580 Homework 4

Yiming Mao

March 10, 2025

1 Gradient Descent and Convexity

(a) Gradient of the Negative Log-Likelihood

The negative log-likelihood function is given by:

$$\ell(w) = \sum_{i=1}^{n} y_i(w^\top x_i) - \log\left(1 + \exp(w^\top x_i)\right).$$

The gradient is computed as:

$$\nabla\ell(w) = \sum_{i=1}^{n} \left(y_i - \sigma(w^\top x_i)\right) x_i,$$

where the sigmoid function is defined as:

$$\sigma(z) = \frac{1}{1+\exp(-z)}.$$

Thus, the i-th entry of the gradient is:

$$[\nabla\ell(w)]_i = \sum_{j=1}^{n} \left(y_j - \sigma(w^\top x_j)\right) x_{j,i}.$$

(b) Hessian of the Negative Log-Likelihood

The Hessian is given by:

$$\nabla^2\ell(w) = \sum_{i=1}^{n} c(x_i, w)(1 - c(x_i, w))x_i x_i^\top.$$

Here, the scalar function $c(x_i, w)$ is:

$$c(x_i, w) = \sigma(w^\top x_i).$$

(c) Gradient Lipschitz Condition

We need to show that $\ell(w)$ is M-gradient Lipschitz with

$$M = \frac{1}{4}\lambda_{\max},$$

where $\lambda_{\max}$ is the maximum eigenvalue of $\sum_{i=1}^{n} x_i x_i^\top$.

From part (b), we note that:

$$0 \le c(x_i, w)(1 - c(x_i, w)) \le \frac{1}{4}.$$

Thus, the Hessian satisfies:

$$\nabla^2\ell(w) \preceq \frac{1}{4}\sum_{i=1}^{n} x_i x_i^\top.$$

Taking the operator norm, we get:

$$\|\nabla^2\ell(w)\|_2 \le \frac{1}{4}\lambda_{\max}.$$

(d) Proving Gradient Lipschitz Condition

We need to show that:

$$\|\nabla^2 \ell(w)\|_2 \leq M.$$

Since from part (c) we have:

$$\|\nabla^2 \ell(w)\|_2 \leq \frac{1}{4}\lambda_{\max},$$

we conclude that $\ell(w)$ satisfies the M-gradient Lipschitz condition with $M = \frac{1}{4}\lambda_{\max}$.

(e) Convexity

A function is convex if its Hessian is positive semi-definite:

$$\nabla^2 \ell(w) \succeq 0.$$

Since $c(x_i, w)(1 - c(x_i, w)) \geq 0$ and $x_i x_i^\top$ is positive semi-definite for all i, we conclude that:

$$\nabla^2 \ell(w) = \sum_{i=1}^{n} c(x_i, w)(1 - c(x_i, w)) x_i x_i^\top \succeq 0.$$

Thus, $\ell(w)$ is convex.

(f) Strong Convexity with L_2 Regularization

The regularized loss function is:

$$\ell_{\text{reg},c}(w) = \sum_{i=1}^{n} y_i (w^\top x_i) - \log\left(1 + \exp(w^\top x_i)\right) + c\|w\|_2^2.$$

The Hessian of the regularized function is:

$$\nabla^2 \ell_{\text{reg},c}(w) = \nabla^2 \ell(w) + 2cI.$$

Since $\nabla^2 \ell(w) \succeq 0$ from part (e), adding $2cI$ ensures:

$$\nabla^2 \ell_{\text{reg},c}(w) \succeq 2cI.$$

Thus, $\ell_{\text{reg},c}(w)$ is $2c$-strongly convex.

2 Implementing Gradient Descent

(a)

Input

```
# Learning rates to test
learning_rates = [1, 0.1, 0.01]
num_steps = 100

# Plot setup
plt.figure(figsize=(10, 6))

# Run gradient descent for each learning rate
for lr in learning_rates:
    # Initialize w to zero for each experiment
    w = torch.zeros(dim, requires_grad=True)

    # Use SGD optimizer with specified learning rate
    optimizer = torch.optim.SGD([w], lr=lr)

    # Store loss history
    loss_history = []

    # Gradient Descent Loop
    for step in range(num_steps):
        loss = logistic_neg_log_likelihood(w, X, Y)  # Compute loss
        loss.backward()  # Compute gradient
        optimizer.step()  # Update w
        optimizer.zero_grad()  # Reset gradients
        loss_history.append(loss.item())  # Store loss for plotting
```

(b)

