My week 4 functions to use in week 8

The functions I used in week 4 are:

Function	Expression
$f_1(x,y)$	$9(x-5)^4 + 10(y-2)^2$
$f_2(x,y)$	$max(x-5,0) + 10 \cdot \|y-2\|$

Table 1: Functions as expressions

Partial derivative	Expression
$\frac{\partial f_1}{\partial x}$	$36(x-5)^3$
$\frac{\partial f_1}{\partial y}$	$20y - 40$
$\frac{\partial f_2}{\partial x}$	$\theta(x-5)$
$\frac{\partial f_2}{\partial y}$	$10 \cdot sign(y-2)$

Table 2: Partial derivatives as expressions

Question (a)

Part (i)

The global random search algorithm starts by setting the best function value to infinity so the first point evaluated becomes the new best point, since an initial function value from search will almost certainly be smaller than our large value. Subsequently, N iterations are completed to generate a random parameter vector called `current_point` where each element is `current_point`^{i} and in the range of $[l_i, u_i]$. While iterating, the best function value (i.e. the lowest) and its corresponding parameter vector is tracked and stored as shown below:

```
best_function_value = float('inf')
best_point = None
for _ in range(N):
    current_point = []
    for i in range(number_of_parameters):
        current_point.append(uniform(l[i], u[i]))
    current_function_value = function_to_optimize(*current_point)
    if current_function_value < best_function_value:
        best_function_value = current_function_value
        best_point = deepcopy(current_point)
```

Part (ii)

I used the gradient descent algorithm from the week 2 assignment which used $\alpha = 0.01$ so that I can compare it with the global random search algorithm. I used the `timeit` module to measure the average time it took to run both algorithms for 200 iterations. For global random search, it was ran for N iterations which has N calls to the main function (where $N = 200$). There are no derivative evaluations since global random search only uses function evaluations to randomly explore the parameter space. On the other hand, gradient descent is ran for N iterations which has $2 * N$ calls to the derivative function and N calls to the main function (where $N = 200$) which makes it more computationally expensive but able to converge efficiently compared to global random search. For both algorithms, the measured running times are divided by the number of calls to the main function for calculating an evaluation's average time. Time is hard to measure accurately here since the time to run is so small, inaccuracies from measuring at this granularity might make results useless. For the fairest comparison I could make, after each evaluation and second (unit of time I used), I plotted the value of the function. These results are shown for f_1 below.

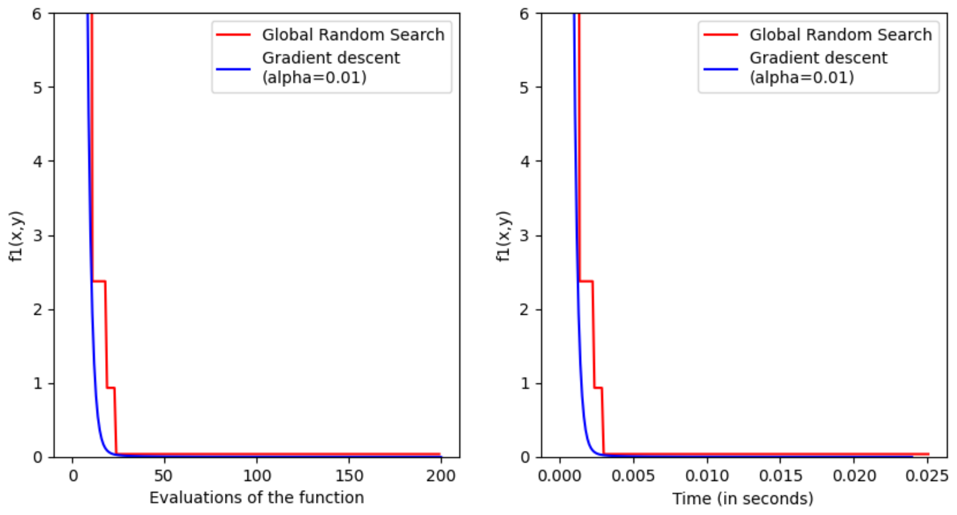

Figure 1: Evaluations and time across change in $f_1(x)$

The plot above shows that for f_1, gradient descent manages to converge to the minimum in less than 40 iterations. However, the global random search algorithm finds a similar value to the minimum, but does not reach the same minimum (seen by the red line being slightly higher than the blue line). This happened because the chance of randomly choosing appropriate values for `current_point` is very low. In terms of time, both algorithms use a very similar amount of time, so the time measurements are not useful here. This aligns with what I mentioned about the difficulty of granularity for time taken being so small and potentially making results mean very little. Next, the results are shown for f_2 below.

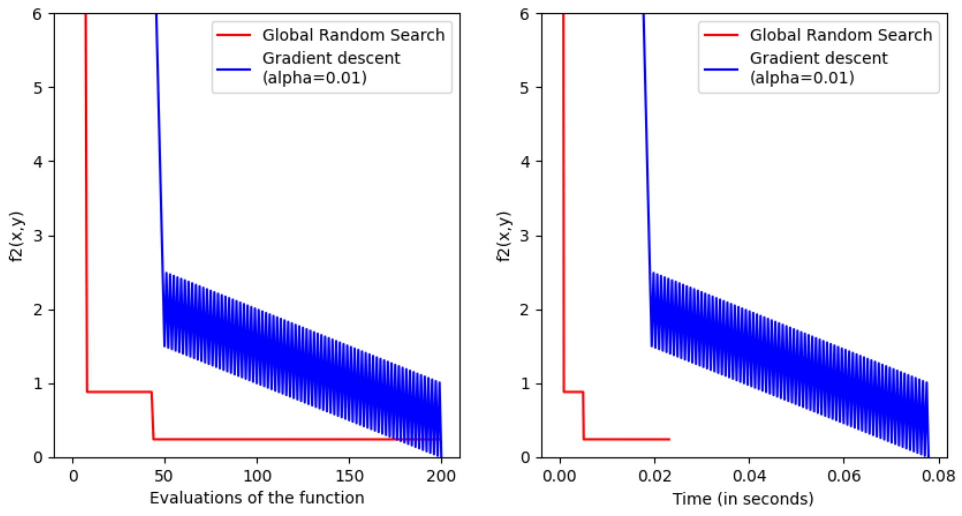

Figure 2: Evaluations and time across change in $f_2(x)$

The plot above shows for that for f_2, gradient descent does not converge since it oscillates up and down when going toward the minimum since α is too big. On the other hand, global random search is good since it converges on a relatively small value while not being much worse than the gradient descent algorithm. In terms of time, the global random search algorithm is faster compared to the gradient descent algorithm, which has a call to the derivative function containing 2 `NumPy` functions named `sign()` and `heaviside()` which need to be called.

