凸优化作业二

1190200122 袁野

测试函数

```
import numpy as np
import math

def Ackley(x1, x2):
        y = -20 * np.exp(-0.2 * np.sqrt(0.5 * (x1**2 + x2**2))) - np.exp(0.5
* (np.cos(2 * np.pi * x1) + np.cos(2 * np.pi * x2))) + np.e + 20
        return y

def Booth(x1, x2):
        y = np.power(x1 + 2 * x2 - 7, 2) + np.power(2 * x1 + x2 - 5, 2)
        return y

def Branin(x1, x2):
        y = math.pow(x2 - (5.1 / (4.0 * math.pi * math.pi)) * x1 * x1 + 5.0
/ math.pi * x1 -6, 2) + 10 * (1 - 1 / (8.0 * math.pi)) * math.cos(x1) + 10
        return y

def rosenbrock(x1, x2):
        y = np.sum(100.0 * (x2 - x1**2)**2 + (1 - x2)**2)
        return y

def michalewicz(x, m=10):
        n = x.size
        return -np.sum(np.sin(x) * np.sin(np.arange(1, n + 1) * x**2 /
np.pi)**(2 * m))

def wheeler(x):
        return np.sum(x**2 + 10 * np.sin(5 * x)**2)

def circle(x):
        return np.sum(x**2 - 10 * np.cos(2 * np.pi * x))
```

共轭梯度法、DFP、BFGS、FRCG等在解决局部最小化和多峰问题时表现更好。

1、

```
import numpy as np

def calc_grad(x):
        return np.array([1.0+4*x[0]+2*x[1], -1.0+2*x[0]+2*x[1]])

def func(x):
        return x[0]-x[1]+2*x[0]**2+2*x[0]*x[1]+x[1]**2

def solve(x, eps, calc_grad, func):
        grad = calc_grad(x)
        p = -grad
        while np.linalg.norm(grad) > eps:
                alpha = np.sum(grad**2) / np.sum(p * (calc_grad(x + p) -
grad))
                x += alpha * p
                new_grad = calc_grad(x)
                beta = np.sum(new_grad**2) / np.sum(grad**2)
                p = -new_grad + beta * p
                grad = new_grad
        return x, func(x)

x0 = np.array([0.0, 0.0])

x, fx = solve(x0, 1e-6, calc_grad, func)

print(x, fx)
```

输出结果为

```
x*: [-1.   1.5] y*: -1.25
```

2、

```
import math

def golden_section_search(f, l, r, eps=1e-6):
        golden_ratio = (math.sqrt(5) - 1) / 2
        while abs(r-l) > eps:
                lmid = r - golden_ratio * (r - l)
                rmid = l + golden_ratio * (r - l)
                if f(lmid) < f(rmid):
                        r = rmid
                else:
                        l = lmid
        return l, f(l)
```

```
def fibonacci_search(f, l, r, delta):
        fib = [1, 1]
        while (fib[-1] < (r-l)/delta):
                fib.append(fib[-1] + fib[-2])
        n = len(fib) - 1
        while r-l > delta:
                lmid = l + (fib[n-2]/fib[n]) * (r-l)
                rmid = l + (fib[n-1]/fib[n]) * (r-l)
                f1, f2 = f(lmid), f(rmid)
                if f1 < f2:
                        r = rmid
                else:
                        l = lmid
        return l, f(l)

def bisection_search(f, l, r, eps):
        while abs(r-l) > eps:
                mid = (l+r)/2
                if f(mid) > 0.0:
                        r = mid
                else:
                        l = mid
        return l, f(l)

def dichotomous_search(f, l, r, eps):
        delta = eps / 2
        while abs(r - l) > eps:
                lmid = (l + r - delta) / 2
                rmid = (l + r + delta) / 2
                if f(lmid) < f(rmid):
                        r = rmid
                else:
                        l = lmid
        return l, f(l)

def f1(x):
        return 2*x**2 - x - 1

def df1(x):
        return 4*x-1

def f2(x):
        return 3*x**2 - 21.6*x - 1

def df2(x):
```

```
        return 6*x - 21.6

print("Problem A:")
x_min, f_min = golden_section_search(f1, -1, 1, 0.06)
print("黄金分割法: 最小值点: ", x_min, "最小值: ", f_min)
x_min, f_min = fibonacci_search(f1, -1, 1, 0.06)
print("斐波那契亚数列法: 最小值点: ", x_min, "最小值: ", f_min)
x_min, f_min = bisection_search(df1, -1, 1, 0.06)
print("二分法: 最小值点: ", x_min, "最小值: ", f1(f_min))
x_min, f_min = dichotomous_search(f1, -1, 1, 0.06)
print("Dichotomous: 最小值点: ", x_min, "最小值: ", f_min)

print("Problem B:")
x_min, f_min = golden_section_search(f2, 0, 25, 0.08)
print("黄金分割法: 最小值点: ", x_min, "最小值: ", f_min)
x_min, f_min = fibonacci_search(f2, 0, 25, 0.08)
print("斐波那契亚数列法: 最小值点: ", x_min, "最小值: ", f_min)
x_min, f_min = bisection_search(df2, 0, 25, 0.08)
print("二分法: 最小值点: ", x_min, "最小值: ", f2(f_min))
x_min, f_min = dichotomous_search(f2, 0, 25, 0.08)
print("Dichotomous: 最小值点: ", x_min, "最小值: ", f_min)
```