Input

```
# Learning rates to test
learning_rates = [1, 0.1, 0.01]
num_steps = 100

# Plot setup
plt.figure(figsize=(10, 6))

for lr in learning_rates:
    # Initialize w to zero for each experiment (shape: (2,1))
    w = torch.zeros(dim, requires_grad=True)

    # Use SGD optimizer with specified learning rate
    optimizer = torch.optim.SGD([w], lr=lr)

    # Store loss history
    loss_history = []

    # Gradient Descent Loop
    for step in range(num_steps):
        loss = logistic_neg_log_likelihood(w, X, Y)  # Compute loss
        loss.backward()  # Compute gradient
        optimizer.step()  # Update w
        optimizer.zero_grad()  # Reset gradients
        loss_history.append(loss.item())  # Store loss for plotting

    # Plot loss curve for this learning rate
    plt.plot(range(1, num_steps + 1), loss_history, label=f'LR={lr}')

# Finalize plot
plt.xlabel('Iteration')
plt.ylabel('Negative Log-Likelihood Loss')
plt.title('Convergence of Logistic Regression for Different Learning Rates')
plt.legend()
plt.grid(True)
plt.show()
```

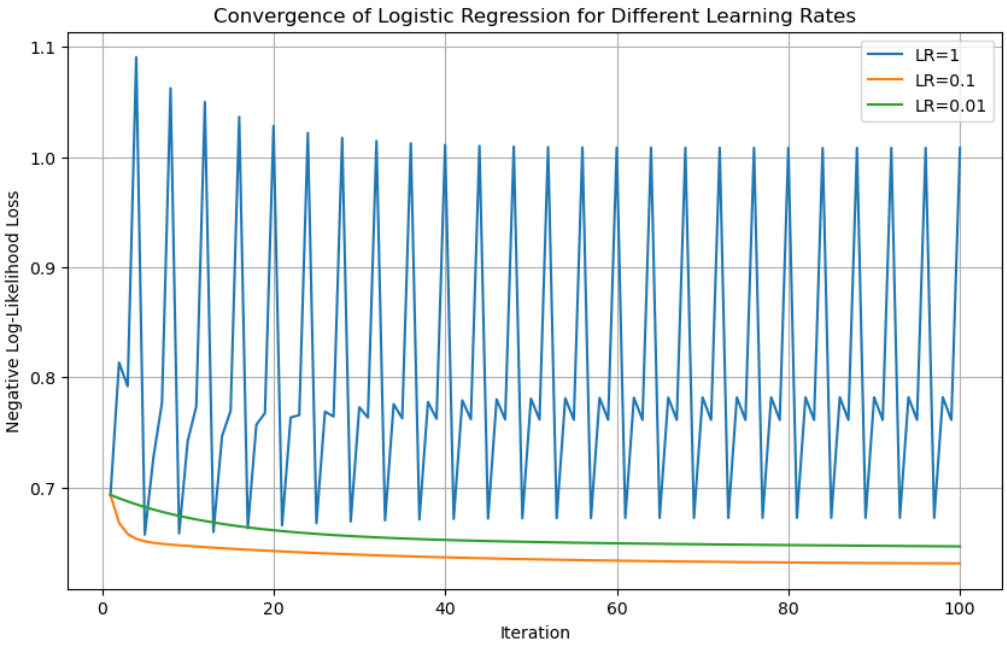

Figure 1: 2(b) Output

Therefore, the best learning rate out of the three choices is 0.1.

(c)

Input

```
# Re-train using the best learning rate (LR = 0.1)
best_lr = 0.1  # From part (b)
num_steps = 100

# Initialize w to zero
w_final = torch.zeros(dim, requires_grad=True)

# Define optimizer
optimizer = torch.optim.SGD([w_final], lr=best_lr)

# Run gradient descent
for step in range(num_steps):
    loss = logistic_neg_log_likelihood(w_final, X, Y)
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

# Detach w_final to remove gradient tracking
w_final = w_final.detach()

# Define function for decision boundary
f_final = lambda X: logistic_probability(w_final, X)

# Visualize decision boundary (provided function)
visualize_boundary(X, Y, f_final)
```

