

Week 2 Optimisation for Machine Learning

Neimhin Robinson Gunning, 16321701

March 11, 2024

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 $f(x, y)$ is 0 and they is minimized by $x = 5, y = 9$. The other function $g(x, y)$ also has minimum 0 but is minized by any of $x \in [-\infty, 5]$ and $y = 9$.

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 $f^* \approx \min_x f(x)$.

```
funcs.txt           Wed Feb 21 15:03:56 2024           1  
function: 3*(x-5)^4+10*(y-9)^2  
function: Max(x-5, 0)+10*|y-9|
```

Figure 1: Two bivariate functions downloaded from <https://www.scss.tcd.ie/Doug.Leith/CS7DS2/week4.php>

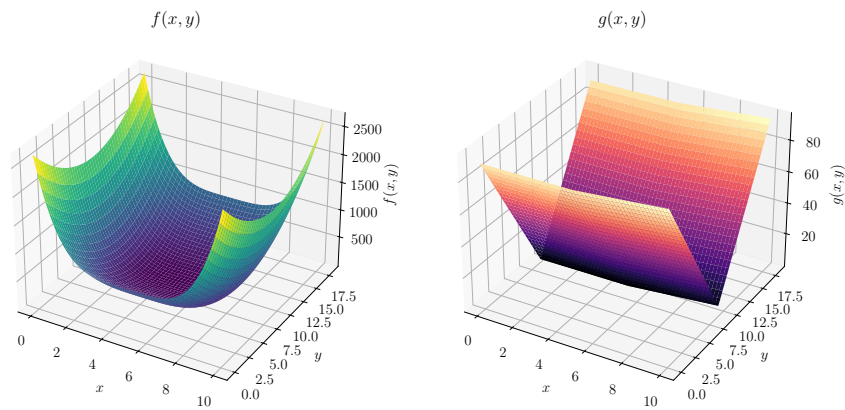


Figure 2

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: import json
4:
5:
6: def iterate(self):
7:     x_value = self._start
8:     old_x_value = None
9:     iteration = 0
10:    m = np.zeros(x_value.shape, dtype=np.float64)
11:    v = np.zeros(x_value.shape, dtype=np.float64)
12:    converged = False
13:    grad_value = self._gradient(x_value)
14:
15:    def yielded():
16:        print(x_value)
17:        print(iteration)
18:        return {
19:            "alg": "adam",
20:            "function_name": self.function_name,
21:            "iteration": iteration,
22:            "x0": x_value[0],
23:            "x1": x_value[1],
24:            "f(x)": self._function(x_value),
25:            "epsilon": self._epsilon,
26:            "converged": converged,
27:            "gradient": grad_value,
28:            "m": m,
29:            "v": v,
30:            "beta1": self._beta,
31:            "beta2": self._beta2,
32:            "alpha": self._step_size,
33:        }
34:
35:    yield yielded()
36:
37:    while not converged:
38:        if self._max_iter > 0 and iteration > self._max_iter:
39:            break
40:        grad_value = self._gradient(x_value)
41:        m = self._beta * m + (1-self._beta)*grad_value
42:        # grad_value * grad_value gives element-wise product of np array
43:        v = self._beta2 * v + (1-self._beta2) * (grad_value*grad_value)
44:        old_x_value = x_value
45:        iteration += 1
46:        m_hat = m / (1-(self._beta ** iteration))
47:        v_hat = np.array(v / (1-(self._beta2 ** iteration)))
48:        print('v', v, type(v))
49:        print('v_hat', v_hat, type(v_hat))
50:        print(np, type(np))
51:        v_hat_aug = v_hat**(0.5) + self._epsilon
52:        adam_grad = m_hat / v_hat_aug
53:        x_value = x_value - self._step_size * adam_grad
54:        converged = self._converged(x_value, old_x_value)
55:        yield yielded()
56:
57:
58: if __name__ == "__main__":
59:     adam = lib.GradientDescent()
60:     adam.epsilon(0.0001)
61:     adam.step_size(10**-2)
62:     adam.beta(0.8)
63:     adam.beta2(0.9)
64:     adam.max_iter(-1)
65:     adam.start(np.array([0, 0]))
66:
67:     def converged(x1, x2):
68:         d = np.max(x1-x2)
69:         return d < 0.000001
70:
71:     def fn(x):
72:         return lib.f.subs(lib.x, x[0]).subs(lib.y, x[1])
73:
74:     def grad(x):
75:         return np.array(
76:             [lib.f.diff(var).subs(lib.x, x[0]).subs(lib.y, x[1])
77:              for var in (lib.x, lib.y)])
78:     adam.converged(converged)
79:     adam.function(fn)
80:     adam.gradient(grad)
81:     adam.set_iterate(iterate)
82:     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 $\\mathrm{arg\\,\\,min}_x x^4$")
51: ax[0].set_xlabel("iteration")
52: ax[1].set_ylabel("$y(\\hat{x})$")
