第 2 章 勾配降下法

ディープ ラーニングの中心的なタスクは、サンプル データを使用して関数モデルをトレーニングすること、またはこれらのサンプル データを表現または説明するための最適な関数を見つけることです。最良の関数モデルを解くことは、数学的最適化問題、より正確には、特定の損失関数の最大値 (極値) を見つける問題に帰着します。深層学習では、**勾配降下法**を使用して、この最大値問題またはモデル パラメーターを解決します。

この章では、関数の極値の必要条件から始めて、勾配降下アルゴリズムの理論的基礎、アルゴリズム原理、およびコード実装を紹介し、勾配降下法の解変数 (パラメーター) を更新するためのさまざまな最適化戦略を紹介します。

2.1 関数極値の必要条件

関数 $y = f(x)$ は、特定の点 x_0 で **最小値** を取得します。これは、特定の正の数 ϵ があることを意味し、間隔 $(x_0 - \ epsilon, x_0 + \ epsilon)$ 各 x は $f(x_0) \leq f(x)$ を満たします。 x_0 は関数の **最小点** と呼ばれ、$f(x_0)$ は関数の **最小点** と呼ばれます。

関数 $y = f(x)$ は、特定の点 x_0 で **最大値** を取得します。これは、特定の正の数 ϵ があることを意味し、間隔 $(x_0 - \ epsilon, x_0 + \ epsilon)$ 各 x は $f(x) \leq f(x_0)$ を満たします。 x_0 は関数の **最大点** と呼ばれ、$f(x_0)$ は関数の **最大値** と呼ばれます。

最小値と最大値をまとめて極値と呼び、最小値点と最大値点をまとめて極値点と呼ぶ。

関数 $f(x)$ のドメイン内のすべての x が $f(x_0) \leq f(x)$ を満たす場合、x_0 は関数の **最小点** と呼ばれ、$f(x_0)$ は、関数の **最小値** と呼ばれます。

関数 $f(x)$ のドメイン内のすべての x が $f(x) \leq f(x_0)$ を満たす場合、x_0 は関数の **最大点** と呼ばれ、$f(x_0)$ は、関数の **最大値** と呼ばれます。

すなわち、最小値はグローバルレンジの最小値であり、最大値はグローバルレンジの最大値である。最小値と最大値をまとめて **Most Value** と呼び、最小点と最大点をまとめて **Most Value Points** と呼びます。

関数極値に必要な条件: x_0 が関数 $f(x)$ の極値点であり、関数が x_0 で導出可能である場合、$f'(x_0) = 0$ は極値での微分値は 0 でなければなりません。

例えば、前の関数 $f(x) = x^2$ は $x = 0$ で最小値 (もちろん最小値でもあります) を取得し、導出できるため、$x = 0$ ではその導関数を値 $f'(0) = 2 \times 0 = 0$ は 0 でなければなりません。

この命題は簡単に証明できます. x_0 が関数 $f(x)$ の極値である場合, $f(x_0) \leq$ を満たす区間(x_0-\epsilon,x_0+\epsilon)が存在します. $f(x)$、したがって $f(x) - f(x_0) \geq 0$、一方:

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

x が左右から x_0 の傾向にある場合、Δx はそれぞれ負の数と正の数であり、分子は常に正であり、x から右から x_0 の傾向がある場合、その限界値は ≥ 0, from x 左から x_0 の傾向にあり、その限界値は ≤ 0 のはずで、この限界値が存在するため、その値は 0 しかありません。

極限式によると、x_0 での導関数が正の数である場合、関数 f(x) がこの点を中心に単調増加している、つまり $x_1 < x_2$ であるという規則も見つかります。の場合、$f(x_1) < f(x_2)$、つまり、x が増加すると $f(x)$ が増加します。または、Δx が正数の場合、Δy も正数です。たとえば、$y = f(x) = x^2$ の導関数は $f'(x) = 2x$ です. x が 0 より大きい場合、導関数は正です. したがって、関数曲線は単独で0 の場合、導関数はすべて負の数であるため、関数曲線は単独で減少します。つまり、$x_1 < x_2$ の場合、代わりに $f(x_1) > f(x_2)$.

たとえば、関数 $f\left(x\right) = x^3 – 3x^2 – 9x + 2$. の導関数 $f'\left(x\right) = 0$:

$f'\left(x\right) = 0, \ \Rightarrow \left(x^3 – 3x^2 – 9x + 2\right) = 0, \ \Rightarrow 3x^2 – 6x – 9 = 0, \ \Rightarrow x^2 – 2x – 3 = 0, \ \Rightarrow x_1 = –1, \ x_2 = 3$
$x_1 = –1, \ x_2 = 3$ という導関数が 0 の 2 つの点が得られます。この関数とその微分関数 $f'(x)$ の単調な変化を図 2-1 に示します。

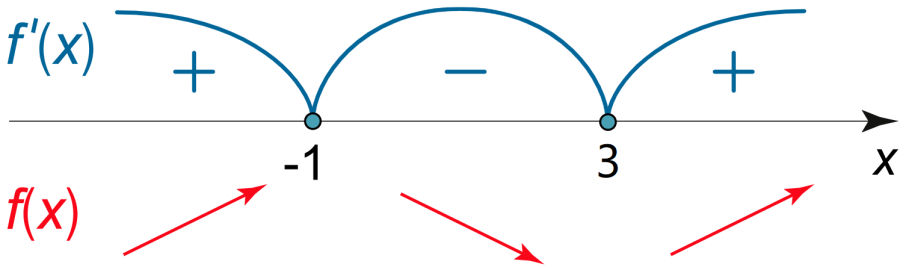

図 2-1 $f'(x) > 0$、関数は単調増加、$f'(x) < 0$、関数は単調減少

区間 $(-\infty, -1]$ では、$f'(x)$ は正の数なので、関数 f(x) は単調増加し、区間 $(-1, 3)$ では、$f'(x)$ は負の数であるため、関数 f(x) は区間 $[3, \infty)$ で単調減少し、$f'(x)$ は正の数であるため、関数f(x) は単調増加です。

次のコードは、この関数とその導関数の曲線を描き、単調な変化と極値の状況をより直感的に確認できます。

```
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()
```

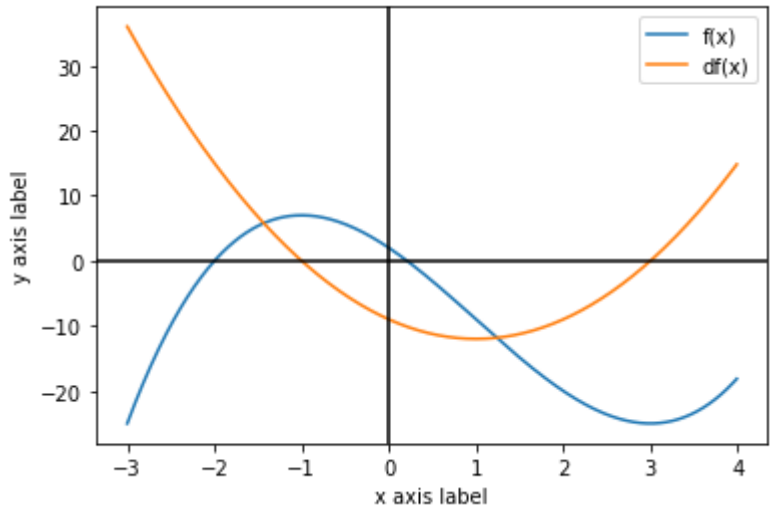

図 2-2 $f(x) = x^3 - 3x^2 - 9x + 2$ とその導関数 $f'(x)$ の関数曲線

上記の命題は、関数の極値における必要条件のみを示しており、十分条件を示していないことに注意してください。つまり、関数 x_0 での導関数 $f'(x_0) = 0$ は、x_0 は極値でなければなりません。たとえば、$x = 0$ での $f(x) = x^3$ の導関数 $f'(0)$ も 0 ですが、この点は関数の極値ではありません。実際、この関数は、図 2-3 に示すように、単調に増加する曲線です。

```
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()
```