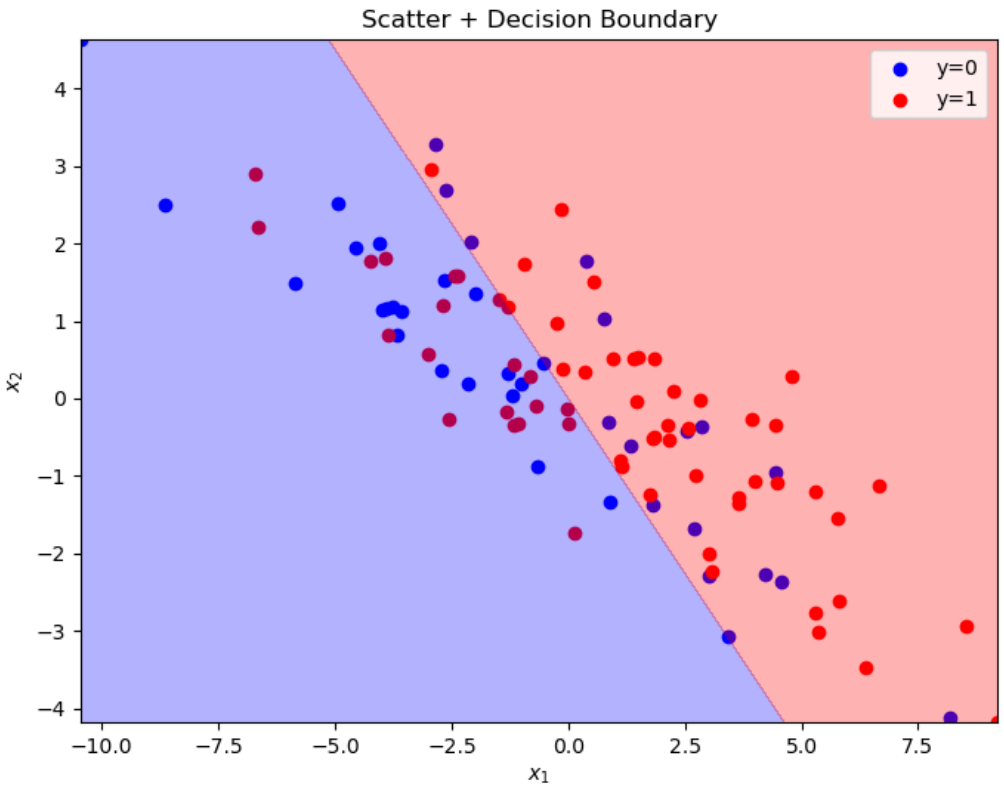

Figure 2: 2(c) Output

3 Classification on a swiss-roll

(a)

Input

```
# Learning rates to test
learning_rate = 0.2
num_steps = 200

# Initialize w to zero
w_final = torch.zeros(dim, requires_grad=True)

# Define optimizer (SGD with given learning rate)
optimizer = torch.optim.SGD([w_final], lr=learning_rate)

# Store history for loss and gradient norm
loss_history = []
accuracy_history = []
grad_norm_history = []

for step in range(num_steps):
    optimizer.zero_grad()
    loss = logistic_neg_log_likelihood(w_final, X, Y)  # Compute loss
    loss.backward()  # Compute gradient

    # Compute predictions
    y_pred = (logistic_probability(w_final, X) >= 0.5).float()  # Convert probabilities to
        binary labels
    accuracy = (y_pred == Y).float().mean().item()  # Compute accuracy

    # Compute gradient norm (L2 norm)
    grad_norm = torch.norm(w_final.grad, p=2).item()

    # Update weights
    optimizer.step()

    # Store metrics
    loss_history.append(loss.item())
    accuracy_history.append(accuracy)
    grad_norm_history.append(grad_norm)

# Detach w_final to remove gradient tracking
w_final = w_final.detach().clone()

# Define function for decision boundary
f_final = lambda X: logistic_probability(w_final, X)

# Visualize decision boundary (provided function)
visualize_boundary(X, Y, f_final)
```

