🧠 Neural Networks from Scratch — January 24, 2026

100xSchool Bootcamp 1.0

Week 02: Building & Training Your First Neural Network

PYTHON NUMPY MATPLOTLIB PYTORCH

📚 Class Overview

This session was a deep dive into **Neural Networks from first principles**. We moved from biological inspiration to mathematical implementation, manually coding every component—forward pass, loss calculation, backpropagation, and gradient descent—to solve the famous **XOR Problem**. This problem famously "killed" AI research in the 1970s, and understanding *why* reveals the fundamental importance of **hidden layers** and **non-linearity**. Finally, we compared our hand-rolled NumPy solution to a **PyTorch** implementation, seeing how modern frameworks automate the calculus.

> [!TIP] **Real-World Application**: High-Frequency Trading (HFT) firms still heavily rely on **traditional Machine Learning** (not deep learning). Why? Traditional ML models (like Gradient Boosted Trees) are faster, more interpretable, and excel at finding the most subtle patterns in structured, tabular data—exactly what financial tick data looks like.

💡 Machine Learning in One Sentence

> **Training is a simple, iterative loop of guessing and correcting.**

Step	Action	Description
01	Start with a guess	Initialize weights with random values.
02	Measure error	How far is the guess from the truth? (Loss function)
03	Adjust	Modify the guess to be slightly less wrong. (Backprop + Gradient Descent)
04	Repeat	Do this millions of times until error is minimized.

📖 Table of Contents

▶ **Click to Expand Navigation**

#	Topic	Summary
4️⃣	Activation Functions	Sigmoid, ReLU, Tanh — formulas & derivatives
5️⃣	Forward Pass	Matrix math: $z = Wx + b$, $a = \sigma(z)$
6️⃣	Loss Function	MSE and why we square the error
7️⃣	Backpropagation	Chain Rule, computing $\frac{\partial L}{\partial w}$
8️⃣	Gradient Descent	Update rule: $w \leftarrow w - \eta \nabla L$
9️⃣	Training Loop	The 4-step cycle with complete code
🔟	PyTorch Implementation	The same network in 10 lines
1️⃣1️⃣	Scaling Laws	From 20 parameters to 1.7 Trillion

🎯 Topics Covered

▼

1️⃣ The Hardware of Intelligence: The Neuron

Neural networks are inspired by the brain, but our digital "neurons" are pure math.

🧬 Biological Neuron vs. Artificial Neuron (Perceptron)

Component	Biological Role	Artificial Analog	Mathematical Form
Dendrites	Receive signals	Inputs ($x_1, x_2, ..., x_n$)	Input vector $\mathbf{x}$
Synapses	Signal strength	Weights ($w_1, w_2, ..., w_n$)	Weight vector $\mathbf{w}$
Cell Body (Soma)	Aggregates signals	Weighted Sum + Bias	$z = \sum w_i x_i + b = \mathbf{w} \cdot \mathbf{x} + b$
Axon Hillock	Fires if threshold is met	Activation Function	$a = f(z)$
Axon Terminal	Sends output	Output	$\hat{y}$

🔢 The Perceptron Equation

A single neuron computes: $$\hat{y} = f\left(\sum_{i=1}^{n} w_i x_i + b \right) = f(\mathbf{w}^T \mathbf{x} + b)$$

Where:

- $\mathbf{x}$: Input features

- $\mathbf{w}$: **Learnable weights** — These are the parameters that are *learned* during training. Each weight represents the **importance** of a particular input to the neuron's decision. Higher weight = more influence.
- b: Bias (shift the decision boundary)
- f: Activation function (introduces non-linearity)

> [!IMPORTANT] **Weights are learned, not programmed.** The entire goal of training is to find the optimal values for these weights such that the network's predictions match the ground truth. This is what separates Machine Learning from rule-based programming.

> [!NOTE] A single neuron without a non-linear activation function is just **Linear Regression**. It can only learn linear relationships (straight lines/planes).

▼

2 The Problem That "Almost Killed AI": XOR

In 1969, Marvin Minsky and Seymour Papert published *Perceptrons*, proving that a single-layer network **cannot** solve the XOR problem. This triggered the first "AI Winter".

