本节要点

- 梯度的定义与性质。
- 梯度下降求解损失函数极值。
- 数据标准化对梯度的意义。

梯度

梯度的概念

梯度是一个向量，其分量为函数f对每个自变量的偏导数。

$$\nabla f(x_1, x_2, \ldots, x_n) = \left(\frac{\partial f(x_1,x_2,\ldots,x_n)}{\partial x_1}, \frac{\partial f(x_1,x_2,\ldots,x_n)}{\partial x_2}, \frac{\partial f(x_1,x_2,\ldots,x_n)}{\partial x_n}\right)$$

- $f(x_1, x_2, \ldots, x_n)$：n元函数。
- ∇：梯度。

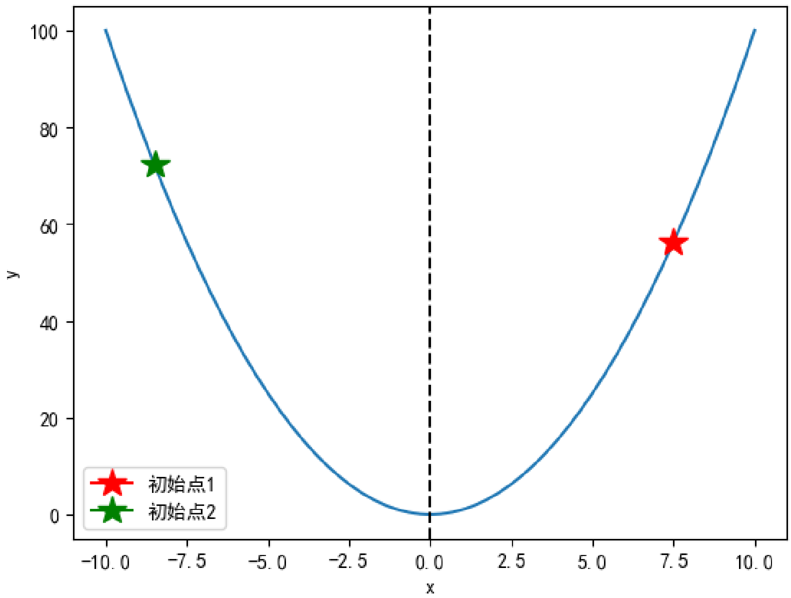
由此可知，当$n = 1$（f为一元函数）时，梯度就是导数。

函数极值求解

> 给定某函数，如果从某初始点开始，不断更新x，尝试找到函数的极小值。

在更新中，我们会遇到如下的问题：

- 初始点的选择往往是随意的，在不同的初始点下，该选择朝哪个方向更新x的值，才能确保函数值y减小？
- 每次更新的幅度是多少？是固定还是不固定的？
- 随着更新的幅度不同，x可能会在中心线两侧反复，如何确定后续的更新方向？



梯度的性质

给定函数$y = f(x_1, x_2, \ldots, x_n)$，在某一点$x$处，梯度具有这样的性质：

- 沿着梯度的方向更新x，函数值y上升最快。
- 沿着负梯度的方向更新x，函数值y下降最快。

一元函数

函数$y = f(x)$在$(-\infty, +\infty)$连续可导，则一阶导数$f'(x)$在点x_0处，具有如下结论：

1. $f'(x_0) > 0 \Leftrightarrow$ 函数单调递增。
2. $f'(x_0) < 0 \Leftrightarrow$ 函数单调递减。
3. $f'(x_0) = 0 \Leftrightarrow$ 函数极值点。

多元函数

给定多元函数$y = f(x_1, x_2, \ldots, x_n)$，此时$x$与$y$是一个向量：

- $\vec{x} = (x_1, x_2, \ldots, x_n)$
- $\vec{g} = (g_1, g_2, \ldots, g_n)$

与一元函数类似，我们可以对每个分量，按照类似一元函数的方式更新，使函数f更大（小）。

梯度法步骤

我们可以通过梯度指引的方向，来求解函数$y = f(x_1, x_2, \ldots, x_n)$的极值。根据求解极值的不同（极大值还是极小值），我们可以分为**梯度上升**或者**梯度下降**。在机器学习领域中，因为习惯上对损失函数求解最小值，故梯度下降的应用会更多一些。

我们以梯度下降求解函数极小值为例，步骤如下：

1. 设定自变量的初始值（初始点位置）$(x_1, x_2, \ldots, x_n)$。
2. 求解该点的梯度$(g_1, g_2, \ldots, g_n)$。

$$g_i = \frac{\partial f(x_1,x_2,...,x_n)}{\partial x_i}$$

3. 根据梯度值指引的方向，移动一小段距离，更新点（自变量）的坐标值。
    - $x_1 = x_1 - \eta g_1$
    - $x_2 = x_2 - \eta g_2$
    - $\ldots$
    - $x_n = x_n - \eta g_n$
4. 重复步骤2-3，直到：
    - 迭代次数达到最大迭代次数（max_iter）。
    - 在连续若干次迭代过程中，函数值 y 的变化 Δy 小于指定的阈值（tol）。

