第2章 梯度下降法

深度學習的核心任務就是通過樣本數據訓練一個函數模型，或者說找到一個最佳的函數表示或刻畫這些樣本數據。求解最佳的函數模型歸結為一個數學優化問題，更準確地說求某種損失函數的最值（極值）問題。深度學習中都是用**梯度下降法**求解這個最值問題或者說求解模型參數的。

本章從函數極值的必要條件出發介紹了梯度下降法算法的理論根據、算法原理與代碼實現，並介紹了梯度下降法中對求解變量（參數）進行更新的不同優化策略。

2.1 函數極值的必要條件

函數$y=f(x)$在某個點x_0處取得**極小值**：是指存在某個正數ϵ，使得對於區間$(x_0-\epsilon,x_0+\epsilon)$的每個x都滿足$f(x_0)\leq f(x)$。$x_0$稱為函數的**極小值點**，而$f(x_0)$稱為函數的**極小值**。

函數$y=f(x)$在某個點x_0處取得**極大值**：是指存在某個正數ϵ，使得對於區間$(x_0-\epsilon,x_0+\epsilon)$的每個x都滿足$f(x)\leq f(x_0)$。$x_0$稱為函數的**極大值點**，而$f(x_0)$稱為函數的**極大值**。

極小值和極大值統稱為**極值**，極小值點和極大值點統稱為**極值點**。

如果對函數$f(x)$的定義域的所有x都滿足$f(x_0)\leq f(x)$，則x_0稱為函數的**最小值點**，而$f(x_0)$稱為函數的**最小值**。

如果對函數$f(x)$的定義域的所有x都滿足$f(x)\leq f(x_0)$，則x_0稱為函數的**最大值點**，而$f(x_0)$稱為函數的**最大值**。

即最小值是一個全局範圍的極小值，最大值是一個全局範圍的極大值。最小值和最大值統稱為**最值**，最小值點和最大值點統稱為**最值點**。

函數極值的必要條件：如果x_0是函數$f(x)$的極值點，且函數在x_0可導，則必有$f'(x_0)=0$即極值點處的導數值必然為0。

例如前面的函數$f(x)=x^2$在$x=0$取得最小值（當然也是極小值）且可導，因此在$x=0$其導數值$f'(0)=2\times 0=0$必是0。

這個命題很容易證明，假如x_0是函數$f(x)$的極值點，即存在區間$(x_0-\epsilon,x_0+\epsilon)$滿足$f(x_0)\leq f(x)$，因此$f(x)-f(x_0)\geq 0$，而：

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

當x從左右兩邊趨向於x_0是，Δx分別是負數和正數，而分子總是正數，從x從右邊趨向於x_0，其極限值應該≥ 0，從x從左邊趨向於x_0，其極限值應該≤ 0，而這個極限是存在的，因此，其值只能是0。

根據極限公式，還可以發現一個規律，如果在x_0導數是正數，說明在這點附近函數f(x)是單調遞增的，即如果$x_1<x_2$，則$f(x_1)<f(x_2)$，即$f(x)$隨著x的增大而增大。或者說Δx如果是正數，那麼 說Δy也是正數。例如$y=f(x)=x^2$的導數是$f'(x)=2x$，當x大於0時，導數都是正數，因此，函數曲線時單挑遞增的，而x小於0時，導數都是負數，因此，函數曲線是單挑遞減的，即如果$x_1<x_2$，反而$f(x_1)>f(x_2)$。

例如，函數$f(x)=x^3-3x^2-9x+2.$，令其導數$f'(x)=0$:

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

可以得到導數為0的兩個點$x_1=-1,\ x_2=3$。該函數和其導函數$f'(x)$的單調變化情況如圖2-1所示：

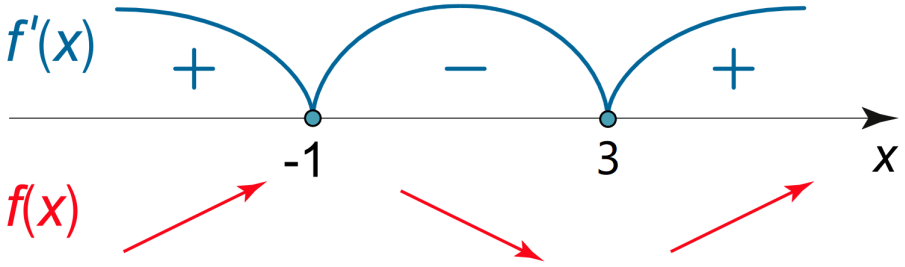

圖 2-1 $f'(x) > 0$則函數單調上升，$f'(x) < 0$則函數單調下降

在區間$(-\infty, -1]$中，$f'(x)$是正數，因此函數f(x)是單調遞增的，在區間$(-1, 3)$中，$f'(x)$是負數，因此函數f(x)是單調遞減的，在區間$[3, \infty)$中，$f'(x)$是正數，因此函數f(x)是單調遞增的。

下面代碼繪製了這個函數和其導函數的曲線，可以更直觀地看出單調變化和極值點情況。

```
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()
```

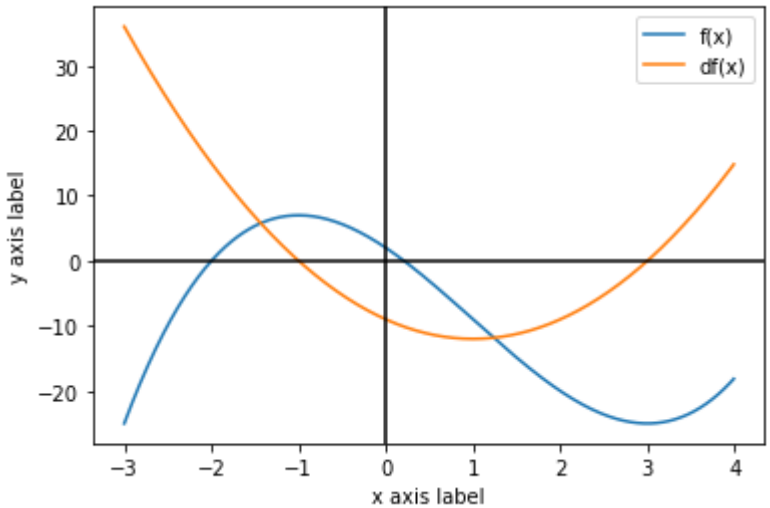

圖 2-2 $f(x) = x^3 - 3x^2 - 9x + 2$及其導函數$f'(x)$的函數曲線

注意，上述命題只是說明了函數極值點處的必要條件，但不是充分條件，也就是說，一個函數x_0處的導數$f'(x_0) = 0$並不表示x_0一定是一個極值點。如$f(x) = x^3$在$x = 0$處的導數$f'(0)$也是0，但該點並不是函數的極值點。實際上，這個函數是一個單調遞增的曲線，如圖2-3所示。

```
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()
```

