

# Machine Learning (CE 40717)

## Fall 2024

Ali Sharifi-Zarchi

CE Department  
Sharif University of Technology

October 21, 2024



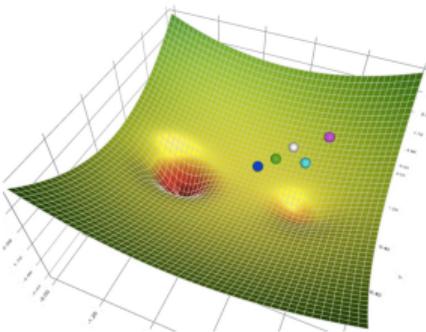
## 1 Gradient Descent

## ② Backpropagation

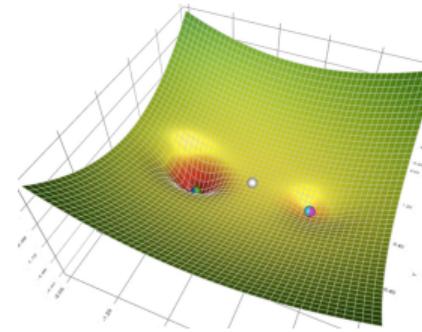
## 4 References

## How to Update Weights?

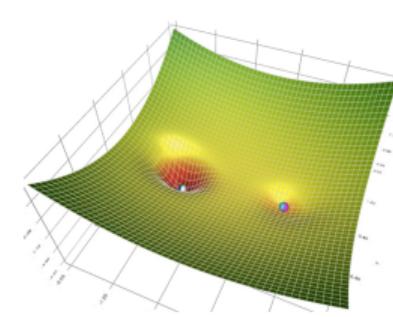
- Imagine training a large model like ChatGPT. It has billions of parameters that need to be adjusted.
  - If we used **random search** to update these weights, it would take an astronomical number of trials to find good parameters.
  - How to make training feasible at this massive scale?
  - Image adapted from Medium: AdaGrad Optimization Algorithm



## Visualizing parameter space with random initial points.



Random search approach in  
high-dimensional space.



Challenges in finding optimal weights using random search.

## How to Update Weights?

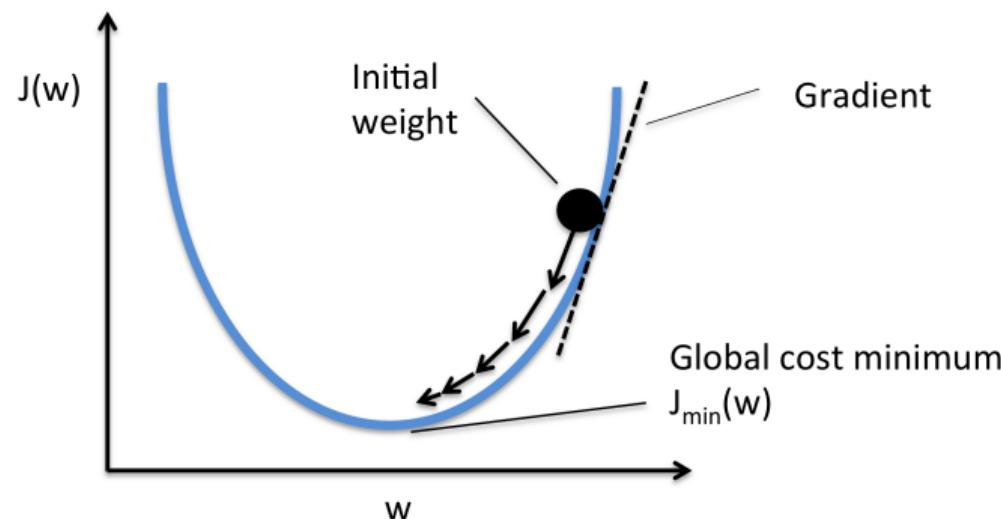
## Options for Updating Weights:

- **Random Search:** Tries values randomly—inefficient and impractical.
  - **Gradient Descent:** Follows the slope of the loss function—efficient and guided.

## Why Gradient Descent?

- It updates weights by following the slope, reducing error with each step.
  - Controlled, stepwise updates ensure we move closer to minimizing the loss effectively.

## How to Update Weights?



**Figure 1:** Gradient descent: adjusting weights to minimize cost by following the gradient.  
<https://sebastianraschka.com/images/faq/gradient-optimization/ball.png>

## Gradient Descent: Concept and Weight Updates

**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:

- $\eta$  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:  $L(w) = (\gamma - wx)^2$

**Example:** For  $x = 3$ ,  $y = 10$ , and  $w_0 = 2$ ,

## Gradient Calculation:

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

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

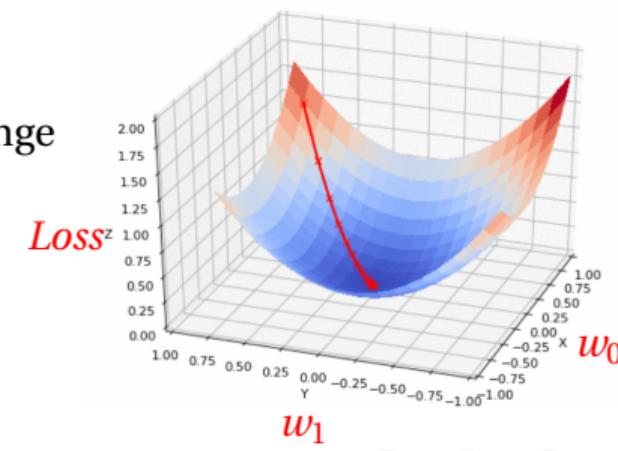
# Gradient Descent: Formula and Process

## Weight Update Formula:

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

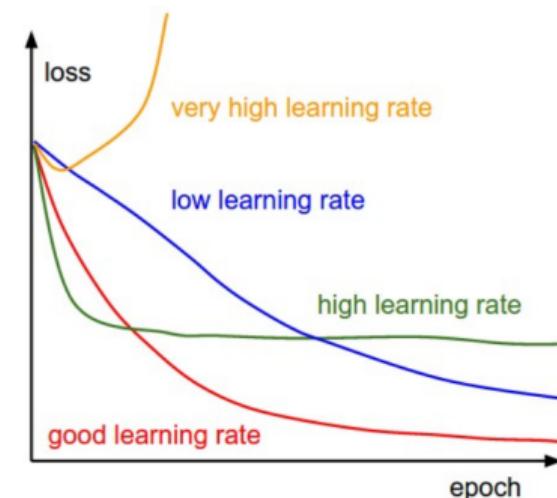
## Steps in Gradient Descent:

- 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



## 1 Gradient Descent

## 2 Backpropagation

Forward and Backward Passes

Vectorized Backpropagation

## 3 Foundations in Detail: Initialization, Loss, and Activation

## 4 References

## 1 Gradient Descent

## 2 Backpropagation

Forward and Backward Passes

Vectorized Backpropagation

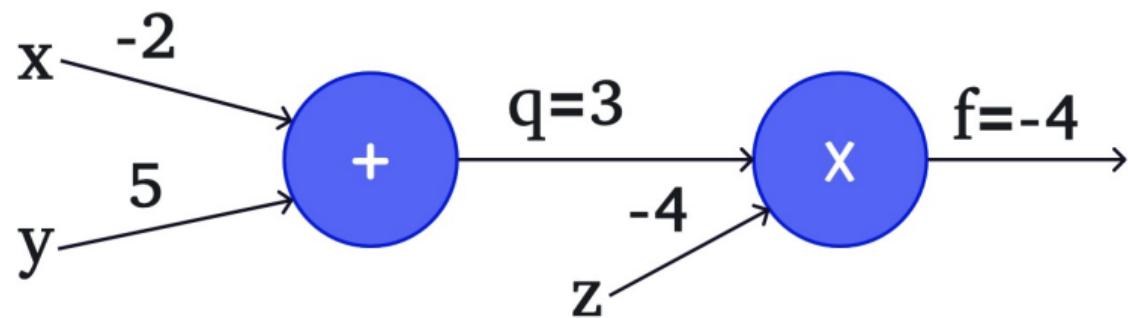
## 3 Foundations in Detail: Initialization, Loss, and Activation

## 4 References

## A Simple Example

## Function:

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



## A Simple Example: Forward Pass

## Function:

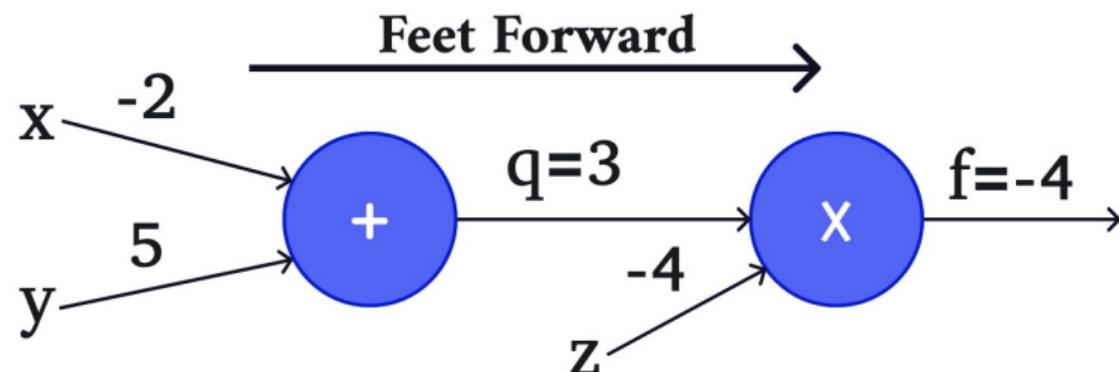
$$f(x, y, z) = (x + y)z$$

## Example:

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

### **Steps:**

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



# Chain Rule for Gradients

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.

# Finite Difference Method

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

- Change  $w_i$  by a small value  $\epsilon$ .
- 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.

# Backpropagation

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.

