Name: ______________________________ Student ID: __________________________ Group No: ___

LAB 4: Loss function and optimizing of Gradient descent

Objective:

1) To learn loss function
   a. Mean squared error
   b. Absolute error
   c. Huber loss
   d. Binary cross-entropy
2) To learn optimizing the loss function : Gradient descent

Theory

1. Loss function and Cost function are often used inter-changeably but they are different:
   1.1.1. The loss function measures the network performance on a single data-point
   1.1.2. The cost function is the average of the losses over the entire training dataset
   1.2. Mean Squared Error (MSE) is defined as mean or average of the square of the difference between expectation and prediction value can be determined by

$$MSE = \frac{1}{n}\sum(y - f(x))^2$$

   1.3. The Mean Absolute Error (is the quantifiable difference between a measured value and its actual value. It is obtained by taking the absolute value between the predicted or observed value and the expectation value

$$MAE = \frac{1}{n}\sum|y - f(x)|$$

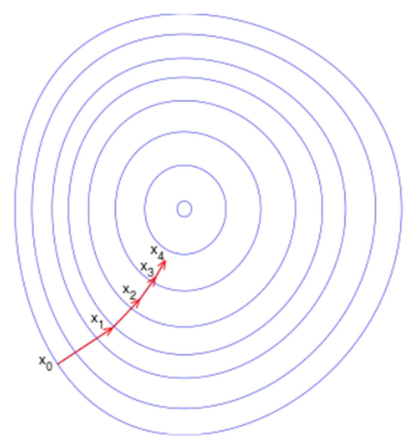
   1.4. Huber loss function is a combination of the mean squared error function and the absolute value function.

$$Loss = \begin{cases} \frac{1}{2}(y - f(x))^2, & if\ |y - f(x)| \le \delta \\ \delta|y - f(x)| - \frac{1}{2}\delta^2, & otherwise \end{cases}$$

1.5. Binary Cross Entropy is a loss function used in machine learning and deep learning to measure the difference between predicted binary outcomes and actual binary labels. The pi is the probability of class 1, and (1-pi) is the probability of class 0, y is target and p is prediction.

$$\text{Log loss} = \frac{1}{N} \sum_{i=1}^{N} -\left(y_i * \log(p_i) + (1-y_i) * \log(1-p_i) \right)$$

2. Gradient descent is an optimization algorithm that's used when training a machine learning model. It's based on a convex function and tweaks its parameters iteratively to minimize a given function to its local minimum.



https://en.wikipedia.org/wiki/Gradient_descent

Procedures

Part 1:

1. Mean squared error (MSE)

1.1. Type the python code for mean squared error (MSE). Adjust expectation, prediction value and record the results in Table 1.1

```python
import numpy as np

# Create numpy arrays for the actual and predicted values
expect = np.array([0,0,1,1])
predict = np.array([0,0,0,0])

sum = 0

# for loop for iteration
for i in range(len(expect)):
    sum += (expect[i] - predict[i])**2

error = sum/(len(expect))

print('Mean squared error:',error)
print(expect.mean())
```

Table 1.1 Mean squared error (MSE)

Expectation	Prediction	Mean squared error (MSE)
[1, 0, 1, 0]	[1, 0, 0, 0]	
[12.5, 0.5, 13.2, 1]	[11, 0.5, 1.2, 4]	
[0.5, 0.9, -3, 1]	[0.9, 1, -3, 1]	
[0.8, 1.9, -10, 13]	[0.8, 1.9, -10, 12]	

2. Mean absolute error (MAE)

2.1. Type the python code for absolute error Adjust expectation, prediction value and record the results in Table 1.2

```python
# Absolute Error
import numpy as np

# Create numpy arrays for the actual and predicted values
expect = np.array([1,0,0,1])
predict = np.array([1,1,0,1])

sum = 0

# for loop for iteration
for i in range(len(expect)):
    sum += abs(expect[i] - predict[i])

error = sum/(len(expect))

# Result
print('Absolute Error: ',error)
```

Table 1.2 Mean absolute error (MAE)

Expectation	Prediction	Mean absolute error (MAE)
[1, 0, 1, 0]	[1, 0, 0, 0]	
[12.5, 0.5, 13.2, 1]	[11, 0.5, 1.2, 4]	
[0.5, 0.9, -3, 1]	[0.9, 1, -3, 1]	
[0.8, 1.9, -10, 13]	[0.8, 1.9, -10, 12]	

