CS7DS2: Week 8 Assignment

Conor McCauley - 17323203

April 4, 2022

Functions

The two functions that will be used during this assignment are:

$$f_1(x_0, x_1) = 9(x_0 - 5)^4 + 10(x_1 - 2)^2$$
$$f_2(x_0, x_1) = \max(x_0 - 5, 0) + 10 \cdot |x_1 - 2|$$

Their partial derivatives, as calculated in the week 2 assignment, are:

$$\frac{\partial f_1}{\partial x_0} = 36(x_0 - 5)^3, \frac{\partial f_1}{\partial x_1} = 20x_1 - 40$$
$$\frac{\partial f_2}{\partial x_0} = \theta(x_0 - 5), \frac{\partial f_2}{\partial x_1} = 10 \cdot sign(x_1 - 2)$$

Question (a)

(i) The following snippet of code implements the global random search (GRS) algorithm. It begins by initialising the best function value found thus far to a large value. It then, for N iterations, generates a random parameter vector x with values of x_i in the range $[l_i, u_i]$. At each iteration it tracks the best function value found and its associated parameter vector.

```
best_x = None
best_f = 1e20
for _ in range(N):
    this_x = [uniform(l[i], u[i]) for i in range(n)]
    this_f = f(*this_x)
    if this_f < best_f:
        best_x = deepcopy(this_x)
        best_f = this_f
```

(ii) For comparison, the gradient descent (GD) algorithm implemented in the week 2 assignment was used with $\alpha = 0.01$. Running the GRS algorithm for N iterations results in N main function calls while running the GRS algorithm for N iterations requires N main function calls and $2N$ derivative function calls. The `timeit` module was used to test the average running times of both algorithms when $N = 200$. These running times were divided by the number of main function calls made to find their average times per evaluation. In order to fairly compare their results both the value of the function after each evaluation and the value of the function after each unit of time were plotted. The results for f_1 and f_2 can be seen in figures 1 and 2, respectively.

For f_1, the GD algorithm manages to converge on the minimum in fewer than 50 iterations while the GSR algorithm, despite finding a fairly small value, fails to reach the same minimum. This is due to the low probability of choosing appropriate values for x by random chance. Both algorithms required a similar amount of time to run so the second plot is irrelevant for this example.

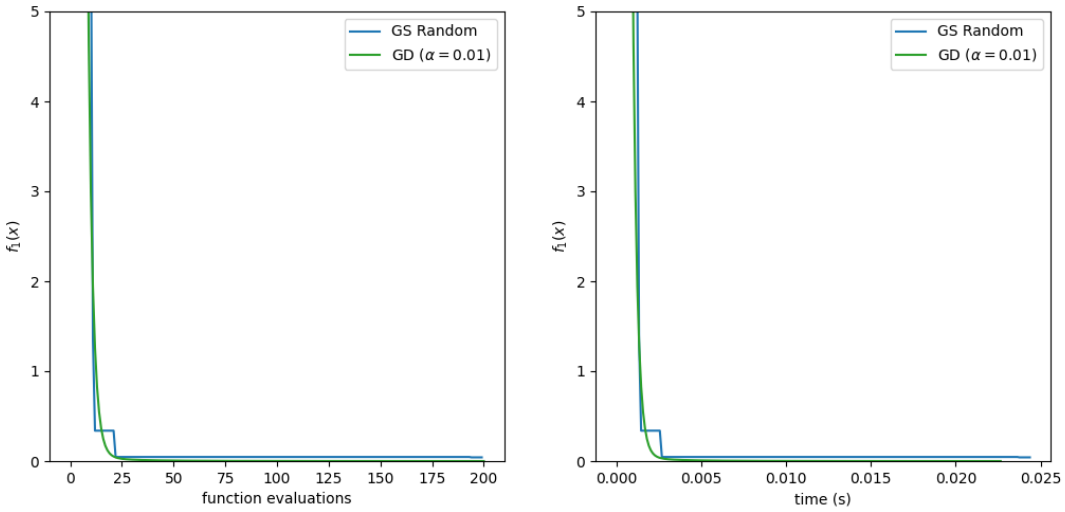

Figure 1: Change in $f_1(x)$ over evaluations and time (GD and GRS)