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 rmsprop
6: import adam
7: import heavy_ball
8:
9: def converged(x1, x2):
10:     d = np.max(x1-x2)
11:     return d < 0.0001
12:
13:
14: parser = argparse.ArgumentParser(
15:     prog="Run Gradient Descent A Step Size Algorithm")
16:
17: parser.add_argument('-al', '--algorithm', choices=[
18:     'rmsprop', 'adam', 'polyak', 'heavy_ball'], required=True)
19:
20: parser.add_argument('-b', '--beta', type=float)
21: parser.add_argument('-b2', '--beta2', type=float)
22: parser.add_argument('-a', '--alpha', type=float)
23: parser.add_argument('-f', '--function', type=str)
24: parser.add_argument('filename')
25:
26: args = parser.parse_args()
27:
28: print(args.filename)
29:
30: gd = lib.GradientDescent()
31:
32: function = lib.f if args.function == 'f' else lib.g
33: print(args.function)
34: print(function)
35: if 0:
36:     pass
37: elif args.algorithm == 'rmsprop':
38:     gd.set_iterate(rmsprop.iterate)
39: elif args.algorithm == 'adam':
40:     gd.set_iterate(adam.iterate)
41: elif args.algorithm == 'heavy_ball':
42:     gd.set_iterate(heavy_ball.iterate)
43: else:
44:     print("no algorithm")
45:     sys.exit(1)
46:
47: gd.start(np.array([0, 0]))
48: gd.converged(converged)
49: gd.step_size(args.alpha)
50: gd.beta(args.beta)
51: gd.beta2(args.beta2)
52: gd.epsilon(0.0001)
53: gd.max_iter(-1)
54:
55:
56: def fn(x):
57:     return function.subs(lib.x, x[0]).subs(lib.y, x[1])
58:
59:
60: def grad(x):
61:     return np.array([
62:         function.diff(var).subs(
63:             lib.x, x[0]
64:         ).subs(
65:             lib.y, x[1]
66:         ) for var in (lib.x, lib.y)])
67:
68:
69: gd.converged(converged)
70: gd.function(fn, function_name=args.function)
71: gd.gradient(grad)
72: 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:             "alg": "heavy_ball",
18:             "function_name": self.function_name,
19:             "iteration": iteration,
20:             "z": z,
21:             "x0": x_value[0],
22:             "x1": x_value[1],
23:             "alpha": self._step_size,
24:             "beta1": self._beta,
25:             "f(x)": self._function(x_value),
26:             "epsilon": self._epsilon,
27:             "converged": converged,
28:             "gradient": grad_value,
29:         }
30:
31:     yield yielded() # yield initial values
32:
33:     while not converged:
34:         iteration += 1
35:         if self._max_iter > 0 and iteration > self._max_iter:
36:             break
37:         grad_value = self._gradient(x_value)
38:         old_x_value = x_value
39:         z = self._beta * z + self._step_size * grad_value
40:         x_value = x_value - z
41:         converged = self._converged(x_value, old_x_value)
42:         yield yielded()
43:
44:
45: if __name__ == "__main__":
46:     import numpy as np
47:     hb = lib.GradientDescent()
48:     hb.step_size(10**-3)
49:     hb.beta(0.5)
50:     hb.max_iter(-1)
51:     hb.start(np.array([0, 0]))
52:
53:     def converged(x1, x2):
54:         d = np.max(x1-x2)
55:         return d < 0.000001
56:
57:     def fn(x):
58:         return lib.f.subs(lib.x, x[0]).subs(lib.y, x[1])
59:
60:     def grad(x):
61:         return np.array([
62:             lib.f.diff(var).subs(lib.x, x[0]).subs(lib.y, x[1])
63:             for var in (lib.x, lib.y)])
64:     hb.converged(converged)
65:     hb.function(fn)
66:     hb.gradient(grad)
67:     hb.set_iterate(iterate)
68:     hb.run2csv("hb.csv")
```

