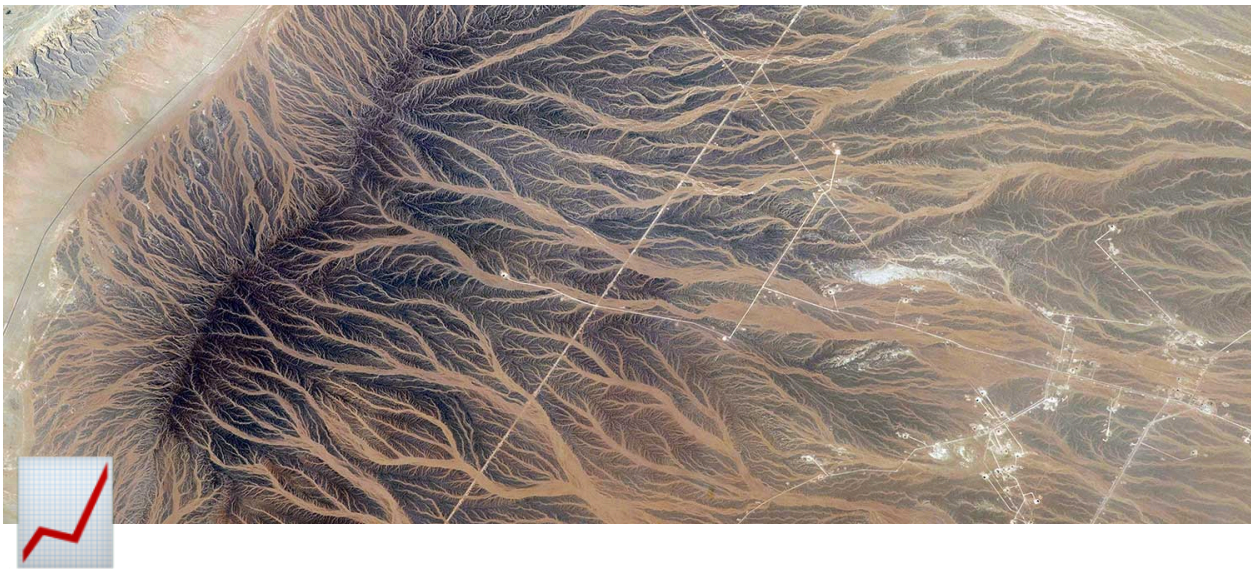


STATS-413 HW-1

Author: Hochan Son

UID: 206547205

Date: @October 10, 2025

Problem 1: For linear and logistic regressions,

assuming $s_i = \sum_j \beta_j x_{ij} = x_i^T \beta$.

(1) Find the log-likelihood function

$$J(\beta) = \sum_{i=1}^{n} log p(y_i | s_i = x_i^T \beta)$$

For a single observation, the likelihood is, where $\sigma = \text{sigmoid function}$:

$$p(y_i | s_i) = \sigma(s_i) = \frac{e^{y_i s_i}}{1 + e^{s_i}}$$

Taking the log, for the chain rule:

$$logp(y_i|s_i) = y_i s_i - log(1 + e^{s_i})$$

for $s_i = x_i^T \beta$,

$$logp(y_i|s_i = x_i^T \beta) = y_i x_i^T \beta - log(1 + e^{x_i^T \beta})$$

MLE (maximum log likelihood):

$$\sum_{i=1}^{n} logp(y_i|s_i = x_i^T \beta) = \sum_{i=1}^{n} y_i x_i^T \beta - log(1 + e^{x_i^T \beta}) \tag{1}$$

The log likelihood function:

$$J(\beta) = \sum_{i=1}^{n} [logp(y_i|s_i = x_i^T \beta)] = \sum [y_i s_i - log(1 + e^{s_i})] \tag{2}$$

Plug $s_i = x_i^T \beta$ into (2),

$$J(\beta) = \sum_{i=1}^{n} logp(y_i|s_i = x_i^T \beta) = \sum_{i=1}^{n} y_i x_i^T \beta - log(1 + e^{x_i^T \beta}) \tag{3}$$

Therefore, (1) == (3)

(2) Find the gradient $J'(\beta)$ by calculating $\frac{dJ}{d\beta}$, and write down the gradient ascent algorithm.

$$J'(\beta) = \frac{d}{d\beta} \sum_{i=1}^{j} logp(y_i|s_i = x_i^T \beta)$$
$$= \frac{d}{d\beta} \sum_{i=1}^{n} y_i x_i^T \beta - log(1 + e^{x_i^T \beta}),$$
$$\text{where } y_i = const$$

$$= y_i \frac{d}{d\beta} \sum_{i=1}^{n} x_i^T \beta - \frac{dJ}{d\beta} log(1 + e^{x_i^T \beta}) \tag{4}$$

Let's substitute $(1) : u = {x_i}^T \beta, (2) : v = e^u, (3) : w = \log(1 + v)$, chain rule:

$$\frac{d}{d\beta} log(1 + e^{{x_i}^T \beta})$$
$$= \frac{dw}{dv} * \frac{dv}{du} * \frac{du}{d\beta}$$

For (1),

$$\frac{du}{d\beta} x_i^T \beta = \frac{du}{d\beta}(x_{i1}\beta + x_{i2}\beta + ... + x_{ip}\beta) = x_i \ (scalar)$$

For (2),

$$\frac{dv}{du} v = \frac{dv}{du} e^u = e^u$$
$$= e^{x_i^T \beta}$$

For (3),

$$\frac{dw}{dv} log(1 + v) = \frac{1}{1 + v} \times \frac{dw}{dv}(1 + v) = \frac{1}{1 + v} \times 1 = \frac{1}{1 + v}$$
$$\text{since } v = e^{x_i},$$
$$= \frac{1}{1 + e^{x_i^T \beta}}$$

To combine (1) (2), and (3),

$$(1)(2)(3) = \frac{1}{1 + e^{x_i^T \beta}} \times e^{x_i^T \beta} \times x_i$$
$$= \frac{e^{x_i^T \beta}}{1 + e^{x_i^T \beta}} \times x_i$$

Let's substitute $x_i^T \beta$ to α:

$$= \frac{e^{\alpha}}{1+e^{\alpha}} x_i = \text{sigmoid function}(\sigma)$$
$$= \sigma(\alpha) x_i$$

$$\text{ans} = \sigma(x_i^T \beta) x_i$$
$$(since\ s_i = x_i^T \beta),$$
$$= s_i x_i$$

Problem 2: For logistic regression based on a simple neural network, with

$$s_i = \beta_0 + \sum_{k=1}^{d} \beta_k h_{ik}, \tag{5}$$
$$h_{ik} = \max(0, s_{ik}), -ReLU \tag{6}$$
$$s_{ik} = \alpha_{k0} + \sum_{j=1}^{p} \alpha_{kj} x_{ij} \tag{7}$$

(1) Using one-dimensional and two-dimensional input x_i, explain how the above neural network defines a piecewise linear function $s_i = f(x_i)$

For (5) is a linear function, if h_{ik} is linear,

$$s_i = \beta_o + \beta_i h_{i1} + \beta_2 h_{i2} + ... + \beta_d h_{id}$$

However, for (6), given h_{ik} is ReLU (Rectified Linear Unit), it is conditional linear (or a.k.a. piecewise linear)

$$\text{When } s_{ik} > 0, s_{ik} \text{ is linear, otherwise } 0$$

Given that the conditional linearity makes it work like a piecewise linear function with respect to $s_i = f(x_i)$

(2) Explain how the last line can be expressed in terms of vectors and matrices.

For (7),

$$s_{ik} = \alpha_{k0} + \sum_{j=1}^{p} \alpha_{kj} x_{ij}$$
$$= a_{k0} + a_{k1}x_{i1} + a_{k2}x_{i2} + ... + a_{kp}x_{ip}$$

It can be simplified to

$$Let\ \alpha_0 = \begin{bmatrix} \alpha_0 & \alpha_1 & \alpha_2 & ... & \alpha_p \end{bmatrix}; (p \times 1\ matrix)$$

$$Let\ A^T = \begin{bmatrix} a_{00} & a_1 & a_2 & ... & a_p \\ a_{01} & & & & \\ & & & & \end{bmatrix}; (d \times p\ matrix)$$

$$Let\ X = \begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ ... \\ X_p \end{bmatrix},\ (1 \times p\ matrix)$$

$$s_{ik} = \alpha_{k0} + 1\sum_{j=1}^{p} \alpha_{kj} x_{ij} = \alpha_{k0} + \alpha_k^T x_i = \alpha_0 + A^T X$$

(3) Let $\theta = (\beta_k, \alpha_{kj}, \forall_k, j)$. Find the gradient $J'(\theta)$ by calculating $\frac{\partial J}{\partial \beta k}\ and\ \frac{\partial J}{\partial \alpha kj}$, and write down the gradient ascent algorithm with momentum.