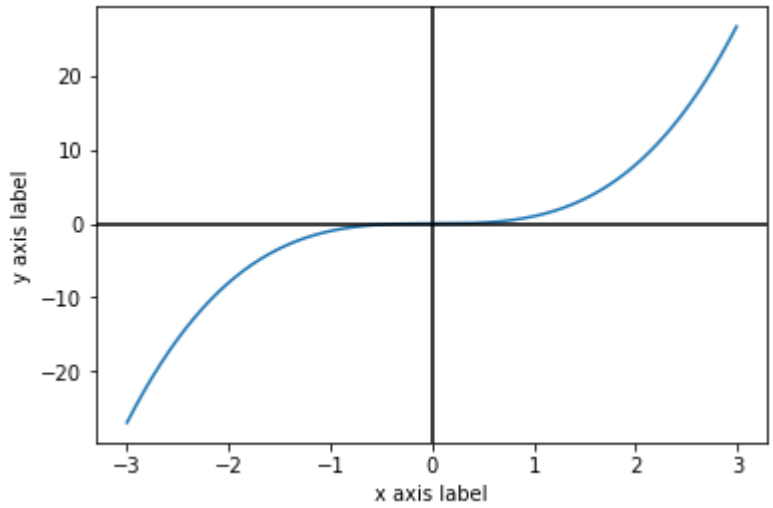

圖 2-3 $f(x) = x^3$的函數曲線

顯然函數極值的必要條件可以推廣到多變量函數，即對一個多變量函數$f(x_1, x_2, \cdots, x_n)$，如果該函數在某點$x^* = (x_1^*, x_2^*, \cdots, x_n^*)$取得極值且該點處的梯度存在（即所有偏導數存在），那麼該點處的梯度必然為0（即每個偏導數值都是0）。即：

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

2.2 梯度下降法(gradient descent)

對於一個一元函數f(x)，如果在某點x附近有微小的變化Δx，則f(x)的變化$f(x + \Delta x) - f(x)$可表示成如下的微分形式：

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

即在x附近，如果Δx和$f'(x)$符號相同，那麼$f'(x)\Delta x$即$f(x + \Delta x) - f(x)$就是正數，如果Δx和$f'(x)$符號相反，那麼$f'(x)\Delta x$即$f(x + \Delta x) - f(x)$就是負數。如取$\Delta x = -\alpha f'(x)$（其中α是一個小的正數），那麼$f(x + \Delta x) - f(x) = -\alpha f'(x)^2$就是負數，即$f(x + \Delta x)$的值會比$f(x)$更小。或者說x沿著$f'(x)$的反方向$-f'(x)$運動$\Delta x$，其函數值$f(x + \Delta x)$比原來的$f(x)$更小。

如圖2.4，函數$f(x) = x^2 + 0.2$在$x = 1.5$的函數值$f(x)$是2.45，導數值$f'(x)$是3.0，是一個正數，在$f(x)$的定義域即x軸上指向x軸的正向，如圖中長箭頭所示。

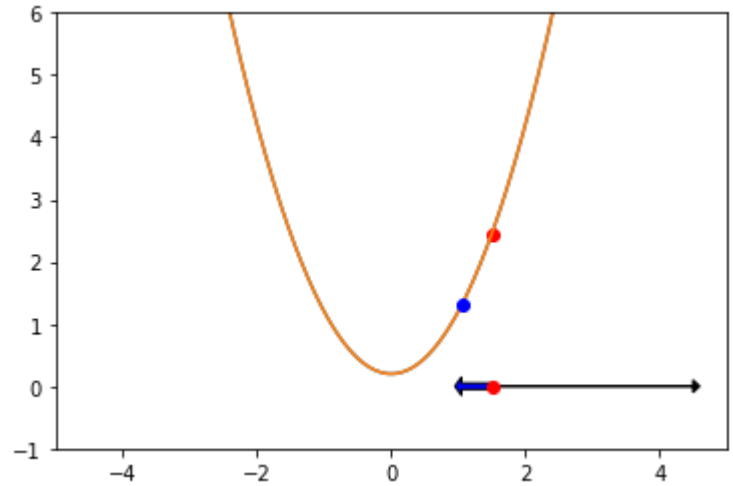

圖 2-4 $f'(1.5) = 2.45 > 0$，Δx和$f'(x)$符號相同移動則函數值增加，相反則函數值減少

令$\alpha = 0.15$，$\Delta x = -\alpha f'(x) = -0.449$，將x沿著這個$\Delta x$（如圖中藍色箭頭方向）移到到$x_{new} = x + \Delta x = 1.05$，得到的新的$x = 1.05$處的f(1.05)函數值是1.3025，即圖中曲線上藍色點的y坐標值。因為Δx和$f'(x)$方向相反（一負一正），這個f(1.05)肯定小於原先的f(1.5)。

只要不斷地重複這個過程，即將x沿著其導數$f'(x)$的反方向（$-f'(x)$）移動一個微小的增量$-\alpha f'(x)$就能到達一個新的$x_{new} = x - \alpha f'(x)$，這個新的$x_{new}$的函數值$f'(x_{new})$肯定小於之前的函數值$f'(x)$。隨著x不斷接近最小值點的x值，導數f'(x)也接近0（因為函數極值點x^*的導數$f(x^*) = 0$），x移動的增量Δx越來越接近於0。

這就是**梯度下降法**的思想，即從一個初始的x出發，不斷用下面的公式更新x的值：

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

對當前的x，沿著其負的導數（梯度）方向（即$-f'(x)$）移動x，就能使$f(x)$不斷變小。理想情況下，達到最小值$f(x)$的x，此時$f'(x) = 0$。再迭代更新x，x的值將不再變化。如圖2-5所示，x不斷迭代更新，從而不斷接近極值點。

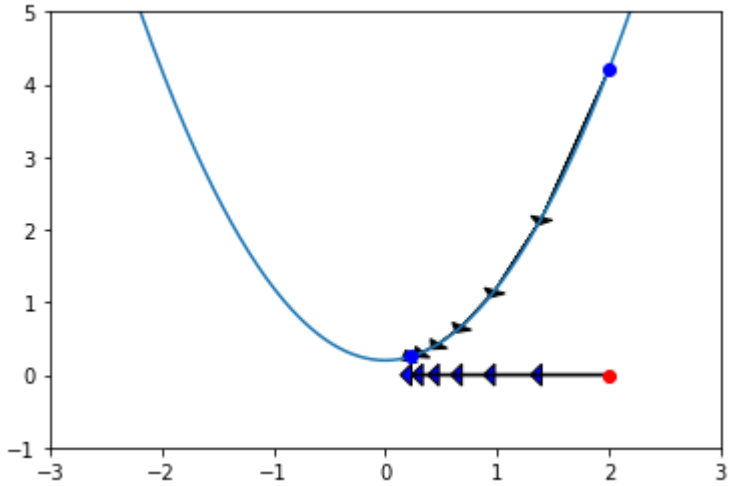

圖 2-5 x沿著$-f'(x)$運動，函數值不斷減少

當然這個移動的步伐（即$-\alpha f'(x)$）不能太大，因為根據導數的定義，上述近似公式只適用於x附近。如果移動步伐太大，可能跳過最優值的x，使得x的值不斷地來回震盪。如圖2-6所示。

