Assignment:3

(Hill climbing algorithm)

Que:

Maximize the function f(x, y) = - (x^2 + y^2) + 10 * cos(2πx) + 10 * cos(2πy) over the range of x, y in [-5, 5].

(Hill climbing in 2 dimentions)

```
import random
import math

# Objective function to maximize
def objective_function(x, y):
    return -(x**2 + y**2) + 10 * math.cos(2 * math.pi * x) + 10 * math.cos(2 * math.pi
* y)

# Hill Climbing algorithm with random restarts
def hill_climbing_2d(step_size=0.1, iterations=1000, restarts=5):
    best_solution = None
    best_value = float('-inf')

    for restart in range(restarts):
        # Start with a random solution within the range [-5, 5]
        current_solution = [random.uniform(-5, 5), random.uniform(-5, 5)]
        current_value = objective_function(current_solution[0], current_solution[1])

        for i in range(iterations):
            # Generate neighboring solution by taking a small step in both x and y
directions
            new_solution = [
                current_solution[0] + random.uniform(-step_size, step_size),
                current_solution[1] + random.uniform(-step_size, step_size)
            ]

            # Ensure new solution is within bounds [-5, 5]
            new_solution[0] = max(-5, min(5, new_solution[0]))
            new_solution[1] = max(-5, min(5, new_solution[1]))

            # Evaluate the new solution
            new_value = objective_function(new_solution[0], new_solution[1])
```

```
            # If the new solution is better, move to the new solution
            if new_value > current_value:
                current_solution = new_solution
                current_value = new_value

            # Optionally, print progress for each iteration
            print(f"Restart {restart+1}, Iteration {i+1}: (x, y) = ({current_solution[0]:.4f}, {current_solution[1]:.4f}), f(x, y) = {current_value:.4f}")

        # Keep track of the best solution found across all restarts
        if current_value > best_value:
            best_solution = current_solution
            best_value = current_value

    return best_solution, best_value

# Run the Hill Climbing algorithm with random restarts
best_solution, best_value = hill_climbing_2d()

print(f"\nBest solution: (x, y) = ({best_solution[0]:.4f}, {best_solution[1]:.4f})")
print(f"Maximum value of the function: f(x, y) = {best_value:.4f}")
```

Assignment:4

(Genetic Algorithm)

Que: Solve classic knapsack problem using genetic algorithm

```
import random

# Define the items: (value, weight)
items = [(60, 10), (100, 20), (120, 30), (90, 40), (70, 50)]
weight_limit = 100  # The maximum weight that can be carried in the knapsack

# Generate a random individual (chromosome) where each gene is either 1 (include
item) or 0 (exclude item)
def generate_individual():
    return [random.randint(0, 1) for _ in range(len(items))]

# Generate an initial population of random individuals
def generate_population(pop_size):
    return [generate_individual() for _ in range(pop_size)]

# Calculate the fitness of an individual
def fitness(individual):
    total_value = 0
    total_weight = 0
    for i in range(len(individual)):
        if individual[i] == 1:  # If the item is included
            total_value += items[i][0]  # Add the value
            total_weight += items[i][1]  # Add the weight

    if total_weight > weight_limit:
        # Penalize individuals that exceed the weight limit
        return 0
    else:
        return total_value

# Roulette wheel selection: Select individuals based on their fitness proportionally
def roulette_wheel_selection(population, fitnesses):
    total_fitness = sum(fitnesses)
    pick = random.uniform(0, total_fitness)
    current = 0
    for i, individual in enumerate(population):
        current += fitnesses[i]
```

```
        if current > pick:
            return individual

# Single-point crossover between two parents
def crossover(parent1, parent2, crossover_rate=0.7):
    if random.random() < crossover_rate:
        point = random.randint(1, len(parent1) - 1)
        child1 = parent1[:point] + parent2[point:]
        child2 = parent2[:point] + parent1[point:]
        return child1, child2
    else:
        return parent1, parent2

# Mutate an individual by flipping a random bit
def mutate(individual, mutation_rate=0.1):
    for i in range(len(individual)):
        if random.random() < mutation_rate:
            individual[i] = 1 - individual[i]  # Flip the bit
    return individual

# Genetic Algorithm
def genetic_algorithm(pop_size=20, generations=100, crossover_rate=0.7,
mutation_rate=0.1):
    # Step 1: Initialize population
    population = generate_population(pop_size)

    for generation in range(generations):
        # Step 2: Evaluate fitness of all individuals
        fitnesses = [fitness(individual) for individual in population]

        # Step 3: Create a new population using selection, crossover, and mutation
        new_population = []

        while len(new_population) < pop_size:
            # Step 4: Selection
            parent1 = roulette_wheel_selection(population, fitnesses)
            parent2 = roulette_wheel_selection(population, fitnesses)

            # Step 5: Crossover
            child1, child2 = crossover(parent1, parent2, crossover_rate)

            # Step 6: Mutation
            child1 = mutate(child1, mutation_rate)
            child2 = mutate(child2, mutation_rate)
```

```
            new_population.extend([child1, child2])

        # Ensure population size remains constant
        population = new_population[:pop_size]

        # Optionally, print the best solution of each generation
        best_fitness = max(fitnesses)
        best_individual = population[fitnesses.index(best_fitness)]
        print(f"Generation {generation+1}: Best individual = {best_individual}, Fitness =
{best_fitness}")

    # Return the best solution from the final population
    best_fitness = max(fitnesses)
    best_individual = population[fitnesses.index(best_fitness)]
    return best_individual, best_fitness

# Run the Genetic Algorithm
best_solution, best_value = genetic_algorithm()

# Display the best solution
selected_items = [i for i, included in enumerate(best_solution) if included == 1]
print(f"\nBest solution: {best_solution}")
print(f"Items selected: {selected_items}")
print(f"Maximum value of the knapsack: {best_value}")
```

