

# **Neural Networks**

## **Foundation**

**From Perceptrons to Deep Learning**

**Nipun Batra | IIT Gandhinagar**

# Learning Goals

By the end of this lecture, you will:

Goal	What You'll Learn
Understand	Why we need neural networks
Build	Neurons, layers, and architectures
Choose	Activation functions (ReLU, sigmoid)
Train	Gradient descent and backpropagation
Implement	PyTorch fundamentals

# The Journey So Far

Week	Model	What It Does	Limitation
3	Linear Regression	Fit a line	Only linear patterns
3	Logistic Regression	Binary classification	Only linear boundaries
3	Decision Trees	If-then rules	Can overfit, unstable
4	Random Forest	Many trees	Hard to interpret
5	<b>Neural Networks</b>	<b>Learn any pattern!</b>	Need lots of data

# The Big Picture

Traditional ML:

Features —→ Model —→ Output

You design  
features

Deep Learning:

Raw Data —→ [Learn Features] —→ Output

Neural Network  
learns automatically!

Neural networks learn **both** features and the model!

# **Part 1: Why Neural Networks?**

## **The Limits of Linear Models**

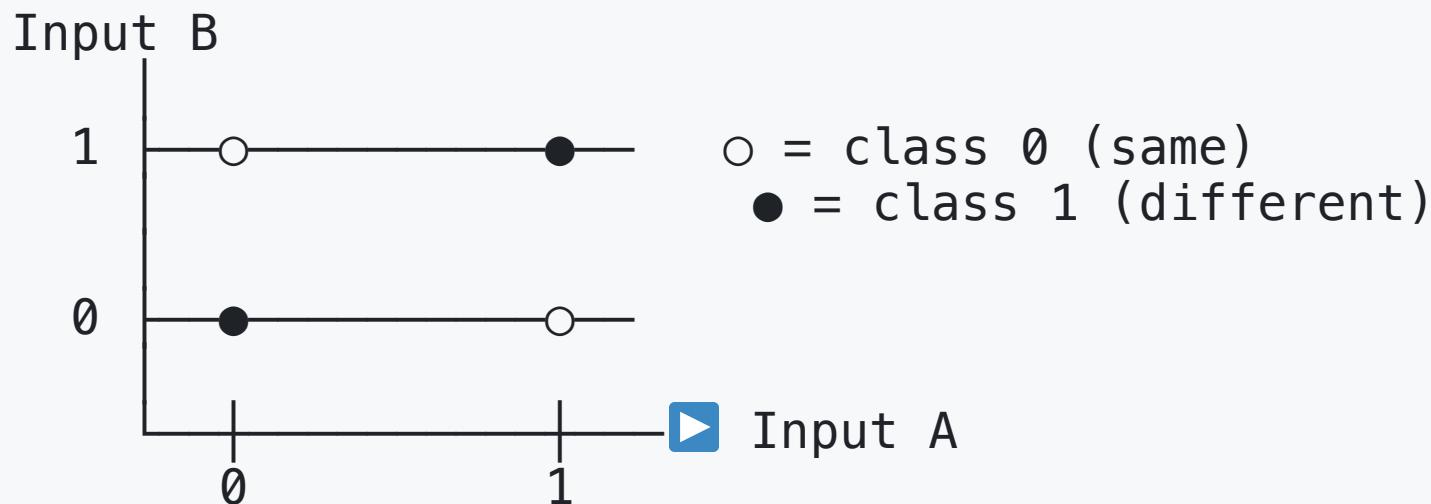
# A Problem Linear Models Can't Solve

The XOR Function:

Input A	Input B	Output (A XOR B)
0	0	0
0	1	1
1	0	1
1	1	0

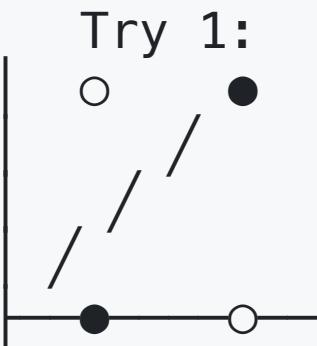
"Output is 1 if inputs are different"

# XOR: Visualized

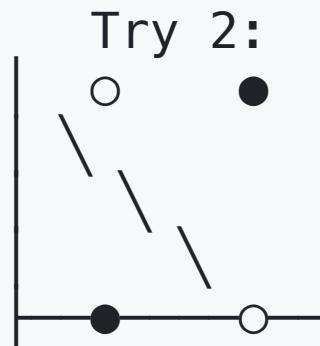


Can you draw ONE line to separate ● from ○?

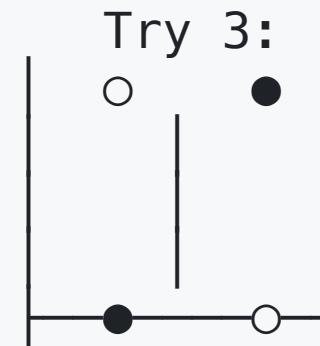
# No Line Works!



✗ Fails



✗ Fails



✗ Fails

XOR is NOT linearly separable!

# The Historical Impact

1969: Minsky & Papert proved perceptrons can't solve XOR.

