

CSE 417T

# Introduction to Machine Learning

Lecture 19

Instructor: Chien-Ju (CJ) Ho

# Logistics

- Homework 5 is due Apr 19 (Tuesday)
- Exam 2 will be on April 28 (Thursday)
  - Will focus on the topics in the second half of the semester
  - Format / logistics will be similar to Exam 1
  - More details to come

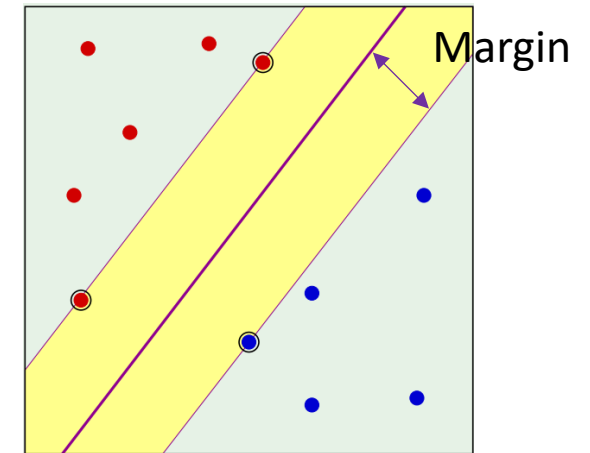
Recap

# Support Vector Machines

- Goal: Find the **max-margin** linear separator
- If the data is linearly separable
  - **Hard-Margin SVM** (Assume data is linearly separable)

$$\begin{array}{ll} \text{minimize}_{\vec{w}, b} & \frac{1}{2} \vec{w}^T \vec{w} \\ \text{subject to} & y_n (\vec{w}^T \vec{x}_n + b) \geq 1, \forall n \end{array}$$

- $g(\vec{x}) = \text{sign}(\vec{w}^{*T} \vec{x} + b^*)$

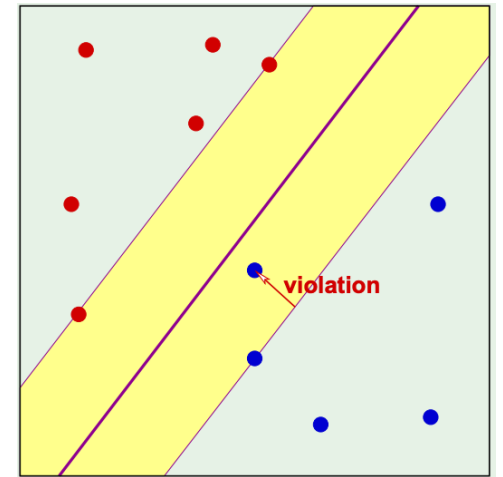


- If the data is not linearly separable
  - **Soft-margin SVM**
  - Nonlinear transformation – **Dual Formulation** and **Kernel Tricks**

# Soft-Margin SVM

- For each point  $(\vec{x}_n, y_n)$ , we allow some violation  $\xi_n \geq 0$ 
  - The constraint becomes:  $y_n(\vec{w}^T \vec{x}_n + b) \geq 1 - \xi_n$
  - We add a penalty for each violation: Total penalty  $C \sum_{n=1}^N \xi_n$

$$\begin{aligned} & \text{minimize}_{\vec{w}, b, \xi} \quad \frac{1}{2} \vec{w}^T \vec{w} + C \sum_{n=1}^N \xi_n \\ & \text{subject to} \quad y_n(\vec{w}^T \vec{x}_n + b) \geq 1 - \xi_n, \forall n \\ & \quad \quad \quad \xi_n \geq 0, \forall n \end{aligned}$$



Remarks:

- $C$  is a hyper-parameter we can choose, e.g., using validation
  - Larger  $C \Rightarrow$  less tolerable to noise  $\Rightarrow$  smaller margin
- Soft-margin SVM is still a Quadratic Program, with efficient solvers
- $\xi_n^*$  indicates where  $\vec{x}_n$  is with respect to the separator and the margin

# Primal-Dual Formulations of Hard-Margin SVM

- Primal

$$\begin{aligned} & \text{minimize}_{\vec{w}, b} \quad \frac{1}{2} \vec{w}^T \vec{w} \\ & \text{subject to} \quad y_n (\vec{w}^T \vec{x}_n + b) \geq 1, \forall n \end{aligned}$$

- Dual

$$\begin{aligned} & \text{maximize}_{\vec{\alpha}} \quad \sum_{n=1}^N \alpha_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N \alpha_n \alpha_m y_n y_m \vec{x}_n^T \vec{x}_m \\ & \text{subject to} \quad \sum_{n=1}^N \alpha_n y_n = 0 \\ & \quad \alpha_n \geq 0, \forall n \end{aligned}$$

- Key messages:

- Both can be efficiently solved using QP solver
- We can infer the solution from one to the other

Reminders of definitions in general convex program:

$$\begin{aligned} & \text{minimize}_{\vec{w}} \quad f(\vec{w}) \\ & \text{subject to} \quad g_i(\vec{w}) \leq 0, \quad i = 1, \dots, k \\ & \quad \quad \quad h_j(\vec{w}) = 0, \quad j = 1, \dots, \ell \end{aligned}$$

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

$$\text{Primal: } \min_{\vec{w}} \max_{\vec{\alpha}, \vec{\beta}; \alpha_i \geq 0} L(\vec{w}, \vec{\alpha}, \vec{\beta})$$

$$\text{Dual: } \max_{\vec{\alpha}, \vec{\beta}; \alpha_i \geq 0} \min_{\vec{w}} L(\vec{w}, \vec{\alpha}, \vec{\beta})$$

Given optimal  $\vec{\alpha}^*$ :

- $\vec{w}^* = \sum_{\alpha_n^* > 0} \alpha_n^* y_n \vec{x}_n$
- Find a  $\alpha_n^* > 0$ ,  $b^* = y_n - \vec{w}^{*T} \vec{x}_n$

# Kernel Functions

- Define kernel function  $K_{\Phi}(\vec{x}, \vec{x}') = \Phi(\vec{x})^T \Phi(\vec{x}') (= \vec{z}^T \vec{z}')$ 
  - The **similarity** of two vectors in the projected space
- Goal: Compute  $K_{\Phi}(\vec{x}, \vec{x}')$  **without** transforming  $\vec{x}$  and  $\vec{x}'$
- Why? This enables us to operate in higher dimensional spaces without really worrying about the computational overhead.

# Kernel Trick: Utilize Dual and Kernel Functions

- The dual with nonlinear transform

$$\begin{aligned} & \text{maximize}_{\vec{\alpha}} \sum_{n=1}^N \alpha_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N \alpha_n \alpha_m y_n y_m \vec{z}_n^T \vec{z}_m \\ & \text{subject to } \sum_{n=1}^N \alpha_n y_n = 0 \\ & \quad \alpha_n \geq 0, \forall n \end{aligned}$$

- Plug in the kernel function  $K_{\Phi}(\vec{x}, \vec{x}') = \Phi(\vec{x})^T \Phi(\vec{x}')$

$$\begin{aligned} & \text{maximize}_{\vec{\alpha}} \sum_{n=1}^N \alpha_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N \alpha_n \alpha_m y_n y_m K_{\Phi}(\vec{x}_n, \vec{x}_m) \\ & \text{subject to } \sum_{n=1}^N \alpha_n y_n = 0 \\ & \quad \alpha_n \geq 0, \forall n \end{aligned}$$

- If the kernel can be computed efficiently, we can solve  $\vec{\alpha}^*$  efficiently.
- With kernel tricks, we can avoid the dependency on the dimension of  $\vec{z}$



# Recover $(\vec{w}^*, b^*)$ from $\vec{\alpha}^*$ with Kernel Tricks

- Note that  $\vec{\alpha}^*$  is solved in the  $\vec{z}$  space

- $\vec{w}^* = \sum_{\alpha_n^* > 0} \alpha_n^* y_n \Phi(\vec{x}_n)$
- Find a  $\alpha_n^* > 0$ ,  $b^* = y_n - \vec{w}^{*T} \Phi(\vec{x}_n)$
- We want to avoid the transformation!

