

```
1: import global_random_search
2: import lib
3: import numpy as np
4: import sg
5: import matplotlib.pyplot as plt
6: from matplotlib.lines import Line2D
7: import pandas as pd
8:
9: f = {
10:     "function": lib.f_real,
11:     "gradient": lib.f_grad,
12: }
13:
14: g = {
15:     "function": lib.g_real,
16:     "gradient": lib.g_grad,
17: }
18:
19:
20: def gradient_descent_constant(step_size=0.0065, start=[0, 0], funcs=f, max_iter=10000, exp="exp/aii-gd-constant.csv"):
21:     start = np.array(start)
22:     g = sg.StochasticGradientDescent()
23:     g.max_iter(max_iter)
24:     g.step_size(step_size)
25:     g.start(start)
26:     def function_generator():
27:         while True:
28:             yield funcs["function"], funcs["gradient"]
29:     g.function_generator(function_generator())
30:     g.debug(True)
31:     g.alg("constant")
32:     for i in range(max_iter):
33:         g.step()
34:         yield {
35:             "x": g._x_value,
36:         }
37:
38: if __name__ == "__main__":
39:     res = list(gradient_descent_constant(max_iter=1000))
40:     res = pd.DataFrame(res)
41:     res["f(x)"] = res["x"].apply(f["function"])
42:     print(res)
43:
44:     ps = [{"min": 0, "max": 10}, {"min": 0, "max": 18}]
45:     grs = global_random_search.a(costf=f["function"], parameters=ps, N=1000)
46:
47:     plt.figure()
48:
49:     print(res["f(x)"], len(res["f(x)"]))
50:     plt.plot(list(range(len(res["f(x)"]))), res["f(x)"], label="gradient descent", color="black")
51:     costs = grs['stats']['it_best_costs']
52:     plt.plot(list(range(len(costs))), costs, label="global random search", color="orange")
53:     plt.title("Global Random Search vs Gradient Descent on $f(x)$")
54:     custom_lines = [
55:         Line2D([0], [0], color='black', lw=2),
56:         Line2D([0], [0], color='orange', lw=2),
57:     ]
58:     custom_labels = ['gradient descent', 'rnd search a']
59:     plt.legend(custom_lines, custom_labels)
60:     plt.yscale('log')
61:     plt.xlabel('iteration')
62:     plt.tight_layout()
63:     plt.savefig('fig/aii-iterations-f.pdf')
64:
65:     res = list(gradient_descent_constant(max_iter=1000, step_size=0.003, funcs=g))
66:     res = pd.DataFrame(res)
67:     res["f(x)"] = res["x"].apply(g["function"])
68:     print(res)
69:
70:     ps = [{"min": 0, "max": 10}, {"min": 0, "max": 18}]
71:     grs = global_random_search.a(costf=g["function"], parameters=ps, N=1000)
72:
73:     plt.figure()
74:
75:     print(res["f(x)"], len(res["f(x)"]))
76:     plt.plot(list(range(len(res["f(x)"]))), res["f(x)"], label="gradient descent", color="black")
77:     costs = grs['stats']['it_best_costs']
78:     plt.plot(list(range(len(costs))), costs, label="global random search", color="orange")
79:     plt.title("Global Random Search vs Gradient Descent on $g(x)$")
80:     plt.legend()
81:     plt.yscale('log')
82:     plt.xlabel('iteration')
83:     plt.tight_layout()
84:     plt.savefig('fig/aii-iterations-g.pdf')
```

```
1: import global_random_search
2: import lib
3: import numpy as np
4: import sgd
5: import matplotlib.pyplot as plt
6: from matplotlib.lines import Line2D
7: import pandas as pd
8: import time
9:
10: f = {
11:     "function": lib.f_real,
12:     "gradient": lib.f_grad,
13:     "dname": "$f(x)$",
14:     "name": "f",
15:     "alpha": 0.0065,
16: }
17:
18: g = {
19:     "function": lib.g_real,
20:     "gradient": lib.g_grad,
21:     "dname": "$g(x)$",
22:     "name": "g",
23:     "alpha": 0.003,
24: }
25:
26:
27: def gradient_descent_constant(step_size=0.0065, start=[0, 0], funcs=f, max_time=1):
28:     start = np.array(start)
29:     g = sgd.StochasticGradientDescent()
30:     g.step_size(step_size)
31:     g.start(start)
32:     def function_generator():
33:         while True:
34:             yield funcs["function"], funcs["gradient"]
35:     g.function_generator(function_generator())
36:     g.debug(True)
37:     g.alg("constant")
38:     start_time = time.perf_counter()
39:     current_time = 0
40:     while current_time < max_time:
41:         current_time = time.perf_counter() - start_time
42:         g.step()
43:         yield {
44:             "f(x)": g._function(g._x_value),
45:             "x": g._x_value,
46:             "time": time.perf_counter() - start_time,
47:         }
48:
49: max_time=1
50: if __name__ == "__main__":
51:     for funcs in f, g:
52:         res = list(gradient_descent_constant(max_time=max_time, funcs=funcs, step_size=funcs["alpha"]))
53:         res = pd.DataFrame(res)
54:
55:         plt.figure()
56:
57:         for i in range(3):
58:             ps = [{"min": 0, "max": 10}, {"min": 0, "max": 18}]
59:             grs = global_random_search.a(
60:                 costf=funcs["function"], parameters=ps, max_time=max_time)
61:             costs = grs['stats']['it_best_costs']
62:             plt.plot(grs['stats']['time'], costs, label="global random search", color="orange")
63:             print(funcs["name"], "total iterations global random search: ", len(grs['stats']['time']))
64:
65:
66:         plt.plot(res["time"], res["f(x)"], label="gradient descent", color="black")
67:         plt.title(f"Global Random Search vs Gradient Descent on {funcs['dname']}")
68:         custom_lines = [
69:             Line2D([0], [0], color='black', lw=2),
70:             Line2D([0], [0], color='orange', lw=2),
71:         ]
72:         custom_labels = ['gradient descent', 'rnd search a']
73:         plt.legend(custom_lines, custom_labels)
74:         plt.yscale('log')
75:         plt.xlabel("time (seconds)")
76:         plt.ylabel(funcs['dname'])
77:         plt.tight_layout()
78:         plt.savefig(f"fig/aii-time-{funcs['name']}.pdf")
79:         print(funcs["name"], "total iterations gradient descent: ", len(res))
```

```
1: import lib
2: import numpy as np
3: import sgd
4: import matplotlib.pyplot as plt
5: import pandas as pd
6:
7: f = {
8:     "function": lib.f_real,
9:     "gradient": lib.f_grad,
10: }
11:
12: g = {
13:     "function": lib.g_real,
14:     "gradient": lib.g_grad,
15: }
16:
17:
18: def gradient_descent_constant(step_size=0.0065, start=[0, 0], funcs=f, max_iter=10000, exp="exp/aii-gd-constant.csv"):
19:     start = np.array(start)
20:     g = sgd.StochasticGradientDescent()
21:     g.max_iter(max_iter)
22:     g.step_size(step_size)
23:     g.start(start)
24:     def function_generator():
25:         while True:
26:             yield funcs["function"], funcs["gradient"]
27:     g.function_generator(function_generator())
28:     g.debug(True)
29:     g.alg("constant")
30:     for i in range(max_iter):
31:         g.step()
32:         yield {
33:             "f(x)": g._function(g._x_value),
34:             "x": g._x_value,
35:         }
36:
37: if __name__ == "__main__":
38:
39:     plt.figure()
40:
41:     for alpha in [0.004, 0.0035, 0.003, 0.0025]:
42:         res = list(gradient_descent_constant(max_iter=1000, step_size=alpha, funcs=g))
43:         res = pd.DataFrame(res)
44:         plt.plot(list(range(len(res["f(x)"]))),
45:                  res["f(x)"], label=f"$\\alpha={alpha}$")
46:     plt.title("Tuning step size for gradient descent on $g(x)$")
47:     plt.legend()
48:     plt.yscale('log')
49:     plt.tight_layout()
50:     plt.savefig('fig/aii-tune-g.pdf')
```

```
1: import even_samples
2: import cifar_costf
3: import numpy as np
4: import keras
5:
6: a = {
7:     "best_params": [
8:         913.957430854217,      # minibatch
9:         0.0015701252586464568, # alpha
10:        0.6575874719325618,    # beta_1
11:        0.932720394784433,     # beta_2
12:        81.32088463431727      # num_epochs
13:    ],
14:     "best_cost": 1.8064099550247192
15: }
16:
17:
18: b = {
19:     "best_params": [
20:         534.4469442210992,    # minibatch
21:         0.0006231460669478447, # alpha
22:         0.7991814790199026,    # beta_1
23:         0.9007039736299371,    # beta_2
24:         44.05592177501114      # num_epochs
25:    ],
26:     "best_cost": 1.7486121654510498
27: }
28:
29: b_mod = {
30:     "best_params": [
31:         742.2428227795274,    # minibatch
32:         0.0009079703308546692, # alpha
33:         0.8199336231638713,    # beta_1
34:         0.6038924210437369,    # beta_2
35:         64.06011278706069      # num_epochs
36:    ],
37:     "best_cost": 1.7933474779129028
38: }
39:
40: b_early = [
41:     629.5247124786772,
42:     0.0006845628875473787,
43:     0.7511800761780283,
44:     0.5624740720563961,
45:     86.87354850522438
46: ]
47:
48: versions = [("a", a), ("b", b), ("b_mod", b_mod)]
49:
50: (x_train, y_train), (x_test, y_test)= even_samples.even_sample_categories(50000)
51: params = np.array(b_early)
52: cost = cifar_costf.costf(params, (x_train[:1000],y_train[:1000]), (x_train[1000:],y_train[1000:]))
53: print(cost)
```

```
1: import global_random_search
2: import lib
3: import numpy as np
4: import sgd
5: import matplotlib.pyplot as plt
6: from matplotlib.lines import Line2D
7: import pandas as pd
8: import time
9: import json
10:
11: f = {
12:     "function": lib.f_real,
13:     "gradient": lib.f_grad,
14:     "dname": "$f(x)$",
15:     "name": "f",
16:     "alpha": 0.0065,
17: }
18:
19: g = {
20:     "function": lib.g_real,
21:     "gradient": lib.g_grad,
22:     "dname": "$g(x)$",
23:     "name": "g",
24:     "alpha": 0.003,
25: }
26:
27:
28: def gradient_descent_constant(step_size=0.0065, start=[0, 0], funcs=f, max_iter=20000):
29:     start = np.array(start)
30:     g = sgd.StochasticGradientDescent()
31:     g.step_size(step_size)
32:     g.start(start)
33:     def function_generator():
34:         while True:
35:             yield funcs["function"], funcs["gradient"]
36:     g.function_generator(function_generator())
37:     g.debug(True)
38:     g.alg("constant")
39:     it = 0
40:     while it < max_iter:
41:         it += 1
42:         g.step()
43:         yield {
44:             "x": g._x_value,
45:         }
46:
47:
48: custom_lines = [
49:     Line2D([0], [0], color='purple', lw=2),
50:     Line2D([0], [0], color='blue', lw=2),
51:     Line2D([0], [0], color='orange', lw=2),
52:     Line2D([0], [0], color='black', lw=2),
53: ]
54: custom_labels = ['rnd search b_mod', 'rnd search b', 'rnd search a', 'gradient descent']
55:
56: def vis_results(results):
57:     def f(x, y):
58:         return 3 * (x - 5)**4 + 10 * (y - 9)**2
59:     def g(x, y):
60:         return np.maximum(x - 5, 0) + 10 * np.abs(y - 9)
61:
62:     x = np.linspace(0, 10, 400)
63:     y = np.linspace(0, 18, 400)
64:     X, Y = np.meshgrid(x, y)
65:     Z_f = f(X, Y)
66:     Z_g = g(X, Y)
67:
68:     fig = plt.figure(figsize=(12, 6))
69:
70:     axf = fig.add_subplot(1, 2, 1)
71:     axf.contourf(X, Y, Z_f, levels=30, cmap='viridis')
72:     axf.set_title('$f(x, y)$')
73:     axf.set_xlabel('$x$')
74:     axf.set_ylabel('$y$')
75:
76:     axg = fig.add_subplot(1, 2, 2)
77:     axg.contourf(X, Y, Z_g, levels=30, cmap='viridis')
78:     axg.set_title('$g(x, y)$')
79:     axg.set_xlabel('$x$')
80:     axg.set_ylabel('$y$')
81:
82:     for b_results in results['f']['b']:
83:         x_coords = [point[0] for point in b_results['stats']['it_best_params']]
84:         y_coords = [point[1] for point in b_results['stats']['it_best_params']]
85:         axf.plot(x_coords, y_coords, linestyle='--', label="rndsearch b", color='orange')
86:     for b_results in results['g']['b']:
87:         x_coords = [point[0] for point in b_results['stats']['it_best_params']]
88:         y_coords = [point[1] for point in b_results['stats']['it_best_params']]
89:         axg.plot(x_coords, y_coords, linestyle='--', label="rndsearch b", color='orange')
90:     for a_results in results['f']['a']:
91:         x_coords = [point[0] for point in a_results['stats']['it_best_params']]
92:         y_coords = [point[1] for point in a_results['stats']['it_best_params']]
93:         axf.plot(x_coords, y_coords, linestyle='--', label="rndsearch a", color='blue')
94:     for a_results in results['g']['a']:
95:         x_coords = [point[0] for point in a_results['stats']['it_best_params']]
96:         y_coords = [point[1] for point in a_results['stats']['it_best_params']]
97:         axg.plot(x_coords, y_coords, linestyle='--', label="rndsearch a", color='blue')
98:
99:     axf.legend()
100:     axg.legend()
```

