```python
import global_random_search
import lib
import numpy as np
import sgd
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D
import pandas as pd

f = {
    "function": lib.f_real,
    "gradient": lib.f_grad,
}

g = {
    "function": lib.g_real,
    "gradient": lib.g_grad,
}


def gradient_descent_constant(step_size=0.0065, start=[0, 0], funcs=f, max_iter=10000, exp="exp/aii-gd-constant.csv"):
    start = np.array(start)
    g = sgd.StochasticGradientDescent()
    g.max_iter(max_iter)
    g.step_size(step_size)
    g.start(start)
    def function_generator():
        while True:
            yield funcs["function"], funcs["gradient"]
    g.function_generator(function_generator())
    g.debug(True)
    g.alg("constant")
    for i in range(max_iter):
        g.step()
        yield {
                "x": g._x_value,
        }

if __name__ == "__main__":
    res = list(gradient_descent_constant(max_iter=1000))
    res = pd.DataFrame(res)
    res["f(x)"] = res["x"].apply(f["function"])
    print(res)

    ps = [{"min": 0, "max": 10}, {"min": 0, "max": 18}]
    grs = global_random_search.a(costf=f["function"], parameters=ps, N=1000)

    plt.figure()

    print(res["f(x)"], len(res["f(x)"]))
    plt.plot(list(range(len(res["f(x)"]))), res["f(x)"], label="gradient descent", color="black")
    costs = grs['stats']['it_best_costs']
    plt.plot(list(range(len(costs))), costs, label="global random search", color="orange")
    plt.title("Global Random Search vs Gradient Descent on $f(x)$")
    custom_lines = [
            Line2D([0], [0], color='black', lw=2),
            Line2D([0], [0], color='orange', lw=2),
            ]
    custom_labels = ['gradient descent', 'rnd search a' ]
    plt.legend(custom_lines, custom_labels)
    plt.yscale('log')
    plt.xlabel('iteration')
    plt.tight_layout()
    plt.savefig('fig/aii-iterations-f.pdf')

    res = list(gradient_descent_constant(max_iter=1000, step_size=0.003, funcs=g))
    res = pd.DataFrame(res)
    res["f(x)"] = res["x"].apply(g["function"])
    print(res)

    ps = [{"min": 0, "max": 10}, {"min": 0, "max": 18}]
    grs = global_random_search.a(costf=g["function"], parameters=ps, N=1000)

    plt.figure()

    print(res["f(x)"], len(res["f(x)"]))
    plt.plot(list(range(len(res["f(x)"]))), res["f(x)"], label="gradient descent", color="black")
    costs = grs['stats']['it_best_costs']
    plt.plot(list(range(len(costs))), costs, label="global random search", color="orange")
    plt.title("Global Random Search vs Gradient Descent on $g(x)$")
    plt.legend()
    plt.yscale('log')
    plt.xlabel('iteration')
    plt.tight_layout()
    plt.savefig('fig/aii-iterations-g.pdf')
```

```python
import global_random_search
import lib
import numpy as np
import sgd
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D
import pandas as pd
import time

f = {
    "function": lib.f_real,
    "gradient": lib.f_grad,
    "dname": "$f(x)$",
    "name": "f",
    "alpha": 0.0065,
}

g = {
    "function": lib.g_real,
    "gradient": lib.g_grad,
    "dname": "$g(x)$",
    "name": "g",
    "alpha": 0.003,
}


def gradient_descent_constant(step_size=0.0065, start=[0, 0], funcs=f, max_time=1):
    start = np.array(start)
    g = sgd.StochasticGradientDescent()
    g.step_size(step_size)
    g.start(start)
    def function_generator():
        while True:
            yield funcs["function"], funcs["gradient"]
    g.function_generator(function_generator())
    g.debug(True)
    g.alg("constant")
    start_time = time.perf_counter()
    current_time = 0
    while current_time < max_time:
        current_time = time.perf_counter() - start_time
        g.step()
        yield {
                "f(x)": g._function(g._x_value),
                "x": g._x_value,
                "time": time.perf_counter() - start_time,
        }

max_time=1
if __name__ == "__main__":
    for funcs in f, g:
        res = list(gradient_descent_constant(max_time=max_time, funcs=funcs, step_size=funcs["alpha"]))
        res = pd.DataFrame(res)

        plt.figure()

        for i in range(3):
            ps = [{"min": 0, "max": 10}, {"min": 0, "max": 18}]
            grs = global_random_search.a(
                costf=funcs["function"], parameters=ps, max_time=max_time)
            costs = grs['stats']['it_best_costs']
            plt.plot(grs['stats']['time'], costs, label="global random search", color="orange")
            print(funcs["name"], "total iterations global random search: ", len(grs['stats']['time']))


        plt.plot(res["time"], res["f(x)"], label="gradient descent", color="black")
        plt.title(f"Global Random Search vs Gradient Descent on {funcs['dname']}")
        custom_lines = [
                Line2D([0], [0], color='black', lw=2),
                Line2D([0], [0], color='orange', lw=2),
                ]
        custom_labels = ['gradient descent', 'rnd search a' ]
        plt.legend(custom_lines, custom_labels)
        plt.yscale('log')
        plt.xlabel("time (seconds)")
        plt.ylabel(funcs['dname'])
        plt.tight_layout()
        plt.savefig(f"fig/aii-time-{funcs['name']}.pdf")
        print(funcs["name"], "total iterations gradient descent: ", len(res))
```

```python
import lib
import numpy as np
import sgd
import matplotlib.pyplot as plt
import pandas as pd

f = {
    "function": lib.f_real,
    "gradient": lib.f_grad,
}

g = {
    "function": lib.g_real,
    "gradient": lib.g_grad,
}


def gradient_descent_constant(step_size=0.0065, start=[0, 0], funcs=f, max_iter=10000, exp="exp/aii-gd-constant.csv"):
    start = np.array(start)
    g = sgd.StochasticGradientDescent()
    g.max_iter(max_iter)
    g.step_size(step_size)
    g.start(start)
    def function_generator():
        while True:
            yield funcs["function"], funcs["gradient"]
    g.function_generator(function_generator())
    g.debug(True)
    g.alg("constant")
    for i in range(max_iter):
        g.step()
        yield {
                "f(x)": g._function(g._x_value),
                "x": g._x_value,
        }

if __name__ == "__main__":

    plt.figure()

    for alpha in [0.004, 0.0035, 0.003, 0.0025]:
        res = list(gradient_descent_constant(max_iter=1000, step_size=alpha, funcs=g))
        res = pd.DataFrame(res)
        plt.plot(list(range(len(res["f(x)"]))),
                 res["f(x)"], label=f"$\\alpha={alpha}$")
    plt.title("Tuning step size for gradient descent on $g(x)$")
    plt.legend()
    plt.yscale('log')
    plt.tight_layout()
    plt.savefig('fig/aii-tune-g.pdf')
```

```python
import even_samples
import cifar_costf
import numpy as np
import keras

a = {
  "best_params": [
    913.957430854217,       # minibatch
    0.00157012586464568, # alpha
    0.6575874719325618,     # beta_1
    0.932720394784433,      # beta_2
    81.32088463431727       # num_epochs
  ],
  "best_cost": 1.8064099550247192
}


b = {
  "best_params": [
    534.4469442210992,       # minibatch
    0.0006231460669478447,  # alpha
    0.7991814790199026,      # beta_1
    0.9007039736299371,      # beta_2
    44.05592177501114        # num_epochs
  ],
  "best_cost": 1.7486121654510498
}

b_mod = {
  "best_params": [
    742.2428227795274,       # minibatch
    0.0009079703308546692,  # alpha
    0.8199336231638713,      # beta_1
    0.6038924210437369,      # beta_2
    64.06011278706069        # num_epochs
  ],
  "best_cost": 1.7933474779129028
}

b_early = [
  629.5247124786772,
  0.0006845628875473787,
  0.7511800761780283,
  0.5624740720563961,
  86.87354850522438
]

versions = [("a", a), ("b", b),("b_mod", b_mod)]

(x_train, y_train), (x_test, y_test)= even_samples.even_sample_categories(50000)
params = np.array(b_early)
cost = cifar_costf.costf(params, (x_train[:1000],y_train[:1000]), (x_train[1000:],y_train[1000:]))
print(cost)
```

```python
import global_random_search
import lib
import numpy as np
import sgd
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D
import pandas as pd
import time
import json

f = {
    "function": lib.f_real,
    "gradient": lib.f_grad,
    "dname": "$f(x)$",
    "name": "f",
    "alpha": 0.0065,
}

g = {
    "function": lib.g_real,
    "gradient": lib.g_grad,
    "dname": "$g(x)$",
    "name": "g",
    "alpha": 0.003,
}


def gradient_descent_constant(step_size=0.0065, start=[0, 0], funcs=f, max_iter=20000):
    start = np.array(start)
    g = sgd.StochasticGradientDescent()
    g.step_size(step_size)
    g.start(start)
    def function_generator():
        while True:
            yield funcs["function"], funcs["gradient"]
    g.function_generator(function_generator())
    g.debug(True)
    g.alg("constant")
    it = 0
    while it < max_iter:
        it += 1
        g.step()
        yield {
                "x": g._x_value,
        }


custom_lines = [
        Line2D([0], [0], color='purple', lw=2),
        Line2D([0], [0], color='blue', lw=2),
        Line2D([0], [0], color='orange', lw=2),
        Line2D([0], [0], color='black', lw=2),
        ]
custom_labels = ['rnd search b_mod', 'rnd search b', 'rnd search a', 'gradient descent']

def vis_results(results):
    def f(x, y):
        return 3 * (x - 5)**4 + 10 * (y - 9)**2
    def g(x, y):
        return np.maximum(x - 5, 0) + 10 * np.abs(y - 9)

    x = np.linspace(0, 10, 400)
    y = np.linspace(0, 18, 400)
    X, Y = np.meshgrid(x, y)
    Z_f = f(X, Y)
    Z_g = g(X, Y)

    fig = plt.figure(figsize=(12, 6))

    axf = fig.add_subplot(1, 2, 1)
    axf.contourf(X, Y, Z_f, levels=30, cmap='viridis')
    axf.set_title('$f(x, y)$')
    axf.set_xlabel('$x$')
    axf.set_ylabel('$y$')

    axg = fig.add_subplot(1, 2, 2)
    axg.contourf(X, Y, Z_g, levels=30, cmap='viridis')
    axg.set_title('$g(x, y)$')
    axg.set_xlabel('$x$')
    axg.set_ylabel('$y$')

    for b_results in results['f']['b']:
        x_coords = [point[0] for point in b_results['stats']['it_best_params']]
        y_coords = [point[1] for point in b_results['stats']['it_best_params']]
        axf.plot(x_coords, y_coords, linestyle='-', label="rndsearch b", color='orange')
    for b_results in results['g']['b']:
        x_coords = [point[0] for point in b_results['stats']['it_best_params']]
        y_coords = [point[1] for point in b_results['stats']['it_best_params']]
        axg.plot(x_coords, y_coords, linestyle='-', label="rndsearch b", color='orange')
    for a_results in results['f']['a']:
        x_coords = [point[0] for point in a_results['stats']['it_best_params']]
        y_coords = [point[1] for point in a_results['stats']['it_best_params']]
        axf.plot(x_coords, y_coords, linestyle='-', label="rndsearch a", color='blue')
    for a_results in results['g']['a']:
        x_coords = [point[0] for point in a_results['stats']['it_best_params']]
        y_coords = [point[1] for point in a_results['stats']['it_best_params']]
        axg.plot(x_coords, y_coords, linestyle='-', label="rndsearch a", color='blue')

    axf.legend()
