Xiangyu Zheng 21331025 CS7CS2 Optimisation for Machine Learning week6 assignment

(a)(i)

```
def miniBatchSGD(x, max_iter, batch_size, data, algo_func, *params):
```

The function has some parameters:

x is the point to optimize and is an array. max_iter is the maximum number of iterations.

batch_size controls the size of small samples. data is the complete dataset.

algo_func is the name of the function of the algorithm to use. *params are some parameters required by the algorithm such as alpha, beta.

First, the function has the first layer of for loop and will stop when the maximum number of iterations is reached. After that shuffle the data:

```
while i < max_iter:
        np.random.shuffle(data)
```

The second layer loop sets a current i to control the index of the small sample. There is also an end index, which controls the length of the small sample. If end exceeds the entire data limit, it will be set to the length of the entire data set.

```
c_i = 0
while c_i < dataLen:
    end = c_i + batch_size
    if end > dataLen:
        end = dataLen
```

Get a small sample based on index:

```
sample_data = data[c_i:end]
```

Call the algorithm and pass in the parameters

```
x, sums = algo_func(x, sample_data, sums, *params)
```

The algorithms included in algo_func are constant step size, Polyak, RMSProp, Heavy Ball and Adam steps

```
def consStepSize(x, minibatch, sums, *param):
def polyak(x, minibatch, sums, *param):
def RMSProp(x, minibatch, sums, *param):
def heavyBall(x, minibatch, sums, *param):
def Adam(x, minibatch, sums, *param):
```

(a)(ii)

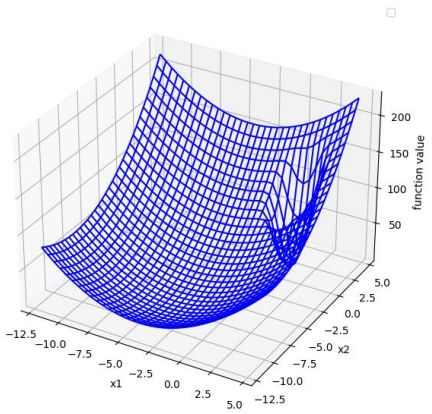

Figure 1 wireframe diagram of the function

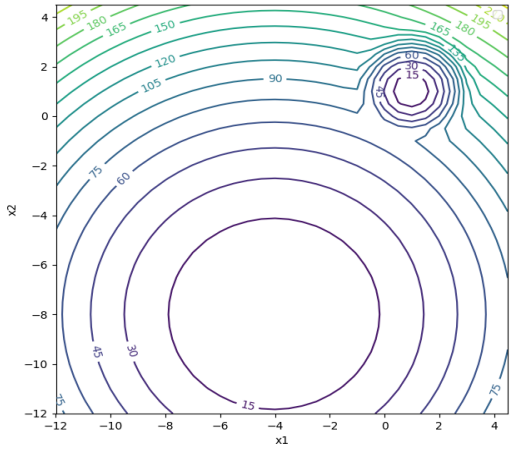

Figure 2 The contour plot of the function

We can see from Figures 1 and 2 that I chose values for both x1 and x2 from 5 to -12.5, because this makes it clear that the function has a small bowl (local minima) and a large bowl (global minimum)

(a)(iii)

I am using the finite difference method to calculate the partial derivatives of x and y.

According to the formula:

$$\frac{\partial u}{\partial x} = \frac{f(x+h,y) - f(x,y)}{h} \; and \; \frac{\partial u}{\partial y} = \frac{f(x,y+h) - f(x,y)}{h}$$

Pass in x1, x2 and a small sample to calculate the partial derivative value:

```
def getDx(x, minibatch):
    x1 = [x[0] + delta, x[1]]
    return (f(x1, minibatch) - f(x, minibatch)) / delta
```

```
def getDy(x, minibatch):
    x1 = [x[0], x[1] + delta]
    return (f(x1, minibatch) - f(x, minibatch)) / delta
```

(b)(i)

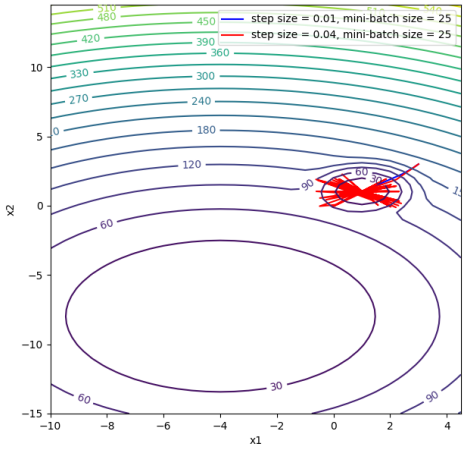

Figure 3 contour plot with step size of 0.01 and 0.04

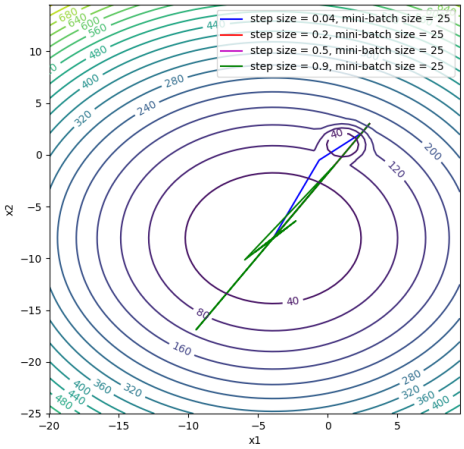

Figure 4 contour plot with step size of 0.04, 0.2, 0.5, 0.9

The step size I selected is 0.04, and we can see from Figure 3 that when the step size=0.04, the function cannot go out of the local minima, and it oscillates a lot in it. As can be seen in Figure 4, when the randomly generated data set changes, the function value can get out of the local minima, and gradually iterate to the global optimum. So this step size is worth studying.

