Artificial Intelligence

Lecture 9: Introduction to Neural Networks

Prof. Antonio Khalil Moretti

April 14, 2026

SCIS 432
Spelman College

Outline

Motivation

Review: Logistic Regression is Linear

- Logistic regression computes:

$$\hat{y} = \sigma(w^\top x + b)$$

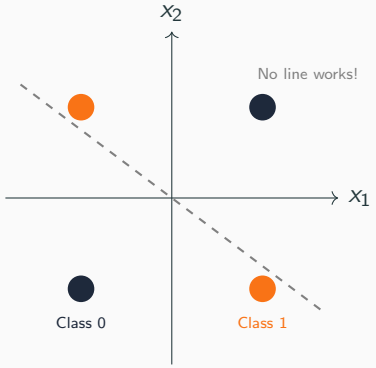
- Decision boundary:

$$w^\top x + b = 0$$

- This produces a **linear hyperplane**

Limitation: linear models cannot capture complex nonlinear structure in data.

Why Linear Models Fail: XOR



XOR-like patterns cannot be separated by any single line.

We need *nonlinear* decision boundaries — this motivates neural networks.

Building Blocks

Composing Nonlinear Functions

Stack two sigmoid units:

$$h = \sigma(w_1 x + b_1), \qquad f(x) = \sigma(w_2\, h + b_2)$$

Combined:

$$f(x) = \sigma\big(w_2\, \sigma(w_1 x + b_1) + b_2\big)$$

Composition of nonlinear functions yields **nonlinear** mappings.

With $w_1 = 8,\ b_1 = 0,\ w_2 = 8,\ b_2 = -4$ this becomes the step-like function plotted on the next slide.

Composition and Nonlinear Boundaries

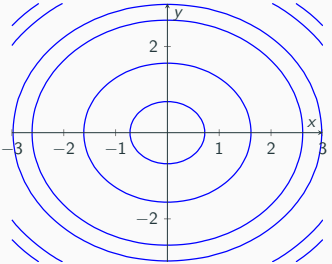
- **Linear Boundary:** $f(x, y) = ax + by + c = 0$
- **Nonlinear Boundary:** $(f \circ g)(x, y) = 0$

Example: The Radial Sine Warp

Let $g(x, y) = x^2 + y^2$ (A quadratic distance function)

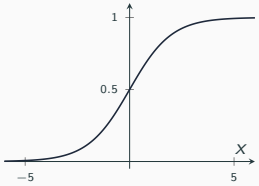
Let $f(z) = \sin(z)$ (A periodic nonlinear function)

Boundary: $\sin(x^2 + y^2) = 0.5$

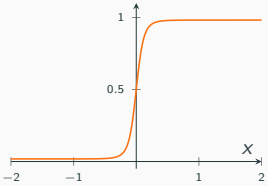


Sigmoid Composition Approximates a Step Function

$\sigma(x)$ — single sigmoid



$\sigma(8\,\sigma(8x) - 4)$ — composed



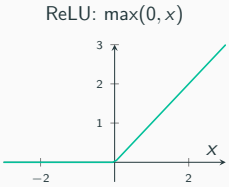
Try it yourself: `plot sigmoid(8*sigmoid(8*x) - 4)` in WolframAlpha

Why Layers?

- Each layer learns **intermediate representations**
- Early layers detect *simple* patterns (edges, tones)
- Later layers combine them into *complex* features (shapes, objects)
- Depth enables **hierarchical** feature learning

Each neuron can be viewed as a feature detector — deeper neurons build on shallower ones.