⚙ The XOR Truth Table

x_1	x_2	y (XOR)
0	0	0
0	1	1
1	0	1
1	1	0

*XOR is "Exclusive OR": True if inputs are **different**, False if they're the **same**.*

🚫 Why a Single Neuron Fails: Linear Separability

- **AND Gate**: A line can separate (0,0), (0,1), (1,0) from (1,1). ✅
- **OR Gate**: A line can separate (0,0) from (0,1), (1,0), (1,1). ✅
- **XOR Gate**: No single line can separate (0,0), (1,1) from (0,1), (1,0). ❌

```
XOR Problem Visualization:

    x2
    ^
  1 |  ●(0,1)=1        ○(1,1)=0
    |
    |
  0 |  ○(0,0)=0        ●(1,0)=1
    +------------------------> x1
       0               1
```

```
● = Output 1,  ○ = Output 0
No single straight line can separate ● from ○
```

💡 The Solution: Hidden Layers

By adding a **hidden layer**, the network learns to *transform* the input space into a new representation where XOR *becomes* linearly separable.

```
graph LR
    subgraph Input Layer
        X1((x₁))
        X2((x₂))
    end
    subgraph Hidden Layer
        H1((h₁))
        H2((h₂))
        H3((h₃))
        H4((h₄))
    end
    subgraph Output
        Y((ŷ))
    end
    X1 --> H1 & H2 & H3 & H4
    X2 --> H1 & H2 & H3 & H4
    H1 & H2 & H3 & H4 --> Y
```

> [!IMPORTANT] **Depth matters.** The hidden layer acts as a "feature extractor", creating new representations of the input that make the problem solvable. This is the core idea behind **Deep Learning**.

▼

3 Network Architecture: Layers & Sizes

For XOR, we use a **2-4-1** architecture:

Layer	Size	Description
Input	2	Two features: x_1 and x_2
Hidden	4	Four neurons (arbitrary choice, but enough to learn XOR)
Output	1	One output: probability of $y=1$

📐 Parameter Count

Connection	Shape	# Parameters
Input → Hidden Weights	(2, 4)	8

Connection	Shape	# Parameters
Hidden Biases	(1, 4)	4
Hidden → Output Weights	(4, 1)	4
Output Bias	(1, 1)	1
Total		**17**

```
# Architecture Constants
INPUT_SIZE = 2
HIDDEN_SIZE = 4
OUTPUT_SIZE = 1

# Initialize weights (small random values)
import numpy as np
np.random.seed(42)

weights_input_hidden = np.random.randn(INPUT_SIZE, HIDDEN_SIZE) * 0.5  # (2, 4)
bias_hidden = np.zeros((1, HIDDEN_SIZE))                                # (1, 4)
weights_hidden_output = np.random.randn(HIDDEN_SIZE, OUTPUT_SIZE) * 0.5 # (4, 1)
bias_output = np.zeros((1, OUTPUT_SIZE))                                # (1, 1)
```

> [!TIP] We initialize weights with small random values (not zero!) to "break symmetry". If all weights start the same, all neurons learn the same thing—defeating the purpose of having multiple neurons.

▼

4 Activation Functions: The Source of Non-Linearity

Without activation functions, stacking layers is pointless: $W_2(W_1 \mathbf{x}) = W_{combined} \mathbf{x}$ is still just a linear transformation.

📊 The Activation Function Menu

Function	Formula	Derivative	Range	When to Use
Sigmoid	$\sigma(z) = \frac{1}{1 + e^{-z}}$	$\sigma(z)(1 - \sigma(z))$	(0, 1)	Output layer (binary classification)
Tanh	$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	$1 - \tanh^2(z)$	(-1, 1)	Hidden layers (zero-centered)
ReLU	$\max(0, z)$	$\begin{cases} 1 & z > 0 \\ 0 & z \le 0 \end{cases}$	[0, ∞)	Hidden layers (most common)
Leaky ReLU	$\max(0.01z, z)$	$\begin{cases} 1 & z > 0 \\ 0.01 & z \le 0 \end{cases}$	(-∞, ∞)	Avoids "dead neurons"

Function	Formula	Derivative	Range	When to Use
GELU	$z \cdot \Phi(z)$	*Complex*	(-∞, ∞)	Transformers (GPT, BERT)
Swish/SiLU	$z \cdot \sigma(z)$	$\sigma(z) + z \cdot \sigma(z)(1 - \sigma(z))$	(-∞, ∞)	Modern CNNs, EfficientNet

🚀 Modern Activations: GELU & Swish (Deep Dive)

These are the activations used in **state-of-the-art models** like GPT-4, BERT, and LLaMA.

