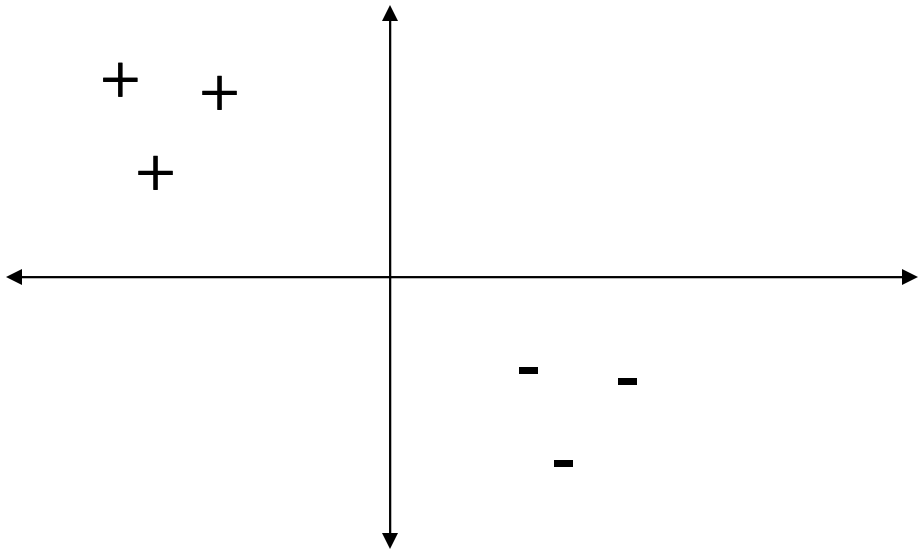
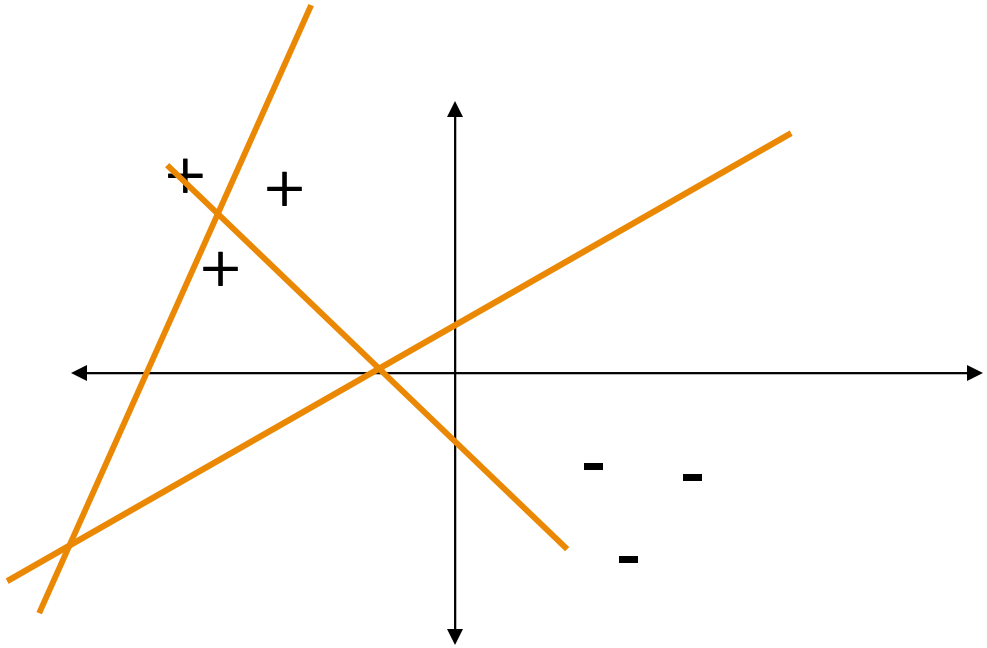
# ARTIFICIAL NEURAL NETWORKS

# Basic Idea_




**Decision boundary (WX = 0)**

# Basic Idea_




**Decision boundary (WX = 0)**

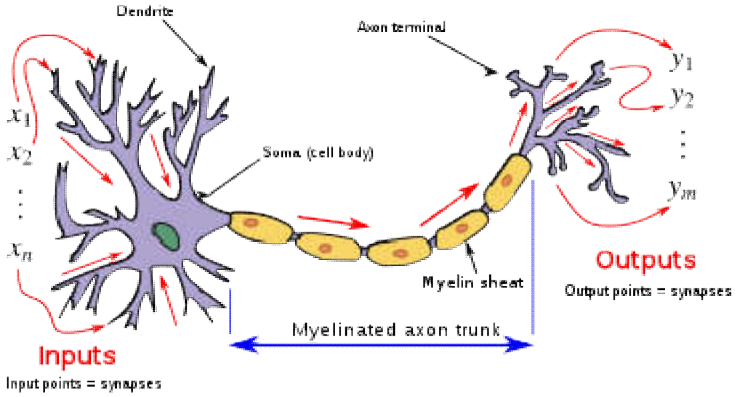
با یک خط می خوای متمایز کنیم --> اینور خط و اونور خط

Decision boundary : معادله یک خطی است که نقاطی روی اون خط مقدارشون صفره

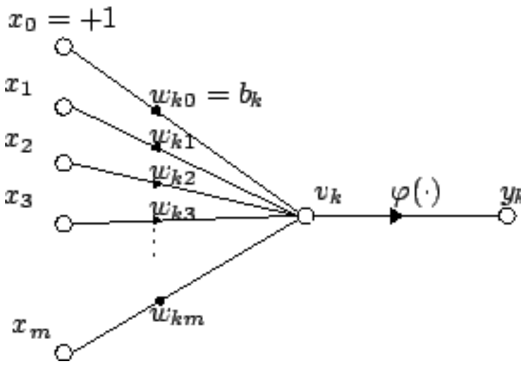
ما توی شبکه عصبی به دنبال بهترین خط است

# Artificial Neural Networks (ANN)

- **Basic Idea:** A complex non-linear function can be learned as a composition of simple processing units




https://en.wikipedia.org/

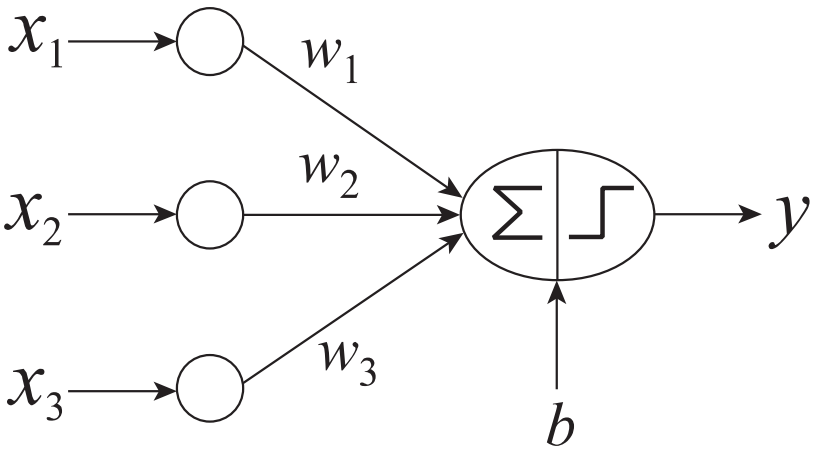


$$Y \sim WX$$

# Artificial Neural Networks (ANN)

- **Basic Idea:** A complex non-linear function can be learned as a composition of simple processing units
- ANN is a collection of simple processing units (nodes) that are connected by directed links (edges)
  - Every node receives signals from incoming edges, performs computations, and transmits signals to outgoing edges
  - Analogous to *human brain* where nodes are neurons and signals are electrical impulses
  - Weight of an edge determines the strength of connection between the nodes
- Simplest ANN: **Perceptron** (single neuron)

# Basic Architecture of Perceptron




$$y = \begin{cases} 1, & \text{if } \mathbf{w}^T\mathbf{x} + b > 0. \\ -1, & \text{otherwise.} \end{cases}$$

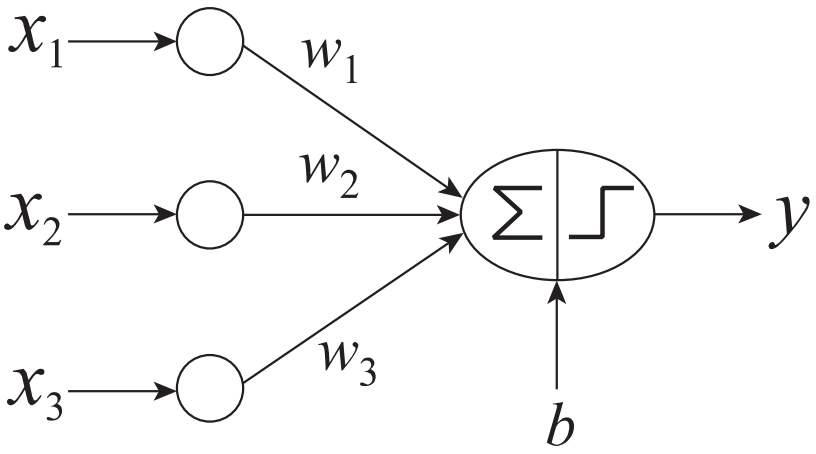
$$\tilde{\mathbf{w}} = (\mathbf{w}^T \; b)^T \qquad \tilde{\mathbf{x}} = (\mathbf{x}^T \; 1)^T$$

$$\hat{y} = sign(\tilde{\mathbf{w}}^T \tilde{\mathbf{x}})$$

Activation Function

- Learns linear decision boundaries
- Related to logistic regression (activation function is sign instead of sigmoid)

# Basic Architecture of Perceptron




$$y = \begin{cases} 1, & \text{if } \mathbf{w}^T\mathbf{x} + b > 0. \\ -1, & \text{otherwise.} \end{cases}$$

$$\tilde{\mathbf{w}} = (\mathbf{w}^T \; b)^T \qquad \tilde{\mathbf{x}} = (\mathbf{x}^T \; 1)^T$$

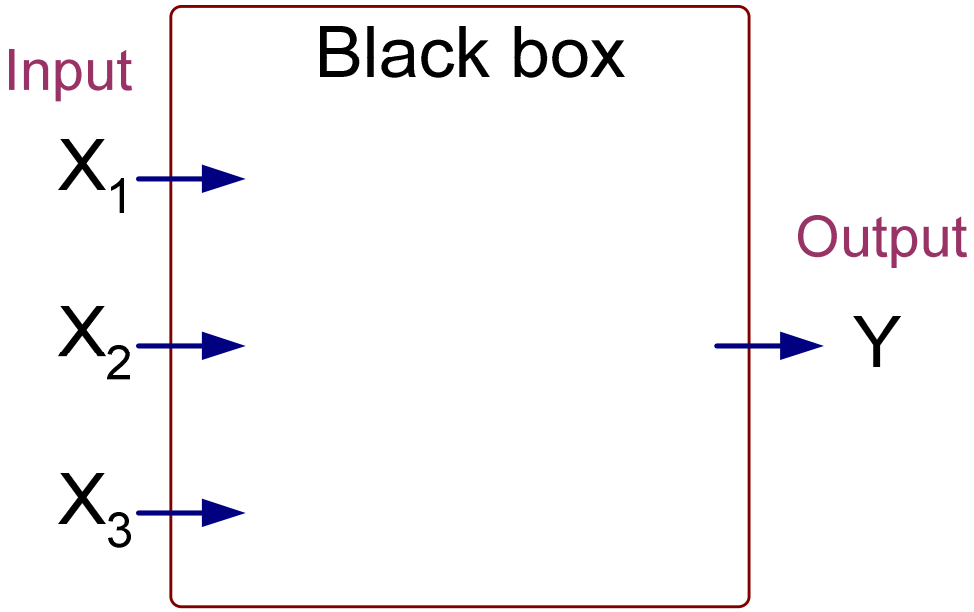
$$\hat{y} = sign(\tilde{\mathbf{w}}^T \tilde{\mathbf{x}})$$

Activation Function

- Learns linear decision boundaries
- Related to logistic regression (activation function is sign instead of sigmoid)

# Perceptron Example

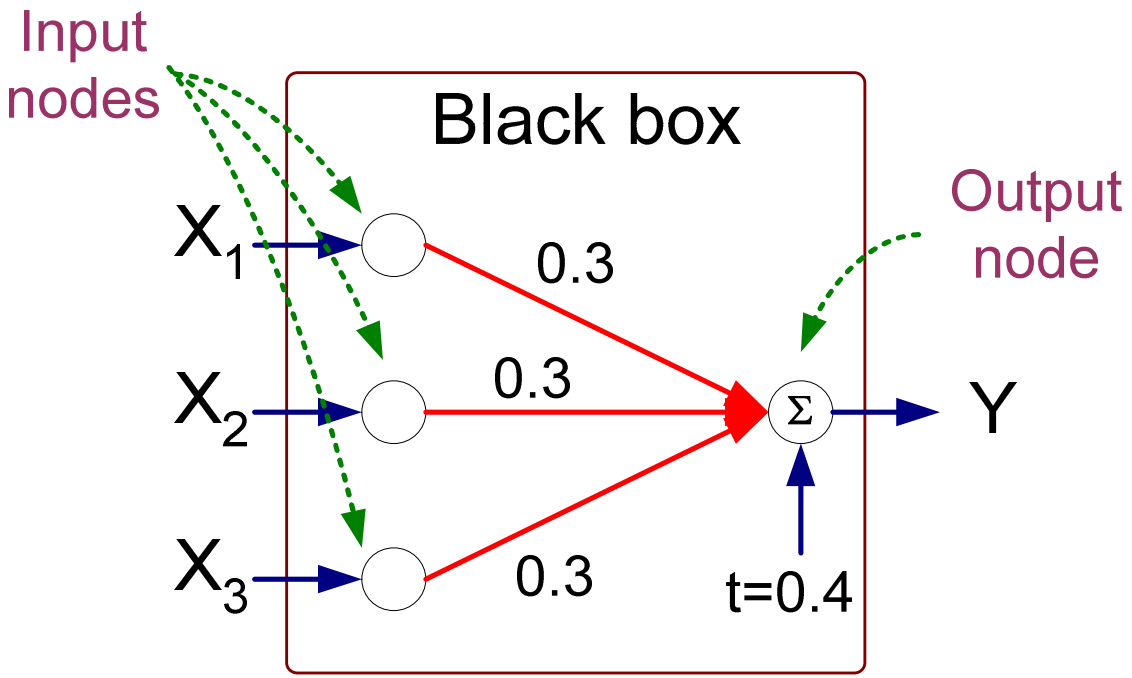
| $X_1$ | $X_2$ | $X_3$ | Y |
|---|---|---|---|
| 1 | 0 | 0 | -1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |
| 0 | 0 | 1 | -1 |
| 0 | 1 | 0 | -1 |
| 0 | 1 | 1 | 1 |
| 0 | 0 | 0 | -1 |




Output Y is 1 if at least two of the three inputs are equal to 1.