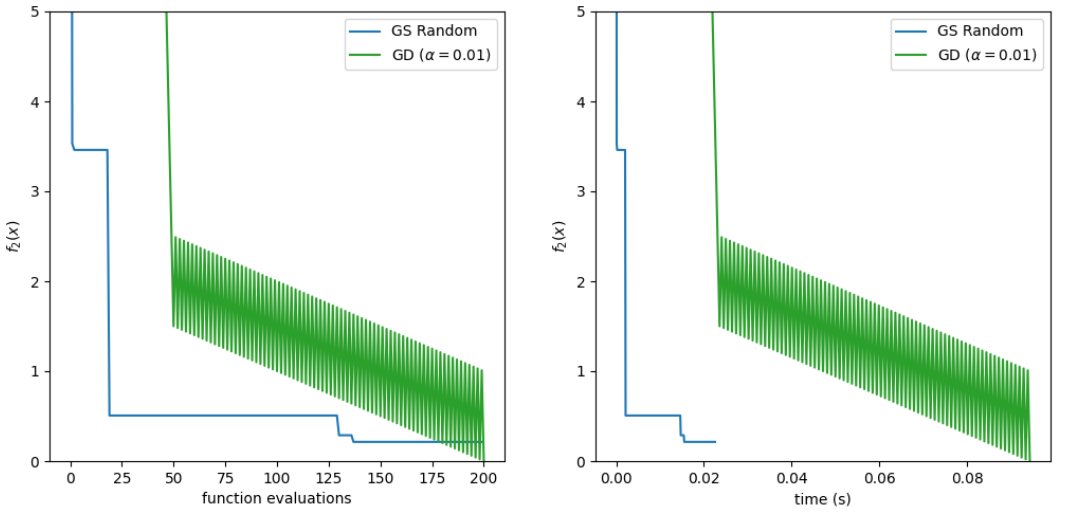

Figure 2: Change in $f_2(x)$ over evaluations and time (GD and GRS)

For f_2, the GD algorithm fails to converge and instead oscillates back and forth toward the minimum due to α being too large. The GSR algorithm performs fairly well and reaches a small value without being entirely beaten by the GD algorithm. From the second plot it is clear that the GSR algorithm runs much faster for this function than the GD algorithm does. This is most likely due to the derivative function calls (made by the GD algorithm) requiring subsequent calls to two NumPy functions, `heaviside()` and `sign()`.

Question (b)

(i) The following snippet of code implements the global population search (GPS) algorithm. It begins by generating a list of N random parameter vectors x and their corresponding function values. This process is similar to the one carried out in the GRS algorithm.

```
this_xs = [
    [uniform(l[i], u[i]) for i in range(n)]
    for _ in range(N)
]
this_fs = [f(*x) for x in this_xs]
```

Then, for a certain number of iterations it repeats the following process: select the best

M results from the list of N vectors, for each of these M best results, m, replace $\frac{N-M}{M}$ of the bottom $N-M$ results with a randomly perturbed version of m. The perturbation process involves multiplying each parameter in m by a random value in the range $(0.8, 1.2)$.

```
num_new = (N - M) // M
for _ in range(num_iters):
    this_fs, this_xs = sort_lists(this_fs, this_xs)
    for i in range(M):
        this_x = this_xs[i]
        for j in range(num_new):
            pert_x = [x * uniform(0.8, 1.2) for x in this_x]
            k = M + (i * num_new) + j
            this_xs[k] = deepcopy(pert_x)
            this_fs[k] = f(*pert_x)
this_fs, this_xs = sort_lists(this_fs, this_xs)
```

(ii) For fairness, as in (a), the `timeit` library was used to test the average running time of the GPS algorithm when $N = 20, M = 5, num\_iters = 10$ which amounts to $N + num\_iters \cdot (N - M)$ function evaluations. The value of the function after each evaluation and the value of the function after each unit of time were plotted for the GPS, GRS and GD algorithms. These values for f_1 and f_2 can be seen in figures 3 and 5, respectively. The change in x for each algorithm was also plotted for each function and can been seen in figures 4 and 6.