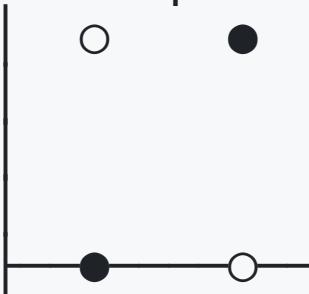
Before 1969	After 1969
"Neural networks will solve everything!"	"Neural networks are useless"
Lots of funding	"AI Winter" - funding cut
Active research	Field nearly died

**Solution discovered:** Multi-layer networks! (But took until 1986)

# The Solution: Multiple Layers

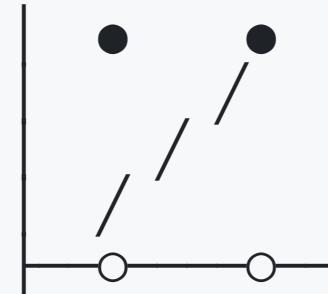
**Key insight:** Transform the space so XOR becomes separable!

Original Space:



Can't separate

After Hidden Layer:



TRANSFORM  
→

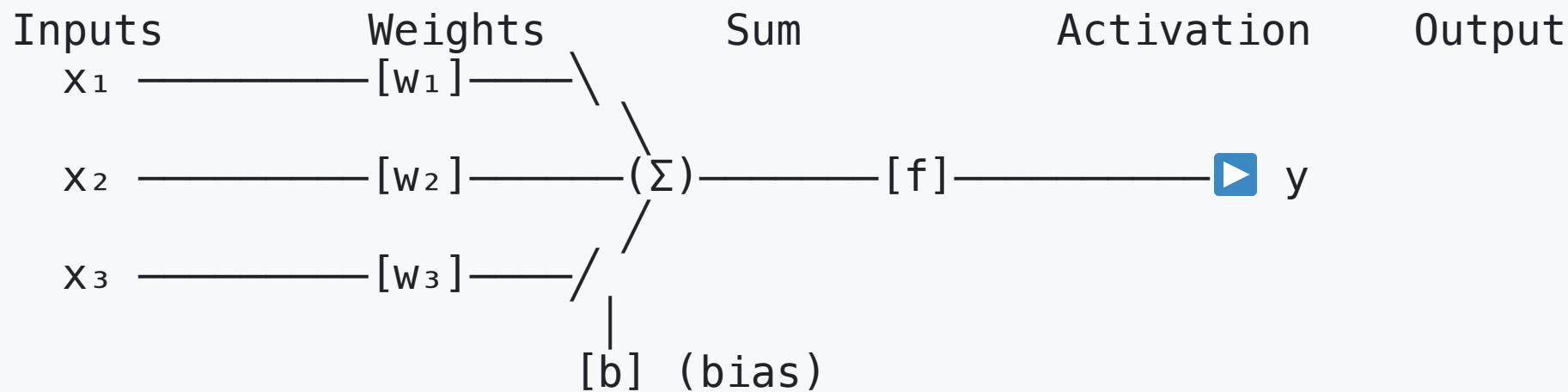
Now we CAN draw a line!

# Inspiration from Biology

The brain has ~86 billion neurons, each connected to ~10,000 others.

Biological Neuron	Artificial Neuron
Receives signals via dendrites	Receives inputs ( $x_1, x_2, \dots$ )
Processes in cell body	Computes weighted sum
Fires if threshold exceeded	Applies activation function
Sends signal via axon	Produces output
Connections strengthen with use	Weights updated during training

# The Artificial Neuron



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

$$y = f(z)$$

## **Part 2: The Perceptron**

**The Simplest Neural Network**

# The Perceptron (Frank Rosenblatt, 1958)

The first artificial neural network!

$$z = \sum_{i=1}^n w_i x_i + b = \mathbf{w}^T \mathbf{x} + b$$

$$y = \text{activation}(z)$$

Component	Symbol	Meaning
Inputs	$x_1, x_2, \dots$	Features
Weights	$w_1, w_2, \dots$	Importance of each input
Bias	$b$	Threshold adjustment
Activation	$f$	Non-linear function

# Perceptron: A Concrete Example

Task: Classify emails as spam (1) or not spam (0)

Input	Meaning	Value
$x_1$	Has "FREE"	1
$x_2$	Has attachment	0
$x_3$	From known contact	0

