Lecture 5:

From Logistic Regression to Multi-layer Perceptrons

Ho-min Park

homin.park@ghent.ac.kr

powersimmani@gmail.com

Lecture Contents

Part 1: Neural Network Motivation

Part 2: MLP Structure and Forward Propagation

Part 3: Backpropagation and Learning

Part 1/3:

Neural Network Motivation

1. Limitations of Logistic Regression
2. XOR Problem - Linear Inseparability
3. Need for Feature Space Transformation
4. The Idea of Multi-layer Structure
5. Biological Neuron vs Artificial Neuron
6. Role of Activation Functions
7. Universal Approximation Theorem
8. Expressiveness and Depth

Limitations of Logistic Regression

1 Linear Decision Boundary

Can only separate data with linear boundaries. Fails on non-linear patterns like XOR problem.

2 Limited Feature Interactions

Cannot automatically learn complex feature interactions without manual feature engineering.

3 Single-layer Architecture

Lacks hierarchical representation learning. Cannot build abstract features from raw inputs.

4 Low Model Capacity

Limited expressiveness for complex patterns. Struggles with high-dimensional non-linear data.

↓

Solution: Multi-layer Neural Networks

Neural networks overcome these limitations through multiple layers and non-linear activations

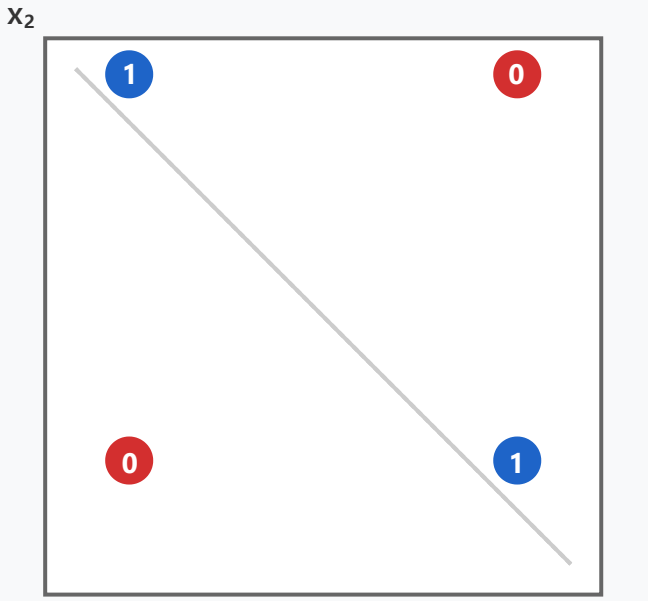
XOR Logic Gate

Truth Table

x_1	x_2	Output
0	0	0
0	1	1
1	0	1
1	1	0

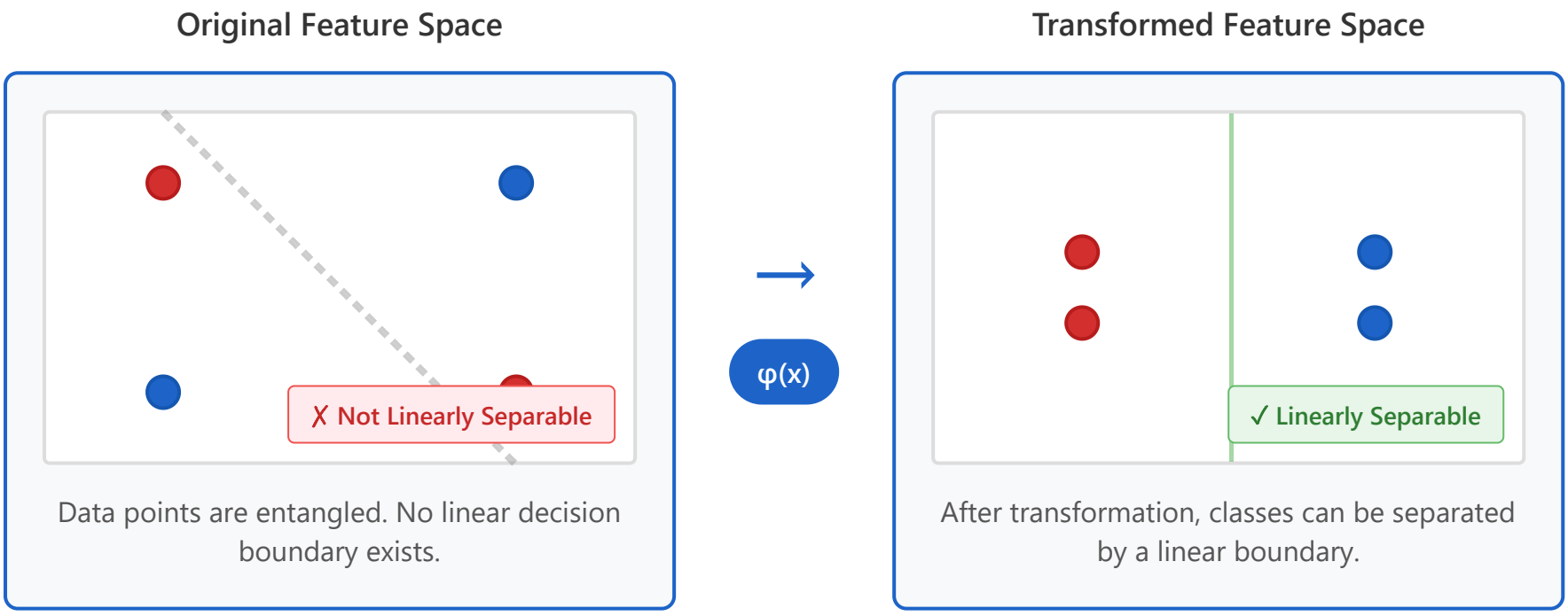
Key Issue: No single straight line can separate the blue and red points. This demonstrates why logistic regression cannot solve XOR.

Feature Space Visualization



Output = 1 Output = 0

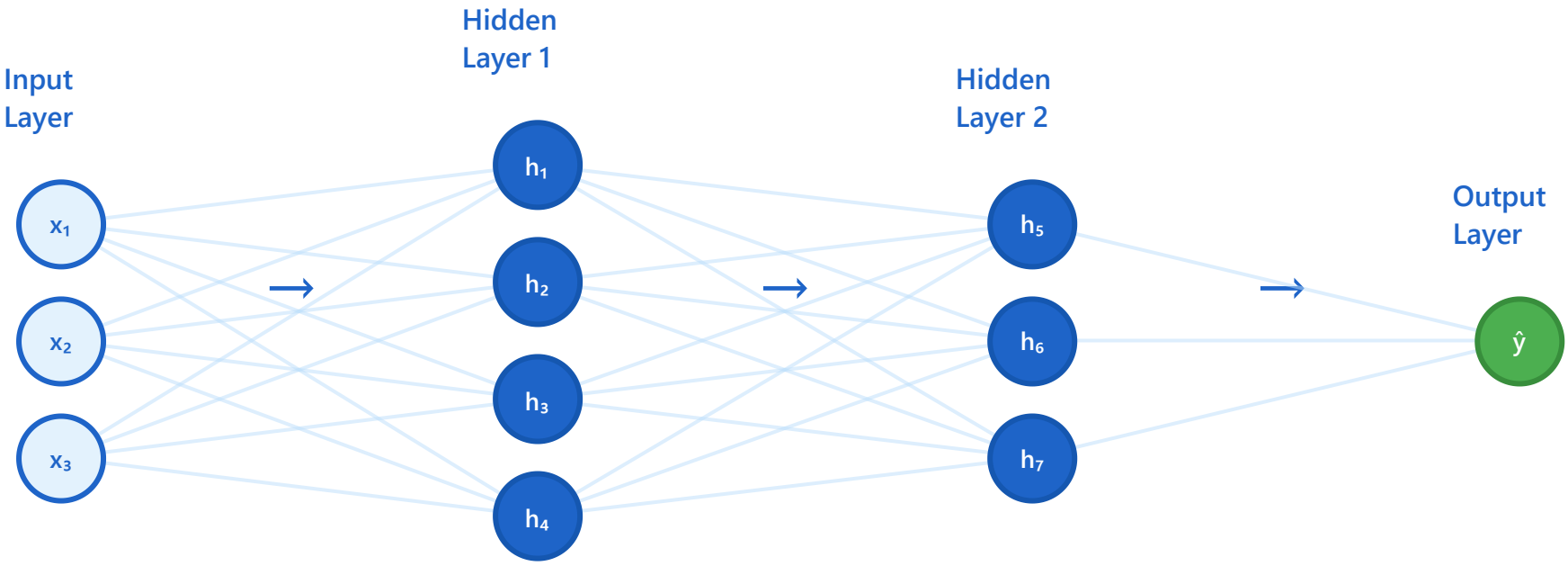
Need for Feature Space Transformation



Key Insight: Neural networks learn non-linear transformations φ(x) automatically through hidden layers, mapping complex patterns into spaces where they become linearly separable.