Question (b)

Part (i)

The global population search algorithm is similar to global random search since it starts by creating a list of N parameter vectors called `list_of_points` with their corresponding function values as shown below:

```
list_of_function_values = [0] * N
list_of_points = []
for _ in range(N):
    point = []
    for i in range(number_of_parameters):
        point.append(uniform(l[i], u[i]))
    list_of_points.append(point)
for n_index in range(N):
    current_point = list_of_points[n_index]
    current_function_value = function_to_optimize(*current_point)
    list_of_function_values[n_index] = current_function_value
```

After this, the list of N vectors is used to choose the best M results. From the best M results, with each element being m, the bottom $N-M$ results are taken and $\frac{N-M}{M}$ of them are replaced by a randomly perturbed version of m (which is calculated by multiplying every parameter in m by a random number from 0.8 to 1.2) as shown:

```
sort_two_lists_based_on_first_list = lambda list_one, list_two: map(list,
zip(*sorted(zip(list_one, list_two))))
fraction_to_replace_bottom_results_by = (N - M) // M
lower_multiplication_value_for_m = 0.8
upper_multiplication_value_for_m = 1.2
for _ in range(iterations):
    list_of_function_values, list_of_points =
    sort_two_lists_based_on_first_list(list_of_function_values,
    list_of_points)
    for m_index in range(M):
        current_point = list_of_points[m_index]
        for fraction_index in range(fraction_to_replace_bottom_results_by):
            perturbed_point = [x *
            uniform(lower_multiplication_value_for_m,
            upper_multiplication_value_for_m) for x in current_point]
            point_being_perturbed = M + (m_index *
            fraction_to_replace_bottom_results_by) + fraction_index
            list_of_function_values[point_being_perturbed] =
            function_to_optimize(*perturbed_point)
            list_of_points[point_being_perturbed] =
            deepcopy(perturbed_point)
```

Part (ii)

I applied the global population search algorithm to the two functions from the week 4 assignment. The `timeit` module was used once again to measure the average running time of the global population search algorithm for $N = 20$, $M = 5$ and $iterations = 10$. This results in $N + iterations \cdot (N - M)$ function evaluations. Like global random search, no derivative evaluations are used. I plotted the value of the function and after each evaluation and second (unit of time I used) for global population search, as well as the gradient descent and global random search algorithms from question (a). The results for f_1 are shown below.

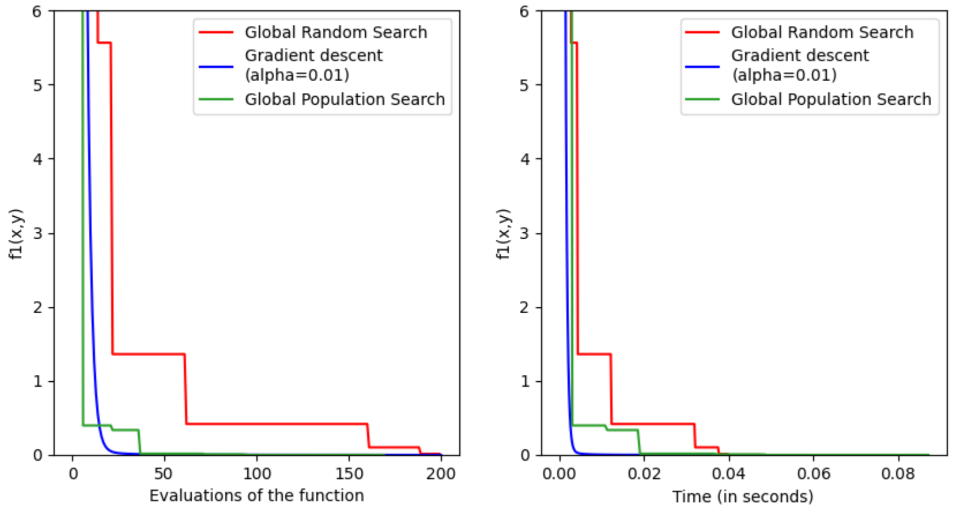

Figure 3: Evaluations and time across change in $f_1(x)$

For f_1, it is clear that global population search is much better than global random search (since the green line converges to the minimum in fewer iterations compared to the red line). Gradient descent converges marginally quicker than global population search in this case. In terms of time, global population search takes almost double the time per evaluation compared to gradient descent, with global random search taking the longest time. A contour plot is shown for f_1 below.

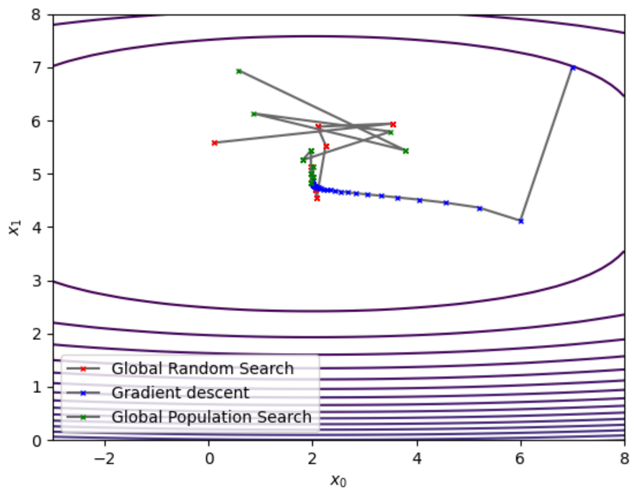

Figure 4: Change in x across evaluations

The contour plot shows that global population search descends in a rough manner that is not smooth like gradient descent. Each new best value is very close to the previous best value, and it is evident that it descends in a jagged manner (seen as zig-zags in green above). The results for f_2 are shown below.

