Machine Learning(CS331)
Lab Assignment 1

Hasanali Malvi(2103309)
Amit Sonaje(2103107)
Mulayam(2103122)
Aryan Goel(2103109)

4 February 2024

1 Implement the Following Operations (Forward and Backward Pass)

a) **Matrix Multiplication Layer (WX):**

Forward Pass: Given input matrix X and weight matrix W, the forward pass computes the matrix product $X \times W$.

$$\text{MatMul}(X, w) = Xw \tag{1}$$

```
# Matrix Multiplication Forward Pass
def F1(X, W):
    return X@W
```

Backward Pass: In the backward pass, the gradients with respect to W and X are calculated using the chain rule to update the weights during the training process.

$$\frac{\partial N_{ij}}{\partial W_{pq}} = \begin{cases} X_{il} & \text{if } j = m \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

$$\frac{\partial N_{ij}}{\partial X_{ab}} = \begin{cases} W_{bj} & \text{if } i = a \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

```
# Matrix Multiplication Backward Pass with respect to X
def B1_X(N,X,W):
  row1 = np.shape(X)[0]
  col1 = np.shape(X)[1]
  row2 = np.shape(N)[0]
  col2 = np.shape(N)[1]
  shape = (row1, col1, row2, col2)
  N_X = np.zeros(shape)
  for i in range(row1):
    for j in range(col1):
      for k in range(col2):
        N_X[i][j][i][k] = W[j][k]
  return N_X

# Matrix Multiplication Backward Pass with respect to W
def B1_W(N,X,W):
  return X


```

b) **Bias Addition Layer:**

Forward Pass: Adds a bias vector to the output of the previous layer or operation.

$$\text{BiasAddition}(X, b) = X + b \tag{4}$$

```
# Bias Addition Layer
def F2(N, B):
  return N + B


```

Backward Pass: The gradient of the loss with respect to the bias is calculated to update the bias during training.

$$\frac{\partial P_{ij}}{\partial N_{lm}} = \begin{cases} 1 & \text{if } j = m \text{ and } i = l \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

```
# Backward Pass of Bias Addition Layer with respect to N
def B2_N(P):
  n = np.shape(P)[0]
  return np.eye(n)
```

```

# Backward Pass of Bias Addition Layer with respect to P
def B2_B(P):
  shape = (np.shape(P)[0],1)
  return np.ones(shape)

```

c) **Mean Squared Loss Layer:**

Forward Pass: Computes the mean squared difference between the predicted values and the actual target values.

$$MeanSquaredError : \sum_{i=1}^{n}(P_i - Y_i)^2 \tag{6}$$

```
# Mean Squared Loss Layer
def F3(P, Y):
    return np.sum((P - Y)**2)

```

Backward Pass: Derivative of the loss with respect to the predicted values is calculated, allowing the model to update its parameters during backpropagation.

$$\frac{\partial L}{\partial P} = (2 \times (P - Y))^T \tag{7}$$

```
# Backward Pass of Mean Squared Loss Layer
def B3(P, Y):
  return 2*(P-Y)

```

d) **Softmax Layer:**

Forward Pass: Applies the softmax function to convert raw scores into probabilities, ensuring that the output values sum to 1.

$$\text{softmax}(P_i) = \frac{e^{P_i}}{\sum_{j=1}^{n} e^{P_j}} \tag{8}$$

```
# Softmax Layer
def F4(P):
  exp_matrix = np.exp(P)
  return exp_matrix / np.sum(exp_matrix, axis=1, keepdims=True)


```

Backward Pass: Derivatives of the loss with respect to the input are computed, facilitating the update of weights during training.