(b)(ii)

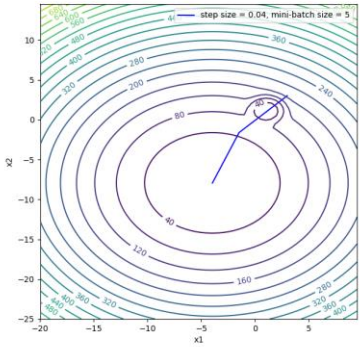

Figure 5 contour plot with step size = 0.04 and batch size = 5

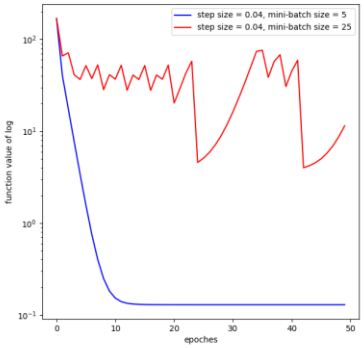

Figure 6 function value is compared between batch size = 5 and 25

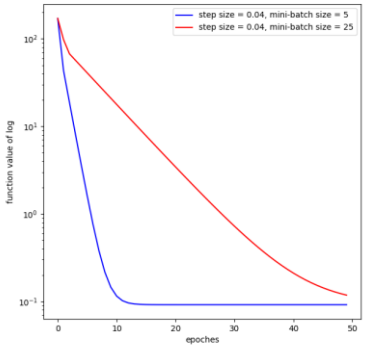

Figure 7 function value is compared between batch size = 5 and 25

I tried running the program multiple times, the local minima did not oscillate when batch size = 5, and the probability is very high when batch size = 25. This is because the mini-batch samples are small, resulting in a large deviation of the partial derivative of the approximate estimation of noise influence, but this is beneficial to get out of the local minima. We can see from Figure 6 and Figure 7. We can also see that when batch size = 5, the convergence speed is significantly faster. This is because when batch size = 5, there are 5 iterations per epoch, which is 4 more than when batch size = 25, and the convergence is faster.

(b)(iii)

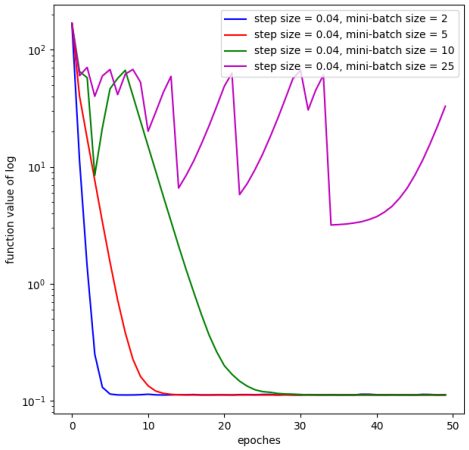

Figure 8 Change of function value with step size = 0.04 and different batch sizes

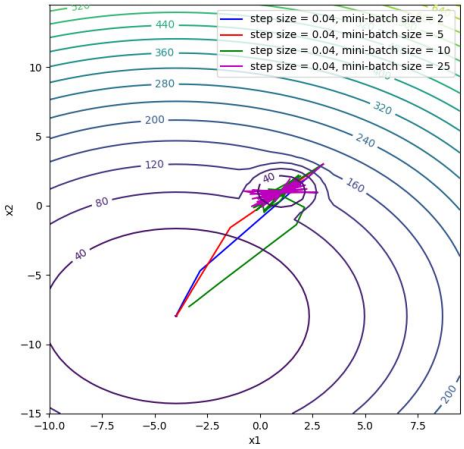

Figure 9 Change of function value with x1 and x2 and step size = 0.04 and different batch size in contour plot

As can be seen from Figures 8 and 9, as the batch size increases, the iteration speed per epoch starts to slow down. This is because the small batch of samples increases the number of iterations per epoch. In Figure 9, as the batch size increases to 10, there are still local iterations. But it finally came out. When the batch size is increased to 25, the function cannot get out of the local minimization. This is because the function is approximated, the smaller the sample is less accurate, the more noise is introduced, but this helps the function get out of the local minimization. Appropriate batch size instead brings invisible regularization.

(b)(iv)

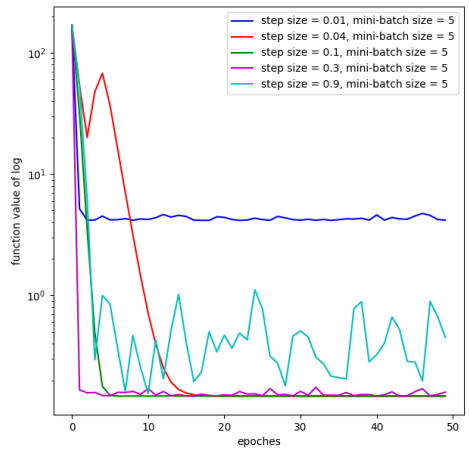

Figure 10 The function value changes with mini batch size = 5 and different step sizes

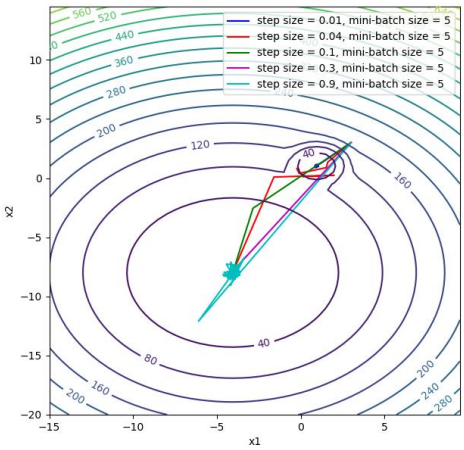

Figure 11 Change of function value with x1 and x2 and mini batch size = 5 and different step size in contour plot

We can see from Figure 10 that when step size = 0.01, the function has been unable to converge to the global minimum at the local minimum. When step size = 0.04, the function wanders in the local minimum area, but leaves, and eventually converges to the global minimum, which is different from when mini batch size = 25 in iii, and sometimes cannot get out of the local minimum. and mini batch size = 10, less jitter. We can find that when the step size is gradually increased, the iteration speed can also be increased, and it can also help to quickly get out of the local minimum. But when the step size is too large, there will be vibrations around the global minimum, which can also be

seen from Figure 11.

