一、实验内容

基于教材相关基础代码，自行设计对比实验，通过对实验结果分析回答：

当我们执行带冲量法的随机梯度下降时会有什么变化？当我们使用带冲量法的小批量随机梯度下降时会发生什么？

二、实验摘要

本次实验分别设计了两组对比实验：

第一组对比带冲量法的随机梯度下降算法和普通的随机梯度下降算法的优化结果，得出结论：使用带有冲量的随机梯度下降相比于普通的随机梯度下降，模型参数的更新更加平稳，收敛速度更快，同时也减少了震荡。

第二组对比带冲量法的小批量随机梯度下降算法和普通的小批量随机梯度下降算法的优化结果，得出结论：使用冲量法的小批量梯度下降的收敛速度更快，下降更新参数也更加平滑稳定。

三、实验内容

1、简单介绍随机梯度下降算法、小批量随机梯度下降算法、冲量法

随机梯度下降（Stochastic Gradient Descent, SGD）是一种常用的优化算法，用于在机器学习和深度学习中训练模型。它的核心思想是通过迭代更新模型参数，使损失函数最小化。SGD 每次迭代只使用一个样本来计算梯度，并更新参数。其主要步骤包括计算梯度、更新参数和重复迭代，直到达到停止条件。

小批量随机梯度下降（Mini-batch Stochastic Gradient Descent）是对随机梯度下降的一种改进。与 SGD 每次只使用一个样本进行更新不同，小批量随机梯度下降每次使用一小批样本（通常为几十到几百个）来计算梯度并更新参数。这样做的好处是可以利用矩阵运算的并行性，提高计算效率，并且减少了参数更新的方差，使得更新更加稳定。小批量随机梯度下降在实践中更常用，因为它在计算效率和参数更新稳定性之间取得了一个折中。

冲量法（Momentum）是一种在梯度下降中使用动量的技巧，旨在加快模型的训练速度。冲量法基于模拟物体在运动过程中的动量概念。它引入了一个动量变量，用于累积历史梯度，并在更新参数时参与计算。冲量法的核心思想是在更新参数时，除了考虑当前的梯度，还考虑历史梯度的方向和大小。这样可以减少参数更新的方差，使得参数更新更加平滑稳定，并且有助于跳出局部最小值，加速收敛速度。冲量法通过引入动量参数来控制历史梯度的权重，典型的冲量参数取值范围在 0 到 1 之间。

2、设计对比实验

a、带冲量法的随机梯度下降算法和普通的随机梯度下降算法

Part 1 对比带冲量法和不带的随机梯度下降算法

随机梯度下降

随机梯度下降是一种高效的优化算法，通过迭代地使用随机选择的样本来更新模型参数，以达到最小化损失函数的目标。

随机梯度下降与传统的梯度下降（Gradient Descent）的区别在于，每次迭代中只使用一个样本来计算梯度，而不是使用所有样本。这样可以减少计算开销，加快训练速度。但同时，由于随机选择的样本可能具有噪声或偏差，因此梯度的估计可能不够准确。为了解决这个问题，通常会引入一些改进的随机梯度下降算法，如小批量随机梯度下降（Mini-batch SGD）和带动量的随机梯度下降（Momentum SGD）。

In [39]:
```
%matplotlib inline
import math
import torch
from d2l import torch as d2l
```

