Lab 1

Prakhar Gupta 2103126
Ayushman Baghel   2103305
Abhay Tiwari 2103301
Vibesh Kumar 2103140

Section 1: Neural Network Implementation

This section defines a simple neural network with functions for forward and backward propagation. It includes the following functions:

forward_layer_matrixMult(x, w): Forward pass for matrix multiplication layer.

```
def forward_layer_matrixMult(x, w):
    # both w,x are row vectors
    # x : n * d, w : d * 1
    # N = xw

    return x @ w
```

backward_layer_matrixMult(x, w): Backward pass for matrix multiplication layer.

```
def backward_layer_matrixMult(x, w):
    # N = xw
    # dN/dw, dN/dx

    return x, w
```

forward_layer_biasAdd(N, b): Forward pass for bias addition layer.

```
def forward_layer_biasAdd(N, b):
    # N : n * 1, b : 1 * 1
    # P = N + b

    # N: n * k, b: k * 1

    return N + b
```

backward_layer_biasAdd(N, b): Backward pass for bias addition layer.

```
def backward_layer_biasAdd(N, b):
    # P = N + b (n * 1)
    # dP/dN = Identity (n * n)
    # dP/db = 1 (n * 1)

    return np.identity(N.shape[0]), np.ones((N.shape[0], 1))
```

forward_layer_sigmoid(P): Forward pass for the sigmoid activation layer.

```
def forward_layer_sigmoid(P):
    # Q = 1 / 1 + e^-P_i

    return 1 / (1 + np.exp(-P))
```

backward_layer_sigmoid(P): Backward pass for the sigmoid activation layer.

```
def backward_layer_sigmoid(P):
    # dQ_i/dP_i = Q_i * (1 - Q_i)

    Q = forward_layer_sigmoid(P)
    Q1 = Q * (1 - Q)

    return np.diag(Q1)
```

forward_layer_softMax(Q): Forward pass for the softmax activation layer.

```
def forward_layer_softMax(Q):
    denom = np.sum(np.exp(Q))
    return np.exp(Q) / denom
```

backward_layer_softMax(P): Backward pass for the softmax activation layer.

```
def backward_layer_softMax():
    # dQ/dP = formula written in copy

    sum_ek = np.sum(np.exp(P))
    n = P.shape[0]

    dQ = np.zeros(n, n)

    for i in range(n):
        for j in range(n):
            if i == j :
                dQ[i][j] = sum_ek

            dQ[i][j] -= np.exp(P[i])
            dQ[i][j] *= np.exp(P[j])
            dQ[i][j] /= sum_ek * sum_ek

    return dQ
```

forward_layer_meanSqrLoss(P, y): Forward pass for the mean squared loss layer.

```
def forward_layer_meanSqrLoss(P,y):
    # P : n * 1, y : n * 1
    # L = 1/n * sum((P - y) ^ 2)     : (1 * 1)

    return np.sum((P - y)**2)
```

backward_layer_meanSqrLoss(P, y): Backward pass for the mean squared loss layer.

```
def backward_layer_meanSqrLoss(P, y):
    # L = 1/n * sum((P - y) ^ 2)     : (1 * 1)
    # dL/dP = 2 * (P - y) : (n * 1)

    return 2 * (P - y)
```

forward_layer_crossEntropy(P, y): Forward pass for the cross-entropy loss layer.

```
def forward_layer_crossEntropy(P,y):
    #y can be multiclass or
    #P,y both are matrix of same shape
        # returns loss_vector

    loss = np.log((P ** y))
    return -np.sum(loss)
```

backward_layer_crossEntropy(Q, y): Backward pass for the cross-entropy loss layer.

```
def backward_layer_crossEntropy(Q,y):
    # dl/dp[i][j] = Q[i][j] - y[i][j]
    return Q - y
```

Section 2: Boston Dataset (Q2)

This section loads the California housing dataset, normalizes the data, and uses a simple neural network for regression. It includes stochastic gradient descent (SGD) training and prints the error during each iteration.

Section 3: Iris Dataset (Q3)

This section loads the Iris dataset, performs one-hot encoding for multiclass classification, and uses a simple neural network for classification. It includes gradient descent training and prints the error during each iteration. The final accuracy on the training set is also printed.