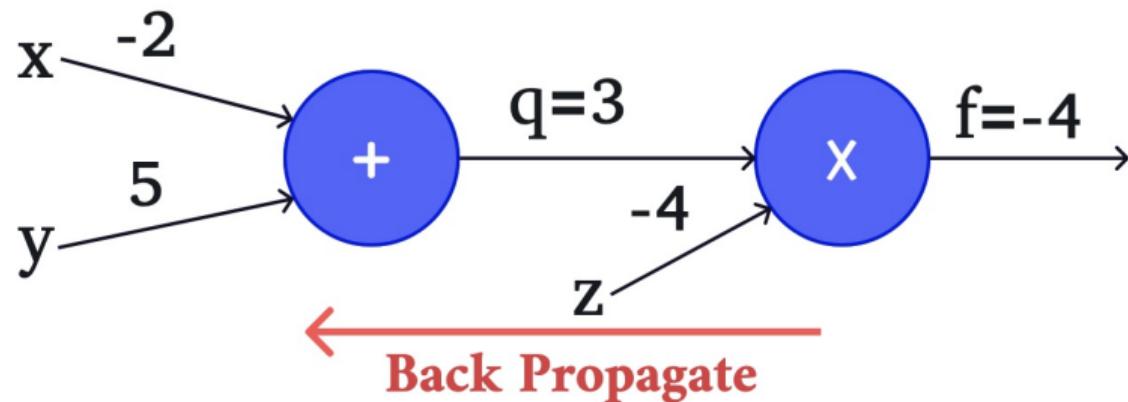
# Backpropagation: A Simple Example

**Function:**

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

**Example:**

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



## A Simple Example: Backpropagation

## Function:

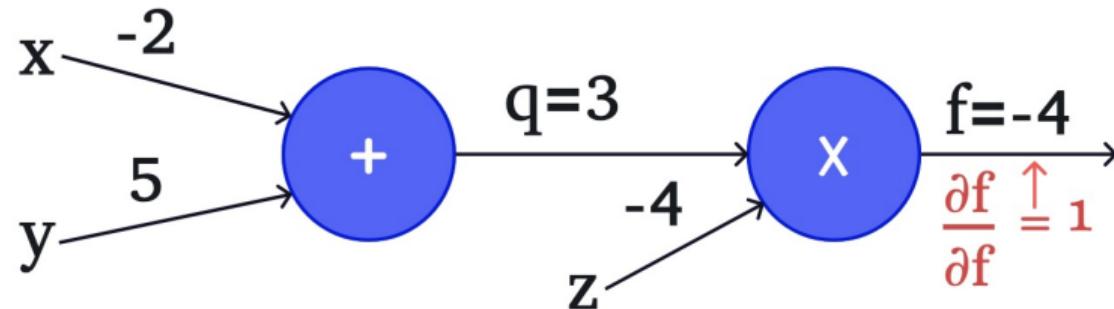
$$f(x, y, z) = (x + y)z$$

## Example:

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

## Step 1:

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



# A Simple Example: Backpropagation

## Function:

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

## Example:

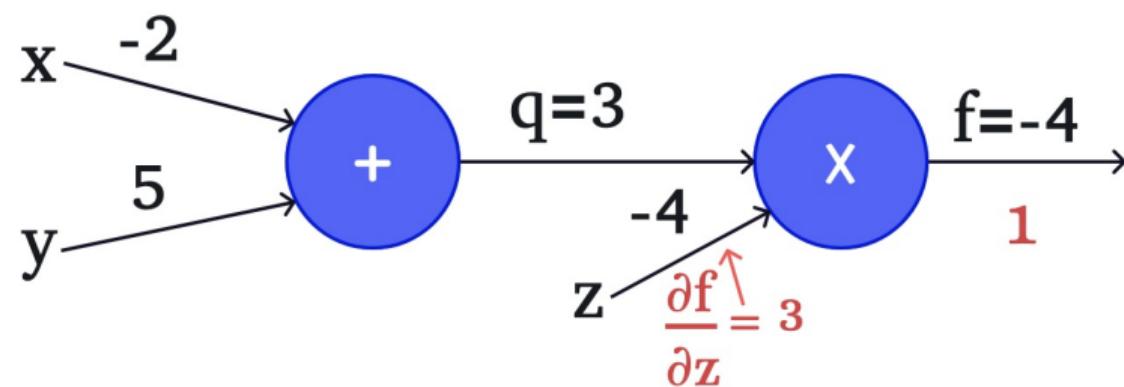
$$x = -2, \quad y = 5, \quad z = -4$$

### Step 1:

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

### Step 2:

$$f = qz, \quad \frac{\partial f}{\partial q} = z = -4, \quad \frac{\partial f}{\partial z} = q = 3$$



# A Simple Example: Backpropagation

## Function:

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

## Example:

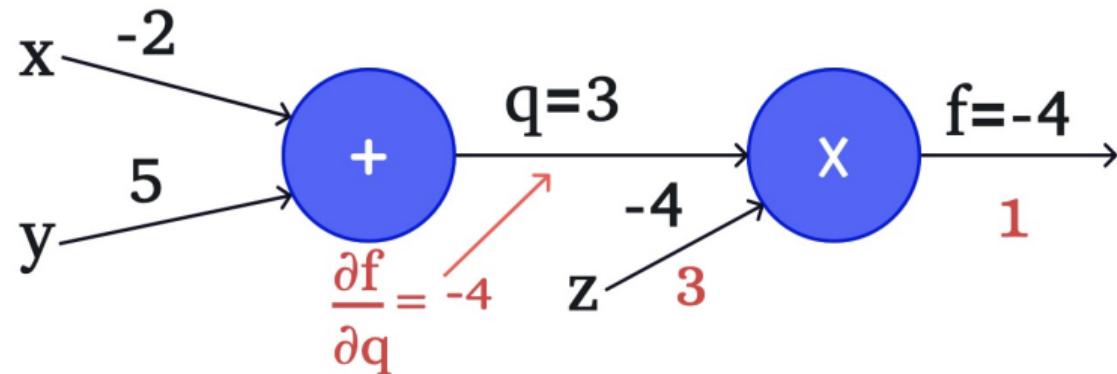
$$x = -2, \quad y = 5, \quad z = -4$$

### Step 1:

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

### Step 2:

$$f = qz, \quad \frac{\partial f}{\partial q} = z = -4, \quad \frac{\partial f}{\partial z} = q = 3$$