说明：

- η：学习率，用来控制更新幅度的大小。
- 梯度的方向不一定指向极值，但是，沿着梯度的方向更新可以让函数值朝着极值靠近。

如果通过梯度上升求解函数极大值，我们该如何更新点的坐标值？

A $x_i = x_i + \eta g_i$

B $x_i = x_i - \eta g_i$

C $x_i = x_i * \eta g_i$

D $x_i = x_i / (\eta g_i)$

一元函数求极值

使用梯度下降法求解一元函数$y = x^2 - 2x + 1$的最小值。

编写程序

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

plt.rcParams["font.family"] = "SimHei"
```

```python
plt.rcParams["axes.unicode_minus"] = False


def fun(x):
    """定义函数y = x^2 - 2x + 1。

    Parameters
    ----------
    x : ndarray
        自变量。

    Returns
    -------
    y : ndarray
        根据自变量x，返回对应的y值。
    """
    return x ** 2 - 2 * x + 1


def grad_f(x):
    """定义一元函数的导函数。y = 2x - 2，根据自变量x返回函数值y。

    Parameters
    ----------
    x : ndarray
        自变量。

    Returns
    -------
    y : ndarray
        根据自变量x，返回对应的y值。
    """
    return 2 * x - 2


def grad_descent(f, g, x_init, eta=0.1, tol=1e-4, max_iter=100, verbose=True):
    """通过梯度下降求解一元函数的极小值。

    根据给定的函数f与初始点，通过在循环中计算梯度，并更新初始点位置，
    从而逐渐接近函数的极小值。

    Parameters
    ----------
    f : funciton
        原函数，即需要求解极值的函数。
    g : function
        梯度函数。对一元函数来说，梯度函数就是导函数。
    x_init : float
        自变量初始点的位置（值）。
    eta : float, default=0.1
        学习率。控制梯度下降中x_init的更新幅度。
    tol : float, default=1e-4
        容忍值。当多次函数值的差值小于tol时，停止迭代。
    max_iter : int, default=100
        最大迭代次数。当达到最大迭代次数后，停止迭代。
    verbose : bool, default=True。
        是否打印更新过程中的轨迹信息。

    Returns
    -------
    x_list : list
        自变量x的更新轨迹。
    y_list : list
        函数值y的更新轨迹。
    """
    # 定义x_list与y_list，用来保存梯度下降过程中，自变量x与函数值y的更新轨迹。
    x_list = []
    y_list = []
    x = x_init

    for i in range(max_iter):
        x_list.append(x)
        y = f(x)
        y_list.append(y)
        # 如果在两次迭代中，函数y比上一次的减少值小于tol，则停止迭代。
        if len(y_list) >= 2 and y_list[-2] - y < tol:
            break
        x -= eta * g(x)
    if verbose:
        for index, (a, b) in enumerate(zip(x_list, y_list), start=1):
```

