Anatomy of Deep Learning Principles

Coding a deep learning library from scratch

hwdong

hwdong-net.github.io

Brief introduction

This book introduces the basic principles and implementation process of deep learning in a simple way, and uses python's numpy library to build its own deep learning library from scratch instead of using existing deep learning libraries. On the basis of introducing basic knowledge of Python programming, calculus, and probability statistics, the core basic knowledge of deep learning such as regression model, neural network, convolutional neural network, recurrent neural network, and generative network is introduced in sequence according to the development of deep learning. While analyzing the principle in a simple way, it provides a detailed code implementation process. It is like not teaching you how to use weapons and mobile phones, but teaching you how to make weapons and mobile phones by yourself. This book is not a tutorial on the use of existing deep learning libraries, but an analysis of how to develop deep learning libraries from 0. This method of combining the principle from 0 with code implementation can enable readers to better understand the basic principles of deep learning and the design ideas of popular deep learning libraries.

Preface

Since the invention of computers, it has been the goal of computer scientists to make machines have human-like intelligence. Since the concept of "artificial intelligence" was proposed in 1956, artificial intelligence research has experienced many ups and downs from peak to trough, trough to peak In the development process of AI, from rule-based reasoning based on mathematical logic to state-space search reasoning, from expert systems to statistical learning, from crowd intelligence algorithms to machine learning, from neural networks to support vector machines, different artificial intelligence technologies used to lead the way.

In the past 6 years, deep learning using deep neural networks has been brilliant and advanced by leaps and bounds. Successful applications of deep learning such as AlphaGo defeating the human Go champion, automatic driving, machine translation, speech recognition, and deep face changing continue to attract people's attention. As a branch of machine learning, deep learning brings traditional neural network technology back to life, and has established itself as the overlord of modern artificial intelligence among all artificial intelligence technologies. status.

Deep learning has no complex and esoteric theories. In principle, it is still a traditional neural network, that is, some simple neuron functions are combined into a complex function and a simple gradient descent method is used to learn the model in the neural network based on actual sample data. parameter. Its success is mainly attributed to computer hardware, especially graphics processors GPUs with increasingly powerful parallel computing performance and more and more big data.

The future society will be a society of artificial intelligence. Artificial intelligence will be everywhere. Many jobs will be replaced by artificial intelligence. course.

With the help of some deep learning platforms such as tensorflow, pytorch, and caffe, a primary school student can easily use the deep learning library to do various applications such as face recognition and speech recognition. What he does is to directly call the APIs of these platforms to define the model of the deep neural network. Structure and tune training parameters. These platforms make deep learning very easy, make deep learning enter the homes of ordinary people, and artificial intelligence is no longer mysterious. From universities to enterprises, people from all walks of life are using deep learning to carry out various research and applications.

Like any technology, only by thoroughly understanding the principles behind the technology can the technology be better applied. There are a large number of scattered articles on the Internet explaining the principles of deep learning, and there are also some deep learning courses and tutorials. Books are still an important way to learn systematically. The written deep learning books are mainly divided into several categories: one is books that focus on mathematical theory for experts or professional researchers. These books, like academic papers, are difficult for readers to understand. These books lack in-depth analysis of the principles and code

implementation, and readers may still not know how to implement them even if they work hard to understand the principles. The other category is tool books, which mainly introduce how to use various deep learning platforms, with very little explanation of the principles, making readers unable to understand the principles behind the code, and can only follow the gourd. There are also some books that are just popular books, and they have a taste of every technology, and the principles and codes are often superficial. There are also very few books that introduce the principles and also have code implementations, and avoid the derivation of mathematical formulas as much as possible.

The author believes that platform tutorial books are time-sensitive, and the publication cycle of the book is usually as long as one year, and the interface of the platform may have undergone some changes or even major changes. For the changing platform, such books are almost worthless . Principle books should be easy to understand, try to avoid complex and esoteric mathematics, but completely abandon the classic advanced mathematics developed by mathematicians for thousands of years, and using elementary school mathematics to describe functions for derivation is not suitable for readers with advanced mathematics knowledge. Not an optimal choice. However, there is a special lack of easy-to-understand books on the market that introduce the principles and how to implement deep learning from the bottom instead of using deep learning libraries.

In order to take care of readers who are difficult in mathematics, the first chapter of this book not only introduces the necessary knowledge of python programming, but also introduces some necessary knowledge of calculus and probability as popularly as possible. On this basis, this book transitions from the simplest regression model to the neural network model from the shallower to the deeper, and uses the method from problem to concept to explain the basic concepts and principles in an easy-to-understand manner. Avoiding long speeches and avoiding "treasure words like gold", use simple examples and concise and popular language to analyze the core principles of models and algorithms. On the basis of understanding the principle, further use python's numpy library to implement the code from the bottom layer, so that readers can be enlightened on the principle and implementation. Through reading this book, readers can follow step by step to build a deep learning library from 0 without any deep learning platform. Finally, as a comparison, the use of the deep learning platform Pytorch is introduced, so that readers can easily learn to use this deep learning platform, which will help readers understand the design ideas of these platforms more deeply, so as to better grasp and use these deep learning platforms. Learning platform.

This book is suitable not only for beginners without any deep learning knowledge, but also for practitioners who have experience in using deep learning libraries and want to understand its underlying implementation principles. This book is especially suitable as a deep learning textbook for colleges and universities.

Relevant resources of the book (including algorithm codes) can be found on the author's website https://hwdong-net.github.io.

The English version of this book is translated using Google Translate on the basis of the Chinese version. We will continue to improve the quality of the translation in the future, and we hope readers can help me correct errors.

My email: hwdong.cn@gmail.com

eBook link (English Version): https://leanpub.com/dle/

eBook link (ChineseVersion): https://leanpub.com/dl_0

eBook link (Japanese): https://leanpub.com/dl_jp

eBook link (Traditional Chinese): https://leanpub.com/dl_tw

Chapter 2 Gradient descent method

The core task of deep learning is to train a function model through sample data, or to find an optimal function to represent or describe these sample data. Solving the best function model comes down to a mathematical optimization problem, more precisely, the problem of finding the extreme value of a certain loss function. In deep learning, the **gradient descent method** is used to solve this extreme value problem or solve the model parameters.

This chapter introduces the theoretical basis, algorithm principle and code implementation of the gradient descent algorithm starting from the necessary conditions for the extreme value of the function, and introduces different optimization strategies for updating the solution variables (parameters) in the gradient descent method.

2.1 Necessary conditions for function extremum

The function $y=f(x)$ obtains the **Minimal value** at a certain point x_0: it means that there is a certain positive number ϵ, so that for the interval $(x_0-\epsilon, x_0+\epsilon)$ each x satisfies $f(x_0) \leq f(x)$. This x_0 is called the **Minimal value point** of the function, and $f(x_0)$ is called the **Minimal value** of the function.

The function $y=f(x)$ obtains the **maximal value** at a certain point x_0: it means that there is a certain positive number ϵ, so that for the interval $(x_0-\epsilon, x_0+\epsilon)$ each x satisfies $f(x) \leq f(x_0)$. x_0 is called the **maximal value point** of the function, and $f(x_0)$ is called the **maximal value** of the function.

The minimum value and maximum value are collectively referred to as **extreme value**, and the minimum value point and maximum value point are collectively referred to as **extreme value point**.

If all x in the domain of the function $f(x)$ satisfy $f(x_0) \leq f(x)$, then x_0 is called the **minimum point** of the function, and $f(x_0)$ is called the **minimum value** of the function.

If all x in the domain of the function $f(x)$ satisfy $f(x) \leq f(x_0)$, then x_0 is called the **maximum point** of the function, and $f(x_0)$ is called the **maximum value** of the function.

That is, the minimum value is a minimum value of a global range, and the maximum value is a maximum value of a global range. The minimum and maximum values are collectively referred to as **Most Value**, and the minimum and maximum points are collectively referred to as **Most Value Points**.

Necessary conditions for function extremum: If x_0 is the extreme point of the function $f(x)$, and the function is derivable at x_0, then there must be $f'(x_0)=0$, that is, the derivative value at the extreme point must be 0 .

For example, the previous function $f(x)=x^2$ obtains the minimum value at $x=0$ (of course it is also a minimum value) and can be derived, so at $x=0$ its derivative value $f'(0)=2\times 0=0$ must be 0.

This proposition is easy to prove. If x_0 is the extreme point of the function $f(x)$, there is an interval $(x_0-\epsilon, x_0+\epsilon)$ that satisfies $f(x_0) \leq f(x)$, so $f(x)-f(x_0) \geq 0$, while:

$$f'(x_0)=\lim_{\Delta x\to 0}\frac{\Delta y}{\Delta x}=\lim_{\Delta x\to 0}\frac{f(x_0+\Delta x)-f(x_0)}{\Delta x}=\lim_{\Delta x\to x_0}\frac{f(x)-f(x_0)}{x-x_0}$$

When x tends to x_0 from the left and right sides, Δx is negative and positive respectively, and the numerator is always positive.
When x tends to x_0 from the right, its limit value should be ≥ 0, When x tends to x_0 from the left, its limit value should be ≤ 0, and this limit exists, so its value can only be 0.

According to the limit formula, a rule can also be found: if the derivative at x_0 is a positive number, it means that the function f(x) is monotonically increasing around this point, that is, if $x_1 < x_2$, then $f(x_1) < f(x_2)$, that is, $f(x)$ increases as x increases. Or if Δx is a positive number, then Δy is also a positive number. For example, the derivative of $y = f(x) = x^2$ is $f'(x) = 2x$, when x is greater than 0, the derivative is positive, therefore, the function curve is monotonically increasing. Similarly, if the derivative at x_0 is positive, it means that the function f(x) is monotonically decreasing around this point, that is, if $x_1 < x_2$, then $f(x_1) > f(x_2)$, That is, $f(x)$ becomes smaller as x increases. For example, when x is less than 0, the derivative of $y = f(x) = x^2$ is negative, so the function curve is monotonically decreasing, that is, if $x_1 < x_2$, instead $f(x_1) > f(x_2)$.

