Artificial Neural Networks

Jihoon Yang

Machine Learning Research Laboratory
Department of Computer Science & Engineering
Sogang University
Email: yangjh@sogang.ac.kr
URL: mllab.sogang.ac.kr

Neural Networks

- Decision trees are good at modeling nonlinear interactions among a small subset of attributes
- Sometimes we are interested in linear interactions among all attributes
- Simple neural networks are good at modeling such interactions
- The resulting models have close connections with naïve Bayes

Learning Threshold Functions

- Outline
- Background
- Threshold logic functions
- Connection to logic
- Connection to geometry
- Learning threshold functions – perceptron algorithms and its variants
- Perceptron convergence theorem

Background – Neural computation

- 1900: Birth of neuroscience – Ramon Cajal et al.
- 1913: Behaviorist or stimulus response psychology
- 1930-50: Theory of Computation, Church-Turing Thesis
- 1943: McCulloch & Pitts “A logical calculus of neuronal activity”
- 1949: Hebb – Organization of Behavior
- 1956: Birth of Artificial Intelligence – “Computers and Thought”
- 1960-65: Perceptron model developed by Rosenblatt

SOGANG UNIVERSITY

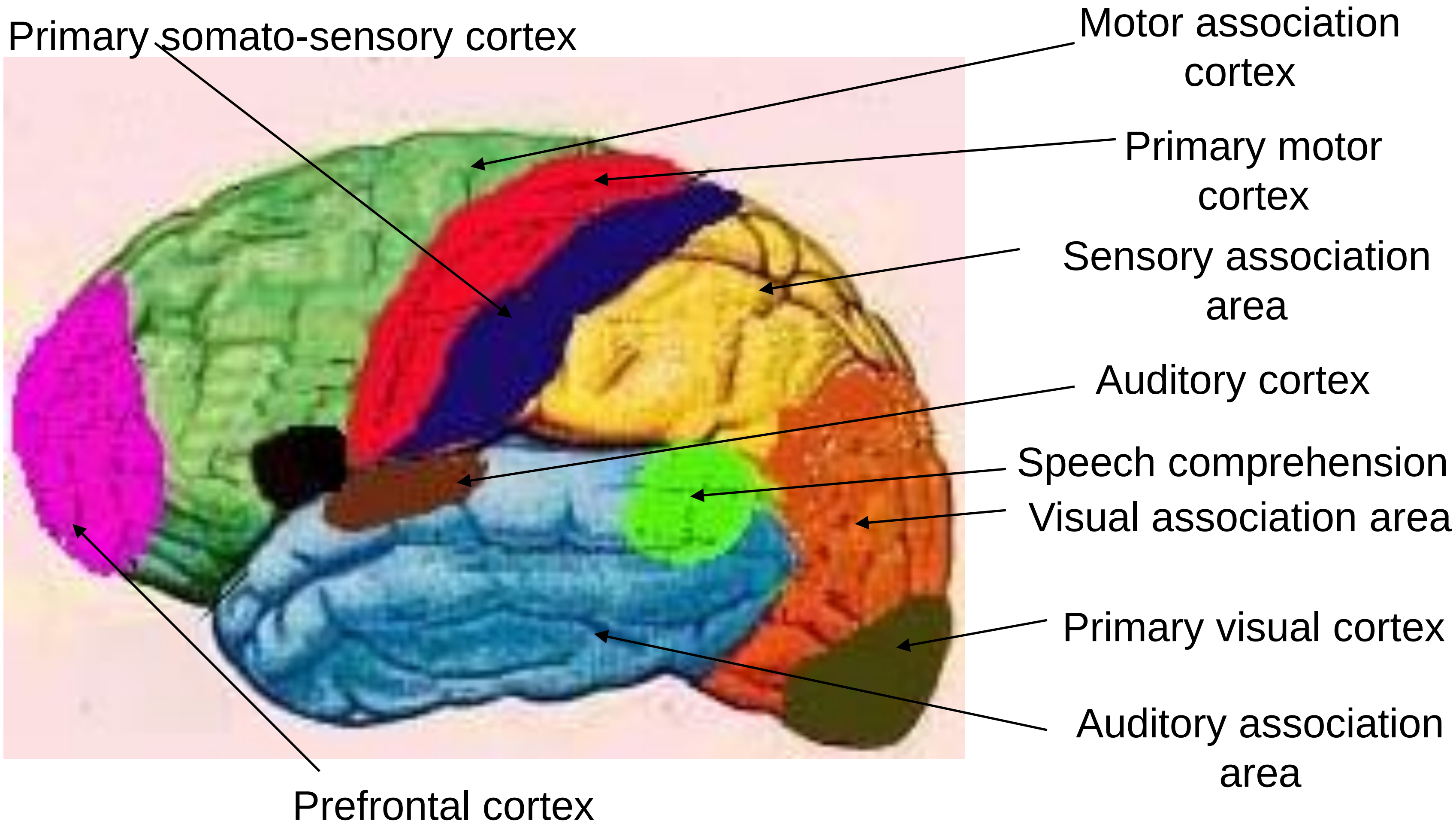
Background – Neural computation

- 1969: Minsky and Papert criticize Perceptron
- 1969: Chomsky argues for universal innate grammar
- 1970: Rise of cognitive psychology and knowledge-based AI
- 1975: Learning algorithms for multi-layer neural networks
- 1985: Resurgence of neural networks and machine learning
- 1988: Birth of computational neuroscience
- 1990: Successful applications (stock market, OCR, robotics)
- 1990-2000: New synthesis of behaviorist and cognitive or representational approaches in AI and psychology
- 2000-: Synthesis of logical and probabilistic approaches to representation and learning

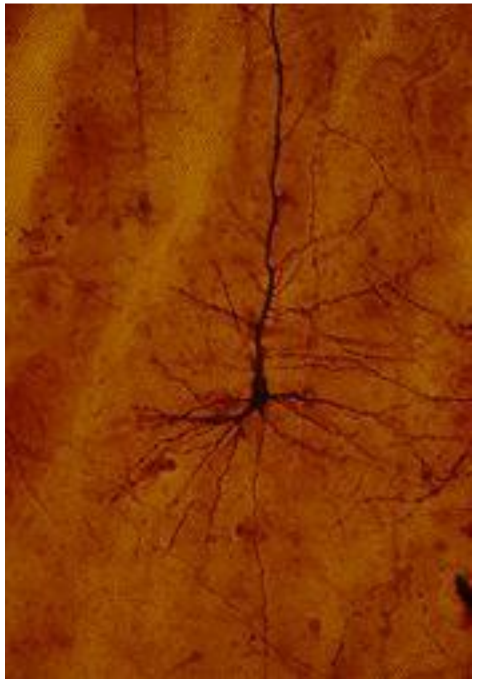
Background – Brains and Computers

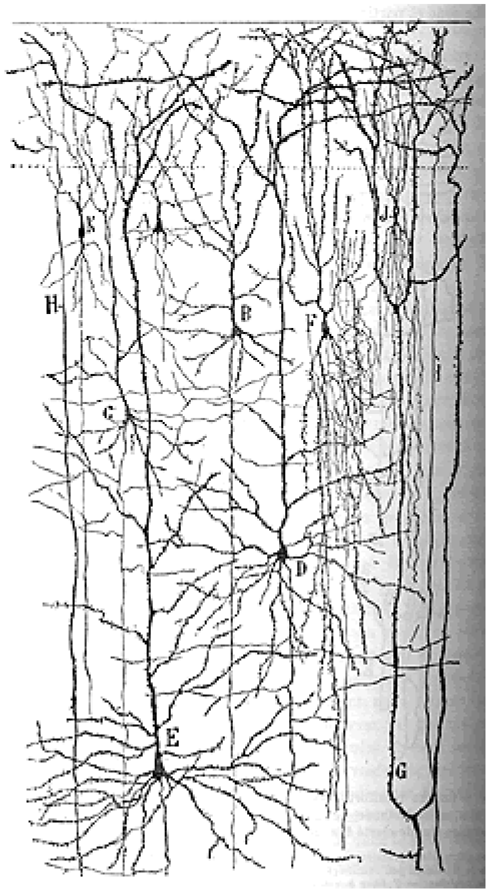
- Brain consists of 10^{11} neurons, each of which is connected to 10^{4} neighbors
- Each neuron is slow (1 millisecond to respond to a stimulus) but the brain is astonishingly fast at perceptual tasks (e.g. face recognition)
- Brain processes and learns from multiple sources of sensory information (visual, tactile, auditory…)
- Brain is massively parallel, shallowly serial, modular and hierarchical with recurrent and lateral connectivity within and between modules
- If cognition is – or at least can be modeled by – computation, it is natural to ask how and what brains compute

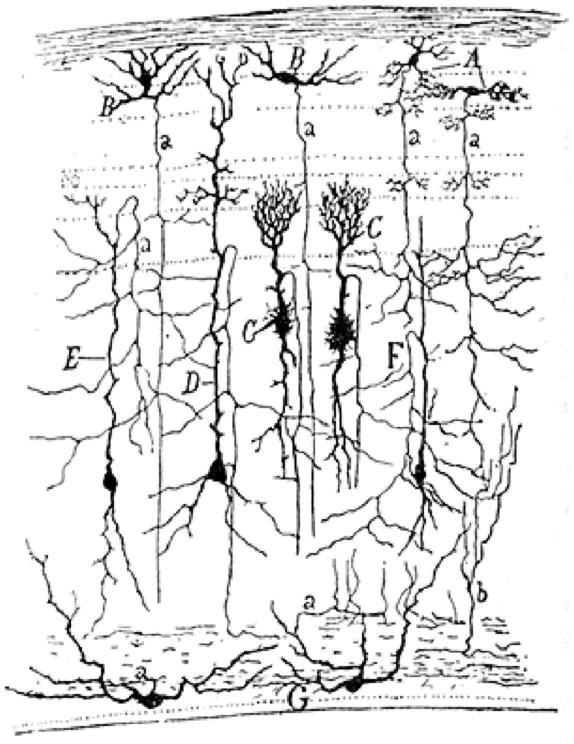
Brain and information processing



Neural Networks

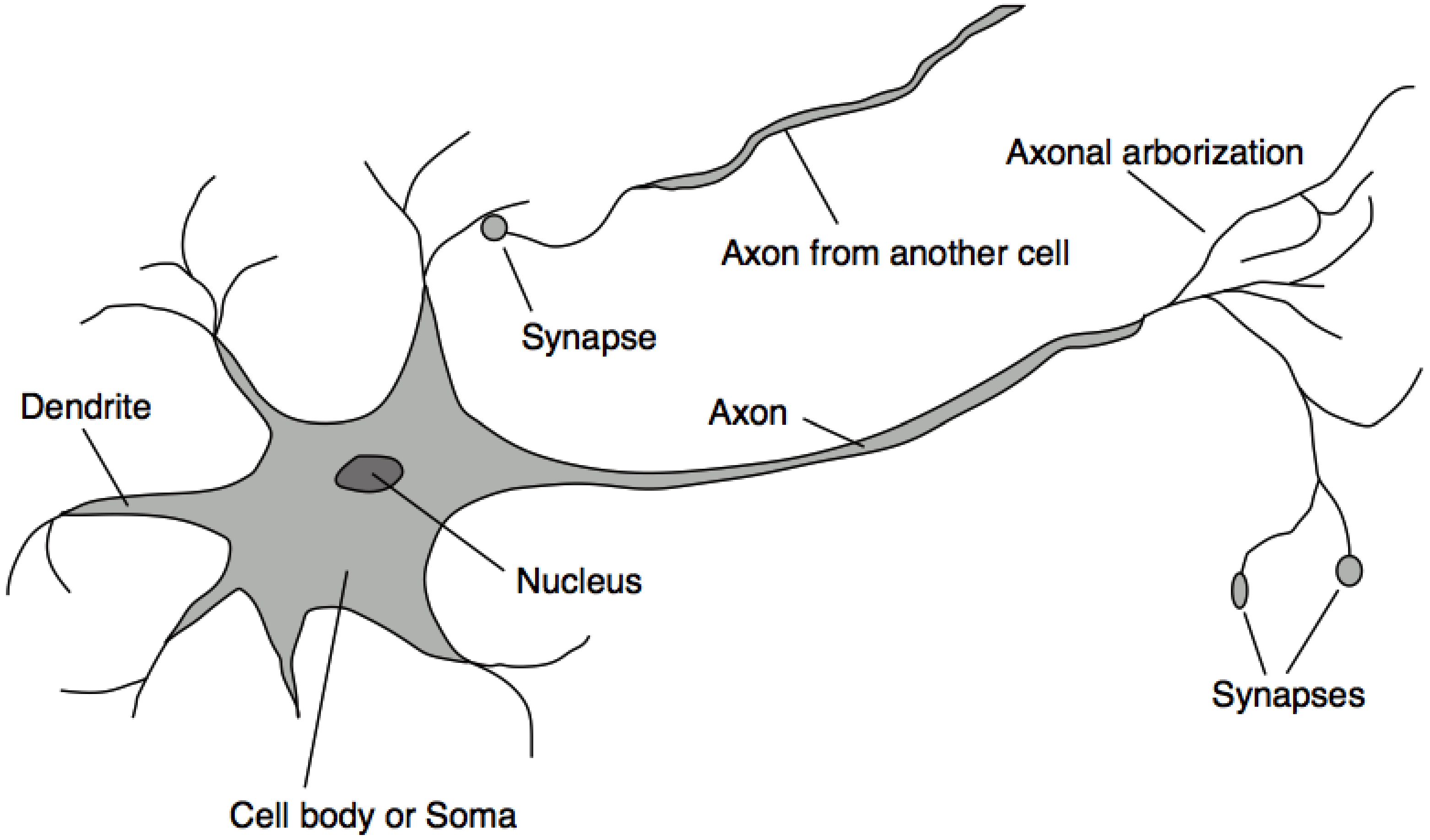




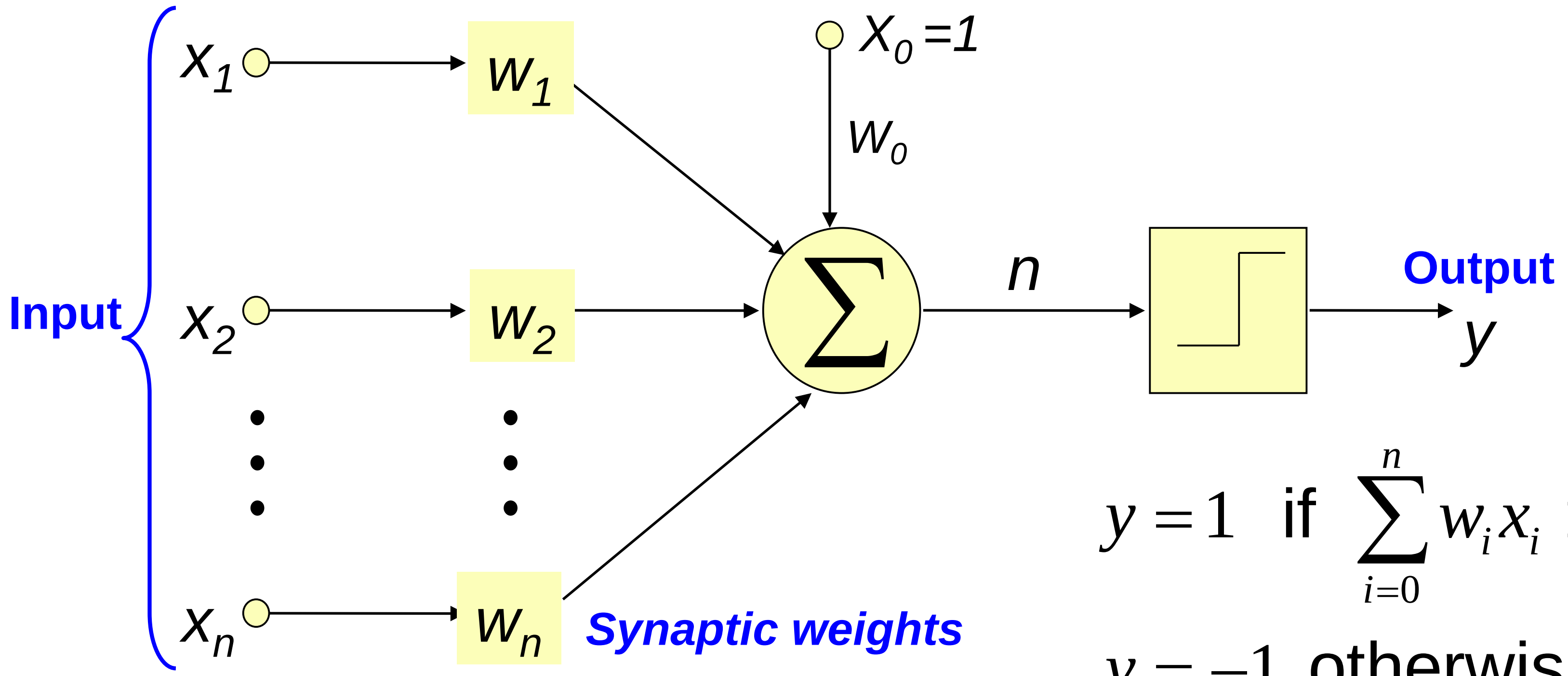


Ramon Cajal, 1900

Neurons and Computation



McCulloch-Pitts computational model of a neuron

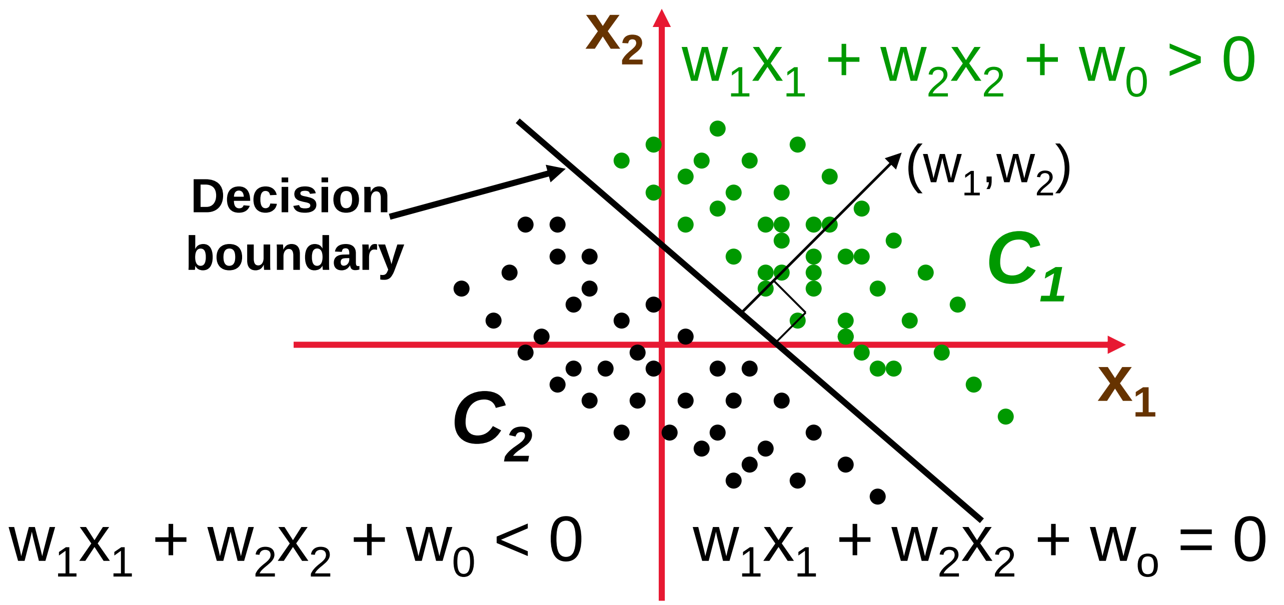


$$y = 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0$$

$$y = -1 \text{ otherwise}$$

When a neuron receives input signals from other neurons, its membrane voltage increases. When it exceeds a certain threshold, the neuron "fires" a burst of pulses.

Threshold neuron – Connection with Geometry



x_2
$w_1x_1 + w_2x_2 + w_0 > 0$
(w_1,w_2)
Decision boundary
C_1
C_2
x_1
$w_1x_1 + w_2x_2 + w_0 < 0$
$w_1x_1 + w_2x_2 + w_o = 0$

$\sum_{i=1}^{n} w_i x_i + w_0 = 0$ **describes a hyperplane which divides the instance space $\Re^n$ into two half–spaces**

$\chi_+ = \{X_p \in \Re^n | W \bullet X_p + w_0 > 0\}$ and $\chi_- = \{X_p \in \Re^n | W \bullet X_p + w_0 < 0\}$

McCulloch-Pitts neuron or Threshold neuron

$$
\begin{aligned}
y &= sign\left(\mathrm{W} \bullet \mathrm{X} + w_0\right) \\
&= sign\left(\sum_{i=0}^{n} w_i x_i\right) \\
&= sign\left(\mathrm{W}^T \mathrm{X} + w_0\right)
\end{aligned}
\qquad
\mathbf{X} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}
\qquad
\mathbf{W} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}
$$

$$
\begin{aligned}
sign(v) &= 1 \text{ if } v > 0 \\
&= 0 \text{ otherwise}
\end{aligned}
$$

Threshold neuron – Connection with Geometry

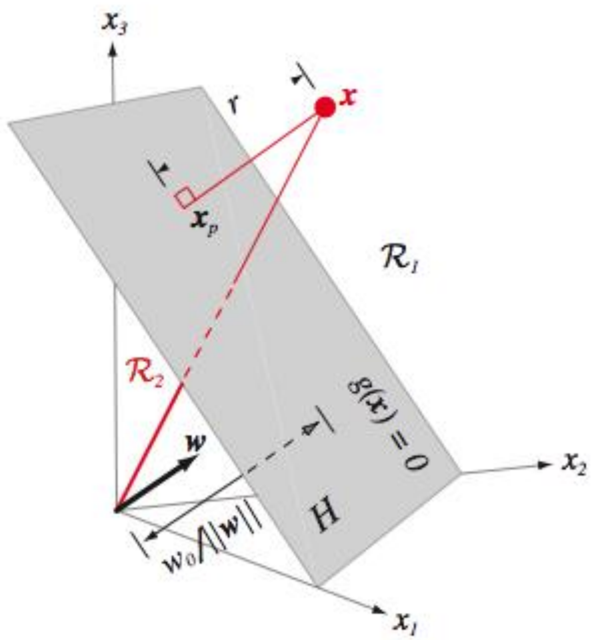


FIGURE 5.2. The linear decision boundary H, where $g(\mathbf{x}) = \mathbf{w}^t\mathbf{x} + w_0 = 0$, separates the feature space into two half-spaces $\mathcal{R}_1$ (where $g(\mathbf{x}) > 0$) and $\mathcal{R}_2$ (where $g(\mathbf{x}) < 0$). From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

Threshold neuron – Connection with Geometry

- **Instance space** $\Re^n$

- **Hypothesis space is the set of (*n*-1)-dimensional hyperplanes defined in the *n*-dimensional instance space**

- **A hypothesis is defined by** $\sum_{i=0}^{n} w_i x_i = 0$

- **Orientation of the hyperplane is governed by** $(w_1 \ldots w_n)^T$

- ***W* determines the orientation of the hyperplane *H*: given two points *X1* and *X2* on the hyperplane,**

$$W(X_1 - X_2) = 0$$

→ *W* is normal to any vector lying in *H*

Threshold neuron – Connection with Geometry

- $g(\vec{x})$ gives the algebraic distance from a point $\vec{x}$ to the hyperplane H

$$r = \frac{g(\vec{x})}{||\vec{w}||} = \frac{\sum_{i=1}^{n} w_i x_i + w_0}{\sqrt{w_1^2 + \ldots + w_n^2}}$$

- The Euclidean norm or *length* of a vector
  $||\vec{w}|| = \sqrt{\vec{w}^t \vec{w}} = \sqrt{w_1^2 + \ldots + w_n^2}$
- The algebraic distance from the origin to H is given by $w_0/||w||$

SOGANG UNIVERSITY

Threshold neuron – Connection with Geometry

- In summary, the linear function $g(\vec{x}) = \sum_{i=1}^{n} w_i x_i + w_0$ divides the feature space into two half-spaces by a hyperplane decision surface
- The orientation of the surface is determined by the normal vector $\vec{w}$, and the location of the surface is determined by the bias w_0.
- The value of $g(\vec{x})$ is proportional to the signed distance from any point $\vec{x}$ to the hyperplane

Threshold neuron as a pattern classifier

- The threshold neuron can be used to classify a set of instances into one of two classes C_1, C_2
- If the output of the neuron for input pattern X_p is +1 then X_p is assigned to class C_1
- If the output is -1 then the pattern X_p is assigned to C_2
- <u>Example</u>

$$[w_0\ w_1\ w_2]^T = [-1 -1\ 1]^T$$

$$\mathrm{X}_p^T = [1\,0]^T \quad \mathrm{W} \bullet \mathrm{X}_p + w_0 = -1 + (-1) = -2$$

X_p is assigned to class C_2

Threshold neuron – Connection with Logic

- Suppose the input space is $\{0,1\}^n$
- Then threshold neuron computes a Boolean function
  f: $\{0,1\}^n \rightarrow \{-1,1\}$

- <u>Example</u>
  - Let $w_0 = -1.5$; $w_1 = w_2 = 1$
  - In this case, the threshold neuron implements the logical AND function

x_1	x_2	$g(X)$	y
0	0	-1.5	-1
0	1	-0.5	-1
1	0	-0.5	-1
1	1	0.5	1

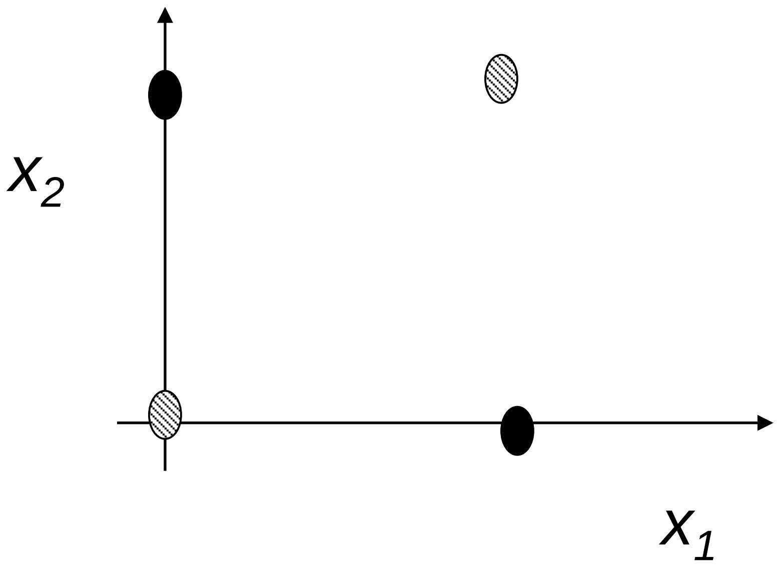
Threshold neuron – Connection with Logic

- A threshold neuron with the appropriate choice of weights can implement Boolean AND, OR, and NOT function
- Theorem: For any arbitrary Boolean function *f*, there exists a network of threshold neurons that can implement *f*
- Theorem: Any arbitrary finite state automaton can be realized using threshold neurons and *delay* units
- Networks of threshold neurons, given access to unbounded memory, can compute any Turing-computable function
- Corollary: Brains if given access to enough working memory, can compute any computable function

Threshold neuron – Connection with Logic

- **Theorem: There exist functions that cannot be implemented by a *single* threshold neuron**

- **Example: Exclusive OR**



Why?

Threshold neuron – Connection with Logic

- <u>Definition</u>: A function that can be computed by a single threshold neuron is called a threshold function
- Of the 16 2-input Boolean functions, 14 are Boolean threshold functions
- As *n* increases, the number of Boolean threshold functions becomes an increasingly small fraction of the total number of *n*-input Boolean functions

$$N_{Threshold}(n) \leq 2^{n^2}; \qquad N_{Boolean}(n) = 2^{2^n}$$

Terminology and Notation

- <u>Synonyms</u>: Threshold function, Linearly separable function, Linear discriminant function

- <u>Synonyms</u>: Threshold neuron, McCulloch-Pitts neuron, Perceptron, Threshold Logic Unit (TLU)

- We often include w_0 as one of the components of W and incorporate x_0 as the corresponding component of X with the understanding that $x_0 = 1$; Then $y = 1$ if $W \cdot X > 0$ and $y = -1$ otherwise

Learning Threshold functions

- **A training example *E_k* is an ordered pair (*X_k, d_k*) where**

$X_k = \left[x_{0k}\ x_{1k}\ \ldots.\ x_{nk}\right]^T$ **is an (*n+1*) dimensional input pattern, and**

$d_k = f(X_k) \in \{-1, 1\}$ **is the desired output of the classifier and *f* is an unknown target function to be learned**

- **A training set *E* is simply a multi-set of examples**

Learning Threshold functions

$$S^+ = \{X_k | (X_k, d_k) \in E \text{ and } d_k = 1\}$$
$$S^- = \{X_k | (X_k, d_k) \in E \text{ and } d_k = -1\}$$

- **We say that a training set *E* is linearly separable if and only if**

$$\exists W^* \text{ such that } \forall X_p \in S^+, W^* \bullet X_p > 0$$
$$\text{and } \forall X_p \in S^-, W^* \bullet X_p < 0$$

- **<u>Learning task</u>: Given a linearly separable training set *E*, find a solution $\mathbf{W}^*$**

$$\text{such that } \forall \mathbf{X}_p \in S^+, \mathbf{W}^* \bullet \mathbf{X}_p > 0 \text{ and } \forall \mathbf{X}_p \in S^-, \mathbf{W}^* \bullet \mathbf{X}_p < 0$$

Rosenblatt's Perceptron Learning Algorithm

1. Initialize $\mathrm{W} = [0\ 0 \ldots . 0]^T$

2. Set learning rate $\eta > 0$

3. Repeat until a complete pass through *E* results in no weight updates

```
    For each training example $E_k \in E$
    {
        $y_k \leftarrow sign(\mathrm{W} \bullet \mathrm{X}_k)$
        $\mathrm{W} \leftarrow \mathrm{W} + \eta(d_k - y_k)\mathrm{X}_k$
    }
```

4. $\mathbf{W}^* \leftarrow \mathbf{W}$; Return $\mathbf{W}^*$

Perceptron Learning Algorithm – Example

Let

$S^+ = \{(1, 1, 1), (1, 1, -1), (1, 0, -1)\}$

$S^- = \{(1,-1, -1), (1,-1, 1), (1,0, 1)\}$

$W = (0\ 0\ 0)$

$$\eta = \frac{1}{2}$$

X_k	d_k	W	$W.X_k$	y_k	Update?	Updated W
(1, 1, 1)	1	(0, 0, 0)	0	-1	Yes	(1, 1, 1)
(1, 1, -1)	1	(1, 1, 1)	1	1	No	(1, 1, 1)
(1,0, -1)	1	(1, 1, 1)	0	-1	Yes	(2, 1, 0)
(1, -1, -1)	-1	(2, 1, 0)	1	1	Yes	(1, 2, 1)
(1,-1, 1)	-1	(1, 2, 1)	0	-1	No	(1, 2, 1)
(1,0, 1)	-1	(1, 2, 1)	2	1	Yes	(0, 2, 0)
(1, 1, 1)	1	(0, 2, 0)	2	1	No	(0, 2, 0)

Perceptron Convergence Theorem (Novikoff)

Theorem:

Let $E = \{(\mathbf{X}_k, d_k)\}$ **be a training set where** $\mathbf{X}_k \in \{1\} \times \Re^n$ **and** $d_k \in \{-1, 1\}$

Let $S^+ = \{\mathbf{X}_k | (\mathbf{X}_k, d_k) \in E \;\&\; d_k = 1\}$ and $S^- = \{\mathbf{X}_k | (\mathbf{X}_k, d_k) \in E \;\&\; d_k = -1\}$

The perceptron algorithm is guaranteed to terminate after a bounded number *t* of weight updates with a weight vector $\mathbf{W}^*$

such that $\forall\, \mathbf{X}_k \in S^+, \mathbf{W}^* \bullet \mathbf{X}_k \geq \delta$ and $\forall\, \mathbf{X}_k \in S^-, \mathbf{W}^* \bullet \mathbf{X}_k \leq -\delta$

for some $\delta > 0$**, whenever such** $\mathbf{W}^* \in \Re^{n+1}$ and $\delta > 0$ **exist**

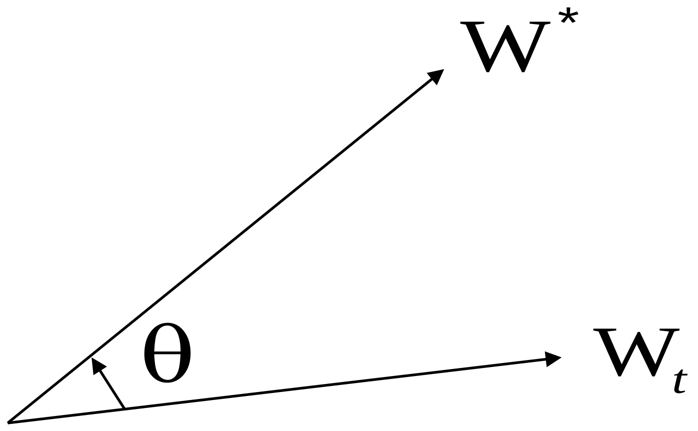
– that is, *E is linearly separable.*

The bound on the number *t* of weight updates is given by

$$t \leq \left(\frac{\|\mathbf{W}^*\| L}{\delta} \right)^2 \quad \text{where } L = \max_{\mathbf{X}_k \in S} \|\mathbf{X}_k\| \quad \text{and } S = S^+ \cup S^-$$

Proof of Perceptron Convergence Theorem

Let $\mathbf{w}_t$ be the weight vector after *t* weight updates



$\text{Invariant}: \ \forall\theta \quad |\cos\theta| \le 1$

Proof of Perceptron Convergence Theorem

Let W^* be such that

$$\forall X_k \in S^+, W^* \bullet X_k \geq \delta \text{ and } \forall X_k \in S^-, W^* \bullet X_k \leq -\delta$$

WLOG assume that $W^* \bullet X = 0$ passes through the origin.