1. Calculate $\frac{\partial J}{\partial \beta_k}$, using chain rule:
    - to find $\frac{\partial J}{\partial s_i}$, where p_i is the predicted probability

$$\frac{\partial J}{\partial \beta_k} = \sum_{i=1}^{n} \frac{\partial J}{\partial s_i} \times \frac{\partial S_i}{\partial \beta_k}$$

$$\frac{\partial J}{\partial s_i} = y_i - \frac{e_i^s}{1+e_i^s} = y_i - \sigma(s_i) = y_i - p_i$$

- to find $\frac{\partial s_i}{\partial \beta_k}$:

$$\frac{\partial s_i}{\partial \beta_k} = h_{ik}$$

$$\text{therefore,}$$

$$\frac{\partial J}{\partial \beta_k} = \sum_{i=1}^{n} (y_i - p_i) h_{ik}$$

- For h_ik (= ReLU),

$$h_{ik} = max\{0, \alpha_{k0} + \sum_{j=1}^{p} \alpha_{kj} x_{ij}\}$$

- For β_0,

$$\frac{\partial J}{\partial \beta_k} = \sum_{i} (y_i - p_i)$$

- For B_k,

$$\frac{\partial J}{\partial \beta_k} = \sum_{i=1}^{n} (y_i - p_i) max\{0, \alpha_{k0} + \sum_{j=1}^{p} \alpha_{kj} x_{ij}\}$$

2. Calculate $\frac{\partial J}{\partial \alpha_{kj}}$, using chain rule:

$$\frac{\partial J}{\partial \alpha_{kj}} = \frac{\partial J}{\partial s_i} \frac{\partial s_i}{\partial h_{ik}} \frac{\partial h_{ik}}{\partial \alpha_{kj}} = \frac{\partial J}{\partial s_i} \frac{\partial s_i}{\partial \alpha_{kj}}$$

$$= \sum (y_i - \frac{e_i^s}{1+e_i^s})(\beta_i \times max\{0, x_{ij}\})$$

3. For J'(theta),

$$J'(\theta) = [\frac{\partial J}{\partial \beta_1}, ..., \frac{\partial J}{\partial \beta_d}, \frac{\partial J}{\partial \alpha_1 1}, ..., \frac{\partial J}{\partial \alpha_{dp}}]^T$$

4. Gradient Ascent algorithm with momentum

$$\theta^{(t+1)} = \theta^{(t)} + \mu_t V^{(t)}$$
$$V^{(t)} = \gamma v^{t-1} + J'(\theta)$$

Problem 3: Write Python code to implement the algorithm in Problem 2, where $x_i = (x_{i,1}, x_{i,2})\ Uniform[0,1]^2$, i.e., a unit square, and $y_i = 1(x_{i,1}^2 + x_{i,2}^2 < 1)$, i.e., within a unit circle. Plot the decision boundary defined by the learned neural network.

I will introduce three Neural network models and compare the performance benchmark with data pipeline diagram

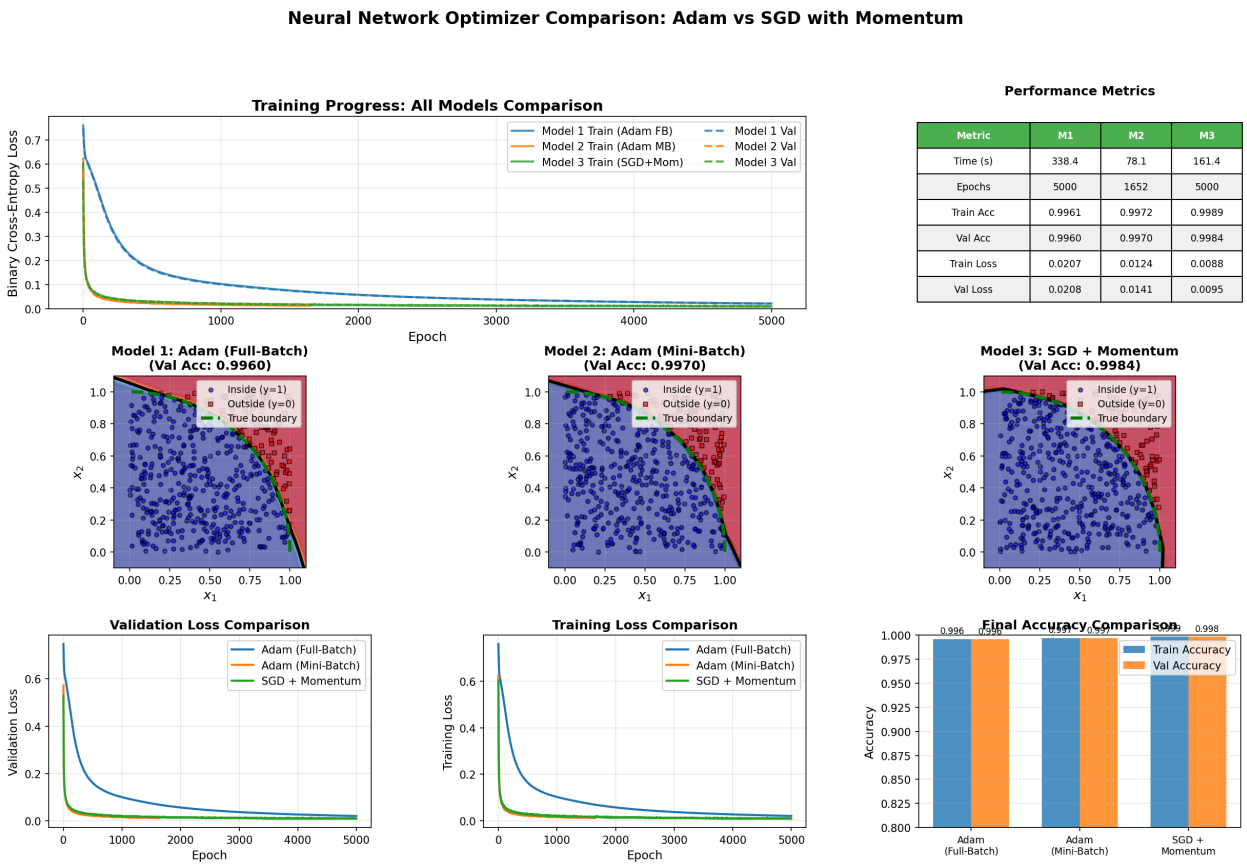
1. NeuralNetwork-Model 1: Adam Optimizer with Full batch load
2. NeuralNetwork-Model 2: Adam Optimizer with Fully Optimized Python code and mini batch
3. NeuralNetwork-Model3: Scohestic SGD with Momentum

Overview of the Data pipeline Algorithms

- Model Selection Layer: NN-Adam, NN-AdamFullOptimized, and NN-SGD+Momentum
- Training pipeline with Epoch based
- Prediction layer: Predict-probability and Predict
- Source Code: I’ve attached as distinct files.

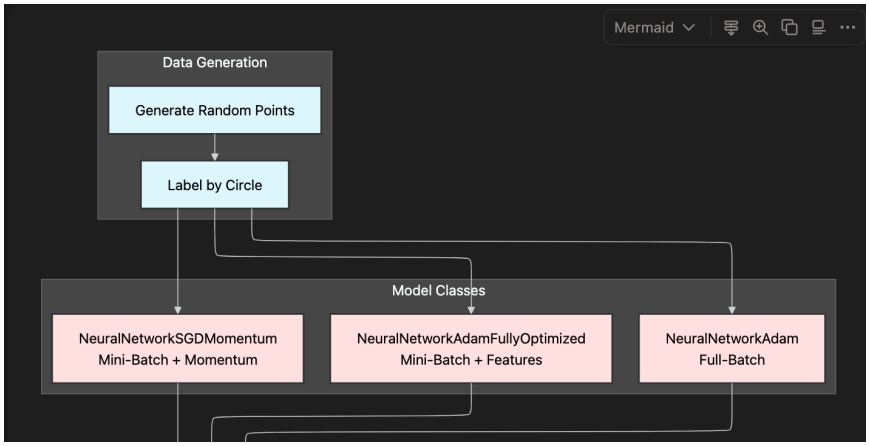
- hw1.py: Main function
- NeuralNetworkAdam.py : Adam optimizer with full batch
- NeuralNetworkAdamFullyOptimized.py
- Stochastic Gradient Descent with Momentum