(c)(i) Polyak step size

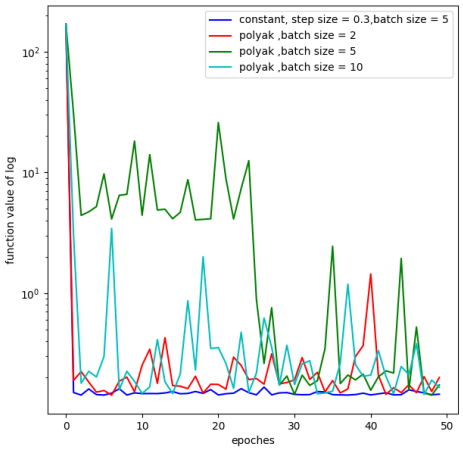

Figure 12 Function value changes are compared in polyak with different batch sizes and constants

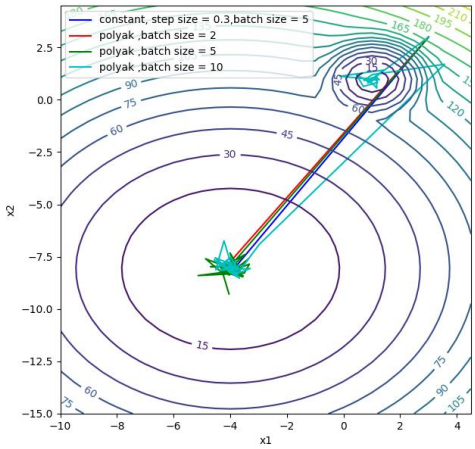

Figure 13 The function value changes in the polyak with different batch sizes and constants compared in the contour plot

The baseline algorithm I chose is constant step size. The parameters are step size = 0.3, batch size = 5 because these parameters perform the best in b.

We can see from Figures 12 and 13 that in polyak, as the batch size increases, it will linger in the local optimization for a while, and then it will gradually converge to the global minimum, but the vibration is very strong in the vicinity, but the convergence speed If not in Wandering in the local optimum, there seems to be no significant change. And the jitter seems to be more severe as the batch size increases.

We can according to polyak's step size formula:$\alpha = \frac{\mathrm{f(x)}-f^*}{\sum_{i=1}^{n}\frac{af}{ax_i}(x)^2}$. Due to the approximate derivative algorithm, the square in the denominator will aggravate the influence of noise, so the step size may be very small under certain circumstances, resulting in a local situation in Figure 13. According to the formula, the step size of polyak will be constant at a certain value, and it is unlikely to be very small. In this data set, a large step size causes a large vibration near the global optimum. As for the case of increasing the batch size with severe vibration, due to the random strategy, the small batch size will iterate more times in the epoch and will be closer to the real value.

(c)(ii)

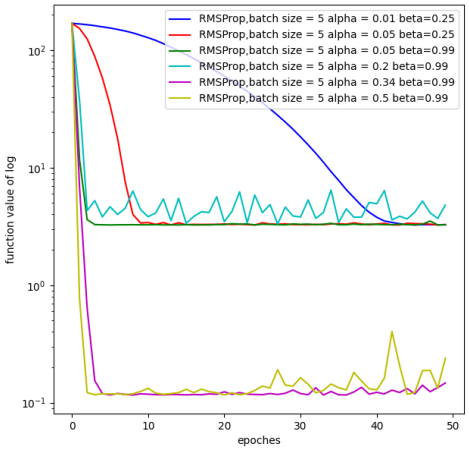

Figure 14 function value varies with different alpha and bate

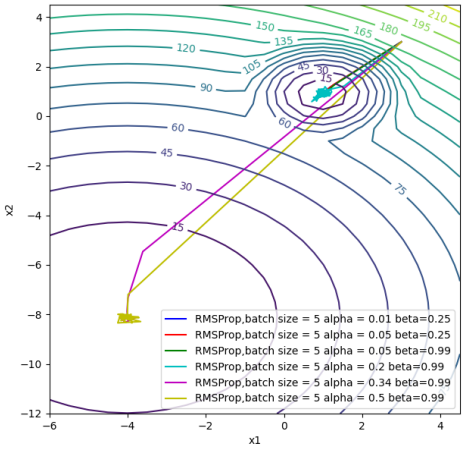

Figure 15 x1 and x2 vary with different alpha and bate

We can see from Figures 14 and 15 that increasing both alpha and beta increases iteration speed and oscillation, and requires tuning both parameters to get out of the local optimization.

This is because according to the formula of RMSProp:

$$\alpha_t = \frac{\alpha_0}{\sqrt{(1-\beta)\beta^t \frac{df}{dx}(x_0)^2 + (1-\beta)\beta^{t-1}\frac{df}{dx}(x_1)^2 + \cdots + (1-\beta)\frac{df}{dx}(x_{t-1})^2 + \epsilon}}$$

Alpha0 and beta will act on the step size at the same time, that is, the larger the alpha0 is, the larger the step size will be. The larger the beta, the more attention will be paid to the historical data, and the historical data will be small at the beginning, which will lead to the beginning of the step size will greatly help out of the local optimization. , as the function becomes smaller in the later stage, the historical data also begins to become smaller, which also leads to a larger step size, which will oscillate near the global optimum, which can be seen from the yellow lines in Figures 14 and 15.