Class 0 Class 1

The Idea of Multi-layer Structure



Layer Hierarchy

Each layer transforms input features into increasingly abstract representations

Non-linear Mapping

Activation functions enable learning complex non-linear decision boundaries

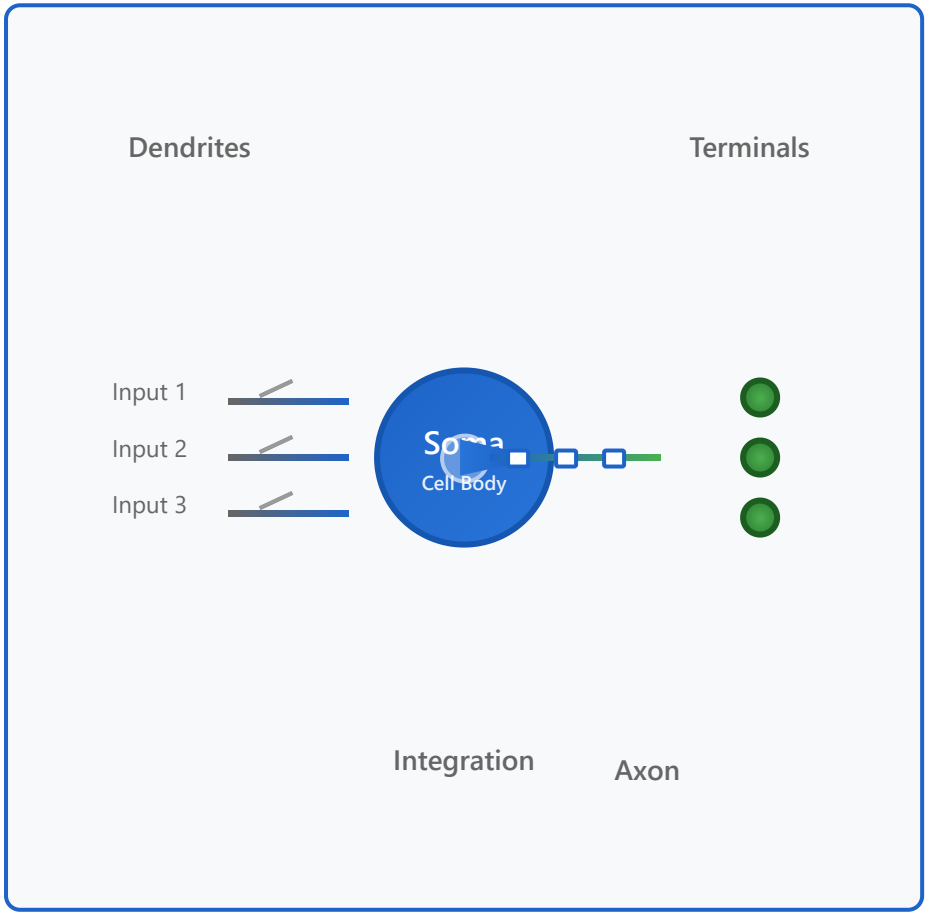
Depth Advantage

Multiple layers allow hierarchical feature learning and greater expressiveness

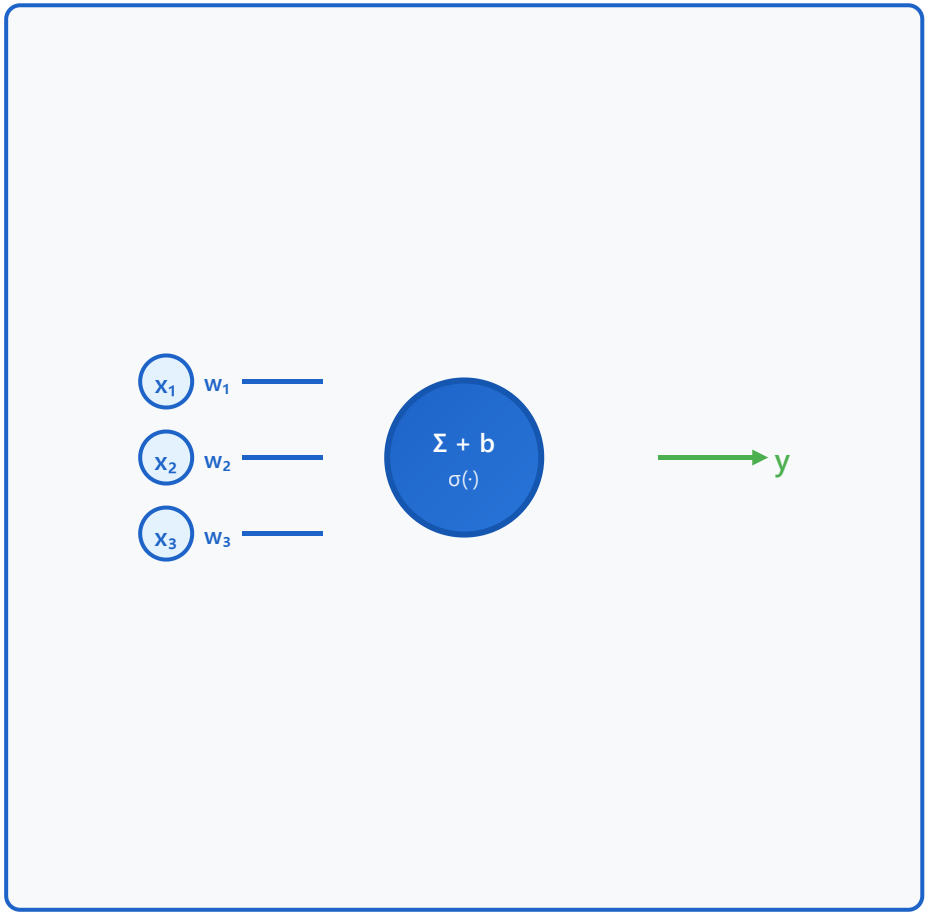
Core Principle: By stacking multiple layers with non-linear activations, neural networks can learn to transform linearly inseparable problems into linearly separable ones

Biological Neuron vs Artificial Neuron

Biological Neuron



Artificial Neuron (Perceptron)



Dendrites → Soma → Axon ≈ Weighted Inputs → Summation & Activation → Output

🔢 Perceptron Calculation Example

📊 Given Data

Variable	Value	Description
x_1	2.0	Input 1
x_2	3.0	Input 2
x_3	-1.0	Input 3
w_1	0.5	Weight 1
w_2	-0.3	Weight 2
w_3	0.8	Weight 3
b	1.0	Bias

📐 Perceptron Formula:

$$z = w_1x_1 + w_2x_2 + w_3x_3 + b$$

$$y = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

📝 Step-by-Step Calculation

Step 1: Calculate Weighted Sum

Multiply each input by its weight and sum all products

$$z = w_1x_1 + w_2x_2 + w_3x_3 + b$$
$$= (0.5 \times 2.0) + (-0.3 \times 3.0) + (0.8 \times -1.0) + 1.0$$
$$= 1.0 + (-0.9) + (-0.8) + 1.0$$

Step 2: Sum All Terms

Add all terms together

$$y = \sigma(\Sigma\, w_i x_i + b)$$
$$= \sigma(w_1 x_1 + w_2 x_2 + w_3 x_3 + b)$$

🔄 Activation Function (Step Function):

