Neural Networks

Fatemeh Mansoori

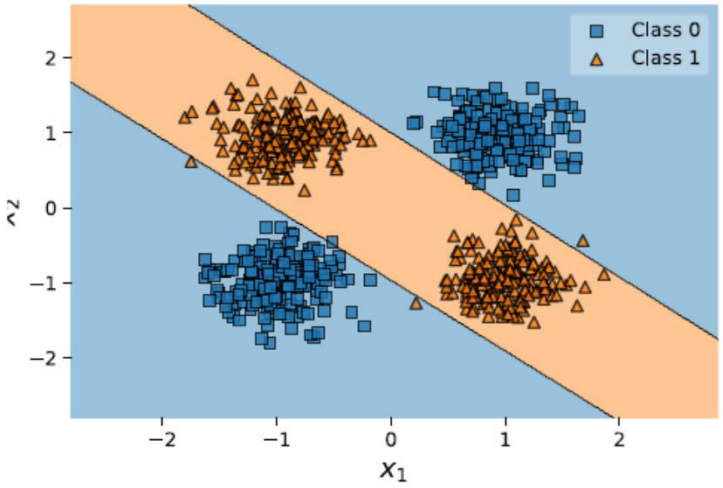
University of Isfahan

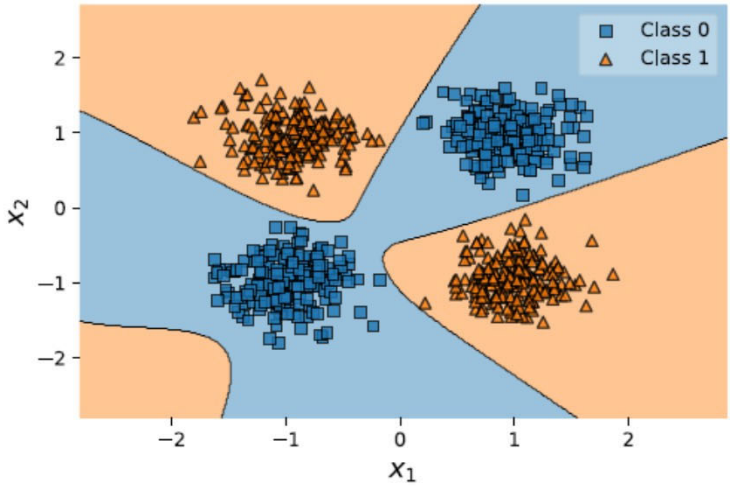
Some of the slide are based on slides from the machine learning course by sharifi zarchi

Limitation of perceptron

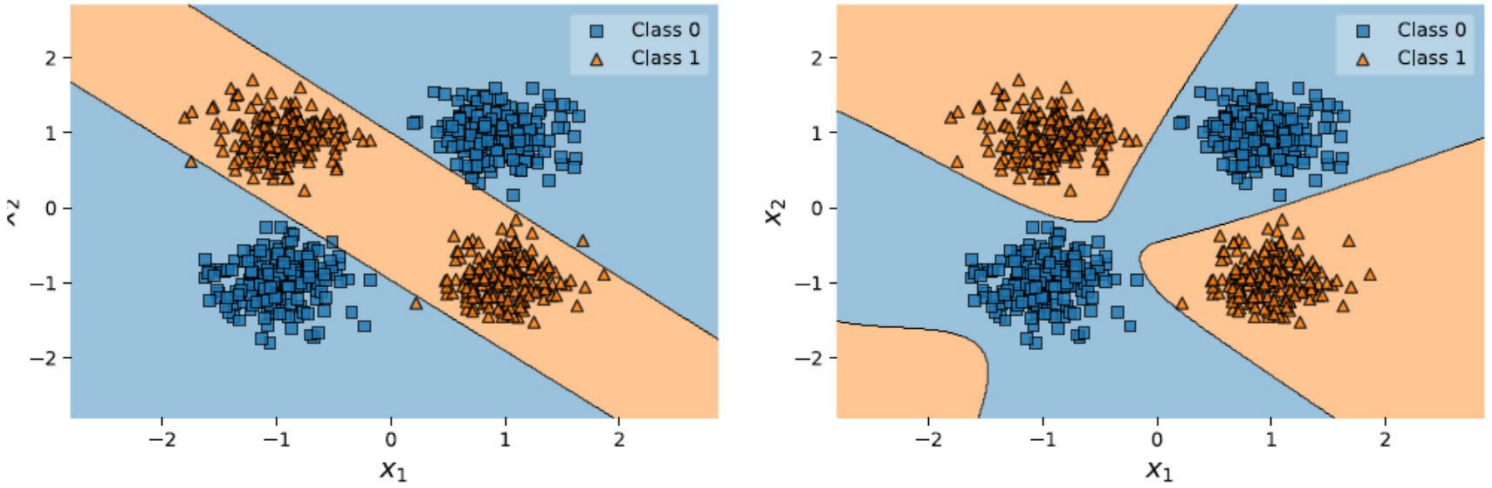
- XOR can not be represented by perceptron
- We need a deeper network
- No one knew how to train deeper networks

Why can’t a perceptron represent XOR?



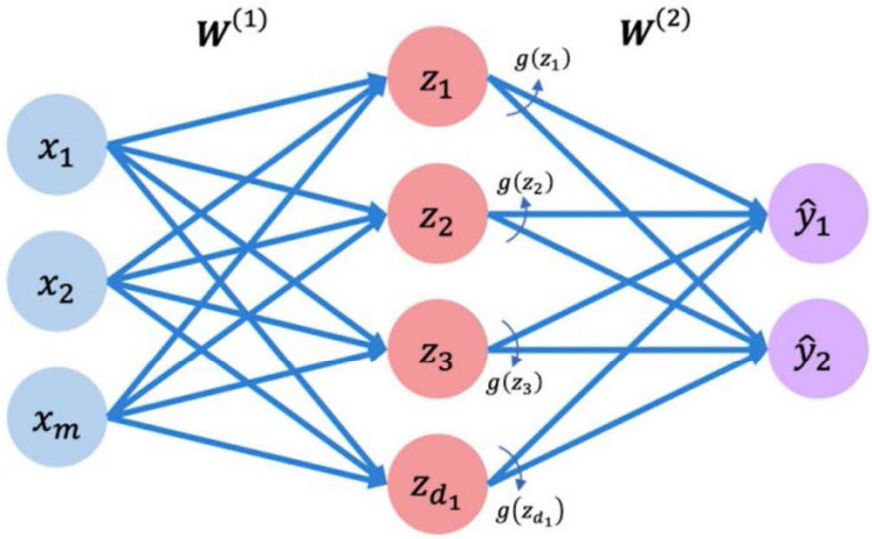


Multilayer Neural Networks Can Solve XOR Problems



- Decision boundaries of two different multilayer perceptions on simulated data solving the XOR problem

Neural Network with 1 Hidden Layer

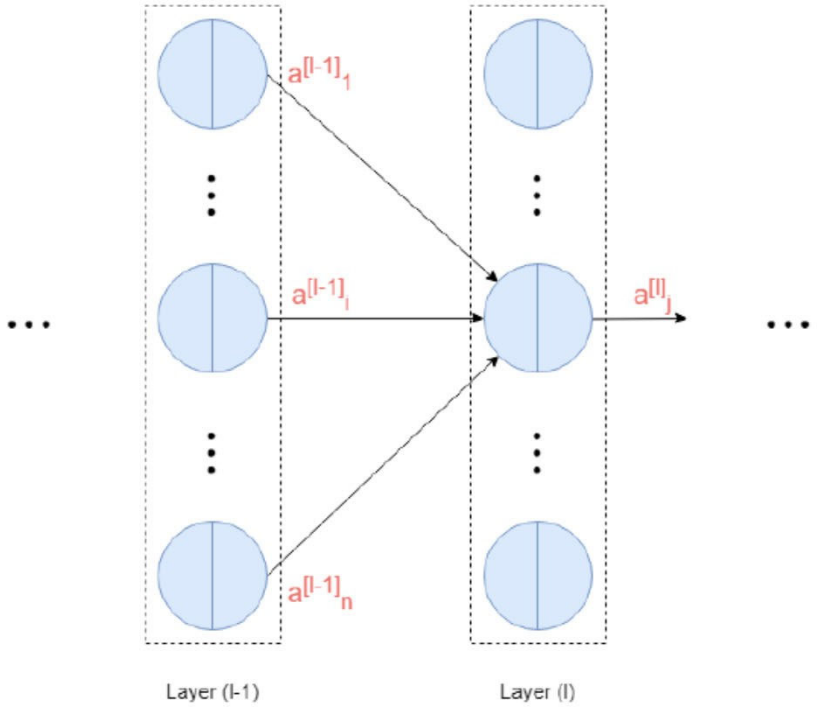


$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j \, w_{j,i}^{(1)} \qquad \hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} g(z_j) \, w_{j,i}^{(2)}\right)$$

Standard notations for MLP

$a_i^{[l]}$: i-th neuron output in layer l

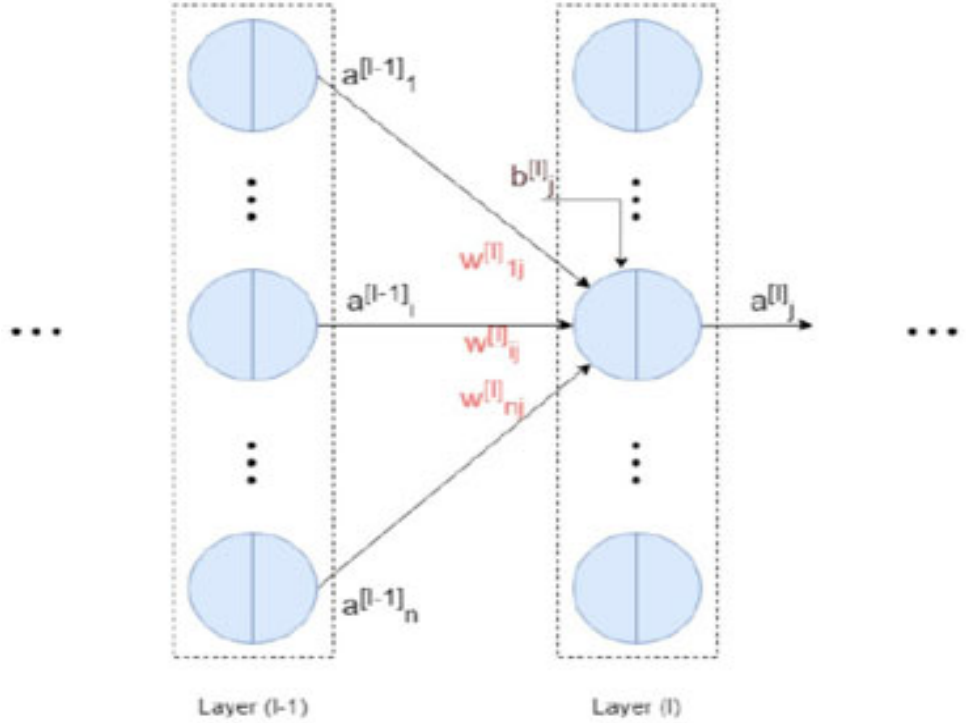
$\boldsymbol{a}^{[l]}$: layer l output in vector form