- Let's look at the hypothesis

- $g(\vec{x}) = \text{sign}(\vec{w}^{*T} \Phi(\vec{x}) + b^*)$

$$\begin{aligned}\vec{w}^{*T} \Phi(\vec{x}) &= \left( \sum_{\alpha_n^* > 0} \alpha_n^* y_n \Phi(\vec{x}_n) \right)^T \Phi(\vec{x}) \\ &= \sum_{\alpha_n^* > 0} \alpha_n^* y_n \Phi(\vec{x}_n)^T \Phi(\vec{x}) \\ &= \sum_{\alpha_n^* > 0} \alpha_n^* y_n K(\vec{x}_n, \vec{x})\end{aligned}$$

Instead of storing  $(\vec{w}^*, b^*)$ , we can store “support vectors” (points with  $\alpha_n^* > 0$ ) and make predictions accordingly.

$$\begin{aligned}b^* &= y_n - \vec{w}^{*T} \Phi(\vec{x}_n) \\ &= y_n - \left( \sum_{\alpha_m^* > 0} \alpha_m^* y_m \Phi(\vec{x}_m) \right)^T \Phi(\vec{x}_n) \\ &= y_n - \sum_{\alpha_m^* > 0} \alpha_m^* y_m K(\vec{x}_m, \vec{x}_n)\end{aligned}$$

- Still can be computed in the  $\vec{x}$  space!

# Today's Lecture

The notes are not intended to be comprehensive. They should be accompanied by lectures and/or textbook.  
Let me know if you spot errors.

# Kernel Functions

$K_{\Phi}(\vec{x}, \vec{x}')$ : **Inner products** of two points  $\Phi(\vec{x})^T \Phi(\vec{x}')$  in the transformed space  
**Similarity** of two points  $\Phi(\vec{x})$  and  $\Phi(\vec{x}')$  in the transformed space

# Polynomial Kernel

$$\text{Kernel } K(\vec{x}, \vec{x}') = \Phi(\vec{x})^T \Phi(\vec{x}')$$

- Example in the last lecture: 2<sup>nd</sup> order polynomial for 2-d  $\vec{x}$ 
  - $\vec{x} = (x_1, x_2)$
  - $\vec{z} = \Phi_2(\vec{x}) = (1, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_1x_2, x_1^2, x_2^2)$
  - $\vec{z}' = \Phi_2(\vec{x}') = (1, \sqrt{2}x'_1, \sqrt{2}x'_2, \sqrt{2}x'_1x'_2, x_1'^2, x_2'^2)$
  - $$\begin{aligned}\vec{z}^T \vec{z}' &= 1 + 2x_1x'_1 + 2x_2x'_2 + 2x_1x'_1x_2x'_2 + (x_1x'_1)^2 + (x_2x'_2)^2 \\ &= (1 + x_1x'_1 + x_2x'_2)^2 \\ &= (1 + \vec{x}^T \vec{x}')^2\end{aligned}$$
- General 2<sup>nd</sup> order polynomial
  - $\vec{x} = (x_1, x_2, \dots, x_d)$
  - $$\begin{aligned}K_{\Phi_2}(\vec{x}, \vec{x}') &= (1 + \vec{x}^T \vec{x}')^2 \\ &= (1 + x_1x'_1 + x_2x'_2 + \dots + x_dx'_d)^2\end{aligned}$$

# Polynomial Kernel

General form of polynomial kernel:

$$K(\vec{x}, \vec{x}') = (\mathbf{a}\vec{x}^T\vec{x}' + \mathbf{b})^Q$$

- $\vec{x} = (x_1, x_2, \dots, x_d)$
- 2<sup>nd</sup> order polynomial kernel  $K_{\Phi_2}(\vec{x}, \vec{x}') = (1 + \vec{x}^T\vec{x}')^2$
- Q-th order Polynomial kernel  $K_{\Phi_Q}(\vec{x}, \vec{x}') = (1 + \vec{x}^T\vec{x}')^Q$   
 $= (1 + x_1x'_1 + \dots + x_dx'_d)^Q$
- Computational complexity
  - Dimension of  $\Phi_Q(\vec{x})$ :  $\binom{Q+d}{Q}$
  - Direct computation of  $\Phi_Q(\vec{x})^T\Phi_Q(\vec{x}')$ :  $O\left(\binom{Q+d}{Q}\right)$
  - Computation through kernel  $K_{\Phi_Q}(\vec{x}, \vec{x}')$ :  $O(d)$

# We Only Need $\vec{z}$ Space to Exist

- In the discussion of polynomial kernels
  - We have a target transformation in mind
  - We want to find a corresponding kernel function
- In fact, as long as  $K(\vec{x}, \vec{x}')$  is an inner product in **some**  $\vec{z}$  space, we are good
  - Just plug in the kernel in the dual formulation
  - We obtain a linear separator in the corresponding  $\vec{z}$  space

$$\begin{aligned} & \text{maximize}_{\vec{\alpha}} \sum_{n=1}^N \alpha_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N \alpha_n \alpha_m y_n y_m K_{\Phi}(\vec{x}_n, \vec{x}_m) \\ & \text{subject to } \sum_{n=1}^N \alpha_n y_n = 0 \\ & \quad \alpha_n \geq 0, \forall n \end{aligned}$$

# Gaussian RBF Kernel

- $K(\vec{x}, \vec{x}') = e^{-\gamma \|\vec{x} - \vec{x}'\|^2}$
- What's the corresponding  $\vec{z}$  space? (What is  $\Phi$  such that  $\Phi(\vec{x})^T \Phi(\vec{x}') = e^{-\gamma \|\vec{x} - \vec{x}'\|^2}$ )
  - For illustrative purpose, make  $\vec{x} = x$  be 1 dimensional and  $\gamma = 1$

$$\begin{aligned} K(\vec{x}, \vec{x}') &= e^{-(x-x')^2} \\ &= e^{-x^2 + 2xx' - x'^2} \\ &= e^{-x^2} e^{-x'^2} e^{2xx'} \\ &= e^{-x^2} e^{-x'^2} \sum_{k=0}^{\infty} \frac{(2xx')^k}{k!} \\ &= \sum_{k=0}^{\infty} e^{-x^2} \sqrt{\frac{2^k}{k!}} x^k e^{-x'^2} \sqrt{\frac{2^k}{k!}} x'^k \end{aligned}$$

$$\text{Taylor expansion: } e^{2xx'} = \sum_{k=0}^{\infty} \frac{(2xx')^k}{k!}$$

- The corresponding  $\Phi(x) = e^{-x^2} \left( 1, \sqrt{\frac{2}{1}} x, \sqrt{\frac{2^2}{2!}} x^2, \dots \right)$

# Gaussian RBF Kernel

- $K(\vec{x}, \vec{x}') = e^{-\gamma \|\vec{x} - \vec{x}'\|^2}$
- The corresponding transform in 1-dim input  $\vec{x} = x$ 
  - $\Phi(x) = e^{-x^2} \left( 1, \sqrt{\frac{2}{1}} x, \sqrt{\frac{2^2}{2!}} x^2, \dots \right)$
- $K(\vec{x}, \vec{x}')$  is the inner product of two vectors in an **infinite dimensional** space!
- When we plug in  $K(\vec{x}, \vec{x}')$  in dual SVM
  - We are finding the **max-margin** separator in an **infinite dimensional** space
  - Seems to introduce infinite generalization error?
    - Maximizing margin help mitigate this issue
    - The number of support vectors provides indicators on the generalization



# Design Your Own Kernel? [Safe to Skip]

- Say we design a kernel function, how do we know whether it is valid, i.e., whether there is a corresponding  $\vec{z}$  space?
- Mercer's condition (See discussion in LFD 8.3.2)
  - Kernel matrix

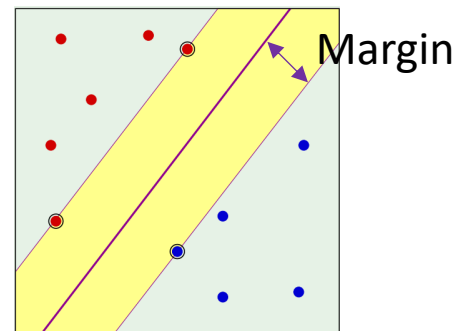
$$\begin{bmatrix} K(\mathbf{x}_1, \mathbf{x}_1) & K(\mathbf{x}_1, \mathbf{x}_2) & \dots & K(\mathbf{x}_1, \mathbf{x}_N) \\ K(\mathbf{x}_2, \mathbf{x}_1) & K(\mathbf{x}_2, \mathbf{x}_2) & \dots & K(\mathbf{x}_2, \mathbf{x}_N) \\ \dots & \dots & \dots & \dots \\ K(\mathbf{x}_N, \mathbf{x}_1) & K(\mathbf{x}_N, \mathbf{x}_2) & \dots & K(\mathbf{x}_N, \mathbf{x}_N) \end{bmatrix}$$

