

```
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:        self._adam_grad = m_hat / v_hat_aug
30:        self._x_value = self._x_value - self._step_size * self._adam_grad
31:        self._converged_value = self._converged(self._x_value, self._old_x_value)
32:        yield self.state_dict()
```

```
1: import numpy as np
2: import sympy as sp
3:
4: def gradient_function_fd(minibatch, epsilon=10**(-15)):
5:     def gradient_fd(x):
6:         dydx1 = (f(x + np.array([epsilon, 0]), minibatch) - f(x, minibatch)) / epsilon
7:         dydx2 = (f(x + np.array([0, epsilon]), minibatch) - f(x, minibatch)) / epsilon
8:         return np.array([dydx1, dydx2])
9:     return gradient_fd
10:
11: def loss(x, w):
12:     z = x - w - 1
13:     left = 10 * (z[0]**2+z[1]**2)
14:     right = (z[0]+2)**2+(z[1]+4)**2
15:     return min(left, right)
16:
17: def f_clear(x, minibatch):
18:     return sum(loss(x, w) for w in minibatch) / len(minibatch)
19:
20: def generate_minibatches(T, n=5, seed=42, shuffle=True):
21:     if shuffle:
22:         T = T.copy()
23:         np.random.seed(seed)
24:         np.random.shuffle(T)
25:     num_rows = T.shape[0]
26:     i = 0
27:
28:     minibatch = np.zeros((n, T.shape[1]), T.dtype)
29:     while True:
30:         for j in range(n):
31:             minibatch[j] = T[i % num_rows]
32:             i += 1
33:             if shuffle and i >= num_rows:
34:                 # begin next epoch
35:                 np.random.shuffle(T)
36:                 i = 0
37:         current_minibatch = minibatch
38:         yield minibatch
39:
40: def generate_optimisation_functions(batch, minibatch_size=5, finite_difference=True, **kwargs):
41:     minibatch_generator = generate_minibatches(
42:         batch, n=minibatch_size, **kwargs)
43:     for minibatch in minibatch_generator:
44:         def optim_func(x):
45:             return f_clear(x, minibatch)
46:         gradf = gradient_function_fd(minibatch)
47:         yield (optim_func, gradf)
48:     yield "finished"
```

```
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: relu = sp.Max(x, 0)
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: def relu_real(x):
18:     return np.maximum(x, 0)
19:
20:
21: def apply_sym(x, f):
22:     for x_sym, x_val in zip(f.free_symbols, x):
23:         f = f.subs(x_sym, x_val)
24:     return f
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._step_coeff = None
58:         self._converged_value = None
59:         self._grad_value = None
60:         self._m = None
61:         self._v = None
62:         self._adam_grad = None
63:         self._beta = None
64:         self._beta2 = None
65:         self._step_size = None
66:         self._z = None
67:         self._f_star = None
68:
69:     def step_size(self, a):
70:         self._step_size = a
71:         return self
72:
73:     def beta(self, b):
74:         self._beta = b
75:         return self
76:
77:     def beta2(self, b):
78:         self._beta2 = b
79:         return self
80:
81:     def epsilon(self, e):
82:         self._epsilon = e
83:         return self
84:
85:     def function(self, f, function_name=None, dimension=None):
86:         self._function = f
87:         self.function_name = function_name
88:         self._dimension = dimension
89:         return self
90:
91:     def sym_function(self, function, function_name=None):
92:         self.function_name = function_name
93:         self._dimension = len(function.free_symbols)
94:         def fn(x):
95:             return apply_sym(x, function)
96:
97:         diffs = [function.diff(var) for var in function.free_symbols]
98:
99:         def grad(x):
100:             return np.array([
```

```
101:         apply_sym(x, diff) for diff in diffs])
102:
103:     self._function = fn
104:     self._gradient = grad
105:     return self
106:
107: def gradient(self, g):
