Neural Networks

Fatemeh Mansoori

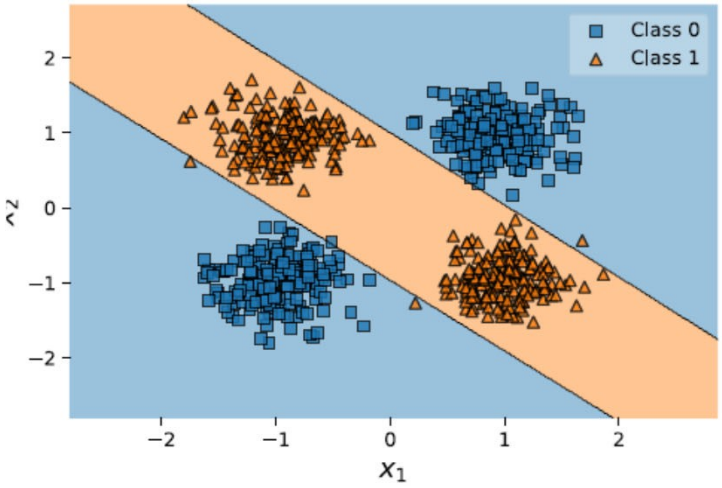
University of Isfahan

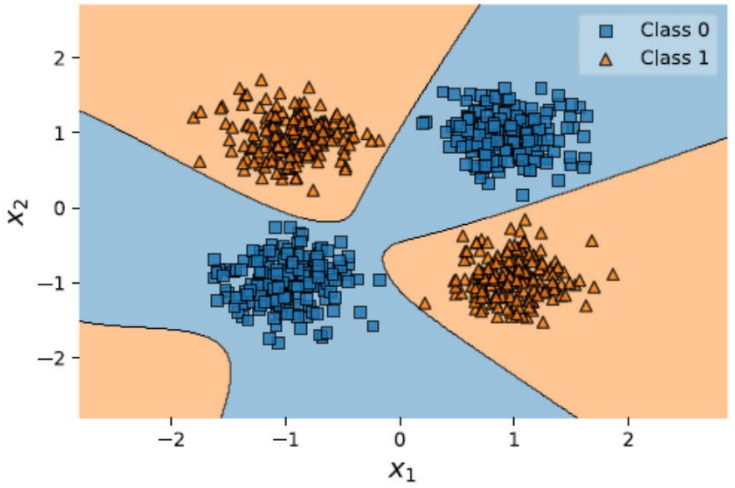
Some of the slide are based on slides from the machine learning course by sharifi zarchi

Limitation of perceptron

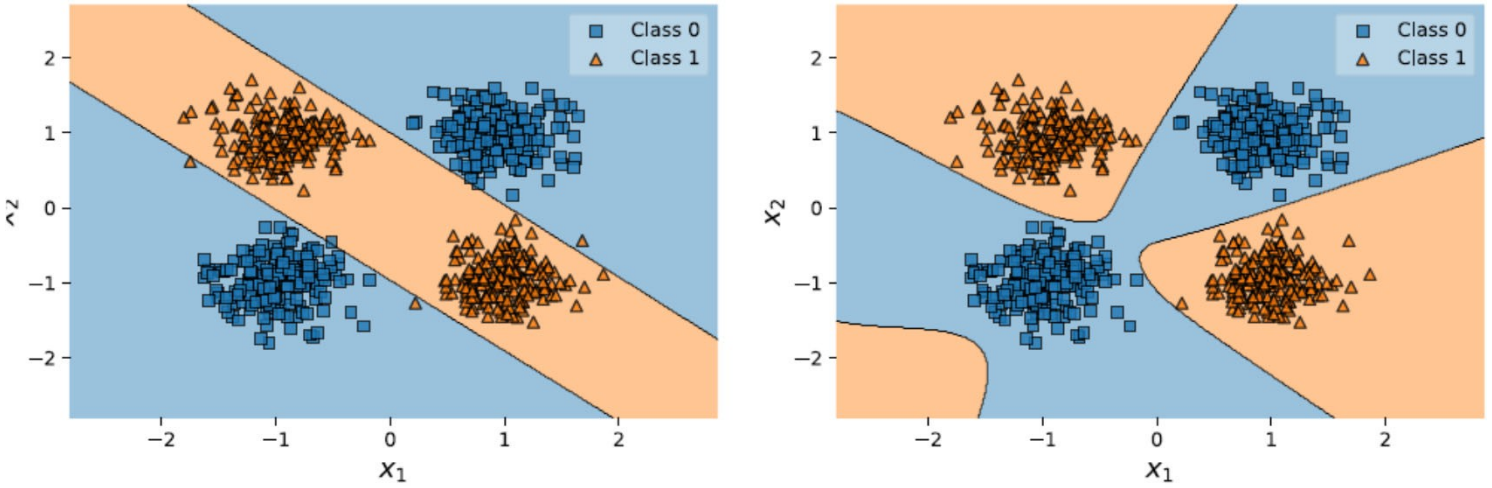
- XOR can not be represented by perceptron
- We need a deeper network
- No one knew how to train deeper networks

Why can’t a perceptron represent XOR?



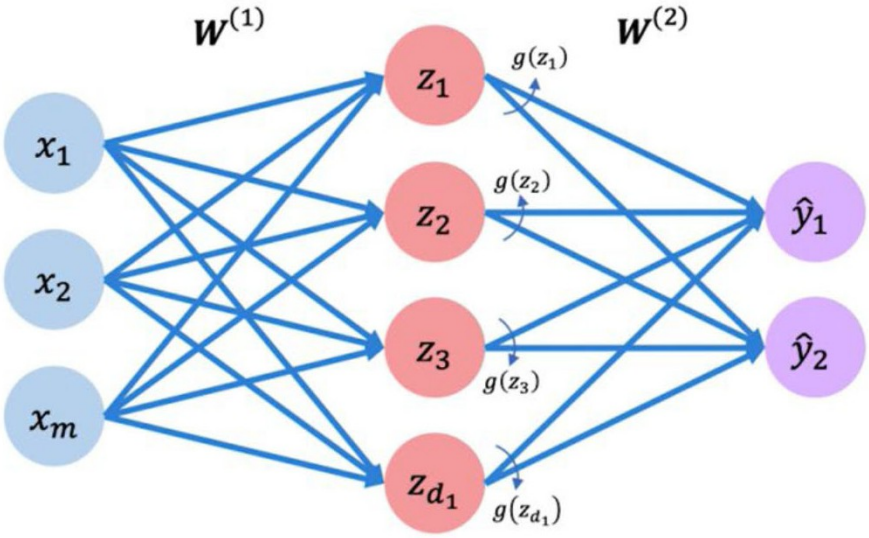


Multilayer Neural Networks Can Solve XOR Problems



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

Neural Network with 1 Hidden Layer

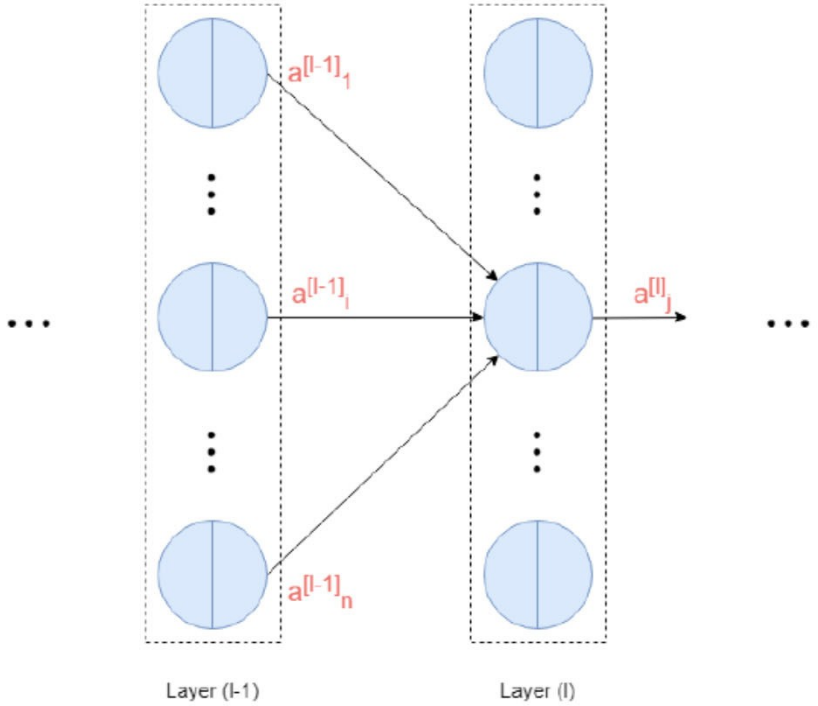


$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j \, w_{j,i}^{(1)} \qquad \hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} g(z_j) \, w_{j,i}^{(2)}\right)$$

Standard notations for MLP

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

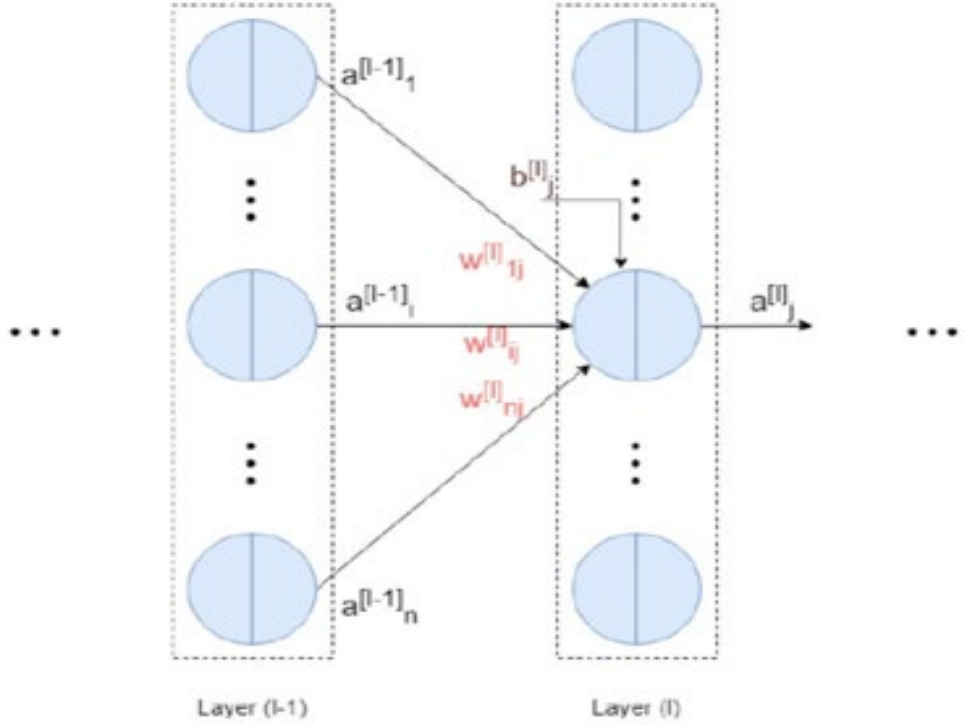
$\boldsymbol{a}^{[l]}$: layer l output in vector form



MLP notation

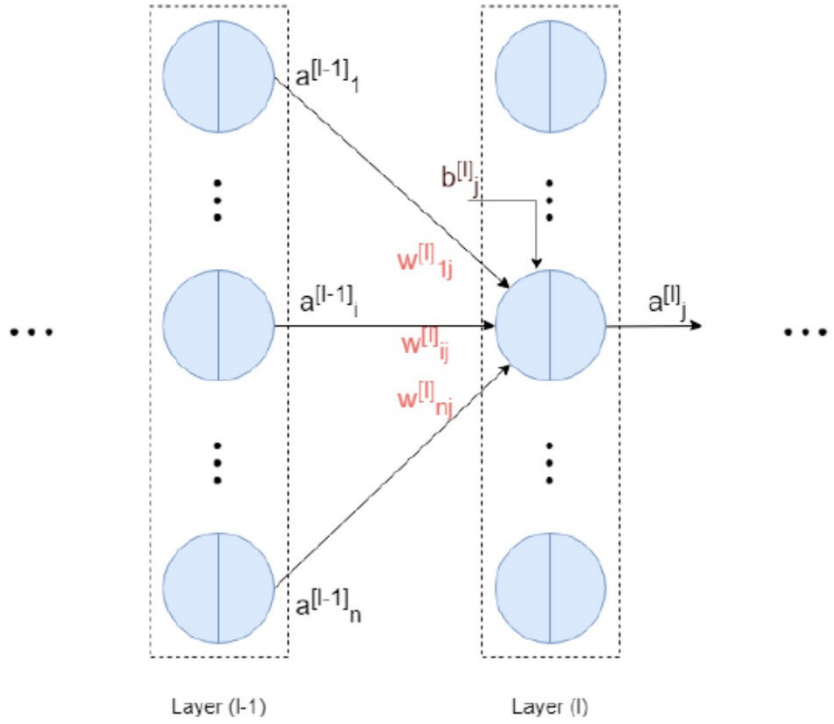
$b_i^{[l]}$: i-th neuron bias in layer l

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



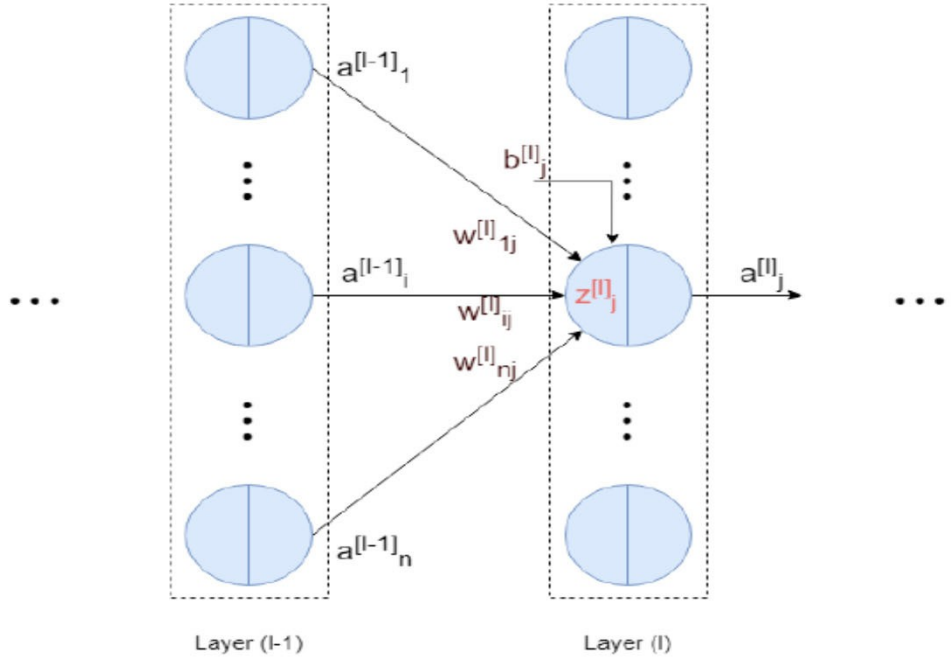
MLP notation

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



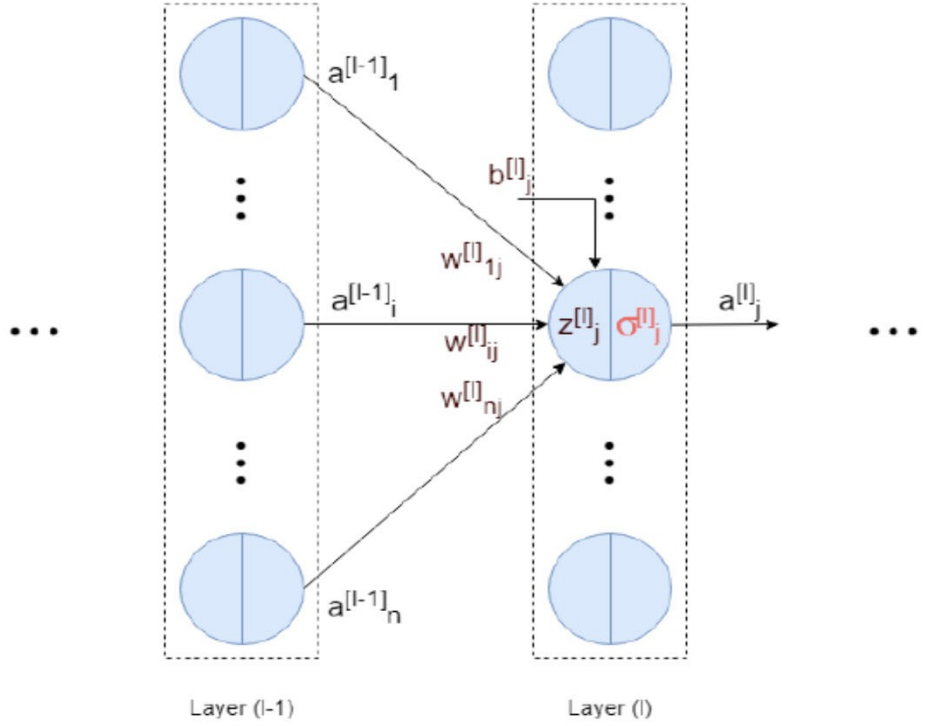
MLP notation

- $z_j^{[l]}$: 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

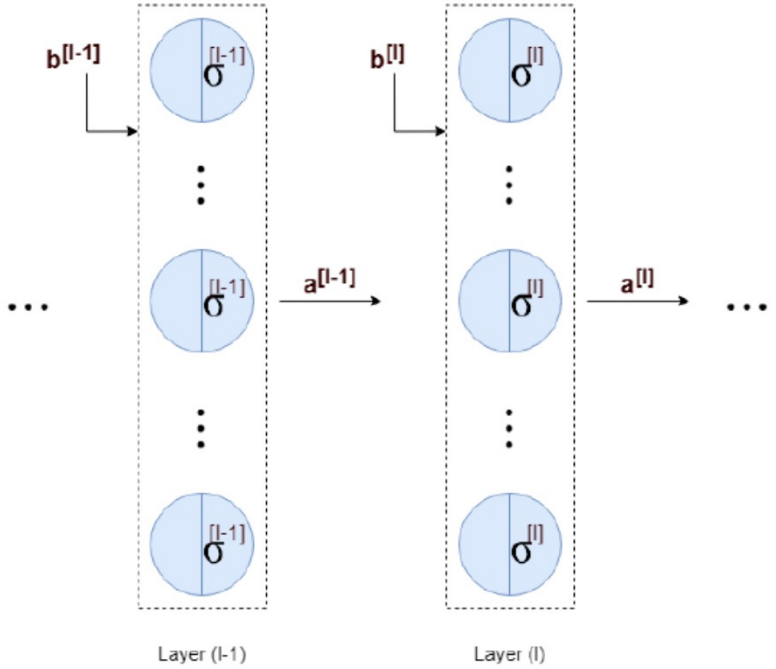
$\sigma_j^{[l]}$: j-th neuron activation function in layer l



MLP notation

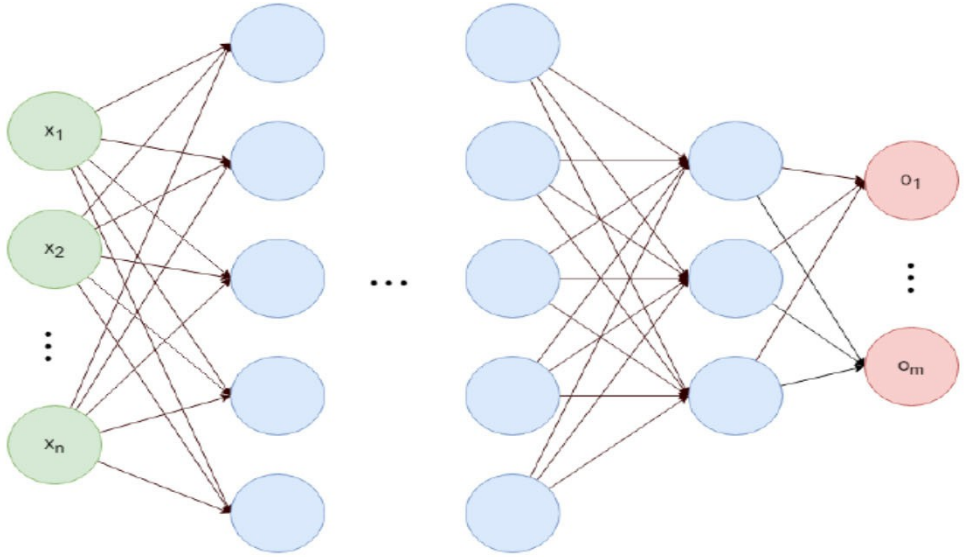
- If all neurons in one layer have the same activation function then :