$$\frac{\partial Q_{ij}}{\partial P_{lm}} = Q_{ij}(\delta_{il} - Q_{im}) \tag{9}$$

```
# Backward Pass of Softmax Layer
def B4(P, Q):
  row1 = np.shape(P)[0]
  col1 = np.shape(P)[1]
  row2 = np.shape(Q)[0]
  col2 = np.shape(Q)[1]
  shape = (row1, col1, row2, col2)
  Q_P = np.zeros(shape)
  for i in range(row1):
    for j in range(col1):
      for k in range(col2):
```

```
        if k==j:
          Q_P[i][j][i][k] = Q[i][j]*(1-Q[i][k])
        else:
          Q_P[i][j][i][k] = -1*Q[i][j]*(Q[i][k])
  return Q_P


```

e) **Sigmoid Layer:**

Forward Pass: Applies the sigmoid activation function element-wise to the input, squashing values between 0 and 1.

$$\text{sigmoid}(P) = \frac{1}{1+e^{-P}} \tag{10}$$

```
# Sigmoid Layer
def F5(P):
  return 1 / (1 + np.exp(-P))

```

Backward Pass: Calculates the derivative of the loss with respect to the input, allowing for the update of weights during backpropagation.

$$\frac{\partial Q}{\partial P} = Q \cdot (1 - Q) \tag{11}$$

```
# Backward Pass of Sigmoid layer
def B5(P,Q):
  row1 = np.shape(P)[0]
  shape = (row1, row1)
  Q_P = np.zeros(shape)
  for i in range(row1):
    Q_P[i][i] = Q[i][1]*(1-Q[i][1])
  return Q_P

```

f) **Cross-Entropy Loss Layer:**

Forward Pass: Computes the cross-entropy loss, measuring the difference between predicted probabilities and true class probabilities.

$$L = -\sum_{i=1}^{n}\sum_{j=1}^{m} Y_{ij}\log(Q_{ij}) \tag{12}$$

```
# Cross Entropy Loss Layer
def F6(Q, Y):
  logQ=np.log(Q)
  return -1*np.sum(Y*logQ)

```

Backward Pass: Calculates the gradient of the loss with respect to the predicted probabilities, facilitating the update of model parameters during backpropagation.

$$\frac{\partial L}{\partial Q_{ij}} = -\frac{Y_{ij}}{Q_{ij}} \tag{13}$$

```
# Backward Pass of Cross Entropy Loss Layer
def B6(Q, Y, L):
  L_Q = np.zeros(Q.shape)
  for i in range(0,Q.shape[0]):
    for j in range(0,Q.shape[1]):
      L_Q[i][j]=-1*(Y[i][j]/Q[i][j])
  return L_Q

```

2 Using the sklearn.load california() function, obtain boston house pricing dataset.Train a regression model using the operations implemented above. You need to write a stochastic gradient descent function to train.

- In this analysis, we utilized the `fetch_california_housing` function from the `sklearn_datasets` module to obtain the California housing dataset. The data is stored in the `housing` variable, loaded as follows:

```
# Fetching housing data
from sklearn.datasets import fetch_california_housing
housing=fetch_california_housing()
```

- We have considered only the first 1000 points from the dataset of 20000 and normalized the data using max normalization. From the dataset, we have split the data into 75% for training and 25% for testing.

```
# Taking top 1000 samples and normalizing taking max value
X_2 = housing.data[0:1000] / np.max(housing.data[0:1000], axis=0, keepdims=True)
Y_2 = (housing.target[0:1000] / np.max(housing.target[0:1000])).reshape(-1, 1)
```

```
# Train-Test Split
X_train_2, X_test_2, Y_train_2, Y_test_2 = train_test_split(X_2, Y_2, test_size=0.25, random_state
    =42)
```

- Then we have generated w and b randomly using the random function. Afterward, the learning rate is set to be 0.001, and the number of iterations is chosen to be 1000.

```
# Initializing random weights and bias
shape = (np.shape(X_2)[1],1)
W_2 = np.random.randn(*shape)
B_2 = np.random.rand()
print(W_2)
print(B_2)
```