108:     self._gradient = g
109:     return self
110:
111: def max_iter(self, m):
112:     self._max_iter = m
113:     return self
114:
115: def start(self, s):
116:     self._start = s
117:     self._x_value = s
118:     return self
119:
120: def debug(self, d):
121:     self._debug = d
122:     return self
123:
124: def converged(self, c):
125:     self._converged = c
126:     return self
127:
128: def set_iterate(self, f):
129:     self.iterate = functools.partial(f, self)
130:     return self
131:
132: def algorithm(self, alg):
133:     self._algorithm = alg
134:     if self._algorithm == "rmsprop":
135:         import rmsprop
136:         self.set_iterate(rmsprop.iterate)
137:     elif self._algorithm == "adam":
138:         import adam
139:         self.set_iterate(adam.iterate)
140:     elif self._algorithm == "heavy_ball":
141:         import heavy_ball
142:         self.set_iterate(heavy_ball.iterate)
143:     else:
144:         raise Exception("Unknown algorithm:" + alg)
145:     return self
146:
147: def state_dict(self):
148:     print(self._function(self._x_value))
149:     return {
150:         "alg": self._algorithm,
151:         "function_name": self.function_name,
152:         "iteration": self._iteration,
153:         "step_coeff": self._step_coeff,
154:         "adam_grad": self._adam_grad,
155:         "f(x)": self._function(self._x_value),
156:         "epsilon": self._epsilon,
157:         "converged": self._converged_value,
158:         "gradient": self._grad_value,
159:         "m": self._m,
160:         "v": self._v,
161:         "beta1": self._beta,
162:         "beta2": self._beta2,
163:         "alpha": self._step_size,
164:         "sum": self._sum,
165:         "z": self._z,
166:         **{"x" + str(i): self._x_value[i] for i in range(len(self._x_value))},
167:     }
168:
169: def run2csv(self, fname, summarise=True):
170:     import pandas as pd
171:     iterations = list(self.iterate())
172:     df = pd.DataFrame(iterations)
173:     df.to_csv(fname)
174:     if summarise:
175:         with open(fname + ".summary", "w") as f:
176:             print(f"iterations: {len(df)}", file=f)
177:             print(f"start: {df['x0'][0]} {df['x1'][0]}", file=f)
178:             print(f"final: {df['x0'][len(df) - 1]} {df['x1'][len(df) - 1]}", file=f)
179:
180:
181: if __name__ == "__main__":
182:     print(f.diff(x), f.diff(y))
183:     print(g.diff(x), g.diff(y))
```

```
1: import pandas as pd
2: import matplotlib.pyplot as plt
3:
4:
5: df = pd.read_csv("data/T.csv")
6:
7:
8: plt.scatter(df["0"], df["1"])
9: plt.xlabel("$x$")
10: plt.ylabel("$y$")
11: plt.title("Traning Data")
12: plt.savefig("fig/T.pdf")
```

```
1: import numpy as np
2: import pandas as pd
3: import week6
4: import matplotlib.pyplot as plt
5: import argparse
6: from mpl_toolkits.mplot3d import Axes3D
7:
8: ap = argparse.ArgumentParser()
9: ap.add_argument("--show", action="store_true")
10: args = ap.parse_args()
11:
12: # Global variables for extents
13: x_min, x_max = -5, 5
14: y_min, y_max = -5, 5
15:
16: def plot_wireframe_and_contour(f, T, resolution=100):
17:     global x_min, x_max, y_min, y_max
18:
19:     # Generate data for wireframe plot
20:     x_range = np.linspace(x_min, x_max, resolution)
21:     y_range = np.linspace(y_min, y_max, resolution)
22:     X, Y = np.meshgrid(x_range, y_range)
23:     Z = np.zeros_like(X)
24:     for i in range(resolution):
25:         for j in range(resolution):
26:             Z[i, j] = f([X[i, j], Y[i, j]], T)
27:
28:     # Plot wireframe
29:     fig = plt.figure(figsize=(12, 6))
30:
31:     ax_wireframe = fig.add_subplot(121, projection='3d')
32:     ax_wireframe.plot_wireframe(X, Y, Z, color='blue')
33:     ax_wireframe.set_xlabel('X')
34:     ax_wireframe.set_ylabel('Y')
35:     ax_wireframe.set_zlabel('f(x, T)')
36:     ax_wireframe.set_title('Wireframe Plot of f(x, T)')
