Data Mining and Machine Learning

Part 3: Neural Nets, SVMs, and Advanced Topics

Melih Kandemir

University of Southern Denmark

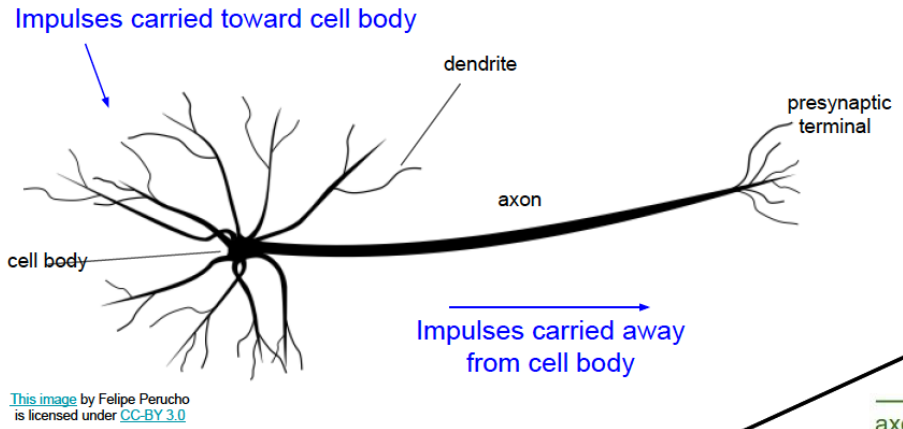
Spring 2022

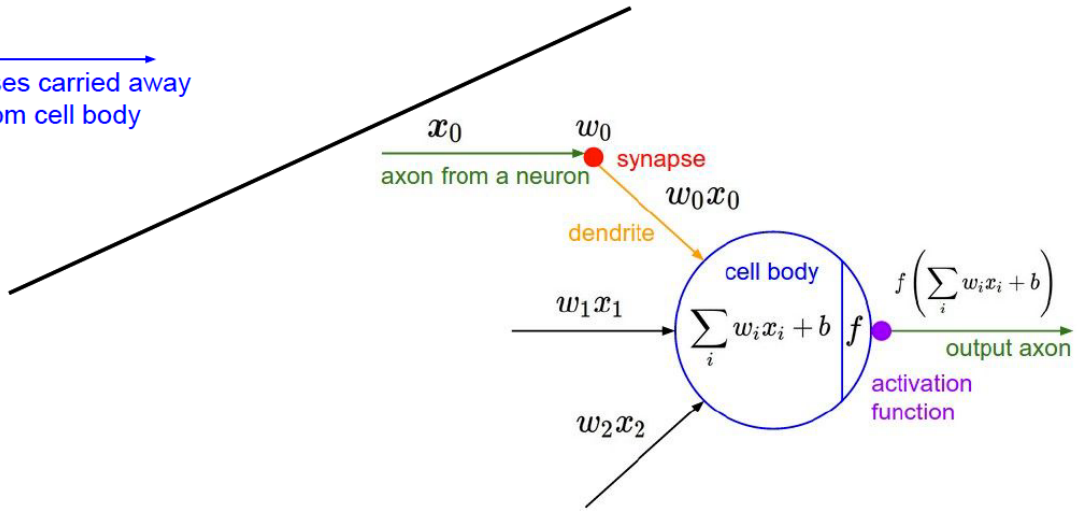
Motivation

- Neural networks have been developed as a machine and computational model, simulating some principles of biological brains:
  - neural network (or short: neural net) is a set of neurons that are connected by edges
  - neuron: mimics the function of a biological neuron, being activated by input-signals at synapses and firing impulses to other neurons under certain conditions
- Human brain:
  - 10^{11} neurons each connected with approx. 10^4 others
  - fastest response time: 10^{-3} sec
    (3GHz processor: approx. 10^{-10} sec.)
  - derives complex decisions very fast (recognizing the face of your mother typically takes 100 steps, i.e., 10^{-1} sec.)

Perceptron: The atom of intelligence

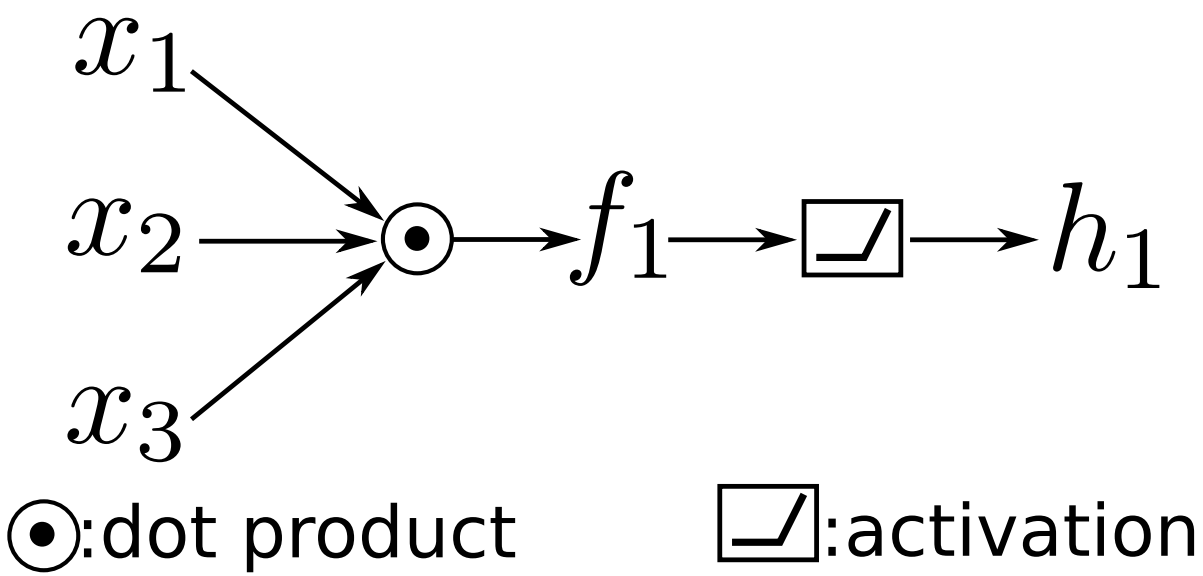
North-West: Biological neuron
South-East: Artificial neuron (perceptron)



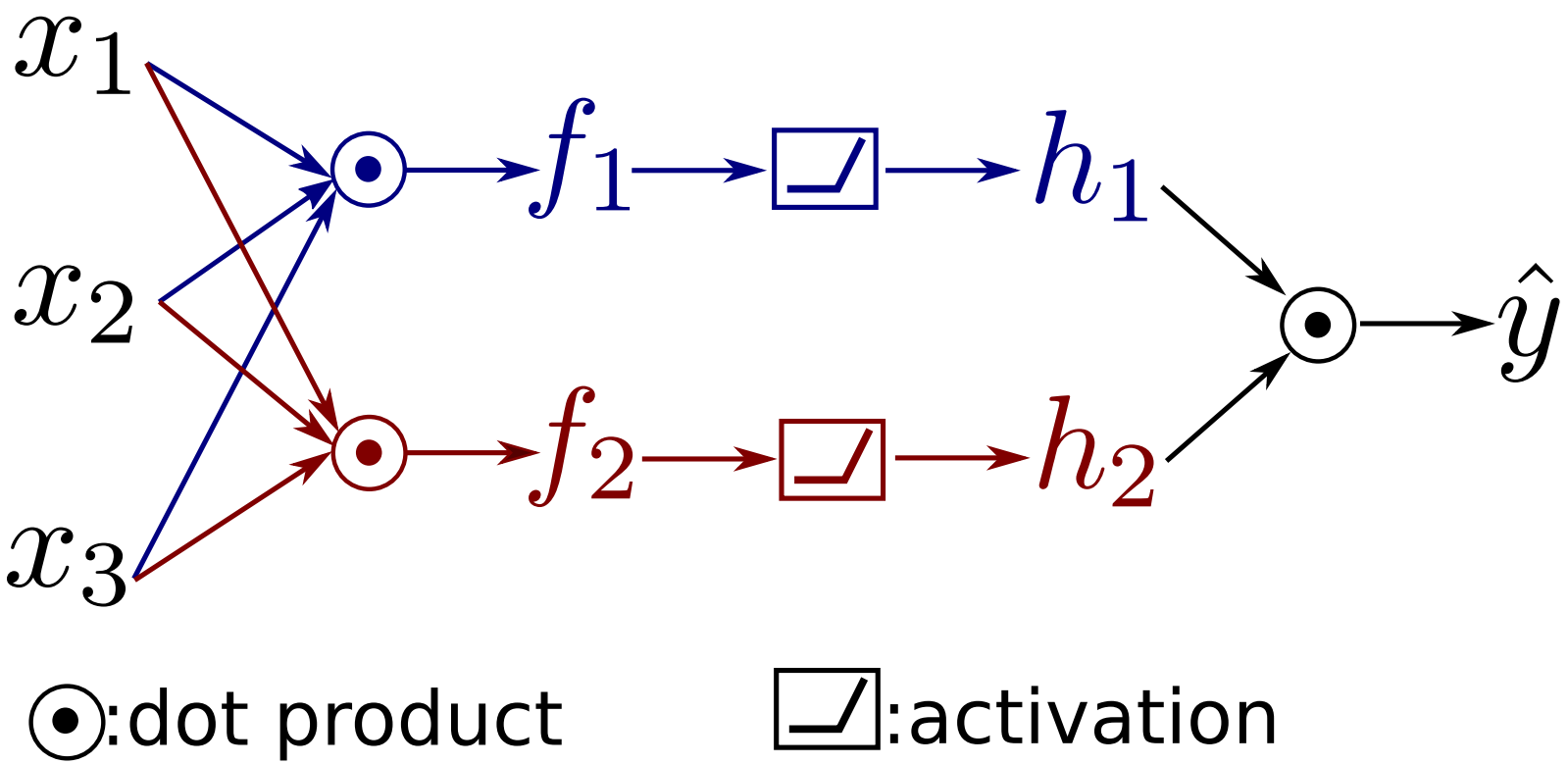


The computational graph of a perceptron

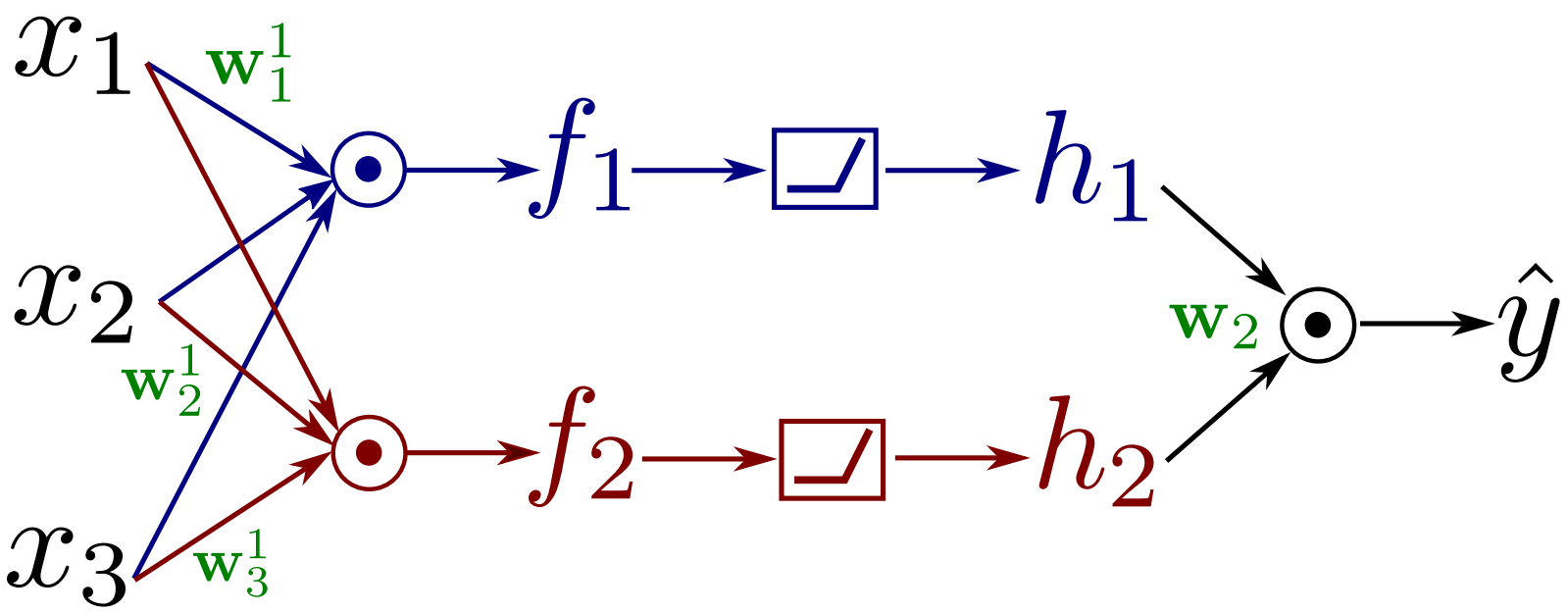
Computational graph: block diagram of mathematical operations.