For example, the function $f\,(x) = x^3–3x^2–9x + 2.$, let its derivative $f'\,(x) = 0$:

$$f'\,(x) = 0, \quad \Rightarrow \left(x^3–3x^2–9x + 2\right) = 0, \quad \Rightarrow 3x^2–6x–9 = 0, \quad \Rightarrow x^2–2x–3 = 0, \quad \Rightarrow x_1 = –1,\ x_2 = 3$$

Two points $x_1 = –1,\ x_2 = 3$ with a derivative of 0 can be obtained. The change of this function and its derivative function $f'(x)$ is shown in Figure 2-1:

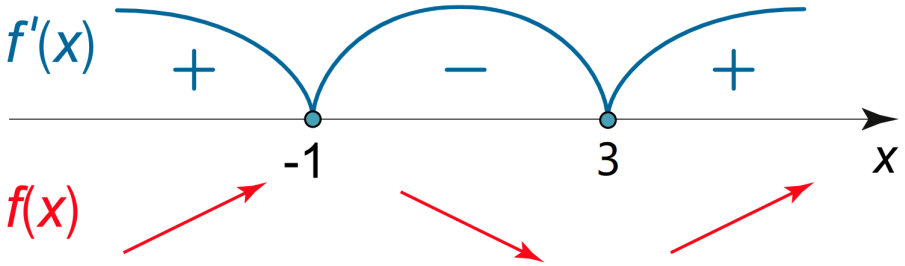

Figure 2-1 $f'(x) > 0$, the function increases monotonically, $f'(x) < 0$, the function decreases monotonically

In the interval $(-\infty, -1]$, $f'(x)$ is a positive number, so the function f(x) is monotonically increasing, and in the interval $(-1, 3)$, $f'(x)$ is a negative number, so the function f(x) is monotonically decreasing. In the interval $[3, \infty)$, $f'(x)$ is a positive number, so the function f(x) is monotonically increasing .

The following code draws the curve of this function and its derivative function, which can more intuitively reflect the monotonous change and extreme point of the function.

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

x = np.arange(-3, 4, 0.01)
f_x = np.power(x,3)-3*x**2-9*x+2
df_x = 3*x**2-6*x-9

plt.plot(x,f_x)
plt.plot(x,df_x)
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.legend(['f(x)', "df(x)"])
plt.axvline(x=0, color='k')
plt.axhline(y=0, color='k')
plt.show()
```

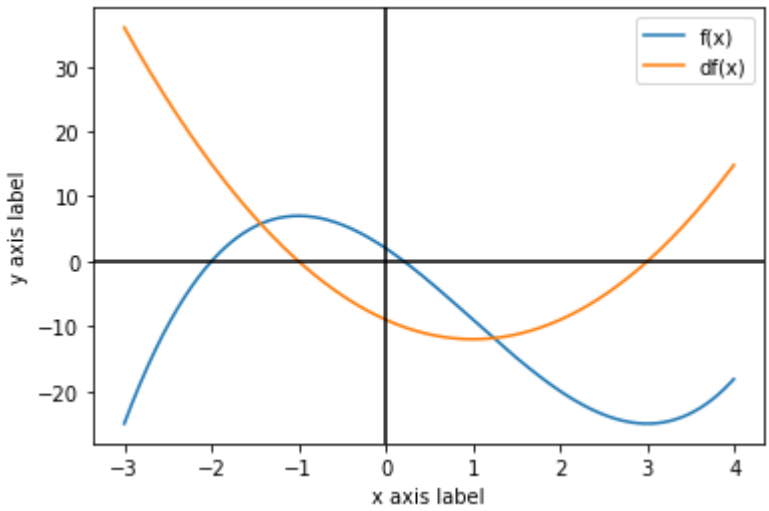

Figure 2-2 $f(x) = x^3–3x^2–9x + 2$ and the function curve of its derivative $f'(x)$

Note that the above proposition only illustrates the necessary condition at the extreme point of the function, but not the sufficient condition, that is to say, the derivative $f'(x_0) = 0$ at a function x_0 does not mean that x_0 must be an extreme point. For example, the derivative $f'(0)$ of $f(x) = x^3$ at $x = 0$ is also 0, but this point is not the extreme point of the function. In fact, this is a monotonically increasing function, as shown in Figure 2-3.

```
x = np.arange(-3, 3, 0.01)
f_x = np.power(x,3)

plt.plot(x,f_x)
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.axvline(x=0, color='k')
plt.axhline(y=0, color='k')
plt.show()
```

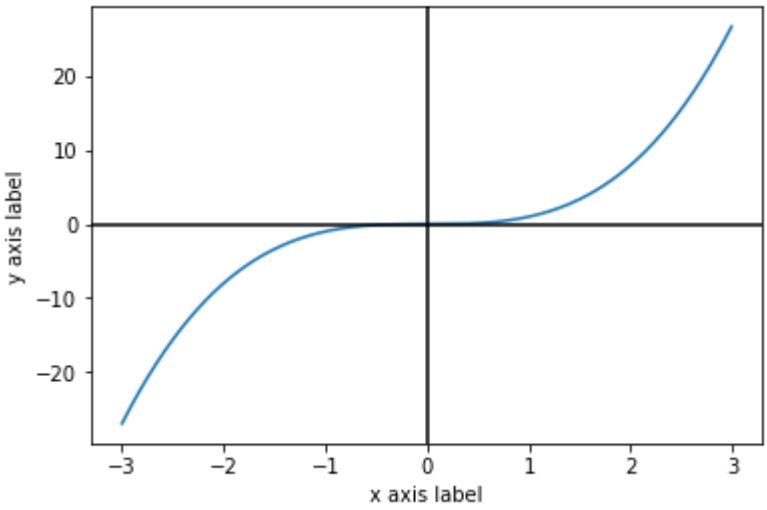

Figure 2-3 Function curve of $f(x) = x^3$

Obviously, the necessary conditions for the extremum of the function can be extended to multivariate functions, that is, for a multivariate function $f(x_1, x_2, \cdots, x_n)$, if the function is at a certain point $x^* = (x_1^*, x_2^*, \cdots, x_n^*)$ obtains an extreme value and the gradient at this point exists (that is, all partial derivatives exist), then the gradient at this point must be 0 (that is, each partial derivative value is is 0). Right now:

$$\frac{\partial f(x_1,x_2,\cdots,x_n)}{\partial x_i}|_{x^*} = 0, \quad i = 1, 2, \cdots, n$$

2.2 Gradient descent method (gradient descent)

For a one-variable function f(x), if there is a small change Δx near a certain point x, then the change $f(x+\Delta x)-f(x)$ of f(x) can be expressed as follows Differential form of :

$$f(x+\Delta x)-f(x) \simeq f'(x)\Delta x$$

Therefore, near x, if Δx and $f'(x)$ have the same sign, then $f'(x)\Delta x$ or $f(x+\Delta x)-f(x)$ is a positive number, and if Δx and $f'(x)$ have opposite signs, then $f'(x)\Delta x$ or $f(x+\Delta x)-f(x)$ is a negative number. If you take $\Delta x=-\alpha f'(x)$ (where α is a small positive number),then $f(x+\Delta x)-f(x)=-\alpha f'(x)^2$ is a negative number, that is, the value of $f(x+\Delta x)$ will be smaller than $f(x)$. In other words, x moves Δx along the opposite direction $-f'(x)$ of $f'(x)$, and its function value $f(x+\Delta x)$ is smaller than the original $f(x)$.

As shown in Figure 2.4, the function value $f(x)$ of the function $f(x)=x^2+0.2$ at $x=1.5$ is 2.45, and the derivative value $f'(x)$ is 3.0, which is a positive number, pointing to the positive direction of the x axis on the domain of $f(x)$, that is, the x axis, as shown by the long arrow in the figure.

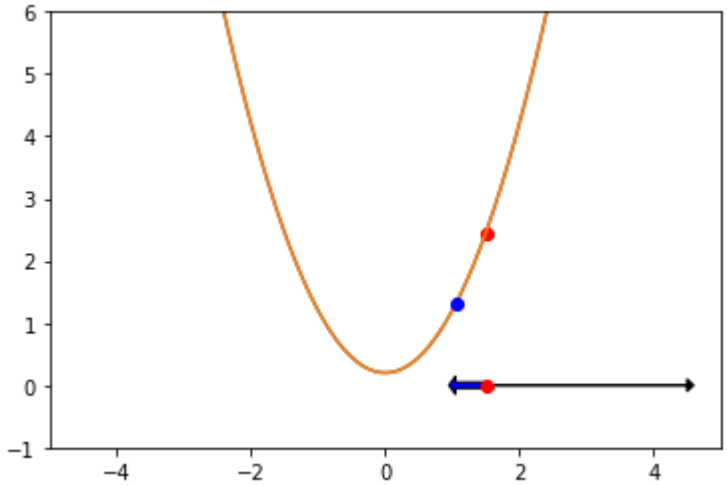

Figure 2-4 $f'(1.5)=2.45>0$, Δx and $f'(x)$ move with the same sign, the function value increases, otherwise, the function value decreases

Let $\alpha=0.15$, $\Delta x=-\alpha f'(x)=-0.449$, move x along this Δx (in the direction of the blue arrow in the figure) to $x_{new}=x+\Delta x=1.05$, the f(1.05) function value at the new $x=1.05$ obtained is 1.3025, which is the y coordinate value of the blue point on the curve in the figure. Because Δx and $f'(x)$ are in opposite directions (one negative and one positive), this f(1.05) must be smaller than the original f(1.5).

As long as this process is repeated continuously, that is, moving x along the opposite direction ($-f'(x)$) of its derivative $f'(x)$ by a small increment $-\alpha f'(x)$ can reach a new $x_{new}=x-\alpha f'(x)$, and the function value $f'(x_{new})$ of this new x_{new} must be smaller than the previous function value $f'(x)$.

As x continues to approach the x value of the minimum point, the derivative f'(x) is also close to 0 (because the derivative $f(x^*)=0$ of the function extreme point x^*), and the increment Δx of x movement is getting closer and closer to 0.

This is the idea of **gradient descent method**, that is, starting from an initial x, the value of x is continuously updated with the following formula:

$$x=x-\alpha f'(x)$$

For the current x, moving x along its negative derivative (gradient) direction (ie $-f'(x)$) can make $f(x)$ keep getting smaller. Ideally, x of minimum $f(x)$ is reached, where $f'(x)=0$. Then update x iteratively, and the value of x will no longer change. As shown in Figure 2-5, x is constantly updated iteratively, so that it is constantly approaching the extreme point.

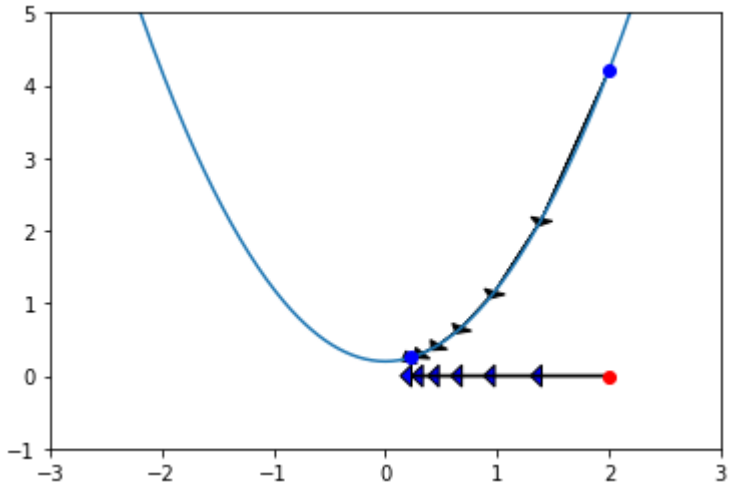

Figure 2-5 x moves along $-f'(x)$, the function value keeps decreasing

Of course, the pace of this movement (ie $-\alpha f'(x)$) cannot be too large, because according to the definition of the derivative, the above approximate formula is only applicable near x. If the moving pace is too large, the optimal value of x may be skipped, making the value of x constantly oscillating back and forth. As shown in Figure 2-6.

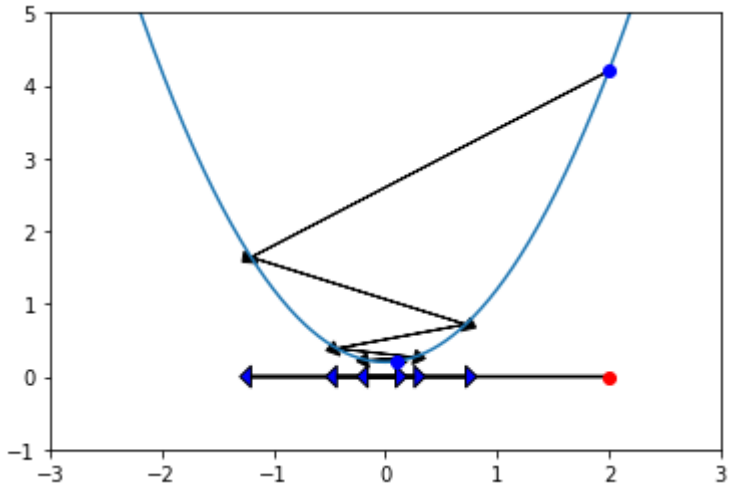

Figure 2-6 The magnitude of x change $-\alpha f'(x)$ is too large, the function value will oscillate

The gradient descent method is to find an approximate optimal solution. In order to avoid iterating, the following methods can be used to check whether it is close enough to the optimal solution:

- The absolute value of the derivative (gradient) $f'(x)$ is small enough.
- The number of iterations has reached the preset maximum number of iterations.

The following is the code of the gradient descent method, where the parameter df is used to calculate the derivative $f'(x)$ of a function $f(x)$, x is the initial value of the variable, alpha is the learning rate, and iterations represent The number of iterations, epsilon checks whether the value of df=$f'(x)$ is close to 0.