3. Huber loss

   3.1. Type the python code for Huber loss and plot graph as below

```python
# Huber loss
import matplotlib.pyplot as plt
import numpy as np

# Huber loss function
def huber_loss(predict, target, delta):
    huber_mse = 0.5*(target-predict)**2
    huber_mae = delta * (np.abs(target - predict) - 0.5 * delta)
    return np.where(np.abs(target - predict) <= delta, huber_mse, huber_mae)


predict = np.array([i for i in range(-110,110)])
target = np.array([2*i for i in range(-110,110)])
delta = 1

y_huber = huber_loss(predict,target,delta)
loss = target-predict
plt.plot(loss, y_huber, 'green')
plt.grid(True, which='major')
plt.title(' Huber loss vs Loss ')
plt.ylabel('Huber loss')
plt.xlabel('Loss')
plt.show()
```

delta = 1

delta = 100

4. Binary cross entropy

4.1. Type the python code for binary cross and plot graph as below

```
#Binary cross entropy

import numpy as np
import matplotlib.pyplot as plt

def binary_cross_entropy(target, predict):
    loss = -1*(target*np.log(predict) + (1-target)*np.log(1-predict))
    return loss

target = np.ones(10)
predict = np.linspace(0., 1., 10)
loss_binary = binary_cross_entropy(target, predict)


plt.plot(predict,loss_binary, "o--")
plt.xlabel("prediction")
plt.ylabel("loss_prediction")
plt.grid()
plt.show()
```

5. Gradient descent

5.1. Type the python code for Gradient descent and plot graph as below

Loss vs Weights, iterations = 10

Prediction vs Target, iterations = 10

Loss vs Weights, iterations = 100

Prediction vs Target, iterations = 100

Loss vs Weights, iterations = 1000

Prediction vs Target, iterations = 1000

```python
# Importing Libraries
import numpy as np
import matplotlib.pyplot as plt

iterations = 100
learning_rate = 0.0001
stopping_threshold = 1e-6

def mean_squared_error(y_true, y_predicted):

    # Calculating the loss (Mean squared error)
    cost = np.sum((y_true-y_predicted)**2) / len(y_true)
    return cost

def gradient_descent(x, y, iterations, learning_rate, stopping_threshold):

    # Initializing weight, bias, learning rate and iterations
    current_weight = 0.1
    current_bias = 0.01
    iterations = iterations
    learning_rate = learning_rate
    n = float(len(x))
    costs = []
    weights = []
    previous_cost = None
    # Estimation of optimal parameters
    for i in range(iterations):

        # Making predictions
        y_predicted = (current_weight * x) + current_bias

        # Calculating the current cost
        current_cost = mean_squared_error(y, y_predicted)

        # If the change in cost is less than or equal to
        # stopping_threshold we stop the gradient descent
        if previous_cost and abs(previous_cost-current_cost)<=stopping_threshold:
            break

        previous_cost = current_cost

        costs.append(current_cost)
        weights.append(current_weight)

        # Calculating the gradients
        weight_derivative = -(2/n) * sum(x * (y-y_predicted))
        bias_derivative = -(2/n) * sum(y-y_predicted)

        # Updating weights and bias
        current_weight = current_weight - (learning_rate * weight_derivative)
        current_bias = current_bias - (learning_rate * bias_derivative)
```

```python
            print(f"Iteration {i+1}: Cost {current_cost}, Weight \
            {current_weight}, Bias {current_bias}")


    # Visualizing the weights and cost at for all iterations
    plt.figure(figsize = (8,6))
    plt.plot(weights, costs)
    plt.scatter(weights, costs, marker='o', color='red')
    plt.title('Loss vs Weights')
    plt.ylabel('Loss')
    plt.xlabel('Weight')
    plt.show()

    return current_weight, current_bias

def main():

    # Data
    X = np.array([i for i in range(11)])
    Y = np.array([2*i for i in range(11)])

    # Estimating weight and bias using gradient descent
    estimated_weight, estimated_bias = gradient_descent(X, Y, iterations, learning_rate, stopping_threshold)
    print(f"Estimated Weight: {estimated_weight}\nEstimated Bias: {estimated_bias}")

    # Making predictions using estimated parameters
    Y_pred = estimated_weight*X + estimated_bias

    # Plotting the regression line
    plt.figure(figsize = (8,6))
    plt.scatter(X, Y, marker='o', color='red')
    plt.plot([min(X), max(X)], [min(Y_pred), max(Y_pred)], color='blue',markerfacecolor='red',
            markersize=10,linestyle='dashed')
    plt.title('Prediction vs Target')
    plt.xlabel('Prediction')
    plt.ylabel('Target')
    plt.show()
    print(X)
    print(Y)


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