CSC 4315: INTRODUCTION TO ARTIFICIAL INTELLIGENCE

Week 7, Lecture 13
Backpropagation Algorithm and Chain Rule

Lecture Overview

Backpropagation is the algorithm that makes training deep neural networks possible. This lecture derives backpropagation from first principles using the chain rule of calculus.

Topics Covered:

- Gradient-based optimization review
- The chain rule of calculus
- Computational graphs
- Backpropagation derivation
- Computing gradients layer by layer
- Implementing backprop from scratch

1. The Chain Rule

1.1 Calculus Foundation

```
import numpy as np

print("The Chain Rule of Calculus")
print("="*70)

print("Single Variable Chain Rule:")
print("  If y = f(g(x)), then dy/dx = (df/dg) * (dg/dx)")
print("\nExample: y = (2x + 3)^2")
print("  Let u = 2x + 3, y = u^2")
print("  dy/dx = (dy/du) * (du/dx) = 2u * 2 = 4(2x + 3)")

# Verify numerically
def f(x):
    return (2*x + 3)**2

def df_analytic(x):
    return 4 * (2*x + 3)

x = 2.0
h = 1e-5
df_numeric = (f(x + h) - f(x - h)) / (2 * h)

print(f"\nAt x = {x}:")
print(f"  Analytical derivative: {df_analytic(x):.4f}")
print(f"  Numerical derivative:  {df_numeric:.4f}")

print("\nMultivariable Chain Rule:")
print("  If L = f(g(h(x))), then:")
print("  dL/dx = (dL/dg) * (dg/dh) * (dh/dx)")
print("\n  This extends to any depth!")

# Neural network example
print("\nNeural Network Example:")
print("  L = Loss(sigmoid(W2 @ relu(W1 @ x + b1) + b2), y)")
print("  To get dL/dW1, we chain through each operation:")
print("    dL/dW1 = (dL/dsigmoid) * (dsigmoid/dz2) * (dz2/drelu)")
print("             * (drelu/dz1) * (dz1/dW1)")
```

```
The Chain Rule of Calculus
======================================================================
Single Variable Chain Rule:
  If y = f(g(x)), then dy/dx = (df/dg) * (dg/dx)

Example: y = (2x + 3)^2
  Let u = 2x + 3, y = u^2
  dy/dx = (dy/du) * (du/dx) = 2u * 2 = 4(2x + 3)

At x = 2.0:
  Analytical derivative: 28.0000
  Numerical derivative:  28.0000

Multivariable Chain Rule:
  If L = f(g(h(x))), then:
  dL/dx = (dL/dg) * (dg/dh) * (dh/dx)

  This extends to any depth!

Neural Network Example:
  L = Loss(sigmoid(W2 @ relu(W1 @ x + b1) + b2), y)
```

```
To get dL/dW1, we chain through each operation:
  dL/dW1 = (dL/dsigmoid) * (dsigmoid/dz2) * (dz2/drelu)
           * (drelu/dz1) * (dz1/dW1)
```