MLP notation

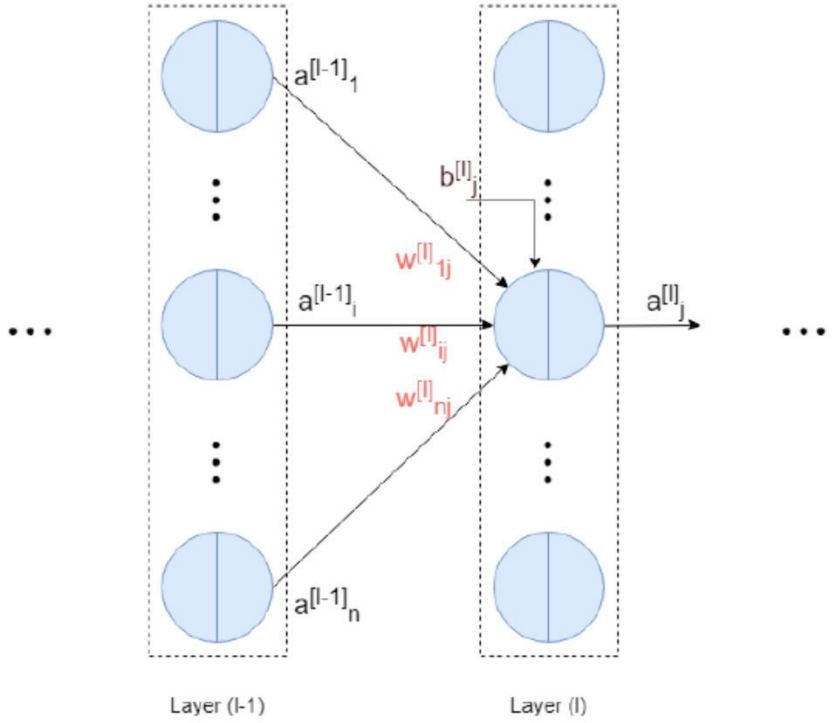
$b_i^{[l]}$: i-th neuron bias in layer l

$\boldsymbol{b}^{[l]}$: layer l biases in vector form



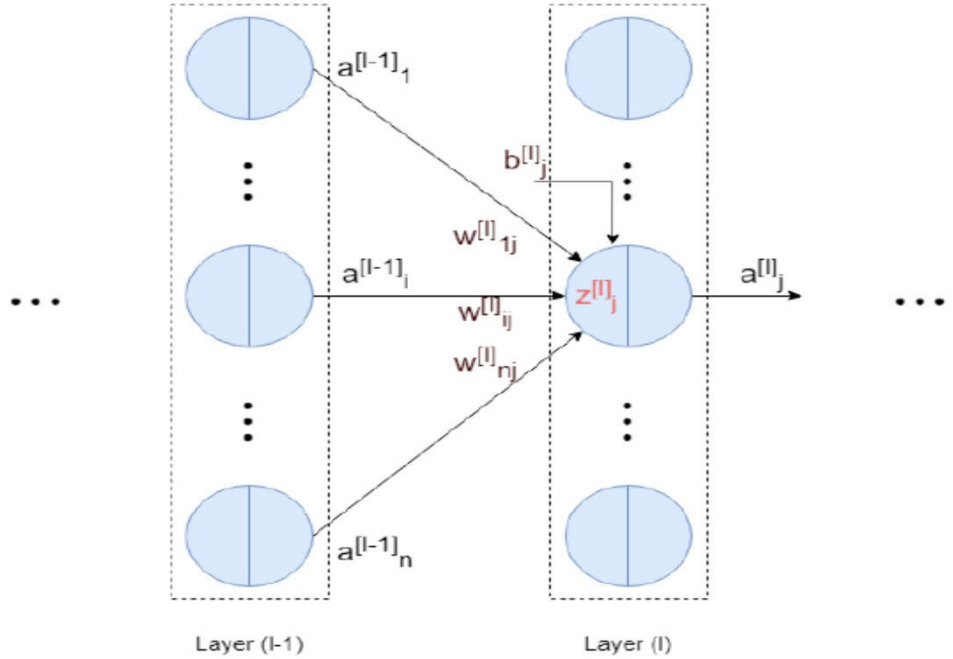
MLP notation

$W_{ij}^{[l]}$: weight of the edge between i-th neuron in layer $l-1$ and j-th neuron in layer l



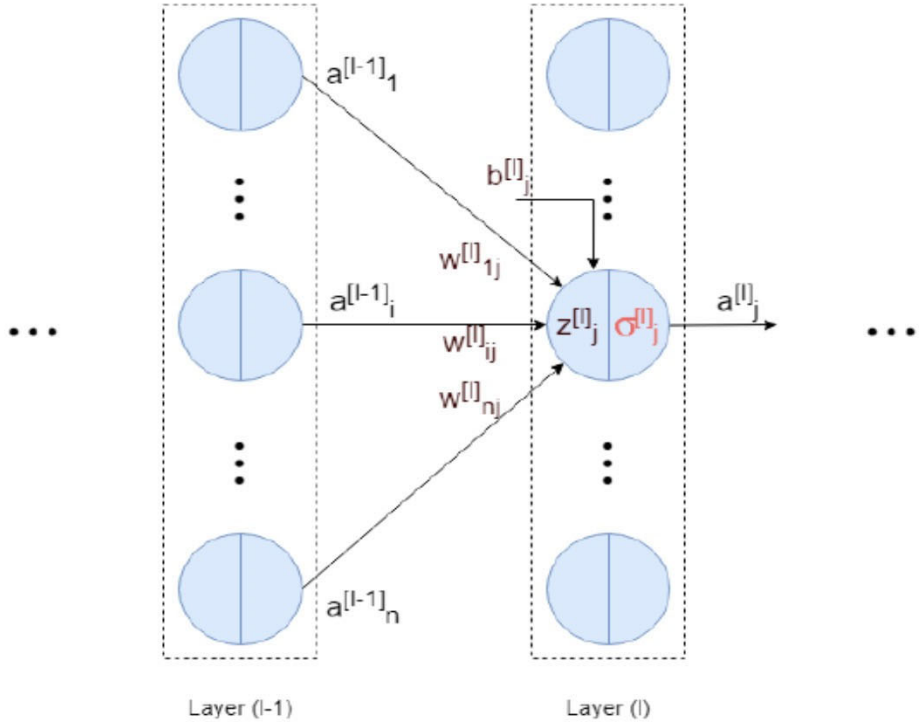
MLP notation

- $z_j^{[l]}$: j-th neuron input in layer l
- $z_j^{[l]} = b_j^{[l]} + \sum_{i=1}^{n} W_{ij}^{[l]} a_i^{[l-1]}$



MLP notation

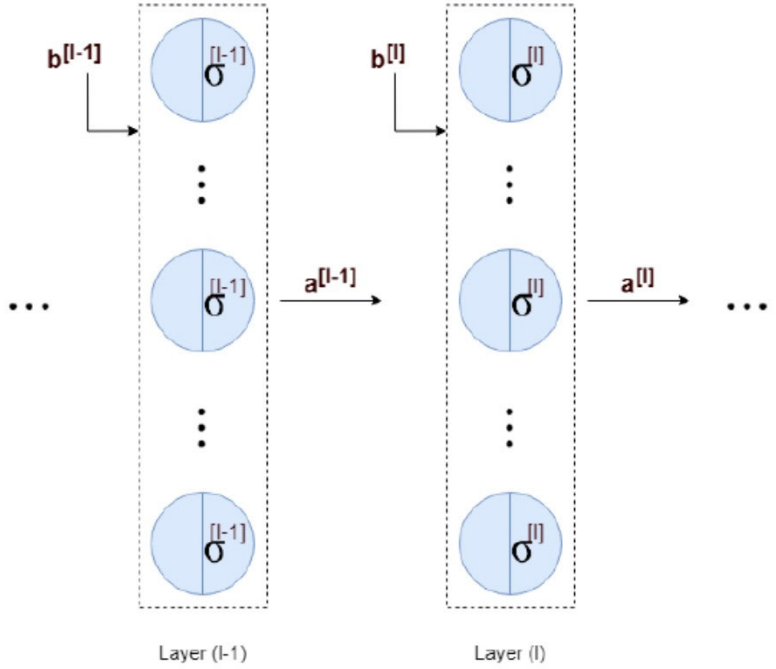
$\sigma_j^{[l]}$: j-th neuron activation function in layer l



MLP notation

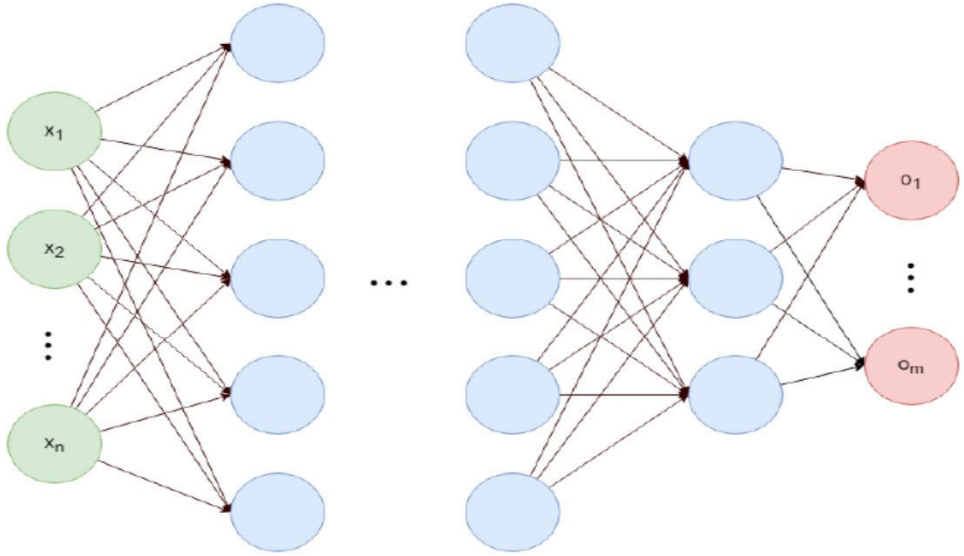
- If all neurons in one layer have the same activation function then :

$$\boldsymbol{a}^{[l]} = \sigma^{[l]}\left(\boldsymbol{b}^{[l]} + (W^{[l]})^T \boldsymbol{a}^{[l-1]}\right)$$



