MTH416 Assignment # 1

2466883 Jiaxin Liu

March 2025

1

a)

Known that:

$$f(x) = x^T Ax,$$

where A is a symmetric matrix.
the gradient

$$\nabla_x f(x) = \begin{bmatrix} \frac{\partial f(x)}{\partial x_1} \\ \frac{\partial f(x)}{\partial x_2} \\ ... \\ \frac{\partial f(x)}{\partial x_n} \end{bmatrix}$$

So the gradient is:

$$\nabla_x f(x) = (A + A^T)x = 2Ax \quad \text{(since } A \text{ is symmetric, } A = A^T\text{)}.$$

The Hessian matrix is the derivative of the gradient

$$\nabla_x^2 f(x) = \frac{\partial}{\partial x}(2Ax) = 2A.$$

b)

Known that:

$$g(A) = x^T Ax,$$

where A is a symmetric matrix.
The gradient of a scalar with respect to a matrix A is a matrix whose elements are $\frac{\partial g(A)}{\partial A_{ij}}$. Expanding $g(A)$:

$$g(A) = \sum_{i=1}^{N} \sum_{j=1}^{N} A_{ij} x_i x_j.$$

Taking the partial derivative with respect to A_{kl}:

$$\frac{\partial g(A)}{\partial A_{kl}} = x_k x_l.$$

Thus, the gradient matrix is:

$$\nabla_A g(A) = xx^T.$$

c)

Known that:

$$h(A) = \text{trace}(xx^T A) = x_1^2 A_{11} + x_2^2 A_{22} + ... + x_N^2 A_{NN} h(A) = \sum_{i=1}^{N} x_i^2 * A_{ii}$$

Thus, its gradient is

$$\nabla_A \text{trace}(xx^T A) = x_1^2 + x_2^2 + ... + x_N^2.$$

2

a)

Known that:

$$L_1(\theta) = \frac{1}{2}\|X\theta - y\|_2^2,$$

The gradient of $L_1(\theta)$ with respect to θ is:

$$\nabla_\theta L_1(\theta) = X^T(X\theta - y).$$

Setting the gradient to zero:

$$X^T(X\theta - y) = 0$$
$$X^T X\theta = X^T y.$$

Since $X^T X$ is invertible. Thus, the solution is:

$$\theta = (X^T X)^{-1} X^T y.$$

b)

Known that:

$$L_2(\theta) = \frac{1}{2}\|X\theta - y\|_2^2 + \lambda\|\theta\|_2^2,$$

The gradient of $L_2(\theta)$ with respect to θ is:

$$\nabla_\theta L_2(\theta) = \frac{1}{2}X^T(X\theta - y) + \lambda\theta.$$

Setting the gradient to zero:

$$X^T(X\theta - y) + 2\lambda\theta = 0 \implies (X^T X + 2\lambda I)\theta = X^T y.$$

The matrix $X^T X + 2\lambda I$ is always invertible (since $\lambda > 0$), and the solution is:

$$\theta = (X^T X + 2\lambda I)^{-1} X^T y.$$

c)

- **Linear Regression**:
  - The solution is $\beta_{\text{linear}} = (X^T X)^{-1} X^T y$.
  - Assumes $X^T X$ is invertible (requires $\text{rank}(X) = P$).
  - No regularization, which can lead to overfitting if P is large or X is ill-conditioned.
- **Ridge Regression**:
  - The solution is $\beta_{\text{ridge}} = (X^T X + 2\lambda I)^{-1} X^T y$.
  - Adds a regularization term $\frac{\lambda}{2}\|\beta\|_2^2$ to the objective function.
  - Ensures $X^T X + \lambda I$ is always invertible, even if $X^T X$ is singular.
  - Controls overfitting by shrinking the coefficients β towards zero.
- **Differences**:
  - Ridge regression introduces a bias (through λ) to reduce variance, improving generalization.
  - Linear regression is unbiased but can have high variance, especially with multicollinearity or small datasets.
  - Ridge regression is more robust to ill-conditioned or singular $X^T X$.