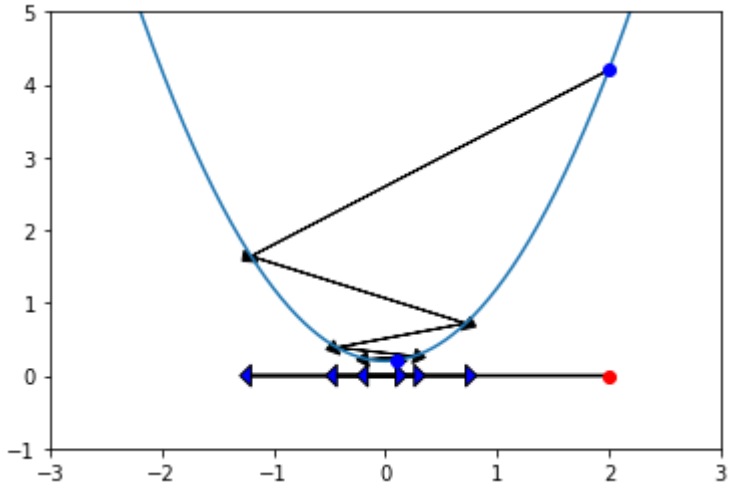

圖 2-6 x改變的幅度$-\alpha f'(x)$太大，函數值會振盪

梯度下降法是求一個逼近的最優解，為了避免一直迭代下去，可以用下列方法檢驗是否足夠逼近最優解：

- 導數（梯度）$f'(x)$的絕對值已經足夠小。
- 迭代次數達到了預先設定的最大迭代次數。

下面是梯度下降法的代碼，其中的參數df用於計算某個函數$f(x)$的導數$f'(x)$，x是變量的初始值，alpha是學習率，iterations表示迭代次數，epsilon檢查df=$f'(x)$的值是否接近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("梯度足够小！")
            break
        x = x-alpha* df(x)
        history.append(x)
    return history
```

這個梯度下降法函數將迭代過程中的所有更新的x都保存在一個python的list對象history裡，並返回這個對象。

對於上面的函數$f(x) = x^3–3x^2–9x + 2$，其導數$f'(x) = 3x^2–6x–9$。假如要求$x = 1$附近函數$f(x)$的極小值，可以調用這個函數gradient_descent()：

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

```
    梯度足够小！
    2.999999999256501
```

得到了$f(x)$的極值點x=2.999999999256501。可以將迭代過程中的x對應的曲線上的點繪製出來：

```
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()
```

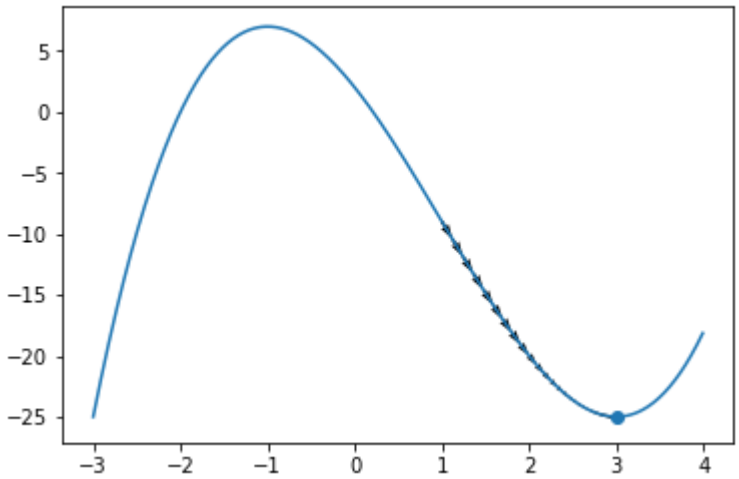

圖 2-7 x逐漸收斂到最小值點

其中，matplotlib的 quiver函數可以用箭頭繪製速度向量，其函數格式是：

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

其中X,Y是1D或2D數組，表示箭頭的位置，而U, V也是同樣的1D或2D數組，表示箭頭的速度（向量）。其他參數請查詢官方文檔。

對於多變量函數，梯度下降法的原理是一樣的，只不過用梯度代替了導數。

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

下面是wiki百科的 Beale's function函數。

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

這個函數的全局極小值是$(3,0.5)$。可以用下面的python代碼計算函數值:

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

為了繪製這個曲面，先在x、y軸上取一些均勻分佈的坐標值:

```
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)
```

然後用np.meshgrid()函數根據上述的x_list和y_list，得到它們交叉處的網格點(x,y)，併計算出這些網格坐標點對應的函數值:

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

最後可以調用plot_surface()函數繪製這個曲面:

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

完整代碼如下:

```
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()
```

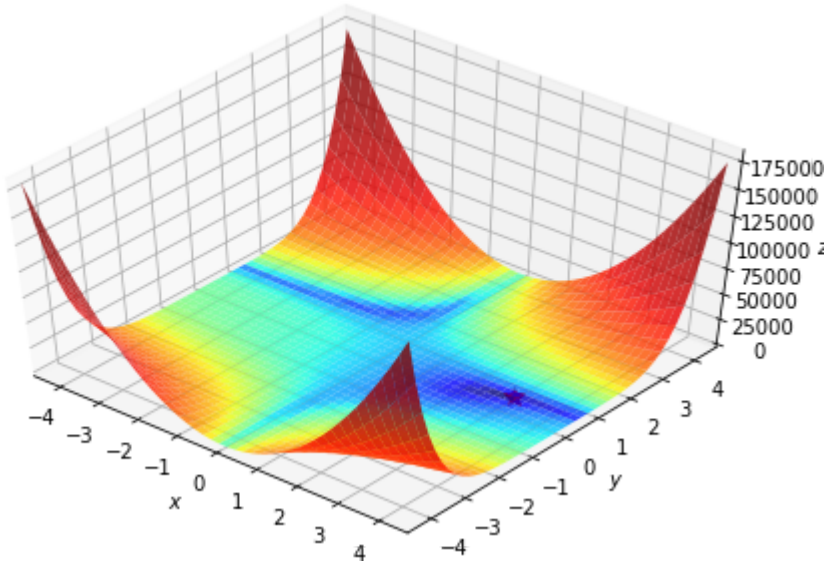

圖 2-8 繪製的f(x,y)曲面

$f(x,y)$關於x,y的偏導數是:

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

可以用matplotlib的quiver函數在2D坐標平面上繪製出這些網格點處的梯度方向。