```
src/lib.py      Mon Mar 11 17:23:30 2024      1
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, function_name=None):
34:         self._function = f
35:         self.function_name = function_name
36:         return self
37:
38:     def gradient(self, g):
39:         self._gradient = g
40:         return self
41:
42:     def max_iter(self, m):
43:         self._max_iter = m
44:         return self
45:
46:     def start(self, s):
47:         self._start = s
48:         return self
49:
50:     def debug(self, d):
51:         self._debug = d
52:         return self
53:
54:     def converged(self, c):
55:         self._converged = c
56:         return self
57:
58:     def set_iterate(self, f):
59:         self.iterate = functools.partial(f, self)
60:         return self
61:
62:     # def iterate(self):
63:     #     x_value = self._start
64:     #     old_x_value = None
65:     #     iteration = 0
66:     #     while True:
67:     #         yield [iteration, x_value, self._function(x_value)]
68:     #         iteration += 1
69:     #         if self._max_iter > 0 and iteration > self._max_iter:
70:     #             break
71:     #         grad_value = self._gradient(x_value)
72:     #         print(x_value, type(x_value))
73:     #         print(grad_value, type(grad_value))
74:     #         x_value -= self._step_size * grad_value # Update step
75:     #         if old_x_value is not None and self._converged(x_value, old_x_value):
76:     #             yield [iteration, float(x_value), float(self._function(old_x_value))]
77:     #             print("converged")
78:     #             break
79:     #         old_x_value = x_value
80:
81:     def run2csv(self, fname, summarise=True):
82:         import pandas as pd
83:         iterations = list(self.iterate())
84:         df = pd.DataFrame(iterations)
85:         df.to_csv(fname)
86:         if summarise:
87:             with open(fname + ".summary", "w") as f:
88:                 print(f"iterations: {len(df)}", file=f)
89:                 print(f"start: {df['x0'][0]} {df['x1'][0]}", file=f)
90:                 print(f"final: {df['x0'][len(df) - 1]} {df['x1'][len(df) - 1]}", file=f)
91:
92:
93: if __name__ == "__main__":
94:     print(f.diff(x), f.diff(y))
95:     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: import json
3:
4:
5: def iterate(self):
6:     import numpy as np
7:     x_value = self._start
8:     old_x_value = None
9:     iteration = 0
10:    sum = np.zeros(x_value.shape)
11:    alpha_n = np.zeros(x_value.shape)
12:    alpha_n.fill(self._step_size)
13:    converged = False
14:    grad_value = self._gradient(x_value)
15:
16:    def yielded():
17:        print(x_value)
18:        print(iteration)
19:        return {
20:            "alg": "rmsprop",
21:            "function_name": self.function_name,
22:            "iteration": iteration,
23:            "x0": x_value[0],
24:            "x1": x_value[1],
25:            "f(x)": self._function(x_value),
26:            "sum": sum,
27:            "alpha": self._step_size,
28:            "beta1": self._beta,
29:            "epsilon": self._epsilon,
30:            "converged": converged,
31:            "gradient": grad_value,
32:            "alpha_n": alpha_n,
33:        }
34:
35:    yield yielded()
36:
37:    while not converged:
38:        iteration += 1
39:        if self._max_iter > 0 and iteration > self._max_iter:
40:            break
41:        grad_value = self._gradient(x_value)
42:        print(grad_value)
43:        old_x_value = x_value
44:        print(grad_value, type(grad_value))
45:        print(alpha_n, type(alpha_n))
46:        print(x_value, type(x_value))
47:        x_value = x_value - alpha_n * grad_value
48:        sum = self._beta * sum + (1-self._beta) * (grad_value**2)
49:        alpha_n = self._step_size / (sum*0.5+self._epsilon)
50:        converged = self._converged(x_value, old_x_value)
51:        yield yielded()
52:
53:
54: def rms_gradient_descent():
55:     rms = lib.GradientDescent()
56:     rms.set_iterate(iterate)
57:     return rms
58:
59:
60: if __name__ == "__main__":
61:     import numpy as np
62:     rms = lib.GradientDescent()
63:     rms.epsilon(0.0001)
64:     rms.step_size(10**-2)
65:     rms.beta(0.1)
66:     rms.max_iter(-1)
67:     rms.start(np.array([0, 0]))
68:
69:     def converged(x1, x2):
70:         d = np.max(x1-x2)
71:         return d < 0.000001
72:
73:     def fn(x):
74:         return lib.f.subs(lib.x, x[0]).subs(lib.y, x[1])
75:
76:     def grad(x):
77:         return np.array([lib.f.diff(var).subs(lib.x, x[0]).subs(lib.y, x[1]) for var in (lib.x, lib.y)])
78:     rms.converged(converged)
79:     rms.function(fn)
80:     rms.gradient(grad)
81:     rms.set_iterate(iterate)
82:     rms.run2csv("rms2.csv")
```

```
1: import sys
2: import pandas as pd
3:
4: outfile = sys.argv[1]
5: infiles = sys.argv[2:]
6:
7: print('out', outfile)
8: print('in', infiles)
9:
10:
11: for f in infiles:
12:     df = pd.read_csv(f)
13:     print(f)
14:     print(df)
15:     print(df['x'][0], type(df['x'][0]))
```

```
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: 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.contour(X, Y, Z_f, 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.contour(X, Y, Z_g, cmap='magma')
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:
```