Weights (learned):  $w_1 = 2.0, w_2 = 0.5, w_3 = -1.5, b = -0.5$

$$z = 2.0(1) + 0.5(0) + (-1.5)(0) + (-0.5) = 1.5$$

If  $z > 0 \rightarrow$  Spam! ✓

# Perceptron in Python

```
import numpy as np

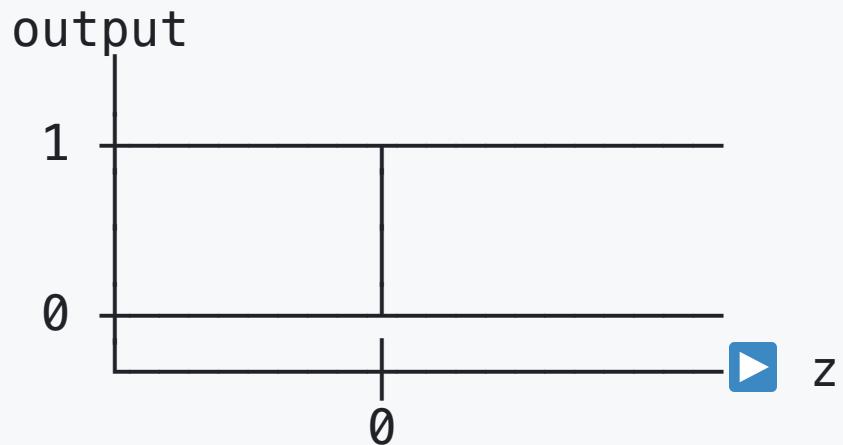
def perceptron(x, w, b):
    """Single perceptron with step activation."""
    z = np.dot(w, x) + b      # Weighted sum
    return 1 if z > 0 else 0  # Step activation

# Example
x = np.array([1, 0, 0])      # Has FREE, no attachment, unknown sender
w = np.array([2.0, 0.5, -1.5]) # Learned weights
b = -0.5                      # Learned bias

output = perceptron(x, w, b)  # → 1 (spam)
```

# The Step Activation Function

$$f(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases}$$



**Problem:** Not differentiable at  $z=0$  (can't use calculus!)

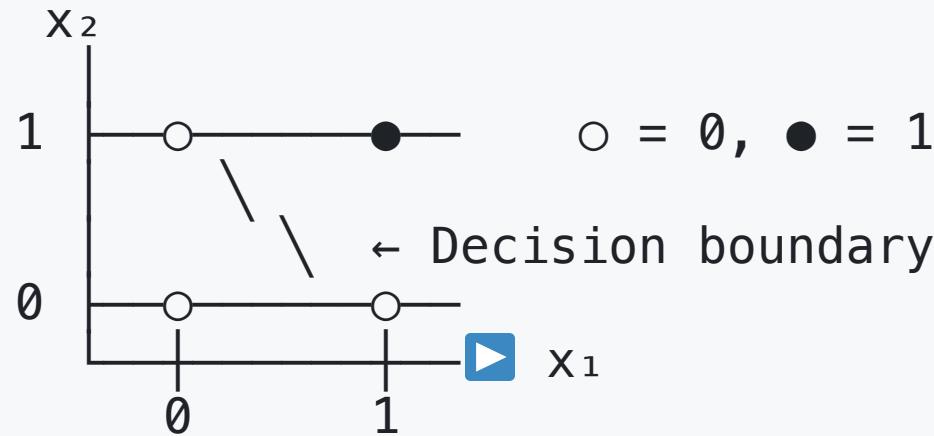
# What Can a Single Perceptron Learn?

Function	Can Learn?	Why
AND	<input checked="" type="checkbox"/> Yes	Linearly separable
OR	<input checked="" type="checkbox"/> Yes	Linearly separable
NOT	<input checked="" type="checkbox"/> Yes	Linearly separable
XOR	<input type="checkbox"/> No	Not linearly separable

Single perceptron = Linear decision boundary

# Perceptron Learning (AND Gate)

Goal: Output 1 only when BOTH inputs are 1



Perceptron CAN learn this line!

Weights:  $w_1=1$ ,  $w_2=1$ ,  $b=-1.5$

$z = x_1 + x_2 - 1.5 > 0$  only when both are 1

## **Part 3: Activation Functions**

**The Secret to Non-Linearity**

# Why Do We Need Activation Functions?

Without activation, stacking layers is useless!

$$\text{Layer 1: } h = W_1x + b_1$$

$$\text{Layer 2: } y = W_2h + b_2$$

$$\text{Combined: } y = W_2(W_1x + b_1) + b_2 = \underbrace{(W_2W_1)x}_{W'} + \underbrace{(W_2b_1 + b_2)}_{b'}$$

Still linear! Multiple layers collapse to one.

# The Magic of Non-Linearity

With activation:

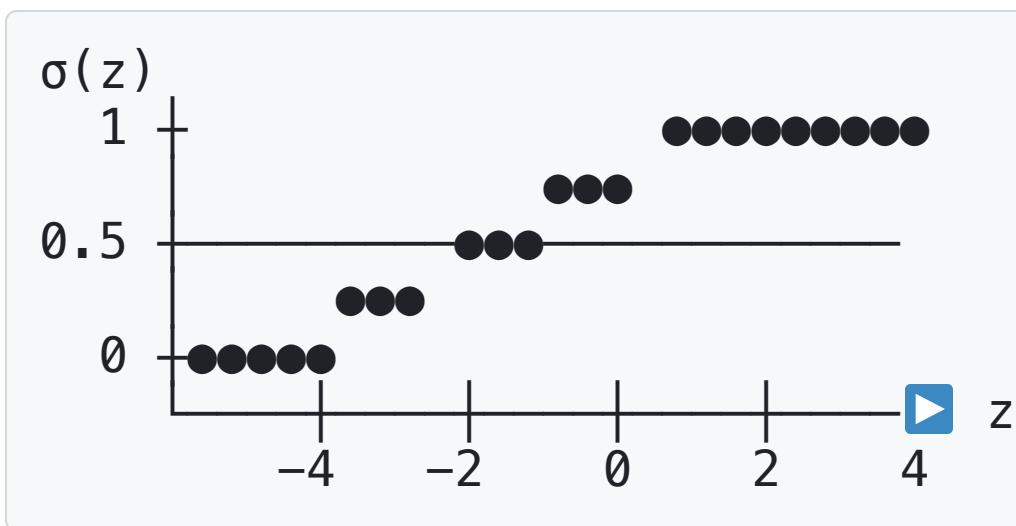
$$h = \text{ReLU}(W_1x + b_1)$$
$$y = W_2h + b_2$$

**Cannot simplify!** The ReLU breaks the linearity.

Activation functions allow networks to learn **curves**, not just lines!