    axg.legend()
```

```
    plt.tight_layout()
    plt.savefig("fig/bii-contours.pdf")

    return axf, axg

max_time=1
if __name__ == "__main__":
    all_results = {}
    for funcs in f, g:
        res = list(gradient_descent_constant(funcs=funcs, step_size=funcs["alpha"]))


        plt.figure()
        results = {
                "b_mod": [],
                "b": [],
                "a": [],
        }
        res = pd.DataFrame(res)
        res["f(x)"] = res["x"].apply(funcs["function"])

        for i in range(5):
            # ps = [{"min": 0, "max": 10}, {"min": 0, "max": 18}]
            # grs = global_random_search.b_mod(
            #     costf=funcs["function"], iterations=100, parameters=ps, N=1000, M=100, max_time=max_time)
            # costs = grs['stats']['it_best_costs']
            # plt.plot(grs['stats']['time'], costs, label="rnd search b_mod")
            # print(funcs["name"], "total iterations global random search b_mod: ", len(grs['stats']['time']))
            ps = [{"min": 0, "max": 10}, {"min": 0, "max": 18}]
            grs = global_random_search.b_mod(
                costf=funcs["function"], iterations=100, parameters=ps, N=20, M=10)
            costs = grs['stats']['it_best_costs']
            plt.plot(list(range(len(costs))), costs, label="rnd search b_mod", color="purple")
            results["b_mod"].append(grs)
            print(funcs["name"], "total iterations global random search b_mod: ", len(grs['stats']['time']))

            ps = [{"min": 0, "max": 10}, {"min": 0, "max": 18}]
            grs = global_random_search.b(
                costf=funcs["function"], iterations=250, parameters=ps, perturb_pc=0.0001, N=400, M=100)
            costs = grs['stats']['it_best_costs']
            plt.plot(list(range(len(costs))), costs, label="rnd search b", color="blue")
            results["b"].append(grs)
            print(funcs["name"], "total iterations global random search b: ", len(grs['stats']['time']))

            ps = [{"min": 0, "max": 10}, {"min": 0, "max": 18}]
            grs = global_random_search.a(
                costf=funcs["function"], parameters=ps, N=100000)
            costs = grs['stats']['it_best_costs']
            plt.plot(list(range(len(costs))), costs, label="rnd search a", color="orange")
            results["a"].append(grs)
            print(funcs["name"], "total iterations global random search a: ", len(grs['stats']['time']))


        plt.plot(list(range(len(res["f(x)"]))), res["f(x)"], label="gradient descent", color="black")
        plt.title(f"Global Random Search vs Gradient Descent on {funcs['dname']}")
        plt.legend(custom_lines, custom_labels, loc='lower right')
        plt.yscale('log')
        plt.xlabel("function/gradient evals")
        plt.ylabel(funcs['dname'])
        plt.tight_layout()
        plt.savefig(f"fig/bii-evals-{funcs['name']}.pdf")
        print(funcs["name"], "total iterations gradient descent: ", len(res))

        all_results[funcs['name']] = results
```

```python
import global_random_search
import lib
import numpy as np
import sgd
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D
import pandas as pd
import time
import json

f = {
    "function": lib.f_real,
    "gradient": lib.f_grad,
    "dname": "$f(x)$",
    "name": "f",
    "alpha": 0.0065,
}

g = {
    "function": lib.g_real,
    "gradient": lib.g_grad,
    "dname": "$g(x)$",
    "name": "g",
    "alpha": 0.003,
}


def gradient_descent_constant(step_size=0.0065, start=[0, 0], funcs=f, max_time=0.4):
    start = np.array(start)
    g = sgd.StochasticGradientDescent()
    g.step_size(step_size)
    g.start(start)
    def function_generator():
        while True:
            yield funcs["function"], funcs["gradient"]
    g.function_generator(function_generator())
    g.debug(True)
    g.alg("constant")
    start_time = time.perf_counter()
    current_time = 0
    while current_time < max_time:
        current_time = time.perf_counter() - start_time
        g.step()
        yield {
                "f(x)": g._function(g._x_value),
                "x": g._x_value,
                "time": time.perf_counter() - start_time,
        }


custom_lines = [
        Line2D([0], [0], color='purple', lw=2),
        Line2D([0], [0], color='blue', lw=2),
        Line2D([0], [0], color='orange', lw=2),
        Line2D([0], [0], color='black', lw=2),
        ]
custom_labels = ['rnd search b_mod', 'rnd search b', 'rnd search a', 'gradient descent']

def thin(array, step = 30):
    return [array[i] for i in range(0, len(array), step)]

def vis_results(results):
    def f(x, y):
        return 3 * (x - 5)**4 + 10 * (y - 9)**2
    def g(x, y):
        return np.maximum(x - 5, 0) + 10 * np.abs(y - 9)

    x = np.linspace(0, 10, 400)
    y = np.linspace(0, 18, 400)
    X, Y = np.meshgrid(x, y)
    Z_f = f(X, Y)
    Z_g = g(X, Y)

    fig = plt.figure(figsize=(12, 6))

    axf = fig.add_subplot(1, 2, 1)
    axf.contourf(X, Y, Z_f, levels=30, cmap='viridis')
    axf.set_title('$f(x, y)$')
    axf.set_xlabel('$x$')
    axf.set_ylabel('$y$')

    axg = fig.add_subplot(1, 2, 2)
    axg.contourf(X, Y, Z_g, levels=30, cmap='viridis')
    axg.set_title('$g(x, y)$')
    axg.set_xlabel('$x$')
    axg.set_ylabel('$y$')

    cmap = plt.cm.Oranges
    for a_results in results['f']['a']:
        x_coords = thin([point[0] for point in a_results['stats']['it_best_params']])
        y_coords = thin([point[1] for point in a_results['stats']['it_best_params']])
        color = [cmap(i / len(x_coords)) for i in range(len(x_coords))]
        axf.scatter(x_coords, y_coords, linestyle='-', label="rndsearch a", color=color)
    for a_results in results['g']['a']:
        x_coords = thin([point[0] for point in a_results['stats']['it_best_params']])
        y_coords = thin([point[1] for point in a_results['stats']['it_best_params']])
        color = [cmap(i / len(x_coords)) for i in range(len(x_coords))]
        axg.scatter(x_coords, y_coords, linestyle='-', label="rndsearch a", color=color)
    plt.tight_layout()
    plt.savefig("fig/bii-contours-a.pdf")
```

```python

    x = np.linspace(0, 10, 400)
    y = np.linspace(0, 18, 400)
    X, Y = np.meshgrid(x, y)
    Z_f = f(X, Y)
    Z_g = g(X, Y)

    fig = plt.figure(figsize=(12, 6))

    axf = fig.add_subplot(1, 2, 1)
    axf.contourf(X, Y, Z_f, levels=30, cmap='viridis')
    axf.set_title('$f(x, y)$')
    axf.set_xlabel('$x$')
    axf.set_ylabel('$y$')

    axg = fig.add_subplot(1, 2, 2)
    axg.contourf(X, Y, Z_g, levels=30, cmap='viridis')
    axg.set_title('$g(x, y)$')
    axg.set_xlabel('$x$')
    axg.set_ylabel('$y$')

    cmap = plt.cm.Blues
    for b_results in results['f']['b']:
        x_coords = thin([point[0] for point in b_results['stats']['it_best_params']])
        y_coords = thin([point[1] for point in b_results['stats']['it_best_params']])
        color = [cmap(i / len(x_coords)) for i in range(len(x_coords))]
        axf.plot(x_coords, y_coords, linestyle='-', label="rndsearch b", color=color)
    for b_results in results['g']['b']:
        x_coords = thin([point[0] for point in b_results['stats']['it_best_params']])
        y_coords = thin([point[1] for point in b_results['stats']['it_best_params']])
        color = [cmap(i / len(x_coords)) for i in range(len(x_coords))]
        axg.plot(x_coords, y_coords, linestyle='-', label="rndsearch b", color=color)
    plt.tight_layout()
    plt.savefig("fig/bii-contours-b.pdf")

    # axf.legend(custom_lines[1:3], custom_labels[1:3])
    # axg.legend(custom_lines[1:3], custom_labels[1:3])

    return axf, axg

max_time=1
if __name__ == "__main__":
    all_results = {}
    for funcs in f, g:
        res = list(gradient_descent_constant(max_time=max_time, funcs=funcs, step_size=funcs["alpha"]))

        plt.figure()
        results = {
                "b_mod": [],
                "b": [],
                "a": [],
        }
        res = pd.DataFrame(res)

        for i in range(5):
            # ps = [{"min": 0, "max": 10}, {"min": 0, "max": 18}]
            # grs = global_random_search.b_mod(
            #     costf=funcs["function"], iterations=100, parameters=ps, N=1000, M=100, max_time=max_time)
            # costs = grs['stats']['it_best_costs']
            # plt.plot(grs['stats']['time'], costs, label="rnd search b_mod")
            # print(funcs["name"], "total iterations global random search b_mod: ", len(grs['stats']['time']))
            ps = [{"min": 0, "max": 10}, {"min": 0, "max": 18}]
            grs = global_random_search.b_mod(
                costf=funcs["function"], iterations=200, parameters=ps, N=400, M=100, max_time=max_time)
            costs = grs['stats']['it_best_costs']
            plt.plot(grs['stats']['time'], costs, label="rnd search b_mod", color="purple")
            results["b_mod"].append(grs)
            print(funcs["name"], "total iterations global random search b_mod: ", len(grs['stats']['time']))

            ps = [{"min": 0, "max": 10}, {"min": 0, "max": 18}]
            grs = global_random_search.b(
                costf=funcs["function"], iterations=200, parameters=ps, perturb_pc=0.001, N=400, M=100, max_time=max_time
            costs = grs['stats']['it_best_costs']
            plt.plot(grs['stats']['time'], costs, label="rnd search b", color="blue")
            results["b"].append(grs)
            print(funcs["name"], "total iterations global random search b: ", len(grs['stats']['time']))

            ps = [{"min": 0, "max": 10}, {"min": 0, "max": 18}]
            grs = global_random_search.a(
                costf=funcs["function"], parameters=ps, max_time=max_time)
            costs = grs['stats']['it_best_costs']
            plt.plot(grs['stats']['time'], costs, label="rnd search a", color="orange")
            results["a"].append(grs)
            print(funcs["name"], "total iterations global random search a: ", len(grs['stats']['time']))


        plt.plot(res["time"], res["f(x)"], label="gradient descent", color="black")
        plt.title(f"Global Random Search vs Gradient Descent on {funcs['dname']}")
        plt.legend(custom_lines, custom_labels, loc='lower right')
        plt.yscale('log')
        plt.xlabel("time (seconds)")
        plt.ylabel(funcs['dname'])
        plt.tight_layout()
        plt.savefig(f"fig/bii-time-{funcs['name']}.pdf")
        print(funcs["name"], "total iterations gradient descent: ", len(res))

        all_results[funcs['name']] = results

    # with open("data/bii-time.json", "w") as f:
    #     json.dump(all_results, f)
```

```
201:
```

```python
import global_random_search
import lib
import numpy as np
import sgd
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D
import pandas as pd
import time
import json

f = {
    "function": lib.f_real,
    "gradient": lib.f_grad,
    "dname": "$f(x)$",
    "name": "f",
    "alpha": 0.0065,
}

g = {
    "function": lib.g_real,
    "gradient": lib.g_grad,