GELU (Gaussian Error Linear Unit)

$$\text{GELU}(z) = z \cdot \Phi(z) = z \cdot \frac{1}{2}\left[1 + \text{erf}\left(\frac{z}{\sqrt{2}}\right)\right]$$

Approximation (faster to compute): $$\text{GELU}(z) \approx 0.5z\left(1 + \tanh\left[\sqrt{\frac{2}{\pi}}(z + 0.044715z^3)\right]\right)$$

Intuition: GELU is like a *smooth, probabilistic* version of ReLU. Instead of hard-cutting at 0, it weights inputs by their probability of being positive under a Gaussian distribution. Small negative values can still pass through (unlike ReLU), which helps preserve gradient flow.

```python
import numpy as np

def gelu(z):
    """GELU activation (exact using error function)."""
    return 0.5 * z * (1 + np.tanh(np.sqrt(2 / np.pi) * (z + 0.044715 * z**3)))

# Or using scipy for exact:
# from scipy.special import erf
# def gelu_exact(z):
#     return z * 0.5 * (1 + erf(z / np.sqrt(2)))
```

Swish / SiLU (Sigmoid Linear Unit)

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

Intuition: Swish is the input **multiplied by its own sigmoid**. This creates a smooth, non-monotonic curve (it can dip below zero for negative inputs before approaching zero). This non-monotonicity helps the model learn more complex patterns.

```python
def swish(z):
    """Swish/SiLU activation."""
    return z * (1 / (1 + np.exp(-z)))

# Or simply:
```

```python
def silu(z):
    return z * sigmoid(z)
```

Why GELU & Swish are Better for Deep Networks

Property	ReLU	GELU/Swish
Smoothness	Sharp corner at 0	Smooth everywhere
Dead Neurons	Common (gradients = 0 for z < 0)	Rare (small gradient even for z < 0)
Non-Monotonic	No	Yes (can dip negative)
Performance	Good baseline	Often 1-2% better accuracy
Computation	Very fast	Slightly slower (exponentials)

> [!TIP] **Rule of Thumb**: Use **ReLU** for simple/fast models. Use **GELU** for Transformers (NLP). Use **Swish/SiLU** for vision models (CNNs).

🧮 Sigmoid: Our Choice for XOR

We use **Sigmoid** because:

1. Output is between 0 and 1 (interpretable as probability).
2. It's smooth and differentiable everywhere (easy to backpropagate).

```python
def sigmoid(z):
    """The Sigmoid activation function."""
    return 1 / (1 + np.exp(-z))

def sigmoid_derivative(a):
    """Derivative of sigmoid, given the activation output 'a'."""
    # Note: Input is the ACTIVATED value, not the raw z.
    return a * (1 - a)
```

> [!CAUTION] **Vanishing Gradient Problem**: For large $|z|$, $\sigma'(z) \approx 0$. Gradients "vanish" during backprop, making deep networks very slow to train. This is why ReLU/GELU are preferred for hidden layers in practice.

▼

5️⃣ The Forward Pass: Making Predictions

The forward pass is the flow of data from input to output.

📝 Step-by-Step

1. **Input to Hidden (Linear)**: $z_h = X \cdot W_{ih} + b_h$
2. **Hidden Activation**: $a_h = \sigma(z_h)$
3. **Hidden to Output (Linear)**: $z_o = a_h \cdot W_{ho} + b_o$
4. **Output Activation**: $\hat{y} = \sigma(z_o)$