$$\sigma(z) = 1 \text{ if } z \geq 0$$
$$\sigma(z) = 0 \text{ if } z < 0$$

$$z = 1.0 - 0.9 - 0.8 + 1.0$$
$$z = 0.3$$

Step 3: Apply Activation Function

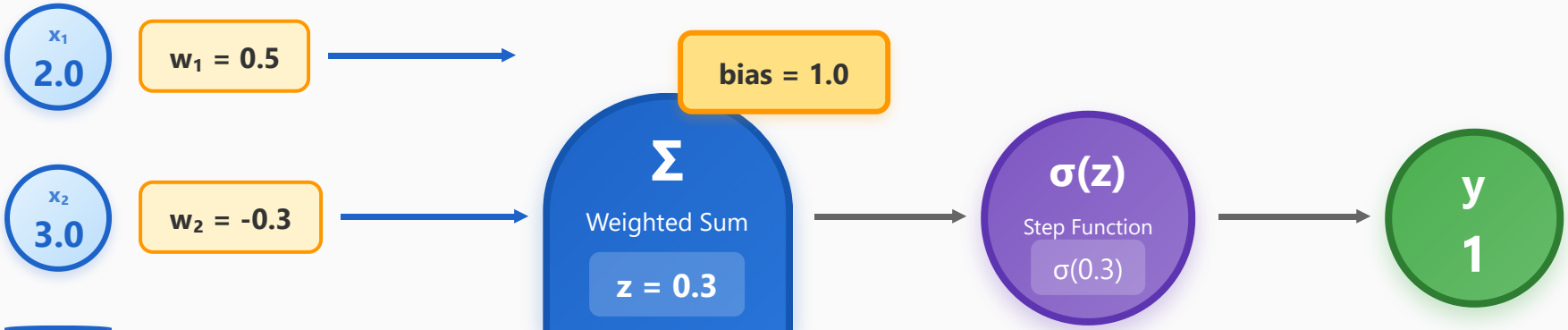
Apply Step function: Since z ≥ 0

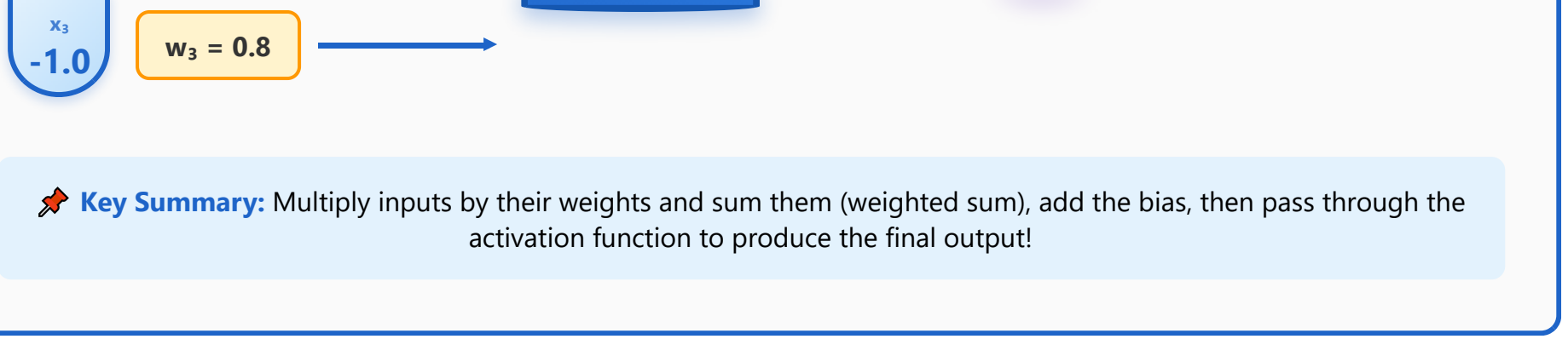
$$y = \sigma(0.3) = 1$$
(Activated because 0.3 ≥ 0!)

🎯 **Final Output**

y = 1

🎨 Visualization: Perceptron Operation Process





📌 **Key Summary:** Multiply inputs by their weights and sum them (weighted sum), add the bias, then pass through the activation function to produce the final output!

Role of Activation Functions

✗ Without Activation

$$y = W_2(W_1x + b_1) + b_2$$

$$y = W'x + b'$$

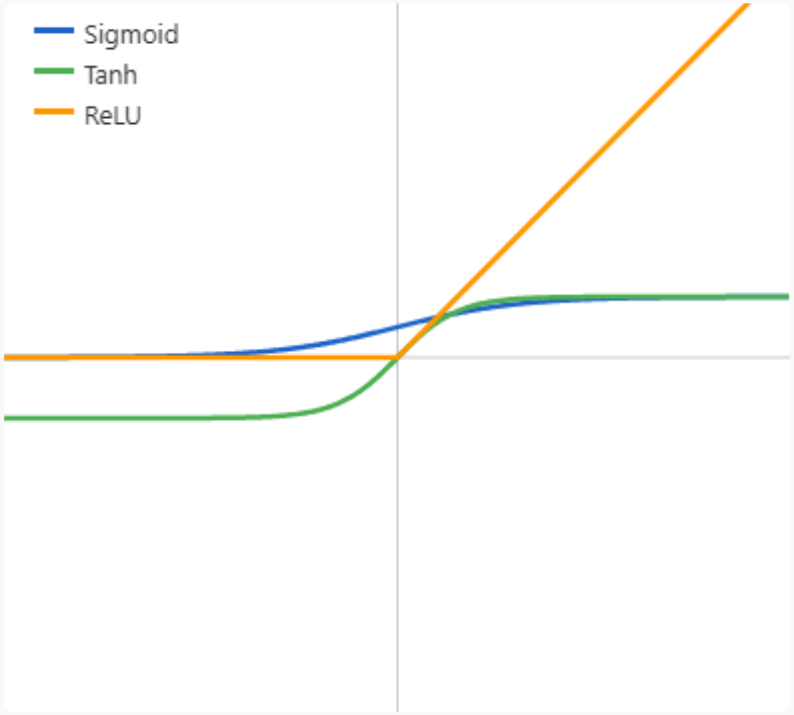
Multiple layers collapse into a single linear transformation. No advantage over logistic regression!

✓ With Activation

$$y = \sigma(W_2\sigma(W_1x + b_1) + b_2)$$

Non-linear transformations enable learning complex patterns and hierarchical representations.

Common Activation Functions



1 Non-linearity

Introduce non-linear transformations to learn complex patterns

2 Gradient Flow

Enable backpropagation by providing differentiable functions

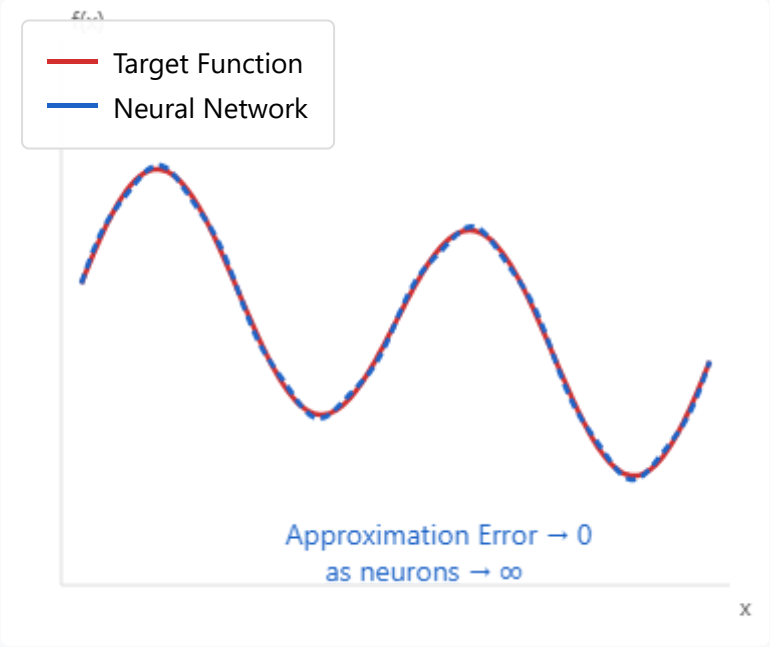
3 Expressiveness

Increase model capacity to approximate complex functions

Universal Approximation Theorem

A neural network with a single hidden layer containing a finite number of neurons can approximate any continuous function on a compact subset of $\mathbb{R}^n$ to arbitrary accuracy.

Function Approximation



Key Insights

1 Theoretical Power

Single hidden layer is theoretically sufficient for any continuous function

2 Width vs Depth

May need exponentially many neurons; deeper networks are more efficient

3 Practical Learning

Theorem doesn't guarantee efficient learning or generalization

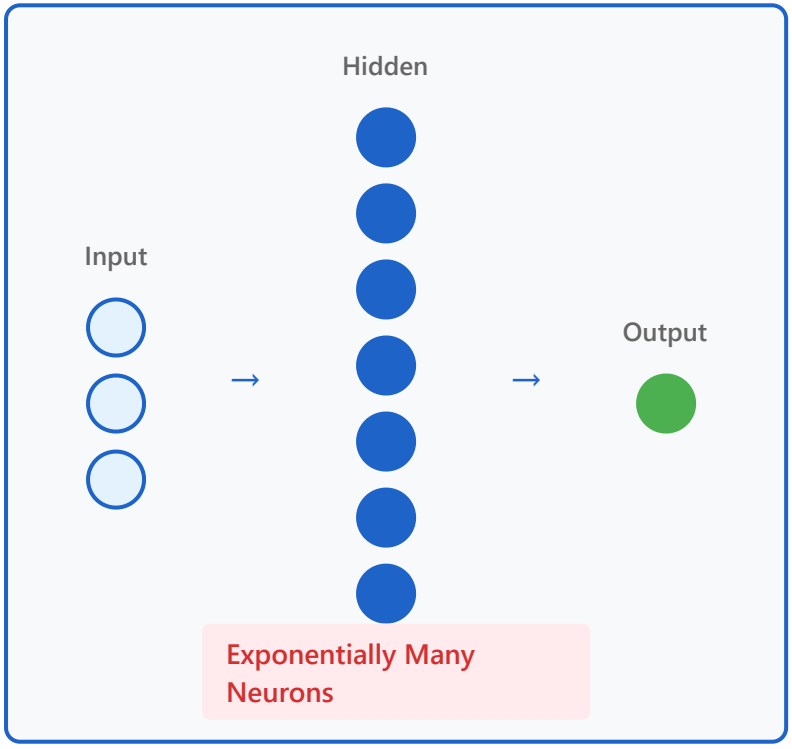
⚡ Practical Implication

While theoretically powerful, we use deep networks in practice because they learn hierarchical features more efficiently and generalize better than wide shallow networks.

