

Week 2 Optimisation for Machine Learning

Neimhin Robinson Gunning, 16321701

March 11, 2024

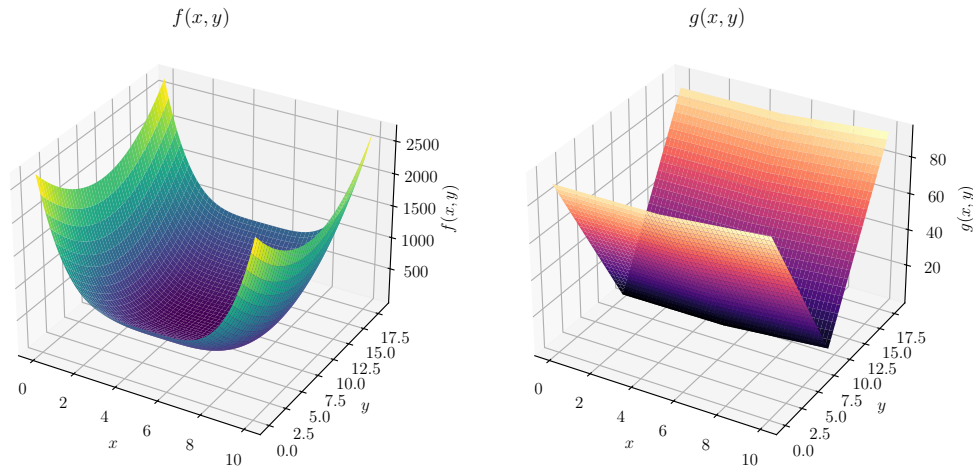
funcs.txt**Wed Feb 21 15:03:56 2024****1**function: $3 * (x-5)^4 + 10 * (y-9)^2$ function: $\text{Max}(x-5, 0) + 10 * |y-9|$ Figure 1: Two bivariate functions downloaded from <https://www.scss.tcd.ie/Doug.Leith/CS7DS2/week4.php>

Figure 2

Let

$$f(x, y) = 3(x - 5)^4 + 10(y - 9)^2 \quad (1)$$

and

$$g(x, y) = \max(x - 5, 0) + 10|y - 9| \quad (2)$$

Using sympy we find the derivatives:

$$\nabla f = \left[\frac{df}{dx}, \frac{df}{dy} \right] = [12(x - 5)^3, 20y - 180]$$

$$\nabla g = \left[\frac{dg}{dx}, \frac{dg}{dy} \right] = [\text{Heaviside}(x - 5), 10\text{sign}(y - 9)]$$

Clearly, the minimum of both $f(x, y)$ and $g(x, y)$ is 0 and they are both minimized by $x = 5, y = 9$.

The Polyak step size is

$$\alpha_{\text{Polyak}} = \frac{f(x) - f^*}{\nabla f(x)^T \nabla f(x)} \quad (3)$$

where x is the parameter vector, $f(x)$ is the function to optimise, and $f^* \approx \min_x f(x)$.

Listing 1: A python function to calculate the Polyak step size on a `sympy` function.

src/polyak_step_size.py

Wed Feb 14 15:12:30 2024

1

```
1: import numpy as np
2:
3:
4: def polyak_step_size(self, sp_func, sp_x, x, f_star):
5:     assert len(sp_x) == len(x)
6:     subs = {sp_xi: xi for sp_xi, xi in zip(sp_x, x)}
7:     fx = sp_func.subs(subs)
8:     grad = [sp_func.diff(sp_xi).subs(subs) for sp_xi in sp_x]
9:     grad = np.array(grad)
10:    denominator = sum(grad * grad)
11:    numerator = fx - f_star
12:    return numerator / denominator
```

```

1: import lib
2: import numpy as np
3:
4:
5: def iterate(self):
6:     x_value = self._start
7:     old_x_value = None
8:     iteration = 0
9:     m = np.zeros(x_value.shape, dtype=np.float64)
10:    v = np.zeros(x_value.shape, dtype=np.float64)
11:    converged = False
12:    grad_value = self._gradient(x_value)
13:
14:    def yielded():
15:        print(x_value)
16:        print(iteration)
17:        return {
18:            "iteration": iteration,
19:            "x": x_value,
20:            "f(x)": self._function(x_value),
21:            "sum": sum,
22:            "epsilon": self._epsilon,
23:            "converged": converged,
24:            "gradient": grad_value,
25:            "m": m,
26:            "v": v,
27:            "beta1": self._beta,
28:            "beta2": self._beta2,
29:            "alpha": self._step_size,
30:        }
31:
32:    yield yielded()
33:
34:    while not converged:
35:        if self._max_iter > 0 and iteration > self._max_iter:
36:            break
37:        grad_value = self._gradient(x_value)
38:        m = self._beta * m + (1-self._beta)*grad_value
39:        # grad_value * grad_value gives element-wise product of np array
40:        v = self._beta2 * v + (1-self._beta2) * (grad_value*grad_value)
41:        old_x_value = x_value
42:        iteration += 1
43:        m_hat = m / (1-(self._beta ** iteration))
44:        v_hat = np.array(v / (1-(self._beta2 ** iteration)))
45:        print('v', v, type(v))
46:        print('v_hat', v_hat, type(v_hat))
47:        print(np, type(np))
48:        v_hat_aug = v_hat**(0.5) + self._epsilon
49:        adam_grad = m_hat / v_hat_aug
50:        x_value = x_value - self._step_size * adam_grad
51:        converged = self._converged(x_value, old_x_value)
52:        yield yielded()
53:
54:
55: if __name__ == "__main__":
56:     adam = lib.GradientDescent()
57:     adam.epsilon(0.0001)
58:     adam.step_size(10**(-2))
59:     adam.beta(0.8)
60:     adam.beta2(0.9)
61:     adam.max_iter(-1)
62:     adam.start(np.array([0, 0]))
63:
64:     def converged(x1, x2):
65:         d = np.max(x1-x2)
66:         return d < 0.000001
67:
68:     def fn(x):
69:         return lib.f.subs(lib.x, x[0]).subs(lib.y, x[1])
70:
71:     def grad(x):
72:         return np.array(
73:             [lib.f.diff(var).subs(lib.x, x[0]).subs(lib.y, x[1])
74:              for var in (lib.x, lib.y)])
75:     adam.converged(converged)
76:     adam.function(fn)
77:     adam.gradient(grad)
78:     adam.set_iterate(iterate)
79:     adam.run2csv("adam.csv")

