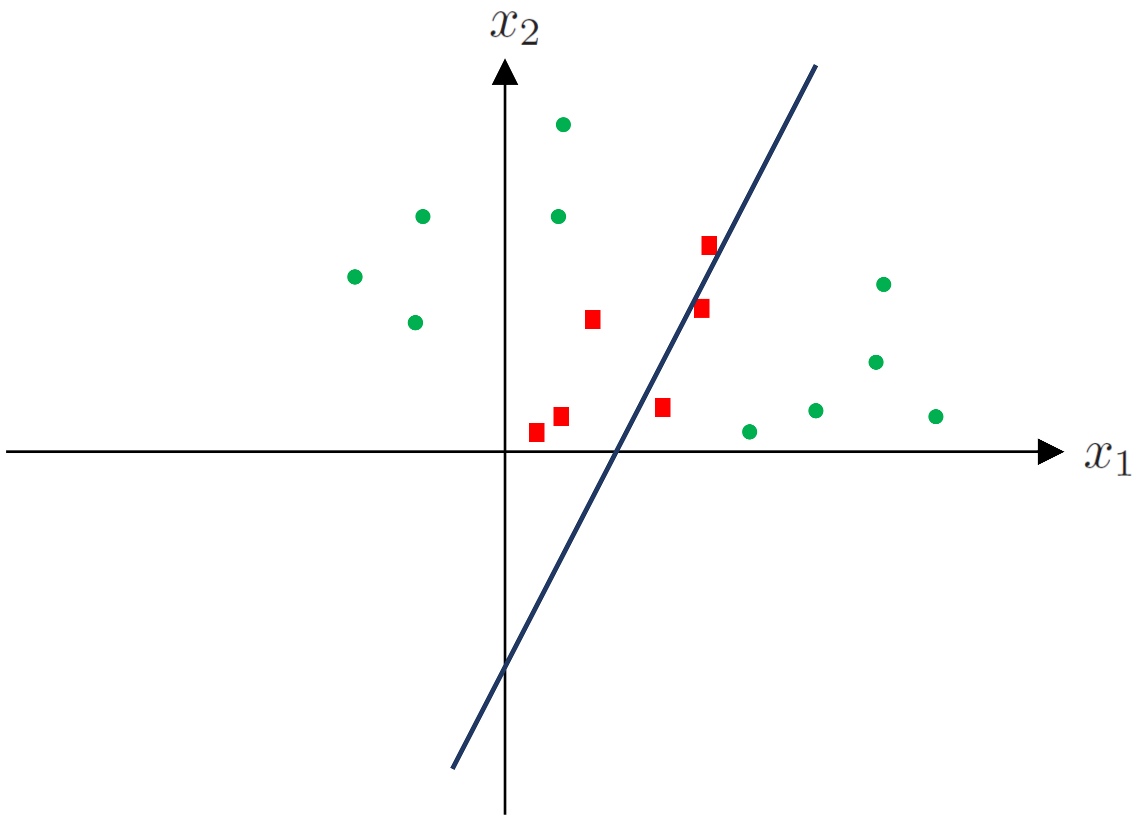
Introduction to Deep Learning: Multi-layer Perceptron

Dripta Mj
Department of Mathematics
Ramakrishna Mission Vivekananda Educational and Research Institute
Belur Math, India

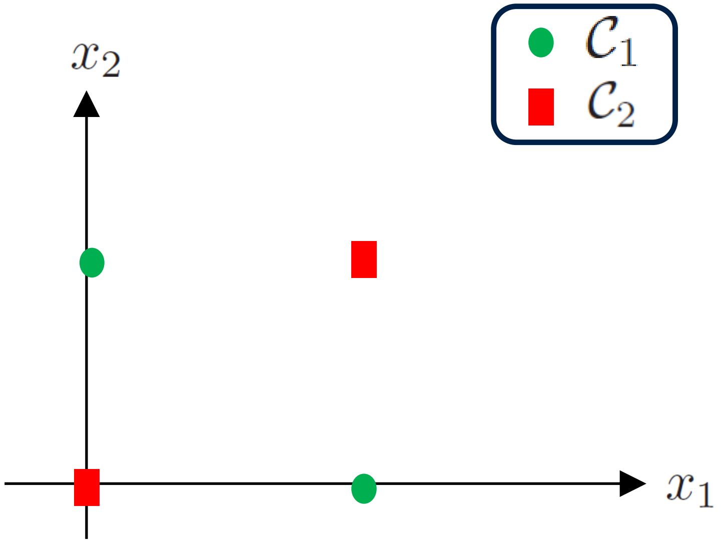
Shortcomings of single layer perceptron Learning



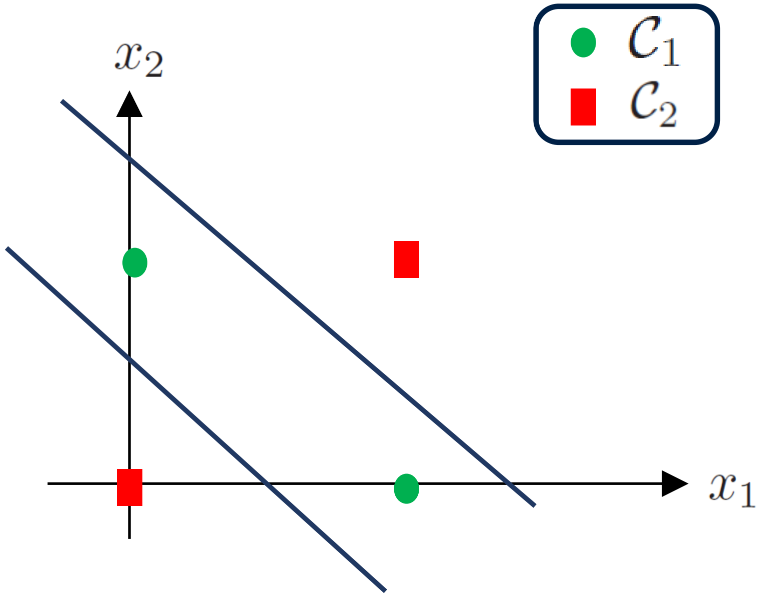
XOR function

- XOR data is not linearly separable.

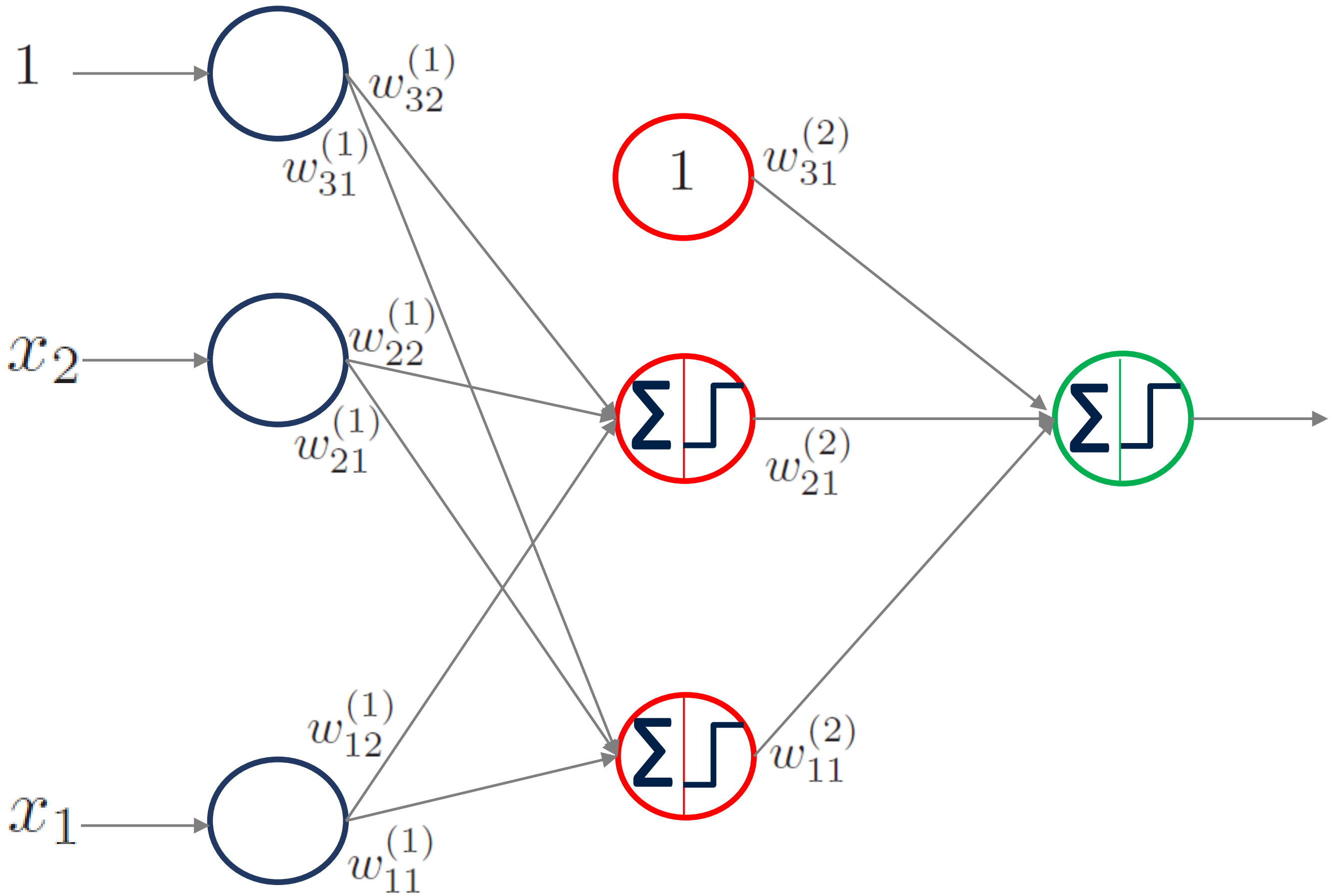
x_1	x_2	XOR	Class label
0	0	0	$\mathcal{C}_2$
0	1	1	$\mathcal{C}_1$
1	0	1	$\mathcal{C}_1$
1	1	0	$\mathcal{C}_2$



