

OPTIMIZATION IN BIG DATA RESEARCH  
GRADIENT DESCENT METHOD REPORT

---

Zixuan Li

student number 3118103163

professor: Jianyong Sun

December 12, 2018

# Contents

<b>1</b>	<b>Research Background</b>	<b>2</b>
1.1	research purpose . . . . .	2
1.2	compile environment . . . . .	4
1.3	procedure . . . . .	4
<b>2</b>	<b>Gradient Descent Method</b>	<b>5</b>
2.1	algorithm introduction . . . . .	5
2.2	result . . . . .	5
2.3	source code . . . . .	6
<b>3</b>	<b>Momentum Method</b>	<b>7</b>
3.1	algorithm introduction . . . . .	7
3.2	result . . . . .	7
3.3	source code . . . . .	7
<b>4</b>	<b>Nesterov accerlerated gradient Method</b>	<b>9</b>
4.1	algorithm introduction . . . . .	9
4.2	result . . . . .	9
4.3	source code . . . . .	9
<b>5</b>	<b>Adaptive Moment Estimation Method</b>	<b>11</b>
5.1	algorithm introduction . . . . .	11
5.2	result . . . . .	11
5.3	source code . . . . .	12
<b>6</b>	<b>Discussion &amp; Conclusion</b>	<b>13</b>
<b>7</b>	<b>Reference</b>	<b>14</b>
<b>8</b>	<b>Appendix</b>	<b>15</b>

# 1 Research Background

## 1.1 research purpose

The purpose of this experiment is to optimize Beale function by using gradient descent, momentum, Nesterov accelerated gradient(NAG), and Adaptive Moment Estimation (Adam) algorithms.

According to the definition by Khon nen, neural network is an extensive parallel interconnected network composed of simple units with adaptability, and its organization can simulate the interaction of biological nervous system with real world objects [ Koho nen, 1988 ]. In machine learning, the neural network means neural network leaning which is an interdisciplinary field of study.

The most basic component of the neural network is the neuron model. In the neural network, each neuron is connected to other neurons, and when it is stimulated, it will send signal to the other connected neurons. And the connection have different weight, thus has different impact on the other neurons. The total input value received by the neuron will be compared with the threshold value of the neuron and then processed by the activation function to produce the output of the neuron. Training neural network is to give a training data set, and continuously adjust the transmission weight and activation function in the neural network to reduce the gap between the predicted value and the data set. If the error of the neural network in the training set is expressed by  $\epsilon$  and it is obviously a function of the connection weight  $\omega$  and the threshold value  $\theta$  . At this time, the training process of the neural network can be regarded as a parameter optimization process. That is, in the parameter space, finding a set of optimal parameters makes  $\epsilon$  minimum.

Gradient descent is the most widely used parameter optimization method, in which we start from some initial solutions. In each iteration, we first calculate the gradient of the error function  $\epsilon(x)$  at the current point and then determine the search direction according to the gradient. The learning rate  $\eta$  determines the step of each iteration. Many deep learning project use various algorithms to optimize gradient descent, but in these large models, the optimization algorithm is often a black box model. Therefore, the purpose of this experiment is to optimize a test function by using several commonly used gradient descent algorithms, and to gain a more intuitive understanding of several gradient descent algorithms by writing code and visualizing the descent process.

The test function used in this research is Beale function:

$$f(x, y) = (1.5 - x + xy)^2 + (2.25 - x + xy^2)^2$$

Global minimum is  $f(3, 0.5) = 0$  in search domain  $-4.5 \leq x, y \leq 4.5$ .