```
            print(f"第{index}次迭代，x={a}, y={b}")
    return x_list, y_list


x_list, y_list = grad_descent(fun, grad_f, x_init=10)
```

```
第1次迭代，x=10, y=81
第2次迭代，x=8.2, y=51.839999999999996
第3次迭代，x=6.76, y=33.1776
第4次迭代，x=5.608, y=21.233663999999997
第5次迭代，x=4.6864, y=13.58954496
第6次迭代，x=3.9491199999999997, y=8.697308774399998
第7次迭代，x=3.3592959999999996, y=5.5662776156159985
第8次迭代，x=2.8874367999999997, y=3.562417673994238
第9次迭代，x=2.5099494399999998, y=2.279947311356313
第10次迭代，x=2.2079595519999997, y=1.4591662792680404
第11次迭代，x=1.9663676415999998, y=0.9338664187315455
第12次迭代，x=1.7730941132799998, y=0.5976745079881893
第13次迭代，x=1.6184752906239999, y=0.38251168511244105
第14次迭代，x=1.4947802324992, y=0.24480747847196227
第15次迭代，x=1.3958241859993599, y=0.15667678622205594
第16次迭代，x=1.316659348799488, y=0.10027314318211578
第17次迭代，x=1.2533274790395903, y=0.06417481163655414
第18次迭代，x=1.2026619832316723, y=0.041071879447394544
第19次迭代，x=1.1621295865853378, y=0.026286002846332535
第20次迭代，x=1.1297036692682703, y=0.01682304182165284
第21次迭代，x=1.1037629354146161, y=0.010766746765857787
第22次迭代，x=1.0830103483316929, y=0.0068907179301489485
第23次迭代，x=1.0664082786653544, y=0.00441005947529538
第24次迭代，x=1.0531266229322835, y=0.0028224380641890257
第25次迭代，x=1.0425012983458268, y=0.0018063603610809498
第26次迭代，x=1.0340010386766614, y=0.001156070631091799
第27次迭代，x=1.0272008309413292, y=0.0007398852038986714
第28次迭代，x=1.0217606647530633, y=0.0004735265304951497
第29次迭代，x=1.0174085318024506, y=0.00030305697951682475
第30次迭代，x=1.0139268254419604, y=0.00019395646689090995
第31次迭代，x=1.0111414603535684, y=0.00012413213881012908
```

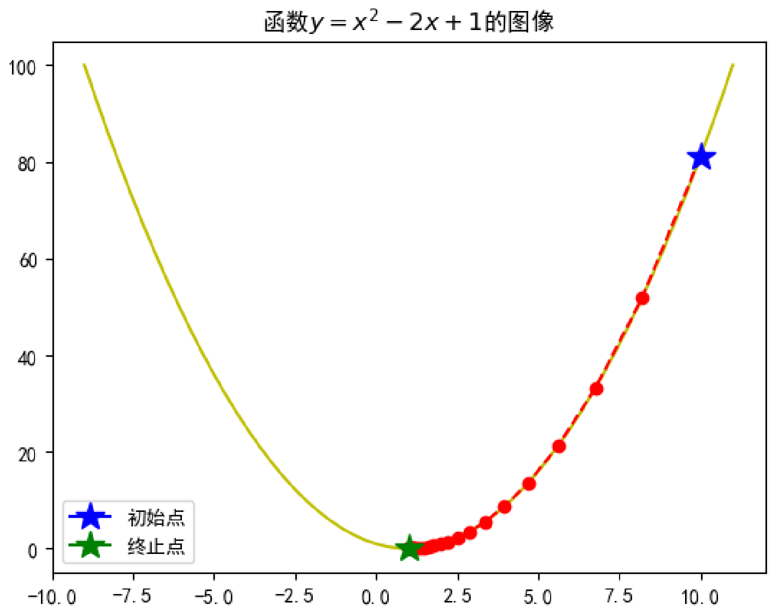
结果可视化

```
def plot_result(f, x, x_li, y_li):
    """绘制可视化结果。

    Parameters
    ----------
    f : funciton
        原函数。
    x : array
        绘制图像的定义域范围。
    x_li : list
        自变量x的更新轨迹。
    y_li : list
        自变量y的更新轨迹。
    """

    y = f(x)
    plt.plot(x, y, c="y")
    plt.title("函数$y=x^{2}-2x+1$的图像")
    plt.plot(x_list, y_list, marker="o", ls="--", c="r")
    plt.plot(x_list[0], y_list[0], marker="*", ms=15, c="b", label="初始点")
    plt.plot(x_list[-1], y_list[-1], marker="*", ms=15, c="g", label="终止点")
    plt.legend()
    plt.show()


domain = np.linspace(-9, 11, 200)
plot_result(fun, domain, x_list, y_list)
```



学习率对梯度下降的影响

在梯度下降过程中，学习率控制每次更新的幅度。学习率的选择应该适度，既不是越大越好，也不是越小越好。

较大的学习率

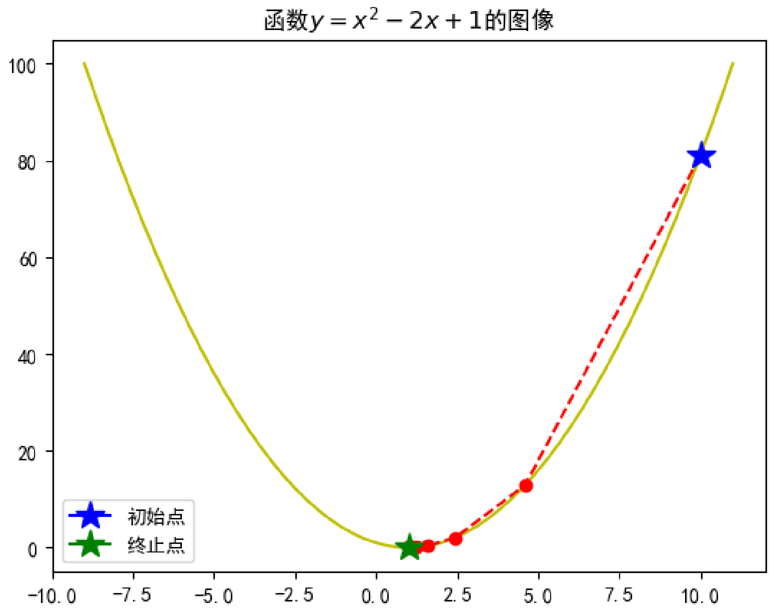
如果学习率较大，则：

- 优点：每次更新的幅度加快，减少迭代次数，能够更快的到达极值点。
- 缺点：如果学习率过大，容易跳过极值点，出现震荡，严重时会导致发散效果，函数值不降反升。

较大的学习率，可以更快的达到极值点。