Expressiveness and Depth

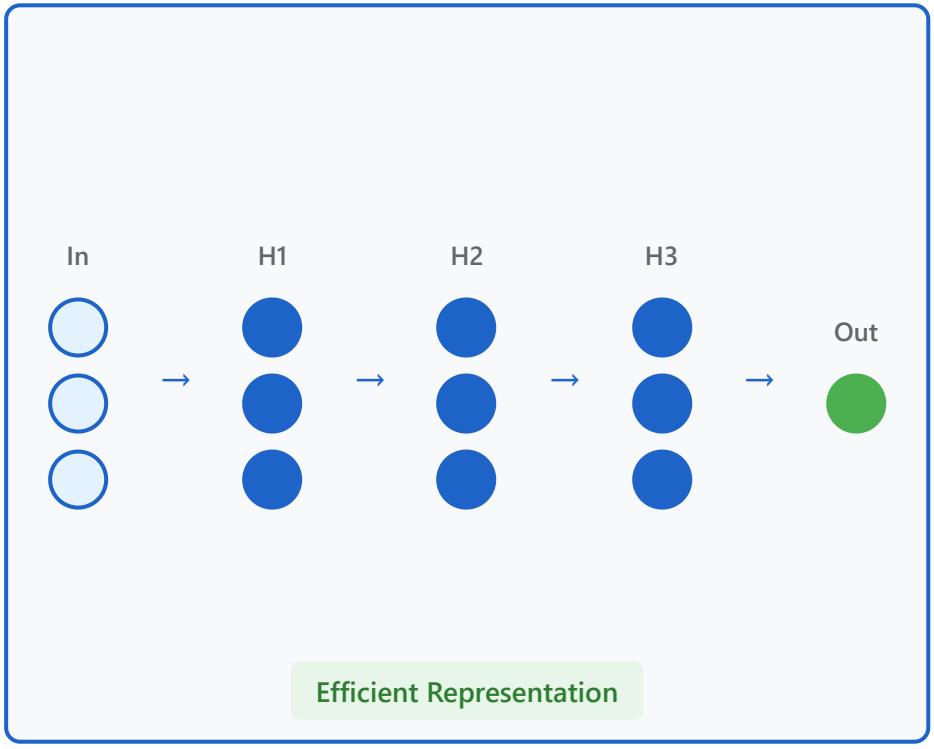
Shallow & Wide

Single Hidden Layer



Deep & Narrow

Multiple Hidden Layers



Number of Parameters

$O(2^n)$ vs $O(n^2)$

Feature Learning

Flat vs Hierarchical

Generalization

Weaker vs Stronger

Compositionality

Limited vs Natural

Key Insight: Deep networks learn hierarchical abstractions efficiently, achieving exponential expressiveness with polynomial parameters through compositional depth

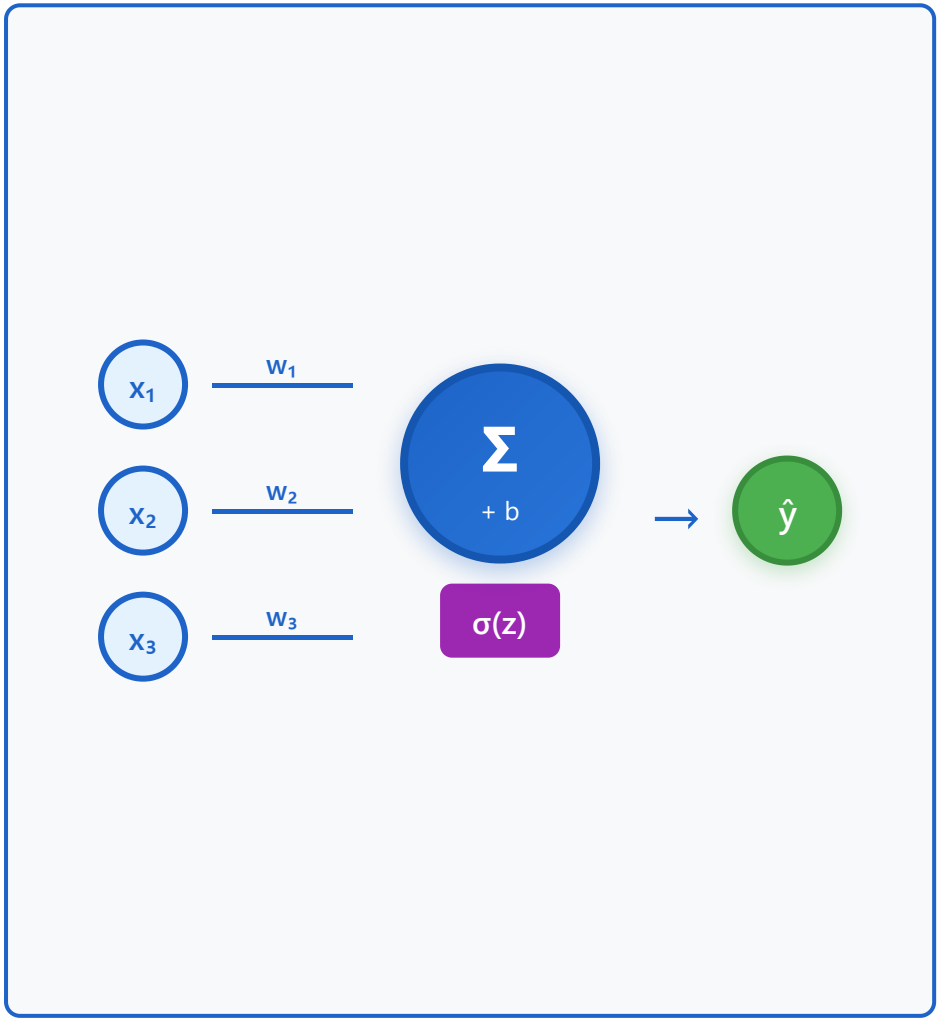
Part 2/3:

MLP Structure and Forward Propagation

9. Single-layer Perceptron Review
10. Multi-layer Perceptron Architecture
11. Weights and Biases
12. Forward Propagation Algorithm
13. Activation Functions - Sigmoid, Tanh
14. ReLU and Its Variants
15. Output Layer Design (Regression vs Classification)
16. Network Capacity and Complexity

Single-layer Perceptron Review

Architecture



Mathematical Form

Linear Combination

$$z = \Sigma\ w_i x_i + b$$

Activation

$$\hat{y} = \sigma(z)$$

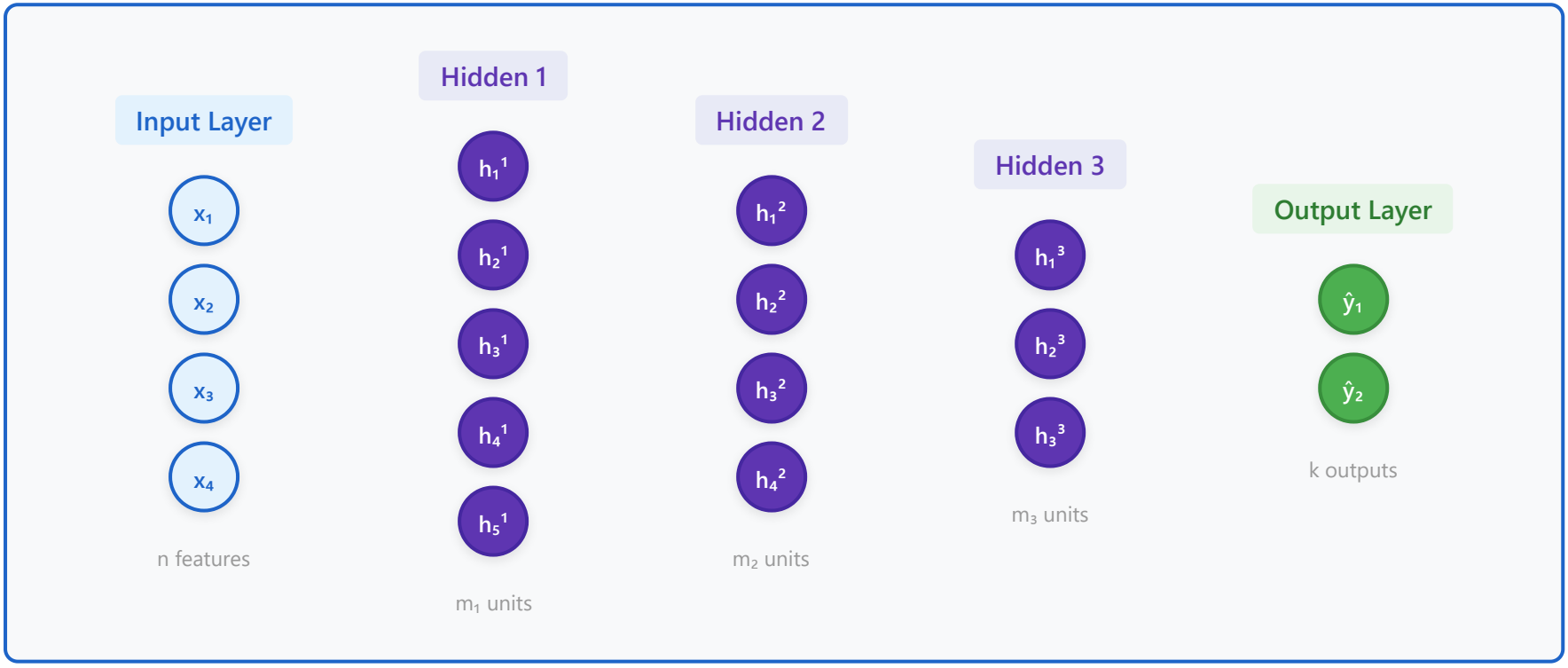
Key Components

- **Weights (w):** Feature importance
- **Bias (b):** Decision threshold
- **Activation (σ):** Non-linearity

⚠ Limitation

Only learns linear decision boundaries - cannot solve XOR problem

Multi-layer Perceptron Architecture



Layers
L = 5

Hidden Layers
3 layers

Parameters
Weights + Biases

Connectivity
Fully Connected

$$h^{(l)} = \sigma(W^{(l)}h^{(l-1)} + b^{(l)}) \rightarrow \text{Layer-wise Computation}$$