# Sigmoid Activation

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



Property	Value
Output range	(0, 1)
At $z=0$	0.5
As $ z  \rightarrow \infty$	$\sigma(z) \rightarrow 1$

# Sigmoid: Pros and Cons

Pros	Cons
Output is probability-like (0-1)	<b>Vanishing gradient</b> (saturates)
Smooth, differentiable	Outputs not centered at 0
Historically important	Computationally expensive (exp)

**Vanishing gradient:** When  $z$  is very large or small, gradient  $\approx 0$

→ Network stops learning!

# Tanh Activation

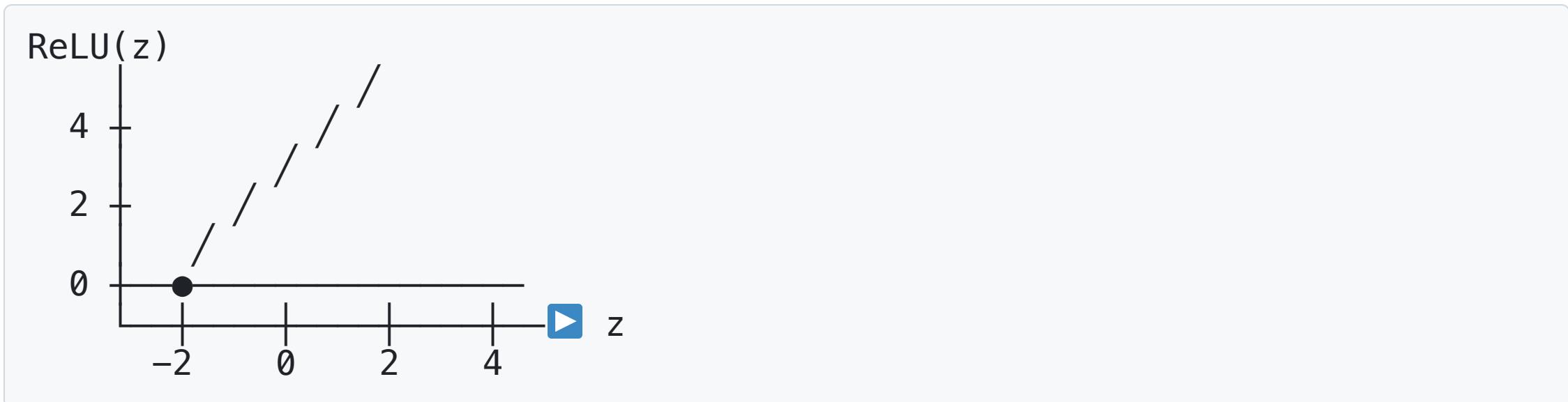
$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$



Property	Value
Output range	(-1, 1)
Zero-centered	✓ Yes

# ReLU: The Modern Choice

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



Property	Value
Output range	$[0, \infty)$
Gradient ( $z > 0$ )	1 (constant!)
Gradient ( $z < 0$ )	0

# ReLU: Pros and Cons

Pros	Cons
No vanishing gradient ( $z>0$ )	"Dead neurons" ( $z<0$ forever)
Very fast (just max)	Not zero-centered
Sparse activation (many zeros)	Unbounded output
Works extremely well in practice	

**Use ReLU by default!** It's the standard for hidden layers.

# ReLU Variants

Variant	Formula	Fixes
Leaky ReLU	$\max(0.01z, z)$	Dead neurons
ELU	$z \text{ if } z > 0, \text{ else } \alpha(e^z - 1)$	Smoothness
GELU	$z \cdot \Phi(z)$	Used in transformers
Swish	$z \cdot \sigma(z)$	Used in EfficientNet

```
# In PyTorch
F.relu(x)
F.leaky_relu(x, negative_slope=0.01)
F.gelu(x)
```