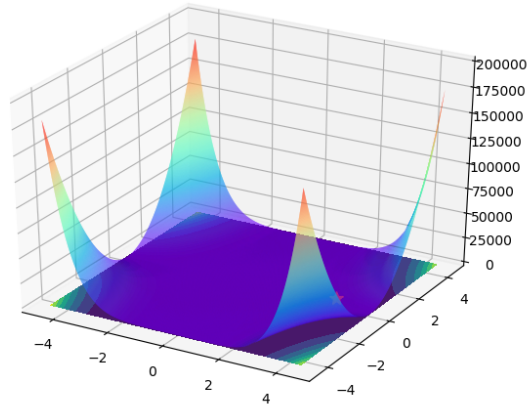


Figure 1: 3d plot of Beale function

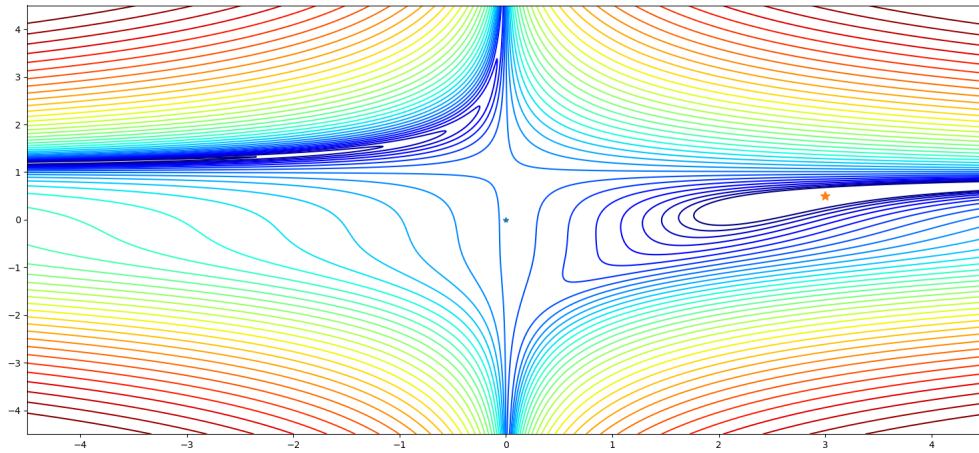


Figure 2: contour plot of Beale function and global minimum

In this paper, Gradient descent, Momentum, Nesterov accelerated gradient, and Adaptive Moment Estimation will be used to solve the global minimal point of Beale Function. The characteristics of various gradient descent methods are known through experiments. The report will consist of the following five main parts:

- Gradient descent method
- Momentum methods
- Nesterov accelerated gradient(NAG) method
- Adaptive Moment Estimation(ADAM) methon
- Conclusion and Discussion

## 1.2 compile environment

The Python language is used for experiments. The relevant compilation environment is as follows:

- OS Windows 10
- Compiler python3.6
- numpy scientific computing package
- matplotlib python visualize package

## 1.3 procedure

1. programming to visualize Beale function.
2. programming the four algorithm.
3. Adjust parameters in each algorithms, such as learning rate, to achieve fastest convergence rate.
4. Run the program, examine the route of convergence to gain insight into each algorithm.
5. Output related graphs and data
6. Finish the experiment report.

## 2 Gradient Descent Method

### 2.1 algorithm introduction

The gradient descent method is very easy to understand. The gradient direction indicates the direction where the function grows fastest, so the opposite direction is the direction where the function decreases fastest. For the problem of machine learning model optimization, when we need to solve the minimum value, we can find the optimal value by moving in the direction of gradient descent.

The main process of GD algorithm can be summarized as follows:

1. input: Function to be solved  $f(x, y)$ , derivative of function to be solved  $f(x, y)'$ , start point  $P(x_0, y_0)$ , the step size along the gradient descent direction  $step$
2. for  $(abs(grad) > 1e - 6)$  do
3. compute the derivative of current point  $P(x, y)$
4.  $gradient = -f'(P_t)'$
5. updated the next point as  $P_{t+1} = P_t - step * gradient$
6. end
7. output: the convergence point  $P_{convergence}$

### 2.2 result

The start point as the exercise requirement is set at  $P(3, 4)$ . As for the step size, If it is less than  $1/L$ . Then it can guarantee convergence ( $L$  is the Lipschitz constant of the gradient of the objective function).

$$|f(x_1) - f(x_2)| = |f(\xi)'(x_1 - x_2)| \leq L|x_1 - x_2|$$

I computed the numerical solution of Beale function's Lipschitz constant. And the result is  $L = 248533.18$ . so as long as  $dp \leq 4 \times 10^{-6}$  the algorithm will convergence. After weighing convergence and convergence speed, I finally choose  $step = 5 \times 10^{-6}$ .

