Optimisation Algorithms for Data Analysis Week 6 Assignment

Jiaming Deng 22302794

The loss functin I used in this assignment is attached in the appendix.

(a)

(i) I implemented the mini-batch Stochastic Gradient Descent (SGD) algorithm by implementing a class called SGD. Firstly I started with a constructor which accepts as input parameters the function, the derivative of the function, the starting value of x, the chosen algorithm, the parameters, the batch size and the training data.

```
def __init__(self, f, df, x, algorithm, params, batch_size, training_data):
```

Secondly, the minibash() function is used to implement mini-batch SGD. The idea is to scramble the training data first. The batch_size parameter of the constructor determines the batch size and each cycle uses batch_size data. The algorithm chosen by the constructor and records the result in records.

```
def minibatch(self):
    # shuffle training data
    np.random.shuffle(self.training_data)
    n = len(self.training_data)
    for i in range(0, n, self.batch_size):
        if i + self.batch_size > n:
            continue
        data = self.training_data[i:(i + self.batch_size)]
        self.algorithm(data)
    self.records['x'].append(deepcopy(self.x))
    self.records['f'].append(self.f(self.x, self.training_data))
```

Thirdly, the function of the get_derivative() function is to calculate the derivative value.

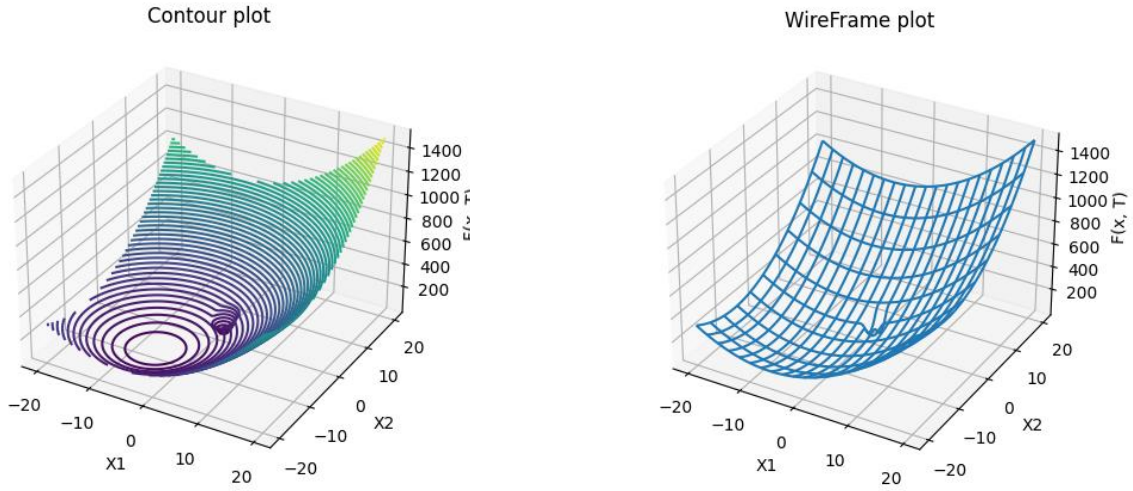
```
def get_derivative(self, i, data):
    Sum = 0
    for j in range(self.batch_size):
        Sum = Sum + self.df[i](*self.x, *data[j])
    return Sum / self.batch_size
```

Finally, in order to help users choose the right algorithm, I use enum type data to define our 5 algorithms, and users only need to use the name of the algorithm to use the algorithm. The process how I implement the five algorithms have been implemented in the last assignment.

```
class ALGORITHM(Enum):
    constant = 0
    polyak = 1
    rmsprop = 2
    heavyball = 3
    adam = 4
```

```
def get_algorithm_via_enum_name(self, algorithm):
    if algorithm == StepSize.constant:
        return self.constant
    elif algorithm == StepSize.polyak:
        return self.polyak
    elif algorithm == StepSize.rmsprop:
        return self.rmsprop
    elif algorithm == StepSize.heavyball:
        return self.heavy_ball
    else:
        return self.adam
```

(ii) The following figure shows the contour plot and the wireframe plot of the function f(x, N). I chose a range of x1 between -20 and 20 and x2 also between -20 and 20. Since the minimum value of the function f can be found under this range, you can see that the broad variation of the function includes plunges of the function between 0 and 10 for x1, and almost no variation of the function between -5 and 0 for x1.



(iii) The formula I needed to implement to find the derivative and the code I implemented are as follows.

$$f = \min(33({Z_0}^2 + {Z_1}^2), (Z_0 + 10)^2, (Z_1 + 7)^2), Z_i = X_i - W_i - 1$$

```
x0, x1, w0, w1 = sp.symbols('x0 x1 w0 w1', real=True)
# as z = x - w - 1
# y = y + min(33 * (z[0] ** 2 + z[1] ** 2), (z[0] + 10) ** 2 + (z[1] + 7) ** 2)
f = sp.Min(33 * ((x0-w0-1)**2 + (x1-w1-1)**2), (x0-w0+9)**2 + (x1-w1+6)**2)
df0 = sp.diff(f, x0)
df1 = sp.diff(f, x1)
print(df0)
print(df1)
```

First I use SymPy's library function to create 4 variables x0, x1, w0, w1. Since zi = xi - wi - 1, the expression f in the code is obtained by substituting x, w into the formula and using the diff function to find the partial derivatives of x0 and x1 respectively. The results obtained are as follows.

$$\begin{aligned} df_0 = (&-66W_0 + 66X_0 - 66) * \text{Heaviside}(-33(-W_0 + X_0 - 1)^2 \\ &+ (-W_0 + X_0 + 9)^2 - 33(-W_1 + X_1 - 1)^2 \\ &+ (-W_1 + X_1 + 6)^2) + (-2W_0 + 2X_0 + 18) \\ &* \text{Heaviside}(33(-W_0 + X_0 - 1)^2 - (-W_0 + X_0 + 9)^2 \\ &+ 33(-W_1 + X_1 - 1)^2 - (-W_1 + X_1 + 6)^2) \end{aligned}$$

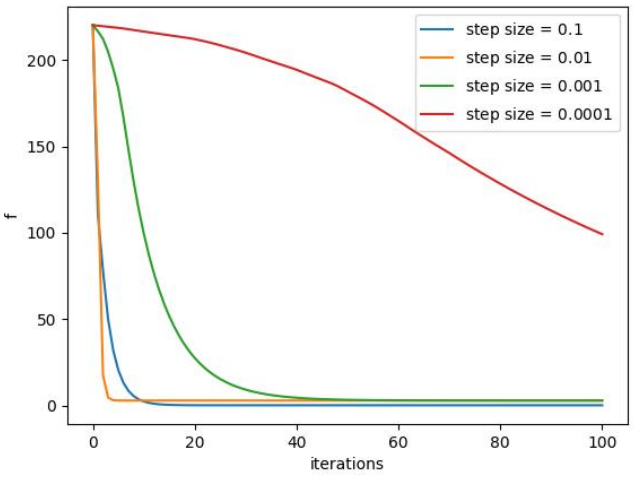
$$\begin{aligned} df_1 = (&-66W_1 + 66X_1 - 66) * \text{Heaviside}(-33(-W_0 + X_0 - 1)^2 \\ &+ (-W_0 + X_0 + 9)^2 - 33(-W_1 + X_1 - 1)^2 \\ &+ (-W_1 + X_1 + 6)^2) + (-2W_1 + 2X_1 + 12) \\ &* \text{Heaviside}(33(-W_0 + X_0 - 1)^2 - (-W_0 + X_0 + 9)^2 \\ &+ 33(-W_1 + X_1 - 1)^2 - (-W_1 + X_1 + 6)^2) \end{aligned}$$