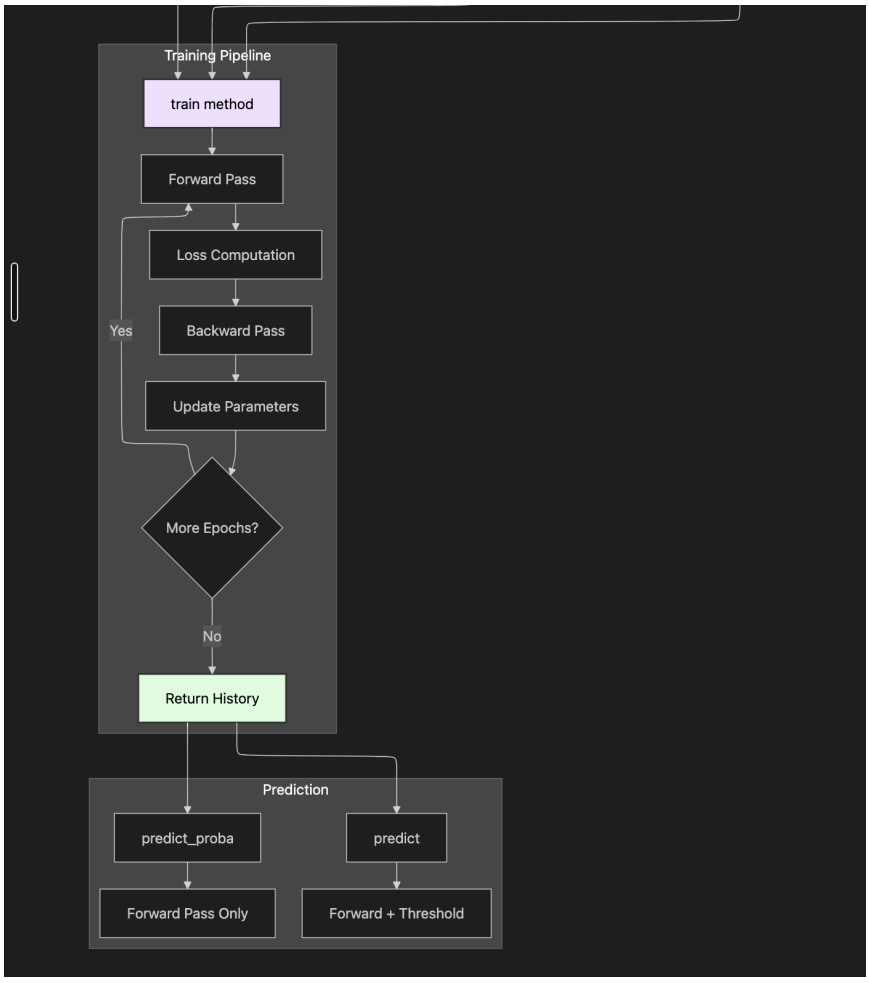
- Boundary detection with three different models and optimizer performance comparison
  - As seen the graphs below, SGD+Momentum has absolutely great accuracy (Val Acc:0.9984) by outperformed other Adam Optimizer models



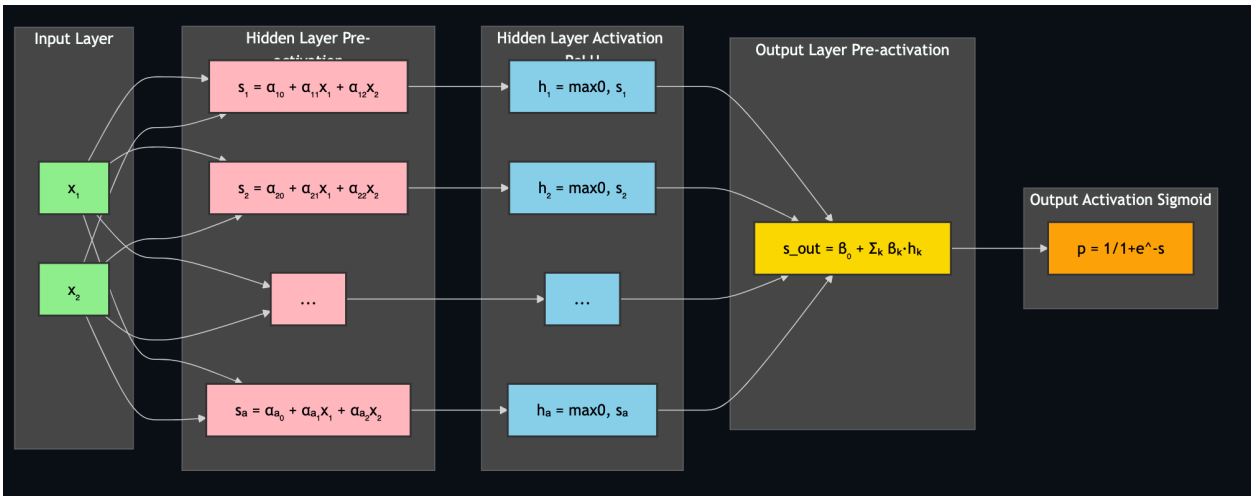
Metric	M1	M2	M3
Time (s)	338.4	78.1	161.4
Epochs	5000	1652	5000
Train Acc	0.9961	0.9972	0.9989
Val Acc	0.9960	0.9970	0.9984
Train Loss	0.0207	0.0124	0.0088
Val Loss	0.0208	0.0141	0.0095