# A Simple Example: Backpropagation

## Function:

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

## Example:

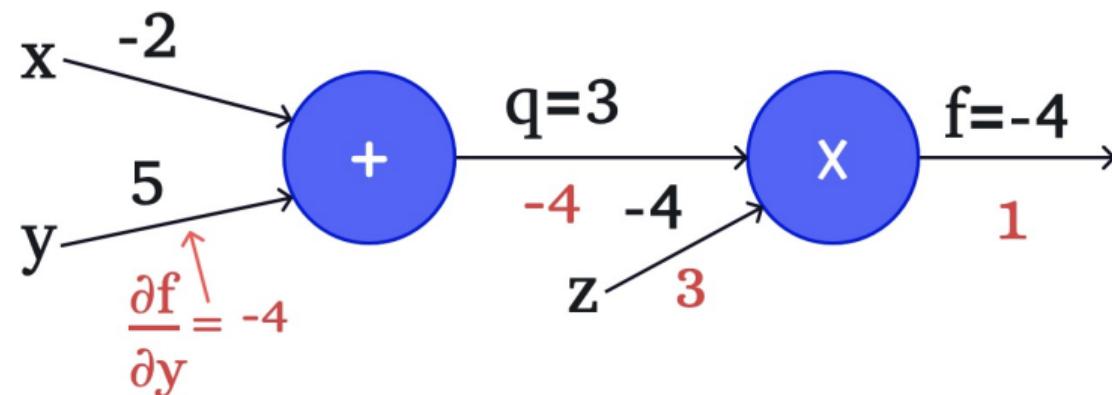
$$x = -2, \quad y = 5, \quad z = -4$$

### Step 1:

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

### Step 2:

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



# A Simple Example: Backpropagation

## Function:

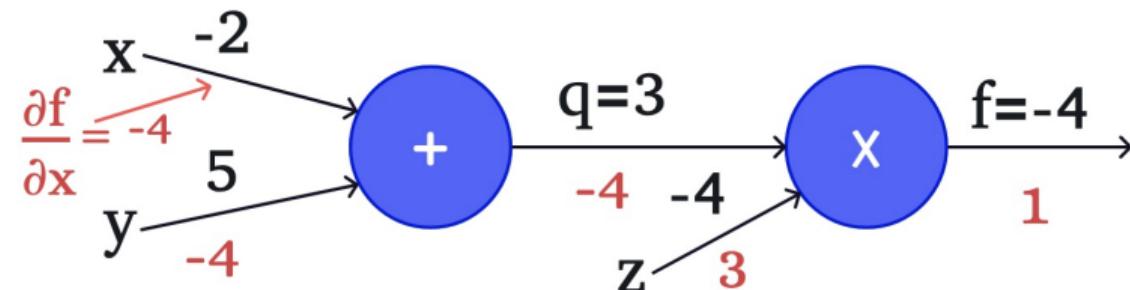
$$f(x, y, z) = (x + y)z$$

## Example:

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

### Step 1:

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



### Step 2:

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

# A Simple Example: Backpropagation

## Function:

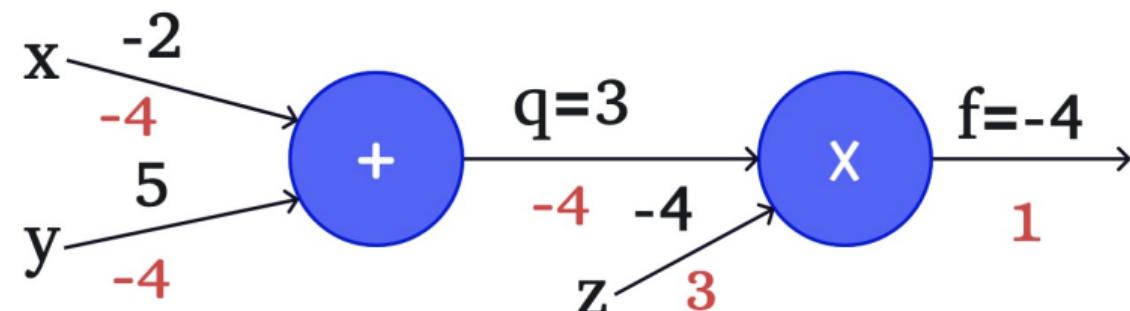
$$f(x, y, z) = (x + y)z$$

## Example:

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

### Step 1:

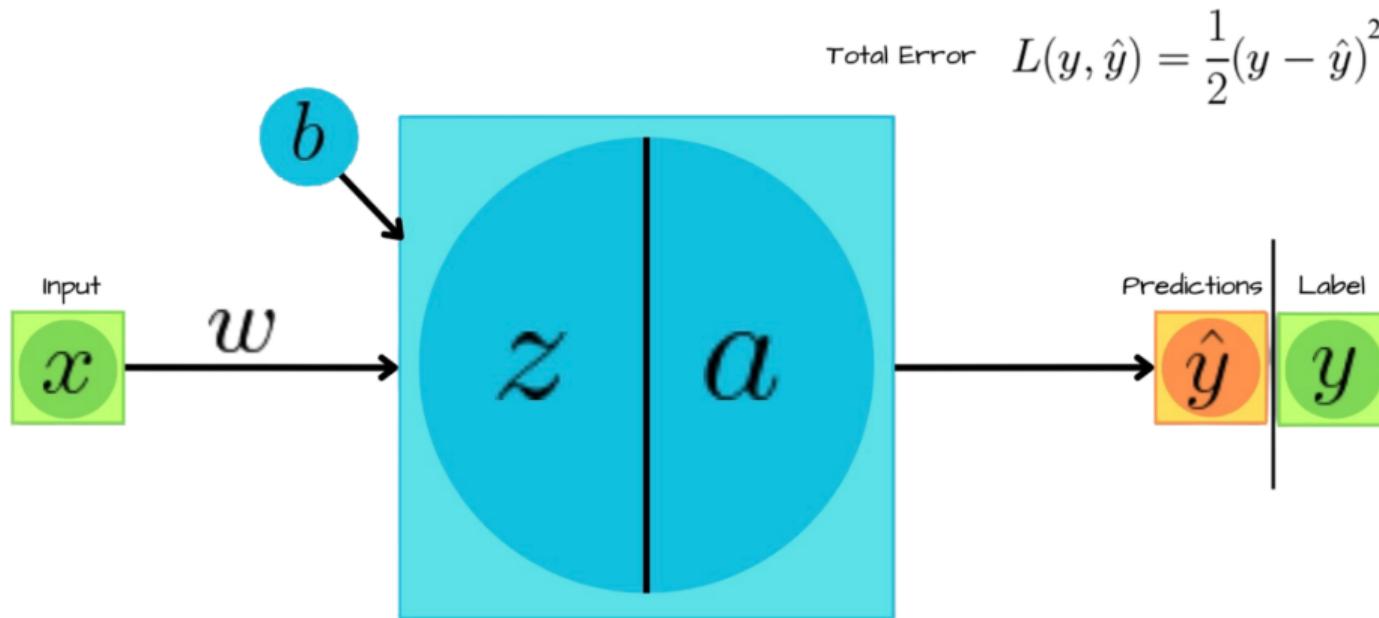
$$q = x + y, \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1$$



### Step 2:

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

## Example: One Neuron

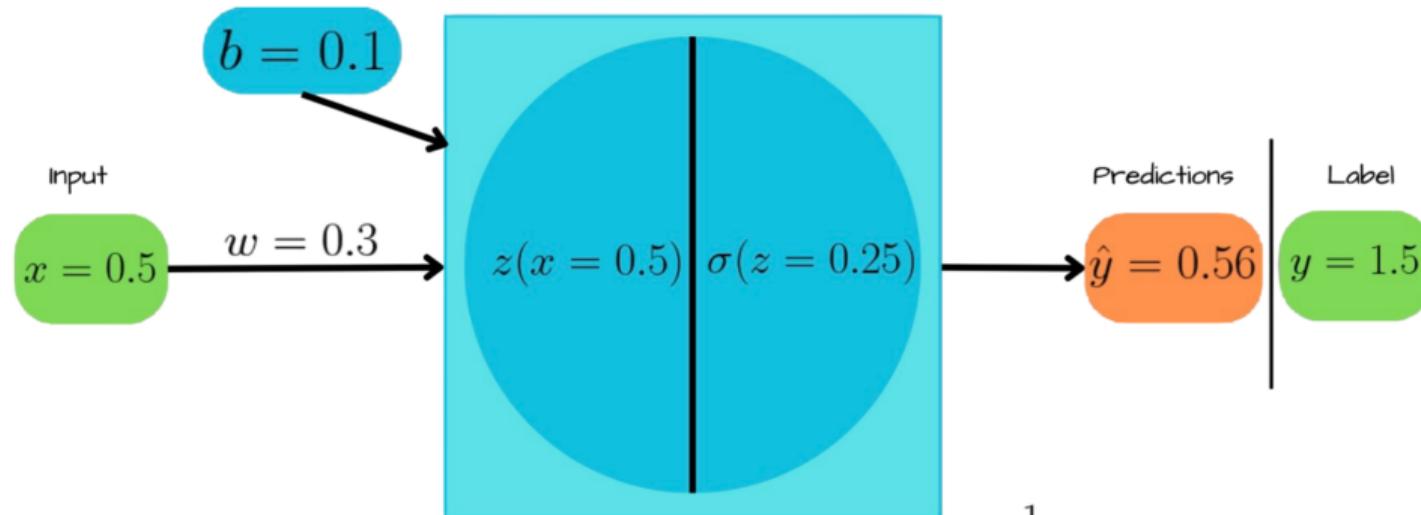


$$z(x) = wx + b \quad a = \sigma(z) = \frac{1}{1 + e^{-z}}$$

Image adapted from "Deep learning backpropagation" Web article

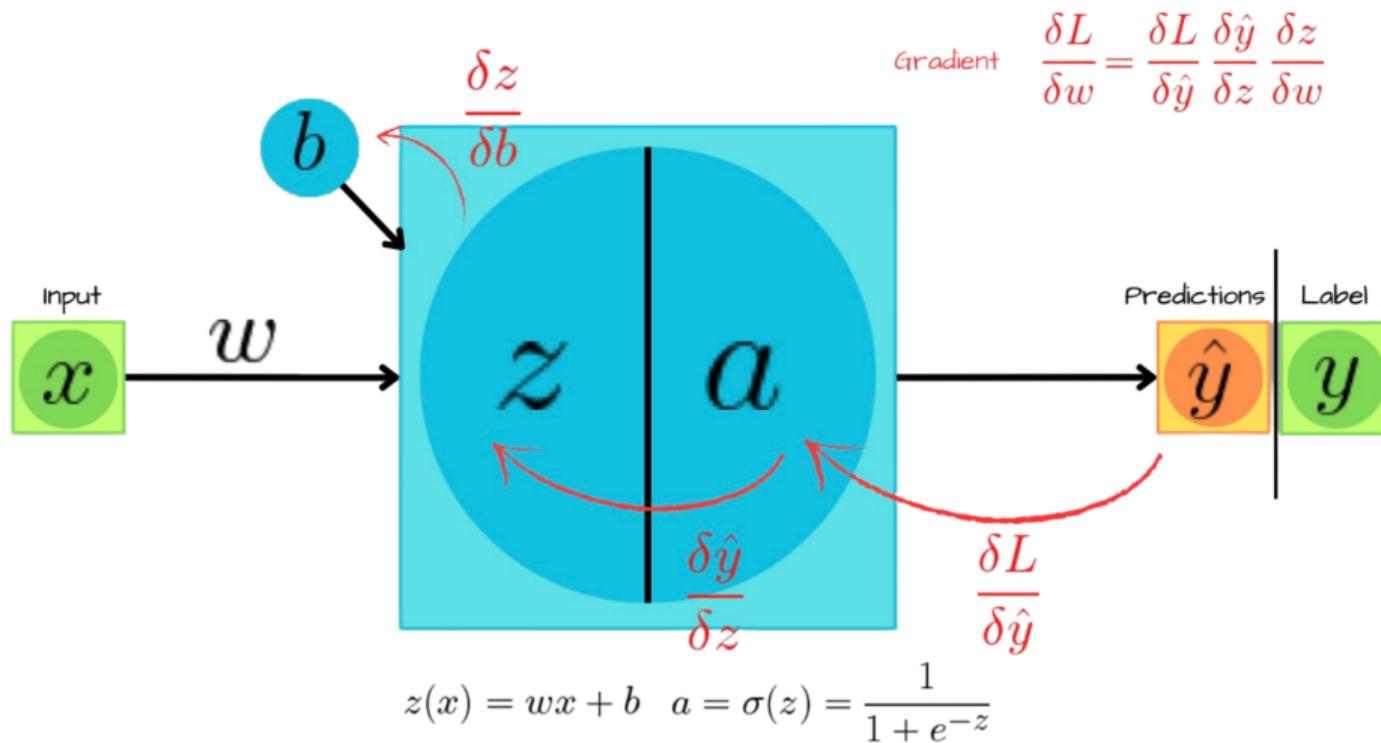
## Forward Pass

$$\text{Total Error } L(1.5, 0.56) = \frac{1}{2}(1.5 - 0.56)^2 = 0.44$$

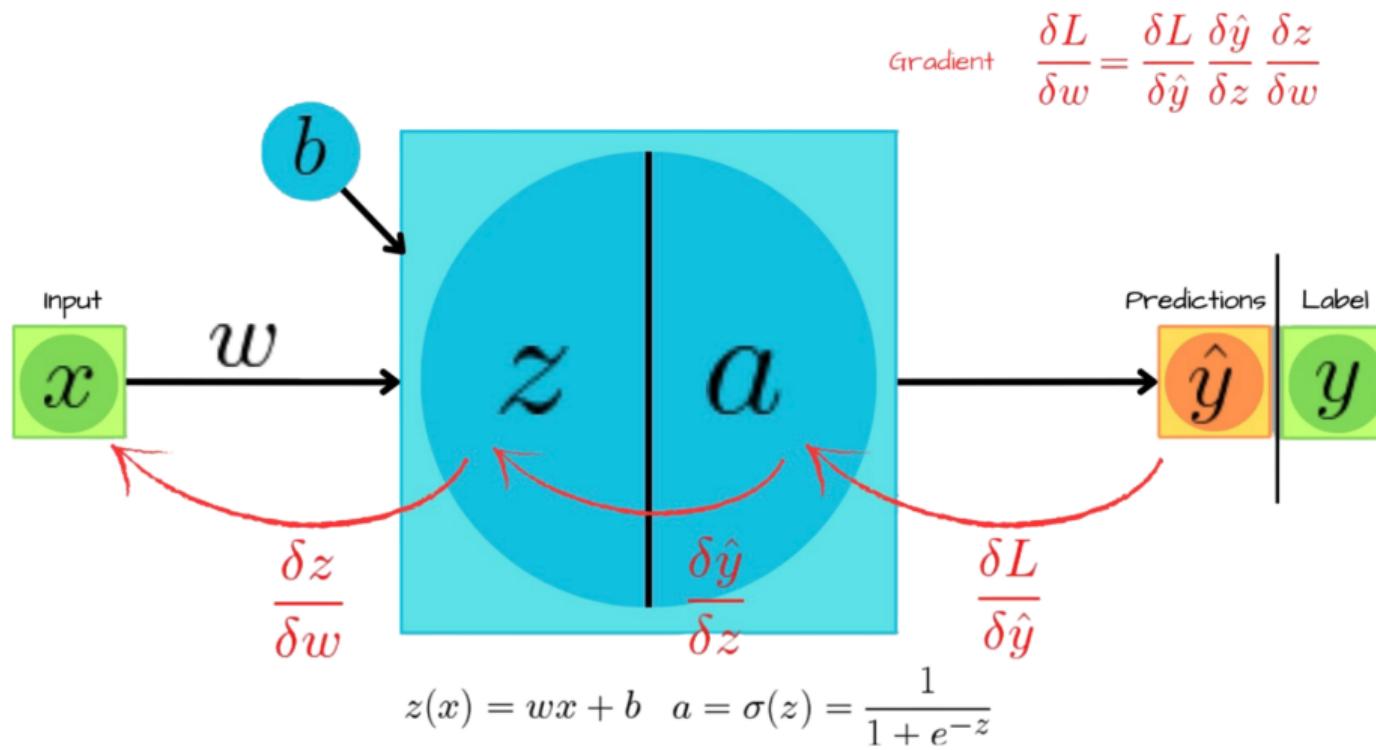


$$z(0.5) = 0.3 \cdot 0.5 + 0.1 = 0.25 \quad \sigma(z = 0.25) = \frac{1}{1 + e^{-0.25}} = 0.56$$