```
101:     plt.tight_layout()
102:     plt.savefig("fig/bii-contours.pdf")
103:
104:     return axf, axg
105:
106: max_time=1
107: if __name__ == "__main__":
108:     all_results = {}
109:     for funcs in f, g:
110:         res = list(gradient_descent_constant(funcs=funcs, step_size=funcs["alpha"]))
111:
112:
113:     plt.figure()
114:     results = {
115:         "b_mod": [],
116:         "b": [],
117:         "a": [],
118:     }
119:     res = pd.DataFrame(res)
120:     res["f(x)"] = res["x"].apply(funcs["function"])
121:
122:     for i in range(5):
123:         # ps = [{"min": 0, "max": 10}, {"min": 0, "max": 18}]
124:         # grs = global_random_search.b_mod(
125:         #     costf=funcs["function"], iterations=100, parameters=ps, N=1000, M=100, max_time=max_time)
126:         # costs = grs['stats']['it_best_costs']
127:         # plt.plot(grs['stats']['time'], costs, label="rnd search b_mod")
128:         # print(funcs["name"], "total iterations global random search b_mod: ", len(grs['stats']['time']))
129:         ps = [{"min": 0, "max": 10}, {"min": 0, "max": 18}]
130:         grs = global_random_search.b_mod(
131:             costf=funcs["function"], iterations=100, parameters=ps, N=20, M=10)
132:         costs = grs['stats']['it_best_costs']
133:         plt.plot(list(range(len(costs))), costs, label="rnd search b_mod", color="purple")
134:         results["b_mod"].append(grs)
135:         print(funcs["name"], "total iterations global random search b_mod: ", len(grs['stats']['time']))
136:
137:         ps = [{"min": 0, "max": 10}, {"min": 0, "max": 18}]
138:         grs = global_random_search.b(
139:             costf=funcs["function"], iterations=250, parameters=ps, perturb_pc=0.0001, N=400, M=100)
140:         costs = grs['stats']['it_best_costs']
141:         plt.plot(list(range(len(costs))), costs, label="rnd search b", color="blue")
142:         results["b"].append(grs)
143:         print(funcs["name"], "total iterations global random search b: ", len(grs['stats']['time']))
144:
145:         ps = [{"min": 0, "max": 10}, {"min": 0, "max": 18}]
146:         grs = global_random_search.a(
147:             costf=funcs["function"], parameters=ps, N=100000)
148:         costs = grs['stats']['it_best_costs']
149:         plt.plot(list(range(len(costs))), costs, label="rnd search a", color="orange")
150:         results["a"].append(grs)
151:         print(funcs["name"], "total iterations global random search a: ", len(grs['stats']['time']))
152:
153:
154:     plt.plot(list(range(len(res["f(x)"]))), res["f(x)"], label="gradient descent", color="black")
155:     plt.title(f"Global Random Search vs Gradient Descent on {funcs['dname']}")
156:     plt.legend(custom_lines, custom_labels, loc='lower right')
157:     plt.yscale('log')
158:     plt.xlabel("function/gradient evals")
159:     plt.ylabel(funcs['dname'])
160:     plt.tight_layout()
161:     plt.savefig(f"fig/bii-evals-{funcs['name']}.pdf")
162:     print(funcs["name"], "total iterations gradient descent: ", len(res))
163:
164:     all_results[funcs['name']] = results
```

```
1: import global_random_search
2: import lib
3: import numpy as np
4: import sgd
5: import matplotlib.pyplot as plt
6: from matplotlib.lines import Line2D
7: import pandas as pd
8: import time
9: import json
10:
11: f = {
12:     "function": lib.f_real,
13:     "gradient": lib.f_grad,
14:     "dname": "$f(x)$",
15:     "name": "f",
16:     "alpha": 0.0065,
17: }
18:
19: g = {
20:     "function": lib.g_real,
21:     "gradient": lib.g_grad,
22:     "dname": "$g(x)$",
23:     "name": "g",
24:     "alpha": 0.003,
25: }
26:
27:
28: def gradient_descent_constant(step_size=0.0065, start=[0, 0], funcs=f, max_time=0.4):
29:     start = np.array(start)
30:     g = sgd.StochasticGradientDescent()
31:     g.step_size(step_size)
32:     g.start(start)
33:     def function_generator():
34:         while True:
35:             yield funcs["function"], funcs["gradient"]
36:     g.function_generator(function_generator())
37:     g.debug(True)
38:     g.alg("constant")
39:     start_time = time.perf_counter()
40:     current_time = 0
41:     while current_time < max_time:
42:         current_time = time.perf_counter() - start_time
43:         g.step()
44:         yield {
45:             "f(x)": g._function(g._x_value),
46:             "x": g._x_value,
47:             "time": time.perf_counter() - start_time,
48:         }
49:
50:
51: custom_lines = [
52:     Line2D([0], [0], color='purple', lw=2),
53:     Line2D([0], [0], color='blue', lw=2),
54:     Line2D([0], [0], color='orange', lw=2),
55:     Line2D([0], [0], color='black', lw=2),
56: ]
57: custom_labels = ['rnd search b_mod', 'rnd search b', 'rnd search a', 'gradient descent']
58:
59: def thin(array, step = 30):
60:     return [array[i] for i in range(0, len(array), step)]
61:
62: def vis_results(results):
63:     def f(x, y):
64:         return 3 * (x - 5)**4 + 10 * (y - 9)**2
65:     def g(x, y):
66:         return np.maximum(x - 5, 0) + 10 * np.abs(y - 9)
67:
68:     x = np.linspace(0, 10, 400)
69:     y = np.linspace(0, 18, 400)
70:     X, Y = np.meshgrid(x, y)
71:     Z_f = f(X, Y)
72:     Z_g = g(X, Y)
73:
74:     fig = plt.figure(figsize=(12, 6))
75:
76:     axf = fig.add_subplot(1, 2, 1)
77:     axf.contourf(X, Y, Z_f, levels=30, cmap='viridis')
78:     axf.set_title('$f(x, y)$')
79:     axf.set_xlabel('$x$')
80:     axf.set_ylabel('$y$')
81:
82:     axg = fig.add_subplot(1, 2, 2)
83:     axg.contourf(X, Y, Z_g, levels=30, cmap='viridis')
84:     axg.set_title('$g(x, y)$')
85:     axg.set_xlabel('$x$')
86:     axg.set_ylabel('$y$')
87:
88:     cmap = plt.cm.Oranges
89:     for a_results in results['f']['a']:
90:         x_coords = thin([point[0] for point in a_results['stats']['it_best_params']])
91:         y_coords = thin([point[1] for point in a_results['stats']['it_best_params']])
92:         color = [cmap(i / len(x_coords)) for i in range(len(x_coords))]
93:         axf.scatter(x_coords, y_coords, linestyle='-', label="rndsearch a", color=color)
94:     for a_results in results['g']['a']:
95:         x_coords = thin([point[0] for point in a_results['stats']['it_best_params']])
96:         y_coords = thin([point[1] for point in a_results['stats']['it_best_params']])
97:         color = [cmap(i / len(x_coords)) for i in range(len(x_coords))]
98:         axg.scatter(x_coords, y_coords, linestyle='-', label="rndsearch a", color=color)
99:     plt.tight_layout()
100:     plt.savefig("fig/bii-contours-a.pdf")
```



```

101:
102:     x = np.linspace(0, 10, 400)
103:     y = np.linspace(0, 18, 400)
104:     X, Y = np.meshgrid(x, y)
105:     Z_f = f(X, Y)
106:     Z_g = g(X, Y)
107:
108:     fig = plt.figure(figsize=(12, 6))
109:
110:     axf = fig.add_subplot(1, 2, 1)
111:     axf.contourf(X, Y, Z_f, levels=30, cmap='viridis')
112:     axf.set_title('$f(x, y)$')
113:     axf.set_xlabel('$x$')
114:     axf.set_ylabel('$y$')
115:
116:     axg = fig.add_subplot(1, 2, 2)
117:     axg.contourf(X, Y, Z_g, levels=30, cmap='viridis')
118:     axg.set_title('$g(x, y)$')
119:     axg.set_xlabel('$x$')
120:     axg.set_ylabel('$y$')
121:
122:     cmap = plt.cm.Blues
123:     for b_results in results['f']['b']:
124:         x_coords = thin([point[0] for point in b_results['stats']['it_best_params']])
125:         y_coords = thin([point[1] for point in b_results['stats']['it_best_params']])
126:         color = [cmap(i / len(x_coords)) for i in range(len(x_coords))]
127:         axf.plot(x_coords, y_coords, linestyle='-', label="rndsearch b", color=color)
128:     for b_results in results['g']['b']:
129:         x_coords = thin([point[0] for point in b_results['stats']['it_best_params']])
130:         y_coords = thin([point[1] for point in b_results['stats']['it_best_params']])
131:         color = [cmap(i / len(x_coords)) for i in range(len(x_coords))]
132:         axg.plot(x_coords, y_coords, linestyle='-', label="rndsearch b", color=color)
133:     plt.tight_layout()
134:     plt.savefig("fig/bii-contours-b.pdf")
135:
136:     # axf.legend(custom_lines[1:3], custom_labels[1:3])
137:     # axg.legend(custom_lines[1:3], custom_labels[1:3])
138:
139:     return axf, axg
140:
141: max_time=1
142: if __name__ == "__main__":
143:     all_results = {}
144:     for funcs in f, g:
145:         res = list(gradient_descent_constant(max_time=max_time, funcs=funcs, step_size=funcs["alpha"]))
146:
147:         plt.figure()
148:         results = {
149:             "b_mod": [],
150:             "b": [],
151:             "a": [],
152:         }
153:         res = pd.DataFrame(res)
154:
155:         for i in range(5):
156:             # ps = [{"min": 0, "max": 10}, {"min": 0, "max": 18}]
157:             # grs = global_random_search.b_mod(
158:             #     costf=funcs["function"], iterations=100, parameters=ps, N=1000, M=100, max_time=max_time)
159:             # costs = grs['stats']['it_best_costs']
160:             # plt.plot(grs['stats']['time'], costs, label="rnd search b_mod")
161:             # print(funcs["name"], "total iterations global random search b_mod: ", len(grs['stats']['time']))
162:             ps = [{"min": 0, "max": 10}, {"min": 0, "max": 18}]
163:             grs = global_random_search.b_mod(
164:                 costf=funcs["function"], iterations=200, parameters=ps, N=400, M=100, max_time=max_time)
165:             costs = grs['stats']['it_best_costs']
166:             plt.plot(grs['stats']['time'], costs, label="rnd search b_mod", color="purple")
167:             results["b_mod"].append(grs)
168:             print(funcs["name"], "total iterations global random search b_mod: ", len(grs['stats']['time']))
169:
170:             ps = [{"min": 0, "max": 10}, {"min": 0, "max": 18}]
171:             grs = global_random_search.b(
172:                 costf=funcs["function"], iterations=200, parameters=ps, perturb_pc=0.001, N=400, M=100, max_time=max_time)
173:             costs = grs['stats']['it_best_costs']
174:             plt.plot(grs['stats']['time'], costs, label="rnd search b", color="blue")
175:             results["b"].append(grs)
176:             print(funcs["name"], "total iterations global random search b: ", len(grs['stats']['time']))
177:
178:             ps = [{"min": 0, "max": 10}, {"min": 0, "max": 18}]
179:             grs = global_random_search.a(
180:                 costf=funcs["function"], parameters=ps, max_time=max_time)
181:             costs = grs['stats']['it_best_costs']
182:             plt.plot(grs['stats']['time'], costs, label="rnd search a", color="orange")
183:             results["a"].append(grs)
184:             print(funcs["name"], "total iterations global random search a: ", len(grs['stats']['time']))
185:
186:
187:         plt.plot(res["time"], res["f(x)"], label="gradient descent", color="black")
188:         plt.title(f"Global Random Search vs Gradient Descent on {funcs['dname']}")
189:         plt.legend(custom_lines, custom_labels, loc='lower right')
190:         plt.yscale('log')
191:         plt.xlabel("time (seconds)")
192:         plt.ylabel(funcs['dname'])
193:         plt.tight_layout()
194:         plt.savefig(f"fig/bii-time-{funcs['name']}.pdf")
195:         print(funcs["name"], "total iterations gradient descent: ", len(res))
196:
197:         all_results[funcs['name']] = results
198:
199:     # with open("data/bii-time.json", "w") as f:
200:     #     json.dump(all_results, f)

```


src/bii-time.py Wed Apr 10 21:50:57 2024 3

201:

```
1: import global_random_search
2: import lib
3: import numpy as np
4: import sgd
5: import matplotlib.pyplot as plt
6: from matplotlib.lines import Line2D
7: import pandas as pd
8: import time
9: import json
10:
11: f = {
12:     "function": lib.f_real,
13:     "gradient": lib.f_grad,
14:     "dname": "$f(x)$",
15:     "name": "f",
16:     "alpha": 0.0065,
17: }
18:
19: g = {
20:     "function": lib.g_real,
21:     "gradient": lib.g_grad,
22:     "dname": "$g(x)$",
23:     "name": "g",
24:     "alpha": 0.003,
25: }
26:
27:
28: def gradient_descent_constant(step_size=0.0065, start=[0, 0], funcs=f, max_time=1):
29:     start = np.array(start)
30:     g = sgd.StochasticGradientDescent()
31:     g.step_size(step_size)
32:     g.start(start)
33:     def function_generator():
34:         while True:
35:             yield funcs["function"], funcs["gradient"]
36:     g.function_generator(function_generator())
37:     g.debug(True)
38:     g.alg("constant")
39:     start_time = time.perf_counter()
40:     current_time = 0
41:     while current_time < max_time:
42:         current_time = time.perf_counter() - start_time
43:         g.step()
44:         yield {
45:             "f(x)": g._function(g._x_value),
46:             "x": g._x_value,
47:             "time": time.perf_counter() - start_time,
48:         }
49:
50:
51: custom_lines = [
52:     Line2D([0], [0], color='purple', lw=2),
53:     Line2D([0], [0], color='blue', lw=2),
54:     Line2D([0], [0], color='orange', lw=2),
55:     Line2D([0], [0], color='black', lw=2),
56: ]
57: custom_labels = ['rnd search b_mod', 'rnd search b', 'rnd search a', 'gradient descent']
58:
59: def thin(array, step = 30):
60:     return [array[i] for i in range(0, len(array), step)]
61:
62: def vis_results(results):
63:     def f(x, y):
64:         return 3 * (x - 5)**4 + 10 * (y - 9)**2
65:     def g(x, y):
66:         return np.maximum(x - 5, 0) + 10 * np.abs(y - 9)
67:
68:     x = np.linspace(0, 10, 400)
69:     y = np.linspace(0, 18, 400)
70:     X, Y = np.meshgrid(x, y)
71:     Z_f = f(X, Y)
72:     Z_g = g(X, Y)
73:
74:     fig = plt.figure(figsize=(12, 6))
75:
76:     axf = fig.add_subplot(1, 2, 1)
77:     axf.contourf(X, Y, Z_f, levels=30, cmap='viridis')
78:     axf.set_title('$f(x, y)$')
79:     axf.set_xlabel('$x$')
80:     axf.set_ylabel('$y$')
81:
82:     axg = fig.add_subplot(1, 2, 2)
83:     axg.contourf(X, Y, Z_g, levels=30, cmap='viridis')
84:     axg.set_title('$g(x, y)$')
85:     axg.set_xlabel('$x$')
86:     axg.set_ylabel('$y$')
87:
88:     cmap = plt.cm.Oranges
89:     for a_results in results['f']['a']:
90:         x_coords = thin([point[0] for point in a_results['stats']['it_best_params']])
91:         y_coords = thin([point[1] for point in a_results['stats']['it_best_params']])
92:         color = [cmap(i / len(x_coords)) for i in range(len(x_coords))]
93:         axf.scatter(x_coords, y_coords, linestyle='-', label="rndsearch a", color=color)
94:     for a_results in results['g']['a']:
95:         x_coords = thin([point[0] for point in a_results['stats']['it_best_params']])
96:         y_coords = thin([point[1] for point in a_results['stats']['it_best_params']])
97:         color = [cmap(i / len(x_coords)) for i in range(len(x_coords))]
98:         axg.scatter(x_coords, y_coords, linestyle='-', label="rndsearch a", color=color)
99:     plt.tight_layout()
100:     plt.savefig("fig/bii-contours-a.pdf")
```

```

101:
102:     x = np.linspace(0, 10, 400)
103:     y = np.linspace(0, 18, 400)
104:     X, Y = np.meshgrid(x, y)
105:     Z_f = f(X, Y)
106:     Z_g = g(X, Y)
107:
108:     fig = plt.figure(figsize=(12, 6))
109:
110:     axf = fig.add_subplot(1, 2, 1)
111:     axf.contourf(X, Y, Z_f, levels=30, cmap='viridis')
112:     axf.set_title('$f(x, y)$')
113:     axf.set_xlabel('$x$')
114:     axf.set_ylabel('$y$')
115:
116:     axg = fig.add_subplot(1, 2, 2)
117:     axg.contourf(X, Y, Z_g, levels=30, cmap='viridis')
118:     axg.set_title('$g(x, y)$')
119:     axg.set_xlabel('$x$')
120:     axg.set_ylabel('$y$')
121:
122:     cmap = plt.cm.Blues
123:     for b_results in results['f']['b']:
124:         x_coors = thin([point[0] for point in b_results['stats']['it_best_params']])
125:         y_coors = thin([point[1] for point in b_results['stats']['it_best_params']])
126:         color = [cmap(i / len(x_coors)) for i in range(len(x_coors))]
127:         axf.plot(x_coors, y_coors, linestyle='--', label="rndsearch b", color=color)
128:     for b_results in results['g']['b']:
129:         x_coors = thin([point[0] for point in b_results['stats']['it_best_params']])
130:         y_coors = thin([point[1] for point in b_results['stats']['it_best_params']])
131:         color = [cmap(i / len(x_coors)) for i in range(len(x_coors))]
132:         axg.plot(x_coors, y_coors, linestyle='--', label="rndsearch b", color=color)
133:     plt.tight_layout()
134:     plt.savefig("fig/bii-contours-b.pdf")
135:
136:     # axf.legend(custom_lines[1:3], custom_labels[1:3])
137:     # axg.legend(custom_lines[1:3], custom_labels[1:3])
138:
139:     return axf, axg
140:
141: max_time=0.1
142: if __name__ == "__main__":
143:     all_results = {}
144:     for funcs in f, g:
145:         res = list(gradient_descent_constant(max_time=max_time, funcs=funcs, step_size=funcs["alpha"]))
146:
147:         plt.figure()
148:         results = {
149:             "b_mod": [],
150:             "b": [],
151:             "a": [],
152:         }
153:         res = pd.DataFrame(res)
154:
155:         for i in range(5):
156:             # ps = [{"min": 0, "max": 10}, {"min": 0, "max": 18}]
157:             # grs = global_random_search.b_mod(
158:             #     costf=funcs["function"], iterations=100, parameters=ps, N=1000, M=100, max_time=max_time)
159:             # costs = grs['stats']['it_best_costs']
160:             # plt.plot(grs['stats']['time'], costs, label="rnd search b_mod")
161:             # print(funcs["name"], "total iterations global random search b_mod: ", len(grs['stats']['time']))
162:             ps = [{"min": 0, "max": 10}, {"min": 0, "max": 18}]
163:             grs = global_random_search.b_mod(
164:                 costf=funcs["function"], iterations=8, parameters=ps, N=100, M=50)
165:             costs = grs['stats']['it_best_costs']
166:             plt.plot(grs['stats']['time'], costs, label="rnd search b_mod", color="purple")
167:             results["b_mod"].append(grs)
168:             print(funcs["name"], "total iterations global random search b_mod: ", len(grs['stats']['time']))
169:
170:             ps = [{"min": 0, "max": 10}, {"min": 0, "max": 18}]
171:             grs = global_random_search.b(
172:                 costf=funcs["function"], iterations=8, parameters=ps, perturb_pc=0.01, N=100, M=50)
173:             costs = grs['stats']['it_best_costs']
174:             plt.plot(grs['stats']['time'], costs, label="rnd search b", color="blue")
175:             results["b"].append(grs)
176:             print(funcs["name"], "total iterations global random search b: ", len(grs['stats']['time']))
177:
178:             ps = [{"min": 0, "max": 10}, {"min": 0, "max": 18}]
179:             grs = global_random_search.a(
180:                 costf=funcs["function"], parameters=ps, N=300)
181:             costs = grs['stats']['it_best_costs']
182:             plt.plot(grs['stats']['time'], costs, label="rnd search a", color="orange")
183:             results["a"].append(grs)
184:             print(funcs["name"], "total iterations global random search a: ", len(grs['stats']['time']))
185:
186:
187:         plt.plot(res["time"], res["f(x)"], label="gradient descent", color="black")
188:         plt.title(f"Global Random Search vs Gradient Descent on {funcs['dname']}")
189:         plt.legend(custom_lines, custom_labels, loc='lower right')
190:         plt.yscale('log')
191:         plt.xlabel("time (seconds)")
192:         plt.ylabel(funcs['dname'])
193:         plt.tight_layout()
194:         print(funcs["name"], "total iterations gradient descent: ", len(res))
195:
196:         all_results[funcs['name']] = results
197:
198: with open("data/bii-time.json", "w") as f:
199:     json.dump(all_results, f)

```

```
src/bii-time-vis.py      Wed Apr 10 16:10:22 2024      1
1: import global_random_search
2: import lib
3: import numpy as np
4: import sgd
5: import matplotlib.pyplot as plt
6: from matplotlib.lines import Line2D
7: import pandas as pd
8: import time
9: import json
10:
11: f = {
12:     "function": lib.f_real,
13:     "gradient": lib.f_grad,
14:     "dname": "$f(x)$",
15:     "name": "f",
16:     "alpha": 0.0065,
17: }
18:
19: g = {
20:     "function": lib.g_real,
21:     "gradient": lib.g_grad,
22:     "dname": "$g(x)$",
23:     "name": "g",
24:     "alpha": 0.003,
25: }
26:
27:
28: def gradient_descent_constant(step_size=0.0065, start=[0, 0], funcs=f, max_time=1):
29:     start = np.array(start)
30:     g = sgd.StochasticGradientDescent()
31:     g.step_size(step_size)
32:     g.start(start)
33:     def function_generator():
34:         while True:
35:             yield funcs["function"], funcs["gradient"]
36:     g.function_generator(function_generator())
37:     g.debug(True)
38:     g.alg("constant")
39:     start_time = time.perf_counter()
40:     current_time = 0
41:     while current_time < max_time:
42:         current_time = time.perf_counter() - start_time
43:         g.step()
44:         yield {
45:             "f(x)": g._function(g._x_value),
46:             "x": g._x_value,
47:             "time": time.perf_counter() - start_time,
48:         }
49:
50:
51: custom_lines = [
52:     Line2D([0], [0], color='purple', lw=2),
53:     Line2D([0], [0], color='blue', lw=2),
54:     Line2D([0], [0], color='orange', lw=2),
55:     Line2D([0], [0], color='black', lw=2),
56: ]
57: custom_labels = ['rnd search b_mod', 'rnd search b', 'rnd search a', 'gradient descent']
58:
59: def thin(array, step = 5):
60:     return [array[i] for i in range(0, len(array), step)]
61:
62: def vis_results(results):
63:     print("starting vis")
64:     def f(x, y):
65:         return 3 * (x - 5)**4 + 10 * (y - 9)**2
66:     def g(x, y):
67:         return np.maximum(x - 5, 0) + 10 * np.abs(y - 9)
68:
69:     x = np.linspace(0, 10, 400)
70:     y = np.linspace(0, 18, 400)
71:     X, Y = np.meshgrid(x, y)
72:     Z_f = f(X, Y)
73:     Z_g = g(X, Y)
74:
75:     fig = plt.figure(figsize=(12, 6))
76:
77:     axf = fig.add_subplot(1, 2, 1)
78:     axf.contourf(X, Y, Z_f, levels=30, cmap='viridis')
79:     axf.set_title('$f(x, y)$')
80:     axf.set_xlabel('$x$')
81:     axf.set_ylabel('$y$')
82:
83:     axg = fig.add_subplot(1, 2, 2)
84:     axg.contourf(X, Y, Z_g, levels=30, cmap='viridis')
85:     axg.set_title('$g(x, y)$')
86:     axg.set_xlabel('$x$')
87:     axg.set_ylabel('$y$')
88:
89:     cmap = plt.cm.Oranges
90:     for a_results in results['f']['a'][:1]:
91:         x_coords, y_coords = zip(*thin(a_results['stats']['it_best_params']))
92:         # y_coords = thin([point[1] for point in a_results['stats']['it_best_params']])
93:         color = [cmap(i / len(x_coords)) for i in range(len(x_coords))]
94:         for x,y,c in zip(x_coords, y_coords, color):
95:             axf.scatter(x, y, color=c)
96:     for a_results in results['g']['a'][:1]:
97:         x_coords = thin([point[0] for point in a_results['stats']['it_best_params']])
98:         y_coords = thin([point[1] for point in a_results['stats']['it_best_params']])
99:         color = [cmap(i / len(x_coords)) for i in range(len(x_coords))]
100:         for x,y,c in zip(x_coords, y_coords, color):
```

```
101:         axf.scatter(x, y, color=c)
102:     plt.tight_layout()
103:     plt.savefig("fig/bii-contours-a.pdf")
104:
105:     x = np.linspace(0, 10, 400)
106:     y = np.linspace(0, 18, 400)
107:     X, Y = np.meshgrid(x, y)
108:     Z_f = f(X, Y)
109:     Z_g = g(X, Y)
110:
111:     fig = plt.figure(figsize=(12, 6))
112:
113:     axf = fig.add_subplot(1, 2, 1)
114:     axf.contourf(X, Y, Z_f, levels=30, cmap='viridis')
115:     axf.set_title('$f(x, y)$')
116:     axf.set_xlabel('$x$')
117:     axf.set_ylabel('$y$')
118:
119:     axg = fig.add_subplot(1, 2, 2)
120:     axg.contourf(X, Y, Z_g, levels=30, cmap='viridis')
121:     axg.set_title('$g(x, y)$')
122:     axg.set_xlabel('$x$')
123:     axg.set_ylabel('$y$')
124:
125:     cmap = plt.cm.Blues
126:     for b_results in results['f']['b'][:1]:
127:         x_coords = thin([point[0] for point in b_results['stats']['it_best_params']])
128:         y_coords = thin([point[1] for point in b_results['stats']['it_best_params']])
129:         color = [cmap(i / len(x_coords)) for i in range(len(x_coords))]
130:         for x,y,c in zip(x_coords, y_coords, color):
131:             axf.scatter(x, y, color=c)
132:     for b_results in results['g']['b'][:1]:
133:         x_coords = thin([point[0] for point in b_results['stats']['it_best_params']])
134:         y_coords = thin([point[1] for point in b_results['stats']['it_best_params']])
135:         color = [cmap(i / len(x_coords)) for i in range(len(x_coords))]