Let $\forall X_k \in S^+, Z_k = X_k,$

$\forall X_k \in S^-, Z_k = -X_k,$

$Z = \{Z_k\}$

$$\left(\forall X_k \in S^+, W^* \bullet X_k \geq \delta \ \& \ \forall X_k \in S^-, W^* \bullet X_k \leq -\delta\right)$$

$$\Leftrightarrow \left(\forall Z_k \in Z, W^* \bullet Z_k \geq \delta\right).$$

Let $E' = \{(Z_k, 1)\}$

Proof of Perceptron Convergence Theorem

$$W_{t+1} = W_t + \eta(d_k - y_k)Z_k$$

where $W_0 = [0\,0\,....\,0]^T$ and $\eta > 0$

$[$Weight update based on example $(Z_k, 1)$ $]$

$$\Leftrightarrow [(d_k = 1) \wedge (y_k = -1)]$$

$$\therefore W^* \bullet W_{t+1} = W^* \bullet (W_t + 2\eta Z_k)$$

$$= (W^* \bullet W_t) + 2\eta(W^* \bullet Z_k)$$

Since $\forall Z_k \in Z, (W^* \bullet Z_k \geq \delta), W^* \bullet W_{t+1} \geq W^* \bullet W_t + 2\eta\delta$

$\therefore \forall t \quad W^* \bullet W_t \geq 2t\eta\delta$ ……………………………………(a)

Proof of Perceptron Convergence Theorem

$$\begin{aligned}\left\|\mathrm{W}_{t+1}\right\|^2 &\equiv \mathrm{W}_{t+1} \bullet \mathrm{W}_{t+1} \\ &= \left(\mathrm{W}_t + 2\eta \mathrm{Z}_k\right) \bullet \left(\mathrm{W}_t + 2\eta \mathrm{Z}_k\right) \\ &= \left(\mathrm{W}_t \bullet \mathrm{W}_t\right) + 4\eta\left(\mathrm{W}_t \bullet \mathrm{Z}_k\right) + 4\eta^2\left(\mathrm{Z}_k \bullet \mathrm{Z}_k\right)\end{aligned}$$

Note weight update based on $\mathrm{Z}_k \Leftrightarrow \left(\mathrm{W}_t \bullet \mathrm{Z}_k \le 0\right)$

$$\therefore \left\|\mathrm{W}_{t+1}\right\|^2 \le \left\|\mathrm{W}_t\right\|^2 + 4\eta^2\left\|\mathrm{Z}_k\right\|^2 \le \left\|\mathrm{W}_t\right\|^2 + 4\eta^2 L^2$$

Hence $\left\|\mathrm{W}_t\right\|^2 \le 4t\eta^2 L^2$

$\therefore \forall t \ \left\|\mathrm{W}_t\right\| \le 2\eta L\sqrt{t}$ ……………………………………(b)

Department of Computer Science & Engineering
Machine Learning Research Laboratory

SOGANG UNIVERSITY

Proof of Perceptron Convergence Theorem

From (a) we have : $\forall t \ \left(\mathrm{W}^* \bullet \mathrm{W}_t\right) \geq 2t\eta\delta$

$\Rightarrow \left\{\forall t \quad 2t\eta\delta \leq \left(\mathrm{W}^* \bullet \mathrm{W}_t\right)\right\} \Rightarrow \left\{\forall t \quad 2t\eta\delta \leq \left\|\mathrm{W}^*\right\|\left\|\mathrm{W}_t\right\| \cos\theta\right\}$

$\Rightarrow \left\{\forall t \quad 2t\eta\delta \leq \left\|\mathrm{W}^*\right\|\left\|\mathrm{W}_t\right\|\right\} \because \forall\theta \quad \cos\theta \leq 1,$

Substituting for an upper bound on $\left\|\mathrm{W}_t\right\|$ from (b),

$\forall t \ \left\{2t\eta\delta \leq \left\|\mathrm{W}^*\right\| 2\eta L\sqrt{t}\right\} \Rightarrow \left\{\forall t \quad \left(\delta\sqrt{\mathrm{t}} \leq \left\|\mathrm{W}^*\right\| L\right)\right\}$

$$\Rightarrow \left\{\forall t \quad \left\{t \leq \left(\frac{\left\|\mathrm{W}^*\right\| L}{\delta}\right)^2\right\}\right\}$$

Notes on the Perceptron Convergence Theorem

- **The bound on the number of weight updates does not depend on the learning rate**

- **The bound is not useful in determining when to stop the algorithm because it depends on the norm of the unknown weight vector and delta**

- **The convergence theorem offers no guarantees when the training data set is not linearly separable**

- **<u>Exercise</u>: Prove that the perceptron algorithm is robust with respect to fluctuations in the learning rate**

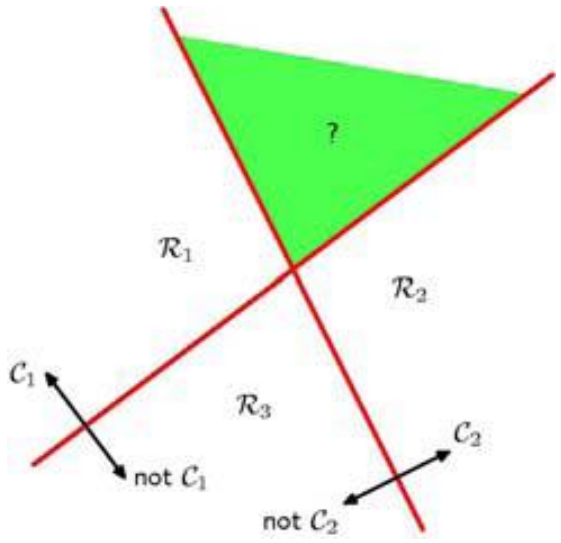
$$0 < \eta_{\text{min}} \le \eta_t \le \eta_{\text{max}} < \infty$$

Multicategory classification

- Assuming c categories: $\omega_1, \ldots, \omega_c$
- Possible approaches
  - use $c-1$ (or c) perceptrons: each solving ω_i/not ω_i dichotomies
  - use $c(c-1)/2$ perceptrons: one for every pair ω_i/ω_j dichotomies
- Problem: both can lead to regions in which the classification is undefined

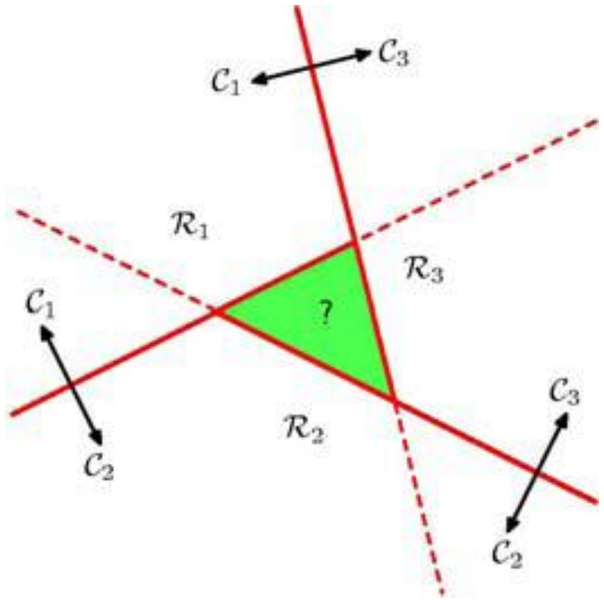
Multiple classes

K-1 **binary classifiers**



One-versus-rest

K(K-1)/2 **binary classifiers**

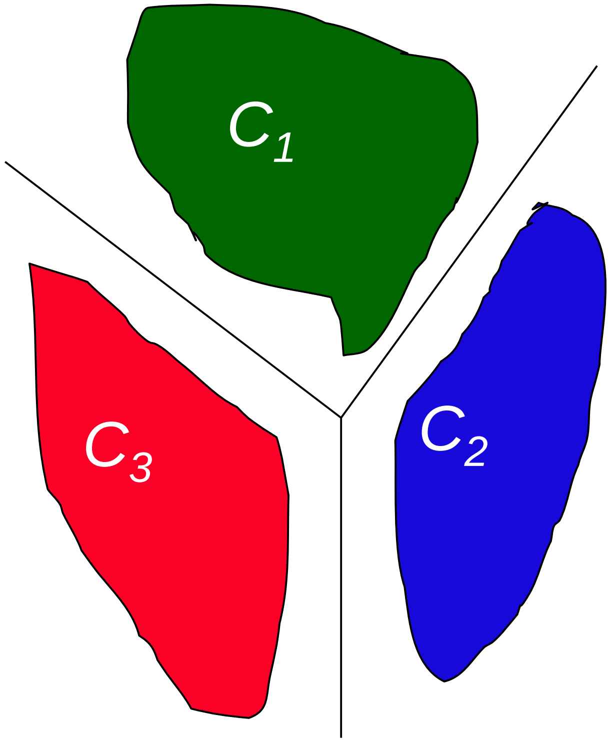


One-versus-one

Problem: Green region has ambiguous class membership

Multi-category classifiers

Winner-Take-All Network



- **Define *K* linear functions of the form:**

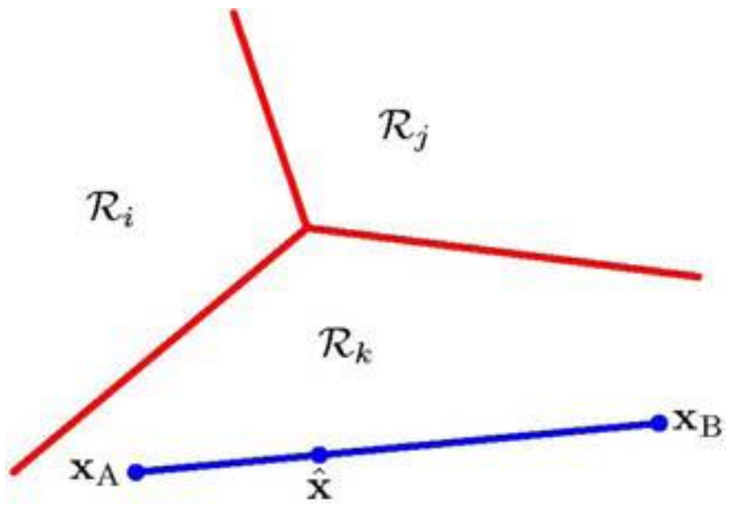
$$y_k(X) = W_k^T X + w_{k0}$$

$$h(X) = \arg\max_k y_k(X)$$
$$= \arg\max_k \left(W_k^T X + w_{k0}\right)$$

- **Decision surface between class *Ck* and *Cj***

$$\left(W_k - W_j\right)^T X + \left(w_{k0} - w_{j0}\right) = 0$$

Linear separator for *K* classes



- **Decision regions defined by**

$$\left(W_k - W_j\right)^T X + \left(w_{k0} - w_{j0}\right) = 0$$

are simply connected and convex

- **For any points** $X_A, X_B \in R_k$, **any** $\hat{X}$ **that lies on the line connecting *XA* and *XB***

$$\hat{X} = \lambda X_A + (1-\lambda) X_B \text{ where } 0 \le \lambda \le 1$$

also lies in *Rk*

Winner-Take-All Networks

$$y_{ip} = 1 \text{ iff } W_i \bullet X_p > W_j \bullet X_p \quad \forall j \neq i$$

$$y_{ip} = 0 \text{ otherwise}$$

$$W_1 = [1\ \ -1\ \ -1]^T, W_2 = [1\ \ 1\ \ 1]^T, W_3 = [2\ \ 0\ \ 0]^T$$

Note: *Wj* are *augmented* weight vectors

			$W_1.X_p$	$W_2.X_p$	$W_3.X_p$	y_1	y_2	y_3
1	-1	-1	3	-1	2	1	0	0
1	-1	+1	1	1	2	0	0	1
1	+1	-1	1	1	2	0	0	1
1	+1	+1	-1	3	2	0	1	0

What does neuron 3 compute?

Linear separability of multiple classes

Let $S_1, S_2, S_3 \ldots S_M$ be multisets of instances

Let $C_1, C_2, C_3 \ldots C_M$ be disjoint classes

$\forall i \;\; S_i \subseteq C_i$

$\forall i \neq j \;\; C_i \bigcap C_j = \varnothing$

We say that the sets $S_1, S_2, S_3 \ldots S_M$ are linearly separable iff $\exists$ weight vectors $\mathrm{W}_1^*, \mathrm{W}_2^*, ..\mathrm{W}_M^*$ such that

$$\forall i \;\; \left\{ \forall \mathrm{X}_p \in S_i, \left(\mathrm{W}_i^* \bullet \mathrm{X}_p > \mathrm{W}_j^* \bullet \mathrm{X}_p \right) \forall j \neq i \right\}$$

Training WTA Classifiers

$d_{kp} = 1$ iff $\mathbf{X}_p \in C_k$; $d_{kp} = 0$ otherwise

$y_{kp} = 1$ iff $\mathbf{W}_k \bullet \mathbf{X}_p > \mathbf{W}_j \bullet \mathbf{X}_p \quad \forall k \neq j$

Suppose $d_{kp} = 1, y_{jp} = 1$ and $y_{kp} = 0$

$\mathbf{W}_k \leftarrow \mathbf{W}_k + \eta\mathbf{X}_p; \mathbf{W}_j \leftarrow \mathbf{W}_j - \eta\mathbf{X}_p;$

All other weights are left unchanged.

Suppose $d_{kp} = 1, y_{jp} = 0$ and $y_{kp} = 1$.

The weights are unchanged.

Suppose $d_{kp} = 1, \forall j \; y_{jp} = 0$ (there was a tie)

$\mathbf{W}_k \leftarrow \mathbf{W}_k + \eta\mathbf{X}_p$

All other weights are left unchanged.

WTA Convergence Theorem

- Given a linearly separable training set, the WTA learning algorithm is guaranteed to converge to a solution within a finite number of weight updates

- <u>Proof sketch</u>: Transform the WTA training problem to the problem of training a single perceptron using a suitably transformed training set; Then the proof of WTA learning algorithm reduces to the proof of perceptron learning algorithm

WTA Convergence Theorem

Let $\mathbf{W}^T = [\mathbf{W}_1 \mathbf{W}_2 \ldots . \mathbf{W}_M]^T$ be a concatenation of the weight vectors associated with the M neurons in the WTA group. Consider a multi − category training set $E = \{(\mathbf{X}_p, f(\mathbf{X}_p))\}$ where $\forall \mathbf{X}_p \; f(\mathbf{X}_p) \in \{C_1, \ldots C_M\}$

Let $\mathbf{X}_p \in C_1$. Generate $(M-1)$ training examples using $\mathbf{X}_p$ for an $M(n+1)$ input perceptron :

$\mathbf{X}_{p12} = [\mathbf{X}_p \; -\mathbf{X}_p \; \phi \quad \phi \ldots \phi]$

$\mathbf{X}_{p13} = [\mathbf{X}_p \quad \phi \; -\mathbf{X}_p \; \phi \ldots \phi]$

$\ldots$

$\mathbf{X}_{p1M} = [\mathbf{X}_p \; \phi \quad \phi \ldots \phi - \mathbf{X}_p]$

where ϕ is an all zero vector with the same dimension as $\mathbf{X}_p$ and set the desired output of the corresponding perceptron to be 1 in each case.

Similarly, from each training example for an $(n+1)$ – input WTA, we can generate $(M-1)$ examples for an $M(n+1)$ input single neuron.

Let the union of the resulting $|E|(M-1)$ examples be E'

WTA Convergence Theorem

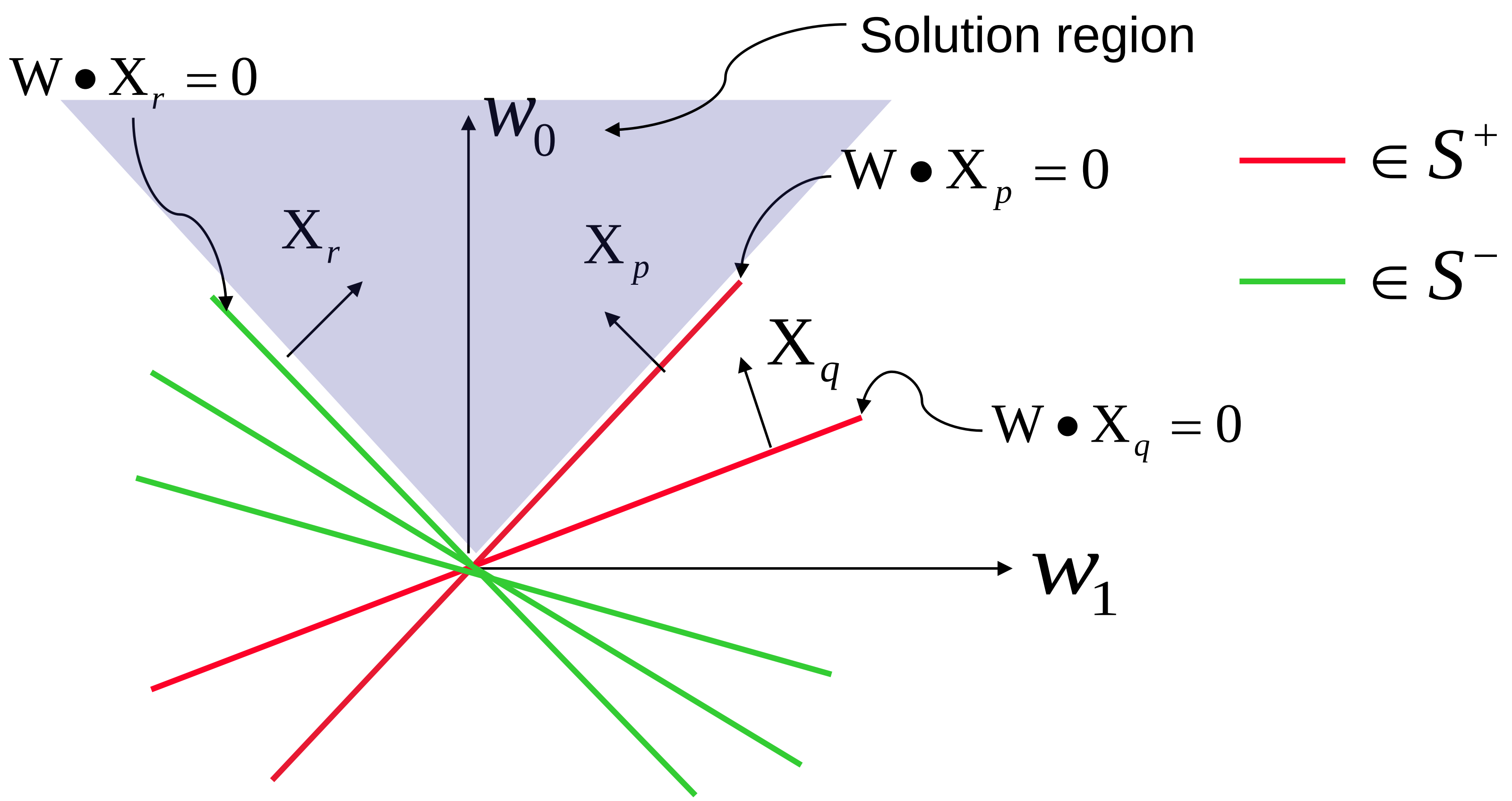
By construction, there is a one – to – one correspondence between the weight vector $\mathbf{W}^T = [\mathbf{W}_1 \mathbf{W}_2 \ldots . \mathbf{W}_M]^T$ that results from training an M – neuron WTA on the multi – category set of examples E and the result of training an $M(n+1)$ input perceptron on the transformed training set E'. Hence the convergence proof of WTA learning algorithm follows from the perceptron convergence theorem.

Weight space representation

- Pattern space representation:
  - Coordinates of space correspond to attributes (features)
  - A point in the space represents an instance
  - Weight vector W_v defines a hyperplane $W_v \cdot X = 0$

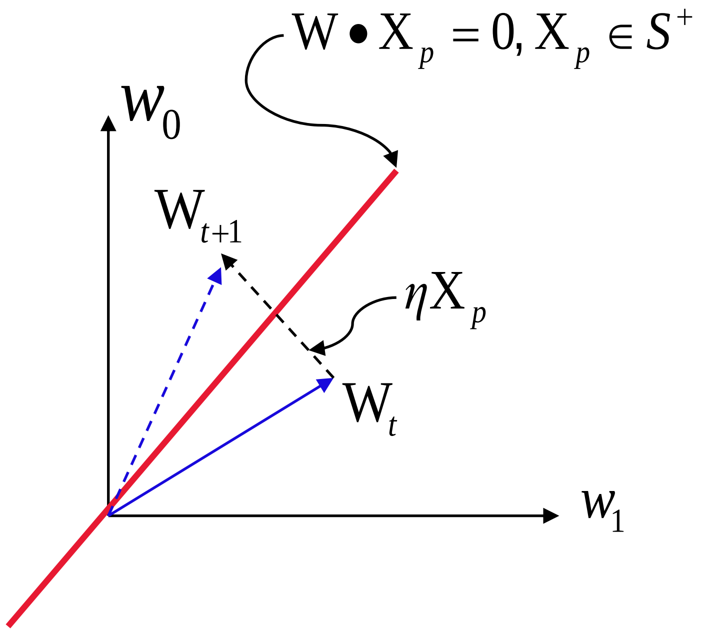
- Weight space (dual) representation:
  - Coordinates define a weight space
  - A point in the space represents a choice of weights W_v
  - An instance X_p defines a hyperplane $W \cdot X_p = 0$

Weight space representation



SOGANG UNIVERSITY

Weight space representation



$$\mathbf{W}_{t+1} \leftarrow \mathbf{W}_t + \eta \mathbf{X}_p$$

Fractional correction rule

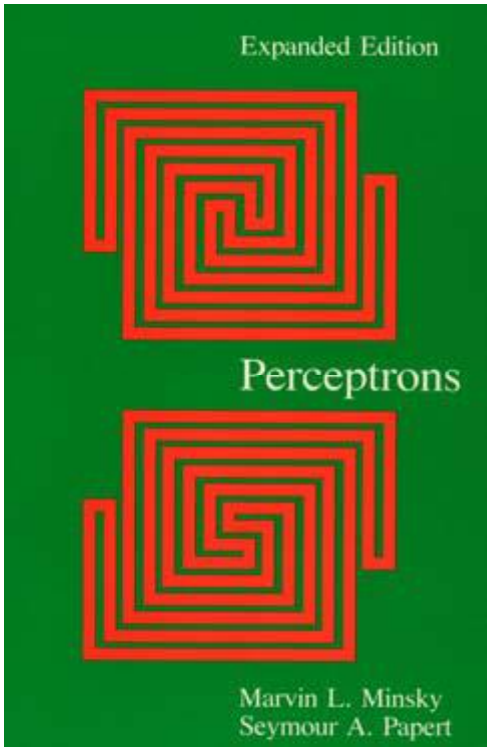
$$\mathbf{W}_{t+1} \leftarrow \mathbf{W}_t + \lambda \left(\frac{\left| \mathbf{W}_t \bullet \mathbf{X}_p \right| + \varepsilon}{\mathbf{X}_p \bullet \mathbf{X}_p + \varepsilon} \right) (d_p - y_p) \mathbf{X}_p$$

$0 < \lambda < 1; \lambda \neq 0.5$ when $d_p, y_p \in \{-1, 1\}$

$\varepsilon > 0$ is a constant (to handle the case when the dot product $\mathbf{W}_t \bullet \mathbf{X}_p$ or $\mathbf{X}_p \bullet \mathbf{X}_p$ (or both) approach zero.

"Perceptrons" (1969)

"The perceptron […] has many features that attract attention: its linearity, its intriguing learning theorem; its clear paradigmatic simplicity as a kind of parallel computation. *There is no reason to suppose that any of these virtues carry over to the many-layered version*. Nevertheless, *we consider it to be an important research problem to elucidate (or reject) our intuitive judgement that the extension is sterile*." [pp. 231 – 232]

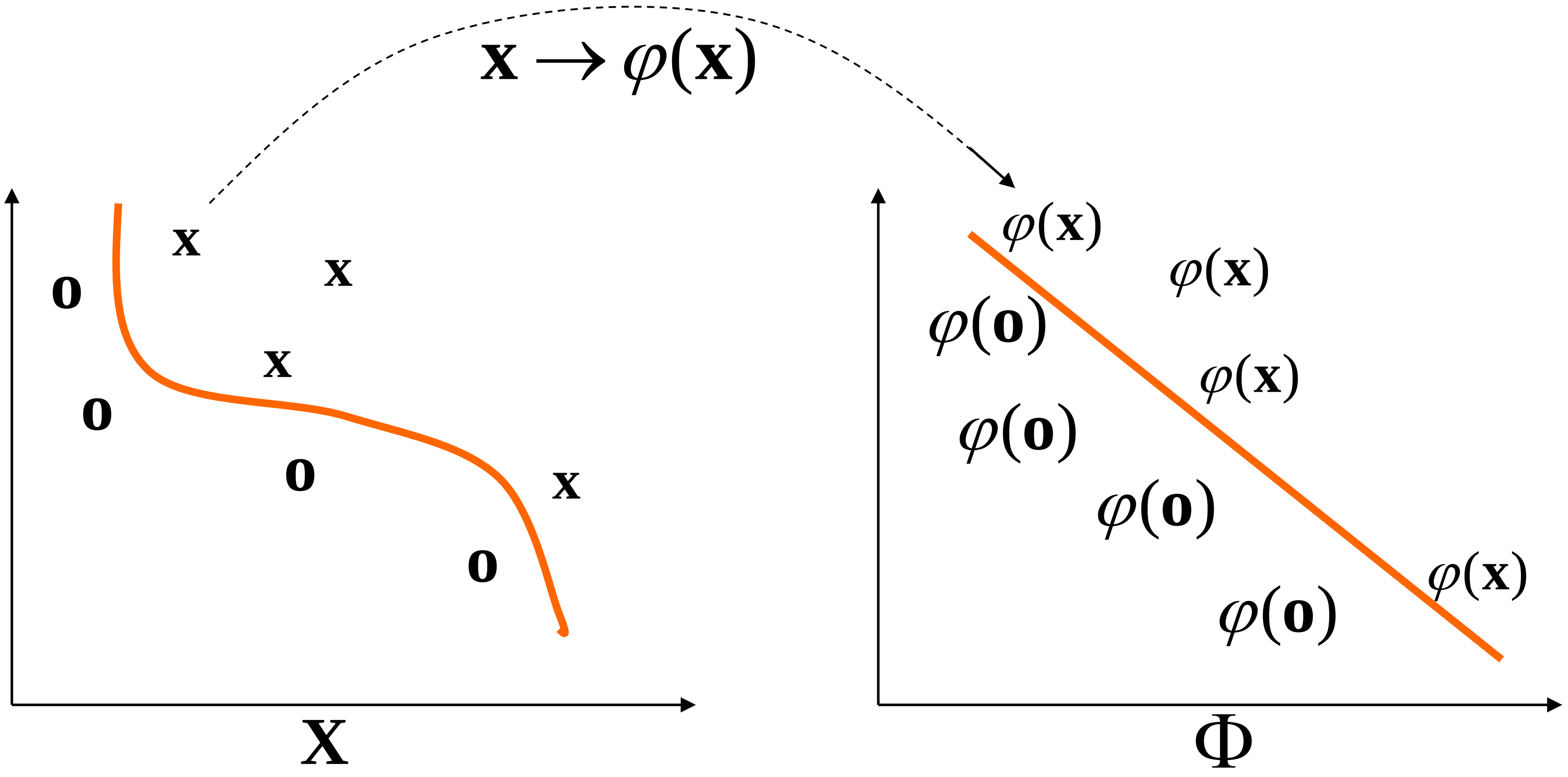


Limitations of Perceptrons

- Perceptrons can only represent threshold functions
- Perceptrons can only learn linear decision boundaries
- What if the data are not linearly separable?
  - Modify the learning procedure or the weight update equation?
    (e.g. Pocket algorithm, Thermal perceptron)
  - More complex networks?
  - Non-linear transformations into a feature space where the data become separable?

Extending Linear Classifiers: Learning in feature spaces

- Map data into a feature space where they are linearly separable

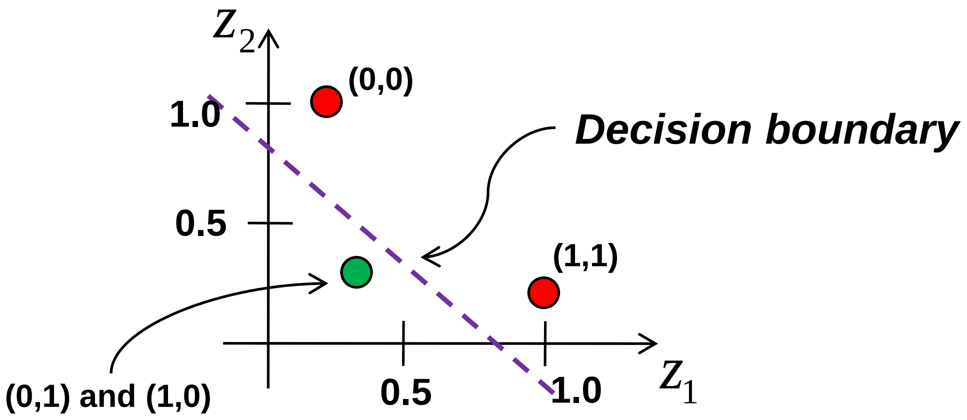


Exclusive OR revisited

- In the feature (hidden) space:

$$\varphi_1(x_1, x_2) = e^{-\|X - W_1\|^2} = z_1 \qquad W_1 = [1,1]^T$$

$$\varphi_2(x_1, x_2) = e^{-\|X - W_2\|^2} = z_2 \qquad W_2 = [0,0]^T$$



- When mapped into the feature space <*z1, z2*>, *C1* and *C2* become *linearly separable*. So a linear classifier with *φ1(X)* and *φ2(X)* as inputs can be used to solve the XOR problem.

Learning in the Feature Space

- **High dimensional feature spaces**

$$X = (x_1, x_2, \cdots, x_n) \rightarrow \varphi(X) = (\varphi_1(X), \varphi_2(X), \cdots, \varphi_d(X))$$

where typically *d* >> *n* solve the problem of expressing complex functions

- **But this introduces**
  - **Computational problem (working with very large vectors)**
    - **→ Solved using the kernel trick – implicit feature spaces**
  - **Generalization problem (curse of dimensionality)**
    - **→ Solved by maximizing the margin of separation – first implemented in SVM (Vapnik)**

We will return to SVM later..

Linear Classifiers – linear discriminant functions

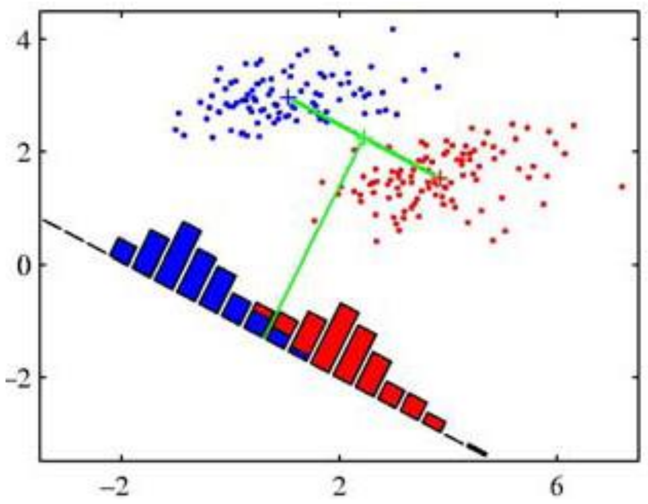
- Perceptron implements a linear discriminant function – a linear decision surface given by

$$X = \begin{bmatrix} x_1 \\ x_2 \\ \\ \\ x_n \end{bmatrix} \quad W = \begin{bmatrix} w_1 \\ w_2 \\ \\ \\ w_n \end{bmatrix} \qquad y(X) = W^T X + w_0 = 0$$

- The solution hyperplane simply has to separate the classes

- We can consider alternative criteria for separating hyperplanes

Project data onto a line joining the means of the two classes



Measure of separation of classes
– separation of the projected class means

$$m_2 - m_1 = W^T(\mu_2 - \mu_1)$$

Problems:

- **Separation can be made arbitrarily large by increasing the magnitude of *W* – constrain *W* to be of unit length**

- **Classes that are well separated in the original space can have non trivial overlap in the projection – maximize the separation between the projected class means while giving a small variance within each class, thereby minimizing the class overlap**

Fisher's Linear Discriminant

- Given two classes, find the linear discriminant $W \in \Re^n$ that maximizes Fisher's discriminant ratio:

$$f(W;\mu_1,\Sigma_1,\mu_2,\Sigma_2) = \frac{\left(W^T(\mu_1-\mu_2)\right)^2}{W^T(\Sigma_1+\Sigma_2)W}$$

$$= \frac{W^T(\mu_1-\mu_2)(\mu_1-\mu_2)^T W}{W^T(\Sigma_1+\Sigma_2)W}$$

- Set

$$\frac{\partial f(W;\mu_1,\Sigma_1,\mu_2,\Sigma_2)}{\partial W} = 0$$

Fisher's Linear Discriminant

$$f(W;\mu_1,\Sigma_1,\mu_2,\Sigma_2)=\frac{W^T(\mu_1-\mu_2)(\mu_1-\mu_2)^T W}{W^T(\Sigma_1+\Sigma_2)W}$$

$$\frac{\partial f(W;\mu_1,\Sigma_1,\mu_2,\Sigma_2)}{\partial W}$$

$$=\frac{(W^T(\Sigma_1+\Sigma_2)W)\frac{\partial(W^T(\mu_1-\mu_2)(\mu_1-\mu_2)^T W)}{\partial W}-(W^T(\mu_1-\mu_2)(\mu_1-\mu_2)^T W)\frac{\partial}{\partial W}(W^T(\Sigma_1+\Sigma_2)W)}{(W^T(\Sigma_1+\Sigma_2)W)^2}$$

$$(W^T(\Sigma_1+\Sigma_2)W)\frac{\partial(W^T(\mu_1-\mu_2)(\mu_1-\mu_2)^T W)}{\partial W}-(W^T(\mu_1-\mu_2)(\mu_1-\mu_2)^T W)\frac{\partial}{\partial W}(W^T(\Sigma_1+\Sigma_2)W)=0$$

$$(W^T(\Sigma_1+\Sigma_2)W)2(\mu_1-\mu_2)(\mu_1-\mu_2)^T W-(W^T(\mu_1-\mu_2)(\mu_1-\mu_2)^T W)2(\Sigma_1+\Sigma_2)W=0$$

$$(\mu_1-\mu_2)(\mu_1-\mu_2)^T W=k(\Sigma_1+\Sigma_2)W\ (k=\text{const})$$

$$(\mu_1-\mu_2)=k'(\Sigma_1+\Sigma_2)W\ (k'=\text{const}\ \because(\mu_1-\mu_2)(\mu_1-\mu_2)^T W\text{ has the same direction as }(\mu_1-\mu_2))$$

$$\boxed{W^*\propto(\Sigma_1+\Sigma_2)^{-1}(\mu_1-\mu_2)}$$

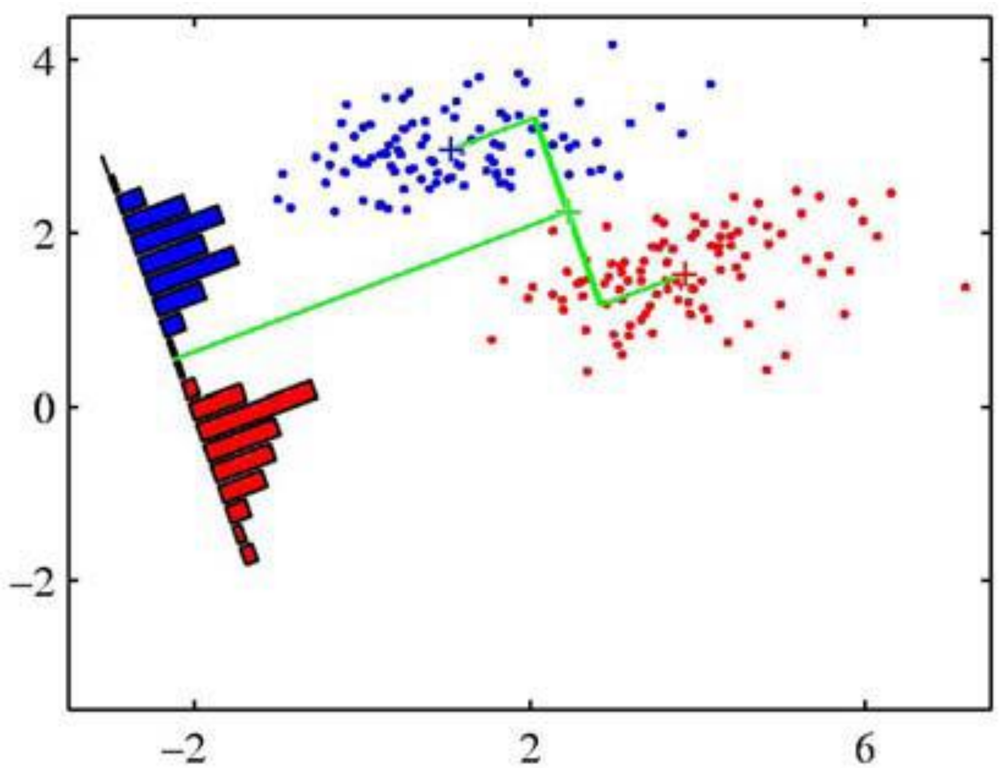
Fisher's Linear Discriminant

$W \in \Re^n$ **that maximizes Fisher's discriminant ratio:**

$$W^* = (\Sigma_1 + \Sigma_2)^{-1}(\mu_1 - \mu_2)$$

- **Unique solution**
- **Easy to compute**
- **Has a probabilistic interpretation (e.g. model *P(y|ci)* as normal dist., estimate parameters by MLE, and use Bayes decision rule)**
- **Can be updated incrementally as new data become available**
- **Naturally extends to *K*-class problems**
- **Can be generalized (using kernel trick) to handle non linearly separable class boundaries**

Project data based on Fisher discriminant



Fisher’s Linear Discriminant

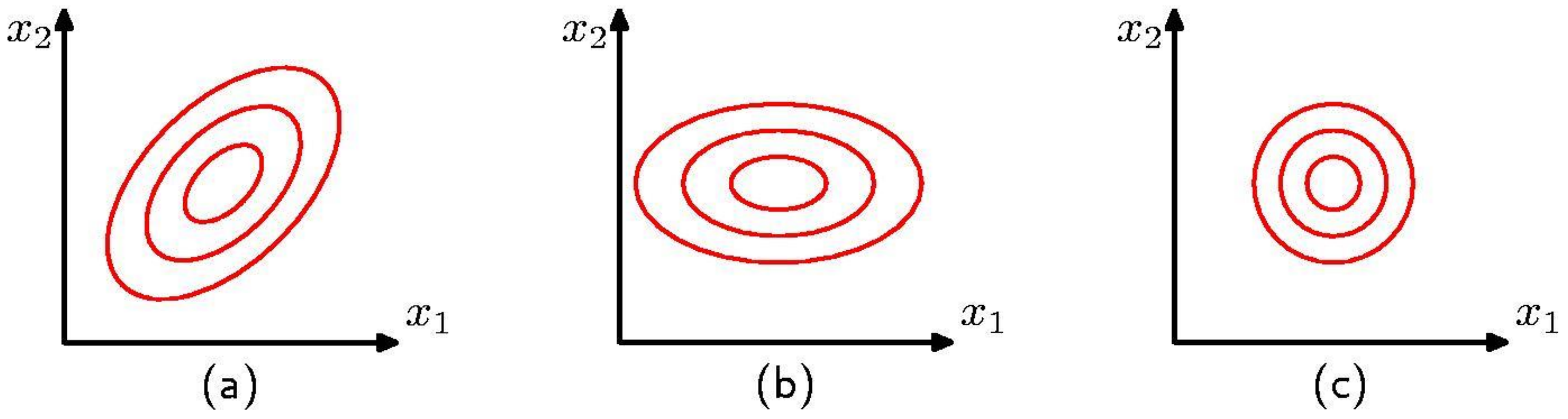
- Can be shown to maximize between class separation

- If the samples in each class have Gaussian distribution, then classification using the Fisher discriminant can be shown to yield minimum error classifier

- If the within class variance is isotropic, then Σ_1 and Σ_2 are proportional to the identity matrix *I* and *W* corresponding to the Fisher discriminant is proportional to the difference between the class means $(\mu_1 - \mu_2)$

- Can be generalized to *K* classes

Contours of constant probability density for a Gaussian distribution in 2D



General (non-diagonal)

Diagonal

Isotropic

Generative vs. Discriminative Models

- Bayesian decision theory revisited
- Generative models
  - Naïve Bayes
- Discriminative models
  - Perceptron, Fisher discriminant, Support vector machines
- Relating generative and discriminative models
- Tradeoffs between generative and discriminative models
- Generalizations and extensions

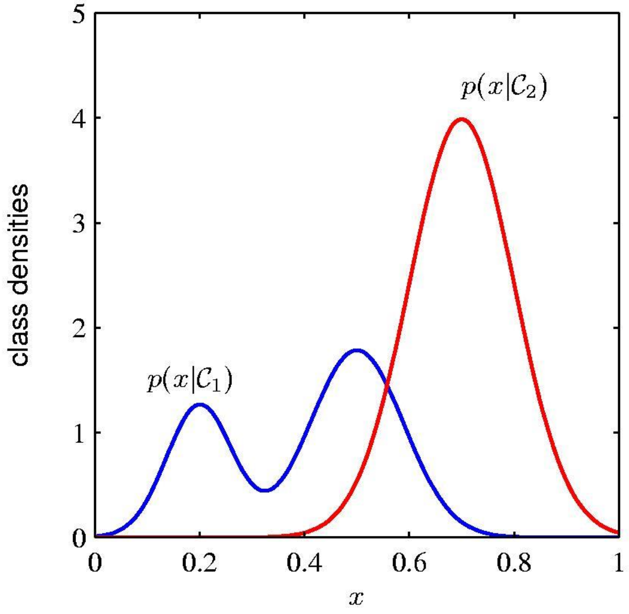
Generative vs. Discriminative Classifiers

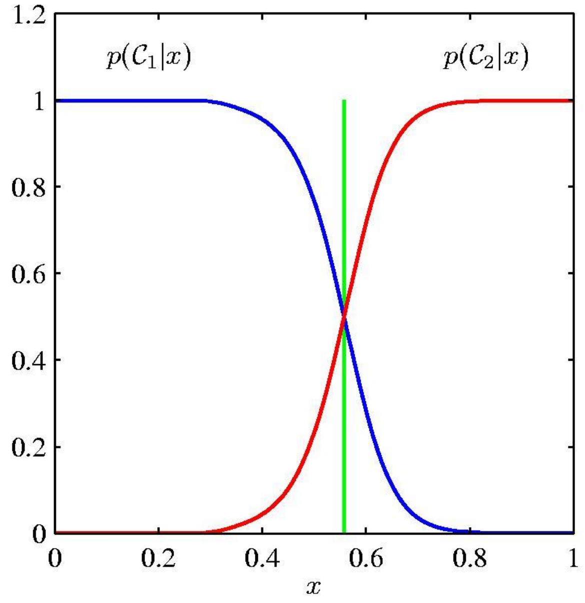
- **Generative classifiers**
  - Assume some functional form for *P(X|C)*, *P(C)*
  - Estimate parameters of *P(X|C)*, *P(C)* directly from training data
  - Use Bayes rule to calculate *P(C|X=x)*

- **Discriminative classifiers – conditional version**
  - Assume some functional form for *P(C|X)*
  - Estimate parameters of *P(C|X)* directly from training data

- **Discriminative classifiers – maximum margin version**
  - Assume some functional form *f(W)* for the discriminant
  - Find *W* that maximizes the margin of separation between classes (e.g. SVM)

Generative vs. Discriminative Models





Which chef cooks a better Bayesian recipe?