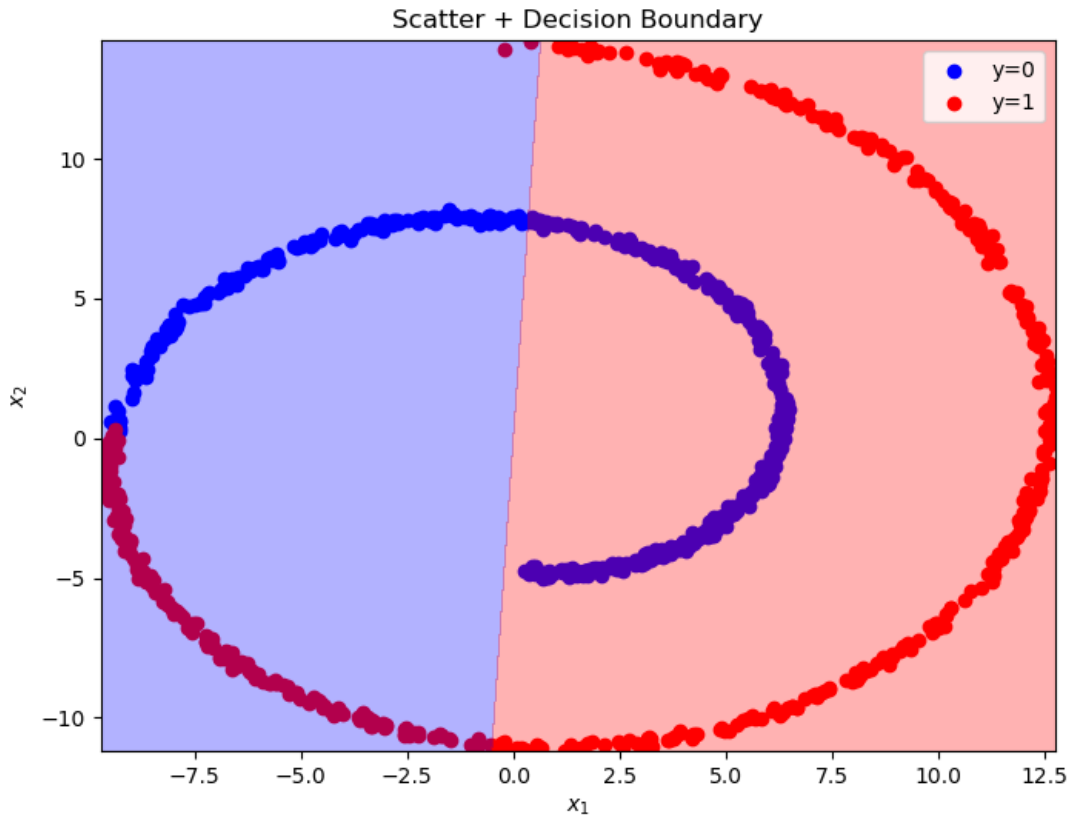

Figure 3: 3(a) decision boundary

(b)

Input

```
plt.figure(figsize=(8, 5))
plt.plot(range(1, num_steps + 1), loss_history, label="Negative Log-Likelihood", color='blue')
plt.xlabel("Iteration")
plt.ylabel("Negative Log-Likelihood Loss")
plt.title("Loss Convergence on Swiss Roll Data")
plt.legend()
plt.grid(True)
plt.show()

plt.figure(figsize=(8, 5))
plt.plot(range(1, num_steps + 1), accuracy_history, label="Accuracy", color='green')
plt.xlabel("Iteration")
plt.ylabel("Accuracy")
plt.title("Accuracy Convergence on Swiss Roll Data")
plt.legend()
plt.grid(True)
plt.show()

plt.figure(figsize=(8, 5))
plt.plot(range(1, num_steps + 1), torch.log(torch.tensor(grad_norm_history) + 1e-8), label="
    Log(Gradient Norm)", color='red')
plt.xlabel("Iteration")
plt.ylabel("Log(Gradient Norm)")
plt.title("Gradient Norm Convergence")
plt.legend()
plt.grid(True)
plt.show()
```