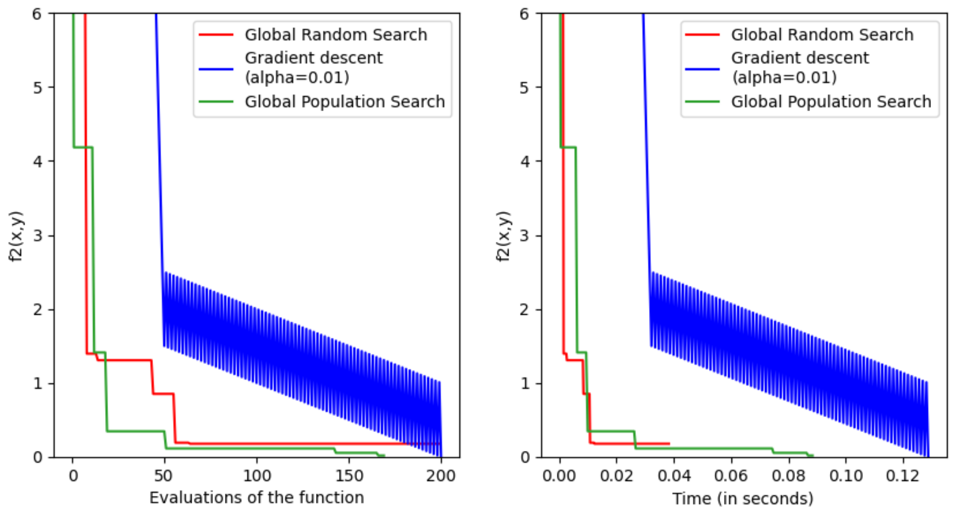

Figure 5: Evaluations and time across change in $f_2(x)$

For f_2, it is clear that global population search algorithm is better at converging on the minimum compared to gradient descent and global random search. Gradient descent does not find the lowest minimum (unlike global population search which does). Global random search oscillates up and down and fails to converge as seen in question (a). In terms of time, global population search is more than twice as fast compared to gradient descent. Therefore, global population search finds a minimum lower than gradient descent while also being more than twice as fast. A contour plot is shown for f_2 below.

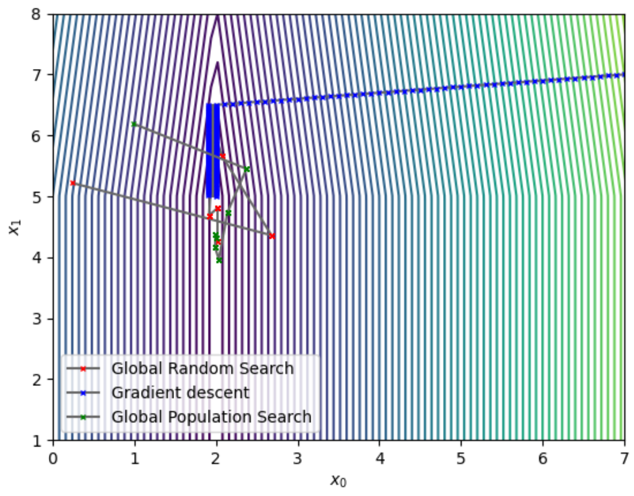

Figure 6: Change in x across evaluations

The contour plot shows that global population search finds a minimum very quickly (since it reaches a minimal point quickly in green above). On the other hand, gradient descent is oscillating as seen before across the minimum ridge (since the blue line is thick showing the oscillations occurring). We can conclude that global population search is an effective algorithm which is twice as quick as gradient descent and finds a lower minimum for f_1 and f_2.

Question (c)

I passed the compilation and training in the conv net model that was downloaded into a separate method which takes mini-batch size, α, β_1, β_2 and the number of epochs to run, and returned the categorical cross-entropy loss of the model. This method was passed to the global random search and global population search algorithms which were developed in question (a) and (b), so that optimal hyperparameters can be found. Global random search was run for 50 iterations, and global population search was completed using $iterations = 4, M = 4$ and $N = 12$ which amounted to 44 iterations.

Part (i)

I used a constant number of epochs, which was 20. I also kept the Adam parameters constant at $\alpha = 0.001, \beta_1 = 0.9$ and $\beta_2 = 0.999$. A range of mini-batch sizes were searched for in a range from 1 to 128. After running this experiment, it was observed that the global random search algorithm runs a little better than the global population search algorithm since it has a lower model loss while using less function evaluations as shown below:

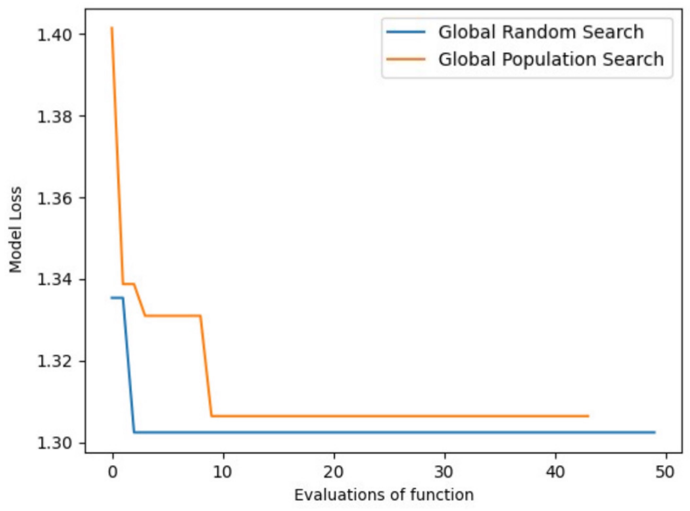

Figure 7: Iterations causing change in model loss (when choosing batch size)

Subsequently, it was found that the best batch size is 45. However, it is important to note that the global population search algorithm was not necessarily ”inefficient”, since it still performed fairly well. The better attempt by the global random search algorithm is because of the parameter search which only uses discrete values instead of values with decimal points (ranging across basically a bigger and infinite set of values), so this limitation for parameter search gives it an advantage.

