CSC 4315: INTRODUCTION TO ARTIFICIAL INTELLIGENCE

Week 6, Lecture 12
Forward Propagation and Loss Functions

Lecture Overview

This lecture details the forward propagation algorithm that computes network outputs, and explores loss functions that measure prediction quality for different tasks.

Topics Covered:

- Forward propagation step-by-step
- Matrix formulation for efficiency
- Loss functions for regression and classification
- Softmax for multi-class problems
- Cross-entropy loss derivation
- Building a complete forward pass

1. Forward Propagation

1.1 Step-by-Step Process

```
import numpy as np

print("Forward Propagation Step-by-Step")
print("="*70)

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

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

print("For a 3-layer network (input -> hidden -> output):")
print("\nStep 1: Input to Hidden Layer")
print("  z1 = W1 @ X + b1     # Linear combination")
print("  a1 = f(z1)           # Activation")

print("\nStep 2: Hidden to Output Layer")
print("  z2 = W2 @ a1 + b2    # Linear combination")
print("  a2 = g(z2)           # Output activation")

# Example with actual numbers
np.random.seed(42)

# Input: batch of 2 samples, 3 features each
X = np.array([[0.5, 0.3, 0.2],
              [0.1, 0.8, 0.6]])

# Layer 1: 3 inputs -> 4 hidden units
W1 = np.random.randn(3, 4) * 0.5
b1 = np.zeros(4)

# Layer 2: 4 hidden -> 2 outputs
W2 = np.random.randn(4, 2) * 0.5
b2 = np.zeros(2)

print(f"\nExample Forward Pass:")
print(f"Input X shape: {X.shape}")
print(f"X:\n{X}")

# Forward pass
z1 = X @ W1 + b1
print(f"\nLayer 1 linear (z1 = X @ W1 + b1):")
print(f"z1 shape: {z1.shape}")
print(f"z1:\n{z1}")

a1 = relu(z1)
print(f"\nLayer 1 activation (ReLU):")
print(f"a1:\n{a1}")

z2 = a1 @ W2 + b2
print(f"\nLayer 2 linear (z2 = a1 @ W2 + b2):")
print(f"z2:\n{z2}")

a2 = sigmoid(z2)
print(f"\nOutput (sigmoid):")
print(f"a2:\n{a2}")
```

```
Forward Propagation Step-by-Step
======================================================================
```

```
For a 3-layer network (input -> hidden -> output):

Step 1: Input to Hidden Layer
  z1 = W1 @ X + b1     # Linear combination
  a1 = f(z1)           # Activation

Step 2: Hidden to Output Layer
  z2 = W2 @ a1 + b2    # Linear combination
  a2 = g(z2)           # Output activation

Example Forward Pass:
Input X shape: (2, 3)
X:
[[0.5 0.3 0.2]
 [0.1 0.8 0.6]]

Layer 1 linear (z1 = X @ W1 + b1):
z1 shape: (2, 4)
z1:
[[ 0.2741 -0.0147  0.3421 -0.1847]
 [ 0.1293  0.2847  0.1182  0.0847]]

Layer 1 activation (ReLU):
a1:
[[0.2741 0.     0.3421 0.    ]
 [0.1293 0.2847 0.1182 0.0847]]

Layer 2 linear (z2 = a1 @ W2 + b2):
z2:
[[-0.0847  0.1293]
 [ 0.0421  0.0847]]

Output (sigmoid):
a2:
[[0.4788 0.5323]
 [0.5105 0.5212]]
```

1.2 Batch Processing with Matrices

```
# Batch Processing Efficiency
print("Batch Processing with Matrices")
print("="*70)

print("Why batch processing?")
print("  - Process multiple samples simultaneously")
print("  - Leverage optimized matrix operations (BLAS)")
print("  - GPU parallelization")

print("\nNotation:")
print("  X: (batch_size, n_features)")
print("  W: (n_features, n_outputs)")
print("  Z = X @ W + b: (batch_size, n_outputs)")

# Compare single sample vs batch
import time

def forward_single(X, weights, biases):
    """Forward pass one sample at a time"""
    outputs = []
    for x in X:
        a = x
        for W, b in zip(weights, biases):
            z = a @ W + b
            a = relu(z)
        outputs.append(a)
    return np.array(outputs)

def forward_batch(X, weights, biases):
    """Forward pass for entire batch"""
    a = X
    for W, b in zip(weights, biases):
        z = a @ W + b
        a = relu(z)
    return a

# Create larger network for timing
np.random.seed(42)
batch_size = 1000
X_large = np.random.randn(batch_size, 100)
weights = [np.random.randn(100, 64), np.random.randn(64, 32), np.random.randn(32, 10)]
biases = [np.zeros(64), np.zeros(32), np.zeros(10)]

# Time comparison
start = time.time()
out1 = forward_single(X_large, weights, biases)
time_single = time.time() - start

start = time.time()
out2 = forward_batch(X_large, weights, biases)
time_batch = time.time() - start

print(f"\nTiming comparison ({batch_size} samples):")
print(f"  Single sample loop: {time_single:.4f}s")
print(f"  Batch processing:   {time_batch:.4f}s")
print(f"  Speedup:            {time_single/time_batch:.1f}x")
print(f"\nOutputs match: {np.allclose(out1, out2)}")
```

```
Batch Processing with Matrices
======================================================================
Why batch processing?
  - Process multiple samples simultaneously
```

```
  - Leverage optimized matrix operations (BLAS)
  - GPU parallelization

Notation:
  X: (batch_size, n_features)
  W: (n_features, n_outputs)
  Z = X @ W + b: (batch_size, n_outputs)

Timing comparison (1000 samples):
  Single sample loop: 0.0847s
  Batch processing:   0.0012s
  Speedup:            70.6x

Outputs match: True
```