🐍 Complete Forward Pass Code

```python
def forward(X):
    """
    Perform a forward pass through the network.

    Args:
        X: Input data of shape (batch_size, 2)

    Returns:
        z_hidden: Pre-activation of hidden layer (batch_size, 4)
        a_hidden: Activation of hidden layer (batch_size, 4)
        z_output: Pre-activation of output layer (batch_size, 1)
        a_output: Final prediction (batch_size, 1)
    """
    # Layer 1: Input -> Hidden
    z_hidden = np.dot(X, weights_input_hidden) + bias_hidden
    a_hidden = sigmoid(z_hidden)

    # Layer 2: Hidden -> Output
    z_output = np.dot(a_hidden, weights_hidden_output) + bias_output
    a_output = sigmoid(z_output)

    return z_hidden, a_hidden, z_output, a_output
```

🔢 Numerical Example

For input $X = [1, 0]$ (expecting output $y = 1$):

```
Step 1: z_h = [1, 0] · W_ih + b_h
            = [1, 0] · [[0.31, -0.25, 0.49, -0.45],
                        [-0.06, 1.14, -0.04, 1.22]] + [0, 0, 0, 0]
            = [0.31, -0.25, 0.49, -0.45]

Step 2: a_h = sigmoid(z_h) = [0.58, 0.44, 0.62, 0.39]

Step 3: z_o = a_h · W_ho + b_o
            = [0.58, 0.44, 0.62, 0.39] · [[-0.43], [0.06], [-0.60], ...] + [0]
            = [-0.15]

Step 4: a_o = sigmoid(z_o) = [0.46]  ← Initial (wrong) prediction
```

▼

6️⃣ The Loss Function: Measuring "Wrongness"

How do we convert "how wrong is this prediction?" into a single number?

🧮 Mean Squared Error (MSE)

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

- y_i: True label (0 or 1)
- $\hat{y}_i$: Predicted probability
- n: Number of samples

Why square the error?

1. Makes all errors positive (no cancellation).
2. Penalizes large errors more heavily (quadratic penalty).
3. Smooth and differentiable (easy to optimize).

```python
def compute_loss(y_true, y_pred):
    """
    Compute Mean Squared Error loss.

    Args:
        y_true: Ground truth labels (batch_size, 1)
        y_pred: Predicted probabilities (batch_size, 1)

    Returns:
        Scalar loss value
    """
    return np.mean((y_true - y_pred) ** 2)
```

📊 Loss Landscape Intuition

Imagine the loss as a terrain. Each point represents a particular set of weights. Our goal: **find the lowest valley** (global minimum).

```
         Loss
           ^
           |     /\      /\
           |    /  \    /  \
           |   /    \  /    \
           |  /      \/      \
           | /   Global       \
           |/    Minimum       \
           +------------------------> Weight
```

> [!NOTE] **Binary Cross-Entropy** is technically a better loss for classification (it's what PyTorch uses), but MSE works fine for our XOR example and is easier to understand.

▼

7️⃣ Backpropagation: The Heart of Learning

Backpropagation answers: **"How much did each weight contribute to the error?"**

We use the **Chain Rule** from calculus to compute this, working backwards from the output.

🔗 The Chain Rule

If L depends on w through intermediate variables: $$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w}$$

"Blame flows backwards through the network."

📐 Derivation for Our Network

Step 1: Output Layer Gradient We need $\frac{\partial L}{\partial W_{ho}}$ and $\frac{\partial L}{\partial b_o}$.

Start with the loss: $$L = \frac{1}{n} \sum (y - \hat{y})^2$$

Using chain rule: $$\frac{\partial L}{\partial \hat{y}} = -\frac{2}{n}(y - \hat{y}) = \frac{2}{n}(\hat{y} - y)$$

Then through sigmoid: $$\frac{\partial L}{\partial z_o} = \frac{\partial L}{\partial \hat{y}} \cdot \sigma'(z_o) = \frac{2}{n}(\hat{y} - y) \cdot \hat{y}(1 - \hat{y})$$

We define the **output error signal**: $\delta_o = (\hat{y} - y) \cdot \hat{y}(1 - \hat{y})$

Then: $$\frac{\partial L}{\partial W_{ho}} = a_h^T \cdot \delta_o$$ $$\frac{\partial L}{\partial b_o} = \sum \delta_o$$

Step 2: Hidden Layer Gradient The error "flows back" through the output weights: $$\delta_h = (\delta_o \cdot W_{ho}^T) \cdot a_h(1 - a_h)$$