Part (ii)

I used a constant mini-batch size of 45 which was discovered in part (i). I also used a constant number of epochs, which was 20. A range of adam parameters were searched for in a range with α being between 0.0001 and 0.1, β_1 being between 0.25 and 0.99, and β_2 being between 0.9 and 0.9999. After running this experiment, it was found that the global population search algorithm was a bit better than the global random search algorithm when looking for Adam parameters since it has a lower model loss but took more function evaluations. However, the default values (i.e. $\alpha = 0.001, \beta_1 = 0.9$ and $\beta_2 = 0.999$) were not able to be beaten by both algorithms.

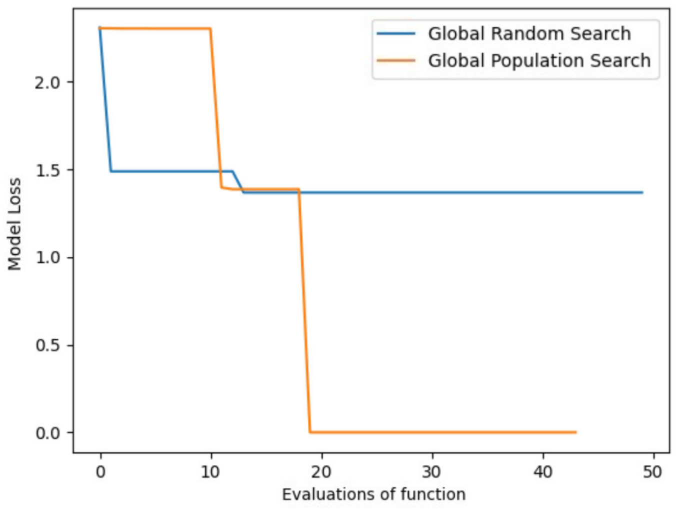

Figure 8: Iterations causing change in model loss (when choosing Adam parameters)

The reason the default values were not beaten is because the parameters are randomly searched in a uniform way. For example, it is common to use small values for α, but random values for α being 0.0001 (very small) is not as likely compared to 0.1 (relatively bigger). This could potentially generate values randomly in a way that is logarithmic. Therefore, the default values (i.e. $\alpha = 0.001, \beta_1 = 0.9$ and $\beta_2 = 0.999$) seem to beat any other algorithm.

Part (iii)

I used a constant mini-batch size of 45 which was discovered in part (i). I used the default Adam parameters which were found to be the best from part (ii) (i.e. $\alpha = 0.001, \beta_1 = 0.9$ and $\beta_2 = 0.999$). A range epochs was searched for in a range from 5 to 30. The global random search algorithm was better (18 epochs) than the global population search algorithm (21 epochs). But both of these algorithms once again did not beat an even better outcome from the default number of epochs (20). Overall, the best value for mini-size was 45. The best values for Adam parameters was the defaults (i.e. $\alpha = 0.001, \beta_1 = 0.9$ and $\beta_2 = 0.999$), and the best values for the number of epochs was also the default of 20. Overall, global random search performed surprisingly well due to advantages such as a limited parameter search (discrete values only), but global population search was also not bad - Furthermore, a repeating pattern was that the default values for hyperparameters were not able to be beaten by both algorithms.

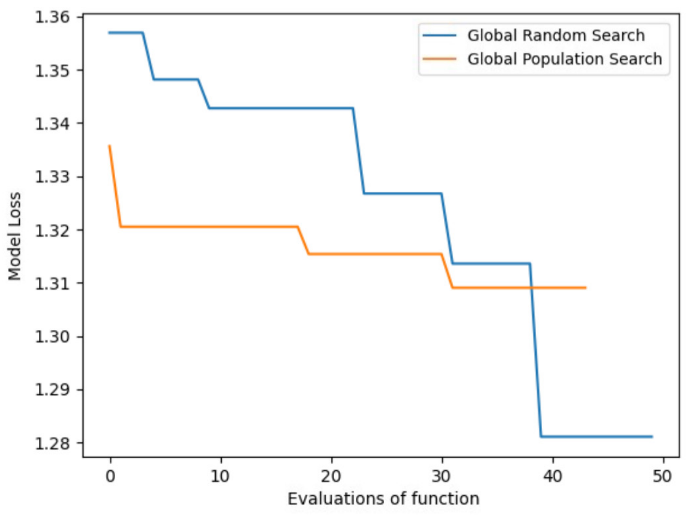

Figure 9: Iterations causing change in model loss (when choosing number of epochs)