```
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()
```

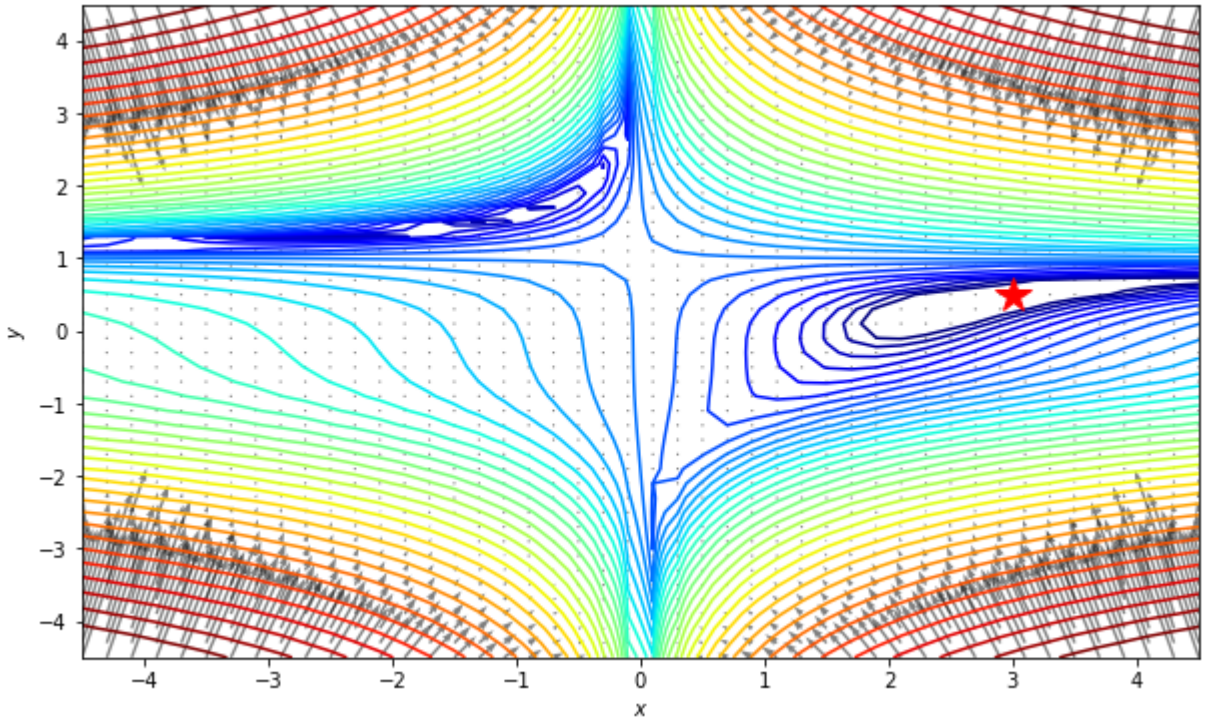

圖 2-9 函數f(x,y)的等值面的定義域坐標輪廓和網格點處的梯度方向

為了直接利用前面的梯度下降法代碼，可以將之前梯度下降法代碼中的x用一個numpy向量表示，將

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

修改為：

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

先將分離的x、y坐標數組組合為一個數組:

```
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)

可以定義一個針對這個向量化的坐標點x的梯度函數df，下面代碼還給出了修改後的向量化版本梯度下降算法實現：

```
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("梯度足够小！")
            break
        x = x-alpha* df(x)
        history.append(x)
    return history
```

下面代碼從x0=(3., 4.)出發求解這個曲面的極值點：

```
x0=np.array([3., 4.])
print("初始点",x0,"的梯度",df(x0))

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

```
    初始点 [3. 4.] 的梯度 [25625.25 57519.  ]
    极值点： [2.70735828 0.41689171]
```

因為初始x的梯度值開始很大，學習率α必須取很小的數（如0.000005），不然會導致震盪或無窮大的值，最後收斂到`[2.70735828 0.41689171]`，但還不是最優點，可以通過繪製迭代過程中的x的變化情況，更直觀地看清楚這種情況。

```
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)
```

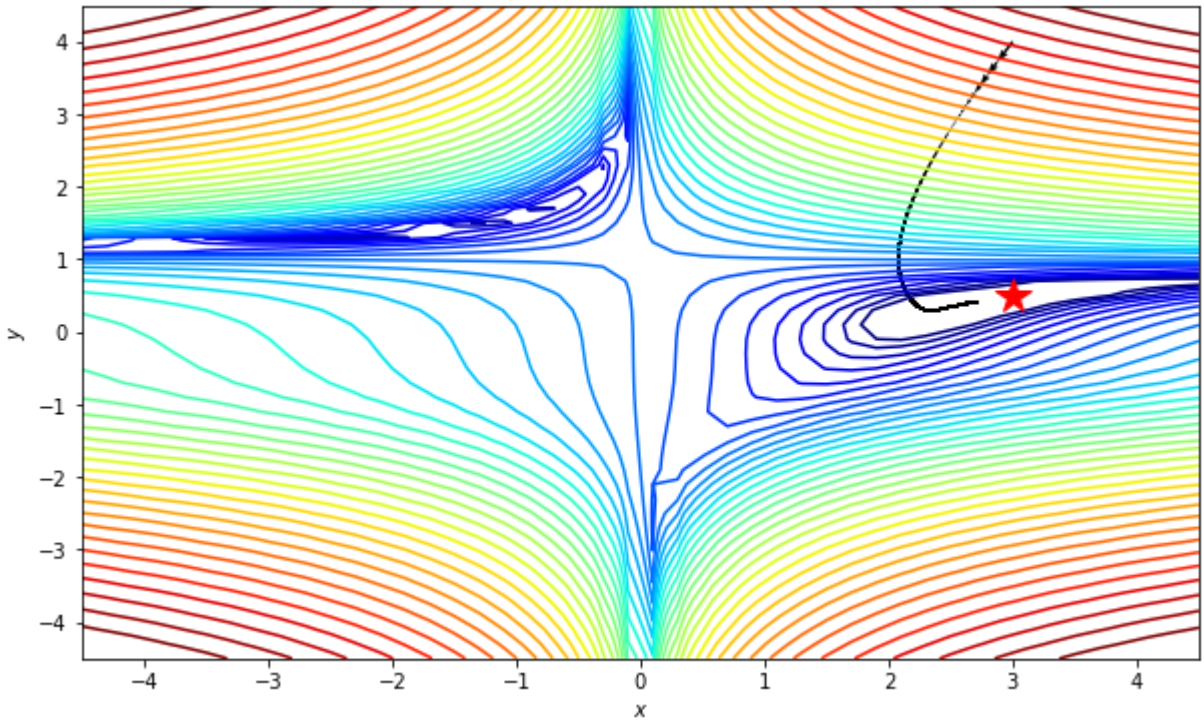

圖 2-10 迭代過程中，梯度值變得越來越小，收斂越來越慢

迭代過程中，梯度值變得越來越小，同樣的學習率使得x的更新變得非常緩慢，即使迭代了10萬次，仍然未能逼近最優解。一種自然的辦法是採用自適應的學習率，即當梯度變得很小時，增加學習率。作為練習，讀者可以嘗試修改梯度下降算法，以便更好更快地得到最優解。

2.3 梯度下降法的參數優化策略

基本的梯度下降算法中的學習率是一個固定的值，而迭代過程中梯度大小是不斷變化的，學習率過大會導致待求解變量來回震盪，學習率過小，使收斂非常緩慢甚至停滯，即使最初的學習率適中，但隨著收斂接近最優解，其梯度也接近0，也會造成停滯。很自然的，學習率應該在迭代過程中，隨著逐漸收斂而進行調整，即在迭代過程中用一個可以變化的學習率來更新待求解變量x。

為了保證能更好更快地逼近最優解，人們提出了許多對梯度下降法的改進，這些改進都是用變化的學習率或策略對求解變量（也稱為參數）進行更新。對變量（參數）的更新策略或方法有：Momentum、Nesterov accelerated gradient、Adagrad、Adadelta、RMSprop、Adam、AdaMax、Nadam、AMSGrad等。

需要說明的是，函數可能是一個多變量的函數，因此，其變量x可以是多個值構成的向量$\boldsymbol{x}$。下面僅對其中一些常用的優化策略進行說明。

2.3.1 Momentum動量法

梯度下降法每次用學習率α和梯度的負方向即$-\alpha\nabla f(x)$更新x，即更新x的向量$-\alpha\nabla f(x)$完全取決於當前計算的梯度，而Momentum動量方法更新x的向量不但考慮當前的梯度，還考慮上次更新向量，即認為更新的向量具有慣性。假設v_{t-1}是前一次用於更新的向量，則當前更新的向量為：

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

用這個v更新x:

$$x = x - v_t$$

這個用於更新x的向量稱為**動量**。動量法將更新向量看成是一個運動物體的速度，而速度是有慣性的。由於結合了之前的更新向量和當前的梯度，它緩解了不同時刻梯度的劇烈變化，使得更新的向量更加光滑，即保持了之前運動的慣性，使得在梯度小的地方，仍然具有較大的運動速度，又不會因梯度的突然變大也過衝。該方法如同一個具有重量的小球向坡下滾動，在尋找最陡下降路徑的同時還保持了一定的慣性。而普通的梯度下降只是根據陡峭程度決定運動速度，如同陡的地方沖得快、平坦的地方幾乎不動。

v的初始值为0，用python代碼可表示为：

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

即v是和x一樣形狀的初始值為0的張量。在迭代過程中，先更新v，再更新函數的參數x：

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

下面是基於動量法的梯度下降法：