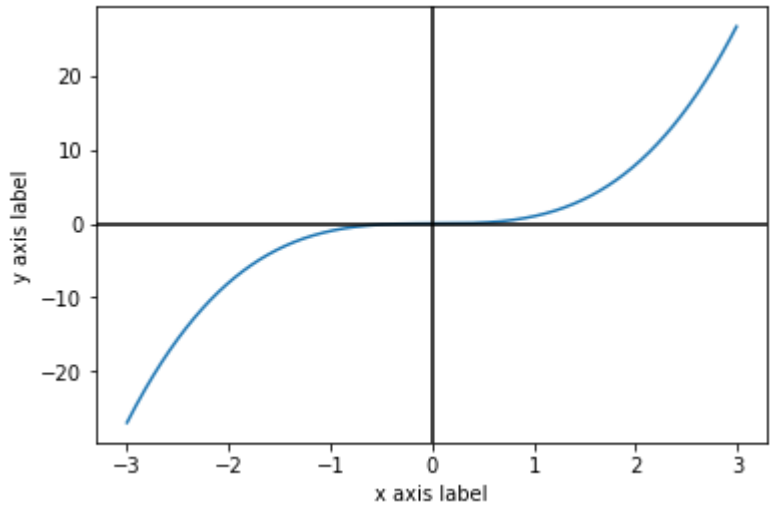

図 2-3 $f(x) = x^3$ の関数曲線

明らかに、関数の極値に必要な条件は多変量関数に拡張できます。つまり、関数が特定の点 $x^* = (x_1^*, x_2^*, \cdots, x_n^*)$ が極値を取得し、この点での勾配が存在する (つまり、すべての偏導関数が存在する) 場合、この点での勾配は0 である (つまり、各偏導関数の値は 0 です)。たった今:

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

2.2 勾配降下法 (gradient descent)

1 変数関数 f(x) の場合、特定の点 x の近くで Δx の小さな変化がある場合、f(x) の変化 $f(x + \Delta x) - f(x)$の微分形式 :

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

つまり、x の近くで、Δx と $f'(x)$ が同じ符号の場合、$f'(x)\Delta x$ は $f(x + \Delta x) - f(x)$ 正の数です. Δx と $f'(x)$ の符号が逆なら $f'(x)\Delta x$ つまり $f(x + \Delta x) - f(x)$ は負の数です。 $\Delta x = -\alpha f'(x)$ (α は小さい正の数) の場合、$f(x + \Delta x) - f(x) = -\alpha f'(x)^2$ は負の数です。つまり、$f(x + \Delta x)$ の値は $f(x)$ よりも小さくなります。つまり、x は $f'(x)$ の反対方向 $-f'(x)$ に沿って Δx を移動し、その関数値 $f(x + \Delta x)$ は元の $f(x)$ は小さいです。

図 2.4 に示すように、$x = 1.5$ における関数 $f(x) = x^2 + 0.2$ の関数値 $f(x)$ は 2.45 であり、微分値 $f'(x)$は 3.0 で、図の長い矢印で示されているように、$f(x)$ のドメイン上の x 軸、つまり x 軸の正の方向を指す正の数です。

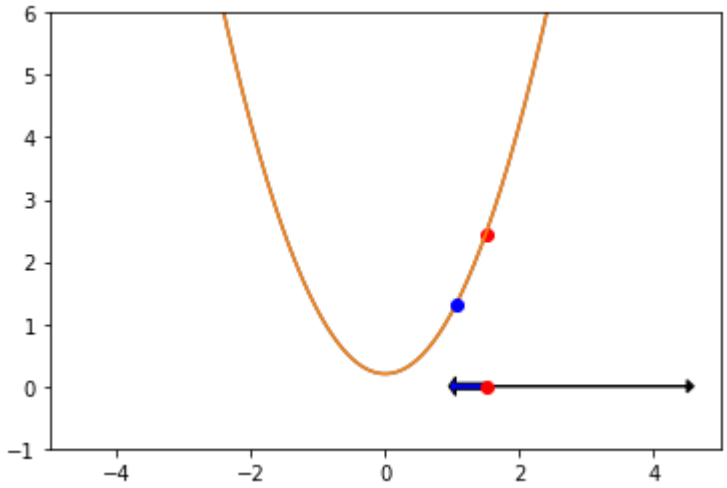

図 2-4 $f'(1.5) = 2.45 > 0$、Δx と $f'(x)$ が同じ符号で動く場合、関数値は増加し、それ以外の場合、関数値は減少します

$\alpha = 0.15, \Delta x = -\alpha f'(x) = -0.449$ とし、この Δx に沿って (図の青い矢印の方向に) x を $x_{に}$移動します。$new = x + \Delta x = 1.05$ とすると、得られた新しい $x = 1.05$ での f(1.05) 関数値は 1.3025 で、これは図の曲線上の青い点の y 座標値です。 Δx と $f'(x)$ は反対方向 (一方が負で他方が正) であるため、この f(1.05) は元の f(1.5) よりも小さくなければなりません。

このプロセスを繰り返し続けてください。つまり、微分 $f'(x)$ の反対方向 ($-f'(x)$) に沿って x を小さな増分 $-\alpha f'(x)$ だけ移動します。新しい $x_{new} = x - \alpha f'(x)$ に達すると、この新しい x_{new} の関数値 $f'(x_{new})$ は前の関数値よりも小さくなければなりません$f'(x)$. x が最小点の x 値に近づくと、導関数 f'(x) も 0 に近くなります (関数極値 x^* の導関数 $f(x^*) = 0$ のため)、 x 運動の増加量 Δx はますます 0 に近づいています。

これが **勾配降下法**の考え方です。つまり、最初の x から開始して、x の値は次の式で継続的に更新されます。

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

現在の x に対して、負の微分 (勾配) 方向 (つまり、$-f'(x)$) に沿って x を移動すると、$f(x)$ が小さくなり続ける可能性があります。理想的には、$f'(x) = 0$ の最小 $f(x)$ の x に到達します。次に、x を繰り返し更新すると、x の値は変更されなくなります。図 2-5 に示すように、x は常に更新を繰り返しているため、常に極値に近づいています。

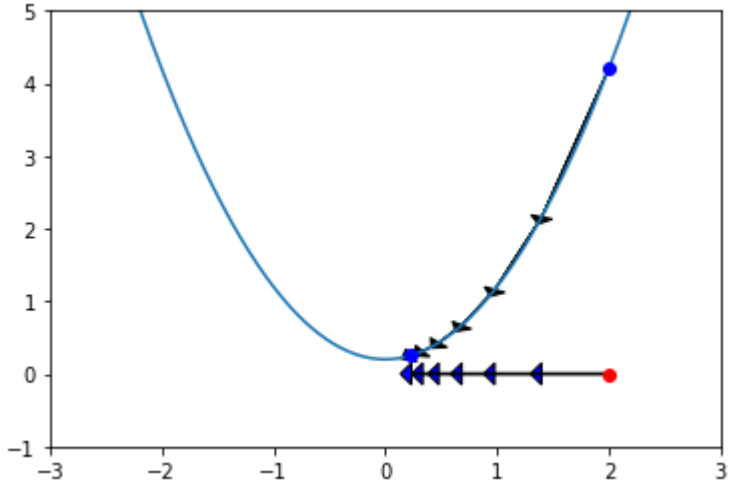

図 2-5 x は $-f'(x)$ に沿って移動し、関数値は減少し続けます

もちろん、この動きのペース (つまり、$-\alpha f'(x)$) が大きすぎることはありません。なぜなら、導関数の定義によれば、上記の近似式は x の近くでしか適用できないからです。移動ペースが大きすぎる場合、x の最適値がスキップされ、x の値が常に前後に振動する可能性があります。図 2-6 に示すように。

