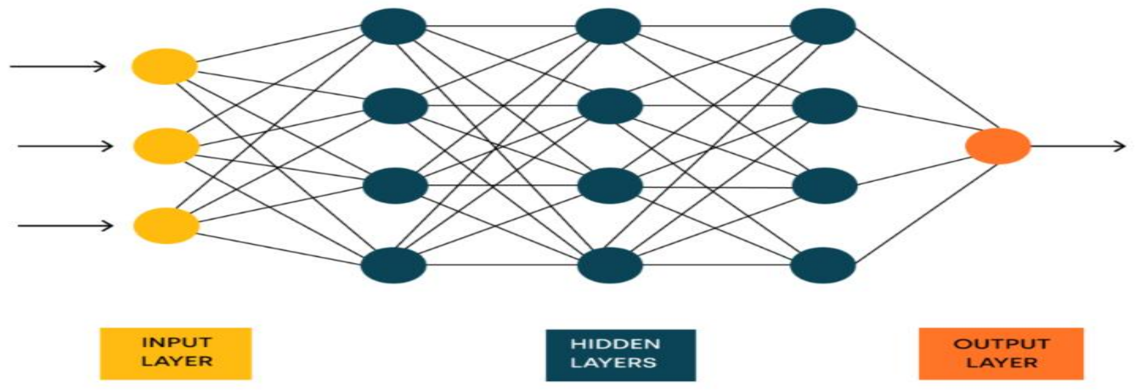
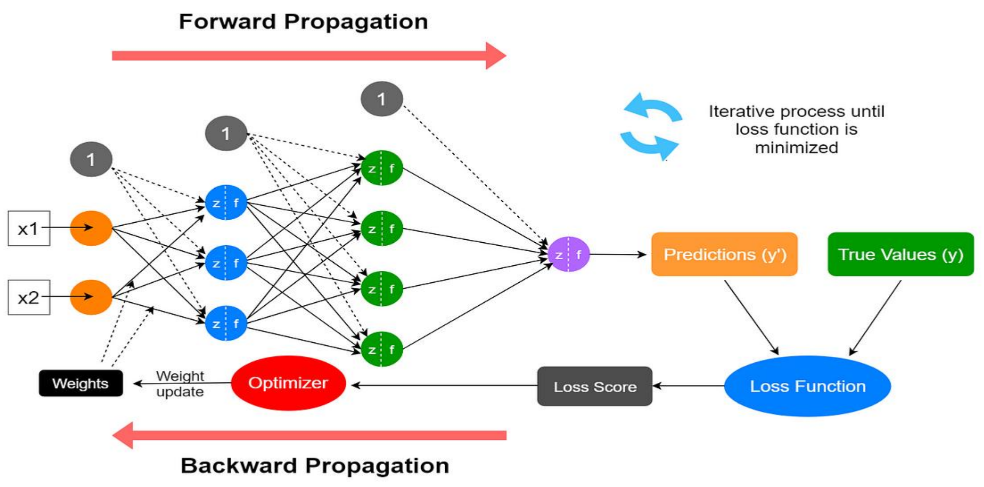
HANDOUT: BACKWARD PROPAGATION ALGORITHM

1) Neural Network

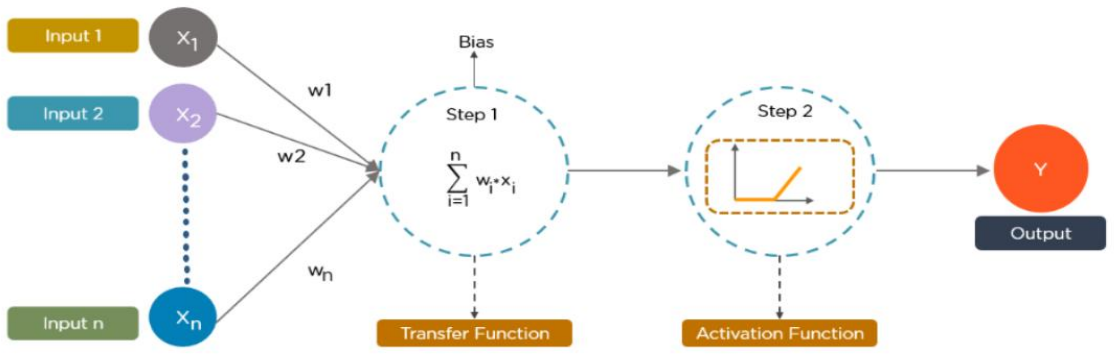


2) Three Steps in Neural Network Working

1. **Forward Propagation** – The input data is passed through the network layer by layer to produce an output or prediction.
2. **Loss Calculation** – The predicted output is compared with the actual value to measure the error or difference.
3. **Backward Propagation** – The error is propagated backward through the network so that weights and biases can be updated to reduce the loss.