    "dname": "$g(x)$",
    "name": "g",
    "alpha": 0.003,
}


def gradient_descent_constant(step_size=0.0065, start=[0, 0], funcs=f, max_time=1):
    start = np.array(start)
    g = sgd.StochasticGradientDescent()
    g.step_size(step_size)
    g.start(start)
    def function_generator():
        while True:
            yield funcs["function"], funcs["gradient"]
    g.function_generator(function_generator())
    g.debug(True)
    g.alg("constant")
    start_time = time.perf_counter()
    current_time = 0
    while current_time < max_time:
        current_time = time.perf_counter() - start_time
        g.step()
        yield {
                "f(x)": g._function(g._x_value),
                "x": g._x_value,
                "time": time.perf_counter() - start_time,
        }


custom_lines = [
        Line2D([0], [0], color='purple', lw=2),
        Line2D([0], [0], color='blue', lw=2),
        Line2D([0], [0], color='orange', lw=2),
        Line2D([0], [0], color='black', lw=2),
        ]
custom_labels = ['rnd search b_mod', 'rnd search b', 'rnd search a', 'gradient descent']

def thin(array, step = 30):
    return [array[i] for i in range(0, len(array), step)]

def vis_results(results):
    def f(x, y):
        return 3 * (x - 5)**4 + 10 * (y - 9)**2
    def g(x, y):
        return np.maximum(x - 5, 0) + 10 * np.abs(y - 9)

    x = np.linspace(0, 10, 400)
    y = np.linspace(0, 18, 400)
    X, Y = np.meshgrid(x, y)
    Z_f = f(X, Y)
    Z_g = g(X, Y)

    fig = plt.figure(figsize=(12, 6))

    axf = fig.add_subplot(1, 2, 1)
    axf.contourf(X, Y, Z_f, levels=30, cmap='viridis')
    axf.set_title('$f(x, y)$')
    axf.set_xlabel('$x$')
    axf.set_ylabel('$y$')

    axg = fig.add_subplot(1, 2, 2)
    axg.contourf(X, Y, Z_g, levels=30, cmap='viridis')
    axg.set_title('$g(x, y)$')
    axg.set_xlabel('$x$')
    axg.set_ylabel('$y$')

    cmap = plt.cm.Oranges
    for a_results in results['f']['a']:
        x_coords = thin([point[0] for point in a_results['stats']['it_best_params']])
        y_coords = thin([point[1] for point in a_results['stats']['it_best_params']])
        color = [cmap(i / len(x_coords)) for i in range(len(x_coords))]
        axf.scatter(x_coords, y_coords, linestyle='-', label="rndsearch a", color=color)
    for a_results in results['g']['a']:
        x_coords = thin([point[0] for point in a_results['stats']['it_best_params']])
        y_coords = thin([point[1] for point in a_results['stats']['it_best_params']])
        color = [cmap(i / len(x_coords)) for i in range(len(x_coords))]
        axg.scatter(x_coords, y_coords, linestyle='-', label="rndsearch a", color=color)
    plt.tight_layout()
    plt.savefig("fig/bii-contours-a.pdf")
```

```

    x = np.linspace(0, 10, 400)
    y = np.linspace(0, 18, 400)
    X, Y = np.meshgrid(x, y)
    Z_f = f(X, Y)
    Z_g = g(X, Y)

    fig = plt.figure(figsize=(12, 6))

    axf = fig.add_subplot(1, 2, 1)
    axf.contourf(X, Y, Z_f, levels=30, cmap='viridis')
    axf.set_title('$f(x, y)$')
    axf.set_xlabel('$x$')
    axf.set_ylabel('$y$')

    axg = fig.add_subplot(1, 2, 2)
    axg.contourf(X, Y, Z_g, levels=30, cmap='viridis')
    axg.set_title('$g(x, y)$')
    axg.set_xlabel('$x$')
    axg.set_ylabel('$y$')

    cmap = plt.cm.Blues
    for b_results in results['f']['b']:
        x_coords = thin([point[0] for point in b_results['stats']['it_best_params']])
        y_coords = thin([point[1] for point in b_results['stats']['it_best_params']])
        color = [cmap(i / len(x_coords)) for i in range(len(x_coords))]
        axf.plot(x_coords, y_coords, linestyle='-', label="rndsearch b", color=color)
    for b_results in results['g']['b']:
        x_coords = thin([point[0] for point in b_results['stats']['it_best_params']])
        y_coords = thin([point[1] for point in b_results['stats']['it_best_params']])
        color = [cmap(i / len(x_coords)) for i in range(len(x_coords))]
        axg.plot(x_coords, y_coords, linestyle='-', label="rndsearch b", color=color)
    plt.tight_layout()
    plt.savefig("fig/bii-contours-b.pdf")

    # axf.legend(custom_lines[1:3], custom_labels[1:3])
    # axg.legend(custom_lines[1:3], custom_labels[1:3])

    return axf, axg

max_time=0.1
if __name__ == "__main__":
    all_results = {}
    for funcs in f, g:
        res = list(gradient_descent_constant(max_time=max_time, funcs=funcs, step_size=funcs["alpha"]))

        plt.figure()
        results = {
                "b_mod": [],
                "b": [],
                "a": [],
        }
        res = pd.DataFrame(res)

        for i in range(5):
            # ps = [{"min": 0, "max": 10}, {"min": 0, "max": 18}]
            # grs = global_random_search.b_mod(
            #     costf=funcs["function"], iterations=100, parameters=ps, N=1000, M=100, max_time=max_time)
            # costs = grs['stats']['it_best_costs']
            # plt.plot(grs['stats']['time'], costs, label="rnd search b_mod")
            # print(funcs["name"], "total iterations global random search b_mod: ", len(grs['stats']['time']))
            ps = [{"min": 0, "max": 10}, {"min": 0, "max": 18}]
            grs = global_random_search.b_mod(
                costf=funcs["function"], iterations=8, parameters=ps, N=100, M=50)
            costs = grs['stats']['it_best_costs']
            plt.plot(grs['stats']['time'], costs, label="rnd search b_mod", color="purple")
            results["b_mod"].append(grs)
            print(funcs["name"], "total iterations global random search b_mod: ", len(grs['stats']['time']))

            ps = [{"min": 0, "max": 10}, {"min": 0, "max": 18}]
            grs = global_random_search.b(
                costf=funcs["function"], iterations=8, parameters=ps, perturb_pc=0.01, N=100, M=50)
            costs = grs['stats']['it_best_costs']
            plt.plot(grs['stats']['time'], costs, label="rnd search b", color="blue")
            results["b"].append(grs)
            print(funcs["name"], "total iterations global random search b: ", len(grs['stats']['time']))

            ps = [{"min": 0, "max": 10}, {"min": 0, "max": 18}]
            grs = global_random_search.a(
                costf=funcs["function"], parameters=ps, N=300)
            costs = grs['stats']['it_best_costs']
            plt.plot(grs['stats']['time'], costs, label="rnd search a", color="orange")
            results["a"].append(grs)
            print(funcs["name"], "total iterations global random search a: ", len(grs['stats']['time']))


        plt.plot(res["time"], res["f(x)"], label="gradient descent", color="black")
        plt.title(f"Global Random Search vs Gradient Descent on {funcs['dname']}")
        plt.legend(custom_lines, custom_labels, loc='lower right')
        plt.yscale('log')
        plt.xlabel("time (seconds)")
        plt.ylabel(funcs['dname'])
        plt.tight_layout()
        print(funcs["name"], "total iterations gradient descent: ", len(res))

        all_results[funcs['name']] = results

    with open("data/bii-time.json", "w") as f:
        json.dump(all_results, f)
```

```python
import global_random_search
import lib
import numpy as np
import sgd
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D
import pandas as pd
import time
import json

f = {
    "function": lib.f_real,
    "gradient": lib.f_grad,
    "dname": "$f(x)$",
    "name": "f",
    "alpha": 0.0065,
}

g = {
    "function": lib.g_real,
    "gradient": lib.g_grad,
    "dname": "$g(x)$",
    "name": "g",
    "alpha": 0.003,
}


def gradient_descent_constant(step_size=0.0065, start=[0, 0], funcs=f, max_time=1):
    start = np.array(start)
    g = sgd.StochasticGradientDescent()
    g.step_size(step_size)
    g.start(start)
    def function_generator():
        while True:
            yield funcs["function"], funcs["gradient"]
    g.function_generator(function_generator())
    g.debug(True)
    g.alg("constant")
    start_time = time.perf_counter()
    current_time = 0
    while current_time < max_time:
        current_time = time.perf_counter() - start_time
        g.step()
        yield {
                "f(x)": g._function(g._x_value),
                "x": g._x_value,
                "time": time.perf_counter() - start_time,
        }


custom_lines = [
        Line2D([0], [0], color='purple', lw=2),
        Line2D([0], [0], color='blue', lw=2),
        Line2D([0], [0], color='orange', lw=2),
        Line2D([0], [0], color='black', lw=2),
        ]
custom_labels = ['rnd search b_mod', 'rnd search b', 'rnd search a', 'gradient descent']

def thin(array, step = 5):
    return [array[i] for i in range(0, len(array), step)]

def vis_results(results):
    print("starting vis")
    def f(x, y):
        return 3 * (x - 5)**4 + 10 * (y - 9)**2
    def g(x, y):
        return np.maximum(x - 5, 0) + 10 * np.abs(y - 9)

    x = np.linspace(0, 10, 400)
    y = np.linspace(0, 18, 400)
    X, Y = np.meshgrid(x, y)
    Z_f = f(X, Y)
    Z_g = g(X, Y)

    fig = plt.figure(figsize=(12, 6))

    axf = fig.add_subplot(1, 2, 1)
    axf.contourf(X, Y, Z_f, levels=30, cmap='viridis')
    axf.set_title('$f(x, y)$')
    axf.set_xlabel('$x$')
    axf.set_ylabel('$y$')

    axg = fig.add_subplot(1, 2, 2)
    axg.contourf(X, Y, Z_g, levels=30, cmap='viridis')
    axg.set_title('$g(x, y)$')
    axg.set_xlabel('$x$')
    axg.set_ylabel('$y$')

    cmap = plt.cm.Oranges
    for a_results in results['f']['a'][:1]:
        x_coords, y_coords = zip(*thin(a_results['stats']['it_best_params']))
        # y_coords = thin([point[1] for point in a_results['stats']['it_best_params']])
        color = [cmap(i / len(x_coords)) for i in range(len(x_coords))]
        for x,y,c in zip(x_coords, y_coords, color):
            axf.scatter(x, y, color=c)
    for a_results in results['g']['a'][:1]:
        x_coords = thin([point[0] for point in a_results['stats']['it_best_params']])
        y_coords = thin([point[1] for point in a_results['stats']['it_best_params']])
        color = [cmap(i / len(x_coords)) for i in range(len(x_coords))]
        for x,y,c in zip(x_coords, y_coords, color):
```

```
            axf.scatter(x, y, color=c)
    plt.tight_layout()
    plt.savefig("fig/bii-contours-a.pdf")

    x = np.linspace(0, 10, 400)
    y = np.linspace(0, 18, 400)
    X, Y = np.meshgrid(x, y)
    Z_f = f(X, Y)
    Z_g = g(X, Y)

    fig = plt.figure(figsize=(12, 6))

    axf = fig.add_subplot(1, 2, 1)
    axf.contourf(X, Y, Z_f, levels=30, cmap='viridis')
    axf.set_title('$f(x, y)$')
    axf.set_xlabel('$x$')
    axf.set_ylabel('$y$')

    axg = fig.add_subplot(1, 2, 2)
    axg.contourf(X, Y, Z_g, levels=30, cmap='viridis')
    axg.set_title('$g(x, y)$')
    axg.set_xlabel('$x$')
    axg.set_ylabel('$y$')

    cmap = plt.cm.Blues
    for b_results in results['f']['b'][:1]:
        x_coords = thin([point[0] for point in b_results['stats']['it_best_params']])
        y_coords = thin([point[1] for point in b_results['stats']['it_best_params']])