3

a)

For a fair coin, the probability of heads and tails are both 0.5. The entropy H is calculated as:

$$H = -\sum_i p_i \log_2 p_i.$$

For the fair coin:

$$H = -0.5 \log_2 0.5 - 0.5 \log_2 0.5.$$

Since $\log_2 0.5 = -1$:

$$H = 0.5 + 0.5 = 1.$$

b)

For a fair die with 6 outcomes, each outcome has probability $\frac{1}{6}$. The entropy H is:

$$H = -\sum_{i=1}^{6} p_i \log_2 p_i.$$

Since $p_i = \frac{1}{6}$ for all i:

$$H = -6 \times \left(\frac{1}{6} \log_2 \frac{1}{6} \right).$$

$$H = -\log_2 \frac{1}{6} = \log_2 6 \approx 2.585 \text{ bits}.$$

c)

For a biased coin with $p_1 = 0.7$ and $p_0 = 0.3$, the entropy H is:

$$H = -p_0 \log_2 p_0 - p_1 \log_2 p_1.$$

Substitute the probabilities:

$$H = -0.7 \log_2 0.7 - 0.3 \log_2 0.3.$$

Calculate $\log_2 0.7$ and $\log_2 0.3$:

$$\log_2 0.7 \approx -0.5146, \quad \log_2 0.3 \approx -1.737.$$

Thus:

$$H = -0.7 \times (-0.5146) - 0.3 \times (-1.737) \approx 0.3602 + 0.5211 = 0.8813.$$

d)

$$H(p, q) = H(p) + D_{\text{KL}}(p \parallel q).$$

By definition:

$$H(p, q) = -\sum_i p_i \log_2 q_i,$$

$$H(p) = -\sum_i p_i \log_2 p_i,$$

$$D_{\text{KL}}(p \parallel q) = \sum_i p_i \log_2 \left(\frac{p_i}{q_i} \right).$$

Add $H(p)$ and $D_{\text{KL}}(p \parallel q)$:

$$H(p) + D_{\text{KL}}(p \parallel q) = -\sum_i p_i \log_2 p_i + \sum_i p_i \log_2 \left(\frac{p_i}{q_i} \right).$$

Simplify:

$$H(p) + D_{\mathrm{KL}}(p \parallel q) = -\sum_i p_i \log_2 p_i + \sum_i p_i(\log_2 p_i - \log_2 q_i).$$

$$H(p) + D_{\mathrm{KL}}(p \parallel q) = -\sum_i p_i \log_2 q_i = H(p, q).$$

e)

The KL divergence $D_{\mathrm{KL}}(p \parallel q)$ is always non-negative:

$$D_{\mathrm{KL}}(p \parallel q) \geq 0.$$

Proof: Known that:

$$\log a < a - 1$$

$$-\log a > 1 - a$$

$$\sum_i p_i \log_2 \left(\frac{p_i}{q_i}\right) = -\sum_i p_i \log_2 \left(\frac{q_i}{p_i}\right) \geq \sum_i p_i(1 - \frac{q_i}{p_i} = \sum_i p_i - q_i) = \sum_i p_i - \sum_i q_i = 1 - 1 = 0.$$

So:

$$D_{\mathrm{KL}}(p \parallel q) \geq 0.$$

4

Known that: The forward difference approximation is:

$$\frac{df(x)}{dx} \approx \frac{f(x+h) - f(x)}{h}.$$

Using Taylor series expansion for $f(x+h)$:

$$f(x+h) = f(x) + hf'(x) + O(h^2).$$

Substitute into the forward difference formula:

$$\frac{f(x+h) - f(x)}{h} = \frac{hf'(x) + O(h^2)}{h} = f'(x) + O(h^2).$$

The error term is:

$$\text{Error} = f'(x) + O(h) - f'(x) = O(h).$$

The central difference approximation is:

$$\frac{df(x)}{dx} \approx \frac{f(x+h) - f(x-h)}{2h}.$$

Using Taylor series expansion for $f(x+h)$ and $f(x-h)$:

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2} f''(x) + O(h^3),$$

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2} f''(x) - O(h^3).$$

$$\frac{f(x+h) - f(x-h)}{2h} = \frac{2hf'(x) + 2O(h^3)}{2h} = f'(x) + O(h^2).$$

The error term is:

$$\text{Error} = f'(x) + O(h^2) - f'(x) = O(h^2).$$

The forward difference has an error term of order $O(h)$, while the central difference has an error term of order $O(h^2)$. This shows that the central difference is more accurate for small error.