- $K(\vec{x}, \vec{x}')$  is a valid kernel if and only if the kernel matrix is always **symmetric positive semi-definite** for any  $\vec{x}_1, \dots, \vec{x}^N$

# Summary of What We Talked About So Far

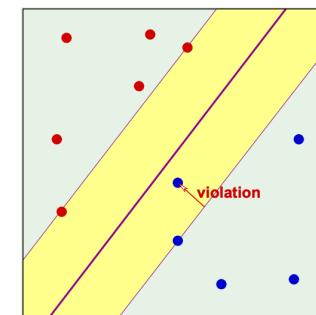
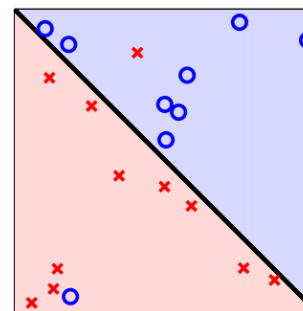
## Hard-Margin SVM (Separable Data)

$$\begin{aligned} &\text{minimize}_{\vec{w}, b} \quad \frac{1}{2} \vec{w}^T \vec{w} \\ &\text{subject to} \quad y_n (\vec{w}^T \vec{x}_n + b) \geq 1, \forall n \end{aligned}$$



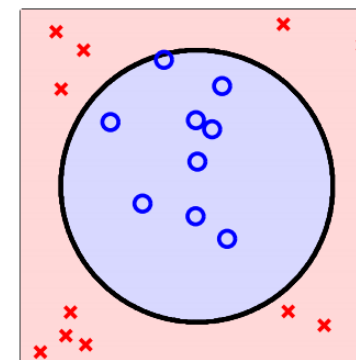
## Soft-Margin SVM (Tolerate Noise)

$$\begin{aligned} &\text{minimize}_{\vec{w}, b, \vec{\xi}} \quad \frac{1}{2} \vec{w}^T \vec{w} + C \sum_{n=1}^N \xi_n \\ &\text{subject to} \quad y_n (\vec{w}^T \vec{x}_n + b) \geq 1 - \xi_n, \forall n \\ &\quad \quad \quad \xi_n \geq 0, \forall n \end{aligned}$$



## Kernel Formulation of Hard-Margin SVM

$$\begin{aligned} &\text{maximize}_{\vec{\alpha}} \quad \sum_{n=1}^N \alpha_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N \alpha_n \alpha_m y_n y_m K_{\Phi}(\vec{x}_n, \vec{x}_m) \\ &\text{subject to} \quad \sum_{n=1}^N \alpha_n y_n = 0 \\ &\quad \quad \quad \alpha_n \geq 0, \forall n \end{aligned}$$



# Kernel Version of Soft-Margin SVM

- Soft-Margin SVM

$$\begin{aligned} & \text{minimize}_{\vec{w}, b, \vec{\xi}} \quad \frac{1}{2} \vec{w}^T \vec{w} + C \sum_{n=1}^N \xi_n \\ & \text{subject to} \quad y_n (\vec{w}^T \vec{x}_n + b) \geq 1 - \xi_n, \forall n \\ & \quad \quad \quad \xi_n \geq 0, \forall n \end{aligned}$$

- Kernel Version of Soft-Margin SVM

$$\begin{aligned} & \text{maximize}_{\vec{\alpha}} \quad \sum_{n=1}^N \alpha_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N \alpha_n \alpha_m y_n y_m K_{\Phi}(\vec{x}_n, \vec{x}_m) \\ & \text{subject to} \quad \sum_{n=1}^N \alpha_n y_n = 0 \\ & \quad \quad \quad 0 \leq \alpha_n \leq C, \forall n \end{aligned}$$

- It can be obtained by similar procedure as hard-margin version
- We can obtain the same relationship between  $\vec{\alpha}^*$  and  $(\vec{w}^*, b^*)$

# Interpretation of Support Vectors

- $\alpha_n^* > 0 \Rightarrow (\vec{x}_n, y_n)$  is a support vector

- $y_n(\vec{w}^{*T} \vec{x}_n + b^*) = 1 - \xi_n$

- Utilizing complementary slackness

- When  $0 < \alpha_n^* < C$

- $\xi_n = 0$

- $y_n(\vec{w}^{*T} \vec{x}_n + b^*) = 1$

- $(\vec{x}_n, y_n)$  is a “margin” support vector

- When  $\alpha_n^* = C$

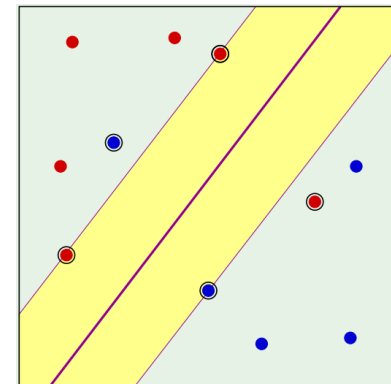
- $\xi_n > 0$

- $y_n(\vec{w}^{*T} \vec{x}_n + b^*) < 1$

- $(\vec{x}_n, y_n)$  is a “non-margin” support vector

$$\begin{aligned} & \text{minimize}_{\vec{w}, b, \vec{\xi}} \quad \frac{1}{2} \vec{w}^T \vec{w} + C \sum_{n=1}^N \xi_n \\ & \text{subject to} \quad y_n(\vec{w}^T \vec{x}_n + b) \geq 1 - \xi_n, \forall n \\ & \quad \quad \quad \xi_n \geq 0, \forall n \end{aligned}$$

$$\begin{aligned} & \text{maximize}_{\vec{\alpha}} \quad \sum_{n=1}^N \alpha_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N \alpha_n \alpha_m y_n y_m K_{\Phi}(\vec{x}_n, \vec{x}_m) \\ & \text{subject to} \quad \sum_{n=1}^N \alpha_n y_n = 0 \\ & \quad \quad \quad 0 \leq \alpha_n \leq C, \forall n \end{aligned}$$



# Another Look at Primal vs. Dual SVM

- Primal

$$\begin{aligned} &\text{minimize}_{\vec{w}, b} \quad \frac{1}{2} \vec{w}^T \vec{w} \\ &\text{subject to} \quad y_n (\vec{w}^T \vec{z}_n + b) \geq 1, \forall n \end{aligned}$$

- Learned hypothesis

- $g(\vec{x}) = \text{sign}(\vec{w}^*{}^T \Phi(\vec{x}) + b^*)$

- Primal view of SVM (**parametric**)

- We are learning the weights for SVM, i.e.,  $(\vec{w}^*, b^*)$
  - When using RBF Kernel, there are infinite number of parameters

- Dual kernel view of SVM (**nonparametric**)

- We are learning the support vectors, and use those for prediction

- Dual

$$\begin{aligned} &\text{maximize}_{\vec{\alpha}} \quad \sum_{n=1}^N \alpha_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N \alpha_n \alpha_m y_n y_m \vec{z}_n^T \vec{z}_m \\ &\text{subject to} \quad \sum_{n=1}^N \alpha_n y_n = 0 \\ &\quad \quad \quad \alpha_n \geq 0, \forall n \end{aligned}$$

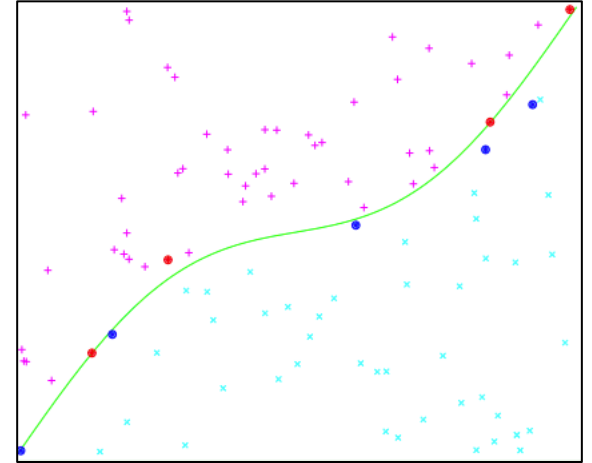
- Learned hypothesis

- $g(\vec{x}) = \text{sign}(\sum_{\alpha_n^* > 0} \alpha_n^* y_n K(\vec{x}_n, \vec{x}) + b^*))$
  - $(\alpha_n^* > 0 \Rightarrow \vec{x}_n \text{ is a support vector})$