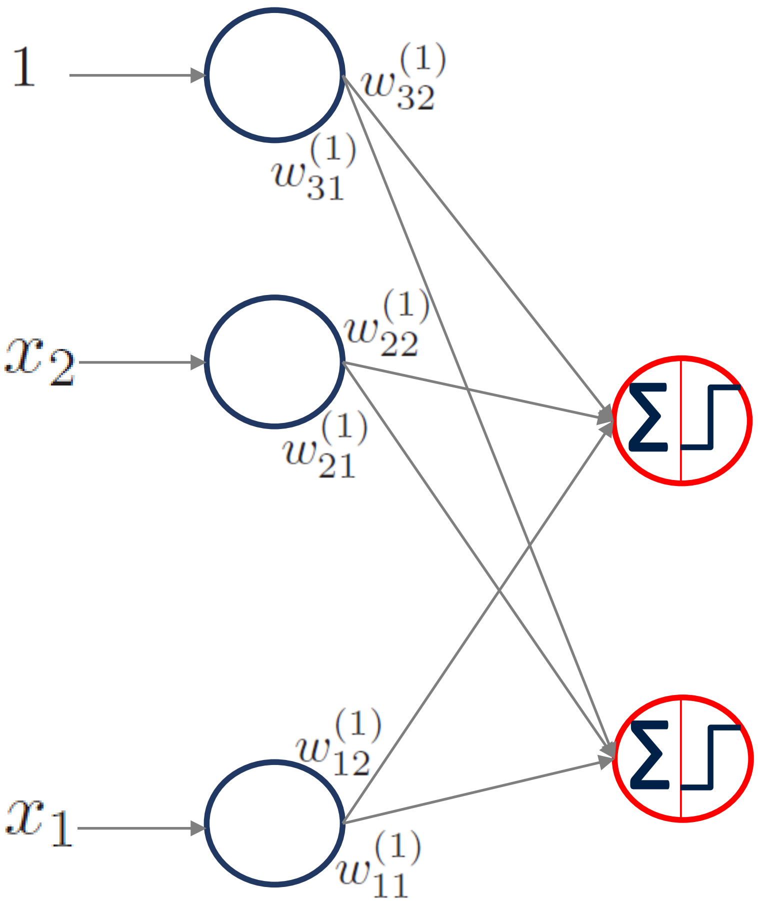
Combination of classifiers



Multi-layer perceptron



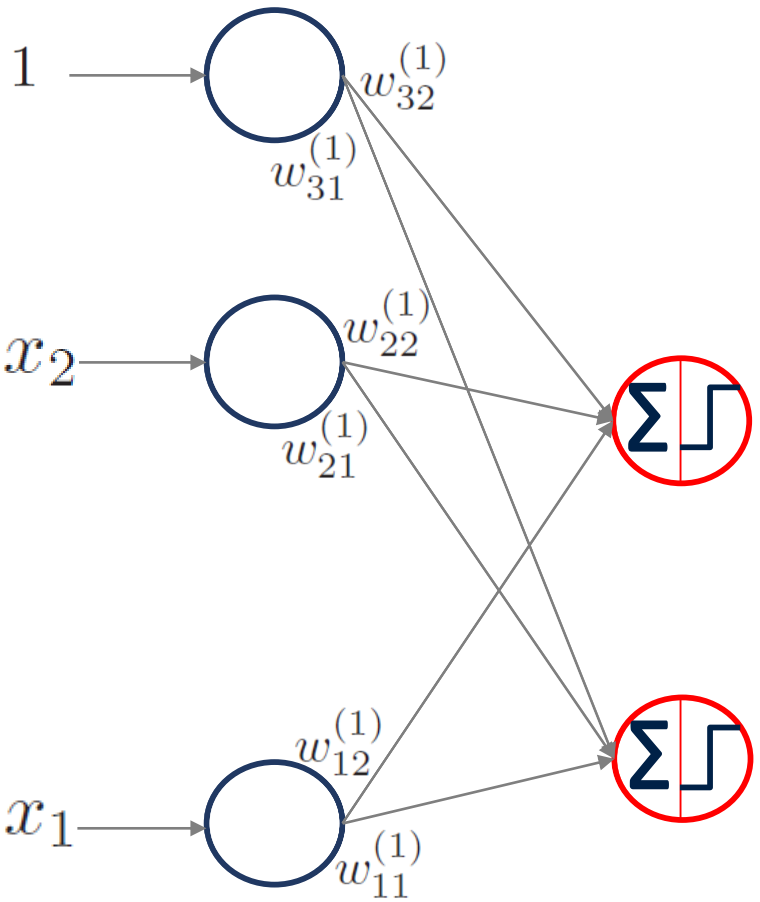
First layer

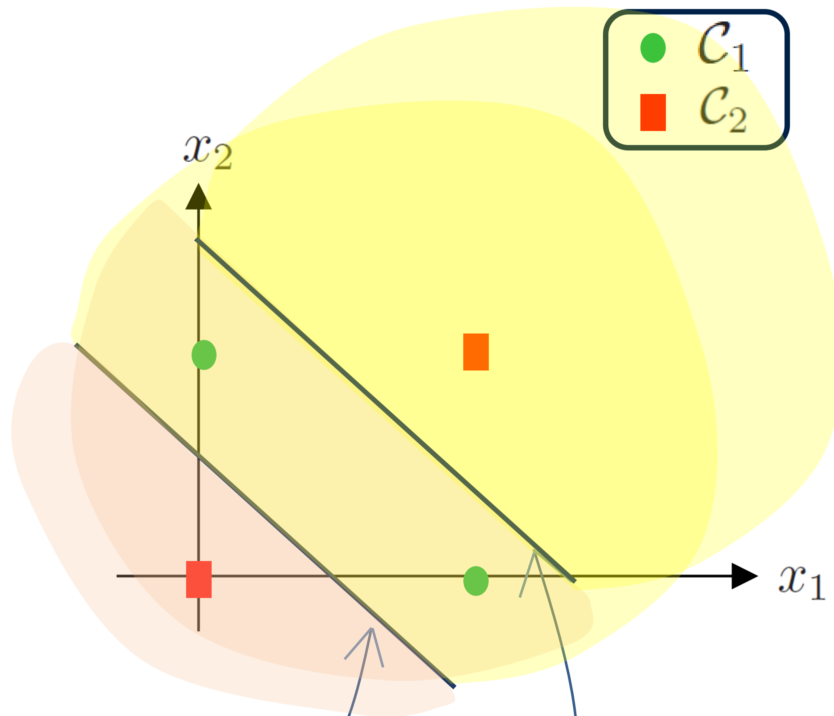


$$z_2 = \mathcal{A}(w_{12}^{(1)} x_1 + w_{22}^{(1)} x_2 + w_{32}^{(1)})$$

$$z_1 = \mathcal{A}(w_{11}^{(1)} x_1 + w_{21}^{(1)} x_2 + w_{31}^{(1)})$$

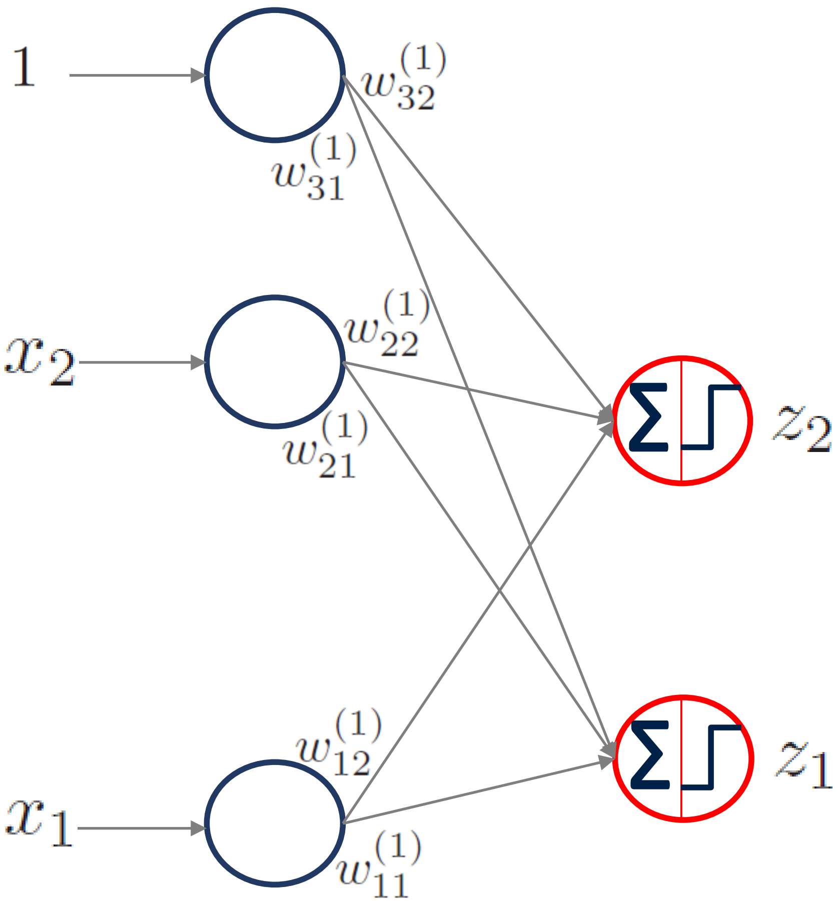
First layer: geometry

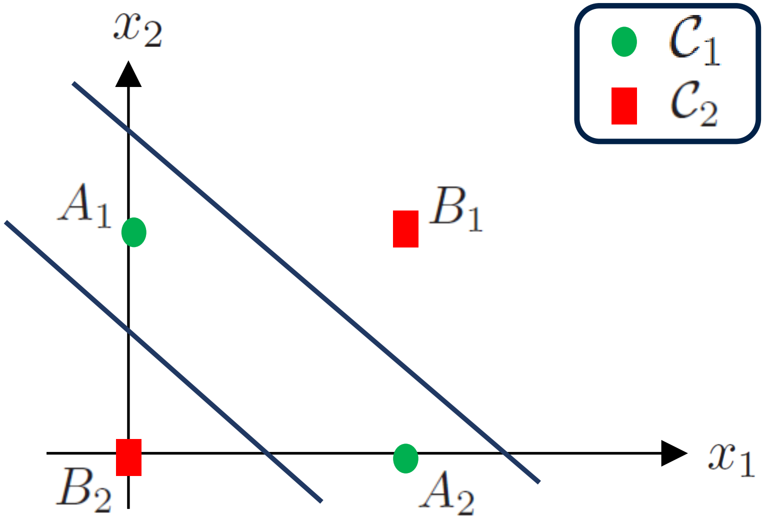


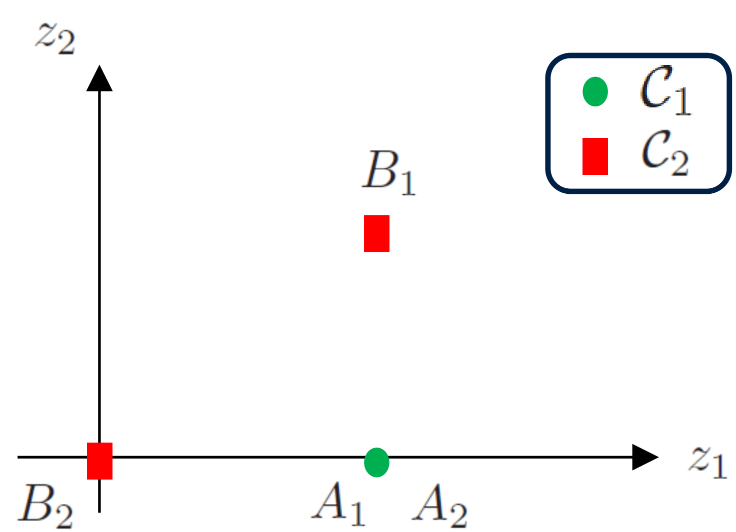


- Take $w_{11}^{(1)} = 1$, $w_{21}^{(1)} = 1$, and $w_{31}^{(1)} = -0.5$.
- Take $w_{12}^{(1)} = 1$, $w_{22}^{(1)} = 1$, and $w_{32}^{(1)} = -1.5$.