```
def gradient_descent(df,x,alpha=0.01, iterations = 100,epsilon = 1e-8):
    history=[x]
    for i in range(iterations):
        if abs(df(x))<epsilon:
            print("The gradient is small enough!")
            break
        x = x-alpha* df(x)
        history.append(x)
    return history
```

This gradient descent function saves all updated x during the iteration process in a python list object history and returns this object.

For the above function $f(x) = x^3 - 3x^2 - 9x + 2$, its derivative $f'(x) = 3x^2 - 6x - 9$. If you want the minimum value of the function $f(x)$ near $x = 1$, you can call this function gradient_descent():

```
df = lambda x: 3*x**2-6*x-9
path = gradient_descent(df,1.,0.01,200)
print(path[-1])
```

```
The gradient is small enough!
    2.999999999256501
```

Get the extreme point x=2.999999999256501 of $f(x)$. The points on the curve corresponding to x in the iteration process can be drawn:

```
f = lambda x: np.power(x,3)-3*x**2-9*x+2
x = np.arange(-3, 4, 0.01)
y= f(x)
plt.plot(x,y)

path_x = np.asarray(path) #.reshape(-1,1)
path_y=f(path_x)
plt.quiver(path_x[:-1], path_y[:-1], path_x[1:]-path_x[:-1], path_y[1:]-path_y[:-1],
scale_units='xy', angles='xy', scale=1, color='k')
plt.scatter(path[-1],f(path[-1]))
plt.show()
```

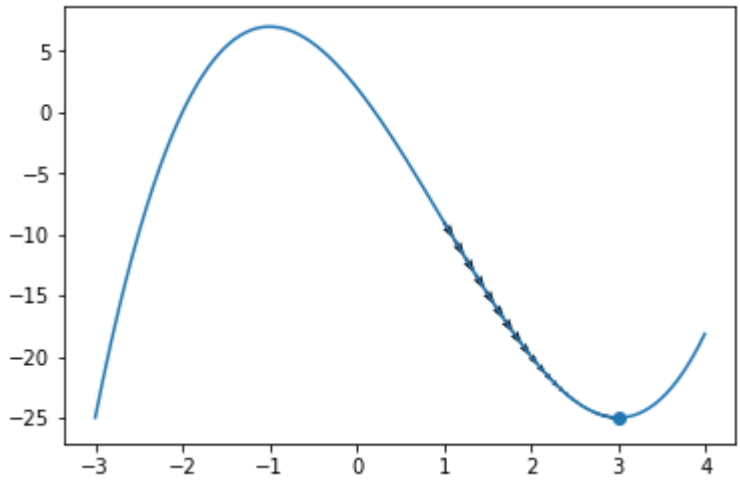

Figure 2-7 x gradually converges to the minimum point

Among them, the quiver function of matplotlib can use arrows to draw velocity vectors, and its function format is:

```
quiver([X, Y], U, V, [C], **kw)
```

Where X, Y are 1D or 2D arrays, indicating the position of the arrow, and U, V are the same 1D or 2D arrays, indicating the speed (vector) of the arrow. For other parameters, please refer to the official documentation.

For multivariable functions, the principle of the gradient descent method is the same, but the gradient is used instead of the derivative.

$$f(x+\Delta x)-f(x)\simeq \nabla f(x)\Delta x$$

The following is the Beale's function of Wikipedia.

$$f(x,y)=(1.5-x+xy)^2+(2.25-x+xy^2)^2+(2.625-x+xy^3)^2$$

The global minimum of this function is $(3,0.5)$. The function value can be calculated with the following python code:

```
f  = lambda x, y: (1.5 - x + x*y)**2 + (2.25 - x + x*y**2)**2 + (2.625 - x + x*y**3)**2
```

To draw this surface, first take some evenly distributed coordinate values on the x and y axes:

```
xmin, xmax, xstep = -4.5, 4.5, .2
ymin, ymax, ystep = -4.5, 4.5, .2
x_list = np.arange(xmin, xmax + xstep, xstep)
y_list = np.arange(ymin, ymax + ystep, ystep)
```

Then use the np.meshgrid() function to get the grid points (x, y) at their intersections according to the above x_list and y_list, and calculate the function values corresponding to these grid coordinate points:

```
x, y = np.meshgrid(x_list, y_list)
z = f(x, y)
```

Finally, the plot_surface() function can be called to draw this surface:

```
ax.plot_surface(x, y, z, norm=LogNorm(), rstride=1, cstride=1,
                edgecolor='none', alpha=.8, cmap=plt.cm.jet)
```

The complete code is as follows:

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

%matplotlib inline

f  = lambda x, y: (1.5 - x + x*y)**2 + (2.25 - x + x*y**2)**2 + (2.625 - x + x*y**3)**2

minima = np.array([3., .5])
minima_ = minima.reshape(-1, 1)

xmin, xmax, xstep = -4.5, 4.5, .2
ymin, ymax, ystep = -4.5, 4.5, .2
x_list = np.arange(xmin, xmax + xstep, xstep)
y_list = np.arange(ymin, ymax + ystep, ystep)
x, y = np.meshgrid(x_list, y_list)
z = f(x, y)

fig = plt.figure(figsize=(8, 5))
ax = plt.axes(projection='3d', elev=50, azim=-50)

ax.plot_surface(x, y, z, norm=LogNorm(), rstride=1, cstride=1,
```

```
                edgecolor='none', alpha=.8, cmap=plt.cm.jet)
ax.plot(*minima_, f(*minima_), 'r*', markersize=10)

ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
ax.set_zlabel('$z$')

ax.set_xlim((xmin, xmax))
ax.set_ylim((ymin, ymax))

plt.show()
```

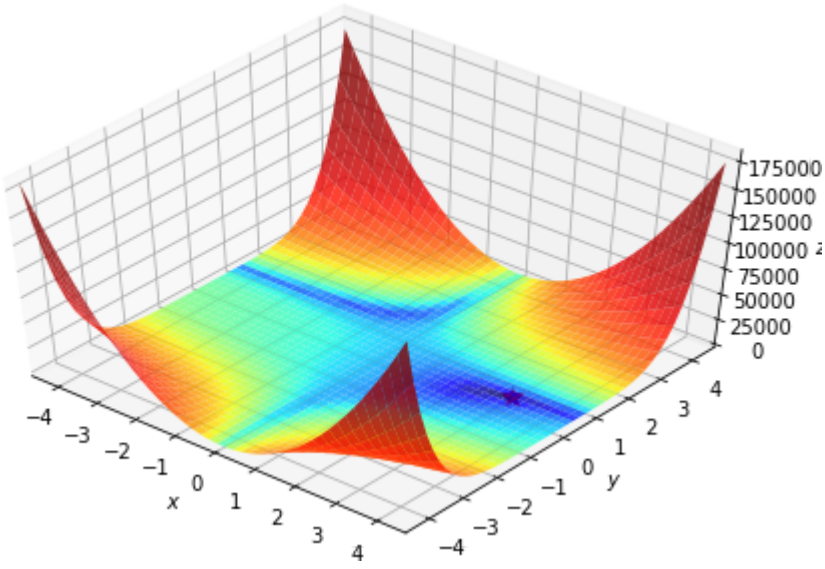

Figure 2-8 Drawn f(x,y) surface

The partial derivative of $f(x, y)$ with respect to x, y is:

$$\frac{\partial f(x,y)}{\partial x} = 2(1.5 - x + xy)(y - 1) + 2(2.25 - x + xy^2)(y^2 - 1) + 2(2.625 - x + xy^3)(y^3 - 1)$$
$$\frac{\partial f(x,y)}{\partial y} = 2(1.5 - x + xy)x + 2(2.25 - x + xy^2)(2yx) + 2(2.625 - x + xy^3)(3y^2x)$$