# Kernel SVM and Radial Basis Functions

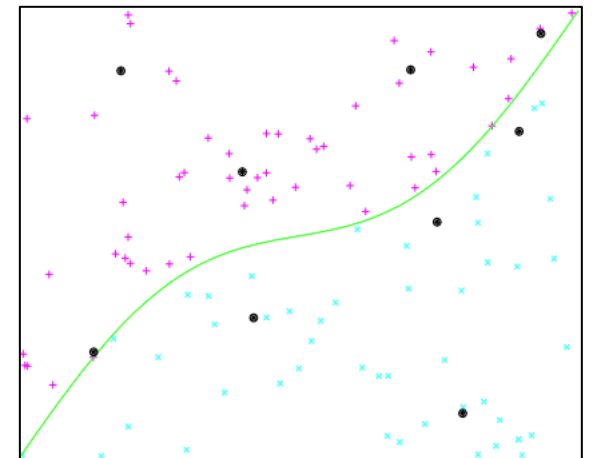
- Kernel SVM

- $g(\vec{x}) = \text{sign}\left(\sum_{\alpha_n^* > 0} \alpha_n^* y_n K(\vec{x}_n, \vec{x}) + b^*\right)$
- Use **support vectors** to characterize a hypothesis



- Radial Basis Functions

- $h(\vec{x}) = \text{sign}\left(\sum_{k=1}^K w_k \phi\left(\frac{\|\vec{x} - \vec{\mu}_k\|}{r}\right)\right)$
- Use **cluster centers** to characterize a hypothesis



# Neural Networks

# Perceptron

- What is a hypothesis in Perceptron

$$h(\vec{x}) = \text{sign}(\vec{w}^T \vec{x})$$

- Note that we have reverted back to our original notations

- $\vec{x} = (x_0, x_1, \dots, x_d)$
- $\vec{w} = (w_0, w_1, \dots, w_d)$
- Linear separator

$$h(\vec{x}) = \text{sign}(\vec{w}^T \vec{x})$$

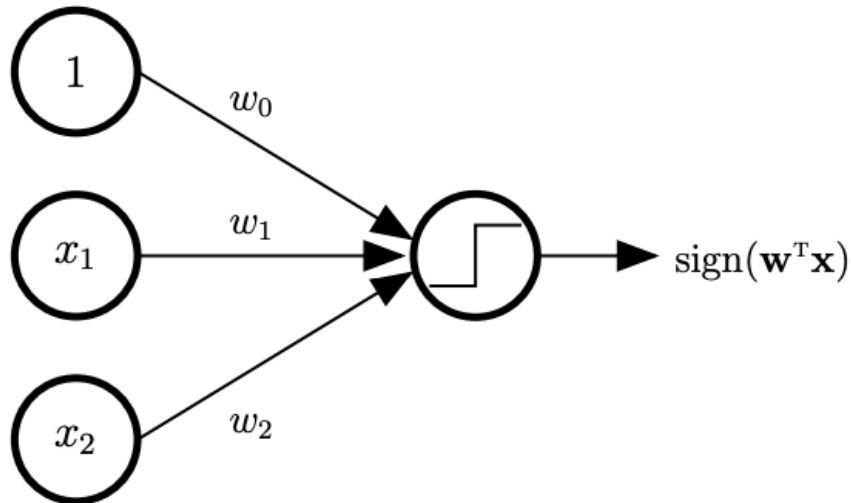


# Perceptron

- What is a hypothesis in Perceptron

$$h(\vec{x}) = \text{sign}(\vec{w}^T \vec{x})$$

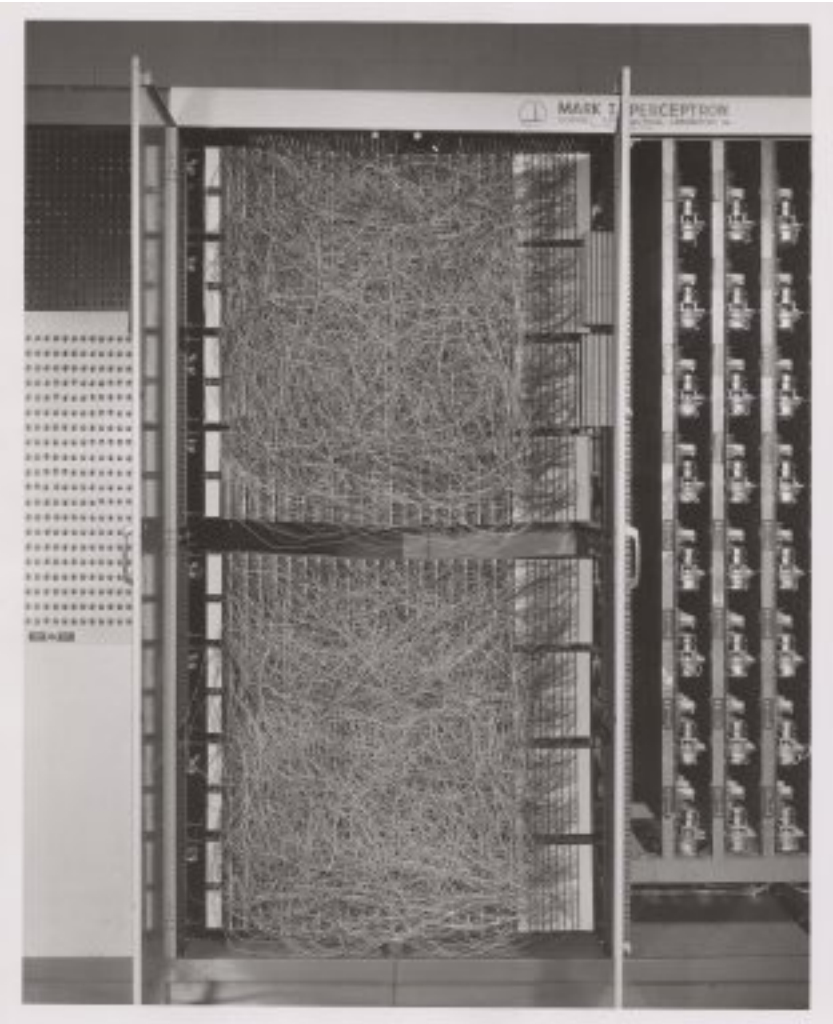
- Graphical representation of Perceptron



Inspired by [neurons](#):

The output signal is triggered when the weighted combination of the inputs is larger than some threshold

# The First Perceptron Machine



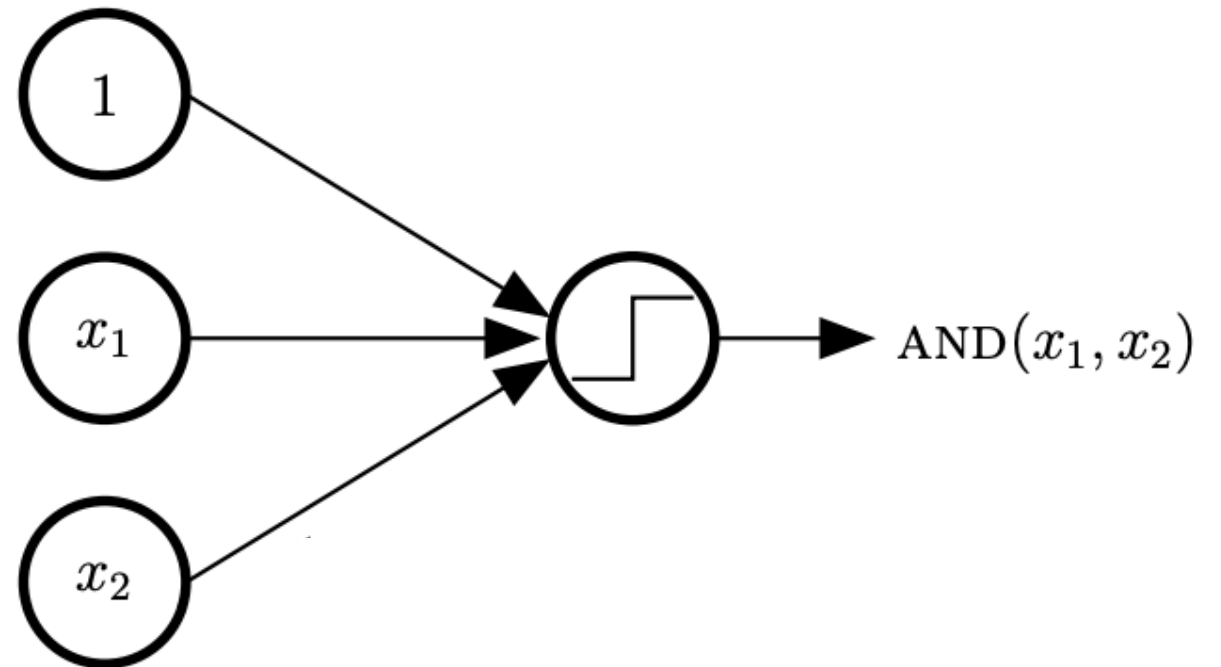
Mark I Perceptron machine, the first implementation of the perceptron algorithm. (From Wikipedia)