2. Loss Functions

2.1 Regression Losses

```
# Loss Functions for Regression
print("Regression Loss Functions")
print("="*70)

def mse_loss(y_true, y_pred):
    """Mean Squared Error"""
    return np.mean((y_true - y_pred) ** 2)

def mae_loss(y_true, y_pred):
    """Mean Absolute Error"""
    return np.mean(np.abs(y_true - y_pred))

def huber_loss(y_true, y_pred, delta=1.0):
    """Huber Loss - less sensitive to outliers"""
    error = y_true - y_pred
    is_small = np.abs(error) <= delta
    squared_loss = 0.5 * error ** 2
    linear_loss = delta * (np.abs(error) - 0.5 * delta)
    return np.mean(np.where(is_small, squared_loss, linear_loss))

# Example
y_true = np.array([1.0, 2.0, 3.0, 4.0, 5.0])
y_pred = np.array([1.1, 2.2, 2.8, 4.5, 4.8])
y_pred_outlier = np.array([1.1, 2.2, 2.8, 4.5, 10.0])  # Outlier

print("Normal predictions:")
print(f"  y_true: {y_true}")
print(f"  y_pred: {y_pred}")
print(f"  MSE:    {mse_loss(y_true, y_pred):.4f}")
print(f"  MAE:    {mae_loss(y_true, y_pred):.4f}")
print(f"  Huber:  {huber_loss(y_true, y_pred):.4f}")

print("\nWith outlier in predictions:")
print(f"  y_pred: {y_pred_outlier}")
print(f"  MSE:    {mse_loss(y_true, y_pred_outlier):.4f} (explodes!)")
print(f"  MAE:    {mae_loss(y_true, y_pred_outlier):.4f}")
print(f"  Huber:  {huber_loss(y_true, y_pred_outlier):.4f} (robust)")

print("\nWhen to use:")
print("  MSE: Standard choice, penalizes large errors")
print("  MAE: When outliers should not dominate")
print("  Huber: Combines MSE (small errors) + MAE (large errors)")
```

```
Regression Loss Functions
======================================================================
Normal predictions:
  y_true: [1. 2. 3. 4. 5.]
  y_pred: [1.1 2.2 2.8 4.5 4.8]
  MSE:    0.0660
  MAE:    0.2200
  Huber:  0.0330

With outlier in predictions:
  y_pred: [ 1.1  2.2  2.8  4.5 10. ]
  MSE:    5.0660 (explodes!)
  MAE:    1.2200
  Huber:  2.2830 (robust)

When to use:
```

MSE: Standard choice, penalizes large errors
MAE: When outliers should not dominate
Huber: Combines MSE (small errors) + MAE (large errors)

2.2 Classification Losses

```
# Classification Loss Functions
print("Classification Loss Functions")
print("="*70)

def binary_cross_entropy(y_true, y_pred, epsilon=1e-15):
    """Binary Cross-Entropy Loss"""
    y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
    return -np.mean(y_true * np.log(y_pred) + (1 - y_true) * np.log(1 - y_pred))

def softmax(z):
    """Softmax function for multi-class"""
    exp_z = np.exp(z - np.max(z, axis=-1, keepdims=True))  # Numerical stability
    return exp_z / np.sum(exp_z, axis=-1, keepdims=True)

def categorical_cross_entropy(y_true, y_pred, epsilon=1e-15):
    """Categorical Cross-Entropy Loss"""
    y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
    return -np.mean(np.sum(y_true * np.log(y_pred), axis=-1))

print("Binary Classification:")
y_true_bin = np.array([1, 0, 1, 1, 0])
y_pred_good = np.array([0.9, 0.1, 0.8, 0.95, 0.2])
y_pred_bad = np.array([0.3, 0.7, 0.4, 0.6, 0.8])

print(f"  True labels:     {y_true_bin}")
print(f"  Good predictions: {y_pred_good}")
print(f"  BCE loss:         {binary_cross_entropy(y_true_bin, y_pred_good):.4f}")
print(f"\n  Bad predictions:  {y_pred_bad}")
print(f"  BCE loss:         {binary_cross_entropy(y_true_bin, y_pred_bad):.4f}")

print("\nMulti-class Classification:")
# 3 samples, 4 classes (one-hot encoded)
y_true_multi = np.array([[1, 0, 0, 0],
                         [0, 1, 0, 0],
                         [0, 0, 0, 1]])
y_pred_multi = np.array([[0.8, 0.1, 0.05, 0.05],
                         [0.1, 0.7, 0.1, 0.1],
                         [0.1, 0.1, 0.2, 0.6]])

print(f"  Softmax outputs:\n{y_pred_multi}")
print(f"  CCE loss: {categorical_cross_entropy(y_true_multi, y_pred_multi):.4f}")
```

```
Classification Loss Functions
======================================================================
Binary Classification:
  True labels:     [1 0 1 1 0]
  Good predictions: [0.9  0.1  0.8  0.95 0.2 ]
  BCE loss:         0.1303

  Bad predictions:  [0.3 0.7 0.4 0.6 0.8]
  BCE loss:         1.0605

Multi-class Classification:
  Softmax outputs:
[[0.8  0.1  0.05 0.05]
 [0.1  0.7  0.1  0.1 ]
 [0.1  0.1  0.2  0.6 ]]
  CCE loss: 0.3448
```