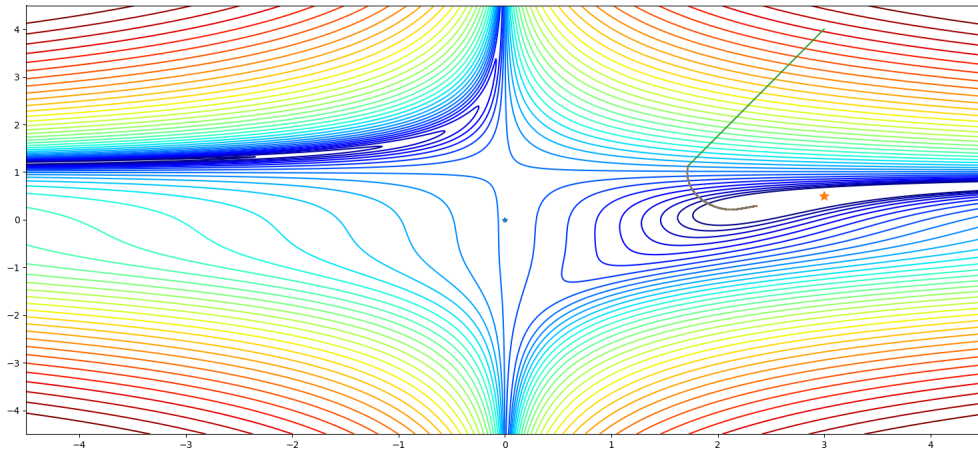


Figure 3: gradient descent process

## 2.3 source code

The part of source code of this algorithm is as follows.

```
1 def gd(x_start, step, g): # Gradient Descent
2     x = np.array(x_start, dtype='float64')
3     passing_dot = [x.copy()]
4     for i in range(n_iter):
5         grad = g(x)
6         x -= grad * step
7         passing_dot.append(x.copy())
8         if abs(sum(grad)) < 1e-6:
9             break;
10    return x, passing_dot
```

GD.py

## 3 Momentum Method

### 3.1 algorithm introduction

As its name suggests, momentum acts as a constant catalyst for the previous optimization in the process of optimization. The influence of an already declining direction will not disappear immediately, but will decay in a certain form.

The main process of Momentum algorithm can be summarized as follows:

1. input: Function to be solved  $f(x, y)$ , derivative of function to be solved  $f(x, y)'$ , start point  $P(x_0, y_0)$ , the step size along the gradient descent direction  $step$ , momentum term  $\gamma$
2. for  $(abs(grad) > 1e-6)$  do
3. compute the derivative of current point  $x$
4. compute update vector  $v_t = \gamma v_{t-1} + step f(x, y)'$
5. updated the next point as  $P_{t+1} = P_t - v_t$
6. end
7. output: the convergence point  $P_{convergence}$

The momentum term  $\gamma$  is usually set to 0.9.

### 3.2 result

The start point as the exercise requirement is set at  $P(3, 4)$ . As for the step size, After weighing convergence and convergence speed, I finally choose  $step = 5 \times 10^{-7}$ . To restrict the iteration time, the program is set to stop when it iteraes 10000 times.