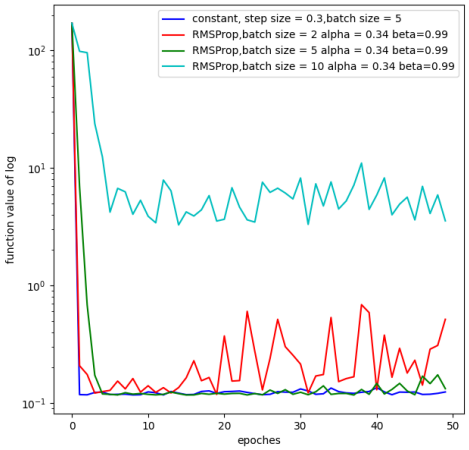

Figure 16 function value varies with fixed alpha and bate and with different batch sizes

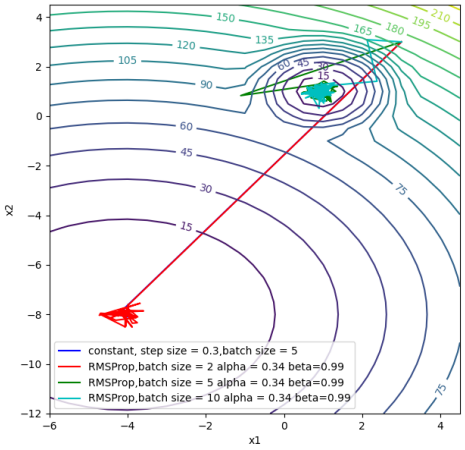

Figure 17 x1 and x2 vary with fixed alpha and bate and different batch sizes

From Figures 14 and 15, select a more appropriate value alpha=0.34 beta=0.99 to see how the batch size is adjusted. From Figure 16 and Figure 17, we can see that as the batch size increases, it is also possible to fall into a local minimum, which is the same as b(iii). Local optimization. But we see that the smaller batch size fluctuates a lot, because the

smaller batch size has a larger noise effect and the increase in the number of iterations, which causes the approximate derivative to be inaccurate and recorded in the historical data many times.

(c)(iii)heavyBall

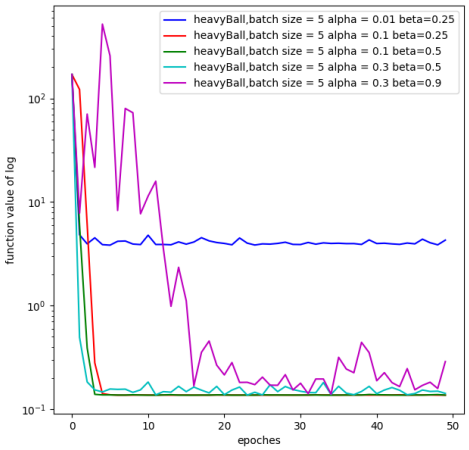

Figure 18 Change of heavyBall function value with batch size =5 and different alpha and beta

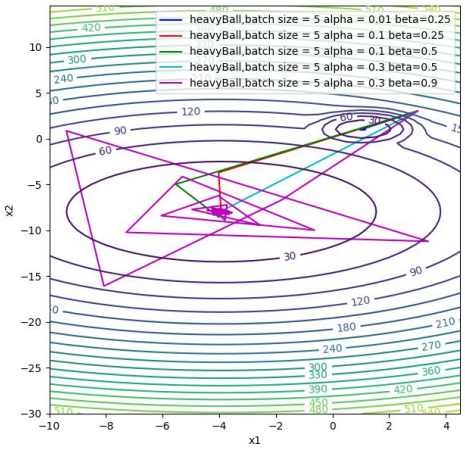

Figure 19 x1 and x2 vary with fixed batch size and different alpha and beta

From Figures 18 and 19, we can see that when the alpha and beta values are small, the function value cannot go out of the local area. As alpha increases, according to heavyball's step formula: beta * z + alpha * dx/dy, both beta and alpha increase. Can increase the step, which will increase the iteration speed, but when the alpha or beta is too large, the step will be larger, which will increase the oscillation.

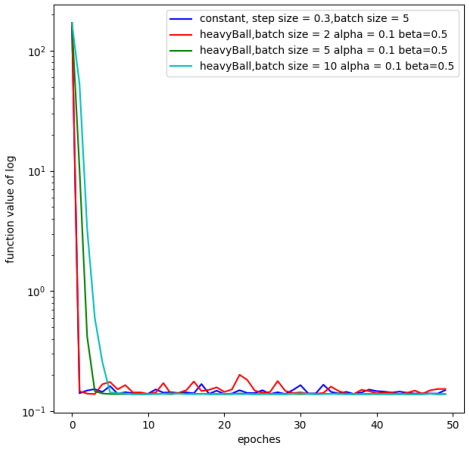

Figure 20 different batch size and fixed alpha and beta in heavy ball

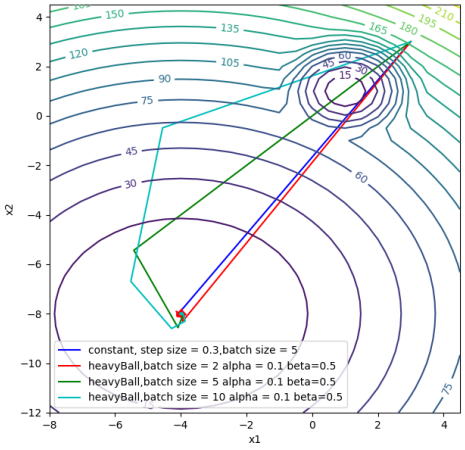

Figure 21 Changes in x and y with different batch sizes and fixed alpha and beta

Choosing moderate alpha and beta, compared with the constant step size, we found that the function value fluctuated a little when increasing the batch size. This is because the batch size of the small batch has more noise, and the number of iterations is increased, which is added to the historical data. There is no square operation to obtain an approximate function, and the impact is smaller than c(i) and c (ii). Relative constant step size is also as bad as c(i) and c(ii).