```

```
1: from sympy import symbols, diff, solve
2: import sympy as sp
3:
4: # Define the symbolic variables
5: x, y = symbols('x y', real=True)
6:
7: # Define the functions
8: f = 3 * (x - 5)**4 + 10 * (y - 9)**2
9: g = sp.Max(x - 5, 0) + 10 * sp.Abs(y - 9)
10:
11: grad_f = [diff(f, var) for var in (x, y)]
12: argmin_f = solve(grad_f, (x, y))
13: print(f"Argmin of f(x, y): {argmin_f}")
14:
15: grad_g = [diff(g, var) for var in (x, y)]
16: argmin_g = solve(grad_g, (x, y))
17: print(f"Argmin of g(x, y): {argmin_g}")
```

```
1: import sympy as sp
2: import sys
3: import numpy as np
4: import matplotlib.pyplot as plt
5: from matplotlib.pyplot import cm
6: import seaborn as sns
7: import pandas as pd
8: from lib import GradientDescent
9:
10: LINEWIDTH = 0.5
11: x = sp.symbols('x')
12: y = x**4
13: dydx = y.diff()
14:
15: fig, ax = plt.subplots(1, 3, figsize=(12, 8))
16:
17: blowup = 0.8
18:
19: # alpha * (blowup ** 3) * 4 = 1.2
20:
21: results = {
22:     "alpha": [],
23:     "start": [],
24:     "convergence time": [],
25:     "final guess": [],
26: }
27: iota = 0.000000000001
28: settings = [
29:     (0.1, 1),
30:     (0.03, 1),
31:     (0.5, 1),
32:     (0.25, 1),
33:     ((2*blowup)/((blowup**3)*4) + iota, blowup),
34:     ((2*blowup)/((blowup**3)*4) - iota, blowup),
35:     (0.05, 0.7),
36:     (0.1, 0.7),
37:     (0.15, 0.7),
38:     (0.1, 2),
39: ]
40: color = cm.rainbow(np.linspace(0, 1, len(settings)))
41: settings_with_color = zip(settings, color)
42: for (step_size, start), color in settings_with_color:
43:     print(step_size, start, color)
44:     g = GradientDescent()
45:     g.max_iter(100)
46:     g.step_size(step_size)
47:     g.start(start)
48:     g.function(lambda x1: float(y.subs(x, x1)))
49:     y_diff = y.diff()
50:     g.gradient(lambda x1: float(y_diff.subs(x, x1)))
51:     g.debug(True)
52:
53:     def is_inf(x):
54:         import math
55:         if x == math.inf or x == -math.inf:
56:             return True
57:
58:     def converged(x1, x2):
59:         if is_inf(x1) or is_inf(x2):
60:             return True
61:         abs = np.abs(x1-x2)
62:         print(abs, x1, x2)
63:         return abs < 0.001
64:     g.converged(converged)
65:     iterations, estimates, y_of_x = zip(*[
66:         (x[0], x[1], x[2]) for x in g.iterate()])
67:     results["alpha"].append(step_size)
68:     results["start"].append(start)
69:     results["convergence time"].append(len(iterations))
70:     results["final guess"].append(estimates[-1])
71:     print('y_of_x', y_of_x)
72:     print('iterations', iterations)
73:     print('estimates', estimates)
74:     sns.lineplot(
75:         x=iterations,
76:         y=np.abs(np.array(estimates)),
77:         ax=ax[0],
78:         linewidth=LINEWIDTH,
79:         legend=False,
80:         color=color,
81:         label=f"$\\alpha={step_size}$, $x={start}$")
82:     sns.lineplot(
83:         x=iterations,
84:         y=y_of_x,
85:         ax=ax[1],
86:         linewidth=LINEWIDTH,
87:         color=color,
88:         label=f"$\\alpha={step_size}$, $x={start}$")
89:     ax[2].step(
90:         estimates,
91:         y_of_x,
92:         linewidth=LINEWIDTH,
93:         color=color,
94:         label=f"$\\alpha={step_size}$, $x={start}$")
95:     xs = np.arange(-2, 2, 0.01)
96:     ys = [y.subs(x, xi) for xi in xs]
97:     ax[2].plot(
98:         xs,
99:         ys,
100:         linewidth=LINEWIDTH,
```

```
101:         label="$x^4$",
102:         color='yellow',
103:     )
104:     ax[2].scatter(
105:         start,
106:         g._function(start),
107:         color=color)
108:
109: ax[1].legend(framealpha=1)
110: ax[0].set_ylabel("$|\hat{x}|$")
111: ax[0].set_xlabel("iteration")
112: ax[0].set_yscale('log')
113: ax[1].set_yscale('log')
114: ax[0].set_title("(a)")
115: ax[1].set_ylabel("$y(\hat{x})$")
116: ax[1].set_xlabel("iteration")
117: ax[1].set_title("(b)")
118: ax[2].set_xlabel("$x$")
119: ax[2].set_ylabel("$y$")
120: ax[2].set_title("(c)")
121: ax[0].set_ylim([10**-2, 1.5])
122: ax[1].set_ylim([10**-6, 1.5])
123: ax[2].set_ylim([-0.2, 2.2])
124: ax[2].set_xlim([-2, 2])
125:
126: plt.tight_layout()
127:
128: outfile = "fig/gradient-descent-b1.pdf"
129: if len(sys.argv) > 1:
130:     outfile = sys.argv[1]
131: plt.savefig(outfile)
132: df = pd.DataFrame(results)
133: print(df)
134: df.to_csv("fig/gradient-descent-b1.csv")
```

```
1: import sympy as sp
2: import sys
3: import numpy as np
4: import matplotlib.pyplot as plt
5: import seaborn as sns
6: from lib import GradientDescent
7:
8: LINEWIDTH = 0.7
9: x = sp.symbols('x')
10: y = x**4
11: dydx = y.diff()
12:
13: fig, ax = plt.subplots(1, 2)
14:
15: for step_size in np.array([0.5]):
16:     for start in np.array([1.00001]):