- In theory, generative and conditional models produce identical results in the limit
  - The classification produced by the generative model is the same as that produced by the discriminative model
  - That is, given unlimited data, assuming that both approaches select the correct form for the relevant probability distribution or the model for the discriminant function, they will produce identical results
  - If the assumed form of the probability distribution is incorrect, then it is possible that the generative model might have a higher classification error than the discriminative model
- How about in practice?

Which chef cooks a better Bayesian recipe?

- In practice
  - The error of the classifier that uses the discriminative model can be lower than that of the classifier that uses the generative model
  - Naïve Bayes is a generative model
  - A perceptron is a discriminative model, and so is SVM
  - An SVM can outperform naïve Bayes on classification
- If the goal is classification, it might be useful to consider discriminative models that directly learn the classifier without going solving the harder intermediate problem of modeling the joint probability distribution of inputs and classes (Vapnik)

From generative to discriminative models

- **Assume classes are binary** $y \in \{0,1\}$

- **Suppose we model the class by a binomial distribution with parameter *q***

$$p(y \mid q) = q^{y}(1-q)^{(1-y)}$$

- **Assume each component *Xj* of input *X* each have Gaussian distributions with parameters *ѳj* and are independent given the class**

$$p(x, y \mid \Theta) = p(y \mid q)\prod_{j=1}^{n} p(x_j \mid y, \theta_j)$$

$$\text{where } \Theta = (q, \theta_1, \ldots, \theta_n)$$

From generative to discriminative models

$$p(x_j \mid y=0, \theta_j) = \frac{1}{(2\pi\sigma_j^2)^{1/2}} \exp\left\{-\frac{1}{2\sigma_j^2}(x_j - \mu_{0j})^2\right\}$$

$$p(x_j \mid y=1, \theta_j) = \frac{1}{(2\pi\sigma_j^2)^{1/2}} \exp\left\{-\frac{1}{2\sigma_j^2}(x_j - \mu_{1j})^2\right\}$$

$$\text{where } \theta_j = (\mu_{0j}, \mu_{1j}, \sigma_j)$$

$$(\text{Note: we have assumed that } \forall j \, \sigma_{0j} = \sigma_{1j} = \sigma_j)$$

From generative to discriminative models

- **The calculation of the posterior probability *p(Y=1|x,Θ)* is simplified if we use matrix notation**

$$p(x \mid y=1,\Theta)=\prod_{j=1}^{n}\left(\frac{1}{(2\pi)^{1/2}\sigma_j}\exp\left\{-\frac{1}{2}\left(\frac{x_j-\mu_{1j}}{\sigma_j}\right)^2\right\}\right)$$

$$=\frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}}\exp\left\{-\frac{1}{2}(x-\mu_1)^T\Sigma^{-1}(x-\mu_1)\right\}$$

$$\text{where } \mu_1=(\mu_{11},\dots,\mu_{1n})^T;\text{ and } \Sigma=diag(\sigma_1^2\dots\sigma_n^2)=\begin{bmatrix}\sigma_1^2 & 0 & 0\\ 0 & . & 0\\ 0 & 0 & \sigma_n^2\end{bmatrix}$$

From generative to discriminative models

$$p(y=1|x,\Theta)=\frac{p(x|y=1,\Theta)\,p(y=1|q)}{p(x|y=1,\Theta)\,p(y=1|q)+p(x|y=0,\Theta)\,p(y=0|q)}$$

$$=\frac{q\exp\left\{-\frac{1}{2}(x-\mu_1)^T\Sigma^{-1}(x-\mu_1)\right\}}{q\exp\left\{-\frac{1}{2}(x-\mu_1)^T\Sigma^{-1}(x-\mu_1)\right\}+(1-q)\exp\left\{-\frac{1}{2}(x-\mu_0)^T\Sigma^{-1}(x-\mu_0)\right\}}$$

$$=\frac{1}{1+\exp\left\{-\log\left(\frac{q}{1-q}\right)+\frac{1}{2}(x-\mu_1)^T\Sigma^{-1}(x-\mu_1)-\frac{1}{2}(x-\mu_0)^T\Sigma^{-1}(x-\mu_0)\right\}}$$

$$=\frac{1}{1+\exp\left\{-\underbrace{(\mu_1-\mu_0)^T\Sigma^{-1}}_{\beta^T}x+\underbrace{\frac{1}{2}(\mu_1-\mu_0)^T\Sigma^{-1}(\mu_1+\mu_0)-\log\left(\frac{q}{1-q}\right)}_{\gamma}\right\}}$$

$$=\frac{1}{1+\exp(-\beta^T x-\gamma)}$$

where we have used $A^T DA-B^T DB=(A+B)^T D(A-B)$ for a symmetric matrix D

From generative to discriminative models

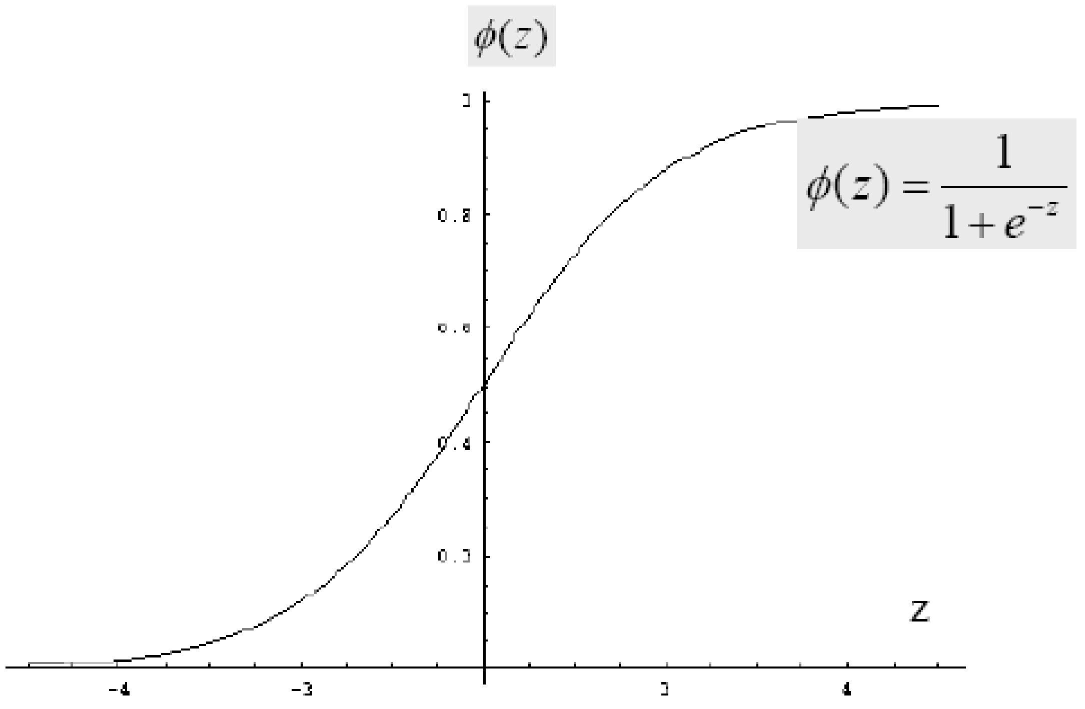
$$p(y=1\,|\,x,\Theta)=\frac{1}{1+\exp(-\beta^T x-\gamma)}$$

- **The posterior probability that *Y = 1* takes the form**

$$\phi(z)=\frac{1}{1+e^{-z}}$$

where $z=\beta^T x+\gamma$ is an affine function of x

Sigmoid or Logistic Function



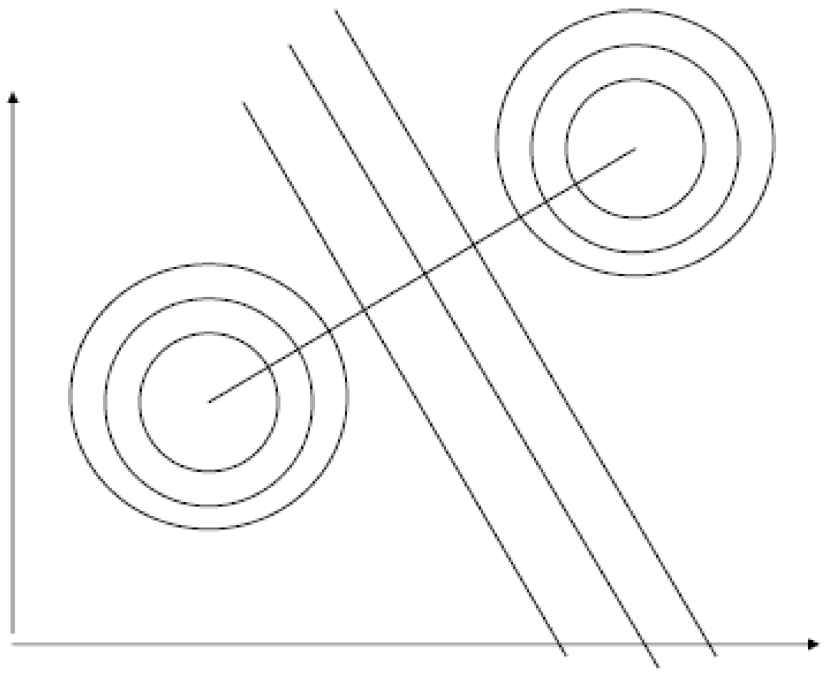
$$\phi(z) = \frac{1}{1+e^{-z}}$$

Implications of the logistic posterior

- **Posterior probability of *Y* is a logistic function of an affine function of *x***
- **Contours of equal posterior probability are lines in the input space**
- $\beta^T x$ **is proportional to the projection of *x* on *β* and this projection is equal for all vectors *x* that lie along a line that is orthogonal to *β***
- **Special case**
  - **Variances of Gaussians = 1**
  - **The contours of equal posterior probability are lines that are orthogonal to the difference vector between the means of the two classes**
- **Equal posterior for the two classes when *z = 0***

Geometric interpretation (diagonal *Σ*)

Contour plot



Geometric interpretation

$$p(y=1|x,\Theta)=\frac{p(x|y=1,\Theta)p(y=1|q)}{p(x|y=1,\Theta)p(y=1|q)+p(x|y=0,\Theta)p(y=0|q)}$$

$$=\frac{1}{1+\exp\left\{-\underbrace{(\mu_1-\mu_0)^T\Sigma^{-1}}_{\beta^T}x+\underbrace{\frac{1}{2}(\mu_1-\mu_0)^T\Sigma^{-1}(\mu_1+\mu_0)-\log\left(\frac{q}{1-q}\right)}_{\gamma}\right\}}$$

$$=\frac{1}{1+\exp(-\beta^T x-\gamma)}=\frac{1}{1+e^{-z}}$$

$$\text{when } q=1-q, z=(\mu_1-\mu_0)^T\Sigma^{-1}\left(x-\frac{(\mu_1+\mu_0)}{2}\right)$$

- **In this case, the posterior probabilities for the two classes are equal when *x* is equidistant from the two means**

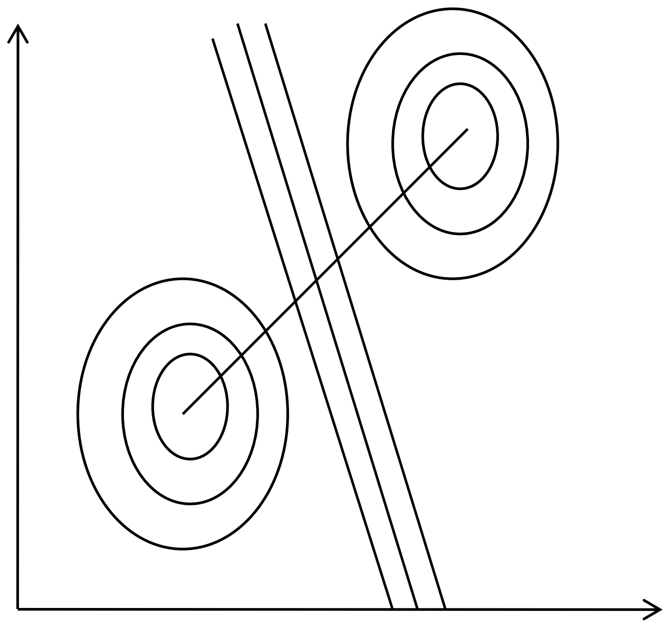
Geometric interpretation

- If the prior probabilities of the classes are such that $q > 0.5$ the effect is to shift the logistic function to the left resulting in a larger value for the posterior probability for $Y = 1$ for any given point in the input space

- $q < 0.5$ results in a shift of the logistic function to the right resulting in a smaller value for the posterior probability for $Y = 1$ (or larger value for the posterior probability for $Y = 0$)

Geometric interpretation (general Σ)

Contour plot



Now the equi-probability contours are still lines in the input space although the lines are no longer orthogonal to the difference in means of the two classes

Generalization to multiple classes – Softmax function

- *Y* is a multinomial variable which takes on one of *K* values

$$q_k = p(y=k \mid q) = p(y^k = 1 \mid q)$$

$$\text{where } (y=k) \equiv (y^k = 1), q = (q_1 q_2 \ldots q_K)$$

- As before, *X* is a multivariate Gaussian

$$p(X \mid y^k = 1, \Theta) = \frac{1}{(2\pi)^{n/2} \mid \Sigma \mid^{1/2}} \exp\left\{ -\frac{1}{2} (X - \mu_k)^T \Sigma^{-1} (X - \mu_k) \right\}$$

$$\text{where } \mu_k = (\mu_{k1} \ldots \mu_{kn}); \text{ and } \forall k \Sigma_k = \Sigma$$

(covariance matrix is assumed to be same for each class)

Generalization to multiple classes – Softmax function

- **Posterior probability for class *k* is obtained via Bayes rule**

$$p(y^k = 1 \mid X, \Theta) = \frac{p(X \mid y^k = 1, \Theta)\, p(y^k = 1 \mid q)}{\sum_{l=1}^{K} p(X \mid y^l = 1, \Theta)\, p(y^l = 1 \mid q)}$$

$$= \frac{q_k \exp\left\{ -\frac{1}{2}(X - \mu_k)^T \Sigma^{-1} (X - \mu_k) \right\}}{\sum_{l=1}^{K} q_l \exp\left\{ -\frac{1}{2}(X - \mu_l)^T \Sigma^{-1} (X - \mu_l) \right\}}$$

$$= \frac{\exp\left\{ \mu_k^T \Sigma^{-1} X - \frac{1}{2}\mu_k^T \Sigma^{-1} \mu_k + \log q_k) \right\}}{\sum_{l=1}^{K} \exp\left\{ \mu_l^T \Sigma^{-1} X - \frac{1}{2}\mu_l^T \Sigma^{-1} \mu_l + \log q_l \right\}}$$

Generalization to multiple classes – Softmax function

- **We have shown that**

$$p(y^k=1\mid X,\Theta)=\frac{\exp\left\{\mu_k^T\Sigma^{-1}X-\frac{1}{2}\mu_k^T\Sigma^{-1}\mu_k+\log q_k)\right\}}{\sum_{l=1}^{K}\exp\left\{\mu_l^T\Sigma^{-1}X-\frac{1}{2}\mu_l^T\Sigma^{-1}\mu_l+\log q_l\right\}}$$

- **Defining parameter vectors and augmenting the input vector *X* by adding a constant input of 1 we have**

$$\beta_k=\begin{bmatrix}-\frac{1}{2}\mu_k^T\Sigma^{-1}\mu_k+\log q_k\\ \Sigma^{-1}\mu_k\end{bmatrix}$$

$$p(y^k=1\mid X,\Theta)=\frac{e^{\beta_k^T X}}{\sum_{l=1}^{K}e^{\beta_l^T X}}=\frac{e^{\langle\beta_k,X\rangle}}{\sum_{l=1}^{K}e^{\langle\beta_l,X\rangle}}$$

Generalization to multiple classes – Softmax function

$$p(y^k = 1 | X, \Theta) = \frac{e^{\beta_k^T X}}{\sum_{l=1}^{K} e^{\beta_l^T X}} = \frac{e^{\langle \beta_k, X \rangle}}{\sum_{l=1}^{K} e^{\langle \beta_l, X \rangle}}$$

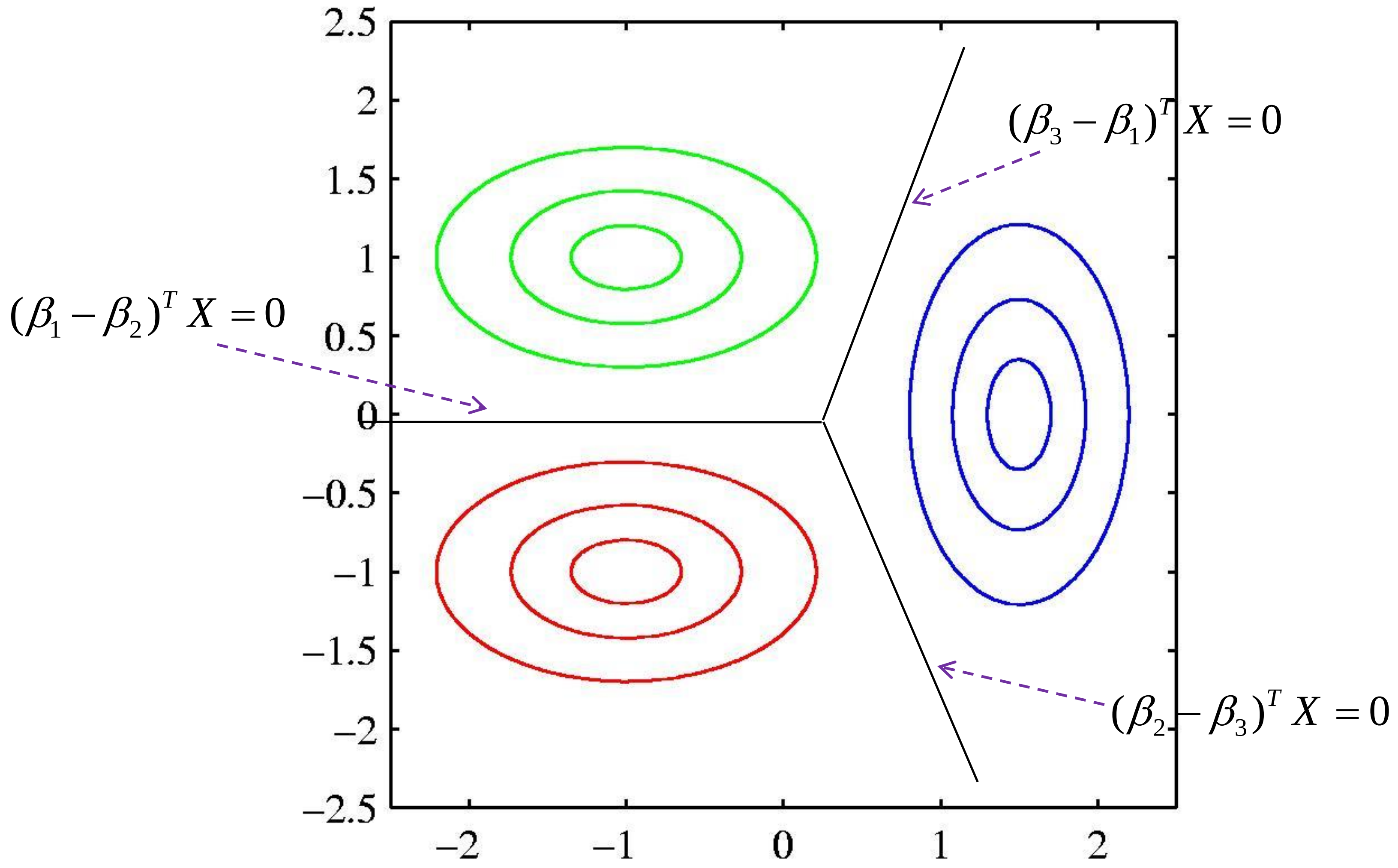
- **Corresponds to the decision rule**

$$h(X) = \arg\max_k p(y^k = 1 | X, \Theta) = \arg\max_k e^{\langle \beta_k, X \rangle} = \arg\max_k \langle \beta_k, X \rangle$$

- **Consider the ratio of posterior prob. for classes *k* and *j ≠ k***

$$\frac{p(y^k = 1 | X, \Theta)}{p(y^j = 1 | X, \Theta)} = \frac{e^{\langle \beta_k, X \rangle}}{\sum_{l=1}^{K} e^{\langle \beta_l, X \rangle}} \frac{\sum_{l=1}^{K} e^{\langle \beta_l, X \rangle}}{e^{\langle \beta_j, X \rangle}} = \frac{e^{\langle \beta_k, X \rangle}}{e^{\langle \beta_j, X \rangle}} = e^{\langle (\beta_k - \beta_j), X \rangle}$$

Equi-probability contours of the softmax function



From generative to discriminative models

- A curious fact about all of the generative models we have considered so far is that
  - The posterior probability of class can be expressed in the form of a logistic function in the case of a binary classifier and a softmax function in the case of a *K*-class classifier

- For multinomial and Gaussian class conditional densities (in the case of the latter, with equal but otherwise arbitrary covariance matrices)
  - The contours of equal posterior probabilities of classes are hyperplanes in the input (feature) space

- The result is a simple linear classifier analogous to the perceptron (for binary classification) or winner-take-all network (for *K*-ary classification)

- These results hold for a more general class of distributions

The exponential family of distributions

- **The exponential family is specified by**

$$p(X \mid \eta) = h(X) e^{\left\{\eta^T G(X) - A(\eta)\right\}}$$

where *η* is a parameter vector and *A(η)*, *h(X)* and *G(X)* are appropriately chosen functions

- **Gaussian, binomial, and multinomial (and many other) distributions belong to the exponential family**

The Gaussian distribution belongs to the exponential family

$$p(X \mid \eta) = h(X) e^{\left\{\eta^T G(X) - A(\eta)\right\}}$$

- **Univariate Gaussian distribution can be written as**

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

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

- **We see that Gaussian distribution belongs to the exponential family by choosing**

$$\eta = \begin{bmatrix} \mu/\sigma^2 \\ -1/2\sigma^2 \end{bmatrix}; A(\eta) = \frac{\mu^2}{2\sigma^2} + \ln\sigma$$

$$G(x) = \begin{bmatrix} x \\ x^2 \end{bmatrix}; h(x) = \frac{1}{(2\pi)^{1/2}}$$

The exponential family of distributions

- The exponential family which is given by

$$p(X \mid \eta) = h(X) e^{\left\{\eta^T G(X) - A(\eta)\right\}}$$

where *η* is a parameter vector and *A(η)*, *h(X)* and *G(X)* are appropriately chosen functions – can be shown to include several additional distributions such as the multinomial, the Poisson, the Gamma, the Dirichlet, among others

From generative to discriminative models

- In the case of the generative models we have seen
  - The posterior probability of class can be expressed in the form of a logistic function in the case of a binary classifier and a softmax function in the case of a *K*-class classifier
  - The contours of equal posterior probabilities of classes are hyperplanes in the input (feature) space yielding a linear classifier (for binary classification) or winner-take-all network (for *K*-ary classification)
- We just showed that the probability distributions underlying the generative models considered belong to the exponential family
- What can we say about the classifiers when the underlying generative models are distributions from the exponential family?

Classification problem for generic class conditional density from the exponential family

$$p(X \mid \eta) = h(X) e^{\left\{\eta^T G(X) - A(\eta)\right\}}$$

- **Consider *binary* classification task with density for class *0* and class *1* parameterized by *η0* and *η1* . Further assume *G(x)* is a linear function of *x* (before augmenting *x* with a 1)**

$$p(y=1 \mid \mathbf{x}, \boldsymbol{\eta}) = \frac{p(\mathbf{x} \mid y=1, \boldsymbol{\eta})\, p(y=1 \mid q)}{p(\mathbf{x} \mid y=1, \boldsymbol{\eta})\, p(y=1 \mid q) + p(\mathbf{x} \mid y=0, \boldsymbol{\eta})\, p(y=0 \mid q)}$$

$$= \frac{\exp\left\{\boldsymbol{\eta}_1^T G(\mathbf{x}) - A(\boldsymbol{\eta}_1)\right\} h(\mathbf{x}) q_1}{\exp\left\{\boldsymbol{\eta}_1^T G(\mathbf{x}) - A(\boldsymbol{\eta}_1)\right\} h(\mathbf{x}) q_1 + \exp\left\{\boldsymbol{\eta}_0^T G(\mathbf{x}) - A(\boldsymbol{\eta}_0)\right\} h(\mathbf{x}) q_0}$$

$$= \frac{1}{1 + \exp\left\{-(\boldsymbol{\eta}_0 - \boldsymbol{\eta}_1)^T G(\mathbf{x}) - A(\boldsymbol{\eta}_0) + A(\boldsymbol{\eta}_1) + \log \frac{q_0}{q_1}\right\}}$$

- **Note that this is a logistic function of a linear function of *x***

Classification problem for generic class conditional density from the exponential family

$$p(X \mid \eta) = h(X) e^{\left\{\eta^T G(X) - A(\eta)\right\}}$$

- **Consider *K*-ary classification task; Suppose *G(x)* is a linear function of *x***

$$p(y^k = 1 \mid \mathbf{x}, \boldsymbol{\eta}) = \frac{\exp\left\{\boldsymbol{\eta}_k^T G(\mathbf{x}) - A(\boldsymbol{\eta}_k)\right\} q_k}{\sum_{l=1}^{K} \exp\left\{\boldsymbol{\eta}_l^T G(\mathbf{x}) - A(\boldsymbol{\eta}_l)\right\} q_l}$$

$$= \frac{\exp\left\{\boldsymbol{\eta}_k^T G(\mathbf{x}) - A(\boldsymbol{\eta}_k) + \log q_k\right\}}{\sum_{l=1}^{K} \exp\left\{\boldsymbol{\eta}_l^T G(\mathbf{x}) - A(\boldsymbol{\eta}_l) + \log q_l\right\}}$$

which is a softmax function of a linear function of *x* !!

Summary

- A variety of class conditional densities all yield the same logistic-linear or softmax-linear (with respect to parameters) form for the posterior probability
- In practice, choosing a class conditional density can be difficult – especially in high dimensional spaces – e.g. multivariate Gaussian where the covariance matrix grows quadratically in the number of dimensions
- The invariance of the functional form of the posterior probability with respect to the choice of the distribution is good news!
- It is not necessary to specify the class conditional density at all if we can work directly with the posterior – which brings us to discriminative models!

Maximum margin classifiers

- Discriminative classifiers that maximize the margin of separation

- Support Vector Machines
  - Feature spaces
  - Kernel machines
  - VC theory and generalization bounds
  - Maximum margin classifiers

Perceptrons revisited

- Perceptrons
  - Can only compute threshold functions
  - Can only represent linear decision surfaces
  - Can only learn to classify linearly separable training data
- How can we deal with non linearly separable data?
  - Map data into a typically higher dimensional feature space where the classes become separable
- Two problems must be solved
  - Computational problem of working with high dimensional feature space
  - Overfitting problem in high dimensional feature spaces

Maximum margin model

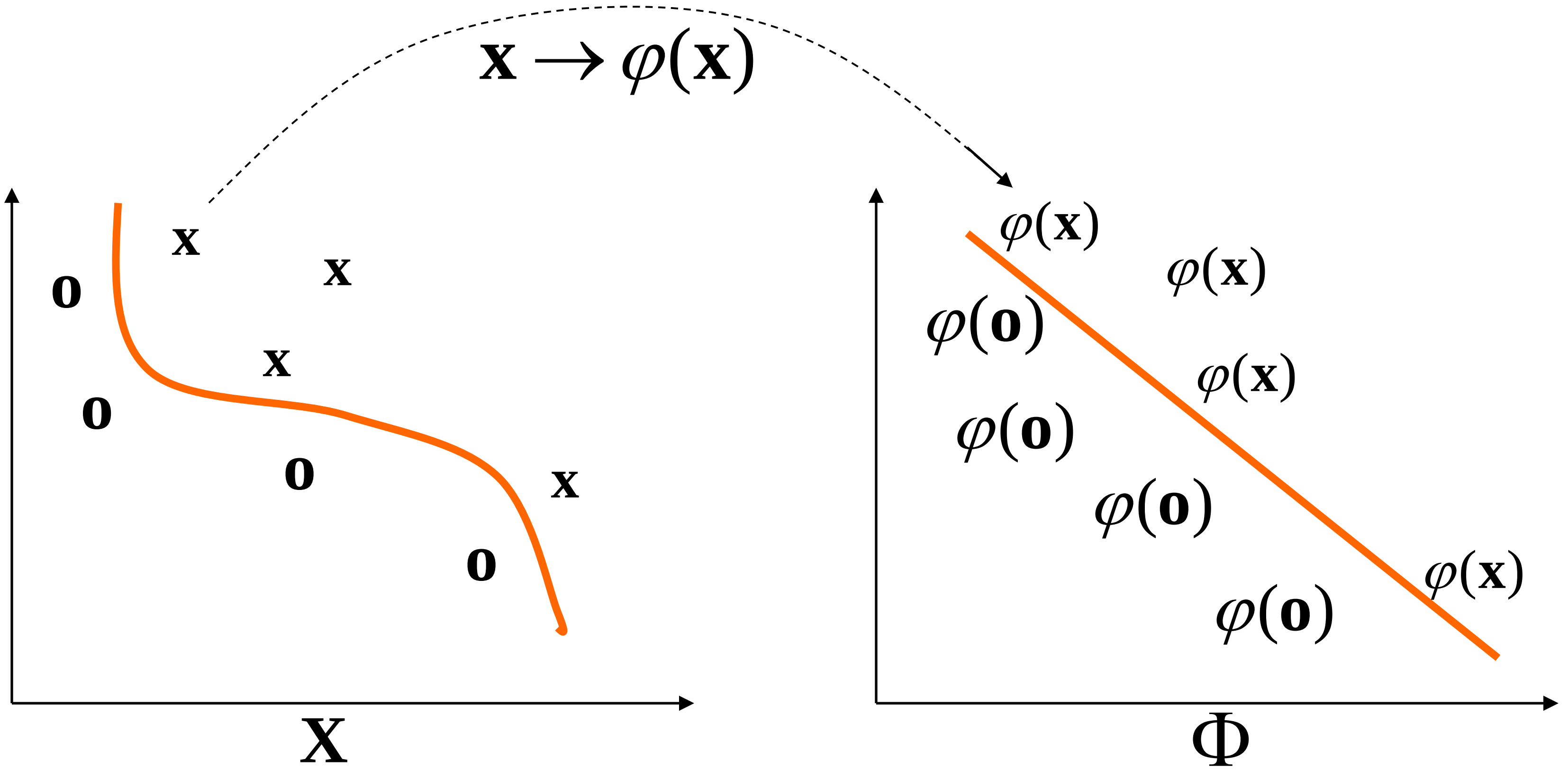
- We can not outperform Bayes optimal classifier when
  - *The generative model assumed is correct*
  - *The data set is large enough to ensure reliable estimation of parameters of the models*

- But discriminative models may be better than generative models when
  - *The correct generative model is seldom known*
  - *The data set is often simply not large enough*

- Maximum margin classifiers are a kind of discriminative classifiers designed to circumvent the overfitting problem

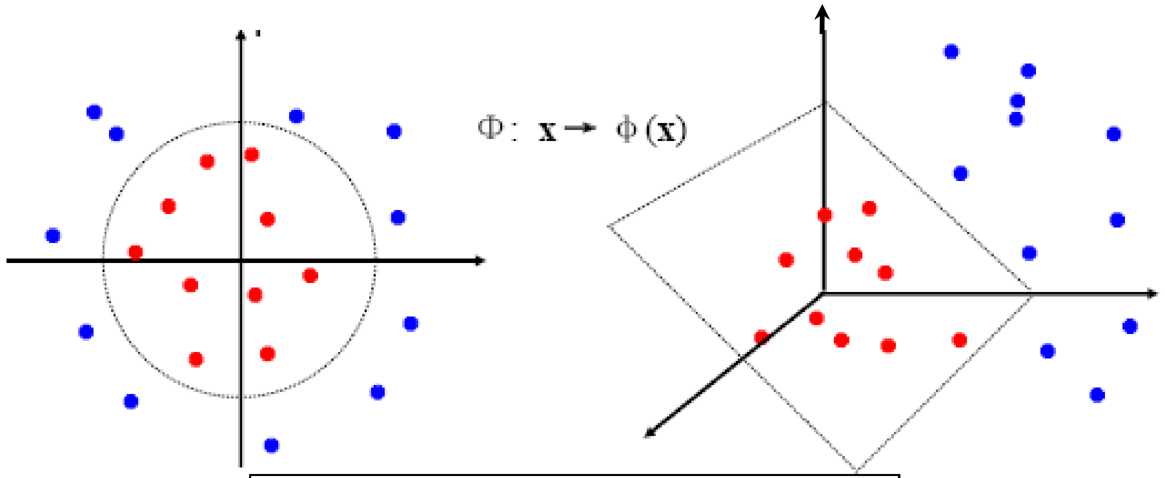
Extending Linear Classifiers: Learning in feature spaces