Assignment:5

(Neutral Network)

Que: Basic neutral network

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt

data = pd.read_csv('/kaggle/input/digit-recognizer/train.csv')
data = np.array(data)
m, n = data.shape
np.random.shuffle(data) # shuffle before splitting into dev and training sets

data_dev = data[0:1000].T
Y_dev = data_dev[0]
X_dev = data_dev[1:n]
X_dev = X_dev / 255.

data_train = data[1000:m].T
Y_train = data_train[0]
X_train = data_train[1:n]
X_train = X_train / 255.
_,m_train = X_train.shape
Y_train
```

Our NN will have a simple two-layer architecture. Input layer $a^{[0]}$ will have 784 units corresponding to the 784 pixels in each 28x28 input image. A hidden layer $a^{[1]}$ will have 10 units with ReLU activation, and finally our output layer $a^{[2]}$ will have 10 units corresponding to the ten digit classes with softmax activation.

Forward propagation:

$Z^{[1]} = W^{[1]} X + b^{[1]}$
$A^{[1]} = g_{\text{ReLU}}(Z^{[1]})¿$
$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$
$A^{[2]} = g_{\text{softmax}}(Z^{[2]})$

Backward propagation:

$dZ^{[2]} = A^{[2]} - Y$

$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$$

$$dB^{[2]} = \frac{1}{m} \Sigma dZ^{[2]}$$

$$dZ^{[1]} = W^{[2]T} dZ^{[2]} . \square g^{[1]\prime} (z^{[1]})$$

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} A^{[0]T}$$

$$dB^{[1]} = \frac{1}{m} \Sigma dZ^{[1]}$$

Parameter updates:

$$W^{[2]} := W^{[2]} - \alpha dW^{[2]}$$
$$b^{[2]} := b^{[2]} - \alpha db^{[2]}$$
$$W^{[1]} := W^{[1]} - \alpha dW^{[1]}$$
$$b^{[1]} := b^{[1]} - \alpha db^{[1]}$$

Vars and shapes:

Forward prop:

- $A^{[0]} = X$: 784 x m
- $Z^{[1]} \square A^{[1]}$: 10 x m
- $W^{[1]}$: 10 x 784 (as $W^{[1]} A^{[0]} \square Z^{[1]}$)
- $B^{[1]}$: 10 x 1
- $Z^{[2]} \square A^{[2]}$: 10 x m
- $W^{[1]}$: 10 x 10 (as $W^{[2]} A^{[1]} \square Z^{[2]}$)
- $B^{[2]}$: 10 x 1

Backprop:

- $dZ^{[2]}$: 10 x m ($A^{[2]}$)
- $dW^{[2]}$: 10 x 10
- $dB^{[2]}$: 10 x 1
- $dZ^{[1]}$: 10 x m ($A^{[1]}$)
- $dW^{[1]}$: 10 x 10
- $dB^{[1]}$: 10 x 1