17:         print(step_size, start)
18:         g = GradientDescent()
19:         g.max_iter(100)
20:         g.step_size(step_size)
21:         g.start(start)
22:         g.function(lambda x1: float(y.subs(x, x1)))
23:         y_diff = y.diff()
24:         g.gradient(lambda x1: float(y_diff.subs(x, x1)))
25:         g.debug(True)
26:
27:         def converged(x1, x2):
28:             abs = np.abs(x1-x2)
29:             print(abs, x1, x2)
30:             return abs < 0.001
31:         g.converged(converged)
32:         iterations, estimates, y_of_x = zip(*[(x[0], x[1], x[2]) for x in g.iterate()])
33:         print('y_of_x', y_of_x)
34:         print('iterations', iterations)
35:         print('estimates', estimates)
36:         sns.lineplot(
37:             x=iterations,
38:             y=estimates,
39:             ax=ax[0],
40:             linewidth=LINEWIDTH,
41:             legend=False,
42:             label=f"$\\alpha={step_size}$, $x={start}$")
43:         sns.lineplot(
44:             x=iterations,
45:             y=y_of_x,
46:             ax=ax[1],
47:             linewidth=LINEWIDTH,
48:             label=f"$\\alpha={step_size}$, $x={start}$")
49:
50: ax[0].set_ylabel("estimate of  $\arg\min_x x^4$ ")
51: ax[0].set_xlabel("iteration")
52: ax[1].set_ylabel(" $\hat{y}(x)$ ")
53: ax[1].set_xlabel("iteration")
54: ax[0].set_ylim([-10000, 10000])
55: ax[1].set_ylim([-100, 10000])
56: plt.tight_layout()
57:
58: outfile = "fig/gradient-descent-x^4-crazy.pdf"
59: if len(sys.argv) > 1:
60:     outfile = sys.argv[1]
61: print(outfile)
62: plt.savefig(outfile)
```



```
1: import sympy as sp
2: import sys
3: import numpy as np
4: import matplotlib.pyplot as plt
5: import seaborn as sns
6: import pandas as pd
7: from lib import GradientDescent
8:
9: LINEWIDTH = 0.5
10: x = sp.symbols('x')
11: y = x**4
12: dydx = y.diff()
13:
14: fig, ax = plt.subplots(1, 3, figsize=(12, 4))
15:
16: blowup = 0.8
17:
18: # alpha * (blowup ** 3) * 4 = 1.2
19:
20: results = {
21:     "alpha": [],
22:     "start": [],
23:     "convergence time": [],
24:     "final guess": [],
25: }
26: iota = 0.000000000001
27: for (step_size, start, color) in [
28:     (0.1, 1, 'gray'),
29: ]:
30:     print(step_size, start, color)
31:     g = GradientDescent()
32:     g.max_iter(100)
33:     g.step_size(step_size)
34:     g.start(start)
35:     g.function(lambda x1: float(y.subs(x, x1)))
36:     y_diff = y.diff()
37:     g.gradient(lambda x1: float(y_diff.subs(x, x1)))
38:     g.debug(True)
39:
40:     def is_inf(x):
41:         import math
42:         if x == math.inf or x == -math.inf:
43:             return True
44:
45:     def converged(x1, x2):
46:         if is_inf(x1) or is_inf(x2):
47:             return True
48:         abs = np.abs(x1-x2)
49:         print(abs, x1, x2)
50:         return abs < 0.001
51:     g.converged(converged)
52:     iterations, estimates, y_of_x = zip(*[
53:         (x[0], x[1], x[2]) for x in g.iterate())
54:     results["alpha"].append(step_size)
55:     results["start"].append(start)
56:     results["convergence time"].append(len(iterations))
57:     results["final guess"].append(estimates[-1])
58:     print('y_of_x', y_of_x)
59:     print('iterations', iterations)
60:     print('estimates', estimates)
61:     sns.lineplot(
62:         x=iterations,
63:         y=np.abs(np.array(estimates)),
64:         ax=ax[0],
65:         linewidth=LINEWIDTH,
66:         legend=False,
67:         color=color,
68:         label=f"$\\alpha={step_size}$, $x={start}$")
69:     sns.lineplot(
70:         x=iterations,
71:         y=y_of_x,
72:         ax=ax[1],
73:         linewidth=LINEWIDTH,
74:         color=color,
75:         label=f"$\\alpha={step_size}$, $x={start}$")
76:     ax[2].step(
77:         estimates,
78:         y_of_x,
79:         linewidth=LINEWIDTH,
80:         color=color,
81:         label=f"$\\alpha={step_size}$, $x={start}$")
82:     xs = np.arange(-2, 2, 0.01)
83:     ys = [y.subs(x, xi) for xi in xs]
84:     ax[2].plot(
85:         xs,
86:         ys,
87:         linewidth=LINEWIDTH,
88:         label="$x^4$",
89:         color='yellow',
90:     )
91:     ax[2].scatter(
92:         start,
93:         g._function(start),
94:         color=color)
95:
96:
97: ax[0].set_ylabel("$\\hat{x}$")
98: ax[0].set_xlabel("iteration")
99: ax[0].set_yscale('log')
100: ax[0].set_title("(a)")
```

```
101: ax[1].set_yscale('log')
102: ax[1].set_ylabel("$y(\\hat{x})$")
103: ax[1].set_xlabel("iteration")
104: ax[1].set_title("(b)")
105: ax[2].set_xlabel("$x$")
106: ax[2].set_ylabel("$y$")
107: ax[2].set_title("(c)")
108: # ax[0].set_ylim([-7, 7])
109: # ax[1].set_ylim([-1, 4])
110: ax[2].set_ylim([-0.2, 1.2])
111: # ax[2].set_xlim([-2, 2])
112: plt.tight_layout()