1 Appendix

1.1 Question (a)

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


def global_random_search(function_to_optimize, number_of_parameters,
min_and_max_for_each_param, N, measure_time=False):
    # Min and max value of each parameter (l for lower bound and u for
    upper bound)
    l = []
    u = []
    for tuple_of_min_and_max in min_and_max_for_each_param:
        l.append(tuple_of_min_and_max[0])
        u.append(tuple_of_min_and_max[1])
    best_function_value_at_each_iteration = []
    best_point_at_each_iteration = []
    best_function_value = float('inf')
    best_point = None
    for _ in range(N):
        current_point = []
        for i in range(number_of_parameters):
            current_point.append(uniform(l[i], u[i]))
        current_function_value = function_to_optimize(*current_point)
        if current_function_value < best_function_value:
            best_function_value = current_function_value
            best_point = deepcopy(current_point)
        if not measure_time:
            best_function_value_at_each_iteration
            .append(best_function_value)
            best_point_at_each_iteration.append(deepcopy(best_point))
    return best_point_at_each_iteration,
    best_function_value_at_each_iteration


def gradient_descent(function, partial_derivative_functions, n,
initial_x_value, alpha, iterations, measure_time=False):
    current_point = deepcopy(initial_x_value)
    current_function_value = function(*current_point)
    function_values = []
    parameter_values = []
    if not measure_time:
        function_values.append(current_function_value)
        parameter_values.append(deepcopy(current_point))
    for _ in range(iterations):
        for i in range(n):
            current_point[i] = current_point[i] - (alpha *
            partial_derivative_functions[i](current_point[i]))
        current_function_value = function(*current_point)
        if not measure_time:
            function_values.append(current_function_value)
            parameter_values.append(deepcopy(current_point))

    return parameter_values, function_values


def get_first_function_and_derivatives():
    function = lambda x, y: 9 * (x - 5) ** 4 + 10 * (y - 2) ** 2
    derive_function_with_respect_to_x = lambda x: 36 * (x - 5) ** 3
    derive_function_with_respect_to_y = lambda y: 20 * y - 40
    return function, (derive_function_with_respect_to_x,
    derive_function_with_respect_to_y)


def get_second_function_and_derivatives():
    function = lambda x, y: 10 * abs(y - 2) + max(0, x - 5)
    derive_function_with_respect_to_x = lambda x: np.heaviside(x - 5, 0)
    derive_function_with_respect_to_y = lambda y: 10 * np.sign(y - 2)
    return function, (derive_function_with_respect_to_x,
    derive_function_with_respect_to_y)


def part_ii(function_name, function_and_derivatives):
    min_and_max_for_each_parameter = [[3, 7], [0, 4]]
    initial_x_value = [7, 7]
    number_of_parameters = 2
    f, df = function_and_derivatives
    iterations = 200
    print(f'Comparing global random search to gradient descent using
    {function_name}')
    # Measure time for algorithms
```

```
time_for_global_random_search = timeit(lambda: global_random_search(f,
number_of_parameters, min_and_max_for_each_parameter, N=iterations,
measure_time=True), number=100)
time_for_gradient_descent = timeit(lambda: gradient_descent(f, df,
number_of_parameters, initial_x_value, iterations=iterations,
measure_time=True, alpha=0.01), number=100)
print(f'Time for global random search:
{time_for_global_random_search}')
print(f'Time for gradient descent: {time_for_gradient_descent}')
# Run the algorithms
_, global_random_search_best_function_value = global_random_search(f,
number_of_parameters, min_and_max_for_each_parameter, N=iterations)
_, gradient_descent_best_function_value = gradient_descent(f, df,
number_of_parameters, initial_x_value, iterations=iterations,
alpha=0.01)
print(f'Minimum for global random search =
{global_random_search_best_function_value[-1]}')
print(f'Minimum for gradient descent =
{gradient_descent_best_function_value[-1]}')
# Plot measurements
global_random_search_time_per_iteration =
time_for_global_random_search / len(
    global_random_search_best_function_value)
gradient_descent_time_per_iteration = time_for_gradient_descent /
len(gradient_descent_best_function_value)
global_random_search_function_evaluations =
list(range(len(global_random_search_best_function_value)))
gradient_descent_function_evaluations =
list(range(len(gradient_descent_best_function_value)))
global_random_search_execution_time = np.array(
    global_random_search_function_evaluations) *
    global_random_search_time_per_iteration
gradient_descent_search_execution_time = np.array(
    gradient_descent_function_evaluations) *
    gradient_descent_time_per_iteration
_, (plot_1, plot_2) = plt.subplots(1, 2, figsize=(10, 5))
plot_1.plot(global_random_search_function_evaluations,
global_random_search_best_function_value,
            label='Global Random Search', color='red')

plot_1.plot(gradient_descent_function_evaluations,
gradient_descent_best_function_value,
            label=f'Gradient descent \n(alpha=0.01)', color='blue')
plot_1.set_ylim(0, 6)
plot_1.set_xlabel('Evaluations of the function')
plot_1.set_ylabel(function_name)
plot_1.legend()
plot_2.plot(global_random_search_execution_time,
global_random_search_best_function_value,
            label='Global Random Search', color='red')
plot_2.plot(gradient_descent_search_execution_time,
gradient_descent_best_function_value,
            label=f'Gradient descent \n(alpha=0.01)', color='blue')
plot_2.set_ylim(0, 6)
plot_2.set_xlabel('Time (in seconds)')
```

```
    plot_2.set_ylabel(function_name)
    plot_2.legend()
    plt.savefig(f"plots/a_{function_name[0:2]}")
    plt.show()


if __name__ == '__main__':
    part_ii("f1(x,y)", get_first_function_and_derivatives())
    part_ii("f2(x,y)", get_second_function_and_derivatives())
```

1.2 Question (b)

```
from copy import deepcopy
from random import uniform
from timeit import timeit