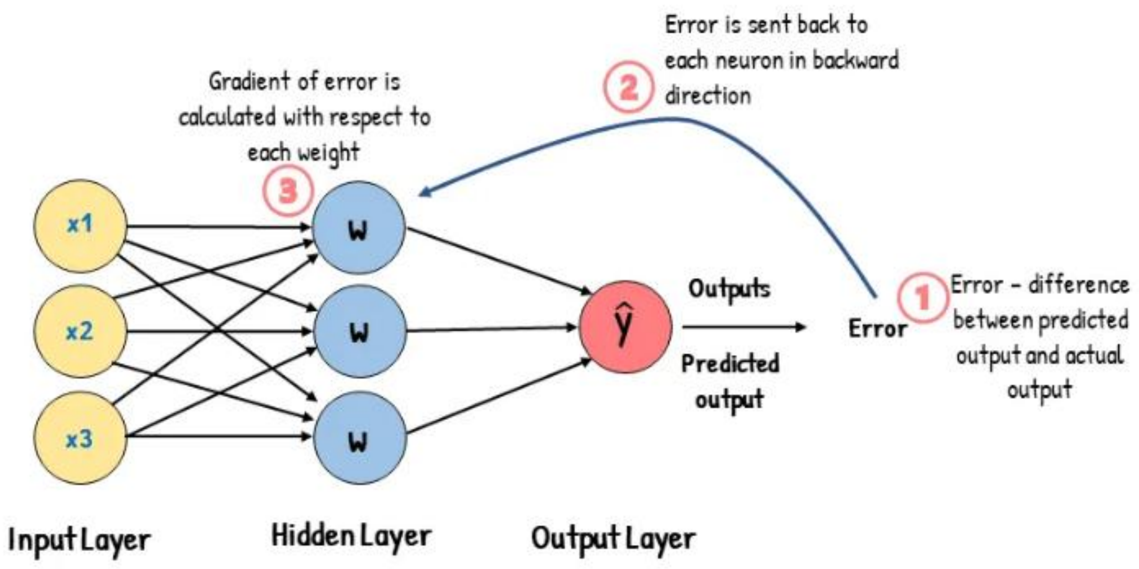


Forward propagation gives us predictions. But what if they're wrong? How do we know which weight caused the mistake, and by how much?



Backpropagation answers this question by sending the error backward, layer by layer, using the chain rule of calculus. In simple terms, backpropagation, or backward propagation of errors, is the key algorithm that enables neural networks to learn. It calculates how much each weight contributed to the overall error and updates them accordingly so that the network continuously improves its accuracy by reducing mistakes step by step**.**

- **Definition**: Backpropagation = algorithm to compute gradients of the loss w.r.t. weights using the chain rule.
- **Purpose**: Update weights to reduce error.
- **Steps**:
    1. Forward Pass → compute output & loss.
    2. Backward Pass → compute gradients (error flows backward).
    3. Update Weights → Gradient Descent.
- **Key Concept**: Repeated application of **Chain Rule**.


Error is sent back to each neuron in backward direction
2
Gradient of error is calculated with respect to each weight
3
x1
x2
x3
W
W
W
ŷ
Outputs
Predicted output
Error
1
Error - difference between predicted output and actual output
Input Layer
Hidden Layer
Output Layer

3) How do Gradients Work in Backpropagation?

Backpropagation: Using the chain rule from calculus, we calculate the gradients of the loss with respect to the network's parameters (weights and biases) by moving backward through the network, layer by layer.

The key here is that gradients for one layer depend on the gradients from the next layer, which is where the chain rule comes in.

Gradient Descent Algorithm

Once gradients are computed, they are used in gradient descent to update the parameters and reduce the loss. This process works as follows:

1. Initialize: Start with random weights and biases.
2. Compute Gradients: For each parameter (weight, bias), compute how much the loss changes when that parameter changes (this is the gradient).
3. Update Parameters: Update each parameter by moving in the direction of the negative gradient.

where θ is the parameter (weights or biases), $\partial L/\partial\theta$ is the gradient of the loss with respect to θ and η is the learning rate (step size).