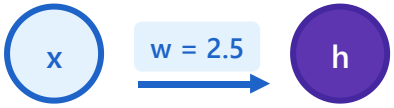
Weights and Biases

Weights (W)

Matrix Form

$$W^{(l)} \in \mathbb{R}^{(m \times n)}$$

$$\begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$



1. **Connection strength** between neurons
2. **Feature importance:** larger |w| = more influence
3. **Learned during training** via gradient descent

Biases (b)

Vector Form

$$b^{(l)} \in \mathbb{R}^{m}$$

$$\begin{bmatrix} b_1 \\ b_2 \\ \vdots \end{bmatrix}$$

1. **Shifts activation:** controls threshold
2. **Model flexibility:** allows fitting data better
3. **One per neuron** in the layer

Layer Computation

Weighted Sum

$W^{(l)}h^{(l-1)}$

+

Bias Term

$b^{(l)}$

Complete Formula: $z^{(l)} = W^{(l)}h^{(l-1)} + b^{(l)} \rightarrow h^{(l)} = \sigma(z^{(l)})$

Forward Propagation Algorithm

Step-by-Step Process

1 Input Layer

$h^{(0)} = x$

↓

2 Linear Transformation

$z^{(l)} = W^{(l)}h^{(l-1)} + b^{(l)}$

↓

3 Activation Function

$h^{(l)} = \sigma(z^{(l)})$

↓

4 Repeat for All Layers

$l = 1, 2, \ldots, L$

↓

5 Output Layer

$\hat{y} = h^{(L)}$

Pseudocode

Forward Propagation

```
# Input: x, weights W, biases b
function forward_propagation(x):
    h = x  # Initialize with input

    for l = 1 to L:
        # Linear combination
        z = W[l] * h + b[l]

        # Apply activation
        h = activation(z)

    return h  # Final output ŷ
```

Example: 3-Layer Network

$x \rightarrow [W^1, b^1] \rightarrow \sigma \rightarrow h^1 \rightarrow [W^2, b^2] \rightarrow \sigma \rightarrow h^2 \rightarrow [W^3, b^3] \rightarrow \sigma \rightarrow \hat{y}$

Sequential Flow

Data flows layer by layer from input to output

Two Operations

Each layer: (1) linear transform (2) activation

Matrix Operations

Efficient computation using matrix multiplication

Numerical Example: Step-by-Step Calculation

2-Layer Network with Sigmoid Activation ($\sigma(z) = 1/(1+e^{-z})$)

Initial Setup

Input: $x = [1.0, 2.0]^T$

Layer 1 Weights: $W^{(1)} = [[0.5, 0.3], [0.2, 0.4]]$

Layer 1 Bias: $b^{(1)} = [0.1, 0.2]^T$

Layer 2 Weights: $W^{(2)} = [[0.6, 0.7]]$

Layer 2 Bias: $b^{(2)} = [0.3]$

Layer 1: Hidden Layer

Step 1: Linear Transformation

$$z^{(1)} = W^{(1)}x + b^{(1)}$$

$z_1^{(1)} = 0.5(1.0) + 0.3(2.0) + 0.1 = 0.5 + 0.6 + 0.1 = \mathbf{1.2}$

$z_2^{(1)} = 0.2(1.0) + 0.4(2.0) + 0.2 = 0.2 + 0.8 + 0.2 = \mathbf{1.2}$

$$z^{(1)} = [1.2, 1.2]^T$$

↓

Step 2: Activation Function

$$h^{(1)} = \sigma(z^{(1)})$$

$h_1^{(1)} = \sigma(1.2) = 1/(1+e^{-1.2}) \approx \mathbf{0.769}$

$h_2^{(1)} = \sigma(1.2) = 1/(1+e^{-1.2}) \approx \mathbf{0.769}$

$$h^{(1)} = [0.769, 0.769]^T$$

🎯 Layer 2: Output Layer

Step 3: Linear Transformation

$$z^{(2)} = W^{(2)}h^{(1)} + b^{(2)}$$

$z^{(2)} = 0.6(0.769) + 0.7(0.769) + 0.3$

$z^{(2)} = 0.461 + 0.538 + 0.3 = \mathbf{1.299}$

$$z^{(2)} = [1.299]$$

↓

Step 4: Final Activation

$$\hat{y} = h^{(2)} = \sigma(z^{(2)})$$

$\hat{y} = \sigma(1.299) = 1/(1+e^{-1.299}) \approx \mathbf{0.786}$

$$\hat{y} = 0.786$$

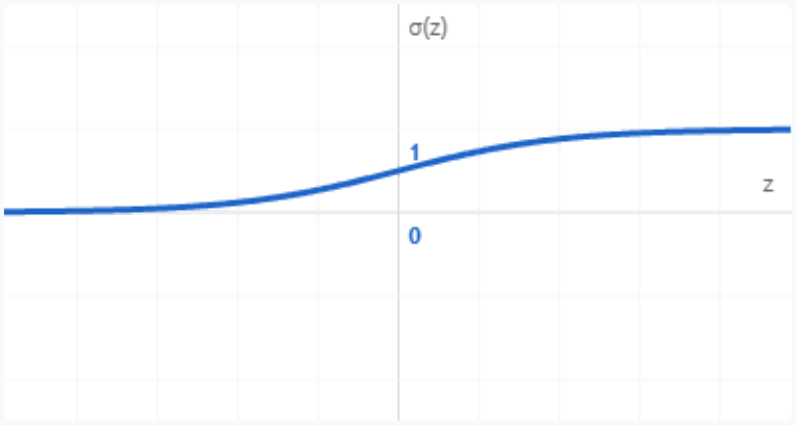
✅ Summary

Input		Hidden		Output
[1.0, 2.0]	→	[0.769, 0.769]	→	0.786

💡 This output represents the network's prediction for the given input after forward propagation through all layers.

Activation Functions: Sigmoid & Tanh

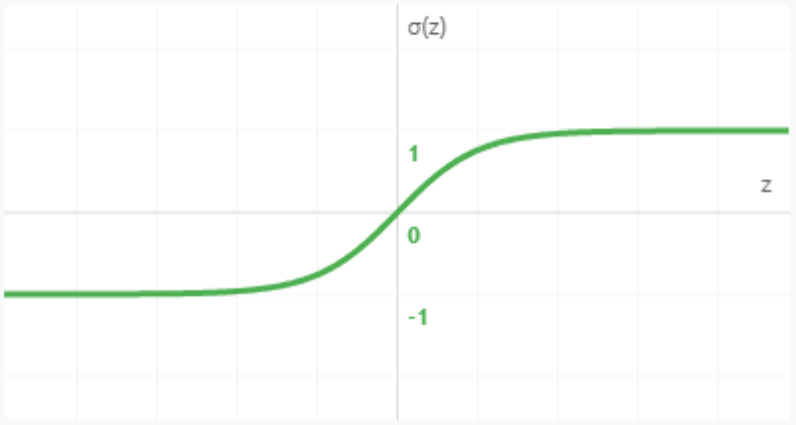
Sigmoid (σ)



Formula

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

Tanh (tanh)



Formula

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

Key Properties

Range	(0, 1)	(-1, 1)
Zero-centered	✗ No	✓ Yes

Gradient	max = 0.25	max = 1.0
Use Case	Output layer	Hidden layers

⚠ Common Issues

- **Vanishing Gradient:** Gradients become very small for large |z|, slowing learning
- **Tanh Advantage:** Zero-centered output helps optimization converge faster

ReLU and Its Variants

ReLU

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

Leaky ReLU

$$f(z) = \max(\alpha z, z)$$
$$\alpha = 0.01$$

ELU

$$f(z) = z \text{ if } z > 0$$
$$\alpha(e^z - 1) \text{ if } z \leq 0$$

ReLU Advantages

- ✓ Simple computation
- ✓ No vanishing gradient
- ✓ Sparse activation
- ✗ Dead neurons (z < 0)

Leaky ReLU

- ✓ Fixes dying ReLU
- ✓ Small negative slope
- ✓ All neurons active
- ✓ Fast convergence

ELU Advantages

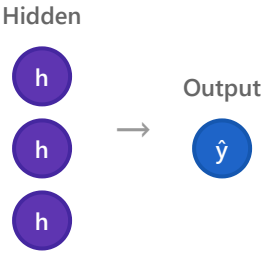
- ✓ Smooth everywhere
- ✓ Negative values push mean to zero
- ✓ Robust to noise
- ✗ Slower (exponential)

Feature	ReLU	Leaky ReLU	ELU
Computation Speed	Fast	Fast	Moderate
Dying Neurons	Yes	No	No

Feature	ReLU	Leaky ReLU	ELU
Zero-Centered	No	No	Nearly
Most Common Use	Default	Alternative	Special Cases

Output Layer Design: Regression vs Classification

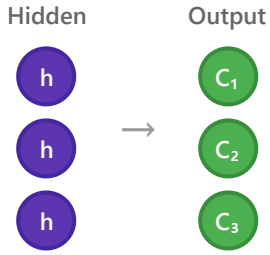
Regression



Linear / No Activation

Output Units:	`1 (or n)`
Activation:	`None (linear)`
Loss Function:	`MSE, MAE`
Output Range:	`(-∞, +∞)`