# Perceptron Example

| $X_1$ | $X_2$ | $X_3$ | Y |
|---|---|---|---|
| 1 | 0 | 0 | -1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |
| 0 | 0 | 1 | -1 |
| 0 | 1 | 0 | -1 |
| 0 | 1 | 1 | 1 |
| 0 | 0 | 0 | -1 |

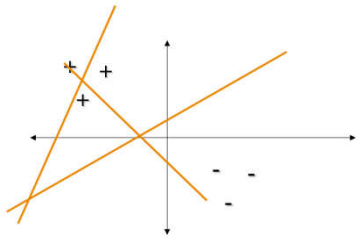



$$Y = sign\,(0.3X_1 + 0.3X_2 + 0.3X_3 - 0.4)$$

$$\text{where} \quad sign\,(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases}$$

# Perceptron Learning Rule

- Initialize the weights ($w_0$, $w_1$, …, $w_d$)
- Repeat
  - For each training example ($x_i$, $y_i$)
    - Compute $\hat{y}_i$
    - Update the weights:



$$w_j^{(k+1)} = w_j^{(k)} + \lambda\left(y_i - \hat{y}_i^{(k)}\right)x_{ij}$$

- Until stopping condition is met

- k: iteration number; $\lambda$: learning rate

# Perceptron Learning Rule

- Weight update formula:

$$w_j^{(k+1)} = w_j^{(k)} + \lambda\left(y_i - \hat{y}_i^{(k)}\right)x_{ij}$$

- Intuition:
  - Update weight based on error: e = $(y_i - \hat{y}_i)$
    - If y = $\hat{y}$, e=0: no update needed
    - If y > $\hat{y}$, e=2: weight must be increased (assuming Xij is positive) so that $\hat{y}$ will increase
    - If y < $\hat{y}$, e=-2: weight must be decreased (assuming Xij is positive) so that $\hat{y}$ will decrease

# Example of Perceptron Learning

$$\lambda = 0.1$$

| $X_1$ | $X_2$ | $X_3$ | Y |
|---|---|---|---|
| 1 | 0 | 0 | -1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |
| 0 | 0 | 1 | -1 |
| 0 | 1 | 0 | -1 |
| 0 | 1 | 1 | 1 |
| 0 | 0 | 0 | -1 |

| | $w_0$ | $w_1$ | $w_2$ | $w_3$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | -0.2 | -0.2 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0.2 |
| 3 | 0 | 0 | 0 | 0.2 |
| 4 | 0 | 0 | 0 | 0.2 |
| 5 | -0.2 | 0 | 0 | 0 |
| 6 | -0.2 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0.2 | 0.2 |
| 8 | -0.2 | 0 | 0.2 | 0.2 |

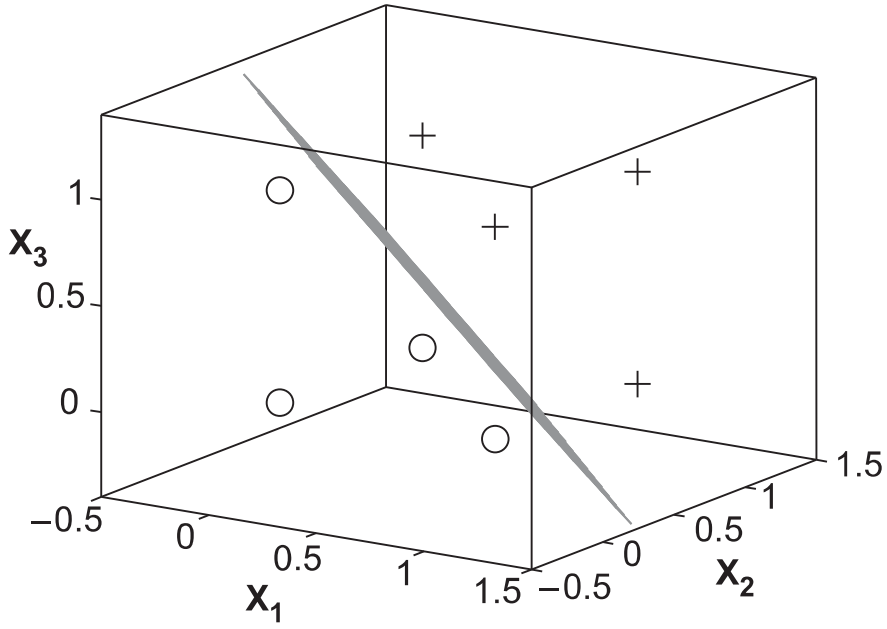
Weight updates over first epoch

| Epoch | $w_0$ | $w_1$ | $w_2$ | $w_3$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | -0.2 | 0 | 0.2 | 0.2 |
| 2 | -0.2 | 0 | 0.4 | 0.2 |
| 3 | -0.4 | 0 | 0.4 | 0.2 |
| 4 | -0.4 | 0.2 | 0.4 | 0.4 |
| 5 | -0.6 | 0.2 | 0.4 | 0.2 |
| 6 | -0.6 | 0.4 | 0.4 | 0.2 |

Weight updates over all epochs

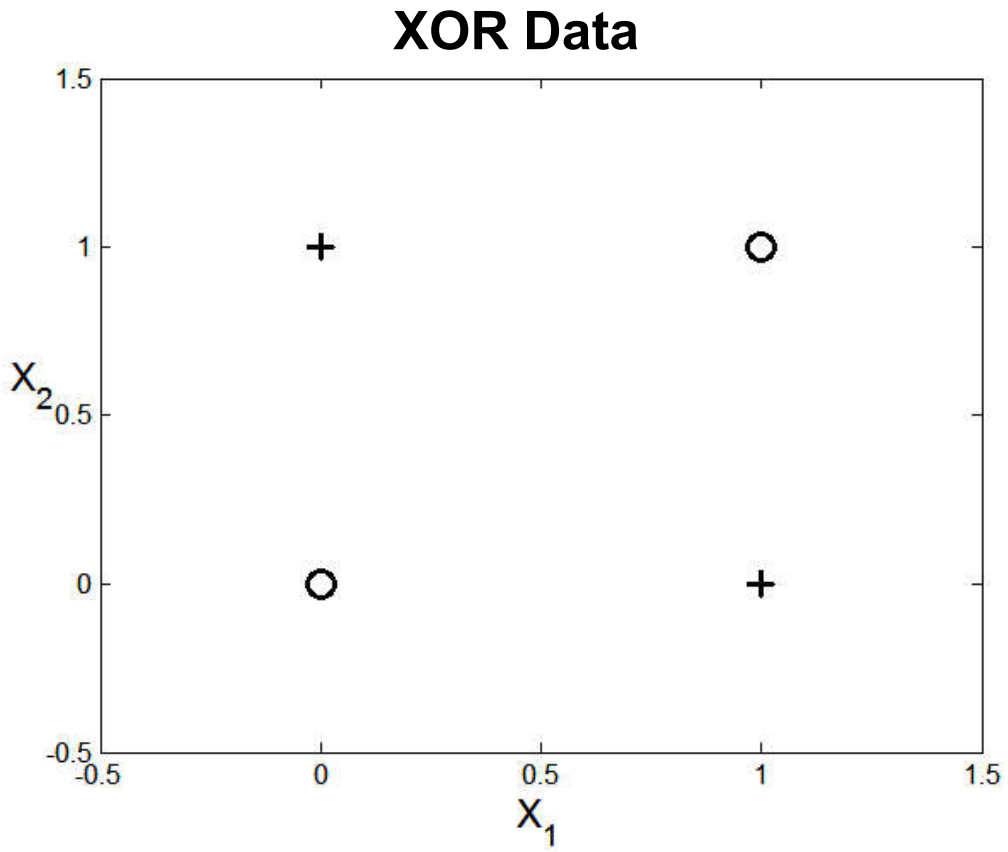
# Perceptron Learning

- Since y is a linear combination of input variables, decision boundary is linear



# Nonlinearly Separable Data

| $x_1$ | $x_2$ | y |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | -1 |

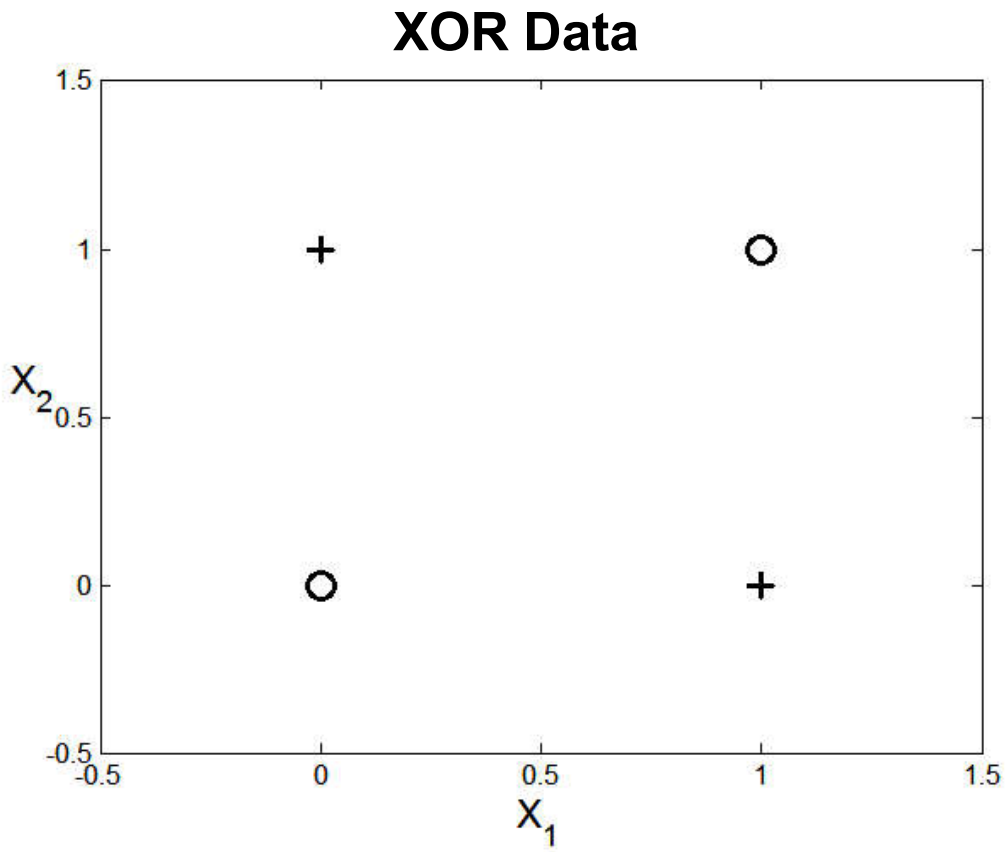


# Nonlinearly Separable Data

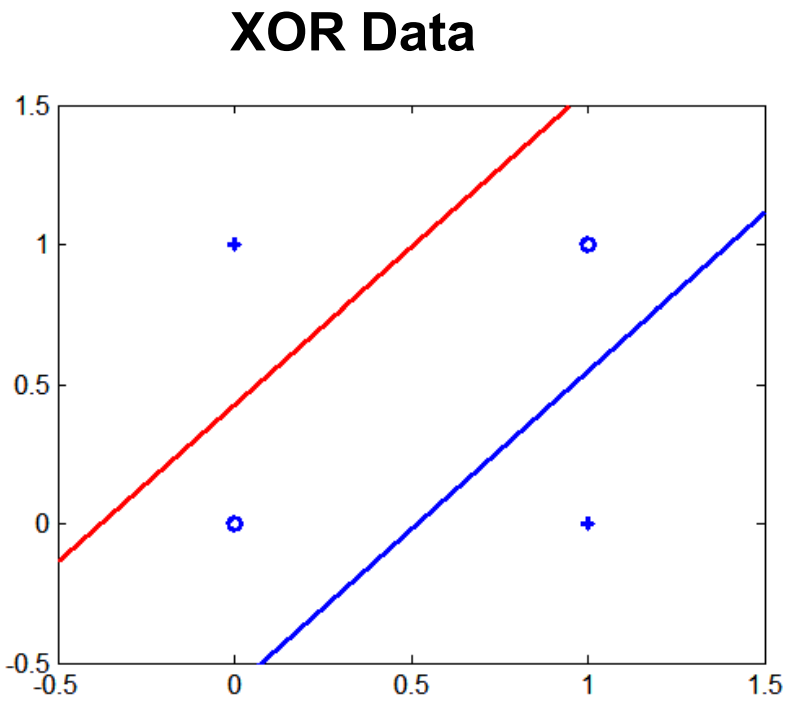
For nonlinearly separable problems, perceptron learning algorithm will fail because no linear hyperplane can separate the data perfectly