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



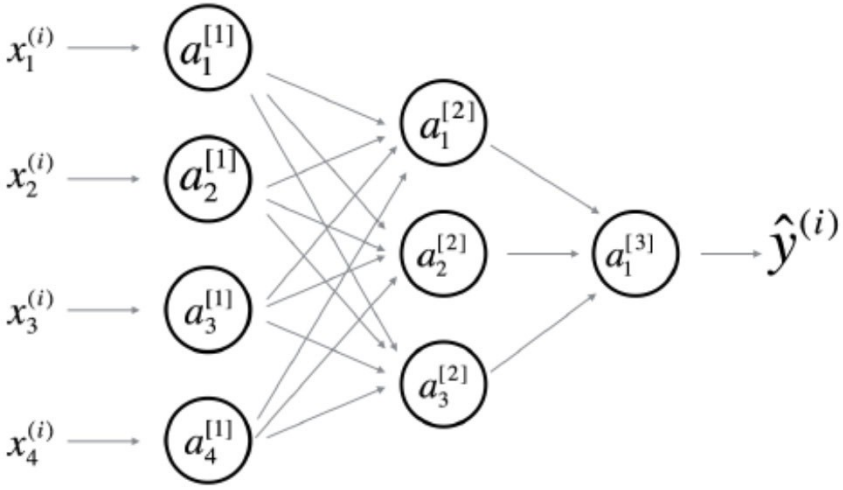
MLP notation

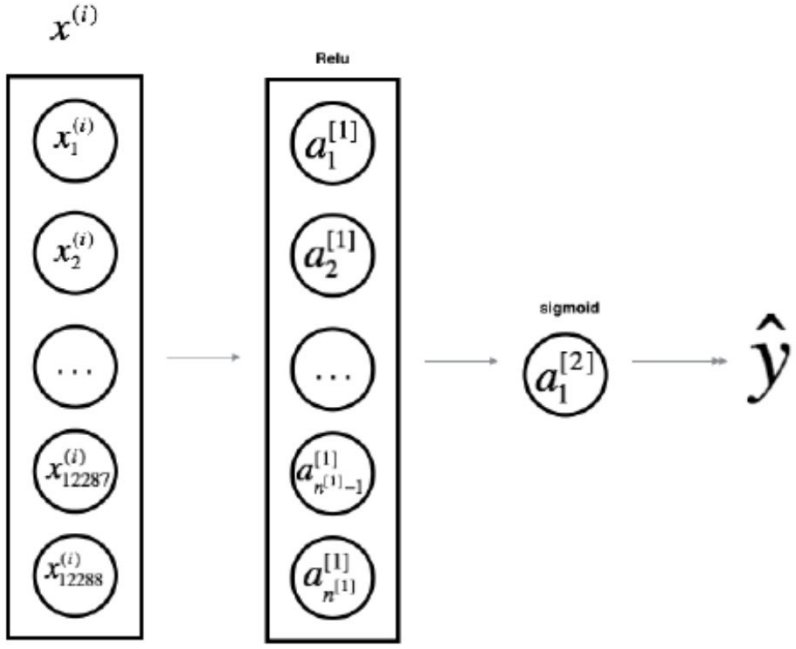
- For a network with L layer and x as its input we have :



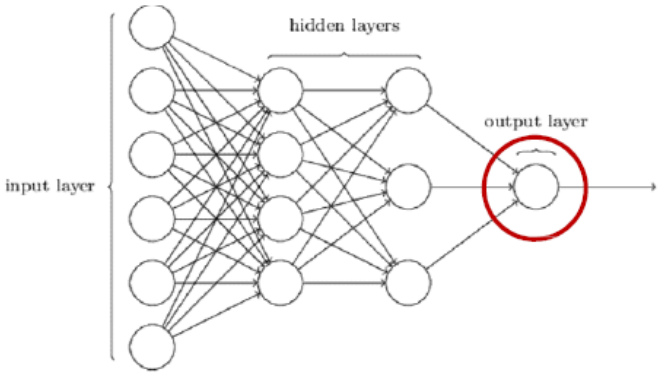
$$\boldsymbol{o} = \boldsymbol{a}^{[L]} = \sigma^{[L]}\left(\boldsymbol{b}^{[L]} + (W^{[L]})^T \sigma^{[L-1]}\left(\cdots \sigma^{[1]}\left(\boldsymbol{b}^{[1]} + (W^{[1]})^T \boldsymbol{x}\right)\cdots\right)\right)$$

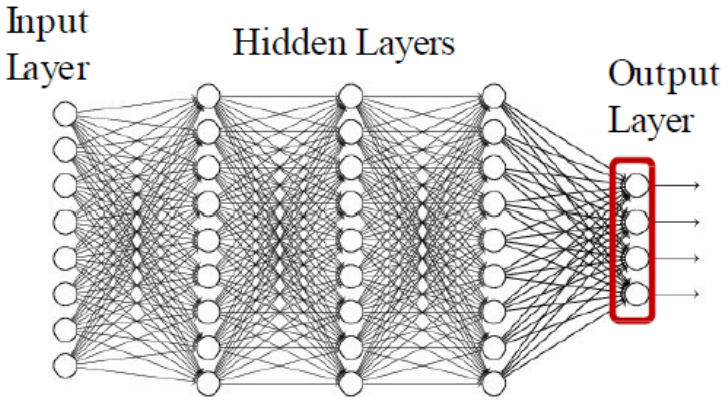
Deep Learning representations





Representing the output

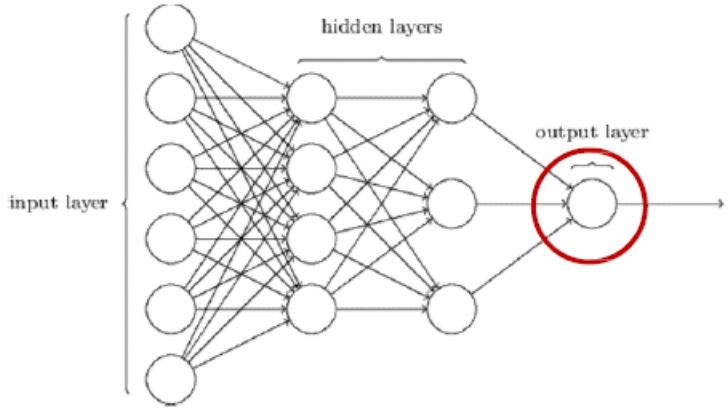




If the desired output is real-valued, no special tricks are necessary

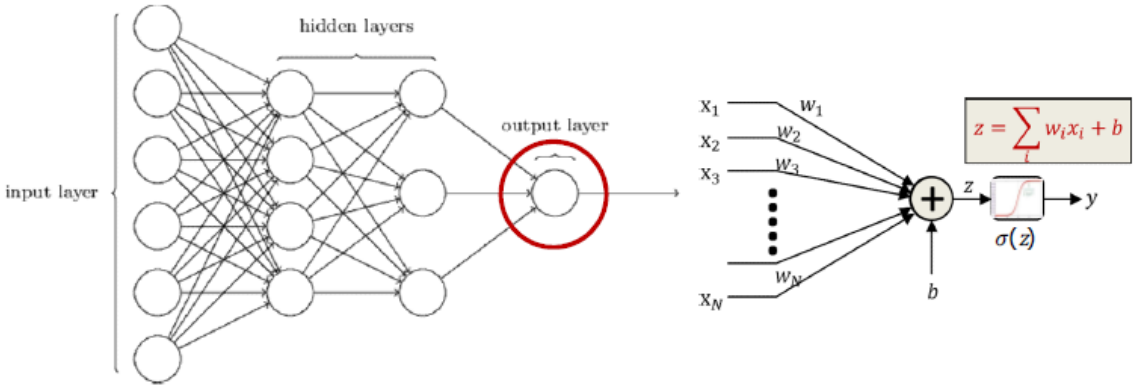
- Scalar Output : single output neuron
  - o = scalar (real value)
- Vector Output : as many output neurons as the dimension of the desired output
  - $\boldsymbol{o} = [o_1, o_2, \ldots, o_K]^T$ (vector of real values)

Representing the output



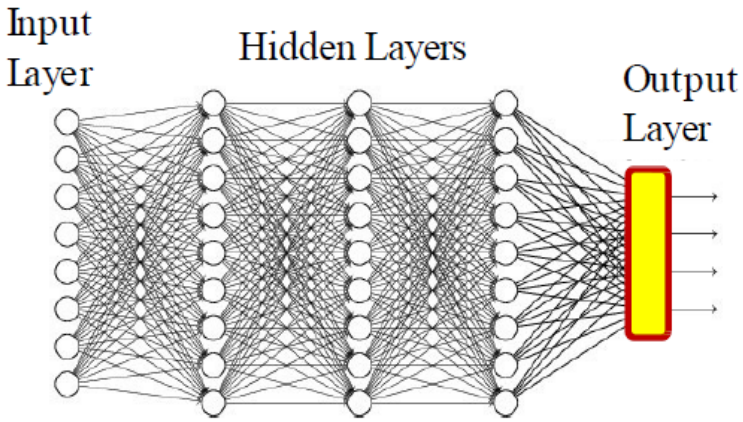
- If the desired output is binary (is this a cat or not), use a simple 1/0 representation of the desired output
  - 1 =YES it's a cat
  - 0 = NO it's not a cat.

Representing the output



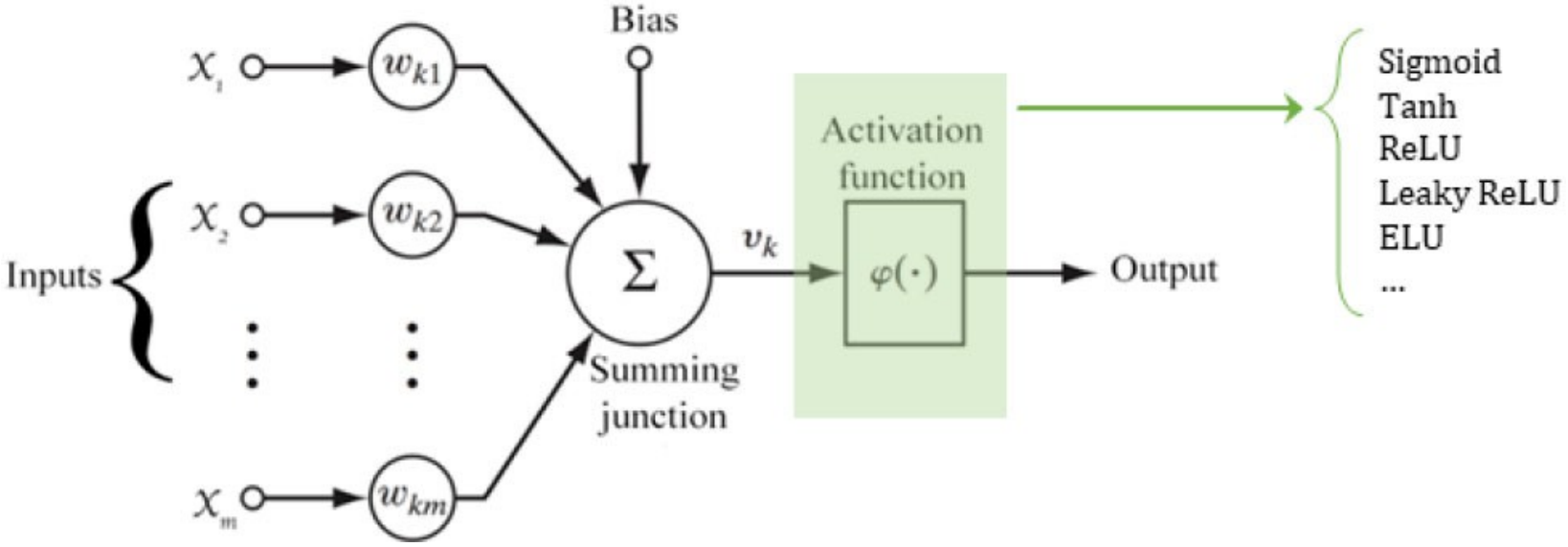
- If the desired output is binary, use a simple 1/0 representation of the desired output
- Output activation: Typically a sigmoid
  - Viewed as the probability $P(Y = 1|\boldsymbol{x})$ of class value 1
    - Indicating the fact that for actual data, in general a feature vector may occur for both classes,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

Multi class networks



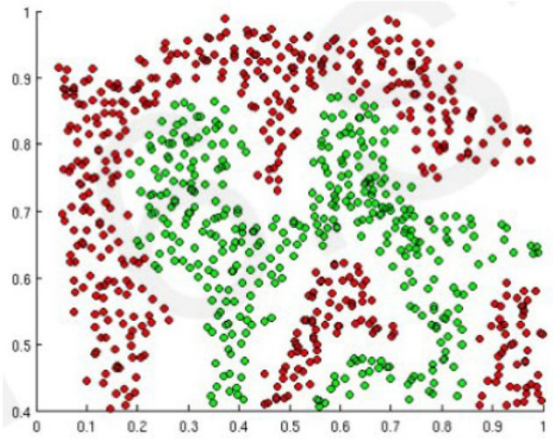
- For a multi-class classifier with K classes, the one-hot representation for the desired output y
  - The neural network's output too must ideally be binary (K-1 zeros and a single 1 in the right place)
- The network's output will be a probability vector
  - K probability values that sum to 1.

Activation function



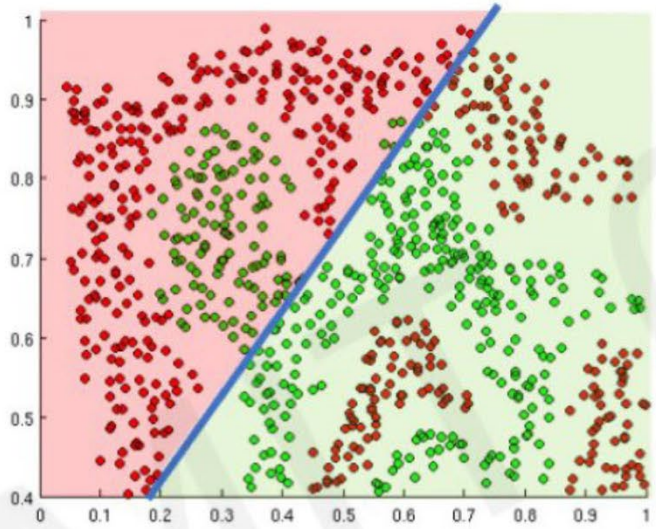
Importance of Activation Function

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

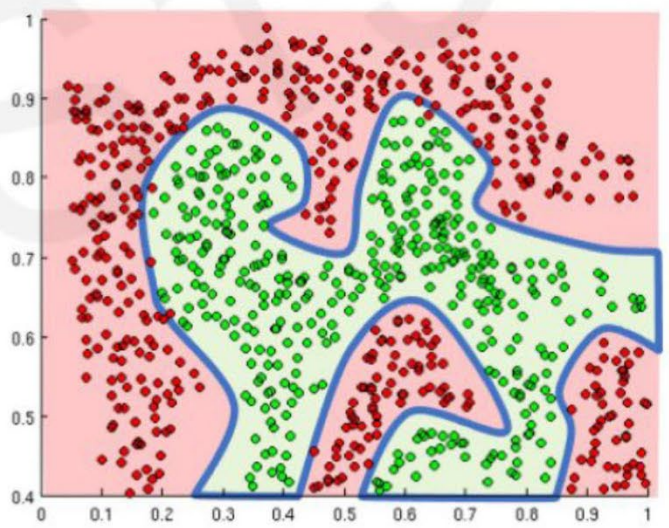


- What if we wanted to build a neural network to distinguish green vs red points?

Importance of Activation Function



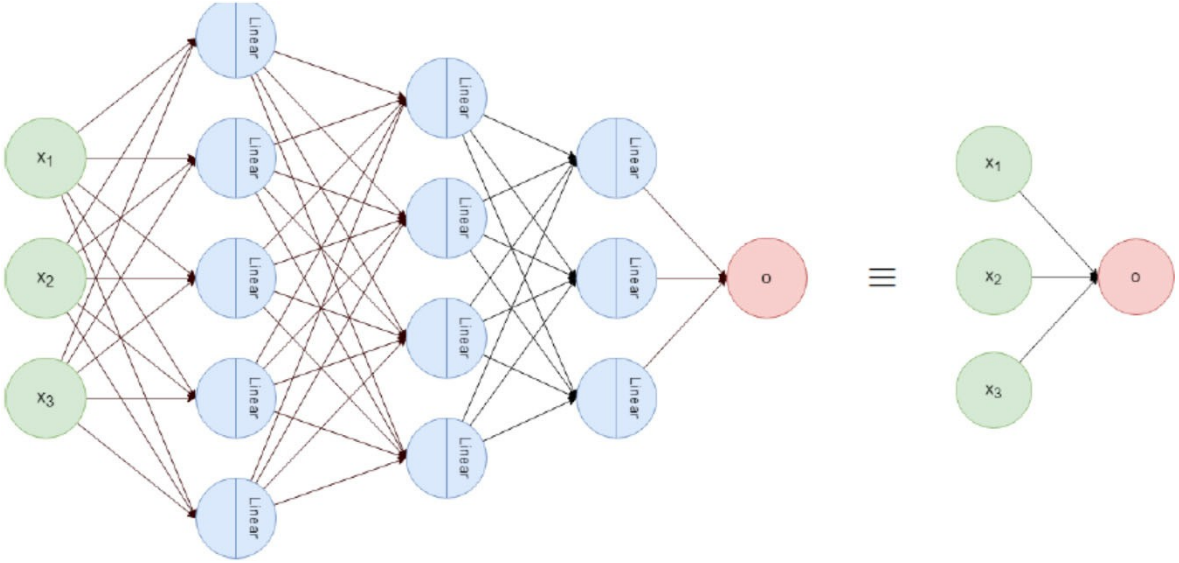
Linear activation functions produce linear decisions no matter the network size



Non-linearities allow us to approximate arbitrarily complex functions

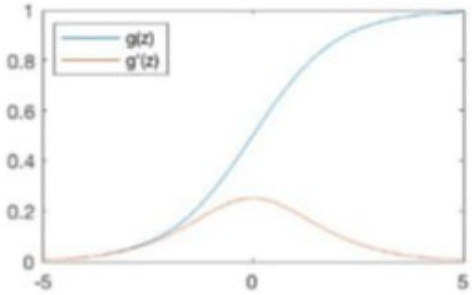
Activation Function of Hidden Layers

- 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

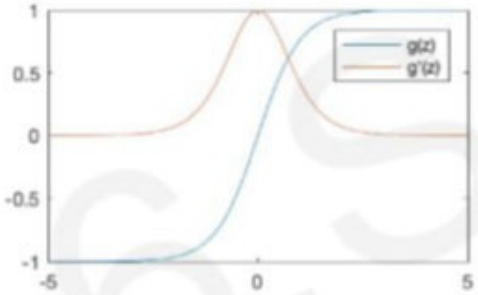
Sigmoid Function



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

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

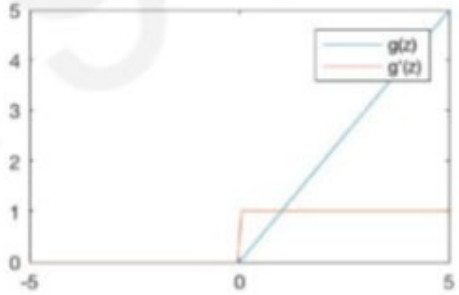
Hyperbolic Tangent



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

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

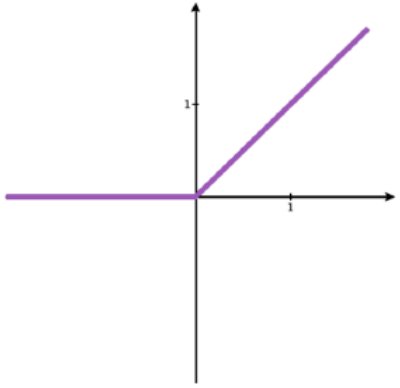
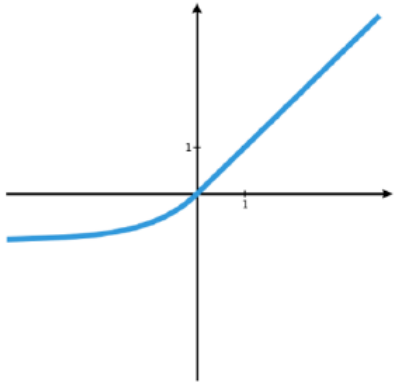
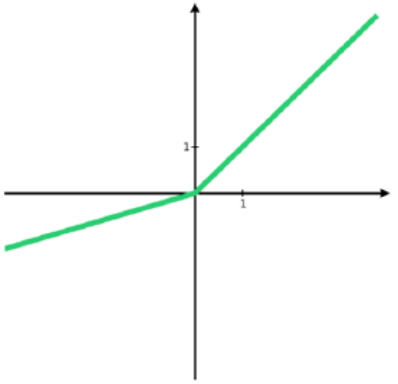
Rectified Linear Unit (ReLU)



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

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

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 ?

ReLU

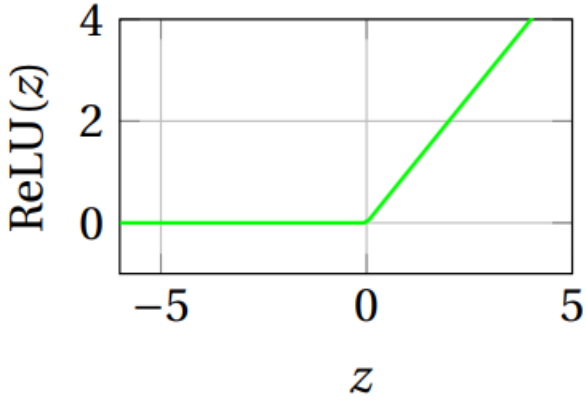
Characteristics of ReLU:

$$\text{ReLU}(z) = \max(0, z)$$

- Faster convergence: Efficient computation, especially for deep networks.