```
x_list, y_list = grad_descent(fun, grad_f, x_init=10, eta=0.3)
plot_result(fun, domain, x_list, y_list)
```

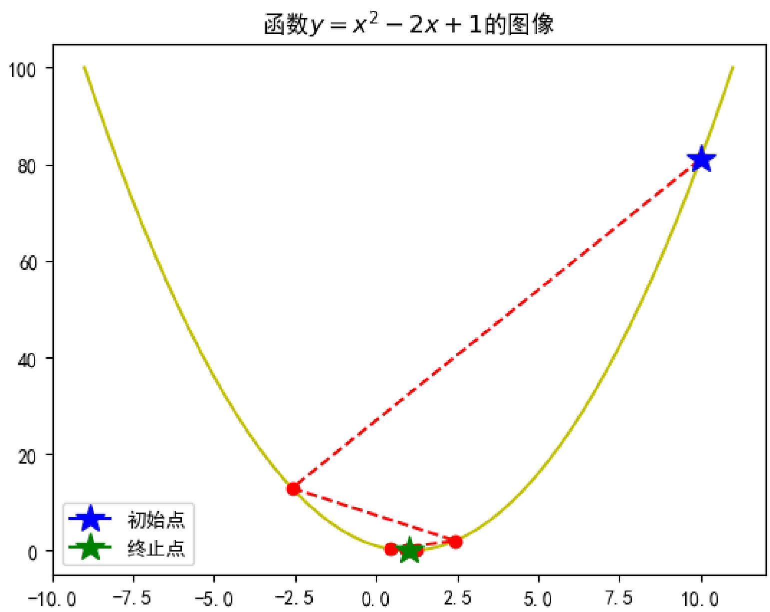
```
第1次迭代，x=10, y=81
第2次迭代，x=4.6000000000000005, y=12.960000000000003
第3次迭代，x=2.4400000000000004, y=2.0736000000000008
第4次迭代，x=1.576, y=0.33177600000000007
第5次迭代，x=1.2304, y=0.05308416000000005
第6次迭代，x=1.09216, y=0.008493465599999972
第7次迭代，x=1.036864, y=0.0013589544959999866
第8次迭代，x=1.0147456, y=0.00021743271936000334
第9次迭代，x=1.00589824, y=3.478923509758758e-05
第10次迭代，x=1.002359296, y=5.566277615720594e-06
```



如果学习率过大，会出现震荡的现象，但依然能够到达极值点。

```
x_list, y_list = grad_descent(fun, grad_f, x_init=10, eta=0.7)
plot_result(fun, domain, x_list, y_list)
```

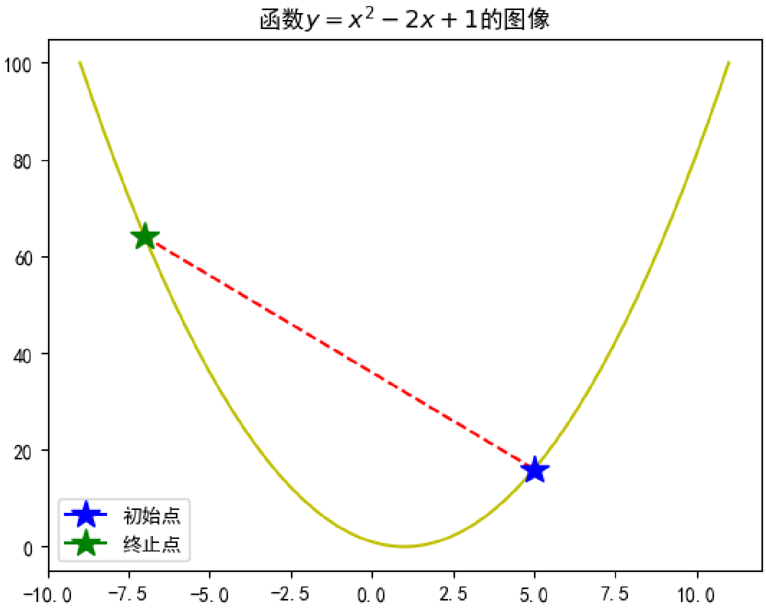
```
第1次迭代，x=10, y=81
第2次迭代，x=-2.5999999999999996, y=12.959999999999997
第3次迭代，x=2.4399999999999995, y=2.073599999999999
第4次迭代，x=0.4240000000000004, y=0.3317759999999996
第5次迭代，x=1.2304, y=0.05308416000000005
第6次迭代，x=0.90784, y=0.008493465599999972
第7次迭代，x=1.036864, y=0.0013589544959999866
第8次迭代，x=0.9852544, y=0.0002174327193600334
第9次迭代，x=1.00589824, y=3.478923509758758e-05
第10次迭代，x=0.9976407039999999, y=5.566277615609572e-06
```



如果学习率过大，会令函数值不降反升。

```
x_list, y_list = grad_descent(fun, grad_f, x_init=5, eta=1.5)
plot_result(fun, domain, x_list, y_list)
```