Neural net: A group of connected perceptrons



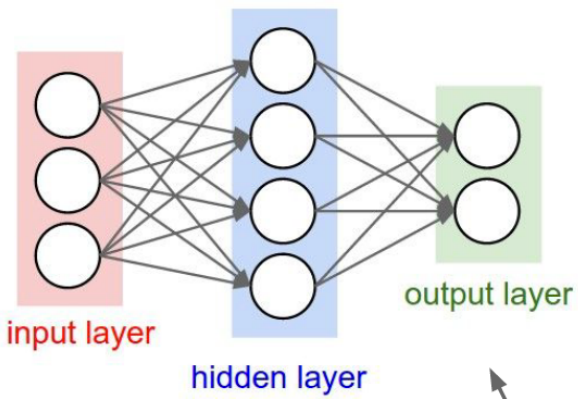
Neural net: A group of connected perceptrons

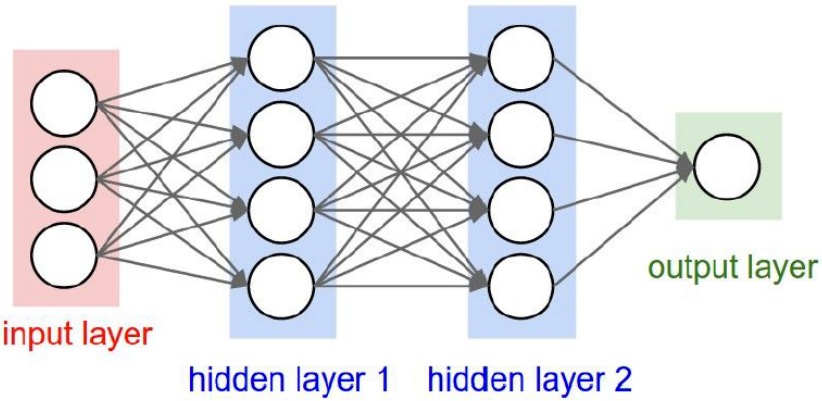


$\odot$:dot product $\_/$:activation

$\mathbf{W} = \{\mathbf{w}_1^1, \mathbf{w}_2^1, \mathbf{w}_3^1, \mathbf{w}_2\}$:params

The Multi-Layer Perceptron (MLP)





"2-layer Neural Net", or
"1-hidden-layer Neural Net"

"3-layer Neural Net", or
"2-hidden-layer Neural Net"

"Fully-connected" layers

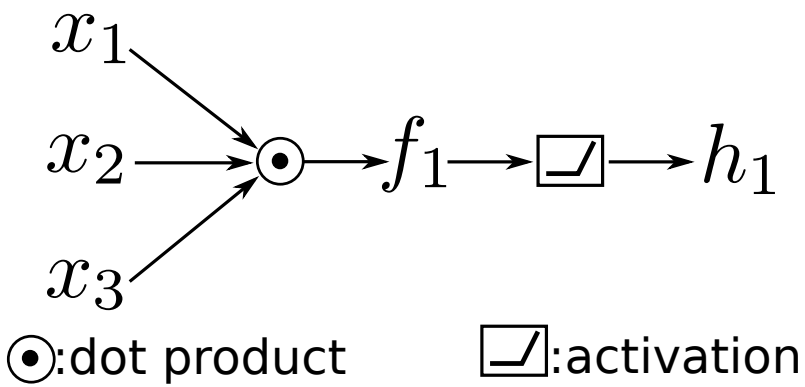
Formal definition of the perceptron

Define the input vector as $\mathbf{x} = \{x_1, x_2, x_3\}$ and the activation function as $v = \sigma(u)$. Then, the perceptron i is

$$f_i = \mathbf{w}_i^T \mathbf{x},$$
$$h_i = \sigma(f_i),$$

or in short hand

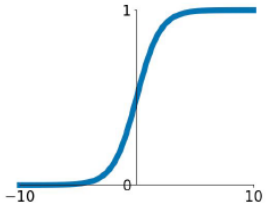
$$h_i = \sigma(\mathbf{w}_i^T \mathbf{x}).$$



Activation functions

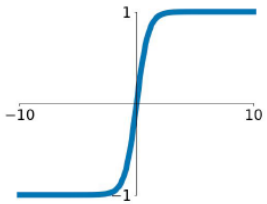
Sigmoid

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



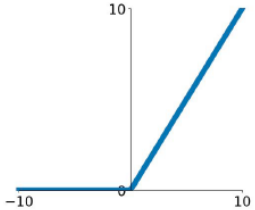
tanh

$$\tanh(x)$$



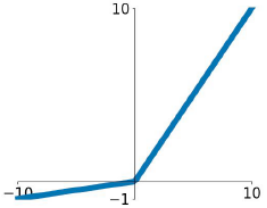
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

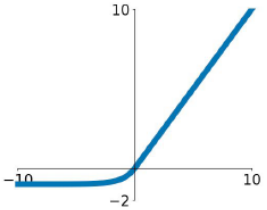


Maxout

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

ELU

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



Formal definition of the neural net

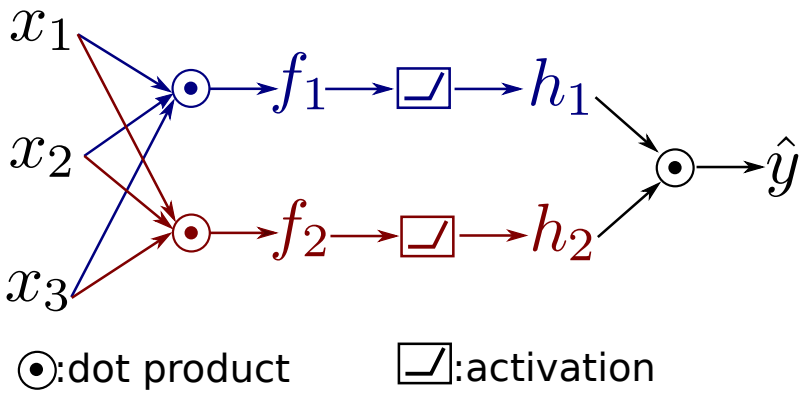
Define $\mathbf{h} = [h_1, h_2]$ and $\mathbf{v} = \sigma(\mathbf{u}) = [\sigma(u_1), \sigma(u_2)]$, then

$$\mathbf{f} = \mathbf{W}_1^T \mathbf{x},$$
$$\mathbf{h} = \sigma(\mathbf{f}),$$
$$\hat{y} = \mathbf{w}_2^T \mathbf{h},$$

or combined

$$\hat{y} = \mathbf{w}_2^T \sigma(\mathbf{W}_1^T \mathbf{x}).$$

Here, $\mathbf{h}$ is the output of the first (in this case the only) *hidden layer*. It is also referred to as the *activation map* of Layer 1.



The architecture of a neural net

The architecture consists of the design choices below:

- Number of input/output nodes: Determined by data
- Number of hidden layers
- Numbers of perceptrons for a hidden layer (does not have to be equal for different layers)
- The activation function of a hidden layer

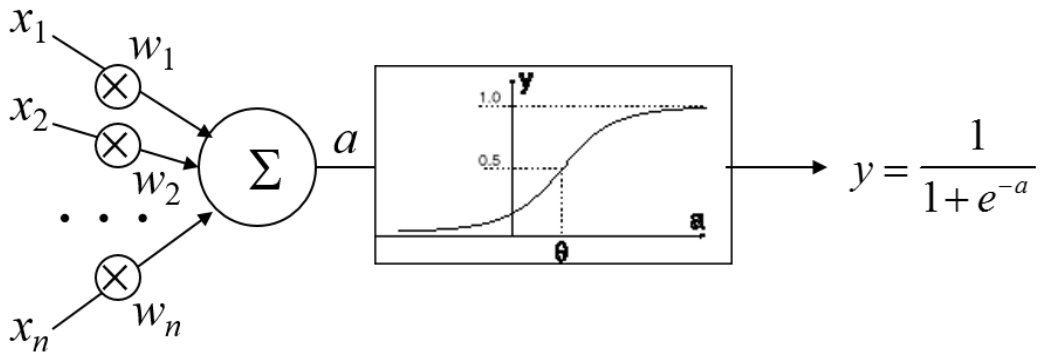
There are other architectural elements in modern machine learning such as Convolutional layers, Pooling layers, BatchNorm/LayerNorm layers, Transformer layers, Residual Layers, etc.
Neural Architecture Search (NAS) was research until recent. It has been automated now. There exist neural nets that search for optimal architectures for other neural nets!