平均而言，随机梯度是对梯度的良好估计。

现在，我们将把它与梯度下降进行比较，方法是向梯度添加均值为0、方差为1的随机噪声，以模拟随机梯度下降。

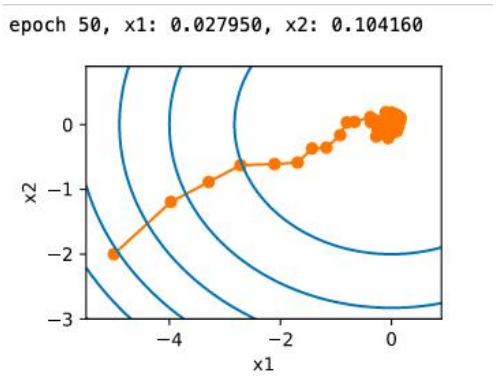
In [47]:
```
def f(x1, x2):  # 目标函数
    return x1 ** 2 + 2 * x2 ** 2

def f_grad(x1, x2):  # 目标函数的梯度
    return 2 * x1, 4 * x2

def sgd(x1, x2, s1, s2, f_grad):  #随机梯度下降的更新函数
    g1, g2 = f_grad(x1, x2)

    # 模拟有噪声的梯度
    g1 += torch.normal(0.0, 1, (1,))
    g2 += torch.normal(0.0, 1, (1,))

    eta_t = eta * lr()  # 学习率 eta 学习速率 lr()
    return (x1 - eta_t * g1, x2 - eta_t * g2, 0, 0)  # 模型参数 x1 和 x2 更新

def constant_lr():  # 常数学习速度
    return 1

eta = 0.1
lr = constant_lr  # 常数学习速度
d2l.show_trace_2d(f, d2l.train_2d(sgd, steps=50, f_grad=f_grad))
```

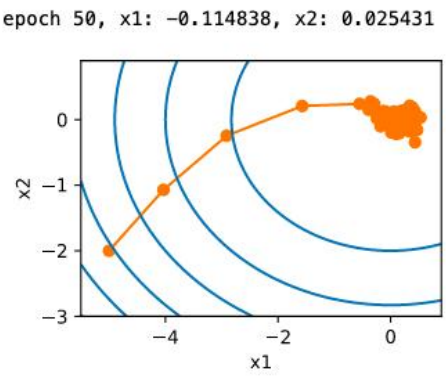
```
epoch 50, x1: 0.027950, x2: 0.104160
```

带冲量法的随机梯度下降

In [67]:
```
def momentum_sgd(x1, x2, v1, v2, f_grad):  # 带冲量法的随机梯度下降的更新函数
    g1, g2 = f_grad(x1, x2)

    # 模拟有噪声的梯度
    g1 += torch.normal(0.0, 1, (1,))
    g2 += torch.normal(0.0, 1, (1,))

    eta_t = eta * lr()  # 学习率 eta 学习速率 lr()
    # 动量项的更新
    v1 = beta * v1 +  g1
    v2 = beta * v2 +  g2
    return (x1 - eta_t * v1, x2 - eta_t * v2, v1,v2)  # 模型参数 x1 和 x2 更新


beta = 0.5  # 动量参数
d2l.show_trace_2d( f,d2l.train_2d(momentum_sgd, steps=50, f_grad=f_grad))
```

```
epoch 50, x1: -0.114838, x2: 0.025431
```





第一组对比实验结论:

可以观察到，使用带有冲量的随机梯度下降 (右图) 相比于普通的随机梯度下降 (左图)，模型参数的更新更加平稳，收敛速度更快，同时也减少了震荡。具体分析如下：

1、收敛速度：带有冲量的随机梯度下降通常具有更快的收敛速度。这是因为冲量可以利用历史梯度的信息来调整参数更新的方向和步长，从而更有效地朝向全局最小值或优化目标的方向前进。相比之下，普通的随机梯度下降只依赖当前的梯度信息进行更新，可能会在优化过程中出现震荡或在平坦区域中徘徊，导致收敛速度较慢。

2、平稳性和鲁棒性：带有冲量的随机梯度下降在参数更新过程中更平稳和鲁棒。冲量的引入可以减少参数更新的方差，使得参数更新更加平滑。这有助于稳定优化过程，减少震荡和梯度爆炸的问题。相比之下，普通的随机梯度下降可能会因为梯度的变化剧烈而导致参数更新不稳定，需要更小的学习率或其他技巧来控制优化过程的稳定性。

b、带冲量法的小批量随机梯度下降算法和普通的小批量随机梯度下降算法

在进行对比实验前，通过使用 PyTorch 库计算矩阵乘法的不同方法，并通过计时器对象测量它们的性能。解释为什么读取数据的小批量，而不是观测单个数据来更新参数：处理单个观测值需要我们执行许多单一矩阵-矢量（甚至矢量-矢量）乘法，这耗费相当大，而且对应深度学习框架也要巨大的开销。这既适用于计算梯度以更新参数时，也适用于用神经网络预测。