- We implemented a regression model training function, `regression`, using stochastic gradient descent with a batch size of 1. For each iteration, a loss (l) was calculated, and both the forward and backward passes were executed to update the weights and bias.
- l is then appended to the list L. For each iteration, we applied stochastic gradient descent.
- Finally, the weights and bias were updated using the calculated gradients:

$$w = w - \eta \frac{\partial L}{\partial w} \tag{14}$$

$$b = b - \eta \frac{\partial L}{\partial b} \tag{15}$$

$$\frac{\partial L}{\partial W} = \left(\frac{\partial L}{\partial P} \cdot \frac{\partial P}{\partial N} \cdot \frac{\partial N}{\partial W} \right)^T \tag{16}$$

$$\frac{\partial L}{\partial b} = \left(\frac{\partial L}{\partial P} \cdot \frac{\partial P}{\partial b} \right)^T \tag{17}$$

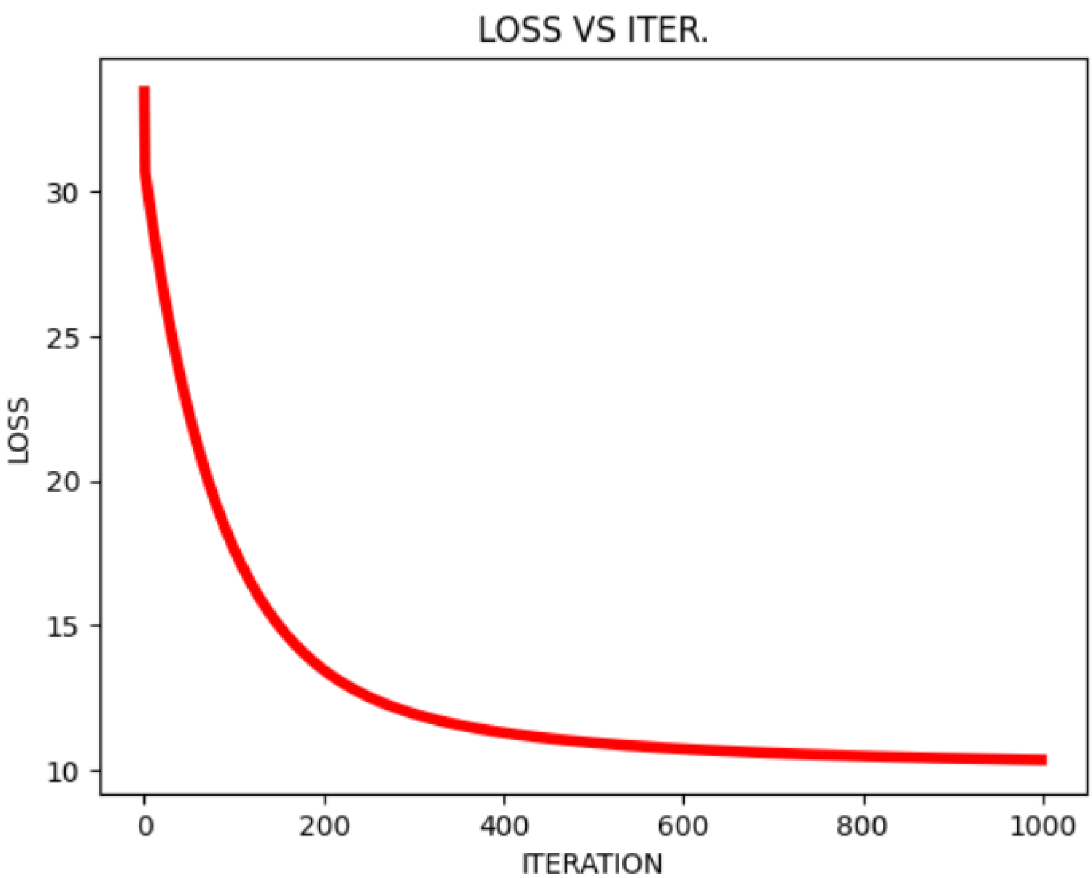
```
# Regression model which returns final weights and final bias
def regression(learning_rate, max_iterations, X, Y, W, B):
  iteration = []
  L = []
  #Stochastic Gradient Descent
  for k in range(max_iterations):
    l=0
    for i in range(X.shape[0]):
      input = X[i:i+1,:]
      output = Y[i:i+1,:]
```