4) Derivatives and Gradients

Derivatives

The **derivative** of a function measures **how much the output changes** with a small change in the input.

$$\frac{\partial L}{\partial w}$$

Derivative (with respect to one weight): It Measures how the loss changes with a small change in weight

Gradients

A **gradient** is the **vector of all partial derivatives** of a function with respect to its inputs.
It shows how the function changes in **multiple directions**.

$$\nabla_w L = \frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \ldots, \frac{\partial L}{\partial w_n}$$

Gradient (vector of all derivatives): It tells how to adjust **all weights** to reduce loss

What is the difference between derivatives and gradients, and how are they used in deep learning?

1. **Difference:**
   - A **derivative** is the rate of change of a function with respect to a **single variable** (used in simple functions).
   - A **gradient** is a **vector of partial derivatives** with respect to **multiple variables** (used in multivariable functions like neural networks).
2. **Use in Deep Learning:**
   - Gradients are used in **backpropagation** to compute how much each weight contributes to the loss.
   - These gradients help in **updating the weights** using optimization algorithms like gradient descent to minimize the loss and improve model accuracy.

Q. How does the derivative of the activation function influence the overall gradient passed back to earlier layers?

A: Activation derivatives multiply into the chain rule during backpropagation. If they are small, they shrink gradients, weakening updates to earlier layers; if large, they preserve or amplify gradient flow.

Q. Why can very small derivatives in deep networks cause weights in early layers to update extremely slowly?

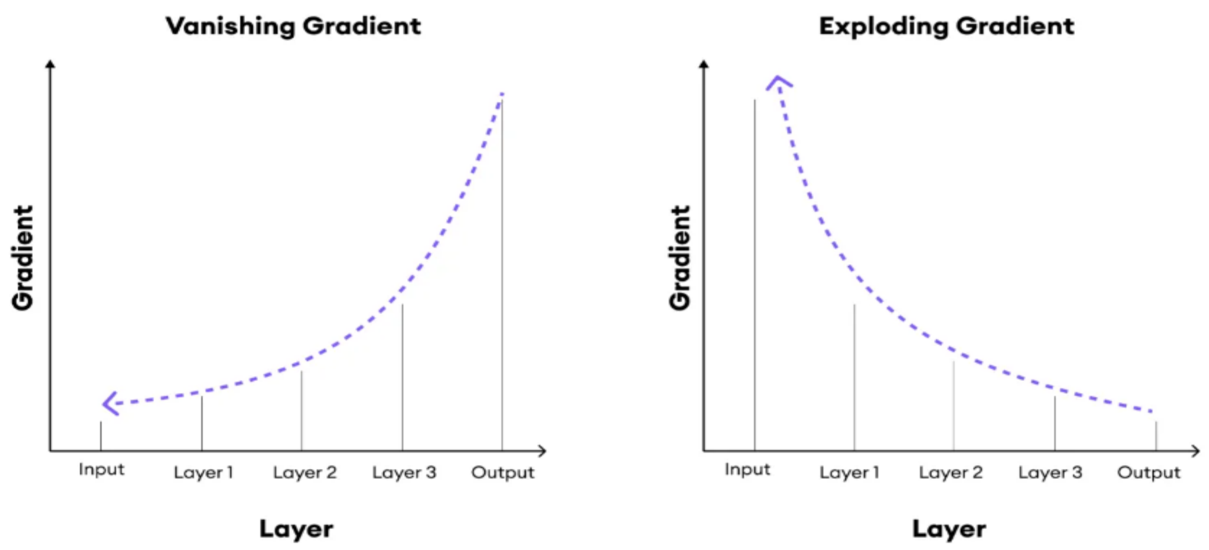
A: Small derivatives multiply repeatedly through layers, causing the **vanishing gradient problem**, where updates to early layers become negligible.

Q: How do derivatives ensure that gradient descent is moving toward a local minimum rather than away from it?

A: The derivative's sign tells us whether the slope is upward or downward; gradient descent always steps opposite to the slope, heading toward a minimum.

Challenges with Gradients

- Vanishing Gradients: In deep networks, the gradient can become very small in the early layers (close to zero), making learning slow or ineffective. This is called the vanishing gradient problem.
- Exploding Gradients: Gradients can also become excessively large, causing the model's parameters to diverge (get too large), which is the exploding gradient problem.



5) Common Properties of Derivatives

1. Constant Rule

$$\frac{d}{dx}(c) = 0$$

Derivative of a constant is always zero.