5

Konwn that the binary SVM loss function is:

$$L_i = C \cdot \max(0, 1 - y_i\theta^\top x_i) + R(\theta),$$

The multi-class SVM loss function is:

$$L_i = \sum_{j \neq y_i} \max(0, w_j^\top x_i - w_{y_i}^\top x_i + 1) + \lambda R(W),$$

where: $W = [w_1, w_2, \ldots, w_K]$ is the weight matrix, and K is the number of classes;
For binary classification (with class labels $y_i \in \{-1, 1\}$), parameterize the weight matrix W as:

$$W = [w_1, w_2], \quad \text{where } w_1 = -\theta,\ w_2 = \theta.$$

The multi-class SVM loss function is:

$$L_i = \sum_{j \neq y_i} \max(0, w_j^\top x_i - w_{y_i}^\top x_i + 1) + \lambda R(W).$$

For the binary classification scenario:

When $y_i = 1$: The true class is $w_2 = \theta$, and the non-true class is $w_1 = -\theta$. The loss term is:

$$\max(0, w_1^\top x_i - w_2^\top x_i + 1) = \max(0, (-\theta^\top x_i) - (\theta^\top x_i) + 1) = \max(0, 1 - 2\theta^\top x_i).$$

Since there are only two classes, the summation contains only one term. When $y_i = -1$: The true class is $w_1 = -\theta$, and the non-true class is $w_2 = \theta$. The loss term is:

$$\max(0, w_2^\top x_i - w_1^\top x_i + 1) = \max(0, \theta^\top x_i - (-\theta^\top x_i) + 1) = \max(0, 1 + 2\theta^\top x_i).$$

The binary labels are $y_i \in \{-1, 1\}$, the loss terms for both cases can be combined as:

$$L_i = \max(0, 1 - y_i \cdot 2\theta^\top x_i) + \lambda R(W).$$

The regularization term for multi-class SVM is typically the Frobenius norm of the weight matrix:

$$R(W) = \|W\|_F^2 = \|w_1\|^2 + \|w_2\|^2 = \|\theta\|^2 + \|\theta\|^2 = 2\|\theta\|^2.$$

Substituting this into the loss function:

$$\lambda R(W) = 2\lambda\|\theta\|^2.$$

Let $C = 2\lambda$ for the binary SVM. Then the regularization term can be rewritten as:

$$\lambda R(W) = C\|\theta\|^2.$$

The multi-class SVM loss function becomes:

$$L_i = \max(0, 1 - y_i \cdot 2\theta^\top x_i) + C\|\theta\|^2.$$

The original binary SVM loss function is:

$$L_i = C \cdot \max(0, 1 - y_i\theta^\top x_i) + R(\theta).$$

To match this, perform the following adjustments: Scale the weights in the multi-class SVM as $\theta \to \frac{\theta}{2}$, i.e., define $w_1 = -\frac{\theta}{2},\ w_2 = \frac{\theta}{2}$. Ensure consistency in the regularization term: $R(\theta) = \|\theta\|^2$, and $C = 2\lambda$.
The multi-class SVM loss function now reduces to:

$$L_i = C \cdot \max(0, 1 - y_i\theta^\top x_i) + R(\theta).$$

When there are only two classes, by constraining the weight matrix of the multi-class SVM as $W = [-\theta, \theta]$ (or an equivalent scaled form), its loss function becomes identical to the binary SVM loss function. So, the binary SVM loss is a special case of the multi-class SVM loss for binary classification.

6

Known that:

$$L_i(W) = -w_{y_i}^T x_i + \log\left(\sum_j e^{w_j^\top x_i}\right).$$

The gradient of $L_i(W)$ with respect to W. The gradient has two terms:
1. Gradient of $-w_{y_i}^T x_i$: This term is linear in w_{y_i}, so its gradient is:

$$\nabla_W\left(-w_{y_i}^T x_i\right) = \begin{cases} -x_i & \text{if } w = w_{y_i}, \\ 0 & \text{otherwise.} \end{cases}$$