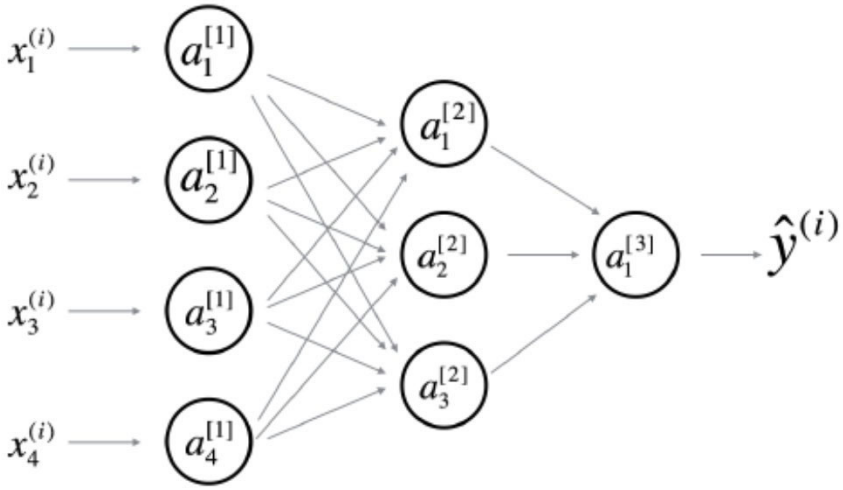
MLP notation

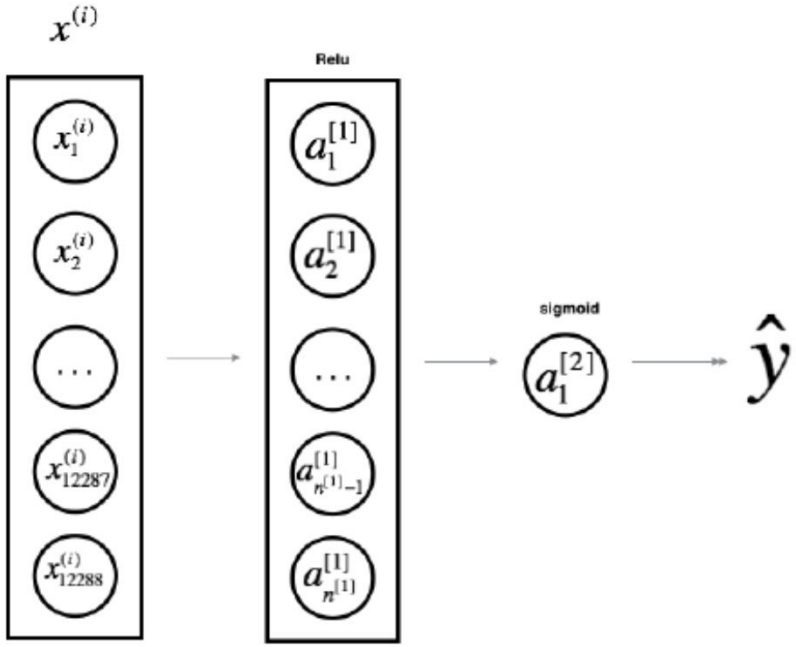
- For a network with L layer and x as its input we have :



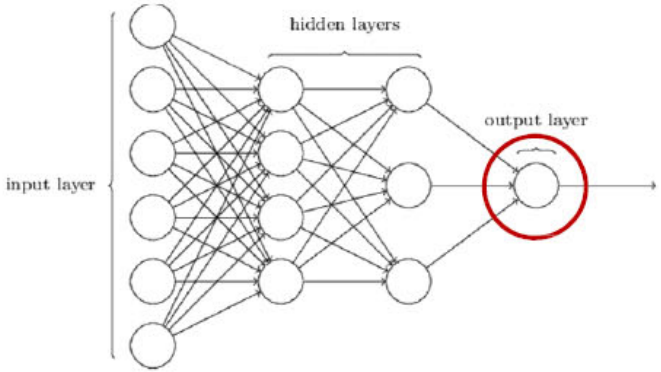
$$\boldsymbol{o} = \boldsymbol{a}^{[L]} = \sigma^{[L]}\left(\boldsymbol{b}^{[L]} + (W^{[L]})^T \sigma^{[L-1]}\left(\cdots \sigma^{[1]}\left(\boldsymbol{b}^{[1]} + (W^{[1]})^T \boldsymbol{x}\right)\cdots\right)\right)$$

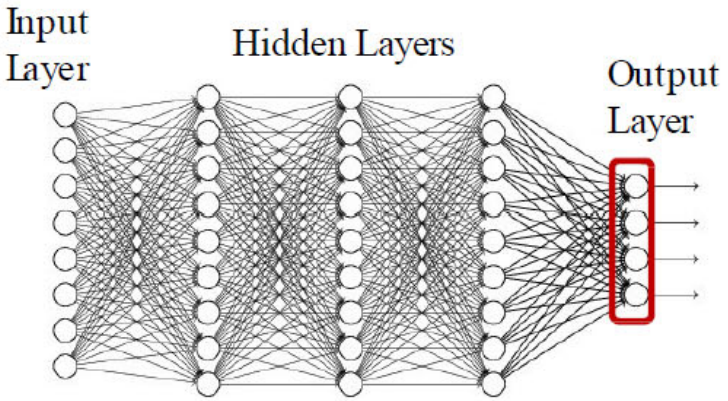
Deep Learning representations





Representing the output

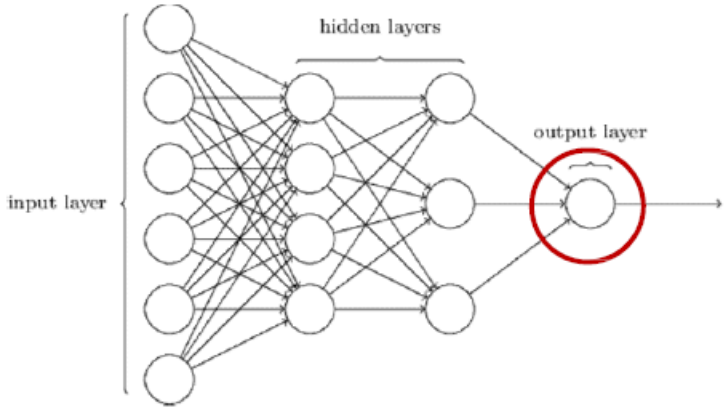




If the desired output is real-valued, no special tricks are necessary

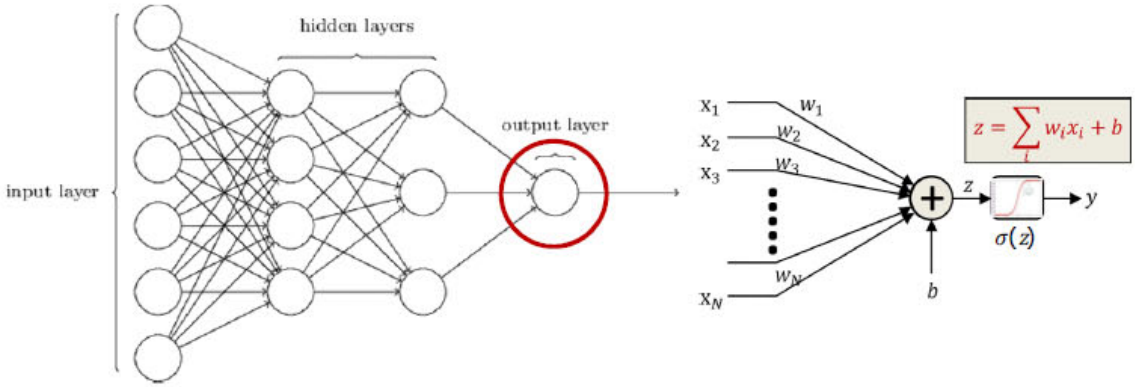
- Scalar Output : single output neuron
  - o = scalar (real value)
- Vector Output : as many output neurons as the dimension of the desired output
  - $\boldsymbol{o} = [o_1, o_2, \ldots, o_K]^T$ (vector of real values)

Representing the output



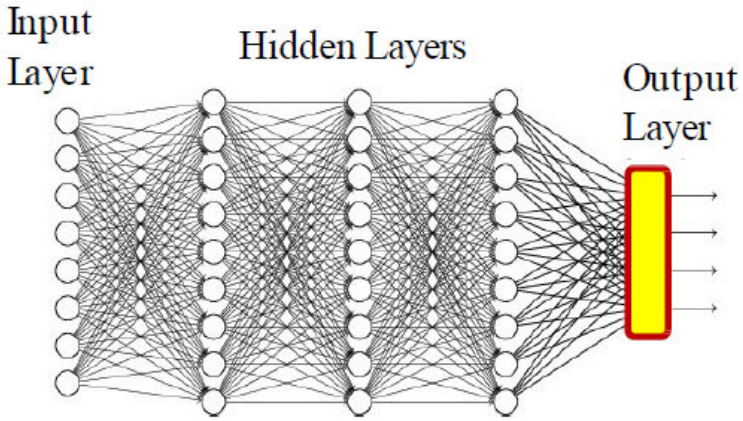
- If the desired output is binary (is this a cat or not), use a simple 1/0 representation of the desired output
  - 1 =YES it's a cat
  - 0 = NO it's not a cat.

Representing the output



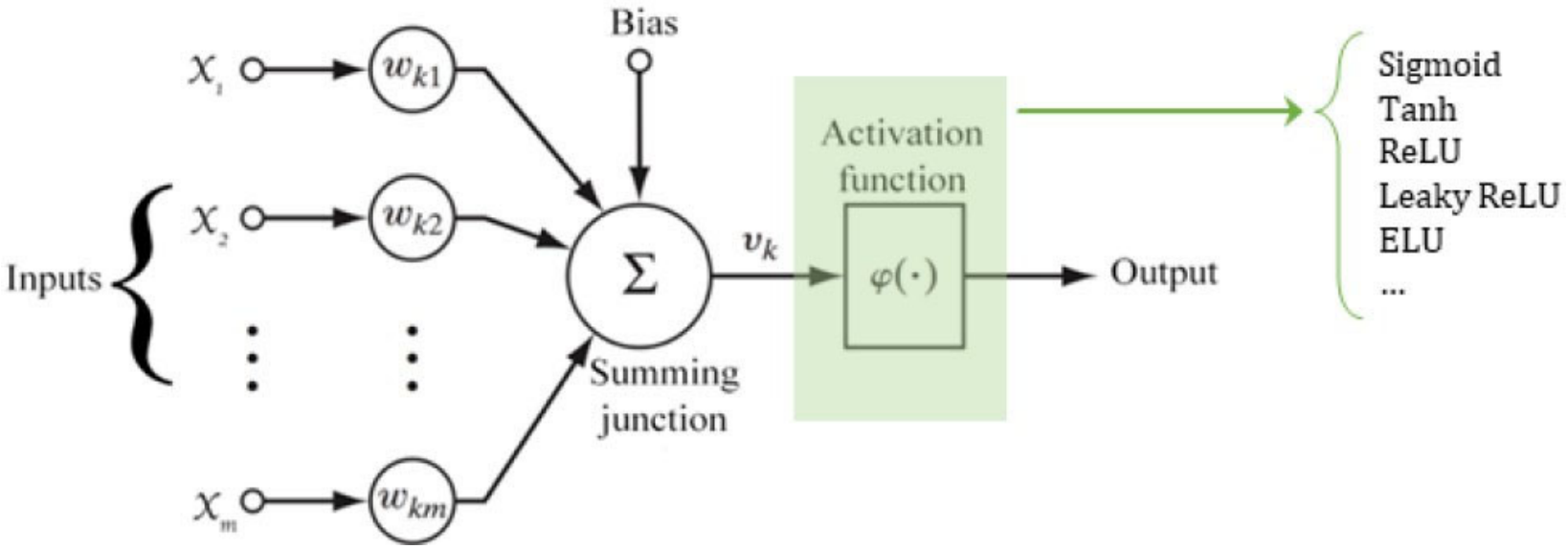
- If the desired output is binary, use a simple 1/0 representation of the desired output
- Output activation: Typically a sigmoid
  - Viewed as the probability $P(Y = 1|\boldsymbol{x})$ of class value 1
    - Indicating the fact that for actual data, in general a feature vector may occur for both classes, but with different probabilities
    - Is differentiable

Multi class networks



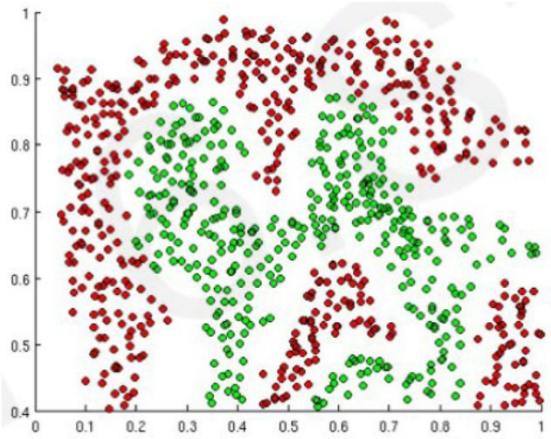
- For a multi-class classifier with K classes, the one-hot representation for the desired output y
  - The neural network's output too must ideally be binary (K-1 zeros and a single 1 in the right place)
- The network's output will be a probability vector
  - K probability values that sum to 1.

