

Week 2 Optimisation for Machine Learning

Neimhin Robinson Gunning, 16321701

March 12, 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 $f(x, y)$ 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 $y = 9$.

1 (a)

1.1 (a) (i) Polyak

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>

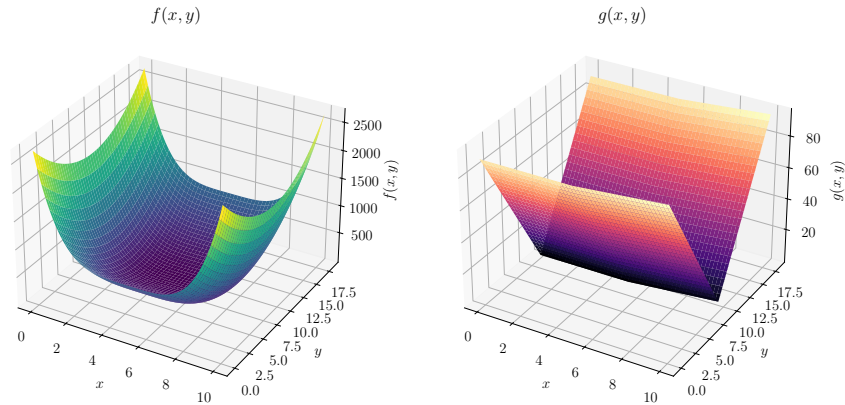


Figure 2

Gradient descent iteration with Polyak step size is implemented in Listing 1. The function is evaluated at the current value for x and the numerator is calculated: $f(x) - f^*$. A reasonable estimate for the minimum of the function, f^* , is required, here assumed to be 0. The dot product of the gradient is taken as the denominator. The step size is $\frac{f(x) - f^*}{\nabla f(x)^T \nabla f(x)}$. We multiply the step size by the gradient and subtract the result from the current x .

Listing 1: An implementation of the update step of gradient descent using Polyak step size.

```
src/polyak.py      Tue Mar 12 15:12:27 2024      1
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()
```

1.2 (a) (ii) RMSProp

The RMSProp step size at iteration t is

$$\alpha_t = \frac{\alpha_0}{\epsilon + \sqrt{(1 - \beta) \sum_{i=0}^{t-1} \beta^{t-i} (\nabla f(x_i))^2}} \quad (4)$$

and the update rule is

$$x_{t+1} := x_t - \alpha_t * \nabla f(x_t) \quad (5)$$

where ϵ is some small value to prevent divide by zero, α_0 and β are hyperparameters to be set, noting that $0 < \beta \leq 1$. The result is that previous gradients influence the current step size, but are gradually forgotten due to the β^{t-i} term.

A Python implementation of the update step is provided in Listing 2. The term inside the square root can be calculated iteratively, as in line 25 of Listing 2.

Listing 2: An implementation of the update step of gradient descent using RMSProp step size.

```
src/rmsprop.py      Tue Mar 12 15:12:45 2024      1
1: def iterate(self):
2:     import numpy as np
3:     self._x_value = self._start
4:     old_x_value = None
5:     self._iteration = 0
6:     self._sum = np.zeros(self._x_value.shape)
7:     alpha_n = np.zeros(self._x_value.shape)
8:     alpha_n.fill(self._step_size)
9:     self._converged_value = False
10:    self._grad_value = self._gradient(self._x_value)
11:
12:    yield self.state_dict()
13:
14:    while not self._converged_value:
15:        self._iteration += 1
16:        if self._max_iter > 0 and self._iteration > self._max_iter:
17:            break
18:        self._grad_value = self._gradient(self._x_value)
19:        old_x_value = self._x_value
20:        self._x_value = self._x_value - alpha_n * self._grad_value
21:        self._sum = self._beta * self._sum + (1-self._beta) * (self._grad_value**2)
22:        alpha_n = self._step_size / (self._sum**0.5+self._epsilon)
23:        self._converged_value = self._converged(self._x_value, old_x_value)
24:        yield self.state_dict()
```

1.3 (a) (iii) Heavy Ball

The Heavy Ball step is

$$z_{t+1} = \beta z_t + \alpha \nabla f(x_t) \quad (6)$$

with the update rule

$$x_{t+1} = x_t - z_{t+1} \quad (7)$$

where t is the current iteration (starting at 0), $z_0 = 0$, and x_0 , α , and β have to be set.