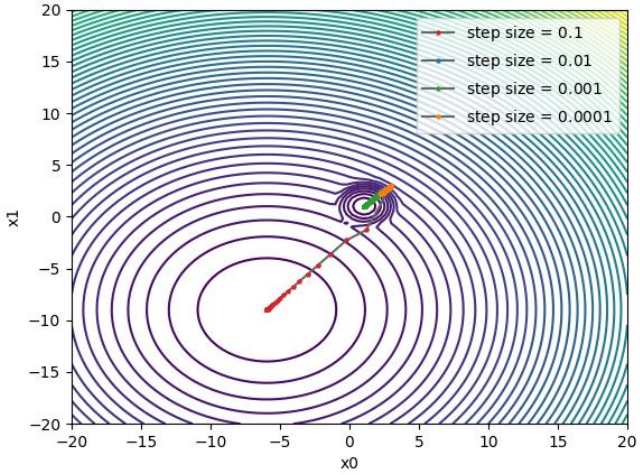
(b)

(i) As I need to use gradient descent with a constant step-size, the batch size should be the size of the training data set. The step size of the initialization experiment is 0.0001, 0.001, 0.01 and 0.1. For each step size, iterates 100 times each time, and the code and result are shown in the figure below.

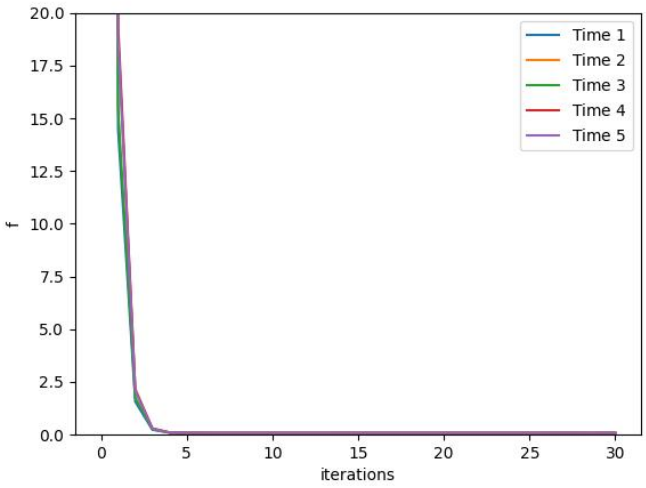
It can be seen from observation that when the step size is 0.0001, the function does not converge to 0 until the 100th iteration. When the step size is 0.01 and 0.001, the functions are slightly greater than 0, and they do not converge to the minimum. But when the step size is 0.1, the function converges to 0 the fastest, so this step size is optimal. The reason can be seen from the contour plot, because there is noise in the data between 0-5 in the training data, forming a local convergence, except for the case where the step size is 0.1, other step sizes converge here. Therefore, their convergence to slightly greater than 0 no longer converges.

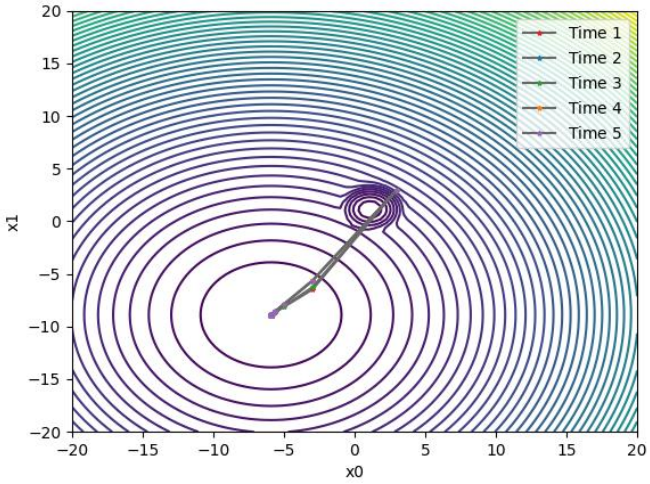
```
trainging_data = generate_trainingdata()
cnt = 100
iters = list(range(cnt + 1))
alphas = [0.1, 0.01, 0.001, 0.0001]
labels = [f'$\\alpha={alpha}$' for alpha in alphas]
xs, fs = [], []
for i, alpha in enumerate(alphas):
    sgd = SGD(func, df, [3, 3], ALGORITHM.constant, {'alpha': alpha}, batch_size=len(trainging_data), training_data=trainging_data)
    for _ in range(cnt):
        sgd.minibatch()
    plt.plot(iters, sgd.records['f'])
    xs.append(deepcopy(sgd.records['x']))
    fs.append(deepcopy(sgd.records['f']))

plt.xlabel('iterations')
plt.ylabel('f')
plt.legend(labels)
plt.show()
```





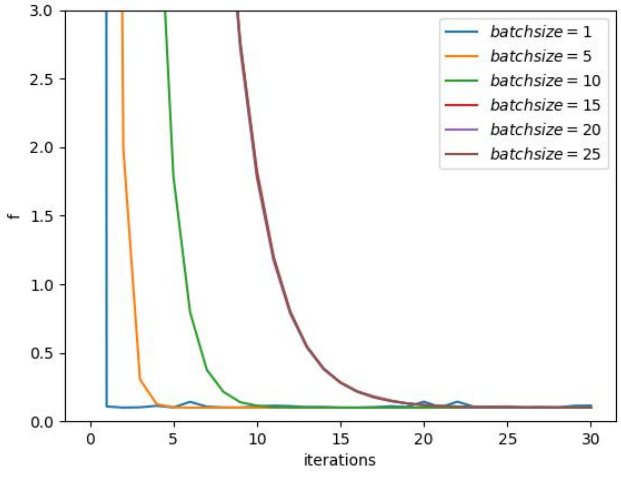
(ii) According to the previous question, the step size in this question is 0.1, and a same training data is used to run 5 times, and the result is shown in the figure below.