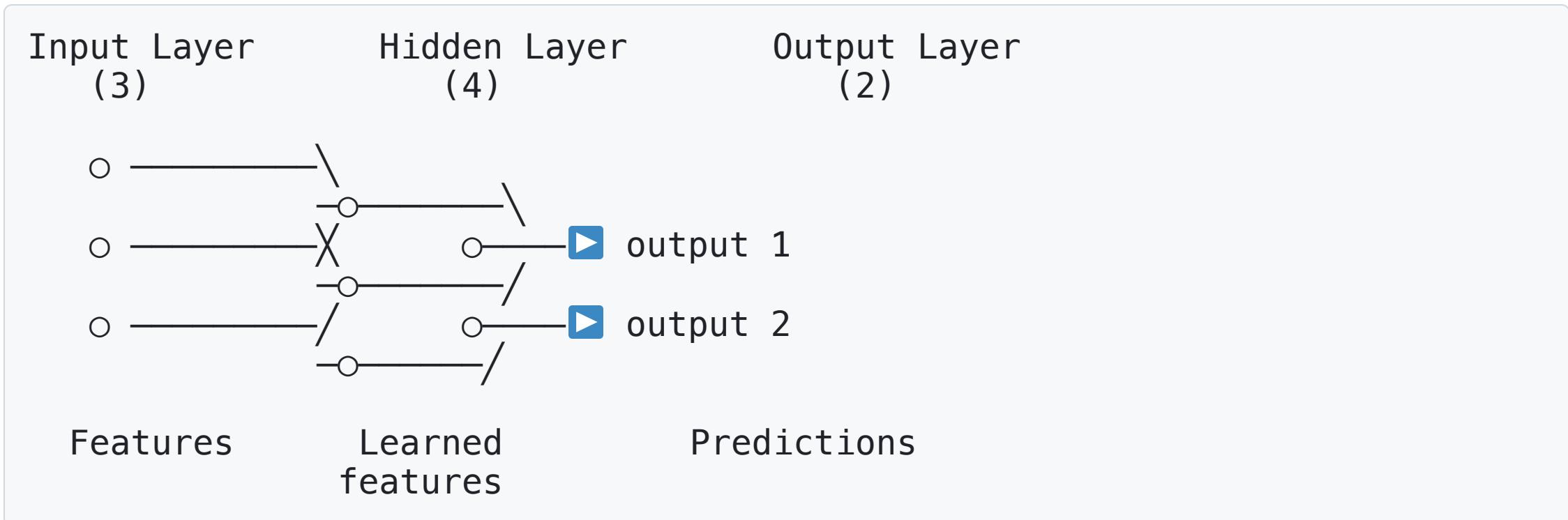
# Activation Function Summary

Function	Use Case	Range
ReLU	Hidden layers (default)	$[0, \infty)$
Sigmoid	Binary classification output	$(0, 1)$
Tanh	RNNs, specific architectures	$(-1, 1)$
Softmax	Multi-class output	$(0, 1)$ , sums to 1
Linear	Regression output	$(-\infty, \infty)$

## **Part 4: Multi-Layer Networks**

**Going Deep**

# The Multi-Layer Perceptron (MLP)



# How Information Flows

```
# Input: x (shape: 3)
# Hidden: 4 neurons
# Output: 2 classes

# Layer 1: Linear + Activation
z1 = W1 @ x + b1      # Shape: (4,) - Linear transform
h1 = relu(z1)           # Shape: (4,) - Non-linearity

# Layer 2: Linear + Activation
z2 = W2 @ h1 + b2      # Shape: (2,) - Linear transform
output = softmax(z2)    # Shape: (2,) - Probabilities
```

# Weight Matrix Dimensions

From	To	Weight Shape	Bias Shape
3 inputs	4 hidden	(4, 3)	(4,)
4 hidden	2 outputs	(2, 4)	(2,)

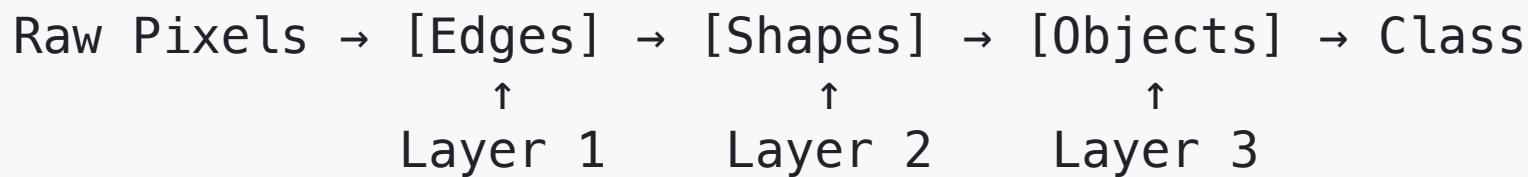
```
# Total parameters:  
# W1: 4 × 3 = 12  
# b1: 4  
# W2: 2 × 4 = 8  
# b2: 2  
# Total: 26 parameters to learn!
```

# Why Multiple Layers Work

**Layer 1:** Learn simple features (edges, colors)

**Layer 2:** Combine into patterns (shapes, textures)

**Layer 3:** Combine into objects (faces, cars)



**Each layer builds on the previous!**

# Universal Approximation Theorem

Theorem (Cybenko, 1989):

*A neural network with a single hidden layer can approximate any continuous function to arbitrary accuracy.*

Catch	Reality
May need infinitely many neurons	Deep > Wide in practice
Doesn't say how to find weights	Training is still hard
Doesn't guarantee generalization	More depth helps

# Deep Networks: Why More Layers?

Layers	What It Can Learn	Examples
1	Linear functions	Linear regression
2	Simple curves	XOR, simple patterns
3-10	Complex functions	MNIST, tabular data
10-100	Hierarchical patterns	ImageNet, NLP
100+	Very complex	GPT, state-of-the-art

# Solving XOR with 2 Layers

```
# Architecture: 2 → 2 → 1
# Input: [x1, x2]
# Hidden: 2 neurons with ReLU
# Output: 1 neuron with sigmoid

# Hidden layer transforms space:
# (0,0) → hidden representation
# (0,1) → hidden representation
# (1,0) → hidden representation
# (1,1) → hidden representation

# Output layer draws a line in the NEW space!
```

# Softmax: Multi-Class Output

**Problem:** Network outputs raw scores (logits). How to get probabilities?

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}}$$

Logits	Softmax
[2.0, 1.0, 0.1]	[0.66, 0.24, 0.10]