(c)(iv) adam

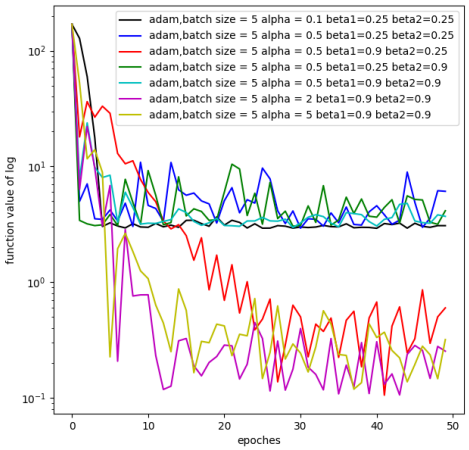

Figure 22 The function value of adam varies with batch size=5 and different alpha, beta1, beta2

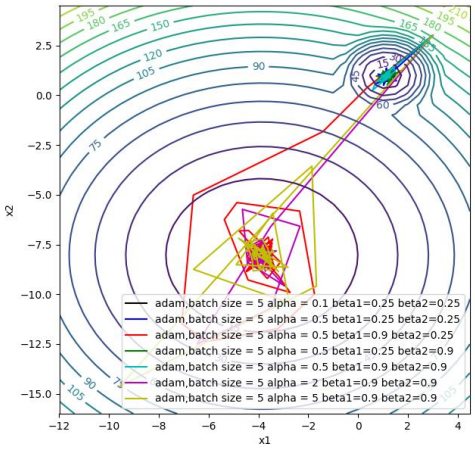

Figure 23 x1 and x2 in adam with batch size=5 and different alpha, beta1, beta2

We can see from Figures 22 and 23 that it is difficult for small alpha values in adam to converge to the global optimum, because according to the formula:

$$\hat{m} = \frac{m_{t+1}}{(1-\beta_1^t)}, \hat{v} = \frac{v_{t+1}}{(1-\beta_2^t)}$$

And

$$x_{t+1} = x_t - \alpha \frac{\widehat{m_1}}{\sqrt{\widehat{v_1}} - \epsilon}$$

After the correction of step, the value is relatively smooth, and then according to the formula:

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

And

$$v_{t+1} = \beta_2 v_t + (1-\beta_2)\frac{\partial f}{\partial x_1}(x_t)^2$$

It can be seen that beta1 affects m, that is, the molecule. The larger the value, the more attention should be paid to historical data. In some cases, the step value will be larger, and beta2 affects v on the contrary.

We can also see from the figure that when beta1=0.9 and beta2=0.25, I have tried many times with a certain probability to converge to the global minimum. When beta1=0.25 and beta2=0.9 and beta1 and beta2 are equivalent, according to the formula, the step in these cases is relatively small and gentle. Can't get out of local optimization.

We can also increase the step to get out of the local optimization by increasing the alpha method, which can also improve the convergence speed, but the oscillation will also increased.

Select the parameters alpha =2 beta1=0.9 and beta2=0.9 with better performance from Figure 22 to conduct experiments under different batch sizes and compare the constant step size to get the following figure:

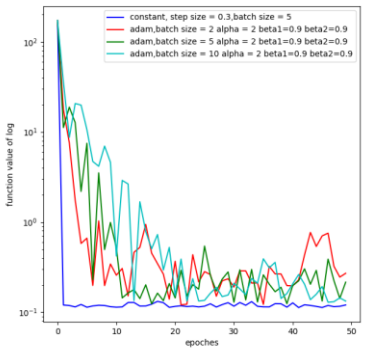

Figure 24 Variation of the value of adam function with fixed alpha, beta1 and beta2 and different batch sizes

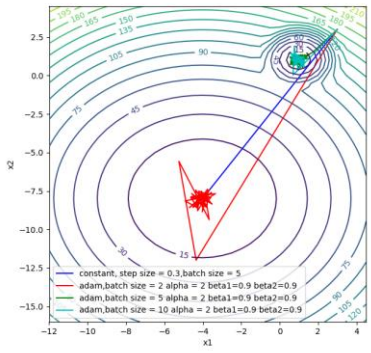

Figure 25 The changes of adam x1 and x2 under fixed alpha, beta1 and beta2 and different batch sizes have a certain probability that they cannot converge to the global optimum

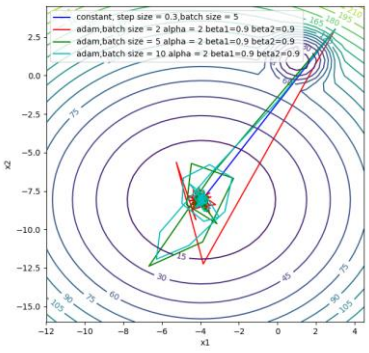

Figure 26 Changes of adam x1 and x2 under fixed alpha, beta1 and beta2 and different batch sizes

We can see from Figures 24, 25 and 26 that the function converges a little faster as the value of batch size gets smaller, because the number of iterations increases. But the jitter does not improve because adam corrects the step value as described in Figures 22 and 23. This proves that adam can control the impact of noise very well.