113:
114: outfile = "fig/gradient-descent-bi.pdf"
115: if len(sys.argv) > 1:
116:     outfile = sys.argv[1]
117: plt.savefig(outfile)
118: df = pd.DataFrame(results)
119: print(df)
120: df.to_csv("fig/gradient-descent-bi.csv")
```

```
1: import sympy as sp
2: import sys
3: import numpy as np
4: import matplotlib.pyplot as plt
5: import seaborn as sns
6: import pandas as pd
7: from lib import GradientDescent
8:
9: LINEWIDTH = 0.1
10: x = sp.symbols('x')
11: y = x**4
12: dydx = y.diff()
13:
14: fig, ax = plt.subplots(1, 3, figsize=(12,4))
15:
16: blowup = 0.8
17:
18: # alpha * (blowup ** 3) * 4 = 1.2
19:
20: results = {
21:     "alpha": [],
22:     "start": [],
23:     "gamma": [],
24:     "$f(x)$": [],
25:     "convergence time": [],
26:     "final guess": [],
27: }
28: iota = 0.005
29: def run(gamma, color, max_iter=99, plot=True):
30:     g = GradientDescent()
31:     g.max_iter(max_iter)
32:     alpha = 1
33:     start = 1
34:     g.step_size(alpha)
35:     g.start(start)
36:     y = gamma * (x**2)
37:     g.function(lambda x1: float(y.subs(x, x1)))
38:     y_diff = y.diff()
39:     g.gradient(lambda x1: float(y_diff.subs(x, x1)))
40:     g.debug(True)
41:
42:     def is_inf(x):
43:         import math
44:         if x == math.inf or x == -math.inf:
45:             return True
46:
47:     def converged(x1, x2):
48:         if is_inf(x1) or is_inf(x2):
49:             return True
50:         abs = np.abs(x1-x2)
51:         print(abs, x1, x2)
52:         return abs < 0.001
53:     g.converged(converged)
54:     iterations, estimates, y_of_x = zip(*[
55:         (x[0], x[1], x[2]) for x in g.iterate())])
56:     results["alpha"].append(alpha)
57:     results["gamma"].append(gamma)
58:     results["$f(x)$"].append(str(y))
59:     results["start"].append(start)
60:     results["convergence time"].append(len(iterations))
61:     results["final guess"].append(estimates[-1])
62:     if plot:
63:         sns.lineplot(
64:             x=iterations,
65:             y=estimates,
66:             ax=ax[0],
67:             linewidth=LINEWIDTH,
68:             legend=False,
69:             color=color,
70:             label=f"$\\gamma={gamma}$")
71:         sns.lineplot(
72:             x=iterations,
73:             y=y_of_x,
74:             ax=ax[1],
75:             linewidth=LINEWIDTH,
76:             color=color,
77:             label=f"$\\gamma={gamma}$")
78:         ax[2].step(
79:             estimates,
80:             y_of_x,
81:             linewidth=LINEWIDTH,
82:             color=color,
83:             label=f"$\\gamma={gamma}$")
84:         xs = np.arange(-2, 2, 0.01)
85:         ys = [y.subs(x, xi) for xi in xs]
86:         ax[2].plot(
87:             xs,
88:             ys,
89:             linewidth=LINEWIDTH,
90:             label="$\\gamma x^2$",
91:             color='yellow',
92:         )
93:         ax[2].scatter(
94:             start,
95:             g._function(start),
96:             color=color)
97:
98:
99: for (gamma, color) in [
100:     (0.01, 'green'),
```

```
101:         ( 0.1, 'blue'),
102:         ( 1 - iota, 'black'),
103:         (1 + iota, 'orange'),
104:         ( 1, 'red'),
105:         (-0.05, 'purple'),
106:     ]:
107:         run(gamma, color)
108:
109: run(-1000, 'pink', max_iter=10000, plot=False)
110:
111: ax[0].set_ylabel("$x$")
112: ax[0].set_xlabel("iteration")
113: ax[0].set_title("(a)")
114: ax[1].set_ylabel("$y(\hat{x})$")
115: ax[1].set_xlabel("iteration")
116: ax[1].set_title("(b)")
117: ax[2].set_xlabel("$x$")
118: ax[2].set_ylabel("$y$")
119: ax[2].set_title("(c)")
120: ax[0].set_ylim([-7, 7])
121: ax[1].set_ylim([-1, 4])
122: ax[2].set_ylim([-1, 2.2])
123: ax[2].set_xlim([-2, 2])
124: plt.tight_layout()
125:
126: outfile = "fig/gradient-descent-ci.pdf"
127: if len(sys.argv) > 1:
128:     outfile = sys.argv[1]
129: plt.savefig(outfile)
130: df = pd.DataFrame(results)
131: print(df)
132: df.to_csv("fig/gradient-descent-ci.csv")
```

```
1: #!/usr/bin/env python
2:
3: import pandas as pd
4: import sys
5: import subprocess
6: import os
7:
8:
9: def csv_to_latex_pdf(input_csv, output_pdf="output.pdf"):
10:     # Read the CSV file into a pandas DataFrame
11:     df = pd.read_csv(input_csv, dtype=str)
12:
13:     # Convert the DataFrame to LaTeX tabular format
14:     df_to_latex_pdf(df, output_pdf=output_pdf)
15:
16:
17: def format_float(x):
18:     if isinstance(x, float):
19:         import math
20:         if x == math.inf:
21:             return "$\\infty$"
22:         if x == -math.inf:
23:             return "$-\\infty$"
24:         if x == math.nan:
25:             return "NaN"
26:         return ("\\num{{{0:.2g}}}".format(x))
27:
28:
29: def df_to_latex_pdf(df, output_pdf="output.pdf"):