输出结果为:

```
Problem A:
黄金分割法: 最小值点:  0.23606797749978964 最小值:  -1.1246117974981074
斐波那契亚数列法: 最小值点:  0.2364003114405218 最小值:  -1.1246300969421703
二分法: 最小值点:  0.25 最小值:  -1.0
Dichotomous: 最小值点:  0.23124999999999996 最小值:  -1.124296875
Problem B:
黄金分割法: 最小值点:  3.569810343379341 最小值:  -39.87726575389938
斐波那契亚数列法: 最小值点:  3.569855646796231 最小值:  -39.877273953909786
二分法: 最小值点:  3.564453125 最小值:  3.7433416748047197
Dichotomous: 最小值点:  3.5587499999999994 最小值:  -39.8748953125
```

3、

```
import numpy as np

def gold_stein(start_point, direction, func, grad,
                                rho=0.1, alpha=1.5, beta=0.5):
        step_length = 1
        while True:
                step = step_length * direction
```

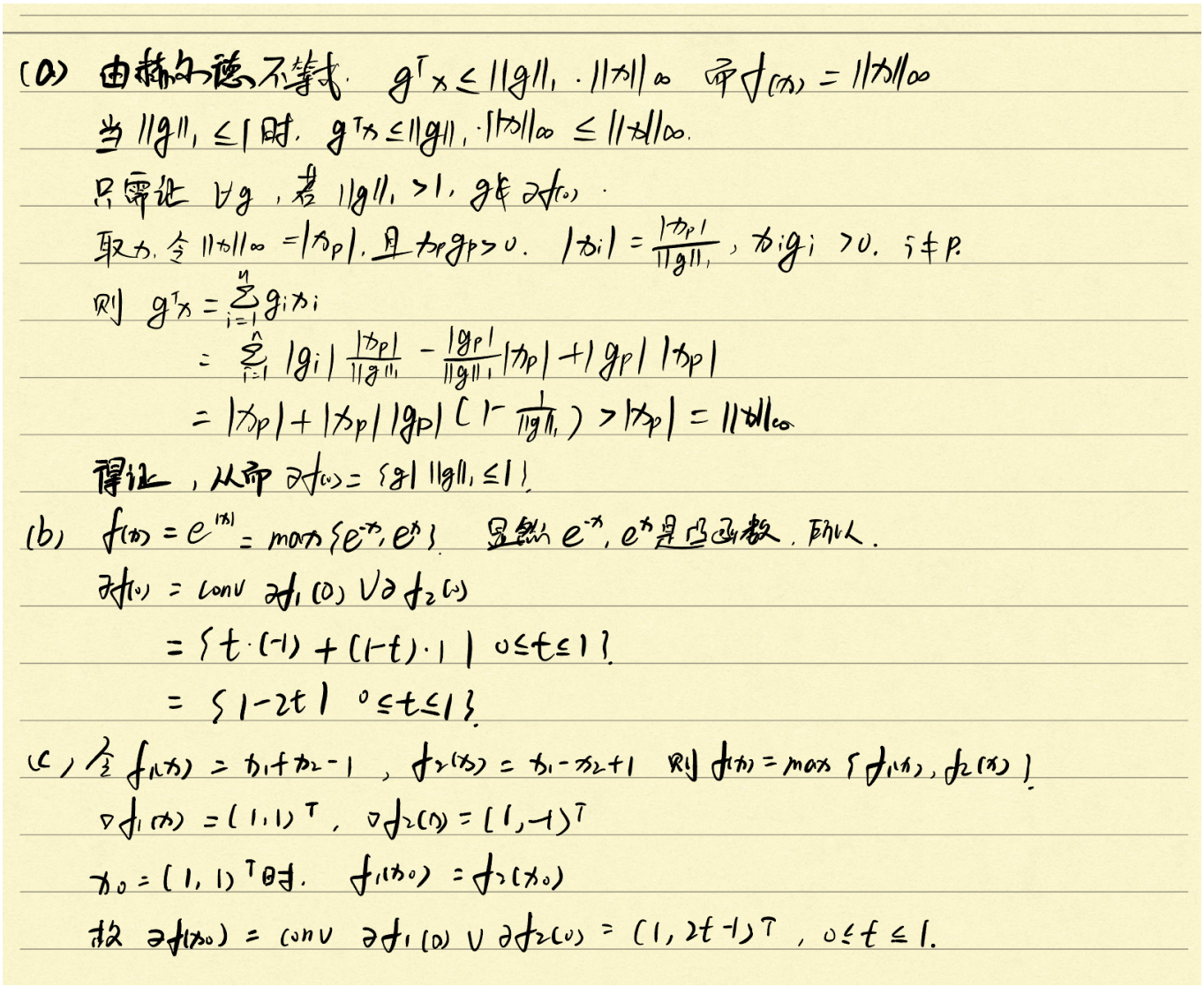
```
                next_point = start_point + step
                next_val = func(next_point)
                current_val = func(start_point)
                if next_val > current_val + rho * np.dot(grad(start_point),
step):
                        step_length = step_length * beta
                elif next_val >= current_val + (1 - rho) *
np.dot(grad(start_point), step):
                        break
                else:
                        step_length = step_length * alpha

        final_solu = next_point
        final_val = next_val

        print(f"Goldstein result: x*:{final_solu}, y*:{final_val}")

def wolfe_powell(start_point, direction, func, grad,
                                rho=0.1, alpha=1.5, beta=0.5, sigma=0.2):
        step_length = 1
        while True:
                step = step_length * direction
                next_point = start_point + step
                next_val = func(next_point)
                current_val = func(start_point)

                if next_val <= current_val + rho * np.dot(grad(start_point),
step):
                        if np.dot(grad(next_point), step) >= sigma *
np.dot(grad(start_point), step):
                                break
                        else:
                                step_length = step_length * alpha
                else:
                        step_length = step_length * beta

        final_solu = next_point
        final_val = next_val

