**Functions Used:  
Function1: 8\*(x-3) ^4+4\*(y-1)^2 , Function2: Max(x-3,0) +4\*|y-1|**

1)

a)

The below given Python function gs\_random implements the Global Random Search algorithm. The gs\_random algorithm takes four inputs: a cost function to minimize, the dimensionality of the problem, a list of tuples specifying the minimum and maximum bounds for each parameter, and the number of random samples to generate and evaluate.

For each sample the algorithm generates a random vector within the specified bounds for each parameter. It does this by drawing each parameter's value uniformly at random between its specified minimum and maximum values. The cost function is then evaluated at this vector. The algorithm keeps track of the history of all generated parameter vectors, their associated costs, and the lowest cost observed so far. If the cost for a newly generated vector is lower than any observed previously, the vector is recorded as a potentially optimal solution. After repeating this process for the specified number of samples, the function returns the history of the lowest costs found, the cost for each sample, all sampled parameter vectors, and all parameter vectors that improved upon the lowest observed cost.

Code:

def gs\_random(evaluate\_cost, dimensions, limits, num\_samples):

    minimum\_cost = float('inf')

    history\_cost = []

    history\_minimum\_cost = []

    sequence\_history = []

    optimal\_sequence\_history = []

    for \_ in range(num\_samples):

        trial\_sequence = [uniform(limits[i][0], limits[i][1]) for i in range(dimensions)]

        evaluation = evaluate\_cost(\*trial\_sequence)

        sequence\_history.append(trial\_sequence)

        history\_cost.append(evaluation)

        if evaluation < minimum\_cost:

            minimum\_cost = evaluation

            optimal\_sequence\_history.append(trial\_sequence)

        history\_minimum\_cost.append(minimum\_cost)

    return history\_minimum\_cost, history\_cost, sequence\_history, optimal\_sequence\_history

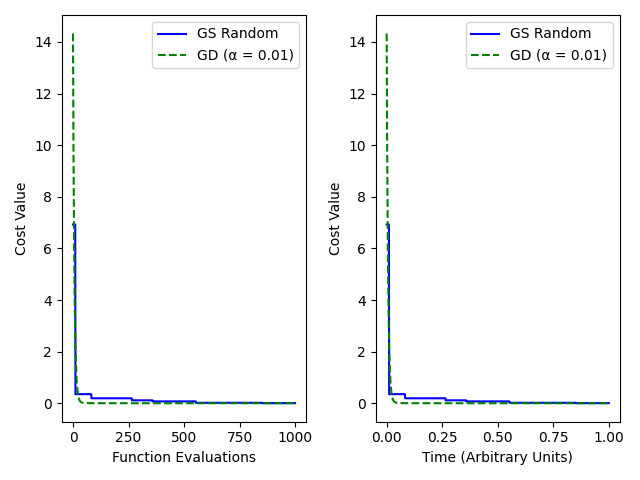
b)

Firstly, performed comparative analysis of the Global Random Search and Gradient Descent algorithms by applying them to two cost functions from week 4 mentioned at the start of the report. Then visualized their performance in terms of cost value over the number of function evaluations and over time.

Random search is implemented as explained in question 1 with alpha as 0.01. For Gradient Descent, it iteratively adjusts parameters in the direction that reduces the cost function, guided by the function's gradient. It proceeds in steps proportional to the negative of the gradient, with the step size controlled by a learning rate.

The evaluation is plotted in two ways for each cost function, first being function Evaluations vs. Cost Value and second being Time vs. Cost Value.

For Function 1 (Figure\_1):

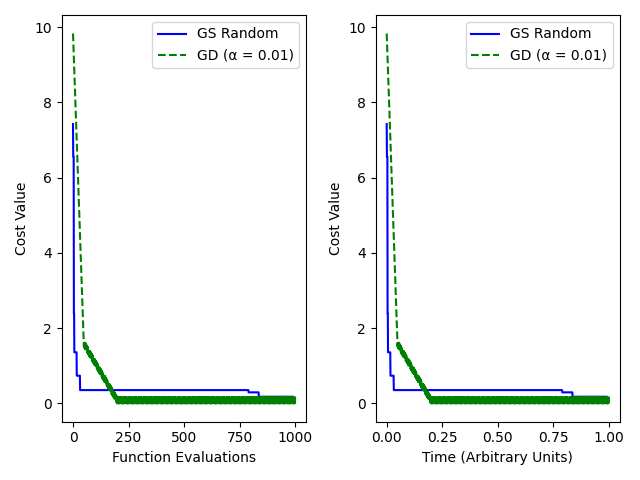


The first plot is function evaluations vs. cost value and from that we can see that both algorithms show a steep decline in the cost value at the beginning. This indicates that both methods quickly move towards areas of lower cost in the parameter space. The GS Random algorithm seems to achieve a significant decrease in cost with fewer function evaluations compared to GD. This suggests that, at least for this particular function, the randomness of GS Random finds good solutions quickly. After the initial rapid decrease, both lines flatten out, suggesting that both algorithms have reached a plateau near the minimum cost. Any further evaluations do not significantly improve the result.