30:     # Create the tmp directory if it doesn't exist
31:     if not os.path.exists("tmp"):
32:         os.makedirs("tmp")
33:     latex_tabular = df.to_latex(float_format=format_float)
34:
35:     # Wrap the tabular code in a LaTeX document
36:     latex_document = r"""\documentclass{article}
37: \usepackage{booktabs}
38: \usepackage{siunitx}
39: \begin{document}
40: \thispagestyle{empty}
41: """ + latex_tabular + r"""\end{document}"""
42:
43:     output_tex = "tmp/output.tex"
44:
45:     # Save the LaTeX code to a file
46:     with open(output_tex, 'w') as f:
47:         f.write(latex_document)
48:
49:     # Compile the LaTeX file using pdflatex
50:     subprocess.run(["pdflatex", "-jobname=tmp/output", output_tex])
51:     subprocess.run(["pdfcrop", "tmp/output.pdf", output_pdf])
52:
53:     print(f"PDF generated as {output_pdf}")
54:
55:
56: if __name__ == "__main__":
57:     if len(sys.argv) != 3:
58:         print("Usage: python script_name.py input.csv output.pdf")
59:         sys.exit(1)
60:
61:     input_csv = sys.argv[1]
62:     output_pdf = sys.argv[2]
63:     csv_to_latex_pdf(input_csv, output_pdf)
```

```
1: import lib
2: import sys
3: import argparse
4: import numpy as np
5: import rms2
6: import adam
7: import hb
8:
9:
10: def converged(x1, x2):
11:     d = np.max(x1-x2)
12:     return d < 0.001
13:
14:
15: parser = argparse.ArgumentParser(
16:     prog="Run Gradient Descent A Step Size Algorithm")
17:
18: parser.add_argument('-al', '--algorithm', choices=[
19:     'rmsprop', 'adam', 'polyak', 'heavy_ball'], required=True)
20:
21: parser.add_argument('-b', '--beta', type=float)
22: parser.add_argument('-b2', '--beta2', type=float)
23: parser.add_argument('-a', '--alpha', type=float)
24: parser.add_argument('filename')
25:
26: args = parser.parse_args()
27:
28: print(args.filename)
29:
30: gd = lib.GradientDescent()
31: if 0:
32:     pass
33: elif args.algorithm == 'rmsprop':
34:     gd.set_iterate(rms2.iterate)
35: elif args.algorithm == 'adam':
36:     gd.set_iterate(adam.iterate)
37: elif args.algorithm == 'heavy_ball':
38:     gd.set_iterate(hb.iterate)
39: else:
40:     print("no algorithm")
41:     sys.exit(1)
42:
43: gd.start(np.array([4, 8]))
44: gd.converged(converged)
45: gd.step_size(args.alpha)
46: gd.beta(args.beta)
47: gd.beta2(args.beta2)
48: gd.epsilon(0.0001)
49: gd.max_iter(-1)
50:
51:
52: def fn(x):
53:     return lib.f.subs(lib.x, x[0]).subs(lib.y, x[1])
54:
55:
56: def grad(x):
57:     return np.array([
58:         lib.f.diff(var).subs(
59:             lib.x, x[0]
60:         ).subs(
61:             lib.y, x[1]
62:         ) for var in (lib.x, lib.y)])
63:
64:
65: gd.converged(converged)
66: gd.function(fn)
67: gd.gradient(grad)
68: gd.run2csv(args.filename)
```

```
1: class GradientDescent():
2:     # ...
3:     def iterate(self):
4:         import math
5:         x_value = self._start
6:         old_x_value = None
7:         iteration = 0
8:         while True:
9:             yield [iteration, float(x_value), float(self._function(x_value))]
10:            iteration += 1
11:            if self._max_iter > 0 and iteration > self._max_iter:
12:                break
13:            grad_value = self._gradient(x_value)
14:            x_value -= self._step_size * grad_value # Update step
15:            if old_x_value is not None and self._converged(x_value, old_x_value):
16:                yield [iteration, float(x_value), float(self._function(old_x_value))]
17:                print("converged")
18:                break
19:            old_x_value = x_value
```

```
1: import lib
2:
3:
4: def iterate(self):
5:     x_value = self._start
6:     old_x_value = None
7:     iteration = 0
8:     converged = False
9:     grad_value = self._gradient(x_value)
10:
11:     z = 0
12:
13:     def yielded():
14:         print(x_value)
15:         print(iteration)
16:         return {
17:             "iteration": iteration,
18:             "z": z,
19:             "x": x_value,
20:             "f(x)": self._function(x_value),
21:             "sum": sum,
22:             "epsilon": self._epsilon,
23:             "converged": converged,
24:             "gradient": grad_value,
25:         }
26:
27:     yield yielded() # yield initial values
28:
29:     while not converged:
30:         iteration += 1
31:         if self._max_iter > 0 and iteration > self._max_iter:
32:             break
33:         grad_value = self._gradient(x_value)
34:         old_x_value = x_value
35:         z = self._beta * z + self._step_size * grad_value
36:         x_value = x_value - z
37:         converged = self._converged(x_value, old_x_value)
38:         yield yielded()
39:
40:
41: if __name__ == "__main__":
42:     import numpy as np