```
第1次迭代，x=5, y=16
第2次迭代，x=-7.0, y=64.0
```

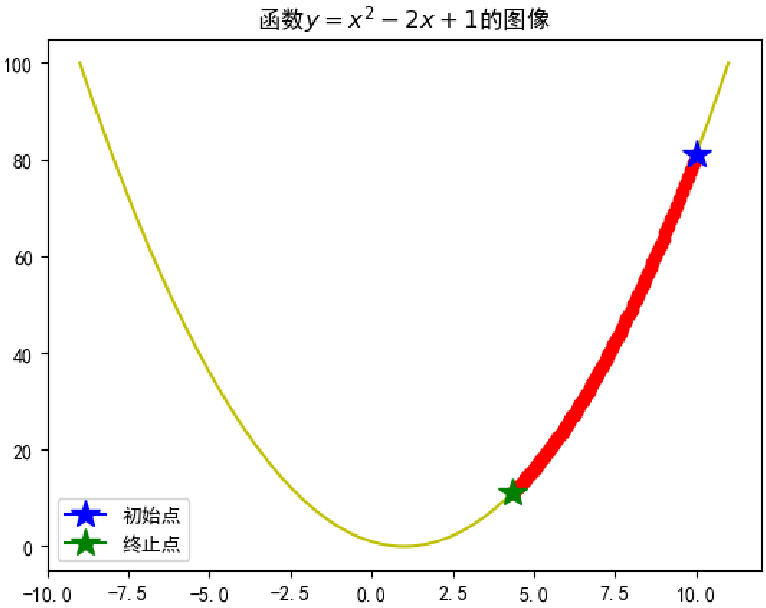


较小的学习率

如果学习率较小，则：

- 优点：不容易出现震荡现象（跳过极值点），函数值能够稳定下降。
- 缺点：更新幅度过慢，需要更多次的迭代，降低程序的性能。可能在达到最大迭代次数时依然没有靠近极值点。

```python
# 因会产生大量输出，这里将verbose设置为False。
x_list, y_list = grad_descent(fun, grad_f, x_init=10, eta=0.005, verbose=False)
plot_result(fun, domain, x_list, y_list)
```



课堂练习

使用梯度法求解函数极值中，梯度为我们提供什么信息？
A 每次更新的方向。
B 每次更新值的大小。
C A与B。

二元函数求极值

使用梯度下降求解方程$y = 0.2(x_1 + x_2)^2 - 0.3x_1x_2 + 0.4$的最小值。

编写程序

```
from mpl_toolkits.mplot3d import Axes3D
%matplotlib qt


def fun(x1, x2):
    """定义二元函数。

    Parameters
    ----------
    x1, x2 : ndarray
        自变量。

    Returns
    -------
    y : ndarray
        根据自变量x1与x2，返回其所对应的y值。
    """
    return 0.2 * (x1 + x2) ** 2 - 0.3 * x1 * x2 + 0.4


def grad_f_x1(x1, x2):
    """定义原函数对x1的偏导函数。

    Parameters
    ----------
    x1, x2 : ndarray
        自变量。

    Returns
    -------
    y : ndarray
        根据自变量x1与x2，返回其所对应的y值。
    """
    return 0.4 * (x1 + x2) - 0.3 * x2


def grad_f_x2(x1, x2):
    """定义原函数对x2的偏导函数。

    Parameters
    ----------
    x1, x2 : ndarray
        自变量。

    Returns
    -------
    y : ndarray
        根据自变量x1与x2，返回其所对应的y值。
    """
    return 0.4 * (x1 + x2) - 0.3 * x1


def grad_descent(f, g, x_init, eta=0.1, tol=1e-4, max_iter=100, verbose=True):
    """通过梯度下降求解二元函数的极小值。

    根据给定的函数f与初始点，通过在循环中计算梯度，并更新初始点位置，
    从而逐渐接近函数的极小值。

    Parameters
    ----------
    f : funciton
        原函数。
    g : tuple, shape=(2, )
        f对x1与x2的偏导函数。
    x_init : tuple, shape=(2, )
        自变量初始点的位置（值）。
    eta : float, default=0.1
        学习率。控制梯度下降中x_init的更新幅度。
    tol : float, default=1e-4
        容忍值。当两次函数值的差值小于tol时，停止迭代。
    max_iter : int, default=100
        最大迭代次数。当达到最大迭代次数后，停止迭代。
    verbose : bool, default=True。
```