$$y = x_1 \oplus x_2$$

| $x_1$ | $x_2$ | y |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | -1 |

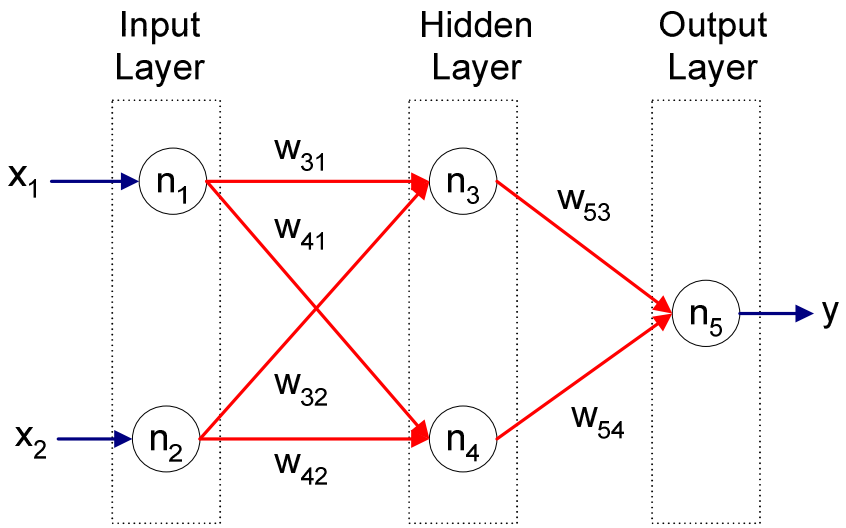


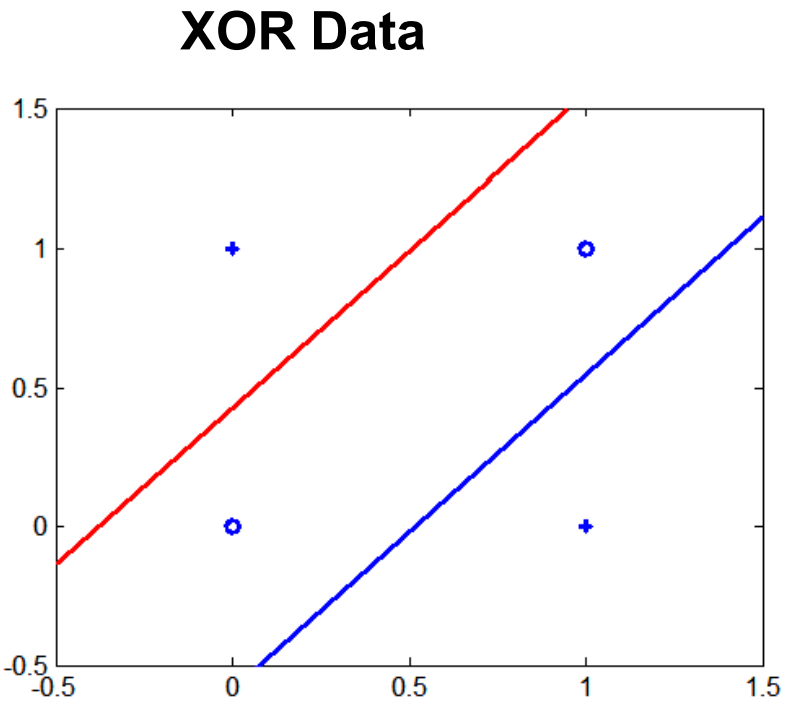
# Nonlinearly Separable Data



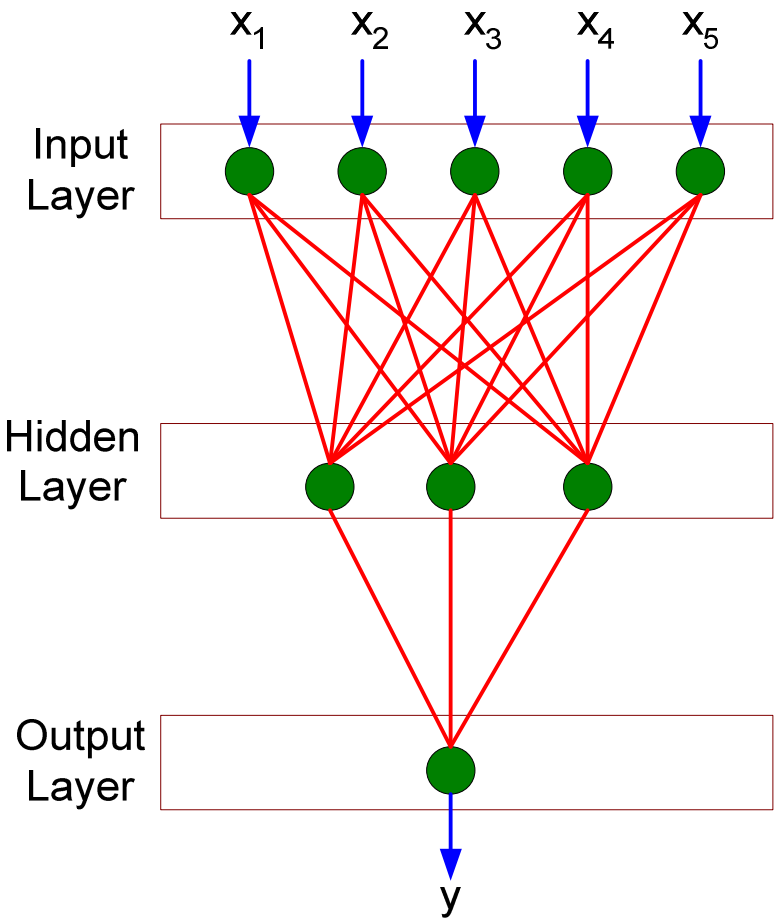
# Multi-layer Neural Network

- Multi-layer neural networks with at least one hidden layer can solve any type of classification task involving nonlinear decision surfaces



# Multi-layer Neural Network



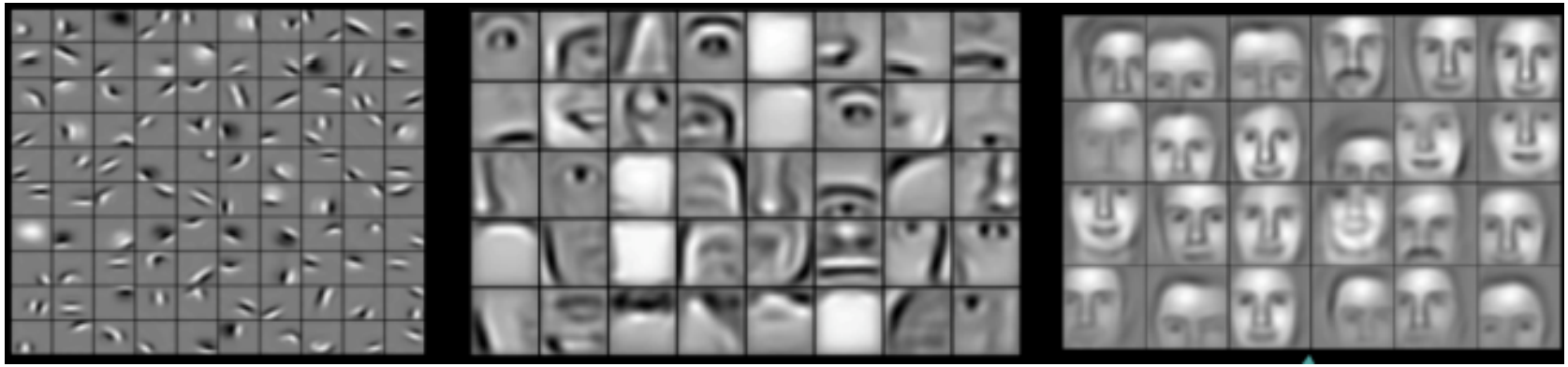

- More than one *hidden layer* of computing nodes
- Every node in a hidden layer operates on activations from preceding layer and transmits activations forward to nodes of next layer
- Also referred to as "feedforward neural networks"

# Why Multiple Hidden Layers?

- Activations at hidden layers can be viewed as

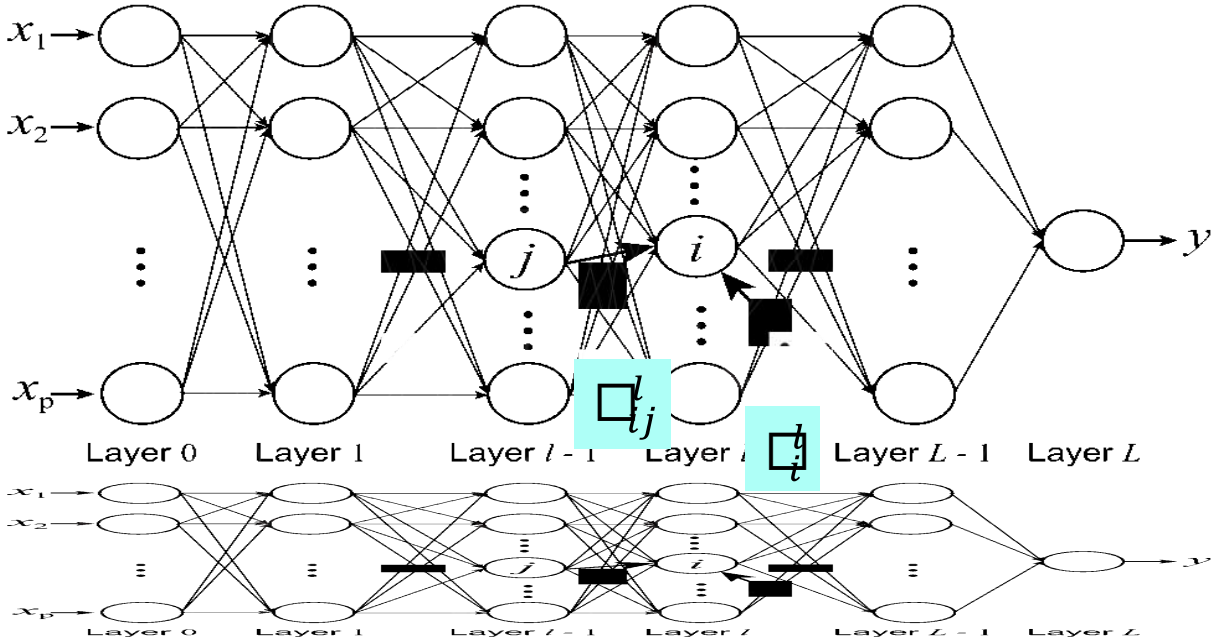
  features extracted as functions of inputs

- Every hidden layer represents a level of abstraction
  - *Complex features are compositions of simpler features*



- Number of layers is known as **depth** of ANN
  - *Deeper networks express complex hierarchy of features*

# Multi-Layer Network Architecture




$$a_i^l = f(z_i^l) = f\Big(\sum_j w_{ij}^l a_j^{l-1} + b_i^l\Big)$$

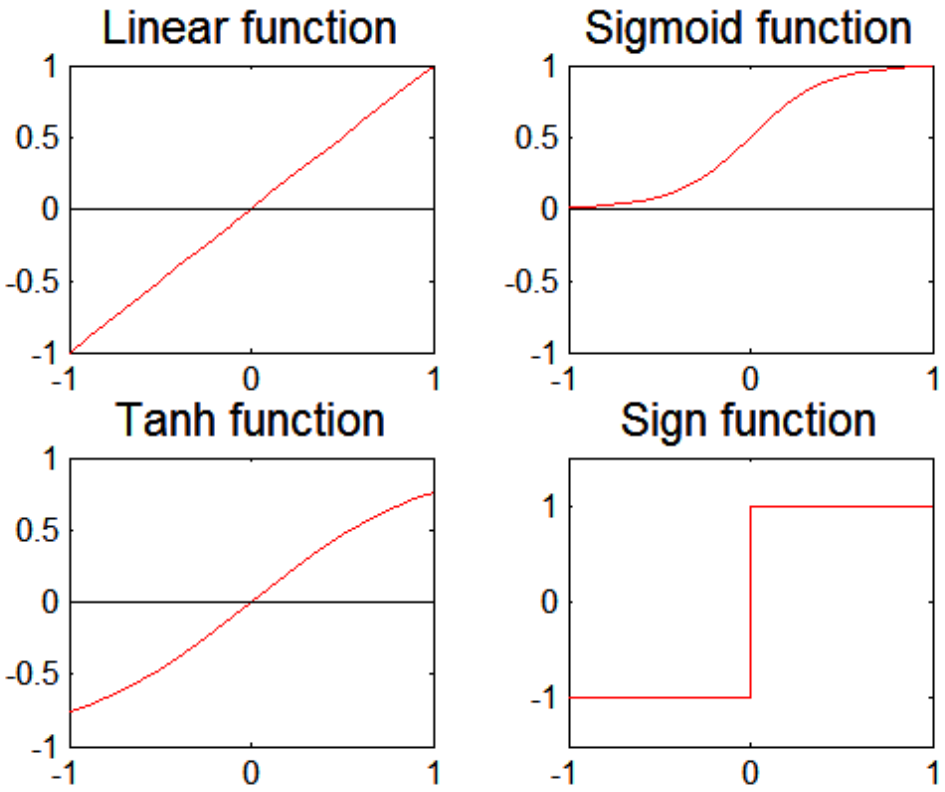
**Activation value**
at node i at layer l