37:
38:     # Generate data for contour plot
39:     Z_contour = np.zeros_like(X)
40:     for i in range(resolution):
41:         for j in range(resolution):
42:             Z_contour[i, j] = f([X[i, j], Y[i, j]], T)
43:
44:     # Plot contour
45:     ax_contour = fig.add_subplot(122)
46:     contour = ax_contour.contourf(X, Y, Z_contour, levels=20, cmap='viridis')
47:     plt.colorbar(contour, ax=ax_contour, label='f(x, T)')
48:     ax_contour.set_xlabel('X')
49:     ax_contour.set_ylabel('Y')
50:     ax_contour.set_title('Contour Plot of f(x, T)')
51:
52:     plt.tight_layout()
53:     if args.show:
54:         plt.show()
55:     else:
56:         plt.savefig("fig/wire-contour.pdf")
57:
58: if __name__ == "__main__":
59:     df = pd.read_csv("data/T.csv")
60:     T = df.values
61:     plot_wireframe_and_contour(week6.f, T) # Call the function to plot wireframe and contour
62:
```

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



```
1: import sgd
2: import week6
3: import pandas as pd
4: import numpy as np
5:
6: if __name__ == "__main__":
7:     T = pd.read_csv("data/T.csv").values
8:     o = sgd.StochasticGradientDescent().alg("polyak")
9:     fg = week6.generate_optimisation_functions(T, minibatch_size=5)
10:    o.function_generator(fg)
11:    o.start(np.array([3, 3]))
12:    for i in range(100):
13:        o.step()
14:        print("grad", o._grad_value)
15:    print("polyak", o._x_value)
```

```
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:    self._step_coeff = self._step_size
12:
13:    yield self.state_dict()
14:
15:    while not self._converged_value:
16:        self._iteration += 1
17:        if self._max_iter > 0 and self._iteration > self._max_iter:
18:            break
19:        self._grad_value = self._gradient(self._x_value)
20:        old_x_value = self._x_value
21:        self._x_value = self._x_value - alpha_n * self._grad_value
22:        self._sum = self._beta * self._sum + (1-self._beta) * (self._grad_value**2)
23:        alpha_n = self._step_size / (self._sum**0.5+self._epsilon)
24:        self._step_coeff = alpha_n
25:        self._converged_value = self._converged(self._x_value, old_x_value)
26:        yield self.state_dict()
```

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

```
101:         return True
102:
103:     # pass a function which generates the function to be evaluated,
104:     # e.g. with different minibatches at each iteration
105:     def function_generator(self, fg):
106:         self._function_generator = fg
107:         return self
108:
109:     def alg(self, a):
110:         if a == "constant":
111:             self.step = self.constant_step
112:         elif a == "polyak":
113:             self.step = self.polyak_step
114:         elif a == "rmsprop":
115:             self.step = self.rmsprop_step
116:             if self._step_size is None:
117:                 raise Exception("Need step_size to initialize rmsprop")
118:             if self._x_value is None:
119:                 raise Exception("Need start/x_value to initialize rmsprop")
120:             self._sum = np.zeros(self._x_value.shape)
121:             self._alpha_n = np.zeros(self._x_value.shape)
122:             self._alpha_n.fill(self._step_size)
123:         elif a == "adam":
124:             self.step = self.adam_step
125:             if self._x_value is None:
126:                 raise Exception("Need start/x_value to initialize rmsprop")
127:             self._m = np.zeros(self._x_value.shape, dtype=np.float64)
128:             self._v = np.zeros(self._x_value.shape, dtype=np.float64)
129:         elif a == "heavy_ball":
130:             self.step = self.heavy_ball_step
131:             self._z = 0
132:         else:
133:             raise Exception(f"Alg {a} NYI")
134:         self.function_name = a
135:         return self