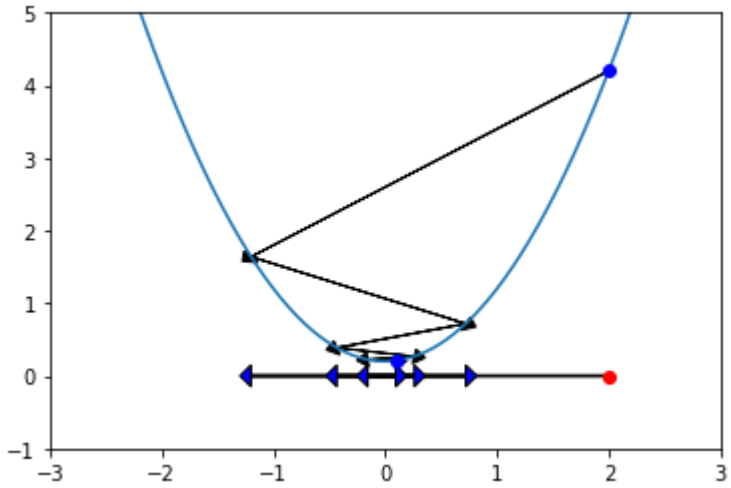

図 2-6 x の変化の大きさ $-\alpha f'(x)$ が大きすぎるため、関数値が振動します

勾配降下法は、おおよその最適解を見つけることです.反復を避けるために、次の方法を使用して、最適解に十分に近いかどうかを確認できます:

- 導関数 (勾配) $f'(x)$ の絶対値は十分に小さい。
- 反復回数が事前設定された最大反復回数に達しました。

以下は勾配降下法のコードで、パラメータ df を使用して関数 $f(x)$ の導関数 $f'(x)$ を計算します。x は変数の初期値、アルファです。は学習率、 iterations は反復回数を表し、epsilon は df=$f'(x)$ の値が 0 に近いかどうかをチェックします。

```
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("勾配は十分に小さいです！")
            break
        x = x-alpha* df(x)
        history.append(x)
    return history
```

この勾配降下関数は、反復プロセス中に更新されたすべての x を Python リスト オブジェクトの履歴に保存し、このオブジェクトを返します。

上記の関数 $f(x) = x^3–3x^2–9x + 2$ の場合、その導関数 $f'(x) = 3x^2–6x–9$ ドル。x=1 に近い関数f(x)$ の最小値が必要な場合は、この関数 gradient_descent() を呼び出すことができます。

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

```
    勾配は十分に小さいです！
    2.999999999256501
```

$f(x)$ の極値 x=2.999999999256501 を取得します。反復プロセスで x に対応する曲線上の点を描くことができます。

```
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()
```

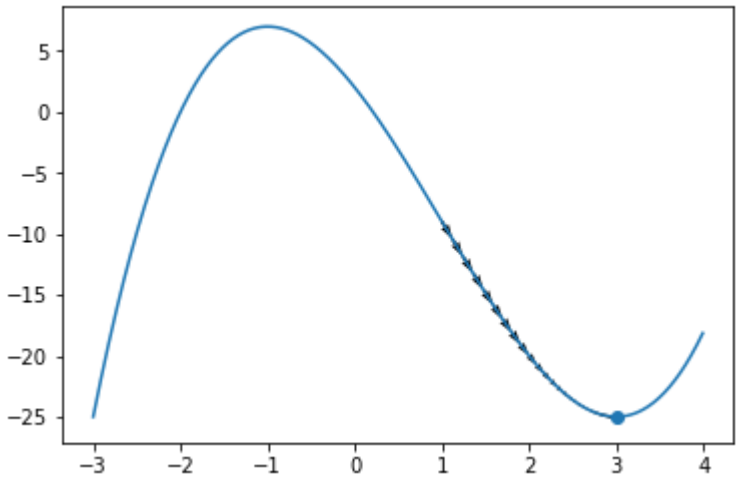

図 2-7 x は徐々に最小点に収束します

その中で、matplotlib の quiver 関数は矢印を使用して速度ベクトルを描画でき、その関数形式は次のとおりです。

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

ここで、X、Y は 1D または 2D 配列で、矢印の位置を示し、U、V は同じ 1D または 2D 配列で、矢印の速度 (ベクトル) を示します。その他のパラメータについては、公式ドキュメントを参照してください。

多変数関数の場合、勾配降下法の原理は同じですが、導関数の代わりに勾配が使用されます。

$f(x+\Delta x)-f(x)\simeq \nabla f(x)\Delta x$
以下はウィキペディアのビールの機能です。

$f(x,y)=(1.5-x+xy)^2+(2.25-x+xy^2)^2+(2.625-x+xy^3)^2$
この関数のグローバル最小値は $(3,0.5)$ です。関数値は、次の python コードで計算できます。

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

このサーフェスを描画するには、まず x 軸と y 軸に均等に分散された座標値を取得します。

```
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)
```

次に、np.meshgrid() 関数を使用して、上記の x_list と y_list に従って交点のグリッド ポイント (x, y) を取得し、これらのグリッド座標ポイントに対応する関数値を計算します。

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

最後に、plot_surface() 関数を呼び出して、このサーフェスを描画できます。

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

完全なコードは次のとおりです。

```
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()
```

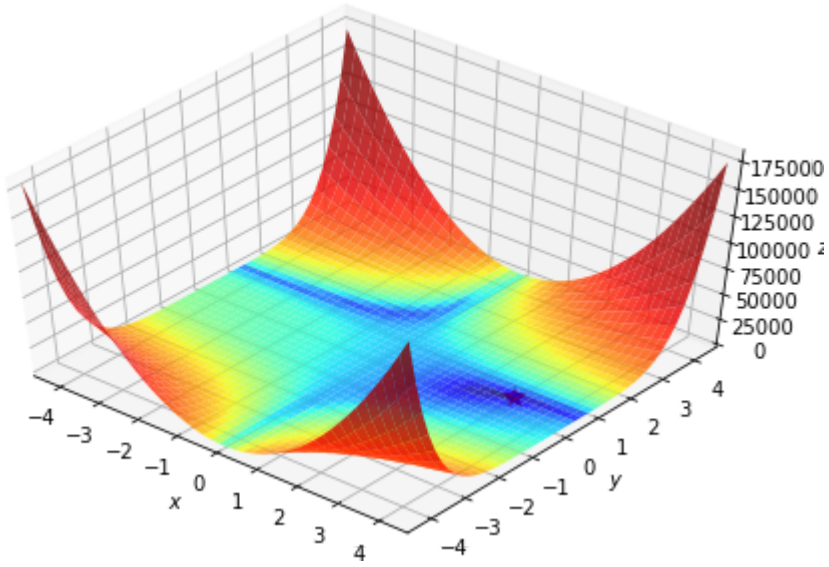

図 2-8 描かれた f(x,y) 面

x, y に関する $f(x, y)$ の偏導関数は次のとおりです。

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

これらのグリッド ポイントでの勾配方向は、matplotlib の quiver 関数を使用して 2D 座標平面にプロットできます。

```
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()
```