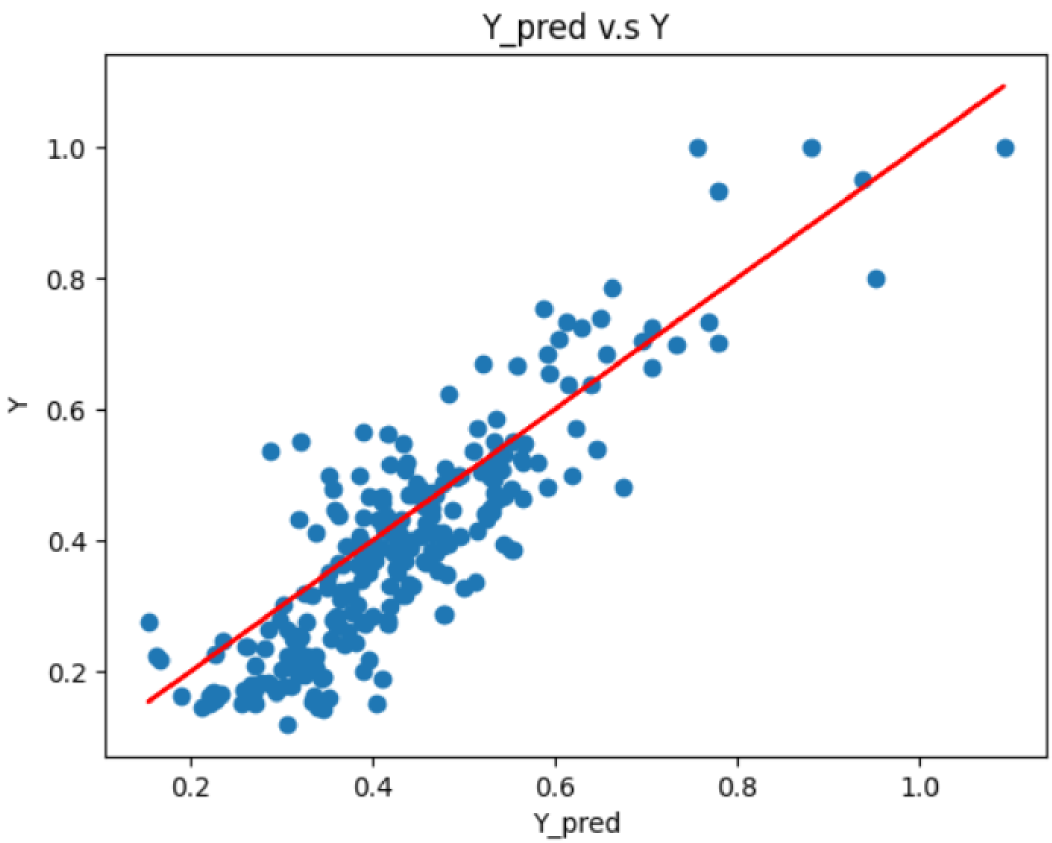
```
        N = F1(input,W)
        P = F2(N,B)
        l += F3(P,output)
        dl_dp = B3(P,output)
        dp_dn = B2_N(P)
        dn_dw = B1_W(N,input,W)
        dp_db = B2_B(P)
        dl_dw = ((dl_dp)@(dp_dn)@(dn_dw)).T
        dl_db = ((dl_dp)@(dp_db))[0][0]
        W = W - learning_rate*dl_dw
        B = B - learning_rate*dl_db
      L.append(l)
      iteration.append(k)

  return (L, iteration,W,B)
```

```
learning_rate = 0.001
max_iterations = 1000

# Training model
(L_2,iteration_2,W_final_2,B_final_2) = regression(learning_rate, max_iterations, X_train_2,
    Y_train_2, W_2, B_2)
```