136:
137:     def polyak_init(self):
138:         self._x_value = self._start
139:         self._old_x_value = None
140:         self._f_star = 0
141:         self._iteration = 0
142:         self._converged_value = False
143:         self._grad_value = self._gradient(self._x_value)
144:
```

```
1: import sgd
2: import week6
3: import pandas as pd
4: import numpy as np
5:
6: if __name__ == "__main__":
7:     T = pd.read_csv("data/T.csv").values
8:
9:     o = sgd.StochasticGradientDescent().alg("constant")
10:    fg = week6.generate_optimisation_functions(T, minibatch_size=5)
11:    o.function_generator(fg)
12:    o.step_size(0.01)
13:    o.start(np.array([3, 3]))
14:    for i in range(100):
15:        o.step()
16:    print("constant", o._x_value)
17:
18:    o = sgd.StochasticGradientDescent().alg("polyak")
19:    fg = week6.generate_optimisation_functions(T, minibatch_size=10, shuffle=False)
20:    o.function_generator(fg)
21:    o.start(np.array([0.9, 0.9]))
22:    for i in range(100):
23:        o.step()
24:    print("polyak", o._x_value)
25:
26:    o = sgd.StochasticGradientDescent().alg("polyak")
27:    fg = week6.generate_optimisation_functions(T, minibatch_size=5)
28:    o.function_generator(fg)
29:    o.start(np.array([3, 3]))
30:    for i in range(100):
31:        o.step()
32:    print("polyak", o._x_value)
33:
34:    o = sgd.StochasticGradientDescent()
35:    fg = week6.generate_optimisation_functions(T, minibatch_size=5)
36:    o.function_generator(fg)
37:    o.start(np.array([3, 3]))
38:    o.step_size(0.00001)
39:    o.beta(0.99)
40:    o.alg("rmsprop")
41:    for i in range(100):
42:        o.step()
43:    print("rmsprop", o._x_value)
44:
45:    o = sgd.StochasticGradientDescent()
46:    fg = week6.generate_optimisation_functions(T, minibatch_size=5)
47:    o.function_generator(fg)
48:    o.start(np.array([3, 3]))
49:    o.step_size(0.00001)
50:    o.beta(0.99)
51:    o.alg("heavy_ball")
52:    for i in range(100):
53:        o.step()
54:    print("heavy_ball", o._x_value)
55:
56:    o = sgd.StochasticGradientDescent()
57:    fg = week6.generate_optimisation_functions(T, minibatch_size=5)
58:    o.function_generator(fg)
59:    o.start(np.array([3, 3]))
60:    o.step_size(0.00001)
61:    o.beta(0.99)
62:    o.beta2(0.25)
63:    o.alg("adam")
64:    for i in range(100):
65:        o.step()
66:    print("adam", o._x_value)
```

```
1: import week6
2: import pandas as pd
3: import sympy as sp
4:
5: T = pd.read_csv("data/T.csv").values
6:
7: sympy_loss = week6.sympy_loss(T)
8:
9: x1, x2 = sp.symbols('x1 x2', real=True)
10:
11: dydx1 = sp.diff(sympy_loss, x1)
12: dydx2 = sp.diff(sympy_loss, x2)
13:
14: solutions = sp.solve([dydx1, dydx2], (x1, x2), simplify=False, rational=False)
15:
16: print(solutions)
```

```
1: import numpy as np
2: import pandas as pd
3: import week6
4: import matplotlib.pyplot as plt
5: from mpl_toolkits.mplot3d import Axes3D
6: from matplotlib.colors import LogNorm
7:
8: # Global variables for extents
9: x_min, x_max = -5, 5
10: y_min, y_max = -5, 5
11:
12: def plot_wireframe_and_contour(f, T, resolution=100):
13:     global x_min, x_max, y_min, y_max
14:
15:     # Generate data for wireframe plot
16:     x_range = np.linspace(x_min, x_max, resolution)
17:     y_range = np.linspace(y_min, y_max, resolution)
18:     X, Y = np.meshgrid(x_range, y_range)
19:     Z = np.zeros_like(X)
20:     for i in range(resolution):
21:         for j in range(resolution):
22:             Z[i, j] = f([X[i, j], Y[i, j]], T)
23:
24:     # Plot wireframe
25:     fig = plt.figure(figsize=(12, 6))
26:
27:     ax_wireframe = fig.add_subplot(121, projection='3d')
28:     ax_wireframe.plot_wireframe(X, Y, Z, color='blue')
29:     ax_wireframe.set_xlabel('X')
30:     ax_wireframe.set_ylabel('Y')
31:     ax_wireframe.set_zlabel('f(x, T)')
