DA6401:Introduction to Deep Learning
Module 2A - BackPropagation in MLPs

Ganapathy Krishnamurthi

Department of Data Science and Artificial Intelligence , IIT Madras

Story so far - The Supervised Learning Setup

A typical supervised machine learning system consists of five key components:

1. **Data:** A collection of n labeled examples, denoted as $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{n}$.

Story so far - The Supervised Learning Setup

A typical supervised machine learning system consists of five key components:

1. **Data:** A collection of n labeled examples, denoted as $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{n}$.
   Typically comes from benchmarks and gold standard datasets.

Story so far - The Supervised Learning Setup

A typical supervised machine learning system consists of five key components:

1. **Data:** A collection of n labeled examples, denoted as $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{n}$.
   Typically comes from benchmarks and gold standard datasets.
2. **Model:** An approximation function that maps input x to output y.

Story so far - The Supervised Learning Setup

A typical supervised machine learning system consists of five key components:

1. **Data:** A collection of n labeled examples, denoted as $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{n}$.
   Typically comes from benchmarks and gold standard datasets.
2. **Model:** An approximation function that maps input x to output y.
   In our case, a Neural Network (i.e,. a Multi-Layer Perceptron) with some fixed hyperparameters(i.e,. number of layers, number of neurons, etc)

Story so far - The Supervised Learning Setup

A typical supervised machine learning system consists of five key components:

1. **Data:** A collection of n labeled examples, denoted as $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{n}$.
   Typically comes from benchmarks and gold standard datasets.
2. **Model:** An approximation function that maps input x to output y.
   In our case, a Neural Network (i.e,. a Multi-Layer Perceptron) with some fixed hyperparameters(i.e,. number of layers, number of neurons, etc)
3. **Parameters:** The unknown variables within the model that determine its behavior. These must be learned.

Story so far - The Supervised Learning Setup

A typical supervised machine learning system consists of five key components:

1. **Data:** A collection of n labeled examples, denoted as $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{n}$.
   Typically comes from benchmarks and gold standard datasets.
2. **Model:** An approximation function that maps input x to output y.
   In our case, a Neural Network (i.e,. a Multi-Layer Perceptron) with some fixed hyperparameters(i.e,. number of layers, number of neurons, etc)
3. **Parameters:** The unknown variables within the model that determine its behavior. These must be learned.
   The weights and biases of the above defined MLP.

Key Components to explore

4. **Objective / Loss Function:** A metric $J(\mathbf{w})$ that quantifies the difference between the model's prediction $\hat{y}$ and the true label y.

Key Components to explore

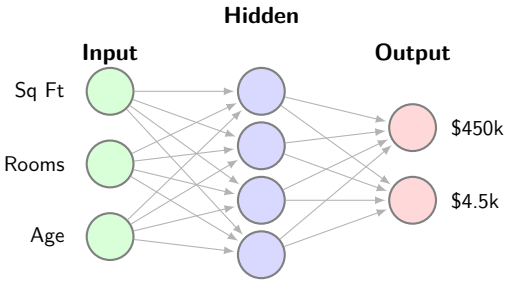
4. **Objective / Loss Function:** A metric $J(\mathbf{w})$ that quantifies the difference between the model's prediction $\hat{y}$ and the true label y.
5. **Learning Algorithm:** The optimization method used to adjust the parameters $\mathbf{w}$ to minimize the Loss Function.

Output Activations & Loss Functions

Recap: Regression vs. Classification

1. Regression

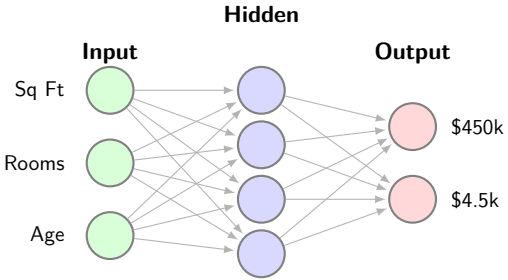
- **Goal:** Predict continuous quantities.
- **Example:** Predicting House Price & Tax.



Regression: Continuous Output

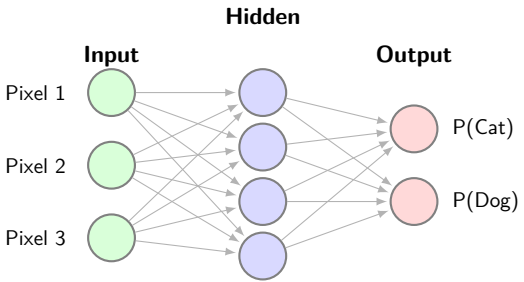
Recap: Regression vs. Classification

1. **Regression**
   - **Goal:** Predict continuous quantities.
   - **Example:** Predicting House Price & Tax.



Regression: Continuous Output

2. **Classification**
   - **Goal:** Predict class probabilities.
   - **Example:** Is the image a Cat or Dog?



Classification: Probabilities

Regression: Activations and Loss

Output Activation: Linear (Identity)

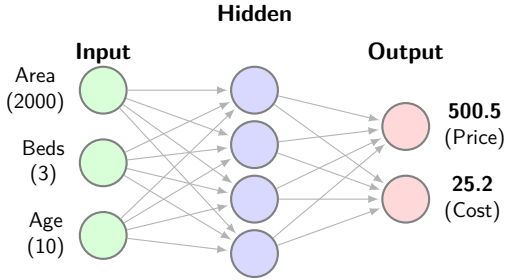
- $a(z) = z$ (Passes raw value through).