1. Full Neural Network Implementation Diagram

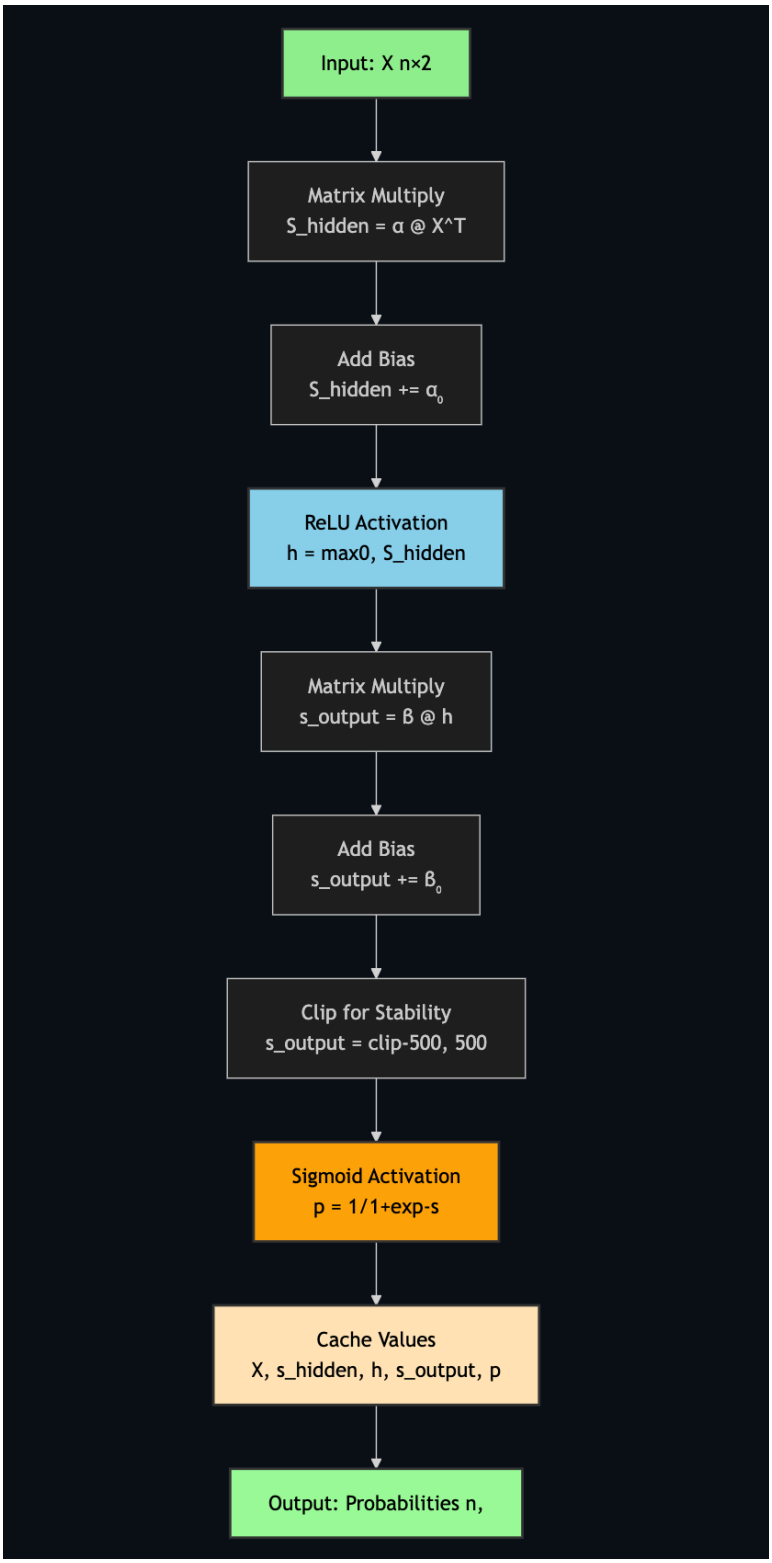




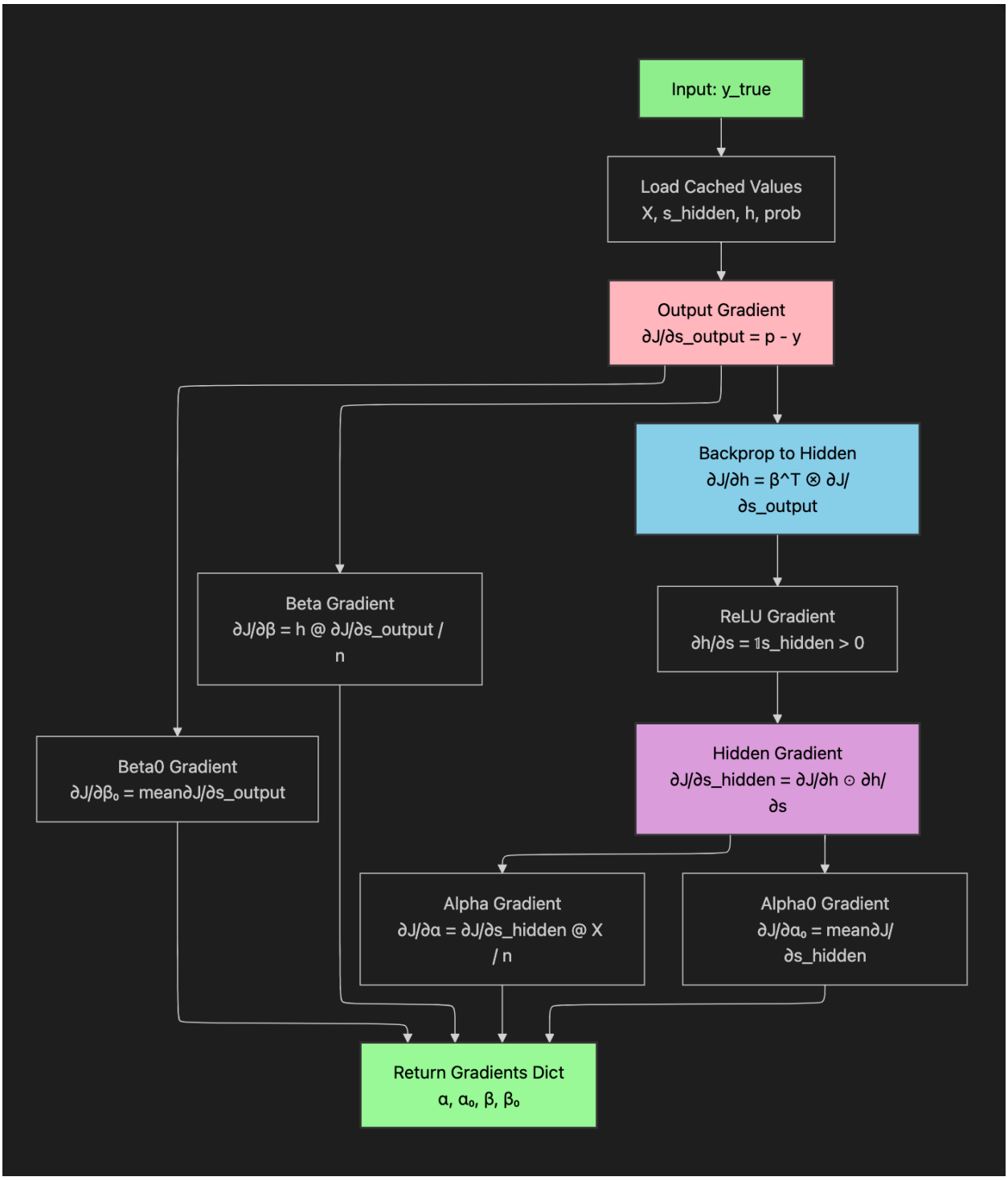
2. Neural Network Layer-by-Layer Flow



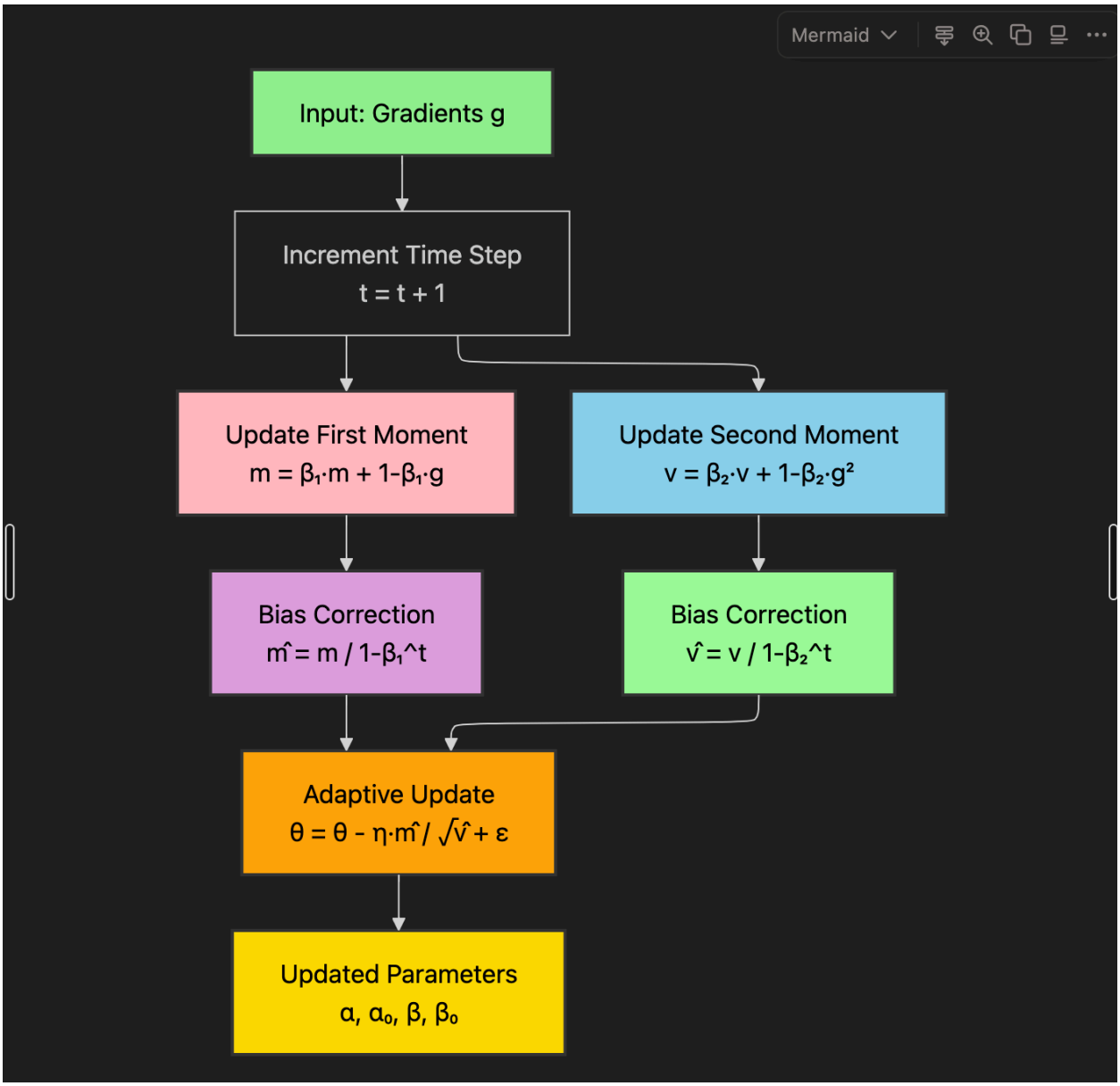
3. Forward pass



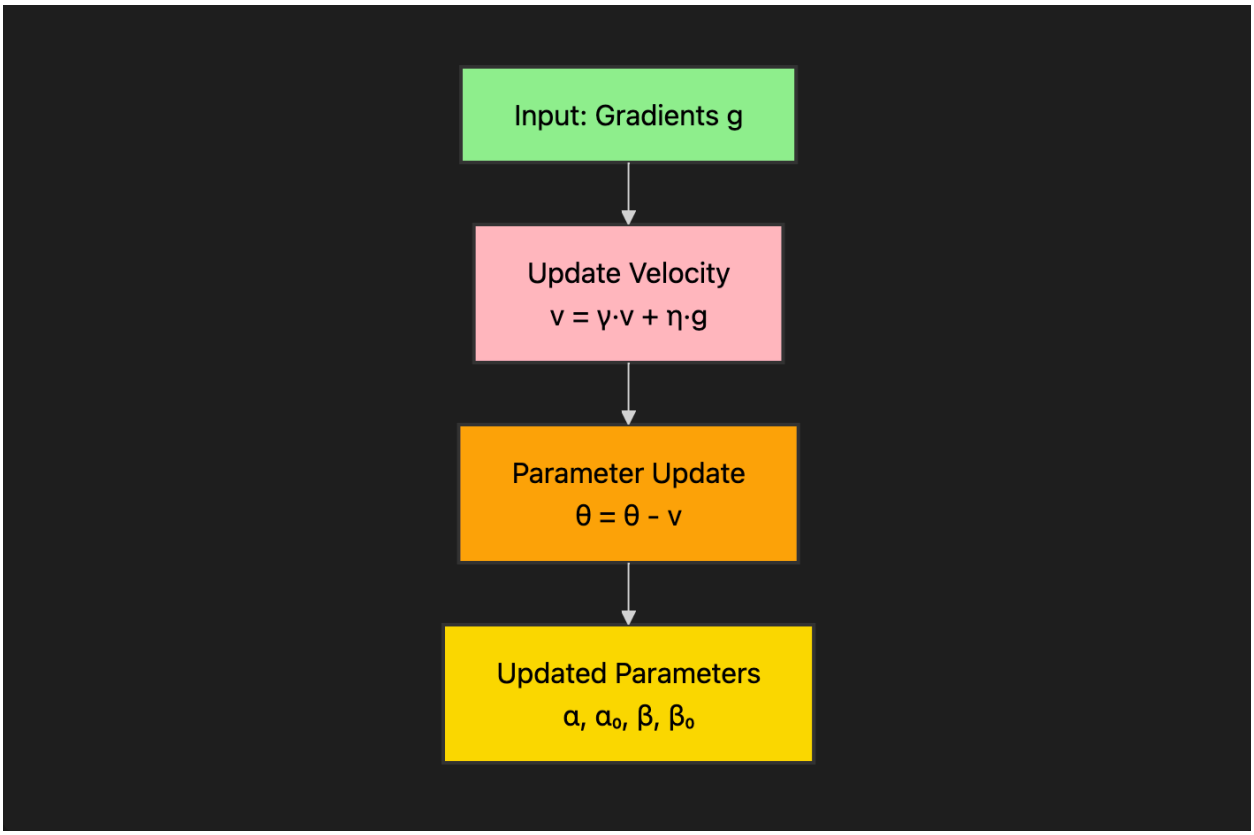
4. Backward Pass (Backpropagation)