A Python implementation of the update step is provided in Listing 3.

Listing 3: An implementation of the update step of gradient descent using Heavy Ball step size.

```
src/heavy_ball.py      Tue Mar 12 14:57:31 2024      1
1: import lib
2:
3:
4: def iterate(self):
5:     self._x_value = self._start
6:     self._old_x_value = None
7:     self._iteration = 0
8:     self._converged_val = False
9:     self._grad_value = self._gradient(self._x_value)
10:    self._z = 0
11:    yield self.state_dict() # yield initial values
12:
13:    while not self._converged_val:
14:        self._iteration += 1
15:        if self._max_iter > 0 and self._iteration > self._max_iter:
16:            break
17:        self._grad_value = self._gradient(self._x_value)
18:        self._old_x_value = self._x_value
19:        self._z = self._beta * self._z + self._step_size * self._grad_value
20:        self._x_value = self._x_value - self._z
21:        self._converged_val = self._converged(self._x_value, self._old_x_value)
22:        yield self.state_dict()
```

1.4 (a) (iv) Adam

The Adam step size is calculated in terms of

$$m_{t+1} = \beta_1 m_t + (1 - \beta_1) \nabla f(x_t) \quad (8)$$

and

$$v_{t+1} = \beta_2 v_t + (1 - \beta_2) [\nabla f(x_t) \circ \nabla f(x_t)] \quad (9)$$

from which we get

$$\hat{m} = \frac{m_{t+1}}{(1 - \beta_1^t)} \quad (10)$$

and

$$\hat{v} = \frac{v_{t+1}}{(1 - \beta_2^t)} \quad (11)$$

which are used in the update step as

$$x_{t+1} = x_t - \alpha \left[\frac{\hat{m}_1}{\epsilon + \sqrt{\hat{v}_1}}, \dots, \frac{\hat{m}_n}{\epsilon + \sqrt{\hat{v}_n}} \right] \quad (12)$$

where t is the iteration, α , β_1 , and β_2 are hyperparameters, and ϵ is some small value to prevent divide-by-zero.

A Python implementation of the update step is provided in Listing 4.

Listing 4: An implementation of the update step of gradient descent using Adam step size.

```

src/adam.py      Tue Mar 12 14:57:31 2024      1
1: import lib
2: import numpy as np
3: import json
4:
5:
6: def iterate(self):
7:     self._x_value = self._start
8:     self._old_x_value = None
9:     self._iteration = 0
10:    self._m = np.zeros(self._x_value.shape, dtype=np.float64)
11:    self._v = np.zeros(self._x_value.shape, dtype=np.float64)
12:    self._converged_value = False
13:    self._grad_value = self._gradient(self._x_value)
14:
15:    yield self.state_dict()
16:
17:    while not self._converged_value:
18:        if self._max_iter > 0 and self._iteration > self._max_iter:
19:            break
20:        self._grad_value = self._gradient(self._x_value)
21:        self._m = self._beta * self._m + (1-self._beta)*self._grad_value
22:        # grad_value * grad_value gives element-wise product of np array
23:        self._v = self._beta2 * self._v + (1-self._beta2) * (self._grad_value*self._grad_value)
24:        self._old_x_value = self._x_value
25:        self._iteration += 1
26:        m_hat = self._m / (1-(self._beta ** self._iteration))
27:        v_hat = np.array(self._v / (1-(self._beta2 ** self._iteration)))
28:        v_hat_aug = v_hat**(0.5) + self._epsilon
29:        adam_grad = m_hat / v_hat_aug
30:        self._x_value = self._x_value - self._step_size * adam_grad
31:        self._converged_value = self._converged(self._x_value, self._old_x_value)
32:        yield self.state_dict()