Loss Function 1: Mean Squared Error (MSE)
Penalizes large errors heavily.

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

Loss Function 2: Mean Absolute Error (MAE)
More robust to outliers.

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



Outputs are unbounded $\mathbb{R}$

Classification: Multi-Class vs. Multi-Label

Scenario 1: Multi-Class

- **Constraint:** Mutually Exclusive (One-Hot).
- *Example:* Digit is '3'. It cannot be '5'.

Classification: Multi-Class vs. Multi-Label

Scenario 1: Multi-Class

- **Constraint:** Mutually Exclusive (One-Hot).
- *Example:* Digit is '3'. It cannot be '5'.

Scenario 2: Multi-Label

- **Constraint:** Independent / Non-exclusive.
- *Example:* Movie is 'Action' AND 'Sci-Fi'.

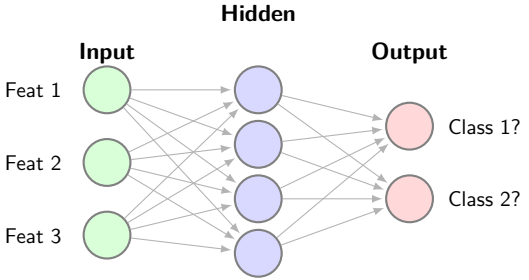
Classification: Multi-Class vs. Multi-Label

Scenario 1: Multi-Class

- **Constraint:** Mutually Exclusive (One-Hot).
- *Example:* Digit is '3'. It cannot be '5'.

Scenario 2: Multi-Label

- **Constraint:** Independent / Non-exclusive.
- *Example:* Movie is 'Action' AND 'Sci-Fi'.



Same architecture, Different logic.

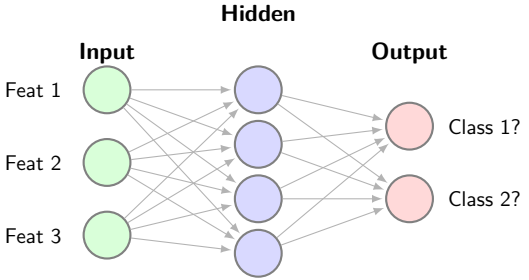
Classification: Multi-Class vs. Multi-Label

Scenario 1: Multi-Class

- **Constraint:** Mutually Exclusive (One-Hot).
- *Example:* Digit is '3'. It cannot be '5'.

Scenario 2: Multi-Label

- **Constraint:** Independent / Non-exclusive.
- *Example:* Movie is 'Action' AND 'Sci-Fi'.

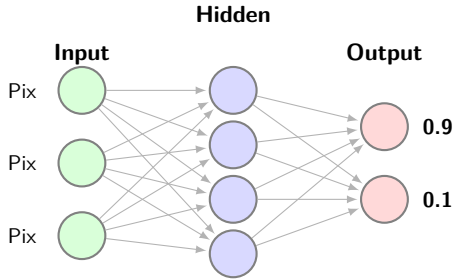


Same architecture, Different logic.

Question: *In which scenario should the output probabilities sum to 1?*

Solutions: Activations

Multi-Class (Mutually Exclusive)

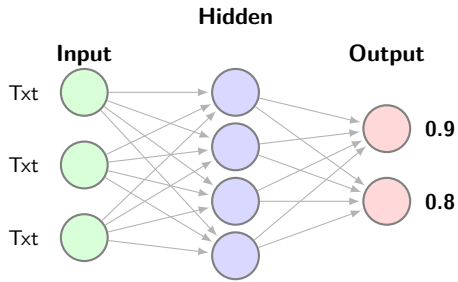


Softmax: Sums to 1

Activation: Softmax

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

Multi-Label (Independent)



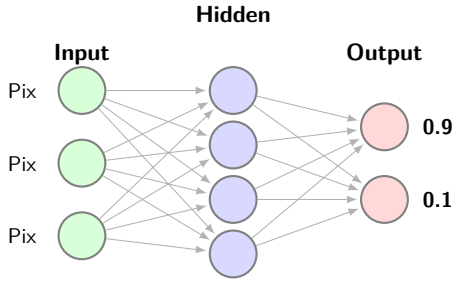
Sigmoid: Independent

Activation: Sigmoid

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

Solutions: Loss Function

Multi-Class (Mutually Exclusive)



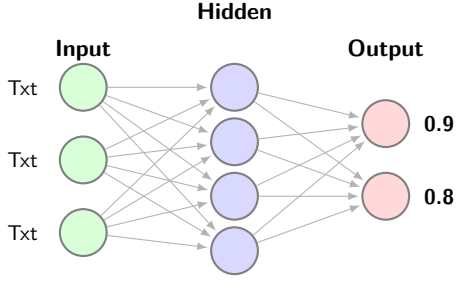
Softmax: Sums to 1

Loss: Categorical Cross-Entropy

$$\mathcal{L} = -\sum_{c=1}^{C} y_c \log(\hat{y}_c)$$

(where $y_c = 1$ for the true class)

Multi-Label (Independent)



Sigmoid: Independent

Loss: Binary Cross-Entropy

$$\mathcal{L} = -\frac{1}{M}\sum_{j=1}^{M}[y_j \log(\hat{y}_j) + (1 - y_j)\log(1 - \hat{y}_j)]$$

(Summed over all output nodes)

Introduction to Backpropagation

What is Backpropagation?

- **Definition:** Backpropagation ("backward propagation of errors") is an algorithm for computing the gradient of the loss function with respect to the weights.

What is Backpropagation?