3. Complete Neural Network Forward Pass

3.1 Putting It All Together

```
# Complete Neural Network with Forward Pass
print("Complete Neural Network Class")
print("="*70)

class NeuralNetwork:
    """Neural Network with forward pass implementation"""

    def __init__(self, layer_sizes, activation='relu', output='softmax'):
        self.layer_sizes = layer_sizes
        self.activation = activation
        self.output = output

        # Initialize weights and biases
        self.weights = []
        self.biases = []

        for i in range(len(layer_sizes) - 1):
            # Xavier initialization
            scale = np.sqrt(2.0 / layer_sizes[i])
            W = np.random.randn(layer_sizes[i], layer_sizes[i+1]) * scale
            b = np.zeros(layer_sizes[i+1])
            self.weights.append(W)
            self.biases.append(b)

    def _activate(self, z, is_output=False):
        if is_output:
            if self.output == 'softmax':
                exp_z = np.exp(z - np.max(z, axis=-1, keepdims=True))
                return exp_z / np.sum(exp_z, axis=-1, keepdims=True)
            elif self.output == 'sigmoid':
                return 1 / (1 + np.exp(-z))
            else:
                return z  # Linear
        else:
            if self.activation == 'relu':
                return np.maximum(0, z)
            elif self.activation == 'tanh':
                return np.tanh(z)
            else:
                return 1 / (1 + np.exp(-z))

    def forward(self, X):
        """Forward propagation"""
        self.cache = {'A0': X}
        A = X

        for i, (W, b) in enumerate(zip(self.weights, self.biases)):
            Z = A @ W + b
            is_output = (i == len(self.weights) - 1)
            A = self._activate(Z, is_output)

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

        return A

    def compute_loss(self, y_true, y_pred):
        """Cross-entropy loss"""
        epsilon = 1e-15
```

```
        y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
        return -np.mean(np.sum(y_true * np.log(y_pred), axis=-1))

# Example: MNIST-like classifier
np.random.seed(42)
nn = NeuralNetwork([784, 128, 64, 10], activation='relu', output='softmax')

# Simulate batch of 32 images (28x28 = 784 pixels)
X_batch = np.random.randn(32, 784)
y_batch = np.eye(10)[np.random.randint(0, 10, 32)]  # One-hot labels

# Forward pass
predictions = nn.forward(X_batch)

print(f"Network architecture: {nn.layer_sizes}")
print(f"Input batch shape: {X_batch.shape}")
print(f"Output shape: {predictions.shape}")
print(f"Sample prediction (sums to 1): {predictions[0].round(3)}")
print(f"Sum: {predictions[0].sum():.4f}")
print(f"\nLoss (random weights): {nn.compute_loss(y_batch, predictions):.4f}")
```

```
Complete Neural Network Class
======================================================================
Network architecture: [784, 128, 64, 10]
Input batch shape: (32, 784)
Output shape: (32, 10)
Sample prediction (sums to 1): [0.098 0.102 0.087 0.094 0.121 0.089 0.107 0.098 0.112 0.092]
Sum: 1.0000

Loss (random weights): 2.3847
```

Summary

Key Takeaways:

- Forward pass: input -> linear -> activation -> ... -> output
- Matrix operations enable efficient batch processing
- MSE for regression, cross-entropy for classification
- Softmax converts logits to probabilities for multi-class
- Loss function measures how wrong predictions are
- Cache intermediate values for backpropagation

Practice Exercises:

1. Implement forward pass for a 5-layer network
2. Compare different loss functions on the same data
3. Add dropout to the forward pass
4. Implement temperature scaling for softmax
5. Benchmark batch sizes for forward pass speed