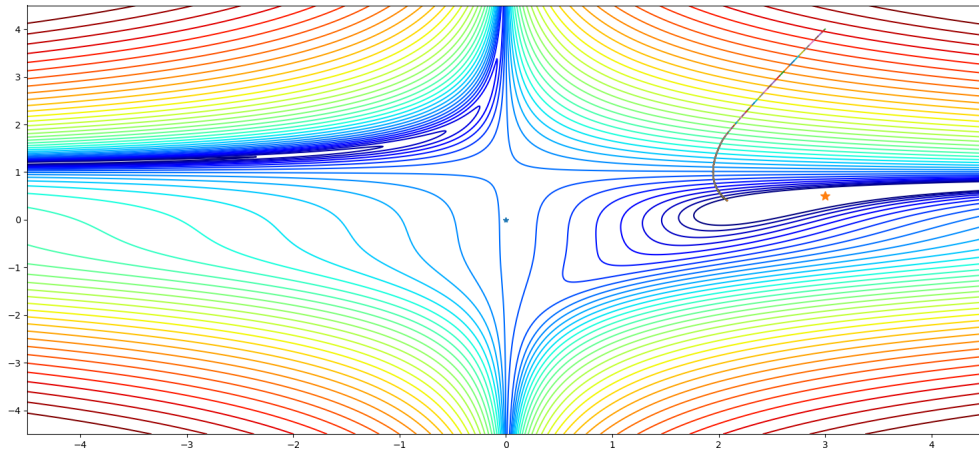


Figure 4: momentum descent process

### 3.3 source code

The part of source code of this algorithm is as follows.



```

1 def momentum(x_start, step, g, discount = 0.9):    # momentum
2     x = np.array(x_start, dtype='float64')
3     passing_dot = [x.copy()]
4     pre_grad = np.zeros_like(x)
5     for i in range(n_iter):
6         grad = g(x)
7         pre_grad = pre_grad * discount + grad
8         x -= pre_grad * step
9
10        passing_dot.append(x.copy())
11        if abs(sum(grad)) < 1e-6:
12            break;
13    return x, passing_dot

```

momentum.py

## 4 Nesterov accerlerated gradient Method

### 4.1 algorithm introduction

The momentum method can be thought as a ball rolling down a hill with momentum. Then the NAG method is smarter ball which can slowing down before climbing up another hill.

The main process of Momentum algorithm can be summarized as follows:

1. input: Function to be solved  $f(x, y)$ , derivative of function to be solved  $f(x, y)'$ , start point  $P(x_0, x_0)$ , the step size along the gradient descent direction  $step$ , momentum term  $\gamma$
2. for  $(abs(grad) > 1e-6)$  do
3. compute the derivative of current point  $f(P_t)'$
4. compute update vector  $v_t = \gamma v_{t-1} + step P_t - \gamma v_{t-1}'$
5. updated the next point as  $P_{t+1} = P_t - v_t$
6. end
7. output: the convergence point  $P_{convergence}$

The difference between Nag and Momentum is that, NAG use the momentum of present point to get a proximation of the next gradient rather than using the present gradient directly as Momentum method.

### 4.2 result

The start point as the exercise requirement is set at  $P(3, 4)$ . The momentum term  $\gamma$  is also usually set to 0.9. The step size is same as momentum method  $step = 5 \times 10^{-7}$ . To restrict the iteraion time, the program is set to stop when it ierates 10000 times.

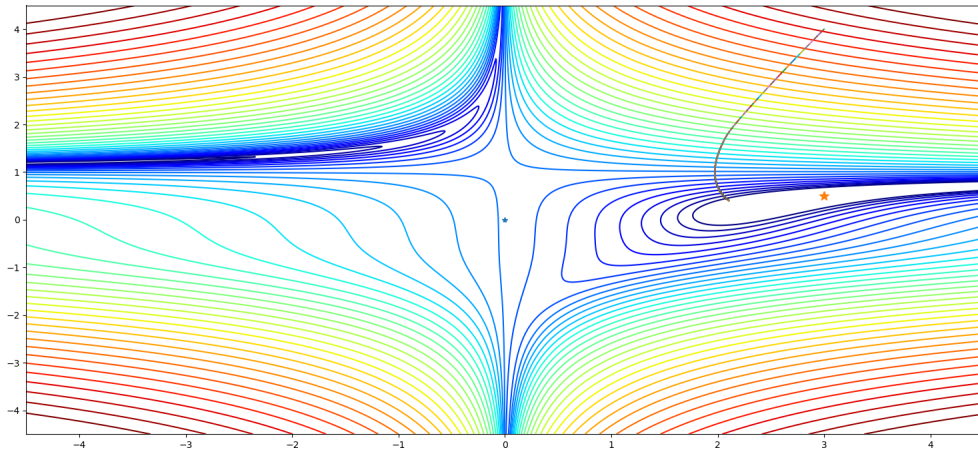


Figure 5: NAG descent process

### 4.3 source code

The part of source code of this algorithm is as follows.

```

1 def nesterov(x_start, step, g, discount=0.9):    #Nesterov accelerated gradient
2     x = np.array(x_start, dtype='float64')
3     passing_dot = [x.copy()]
4     pre_grad = np.zeros_like(x)
5     for i in range(n_iter):
6         x_future = x - step * discount * pre_grad
7         grad = g(x_future)
8         pre_grad = pre_grad * discount + grad
9         x -= pre_grad * step
10
11     passing_dot.append(x.copy())
12     if abs(sum(grad)) < 1e-6:
13         break;
14     return x, passing_dot

```

nag.py

## 5 Adaptive Moment Estimation Method

### 5.1 algorithm introduction

Adaptive Moment Estimation (Adam) is a method that computes adaptive learning rates for each parameter. Adaptive combines the advantages of RMSprop and Momentum method. It stores the an exponentially decaying average of past squared gradients  $v_t$  and exponentially decaying average of past gradients  $m_t$ .