**Properties:**

- All outputs between 0 and 1
- All outputs sum to 1
- Largest logit → highest probability

# Common Architectures

Task	Input	Hidden	Output Activation
Binary Classification	Features	ReLU layers	Sigmoid (1 neuron)
Multi-class (10 classes)	Features	ReLU layers	Softmax (10 neurons)
Regression	Features	ReLU layers	Linear (1 neuron)

## **Part 5: Training Neural Networks**

**How Do Networks Learn?**

# The Training Process

## TRAINING LOOP

1. FORWARD PASS  
↓  
Compute prediction from input
2. COMPUTE LOSS  
↓  
How wrong are we?
3. BACKWARD PASS  
↓  
Which weights caused the error?
4. UPDATE WEIGHTS  
↓  
Adjust to reduce error
5. REPEAT  
Until loss is small enough

# Loss Functions: Measuring Error

Task	Loss Function	Formula
Regression	MSE	$\frac{1}{n} \sum (y - \hat{y})^2$
Binary Class.	Binary Cross-Entropy	$-(y \log \hat{y} + (1 - y) \log(1 - \hat{y}))$
Multi-class	Cross-Entropy	$-\sum_c y_c \log(\hat{y}_c)$

Lower loss = Better predictions!

# Cross-Entropy Loss Intuition

If true label is class 0:

Prediction for class 0	Loss
0.99 (confident, correct)	0.01
0.50 (unsure)	0.69
0.01 (confident, WRONG)	4.60

Severely punishes confident wrong predictions!

# Gradient Descent: The Key Idea

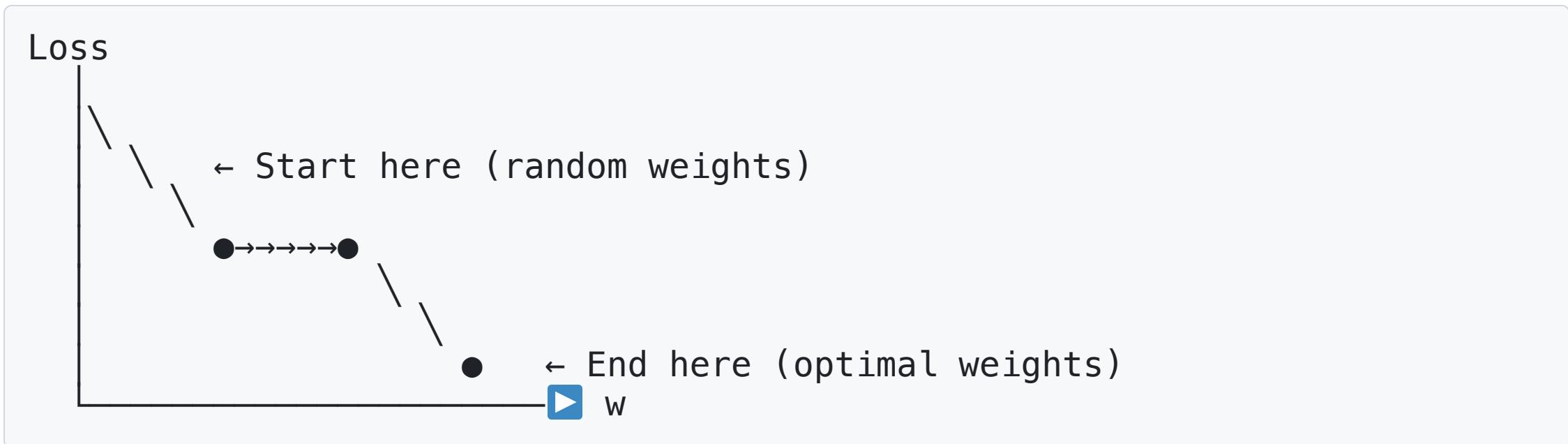
**Goal:** Find weights that minimize loss.

**Method:** Follow the slope downhill.

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

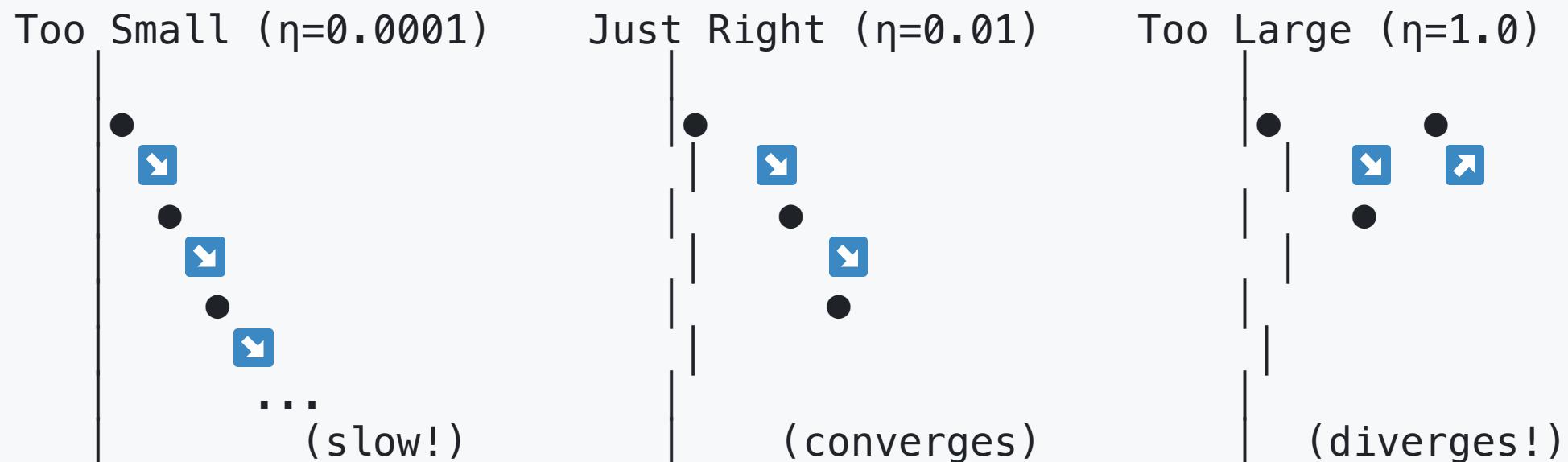
Symbol	Meaning
$w$	Weight to update
$\eta$	Learning rate (step size)
$\frac{\partial \text{Loss}}{\partial w}$	Gradient (slope)