Neurons

general neuron (a.k.a. perceptron):

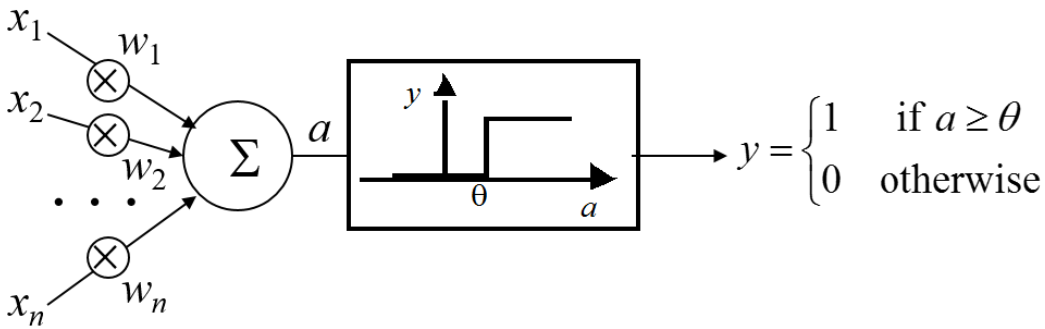
activation value a

$$a = \sum_{i=1}^{n} w_i \cdot x_i$$



threshold logic unit (TLU):

all or nothing,
threshold θ

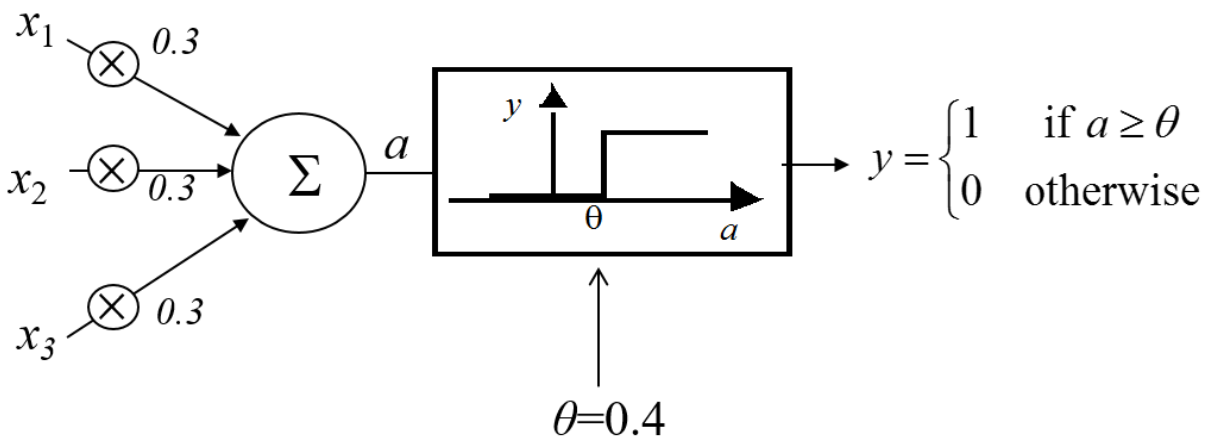


Example: Model for a Boolean Function

$$f(x) = [x_1 \wedge (x_2 \vee x_3)] \vee (x_2 \wedge x_3)$$

Truth Table

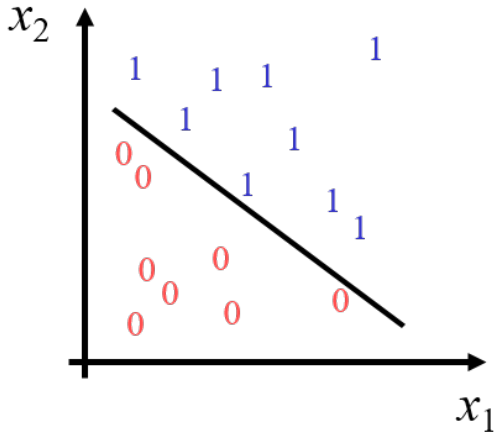
x_1	x_2	x_3	y
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1
0	0	1	0
0	1	0	0
0	1	1	1
0	0	0	0



To build the classifier, we need to learn the weights w_1, w_2, w_3.

Geometric Interpretation of a TLU Model

- $\sum_{i=1}^{d} w_i \cdot x_i = \theta$: hyperplane in d-dimensional space
- TLU learnes a separating hyperplane: examples of class 0 should be on one side, examples of class 1 on the other.

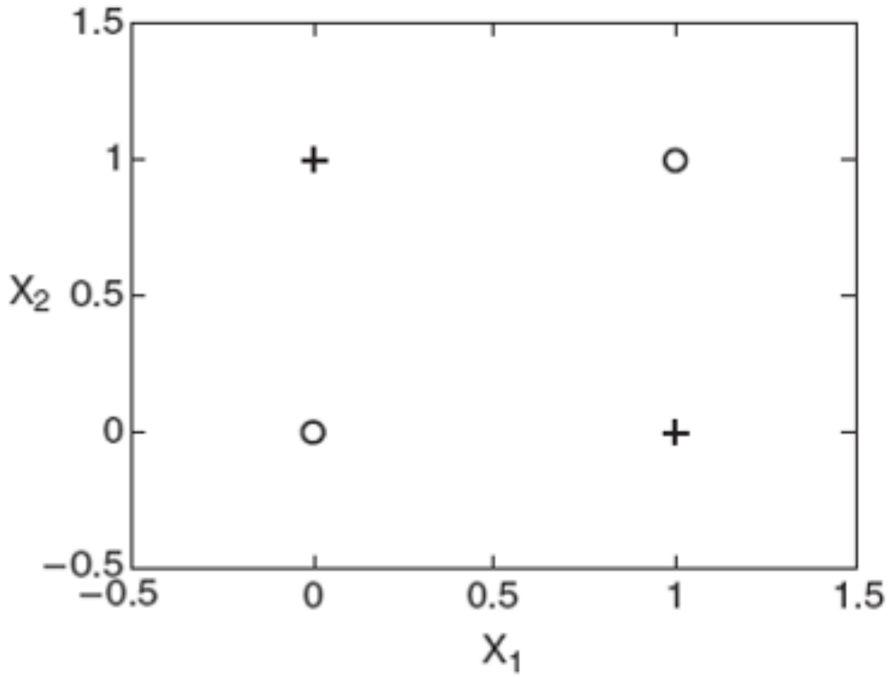


- The training of a TLU is about adapting iteratively the weights w_i until all training examples are separated by the hyperplane according to their respective class.
- Training step: rotation of the hyperplane defined by w_i and θ in the direction of point v if v is not (yet) on the correct side of the hyperplane.

Not Linearly Separable Learning Problem

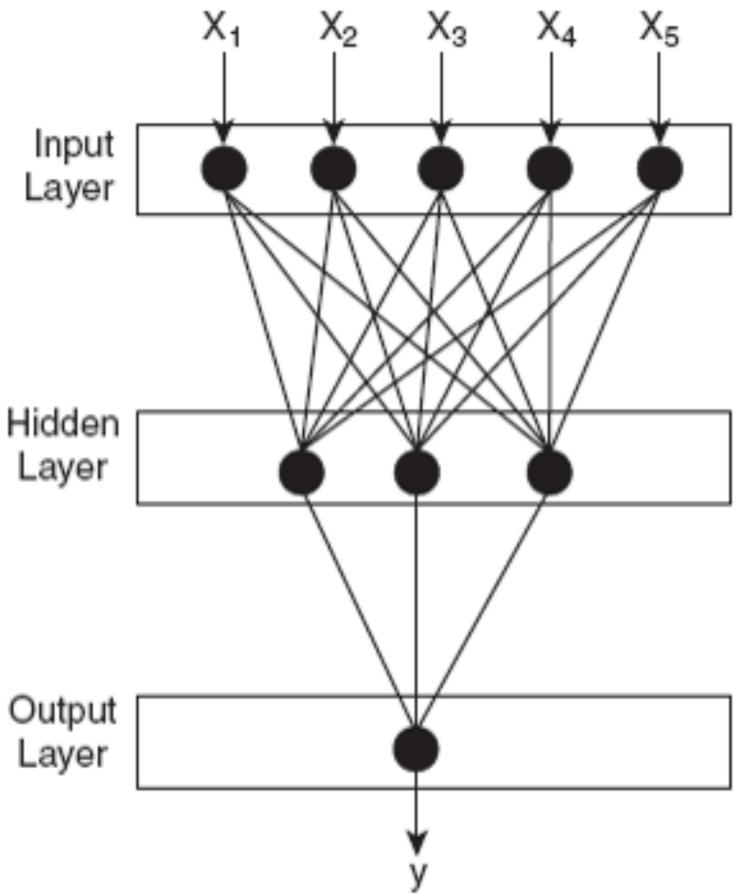
The XOR function, represented in a two dimensional space $[0, 1]^2$, is not linearly separable:

X_1	X_2	y
0	0	−1
1	0	1
0	1	1
1	1	−1



Combination of Several Neurons

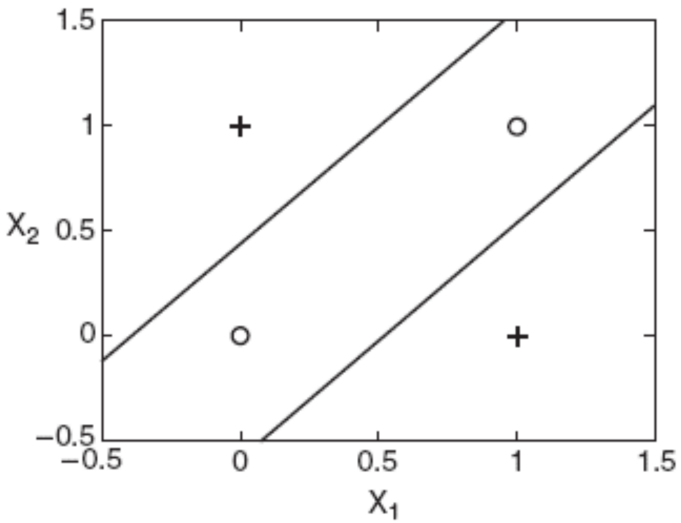
Non-linearly separable classes require a hidden layer:

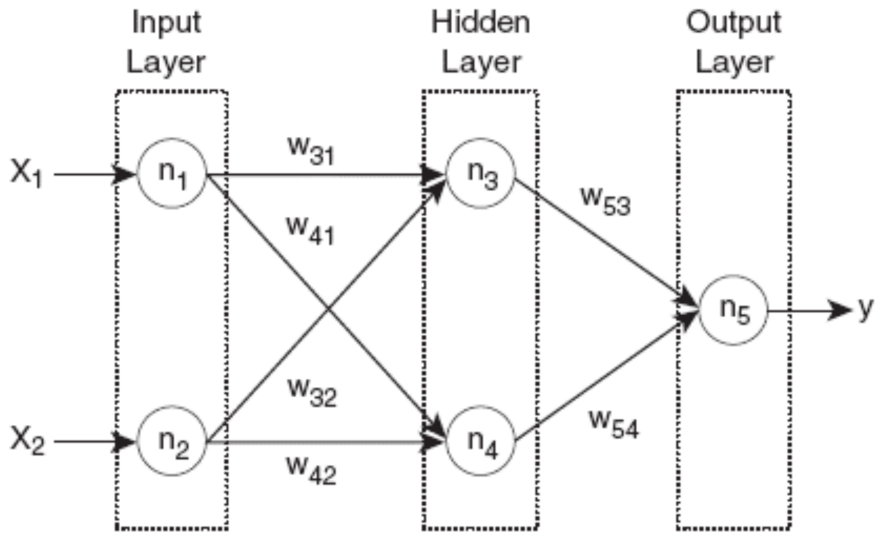


Architecture variants: feed-forward or recurrent (allows also connections within the same layer or to previous layers)

Solution for XOR

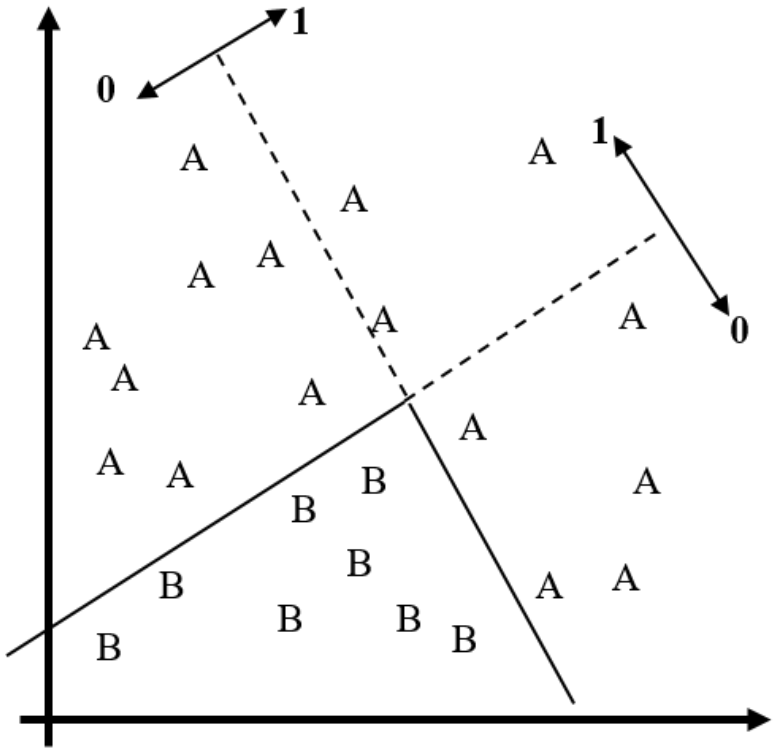
A possible neural net modelling the XOR function:

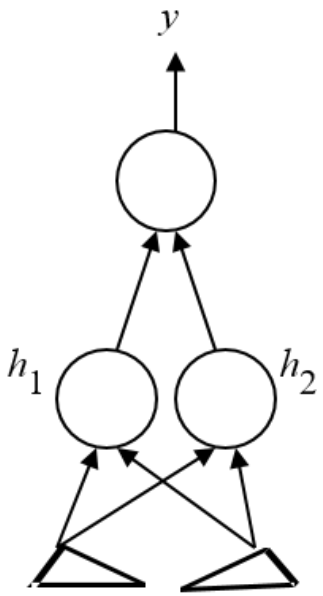




Geometrical Interpretation of the Neural Net Model

A hidden layer models the combination of several hyperplanes:





$h_1 = 0 \wedge h_2 = 0 : y = 0$ (classB)

otherwise: $y = 1$ (classA)

Learning Algorithm for Complex Networks

As long as the predicted class is not the given class on the examples: adaptation of the weights for several nodes.