**Activation Function**

**Linear Predictor**

# Activation Functions

$$a_i^l = f(z_i^l) = f\Big(\sum_j w_{ij}^l a_j^{l-1} + b_i^l\Big)$$




$$a_i^l = \sigma(z_i^l) = \frac{1}{1+e^{-z_i^l}}.$$

$$\frac{\partial a_i^l}{\partial z_i^l} = \frac{\partial\ \sigma(z_i^l)}{\partial z_i^l} = a_i^l(1-a_i^l)$$

# Learning Multi-layer Neural Network

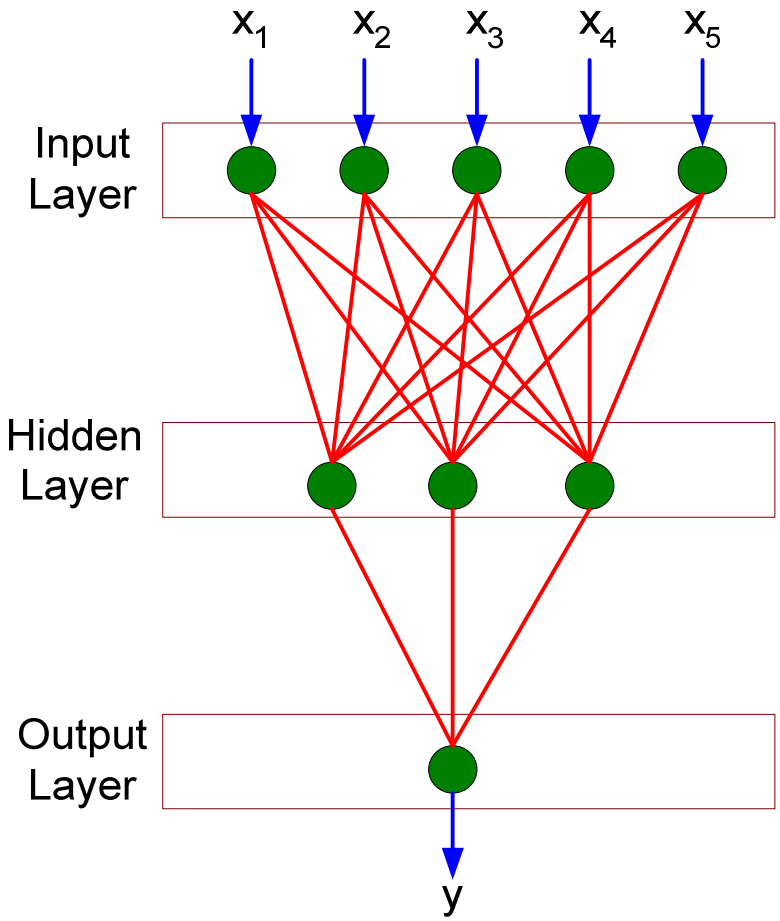
- Can we apply perceptron learning rule to each node, including hidden nodes?
  - Perceptron learning rule computes error term e = y - $\hat{y}$ and updates weights accordingly
    - Problem: how to determine the true value of y for hidden nodes?

$$w_j^{(k+1)} = w_j^{(k)} + \lambda\left(y_i - \hat{y}_i^{(k)}\right)x_{ij}$$

# Learning Multi-layer Neural Network

- Can we apply perceptron learning rule to each node, including hidden nodes?
  - Perceptron learning rule computes error term e = y - $\hat{y}$ and updates weights accordingly
    - Problem: how to determine the true value of y for hidden nodes?
  - Approximate error in hidden nodes by error in the output nodes
    - Problem:
      - Not clear how adjustment in the hidden nodes affect overall error
      - No guarantee of convergence to optimal solution

# Learning Multi-layer Neural Network



# Gradient Descent

- Loss Function to measure errors across all training points

$$E(\mathbf{w}, \mathbf{b}) = \sum_{k=1}^{n} \text{Loss}\ (y_k,\ \hat{y}_k)$$

Squared Loss:

$$\text{Loss}\ (y_k,\ \hat{y}_k) = (y_k - \hat{y}_k)^2$$

- Gradient descent: Update parameters in the direction of "maximum descent" in the loss function across all points

$$w_{ij}^l \longleftarrow w_{ij}^l - \lambda \frac{\partial E}{\partial w_{ij}^l},$$

$$b_i^l \longleftarrow b_i^l - \lambda \frac{\partial E}{\partial b_i^l},$$

$\lambda$: learning rate

- Stochastic gradient descent (SGD): update the weight for every instance (minibatch SGD: update over min-batches of instances)

# Computing Gradients

- $\frac{\partial E}{\partial w_{ij}^l} = \sum_{k=1}^{n} \frac{\partial \text{ Loss } (y_k,\ \hat{y}_k)}{\partial w_{ij}^l}.$

$$\hat{\boldsymbol{y}} = \boldsymbol{a}^{\boldsymbol{L}}$$

$$a_i^l = f(z_i^l) = f\Big(\sum_j w_{ij}^l a_j^{l-1} + b_i^l\Big)$$

- Using chain rule of differentiation (on a single instance):

$$\frac{\partial \text{ Loss}}{\partial w_{ij}^l} = \frac{\partial \text{ Loss}}{\partial a_i^l} \times \frac{\partial a_i^l}{\partial z_i^l} \times \frac{\partial z_i^l}{\partial w_{ij}^l}.$$

- For sigmoid activation function:

$$\frac{\partial \text{ Loss}}{\partial w_{ij}^l} = \delta_i^l \times a_i^l(1 - a_i^l) \times a_j^{l-1},$$

$$\text{where } \delta_i^l = \frac{\partial \text{ Loss}}{\partial a_i^l}.$$

- How can we compute $\delta_i^l$ for every layer?

# Backpropagation Algorithm

- At output layer L:

$$\delta^L = \frac{\partial \text{ Loss}}{\partial a^L} = \frac{\partial \ (y - a^L)^2}{\partial a^L} = 2(a^L - y).$$

- At a hidden layer $l$ (using chain rule):

$$\delta_j^l = \sum_i (\delta_i^{l+1} \times a_i^{l+1}(1 - a_i^{l+1}) \times w_{ij}^{l+1}).$$

  - Gradients at layer l can be computed using gradients at layer l + 1
  - Start from layer L and "backpropagate" gradients to all previous layers

- Use gradient descent to update weights at every epoch
- For next epoch, use updated weights to compute loss fn. and its gradient
- Iterate until convergence (loss does not change)

# Design Issues in ANN

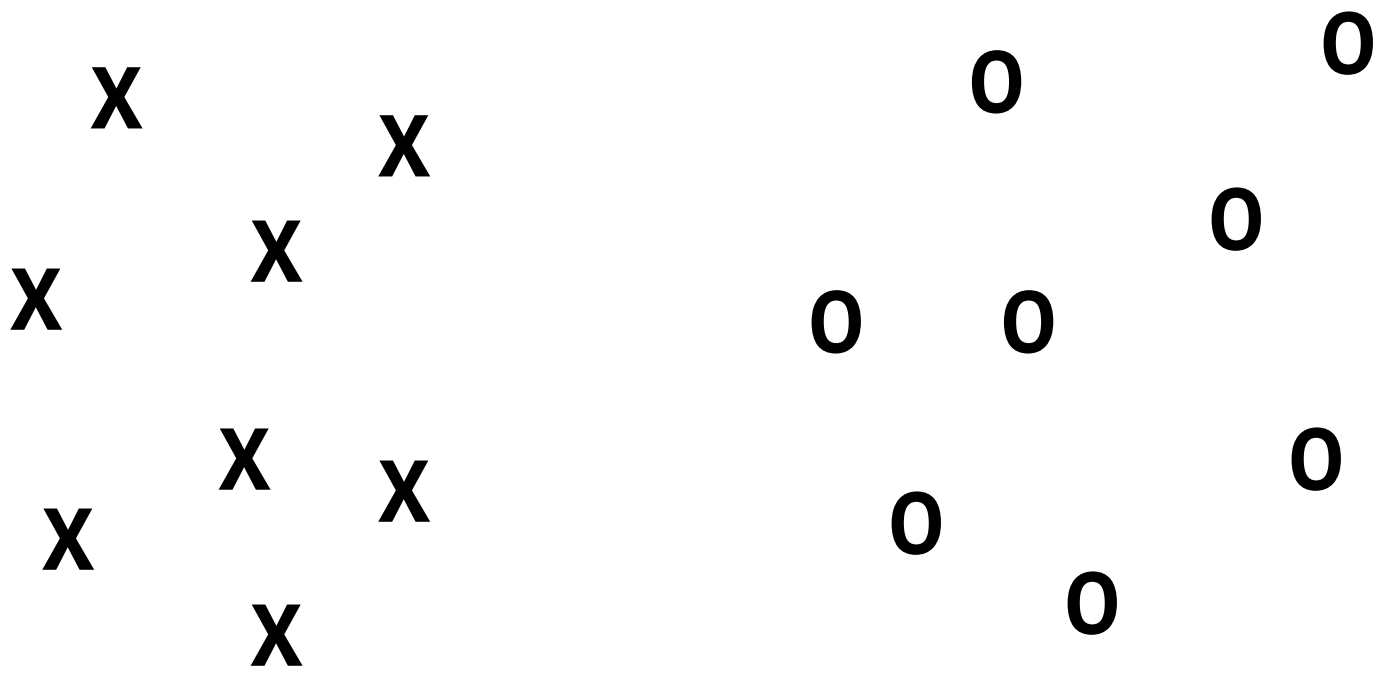
- Number of nodes in input layer
  - One input node per **binary/continuous** attribute
  - k or $\log_2$ k nodes for each **categorical** attribute with k values
- Number of nodes in output layer
  - One output for binary class problem
  - k or $\log_2$ k nodes for k-class problem
- Number of hidden layers and nodes per layer
- Initial weights and biases
- Learning rate, max. number of epochs, mini-batch size for mini-batch SGD, …

# Characteristics of ANN

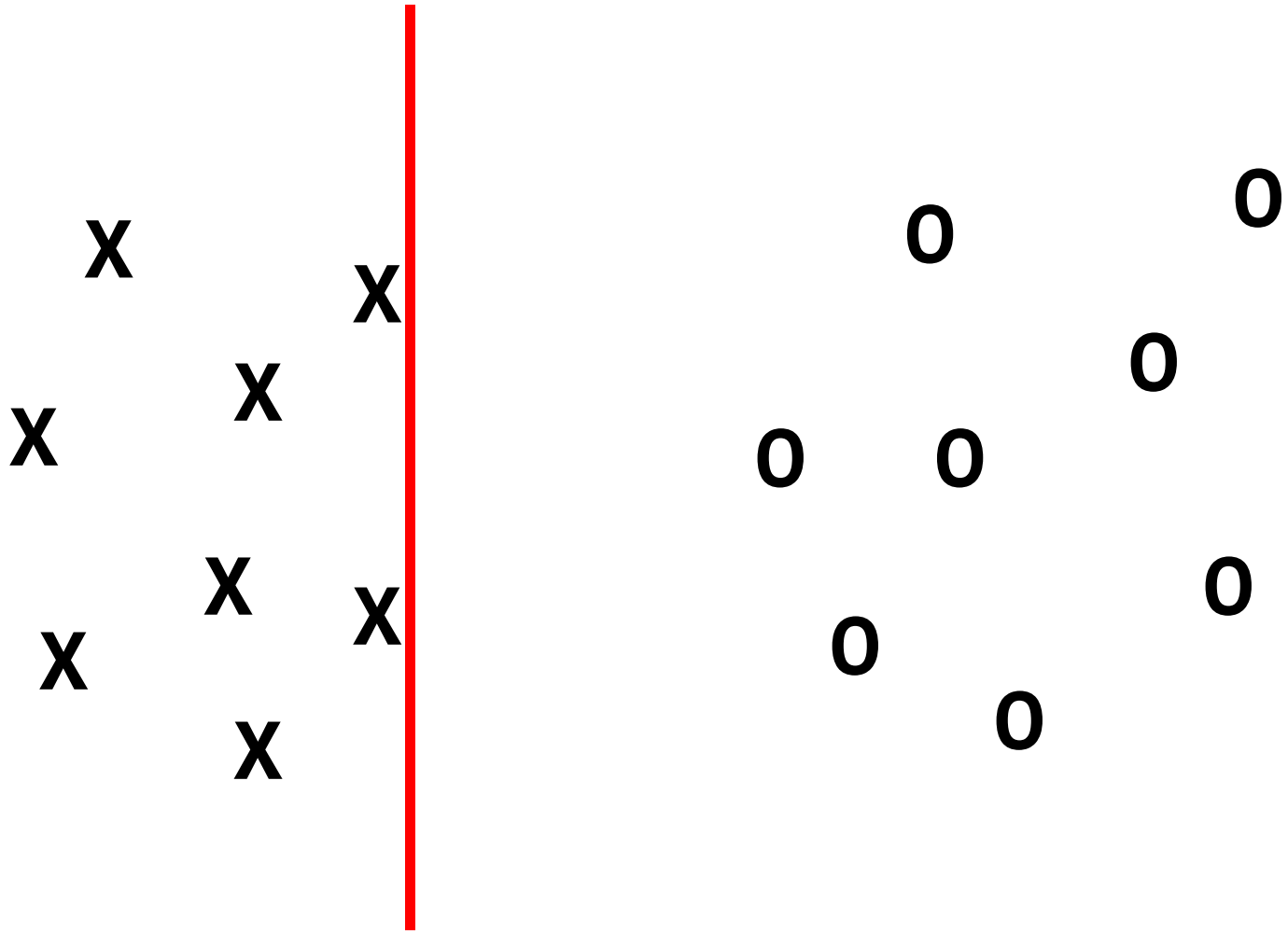
- Multilayer ANN are universal approximators but could suffer from overfitting if the network is too large
  - Naturally represents a hierarchy of features at multiple levels of abstractions
- Gradient descent may converge to local minimum
- Model building is compute intensive, but testing is fast
- Can handle redundant and irrelevant attributes because weights are automatically learnt for all attributes
- Sensitive to noise in training data
  - This issue can be addressed by incorporating model complexity in the loss function
- Difficult to handle missing attributes

