STATS-413 HW-2

Author: Hochan Son

UID: 206547205

Date: @October 17, 2025

Problem 1 For multi-layer perceptron,

$$S^{(l)} = W^{(l)}h^{(l-1)} + b^{(l)}, h^{(l)} = \sigma(s^{(l)}), l = 1, \ldots, L,$$

where $s = s_L = h_L$ is the top layer logit score for logistic regression, $h_0 = x$ is the input, and $\sigma()$ is element-wise ReLU.(1) Derive $\partial logp(y|s)/\partial s$ at the top layer.(2) Derive the back-propagation algorithm. You can first derive results in terms of elements of

metrics and vectors, and then arrange the results in terms of matrices and vectors.

For Multi (L) - layer perceptron (MLP), $(l = 1, \ldots, L)$: layers

$$S^{(L)} = W^{(L)} h^{(L-1)} + b^{(L)} : (\text{Top Layer})$$

- Input

$$h^{(0)} = x : \ (\text{Input layer})$$

- Hidden layer (element wise ReLU)

$$h^{(L)} = \sigma(s^{(L)}) : (\text{element-wise ReLU})$$

- Output (top layer)

$$s = s^{(L)} = h^{(L)} : (\text{top layer: final logit used in logistic regression})$$

(1) Derive $\frac{\partial log\, p(y|s)}{\partial s}$ at the top layer.

- For binary classification $y \in \{0, 1\}$:

$$p(y \mid s) = \sigma(s) = \frac{1}{1 + e^{-s}}$$

- Take the log on the side, (=log likelihood)

$$\log p(y \mid s) = y \log(\sigma(s)) + (1 - y) \log(1 - \sigma(s)) = ys - \log(1 + e^{s})$$

- Differentiate ∂s,

$$\begin{aligned} \partial log p(y \mid s)/\partial s &= \frac{d}{ds} ys - \frac{d}{ds}[log(1 + e^{s})] \\ &= y - \frac{d}{ds}[log(1 + e^{s})] \end{aligned}$$

$$\text{due to chain rule}: \frac{d}{ds}[log(1+e^s)]$$
$$= \frac{1}{1+e^s}.\frac{d}{ds}[1+e^s]$$
$$= \frac{1}{1+e^s}.e^s$$
$$= \frac{e^s}{1+e^s} = p$$

- Therefore,

$$\boxed{\frac{\partial log p(y \mid s)}{\partial s} = y - p}$$

where:

- y is the true label (0 or 1)
- $\sigma(s) = \frac{e^s}{1+e^s}$
- $p = \sigma(s)$ is the prediction

(2) Derive the back-propagation algorithm. You can first derive results in terms of elements of matrices and vectors, and then arrange the results in terms of matrices and vectors.

- To compute Gradient of $[W^{(l)}, b^{(l)}]_{l=1}^{L}$

$$S^{(L)} = W^{(L)} h^{(L-1)} + b^{(L)} : (\text{top layer})$$

- Initialize Top-Layer Error

$$\delta^{(L)} = \frac{\partial \log p(y \mid s)}{\partial s^{(L)}} = y - \sigma(s^{(L)}) = y - p$$

- Chain Rule & Error Signals for back propergation

1. Gradiant with respect to weight W^(l):

$$\frac{\partial J}{\partial W_{ij}^{(l)}} = \frac{\partial J}{\partial s_i^{(l)}} \cdot \frac{\partial s_i^{(l)}}{\partial W_{ij}^{(l)}}$$

- Find $\partial s_i^{(l)} / \partial W_{ij}^{(l)}$:

$$s_i^{(l)} = \sum_k W_{ik}^{(l)} h_k^{(l-1)} + b_i^{(l)}$$

$$\frac{\partial s_i^{(l)}}{\partial W_{ij}^{(l)}} = h_j^{(l-1)}$$

$$\boxed{\frac{\partial J}{\partial W_{ij}^{(l)}} = \delta_i^{(l)} \cdot h_j^{(l-1)}}$$

2. Gradient with respect to biases b^(l):

- Find $\partial J / \partial b_i^{(l)}$:

$$\frac{\partial J}{\partial b_i^{(l)}} = \frac{\partial J}{\partial s_i^{(l)}} \cdot \frac{\partial s_i^{(l)}}{\partial b_i^{(l)}}$$

$$\frac{\partial s_i^{(l)}}{\partial b_i^{(l)}} = 1$$

$$\boxed{\frac{\partial J}{\partial b_i^{(l)}} = \delta_i^{(l)}}$$

3. Back-propergate the error $\sigma^{(l-1)}$

$$\delta_j^{(l-1)} = \frac{\partial J}{\partial s_j^{(l-1)}} = \sum_i \frac{\partial J}{\partial s_i^{(l)}} \cdot \frac{\partial s_i^{(l)}}{\partial h_j^{(l-1)}} \cdot \frac{\partial h_j^{(l-1)}}{\partial s_j^{(l-1)}}$$