Can be written as:

$$\nabla_W\left(-w_{y_i}^T x_i\right) = -x_i \cdot \mathbf{1}_{y_i},$$

where $\mathbf{1}_{y_i}$ is a one-hot vector with 1 at the position corresponding to the true class y_i and 0 elsewhere.
2. Gradient of $\log\left(\sum_j e^{w_j^T x_i}\right)$: Let $Z = \sum_j e^{w_j^T x_i}$. The gradient of $\log Z$ with respect to w_k is:

$$\nabla_{w_k} \log Z = \frac{1}{Z} \cdot \nabla_{w_k} Z.$$

Since $Z = \sum_j e^{w_j^T x_i}$, the gradient of Z with respect to w_k is:

$$\nabla_{w_k} Z = e^{w_k^T x_i} \cdot x_i.$$

Therefore:

$$\nabla_{w_k} \log Z = \frac{e^{w_k^T x_i}}{Z} \cdot x_i = p_k \cdot x_i,$$

where $p_k = \frac{e^{w_k^T = x_i}}{\sum_j e^{w_j^T x_i}}$ is the softmax probability for class k.
Combining this for all classes, the gradient of $\log Z$ with respect to W is:

$$\nabla_W \log Z = x_i \cdot \mathbf{p}^\top,$$

where $\mathbf{p} = [p_1, p_2, \ldots, p_K]^T$ is the vector of softmax probabilities.
Combining the gradients of the two terms, we get:

$$\nabla_W L_i(W) = -x_i \cdot \mathbf{1}_{y_i}^T + x_i \cdot \mathbf{p}^T.$$

This can be simplified as:

$$\nabla_W L_i(W) = x_i \cdot (\mathbf{p} - \mathbf{1}_{y_i})^T.$$

The gradient of the cross-entropy loss with respect to W is:

$$\nabla_W L_i(W) = x_i \cdot (\mathbf{p} - \mathbf{1}_{y_i})^T$$

- x_i: The input feature vector for the i-th example.
- $\mathbf{p}$: The vector of softmax probabilities for all classes. $p_k = \frac{e^{w_k^T = x_i}}{\sum_j e^{w_j^T x_i}}$
- $\mathbf{1}_{y_i}$: A one-hot vector where the true class y_i is 1 and all other entries are 0.

7

As mentioned in the assignment:
(3):

$$v^{k+1} = \rho v^k - \alpha \nabla f(w^k),$$
$$w^{k+1} = w^k + v^{k+1}.$$

(4):

$$v^{k+1} = \rho v^k + \nabla f(w^k),$$
$$w^{k+1} = w^k - \alpha v^{k+1}.$$

Plug v^k into w^k to get itEasy to know that:
As for (3)

$$w_e^{k+1} = w^k + \rho v^k - \alpha \nabla f(w^k)$$

As for (4)

$$w_e^{k+1} = w^k - \rho \alpha v^k - \alpha \nabla f(w^k)$$

The only difference between the two schemes is the sign of the ρv^k term.
Let $v^k = -\frac{1}{\alpha}\tilde{v}^k$ in (4).
Substitute $\tilde{v}^{k+1}$:

$$w^{k+1} = w^k - \alpha\rho\tilde{v}^k - \alpha \nabla f(w^k).$$

Since $v^k = -\tilde{v}^k$, this becomes:

$$w^{k+1} = w^k + \alpha * \frac{1}{\alpha}\rho v^k - \alpha \nabla f(w^k) = w^k + \rho v^k - \alpha \nabla f(w^k).$$

This is identical to the update rule for w^{k+1} in Scheme (3).

8

(1)

a)

$$\frac{\partial L}{\partial U} = \frac{\partial L}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial \theta} * \frac{\partial \theta}{\partial U}$$
$$L = cross - entropy(y, \hat{y}), \hat{y} = softmax(\theta)$$

Easy to konw that:

$$L = -\sum_{i=1}^{C} y_i \log \hat{y}_i$$

$$\hat{y} = \begin{bmatrix} \frac{e_1^\theta}{\sum_{i=1}^C e^{\theta_i}} \\ \frac{e_2^\theta}{\sum_{i=1}^C e^{\theta_i}} \\ \cdots \\ \frac{e_C^\theta}{\sum_{i=1}^C e^{\theta_i}} \end{bmatrix}$$