The main process of Momentum algorithm can be summarized as follows:

1. input: Function to be solved  $f(x, y)$ , derivative of function to be solved  $f(x, y)'$ , start point  $P(3, 4)$ , the step size along the gradient descent direction  $step$ , super-parameter  $\beta_1, \beta_2$ .
2. for  $(abs(grad) > 1e - 6)$  do
3. compute the gradient of current point  $g_t = -f(P_t)'$
4. compute the decaying averages of past gradient  $m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$
5. compute the decaying averages of past squared gradient  $v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
6. computing bias-corrected first moment estimates  $\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$
7. computing bias-corrected second moment estimates  $\hat{v}_t = \frac{m_t}{1 - \beta_2^t}$
8. compute update vector  $v_t = \frac{step}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t$
9. updated the next point as  $P_{t+1} = P_t - v_t$
10. end
11. output: the convergence point  $P_{convergence}$

### 5.2 result

The default values proposed by the authors of this methd is  $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$  And The start point as the exercise requirement is set at  $P(3, 4)$ . The step size is set to 1 by testing to have a relatively fast convergence rate. To restrict the iteraion time, the program is set to stop when it ierates 10000 times.

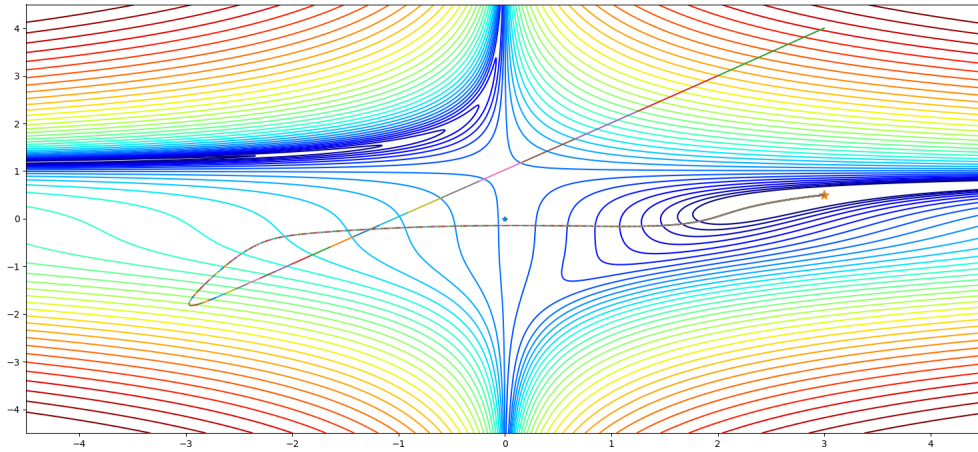


Figure 6: ADAM descent process

We can see that the Adam algorithm behaves like a heavy ball with friction because it stores an exponentially decaying average of past gradients.

### 5.3 source code

The part of source code of this algorithm is as follows.

```
1 def adam(x_start, step, g):                                     #Adaptive Moment Estimation
2     x = np.array(x_start, dtype='float64')
3     beta1=0.9
4     beta2=0.999
5     mt = np.zeros_like(x)
6     vt = np.zeros_like(x)
7     dp = np.zeros_like(x)
8     passing_dot = [x.copy()]
9     for i in range(n_iter):
10         beta1t=beta1**(i+1)
11         beta2t=beta2**(i+1)
12         temp=g(x)
13         mt=beta1*mt + (1-beta1) * g(x)
14         vt=beta2*vt + (1-beta2) * g(x)**2
15         mt1=mt/(1-beta1t)
16         vt1=vt/(1-beta2t)
17         dp[0]= step * mt1[0]/(math.sqrt(vt1[0])+0.00000001)
18         dp[1] = step * mt1[1] / (math.sqrt(vt1[1]) + 0.00000001)
19         x -= dp
20         passing_dot.append(x.copy())
21         if abs(sum(mt1)) < 1e-6:
22             break;
23     return x, passing_dot
```

adam.py

## 6 Discussion & Conclusion

The objective of the research is to implement four gradient descent method on Beale function to gain insight into the characteristics of each method. The performance of each method can be evaluated by the final convergence point less than 10000 times iteration. And the result is summarized as the following table.

