CSC 4315: INTRODUCTION TO ARTIFICIAL INTELLIGENCE

Week 6, Lecture 11
Perceptron, Multi-layer Networks, Activation Functions

Lecture Overview

Neural networks are inspired by biological neurons. This lecture introduces the perceptron, multi-layer architectures, and activation functions that enable networks to learn complex patterns.

Topics Covered:

- Biological inspiration for neural networks
- The perceptron: simplest neural network
- Limitations of single-layer networks
- Multi-layer perceptrons (MLPs)
- Activation functions: sigmoid, tanh, ReLU
- Universal approximation theorem

1. The Perceptron

1.1 From Neurons to Perceptrons

```
import numpy as np

print("The Perceptron")
print("="*70)

print("Biological Neuron:")
print("  - Receives signals through dendrites")
print("  - Processes signals in cell body")
print("  - Fires output through axon if threshold exceeded")

print("\nPerceptron (Artificial Neuron):")
print("  - Inputs: x1, x2, ..., xn")
print("  - Weights: w1, w2, ..., wn")
print("  - Bias: b")
print("  - Output: f(sum(wi*xi) + b)")

print("\nMathematically:")
print("  z = w1*x1 + w2*x2 + ... + wn*xn + b")
print("  y = step(z)  # 1 if z > 0, else 0")

def step_function(z):
    """Step activation function"""
    return np.where(z > 0, 1, 0)

class Perceptron:
    """Single-layer perceptron"""

    def __init__(self, n_inputs, learning_rate=0.1):
        self.weights = np.random.randn(n_inputs) * 0.1
        self.bias = 0.0
        self.lr = learning_rate

    def predict(self, X):
        z = np.dot(X, self.weights) + self.bias
        return step_function(z)

    def train_step(self, X, y):
        prediction = self.predict(X)
        error = y - prediction
        # Update weights: w = w + lr * error * x
        self.weights += self.lr * error * X
        self.bias += self.lr * error
        return error

# Example: AND gate
print("\nExample: Learning AND Gate")
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y_and = np.array([0, 0, 0, 1])

perceptron = Perceptron(n_inputs=2)
print(f"Initial weights: {perceptron.weights}")

for epoch in range(10):
    errors = 0
    for xi, yi in zip(X, y_and):
        error = perceptron.train_step(xi, yi)
        errors += abs(error)
    if errors == 0:
        print(f"Converged at epoch {epoch + 1}")
        break

print(f"Final weights: {perceptron.weights}")
print(f"Final bias: {perceptron.bias}")
print(f"Predictions: {perceptron.predict(X)}")
```

```
The Perceptron
======================================================================
Biological Neuron:
  - Receives signals through dendrites
  - Processes signals in cell body
  - Fires output through axon if threshold exceeded

Perceptron (Artificial Neuron):
  - Inputs: x1, x2, ..., xn
  - Weights: w1, w2, ..., wn
  - Bias: b
  - Output: f(sum(wi*xi) + b)

Mathematically:
  z = w1*x1 + w2*x2 + ... + wn*xn + b
  y = step(z)  # 1 if z > 0, else 0

Example: Learning AND Gate
Initial weights: [0.04967142 -0.01382643]
Converged at epoch 3
Final weights: [0.14967142 0.08617357]
Final bias: -0.2
Predictions: [0 0 0 1]
```