```
def gradient_descent_momentum(df,x,alpha=0.01,gamma = 0.8, iterations = 100,epsilon = 1e-6):
    history=[x]
    v= np.zeros_like(x)            #动量
    for i in range(iterations):
        if np.max(np.abs(df(x)))<epsilon:
            print("梯度足够小！")
            break
        v = gamma*v+alpha* df(x)  #更新动量
        x = x-v                   #更新变量（参数）

        history.append(x)
    return history
```

用這個動量法的梯度下降法求解上述問題：

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

[2.96324633 0.49067782]

可以看到，動量法的解已經非常接近最優解了。如圖所示。

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

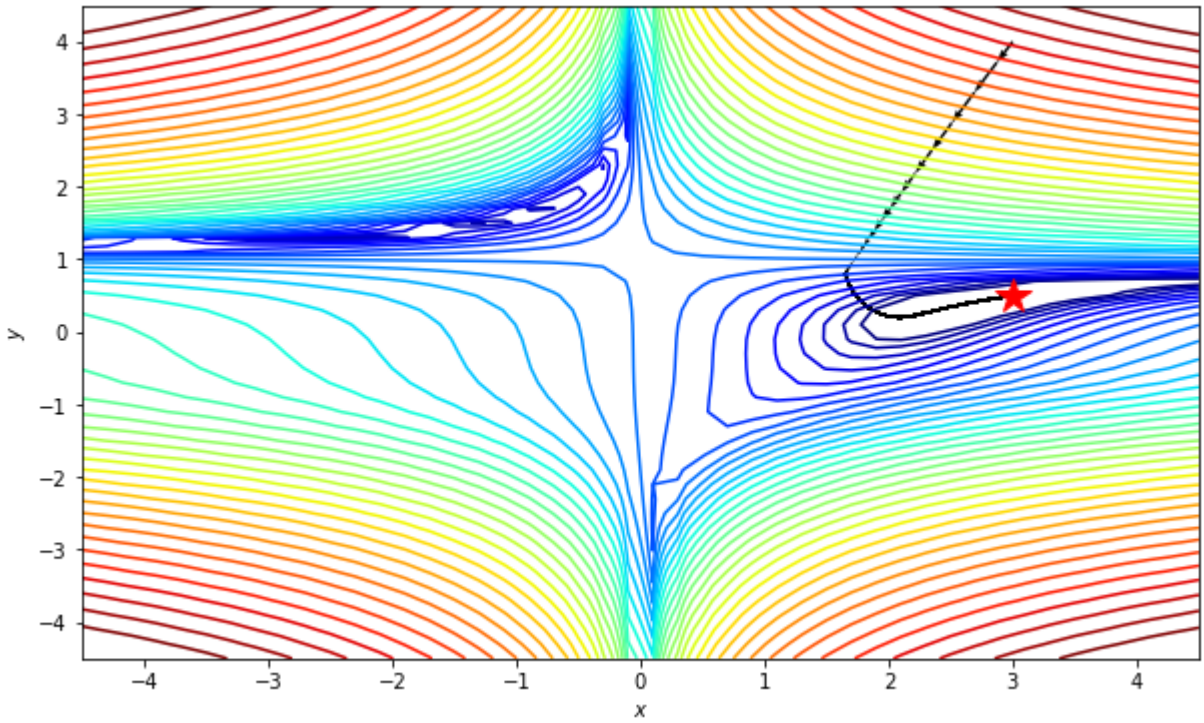

圖 2-11 動量法很快收斂到接近最優解

2.3.2 Adagrad法

根據梯度下降法的變量更新公式$\boldsymbol{x} = \boldsymbol{x} - \alpha\nabla f(\boldsymbol{x})$，影響變量更新的是學習率和梯度的乘積$\alpha\nabla f(\boldsymbol{x})$，梯度的過大或過小和學習率的過大過小一樣會影響算法的收斂。

對於一個多變量函數，每個變量的偏導數的大小可能會相差很大。如2個變量的函數$f(x_1, x_2)$在某點(x_1, x_2)的偏導數$\frac{\partial f}{\partial x_1}$, $\frac{\partial f}{\partial x_2}$數值的絕對值可能相差很大。
對它們採用同一個學習率是不合適的，對一個分量合適的學習率對另一個分量來說反而過大或過小，從而造成震盪和停滯。即直接用下式更新是不合適的：

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

Adagrad法從名詞可翻譯為“自適應(ada)梯度(grad)”，它將每個梯度分量除以該梯度分量的歷史累加值，從而可以消除不同分量的梯度大小不均衡的問題。對於2個分量(x_1, x_2)，如果分別計算出各自分量的歷史累加值(G_1, G_2)，則2個分量的更新公式為：

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

用記號$g_{t,i} = \nabla_\theta f(x_{t,i})$表示第t輪迭代中分量$x_i$的偏導數$\frac{\partial f}{\partial x_i}$，從`t'=1`到`t'=t`的所有輪的該分量梯度可以計算如下的累加和：

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

將$g_{t,i}$除以$G_{t,i}$去更新該分量：

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

為了防止除數為0，可在這個分母上增加一個很小的正數ϵ，從而AdaGrad的參數更新公式為：

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

對比基本的參數更新公式：

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

可以看出，AdaGrad法消除了分量梯度大小的不均衡問題。 AdaGrad的參數更新公式可寫成向量形式：

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

可用初始值為0的變量gl記錄累積的G_t^2，迭代每一輪中，AdaGrad的參數更新的python代碼如下：

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

AdaGrad法的主要優點是消除了不同梯度值大小差異的影響，從而可以將學習率設置為一個固定的值而不需要在迭代過程中不斷調整學習率，一般學習率設置為0.01。 AdaGrad法的主要缺點是隨著迭代過程，累加和$\sum_{t'=1}^{t} \boldsymbol{g}_{t'}^2$會越來越大，因為其中每一項都是正數。這會導致學習變得很慢，甚至停滯。另外，使每個分量梯度具有一致的步伐可能不復合實際情況，會使前進方向偏離最佳解的方向。

基於Adagrad參數更新法的梯度下降法代碼如下：