The gradient directions at these grid points can be plotted on a 2D coordinate plane using matplotlib's quiver function.

```
df_x  = lambda x, y: 2*(1.5 - x + x*y)*(y-1) + 2*(2.25 - x + x*y**2)*(y**2-1) + 2*(2.625 - x
+ x*y**3)*(y**3-1)
df_y  = lambda x, y: 2*(1.5 - x + x*y)*x + 2*(2.25 - x + x*y**2)*(2*x*y) + 2*(2.625 - x +
x*y**3)*(3*x*y**2)
dz_dx = df_x(x, y)
dz_dy = df_y(x, y)

fig, ax = plt.subplots(figsize=(10, 6))

ax.contour(x, y, z, levels=np.logspace(0, 5, 35), norm=LogNorm(), cmap=plt.cm.jet)
ax.quiver(x, y, x - dz_dx, y - dz_dy, alpha=.5)
ax.plot(*minima_, 'r*', markersize=18)

ax.set_xlabel('$x$')
ax.set_ylabel('$y$')

ax.set_xlim((xmin, xmax))
ax.set_ylim((ymin, ymax))
```

```
plt.show()
```

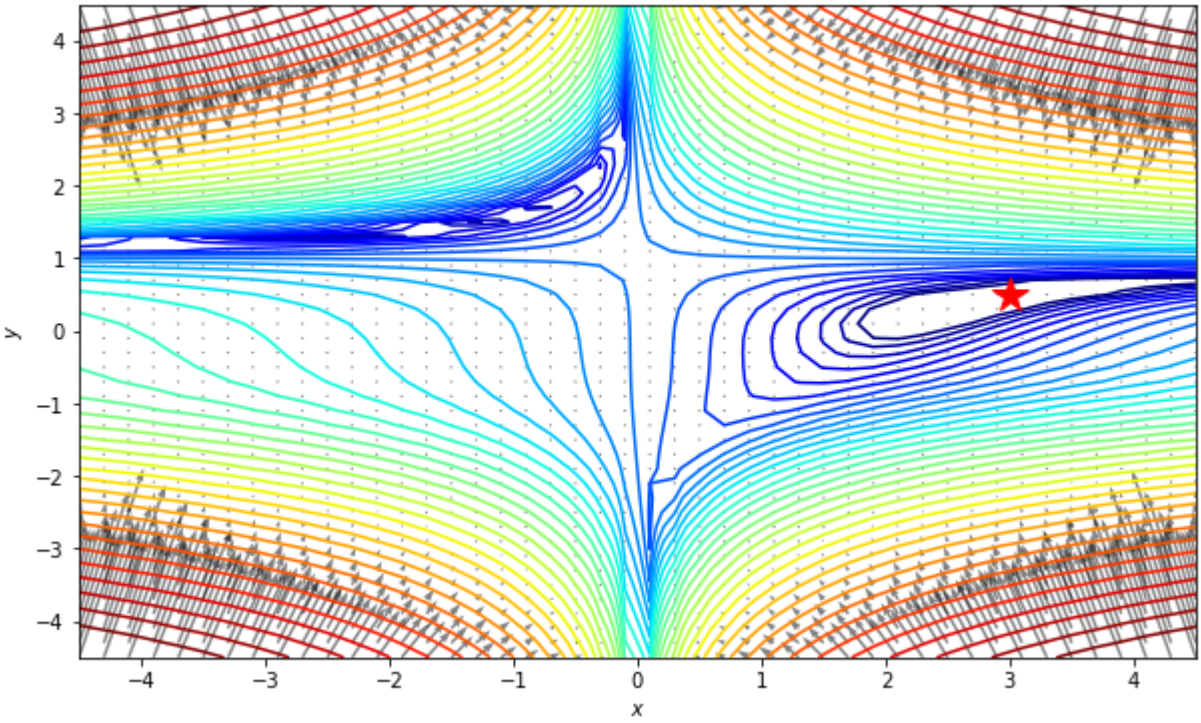

Figure 2-9. Domain coordinate contours and gradient directions at grid points of the isosurface of the function f(x,y)

In order to directly use the previous gradient descent method code, x in the previous gradient descent method code can be represented by a numpy vector, and

```
if abs(df(x))<epsilon:
```

change into:

```
if np.max(np.abs(df(x)))<epsilon:
```

First combine the separated x and y coordinate arrays into one array:

```
print(x.shape)
print(y.shape)

x_ = np.vstack((x.reshape(1, -1) ,y.reshape(1, -1) ))
print(x_.shape)
```

```
(46, 46)
    (46, 46)
    (2, 2116)
```

You can define a gradient function df for this vectorized coordinate point x. The following code also gives the implementation of the modified vectorized version of the gradient descent algorithm:

```
df = lambda x: np.array( [2*(1.5 - x[0] + x[0]*x[1])*(x[1]-1) + 2*(2.25 - x[0] +
x[0]*x[1]**2)*(x[1]**2-1)
                                        + 2*(2.625 - x[0] + x[0]*x[1]**3)*(x[1]**3-1),
                          2*(1.5 - x[0] + x[0]*x[1])*x[0] + 2*(2.25 - x[0] + x[0]*x[1]**2)*
(2*x[0]*x[1])
```

```
                                        + 2*(2.625 - x[0] + x[0]*x[1]**3)*(3*x[0]*x[1]**2)])

def gradient_descent(df,x,alpha=0.01, iterations = 100,epsilon = 1e-8):
    history=[x]
    for i in range(iterations):
        if np.max(np.abs(df(x)))<epsilon:
            print("The gradient is small enough!")
            break
        x = x-alpha* df(x)
        history.append(x)
    return history
```

The following code starts from x0=(3., 4.) to solve the extreme point of this surface:

```
x0=np.array([3., 4.])
print("initial point",x0,"gradient",df(x0))

path = gradient_descent(df,x0,0.000005,300000)
print("Extreme point: ",path[-1])
```

Gradient [25625.25 57519.] of initial point [3. 4.]
Extreme points: [2.70735828 0.41689171]

Because the initial gradient value of x starts to be very large, the learning rate α must take a small number (such as 0.000005), otherwise it will cause shock or infinite value, and finally converge to `[2.70735828 0.41689171]`, But it is not the best point, you can see this situation more intuitively by drawing the change of x during the iteration process.

```
def plot_path(path,x,y,z,minima_,xmin, xmax,ymin, ymax):
    fig, ax = plt.subplots(figsize=(10, 6))
    ax.contour(x, y, z, levels=np.logspace(0, 5, 35), norm=LogNorm(), cmap=plt.cm.jet)
    #ax.scatter(path[0],path[1]);
    ax.quiver(path[:-1,0], path[:-1,1], path[1:,0]-path[:-1,0], path[1:,1]-path[:-1,1],
scale_units='xy', angles='xy', scale=1, color='k')
    ax.plot(*minima_, 'r*', markersize=18)

    ax.set_xlabel('$x$')
    ax.set_ylabel('$y$')
    ax.set_xlim((xmin, xmax))
    ax.set_ylim((ymin, ymax))

path = np.asarray(path)
plot_path(path,x,y,z,minima_,xmin, xmax,ymin, ymax)
```

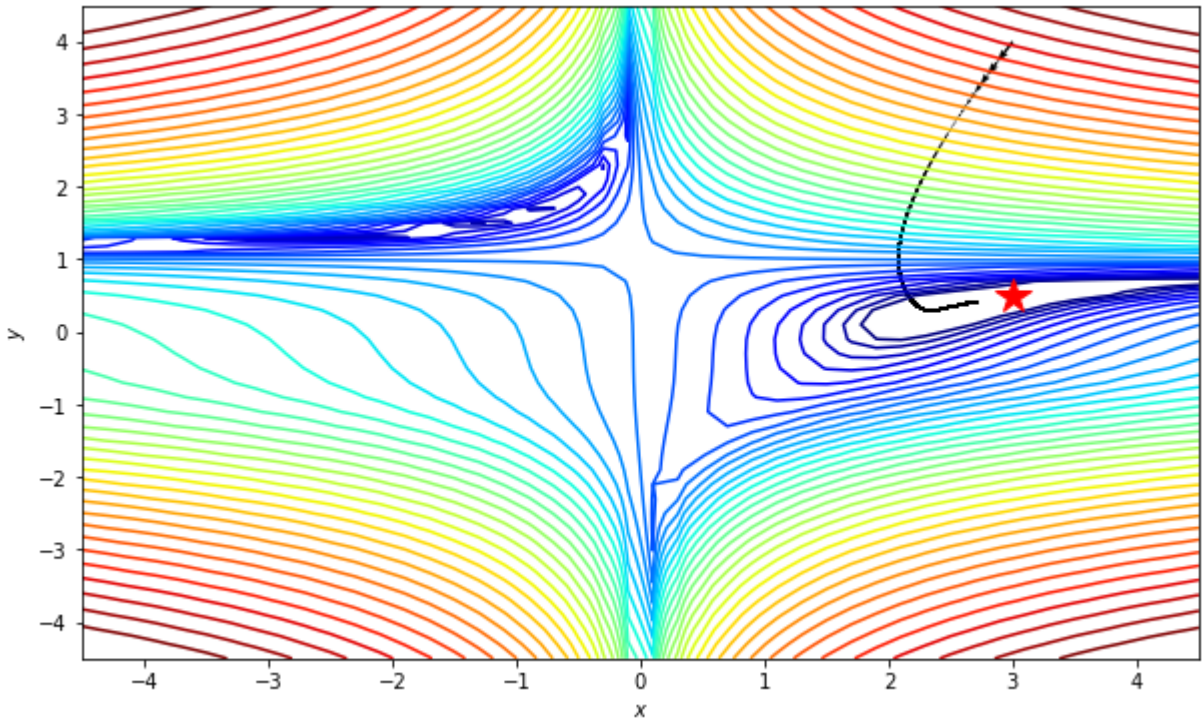

Figure 2-10 During the iteration process, the gradient value becomes smaller and smaller, and the convergence becomes slower and slower

During the iterative process, the gradient value becomes smaller and smaller, and the same learning rate makes the update of x very slow. Even after 100,000 iterations, it still fails to approach the optimal solution. A natural approach is to use an adaptive learning rate, i.e. increase the learning rate when the gradient becomes small. As an exercise, the reader can try to modify the gradient descent algorithm to get to the optimal solution better and faster.