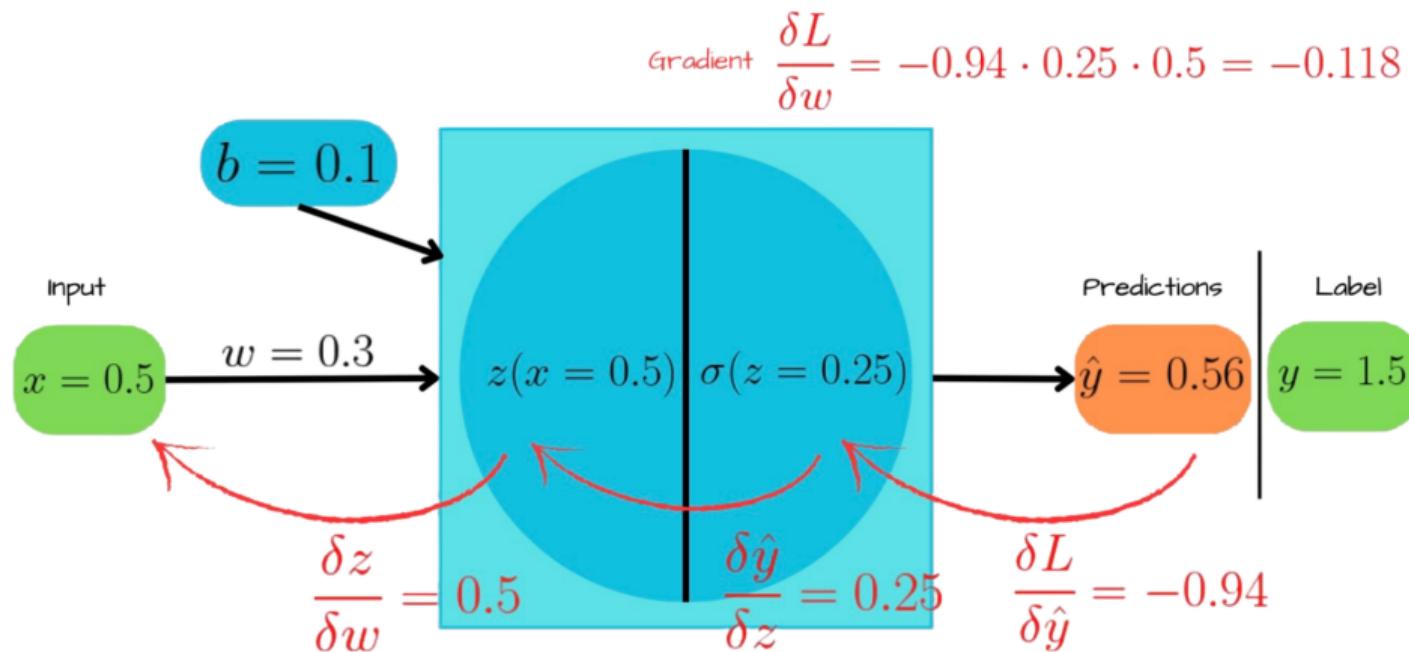
## Backward Pass



## Backward Pass



## Backward Pass



## 1 Gradient Descent

## 2 Backpropagation

Forward and Backward Passes

Vectorized Backpropagation

## 3 Foundations in Detail: Initialization, Loss, and Activation

## 4 References

# Vectorized Backpropagation

The faster you compute gradients, the quicker each parameter update in gradient descent.

**Derivative of a Vector by a Vector:** leveraging matrix operations for faster computation.

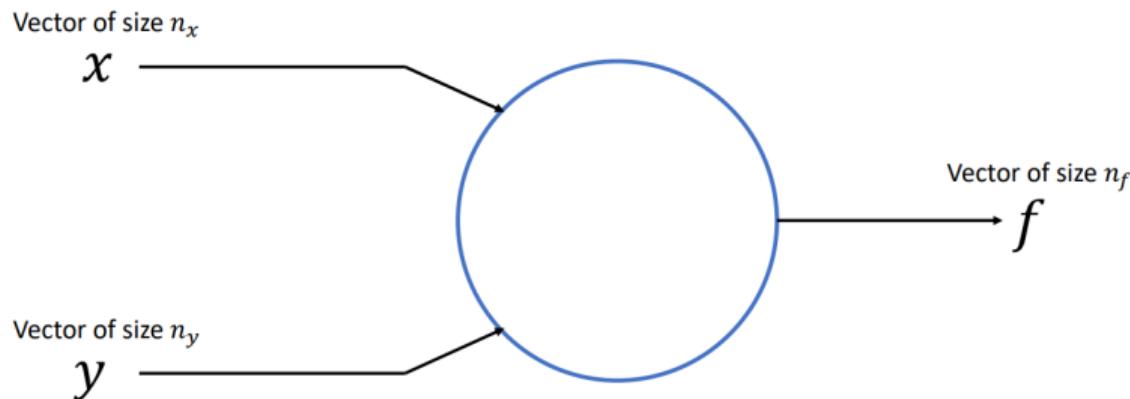
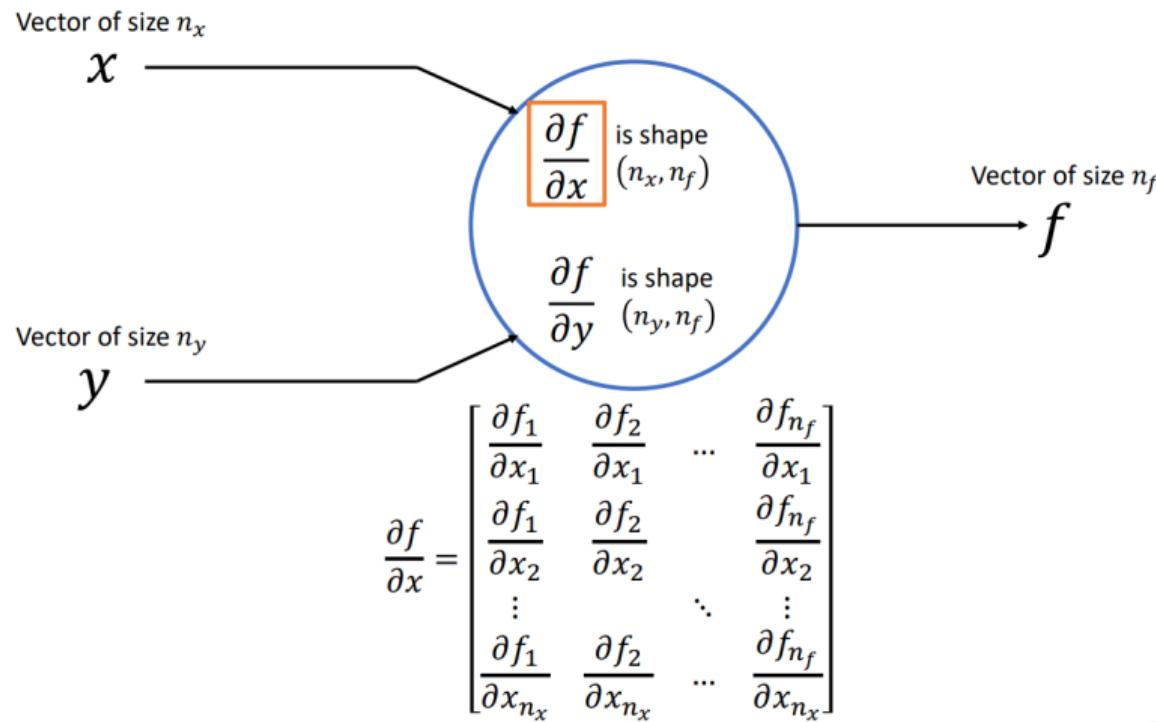


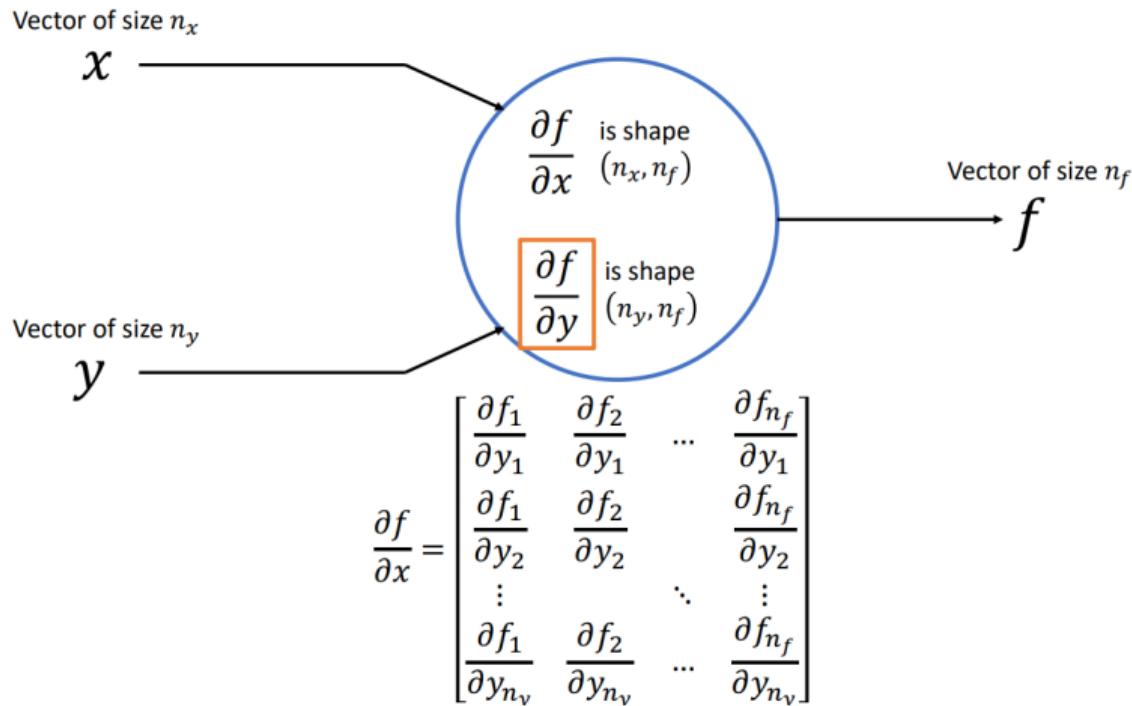
Image adapted from Elec502 vectorized backpropagation slides