```
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("梯度足够小！")
            break
        grad = df(x)
        gl += grad**2
        x = x-alpha* grad/(np.sqrt(gl)+epsilon)
        history.append(x)
    return history
```

對上述問題，執行梯度下降算法：

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

```
    [-0.69240717  1.76233766]
```

可以看到，因為分量梯度的均衡化，使得變量更新的前進方向偏離了最佳解的方法，而收斂到了另外的局部最優解。

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

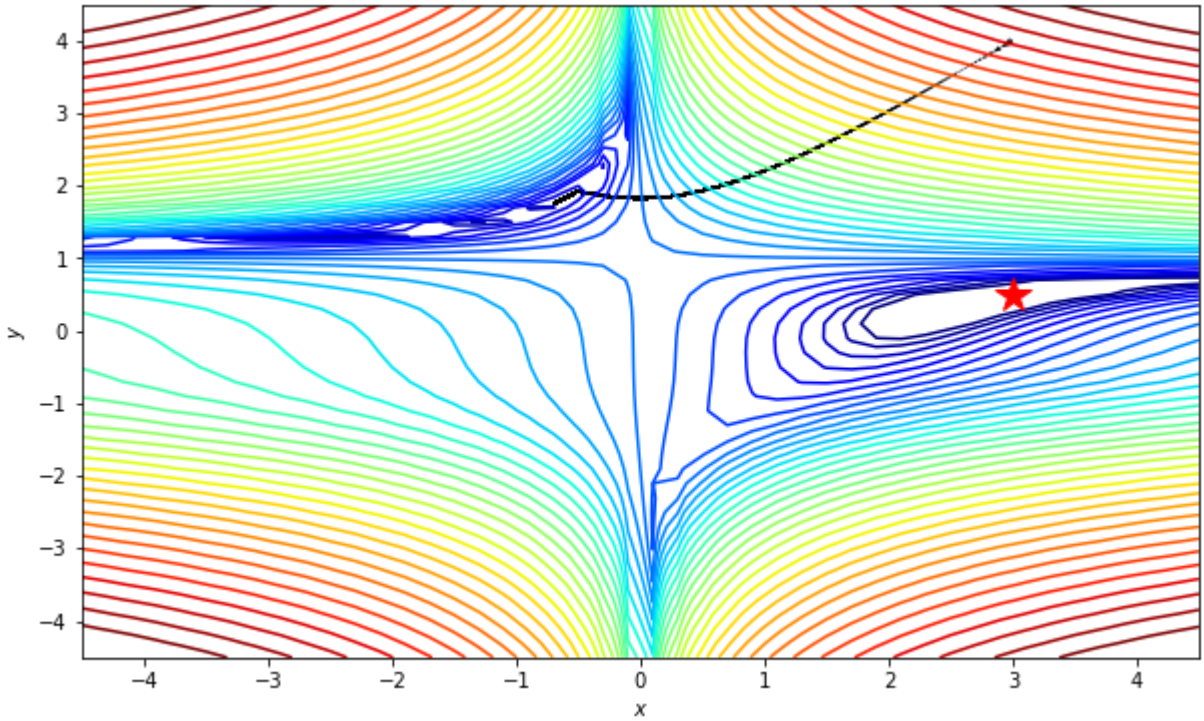

圖 2-12 Adagrad法收斂到另外的局部最小值

2.3.3 Adadelta法

回顧基本的參數更新法，用$\Delta \boldsymbol{x}_t$表示參數的更新向量：

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

AdaGrad法的更新向量是：

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

這裡的$G_t = \sum \boldsymbol{g}_t^2$是$g_t$的歷史平方和，隨著迭代過程，這個值$G_t$越來越大，導致$\Delta \boldsymbol{x}_t$越來越小，從而收斂越來越慢。解決方法是用均方和$E[g^2]_t = \frac{G_t}{t}$而不是平方和代替$G_t$。這個$E[g^2]_t$可以用移動平均方法來計算，即將上一次的平均值和當前的值再做一個平均：

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

Adadelta法更進一步，對更新向量也採用這樣移動平均的方法以便使得更新向量的變化更加平滑。

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

最終的更新向量為：

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

分別用$RMS[\Delta \boldsymbol{x}]_{t-1}, RMS[g]_t$表示$E[\Delta \boldsymbol{x}^2]_{t-1} + \epsilon$和$E[g^2]_t + \epsilon$，則更新向量可表示為：

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

從而參數更新公式為：

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

Adadelta法的python代碼如下：

```
Eg = rho*Eg+(1-rho)*(grad**2)                    #更新梯度的累积平方和
delta = np.sqrt((Edelta+epsilon)/(Eg+epsilon))*grad     # 计算更新向量
x = x- alpha* delta
Edelta = rho*Edelta+(1-rho)*(delta**2)              #更新向量的累积更新
```

Adadelta法的衰減率參數ρ通常設置為0.9，$\Delta \boldsymbol{x}_t, E[\Delta \boldsymbol{x}^2]_t, E[g^2]_t$初始值也為0。基於Adadelta參數更新法的梯度下降法代碼如下：

```
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("梯度足够小！")
            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]
```

可以看到Adadelta法也能收斂到接近最優解。

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

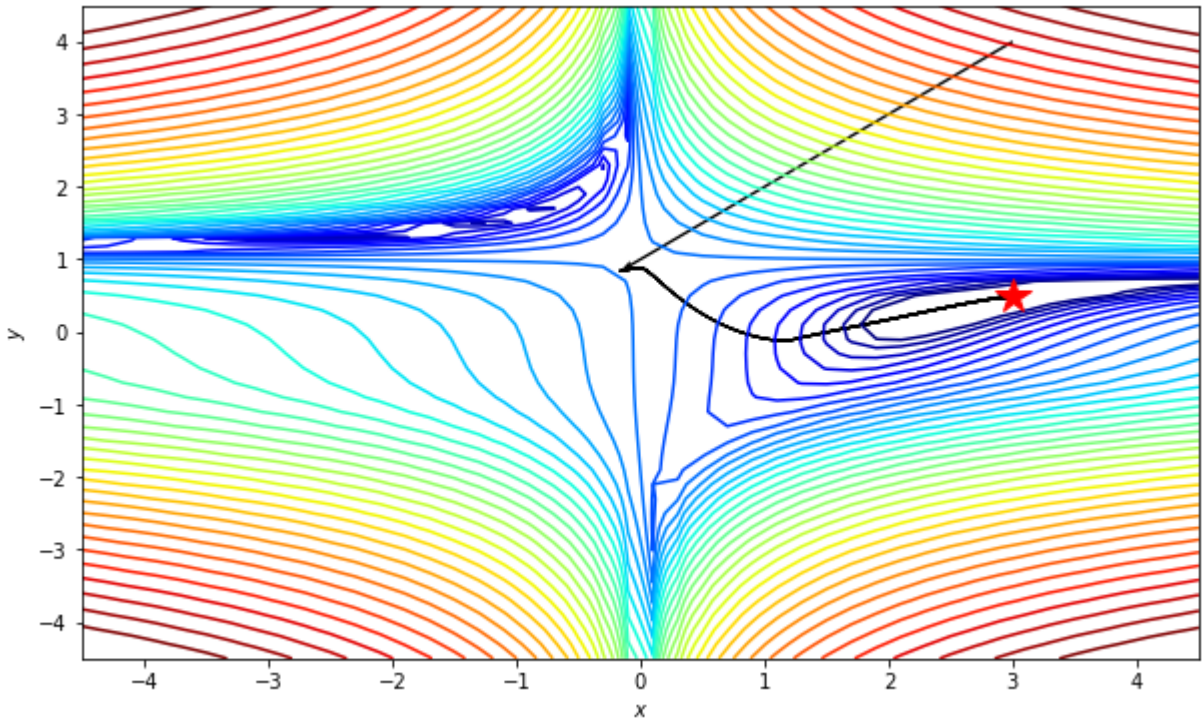

圖 2-13 Adadelta法也能收斂到接近最優解

2.3.4 RMSprop法

和動量法類似，RMSprop採用下列公式更新動量和參數：

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

其思想是將梯度的每個數值都除以長度（數值的絕對值），即轉化為單位長度，從而使得總是以固定步長α更新參數x。為了計算梯度的每個分量的長度，RMSprop類似動量法計算梯度數值的移動平均長度的平方值，即$f(x)^2$。

RMSprop法更新模型參數的python代碼如下：

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

基於RMSprop參數更新法的梯度下降法代碼如下：