The second plot is for Time vs. Cost Value and from that we can see that the GS Random and GD lines overlap closely. This means that despite the differences in the number of evaluations to reach a lower cost, the computational time to achieve these reductions is similar. The close overlap might indicate that while GS Random can find good solutions with fewer evaluations, each evaluation is faster for GD, possibly because the GS Random requires generating random numbers and evaluating the cost at these new points.

For Function 2 (Figure\_2):

For function two in case of Function Evaluations vs. Cost Value the trend is kind of similar to that observed for Function 1, where both algorithms initially reduce the cost quickly. As with Function 1, GS Random achieves a noticeable reduction in cost early on in the number of evaluations. This may suggest that for this function, GS Random is efficient in exploring the parameter space and finding regions of low cost. The cost reduction then levels off, showing that the algorithms have likely reached the vicinity of the minimum cost and are making incremental improvements. In case of GD we can see that it oscillated a lot back and forth before reaching minima.



In case of Time vs. Cost Value both GS Random and GD converge towards a low cost at a comparable rate. This suggests that while GS Random may appear faster in terms of evaluations, the time to compute these evaluations versus performing gradient calculations and taking a step in GD is on par. The similarity in time performance despite the different evaluation counts could imply that the overhead for function evaluations in GS Random is less than or equal to the overhead for gradient computations plus step calculations in GD.

2)

a)

The provided gs\_population function modifies the original Global Random Search algorithm by including a population-based approach. This modified algorithm first generates a set of random parameter vectors within given bounds. It evaluates a cost function for each vector and then selects a subset containing the best-performing vectors based on the lowest cost values. In following iterations, the algorithm focuses on the neighbourhood of each of these best-performing vectors, generating new sets of random vectors around them. It does this by perturbing the parameters of each best vector within a small range defined by epsilon. These perturbations ensure that the search is concentrated around the promising areas of the parameter space identified in previous steps. By repeatedly refining the population through selection of the best candidates and exploring their neighbourhoods, the algorithm iteratively converges towards the lowest-cost regions of the parameter space. The function takes the following inputs: a function to minimize, the number of dimensions for the parameters, the boundaries for each parameter, the number of random samples to generate initially, the number of best vectors to retain for neighbourhood exploration, the number of iterations to perform this process, and the range (epsilon) within which to explore around the best solutions. This method attempts to balance exploration of the parameter space with the exploitation of promising regions identified in the process. By using this approach, the algorithm can more effectively navigate complex landscapes with multiple local minima and potentially find a better approximation of the global minimum than a simple random search might.

Code:

def gs\_population(evaluate\_cost, dim, bounds, samples, keep\_best, iterations, epsilon):

    population = [np.array([uniform(bounds[i][0], bounds[i][1]) for i in range(dim)]) for \_ in range(samples)]

    cost\_history = []

    parameter\_history = []

    for \_ in range(iterations):

        costs = [(evaluate\_cost(\*candidate), candidate) for candidate in population]

        sorted\_population = sorted(costs, key=lambda x: x[0])

        best\_candidates = sorted\_population[:keep\_best]

        cost\_history.extend([cost for cost, \_ in best\_candidates])

        parameter\_history.extend([candidate.tolist() for \_, candidate in best\_candidates])

        new\_population = []

        for cost, best\_candidate in best\_candidates:

            for \_ in range(samples // keep\_best):

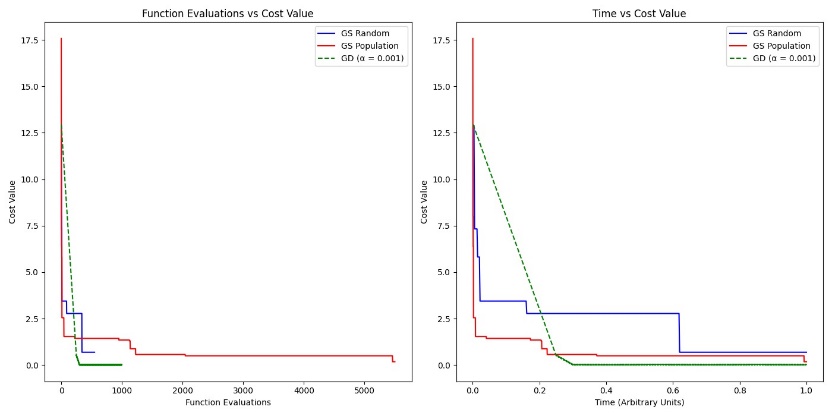
                neighbor = np.array([np.clip(uniform(best\_candidate[i] - epsilon, best\_candidate[i] + epsilon), bounds[i][0], bounds[i][1]) for i in range(dim)])