2. Power Rule

$$\frac{d}{dx}(x^n) = n \cdot x^{n-1}$$

Used very often when differentiating polynomials.

3. Constant Multiple Rule

$$\frac{d}{dx}[c \cdot f(x)] = c \cdot f'(x)$$

A constant can be factored out of the derivative.

4. Sum/Difference Rule

$$\frac{d}{dx}[f(x)+g(x)] = f'(x)+g'(x)$$

$$\frac{d}{dx}[f(x)-g(x)] = f'(x)-g'(x)$$

Derivative of a sum/difference = sum/difference of derivatives.

5. Product Rule

$$\frac{d}{dx}[f(x)g(x)] = f'(x)g(x) + f(x)g'(x)$$

6. Quotient Rule

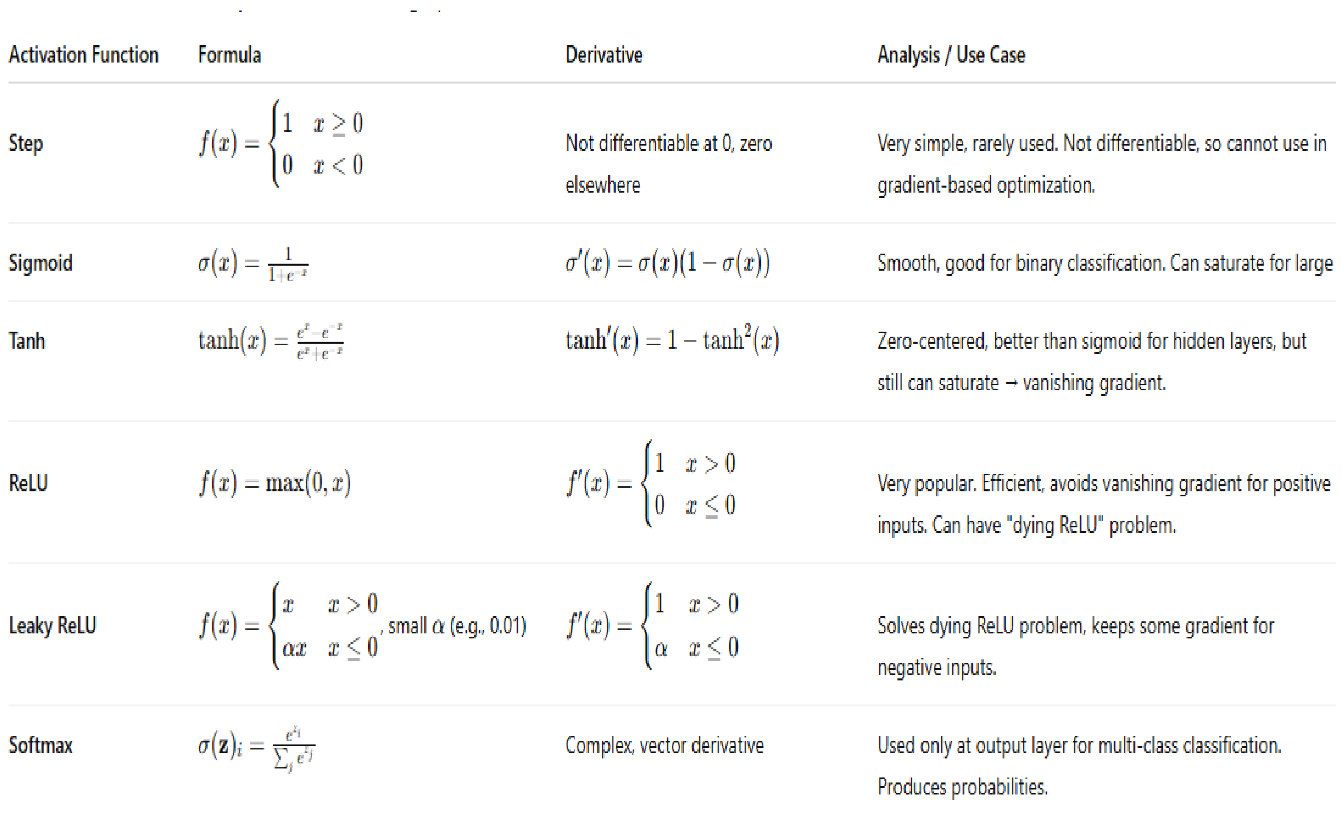
$$\frac{d}{dx}\left(\frac{f(x)}{g(x)}\right) = \frac{f'(x)g(x)-f(x)g'(x)}{[g(x)]^2}, \quad g(x)\neq 0$$