Then: $$\frac{\partial L}{\partial W_{ih}} = X^T \cdot \delta_h$$ $$\frac{\partial L}{\partial b_h} = \sum \delta_h$$

🔙 Complete Backward Pass Code

```
def backward(X, y, z_hidden, a_hidden, z_output, a_output, learning_rate):
    """
    Perform backpropagation and update weights.

    Args:
        X: Input data
        y: True labels
        z_hidden: Pre-activation of hidden layer
        a_hidden: Activation of hidden layer
        z_output: Pre-activation of output layer
        a_output: Final prediction
        learning_rate: Step size for gradient descent
    """
    global weights_input_hidden, bias_hidden
    global weights_hidden_output, bias_output

    batch_size = X.shape[0]

    # ====== OUTPUT LAYER GRADIENTS ======
    # Error signal at output
    error_output = a_output - y  # (batch, 1)
    delta_output = error_output * sigmoid_derivative(a_output)  # (batch, 1)

    # Gradients for output weights and bias
    grad_weights_ho = np.dot(a_hidden.T, delta_output) / batch_size  # (4, 1)
    grad_bias_o = np.sum(delta_output, axis=0, keepdims=True) / batch_size  # (1,
1)

    # ====== HIDDEN LAYER GRADIENTS ======
    # Backpropagate error to hidden layer
    error_hidden = np.dot(delta_output, weights_hidden_output.T)  # (batch, 4)
    delta_hidden = error_hidden * sigmoid_derivative(a_hidden)  # (batch, 4)

    # Gradients for hidden weights and bias
    grad_weights_ih = np.dot(X.T, delta_hidden) / batch_size  # (2, 4)
    grad_bias_h = np.sum(delta_hidden, axis=0, keepdims=True) / batch_size  # (1,
4)

    # ====== UPDATE WEIGHTS (Gradient Descent) ======
    weights_hidden_output -= learning_rate * grad_weights_ho
    bias_output -= learning_rate * grad_bias_o
    weights_input_hidden -= learning_rate * grad_weights_ih
    bias_hidden -= learning_rate * grad_bias_h
```

> [!IMPORTANT] **Key Insight**: The gradient for the hidden layer depends on the output layer's delta. This is why we call it "back" propagation—we compute gradients from output to input.

▼

8 Gradient Descent: Walking Downhill

Once we have the gradients, we **update the weights** to reduce the loss.

⛰ The Update Rule

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

Where $\eta$ (eta) is the **learning rate** — the most critical hyperparameter in machine learning.

"Move in the opposite direction of the gradient (steepest descent)."

Learning Rate: A Deep Dive

The learning rate $\eta$ controls **how big of a step** we take when updating weights. It's the single most important hyperparameter to tune.

The Mountain Descent Analogy

Imagine you're blindfolded on a mountain, trying to find the lowest valley:

- **Gradient**: Tells you which direction is downhill.
- **Learning Rate**: How big of a step you take in that direction.

$\eta$ Value	Analogy	Result
Too Small (0.0001)	Taking tiny baby steps	You'll eventually get there, but it takes forever. Might get stuck in a small ditch (local minimum).
Too Large (100)	Jumping wildly	You overshoot the valley, bounce off the other side, and fly off the mountain (loss = NaN).
Just Right (0.01 - 1.0)	Confident strides	Steady descent to the bottom.

Mathematical Intuition

The update rule again: $$w_{t+1} = w_t - \eta \cdot \nabla L(w_t)$$

- If $\eta$ is tiny, $w_{t+1} \approx w_t$ — almost no change per step.
- If $\eta$ is huge, the weight change is massive — could completely bypass the minimum.

```
# Experiment: Learning Rate Effects
learning_rate_low = 0.0001  # Takes 100,000+ epochs
learning_rate_high = 100.0  # Loss: 0.25 → 0.50 → 2.3 → NaN 💥
learning_rate_good = 2.0    # Converges in ~5,000 epochs
```

Visual: The Loss Curve for Different Learning Rates

```
    Loss
     ^
     |   η=100 (explodes)        ___/
     |        /\/\/\            /
     |                         /
     |   η=0.0001 (too slow)   /
     |   ---------------------|-----
     |         η=2.0 (just right)
     |        \_______________
     +--------------------------------> Epochs
```