Activation function



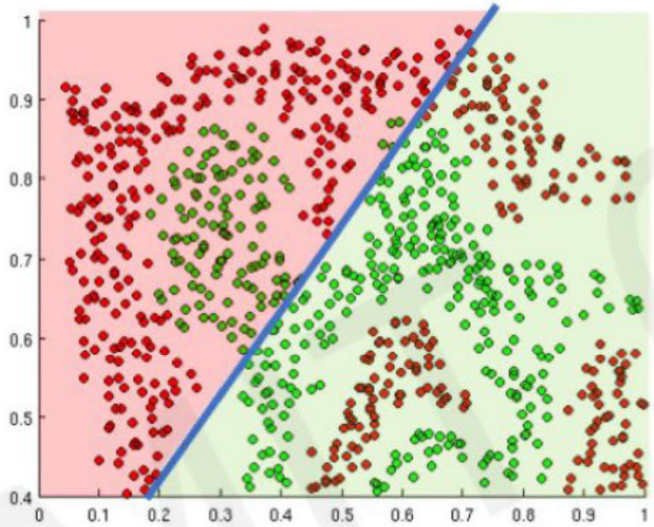
Importance of Activation Function

- The propose of activation function is to introduce non-linearities into network

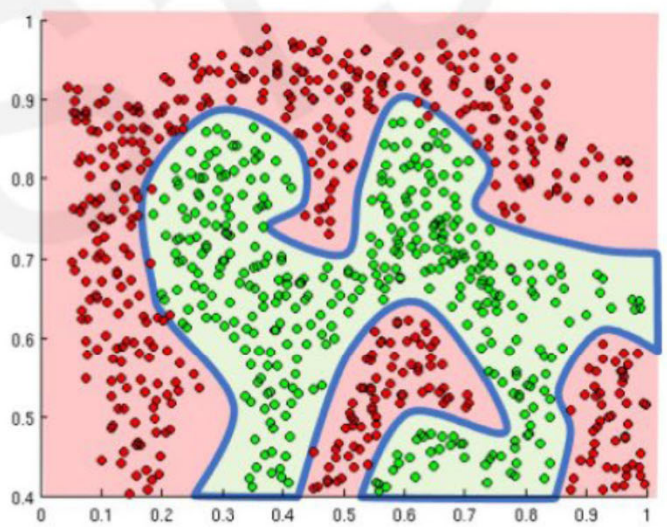


- What if we wanted to build a neural network to distinguish green vs red points?

Importance of Activation Function



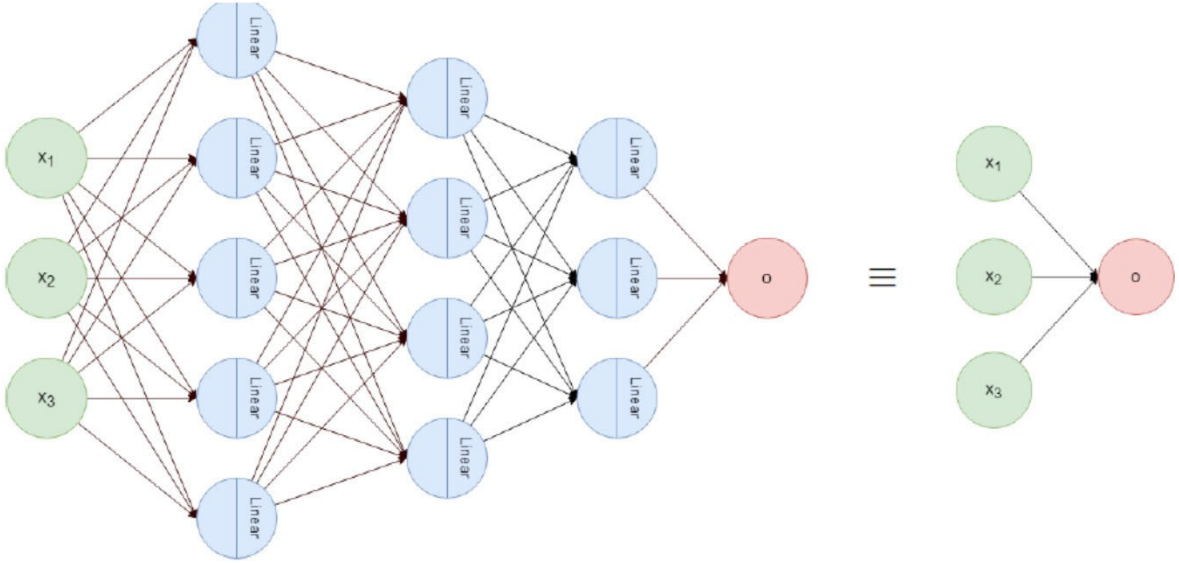
Linear activation functions produce linear decisions no matter the network size



Non-linearities allow us to approximate arbitrarily complex functions

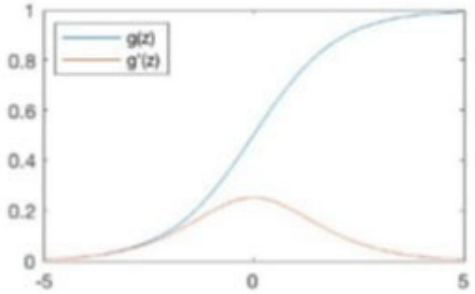
Activation Function of Hidden Layers

- One can use any activation function for each hidden units
- Usually people use the same activation function for all neurons in one Layer
- The important point is to use nonlinear activation functions



Common Activation functions

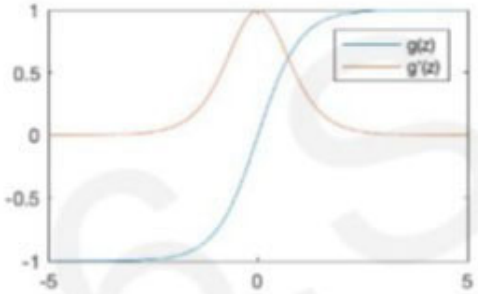
Sigmoid Function



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

$$g'(z) = g(z)(1 - g(z))$$

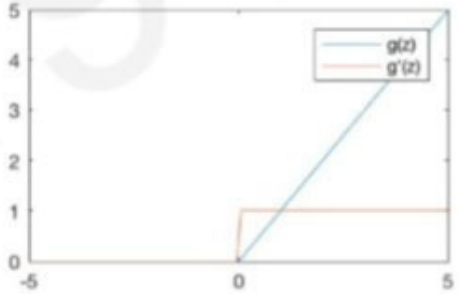
Hyperbolic Tangent



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

$$g'(z) = 1 - g(z)^2$$

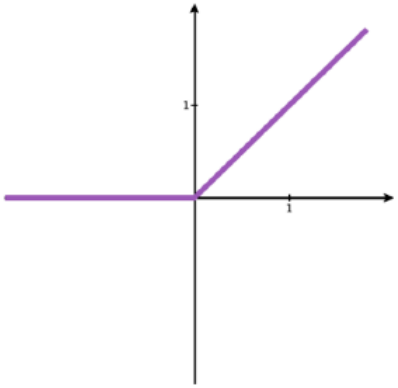
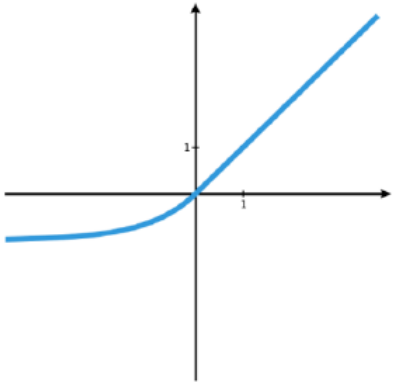
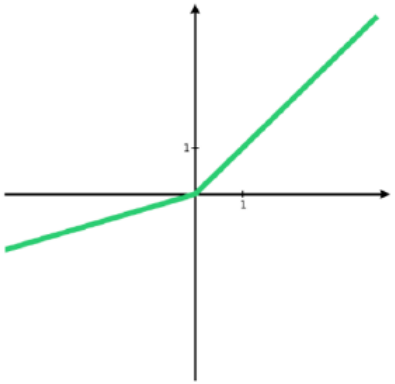
Rectified Linear Unit (ReLU)



$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

Variation of ReLU

ReLU	ELU	Leaky ReLU
$f(x) = \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases}$	$f(x) = \begin{cases} x & x > 0 \\ \alpha(e^x - 1) & x \leq 0 \end{cases}$	$f(x) = \begin{cases} x & x \geq 0 \\ 0.01x & x < 0 \end{cases}$
		

Softmax activation function

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, \ldots, K \text{ and } \mathbf{z} = (z_1, \ldots, z_K) \in \mathbb{R}^K. \qquad \sigma : \mathbb{R}^K \to (0,1)^K,$$

The softmax activation function takes in a vector of **raw outputs** of the neural network and returns a vector of **probability scores**.