7. Chain Rule

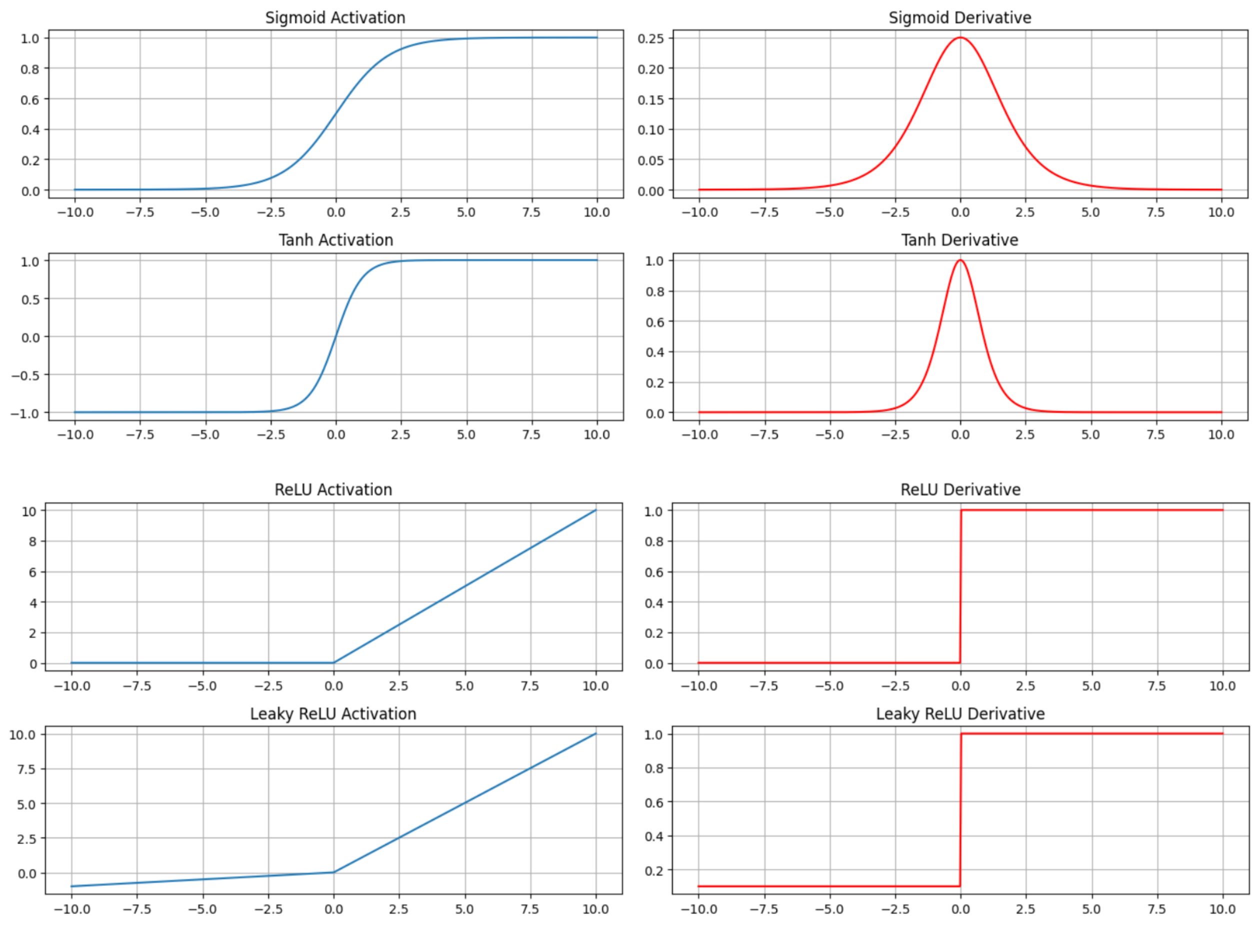
$$\frac{d}{dx}[f(g(x))] = f'(g(x))\cdot g'(x)$$

Most important rule for backpropagation (since neural networks are compositions of functions).