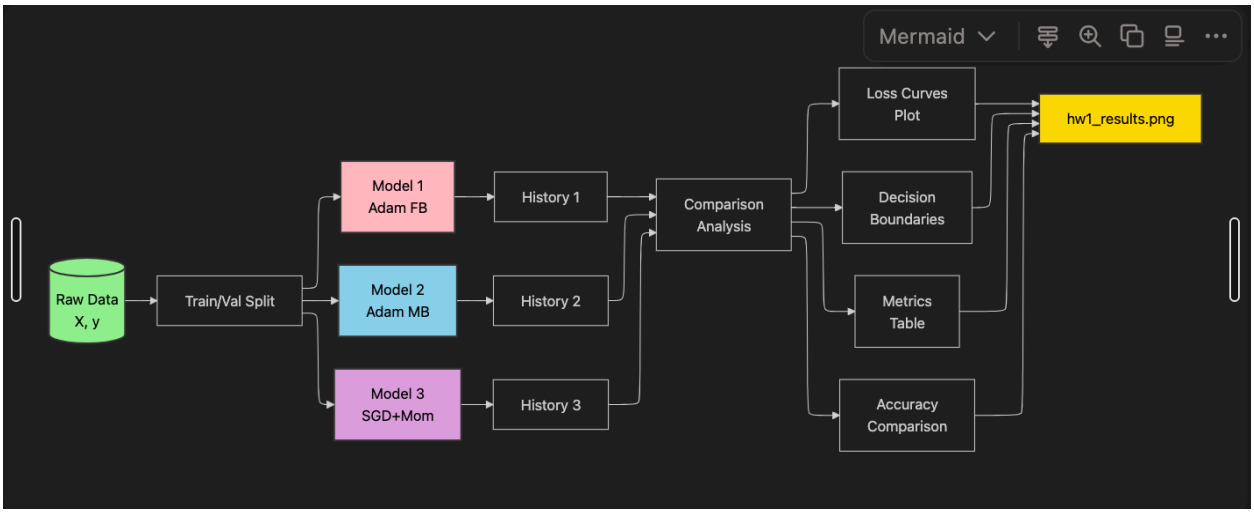
5. **Adam Optimizer Update Process**



6. **SGD with Momentum Update Process**



7. Data flow entire system



Codebase explanation

- Main function

```
"""
STATS 413 - Homework 1
Neural Network Implementation Comparison

Compares three optimization algorithms:
1. Adam (full-batch)
2. Adam (mini-batch with optimizations)
3. SGD with Momentum (mini-batch)

Task: Learn circular decision boundary (x₁² + x₂² < 1)
"""

import numpy as np
import matplotlib.pyplot as plt
import time
from NeuralNetworkAdam import NeuralNetworkAdam
from NeuralNetworkAdamFullyOptimized import NeuralNetworkAdamFullyOpti
mized
from NeuralNetworkSGDMomentum import NeuralNetworkSGDMomentum


def generate_data(n_samples):
    """
    Generate training data: uniform in [0,1]^2, labeled by unit circle

    Args:
        n_samples: Number of samples to generate

    Returns:
        X: Input features of shape (n_samples, 2)
        y: Binary labels of shape (n_samples,)
    """
    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):
    """
    Plot decision boundary for a trained model

    Args:
        ax: Matplotlib axis object
        model: Trained neural network model
        X_val: Validation data
        y_val: Validation labels
        title: Plot title
        val_acc: Validation accuracy
    """
    # Create meshgrid for decision boundary
    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, 300),
                           np.linspace(x2_min, x2_max, 300))
    X_grid = np.c_[xx1.ravel(), xx2.ravel()]

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

    # Plot filled contour
    contourf = 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 validation points for visualization
    sample_size = min(500, len(y_val))
    sample_idx = np.random.choice(len(y_val), sample_size, replace=False)

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

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

    # Plot 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(Val Acc: {val_acc:.4f})', fontsize=12, 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)

    return contourf


def main():
    """Main training and evaluation function"""

    # Set random seed for reproducibility
    np.random.seed(2024)

    print("=" * 80)
    print("STATS 413 - HW1: Neural Network Optimizer Comparison")
    print("Task: Learn quarter circle decision boundary")
    print("=" * 80)
    print()

    # ============================================
    # Generate Data
```

```
    # ============================================
    print("Generating data...")
    N_train = 10000
    N_val = 5000

    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 samples (inside circle): {np.sum(y_train)}/{N_train} = {np.me
an(y_train):.2%}")
    print()

    # ============================================
    # Model 1: Adam (Full-Batch)
    # ============================================
    print("=" * 80)
    print("MODEL 1: Adam Optimizer (Full-Batch)")
    print("=" * 80)

    model1 = NeuralNetworkAdam(
        input_dim=2,
        hidden_dim=128,
        learning_rate=0.001,
        beta1=0.9,
        beta2=0.999
    )

    print(f"Architecture: Input(2) → Hidden(128) → ReLU → Output(1) → Sigmoi
d")
    print(f"Optimizer: Adam (lr={model1.lr}, β₁={model1.beta1}, β₂={model1.beta
2})")
    print(f"Total parameters: {model1.alpha.size + model1.alpha0.size + model1.
beta.size + 1}")
    print()
```

```
print("Training Model 1...")
start_time1 = time.time()
history1 = model1.train(
    X_train, y_train, X_val, y_val,
    epochs=5000,
    print_every=1000,
    verbose=True
)
time1 = time.time() - start_time1

# Evaluate Model 1
y_train_pred1 = model1.predict(X_train)
y_val_pred1 = model1.predict(X_val)
train_acc1 = np.mean(y_train_pred1 == y_train)
val_acc1 = np.mean(y_val_pred1 == y_val)
final_train_loss1 = history1['train_loss'][max(history1['train_loss'].keys())]
final_val_loss1 = history1['val_loss'][max(history1['val_loss'].keys())]

print()
print("Model 1 Results:")
print(f"  Training Time: {time1:.2f}s")
print(f"  Training Loss: {final_train_loss1:.5f} | Accuracy: {train_acc1:.4f}")
print(f"  Validation Loss: {final_val_loss1:.5f} | Accuracy: {val_acc1:.4f}")
print()

# ============================================
# Model 2: Adam Fully Optimized (Mini-Batch)
# ============================================
print("=" * 80)
print("MODEL 2: Adam Optimizer (Mini-Batch + Optimizations)")
print("=" * 80)

model2 = NeuralNetworkAdamFullyOptimized(
    input_dim=2,
    hidden_dim=128,
```

```
        learning_rate=0.001,
        beta1=0.9,
        beta2=0.999,
        batch_size=256
    )

    print(f"Architecture: Input(2) → Hidden(128) → ReLU → Output(1) → Sigmoi
d")
    print(f"Optimizer: Adam (lr={model2.lr}, β₁={model2.beta1}, β₂={model2.bet
a2})")
    print(f"Batch Size: {model2.batch_size}")
    print(f"Features: Mini-batch, Early Stopping, LR Scheduling, Gradient Clippi
ng")
    print(f"Total parameters: {model2.alpha.size + model2.alpha0.size + model
2.beta.size + 1}")
    print()

    print("Training Model 2...")
    start_time2 = time.time()
    history2 = model2.train(
        X_train, y_train, X_val, y_val,
        epochs=5000,
        print_every=1000,
        early_stopping_patience=50,
        lr_schedule='cosine',
        gradient_clip=1.0,
        verbose=True
    )
    time2 = time.time() - start_time2

    # Evaluate Model 2
    y_train_pred2 = model2.predict(X_train)
    y_val_pred2 = model2.predict(X_val)
    train_acc2 = np.mean(y_train_pred2 == y_train)
    val_acc2 = np.mean(y_val_pred2 == y_val)
    final_train_loss2 = history2['train_loss'][max(history2['train_loss'].keys())]
```

```
    final_val_loss2 = history2['val_loss'][max(history2['val_loss'].keys())]

    print()
    print("Model 2 Results:")
    print(f"  Training Time: {time2:.2f}s")
    print(f"  Training Loss: {final_train_loss2:.5f} | Accuracy: {train_acc2:.4f}")
    print(f"  Validation Loss: {final_val_loss2:.5f} | Accuracy: {val_acc2:.4f}")
    print()

    # ============================================
    # Model 3: SGD with Momentum (Mini-Batch)
    # ============================================
    print("=" * 80)
    print("MODEL 3: SGD with Momentum (Mini-Batch)")
    print("=" * 80)

    model3 = NeuralNetworkSGDMomentum(
        input_dim=2,
        hidden_dim=128,
        learning_rate=0.01,
        momentum=0.9,
        batch_size=256
    )

    print(f"Architecture: Input(2) → Hidden(128) → ReLU → Output(1) → Sigmoi
d")
    print(f"Optimizer: SGD with Momentum (lr={model3.lr}, γ={model3.moment
um})")
    print(f"Batch Size: {model3.batch_size}")
    print(f"Total parameters: {model3.alpha.size + model3.alpha0.size + model
3.beta.size + 1}")
    print()

    print("Training Model 3...")
    start_time3 = time.time()
    history3 = model3.train(
```

```
        X_train, y_train, X_val, y_val,
        epochs=5000,
        print_every=1000,
        verbose=True
    )
    time3 = time.time() - start_time3

    # Evaluate Model 3
    y_train_pred3 = model3.predict(X_train)
    y_val_pred3 = model3.predict(X_val)
    train_acc3 = np.mean(y_train_pred3 == y_train)
    val_acc3 = np.mean(y_val_pred3 == y_val)
    final_train_loss3 = history3['train_loss'][max(history3['train_loss'].keys())]
    final_val_loss3 = history3['val_loss'][max(history3['val_loss'].keys())]

    print()
    print("Model 3 Results:")
    print(f"  Training Time: {time3:.2f}s")
    print(f"  Training Loss: {final_train_loss3:.5f} | Accuracy: {train_acc3:.4f}")