$$\frac{\partial L}{\partial \theta} = \frac{\partial L}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial \theta} = \hat{y} - y = \begin{bmatrix} \frac{e_1^\theta}{\sum_{i=1}^C e^{\theta_i}} - y_1 \\ \frac{e_2^\theta}{\sum_{i=1}^C e^{\theta_i}} - y_2 \\ \cdots \\ \frac{e_C^\theta}{\sum_{i=1}^C e^{\theta_i}} - y_C \end{bmatrix}$$

$$\theta = Uh + b_2, \text{where } U \in R^{C*H} \text{ and } b_2 \in R^C$$

$$\frac{\partial \theta}{\partial U} = \begin{bmatrix} h_1 & h1 & ... & h_1 \\ h_2 & h_2 & ... & h_2 \\ ... & ... & ... & ... \\ h_C & h_C & ... & h_C \end{bmatrix}_{C*H} = U$$

$$\frac{\partial L}{\partial U} = U^T * (y - \hat{y})$$

b)

$$\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial \theta} * \frac{\partial \theta}{\partial b_2}$$

Since$\theta = Uh + b_2$:

$$\frac{\partial \theta}{\partial b_2} = 1$$

So:

$$\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial \theta} * 1 = y - \hat{y} = \begin{bmatrix} \frac{e_1^\theta}{\sum_{i=1}^C e^{\theta_i}} - y_1 \\ \frac{e_2^\theta}{\sum_{i=1}^C e^{\theta_i}} - y_2 \\ ... \\ \frac{e_C^\theta}{\sum_{i=1}^C e^{\theta_i}} - y_C \end{bmatrix}$$

c)

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial \theta} * \frac{\partial \theta}{\partial h} * \frac{\partial h}{\partial z} * \frac{\partial z}{\partial W}$$

Already konw that:

$$\frac{\partial L}{\partial \theta} = \frac{\partial L}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial \theta} = \hat{y} - y = \begin{bmatrix} \frac{e_1^\theta}{\sum_{i=1}^C e^{\theta_i}} - y_1 \\ \frac{e_2^\theta}{\sum_{i=1}^C e^{\theta_i}} - y_2 \\ ... \\ \frac{e_C^\theta}{\sum_{i=1}^C e^{\theta_i}} - y_C \end{bmatrix}$$

$$\frac{\partial \theta}{\partial h} = \begin{bmatrix} \theta_{11} & \theta_{12} & ...\theta_{1h} \\ \theta_{21} & \theta_{12} & ...\theta_{1h} \\ ... & ... & ... \\ \theta_{h1} & \theta_{h2} & ...\theta_{hh} \end{bmatrix}_{C*H}$$

konwn that:

$$z = Wx + b_1, \text{ where } W \in R^{C*H} \text{ and } b_1 \in R^H$$

$$h = ReLU(z)$$

So:

$$\frac{\partial h}{\partial z} = \mathbf{1}(z > 0)_{H*D} = \begin{cases} 1 & z_{C*H} > 0 \\ 0 & z_{C*H} \leq 0 \end{cases}$$

$$\frac{\partial h}{\partial z} = \begin{bmatrix} \mathbf{1}(z > 0)_{1*1} & \mathbf{1}(z > 0)_{1*2} & ... & \mathbf{1}(z > 0)_{1*D} \\ \mathbf{1}(z > 0)_{2*1} & \mathbf{1}(z > 0)_{2*2} & ... & \mathbf{1}(z > 0)_{2*D} \\ ... & & & \\ \mathbf{1}(z > 0)_{H*1} & \mathbf{1}(z > 0)_{H*2} & ... & \mathbf{1}(z > 0)_{H*D} \end{bmatrix}_{H*D}$$

$$z = Wx + b1$$

So:

$$\frac{\partial z}{\partial w} = x$$

$$\frac{\partial L}{\partial W} = U^T * (y - \hat{y}) * 1(z > 0)_{H*D} * x^T$$

d)

$$\frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial z} * \frac{\partial z}{\partial b_1}$$

Because $\frac{\partial L}{\partial z}$ we already have. So we just need to compute $\frac{\partial z}{\partial b_1}$

$$z = Wx + b_1$$

$$\frac{\partial z}{\partial b_1} = 1$$

$$\frac{\partial L}{\partial b_1} = U^T * (y - \hat{y}) * 1(z > 0)_{H*D}$$

e)