- **Definition:** Backpropagation ("backward propagation of errors") is an algorithm for computing the gradient of the loss function with respect to the weights.
- **Core Mechanism:** It applies the **Chain Rule** of calculus recursively from the output layer back to the input layer.

What is Backpropagation?

- **Definition:** Backpropagation ("backward propagation of errors") is an algorithm for computing the gradient of the loss function with respect to the weights.
- **Core Mechanism:** It applies the **Chain Rule** of calculus recursively from the output layer back to the input layer.
- **Purpose:** It tells us how much each weight contributed to the error, allowing us to adjust them to reduce error.

Use of Gradients

The Goal: Minimize Loss $\mathcal{L}$

Use of Gradients

The Goal: Minimize Loss $\mathcal{L}$

- We view the Loss Surface as a landscape.

Use of Gradients

The Goal: Minimize Loss $\mathcal{L}$

- We view the Loss Surface as a landscape.
- The **Gradient** ($\nabla_\theta \mathcal{L}$) points in the direction of steepest ascent (up the hill).

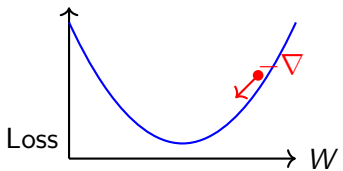
Use of Gradients

The Goal: Minimize Loss $\mathcal{L}$

- We view the Loss Surface as a landscape.
- The **Gradient** ($\nabla_\theta \mathcal{L}$) points in the direction of steepest ascent (up the hill).
- To minimize error, we want to move in the direction of **Steepest Descent**.

$$\Delta W \propto -\frac{\partial \mathcal{L}}{\partial W}$$

Backprop provides this gradient vector.



Objective of Backpropagation

Mathematically, for every weight $w_{jk}^{(l)}$ and bias $b_k^{(l)}$ in the network, we need to compute:

$$\frac{\partial \mathcal{L}}{\partial w_{jk}^{(l)}} \quad \text{and} \quad \frac{\partial \mathcal{L}}{\partial b_k^{(l)}}$$

Where:

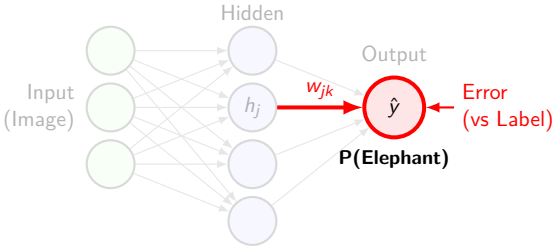
- $\mathcal{L}$: The Loss function (e.g., MSE, Cross-Entropy).
- $w_{jk}^{(l)}$: Weight connecting neuron j in layer $(l-1)$ to neuron k in layer l.
- $b_k^{(l)}$: Bias associated with neuron k in layer l.

The Setup: Binary Classification

We want to train an MLP to predict the presence or absence of "Elephant" from an input image.

Definitions:

- y: True Label (1 = Elephant, 0 = Not).
- $\hat{y}$: Prediction ($0.0 \to 1.0$).
- z: Weighted sum into the output node.



Chain Rule:

$$\frac{\partial \mathcal{L}}{\partial w_{jk}} = \underbrace{\frac{\partial \mathcal{L}}{\partial \hat{y}}}_{\text{Step 1}} \cdot \underbrace{\frac{\partial \hat{y}}{\partial z}}_{\text{Step 2}} \cdot \underbrace{\frac{\partial z}{\partial w_{jk}}}_{\text{Step 3}}$$

Step 1: Derivative of Loss (BCE)

Loss Function: Binary Cross-Entropy

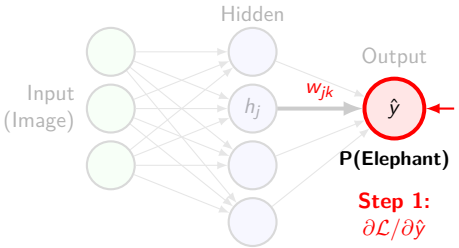
$$\mathcal{L} = -[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})]$$

The Derivative: How does error change as prediction $\hat{y}$ changes?

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = -\frac{y}{\hat{y}} + \frac{1 - y}{1 - \hat{y}}$$

If we simplify this fraction:

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = \frac{\hat{y} - y}{\hat{y}(1 - \hat{y})}$$



Step 2: Derivative of Activation (Sigmoid)

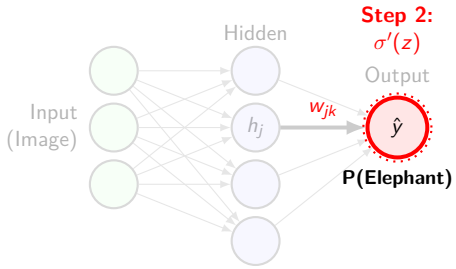
Activation Function: Sigmoid We squash the sum z to a probability:

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

The Derivative: The slope of the sigmoid curve at z.

$$\frac{\partial \hat{y}}{\partial z} = \sigma'(z) = \hat{y}(1 - \hat{y})$$

Note: This matches the denominator from Step 1!



The "Magic" Simplification

Before Step 3, let's combine Step 1 and Step 2 to find the local error δ.

$$\delta = \text{Step 1} \times \text{Step 2}$$

$$\delta = \left(\frac{\hat{y} - y}{\hat{y}(1 - \hat{y})} \right) \cdot (\hat{y}(1 - \hat{y}))$$

$$\delta = \hat{y} - y$$

Result: The local error is just the difference between prediction and truth!