```

```
1: import lib
2: import numpy as np
3: import json
4:
5:
6: def iterate(self):
7:     self._x_value = self._start
8:     self._old_x_value = None
9:     self._iteration = 0
10:    self._m = np.zeros(self._x_value.shape, dtype=np.float64)
11:    self._v = np.zeros(self._x_value.shape, dtype=np.float64)
12:    self._converged_value = False
13:    self._grad_value = self._gradient(self._x_value)
14:
15:    yield self.state_dict()
16:
17:    while not self._converged_value:
18:        if self._max_iter > 0 and self._iteration > self._max_iter:
19:            break
20:        self._grad_value = self._gradient(self._x_value)
21:        self._m = self._beta * self._m + (1-self._beta)*self._grad_value
22:        # grad_value * grad_value gives element-wise product of np array
23:        self._v = self._beta2 * self._v + (1-self._beta2) * (self._grad_value*self._grad_value)
24:        self._old_x_value = self._x_value
25:        self._iteration += 1
26:        m_hat = self._m / (1-(self._beta ** self._iteration))
27:        v_hat = np.array(self._v / (1-(self._beta2 ** self._iteration)))
28:        v_hat_aug = v_hat**(0.5) + self._epsilon
29:        adam_grad = m_hat / v_hat_aug
30:        self._x_value = self._x_value - self._step_size * adam_grad
31:        self._converged_value = self._converged(self._x_value, self._old_x_value)
32:        yield self.state_dict()
```

```
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 rmsprop
6: import adam
7: import heavy_ball
8:
9: def converged(x1, x2):
10:     d = np.max(x1-x2)
11:     return d < 0.001
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:     choices=['f', 'g', 'relu'])
25: parser.add_argument('filename')
26:
27: args = parser.parse_args()
28:
29: print(args.filename)
30:
31: gd = lib.GradientDescent()
32: function_handle = lib.config[args.function]
33: function = function_handle['sym']
34:
35:
36: def fn(x):
37:     return function.subs(lib.x, x[0]).subs(lib.y, x[1])
38:
39:
40: def grad(x):
41:     return np.array([
42:         function.diff(var).subs(
43:             lib.x, x[0]
44:         ).subs(
45:             lib.y, x[1]
46:         ) for var in (lib.x, lib.y)])
47:
48: gd.algorithm(args.algorithm)
49: gd.start(np.array([4, 8]))
50: gd.converged(converged)
51: gd.step_size(args.alpha)
52: gd.beta(args.beta)
53: gd.beta2(args.beta2)
54: gd.epsilon(0.0001)
55: gd.max_iter(-1)
56: gd.converged(converged)
57: # gd.sym_function(function_handle["sym"], function_name=args.function)
58: gd.function(fn, function_name=args.function, dimension=2)
59: gd.gradient(grad)
60: 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:     self._x_value = self._start
6:     self._old_x_value = None
7:     self._iteration = 0
8:     self._converged_val = False
9:     self._grad_value = self._gradient(self._x_value)
10:    self._z = 0
11:    yield self.state_dict()    # yield initial values
12:
13:    while not self._converged_val:
14:        self._iteration += 1
15:        if self._max_iter > 0 and self._iteration > self._max_iter:
16:            break
17:        self._grad_value = self._gradient(self._x_value)
18:        self._old_x_value = self._x_value
19:        self._z = self._beta * self._z + self._step_size * self._grad_value
20:        self._x_value = self._x_value - self._z
21:        self._converged_val = self._converged(self._x_value, self._old_x_value)
22:        yield self.state_dict()
```