- gradient descent (optimization approach) to minimize the overall error (backpropagation algorithm)
- Overall error is the sum of quadratic deviations of the output (vector) y from the desired output for the training data (least squares optimization).
- Required: output y is a *continuous and differentiable* function of the activation a (typically a loss function is used).

Deep neural nets

- We can trivially extend the number of hidden layers.
- No agreement on how many layers make a neural net *deep*. Just take it as one with *many* layers, whatever many means.
- Nowadays 100-layer nets are deep, but not very deep.

Formally, a neural net with two hidden layers reads

$$\begin{aligned}\mathbf{f}_1 &= \mathbf{W}_1^T \mathbf{x},\\ \mathbf{h}_1 &= \sigma(\mathbf{f}_1),\\ \mathbf{f}_2 &= \mathbf{W}_2^T \mathbf{h}_1,\\ \mathbf{h}_2 &= \sigma(\mathbf{f}_2),\\ \hat{y} &= \mathbf{w}_3^T \mathbf{h}_2.\end{aligned}$$

How does the computational graph now look like?

Learning with neural nets

Remember Mitchell's definition of learning. Maximize performance on experience

$$\underset{\mathbf{W}}{\operatorname{argmin}} \underbrace{\frac{1}{2}(y-\hat{y})^2}_{J(\mathbf{W})}.$$

Here,

- **Experience:** y.
- **Model:** $\hat{y}$.
- **Parameters:** $\mathbf{W}$ (collection of all weights in the net).
- **Performance:** J.

Learn as in linear regression: Find the point with minimum gradient

$\nabla_{\mathbf{W}} J \triangleq 0$ cannot be solved (i.e. no closed-form solution). Instead, start from a random point and take steps towards the gradient

$$\mathbf{W}^{(t+1)} \leftarrow \mathbf{W}^{(t)} - \alpha \nabla_{\mathbf{W}} J.$$

- This technique is called *gradient descent*.
- The step size α is called the *learning rate*.
- Each step (t) is called an *iteration*.

Learning for neural nets

$$\nabla_{\mathbf{w}} J = (y - \hat{y}) \nabla_{\mathbf{w}} \hat{y}$$

Learning for neural nets

$$\nabla_{\mathbf{w}} J = (y - \hat{y}) \nabla_{\mathbf{w}} \hat{y}$$

$\hat{y}$ is the predicted output of the model for a given input. The prediction can be computed by passing an input $\mathbf{x}$ through all the layers up to the output. This is called a *forward pass*.

Learning for neural nets

$$\nabla_{\mathbf{w}} J = (y - \hat{y}) \nabla_{\mathbf{w}} \hat{y}$$

$(y - \hat{y})$ is the prediction *error* of the model with the current parameter values.

Learning for neural nets

$$\nabla_{\mathbf{w}} J = (y - \hat{y}) \nabla_{\mathbf{w}} \hat{y}$$

$\nabla_{\mathbf{w}} \hat{y}$ is the gradient of the model wrt its parameters. Thanks to the chain rule, not as hard as it looks.

Chain rule: Given $y = f(u)$ and $u = g(x)$,

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u} \frac{\partial u}{\partial x}.$$

Remember the computational graphs!

Chain rule for vector-variate functions

Given $y = f(\mathbf{u})$ and $\mathbf{u} = g(\mathbf{x})$ where $\mathbf{u}$ and $\mathbf{x}$ are M and N dimensional vectors, respectively,

$$\frac{\partial y}{\partial x_i} = \sum_{j=1}^{M} \frac{\partial y}{\partial u_j} \frac{\partial u_j}{\partial x_i} = \frac{\partial y}{\partial \mathbf{u}}^T \frac{\partial \mathbf{u}}{\partial x_i}.$$

Applying this rule to all entries x_i of vector $\mathbf{x}$,

$$\frac{\partial y}{\partial \mathbf{x}} = \left[\frac{\partial y}{\partial x_1}, \cdots, \frac{\partial y}{\partial x_N} \right] = \left[\frac{\partial y}{\partial \mathbf{u}}^T \frac{\partial \mathbf{u}}{\partial x_1}, \cdots, \frac{\partial y}{\partial \mathbf{u}}^T \frac{\partial \mathbf{u}}{\partial x_N} \right]$$

$$= \frac{\partial y}{\partial \mathbf{u}}^T \left[\frac{\partial \mathbf{u}}{\partial x_1}, \cdots, \frac{\partial \mathbf{u}}{\partial x_N} \right] = \frac{\partial y}{\partial \mathbf{u}}^T \frac{\partial \mathbf{u}}{\partial \mathbf{x}},$$

where $\frac{\partial \mathbf{u}}{\partial \mathbf{x}}$ is the *Jacobian matrix*, which has the derivative $\frac{\partial u_i}{\partial x_j}$ on its (i,j)th element.

Updating Layer 4

$$\mathbf{f}_1 = \mathbf{W}_1^T \mathbf{x},$$
$$\mathbf{h}_1 = \sigma(\mathbf{f}_1),$$
$$\mathbf{f}_2 = \mathbf{W}_2^T \mathbf{h}_1,$$
$$\mathbf{h}_2 = \sigma(\mathbf{f}_2),$$
$$\mathbf{f}_3 = \mathbf{W}_3^T \mathbf{h}_2,$$
$$\mathbf{h}_3 = \sigma(\mathbf{f}_3),$$
$$\hat{y} = \mathbf{w}_4^T \mathbf{h}_3. \Rightarrow \text{Need to reach here}$$

The gradient wrt $\mathbf{w}_4$ reads

$$\nabla_{\mathbf{w}_4} \hat{y} = \nabla_{\mathbf{w}_4} \mathbf{w}_4^T \mathbf{h}_3 = \mathbf{h}_3.$$

Note that $\mathbf{h}_3$ needs to be stored during the forward pass!

Updating Layer 3

$$\mathbf{f}_1 = \mathbf{W}_1^T \mathbf{x},$$
$$\mathbf{h}_1 = \sigma(\mathbf{f}_1),$$
$$\mathbf{f}_2 = \mathbf{W}_2^T \mathbf{h}_1,$$
$$\mathbf{h}_2 = \sigma(\mathbf{f}_2),$$
$$\mathbf{f}_3 = \mathbf{W}_3^T \mathbf{h}_2, \Rightarrow \text{Need to reach here}$$
$$\mathbf{h}_3 = \sigma(\mathbf{f}_3),$$
$$\hat{y} = \mathbf{w}_4^T \mathbf{h}_3.$$

The gradient wrt $\mathbf{w}_r^3$, weights connecting Layer 2 neuron r to Layer 3 reads

$$\nabla_{\mathbf{w}_r^3} \hat{y} = \frac{\partial \hat{y}}{\partial \mathbf{h}_3}^T \frac{\partial \mathbf{h}_3}{\partial \mathbf{f}_3} \frac{\partial \mathbf{f}_3}{\partial \mathbf{w}_r^3}.$$

Updating Layer 2

$$\mathbf{f}_1 = \mathbf{W}_1^T \mathbf{x},$$
$$\mathbf{h}_1 = \sigma(\mathbf{f}_1),$$
$$\mathbf{f}_2 = \mathbf{W}_2^T \mathbf{h}_1, \Rightarrow \text{Need to reach here}$$
$$\mathbf{h}_2 = \sigma(\mathbf{f}_2),$$
$$\mathbf{f}_3 = \mathbf{W}_3^T \mathbf{h}_2,$$
$$\mathbf{h}_3 = \sigma(\mathbf{f}_3),$$
$$\hat{y} = \mathbf{w}_4^T \mathbf{h}_3.$$

The gradient wrt $\mathbf{w}_r^2$, weights connecting Layer 1 neuron r to Layer 2 reads

$$\nabla_{\mathbf{w}_r^2} \hat{y} = \frac{\partial \hat{y}}{\partial \mathbf{h}_3}^T \frac{\partial \mathbf{h}_3}{\partial \mathbf{f}_3} \frac{\partial \mathbf{f}_3}{\partial \mathbf{h}_2} \frac{\partial \mathbf{h}_2}{\partial \mathbf{f}_2} \frac{\partial \mathbf{f}_2}{\partial \mathbf{w}_r^2}.$$

Note how the factors in red can be reused from Layer 3!

Updating Layer 2

$$\mathbf{f}_1 = \mathbf{W}_1^T \mathbf{x},$$
$$\mathbf{h}_1 = \sigma(\mathbf{f}_1),$$
$$\mathbf{f}_2 = \mathbf{W}_2^T \mathbf{h}_1, \Rightarrow \text{Need to reach here}$$
$$\mathbf{h}_2 = \sigma(\mathbf{f}_2),$$
$$\mathbf{f}_3 = \mathbf{W}_3^T \mathbf{h}_2,$$
$$\mathbf{h}_3 = \sigma(\mathbf{f}_3),$$
$$\hat{y} = \mathbf{w}_4^T \mathbf{h}_3.$$

The gradient wrt $\mathbf{w}_r^2$, weights connecting Layer 1 neuron r to Layer 2 reads

$$\nabla_{\mathbf{w}_r^2} \hat{y} = \frac{\partial \hat{y}}{\partial \mathbf{h}_3}^T \frac{\partial \mathbf{h}_3}{\partial \mathbf{f}_3} \frac{\partial \mathbf{f}_3}{\partial \mathbf{h}_2} \frac{\partial \mathbf{h}_2}{\partial \mathbf{f}_2} \frac{\partial \mathbf{f}_2}{\partial \mathbf{w}_r^2}.$$

Updating Layer 1

$$\begin{aligned}
\mathbf{f}_1 &= \mathbf{W}_1^T \mathbf{x}, \Rightarrow \text{Need to reach here} \\
\mathbf{h}_1 &= \sigma(\mathbf{f}_1), \\
\mathbf{f}_2 &= \mathbf{W}_2^T \mathbf{h}_1, \\
\mathbf{h}_2 &= \sigma(\mathbf{f}_2), \\
\mathbf{f}_3 &= \mathbf{W}_3^T \mathbf{h}_2, \\
\mathbf{h}_3 &= \sigma(\mathbf{f}_3), \\
\hat{y} &= \mathbf{w}_4^T \mathbf{h}_3.
\end{aligned}$$

The gradient wrt $\mathbf{w}_r^1$, weights connecting input neuron r to Layer 1 reads

$$\nabla_{\mathbf{w}_r^1} \hat{y} = \frac{\partial \hat{y}}{\partial \mathbf{h}_3}^T \frac{\partial \mathbf{h}_3}{\partial \mathbf{f}_3} \frac{\partial \mathbf{f}_3}{\partial \mathbf{h}_2} \frac{\partial \mathbf{h}_2}{\partial \mathbf{f}_2} \frac{\partial \mathbf{f}_2}{\partial \mathbf{h}_1} \frac{\partial \mathbf{h}_1}{\partial \mathbf{f}_1} \frac{\partial \mathbf{f}_1}{\partial \mathbf{w}_r^1}.$$

Note how the factors in red can be reused from Layer 2!