        color = [cmap(i / len(x_coords)) for i in range(len(x_coords))]
        for x,y,c in zip(x_coords, y_coords, color):
            axf.scatter(x, y, color=c)
    for b_results in results['g']['b'][:1]:
        x_coords = thin([point[0] for point in b_results['stats']['it_best_params']])
        y_coords = thin([point[1] for point in b_results['stats']['it_best_params']])
        color = [cmap(i / len(x_coords)) for i in range(len(x_coords))]
        for x,y,c in zip(x_coords, y_coords, color):
            axf.scatter(x, y, color=c)
    plt.tight_layout()
    plt.savefig("fig/bii-contours-b.pdf")

    # axf.legend(custom_lines[1:3], custom_labels[1:3])
    # axg.legend(custom_lines[1:3], custom_labels[1:3])

    return axf, axg


if __name__ == "__main__":
    results = None
    with open("data/bii-time.json", "r") as f:
        results = json.load(f)
    vis_results(results)
```

```python
import global_random_search
import lib
import numpy as np
import sgd
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D
import pandas as pd
import time
import json

f = {
    "function": lib.f_real,
    "gradient": lib.f_grad,
    "dname": "$f(x)$",
    "name": "f",
    "alpha": 0.0065,
}

g = {
    "function": lib.g_real,
    "gradient": lib.g_grad,
    "dname": "$g(x)$",
    "name": "g",
    "alpha": 0.003,
}


def gradient_descent_constant(step_size=0.0065, start=[0, 0], funcs=f, max_time=1):
    start = np.array(start)
    g = sgd.StochasticGradientDescent()
    g.step_size(step_size)
    g.start(start)
    def function_generator():
        while True:
            yield funcs["function"], funcs["gradient"]
    g.function_generator(function_generator())
    g.debug(True)
    g.alg("constant")
    start_time = time.perf_counter()
    current_time = 0
    while current_time < max_time:
        current_time = time.perf_counter() - start_time
        g.step()
        yield {
                "f(x)": g._function(g._x_value),
                "x": g._x_value,
                "time": time.perf_counter() - start_time,
        }


custom_lines = [
        Line2D([0], [0], color='purple', lw=2),
        Line2D([0], [0], color='blue', lw=2),
        Line2D([0], [0], color='orange', lw=2),
        Line2D([0], [0], color='black', lw=2),
        ]
custom_labels = ['rnd search b_mod', 'rnd search b', 'rnd search a', 'gradient descent']

def thin(array, step = 1):
    return [array[i] for i in range(0, len(array), step)]

def vis_results(results, args):
    print("starting vis")
    params = thin(results)
    def f(x, y):
        return 3 * (x - 5)**4 + 10 * (y - 9)**2
    def g(x, y):
        return np.maximum(x - 5, 0) + 10 * np.abs(y - 9)

    x = np.linspace(0, 10, 400)
    y = np.linspace(0, 18, 400)
    X, Y = np.meshgrid(x, y)
    Z_f = f(X, Y) if args.function == "f" else g(X,Y)
    fig = plt.figure(figsize=(4, 4))

    axf = fig.add_subplot(1, 1, 1)
    axf.contourf(X, Y, Z_f, levels=30, cmap='viridis')
    axf.set_title(args.title)
    axf.set_xlabel('$x$')
    axf.set_ylabel('$y$')
    x_coords, y_coords = zip(*thin(params, step=args.thin))
    # y_coords = thin([point[1] for point in params], step=args.thin)
    cmap = plt.cm.Blues
    color = [cmap(i / len(x_coords)) for i in range(len(x_coords))]
    for x,y,c in zip(x_coords, y_coords, color):
        print(".", end="", flush=True)
        axf.scatter(x, y, s=3, color=c)
    plt.tight_layout()


if __name__ == "__main__":
    import argparse
    ap = argparse.ArgumentParser()
    ap.add_argument("-i", type=str)
    ap.add_argument("-o", type=str)
    ap.add_argument("--title", type=str)
    ap.add_argument("--function", type=str)
    ap.add_argument("--thin", type=int, default=20)
    args = ap.parse_args()
    results = None
```

```
    with open(args.i, "r") as f:
        results = json.load(f)
    vis_results(results, args)
    print("saving fig")
    plt.savefig(args.o)
```

```python
import matplotlib.pyplot as plt
import numpy as np
import sys


def f(x, y):
    return 3 * (x - 5)**4 + 10 * (y - 9)**2


def g(x, y):
    return np.maximum(x - 5, 0) + 10 * np.abs(y - 9)


def main(outfile):
    x = np.linspace(0, 10, 400)
    y = np.linspace(0, 18, 400)
    X, Y = np.meshgrid(x, y)
    Z_f = f(X, Y)
    Z_g = g(X, Y)

    fig = plt.figure(figsize=(12, 6))

    ax = fig.add_subplot(1, 2, 1, projection='3d')
    ax.plot_surface(X, Y, Z_f, cmap='viridis')
    ax.set_title('$f(x, y)$')
    ax.set_xlabel('$x$')
    ax.set_ylabel('$y$')
    ax.set_zlabel('$f(x, y)$')

    ax = fig.add_subplot(1, 2, 2, projection='3d')
    ax.plot_surface(X, Y, Z_g, cmap='magma')
    ax.set_title('$g(x, y)$')
    ax.set_xlabel('$x$')
    ax.set_ylabel('$y$')
    ax.set_zlabel('$g(x, y)$')

    plt.savefig(outfile)
    plt.show()

def main_contour(outfile):
    x = np.linspace(0, 10, 400)
    y = np.linspace(0, 18, 400)
    X, Y = np.meshgrid(x, y)
    Z_f = f(X, Y)
    Z_g = g(X, Y)

    fig = plt.figure(figsize=(12, 6))

    ax = fig.add_subplot(1, 2, 1)
    ax.contourf(X, Y, Z_f, levels=30, cmap='viridis')
    ax.set_title('$f(x, y)$')
    ax.set_xlabel('$x$')
    ax.set_ylabel('$y$')
    # ax.set_zlabel('$f(x, y)$')

    ax = fig.add_subplot(1, 2, 2)
    ax.contourf(X, Y, Z_g, levels=30, cmap='viridis')
    ax.set_title('$g(x, y)$')
    ax.set_xlabel('$x$')
    ax.set_ylabel('$y$')
    # ax.set_zlabel('$g(x, y)$')

    plt.savefig(outfile)
    plt.show()


if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: python script.py <output_file>")
        sys.exit(1)

    outfile = sys.argv[1]
    main_contour(outfile)
```

```
import global_random_search
import lib
import numpy as np
import sgd
import matplotlib.pyplot as plt
import pandas as pd
import time
import cifar_costf
import json
import argparse
import c_vis
import even_samples
import math
import cps
ps = cps.ps

ap = argparse.ArgumentParser()
# ap.add_argument("--exp", type=str, required=True)
ap.add_argument("--M", type=int, required=True)
ap.add_argument("--N", type=int, required=True)
ap.add_argument("--n", type=int, required=True)
ap.add_argument("--iterations", type=int, required=True)
args = ap.parse_args()

f = {
    "function": lib.f_real,
    "gradient": lib.f_grad,
    "dname": "$f(x)$",
    "name": "f",
    "alpha": 0.0065,
}

g = {
    "function": lib.g_real,
    "gradient": lib.g_grad,
    "dname": "$g(x)$",
    "name": "g",
    "alpha": 0.003,
}


def gradient_descent_constant(step_size=0.0065, start=[0, 0], funcs=f, max_time=1):
    start = np.array(start)
    g = sgd.StochasticGradientDescent()
    g.step_size(step_size)
    g.start(start)
    def function_generator():
        while True:
            yield funcs["function"], funcs["gradient"]
    g.function_generator(function_generator())
    g.debug(True)
    g.alg("constant")
    start_time = time.time()
    current_time = 0
    while current_time < max_time:
        current_time = time.time() - start_time
        g.step()
        yield {
                "f(x)": g._function(g._x_value),
                "x": g._x_value,
                "time": time.time() - start_time,
        }

if __name__ == "__main__":
    train, test = even_samples.even_sample_categories(math.floor(args.n))

    def costf(x):
        return cifar_costf.costf(x, train, test)

    grs = global_random_search.a(
        debug=True,
        costf=costf, parameters=ps, N=args.N*args.iterations)

    fname = f"data/c-a-N{args.N*args.iterations}.json"
    save = {
        'results': grs,
        'param-limits': ps,
        'args': vars(args),
        'name': None,
    }
    with open(fname, "w") as f:
        json.dump(save, f)
```

```python
import global_random_search
import lib
import numpy as np
import sgd
import matplotlib.pyplot as plt
import pandas as pd
import time
import cifar_costf
import json
import argparse
import c_vis
import even_samples
import math
import cps
ps = cps.ps

ap = argparse.ArgumentParser()
# ap.add_argument("--exp", type=str, required=True)
ap.add_argument("--M", type=int, required=True)
ap.add_argument("--N", type=int, required=True)
ap.add_argument("--n", type=int, required=True)
ap.add_argument("--iterations", type=int, required=True)
args = ap.parse_args()

f = {
    "function": lib.f_real,
    "gradient": lib.f_grad,
    "dname": "$f(x)$",
    "name": "f",
    "alpha": 0.0065,
}

g = {
    "function": lib.g_real,
    "gradient": lib.g_grad,
    "dname": "$g(x)$",
    "name": "g",
    "alpha": 0.003,
}


def gradient_descent_constant(step_size=0.0065, start=[0, 0], funcs=f, max_time=1):
    start = np.array(start)
    g = sgd.StochasticGradientDescent()
    g.step_size(step_size)
    g.start(start)
    def function_generator():
        while True:
            yield funcs["function"], funcs["gradient"]
    g.function_generator(function_generator())
    g.debug(True)
    g.alg("constant")
    start_time = time.time()
    current_time = 0
    while current_time < max_time:
        current_time = time.time() - start_time
        g.step()
        yield {
                "f(x)": g._function(g._x_value),
                "x": g._x_value,
                "time": time.time() - start_time,
        }

if __name__ == "__main__":
    train, test = even_samples.even_sample_categories(math.floor(args.n))

    def costf(x):
        return cifar_costf.costf(x, train, test)

    grs = global_random_search.b_mod(
        debug=True,
        costf=costf, parameters=ps, N=args.N, M=args.M, iterations=args.iterations)

    fname = f"data/c-b_mod-N{args.N}-M{args.M}-n{args.n}-it{args.iterations}.json"
    save = {
        'results': grs,
        'param-limits': ps,
        'args': vars(args),
        'name': None,
    }
    with open(fname, "w") as f:
        json.dump(save, f)
```

```python
import global_random_search
import lib
import numpy as np
import sgd
import matplotlib.pyplot as plt
import pandas as pd
import time
import cifar_costf
import json
import argparse
import c_vis
import even_samples
import math
import cps
ps = cps.ps

ap = argparse.ArgumentParser()
# ap.add_argument("--exp", type=str, required=True)
ap.add_argument("--M", type=int, required=True)
ap.add_argument("--N", type=int, required=True)
ap.add_argument("--n", type=int, required=True)
ap.add_argument("--iterations", type=int, required=True)
args = ap.parse_args()

f = {
    "function": lib.f_real,
    "gradient": lib.f_grad,
    "dname": "$f(x)$",
    "name": "f",
    "alpha": 0.0065,
}

g = {
    "function": lib.g_real,
    "gradient": lib.g_grad,
    "dname": "$g(x)$",
    "name": "g",
    "alpha": 0.003,
}


def gradient_descent_constant(step_size=0.0065, start=[0, 0], funcs=f, max_time=1):
    start = np.array(start)
    g = sgd.StochasticGradientDescent()
    g.step_size(step_size)
    g.start(start)
    def function_generator():
        while True:
            yield funcs["function"], funcs["gradient"]
    g.function_generator(function_generator())
    g.debug(True)
    g.alg("constant")