    print(f"  Validation Loss: {final_val_loss3:.5f} | Accuracy: {val_acc3:.4f}")
    print()

    # ============================================
    # Performance Comparison Summary
    # ============================================
    print("=" * 80)
    print("PERFORMANCE COMPARISON SUMMARY")
    print("=" * 80)
    print()
    print(f"{'Metric':<30} {'Model 1 (Adam FB)':<20} {'Model 2 (Adam MB)':<2
0} {'Model 3 (SGD+Mom)':<20}")
    print("-" * 90)
    print(f"{'Training Time':<30} {time1:>15.2f}s     {time2:>15.2f}s     {time3:>15.
2f}s")
    print(f"{'Epochs Completed':<30} {max(history1['train_loss'].keys())+1:>20}
{max(history2['train_loss'].keys())+1:>20} {max(history3['train_loss'].keys())+
```

```
1:>20}")
    print(f"{'Final Train Loss':<30} {final_train_loss1:>20.5f} {final_train_loss2:>
20.5f} {final_train_loss3:>20.5f}")
    print(f"{'Final Val Loss':<30} {final_val_loss1:>20.5f} {final_val_loss2:>20.5f}
{final_val_loss3:>20.5f}")
    print(f"{'Train Accuracy':<30} {train_acc1:>20.4f} {train_acc2:>20.4f} {train
_acc3:>20.4f}")
    print(f"{'Val Accuracy':<30} {val_acc1:>20.4f} {val_acc2:>20.4f} {val_acc3:
>20.4f}")
    print()

    # ================================================
    # Visualization
    # ================================================
    print("Creating comprehensive visualizations...")

    fig = plt.figure(figsize=(20, 12))
    gs = fig.add_gridspec(3, 3, hspace=0.35, wspace=0.35)

    # Row 1: Loss curves
    ax1 = fig.add_subplot(gs[0, :2])
    epochs1 = list(history1['train_loss'].keys())
    epochs2 = list(history2['train_loss'].keys())
    epochs3 = list(history3['train_loss'].keys())

    # Training losses
    ax1.plot(epochs1, list(history1['train_loss'].values()),
             label='Model 1 Train (Adam FB)', linewidth=2, linestyle='-', alpha=0.8,
color='#1f77b4')
    ax1.plot(epochs2, list(history2['train_loss'].values()),
             label='Model 2 Train (Adam MB)', linewidth=2, linestyle='-', alpha=0.8,
color='#ff7f0e')
    ax1.plot(epochs3, list(history3['train_loss'].values()),
             label='Model 3 Train (SGD+Mom)', linewidth=2, linestyle='-', alpha=0.
8, color='#2ca02c')
```

```
    # Validation losses
    ax1.plot(epochs1, list(history1['val_loss'].values()),
             label='Model 1 Val', linewidth=2, linestyle='--', alpha=0.8, color='#1f77
b4')
    ax1.plot(epochs2, list(history2['val_loss'].values()),
             label='Model 2 Val', linewidth=2, linestyle='--', alpha=0.8, color='#ff7f
0e')
    ax1.plot(epochs3, list(history3['val_loss'].values()),
             label='Model 3 Val', linewidth=2, linestyle='--', alpha=0.8, color='#2ca
02c')

    ax1.set_xlabel('Epoch', fontsize=12)
    ax1.set_ylabel('Binary Cross-Entropy Loss', fontsize=12)
    ax1.set_title('Training Progress: All Models Comparison', fontsize=14, fontwe
ight='bold')
    ax1.legend(fontsize=10, loc='upper right', ncol=2)
    ax1.grid(True, alpha=0.3)
    ax1.set_ylim(bottom=0)

    # Performance metrics table
    ax_table = fig.add_subplot(gs[0, 2])
    ax_table.axis('off')

    table_data = [
        ['Metric', 'M1', 'M2', 'M3'],
        ['Time (s)', f'{time1:.1f}', f'{time2:.1f}', f'{time3:.1f}'],
        ['Epochs', f'{max(epochs1)+1}', f'{max(epochs2)+1}', f'{max(epochs3)+
1}'],
        ['Train Acc', f'{train_acc1:.4f}', f'{train_acc2:.4f}', f'{train_acc3:.4f}'],
        ['Val Acc', f'{val_acc1:.4f}', f'{val_acc2:.4f}', f'{val_acc3:.4f}'],
        ['Train Loss', f'{final_train_loss1:.4f}', f'{final_train_loss2:.4f}', f'{final_train
_loss3:.4f}'],
        ['Val Loss', f'{final_val_loss1:.4f}', f'{final_val_loss2:.4f}', f'{final_val_loss3:.
4f}']
    ]
```

```
    table = ax_table.table(cellText=table_data, cellLoc='center', loc='center',
                           colWidths=[0.35, 0.22, 0.22, 0.22])
    table.auto_set_font_size(False)
    table.set_fontsize(9)
    table.scale(1, 2)

    # Header styling
    for i in range(4):
        table[(0, i)].set_facecolor('#4CAF50')
        table[(0, i)].set_text_props(weight='bold', color='white')

    # Alternate row colors
    for i in range(1, len(table_data)):
        for j in range(4):
            if i % 2 == 0:
                table[(i, j)].set_facecolor('#f0f0f0')

    ax_table.set_title('Performance Metrics', fontsize=12, fontweight='bold', pad
=20)

    # Row 2: Decision boundaries
    ax2 = fig.add_subplot(gs[1, 0])
    plot_decision_boundary(ax2, model1, X_val, y_val, 'Model 1: Adam (Full-Batc
h)', val_acc1)

    ax3 = fig.add_subplot(gs[1, 1])
    plot_decision_boundary(ax3, model2, X_val, y_val, 'Model 2: Adam (Mini-Bat
ch)', val_acc2)

    ax4 = fig.add_subplot(gs[1, 2])
    plot_decision_boundary(ax4, model3, X_val, y_val, 'Model 3: SGD + Moment
um', val_acc3)

    # Row 3: Validation loss comparison (zoomed)
    ax5 = fig.add_subplot(gs[2, 0])
    ax5.plot(epochs1, list(history1['val_loss'].values()),
```

```
        label='Adam (Full-Batch)', linewidth=2, color='#1f77b4')
ax5.plot(epochs2, list(history2['val_loss'].values()),
        label='Adam (Mini-Batch)', linewidth=2, color='#ff7f0e')
ax5.plot(epochs3, list(history3['val_loss'].values()),
        label='SGD + Momentum', linewidth=2, color='#2ca02c')
ax5.set_xlabel('Epoch', fontsize=11)
ax5.set_ylabel('Validation Loss', fontsize=11)
ax5.set_title('Validation Loss Comparison', fontsize=12, fontweight='bold')
ax5.legend(fontsize=10)
ax5.grid(True, alpha=0.3)

# Training loss comparison (zoomed)
ax6 = fig.add_subplot(gs[2, 1])
ax6.plot(epochs1, list(history1['train_loss'].values()),
        label='Adam (Full-Batch)', linewidth=2, color='#1f77b4')
ax6.plot(epochs2, list(history2['train_loss'].values()),
        label='Adam (Mini-Batch)', linewidth=2, color='#ff7f0e')
ax6.plot(epochs3, list(history3['train_loss'].values()),
        label='SGD + Momentum', linewidth=2, color='#2ca02c')
ax6.set_xlabel('Epoch', fontsize=11)
ax6.set_ylabel('Training Loss', fontsize=11)
ax6.set_title('Training Loss Comparison', fontsize=12, fontweight='bold')
ax6.legend(fontsize=10)
ax6.grid(True, alpha=0.3)

# Accuracy comparison
ax7 = fig.add_subplot(gs[2, 2])
models = ['Adam\n(Full-Batch)', 'Adam\n(Mini-Batch)', 'SGD +\nMomentum']
train_accs = [train_acc1, train_acc2, train_acc3]
val_accs = [val_acc1, val_acc2, val_acc3]

x = np.arange(len(models))
width = 0.35

bars1 = ax7.bar(x - width/2, train_accs, width, label='Train Accuracy',
              color='#1f77b4', alpha=0.8)
```

```
    bars2 = ax7.bar(x + width/2, val_accs, width, label='Val Accuracy',
                    color='#ff7f0e', alpha=0.8)

    ax7.set_ylabel('Accuracy', fontsize=11)
    ax7.set_title('Final Accuracy Comparison', fontsize=12, fontweight='bold')
    ax7.set_xticks(x)
    ax7.set_xticklabels(models, fontsize=9)
    ax7.legend(fontsize=10)
    ax7.grid(True, alpha=0.3, axis='y')
    ax7.set_ylim([0.8, 1.0])

    # Add value labels on bars
    for bars in [bars1, bars2]:
        for bar in bars:
            height = bar.get_height()
            ax7.annotate(f'{height:.3f}',
                         xy=(bar.get_x() + bar.get_width() / 2, height),
                         xytext=(0, 3),
                         textcoords="offset points",
                         ha='center', va='bottom', fontsize=8)

    plt.suptitle('Neural Network Optimizer Comparison: Adam vs SGD with Mom
entum',
                 fontsize=16, fontweight='bold', y=0.995)

    # Save figure
    plt.savefig('hw1_results.png', dpi=150, bbox_inches='tight')
    print("Visualization saved as 'hw1_results.png'")
    print()

    print("=" * 80)
    print("Analysis Complete!")
    print("=" * 80)
```