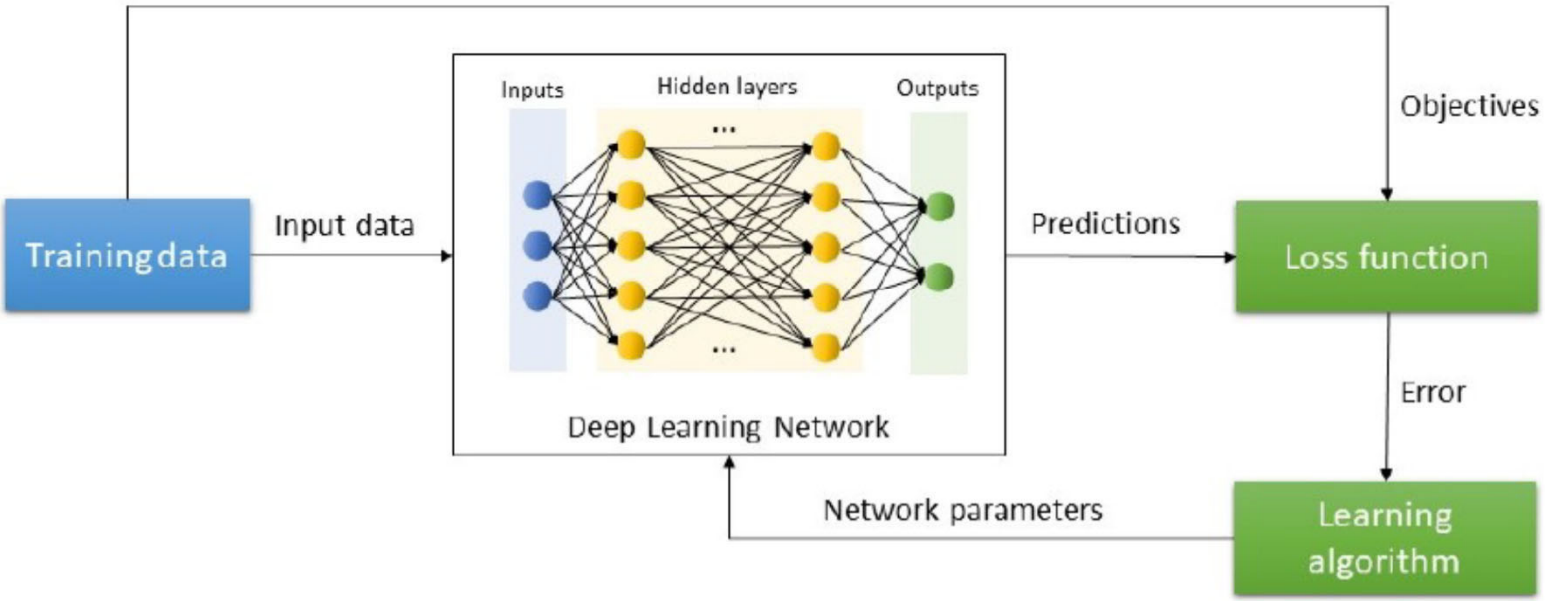
- **z** is the vector of raw outputs from the neural network

- The i-th entry in the softmax output vector softmax(**z**) can be thought of as the predicted probability of the test input belonging to class i.

regardless of whether the input x is positive, negative, or zero, e^x is always a positive number.

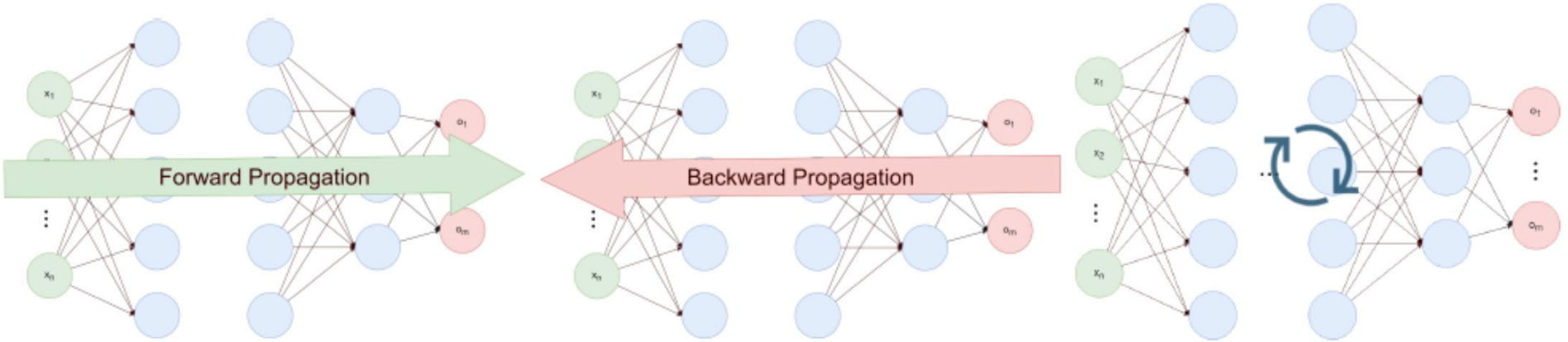
Why Won't Normalization by the Sum Suffice ?

Training Process



Training process

- The idea is to use gradient descent



(a) Forward pass (b) Backward pass (c) Update parameters

Learning MLPs

Let's define the learning problem more formal:

- $\{(x^{(i)}, y^{(i)})\}_{i=1}^{n}$: dataset
- f: network
- W: all weights and biases of the network ($W^{[l]}$ and $\boldsymbol{b}^{[l]}$ for different l)
- L: loss function

We want to find W^* which minimizes following cost function:

$$\mathcal{J}(W) = \sum_{i=1}^{n} L\left(f(x^{(i)}; W), y^{(i)}\right)$$

We are going to use gradient descent, so we need to find $\nabla_W \mathcal{J}$.

Forward propagation

First of all we need to find loss value.

It only requires to know the inputs of each neuron.

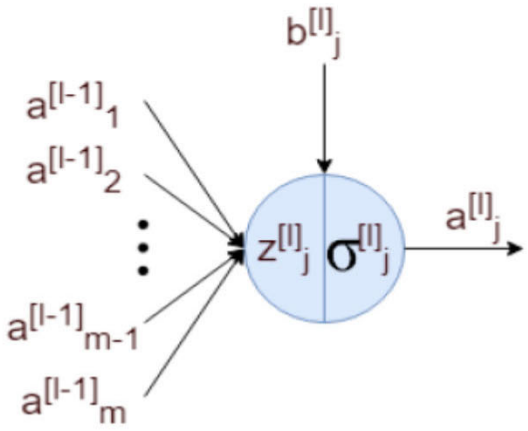


Figure: $a_j^{[l]} = \sigma_j^l(\sum_{i=1}^{m} W_{ij}^{[l]} a_i^{[l-1]} + b_j^{[l]})$

So we can calculate these outputs layer by layer.

Forward propagation

After forward pass we will know:

- Loss value
- Network output
- Middle values

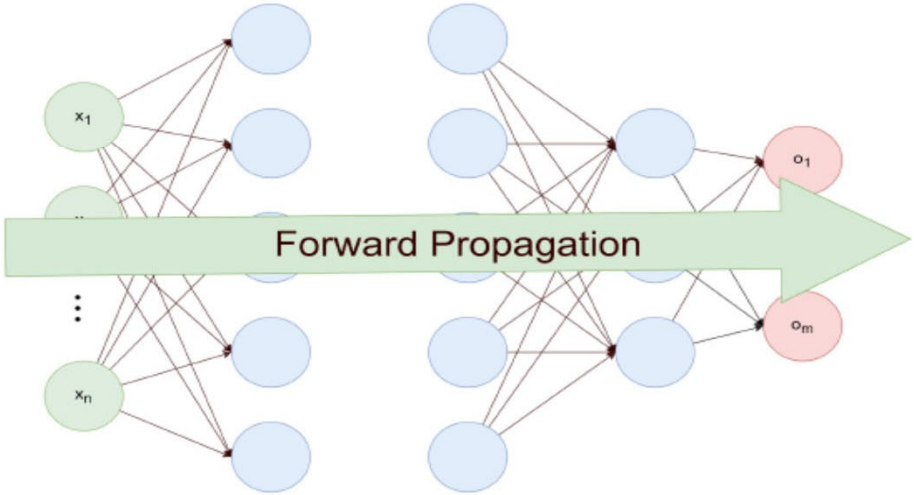


Figure: Forward pass

Backward Propagation

Now we need to calculate $\nabla_W \mathcal{J}$.

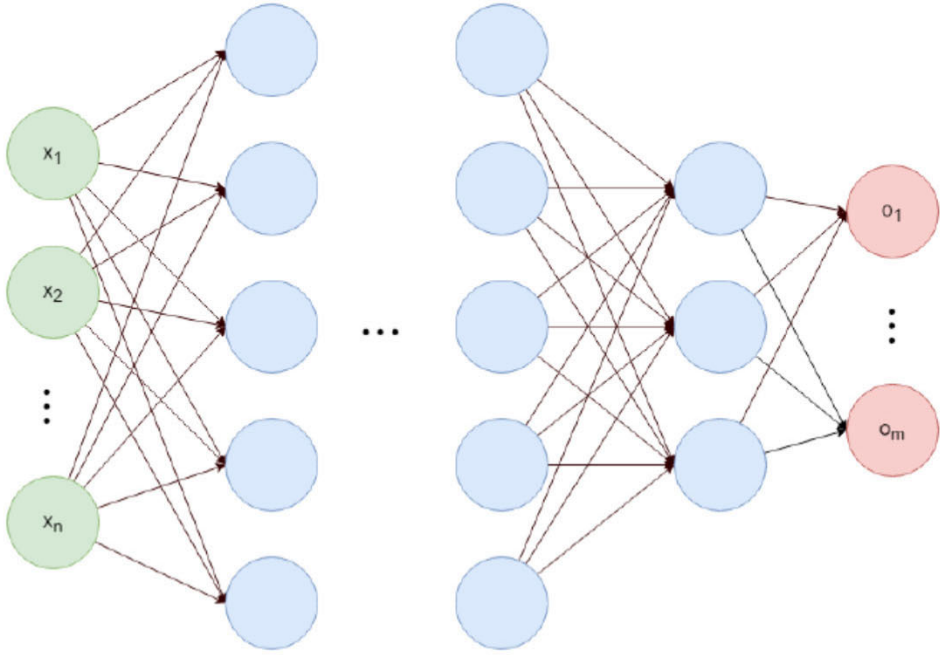
First idea:

- Use analytical approach.
- Write down derivatives on paper.
- Find the close form of $\nabla_W \mathcal{J}$ (if it is possible to do so).
- Implement this gradient as a function to work with.