method	x	y	distance to global nominal
GD	2.36625941	0.29157283	0.667134934
Momentum	2.07978324	0.40197178	0.925423372
NAG	2.10316029	0.40410201	0.901952266
ADAM	2.99999789	0.49999947	2.17555E-06

Table 1: Final Convergence Point Table

We can see that the Adam method outperformed the rest methods. And in practice, Adam is also a very good choice with fairly fast speed. To avoid stuck in saddle points, optimal gradient descent method not only can solve this problem, but also have faster convergence rate.

During the experiment, There were two main difficulties I encountered. The first is about tuning the parameter of each algorithm, I used excel to record each parameter and corresponding convergence rate, and finally get a fairly fast parameter setting. The second difficulty is about visualizing the descent process. Actually the plotting process takes much more time than the solution time. Because the program have to plot each passing dot which is a time consuming process. So I limited the total iteration number. And by doing this, I can plot quicker and get an visualize compare result of convergence speed between each method since the iteration time is the same.

In summary, a first hand experience on how to implement gradient descent method using python has gained through the experiment. And I have gained more intuitive understanding of how each method works and what difficulties each method intends to tackle.

## 7 Reference

1. Sebastian Ruder (2016). An overview of gradient descent optimisation algorithms.arXiv preprint arXiv:1609.04747.
2. Zhou Zhihua (2016). Machine Learning. ISBN 978:7-302-42328-7