Step 3: Derivative of Weights

Question: How much of the error is due to weight w_{jk}?

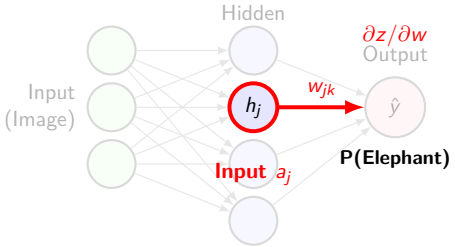
Recall forward pass:

$$z = \cdots + w_{jk} \cdot a_j + \ldots$$

Derivative:

$$\frac{\partial z}{\partial w_{jk}} = a_j$$

Where a_j is the activation of the hidden neuron detecting "Elephant Features" (e.g., trunk shape).



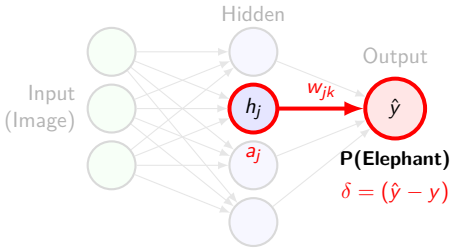
Final Gradient Calculation

Combining all parts for the weight update:

$$\frac{\partial \mathcal{L}}{\partial w_{jk}} = \underbrace{(\hat{y} - y)}_{\delta} \cdot \underbrace{a_j}_{\text{Input}}$$

Interpretation:

- If Prediction $\approx$ Truth, gradient is 0 (Stop learning).
- If Input a_j is high (feature present) AND error is high, update weight significantly.



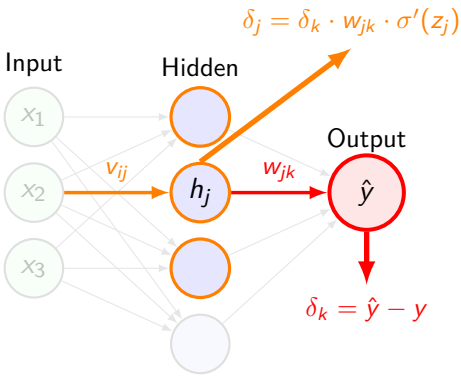
Extending to Deeper Layers: The Challenge

What we've learned so far:

- We know how to compute gradients for weights connecting the hidden layer to the output layer: w_{jk}
- Error at output: $\delta_k = \hat{y} - y$

The Question: How do we update weights v_{ij} connecting the input layer to the hidden layer?

Hint: We need to propagate the error backward through the network!



Step 1: Error at the Output Layer (Review)

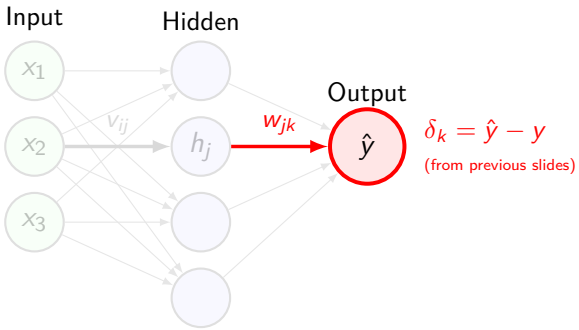
Output Layer Error: We already computed this!

$$\delta_k = \frac{\partial \mathcal{L}}{\partial z_k} = \hat{y} - y$$

where z_k is the weighted sum into the output neuron.

This is our starting point for backpropagation.

Next: Propagate this error to the hidden layer.



Step 2: Error at the Hidden Layer

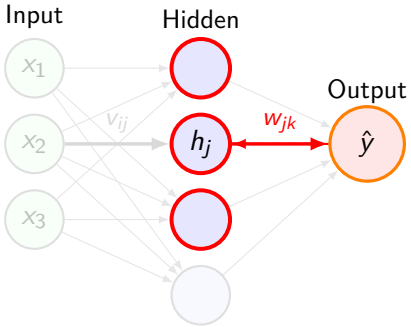
Chain Rule to Hidden Neuron j:

$$\delta_j = \frac{\partial \mathcal{L}}{\partial z_j} = \frac{\partial \mathcal{L}}{\partial z_k} \cdot \frac{\partial z_k}{\partial a_j} \cdot \frac{\partial a_j}{\partial z_j}$$

Breaking it down:

- $\frac{\partial \mathcal{L}}{\partial z_k} = \delta_k$ (from output)
- $\frac{\partial z_k}{\partial a_j} = w_{jk}$ (weight connecting j to k)
- $\frac{\partial a_j}{\partial z_j} = \sigma'(z_j)$ (activation derivative)

$$\delta_j = \delta_k \cdot w_{jk} \cdot \sigma'(z_j)$$



Propagate error:
$\delta_j = \delta_k \cdot w_{jk} \cdot \sigma'(z_j)$

Error propagates through the weight w_{jk}

Step 3: Updating Input-to-Hidden Weights

Gradient for weight v_{ij}:
Now that we have δ_j for the hidden layer, we can compute:

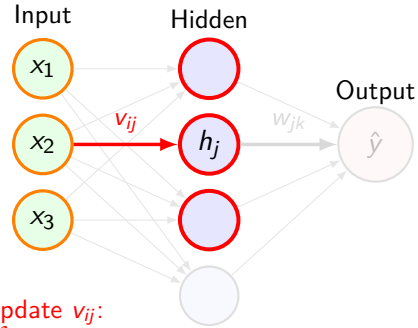
$$\frac{\partial \mathcal{L}}{\partial v_{ij}} = \delta_j \cdot x_i$$

where:

- δ_j = error at hidden neuron j
- x_i = input from pixel/feature i

Pattern Recognition: This has the exact same form as before!

$$\frac{\partial \mathcal{L}}{\partial \text{weight}} = \text{error} \times \text{input}$$



Vectorization: Why the Outer Product?