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

SGD with Momentum

```
import numpy as np
from typing import Dict

class NeuralNetworkSGDMomentum:
    """
    Two-layer neural network with ReLU activation and SGD with Momentum op
timizer

    Architecture:
        Input(2) → Hidden(d) → ReLU → Output(1) → Sigmoid

    Mathematical formulation:
        s_ik = α_k0 + Σ_j α_kj·x_ij  (hidden layer)
        h_ik = max(0, s_ik)          (ReLU)
        s_i = β_0 + Σ_k β_k·h_ik     (output layer)
        p_i = σ(s_i) = 1/(1+e^(-s_i)) (sigmoid)
    """

    def __init__(self, input_dim: int = 2, hidden_dim: int = 128,
                 learning_rate: float = 0.01, momentum: float = 0.9,
                 batch_size: int = 256):
        """
        Initialize network parameters and SGD with Momentum optimizer state

        Args:
            input_dim: Number of input features (p)
            hidden_dim: Number of hidden units (d)
            learning_rate: Learning rate (η)
            momentum: Momentum coefficient (γ), typically 0.9
            batch_size: Size of mini-batches for training
```

```
        """
        self.p = input_dim
        self.d = hidden_dim
        self.lr = learning_rate
        self.momentum = momentum
        self.batch_size = batch_size

        # ============================================
        # Initialize Network Parameters
        # ============================================
        # Hidden layer (α parameters)
        self.alpha = np.random.randn(hidden_dim, input_dim) * np.sqrt(2.0 / input
_dim)
        self.alpha0 = np.zeros(hidden_dim)

        # Output layer (β parameters)
        self.beta = np.random.randn(hidden_dim) * np.sqrt(2.0 / hidden_dim)
        self.beta0 = 0.0

        # ============================================
        # Initialize Momentum State
        # ============================================
        # Velocity (momentum buffer) - v
        self.v_alpha = np.zeros_like(self.alpha)
        self.v_alpha0 = np.zeros_like(self.alpha0)
        self.v_beta = np.zeros_like(self.beta)
        self.v_beta0 = 0.0

        # Cache for backpropagation
        self.cache = {}

    def forward(self, X: np.ndarray) → np.ndarray:
        """
        Forward pass through the network

        Mathematical operations:
```

```
    1. s_hidden = α₀ + α @ X^T        (hidden pre-activation)
    2. h = max(0, s_hidden)           (ReLU activation)
    3. s_output = β₀ + β^T @ h        (output pre-activation)
    4. prob = 1 / (1 + exp(-s_output)) (sigmoid activation)

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

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

# Step 1: Hidden layer pre-activation
s_hidden = self.alpha0[:, np.newaxis] + self.alpha @ X.T  # (d, n)

# Step 2: ReLU activation
h = np.maximum(0, s_hidden)  # (d, n)

# Step 3: Output layer pre-activation
s_output = self.beta0 + self.beta @ h  # (n,)

# Step 4: Sigmoid activation
s_output_clipped = np.clip(s_output, -500, 500)  # Numerical stability
prob = 1.0 / (1.0 + np.exp(-s_output_clipped))  # (n,)

# Cache intermediate values for backpropagation
self.cache = {
    'X': X,
    's_hidden': s_hidden,
    'h': h,
    's_output': s_output,
    'prob': prob,
    'n': n
}
```

```
        return prob

    def loss_fn(self, y_true: np.ndarray, y_pred: np.ndarray = None) → float:
        """
        Binary cross-entropy loss (negative log-likelihood)

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

        Args:
            y_true: True labels of shape (n,)
            y_pred: Predicted probabilities of shape (n,) [optional, uses cache if No
ne]

        Returns:
            loss: Scalar binary cross-entropy loss
        """
        if y_pred is None:
            # Use numerically stable version from logits
            s = self.cache['s_output']
            loss = np.mean(
                np.maximum(0, s) - s * y_true + np.log(1 + np.exp(-np.abs(s)))
            )
        else:
            eps = 1e-15
            y_pred_clipped = np.clip(y_pred, eps, 1 - eps)
            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[str, np.ndarray]:
        """
        Backpropagation: Compute gradients using chain rule
```

```
    Mathematical derivation:

    Step 1: Gradient at output (sigmoid + cross-entropy)
        ∂J/∂s_output = p - y

    Step 2: Gradients for β parameters
        ∂J/∂β_k = (1/n) Σᵢ (pᵢ - yᵢ)·h_ik
        ∂J/∂β₀ = (1/n) Σᵢ (pᵢ - yᵢ)

    Step 3: Backprop through output layer
        ∂J/∂h_ik = (pᵢ - yᵢ)·β_k

    Step 4: Gradient through ReLU
        ∂J/∂s_ik = ∂J/∂h_ik · 𝟙(s_ik > 0)

    Step 5: Gradients for α parameters
        ∂J/∂α_kj = (1/n) Σᵢ (∂J/∂s_ik)·x_ij
        ∂J/∂α_k0 = (1/n) Σᵢ (∂J/∂s_ik)

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

    Returns:
        gradients: Dictionary containing all parameter gradients
    """
    # Retrieve cached values from forward pass
    X = self.cache['X']
    s_hidden = self.cache['s_hidden']
    h = self.cache['h']
    prob = self.cache['prob']
    n = self.cache['n']

    # Step 1: Gradient at output layer
    dJ_ds_output = prob - y_true  # (n,)

    # Step 2: Gradients for β parameters
```

```
        grad_beta = (h @ dJ_ds_output) / n  # (d,)
        grad_beta0 = np.mean(dJ_ds_output)  # scalar

        # Step 3: Backprop through output layer
        dJ_dh = np.outer(self.beta, dJ_ds_output)  # (d, n)

        # Step 4: Gradient through ReLU
        dh_ds_hidden = (s_hidden > 0).astype(float)  # (d, n)
        dJ_ds_hidden = dJ_dh * dh_ds_hidden  # (d, n)

        # Step 5: Gradients for α parameters
        grad_alpha = (dJ_ds_hidden @ X) / n  # (d, p)
        grad_alpha0 = np.mean(dJ_ds_hidden, axis=1)  # (d,)

        # Return all gradients as dictionary
        gradients = {
            'alpha': grad_alpha,
            'alpha0': grad_alpha0,
            'beta': grad_beta,
            'beta0': grad_beta0
        }

        return gradients

    def update_parameters(self, gradients: Dict[str, np.ndarray]) → None:
        """
        Update parameters using SGD with Momentum optimizer

        Momentum Algorithm:

        For each parameter θ and its gradient g:
            1. v = γ·v + η·g     (update velocity with momentum)
            2. θ = θ - v         (parameter update)

        Intuition:
            - v (velocity) accumulates gradient history
```

```
        - γ (momentum) determines how much past gradients influence current
update
        - Helps accelerate SGD in relevant directions and dampen oscillations
        - Typical values: γ ∈ [0.5, 0.9, 0.99]

    Mathematical Note:
        Momentum can be viewed as exponentially weighted moving average:
        v_t = γ·v_{t-1} + η·g_t
            = η·Σᵢ γⁱ·g_{t-i}

    Args:
        gradients: Dictionary of gradients from backward pass
    """
    # Extract gradients
    grad_alpha = gradients['alpha']
    grad_alpha0 = gradients['alpha0']
    grad_beta = gradients['beta']
    grad_beta0 = gradients['beta0']

    # ============================================
    # Update α parameters (hidden layer weights)
    # ============================================
    # Update velocity: v = γ·v + η·g
    self.v_alpha = self.momentum * self.v_alpha + self.lr * grad_alpha

    # Update parameter: θ = θ - v
    self.alpha -= self.v_alpha

    # ============================================
    # Update α₀ parameters (hidden layer biases)
    # ============================================
    self.v_alpha0 = self.momentum * self.v_alpha0 + self.lr * grad_alpha0
    self.alpha0 -= self.v_alpha0

    # ============================================
    # Update β parameters (output layer weights)
```

```
        # ============================================
        self.v_beta = self.momentum * self.v_beta + self.lr * grad_beta
        self.beta -= self.v_beta

        # ============================================
        # Update β₀ parameter (output layer bias)
        # ============================================
        self.v_beta0 = self.momentum * self.v_beta0 + self.lr * grad_beta0
        self.beta0 -= self.v_beta0

    def train(self, X_train: np.ndarray, y_train: np.ndarray,
              X_val: np.ndarray = None, y_val: np.ndarray = None,
              epochs: int = 50000, print_every: int = 1000,
              verbose: bool = True) → Dict[str, Dict[int, float]]:
        """
        Training loop with mini-batch SGD and momentum

        Algorithm:
            For each epoch:
                1. Shuffle training data
                2. For each mini-batch:
                    a. Forward pass: compute predictions
                    b. Compute loss: measure error
                    c. Backward pass: compute gradients
                    d. Update parameters: apply SGD with momentum
                3. (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
        """
        n_samples = X_train.shape[0]
        n_batches = int(np.ceil(n_samples / self.batch_size))

        history = {
            'train_loss': {},
            'val_loss': {}
        }

        for epoch in range(epochs):
            # Shuffle data for stochastic gradient descent
            indices = np.random.permutation(n_samples)
            X_shuffled = X_train[indices]
            y_shuffled = y_train[indices]

            epoch_losses = []

            # ============================================
            # Mini-batch Training
            # ============================================
            for batch_idx in range(n_batches):
                start_idx = batch_idx * self.batch_size
                end_idx = min(start_idx + self.batch_size, n_samples)

                X_batch = X_shuffled[start_idx:end_idx]
                y_batch = y_shuffled[start_idx:end_idx]

                # 1. Forward pass
                y_pred = self.forward(X_batch)

                # 2. Compute loss
                batch_loss = self.loss_fn(y_batch)
                epoch_losses.append(batch_loss)
```

```
            # 3. Backward pass
            gradients = self.backward(y_batch)

            # 4. Update parameters
            self.update_parameters(gradients)

        # Average loss for the epoch
        train_loss = np.mean(epoch_losses)
        history['train_loss'][epoch] = train_loss

        # ============================================
        # 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:05d}/{epochs:05d} | 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((self.forward(X_train) > 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)
```