Part 2 对比带冲量法和不带的小批量梯度下降算法

下述代码通过使用PyTorch库计算矩阵乘法的不同方法，并通过计时器对象测量它们的性能。解释了为什么读取数据的小批量，而不是观测单个数据来更新参数：处理单个观测值需要我们执行许多单一矩阵–矢量（甚至矢量–矢量）乘法，这耗费相当大，而且对应深度学习框架也要巨大的开销。这既适用于计算梯度以更新参数时，也适用于用神经网络预测。

In [118]:
```
%matplotlib inline
import numpy as np
import torch
from torch import nn
from d2l import torch as d2l

timer = d2l.Timer()  # 创建了一个计时器对象timer，用于测量代码执行的时间。

#创建了256x256的全零张量ABC。
A = torch.zeros(256, 256)
B = torch.randn(256, 256)
C = torch.randn(256, 256)
```

逐元素计算A=BC，这种计算方式效率较低，适用于特定需要逐元素计算的情况。

In [93]:
```
# 逐元素计算A=BC
timer.start()
for i in range(256):
    for j in range(256):
        A[i, j] = torch.dot(B[i, :], C[:, j])
timer.stop()
```

Out[93]: 2.737057685852051

click to scroll output; double click to hide

逐列计算A=BC，使用torch.mv函数计算B和C[:, j]的矩阵-向量乘法，并将结果赋值给A[:, j]。

这种计算方式在某些情况下比逐元素计算更高效，但仍然需要循环计算每一列。

In [94]:
```
# 逐列计算A=BC
timer.start()
for j in range(256):
    A[:, j] = torch.mv(B, C[:, j])
timer.stop()
```