import numpy as np
from matplotlib import pyplot as plt

from part_a import global_random_search


def global_population_search(function_to_optimize, number_of_parameters,
min_and_max_for_each_param, N, M, iterations,
                                measure_time=False):
    # Min and max value of each parameter (l for lower bound and u for
    upper bound)
    l = []
    u = []
    for tuple_of_min_and_max in min_and_max_for_each_param:
        l.append(tuple_of_min_and_max[0])
        u.append(tuple_of_min_and_max[1])
    best_function_value_at_each_iteration = []
    best_point_at_each_iteration = []
    best_function_value = float('inf')
    best_point = None
    list_of_function_values = [0] * N
    list_of_points = []
    for _ in range(N):
        point = []
        for i in range(number_of_parameters):
            point.append(uniform(l[i], u[i]))
        list_of_points.append(point)
    for n_index in range(N):
        current_point = list_of_points[n_index]
        current_function_value = function_to_optimize(*current_point)
        list_of_function_values[n_index] = current_function_value
        if current_function_value < best_function_value:
            best_function_value = current_function_value
            best_point = deepcopy(current_point)
        if not measure_time:
            best_function_value_at_each_iteration.append(
            best_function_value)
            best_point_at_each_iteration.append(deepcopy(best_point))
    sort_two_lists_based_on_first_list = lambda list_one, list_two:
    map(list, zip(*sorted(zip(list_one, list_two))))
    fraction_to_replace_bottom_results_by = (N - M) // M
    lower_multiplication_value_for_m = 0.8
    upper_multiplication_value_for_m = 1.2
    for _ in range(iterations):
        list_of_function_values, list_of_points =
        sort_two_lists_based_on_first_list(list_of_function_values,
        list_of_points)
        for m_index in range(M):
            current_point = list_of_points[m_index]
            for fraction_index in
            range(fraction_to_replace_bottom_results_by):
                perturbed_index = M + (m_index *
                fraction_to_replace_bottom_results_by) + fraction_index
                perturbed_point = [x *
                uniform(lower_multiplication_value_for_m,
                upper_multiplication_value_for_m) for x
                                   in current_point]
                list_of_function_values[perturbed_index] =
                function_to_optimize(*perturbed_point)
                list_of_points[perturbed_index] = deepcopy(perturbed_point)
                if list_of_function_values[perturbed_index] <
                best_function_value:
                    best_function_value =
                    list_of_function_values[perturbed_index]
                    best_point = deepcopy(list_of_points[perturbed_index])
                if not measure_time:
                    best_function_value_at_each_iteration.append(
                    best_function_value)
                    best_point_at_each_iteration.append(deepcopy(
                    best_point))

    return best_point_at_each_iteration,
    best_function_value_at_each_iteration


def gradient_descent(function, partial_derivative_functions, n,
initial_x_value, alpha, iterations, measure_time=False):
    current_point = deepcopy(initial_x_value)
    current_function_value = function(*current_point)
    function_values = []
    parameter_values = []
    if not measure_time:
        function_values.append(current_function_value)
        parameter_values.append(deepcopy(current_point))
    for _ in range(iterations):
        for i in range(n):
            current_point[i] = current_point[i] - (alpha *
            partial_derivative_functions[i](current_point[i]))
        current_function_value = function(*current_point)
        if not measure_time:
            function_values.append(current_function_value)
            parameter_values.append(deepcopy(current_point))
    return parameter_values, function_values


def get_first_function_and_derivatives():
    function = lambda x, y: 9 * (x - 5) ** 4 + 10 * (y - 2) ** 2
    derive_function_with_respect_to_x = lambda x: 36 * (x - 5) ** 3
    derive_function_with_respect_to_y = lambda y: 20 * y - 40
    return function, (derive_function_with_respect_to_x,
    derive_function_with_respect_to_y)


def get_second_function_and_derivatives():
    function = lambda x, y: 10 * abs(y - 2) + max(0, x - 5)
    derive_function_with_respect_to_x = lambda x: np.heaviside(x - 5, 0)
    derive_function_with_respect_to_y = lambda y: 10 * np.sign(y - 2)
    return function, (derive_function_with_respect_to_x,
    derive_function_with_respect_to_y)


def part_ii(function_name, function_and_derivatives):
    min_and_max_for_each_parameter = [[3, 7], [0, 4]]
    initial_x_value = [7, 7]
    number_of_parameters = 2
    f, df = function_and_derivatives
    iterations = 200
    N = 20
    M = 5
    iterations_for_pruning = 10

    print(f'Comparing global random search to gradient descent using
    {function_name}')
    # Measure time for algorithms
    time_for_global_random_search = timeit(lambda: global_random_search(f,
    number_of_parameters, min_and_max_for_each_parameter, N=iterations,
    measure_time=True), number=100)
    time_for_gradient_descent = timeit(lambda: gradient_descent(f, df,
    number_of_parameters, initial_x_value, iterations=iterations,
    measure_time=True, alpha=0.01), number=100)
    time_for_global_population_search = timeit(
        lambda: global_population_search(f, number_of_parameters,
        min_and_max_for_each_parameter, N=N, M=M,
                                                iterations=iterations_for_pruning,
                                                measure_time=True), number=100)

    print(f'Time for global random search:
    {time_for_global_random_search}')
    print(f'Time for gradient descent: {time_for_gradient_descent}')
    print(f'Time for global population search:
    {time_for_global_population_search}')
    # Run the algorithms
    global_random_search_best_point,
    global_random_search_best_function_value = global_random_search(f,
    number_of_parameters, min_and_max_for_each_parameter, N=iterations)
```

```
gradient_descent_best_point, gradient_descent_best_function_value =
gradient_descent(f, df, number_of_parameters, initial_x_value,
iterations=iterations, alpha=0.01)
global_population_search_best_point,
global_population_search_best_function_value =
global_population_search(f, number_of_parameters,
min_and_max_for_each_parameter, N=N, M=M,
iterations=iterations_for_pruning)
print(f'Minimum for global random search =
{global_random_search_best_function_value[-1]}')
print(f'Minimum for gradient descent =
{gradient_descent_best_function_value[-1]}')
print(f'Minimum for global population search =
{global_population_search_best_function_value[-1]}')

# Plot measurements
global_random_search_time_per_iteration =
time_for_global_random_search / len(
    global_random_search_best_function_value)
gradient_descent_time_per_iteration = time_for_gradient_descent /
len(gradient_descent_best_function_value)
global_population_search_time_per_iteration =
time_for_global_population_search / len(
    global_population_search_best_function_value)
global_random_search_function_evaluations =
list(range(len(global_random_search_best_function_value)))
gradient_descent_function_evaluations =
list(range(len(gradient_descent_best_function_value)))
global_population_search_function_evaluations =
list(range(len(global_population_search_best_function_value)))
global_random_search_execution_time = np.array(
    global_random_search_function_evaluations) *
    global_random_search_time_per_iteration
gradient_descent_search_execution_time = np.array(
    gradient_descent_function_evaluations) *
    gradient_descent_time_per_iteration
global_population_search_execution_time = np.array(
    global_population_search_function_evaluations) *
    global_population_search_time_per_iteration