# Vectorized Backpropagation

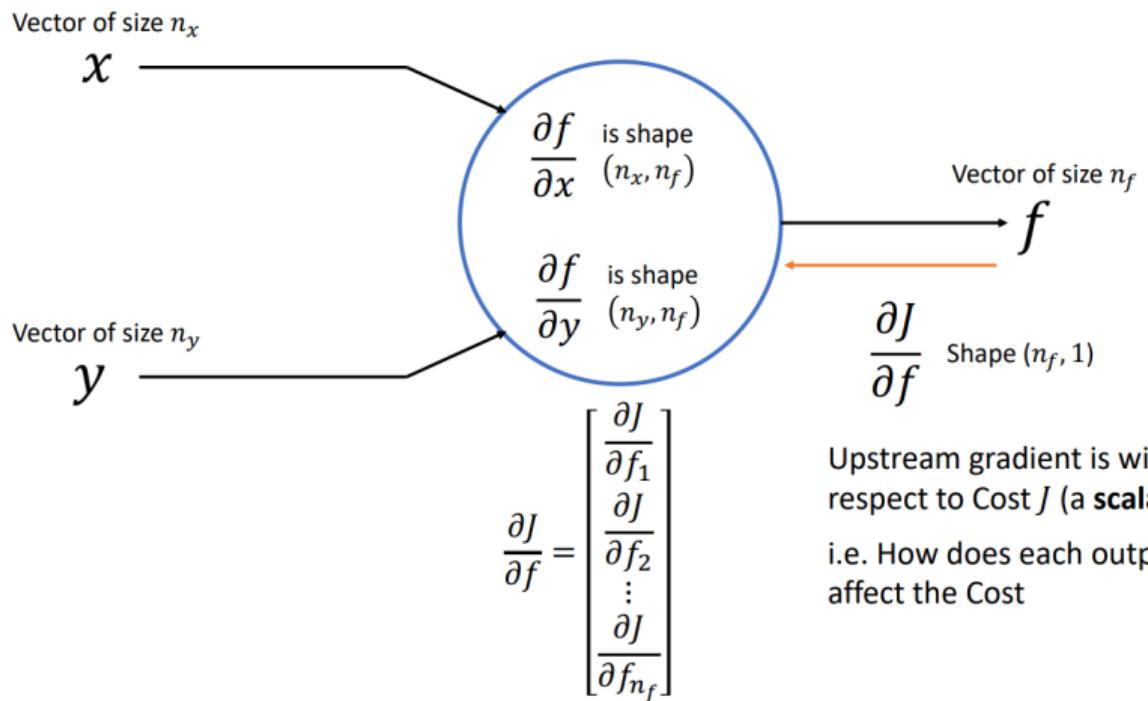
## Local Derivatives are Jacobian Matrices



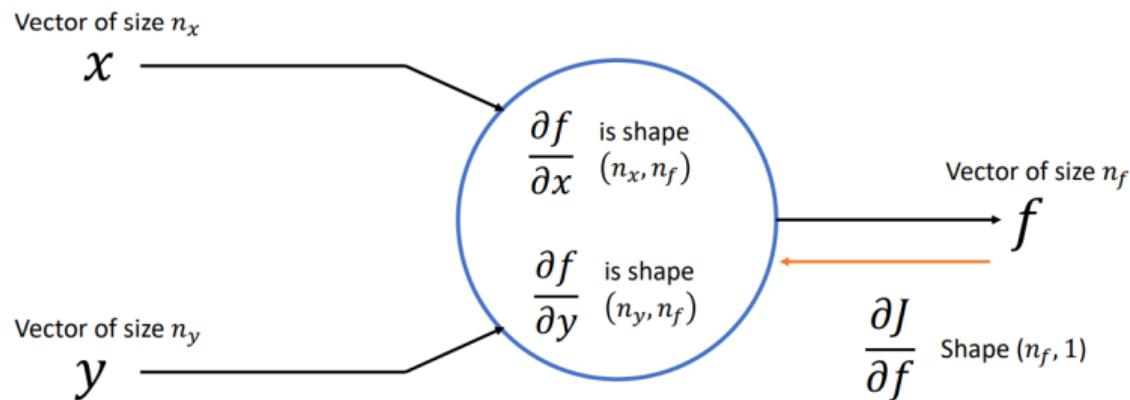
# Vectorized Backpropagation



# Vectorized Backpropagation

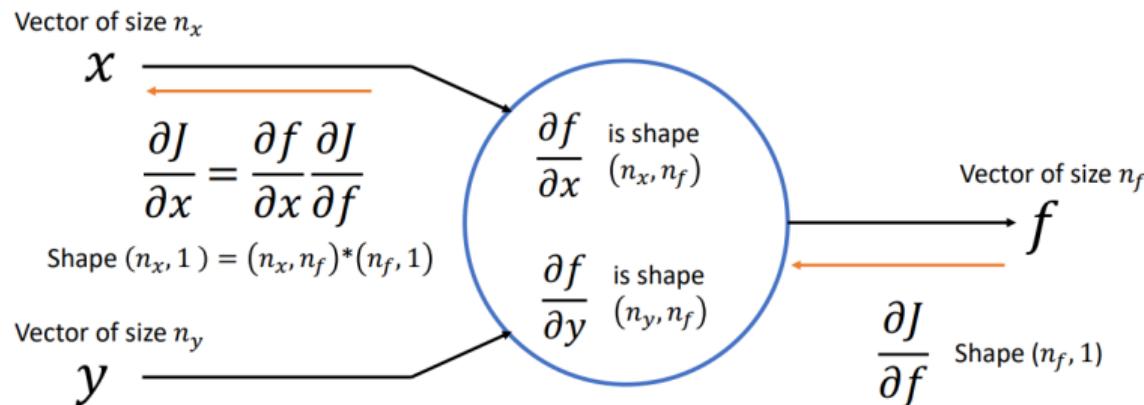


# Vectorized Backpropagation



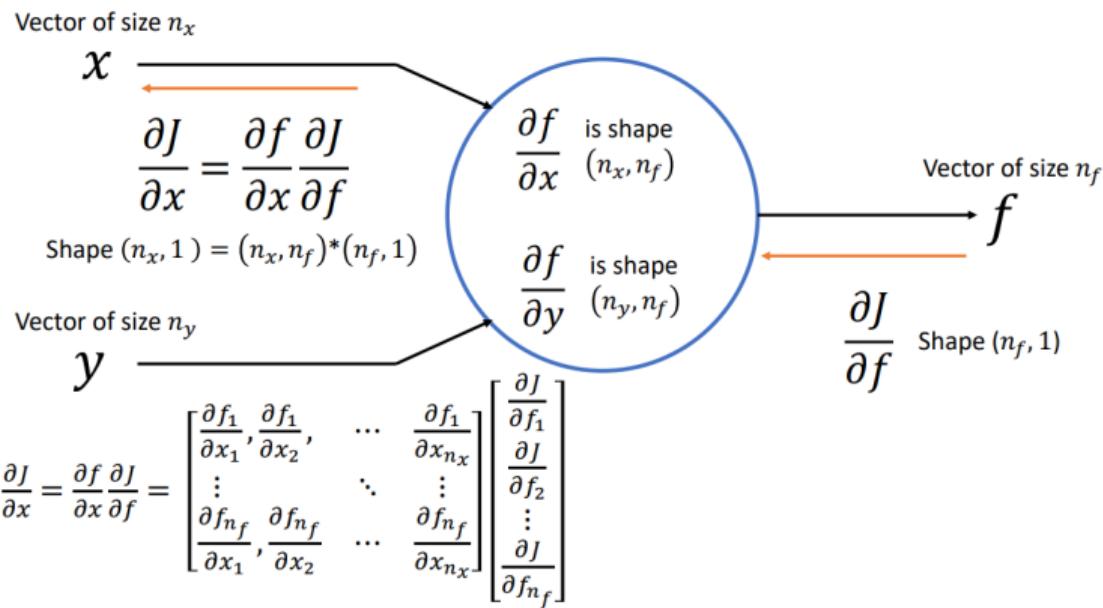
Apply chain rule like before!