43:     hb = lib.GradientDescent()
44:     hb.step_size(10**-3)
45:     hb.beta(0.5)
46:     hb.max_iter(-1)
47:     hb.start(np.array([0, 0]))
48:
49:     def converged(x1, x2):
50:         d = np.max(x1-x2)
51:         return d < 0.000001
52:
53:     def fn(x):
54:         return lib.f.subs(lib.x, x[0]).subs(lib.y, x[1])
55:
56:     def grad(x):
57:         return np.array([
58:             lib.f.diff(var).subs(lib.x, x[0]).subs(lib.y, x[1])
59:             for var in (lib.x, lib.y)])
60:     hb.converged(converged)
61:     hb.function(fn)
62:     hb.gradient(grad)
63:     hb.set_iterate(iterate)
64:     hb.run2csv("hb.csv")
```



```
1: import lib
2:
3:
4: def iterate(self):
5:     x_value = self._start
6:     old_x_value = None
7:     iteration = 0
8:     converged = False
9:     grad_value = self._gradient(x_value)
10:
11:     z = 0
12:
13:     def yielded():
14:         print(x_value)
15:         print(iteration)
16:         return {
17:             "iteration": iteration,
18:             "z": z,
19:             "x": x_value,
20:             "f(x)": self._function(x_value),
21:             "sum": sum,
22:             "epsilon": self._epsilon,
23:             "converged": converged,
24:             "gradient": grad_value,
25:         }
26:
27:     yield yielded() # yield initial values
28:
29:     while not converged:
30:         iteration += 1
31:         if self._max_iter > 0 and iteration > self._max_iter:
32:             break
33:         grad_value = self._gradient(x_value)
34:         old_x_value = x_value
35:         z = self._beta * z + self._step_size * grad_value
36:         x_value = x_value - z
37:         converged = self._converged(x_value, old_x_value)
38:         yield yielded()
39:
40:
41: if __name__ == "__main__":
42:     import numpy as np
43:     hb = lib.GradientDescent()
44:     hb.step_size(10**-3)
45:     hb.beta(0.5)
46:     hb.max_iter(-1)
47:     hb.start(np.array([0, 0]))
48:
49:     def converged(x1, x2):
50:         d = np.max(x1-x2)
51:         return d < 0.000001
52:
53:     def fn(x):
54:         return lib.f.subs(lib.x, x[0]).subs(lib.y, x[1])
55:
56:     def grad(x):
57:         return np.array([
58:             lib.f.diff(var).subs(lib.x, x[0]).subs(lib.y, x[1])
59:             for var in (lib.x, lib.y)])
60:     hb.converged(converged)
61:     hb.function(fn)
62:     hb.gradient(grad)
63:     hb.set_iterate(iterate)
64:     hb.run2csv("hb.csv")
```

```
1: import sympy as sp
2: import functools
3:
4: x, y = sp.symbols('x y', real=True)
5: f = 3 * (x - 5)**4 + (10 * ((y - 9)**2))
6: g = sp.Max(x - 5, 0) + (10 * sp.Abs(y - 9))
7:
8:
9: class GradientDescent():
10:     def __init__(self):
11:         self._max_iter = 1000
12:         self._debug = False
13:         self._converged = lambda x1, x2: False
14:         self._epsilon = 0.0001
15:         self._beta = 0
16:
17:     def step_size(self, a):
18:         self._step_size = a
19:         return self
20:
21:     def beta(self, b):
22:         self._beta = b
23:         return self
24:
25:     def beta2(self, b):
26:         self._beta2 = b
27:         return self
28:
29:     def epsilon(self, e):
30:         self._epsilon = e
31:         return self
32:
33:     def function(self, f):
34:         self._function = f
35:         return self
36:
37:     def gradient(self, g):
38:         self._gradient = g
39:         return self
40:
41:     def max_iter(self, m):
42:         self._max_iter = m
43:         return self
44:
45:     def start(self, s):
46:         self._start = s
47:         return self
48:
49:     def debug(self, d):
50:         self._debug = d
51:         return self
52:
53:     def converged(self, c):
54:         self._converged = c
55:         return self
56:
57:     def set_iterate(self, f):
58:         self.iterate = functools.partial(f, self)
59:         return self
60:
61:     def iterate(self):
62:         x_value = self._start
63:         old_x_value = None
64:         iteration = 0
65:         while True:
66:             yield [iteration, x_value, self._function(x_value)]
67:             iteration += 1
68:             if self._max_iter > 0 and iteration > self._max_iter:
69:                 break
70:             grad_value = self._gradient(x_value)
71:             print(x_value, type(x_value))
72:             print(grad_value, type(grad_value))
73:             x_value -= self._step_size * grad_value # Update step
74:             if old_x_value is not None and self._converged(x_value, old_x_value):
75:                 yield [iteration, float(x_value), float(self._function(old_x_value))]
76:                 print("converged")
77:                 break
78:             old_x_value = x_value
79:
80:     def run2csv(self, fname, summarise=True):