- Pros:
  - Fast
  - Exact

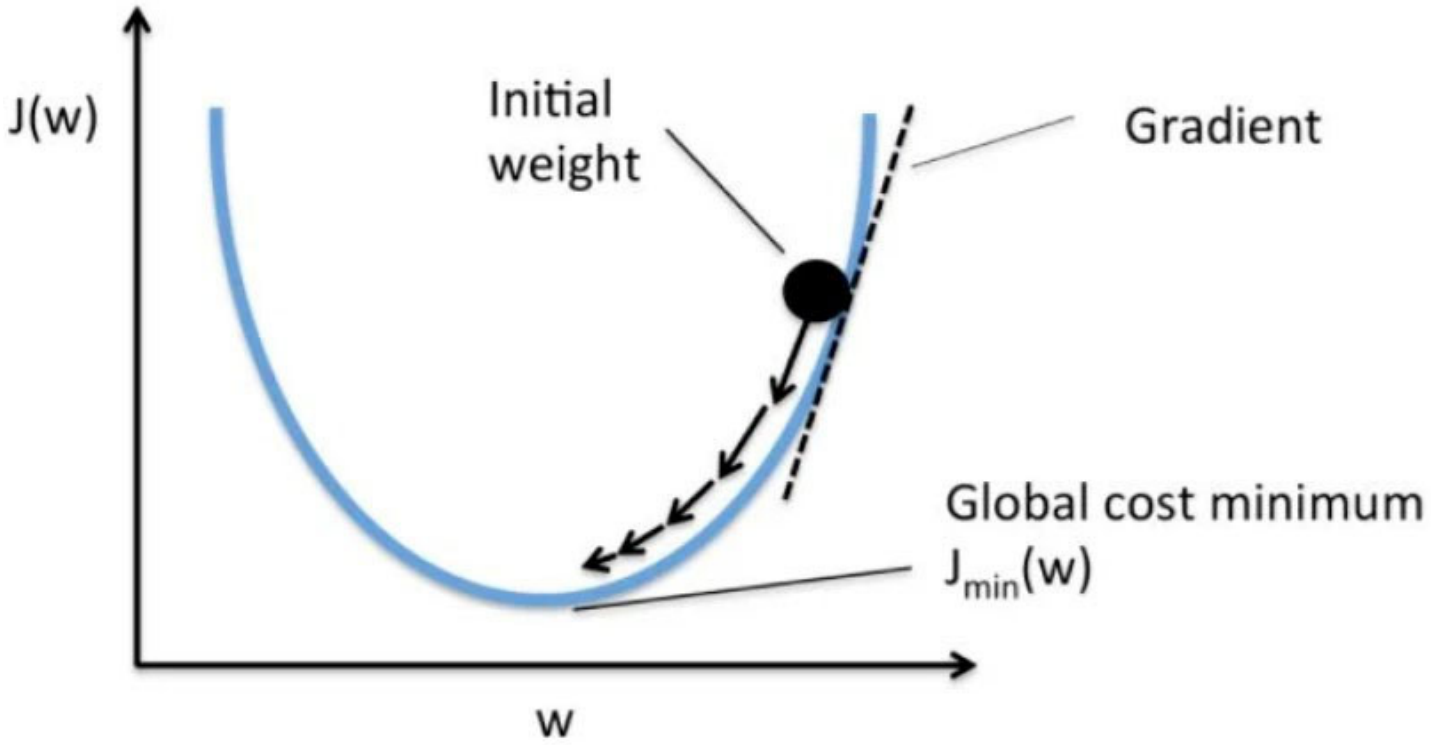
- Cons:
  - Need to rewrite calculation for different architectures

The problem of first idea



$$\boldsymbol{o} = \boldsymbol{a}^{[L]} = \sigma^{[L]}\left(\boldsymbol{b}^{[L]} + (W^{[L]})^T \sigma^{[L-1]}\left(\cdots \sigma^{[1]}\left(\boldsymbol{b}^{[1]} + (W^{[1]})^T \boldsymbol{x}\right)\cdots\right)\right)$$

Gradient Descent



Gradient Descent

Gradient Descent: Minimizes the loss function by updating weights based on the gradient.

Weight Update Rule:

$$w_{\text{new}} = w_{\text{old}} - \eta \cdot \frac{\partial L}{\partial w}$$

Where:

- η is the learning rate (step size).
- $\frac{\partial L}{\partial w}$ is the gradient of the loss function with respect to w.

Example: Gradient Descent and Updating Weights

Example Problem:

- Initial weight: $w_0 = 2$
- Learning rate: $\eta = 0.1$
- Loss function: $L(w) = (y - wx)^2$
  Example: For $x = 3$, $y = 10$, and $w_0 = 2$,

Gradient Calculation:

$$\frac{\partial L}{\partial w} = -2x(y - wx)$$

$$\frac{\partial L}{\partial w} = -24, \quad w_{\text{new}} = 4.4$$

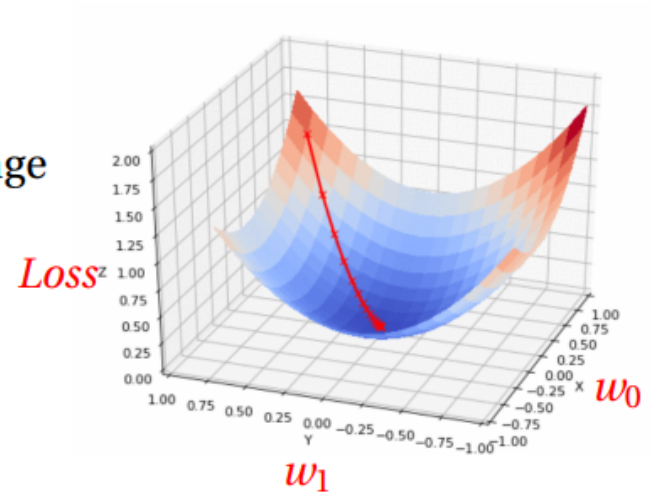
Gradient Descent: Formula and Process

Weight Update Formula:

$$w_{\text{new}} = w_{\text{old}} - \eta \cdot \frac{\partial L}{\partial w}$$

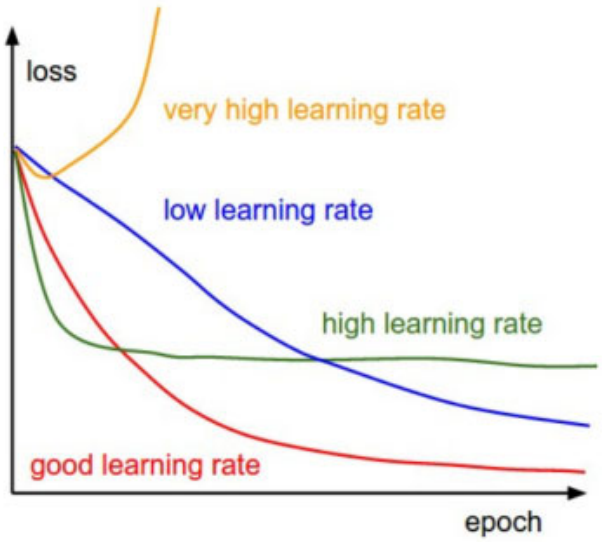
Steps in Gradient Descent:

- Compute the gradient of the loss function.
- Update the weights using the update rule.
- Repeat until convergence.
- Image adapted from Data Science Stack Exchange



Identifying an Optimal Learning Rate

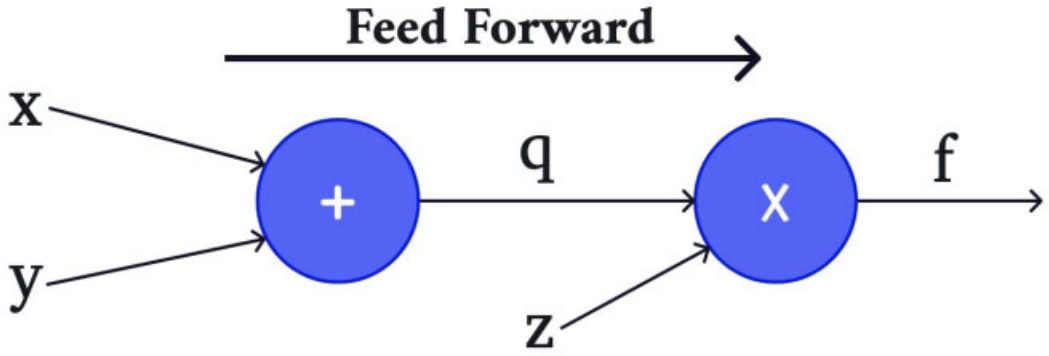
- Look for a **smooth, gradual** decrease in loss over time.
- Very low learning rate -> slow convergence
- Very high learning rate -> erratic fluctuations
- Image adapted from Towards Data Science: Understanding Learning Rates and How It Improves Performance in Deep Learning



Computing derivation (simple example)

Function:

$$f(x, y, z) = (x + y)z$$



Forward path

Function:

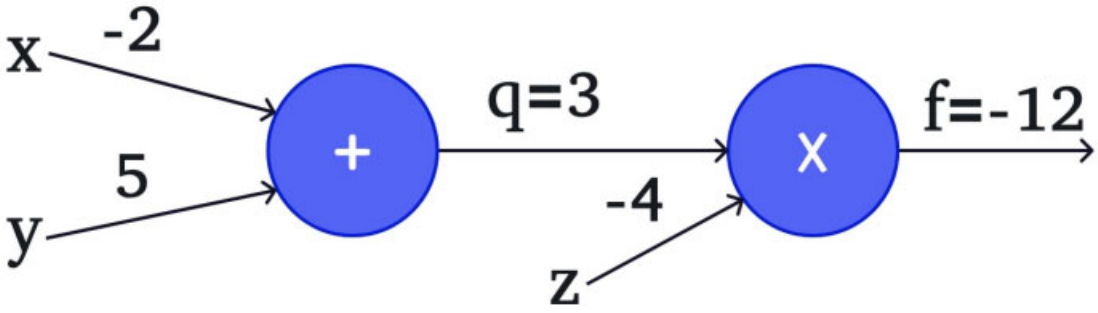
$$f(x, y, z) = (x + y)z$$

Example:

$$x = -2, \quad y = 5, \quad z = -4$$

Steps:

- $\mathbf{q = x + y = 3}$
- $\mathbf{f = q \times z = -12}$



Finite difference method

To approximate the gradient $\frac{\partial L}{\partial w_i}$:

- Change w_i by a small value ϵ.
- Approximate the gradient as:

$$\frac{\partial L}{\partial w_i} \approx \frac{L(w_i + \epsilon) - L(w_i)}{\epsilon}$$

- Simple but inefficient.

Problem:
Complexity is $\Theta(n^2)$ for n weights, which is slow for large models.

Back Propagation

A more efficient method:

- Use the chain rule to compute all gradients in one backward pass.
- This method avoids changing each weight separately.

Advantages:

- Complexity is reduced to $\Theta(n)$.
- Much faster, especially for large neural networks.

Chain rule

The **chain rule** helps us find the gradient of a function that is composed of other functions.

Example:

$$z = f(g(x))$$

- f is a function of $g(x)$
- $g(x)$ is a function of x

To find $\frac{\partial z}{\partial x}$, we use the chain rule:

$$\frac{\partial z}{\partial x} = \frac{\partial f}{\partial g} \times \frac{\partial g}{\partial x}$$

This means we multiply the gradient of the outer function by the gradient of the inner function.