136:         for x,y,c in zip(x_coords, y_coords, color):
137:             axf.scatter(x, y, color=c)
138:     plt.tight_layout()
139:     plt.savefig("fig/bii-contours-b.pdf")
140:
141:     # axf.legend(custom_lines[1:3], custom_labels[1:3])
142:     # axg.legend(custom_lines[1:3], custom_labels[1:3])
143:
144:     return axf, axg
145:
146:
147: if __name__ == "__main__":
148:     results = None
149:     with open("data/bii-time.json", "r") as f:
150:         results = json.load(f)
151:     vis_results(results)
```

```
1: import global_random_search
2: import lib
3: import numpy as np
4: import sgd
5: import matplotlib.pyplot as plt
6: from matplotlib.lines import Line2D
7: import pandas as pd
8: import time
9: import json
10:
11: f = {
12:     "function": lib.f_real,
13:     "gradient": lib.f_grad,
14:     "dname": "$f(x)$",
15:     "name": "f",
16:     "alpha": 0.0065,
17: }
18:
19: g = {
20:     "function": lib.g_real,
21:     "gradient": lib.g_grad,
22:     "dname": "$g(x)$",
23:     "name": "g",
24:     "alpha": 0.003,
25: }
26:
27:
28: def gradient_descent_constant(step_size=0.0065, start=[0, 0], funcs=f, max_time=1):
29:     start = np.array(start)
30:     g = sgd.StochasticGradientDescent()
31:     g.step_size(step_size)
32:     g.start(start)
33:     def function_generator():
34:         while True:
35:             yield funcs["function"], funcs["gradient"]
36:     g.function_generator(function_generator())
37:     g.debug(True)
38:     g.alg("constant")
39:     start_time = time.perf_counter()
40:     current_time = 0
41:     while current_time < max_time:
42:         current_time = time.perf_counter() - start_time
43:         g.step()
44:         yield {
45:             "f(x)": g._function(g._x_value),
46:             "x": g._x_value,
47:             "time": time.perf_counter() - start_time,
48:         }
49:
50:
51: custom_lines = [
52:     Line2D([0], [0], color='purple', lw=2),
53:     Line2D([0], [0], color='blue', lw=2),
54:     Line2D([0], [0], color='orange', lw=2),
55:     Line2D([0], [0], color='black', lw=2),
56: ]
57: custom_labels = ['rnd search b_mod', 'rnd search b', 'rnd search a', 'gradient descent']
58:
59: def thin(array, step = 1):
60:     return [array[i] for i in range(0, len(array), step)]
61:
62: def vis_results(results, args):
63:     print("starting vis")
64:     params = thin(results)
65:     def f(x, y):
66:         return 3 * (x - 5)**4 + 10 * (y - 9)**2
67:     def g(x, y):
68:         return np.maximum(x - 5, 0) + 10 * np.abs(y - 9)
69:
70:     x = np.linspace(0, 10, 400)
71:     y = np.linspace(0, 18, 400)
72:     X, Y = np.meshgrid(x, y)
73:     Z_f = f(X, Y) if args.function == "f" else g(X, Y)
74:     fig = plt.figure(figsize=(4, 4))
75:
76:     axf = fig.add_subplot(1, 1, 1)
77:     axf.contourf(X, Y, Z_f, levels=30, cmap='viridis')
78:     axf.set_title(args.title)
79:     axf.set_xlabel('$x$')
80:     axf.set_ylabel('$y$')
81:     x_coords, y_coords = zip(*thin(params, step=args.thin))
82:     # y_coords = thin([point[1] for point in params], step=args.thin)
83:     cmap = plt.cm.Blues
84:     color = [cmap(i / len(x_coords)) for i in range(len(x_coords))]
85:     for x,y,c in zip(x_coords, y_coords, color):
86:         print(".", end="", flush=True)
87:         axf.scatter(x, y, s=3, color=c)
88:     plt.tight_layout()
89:
90:
91: if __name__ == "__main__":
92:     import argparse
93:     ap = argparse.ArgumentParser()
94:     ap.add_argument("-i", type=str)
95:     ap.add_argument("-o", type=str)
96:     ap.add_argument("--title", type=str)
97:     ap.add_argument("--function", type=str)
98:     ap.add_argument("--thin", type=int, default=20)
99:     args = ap.parse_args()
100:     results = None
```

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



```
1: import matplotlib.pyplot as plt
2: import numpy as np
3: import sys
4:
5:
6: def f(x, y):
7:     return 3 * (x - 5)**4 + 10 * (y - 9)**2
8:
9:
10: def g(x, y):
11:     return np.maximum(x - 5, 0) + 10 * np.abs(y - 9)
12:
13:
14: def main(outfile):
15:     x = np.linspace(0, 10, 400)
16:     y = np.linspace(0, 18, 400)
17:     X, Y = np.meshgrid(x, y)
18:     Z_f = f(X, Y)
19:     Z_g = g(X, Y)
20:
21:     fig = plt.figure(figsize=(12, 6))
22:
23:     ax = fig.add_subplot(1, 2, 1, projection='3d')
24:     ax.plot_surface(X, Y, Z_f, cmap='viridis')
25:     ax.set_title('$f(x, y)$')
26:     ax.set_xlabel('$x$')
27:     ax.set_ylabel('$y$')
28:     ax.set_zlabel('$f(x, y)$')
29:
30:     ax = fig.add_subplot(1, 2, 2, projection='3d')
31:     ax.plot_surface(X, Y, Z_g, cmap='magma')
32:     ax.set_title('$g(x, y)$')
33:     ax.set_xlabel('$x$')
34:     ax.set_ylabel('$y$')
35:     ax.set_zlabel('$g(x, y)$')
36:
37:     plt.savefig(outfile)
38:     plt.show()
39:
40: def main_contour(outfile):
41:     x = np.linspace(0, 10, 400)
42:     y = np.linspace(0, 18, 400)
43:     X, Y = np.meshgrid(x, y)
44:     Z_f = f(X, Y)
45:     Z_g = g(X, Y)
46:
47:     fig = plt.figure(figsize=(12, 6))
48:
49:     ax = fig.add_subplot(1, 2, 1)
50:     ax.contourf(X, Y, Z_f, levels=30, cmap='viridis')
51:     ax.set_title('$f(x, y)$')
52:     ax.set_xlabel('$x$')
53:     ax.set_ylabel('$y$')
54:     # ax.set_zlabel('$f(x, y)$')
55:
56:     ax = fig.add_subplot(1, 2, 2)
57:     ax.contourf(X, Y, Z_g, levels=30, cmap='viridis')
58:     ax.set_title('$g(x, y)$')
59:     ax.set_xlabel('$x$')
60:     ax.set_ylabel('$y$')
61:     # ax.set_zlabel('$g(x, y)$')
62:
63:     plt.savefig(outfile)
64:     plt.show()
65:
66:
67: if __name__ == "__main__":
68:     if len(sys.argv) != 2:
69:         print("Usage: python script.py <output_file>")
70:         sys.exit(1)
71:
72:     outfile = sys.argv[1]
73:     main_contour(outfile)
74:
```

```
1: import global_random_search
2: import lib
3: import numpy as np
4: import sgd
5: import matplotlib.pyplot as plt
6: import pandas as pd
7: import time
8: import cifar_costf
9: import json
10: import argparse
11: import c_vis
12: import even_samples
13: import math
14: import cps
15: ps = cps.ps
16:
17: ap = argparse.ArgumentParser()
18: # ap.add_argument("--exp", type=str, required=True)
19: ap.add_argument("--M", type=int, required=True)
20: ap.add_argument("--N", type=int, required=True)
21: ap.add_argument("--n", type=int, required=True)
22: ap.add_argument("--iterations", type=int, required=True)
23: args = ap.parse_args()
24:
25: f = {
26:     "function": lib.f_real,
27:     "gradient": lib.f_grad,
28:     "dname": "$f(x)$",
29:     "name": "f",
30:     "alpha": 0.0065,
31: }
32:
33: g = {
34:     "function": lib.g_real,
35:     "gradient": lib.g_grad,
36:     "dname": "$g(x)$",
37:     "name": "g",
38:     "alpha": 0.003,
39: }
40:
41:
42: def gradient_descent_constant(step_size=0.0065, start=[0, 0], funcs=f, max_time=1):
43:     start = np.array(start)
44:     g = sgd.StochasticGradientDescent()
45:     g.step_size(step_size)
46:     g.start(start)
47:     def function_generator():
48:         while True:
49:             yield funcs["function"], funcs["gradient"]
50:     g.function_generator(function_generator())
51:     g.debug(True)
52:     g.alg("constant")
53:     start_time = time.time()
54:     current_time = 0
55:     while current_time < max_time:
56:         current_time = time.time() - start_time
57:         g.step()
58:         yield {
59:             "f(x)": g._function(g._x_value),
60:             "x": g._x_value,
61:             "time": time.time() - start_time,
62:         }
63:
64: if __name__ == "__main__":
65:     train, test = even_samples.even_sample_categories(math.floor(args.n))
66:
67:     def costf(x):
68:         return cifar_costf.costf(x, train, test)
69:
70:     grs = global_random_search.a(
71:         debug=True,
72:         costf=costf, parameters=ps, N=args.N*args.iterations)
73:
74:     fname = f"data/c-a-N{args.N*args.iterations}.json"
75:     save = {
76:         'results': grs,
77:         'param-limits': ps,
78:         'args': vars(args),
79:         'name': None,
80:     }
81:     with open(fname, "w") as f:
82:         json.dump(save, f)
```

```
1: import global_random_search
2: import lib
3: import numpy as np
4: import sgd
5: import matplotlib.pyplot as plt
6: import pandas as pd
7: import time
8: import cifar_costf
9: import json
10: import argparse
11: import c_vis
12: import even_samples
13: import math
14: import cps
15: ps = cps.ps
16:
17: ap = argparse.ArgumentParser()
18: # ap.add_argument("--exp", type=str, required=True)
19: ap.add_argument("--M", type=int, required=True)
20: ap.add_argument("--N", type=int, required=True)
21: ap.add_argument("--n", type=int, required=True)
22: ap.add_argument("--iterations", type=int, required=True)
23: args = ap.parse_args()
24:
25: f = {
26:     "function": lib.f_real,
27:     "gradient": lib.f_grad,
28:     "dname": "$f(x)$",
29:     "name": "f",
30:     "alpha": 0.0065,
31: }
32:
33: g = {
34:     "function": lib.g_real,
35:     "gradient": lib.g_grad,
36:     "dname": "$g(x)$",
37:     "name": "g",
38:     "alpha": 0.003,
39: }
40:
41:
42: def gradient_descent_constant(step_size=0.0065, start=[0, 0], funcs=f, max_time=1):
43:     start = np.array(start)
44:     g = sgd.StochasticGradientDescent()
45:     g.step_size(step_size)
46:     g.start(start)
47:     def function_generator():
48:         while True:
49:             yield funcs["function"], funcs["gradient"]
50:     g.function_generator(function_generator())
51:     g.debug(True)
52:     g.alg("constant")
53:     start_time = time.time()
54:     current_time = 0
55:     while current_time < max_time:
56:         current_time = time.time() - start_time
57:         g.step()
58:         yield {
59:             "f(x)": g._function(g._x_value),
60:             "x": g._x_value,
61:             "time": time.time() - start_time,
62:         }
63:
64: if __name__ == "__main__":
65:     train, test = even_samples.even_sample_categories(math.floor(args.n))
66:
67:     def costf(x):
68:         return cifar_costf.costf(x, train, test)
69:
70:     grs = global_random_search.b_mod(
71:         debug=True,
72:         costf=costf, parameters=ps, N=args.N, M=args.M, iterations=args.iterations)
73:
74:     fname = f"data/c-b_mod-N{args.N}-M{args.M}-n{args.n}-it{args.iterations}.json"
75:     save = {
76:         'results': grs,
77:         'param-limits': ps,
78:         'args': vars(args),
79:         'name': None,
80:     }
81:     with open(fname, "w") as f:
82:         json.dump(save, f)
```

```
1: import global_random_search
2: import lib
3: import numpy as np
4: import sgd
5: import matplotlib.pyplot as plt
6: import pandas as pd
7: import time
8: import cifar_costf
9: import json
10: import argparse
11: import c_vis
12: import even_samples
13: import math
14: import cps
15: ps = cps.ps
16:
17: ap = argparse.ArgumentParser()
18: # ap.add_argument("--exp", type=str, required=True)
19: ap.add_argument("--M", type=int, required=True)
20: ap.add_argument("--N", type=int, required=True)
21: ap.add_argument("--n", type=int, required=True)
22: ap.add_argument("--iterations", type=int, required=True)
23: args = ap.parse_args()
24:
25: f = {
26:     "function": lib.f_real,
27:     "gradient": lib.f_grad,
28:     "dname": "$f(x)$",
29:     "name": "f",
30:     "alpha": 0.0065,
31: }
32:
33: g = {
34:     "function": lib.g_real,
35:     "gradient": lib.g_grad,
36:     "dname": "$g(x)$",
37:     "name": "g",
38:     "alpha": 0.003,
39: }
40:
41:
42: def gradient_descent_constant(step_size=0.0065, start=[0, 0], funcs=f, max_time=1):
43:     start = np.array(start)
44:     g = sgd.StochasticGradientDescent()
45:     g.step_size(step_size)
46:     g.start(start)
47:     def function_generator():
48:         while True:
49:             yield funcs["function"], funcs["gradient"]
50:     g.function_generator(function_generator())
51:     g.debug(True)
52:     g.alg("constant")
53:     start_time = time.time()
54:     current_time = 0
55:     while current_time < max_time:
56:         current_time = time.time() - start_time
57:         g.step()
58:         yield {
59:             "f(x)": g._function(g._x_value),
60:             "x": g._x_value,
61:             "time": time.time() - start_time,
62:         }
63:
64: if __name__ == "__main__":
65:
66:     train, test = even_samples.even_sample_categories(math.floor(args.n))