Advantages of ReLU:

- Does not saturate for positive values, helping to avoid the vanishing gradient problem.
- Computationally efficient (simpler than Sigmoid/Tanh).



ReLU

Limitation:

- **Dead ReLU Problem**: Neurons can become inactive during training, outputting 0 for all inputs if they receive negative values consistently.

Question:

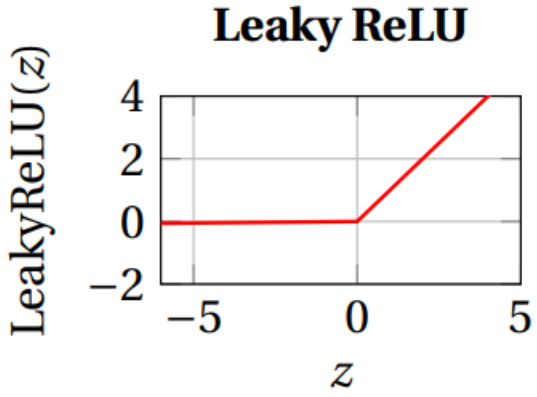
- Why does ReLU lead to faster training in deep networks?

Variant of ReLU

- Allows a small, non-zero gradient for negative inputs.

$$\text{LeakyReLU}(z) = \max(\alpha z, z), \quad \alpha = 0.01$$

- Helps prevent the "dead ReLU" problem, where neurons stop updating.



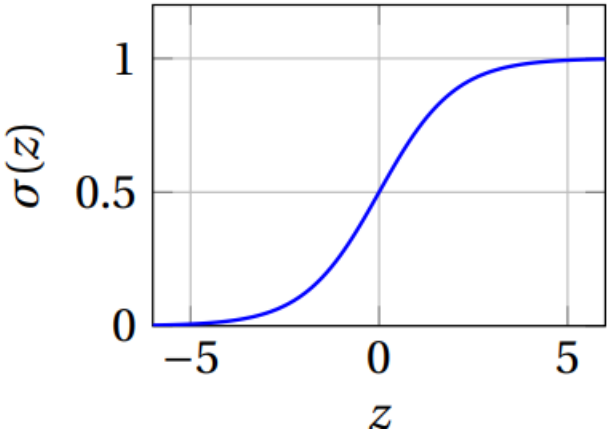
Leaky ReLU: Allows a small, non-zero gradient for negative inputs.

Sigmoid

Characteristics of Sigmoid:

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

- $\sigma'(z) = \sigma(z)(1 - \sigma(z))$
- Squashes the input between 0 and 1, which makes it useful in probabilistic interpretations (e.g., logistic regression).
- Often used in output layers for binary classification problems.

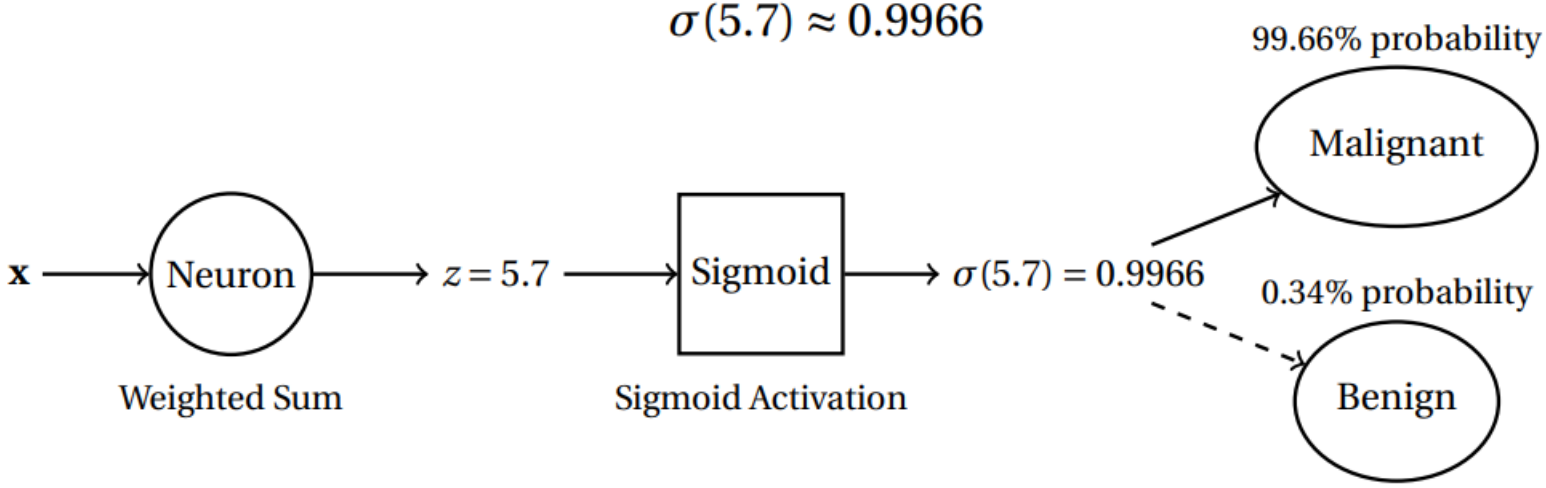


Classification : Tumor Detection

Sigmoid Activation: Useful for binary classification!

- Example: For $z = 5.7$,

$$\sigma(5.7) \approx 0.9966$$



Sigmoid

Limitations of Sigmoid:

- **Gradient Saturation:** When z is very large or very small, the gradient becomes nearly zero, causing slow learning (vanishing gradient problem).
- **Not Zero-Centered:** The output is not zero-centered, which can make optimization more difficult.

Question:

- Why does the vanishing gradient problem occur with Sigmoid during backpropagation? (To be discussed in more detail later)

Tanh

Characteristics of Tanh:

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

- Squashes input between -1 and 1, making it zero-centered (Balanced Updates → Reduced Bias in Gradient Descent → Faster Convergence)

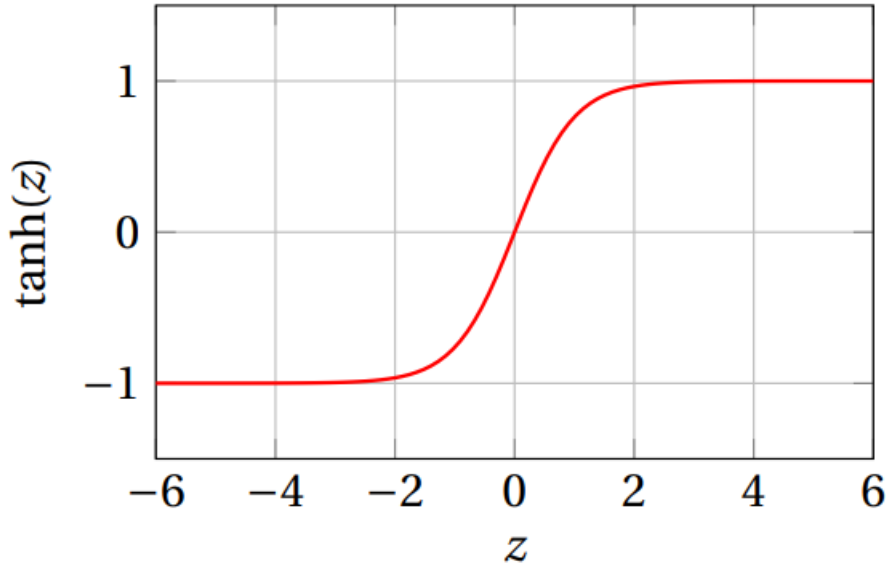
Advantages of Tanh:

- **Zero-Centered**: Output ranges from -1 to 1, making optimization easier.
- Better for **hidden layers** than Sigmoid due to zero-centered output.

Limitations:

- Similar saturation issues as Sigmoid: large input values push gradients towards zero (vanishing gradient problem).

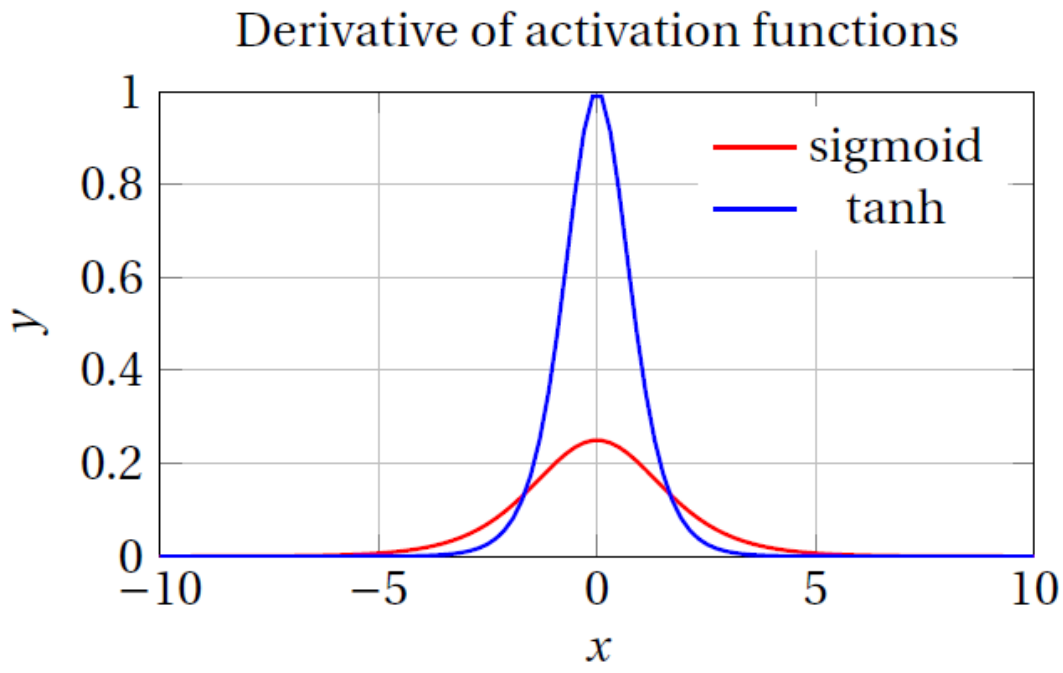
Tanh



Question:

- How does Tanh help with faster convergence compared to Sigmoid?

Comparison: sigmoid vs tanh

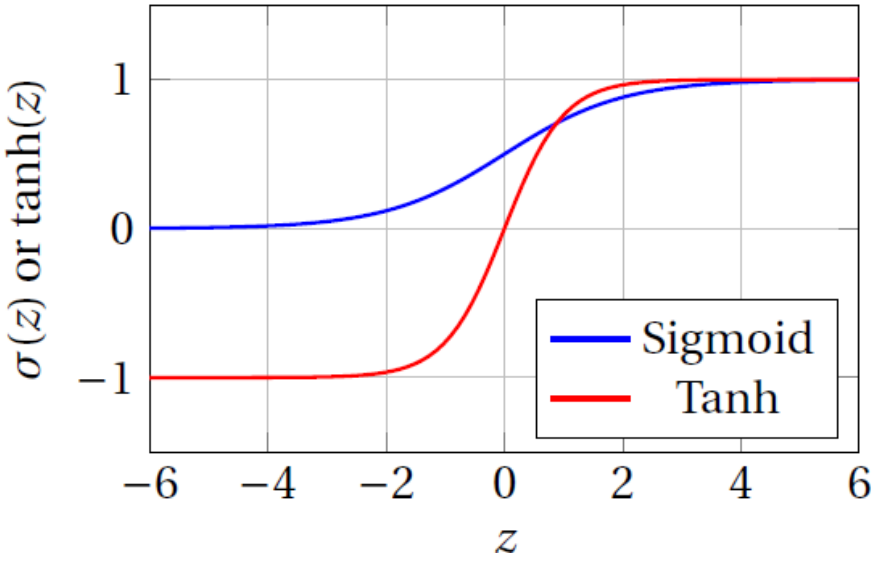


- The derivative of the Tanh function has a much steeper slope at $x = 0$, meaning it provides a larger gradient for backpropagation compared to the Sigmoid function.

Comparison: sigmoid vs tanh

Key Differences:

- **Sigmoid**: Maps input to [0, 1]. Output is not zero-centered.
- **Tanh**: Maps input to [-1, 1]. Output is zero-centered, leading to easier optimization.

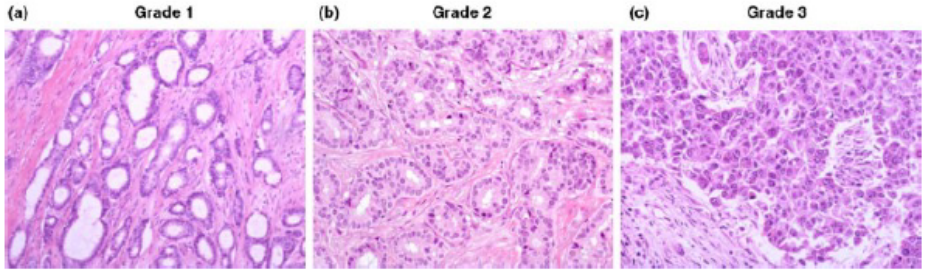


Problem: Multi class Tumor Classification

Scenario: We want to classify a tumor into one of three categories:

- **Class 0**: Benign
- **Class 1**: Malignant
- **Class 2**: Pre-cancerous

Goal: Given a set of tumor features, predict which class the tumor belongs to.
This is a **multi-class classification problem**, and we will use the **Softmax activation function** to assign probabilities to each class.



Microscopic images of tumor tissue, classified into grades representing different severity levels. Image adapted from Visual Analytics in Digital Computational Pathology

Softmax

In multi-class classification, the Softmax function is used to convert raw outputs (logits) into probabilities for each class.

Softmax Function:

$$P(y = i|X) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Where:

- $P(y = i|X)$ is the probability of the sample X belonging to class i.
- z_i is the raw output (logit) for class i.
- K is the number of classes (e.g., benign, malignant, pre-cancerous).

The Softmax function ensures that the sum of the probabilities for all classes is 1, and the class with the highest probability is chosen as the prediction.

Example : softmax

Consider a tumor with the following logits from a neural network:

- Logit for Benign (Class 0): $z_0 = 1.5$
- Logit for Malignant (Class 1): $z_1 = 0.8$
- Logit for Pre-Cancerous (Class 2): $z_2 = -0.5$

Step 1: Exponentiate the logits

$$e^{z_0} = e^{1.5} \approx 4.48, \quad e^{z_1} = e^{0.8} \approx 2.23, \quad e^{z_2} = e^{-0.5} \approx 0.61$$

Step 2: Compute the sum of exponentials

$$\text{Sum} = e^{z_0} + e^{z_1} + e^{z_2} = 4.48 + 2.23 + 0.61 = 7.32$$

Example : softmax

Step 3: Calculate Softmax probabilities for each class

$$P(\text{Benign}) = \frac{4.48}{7.32} \approx 0.612, \quad P(\text{Malignant}) = \frac{2.23}{7.32} \approx 0.305, \quad P(\text{Pre-Cancerous}) = \frac{0.61}{7.32} \approx 0.083$$

Step 4: Make a classification decision

- The highest probability is 0.612 for the **Benign** class.
- Therefore, the model predicts that the tumor is **Benign** (Class 0).

Softmax

Key Points:

- Softmax is used in the output layer for **multi-class classification**.
- It converts logits into a **probability distribution** across classes.
- The class with the highest probability is selected as the prediction.

Main Idea: Softmax ensures all outputs sum to 1, making it ideal for choosing one class out of multiple options.

Types of Loss Functions

- **Mean Squared Error (MSE):** Used in regression to minimize squared differences between predicted and true values.
- **Mean Absolute Error (MAE):** Minimizes absolute differences, also for regression tasks.
- **Binary Cross-Entropy:** Used for binary classification to compare predicted probabilities with binary labels.
- **Categorical Cross-Entropy:** For multi-class classification, comparing predicted probabilities across multiple classes.

Mean Squared Error

Definition:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

Characteristics:

- Amplifies larger errors due to squaring, making it sensitive to outliers.

Example of MSE calculation

Example:

- Predicted values: $\hat{y} = [4.2, 3.8, 5.1]$
- True values: $y = [5.0, 4.0, 4.9]$
- Calculation:

$$\text{MSE} = \frac{1}{3}\left[(5.0-4.2)^2 + (4.0-3.8)^2 + (4.9-5.1)^2\right]$$

$$= \frac{1}{3}\left[0.64 + 0.04 + 0.04\right] = \frac{0.72}{3} = 0.24$$

Mean Absolute Error (MAE)

Definition:

$$\mathrm{MAE} = \frac{1}{N}\sum_{i=1}^{N}\left|y^{(i)} - \hat{y}^{(i)}\right|$$

Characteristics:

- Provides a linear measure of error, treating all deviations equally.

Example of MAE Calculation

Example:

- Predicted values: $\hat{y} = [4.2, 3.8, 5.1]$
- True values: $y = [5.0, 4.0, 4.9]$
- Calculation:

$$\text{MAE} = \frac{1}{3}\left(|5.0 - 4.2| + |4.0 - 3.8| + |4.9 - 5.1|\right)$$

$$= \frac{1}{3}(0.8 + 0.2 + 0.2) = \frac{1.2}{3} \approx 0.4$$

Impact on model outputs and optimization

MSE (Mean Squared Error):

- Heavily penalizes large errors, promoting smoother outputs.
- Its quadratic gradient leads to faster convergence for large errors, but it can be sensitive to outliers.

MAE (Mean Absolute Error):

- Treats all errors uniformly, resulting in sharper outputs and better handling of outliers.
- Its constant gradient ensures stable optimization but can slow convergence with large errors.

Binary Classification Loss-Binary Cross Entropy

Binary Cross-Entropy:

$$\mathscr{L}_{\text{BCE}} = -\frac{1}{n}\sum_{i=1}^{n}\left[y^{(i)}\log(\hat{y}^{(i)}) + (1-y^{(i)})\log(1-\hat{y}^{(i)})\right]$$

Example:

- Predicted probabilities: $\hat{y} = [0.7, 0.3, 0.9]$
- True labels: $y = [1, 0, 1]$

$$\mathscr{L}_{\text{BCE}} = -\frac{1}{3}\left[1\cdot\log(0.7) + (1-1)\cdot\log(1-0.7)\right.$$

$$\left. +0\cdot\log(0.3) + (1-0)\cdot\log(1-0.3) + 1\cdot\log(0.9) + (1-1)\cdot\log(1-0.9)\right]$$

$$= -\frac{1}{3}\left(\log(0.7) + \log(0.7) + \log(0.9)\right) \approx -\frac{1}{3}\left(-0.357 + -0.357 + -0.105\right) \approx 0.273$$

Used to minimize the error between predicted probabilities and binary labels.

Categorical Cross-Entropy

Categorical Cross-Entropy Formula:

$$\mathscr{L}_{CCE} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{c=1}^{C} y_c^{(i)} \log(\hat{y}_c^{(i)})$$

One-Hot encoding

One-hot encoding represents categorical variables as binary vectors, with a 1 indicating the actual class and 0s elsewhere.