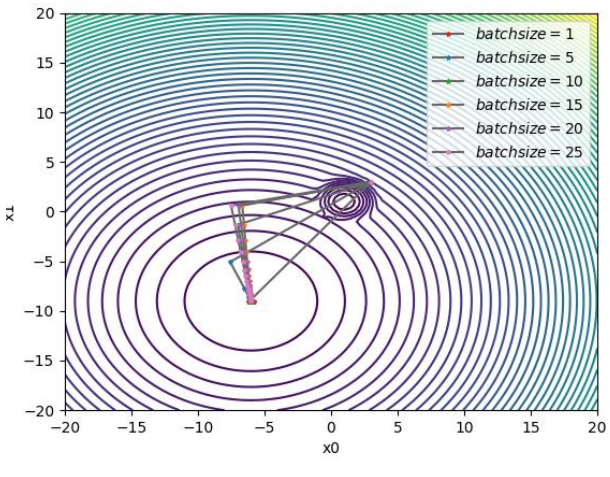




It can be seen from observation that the iterative process of each SGD is almost the same, but there are some subtle differences, probably because the training data will be disrupted each time. Each iteration converges to the minimum value at the fifth time, and there is no case where the decrease stops when it converges to close to 0. Compared with the results of (i), in this question, iterates 5 times to converge to the minimum value, while (i) needs to iterate about 10 times. Moreover, all the results in this question are iterated to the minimum value, and in (i) the function converges to the minimum value only when the step size is 0.1.

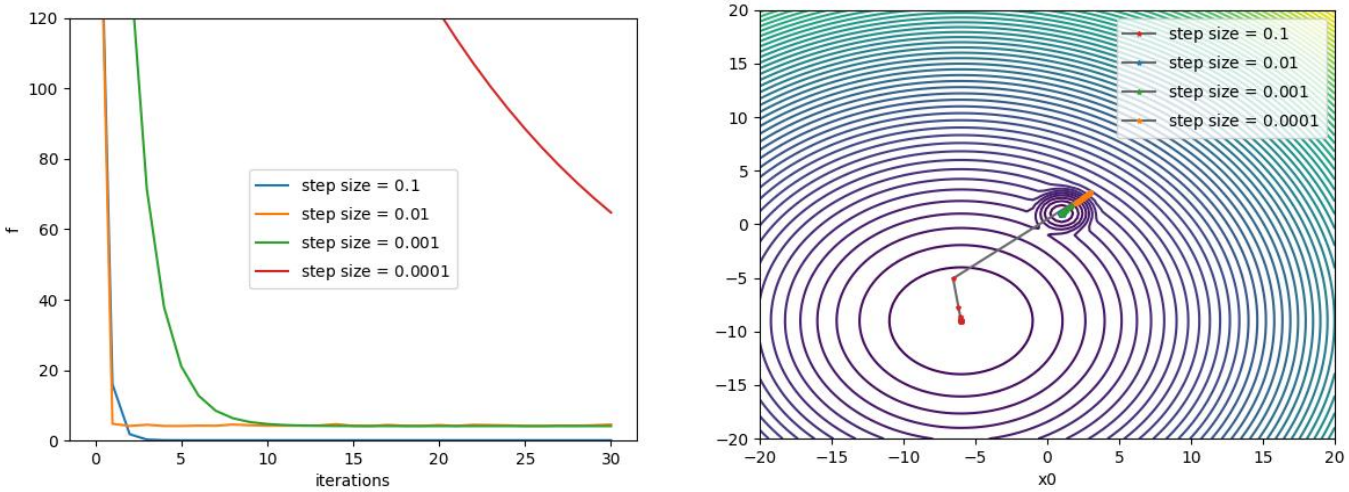
(iii) Firstly, I choose 6 different batch sizes, namely 1, 5, 10, 15, 20, 25, and iterate 30 times on the same training data. The result is shown in the figure below.





By observation, using all experimental batch sizes, the function converges to the minimum value. Because the step size is chosen to be 0.1, it is large enough to cross the local convergence limit. The larger the batch size, the more times the function needs to converge. However, when the batch size is equal to 1, the function still fluctuates after it converges to 0, because the batch size is small, and the noise in the training data affects the convergence of the function. But the effect of this noise is gone when the batch size is equal to 5.

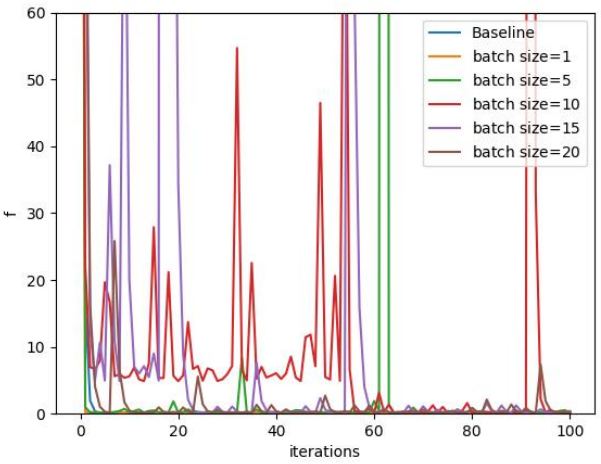
(iv) Firstly, fix the batch size to 5, use different step sizes of 0.0001, 0.001, 0.01, and 0.1, and set the number of iterations to 30. The results are shown in the figure.

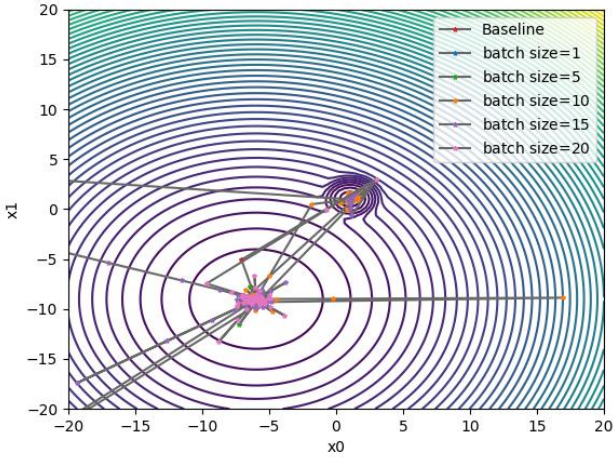


The function converges to the minimum value only when the step size is 0.1, and their situation is the same as that of b(i). When the step size is 0.1, the function does not converge after 30 iterations; when the step size is 0.01 and 0.001, the function converges to the local minimum. Because only when the step size is large enough, x can increase enough to avoid the influence of training data noise. It can be seen from b(iii) that when the batch size is 5, the function converges the fastest, but if the step size is reduced, the function still does not converge to the minimum value, indicating that the smaller the step size, the slower the function converges, and The possibility of being affected by noise is also greater.

(C)

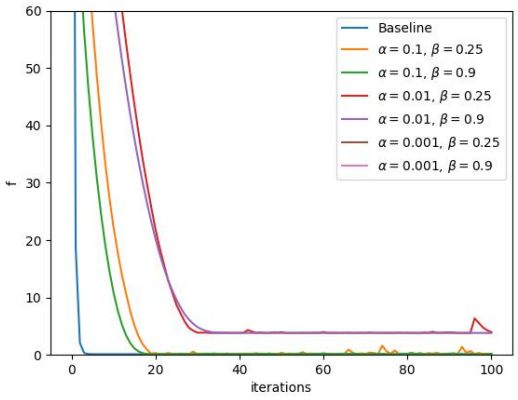
(i) This question selects the Polyak step size algorithm, selects the batch size as 1, 5, 10, 15, and 20, and runs 100 iterations each time. The result is shown in the figure below.