# SUPPORT VECTOR MACHINES

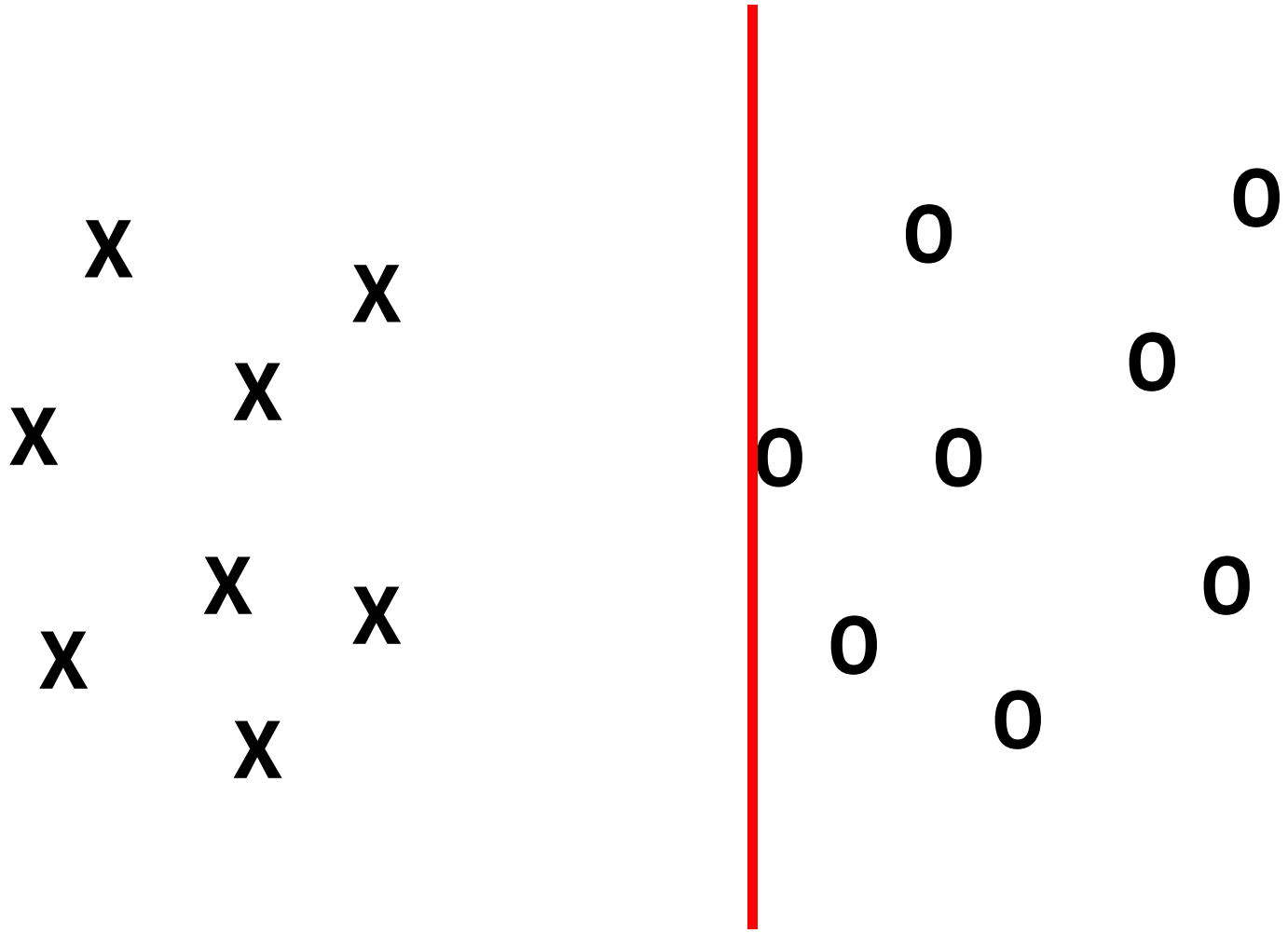
# Intuitions



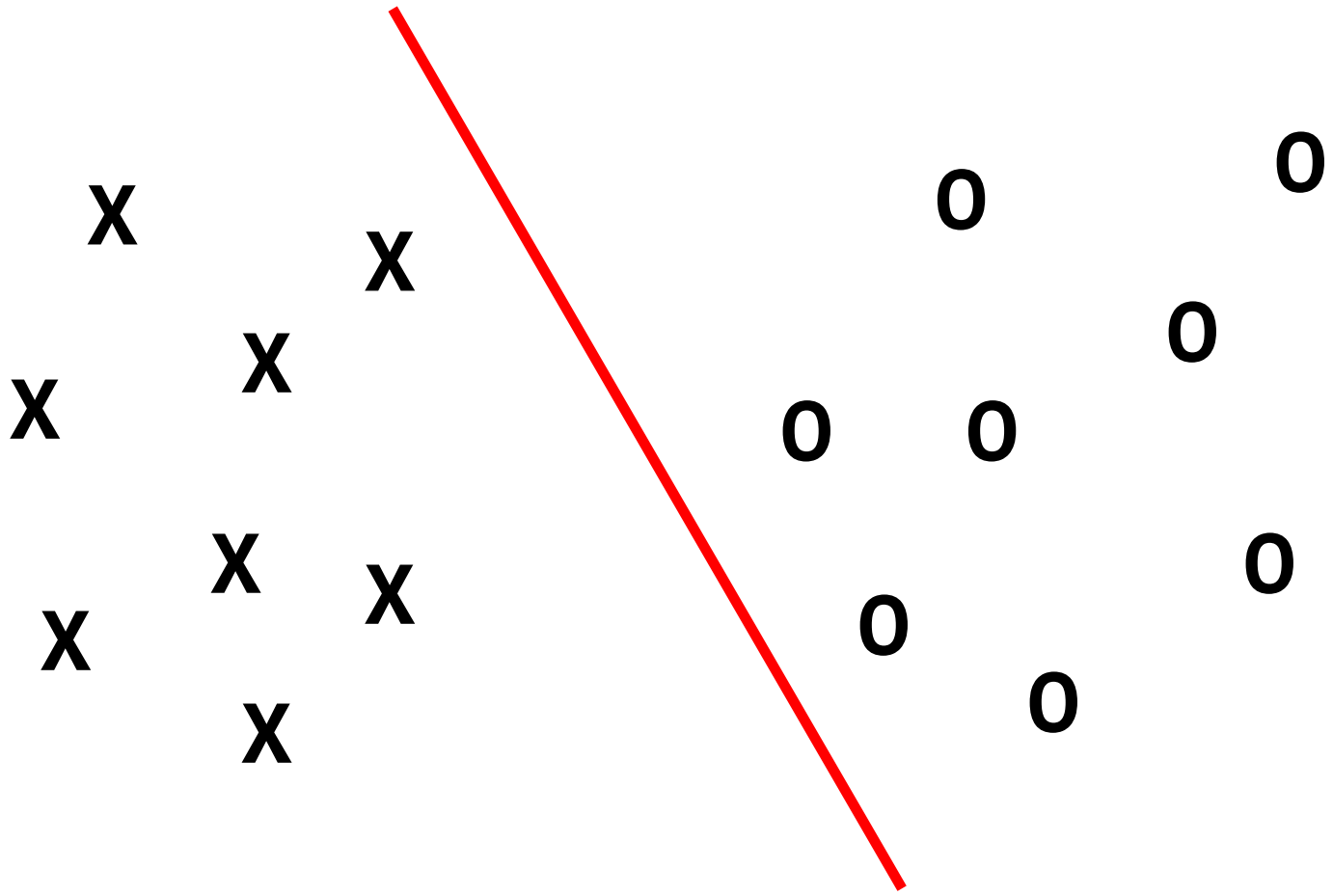
# Intuitions



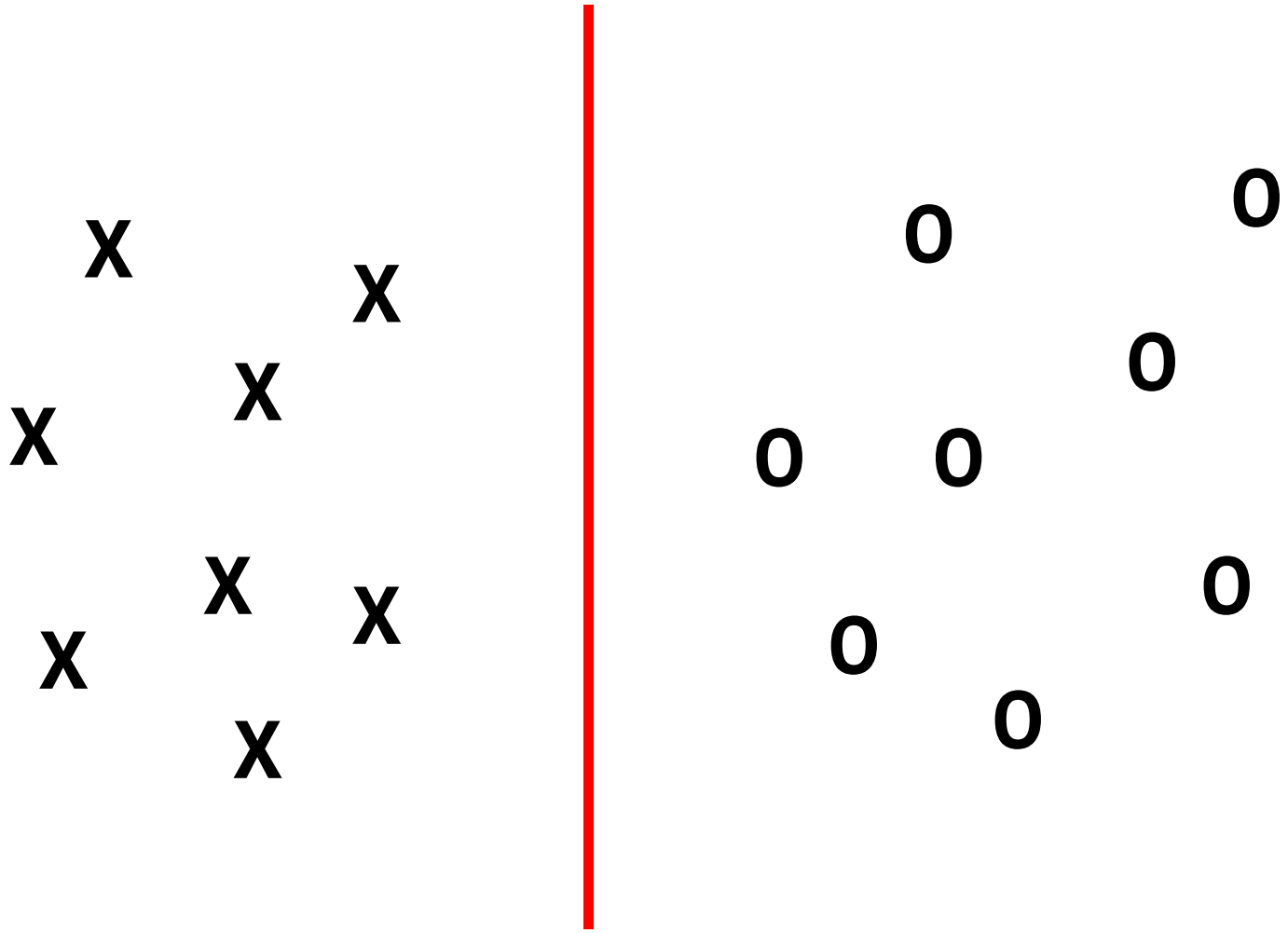
# Intuitions



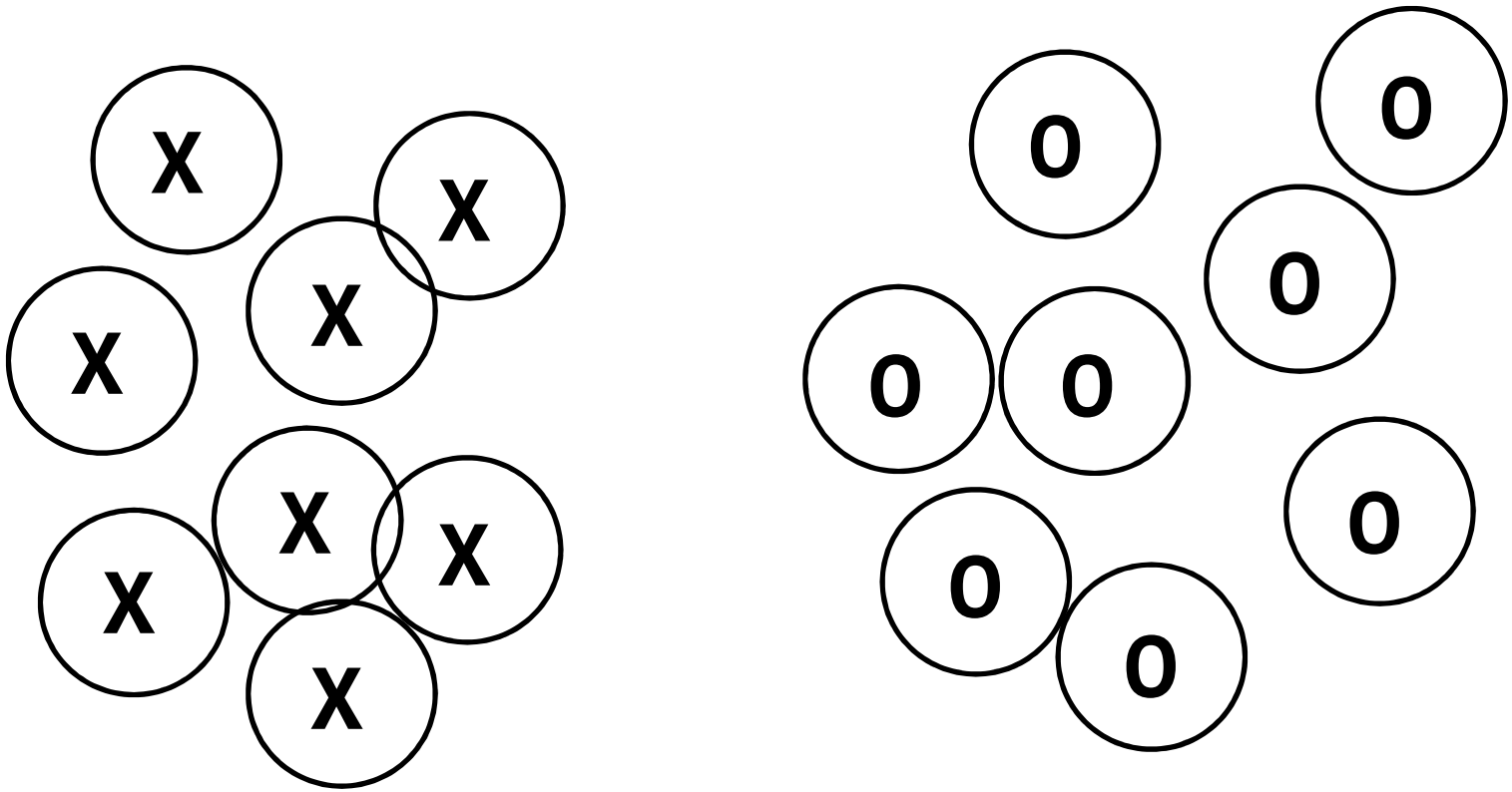
# Intuitions



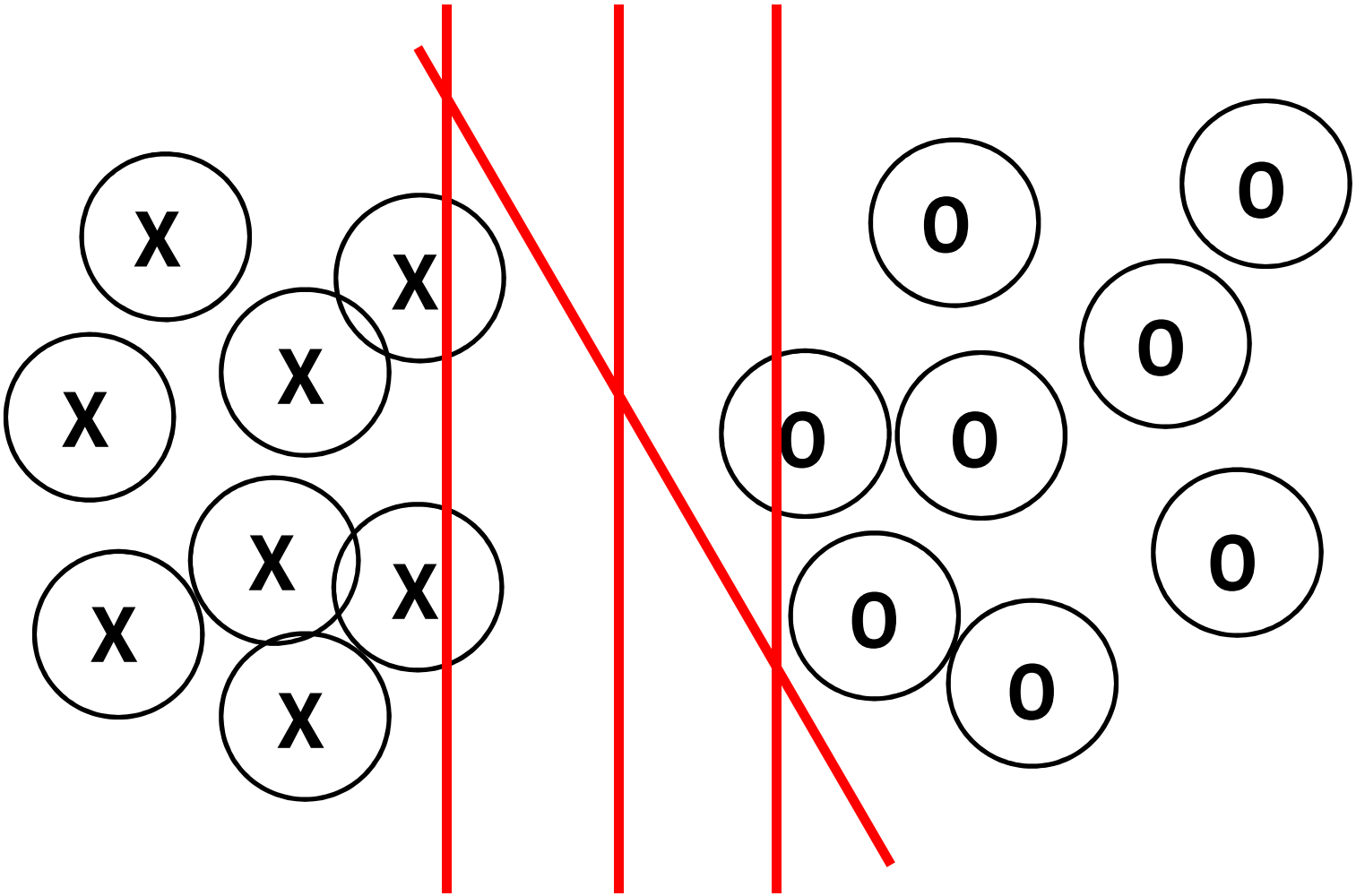
# A "Good" Separator



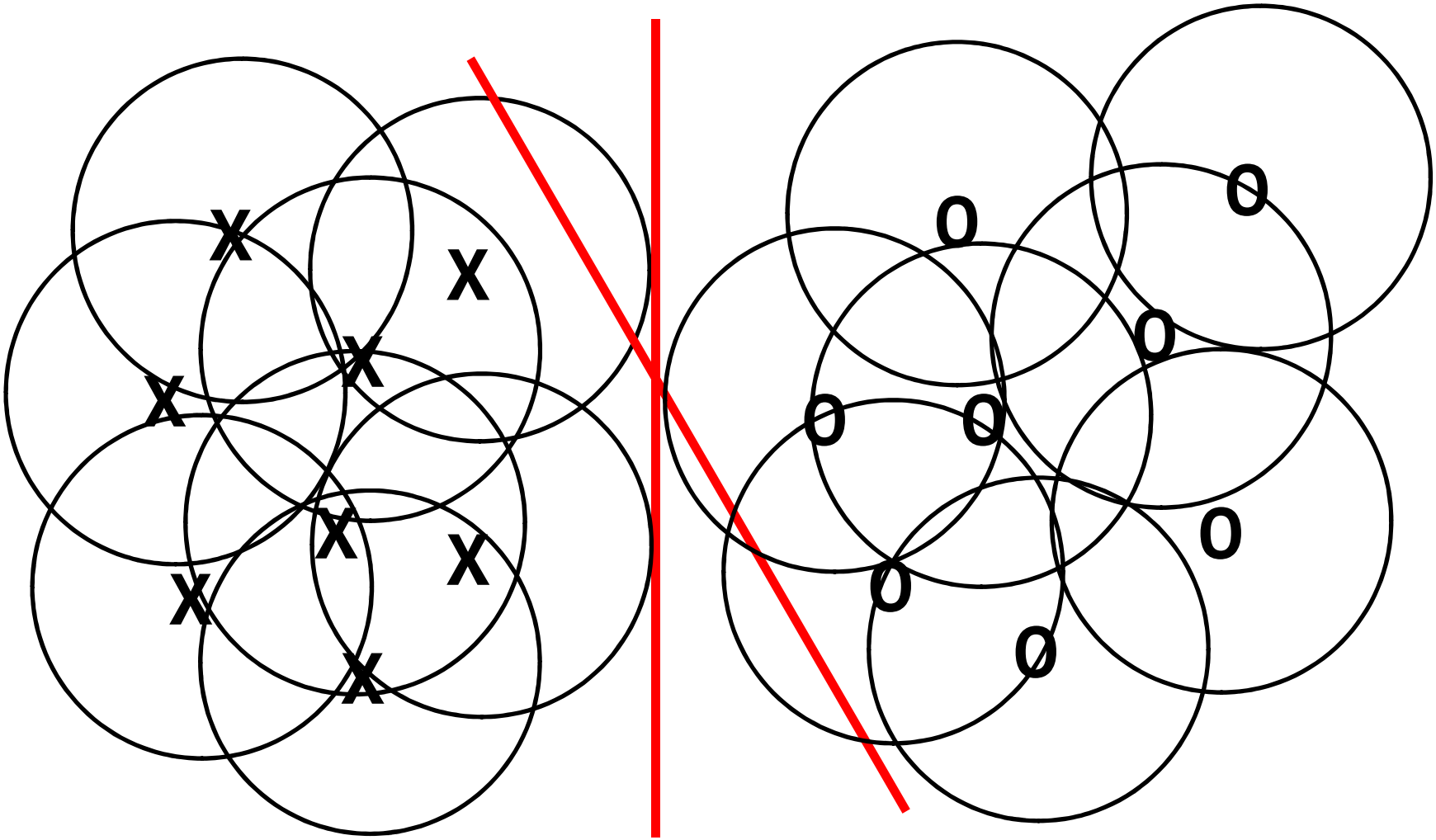
# Noise in the Observations



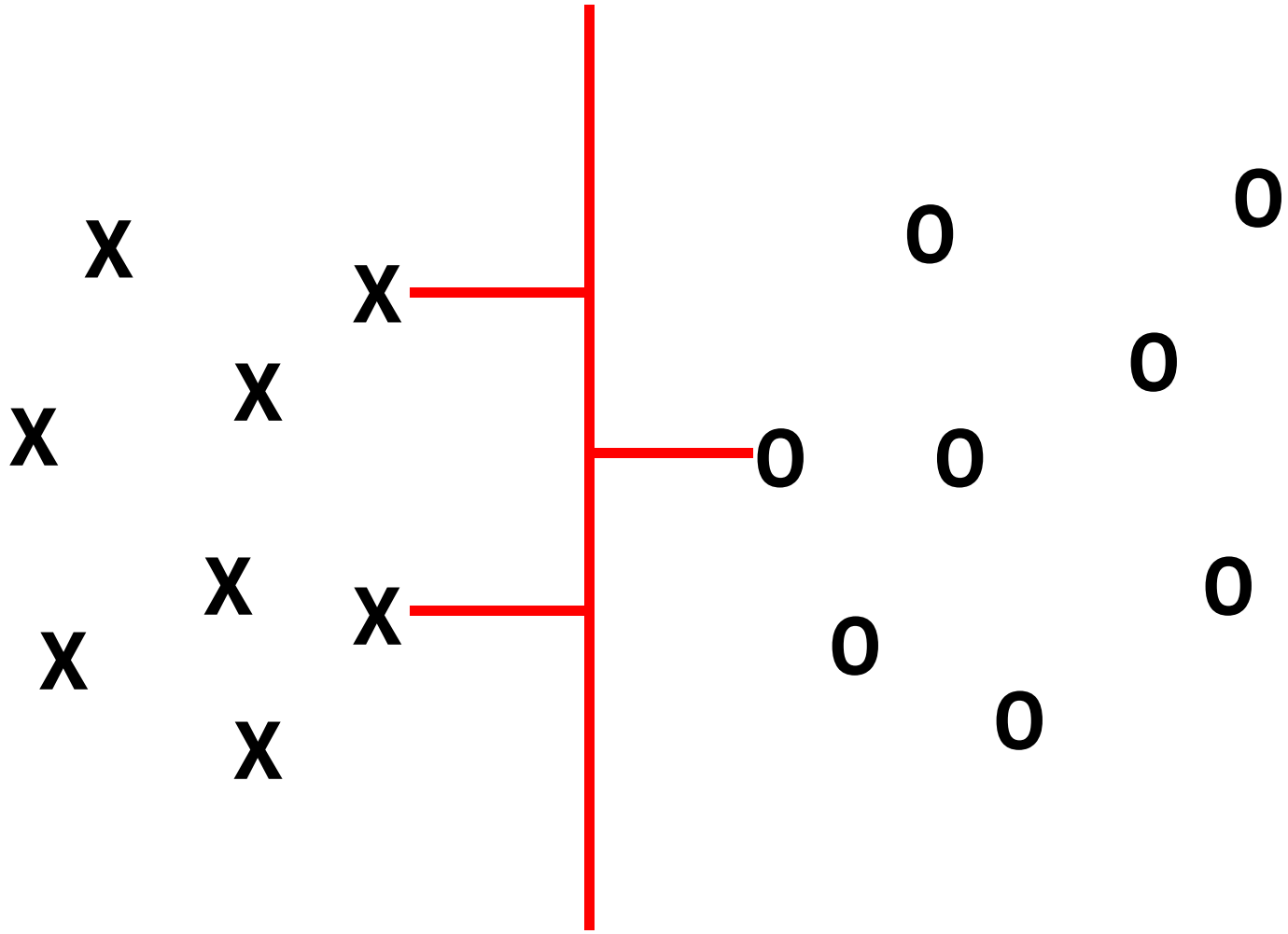
# Ruling Out Some Separators



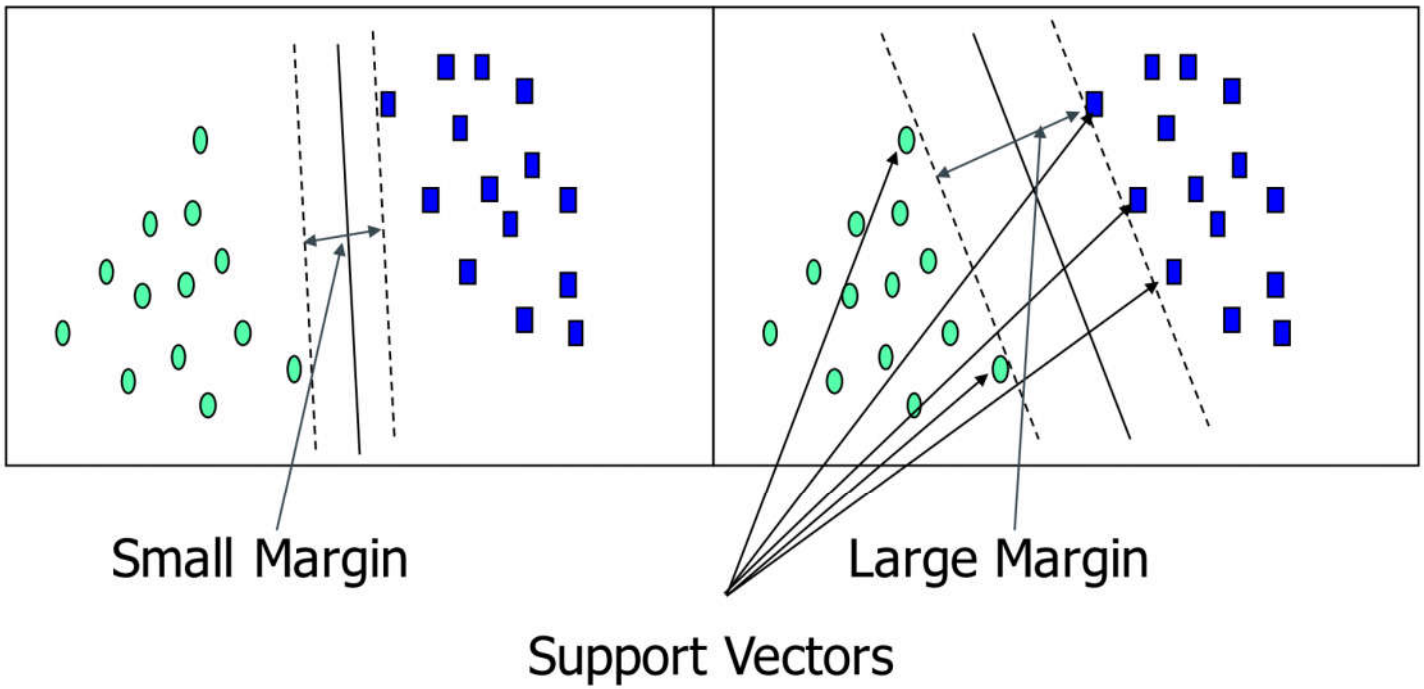
# Lots of Noise



# Maximizing the Margin

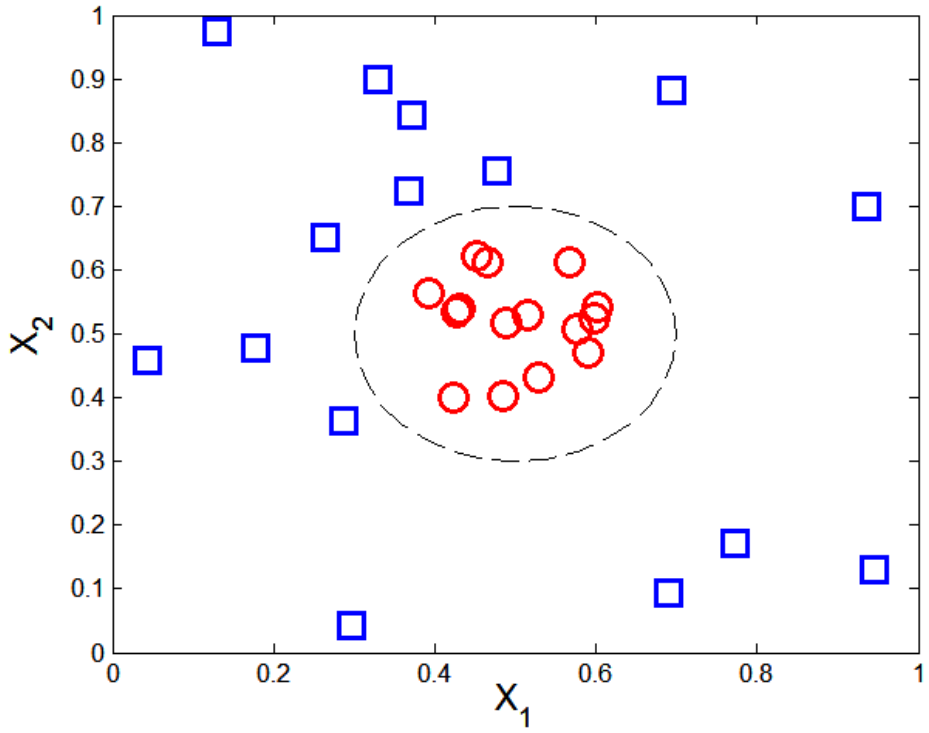