Example:

- Class 1: $[1, 0, 0]$
- Class 2: $[0, 1, 0]$
- Class 3: $[0, 0, 1]$

Example Calculation

Given:

- True labels (One-hot): Class 2, Class 1, Class 3
- Predicted probabilities:

$$\hat{y} = \underbrace{\begin{bmatrix} 0.1 & 0.7 & 0.2 \\ 0.6 & 0.3 & 0.1 \\ 0.1 & 0.6 & 0.3 \end{bmatrix}}_{\text{Classes}} \quad \text{Samples}$$

Solution

1. True labels:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

2. Calculation:

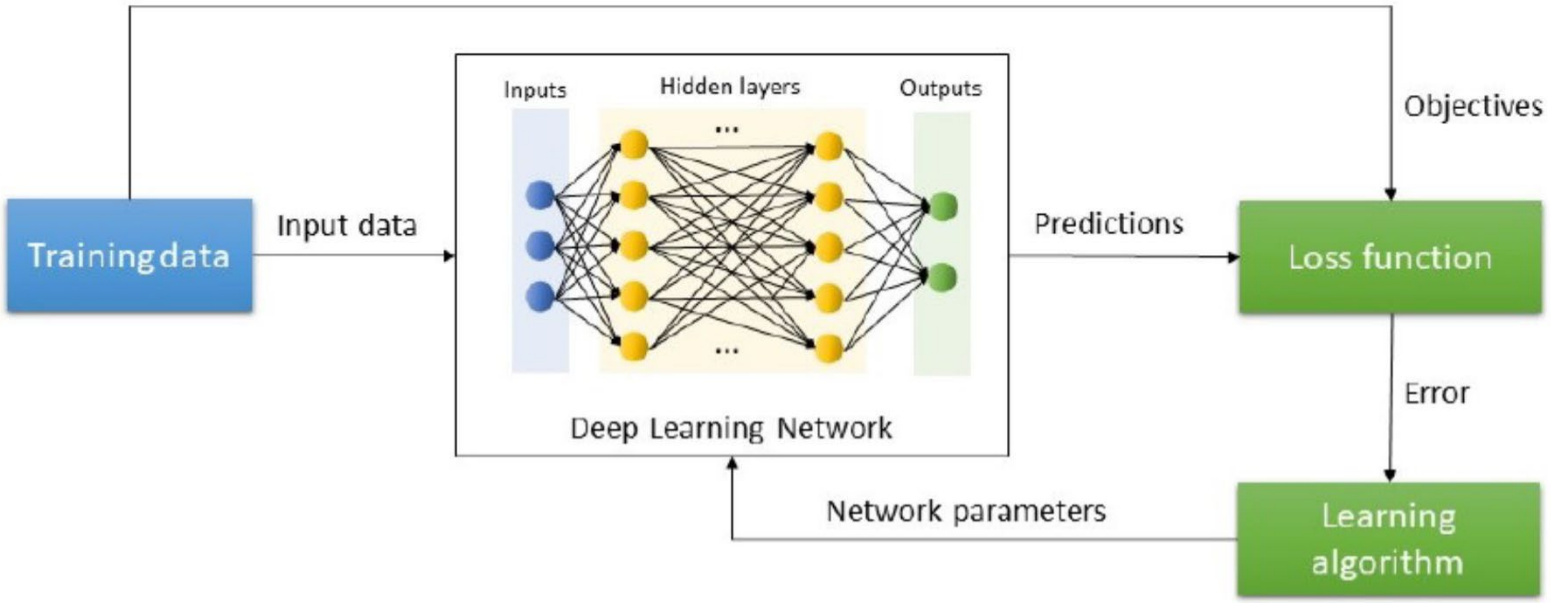
$$\mathscr{L}_{CCE} = -\frac{1}{3}\left(\log(0.7) + \log(0.6) + \log(0.3)\right)$$

3. Compute log terms:

$$\log(0.7) \approx -0.357, \quad \log(0.6) \approx -0.511, \quad \log(0.3) \approx -1.204$$

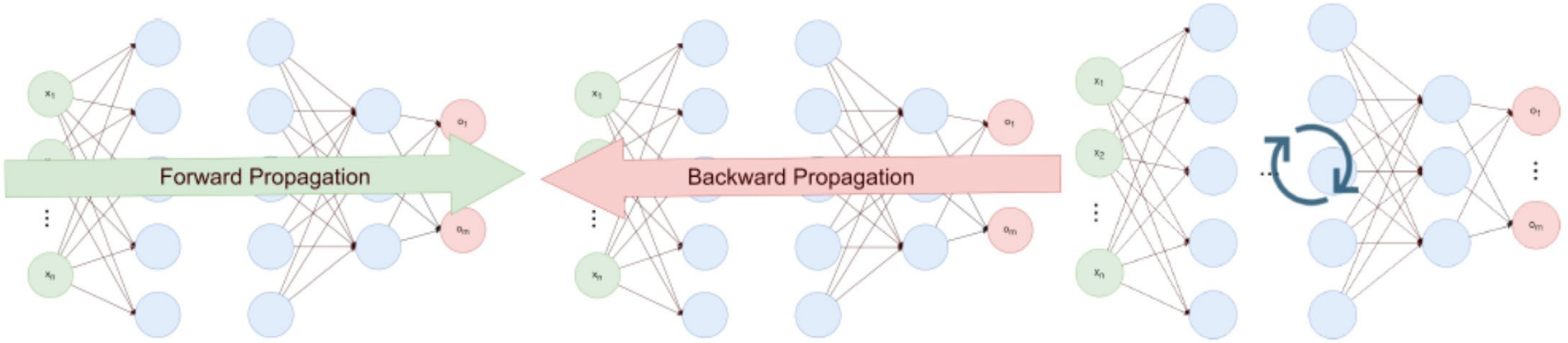
$$\mathscr{L}_{CCE} = \frac{1}{3} \times 2.072 \approx 0.691$$

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.

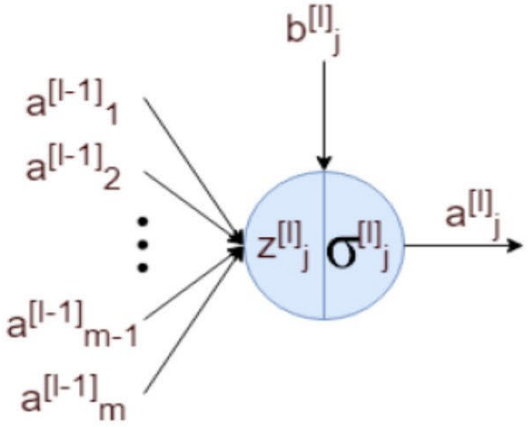


Figure: $a_j^{[l]} = \sigma_j^l(\sum_{i=1}^{m} W_{ij}^{[l]} a_i^{[l-1]} + b_j^{[l]})$

So we can calculate these outputs layer by layer.

Forward propagation

After forward pass we will know:

- Loss value
- Network output
- Middle values

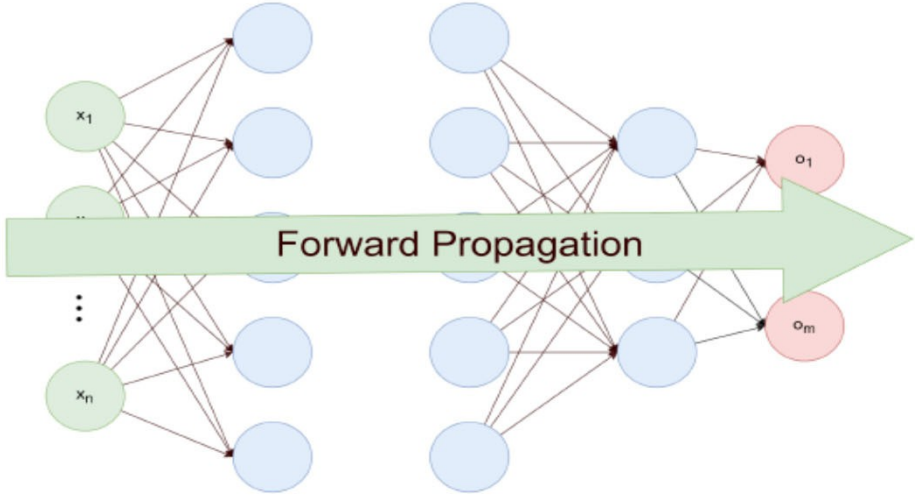


Figure: Forward pass

Backward Propagation

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

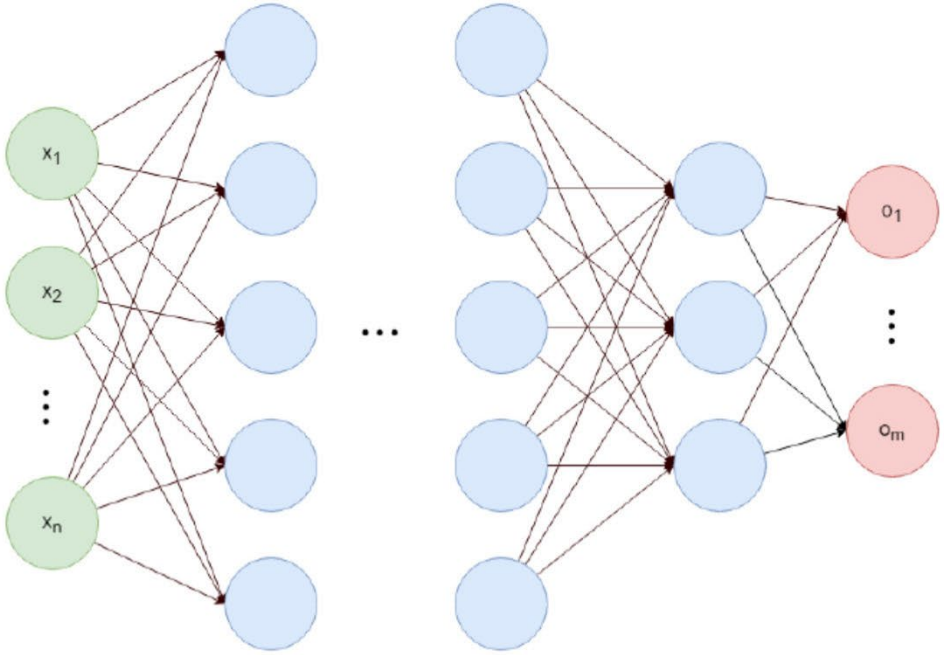
First idea:

- 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

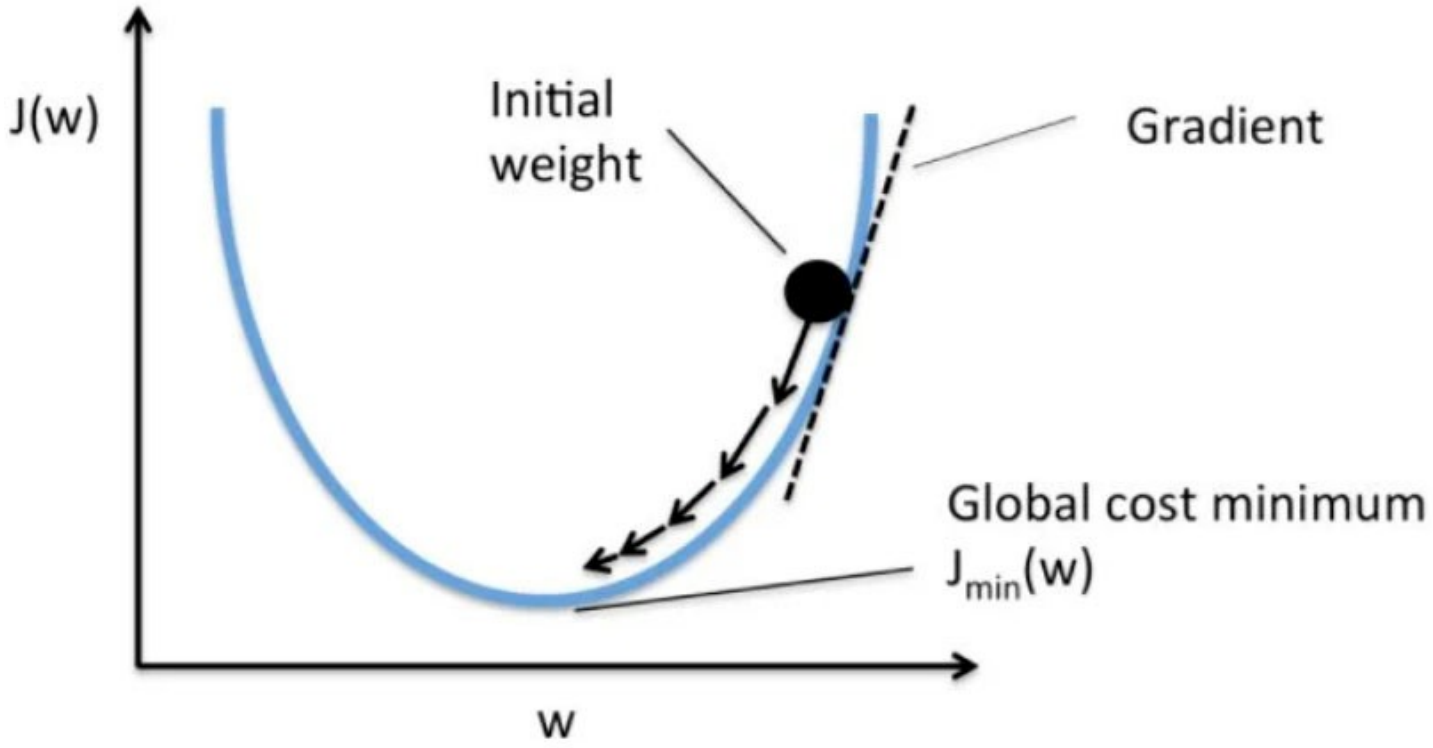
- Cons:
  - Need to rewrite calculation for different architectures

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:

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

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

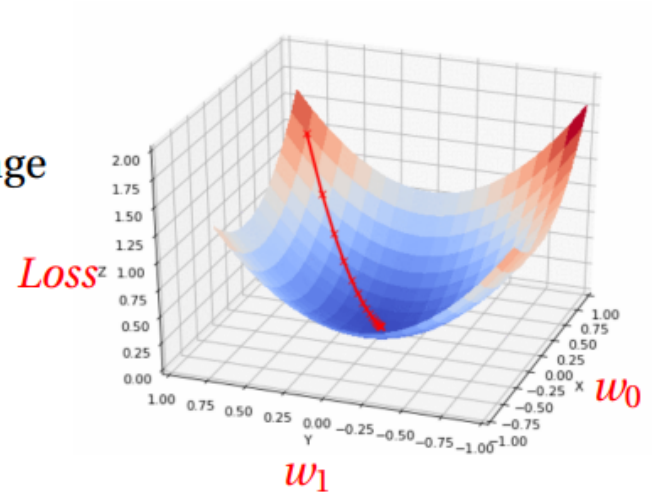
Gradient Descent: Formula and Process

Weight Update Formula:

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

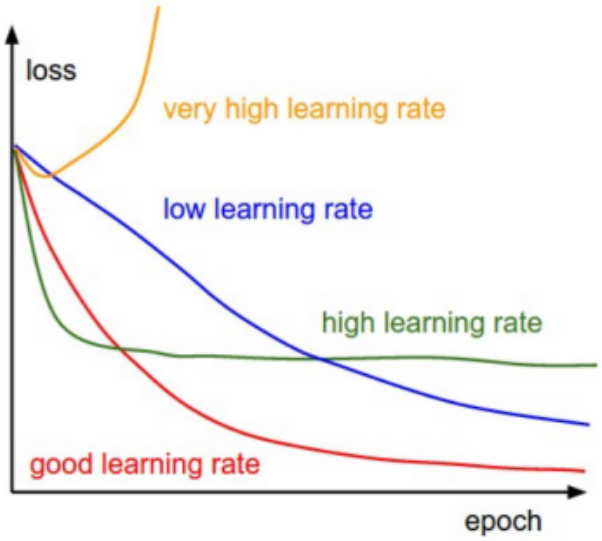
Steps in Gradient Descent:

- 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

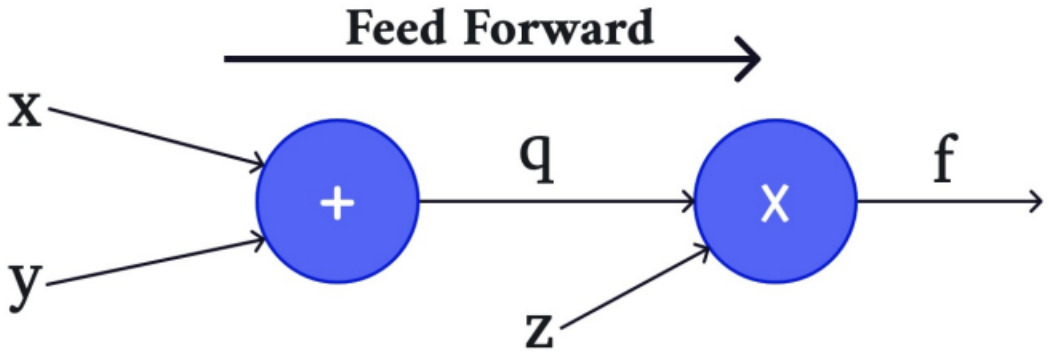
- Look for a **smooth, gradual** decrease in loss over time.
- Very low learning rate -> slow convergence
- Very high learning rate -> erratic fluctuations
- Image adapted from Towards Data Science: Understanding Learning Rates and How It Improves Performance in Deep Learning



Computing derivation (simple example)

Function:

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



Forward path

Function:

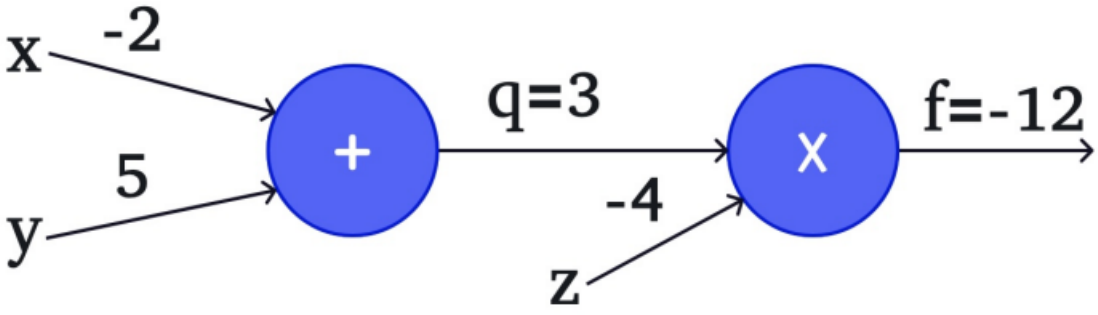
$$f(x, y, z) = (x + y)z$$

Example:

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

Steps:

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



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.

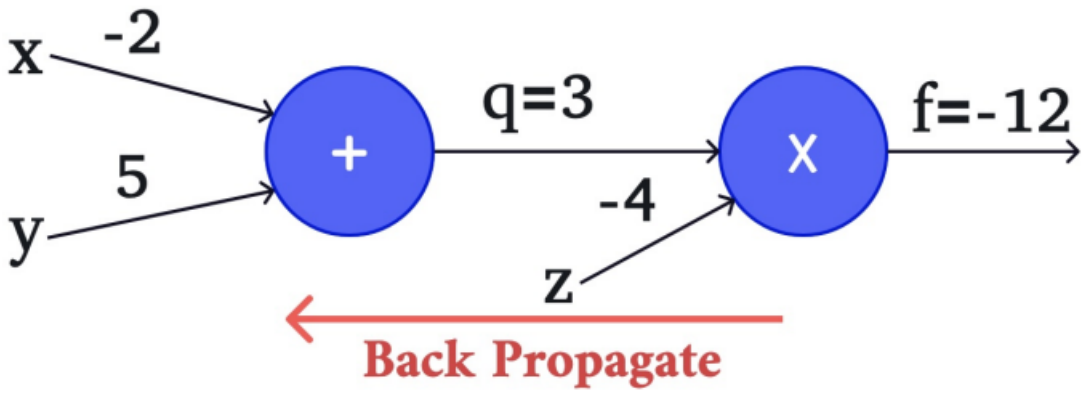
Back propagation (simple example)

Function:

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

Example:

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



Back propagation (simple example)

Function:

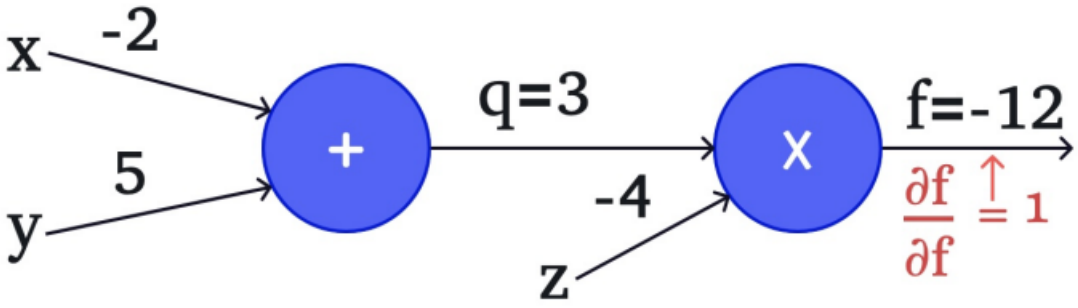
$$f(x, y, z) = (x + y)z$$

Example:

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

Step 1:

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



Back propagation (simple example)

Function:

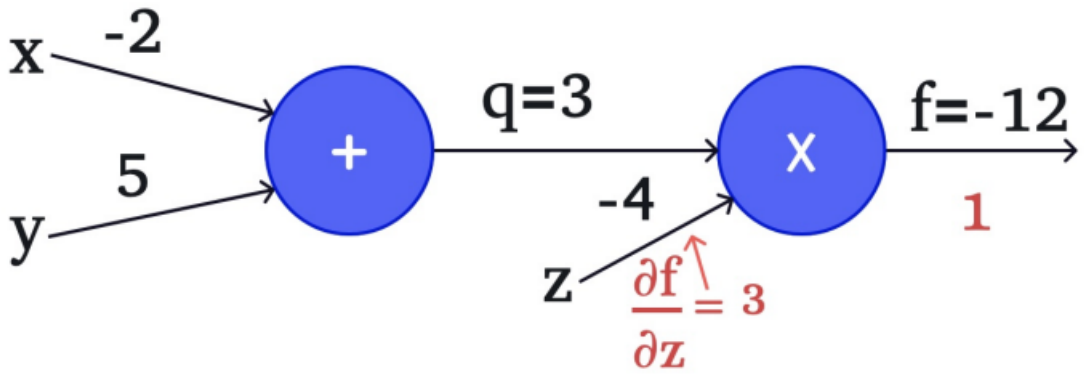
$$f(x, y, z) = (x + y)z$$

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:

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

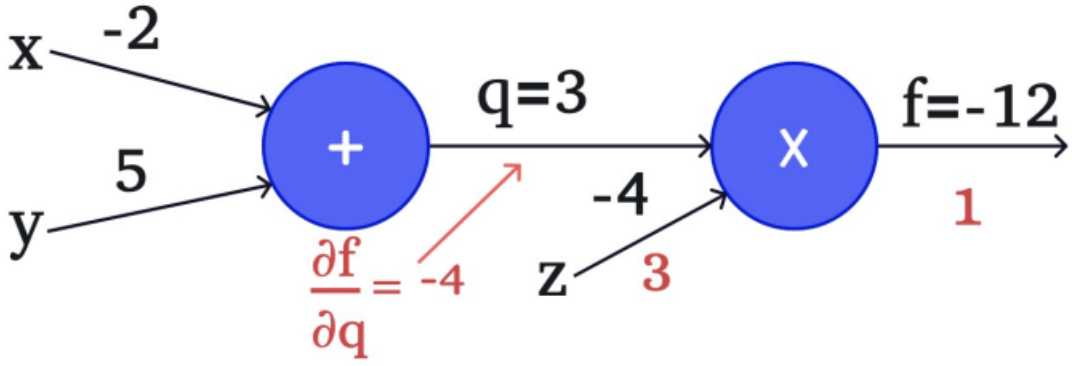
Example:

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

Step 2:

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

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



Back propagation (simple example)

Function:

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

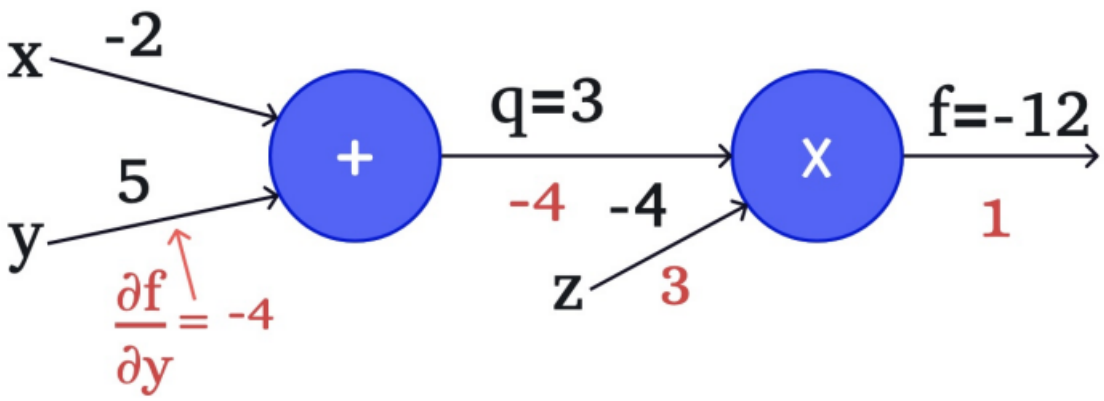
Example:

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

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:

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

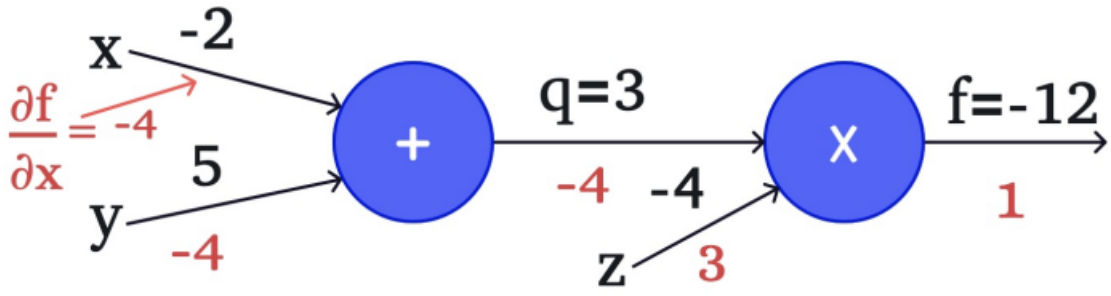
Example:

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

Step 3:

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

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



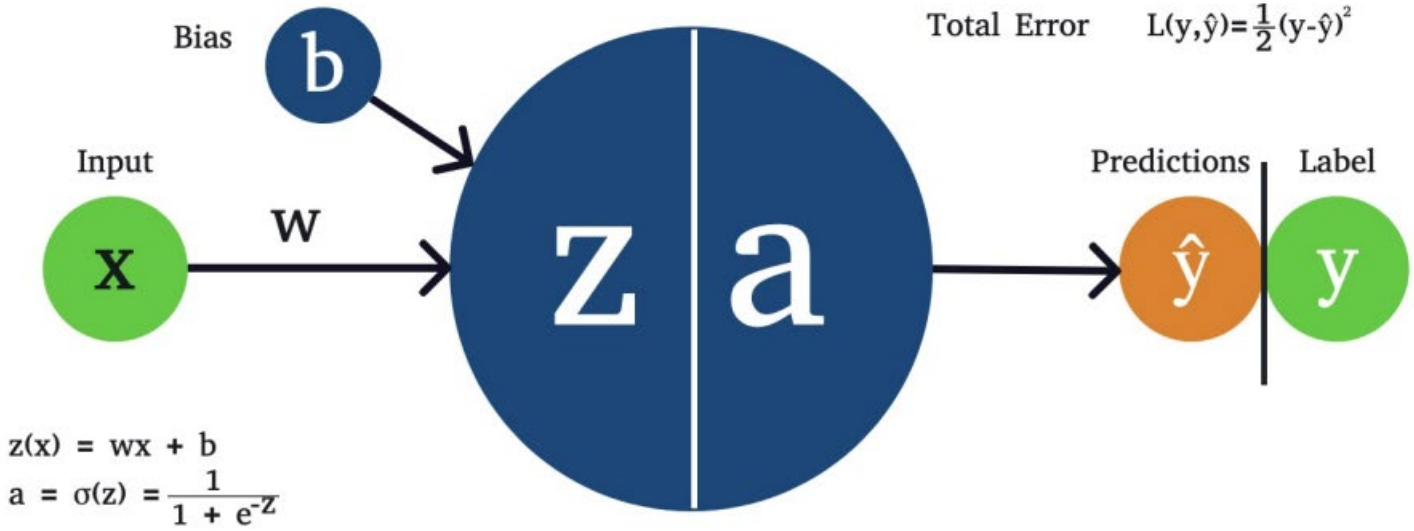
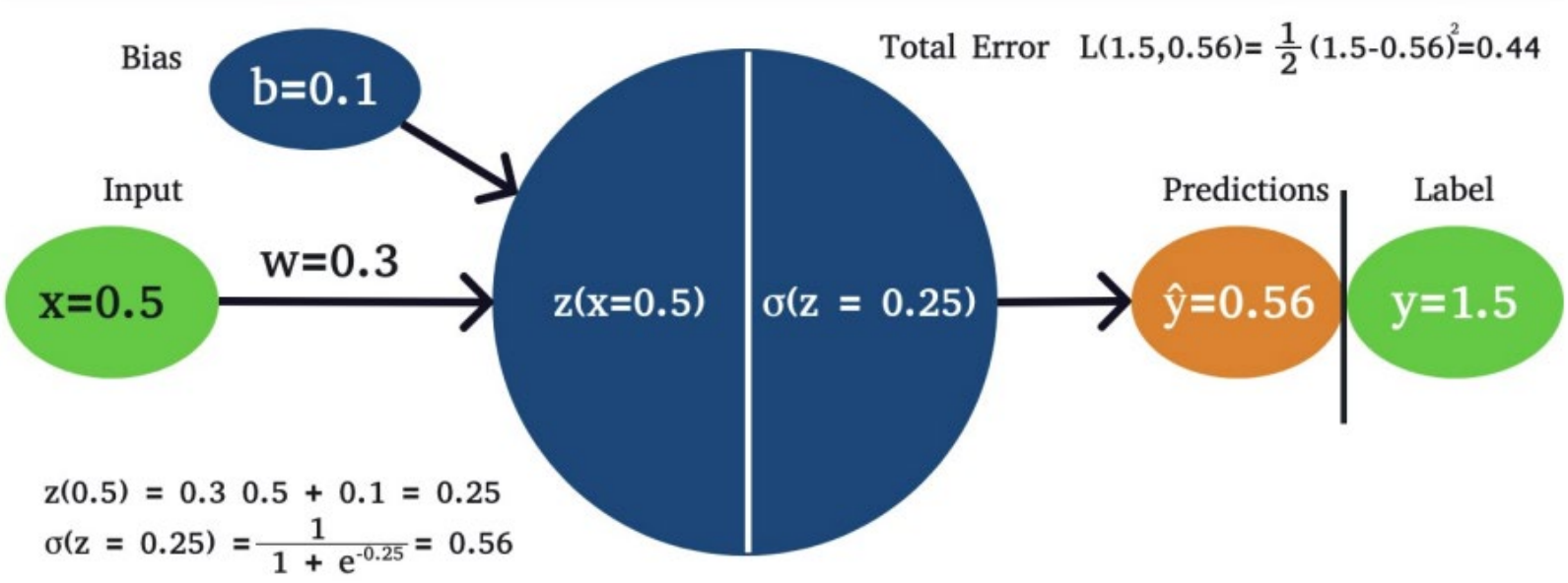
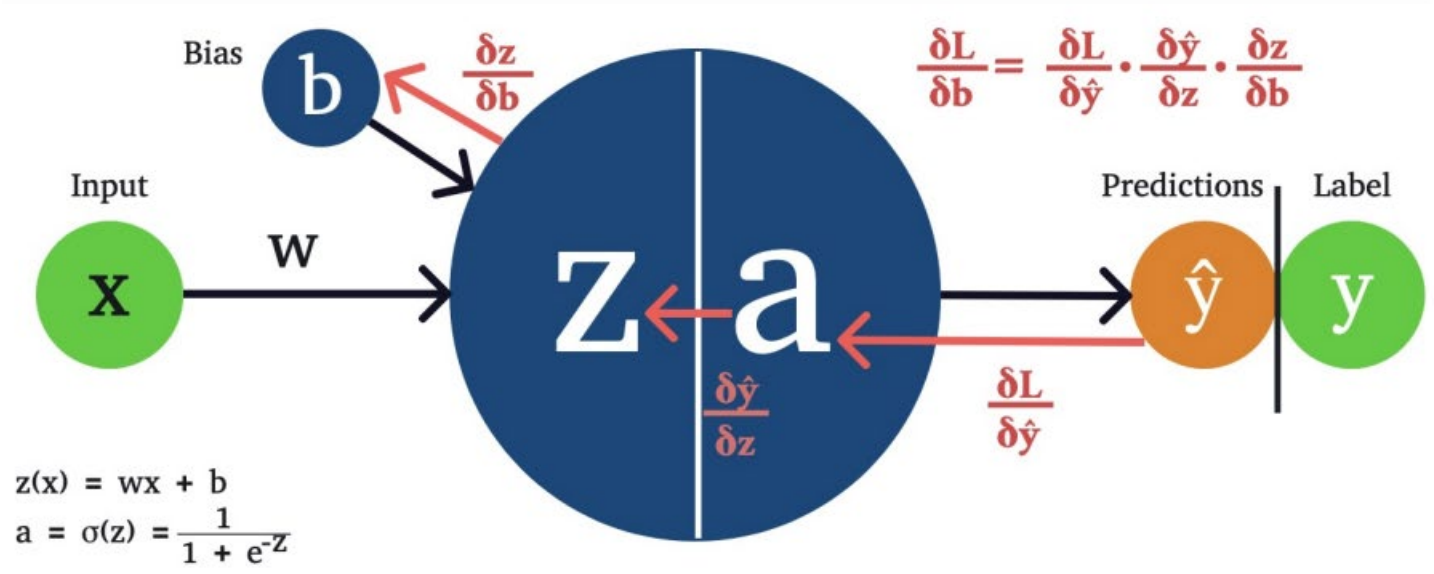


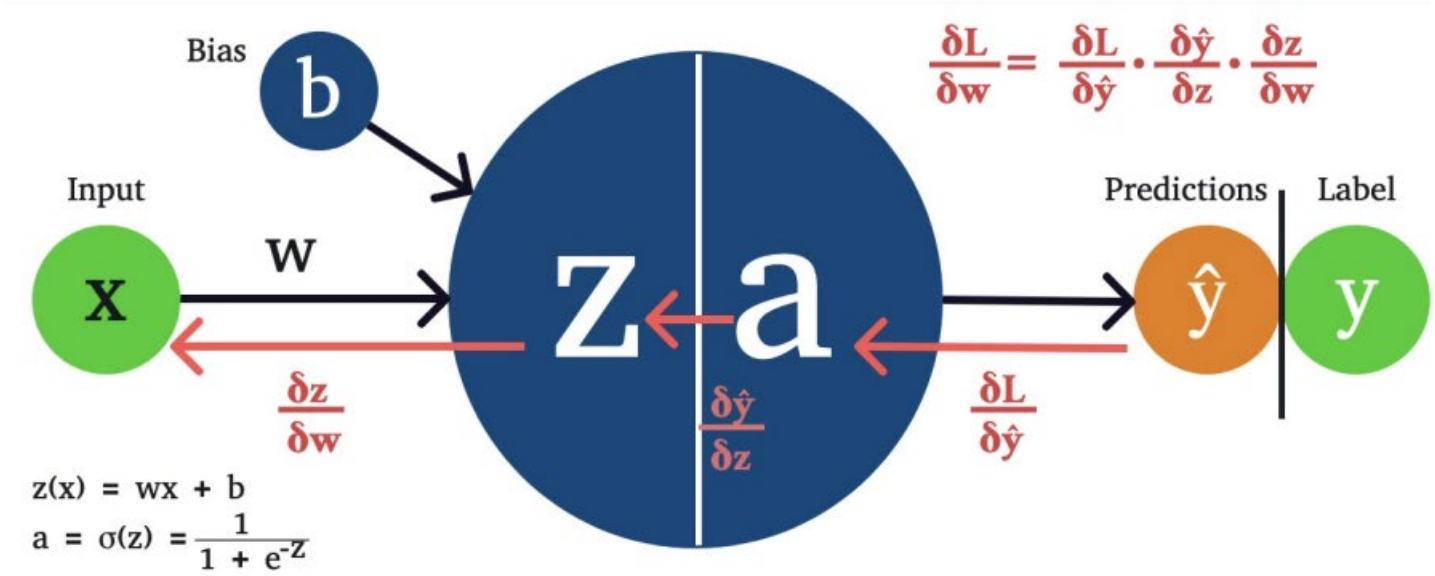
Image adapted from "Deep learning backpropagation" Web article

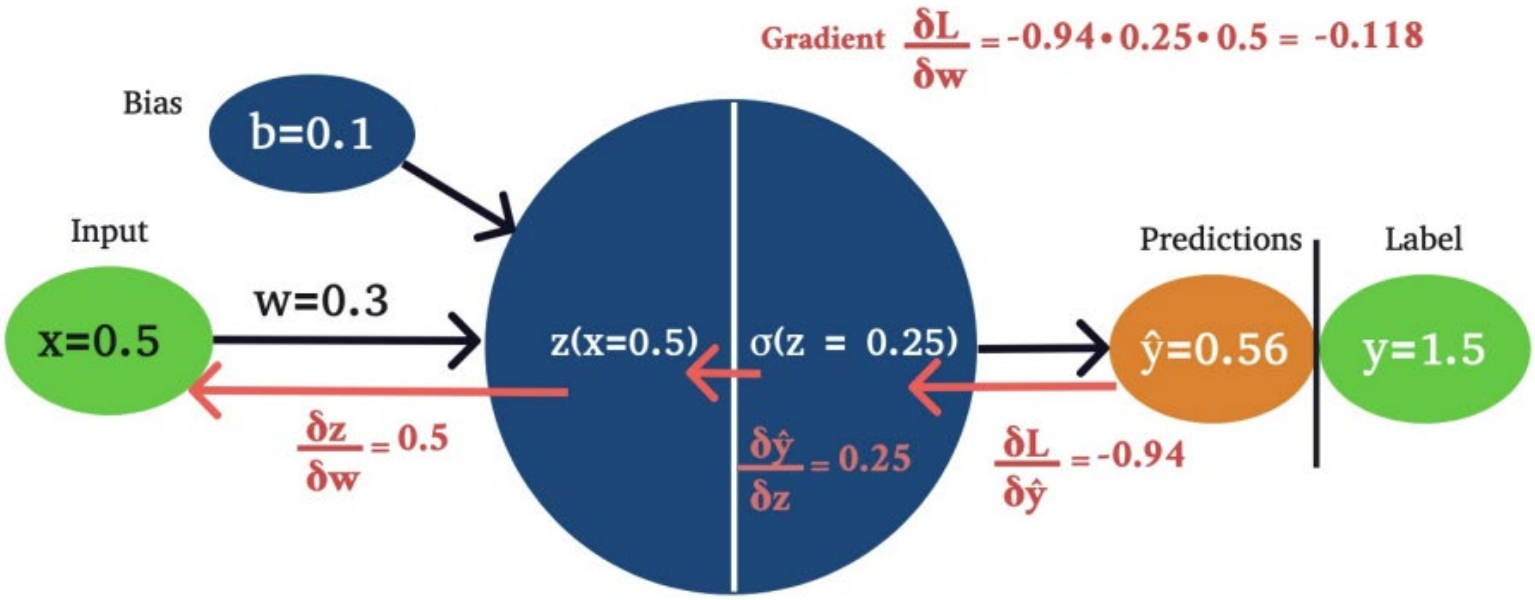
Forward pass



Backward pass



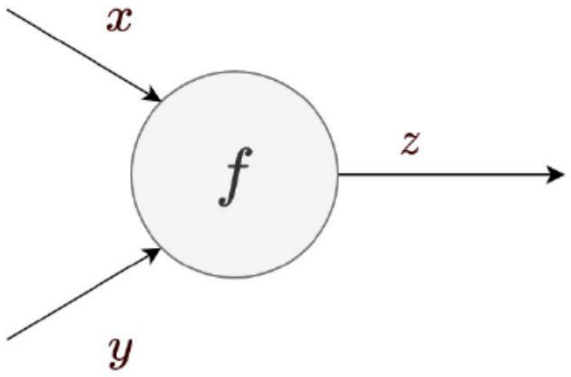

Bias
b
Input
x
w
z
a
Predictions
Label
ŷ
y
$\frac{\delta L}{\delta w} = \frac{\delta L}{\delta \hat{y}} \cdot \frac{\delta \hat{y}}{\delta z} \cdot \frac{\delta z}{\delta w}$
$\frac{\delta z}{\delta w}$
$\frac{\delta \hat{y}}{\delta z}$
$\frac{\delta L}{\delta \hat{y}}$
$z(x) = wx + b$
$a = \sigma(z)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


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
$\frac{\delta z}{\delta w} = 0.5$
$\frac{\delta \hat{y}}{\delta z} = 0.25$
$\frac{\delta L}{\delta \hat{y}} = -0.94$

Backward Propagation

In this approach if we know how to calculate gradient for single node or module, then we can find gradient with respect to each variables.

Let's say we have a module as follow:



It gets x and y as its input and returns $z = f(x, y)$ as its output.

How to calculate derivative of loss with respect to module inputs?

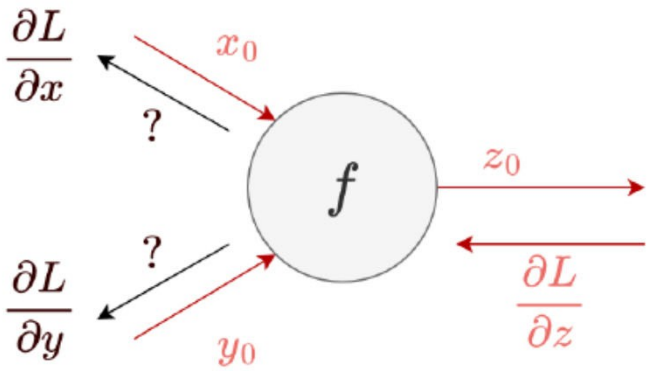
Backward propagation

We know:

- Module output for x_0 and y_0, let's call it z_0.
- Gradient of loss with respect to module output at z_0, $\left(\frac{\partial L}{\partial z}\right)$.

We want:

- Gradient of loss with respect to module inputs at x_0 and y_0, $\left(\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}\right)$.

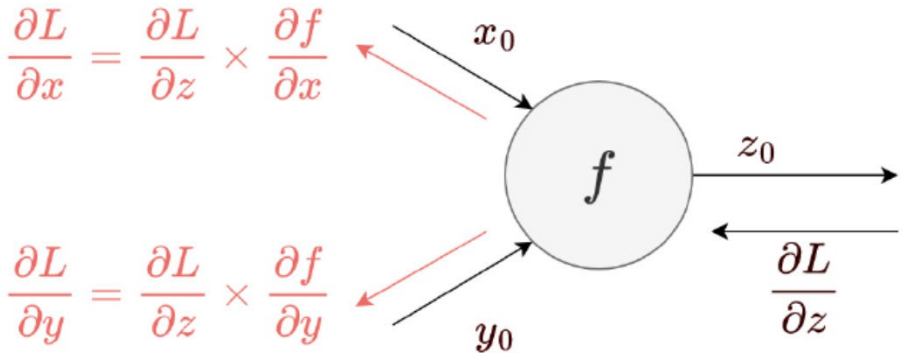


Backward Propagation

- We can use chain rule to do so.