Classification



Sigmoid / Softmax

Output Units:	`K classes`
Activation:	`Softmax`
Loss Function:	`Cross-Entropy`
Output Range:	`[0, 1] (probs)`

Regression Details

Classification Details

- **Task:** Predict continuous values
- **Examples:** House price, temperature, stock price
- **Loss:** MSE = $(1/n)\Sigma(\hat{y} - y)^2$
- **No constraint** on output values

- **Task:** Predict discrete classes
- **Examples:** Image labels, sentiment, diagnosis
- **Loss:** CE = $-\Sigma\, y \log(\hat{y})$
- **Probabilities** sum to 1.0

📊 Regression Example

Input: [bedroom=3, sqft=2000]
Output: $450,000 (continuous)

🏷️ Classification Example

Input: [image pixels]
Output: [cat: 0.8, dog: 0.15, bird: 0.05]

Network Capacity and Complexity

Capacity Levels

Low Capacity

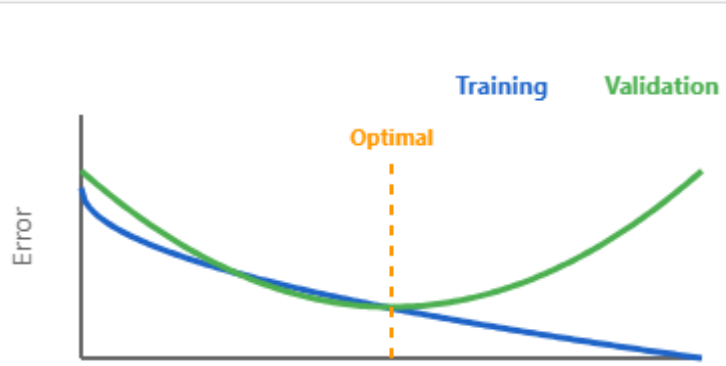
Few parameters
Simple patterns

Medium Capacity

Balanced
Good
generalization

High Capacity

Many parameters
Risk of overfitting



Key Factors

1. **Number of Layers**
   Depth increases model expressiveness

2. **Units per Layer**
   Width affects pattern complexity

3. **Total Parameters**
   More params = higher capacity

Parameters Count

$$P = \Sigma\ (n_l \times n_{l+1} + n_{l+1})$$

📉 Underfitting

Too simple model
High bias
→ Increase capacity

✓ Optimal

Right complexity
Good generalization
→ Balance achieved

📈 Overfitting

Too complex model
High variance
→ Reduce capacity / Regularize

Part 3/3:

Backpropagation and Learning

17. Loss Function Definition
18. Chain Rule
19. Backpropagation Algorithm Derivation
20. Computational Graph
21. Gradient Calculation Example
22. Automatic Differentiation (Autograd)
23. Mini-batch Gradient Descent
24. Implementation Tips and Debugging
25. PyTorch/TensorFlow Hands-on

Loss Function Definition

A loss function L(ŷ, y) measures the discrepancy between the predicted output ŷ and the true target y, guiding the network to minimize prediction errors during training.

Regression Tasks

Mean Squared Error (MSE)

$$L = (1/n) \Sigma (\hat{y}_i - y_i)^2$$

- **Penalizes large errors** quadratically
- **Differentiable** everywhere
- **Sensitive to outliers**

Example Calculation

Prediction		Target		Loss
5.2	−	4.0	→	1.44

Classification Tasks

Cross-Entropy Loss

$$L = -\Sigma y_i \log(\hat{y}_i)$$

- **Measures probability distribution** difference
- **Works with softmax** output
- **Penalizes confident mistakes** heavily

Example Calculation

Prediction		Target		Loss
0.7	log	1	→	0.36

Common Loss Functions

MSE

Task: Regression

Formula: $\Sigma(\hat{y} - y)^2$

Use Case: Continuous values

Cross-Entropy

Task: Classification

Formula: $-\Sigma\ y\ \log(\hat{y})$

Use Case: Class probabilities

MAE

Task: Regression

Formula: $\Sigma|\hat{y} - y|$

Use Case: Robust to outliers

🎯 Training Objective

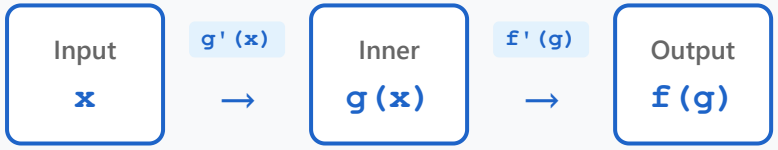
The goal is to find parameters θ that minimize the average loss: $\boldsymbol{\theta^* = \text{argmin}\ (1/n)\ \Sigma\ L(f(x_i; \theta), y_i)}$

Chain Rule

Mathematical Foundation

Chain Rule Formula

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



The derivative flows backward:

$$\partial f/\partial x = \partial f/\partial g \times \partial g/\partial x$$

Step-by-Step Example

Compute: d/dx $\sin(x^2)$

1. $f(g) = \sin(g),\ g(x) = x^2$
2. $f'(g) = \cos(g) = \cos(x^2)$
3. $g'(x) = 2x$
4. $d/dx = \cos(x^2) \cdot 2x$

Applications in Neural Networks

Multi-layer Composition

Neural networks chain multiple functions through layers

$$y = f_3(f_2(f_1(x)))$$
$$\partial y/\partial x = \partial f_3/\partial f_2 \cdot \partial f_2/\partial f_1 \cdot \partial f_1/\partial x$$

Backpropagation

Gradients propagate backward through the network

$$\partial L/\partial w^{(1)} = \partial L/\partial z^{(1)} \cdot \partial z^{(1)}/\partial w^{(1)}$$
$$\text{where } z^{(1)} = w^{(1)} h^{(1-1)} + b^{(1)}$$

Key Insight

The chain rule is the mathematical foundation of backpropagation. It allows us to compute gradients efficiently by decomposing complex derivatives into simpler parts, multiplying local gradients as we traverse backward through the computational graph.

Backpropagation Algorithm Derivation

Derivation Steps

1 Output Layer Error

$$\delta^{(L)} = \partial L/\partial z^{(L)} = \partial L/\partial a^{(L)} \cdot \sigma'(z^{(L)})$$

Error at output layer using chain rule

2 Hidden Layer Error

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

Propagate error backward through weights

3 Weight Gradient

$$\partial L/\partial W^{(l)} = \delta^{(l)} (a^{(l-1)})^T$$

Gradient w.r.t. weights at layer l

4 Bias Gradient

$$\partial L/\partial b^{(l)} = \delta^{(l)}$$

Forward & Backward Pass

Information Flow

Input
x, W, b

↓ **Forward**

Compute
z, a, L

↑ **Backward**

Compute
∂L/∂W, ∂L/∂b

↓ **Update**

Output
W', b'

Key Formulas

Gradient w.r.t. biases at layer l

5 Parameter Update

$$W^{(l)} \leftarrow W^{(l)} - \eta \cdot \partial L / \partial W^{(l)}$$

Update parameters using learning rate η

Forward:

$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}$$

Activation:

$$a^{(l)} = \sigma(z^{(l)})$$

Backward:

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

Backpropagation Algorithm

```
# Forward Pass
for l = 1 to L:
    z[l] = W[l] * a[l-1] + b[l]
    a[l] = activation(z[l])

# Compute Loss
L = loss_function(a[L], y)

# Backward Pass
delta[L] = dL/da[L] * activation_derivative(z[L])
for l = L-1 to 1:
    delta[l] = (W[l+1].T * delta[l+1]) * activation_derivative(z[l])
    dW[l] = delta[l] * a[l-1].T
    db[l] = delta[l]

# Update Parameters
for l = 1 to L:
    W[l] -= learning_rate * dW[l]
    b[l] -= learning_rate * db[l]
```

Key Insight: Backpropagation efficiently computes gradients by reusing intermediate values from the forward pass and applying the chain rule backward through layers. Each layer's gradient depends only on the next layer's gradient and local derivatives.

