OPTIMIZATION IN BIG DATA RESEARCH

GRADIENT DESCENT METHOD REPORT

Zixuan Li

student number 3118103163

professor: Jianyong Sun

December 13, 2018

Contents

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 interdisciplinanry 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 ϵ and it is obviously a function of the connection weight ω and the threshold value θ . 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 ϵ 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 η 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:

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

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

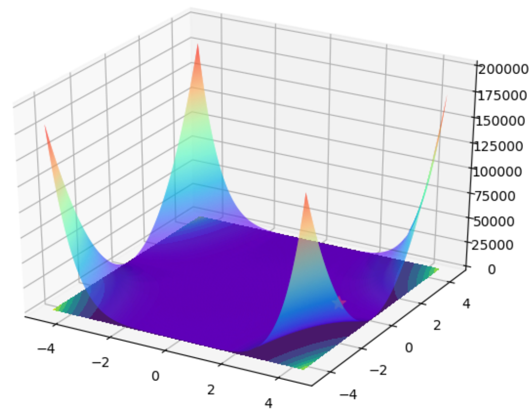

Figure 1: 3d plot of Beale function

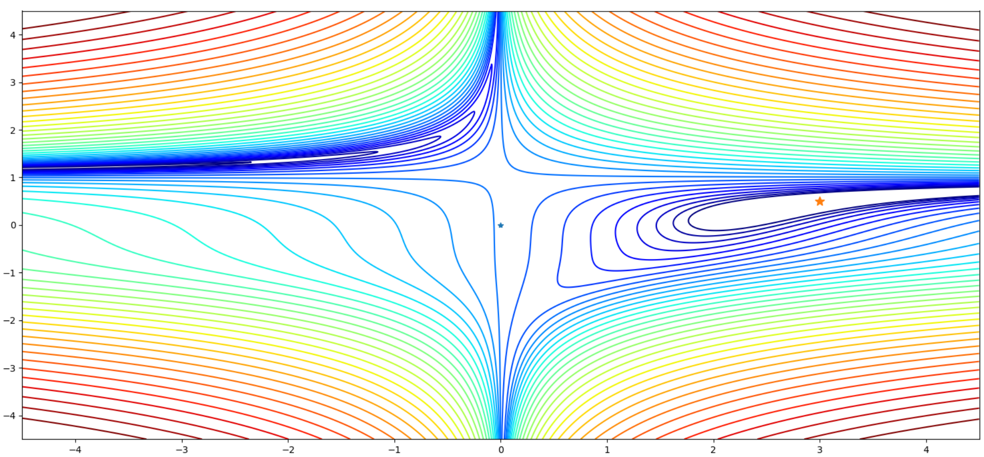

Figure 2: contour plot of Beale function and global minimum

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

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

- OS Windows 10
- Complier 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 funcition 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). The definition of Lipschitz can be expressed as followed.

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

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. One way to adaptive choose the step size is to use backtracking line search. However, since this problem is faily easy to compute, I decided to use fixed step size. After weighing convergence and convergence speed, I finally choose $step = 5 \times 10^{-6}$. To restrict the iteraion time, the program is set to stop when it iteraes 10000 times.

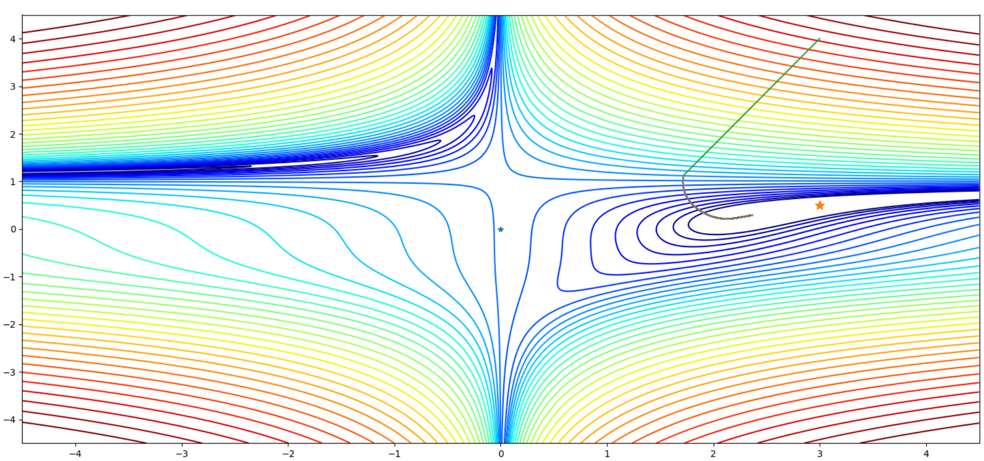

Figure 3: gradient descent process

2.3 source code

The source code of this algorithm is as follows.