From Scalar to Matrix: In a fully connected layer, we don't update weights one by one. We update the entire weight matrix $W^{(\ell)}$ at once.

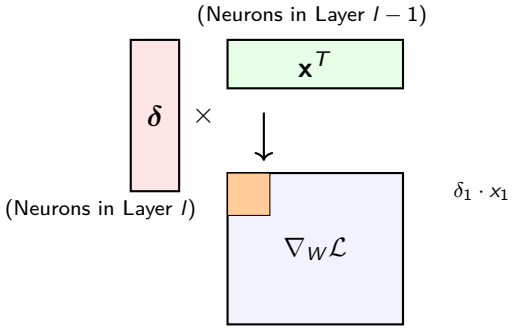
Individual Weight Gradient:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \delta_j \cdot x_i$$

Matrix Form: To get the gradient for the whole matrix, we take the **Outer Product** of the error vector $\boldsymbol{\delta}$ and the input vector $\mathbf{x}^T$:

$$\nabla_W \mathcal{L} = \boldsymbol{\delta} \mathbf{x}^T$$

Visualization of $\delta \mathbf{x}^T$



Vectorization: Why the Outer Product?

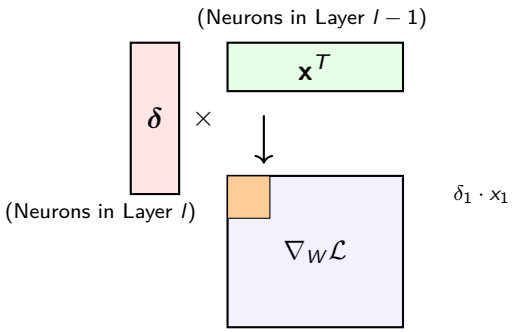
Individual Weight Gradient:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \delta_j \cdot x_i$$

Matrix Form: To get the gradient for the whole matrix, we take the **Outer Product** of the error vector $\boldsymbol{\delta}$ and the input vector $\mathbf{x}^T$:

$$\nabla_W \mathcal{L} = \boldsymbol{\delta} \mathbf{x}^T$$

Visualization of $\delta \mathbf{x}^T$



The Derivation: Since $z_i = \sum_j w_{ij} a_j$, the partial derivative $\frac{\partial z_i}{\partial w_{ij}}$ is only non-zero (a_j) when the indices match. This maps every combination of error i and input j into its corresponding slot in the weight matrix.

The General Backpropagation Recipe

For a network with L layers, we backpropagate errors from layer L to layer 1:

1. Output Layer (Layer L):

$$\delta^{(L)} = \hat{y} - y$$

2. Hidden Layers ($\ell = L-1, L-2, \ldots, 2$):

$$\delta^{(\ell)} = \left(W^{(\ell+1)}\right)^T \delta^{(\ell+1)} \odot \sigma'\left(z^{(\ell)}\right)$$

(where $\odot$ is element-wise multiplication)

3. Weight Updates (all layers):

$$\frac{\partial \mathcal{L}}{\partial W^{(\ell)}} = \delta^{(\ell)} \left(a^{(\ell-1)}\right)^T$$

The error "flows backward" through the weights, scaled by activation derivatives.

Key Insights: Why Backpropagation Works

1. **Chain Rule is Everything:**
   We decompose the complex derivative into simple, local computations at each layer.

2. **Reuse Computations:**
   Each δ is computed once and reused for all weights in that layer. This makes backprop efficient!

3. **Error Attribution:**
   Weights that contributed more to the error (high $\delta \cdot$ input) get larger updates.

4. **Vanishing/Exploding Gradients:**
   As we go deeper, δ can become very small (vanishing) or very large (exploding) due to repeated multiplication by $w \cdot \sigma'(z)$. This is why deep networks can be hard to train!

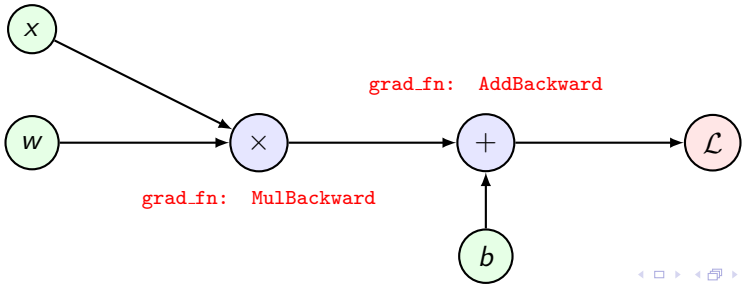
Automatic Differentiation in Frameworks

The Mechanics of Autograd

To compute gradients, PyTorch builds a **Dynamic Directed Acyclic Graph (DAG)**.

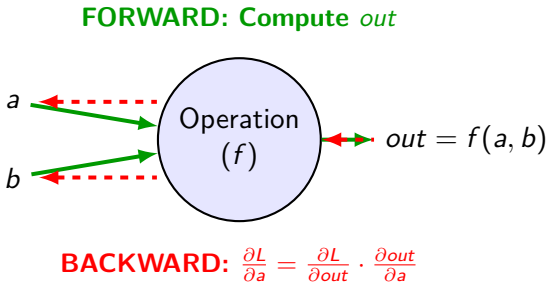
- **Tensors (Nodes):** Store the data and the accumulated gradient (`.grad`).
- **Function Objects (Edges):** Every operation (add, mul, exp) creates a `grad_fn` that knows its own derivative.
- **Leaf Tensors:** Tensors created by the user (weights w, bias b, input x).