Error Backpropagation

Put everything together, learn the parameters of Layer l following the update rule below:

$$\mathbf{w}_r^{l\,(t+1)} \leftarrow \mathbf{w}_r^{l\,(t)} - \alpha \underbrace{(y-\hat{y})\nabla_{\mathbf{w}_r^l}\hat{y}}_{\nabla_{\mathbf{w}_r^l}\hat{y}}.$$

Looking closer, we basically update weights by rescaling the prediction error $(y-\hat{y})$ by the gradient of the model $\hat{y}$ wrt them. Hence, prediction error propagates from the top layer to bottom at different levels of importance. This is called **error backpropagation**.

Gradient Backpropagation

- The gradient at Layer $l+1$ contains a portion of factors required to calculate the gradient at Layer l.
- Then update from top to bottom. Store the reusable factors of each gradient before moving down. This is called the *backward pass*.

Other things than the gradient can backpropagate as well (e.g. random variables or their moments!).

The Backprop Algorithm

Given an input $\mathbf{x}$ and a model $\hat{y}$ with L hidden layers.

- Do a forward pass (i.e. compute $\hat{y}(\mathbf{x})$). Store activation maps on the way $\mathbf{h}_1, \cdots, \mathbf{h}_L$.
- Do a backward pass (i.e. compute gradients $\nabla_{\mathbf{w}_r^l} \hat{y}$ for all r and l). Store the reusable factors of the gradients on the way.
- Perform the parameter update.

Discussion

pros

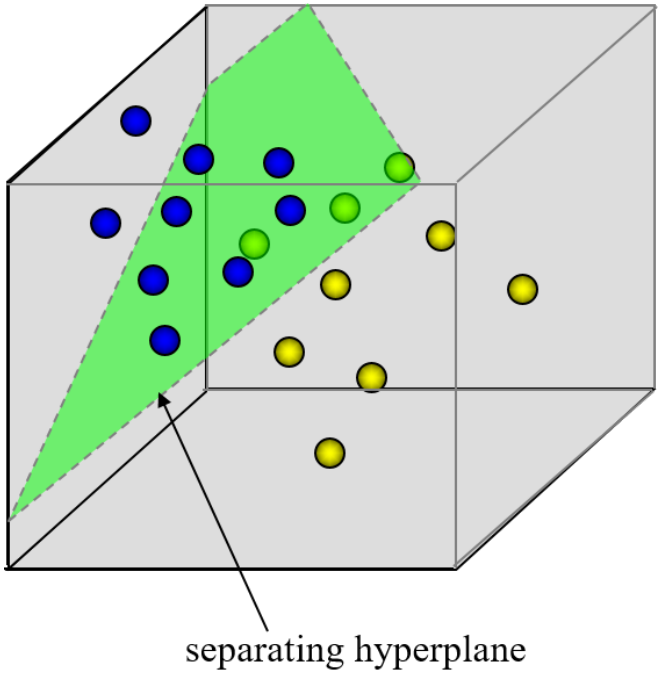
- typically good classification accuracy achievable, arbitrarily complex decision boundaries (choose a topology as simple as possible to avoid overfitting)
- redudant features are not problematic: weights will become small
- application is efficient

cons

- model (of more complex networks with hidden layers) is hardly interpretable: learns only weights, no class description
- inefficient learning (training can take a tremendous amount of time)
- susceptible to errors in training data
- possibly converging towards a *local* minimum

Linear separation

Two classes are linearly separable if all data points of two classes can be perfectly separated by a hyperplane.



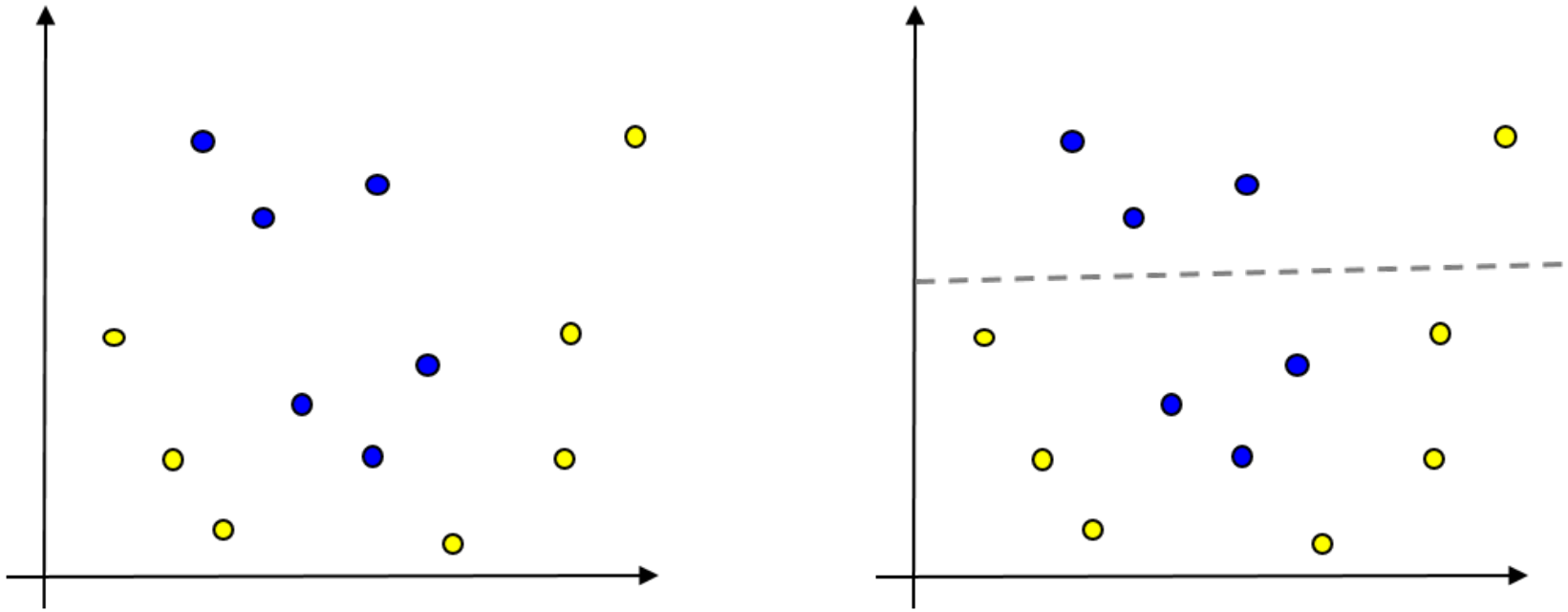
Two problems arise:

i) If there is no such hyperplane, what should be done? $\Rightarrow$ Transform feature space

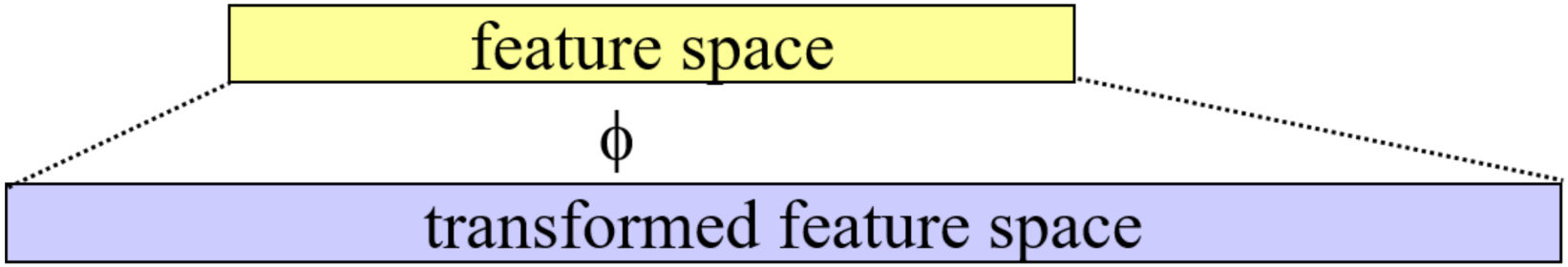
ii) If there are multiple separating hyperplanes, which one should we choose? $\Rightarrow$ Maximize margin

Problem 1: Transforming the feature space

Given data points $x_n \in \mathcal{X}$ that are not linearly separable in feature space $\mathcal{X}$ (left), apply a transformation $\phi : \mathcal{X} \rightarrow \mathcal{F}$ such that they are separable in the transformed space $\mathcal{F}$ (right).

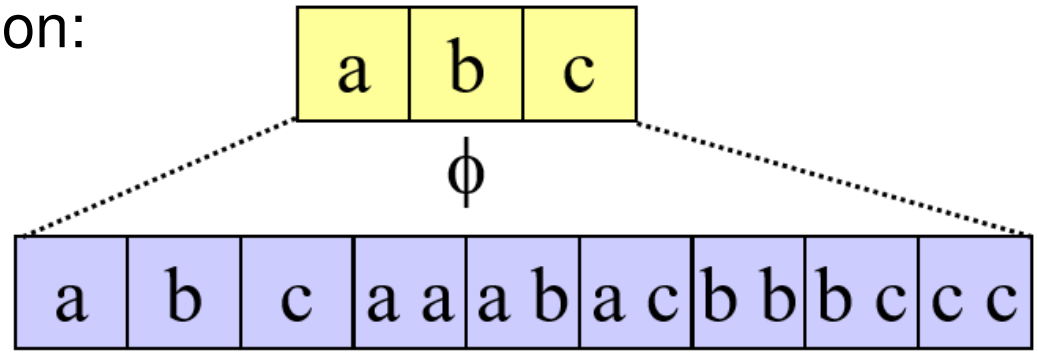


Feature transformation – Principle



Try to find a linear separation in the transformed feature space.

Example transformation:



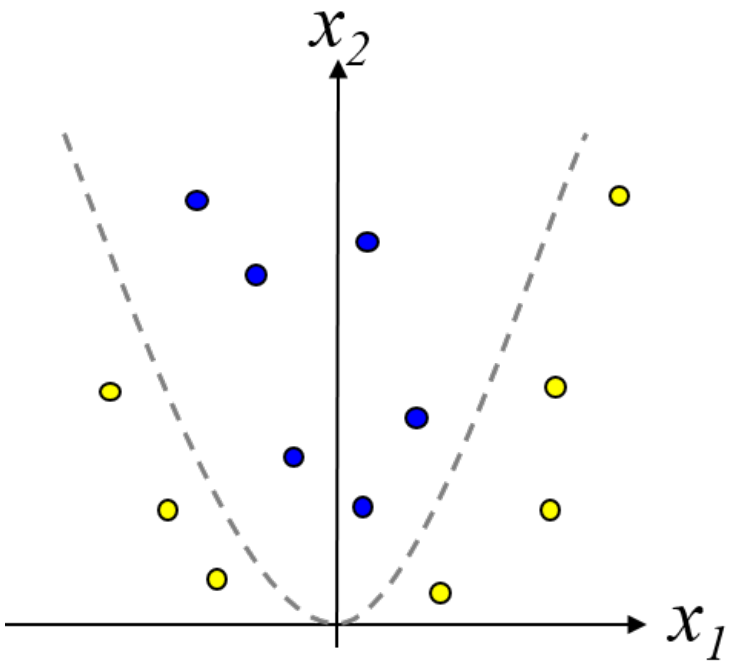
A hyperplane in this feature space corresponds to a polynomial (2nd degree) separation surface in the original feature space.