6) Derivative of Activation Function



Activation Function	Formula	Derivative	Analysis / Use Case
Step	$f(x)=\begin{cases}1 & x\geq 0\\ 0 & x<0\end{cases}$	Not differentiable at 0, zero elsewhere	Very simple, rarely used. Not differentiable, so cannot use in gradient-based optimization.
Sigmoid	$\sigma(x)=\frac{1}{1+e^{-x}}$	$\sigma'(x)=\sigma(x)(1-\sigma(x))$	Smooth, good for binary classification. Can saturate for large
Tanh	$\tanh(x)=\frac{e^x-e^{-x}}{e^x+e^{-x}}$	$\tanh'(x)=1-\tanh^2(x)$	Zero-centered, better than sigmoid for hidden layers, but still can saturate → vanishing gradient.
ReLU	$f(x)=\max(0,x)$	$f'(x)=\begin{cases}1 & x>0\\ 0 & x\leq 0\end{cases}$	Very popular. Efficient, avoids vanishing gradient for positive inputs. Can have "dying ReLU" problem.
Leaky ReLU	$f(x)=\begin{cases}x & x>0\\ \alpha x & x\leq 0\end{cases}$, small α (e.g., 0.01)	$f'(x)=\begin{cases}1 & x>0\\ \alpha & x\leq 0\end{cases}$	Solves dying ReLU problem, keeps some gradient for negative inputs.
Softmax	$\sigma(\mathbf{z})_i=\frac{e^{z_i}}{\sum_j e^{z_j}}$	Complex, vector derivative	Used only at output layer for multi-class classification. Produces probabilities.



Q. Why does sigmoid cause vanishing gradients?

A: For large |x|, σ(x) saturates near 0 or 1, making σ'(x) close to 0. This drastically reduces the magnitude of gradients during backpropagation, causing the **vanishing gradient problem**, where earlier layers in the network receive extremely small updates and learn very slowly or not at all.

Q: Why is ReLU considered better than sigmoid in many deep learning models?

A: ReLU avoids the severe vanishing gradient problem of sigmoid because for positive inputs its derivative is 1, allowing stronger gradient flow and faster training. It is also computationally simpler, requiring only a threshold check instead of exponentials.

Q: Why is Leaky ReLU considered better than ReLU in some cases?

A: Leaky ReLU addresses the “dead neuron” problem of ReLU by having a small non-zero slope (α) for negative inputs, ensuring gradients still flow even when inputs are negative, which keeps more neurons active during training.

Q: You are building a model to classify emails into "spam" or "not spam," and another model to classify images into one of ten animal categories. Which activation function would you use in each case, sigmoid or softmax, and why?

A. For the **spam vs not spam** model, use **sigmoid** because it's a binary classification problem and we only need a single probability output between 0 and 1.

For the **ten animal categories** model, use **softmax** because it's a multi-class classification problem with mutually exclusive classes, and we need a probability distribution that sums to 1 across all categories.

7) Gradient of Mean Square Error Function(Loss Function)

Mean Squared Error (MSE) Loss Function

For a dataset with n samples:

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

- yi= true value (target)
- y^i= predicted value (output of your model)

It measures how far off our predictions are from the actual values. Squaring emphasizes larger errors.

Gradient of MSE

The gradient is the derivative of the loss with respect to the model's predictions (y^)

For one sample:

Derivative w.r.t. y^i

$$\frac{\partial L_i}{\partial \hat{y}_i} = 2(\hat{y}_i - y_i)$$

For all n samples (average):

$$\frac{\partial L}{\partial \hat{y}_i} = \frac{2}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)$$

This is the gradient of the MSE loss.

- If $\hat{y}_i > y_i$, the gradient is **positive** → we should **decrease** $\hat{y}_i$.
- If $\hat{y}_i < y_i$, the gradient is **negative** → we should **increase** $\hat{y}_i$.
- The factor 2 just comes from the derivative of a square; it **scales the step size**.

Usage in Gradient Descent

In gradient descent, we update model parameters θ\thetaθ as:

$$\theta \leftarrow \theta - \eta \frac{\partial L}{\partial \theta}$$

Where $\frac{\partial L}{\partial \theta}$ is computed using the chain rule:

$$\frac{\partial L}{\partial \theta} = \frac{\partial L}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial \theta}$$

First part:

$$\frac{\partial L}{\partial \hat{y}_i} = 2(\hat{y}_i - y_i)$$

Second part: derivative of the model output w.r.t. parameters.