        print(f"Wolfe-Powell result: x*:{final_solu}, y*:{final_val}")

def f(x):
        return 100 * (x[1] - x[0]**2)**2 + (1 - x[0])**2

def grad_f(x):
        dfdx1 = 400 * x[0]**3 - 400 * x[0] * x[1] + 2 * x[0] - 2
```

```
        dfdx2 = 200 * (x[1] - x[0]**2)
        return np.array([dfdx1, dfdx2])

x0 = np.array([-1, 1])
d = np.array([1, 1])

gold_stein(x0, d, f, grad_f)
wolfe_powell(x0, d, f, grad_f)
```

输出结果为：

```
Goldstein result: x*:[-0.99609375  1.00390625], y*:3.9980874294415116
Wolfe-Powell result: x*:[-0.99609375  1.00390625], y*:3.9980874294415116
```

4、



5、

```
import numpy as np
import sympy as sp

def jacobian(f, x):
        x1, x2 = sp.symbols('x1 x2')
        x3 = [x1, x2]
        df = np.zeros((x.shape[0], 1))
        for i in range(x.shape[0]):
                df[i, 0] = sp.diff(f, x3[i]).subs({x1: x[0][0], x2: x[1]
[0]})
        return df

def hesse(f, x):
        x1, x2 = sp.symbols('x1 x2')
        x3 = [x1, x2]
        G = np.zeros((x.shape[0], x.shape[0]))
        for i in range(x.shape[0]):
                for j in range(i, x.shape[0]):
                        G[i, j] = sp.diff(f, x3[i], x3[j]).subs({x1: x[0]
[0], x2: x[1][0]})
                        G[j, i] = G[i, j]
        return G

def dfp(f, x, iters):
        H = np.eye(x.shape[0])
        epsilon = 1e-3
        for i in range(1, iters+1):
                g = jacobian(f, x)
                if np.linalg.norm(g) < epsilon:
                        x_best = np.round(x).astype(int).tolist()
                        break
                d = -np.dot(H, g)
                a = -(np.dot(g.T, d) / np.dot(d.T, np.dot(hesse(f, x), d)))
                x_new = x + a * d
                g_new = jacobian(f, x_new)
                y = g_new - g
                s = x_new - x
                H = H + np.dot(s, s.T) / np.dot(s.T, y) - np.dot(
                H, np.dot(y, np.dot(y.T, H))) / np.dot(y.T, np.dot(H, y))
                x = x_new
        else:
                x_best = np.round(x).astype(int).tolist()
        return x_best

x1, x2 = sp.symbols('x1 x2')
x = np.array([[0.1], [1]])
```

```
f = 10 * x1**2 + x2**2
print(f'result is :\n{dfp(f, x, 5)}')
```