_, (plot_1, plot_2) = plt.subplots(1, 2, figsize=(10, 5))
plot_1.plot(global_random_search_function_evaluations,
global_random_search_best_function_value,
            label='Global Random Search', color='red')

plot_1.plot(gradient_descent_function_evaluations,
gradient_descent_best_function_value,
            label=f'Gradient descent \n(alpha=0.01)', color='blue')

plot_1.plot(global_population_search_function_evaluations,
global_population_search_best_function_value,
            label='Global Population Search', color='tab:green')

plot_1.set_ylim(0, 6)
plot_1.set_xlabel('Evaluations of the function')
```

```
    plot_1.set_ylabel(function_name)
    plot_1.legend()
    plot_2.plot(global_random_search_execution_time,
    global_random_search_best_function_value,
                label='Global Random Search', color='red')
    plot_2.plot(gradient_descent_search_execution_time,
    gradient_descent_best_function_value,
                label=f'Gradient descent \n(alpha=0.01)', color='blue')
    plot_2.plot(global_population_search_execution_time,
    global_population_search_best_function_value,
                label='Global Population Search', color='tab:green')
    plot_2.set_ylim(0, 6)
    plot_2.set_xlabel('Time (in seconds)')
    plot_2.set_ylabel(function_name)
    plot_2.legend()
    plt.savefig(f"plots/b_{function_name[0:2]}")
    plt.show()
    X = np.linspace(-3, 8, 100)
    Y = np.linspace(-3, 8, 100)
    if function_name == "f2(x,y)":
        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)
    X, Y = np.meshgrid(X, Y)
    Z = np.array(Z)
    plt.ylabel('x_1')
    plt.xlabel('x_0')
    plt.contour(X, Y, Z, 100)
    plt.plot([x[1] for x in global_random_search_best_point], [x[0] for x
    in global_random_search_best_point],
             color='dimgrey', label='Global Random Search', marker='x',
             markeredgecolor='red', markersize=3)
    plt.plot([x[1] for x in gradient_descent_best_point], [x[0] for x in
    gradient_descent_best_point],
             color='dimgrey', label='Gradient descent', marker='x',
             markeredgecolor='blue', markersize=3)
    plt.plot([x[1] for x in global_population_search_best_point], [x[0]
    for x in global_population_search_best_point],
             color='dimgrey', label='Global Population Search',
             marker='x', markeredgecolor='green', markersize=3)
    plt.legend()
    plt.xlim([-3, 8])
    plt.ylim([0, 8])
    if function_name == "f2(x,y)":
        plt.xlim([0, 7])
        plt.ylim([1, 8])
    plt.savefig(f"plots/b_contour_{function_name[0:2]}")
    plt.show()


if __name__ == '__main__':
    part_ii("f1(x,y)", get_first_function_and_derivatives())
    part_ii("f2(x,y)", get_second_function_and_derivatives())
```

1.3 Question (c)

```
import matplotlib.pyplot as plt
from copy import deepcopy
from random import uniform
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D
from keras.losses import CategoricalCrossentropy
from keras import datasets, utils, Sequential, regularizers
from keras.optimizers import Adam


def part_i():
    # 5 parameters
    min_and_max_for_each_param = [[1, 128], [0.001, 0.001], [0.9, 0.9],
    [0.999, 0.999], [20, 20]]
    global_population_search_x_list, global_population_search_f_list =
    global_population_search(model_loss, 5, min_and_max_for_each_param,
    N=12, M=4, iterations=4)
    global_random_search_x_list, global_random_search_f_list =
    global_random_search(model_loss, 5, min_and_max_for_each_param, N=50)
    return (global_random_search_x_list, global_random_search_f_list,
    global_population_search_x_list,
            global_population_search_f_list)


def part_ii():
    # 5 parameters and [45, 45] is from part (i)
    min_and_max_for_each_param = [[45, 45], [0.1, 0.0001], [0.25, 0.99],
    [0.9, 0.9999], [20, 20]]
    global_population_search_x_list, global_population_search_f_list =
    global_population_search(model_loss, 5, min_and_max_for_each_param,
    N=12, M=4, iterations=4)
    global_random_search_x_list, global_random_search_f_list =
    global_random_search(model_loss, 5, min_and_max_for_each_param, N=50)
    return global_random_search_x_list, global_random_search_f_list,
    global_population_search_x_list, global_population_search_f_list


def part_iii():
    # 5 parameters and [45, 45] is from part (i), index 1 to 3 were
    ignored from part (ii), then index 4 (i.e. [5,30]) is used now
    min_and_max_for_each_param = [[45, 45], [0.001, 0.001], [0.9, 0.9],
    [0.999, 0.999], [5, 30]]
    global_population_search_x_list, global_population_search_f_list =
    global_population_search(model_loss, 5, min_and_max_for_each_param,
    N=12, M=4, iterations=4)
    global_random_search_x_list, global_random_search_f_list =
    global_random_search(model_loss, 5, min_and_max_for_each_param, N=50)
    return global_random_search_x_list, global_random_search_f_list,
    global_population_search_x_list, global_population_search_f_list
```

```
def plot(global_random_search_function_value,
global_population_search_function_value, hyperparameter):
    global_random_search_function_evaluations,
    global_population_search_function_evaluations =
    len(global_random_search_function_value),
    len(global_population_search_function_value)
    global_random_search_evaluation_indices,
    global_population_search_evaluation_indices =
    list(range(global_random_search_function_evaluations)),
    list(range(global_population_search_function_evaluations))
    plt.plot(global_population_search_evaluation_indices,
    global_population_search_function_value, label='Global Population
    Search')
    plt.plot(global_random_search_evaluation_indices,
    global_random_search_function_value, label='Global Random Search')
    plt.ylabel('Model Loss')
    plt.xlabel('Evaluations of function')
    plt.legend()
    plt.savefig(f"plots/c_{hyperparameter}")
    plt.show()