🔢 Numerical Example: Complete Backpropagation

Tiny Network (2-2-1) with Sigmoid Activation

Network Architecture


```
Input: x = [0.5, 1.0]
```

```
Target: y = 1.0
```

$$W^{(1)} = [[0.4, 0.6], [0.3, 0.5]]$$

$$b^{(1)} = [0.1, 0.2]$$

$$W^{(2)} = [[0.7, 0.8]]$$

$$b^{(2)} = [0.3]$$

→ Phase 1: Forward Propagation

Layer 1: Input → Hidden

$$z^{(1)} = W^{(1)}x + b^{(1)}$$

$z_1^{(1)} = 0.4(0.5) + 0.6(1.0) + 0.1 = 0.2 + 0.6 + 0.1 = \mathbf{0.9}$

$z_2^{(1)} = 0.3(0.5) + 0.5(1.0) + 0.2 = 0.15 + 0.5 + 0.2 = \mathbf{0.85}$

$$\mathbf{z^{(1)} = [0.9,\ 0.85]}$$

$$a^{(1)} = \sigma(z^{(1)})$$

$a_1^{(1)} = \sigma(0.9) = 1/(1+e^{-0.9}) \approx \mathbf{0.711}$

$a_2^{(1)} = \sigma(0.85) = 1/(1+e^{-0.85}) \approx \mathbf{0.701}$

$$\mathbf{a^{(1)} = [0.711,\ 0.701]}$$

Layer 2: Hidden → Output

$$z^{(2)} = W^{(2)}a^{(1)} + b^{(2)}$$

$z^{(2)} = 0.7(0.711) + 0.8(0.701) + 0.3$

$z^{(2)} = 0.498 + 0.561 + 0.3 = \mathbf{1.359}$

$$\mathbf{z^{(2)} = 1.359}$$

$$\hat{y} = \sigma(z^{(2)})$$

$\hat{y} = \sigma(1.359) = 1/(1+e^{-1.359}) \approx \mathbf{0.796}$

$$\mathbf{\hat{y} = 0.796}$$

Loss Calculation (MSE)

$$L = \tfrac{1}{2}(y - \hat{y})^2$$

$L = \frac{1}{2}(1.0 - 0.796)^2$

$L = \frac{1}{2}(0.204)^2 = \frac{1}{2}(0.0416) = \mathbf{0.0208}$

```
Loss = 0.0208
```

← Phase 2: Backward Propagation (Computing Gradients)

Step 1: Output Layer Gradient

$$\partial L/\partial z^{(2)} = (\hat{y} - y) \cdot \sigma'(z^{(2)})$$

$\sigma'(z^{(2)}) = \sigma(z^{(2)})(1 - \sigma(z^{(2)})) = 0.796(1 - 0.796) = 0.796 \times 0.204 = \mathbf{0.162}$

$\partial L/\partial z^{(2)} = (0.796 - 1.0) \times 0.162 = -0.204 \times 0.162 = \mathbf{-0.033}$

```
δ⁽²⁾ = -0.033
```

Step 2: Layer 2 Weight & Bias Gradients

$$\partial L/\partial W^{(2)} = \delta^{(2)} \cdot (a^{(1)})^T$$

$\partial L/\partial w_1^{(2)} = -0.033 \times 0.711 = \mathbf{-0.023}$

$\partial L/\partial w_2^{(2)} = -0.033 \times 0.701 = \mathbf{-0.023}$

```
∂L/∂W⁽²⁾ = [-0.023, -0.023]
```

$$\partial L/\partial b^{(2)} = \delta^{(2)}$$

```
∂L/∂b⁽²⁾ = -0.033
```

Step 3: Hidden Layer Gradient

$$\partial L/\partial z^{(1)} = (W^{(2)})^T \cdot \delta^{(2)} \cdot \sigma'(z^{(1)})$$

$\sigma'(z_1^{(1)}) = 0.711(1 - 0.711) =$ **0.205**

$\sigma'(z_2^{(1)}) = 0.701(1 - 0.701) =$ **0.210**

$\partial L/\partial z_1^{(1)} = 0.7 \times (-0.033) \times 0.205 =$ **-0.0047**

$\partial L/\partial z_2^{(1)} = 0.8 \times (-0.033) \times 0.210 =$ **-0.0055**

$$\delta^{(1)} = [-0.0047, -0.0055]$$

Step 4: Layer 1 Weight & Bias Gradients

$$\partial L/\partial W^{(1)} = \delta^{(1)} \cdot x^T$$

$\partial L/\partial w_{11}^{(1)} = -0.0047 \times 0.5 =$ **-0.0024**

$\partial L/\partial w_{12}^{(1)} = -0.0047 \times 1.0 =$ **-0.0047**

$\partial L/\partial w_{21}^{(1)} = -0.0055 \times 0.5 =$ **-0.0028**

$\partial L/\partial w_{22}^{(1)} = -0.0055 \times 1.0 =$ **-0.0055**

$$\partial L/\partial W^{(1)} = [[-0.0024, -0.0047], [-0.0028, -0.0055]]$$

$$\partial L/\partial b^{(1)} = \delta^{(1)}$$

$$\partial L/\partial b^{(1)} = [-0.0047, -0.0055]$$

✅ All Computed Gradients

$\partial L/\partial W^{(2)}$

$[-0.023, -0.023]$

$\partial L/\partial b^{(2)}$

-0.033

$\partial L/\partial W^{(1)}$ (row 1)

$[-0.0024, -0.0047]$

$\partial L/\partial W^{(1)}$ (row 2)

$[-0.0028, -0.0055]$

$\partial L/\partial b^{(1)}$

$[-0.0047, -0.0055]$

🔄 Parameter Update (Learning Rate $\alpha = 0.5$)

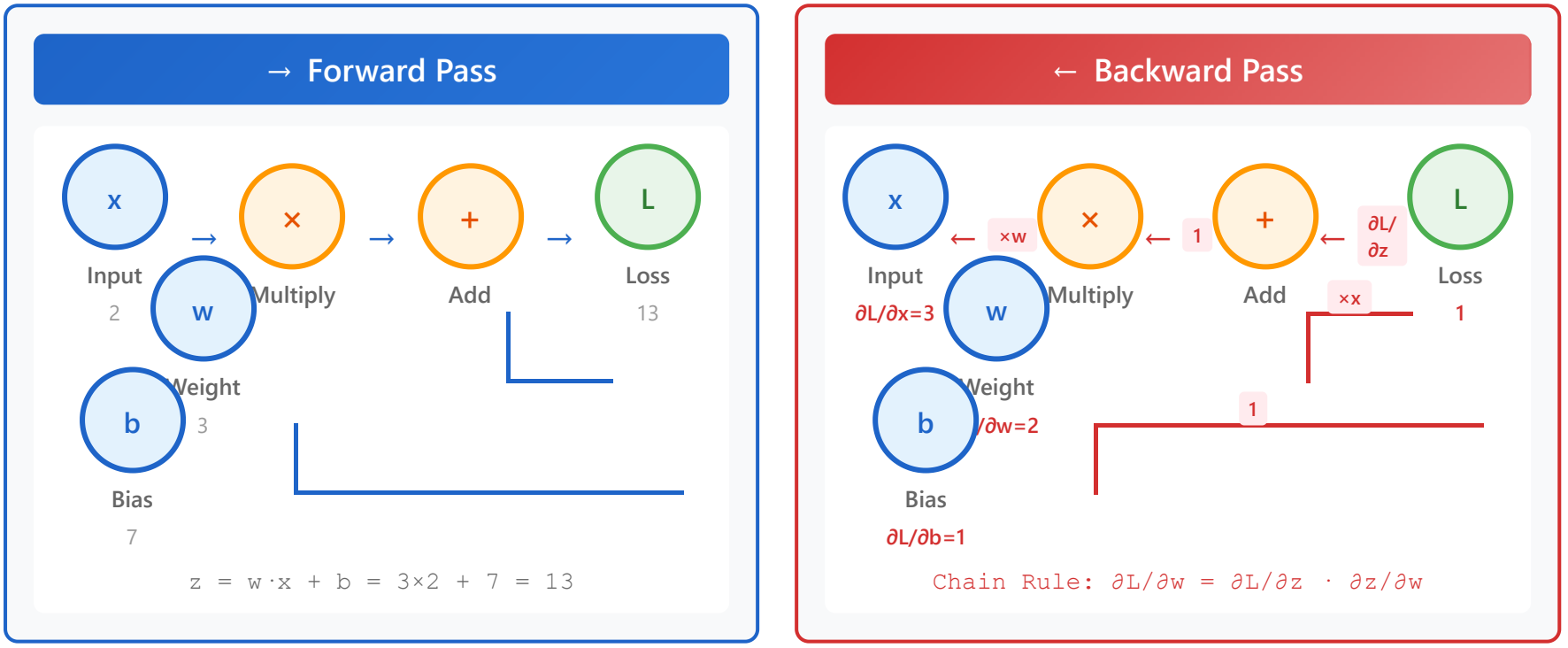
$$W^{(2)}_{new} = [0.7, 0.8] - 0.5 \times [-0.023, -0.023] = [0.7115, 0.8115]$$

$$b^{(2)}_{new} = 0.3 - 0.5 \times (-0.033) = 0.3165$$

$$W^{(1)}_{new} = [[0.4012, 0.6024], [0.3014, 0.5028]]$$

$$b^{(1)}_{new} = [0.1024, 0.2028]$$

Computational Graph



Nodes

Variables and operations represented as graph nodes

Edges

Dependencies between operations shown as directed edges

Automatic

Frameworks build graphs automatically and compute gradients

Key Advantage

Computational graphs enable automatic differentiation by storing intermediate values during forward pass and applying chain rule systematically during backward pass

Gradient Calculation Example

Simple Network: y = σ(wx + b)

Given Values

Input	Weight	Bias
$x = 2$	$w = 0.5$	$b = 1$

Target	Learning Rate
$y^* = 1$	$\eta = 0.1$

FORWARD Step 1: Linear Combination

$$z = wx + b = 0.5 \times 2 + 1$$

$$z = 2$$

FORWARD Step 2: Activation

$$\hat{y} = \sigma(z) = 1/(1+e^{-2})$$

$$\hat{y} = 0.881$$

FORWARD Step 3: Loss

$$L = (\hat{y} - y^*)^2 = (0.881 - 1)^2$$

Computed Gradients

Final Gradients

$\partial L/\partial w$	$= \partial L/\partial z \cdot x$	-0.05
$\partial L/\partial b$	$= \partial L/\partial z$	-0.025

