

本笔记是针对《动手学深度学习》一书的PyTorch实现

吴振宇

QQ:807698462

最后修改：2021年1月20日

版本号：V1.0

# DIVE INTO DEEP LEARNING

## 目录

### 第1章 预备知识

- 1.1 数据操作
  - 1.1.1 创建Tensor
  - 1.1.2 运算
  - 1.1.3 广播机制
  - 1.1.4 索引
  - 1.1.5 运算的内存开销
  - 1.1.6 Tensor和NumPy相互变换

- 1.2 自动求梯度

- 1.2.1 简单例子

- 1.3 查阅文档

- 1.3.1 查找模块里所有函数和类
  - 1.3.2 查找特定函数和类的使用
  - 1.3.3 在PyTorch网站上查阅

- 1.4 本章附录

### 第2章 深度学习基础

- 2.1 线性回归

- 2.1.1 线性回归的基本要素
  - 2.1.2 线性回归的表示方法

- 2.2 线性回归从零开始实现

- 2.2.1 生成数据集
  - 2.2.2 读取数据集
  - 2.2.3 初始化模型参数
  - 2.2.4 定义模型
  - 2.2.5 定义损失函数
  - 2.2.6 定义优化算法

- 2.3 线性回归的简洁实现

- 2.3.1 生成数据集
  - 2.3.2 读取数据集
  - 2.3.3 定义模型
  - 2.3.4 初始化模型参数
  - 2.3.5 定义损失函数
  - 2.3.6 定义优化算法
  - 2.3.7 训练模型

- 2.4 softmax回归

- 2.4.1 分类问题
  - 2.4.2 softmax回归模型
  - 2.4.3 单样本分类的矢量计算表达式
  - 2.4.4 小批量样本分类的矢量计算表达式
  - 2.4.5 交叉熵损失函数
  - 2.4.6 模型预测及评价

- 2.5 图像分类数据集 (Fashion-MNIST)

- 2.5.1 获取数据集
  - 2.5.2 读取小批量

- 2.6 softmax回归的从零开始实现

- 2.6.1 读取数据集
  - 2.6.2 初始化模型参数
  - 2.6.3 实现softmax运算
  - 2.6.4 定义模型
  - 2.6.5 定义损失函数
  - 2.6.6 计算分类准确率
  - 2.6.7 训练模型

- 2.6.8 预测
- 2.7 softmax回归的简洁实现
  - 2.7.1 读取数据集
  - 2.7.2 定义和初始化模型
  - 2.7.3 softmax和交叉熵损失函数
  - 2.7.4 定义优化算法
  - 2.7.5 训练模型
- 2.8 多层感知机
  - 2.8.1 隐藏层
  - 2.8.2 激活函数
  - 2.8.3 多层感知机
- 2.9 多层感知机的从零开始实现
  - 2.9.1 读取数据
  - 2.9.2 定义模型参数
  - 2.9.3 定义激活函数
  - 2.9.4 定义模型
  - 2.9.5 定义损失函数
  - 2.9.6 训练模型
- 2.10 多层感知机的简洁实现
  - 2.10.1 定义模型
  - 2.10.2 训练模型
- 2.11 模型选择、欠拟合和过拟合
  - 2.11.1 训练误差和泛化误差
  - 2.11.2 模型选择
  - 2.11.3 欠拟合和过拟合
  - 2.11.4 多项式函数拟合实验
- 2.12 权重衰减
  - 2.12.1 方法
  - 2.12.2 高维线性回归实验
  - 2.12.3 从零开始实现
  - 2.12.4 简洁实现
- 2.13 丢弃法
  - 2.13.1 方法
  - 2.13.2 从零开始实现
  - 2.13.3 简洁实现
- 2.14 正向传播、反向传播和计算图
  - 2.14.1 正向传播
  - 2.14.2 正向传播的计算图
  - 2.14.3 反向传播
  - 2.14.4 训练深度学习模型
- 2.15 数值稳定性和模型初始化
  - 2.15.1 衰减和爆炸
  - 2.15.2 随机初始化模型参数
- 2.16 实战Kaggle比赛：房价预测
  - 2.16.1 Kaggle比赛
  - 2.16.2 读取数据集
  - 2.16.3 预处理数据集
  - 2.16.4 训练模型
  - 2.16.5 k折交叉验证
  - 2.16.6 模型选择
  - 2.16.7 预测并在Kaggle提交结果
- 2.17 本章附录