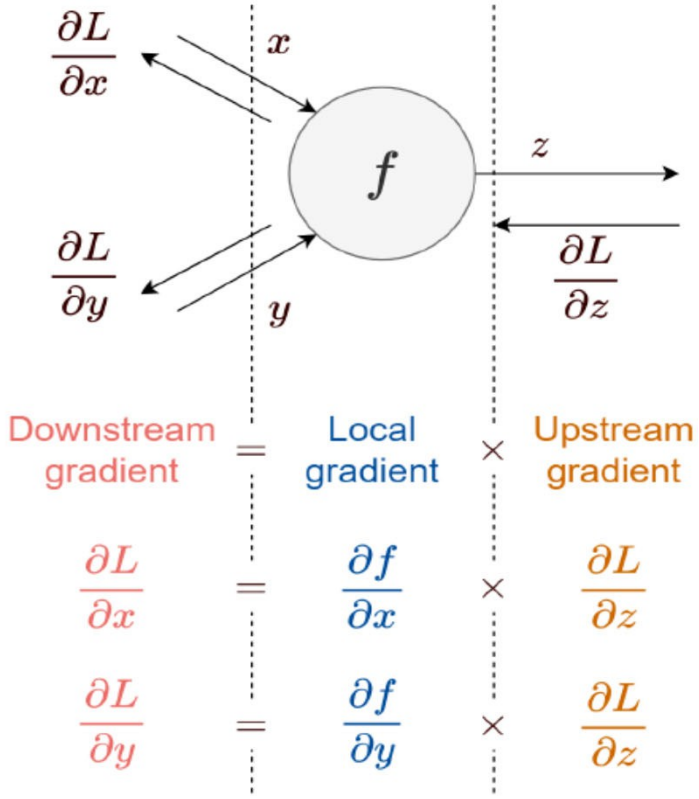
Chain rule:

$$\left.\begin{aligned} z &= f(x,y) \\ L &= L(z) \end{aligned}\right\} \quad \Longrightarrow \quad \frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \times \frac{\partial z}{\partial x}$$



Backward Propagation

Backpropagation for single module:

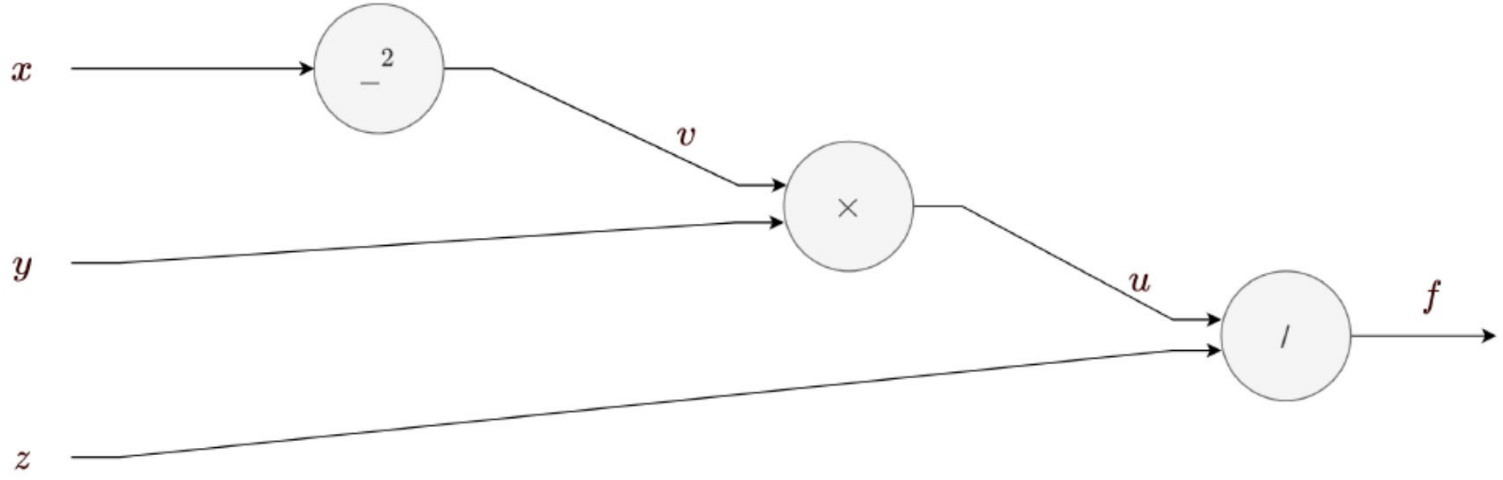


Backward Propagation : Example

Let's solve a simple example using backpropagation.

We have $f(x, y, z) = \frac{x^2 y}{z}$.

Find $\frac{\partial f}{\partial x}$, $\frac{\partial f}{\partial y}$ and $\frac{\partial f}{\partial z}$ at $x = 3$, $y = 4$ and $z = 2$.



Backward Propagation: Example

First let's find gradient analytically.

We have:

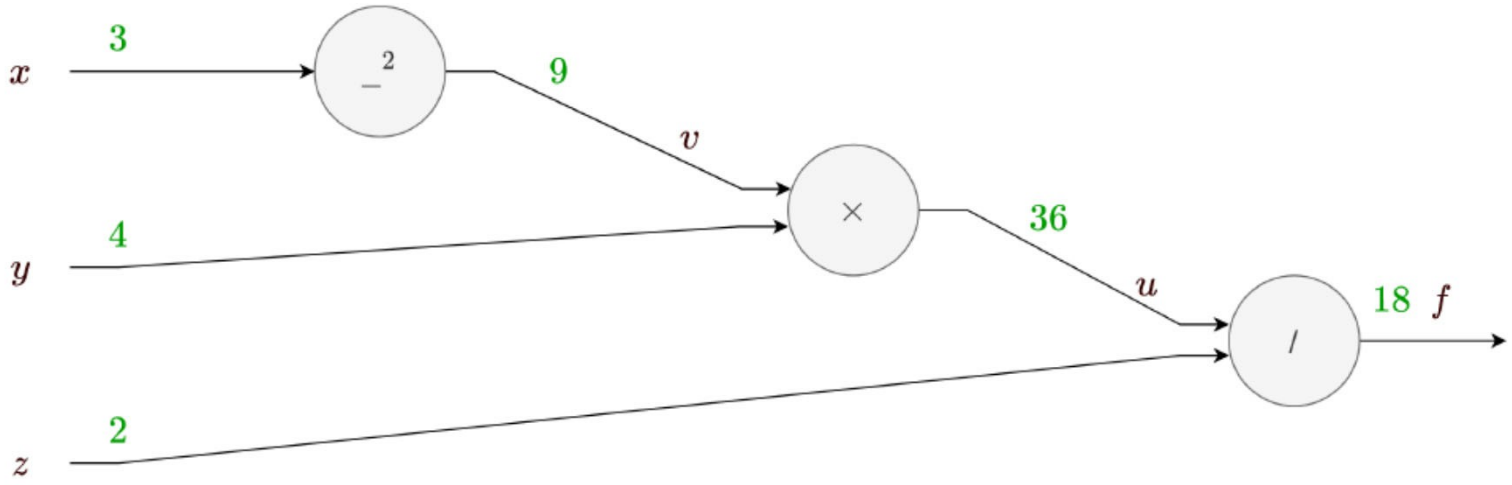
$$\begin{cases} v = x^2 \\ u = vy \\ f = \frac{u}{z} \end{cases} \qquad \begin{cases} \frac{\partial f}{\partial z} = -\frac{u}{z^2} \\ \frac{\partial f}{\partial y} = \frac{\partial u}{\partial y} \times \frac{\partial f}{\partial u} = v \times \frac{1}{z} = \frac{v}{z} \\ \frac{\partial f}{\partial x} = \frac{\partial v}{\partial x} \times \frac{\partial u}{\partial v} \times \frac{\partial f}{\partial u} = 2x \times y \times \frac{1}{z} = \frac{2xy}{z} \end{cases}$$

$$(x = 3, y = 4, z = 2) \implies \begin{cases} v = 9 \\ u = 36 \\ f = 18 \end{cases} \implies \begin{cases} \frac{\partial f}{\partial z} = -9 \\ \frac{\partial f}{\partial y} = 4.5 \\ \frac{\partial f}{\partial x} = 12 \end{cases}$$

Backward Propagation : Example

Now let's use backpropagation.

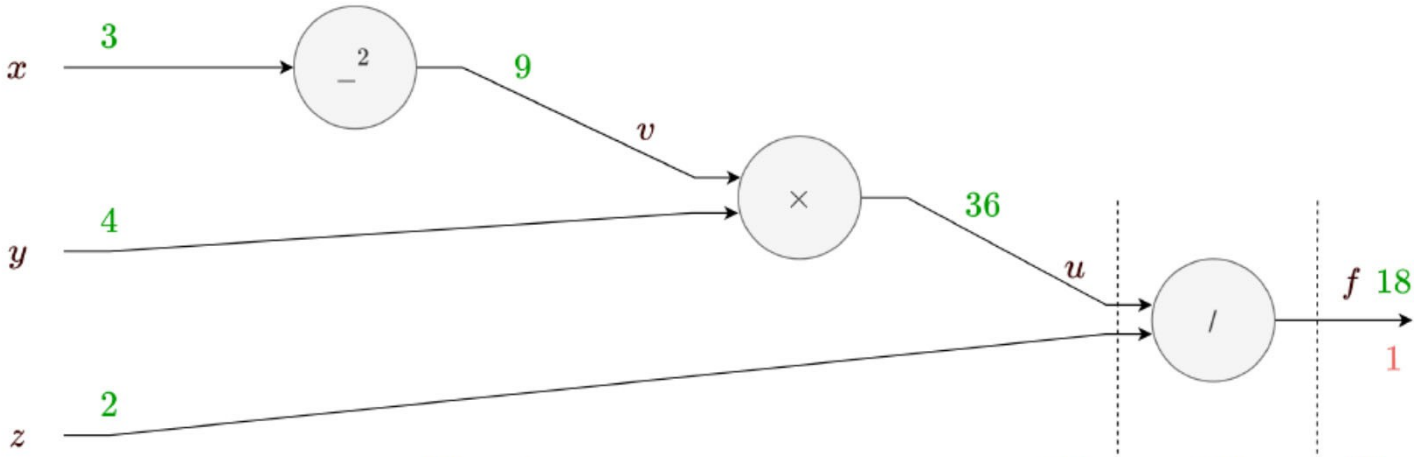
First we do forward propagation.



Second we will do backpropagation for each module.

Backward Propagation

Backpropagation for / module:

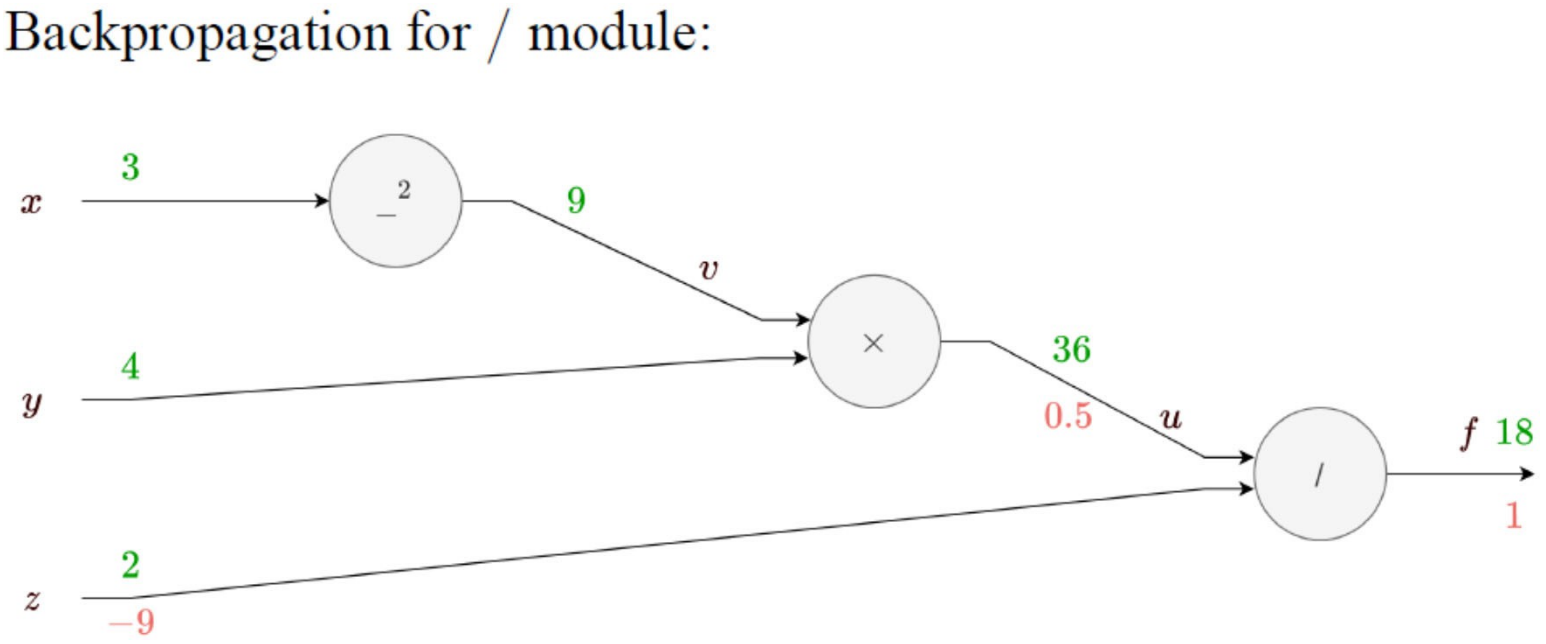


$$\begin{cases} \frac{\partial f}{\partial u} = \frac{1}{z} = 0.5 \\ \frac{\partial f}{\partial z} = -\frac{u}{z^2} = -9 \end{cases} \qquad \begin{cases} \frac{\partial f}{\partial f} = 1 \end{cases}$$

$$\frac{\partial f}{\partial u} = \frac{\partial f}{\partial u} \times \frac{\partial f}{\partial f}$$

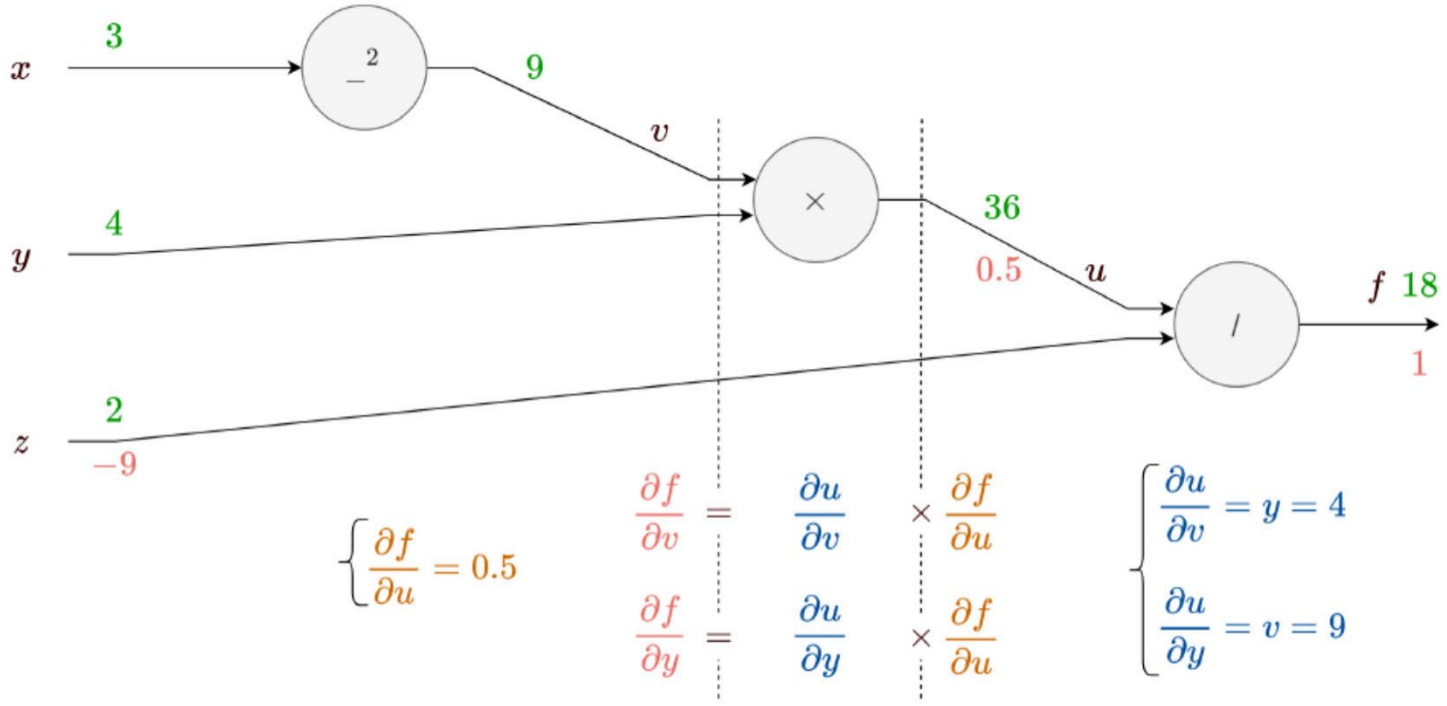
$$\frac{\partial f}{\partial z} = \frac{\partial f}{\partial z} \times \frac{\partial f}{\partial f}$$

Backward Propagation : example



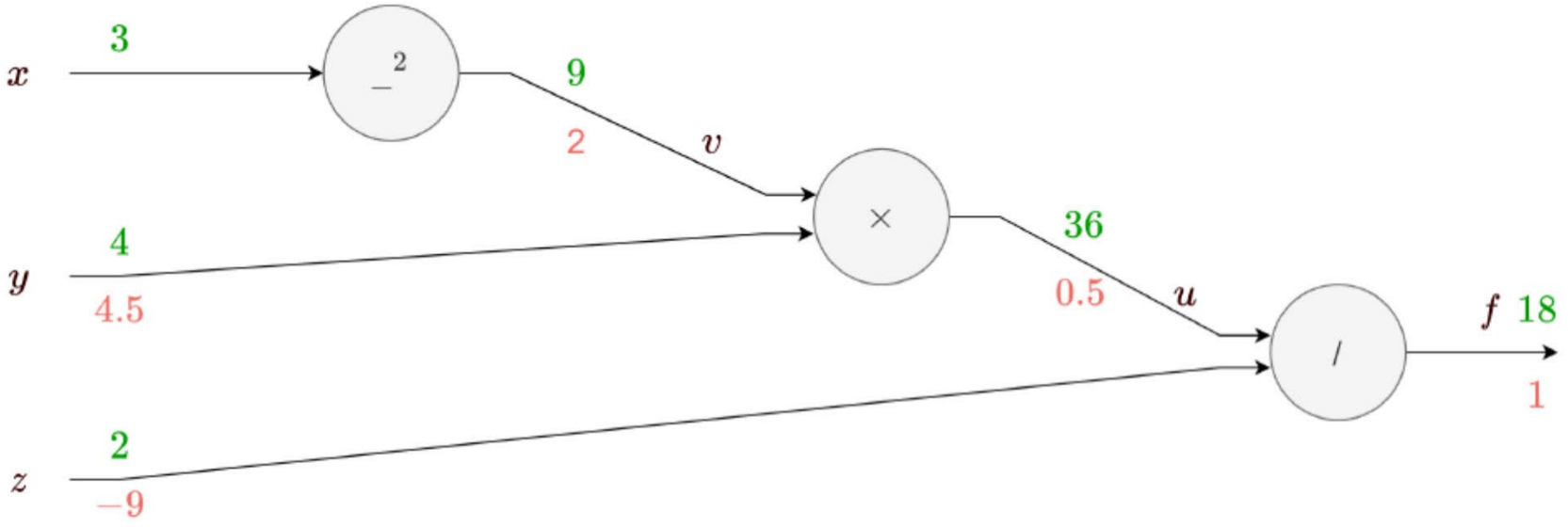
Backward propagation : example

Backpropagation for $\times$ module:



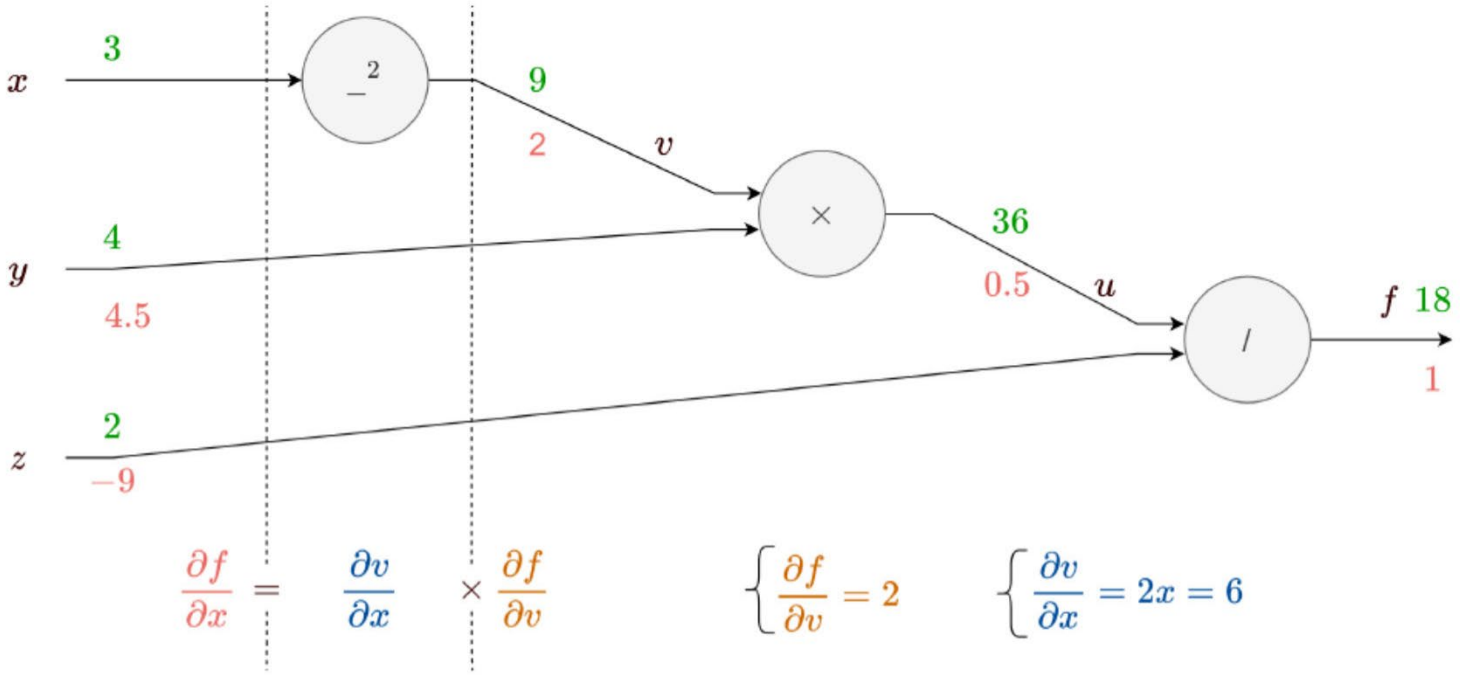
Backward propagation : example

Backpropagation for $\times$ module:



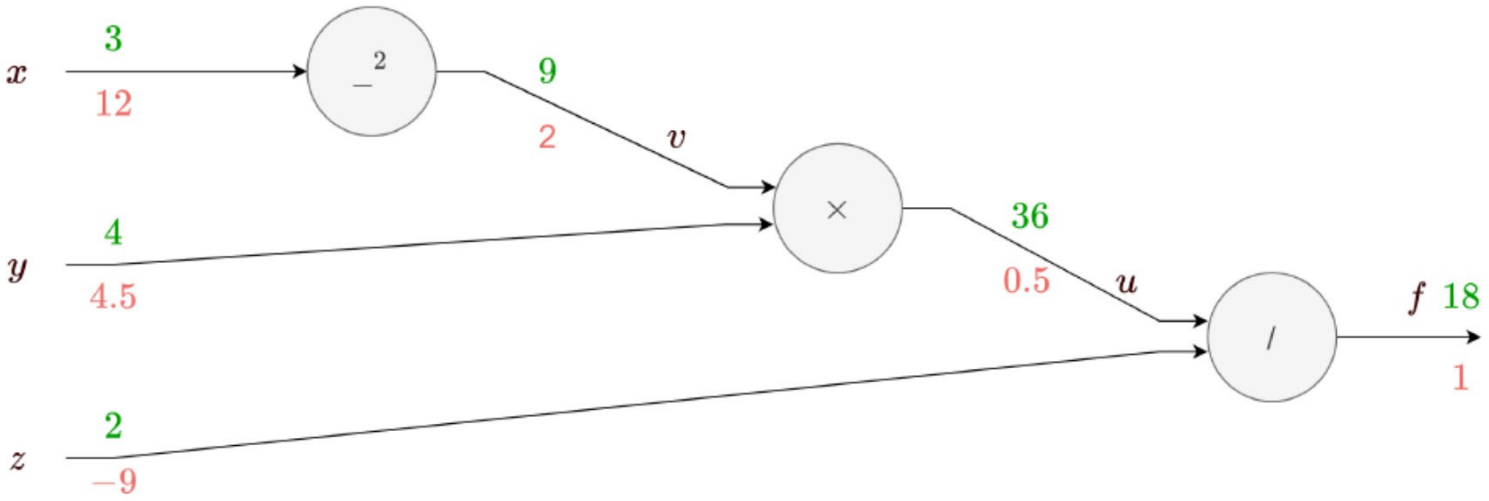
Backward propagation : example

Backpropagation for $\_^2$ module:



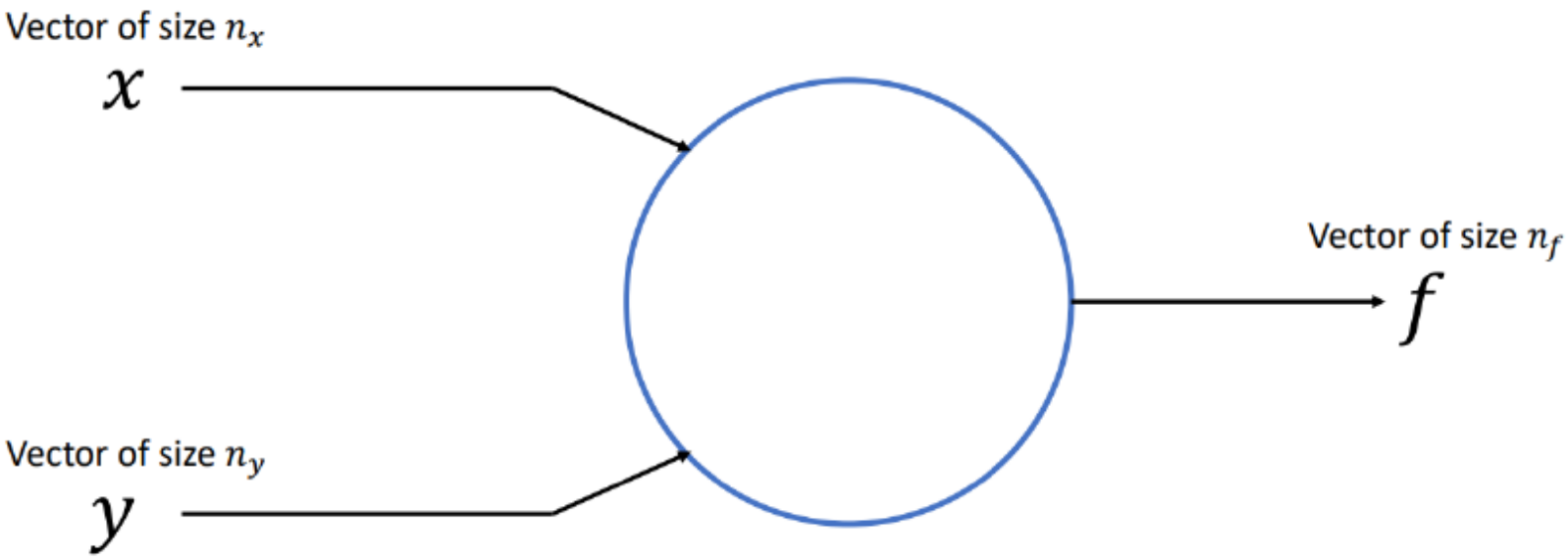
Backward propagation : example

Backpropagation for $\_^2$ module:



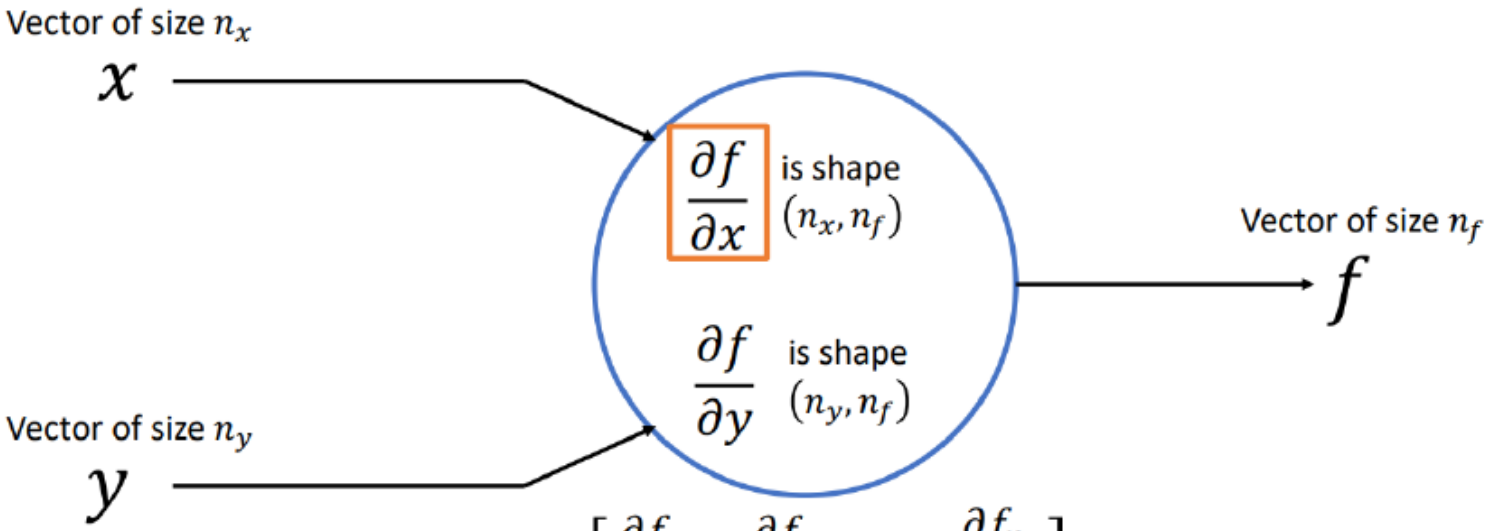
Results are the same as analytical results.

Vectorized Backpropagation



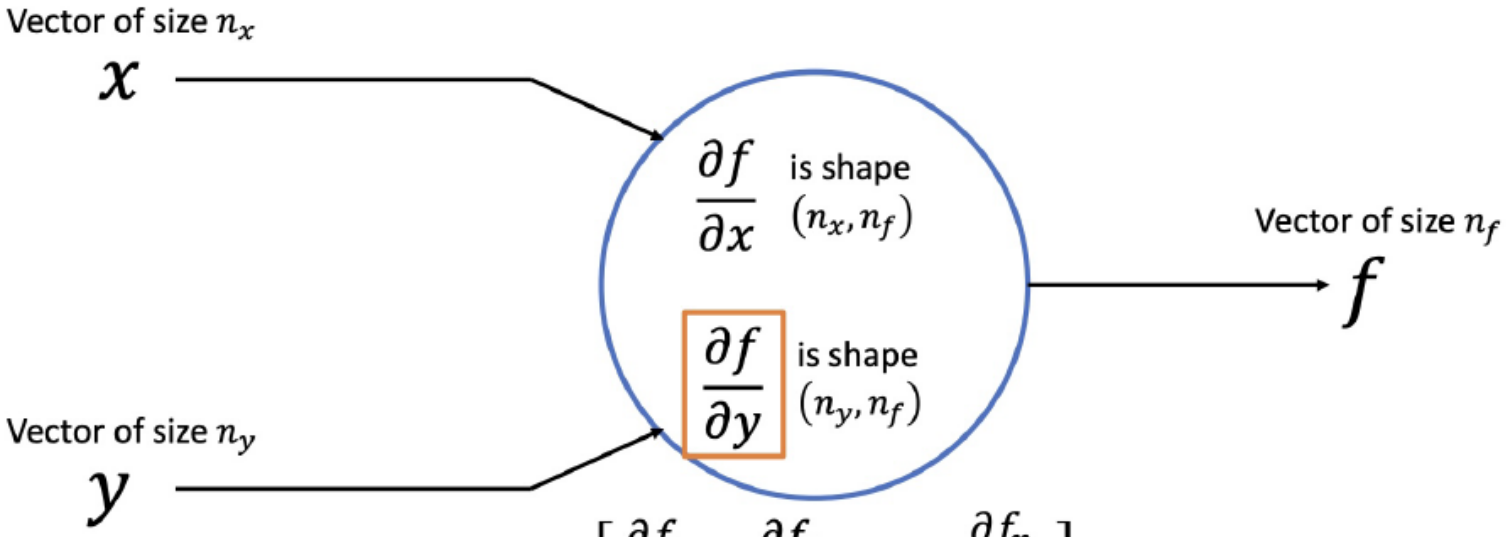
Vectorized Backpropagation

Local Derivatives are Jacobian Matrices



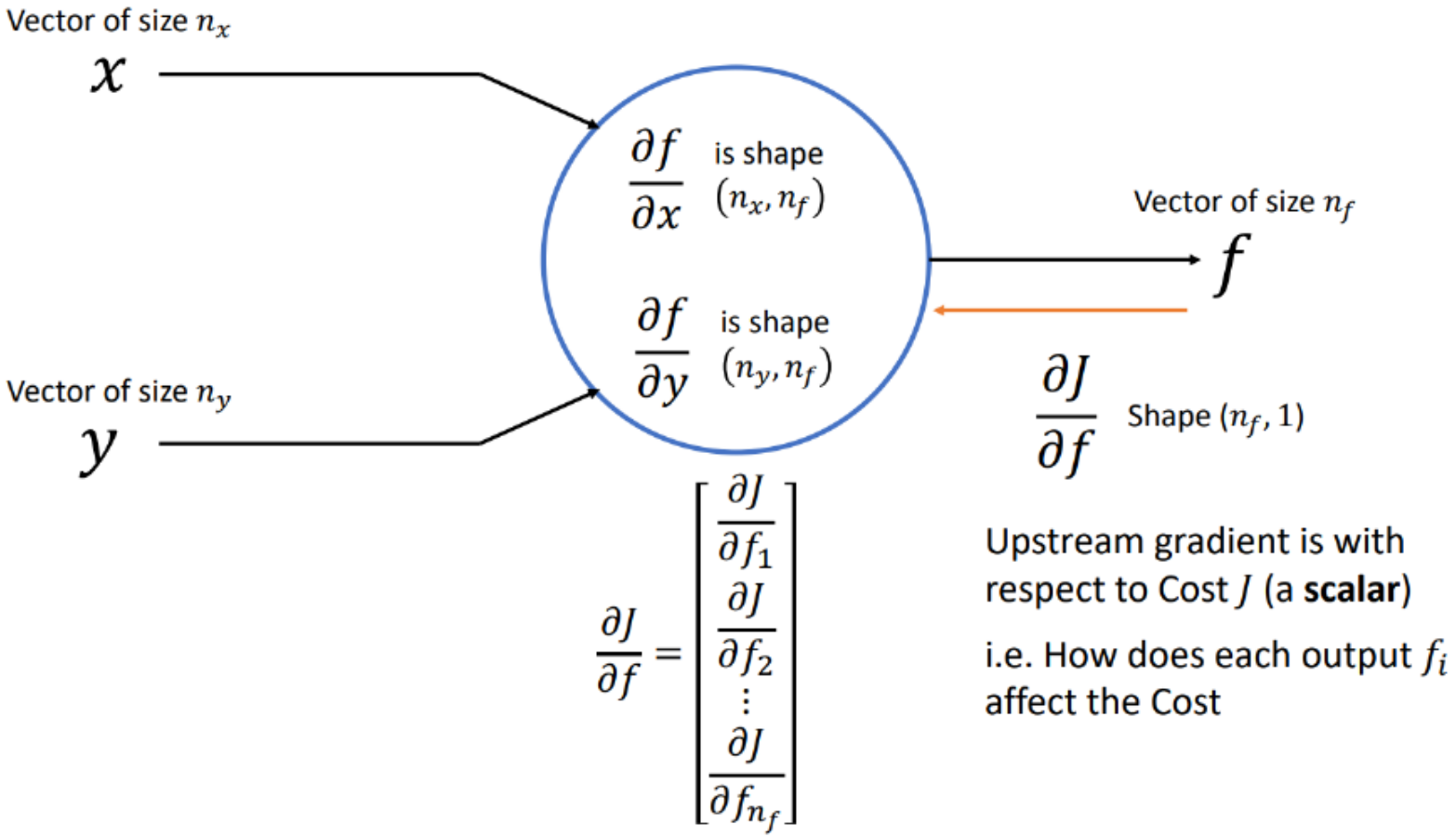
$$\frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_2}{\partial x_1} & \cdots & \frac{\partial f_{n_f}}{\partial x_1} \\ \frac{\partial f_1}{\partial x_2} & \frac{\partial f_2}{\partial x_2} & & \frac{\partial f_{n_f}}{\partial x_2} \\ \vdots & & \ddots & \vdots \\ \frac{\partial f_1}{\partial x_{n_x}} & \frac{\partial f_2}{\partial x_{n_x}} & \cdots & \frac{\partial f_{n_f}}{\partial x_{n_x}} \end{bmatrix}$$

Vectorized backpropagation

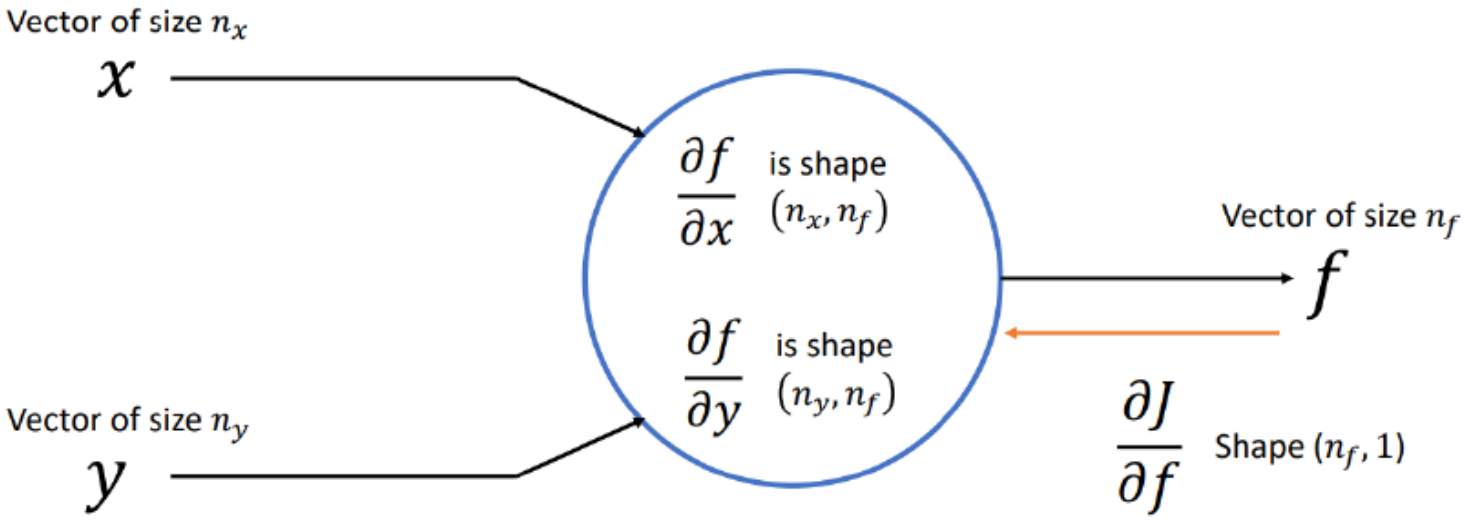


$$\frac{\partial f}{\partial y} = \begin{bmatrix} \frac{\partial f_1}{\partial y_1} & \frac{\partial f_2}{\partial y_1} & \cdots & \frac{\partial f_{n_f}}{\partial y_1} \\ \frac{\partial f_1}{\partial y_2} & \frac{\partial f_2}{\partial y_2} & & \frac{\partial f_{n_f}}{\partial y_2} \\ \vdots & & \ddots & \vdots \\ \frac{\partial f_1}{\partial y_{n_y}} & \frac{\partial f_2}{\partial y_{n_y}} & \cdots & \frac{\partial f_{n_f}}{\partial y_{n_y}} \end{bmatrix}$$