2.3 Parameter optimization strategy of gradient descent method

The learning rate in the basic gradient descent algorithm is a fixed value, and the gradient size is constantly changing during the iterative process. If the learning rate is too large, the variable to be solved will oscillate back and forth. If the learning rate is too small, the convergence will be very slow or even stagnant. The initial learning rate is moderate, but as the convergence approaches the optimal solution, its gradient is also close to 0, which will also cause stagnation. Naturally, the learning rate should be adjusted as it gradually converges during the iterative process, that is, a variable learning rate is used to update the variable x to be solved during the iterative process.

In order to ensure that the optimal solution can be approached better and faster, many improvements to the gradient descent method have been proposed. These improvements use a changing learning rate or strategy to update the solution variables (also called parameters). The update strategies or methods for variables (parameters) include: Momentum, Nesterov accelerated gradient, Adagrad, Adadelta, RMSprop, Adam, AdaMax, Nadam, AMSGrad, etc.

It should be noted that the function may be a multi-variable function, therefore, its variable x can be a vector $\boldsymbol{x}$ composed of multiple values. Only some of the commonly used optimization strategies are described below.

2.3.1 Momentum momentum method

The gradient descent method uses the learning rate α and the negative direction of the gradient, that is, $-\alpha\nabla f(x)$ to update x each time, that is, to update the vector $-\alpha\nabla f(x) of x)$ depends entirely on the current calculated gradient, and the Momentum momentum method updates the vector of x not only considering the current gradient, but also considering the last update vector, that is, the updated vector is considered to have

inertia. Assuming that v_{t-1} is the vector used for update last time, the current updated vector is:

$v_t = \gamma v_{t-1} + \alpha \nabla f(x)$

Update x with this v:

$x = x - v_t$

This vector used to update x is called **momentum**. The momentum method regards the update vector as the velocity of a moving object, and the velocity has inertia. Due to the combination of the previous update vector and the current gradient, it alleviates the sharp changes in the gradient at different times, making the updated vector smoother, that is, maintaining the inertia of the previous motion, so that where the gradient is small, there is still a large motion. The speed will not overshoot due to the sudden increase of the gradient. This method is like a ball with weight rolling downhill, maintaining a certain amount of inertia while looking for the steepest descent path. Ordinary gradient descent only determines the speed of movement according to the degree of steepness, just like rushing fast in steep places and hardly moving in flat places.

The initial value of $\boldsymbol{v}$ is 0, which can be expressed in python code as:

```
v= np.zeros_like(x)
```

That is, $\boldsymbol{v}$ is a tensor with the same shape as $\boldsymbol{x}$ with an initial value of 0. In the iterative process, $\boldsymbol{v}$ is updated first, and then the parameter $\boldsymbol{x}$ of the function is updated:

```
v = gamma*v+alpha* df(x)
x = x-v
```

The following is the gradient descent method based on the momentum method:

```
def gradient_descent_momentum(df, x, alpha=0.01, gamma = 0.8, iterations = 100, epsilon = 1e-
6):
    history=[x]
    v= np.zeros_like(x)            # momentum
    for i in range(iterations):
        if np.max(np.abs(df(x)))<epsilon:
            print("The gradient is small enough!")
            break
        v = gamma*v+alpha* df(x)           # update momentum
        x = x-v                    # Update variables (parameters)

        history.append(x)
    return history
```

Use the gradient descent method of this momentum method to solve the above problem:

```
path = gradient_descent_momentum(df,x0,0.000005,0.8,300000)
print(path[-1])
path = np.asarray(path)
```

[2.96324633 0.49067782]

It can be seen that the solution of the momentum method is very close to the optimal solution. as the picture shows.

```
plot_path(path,x,y,z,minima_,xmin, xmax,ymin, ymax)
```

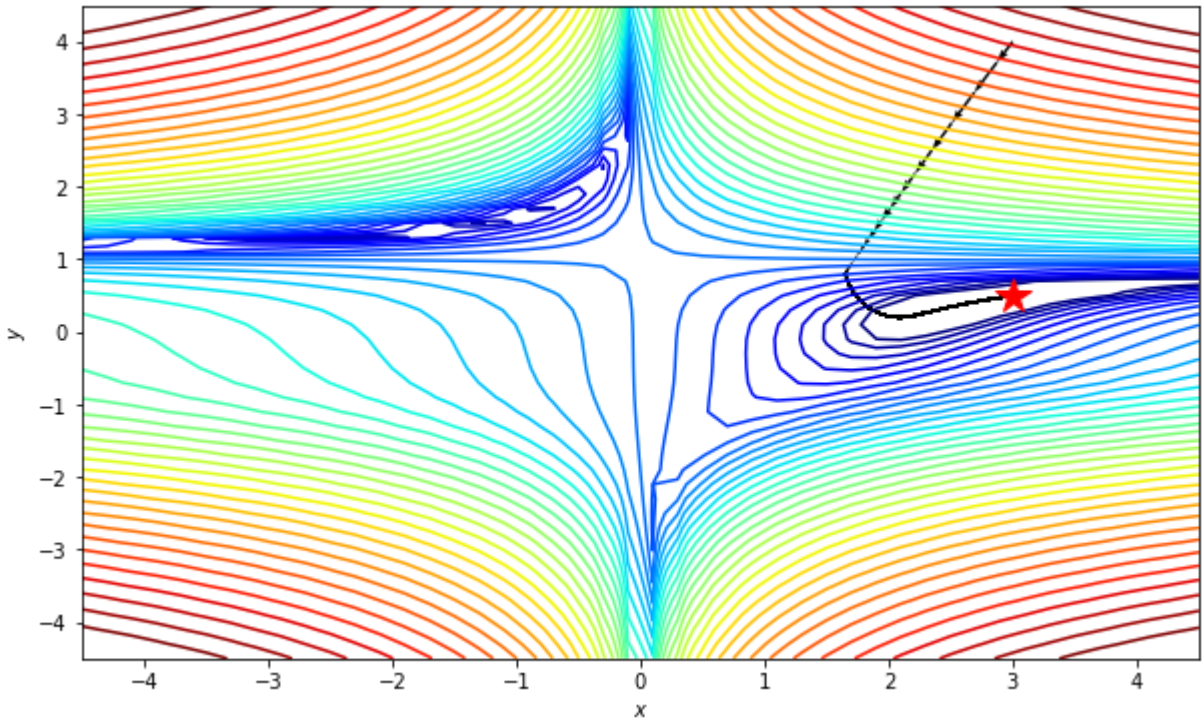

Figure 2-11 The momentum method quickly converges to a near-optimal solution

2.3.2 Adagrad method

According to the variable update formula of the gradient descent method $\boldsymbol{x} = \boldsymbol{x} - \alpha \nabla f(\boldsymbol{x})$, what affects the variable update is the product of the learning rate and the gradient $\alpha \nabla f(\boldsymbol{x})$, the gradient is too large or too small and the learning rate is too large or too small will affect the convergence of the algorithm.

For a multivariate function, the magnitude of the partial derivatives for each variable can vary widely. For example, the absolute values of the partial derivatives $\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}$ of a function $f(x_1, x_2)$ of two variables at a certain point (x_1, x_2) may differ greatly.It is inappropriate to use the same learning rate for them. The appropriate learning rate for one component is too large or too small for the other component, resulting in shock and stagnation. That is, it is inappropriate to directly update with the following formula:

$$x_1 = x_1 - \alpha \frac{\partial f}{\partial x_1}$$
$$x_2 = x_2 - \alpha \frac{\partial f}{\partial x_2}$$

The Adagrad method can be translated as "adaptive (ada) gradient (grad)" from the noun, which divides each gradient component by the historical cumulative value of the gradient component, so that the problem of unbalanced gradient sizes of different components can be eliminated. For 2 components (x_1, x_2), if the historical cumulative value (G_1, G_2) of the respective components is calculated respectively, the update formula of the 2 components is:

$$x_1 = x_1 - \alpha \frac{1}{G_1} \frac{\partial f}{\partial x_1}$$
$$x_2 = x_2 - \alpha \frac{1}{G_2} \frac{\partial f}{\partial x_2}$$

Use the notation $g_{t,i} = \nabla_\theta f(x_{t,i})$ to represent the partial derivative $\frac{\partial f}{\partial x_i}$ of the component x_i in the t-th iteration, the component gradient of all rounds from `t'=1` to `t'=t` can be calculated as follows:

$$G_{t,i} = \sqrt{\sum_{t'=1}^{t} g_{t',i}^2}$$

Divide $g_{t,i}$ by $G_{t,i}$ to update the component:

$$x_{t+1,i} = x_{t,i} - \alpha \frac{1}{\sqrt{\sum_{t'=1}^{t} g_{t',i}^2}} g_{t,i}$$

In order to prevent the divisor from being 0, a small positive number ϵ can be added to this denominator, so that the parameter update formula of AdaGrad is:

$$x_{t+1,i} = x_{t,i} - \alpha \frac{1}{\sqrt{\sum_{t'=1}^{t} g_{t',i}^2 + \epsilon}} g_{t,i}$$

Compare the basic parameter update formula:

$$x_{t+1,i} = x_{t,i} - \alpha g_{t,i}$$

It can be seen that the AdaGrad method eliminates the unbalanced problem of component gradient sizes. The parameter update formula of AdaGrad can be written in vector form:

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t - \alpha \frac{1}{\sqrt{\sum_{t'=1}^{t} \boldsymbol{g}_{t'}^2 + \epsilon}} \odot \boldsymbol{g}_t$$

The accumulated G_t^2 can be recorded with the variable gl with an initial value of 0. In each round of iteration, the python code for AdaGrad parameter update is as follows:

```
gl += df(x)**2
x = x-alpha* df(x)/(sqrt(gl)+epsilon)
```

The main advantage of the AdaGrad method is that it eliminates the influence of different gradient values, so that the learning rate can be set to a fixed value without continuously adjusting the learning rate in the iterative process. The general learning rate is set to 0.01. The main disadvantage of the AdaGrad method is that with the iterative process, the cumulative sum $\sum_{t'=1}^{t} \boldsymbol{g}_{t'}^2$ will become larger and larger, because each of them is a positive number . This can lead to slow learning, or even a standstill. Also, making each component gradient have a consistent pace may not be realistic and can divert the direction of progress from the direction of the optimal solution.