First layer: geometry

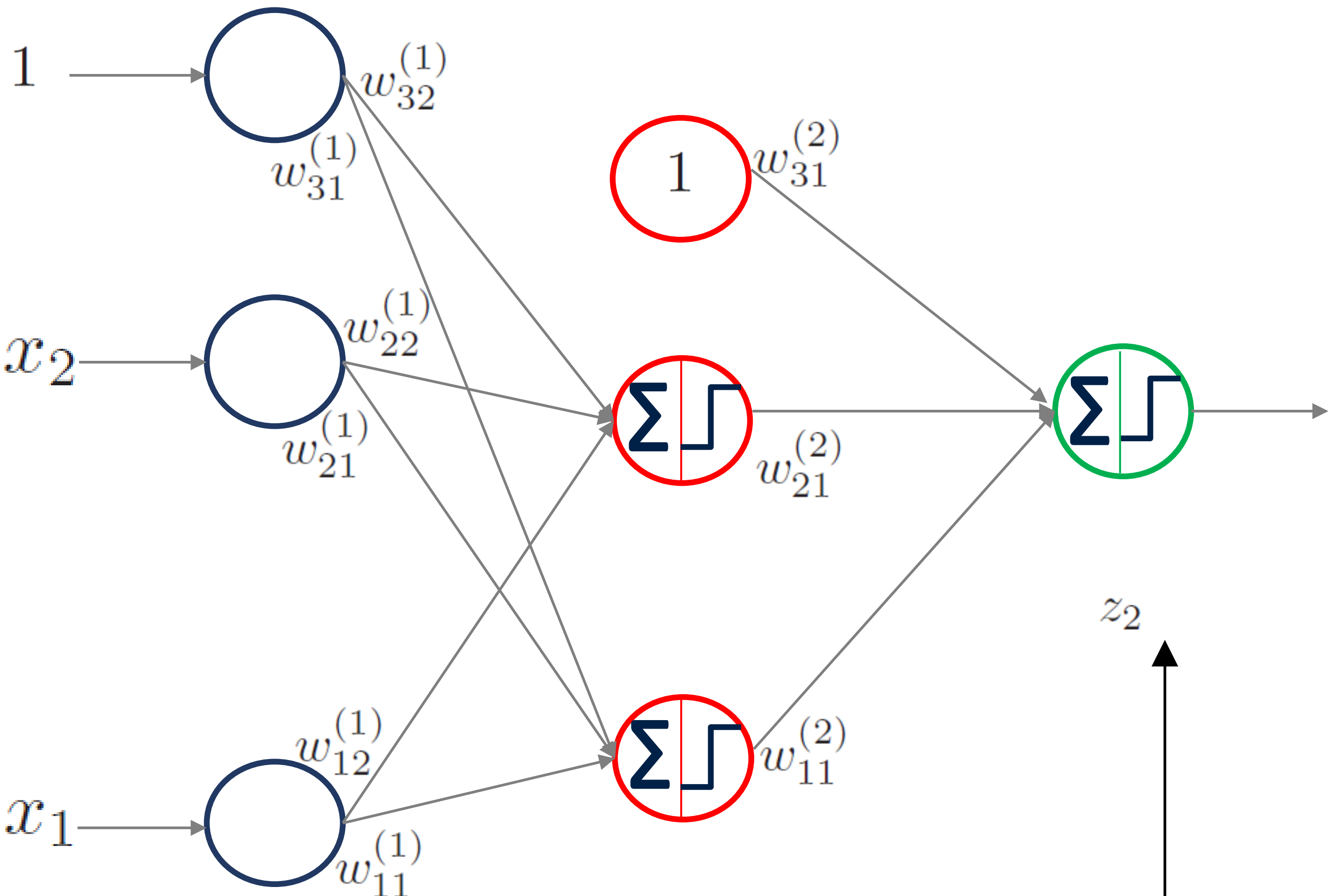


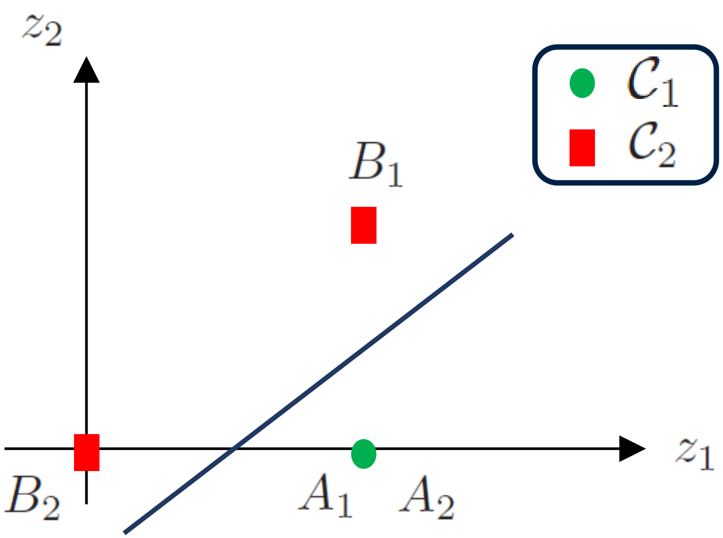




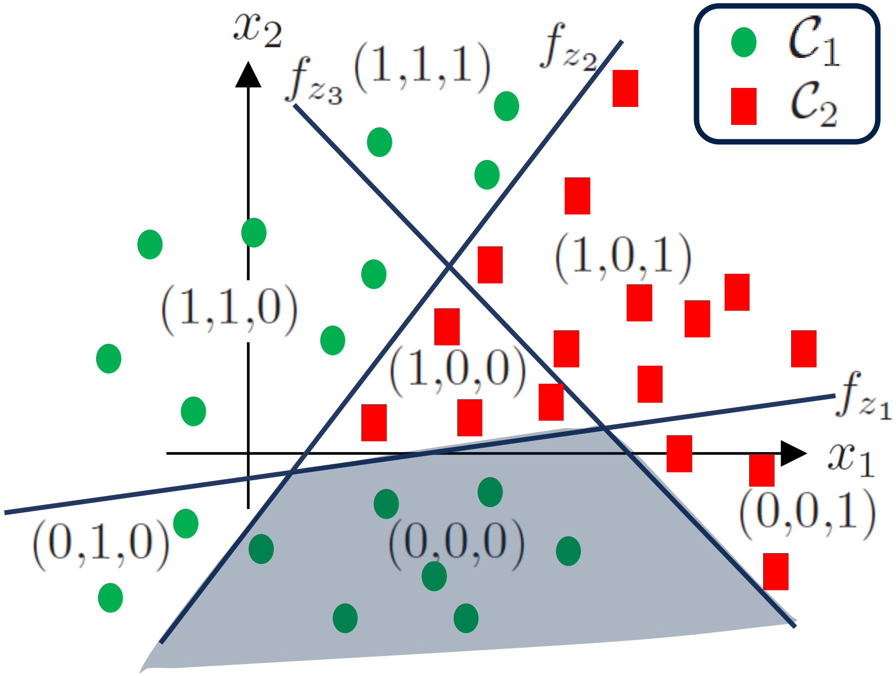
- Now what do you think you need to do?