## 第3章 深度学习计算

- 3.1 模型构造
  - 3.1.1 继承Module类来构造模型
  - 3.1.2 Sequential类继承自Module类
  - 3.1.3 构造复杂的模型
- 3.2 模型参数的访问、初始化和共享

- 3.2.1 访问模型参数
- 3.2.2 初始化模型参数
- 3.2.3 自定义初始化方法
- 3.2.4 共享模型参数
- 3.3 自定义层
  - 3.3.1 不含模型参数的自定义层
  - 3.3.2 含模型参数的自定义层
- 3.4 读取和存储
  - 3.4.1 读写Tensor
  - 3.4.2 读写模型的参数
- 3.5 GPU计算
  - 3.5.1 计算设备
  - 3.5.2 Tensor的GPU计算
  - 3.5.3 模型的GPU计算
- 3.6 本章附录

## 第4章 卷积神经网络

- 4.1 二维卷积层
  - 4.1.1 二维互相关运算
  - 4.1.2 二维卷积层
  - 4.1.3 图像中物体边缘检测
  - 4.1.4 通过数据学习核数组
  - 4.1.5 互相关运算和卷积运算
  - 4.1.6 特征图和感受野
- 4.2 填充和步幅
  - 4.2.1 填充
  - 4.2.2 步幅
- 4.3 多输入通道和多输出通道
  - 4.3.1 多输入通道
  - 4.3.2 多输出通道
  - 4.3.3  $1 \times 1$  卷积层
- 4.4 池化层
  - 4.4.1 二维最大池化层和平均池化层
  - 4.4.2 填充和步幅
  - 4.4.3 多通道
- 4.5 卷积神经网络 (LeNet)
  - 4.5.1 LeNet模型
  - 4.5.2 训练模型
- 4.6 深度卷积神经网络 (AlexNet)
  - 4.6.1 学习特征表示
  - 4.6.2 AlexNet
  - 4.6.3 读取数据集
  - 4.6.4 训练模型
- 4.7 使用重复元素的网络 (VGG)
  - 4.7.1 VGG块
  - 4.7.2 VGG网络
  - 4.7.3 训练模型
- 4.8 网络中的网络 (NiN)
  - 4.8.1 NiN块
  - 4.8.2 NiN模型
  - 4.8.3 训练模型
- 4.9 含并行连结的网络 (GoogLeNet)
  - 4.9.1 Inception块
  - 4.9.2 GoogLeNet模型
  - 4.9.3 训练模型
- 4.10 批量归一化
  - 4.10.1 批量归一化层
  - 4.10.2 从零开始实现
  - 4.10.3 使用批量归一化层的LeNet