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

```
101:         self._function = fn
102:         self._gradient = grad
103:         return self
104:
105:     def gradient(self, g):
106:         self._gradient = g
107:         return self
108:
109:     def max_iter(self, m):
110:         self._max_iter = m
111:         return self
112:
113:     def start(self, s):
114:         self._start = s
115:         return self
116:
117:     def debug(self, d):
118:         self._debug = d
119:         return self
120:
121:     def converged(self, c):
122:         self._converged = c
123:         return self
124:
125:     def set_iterate(self, f):
126:         self.iterate = functools.partial(f, self)
127:         return self
128:
129:     def algorithm(self, alg):
130:         self._algorithm = alg
131:         if self._algorithm == "rmsprop":
132:             import rmsprop
133:             self.set_iterate(rmsprop.iterate)
134:         elif self._algorithm == "adam":
135:             import adam
136:             self.set_iterate(adam.iterate)
137:         elif self._algorithm == "heavy_ball":
138:             import heavy_ball
139:             self.set_iterate(heavy_ball.iterate)
140:         else:
141:             raise Exception("Unknown algorithm:" + alg)
142:         return self
143:
144:     def state_dict(self):
145:         print(self._function(self._x_value))
146:         return {
147:             "alg": self._algorithm,
148:             "function_name": self.function_name,
149:             "iteration": self._iteration,
150:             "f(x)": self._function(self._x_value),
151:             "epsilon": self._epsilon,
152:             "converged": self._converged_value,
153:             "gradient": self._grad_value,
154:             "m": self._m,
155:             "v": self._v,
156:             "beta1": self._beta,
157:             "beta2": self._beta2,
158:             "alpha": self._step_size,
159:             "sum": self._sum,
160:             "z": self._z,
161:             **{"x" + str(i): self._x_value[i] for i in range(len(self._x_value))},
162:         }
163:
164:     def run2csv(self, fname, summarise=True):
165:         import pandas as pd
166:         iterations = list(self.iterate())
167:         df = pd.DataFrame(iterations)
168:         df.to_csv(fname)
169:         if summarise:
170:             with open(fname + ".summary", "w") as f:
171:                 print(f"iterations: {len(df)}", file=f)
172:                 print(f"start: {df['x0'][0]} {df['x1'][0]}", file=f)
173:                 print(f"final: {df['x0'][len(df) - 1]} {df['x1'][len(df) - 1]}", file=f)
174:
175:
176: if __name__ == "__main__":
177:     print(f.diff(x), f.diff(y))
178:     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: def iterate(self):
2:     import numpy as np
3:     self._x_value = self._start
4:     old_x_value = None
5:     self._iteration = 0
6:     self._sum = np.zeros(self._x_value.shape)
7:     alpha_n = np.zeros(self._x_value.shape)
8:     alpha_n.fill(self._step_size)
9:     self._converged_value = False
10:    self._grad_value = self._gradient(self._x_value)
11:
12:    yield self.state_dict()
13:
14:    while not self._converged_value:
15:        self._iteration += 1
16:        if self._max_iter > 0 and self._iteration > self._max_iter:
17:            break
18:        self._grad_value = self._gradient(self._x_value)
19:        old_x_value = self._x_value
20:        self._x_value = self._x_value - alpha_n * self._grad_value
21:        self._sum = self._beta * self._sum + (1-self._beta) * (self._grad_value**2)
22:        alpha_n = self._step_size / (self._sum**0.5+self._epsilon)
23:        self._converged_value = self._converged(self._x_value, old_x_value)
24:        yield self.state_dict()
```

```
1: import sys
2: import pandas as pd
3: import lib
4: import numpy as np
5: import matplotlib.pyplot as plt
6:
7: outfile = sys.argv[1]
8: infiles = sys.argv[2:]