# Vectorized Backpropagation



Applying the chain rule involves matrix-vector multiplication

# Vectorized Backpropagation



# Chain Rule – Matrix-Vector Multiply

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

$$\text{Shape } (n_x, 1) = (n_x, n_f) * (n_f, 1)$$

# Chain Rule – Matrix-Vector Multiply

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

Jacobian

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

# Chain Rule – Matrix-Vector Multiply

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

Jacobian

Upstream Gradient

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

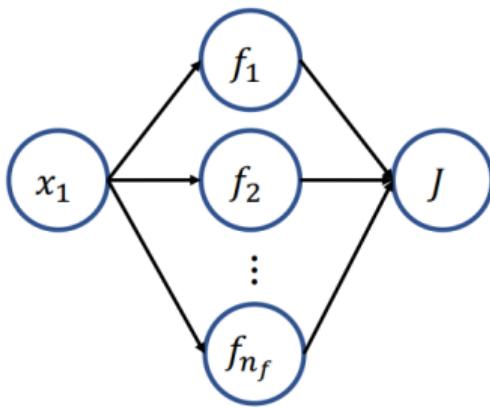
# Chain Rule – Matrix-Vector Multiply

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

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

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

## Chain Rule – Matrix-Vector Multiply

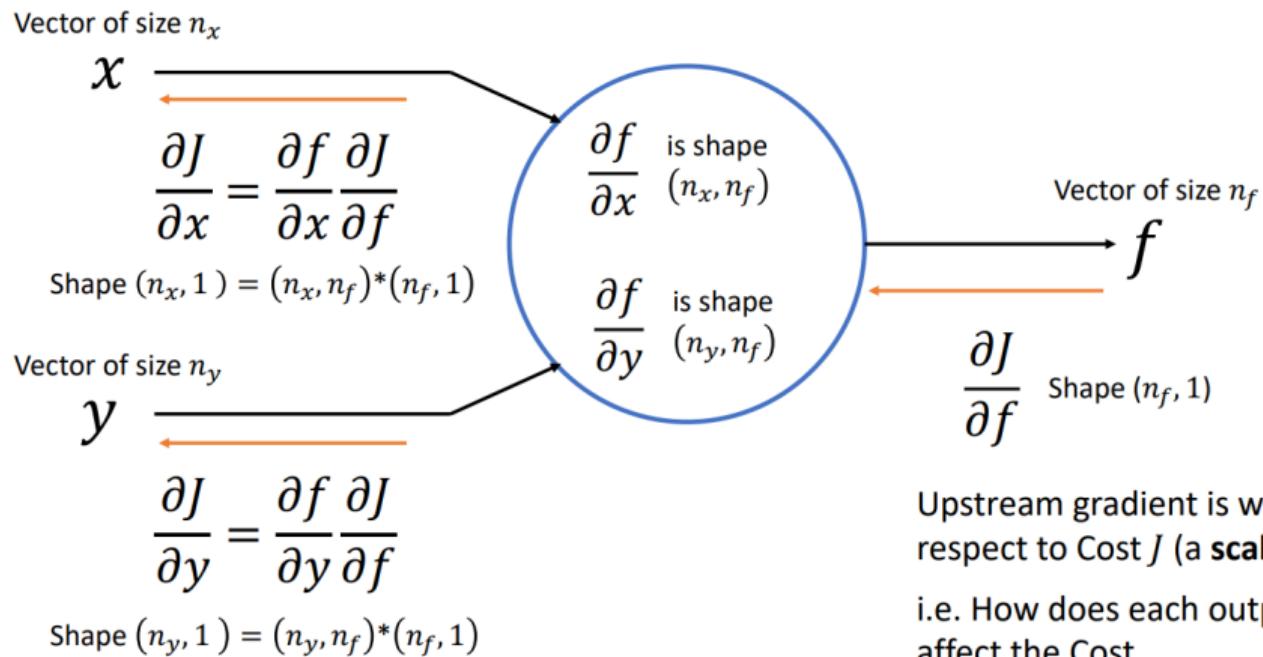


$$\frac{\partial J}{\partial x} = \frac{\partial f}{\partial x} \frac{\partial J}{\partial f} \rightarrow \begin{bmatrix} \frac{\partial J}{\partial x_1} \\ \frac{\partial J}{\partial x_2} \\ \vdots \\ \frac{\partial J}{\partial x_{n_x}} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1},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

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

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

# Chain Rule – Matrix-Vector Multiply



Chain Rule application is Matrix-Vector Multiply

## 1 Gradient Descent

## 2 Backpropagation

## 3 Foundations in Detail: Initialization, Loss, and Activation

Weight Initialization

Loss Functions

Activation Functions

## 4 References

## 1 Gradient Descent

## 2 Backpropagation

## 3 Foundations in Detail: Initialization, Loss, and Activation

Weight Initialization

Loss Functions

Activation Functions

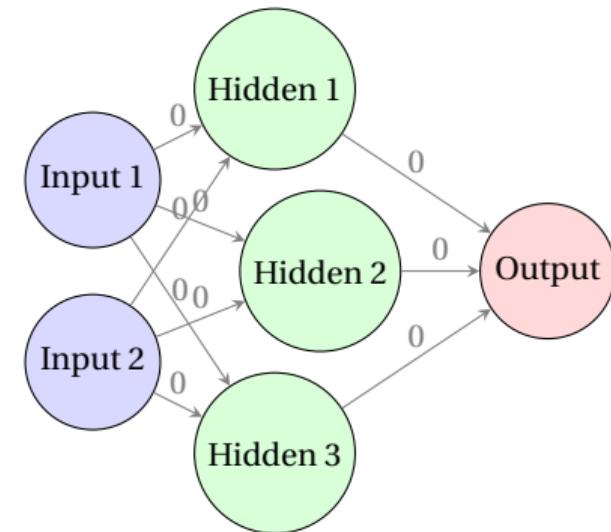
## 4 References

# Weight Initialization

**Example:** Imagine a network where all weights are initialized to zero.

**Issue:** If all weights are zero, each neuron in a layer will produce **identical** outputs. This symmetry prevents the network from learning **distinct features**, as every neuron updates identically.

**Solution:** To break this symmetry, weights need to be initialized with small random values, allowing neurons to learn unique features and avoid identical updates.



All weights initialized to zero

# Why Weight Initialization Matters

## Importance:

- Proper **initialization** ensures **faster convergence** and improves **training stability**.
- Prevents issues like **vanishing** or **exploding gradients**, which can make training slow or unstable.

**Question:** How can we initialize weights to maximize **learning efficiency** and prevent gradient problems?

# Zero Initialization and Random Initialization

## Zero Initialization

- **Description:** Set all weights to zero.
- **Key Point:** Rarely used, as it leads to identical updates for all neurons, preventing the network from learning distinct features.

## Random Initialization

- **Description:** Assign small random values to weights.
- **Distribution:** Typically, weights are initialized using a uniform or normal distribution.

$$w \sim \mathcal{U}(-\epsilon, \epsilon) \quad \text{or} \quad w \sim \mathcal{N}(0, \sigma^2)$$

- **Key Point:** Helps break symmetry but can still cause issues with gradient magnitudes.

# Xavier Initialization

**Description:** Xavier Initialization is designed to keep the variance of activations consistent across layers, ideal for sigmoid and tanh activations.

**Objective:** Prevents the shrinking or exploding of signal magnitudes during forward and backward propagation.

**Condition:**

$$\frac{1}{n_l} \text{Var}[w] = 1$$

**Initialization Scheme:**

$$w \sim \mathcal{U}\left(-\sqrt{\frac{1}{n_l}}, \sqrt{\frac{1}{n_l}}\right)$$

This results in a uniform distribution within the range  $-\sqrt{\frac{1}{n_l}}$  to  $\sqrt{\frac{1}{n_l}}$ , ensuring stable signal variance across layers.

# He Initialization

**Description:** He Initialization (or Kaiming Initialization) is designed for neural networks with ReLU activations, considering the non-linearity of these functions.

**Objective:** Aims to prevent the exponential growth or reduction of input signal magnitudes through layers.

**Condition:**

$$\frac{1}{2} n_l \text{Var}[w] = 1$$

**Initialization Scheme:**

$$w_l \sim \mathcal{N}\left(0, \frac{2}{n_l}\right)$$

This implies a zero-centered Gaussian distribution with a standard deviation of  $\sqrt{\frac{2}{n_l}}$ , where biases are initialized to 0.

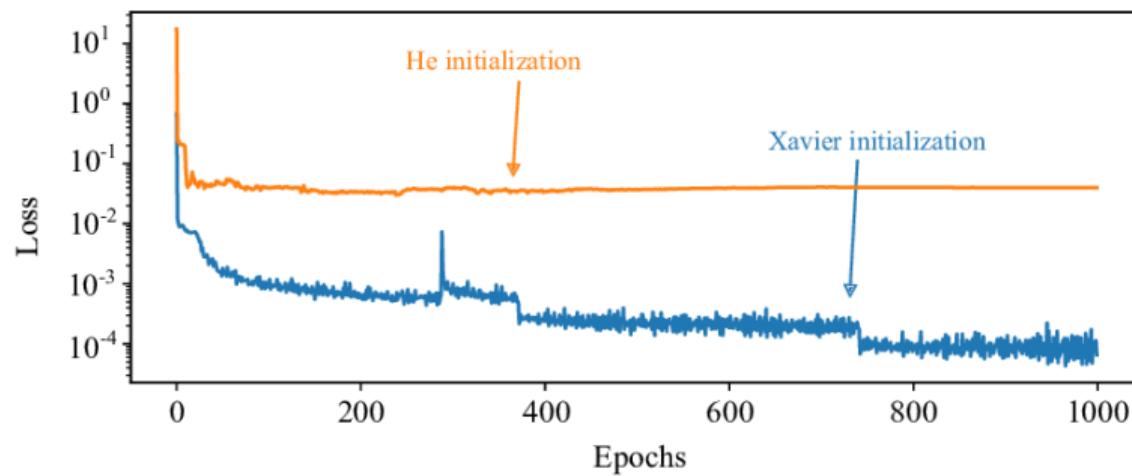
# Key Point Summary

Proper initialization:

- Reduces the risk of gradient issues (vanishing/exploding gradients).
- Helps the network converge faster.

## Xavier vs He

- Evolution of loss term for Xavier weight initialization and He weight initialization.
- Image adapted from A Knowledge-driven Physics-Informed Neural Network model; Pyrolysis and Ablation of Polymers



# Choosing the Right Initialization – Examples

- **Scenario 1:** Using ReLU activation functions in a deep network.
  - **Best Choice:** He Initialization.
  - **Reason:** Helps maintain gradient flow through the layers.
- **Scenario 2:** Using Sigmoid activation functions in a shallow network.
  - **Best Choice:** Xavier Initialization.
  - **Reason:** Keeps variance balanced, which is crucial for non-ReLU activations.
- **Experiment:** Try initializing with zeros and random weights to see how it impacts training speed and performance.

# Transition to Loss and Activation Functions

**Recap:** Proper weight initialization:

- Ensures stability during training by maintaining gradient magnitudes.
- Helps the network converge faster and learn more effectively.

**Next Steps:**

- Once weights are initialized, the network needs a measure of error — this is where **loss functions** come in.
- After initializing weights, **activation functions** determine the output of each neuron, enabling the network to learn complex patterns.

**Question:** How do we measure the error in predictions and adjust our weights to minimize it?

## 1 Gradient Descent

## 2 Backpropagation

## 3 Foundations in Detail: Initialization, Loss, and Activation

Weight Initialization

Loss Functions

Activation Functions

## 4 References

# 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 (MSE)

## Definition:

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

## 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:

$$\begin{aligned}\text{MSE} &= \frac{1}{3} [(5.0 - 4.2)^2 + (4.0 - 3.8)^2 + (4.9 - 5.1)^2] \\ &= \frac{1}{3} [0.64 + 0.04 + 0.04] = \frac{0.72}{3} = 0.24\end{aligned}$$

# Mean Absolute Error (MAE)

## Definition:

$$\text{MAE} = \frac{1}{N} \sum_{i=1}^N |y_i - \hat{y}_i|$$

## 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:

$$\begin{aligned}\text{MAE} &= \frac{1}{3} (|5.0 - 4.2| + |4.0 - 3.8| + |4.9 - 5.1|) \\ &= \frac{1}{3} (0.8 + 0.2 + 0.2) = \frac{1.2}{3} \approx 0.4\end{aligned}$$

# Impact on Model Outputs

## MSE Impact:

- Penalizes large errors heavily, leading to smoother outputs, useful in cases like refining blurry autoencoder images.