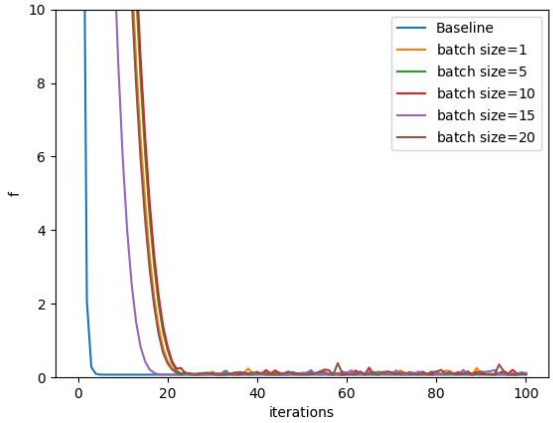


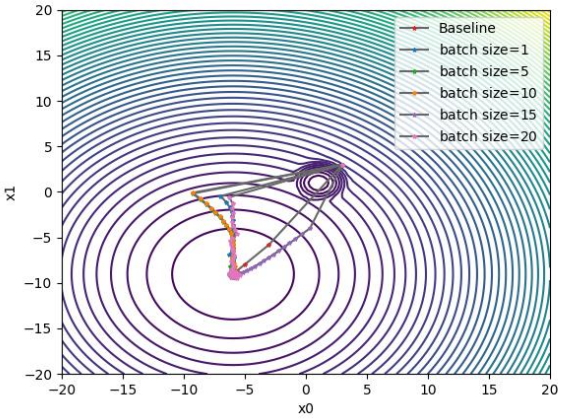
When the batch size is 1, the function converges the fastest. When the batch size takes other values, the convergence is not as fast as the baseline. And they continue to change after they converge. Because the step size during each iteration is large than necessary step size.

(ii) This question selects the RMSProp algorithm. Firstly, according to the last assignment, I chose alpha to be 0.001, 0.01, 0.1, and beta to be 0.25 and 0.9 for experiments, and the results are shown in the figure below.
As can be seen from the figure below, when alpha is 0.1 and beta is 0.9, the function converges to the minimum value of 0 and the convergence speed is the fastest. T herefore, I choose alpha to be 0.1 and beta to be 0.9 as the parameters of the RMSProp algorithm.

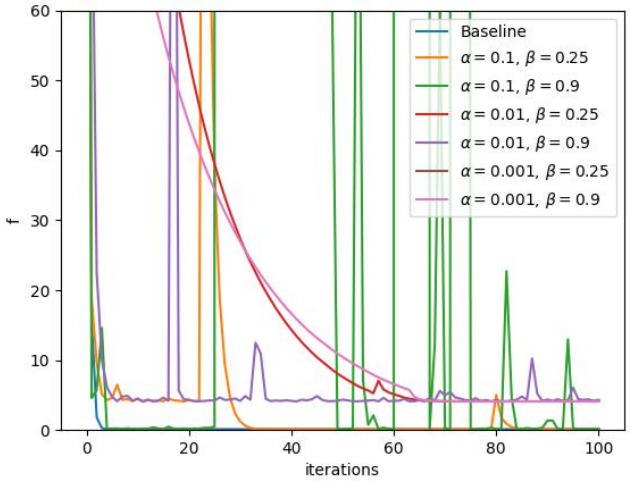


Baseline performs the best. In all cases the function converges to a minimum of 0, with some small changes after convergence. Among them, the function converges the fastest when the batch size is 15, and there is little difference in the convergence speed in other cases.

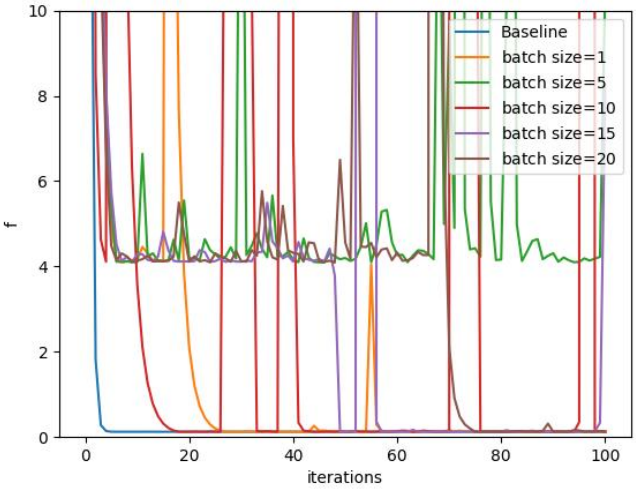


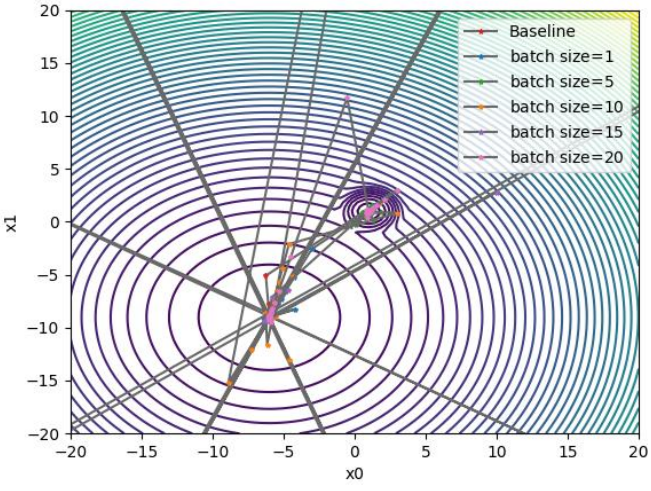


(iii) This question selects the HeavyBall algorithm. Firstly, according to the last assignment, I chose alpha to be 0.001, 0.01, 0.1, and beta to be 0.25 and 0.9 for experiments, and the results are shown in the figure below.



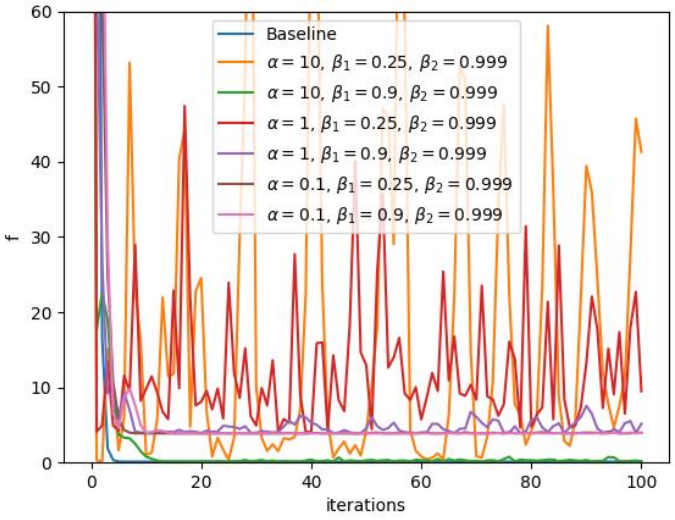
Although we can't find a suitable alpha and beta in our values, the function can converge to the minimum value of 0 and there is little fluctuation after convergence. But when alpha is 0.1 and beta is 0.25, the performance of function convergence is relatively best. So choose alpha to be 0.1 and beta to be 0.25.