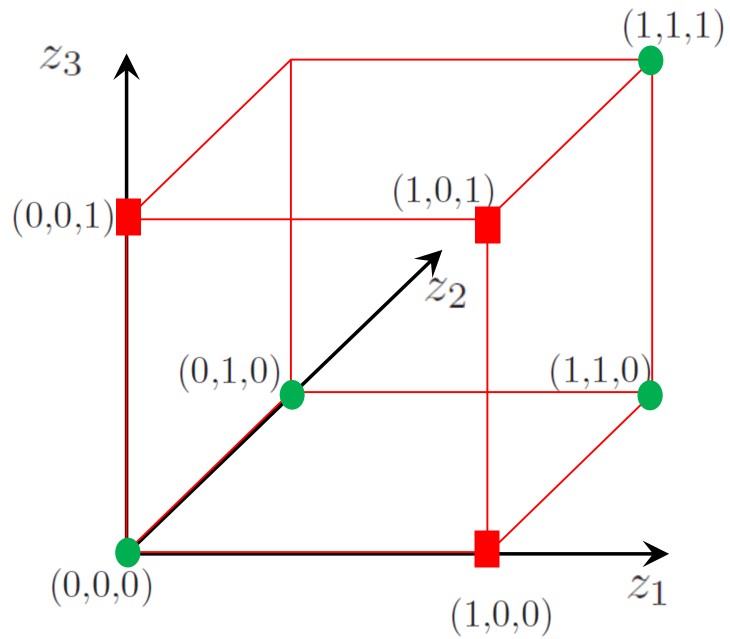
Second layer: geometry

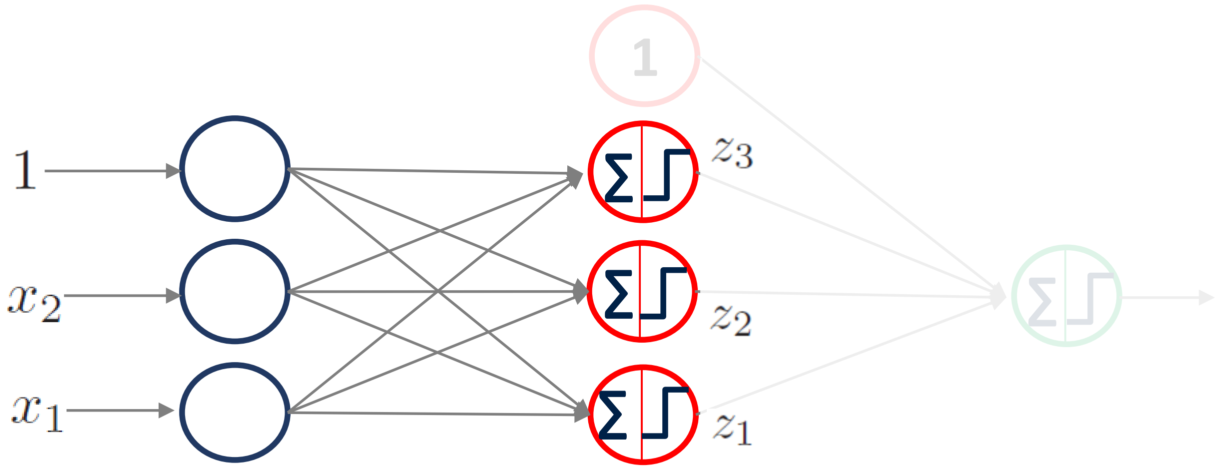




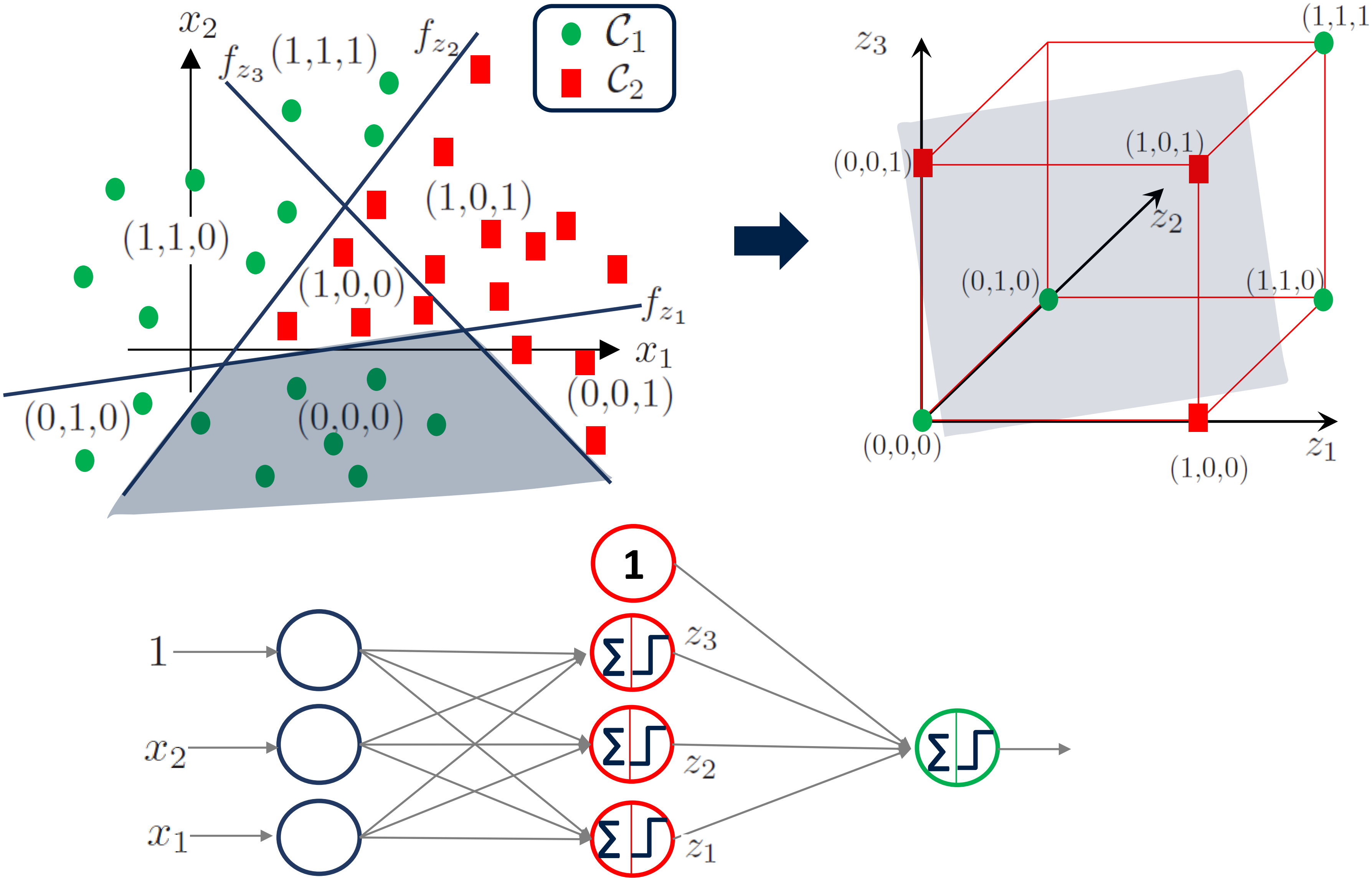
Single hidden layer







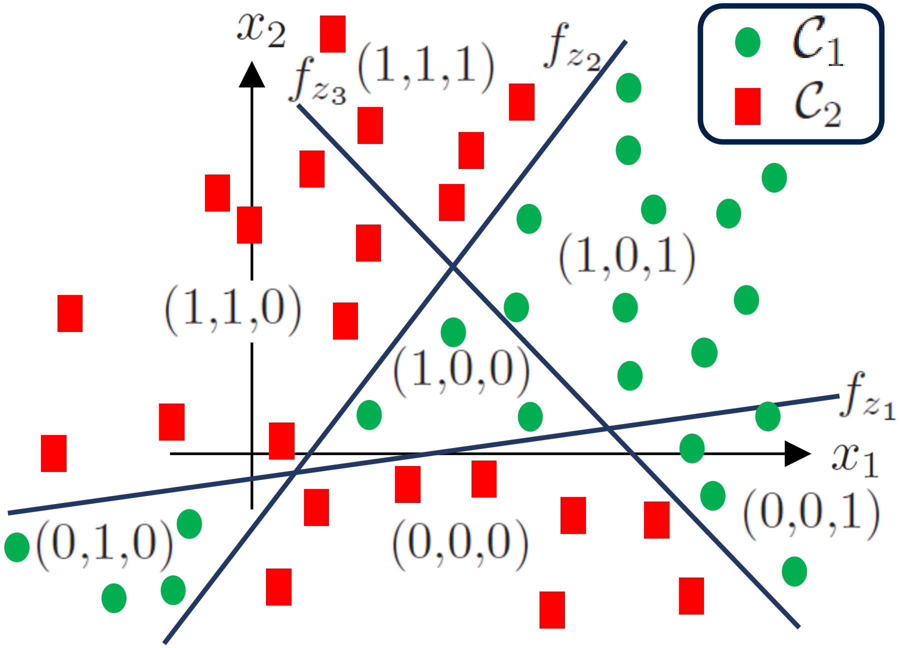
Single hidden layer

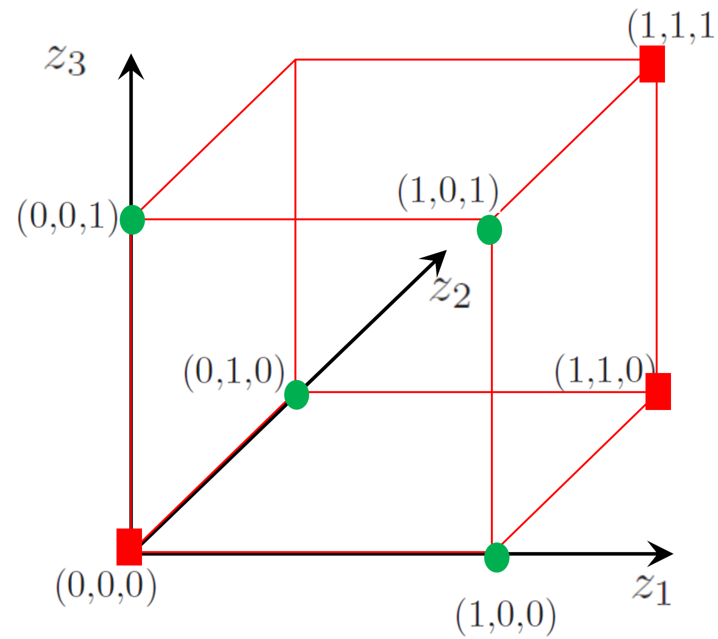


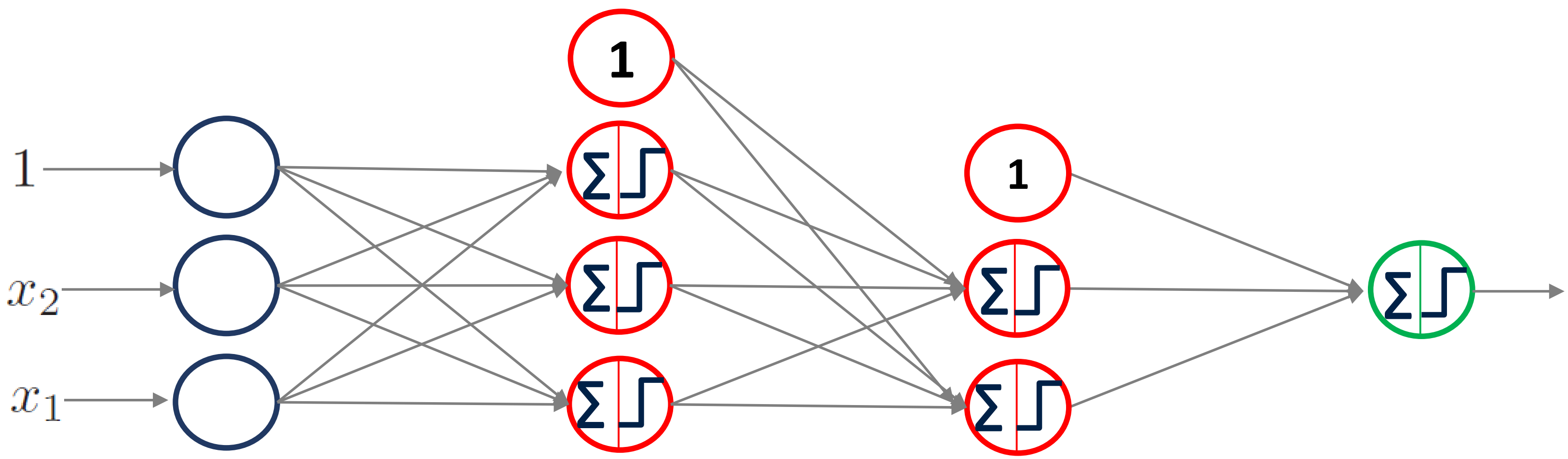
Limitation of single hidden layer

- A network with a single hidden layer can classify data points into classes comprising union of regions.
  - In the previous example, the class $\mathcal{C}_1$ comprised regions $(1, 1, 1)$, $(1, 1, 0)$, $(0, 1, 0)$ and $(0, 0, 0)$.
- But the hidden layer (in the last example) cannot generate classes with any arbitrary union of regions.
- How to adapt to more complex decision boundaries with the same hidden layer structure?
- Solution: Increase the number of hidden layers.