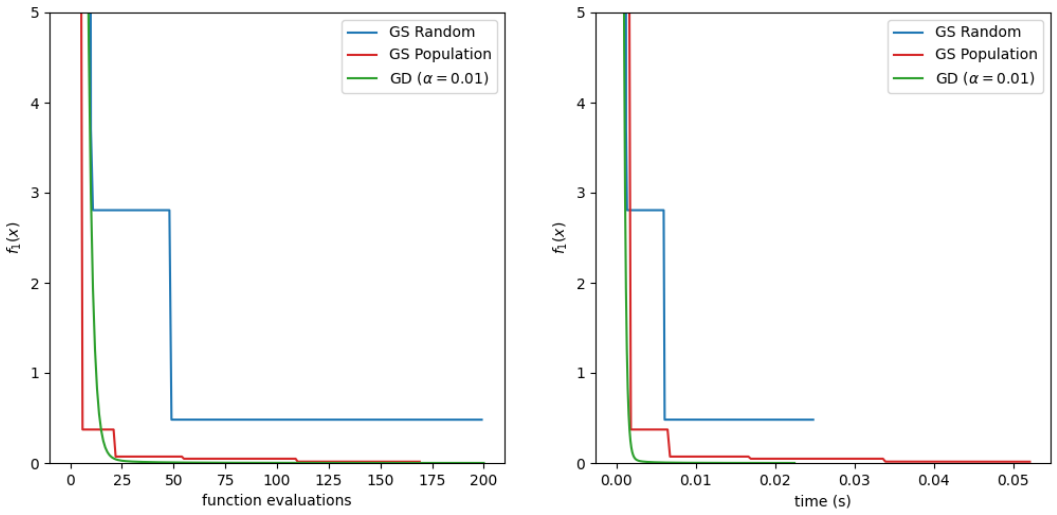

Figure 3: Change in $f_1(x)$ over evaluations and time (GD, GRS and GPS)

For f_1, the GPS algorithm produces much better results than GRS with GD only slightly beating it. However, the GPS algorithm takes almost twice as long as the other two to run per evaluation.

From the contour plot it is clear that, in contrast to the smooth descent of GD, the GPS algorithm descends in a more jagged manner with each new best value being in the same neighbourhood as the previous one.

For f_2, the GPS algorithm produces better results than both the GRS and GD algorithms. Like in (a), the GD algorithm oscillates back and forth toward the minimum and never converges while the GRS algorithm fails to find a satisfactory minimum. The GPS algorithm also ran twice as fast as GD making it an even more optimal choice.

From the contour plot it is clear that the GD algorithm is oscillating back and forth across the minimum ridge while the GPS algorithm finds a minimal point almost immediately.

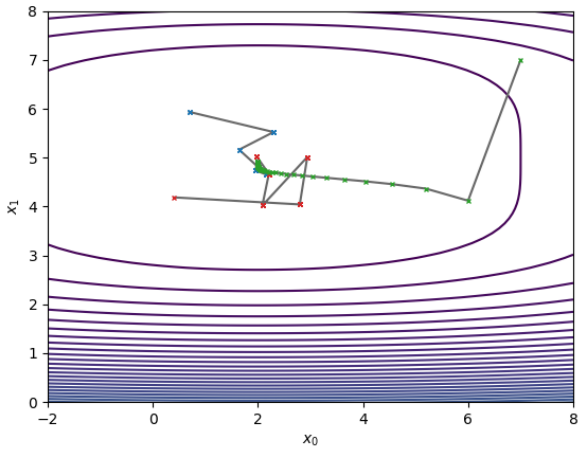

Figure 4: Change in x over evaluations (GD, GRS and GPS)