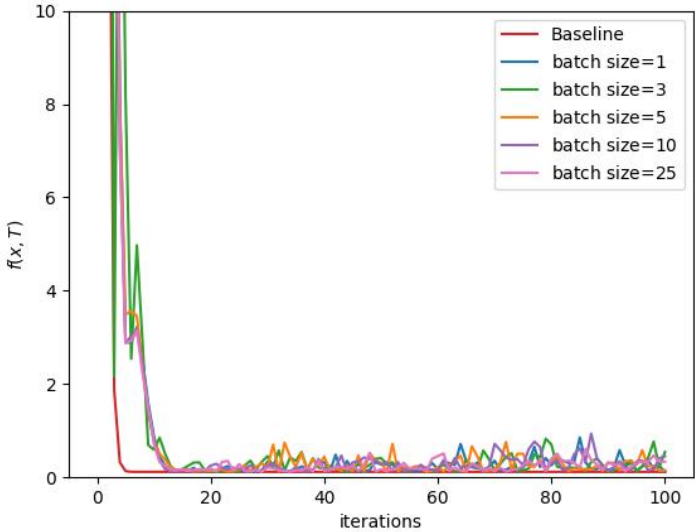
As can be seen from the figure above, the baseline is the best performer. In the value of batch size, when the batch size is 5, the function will no longer decrease when it drops to about 4. When the batch size is 10, the convergence is the fastest but very unstable. When the batch size is 1, the convergence is the second fastest, and the stability is better than the fastest. When the batch size is 20, the convergence is the slowest, but it is very stable after convergence.

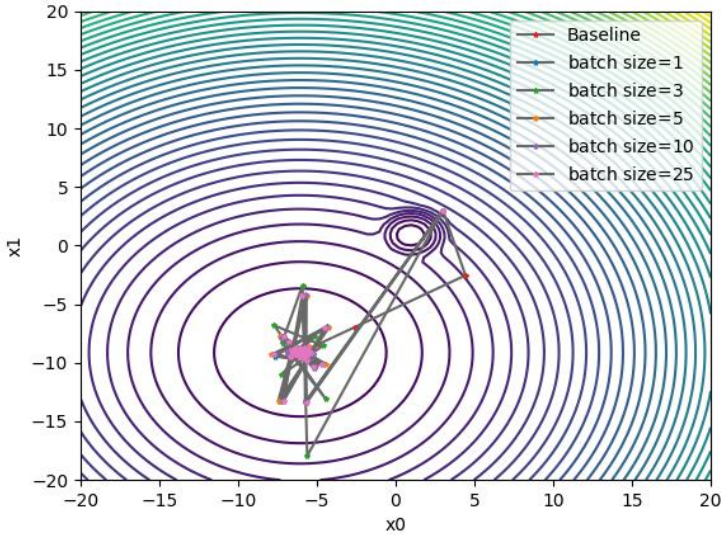
(iv) This question selects the Adam algorithm. Firstly, according to the last assignment, I chose alpha to be 0.1, 1, 10, beta1 to be 0.25 and beta2 to be 0.999 for experiments, and the results are shown in the figure below.



It can be seen from the figure that when alpha is 10, beta1 is 0.9 and beta2 is 0.999, the function converges to 0 the fastest and the fluctuation is small.

As can be seen from the figure below, the function converges to 0 at almost the same speed in all cases, but they are all unstable. Among them, Baseline and batch size are relatively stable at 25.





Appendix

Function

```
1. def generate_trainingdata(m=25):
2.     return np.array([0, 0]) + 0.25 * np.random.rand
   n(m, 2)
3.
4.
5. def f(x, minibatch):
6.     # loss function sum_{w in training data} f(x,w)

7.     y = 0
8.     count = 0
9.     for w in minibatch:
10.        z = x - w - 1
11.        y = y + min(33 * (z[0] ** 2 + z[1] ** 2), (
   z[0] + 10) ** 2 + (z[1] + 7) ** 2)
12.        count = count + 1
13.    return y / count
```

(a) (ii)

```
1. import matplotlib
2. matplotlib.use('TkAgg')
3. import matplotlib.pyplot as plt
4. import numpy as np
5.
6.
7. def generate_trainingdata(m=25):
8.     return np.array([0, 0]) + 0.25 * np.random.rand
   n(m, 2)
9.
10.
11.def f(x, minibatch):
12.    # loss function sum_{w in training data} f(x,w)

13.    y = 0
14.    count = 0
15.    for w in minibatch:
16.        z = x - w - 1
17.        y = y + min(33 * (z[0] ** 2 + z[1] ** 2), (
   z[0] + 10) ** 2 + (z[1] + 7) ** 2)
18.        count = count + 1
19.    return y / count
20.
```

```
21.
22. T = generate_trainingdata()
23. x = np.linspace(start=-20, stop=20, num=100)
24. y = np.linspace(start=-20, stop=20, num=100)
25. z = []
26. for i in x:
27.     k = []
28.     for j in y:
29.         params = [i, j]
30.         k.append(f(params, T))
31.     z.append(k)
32. z = np.array(z)
33. x, y = np.meshgrid(x, y)
34. contour_ax = plt.subplot(111, projection='3d')
35. contour_ax.contour3D(x, y, z, 60)
36. contour_ax.set_xlabel('X1')
37. contour_ax.set_ylabel('X2')
38. contour_ax.set_zlabel('F(x, T)')
39. contour_ax.set_title('Contour plot')
40. wireFrame_ax = plt.subplot(111, projection='3d')
41. wireFrame_ax.plot_wireframe(x, y, z, rstride=10, cs
    tride=5)
42. wireFrame_ax.set_xlabel('X1')
43. wireFrame_ax.set_ylabel('X2')
44. wireFrame_ax.set_zlabel('F(x, T)')
45. wireFrame_ax.set_title('WireFrame plot')
46. plt.show()
```

(a) (iii)

```
1. import matplotlib
2. matplotlib.use('TkAgg')
3. import matplotlib.pyplot as plt
4. import numpy as np
5. import sympy as sp
6.
7.
8. def generate_trainingdata(m=25):
9.     return np.array([0, 0]) + 0.25 * np.random.rand
   n(m, 2)
10.
11.
12. def f(x, minibatch):
13.     # loss function sum_{w in training data} f(x,w)
```

```
    y = 0
    count = 0
    for w in minibatch:
        z = x - w - 1
        y = y + min(33 * (z[0] ** 2 + z[1] ** 2), (z[0] + 10) ** 2 + (z[1] + 7) ** 2)
        count = count + 1
    return y / count


x0, x1, w0, w1 = sp.symbols('x0 x1 w0 w1', real=True)
# as z = x - w - 1
# y = y + min(33 * (z[0] ** 2 + z[1] ** 2), (z[0] + 10) ** 2 + (z[1] + 7) ** 2)
f = sp.Min(33 * ((x0-w0-1)**2 + (x1-w1-1)**2), (x0-w0+9)**2 + (x1-w1+6)**2)
df0 = sp.diff(f, x0)
df1 = sp.diff(f, x1)
print(df0)
print(df1)
```