The code of the gradient descent method based on the Adagrad parameter update method is as follows:

```
def gradient_descent_Adagrad(df,x,alpha=0.01,iterations = 100,epsilon = 1e-8):
    history=[x]
    #v= np.zeros_like(x)
    gl = np.ones_like(x)
    for i in range(iterations):
        if np.max(np.abs(df(x)))<epsilon:
            print("The gradient is small enough!")
            break
        grad = df(x)
        gl += grad**2
        x = x-alpha* grad/(np.sqrt(gl)+epsilon)
        history.append(x)
    return history
```

For the above problem, perform the gradient descent algorithm:

```
path = gradient_descent_Adagrad(df,x0,0.1,300000,1e-8)
print(path[-1])
path = np.asarray(path)
```

[-0.69240717 1.76233766]

It can be seen that due to the equalization of the component gradients, the forward direction of the variable update deviates from the optimal solution method, and converges to another local optimal solution.

```
plot_path(path,x,y,z,minima_,xmin, xmax,ymin, ymax)
```

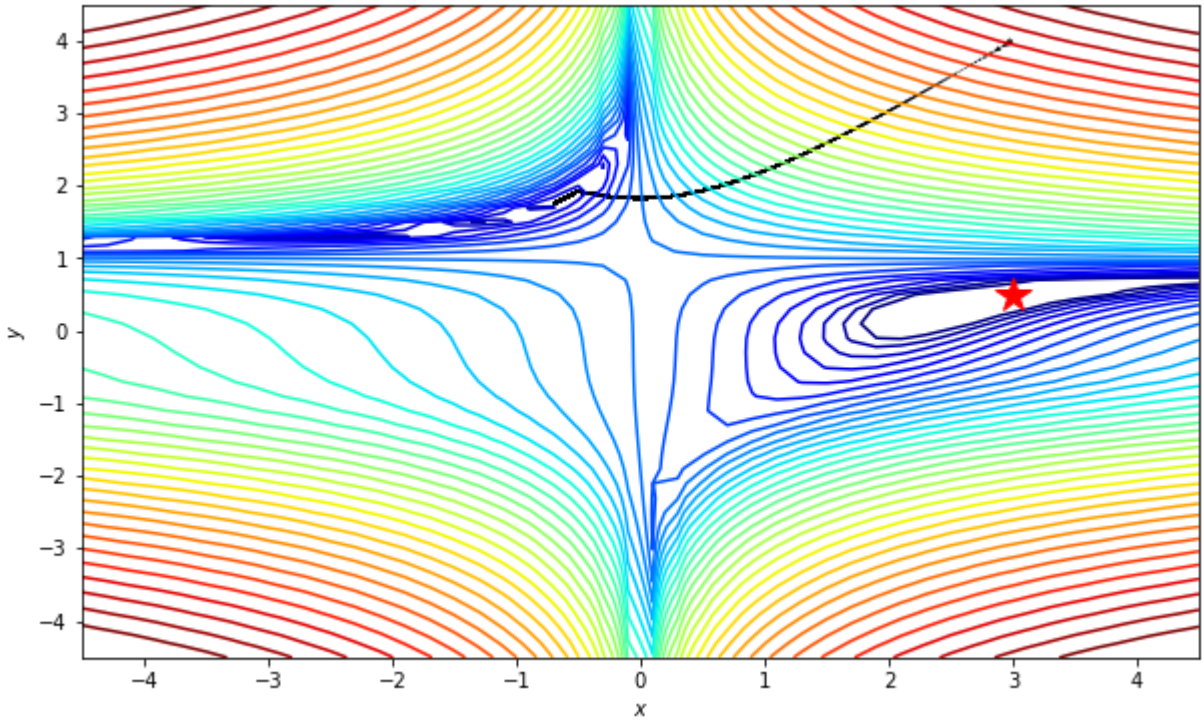

Figure 2-12 Adagrad method converges to another local minimum

2.3.3 Adadelta method

Reviewing the basic parameter update method, use $\Delta \boldsymbol{x}_t$ to represent the update vector of the parameter:

$\Delta \boldsymbol{x}_t = -\eta \cdot \boldsymbol{g}_t$
$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t + \Delta \boldsymbol{x}_t$

The update vector of the AdaGrad method is:

$$\Delta \boldsymbol{x}_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot \boldsymbol{g}_t$$

Here $G_t = \sum \boldsymbol{g}_t^2$ is the historical sum of squares of g_t. With the iterative process, this value G_t is getting bigger and bigger, resulting in $\Delta \boldsymbol{x}_t$ is getting smaller and smaller, so the convergence is getting slower and slower. The solution is to replace G_t with the sum of mean squares $E[g^2]_t = \frac{G_t}{t}$ instead of the sum of squares. This $E[g^2]_t$ can be calculated using the moving average method, that is, to make an average of the last average value and the current value:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

The Adadelta method goes a step further and uses such a moving average method for the update vector to make the change of the update vector smoother.

$$E[\Delta \boldsymbol{x}^2]_t = \gamma E[\Delta \boldsymbol{x}^2]_{t-1} + (1-\gamma) \Delta \boldsymbol{x}_t^2$$

The final update vector is:

$$\Delta \boldsymbol{x}_t = -\sqrt{\frac{E[\Delta \boldsymbol{x}^2]_{t-1} + \epsilon}{E[g^2]_t + \epsilon}} g_t$$

Use $RMS[\Delta \boldsymbol{x}]_{t-1}, RMS[g]_t$ to represent $E[\Delta \boldsymbol{x}^2]_{t-1} + \epsilon$ and $E[g^2]_t + \epsilon$, the update vector can be expressed as:

$$\Delta \boldsymbol{x}_t = -\sqrt{\frac{RMS[\Delta \boldsymbol{x}]_{t-1}}{RMS[g]_t}} g_t$$

So the parameter update formula is:

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t + \alpha \Delta \boldsymbol{x}_t$$

The python code for the Adadelta method is as follows:

```
Eg = rho*Eg+(1-rho)*(grad**2)                # Update the cumulative sum of squares of the
gradient
delta = np.sqrt((Edelta+epsilon)/(Eg+epsilon))*grad  # calculate update vector
x = x - alpha* delta
Edelta = rho*Edelta+(1-rho)*(delta**2)       # Cumulative update of update vector
```

The decay rate parameter ρ of the Adadelta method is usually set to 0.9, and the initial value of $\Delta \boldsymbol{x}_t, E[\Delta \boldsymbol{x}^2]_t, E[g^2]_t$ is also 0. The code of the gradient descent method based on the Adadelta parameter update method is as follows:

```
def gradient_descent_Adadelta(df,x,alpha = 0.1,rho=0.9,iterations = 100,epsilon = 1e-8):
    history=[x]
    Eg = np.ones_like(x)
    Edelta = np.ones_like(x)
    for i in range(iterations):
        if np.max(np.abs(df(x)))<epsilon:
            print("The gradient is small enough!")
            break
        grad = df(x)
        Eg = rho*Eg+(1-rho)*(grad**2)
        delta = np.sqrt((Edelta+epsilon)/(Eg+epsilon))*grad
        x = x- alpha*delta
        Edelta = rho*Edelta+(1-rho)*(delta**2)
        history.append(x)
    return history
```

```
path = gradient_descent_Adadelta(df,x0,1.0,0.9,300000,1e-8)
print(path[-1])
path = np.asarray(path)
```

```
[2.9386002 0.45044889]
```

It can be seen that the Adadelta method can also converge to a close to the optimal solution.

```
plot_path(path,x,y,z,minima_,xmin, xmax,ymin, ymax)
```

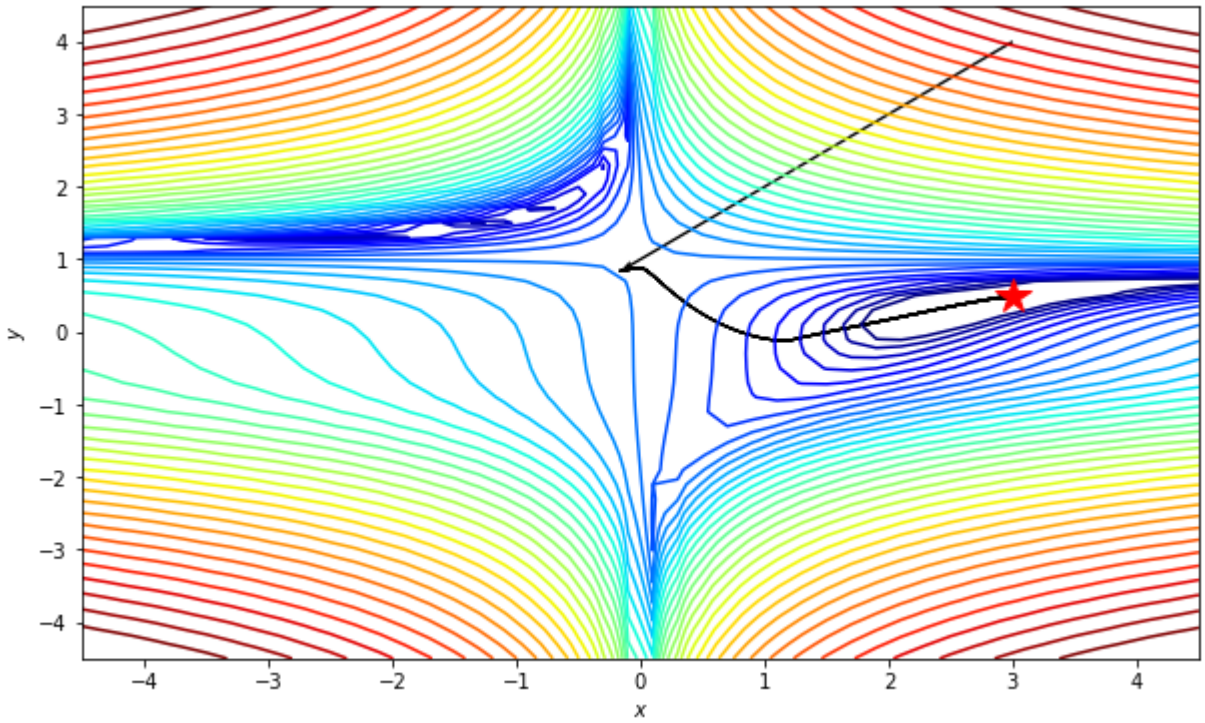

Figure 2-13 Adadelta method can also converge to a near optimal solution

2.3.4 RMSprop method

Similar to the momentum method, RMSprop uses the following formulas to update the momentum and parameters:

$$v_t = \beta v_{t-1} + (1-\beta)\nabla f(x)^2$$
$$x = x - \alpha \frac{1}{\sqrt{v_t+\epsilon}} \nabla f(x)$$

The idea is to divide each value of the gradient by the length (the absolute value of the value), that is, convert it into a unit length, so that the parameter x is always updated with a fixed step size α. In order to calculate the length of each component of the gradient, RMSprop is similar to the momentum method to calculate the square value of the moving average length of the gradient value, that is, $f(x)^2$.