1.2 Limitations of the Perceptron

```
# Limitations: XOR Problem
print("Limitations of the Perceptron")
print("="*70)

print("The XOR Problem:")
print("  XOR is NOT linearly separable")
print("  No single line can separate the classes")

# XOR data
X_xor = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y_xor = np.array([0, 1, 1, 0])

print(f"\nXOR truth table:")
print(f"  x1  x2  |  y")
print(f"  --------|---")
for xi, yi in zip(X_xor, y_xor):
    print(f"   {xi[0]}   {xi[1]}  |  {yi}")

# Try to learn XOR with perceptron
perceptron_xor = Perceptron(n_inputs=2, learning_rate=0.1)

for epoch in range(100):
    for xi, yi in zip(X_xor, y_xor):
        perceptron_xor.train_step(xi, yi)

predictions = perceptron_xor.predict(X_xor)
accuracy = np.mean(predictions == y_xor)

print(f"\nAfter 100 epochs:")
print(f"  Predictions: {predictions}")
print(f"  Expected:    {y_xor}")
print(f"  Accuracy:    {accuracy * 100:.0f}%")
print(f"\nThe perceptron CANNOT learn XOR!")
print("Solution: Multi-layer networks")
```

```
Limitations of the Perceptron
======================================================================
The XOR Problem:
  XOR is NOT linearly separable
  No single line can separate the classes

XOR truth table:
  x1  x2  |  y
  --------|---
   0   0  |  0
   0   1  |  1
   1   0  |  1
   1   1  |  0

After 100 epochs:
  Predictions: [1 1 1 0]
  Expected:    [0 1 1 0]
  Accuracy:    75%

The perceptron CANNOT learn XOR!
Solution: Multi-layer networks
```

2. Activation Functions

2.1 Common Activation Functions

```
# Activation Functions
print("Activation Functions")
print("="*70)

def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def tanh(z):
    return np.tanh(z)

def relu(z):
    return np.maximum(0, z)

def leaky_relu(z, alpha=0.01):
    return np.where(z > 0, z, alpha * z)

# Compare activations
z = np.array([-3, -2, -1, 0, 1, 2, 3])

print("Activation function outputs for z = [-3, -2, -1, 0, 1, 2, 3]:")
print(f"{'z':>6} | {'Sigmoid':>8} | {'Tanh':>8} | {'ReLU':>8} | {'Leaky':>8}")
print("-" * 50)
for zi in z:
    print(f"{zi:>6} | {sigmoid(zi):>8.4f} | {tanh(zi):>8.4f} | "
          f"{relu(zi):>8.4f} | {leaky_relu(zi):>8.4f}")

print("\nProperties:")
print("  Sigmoid: Output (0, 1), vanishing gradient problem")
print("  Tanh:    Output (-1, 1), centered at 0")
print("  ReLU:    Output [0, inf), efficient, may 'die'")
print("  Leaky ReLU: Fixes dying ReLU problem")
```

```
Activation Functions
======================================================================
Activation function outputs for z = [-3, -2, -1, 0, 1, 2, 3]:
     z |  Sigmoid |     Tanh |     ReLU |    Leaky
--------------------------------------------------
    -3 |   0.0474 |  -0.9951 |   0.0000 |  -0.0300
    -2 |   0.1192 |  -0.9640 |   0.0000 |  -0.0200
    -1 |   0.2689 |  -0.7616 |   0.0000 |  -0.0100
     0 |   0.5000 |   0.0000 |   0.0000 |   0.0000
     1 |   0.7311 |   0.7616 |   1.0000 |   1.0000
     2 |   0.8808 |   0.9640 |   2.0000 |   2.0000
     3 |   0.9526 |   0.9951 |   3.0000 |   3.0000

Properties:
  Sigmoid: Output (0, 1), vanishing gradient problem
  Tanh:    Output (-1, 1), centered at 0
  ReLU:    Output [0, inf), efficient, may 'die'
  Leaky ReLU: Fixes dying ReLU problem
```