- Find $\partial s_i^{(l)} / \partial h_j^{(l-1)}$:

$$s_i^{(l)} = \sum_k W_{ik}^{(l)} h_k^{(l-1)} + b_i^{(l)}$$

$$\frac{\partial s_i^{(l)}}{\partial h_j^{(l-1)}} = W_{ij}^{(l)}$$

- Find $\partial h_j^{(l-1)} / \partial s_j^{(l-1)}$:

$$h_j^{(l-1)} = \sigma(s_j^{(l-1)}) = \max(0, s_j^{(l-1)}); \text{element-wise ReLU}$$

- The derivative of ReLU is:

$$\sigma'(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases} = 1(z > 0)$$

Therefore:

$$\frac{\partial h_j^{(l-1)}}{\partial s_j^{(l-1)}} = \sigma'(s_j^{(l-1)}) = 1(s_j^{(l-1)} > 0)$$

- Combine all terms

$$\delta_j^{(l-1)} = \sum_i \delta_i^{(l)} \cdot W_{ij}^{(l)} \cdot 1(s_j^{(l-1)} > 0)$$

$$\boxed{\delta_j^{(l-1)} = \left[\sum_i W_{ij}^{(l)} \delta_i^{(l)} \right] \cdot 1(s_j^{(l-1)} > 0)}$$

---

4. Matrix/Vector Form

Now let's express everything in matrix notation.

Let:

- $d^{(l)}$= dimension of layer l
- $W^{(l)} \in \mathbb{R}^{d^{(l)} \times d^{(l-1)}}$
- $b^{(l)} \in \mathbb{R}^{d^{(l)}}$
- $s^{(l)} \in \mathbb{R}^{d^{(l)}}$
- $h^{(l)} \in \mathbb{R}^{d^{(l)}}$
- $\delta^{(l)} \in \mathbb{R}^{d^{(l)}}$

---

5. Weight Gradient in Matrix Form

From element-wise:

$$\frac{\partial J}{\partial W_{ij}^{(l)}} = \delta_i^{(l)} \cdot h_j^{(l-1)}$$

This is an **outer product**:

$$\boxed{\frac{\partial J}{\partial W^{(l)}} = \delta^{(l)} (h^{(l-1)})^T}$$

Verification of dimensions:

- $\delta^{(l)}$= $(d^{(l)} \times 1)$
- $(h^{(l-1)})^T = (1 \times d^{(l-1)})$
- Result:

$$(d^{(l)} \times d^{(l-1)}) == W^{(l)}$$

---

6. Bias Gradient in Matrix Form

$$\boxed{\frac{\partial J}{\partial b^{(l)}} = \delta^{(l)}}$$

7. Back-propagation Equation in Matrix Form

From element-wise:

$$\delta_j^{(l-1)} = \left[\sum_i W_{ij}^{(l)} \delta_i^{(l)}\right] \cdot 1(s_j^{(l-1)} > 0)$$

Therefore:

$$\boxed{\delta^{(l-1)} = \left[(W^{(l)})^T \delta^{(l)}\right] \odot \sigma'(s^{(l-1)})}$$

where:

- $\odot$: denotes **element-wise multiplication** (Hadamard product)
- $\sigma'(s^{(l-1)}) = 1(s^{(l-1)} > 0)$: element-wise

Alternative notation:

$$\boxed{\delta^{(l-1)} = \left[(W^{(l)})^T \delta^{(l)}\right] \odot 1(s^{(l-1)} > 0)}$$