## MAE Impact:

- Treats errors uniformly, often resulting in sharper outputs and better handling of outliers.

# Gradient and Optimization Differences

- **MSE** promotes faster convergence for large errors due to its quadratic gradient.
- **MAE** provides a constant gradient, making optimization stable but slower with large errors.

# Summary

- **MSE** is suitable for models where large errors need higher penalization.
- **MAE** is better for robust models that handle outliers and avoid smoothing effects.
- Choice of loss function affects model behavior and output characteristics.

# Binary Classification Loss – Binary Cross-Entropy

## Binary Cross-Entropy:

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

### Example:

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

$$\begin{aligned}\mathcal{L}_{\text{BCE}} &= -\frac{1}{3} [1 \cdot \log(0.7) + (1 - 1) \cdot \log(1 - 0.7) \\ &\quad + 0 \cdot \log(0.3) + (1 - 0) \cdot \log(1 - 0.3) + 1 \cdot \log(0.9) + (1 - 1) \cdot \log(1 - 0.9)] \\ &\approx -\frac{1}{3} (\log(0.7) + \log(0.7) + \log(0.9)) \approx -\frac{1}{3} (-0.357 + -0.357 + -0.105) \approx 0.273\end{aligned}$$

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

# Categorical Cross-Entropy

## Categorical Cross-Entropy Formula:

$$L_{CCE} = -\frac{1}{n} \sum_{i=1}^n \sum_{c=1}^C y_{i,c} \log(\hat{y}_{i,c})$$

# 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 (3-Class)

**Given:**

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

$$\hat{y} = \begin{bmatrix} 0.1 & 0.7 & 0.2 \\ 0.6 & 0.3 & 0.1 \\ 0.1 & 0.6 & 0.3 \end{bmatrix}$$

# Solution

① True labels:

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

② Calculation:

$$L_{CCE} = -\frac{1}{3} (\log(0.7) + \log(0.6) + \log(0.3))$$

## Solution Continued

### Compute Log Terms:

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

The categorical cross-entropy loss for the given data is:

$$L_{CCE} = \frac{1}{3} \times 2.072 \approx 0.691$$

## 1 Gradient Descent

## 2 Backpropagation

## 3 Foundations in Detail: Initialization, Loss, and Activation

Weight Initialization

Loss Functions

Activation Functions

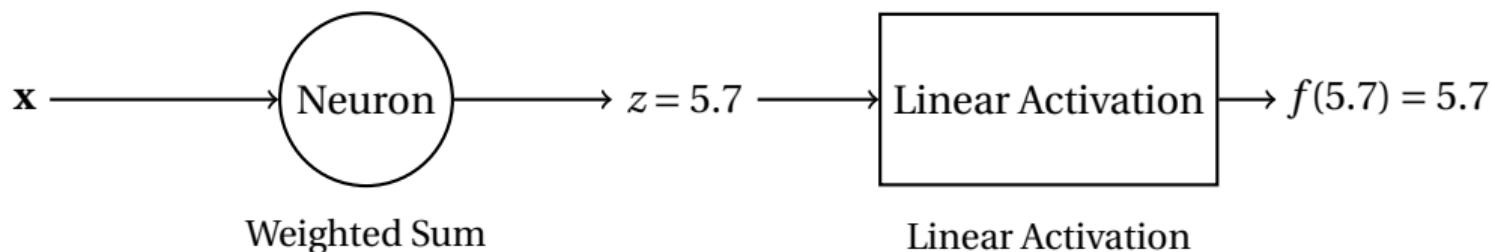
## 4 References

# Linear Activation - A Limitation

## Linear Activation:

$$f(z) = z$$

- Example: If a neuron produces a raw output  $z = 5.7$ , linear activation would pass this unchanged.



# Limitation of Linear Activation

**Why Transform Outputs?** Raw outputs need to be transformed into meaningful values, such as probabilities.

**The Problem:** Linear activation lacks non-linearity, restricting the model to simple linear relationships.

# Neural Networks: Why is the Max Operator Important?

- **Before:** Linear score function:

$$f = Wx$$

- **Now:** 2-layer Neural Network:

$$f = W_2 \max(0, W_1 x)$$

- The function  $\max(0, z)$  is called an activation function (in this case, ReLU).
- **Q:** What if we try to build a neural network without an activation function?

$$f = W_2 W_1 x$$

$$W_3 = W_2 W_1 \in \mathbb{R}^{C \times H}, \quad f = W_3 x$$

- **A:** We end up with a linear classifier again!

# ReLU

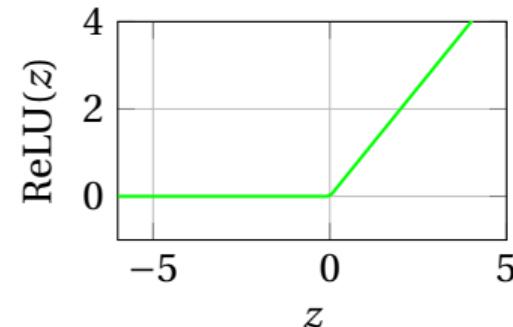
## Characteristics of ReLU:

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

- 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:

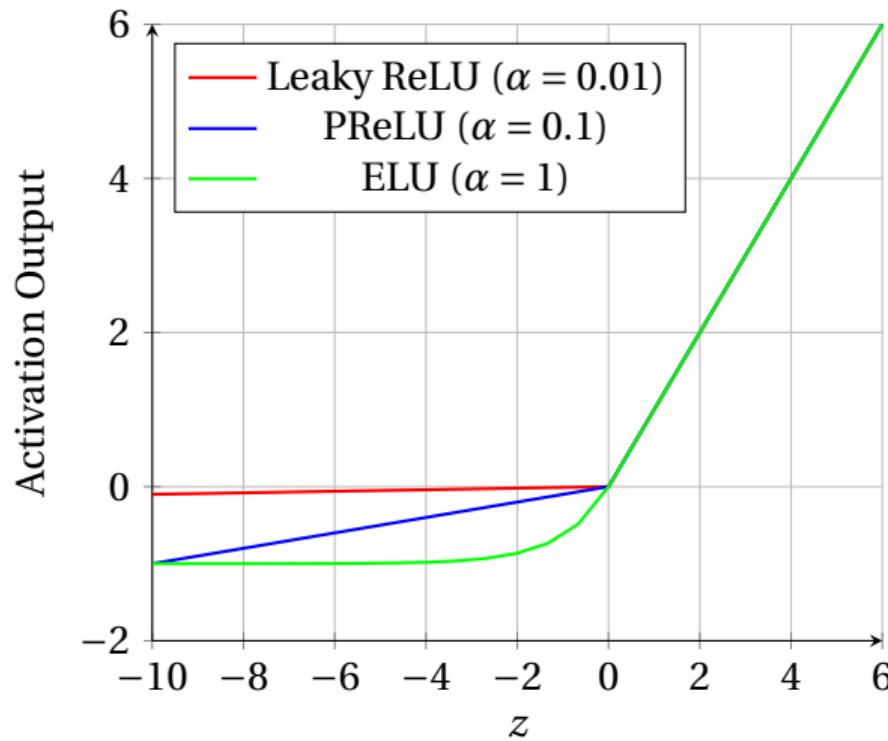
- **Dead ReLU Problem:** Neurons can become inactive during training, outputting 0 for all inputs if they receive negative values consistently.

## Question:

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

## Variants of ReLU: Leaky ReLU, PReLU, ELU

### Leaky ReLU, PReLU, and ELU Activation Functions

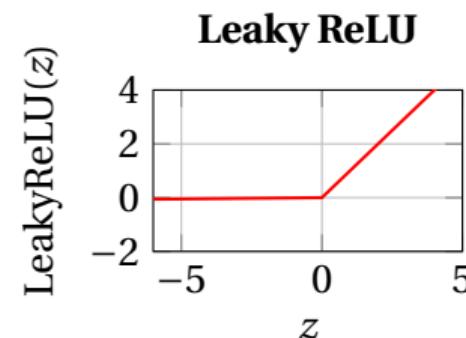


## Variants of ReLU: Leaky ReLU

- 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.



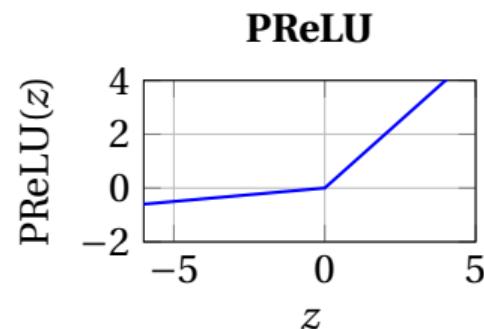
**Leaky ReLU:** Allows a small, non-zero gradient for negative inputs.

## Variants of ReLU: PReLU (Parametric ReLU)

- Similar to Leaky ReLU, but the slope for negative inputs ( $\alpha$ ) is learned during training.

$$\text{PReLU}(z) = \max(\alpha z, z), \quad \alpha \text{ is learned}$$

- Provides more flexibility by adjusting the slope for negative inputs based on data.



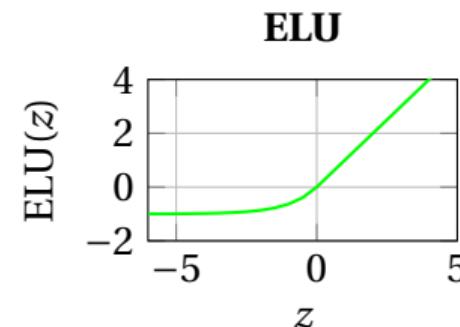
**PReLU:** Similar to Leaky ReLU, with a learnable slope.

## Variants of ReLU: ELU (Exponential Linear Unit)

- Similar to ReLU for positive values but smoother for negative inputs.

$$\text{ELU}(z) = \begin{cases} z, & \text{if } z > 0 \\ \alpha(e^z - 1), & \text{if } z \leq 0 \end{cases}, \quad \alpha = 1$$

- Provides faster convergence and reduces bias shift by smoothing negative values.