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} * \frac{\partial z}{\partial x}$$

we already have $\frac{\partial L}{\partial z}$. So we only need to get $\frac{\partial z}{\partial x}$:

$$\frac{\partial z}{\partial x} = \begin{bmatrix} w_{11} & w_{12} & ... & w_{1D} \\ w_{21} & w_{12} & ... & w_{2D} \\ ... & ... & ... & \\ w_{H1} & w_{H2} & ... & w_{HD} \end{bmatrix}_{H*D} = W$$

So:

$$\frac{\partial L}{\partial x} = W^T * (U^T * (y - \hat{y}) * 1(z > 0)_{H*D})$$

(2)and(3)

[3]:

```
import numpy as np
# Define the network parameters
D, H, C = 3, 4, 2  # Input dimension (D=3), hidden layer size (H=4), number of␣
 ↪output classes (C=2)

W = np.array([
    [1.0, 0.5, -0.3],    # Weights for the neurons in the hidden layer
    [0.2, -1.0, 0.8],
    [0.4, 0.7, -0.9],
    [-0.6, 0.1, 0.5]
])      # Shape (4,3)

b1 = np.array([0.1, -0.2, 0.3, -0.4])  # Hidden layer bias (4,)

U = np.array([
    [0.5, -0.3, 0.2, 0.1],    # Weights for the 1st output class
    [-0.4, 0.6, 0.7, -0.8]    # Weights for the 2nd output class
])          # Shape (2,4)

b2 = np.array([0.2, -0.1])  # Output layer bias (2,)

x = np.array([0.5, -1.0, 0.8])  # Input vector (3,)

y_true = np.array([0, 1])       # True label and output

# Define the forward pass
def relu(z):
    return np.maximum(0, z)

def softmax(theta):
    exp_theta = np.exp(theta - np.max(theta))
    return exp_theta / np.sum(exp_theta)

def cross_entropy(y, y_true):
    return -np.sum(y_true * np.log(y))

def forward(x, W, b1, U, b2):
    z = W @ x + b1
    h = relu(z)
    theta = U @ h + b2
    y = softmax(theta)
    loss = cross_entropy(y, y_true)
    return loss, z, h, theta, y

# Compute analytical gradients
loss, z, h, theta, y = forward(x, W, b1, U, b2)
dL_dtheta = y - y_true
```

```python
dL_dU = np.outer(dL_dtheta, h)
dL_db2 = dL_dtheta
dL_dh = U.T @ dL_dtheta
dL_dz = dL_dh * (z > 0)
dL_dW = np.outer(dL_dz, x)
dL_db1 = dL_dz
dL_dx = W.T @ dL_dz

# Numerical gradient approximation
h_numerical = 1e-5

def numerical_gradient(f, param):
    param_plus = param + h_numerical
    param_minus = param - h_numerical
    return (f(param_plus) - f(param_minus)) / (2 * h_numerical)

# Compute numerical gradients
numerical_dL_dU = np.zeros_like(U)
for i in range(U.shape[0]):
    for j in range(U.shape[1]):
        def f_U(u):
            U_temp = U.copy()
            U_temp[i, j] = u
            loss, _, _, _, _ = forward(x, W, b1, U_temp, b2)
            return loss
        numerical_dL_dU[i, j] = numerical_gradient(f_U, U[i, j])

numerical_dL_db2 = np.zeros_like(b2)
for i in range(b2.shape[0]):
    def f_b2(b):
        b2_temp = b2.copy()
        b2_temp[i] = b
        loss, _, _, _, _ = forward(x, W, b1, U, b2_temp)
        return loss
    numerical_dL_db2[i] = numerical_gradient(f_b2, b2[i])

numerical_dL_dW = np.zeros_like(W)
for i in range(W.shape[0]):
    for j in range(W.shape[1]):
        def f_W(w):
            W_temp = W.copy()
            W_temp[i, j] = w
            loss, _, _, _, _ = forward(x, W_temp, b1, U, b2)
            return loss
        numerical_dL_dW[i, j] = numerical_gradient(f_W, W[i, j])

numerical_dL_db1 = np.zeros_like(b1)
```

```
for i in range(b1.shape[0]):
    def f_b1(b):
        b1_temp = b1.copy()
        b1_temp[i] = b
        loss, _, _, _, _ = forward(x, W, b1_temp, U, b2)
        return loss
    numerical_dL_db1[i] = numerical_gradient(f_b1, b1[i])

numerical_dL_dx = np.zeros_like(x)
for i in range(x.shape[0]):
    def f_x(x_val):
        x_temp = x.copy()
        x_temp[i] = x_val
        loss, _, _, _, _ = forward(x_temp, W, b1, U, b2)
        return loss
    numerical_dL_dx[i] = numerical_gradient(f_x, x[i])

# Compare analytical and numerical gradients
print("Analytical dL_dU:\n", dL_dU)
print("Numerical dL_dU:\n", numerical_dL_dU)
print("Analytical dL_db2:\n", dL_db2)
print("Numerical dL_db2:\n", numerical_dL_db2)
print("Analytical dL_dW:\n", dL_dW)
print("Numerical dL_dW:\n", numerical_dL_dW)
print("Analytical dL_db1:\n", dL_db1)
print("Numerical dL_db1:\n", numerical_dL_db1)
print("Analytical dL_dx:\n", dL_dx)
print("Numerical dL_dx:\n", numerical_dL_dx)
```

```
Analytical dL_dU:
 [[ 0.          0.38865327  0.          0.        ]
 [-0.         -0.38865327 -0.         -0.        ]]
Numerical dL_dU:
 [[ 0.          0.38865327  0.          0.        ]
 [ 0.         -0.38865327  0.          0.        ]]
Analytical dL_db2:
 [ 0.25237225 -0.25237225]
Numerical dL_db2:
 [ 0.25237225 -0.25237225]
Analytical dL_dW:
 [[ 0.         -0.          0.        ]
 [-0.11356751  0.22713503 -0.18170802]
 [-0.          0.         -0.        ]
 [ 0.         -0.          0.        ]]
Numerical dL_dW:
 [[ 0.          0.          0.        ]
 [-0.11356751  0.22713503 -0.18170802]
 [ 0.          0.          0.        ]
```

```
 [ 0.          0.          0.        ]]
Analytical dL_db1:
 [ 0.         -0.22713503 -0.          0.        ]
Numerical dL_db1:
 [ 0.         -0.22713503  0.          0.        ]
Analytical dL_dx:
 [-0.04542701  0.22713503 -0.18170802]
Numerical dL_dx:
 [-0.04542701  0.22713503 -0.18170802]
```