4.10.4 简洁实现

#### 4.11 残差网络 (ResNet)

4.11.1 残差块

4.11.2 ResNet模型

4.11.3 训练模型

#### 4.12 稠密连接网络 (DenseNet)

4.12.1 稠密块

4.12.2 过渡层

4.12.3 DenseNet模型

4.12.4 训练模型

#### 4.13 本章附录

### 第5章 循环神经网络

#### 5.1 语言模型

5.1.1 语言模型的计算

5.1.2 n元语法

#### 5.2 循环神经网络

5.2.1 不含隐藏状态的神经网络

5.2.2 含隐藏状态的循环神经网络

5.2.3 应用：基于字符级循环神经网络的语言模型

#### 5.3 语言模型数据集（歌词）

5.3.1 读取数据集

5.3.2 建立字符索引

5.3.3 时序数据的采样

#### 5.4 循环神经网络的从零开始实现

5.4.1 one-hot向量

5.4.2 初始化模型参数

5.4.3 定义模型

5.4.4 定义预测函数

5.4.5 裁剪梯度

5.4.6 困惑度

5.4.7 定义模型训练函数

5.4.8 训练模型并创作歌词

#### 5.5 循环神经网络的简洁实现

5.5.1 定义模型

5.5.2 训练模型

#### 5.6 通过时间反向传播

5.6.1 定义模型

5.6.2 模型计算图

5.6.3 方法

#### 5.7 门控循环单元 (GRU)

5.7.1 门控循环单元

5.7.2 读取数据集

5.7.3 从零开始实现

5.7.4 简洁实现

#### 5.8 长短期记忆 (LSTM)

5.8.1 长短期记忆

5.8.2 读取数据集

5.8.3 从零开始实现

5.8.4 简洁实现

#### 5.9 深度循环神经网络

#### 5.10 双向循环神经网络

#### 5.11 本章附录

### 第6章 优化算法

#### 6.1 优化与深度学习

6.1.1 优化与深度学习的关系

6.1.2 优化在深度学习中的挑战

#### 6.2 梯度下降和随机梯度下降

6.2.1 一维梯度下降

- 6.2.2 学习率
- 6.2.3 多维梯度下降
- 6.2.4 随机梯度下降
- 6.3 小批量随机梯度下降
  - 6.3.1 读取数据集
  - 6.3.2 从零开始实现
  - 6.3.3 简洁实现
- 6.4 动量法
  - 6.4.1 梯度下降的问题
  - 6.4.2 动量法
  - 6.4.3 从零开始实现
  - 6.4.4 简洁实现
- 6.5 AdaGrad算法
  - 6.5.1 算法
  - 6.5.2 特点
  - 6.5.3 从零开始实现
  - 6.5.4 简洁实现
- 6.6 RMSProp算法
  - 6.6.1 算法
  - 6.6.2 从零开始实现
  - 6.6.3 简洁实现
- 6.7 AdaDelta算法
  - 6.7.1 算法
  - 6.7.2 从零开始实现
  - 6.7.3 简洁实现
- 6.8 Adam算法
  - 6.8.1 算法
  - 6.8.2 从零开始实现
  - 6.8.3 简洁实现
- 6.9 本章附录

## 第7章 计算性能

- 7.1 命令式编程和符号式编程
- 7.2 自动并行计算
- 7.3 多GPU计算
  - 7.3.1 数据并行
  - 7.3.2 多GPU计算
  - 7.3.3 多GPU模型的保存与加载
- 7.4 本章附录

## 第8章 计算机视觉

- 8.1 图像增广
  - 8.1.1 常用的图像增广方法
  - 8.1.2 使用图像增广训练模型
- 8.2 微调
- 8.3 目标检测和边界框
- 8.4 锚框
  - 8.4.1 生成多个锚框
  - 8.4.2 交并比
  - 8.4.3 标注训练集的锚框
  - 8.4.4 输出预测边界框
- 8.5 多尺度目标检测
- 8.6 目标检测数据集（皮卡丘）
  - 8.6.1 读取数据集
  - 8.6.2 图示数据
- 8.7 单发多框检测（SSD）
  - 8.7.1 定义模型
  - 8.7.2 训练模型
  - 8.7.3 预测目标
- 8.8 区域卷积神经网络（R-CNN）系列