“the embryo of an electronic computer that [the Navy] expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence.” [1958]

# Implement Logic Gates with Perceptron

- $\text{AND}(x_1, x_2)$ 
  - Use +1 to denote “true” and -1 to denote “false”

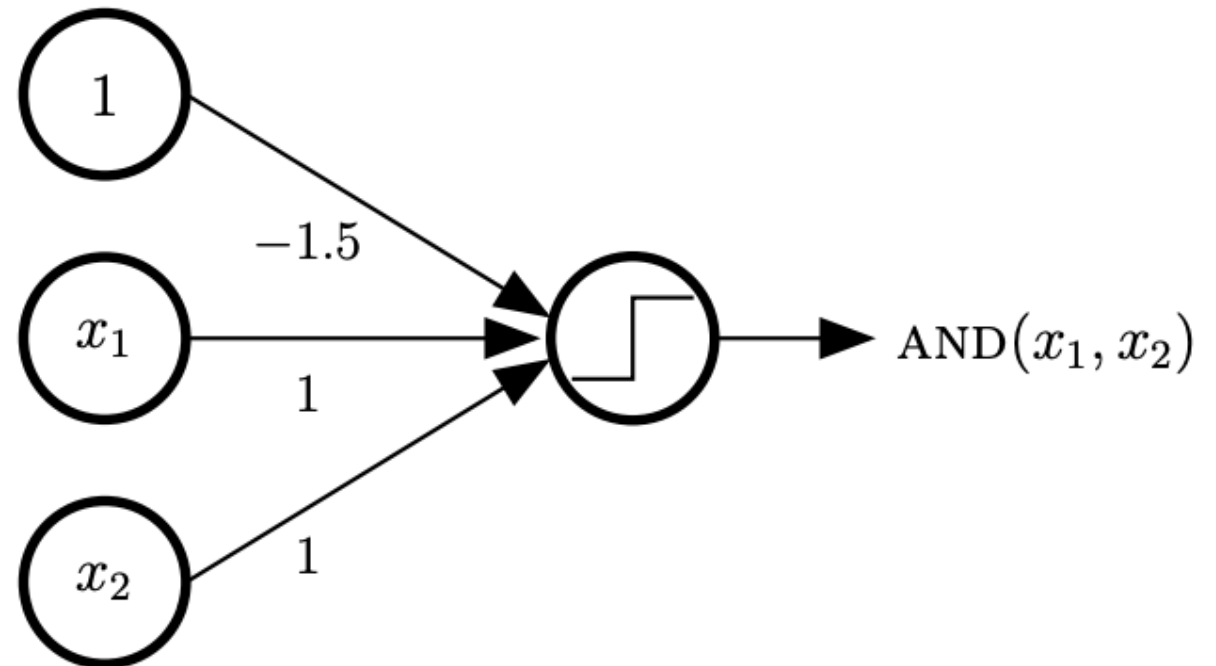
$x_1$	$x_2$	$\text{AND}(x_1, x_2)$
+1	+1	+1
+1	-1	-1
-1	+1	-1
-1	-1	-1



# Implement Logic Gates with Perceptron

- $\text{AND}(x_1, x_2)$ 
  - Use +1 to denote “true” and -1 to denote “false”

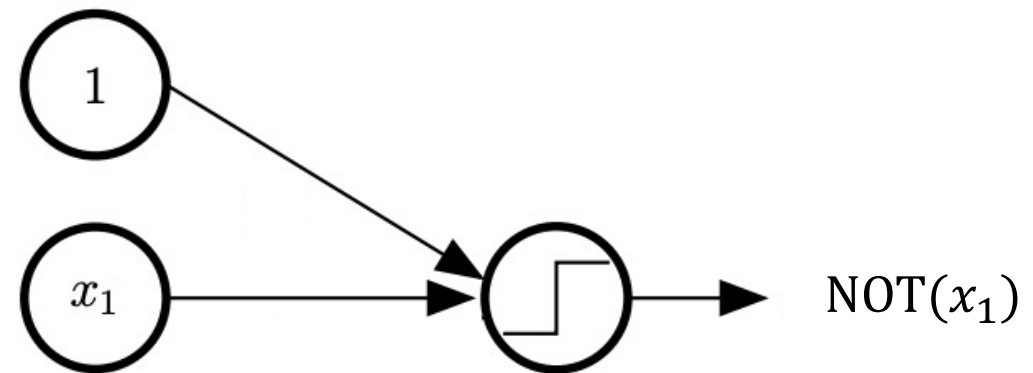
$x_1$	$x_2$	$\text{AND}(x_1, x_2)$
+1	+1	+1
+1	-1	-1
-1	+1	-1
-1	-1	-1



# Implement Logic Gates with Perceptron

- $\text{NOT}(x_1)$ 
  - Use  $+1$  to denote “true” and  $-1$  to denote “false”

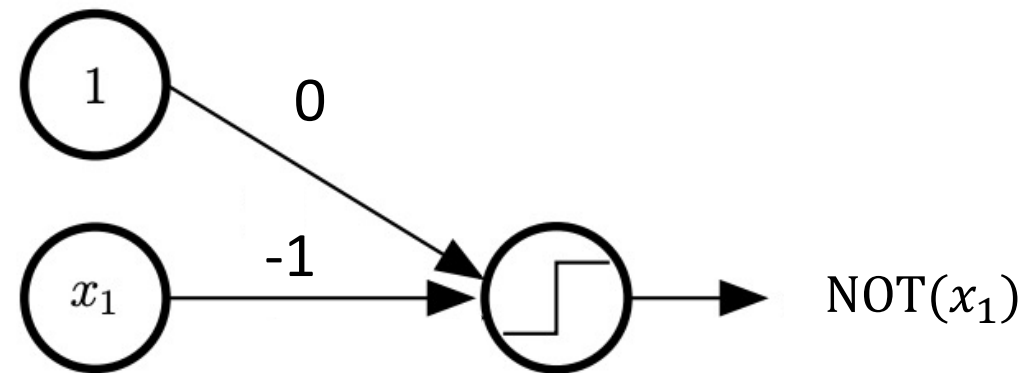
$x_1$	$\text{OR}(x)$
$+1$	$-1$
$-1$	$+1$



# Implement Logic Gates with Perceptron

- $\text{NOT}(x_1)$ 
  - Use +1 to denote “true” and -1 to denote “false”

$x_1$	$\text{OR}(x)$
+1	-1
-1	+1



# Practice: How to Implement OR and XOR?

- Use  $+1$  to denote “true” and  $-1$  to denote “false”

- $\text{OR}(x_1, x_2)$

$x_1$	$x_2$	$\text{OR}(x_1, x_2)$
$+1$	$+1$	$+1$
$+1$	$-1$	$+1$
$-1$	$+1$	$+1$
$-1$	$-1$	$-1$