Common Activation Functions

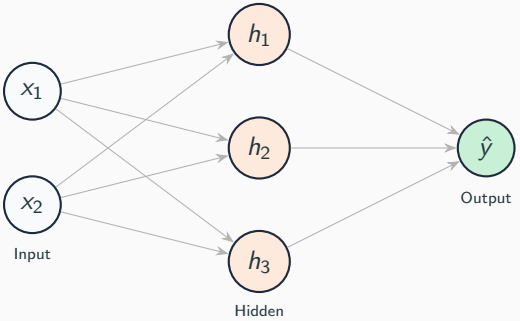


Name	Formula	Use
Sigmoid	$\frac{1}{1+e^{-x}}$	Output
Tanh	$\tanh(x)$	Hidden
ReLU	$\max(0, x)$	Hidden

ReLU is most commonly used in hidden layers in modern networks.

Network Architecture

Two-Layer Feedforward Network



Forward equations:

$$h = \sigma(W_1 x + b_1)$$

$$\hat{y} = \sigma(W_2 h + b_2)$$

This is a *fully connected* (dense) network with one hidden layer.

Matrix Dimensions

Given $x \in \mathbb{R}^2$ and 3 hidden neurons:

Param	Shape	Size
W_1	3×2	6
b_1	3	3
W_2	1×3	3
b_2	1	1
Total		**13**

General rule:

$$W_\ell \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$$

where n_ℓ = neurons in layer ℓ.

Always verify dimensions align before coding.

Training

Forward Pass

General equations

Layer 1 (hidden):

$$z^{(1)} = W_1 x + b_1$$

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

Layer 2 (output):

$$z^{(2)} = W_2 a^{(1)} + b_2$$

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

We store intermediate values $z^{(\ell)}, a^{(\ell)}$ — they are needed for backprop.

Forward Pass: Worked Example

Classifying a point as "cat" ($y = 1$) vs. "not cat" ($y = 0$)

$$x = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad W_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad b_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad W_2 = \begin{bmatrix} 1 & 1 \end{bmatrix}, \quad b_2 = -0.5$$

Hidden layer:

$$z^{(1)} = W_1 x = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$a^{(1)} = \sigma\begin{pmatrix} 1 \\ 0 \end{pmatrix} \approx \begin{bmatrix} 0.731 \\ 0.500 \end{bmatrix}$$

Output layer:

$$z^{(2)} = W_2\, a^{(1)} + b_2$$

$$= 0.731 + 0.500 - 0.5 = 0.731$$

$$\hat{y} = \sigma(0.731) \approx \mathbf{0.675}$$

$\hat{y} = 0.675 > 0.5 \;\Rightarrow\;$ predicted **cat** ✓

Loss Function

For binary classification, use **Binary Cross-Entropy**:

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

- $y \in \{0, 1\}$ is the true label
- $\hat{y} \in (0, 1)$ is the predicted probability
- Loss $= 0$ when prediction is perfect; grows without bound for confident wrong answers
- Same loss function as logistic regression

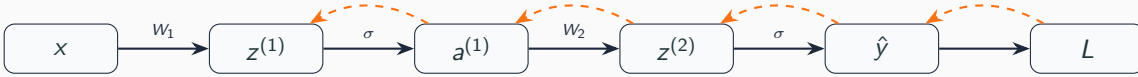
Backpropagation

Goal: minimize $L(\hat{y}, y)$ w.r.t. all parameters θ.

We need $\frac{\partial L}{\partial W_1}, \frac{\partial L}{\partial b_1}, \frac{\partial L}{\partial W_2}, \frac{\partial L}{\partial b_2}$ via the **chain rule**:

$$\frac{\partial L}{\partial W_2} = \underbrace{\frac{\partial L}{\partial \hat{y}}}_{\text{from loss}} \cdot \underbrace{\frac{\partial \hat{y}}{\partial z^{(2)}}}_{\sigma'(z^{(2)})} \cdot \underbrace{\frac{\partial z^{(2)}}{\partial W_2}}_{a^{(1)\top}}$$

Derivatives flow *backward* through the graph, reusing intermediate results.