输出为：

```
result is :
[[0], [0]]
```

6、

```
import numpy as np

def f(x):
        x1, x2 = x
        return x1**2 + 4*x2**2 - 4*x1 - 8*x2

def grad(x):
        x1, x2 = x
        return np.array([2*x1 - 4, 8*x2 - 8])

H0 = np.eye(2)
x0 = np.array([0, 0]).reshape(-1, 1)
epsilon = 1e-6
last_x = x0
last_grad = grad(last_x)
last_H = H0

for i in range(1000):
        p = -np.dot(last_H, last_grad)
        alpha = 1.0
        while f(last_x + alpha * p) >= f(last_x):
                alpha *= 0.5
        new_x = last_x + alpha * p
        new_grad = grad(new_x)
        s = new_x - last_x
        y = new_grad - last_grad
        rho = 1 / np.dot(y.T, s)
        new_H = (np.eye(2) - rho * np.dot(s, y.T)) @ last_H @ (np.eye(2) -
rho * np.dot(y, s.T)) + rho * np.dot(s, s.T)
        if np.linalg.norm(new_grad) < epsilon:
                break
        last_x = new_x
        last_grad = new_grad
        last_H = new_H
```

```
print("result is: ", last_x.flatten())
```

输出结果为：

```
result is:  [1.99997955 1.00000131]
```

7、

```
import numpy as np
import sympy as sp

class DFP:

        def jacobian(self, f, x):
                x1, x2 = sp.symbols('x1 x2')
                x3 = [x1, x2]
                df = np.zeros((x.shape[0], 1))
                for i in range(x.shape[0]):
                        df[i, 0] = sp.diff(f, x3[i]).subs({x1: x[0][0], x2:
x[1][0]})
                return df

        def hesse(self, f, x):
                x1, x2 = sp.symbols('x1 x2')
                x3 = [x1, x2]
                G = np.zeros((x.shape[0], x.shape[0]))
                for i in range(x.shape[0]):
                        for j in range(i, x.shape[0]):
                                G[i, j] = sp.diff(f, x3[i], x3[j]).subs({x1:
x[0][0], x2: x[1][0]})
                                G[j, i] = G[i, j]
                return G

        def dfp(f, x, iters):
                H = np.eye(x.shape[0])
                epsilon = 1e-3
                for i in range(1, iters+1):
                        g = jacobian(f, x)
                        if np.linalg.norm(g) < epsilon:
                                x_best = np.round(x).astype(int).tolist()
                                break
                        d = -np.dot(H, g)
                        a = -(np.dot(g.T, d) / np.dot(d.T, np.dot(hesse(f,
x), d)))
```

```
                        x_new = x + a * d
                        g_new = jacobian(f, x_new)
                        y = g_new - g
                        s = x_new - x
                        H = H + np.dot(s, s.T) / np.dot(s.T, y) - np.dot(
                        H, np.dot(y, np.dot(y.T, H))) / np.dot(y.T,
np.dot(H, y))
                        x = x_new
                else:
                        x_best = np.round(x).astype(int).tolist()
                return x_best

        def main(self):
                x1, x2 = sp.symbols('x1 x2')
                x = np.array([[0.0], [0]])
                f = x1**2 + x2**2 - x1*x2 - 10*x1 - 4*x2 + 60
                print(f'DFP result is :{self.dfp(f, x, 5)}')

class BFGS:
        def main(self):
                H0 = np.eye(2)
                x0 = np.array([0, 0]).reshape(-1, 1)
                epsilon = 1e-6
                last_x = x0
                last_grad = grad(last_x)
                last_H = H0
                for i in range(1000):
                        p = -np.dot(last_H, last_grad)
                        alpha = 1.0
                        while f(last_x + alpha * p) >= f(last_x):
                                alpha *= 0.5
                        new_x = last_x + alpha * p
                        new_grad = grad(new_x)
                        s = new_x - last_x
                        y = new_grad - last_grad
                        rho = 1 / np.dot(y.T, s)
                        new_H = (np.eye(2) - rho * np.dot(s, y.T)) @ last_H
@ (np.eye(2) - rho * np.dot(y, s.T)) + rho * np.dot(s, s.T)
                        if np.linalg.norm(new_grad) < epsilon:
                                break
                        last_x = new_x
                        last_grad = new_grad
                        last_H = new_H
                print("BFGS result is: ", last_x.flatten())

class FR:
```

```
        def solve(self, x, eps, calc_grad, func):
        grad = calc_grad(x)
        p = -grad
        while np.linalg.norm(grad) > eps:
                alpha = np.sum(grad**2) / np.sum(p * (calc_grad(x + p) -
grad))
                x += alpha * p
                new_grad = calc_grad(x)
                beta = np.sum(new_grad**2) / np.sum(grad**2)
                p = -new_grad + beta * p
                grad = new_grad
        return x, func(x)

        def main(self):
                x0 = np.array([0.0, 0.0])
                x, fx = self.solve(x0, 1e-6, grad, f)
                print("BFGS result is: ", x)




def f(x):
        x1, x2 = x
        return x1**2 + x2**2 - x1*x2 - 10*x1 - 4*x2 + 60

def grad(x):
        x1, x2 = x
        return np.array([2*x1 - x2 - 10, 2*x2 - x1 - 4])

DFP().main()
BFGS().main()
FR().main()
```