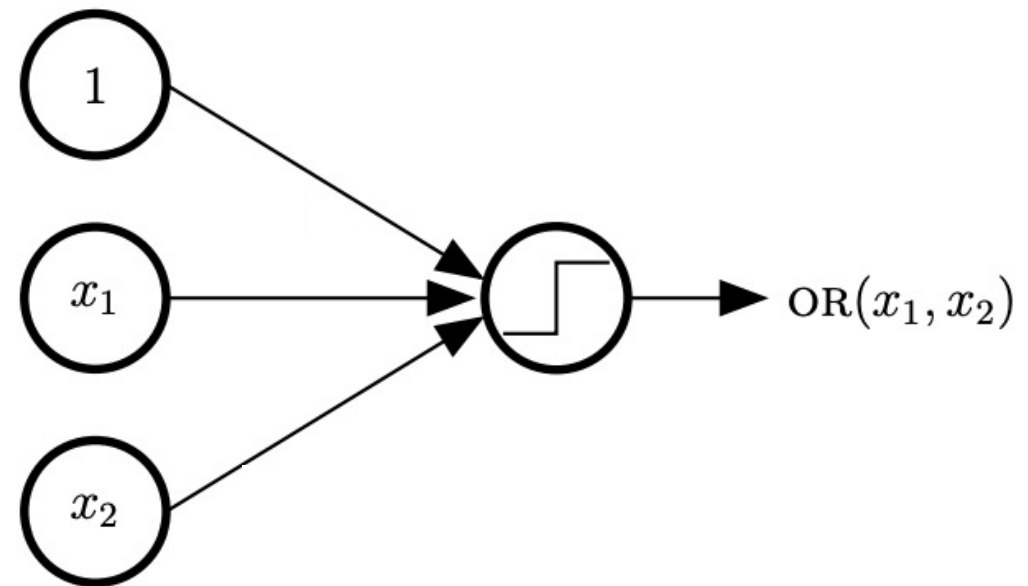
- $\text{XOR}(x_1, x_2)$

$x_1$	$x_2$	$\text{XOR}(x_1, x_2)$
$+1$	$+1$	$-1$
$+1$	$-1$	$+1$
$-1$	$+1$	$+1$
$-1$	$-1$	$-1$

# Implement Logic Gates with Perceptron

- $\text{OR}(x_1, x_2)$ 
  - Use +1 to denote “true” and -1 to denote “false”

$x_1$	$x_2$	$\text{OR}(x_1, x_2)$
+1	+1	+1
+1	-1	+1
-1	+1	+1
-1	-1	-1

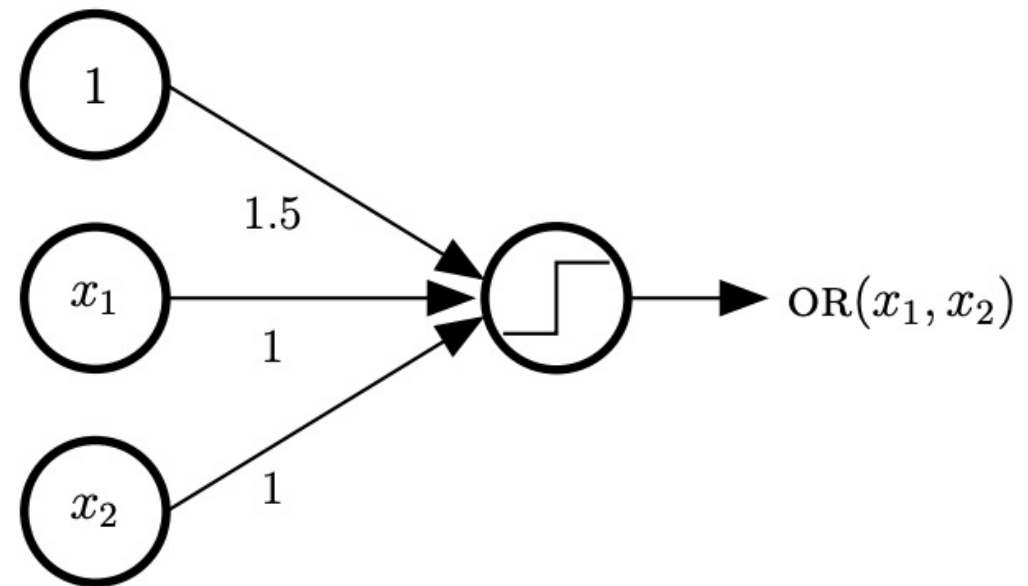




# Implement Logic Gates with Perceptron

- $\text{OR}(x_1, x_2)$ 
  - Use +1 to denote “true” and -1 to denote “false”

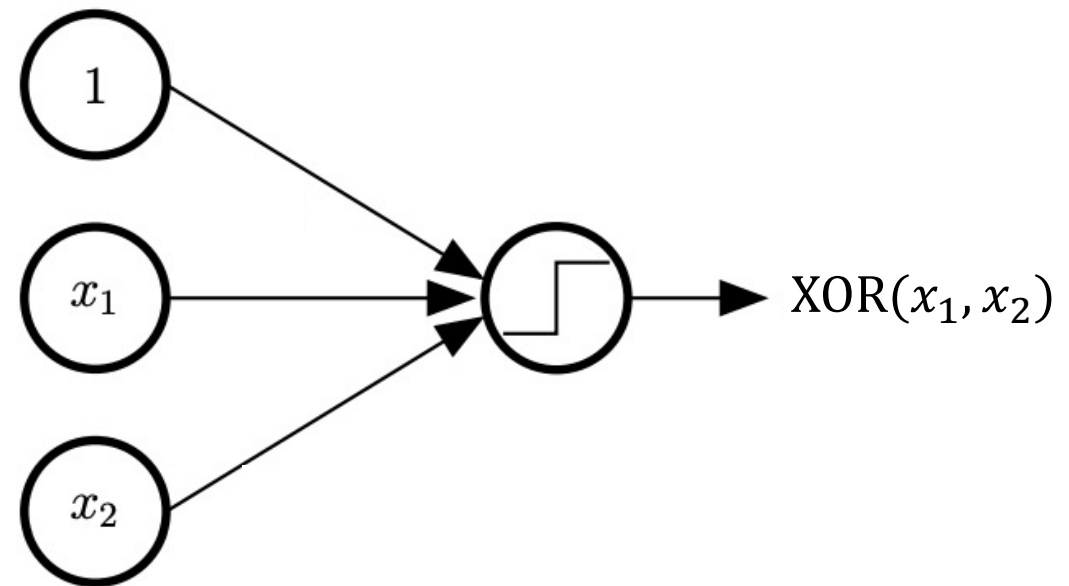
$x_1$	$x_2$	$\text{OR}(x_1, x_2)$
+1	+1	+1
+1	-1	+1
-1	+1	+1
-1	-1	-1



# Implement Logic Gates with Perceptron

- $\text{XOR}(x_1, x_2)$ 
  - Use +1 to denote “true” and -1 to denote “false”

$x_1$	$x_2$	$\text{XOR}(x_1, x_2)$
+1	+1	-1
+1	-1	+1
-1	+1	+1
-1	-1	-1



# Implement Logic Gates with Perceptron

- $\text{XOR}(x_1, x_2)$ 
  - Use +1 to denote “true” and -1 to denote “false”

$x_1$	$x_2$	$\text{XOR}(x_1, x_2)$
+1	+1	-1
+1	-1	+1
-1	+1	+1
-1	-1	-1

It is **impossible** to implement XOR using a single perceptron (draw the points in the 2-D space, you will see they are not linearly separable)

Stronger version:

It is **impossible** to implement XOR using a single **layer of** perceptrons

# Multi-Layer Perceptron

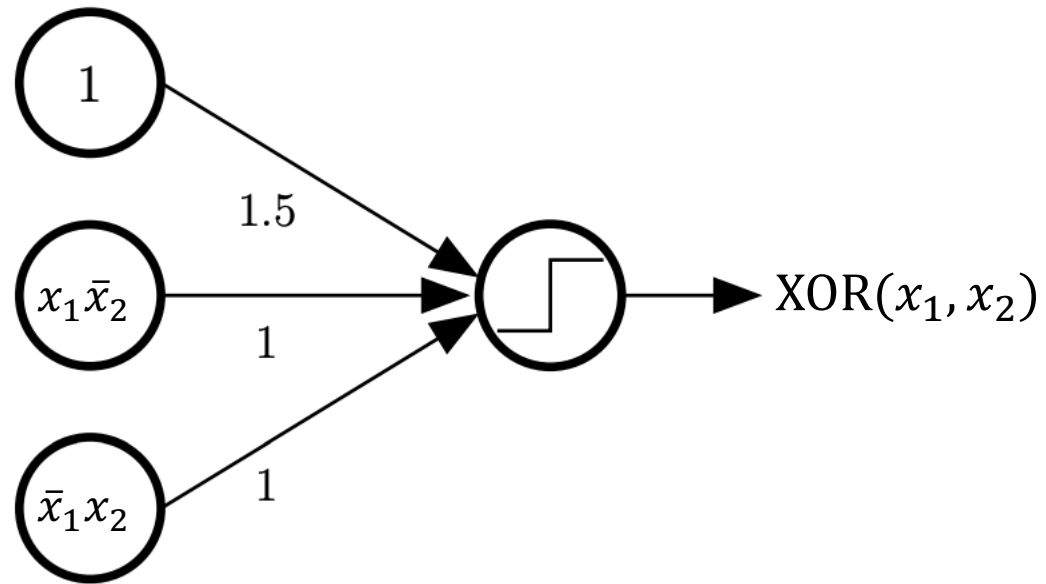


# Representing Boolean Operations

- $\text{AND}(x_1, x_2) \rightarrow x_1 x_2$
- $\text{OR}(x_1, x_2) \rightarrow x_1 + x_2$
- $\text{NOT}(x_1) \rightarrow \bar{x}_1$
  