Rule: Gradient at Node = Incoming Gradient $\times$ Local Derivative



The Anatomy of a Graph Node

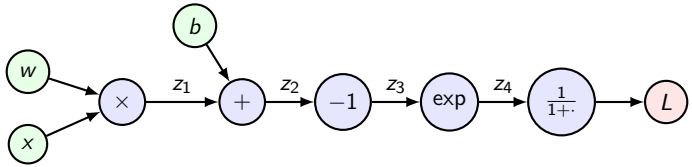
In PyTorch, every operation (e.g., $+, \times, \exp$) is a node that implements a `forward()` and a `backward()` method.



A 5-Step Example: $L = \sigma(w \cdot x + b)$

Let's decompose the operation $L = \frac{1}{1+e^{-(wx+b)}}$ into atomic steps:

1. $z_1 = w \times x$ — *Mul*
2. $z_2 = z_1 + b$ — *Add*
3. $z_3 = -z_2$ — *Neg*
4. $z_4 = \exp(z_3)$ — *Exp*
5. $L = \frac{1}{1+z_4}$ — *Inv + Add*

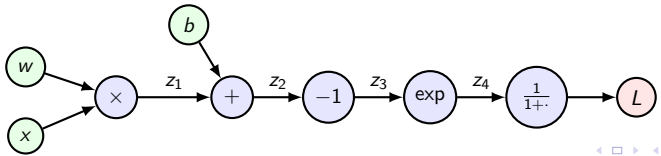


Step-by-Step Backpropagation Trace

Starting with $\frac{\partial L}{\partial L} = 1$, we move backwards node-by-node:

Node	Local Grad ($\frac{\partial out}{\partial in}$)	Chain Rule Expression	Result
$L = \frac{1}{1+z_4}$	$\frac{\partial L}{\partial z_4} = -\frac{1}{(1+z_4)^2}$	$\frac{\partial L}{\partial L} \cdot \frac{\partial L}{\partial z_4} = 1 \cdot (-L^2)$	$\frac{\partial L}{\partial z_4} = -L^2$
$z_4 = \exp(z_3)$	$\frac{\partial z_4}{\partial z_3} = \exp(z_3)$	$\frac{\partial L}{\partial z_4} \cdot \frac{\partial z_4}{\partial z_3} = (-L^2) \cdot z_4$	$\frac{\partial L}{\partial z_3} = -L^2 z_4$
$z_3 = -z_2$	$\frac{\partial z_3}{\partial z_2} = -1$	$\frac{\partial L}{\partial z_3} \cdot \frac{\partial z_3}{\partial z_2} = (-L^2 z_4) \cdot (-1)$	$\frac{\partial L}{\partial z_2} = L^2 z_4$
$z_2 = z_1 + b$	$\frac{\partial z_2}{\partial b} = 1$	$\frac{\partial L}{\partial z_2} \cdot \frac{\partial z_2}{\partial b} = (L^2 z_4) \cdot 1$	$\frac{\partial L}{\partial b} = L^2 z_4$
$z_1 = w \cdot x$	$\frac{\partial z_1}{\partial w} = x$	$\frac{\partial L}{\partial z_1} \cdot \frac{\partial z_1}{\partial w} = (L^2 z_4) \cdot x$	$\frac{\partial L}{\partial w} = L^2 z_4 x$

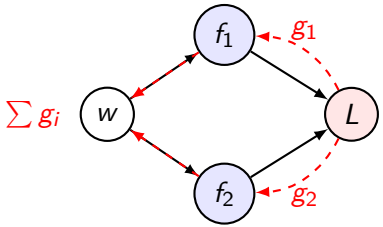
Key Point: The weight gradient $\frac{\partial L}{\partial w}$ is the product of all local derivatives encountered on the path from L back to w.



Why the Graph? Gradient Accumulation

If a variable w affects L through **two different paths**, the gradients are **summed** at the junction.

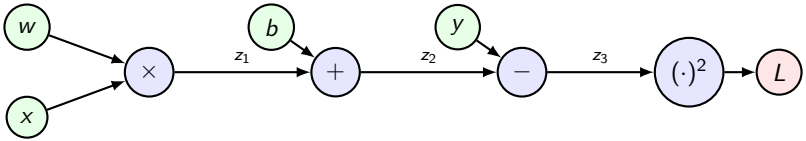
- Path 1 gives $\frac{\partial L}{\partial p_1}$
- Path 2 gives $\frac{\partial L}{\partial p_2}$
- **Total:** $\frac{\partial L}{\partial w} = \frac{\partial L}{\partial p_1} + \frac{\partial L}{\partial p_2}$



Step-by-Step Example: $L = (w \cdot x + b - y)^2$

Let's trace a forward pass with 5 operations:

1. $z_1 = w \cdot x$ (Multiplication)
2. $z_2 = z_1 + b$ (Addition)
3. $z_3 = z_2 - y$ (Subtraction)
4. $L = z_3^2$ (Power)



The Backward Pass: Message Passing

When we call `L.backward()`, the gradient $g = \frac{\partial L}{\partial L} = 1$ flows backward:

- **At Power Node (z_3^2):** Local $\frac{\partial L}{\partial z_3} = 2z_3$. Grad sent back: $1 \cdot 2z_3$.
- **At Subtraction Node ($z_2 - y$):** Local $\frac{\partial z_3}{\partial z_2} = 1$. Grad sent back: $(2z_3) \cdot 1$.
- **At Addition Node ($z_1 + b$):** Local $\frac{\partial z_2}{\partial z_1} = 1$ and $\frac{\partial z_2}{\partial b} = 1$.
- **At Multiplication Node ($w \cdot x$):** Local $\frac{\partial z_1}{\partial w} = x$. Final Grad: $(2z_3 \cdot 1 \cdot 1) \cdot x$.