(b)(i)

```
import matplotlib

matplotlib.use('TkAgg')
import matplotlib.pyplot as plt
import numpy as np
from copy import deepcopy
from enum import Enum


def generate_trainingdata(m=25):
    return np.array([0, 0]) + 0.25 * np.random.randn(m, 2)


def f(x, minibatch):
    # loss function sum_{w in training data} f(x,w)
    y = 0
    count = 0
    for w in minibatch:
        z = x - w - 1
```

```
            y = y + min(33 * (z[0] ** 2 + z[1] ** 2), (z[0] + 10) ** 2 + (z[1] + 7) ** 2)
            count = count + 1
        return y / count


class ALGORITHM(Enum):
    constant = 0
    polyak = 1
    rmsprop = 2
    heavyball = 3
    adam = 4


class SGD:
    def __init__(self, f, df, x, algorithm, params, batch_size, training_data):
            self.epsilon = 1e-8
            self.f = f
            self.df = df
            self.x = deepcopy(x)
            self.n = len(x)
            self.params = params
            self.batch_size = batch_size
            self.training_data = training_data
            self.records = {
                'x': [deepcopy(self.x)],
                'f': [self.f(self.x, self.training_data)],
                'step': []
            }
            self.algorithm = self.get_algorithm_via_enum_name(algorithm)
            self.initial(algorithm)

    def minibatch(self):
            # shuffle training data
            np.random.shuffle(self.training_data)
            n = len(self.training_data)
            for i in range(0, n, self.batch_size):
                if i + self.batch_size > n:
                    continue
                data = self.training_data[i:(i + self.batch_size)]
```

```
59.             self.algorithm(data)
60.         self.records['x'].append(deepcopy(self.x))
61.         self.records['f'].append(self.f(self.x, self.training_data))
62.
63.     def get_algorithm_via_enum_name(self, algorithm):
64.         if algorithm == ALGORITHM.constant:
65.             return self.constant
66.         elif algorithm == ALGORITHM.polyak:
67.             return self.polyak
68.         elif algorithm == ALGORITHM.rmsprop:
69.             return self.rmsprop
70.         elif algorithm == ALGORITHM.heavyball:
71.             return self.heavy_ball
72.         else:
73.             return self.adam
74.
75.     def initial(self, algorithm):
76.         if algorithm == ALGORITHM.rmsprop:
77.             self.records['step'] = [[self.params['alpha']] * self.n]
78.             self.vars = {
79.                 'sums': [0] * self.n,
80.                 'alphas': [self.params['alpha']] * self.n
81.             }
82.         elif algorithm == ALGORITHM.heavyball:
83.             self.records['step'] = [0]
84.             self.vars = {
85.                 'z': 0
86.             }
87.         elif algorithm == ALGORITHM.adam:
88.             self.records['step'] = [[0] * self.n]
89.             self.vars = {
90.                 'ms': [0] * self.n,
91.                 'vs': [0] * self.n,
92.                 'step': [0] * self.n,
93.                 't': 0
94.             }
95.
96.     def constant(self, data):
97.         alpha = self.params['alpha']
```

```
98.         for i in range(self.n):
99.             self.x[i] -= alpha * self.get_derivativ
  e(i, data)
100.         self.records['step'].append(alpha)
101.
102.     def polyak(self, data):
103.         Sum = 0
104.         for i in range(self.n):
105.             Sum = Sum + self.get_derivative(i, d
  ata) ** 2
106.         step = self.f(self.x, data) / (Sum + sel
  f.epsilon)
107.         for i in range(self.n):
108.             self.x[i] -= step * self.get_derivat
  ive(i, data)
109.         self.records['step'].append(step)
110.
111.     def rmsprop(self, data):
112.         alpha = self.params['alpha']
113.         beta = self.params['beta']
114.         alphas = self.vars['alphas']
115.         sums = self.vars['sums']
116.         for i in range(self.n):
117.             self.x[i] -= alphas[i] * self.get_de
  rivative(i, data)
118.             sums[i] = (beta * sums[i]) + ((1 - b
  eta) * (self.get_derivative(i, data) ** 2))
119.             alphas[i] = alpha / ((sums[i] ** 0.5)
   + self.epsilon)
120.         self.records['step'].append(deepcopy(alp
  has))
121.
122.     def heavy_ball(self, data):
123.         alpha = self.params['alpha']
124.         beta = self.params['beta']
125.         z = self.vars['z']
126.         Sum = 0
127.         for i in range(self.n):
128.             Sum += self.get_derivative(i, data)
  ** 2
129.
130.         z = (beta * z) + (alpha * self.f(self.x,
   data) / (Sum + self.epsilon))
131.         for i in range(self.n):
```

```
                self.x[i] -= z * self.get_derivative
(i, data)
            self.vars['z'] = z
            self.records['step'].append(z)

        def adam(self, data):
            alpha = self.params['alpha']
            beta1 = self.params['beta1']
            beta2 = self.params['beta2']
            ms = self.vars['ms']
            vs = self.vars['vs']
            step = self.vars['step']
            t = self.vars['t']
            t += 1
            for i in range(self.n):
                ms[i] = (beta1 * ms[i]) + ((1 - beta
1)*self.get_derivative(i, data))
                vs[i] = (beta2 * vs[i]) + ((1 - beta
2)*(self.get_derivative(i, data) ** 2))
                _m = ms[i] / (1 - (beta1 ** t))
                _v = vs[i] / (1 - (beta2 ** t))
                step[i] = alpha * (_m / ((_v ** 0.5)
 + self.epsilon))
                self.x[i] -= step[i]
            self.vars['t'] = t
             self.records['step'].append(deepcopy(ste
p))

        def get_derivative(self, i, data):
            Sum = 0
            for j in range(self.batch_size):
                Sum = Sum + self.df[i](*self.x, *dat
a[j])
            return Sum / self.batch_size


    def df0(x0, x1, w0, w1):
        return (-66 * w0 + 66 * x0 - 66) * np.heavis
ide(
             -33 * (-w0 + x0 - 1) ** 2 + (-w0 + x0 +
9) ** 2 - 33 * (-w1 + x1 - 1) ** 2 + (-w1 + x1 + 6)
 ** 2, 0) + (
                        -2 * w0 + 2 * x0 + 18) * np.h
eaviside(
```

```
            33 * (-w0 + x0 - 1) ** 2 - (-w0 + x0 + 9)
    ** 2 + 33 * (-w1 + x1 - 1) ** 2 - (-w1 + x1 + 6) *
   * 2, 0)


 def df1(x0, x1, w0, w1):
      return (-66 * w1 + 66 * x1 - 66) * np.heavis
   ide(
           -33 * (-w0 + x0 - 1) ** 2 + (-w0 + x0 +
   9) ** 2 - 33 * (-w1 + x1 - 1) ** 2 + (-w1 + x1 + 6)
    ** 2, 0) + (
                       -2 * w1 + 2 * x1 + 12) * np.h
   eaviside(
            33 * (-w0 + x0 - 1) ** 2 - (-w0 + x0 + 9)
    ** 2 + 33 * (-w1 + x1 - 1) ** 2 - (-w1 + x1 + 6) *
   * 2, 0)


 colors = ['tab:red', 'tab:blue', 'tab:green', 't
   ab:orange', 'tab:purple']


 def helper(f, training_data, xs, legend):
      X_Data = np.linspace(-20, 20, 100)
      Y_Data = np.linspace(-20, 20, 100)
      Z_Data = []
      for x in X_Data:
          z = []
           for y in Y_Data: z.append(f([x, y], trai
   ning_data))
          Z_Data.append(z)
      Z = np.array(Z_Data)
      X, Y = np.meshgrid(X_Data, Y_Data)
      plt.contour(X, Y, Z, 60)
      plt.xlabel('x0')
      plt.ylabel('x1')
      for i in range(len(xs)):
          x0 = [x[1] for x in xs[i]]
          x1 = [x[0] for x in xs[i]]
           plt.plot(x0, x1, color='dimgrey', marker
   ='*', markeredgecolor=colors[i], markersize=3)
          plt.xlim([-20, 20])
          plt.ylim([-20, 20])
      plt.legend(legend)
```

```
199.       plt.show()
200.
201.
202.  def b_i(f, df):
203.       training_data = generate_trainingdata()
204.       cnt = 100
205.       iters = list(range(cnt + 1))
206.       step_sizes = [0.1, 0.01, 0.001, 0.0001]
207.       labels = [f'step size = ${step_size}$' for s
   tep_size in step_sizes]
208.       xs = []
209.       fs = []
210.       for i, step_size in enumerate(step_sizes):
211.            sgd = SGD(f, df, [3, 3], ALGORITHM.const
   ant, {'alpha': step_size}, batch_size=len(training_
   data),
212.                      training_data=training_data)
213.           for _ in range(cnt):
214.               sgd.minibatch()
215.           plt.plot(iters, sgd.records['f'])
216.           xs.append(deepcopy(sgd.records['x']))
217.           fs.append(deepcopy(sgd.records['f']))
218.
219.       plt.xlabel('iterations')
220.       plt.ylabel('f')
221.       plt.legend(labels)
222.       plt.show()
223.       helper(f, training_data, xs, labels)
224.
225.
226.  if __name__ == '__main__':
227.       b_i(f, [df0, df1])
```