Vectorized Backpropagation

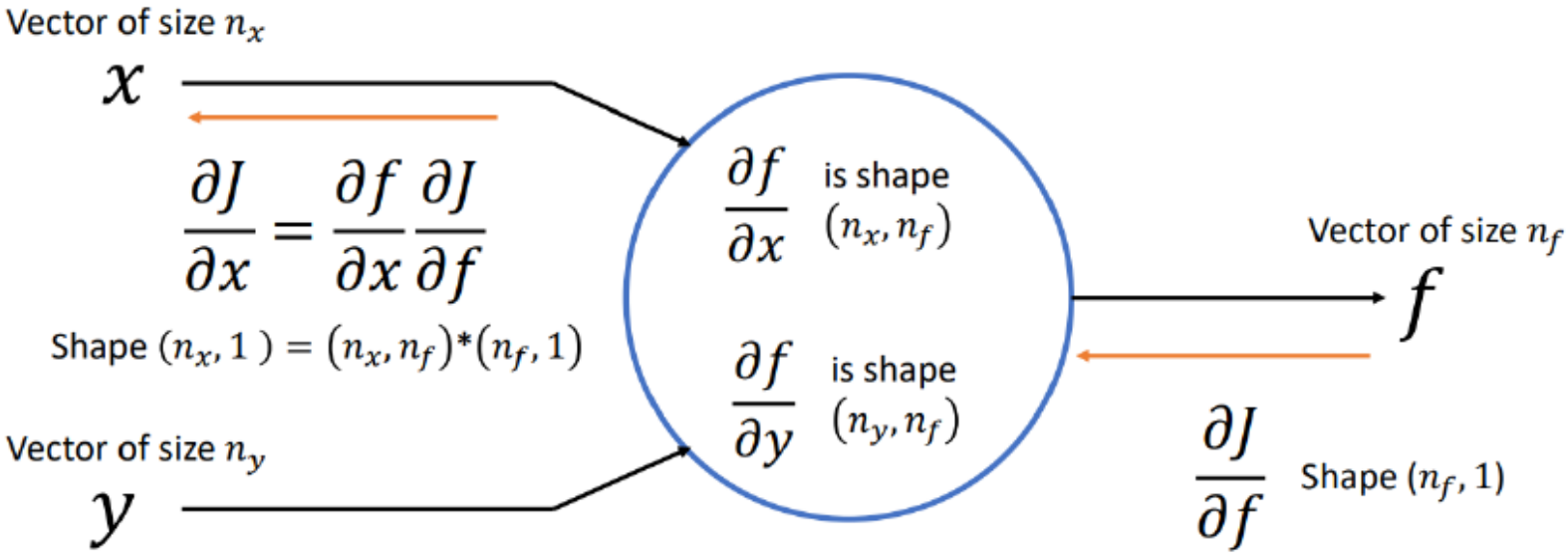


Vectorized Backpropagation



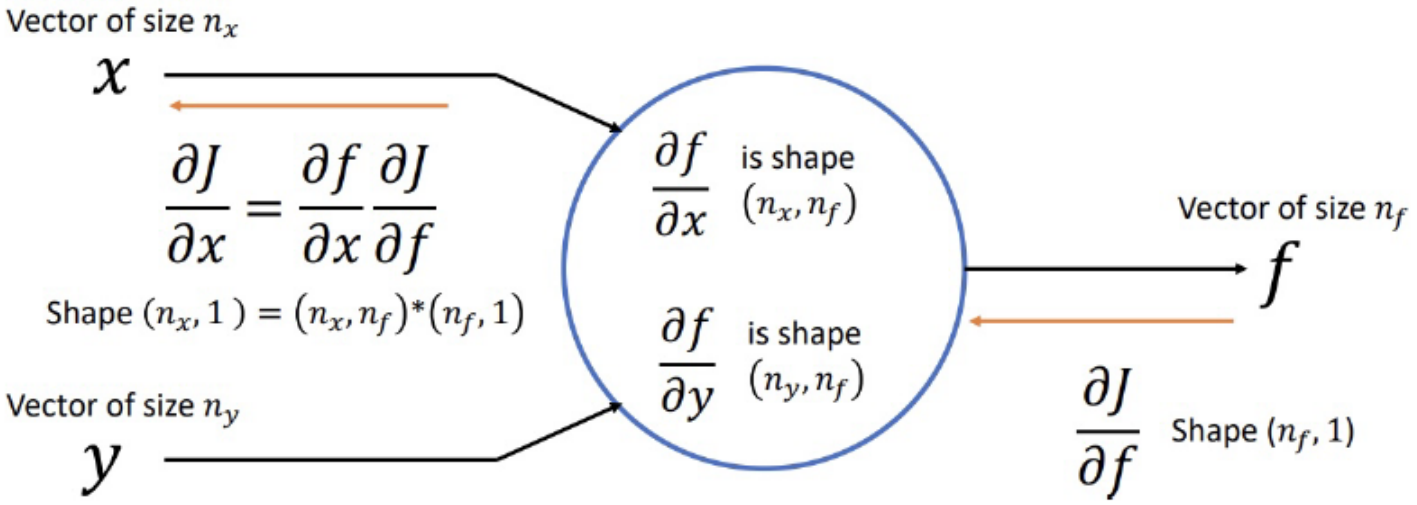
Apply chain rule like before!

Vectorized Backpropagation



Applying the chain rule involves matrix-vector multiplication

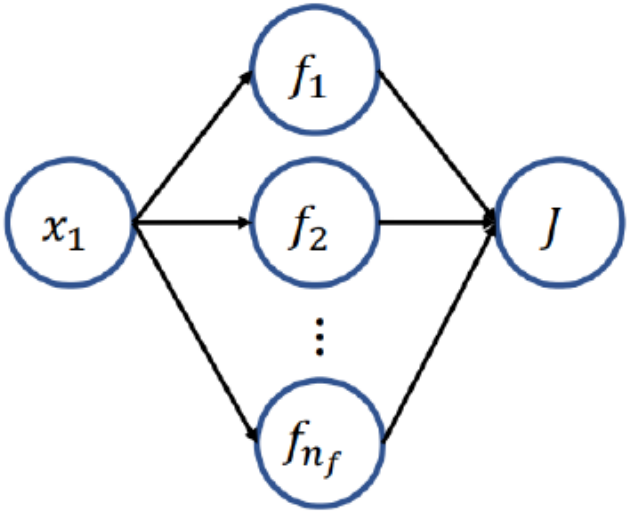
Vectorized Backpropagation



$$\frac{\partial J}{\partial x} = \frac{\partial f}{\partial x}\frac{\partial J}{\partial f} \rightarrow \begin{bmatrix} \frac{\partial J}{\partial x_1} \\ \frac{\partial J}{\partial x_2} \\ \vdots \\ \frac{\partial J}{\partial x_{n_x}} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1}, \frac{\partial f_2}{\partial x_1}, & \cdots & \frac{\partial f_{n_f}}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_1}{\partial x_{n_x}}, \frac{\partial f_2}{\partial x_{n_x}} & \cdots & \frac{\partial f_{n_f}}{\partial x_{n_x}} \end{bmatrix} \begin{bmatrix} \frac{\partial J}{\partial f_1} \\ \frac{\partial J}{\partial f_2} \\ \vdots \\ \frac{\partial J}{\partial f_{n_f}} \end{bmatrix}$$

Vectorized Backpropagation

$$\frac{\partial J}{\partial x} = \frac{\partial f}{\partial x}\frac{\partial J}{\partial f} \rightarrow \begin{bmatrix} \frac{\partial J}{\partial x_1} \\ \frac{\partial J}{\partial x_2} \\ \vdots \\ \frac{\partial J}{\partial x_{n_x}} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1}, \frac{\partial f_2}{\partial x_1}, & \cdots & \frac{\partial f_{n_f}}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_1}{\partial x_{n_x}}, \frac{\partial f_2}{\partial x_{n_x}} & \cdots & \frac{\partial f_{n_f}}{\partial x_{n_x}} \end{bmatrix} \begin{bmatrix} \frac{\partial J}{\partial f_1} \\ \frac{\partial J}{\partial f_2} \\ \vdots \\ \frac{\partial J}{\partial f_{n_f}} \end{bmatrix}$$



Shape $(n_x, 1) = (n_x, n_f)^*(n_f, 1)$

$$\frac{\partial J}{\partial x_1} = \frac{\partial f_1}{\partial x_1}\frac{\partial J}{\partial f_1} + \frac{\partial f_2}{\partial x_1}\frac{\partial J}{\partial f_2} + \cdots + \frac{\partial f_{n_f}}{\partial x_1}\frac{\partial J}{\partial f_{n_f}}$$

Backward Propagation

So after backward propagation we will have:

- Gradient of loss with respect to each parameter.
- We can apply gradient descent to update parameters.

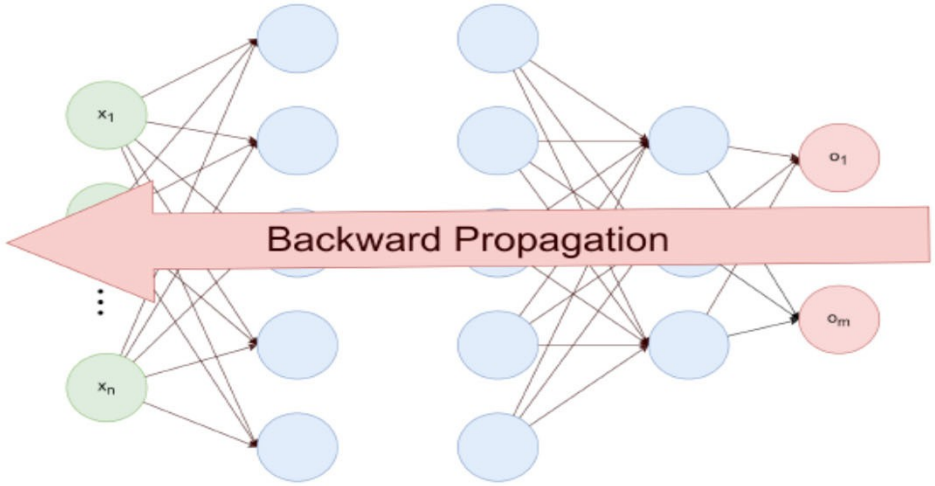


Figure: Backward pass

Various GD type

So far you got familiar with gradient-based optimization.

If $\boldsymbol{g} = \nabla_{\boldsymbol{\theta}} \mathcal{J}$, then we will update parameters with this simple rule:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \boldsymbol{g}$$

But there is one question here, how to compute $\boldsymbol{g}$?

Based on how we calculate $\boldsymbol{g}$ we will have different types of gradient descent:

- Batch Gradient Descent
- Stochastic Gradient Descent
- Mini-Batch Gradient Descent

Various GD types: Batch Gradient Descent

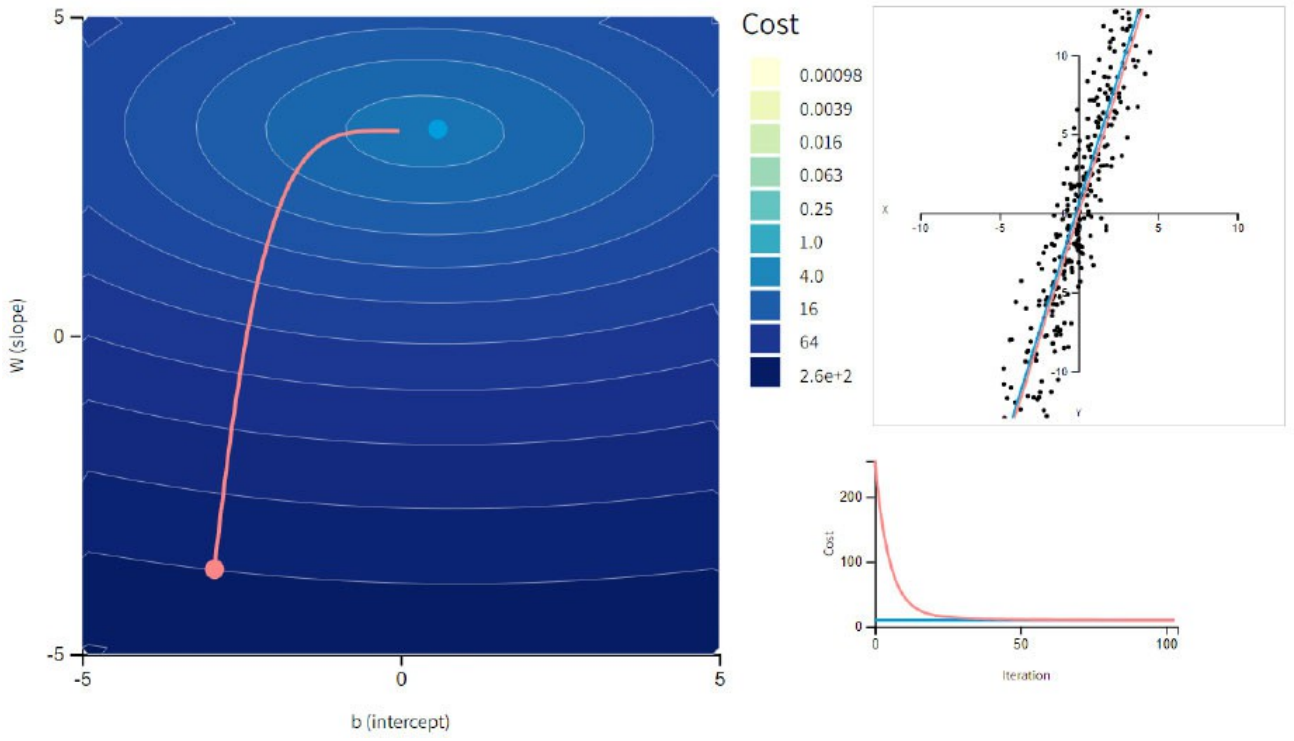
- In this type we use entire training set to calculate gradient.

Batch Gradient:

$$\boldsymbol{g} = \frac{1}{n} \sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} \mathcal{L}(d_i, \boldsymbol{\theta})$$

- Using this method with very large training set:
  - Your data can be too large to process in your memory.
  - It requires a lot of processing to compute gradient for all samples.
- Using exact gradient may lead us to local minima.
- Moving noisy may help us get out of this local minimas.

Various GD types: Batch Gradient Descent



Various GD types: Stochastic Gradient Descent

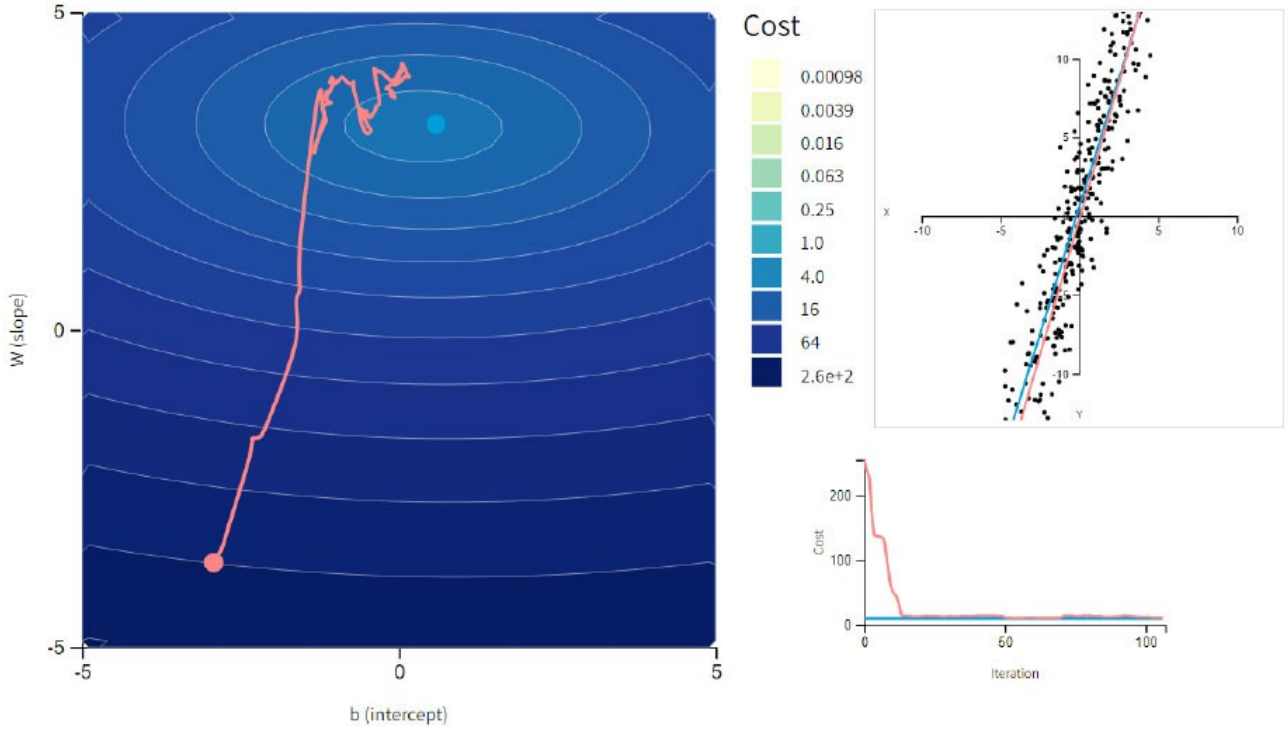
- Instead of calculating exact gradient, we can estimate it using our data.
- This is exactly what SGD does, it estimates gradient using only single data point.

Stochastic Gradient:

$$\hat{\boldsymbol{g}} = \nabla_{\boldsymbol{\theta}} \mathcal{L}(d_i, \boldsymbol{\theta})$$

- As we use an approximation of gradient, instead of gently decreasing, the cost function will bounce up and down and decrease only on average.
- This method is really computationally efficient cause we only need to calculate gradient for one point per iteration.

Various GD types: Stochastic Gradient Descent



Various GD types: MiniBatch Gradient Descent

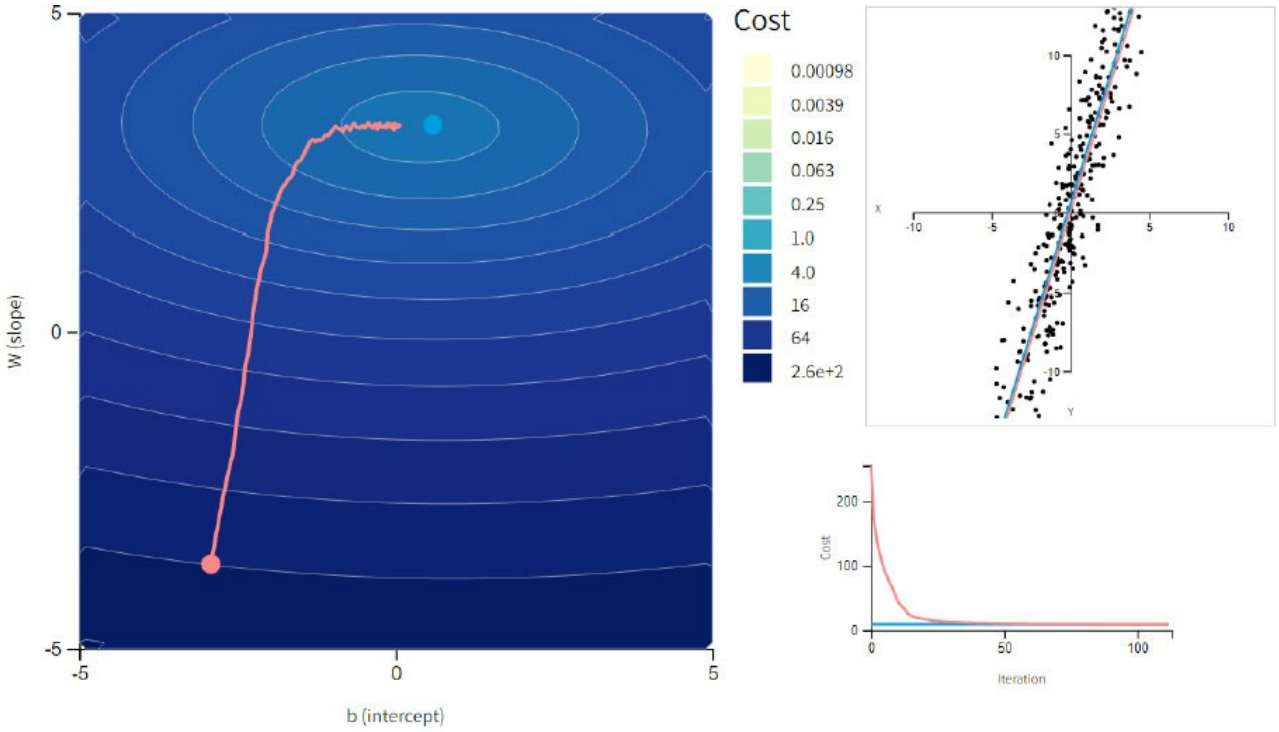
- In this method we still use estimation idea But use a batch of data instead of one point.

Mini-Batch Gradient:

$$\hat{g} = \frac{1}{|\mathcal{B}|} \sum_{d \in \mathcal{B}} \nabla_{\boldsymbol{\theta}} \mathcal{L}(d, \boldsymbol{\theta}), \quad \mathcal{B} \subset \mathcal{D}$$

- This is a better estimation than SGD.
- With this way we can get a performance boost from hardware optimization, especially when using GPUs.
- Batch size ($|\mathcal{B}|$) is a hyperparameter you need to tune.

Various GD types: MiniBatch Gradient Descent



Various GD types

Now that we know what a batch is, we can define epoch and iteration:

- One Epoch is when an entire dataset is passed forward and backward through the network only once.
- One Iteration is when a batch is passed forward and backward through the network.