67:
68:     def costf(x):
69:         return cifar_costf.costf(x, train, test)
70:
71:     grs = global_random_search.b(
72:         debug=True,
73:         costf=costf, parameters=ps, N=args.N, M=args.M, iterations=args.iterations)
74:
75:     fname = f"data/c-b-N{args.N}-M{args.M}-n{args.n}-it{args.iterations}.json"
76:     save = {
77:         'results': grs,
78:         'param-limits': ps,
79:         'args': vars(args),
80:         'name': None,
81:     }
82:     with open(fname, "w") as f:
83:         json.dump(save, f)
```

```
1: import tensorflow as tf
2: import numpy as np
3: import math
4: from tensorflow import keras
5: from tensorflow.keras import layers, regularizers
6: from keras.layers import Dense, Dropout, Activation, Flatten, BatchNormalization
7: from keras.layers import Conv2D, MaxPooling2D, LeakyReLU
8: from sklearn.metrics import confusion_matrix, classification_report
9: from sklearn.utils import shuffle
10: import matplotlib.pyplot as plt
11: plt.rc('font', size=18)
12: plt.rcParams['figure.constrained_layout.use'] = True
13: import sys
14: from sklearn.metrics import roc_auc_score
15: import multiprocessing
16: import even_samples
17:
18: #num_cores = int(multiprocessing.cpu_count()/2)
19: #tf.config.threading.set_inter_op_parallelism_threads(num_cores)
20: #tf.config.threading.set_intra_op_parallelism_threads(num_cores)
21:
22: # Model / data parameters
23: num_classes = 10
24: input_shape = (32, 32, 3)
25:
26: (x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
27:
28: # Scale images to the [0, 1] range
29: print("orig x_train shape:", x_train.shape)
30: x_train = x_train.astype("float32") / 255
31: x_test = x_test.astype("float32") / 255
32:
33: # convert class vectors to binary class matrices
34: y_train = keras.utils.to_categorical(y_train, num_classes)
35: y_test = keras.utils.to_categorical(y_test, num_classes)
36:
37: def params2dict(x):
38:     minibatch, alpha, beta1, beta2, epochs = x
39:     return {
40:         'minibatch': minibatch,
41:         'alpha': alpha,
42:         'beta1': beta1,
43:         'beta2': beta2,
44:         'epochs': epochs,
45:     }
46:
47: def compute_auc_loss(model, x_test, y_test):
48:     # Get predicted probabilities for each class
49:     preds = model.predict(x_test)
50:
51:     # Compute AUC score for each class
52:     auc_scores = []
53:     for class_idx in range(num_classes):
54:         auc_score = roc_auc_score(y_test[:, class_idx], preds[:, class_idx])
55:         auc_scores.append(auc_score)
56:
57:     return auc_scores
58:
59: def compute_macro_auc(model, x_test, y_test):
60:     # Get predicted probabilities for each class
61:     preds = model.predict(x_test)
62:
63:     # Compute AUC score for each class
64:     auc_scores = []
65:     for class_idx in range(num_classes):
66:         auc_score = roc_auc_score(y_test[:, class_idx], preds[:, class_idx])
67:         auc_scores.append(auc_score)
68:
69:     # Compute macro-average AUC
70:     macro_auc = sum(auc_scores) / len(auc_scores)
71:
72:     return macro_auc
73:
74: def costf(x, train, test):
75:     x_train, y_train = train
76:     x_test, y_test = test
77:     print("params: ", params2dict(x))
78:     print("training data:", len(x_train))
79:     x_train_sub = x_train
80:     y_train_sub = y_train
81:     minibatch, alpha, beta1, beta2, epochs = x
82:     minibatch = math.floor(minibatch)
83:     epochs = math.floor(epochs)
84:     model = keras.Sequential()
85:     model.add(Conv2D(16, (3,3), padding='same', input_shape=x_train_sub.shape[1:], activation='relu'))
86:     model.add(Conv2D(16, (3,3), strides=(2,2), padding='same', activation='relu'))
87:     model.add(Conv2D(32, (3,3), padding='same', activation='relu'))
88:     model.add(Conv2D(32, (3,3), strides=(2,2), padding='same', activation='relu'))
89:     model.add(Dropout(0.5))
90:     model.add(Flatten())
91:     model.add(Dense(num_classes, activation='softmax', kernel_regularizer=regularizers.l1(0.0001)))
92:
93:     adam_optimizer = keras.optimizers.Adam(learning_rate=alpha, beta_1=beta1, beta_2=beta2)
94:     model.compile(loss="categorical_crossentropy", optimizer=adam_optimizer, metrics=["accuracy"])
95:     model.summary()
96:     batch_size = minibatch
97:     print(x_train_sub.shape, y_train_sub.shape)
98:     history = model.fit(x_train_sub, y_train_sub, batch_size=batch_size, epochs=epochs, validation_split=0.1)
99:     test_loss, _ = model.evaluate(x_test, y_test, verbose=0)
100:     return test_loss
```

src/cifar_costf.py

Tue Apr 09 13:38:16 2024

2

101:

102: **if** `__name__` == `"__main__"`:

103: **print**(costf(np.array([5, 0.0001, 0.9, 0.999, 3]), n=50000))

```
1: ps = [  
2:     { "min": 1,          "max": 1024 },      # minibatch  
3:     { "min": 0.000000000001, "max": 0.01 },  # alpha  
4:     { "min": 0.5,        "max": 1 },         # beta1  
5:     { "min": 0.5,        "max": 1 },         # beta2  
6:     { "min": 10,         "max": 100 },       # epochs  
7: ]
```



```
1: import global_random_search
2: import lib
3: import numpy as np
4: import sgd
5: import matplotlib.pyplot as plt
6: import pandas as pd
7: import time
8: import cifar_costf
9: import json
10: import argparse
11:
12: ap = argparse.ArgumentParser()
13: # ap.add_argument("--exp", type=str, required=True)
14: ap.add_argument("--M", type=int, required=True)
15: ap.add_argument("--N", type=int, required=True)
16: ap.add_argument("--n", type=int, required=True)
17: ap.add_argument("--iterations", type=int, required=True)
18: args = ap.parse_args()
19:
20: f = {
21:     "function": lib.f_real,
22:     "gradient": lib.f_grad,
23:     "dname": "$f(x)$",
24:     "name": "f",
25:     "alpha": 0.0065,
26: }
27:
28: g = {
29:     "function": lib.g_real,
30:     "gradient": lib.g_grad,
31:     "dname": "$g(x)$",
32:     "name": "g",
33:     "alpha": 0.003,
34: }
35:
36:
37: def gradient_descent_constant(step_size=0.0065, start=[0, 0], funcs=f, max_time=1):
38:     start = np.array(start)
39:     g = sgd.StochasticGradientDescent()
40:     g.step_size(step_size)
41:     g.start(start)
42:     def function_generator():
43:         while True:
44:             yield funcs["function"], funcs["gradient"]
45:     g.function_generator(function_generator())
46:     g.debug(True)
47:     g.alg("constant")
48:     start_time = time.time()
49:     current_time = 0
50:     while current_time < max_time:
51:         current_time = time.time() - start_time
52:         g.step()
53:         yield {
54:             "f(x)": g._function(g._x_value),
55:             "x": g._x_value,
56:             "time": time.time() - start_time,
57:         }
58:
59: if __name__ == "__main__":
60:     ps = [
61:         {"min": 1, "max": args.n},, # minibatch
62:         {"min": 0.0000000001, "max": 5},, # alpha
63:         {"min": 0, "max": 1},, # beta1
64:         {"min": 0, "max": 1},, # beta2
65:         {"min": 1, "max": 40},, # epochs
66:     ]
67:
68:     def costf(x):
69:         return cifar_costf.costf(x, n=args.n)
70:
71:     grs = global_random_search.b_mod(
72:         debug=True,
73:         costf=costf, parameters=ps, N=args.N, M=args.M, iterations=args.iterations)
74:     costs = grs['stats']['it_best_costs']
75:
76:     print(grs)
77:     timei = time.time()
78:     fname = f"data/c-N{args.N}-M{args.M}-n{args.n}-it{args.iterations}"
79:     save = {
80:         'results': grs,
81:         'param-limits': ps,
82:         'args': vars(args),
83:         'name': None,
84:     }
85:     with open(f"{fname}.json", "w") as f:
86:         json.dump(grs, f)
```

```
1: import json
2: import matplotlib.pyplot as plt
3: from matplotlib.lines import Line2D
4:
5: def fix_costs_monotonic(costs):
6:     costs_monotonic = []
7:     best_cost = costs[0]
8:     for cost in costs:
9:         if cost <= best_cost:
10:             best_cost = cost
11:     costs_monotonic.append(best_cost)
12:     return costs_monotonic
13:
14: def visualize_stats_time_vs_it_best_costs(json_file, **kwargs):
15:     with open(json_file, 'r') as f:
16:         results = json.load(f)
17:         print(results)
18:         time = results['results']['stats']['time']
19:         it_best_costs = results['results']['stats']['it_best_costs']
20:         it_best_costs = fix_costs_monotonic(it_best_costs)
21:         plt.plot(list(range(len(it_best_costs))), it_best_costs, linestyle='--', **kwargs)
22:
23: # 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
24: if __name__ == "__main__":
25:     plt.figure(figsize=(8, 6))
26:     plt.ylim(1.5, 2.2)
27:     visualize_stats_time_vs_it_best_costs('data/c-a-N99.json', label='rnd search a $(N=99,n=1000)$', color='orange')
28:     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')
29:     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')
30:     # visualize_stats_time_vs_it_best_costs('data/c-a-N100-M-1-n1000-it3.json', label='rnd search b')
31:
32:     plt.axhline(y=1.8646, color='red', linestyle='--')
33:     plt.xlabel('function evaluations')
34:     plt.ylabel('logistic loss on test ($n=10000$)')
35:     custom_lines = [
36:         Line2D([0], [0], color='blue', lw=2),
37:         Line2D([0], [0], color='orange', lw=2),
38:         Line2D([0], [0], color='purple', lw=2),
39:         Line2D([0], [0], color='red', lw=2, linestyle='--'),
40:     ]
41:     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']
42:     plt.legend(custom_lines, custom_labels)
43:     plt.savefig('fig/c.pdf')
```

```
1: import numpy as np
2: import keras
3: from sklearn.model_selection import StratifiedShuffleSplit
4:
5: num_classes = 10
6:
7: def even_sample_categories(n):
8:     (x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
9:     x_train = x_train.astype("float32") / 255
10:    x_test = x_test.astype("float32") / 255
11:    y_train = keras.utils.to_categorical(y_train, num_classes)
12:    y_test = keras.utils.to_categorical(y_test, num_classes)
13:
14:    return (x_train[1:n], y_train[1:n]), (x_test, y_test)
15:
16: if __name__ == "__main__":
17:     # Example usage
18:     X_train_even, y_train_even = even_sample_categories(num_samples_per_class=10)
19:     print(X_train_even, y_train_even)
20:     print(x_train.shape, y_train.shape)
21:     print(X_train_even.shape, y_train_even.shape)
```

```
1: import even_samples
2: import cifar_costf
3: import numpy as np
4: import keras
5:
6: a = {
7:     "best_params": [
8:         913.957430854217,      # minibatch
9:         0.0015701252586464568, # alpha
10:        0.6575874719325618,    # beta_1
11:        0.932720394784433,     # beta_2
12:        81.32088463431727      # num_epochs
13:    ],
14:     "best_cost": 1.8064099550247192
15: }
16:
17:
18: b = {
19:     "best_params": [
20:         534.4469442210992,      # minibatch
21:         0.0006231460669478447, # alpha
22:         0.7991814790199026,    # beta_1
23:         0.9007039736299371,    # beta_2
24:         44.05592177501114      # num_epochs
25:    ],
26:     "best_cost": 1.7486121654510498
27: }
28:
29: b_mod = {
30:     "best_params": [
31:         742.2428227795274,      # minibatch
32:         0.0009079703308546692, # alpha
33:         0.8199336231638713,    # beta_1
34:         0.6038924210437369,    # beta_2
35:         64.06011278706069      # num_epochs
36:    ],
37:     "best_cost": 1.7933474779129028
38: }
39:
40: versions = [("a", a), ("b", b), ("b_mod", b_mod)]
41:
42: (x_train, y_train), (x_test, y_test)= even_samples.even_sample_categories(50000)
43: with open("final-check.txt", "w") as f:
44:     for name, version in versions:
45:         params = np.array(version["best_params"])
46:         cost = cifar_costf.costf(params, (x_train[:1000],y_train[:1000]), (x_train[1000:],y_train[1000:]))
47:         version["test_cost"] = cost
48:         version["name"] = name
49:         print(version, file=f)
```

```
1: class GradientDescent():
2:     def __init__(self):
3:         self._max_iter = 1000
4:         self._debug = False
5:         self._converged = lambda x1, x2: False
6:         self._epsilon = 0.0001
7:         self._dimension = None
8:         self._beta = 0
9:         self._algorithm = None
10:        self._iteration = None
11:        self._function = None
12:        self._sum = None
13:        self._x_value = None
14:        self._step_coeff = None
