Neural Networks Foundation

From Perceptrons to Deep Learning

Nipun Batra | IIT Gandhinagar

The Journey So Far

Week	Model	Limitation
3	Linear/Logistic Regression	Only linear patterns
3	Decision Trees	Can overfit
4	Random Forest	Hard to interpret
5	**Neural Networks**	**Can learn anything!**

Today's Agenda

1. **From Linear to Neural** - Why do we need NNs?
2. **The Perceptron** - Simplest neural network
3. **Activation Functions** - Adding non-linearity
4. **Multi-Layer Networks** - Going deep
5. **Training** - Backpropagation & Gradient Descent
6. **PyTorch Basics** - Implementing NNs

Part 1: Why Neural Networks?

Limitations of Linear Models

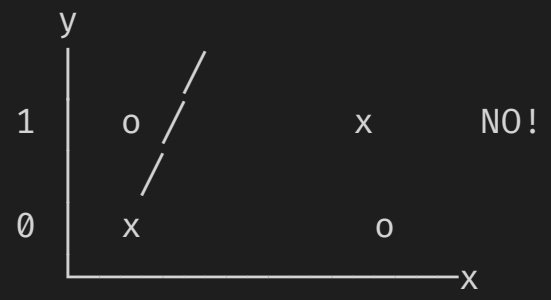
The XOR Problem

Can we classify these points with a line?

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

Linear Models Fail!

No single line can separate the classes:



XOR is not linearly separable!

The Solution: Non-Linearity

What if we could:

1. Transform the data into a new space
2. Draw a line there
3. Transform back

Neural networks do this automatically!

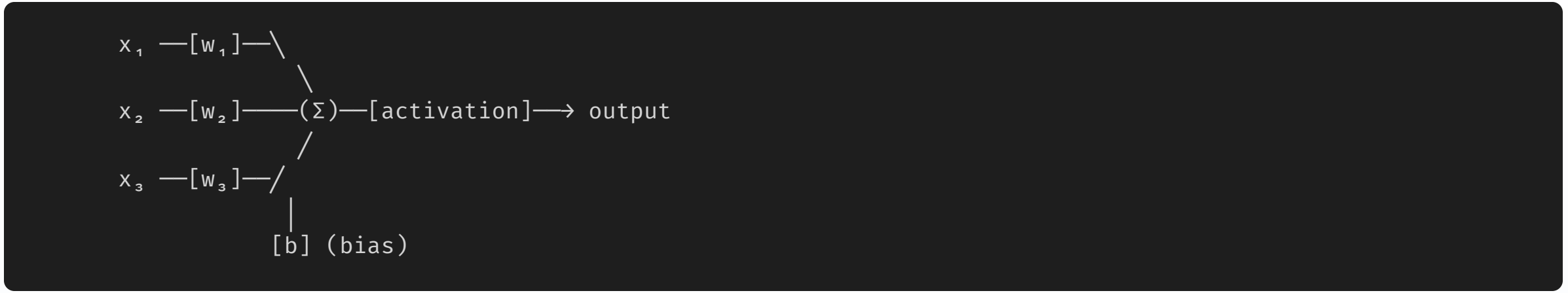
Inspiration: The Brain

Biological Neuron	Artificial Neuron
Receives signals via dendrites	Receives inputs
Processes in cell body	Computes weighted sum
Fires if threshold exceeded	Applies activation function
Sends signal via axon	Produces output

Part 2: The Perceptron

Simplest Neural Network

The Perceptron (1958)



$$z = w_1 x_1 + w_2 x_2 + w_3 x_3 + b$$

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

Perceptron: Just Linear Regression!

$$z = \mathbf{w}^T \mathbf{x} + b = \sum_i w_i x_i + b$$

If activation is identity: **Linear Regression**

If activation is sigmoid: **Logistic Regression**

Step Activation (Original)

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

Perceptron in Code

```python
import numpy as np

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

# Example
x = np.array([1, 0])
w = np.array([0.5, -0.5])
b = 0.1

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

Perceptron Limitation

Single perceptron = linear decision boundary

It **cannot** solve XOR!

This was proven in 1969 (Minsky & Papert) and caused the first "AI Winter."

Part 3: Activation Functions

Adding Non-Linearity

Why Activation Functions?

Without activation:

$$\text{Layer 2}(\text{Layer 1}(x)) = W_2(W_1 x + b_1) + b_2 = W'x + b'$$

Still linear! Multiple layers collapse to one.

> Activation functions allow networks to learn non-linear patterns.

Sigmoid

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

Squashes output to (0, 1)

Tanh

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

Output range: (-1, 1), centered at zero

ReLU (Rectified Linear Unit)

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

Most popular today! Simple and effective.

Activation Comparison

Function	Range	Pros	Cons
Sigmoid	(0, 1)	Probabilistic	Vanishing gradient
Tanh	(-1, 1)	Zero-centered	Vanishing gradient
ReLU	[0, ∞)	Fast, effective	"Dead neurons"