- $\text{XOR}(x_1, x_2) \rightarrow x_1 \bar{x}_2 + \bar{x}_1 x_2$

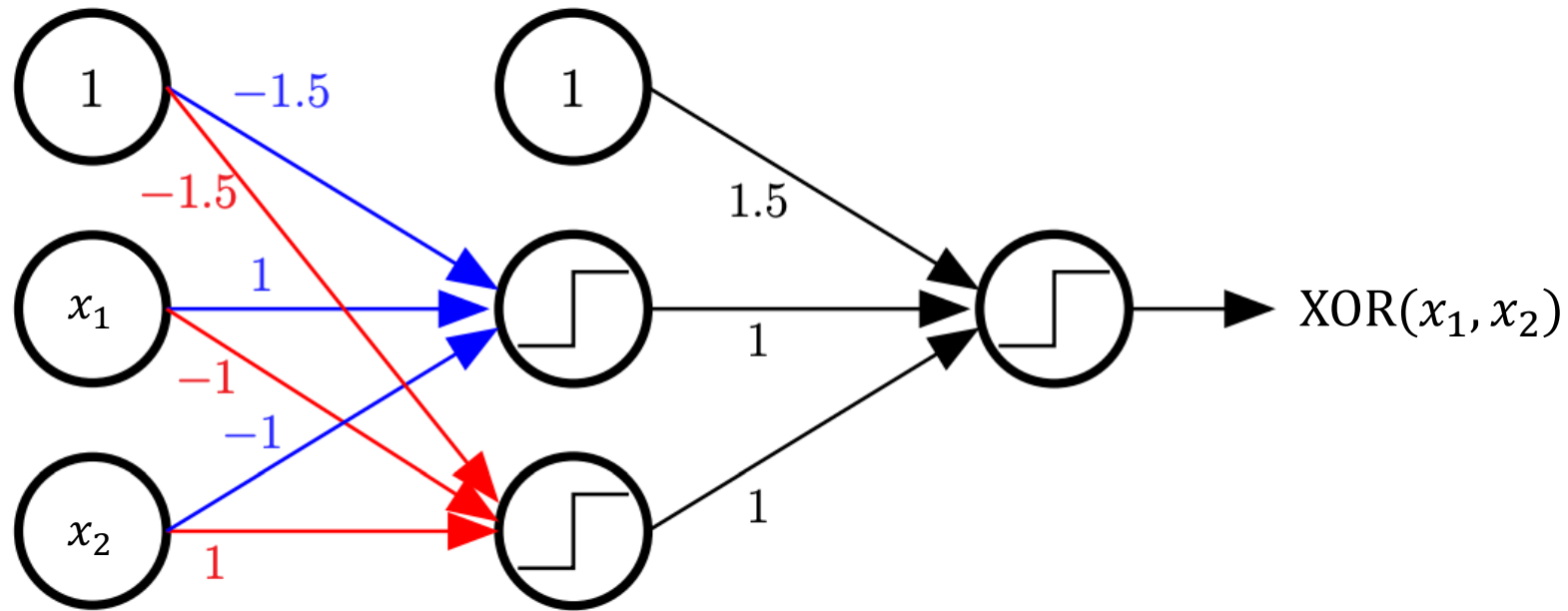
# Implementing XOR

- $\text{XOR}(x_1, x_2) \rightarrow x_1\bar{x}_2 + \bar{x}_1x_2$

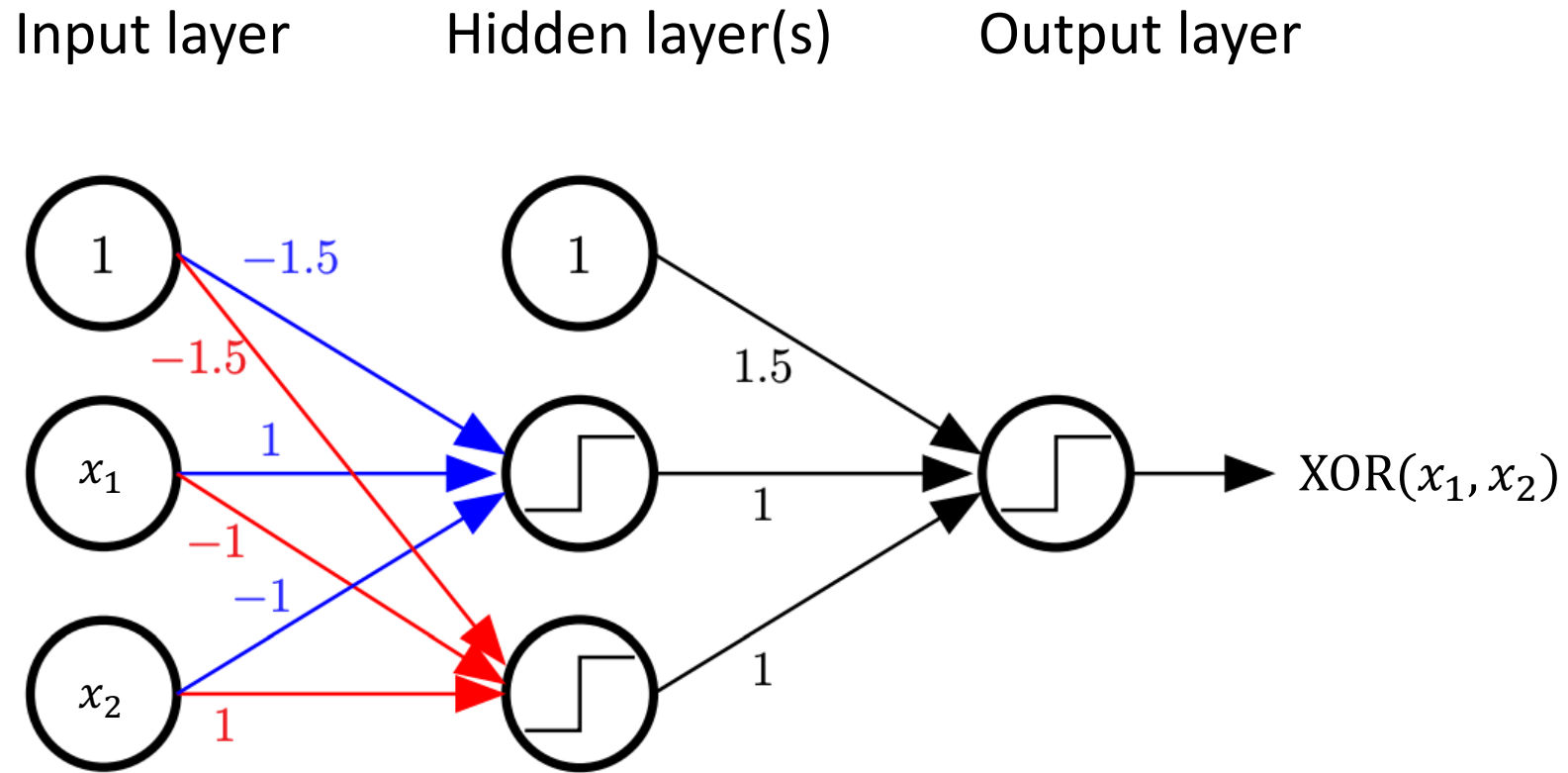


# Implementing XOR

- $\text{XOR}(x_1, x_2) \rightarrow x_1\bar{x}_2 + \bar{x}_1x_2$



# Multi-Layer Perceptron (MLP)



Feed-forward network

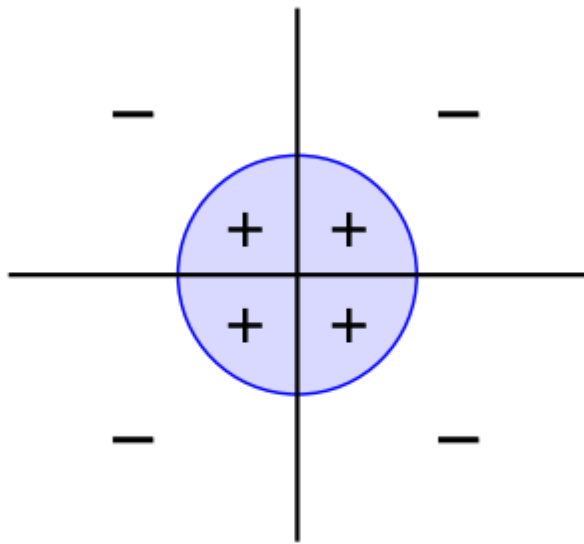


# The Power of Multi-Layer Perceptron (MLP)

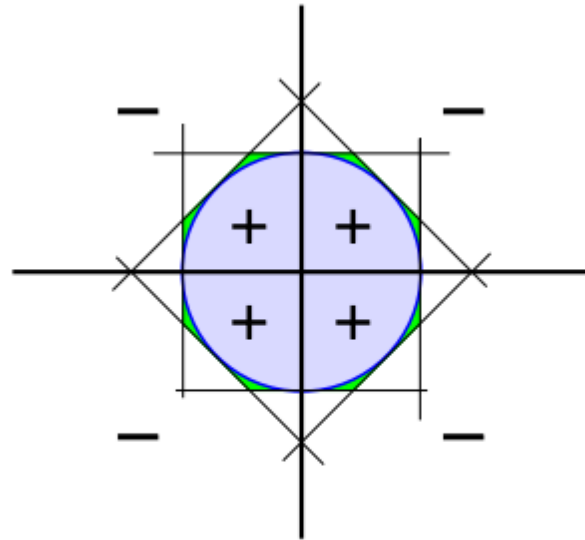
- We now know that we can implement XOR by introducing the hidden layer in MLP. But generally how powerful is MLP?
- Universal approximation theorem
  - a feed-forward network with **a single hidden layer** containing a finite number of neurons can approximate continuous functions on compact subsets of  $\mathbb{R}^n$ , under mild assumptions on the **activation function**.