    start_time = time.time()
    current_time = 0
    while current_time < max_time:
        current_time = time.time() - start_time
        g.step()
        yield {
                "f(x)": g._function(g._x_value),
                "x": g._x_value,
                "time": time.time() - start_time,
        }

if __name__ == "__main__":

    train, test = even_samples.even_sample_categories(math.floor(args.n))

    def costf(x):
        return cifar_costf.costf(x, train, test)

    grs = global_random_search.b(
        debug=True,
        costf=costf, parameters=ps, N=args.N, M=args.M, iterations=args.iterations)

    fname = f"data/c-b-N{args.N}-M{args.M}-n{args.n}-it{args.iterations}.json"
    save = {
        'results': grs,
        'param-limits': ps,
        'args': vars(args),
        'name': None,
    }
    with open(fname, "w") as f:
        json.dump(save, f)
```

```python
import tensorflow as tf
import numpy as np
import math
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 sklearn.metrics import confusion_matrix, classification_report
from sklearn.utils import shuffle
import matplotlib.pyplot as plt
plt.rc('font', size=18)
plt.rcParams['figure.constrained_layout.use'] = True
import sys
from sklearn.metrics import roc_auc_score
import multiprocessing
import even_samples

#num_cores = int(multiprocessing.cpu_count()/2)
#tf.config.threading.set_inter_op_parallelism_threads(num_cores)
#tf.config.threading.set_intra_op_parallelism_threads(num_cores)

# Model / data parameters
num_classes = 10
input_shape = (32, 32, 3)

(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()

# Scale images to the [0, 1] range
print("orig x_train shape:", x_train.shape)
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255

# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

def params2dict(x):
    minibatch, alpha, beta1, beta2, epochs = x
    return {
            'minibatch': minibatch,
            'alpha': alpha,
            'beta1': beta1,
            'beta2': beta2,
            'epochs': epochs,
    }

def compute_auc_loss(model, x_test, y_test):
    # Get predicted probabilities for each class
    preds = model.predict(x_test)

    # Compute AUC score for each class
    auc_scores = []
    for class_idx in range(num_classes):
        auc_score = roc_auc_score(y_test[:, class_idx], preds[:, class_idx])
        auc_scores.append(auc_score)

    return auc_scores

def compute_macro_auc(model, x_test, y_test):
    # Get predicted probabilities for each class
    preds = model.predict(x_test)

    # Compute AUC score for each class
    auc_scores = []
    for class_idx in range(num_classes):
        auc_score = roc_auc_score(y_test[:, class_idx], preds[:, class_idx])
        auc_scores.append(auc_score)

    # Compute macro-average AUC
    macro_auc = sum(auc_scores) / len(auc_scores)

    return macro_auc

def costf(x, train, test):
    x_train, y_train = train
    x_test, y_test = test
    print("params: ", params2dict(x))
    print("training data:", len(x_train))
    x_train_sub = x_train
    y_train_sub = y_train
    minibatch, alpha, beta1, beta2, epochs = x
    minibatch = math.floor(minibatch)
    epochs = math.floor(epochs)
    model = keras.Sequential()
    model.add(Conv2D(16, (3,3), padding='same', input_shape=x_train_sub.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)))

    adam_optimizer = keras.optimizers.Adam(learning_rate=alpha, beta_1=beta1, beta_2=beta2)
    model.compile(loss="categorical_crossentropy", optimizer=adam_optimizer, metrics=["accuracy"])
    model.summary()
    batch_size = minibatch
    print(x_train_sub.shape, y_train_sub.shape)
    history = model.fit(x_train_sub, y_train_sub, batch_size=batch_size, epochs=epochs, validation_split=0.1)
    test_loss, _ = model.evaluate(x_test, y_test, verbose=0)
    return test_loss
```

```

if __name__ == "__main__":
    print(costf(np.array([5, 0.0001, 0.9, 0.999, 3]), n=50000))
```

```
ps = [
        {"min": 1,             "max": 1024},        # minibatch
        {"min": 0.0000000001,  "max": 0.01},        # alpha
        {"min": 0.5,           "max": 1},           # beta1
        {"min": 0.5,           "max": 1},           # beta2
        {"min": 10,            "max": 100},         # epochs
]
```

```python
import global_random_search
import lib
import numpy as np
import sgd
import matplotlib.pyplot as plt
import pandas as pd
import time
import cifar_costf
import json
import argparse

ap = argparse.ArgumentParser()
# ap.add_argument("--exp", type=str, required=True)
ap.add_argument("--M", type=int, required=True)
ap.add_argument("--N", type=int, required=True)
ap.add_argument("--n", type=int, required=True)
ap.add_argument("--iterations", type=int, required=True)
args = ap.parse_args()

f = {
    "function": lib.f_real,
    "gradient": lib.f_grad,
    "dname": "$f(x)$",
    "name": "f",
    "alpha": 0.0065,
}

g = {
    "function": lib.g_real,
    "gradient": lib.g_grad,
    "dname": "$g(x)$",
    "name": "g",
    "alpha": 0.003,
}


def gradient_descent_constant(step_size=0.0065, start=[0, 0], funcs=f, max_time=1):
    start = np.array(start)
    g = sgd.StochasticGradientDescent()
    g.step_size(step_size)
    g.start(start)
    def function_generator():
        while True:
            yield funcs["function"], funcs["gradient"]
    g.function_generator(function_generator())
    g.debug(True)
    g.alg("constant")
    start_time = time.time()
    current_time = 0
    while current_time < max_time:
        current_time = time.time() - start_time
        g.step()
        yield {
                "f(x)": g._function(g._x_value),
                "x": g._x_value,
                "time": time.time() - start_time,
        }

if __name__ == "__main__":
    ps = [
            {"min": 1, "max": args.n},            # minibatch
            {"min": 0.0000000001, "max": 5},     # alpha
            {"min": 0, "max": 1},                # beta1
            {"min": 0, "max": 1},                # beta2
            {"min": 1, "max": 40},               # epochs
    ]

    def costf(x):
        return cifar_costf.costf(x, n=args.n)

    grs = global_random_search.b_mod(
        debug=True,
        costf=costf, parameters=ps, N=args.N, M=args.M, iterations=args.iterations)
    costs = grs['stats']['it_best_costs']

    print(grs)
    timei = time.time()
    fname = f"data/c-N{args.N}-M{args.M}-n{args.n}-it{args.iterations}"
    save = {
        'results': grs,
        'param-limits': ps,
        'args': vars(args),
        'name': None,
    }
    with open(f"{fname}.json", "w") as f:
        json.dump(grs, f)
```

```python
import json
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D

def fix_costs_monotonic(costs):
    costs_monotonic = []
    best_cost = costs[0]
    for cost in costs:
        if cost <= best_cost:
            best_cost = cost
        costs_monotonic.append(best_cost)
    return costs_monotonic

def visualize_stats_time_vs_it_best_costs(json_file, **kwargs):
    with open(json_file, 'r') as f:
        results = json.load(f)
    print(results)
    time = results['results']['stats']['time']
    it_best_costs = results['results']['stats']['it_best_costs']
    it_best_costs = fix_costs_monotonic(it_best_costs)
    plt.plot(list(range(len(it_best_costs))), it_best_costs, linestyle='-', **kwargs)

# c-a-N30.json  c-b_mod-N20-M3-n1000-it3.json  c-b_mod-N20-M3-n5000-it3.json  c-b_mod-N20-M3-n500-it3.json  c-b-N10-M4-n500-it3.json
if __name__ == "__main__":
    plt.figure(figsize=(8, 6))
    plt.ylim(1.5, 2.2)
    visualize_stats_time_vs_it_best_costs('data/c-a-N99.json', label='rnd search a $(N=99,n=1000)$', color='orange')
    visualize_stats_time_vs_it_best_costs('data/c-b_mod-N33-M10-n1000-it3.json', label='rnd search b_mod $(N=33,M=10,n=1000)$', color='purple')
    visualize_stats_time_vs_it_best_costs('data/c-b-N33-M10-n1000-it3.json', label='rnd search b $(N=33,M=10,n=1000)$', color='blue')
    # visualize_stats_time_vs_it_best_costs('data/c-a-N100-M-1-n1000-it3.json', label='rnd search b')

    plt.axhline(y=1.8646, color='red', linestyle='--')
    plt.xlabel('function evaluations')
    plt.ylabel('logistic loss on test ($n=10000$)')
    custom_lines = [
            Line2D([0], [0], color='blue', lw=2),
            Line2D([0], [0], color='orange', lw=2),
            Line2D([0], [0], color='purple', lw=2),
            Line2D([0], [0], color='red', lw=2, linestyle='--'),
            ]
    custom_labels = ['rnd search b $(N=33,M=10,n=1000)$', 'rnd search a $(N=99,n=1000)$', 'rnd search b_mod $(N=33,M=10,n=1000)$', 'baseline']
    plt.legend(custom_lines, custom_labels)
    plt.savefig('fig/c.pdf')
```

```python
import numpy as np
import keras
from sklearn.model_selection import StratifiedShuffleSplit

num_classes = 10

def even_sample_categories(n):
    (x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
    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)

    return (x_train[1:n], y_train[1:n]), (x_test, y_test)

if __name__ == "__main__":
    # Example usage
    X_train_even, y_train_even = even_sample_categories(num_samples_per_class=10)
    print(X_train_even, y_train_even)
    print(x_train.shape, y_train.shape)
    print(X_train_even.shape, y_train_even.shape)
```

```python
import even_samples
import cifar_costf
import numpy as np
import keras

a = {
  "best_params": [
    913.957430854217,       # minibatch
    0.0015701252586464568, # alpha
    0.6575874719325618,     # beta_1
    0.932720394784433,      # beta_2
    81.32088463431727       # num_epochs
  ],
  "best_cost": 1.8064099550247192
}


b = {
  "best_params": [
    534.4469442210992,       # minibatch
    0.0006231460669478447,  # alpha
    0.7991814790199026,      # beta_1
    0.9007039736299371,      # beta_2
    44.05592177501114        # num_epochs
  ],
  "best_cost": 1.7486121654510498
}

b_mod = {
  "best_params": [
    742.2428227795274,       # minibatch
    0.0009079703308546692,  # alpha
    0.8199336231638713,      # beta_1
    0.6038924210437369,      # beta_2
    64.06011278706069        # num_epochs
  ],
  "best_cost": 1.7933474779129028
}

versions = [("a", a), ("b", b),("b_mod", b_mod)]

(x_train, y_train), (x_test, y_test)= even_samples.even_sample_categories(50000)
with open("final-check.txt", "w") as f:
    for name, version in versions:
        params = np.array(version["best_params"])
        cost = cifar_costf.costf(params, (x_train[:1000],y_train[:1000]), (x_train[1000:],y_train[1000:]))
        version["test_cost"] = cost
        version["name"] = name
        print(version, file=f)
```

```python
class GradientDescent():
    def __init__(self):
        self._max_iter = 1000
        self._debug = False
        self._converged = lambda x1, x2: False
        self._epsilon = 0.0001
        self._dimension = None
        self._beta = 0
        self._algorithm = None
        self._iteration = None
        self._function = None