```
import torch.nn.functional as F

F.sigmoid(x)
F.tanh(x)
F.relu(x)
```

Part 4: Multi-Layer Networks

Going Deep

The Multi-Layer Perceptron (MLP)

Hidden layers transform the data!

Layer by Layer

```
# Input: x (3 features)
# Hidden: 4 neurons
# Output: 1 value

z1 = W1 @ x + b1    # (4,)  Linear
h1 = relu(z1)        # (4,)  Non-linear

z2 = W2 @ h1 + b2   # (1,)  Linear
output = sigmoid(z2) # (1,)  Final activation
```

Why "Deep"?

Layers	What It Can Learn
1 (linear)	Lines, planes
2	Any continuous function*
3+	Hierarchical features

Universal Approximation Theorem (Cybenko, 1989)

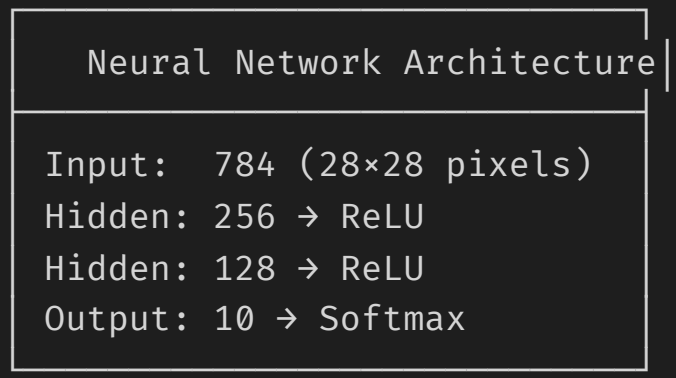
XOR: Now Solvable!

With a hidden layer, we can solve XOR:

```
# Hidden layer transforms:
# (0,0) → region A
# (1,1) → region A
# (0,1) → region B
# (1,0) → region B

# Then output layer separates A from B!
```

Network Architecture



Softmax: Multi-class Output

For classification with K classes:

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

Properties:

- All outputs sum to 1
- Each output is a probability
- Largest input gets highest probability

Part 5: Training Neural Networks

How Do We Learn the Weights?

The Training Process