                new\_population.append(neighbor)

        population = new\_population

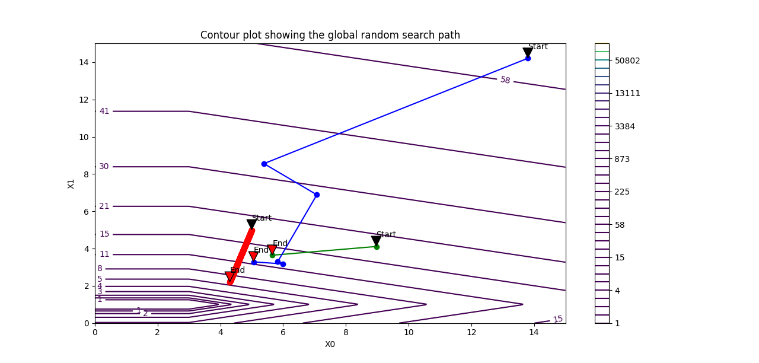
    return cost\_history, parameter\_history

b) Now using the new population-based approach we will compare performance for all three using plots and contour plots. In the line plot for Function 1, the x-axis represents the number of function evaluations for the optimization algorithms, and the y-axis represents the cost value obtained by each method. There are three different lines representing the Global Random Search (GS Random), the improved Global Population Search (GS Population), and Gradient Descent (GD). For GS Random it shows rapid initial progress, reducing the cost significantly, which then plateaus, suggesting that it has found a region of low cost. In case of GS Population, the line indicates the performance of the enhanced population-based search. The rapid decrease and subsequent plateauing at the same level as GS Random indicate it is effectively refining the search around promising areas, leading to quick convergence to low-cost solutions. The gradient descent line demonstrates a similar rapid decline in cost, closely following the performance of the population-based search.

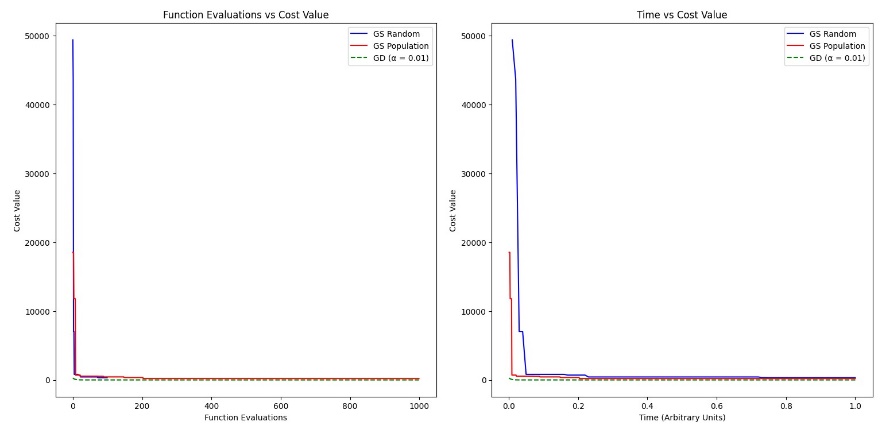


In the contour plot, the x and y axes represent the two parameters of the Function 1 cost, with contour lines indicating areas of equal cost. The paths taken by the optimization algorithms are superimposed:

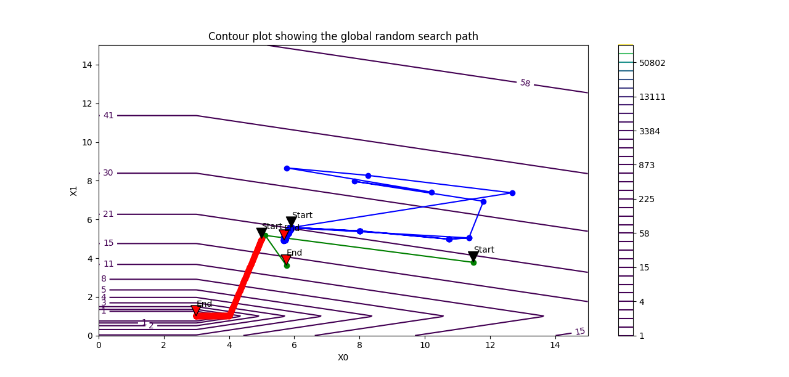
GS Random which is green line with 'o' markers represents the progression of parameter choices over time using GS Random. The 'Start' and 'End' annotations indicate the starting point and final point of the search, respectively. The path appears to quickly find a region of low cost, which is evidenced by the path entering a zone with tighter contour lines indicating lower cost and the annotations suggesting the end of the search is within a low-cost area. GS Population which blue line with 'o' markers shows the progression of the population-based search. The path's movement towards areas with denser contour lines implies that the search is zoning in on a minimum. The markers indicate steps in the algorithm where the population of points is refined to focus on the best-performing areas. GD which is red line with 'o' markers shows the gradient descent algorithm's journey through the parameter space. It moves decisively towards the low-cost area, with the start and end points suggesting an efficient progression towards the cost function's minimum.