        self._sum = None
        self._x_value = None
        self._step_coeff = None
        self._converged_value = None
        self._grad_value = None
        self._m = None
        self._v = None
        self._adam_grad = None
        self._beta = None
        self._beta2 = None
        self._step_size = None
        self._z = None
        self._f_star = None

    def step_size(self, a):
        self._step_size = a
        return self

    def beta(self, b):
        self._beta = b
        return self

    def beta2(self, b):
        self._beta2 = b
        return self

    def epsilon(self, e):
        self._epsilon = e
        return self

    def function(self, f, function_name=None, dimension=None):
        self._function = f
        self.function_name = function_name
        self._dimension = dimension
        return self

    def sym_function(self, function, function_name=None):
        self.function_name = function_name
        self._dimension = len(function.free_symbols)
        def fn(x):
            return apply_sym(x, function)

        diffs = [function.diff(var) for var in function.free_symbols]

        def grad(x):
            return np.array([
                apply_sym(x, diff) for diff in diffs])

        self._function = fn
        self._gradient = grad
        return self

    def gradient(self, g):
        self._gradient = g
        return self

    def max_iter(self, m):
        self._max_iter = m
        return self

    def start(self, s):
        self._start = s
        self._x_value = s
        return self

    def debug(self, d):
        self._debug = d
        return self

    def converged(self, c):
        self._converged = c
        return self

    def set_iterate(self, f):
        self.iterate = functools.partial(f, self)
        return self

    def algorithm(self, alg):
        self._algorithm = alg
        if self._algorithm == "rmsprop":
            import rmsprop
            self.set_iterate(rmsprop.iterate)
        elif self._algorithm == "adam":
            import adam
            self.set_iterate(adam.iterate)
        elif self._algorithm == "heavy_ball":
            import heavy_ball
            self.set_iterate(heavy_ball.iterate)
        else:
```

```
            raise Exception("Unknown algorithm:" + alg)
        return self

    def state_dict(self):
        print(self._function(self._x_value))
        return {
            "alg": self._algorithm,
            "function_name": self.function_name,
            "iteration": self._iteration,
            "step_coeff": self._step_coeff,
            "adam_grad": self._adam_grad,
            "f(x)": self._function(self._x_value),
            "epsilon": self._epsilon,
            "converged": self._converged_value,
            "gradient": self._grad_value,
            "m": self._m,
            "v": self._v,
            "beta1": self._beta,
            "beta2": self._beta2,
            "alpha": self._step_size,
            "sum": self._sum,
            "z": self._z,
            **{"x" + str(i): self._x_value[i] for i in range(len(self._x_value))},
        }

    def run2csv(self, fname, summarise=True):
        import pandas as pd
        iterations = list(self.iterate())
        df = pd.DataFrame(iterations)
        df.to_csv(fname)
        if(summarise):
            with open(fname + ".summary", "w") as f:
                print(f"iterations: {len(df)}", file=f)
                print(f"start: {df['x0'][0]} {df['x1'][0]}", file=f)
                print(f"final: {df['x0'][len(df) - 1]} {df['x1'][len(df) - 1]}", file=f)
```

```python
def perturb(x, alpha=0.1):
    # generate random point in the unit hypersphere
    ndim = x.shape[0]
    random_point = np.random.normal(size=ndim)
    random_point /= np.linalg.norm(random_point)

    # scale and translate the point to fit the specified center and radius
    perturbed_point = x + alpha * x * random_point

    return perturbed_point

def b_mod(costf=None, parameters=None, alpha=0.1, iterations=2, N=100, M=10, max_time=-1, debug=False):
    if costf is None:
        raise Exception("costf is a required kwarg")
    if parameters is None:
        raise Exception("parameters is a required kwarg")
    it_best_costs = []
    start_time = time.time()
    best_cost = None
    best_params = None
    times = []
    if max_time > 0:
        N = -1
    current_time = 0
    params = []
    costs = []
    it = 0
    while (it < N or N < 0) and (current_time < max_time or max_time < 0):
        it += 1
        ps = gen_params(parameters)
        cost = costf(ps)
        params.append(ps)
        costs.append(cost)
        if best_cost is None or cost < best_cost:
            best_params = ps
            best_cost = cost
        it_best_costs.append(best_cost)
        current_time = time.time() - start_time
        times.append(current_time)
        if debug:
            print("parameters:", ps, end="\t")
            print("cost:", cost, end="\t")
            print("best cost:", best_cost)
    bests = best_m(params, costs, M=M)

    for i in range(iterations):
        params = []
        costs = []
        it = 0
        while it < N and (current_time < max_time or max_time < 0):
            it += 1
            choice = random.choice(bests)
            new_params = perturb(choice, alpha=alpha)
            new_cost = costf(choice)
            params.append(new_params)
            costs.append(new_cost)
            if new_cost < best_cost:
                best_cost = new_cost
                best_params = new_params

    return {
        "results": {
            "best_params": best_params,
            "best_cost": best_cost,
        },
        "stats": {
            "it_best_costs": it_best_costs,
            "time": times,
        }
    }
```

```python
import numpy as np
import lib
import time
import random

def gen_params(parameters):
    p = np.zeros(len(parameters), dtype=np.float64)
    for i, par in enumerate(parameters):
        mini = par["min"]
        maxi = par["max"]
        p[i] = np.random.uniform(mini, maxi)
    return p


def a(costf=None, parameters=None, N=100, max_time=-1, debug=False):
    if costf is None:
        raise Exception("costf is a required kwarg")
    if parameters is None:
        raise Exception("parameters is a required kwarg")
    best_params = None
    best_cost = None
    it_best_costs = []
    it_best_params = []
    it_params = []
    start_time = time.perf_counter()
    times = []
    it = 0
    if max_time > 0:
        N = -1
    current_time = 0
    while (it < N or N < 0) and (current_time < max_time or max_time < 0):
        it += 1
        ps = gen_params(parameters)
        cost = costf(ps)
        if best_cost is None or np.isnan(best_cost) or cost < best_cost:
            best_params = ps
            best_cost = cost
        it_best_costs.append(best_cost)
        it_best_params.append(best_params)
        it_params.append(ps)
        current_time = time.perf_counter() - start_time
        times.append(current_time)
        if debug:
            print("parameters:", ps, end="\t")
            print("cost:", cost, end="\t")
            print("best cost:", best_cost)
    return {
        "results": {
            "best_params": best_params.tolist(),
            "best_cost": best_cost,
        },
        "stats": {
            "it_best_costs": it_best_costs,
            "it_best_params": list(map(lambda x: x.tolist(), it_best_params)),
            "it_params": list(map(lambda x: x.tolist(), it_params)),
            "time": times,
        }
    }

def best_m(params, costs, M=10, unzip=True):
    bests = sorted(zip(params, costs), key=lambda x: x[1])
    best_M = bests[:M]
    if unzip:
        return list(zip(*best_M))
    return best_M

def bests2parameters(bests):
    params = bests[0]
    p1 = params[0]
    ps = []
    for i in range(len(p1)):
        param_values = list(map(lambda x: x[i], params))
        ps.append({
            "min": min(param_values),
            "max": max(param_values),
            })
    return ps

def b_mod(costf=None, parameters=None, iterations=2, N=100, M=10, max_time=-1, debug=False):
    if costf is None:
        raise Exception("costf is a required kwarg")
    if parameters is None:
        raise Exception("parameters is a required kwarg")
    it_best_costs = []
    it_best_params = []
    it_params = []
    start_time = time.perf_counter()
    best_cost = None
    best_params = None
    times = []
    current_time = 0
    iteration_results = []
    for i in range(iterations):
        if debug:
            print("iteration: ", i)
        if max_time > 0 and current_time > max_time:
            break
        params = []
        costs = []
        it = 0
```

```
        while it < N:
            it += 1
            ps = gen_params(parameters)
            cost = costf(ps)
            params.append(ps)
            costs.append(cost)
            if best_cost is None or cost < best_cost:
                best_params = ps
                best_cost = cost
            it_best_costs.append(best_cost)
            it_best_params.append(best_params)
            it_params.append(ps)
            current_time = time.perf_counter() - start_time
            times.append(current_time)
            if debug:
                print("parameters:", ps, end="\t")
                print("cost:", cost, end="\t")
                print("best cost:", best_cost)
        bests = best_m(params, costs, M=M)
        parameters = bests2parameters(bests)
        iteration_results.append({
                "M": M,
                "best_m_params": list(map(lambda x: x.tolist(), bests[0])),
                "best_m_costs": bests[1],
                "best_params": best_params.tolist(),
                "best_cost": best_cost,
        })
    return {
        "results": {
            "best_params": best_params.tolist(),
            "best_cost": best_cost,
        },
        "stats": {
            "it_best_costs": it_best_costs,
            "it_best_params": list(map(lambda x: x.tolist(), it_best_params)),
            "it_params": list(map(lambda x: x.tolist(), it_params)),
            "time": times,
        },
        "iteration_results": iteration_results,
    }

def perturb(x, alpha=1.1):
    # generate random point in the unit hypersphere
    print(x, type(x))
    ndim = x.shape[0]
    random_point = np.random.normal(size=ndim)
    random_point /= np.linalg.norm(random_point)

    # scale and translate the point to fit the specified center and radius
    perturbed_point = x + alpha *  random_point

    return perturbed_point

def perturbn(x, alpha):
    """
    Randomly perturbs each element of x by adding noise from [-alpha, alpha].

    Args:
    - x (list or numpy array): The input array.
    - alpha (float): The range of noise to add. The noise is drawn from the interval [-alpha, alpha].

    Returns:
    - list: The perturbed array.