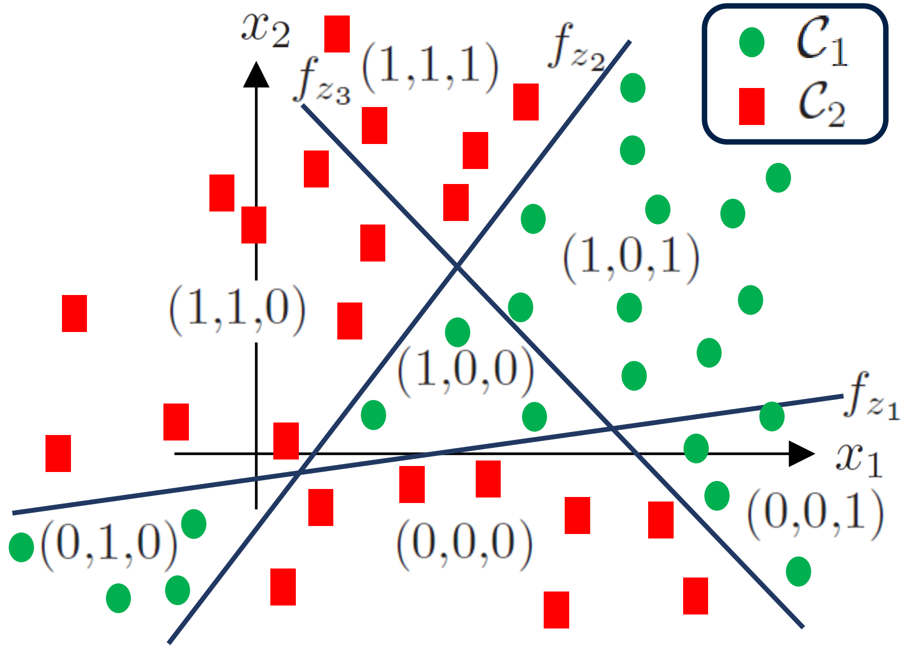
Two hidden layers

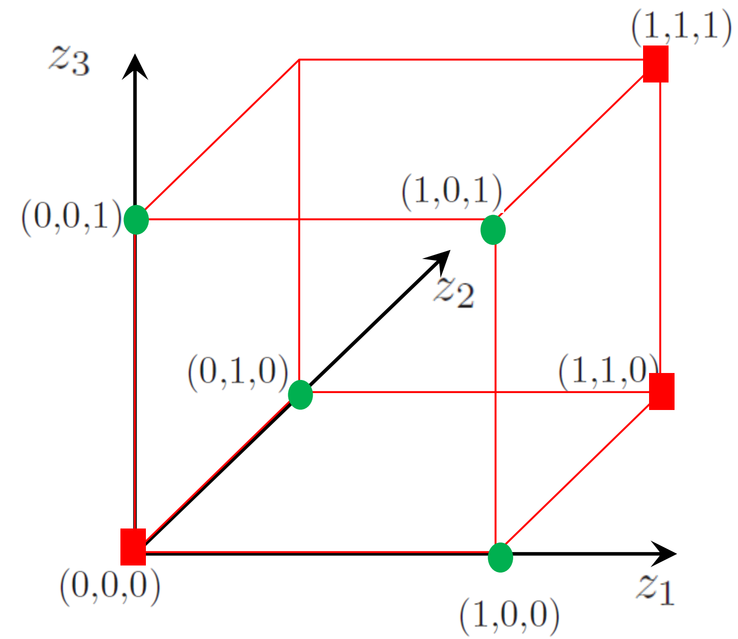


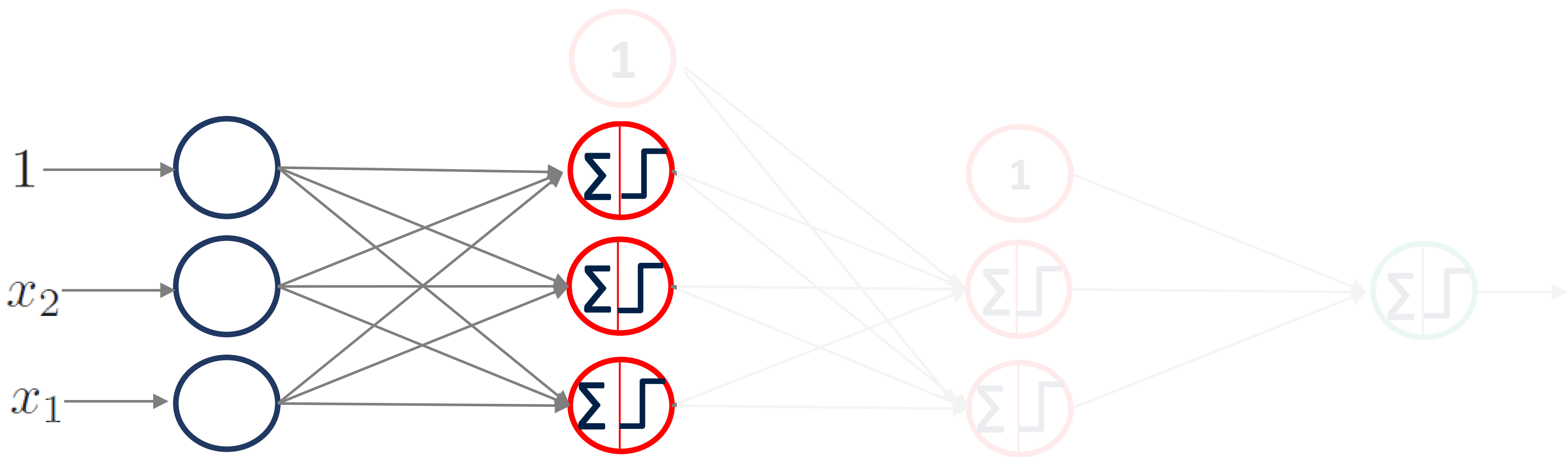




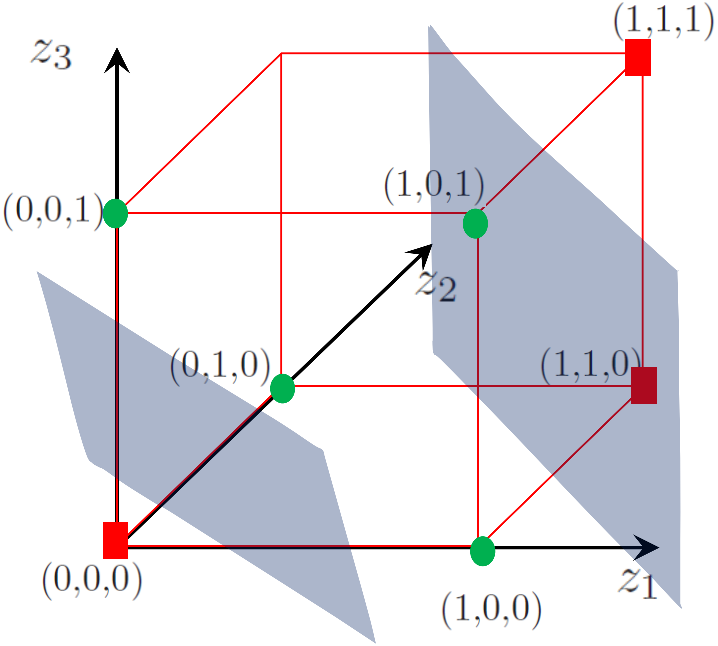
Two hidden layers

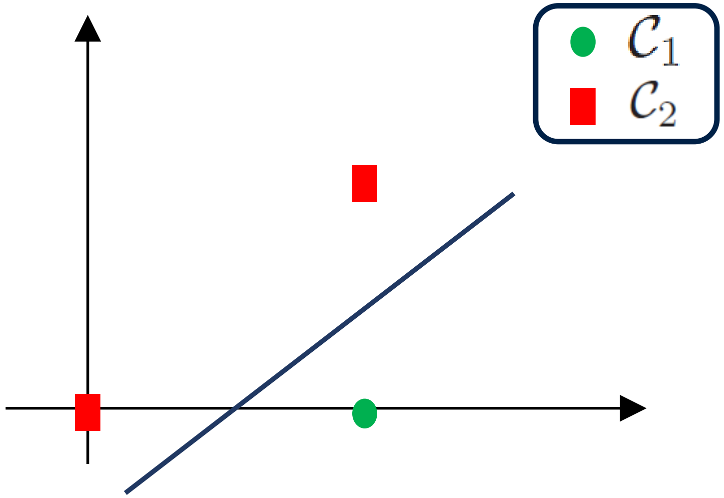


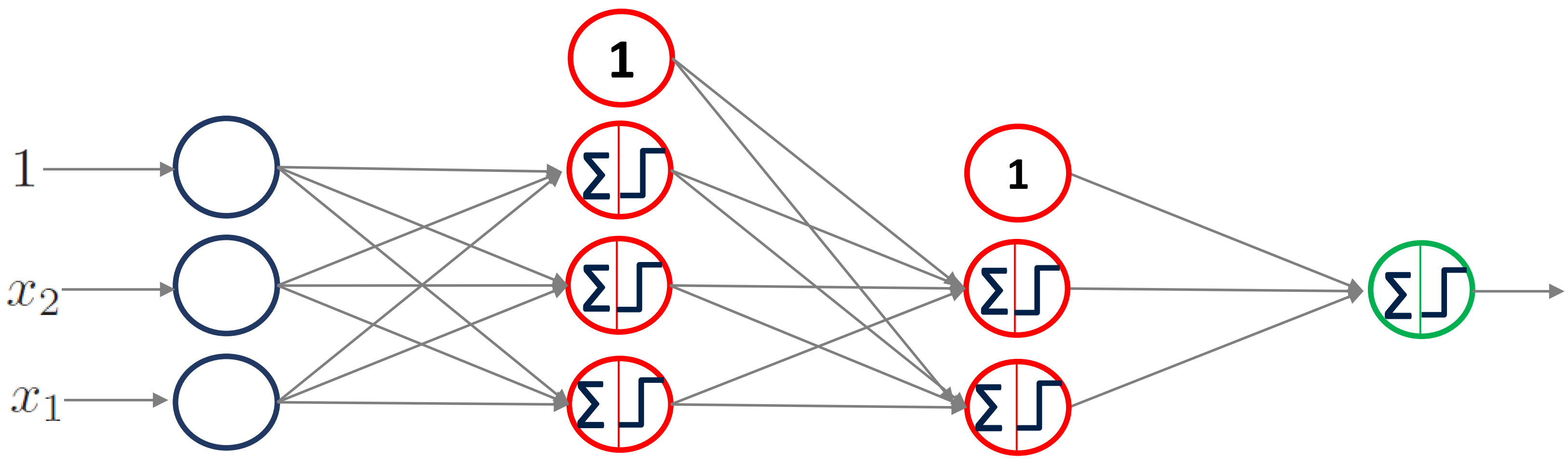




Two hidden layers



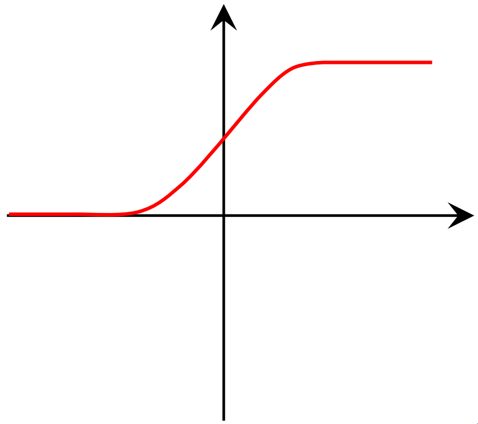




Sigmoid

- Bounded output: [0,1]
- Saturates for large input values, positive or negative.
  - Gradient becomes 0 (almost).
  - Leads to the problem of vanishing gradient in deep networks.
- Outputs not centered at 0.
- Not used much.

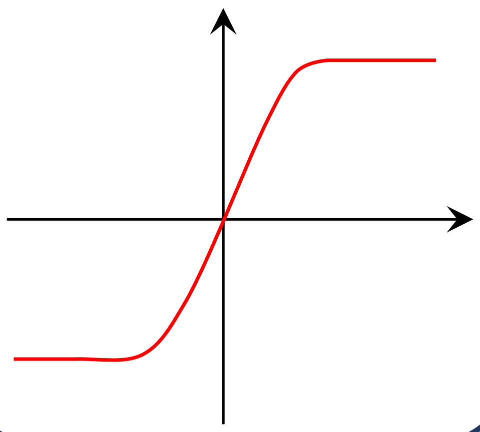
$$\mathcal{A}(z) = \frac{1}{1+\exp(-z)}$$



tanh

- Bounded output: [-1,1]
- Saturates for large input values, positive or negative.
  - Gradient becomes 0 (almost).