8) Gradient of Fully Connected Layer

Forward Pass

```
Input x ∈ ℝⁿ
         |
         ▼
    Fully Connected Layer
        y = W x + b
         |
         ▼
Activation a = f(y)
         |
         ▼
      Output
```

Backward Pass (Gradients)

1. Compute error term from next layer (or loss):

$$\delta = \frac{\partial L}{\partial a} \odot f'(y)$$

2. Gradients:

- w.r.t weights:

$$\frac{\partial L}{\partial W} = \delta x^T$$

- w.r.t bias:

$$\frac{\partial L}{\partial b} = \delta$$

- w.r.t input:

$$\frac{\partial L}{\partial x} = W^T \delta$$

Forward: Input x → linear transformation y=Wx+b → activation a=f(y)

Backward: Start from loss → compute gradient of activation → propagate to weights, bias, and input.

This is **the basic building block of backpropagation in dense layers**.

Q: Why do we need to multiply the gradient from the next layer by the derivative of the activation function during backpropagation?

A: Because the activation function determines how sensitive the output is to changes in the input. Multiplying by its derivative ensures the gradient correctly reflects this sensitivity. Without it, the network wouldn't learn how the non-linearity affects the loss.

Q: Why can gradients become very small in deep fully connected networks?

A: In deep networks, gradients are repeatedly multiplied by weight matrices and activation derivatives. If these derivatives are small (e.g., sigmoid saturates), the gradient shrinks at each layer, leading to **vanishing gradients** and slow learning.

Q: Why do we update weights in proportion to both the input x and the error δ?

A: Because the input determines how much each weight contributed to the output. Multiplying by δ ensures weights that caused larger errors get updated more. It's the network's way of "blaming" weights proportional to their influence.

Q: Why does the bias gradient not depend on the input?

A: Because the bias affects the output directly and uniformly. Its gradient only depends on the error signal δ, not on the input, as it shifts the activation regardless of x.

Q: How does the choice of activation function affect the gradient flow in a fully connected layer?

A: Activation functions control the slope of the output. Functions like ReLU keep gradients for positive inputs large, helping learning, while functions like sigmoid or tanh can squash gradients near 0 or 1, causing slow learning.

9) Chain rule

The **chain rule** is a fundamental calculus concept used in backpropagation. It tells us how to compute the derivative of a **composite function**.

If we have a function:

$$L = f(g(x))$$

Then the derivative of LLL with respect to x is:

$$\frac{dL}{dx} = \frac{dL}{dg} \cdot \frac{dg}{dx}$$

\

In neural networks:

- Each layer can be seen as a function: $a^{[l]} = f(W^{[l]}a^{[l-1]} + b^{[l]})$
- The **loss L** depends on the output. To find gradients w.r.t earlier layers, we **multiply gradients layer by layer** using the chain rule.
- **Intuition:** The chain rule lets us see how a small change in an early layer affects the final loss by "propagating" changes through all subsequent functions.

Q: Why do we multiply gradients of successive layers instead of just taking the gradient at the final layer?

A: Because each layer transforms the input. The chain rule ensures we account for **how each layer's output influences the next layer**, so the gradient at an early layer reflects its true impact on the final loss.

Q: How does the chain rule explain the vanishing gradient problem?

A: The chain rule multiplies derivatives of each layer. If these derivatives are small (like sigmoid near 0 or 1), the product becomes tiny as it propagates backward, causing gradients to **vanish** and slowing learning in earlier layers.

Q: Why does gradient descent rely on the chain rule, and how do activation derivatives fit into this chain?

A: The chain rule allows computing the loss derivative with respect to each weight by multiplying derivatives along the computation path, including the activation function's derivative, which controls gradient flow at that layer.

Q: If the derivative of the loss with respect to weights is always near zero, what does it imply about learning, and what techniques could fix it?

A: It implies the network is stuck due to vanishing gradients or reaching a flat region of the loss surface. Solutions include using ReLU/Leaky ReLU, batch normalization, better initialization, or residual connections.

10.Back Propagation Algorithm:

Backpropagation is the **core method to train neural networks**, used to compute gradients of the loss function with respect to all weights and biases.

It relies on the **chain rule** to propagate errors backward through the network.