2. Computational Graphs

2.1 Representing Computations

```
# Computational Graphs
print("Computational Graphs")
print("="*70)

print("A computational graph represents operations as nodes:")
print("\nExample: f(x, y, z) = (x + y) * z")
print("\n  x ----\\")
print("         (+) ---- a ----\\")
print("  y ----/                (*) ---- f")
print("                   z ----/")
print("\n  where a = x + y, f = a * z")

print("\nForward Pass: Compute values left to right")
print("Backward Pass: Compute gradients right to left")

# Simple computational graph
class Node:
    """Base class for computational graph nodes"""
    def __init__(self):
        self.value = None
        self.grad = 0

    def forward(self):
        raise NotImplementedError

    def backward(self):
        raise NotImplementedError

class Add(Node):
    def __init__(self, a, b):
        super().__init__()
        self.a, self.b = a, b

    def forward(self):
        self.value = self.a.value + self.b.value
        return self.value

    def backward(self):
        # d(a+b)/da = 1, d(a+b)/db = 1
        self.a.grad += self.grad * 1
        self.b.grad += self.grad * 1

class Multiply(Node):
    def __init__(self, a, b):
        super().__init__()
        self.a, self.b = a, b

    def forward(self):
        self.value = self.a.value * self.b.value
        return self.value

    def backward(self):
        # d(a*b)/da = b, d(a*b)/db = a
        self.a.grad += self.grad * self.b.value
        self.b.grad += self.grad * self.a.value

class Variable(Node):
    def __init__(self, value):
        super().__init__()
```

```
        self.value = value

# Build graph: f = (x + y) * z
x = Variable(2.0)
y = Variable(3.0)
z = Variable(4.0)

a = Add(x, y)       # a = x + y
f = Multiply(a, z)  # f = a * z

# Forward pass
a.forward()
f.forward()

print(f"\nExample: f = (x + y) * z, with x=2, y=3, z=4")
print(f"Forward: a = {a.value}, f = {f.value}")

# Backward pass
f.grad = 1.0  # dL/df = 1
f.backward()
a.backward()

print(f"\nBackward pass:")
print(f"  df/dz = {z.grad} (equals a = x + y)")
print(f"  df/da = {a.grad} (equals z)")
print(f"  df/dx = {x.grad} (equals z, since da/dx = 1)")
print(f"  df/dy = {y.grad} (equals z, since da/dy = 1)")
```

```
Computational Graphs
======================================================================
A computational graph represents operations as nodes:

Example: f(x, y, z) = (x + y) * z

  x ----\
         (+) ---- a ----\
  y ----/                (*) ---- f
                    z ----/

  where a = x + y, f = a * z

Forward Pass: Compute values left to right
Backward Pass: Compute gradients right to left

Example: f = (x + y) * z, with x=2, y=3, z=4
Forward: a = 5.0, f = 20.0

Backward pass:
  df/dz = 5.0 (equals a = x + y)
  df/da = 4.0 (equals z)
  df/dx = 4.0 (equals z, since da/dx = 1)
  df/dy = 4.0 (equals z, since da/dy = 1)
```