  - Leads to the problem of vanishing gradient in deep networks.
- Outputs centered at 0.
- Better than sigmoid activation function.

$$\mathcal{A}(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}$$



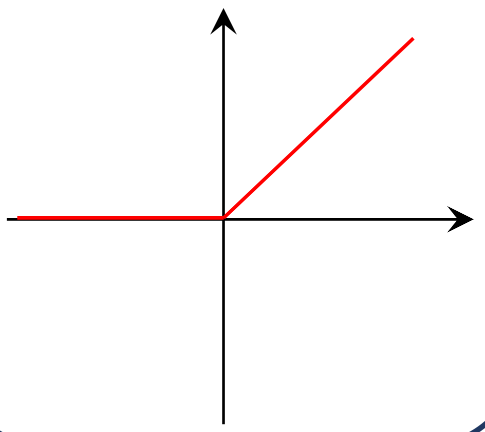
ReLU

- Output not bounded on the positive side.
- Very efficient in derivative computation:

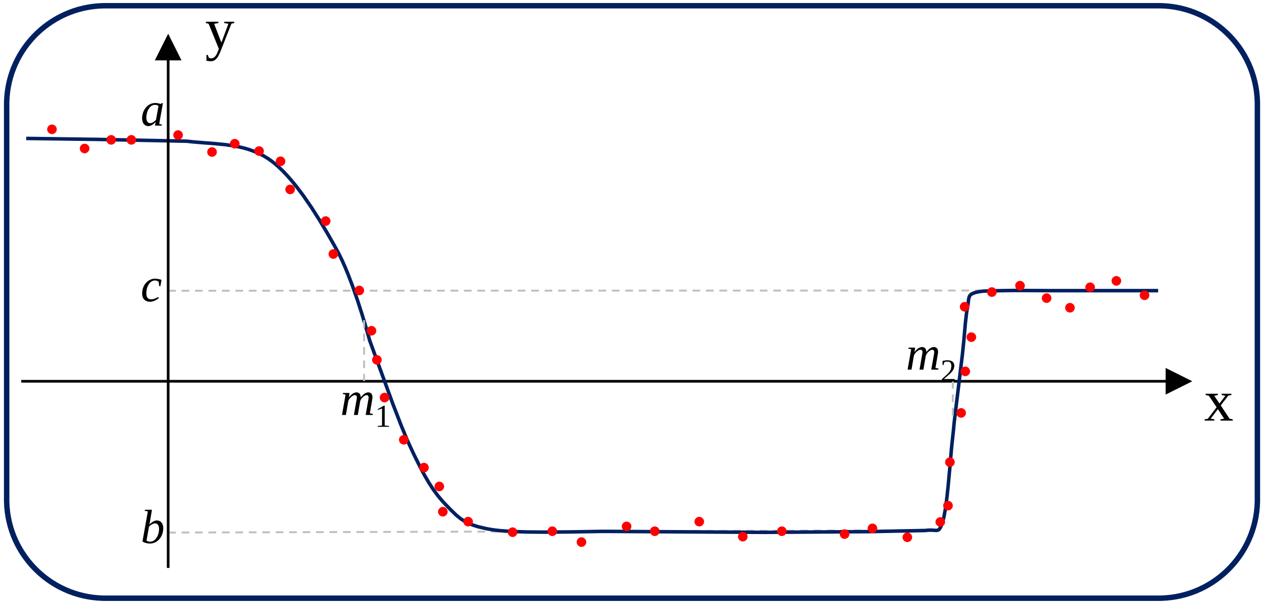
$$\mathcal{A}'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

- Known to have much faster convergence than tanh in some cases.
- If in the negative region, then unit is dead as there is no gradient.

$\mathcal{A}(z) = \max(z, 0)$

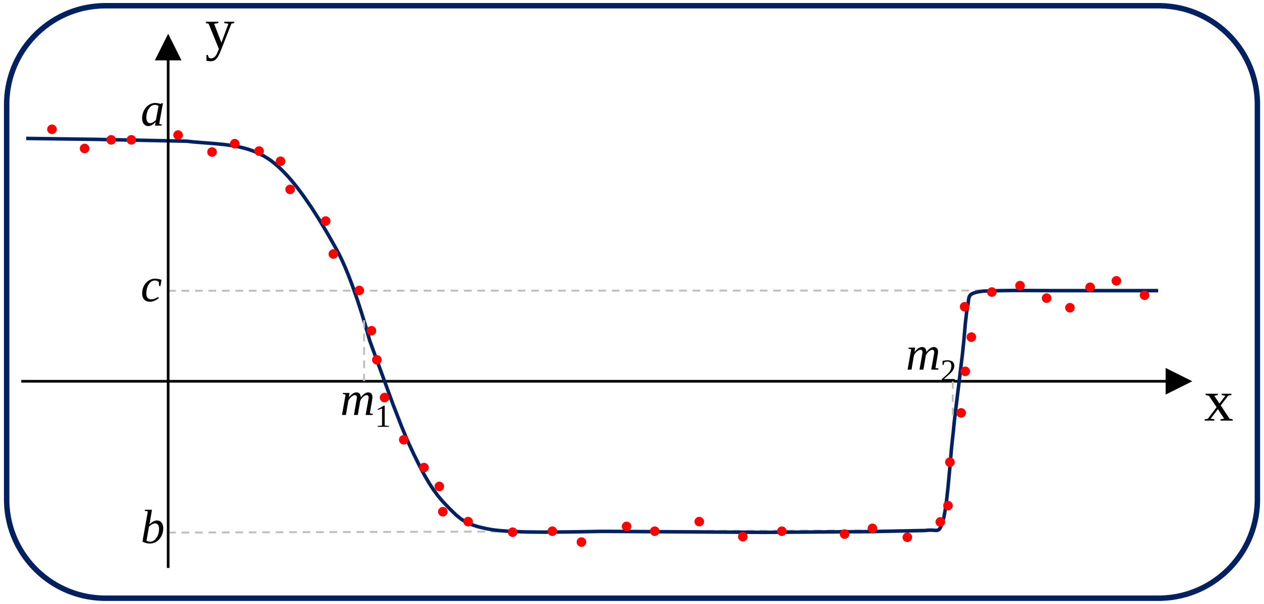


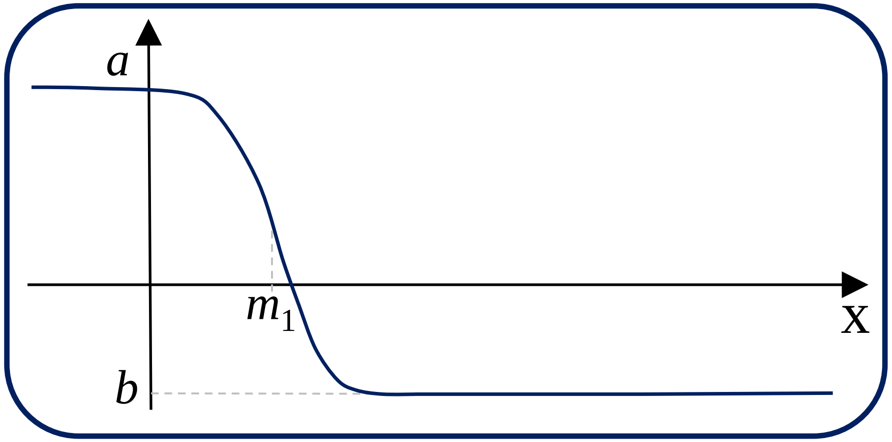
Regression problem



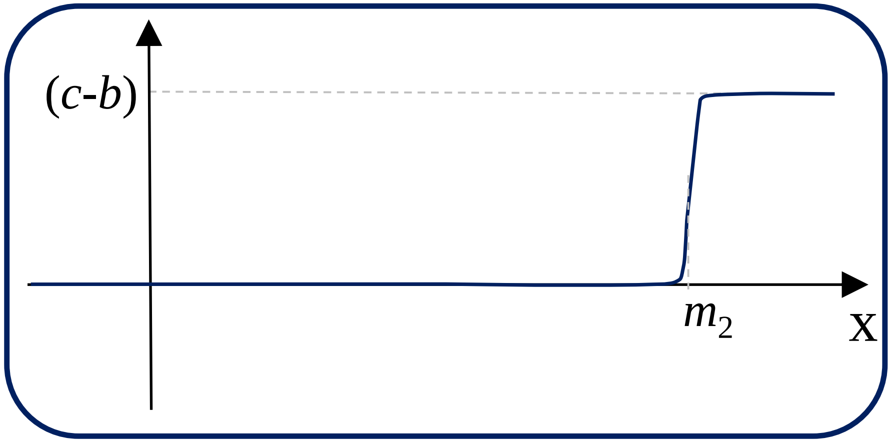
- Consider the following decomposition of the function