For Function 2, we are again comparing three optimization methods: Global Random Search (GS Random), a modified Global Population Search (GS Population), and Gradient Descent (GD). In case of Function Evaluations vs. Cost Value all three methods quickly decrease the cost, with GD showing a rapid decline, followed by GS Population and GS Random. This indicates that all methods are effective at quickly identifying promising areas of the search space. The GS Random method shows a relatively more gradual descent compared to GS Population and GD, suggesting that the latter two methods are more efficient at exploiting the information they gather to zero in on the lower-cost regions. For Time vs. Cost Value the time efficiency appears to be best for the GD, which reaches a low cost value faster than the others when considering the arbitrary time units. This suggests that each iteration of GD, despite possibly requiring more computation per iteration due to gradient calculations, is more effective over time. GS Population also performs well, showing that its strategy of focusing on the best candidates and exploring their neighborhoods leads to efficient optimization over time. GS Random, while still finding a good solution, does so at a slightly slower pace compared to the other methods.



The contour plot for Function 2 shows the parameter space with lines representing areas of equal cost. The paths taken by the optimization algorithms are plotted on this landscape with GS Random i.e green line shows a more exploratory approach, with parameter choices distributed over a broader area before converging to a low-cost region. The spread of points suggests less efficiency in homing in on the optimum. GS Population which is the blue line path shows focused exploration around select candidates, which are iteratively refined. The narrowing of the path towards areas of lower cost indicates the algorithm's effectiveness in localizing and exploiting promising regions. Lastly, the GD path shows a direct and targeted progression towards the low-cost area. This efficiency is due to the use of gradient information to guide the search more precisely towards the minimum.

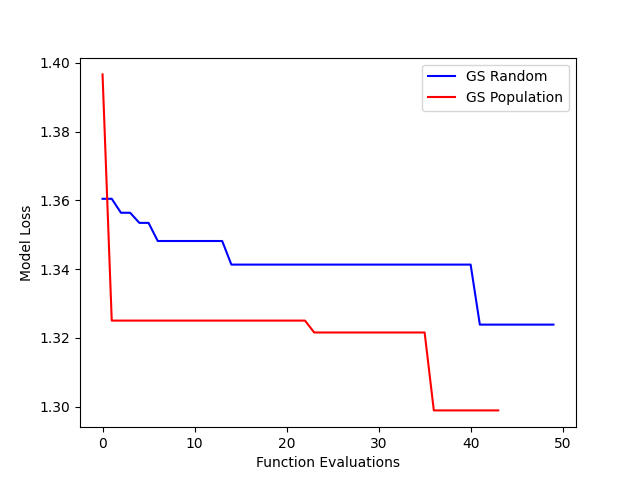


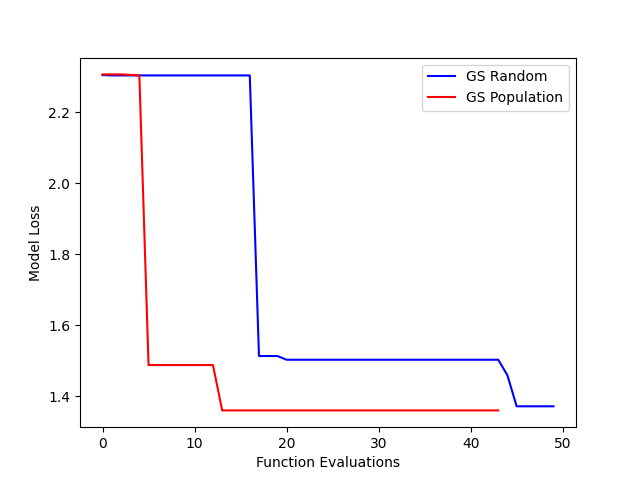
In summary, for Function 2, the contour plot reveals the efficiency of the GD and GS Population methods in navigating the cost landscape. GD uses gradient information to take the most direct route to low-cost regions, while GS Population leverages the best-found solutions to generate new, more promising candidates. GS Random lacks these sophisticated strategies but eventually finds the low-cost area through extensive search.