15:        self._converged_value = None
16:        self._grad_value = None
17:        self._m = None
18:        self._v = None
19:        self._adam_grad = None
20:        self._beta = None
21:        self._beta2 = None
22:        self._step_size = None
23:        self._z = None
24:        self._f_star = None
25:
26:    def step_size(self, a):
27:        self._step_size = a
28:        return self
29:
30:    def beta(self, b):
31:        self._beta = b
32:        return self
33:
34:    def beta2(self, b):
35:        self._beta2 = b
36:        return self
37:
38:    def epsilon(self, e):
39:        self._epsilon = e
40:        return self
41:
42:    def function(self, f, function_name=None, dimension=None):
43:        self._function = f
44:        self.function_name = function_name
45:        self._dimension = dimension
46:        return self
47:
48:    def sym_function(self, function, function_name=None):
49:        self.function_name = function_name
50:        self._dimension = len(function.free_symbols)
51:        def fn(x):
52:            return apply_sym(x, function)
53:
54:        diffs = [function.diff(var) for var in function.free_symbols]
55:
56:        def grad(x):
57:            return np.array([
58:                apply_sym(x, diff) for diff in diffs])
59:
60:        self._function = fn
61:        self._gradient = grad
62:        return self
63:
64:    def gradient(self, g):
65:        self._gradient = g
66:        return self
67:
68:    def max_iter(self, m):
69:        self._max_iter = m
70:        return self
71:
72:    def start(self, s):
73:        self._start = s
74:        self._x_value = s
75:        return self
76:
77:    def debug(self, d):
78:        self._debug = d
79:        return self
80:
81:    def converged(self, c):
82:        self._converged = c
83:        return self
84:
85:    def set_iterate(self, f):
86:        self.iterate = functools.partial(f, self)
87:        return self
88:
89:    def algorithm(self, alg):
90:        self._algorithm = alg
91:        if self._algorithm == "rmsprop":
92:            import rmsprop
93:            self.set_iterate(rmsprop.iterate)
94:        elif self._algorithm == "adam":
95:            import adam
96:            self.set_iterate(adam.iterate)
97:        elif self._algorithm == "heavy_ball":
98:            import heavy_ball
99:            self.set_iterate(heavy_ball.iterate)
100:        else:
```

```
101:         raise Exception("Unknown algorithm:" + alg)
102:     return self
103:
104:     def state_dict(self):
105:         print(self._function(self._x_value))
106:         return {
107:             "alg": self._algorithm,
108:             "function_name": self.function_name,
109:             "iteration": self._iteration,
110:             "step_coeff": self._step_coeff,
111:             "adam_grad": self._adam_grad,
112:             "f(x)": self._function(self._x_value),
113:             "epsilon": self._epsilon,
114:             "converged": self._converged_value,
115:             "gradient": self._grad_value,
116:             "m": self._m,
117:             "v": self._v,
118:             "beta1": self._beta,
119:             "beta2": self._beta2,
120:             "alpha": self._step_size,
121:             "sum": self._sum,
122:             "z": self._z,
123:             **{"x" + str(i): self._x_value[i] for i in range(len(self._x_value))},
124:         }
125:
126:     def run2csv(self, fname, summarise=True):
127:         import pandas as pd
128:         iterations = list(self.iterate())
129:         df = pd.DataFrame(iterations)
130:         df.to_csv(fname)
131:         if summarise:
132:             with open(fname + ".summary", "w") as f:
133:                 print(f"iterations: {len(df)}", file=f)
134:                 print(f"start: {df['x0'][0]} {df['x1'][0]}", file=f)
135:                 print(f"final: {df['x0'][len(df) - 1]} {df['x1'][len(df) - 1]}", file=f)
```

```
1: def perturb(x, alpha=0.1):
2:     # generate random point in the unit hypersphere
3:     ndim = x.shape[0]
4:     random_point = np.random.normal(size=ndim)
5:     random_point /= np.linalg.norm(random_point)
6:
7:     # scale and translate the point to fit the specified center and radius
8:     perturbed_point = x + alpha * x * random_point
9:
10:    return perturbed_point
11:
12: def b_mod(costf=None, parameters=None, alpha=0.1, iterations=2, N=100, M=10, max_time=-1, debug=False):
13:     if costf is None:
14:         raise Exception("costf is a required kwarg")
15:     if parameters is None:
16:         raise Exception("parameters is a required kwarg")
17:     it_best_costs = []
18:     start_time = time.time()
19:     best_cost = None
20:     best_params = None
21:     times = []
22:     if max_time > 0:
23:         N = -1
24:     current_time = 0
25:     params = []
26:     costs = []
27:     it = 0
28:     while (it < N or N < 0) and (current_time < max_time or max_time < 0):
29:         it += 1
30:         ps = gen_params(parameters)
31:         cost = costf(ps)
32:         params.append(ps)
33:         costs.append(cost)
34:         if best_cost is None or cost < best_cost:
35:             best_params = ps
36:             best_cost = cost
37:         it_best_costs.append(best_cost)
38:         current_time = time.time() - start_time
39:         times.append(current_time)
40:         if debug:
41:             print("parameters:", ps, end="\t")
42:             print("cost:", cost, end="\t")
43:             print("best cost:", best_cost)
44:     bests = best_m(params, costs, M=M)
45:
46:     for i in range(iterations):
47:         params = []
48:         costs = []
49:         it = 0
50:         while it < N and (current_time < max_time or max_time < 0):
51:             it += 1
52:             choice = random.choice(bests)
53:             new_params = perturb(choice, alpha=alpha)
54:             new_cost = costf(choice)
55:             params.append(new_params)
56:             costs.append(new_cost)
57:             if new_cost < best_cost:
58:                 best_cost = new_cost
59:                 best_params = new_params
60:
61:     return {
62:         "results": {
63:             "best_params": best_params,
64:             "best_cost": best_cost,
65:         },
66:         "stats": {
67:             "it_best_costs": it_best_costs,
68:             "time": times,
69:         }
70:     }
```



```
1: import numpy as np
2: import lib
3: import time
4: import random
5:
6: def gen_params(parameters):
7:     p = np.zeros(len(parameters), dtype=np.float64)
8:     for i, par in enumerate(parameters):
9:         mini = par["min"]
10:        maxi = par["max"]
11:        p[i] = np.random.uniform(mini, maxi)
12:    return p
13:
14:
15: def a(costf=None, parameters=None, N=100, max_time=-1, debug=False):
16:     if costf is None:
17:         raise Exception("costf is a required kwarg")
18:     if parameters is None:
19:         raise Exception("parameters is a required kwarg")
20:     best_params = None
21:     best_cost = None
22:     it_best_costs = []
23:     it_best_params = []
24:     it_params = []
25:     start_time = time.perf_counter()
26:     times = []
27:     it = 0
28:     if max_time > 0:
29:         N = -1
30:     current_time = 0
31:     while (it < N or N < 0) and (current_time < max_time or max_time < 0):
32:         it += 1
33:         ps = gen_params(parameters)
34:         cost = costf(ps)
35:         if best_cost is None or np.isnan(best_cost) or cost < best_cost:
36:             best_params = ps
37:             best_cost = cost
38:             it_best_costs.append(best_cost)
39:             it_best_params.append(best_params)
40:             it_params.append(ps)
41:             current_time = time.perf_counter() - start_time
42:             times.append(current_time)
43:             if debug:
44:                 print("parameters:", ps, end="\t")
45:                 print("cost:", cost, end="\t")
46:                 print("best cost:", best_cost)
47:     return {
48:         "results": {
49:             "best_params": best_params.tolist(),
50:             "best_cost": best_cost,
51:         },
52:         "stats": {
53:             "it_best_costs": it_best_costs,
54:             "it_best_params": list(map(lambda x: x.tolist(), it_best_params)),
55:             "it_params": list(map(lambda x: x.tolist(), it_params)),
56:             "time": times,
57:         }
58:     }
59:
60: def best_m(params, costs, M=10, unzip=True):
61:     bests = sorted(zip(params, costs), key=lambda x: x[1])
62:     best_M = bests[:M]
63:     if unzip:
64:         return list(zip(*best_M))
65:     return best_M
66:
67: def bests2parameters(bests):
68:     params = bests[0]
69:     p1 = params[0]
70:     ps = []
71:     for i in range(len(p1)):
72:         param_values = list(map(lambda x: x[i], params))
73:         ps.append({
74:             "min": min(param_values),
75:             "max": max(param_values),
76:         })
77:     return ps
78:
79: def b_mod(costf=None, parameters=None, iterations=2, N=100, M=10, max_time=-1, debug=False):
80:     if costf is None:
81:         raise Exception("costf is a required kwarg")
82:     if parameters is None:
83:         raise Exception("parameters is a required kwarg")
84:     it_best_costs = []
85:     it_best_params = []
86:     it_params = []
87:     start_time = time.perf_counter()
88:     best_cost = None
89:     best_params = None
90:     times = []
91:     current_time = 0
92:     iteration_results = []
93:     for i in range(iterations):
94:         if debug:
95:             print("iteration: ", i)
96:         if max_time > 0 and current_time > max_time:
97:             break
98:         params = []
99:         costs = []
100:        it = 0
```

```
101:         while it < N:
102:             it += 1
103:             ps = gen_params(parameters)
104:             cost = costf(ps)
105:             params.append(ps)
106:             costs.append(cost)
107:             if best_cost is None or cost < best_cost:
108:                 best_params = ps
109:                 best_cost = cost
110:             it_best_costs.append(best_cost)
111:             it_best_params.append(best_params)
112:             it_params.append(ps)
113:             current_time = time.perf_counter() - start_time
114:             times.append(current_time)
115:             if debug:
116:                 print("parameters:", ps, end="\t")
117:                 print("cost:", cost, end="\t")
118:                 print("best cost:", best_cost)
119:             bests = best_m(params, costs, M=M)
120:             parameters = bests2parameters(bests)
121:             iteration_results.append({
122:                 "M": M,
123:                 "best_m_params": list(map(lambda x: x.tolist(), bests[0])),
124:                 "best_m_costs": bests[1],
125:                 "best_params": best_params.tolist(),
126:                 "best_cost": best_cost,
127:             })
128:         return {
129:             "results": {
130:                 "best_params": best_params.tolist(),
131:                 "best_cost": best_cost,
132:             },
133:             "stats": {
134:                 "it_best_costs": it_best_costs,
135:                 "it_best_params": list(map(lambda x: x.tolist(), it_best_params)),
136:                 "it_params": list(map(lambda x: x.tolist(), it_params)),
137:                 "time": times,
138:             },
139:             "iteration_results": iteration_results,
140:         }
141:
142: def perturb(x, alpha=1.1):
143:     # generate random point in the unit hypersphere
144:     print(x, type(x))
145:     ndim = x.shape[0]
146:     random_point = np.random.normal(size=ndim)
147:     random_point /= np.linalg.norm(random_point)
148:
149:     # scale and translate the point to fit the specified center and radius
150:     perturbed_point = x + alpha * random_point
151:
152:     return perturbed_point
153:
154: def perturbn(x, alpha):
155:     """
156:     Randomly perturbs each element of x by adding noise from [-alpha, alpha].
157:
158:     Args:
159:     - x (list or numpy array): The input array.
160:     - alpha (float): The range of noise to add. The noise is drawn from the interval [-alpha, alpha].
161:
162:     Returns:
163:     - list: The perturbed array.