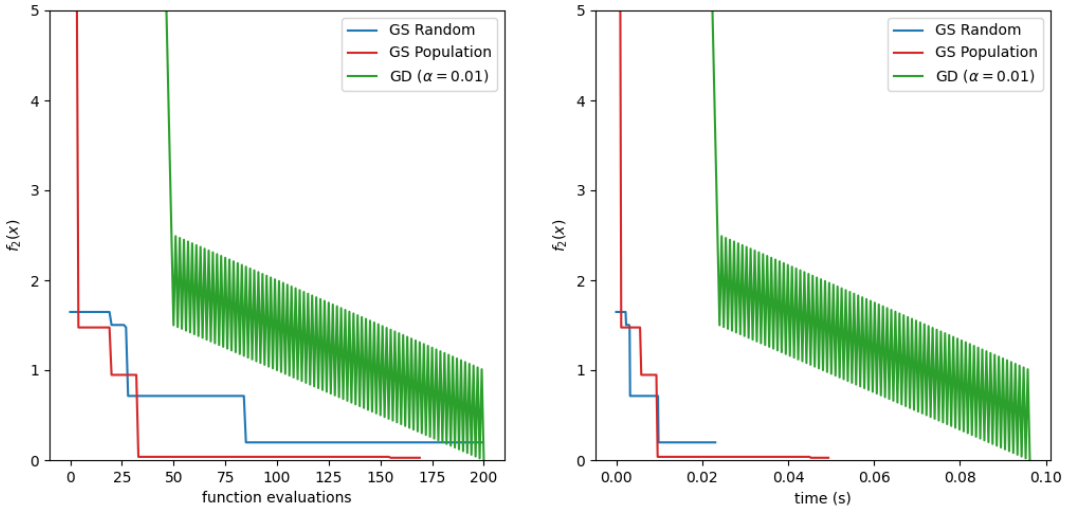

Figure 5: Change in $f_2(x)$ over evaluations and time (GD, GRS and GPS)

Question (c)

The model compiling and training was moved into its own function which takes, as parameters, the mini-batch size, α, β_1, β_2 and the number of epochs. It returns the categorical cross-entropy loss of the model. The function can then be passed to the aforementioned GRS and GPS functions to search for optimal hyperparameters. In each case the GPS algorithm was run for 50 iterations while the GPS algorithm was run for 44 iterations ($N = 12, M = 4, num\_iters = 4$).
(i) A range of mini-batch sizes were searched for in the range $[1, 128]$ while the number of epochs was held constant at 20 and the Adam parameters were kept constant at $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$.

The GPS algorithm performs a good deal better than the GRS algorithm however this does not mean that the GRS algorithm performed poorly. The good performance was most likely due to the parameter search being limited to discrete values as opposed to an effectively infinite continuous range of values. The optimal batch size was found to be 45.
(ii) A range of Adam parameters were searched for in the ranges $\alpha \in [0.0001, 0.1], \beta_1 \in [0.25, 0.99], \beta_2 \in [0.9, 0.9999]$ while the number of epochs was held constant at 20 and the mini-batch size was set to 45 based on the results of (i).

The GRS algorithm performed slightly better than the GPS algorithm when searching for Adam parameters although neither algorithm produced better results than the default value.

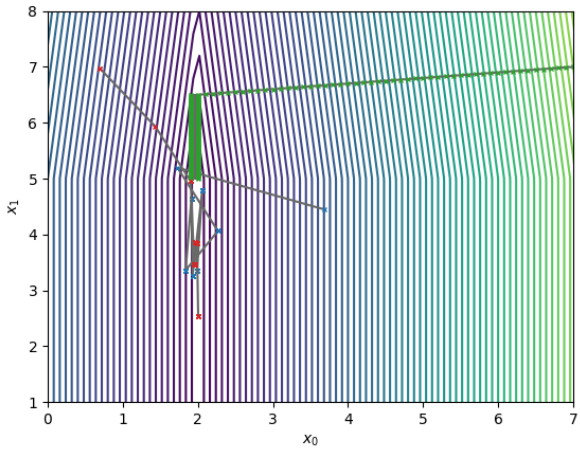

Figure 6: Change in x over evaluations (GD, GRS and GPS)

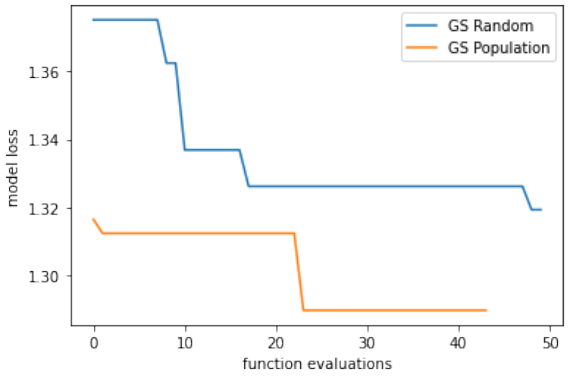

Figure 7: Change in model loss over iterations (batch size search)

The reason for this is most likely due to the parameters being searched for in a uniform way (i.e. random α values closer to 0.0001 are much less likely to appear than values closer to 0.1) this could possibly be generating random values in a logarithmic way.