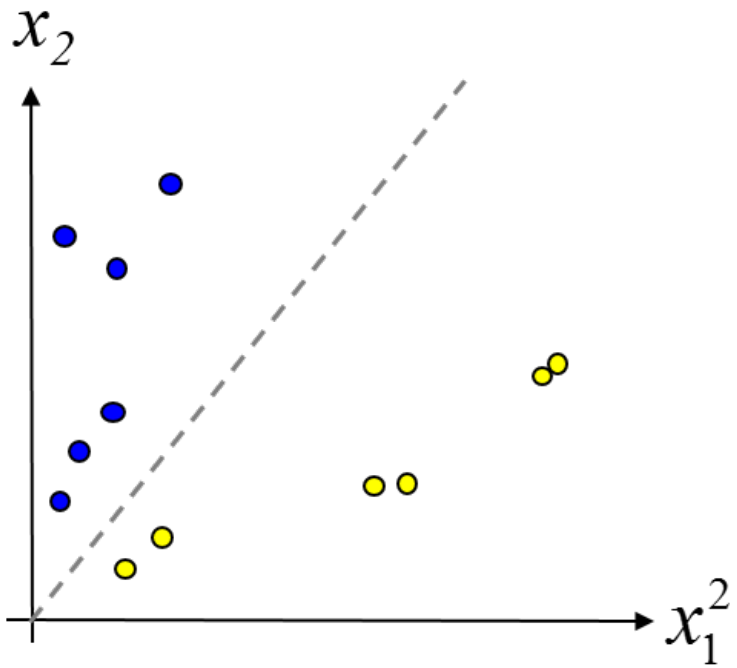
Feature transformation – Example

Feature space $x = (x_1, x_2) \in \mathbb{R}^2$

Transformed feature space e.g.,

$$\phi(x) = (x_1^2, x_2^2, \sqrt{2} \cdot x_1, \sqrt{2} \cdot x_2, \sqrt{2} \cdot x_1 \cdot x_2, 1) \in \mathbb{R}^6$$





Kernel trick

For the optimization, we need to eventually compute scalar products between points.

A special similarity function, a *kernel*, is used for an implicit feature transformation:

- Instead of computing new features for all objects: $\phi(x)$, and then compute scalar products in the new space,
- a kernel computes the inner product in the transformed space directly, without explicitly computing the transformed space first:

$$k(x_n, x_m) = \langle \phi(x_n), \phi(x_m) \rangle$$

- This is an interesting property for computationally expensive feature transformations, called the *kernel trick*.

Kernel

When is a function a kernel function?

Theorem 1 (Mercer's Theorem)

A function $k : \mathbb{R}^n \rightarrow \mathbb{R}_0^+$ is a kernel function, if the kernel-matrix (Gram matrix)

$$K = \begin{pmatrix} k(x_1, x_1) & \cdots & k(x_1, x_n) \\ \vdots & \ddots & \vdots \\ k(x_n, x_1) & \cdots & k(x_n, x_n) \end{pmatrix}$$

is positive semi-definite (i.e., the Gram matrix does not have any negative Eigenvalues).

Some necessary conditions for a kernel function

Necessary (but not sufficient) conditions for k being a kernel function are:

Symmetry:

$$\begin{aligned} k(x_n, x_m) &= \langle \phi(x_n), \phi(x_m) \rangle \\ &= \langle \phi(x_m), \phi(x_n) \rangle \\ &= k(x_m, x_n) \end{aligned}$$

Cauchy-Schwarz Inequality:

$$k(x_n, x_m)^2 \leq k(x_n, x_n) \cdot k(x_m, x_m)$$

Combinations of kernels

Given valid kernel functions k_1, k_2, a constant $a \in \mathbb{R}^+$, and a positive semi-definite matrix B, new valid kernel functions k can be derived following the rules below

- $k(x,y) = k_1(x,y) \cdot k_2(x,y)$
- $k(x,y) = k_1(x,y) + k_2(x,y)$
- $k(x,y) = a \cdot k_1(x,y)$
- $k(x,y) = x^{\mathsf{T}} \cdot B \cdot y$

Common kernel functions

Typical examples for kernel functions are:

linear $k(x, y) = \langle x, y \rangle$

polynomial $k(x, y) = (\langle x, y \rangle + c)^d$

radial basis function $k(x, y) = e^{-\gamma \cdot |x-y|^2}$

squared exponential $k(x, y) = e^{-\frac{||x-y||^2}{2\sigma^2}}$

Sigmoid $k(x, y) = \tanh(\gamma \cdot \langle x, y \rangle + c)$

Ridge regression

Let X be an $N \times D$ matrix with data points x_n^T in its rows and y be an $N-$dimensional vector of the corresponding labels y_n for the $n-$th entry. Apply the linear regression learning objective this time with an L^2-norm regularizer on the weights (called *ridge regression*):

$$L(w) = \sum_{n=1}^{N} \frac{1}{2}(y_n - w^T x_n)^2 + \lambda ||w||_2^2$$
$$= \frac{1}{2} y^T y + \frac{1}{2} w^T X^T X w - y^T X w + \lambda w^T w.$$

Minimize the loss with respect to the weights

$$\nabla_w L(w) = w^T X^T X - y^T X + \lambda w^T \triangleq 0$$
$$\Rightarrow w^T = -\frac{1}{\lambda}(w^T X^T - y^T) X.$$

Dual representation

Denoting $a \triangleq -\frac{1}{\lambda}(Xw - y)$ and the matrix that contains $\phi(\cdot)^T$ in its rows as Φ, we get

$$w^T = a^T \Phi.$$

Placing this expression into the loss function gives its so-called *dual representation* that contains data only as inner products on the transformed space:

$$\begin{aligned} L(w) &= \frac{1}{2} y^T y + \frac{1}{2} w^T \Phi^T \Phi w - y^T \Phi w + \lambda w^T w \\ &= \frac{1}{2} y^T y + \frac{1}{2} a^T \Phi \Phi^T \Phi \Phi^T a - y^T \Phi \Phi^T a + \lambda a^T \Phi \Phi^T a. \end{aligned}$$

Note that $K = \Phi \Phi^T$ is the $N \times N$ dimensional *Gram* matrix with $K_{ij} = \phi(x_i)^T \phi(x_j)$.

Kernel Linear Regression

The inner product $\phi(x_i)^T\phi(x_j)$ can be replaced with any kernel, hence we can have $K_{ij} = k(x_i, x_j)$. The kernelized version of the linear regression then reads

$$L(a) = \frac{1}{2}y^T y + \frac{1}{2}a^T KKa - y^T Ka + \lambda a^T Ka.$$

Setting the gradient of L with respect to a, evaluating at zero and solving for a gives

$$\begin{aligned}
\nabla_a L(a) &\triangleq 0 = KKa - y^T K + \lambda Ka \\
\Rightarrow a &= (KK + \lambda K)^{-1} Ky \\
&= (K(K + \lambda I))^{-1} Ky \\
&= (K + \lambda I)^{-1} K^{-1} Ky \\
&= (K + \lambda I)^{-1} y.
\end{aligned}$$

The prediction function

We predict the label y_* of a query input x_* by

$$\begin{aligned} f(x_*) &= \phi(x_*)\Phi^T a \\ &= k_*^T (K + \lambda I)^{-1} y, \end{aligned}$$

where

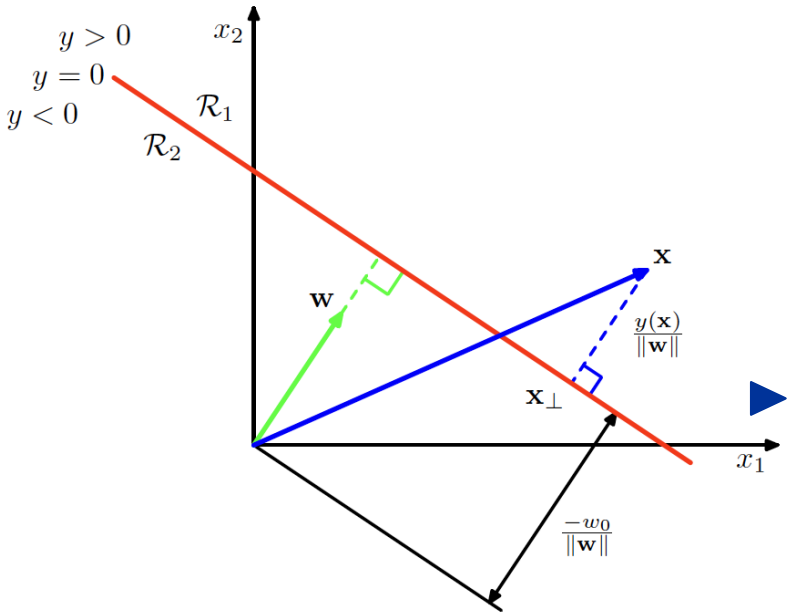
$$k_* = \begin{bmatrix} k(x_*, x_1) \\ k(x_*, x_2) \\ \vdots, \\ k(x_*, x_n) \end{bmatrix}.$$

The discriminant function

The linear function below

$$f(x) = w^T x + b$$

assigns x to class $+1$ if $f(x) \geq 0$ and to class -1 if $f(x) < 0$



- Take two points x_A and x_B on the decision boundary. i.e. $f(x_A) = f(x_B) = 0$. Then $w^T(x_A - x_B) = 0$, hence w is orthogonal to the decision boundary.
- Normal distance from the origin to the decision boundary is $\frac{w^T x}{||w||} = \frac{-b}{||w||}$

Figure: Chris Bishop, *Pattern Recognition and Machine Learning*, Springer, 2006

Margin: Distance to the decision boundary

- The value of $f(x)$ is proportional to the signed perpendicular distance r of x to the decision boundary, called the *margin*. This is the case because for

$$x = x_\perp + r\frac{w}{||w||}$$

we have that

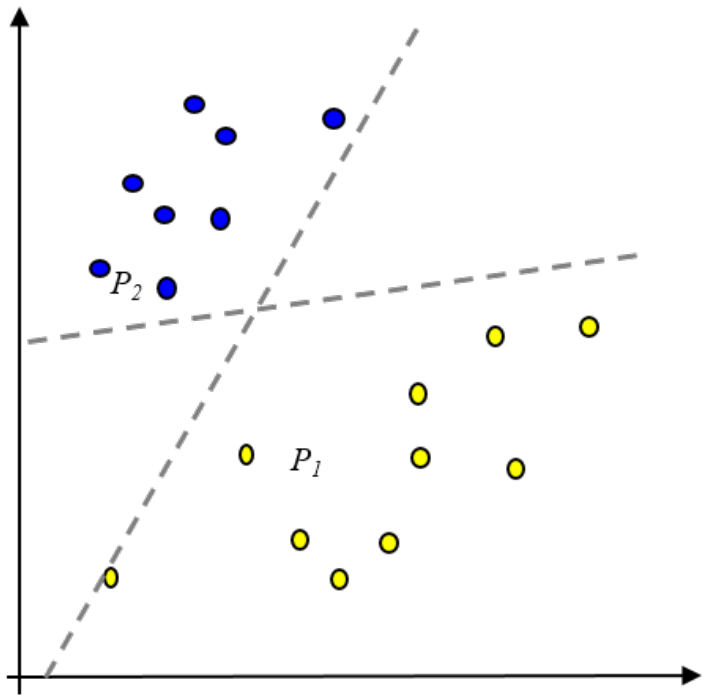
$$\underbrace{w^T x + b}_{f(x)} = \underbrace{w^T x_\perp + b}_{f(x_\perp)=0} + r \underbrace{\frac{w^T w}{||w||}}_{||w||^2/||w||} .$$