9:
10: print('out', outfile)
11: print('in', infiles)
12:
13: def f(x, y):
14:     return 3 * (x - 5)**4 + 10 * (y - 9)**2
15:
16:
17: def g(x, y):
18:     return np.maximum(x - 5, 0) + 10 * np.abs(y - 9)
19:
20:
21: fig = plt.figure(figsize=(12, 6))
22:
23: for f in infiles:
24:     df = pd.read_csv(f)
25:
26:     function_name = df["function_name"][0]
27:     function = f if function_name == 'f' else g
28:
29:     x = np.linspace(0, 10, 400)
30:     y = np.linspace(0, 18, 400)
31:     X, Y = np.meshgrid(x, y)
32:     Z_f = function(X, Y)
33:
34:     ax = fig.add_subplot(1, 2, 1)
35:     ax.contour(X, Y, Z_f, cmap='viridis')
36:     ax.set_title(f'${function_name}(x, y)$')
37:     ax.set_xlabel('$x$')
38:     ax.set_ylabel('$y$')
39:     ax.step(df['x0'], df['x1'])
40:     print(df[['x0', 'x1']])
41:
42: plt.savefig(outfile)
```

```
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 lib
2: import heavy_ball
3: import adam
4: import polyak
5: import rmsprop
6:
7: if __name__ == "__main__":
8:     import numpy as np
9:     hb = lib.GradientDescent()
10:     hb.step_size(10**-3)
11:     hb.beta(0.5)
12:     hb.max_iter(-1)
13:     hb.start(np.array([0, 0]))
14:
15:     def converged(x1, x2):
16:         d = np.max(x1-x2)
17:         return d < 0.000001
18:
19:     def fn(x):
20:         return lib.f.subs(lib.x, x[0]).subs(lib.y, x[1])
21:
22:     def grad(x):
23:         return np.array([
24:             lib.f.diff(var).subs(lib.x, x[0]).subs(lib.y, x[1])
25:             for var in (lib.x, lib.y)])
26:     hb.converged(converged)
27:     hb.function(fn)
28:     hb.gradient(grad)
29:     hb.set_iterate(heavy_ball.iterate)
30:     hb.run2csv("hb.csv")
31:
32:
33: if __name__ == "__main__":
34:     adam = lib.GradientDescent()
35:     adam.epsilon(0.0001)
36:     adam.step_size(10**-2)
37:     adam.beta(0.8)
38:     adam.beta2(0.9)
39:     adam.max_iter(-1)
40:     adam.start(np.array([0, 0]))
41:
42:     def converged(x1, x2):
43:         d = np.max(x1-x2)
44:         return d < 0.000001
45:
46:     def fn(x):
47:         return lib.f.subs(lib.x, x[0]).subs(lib.y, x[1])
48:
49:     def grad(x):
50:         return np.array(
51:             [lib.f.diff(var).subs(lib.x, x[0]).subs(lib.y, x[1])
52:              for var in (lib.x, lib.y)])
53:     adam.converged(converged)
54:     adam.function(fn)
55:     adam.gradient(grad)
56:     adam.set_iterate(adam.iterate)
57:     adam.run2csv("adam.csv")
58:
59: if __name__ == "__main__":
60:     gd = lib.GradientDescent()
61:     gd.epsilon(0.0001)
62:     gd.max_iter(-1)
63:     gd.start(np.array([4.5, 8.5]))
64:
65:     def converged(x1, x2):
66:         d = np.max(x1-x2)
67:         print(f"converged: {d}")
68:         return abs(d) < 0.000001
69:
70:     def fn(x):
71:         return lib.f.subs(lib.x, x[0]).subs(lib.y, x[1])
72:
73:     def grad(x):
74:         return np.array(
75:             [lib.f.diff(var).subs(lib.x, x[0]).subs(lib.y, x[1])
76:              for var in (lib.x, lib.y)])
77:     gd.converged(converged)
78:     gd.function(fn)
79:     gd.gradient(grad)
80:     gd.set_iterate(polyak.iterate)
81:     gd.run2csv("polyak.csv")
82:
83: if __name__ == "__main__":
84:     import numpy as np
85:     rms = lib.GradientDescent()
86:     rms.epsilon(0.0001)
87:     rms.step_size(10**-2)
88:     rms.beta(0.1)
89:     rms.max_iter(-1)
90:     rms.start(np.array([0, 0]))
91:
92:     def converged(x1, x2):
93:         d = np.max(x1-x2)
94:         return d < 0.000001
95:
96:     def fn(x):
97:         return lib.f.subs(lib.x, x[0]).subs(lib.y, x[1])
98:
99:     def grad(x):
100:         return np.array([lib.f.diff(var).subs(lib.x, x[0]).subs(lib.y, x[1]) for var in (lib.x, lib.y)])
```

```
101:    rms.converged(converged)
102:    rms.function(fn)
103:    rms.gradient(grad)
104:    rms.set_iterate(rmsprop.iterate)
105:    rms.run2csv("rms2.csv")
106:
```

```
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:
```