# Credit for code in function:
https://www.scss.tcd.ie/Doug.Leith/CSU44061/week8.py
def model_loss(mini_batch_size, a, b1, b2, number_of_epochs):
    epochs = int(number_of_epochs)
    mini_batch_size = int(mini_batch_size)
    num_classes = 10
    (x_train, y_train), (x_test, y_test) = datasets.cifar10.load_data()
    n = 5000
    y_train = y_train[1:n]
    x_train = x_train[1:n]
    y_test = utils.to_categorical(y_test, num_classes)
    x_test = x_test.astype("float32") / 255
    y_train = utils.to_categorical(y_train, num_classes)
    x_train = x_train.astype("float32") / 255
    model = 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=a, beta_1=b1, beta_2=b2)
    model.compile(loss="categorical_crossentropy", optimizer=optimizer,
                  metrics=["accuracy"])
    model.fit(x_train, y_train, batch_size=mini_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_random_search(function_to_optimize, number_of_parameters,
min_and_max_for_each_param, N):
    # Min and max value of each parameter (l for lower bound and u for
    upper bound)
    l = []
    u = []
    for tuple_of_min_and_max in min_and_max_for_each_param:
        l.append(tuple_of_min_and_max[0])
        u.append(tuple_of_min_and_max[1])
    best_function_value_at_each_iteration = []
    best_point_at_each_iteration = []
    best_function_value = float('inf')
    best_point = None
    function_evaluations = 0
    for _ in range(N):
        current_point = []
        for i in range(number_of_parameters):
            current_point.append(uniform(l[i], u[i]))
        current_function_value = function_to_optimize(*current_point)
        if current_function_value < best_function_value:
            best_function_value = current_function_value
            best_point = deepcopy(current_point)
        best_function_value_at_each_iteration.append(best_function_value)
        best_point_at_each_iteration.append(deepcopy(best_point))
        print(function_evaluations, best_point, best_function_value)
        function_evaluations = function_evaluations + 1
    return best_point_at_each_iteration,
    best_function_value_at_each_iteration


def global_population_search(function_to_optimize, number_of_parameters,
min_and_max_for_each_param, N, M, iterations):
    # Min and max value of each parameter (l for lower bound and u for
    upper bound)
    l = []
    u = []
    for tuple_of_min_and_max in min_and_max_for_each_param:
        l.append(tuple_of_min_and_max[0])
        u.append(tuple_of_min_and_max[1])
    best_function_value_at_each_iteration = []
    best_point_at_each_iteration = []
    best_function_value = float('inf')
    best_point = None
    list_of_function_values = [0] * N
    list_of_points = []
    function_evaluations = 0
    for _ in range(N):
        point = []
        for i in range(number_of_parameters):
            point.append(uniform(l[i], u[i]))
            list_of_points.append(point)
    for n_index in range(N):
        current_point = list_of_points[n_index]
        current_function_value = function_to_optimize(*current_point)
        list_of_function_values[n_index] = current_function_value
        if current_function_value < best_function_value:
            best_function_value = current_function_value
            best_point = deepcopy(current_point)
        best_function_value_at_each_iteration.append(best_function_value)
        best_point_at_each_iteration.append(deepcopy(best_point))
        print(function_evaluations, best_point, best_function_value)
        function_evaluations = function_evaluations + 1

    sort_two_lists_based_on_first_list = lambda list_one, list_two:
    map(list, zip(*sorted(zip(list_one, list_two))))
    fraction_to_replace_bottom_results_by = (N - M) // M
    lower_multiplication_value_for_m = 0.8
    upper_multiplication_value_for_m = 1.2
    function_evaluations = 0
    for _ in range(iterations):
        list_of_function_values, list_of_points =
        sort_two_lists_based_on_first_list(list_of_function_values,
        list_of_points)
        for m_index in range(M):
            current_point = list_of_points[m_index]
            for fraction_index in
            range(fraction_to_replace_bottom_results_by):
                perturbed_index = M + (m_index *
                fraction_to_replace_bottom_results_by) + fraction_index
                perturbed_point = [x *
                uniform(lower_multiplication_value_for_m,
                upper_multiplication_value_for_m) for x
                                   in current_point]
                list_of_function_values[perturbed_index] =
                function_to_optimize(*perturbed_point)
                list_of_points[perturbed_index] = deepcopy(perturbed_point)
                if list_of_function_values[perturbed_index] <
                best_function_value:
                    best_function_value =
                    list_of_function_values[perturbed_index]
                    best_point = deepcopy(list_of_points[perturbed_index])
                best_function_value_at_each_iteration.append(
                best_function_value)
                best_point_at_each_iteration.append(deepcopy(
                best_point))
        print(function_evaluations, best_point_at_each_iteration,
        best_function_value_at_each_iteration)
        function_evaluations = function_evaluations + 1
    return best_point_at_each_iteration,
    best_function_value_at_each_iteration


if __name__ == '__main__':
    _, global_random_search_function_value_part_1, _,
    global_population_search_function_value_part_1 = part_i()
    _, global_random_search_function_value_part_2, _,
    global_population_search_function_value_part_2 = part_ii()
    _, global_random_search_function_value_part_3, _,
    global_population_search_function_value_part_3 = part_iii()
    plot(global_random_search_function_value_part_1,
    global_population_search_function_value_part_1, "batch_size")
    plot(global_random_search_function_value_part_2,
    global_population_search_function_value_part_2, "adam_parameters")
    plot(global_random_search_function_value_part_3,
    global_population_search_function_value_part_3, "epoch_number")
```