The python code for updating model parameters by the RMSprop method is as follows:

```
v= np.ones_like(x)
#...
grad = df(x)
v = beta*v+(1-beta)* grad**2
x = x-alpha*(1/(np.sqrt(v)+epsilon))*grad
```

The code of the gradient descent method based on the RMSprop parameter update method is as follows:

```
def gradient_descent_RMSprop(df,x,alpha=0.01,beta = 0.9, iterations = 100,epsilon = 1e-8):
    history=[x]
    v= np.ones_like(x)
    for i in range(iterations):
        if np.max(np.abs(df(x)))<epsilon:
            print("The gradient is small enough!")
            break
        grad = df(x)
        v = beta*v+(1-beta)*grad**2
        x = x-alpha*grad/(np.sqrt(v)+epsilon)

        history.append(x)
    return history
```

For the above problem, perform the gradient descent algorithm:

```
path = gradient_descent_RMSprop(df,x0,0.000005,0.99999999999,300000,1e-8)
print(path[-1])
path = np.asarray(path)
```

```
[2.70162562 0.41500366]
```

The results for the model parameters are not good enough, you can increase the number of iterations:

```
path = gradient_descent_RMSprop(df,x0,0.000005,0.99999999999,900000,1e-8)
print(path[-1])
path = np.asarray(path)
```

```
[2.9082809 0.47616156]
```

It can be seen that the basic convergence is close to the optimal solution, as shown in Figure 2-14:

```
plot_path(path,x,y,z,minima_,xmin, xmax,ymin, ymax)
```

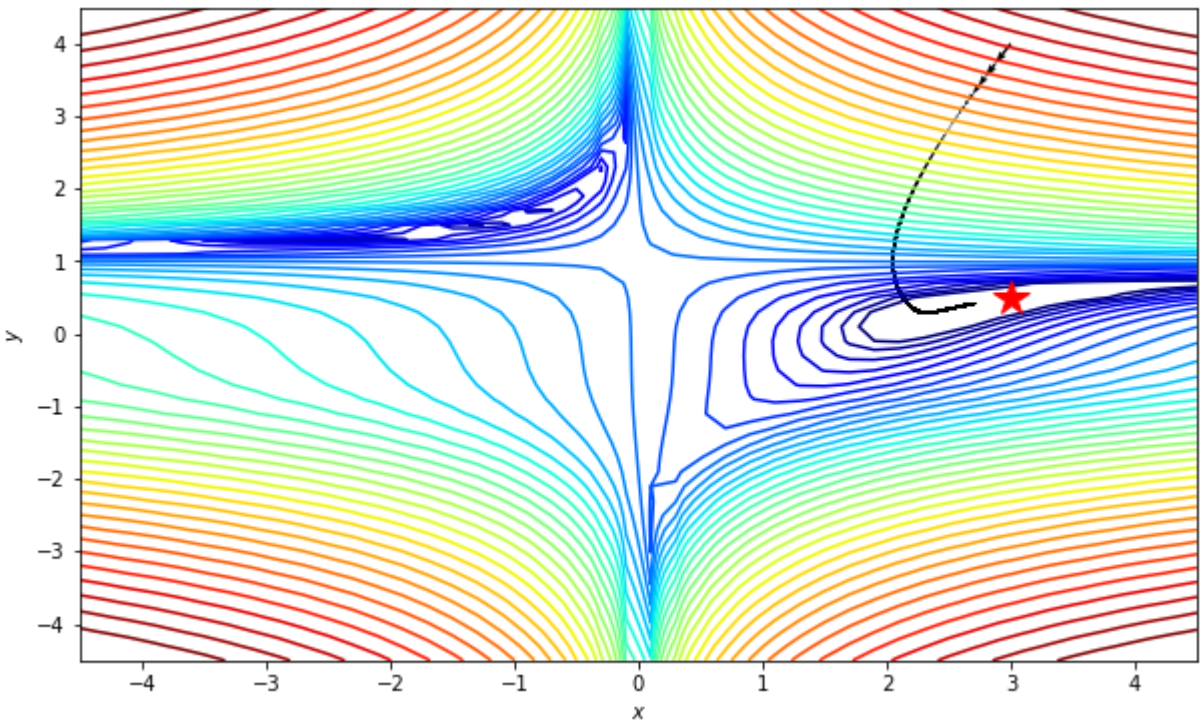

Figure 2-14 The RMSprop method can also basically converge

2.3.5 Adam method

In addition to storing an exponentially decaying cumulative mean of the squares of past gradients like the RMSprop method, it also stores a cumulative mean of the gradients like the momentum method. The momentum method can be seen as a ball running down a slope, but the Adam method behaves like a ball with friction and is therefore better suited for flat minima. Use m_t, v_t to represent the moving average of past gradients and gradient squares:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

They are equivalent to the first-order and second-order momentum of the gradient, because their initial value is 0, Adam's author observed: when the decay rate is small, such as β_1, β_2 is close to 1, they are biased towards zero, Especially in the early stages of an iteration. To correct this problem, the authors used the following correction formula:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

Update x based on this:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

```
#https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c
def gradient_descent_Adam(df,x,alpha=0.01,beta_1 = 0.9,beta_2 = 0.999, iterations =
100,epsilon = 1e-8):
    history=[x]
    m = np.zeros_like(x)
    v = np.zeros_like(x)
    for t in range(iterations):
        if np.max(np.abs(df(x)))<epsilon:
            print("The gradient is small enough!")
```

```
            break
        grad = df(x)
        m = beta_1*m+(1-beta_1)*grad
        v = beta_2*v+(1-beta_2)*grad**2

        #m_1 = m/(1-beta_1)
        #v_1 = v/(1-beta_2)
        t = t+1
        if True:
            m_1 = m/(1-np.power(beta_1, t+1))
            v_1 = v/(1-np.power(beta_2, t+1))
        else:
            m_1 = m / (1 - np.power(beta_1, t)) + (1 - beta_1) * grad / (1 - np.power(beta_1,
t))
            v_1 = v / (1 - np.power(beta_2, t))

        x = x-alpha*m_1/(np.sqrt(v_1)+epsilon)
        #print(x)
        history.append(x)
    return history
```

For the above problem, execute the gradient descent algorithm gradient_descent_Adam:

```
path = gradient_descent_Adam(df,x0,0.001,0.9,0.8,100000,1e-8)
#path = gradient_descent_Adam(df,x0,0.000005,0.9,0.9999,300000,1e-8)
print(path[-1])
path = np.asarray(path)
#plt.plot(path)
```

```
[2.99999653 0.50000329]
```

```
plot_path(path,x,y,z,minima_,xmin, xmax,ymin, ymax)
```

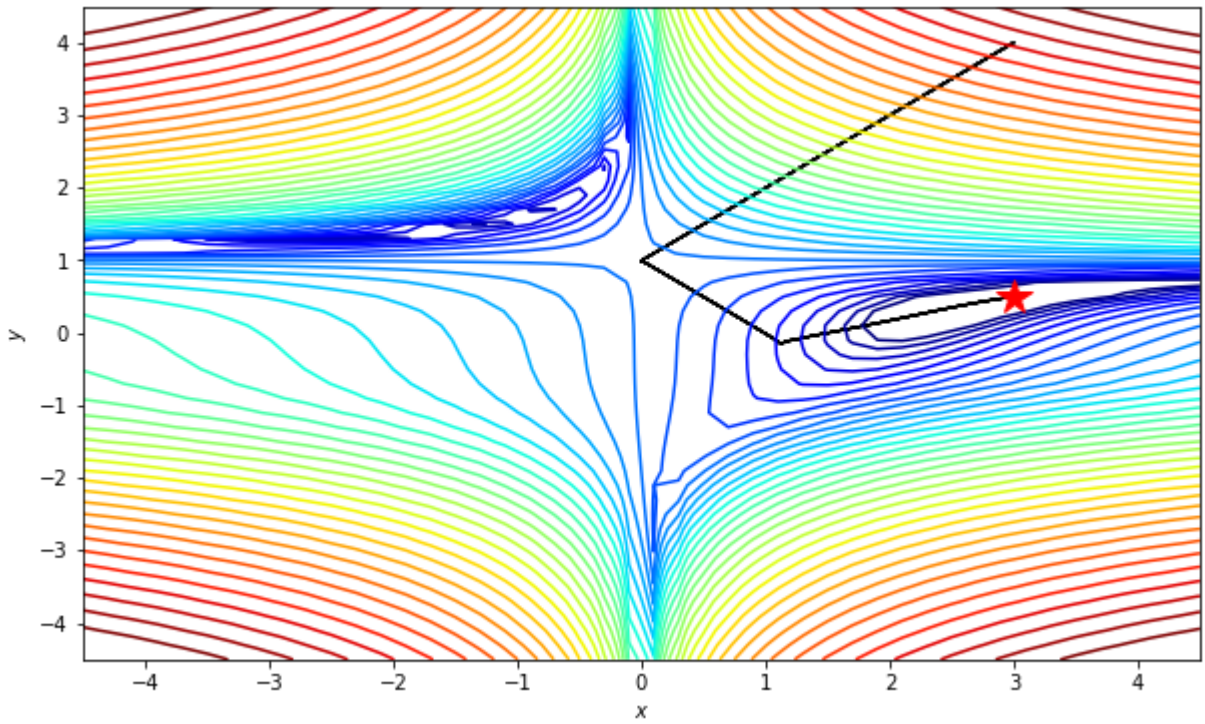

Figure 2-15 The Adam method can also converge to a near optimal solution

2.4 Gradient verification

2.4.1 Comparing numerical and analytical gradients

When writing the code of the gradient descent algorithm, the most likely mistake is that the gradient calculation is incorrect, which leads to the inability of the algorithm to converge. Therefore, in addition to adjusting the learning rate, you should check whether the gradient calculation is correct. To this end, according to the definition of the derivative, that is, the derivative is the rate of change of the function, the derivative (gradient) of the function at a point x can be estimated by the following formula:

$\frac{\partial f(x)}{\partial x} = lim_{\epsilon->0} \frac{f(x+\epsilon)-f(x-\epsilon)}{2\epsilon}$

That is, use the division on the right side of the formula to approximate the derivative (gradient) of $f(x)$ at x. If ϵ is small enough, the derivative (gradient) of this value should be the same as the analytical derivative (gradient) on the left) are close enough.