输出结果为：

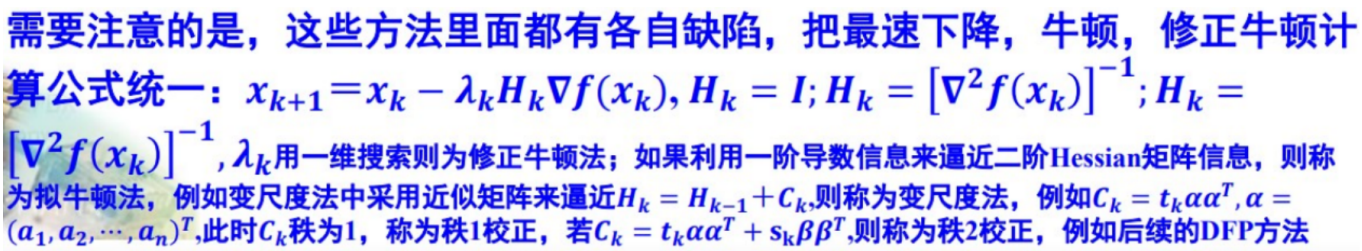
```
DFP result is :[[8], [6]]
BFGS result is:  [8.00000067 5.99999808]
BFGS result is:  [8. 6.]
```

DFP利用迭代方法计算海森矩阵的逆,快速的收敛到局部极小值，并且通常比较稳定。BFGS则是通过更新矩阵来实现海森矩阵的逆，更加不稳定但是更灵活。FR算法不需要计算海森矩阵，计算开销比较小，但是可能不会收敛到全局最优解。

8、

1. 无约束的最优化问题就是在没有对定义域或值域所任何限制的情况下，对函数的最小值进行求解的问题， 简称下山法，首先我们选择一个初始点，然后按照各种不同算法确定一个搜索方向和移动步长。 实际应用中，许多情形被抽象为函数形式后都是凸函数，对于凸函数来说，局部最小值点即为全局最小值点，所以求解比较方便。
2. 1. 修改目标函数，使之转化为凸函数。2) 抛弃一些约束条件，是新可行域为凸集并且包含原来的可行域
3. 1. 根据约束条件的特点，构造出惩罚函数，加入到目标函数中，新目标函数的解与原始目标函数一致。2) 拉格朗日乘子法: 通过拉格朗日乘子，将有约束优化问题转为一些无优化问题来求解。

9、



变尺度算法的基本思想:最速下降法简单但是收敛速度慢，牛顿法收敛速度快，但计算量大，变尺度法整合了两者的有点，在迭代过程中，不断修正构造矩阵来逼近海森矩阵，既可以看作最速下降法的推广，也是牛顿法的推广。