Backprop: Worked Example

1-input, 1-hidden, 1-output scalar network with BCE loss, true label $y = 1$

Setup: $x = 1,\ w_1 = 0.5,\ b_1 = 0,\ w_2 = 1,\ b_2 = 0$

Forward pass:

$$z^{(1)} = 0.5, \quad h = \sigma(0.5) \approx 0.622, \quad z^{(2)} = 0.622, \quad \hat{y} = \sigma(0.622) \approx 0.651$$

$$L = -\log(0.651) \approx 0.429$$

Backward pass (recall $\sigma'(z) = \sigma(z)(1 - \sigma(z))$):

$$\frac{\partial L}{\partial \hat{y}} = \frac{\hat{y} - y}{\hat{y}(1 - \hat{y})} \approx -1.538 \qquad \delta^{(2)} = \frac{\partial L}{\partial \hat{y}} \cdot \sigma'(z^{(2)}) \approx -1.538 \times 0.227 \approx -0.349$$

$$\frac{\partial L}{\partial w_2} = \delta^{(2)} \cdot h \approx -0.349 \times 0.622 \approx \mathbf{-0.217}$$

$$\delta^{(1)} = \delta^{(2)} \cdot w_2 \cdot \sigma'(z^{(1)}) \approx -0.349 \times 1 \times 0.235 \approx -0.082 \qquad \frac{\partial L}{\partial w_1} = \delta^{(1)} \cdot x \approx \mathbf{-0.082}$$

Gradient Descent Training Loop

$$\theta \leftarrow \theta - \eta \nabla_\theta L$$

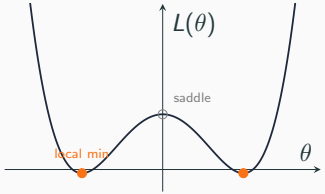
Repeat until convergence:

1. **Forward pass** — compute $\hat{y}$
2. **Compute loss** $L(\hat{y}, y)$
3. **Backward pass** — compute gradients
4. **Update weights** $\theta \leftarrow \theta - \eta \nabla_\theta L$

η is the **learning rate** — a critical hyperparameter.

Too large $\Rightarrow$ diverge.
Too small $\Rightarrow$ slow convergence.

Optimization is Non-Convex



- Neural network losses are **non-convex**
- Multiple local minima and saddle points
- Gradient descent not guaranteed to find the global minimum

Despite this, SGD finds good solutions in large networks in practice.

PyTorch

Automatic Differentiation in PyTorch

```python
import torch

x = torch.tensor([2.0], requires_grad=True)

y = x**2 + 3*x + 1    # y = f(x)

y.backward()          # compute dy/dx automatically

print(x.grad)         # tensor([7.]) since f'(x) = 2x+3 = 7
```

PyTorch builds a *computation graph* during the forward pass and uses it to compute exact gradients during `backward()`.

A Simple Neural Network in PyTorch

```
import torch
import torch.nn as nn

model = nn.Sequential(
    nn.Linear(2, 3),   # W1, b1
    nn.Sigmoid(),
    nn.Linear(3, 1),   # W2, b2
    nn.Sigmoid(),
)

loss_fn   = nn.BCELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)

for epoch in range(100):
    y_hat = model(X)          # forward pass
    loss  = loss_fn(y_hat, y) # compute loss
    optimizer.zero_grad()     # clear old gradients
    loss.backward()           # backprop
    optimizer.step()          # update weights
```

Summary

Summary

Concepts:

- ✓ Logistic regression = linear boundary
- ✓ NNs stack linear layers + nonlinear activations
- ✓ Composition enables nonlinear boundaries
- ✓ Layers learn hierarchical features

Mechanics:

- ✓ Forward pass: compute $\hat{y}$ layer by layer
- ✓ Loss: binary cross-entropy
- ✓ Backprop: chain rule, efficient via DP
- ✓ Gradient descent updates θ
- ✓ PyTorch automates differentiation

Next lecture: deeper networks, regularization, and convolutional neural networks.