# Maximizing the Margin



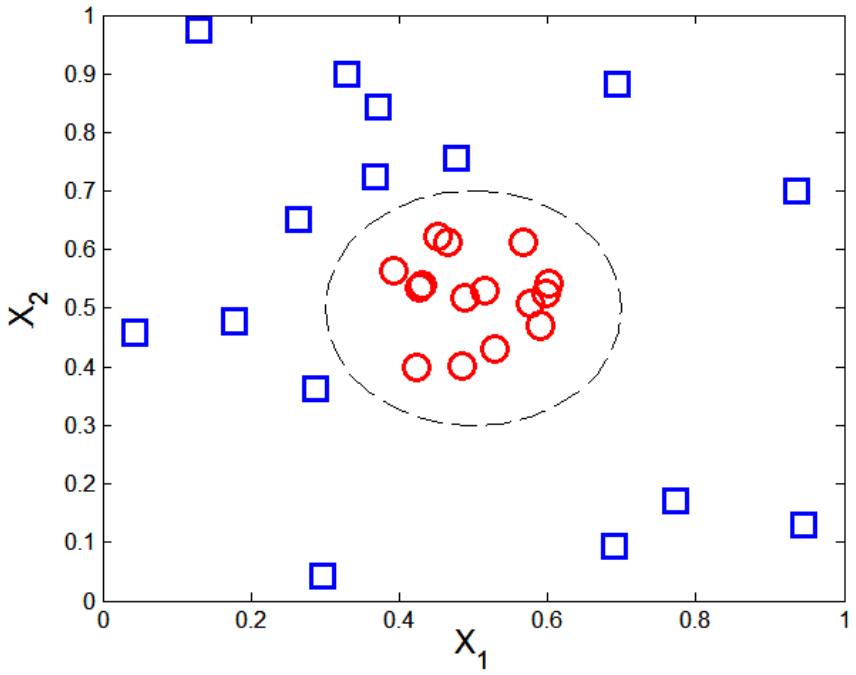
# Nonlinear Support Vector Machines

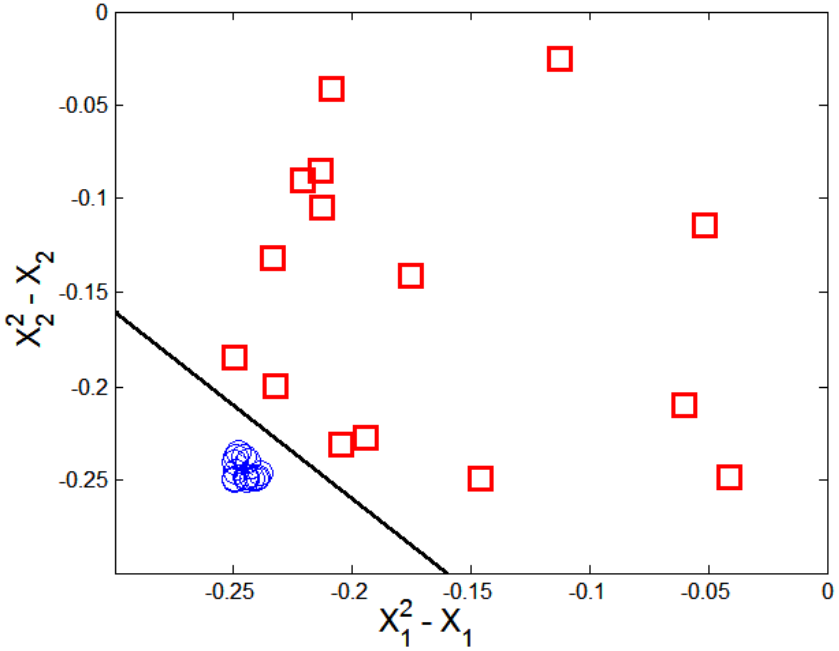
- What if decision boundary is not linear?



# Nonlinear Support Vector Machines

- What if decision boundary is not linear?




Kernel **Trick**

**Decision boundary:**

$$\vec{w} \bullet \Phi(\vec{x}) + b = 0$$

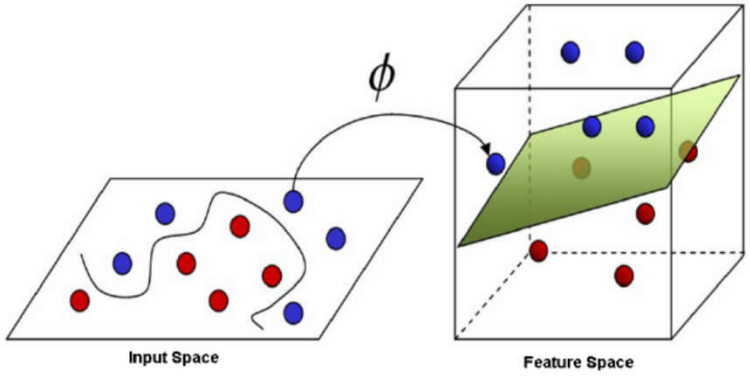
# Learning Nonlinear SVM

- Kernel Trick:
  - $\Phi(x_i) \bullet \Phi(x_j) = K(x_i, x_j)$
  - $K(x_i, x_j)$ is a kernel function (expressed in terms of the coordinates in the original space)
    - Examples:

$$K(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot \mathbf{y} + 1)^p$$

$$K(\mathbf{x}, \mathbf{y}) = e^{-\|\mathbf{x}-\mathbf{y}\|^2/(2\sigma^2)}$$

$$K(\mathbf{x}, \mathbf{y}) = \tanh(k\mathbf{x} \cdot \mathbf{y} - \delta)$$




http://thecaffeinedev.com/2-support-vector-machine-learning-math-behind-part2/

# Characteristics of SVM

- Robust to noise
- Overfitting is handled by maximizing the margin of the decision boundary,
- SVM can handle irrelevant and redundant attributes better than many other techniques
- The user needs to provide the type of kernel function and cost function
- Difficult to handle missing values
- What about categorical variables?