Performance benchmark result

```
============================================================
========================
STATS 413 - HW1: Neural Network Optimizer Comparison
Task: Learn quarter circle decision boundary
============================================================
========================

Generating data...
Training samples: 10000
Validation samples: 5000
Positive samples (inside circle): 7906/10000 = 79.06%

============================================================
========================
MODEL 1: Adam Optimizer (Full-Batch)
============================================================
========================
Architecture: Input(2) → Hidden(128) → ReLU → Output(1) → Sigmoid
Optimizer: Adam (lr=0.001, β₁=0.9, β₂=0.999)
Total parameters: 513

Training Model 1...
Epoch 0000000/0005000 | Train Loss: 0.75979 | Val Loss: 0.74669 | Train Ac
c: 0.4166 | Val Acc: 0.4410
Epoch 0001000/0005000 | Train Loss: 0.10228 | Val Loss: 0.10032 | Train Acc:
0.9665 | Val Acc: 0.9674
Epoch 0002000/0005000 | Train Loss: 0.05742 | Val Loss: 0.05674 | Train Ac
c: 0.9888 | Val Acc: 0.9902
Epoch 0003000/0005000 | Train Loss: 0.03762 | Val Loss: 0.03747 | Train Ac
c: 0.9945 | Val Acc: 0.9936
Epoch 0004000/0005000 | Train Loss: 0.02726 | Val Loss: 0.02723 | Train Ac
c: 0.9945 | Val Acc: 0.9940
Epoch 0004999/0005000 | Train Loss: 0.02072 | Val Loss: 0.02082 | Train Ac
c: 0.9961 | Val Acc: 0.9960
```

```
Model 1 Results:
  Training Time: 338.40s
  Training Loss: 0.02072 | Accuracy: 0.9961
  Validation Loss: 0.02082 | Accuracy: 0.9960

============================================================
========================
MODEL 2: Adam Optimizer (Mini-Batch + Optimizations)
============================================================
========================
Architecture: Input(2) → Hidden(128) → ReLU → Output(1) → Sigmoid
Optimizer: Adam (lr=0.001, β₁=0.9, β₂=0.999)
Batch Size: 256
Features: Mini-batch, Early Stopping, LR Scheduling, Gradient Clipping
Total parameters: 513

Training Model 2...
Epoch 00000 | Train: 0.62285 | Val: 0.57163 | LR: 0.001000
Epoch 01000 | Train: 0.01586 | Val: 0.01601 | LR: 0.000905

Early stopping at epoch 1651
Best val loss: 0.01302

Model 2 Results:
  Training Time: 78.13s
  Training Loss: 0.01242 | Accuracy: 0.9972
  Validation Loss: 0.01413 | Accuracy: 0.9970

============================================================
========================
MODEL 3: SGD with Momentum (Mini-Batch)
============================================================
========================
Architecture: Input(2) → Hidden(128) → ReLU → Output(1) → Sigmoid
Optimizer: SGD with Momentum (lr=0.01, γ=0.9)
Batch Size: 256
```

```
Total parameters: 513

Training Model 3...
Epoch 00000/05000 | Train Loss: 0.60257 | Val Loss: 0.52839 | Train Acc: 0.7
906 | Val Acc: 0.7966
Epoch 01000/05000 | Train Loss: 0.02012 | Val Loss: 0.02002 | Train Acc: 0.9
962 | Val Acc: 0.9964
Epoch 02000/05000 | Train Loss: 0.01409 | Val Loss: 0.01437 | Train Acc: 0.9
974 | Val Acc: 0.9972
Epoch 03000/05000 | Train Loss: 0.01244 | Val Loss: 0.01178 | Train Acc: 0.99
74 | Val Acc: 0.9976
Epoch 04000/05000 | Train Loss: 0.01041 | Val Loss: 0.01104 | Train Acc: 0.99
78 | Val Acc: 0.9964
Epoch 04999/05000 | Train Loss: 0.00880 | Val Loss: 0.00947 | Train Acc: 0.9
989 | Val Acc: 0.9984

Model 3 Results:
  Training Time: 161.37s
  Training Loss: 0.00880 | Accuracy: 0.9989
  Validation Loss: 0.00947 | Accuracy: 0.9984

================================================================
========================
PERFORMANCE COMPARISON SUMMARY
================================================================
========================

Metric                    Model 1 (Adam FB)    Model 2 (Adam MB)    Model 3 (SG
D+Mom)
----------------------------------------------------------------------------
----------
Training Time                    338.40s              78.13s              161.37s
Epochs Completed                    5000                1652                5000
Final Train Loss                 0.02072             0.01242             0.00880
Final Val Loss                   0.02082             0.01413             0.00947
Train Accuracy                    0.9961              0.9972              0.9989
```

```
Val Accuracy                0.9960          0.9970          0.9984

Creating comprehensive visualizations...
Visualization saved as 'hw1_results.png'

================================================================
========================
Analysis Complete!
================================================================
========================
```