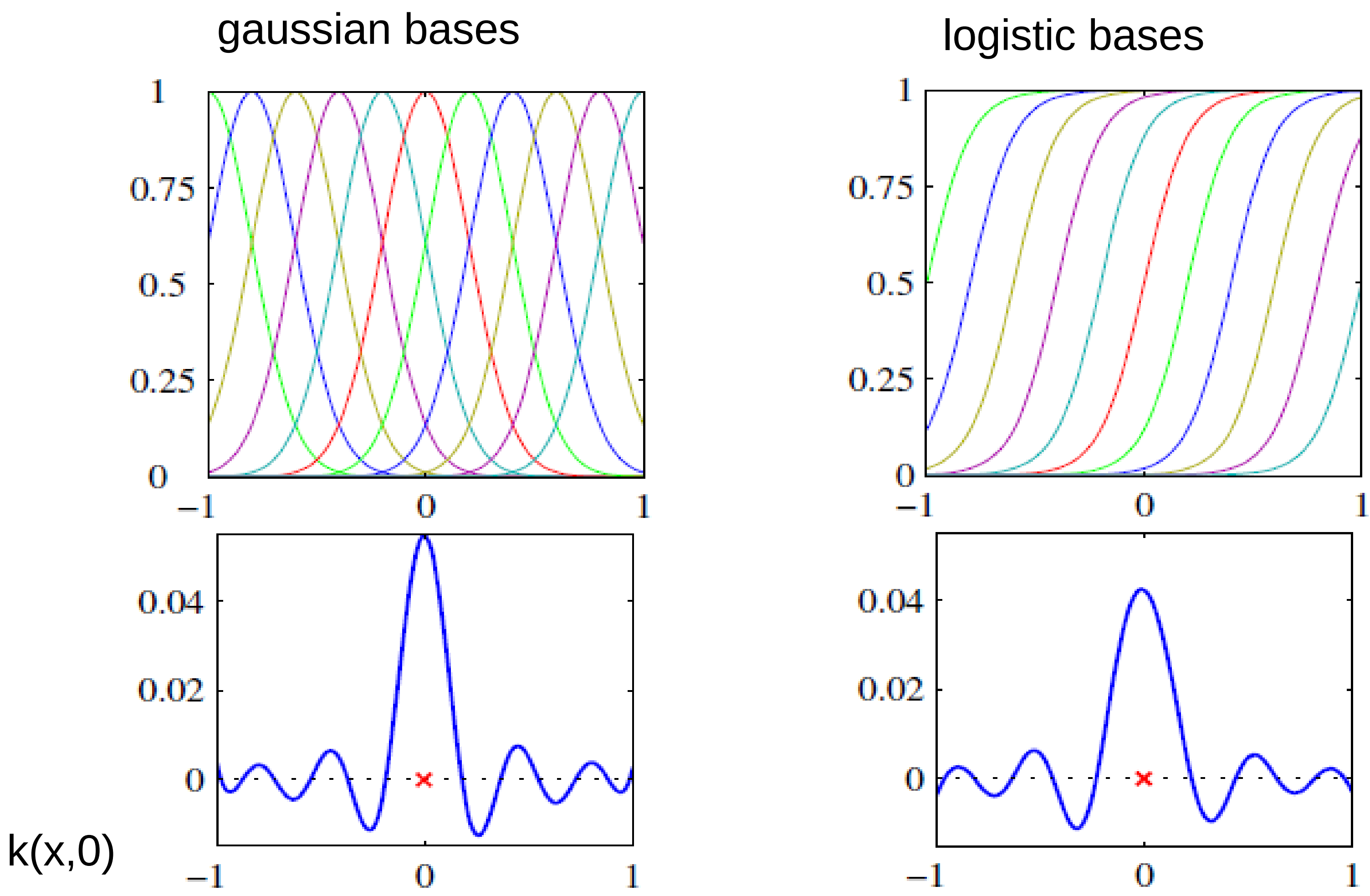


image from [Bishop 2005]

Constructing Kernels

If k1 and k2 are valid kernels, then so are:

$$
\begin{aligned}
k(\mathbf{x}, \mathbf{x}') &= c k_1(\mathbf{x}, \mathbf{x}') \\
k(\mathbf{x}, \mathbf{x}') &= f(\mathbf{x}) k_1(\mathbf{x}, \mathbf{x}') f(\mathbf{x}') \\
k(\mathbf{x}, \mathbf{x}') &= q\left(k_1(\mathbf{x}, \mathbf{x}')\right) \\
k(\mathbf{x}, \mathbf{x}') &= \exp\left(k_1(\mathbf{x}, \mathbf{x}')\right) \\
k(\mathbf{x}, \mathbf{x}') &= k_1(\mathbf{x}, \mathbf{x}') + k_2(\mathbf{x}, \mathbf{x}') \\
k(\mathbf{x}, \mathbf{x}') &= k_1(\mathbf{x}, \mathbf{x}') k_2(\mathbf{x}, \mathbf{x}') \\
k(\mathbf{x}, \mathbf{x}') &= k_3\left(\boldsymbol{\phi}(\mathbf{x}), \boldsymbol{\phi}(\mathbf{x}')\right) \\
k(\mathbf{x}, \mathbf{x}') &= \mathbf{x}^{\mathrm{T}} \mathbf{A} \mathbf{x}' \\
k(\mathbf{x}, \mathbf{x}') &= k_a(\mathbf{x}_a, \mathbf{x}'_a) + k_b(\mathbf{x}_b, \mathbf{x}'_b) \\
k(\mathbf{x}, \mathbf{x}') &= k_a(\mathbf{x}_a, \mathbf{x}'_a) k_b(\mathbf{x}_b, \mathbf{x}'_b)
\end{aligned}
$$

Reading

- Machine Learning
  Bishop : sections 1.1 to 1.2.4, 1.5.1-2, 6.1, 6.2

  Russell & Norvig: 18.9