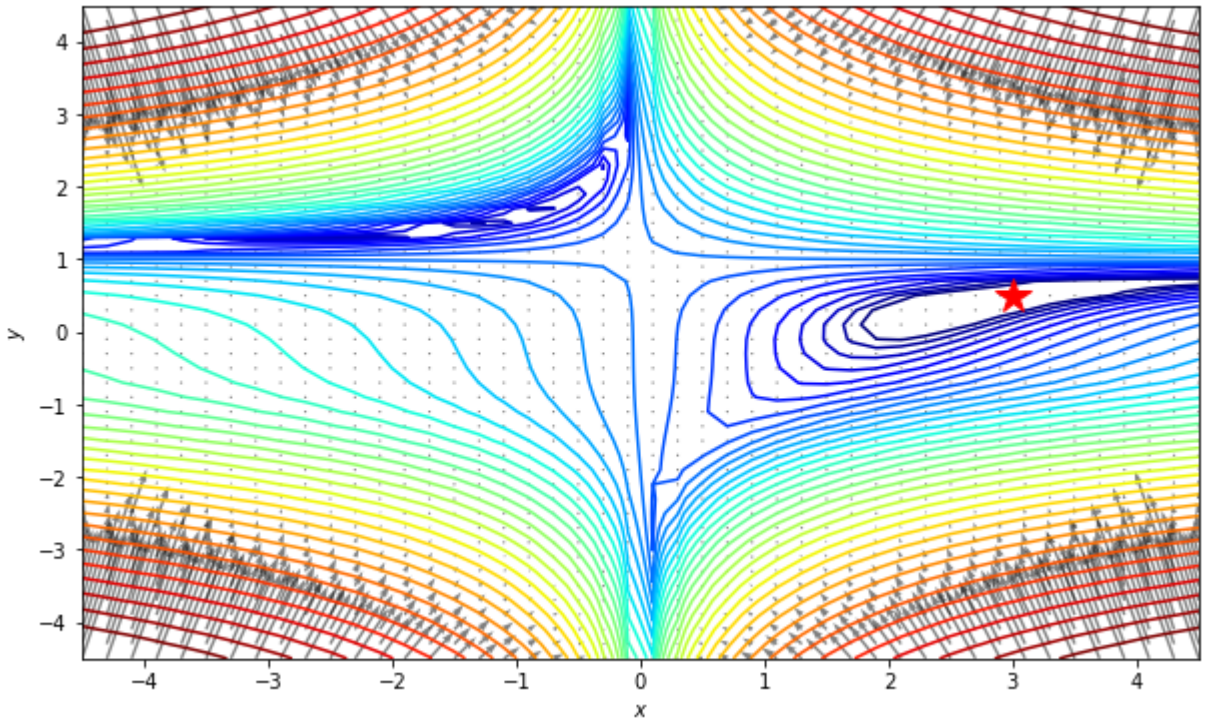

図 2-9. 関数 f(x,y) の等値面の格子点におけるドメイン座標等高線と勾配方向

以前の勾配降下法のコードを直接使用するために、以前の勾配降下法のコードの x を numpy ベクトルで表すことができ、

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

着替える:

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

最初に、分離された x 座標と y 座標の配列を 1 つの配列に結合します。

```
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)

このベクトル化された座標点 x に対して勾配関数 df を定義できます. 次のコードは、勾配降下アルゴリズムの修正されたベクトル化バージョンの実装も提供します:

```
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("勾配は十分に小さいです！")
            break
        x = x-alpha* df(x)
        history.append(x)
    return history
```

次のコードは、x0=(3., 4.) から開始して、このサーフェスの極値を解決します。

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

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

```
    始点 [3. 4.] 勾配 [25625.25 57519.  ]
    極点：  [2.70735828 0.41689171]
```

x の初期勾配値が非常に大きくなり始めるため、学習率 α は小さい数値 (0.000005 など) を取る必要があります。そうしないと、ショックまたは無限値が発生し、最終的に `[2.70735828 0.41689171 ]`, しかし、それは最良の点ではありません.反復プロセス中のxの変化を描くことで、この状況をより直感的に見ることができます.

```
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)
```

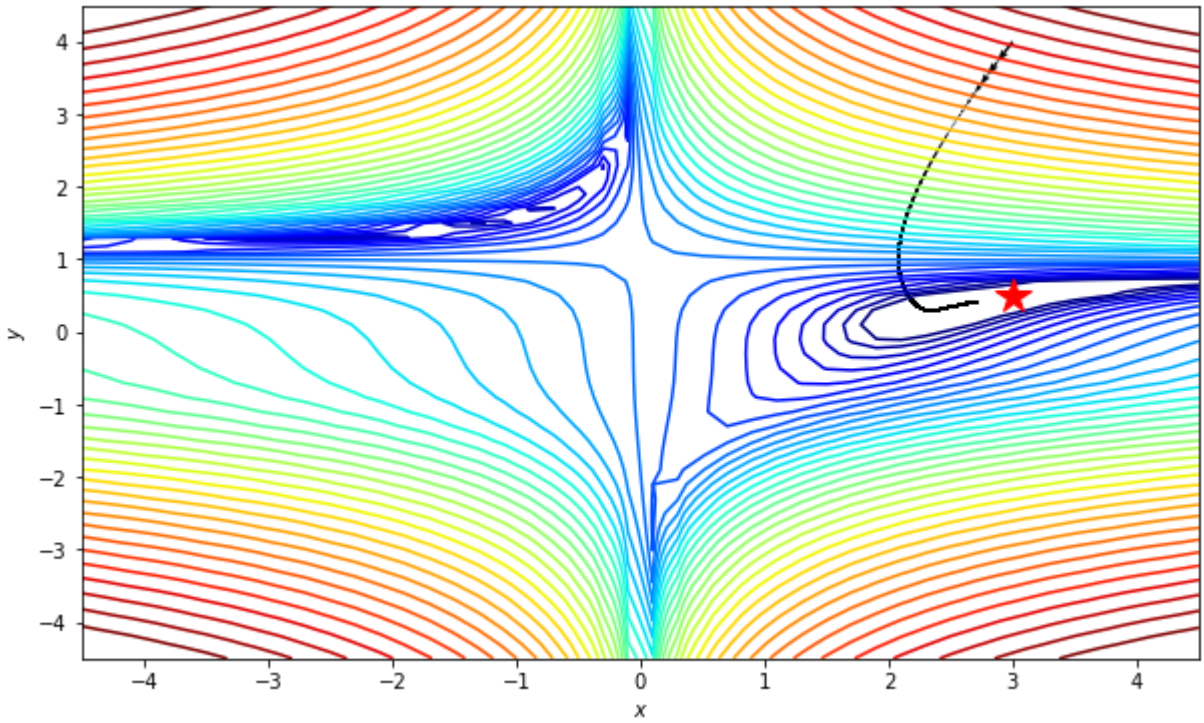

図 2-10 反復処理中、勾配値はどんどん小さくなり、収束はどんどん遅くなります

繰り返しの過程で、勾配値はどんどん小さくなり、同じ学習率でも x の更新は非常に遅くなり、100,000 回繰り返しても最適解に近づくことができません。自然なアプローチは、適応学習率を使用することです。つまり、勾配が小さくなると学習率を上げます。演習として、読者は勾配降下アルゴリズムを変更して、最適なソリューションをより適切かつ迅速に取得することができます。

2.3 勾配降下法のパラメータ最適化戦略

基本的な勾配降下アルゴリズムの学習率は固定値であり、反復プロセス中に勾配の大きさは常に変化します. 学習率が大きすぎると、解決する変数が前後に振動します. 学習率が小さすぎると、収束が非常に遅くなるか、停滞することさえあります. 初期の学習率は中程度ですが、収束が最適解に近づくにつれて、その勾配も 0 に近くなり、これも停滞を引き起こします.当然のことながら、反復プロセス中に徐々に収束するように学習率を調整する必要があります。つまり、可変学習率を使用して、反復プロセス中に解決する変数 x を更新します。

最適解により良く、より速く近づくことができるようにするために、勾配降下法の多くの改善が提案されています. これらの改善では、変化する学習率または戦略を使用して、解変数 (パラメーターとも呼ばれます) を更新します.変数 (パラメーター) の更新戦略または方法には、Momentum、Nesterov 加速勾配、Adagrad、Adadelta、RMSprop、Adam、AdaMax、Nadam、AMSGrad などがあります。

関数は多変数関数である可能性があるため、その変数 x は複数の値で構成されるベクトル $\boldsymbol{x}$ である可能性があることに注意してください。一般的に使用される最適化戦略の一部のみを以下に説明します。

2.3.1 運動量運動量法

勾配降下法は、学習率 α と勾配の負の方向、つまり $-\alpha\nabla f(x)$ を使用して、毎回 x を更新します。つまり、ベクトル を更新します。 $-\alpha\nabla f(x) of x)$ は、現在計算されている勾配に完全に依存し、Momentum Momentum メソッドは、現在の勾配だけでなく、最後の更新ベクトルも考慮して x のベクトルを更新します。つまり、更新されたベクトルには慣性があると見なされます。 v_{t-1} が前回の更新に使用されたベクトルであると仮定すると、現在更新されているベクトルは次のようになります。

$v_t = \gamma v_{t-1} + \alpha\nabla f(x)$
x をこの v で更新します。

$x = x - v_t$
x を更新するために使用されるこのベクトルは **momentum** と呼ばれます。運動量法は、更新ベクトルを移動物体の速度と見なし、速度には慣性があります。以前の更新ベクトルと現在の勾配の組み合わせにより、さまざまな時点での勾配の急激な変化を緩和し、更新されたベクトルをより滑らかにします。つまり、以前の動きの慣性を維持するため、勾配が小さい場所, まだ大きな動きがあります. 勾配の急激な増加によって速度がオーバーシュートすることはありません.この方法は、おもりが転がり落ちるボールのようなもので、最も急な下降経路を探しながら一定量の慣性を維持します。通常の勾配降下は、急なところは急ぎ、平らなところはほとんど動かないのと同じように、急なところは急ぎ足で移動する速度が決まるだけです。

$\boldsymbol{v}$ の初期値は 0 で、Python コードでは次のように表現できます。

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

つまり、$\boldsymbol{v}$ は $\boldsymbol{x}$ と同じ形状のテンソルで、初期値は 0 です。反復プロセスでは、最初に $\boldsymbol{v}$ が更新され、次に関数のパラメーター $\boldsymbol{x}$ が更新されます。

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

以下は、運動量法に基づく勾配降下法です。