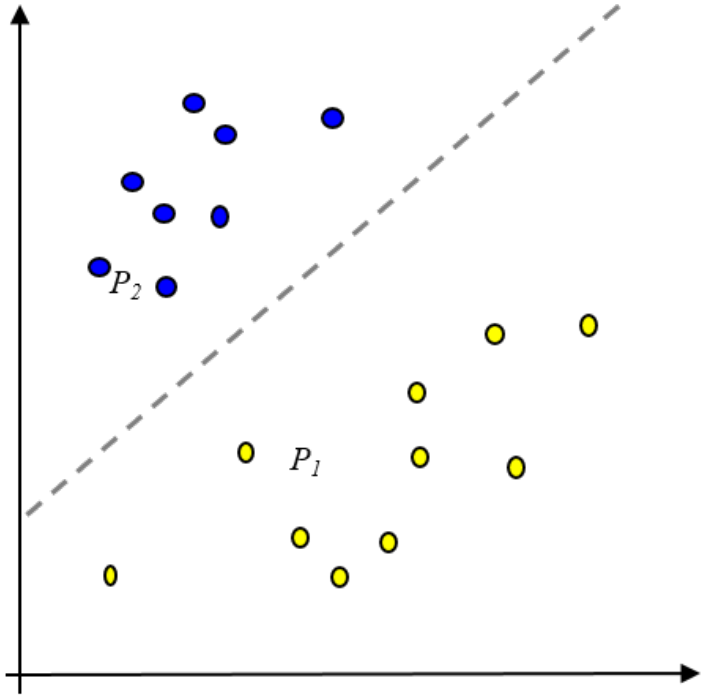
Therefore the margin for x is given as

$$r = \frac{f(x)}{||w||}.$$

Problem 2: Which separating hyperplane to choose?

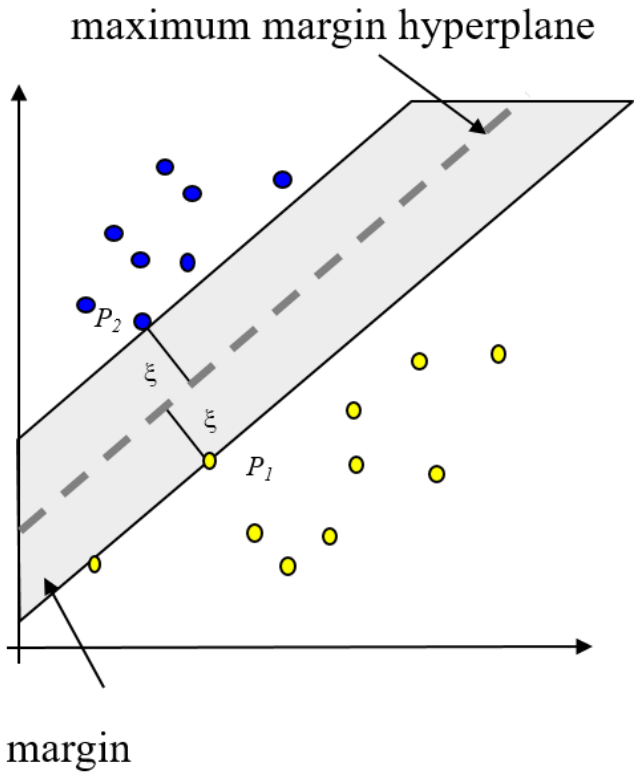
We are interested in the one that minimizes the risk of misclassification at test time.





Solution: Choose the hyperplane with maximum margin

The hyperplane with minimum expected risk has maximum distance to the closes points of both classes. This hyperplane, named as the *maximum margin hyperplane*, has the following favorable properties



- maximal distance to objects of both classes (minimum ξ)
- minimal probability that hyperplane has to be adapted when new training examples are added
- best generalization
- determined only by the closest data points to the hyperplane, called the *support vectors*.

Finding the maximum margin hyperplane

For all input-output pairs (x_n, y_n), our goal is to find an f that provides $y_n f(x_n) > 0$ with the largest margin. Remember that the distance of a data point to the decision boundary is

$$\frac{y_n f(x_n)}{||w||} = \frac{y_n(w^T \phi(x_n) + b)}{||w||}.$$

Hence, we want to solve the optimization problem below

$$\text{argmax}_{w,b} \left\{ \frac{1}{||w||} \min_n \left[y_n(w^T \phi(x_n) + b) \right] \right\}.$$

We need to first re-express this hard combinatorial problem so that it is much easier to solve.

Hard-margin support vector machine

Notice that the distance to the decision boundary is invariant to weight scaling

$$\frac{y_n(w^T\phi(x_n)+b)}{\sqrt{w^Tw}} = \frac{y_n(\kappa w^T\phi(x_n)+\kappa b)}{\sqrt{(\kappa w)^T(\kappa w)}}.$$

for $\kappa \in \mathbb{R}$. Since κ is arbitrary, set $y_n(w^T\phi(x_n)+b)=1$ for the point closest to the decision boundary, called its *canonical representation*. We can re-express the problem as:

$$\begin{aligned} &\underset{w,b}{\text{argmin}} \quad \frac{1}{2}||w||^2 \\ &s.t. \quad y_n(w^T\phi(x_n)+b) \geq 1, \qquad n=1,\ldots,N, \end{aligned}$$

which is called the *Hard-Margin Support Vector Machine (SVM)* as it assumes perfect separation of all training points.

Switch to the dual view to kernelize

Introduce *Lagrange multipliers* $a = [a_1, \ldots, a_N]$ to account for the constraints, which gives the *Lagrangian loss*

$$L(w, b, a) = \frac{1}{2}||w||^2 - \sum_{n=1}^{N} a_n \Big\{ y_n(w^T \phi(x_n) + b) - 1 \Big\}.$$

We are interested in

$$\underset{w,b}{\operatorname{argmin}} \ \underset{a}{\operatorname{argmax}} \quad L(w, b, a) \equiv \underset{a}{\operatorname{argmax}} \ \underset{w,b}{\operatorname{argmin}} \ L(w, b, a).$$

An equivalent representation to this problem is

$$\underset{w,b}{\operatorname{argmin}} \ E_\infty(y_n f(x_n) - 1) + \lambda||w||^2$$

where

$$E_\infty(z) = \begin{cases} 0, & z \geq 0, \\ \infty, & z < 0. \end{cases}$$

Calculating the dual of the Lagrangian loss

Setting $\nabla_{w,b} L \triangleq 0$, we get

$$w = \sum_{n=1}^{N} a_n y_n \phi(x_n), \qquad 0 = \sum_{n=1}^{N} a_n y_n.$$

Eliminating w and b gives objective

$$L(w,b,a) = \frac{1}{2} w^T w - w^T \underbrace{\sum_{n=1}^{N} a_n y_n \phi(x_n)}_{=w} - b \underbrace{\sum_{n=1}^{N} a_n y_n}_{=0} + \sum_{n=1}^{N} a_n$$

$$= \sum_{n=1}^{N} a_n - \frac{1}{2} w^T w$$

$$= \sum_{n=1}^{N} a_n - \sum_{n=1}^{N} \sum_{m=1}^{N} a_n a_m y_n y_m \phi(x_n)^T \phi(x_m).$$

The dual of the hard-margin SVM

$$\underset{a}{\text{argmax}} \sum_{n=1}^{N} a_n - \sum_{n=1}^{N}\sum_{m=1}^{N} a_n a_m y_n y_m k(x_n, x_m)$$

$$\text{s.t.} \qquad a_n \geq 0, \qquad 1, \ldots, N$$

$$\sum_{n=1}^{N} a_n y_n = 0,$$

where the first constraint comes from the Lagrangian assumption and the second from the solution of b.

The prediction function of the hard-margin SVM

The class label y_* of a test query x_* is then predicted as

$$
\begin{aligned}
f(x_*) &= w^T \phi(x_*) + b \\
&= \phi(x_*)^T \sum_{n=1}^{N} a_n y_n \phi(x_n) + b \\
&= \sum_{n=1}^{N} a_n y_n \phi(x_*)^T \phi(x_n) + b \\
&= \sum_{n=1}^{N} a_n y_n k(x_*, x_n) + b
\end{aligned}
$$

Support vectors

It is possible to derive the following three constraints from the optimization problem of the hard-margin SVM, which are called the *Karush-Kuhn-Tucker (KKT)* conditions

$$a_n \geq 0, \quad y_n f(x_n) - 1 \geq 0, \quad a_n \{y_n f(x_n) - 1\} = 0.$$

Due to the right-most condition, for every data point at least one of the below cases is true

- $a_n = 0 \Rightarrow$ Point does not contribute to prediction
- $y_n f(x_n) = 1 \Rightarrow$ Point is on the margin, called a *support vector*.

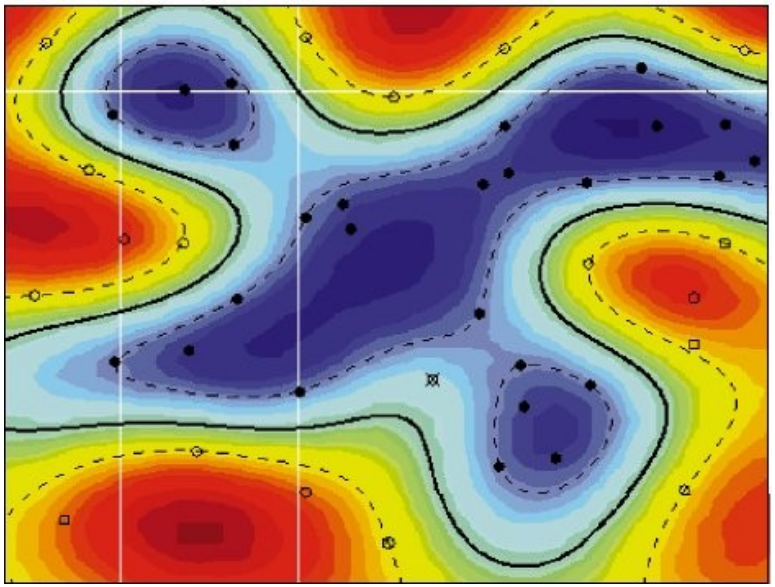
Hence we can rewrite the prediction function as

$$f(x_*) = \sum_{x_n \in \mathcal{S}} a_n y_n k(x_*, x_n) + b,$$

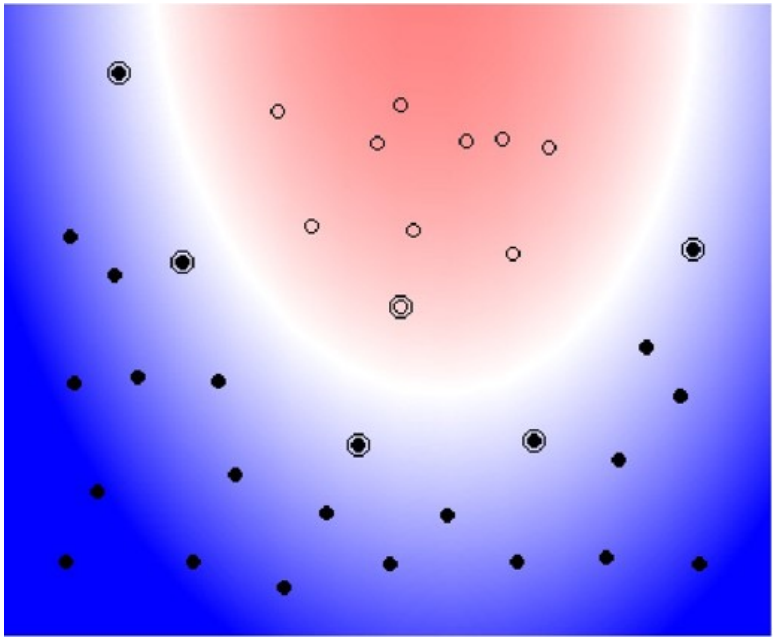
where $\mathcal{S} = \{x_n | a_n > 0\}$ is the set of support vectors.