    """
    perturbed_x = [elem + random.uniform(-alpha, alpha) for elem in x]
    return perturbed_x

def perturb_percent(x, percent=0.1, ps=None):
    if ps is None:
        raise Exception("require parameters ps")
    out = np.zeros(x.shape)
    for i in range(len(x)):
        span = ps[i]['max'] - ps[i]['min']
        low = -span*percent
        high = span*percent
        r = np.random.uniform(low=low, high=high, size=1)
        out[i] = x[i] + r
        out[i] = max(ps[i]['min'], out[i])
        out[i] = min(ps[i]['max'], out[i])
    return out


def b(costf=None, parameters=None, perturb_pc=0.1, iterations=2, N=100, M=10, max_time=-1, debug=False):
    if costf is None:
        raise Exception("costf is a required kwarg")
    if parameters is None:
        raise Exception("parameters is a required kwarg")
    it_best_costs = []
    it_best_params = []
    it_params = []
    start_time = time.perf_counter()
    best_cost = None
    best_params = None
    times = []
    current_time = 0
    params = []
    costs = []
    it = 0
    while (it < N or N < 0) and (current_time < max_time or max_time < 0):
        it += 1
```

```
        ps = gen_params(parameters)
        cost = costf(ps)
        params.append(ps)
        costs.append(cost)
        if best_cost is None or cost < best_cost:
            best_params = ps
            best_cost = cost
        it_best_costs.append(best_cost)
        it_best_params.append(best_params)
        it_params.append(ps)
        current_time = time.perf_counter() - start_time
        times.append(current_time)
        if debug:
            print("parameters:", ps, end="\t")
            print("cost:", cost, end="\t")
            print("best cost:", best_cost)
    bests = best_m(params, costs, M=M)

    for i in range(iterations - 1):
        params = []
        costs = []
        it = 0
        while it < N and (current_time < max_time or max_time < 0):
            it += 1
            choice = random.choice(bests[0])
            new_params = perturb_percent(choice, percent=perturb_pc, ps=parameters)
            new_cost = costf(choice)
            params.append(new_params)
            it_params.append(new_params)
            costs.append(new_cost)
            if new_cost < best_cost:
                best_cost = new_cost
                best_params = new_params
            it_best_costs.append(best_cost)
            it_best_params.append(best_params)
            current_time = time.perf_counter() - start_time
            times.append(current_time)
        bests = best_m(params + list(bests[0]), costs + list(bests[1]), M=M)

    return {
        "results": {
            "best_params": best_params.tolist(),
            "best_cost": best_cost,
        },
        "stats": {
            "it_best_costs": it_best_costs,
            "it_best_params": list(map(lambda x: x.tolist(), it_best_params)),
            "it_params": list(map(lambda x: x.tolist(), it_params)),
            "time": times,
        }
    }


if __name__ == "__main__":
    # costf = lib.f_real
    # parameters=[{"min": 0, "max": 20},{"min": 0, "max": 20}]
    # N=1000
    # out = b(costf=costf, iterations=30, parameters=parameters, N=N, M=300, debug=False, alpha=5)
    # print(out['results']['best_params'])

    x = np.array([0, 0])
    print(x, perturb_percent(x, percent=0.5, ps=[{'min': 0, 'max': 20},{'min': 0, 'max': 20}]))
```

```
import sympy as sp
import numpy as np
import functools

x, y = sp.symbols('x y', real=True)
f = 3 * (x - 5)**4 + (10 * ((y - 9)**2))
g = sp.Max(x - 5, 0) + (10 * sp.Abs(y - 9))
relu = sp.Max(x,0)

def f_real(xv):
    return 3 * (xv[0] - 5)**4 + 10 * (xv[1] - 9)**2

f_diff_x = f.diff(x)
f_diff_y = f.diff(y)
def f_grad(xv):
    return np.array([
        f_diff_x.subs(x, xv[0]).subs(y, xv[1]),
        f_diff_y.subs(x, xv[0]).subs(y, xv[1]),
    ])

g_diff_x = f.diff(x)
g_diff_y = f.diff(y)
def g_grad(xv):
    return np.array([
        g_diff_x.subs(x, xv[0]).subs(y, xv[1]),
        g_diff_y.subs(x, xv[0]).subs(y, xv[1]),
    ])

def g_real(xv):
    return np.maximum(xv[0] - 5, 0) + 10 * np.abs(xv[1] - 9)


def apply_sym(x, f):
    for x_sym, x_val in zip(f.free_symbols, x):
        f = f.subs(x_sym, x_val)
    return f

config = {
    "f": {
        "sym": f,
        "real": f_real,
        "name": "f",
    },
    "g": {
        "sym": g,
        "real": g_real,
        "name": "g",
    },
    "relu": {
        "sym": relu,
        "real": lambda x: max(x, 0),
        "name": "relu",
    }
}

class GradientDescent():
    def __init__(self):
        self._max_iter = 1000
        self._debug = False
        self._converged = lambda x1, x2: False
        self._epsilon = 0.0001
        self._dimension = None
        self._beta = 0
        self._algorithm = None
        self._iteration = None
        self._function = None
        self._sum = None
        self._x_value = None
        self._step_coeff = None
        self._converged_value = None
        self._grad_value = None
        self._m = None
        self._v = None
        self._adam_grad = None
        self._beta = None
        self._beta2 = None
        self._step_size = None
        self._z = None
        self._f_star = None

    def step_size(self, a):
        self._step_size = a
        return self

    def beta(self, b):
        self._beta = b
        return self

    def beta2(self, b):
        self._beta2 = b
        return self

    def epsilon(self, e):
        self._epsilon = e
        return self

    def function(self, f, function_name=None, dimension=None):
        self._function = f
        self.function_name = function_name
        self._dimension = dimension
```

```python
        return self

    def sym_function(self, function, function_name=None):
        self.function_name = function_name
        self._dimension = len(function.free_symbols)
        def fn(x):
            return apply_sym(x, function)

        diffs = [function.diff(var) for var in function.free_symbols]

        def grad(x):
            return np.array([
                apply_sym(x, diff) for diff in diffs])

        self._function = fn
        self._gradient = grad
        return self

    def gradient(self, g):
        self._gradient = g
        return self

    def max_iter(self, m):
        self._max_iter = m
        return self

    def start(self, s):
        self._start = s
        return self

    def debug(self, d):
        self._debug = d
        return self

    def converged(self, c):
        self._converged = c
        return self

    def set_iterate(self, f):
        self.iterate = functools.partial(f, self)
        return self

    def algorithm(self, alg):
        self._algorithm = alg
        if self._algorithm == "rmsprop":
            import rmsprop
            self.set_iterate(rmsprop.iterate)
        elif self._algorithm == "adam":
            import adam
            self.set_iterate(adam.iterate)
        elif self._algorithm == "heavy_ball":
            import heavy_ball
            self.set_iterate(heavy_ball.iterate)
        else:
            raise Exception("Unknown algorithm:" + alg)
        return self

    def state_dict(self):
        print(self._function(self._x_value))
        return {
            "alg": self._algorithm,
            "function_name": self.function_name,
            "iteration": self._iteration,
            "step_coeff": self._step_coeff,
            "adam_grad": self._adam_grad,
            "f(x)": self._function(self._x_value),
            "epsilon": self._epsilon,
            "converged": self._converged_value,
            "gradient": self._grad_value,
            "m": self._m,
            "v": self._v,
            "beta1": self._beta,
            "beta2": self._beta2,
            "alpha": self._step_size,
            "sum": self._sum,
            "z": self._z,
            **{"x" + str(i): self._x_value[i] for i in range(len(self._x_value))},
        }

    def run2csv(self, fname, summarise=True):
        import pandas as pd
        iterations = list(self.iterate())
        df = pd.DataFrame(iterations)
        df.to_csv(fname)
        if(summarise):
            with open(fname + ".summary", "w") as f:
                print(f"iterations: {len(df)}", file=f)
                print(f"start: {df['x0'][0]} {df['x1'][0]}", file=f)
                print(f"final: {df['x0'][len(df) - 1]} {df['x1'][len(df) - 1]}", file=f)


if __name__ == "__main__":
    print(f.diff(x), f.diff(y))
    print(g.diff(x), g.diff(y))
```

```python
import numpy as np

def iterate(self):
    self._x_value = self._start
    self._old_x_value = None
    self._f_star = 0
    self._iteration = 0
    self._converged_value = False
    self._grad_value = self._gradient(self._x_value)

    yield self.state_dict()

    while not self._converged_value:
        if self._max_iter > 0 and self._iteration > self._max_iter:
            break
        numerator = self._function(self._x_value) - self._f_star
        self._grad_value = self._gradient(self._x_value)
        denominator = np.dot(self._grad_value, self._grad_value) # sum of element-wise products
        self._old_x_value = self._x_value
        step = numerator/denominator
        self._x_value = self._x_value - step * self._grad_value
        self._converged_value = self._converged(self._x_value, self._old_x_value)
        yield self.state_dict()
```

```python
import even_samples
import cifar_costf
import numpy as np
import keras

# (x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
# x_train = x_train.astype("float32") / 255
# x_test = x_test.astype("float32") / 255
# num_classes = 10
# y_train = keras.utils.to_categorical(y_train, num_classes)
# y_test = keras.utils.to_categorical(y_test, num_classes)

(x_train, y_train), (x_test, y_test)= even_samples.even_sample_categories(1000)
print(cifar_costf.costf(np.array([128, 0.001, 0.9, 0.999, 40]), (x_train,y_train), (x_test,y_test)))
```

```python
import numpy as np
import functools
import gd
import week6

class StochasticGradientDescent(gd.GradientDescent):
    def __init__(self):
        self._iteration = 0
        self._max_iter = 1000
        self._converged = lambda x1, x2: False
        self._epsilon = 0.0001
        self._f_star = 0
        self._debug = False
        self._beta = 0
        self._function_generator = None
        self._dimension = None
        self._algorithm = None
        self._function = None
        self._sum = None
        self._x_value = None
        self._old_x_value = None
        self._step_coeff = None
        self._converged_value = None
        self._grad_value = None
        self._m = None
        self._v = None
        self._adam_grad = None
        self._beta = None
        self._beta2 = None
        self._step_size = None
        self._z = None

    def adam_step(self):
        self._function, self._gradient = next(self._function_generator)
        if self._function == "finished":
            return False  # did not complet step
        self._grad_value = self._gradient(self._x_value)
        self._m = self._beta * self._m + (1-self._beta)*self._grad_value
        # grad_value * grad_value gives element-wise product of np array
        self._v = self._beta2 * self._v + (1-self._beta2) * (self._grad_value*self._grad_value)
        self._old_x_value = self._x_value
        self._iteration += 1
        m_hat = self._m / (1-(self._beta ** self._iteration))
        v_hat = np.array(self._v / (1-(self._beta2 ** self._iteration)))
        v_hat_aug = v_hat**(0.5) + self._epsilon
        self._adam_grad = m_hat / v_hat_aug
        self._x_value = self._x_value - self._step_size * self._adam_grad
        return True

    def polyak_step(self):
        self._function, self._gradient = next(self._function_generator)