10、

```
import numpy as np
import math

def f(x):
        return 2 * x**2 - x - 1

def grad(x):
        return 4 * x - 1

print(" -------- Momentun --------")
x=0
step_size = 0.1
f_current = f(x)
mu = 0.2
momentum = 0
while True:
        grad_x = grad(x)
        momentum = mu * momentum + step_size * grad_x
        x -= momentum
```

```
        f_new = f(x)
        if abs(f_new - f_current) <= 1e-5:
                break
        f_current = f_new
print(f'x: {x}\nf(x): {f_current}')

print(" -------- Adagrad --------")
x=0
step_size = 0.1
f_current = f(x)
squared_grad_sum = 0
epsilon = 1e-10
while True:
        grad_x = grad(x)
        squared_grad_sum += grad_x ** 2
        step = step_size / ((squared_grad_sum + epsilon) ** 0.5)
        x -= step * grad_x
        f_new = f(x)
        if abs(f_new - f_current) <= 1e-5:
                break
        f_current = f_new
print(f'x: {x}\nf(x): {f_current}')

print(" -------- RMSProp --------")
x=0
step_size = 0.1
f_current = f(x)
n_t = 0
epsilon = 1e-10
gamma = 0.5
while True:
        grad_x = grad(x)
        n_t = gamma * n_t + (1 - gamma) * grad_x ** 2
        step = step_size / ((n_t + epsilon) ** 0.5)
        x -= step * grad_x
        f_new = f(x)
        if abs(f_new - f_current) <= 1e-5:
                break
        f_current = f_new
print(f'x: {x}\nf(x): {f_current}')

print(" -------- SGD --------")
x=0
step_size = 0.1
f_current = f(x)
while True:
```

```
        grad_x = grad(x)
        x -= step_size * grad_x
        f_new = f(x)
        if abs(f_new - f_current) <= 1e-5:
                break
        f_current = f_new
print(f'x: {x}\nf(x): {f_current}')
```

输出结果为：

```
 -------- Momentun --------
x: 0.251168
f(x): -1.1249998848
 -------- Adagrad --------
x: 0.24843456283262597
f(x): -1.1249896594536735
 -------- RMSProp --------
x: 0.25069623562510346
f(x): -1.1249909203440278
 -------- SGD --------
x: 0.2484883456
f(x): -1.1249873050054164
```

11、

一般情况下，深度学习模型的训练过程中，参数变化具有高维特性。这是因为深度学习模型通常包含大量的参数和复杂的损失函数，导致参数更新方向和幅度不稳定、难以预测。然而，一些研究表明，某些情况下深度学习模型的参数变化也可能表现出低维特性，这主要取决于数据的性质和模型的结构等因素。例如，递归神经网络（RNN）或卷积神经网络（CNN）的参数变化可能较为规则，表现出一定的低维特性。

12、

- Momentum-based方法是一种优化算法，它的基本思想是引入一个动量变量来协助参数更新，从而加速梯度下降过程。在该方法中，将传统梯度下降的更新方向与历史上的更新方向进行加权平均，从而使得参数更新更加平稳，并减少更新过程中的震荡。
- 线性加速方法也是一种优化算法，它的基本思想是在每次迭代时，通过将当前迭代点与上一次的迭代点线性组合，得到一个新的迭代点，以加速梯度下降的收敛速度。线性加速方法可以理解为将多个梯度下降的结果进行平均，以获得更好的更新方向。

13、

共轭函数 $f^*(y)$ 是一个对偶变换，它基于函数 $f(x)$ 在对偶空间 X^* 中的取值 x^* ，定义为：

$$f^*(y) = sup_{x \in X}(y^T x - f(x))$$

其中 $y \in X^*$ 是 X 的对偶空间。共轭函数的性质包括：

- $f^*(y)$ 是关于 y 的凸函数；
- $f^*(y^*) = sup_{x \in X}(y^{*T} x - f(x)) = f(x^*) - y^{*T} x^*$，其中 x^* 是 $f(x)$ 在 $x \in X$ 时取得最大值的点。

对偶性是指将问题的对偶问题重新表述成另一个形式的能力，在优化问题中应用广泛。例如，将一个原始问题的拉格朗日对偶问题重新表述后，可以得到原始问题和对偶问题的有效解，且对偶问题具有更好的性质。

共轭函数和对偶性的联系在于，共轭函数是一种基于对偶变换的函数。在将函数进行对偶变换后，其共轭函数所描述的凸性质和最大值问题等依然存在。同时，在优化问题中，对偶问题的重新表述也是一种对原问题的变换，从而得到更加有用的表示形式。这两种变换都基于对偶空间的概念，因此具有内在的联系。