Appendix

Code

```
1.  import math
2.
3.  import numpy as np
4.  from matplotlib import pyplot as plt
5.
6.
7.  def generate_trainingdata(m=25):
8.      return np.array([0, 0]) + 0.25 * np.random.randn(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(28 * (z[0] ** 2 + z[1] ** 2), (z[0] + 5) ** 2 + (z[1] + 9
    ) ** 2)
```

```
        count = count + 1
    return y / count


delta = 0.001


def getDx(x, minibatch):
    x1 = [x[0] + delta, x[1]]
    return (f(x1, minibatch) - f(x, minibatch)) / delta


def getDy(x, minibatch):
    x1 = [x[0], x[1] + delta]
    return (f(x1, minibatch) - f(x, minibatch)) / delta


def consStepSize(x, minibatch, sums, *param):
    alpha = param[0]
    dx = getDx(x, minibatch)
    dy = getDy(x, minibatch)
    x[0] = x[0] - alpha * dx
    x[1] = x[1] - alpha * dy
    return x, sums


def polyak(x, minibatch, sums, *param):
    epsilon = param[0]
    numerator = f(x, minibatch) - 0
    dx = getDx(x, minibatch)
    dy = getDy(x, minibatch)
    alpha = numerator / (dx ** 2 + dy ** 2 + epsilon)
    print(alpha)
    x[0] = x[0] - alpha * dx
    x[1] = x[1] - alpha * dy
    return x, sums


def RMSProp(x, minibatch, sums, *param):
    alpha0 = param[0]
    beta = param[1]
    epsilon = param[2]
    dx = getDx(x, minibatch)
    dy = getDy(x, minibatch)
```

```
    sums[0] = beta * sums[0] + (1 - beta) * (dx ** 2)
    sums[1] = beta * sums[1] + (1 - beta) * (dy ** 2)

    alpha = alpha0 / (math.sqrt(sums[0]) + epsilon)
    alpha2 = alpha0 / (math.sqrt(sums[1]) + epsilon)
    print(alpha)
    x[0] = x[0] - alpha * dx
    x[1] = x[1] - alpha2 * dy
    return x, sums


def heavyBall(x, minibatch, sums, *param):
    alpha = param[0]
    beta = param[1]
    dx = getDx(x, minibatch)
    dy = getDy(x, minibatch)
    sums[0] = beta * sums[0] + alpha * dx
    sums[1] = beta * sums[1] + alpha * dy
    x[0] = x[0] - sums[0]
    x[1] = x[1] - sums[1]
    return x, sums


def adam(x, minibatch, sums, *param):
    alpha = param[0]
    beta1 = param[1]
    beta2 = param[2]
    epsilon = param[3]
    dx = getDx(x, minibatch)
    dy = getDy(x, minibatch)
    # mt
    sums[0] = beta1 * sums[0] + (1 - beta1) * dx
    # mt2
    sums[1] = beta1 * sums[1] + (1 - beta1) * dy
    # vt
    sums[2] = beta2 * sums[2] + (1 - beta2) * (dx ** 2)
    # vt2
    sums[3] = beta2 * sums[3] + (1 - beta2) * (dy ** 2)
       # i
       me = sums[0] / (1 - beta1 ** (sums[4] + 1))
       me2 = sums[1] / (1 - beta1 ** (sums[4] + 1))
       ve = sums[2] / (1 - beta2 ** (sums[4] + 1))
       ve2 = sums[3] / (1 - beta2 ** (sums[4] + 1))
       x[0] = x[0] - alpha * (me / (math.sqrt(ve) + epsilon))
```

```
106.       print((me / (math.sqrt(ve) + epsilon)))
107.       x[1] = x[1] - alpha * (me2 / (math.sqrt(ve2) + epsilon))
108.       sums[4] = sums[4] + 1
109.       return x, sums
110.
111.
112.   def miniBatchSGD(x, max_iter, batch_size, data, ii, xx, yy, ff, algo_func,
     *params):
113.       i = 0
114.       j = 0
115.       dataLen = len(data)
116.       sums = [0, 0, 0, 0, 0]
117.       while i < max_iter:
118.           np.random.shuffle(data)
119.           ii.append(i)
120.           xx.append(x[0])
121.           yy.append(x[1])
122.           ff.append(f(x, data))
123.           c_i = 0
124.           while c_i < dataLen:
125.               # ii.append(j)
126.               # xx.append(x[0])
127.               # yy.append(x[1])
128.               # ff.append(f(x, data))
129.               end = c_i + batch_size
130.               if end > dataLen:
131.                   end = dataLen
132.               sample_data = data[c_i:end]
133.               x, sums = algo_func(x, sample_data, sums, *params)
134.               c_i = c_i + batch_size
135.               j = j + 1
136.           i = i + 1
137.       return ii, xx, yy, ff
138.
139.
140.   data = generate_trainingdata()
141.
142.   ii, xx, yy, ff = miniBatchSGD([3, 3], 50, 5, data, [], [], [], [], consSte
    pSize, 0.3)
143.   # polyak
144.   # ii1, xx1, yy1, ff1 = miniBatchSGD([3, 3], 50, 2, data, [], [], [], [], p
    olyak, 0.001)
145.   # ii2, xx2, yy2, ff2 = miniBatchSGD([3, 3], 50, 5, data, [], [], [], [], p
    olyak, 0.001)
```

```
146.   # ii3, xx3, yy3, ff3 = miniBatchSGD([3, 3], 50, 10, data, [], [], [], [],
    polyak, 0.001)
147.
148.   # RMSProp
149.   # ii1, xx1, yy1, ff1 = miniBatchSGD([3, 3], 50, 5, data, [], [], [], [], R
    MSProp, 0.01,0.25,0.001)
150.   # ii2, xx2, yy2, ff2 = miniBatchSGD([3, 3], 50, 5, data, [], [], [], [], R
    MSProp, 0.05,0.25,0.001)
151.   # ii3, xx3, yy3, ff3 = miniBatchSGD([3, 3], 50, 5, data, [], [], [], [], R