```
1. Forward pass:  Compute prediction from input
2. Loss:          Compare prediction to ground truth
3. Backward pass: Compute gradients
4. Update:        Adjust weights to reduce loss
5. Repeat!
```

Loss Functions

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

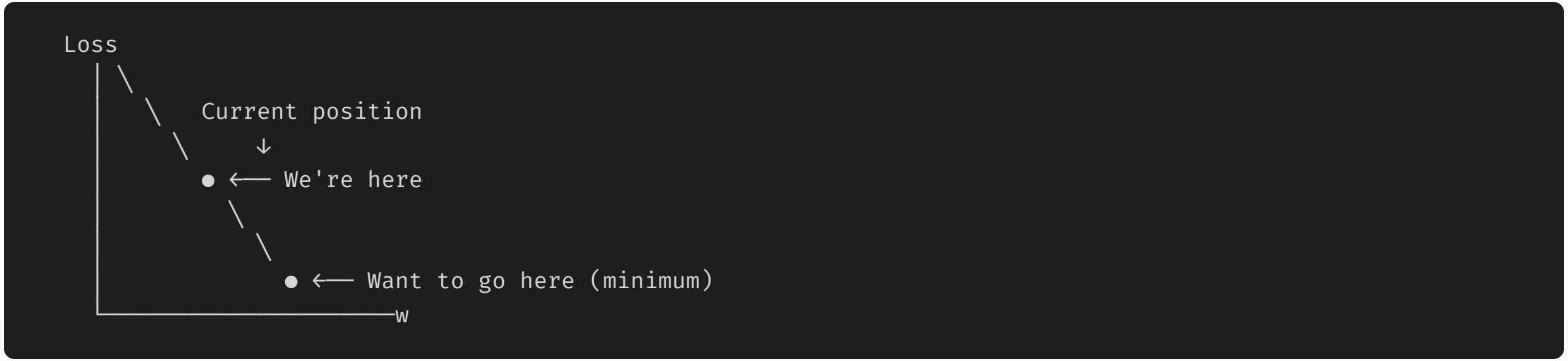
Gradient Descent

Idea: Move weights in direction that reduces loss.

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

Symbol	Meaning
η	Learning rate (step size)
$\frac{\partial \text{Loss}}{\partial w}$	Gradient (slope)

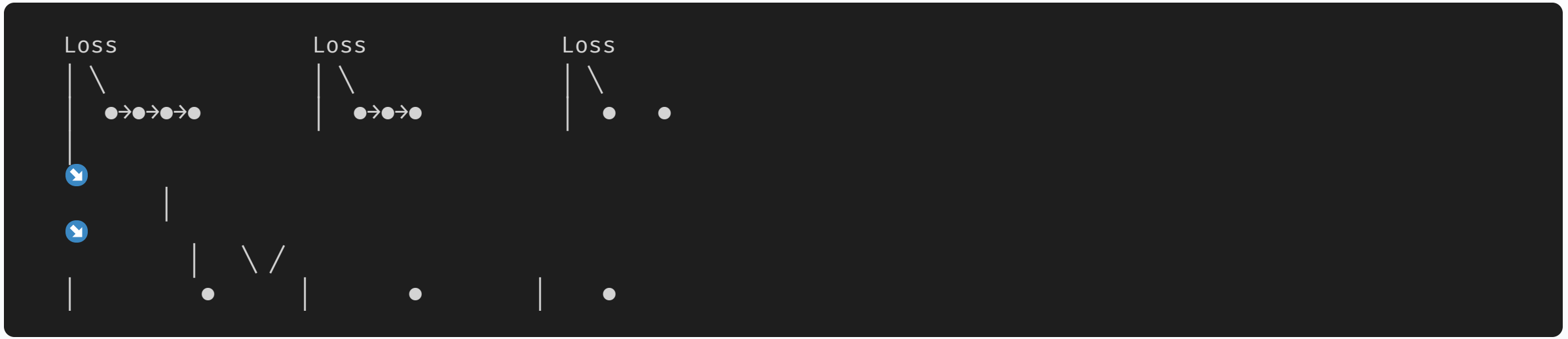
Gradient Descent Intuition



Follow the slope downhill!

Learning Rate

Too small	Just right	Too large
Very slow	Converges	Overshoots
Gets stuck	Finds minimum	Diverges



Backpropagation

The key algorithm: Efficiently compute gradients for all weights.

Chain rule:

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

Propagate gradients **backward** through the network!

Backprop: The Chain Rule

Stochastic Gradient Descent (SGD)

Instead of using ALL data:

1. Shuffle data
2. Take small **batch** (e.g., 32 samples)
3. Compute gradient on batch
4. Update weights
5. Repeat for all batches (= 1 epoch)

SGD Variants

Optimizer	Improvement
SGD	Basic gradient descent
SGD + Momentum	Accumulate past gradients
Adam	Adaptive learning rate per parameter
AdamW	Adam + weight decay

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

Part 6: PyTorch Basics

From Theory to Code

Why PyTorch?

Feature	Benefit
Dynamic graphs	Debug like Python
GPU support	Fast training
Rich ecosystem	Many pre-trained models
Industry standard	Facebook, Tesla, OpenAI

Tensors: PyTorch Arrays

```python
import torch

# Create tensors
x = torch.tensor([1.0, 2.0, 3.0])
W = torch.randn(3, 2)  # Random 3×2 matrix

# Operations
y = x @ W + 0.1        # Matrix multiply + bias
z = torch.relu(y)      # Activation

print(z.shape)  # torch.Size([2])
```

Automatic Differentiation

```
# Tell PyTorch to track gradients
x = torch.tensor([2.0], requires_grad=True)

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

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

print(x.grad)  # tensor([7.]) because dy/dx = 2x + 3 = 7
```

Building a Network

```
import torch.nn as nn

class MLP(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, output_size)

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

model = MLP(784, 256, 10)
```

The Training Loop

```
model = MLP(784, 256, 10)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

for epoch in range(10):
    for batch_x, batch_y in dataloader:
        # Forward
        outputs = model(batch_x)
        loss = criterion(outputs, batch_y)

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

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

Full MNIST Example

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

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

# Train!
for epoch in range(5):
    for images, labels in train_loader:
        images = images.view(-1, 784)  # Flatten
        outputs = model(images)
        loss = criterion(outputs, labels)
        # ... backprop ...
```

Evaluating the Model

```
model.eval()  # Switch to evaluation mode
correct = 0
total = 0

with torch.no_grad():  # Don't track gradients
    for images, labels in test_loader:
        images = images.view(-1, 784)
        outputs = model(images)
        _, predicted = outputs.max(1)  # Get class with highest score
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

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

Key Takeaways

1. **Perceptron** = Linear model with activation
2. **Activation functions** add non-linearity (use ReLU!)
3. **Deep networks** can learn complex patterns
4. **Backpropagation** efficiently computes gradients
5. **PyTorch pattern:**
    - Forward → Loss → Backward → Update

Summary: Neural Network Recipe

```
# 1. Define model
model = nn.Sequential(
    nn.Linear(784, 256),
    nn.ReLU(),
    nn.Linear(256, 10)
)

# 2. Define loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters())

# 3. Train loop: forward → loss → backward → 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?