3. Backpropagation Derivation

3.1 Gradients for Each Layer

```
# Backpropagation Derivation
print("Backpropagation Derivation")
print("="*70)

print("Two-layer network:")
print("  z1 = W1 @ x + b1")
print("  a1 = relu(z1)")
print("  z2 = W2 @ a1 + b2")
print("  a2 = sigmoid(z2)")
print("  L = binary_cross_entropy(y, a2)")

print("\nGoal: Compute dL/dW1, dL/db1, dL/dW2, dL/db2")

print("\nStep 1: Output layer gradients")
print("  dL/da2 = -(y/a2) + (1-y)/(1-a2)")
print("  da2/dz2 = a2 * (1 - a2)        # sigmoid derivative")
print("  dL/dz2 = dL/da2 * da2/dz2 = a2 - y  # simplifies!")

print("\nStep 2: Output layer parameters")
print("  dL/dW2 = dL/dz2 @ a1.T")
print("  dL/db2 = sum(dL/dz2)")

print("\nStep 3: Propagate to hidden layer")
print("  dL/da1 = W2.T @ dL/dz2")
print("  da1/dz1 = 1 if z1 > 0 else 0   # relu derivative")
print("  dL/dz1 = dL/da1 * da1/dz1")

print("\nStep 4: Hidden layer parameters")
print("  dL/dW1 = dL/dz1 @ x.T")
print("  dL/db1 = sum(dL/dz1)")

# Activation derivatives
def sigmoid(z):
    return 1 / (1 + np.exp(-np.clip(z, -500, 500)))

def sigmoid_derivative(z):
    s = sigmoid(z)
    return s * (1 - s)

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

def relu_derivative(z):
    return (z > 0).astype(float)

# Show derivatives
z_test = np.array([-2, -1, 0, 1, 2])
print(f"\nDerivative values at z = {z_test}:")
print(f"  Sigmoid:    {np.round(sigmoid(z_test), 3)}")
print(f"  Sigmoid':   {np.round(sigmoid_derivative(z_test), 3)}")
print(f"  ReLU:       {relu(z_test)}")
print(f"  ReLU':      {relu_derivative(z_test)}")
```

```
Backpropagation Derivation
======================================================================
Two-layer network:
  z1 = W1 @ x + b1
  a1 = relu(z1)
  z2 = W2 @ a1 + b2
```

```
  a2 = sigmoid(z2)
  L = binary_cross_entropy(y, a2)

Goal: Compute dL/dW1, dL/db1, dL/dW2, dL/db2

Step 1: Output layer gradients
  dL/da2 = -(y/a2) + (1-y)/(1-a2)
  da2/dz2 = a2 * (1 - a2)        # sigmoid derivative
  dL/dz2 = dL/da2 * da2/dz2 = a2 - y  # simplifies!

Step 2: Output layer parameters
  dL/dW2 = dL/dz2 @ a1.T
  dL/db2 = sum(dL/dz2)

Step 3: Propagate to hidden layer
  dL/da1 = W2.T @ dL/dz2
  da1/dz1 = 1 if z1 > 0 else 0   # relu derivative
  dL/dz1 = dL/da1 * da1/dz1

Step 4: Hidden layer parameters
  dL/dW1 = dL/dz1 @ x.T
  dL/db1 = sum(dL/dz1)

Derivative values at z = [-2 -1  0  1  2]:
  Sigmoid:    [0.119 0.269 0.5   0.731 0.881]
  Sigmoid':   [0.105 0.197 0.25  0.197 0.105]
  ReLU:       [0 0 0 1 2]
  ReLU':      [0. 0. 0. 1. 1.]
```

4. Complete Backpropagation Implementation

4.1 Neural Network with Backprop

```
# Complete Backpropagation Implementation
print("Complete Backpropagation Implementation")
print("="*70)

class NeuralNetworkBackprop:
    """Neural network with backpropagation"""

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

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

    def forward(self, X):
        """Forward pass, caching values for backprop"""
        self.cache = {'A0': X}
        A = X

        for i, (W, b) in enumerate(zip(self.weights, self.biases)):
            Z = A @ W + b
            self.cache[f'Z{i+1}'] = Z

            if i < len(self.weights) - 1:
                A = relu(Z)
            else:
                A = sigmoid(Z)

            self.cache[f'A{i+1}'] = A

        return A

    def backward(self, y):
        """Backward pass - compute gradients"""
        m = y.shape[0]
        L = len(self.weights)
        grads = {}

        # Output layer
        AL = self.cache[f'A{L}']
        dZ = AL - y  # For sigmoid + BCE

        grads[f'dW{L}'] = (1/m) * self.cache[f'A{L-1}'].T @ dZ
        grads[f'db{L}'] = (1/m) * np.sum(dZ, axis=0, keepdims=True)

        # Hidden layers
        for l in range(L - 1, 0, -1):
            dA = dZ @ self.weights[l].T
            dZ = dA * relu_derivative(self.cache[f'Z{l}'])

            grads[f'dW{l}'] = (1/m) * self.cache[f'A{l-1}'].T @ dZ
            grads[f'db{l}'] = (1/m) * np.sum(dZ, axis=0, keepdims=True)

        return grads

    def update(self, grads, lr):
```

```
        """Update weights using gradients"""
        for i in range(len(self.weights)):
            self.weights[i] -= lr * grads[f'dW{i+1}']
            self.biases[i] -= lr * grads[f'db{i+1}']

# Test on XOR
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])

np.random.seed(42)
nn = NeuralNetworkBackprop([2, 8, 1])

print("Training on XOR:")
for epoch in range(1000):
    # Forward
    pred = nn.forward(X)

    # Backward
    grads = nn.backward(y)

    # Update
    nn.update(grads, lr=1.0)

    if epoch % 200 == 0:
        loss = -np.mean(y*np.log(pred+1e-8) + (1-y)*np.log(1-pred+1e-8))
        print(f"  Epoch {epoch}: Loss = {loss:.4f}")

print(f"\nFinal predictions:")
print(f"  Input    | Target | Predicted")
for xi, yi, pi in zip(X, y.flatten(), pred.flatten()):
    print(f"  {xi}    |   {yi}    |   {pi:.4f}")
```

```
Complete Backpropagation Implementation
======================================================================
Training on XOR:
  Epoch 0: Loss = 0.7182
  Epoch 200: Loss = 0.2847
  Epoch 400: Loss = 0.0521
  Epoch 600: Loss = 0.0184
  Epoch 800: Loss = 0.0098

Final predictions:
  Input    | Target | Predicted
  [0 0]    |   0    |   0.0124
  [0 1]    |   1    |   0.9847
  [1 0]    |   1    |   0.9821
  [1 1]    |   0    |   0.0182
```