```
def gradient_descent_momentum(df,x,alpha=0.01,gamma = 0.8, iterations = 100,epsilon = 1e-6):
    history=[x]
    v= np.zeros_like(x)             #運動量は 0 に初期化されます
    for i in range(iterations):
        if np.max(np.abs(df(x)))<epsilon:
            print("勾配は十分に小さいです！")
            break
        v = gamma*v+alpha* df(x)  #勢いを更新する
        x = x-v                   #更新変数 (パラメーター)

        history.append(x)
    return history
```

この運動量法の勾配降下法を使用して、上記の問題を解決します。

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

[2.96324633 0.49067782]

運動量法の解が最適解に非常に近いことがわかります。写真が示すように。

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

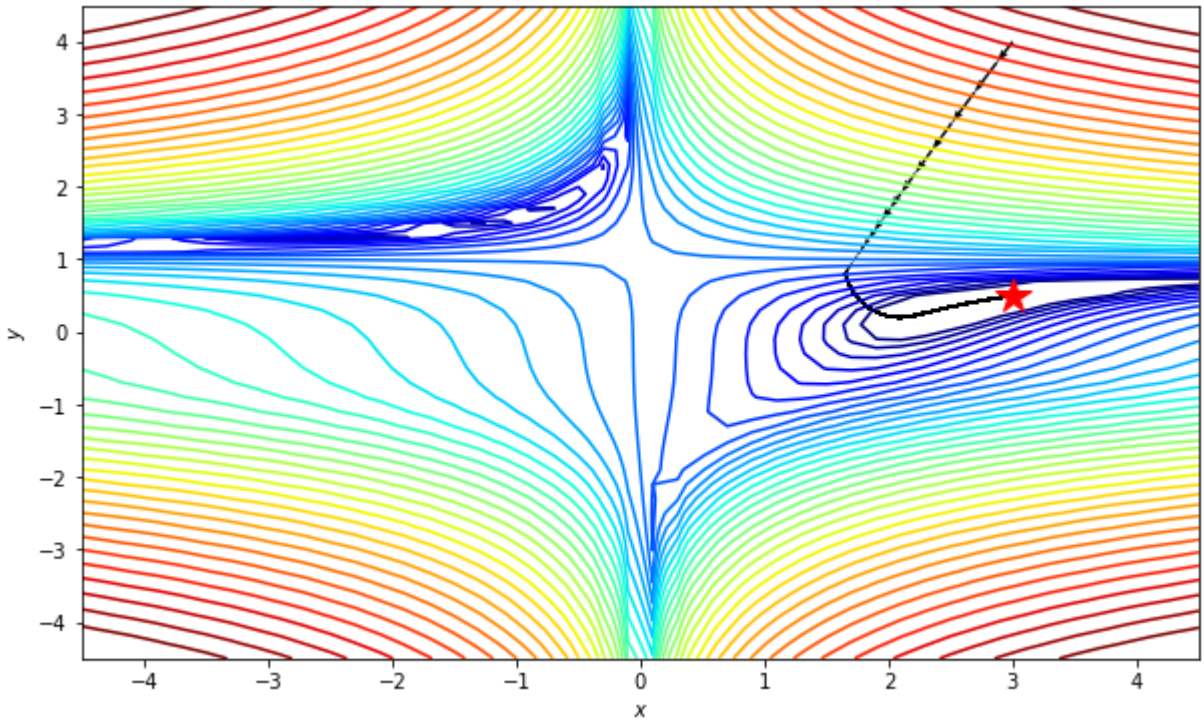

図 2-11 運動量法はすぐに最適に近い解に収束する

2.3.2 アダグラード法

勾配降下法の変数更新式 $\boldsymbol{x} = \boldsymbol{x} - \alpha \nabla f(\boldsymbol{x})$ によれば、変数更新に影響を与えるのは、学習率と勾配 αの積です。$\nabla f(\boldsymbol{x})$、勾配が大きすぎるか小さすぎる、学習率が大きすぎるか小さすぎると、アルゴリズムの収束に影響します。

多変量関数の場合、各変数の偏導関数の大きさは大きく異なる可能性があります。たとえば、偏導関数 $\frac{\partial f}{\partial x_1}, \frac{\partial f}{\text{部分} x_2}$ の絶対値は大きく異なる場合があります。
それらに同じ学習率を使用することは不適切です.一方のコンポーネントの適切な学習率が他方のコンポーネントに対して大きすぎたり小さすぎたりすると、ショックと停滞が生じます。つまり、次の式で直接更新することは不適切です。

$x_1 = x_1 - \alpha \frac{\partial f}{\partial x_1}$
$x_2 = x_2 - \alpha \frac{\partial f}{\partial x_2}$
アダグラード法は名詞から「適応 (ada) 勾配 (grad)」と訳すことができ、各勾配成分を勾配成分の過去の累積値で除算することで、異なる成分の不均衡な勾配サイズの問題を解消することができます。 . 2 成分 (x_1, x_2) について、各成分の履歴累積値 (G_1, G_2) をそれぞれ計算すると、2 成分の更新式は次のようになります。

$x_1 = x_1 - \alpha \frac{1}{G_1} \frac{\partial f}{\partial x_1}$
$x_2 = x_2 - \alpha \frac{1}{G_2} \frac{\partial f}{\partial x_2}$
$g_{t,i} = \nabla_\theta f(x_{t,i})$ という表記を使用して、成分 x_i の偏導関数 $\frac{\partial f}{\partial x_i}$ を表しますt番目の反復、`t'=1` から `t'=t` までのすべてのラウンドの成分勾配は、次のように計算できます。

$G_{t,i} = \sqrt{\sum_{t'=1}^{t} g_{t',i}^2}$
$g_{t,i}$ を $G_{t,i}$ で割り、コンポーネントを更新します。

$x_{t+1,i} = x_{t,i} - \alpha \frac{1}{\sqrt{\sum_{t'=1}^{t} g_{t',i}^2}} g_{t,i}$
除数が 0 になるのを防ぐために、小さな正の数 ϵ をこの分母に追加して、AdaGrad のパラメーター更新式を次のようにします。

$x_{t+1,i} = x_{t,i} - \alpha \frac{1}{\sqrt{\sum_{t'=1}^{t} g_{t',i}^2 + \epsilon}} g_{t,i}$
基本的なパラメーター更新式を比較します。

$x_{t+1,i} = x_{t,i} - \alpha g_{t,i}$
AdaGrad メソッドは、コンポーネントの勾配サイズの不均衡な問題を解消することがわかります。 AdaGrad のパラメータ更新式は、ベクトル形式で記述できます。

$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t - \alpha \frac{1}{\sqrt{\sum_{t'=1}^{t} \boldsymbol{g}_{t'}^2 + \epsilon}} \odot \boldsymbol{g}_t$
累積された G_t^2 は、初期値 0 の変数 gl で記録できます。反復の各ラウンドで、AdaGrad パラメータ更新の Python コードは次のとおりです。

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

AdaGrad 法の主な利点は、さまざまな勾配値の影響を排除することで、反復プロセスで学習率を継続的に調整することなく、学習率を固定値に設定できることです。一般的な学習率は 0.01 に設定されています。 AdaGrad 法の主な欠点は、反復プロセスでは、累積和 $\sum_{t'=1}^{t} \boldsymbol{g}_{t'}^2$ がどんどん大きくなることです。は正の数です。これにより、学習が遅くなったり、停止したりする可能性があります。また、各コンポーネントの勾配を一定のペースにすることは現実的ではなく、最適解の方向から進行方向をそらす可能性があります。

Adagrad パラメーター更新メソッドに基づく勾配降下法のコードは次のとおりです。