(b)(ii)

```
1. def b_ii(f, df):
2.      training_data = generate_trainingdata()
3.      times = 5
4.      cnt = 30
5.      iters = list(range(cnt + 1))
6.      alpha = 0.1
7.      labels = [f'Time ${i + 1}$' for i in range(time
   s)]
8.      xs = []
9.      fs = []
10.     for trial in range(times):
```

```
        sgd = SGD(f, df, [3, 3], ALGORITHM.constant, {'alpha': alpha}, 5, training_data)
        for _ in range(cnt):
            sgd.minibatch()
        plt.plot(iters, sgd.records['f'], label=labels[trial])
        xs.append(deepcopy(sgd.records['x']))
        fs.append(deepcopy(sgd.records['f']))
    plt.ylim([0, 20])
    plt.xlabel('iterations')
    plt.ylabel('f')
    plt.legend()
    plt.show()
    helper(f, training_data, xs, labels)
```

(b)(iii)

```
def b_iii(f, df):
    training_data = generate_trainingdata()
    cnt = 30
    iters = list(range(cnt + 1))
    alpha = 0.1
    batch_sizes = [1, 5, 10, 15, 20, 25]
    labels = [f'$batch size={n}$' for n in batch_sizes]
    xs = []
    fs = []
    for i, n in enumerate(batch_sizes):
        sgd = SGD(f, df, [3, 3], ALGORITHM.constant, {'alpha': alpha}, n, training_data)
        for _ in range(cnt):
            sgd.minibatch()
        plt.plot(iters, sgd.records['f'], label=labels[i])
        xs.append(deepcopy(sgd.records['x']))
        fs.append(deepcopy(sgd.records['f']))
    plt.ylim([0, 3])
    plt.xlabel('iterations')
    plt.ylabel('f')
    plt.legend()
    plt.show()
    helper(f, training_data, xs, labels)
```

(b)(iv)

```
def b_iv(f, df):
    training_data = generate_trainingdata()
    cnt = 30
```

```
    iters = list(range(cnt + 1))
    step_sizes = [0.1, 0.01, 0.001, 0.0001]
    labels = [f'step size = ${step_size}$' for step_size in step_sizes]
    xs = []
    fs = []
    for i, step_size in enumerate(step_sizes):
        sgd = SGD(f, df, [3, 3], ALGORITHM.constant, {'alpha': step_size}, 5, training_data)
        for _ in range(cnt):
            sgd.minibatch()
        plt.plot(iters, sgd.records['f'], label=labels[i])
        xs.append(deepcopy(sgd.records['x']))
        fs.append(deepcopy(sgd.records['f']))
    plt.ylim([0, 120])
    plt.xlabel('iterations')
    plt.ylabel('f')
    plt.legend()
    plt.show()
    helper(f, training_data, xs, labels)
```

(c)(i)

```
def c_i(f, df):
    training_data = generate_trainingdata()
    cnt = 100
    iters = list(range(cnt + 1))
    xs = []
    fs = []
    labels = ['Baseline']
    sgd_baseline = SGD(f, df, [3, 3], ALGORITHM.constant, {'alpha': 0.1}, 5, training_data)
    for _ in range(cnt):
        sgd_baseline.minibatch()
    plt.plot(iters, sgd_baseline.records['f'], label=labels[0])
    xs.append(deepcopy(sgd_baseline.records['x']))

    fs.append(deepcopy(sgd_baseline.records['f']))

    batch_sizes = [1, 5, 10, 15, 20]
    for n in batch_sizes:
        sgd = SGD(f, df, [3, 3], ALGORITHM.polyak, {}, n, training_data)
        for _ in range(cnt):
```

```
            sgd.minibatch()
        labels.append(f'batch size=${n}$')
        plt.plot(iters, sgd.records['f'], label=lab
els[-1])
        xs.append(deepcopy(sgd.records['x']))
        fs.append(deepcopy(sgd.records['f']))
    plt.ylim([0, 60])
    plt.xlabel('iterations')
    plt.ylabel('f')
    plt.legend()
    plt.show()
    helper(f, training_data, xs, labels)
```