Algorithm Steps

1. **Forward Pass**:
   - Inputs → Neurons → Outputs.
   - Compute activations and store them.
2. **Compute Loss**:

$$L = \frac{1}{2}(y_{true} - y_{pred})^2.$$

3. **Backward Pass (Gradient Computation)**:
   - Start at output: compute

$$\frac{\partial L}{\partial y_{pred}}.$$

   - Use chain rule to move backward:

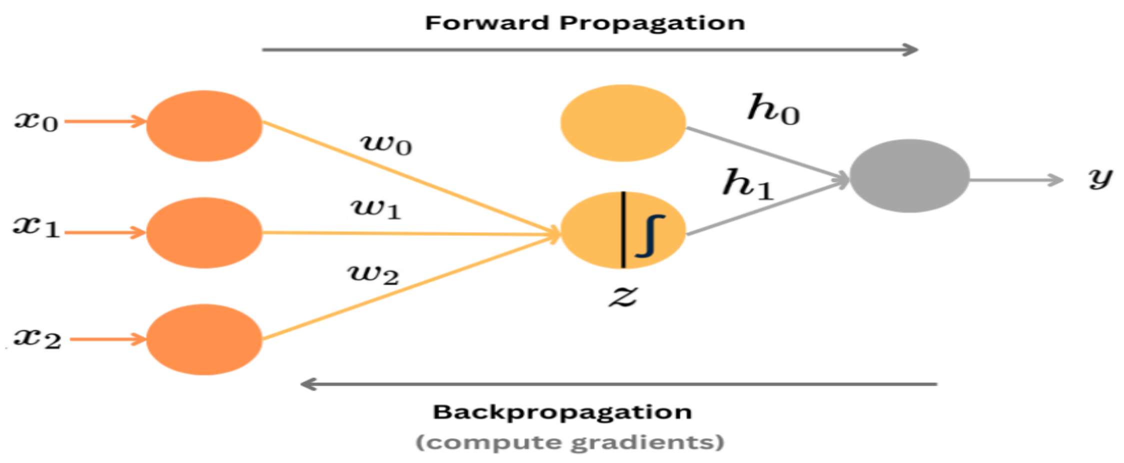
$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial w}$$

4. **Weight Update** (Gradient Descent):

$$w \leftarrow w - \eta \cdot \frac{\partial L}{\partial w}$$

5. **Repeat** steps for many epochs until loss reduces.

Example



forward propagation:

First, initialize the weights, then compute the outputs given the data samples

$$z = \sigma(x_0 w_0 + x_1 w_1 + x_2 w_2)$$

$$y = h_0 + h_1 z$$

Here:

- x_0, x_1, x_2 = inputs
- w_0, w_1, w_2 = weights
- σ = activation function
- z = hidden neuron output
- y = final prediction (linear combination with coefficients h_0, h_1)

Here, σ represents the activation functions that introduce non-linearity to the model. They determine whether to activate a neuron based on the weighted sum of its inputs.

Back Propagation

Then, compute the error and partial derivatives with respect to all these weights w0, w1, w2.

To apply the chain rule, we need to find the gradient for error *E* w.r.t. z and gradients of *z* w.r.t. w0, w1, w2.

🟦 i. Gradient of Loss E with respect to output y:

$$\frac{\partial E}{\partial y}$$

This tells us: *"How much does the loss change when the output y changes?"*

🟦 ii. Gradient of output y with respect to input z:

Let the activation function output be:

$$y = \sigma(z)$$

Then,

$$\frac{\partial y}{\partial z} = h_1$$

Here $h_1 = \sigma'(z)$, i.e., the derivative of the activation function at z.

🟦 iii. Gradients of z with respect to each weight:

We have:

$$z = x_0w_0 + x_1w_1 + x_2w_2$$

Now differentiate z with respect to each w_i:

$$\frac{\partial z}{\partial w_0} = x_0 \cdot \sigma'(x_0w_0 + x_1w_1 + x_2w_2)$$
$$\frac{\partial z}{\partial w_1} = x_1 \cdot \sigma'(x_0w_0 + x_1w_1 + x_2w_2)$$
$$\frac{\partial z}{\partial w_2} = x_2 \cdot \sigma'(x_0w_0 + x_1w_1 + x_2w_2)$$

> 🔸 Here, σ' means the derivative of the activation function.

🟦 iv. Gradients of E with respect to w_0, w_1, w_2 using the Chain Rule:

By chain rule:

$$\frac{\partial E}{\partial w_0} = \frac{\partial E}{\partial y} \times \frac{\partial y}{\partial z} \times \frac{\partial z}{\partial w_0}$$
$$\frac{\partial E}{\partial w_1} = \frac{\partial E}{\partial y} \times \frac{\partial y}{\partial z} \times \frac{\partial z}{\partial w_1}$$
$$\frac{\partial E}{\partial w_2} = \frac{\partial E}{\partial y} \times \frac{\partial y}{\partial z} \times \frac{\partial z}{\partial w_2}$$