Out[94]: 0.08802604675292969

一次性计算A=BC，torch.mm函数实现了矩阵乘法的高效计算，性能通常比逐元素或逐列计算更好。

In [95]:
```
# 一次性计算A=BC
timer.start()
A = torch.mm(B, C)
timer.stop()
```

Out[95]: 0.012260675430297852

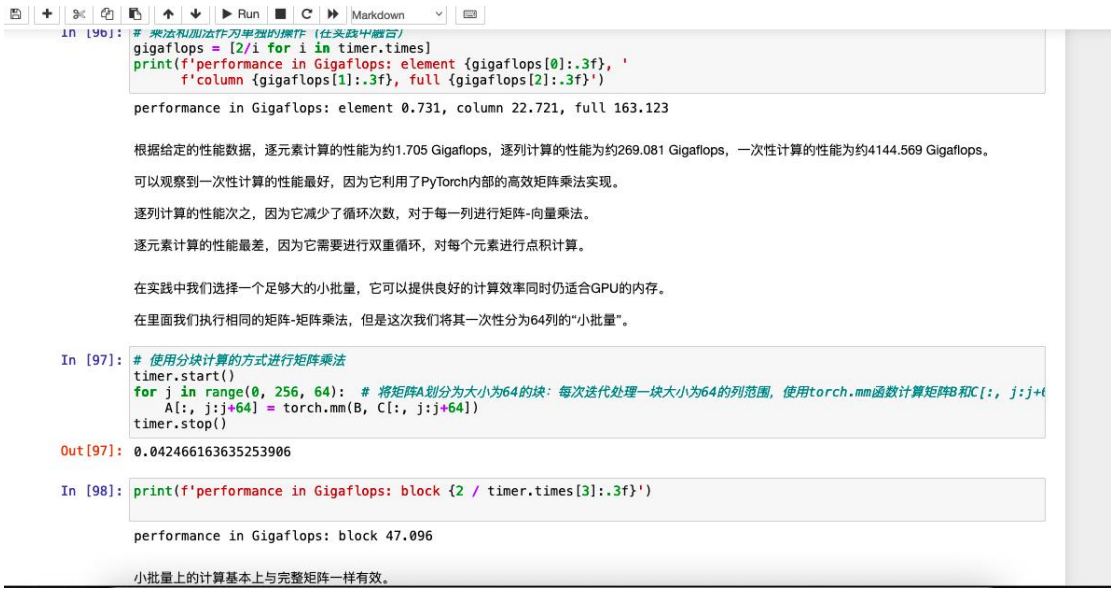
计算每种计算方式的性能，并以Gigaflops为单位打印输出。

计算性能使用计时器对象记录的时间，并将每种计算方式所需的浮点操作数（FLOPs）除以时间得到每秒的Gigaflops数值。

较高的Gigaflops值表示计算性能更好。

根据给定的性能数据，逐元素计算的性能为约 1.705 Gigaflops，逐列计算的性能为约 269.081 Gigaflops，一次性计算的性能为约 4144.569 Gigaflops。
可以观察到一次性计算的性能最好，因为它利用了 PyTorch 内部的高效矩阵乘法实现。
逐列计算的性能次之，因为它减少了循环次数，对于每一列进行矩阵-向量乘法。
逐元素计算的性能最差，因为它需要进行双重循环，对每个元素进行点积计算。

下面在实践中我们选择一个足够大的小批量，它可以提供良好的计算效率同时仍适合 GPU 的内存。 在里面我们执行相同的矩阵-矩阵乘法，但是这次我们将其一次性分为 64 列的“小批量”。



小批量随机梯度下降

通过使用小批量数据进行训练，可以减小内存占用，并且可以更好地利用硬件加速器（如GPU）的并行计算能力。此外，小批量数据还可以提供更好的梯度估计，从而更准确地更新模型的参数，提高训练效果

添加了一个状态输入states并将超参数放在字典hyperparams中。

在训练函数里对各个小批量样本的损失求平均，因此优化算法中的梯度不需要除以批量大小。

In [101]:
```
def sgd(params, states, hyperparams):
    for p in params:
        p.data.sub_(hyperparams['lr'] * p.grad)
        p.grad.data.zero_()
```

读取数据集

In [102]:
```
#@save
d2l.DATA_HUB['airfoil'] = (d2l.DATA_URL + 'airfoil_self_noise.dat',
                           '76e5be1548fd8222e5074cf0faae75edff8cf93f')

#@save
def get_data_ch11(batch_size=10, n=1500):
    data = np.genfromtxt(d2l.download('airfoil'),
                         dtype=np.float32, delimiter='\t')
    data = torch.from_numpy((data - data.mean(axis=0)) / data.std(axis=0))
    data_iter = d2l.load_array((data[:n, :-1], data[:n, -1]),batch_size, is_train=True)
    # 使用d2l.load_array函数将特征数据data[:n, :-1]和标签数据data[:n, -1]打包成数据迭代器。batch_size参数指定了每个小批量的样本数
    return data_iter, data.shape[1]-1
```

训练函数train_ch11，用于在PyTorch中训练一个简单的线性回归模型,然后可以使用小批量随机梯度下降来训练模型。

In [103]:
```
#@save
def train_ch11(trainer_fn, states, hyperparams, data_iter,feature_dim, num_epochs=2):

    # 初始化模型
    w = torch.normal(mean=0.0, std=0.01, size=(feature_dim, 1),requires_grad=True)
    b = torch.zeros((1), requires_grad=True)
    net, loss = lambda X: d2l.linreg(X, w, b), d2l.squared_loss

    # 训练模型
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',xlim=[0, num_epochs], ylim=[0.22, 0.35])
    n, timer = 0, d2l.Timer()
    for _ in range(num_epochs):
        for X, y in data_iter:
            l = loss(net(X), y).mean()  # 对于每个小批量样本，计算损失时使用了.mean()函数对损失进行了求平均
            l.backward()
            trainer_fn([w, b], states, hyperparams)
            n += X.shape[0]
            if n % 200 == 0:
                timer.stop()
                animator.add(n/X.shape[0]/len(data_iter),
                             (d2l.evaluate_loss(net, data_iter, loss),))
                timer.start()
    print(f'loss: {animator.Y[0][-1]:.3f}, {timer.avg():.3f} sec/epoch')
    return timer.cumsum(), animator.Y[0]
```