Key Features of the Graph Approach

Efficiency:

- Only local derivatives are calculated at each node.
- Intermediate results (z_1, z_2, z_3) are cached during forward pass.

Gradient Accumulation:

- If a tensor is used in multiple operations, the gradients from different paths are **summed** at that node.
- This is how PyTorch handles branching.

The Forward and Backward Pass

1. **Forward Pass:**
   - Data flows from left to right.
   - Each operation saves **intermediate values** (tensors) needed for the gradient.
   - Each tensor points to a `grad_fn` (the recipe for its derivative).

The Forward and Backward Pass

1. **Forward Pass:**
   - Data flows from left to right.
   - Each operation saves **intermediate values** (tensors) needed for the gradient.
   - Each tensor points to a `grad_fn` (the recipe for its derivative).

2. **Backward Pass (`loss.backward()`):**
   - The graph is traversed **in reverse**.
   - Local gradients are computed using the saved values.
   - Gradients are multiplied by the incoming "gradient from above" (Chain Rule).

From Math to Code

The Graph Logic:

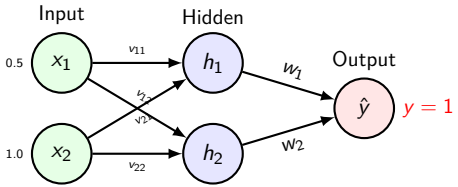
1. Define tensors with `requires_grad=True`.
2. Perform operations (PyTorch builds the graph).
3. Call `.backward()`.
4. Access `.grad` attribute.

```python
import torch

x = torch.tensor(1.0, requires_grad=True)
w = torch.tensor(2.0, requires_grad=True)
b = torch.tensor(0.5, requires_grad=True)

# Forward pass (Building Graph)
z = w * x + b
loss = z**2

# Backward pass (Traversing Graph)
loss.backward()

print(w.grad) # dLoss/dw
```

Illustration of Backpropagation: Setup

Tiny Network for Hand Calculation:

- 2 input features: $x_1 = 0.5$, $x_2 = 1.0$
- 2 hidden neurons
- 1 output neuron (binary classification)
- Sigmoid activation: $\sigma(z) = \frac{1}{1+e^{-z}}$
- True label: $y = 1$



Initial Weights:

$$v_{11} = 0.5, \quad v_{12} = 0.5$$
$$v_{21} = 0.5, \quad v_{22} = 0.5$$
$$w_1 = 1.0, \quad w_2 = 1.0$$

Step 1: Forward Pass - Hidden Layer

Compute Hidden Layer Activations:

Hidden Neuron 1:

$$z_1 = v_{11}x_1 + v_{21}x_2 = 0.5 \times 0.5 + 0.5 \times 1.0$$

$$z_1 = 0.25 + 0.5 = \boxed{0.75}$$

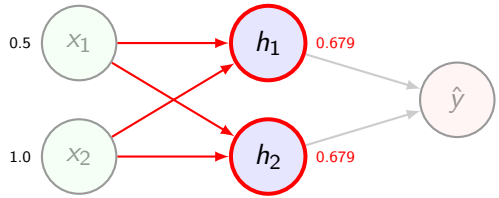
$$h_1 = \sigma(z_1) = \frac{1}{1 + e^{-0.75}} \approx \boxed{0.679}$$

Hidden Neuron 2:

$$z_2 = v_{12}x_1 + v_{22}x_2 = 0.5 \times 0.5 + 0.5 \times 1.0$$

$$z_2 = 0.25 + 0.5 = \boxed{0.75}$$

$$h_2 = \sigma(z_2) = \frac{1}{1 + e^{-0.75}} \approx \boxed{0.679}$$



Both hidden neurons have activation ≈ 0.679

Step 2: Forward Pass - Output Layer

Compute Output:

Weighted Sum:

$$z_{\text{out}} = w_1 h_1 + w_2 h_2 = 1.0 \times 0.679 + 1.0 \times 0.679$$

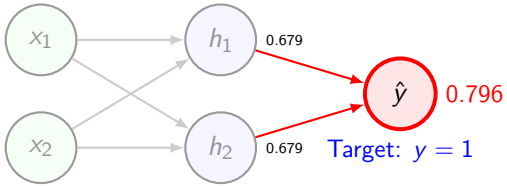
$$z_{\text{out}} = 0.679 + 0.679 = \boxed{1.358}$$

Final Prediction:

$$\hat{y} = \sigma(z_{\text{out}}) = \frac{1}{1 + e^{-1.358}} \approx \boxed{0.796}$$

Loss (Binary Cross-Entropy):

$$\begin{aligned} \mathcal{L} &= -[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})] \\ &= -[1 \times \log(0.796) + 0 \times \log(0.204)] \\ &= -\log(0.796) \approx \boxed{0.228} \end{aligned}$$



Prediction: 0.796
Error: $0.796 - 1 = -0.204$

Step 3: Backward Pass - Output Error

Compute Output Layer Error:

Recall the simplified formula for sigmoid + BCE:

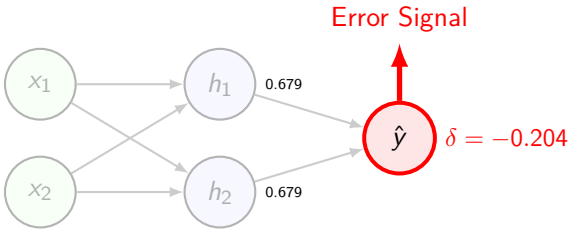
$$\delta_{\text{out}} = \hat{y} - y$$

Calculation:

$$\delta_{\text{out}} = 0.796 - 1 = \boxed{-0.204}$$

Negative error means we predicted too low (should increase output).