3 Load the iris dataset in sklearn. This data sets consists of 3 different types of irises' (Setosa, Versicolour, and Virginica) petal and sepal length, stored in a 150x4 numpy.ndarray. Using the operations implemented above create a multi-class classifier (Cross entropy loss + soft max)

- Importing the dataset using `sklearn` and storing it in `dataset`.
- Data is shuffled because it is given in the order.

```
dataset=load_iris()
X_3=dataset.data
Y_3=dataset.target
X_3, Y_3 = shuffle(X_3, Y_3, random_state=42)
```

- Data is split into 75% and 25%.

```
X_train_3, X_test_3, Y_train_3, Y_test_3 = train_test_split(X_3, Y_3, test_size=0.25, random_state
    =42)
```

- The dataset consists of a 150x4 NumPy array, where `Y_target` gives the labels Y corresponding to Setosa, Versicolour, Virginica (numbers 0, 1, 2). We have implemented one-hot encoding for the data, and the encoded values are stored in `Y`.

```
#one-hot encoding
Y_one_hot = np.zeros((Y_train_3.shape[0],3))
for i in range(Y_train_3.shape[0]):
  Y_one_hot[i][Y_train_3[i]] = 1
Y_train_3 = Y_one_hot
# print(Y_train)
```

- Then, we initialized weights and bias using random values and proceeded to implement the forward pass.

```python
# Initializing weights and bias randomly
shape = (np.shape(X_3)[1],3)
W_3 = np.random.randn(*shape)
B_3 = np.random.rand()
print(W_3)
print(B_3)
```

- Forward pass
  F_1: Matrix Multiplication
  F_2: Bias addition layer
  F_4: Softmax function
  F_6: Cross entropy loss layer

```python
# Forward Pass for multilabel classification
# Output: Softmax matrix and Loss
def forward_pass(X,W,B,Y):
  N=F1(X,W)
  P=F2(N,B)
  Q=F4(P)
  L=F6(Q,Y)
  return (L,P,Q)
```

- This is a function to store $\frac{\partial L}{\partial P} = Q - Y$.

```python
def L_P(Y,Q):
  ans=np.zeros(Q.shape)
  for i in range(Q.shape[0]):
    for j in range(Q.shape[1]):
      ans[i][j] = Q[i][j]-Y[i][j]
  return ans
```

- Backward pass:

$$\frac{\partial L}{\partial W} = (\frac{\partial L}{\partial P} \cdot \frac{\partial P}{\partial W})^T = \left(\frac{\partial P}{\partial W}\right)^T \cdot \left(\frac{\partial L}{\partial P}\right)^T = X^T \cdot (Q - Y) \tag{18}$$

$$\frac{\partial L}{\partial B} = \left(\frac{\partial L}{\partial P} \cdot \frac{\partial P}{\partial B}\right)^T = \left(\frac{\partial P}{\partial B}\right)^T \cdot \left(\frac{\partial L}{\partial P}\right)^T = \sum_{i=1}^{n} \sum_{j=1}^{k} (Q_{ij} - Y_{ij}) \tag{19}$$

```python
# Backward Pass for multilabel classification
# Output: Gradient of Loss w.r.t weight and bias
def backward_pass(X,W,B,P,Q,Y):
  dl_dp = L_P(Y,Q)
  dl_dw = (X.T)@(dl_dp)
  dl_db=np.sum(dl_dp)
  return (dl_dw,dl_db)
```

- This function uses `W_final` and `B_final` to calculate `Q`, from which we select the output with the maximum probability.