32:     ax_wireframe.set_title('Wireframe Plot of f(x, T)')
33:
34:     # Generate data for contour plot
35:     Z_contour = np.zeros_like(X)
36:     for i in range(resolution):
37:         for j in range(resolution):
38:             Z_contour[i, j] = f([X[i, j], Y[i, j]], T)
39:
40:     # Plot contour with log scale color
41:     ax_contour = fig.add_subplot(122)
42:     contour = ax_contour.contourf(X, Y, Z_contour, levels=20, norm=LogNorm(), cmap='viridis')
43:     plt.colorbar(contour, ax=ax_contour, label='f(x, T)')
44:     ax_contour.set_xlabel('X')
45:     ax_contour.set_ylabel('Y')
46:     ax_contour.set_title('Contour Plot of f(x, T)')
47:
48:     plt.tight_layout()
49:     plt.show()
50:
51: if __name__ == "__main__":
52:     df = pd.read_csv("data/T.csv")
53:     T = df.values
54:     plot_wireframe_and_contour(week6.f, T) # Call the function to plot wireframe and contour
55:
```

```
1: import numpy as np
2: import pandas as pd
3: import week6
4: import matplotlib.pyplot as plt
5: from mpl_toolkits.mplot3d import Axes3D
6:
7: # Global variables for extents
8: x_min, x_max = -5, 5
9: y_min, y_max = -5, 5
10:
11: def plot_wireframe_and_contour(f, T, resolution=100):
12:     global x_min, x_max, y_min, y_max
13:
14:     # Generate data for wireframe plot
15:     x_range = np.linspace(x_min, x_max, resolution)
16:     y_range = np.linspace(y_min, y_max, resolution)
17:     X, Y = np.meshgrid(x_range, y_range)
18:     Z = np.zeros_like(X)
19:     for i in range(resolution):
20:         for j in range(resolution):
21:             Z[i, j] = f([X[i, j], Y[i, j]], T)
22:
23:     # Plot wireframe
24:     fig = plt.figure(figsize=(12, 6))
25:
26:     ax_wireframe = fig.add_subplot(121, projection='3d')
27:     ax_wireframe.plot_wireframe(X, Y, Z, color='blue')
28:     ax_wireframe.set_xlabel('X')
29:     ax_wireframe.set_ylabel('Y')
30:     ax_wireframe.set_zlabel('f(x, T)')
31:     ax_wireframe.set_title('Wireframe Plot of f(x, T)')
32:
33:     # Generate data for contour plot
34:     Z_contour = np.zeros_like(X)
35:     for i in range(resolution):
36:         for j in range(resolution):
37:             Z_contour[i, j] = f([X[i, j], Y[i, j]], T)
38:
39:     # Plot contour
40:     ax_contour = fig.add_subplot(122)
41:     contour = ax_contour.contourf(X, Y, Z_contour, levels=20, cmap='viridis')
42:     plt.colorbar(contour, ax=ax_contour, label='f(x, T)')
43:     ax_contour.set_xlabel('X')
44:     ax_contour.set_ylabel('Y')
45:     ax_contour.set_title('Contour Plot of f(x, T)')
46:
47:     plt.tight_layout()
48:     plt.show()
49:
50: if __name__ == "__main__":
51:     df = pd.read_csv("data/T.csv")
52:     T = df.values
53:     plot_wireframe_and_contour(week6.f, T) # Call the function to plot wireframe and contour
54:
```



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

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

```
1: import week6
2: import numpy as np
3:
4: if __name__ == "__main__":
5:     T = week6.generate_trainingdata()
6:     import pandas as pd
7:     df = pd.read_csv("data/T.csv")
8:     T = df.values
9:
10:    x = np.array([3, 3])
11:    print(week6.f(x, T) - week6.f_clear(x, T))
12:
13:    generator = week6.generate_minibatches(T, N=2, shuffle=False)
14:    for i in range(3):
15:        n = next(generator)
16:        print(len(n), n)
17:
18:    fgen = week6.generate_optimisation_functions(T, minibatch_size=5)
19:    zipped = zip(range(10), fgen)
20:    for (i, f) in zipped:
21:        print(f[0](x), f[1](x))
```

```
1: import numpy as np
2: minibatch = np.array([
3:     [0.0918635, -0.0468714],
4:     [-0.66994666, -0.133955],
5:     [-0.08386569, 0.3052427],
6:     [-0.00564624, -0.12876412],
7:     [-0.38826176, 0.23831869]
8: ])
9:
10: x = [0.80697696, 1.05286489]
11:
12: import week6
13: print(minibatch)
14: print("gradient:", week6.gradient_function_fd(minibatch)(x))
```