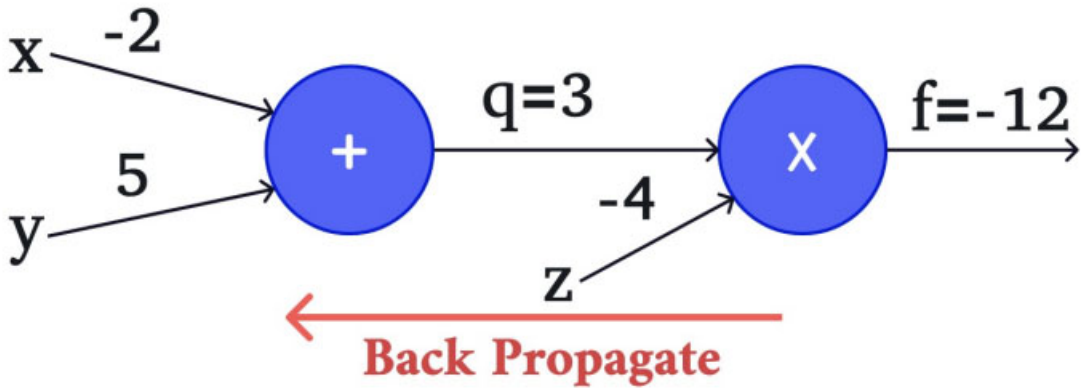
Back propagation (simple example)

Function:

$f(x, y, z) = (x + y)z$

Example:

$x = -2, \quad y = 5, \quad z = -4$



Back propagation (simple example)

Function:

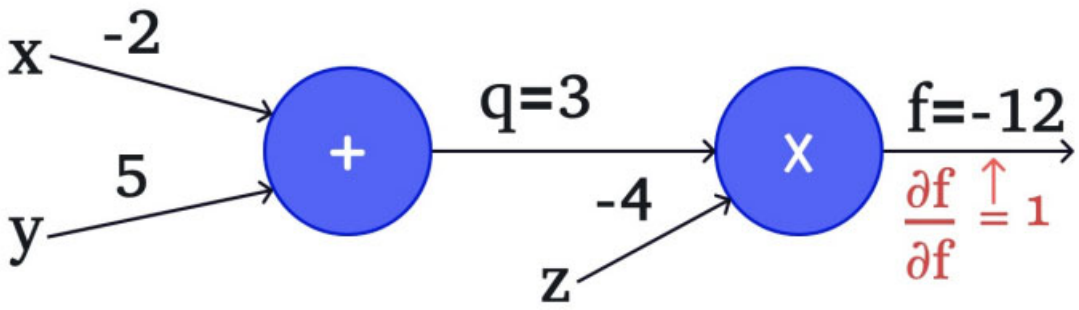
$$f(x, y, z) = (x + y)z$$

Example:

$$x = -2, \quad y = 5, \quad z = -4$$

Step 1:

$$\frac{\partial f}{\partial f} = 1$$



Back propagation (simple example)

Function:

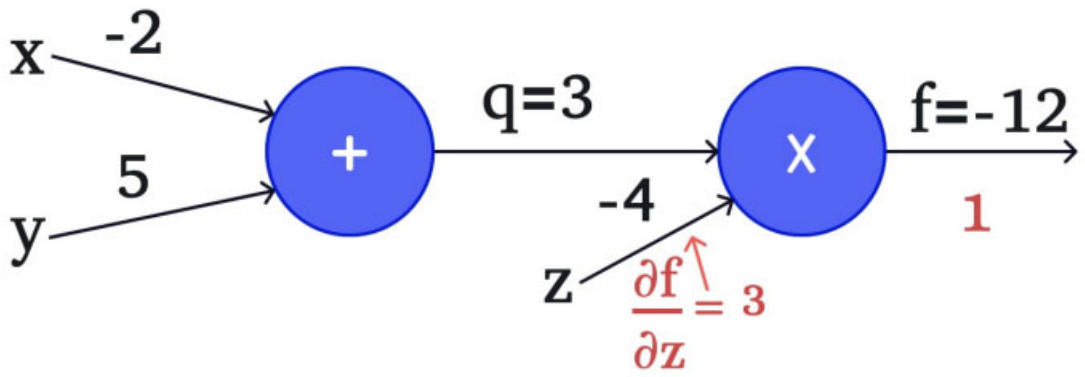
$$f(x, y, z) = (x + y)z$$

Example:

$$x = -2, \quad y = 5, \quad z = -4$$

Step 2:

$$f = qz, \quad \frac{\partial f}{\partial z} = q = 3$$



Back propagation (simple example)

Function:

$$f(x, y, z) = (x + y)z$$

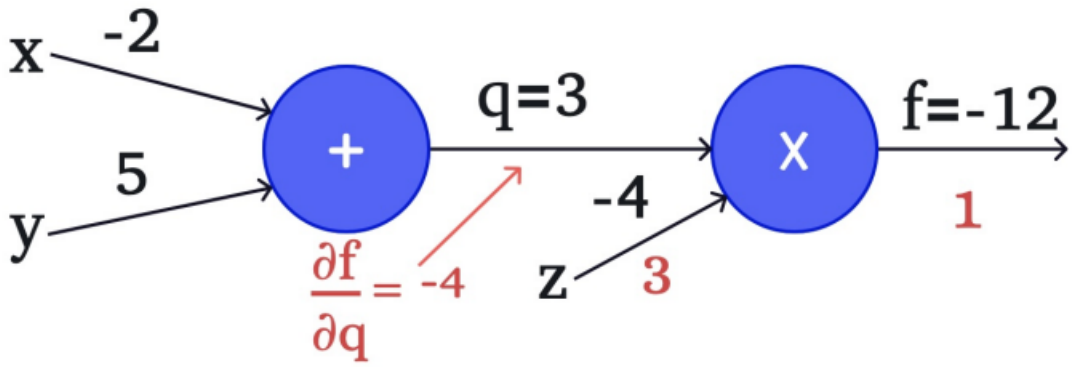
Example:

$$x = -2, \quad y = 5, \quad z = -4$$

Step 2:

$$f = qz, \quad \frac{\partial f}{\partial z} = q = 3$$

$$\frac{\partial f}{\partial q} = z = -4$$



Back propagation (simple example)

Function:

$$f(x, y, z) = (x + y)z$$

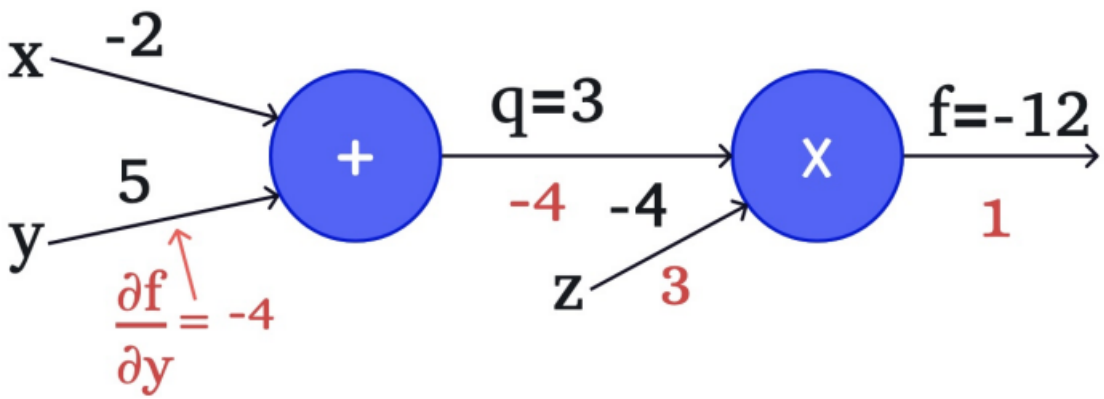
Example:

$$x = -2, \quad y = 5, \quad z = -4$$

Step 3:

$$q = x + y, \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1$$

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial y} = -4 \cdot 1 = -4$$



Function:

$$f(x, y, z) = (x + y)z$$

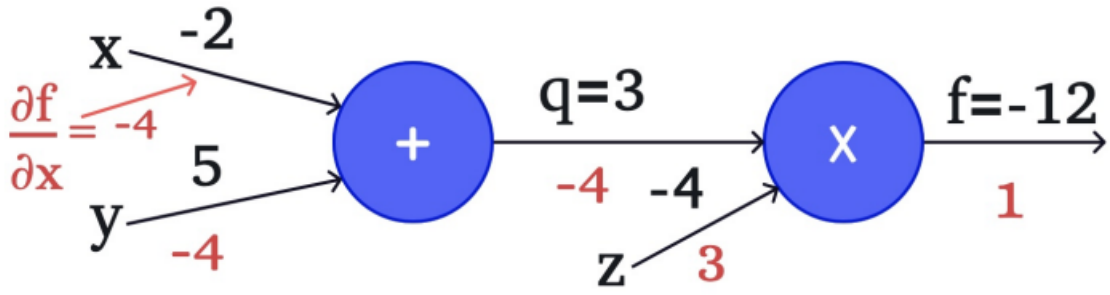
Example:

$$x = -2, \quad y = 5, \quad z = -4$$

Step 3:

$$q = x + y, \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1$$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x} = -4 \cdot 1 = -4$$