    MSProp, 0.05,0.99,0.001)
152.   # ii4, xx4, yy4, ff4 = miniBatchSGD([3, 3], 50, 5, data, [], [], [], [], R
    MSProp, 0.2,0.99,0.001)
153.   # ii5, xx5, yy5, ff5 = miniBatchSGD([3, 3], 50, 5, data, [], [], [], [], R
    MSProp, 0.34,0.99,0.001)
154.   # ii6, xx6, yy6, ff6 = miniBatchSGD([3, 3], 50, 5, data, [], [], [], [], R
    MSProp, 0.34,0.999,0.001)
155.   # ii1, xx1, yy1, ff1 = miniBatchSGD([3, 3], 50, 2, data, [], [], [], [], R
    MSProp, 0.34,0.99,0.001)
156.   # ii2, xx2, yy2, ff2 = miniBatchSGD([3, 3], 50, 5, data, [], [], [], [], R
    MSProp, 0.34,0.99,0.001)
157.   # ii3, xx3, yy3, ff3 = miniBatchSGD([3, 3], 50, 10, data, [], [], [], [],
    RMSProp, 0.34,0.99,0.001)
158.
159.   # heavyBall
160.   # ii1, xx1, yy1, ff1 = miniBatchSGD([3, 3], 50, 5, data, [], [], [], [], h
    eavyBall, 0.01, 0.25)
161.   # ii2, xx2, yy2, ff2 = miniBatchSGD([3, 3], 50, 5, data, [], [], [], [], h
    eavyBall, 0.1, 0.25)
162.   # ii3, xx3, yy3, ff3 = miniBatchSGD([3, 3], 50, 5, data, [], [], [], [], h
    eavyBall, 0.1, 0.5)
163.   # ii4, xx4, yy4, ff4 = miniBatchSGD([3, 3], 50, 5, data, [], [], [], [], h
    eavyBall, 0.3, 0.5)
164.   # ii5, xx5, yy5, ff5 = miniBatchSGD([3, 3], 50, 5, data, [], [], [], [], h
    eavyBall, 0.3, 0.9)
165.   #
166.   # ii1, xx1, yy1, ff1 = miniBatchSGD([3, 3], 50, 2, data, [], [], [], [], h
    eavyBall, 0.1, 0.5)
167.   # ii2, xx2, yy2, ff2 = miniBatchSGD([3, 3], 50, 5, data, [], [], [], [], h
    eavyBall, 0.1, 0.5)
168.   # ii3, xx3, yy3, ff3 = miniBatchSGD([3, 3], 50, 10, data, [], [], [], [],
    heavyBall, 0.1, 0.5)
169.
170.   # adam
```

```
171.   # ii0, xx0, yy0, ff0 = miniBatchSGD([3, 3], 50, 5, data, [], [], [], [], a
    dam, 0.1, 0.25,0.25,0.0001)
172.   # ii1, xx1, yy1, ff1 = miniBatchSGD([3, 3], 50, 5, data, [], [], [], [], a
    dam, 0.5, 0.25,0.25,0.0001)
173.   # ii2, xx2, yy2, ff2 = miniBatchSGD([3, 3], 50, 5, data, [], [], [], [], a
    dam, 0.5, 0.9,0.25,0.0001)
174.   # ii3, xx3, yy3, ff3 = miniBatchSGD([3, 3], 50, 5, data, [], [], [], [], a
    dam, 0.5, 0.25,0.9,0.0001)
175.   # ii4, xx4, yy4, ff4 = miniBatchSGD([3, 3], 50, 5, data, [], [], [], [], a
    dam, 0.5, 0.9,0.9,0.0001)
176.   # ii5, xx5, yy5, ff5 = miniBatchSGD([3, 3], 50, 5, data, [], [], [], [], a
    dam, 2, 0.9,0.9,0.0001)
177.   # ii6, xx6, yy6, ff6 = miniBatchSGD([3, 3], 50, 5, data, [], [], [], [], a
    dam, 5, 0.9,0.9,0.0001)
178.
179.   ii1, xx1, yy1, ff1 = miniBatchSGD([3, 3], 50, 2, data, [], [], [], [], ada
    m, 2, 0.9,0.9,0.0001)
180.   ii2, xx2, yy2, ff2 = miniBatchSGD([3, 3], 50, 5, data, [], [], [], [], ada
    m, 2, 0.9,0.9,0.0001)
181.   ii3, xx3, yy3, ff3 = miniBatchSGD([3, 3], 50, 10, data, [], [], [], [], ad
    am, 2, 0.9,0.9,0.0001)
182.
183.   x = np.arange(-12, 5, 0.5)
184.   y = np.arange(-16, 5, 0.5)
185.
186.   plt.figure(figsize=(7, 7))
187.   # polyak
188.   # plt.plot(ii, ff, color='b', label='constant, step size = 0.3,batch size
    = 5')
189.   # plt.plot(ii2, ff2, color='g', label='polyak ,batch size = 5')
190.   # plt.plot(ii3, ff3, color='c', label='polyak ,batch size = 10')
191.   # plt.plot(xx, yy, color='b', label='constant, step size = 0.3,batch size
    = 5')
192.   # plt.plot(xx1, yy1, color='r', label='polyak ,batch size = 2')
193.   # plt.plot(xx2, yy2, color='g', label='polyak ,batch size = 5')
194.   # plt.plot(xx3, yy3, color='c', label='polyak ,batch size = 10')
195.
196.   # RMSProp
197.   # plt.plot(ii1, ff1, color='b', label='RMSProp,batch size = 5 alpha = 0.01
     beta=0.25')
198.   # plt.plot(ii2, ff2, color='r', label='RMSProp,batch size = 5 alpha = 0.05
     beta=0.25')
199.   # plt.plot(ii3, ff3, color='g', label='RMSProp,batch size = 5 alpha = 0.05
     beta=0.99')
```

```
  # plt.plot(ii4, ff4, color='c', label='RMSProp,batch size = 5 alpha = 0.2 beta=0.99')
  # plt.plot(ii5, ff5, color='m', label='RMSProp,batch size = 5 alpha = 0.34 beta=0.99')
  # plt.plot(ii6, ff6, color='y', label='RMSProp,batch size = 5 alpha = 0.5 beta=0.99')

  # plt.plot(ii, ff, color='b', label='constant, step size = 0.3,batch size = 5')
  # plt.plot(ii1, ff1, color='r', label='RMSProp,batch size = 2 alpha = 0.34 beta=0.99')
  # plt.plot(ii2, ff2, color='g', label='RMSProp,batch size = 5 alpha = 0.34 beta=0.99')
  # plt.plot(ii3, ff3, color='c', label='RMSProp,batch size = 10 alpha = 0.34 beta=0.99')

  # plt.plot(xx1, yy1, color='b', label='RMSProp,batch size = 5 alpha = 0.01 beta=0.25')
  # plt.plot(xx2, yy2, color='r', label='RMSProp,batch size = 5 alpha = 0.05 beta=0.25')
  # plt.plot(xx3, yy3, color='g', label='RMSProp,batch