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

GD.py

3 Momentum Method

3.1 algorithm introduction

As its name suggests, momentum acts like 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 funcition to be solved$f(x, y)'$, start point$P(x_0, y_0)$, the step size along the gradient descent direction $step$, momentum term γ
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 $P_{convergence}$

3.2 result

The start point as the exercise requirement is set at $P(3, 4)$. The momentum term γ is usually set to 0.9. As for the step size,After weighing convergence and convergence speed, I finally choose $step = 5 \times 10^{-7}$. To restrict the iteraion time, the program is set to stop when it iteraes 10000 times.

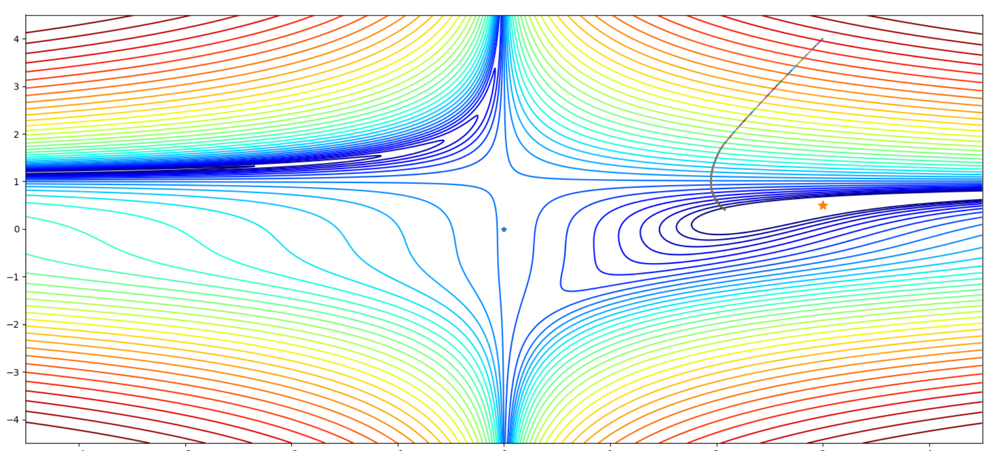

Figure 4: momentum descent process

3.3 source code

The source code of this algorithm is as follows.

```python
def momentum(x_start, step, g, discount = 0.9):   # momentum
    x = np.array(x_start, dtype='float64')
    passing_dot = [x.copy()]
    pre_grad = np.zeros_like(x)
    for i in range(n_iter):
        grad = g(x)
        pre_grad = pre_grad * discount + grad
        x -= pre_grad * step

        passing_dot.append(x.copy())
        if abs(sum(grad)) < 1e-6:
            break;
    return x, passing_dot
```

momentum.py

4 Nesterov accerlerated gradient Method

4.1 algorithm introduction

If the momentum method can be thought as a ball rolling down a hill with momentum. Then the NAG method is a 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 funcition to be solved$f(x,y)'$, start point$P(x_0,x_0)$, the step size along the gradient descent direction $step$, momentum term γ
2. for $(abs(grad) > 1e-6)$ do
3. compute the derivative of current point $f(P_t)'$
4. compute update vector $v_t = \gamma v_{t-1} + stepP_t - \gamma v_t - 1'$
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 γ 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.