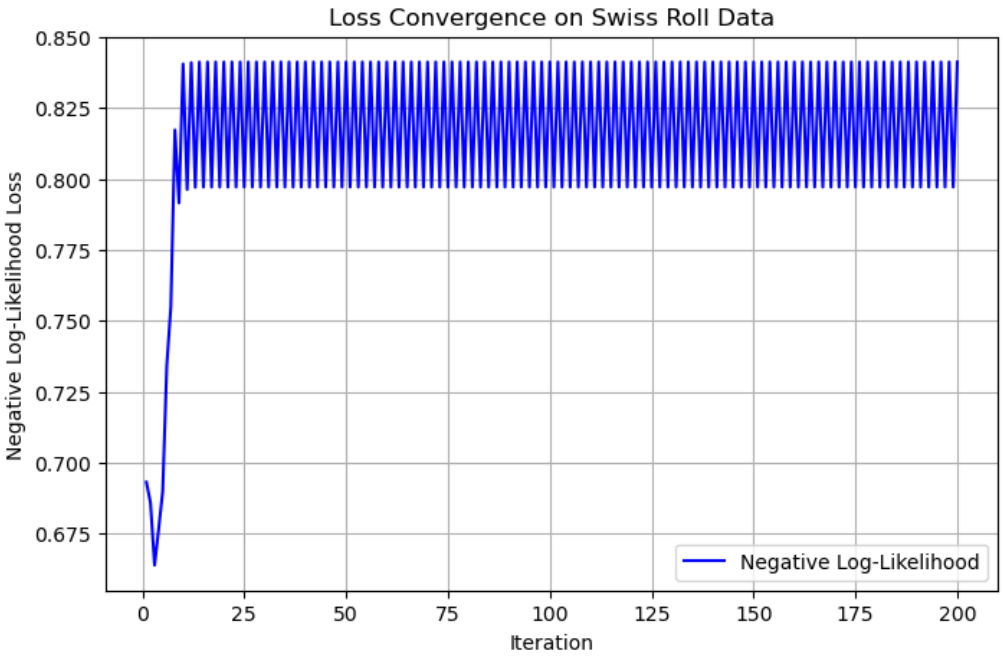

Figure 4: 3(b) negative log likelihood against GD iterations

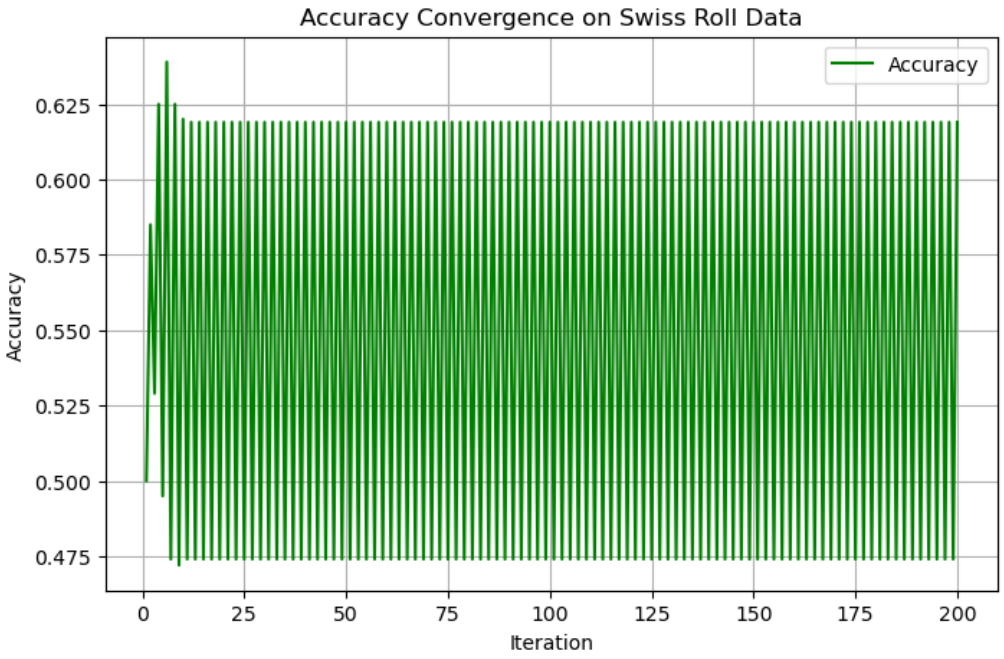

Figure 5: 3(b) log(accuracy) against GD iterations

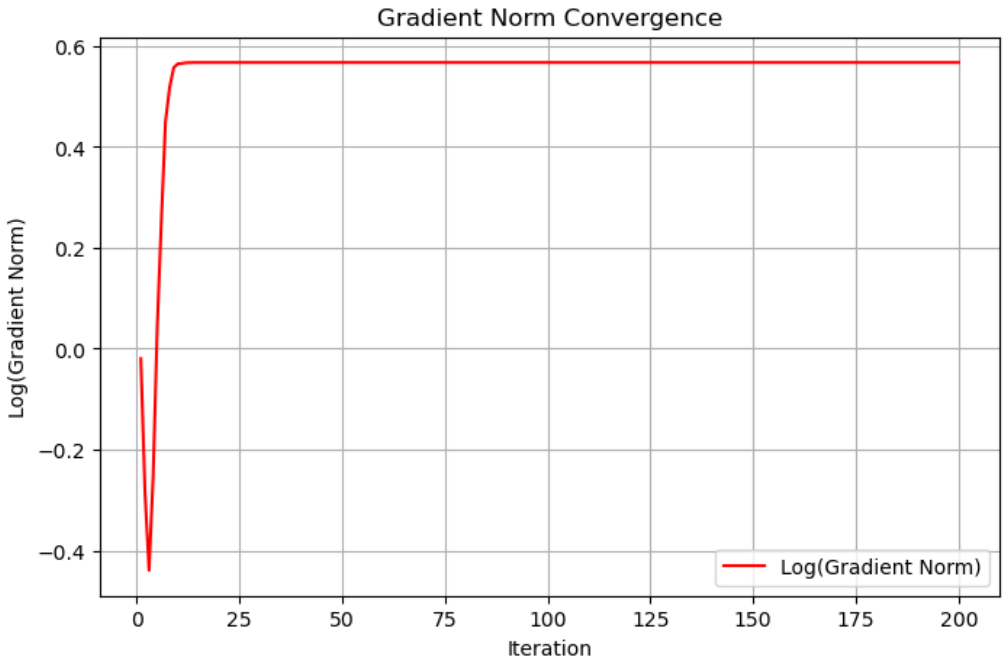

Figure 6: 3(c) log(gradient norm) against GD iterations

- The loss has not converged.
- The classification accuracy has not converged to 100.
- The gradient norm has not converged to 0.