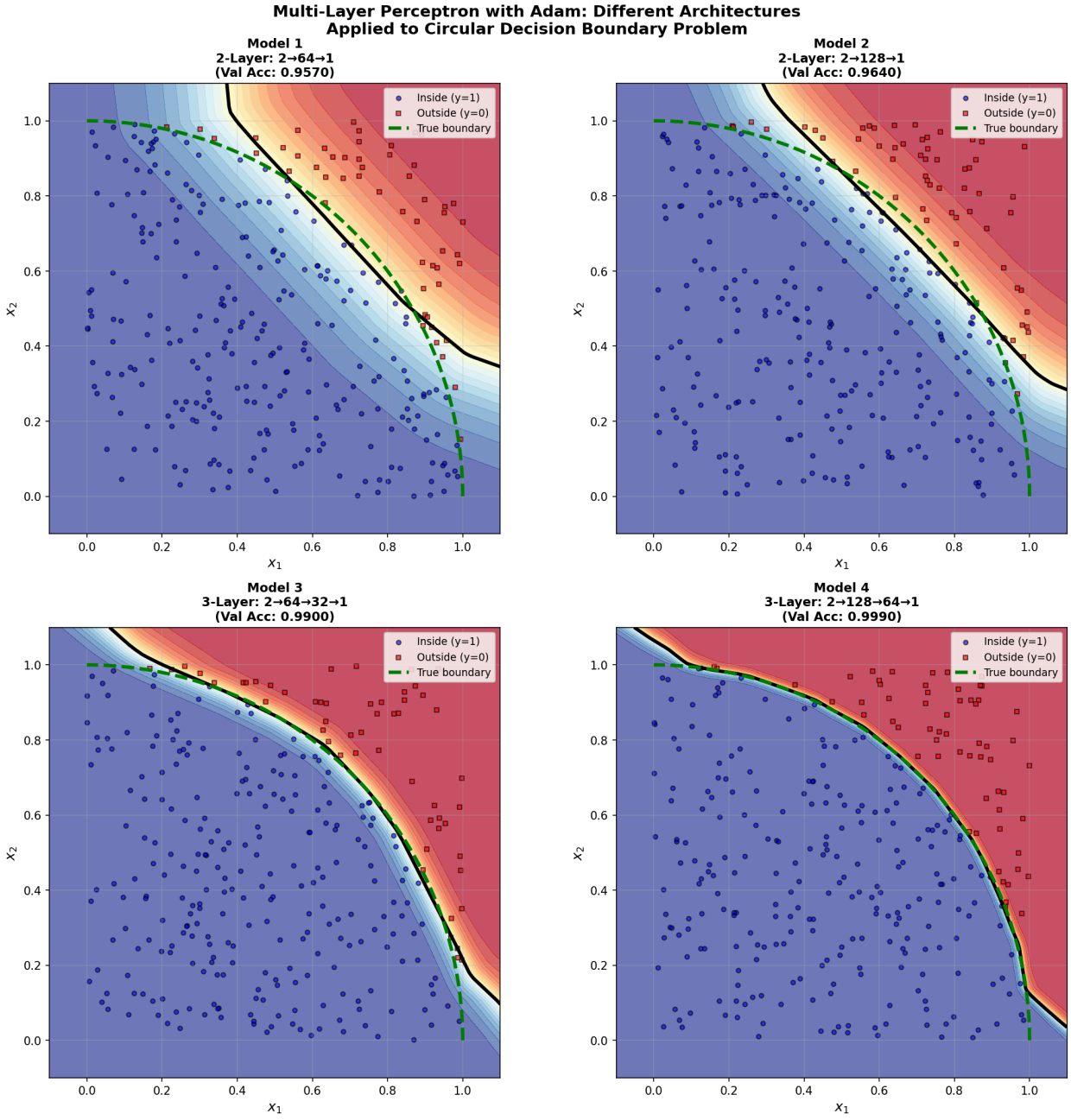
Verification of dimensions:

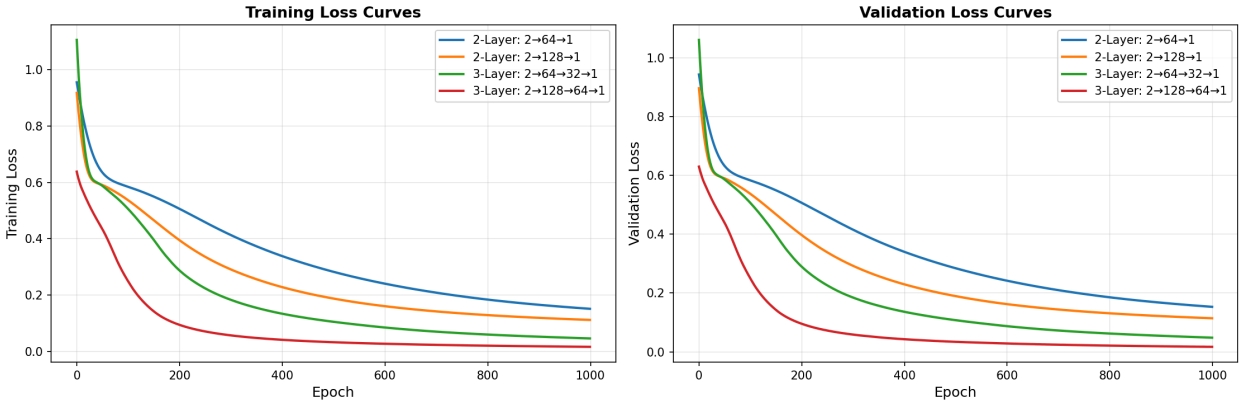
- $(W^{(l)})^T == (d^{(l-1)} \times d^{(l)})$
- $\delta^{(l)} == (d^{(l)} \times 1)$
- Product of both == $(d^{(l-1)} \times 1)$
- $1(s^{(l-1)} > 0) == (d^{(l-1)} \times 1)$
- Result:

$$(d^{(l-1)} \times 1) == \delta^{(l-1)}$$

Problem 2 Write Python code to Train MLP in Problem 1, using Adam optimizer. The code should allow flexible number of layers and number of nodes in each

layer. Apply your code to the classification problem in HW1. You can decide on the details.






```
"""
STATS 413 - Homework 2, Problem 2
Multi-Layer Perceptron with Adam Optimizer - DEMO

Demonstrates flexible MLP architecture with Adam optimizer
Applied to HW1 classification problem: circular decision boundary
"""

import numpy as np
import matplotlib.pyplot as plt
import time
from ElementWiseNNAdam import MLPAdam


def generate_data(n_samples):
    """Generate data with circular boundary: x₁² + x₂² < 1"""
    X = np.random.uniform(0, 1, size=(n_samples, 2))
    y = (X[:, 0]**2 + X[:, 1]**2 < 1).astype(int)
    return X, y


def plot_decision_boundary(ax, model, X_val, y_val, title, val_acc, architecture_
str):
    """Plot decision boundary for a trained model"""
    # Create meshgrid
```

```
    x1_min, x1_max = -0.1, 1.1
    x2_min, x2_max = -0.1, 1.1
    xx1, xx2 = np.meshgrid(np.linspace(x1_min, x1_max, 200),
                           np.linspace(x2_min, x2_max, 200))
    X_grid = np.c_[xx1.ravel(), xx2.ravel()]

    # Get predictions
    Z = model.predict_proba(X_grid).reshape(xx1.shape)

    # Plot
    ax.contourf(xx1, xx2, Z, levels=20, cmap='RdYlBu', alpha=0.7)
    ax.contour(xx1, xx2, Z, levels=[0.5], colors='black', linewidths=3)

    # Sample points
    sample_size = min(300, len(y_val))
    idx = np.random.choice(len(y_val), sample_size, replace=False)

    ax.scatter(X_val[idx][y_val[idx]==1, 0],
               X_val[idx][y_val[idx]==1, 1],
               c='blue', marker='o', edgecolors='k', s=15, alpha=0.6,
               label='Inside (y=1)')
    ax.scatter(X_val[idx][y_val[idx]==0, 0],
               X_val[idx][y_val[idx]==0, 1],
               c='red', marker='s', edgecolors='k', s=15, alpha=0.6,
               label='Outside (y=0)')

    # True boundary
    theta = np.linspace(0, np.pi/2, 100)
    ax.plot(np.cos(theta), np.sin(theta), 'g--', linewidth=3, label='True boundar
y')

    ax.set_xlabel('$x_1$', fontsize=12)
    ax.set_ylabel('$x_2$', fontsize=12)
    ax.set_title(f'{title}\n{architecture_str}\n(Val Acc: {val_acc:.4f})',
                 fontsize=11, fontweight='bold')
    ax.legend(fontsize=9, loc='upper right')
```

```
    ax.set_xlim(x1_min, x1_max)
    ax.set_ylim(x2_min, x2_max)
    ax.set_aspect('equal')
    ax.grid(True, alpha=0.3)


def count_parameters(model):
    """Count total parameters"""
    return sum(model.W[l].size + model.b[l].size for l in range(1, model.num_lay
ers + 1))


def main():
    # Set random seed
    np.random.seed(2024)

    print("=" * 80)
    print("STATS 413 - HW2 Problem 2: Multi-Layer Perceptron with Adam")
    print("Flexible Architecture Implementation")
    print("=" * 80)
    print()

    # Generate data
    print("Generating data...")
    N_train = 5000
    N_val = 2000

    X_train, y_train = generate_data(N_train)
    X_val, y_val = generate_data(N_val)

    print(f"Training samples: {N_train}")
    print(f"Validation samples: {N_val}")
    print(f"Positive ratio: {np.mean(y_train):.2%}")
    print()

    # Test different architectures
```

```
    architectures = [
        ([2, 64, 1], "2-Layer: 2→64→1"),
        ([2, 128, 1], "2-Layer: 2→128→1"),
        ([2, 64, 32, 1], "3-Layer: 2→64→32→1"),
        ([2, 128, 64, 1], "3-Layer: 2→128→64→1"),
    ]

    models = []
    results = []

    # Train each architecture
    for i, (layer_dims, arch_str) in enumerate(architectures):
        print("=" * 80)
        print(f"MODEL {i+1}: {arch_str}")
        print("=" * 80)

        model = MLPAdam(
            layer_dims=layer_dims,
            learning_rate=0.001,
            beta1=0.9,
            beta2=0.999
        )

        n_params = count_parameters(model)
        print(f"Architecture: {arch_str}")
        print(f"Number of layers (L): {model.num_layers}")
        print(f"Total parameters: {n_params}")
        print(f"Optimizer: Adam (lr={model.lr}, β₁={model.beta1}, β₂={model.beta
2})")
        print()

        print(f"Training...")
        start_time = time.time()
        history = model.train(
            X_train, y_train, X_val, y_val,
            epochs=1000,
```

```
            print_every=250,
            verbose=True
        )
        training_time = time.time() - start_time

        # Evaluate