- 8.8.1 R-CNN
  - 8.8.2 Fast R-CNN
  - 8.8.3 Faster R-CNN
  - 8.8.4 Mask R-CNN
  - 8.9 语义分割和数据集
    - 8.9.1 图像分割和实例分割
    - 8.9.2 Pascal VOC2012语义分割数据集
  - 8.10 全卷积网络 (FCN)
    - 8.10.1 转置卷积层
    - 8.10.2 构造模型
    - 8.10.3 初始化转置卷积层
    - 8.10.4 读取数据集
    - 8.10.5 训练模型
    - 8.10.6 预测像素类别
  - 8.11 样式迁移
    - 8.11.1 方法
    - 8.11.2 读取内容图像和样式图像
    - 8.11.3 预处理和后处理图像
    - 8.11.4 抽取特征
    - 8.11.5 定义损失函数
    - 8.11.6 创建和初始化合成图像
    - 8.11.7 训练模型
  - 8.12 实战Kaggle比赛：图像分类 (CIFAR-10)
    - 8.12.1 获取和整理数据集
    - 8.12.2 图像增广
    - 8.12.3 读取数据集
    - 8.12.4 定义模型
    - 8.12.5 定义训练函数
    - 8.12.6 训练模型
    - 8.12.7 对测试集分类并在Kaggle提交结果
  - 8.13 实战Kaggle比赛：狗的品种识别 (ImageNet Dogs)
    - 8.13.1 获取和整理数据集
    - 8.13.2 图像增广
    - 8.13.3 读取数据集
    - 8.13.4 定义模型
    - 8.13.5 定义训练函数
    - 8.13.6 训练模型
    - 8.13.7 对测试集分类并在Kaggle提交结果
  - 8.14 语义分割网络 (U-Net)
    - 8.14.1 网络结构
    - 8.14.2 读取数据集
    - 8.14.3 定义模型
    - 8.14.4 定义训练函数
    - 8.14.5 训练模型
    - 8.14.6 预测像素类别
  - 8.15 本章附录
- ## 第9章 自然语言处理
- 9.1 词嵌入 (word2vec)
    - 9.1.1 为何不采用one-hot向量
    - 9.1.2 跳字模型
    - 9.1.3 连续词袋模型
  - 9.2 近似训练
    - 9.2.1 负采样
    - 9.2.2 层序softmax
  - 9.3 word2vec的实现
    - 9.3.1 预处理数据集
    - 9.3.2 负采样
    - 9.3.3 读取数据集

9.3.4	跳字模型
9.3.5	训练模型
9.3.6	应用词嵌入模型
9.4	子词嵌入 (fastText)
9.5	全局向量的词嵌入 (GloVe)
9.5.1	GloVe模型
9.5.2	从条件概率比值理解GloVe模型
9.6	求近义词和类比词
9.6.1	使用预训练的词向量
9.6.2	应用预训练词向量
9.7	文本情感分类：使用循环神经网络
9.7.1	文本情感分类数据集
9.7.2	使用循环神经网络的模型
9.8	文本情感分类：使用卷积神经网络 (textCNN)
9.8.1	一维卷积层
9.8.2	时序最大池化层
9.8.3	读取和预处理IMDb数据集
9.8.4	textCNN模型
9.9	编码器-解码器 (seq2seq)
9.9.1	编码器
9.9.2	解码器
9.9.3	训练模型
9.10	束搜索
9.10.1	贪婪搜索
9.10.2	穷举搜索
9.10.3	束搜索
9.11	注意力机制
9.11.1	计算背景变量
9.11.2	更新隐藏状态
9.11.3	发展
9.12	机器翻译
9.12.1	读取和预处理数据集
9.12.2	含注意力机制的编码器-解码器
9.12.3	训练模型
9.12.4	预测不定长的序列
9.12.5	评价翻译结果
9.13	本章附录

---

本书GitHub代码链接为：[Dive-Into-Deep-Learning-PyTorch-PDF](#)。

# 第1章 预备知识

## 1.1 数据操作

在深度学习中，我们通常会频繁地对数据进行操作。作为动手学深度学习的基础，本节将介绍如何对内存中的数据进行操作。

在PyTorch中，`torch.Tensor`是一种包含单一数据类型元素的多维矩阵，也是存储和变换数据的主要工具。为了简洁，本书常`Tensor`实例直接称作`Tensor`。如果你之前用过`NumPy`，你会发现`Tensor`和`NumPy`的多维数组非常类似。然而，`Tensor`提供GPU计算和自动求梯度等功能，这些使得`Tensor`更加适合深度学习。