```python
        是否打印更新过程中的轨迹信息。

    Returns
    -------
    x_list : list
        自变量x1与x2的更新轨迹。
    y_list : list
        函数值y的更新轨迹。
    """
    # 定义x_list与y_list，用来保存梯度下降过程中，自变量x与函数值y的更新轨迹。
    x_list = []
    y_list = []
    x1, x2 = x_init
    g1, g2 = g
    for i in range(max_iter):
        x_list.append((x1, x2))
        y = f(x1, x2)
        y_list.append(y)
        # 如果在两次迭代中，函数y比上一次的减少值小于tol，则停止迭代。
        if len(y_list) >= 2 and y_list[-2] - y < tol:
            break
        x1 -= eta * g1(x1, x2)
        x2 -= eta * g2(x1, x2)
    if verbose:
        for index, (a, b) in enumerate(zip(x_list, y_list), start=1):
            print(f"第{index}次迭代，x={a}, y={b}")
    return x_list, y_list


x_list, y_list = grad_descent(fun, g=(grad_f_x1, grad_f_x2), x_init=(4.8, 4.5), eta=0.4)
```

```
第1次迭代，x=(4.8, 4.5), y=11.218000000000002
第2次迭代，x=(3.8519999999999994, 3.62592), y=7.393744353280001
第3次迭代，x=(3.0906431999999993, 2.9221470719999996), y=4.921335177768551
第4次迭代，x=(2.4792544051199994, 2.3554333642751994), y=3.322925602204132
第5次迭代，x=(1.9883563657297916, 1.899029771361976), y=2.2895698153912587
第6次迭代，x=(1.5942581563585458, 1.5314146816897178), y=1.6215250339872198
第7次迭代，x=(1.2779202640735898, 1.2352715220564194), y=1.1896530378735917
第8次迭代，x=(1.0240421609395587, 0.99666639208981), y=0.9104640894897645
第9次迭代，x=(0.8203287595056369, 0.8043866189752149), y=0.729981569022374
第10次迭代，x=(0.6569006932257264, 0.6494087322101514), y=0.6133097490838161
第11次迭代，x=(0.5258202330212041, 0.524470525735679), y=0.5378889713694954
第12次迭代，x=(0.42071017470838423, 0.42372683462963495), y=0.4891349153602483
第13次迭代，x=(0.33644747336985736, 0.34247264215409906), y=0.4576192881072987
第14次迭代，x=(0.26891697194451625, 0.2769203405316626), y=0.43724710050455784
第15次迭代，x=(0.21481344281212714, 0.2240205483341115), y=0.42407826678257415
第16次迭代，x=(0.17148247002882233, 0.18131796179950077), y=0.4155657733546593
第17次迭代，x=(0.13679255635223073, 0.1468353856574914), y=0.4100631655676179
第18次迭代，x=(0.10903233190957415, 0.11898043067590981), y=0.4065061498379129
第19次迭代，x=(0.08682794157700588, 0.096470444104684), y=0.4042067606133l395
第20次迭代，x=(0.06907665316049758, 0.07827210692151465), y=0.40272029886495364
第21次迭代，x=(0.054893504377957376, 0.063552829638954), y=0.4017593155488039
第22次迭代，x=(0.04356843049192603, 0.05164163967704432), y=0.40113800793568855
第23次迭代，x=(0.03453181602613609, 0.041997704687671784), y=0.40073627640459863
第24次迭代，x=(0.027326817274447447, 0.034184999246666396), y=0.40047649054596324
第25次迭代，x=(0.0215871265406692, 0.027851914305573006), y=0.400308470912405
第26次迭代，x=(0.017019109721939207, 0.02271484362780376), y=0.40019978148495444
第27次迭代，x=(0.013387458421316783, 0.018544970310502486), y=0.40012945499525565
```

结果可视化

```python
def plot_result(f, x, x_li, y_li):
    """绘制可视化结果。

    绘制结果包括两张图，一张图为在三维空间中，点坐标的更新轨迹。
    另外一张图为二元函数等高线与坐标更新轨迹在底面的投影。

    Parameters
    ----------
    f : funciton
        原二元函数。
    x : tuple, shape=(2, 定义域取样数)
        x1与x2的定义域数组。
    x_li : list
        自变量x1与x2的更新轨迹。
    y_li : list
        自变量y的更新轨迹。
    """

```

```
    x1, x2 = x
    X1, X2 = np.meshgrid(x1, x2)
    X = np.array([X1.ravel(), X2.ravel()]).T
    y = f(X[:, 0], X[:, 1])
    Y = y.reshape(X1.shape)
    fig = plt.figure()
    ax = Axes3D(fig)
    fig.add_axes(ax)
    surf = ax.plot_surface(X1, X2, Y, rstride=5, cstride=5, cmap="rainbow", alpha=0.8)
    # 将自变量的轨迹列表转换为ndarray数组，方便按列单独提取x1与x2的轨迹。
    array = np.asarray(x_li)
    x1_trace = array[:, 0]
    x2_trace = array[:, 1]
    ax.plot(x1_trace, x2_trace, y_list, c="b", ls="--", marker="o")
    ax.set_title("函数$y = 0.2(x1 + x2) ^ {2} - 0.3x1x2 + 0.4$")
    # 绘制三维图时，参数必须是数组类型，不支持标量，这点与绘制二维图不同。
    ax.plot(x1_trace[0:1], x2_trace[0:1], y_list[0:1], marker="*", ms=15, c="b", label="初始点")
    ax.plot(x1_trace[-1:], x2_trace[-1:], y_list[-1:], marker="*", ms=15, c="g", label="终止点")
    # 为曲面对象增加颜色条。
    fig.colorbar(surf)
    ax.legend()
    # 创建新的画布，用来绘制等高线图。
    fig2 = plt.figure()
    ax2 = fig2.gca()
    # 绘制填充等高线图。
    # 参数 X，Y，Z，levels
    # X，Y：平面点坐标。
    # Z：等高线值，会将所有Z值相同的点（x，y）绘制成等高线。
    # levels：控制等高线的疏密程度，值越大，等高线越密。
    m = ax2.contourf(X1, X2, Y, 10)
    ax2.scatter(x1_trace, x2_trace, c="r")
    # 为等高线图增加颜色条。
    fig2.colorbar(m)
    ax2.set_title("轨迹更新投影图")
    plt.show()