(c)

Input

```
from torch.nn.functional import relu
import torch.nn.functional as F

def MLP_probability(w1,w2,b2,X):

    hidden = F.relu(X @ w1)  # First layer + ReLU
    logits = hidden @ w2 + b2  # Second layer + bias
    return torch.sigmoid(logits)  # Apply sigmoid for probability output
```

(d)

Input

```
torch.manual_seed(0)
w1 = torch.randn([2,10])*0.1
w2 = torch.randn([10,1])*0.1
b2 = torch.randn(1)*0.1
w1.requires_grad = True
w2.requires_grad = True
b2.requires_grad = True

# Training parameters
num_epochs = 3000
lr = 0.3

# Define optimizer
optimizer = torch.optim.SGD([w1, w2, b2], lr=lr)

# Store history for loss and accuracy
loss_history = []
accuracy_history = []
grad_norm_history = []
# Training loop
for epoch in range(num_epochs):
    # Compute predictions
    p = MLP_probability(w1, w2, b2, X)

    # Compute loss (Negative Log-Likelihood / Binary Cross-Entropy)
    loss = F.binary_cross_entropy(p, Y.view(-1,1))  # Ensure Y is the correct shape

    # Compute predictions and accuracy
    y_pred = (p >= 0.5).float()  # Convert probability to class label
    accuracy = (y_pred.view(-1) == Y.view(-1)).float().mean().item()  # Compute classification
        accuracy

    # Backpropagation
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    # Compute gradient norm
    total_grad_norm = torch.norm(
        torch.cat([w1.grad.view(-1), w2.grad.view(-1), b2.grad.view(-1)])
    ).item()

    # Store metrics
    loss_history.append(loss.item())
    accuracy_history.append(accuracy)
    grad_norm_history.append(total_grad_norm)

    # Print every 500 epochs
    if epoch % 500 == 0:
        print(f"Epoch␣{epoch}:␣Loss␣=␣{loss.item():.4f},␣Accuracy␣=␣{accuracy␣*␣100:.2f}%")

# Ensure final accuracy is 100%
print(f"Final␣Accuracy:␣{accuracy_history[-1]␣*␣100:.2f}%")

w1_final = w1.detach().clone()
w2_final = w2.detach().clone()
b2_final = b2.detach().clone()
f_final = lambda X : MLP_probability(w1_final,w2_final,b2_final,X) #will not work until you
    implement MLP_probability
visualize_boundary(X,Y,f_final)
```

Output

```
Epoch 0: Loss = 0.6738, Accuracy = 55.30%
Epoch 500: Loss = 0.0929, Accuracy = 99.50%
Epoch 1000: Loss = 0.0536, Accuracy = 99.80%
Epoch 1500: Loss = 0.0398, Accuracy = 99.80%
Epoch 2000: Loss = 0.0327, Accuracy = 99.80%
Epoch 2500: Loss = 0.0283, Accuracy = 100.00%
Final Accuracy: 100.00%
```