ویژگی های SVM

مقاوم در برابر نویز

تطبیق بیش از حد با به حداکثر رساندن حاشیه مرز تصمیم انجام می شود،

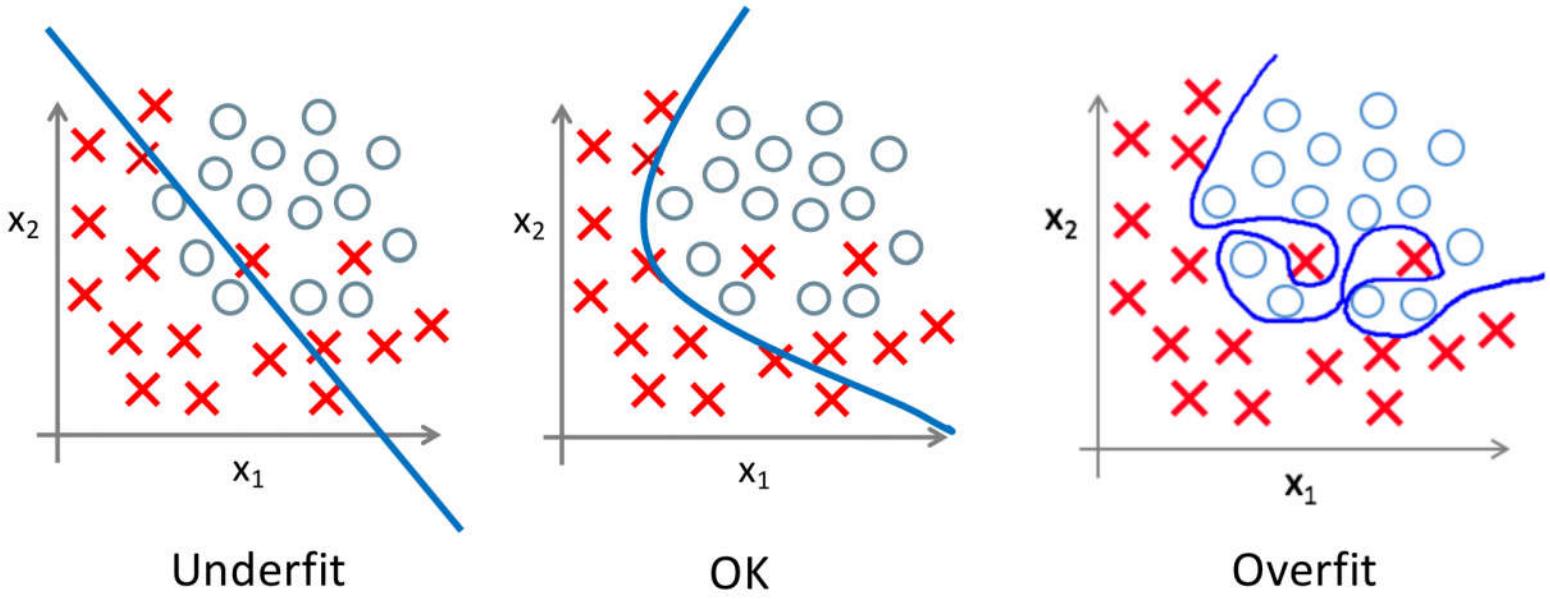
SVM می تواند ویژگی های نامربوط و زائد را بهتر از بسیاری از تکنیک های دیگر مدیریت کند

کاربر باید نوع عملکرد هسته و تابع هزینه را ارائه دهد

رسیدگی به مقادیر از دست رفته مشکل است

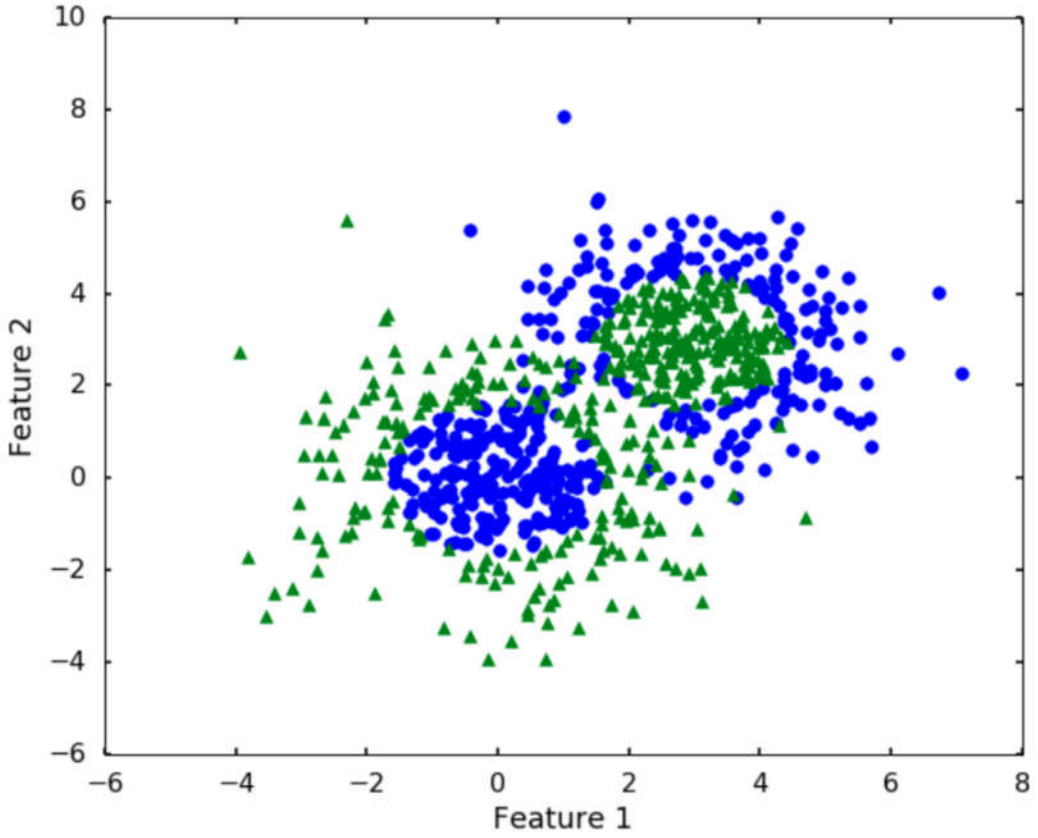
در مورد متغیرهای طبقه بندی چطور؟

# SVM




**In sklearn, this is controlled by C parameter of SVM and gamma parameter of rbf kernel**

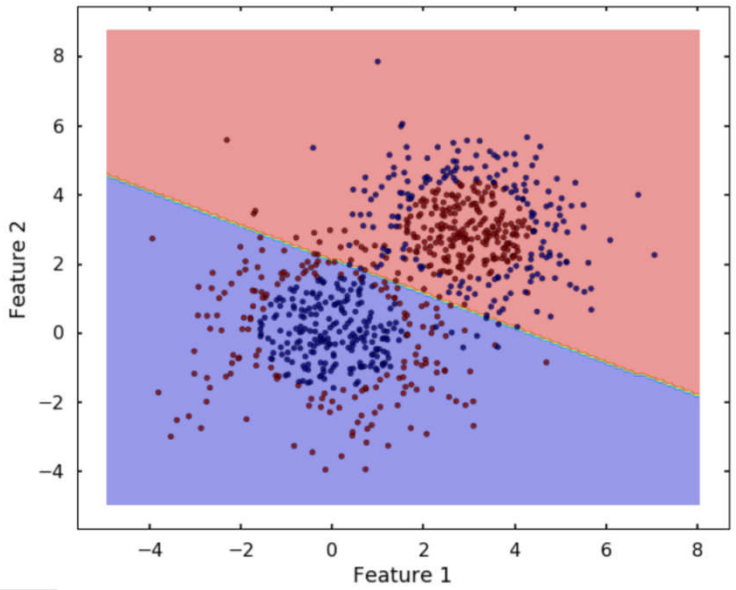
# Python SVM



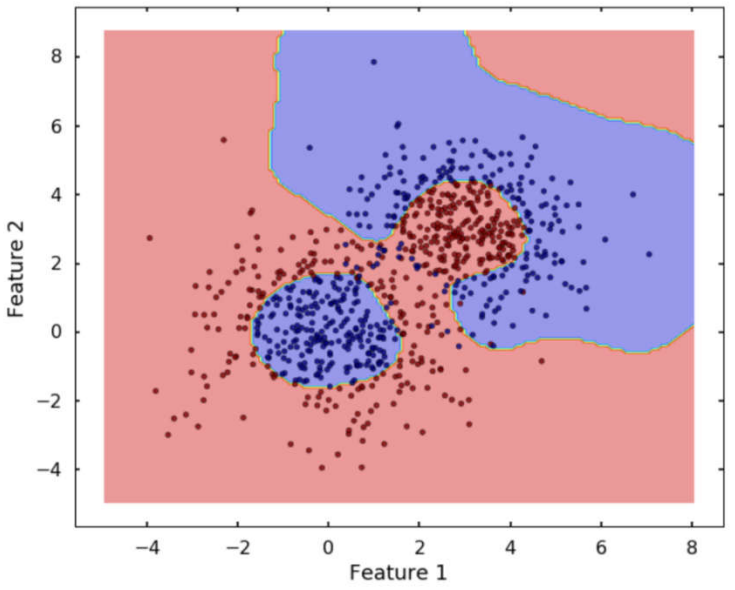

http://qingkaikong.blogspot.com/2016/12/machine-learning-8-support-vector.html

# Python SVM

```
clf = svm.SVC(kernel='linear')
clf.fit(X,y)
```



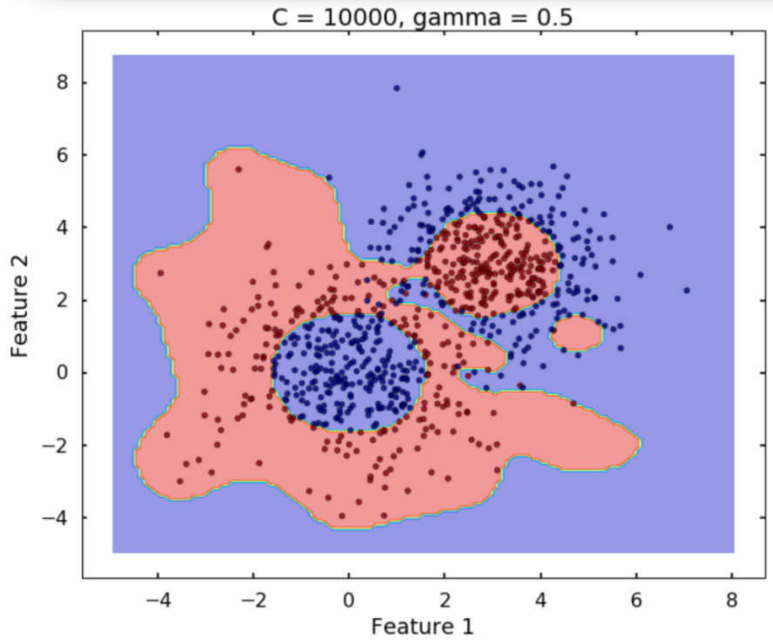

```
clf = svm.SVC(kernel='rbf')
clf.fit(X,y)
```



# Python SVM

- A large C makes the cost of misclassification high.
- This will force the algorithm to fit the data with more flexible model.

```
clf = svm.SVC(kernel='rbf', C = 10000, gamma = 0.5)
clf.fit(X,y)
```

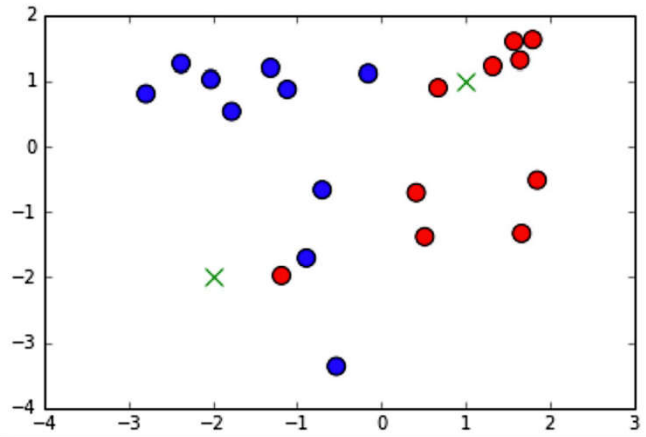



C بزرگ هزینه طبقه بندی اشتباه را بالا می برد.
این الگوریتم را مجبور می کند که داده ها را با مدل انعطاف پذیرتر مطابقت دهد

# Python KNN

```
from sklearn.datasets import make_classification
X, y = make_classification(n_features=2, n_redundant=0,
                           n_informative=2, n_samples=20)
```

```
plt.scatter(X[:,0], X[:,1], cmap='bwr', s=100, c=y)
plt.scatter([-2,1], [-2,1], marker='x',s=100, c='g')
plt.show()
```



# Python KNN

```
from sklearn.neighbors import KNeighborsClassifier

clf = KNeighborsClassifier(n_neighbors=1)
clf.fit(X,y)
print('With k=1: ', clf.predict([[-2,-2], [1,1]]))

clf = KNeighborsClassifier(n_neighbors=3)
clf.fit(X,y)
print('With k=3: ', clf.predict([[-2,-2], [1,1]]))
```

```
With k=1:  [1 1]
With k=3:  [0 1]
```

# Project

- Select your own research problem and justify its importance
- Come up with your hypothesis and find some datasets for verification
- Design your own models or try a large variety of existing models
- Write a 4 to 8 pages report (research-paper like)
- Submit your codes.
- Present the your work.