```
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("梯度足够小！")
            break
        grad = df(x)
        v = beta*v+(1-beta)*grad**2
        x = x-alpha*grad/(np.sqrt(v)+epsilon)

        history.append(x)
    return history
```

對上述問題，執行梯度下降算法：

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

```
    [2.70162562 0.41500366]
```

模型參數的結果還不夠好，可以增大迭代次數：

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

```
    [2.9082809  0.47616156]
```

可以看到，基本收斂接近了最優解，如圖2-14所示：

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

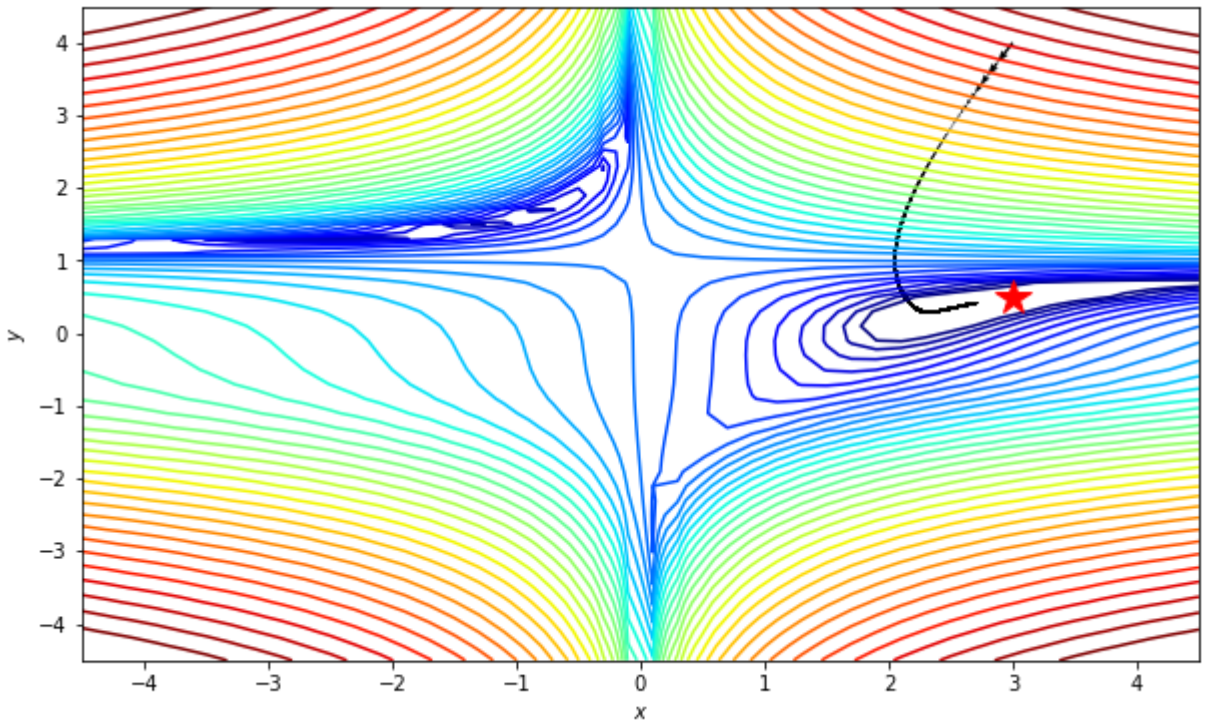

圖 2-14 RMSprop法也基本能收斂

2.3.5 Adam法

除了和RMSprop方法一樣，存儲一個指數衰減的過去梯度的平方的累積平均，還和momentum方法一樣存儲了梯度的累積平均。動量方法可以看作是一個沿著斜坡跑的球，但Adam方法的行為像一個帶有摩擦力的球，因此更適合於平坦的極小值。用m_t, v_t表示過去的梯度和梯度平方的移動平均：

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

它們相當於梯度的一階和二階動量，因為它們的初始值為0，Adam的作者觀察到：當衰減率很小時，如β_1, β_2接近於1時，它們偏向於零，特別是在迭代初期。為了糾正這個問題，作者用了下面的糾正公式：

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

在此基礎上更新x：

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

```
#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("梯度足够小！")
            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
```

對上述問題，執行梯度下降算法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)
```