3) Downloading the function applied that to Global Random search and Population search from the last two problem. The parameter chosen were N=50 , M=5 and no of iteration 3 and minibatch size, adam parameters α, β1, β2 and no of epochs as hyperparameter.

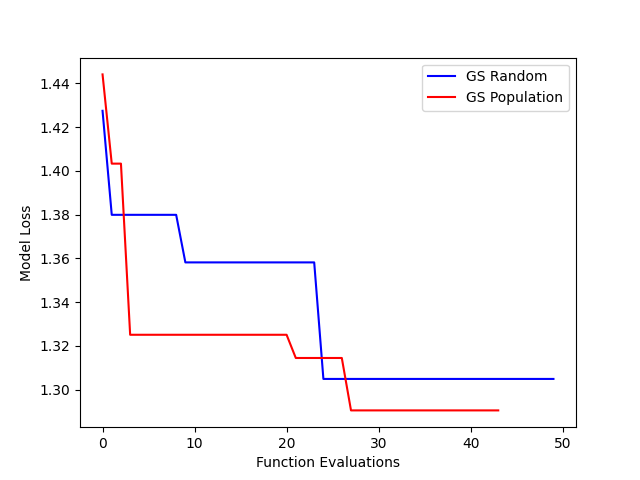
a) The search range for the mini-batch size was between 0 to 130 with fixed Adam parameters (α=0.01, β1=0.9, β2=0.99) and a constant epoch count of 20. The plots shows that the GS Population algorithm consistently outperformed the GS Random approach, which suggests that a population-based method can be more effective in narrowing down the most beneficial mini-batch sizes. The best mini-batch size for GS Random and GS Population were found to be approximately 38 and 20, respectively.





b) For the second set of hyperparameters, we held the mini-batch size constant at 38 which we found from part a, while varying the Adam parameters: α from 0.0001 to 0.1, β1 from 0.25 to 0.99, and β2 from 0.9 to 0.9999. The number of epochs remained at 20. The resulting plot indicates a trend similar to the first, with the GS Population method achieving a lower model loss compared to the GS Random, reinforcing the effectiveness of the population-based approach for this hyperparameter set.

c) For this part, the mini-batch size was 38 that we got from part a, and Adam parameters (α at 0.1, β1 at 0.9, and β2 at 0.999)—our comparison between Global Random Search (GS Random) and Global Population Search (GS Population) highlights the latter's effectiveness. The provided plot shows that GS Population not only outperforms GS Random in terms of lower model loss but also finds a more efficient training duration, with the best number of epochs being approximately 11 for GS Population compared to about 23 for GS Random. This suggests a significant reduction in training time without sacrificing model performance, emphasizing the advantage of using a population-based approach for this aspect of hyperparameter optimization.



Appendix:  
from random import uniform

import matplotlib.pyplot as plt

import numpy as np

def cost\_function\_one(x, y):

    return 8 \* (x - 3) \*\* 4 + 4 \* (y - 1) \*\* 2

def cost\_function\_two(x, y):

    return np.maximum(x - 3, 0) + 4 \* np.abs(y - 1)

def gradient\_one(x, y):

    return np.array([32 \* (x - 3)\*\*3, 8 \* (y - 1)])

def gradient\_two(x, y):

    grad\_x = np.where(x > 3, 1, 0)

    grad\_y = 4 \* np.sign(y - 1)

    return np.array([grad\_x, grad\_y])

# Random Search

def gs\_random(evaluate\_cost, dimensions, limits, num\_samples):

    minimum\_cost = float('inf')

    history\_cost = []

    history\_minimum\_cost = []

    sequence\_history = []

    optimal\_sequence\_history = []

    for \_ in range(num\_samples):

        trial\_sequence = [uniform(limits[i][0], limits[i][1]) for i in range(dimensions)]

        evaluation = evaluate\_cost(\*trial\_sequence)

        sequence\_history.append(trial\_sequence)

        history\_cost.append(evaluation)

        if evaluation < minimum\_cost:

            minimum\_cost = evaluation

            optimal\_sequence\_history.append(trial\_sequence)

        history\_minimum\_cost.append(minimum\_cost)

    return history\_minimum\_cost, history\_cost, sequence\_history, optimal\_sequence\_history