- Map data into a feature space where they are linearly separable



Linear Separability in Feature Spaces

- The original input space can always be mapped to some higher-dimensional feature space where the training data become separable:



$$\Phi : \mathbb{R}^2 \rightarrow \mathbb{R}^3$$
$$(x_1, x_2) \mapsto (z_1, z_2, z_3) := (x_1^2, \sqrt{2}\, x_1 x_2, x_2^2)$$

Learning in the Feature Space

- High dimensional feature spaces

$$\mathbf{X} = (x_1, x_2, \cdots, x_n) \rightarrow \varphi(\mathbf{X}) = (\varphi_1(\mathbf{X}), \varphi_2(\mathbf{X}), \cdots, \varphi_d(\mathbf{X}))$$

where typically *d* >> *n* solve the problem of expressing complex functions

- But this introduces
  - Computational problem (working with very large vectors)
  - Generalization problem (curse of dimensionality)

- SVM offer an elegant solution to both problems

The Perceptron Algorithm (primal form)

```
initialize $\mathbf{W}_0 \leftarrow 0, b_0 \leftarrow 0, k \leftarrow 0, \eta \in \Re^+$
  repeat
    error ← false
    for i = 1.. l
      if $y_i(\langle \mathbf{W}_k, \mathbf{X}_i \rangle + b_k) \le 0$ then
          $\mathbf{W}_{k+1} \leftarrow \mathbf{W}_k + \eta y_i \mathbf{X}_i$
          $b_{k+1} \leftarrow b_k + \eta y_i$
          $k \leftarrow k + 1$
          error ← true
      endif
    endfor
  until (error == false)
  return $k, (\mathbf{W}_k, b_k)$ where k is the number of mistakes
```

The Perceptron Algorithm Revisited

- The perceptron works by adding misclassified positive or subtracting misclassified negative examples to an arbitrary weight vector, which (without loss of generality) we assumed to be the zero vector

- So the final weight vector is a linear combination of training points

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

where, since the sign of the coefficient of $\mathbf{x}_i$ is given by label y_i, the α_i are positive values, proportional to the number of times, misclassification of $\mathbf{x}_i$ has caused the weight to be updated. It is called the *embedding strength* of the pattern $\mathbf{x}_i$

Dual Representation

- **The decision function can be rewritten as:**

$$h(\mathbf{X}) = \operatorname{sgn}\left(\langle \mathbf{W}, \mathbf{X} \rangle + b\right) = \operatorname{sgn}\left(\left\langle \left(\sum_{j=1}^{l} \alpha_j y_j \mathbf{X}_j\right), \mathbf{X} \right\rangle + b\right)$$

$$= \operatorname{sgn}\left(\sum_{j=1}^{l} \alpha_j y_j \langle \mathbf{X}_j, \mathbf{X} \rangle + b\right)$$

- **On training example** $(\mathbf{X}_i, y_i)$**, the update rule is:**

$$\text{if } y_i\left(\sum_{j=1}^{l} \alpha_j y_j \langle \mathbf{X}_j, \mathbf{X}_i \rangle + b\right) \le 0, \text{ then } \alpha_i = \alpha_i + \eta$$

- **WLOG, we can take** $\eta = 1$

Implication of Dual Representation

- When Linear Learning Machines are represented in the dual form

$$h(\mathbf{X}_i) = \operatorname{sgn}\left(\langle \mathbf{W}, \mathbf{X}_i \rangle + b\right) = \operatorname{sgn}\left(\sum_{j=1}^{l} \alpha_j y_j \langle \mathbf{X}_j, \mathbf{X}_i \rangle + b \right)$$

- Data appear only inside dot products (in decision function and in training algorithm)

- The matrix

$$G = \left(\langle \mathbf{X}_i, \mathbf{X}_j \rangle \right)_{i,j=1}^{l}$$

which is the matrix of pairwise dot products between training samples is called the *Gram matrix*

Implicit Mapping to Feature Space

- Kernel machines
  - Solve the computational problem of working with many dimensions
  - Can make it possible to use infinite dimensions efficiently
  - Offer other advantages, both practical and conceptual

Kernel-Induced Feature Space

$$f(\mathbf{X}_i) = \langle \mathbf{W}, \varphi(\mathbf{X}_i) \rangle + b$$
$$h(\mathbf{X}_i) = \operatorname{sgn}\left(f(\mathbf{X}_i) \right)$$

where $\varphi : \mathrm{X} \to \Phi$ is a non-linear map from input space to feature space

- **In the dual representation, the data points only appear inside dot products**

$$h(\mathbf{X}_i) = \operatorname{sgn}\left(\langle \mathbf{W}, \varphi(\mathbf{X}_i) \rangle + b \right) = \operatorname{sgn}\left(\sum_{j=1}^{l} \alpha_j y_j \langle \varphi(\mathbf{X}_j), \varphi(\mathbf{X}_i) \rangle + b \right)$$

Kernels

- Kernel function returns the value of the dot product between the *images* of the two arguments

$$K(x_1, x_2) = \langle \varphi(x_1), \varphi(x_2) \rangle$$

- When using kernels, the dimensionality of the feature space Φ is not necessarily important because of the special properties of kernel functions; We may not even know the map φ

- Given a function *K*, it is possible to verify that it is a kernel

Kernel Machines

- **We can use perceptron learning algorithm in the feature space by taking its dual representation and replacing dot products with kernels**

$$K(x_1, x_2) = \langle \varphi(x_1), \varphi(x_2) \rangle$$

The Kernel Matrix

- Kernel matrix is the Gram matrix in the feature space Φ (the matrix of pairwise dot products between feature vectors corresponding to the training samples)

K =

K(1,1)	*K(1,2)*	*K(1,3)*	…	*K(1,l)*
K(2,1)	*K(2,2)*	*K(2,3)*	…	*K(2,l)*
…	…	…	…	…
K(l,1)	*K(l,2)*	*K(l,3)*	…	*K(l,l)*

Properties of Kernel Matrices

- **It is easy to show that the Gram matrix (and hence the kernel matrix) is**
  - **A Square matrix**
  - **Symmetric** $(\mathbf{K}^T = \mathbf{K})$
  - **Positive semi-definite**
    - **all eigenvalues of *K* are non-negative – Recall that eigenvalues of a square matrix *A* are given by values of *λ* that satisfy |*A – λI*| = *0*, or**
    - $w^T \mathbf{K}\, w \geq 0$ **for all values of the vector w**

- **Any symmetric positive semi-definite matrix can be regarded as a kernel matrix, that is, as an inner product matrix in *some* feature space** Φ

$$\mathbf{K}(\mathbf{X}_i, \mathbf{X}_j) = \left\langle \varphi(\mathbf{X}_i), \varphi(\mathbf{X}_j) \right\rangle$$

Mercer's Theorem: Characterization of Kernel Functions

- **How to decide whether a function is a valid kernel (without explicitly constructing Φ)?**

- **A *function* $K : X \times X \rightarrow \Re$ is said to be (finitely) positive semi-definite if**
  - ***K* is a symmetric function: *K(x,y) = K(y,x)***
  - **Matrices formed by restricting *K* to any finite subset of the space *X* are positive semi-definite**

- ***Every (finitely) positive semi-definite, symmetric function is a kernel*: i.e. there exists a mapping φ such that it is possible to write**

$$K(\mathbf{X}_i, \mathbf{X}_j) = \left\langle \varphi(\mathbf{X}_i), \varphi(\mathbf{X}_j) \right\rangle$$

Examples of Kernels

- **Simple examples of kernels:**

$$K(\mathbf{X},\mathbf{Z}) = \langle \mathbf{X},\mathbf{Z} \rangle^d$$

$$K(X,Z) = e^{-\left(\frac{||X-Z||^2}{2\sigma^2}\right)}$$

Example: Polynomial Kernels

$$x = (x_1, x_2)$$

$$z = (z_1, z_2)$$

$$\begin{aligned}
\langle x, z \rangle^2 &= (x_1 z_1 + x_2 z_2)^2 \\
&= x_1^2 z_1^2 + x_2^2 z_2^2 + 2 x_1 z_1 x_2 z_2 \\
&= \left\langle \left(x_1^2, x_2^2, \sqrt{2} x_1 x_2 \right), \left(z_1^2, z_2^2, \sqrt{2} z_1 z_2 \right) \right\rangle \\
&= \langle \varphi(x), \varphi(z) \rangle
\end{aligned}$$

Making Kernels – Closure Properties

- The set of kernels is closed under some operations; If ***K₁***, ***K₂*** are kernels over $X \times X$, then the following are kernels:

$$K(\mathbf{X},\mathbf{Z}) = K_1(\mathbf{X},\mathbf{Z}) + K_2(\mathbf{X},\mathbf{Z})$$

$$K(\mathbf{X},\mathbf{Z}) = aK_1(\mathbf{X},\mathbf{Z}); a > 0$$

$$K(\mathbf{X},\mathbf{Z}) = K_1(\mathbf{X},\mathbf{Z})K_2(\mathbf{X},\mathbf{Z})$$

$$K(\mathbf{X},\mathbf{Z}) = f(\mathbf{X})f(\mathbf{Z}); f : X \to \Re$$

$$K(\mathbf{X},\mathbf{Z}) = K_3(\varphi(\mathbf{X}),\varphi(\mathbf{Z}));$$

$$(\varphi : X \to \Re^N; K_3 \text{ is a kernel over } \Re^N \times \Re^N)$$

$$K(\mathbf{X},\mathbf{Z}) = \mathbf{X}^T\mathbf{B}\mathbf{Z};$$

$$(B \text{ is a symmetric positive definite } n \times n \text{ matrix and } X \subseteq \Re^N)$$

- We can make complex kernels from simple ones: modularity!

Kernels

- We can define kernels over arbitrary instance spaces including
  - finite dimensional vector spaces
  - Boolean spaces
  - $\sum^*$ where $\sum$ is a finite alphabet
  - Documents, graphs, etc.

- Applied in text categorization, bioinformatics,…

- Kernels need not always be expressed by a closed form formula

- Many useful kernels can be computed by complex algorithms (e.g. diffusion kernels over graphs)

String Kernel (*p*-spectrum kernel)

- The *p*-spectrum of a string is the histogram – vector of number of occurrences of all possible contiguous substrings – of length *p*
- We can define a kernel function *K(s,t)* over $\sum^* \times \sum^*$ as the inner product of the *p*-spectra of *s* and *t*

s = statistics

t = computation

p = 3

Common substrings: *tat, ati*

K(s,t) = 2

Kernel over sets

Let $A, A' \in 2^V$ for some fixed domain V. Then the following is a kernel

$$K(A, A') = 2^{|A \cap A'|}$$

Kernel Machines

- Kernel machines are Linear Learning Machines, that:
  - Use a dual representation
  - Operate in a kernel induced feature space (that is a linear function in the feature space implicitly defined by the Gram matrix corresponding to the data set)

$$h(\mathbf{X}_i) = \operatorname{sgn}\left(\langle \mathbf{W}, \varphi(\mathbf{X}_i) \rangle + b\right) = \operatorname{sgn}\left(\sum_{j=1}^{l} \alpha_j y_j \langle \varphi(\mathbf{X}_j), \varphi(\mathbf{X}_i) \rangle + b \right)$$

Kernels – the good, the bad, and the ugly

- **Bad kernel – A kernel whose Gram (kernel) matrix is mostly diagonal – all data points are orthogonal to each other, and hence the machine is unable to detect hidden structure in the data**

1	0	0	…	0
0	1	0	…	0
		1		
…	…	…	…	…
0	0	0	…	1

Kernels – the good, the bad, and the ugly

- **Good kernel – Corresponds to a Gram (kernel) matrix in which subsets of data points belonging to the same class are similar to each other, and hence the machine can detect hidden structure in the data**

3	2	0	0	0
2	3	0	0	0
0	0	4	3	3
0	0	3	4	2
0	0	3	2	4

🔴	*Class 1*
🔵	*Class 2*

Kernels – the good, the bad, and the ugly

- The kernel expresses similarity between two data points
- In mapping in a space with too may irrelevant features, kernel matrix becomes diagonal
- Need some prior knowledge of target to choose a good kernel

Learning in the Feature Space

- High dimensional feature spaces

$$\mathbf{X} = (x_1, x_2, \cdots, x_n) \rightarrow \varphi(\mathbf{X}) = (\varphi_1(\mathbf{X}), \varphi_2(\mathbf{X}), \cdots, \varphi_d(\mathbf{X}))$$

  where typically *d* >> *n* solve the problem of expressing complex functions

- But this introduces
  - computational problem (working with very large vectors)

    → solved by the kernel trick – implicit computation of dot products in kernel induced feature spaces via dot products in the input space
  - generalization theory problem (curse of dimensionality)

Kernel Substitution:

- **Kernel trick**
  - **Extend many well-known algorithms**
  - **If an algorithm is formulated in such a way that *X* enters only in the form of scalar products, then replace that with kernel**
  - **E.g. nearest-neighbor classifiers, PCA**

The Generalization Problem

- The curse of dimensionality
  - It is easy to overfit in high dimensional spaces

- The learning problem is ill posed
  - There are infinitely many hyperplanes that separate the training data
  - Need a principled approach to choose an optimal hyperplane

The Generalization Problem

- "Capacity" of the machine – ability to learn any training set without error – related to VC dimension
- Excellent memory is not an asset when it comes to learning from limited data
- "A machine with too much capacity is like a botanist with a photographic memory who, when presented with a new tree, concludes that it is not a tree because it has a different number of leaves from anything she has seen before; a machine with too little capacity is like the botanist's lazy brother, who declares that if it's green, it's a tree" – C. Burges

History of Key Developments leading to SVM

- 1958: Perceptron (Rosenblatt)
- 1963: Margin (Vapnik)
- 1964: Kernel Trick (Aizerman)
- 1965: Optimization formulation (Mangasarian)
- 1971: Kernels (Wahba)
- 1992-1994: SVM (Vapnik)
- 1996 – : Rapid growth, numerous applications, extensions to other problems

A Little Learning Theory

- **Suppose:**
  - **We are given *l* training examples *(X_i, y_i)***
  - **Train and test points drawn randomly (i.i.d.) from some unknown probability distribution *D(X, y)***

- **The machine learns the mapping *X_i → y_i* and outputs a hypothesis *h(X, α, b)*; A particular choice of (*α, b)* generates "trained machine"**

- **The expectation of the test error or *expected risk* is**

$$R(\boldsymbol{\alpha}, b) = \frac{1}{2}\int | y - h(\mathbf{X}, \boldsymbol{\alpha}, b) | \, dD(\mathbf{X}, y)$$

A Bound on the Generalization Performance

- The *empirical risk* is:

$$R_{emp}(\boldsymbol{\alpha}, b) = \frac{1}{2l}\sum_{i=1}^{l} | y - h(\mathbf{X}, \boldsymbol{\alpha}, b) |$$

- Choose some ***δ*** such that ***0 < δ < 1***; With probability ***1- δ*** the following bound – risk bound of ***h(X, α)*** in distribution ***D*** holds (Vapnik, 1995)

$$R(\boldsymbol{\alpha}, b) \le R_{emp}(\boldsymbol{\alpha}, b) + \sqrt{\left(\frac{d\left(\log\left(2l/d\right)+1\right)-\log\left(\delta/4\right)}{l}\right)}$$

where $d \ge 0$ is called VC dimension, a measure of "capacity" of machine

A Bound on the Generalization Performance

- The second term in the right-hand side is called *VC confidence*

- Three key points about the actual risk bound:
  - It is independent of *D(X, y)*
  - It is usually not possible to compute the left-hand side
  - If we know *d*, we can compute the right-hand side

- The risk bound gives us a way to compare learning machines!

The Vapnik-Chervonenkis (VC) Dimension

- **<u>Definition:</u>**

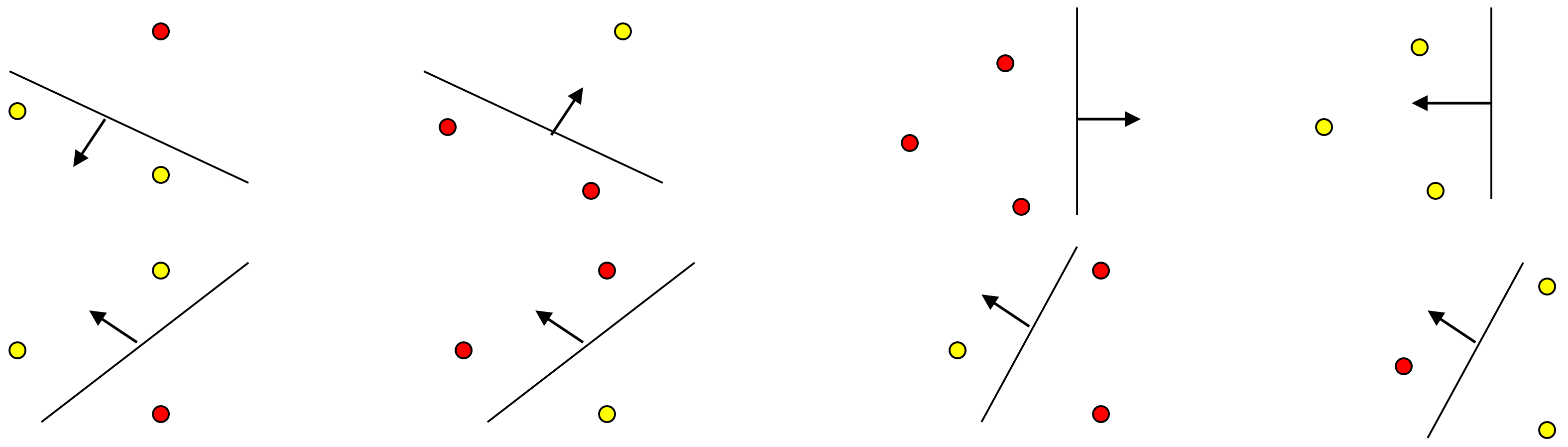
  The VC dimension of a set of functions $H = \{h(X, \alpha, b)\}$ is d if and only if there exists a set of d data points such that each of the 2^d possible labelings (dichotomies) of the d data points, can be realized using some member of H but that no set exists with $q > d$ satisfying this property

The VC Dimension

- **A set *S* of instances is said to be *shattered* by a hypothesis class *H* if and only if for *every* dichotomy of *S*, there exists a hypothesis in *H* that is consistent with the dichotomy**
- **VC dimension of *H* is size of largest subset of *X* shattered by *H***
- **VC dimension measures the capacity of a set *H* of hypotheses (functions)**
- **If for any number *N*, it is possible to find *N* points $X_1, \ldots, X_N$ that can be separated in all the 2^N possible ways, we will say that the VC-dimension of the set is infinite**

The VC Dimension Example

- Suppose that the data live in $\Re^2$ space, and the set ***{h(X, α)}*** consists of oriented straight lines (linear discriminants)
- It is possible to find three points that can be shattered by this set of functions; It is not possible to find four
- So the VC dimension of the set of linear discriminants in $\Re^2$ is three



The VC Dimension

- **VC dimension can be infinite even when the number of parameters of the set *{h(X, α)}* of hypothesis functions is low**

- **Example: *{h(X, α)} ≡ sgn(sin(αX)), X,α*** $\in \Re$

  For any integer *l* with any labels $y_1, \cdots y_l,\ y_i \in \{-1, 1\}$ **, we can find *l* points** $X_1, \cdots X_l$ **and parameter *α* such that those points can be shattered by *h(X, α)***

  Those points are: $x_i = 10^{-i}, \quad i = 1, \ldots, l.$

  and parameter *α* is:

$$\alpha = \pi\left(1 + \sum_{i=1}^{l} \frac{(1 - y_i)10^i}{2}\right)$$

VC Dimension of a Hypothesis Class

- **Definition: The VC dimension *V*(*H*), of a hypothesis class *H* defined over an instance space *X* is the cardinality *d* of the largest subset of *X* that is shattered by *H*. If arbitrarily large finite subsets of *X* can be shattered by *H*, *V*(*H*) = ∞**

- **How can we show that *V*(*H*) is at least *d*?**

  → Find a set of cardinality at least *d* that is shattered by *H*

- **How can we show that *V*(*H*) = *d*?**

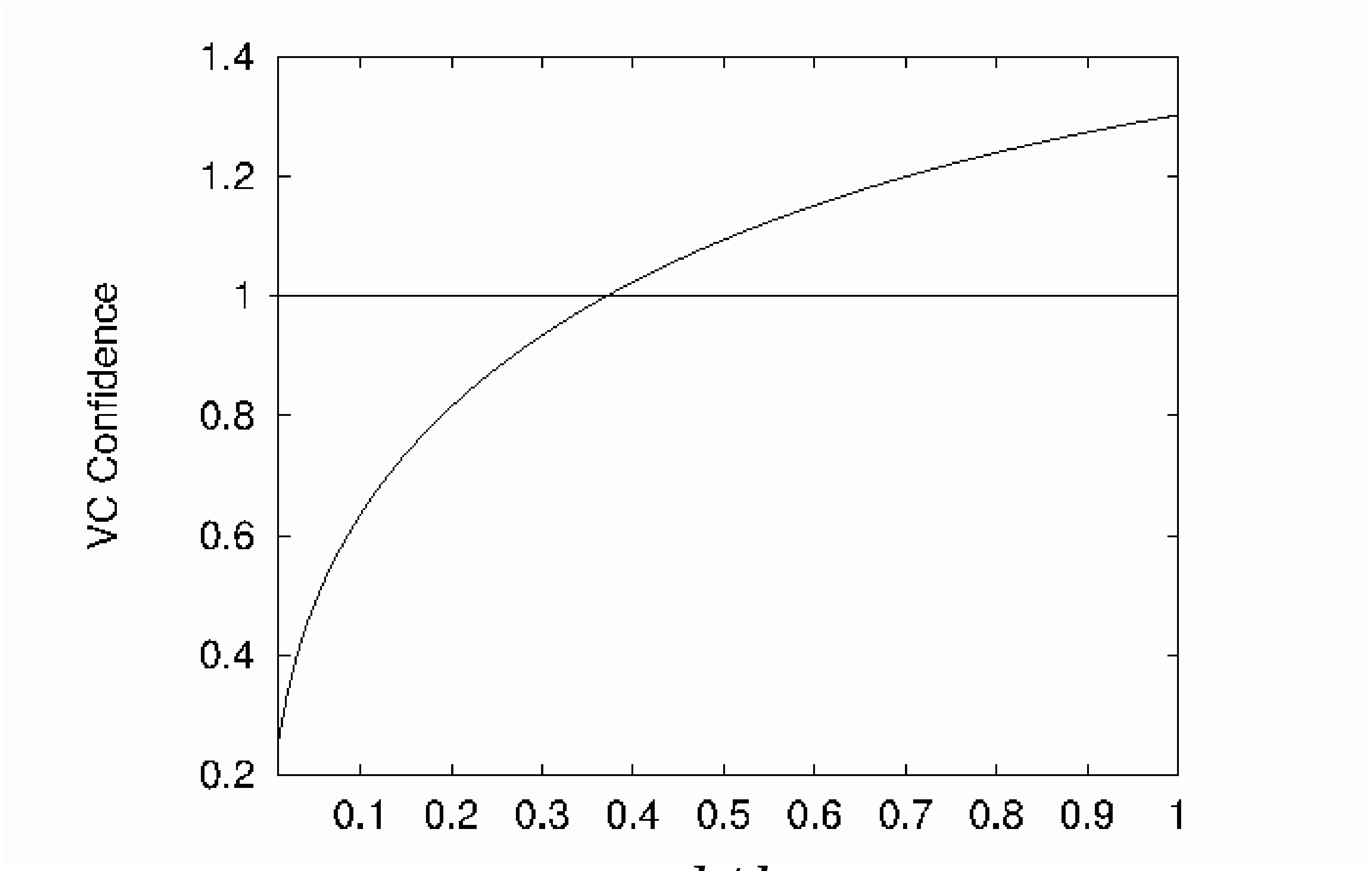
  → Show that *V*(*H*) is at least *d* and no set of cardinality (*d*+1) can be shattered by *H*

Minimizing the Bound by Minimizing *d*

$$R(\boldsymbol{\alpha},b) \leq R_{emp}(\boldsymbol{\alpha},b) + \sqrt{\left(\frac{d\left(\log\left(2l/d\right)+1\right)-\log\left(\delta/4\right)}{l}\right)}$$

- VC confidence (second term) depends on *d/l*
- One should choose that learning machine whose set of functions has minimal *d*
- For large values of *d/l* the bound is not tight

Minimizing the Risk Bound by Minimizing *d*



$d\,/\,l$

Bounds on Error of Classification

- **Vapnik proved that the error *ε* of classification function *h* for separable data sets is**

$$\varepsilon = O\left(\frac{d}{l}\right)$$

where *d* is the VC dimension of the hypothesis class and *l* is the number of training examples

- **The classification error**
  - **Depends on the VC dimension of the hypothesis class**
  - **Is independent of the dimensionality of the feature space**

Structural Risk Minimization

- **Finding a learning machine with the minimum upper bound on the actual risk**
  - **leads us to a method of choosing an optimal machine for a given task**
  - **essential idea of the structural risk minimization (SRM)**

- **Let** $H_1 \subset H_2 \subset H_3 \subset \cdots$ **be a sequence of nested subsets of hypotheses whose VC dimensions satisfy** $d_1 < d_2 < d_3 < \ldots$
  - **SRM consists of finding that subset of functions which minimizes the upper bound on the actual risk**
  - **SRM involves training a series of classifiers, and choosing a classifier from the series for which the sum of empirical risk and VC confidence is minimal**

Margin Based Bounds on Error of Classification

- The error ***ε*** of classification function *h* for separable data sets is

$$\varepsilon = O\left(\frac{d}{l}\right)$$

- Can prove margin based bound:

$$\varepsilon = O\left(\frac{1}{l}\left(\frac{L}{\bar{\gamma}}\right)^2\right)$$

$$L = \max_p \left\| \mathbf{X}_p \right\|$$

$$\bar{\gamma} = \min_i \frac{y_i f(\mathbf{x}_i)}{\|\mathbf{w}\|}$$

$$f(\mathbf{x}) = \langle \mathbf{w}, \mathbf{x} \rangle + b$$

- Important insight:

Error of the classifier trained on a separable data set is inversely proportional to its margin, and is independent of the dimensionality of the input space!

Maximal Margin Classifier

- The bounds on error of classification suggest the possibility of *improving generalization by maximizing the margin*

- Minimize the risk of overfitting by *choosing the maximal margin hyperplane in feature space*

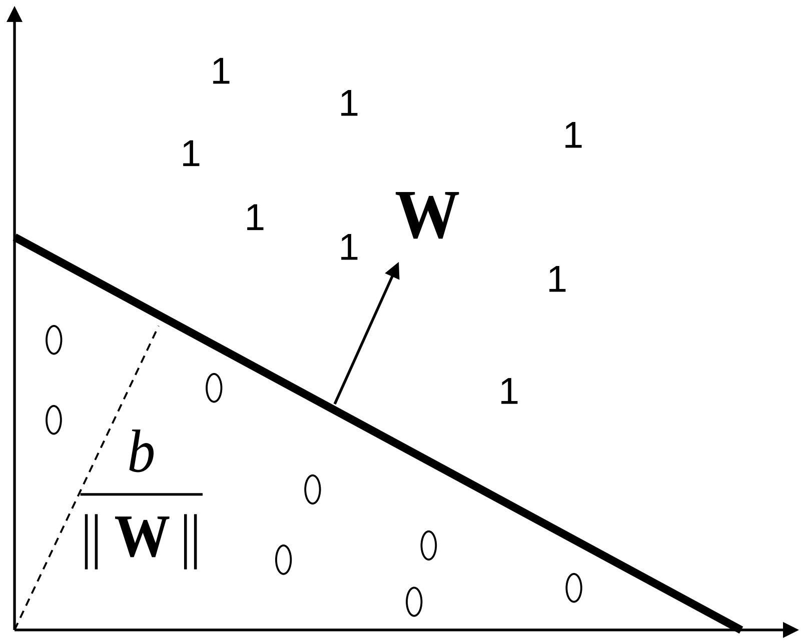
- *SVMs control capacity by increasing the margin, not by reducing the number of features*

Margin

w

- **Linear separation of the input space**

$$f(\mathbf{X}) = <\mathbf{W}, \mathbf{X}> + b$$
$$h(\mathbf{X}) = sign\left(f(\mathbf{X})\right)$$



Functional and Geometric Margin

- The functional margin of a linear discriminant (*w*,*b*) w.r.t. a labeled pattern $(x_i, y_i) \in \mathbf{R}^d \times \{-1,1\}$ is defined as

$$\gamma_i \equiv y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b)$$

- If the functional margin is negative, then the pattern is incorrectly classified, if it is positive then the classifier predicts the correct label

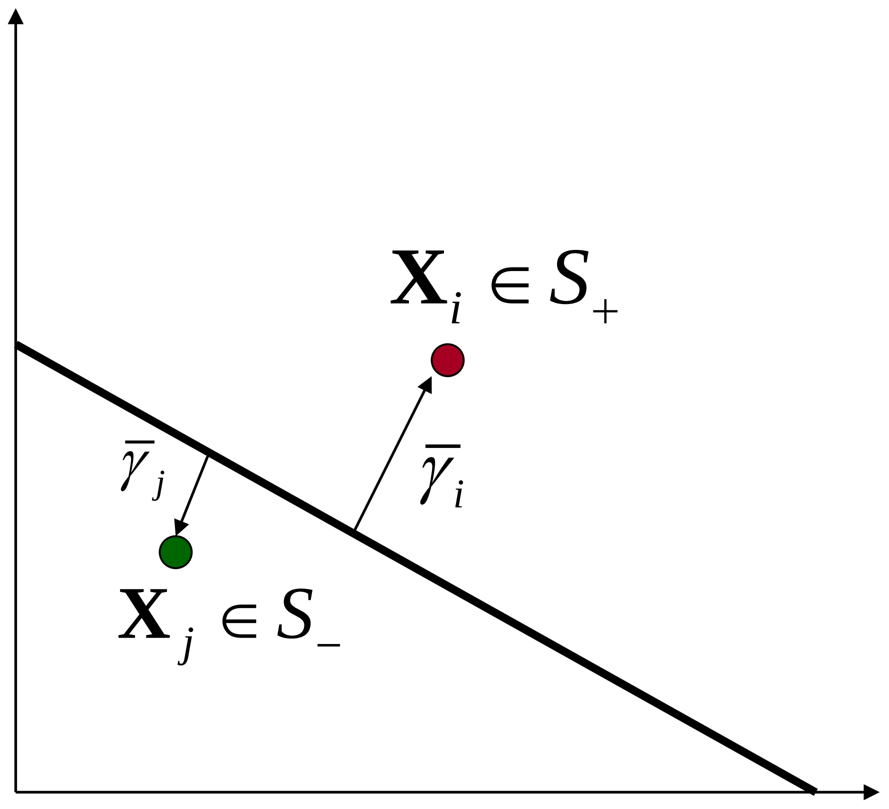
- The larger $|\gamma_i|$, the further away *Xi* is from the discriminant

- This is made more precise in the notion of the geometric margin

$$\bar{\gamma}_i \equiv \frac{\gamma_i}{\|\mathbf{w}\|}$$

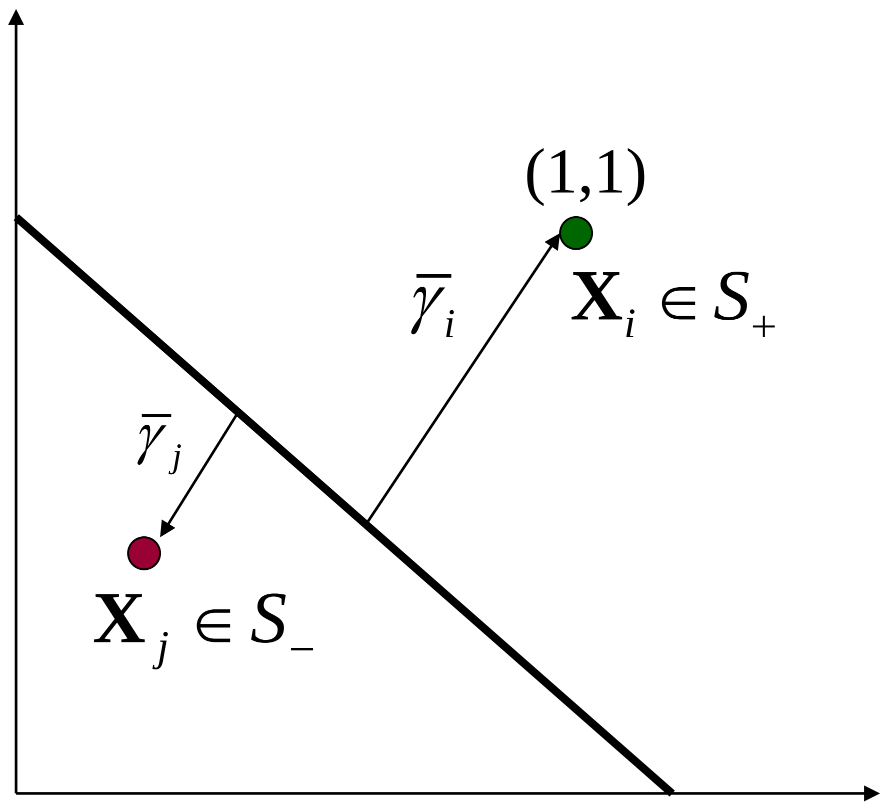
which measures the Euclidean distance of a point from the decision boundary

Geometric Margin



The geometric margin of two points

Geometric Margin



Example

$$\mathbf{W} = (1 \quad 1)$$

$$b = -1$$

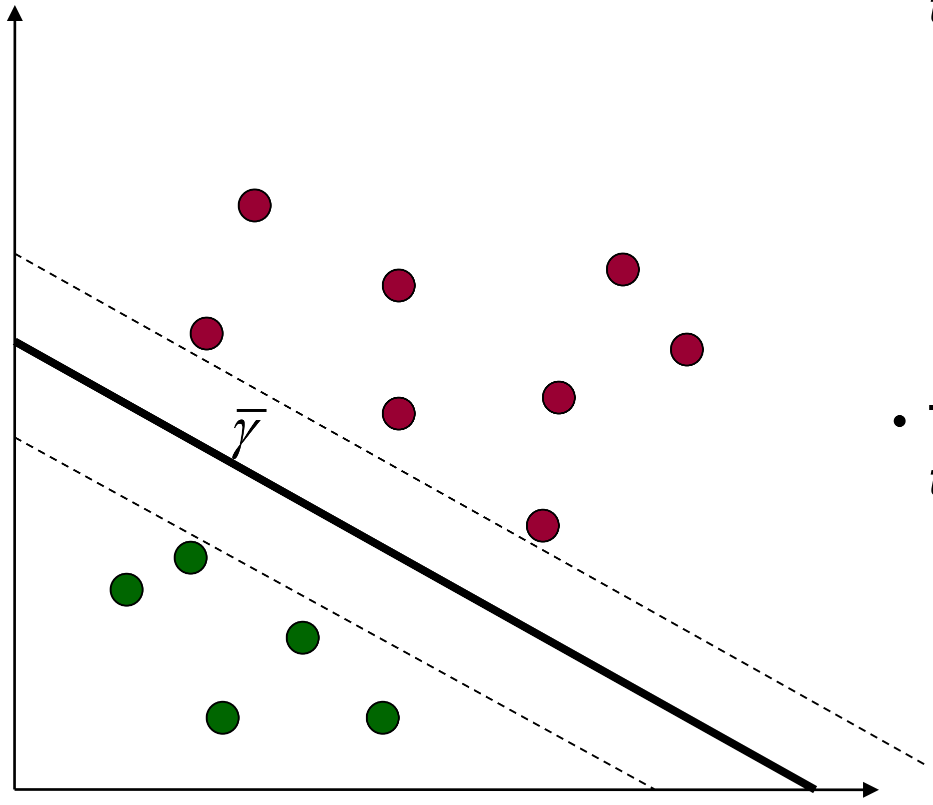
$$\mathbf{X}_i = (1 \quad 1)$$

$$Y_i = 1$$

$$\begin{aligned}\gamma_i &= (Y_i)((1)x_1 + (1).x_2 - 1) \\ &= (1)(1+1-1) = 1\end{aligned}$$

$$\bar{\gamma}_i = \frac{\gamma_i}{\|\mathbf{W}\|} = \frac{1}{\sqrt{2}}$$

Margin of a Training Set



- The functional margin of a *training set*

$$\gamma = \min_i \gamma_i$$

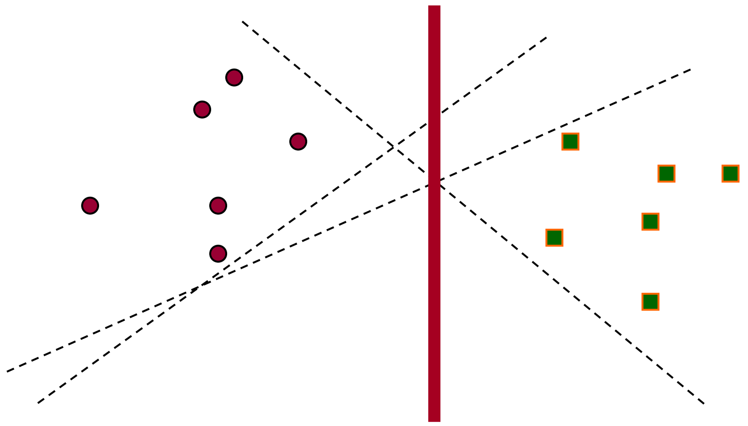
- The geometric margin of a *training set*

$$\bar{\gamma} = \min_i \bar{\gamma}_i$$

Maximum Margin Separating Hyperplane

- $\gamma = \min_i \gamma_i$ **is called the (functional) margin of (*W*,*b*) w.r.t. the data set *S*={(*Xi*,*yi*)}**