L = 0.014

BACKWARD **Step 4: ∂L/∂ŷ**

$$\partial L/\partial \hat{y} = 2(\hat{y} - y^*) = 2(0.881 - 1)$$

$\partial L/\partial \hat{y} = -0.238$

BACKWARD **Step 5: ∂L/∂z**

$$\partial L/\partial z = \partial L/\partial \hat{y} \cdot \sigma'(z) = -0.238 \times 0.105$$

$\partial L/\partial z = -0.025$

BACKWARD **Step 6: ∂L/∂w & ∂L/∂b**

$$\partial L/\partial w = \partial L/\partial z \cdot x = -0.025 \times 2$$

$\partial L/\partial w = -0.05$, $\partial L/\partial b = -0.025$

Parameter Update (η = 0.1)

$$w_{new} = 0.5 - 0.1 \times (-0.05) = 0.505$$
$$b_{new} = 1.0 - 0.1 \times (-0.025) = 1.0025$$

Automatic Differentiation (Autograd)

Automatic differentiation computes derivatives algorithmically by applying the chain rule to elementary operations, enabling efficient gradient computation without manual derivation

Manual

Derive gradients by hand using calculus

- ✓ Full control
- ✓ Symbolic form
- ✗ Error-prone
- ✗ Time-consuming
- ✗ Not scalable

Numerical

Approximate using finite differences

- ✓ Easy to implement
- ✗ Approximation errors
- ✗ Slow (O(n) evaluations)
- ✗ Numerical instability

Automatic (AD)

Compute exact derivatives automatically

- ✓ Exact derivatives
- ✓ Efficient O(1) overhead
- ✓ Scalable to complex models
- ✓ No manual derivation

PyTorch Example

```
import torch # Create tensors with gradient tracking x = torch.tensor([2.0], requires_grad=True) w = torch.tensor([0.5], requires_grad=True) b = torch.tensor([1.0], requires_grad=True) # Forward pass (build computation graph) y = torch.sigmoid(w * x + b)
```

Key Features

1. **Dynamic Graph:** Built during forward pass
2. **Chain Rule:** Applied automatically layer by layer

```
loss = (y - 1)**2 # Backward pass (compute gradients) loss.backward() # Access gradients print(w.grad) # ∂loss/∂w print(b.grad) # ∂loss/∂b
```

Efficient: Reuses intermediate values

4 **Flexible:** Supports arbitrary operations

Why It Matters: Autograd eliminates the need for manual gradient derivation, making deep learning accessible and enabling rapid experimentation with complex architectures.

PyTorch

Dynamic graphs

TensorFlow

Eager execution

JAX

Functional AD

Mini-batch Gradient Descent

Batch GD

$$\theta = \theta - \eta \nabla L(\theta)$$

Batch Size: N (all data)

- ✓ Stable convergence
- ✓ Accurate gradient
- ✗ Slow updates
- ✗ Memory intensive
- ✗ May get stuck

Mini-batch GD ★

$$\theta = \theta - \eta \nabla L\_B(\theta)$$

Batch Size: B (32, 64, 128...)

- ✓ Balanced approach
- ✓ Efficient GPU use
- ✓ Good convergence
- ✓ Escapes local minima
- ✓ Industry standard

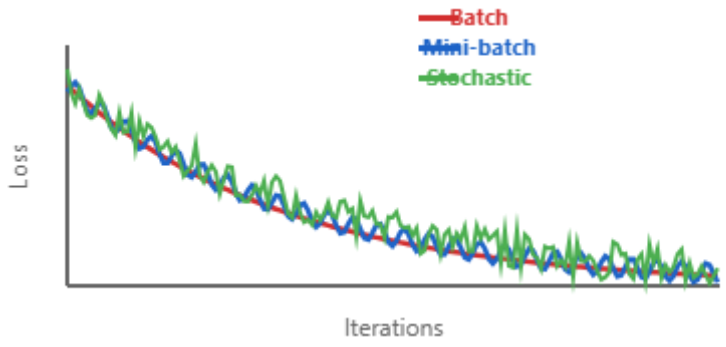
Stochastic GD

$$\theta = \theta - \eta \nabla L\_i(\theta)$$

Batch Size: 1 (single sample)

- ✓ Fast updates
- ✓ Low memory
- ✓ Online learning
- ✗ Noisy gradients
- ✗ Unstable convergence

Convergence Behavior



Key Advantages

- **Efficient:** Balances computation and memory
- **Robust:** Noise helps escape poor local minima
- **Scalable:** Works well with large datasets

Hardware: Optimizes GPU parallelization

Common Batch Sizes

Small datasets: `32, 64`

Medium datasets: `128, 256`

Large datasets: `512, 1024`

Training Process

1. Shuffle data
2. Split into mini-batches
3. Update for each batch
4. Repeat for epochs

Best Practice: Mini-batch GD combines the stability of batch GD with the speed of SGD, making it the default choice for training modern neural networks. The batch size is a critical hyperparameter affecting convergence and generalization.

Implementation Tips and Best Practices

Data & Training

1. **Normalize Inputs**
   Scale features to [0,1] or standardize (mean=0, std=1)
2. **Weight Initialization**
   Use Xavier/He initialization, avoid zeros
3. **Split Data Properly**
   Train (70-80%), Validation (10-15%), Test (10-15%)
4. **Monitor Training**
   Track both training and validation loss
5. **Use Early Stopping**
   Stop when validation loss stops improving

Debugging

1. **Start Simple**
   Begin with small network, overfit small dataset first
2. **Check Gradient Flow**
   Verify gradients are not vanishing or exploding
3. **Visualize Activations**
   Plot activation distributions across layers
4. **Numerical Gradient Check**
   Verify backprop implementation with finite differences
5. **Log Everything**
   Track loss, accuracy, learning rate, gradients

Hyperparameters

Optimization

1. **Learning Rate**
   Start with 0.001-0.01, use learning rate schedules
2. **Batch Size**
   Common: 32, 64, 128. Larger = more stable but slower
3. **Number of Epochs**
   Start high, use early stopping to prevent overfitting
4. **Regularization**
   Try L2 (λ=0.01), dropout (p=0.5), or both
5. **Grid/Random Search**

1. **Use Adam Optimizer**
   Good default choice with adaptive learning rates
2. **Batch Normalization**
   Normalize layer inputs for faster convergence
3. **Learning Rate Decay**
   Reduce LR when validation loss plateaus
4. **Gradient Clipping**
   Prevent exploding gradients (clip at norm=5)
5. **Data Augmentation**

Before Training

- ☑ Data normalized
- ☑ Train/val/test split
- ☑ Weights initialized
- ☑ Loss function chosen

During Training

- ☑ Monitor loss curves
- ☑ Check gradient norms
- ☑ Save best model
- ☑ Log metrics regularly

After Training

- ☑ Evaluate on test set
- ☑ Analyze errors
- ☑ Document results
- ☑ Compare baselines

Common Pitfalls to Avoid

⚠ Not shuffling data • Forgetting to normalize • Using test set for hyperparameter tuning • Setting learning rate too high • Ignoring validation metrics • Not saving checkpoints

🎯 Hands-On: TensorFlow Playground

Visualize and understand how neural networks work! Use TensorFlow Playground to observe the training process in real-time by adjusting various hyperparameters. Start with the Circle dataset and experiment with learning rates, layer structures, activation functions, and more.

Launch TensorFlow Playground

Hands-on with PyTorch/TensorFlow

PyTorch

1. Define Model

```
import torch
import torch.nn as nn

class NeuralNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 128)
        self.fc2 = nn.Linear(128, 10)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
        return x

model = NeuralNet()
```

2. Training Loop

```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters())

for epoch in range(epochs):
    for x_batch, y_batch in train_loader:
```

TensorFlow/Keras

1. Define Model

```
import tensorflow as tf
from tensorflow import keras

model = keras.Sequential([
    keras.layers.Dense(128, activation='relu',
                       input_shape=(784,)),
    keras.layers.Dense(10, activation='softmax')
])

model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
```

2. Training

```
# Simple training with fit()
history = model.fit(
    x_train, y_train,
    batch_size=32,
    epochs=10,
    validation_split=0.2
)
```

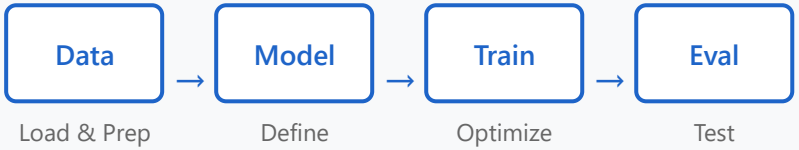
```
        # Forward pass
        outputs = model(x_batch)
        loss = criterion(outputs, y_batch)

        # Backward pass
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

```
# Evaluation
test_loss, test_acc = model.evaluate(
    x_test, y_test
)
```

Key Differences

Style	Pythonic	Keras API
Paradigm	Define-by-run	Sequential
Debugging	Native Python	TensorBoard
Deployment	TorchScript	TF Serving
Learning Curve	Moderate	Easy

Common Workflow



💡 Which to Choose?

PyTorch: Research, experimentation, custom operations, more control

TensorFlow: Production deployment, mobile/edge, high-level API, easier prototyping

Thank you

Ho-min Park

homin.park@ghent.ac.kr

powersimmani@gmail.com