Advanced: Adaptive Learning Rates

In practice, we rarely use a fixed learning rate. Modern optimizers **adapt** $\eta$ for each weight:

Optimizer	Key Idea	When to Use
SGD	Fixed learning rate	Baseline, simple problems
SGD + Momentum	Accumulates velocity in consistent directions	Faster convergence
Adam	Adapts $\eta$ per parameter based on gradient history	Default choice for most problems
AdamW	Adam with decoupled weight decay	Transformers, LLMs

```
# PyTorch Optimizers
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)       # Basic
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)     # Most common
optimizer = torch.optim.AdamW(model.parameters(), lr=3e-4)     # For Transformers
```

Learning Rate Schedules

Start with a higher $\eta$ (explore quickly) and decrease it over time (fine-tune):

Schedule	Description
Step Decay	Reduce by factor every N epochs (e.g., $\eta \rightarrow \eta/10$ every 30 epochs)
Cosine Annealing	Smooth decay following cosine curve
Warmup + Decay	Start small, ramp up, then decay (used in Transformers)

> [!IMPORTANT] **If your loss is NaN or Inf**: Your learning rate is almost certainly too high. Divide by 10 and try again.
>
> **If your loss plateaus immediately**: Your learning rate might be too low. Multiply by 10.

▼

9 The Training Loop: Putting It All Together

Training is just repeating these 4 steps thousands of times:

```
graph LR
    A[1. Forward Pass] --> B[2. Compute Loss]
    B --> C[3. Backward Pass]
    C --> D[4. Update Weights]
    D --> A
```

Complete Training Code

```
# XOR Dataset
X = np.array([[0, 0],
              [0, 1],
              [1, 0],
              [1, 1]])

y = np.array([[0],
              [1],
              [1],
              [0]])

# Hyperparameters
learning_rate = 2.0
epochs = 10000

# Training History
loss_history = []

# ====== THE TRAINING LOOP ======
for epoch in range(epochs):
    # 1. Forward Pass
    z_h, a_h, z_o, prediction = forward(X)

    # 2. Compute Loss
    loss = compute_loss(y, prediction)
    loss_history.append(loss)

    # 3. Backward Pass + 4. Update Weights
    backward(X, y, z_h, a_h, z_o, prediction, learning_rate)

    # Logging
    if epoch % 1000 == 0:
        print(f"Epoch {epoch:5d} | Loss: {loss:.6f}")

# ====== FINAL RESULTS ======
print("\n" + "="*50)
print("Final Predictions:")
for i in range(len(X)):
    pred = prediction[i][0]
    print(f"  {X[i]} → {pred:.4f} (rounded: {round(pred)}) | Target: {y[i][0]}")
```

Expected Output

```
Epoch     0 | Loss: 0.278543
Epoch  1000 | Loss: 0.042381
Epoch  2000 | Loss: 0.009234
...
Epoch  9000 | Loss: 0.000102

==================================================
Final Predictions:
  [0 0] → 0.0124 (rounded: 0) | Target: 0  ✅
  [0 1] → 0.9898 (rounded: 1) | Target: 1  ✅
  [1 0] → 0.9897 (rounded: 1) | Target: 1  ✅
  [1 1] → 0.0098 (rounded: 0) | Target: 0  ✅
```

🎉 **The network learned XOR from random weights!**

▼

🔟 PyTorch: Automating the Calculus

We manually computed every gradient. In practice, **PyTorch does this automatically** using its `autograd` engine.

🔥 The Same Network in PyTorch

```python
import torch
import torch.nn as nn
import torch.optim as optim

# Define the network
class XORNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden = nn.Linear(2, 4)
        self.output = nn.Linear(4, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.sigmoid(self.hidden(x))
        x = self.sigmoid(self.output(x))
        return x

# Create model, loss function, and optimizer
model = XORNet()
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=2.0)

# Data (as PyTorch tensors)
X = torch.tensor([[0., 0.], [0., 1.], [1., 0.], [1., 1.]])
y = torch.tensor([[0.], [1.], [1.], [0.]])

# Training Loop
for epoch in range(10000):
    # Forward
    predictions = model(X)
    loss = criterion(predictions, y)

    # Backward (PyTorch computes ALL gradients automatically!)
    optimizer.zero_grad()  # Reset gradients from previous step
    loss.backward()        # Compute gradients
    optimizer.step()       # Update weights

# Test
with torch.no_grad():
    print(model(X))
```