- **The margin of a training set *S* is the maximum geometric margin over all hyperplanes; A hyperplane realizing this maximum is a maximal margin hyperplane**



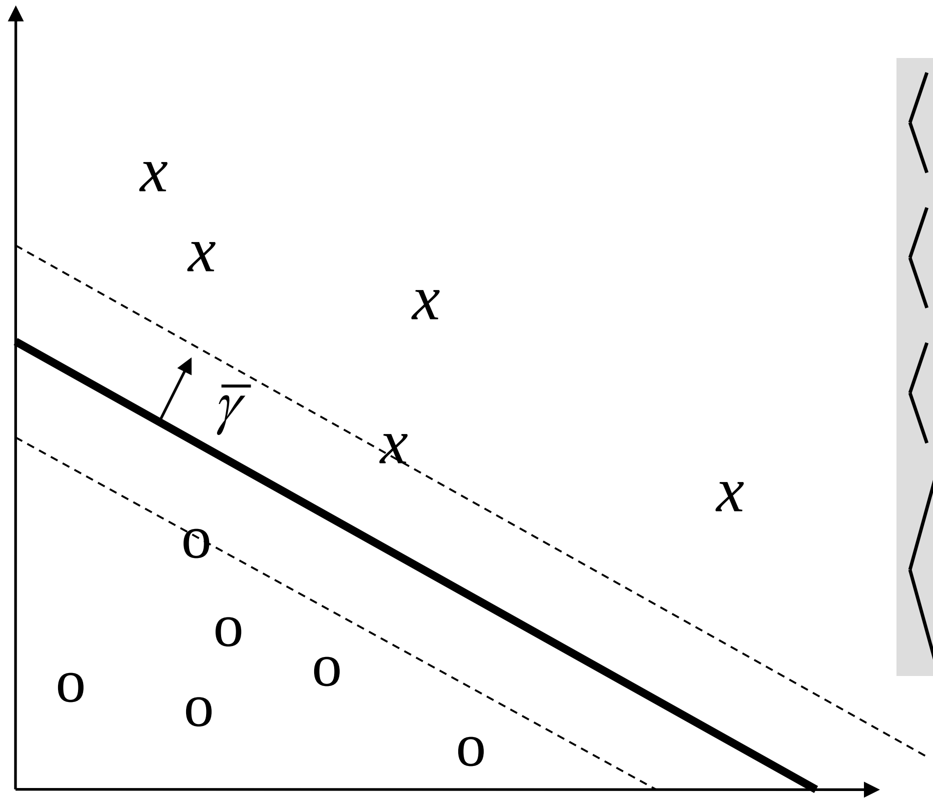
Maximal Margin Hyperplane

Maximizing Margin → Minimizing ||*W*||

- Definition of hyperplane (*W,b*) does not change if we rescale it to (σ*W*, σ*b*), for $\sigma > 0$.
- Functional margin depends on scaling, but geometric margin $\bar{\gamma}$ does not
- If we fix (by rescaling) the functional margin to 1, the geometric margin will be equal 1/||*W*||
- Then, we can maximize the margin by minimizing the norm ||*W*||

Maximizing Margin → Minimizing ||*W*||

- Let $\mathbf{X}^+$ and $\mathbf{X}^-$ be the nearest data points in the sample space. Then we have, by fixing the functional margin to be 1:



$$\langle w, x^+ \rangle + b = +1$$

$$\langle w, x^- \rangle + b = -1$$

$$\langle w, (x^+ - x^-) \rangle = 2$$

$$\left\langle \frac{w}{\| w \|}, (x^+ - x^-) \right\rangle = \frac{2}{\| w \|}$$

Learning as Optimization

- Minimize

$$\langle \mathbf{W}, \mathbf{W} \rangle$$

Subject to:

$$y_i\left(\langle \mathbf{W}, \mathbf{X}_i \rangle + b\right) \geq 1$$

- The problem of finding the maximal margin hyperplane: constrained optimization (quadratic programming) problem

Digression – Minimizing/Maximizing Functions

Consider $f(x)$, a function of a scalar variable x with domain D_x

$f(x)$ is convex over some sub-domain $D \subseteq D_x$ if $\forall X_1, X_2 \in D$,

the chord joining the points $f(X_1)$ and $f(X_2)$ lies above

the graph of $f(x)$

$f(x)$ has a local minimum at $x = X_a$ if $\exists$ neighborhood $U \subseteq D_x$ around

X_a such that $\forall x \in U,\ f(x) > f(X_a)$

We say that $\lim_{x \to a} f(x) = A$ if, for any $\varepsilon > 0$, $\exists \delta > 0$ such that $|f(x) - A| < \varepsilon$

$\forall x$ such that $|x\text{-}a| < \delta$

Minimizing/Maximizing Functions

We say that $f(x)$ is continuous at $x = a$

if $\lim_{\varepsilon \to 0} \left\{ \lim_{x \to a+\in} f(x) \right\} = \lim_{\varepsilon \to 0} \left\{ \lim_{x \to a-\in} f(x) \right\}$

The derivative of the function $f(x)$ is defined as

$$\frac{df}{dx} = \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{(\Delta x)}$$

$$\left.\frac{df}{dx}\right|_{x=X_0} = 0 \text{ if } X_0 \text{ is a local maximum or a local minimum}$$

Minimizing/Maximizing Functions

$$\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx}$$

$$\frac{d(uv)}{dx} = u\frac{dv}{dx} + v\frac{du}{dx}$$

$$\frac{d\left(\frac{u}{v}\right)}{dx} = \frac{v\left(\frac{du}{dx}\right) - u\left(\frac{dv}{dx}\right)}{v^2}$$

Taylor Series Approximation of Functions

Taylor series approximation of $f(x)$

If $f(x)$ is differentiable i.e., its derivatives

$\frac{df}{dx}, \frac{d^2 f}{dx^2} = \frac{d}{dx}\left(\frac{df}{dx}\right), \ldots \frac{d^n f}{dx^n}$ exist at $x = X_0$ and

$f(x)$ is continuous in the neighborhood of $x = X_0$, then

$$f(x) = f(X_0) + \left(\left.\frac{df}{dx}\right|_{x=X_0}\right)(x - X_0) + \ldots.. + \frac{1}{n!}\left(\left.\frac{df^n}{dx^n}\right|_{x=X_0}\right)(x - X_0)^n$$

$$f(x) \approx f(X_0) + \left(\left.\frac{df}{dx}\right|_{x=X_0}\right)(x - X_0)$$

Chain Rule

Let $f(\mathbf{X}) = f(x_{0,} x_1, x_2, ...x_n)$

$\frac{\partial f}{\partial x_i}$ is obtained by treating all $x_i \big| i \neq j$ as constant.

Chain rule

Let $z = \varphi(u_1....u_m)$

Let $u_i = f_i(x_{0,} x_1......x_n)$

Then $\forall k \ \frac{\partial z}{\partial x_k} = \sum_{i=1}^{m} \left(\frac{\partial z}{\partial u_i} \right) \left(\frac{\partial u_i}{\partial x_k} \right)$

Taylor Series Approximation of Multivariate Functions

Let $f(\mathbf{X}) = f(x_{0,} x_1, x_2, ...x_n)$ be differentiable and continuous at $\mathbf{X}_0 = (x_{00,} x_{10}, x_{20}, ...x_{n0})$

Then

$$f(\mathbf{X}) \approx f(\mathbf{X}_0) + \sum_{i=0}^{n} \left(\frac{\partial f}{\partial x}\right)\bigg|_{\mathbf{X}=\mathbf{X}_0} (x_i - x_{i0})$$

Minimizing/Maximizing Multivariate Functions

To find $\mathbf{X}^*$ that minimizes $f(\mathbf{X})$, we change current guess $\mathbf{X}^C$ in the direction of the negative gradient of $f(\mathbf{X})$ evaluated at $\mathbf{X}^C$

$$\mathbf{X}^C \leftarrow \mathbf{X}^C - \eta \left(\frac{\partial f}{\partial x_0}, \frac{\partial f}{\partial x_1} \ldots\ldots\ldots\ldots \frac{\partial f}{\partial x_n} \right) \Bigg|_{\mathbf{X}=\mathbf{X}^C}$$ (why?)

for small (ideally infinitesimally small)

Minimizing/Maximizing Multivariate Functions

Suppose we more from Z_0 to Z_1. We want to ensure $f(Z_1) \le f(Z_0)$.

In the neighborhood of Z_0, using Taylor series expansion, we can write

$$f(Z_1) = f(Z_0 + \Delta Z) = f(Z_0) + \left(\left. \frac{df}{dZ} \right|_{Z=Z_0} \right) (\Delta Z) + \ldots$$

$$\Delta f = f(Z_1) - f(Z_0) \approx \left(\left. \frac{df}{dZ} \right|_{Z=Z_0} \right) (\Delta Z)$$

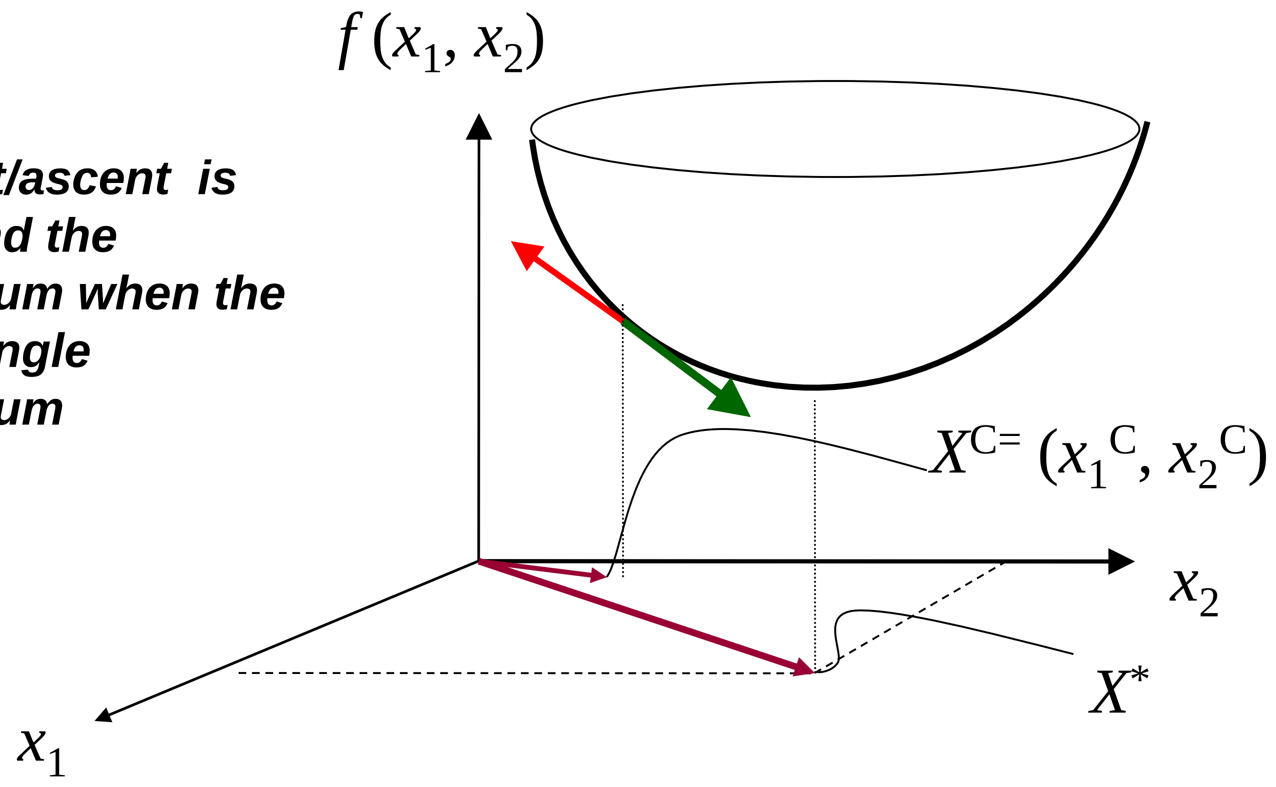
We want to make sure $\Delta f \le 0$.

If we choose

$$\Delta Z = -\eta \left(\left. \frac{df}{dZ} \right|_{Z=Z_0} \right) \rightarrow \Delta f = -\eta \left(\left. \frac{df}{dZ} \right|_{Z=Z_0} \right)^2 \le 0$$

Minimizing/Maximizing Functions

Gradient descent/ascent is guaranteed to find the minimum/maximum when the function has a single minimum/maximum



Constrained Optimization

- **Primal optimization problem**

- **Given functions *f, g_i*, *i=1…k; h_j, j=1..m*; defined on a domain** $\Omega \subseteq \Re^n$

$$\begin{array}{llll} \text{minimize} & f(\mathbf{w}) & \mathbf{w} \in \Omega & \{ \text{objective function} \\ \text{subject to} & g_i(\mathbf{w}) \leq 0 & i = 1 \ldots k, & \{ \text{inequality constraints} \\ & h_j(\mathbf{w}) = 0 & j = 1 \ldots m & \{ \text{equality constraints} \end{array}$$

- **Shorthand**

$$\begin{array}{llll} g(\mathbf{w}) \leq 0 & \text{denotes} & g_i(\mathbf{w}) \leq 0 & i = 1 \ldots k \\ h(\mathbf{w}) = 0 & \text{denotes} & h_j(\mathbf{w}) = 0 & j = 1 \ldots m \end{array}$$

- **Feasible region**

$$F = \{ \mathbf{w} \in \Omega : g(\mathbf{w}) \leq 0, h(\mathbf{w}) = 0 \}$$

Optimization Problems

- **Linear program – objective function as well as equality and inequality constraints are linear**
- **Quadratic program – objective function is quadratic, and the equality and inequality constraints are linear**
- **Inequality constraints $g_i(w) \leq 0$ can be active i.e. $g_i(w) = 0$ or inactive i.e. $g_i(w) < 0$**
- **Inequality constraints are often transformed into equality constraints using slack variables**

$$g_i(w) \leq 0 \leftrightarrow g_i(w) + \xi_i = 0 \text{ with } \xi_i \geq 0$$

- **We will be interested primarily in convex optimization problems**

Convex Optimization Problem

- If function *f* is convex, any local minimum $\mathbf{w}^*$ of an unconstrained optimization problem with objective function *f* is also a global minimum, since for any $\mathbf{u} \neq \mathbf{w}^*$ $f(\mathbf{w}^*) \leq f(\mathbf{u})$

- A set $\Omega \subset \mathbf{R}^n$ is called convex if , $\forall \mathbf{w}, \mathbf{u} \in \Omega$ and for any $\theta \in (0,1)$, the point $(\theta \mathbf{w} + (1-\theta)\mathbf{u}) \in \Omega$

- A convex optimization problem is one in which the set Ω , the objective function and all the constraints are convex

Lagrangian Theory

- **Given an optimization problem with an objective function *f(w)* and *equality* constraints *h_j(w) = 0 j = 1..m*, we define the Lagrangian function as**

$$L(\mathbf{w},\boldsymbol{\beta}) = f(\mathbf{w}) + \sum_{j=1}^{m} \beta_j h_j(\mathbf{w})$$

where β_j are called the Lagrange multipliers. A necessary condition for *w to be minimum of *f(w)* subject to the constraints *h_j(w) = 0 j = 1..m* is given by**

$$\frac{\partial L(\mathbf{w}^*,\boldsymbol{\beta}^*)}{\partial \mathbf{w}} = 0, \quad \frac{\partial L(\mathbf{w}^*,\boldsymbol{\beta}^*)}{\partial \boldsymbol{\beta}} = 0$$

- **The condition is sufficient if *L(w,β*)* is a convex function of *w***

Lagrangian Theory – Example

minimize $\quad f(x,y) = x + 2y$

subject to $\quad x^2 + y^2 - 4 = 0$

$$L(x,y,\lambda) = x + 2y + (x^2 + y^2 - 4)\lambda$$

$$\frac{\partial L(x,y,\lambda)}{\partial x} = 1 + 2\lambda x = 0$$

$$\frac{\partial L(x,y,\lambda)}{\partial y} = 2 + 2\lambda y = 0$$

$$\frac{\partial L(x,y,\lambda)}{\partial \lambda} = x^2 + y^2 - 4 = 0$$

Solving the above, we have:

$$\lambda = \pm\frac{\sqrt{5}}{4}, x = \mp\frac{2}{\sqrt{5}}, y = \mp\frac{4}{\sqrt{5}}$$

f is minimized when $x = -\frac{2}{\sqrt{5}}, y = -\frac{4}{\sqrt{5}}$

Lagrangian Theory – Example

- **Find the lengths *u*, *v*, *w* of sides of the box that has the largest volume for a given surface area *c***

$$\text{minimize} \quad -uvw$$

$$\text{subject to} \quad wu + uv + vw = \frac{c}{2}$$

$$L = -uvw + \beta\left(wu + uv + vw - \frac{c}{2} \right)$$

$$\frac{\partial L}{\partial w} = 0 = -uv + \beta(u+v); \frac{\partial L}{\partial u} = 0 = -vw + \beta(v+w); \frac{\partial L}{\partial v} = 0 = -wu + \beta(u+w);$$

$$\beta v(w-u) = 0 \text{ and } \beta w(u-v) = 0 \Rightarrow u = v = w = \sqrt{\frac{c}{6}}$$

Lagrangian Theory – Example

- The entropy of a probability distribution ***p* = (*p*$_1$...*p*$_n$)** over a finite set {1, 2,...*n*} is defined as

$$H(\mathbf{p}) = -\sum_{i=1}^{n} p_i \log_2 p_i$$

- The maximum entropy distribution can be found by minimizing ***-H(p)*** subject to the constraints

$$\sum_{i=1}^{n} p_i = 1$$
$$\forall i \quad p_i \geq 0$$

$$L(\mathbf{p}, \beta) = \sum_{i=1}^{n} p_i \log_2 p_i + \beta\left(\sum_{i=1}^{n} p_i - 1\right)$$

→The uniform distribution (*p* = *(1/n, …, 1/n)*) has the maximum entropy

Generalized Lagrangian Theory

- Given an optimization problem with domain $\Omega \subseteq \Re^n$

$$\begin{array}{llll} \text{minimize} & f(\mathbf{w}) & \mathbf{w} \in \Omega & \{\text{objective function} \\ \text{subject to} & g_i(\mathbf{w}) \le 0 & i = 1 \ldots k, & \{\text{inequality constraints} \\ & h_j(\mathbf{w}) = 0 & j = 1 \ldots m & \{\text{equality constraints} \end{array}$$

where *f* is convex, and g_i and h_j are affine, we can define the generalized Lagrangian function as

$$L(\mathbf{w}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = f(\mathbf{w}) + \sum_{i=1}^{k} \alpha_i g_i(\mathbf{w}) + \sum_{j=1}^{m} \beta_j h_j(\mathbf{w})$$

- An affine function is a linear function plus a translation:

  F(*x*) is affine if *F*(*x*)=*G*(*x*)+*b* where *G*(*x*) is a linear function of *x* and *b* is a constant

Generalized Lagrangian Theory: Karush-Kuhn-Tucker(KKT) Conditions

- **Given an optimization problem with domain $\Omega \subseteq \Re^n$**

minimize	$f(\mathbf{w})$	$\mathbf{w} \in \Omega$	$\{$ objective function
subject to	$g_i(\mathbf{w}) \le 0$	$i = 1 \ldots k,$	$\{$ inequality constraints
	$h_j(\mathbf{w}) = 0$	$j = 1 \ldots m$	$\{$ equality constraints

where *f* is convex, and g_i and h_j are affine. The necessary and sufficient conditions for *w to be an optimum are the existence of α* and β* such that**

$$\frac{\partial L(\mathbf{w}^*, \boldsymbol{\alpha}^*, \boldsymbol{\beta}^*)}{\partial \mathbf{w}} = 0, \quad \frac{\partial L(\mathbf{w}^*, \boldsymbol{\alpha}^*, \boldsymbol{\beta}^*)}{\partial \boldsymbol{\beta}} = 0,$$

$$\alpha_i^* g_i(\mathbf{w}^*) = 0; \; g_i(\mathbf{w}^*) \le 0 \; ; \alpha_i^* \ge 0; i = 1 \ldots k$$

A solution point can be one of two positions with respect to an inequality constraint: either a constraint is *active* (*gi(W*)*=0), or *inactive* (*gi(W*)*<0) and the corresponding multiplier $\alpha_i^* = 0$

Dual Optimization Problem

- A primal problem can be transformed into a *dual* by simply setting to zero, the derivatives of the Lagrangian with respect to the primal variables and substituting the results back into the Lagrangian to remove dependence on the primal variables, resulting in

$$\theta(\boldsymbol{\alpha}, \boldsymbol{\beta}) = \inf_{\mathbf{w} \in \Omega} L(\mathbf{w}, \boldsymbol{\alpha}, \boldsymbol{\beta})$$

  which contains only dual variables and needs to be maximized under simpler constraints

- The infimum of a set *S* of real numbers is denoted by *inf(S)* and is defined to be the largest real number that is smaller than or equal to every number in *S*. If no such number exists then *inf(S)* = −∞

$$\inf\{x \in \mathbb{R} \mid 0 < x < 1\} = 0$$

SOGANG UNIVERSITY

Dual Optimization Problem

- **Theorem: Let $\mathbf{w} \in \Omega$ be a feasible solution of the primal problem and *(α, β)* a feasible solution of the dual problem. Then**

$$f(\mathbf{w}) \geq \theta(\boldsymbol{\alpha}, \boldsymbol{\beta})$$

- **Corollary: The value of the dual is upper bounded by the value of the primal,**

$$\sup\{\theta(\boldsymbol{\alpha}, \boldsymbol{\beta}) : \boldsymbol{\alpha} \geq 0\} \leq \inf\{f(\mathbf{w}) : g(\mathbf{w}) \leq 0, h(\mathbf{w}) = 0\}$$

- **Theorem: If $f(\mathrm{w}^*)$ is convex, and *gi* and *hj* are affine, then the duality gap $f(\mathrm{w}^*) - \theta(\alpha^*, \beta^*) = 0$ where *W** and (α^*, β^*) are solutions of the primal and dual problem respectively**

- **Corollary: If $f(\mathbf{w}^*) = \theta(\boldsymbol{\alpha}^*, \boldsymbol{\beta}^*)$ where $\boldsymbol{\alpha}^* \geq 0, g(\mathbf{w}^*) \leq 0, h(\mathbf{w}^*) = 0$ then $\mathbf{w}^*$ and $(\boldsymbol{\alpha}^*, \boldsymbol{\beta}^*)$ solve the primal and dual problem respectively. In this case $\alpha_i^* g_i(\mathbf{w}^*) = 0 \text{ for } i = 1, \ldots, k$**

Maximum Margin Hyperplane

- The problem of finding the maximal margin hyperplane is a constrained optimization problem

- Use Lagrange theory (extended by Karush, Kuhn, and Tucker – KKT)

- Minimize $\langle \mathbf{W}, \mathbf{W} \rangle$

  subject to

$$y_i\left(\langle \mathbf{W}, \mathbf{X}_i \rangle + b\right) \geq 1$$

- Lagrangian:

$$L_p(\mathbf{w}) = \frac{1}{2}\langle \mathbf{w}, \mathbf{w} \rangle - \sum \alpha_i [y_i(\langle \mathbf{w}, x_i \rangle + b) - 1] \qquad (4)$$

$$\alpha \geq 0$$

From Primal to Dual

- Minimize ***Lp***(***w***) with respect to (***w,b***) requiring that derivatives of ***Lp***(***w***) with respect to (***w,b***) all vanish, subject to the constraints $\alpha_i \geq 0$

- Differentiating ***Lp***(***w***):

$$\frac{\partial L_p}{\partial \mathbf{w}} = 0, \quad (5)$$

$$\frac{\partial L_p}{\partial b} = 0 \quad (6)$$

$$\mathbf{w} = \sum_i \alpha_i y_i x_i \quad (7)$$

$$\sum_i \alpha_i y_i = 0 \quad (8)$$

- Substituting this equality constraints back into ***Lp***(***w***) we obtain a dual problem

The Dual Problem

- **Maximize:**

$$L_D(\mathbf{w}) = \frac{1}{2}\left\langle \left(\sum_i \alpha_i y_i \mathbf{x}_i\right)\left(\sum_j \alpha_j y_j \mathbf{x}_j\right)\right\rangle$$

$$-\sum_i \alpha_i \left[y_i \left(\left\langle \left(\sum_j \alpha_j y_j \mathbf{x}_j \right), \mathbf{x}_i \right\rangle + b \right) - 1 \right]$$

$$= \frac{1}{2}\sum_{ij} \alpha_i \alpha_j y_i y_j \langle \mathbf{x}_i, \mathbf{x}_j \rangle - \sum_{ij} \alpha_i \alpha_j y_i y_j \langle \mathbf{x}_i, \mathbf{x}_j \rangle - b\sum_i \alpha_i y_i + \sum_i \alpha_i$$

$$= -\frac{1}{2}\sum_{ij} \alpha_i \alpha_j y_i y_j \langle \mathbf{x}_i, \mathbf{x}_j \rangle + \sum_i \alpha_i$$

subject to $\sum_i \alpha_i y_i = 0$ **and** $\alpha_i \geq 0$

- **Duality permits the use of kernels!**
- **This is a quadratic optimization problem: convex, no local minima**
- **Solvable in polynomial time**

The Dual Problem

- **The value of *b* does not appear in the dual problem and so *b* needs to be found from primal constraints**

$$b = -\frac{\max_{y_i=-1}(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle) + \min_{y_i=1}(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle)}{2}$$

- ***b* can also be computed by solving the equation**

$$f(x) = \sum_i \alpha_i y_i < \varphi(x_i), \varphi(x) > +b = y$$

for any sample (*X,y*) associated with nonzero α_i

- **A numerically more stable solution can be obtained by**

$$b = \frac{1}{n_{SV}} \sum_{j \in SV} [y_j - \sum_{i \in SV} \alpha_i y_i K(x_i, x_j)]$$

Karush-Kuhn-Tucker Conditions

$$L_P(\mathbf{w}) = \frac{1}{2}\langle \mathbf{w}, \mathbf{w} \rangle - \sum_i \alpha_i \left[y_i \left(\langle \mathbf{w}, \mathbf{x}_i \rangle + b \right) - 1 \right]$$

$$\frac{\partial L_P}{\partial \mathbf{w}} = 0 \Rightarrow \mathbf{w} = \sum_i \alpha_i y_i \mathbf{x}_i$$

$$\frac{\partial L_P}{\partial b} = 0 \Rightarrow \sum_i \alpha_i y_i = 0$$

$$y_i \left(\langle \mathbf{w}, \mathbf{x}_i \rangle + b \right) - 1 \geq 0$$

$$\alpha_i \left[y_i \left(\langle \mathbf{w}, \mathbf{x}_i \rangle + b \right) - 1 \right] = 0$$

$$\alpha_i \geq 0$$

- **Solving for the SVM is equivalent to finding a solution to the KKT conditions**

Karush-Kuhn-Tucker Conditions for SVM

- **The KKT conditions state that optimal solutions $\alpha_i(\mathbf{w}, b)$ must satisfy**

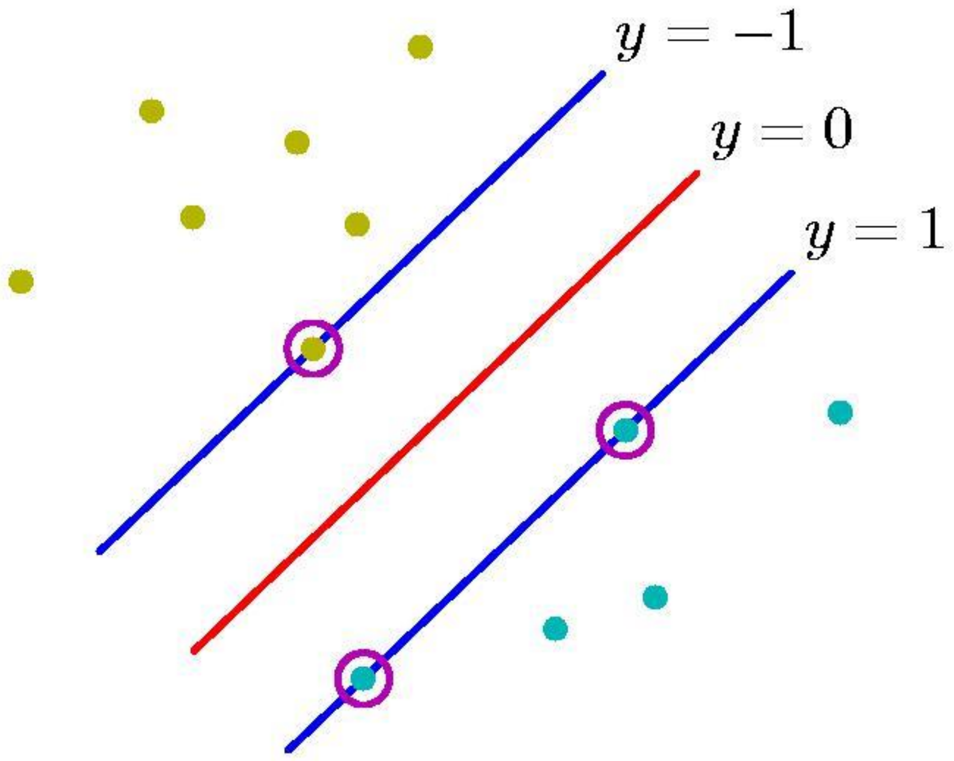
$$\alpha_i\left[y_i\left(\langle \mathbf{w}, \mathbf{x}_i \rangle + b\right) - 1\right] = 0$$

- **Only the training samples *Xi* for which the functional margin = 1 can have nonzero α_i; They are called Support Vectors**

- **The optimal hyperplane can be expressed in the dual representation in terms of this subset of training samples – the support vectors**

$$f(\mathbf{x}, \boldsymbol{\alpha}, b) = \sum_{i=1}^{l} y_i \alpha_i \langle \mathbf{x}_i \cdot \mathbf{x} \rangle + b = \sum_{i \in \mathrm{sv}} y_i \alpha_i \langle \mathbf{x}_i \cdot \mathbf{x} \rangle + b$$

Support Vectors



Karush-Kuhn-Tucker Conditions for SVM

$$\langle \mathbf{w},\mathbf{w}\rangle = \sum_{j=1}^{l}\sum_{i=1}^{l} y_i y_j \alpha_i \alpha_j \langle \mathbf{x}_i, \mathbf{x}_j\rangle$$

$$= \sum_{j\in SV}\alpha_j y_j \sum_{i\in SV}\alpha_i y_i \langle \mathbf{x}_i, \mathbf{x}_j\rangle$$

$$(\because \mathbf{x}_j \in SV, y_j\left(\sum_{i\in SV}\alpha_i y_i \langle \mathbf{x}_i, \mathbf{x}_j\rangle + b\right) = 1 \Rightarrow y_j \sum_{i\in SV}\alpha_i y_i \langle \mathbf{x}_i, \mathbf{x}_j\rangle = 1 - y_j b)$$

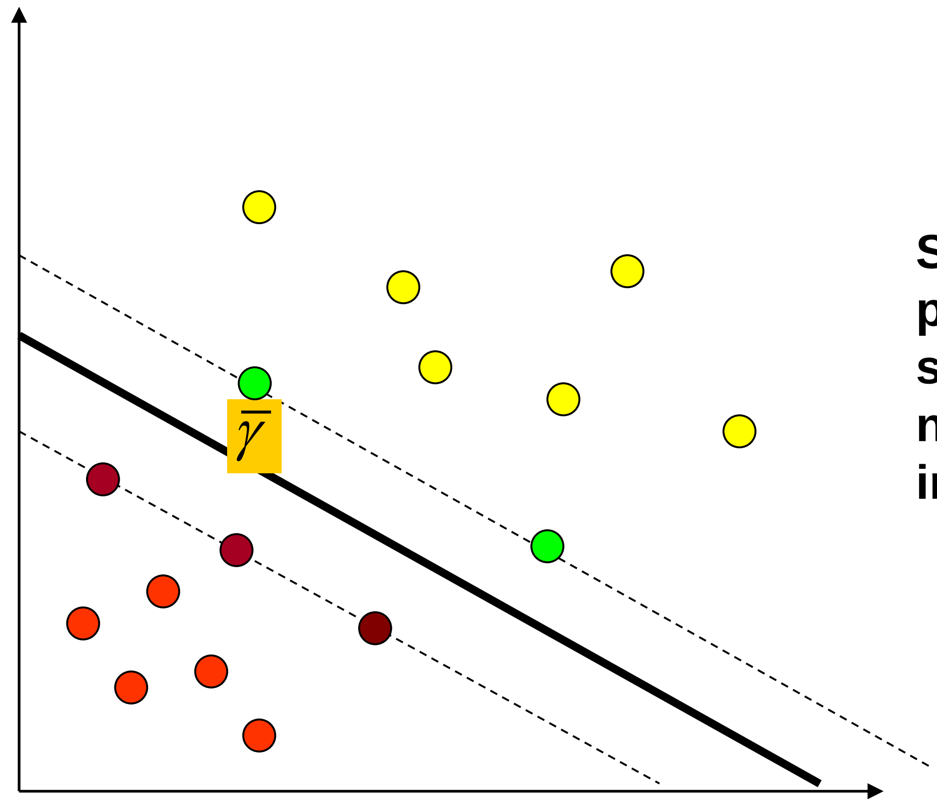
$$= \sum_{j\in SV}\alpha_j (1 - y_j b) = -b\left(\sum_{j\in SV}\alpha_j y_j + \sum_{j\notin SV} 0 y_j\right) + \sum_{j\in SV}\alpha_j$$

$$= -b\underbrace{\left(\sum_{j\in SV}\alpha_j y_j + \sum_{j\notin SV}\alpha_j y_j\right)}_{0 \text{ because of KKT conditions}} + \sum_{j\in SV}\alpha_j$$

$$= \sum_{i\in SV}\alpha_i$$

So we have $\bar{\gamma} = \dfrac{1}{\|\mathbf{w}\|} = \dfrac{1}{\sqrt{\left(\sum_{\mathbf{x}_i\in SV}\alpha_i\right)}}$

Support Vector Machines Yield Sparse Solutions



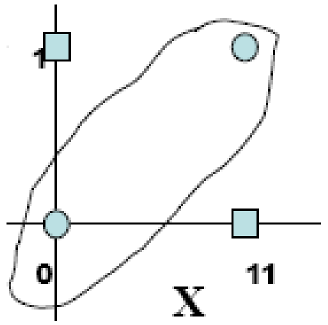
Support vectors from positive (yellow) class are shown in green and the negative class are shown in purple

Example: XOR

$$Q(\alpha)=\sum_{i=1}^{N}\alpha_i-\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j\langle \mathbf{x}_i,\mathbf{x}_j\rangle$$

$$K(\mathbf{x}_i,\mathbf{x}_j)=\left(1+\langle \mathbf{x}_i,\mathbf{x}_j\rangle\right)^2$$

$$\varphi(\mathbf{x})=\left[1,\mathbf{x}_1^2,\sqrt{2}\mathbf{x}_1\mathbf{x}_2,\mathbf{x}_2^2,\sqrt{2}\mathbf{x}_1,\sqrt{2}\mathbf{x}_2\right]^T$$



$\mathbf{X}$	y
[-1,-1]	-1
[-1,+1]	+1
[+1,-1]	+1
[+1,+1]	-1

$$K=\begin{bmatrix} 9 & 1 & 1 & 1 \\ 1 & 9 & 1 & 1 \\ 1 & 1 & 9 & 1 \\ 1 & 1 & 1 & 9 \end{bmatrix}$$