1 | Tip: 我们通常将tensor译为张量，其实可以将它看作多维数组。标量为0维张量、向量为1维张量、矩阵为2维张量

## 1.1.1 创建Tensor

我们先介绍 Tensor 的最基本功能。如果对这里用到的数学操作不是很熟悉，可以参阅附录A。

首先导入PyTorch。

```
1 In [1]:  
2     import torch  
3     print("版本号为: {}".format(torch.__version__))  
4 Out [1]:  
5     版本号为: 1.1.0
```

然后我们用 arange 函数创建一个行向量。

```
1 In [2]:  
2     tensor_a = torch.arange(0, 12)  
3     tensor_a  
4 Out [2]:  
5     tensor([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

这时返回了一个 tensor 实例，其中包含了从0开始的12个连续整数。从打印tensor\_a时的返回结果可以看出，它是长度为12的一维数组。我们可以通过下述代码查看张量的存储位置：

```
1 In [3]:  
2     print("张量的存储位置: {}".format(tensor_a.device))  
3 Out [3]:  
4     张量的存储位置: cpu
```

从返回结果可以看出张量创建在CPU使用的内存上。

我们可以通过 shape 属性来获取 tensor 实例的形状。

```
1 In [4]:  
2     tensor_a.shape  
3 Out [4]:  
4     torch.Size([12])
```

我们也能够通过 numel() 方法得到 tensor 实例中元素 (element) 的总数。

```
1 In [5]:  
2     tensor_a.numel()  
3 Out [5]:  
4     12
```

下面使用 view 函数把行向量tensor\_a的形状改为 (3, 4)，也就是一个3行4列的矩阵，并记作 X (矩阵变量常用大写字母表示)。除了形状改变之外，X中的元素保持不变。

```
1 In [6]:  
2     x = tensor_a.view(3, 4)  
3     x  
4 Out [6]:  
5     tensor([[ 0,  1,  2,  3],  
6               [ 4,  5,  6,  7],  
7               [ 8,  9, 10, 11]])
```

注意，`X`属性中的形状发生了变化。上面`tensor_a.view(3, 4)`也可写成`tensor_a.view(3, -1)`或`tensor_a.view(-1, 4)`。由于`tensor_a`的元素个数是已知的，这里的-1是能够通过元素个数和其他维度的大小推断出来的。

接下来，我们创建一个各元素为0，形状为`(2, 3, 4)`的张量。实际上，之前创建的向量和矩阵都是特殊的张量。

```
1 In [7]:  
2     torch.zeros((2, 3, 4))  
3 Out [7]:  
4     tensor([[[0., 0., 0., 0.],  
5               [0., 0., 0., 0.],  
6               [0., 0., 0., 0.]],  
7  
8               [[[0., 0., 0., 0.],  
9                 [0., 0., 0., 0.],  
10                [0., 0., 0., 0.]]])
```

类似的，我们可以创建各元素为1的张量。

```
1 In [8]:  
2     torch.ones((3, 4))  
3 Out [8]:  
4     tensor([[1., 1., 1., 1.],  
5               [1., 1., 1., 1.],  
6               [1., 1., 1., 1.]])
```

我们也可以通过Python的列表（list）指定需要创建的`Tensor`中每个元素的值。

```
1 In [9]:  
2     Y = torch.tensor([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])  
3     Y  
4 Out [9]:  
5     tensor([[2, 1, 4, 3],  
6               [1, 2, 3, 4],  
7               [4, 3, 2, 1]])
```

有些情况下，我们需要随机生成`Tensor`中每个元素的值。下面我们创建一个形状为`(3, 4)`的`Tensor`。它的每个元素都随机采样于均值为0、标准差为1的正态分布。

```
1 In [10]:  
2     torch.randn((3, 4))  
3 Out [10]:  
4     tensor([[ 0.4323,  0.6941, -0.4098,  0.0299],  
5               [ 0.5798,  0.7531, -0.1006,  1.0074],  
6               [-1.0163,  0.1266,  0.1525, -2.1601]])
```

## 1.1.2 运算

Tensor 支持大量的运算符 (operator)。例如，我们可以对之前创建的两个形状为 (3, 4) 的 Tensor 做按元素加法。所得结果形状不变。