使用随机梯度下降（SGD）优化算法进行训练的函数train_sgd，并调用train_ch11函数进行训练。

使用随机梯度下降（SGD）优化算法进行训练的函数train_sgd，并调用train_ch11函数进行训练。

In [106]:
```
def train_sgd(lr, batch_size, num_epochs=2):
    data_iter, feature_dim = get_data_ch11(batch_size)
    return train_ch11(
        sgd, None, {'lr': lr}, data_iter, feature_dim, num_epochs)
```

当批量大小等于100，学习率等于0.1时，使用小批量随机梯度下降进行优化。

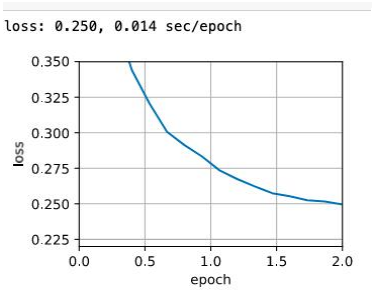
In [116]:
```
mini1_res = train_sgd(.1, 100)
```

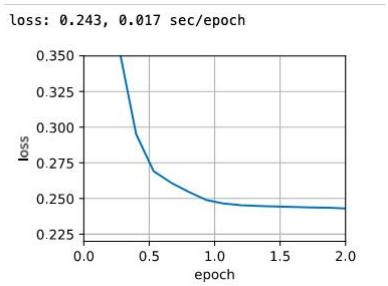
```
loss: 0.250, 0.014 sec/epoch
```

现在在批量大小等于100时，学习率0.1的小批量随机梯度下降算法基础上，添加冲量法，观察训练效果。

In [117]:
```
def momentum_sgd(params, states, hyperparams):
    for p, v in zip(params, states):
        v.data = hyperparams['momentum'] * v.data + hyperparams['lr'] * p.grad  # 使用动量更新状态信息
        p.data.sub_(v.data)  # 使用状态信息更新参数
        p.grad.data.zero_()  # 清零参数的梯度

# 使用动量随机梯度下降进行训练的函数
def train_momentum_sgd(lr, momentum, batch_size, num_epochs=2):
    data_iter, feature_dim = get_data_ch11(batch_size)

    # 初始化模型参数
    w = torch.normal(mean=0.0, std=0.01, size=(feature_dim, 1),requires_grad=True)
    b = torch.zeros((1), requires_grad=True)

    # 函数调用train_ch11函数进行训练
    return train_ch11(
        momentum_sgd, [torch.zeros_like(p) for p in [w, b]],
        {'lr': lr, 'momentum': momentum}, data_iter, feature_dim, num_epochs)

# 调用train_momentum_sgd函数进行训练，学习率为0.4、动量参数为0.5、小批量样本数量为100
momentum_res = train_momentum_sgd(0.1, 0.5, 100)
```

```
loss: 0.243, 0.017 sec/epoch
```





第二组对比实验结论：

对比带冲量法（右图）和不带的小批量梯度下降（左图）训练结果：

1、使用冲量法的小批量梯度下降的收敛速度更快，这是因为动量可以帮助加速收敛过程，特别是在存在平坦区域或者局部最小值附近时。它利用历史梯度的方向和大小来调整参数更新的方向和步长，从而更有效地向全局最小值或优化目标靠近。

2、使用冲量法的小批量梯度下降更新参数更加平滑稳定，这是因为动量的引入可以减少参数更新的方差，使得参数更新更加平滑。这有助于稳定优化过程，减少震荡和梯度爆炸的问题。使用冲量法的小批量梯度下降拥有适应性学习率，动量可以自适应地调整学习率。当梯度变化剧烈时，动量较大，可以提高学习率，加快收敛速度；而当梯度变化较小时，动量较小，可以减小学习率，增加稳定性。