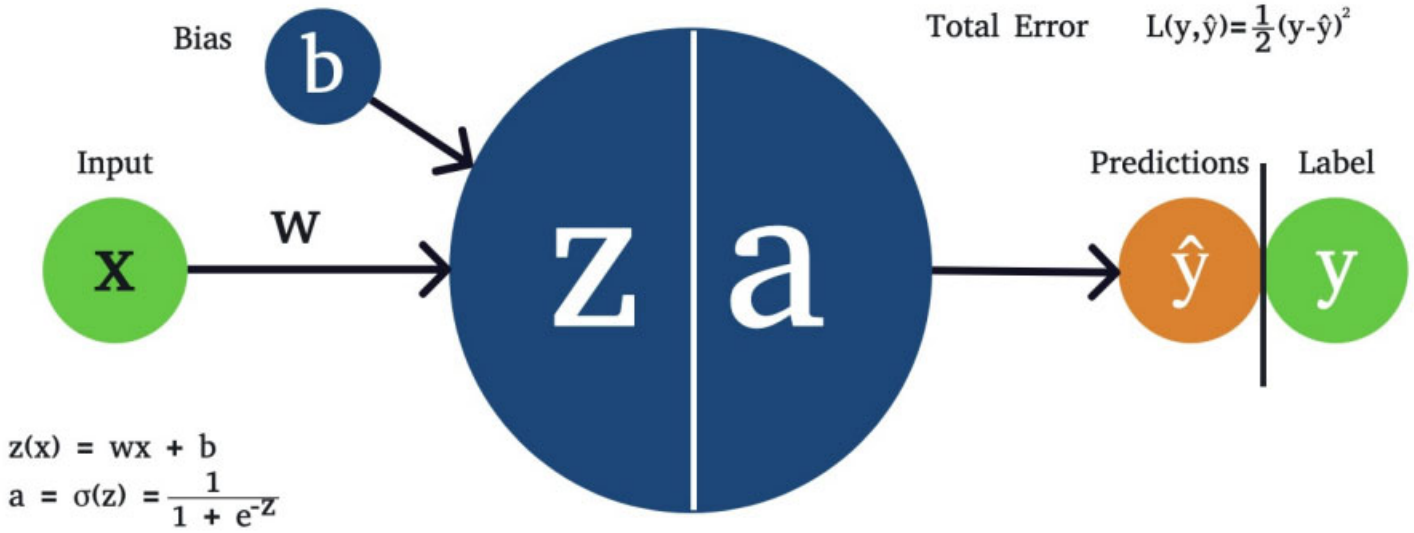
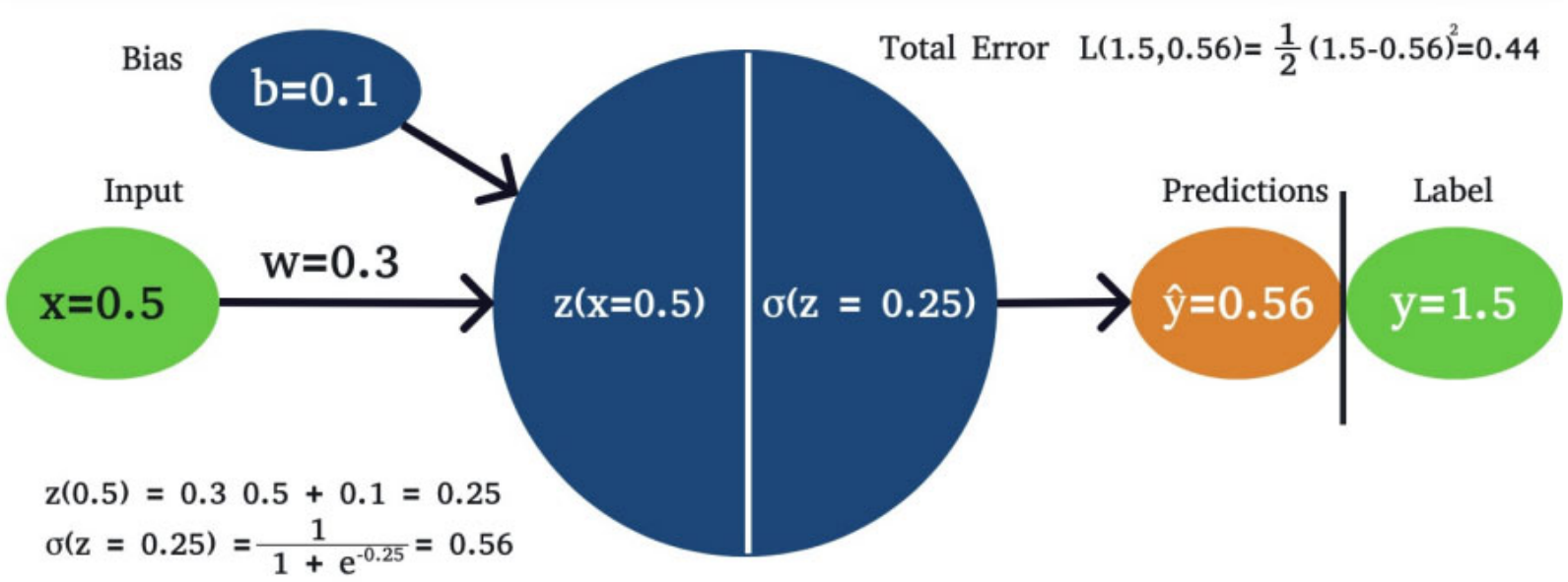
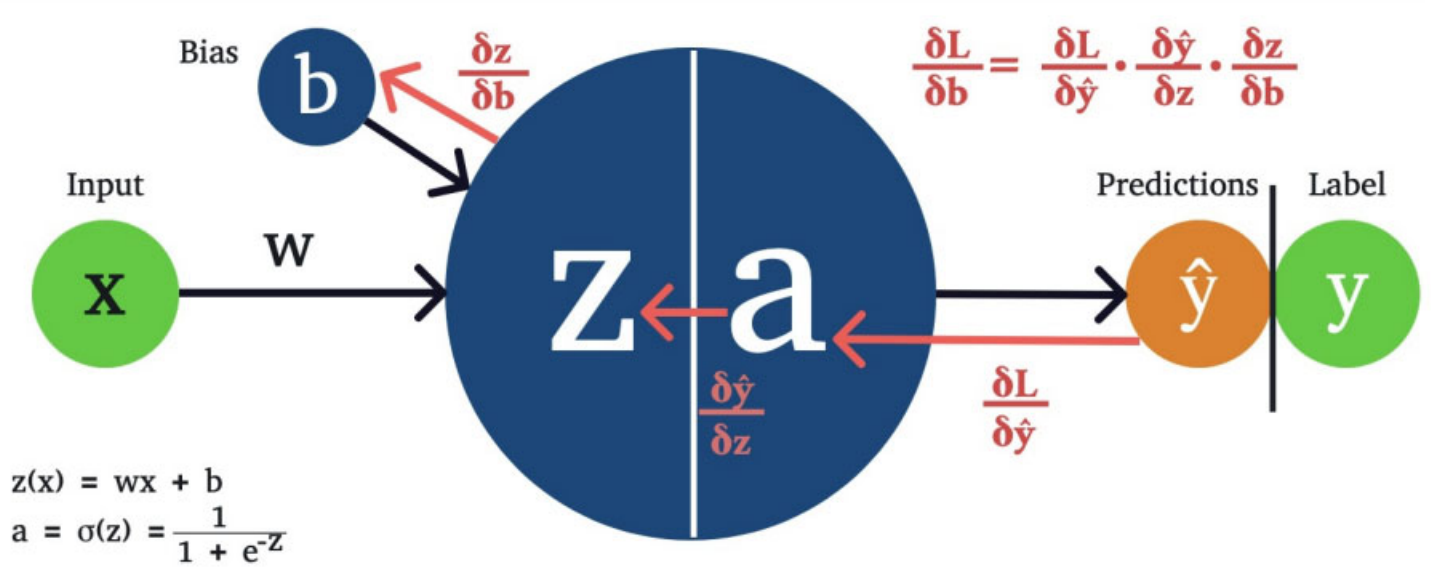


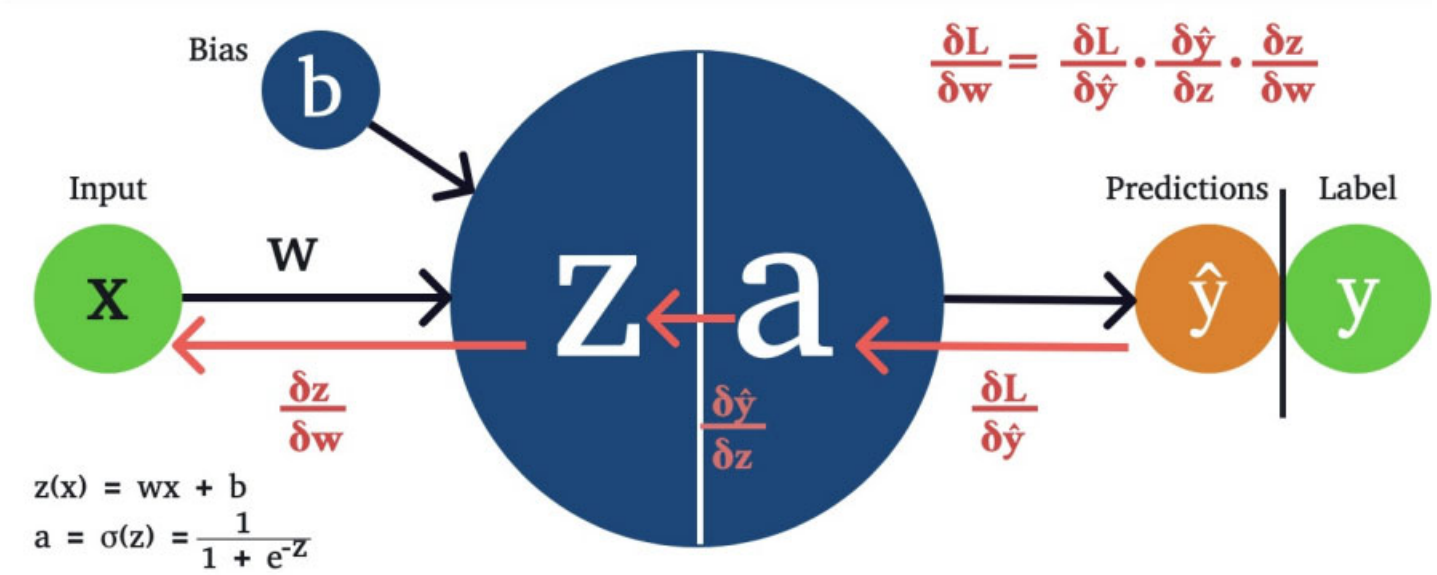
Image adapted from "Deep learning backpropagation" Web article

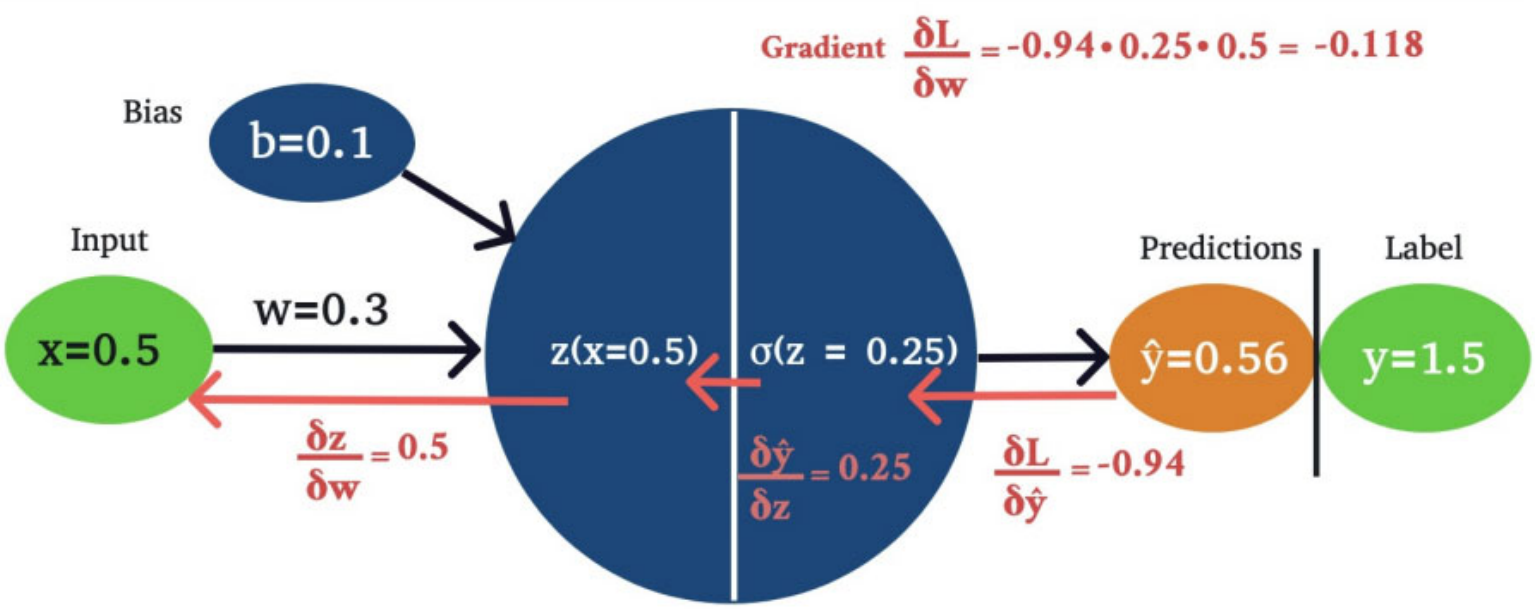
Forward pass



Backward pass




Bias
b
Input
x
w
z
a
Predictions
ŷ
Label
y
$\frac{\delta L}{\delta w} = \frac{\delta L}{\delta \hat{y}} \cdot \frac{\delta \hat{y}}{\delta z} \cdot \frac{\delta z}{\delta w}$
$\frac{\delta z}{\delta w}$
$\frac{\delta \hat{y}}{\delta z}$
$\frac{\delta L}{\delta \hat{y}}$
$z(x) = wx + b$
$a = \sigma(z) = \frac{1}{1 + e^{-z}}$


Gradient $\frac{\delta L}{\delta w} = -0.94 \cdot 0.25 \cdot 0.5 = -0.118$
Bias
b=0.1
Input
x=0.5
w=0.3
z(x=0.5)
σ(z = 0.25)
Predictions
ŷ=0.56
Label
y=1.5
$\frac{\delta z}{\delta w} = 0.5$
$\frac{\delta \hat{y}}{\delta z} = 0.25$
$\frac{\delta L}{\delta \hat{y}} = -0.94$