(iii) A range of epochs was searched for in the range $[5, 30]$ while the mini-batch size was set to 45 based on the results of (i) and the Adam parameters were kept as the defaults $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$ based on the poor results of (ii).

The GRS algorithm actually performs slightly better than GPS with the former settling on 18 epochs and the latter on 21 epochs. However, neither managed to find the default value of 20 which actually produced a better result.

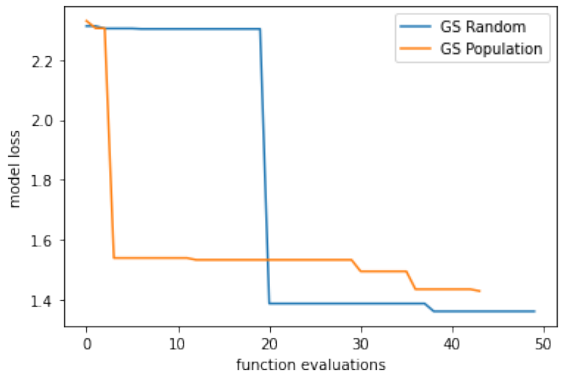

Figure 8: Change in model loss over iterations (Adam parameters search)

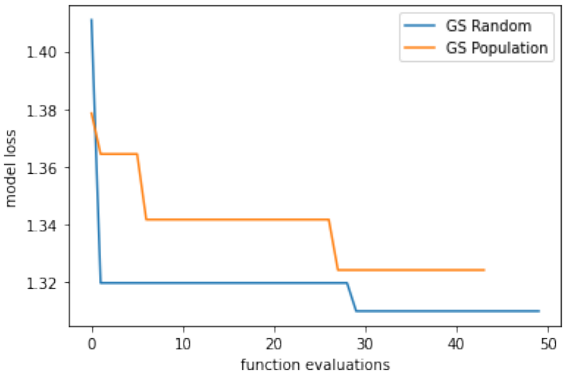

Figure 9: Change in model loss over iterations (epochs search)