Example: XOR

$$Q(\alpha) = \alpha_1 + \alpha_2 + \alpha_3 + \alpha_4 - \frac{1}{2}\left(9\alpha_1^2 - 2\alpha_1\alpha_2 - 2\alpha_1\alpha_3 + 2\alpha_1\alpha_4 + 9\alpha_2^2 + 2\alpha_2\alpha_3 - 2\alpha_2\alpha_4 + 9\alpha_3^2 - 2\alpha_3\alpha_4 + 9\alpha_4^2\right)$$

- **Setting derivative of *Q* w.r.t. the dual variables to *0*:**

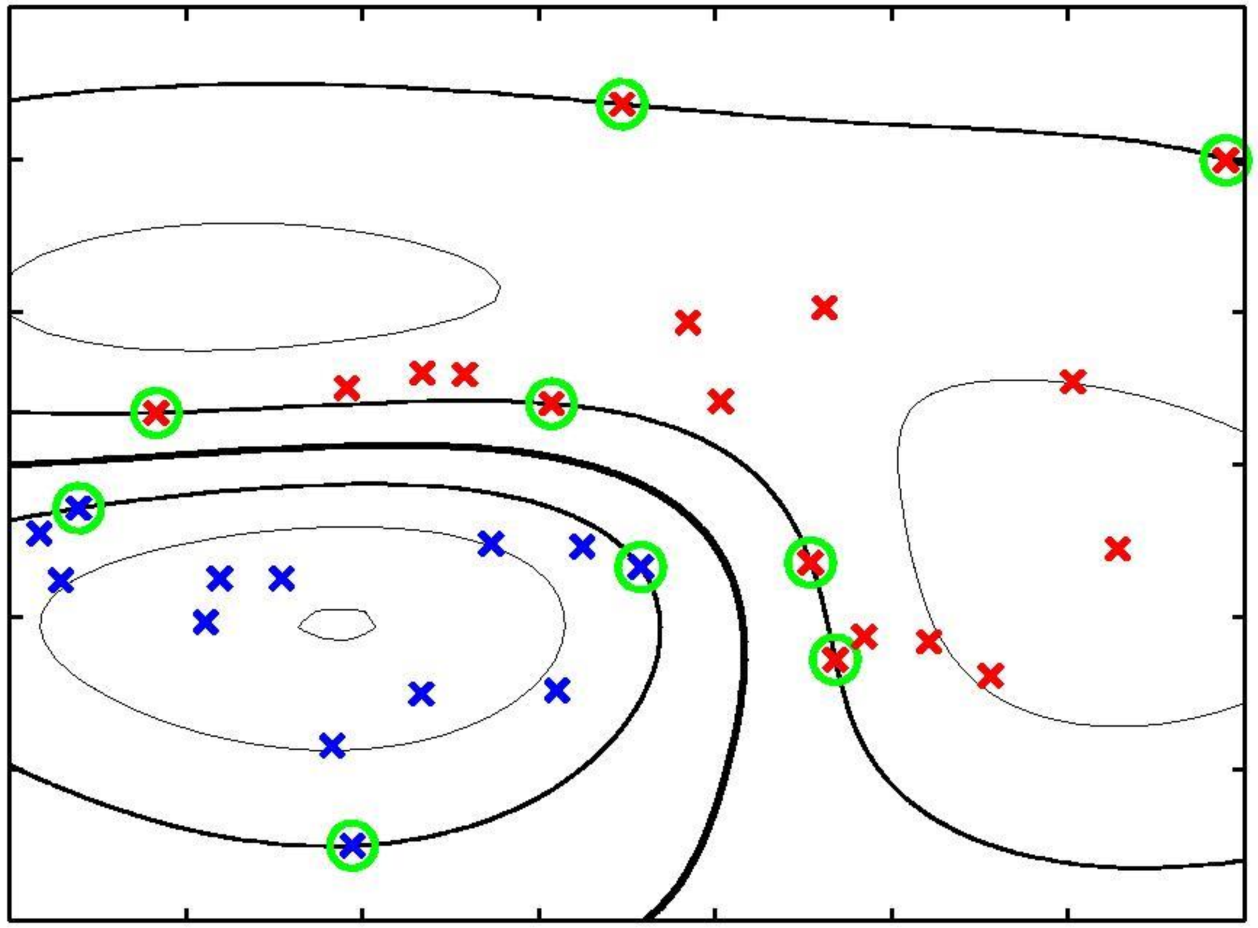
$$\begin{aligned}
&9\alpha_1 - \alpha_2 - \alpha_3 + \alpha_4 = 1\\
&-\alpha_1 + 9\alpha_2 + \alpha_3 - \alpha_4 = 1\\
&-\alpha_1 + \alpha_2 + 9\alpha_3 - \alpha_4 = 1\\
&\alpha_1 - \alpha_2 - \alpha_3 + 9\alpha_4 = 1\\
&\alpha_{0,1} = \alpha_{0,2} = \alpha_{0,3} = \alpha_{0,4} = \frac{1}{8},\ Q_0(\alpha) = \frac{1}{4}
\end{aligned}$$

- **Not surprisingly, each of the training samples is a support vector**

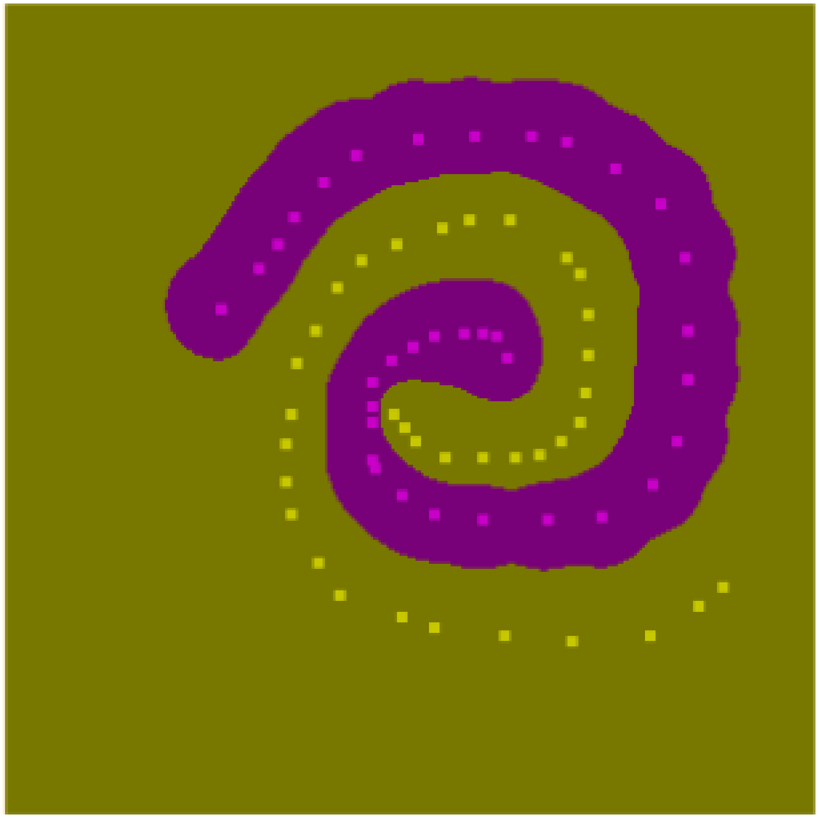
$$\mathbf{w}_0 = \sum_{i=1}^{N} \alpha_i y_i \varphi(\mathbf{x}_i) = \begin{bmatrix} 0 & 0 & -\frac{\sqrt{2}}{2} & 0 & 0 & 0 \end{bmatrix}$$

SVM Example

Synthetic data classified by a SVM with Gaussian kernel



Example: Two Spirals



Summary of Maximal Margin Classifier

- **Good generalization when the training data is noise free and separable in the kernel-induced feature space**

- **Excessively large Lagrange multipliers often signal outliers**
  - **data points that are most difficult to classify**
  - **SVM can be used for identifying outliers**

- **Focuses on boundary points – if they happen to be mislabeled (noisy training data)**
  - **The result is a terrible classifier if the training set is separable**
  - **Noisy training data often makes the training set non separable**
  - **Use of highly nonlinear kernels to make the training set separable increases the likelihood of overfitting – despite maximizing the margin**

Non-Separable Case

- In the case of non-separable data in feature space, the objective function grows arbitrarily large

- Solution: Relax the constraint that all training data be correctly classified but only when necessary to do so

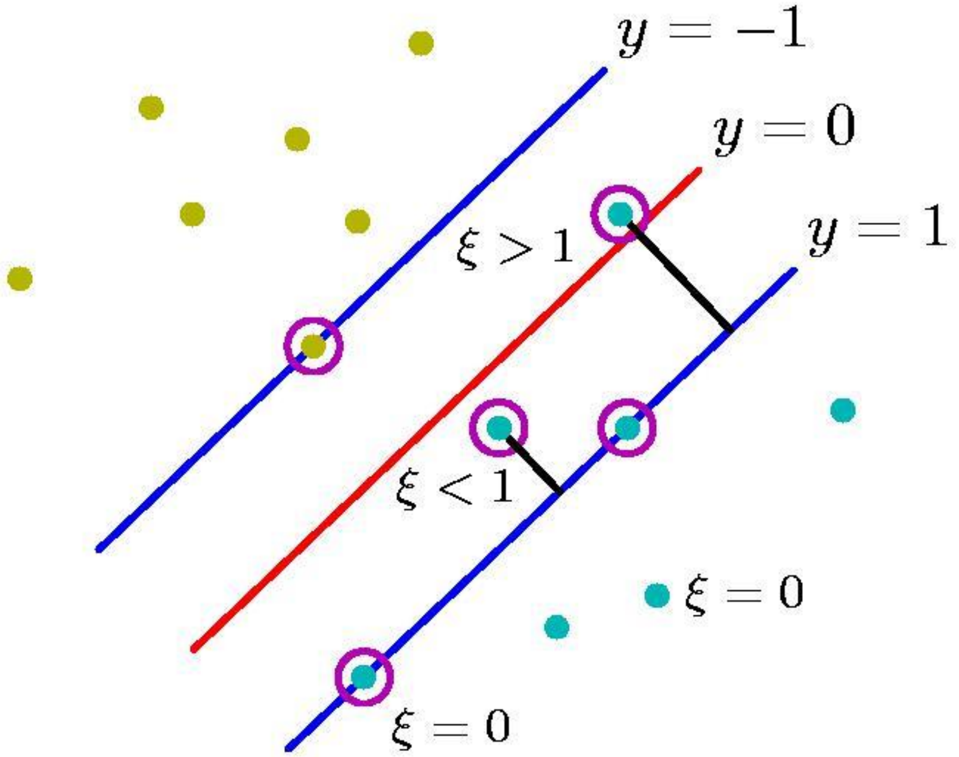
- Cortes and Vapnik, 1995 introduced slack variables in the constraints:

$$\xi_i, \quad i = 1, .., l$$

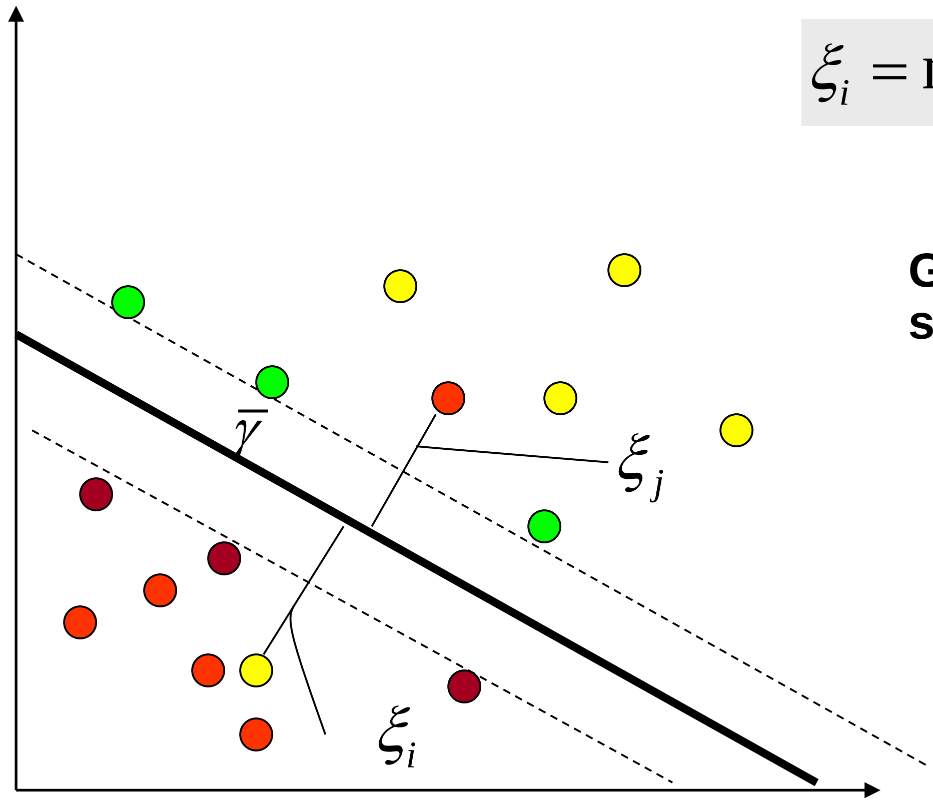
$$\xi_i = \max(0, \gamma - y_i(\langle \mathbf{w}, x_i \rangle + b)) \qquad (17)$$

Slack Variables

$$\xi_i = \max(0, \gamma - y_i(\langle \mathbf{w}, x_i \rangle + b)).$$



Non-Separable Case



$$\xi_i = \max(0, \gamma - y_i(\langle \mathbf{w}, x_i \rangle + b)).$$

Generalization error can be shown to be

$$\varepsilon \le \frac{1}{l} \frac{\left(L + \sqrt{\sum \xi_i^2}\right)^2}{\bar{\gamma}^2}$$

Non-Separable Case

- ξ_i measures how much the point fails to have a margin of 1 from the hyperplane

- For error to occur, the corresponding ξ_i must exceed γ (which was chosen to be unity)

- So $\sum_i \xi_i$ is an upper bound on the number of training errors

- So the objective function is changed from $\|\mathbf{w}\|^2/2$ to $\|\mathbf{w}\|^2/2 + C\sum_i \xi_i^k$ where *C* and *k* are parameters

  (typically *k=1* or *2,* and *C* is a large positive constant that controls the trade-off between the penalty and margin)

SOGANG UNIVERSITY

The Soft-Margin Classifier

- **Minimize:** $\frac{1}{2}\langle \mathbf{w}, \mathbf{w} \rangle + C\sum_i \xi_i$

Subject to $y_i\left(\langle \mathbf{w}, \mathbf{x}_i \rangle + b\right) \geq \left(1 - \xi_i\right)$

$\xi_i \geq 0$

$$L_P = \underbrace{\frac{1}{2}\| \mathbf{w} \|^2 + C\sum_i \xi_i}_{\text{objective function}} - \underbrace{\sum_i \alpha_i\left(y_i\left(\langle \mathbf{w}, \mathbf{x}_i \rangle + b\right) - 1 + \xi_i\right)}_{\text{inequality constraints involving } \xi_i} - \underbrace{\sum_i \mu_i \xi_i}_{\substack{\text{non-negativity} \\ \text{constraints on } \xi_i}}$$

The Soft-Margin Classifier: KKT Conditions

$$\xi_i \geq 0, \alpha_i \geq 0, \mu_i \geq 0$$
$$\alpha_i \left(y_i \left(\langle \mathbf{w}, \mathbf{x}_i \rangle + b \right) - 1 + \xi_i \right) = 0$$
$$\mu_i \xi_i = 0$$

$\alpha_i > 0$ only if

- ***Xi*** **is a true support vector i.e.** $\langle \mathbf{w}, \mathbf{x}_i \rangle + b = \pm 1$ and hence $\xi_i = 0$

OR

- ***Xi*** **is misclassified by** $(\mathbf{w}, b)$ and hence $\xi_i > 0$

From Primal to Dual

$$L_P = \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_i \xi_i - \sum_i \alpha_i\left(y_i\left(\langle \mathbf{w}, \mathbf{x}_i\rangle + b\right) - 1 + \xi_i\right) - \sum_i \mu_i \xi_i$$

Equating the derivatives of *LP* w.r.t. primal variables to *0*,

$$\frac{\partial L_P}{\partial \mathbf{w}} = 0 \Rightarrow \mathbf{w} = \sum_i \alpha_i y_i \mathbf{x}_i$$

$$\frac{\partial L_P}{\partial b} = 0 \Rightarrow \sum_i \alpha_i y_i = 0$$

$$\frac{\partial L_P}{\partial \xi_i} = 0 \Rightarrow \alpha_i + \mu_i = C \Rightarrow 0 \le \alpha_i \le C$$

Substituting back into *LP*, we get the dual *LD* to be *maximized*

$$L_D = \sum_i \alpha_i - \frac{1}{2}\sum_{ij} \alpha_i \alpha_j y_i y_j \langle \mathbf{x}_i, \mathbf{x}_j \rangle$$

Soft-Margin Dual Lagrangian

- **Maximize:** $$L_D = \sum_i \alpha_i - \frac{1}{2}\sum_{ij} \alpha_i \alpha_j y_i y_j \left\langle \mathbf{x}_i, \mathbf{x}_j \right\rangle$$

Subject to $0 \le \alpha_i \le C$ and $\sum_i \alpha_i y_i = 0$

KKT Conditions:

$$\alpha_i \left(y_i \left(\left\langle \mathbf{w}, \mathbf{x}_i \right\rangle + b \right) - 1 + \xi_i \right) = 0$$

$$\xi_i (\alpha_i - C) = 0$$

- **Slack variables are non-zero (inside the margin; misclassified training sample (ξ_i>1) + correctly classified sample (0<ξ_i<=1)) only when $\alpha_i = C$**
- **For true support vectors, $0 < \alpha_i < C$ (on the margin, $\xi_i = 0$)**
- **For all other (correctly classified) training samples, $\alpha_i = 0$**

Implementation Techniques

- **Maximizing a quadratic function, subject to a linear equality and inequality constraints**

$$W(\alpha) = \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j K(x_i, x_j)$$

$$0 \le \alpha_i \le C$$

$$\sum_i \alpha_i y_i = 0$$

$$\frac{\partial W(\boldsymbol{\alpha})}{\partial \alpha_i} = 1 - y_i \sum_j \alpha_j y_j K(x_i, x_j)$$

On-line Algorithm for the 1-Norm Soft Margin (bias *b = 0*)

```
Given training set D
$\boldsymbol{\alpha} \leftarrow \mathbf{0}$    $\eta_i = \frac{1}{K(\mathbf{x}_i, \mathbf{x}_i)}$
repeat
    for i = 1 to l
        $\alpha_i \leftarrow \alpha_i + \eta_i (1 - y_i \sum_j \alpha_j y_j K(x_i, x_j))$
        if $\alpha_i < 0$ then $\alpha_i \leftarrow 0$
        else if $\alpha_i > C$ then $\alpha_i \leftarrow C$
    endfor
until stopping criterion satisfied
return α
```

Note: In the general setting, *b≠0* and we need to solve for *b*

Implementation Techniques

- Use QP packages (MINOS, LOQO, quadprog from MATLAB optimization toolbox). They are not online and require that the data are held in memory in the form of kernel matrix

  → space complexity quadratic in the sample size

- Stochastic Gradient Ascent: Sequentially update 1 weight at the time. So it is online. Gives excellent approximation in most cases.

$$\hat{\alpha}_i \leftarrow \alpha_i + \frac{1}{K(x_i, x_i)}\left(1 - y_i \sum_j \alpha_j y_j K(x_i, x_j)\right)$$

Chunking and Decomposition

```
Given training set D
$\boldsymbol{\alpha} \leftarrow \mathbf{0}$
select an arbitrary working set $\hat{D} \subset D$
repeat
    solve optimization problem on $\hat{D}$
    select new working set from data not satisfying KKT conditions
until stopping criterion satisfied
return $\boldsymbol{\alpha}$
```

Sequential Minimal Optimization (SMO)

- At each step SMO: optimize two weights α_i, α_j simultaneously in order not to violate the linear constraint

$$\sum_i \alpha_i y_i = 0$$

- Optimization of the two weights is performed analytically

- Realizes gradient descent without leaving the linear constraint (J. Platt)

- Online versions exist (Li, Long; Gentile)

SVM Implementations

- SVMLight – one of the first practical implementations of SVM (T. Joachims)
- Matlab toolboxes for SVM
- LIBSVM – one of the best SVM implementations by Chih-Chung Chang and Chih-Jen Lin of the National University of Singapore http://www.csie.ntu.edu.tw/~cjlin/libsvm/
- WLSVM – LIBSVM integrated with WEKA machine learning toolbox Yasser El-Manzalawy of the ISU AI Lab http://www.cs.iastate.edu/~yasser/wlsvm/

Why does SVM work?

- **The feature space is often very high dimensional. Why don't we have the curse of dimensionality?**
- **A classifier in a high-dimensional space has many parameters and is hard to estimate**
- **Vapnik argues that the fundamental problem is not the number of parameters to be estimated. Rather, the problem is the *capacity* of a classifier.**
- **Typically, a classifier with many parameters is very flexible, but there are also exceptions**
  - **Let $x_i=10^i$ where *i* ranges from *1* to *n*. The classifier $y = \text{sign}\big(\sin(\alpha x)\big)$ can classify all x_i correctly for all possible combination of class labels on x_i**
  - **This 1-parameter classifier is very flexible**

Why does SVM work?

- Vapnik argues that the capacity of a classifier should not be characterized by the number of parameters, but by the capacity of a classifier
  - This is formalized by the "VC-dimension" of a classifier

- The minimization of $||w||^2$ subject to the condition that the *margin =1* has the effect of restricting the VC-dimension of the classifier in the feature space

- The SVM performs structural risk minimization: the empirical risk (training error), plus a term related to the generalization ability of the classifier, is minimized

- The term $\frac{1}{2}||w||^2$ "shrinks" the parameters towards zero to avoid overfitting

Choosing the Kernel Function

- Probably the most tricky part of using SVM
- The kernel function should maximize the similarity among instances within a class while accentuating the differences between classes
- A variety of kernels have been proposed (diffusion kernel, Fisher kernel, string kernel, …) for different types of data
- In practice, a low degree polynomial kernel or RBF kernel with a reasonable width is a good initial try for data that live in a fixed dimensional input space
- Low order Markov kernels and its relatives are good ones to consider for structured data – strings, images, etc.

Recap: How to Use SVM

- **Prepare the data set**
- **Select the kernel function to use**
- **Select the parameter of the kernel function and the value of *C***
  - **You can use the values suggested by the SVM software**
  - **You can use a tuning set to tune *C***
- **Execute the training algorithm and obtain the α_i and *b***
- **Unseen data can be classified using the α_i, the support vectors, and *b***

$$f(\mathbf{x},\boldsymbol{\alpha},b)=\sum_{i=1}^{l} y_i\alpha_i\left\langle \mathbf{x}_i\cdot\mathbf{x}\right\rangle+b=\sum_{i\in \mathrm{sv}} y_i\alpha_i\left\langle \mathbf{x}_i\cdot\mathbf{x}\right\rangle+b$$

Strengths and Weaknesses of SVM

- **Strengths**
  - **Training is relatively easy**
    - **No local optima**
  - **It scales relatively well to high dimensional data**
  - **Tradeoff between classifier complexity and error can be controlled explicitly**
  - **Non-traditional data like strings and trees can be used as input to SVM, instead of feature vectors**

- **Weaknesses**
  - **Need to choose a “good” kernel function**

Other aspects of SVM

- How to use SVM for multi-class classification?
  - One can change the QP formulation to become multi-class
  - More often, multiple binary classifiers are combined
  - One can train multiple one-versus-all classifiers, or combine multiple pairwise classifiers “intelligently”

- SVMs for regression

- Relevance vector machine (RVM): Bayesian formulation, output posterior probabilities

Representative Applications of SVM

- Handwritten letter classification
- Text classification
- Image classification (e.g. face recognition)
- Bioinformatics
  - Classification of tissues – healthy versus cancerous – based on gene expression data
  - Protein function/structure classification
  - Protein sub-cellular localization
- …

Learning Linear Functions

- One approach to regression: directly construct an appropriate function $y(\vec{x})$
- *Linear regression* (linear neuron)

$$y(\vec{x}, \vec{w}) = w_0 + \sum_{i=1}^{d} w_i x_i = \vec{w}^t \vec{x}$$

- We want to find a weight vector $\vec{w}$ such that the linear function is a *good* approximation of unknown $f(\vec{x})$ based on training examples.

Learning Linear Functions

- Define a criterion/error function $J(\vec{w})$, and the learning problem reduces to looking for a weight vector that minimizes $J(\vec{w})$.
- Let the output for input $\vec{x}_k = (x_0^k, x_1^k, \ldots, x_d^k)^t$ be

$$o_k = y(\vec{x}_k, \vec{w}) = \sum_{i=0}^{d} w_i x_i^k = \vec{w}^t \vec{x}_k$$

- The error for the labeled input $(\vec{x}_k, t_k)$: $e_k = o_k - t_k$
- We wish to minimize the magnitude of this error irrespective of the sign. We will use the squared error, and we are interested in the error over all training [redacted] samples

Learning Linear Functions

- The sum-of-squared-error (*Mean Squared Error*) function

$$J_s(\vec{w}) = \frac{1}{2}\sum_{k=1}^{n}(o_k - t_k)^2 = \frac{1}{2}\sum_{k=1}^{n}(\vec{w}^t\vec{x}_k - t_k)^2$$

  where n: number of samples

- The learning problem reduces to looking for a weight vector that minimizes $J_s(\vec{w})$
  — a minimum-squared-error (MSE) solution.

Learning Linear Functions:
Computing Gradient

$$\begin{aligned}
\frac{\partial J}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_k (o_k - t_k)^2 = \frac{1}{2}\sum_k \frac{\partial}{\partial w_i}(o_k - t_k)^2 \\
&= \frac{1}{2}\sum_k 2(o_k - t_k)\frac{\partial}{\partial w_i}(o_k - t_k) \\
&= \sum_k (o_k - t_k)\frac{\partial o_k}{\partial w_i}
\end{aligned}$$

$$\frac{\partial o_k}{\partial w_i} = x_i^k$$

Learning Linear Functions:
Computing Gradient

$$\frac{\partial J}{\partial w_i} = \sum_{k=1}^{n} (o_k - t_k) x_i^k$$

$$\nabla J[\vec{w}] = \sum_{k=1}^{n} (o_k - t_k) \vec{x}_k$$

Necessary condition for minimization

$$\frac{\partial J}{\partial w_i} = 0, \qquad i = 0, \ldots, d$$

Equivalently $\nabla J[\vec{w}] = 0$

- This defines a set of linear equations in w_i's. Can be solved explicitly using linear algebra technique [redacted]

SOGANG UNIVERSITY

Learning Linear Functions:
Linear Algebra Solution

- The weight values that minimize the sum-of-squared-error function can be found explicitly by solving the set of linear equations.

$$\sum_{k=1}^{n}\sum_{j=0}^{d} w_j x_j^k x_i^k = \sum_{k=1}^{n} x_i^k t_k$$

$$\mathbf{Y} = \begin{pmatrix} x_0^1 & \cdots & x_d^1 \\ \vdots & \ddots & \vdots \\ x_0^n & \cdots & x_d^n \end{pmatrix} \quad \mathbf{w} = \begin{pmatrix} w_0 \\ \vdots \\ w_d \end{pmatrix} \quad \mathbf{t} = \begin{pmatrix} t_1 \\ \vdots \\ t_n \end{pmatrix}$$

$$(\mathbf{Y}^t\mathbf{Y})\mathbf{w} = \mathbf{Y}^t\mathbf{t}$$

- *Pseudo-inverse* of Y: $\mathbf{Y}^\dagger = (\mathbf{Y}^t\mathbf{Y})^{-1}\mathbf{Y}^t$
- $\vec{w} = \mathbf{Y}^\dagger\mathbf{t}$
- In practice, this solution can lead to numerical difficulties due to the possibility of $\mathbf{Y}^t\mathbf{Y}$ being singular or nearly singular.
- Use the techniques of *singular value decomposition* (SVD)

SOGANG UNIVERSITY

Learning Linear Functions:
Delta/Adaline/Widrow-Hoff/LMS(Least-Mean-Squared) Rule

$$w_i \leftarrow w_i - \eta \frac{\partial E_S}{\partial w_i}$$

$$\frac{\partial E_S}{\partial w_i} = \frac{1}{2}\frac{\partial}{\partial w_i}\left\{\sum_p e_p^2\right\} = \frac{1}{2}\left(\sum_p \frac{\partial}{\partial w_i}\left(e_p^2\right)\right)$$

$$= \frac{1}{2}\left(\sum_p \left(2e_p\right)\frac{\partial e_p}{\partial w_i}\right) = \sum_p e_p\left(\frac{\partial e_p}{\partial y_p}\right)\left(\frac{\partial y_p}{\partial w_i}\right) = \sum_p e_p(-1)\left(\frac{\partial}{\partial w_i}\left(\sum_{j=0}^{n} w_j x_{jp}\right)\right)$$

$$= -\sum_p \left(d_p - y_p\right)\left(\frac{\partial}{\partial w_i}\left(w_i x_{ip} + \sum_{j \neq i} w_j x_{jp}\right)\right)$$

$$= -\sum_p \left(d_p - y_p\right)\left(\frac{\partial}{\partial w_i}\left(w_i x_{ip}\right) + \frac{\partial}{\partial w_i}\left(\sum_{j \neq i} w_j x_{jp}\right)\right)$$

$$= -\sum_p \left(d_p - y_p\right)x_{ip}$$

$$w_i \leftarrow w_i + \eta \sum_p \left(d_p - y_p\right)x_{ip}$$

Learning Real-Valued Functions

- Universal function approximation theorem
- Learning nonlinear functions using gradient descent in weight space
- Practical considerations and examples

Kolmogorov's theorem (Kolmogorov, 1957)

- **In search of more expressive hypothesis spaces**
- **Any continuous function from input to output can be expressed in the form**

$$g(x_1, .. x_n) = \sum_{j=1}^{2n+1} g_j\left(\Sigma u_{ij}(x_i)\right) \quad \forall (x_1, ... x_n) \in I^n (I = [0,1]; n \geq 2)$$

by choosing proper nonlinearities *gj* and *uij*

Universal function approximation theorem (Cybenko, 1989)

- Let $\varphi : \Re \rightarrow \Re$ be a non-constant, bounded (hence non-linear), monotone, continuous function. Let I_N be the N-dimensional unit hypercube in $\Re^N$.

- Let $C(I_N) = \{f: I_N \rightarrow \Re\}$ be the set of all continuous functions with domain I_N and range $\Re$. Then for any function $f \in C(I_N)$ and any $\varepsilon > 0$, $\exists$ an integer L and a set of real values $\theta, \alpha_j, \theta_j, w_{ji}$ $(1 \le j \le L;\ 1 \le i \le N)$ such that

$$F(x_1, x_2 \ldots x_n) = \sum_{j=1}^{L} \alpha_j \varphi\left(\sum_{i=1}^{N} w_{ji} x_i - \theta_j \right) - \theta$$

is a uniform approximation of *f* – that is,

$$\forall (x_1, \ldots x_N) \in I_N, \quad \left| F(x_1, \ldots x_N) - f(x_1, \ldots x_N) \right| < \varepsilon$$

Universal function approximation theorem (UFAT)

$$F(x_1, x_2 \ldots x_n) = \sum_{j=1}^{L} \alpha_j \varphi\left(\sum_{i=1}^{N} w_{ji} x_i - \theta_j \right) - \theta$$

- Unlike Kolmogorov's theorem, UFAT requires only one kind of nonlinearity to approximate any arbitrary nonlinear function to any desired accuracy

- The sigmoid function satisfies the UFAT requirements

$$\varphi(z) = \frac{1}{1 + e^{-az}}; a > 0 \qquad \lim_{z \to -\infty} \varphi(z) = 0; \ \lim_{z \to +\infty} \varphi(z) = 1$$

- Similar universal approximation properties can be guaranteed for other functions (e.g. radial basis functions)

Universal function approximation theorem

- UFAT guarantees the existence of arbitrarily accurate approximations of continuous functions defined over bounded subsets of $\Re^N$
- UFAT tells us the representational power of a certain class of multi-layer networks relative to the set of continuous functions defined on bounded subsets of $\Re^N$
- UFAT is not constructive – it does not tell us how to choose the parameters to construct a desired function
- To learn an unknown function from data, we need an algorithm to search the hypothesis space of multilayer networks
- Generalized delta rule allows the form of the nonlinearity to be learned from the training data

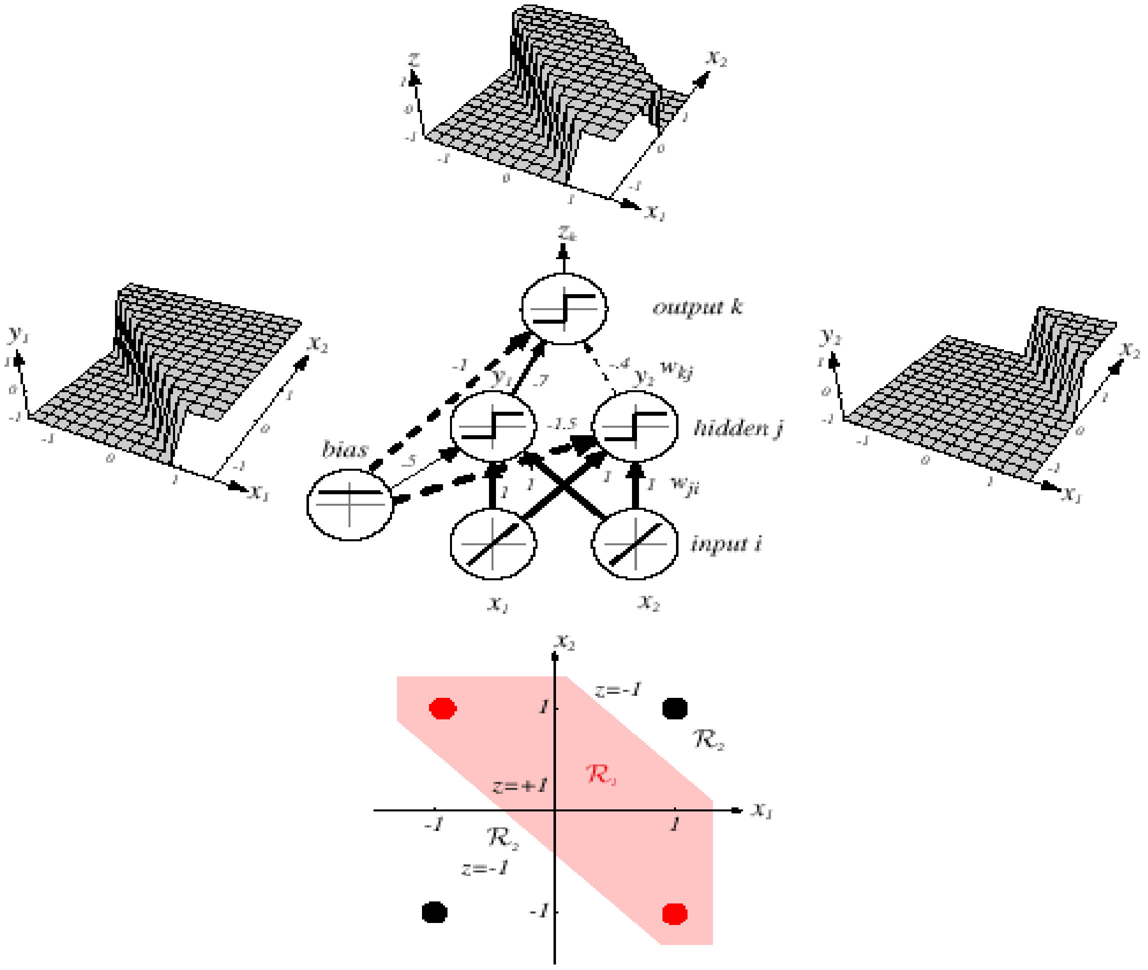
Feed-forward neural networks

- A feed-forward 3-layer network consists of 3 layers of nodes
  - *Input* nodes
  - *Hidden* nodes
  - *Output* nodes

- Interconnected by *modifiable weights* from input nodes to the hidden nodes and the hidden nodes to the output nodes

- More general topologies (with more than 3 layers of nodes, or connections that skip layers – e.g., direct connections between input and output nodes) are also possible

A three layer network that approximates the exclusive OR function



Three-layer feed-forward neural network

- A single *bias* unit is connected to each unit other than the input units

- Net *input*

$$n_j = \sum_{i=1}^{N} x_i w_{ji} + w_{j0} = \sum_{i=0}^{N} x_i w_{ji} \equiv \mathbf{W}_j . \bullet \mathbf{X},$$

where the subscript i indexes units in the input layer, j in the hidden; w_{ji} denotes the input-to-hidden layer weights at the hidden unit j.