- Three-layer MLP can **approximate ANY continuous target function!**

# Informal Intuitions of Universal Approximation

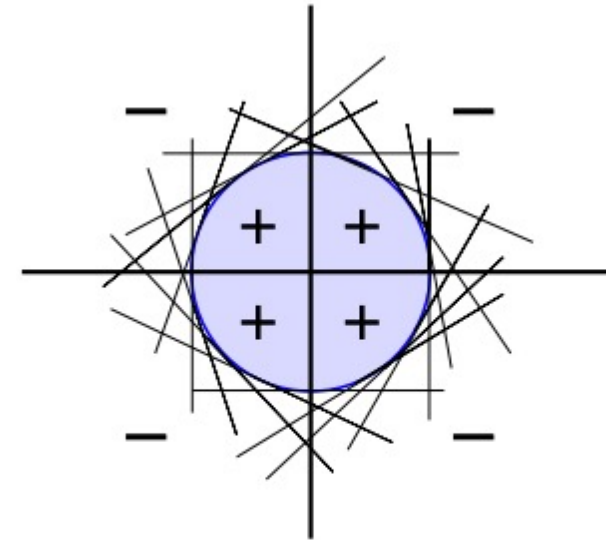
- A continuous separator can be "decomposed" into linear separators



Target



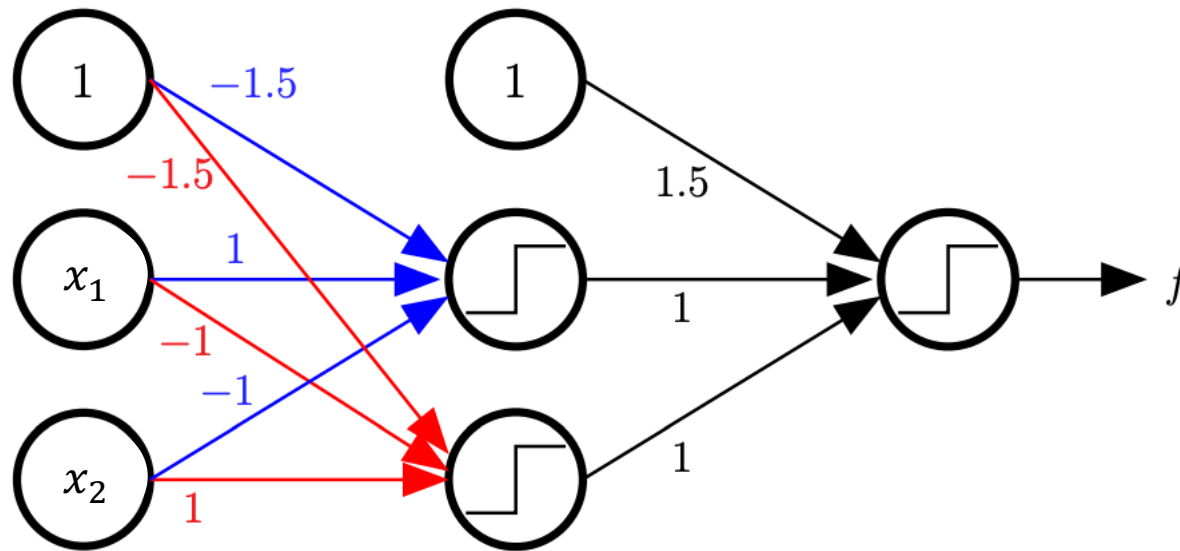
8 perceptrons



16 perceptrons

# How to Learn MLP From Data?

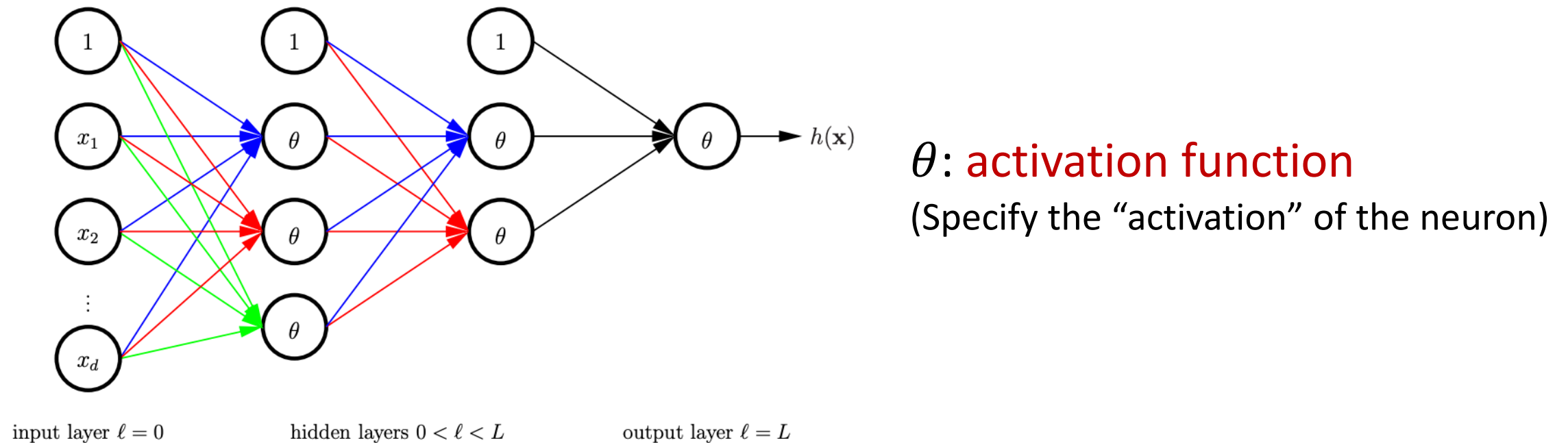
- Given  $D$  and the network structure, how to learn the “weights” (i.e., the weight vectors of every Perceptron)?



- Computationally challenging due to the “sign” function 

# Neural Networks

- A softened version of multi-layer Perceptron (MLP)



Next lecture: formally introduce neural networks and how to learn it from data