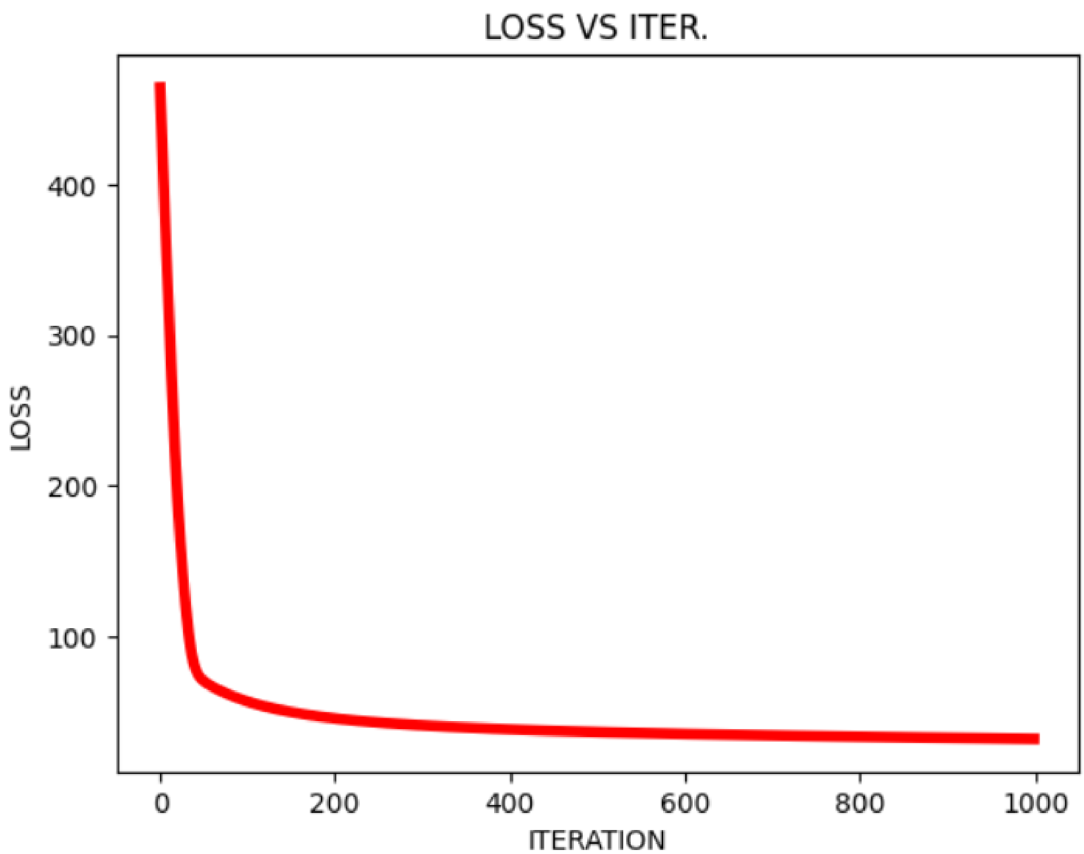
```python
# Predict output
def predict(X,W,B):
  N=F1(X,W)
  P=F2(N,B)
  Q=F4(P)
  return np.argmax(Q,axis=1)
```

- This function calculates the accuracy.

```python
# Shows Performance of the model
def accuracy_score(predictions, Y_test):
  count = 0;
  for i in range(Y_test.shape[0]):
    if predictions[i]==Y_test[i]:
      count+=1
  return (count/Y_test.shape[0])*100
```

- This is the main function that returns the Loss, iteration, updated W, and B.

```python
# Function to train model
# Output: Final weight and Final bias
def multilabel_classification(learning_rate , max_iteration , W, B,X_train,Y_train):
  L = []
  iteration = []
  for i in range(max_iteration):
    l,P,Q = forward_pass(X_train,W,B,Y_train)
    (dl_dw,dl_db) = backward_pass(X_train,W,B,P,Q,Y_train)
    W = W - learning_rate*dl_dw
    B = B - learning_rate*dl_db
    L.append(l)
    iteration.append(i)
  return (L,iteration,W,B)
```




```
[127] # Accuracy score
      print(accuracy_score(predictions,Y_test_3))

      97.36842105263158
```