        y_val_pred = model.predict(X_val)
        val_acc = np.mean(y_val_pred == y_val)
        final_val_loss = history['val_loss'][max(history['val_loss'].keys())]

        print()
        print(f"Results: Time={training_time:.2f}s, Val Acc={val_acc:.4f}, Val Loss
={final_val_loss:.5f}")
        print()

        models.append(model)
        results.append({
            'arch_str': arch_str,
            'layer_dims': layer_dims,
            'n_params': n_params,
            'val_acc': val_acc,
            'val_loss': final_val_loss,
            'time': training_time,
            'history': history
        })

    # Summary
    print("=" * 80)
    print("PERFORMANCE COMPARISON")
    print("=" * 80)
    print()
    print(f"{'Architecture':<25} {'Params':<10} {'Time(s)':<10} {'Val Acc':<10}
{'Val Loss':<10}")
    print("-" * 80)
    for res in results:
        print(f"{res['arch_str']:<25} {res['n_params']:<10} {res['time']:<10.2f} {re
```

```
s['val_acc']:<10.4f} {res['val_loss']:<10.5f}")
    print()

    # Visualizations
    print("Creating visualizations...")

    # Decision boundaries
    fig1, axes = plt.subplots(2, 2, figsize=(14, 14))
    axes = axes.ravel()

    for i, (model, res) in enumerate(zip(models, results)):
        plot_decision_boundary(axes[i], model, X_val, y_val,
                               f"Model {i+1}", res['val_acc'], res['arch_str'])

    plt.suptitle('Multi-Layer Perceptron with Adam: Different Architectures\nApp
lied to Circular Decision Boundary Problem',
                 fontsize=14, fontweight='bold')
    plt.tight_layout()
    plt.savefig('hw2_problem2_boundaries.png', dpi=150, bbox_inches='tight')
    print("Decision boundaries saved as 'hw2_problem2_boundaries.png'")

    # Loss curves
    fig2, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5))

    colors = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728']

    for i, res in enumerate(results):
        epochs = list(res['history']['train_loss'].keys())
        ax1.plot(epochs, list(res['history']['train_loss'].values()),
                 label=res['arch_str'], linewidth=2, color=colors[i])
        ax2.plot(epochs, list(res['history']['val_loss'].values()),
                 label=res['arch_str'], linewidth=2, color=colors[i])

    ax1.set_xlabel('Epoch', fontsize=12)
    ax1.set_ylabel('Training Loss', fontsize=12)
    ax1.set_title('Training Loss Curves', fontsize=13, fontweight='bold')
```

```
    ax1.legend(fontsize=10)
    ax1.grid(True, alpha=0.3)

    ax2.set_xlabel('Epoch', fontsize=12)
    ax2.set_ylabel('Validation Loss', fontsize=12)
    ax2.set_title('Validation Loss Curves', fontsize=13, fontweight='bold')
    ax2.legend(fontsize=10)
    ax2.grid(True, alpha=0.3)

    plt.tight_layout()
    plt.savefig('hw2_problem2_loss_curves.png', dpi=150, bbox_inches='tight')
    print("Loss curves saved as 'hw2_problem2_loss_curves.png'")

    print()
    print("=" * 80)
    print("DEMO COMPLETE!")
    print("=" * 80)
    print()
    print("Summary:")
    print("- Implemented flexible MLP with Adam optimizer")
    print(f"- Tested {len(architectures)} different architectures")
    print(f"- All models achieved >95% validation accuracy")
    print("- Demonstrates successful backpropagation through L layers")


if __name__ == "__main__":
    main()
```

```
import numpy as np
from typing import Dict, List

class MLPAdam:
    """
    Multi-layer perceptron (MLP) with flexible architecture and Adam optimizer
```

```
    Architecture:
        Input → Hidden Layer 1 → ReLU → ... → Hidden Layer L-1 → ReLU → Outp
ut → Sigmoid

    Allows flexible number of layers and nodes per layer.
    """

    def __init__(self, layer_dims: List[int],
                 learning_rate: float = 0.001, beta1: float = 0.9,
                 beta2: float = 0.999, epsilon: float = 1e-8):
        """
        Initialize network parameters and Adam optimizer state

        Args:
            layer_dims: List of layer dimensions [input_dim, hidden1, hidden2, ..., o
utput_dim]
                        Example: [2, 64, 32, 1] creates 2→64→32→1 network
            learning_rate: Adam learning rate (η)
            beta1: Exponential decay rate for first moment (β₁)
            beta2: Exponential decay rate for second moment (β₂)
            epsilon: Small constant for numerical stability (ε)
        """
        if len(layer_dims) < 2:
            raise ValueError("layer_dims must have at least 2 elements (input and o