- The output of a hidden unit is a nonlinear function of its net input. That is, $y_j = f(n_j)$ e.g.,

$$y_j = \frac{1}{1 + e^{-n_j}}$$

Three-layer feed-forward neural network

- **Each output unit similarly computes its net activation based on the hidden unit signals as:**

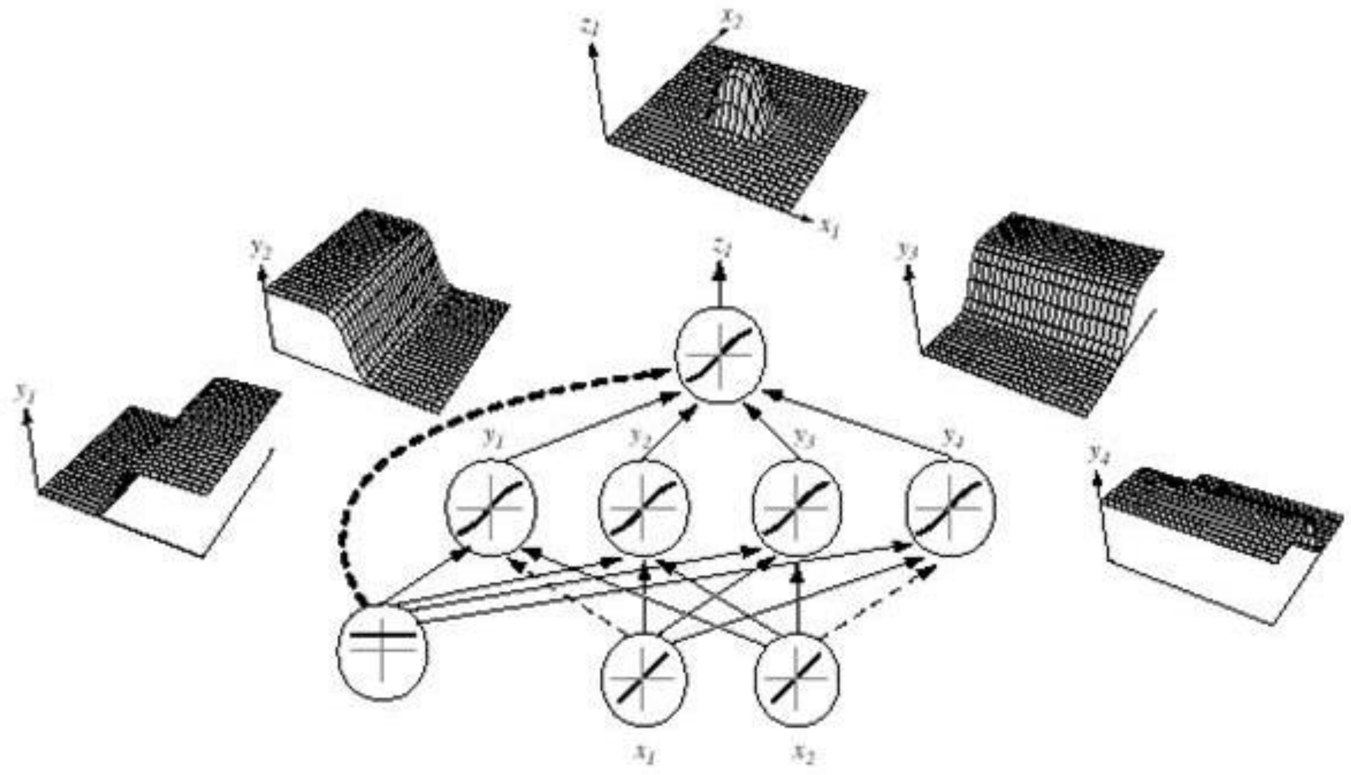
$$n_k = \sum_{j=1}^{n_H} y_j w_{kj} + w_{k0} = \sum_{j=0}^{n_H} y_j w_{kj} = \mathbf{W}_k \bullet \mathbf{Y},$$

where the subscript *k* indexes units in the ouput layer and n_H denotes the number of hidden units

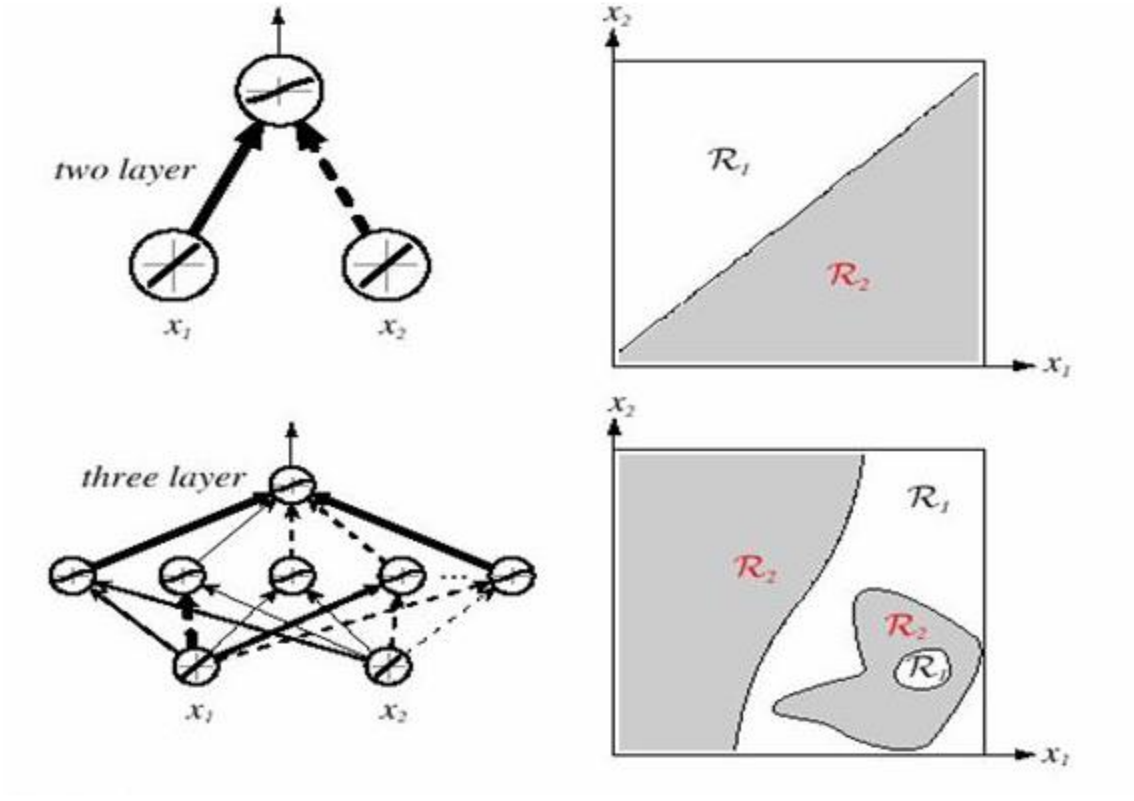
- **The output can be a linear or nonlinear function of the net input e.g.,**

$$y_k = n_k$$

Computing nonlinear functions using a feed-forward neural network



Realizing non linearly separable class boundaries using a 3-layer feed-forward neural network



Learning nonlinear functions

Given a training set determine:

- Network structure – number of hidden nodes or more generally, network topology
  - Start small and grow the network
  - Start with a sufficiently large network and prune away the unnecessary connections

- For a given structure, determine the parameters (weights) that minimize the error on the training samples (e.g., the mean squared error)

- For now, we focus on the *latter*

Generalized delta rule – error back-propagation

- Challenge – we know the desired outputs for nodes in the output layer, but not the hidden layer
- Need to solve the credit assignment problem – dividing the credit or blame for the performance of the output nodes among hidden nodes
- Generalized delta rule offers an elegant solution to the credit assignment problem in feed-forward neural networks in which each neuron computes a differentiable function of its inputs
- Solution can be generalized to other kinds of networks, including networks with cycles

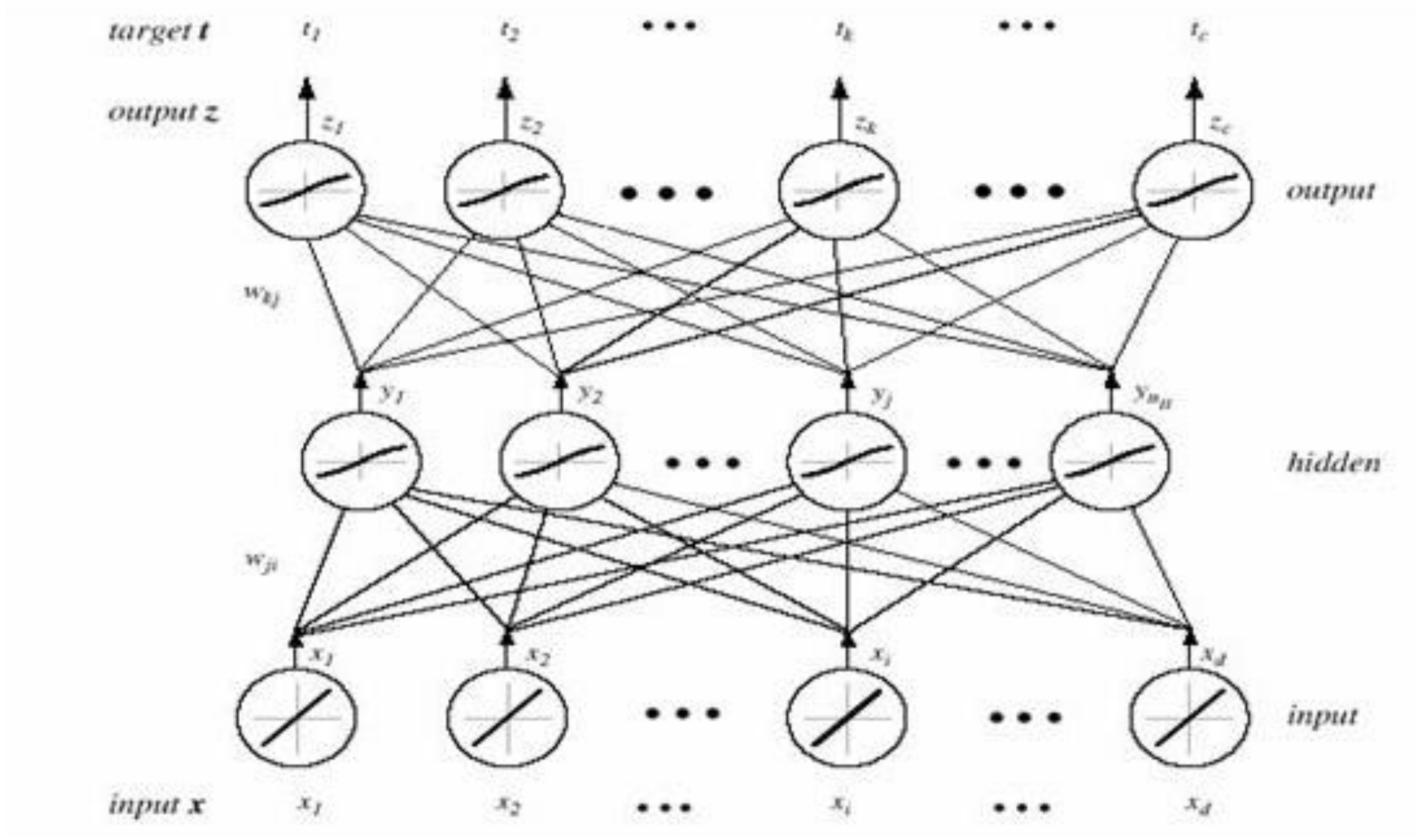
Feed-forward networks

- **Forward operation (computing output for a given input based on the current weights)**

- **Learning – modification of the network parameters (weights) to minimize an appropriate error measure**

- **Because each neuron computes a differentiable function of its inputs if error is a differentiable function of the network outputs, the error is a differentiable function of the weights in the network – so we can perform gradient descent!**

A fully connected 3-layer network



Generalized delta rule

- Let t_{kp} be the k-th target (or desired) output for input pattern $\mathbf{x}_p$ and z_{kp} be the output produced by k-th output node and let $\mathbf{W}$ represent all the weights in the network

- Training error:

$$E_S(\mathbf{W}) = \frac{1}{2}\sum_p \sum_{k=1}^{M}(t_{kp} - z_{kp})^2 = \sum_p E_p(\mathbf{W})$$

- The weights are initialized with pseudo-random values and are changed in a direction that will reduce the error:

$$\Delta w_{ji} = -\eta \frac{\partial E_S}{\partial w_{ji}} \qquad \Delta w_{kj} = -\eta \frac{\partial E_S}{\partial w_{kj}}$$

Generalized delta rule

$\eta > 0$ is a suitable the learning rate $\mathbf{W} \leftarrow \mathbf{W} + \Delta\mathbf{W}$

Hidden–to-output weights

$$\frac{\partial E_p}{\partial w_{kj}} = \frac{\partial E_p}{\partial n_{kp}} \cdot \frac{\partial n_{kp}}{\partial w_{kj}}$$

$$\frac{\partial n_{kp}}{\partial w_{kj}} = y_{jp}$$

$$\frac{\partial E_p}{\partial n_{kp}} = \frac{\partial E_p}{\partial z_{kp}} \cdot \frac{\partial z_{kp}}{\partial n_{kp}} = -(t_{kp} - z_{kp})(1)$$

$$w_{kj} \leftarrow w_{kj} - \eta \frac{\partial E_p}{\partial w_{kj}} = w_{kj} + (t_{kp} - z_{kp}) y_{jp} = w_{kj} + \delta_{kp} y_{jp}$$

Generalized delta rule

Weights from input to hidden units

$$\frac{\partial E_p}{\partial w_{ji}} = \sum_{k=1}^{M} \frac{\partial E_p}{\partial z_{kp}} \frac{\partial z_{kp}}{\partial w_{ji}} = \sum_{k=1}^{M} \frac{\partial E_p}{\partial z_{kp}} \frac{\partial z_{kp}}{\partial y_{jp}} \cdot \frac{\partial y_{jp}}{\partial n_{jp}} \cdot \frac{\partial n_{jp}}{\partial w_{ji}}$$

$$= \sum_{k=1}^{M} \frac{\partial}{\partial z_{kp}} \left[\frac{1}{2} \sum_{l=1}^{M} (t_{lp} - z_{lp})^2 \right] (w_{kj})(y_{jp})(1 - y_{jp})(x_{ip})$$

$$= -\sum_{k=1}^{M} (t_{kp} - z_{kp})(w_{kj})(y_{jp})(1 - y_{jp})(x_{ip})$$

$$= -\underbrace{\left(\sum_{k=1}^{M} \delta_{kp} (w_{kj})(y_{jp})(1 - y_{jp}) \right)}_{\delta_{jp}} (x_{ip}) = -\delta_{jp} x_{ip}$$

$$\boxed{w_{ji} \leftarrow w_{ji} + \eta \delta_{jp} x_{ip}}$$

Back propagation algorithm

```
Start with small random initial weights
Until desired stopping criterion is satisfied do
    Select a training sample from S
    Compute the outputs of all nodes based on current
        weights and the input sample
    Compute the weight updates for output nodes
    Compute the weight updates for hidden nodes
    Update the weights
```

Using neural networks for classification

Network outputs are real valued.

How can we use the networks for classification?

$$F(\mathbf{X}_p) = \underset{k}{\operatorname{argmax}}\, z_{kp}$$

Classify a pattern by assigning it to the class that corresponds to the index of the output node with the largest output for the pattern

Some Useful Tricks

- Initializing weights to small random values that place the neurons in the <u>linear portion</u> of their operating range for most of the patterns in the training set improves speed of convergence e.g.,

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

For input to hidden layer weights with the sign of the weight chosen at random

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

For hidden to output layer weights with the sign of the weight chosen at random

Some Useful Tricks

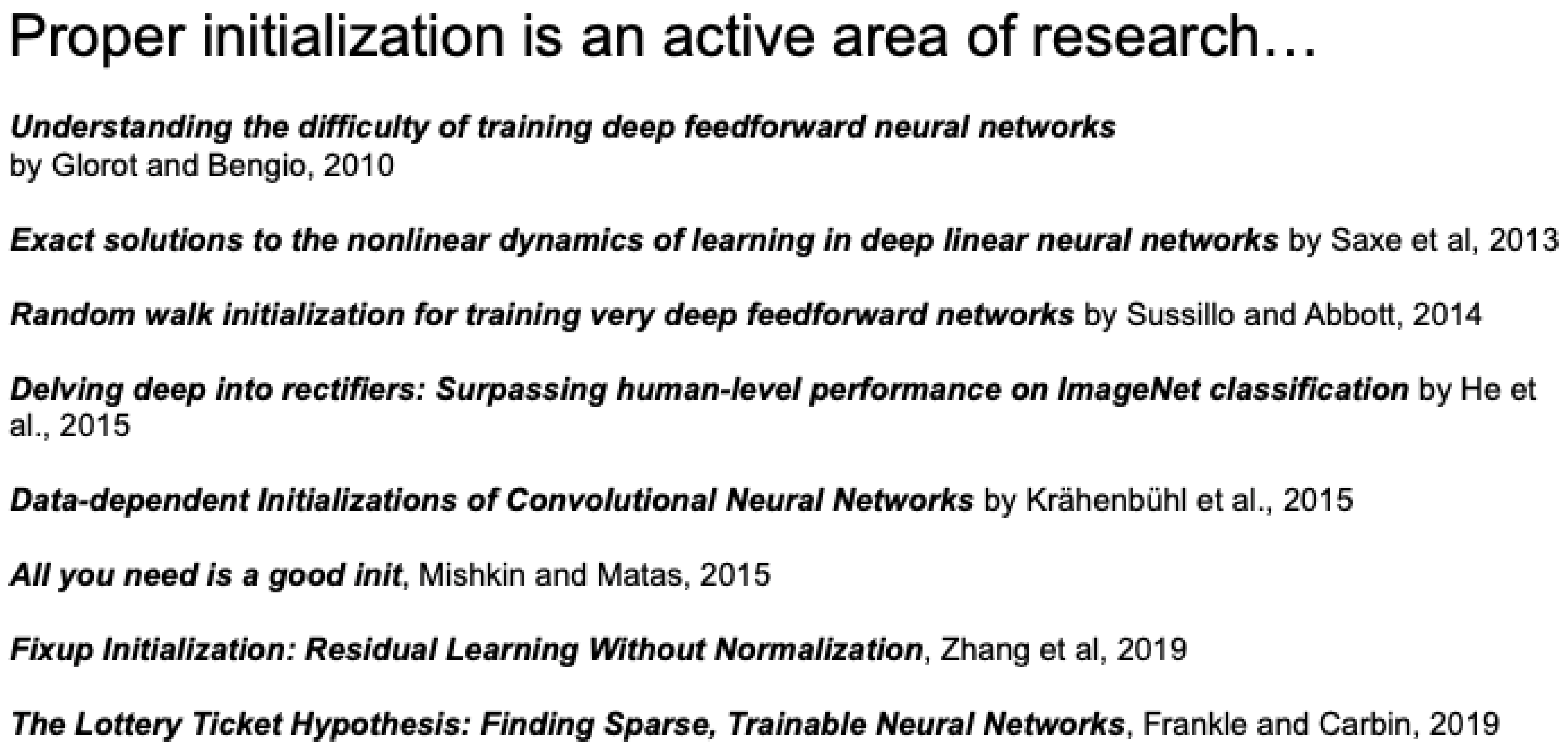
- **Xavier initialization (Glorot and Bengio, 2010): for sigmoid and tanh**

$$w \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_j}+\sqrt{n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j}+\sqrt{n_{j+1}}}\right]$$

- **Kaiming initialization (He *et al*., 2015): for ReLU and Leaky ReLU**

$$w \sim N\left[0, \frac{\sqrt{2}}{\sqrt{n_j}}\right]$$

Some Useful Tricks (from Li's notes)



Proper initialization is an active area of research…

Understanding the difficulty of training deep feedforward neural networks
by Glorot and Bengio, 2010

Exact solutions to the nonlinear dynamics of learning in deep linear neural networks by Saxe et al, 2013

Random walk initialization for training very deep feedforward networks by Sussillo and Abbott, 2014

Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification by He et al., 2015

Data-dependent Initializations of Convolutional Neural Networks by Krähenbühl et al., 2015

All you need is a good init, Mishkin and Matas, 2015

Fixup Initialization: Residual Learning Without Normalization, Zhang et al, 2019

The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks, Frankle and Carbin, 2019

Batch Normalization (from Li's notes)
[Loffe and Szegedy, 2015]

"you want zero-mean unit-variance activations? just make them so."

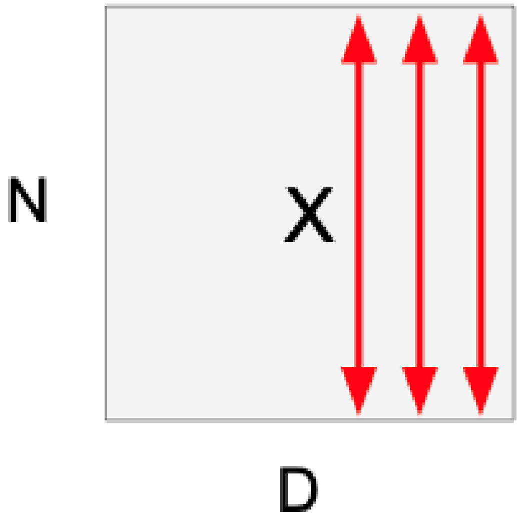
consider a batch of activations at some layer. To make each dimension zero-mean unit-variance, apply:

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

this is a vanilla differentiable function...

Batch Normalization (from Li's notes)

Input: $x : N \times D$



$$\mu_j = \frac{1}{N} \sum_{i=1}^{N} x_{i,j}$$

Per-channel mean, shape is D

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^{N} (x_{i,j} - \mu_j)^2$$

Per-channel var, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

Batch Normalization (from Li's notes)

Input: $x : N \times D$

Learnable scale and shift parameters:

$\gamma, \beta : D$

Learning $\gamma = \sigma$, $\beta = \mu$ will recover the identity function!

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

Per-channel mean, shape is D

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2$$

Per-channel var, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output, Shape is N x D

Batch Normalization (from Li’s notes)

Input: $x : N \times D$

Learnable scale and shift parameters:

$\gamma, \beta : D$

Learning $\gamma = \sigma$, $\beta = \mu$ will recover the identity function!

$$\mu_j = \frac{1}{N} \sum_{i=1}^{N} x_{i,j}$$

Per-channel mean, shape is D

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^{N} (x_{i,j} - \mu_j)^2$$

Per-channel var, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output, Shape is N x D

Batch Normalization (from Li's notes)

Input: $x : N \times D$

Learnable scale and shift parameters:

$\gamma, \beta : D$

During testing batchnorm becomes a linear operator! Can be fused with the previous fully-connected or conv layer

$\mu_j =$ (Running) average of values seen during training — Per-channel mean, shape is D

$\sigma_j^2 =$ (Running) average of values seen during training — Per-channel var, shape is D

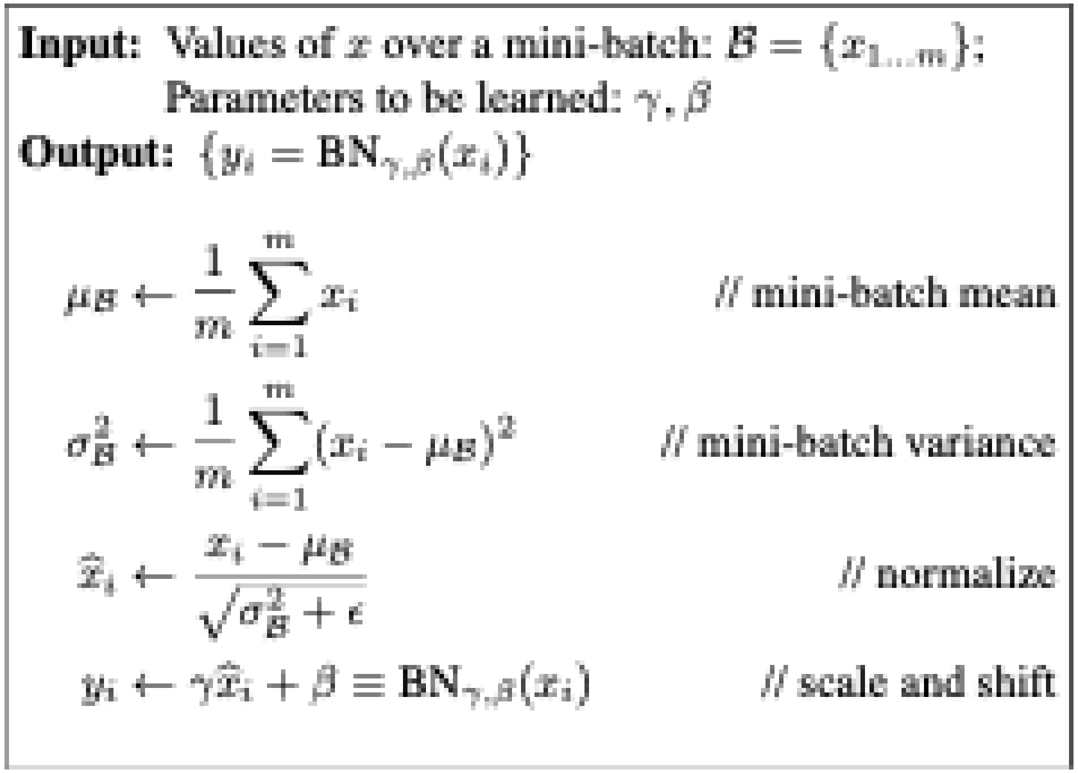
$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output, Shape is N x D

Batch Normalization [Loffe and Szegedy, 2015]

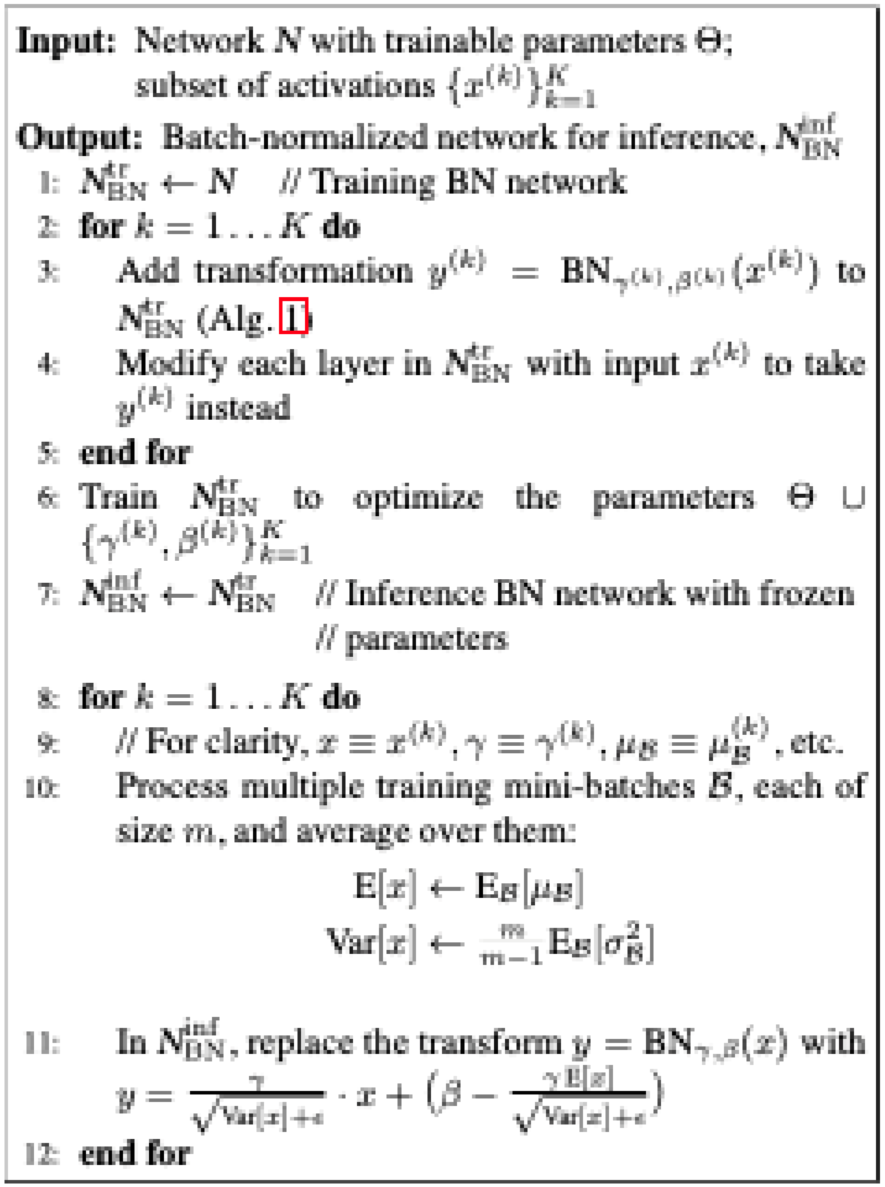


```
Input: Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
       Parameters to be learned: $\gamma, \beta$
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

    $\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$                          // mini-batch mean
    $\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\mathcal{B})^2$        // mini-batch variance
    $\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}}$                   // normalize
    $y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$           // scale and shift
```

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

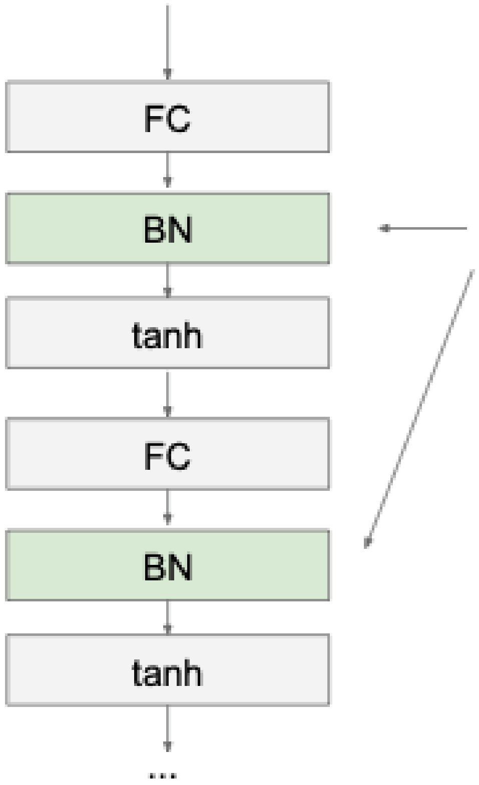
Batch Normalization [Loffe and Szegedy, 2015]



```
Input: Network $N$ with trainable parameters $\Theta$;
       subset of activations $\{x^{(k)}\}_{k=1}^{K}$
Output: Batch-normalized network for inference, $N_{\mathrm{BN}}^{\mathrm{inf}}$
 1: $N_{\mathrm{BN}}^{\mathrm{tr}} \leftarrow N$    // Training BN network
 2: for $k = 1 \ldots K$ do
 3:     Add transformation $y^{(k)} = \mathrm{BN}_{\gamma^{(k)},\beta^{(k)}}(x^{(k)})$ to
        $N_{\mathrm{BN}}^{\mathrm{tr}}$ (Alg. 1)
 4:     Modify each layer in $N_{\mathrm{BN}}^{\mathrm{tr}}$ with input $x^{(k)}$ to take
        $y^{(k)}$ instead
 5: end for
 6: Train $N_{\mathrm{BN}}^{\mathrm{tr}}$ to optimize the parameters $\Theta \cup$
    $\{\gamma^{(k)}, \beta^{(k)}\}_{k=1}^{K}$
 7: $N_{\mathrm{BN}}^{\mathrm{inf}} \leftarrow N_{\mathrm{BN}}^{\mathrm{tr}}$    // Inference BN network with frozen
                        // parameters
 8: for $k = 1 \ldots K$ do
 9:     // For clarity, $x \equiv x^{(k)}, \gamma \equiv \gamma^{(k)}, \mu_{\mathcal{B}} \equiv \mu_{\mathcal{B}}^{(k)}$, etc.
10:     Process multiple training mini-batches $\mathcal{B}$, each of
        size $m$, and average over them:
            $\mathrm{E}[x] \leftarrow \mathrm{E}_{\mathcal{B}}[\mu_{\mathcal{B}}]$
            $\mathrm{Var}[x] \leftarrow \frac{m}{m-1}\mathrm{E}_{\mathcal{B}}[\sigma_{\mathcal{B}}^2]$
11:     In $N_{\mathrm{BN}}^{\mathrm{inf}}$, replace the transform $y = \mathrm{BN}_{\gamma,\beta}(x)$ with
        $y = \frac{\gamma}{\sqrt{\mathrm{Var}[x]+\epsilon}} \cdot x + \left(\beta - \frac{\gamma\,\mathrm{E}[x]}{\sqrt{\mathrm{Var}[x]+\epsilon}}\right)$
12: end for
```

Algorithm 2: Training a Batch-Normalized Network

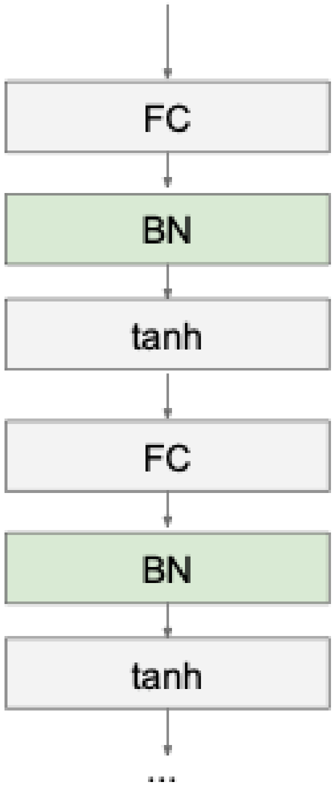
Batch Normalization (from Li's notes)



Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

$$\boxed{\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}}$$

Batch Normalization (from Li's notes)



- Makes deep networks **much** easier to train!
- Improves gradient flow
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as regularization during training
- Zero overhead at test-time: can be fused with conv!
- Behaves differently during training and testing: this is a very common source of bugs!