# Gradient decent

def gd(evaluate\_cost, compute\_gradient, initial\_position, learning\_rate, steps):

    position = np.array(initial\_position, dtype=float)

    cost\_record = []

    position\_history = [position.copy()]

    for \_ in range(steps):

        gradient = compute\_gradient(\*position)

        position -= learning\_rate \* gradient

        cost\_record.append(evaluate\_cost(\*position))

        position\_history.append(position.copy())

    return cost\_record, position\_history

# Global population

def gs\_population(evaluate\_cost, dim, bounds, samples, keep\_best, iterations, epsilon):

    population = [np.array([uniform(bounds[i][0], bounds[i][1]) for i in range(dim)]) for \_ in range(samples)]

    cost\_history = []

    parameter\_history = []

    for \_ in range(iterations):

        costs = [(evaluate\_cost(\*candidate), candidate) for candidate in population]

        sorted\_population = sorted(costs, key=lambda x: x[0])

        best\_candidates = sorted\_population[:keep\_best]

        cost\_history.extend([cost for cost, \_ in best\_candidates])

        parameter\_history.extend([candidate.tolist() for \_, candidate in best\_candidates])

        new\_population = []

        for cost, best\_candidate in best\_candidates:

            for \_ in range(samples // keep\_best):

                neighbor = np.array([np.clip(uniform(best\_candidate[i] - epsilon, best\_candidate[i] + epsilon), bounds[i][0], bounds[i][1]) for i in range(dim)])

                new\_population.append(neighbor)

        population = new\_population

    return cost\_history, parameter\_history

# Stepwise plot

def analyze\_and\_visualize\_stepwise(selected\_function, search\_count, gradient\_steps, iterative\_steps):

    function\_list = [cost\_function\_one, cost\_function\_two]

    gradient\_list = [gradient\_one, gradient\_two]

    range\_limits = [(0, 15), (0, 15)]

    stepwise\_random\_history, random\_evaluation\_record, random\_solution\_history, optimal\_random\_solutions = gs\_random(

        function\_list[selected\_function], 2, range\_limits, search\_count

    )

    stepwise\_iterative\_history, iterative\_solution\_history = gs\_population(

        function\_list[selected\_function], 2, range\_limits, search\_count, keep\_best=10, iterations=iterative\_steps, epsilon=0.1

    )

    gradient\_descent\_history, descent\_solution\_history = gd(

        function\_list[selected\_function], gradient\_list[selected\_function], [3,3], 0.01, gradient\_steps

    )

    plt.figure(figsize=(14, 7))

    plt.subplot(1, 2, 1)

    plt.plot(stepwise\_random\_history, 'b-', label='GS Random')

    plt.plot(stepwise\_iterative\_history, 'r-', label='GS Population')

    plt.plot(gradient\_descent\_history, 'g--', label='GD (α = 0.01)')

    plt.xlabel('Function Evaluations')

    plt.ylabel('Cost Value')

    plt.title('Function Evaluations vs Cost Value')

    plt.legend()

    plt.subplot(1, 2, 2)

    plt.plot(np.cumsum(np.full\_like(stepwise\_random\_history, 1/search\_count)), stepwise\_random\_history, 'b-', label='GS Random')

    plt.plot(np.cumsum(np.full\_like(stepwise\_iterative\_history, 1/(search\_count\*iterative\_steps))), stepwise\_iterative\_history, 'r-', label='GS Population')

    plt.plot(np.cumsum(np.full\_like(gradient\_descent\_history, 1/gradient\_steps)), gradient\_descent\_history, 'g--', label='GD (α = 0.01)')

    plt.xlabel('Time (Arbitrary Units)')

    plt.ylabel('Cost Value')

    plt.title('Time vs Cost Value')

    plt.legend()

    plt.tight\_layout()

    plt.show()

# Contour plot

def analyze\_and\_visualize\_contour(evaluate\_func, calculate\_gradient, sample\_volume, descent\_repetitions, population\_cycles):

    constrained\_bounds = [(5, 15), (3, 15)]

    x = np.linspace(0, 15, 400)

    y = np.linspace(0, 15, 400)

    X, Y = np.meshgrid(x, y)

    Z = evaluate\_func(X, Y)

    plt.figure(figsize=(10, 8))

    contour\_plot = plt.contour(X, Y, Z, levels=np.logspace(0, 5, 35))

    plt.clabel(contour\_plot, inline=1, fontsize=10)

    plt.colorbar(contour\_plot)

    random\_search\_history, \_, random\_solution\_paths, optimal\_random\_solutions = gs\_random(

        evaluate\_func, 2, [(0, 15), (0, 15)], sample\_volume

    )

    iterative\_search\_history, iterative\_solution\_paths = gs\_population(

        evaluate\_func, 2, constrained\_bounds, sample\_volume, keep\_best=10, iterations=population\_cycles, epsilon=0.1