        if self._function == "finished":
            return False  # did not complet step
        self._iteration += 1
        numerator = self._function(self._x_value) - self._f_star
        self._grad_value = self._gradient(self._x_value)
        denominator = np.dot(self._grad_value, self._grad_value) # sum of element-wise products
        if denominator == 0.0:
            # do nothing this step (hope for non-zero on next mini-batch)
            return False
        self._old_x_value = self._x_value
        step = numerator/denominator
        self._x_value = self._x_value - step * self._grad_value
        self._converged_value = self._converged(self._x_value, self._old_x_value)
        return True  # completed step

    def constant_step(self):
        self._function, self._gradient = next(self._function_generator)
        if self._function == "finished":
            return False  # did not complete step
        self._iteration += 1
        self._grad_value = self._gradient(self._x_value)
        self._old_x_value = self._x_value
        self._x_value = self._x_value - self._step_size * self._grad_value
        return True  # completed step

    def rmsprop_step(self):
        self._function, self._gradient = next(self._function_generator)
        if self._function == "finished":
            return False
        self._iteration += 1
        self._grad_value = self._gradient(self._x_value)
        self._old_x_value = self._x_value
        self._x_value = self._x_value - self._alpha_n * self._grad_value
        self._sum = self._beta * self._sum + (1-self._beta) * (self._grad_value**2)
        self._alpha_n = self._step_size / (self._sum**0.5+self._epsilon)
        self._step_coeff = self._alpha_n
        return True


    def heavy_ball_step(self):
        self._function, self._gradient = next(self._function_generator)
        if self._function == "finished":
            return False
        self._iteration += 1
        self._grad_value = self._gradient(self._x_value)
        self._old_x_value = self._x_value
        self._z = self._beta * self._z + self._step_size * self._grad_value
        self._x_value = self._x_value - self._z
        return True
```

```python

    # pass a function which generates the function to be evaluated,
    # e.g. with different minibatches at each iteration
    def function_generator(self, fg):
        self._function_generator = fg
        return self

    def alg(self, a):
        if a == "constant":
            self.step = self.constant_step
        elif a == "polyak":
            self.step = self.polyak_step
        elif a == "rmsprop":
            self.step = self.rmsprop_step
            if self._step_size is None:
                raise Exception("Need step_size to initialize rmsprop")
            if self._x_value is None:
                raise Exception("Need start/x_value to initialize rmsprop")
            self._sum = np.zeros(self._x_value.shape)
            self._alpha_n = np.zeros(self._x_value.shape)
            self._alpha_n.fill(self._step_size)
        elif a == "adam":
            self.step = self.adam_step
            if self._x_value is None:
                raise Exception("Need start/x_value to initialize rmsprop")
            self._m = np.zeros(self._x_value.shape, dtype=np.float64)
            self._v = np.zeros(self._x_value.shape, dtype=np.float64)
        elif a == "heavy_ball":
            self.step = self.heavy_ball_step
            self._z = 0
        else:
            raise Exception(f"Alg {a} NYI")
        self.function_name = a
        return self

    def f_star(self, f_st):
        self._f_star = f_st
        return self

```

```python
import tensorflow as tf
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 sklearn.metrics import confusion_matrix, classification_report
from sklearn.utils import shuffle
import matplotlib.pyplot as plt
plt.rc('font', size=18)
plt.rcParams['figure.constrained_layout.use'] = True
import sys
import even_samples
import keras
import math

(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
# x_train, y_train = even_samples.even_sample_categories(num_samples_per_class=math.floor(100/10))
n=5000
x_train = x_train[1:n]; y_train=y_train[1:n]
num_classes = 10
input_shape = (32, 32, 3)

# Scale images to the [0, 1] range
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255

# convert class vectors to binary class matrices
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)))
model.compile(loss="categorical_crossentropy", optimizer='adam', metrics=["accuracy"])
model.summary()

batch_size = 128
epochs = 20
history = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_split=0.1)
model.save("cifar.model")
plt.subplot(211)
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'val'], loc='upper left')
plt.subplot(212)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss'); plt.xlabel('epoch')
plt.legend(['train', 'val'], loc='upper left')
plt.show()
```

```python
import matplotlib.pyplot as plt
import numpy as np
import sys


def f(x, y):
    return 3 * (x - 5)**4 + 10 * (y - 9)**2


def g(x, y):
    return np.maximum(x - 5, 0) + 10 * np.abs(y - 9)


def main(outfile):
    x = np.linspace(0, 10, 400)
    y = np.linspace(0, 18, 400)
    X, Y = np.meshgrid(x, y)
    Z_f = f(X, Y)
    Z_g = g(X, Y)

    fig = plt.figure(figsize=(12, 6))

    ax = fig.add_subplot(1, 2, 1, projection='3d')
    ax.plot_surface(X, Y, Z_f, cmap='viridis')
    ax.set_title('$f(x, y)$')
    ax.set_xlabel('$x$')
    ax.set_ylabel('$y$')
    ax.set_zlabel('$f(x, y)$')

    ax = fig.add_subplot(1, 2, 2, projection='3d')
    ax.plot_surface(X, Y, Z_g, cmap='magma')
    ax.set_title('$g(x, y)$')
    ax.set_xlabel('$x$')
    ax.set_ylabel('$y$')
    ax.set_zlabel('$g(x, y)$')

    plt.savefig(outfile)
    plt.show()

def main_contour(outfile):
    x = np.linspace(0, 10, 400)
    y = np.linspace(0, 18, 400)
    X, Y = np.meshgrid(x, y)
    Z_f = f(X, Y)
    Z_g = g(X, Y)

    fig = plt.figure(figsize=(12, 6))

    ax = fig.add_subplot(1, 2, 1)
    ax.contourf(X, Y, Z_f, levels=30, cmap='viridis')
    ax.set_title('$f(x, y)$')
    ax.set_xlabel('$x$')
    ax.set_ylabel('$y$')
    # ax.set_zlabel('$f(x, y)$')

    ax = fig.add_subplot(1, 2, 2)
    ax.contourf(X, Y, Z_g, levels=30, cmap='viridis')
    ax.set_title('$g(x, y)$')
    ax.set_xlabel('$x$')
    ax.set_ylabel('$y$')
    # ax.set_zlabel('$g(x, y)$')

    plt.savefig(outfile)
    plt.show()


if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: python script.py <output_file>")
        sys.exit(1)

    outfile = sys.argv[1]
    main_contour(outfile)

```

```python
import numpy as np
import sympy as sp

current_minibatch = None

def generate_trainingdata(m=25):
    return np.array([0,0]) + 0.25 * np.random.randn(m,2)


def f(x, minibatch):
    # loss function sum_{w in training data} f(x,w)
    y = 0
    count = 0
    for w in minibatch:
        z = x - w - 1
        left = 10 * (z[0]**2+z[1]**2)
        right = (z[0]+2)**2+(z[1]+4)**2
        y = y + min(left, right)
        count = count + 1
    return y/count


def gradient_function_fd(minibatch, epsilon=10**(-15)):
    def gradient_fd(x):
        dydx1 = (f(x + np.array([epsilon, 0]), minibatch) - f(x, minibatch)) / epsilon
        dydx2 = (f(x + np.array([0, epsilon]), minibatch) - f(x, minibatch)) / epsilon
        return np.array([dydx1, dydx2])
    return gradient_fd

def sympy_loss(minibatch):
    x1, x2 = sp.symbols('x1 x2', real=True)
    function = 0
    for w in minibatch:
        z1 = x1 - w[0] - 1
        z2 = x2 - w[1] - 1
        left = 10 * (z1**2 + z2**2)
        right = (z1 + 2)**2 + (z2 + 4)**2
        function = sp.Min(left, right) + function
    function = function / len(minibatch)
    return function

def gradient_function(minibatch):
    function = sympy_loss(minibatch)
    def gradient(x):
        dydx1 = function.diff(x1)
        dydx2 = function.diff(x2)
        return np.array([
            dydx1.subs(x1, x[0]).subs(x2, x[1]),
            dydx2.subs(x1, x[0]).subs(x2, x[1]),
        ])

    return gradient


def loss(x, w):
    z = x - w - 1
    left = 10 * (z[0]**2+z[1]**2)
    right = (z[0]+2)**2+(z[1]+4)**2
    return min(left, right)


def f_clear(x, minibatch):
    return sum(loss(x, w) for w in minibatch) / len(minibatch)


def generate_minibatches(T, N=5, seed=42, shuffle=True,):
    global current_minibatch
    if shuffle:
        T = T.copy()
        if seed:
            np.random.seed(seed)
        np.random.shuffle(T)
    num_rows = T.shape[0]
    i = 0

    minibatch = np.zeros((N, T.shape[1]), T.dtype)
    while True:
        for j in range(N):
            minibatch[j] = T[i % num_rows]
            i += 1
        if shuffle and i >= num_rows:
            # begin next epoch
            np.random.shuffle(T)
            i = 0
        current_minibatch = minibatch
        yield minibatch


def generate_optimisation_functions(batch, minibatch_size=5, finite_difference=True, **kwargs):
    minibatch_generator = generate_minibatches(
        batch, N=minibatch_size, **kwargs)
    for minibatch in minibatch_generator:
        def optim_func(x):
            return f_clear(x, minibatch)
        gradf = None
        if finite_difference:
            gradf = gradient_function_fd(minibatch)
        else:
            gradf = gradient_function(minibatch)
        yield (optim_func, gradf)
```

```
    yield "finished"


if __name__ == "__main__":
    import os
    os.makedirs("data", exist_ok=True)
    T = generate_trainingdata()
    import pandas as pd
    df = pd.DataFrame(T)
    df.to_csv("data/T.csv", index=False)

    x = np.array([3, 3])
```

```python
import tensorflow as tf
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 sklearn.metrics import confusion_matrix, classification_report
from sklearn.utils import shuffle
import matplotlib.pyplot as plt
plt.rc('font', size=18)
plt.rcParams['figure.constrained_layout.use'] = True
import sys

# Model / data parameters
num_classes = 10
input_shape = (32, 32, 3)

# the data, split between train and test sets
(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_test=x_test[1:500]; y_test=y_test[1:500]

# Scale images to the [0, 1] range
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255
print("orig x_train shape:", x_train.shape)

# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

use_saved_model = False
if use_saved_model:
        model = keras.models.load_model("cifar.model")
else:
        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)))
        model.compile(loss="categorical_crossentropy", optimizer='adam', metrics=["accuracy"])
        model.summary()

        batch_size = 128
        epochs = 20
        history = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_split=0.1)
        model.save("cifar.model")
        plt.subplot(211)
        plt.plot(history.history['accuracy'])
        plt.plot(history.history['val_accuracy'])
        plt.title('model accuracy')
        plt.ylabel('accuracy')
        plt.xlabel('epoch')
        plt.legend(['train', 'val'], loc='upper left')
        plt.subplot(212)
        plt.plot(history.history['loss'])
        plt.plot(history.history['val_loss'])
        plt.title('model loss')
        plt.ylabel('loss'); plt.xlabel('epoch')
        plt.legend(['train', 'val'], loc='upper left')
        plt.show()

preds = model.predict(x_train)
y_pred = np.argmax(preds, axis=1)
y_train1 = np.argmax(y_train, axis=1)
print(classification_report(y_train1, y_pred))
print(confusion_matrix(y_train1,y_pred))

preds = model.predict(x_test)
y_pred = np.argmax(preds, axis=1)
y_test1 = np.argmax(y_test, axis=1)
print(classification_report(y_test1, y_pred))
print(confusion_matrix(y_test1,y_pred))
```