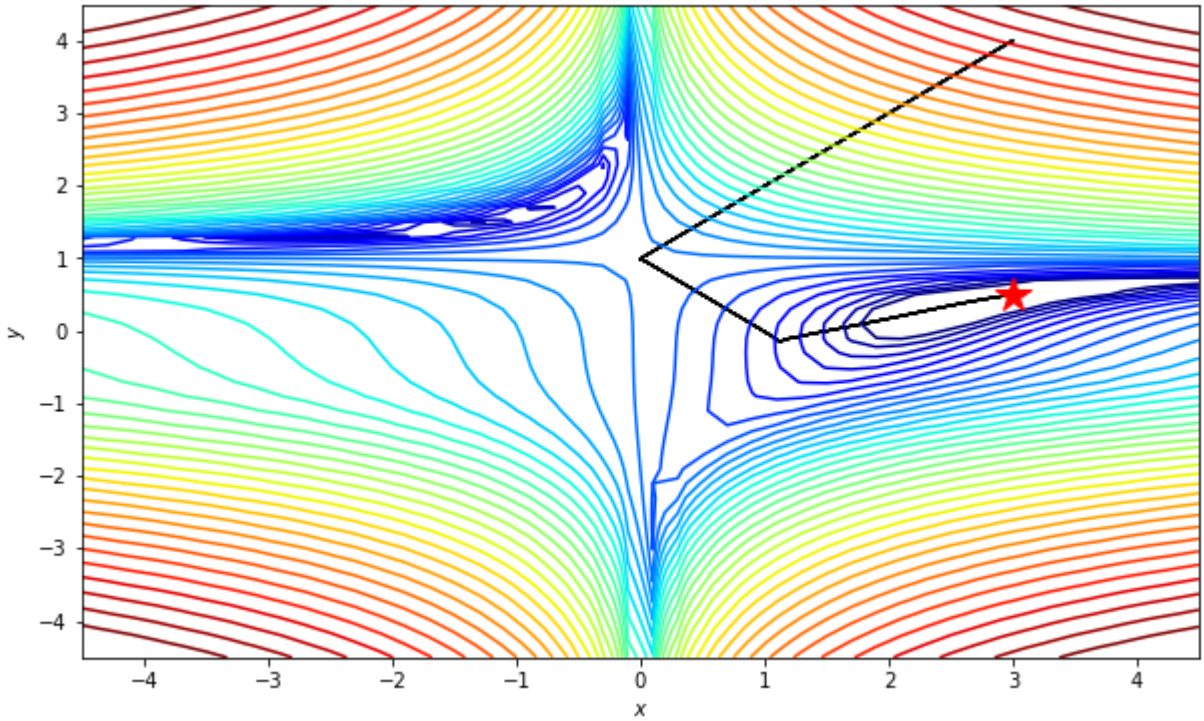

圖 2-15 Adam法也能收斂到接近最優解

2.4 梯度驗證

2.4.1 比較數值梯度和分析梯度

編寫梯度下降算法代碼時，最容易錯的是梯度計算不正確，導致算法無法收斂，因此，除調整學習率外，更應該檢查梯度的計算是否正確。為此，可以根據導數的定義，即導數是函數的變化率，用如下公式估計函數在一點x處的導數（梯度）：

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

即用公式右邊的除式近似表示$f(x)$在x處的的導數（梯度），如果ϵ足夠小，這個數值的導數（梯度）應該和左邊的分析導數（梯度）的值足夠接近。

因此，在梯度下降法訓練模型前，可以先比較數值計算的梯度和分析梯度，以驗證分析梯度的計算是否正確。

例如對於前面的二元函數$f(x,y) = \frac{1}{16}x^2 + 9y^2$，梯度下降法中該函數在一個點$x = (x_0, x_1)$的函數值和分析梯度是通過下面代碼計算的。

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

在該點$x = (x_0, x_1)$的數值梯度可以如下計算：

```
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))
```

下面代碼片段在點$x = [2., 3.]$比較分析和數值梯度的誤差：

```
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]
```

可見只要計算數值梯度的微小增量eps足夠小，這個數值梯度就足夠接近分析梯度，而這正是導數的定義：數值梯度可以足夠逼近分析梯度。如果發現兩者誤差比較大或很大，說明分析梯度或函數值或數值梯度的計算可能有問題，錯誤大多是分析梯度或函數值的計算有問題。

在用梯度下降法求解最優解的之前都應該用梯度驗證的方法來保證分析梯度和函數值的計算是否正確。在此基礎上，再調整梯度下降法的各個超參數如學習率或動量參數等。

2.4.2 通用的數值梯度

機器學習包括後面的深度學習中的假設函數包含非常多的參數，可以寫一個通用的數值梯度計算函數：

```
def numerical_gradient(f,params,eps = 1e-6):
    numerical_grads = []
    for x in params:
        # x可能是一个多维数组，对其每个元素，计算其数值偏导数
        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      #注意：一定要将该权值参数恢复到原来的值。
            it.iternext()           # 循环访问x的下一个元素

        numerical_grads.append(grad)
    return numerical_grads
```

該函數接受的參數f表示要計算梯度的那個函數，而params則表示該函數的參數，因為f可能有多個參數，params表示這多個參數構成的一個集合 （如python的list、tuple等類型對象）。為了更具一般性，假設params的每個元素x又是包含多個元素的一個多維數組。

內層循環中對x的每個下標idx指向的元素$x[idx]$，分別加上微小的增量$x[idx]+eps$和$x[idx]-eps$併計算相應的函數值f()，然後用導數的差分逼近公式計算這個$x[idx]$對應的偏導數並賦值給$grad[idx]$。注意：每次對$x[idx]$修改後，要使恢復為原來的值，不然會影響其他偏導數的計算並在退出這個函數後影響了params的值。

可以用這個通用的數值梯度計算函數去計算前面的函數的數值梯度：

```
x = np.array([2.,3.])
param = np.array(x)        #numerical_gradient的參數param必須是numpy數組
numerical_grads = numerical_gradient(lambda:f(param),[param],1e-6)
print(numerical_grads[0])
```

```
    [ 0.25       54.00000001]
```

注意，numerical_gradient的第一個參數f必須指向一個函數對象而不是函數調用的結果，將上面的`lambda:f(param)`寫成`f(param)`是錯誤的。

對一個包含一些參數如param的函數f，通常可用上面的lambda表達式或下面的包裹函數fun，返回一個在參數param上執行計算的函數對象。

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

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

```
    [ 0.25       54.00000001]
```

在後面的章節中，將會使用這個通用的數值梯度計算函數numerical_gradient()計算模型函數的數值梯度。該函數及其他函數包含在本書的源代碼文件util.py中。

2.5 分離梯度下降算法與參數優化策略

2.5.1 參數優化器

前面將變量（參數）的優化策略硬編碼在梯度下降算法中，不同優化策略的梯度下降法除了其中的參數更新不同外，其梯度下降法的框架式完全一樣的。為了提高代碼的複用性和靈活性，可以將參數的優化策略從梯度下降算法中分類出來。

可以定義一個表示參數優化策略的類：

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

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

params是變量（參數）的列表，step()用於根據梯度grads更新這些參數params。如可以在該類基礎上派生定義採用基本梯度下降法的參數優化策略的參數優化器類SGD：

```
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
```

同樣，可以定義其他的參數優化器，如動量法的SGD_Momentum：

```
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 接受參數優化器的梯度下降法

梯度下降算法只要接受對參數更行的參數優化器就可以按照該優化器的優化策略對參數進行更新：

```
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("梯度足够小！")
            break
        grad = df(x)
        x, = optimizator.step([grad])
        x = x.copy()
        history.append(x)
    return history
```

看一個簡單的凸函數曲面，

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

這是一個碗形曲面，如圖2-16，其最小值在碗底，即(0,0)是整個函數的最小值點，最小值是0。

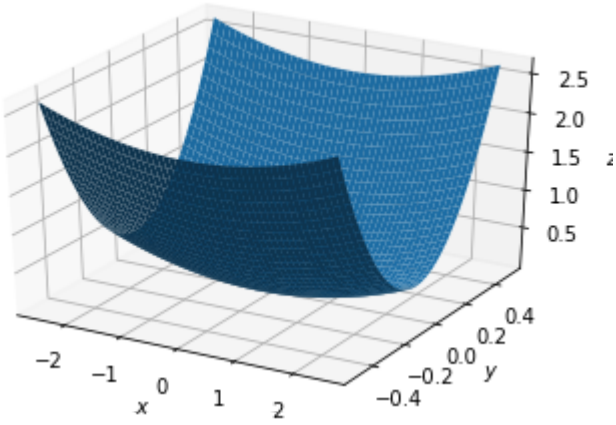

圖 2-16 函數$f(x,y)=\frac{1}{16}x^2+9y^2$曲面

對該函數，應用上述SGD參數優化器：

```
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]
```

逼近了最優解，換用SGD_Momentum優化器：

```
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()
```

同樣更好地逼近了最優解。

```
    梯度足够小!
    [-1.49829905e-08 -4.74284398e-10]
```