```
def init_params():
    W1 = np.random.rand(10, 784) - 0.5
    b1 = np.random.rand(10, 1) - 0.5
    W2 = np.random.rand(10, 10) - 0.5
```

```
    b2 = np.random.rand(10, 1) - 0.5
    return W1, b1, W2, b2

def ReLU(Z):
    return np.maximum(Z, 0)

def softmax(Z):
    A = np.exp(Z) / sum(np.exp(Z))
    return A

def forward_prop(W1, b1, W2, b2, X):
    Z1 = W1.dot(X) + b1
    A1 = ReLU(Z1)
    Z2 = W2.dot(A1) + b2
    A2 = softmax(Z2)
    return Z1, A1, Z2, A2

def ReLU_deriv(Z):
    return Z > 0

def one_hot(Y):
    one_hot_Y = np.zeros((Y.size, Y.max() + 1))
    one_hot_Y[np.arange(Y.size), Y] = 1
    one_hot_Y = one_hot_Y.T
    return one_hot_Y

def backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y):
    one_hot_Y = one_hot(Y)
    dZ2 = A2 - one_hot_Y
    dW2 = 1 / m * dZ2.dot(A1.T)
    db2 = 1 / m * np.sum(dZ2)
    dZ1 = W2.T.dot(dZ2) * ReLU_deriv(Z1)
    dW1 = 1 / m * dZ1.dot(X.T)
    db1 = 1 / m * np.sum(dZ1)
    return dW1, db1, dW2, db2

def update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha):
    W1 = W1 - alpha * dW1
    b1 = b1 - alpha * db1
    W2 = W2 - alpha * dW2
    b2 = b2 - alpha * db2
    return W1, b1, W2, b2
def get_predictions(A2):
    return np.argmax(A2, 0)
```

```
def get_accuracy(predictions, Y):
    print(predictions, Y)
    return np.sum(predictions == Y) / Y.size

def gradient_descent(X, Y, alpha, iterations):
    W1, b1, W2, b2 = init_params()
    for i in range(iterations):
        Z1, A1, Z2, A2 = forward_prop(W1, b1, W2, b2, X)
        dW1, db1, dW2, db2 = backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y)
        W1, b1, W2, b2 = update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha)
        if i % 10 == 0:
            print("Iteration: ", i)
            predictions = get_predictions(A2)
            print(get_accuracy(predictions, Y))
    return W1, b1, W2, b2
W1, b1, W2, b2 = gradient_descent(X_train, Y_train, 0.10, 500)
```

~85% accuracy on training set.

```
def make_predictions(X, W1, b1, W2, b2):
    _, _, _, A2 = forward_prop(W1, b1, W2, b2, X)
    predictions = get_predictions(A2)
    return predictions

def test_prediction(index, W1, b1, W2, b2):
    current_image = X_train[:, index, None]
    prediction = make_predictions(X_train[:, index, None], W1, b1, W2, b2)
    label = Y_train[index]
    print("Prediction: ", prediction)
    print("Label: ", label)

    current_image = current_image.reshape((28, 28)) * 255
    plt.gray()
    plt.imshow(current_image, interpolation='nearest')
    plt.show()
```

Let's look at a couple of examples:

```
test_prediction(0, W1, b1, W2, b2)
test_prediction(1, W1, b1, W2, b2)
test_prediction(2, W1, b1, W2, b2)
test_prediction(3, W1, b1, W2, b2)
```

Finally, let's find the accuracy on the dev set:

```
dev_predictions = make_predictions(X_dev, W1, b1, W2, b2)
get_accuracy(dev_predictions, Y_dev)
```

Still 84% accuracy, so our model generalized from the training data pretty well.