Therefore, before training the model with the gradient descent method, the numerically calculated gradient and the analytical gradient can be compared to verify that the analytical gradient is calculated correctly.

For example, for the previous binary function $f(x,y) = \frac{1}{16}x^2 + 9y^2$, in the gradient descent method, the function is at a point $x = (x_0, x_1)$ The function values and analytical gradients of are calculated by the following code.

```
f = lambda x: (1/16)*x[0]**2+9*x[1]**2
df = lambda x: np.array( ((1/8)*x[0],18*x[1]))
```

The numerical gradient at the point $x = (x_0, x_1)$ can be calculated as follows:

```
df_approx = lambda x,eps:((f([x[0]+eps,x[1]])-f([x[0]-eps,x[1]]) )/(2*eps),(
f([x[0],x[1]+eps])-f([x[0],x[1]-eps]) )/(2*eps))
```

The following code snippet compares the errors of the analytical and numerical gradients at the point $x = [2., 3.]$:

```
x = [2.,3.]
eps = 1e-8
grad = df(x)
grad_approx = df_approx(x,eps)
print(grad)
print(grad_approx)
print(abs(grad-grad_approx))
```

```
[ 0.25 54. ]
    (0.2500001983207767, 54.00000020472362)
    [1.98320777e-07 2.04723619e-07]
```

It can be seen that as long as the small increment eps of calculating the numerical gradient is small enough, this numerical gradient is close enough to the analytical gradient, and this is the definition of the derivative: the numerical gradient can be close enough to the analytical gradient. If it is found that the error of the two is relatively large or large, it means that there may be a problem with the calculation of the analytical gradient or function value or numerical gradient. Most of the errors are problems with the calculation of the analytical gradient or function value.

Before using the gradient descent method to solve the optimal solution, the gradient verification method should be used to ensure that the calculation of the analysis gradient and function value is correct. On this basis, adjust the hyperparameters of the gradient descent method such as learning rate or momentum parameters.

2.4.2 Generic numerical gradients

Machine learning includes the hypothetical function in deep learning that contains a lot of parameters, and a general numerical gradient calculation function can be written:

```
def numerical_gradient(f, params, eps = 1e-6):
    numerical_grads = []
    for x in params:
        # x may be a multidimensional array,
        # for each element, calculate its numerical partial derivative
        grad = np.zeros(x.shape)
        it = np. nditer(x, flags=['multi_index'], op_flags=['readwrite']) #
        while not it. finished:
            idx = it.multi_index
            old_value = x[idx]
            x[idx] = old_value + eps    # x[idx]+eps
            fx = f()
            x[idx] = old_value - eps    # x[idx] - eps
            fx_ = f()
            grad[idx] = (fx - fx_) / (2*eps)
            x[idx] = old_value    # Note: Be sure to restore the weight parameter to its
original value.
            it.iternext()         # Loop through the next element of x

        numerical_grads.append(grad)
    return numerical_grads
```

The parameter f accepted by this function indicates the function to calculate the gradient, and params indicates the parameters of the function, because f may have multiple parameters, and params indicates a set of these multiple parameters (such as python's list, tuple, etc. object). To be more general, assume that each element x of params is a multidimensional array containing multiple elements.

In the inner loop, for the element $x[idx]$ pointed to by each subscript idx of x, add a small increment $x[idx] + eps$ and $x[idx] - eps$ respectively and calculate the corresponding The function value f(), and then use the differential approximation formula of the derivative to calculate the partial derivative corresponding to this $x[idx]$ and assign it to $grad[idx]$. Note: After each modification of $x[idx]$, it must be restored to the original value, otherwise it will affect the calculation of other partial derivatives and affect the value of params after exiting this function.

You can use this general numerical gradient computation function to compute the numerical gradient of the previous function:

```
x = np.array([2.,3.])
param = np.array(x)        # The parameter param of numerical_gradient must be a numpy array
numerical_grads = numerical_gradient(lambda:f(param),[param],1e-6)
print(numerical_grads[0])
```

```
[ 0.25 54.00000001]
```

Note that the first parameter f of numerical_gradient must point to a function object rather than the result of a function call. It is wrong to write `lambda:f(param)` above as `f(param)` .

For a function f that contains some parameters such as param, usually the above lambda expression or the following wrapper function fun can be used to return a function object that performs calculations on the parameter param.

```
def fun():
    return f(param)

numerical_grads = numerical_gradient(fun,[param],1e-6)
print(numerical_grads[0])
```

```
[ 0.25 54.00000001]
```

In the following chapters, this general numerical gradient calculation function numerical_gradient() will be used to calculate the numerical gradient of the model function. This function and others are included in the book's source code file util.py.

2.5 Separation gradient descent algorithm and parameter optimization strategy

2.5.1 Parameter optimizer

The optimization strategy of variables (parameters) is hard-coded in the gradient descent algorithm. The gradient descent method of different optimization strategies has the same framework except for the parameter update. In order to improve code reusability and flexibility, parameter optimization strategies can be classified from gradient descent algorithms.

A class representing a parameter optimization strategy can be defined:

```
class Optimizator:
    def __init__(self,params):
        self.params = params

    def step(self,grads):
       pass
    def parameters(self):
        return self.params
```

params is a list of variables (parameters), and step() is used to update these parameters params according to the gradient grads. For example, the parameter optimizer class SGD that defines the parameter optimization strategy using the basic gradient descent method can be derived on the basis of this class:

```
class SGD(Optimizator):
    def __init__(self,params,learning_rate):
        super().__init__(params)
        self.lr = learning_rate

    def step(self,grads):
        for i in range(len(self.params)):
            self.params[i] -= self.lr*grads[i]
        return self.params
```

Similarly, other parameter optimizers can be defined, such as SGD_Momentum of the momentum method:

```
class SGD_Momentum(Optimizator):
    def __init__(self,params,learning_rate,gamma):
        super().__init__(params)
        self.lr = learning_rate
        self.gamma= gamma
        self.v = []
        for param in params:
            self.v.append(np.zeros_like(param) )

    def step(self,grads):
        for i in range(len(self.params)):
            self.v[i] = self.gamma*self.v[i]+self.lr* grads[i]
            self.params[i] -= self.v[i]
        return self.params
```

2.5.2 Gradient descent method accepting parameter optimizer

As long as the gradient descent algorithm accepts the parameter optimizer that updates the parameters, it can update the parameters according to the optimization strategy of the optimizer:

```
def gradient_descent_(df,optimizator,iterations,epsilon = 1e-8):
    x, = optimizator.parameters()
    x = x.copy()
    history=[x]
    for i in range(iterations):
        if np.max(np.abs(df(x)))<epsilon:
            print("The gradient is small enough!")
            break
        grad = df(x)
        x, = optimizator.step([grad])
        x = x.copy()
        history.append(x)
    return history
```

Looking at a simple convex function surface,

$$f(x,y)=\frac{1}{16}x^2+9y^2$$

This is a bowl-shaped surface, as shown in Figure 2-16. Its minimum value is at the bottom of the bowl, that is, (0,0) is the minimum value point of the entire function, and the minimum value is 0.

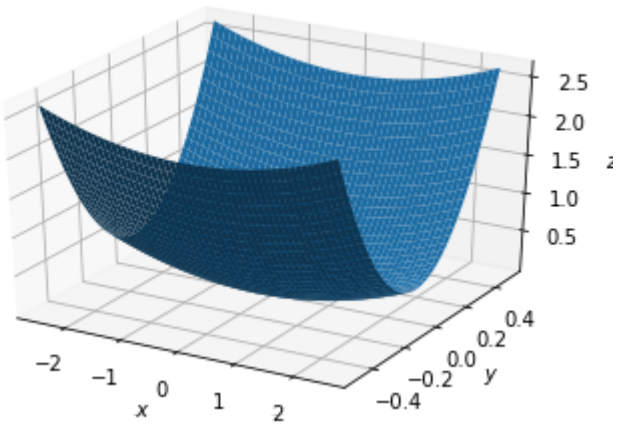

Figure 2-16 Function $f(x,y)=\frac{1}{16}x^2+9y^2$ surface

To this function, apply the SGD parameter optimizer described above:

```
df = lambda x: np.array( ((1/8)*x[0],18*x[1]))
x0=np.array([-2.4, 0.2])

optimizator = SGD([x0],0.1)
path = gradient_descent_(df,optimizator,100)
print(path[-1])
path = np.asarray(path)
path = path.transpose()
```

```
    [-8.26638332e-06 2.46046384e-98]
```

Approaching the optimal solution, switch to the SGD_Momentum optimizer:

```
x0=np.array([-2.4, 0.2])
optimizator = SGD_Momentum([x0],0.1,0.8)
path = gradient_descent_(df,optimizator,1000)
print(path[-1])
path = np.asarray(path)
path = path.transpose()
```

It also better approximates the optimal solution.

```
    The gradient is small enough!
    [-1.49829905e-08 -4.74284398e-10]
```