Appendix A: Code

Code for (a) and (b)

```
from copy import deepcopy
from random import uniform
from timeit import timeit
import matplotlib.pyplot as plt
import numpy as np

def get_function(func_num):
    if func_num == 1:
        f = lambda x, y: 9*(x - 5)**4 + 10*(y - 2)**2
        dfx = lambda x: 36*(x - 5)**3
        dfy = lambda y: 20*y - 40
    else:
        f = lambda x, y: 10*abs(y - 2) + max(0, x - 5)
        dfx = lambda x: np.heaviside(x - 5, 0)
        dfy = lambda y: 10*np.sign(y - 2)
    return f, (dfx, dfy)

def global_search_random(f, n, x_rng, N=1000, is_timed=False):
    # init l and u ranges
    l = [r[0] for r in x_rng]
    u = [r[1] for r in x_rng]
    # init best point and value
    best_x = None
    best_f = 1e20
    # track for plotting
    xs, fs = [], []
    # for each random point
```

```
    for _ in range(N):
        # generate point and value
        this_x = [uniform(l[i], u[i]) for i in range(n)]
        this_f = f(*this_x)
        # keep track of best so far
        if this_f < best_f:
            best_x = deepcopy(this_x)
            best_f = this_f
        # for plotting
        if not is_timed:
            xs.append(deepcopy(best_x))
            fs.append(best_f)
    return xs, fs

def global_search_population(f, n, x_rng, N=100, M=10, num_iters=10, is_timed=False):
    # lambda to sort two lists based on values in first list
    sort_lists = lambda l0, l1: map(list, zip(*sorted(zip(l0, l1))))
    # init l and u ranges
    l = [r[0] for r in x_rng]
    u = [r[1] for r in x_rng]
    # init best point and value
    best_x = None
    best_f = 1e20
    # track for plotting
    xs, fs = [], []
    # generate initial set of N points
    this_xs = [
        [uniform(l[i], u[i]) for i in range(n)]
        for _ in range(N)
    ]
    this_fs = [0] * N
    for i in range(N):
        this_x = this_xs[i]
        this_f = f(*this_x)
        this_fs[i] = this_f
        if this_f < best_f:
            best_x = deepcopy(this_x)
            best_f = this_f
        # for plotting
        if not is_timed:
            xs.append(deepcopy(best_x))
            fs.append(best_f)
    # how many points to replace for each good one
    num_new = (N - M) // M
    # for each pruning iteration
    for _ in range(num_iters):
        # sort set of points with best at front
        this_fs, this_xs = sort_lists(this_fs, this_xs)
        # for each of the best M points
        for i in range(M):
            this_x = this_xs[i]
            # replace some bad points with perturbations of this point
            for j in range(num_new):
                pert_x = [x * uniform(0.8, 1.2) for x in this_x] # perturb by 0.8-1.2
                k = M + (i * num_new) + j
                this_xs[k] = deepcopy(pert_x)
                this_fs[k] = f(*pert_x)
                if this_fs[k] < best_f:
                    best_x = deepcopy(this_xs[k])
                    best_f = this_fs[k]
                # for plotting
                if not is_timed:
                    xs.append(deepcopy(best_x))
```

```
                    fs.append(best_f)
    return xs, fs

def gradient_descent(f, df, n, x0, alpha=0.01, num_iters=1000, is_timed=False):
    # init best point and value
    this_x = deepcopy(x0)
    this_f = f(*this_x)
    # track for plotting
    xs, fs = [], []
    if not is_timed:
        xs.append(deepcopy(this_x))
        fs.append(this_f)
    # for each iteration
    for _ in range(num_iters):
        # update parameter using gradient
        for i in range(n):
            this_x[i] -= alpha * df[i](this_x[i])
        # calculate value
        this_f = f(*this_x)
        # for plotting
        if not is_timed:
            xs.append(deepcopy(this_x))
            fs.append(this_f)
    return xs, fs

def part_ab_ii(func_num=1, is_part_b=False):
    print(f'=== Function {func_num} ===')
    # get function and init param values
    f, df = get_function(func_num)
    n = 2
    x_rng = [[3, 7], [0, 4]]
    x0 = [7, 7]
    num_iters = 200
    gsp_N, gsp_M, gsp_ni = 20, 5, 10
    # time the algorithms
    gsr_time = timeit(lambda: global_search_random(f, n, x_rng, N=num_iters,
        is_timed=True), number=100)
    gsp_time = timeit(lambda: global_search_population(f, n, x_rng, N=gsp_N, M=gsp_M,
        num_iters=gsp_ni, is_timed=True), number=100)
    gd_time = timeit(lambda: gradient_descent(f, df, n, x0, num_iters=num_iters,
        is_timed=True), number=100)
    print(f'> GSR time = {round(gsr_time, 4)}')
    if is_part_b: print(f'> GSP time = {round(gsp_time, 4)}')
    print(f'> GD time = {round(gd_time, 4)}')
    # run the algorithms
    gsr_xs, gsr_fs = global_search_random(f, n, x_rng, N=num_iters)
    gsp_xs, gsp_fs = global_search_population(f, n, x_rng, N=gsp_N, M=gsp_M,
        num_iters=gsp_ni)
    gd_xs, gd_fs = gradient_descent(f, df, n, x0, num_iters=num_iters)
    # output results
    print('> GSR min = %.6f' % gsr_fs[-1])
    if is_part_b: print('> GSP min = %.6f' % gsp_fs[-1])
    print('> GD min = %.6f' % gd_fs[-1])
    # plot results
    gsr_tpi = gsr_time / len(gsr_fs)
    gsp_tpi = gsp_time / len(gsp_fs)
    gd_tpi = gd_time / len(gd_fs)
    X_gsr = list(range(len(gsr_fs)))
    X_gsp = list(range(len(gsp_fs)))
    X_gd = list(range(len(gd_fs)))
    X_time_gsr = np.array(X_gsr) * gsr_tpi
    X_time_gsp = np.array(X_gsp) * gsp_tpi
    X_time_gd = np.array(X_gd) * gd_tpi
```

```
    _, (ax1, ax2) = plt.subplots(1, 2, figsize=(11, 5))
    plt.subplots_adjust(bottom=0.1, left=0.07, top=0.95, right=0.93)
    ax1.plot(X_gsr, gsr_fs, label='GS Random', color='tab:blue')
    if is_part_b: ax1.plot(X_gsp, gsp_fs, label='GS Population', color='tab:red')
    ax1.plot(X_gd, gd_fs, label=f'GD ($\\alpha=0.01$)', color='tab:green')
    ax1.set_xlabel('function evaluations')
    ax1.set_ylabel(f'$f_{func_num}(x)$')
    ax1.set_ylim(0, 5)
    ax1.legend()
    ax2.plot(X_time_gsr, gsr_fs, label='GS Random', color='tab:blue')
    if is_part_b: ax2.plot(X_time_gsp, gsp_fs, label='GS Population', color='tab:red')
    ax2.plot(X_time_gd, gd_fs, label=f'GD ($\\alpha=0.01$)', color='tab:green')
    ax2.set_xlabel('time (s)')
    ax2.set_ylabel(f'$f_{func_num}(x)$')
    ax2.set_ylim(0, 5)
    ax2.legend()
    plt.show()
    if is_part_b:
        if func_num == 1:
            X = np.linspace(-2, 8, 100)
            Y = np.linspace(-2, 8, 100)
        else:
            X = np.linspace(0, 8, 100)
            Y = np.linspace(0, 8, 100)
        Z = []
        for x in X:
            z = []
            for y in Y: z.append(f(x, y))
            Z.append(z)
        Z = np.array(Z)
        X, Y = np.meshgrid(X, Y)
        plt.contour(X, Y, Z, 100)
        plt.xlabel('$x_0$')
        plt.ylabel('$x_1$')
        plt.plot([x[1] for x in gsr_xs], [x[0] for x in gsr_xs],
            color='dimgrey', marker='x', markeredgecolor='tab:blue', markersize=3)
        plt.plot([x[1] for x in gsp_xs], [x[0] for x in gsp_xs],
            color='dimgrey', marker='x', markeredgecolor='tab:red', markersize=3)
        plt.plot([x[1] for x in gd_xs], [x[0] for x in gd_xs],
            color='dimgrey', marker='x', markeredgecolor='tab:green', markersize=3)
        if func_num == 1:
            plt.xlim([-2, 8])
            plt.ylim([0, 8])
        else:
            plt.xlim([0, 7])
            plt.ylim([1, 8])
        plt.show()

part_ab_ii(func_num=1, is_part_b=False)
part_ab_ii(func_num=2, is_part_b=False)
part_ab_ii(func_num=1, is_part_b=True)
part_ab_ii(func_num=2, is_part_b=True)
```