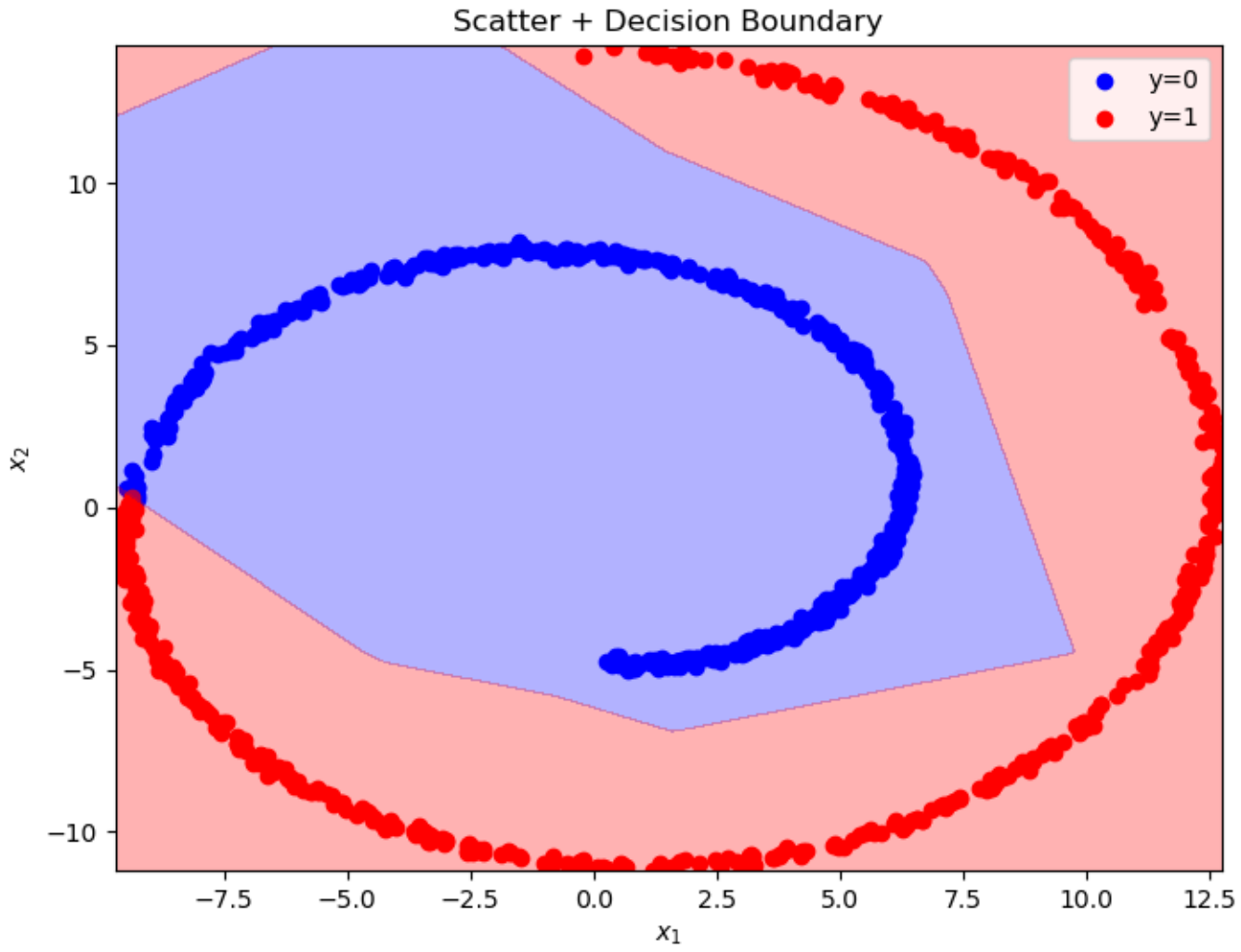

Figure 7: 3(d) Decision Boundary

(e)

Input

```
loss_history = torch.tensor(loss_history)
accuracy_history = torch.tensor(accuracy_history)
grad_norm_history = torch.tensor(grad_norm_history)

# Plot Negative Log Likelihood vs. Iterations
plt.figure(figsize=(8, 5))
plt.plot(range(1, num_epochs + 1), loss_history, label="Negative Log-Likelihood", color='blue'
    )
plt.xlabel("Iteration")
plt.ylabel("Negative Log-Likelihood Loss")
plt.title("MLP Loss Convergence")
plt.legend()
plt.grid(True)
plt.show()

# Plot Accuracy vs. Iterations
plt.figure(figsize=(8, 5))
plt.plot(range(1, num_epochs + 1), accuracy_history, label="Accuracy", color='green')
plt.xlabel("Iteration")
plt.ylabel("Accuracy")
plt.title("MLP Accuracy Convergence")
plt.legend()
plt.grid(True)
plt.show()

# Plot Log(Gradient Norm) vs. Iterations
plt.figure(figsize=(8, 5))
plt.plot(range(1, num_epochs + 1), torch.log(grad_norm_history + 1e-8), label="Log(Gradient
    Norm)", color='red')
plt.xlabel("Iteration")
plt.ylabel("Log(Gradient Norm)")
plt.title("MLP Gradient Norm Convergence")
plt.legend()
plt.grid(True)
plt.show()
```