Some Useful Tricks

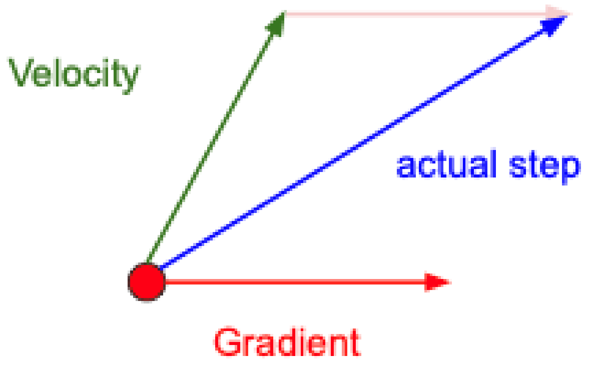
- **Use of momentum term allows the effective learning rate for each weight to adapt as needed and helps speed up convergence – in a network with 2 layers of weights,**

$$\left.\begin{aligned}
w_{ji}(t+1) &= w_{ji}(t) + \Delta w_{ji}(t) \\
\Delta w_{ji}(t) &= -\eta \left.\frac{\partial E_S}{\partial w_{ji}}\right|_{w_{ji}=w_{ji}(t)} + \alpha \Delta w_{ji}(t-1) \\
w_{kj}(t+1) &= w_{kj}(t) + \Delta w_{kj}(t) \\
\Delta w_{kj}(t) &= -\eta \left.\frac{\partial E_S}{\partial w_{kj}}\right|_{w_{kj}=w_{kj}(t)} + \alpha \Delta w_{kj}(t-1)
\end{aligned}\right\} \text{where } 0<\alpha,\eta<1 \text{ with typical values of } \eta = 0.5 \text{ to } 0.6, \alpha = 0.8 \text{ to } 0.9$$

- **Nesterov Momentum (Sutskever *et al*., 2013): Gradient is evaluated *after* the current velocity is applied**

Nesterov Momentum (from Li's notes)

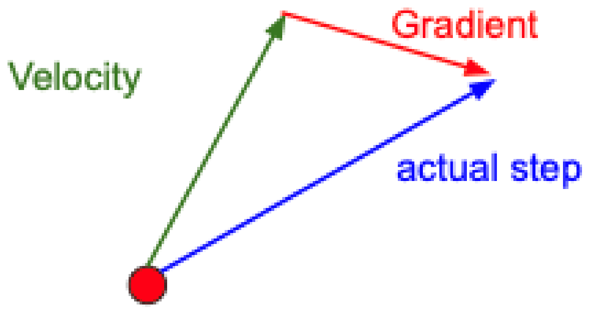
Momentum update:



Combine gradient at current point with velocity to get step used to update weights

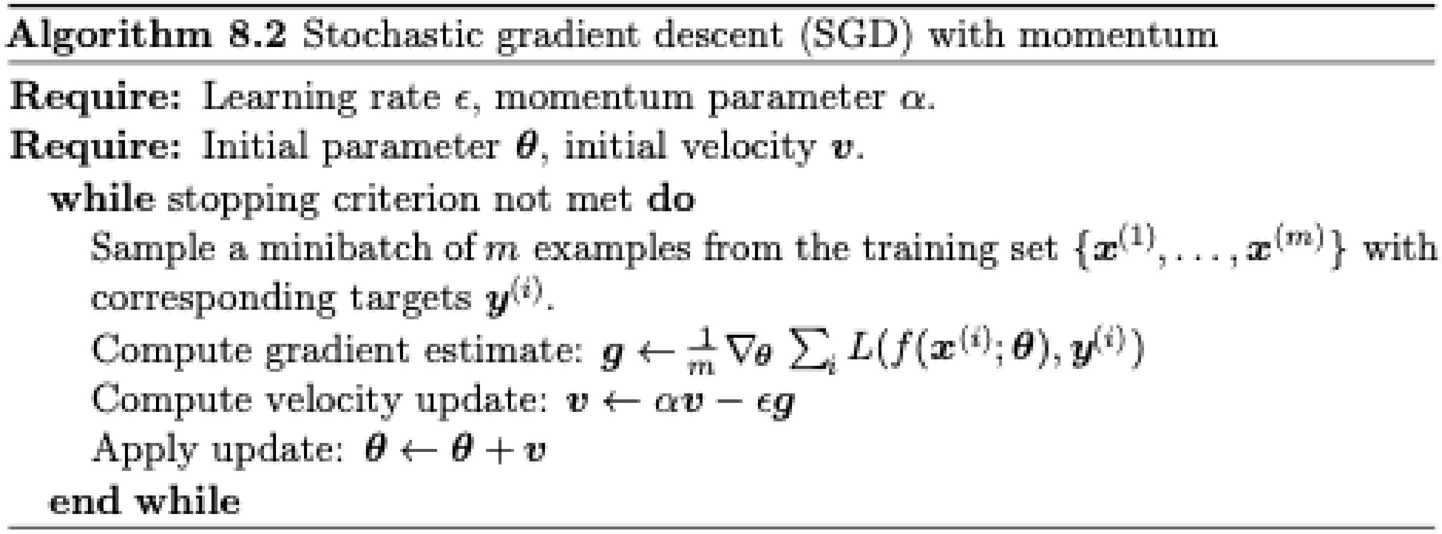
Nesterov, "A method of solving a convex programming problem with convergence rate O(1/k^2)", 1983
Nesterov, "Introductory lectures on convex optimization: a basic course", 2004
Sutskever et al, "On the importance of initialization and momentum in deep learning", ICML 2013

Nesterov Momentum



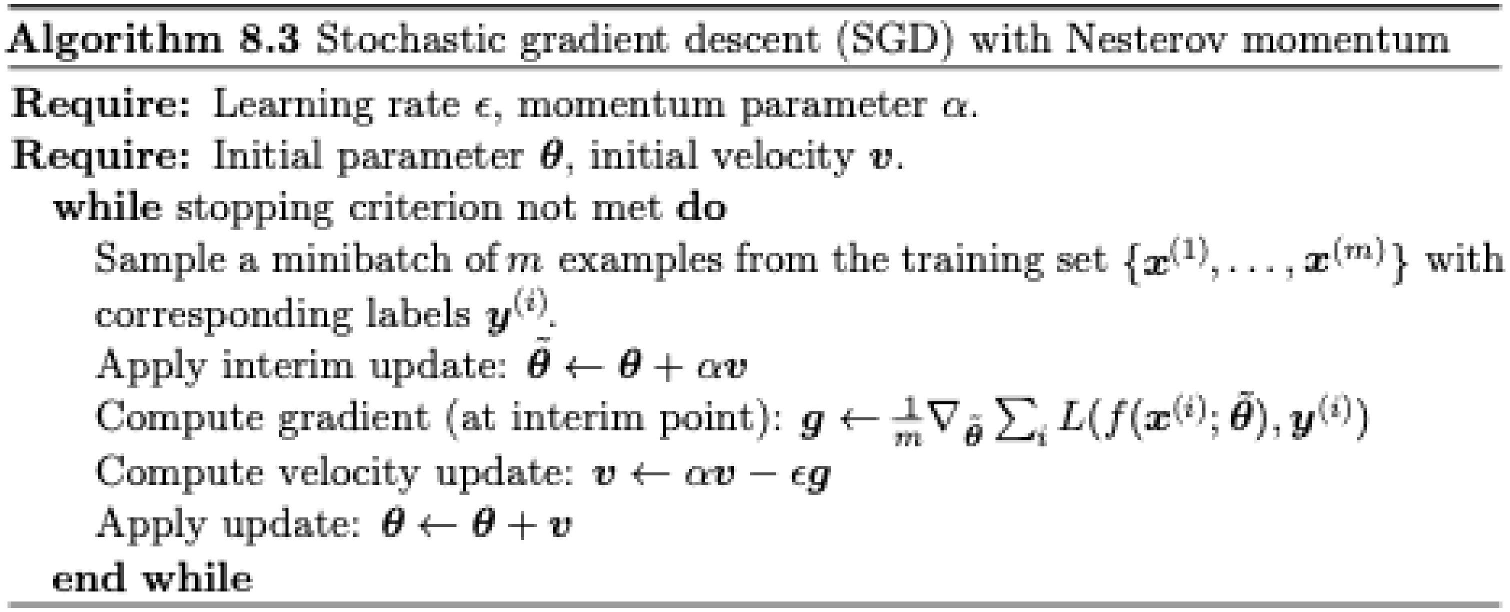
"Look ahead" to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

Some Useful Tricks



Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

```
Require: Learning rate $\epsilon$, momentum parameter $\alpha$.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with
    corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Compute velocity update: $\boldsymbol{v} \leftarrow \alpha\boldsymbol{v} - \epsilon\boldsymbol{g}$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
  end while
```



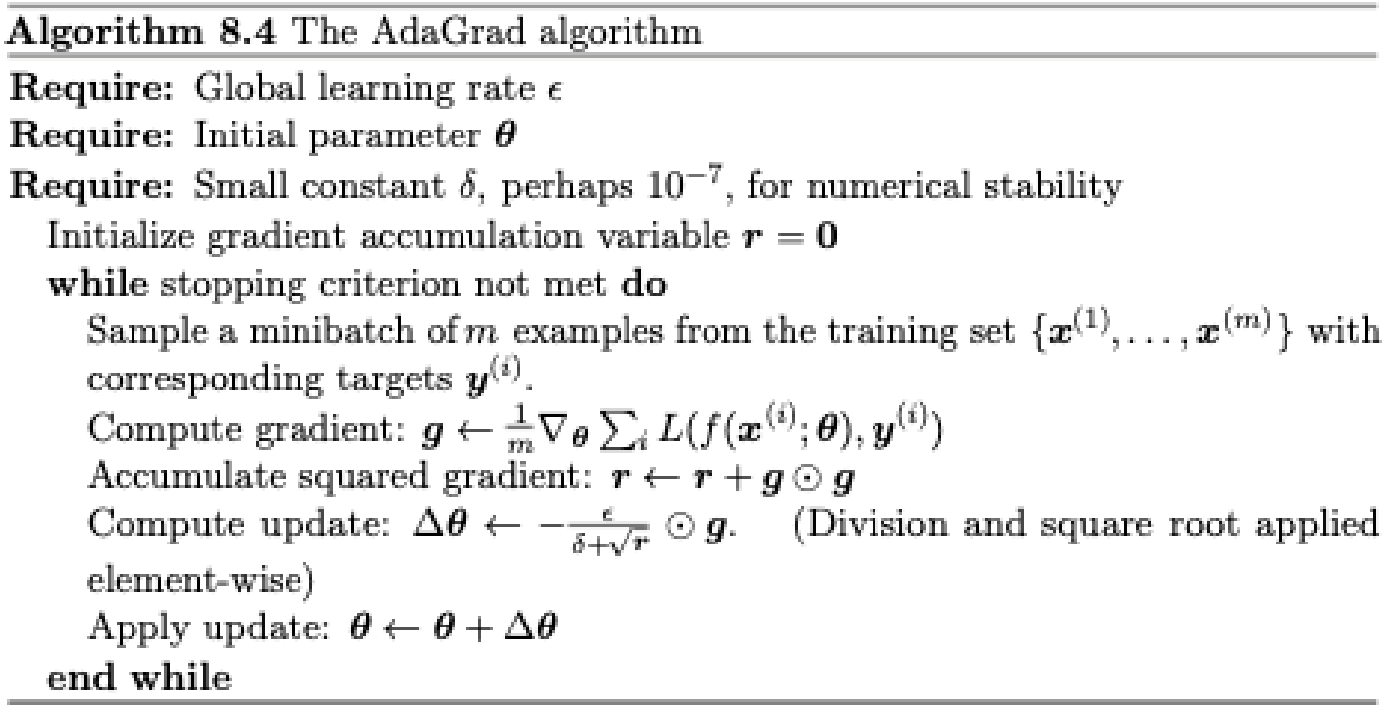
Algorithm 8.3 Stochastic gradient descent (SGD) with Nesterov momentum

```
Require: Learning rate $\epsilon$, momentum parameter $\alpha$.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with
    corresponding labels $\boldsymbol{y}^{(i)}$.
    Apply interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha\boldsymbol{v}$
    Compute gradient (at interim point): $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)};\tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$
    Compute velocity update: $\boldsymbol{v} \leftarrow \alpha\boldsymbol{v} - \epsilon\boldsymbol{g}$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
  end while
```

Some Useful Tricks

- **Algorithms with Adaptive Learning Rates**

- **AdaGrad (Adaptive Gradient, Duchi *et al.*, 2011): scale rates inversely proportional to the square root of the sum of all the historical squared values of the gradient**

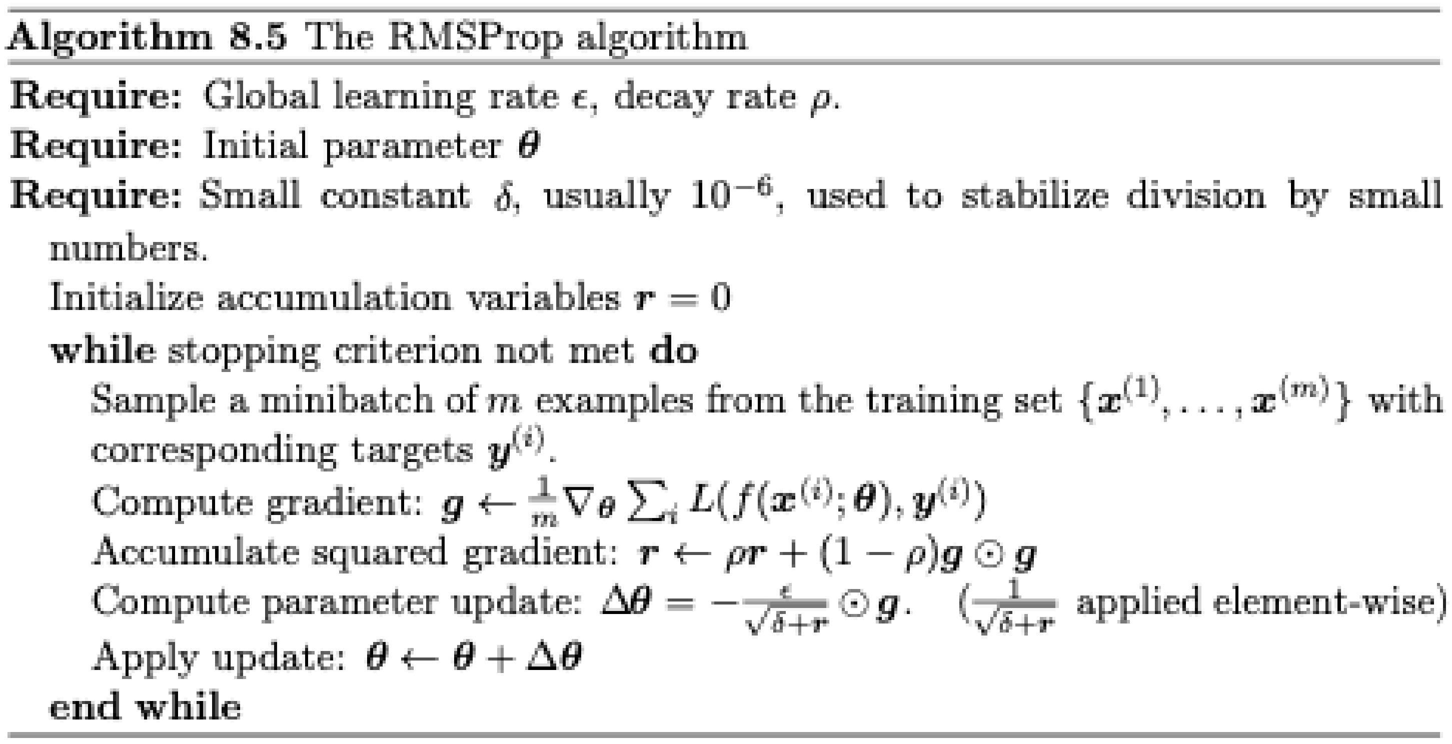


Algorithm 8.4 The AdaGrad algorithm

```
Require: Global learning rate $\epsilon$
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant $\delta$, perhaps $10^{-7}$, for numerical stability
  Initialize gradient accumulation variable $\boldsymbol{r} = \boldsymbol{0}$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with
    corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$
    Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$.   (Division and square root applied
    element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while
```

Some Useful Tricks

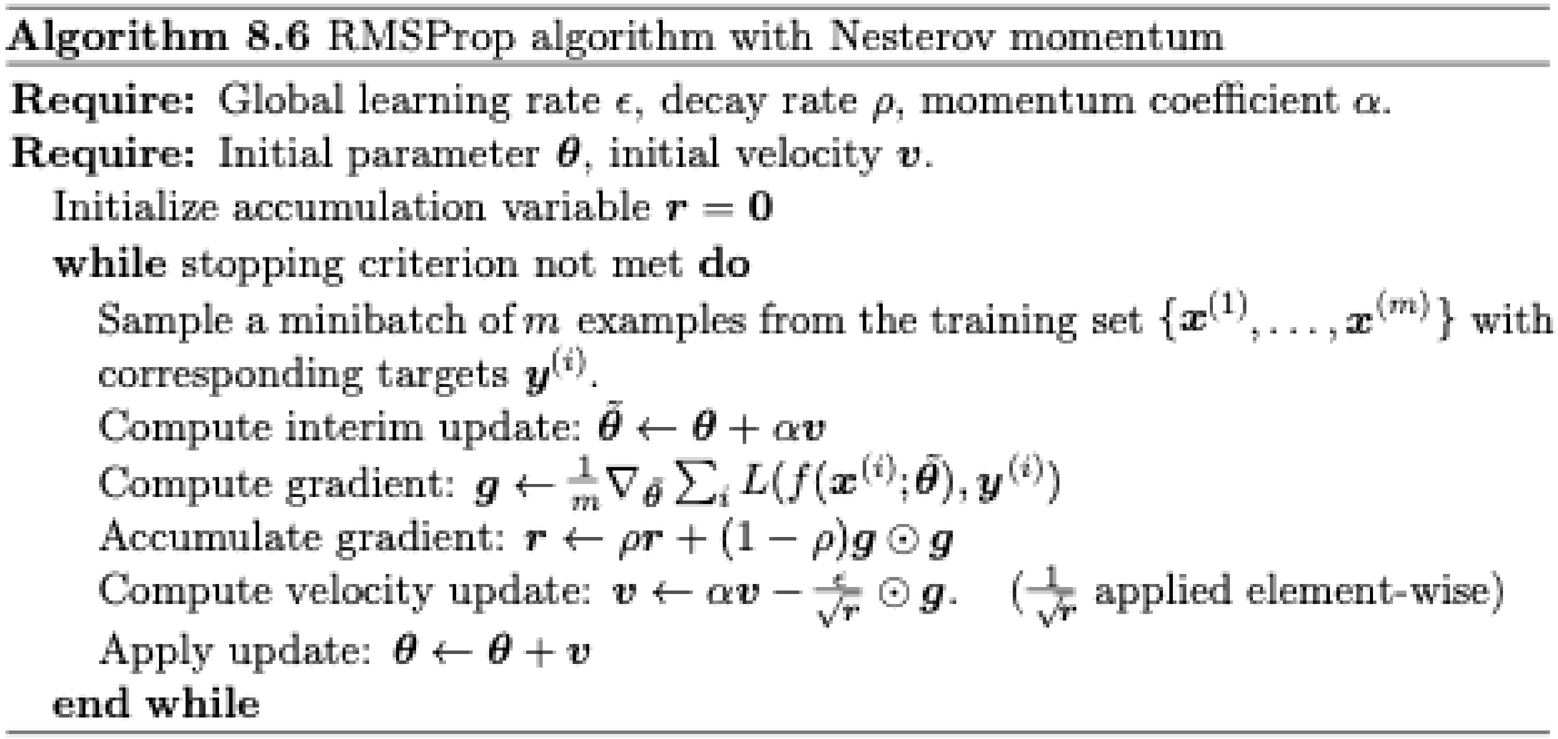
- **RMSProp (Hinton, 2012): modifies AdaGrad to perform better in nonconvex setting by changing the gradient accumulation into an exponentially decaying average**



Algorithm 8.5 The RMSProp algorithm

```
Require: Global learning rate $\epsilon$, decay rate $\rho$.
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant $\delta$, usually $10^{-6}$, used to stabilize division by small
  numbers.
  Initialize accumulation variables $\boldsymbol{r} = 0$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with
    corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$
    Compute parameter update: $\Delta\boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$.   ($\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while
```

Some Useful Tricks



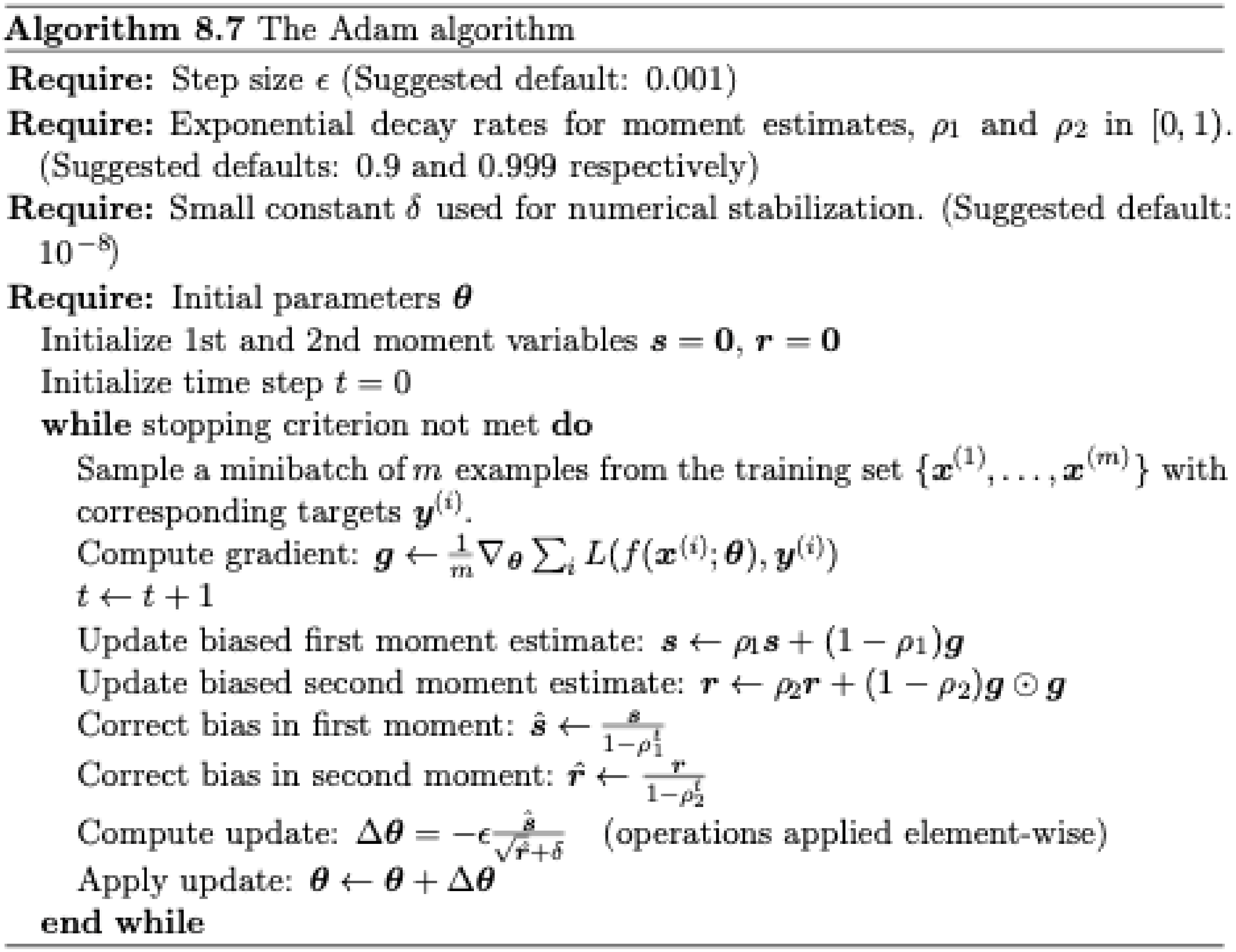
Algorithm 8.6 RMSProp algorithm with Nesterov momentum

```
Require: Global learning rate $\epsilon$, decay rate $\rho$, momentum coefficient $\alpha$.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
  Initialize accumulation variable $\boldsymbol{r} = \boldsymbol{0}$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with
    corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$
    Accumulate gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1-\rho) \boldsymbol{g} \odot \boldsymbol{g}$
    Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \frac{\epsilon}{\sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$.   ($\frac{1}{\sqrt{\boldsymbol{r}}}$ applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
  end while
```

Some Useful Tricks

- Adam (Adaptive Moments, Kingma and Ba, 2015): RMSProp + SGD with momentum
  - Momentum is incorporated as an estimate of the first-order moment with exponential weighting of the gradient
  - Includes bias corrections to estimates of both the 1st order moments (momentum term) and the second order moments to account for their initialization at the origin

Some Useful Tricks

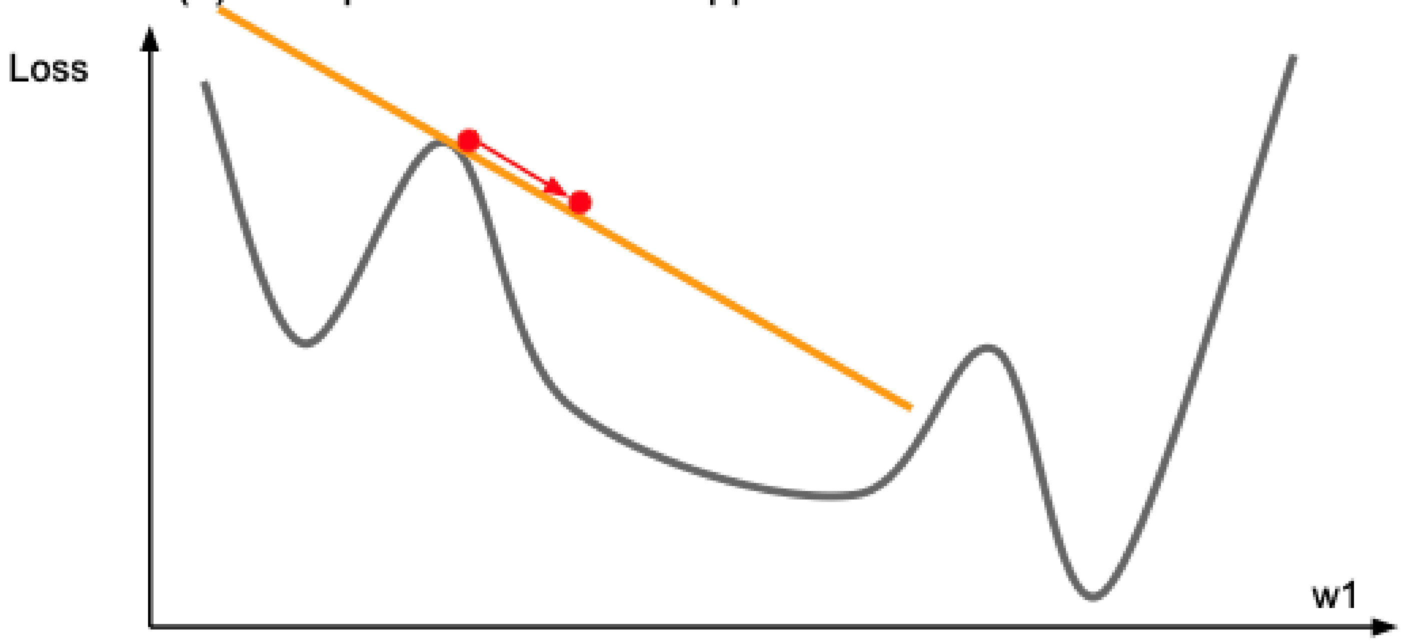


Algorithm 8.7 The Adam algorithm

```
Require: Step size $\epsilon$ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, $\rho_1$ and $\rho_2$ in $[0, 1)$.
  (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant $\delta$ used for numerical stabilization. (Suggested default:
  $10^{-8}$)
Require: Initial parameters $\boldsymbol{\theta}$
  Initialize 1st and 2nd moment variables $\boldsymbol{s} = \boldsymbol{0}$, $\boldsymbol{r} = \boldsymbol{0}$
  Initialize time step $t = 0$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with
    corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    $t \leftarrow t + 1$
    Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
    Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$
    Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\rho_1^t}$
    Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1-\rho_2^t}$
    Compute update: $\Delta\boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}}+\delta}$   (operations applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while
```

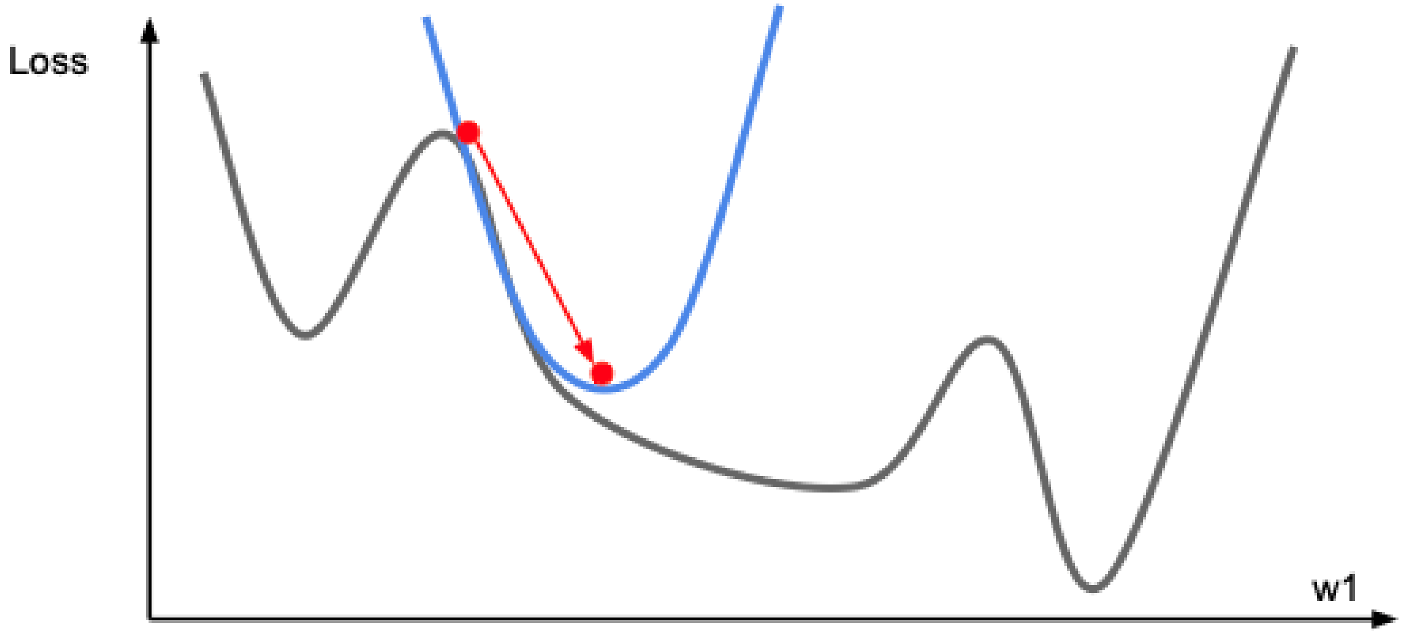
First-Order Optimization (from Li's notes)

(1) Use gradient form linear approximation
(2) Step to minimize the approximation



Second-Order Optimization (from Li's notes)

(1) Use gradient **and Hessian** to form **quadratic** approximation
(2) Step to the **minima** of the approximation



Second-Order Optimization (from Li's notes)

second-order Taylor expansion:

$$\boxed{J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)}$$

Solving for the critical point we obtain the Newton parameter update:

$$\boxed{\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1}\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)}$$

Hessian has O(N^2) elements
Inverting takes O(N^3)
N = (Tens or Hundreds of) Millions

Q: Why is this bad for deep learning?

Second-Order Optimization (from Li's notes)

$$\boxed{\theta^* = \theta_0 - H^{-1} \nabla_{\theta} J(\theta_0)}$$

- Quasi-Newton methods (**BGFS** most popular):
  instead of inverting the Hessian (O(n^3)), approximate inverse Hessian with rank 1 updates over time (O(n^2) each).

- **L-BFGS** (Limited memory BFGS):
  Does not form/store the full inverse Hessian.

In Practice (from Li’s notes)

- **Adam** is a good default choice in many cases; it often works ok even with constant learning rate
- **SGD+Momentum** can outperform Adam but may require more tuning of LR and schedule
  - Try cosine schedule, very few hyperparameters!

- If you can afford to do full batch updates then try out **L-BFGS** (and don’t forget to disable all sources of noise)

Some Useful Tricks

- Use sigmoid function which satisfies φ(–z)=–φ(z) helps speed up convergence

$$\varphi(z) = a\left(\frac{e^{bz} - e^{-bz}}{e^{bz} + e^{-bz}}\right)$$

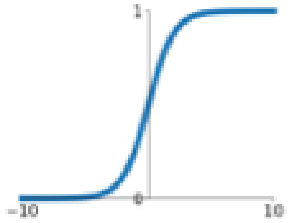
$$a = 1.716, \mathrm{b} = \frac{2}{3} \Rightarrow \left.\frac{\partial\varphi}{\partial z}\right|_{z=0} \approx 1$$

and $\varphi(z)$ is linear in the range $-1 < z < 1$

Some Useful Tricks (from Li's notes)

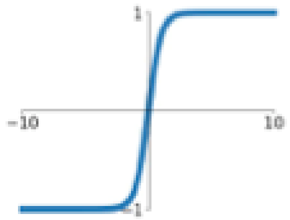
Sigmoid

$\sigma(x) = \frac{1}{1+e^{-x}}$



tanh

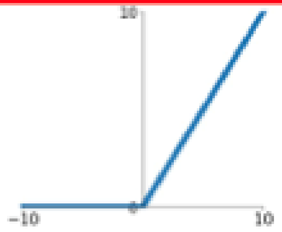
$\tanh(x)$



ReLU

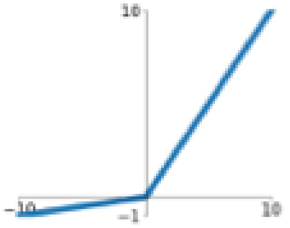
$\max(0, x)$

Good default choice



Leaky ReLU

$\max(0.1x, x)$

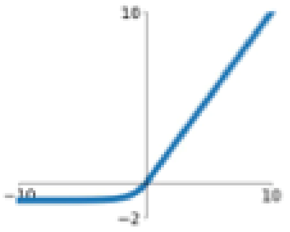


Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Some Useful Tricks

- Randomizing the order of presentation of training examples from one pass to the next helps avoid local minima
- Introducing small amounts of noise in the weight updates (or into examples) during training helps improve generalization – minimizes over fitting, makes the learned approximation more robust to noise, and helps avoid local minima
- If using the suggested sigmoid nodes in the output layer, set target output for output nodes to be 1 for target class and -1 for all others

Some useful tricks

- Regularization helps avoid over fitting and improves generalization

$$R(\mathbf{W}) = \lambda E(\mathbf{W}) + (1-\lambda)C(\mathbf{W}); \ 0 \le \lambda \le 1$$

$$C(\mathbf{W}) = \frac{1}{2}\left(\sum_{ji} w_{ji}^2 + \sum_{kj} w_{kj}^2\right)$$

$$-\frac{\partial C}{\partial w_{ji}} = -w_{ji} \text{ and } -\frac{\partial C}{\partial w_{kj}} = -w_{kj}$$

Start with λ close to 1 and gradually lower it during training. When λ <1, it tends to drive weights toward zero setting up a tension between error reduction and complexity minimization

Some Useful Tricks

Input and output encodings:

- **Do not eliminate *natural* proximity in the input or output space**
  - **Do *not* normalize input patterns to be of unit length if the length is likely to be relevant for distinguishing between classes**

- **Do not introduce *unwarranted* proximity as an artifact**
  - **Do *not* use $\log_2$ M outputs to encode M classes, use M outputs instead to avoid spurious proximity in the output space**

- **Use *error correcting codes* when feasible**

Some Useful Tricks

Examples of a good code:

- **Binary thermometer codes for encoding real values**
  - **Suppose we can use 10 bits to represent a value between -1.0 and +1.0**
  - **We can quantize the interval [-1, 1] into 10 equal parts**
  - **0.38 in thermometer code is 1111000000**
  - **0.60 in thermometer code is 1111110000**
  - **Note values that are close along the real number line have thermometer codes that are close in Hamming distance**

Example of a bad code:

- **Ordinary binary representations of integers**

Some Useful Tricks

- Use of problem specific information (if known) speeds up convergence and improves generalization

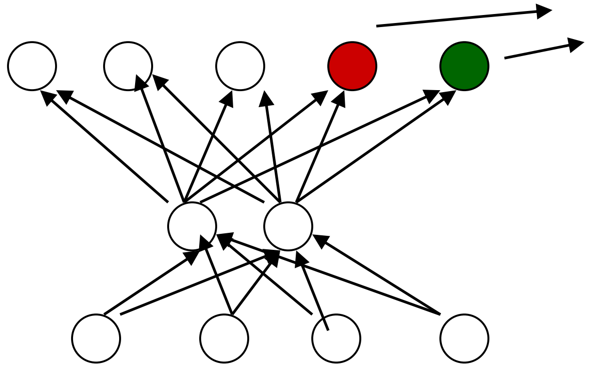
- In networks designed for translation-invariant visual image classification, building in translation invariance as a constraint on the weights helps

- If we know the function to be approximated is smooth, we can build that in as part of the criterion to be minimized – minimize in addition to the error, the gradient of the error with respect to the inputs

Some Useful Tricks

- Manufacture training data – training networks with translated and rotated patterns if translation and rotation invariant recognition is desired

- Incorporating hints during training
- Hints are used as additional outputs during training to help shape the hidden layer representation



Hint nodes (e.g., vowels versus consonants in training a phoneme recognizer)

Some Useful Tricks

- Reducing the effective number of free parameters (degrees of freedom) helps improve generalization

- Regularization

- Preprocess the data to reduce the dimensionality of the input –
  - Train a neural network with output same as input, but with fewer hidden neurons than the number of inputs
  - Use the hidden layer outputs as inputs to a second network to do function approximation

Some Useful Tricks

- Choice of appropriate error function is critical – do not blindly minimize sum squared error – there are many cases where other criteria are appropriate

- Example

$$E_S(\mathbf{W}) = \sum_{p=1}^{P} \sum_{k=1}^{M} t_{kp} \ln\left(\frac{t_{kp}}{z_{kp}}\right)$$

is appropriate for minimizing the distance between the target probability distribution over the *M* output variables and the probability distribution represented by the network

Some Useful Tricks

- **Interpreting the outputs as class conditional probabilities**

- **Use <u>linear</u> output nodes <u>or</u> <u>exponential</u> <u>but not sigmoid</u> output nodes**

$$n_{kp} = \sum_{j=0}^{n_H} w_{kj} y_{jp}$$

$$\text{linear output } z_{kp} = \left(\frac{n_{kp}}{\sum_{l=1}^{M} n_{lp}} \right)$$

$$\text{exponential output } z_{kp} = \left(\frac{e^{n_{kp}}}{\sum_{l=1}^{M} e^{n_{lp}}} \right)$$

Other Topics on ANN

- Constructive/Generative neural network learning
- Hopfield networks with Hebbian learning
  - Associative memory
  - Optimization
- Adaptive Resonance Theory (ART) and its variants
- Kohonen's Self Organizing Map (SOM)
- …