Code for (c)

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from copy import deepcopy
from random import uniform
from tensorflow import keras
from tensorflow.keras import layers, regularizers
from keras.layers import Dense, Dropout, Activation, Flatten, BatchNormalization
```

```
from keras.layers import Conv2D, MaxPooling2D, LeakyReLU
from keras.losses import CategoricalCrossentropy
from tensorflow.keras.optimizers import Adam
from sklearn.metrics import confusion_matrix, classification_report
from sklearn.utils import shuffle

def get_model_loss(batch_size, alpha, beta1, beta2, epochs):
    batch_size = int(batch_size) # ensure random values are ints
    epochs = int(epochs) # ensure random values are ints
    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[1:n]; y_train=y_train[1: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()
    model.add(Conv2D(16, (3,3), padding='same',
        input_shape=x_train.shape[1:],activation='relu'))
    model.add(Conv2D(16, (3,3), strides=(2,2), padding='same', activation='relu'))
    model.add(Conv2D(32, (3,3), padding='same', activation='relu'))
    model.add(Conv2D(32, (3,3), strides=(2,2), padding='same', activation='relu'))
    model.add(Dropout(0.5))
    model.add(Flatten())
    model.add(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"])
    history = 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 = CategoricalCrossentropy()
    return loss(y_test, y_preds).numpy()

def global_search_random(f, n, x_rng, N):
    l = [r[0] for r in x_rng]
    u = [r[1] for r in x_rng]
    best_x = None
    best_f = 1e20
    xs, fs = [], []
    vv = 0
    for _ in range(N):
        this_x = [uniform(l[i], u[i]) for i in range(n)]
        this_f = f(*this_x)
        if this_f < best_f:
            best_x = deepcopy(this_x)
            best_f = this_f
        xs.append(deepcopy(best_x))
        fs.append(best_f)
        print(vv, best_f, best_x)
        vv += 1
    return xs, fs

def global_search_population(f, n, x_rng, N, M, num_iters):
    sort_lists = lambda l0, l1: map(list, zip(*sorted(zip(l0, l1))))
    l = [r[0] for r in x_rng]
    u = [r[1] for r in x_rng]
    best_x = None
    best_f = 1e20
    xs, fs = [], []
```