    )

    descent\_evaluation\_history, descent\_path\_history = gd(

        evaluate\_func, calculate\_gradient, [5, 5], 0.01, descent\_repetitions

    )

    # Plotting paths

    x\_vals\_r, y\_vals\_r = zip(\*optimal\_random\_solutions)

    plt.plot(x\_vals\_r, y\_vals\_r, 'o-', color='green', label='Global Random Search')

    x\_vals\_p, y\_vals\_p = zip(\*iterative\_solution\_paths)

    plt.plot(x\_vals\_p, y\_vals\_p, 'o-', color='blue', label='Global Population Search Stepwise')

    x\_vals\_g, y\_vals\_g = zip(\*descent\_path\_history)

    plt.plot(x\_vals\_g, y\_vals\_g, 'o-', color='red', label='Gradient Descent')

    # start and end points

    plt.annotate('Start', xy=(x\_vals\_r[0], y\_vals\_r[0]), xytext=(x\_vals\_r[0], y\_vals\_r[0]+1),

                 arrowprops=dict(facecolor='black', shrink=0.05))

    plt.annotate('End', xy=(x\_vals\_r[-1], y\_vals\_r[-1]), xytext=(x\_vals\_r[-1], y\_vals\_r[-1]+1),

                 arrowprops=dict(facecolor='red', shrink=0.05))

    plt.annotate('Start', xy=(x\_vals\_p[0], y\_vals\_p[0]), xytext=(x\_vals\_p[0], y\_vals\_p[0]+1),

                 arrowprops=dict(facecolor='black', shrink=0.05))

    plt.annotate('End', xy=(x\_vals\_p[-1], y\_vals\_p[-1]), xytext=(x\_vals\_p[-1], y\_vals\_p[-1]+1),

                 arrowprops=dict(facecolor='red', shrink=0.05))

    plt.annotate('Start', xy=(x\_vals\_g[0], y\_vals\_g[0]), xytext=(x\_vals\_g[0], y\_vals\_g[0]+1),

                 arrowprops=dict(facecolor='black', shrink=0.05))

    plt.annotate('End', xy=(x\_vals\_g[-1], y\_vals\_g[-1]), xytext=(x\_vals\_g[-1], y\_vals\_g[-1]+1),

                 arrowprops=dict(facecolor='red', shrink=0.05))

    plt.legend()

    plt.xlabel('X0')

    plt.ylabel('X1')

    plt.title('Contour plot showing the optimization paths')

    plt.show()

# Comparison for function 1

analyze\_and\_visualize\_stepwise(0, 1000, 1000, 50)

analyze\_and\_visualize\_contour(cost\_function\_one, gradient\_one, 30, 300, 10)

# Comparison for function 2

analyze\_and\_visualize\_stepwise(1, 550, 1000, 10)

analyze\_and\_visualize\_contour(cost\_function\_two, gradient\_two, 30, 700, 10)

import matplotlib.pyplot as plt

import numpy as np

import tensorflow as tf

from random import uniform

from tensorflow import keras

from keras import layers, regularizers

from keras.optimizers import Adam

from keras.losses import CategoricalCrossentropy

def loss(batch\_size, alpha, beta1, beta2, epochs):

    batch\_size = int(batch\_size)

    epochs = int(epochs)

    num\_classes = 10

    input\_shape = (32, 32, 3)

    (x\_train, y\_train), (x\_test, y\_test) = keras.datasets.cifar10.load\_data()

    n = 5000

    x\_train = x\_train[:n]; y\_train = y\_train[:n]

    x\_train = x\_train.astype("float32") / 255

    x\_test = x\_test.astype("float32") / 255

    y\_train = keras.utils.to\_categorical(y\_train, num\_classes)

    y\_test = keras.utils.to\_categorical(y\_test, num\_classes)

    model = keras.Sequential([

        layers.Conv2D(16, (3, 3), padding='same', input\_shape=input\_shape, activation='relu'),

        layers.Conv2D(16, (3, 3), strides=(2, 2), padding='same', activation='relu'),

        layers.Conv2D(32, (3, 3), padding='same', activation='relu'),

        layers.Conv2D(32, (3, 3), strides=(2, 2), padding='same', activation='relu'),

        layers.Dropout(0.5),

        layers.Flatten(),

        layers.Dense(num\_classes, activation='softmax', kernel\_regularizer=regularizers.l1(0.0001))