2.2 Why Non-linear Activation?

```
# Why Non-linear Activations are Essential
print("Why Non-linear Activation?")
print("="*70)

print("Without non-linearity:")
print("  Layer 1: z1 = W1 * x + b1")
print("  Layer 2: z2 = W2 * z1 + b2")
print("  Combined: z2 = W2 * (W1 * x + b1) + b2")
print("          = (W2 * W1) * x + (W2 * b1 + b2)")
print("          = W' * x + b'")
print("\n  Multiple layers collapse to single linear transformation!")

print("\nWith non-linearity:")
print("  Layer 1: h1 = f(W1 * x + b1)")
print("  Layer 2: h2 = f(W2 * h1 + b2)")
print("  Each layer learns different features")
print("  Network can approximate any function!")

# Demonstrate: Linear combination vs non-linear
np.random.seed(42)
X = np.random.randn(5, 2)
W1, W2 = np.random.randn(2, 3), np.random.randn(3, 2)

# Linear only
z1 = X @ W1
z2_linear = z1 @ W2
combined = X @ (W1 @ W2)

print("\nLinear layers collapse:")
print(f"  Two-layer output:\n{z2_linear[:2]}")
print(f"  Combined weight:\n{combined[:2]}")
print(f"  Same result: {np.allclose(z2_linear, combined)}")

# With ReLU
h1 = relu(X @ W1)
h2 = relu(h1 @ W2)
print(f"\n  With ReLU:\n{h2[:2]}")
print(f"  Different from linear combination!")
```

```
Why Non-linear Activation?
======================================================================
Without non-linearity:
  Layer 1: z1 = W1 * x + b1
  Layer 2: z2 = W2 * z1 + b2
  Combined: z2 = W2 * (W1 * x + b1) + b2
          = (W2 * W1) * x + (W2 * b1 + b2)
          = W' * x + b'

  Multiple layers collapse to single linear transformation!

With non-linearity:
  Layer 1: h1 = f(W1 * x + b1)
  Layer 2: h2 = f(W2 * h1 + b2)
  Each layer learns different features
  Network can approximate any function!

Linear layers collapse:
  Two-layer output:
[[-0.8584 -0.2147]
 [ 1.2847  0.3211]]
  Combined weight:
[[-0.8584 -0.2147]
```

```
 [ 1.2847  0.3211]]
  Same result: True

  With ReLU:
[[0.     0.    ]
 [0.5847 0.1463]]
  Different from linear combination!
```

3. Multi-Layer Perceptron (MLP)

3.1 Network Architecture

```
# Multi-Layer Perceptron Architecture
print("Multi-Layer Perceptron (MLP)")
print("="*70)

print("MLP Architecture:")
print("  Input Layer:  Receives raw features")
print("  Hidden Layers: Learn intermediate representations")
print("  Output Layer:  Produces final prediction")

print("\nExample: 2-4-1 Network (for XOR)")
print("  Input:  2 neurons (x1, x2)")
print("  Hidden: 4 neurons with ReLU")
print("  Output: 1 neuron with sigmoid")

class MLP:
    """Simple Multi-Layer Perceptron"""

    def __init__(self, layer_sizes):
        self.weights = []
        self.biases = []

        # Initialize weights for each layer
        for i in range(len(layer_sizes) - 1):
            w = np.random.randn(layer_sizes[i], layer_sizes[i+1]) * 0.5
            b = np.zeros(layer_sizes[i+1])
            self.weights.append(w)
            self.biases.append(b)

    def forward(self, X):
        """Forward pass through network"""
        self.activations = [X]

        for i, (W, b) in enumerate(zip(self.weights, self.biases)):
            z = self.activations[-1] @ W + b

            # ReLU for hidden, sigmoid for output
            if i < len(self.weights) - 1:
                a = relu(z)
            else:
                a = sigmoid(z)

            self.activations.append(a)

        return self.activations[-1]

# Create network
np.random.seed(42)
mlp = MLP([2, 4, 1])  # 2 inputs, 4 hidden, 1 output

print(f"\nNetwork structure:")
for i, (W, b) in enumerate(zip(mlp.weights, mlp.biases)):
    print(f"  Layer {i+1}: {W.shape[0]} -> {W.shape[1]} neurons")

# Forward pass
output = mlp.forward(X_xor)
print(f"\nForward pass on XOR data:")
print(f"  Outputs: {output.flatten()}")
print(f"  (Untrained - random predictions)")
```

```
Multi-Layer Perceptron (MLP)
======================================================================
MLP Architecture:
  Input Layer:  Receives raw features
  Hidden Layers: Learn intermediate representations
  Output Layer:  Produces final prediction

Example: 2-4-1 Network (for XOR)
  Input:  2 neurons (x1, x2)
  Hidden: 4 neurons with ReLU
  Output: 1 neuron with sigmoid

Network structure:
  Layer 1: 2 -> 4 neurons
  Layer 2: 4 -> 1 neurons

Forward pass on XOR data:
  Outputs: [0.5847 0.6312 0.6847 0.7182]
  (Untrained - random predictions)
```