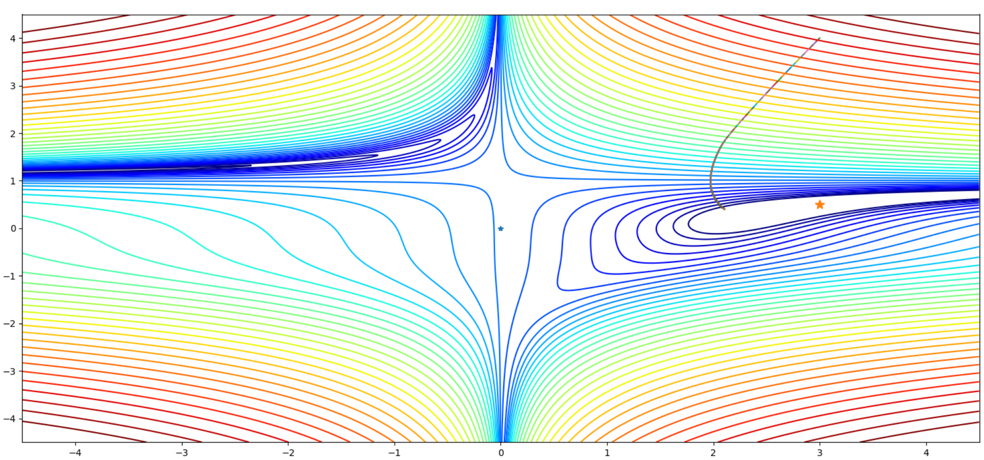

Figure 5: NAG descent process

4.3 source code

The source code of this algorithm is as follows.

```python
def nesterov(x_start, step, g, discount=0.9):   #Nesterov accelerated gradient
    x = np.array(x_start, dtype='float64')
    passing_dot = [x.copy()]
    pre_grad = np.zeros_like(x)
    for i in range(n_iter):
        x_future = x - step * discount * pre_grad
        grad = g(x_future)
        pre_grad = pre_grad * discount + grad
        x -= pre_grad * step

        passing_dot.append(x.copy())
        if abs(sum(grad)) < 1e-6:
            break;
    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 funcition to be solved$f(x, y)'$, start point$P(3, 4)$, the step size along the gradient descent direction $step$, super-parameterβ_1, β_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.

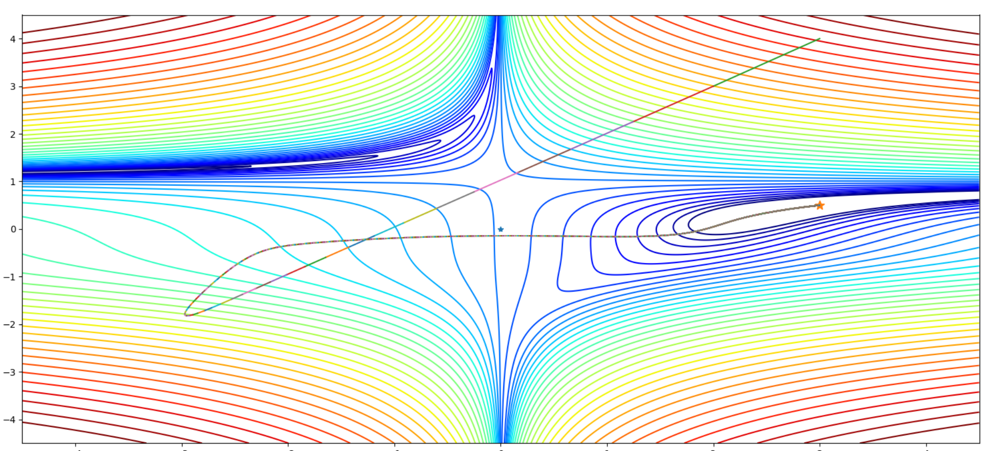

Figure 6: ADAM descent process

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.

```
def adam(x_start, step, g):                #Adaptive Moment Estimation
    x = np.array(x_start, dtype='float64')
    beta1=0.9
    beta2=0.999
    mt = np.zeros_like(x)
    vt = np.zeros_like(x)
    dp = np.zeros_like(x)
    passing_dot = [x.copy()]
    for i in range(n_iter):
        beta1t=beta1**(i+1)
        beta2t=beta2**(i+1)
        temp=g(x)
        mt=beta1*mt + (1-beta1) * g(x)
        vt=beta2*vt + (1-beta2) * g(x)**2
        mt1=mt/(1-beta1t)
        vt1=vt/(1-beta2t)
        dp[0]= step * mt1[0]/(math.sqrt(vt1[0])+0.00000001)
        dp[1] = step * mt1[1] / (math.sqrt(vt1[1]) + 0.00000001)
        x -= dp
        passing_dot.append(x.copy())
        if abs(sum(mt1)) < 1e-6:
            break;
    return x, passing_dot
```

adam.py

6 Discussion & Conclusion

The objective of this 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 evaluted by the final convergence point's distance to global nominal point. 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, opitmal grident descent method not only can solve this problem, but also have faster convergence rate.

During the experiment, There were mainly two difficulties I encountered. The first is about tuning the parameter of each algrithm, I used excel to record each parameter and corresbonding convergence rate, and finally get a fairly good parameter setting. The second difficulty is about visualizing the descent process. Actually the ploting process takes much more time than the solution time. Because the program have to plot each passing dot using a large array which is a time consuming process. So I limited the total iteration number. And by doing this, I can plot quicker and get an visualized result of convergence speed of each method since the iteraion time is the same.

In summary, a first hand experience on how to implement gradient discent 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 souce code:

```
# -*- coding: utf-8 -*-

import numpy as np
import math
import matplotlib.pyplot as plt
from matplotlib.colors import LogNorm
from mpl_toolkits.mplot3d import Axes3D

n_iter = 10000
def f(x):
    x1 = x[0]
    y1 = x[1]

    term1 = (1.5 - x1 + x1*y1)**2
    term2 = (2.25 - x1 + x1*y1**2)**2
    term3 = (2.625 - x1 + x1*y1**3)**2
    z = term1 + term2 + term3;
    return z

def g(x):
    x1 = x[0]
    y1 = x[1]
    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**3-1)
    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*x1)
    return np.array([g1, g2])

#matplotlib inline
def contour(X,Y,Z, arr = None):
    plt.figure(figsize=(15,7))
    plt.contour(X, Y, Z.T, levels=np.logspace(0, 5, 35),norm=LogNorm(),cmap=plt.cm.jet)
    plt.plot(0,0,marker='*')
    plt.plot(3, 0.5, marker='*',markersize=10)
    if arr is not None:
        arr = np.array(arr)
        for i in range(len(arr) - 1):
            plt.plot(arr[i:i+2,0],arr[i:i+2,1])


#contour(X,Y,Z)

def plot3d(X,Y,Z):
    fig = plt.figure()
    #  3    d
    # ax = Axes3D(fig)
    ax = fig.add_subplot(111, projection='3d')
    # rstride    :  cstride    :
    # rcount      :50ccount    :
    #vmaxvmin
    ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=plt.get_cmap('rainbow'))
    xminima=np.array([3.])
    yminima = np.array([.5])
    zminima=np.array([0.0])
    ax.plot(xminima,yminima,zminima, '*',markersize=10)
    # zdir : 'z' | 'x' | 'y'
    # offset :
    ax.contourf(X,Y,Z,zdir='z',offset=-2)
    #           zxy
    ax.set_zlim(-2,200000)

```

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

def momentum(x_start, step, g, discount = 0.9):   # momentum
    x = np.array(x_start, dtype='float64')
    passing_dot = [x.copy()]
    pre_grad = np.zeros_like(x)
    for i in range(n_iter):
        grad = g(x)
        pre_grad = pre_grad * discount + grad
        x -= pre_grad * step

        passing_dot.append(x.copy())
        if abs(sum(grad)) < 1e-6:
            break;
    return x, passing_dot


def nesterov(x_start, step, g, discount=0.9):   #Nesterov accelerated gradient
    x = np.array(x_start, dtype='float64')
    passing_dot = [x.copy()]
    pre_grad = np.zeros_like(x)
    for i in range(n_iter):
        x_future = x - step * discount * pre_grad
        grad = g(x_future)
        pre_grad = pre_grad * discount + grad
        x -= pre_grad * step

        passing_dot.append(x.copy())
        if abs(sum(grad)) < 1e-6:
            break;
    return x, passing_dot

def adam(x_start, step, g):             #Adaptive Moment Estimation
    x = np.array(x_start, dtype='float64')
    beta1=0.9
    beta2=0.999
    mt = np.zeros_like(x)
    vt = np.zeros_like(x)
    dp = np.zeros_like(x)
    passing_dot = [x.copy()]
    for i in range(n_iter):
        beta1t=beta1**(i+1)
        beta2t=beta2**(i+1)
        temp=g(x)
        mt=beta1*mt + (1-beta1) * g(x)
        vt=beta2*vt + (1-beta2) * g(x)**2
        mt1=mt/(1-beta1t)
        vt1=vt/(1-beta2t)
        dp[0]= step * mt1[0]/(math.sqrt(vt1[0])+0.00000001)
        dp[1] = step * mt1[1] / (math.sqrt(vt1[1]) + 0.00000001)
        x -= dp
        passing_dot.append(x.copy())
        if abs(sum(mt1)) < 1e-6:
            break;
    return x, passing_dot
```

```python


xi = np.linspace(-4.5,4.5,1000)
yi = np.linspace(-4.5,4.5,1000)
X,Y = np.meshgrid(xi, yi)
Z = np.empty([len(xi),len(yi)], dtype = float)
for i in range(len(xi)):
    for j in range(len(yi)):
        x =(xi[i],yi[j])
        Z[i,j]=f(x)



plot3d(X,Y,Z)

contour(X,Y,Z)
res, x_arr = gd([3,4], 0.00005, g)
#contour(X,Y,Z, x_arr)
print ('a')
print (res)
res, x_arr = momentum([3,4], 0.0000005, g)
#contour(X,Y,Z, x_arr)
print (res)
res, x_arr = nesterov([3,4], 0.0000005, g)
#contour(X,Y,Z, x_arr)
print (res)
res, x_arr = adam([3,4], 1, g)
print (res)
#contour(X,Y,Z, x_arr)
print ('a')
print ('a')
plt.show()
```

gradient.py

List of Figures

List of Tables