x1_domain = np.arange(-5, 5, 0.1)
x2_domain = np.arange(-5, 5, 0.1)
plot_result(fun, (x1_domain, x2_domain), x_list, y_list)
```

梯度与损失函数

求解损失函数最小值

结合之前的介绍，我们完全可以使用梯度下降的方法来针对损失函数求解极小值。

对于损失函数：

$$J(w) = \frac{1}{2}\sum_{i=1}^{m}(y^{(i)} - w^T x^{(i)})^2$$

我们可以使用梯度下降的方式，不断去调整权重w，进而去减小损失函数$J(w)$的值。经过不断迭代，最终求得最优的权重w，使得损失函数的值最小（近似最小）。调整方式为：

$$w_j = w_j - \eta\frac{\partial J(w)}{\partial w_j}$$

我们先对单个样本（第i个样本）的损失函数求梯度。

$$\frac{\partial J(w)}{\partial w_j} = \frac{\partial}{\partial w_j} \frac{1}{2} (y^{(i)} - w^T x^{(i)})^2$$
$$= 2 * \frac{1}{2} * (y^{(i)} - w^T x^{(i)}) \frac{\partial}{\partial w_j} (y^{(i)} - w^T x^{(i)})$$
$$= (y^{(i)} - w^T x^{(i)}) \frac{\partial}{\partial w_j} (y^{(i)} - \sum_{j=1}^{n} w_j x_j^{(i)})$$
$$= -(y^{(i)} - w^T x^{(i)}) x_j^{(i)}$$
$$= -(y^{(i)} - \hat{y}^{(i)}) x_j^{(i)}$$

对于所有样本的损失函数求梯度，只需要每个样本的梯度值相加即可。

$$\frac{\partial J(w)}{\partial w_j} = \frac{\partial}{\partial w_j} \frac{1}{2} \sum_{i=1}^{m} (y^{(i)} - w^T x^{(i)})^2$$
$$= -\sum_{i=1}^{m} (y^{(i)} - \hat{y}^{(i)}) x_j^{(i)}$$

梯度下降分类

根据权重更新的方式不同，可以将梯度下降分为三类：

- 随机梯度下降（SGD-Stochastic Gradient Descent）
- 批量梯度下降（BGD-Batch Gradient Descent）
- 小批量梯度下降（MBGD-Mini-Batch Gradient Descent）

随机梯度下降

随机梯度下降每次使用一个训练样本更新权重，样本选择的方式可以是：

- 按顺序选择。
- 洗牌之后，按顺序选择。
- 每次随机选择一个样本。

$$w_j = w_j + \eta (y^{(i)} - \hat{y}^{(i)}) x_j^{(i)}$$

批量梯度下降

批量梯度下降使用所有样本来更新权重。

$$w_j = w_j + \frac{1}{m} \eta \sum_{i=1}^{m} (y^{(i)} - \hat{y}^{(i)}) x_j^{(i)}$$

小批量梯度下降

小批量梯度下降每次使用一个批次的样本更新数据，样本批次数量为k。

- 当$k = 1$时，等同于随机梯度下降。
- 当$k = m$时，等同于批量梯度下降。

$$w_j = w_j + \frac{1}{k} \eta \sum_{i=1}^{k} (y^{(i)} - \hat{y}^{(i)}) x_j^{(i)}$$

三种梯度下降的对比

随机梯度下降

优点：

- 更新速度较快。
- 适合大型数据集。
- 可能会越过局部最小值，得到全局最优解。

缺点：

- 更新方向上不够稳定。
- 学习率（η）调整较为困难。

批量梯度下降

优点：

- 更新方向稳定。
- 可以并行化计算梯度。

缺点：

- 更新速度较慢。
- 不适合大型数据集。
- 容易陷入局部最小值。

小批量梯度下降

优点：

- 兼具随机梯度下降与批量梯度下降的优点。
  - 更新速度适中。
  - 更新方向稳定性适中。
  - 可以在一定程度上并行化计算梯度。

缺点：

- 需要事先指定批次的样本数量（batch size）。

对比

对比内容	随机梯度下降（SGD）	批量梯度下降（BGD）	小批量梯度下降（MBGD）
更新速度	快	慢	适中
逃离局部最小值	概率高	概率低	概率适中
大数据集	适用	不适用	适用
收敛稳定性	低	高	适中
并行化计算	不适用	适用	适用
学习率（η）调整	较为困难	容易	适中

应用

- 随机梯度下降
  - 大规模机器学习，其中数据集太大而无法放入内存。
  - 在线学习，随着新数据的出现，模型不断更新。
- 批量梯度下降
  - 小型数据集，注重收敛的稳定性。
- 小批量梯度下降
  - 需要收敛速度与稳定性兼顾。
  - 广泛应用于深度学习中。

sklearn实现随机梯度下降