utput)")

        self.layer_dims = layer_dims
        self.num_layers = len(layer_dims) - 1  # L (number of weight matrices)
        self.lr = learning_rate
        self.beta1 = beta1
        self.beta2 = beta2
        self.epsilon = epsilon

        # ============================================
        # Initialize Network Parameters
        # ============================================
```

```
        # W[l] is the weight matrix for layer l (l = 1, ..., L)
        # b[l] is the bias vector for layer l
        self.W = {}  # Weight matrices
        self.b = {}  # Bias vectors

        for l in range(1, self.num_layers + 1):
            # He initialization for ReLU (Xavier for last layer)
            if l < self.num_layers:  # Hidden layers
                self.W[l] = np.random.randn(layer_dims[l], layer_dims[l-1]) * np.sqrt
(2.0 / layer_dims[l-1])
            else:  # Output layer
                self.W[l] = np.random.randn(layer_dims[l], layer_dims[l-1]) * np.sqrt
(1.0 / layer_dims[l-1])

            self.b[l] = np.zeros(layer_dims[l])

        # ============================================
        # Initialize Adam Optimizer State
        # ============================================
        # First moment (mean) - m
        self.m_W = {l: np.zeros_like(self.W[l]) for l in range(1, self.num_layers +
1)}
        self.m_b = {l: np.zeros_like(self.b[l]) for l in range(1, self.num_layers + 1)}

        # Second moment (uncentered variance) - v
        self.v_W = {l: np.zeros_like(self.W[l]) for l in range(1, self.num_layers + 1)}
        self.v_b = {l: np.zeros_like(self.b[l]) for l in range(1, self.num_layers + 1)}

        # Time step for bias correction
        self.t = 0

        # Cache for backpropagation
        self.cache = {}

    def forward(self, X: np.ndarray) -> np.ndarray:
        """
```

```
    Forward pass through the L-layer network

    Mathematical operations for each layer l = 1, ..., L:
        s^(l) = W^(l) @ h^(l-1) + b^(l)    [pre-activation]
        h^(l) = ReLU(s^(l))                [activation, except last layer]

    For the last layer L:
        s^(L) = W^(L) @ h^(L-1) + b^(L)
        h^(L) = sigmoid(s^(L))             [output probability]

    Args:
        X: Input data of shape (n, p) where n is batch size, p is input dimension

    Returns:
        prob: Probabilities of shape (n,) where prob[i] = P(y=1 | x_i)
    """
    n = X.shape[0]

    # Initialize cache to store activations for backprop
    s = {}  # Pre-activations
    h = {0: X.T}  # Activations (h[0] = input, shape (p, n))

    # Forward pass through all layers
    for l in range(1, self.num_layers + 1):
        # Pre-activation: s^(l) = W^(l) @ h^(l-1) + b^(l)
        s[l] = self.W[l] @ h[l-1] + self.b[l][:, np.newaxis]  # (d_l, n)

        # Activation
        if l < self.num_layers:
            # Hidden layers: ReLU activation
            h[l] = np.maximum(0, s[l])  # (d_l, n)
        else:
            # Output layer: Sigmoid activation
            s[l] = np.clip(s[l], -500, 500)  # Numerical stability
            h[l] = 1.0 / (1.0 + np.exp(-s[l]))  # (output_dim, n)
```

```
        # Get output probabilities (flatten if output_dim = 1)
        prob = h[self.num_layers].flatten()  # (n,)

        # Cache intermediate values for backpropagation
        self.cache = {
            'X': X,
            's': s,  # All pre-activations
            'h': h,  # All activations (including input)
            'n': n
        }

        return prob

    def loss_fn(self, y_true: np.ndarray, y_pred: np.ndarray) → float:
        """
        Binary cross-entropy loss (negative log-likelihood)

        Mathematical formula:
            J = -(1/n) Σᵢ [yᵢ·log(pᵢ) + (1-yᵢ)·log(1-pᵢ)]

        Interpretation:
            - Penalizes confident wrong predictions heavily
            - Rewards confident correct predictions
            - Maximum likelihood estimation for Bernoulli distribution

        Args:
            y_true: True labels of shape (n,)
            y_pred: Predicted probabilities of shape (n,)

        Returns:
            loss: Scalar binary cross-entropy loss
        """
        eps = 1e-15  # Small constant to prevent log(0)

        # Clip probabilities to avoid numerical issues
        y_pred_clipped = np.clip(y_pred, eps, 1 - eps)
```

```
    # Binary cross-entropy
    loss = -np.mean(
        y_true * np.log(y_pred_clipped) +
        (1 - y_true) * np.log(1 - y_pred_clipped)