Regression problem

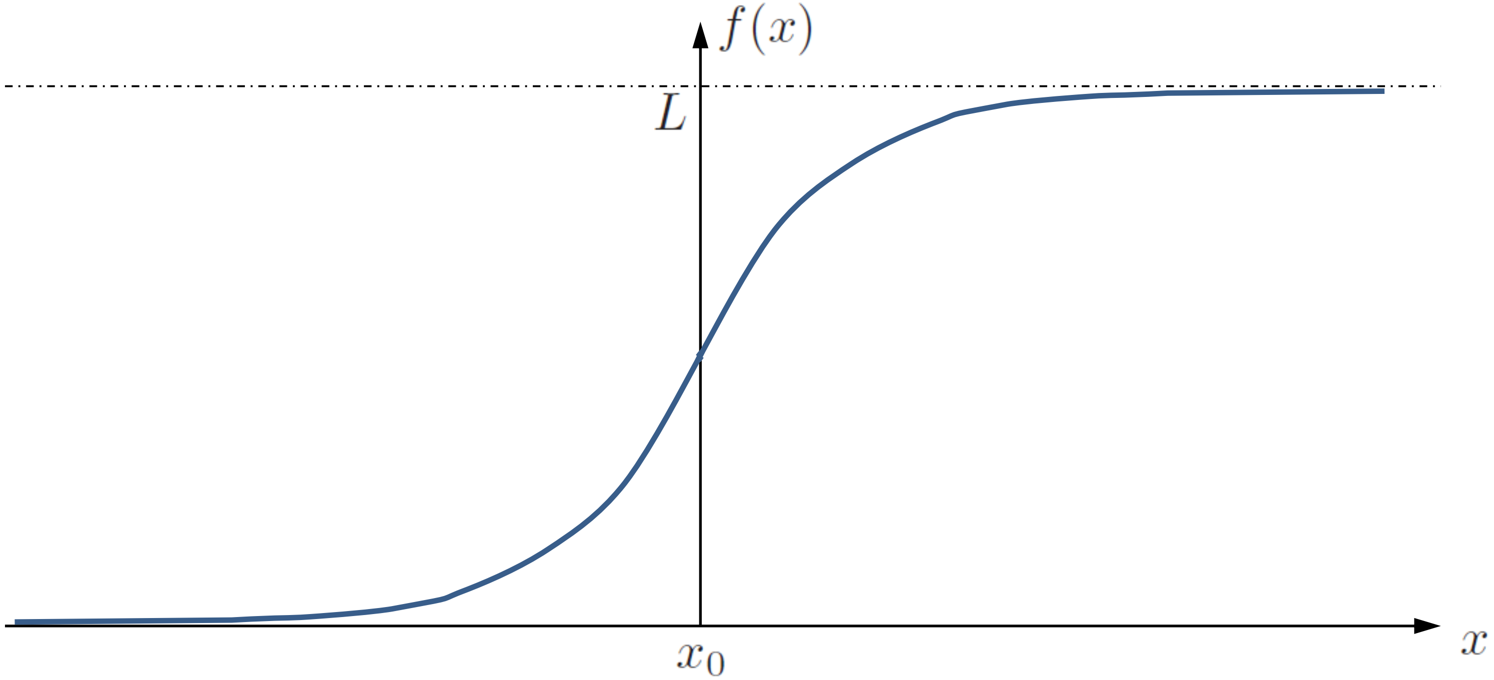




+



Sigmoid

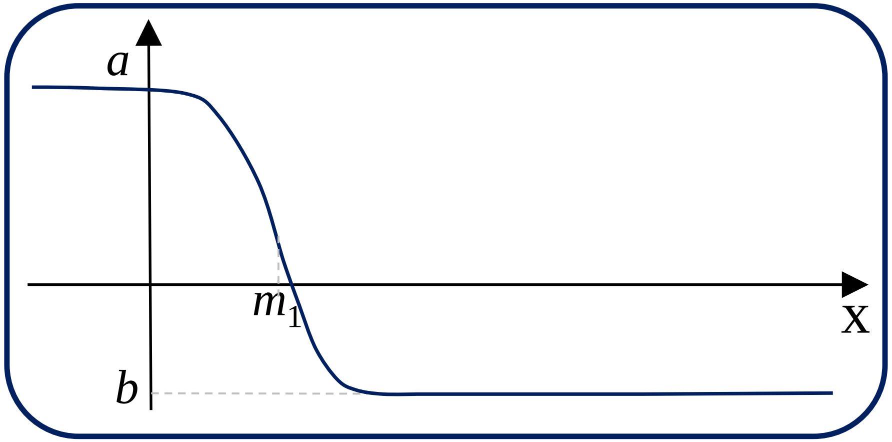


- Basic form of the sigmoid:

$$f(x) = \frac{L}{1 + \exp\left[-k(x - x_0)\right]}$$

  - L is the maximum value of the curve
  - k indicates the steepness of the curve
  - x_0 is the x coordinate of the sigmoid's midpoint

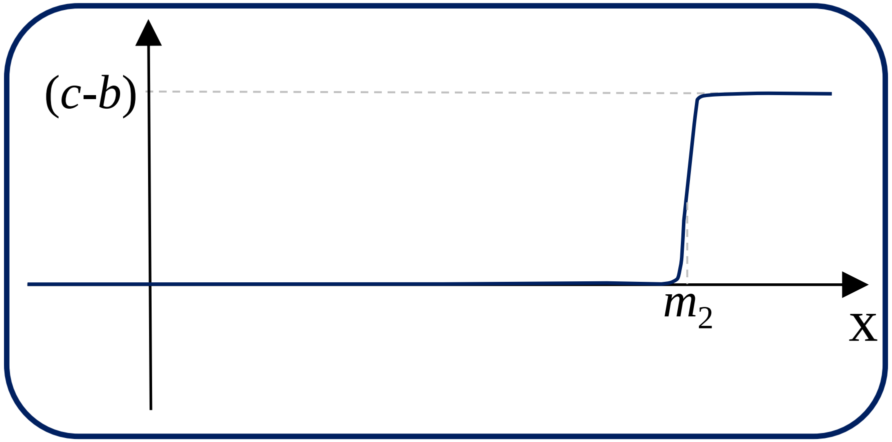
Function approximation



- Sigmoid approximation to the first function:

$$\mathcal{S}_1(x) = \frac{a-b}{1+\exp\left[-k_1(x-m_1)\right]} + b$$

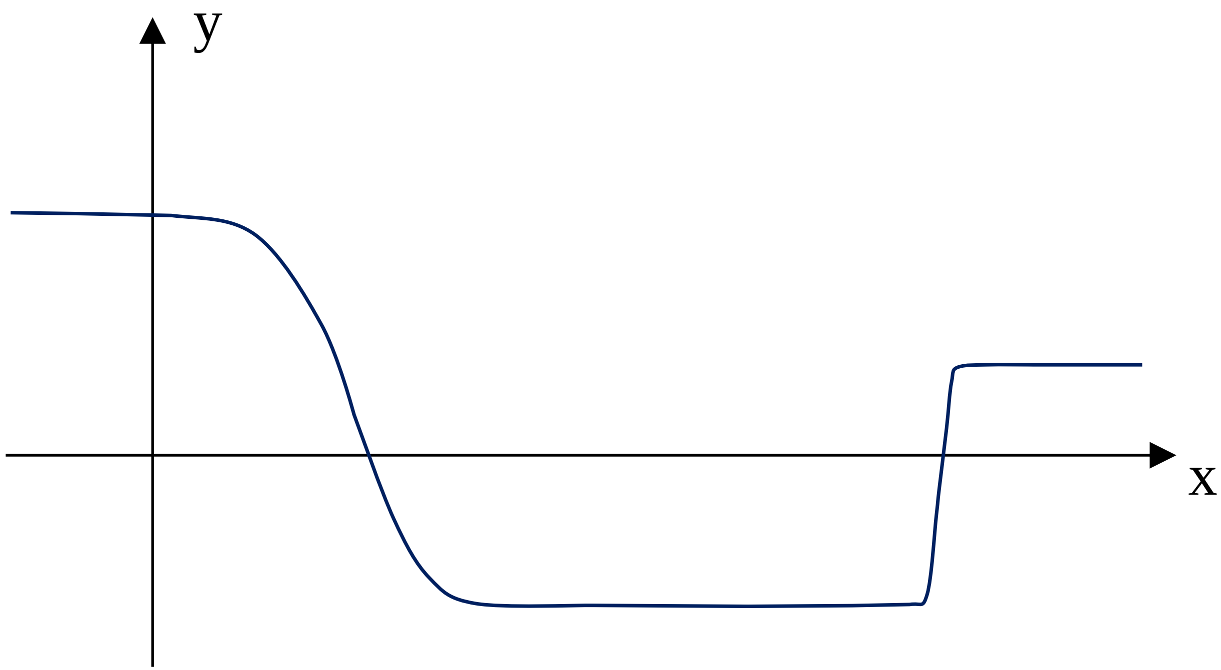
Function approximation



- Sigmoid approximation to the second function:

$$\mathcal{S}_2(x) = \frac{c-b}{1+\exp\left[-k_2(x-m_2)\right]}$$

Function approximation

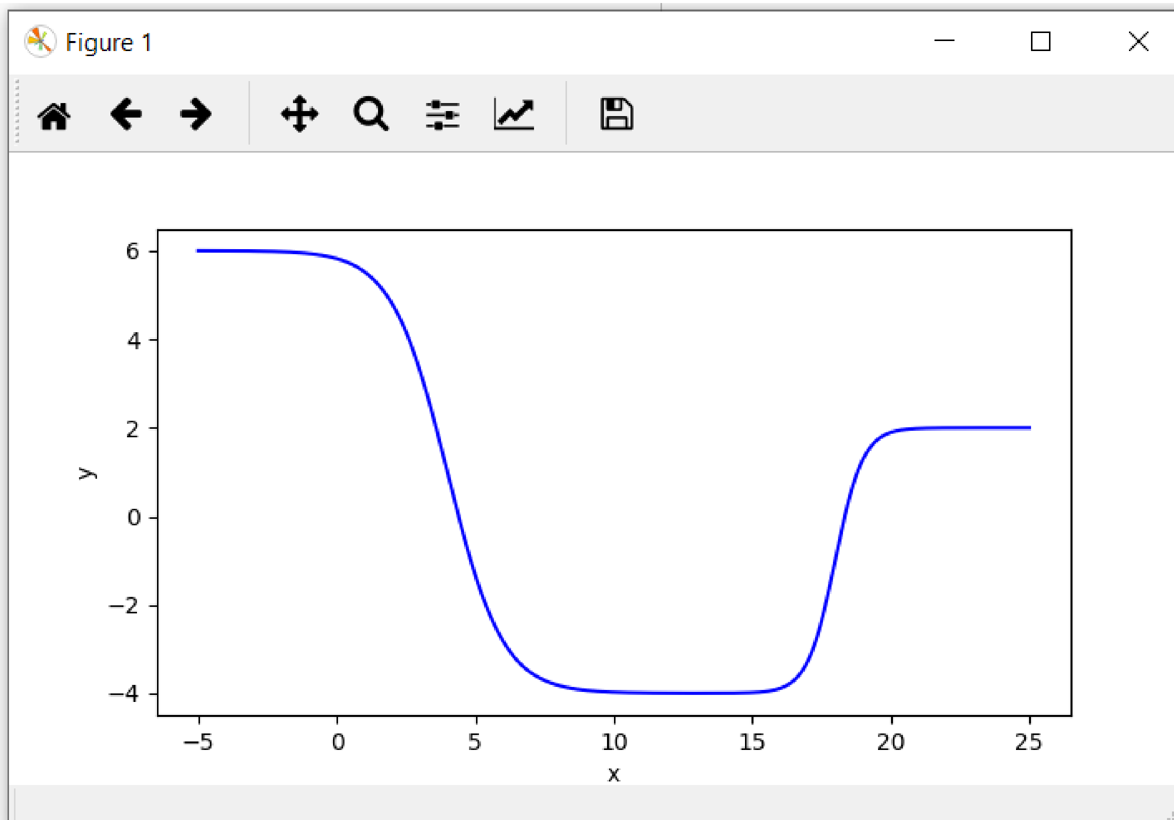


$$\overline{y} = \mathcal{S}_1(x) + \mathcal{S}_2(x)$$

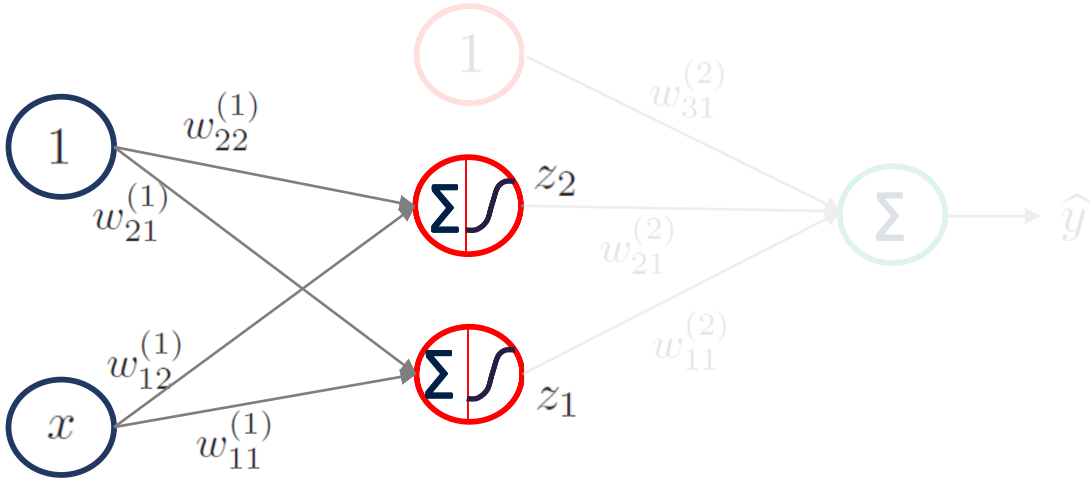
$$= b + \frac{a-b}{1+\exp\left[-k_1(x-m_1)\right]} + \frac{c-b}{1+\exp\left[-k_2(x-m_2)\right]}$$

Python example

```python
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(-5, 25, 0.001);
a=6;
b=-4;
c=2;

m1=4;
m2=18;

k1=-1;
k2=2;

y1 = b + (a-b)/(1+np.exp(-k1*(x-m1)));
y2 = 0 + (c-b)/(1+np.exp(-k2*(x-m2)));

y_hat = y1 + y2;


fig = plt.figure()
ax = fig.add_subplot(1,1,1)
ax.plot(x, y_hat, 'b-')
ax.set_xlabel('x')
ax.set_ylabel('y')
```



Neural network structure



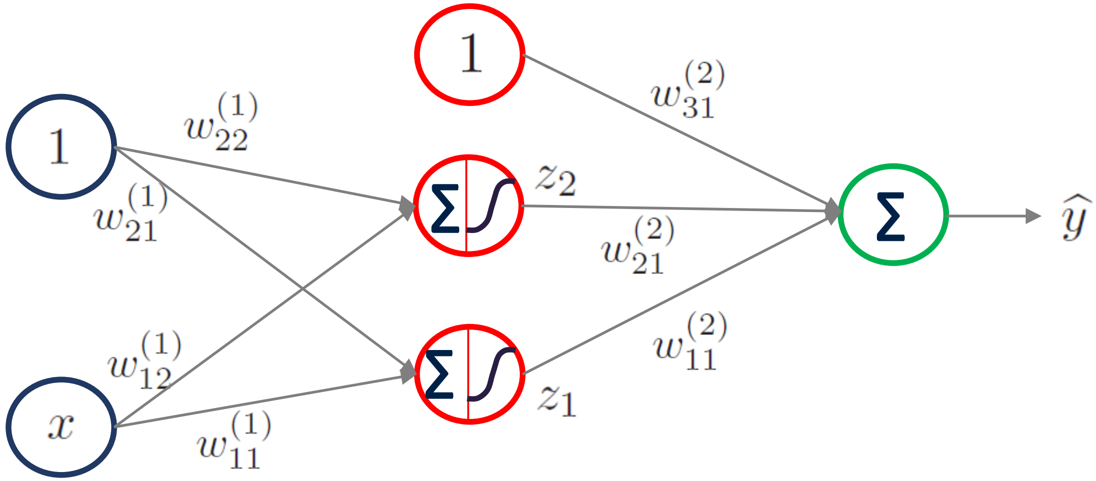
- Output of the first node of the hidden layer:

$$\begin{aligned} z_1 &= \sigma\left(w^{(1)}_{11} x + w^{(1)}_{21}\right) \\ &= \frac{1}{1 + \exp\left[-w^{(1)}_{11} x - w^{(1)}_{21}\right]} \end{aligned}$$

- Output of the second node of the hidden layer:

$$\begin{aligned} z_2 &= \sigma\left(w^{(1)}_{12} x + w^{(1)}_{22}\right) \\ &= \frac{1}{1 + \exp\left[-w^{(1)}_{12} x - w^{(1)}_{22}\right]} \end{aligned}$$

Neural network structure

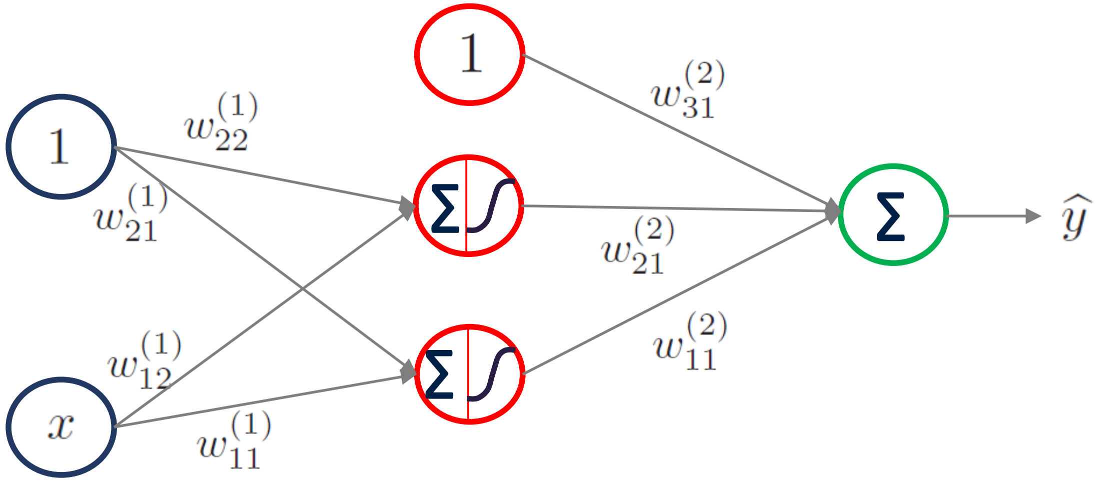


- Final output:

$$\widehat{y} = w_{11}^{(2)} z_1 + w_{21}^{(2)} z_2 + w_{31}^{(2)}$$

$$= \frac{w_{11}^{(2)}}{1 + \exp\left[- w_{11}^{(1)} x - w_{21}^{(1)} \right]} + \frac{w_{21}^{(2)}}{1 + \exp\left[- w_{12}^{(1)} x - w_{22}^{(1)} \right]} + w_{31}^{(2)}$$

Neural network structure



- Consider the following values of the weights:

$$\text{First layer}: w_{11}^{(1)} = k_1; \quad w_{21}^{(1)} = -k_1 m_1; \quad w_{12}^{(1)} = k_2; \quad w_{22}^{(1)} = -k_2 m_2$$

$$\text{Second layer}: w_{11}^{(2)} = (a-b); \quad w_{21}^{(2)} = (c-b); \quad w_{31}^{(2)} = b$$

- On substitution we get

$$\widehat{y} = b + \frac{a-b}{1+\exp\left[-k_1(x-m_1)\right]} + \frac{c-b}{1+\exp\left[-k_2(x-m_2)\right]}$$

$$= \overline{y}$$