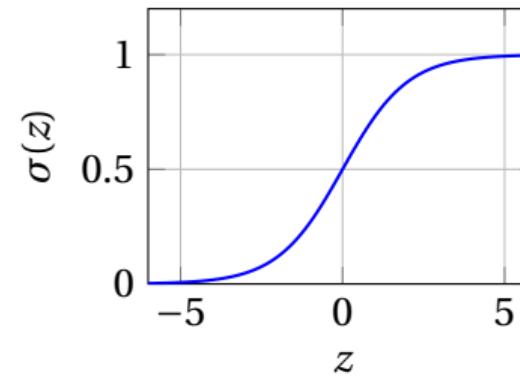
**ELU:** Smoother than ReLU for negative values.

# Sigmoid

## Characteristics of Sigmoid:

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

- 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.

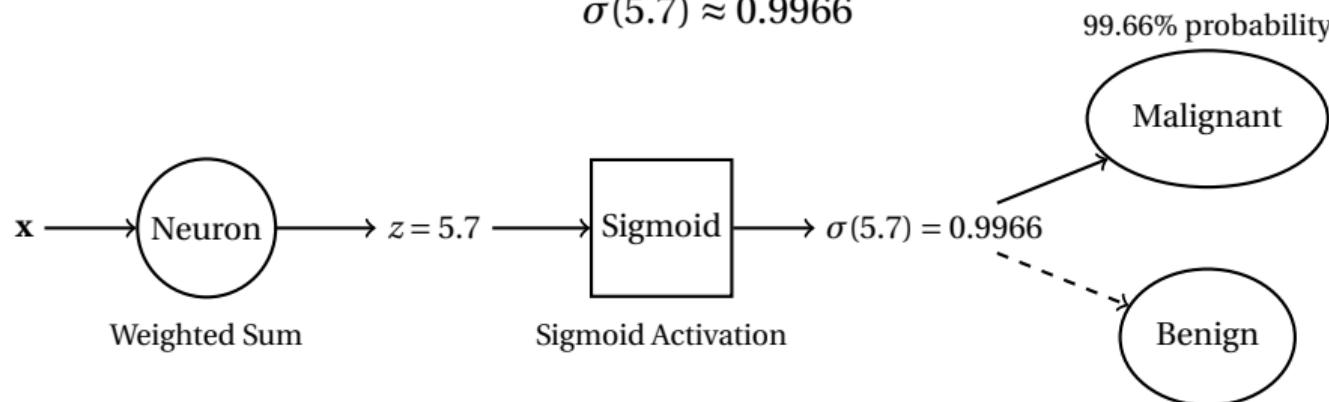


# Classification: Tumor Detection (Malignant vs. Benign)

**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 (Hyperbolic Tangent)

## 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)

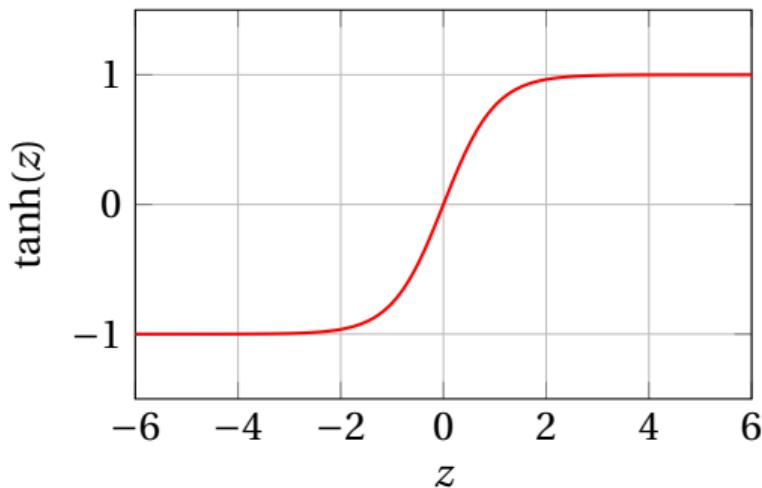
## Advantages of Tanh:

- **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).

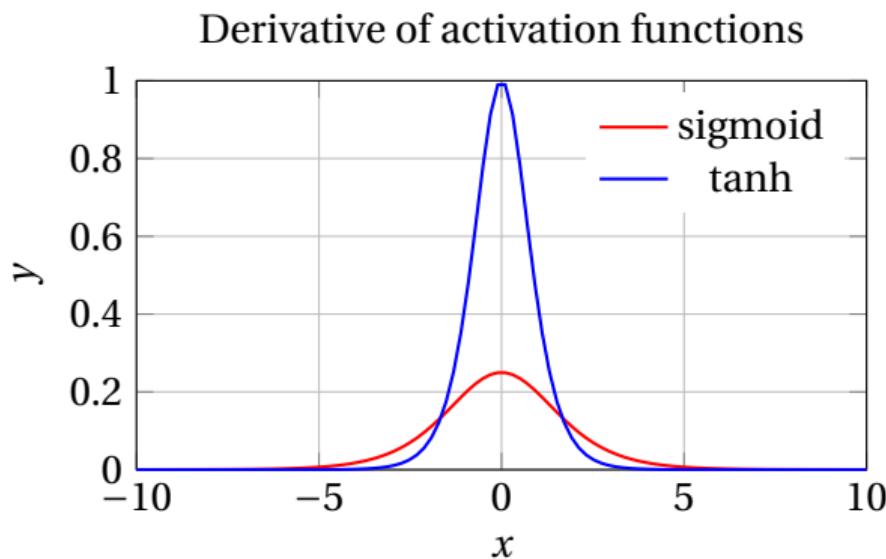
# Tanh (Hyperbolic Tangent)



## Question:

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

# Comparison: Sigmoid vs Tanh

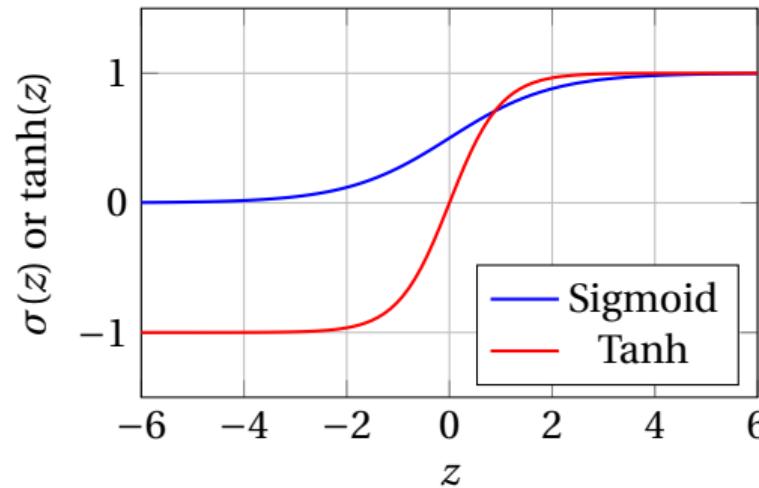


- The derivative of the Tanh function has a much steeper slope at  $x = 0$ , meaning it provides a larger gradient for backpropagation compared to the Sigmoid function.

# 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.



# Comparison: Sigmoid vs Tanh

## When to Use:

- **Sigmoid:** Best for binary classification tasks, particularly in the output layer.
- **Tanh:** More suitable for hidden layers due to its centered output, allowing faster training.

## Question:

- In what scenario might Sigmoid be preferred over Tanh, despite its limitations?

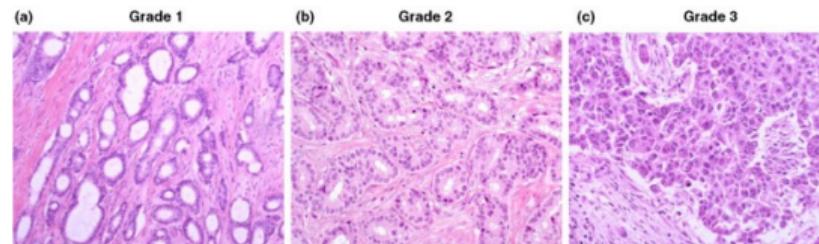
# Problem: Multi-Class Tumor Classification

**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 Activation: The Model

In multi-class classification, Softmax 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:

- $z_i$  is the raw output (logit) for class  $i$ .
- $K$  is the number of classes (in this case, 3: 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 Calculation

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 Probabilities

### 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).

# Conclusion: Softmax Activation for Classification

## 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.

## 1 Gradient Descent

## 2 Backpropagation

## 3 Foundations in Detail: Initialization, Loss, and Activation

## 4 References

# Contributions

- **This slide has been prepared thanks to:**
  - Donya Navabi
  - Mohammad Aghaei
  - Sogand Salehi

- [1] T. Networks, “Benign vs malignant tumors.” *Web article*, 2023.  
<https://www.technologynetworks.com/cancer-research/articles/benign-vs-malignant-tumors-364765>.
- [2] B. Central, “Breast cancer research article.” *Journal article*, 2010.  
<https://breast-cancer-research.biomedcentral.com/articles/10.1186/bcr2607>.
- [3] S. Raschka, “Gradient optimization image.” *Image*, 2020.  
<https://sebastianraschka.com/images/faq/gradient-optimization/ball.png>.
- [4] Xnought, “Backpropagation explainer.” *Website*, 2022.  
<https://xnought.github.io/backprop-explainer/>.
- [5] Datamapu, “Deep learning backpropagation.” *Web article*, 2021.  
[https://datamapu.com/posts/deep\\_learning/backpropagation/](https://datamapu.com/posts/deep_learning/backpropagation/).

- [6] B. Quinton, “Elec502 vectorized backpropagation slides.” *Lecture slides*, 2021.  
[https://people.ece.ubc.ca/bradq/ELEC502Slides/  
ELEC502-Part5VectorizedBackpropagation.pdf](https://people.ece.ubc.ca/bradq/ELEC502Slides/ELEC502-Part5VectorizedBackpropagation.pdf).
- [7] ResearchGate, “Evolution of loss term for xavier and he weight initialization.”  
*ResearchGate image*, 2023.  
[https://www.researchgate.net/publication/363843256/figure/fig7/AS:  
11431281086185056@1664161711792/  
Evolution-of-loss-term-for-Xavier-weight-initialization-and-He-weight-init  
png](https://www.researchgate.net/publication/363843256/figure/fig7/AS:11431281086185056@1664161711792/Evolution-of-loss-term-for-Xavier-weight-initialization-and-He-weight-init.png).
- [8] G. Groups, “Caffe users group discussion.” *Online Discussion*, 2024.  
[https://groups.google.com/g/caffe-users/c/wvyP1B\\_VMwM?pli=1.](https://groups.google.com/g/caffe-users/c/wvyP1B_VMwM?pli=1)

# Any Questions?