size = 5 alpha = 0.05 beta=0.99')
  # plt.plot(xx4, yy4, color='c', label='RMSProp,batch size = 5 alpha = 0.2 beta=0.99')
  # plt.plot(xx5, yy5, color='m', label='RMSProp,batch size = 5 alpha = 0.34 beta=0.99')
  # plt.plot(xx6, yy6, color='y', label='RMSProp,batch size = 5 alpha = 0.5 beta=0.99')

  # plt.plot(xx, yy, color='b', label='constant, step size = 0.3,batch size = 5')
  # plt.plot(xx1, yy1, color='r', label='RMSProp,batch size = 2 alpha = 0.34 beta=0.99')
  # plt.plot(xx2, yy2, color='g', label='RMSProp,batch size = 5 alpha = 0.34 beta=0.99')
  # plt.plot(xx3, yy3, color='c', label='RMSProp,batch size = 10 alpha = 0.34 beta=0.99')

  # heavyBall
  # plt.plot(ii1, ff1, color='b', label='heavyBall,batch size = 5 alpha = 0.01 beta=0.25')
  # plt.plot(ii2, ff2, color='r', label='heavyBall,batch size = 5 alpha = 0.1 beta=0.25')
```

```
224.   # plt.plot(ii3, ff3, color='g', label='heavyBall,batch size = 5 alpha = 0.
    1 beta=0.5')
225.   # plt.plot(ii4, ff4, color='c', label='heavyBall,batch size = 5 alpha = 0.
    3 beta=0.5')
226.   # plt.plot(ii5, ff5, color='m', label='heavyBall,batch size = 5 alpha = 0.
    3 beta=0.9')
227.
228.   # plt.plot(xx1, yy1, color='b', label='heavyBall,batch size = 5 alpha = 0.
    01 beta=0.25')
229.   # plt.plot(xx2, yy2, color='r', label='heavyBall,batch size = 5 alpha = 0.
    1 beta=0.25')
230.   # plt.plot(xx3, yy3, color='g', label='heavyBall,batch size = 5 alpha = 0.
    1 beta=0.5')
231.   # plt.plot(xx4, yy4, color='c', label='heavyBall,batch size = 5 alpha = 0.
    3 beta=0.5')
232.   # plt.plot(xx5, yy5, color='m', label='heavyBall,batch size = 5 alpha = 0.
    3 beta=0.9')
233.
234.   # plt.plot(ii, ff, color='b', label='constant, step size = 0.3,batch size
    = 5')
235.   # plt.plot(ii1, ff1, color='r', label='heavyBall,batch size = 2 alpha = 0.
    1 beta=0.5')
236.   # plt.plot(ii2, ff2, color='g', label='heavyBall,batch size = 5 alpha = 0.
    1 beta=0.5')
237.   # plt.plot(ii3, ff3, color='c', label='heavyBall,batch size = 10 alpha = 0
    .1 beta=0.5')
238.
239.   # plt.plot(xx, yy, color='b', label='constant, step size = 0.3,batch size
    = 5')
240.   # plt.plot(xx1, yy1, color='r', label='heavyBall,batch size = 2 alpha = 0.
    1 beta=0.5')
241.   # plt.plot(xx2, yy2, color='g', label='heavyBall,batch size = 5 alpha = 0.
    1 beta=0.5')
242.   # plt.plot(xx3, yy3, color='c', label='heavyBall,batch size = 10 alpha = 0
    .1 beta=0.5')
243.
244.   # adam
245.   # plt.plot(ii0, ff0, color='black', label='adam,batch size = 5 alpha = 0.1
     beta1=0.25 beta2=0.25')
246.   # plt.plot(ii1, ff1, color='b', label='adam,batch size = 5 alpha = 0.5 bet
    a1=0.25 beta2=0.25')
247.   # plt.plot(ii2, ff2, color='r', label='adam,batch size = 5 alpha = 0.5 bet
    a1=0.9 beta2=0.25')
```

```
248.   # plt.plot(ii3, ff3, color='g', label='adam,batch size = 5 alpha = 0.5 bet
   a1=0.25 beta2=0.9')
249.   # plt.plot(ii4, ff4, color='c', label='adam,batch size = 5 alpha = 0.5 bet
   a1=0.9 beta2=0.9')
250.   # plt.plot(ii5, ff5, color='m', label='adam,batch size = 5 alpha = 2 beta1
   =0.9 beta2=0.9')
251.   # plt.plot(ii6, ff6, color='y', label='adam,batch size = 5 alpha = 5 beta1
   =0.9 beta2=0.9')
252.
253.   # plt.plot(ii, ff, color='b', label='constant, step size = 0.3,batch size
   = 5')
254.   # plt.plot(ii1, ff1, color='r', label='adam,batch size = 2 alpha = 2 beta1
   =0.9 beta2=0.9')
255.   # plt.plot(ii2, ff2, color='g', label='adam,batch size = 5 alpha = 2 beta1
   =0.9 beta2=0.9')
256.   # plt.plot(ii3, ff3, color='c', label='adam,batch size = 10 alpha = 2 beta
   1=0.9 beta2=0.9')
257.
258.   plt.plot(xx, yy, color='b', label='constant, step size = 0.3,batch size =
   5')
259.   plt.plot(xx1, yy1, color='r', label='adam,batch size = 2 alpha = 2 beta1=0
   .9 beta2=0.9')
260.   plt.plot(xx2, yy2, color='g', label='adam,batch size = 5 alpha = 2 beta1=0
   .9 beta2=0.9')
261.   plt.plot(xx3, yy3, color='c', label='adam,batch size = 10 alpha = 2 beta1=
   0.9 beta2=0.9')
262.
263.
264.   X, Y = np.meshgrid(x, y)
265.   Z = np.zeros((len(X), len(X[0])))
266.   for i, v in enumerate(X):
267.       for j, v2 in enumerate(v):
268.           Z[i][j] = f((X[i][j], Y[i][j]), data)
269.
270.   contours = plt.contour(X, Y, Z, 20);
271.   plt.clabel(contours, inline=True, fontsize=10)
272.   # plt.xlabel('epoches')
273.   # plt.ylabel('function value of log')
274.   plt.xlabel('x1')
275.   plt.ylabel('x2')
276.   # plt.yscale("log")
277.   # plt.xscale("log")
278.   # plt.ylim((-0.5, 0.1))
279.   plt.legend()
```

```
280.   plt.show()
```