164:     """
165:     perturbed_x = [elem + random.uniform(-alpha, alpha) for elem in x]
166:     return perturbed_x
167:
168: def perturb_percent(x, percent=0.1, ps=None):
169:     if ps is None:
170:         raise Exception("require parameters ps")
171:     out = np.zeros(x.shape)
172:     for i in range(len(x)):
173:         span = ps[i]['max'] - ps[i]['min']
174:         low = -span*percent
175:         high = span*percent
176:         r = np.random.uniform(low=low, high=high, size=1)
177:         out[i] = x[i] + r
178:         out[i] = max(ps[i]['min'], out[i])
179:         out[i] = min(ps[i]['max'], out[i])
180:     return out
181:
182:
183: def b(costf=None, parameters=None, perturb_pc=0.1, iterations=2, N=100, M=10, max_time=-1, debug=False):
184:     if costf is None:
185:         raise Exception("costf is a required kwarg")
186:     if parameters is None:
187:         raise Exception("parameters is a required kwarg")
188:     it_best_costs = []
189:     it_best_params = []
190:     it_params = []
191:     start_time = time.perf_counter()
192:     best_cost = None
193:     best_params = None
194:     times = []
195:     current_time = 0
196:     params = []
197:     costs = []
198:     it = 0
199:     while (it < N or N < 0) and (current_time < max_time or max_time < 0):
200:         it += 1
```

```
201:         ps = gen_params(parameters)
202:         cost = costf(ps)
203:         params.append(ps)
204:         costs.append(cost)
205:         if best_cost is None or cost < best_cost:
206:             best_params = ps
207:             best_cost = cost
208:         it_best_costs.append(best_cost)
209:         it_best_params.append(best_params)
210:         it_params.append(ps)
211:         current_time = time.perf_counter() - start_time
212:         times.append(current_time)
213:         if debug:
214:             print("parameters:", ps, end="\t")
215:             print("cost:", cost, end="\t")
216:             print("best cost:", best_cost)
217:     bests = best_m(params, costs, M=M)
218:
219:     for i in range(iterations - 1):
220:         params = []
221:         costs = []
222:         it = 0
223:         while it < N and (current_time < max_time or max_time < 0):
224:             it += 1
225:             choice = random.choice(bests[0])
226:             new_params = perturb_percent(choice, percent=perturb_pc, ps=parameters)
227:             new_cost = costf(choice)
228:             params.append(new_params)
229:             it_params.append(new_params)
230:             costs.append(new_cost)
231:             if new_cost < best_cost:
232:                 best_cost = new_cost
233:                 best_params = new_params
234:             it_best_costs.append(best_cost)
235:             it_best_params.append(best_params)
236:             current_time = time.perf_counter() - start_time
237:             times.append(current_time)
238:         bests = best_m(params + list(bests[0]), costs + list(bests[1]), M=M)
239:
240:     return {
241:         "results": {
242:             "best_params": best_params.tolist(),
243:             "best_cost": best_cost,
244:         },
245:         "stats": {
246:             "it_best_costs": it_best_costs,
247:             "it_best_params": list(map(lambda x: x.tolist(), it_best_params)),
248:             "it_params": list(map(lambda x: x.tolist(), it_params)),
249:             "time": times,
250:         }
251:     }
252:
253:
254: if __name__ == "__main__":
255:     # costf = lib.f_real
256:     # parameters=[{"min": 0, "max": 20}, {"min": 0, "max": 20}]
257:     # N=1000
258:     # out = b(costf=costf, iterations=30, parameters=parameters, N=N, M=300, debug=False, alpha=5)
259:     # print(out['results']['best_params'])
260:
261:     x = np.array([0, 0])
262:     print(x, perturb_percent(x, percent=0.5, ps=[{'min': 0, 'max': 20}, {'min': 0, 'max': 20}])))
```

```
1: import sympy as sp
2: import numpy as np
3: import functools
4:
5: x, y = sp.symbols('x y', real=True)
6: f = 3 * (x - 5)**4 + (10 * ((y - 9)**2))
7: g = sp.Max(x - 5, 0) + (10 * sp.Abs(y - 9))
8: relu = sp.Max(x, 0)
9:
10: def f_real(xv):
11:     return 3 * (xv[0] - 5)**4 + 10 * (xv[1] - 9)**2
12:
13: f_diff_x = f.diff(x)
14: f_diff_y = f.diff(y)
15: def f_grad(xv):
16:     return np.array([
17:         f_diff_x.subs(x, xv[0]).subs(y, xv[1]),
18:         f_diff_y.subs(x, xv[0]).subs(y, xv[1]),
19:     ])
20:
21: g_diff_x = f.diff(x)
22: g_diff_y = f.diff(y)
23: def g_grad(xv):
24:     return np.array([
25:         g_diff_x.subs(x, xv[0]).subs(y, xv[1]),
26:         g_diff_y.subs(x, xv[0]).subs(y, xv[1]),
27:     ])
28:
29: def g_real(xv):
30:     return np.maximum(xv[0] - 5, 0) + 10 * np.abs(xv[1] - 9)
31:
32:
33: def apply_sym(x, f):
34:     for x_sym, x_val in zip(f.free_symbols, x):
35:         f = f.subs(x_sym, x_val)
36:     return f
37:
38: config = {
39:     "f": {
40:         "sym": f,
41:         "real": f_real,
42:         "name": "f",
43:     },
44:     "g": {
45:         "sym": g,
46:         "real": g_real,
47:         "name": "g",
48:     },
49:     "relu": {
50:         "sym": relu,
51:         "real": lambda x: max(x, 0),
52:         "name": "relu",
53:     }
54: }
55:
56: class GradientDescent():
57:     def __init__(self):
58:         self._max_iter = 1000
59:         self._debug = False
60:         self._converged = lambda x1, x2: False
61:         self._epsilon = 0.0001
62:         self._dimension = None
63:         self._beta = 0
64:         self._algorithm = None
65:         self._iteration = None
66:         self._function = None
67:         self._sum = None
68:         self._x_value = None
69:         self._step_coeff = None
70:         self._converged_value = None
71:         self._grad_value = None
72:         self._m = None
73:         self._v = None
74:         self._adam_grad = None
75:         self._beta = None
76:         self._beta2 = None
77:         self._step_size = None
78:         self._z = None
79:         self._f_star = None
80:
81:     def step_size(self, a):
82:         self._step_size = a
83:         return self
84:
85:     def beta(self, b):
86:         self._beta = b
87:         return self
88:
89:     def beta2(self, b):
90:         self._beta2 = b
91:         return self
92:
93:     def epsilon(self, e):
94:         self._epsilon = e
95:         return self
96:
97:     def function(self, f, function_name=None, dimension=None):
98:         self._function = f
99:         self.function_name = function_name
100:         self._dimension = dimension
```

```
101:         return self
102:
103:     def sym_function(self, function, function_name=None):
104:         self.function_name = function_name
105:         self._dimension = len(function.free_symbols)
106:         def fn(x):
107:             return apply_sym(x, function)
108:
109:         diffs = [function.diff(var) for var in function.free_symbols]
110:
111:         def grad(x):
112:             return np.array([
113:                 apply_sym(x, diff) for diff in diffs])
114:
115:         self._function = fn
116:         self._gradient = grad
117:         return self
118:
119:     def gradient(self, g):
120:         self._gradient = g
121:         return self
122:
123:     def max_iter(self, m):
124:         self._max_iter = m
125:         return self
126:
127:     def start(self, s):
128:         self._start = s
129:         return self
130:
131:     def debug(self, d):
132:         self._debug = d
133:         return self
134:
135:     def converged(self, c):
136:         self._converged = c
137:         return self
138:
139:     def set_iterate(self, f):
140:         self.iterate = functools.partial(f, self)
141:         return self
142:
143:     def algorithm(self, alg):
144:         self._algorithm = alg
145:         if self._algorithm == "rmsprop":
146:             import rmsprop
147:             self.set_iterate(rmsprop.iterate)
148:         elif self._algorithm == "adam":
149:             import adam
150:             self.set_iterate(adam.iterate)
151:         elif self._algorithm == "heavy_ball":
152:             import heavy_ball
153:             self.set_iterate(heavy_ball.iterate)
154:         else:
155:             raise Exception("Unknown algorithm:" + alg)
156:         return self
157:
158:     def state_dict(self):
159:         print(self._function(self._x_value))
160:         return {
161:             "alg": self._algorithm,
162:             "function_name": self.function_name,
163:             "iteration": self._iteration,
164:             "step_coeff": self._step_coeff,
165:             "adam_grad": self._adam_grad,
166:             "f(x)": self._function(self._x_value),
167:             "epsilon": self._epsilon,
168:             "converged": self._converged_value,
169:             "gradient": self._grad_value,
170:             "m": self._m,
171:             "v": self._v,
172:             "beta1": self._beta,
173:             "beta2": self._beta2,
174:             "alpha": self._step_size,
175:             "sum": self._sum,
176:             "z": self._z,
177:             **{"x" + str(i): self._x_value[i] for i in range(len(self._x_value))},
178:         }
179:
180:     def run2csv(self, fname, summarise=True):
181:         import pandas as pd
182:         iterations = list(self.iterate())
183:         df = pd.DataFrame(iterations)
184:         df.to_csv(fname)
185:         if summarise:
186:             with open(fname + ".summary", "w") as f:
187:                 print(f"iterations: {len(df)}", file=f)
188:                 print(f"start: {df['x0'][0]} {df['x1'][0]}", file=f)
189:                 print(f"final: {df['x0'][len(df) - 1]} {df['x1'][len(df) - 1]}", file=f)
190:
191:
192: if __name__ == "__main__":
193:     print(f.diff(x), f.diff(y))
194:     print(g.diff(x), g.diff(y))
```

```
1: import numpy as np
2:
3: def iterate(self):
4:     self._x_value = self._start
5:     self._old_x_value = None
6:     self._f_star = 0
7:     self._iteration = 0
8:     self._converged_value = False
9:     self._grad_value = self._gradient(self._x_value)
10:
11:     yield self.state_dict()
12:
13:     while not self._converged_value:
14:         if self._max_iter > 0 and self._iteration > self._max_iter:
15:             break
16:         numerator = self._function(self._x_value) - self._f_star
17:         self._grad_value = self._gradient(self._x_value)
18:         denominator = np.dot(self._grad_value, self._grad_value) # sum of element-wise products
19:         self._old_x_value = self._x_value
20:         step = numerator/denominator
21:         self._x_value = self._x_value - step * self._grad_value
22:         self._converged_value = self._converged(self._x_value, self._old_x_value)
23:         yield self.state_dict()
```

```
1: import even_samples
2: import cifar_costf
3: import numpy as np
4: import keras
5:
6: # (x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
7: # x_train = x_train.astype("float32") / 255
8: # x_test = x_test.astype("float32") / 255
9: # num_classes = 10
10: # y_train = keras.utils.to_categorical(y_train, num_classes)
11: # y_test = keras.utils.to_categorical(y_test, num_classes)
12:
13: (x_train, y_train), (x_test, y_test)= even_samples.even_sample_categories(1000)
14: print(cifar_costf.costf(np.array([128, 0.001, 0.9, 0.999, 40]), (x_train,y_train), (x_test,y_test)))
```



```
src/sgd.py Mon Apr 08 14:24:49 2024 1
1: import numpy as np
2: import functools
3: import gd
4: import week6
5:
6: class StochasticGradientDescent(gd.GradientDescent):
7:     def __init__(self):
8:         self._iteration = 0
9:         self._max_iter = 1000
10:         self._converged = lambda x1, x2: False
11:         self._epsilon = 0.0001
12:         self._f_star = 0
13:         self._debug = False
14:         self._beta = 0
15:         self._function_generator = None
16:         self._dimension = None
17:         self._algorithm = None
18:         self._function = None
19:         self._sum = None
20:         self._x_value = None
21:         self._old_x_value = None
22:         self._step_coeff = None
23:         self._converged_value = None
24:         self._grad_value = None
25:         self._m = None
26:         self._v = None
27:         self._adam_grad = None
28:         self._beta = None
29:         self._beta2 = None
30:         self._step_size = None
31:         self._z = None
32:
33:     def adam_step(self):
34:         self._function, self._gradient = next(self._function_generator)
35:         if self._function == "finished":
36:             return False # did not complet step
37:         self._grad_value = self._gradient(self._x_value)
38:         self._m = self._beta * self._m + (1-self._beta)*self._grad_value
39:         # grad_value * grad_value gives element-wise product of np array
40:         self._v = self._beta2 * self._v + (1-self._beta2) * (self._grad_value*self._grad_value)
41:         self._old_x_value = self._x_value
42:         self._iteration += 1
43:         m_hat = self._m / (1-(self._beta ** self._iteration))
44:         v_hat = np.array(self._v / (1-(self._beta2 ** self._iteration)))
45:         v_hat_aug = v_hat**(0.5) + self._epsilon
46:         self._adam_grad = m_hat / v_hat_aug
47:         self._x_value = self._x_value - self._step_size * self._adam_grad
48:         return True
49:
50:     def polyak_step(self):
51:         self._function, self._gradient = next(self._function_generator)
52:         if self._function == "finished":
53:             return False # did not complet step
54:         self._iteration += 1
55:         numerator = self._function(self._x_value) - self._f_star
56:         self._grad_value = self._gradient(self._x_value)
57:         denominator = np.dot(self._grad_value, self._grad_value) # sum of element-wise products
58:         if denominator == 0.0:
59:             # do nothing this step (hope for non-zero on next mini-batch)
60:             return False
61:         self._old_x_value = self._x_value
62:         step = numerator/denominator
63:         self._x_value = self._x_value - step * self._grad_value
64:         self._converged_value = self._converged(self._x_value, self._old_x_value)
65:         return True # completed step