```
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("勾配は十分に小さいです！")
            break
        grad = df(x)
        gl += grad**2
        x = x-alpha* grad/(np.sqrt(gl)+epsilon)
        history.append(x)
    return history
```

上記の問題に対して、勾配降下アルゴリズムを実行します。

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

```
    [-0.69240717  1.76233766]
```

成分勾配の等化により、変数更新の順方向が最適解法から逸脱し、別の局所最適解に収束することがわかります。

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

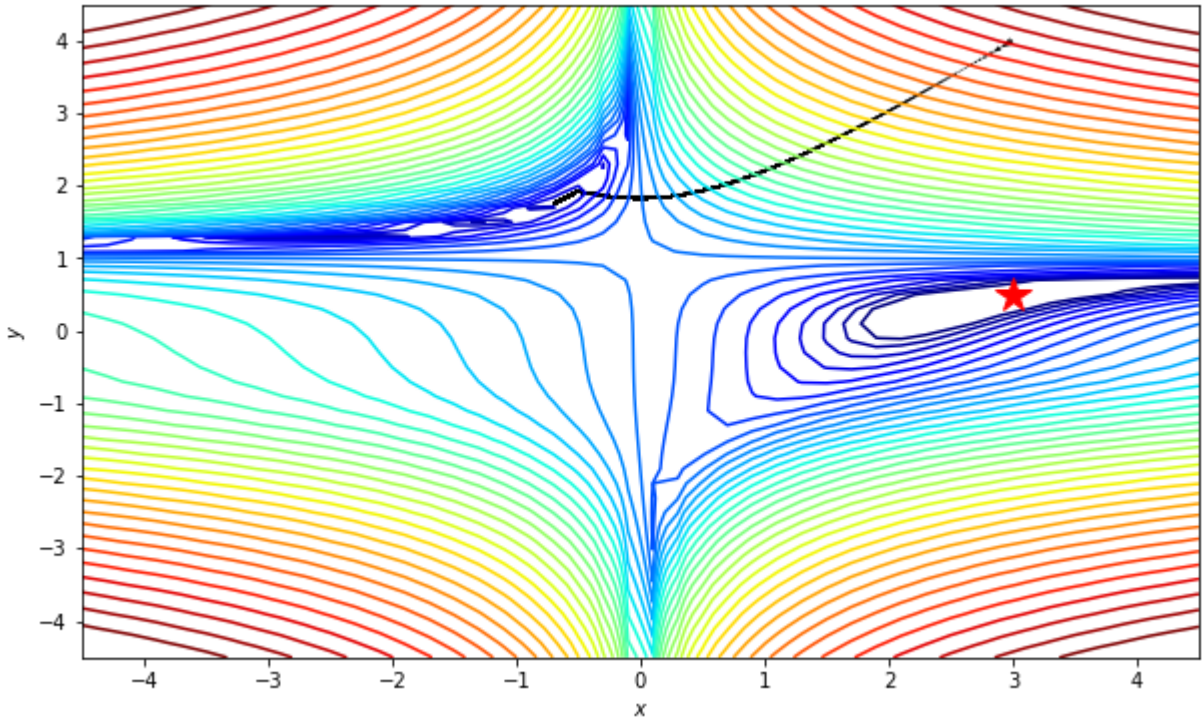

図 2-12 別の極小値に収束する Adagrad 法

2.3.3 アダデルタ法

基本的なパラメーターの更新方法を確認し、 $\Delta \boldsymbol{x}_t$ を使用してパラメーターの更新ベクトルを表します。

$\Delta \boldsymbol{x}_t = -\eta \cdot \boldsymbol{g}_t$
$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t + \Delta \boldsymbol{x}_t$
AdaGrad メソッドの更新ベクトルは次のとおりです。

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

ここで $G_t = \sum \boldsymbol{g}_t^2$ は g_t の過去の二乗和です. 反復プロセスにより、この値 G_t はどんどん大きくなっていますとなり、結果として $\Delta \boldsymbol{x}_t$ がどんどん小さくなっていくので、収束はどんどん遅くなっていきます。解決策は、G_t を平方和ではなく平均平方和 $E[g^2]_t = \frac{G_t}{t}$ に置き換えることです。この $E[g^2]_t$ は、移動平均法を使用して計算できます。つまり、最後の平均値と現在の値の平均を作成します。

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

アダデルタ法はさらに一歩進んで、更新ベクトルにこのような移動平均法を使用して、更新ベクトルの変化をより滑らかにします。

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

最終的な更新ベクトルは次のとおりです。

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

$RMS[\Delta \boldsymbol{x}]_{t-1}, RMS[g]_t$ を使用して $E[\Delta \boldsymbol{x}^2]_{t-1} + \epsilon$ を表し、 $E[g^2]_t + \epsilon$ の場合、更新ベクトルは次のように表すことができます。

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

したがって、パラメータの更新式は次のとおりです。

$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t + \alpha \Delta \boldsymbol{x}_t$
Adadelta メソッドの Python コードは次のとおりです。

```
Eg = rho*Eg+(1-rho)*(grad**2)                    #勾配の累積二乗和を更新する
delta = np.sqrt((Edelta+epsilon)/(Eg+epsilon))*grad     # 更新ベクトルを計算する
x = x- alpha* delta
Edelta = rho*Edelta+(1-rho)*(delta**2)                #更新ベクトルの累積更新
```

通常、アダデルタ法の減衰率パラメータ ρ は 0.9 に設定され、初期値の $\Delta \boldsymbol{x}_t, E[\Delta \boldsymbol{x}^2]_t, E[g^2]_t$ も0です。 Adadelta パラメータ更新メソッドに基づく勾配降下法のコードは次のとおりです。

```
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("勾配は十分に小さいです！")
            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]
```

アダデルタ法も最適解近くまで収束していることがわかる。

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

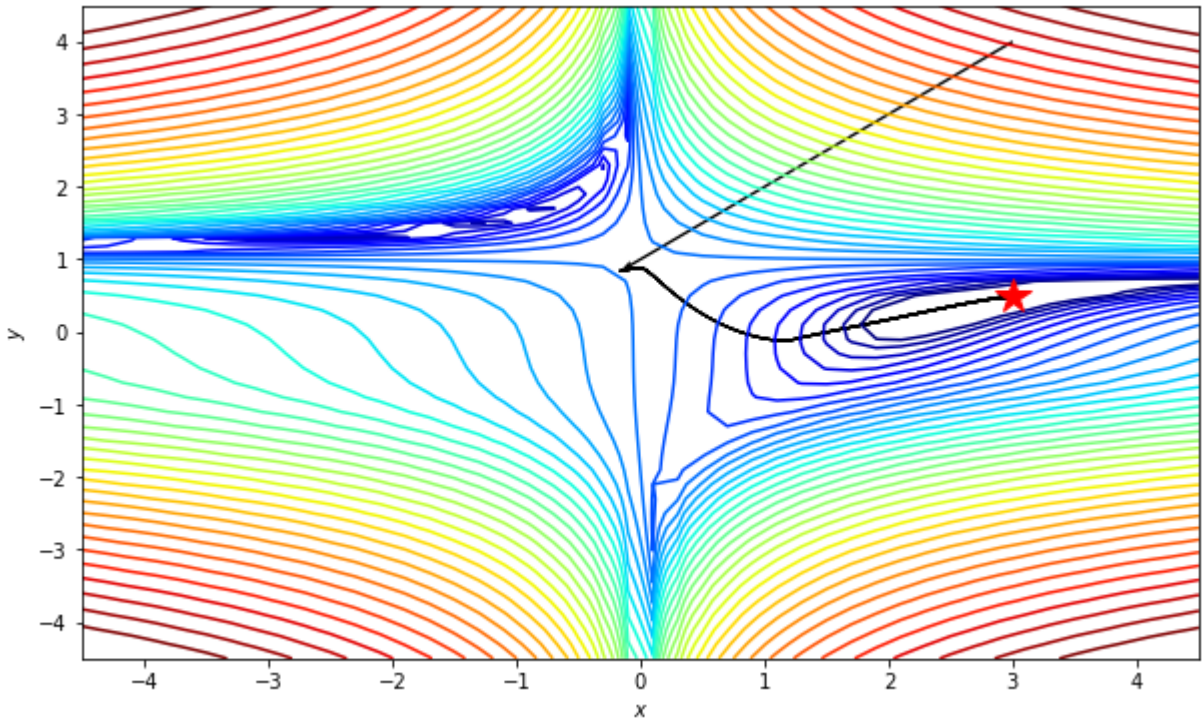

図 2-13 Adadelta 法も最適解に収束できる

2.3.4 RMSprop メソッド

運動量法と同様に、RMSprop は次の式を使用して運動量とパラメーターを更新します。

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

アイデアは、勾配の各値を長さ (値の絶対値) で割る、つまり単位長に変換することです。これにより、パラメーター x は常に固定ステップ サイズ α で更新されます。勾配の各成分の長さを計算するために、RMSprop は運動量法と同様に、勾配値の移動平均の長さの 2 乗値、つまり $f(x)^2$ を計算します。

RMSprop メソッドによってモデル パラメーターを更新するための Python コードは次のとおりです。

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

RMSprop パラメータ更新メソッドに基づく勾配降下法のコードは次のとおりです。