5. Gradient Checking

5.1 Verifying Gradients Numerically

```
# Gradient Checking
print("Gradient Checking")
print("="*70)

print("Numerical gradient approximation:")
print("  dL/dw ≈ (L(w + h) - L(w - h)) / (2h)")
print("\nUsed to verify analytical gradients are correct")

def compute_loss(nn, X, y):
    pred = nn.forward(X)
    return -np.mean(y*np.log(pred+1e-8) + (1-y)*np.log(1-pred+1e-8))

def gradient_check(nn, X, y, epsilon=1e-7):
    """Check gradients numerically"""
    # Get analytical gradients
    nn.forward(X)
    grads = nn.backward(y)

    errors = []

    # Check each weight matrix
    for i, W in enumerate(nn.weights):
        dW_analytical = grads[f'dW{i+1}']
        dW_numerical = np.zeros_like(W)

        # Compute numerical gradient for each element
        for j in range(min(3, W.shape[0])):  # Check first 3 for speed
            for k in range(min(3, W.shape[1])):
                # f(w + h)
                W[j, k] += epsilon
                loss_plus = compute_loss(nn, X, y)

                # f(w - h)
                W[j, k] -= 2 * epsilon
                loss_minus = compute_loss(nn, X, y)

                # Restore
                W[j, k] += epsilon

                dW_numerical[j, k] = (loss_plus - loss_minus) / (2 * epsilon)

        # Compare
        diff = np.abs(dW_analytical[:3, :3] - dW_numerical[:3, :3])
        rel_error = np.max(diff) / (np.max(np.abs(dW_analytical[:3, :3])) + 1e-8)
        errors.append(rel_error)

        print(f"  Layer {i+1} relative error: {rel_error:.2e}")

    return errors

np.random.seed(42)
nn_check = NeuralNetworkBackprop([2, 4, 1])
errors = gradient_check(nn_check, X, y)

print(f"\nGradient check passed: {all(e < 1e-5 for e in errors)}")
print("(Relative error < 1e-5 indicates correct implementation)")
```

```
Gradient Checking
======================================================================
```

```
Numerical gradient approximation:
  dL/dw ≈ (L(w + h) - L(w - h)) / (2h)

Used to verify analytical gradients are correct
  Layer 1 relative error: 2.84e-08
  Layer 2 relative error: 1.47e-08

Gradient check passed: True
(Relative error < 1e-5 indicates correct implementation)
```

Summary

Key Takeaways:

- Chain rule enables computing gradients through compositions
- Computational graphs represent forward computation
- Backprop traverses graph backward, accumulating gradients
- Each layer's gradient depends on gradients from later layers
- Cache forward pass values for use in backward pass
- Gradient checking verifies implementation correctness

Practice Exercises:

1. Derive backprop for a 3-layer network
2. Implement backprop for tanh activation
3. Add gradient checking to your neural network
4. Visualize how gradients flow through the network
5. Implement backprop for softmax + cross-entropy