```
    this_xs = [
        [uniform(l[i], u[i]) for i in range(n)]
        for _ in range(N)
    ]
    this_fs = [0] * N
    vv = 0
    for i in range(N):
        this_x = this_xs[i]
        this_f = f(*this_x)
        this_fs[i] = this_f
        if this_f < best_f:
            best_x = deepcopy(this_x)
            best_f = this_f
        xs.append(deepcopy(best_x))
        fs.append(best_f)
        print(vv, best_f, best_x)
        vv += 1
    num_new = (N - M) // M
    vv = 0
    for _ in range(num_iters):
        this_fs, this_xs = sort_lists(this_fs, this_xs)
        for i in range(M):
            this_x = this_xs[i]
            for j in range(num_new):
                # only perturbate appropriate values
                pert_x = [x for x in this_x]
                for ii in range(n):
                    if l[ii] != u[ii]: pert_x[ii] *= uniform(0.8, 1.2)
                #pert_x = [x * uniform(0.8, 1.2) for x in this_x]
                k = M + (i * num_new) + j
                this_xs[k] = deepcopy(pert_x)
                this_fs[k] = f(*pert_x)
                if this_fs[k] < best_f:
                    best_x = deepcopy(this_xs[k])
                    best_f = this_fs[k]
                xs.append(deepcopy(best_x))
                fs.append(best_f)
        print(vv, best_f, best_x)
        vv += 1
    return xs, fs

def c_i():
    n = 5
    x_rng = [
        [1, 128],
        [0.001, 0.001],
        [0.9, 0.9],
        [0.999, 0.999],
        [20, 20]
    ]
    print('gsr')
    gsr_xs, gsr_fs = global_search_random(get_model_loss, n, x_rng, N=50)
    print('gsp')
    gsp_xs, gsp_fs = global_search_population(get_model_loss, n, x_rng, N=12, M=4,
        num_iters=4)
    return gsr_xs, gsr_fs, gsp_xs, gsp_fs

def c_ii():
    n = 5
    x_rng = [
        [45, 45], # chosen from c(i)
        [0.1, 0.0001],
        [0.25, 0.99],
```

```
        [0.9, 0.9999],
        [20, 20]
    ]
    print('gsr')
    gsr_xs, gsr_fs = global_search_random(get_model_loss, n, x_rng, N=50)
    print('gsp')
    gsp_xs, gsp_fs = global_search_population(get_model_loss, n, x_rng, N=12, M=4,
        num_iters=4)
    return gsr_xs, gsr_fs, gsp_xs, gsp_fs

def c_iii():
    n = 5
    x_rng = [
        [45, 45], # chosen from c(i)
        [0.001, 0.001], # ignored c(ii) result
        [0.9, 0.9], # ignored c(ii) result
        [0.999, 0.999], # ignored c(ii) result
        [5, 30]
    ]
    print('gsr')
    gsr_xs, gsr_fs = global_search_random(get_model_loss, n, x_rng, N=50)
    print('gsp')
    gsp_xs, gsp_fs = global_search_population(get_model_loss, n, x_rng, N=12, M=4,
        num_iters=4)
    return gsr_xs, gsr_fs, gsp_xs, gsp_fs

gsr_xs1, gsr_fs1, gsp_xs1, gsp_fs1 = c_i()
gsr_xs2, gsr_fs2, gsp_xs2, gsp_fs2 = c_ii()
gsr_xs3, gsr_fs3, gsp_xs3, gsp_fs3 = c_iii()

def plot_results(gsr_xs, gsr_fs, gsp_xs, gsp_fs):
    gsr_N, gsp_N = len(gsr_fs), len(gsp_fs)
    gsr_X, gsp_X = list(range(gsr_N)), list(range(gsp_N))
    plt.plot(gsr_X, gsr_fs, label='GS Random')
    plt.plot(gsp_X, gsp_fs, label='GS Population')
    plt.xlabel('function evaluations')
    plt.ylabel('model loss')
    plt.legend()
    plt.show()

plot_results(gsr_xs1, gsr_fs1, gsp_xs1, gsp_fs1)
plot_results(gsr_xs2, gsr_fs2, gsp_xs2, gsp_fs2)
plot_results(gsr_xs3, gsr_fs3, gsp_xs3, gsp_fs3)
```