# Gradient Descent Visualization



Each step moves toward the minimum!

# Learning Rate: Crucial Choice



# Learning Rate Guidelines

Learning Rate	Effect
0.1 - 1.0	Usually too large
0.01 - 0.001	Good starting point
0.0001 - 0.00001	Fine-tuning, large models

**Common practice:** Start high, decrease over time (learning rate schedule)

# Backpropagation: Computing Gradients

**Problem:** Network has millions of weights. How to compute all gradients?

**Solution:** Chain rule applied backward!

$$\frac{\partial \text{Loss}}{\partial w_1} = \frac{\partial \text{Loss}}{\partial y} \cdot \frac{\partial y}{\partial h} \cdot \frac{\partial h}{\partial w_1}$$

# Backprop: Step by Step

Forward pass:

$$x \xrightarrow{[W_1]} h \xrightarrow{[W_2]} y \xrightarrow{\text{Loss}} \text{Loss} = 2.5$$

$\downarrow$                              $\downarrow$   
 $h=3$                              $y=1.2$

Backward pass (compute gradients):

$$\frac{\partial L}{\partial W_1} \leftarrow \frac{\partial L}{\partial h} \leftarrow \frac{\partial L}{\partial y} \leftarrow \frac{\partial L}{\partial \text{Loss}} = 1$$

$|$                              $|$                              $|$   
 $0.4$                              $0.3$                              $0.5$

Then update:

$$W_1 = W_1 - \eta \times 0.4$$

$$W_2 = W_2 - \eta \times 0.3$$

# Stochastic Gradient Descent (SGD)

**Full Batch GD:** Use ALL data to compute gradient (slow!)

**SGD:** Use small random batch each step

Approach	Batch Size	Pros	Cons
Full Batch	All data	Stable	Very slow, memory
Mini-Batch	32-256	Fast, stable	Good default
Pure SGD	1	Very fast	Noisy, unstable

# Epochs and Batches

**Epoch:** One pass through entire dataset

**Batch:** Subset of data used for one gradient update

Dataset: 10,000 samples

Batch size: 100

Batches per epoch =  $10,000 / 100 = 100$  updates

5 epochs = 500 total updates

# Better Optimizers Than SGD

Optimizer	Key Idea	When to Use
SGD	Basic gradient descent	Simple, stable baseline
Momentum	Use past gradients	Faster convergence
Adam	Adaptive learning rate	<b>Default choice</b>
AdamW	Adam + weight decay	State-of-the-art

```
# In PyTorch:  
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

# Adam: Why It Works

**Adam combines:**

- 1. Momentum:** Remember past gradients (don't zigzag)
- 2. RMSProp:** Adapt learning rate per parameter

**Result:** Works well with minimal tuning!

```
# Adam is the default choice  
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

# **Part 6: PyTorch Basics**

**From Theory to Code**

# Why PyTorch?

Feature	Benefit
<b>Dynamic graphs</b>	Debug like normal Python
<b>Autograd</b>	Automatic gradient computation
<b>GPU support</b>	10-100x faster training
<b>Rich ecosystem</b>	Torchvision, HuggingFace, etc.
<b>Industry standard</b>	Tesla, Meta, OpenAI

```
import torch  
import torch.nn as nn
```

# Tensors: PyTorch's Arrays

```
import torch

# Create tensors
x = torch.tensor([1.0, 2.0, 3.0])                      # From list
y = torch.zeros(3, 4)                                    # 3×4 of zeros
z = torch.randn(3, 4)                                    # Random normal

# Operations (like numpy!)
a = x + 1                                              # Add scalar
b = x @ y                                              # Matrix multiply
c = x.sum()                                            # Reduce

# Move to GPU
x_gpu = x.to('cuda') # or x.cuda()
```

# Tensor Properties

```
x = torch.randn(3, 4, 5)

print(x.shape)      # torch.Size([3, 4, 5])
print(x.dtype)      # torch.float32
print(x.device)     # cpu (or cuda:0)
print(x.requires_grad) # False by default
```