⚡ PyTorch vs NumPy Comparison

Aspect	NumPy (Manual)	PyTorch (Automatic)
Forward Pass	Wrote `forward()` function	Defined in `forward()` method
Gradients	Derived and coded every gradient formula	`loss.backward()` does it all
Weight Updates	Manual: `w -= lr * grad`	`optimizer.step()`
GPU Support	No	Yes (just add `.cuda()`)
Lines of Code	~50	~20

> [!TIP] **Why learn the manual way first?** Understanding backpropagation helps you debug, design custom architectures, and understand why things fail. PyTorch is a tool, not black magic.

▼

1️⃣1️⃣ From XOR to GPT: Scaling Up

We solved XOR with ~20 parameters. How do LLMs use the same principles?

📊 The Scale Comparison

Model	Parameters	Layers	Training Time
XOR (Ours)	~20	2	< 1 second
AlexNet (2012)	60 Million	8	Days on 2 GPUs
GPT-2 (2019)	1.5 Billion	48	Weeks on 32 GPUs
GPT-3 (2020)	175 Billion	96	Months on 10,000+ GPUs
GPT-4 (2023)	~1.7 Trillion*	100+*	Months on 25,000+ GPUs

Estimated, not officially disclosed.

🚀 What Changes at Scale?

Component	XOR	Large Models
Activation	Sigmoid	GELU, SiLU, Swish
Optimizer	SGD	Adam, AdamW
Architecture	Dense (MLP)	Transformers (Attention)
Data	4 examples	Trillions of tokens
Hardware	CPU	Thousands of GPUs/TPUs

> [!IMPORTANT] **The core loop is identical**:
>
> 1. Forward Pass
> 2. Compute Loss
> 3. Backpropagation
> 4. Gradient Descent
>
> *What produces intelligence at scale is just... more of the same, done better.*

🔑 Key Takeaways

#	Concept	One-Liner
1	**XOR Problem**	Single neurons can't solve non-linearly separable problems. Hidden layers are essential.
2	**Activation Functions**	They introduce non-linearity; without them, deep networks are just linear regression.
3	**Forward Pass**	$z = Wx + b$, $a = f(z)$ — Matrix multiplication followed by activation.
4	**Loss Function**	MSE = $\frac{1}{n}\sum(y - \hat{y})^2$ — A single number representing "wrongness".
5	**Backpropagation**	Chain rule + moving backwards. Computes $\frac{\partial L}{\partial w}$ for every weight.
6	**Gradient Descent**	$w \leftarrow w - \eta \nabla L$ — Walk downhill to find the minimum loss.
7	**Learning Rate**	Too low = slow; too high = explodes. Find the Goldilocks zone.
8	**Training Loop**	Forward → Loss → Backward → Step → Repeat 10,000 times.
9	**PyTorch**	`loss.backward()` does all the calculus automatically.
10	**Scaling Laws**	More data + more parameters + more compute = emergent intelligence.

🗂 Folder Structure

```
(Week 02) 24-01-2026/
├── 🗂 Codes/
```

```
│   └── class2.ipynb        # XOR implementation (NumPy + PyTorch)
├── 📂 Notes_&_Screenshots/
│   └── [Lecture slides and diagrams]
└── 📄 README.md              # You are here!
```

🔗 Resources

📝 Class Materials

- **Class 2 Notebook (Colab)**

📚 Further Reading

- **Neural Networks & Deep Learning (Free Book)**
- **3Blue1Brown: Neural Networks (YouTube)**
- **Backpropagation Calculus (3B1B)**

🛠️ Libraries & Tools

- **PyTorch Documentation**
- **NumPy Documentation**
- **TensorFlow Playground**
- **Loss Landscape Visualizer**

🧠 Deep Dives

- **Why Momentum Really Works**
- **Visualizing Neural Networks**

📅 Class Date: **January 24, 2026**

Made with 💜 by Akshat Shethia during 100xSchool Bootcamp 1.0

Next Up: Transformers & The Attention Mechanism 🎯