    ])

    optimizer = Adam(learning\_rate=alpha, beta\_1=beta1, beta\_2=beta2)

    model.compile(loss="categorical\_crossentropy", optimizer=optimizer, metrics=["accuracy"])

    model.fit(x\_train, y\_train, batch\_size=batch\_size, epochs=epochs, validation\_split=0.1, verbose=0)

    y\_preds = model.predict(x\_test)

    loss\_fn = CategoricalCrossentropy()

    return loss\_fn(y\_test, y\_preds).numpy()

def gs\_random(evaluate\_cost, dimensions, limits, num\_samples):

    minimum\_cost = float('inf')

    history\_cost = []

    history\_minimum\_cost = []

    sequence\_history = []

    optimal\_sequence\_history = []

    for \_ in range(num\_samples):

        trial\_sequence = [uniform(limits[i][0], limits[i][1]) for i in range(dimensions)]

        evaluation = evaluate\_cost(\*trial\_sequence)

        sequence\_history.append(trial\_sequence)

        history\_cost.append(evaluation)

        if evaluation < minimum\_cost:

            minimum\_cost = evaluation

            optimal\_sequence\_history.append(trial\_sequence)

        history\_minimum\_cost.append(minimum\_cost)

    return history\_minimum\_cost, history\_cost, sequence\_history, optimal\_sequence\_history

def gs\_population(evaluate\_cost, dim, bounds, samples, keep\_best, iterations, epsilon):

    population = [np.array([uniform(bounds[i][0], bounds[i][1]) for i in range(dim)]) for \_ in range(samples)]

    cost\_history = []

    parameter\_history = []

    for \_ in range(iterations):

        costs = [(evaluate\_cost(\*candidate), candidate) for candidate in population]

        sorted\_population = sorted(costs, key=lambda x: x[0])

        best\_candidates = sorted\_population[:keep\_best]

        cost\_history.extend([cost for cost, \_ in best\_candidates])

        parameter\_history.extend([candidate.tolist() for \_, candidate in best\_candidates])

        new\_population = []

        for \_, best\_candidate in best\_candidates:

            for \_ in range(samples // keep\_best):

                neighbor = np.array([np.clip(best\_candidate[i] + uniform(-epsilon, epsilon), bounds[i][0], bounds[i][1]) for i in range(dim)])

                new\_population.append(neighbor)

        population = new\_population

    return cost\_history, parameter\_history

n = 5

range\_minibatch = [[1, 130], [0.01, 0.1], [0.9, 0.99], [0.999, 0.9999], [20, 20]]

gs\_cost\_minibatch, \_, \_, gs\_param\_minibatch = gs\_random(loss, n, range\_minibatch, 50)

gp\_cost\_minibatch, gp\_param\_minibatch = gs\_population(loss, n, range\_minibatch, 12, 4, 4, 0.1)

# Param optimization

range\_param = [[38, 38], [0.0001, 0.1], [0.25, 0.99], [0.9, 0.9999], [20, 20]]

gs\_cost\_param, \_, \_, gs\_param\_param = gs\_random(loss, n, range\_param, 50)

gp\_cost\_param, gp\_param\_param = gs\_population(loss, n, range\_param, 12, 4, 4, 0.1)

# Epochs optimization

range\_epochs = [[38, 38], [0.01, 0.01], [0.9, 0.9], [0.999, 0.999], [10, 30]]

gs\_cost\_epochs, \_, \_, gs\_param\_epochs = gs\_random(loss, n, range\_epochs, 50)

gp\_cost\_epochs, gp\_param\_epochs = gs\_population(loss, n, range\_epochs, 12, 4, 4, 0.1)

# Output best batch sizes from minibatch optimization

best\_batch\_size\_r = gs\_param\_minibatch[0]

best\_batch\_size\_p = gp\_param\_minibatch[0]

print('Best batch size Random = ', best\_batch\_size\_r)

print("Best batch size Population =", best\_batch\_size\_p)

best\_epoch\_r = gs\_param\_epochs[0]

best\_epoch\_p = gp\_param\_epochs[0]

print('Best epochs Random = ',best\_epoch\_r)

print("Best epochs Population =", best\_epoch\_p)

# Function to plot results, now inline

def plot\_results(gs\_param, gs\_cost, gp\_param, gp\_cost):

    plt.plot(range(len(gs\_cost)), gs\_cost, label='GS Random')

    plt.plot(range(len(gp\_cost)), gp\_cost, label='GS Population')

    plt.xlabel('Iterations')

    plt.ylabel('Loss')

    plt.legend()

    plt.show()

# Plotting results for all optimizations

plot\_results(gs\_param\_minibatch, gs\_cost\_minibatch, gp\_param\_minibatch, gp\_cost\_minibatch)

plot\_results(gs\_param\_param, gs\_cost\_param, gp\_param\_param, gp\_cost\_param)

plot\_results(gs\_param\_epochs, gs\_cost\_epochs, gp\_param\_epochs, gp\_cost\_epochs)