```python
from sklearn.linear_model import SGDRegressor
from sklearn.datasets import make_regression

X, y, coef = make_regression(n_samples=1000, n_features=5, bias=2.5, coef=True,
        noise=5, random_state=0)
print(f"真实权重：{coef}")
# eta0 初始学习率。
sgd = SGDRegressor(eta0=0.2)
sgd.fit(X, y)
print(f"预测权重：{sgd.coef_}")
print(f"预测截距：{sgd.intercept_}")
print(f"R^2值：{sgd.score(X, y)}")
```

```
真实权重：[41.20593377 66.49948238 10.71453179 60.19514224 25.96147771]
预测权重：[41.10932215 66.34876076 10.80733056 59.92678553 25.76216547]
预测截距：[2.84770799]
R^2值：0.9974065812517856
```

课堂练习

对于LinearRegression与SGDRegressor，多次运行同一个程序，拟合出来的权重会相同吗?

A 二者都相同。

B 二者都不同。

C 前者相同，后者可能不同。

D 前者可能不同，后者相同。

数据标准化

在很多机器学习算法中，如果数据集的特征之间不是同一个数量级（量纲），则数量级大的特征就可能会成为模型函数的主要影响者，从而让评估器无法从其他特征中学习信息，导致训练的效果较差。

拟合对比

我们以波士顿房价数据集为例，来观察线性回归与梯度下降的表现。

```python
# 从scikit-learn 1.2起，删除波士顿房价数据集。
# from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression, SGDRegressor

data = pd.read_csv("house.csv")
data.head()
```

```css
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MI
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98	24
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14	21
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03	34
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94	33
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33	36

```python
X, y = data.iloc[:, :-1], data["MEDV"]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=0)
lr = LinearRegression()
lr.fit(X_train, y_train)
print("线性回归拟合分值：")
print(lr.score(X_train, y_train))
print(lr.score(X_test, y_test))
sgd = SGDRegressor(eta0=0.01)
sgd.fit(X_train, y_train)
print("随机梯度下降拟合分值：")
print(sgd.score(X_train, y_train))
print(sgd.score(X_test, y_test))
```

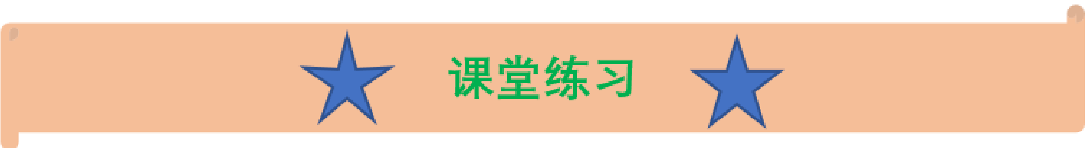
```
线性回归拟合分值：
0.7697699488741149
0.6354638433202124
随机梯度下降拟合分值：
-3.0665806224431205e+26
-3.270731824314644e+26
```

由结果得知，梯度下降远不及线性回归的效果好，然而，这并不表示梯度下降真的很差，仅仅是因为梯度下降需要对训练数据进行标准化而已。

```python
from sklearn.model_selection import train_test_split
from sklearn.linear_model import SGDRegressor
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, MinMaxScaler


pipeline = Pipeline([("ss", StandardScaler()), ("sgd", SGDRegressor(eta0=0.01))])
# pipeline = Pipeline([("ss", MinMaxScaler((-1, 1))), ("sgd", SGDRegressor(eta0=0.01))])
pipeline.fit(X_train, y_train)
print(pipeline.score(X_train, y_train))
print(pipeline.score(X_test, y_test))
```

```
0.7686432426643501
0.6292487006442433
```



在使用梯度下降求解极值前，一定要对训练数据进行标准化，这种说法正确吗？
A 正确
B 错误

作业

- 改写本节梯度下降的程序，不传递梯度函数，来实现同样的功能。
  - 提示：通过数值微分，自行计算梯度。
- 参考SGDRegressor，编写程序，实现自己的梯度下降类。
  - 随机梯度下降。
  - 批量梯度下降。
  - 小批量梯度下降。