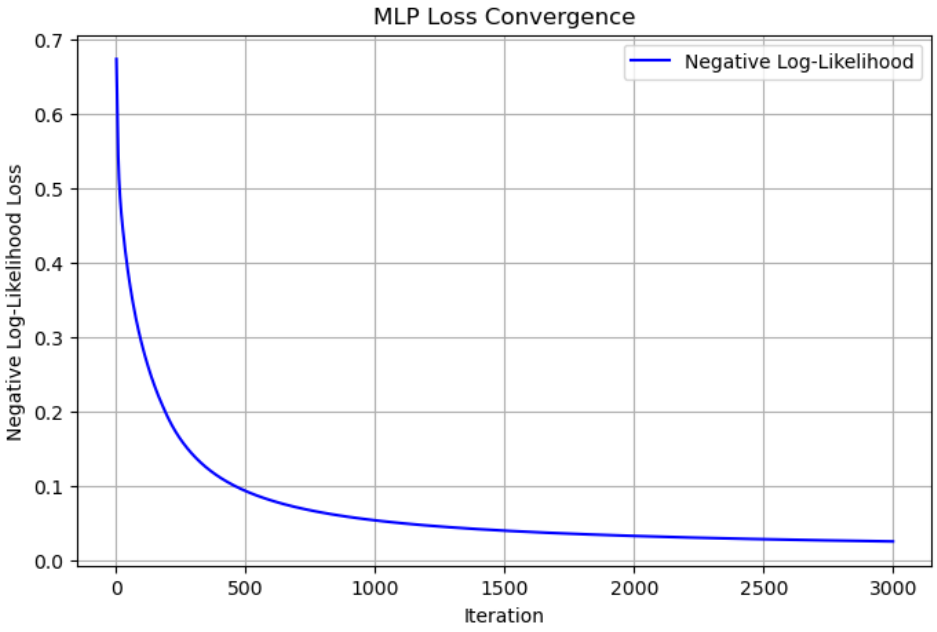

Figure 8: 3(e) MLP negative log likelihood

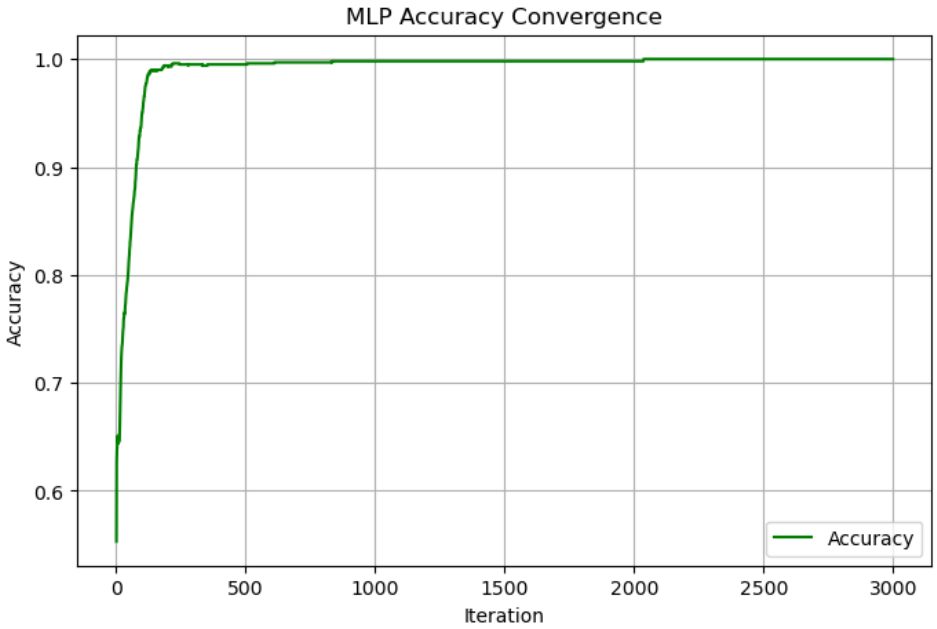

Figure 9: 3(e) accuracy against iterations

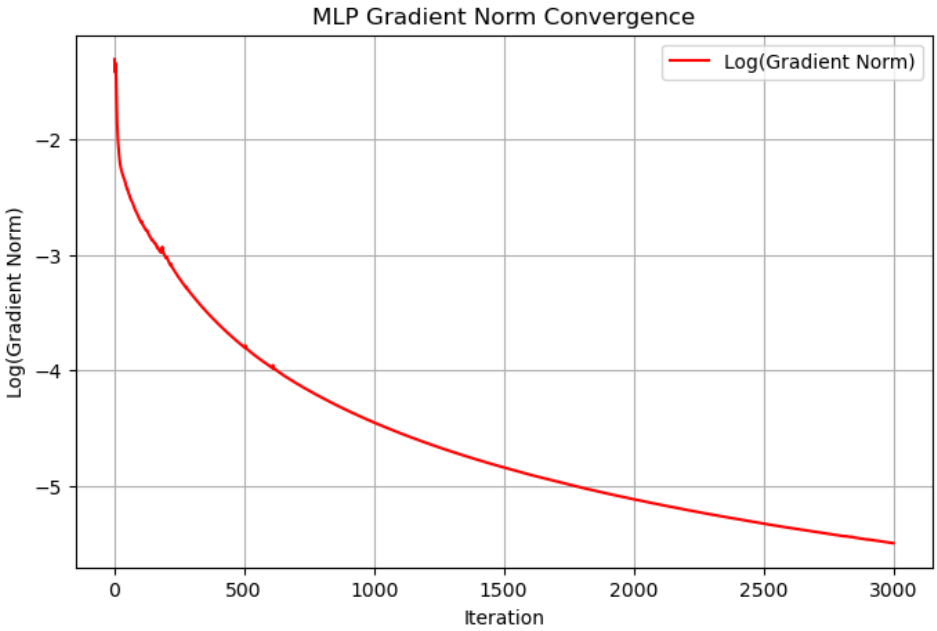

Figure 10: log(gradient norm) against iterations

- The loss has converged.
- The classification accuracy has converged to 100.
- The gradient norm has converged to 0.
- Data distribution is clearly non-linearly separable, whereas logistic regression is a linear classifier. As a result, logistic regression can only learn a linear decision boundary (i.e., a straight line). However, data in this problem requires a more complex decision boundary, which can be better achieved using non-linear models like MLP. Therefore, the results(loss, accuracy, gradient norm) in (e) is much better than that in (b).