```
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("勾配は十分に小さいです！")
            break
        grad = df(x)
        v = beta*v+(1-beta)*grad**2
        x = x-alpha*grad/(np.sqrt(v)+epsilon)

        history.append(x)
    return history
```

上記の問題に対して、勾配降下アルゴリズムを実行します。

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

```
    [2.70162562 0.41500366]
```

モデル パラメータの結果は十分ではありません。反復回数を増やすことができます。

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

```
    [2.9082809  0.47616156]
```

図 2-14 に示すように、基本的な収束が最適解に近いことがわかります。

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

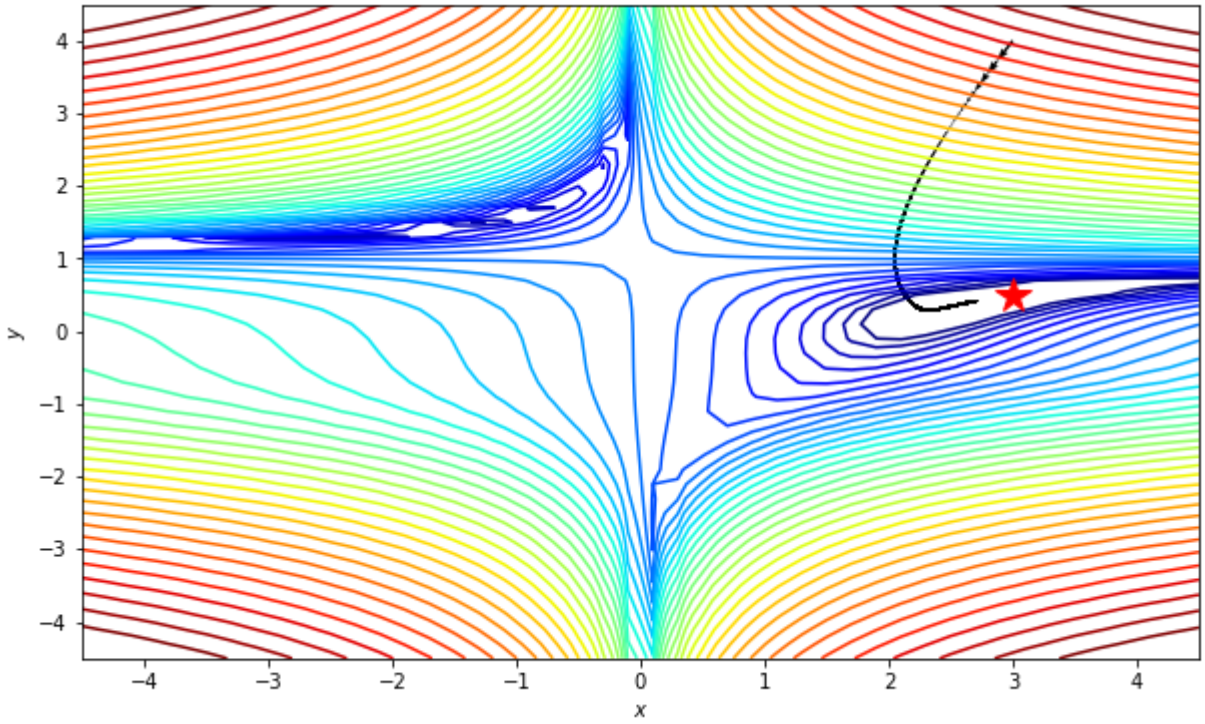

図 2-14 RMSprop 法でも基本的には収束する

2.3.5 アダム法

RMSprop メソッドのように過去の勾配の 2 乗の指数関数的に減衰する累積平均を格納するだけでなく、運動量メソッドのように勾配の累積平均も格納します。運動量法は斜面を転がる球と見なすことができますが、Adam 法は摩擦のある球のように振る舞うため、平坦な最小値に適しています。 m_t, v_t を使用して、過去の勾配と勾配二乗の移動平均を表します。

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$
それらは、初期値が 0 であるため、勾配の 1 次および 2 次の運動量に相当します。Adam の著者は次のように述べています。特に反復の初期段階では、ゼロに偏っています。この問題を修正するために、著者は次の修正式を使用しました。

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$
これに基づいて x を更新します。

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

```
#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("勾配は十分に小さいです！")
            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
```

上記の問題に対して、勾配降下アルゴリズム gradient_descent_Adam を実行します。

```
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)
```

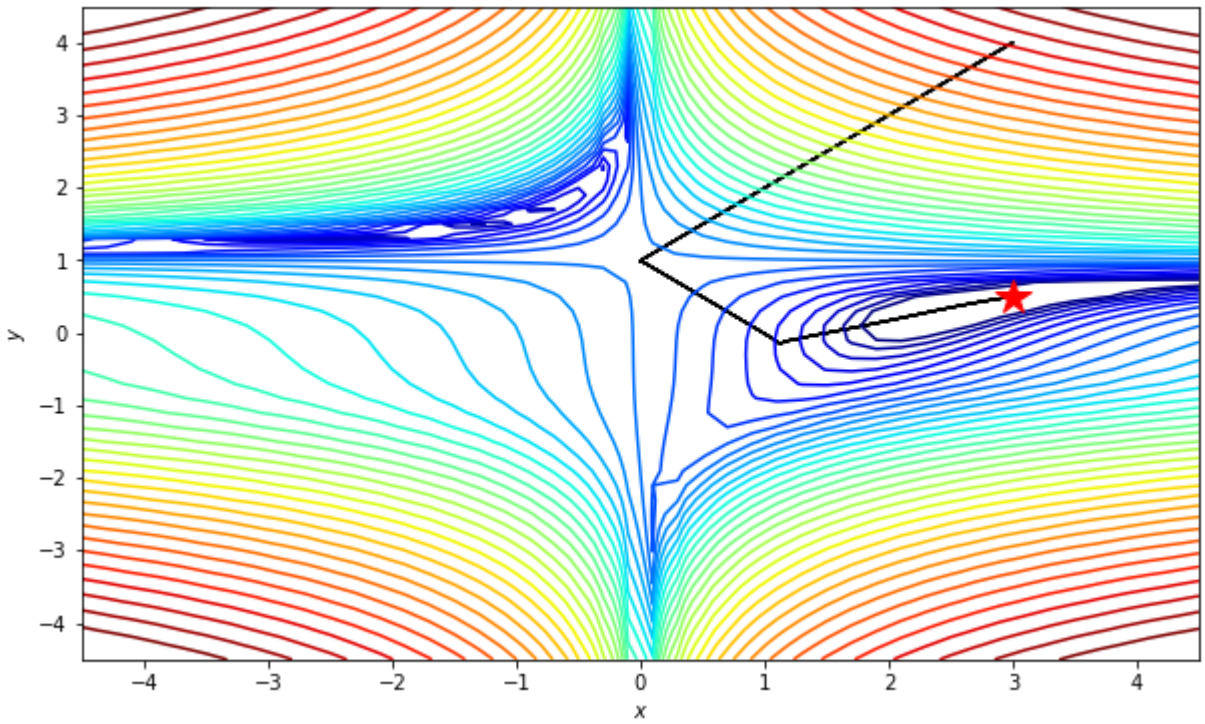

図 2-15 Adam 法も最適解に収束できる

2.4 勾配検証

2.4.1 数値勾配と解析勾配の比較

勾配降下アルゴリズムのコードを書く場合、最も可能性の高い間違いは勾配計算が正しくないことであり、アルゴリズムが収束できないことにつながります.したがって、学習率を調整するだけでなく、勾配計算が正しく行われているかどうかを確認する必要があります.正しい。この目的のために、導関数の定義、つまり、導関数は関数の変化率であるため、点 x での関数の導関数 (勾配) は、次の式で推定できます。

$\frac{\partial f(x)}{\partial x} = lim_{\epsilon->0} \frac{f(x+\epsilon)-f(x-\epsilon)}{2\epsilon}$
つまり、式の右辺の除算を使用して、x での $f(x)$ の導関数 (勾配) を近似します. ϵ が十分に小さい場合、この値の導関数 (勾配) は次のようになります。左の分析導関数 (勾配) と同じである必要があります) は十分に近いです。

したがって、勾配降下法でモデルをトレーニングする前に、数値的に計算された勾配と解析勾配を比較して、解析勾配が正しく計算されていることを確認できます。

たとえば、前のバイナリ関数 $f(x,y) = \frac{1}{16}x^2 + 9y^2$ の場合、勾配降下法では、関数は点 $x = (x_0, x_1)$ の関数値と解析勾配は、次のコードで計算されます。

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

ポイント $x = (x_0, x_1)$ での数値勾配は、次のように計算できます。