81:         import pandas as pd
82:         iterations = list(self.iterate())
83:         df = pd.DataFrame(iterations)
84:         df.to_csv(fname)
85:         if summarise:
86:             with open(fname + ".summary", "w") as f:
87:                 print(f"iterations: {len(df)}", file=f)
88:                 print(f"start: {df['x'][0]}", file=f)
89:                 print(f"final: {df['x'][len(df) - 1]}", file=f)
90:
91:
92: if __name__ == "__main__":
93:     print(f.diff(x), f.diff(y))
94:     print(g.diff(x), g.diff(y))
```

```
1: import numpy as np
2:
3:
4: def polyak_step_size(self, sp_func, sp_x, x, f_star):
5:     assert len(sp_x) == len(x)
6:     subs = {sp_xi: xi for sp_xi, xi in zip(sp_x, x)}
7:     fx = sp_func.subs(subs)
8:     grad = [sp_func.diff(sp_xi).subs(subs) for sp_xi in sp_x]
9:     grad = np.array(grad)
10:    denominator = sum(grad * grad)
11:    numerator = fx - f_star
12:    return numerator / denominator
```

```
1: import lib
2:
3:
4: def iterate(self):
5:     import numpy as np
6:     x_value = self._start
7:     old_x_value = None
8:     iteration = 0
9:     sum = np.zeros(x_value.shape)
10:    alpha_n = np.zeros(x_value.shape)
11:    alpha_n.fill(self._step_size)
12:    converged = False
13:    grad_value = self._gradient(x_value)
14:
15:    def yielded():
16:        print(x_value)
17:        print(iteration)
18:        return {
19:            "iteration": iteration,
20:            "x": x_value,
21:            "f(x)": self._function(x_value),
22:            "sum": sum,
23:            "epsilon": self._epsilon,
24:            "converged": converged,
25:            "gradient": grad_value,
26:            "alpha_n": alpha_n,
27:        }
28:
29:    yield yielded()
30:
31:    while not converged:
32:        iteration += 1
33:        if self._max_iter > 0 and iteration > self._max_iter:
34:            break
35:        grad_value = self._gradient(x_value)
36:        old_x_value = x_value
37:        print(grad_value, type(grad_value))
38:        print(alpha_n, type(alpha_n))
39:        print(x_value, type(x_value))
40:        x_value = x_value - alpha_n * grad_value
41:        sum = self._beta * sum + (1-self._beta) * (grad_value**2)
42:        alpha_n = self._step_size / (sum**0.5+self._epsilon)
43:        converged = self._converged(x_value, old_x_value)
44:        yield yielded()
45:
46:
47: def rms_gradient_descent():
48:     rms = lib.GradientDescent()
49:     rms.set_iterate(iterate)
50:     return rms
51:
52:
53: if __name__ == "__main__":
54:     import numpy as np
55:     rms = lib.GradientDescent()
56:     rms.epsilon(0.0001)
57:     rms.step_size(10**-2)
58:     rms.beta(0.1)
59:     rms.max_iter(-1)
60:     rms.start(np.array([0, 0]))
61:
62:     def converged(x1, x2):
63:         d = np.max(x1-x2)
64:         return d < 0.000001
65:
66:     def fn(x):
67:         return lib.f.subs(lib.x, x[0]).subs(lib.y, x[1])
68:
69:     def grad(x):
70:         return np.array([lib.f.diff(var).subs(lib.x, x[0]).subs(lib.y, x[1]) for var in (lib.x, lib.y)])
71:     rms.converged(converged)
72:     rms.function(fn)
73:     rms.gradient(grad)
74:     rms.set_iterate(iterate)
75:     rms.run2csv("rms2.csv")
```

```
1: class RMSPropGradientDescent():
2:     def __init__(self):
3:         self._max_iter = 1000
4:         self._debug = False
5:         self._converged = lambda x1, x2: False
6:         self._sum = 0
7:         self._epsilon = 0.0001
8:
9:     def step_size(self, a):
10:         self._step_size = a
11:         return self
12:
13:     def epsilon(self, e):
14:         self._epsilon = e
15:         return self
16:
17:     def beta(self, b):
18:         self._beta = b
19:         return self
20:
21:     def function(self, f):
22:         self._function = f
23:         return self
24:
25:     def gradient(self, g):
26:         self._gradient = g
27:         return self
28:
29:     def max_iter(self, m):
30:         self._max_iter = m
31:         return self
32:
33:     def start(self, s):
34:         self._start = s
35:         return self
36:
37:     def debug(self, d):
38:         self._debug = d
39:         return self
40:
41:     def converged(self, c):
42:         self._converged = c
43:         return self
44:
45:     def run2csv(self, fname):
46:         import pandas as pd
47:         iterations = list(self.iterate())
48:         df = pd.DataFrame(iterations)
49:         df.to_csv(fname)
50:
51:     def iterate(self):
52:         import math
53:         import numpy as np
54:         x_value = self._start
55:         old_x_value = None
56:         iteration = 0
57:         sum = np.zeros(x_value.shape)
58:         alpha_n = np.zeros(x_value.shape)
59:         alpha_n.fill(self._step_size)