66:
67:     def constant_step(self):
68:         self._function, self._gradient = next(self._function_generator)
69:         if self._function == "finished":
70:             return False # did not complete step
71:         self._iteration += 1
72:         self._grad_value = self._gradient(self._x_value)
73:         self._old_x_value = self._x_value
74:         self._x_value = self._x_value - self._step_size * self._grad_value
75:         return True # completed step
76:
77:     def rmsprop_step(self):
78:         self._function, self._gradient = next(self._function_generator)
79:         if self._function == "finished":
80:             return False
81:         self._iteration += 1
82:         self._grad_value = self._gradient(self._x_value)
83:         self._old_x_value = self._x_value
84:         self._x_value = self._x_value - self._alpha_n * self._grad_value
85:         self._sum = self._beta * self._sum + (1-self._beta) * (self._grad_value**2)
86:         self._alpha_n = self._step_size / (self._sum**0.5+self._epsilon)
87:         self._step_coeff = self._alpha_n
88:         return True
89:
90:
91:     def heavy_ball_step(self):
92:         self._function, self._gradient = next(self._function_generator)
93:         if self._function == "finished":
94:             return False
95:         self._iteration += 1
96:         self._grad_value = self._gradient(self._x_value)
97:         self._old_x_value = self._x_value
98:         self._z = self._beta * self._z + self._step_size * self._grad_value
99:         self._x_value = self._x_value - self._z
100:         return True
```

```
101:
102:     # pass a function which generates the function to be evaluated,
103:     # e.g. with different minibatches at each iteration
104:     def function_generator(self, fg):
105:         self._function_generator = fg
106:         return self
107:
108:     def alg(self, a):
109:         if a == "constant":
110:             self.step = self.constant_step
111:         elif a == "polyak":
112:             self.step = self.polyak_step
113:         elif a == "rmsprop":
114:             self.step = self.rmsprop_step
115:             if self._step_size is None:
116:                 raise Exception("Need step_size to initialize rmsprop")
117:             if self._x_value is None:
118:                 raise Exception("Need start/x_value to initialize rmsprop")
119:             self._sum = np.zeros(self._x_value.shape)
120:             self._alpha_n = np.zeros(self._x_value.shape)
121:             self._alpha_n.fill(self._step_size)
122:         elif a == "adam":
123:             self.step = self.adam_step
124:             if self._x_value is None:
125:                 raise Exception("Need start/x_value to initialize rmsprop")
126:             self._m = np.zeros(self._x_value.shape, dtype=np.float64)
127:             self._v = np.zeros(self._x_value.shape, dtype=np.float64)
128:         elif a == "heavy_ball":
129:             self.step = self.heavy_ball_step
130:             self._z = 0
131:         else:
132:             raise Exception(f"Alg {a} NYI")
133:         self.function_name = a
134:         return self
135:
136:     def f_star(self, f_st):
137:         self._f_star = f_st
138:         return self
139:
```

```
1: import tensorflow as tf
2: from tensorflow import keras
3: from tensorflow.keras import layers, regularizers
4: from keras.layers import Dense, Dropout, Activation, Flatten, BatchNormalization
5: from keras.layers import Conv2D, MaxPooling2D, LeakyReLU
6: from sklearn.metrics import confusion_matrix, classification_report
7: from sklearn.utils import shuffle
8: import matplotlib.pyplot as plt
9: plt.rc('font', size=18)
10: plt.rcParams['figure.constrained_layout.use'] = True
11: import sys
12: import even_samples
13: import keras
14: import math
15:
16: (x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
17: # x_train, y_train = even_samples.even_sample_categories(num_samples_per_class=math.floor(100/10))
18: n=5000
19: x_train = x_train[1:n]; y_train=y_train[1:n]
20: num_classes = 10
21: input_shape = (32, 32, 3)
22:
23: # Scale images to the [0, 1] range
24: x_train = x_train.astype("float32") / 255
25: x_test = x_test.astype("float32") / 255
26:
27: # convert class vectors to binary class matrices
28: y_train = keras.utils.to_categorical(y_train, num_classes)
29: y_test = keras.utils.to_categorical(y_test, num_classes)
30:
31: model = keras.Sequential()
32: model.add(Conv2D(16, (3,3), padding='same', input_shape=x_train.shape[1:], activation='relu'))
33: model.add(Conv2D(16, (3,3), strides=(2,2), padding='same', activation='relu'))
34: model.add(Conv2D(32, (3,3), padding='same', activation='relu'))
35: model.add(Conv2D(32, (3,3), strides=(2,2), padding='same', activation='relu'))
36: model.add(Dropout(0.5))
37: model.add(Flatten())
38: model.add(Dense(num_classes, activation='softmax', kernel_regularizer=regularizers.l1(0.0001)))
39: model.compile(loss="categorical_crossentropy", optimizer='adam', metrics=["accuracy"])
40: model.summary()
41:
42: batch_size = 128
43: epochs = 20
44: history = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_split=0.1)
45: model.save("cifar.model")
46: plt.subplot(211)
47: plt.plot(history.history['accuracy'])
48: plt.plot(history.history['val_accuracy'])
49: plt.title('model accuracy')
50: plt.ylabel('accuracy')
51: plt.xlabel('epoch')
52: plt.legend(['train', 'val'], loc='upper left')
53: plt.subplot(212)
54: plt.plot(history.history['loss'])
55: plt.plot(history.history['val_loss'])
56: plt.title('model loss')
57: plt.ylabel('loss'); plt.xlabel('epoch')
58: plt.legend(['train', 'val'], loc='upper left')
59: plt.show()
```

```
1: import matplotlib.pyplot as plt
2: import numpy as np
3: import sys
4:
5:
6: def f(x, y):
7:     return 3 * (x - 5)**4 + 10 * (y - 9)**2
8:
9:
10: def g(x, y):
11:     return np.maximum(x - 5, 0) + 10 * np.abs(y - 9)
12:
13:
14: def main(outfile):
15:     x = np.linspace(0, 10, 400)
16:     y = np.linspace(0, 18, 400)
17:     X, Y = np.meshgrid(x, y)
18:     Z_f = f(X, Y)
19:     Z_g = g(X, Y)
20:
21:     fig = plt.figure(figsize=(12, 6))
22:
23:     ax = fig.add_subplot(1, 2, 1, projection='3d')
24:     ax.plot_surface(X, Y, Z_f, cmap='viridis')
25:     ax.set_title('$f(x, y)$')
26:     ax.set_xlabel('$x$')
27:     ax.set_ylabel('$y$')
28:     ax.set_zlabel('$f(x, y)$')
29:
30:     ax = fig.add_subplot(1, 2, 2, projection='3d')
31:     ax.plot_surface(X, Y, Z_g, cmap='magma')
32:     ax.set_title('$g(x, y)$')
33:     ax.set_xlabel('$x$')
34:     ax.set_ylabel('$y$')
35:     ax.set_zlabel('$g(x, y)$')
36:
37:     plt.savefig(outfile)
38:     plt.show()
39:
40: def main_contour(outfile):
41:     x = np.linspace(0, 10, 400)
42:     y = np.linspace(0, 18, 400)
43:     X, Y = np.meshgrid(x, y)
44:     Z_f = f(X, Y)
45:     Z_g = g(X, Y)
46:
47:     fig = plt.figure(figsize=(12, 6))
48:
49:     ax = fig.add_subplot(1, 2, 1)
50:     ax.contourf(X, Y, Z_f, levels=30, cmap='viridis')
51:     ax.set_title('$f(x, y)$')
52:     ax.set_xlabel('$x$')
53:     ax.set_ylabel('$y$')
54:     # ax.set_zlabel('$f(x, y)$')
55:
56:     ax = fig.add_subplot(1, 2, 2)
57:     ax.contourf(X, Y, Z_g, levels=30, cmap='viridis')
58:     ax.set_title('$g(x, y)$')
59:     ax.set_xlabel('$x$')
60:     ax.set_ylabel('$y$')
61:     # ax.set_zlabel('$g(x, y)$')
62:
63:     plt.savefig(outfile)
64:     plt.show()
65:
66:
67: if __name__ == "__main__":
68:     if len(sys.argv) != 2:
69:         print("Usage: python script.py <output_file>")
70:         sys.exit(1)
71:
72:     outfile = sys.argv[1]
73:     main_contour(outfile)
74:
```

```
1: import numpy as np
2: import sympy as sp
3:
4: current_minibatch = None
5:
6: def generate_trainingdata(m=25):
7:     return np.array([0,0]) + 0.25 * np.random.randn(m,2)
8:
9:
10: def f(x, minibatch):
11:     # loss function sum_{w in training data} f(x,w)
12:     y = 0
13:     count = 0
14:     for w in minibatch:
15:         z = x - w - 1
16:         left = 10 * (z[0]**2+z[1]**2)
17:         right = (z[0]+2)**2+(z[1]+4)**2
18:         y = y + min(left, right)
19:         count = count + 1
20:     return y/count
21:
22:
23: def gradient_function_fd(minibatch, epsilon=10**(-15)):
24:     def gradient_fd(x):
25:         dydx1 = (f(x + np.array([epsilon, 0]), minibatch) - f(x, minibatch)) / epsilon
26:         dydx2 = (f(x + np.array([0, epsilon]), minibatch) - f(x, minibatch)) / epsilon
27:         return np.array([dydx1, dydx2])
28:     return gradient_fd
29:
30: def sympy_loss(minibatch):
31:     x1, x2 = sp.symbols('x1 x2', real=True)
32:     function = 0
33:     for w in minibatch:
34:         z1 = x1 - w[0] - 1
35:         z2 = x2 - w[1] - 1
36:         left = 10 * (z1**2 + z2**2)
37:         right = (z1 + 2)**2 + (z2 + 4)**2
38:         function = sp.Min(left, right) + function
39:     function = function / len(minibatch)
40:     return function
41:
42: def gradient_function(minibatch):
43:     function = sympy_loss(minibatch)
44:     def gradient(x):
45:         dydx1 = function.diff(x1)
46:         dydx2 = function.diff(x2)
47:         return np.array([
48:             dydx1.subs(x1, x[0]).subs(x2, x[1]),
49:             dydx2.subs(x1, x[0]).subs(x2, x[1]),
50:         ])
51:
52:     return gradient
53:
54:
55: def loss(x, w):
56:     z = x - w - 1
57:     left = 10 * (z[0]**2+z[1]**2)
58:     right = (z[0]+2)**2+(z[1]+4)**2
59:     return min(left, right)
60:
61:
62: def f_clear(x, minibatch):
63:     return sum(loss(x, w) for w in minibatch) / len(minibatch)
64:
65:
66: def generate_minibatches(T, N=5, seed=42, shuffle=True,):
67:     global current_minibatch
68:     if shuffle:
69:         T = T.copy()
70:         if seed:
71:             np.random.seed(seed)
72:             np.random.shuffle(T)
73:     num_rows = T.shape[0]
74:     i = 0
75:
76:     minibatch = np.zeros((N, T.shape[1]), T.dtype)
77:     while True:
78:         for j in range(N):
79:             minibatch[j] = T[i % num_rows]
80:             i += 1
81:         if shuffle and i >= num_rows:
82:             # begin next epoch
83:             np.random.shuffle(T)
84:             i = 0
85:         current_minibatch = minibatch
86:         yield minibatch
87:
88:
89: def generate_optimisation_functions(batch, minibatch_size=5, finite_difference=True, **kwargs):
90:     minibatch_generator = generate_minibatches(
91:         batch, N=minibatch_size, **kwargs)
92:     for minibatch in minibatch_generator:
93:         def optim_func(x):
94:             return f_clear(x, minibatch)
95:         gradf = None
96:         if finite_difference:
97:             gradf = gradient_function_fd(minibatch)
98:         else:
99:             gradf = gradient_function(minibatch)
100:         yield (optim_func, gradf)
```

```
101:         yield "finished"
102:
103:
104: if __name__ == "__main__":
105:     import os
106:     os.makedirs("data", exist_ok=True)
107:     T = generate_trainingdata()
108:     import pandas as pd
109:     df = pd.DataFrame(T)
110:     df.to_csv("data/T.csv", index=False)
111:
112:     x = np.array([3, 3])
```

```
1: import tensorflow as tf
2: from tensorflow import keras
3: from tensorflow.keras import layers, regularizers
4: from keras.layers import Dense, Dropout, Activation, Flatten, BatchNormalization
5: from keras.layers import Conv2D, MaxPooling2D, LeakyReLU
6: from sklearn.metrics import confusion_matrix, classification_report
7: from sklearn.utils import shuffle
8: import matplotlib.pyplot as plt
9: plt.rc('font', size=18)
10: plt.rcParams['figure.constrained_layout.use'] = True
11: import sys
12:
13: # Model / data parameters
14: num_classes = 10
15: input_shape = (32, 32, 3)
16:
17: # the data, split between train and test sets
18: (x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
19: n=5000
20: x_train = x_train[1:n]; y_train=y_train[1:n]
21: #x_test=x_test[1:500]; y_test=y_test[1:500]
22:
23: # Scale images to the [0, 1] range
24: x_train = x_train.astype("float32") / 255
25: x_test = x_test.astype("float32") / 255
26: print("orig x_train shape:", x_train.shape)
27:
28: # convert class vectors to binary class matrices
29: y_train = keras.utils.to_categorical(y_train, num_classes)
30: y_test = keras.utils.to_categorical(y_test, num_classes)
31:
32: use_saved_model = False
33: if use_saved_model:
34:     model = keras.models.load_model("cifar.model")
35: else:
36:     model = keras.Sequential()
37:     model.add(Conv2D(16, (3,3), padding='same', input_shape=x_train.shape[1:], activation='relu'))
38:     model.add(Conv2D(16, (3,3), strides=(2,2), padding='same', activation='relu'))
39:     model.add(Conv2D(32, (3,3), padding='same', activation='relu'))
40:     model.add(Conv2D(32, (3,3), strides=(2,2), padding='same', activation='relu'))
41:     model.add(Dropout(0.5))
42:     model.add(Flatten())
43:     model.add(Dense(num_classes, activation='softmax', kernel_regularizer=regularizers.l1(0.0001)))
44:     model.compile(loss="categorical_crossentropy", optimizer='adam', metrics=["accuracy"])
45:     model.summary()
46:
47:     batch_size = 128
48:     epochs = 20
49:     history = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_split=0.1)
50:     model.save("cifar.model")
51:     plt.subplot(211)
52:     plt.plot(history.history['accuracy'])
53:     plt.plot(history.history['val_accuracy'])
54:     plt.title('model accuracy')
55:     plt.ylabel('accuracy')
56:     plt.xlabel('epoch')
57:     plt.legend(['train', 'val'], loc='upper left')
58:     plt.subplot(212)
59:     plt.plot(history.history['loss'])
60:     plt.plot(history.history['val_loss'])
61:     plt.title('model loss')
62:     plt.ylabel('loss'); plt.xlabel('epoch')
63:     plt.legend(['train', 'val'], loc='upper left')
64:     plt.show()
65:
66: preds = model.predict(x_train)
67: y_pred = np.argmax(preds, axis=1)
68: y_train1 = np.argmax(y_train, axis=1)
69: print(classification_report(y_train1, y_pred))
70: print(confusion_matrix(y_train1, y_pred))
71:
72: preds = model.predict(x_test)
73: y_pred = np.argmax(preds, axis=1)
74: y_test1 = np.argmax(y_test, axis=1)
75: print(classification_report(y_test1, y_pred))
76: print(confusion_matrix(y_test1, y_pred))
```