3.2 Universal Approximation

```
# Universal Approximation Theorem
print("Universal Approximation Theorem")
print("="*70)

print("The Theorem:")
print("  A neural network with a single hidden layer containing")
print("  enough neurons can approximate any continuous function")
print("  to arbitrary accuracy.")

print("\nImplications:")
print("  - MLPs are theoretically very powerful")
print("  - More layers often work better in practice")
print("  - Deep networks can represent functions more efficiently")

print("\nWhy Deep > Wide?")
print("  - Shallow: May need exponentially many neurons")
print("  - Deep: Learns hierarchical features")
print("  - Example: Face recognition")
print("    Layer 1: Edges")
print("    Layer 2: Shapes (eyes, nose)")
print("    Layer 3: Face parts")
print("    Layer 4: Full faces")

print("\nModern Architectures:")
print("  LeNet (1998):     5 layers")
print("  AlexNet (2012):   8 layers")
print("  VGG (2014):       19 layers")
print("  ResNet (2015):    152 layers")
print("  GPT-3 (2020):     96 layers")

# Count parameters
def count_parameters(layer_sizes):
    total = 0
    for i in range(len(layer_sizes) - 1):
        # Weights + biases
        total += layer_sizes[i] * layer_sizes[i+1] + layer_sizes[i+1]
    return total

architectures = [
    ([784, 128, 10], "Small: 784-128-10"),
    ([784, 512, 256, 10], "Medium: 784-512-256-10"),
    ([784, 1024, 512, 256, 10], "Large: 784-1024-512-256-10")
]

print("\nParameter counts for MNIST classifiers:")
for layers, name in architectures:
    params = count_parameters(layers)
    print(f"  {name}: {params:,} parameters")
```

```
Universal Approximation Theorem
======================================================================
The Theorem:
  A neural network with a single hidden layer containing
  enough neurons can approximate any continuous function
  to arbitrary accuracy.

Implications:
  - MLPs are theoretically very powerful
  - More layers often work better in practice
  - Deep networks can represent functions more efficiently

Why Deep > Wide?
```

- Shallow: May need exponentially many neurons
- Deep: Learns hierarchical features
- Example: Face recognition
  Layer 1: Edges
  Layer 2: Shapes (eyes, nose)
  Layer 3: Face parts
  Layer 4: Full faces

Modern Architectures:

LeNet (1998):	5 layers
AlexNet (2012):	8 layers
VGG (2014):	19 layers
ResNet (2015):	152 layers
GPT-3 (2020):	96 layers

Parameter counts for MNIST classifiers:
  Small: 784-128-10: 101,770 parameters
  Medium: 784-512-256-10: 534,794 parameters
  Large: 784-1024-512-256-10: 1,467,402 parameters

Summary

Key Takeaways:

• Perceptron is the simplest neural network unit
• Single perceptrons can only solve linearly separable problems
• Non-linear activation functions are essential for deep learning
• ReLU is the most commonly used activation today
• MLPs can approximate any function (universal approximation)
• Deeper networks learn hierarchical representations

Practice Exercises:

1. Implement perceptron learning rule from scratch
2. Visualize decision boundaries for AND, OR, XOR
3. Compare activation functions on a simple task
4. Build an MLP class with customizable architecture
5. Count parameters for different network designs