```
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))
```

次のコード スニペットは、点 $x = [2., 3.]$ での解析勾配と数値勾配の誤差を比較します。

```
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]
```

数値勾配を計算する小さな増分 eps が十分に小さい限り、この数値勾配は解析勾配に十分近く、これが導関数の定義であることがわかります。数値勾配は、分析勾配。 2つの誤差が比較的大きいか大きいことが判明した場合は、分析勾配または関数値または数値勾配の計算に問題がある可能性があることを意味します. エラーのほとんどは、分析勾配の計算に問題があります.勾配または関数値。

勾配降下法を使用して最適解を解く前に、勾配検証法を使用して、解析勾配と関数値の計算が正しいことを確認する必要があります。これに基づいて、学習率や運動量パラメーターなどの勾配降下法のハイパーパラメーターを調整します。

2.4.2 一般的な数値勾配

機械学習には、多くのパラメーターを含む深層学習の仮想関数が含まれており、一般的な数値勾配計算関数は次のように記述できます。

```
def numerical_gradient(f,params,eps = 1e-6):
    numerical_grads = []
    for x in params:
        # x は多次元配列の場合があります。各要素について、その数値偏導関数を計算します
        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: 重みパラメーターを必ず元の値に戻してください
            it.iternext()           # x の次の要素を反復する

        numerical_grads.append(grad)
    return numerical_grads
```

この関数で受け入れられるパラメーター f は、勾配を計算する関数を示し、params は関数のパラメーターを示します。これは、f が複数のパラメーターを持つ場合があるためです。params は、これらの複数のパラメーターのセット (python のリスト、タプルなど) を示します。物体） 。より一般的には、params の各要素 x が、複数の要素を含む多次元配列であると仮定します。

内側のループで、x の各添え字 idx が指す要素 $x[idx]$ に対して、小さなインクリメント $x[idx]+eps$ と $x[idx]-eps$ をそれぞれ追加し、対応する関数値 f() を計算し、導関数の微分近似式を使用して、この $x[idx]$ に対応する偏導関数を計算し、$grad[idx]$ に代入します。注: $x[idx]$ を変更するたびに、元の値に戻す必要があります。そうしないと、他の偏導関数の計算に影響し、この関数を終了した後の params の値に影響します。

この一般的な数値勾配計算関数を使用して、前の関数の数値勾配を計算できます。

```
x = np.array([2.,3.])
param = np.array(x)        #numeric_gradient のパラメーター param は numpy 配列でなければなりません
numerical_grads = numerical_gradient(lambda:f(param),[param],1e-6)
print(numerical_grads[0])
```

```
    [ 0.25       54.00000001]
```

numeric_gradient の最初のパラメータ f は、関数呼び出しの結果ではなく関数オブジェクトを指さなければならないことに注意してください. 上記の `lambda:f(param)` を `` `f(param)` `` と書くのは誤りです.

param などのいくつかのパラメーターを含む関数 f の場合、通常、上記のラムダ式または次のラッパー関数 fun を使用して、パラメーター param で計算を実行する関数オブジェクトを返すことができます。

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

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

```
    [ 0.25       54.00000001]
```

以降の章では、この一般的な数値勾配計算関数 numeric_gradient() を使用して、モデル関数の数値勾配を計算します。この関数とその他の関数は、本のソース コード ファイル util.py に含まれています。

2.5 分離勾配降下アルゴリズムとパラメーター最適化戦略

2.5.1 パラメータオプティマイザー

変数 (パラメータ) の最適化戦略は、勾配降下アルゴリズムでハードコーディングされています. 異なる最適化戦略の勾配降下法は、パラメータの更新を除いて同じフレームワークを持っています.コードの再利用性と柔軟性を向上させるために、パラメーター最適化戦略を勾配降下アルゴリズムから分類できます。

パラメータ最適化戦略を表すクラスを定義できます。

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

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

params は変数 (パラメーター) のリストであり、step() を使用して、勾配勾配に従ってこれらのパラメーター params を更新します。たとえば、基本的な勾配降下法を使用してパラメーター最適化戦略を定義するパラメーター オプティマイザー クラス SGD は、このクラスに基づいて導出できます。

```
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
```

同様に、運動量メソッドの SGD_Momentum など、他のパラメーター オプティマイザーを定義できます。

```
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 パラメータオプティマイザを受け入れる勾配降下法

勾配降下アルゴリズムがパラメーターを更新するパラメーター オプティマイザーを受け入れる限り、オプティマイザーの最適化戦略に従ってパラメーターを更新できます。

```
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("勾配は十分に小さいです！")
            break
        grad = df(x)
        x, = optimizator.step([grad])
        x = x.copy()
        history.append(x)
    return history
```

単純な凸関数曲面を見ると、

$f(x,y) = \frac{1}{16}x^2 + 9y^2$
これは、図 2-16 に示すように、お椀型の曲面です。その最小値はお椀の底にあり、つまり、(0,0) は関数全体の最小値ポイントであり、最小値は次のとおりです。 0.

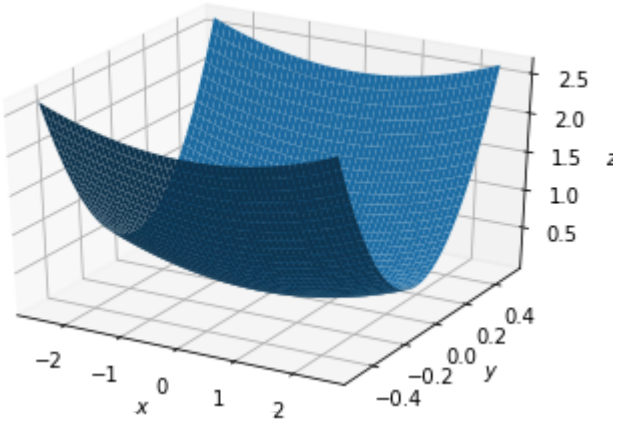

図 2-16 関数 $f(x, y) = \frac{1}{16}x^2 + 9y^2$ 面

この関数に、上記の SGD パラメータ オプティマイザを適用します。

```
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]
```

最適解に近づいたら、SGD_Momentum オプティマイザに切り替えます。

```
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()
```

また、最適解をより適切に近似します。

```
    勾配は十分に小さいです!
    [-1.49829905e-08 -4.74284398e-10]
```