(c)(ii)

```
def c_ii(f, df):
    training_data = generate_trainingdata()
    cnt = 100
    iters = list(range(cnt + 1))
    xs = []
    fs = []
    labels = ['Baseline']
    sgd_baseline = SGD(f, df, [3, 3], ALGORITHM.con
stant, {'alpha': 0.1}, 5, training_data)
    for _ in range(cnt):
        sgd_baseline.minibatch()
    plt.plot(iters, sgd_baseline.records['f'], labe
l=labels[0])
    xs.append(deepcopy(sgd_baseline.records['x']))

    fs.append(deepcopy(sgd_baseline.records['f']))

    alphas = [0.1, 0.01, 0.001]
    betas = [0.25, 0.9]
    for alpha in alphas:
        for beta in betas:
            sgd = SGD(f, df, [3, 3], ALGORITHM.rmsp
rop,
                      {'alpha': alpha, 'beta': beta}
, 5, training_data)
            for _ in range(cnt):
                sgd.minibatch()
            labels.append(f'$\\alpha={alpha},\\,\\b
eta={beta}$')
            plt.plot(iters, sgd.records['f'], label
=labels[-1])
```

```
            xs.append(deepcopy(sgd.records['x']))
            fs.append(deepcopy(sgd.records['f']))
    plt.ylim([0, 60])
    plt.xlabel('iterations')
    plt.ylabel('f')
    plt.legend()
    plt.show()
    xs = []
    fs = []
    labels = ['Baseline']
    plt.plot(iters, sgd_baseline.records['f'], label=labels[0])
    xs.append(deepcopy(sgd_baseline.records['x']))
    fs.append(deepcopy(sgd_baseline.records['f']))
    batch_sizes = [1, 5, 10, 15, 20]
    for batch_size in batch_sizes:
        sgd = SGD(f, df, [3, 3], ALGORITHM.rmsprop, {'alpha': 0.1, 'beta': 0.9}, 5, training_data)
        for _ in range(cnt):
            sgd.minibatch()
        labels.append(f'batch size=${batch_size}$')
        plt.plot(iters, sgd.records['f'], label=labels[-1])
        xs.append(deepcopy(sgd.records['x']))
        fs.append(deepcopy(sgd.records['f']))
    plt.ylim([0, 10])
    plt.xlabel('iterations')
    plt.ylabel('f')
    plt.legend()
    plt.show()
    helper(f, training_data, xs, labels)
```

(c)(iii)

```
def c_iii(f, df):
    training_data = generate_trainingdata()
    cnt = 100
    iters = list(range(cnt + 1))
    xs = []
    fs = []
    labels = ['Baseline']
    sgd_baseline = SGD(f, df, [3, 3], ALGORITHM.constant, {'alpha': 0.1}, 5, training_data)
```

```
    for _ in range(cnt):
        sgd_baseline.minibatch()
    plt.plot(iters, sgd_baseline.records['f'], label=labels[0])
    xs.append(deepcopy(sgd_baseline.records['x']))
    fs.append(deepcopy(sgd_baseline.records['f']))
    alphas = [0.1, 0.01, 0.001]
    betas = [0.25, 0.9]
    for alpha in alphas:
        for beta in betas:
            sgd = SGD(f, df, [3, 3], ALGORITHM.heavyball,
                      {'alpha': alpha, 'beta': beta}, 5, training_data)
            for _ in range(cnt):
                sgd.minibatch()
            labels.append(f'$\\alpha={alpha},\\,\\beta={beta}$')
            plt.plot(iters, sgd.records['f'], label=labels[-1])
            xs.append(deepcopy(sgd.records['x']))
            fs.append(deepcopy(sgd.records['f']))
    plt.ylim([0, 60])
    plt.xlabel('iterations')
    plt.ylabel('f')
    plt.legend()
    plt.show()
    xs, fs = [], []
    labels = ['Baseline']
    plt.plot(iters, sgd_baseline.records['f'], label=labels[0])
    xs.append(deepcopy(sgd_baseline.records['x']))
    fs.append(deepcopy(sgd_baseline.records['f']))
    batch_sizes = [1, 5, 10, 15, 20]
    for batch_size in batch_sizes:
        sgd = SGD(f, df, [3, 3], ALGORITHM.heavyball, {'alpha': 0.1, 'beta': 0.25}, 5, training_data)
        for _ in range(cnt):
            sgd.minibatch()
```

```
41.         labels.append(f'batch size=${batch_size}$')
42.         plt.plot(iters, sgd.records['f'], label=lab
    els[-1])
43.         xs.append(deepcopy(sgd.records['x']))
44.         fs.append(deepcopy(sgd.records['f']))
45.     plt.ylim([0, 10])
46.     plt.xlabel('iterations')
47.     plt.ylabel('f')
48.     plt.legend()
49.     plt.show()
50.     helper(f, training_data, xs, labels)
```

(c)(iv)

```
1. def c_iv(f, df):
2.     training_data = generate_trainingdata()
3.     cnt = 100
4.     iters = list(range(cnt + 1))
5.     xs = []
6.     fs = []
7.     labels = ['Baseline']
8.     sgd_baseline = SGD(f, df, [3, 3], ALGORITHM.con
    stant, {'alpha': 0.1}, 5, training_data)
9.     for _ in range(cnt):
10.         sgd_baseline.minibatch()
11.     plt.plot(iters, sgd_baseline.records['f'], labe
    l=labels[0])
12.     xs.append(deepcopy(sgd_baseline.records['x']))
13.     fs.append(deepcopy(sgd_baseline.records['f']))
14.     alphas = [10, 1, 0.1]
15.     beta1s = [0.25, 0.9]
16.     beta2s = [0.999]
17.     for alpha in alphas:
18.         for beta1 in beta1s:
19.             for beta2 in beta2s:
20.                 sgd = SGD(f, df, [3, 3], ALGORITHM.
    adam,
21.                           {'alpha': alpha, 'beta1':
     beta1, 'beta2': beta2}, 5, training_data)
22.                 for _ in range(cnt):
23.                     sgd.minibatch()
24.                 labels.append(
```

```
                    f'$\\alpha={alpha},\\,\\beta_1={beta1},\\,\\beta_2={beta2}$'
                )
                plt.plot(iters, sgd.records['f'], label=labels[-1])
                xs.append(deepcopy(sgd.records['x']))
                fs.append(deepcopy(sgd.records['f']))
    plt.ylim([0, 60])
    plt.xlabel('iterations')
    plt.ylabel('f')
    plt.legend()
    plt.show()
    xs = []
    fs = []
    labels = ['Baseline']
    plt.plot(iters, sgd_baseline.records['f'], label=labels[0])
    xs.append(deepcopy(sgd_baseline.records['x']))
    fs.append(deepcopy(sgd_baseline.records['f']))
    batch_sizes = [1, 3, 5, 10, 25]
    for batch_size in batch_sizes:
        sgd = SGD(f, df, [3, 3], ALGORITHM.adam,
                  {'alpha': 10, 'beta1': 0.9, 'beta2': 0.999}, 5, training_data)
        for _ in range(cnt):
            sgd.minibatch()
        labels.append(f'batch size=${batch_size}$')
        plt.plot(iters, sgd.records['f'], label=labels[-1])
        xs.append(deepcopy(sgd.records['x']))
        fs.append(deepcopy(sgd.records['f']))
    plt.ylim([0, 10])
    plt.xlabel('iterations')
    plt.ylabel('$f(x, T)$')
    plt.legend()
    plt.show()
    helper(f, training_data, xs, labels)
```