[4]:

```python
import torch

# Define the parameters
W_torch = torch.tensor(W, requires_grad=True)
b1_torch = torch.tensor(b1, requires_grad=True)
U_torch = torch.tensor(U, requires_grad=True)
b2_torch = torch.tensor(b2, requires_grad=True)
x_torch = torch.tensor(x, requires_grad=True)
y_true_torch = torch.tensor(y_true)

# Forward pass
z_torch = W_torch @ x_torch + b1_torch
h_torch = torch.relu(z_torch)
theta_torch = U_torch @ h_torch + b2_torch
y_torch = torch.softmax(theta_torch, dim=0)
loss_torch = -torch.sum(y_true_torch * torch.log(y_torch))

# Backward pass
loss_torch.backward()

# Compare gradients
print("PyTorch dL_dU:\n", U_torch.grad)
print("PyTorch dL_db2:\n", b2_torch.grad)
print("PyTorch dL_dW:\n", W_torch.grad)
print("PyTorch dL_db1:\n", b1_torch.grad)
print("PyTorch dL_dx:\n", x_torch.grad)
```

```
PyTorch dL_dU:
 tensor([[ 0.0000,  0.3887,  0.0000,  0.0000],
        [-0.0000, -0.3887, -0.0000, -0.0000]], dtype=torch.float64)
PyTorch dL_db2:
 tensor([ 0.2524, -0.2524], dtype=torch.float64)
PyTorch dL_dW:
 tensor([[ 0.0000, -0.0000,  0.0000],
        [-0.1136,  0.2271, -0.1817],
        [ 0.0000, -0.0000,  0.0000],
        [ 0.0000, -0.0000,  0.0000]], dtype=torch.float64)
PyTorch dL_db1:
```

```
 tensor([ 0.0000, -0.2271,  0.0000,  0.0000], dtype=torch.float64)
PyTorch dL_dx:
 tensor([-0.0454,  0.2271, -0.1817], dtype=torch.float64)
```