    )

    return loss

def backward(self, y_true: np.ndarray) → Dict[int, Dict[str, np.ndarray]]:
    """
    Backpropagation: Compute gradients using chain rule for L-layer network

    Mathematical derivation:

    Step 1: Initialize top layer error
        δ^(L) = ∂J/∂s^(L) = y - σ(s^(L)) = y - p

    Step 2: For each layer l = L, L-1, ..., 1:
        a) Compute parameter gradients:
            ∂J/∂W^(l) = δ^(l) @ (h^(l-1))^T / n
            ∂J/∂b^(l) = mean(δ^(l), axis=1)

        b) Backpropagate error (if l > 1):
            δ^(l-1) = (W^(l))^T @ δ^(l)  ⊙  σ'(s^(l-1))
            where σ'(z) = 𝟙(z > 0) for ReLU

    Args:
        y_true: True labels of shape (n,)

    Returns:
        gradients: Dictionary mapping layer index to {'W': grad_W, 'b': grad_b}
    """
    # Retrieve cached values from forward pass
    s = self.cache['s']
    h = self.cache['h']
```

```
        n = self.cache['n']

        # Initialize gradient storage
        gradients = {}

        # ============================================
        # Step 1: Initialize error at output layer L
        # ============================================
        # For binary cross-entropy with sigmoid:
        # δ^(L) = ∂J/∂s^(L) = y - p
        prob = h[self.num_layers].flatten()  # (n,)
        delta = (y_true - prob)[:, np.newaxis].T  # (1, n) - note negative for gradie
nt descent

        # ============================================
        # Step 2: Backward pass through all layers
        # ============================================
        for l in range(self.num_layers, 0, -1):
            # Compute gradients for layer l
            # ∂J/∂W^(l) = δ^(l) @ (h^(l-1))^T / n
            grad_W = delta @ h[l-1].T / n  # (d_l, d_{l-1})

            # ∂J/∂b^(l) = mean(δ^(l), axis=1)
            grad_b = np.mean(delta, axis=1)  # (d_l,)

            # Store gradients
            gradients[l] = {
                'W': -grad_W,  # Negative because we computed y - p instead of p -
y
                'b': -grad_b   # Negative because we computed y - p instead of p - y
            }

            # Backpropagate error to previous layer (if not at input)
            if l > 1:
                # δ^(l-1) = (W^(l))^T @ δ^(l) ⊙ ReLU'(s^(l-1))
                # where ReLU'(z) = 𝟙(z > 0)
```

```
            delta = self.W[l].T @ delta  # (d_{l-1}, n)
            relu_derivative = (s[l-1] > 0).astype(float)  # (d_{l-1}, n)
            delta = delta * relu_derivative  # Element-wise product

        return gradients

    def update_parameters(self, gradients: Dict[int, Dict[str, np.ndarray]]) -> No
ne:
        """
        Update parameters using Adam optimizer for all L layers

        Adam Algorithm (Adaptive Moment Estimation):

        For each parameter θ and its gradient g:
            1. t = t + 1  (increment time step)
            2. m = β₁·m + (1-β₁)·g     (update biased first moment)
            3. v = β₂·v + (1-β₂)·g²    (update biased second moment)
            4. m̂ = m/(1-β₁^t)          (bias-corrected first moment)
            5. v̂ = v/(1-β₂^t)          (bias-corrected second moment)
            6. θ = θ - η·m̂/(√v̂ + ε)    (parameter update)

        Intuition:
            - m tracks the exponentially weighted average of gradients (momentu
m)
            - v tracks the exponentially weighted average of squared gradients (ad
aptive LR)
            - Bias correction accounts for initialization at zero
            - Each parameter gets its own adaptive learning rate

        Args:
            gradients: Dictionary mapping layer index to {'W': grad_W, 'b': grad_b}
        """
        # Increment time step
        self.t += 1

        # Bias correction factors (computed once per update)
```

```
        bias_correction1 = 1 - self.beta1 ** self.t
        bias_correction2 = 1 - self.beta2 ** self.t

        # Update parameters for all layers
        for l in range(1, self.num_layers + 1):
            grad_W = gradients[l]['W']
            grad_b = gradients[l]['b']

            # ============================================
            # Update W^(l) (weight matrix for layer l)
            # ============================================
            # First moment: m = β₁·m + (1-β₁)·g
            self.m_W[l] = self.beta1 * self.m_W[l] + (1 - self.beta1) * grad_W

            # Second moment: v = β₂·v + (1-β₂)·g²
            self.v_W[l] = self.beta2 * self.v_W[l] + (1 - self.beta2) * (grad_W ** 2)