Next Step: Use this to compute gradients for w_1 and w_2.



Output error computed!
Now backpropagate...

Step 4: Gradients for Output Weights (w_1, w_2)

Gradient Formula: $\frac{\partial \mathcal{L}}{\partial w_i} = \delta_{\text{out}} \cdot h_i$

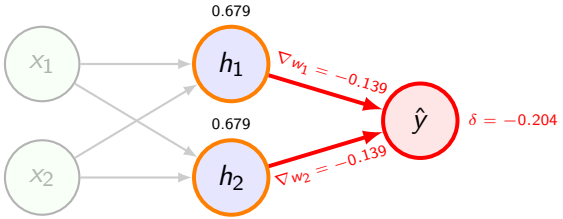
For w_1:

$$\frac{\partial \mathcal{L}}{\partial w_1} = \delta_{\text{out}} \cdot h_1 = (-0.204) \times 0.679$$
$$= \boxed{-0.139}$$

For w_2:

$$\frac{\partial \mathcal{L}}{\partial w_2} = \delta_{\text{out}} \cdot h_2 = (-0.204) \times 0.679$$
$$= \boxed{-0.139}$$

Negative gradients mean we should increase both weights!



Gradients computed!
Both weights need to increase

Step 5: Hidden Layer Errors (δ_1, δ_2)

Backpropagate Error to Hidden Layer:

Formula: $\delta_j = \delta_{\text{out}} \cdot w_j \cdot \sigma'(z_j)$

First, compute $\sigma'(z_j) = h_j(1 - h_j)$:

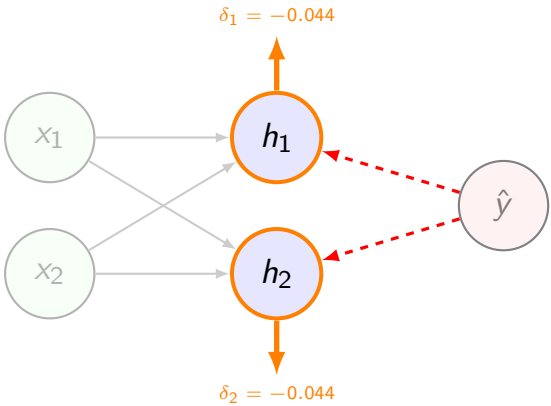
$$\sigma'(z_1) = 0.679 \times (1 - 0.679) = 0.218$$

For Hidden Neuron 1:

$$\begin{aligned}\delta_1 &= \delta_{\text{out}} \cdot w_1 \cdot \sigma'(z_1) \\ &= (-0.204) \times 1.0 \times 0.218 \quad = \boxed{-0.044}\end{aligned}$$

For Hidden Neuron 2:

$$\begin{aligned}\delta_2 &= \delta_{\text{out}} \cdot w_2 \cdot \sigma'(z_2) \\ &= (-0.204) \times 1.0 \times 0.218 \quad = \boxed{-0.044}\end{aligned}$$



Hidden errors computed!
Error propagated backward

Step 6: Gradients for Hidden Weights (v_{ij})

Gradient Formula: $\dfrac{\partial \mathcal{L}}{\partial v_{ij}} = \delta_j \cdot x_i$

Weights to Hidden Neuron 1:

$$\frac{\partial \mathcal{L}}{\partial v_{11}} = \delta_1 \cdot x_1 = (-0.044) \times 0.5 = \boxed{-0.022}$$

$$\frac{\partial \mathcal{L}}{\partial v_{21}} = \delta_1 \cdot x_2 = (-0.044) \times 1.0 = \boxed{-0.044}$$

Weights to Hidden Neuron 2:

$$\frac{\partial \mathcal{L}}{\partial v_{12}} = \delta_2 \cdot x_1 = (-0.044) \times 0.5 = \boxed{-0.022}$$

$$\frac{\partial \mathcal{L}}{\partial v_{22}} = \delta_2 \cdot x_2 = (-0.044) \times 1.0 = \boxed{-0.044}$$

All Gradients Computed!

All hidden layer weights have negative gradients $\rightarrow$ should increase.

Connecting Backpropagation to Optimization

The Story So Far:

- **Forward Pass:** We mapped inputs to predictions and calculated the Loss $\mathcal{L}$.
- **Backpropagation:** We used the chain rule to find exactly how each weight contributes to that loss (∇W).

The Missing Link: Now that we have the gradients, how do we actually **improve** the model?

1. The gradient $\frac{\partial \mathcal{L}}{\partial W}$ points in the direction of steepest *increase*.
2. To minimize error, we must move in the **opposite direction**.

Next Step: Gradient Descent Algorithm and its variants

We will now see how this update rule allows the model to "learn" by iteratively descending the loss landscape.