# Automatic Differentiation

PyTorch computes gradients automatically!

```
# Create tensor that tracks gradients
x = torch.tensor([2.0], requires_grad=True)

# Forward pass: compute y = x2 + 3x + 1
y = x**2 + 3*x + 1

# Backward pass: compute dy/dx
y.backward()

# Gradient is stored in x.grad
print(x.grad) # tensor(7.)
# Because dy/dx = 2x + 3 = 2(2) + 3 = 7
```

# Building Networks: nn.Module

```
import torch.nn as nn

class SimpleNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 256)      # Input → Hidden
        self.fc2 = nn.Linear(256, 10)       # Hidden → Output

    def forward(self, x):
        x = torch.relu(self.fc1(x))       # Hidden + ReLU
        x = self.fc2(x)                  # Output (logits)
        return x

model = SimpleNetwork()
```

# nn.Linear: The Linear Layer

```
# nn.Linear(in_features, out_features)
layer = nn.Linear(3, 4)

# Has learnable parameters:
print(layer.weight.shape) # (4, 3)
print(layer.bias.shape)   # (4,)

# Forward pass:
x = torch.randn(2, 3)    # Batch of 2, 3 features each
y = layer(x)             # → shape (2, 4)
```

# nn.Sequential: Quick Networks

```
# Same network, less code:  
model = nn.Sequential(  
    nn.Linear(784, 256),  
    nn.ReLU(),  
    nn.Linear(256, 128),  
    nn.ReLU(),  
    nn.Linear(128, 10)  
)  
  
# Forward pass  
x = torch.randn(32, 784) # Batch of 32 images  
output = model(x)       # → shape (32, 10)
```

# The Training Loop

```
# 1. Create model, loss, optimizer
model = SimpleNetwork()
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

# 2. Training loop
for epoch in range(num_epochs):
    for batch_x, batch_y in dataloader:
        # Forward pass
        outputs = model(batch_x)
        loss = criterion(outputs, batch_y)

        # Backward pass
        optimizer.zero_grad()      # Clear old gradients
        loss.backward()            # Compute new gradients
        optimizer.step()          # Update weights
```

# Understanding the Training Loop

Line	What It Does
<code>outputs = model(batch_x)</code>	Forward pass: input → prediction
<code>loss = criterion(outputs, batch_y)</code>	Compute how wrong we are
<code>optimizer.zero_grad()</code>	Reset gradients to zero
<code>loss.backward()</code>	Backprop: compute all gradients
<code>optimizer.step()</code>	Update all weights

# Loading Data

```
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

# Define preprocessing
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,)))
])

# Load MNIST
train_data = datasets.MNIST('data', train=True,
                            download=True, transform=transform)
train_loader = DataLoader(train_data, batch_size=64, shuffle=True)

# Iterate
for images, labels in train_loader:
    print(images.shape) # (64, 1, 28, 28)
    print(labels.shape) # (64,)
```

# Complete MNIST Example

```
# Model
model = nn.Sequential(
    nn.Flatten(),                      # (batch, 1, 28, 28) → (batch, 784)
    nn.Linear(784, 256),
    nn.ReLU(),
    nn.Linear(256, 10)
)

# Training
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

for epoch in range(5):
    for images, labels in train_loader:
        outputs = model(images)
        loss = criterion(outputs, labels)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print(f"Epoch {epoch+1}, Loss: {loss.item():.4f}")
```

# Evaluation

```
model.eval() # Switch to evaluation mode (disables dropout, etc.)
correct = 0
total = 0

with torch.no_grad(): # Don't track gradients (saves memory)
    for images, labels in test_loader:
        outputs = model(images)
        _, predicted = outputs.max(1) # Get class with highest score
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

accuracy = 100 * correct / total
print(f"Test Accuracy: {accuracy:.2f}%") # ~97-98%
```

# Common Pitfalls

Mistake	Symptom	Fix
Forgot <code>zero_grad()</code>	Loss doesn't decrease	Add <code>optimizer.zero_grad()</code>
Wrong input shape	Shape error	Check tensor dimensions
Training with <code>no_grad()</code>	Loss stays constant	Remove <code>torch.no_grad()</code>
Not calling <code>model.eval()</code>	Bad test accuracy	Add <code>model.eval()</code>
Learning rate too high	Loss explodes (NaN)	Reduce learning rate

# Key Takeaways

1. **Perceptron** = Linear model + activation (can't solve XOR alone)
2. **Activation functions** add non-linearity (use **ReLU** by default)
3. **Multiple layers** can approximate any function
4. **Backpropagation** efficiently computes all gradients
5. **PyTorch training loop:**
  - Forward → Loss → zero\_grad → Backward → Step

# Summary: Neural Network Recipe

```
# 1. Define architecture
model = nn.Sequential(
    nn.Linear(input_size, hidden_size),
    nn.ReLU(),
    nn.Linear(hidden_size, output_size)
)

# 2. Define loss and optimizer
criterion = nn.CrossEntropyLoss()    # For classification
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

# 3. Training loop
for epoch in range(num_epochs):
    for x, y in dataloader:
        loss = criterion(model(x), y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

# Welcome to Deep Learning!

**Next:** Computer Vision - How Machines See

**Lab:** Build and train a neural network from scratch

*"What I cannot create, I do not understand."*

— Richard Feynman

**Questions?**