Decision boundaries of the kernelized hard-margin SVM

Radial basis function

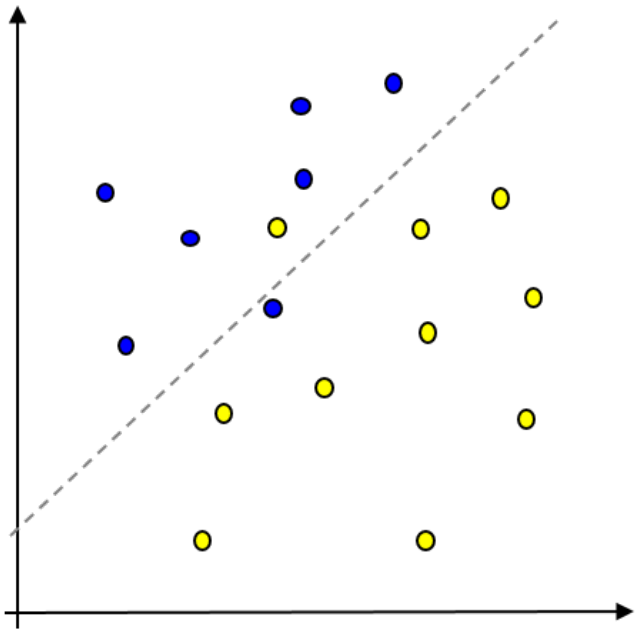


Polynomial (2nd degree)

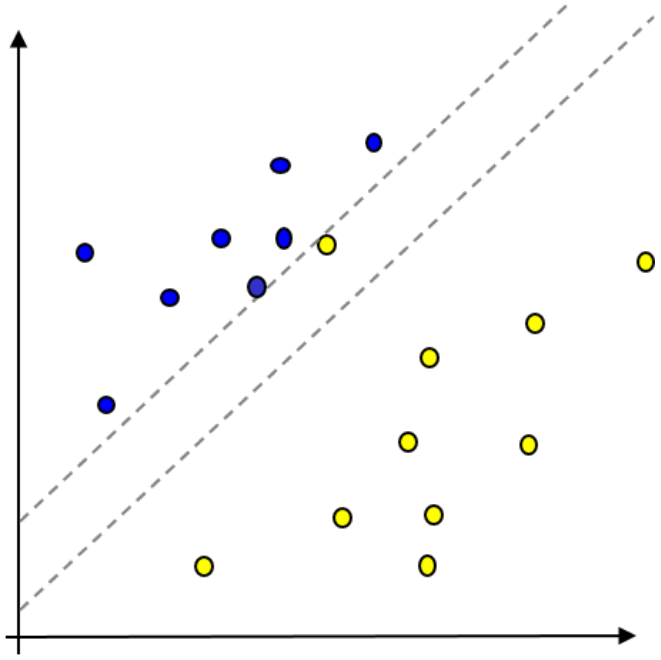


What if a hard margin is not possible?

If the data are not linearly separable



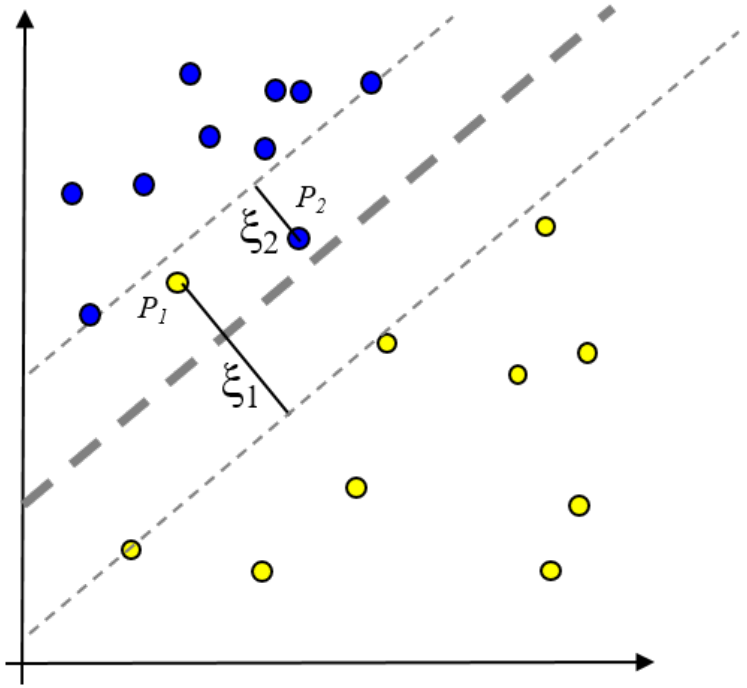
or the separation is not very stable



we can optimize with a trade-off between training error and size of the margin.

Then *soften* the margin

Consider training error during optimization:



- ξ_i: distance of p_i from the border of the margin (slack variable)
- regularization of the influence of each individual observation

Slack variables

Introduce variables $\xi_n \geq 0$ that allow violation of the margin but with a penalty linear to the distance from the margin. These variables, called the *slack variables*[1], satisfy $\xi_n = |y_n - f(x_n)|$. We apply the following constraint on each training data point

$$y_n f(x_n) \geq 1 - \xi_n.$$

The slack variables have the following properties.

- $\xi_n = 0$ when data point is inside the correct region
- $0 < \xi_n < 1$ when data point is correctly classified but in the margin
- $\xi_n = 1$ when data point is on the decision boundary
- $\xi_n > 1$ when data point is incorrectly classified.

[1] Bennett, 1992; Cortes, Vapnik, 1995

The soft-margin SVM

After including the slack variables, the optimization problem for the soft-margin SVM reads

$$\underset{w,b,\xi_1,\ldots,\xi_N}{\operatorname{argmin}} \quad C\sum_{n=1}^{N}\xi_n + \frac{1}{2}||w||^2$$
$$\text{s.t.} \quad y_n f(x_n) \geq 1-\xi_n, \qquad n=1,\ldots,N$$

The value $\sum_{n=1}^{N}\xi_n$ is an upper bound on the number of mis-classified data points. The positive scalar C is a *regularizer* that controls how much the algorithm should sacrifice from perfect separation. We recover hard-margin SVM as $C \to +\infty$.

The Lagrangian of the soft-margin SVM

As the first step towards kernelization, move again to the dual view. First write down the Lagrangian, where $a_n \geq 0$ and $\mu_n \geq 0$ are Lagrange multipliers:

$$L(w,b,a)$$
$$= \frac{1}{2}||w||^2 + C\sum_{n=1}^{N}\xi_n - \sum_{n=1}^{N}a_n\{y_n f(x_n) - 1 + \xi_n\} - \sum_{n=1}^{N}\mu_n\xi_n.$$

The dual of the soft-margin SVM

We set the derivatives of the Lagrangian to zero

$$\frac{\partial L}{\partial w} = 0 \Rightarrow w = \sum_{n=1}^{N} a_n y_n \phi(x_n)$$

$$\frac{\partial L}{\partial b} = 0 \Rightarrow \sum_{n=1}^{N} a_n y_n = 0$$

$$\frac{\partial L}{\partial \xi_n} = 0 \Rightarrow a_n = C - \mu_n$$

and eliminate the primal variables w and b. We also absorb the Lagrange multipliers μ_n of the slack variables into

$$a_n = C - \mu_n \Rightarrow 0 \leq a_n \leq C$$

exploiting the fact that they are positive. These constraints are known as the *box constraints*.

The soft-margin SVM

We arrive at our target model, the kernelized soft-margin SVM, which can be trained by solving

$$\underset{a_1,\cdots,a_N}{\operatorname{argmax}} \sum_{n=1}^{N} a_n - \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N} a_n a_m y_n y_m k(x_n, x_m)$$

$$\text{s.t.} \quad 0 \leq a_n \leq C, \qquad n = 1, \ldots, N,$$

$$\sum_{n=1}^{N} a_n y_n = 0.$$

This is a *quadratic program:* quadratic loss with linear constraints. Global optimum exists but no analytical solution available. Most libraries implement SVMs with a popular iterative algorithm known as *Sequential Minimal Optimization (SMO)* (Platt, 1999).

Support vectors of the soft-margin SVM

The KKT conditions appear in the following way:

$$\left.\begin{aligned} a_n &\geq 0, \\ y_n f(x_n) - 1 + \xi_n &\geq 0, \\ a_n(y_n f(x_n) - 1 + \xi_n) &= 0, \end{aligned}\right\} \text{ for } a_n$$

$$\left.\begin{aligned} \mu_n &\geq 0, \\ \xi_n &\geq 0, \\ \mu_n \xi_n &= 0. \end{aligned}\right\} \text{ for } \mu_n$$

Consequences:

- For some data points $a_n = 0$ and they will not play role in prediction. For others $y_n f(x_n) = 1 - \xi_n$ and $a_n > 0$, which are the *support vectors*.
- If $a_n < C$, then $\mu_n > 0$, which would require $\xi_n = 0$.
- If $a_n = C$, then correctly classified if $\xi_n \leq 1$ and misclassified if $\xi > 1$.

Calculating the bias term

It is required to solve b for making predictions. Define $\mathcal{M} = \{x_n | 0 < a_n < C\}$ for which we have $\xi_n = 0$, hence for any $x_n \in \mathcal{M}$

$$y_n f(x_n) = 1 \Rightarrow y_n \left(\sum_{x_m \in \mathcal{S}} a_m y_m k(x_n, x_m) + b \right) = 1.$$

Multiplying both sides by y_n and using $y_n^2 = 1$, we get $b = y_n - \sum_{x_m \in \mathcal{S}} a_m y_m k(x_n, x_m)$. For numerical stability, it is advised to use the average

$$b = \frac{1}{|\mathcal{M}|} \sum_{x_n \in \mathcal{M}} \left\{ y_n - \sum_{x_m \in \mathcal{S}} a_m y_m k(x_n, x_m) \right\}.$$

The prediction function

Once the optimal values of a_n's are found at training time, the below function is used for predictions

$$f(x_*) = \sum_{x_n \in \mathcal{S}} a_n y_n k(x_*, x_n) + \frac{1}{|\mathcal{M}|} \sum_{x_n \in \mathcal{M}} \left\{ y_n - \sum_{x_m \in \mathcal{S}} a_m y_m k(x_n, x_m) \right\}.$$

Discussion

pros

- can generate classifiers of high accuracy
- can be formulated as a convex optimization problem, thus efficient optimization algorithms can find the *global* optimum
- relatively low tendency to overfit (based on generalization theory) – but one should prefer simpler kernels
- efficient application of the learned model
- compact models

cons

- often extensive training time
- non-trivial implementation (note that we did not go into details of the optimization algorithms)
- derived classification models are hard to interpret