60:         converged = False
61:         grad_value = self._gradient(x_value)
62:
63:         def yielded():
64:             print(x_value)
65:             print(iteration)
66:             return {
67:                 "iteration": iteration,
68:                 "x": x_value,
69:                 "f(x)": self._function(x_value),
70:                 "sum": sum,
71:                 "epsilon": self._epsilon,
72:                 "converged": converged,
73:                 "gradient": grad_value,
74:                 "alpha_n": alpha_n,
75:             }
76:
77:         yield yielded()
78:
79:         while not converged:
80:             iteration += 1
81:             if self._max_iter > 0 and iteration > self._max_iter:
82:                 break
83:             grad_value = self._gradient(x_value)
84:             old_x_value = x_value
85:             print(grad_value, type(grad_value))
86:             print(alpha_n, type(alpha_n))
87:             print(x_value, type(x_value))
88:             x_value = x_value - alpha_n * grad_value
89:             sum = self._beta * sum + (1-self._beta) * (grad_value**2)
90:             alpha_n = self._step_size / (sum**0.5+self._epsilon)
91:             converged = self._converged(x_value, old_x_value)
92:             yield yielded()
93:
94:
95: if __name__ == "__main__":
96:     import numpy as np
97:     import lib
98:     rms = RMSPropGradientDescent()
99:     rms.epsilon(0.0001)
100:    rms.step_size(10**-2)
```

```
101:     rms.beta(0.1)
102:     rms.max_iter(-1)
103:     rms.start(np.array([0, 0]))
104:
105:     def converged(x1, x2):
106:         d = np.max(x1-x2)
107:         return d < 0.000001
108:
109:     def fn(x):
110:         return lib.f.subs(lib.x, x[0]).subs(lib.y, x[1])
111:
112:     def grad(x):
113:         return np.array([lib.f.diff(var).subs(lib.x, x[0]).subs(lib.y, x[1]) for var in (lib.x, lib.y)])
114:     rms.converged(converged)
115:     rms.function(fn)
116:     rms.gradient(grad)
117:     rms.run2csv("rms.csv")
```

```
1: import lib
2:
3:
4: def iterate(self):
5:     import numpy as np
6:     x_value = self._start
7:     old_x_value = None
8:     iteration = 0
9:     sum = np.zeros(x_value.shape)
10:    alpha_n = np.zeros(x_value.shape)
11:    alpha_n.fill(self._step_size)
12:    converged = False
13:    grad_value = self._gradient(x_value)
14:
15:    def yielded():
16:        print(x_value)
17:        print(iteration)
18:        return {
19:            "iteration": iteration,
20:            "x": x_value,
21:            "f(x)": self._function(x_value),
22:            "sum": sum,
23:            "epsilon": self._epsilon,
24:            "converged": converged,
25:            "gradient": grad_value,
26:            "alpha_n": alpha_n,
27:        }
28:
29:    yield yielded()
30:
31:    while not converged:
32:        iteration += 1
33:        if self._max_iter > 0 and iteration > self._max_iter:
34:            break
35:        grad_value = self._gradient(x_value)
36:        old_x_value = x_value
37:        print(grad_value, type(grad_value))
38:        print(alpha_n, type(alpha_n))
39:        print(x_value, type(x_value))
40:        x_value = x_value - alpha_n * grad_value
41:        sum = self._beta * sum + (1-self._beta) * (grad_value**2)
42:        alpha_n = self._step_size / (sum**0.5+self._epsilon)
43:        converged = self._converged(x_value, old_x_value)
44:        yield yielded()
45:
46:
47: def rms_gradient_descent():
48:     rms = lib.GradientDescent()
49:     rms.set_iterate(iterate)
50:     return rms
51:
52:
53: if __name__ == "__main__":
54:     import numpy as np
55:     rms = lib.GradientDescent()
56:     rms.epsilon(0.0001)
57:     rms.step_size(10**-2)
58:     rms.beta(0.1)
59:     rms.max_iter(-1)
60:     rms.start(np.array([0, 0]))
61:
62:     def converged(x1, x2):
63:         d = np.max(x1-x2)
64:         return d < 0.000001
65:
66:     def fn(x):
67:         return lib.f.subs(lib.x, x[0]).subs(lib.y, x[1])
68:
69:     def grad(x):
70:         return np.array([lib.f.diff(var).subs(lib.x, x[0]).subs(lib.y, x[1]) for var in (lib.x, lib.y)])
71:     rms.converged(converged)
72:     rms.function(fn)
73:     rms.gradient(grad)
74:     rms.set_iterate(iterate)
75:     rms.run2csv("rms2.csv")
```

```
1: import sympy as sp
2:
3: x = sp.symbols('x')
4: print(x)
5: f = x ** 4
6: print(f)
7: print(f.diff())
8: print(f.subs(x, x**2))
9: print(f.conjugate())
10: print(f)
11: print(f.subs())
```



```
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:
41: if __name__ == "__main__":
42:     if len(sys.argv) != 2:
43:         print("Usage: python script.py <output_file>")
44:         sys.exit(1)
45:
46:     outfile = sys.argv[1]
47:     main(outfile)
48:
```