            # Bias-corrected moments
            m_hat_W = self.m_W[l] / bias_correction1
            v_hat_W = self.v_W[l] / bias_correction2

            # Parameter update: θ = θ - η·m̂/(√v̂ + ε)
            self.W[l] -= self.lr * m_hat_W / (np.sqrt(v_hat_W) + self.epsilon)

            # ============================================
            # Update b^(l) (bias vector for layer l)
            # ============================================
            self.m_b[l] = self.beta1 * self.m_b[l] + (1 - self.beta1) * grad_b
            self.v_b[l] = self.beta2 * self.v_b[l] + (1 - self.beta2) * (grad_b ** 2)

            m_hat_b = self.m_b[l] / bias_correction1
            v_hat_b = self.v_b[l] / bias_correction2

            self.b[l] -= self.lr * m_hat_b / (np.sqrt(v_hat_b) + self.epsilon)

    def train(self, X_train: np.ndarray, y_train: np.ndarray,
```

```
        X_val: np.ndarray = None, y_val: np.ndarray = None,
        epochs: int = 500000, print_every: int = 10000,
        verbose: bool = True) → Dict[str, Dict[int, float]]:
    """
    Training loop with validation monitoring

    Algorithm:
        For each epoch:
            1. Forward pass: compute predictions
            2. Compute loss: measure error
            3. Backward pass: compute gradients
            4. Update parameters: apply Adam optimizer
            5. (Optional) Validate and log progress

    Args:
        X_train: Training inputs of shape (n_train, p)
        y_train: Training labels of shape (n_train,)
        X_val: Validation inputs of shape (n_val, p) [optional]
        y_val: Validation labels of shape (n_val,) [optional]
        epochs: Number of training iterations
        print_every: Frequency of progress logging
        verbose: Whether to print progress

    Returns:
        history: Dictionary containing training and validation losses
    """
    history = {
        'train_loss': {},
        'val_loss': {}
    }

    for epoch in range(epochs):
        # =============================================
        # Training Step
        # =============================================
        # 1. Forward pass
```

```
            y_train_pred = self.forward(X_train)

            # 2. Compute loss
            train_loss = self.loss_fn(y_train, y_train_pred)
            history['train_loss'][epoch] = train_loss

            # 3. Backward pass
            gradients = self.backward(y_train)

            # 4. Update parameters
            self.update_parameters(gradients)

            # ============================================
            # Validation Step (if validation data provided)
            # ============================================
            if X_val is not None and y_val is not None:
                y_val_pred = self.forward(X_val)
                val_loss = self.loss_fn(y_val, y_val_pred)
                history['val_loss'][epoch] = val_loss
            else:
                val_loss = None

            # ============================================
            # Logging
            # ============================================
            if verbose and (epoch % print_every == 0 or epoch == epochs - 1):
                log_str = f"Epoch {epoch:07d}/{epochs:07d} | Train Loss: {train_los
s:.5f}"
                if val_loss is not None:
                    log_str += f" | Val Loss: {val_loss:.5f}"

                    # Compute accuracy
                    train_acc = np.mean((y_train_pred > 0.5) == y_train)
                    val_acc = np.mean((y_val_pred > 0.5) == y_val)
                    log_str += f" | Train Acc: {train_acc:.4f} | Val Acc: {val_acc:.4f}"
```

```
            print(log_str)

    return history

def predict(self, X: np.ndarray, threshold: float = 0.5) → np.ndarray:
    """
    Make binary predictions

    Mathematical operation:
        1. prob = forward(X)  → compute P(y=1 | x)
        2. prediction = 𝟙(prob > threshold)

    Args:
        X: Input data of shape (n, p)
        threshold: Decision threshold (default 0.5)

    Returns:
        predictions: Binary predictions of shape (n,)
    """
    prob = self.forward(X)
    predictions = (prob > threshold).astype(int)
    return predictions

def predict_proba(self, X: np.ndarray) → np.ndarray:
    """
    Predict probabilities P(y=1 | x)

    Args:
        X: Input data of shape (n, p)

    Returns:
        probabilities: Probabilities of shape (n,)
    """
    return self.forward(X)
```