## 8 Appendix

Full version of source code:

```
1 # -*- coding: utf-8 -*-
2
3 import numpy as np
4 import math
5 import matplotlib.pyplot as plt
6 from matplotlib.colors import LogNorm
7 from mpl_toolkits.mplot3d import Axes3D
8
9 n_iter = 10000
10 def f(x):
11     x1 = x[0]
12     y1 = x[1]
13
14     term1 = (1.5 - x1 + x1*y1)**2
15     term2 = (2.25 - x1 + x1*y1**2)**2
16     term3 = (2.625 - x1 + x1*y1**3)**2
17     z = term1 + term2 + term3;
18     return z
19
20 def g(x):
21     x1 = x[0]
22     y1 = x[1]
23     g1 = 2*(1.5 - x1 + x1*y1)*(y1-1)+2*(2.25-x1+x1*y1**2)*(y1**2-1)+2*(2.625-x1+x1*y1**3)*(y1
24         **3-1)
25     g2 = 2*(1.5 - x1 + x1*y1)*x1 + 2*(2.25-x1+x1*y1**2)*(2*y1*x1)+2*(2.625-x1+x1*y1**3)*(3*y1**2*
26         x1)
27     return np.array([g1, g2])
28
29 #matplotlib inline
30 def contour(X,Y,Z, arr = None):
31     plt.figure(figsize=(15,7))
32     plt.contour(X, Y, Z.T, levels=np.logspace(0, 5, 35), norm=LogNorm(), cmap=plt.cm.jet)
33     plt.plot(0,0, marker='*')
34     plt.plot(3, 0.5, marker='*', markersize=10)
35     if arr is not None:
36         arr = np.array(arr)
37         for i in range(len(arr) - 1):
38             plt.plot(arr[i:i+2,0], arr[i:i+2,1])
39
40 #contour(X,Y,Z)
41
42 def plot3d(X,Y,Z):
43     fig = plt.figure()
44     # 3 d
45     ax = Axes3D(fig)
46     ax = fig.add_subplot(111, projection='3d')
47     # rstride : cstride :
48     # rcount :50ccount :
49     #vmaxvmin
50     ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=plt.get_cmap('rainbow'))
51     xminima=np.array([3.])
52     yminima = np.array([.5])
53     zminima=np.array([0.0])
54     ax.plot(xminima, yminima, zminima, '*', markersize=10)
55     # zdir : 'z' | 'x' | 'y'
56     # offset :
57     ax.contourf(X,Y,Z, zdir='z', offset=-2)
58     # zxy
59     ax.set_zlim(-2,200000)
```



```

60 def gd(x_start, step, g): # Gradient Descent
61     x = np.array(x_start, dtype='float64')
62     passing_dot = [x.copy()]
63     for i in range(n_iter):
64         grad = g(x)
65         x -= grad * step
66         passing_dot.append(x.copy())
67         if abs(sum(grad)) < 1e-6:
68             break;
69     return x, passing_dot
70
71 def momentum(x_start, step, g, discount = 0.9): # momentum
72     x = np.array(x_start, dtype='float64')
73     passing_dot = [x.copy()]
74     pre_grad = np.zeros_like(x)
75     for i in range(n_iter):
76         grad = g(x)
77         pre_grad = pre_grad * discount + grad
78         x -= pre_grad * step
79
80         passing_dot.append(x.copy())
81         if abs(sum(grad)) < 1e-6:
82             break;
83     return x, passing_dot
84
85
86 def nesterov(x_start, step, g, discount=0.9): #Nesterov accelerated gradient
87     x = np.array(x_start, dtype='float64')
88     passing_dot = [x.copy()]
89     pre_grad = np.zeros_like(x)
90     for i in range(n_iter):
91         x_future = x - step * discount * pre_grad
92         grad = g(x_future)
93         pre_grad = pre_grad * discount + grad
94         x -= pre_grad * step
95
96         passing_dot.append(x.copy())
97         if abs(sum(grad)) < 1e-6:
98             break;
99     return x, passing_dot
100
101 def adam(x_start, step, g): #Adaptive Moment Estimation
102     x = np.array(x_start, dtype='float64')
103     beta1=0.9
104     beta2=0.999
105     mt = np.zeros_like(x)
106     vt = np.zeros_like(x)
107     dp = np.zeros_like(x)
108     passing_dot = [x.copy()]
109     for i in range(n_iter):
110         beta1t=beta1**(i+1)
111         beta2t=beta2**(i+1)
112         temp=g(x)
113         mt=beta1*mt + (1-beta1) * g(x)
114         vt=beta2*vt + (1-beta2) * g(x)**2
115         mt1=mt/(1-beta1t)
116         vt1=vt/(1-beta2t)
117         dp[0]= step * mt1[0]/(math.sqrt(vt1[0])+0.00000001)
118         dp[1]= step * mt1[1] / (math.sqrt(vt1[1]) + 0.00000001)
119         x -= dp
120         passing_dot.append(x.copy())
121         if abs(sum(mt1)) < 1e-6:
122             break;
123     return x, passing_dot
124

```

```

125
126
127 xi = np.linspace(-4.5,4.5,1000)
128 yi = np.linspace(-4.5,4.5,1000)
129 X,Y = np.meshgrid(xi, yi)
130 Z = np.empty([len(xi),len(yi)], dtype = float)
131 for i in range(len(xi)):
132     for j in range(len(yi)):
133         x =(xi[i],yi[j])
134         Z[i,j]=f(x)
135
136
137
138 plot3d(X,Y,Z)
139
140 contour(X,Y,Z)
141 res, x_arr = gd([3,4], 0.00005, g)
142 #contour(X,Y,Z, x_arr)
143 print ('a')
144 print (res)
145 res, x_arr = momentum([3,4], 0.0000005, g)
146 #contour(X,Y,Z, x_arr)
147 print (res)
148 res, x_arr = nesterov([3,4], 0.0000005, g)
149 #contour(X,Y,Z, x_arr)
150 print (res)
151 res, x_arr = adam([3,4], 1, g)
152 print (res)
153 #contour(X,Y,Z, x_arr)
154 print ('a')
155 print ('a')
156 plt.show()

```

gradient.py

## List of Figures

1	3d plot of Beale function . . . . .	3
2	contour plot of Beale function and global minimum . . . . .	3
3	gradient descent process . . . . .	5
4	momentum descent process . . . . .	7
5	NAG descent process . . . . .	9
6	ADAM descent process . . . . .	11

List of Tables

1    Final Convergence Point Table . . . . . 13