```
1 In [11]:  
2     X + Y  
3 Out [11]:  
4     tensor([[ 2,  2,  6,  6],  
5                 [ 5,  7,  9, 11],  
6                 [12, 12, 12, 12]])
```

按元素乘法如下：

```
1 In [12]:  
2     X * Y  
3 Out [12]:  
4     tensor([[ 0,  1,  8,  9],  
5                 [ 4, 10, 18, 28],  
6                 [32, 27, 20, 11]])
```

按元素除法如下：

```
1 In [13]:  
2     X.float() / Y.float()  
3 Out [13]:  
4     tensor([[ 0.0000,  1.0000,  0.5000,  1.0000],  
5                 [ 4.0000,  2.5000,  2.0000,  1.7500],  
6                 [ 2.0000,  3.0000,  5.0000, 11.0000]])
```

注意，首先需要将张量中的元素数据类型转为浮点型，否则计算结果仅有整数部分，小数部分会被截断。

按元素做指数运算如下：

```
1 In [14]:  
2     torch.exp(Y.float())  
3 Out [14]:  
4     tensor([[ 7.3891,  2.7183, 54.5981, 20.0855],  
5                 [ 2.7183,  7.3891, 20.0855, 54.5981],  
6                 [54.5981, 20.0855,  7.3891,  2.7183]])
```

除了按元素计算外，我们还可以使用 @ 做矩阵乘法。下面将 X 与 Y 的转置做矩阵乘法。由于 X 是 3 行 4 列的矩阵，Y 的转置为 4 行 3 列的矩阵，因此两个矩阵相乘得到 3 行 3 列的矩阵。

```
1 In [15]:  
2     X @ Y.t()  
3 Out [15]:  
4     tensor([[ 18,  20,  10],  
5                 [ 58,  60,  50],  
6                 [ 98, 100,  90]])
```

我们也可以将多个Tensor连结(concatenate)。下面分别在行上(维度0, 即形状中的最左边元素)和列上(维度1, 即形状中左起第二个元素)连结两个矩阵。可以看到, 输出的第一个Tensor在维度0的长度(6)为两个矩阵在维度0的长度之和(3+3), 而输出的第二个Tensor在维度1的长度(8)为两个输入矩阵在维度1的长度之和(4+4)。

```
1 In [16]:  
2     # 0为纵向拼接, 1为横向拼接  
3     torch.cat((X, Y), 0), torch.cat((X, Y), 1)  
4 Out [16]:  
5     tensor([[ 0,  1,  2,  3],  
6                 [ 4,  5,  6,  7],  
7                 [ 8,  9, 10, 11],  
8                 [ 2,  1,  4,  3],  
9                 [ 1,  2,  3,  4],  
10                [ 4,  3,  2,  1]]),  
11    tensor([[ 0,  1,  2,  3,  2,  1,  4,  3],  
12                 [ 4,  5,  6,  7,  1,  2,  3,  4],  
13                 [ 8,  9, 10, 11,  4,  3,  2,  1]]))
```

使用条件判别式可以得到元素为0或1的新Tensor。以X==Y为例, 如果X和Y在相同位置的条件判别为真(值相等), 那么新的Tensor在相同位置的值为1; 反之为0。

```
1 In [17]:  
2     X == Y  
3 Out [17]:  
4     tensor([[0, 1, 0, 1],  
5                 [0, 0, 0, 0],  
6                 [0, 0, 0, 0]], dtype=torch.uint8)
```

对Tensor中的所有元素求和得到只有一个元素的Tensor。

```
1 In [18]:  
2     torch.sum(X)  
3 Out [18]:  
4     tensor(66)
```

我们可以通过item()函数将结果变换为Python中的标量。下面例子中X的L<sub>2</sub>范数结果同上例一样是单元素Tensor, 但最后结果变换成了Python中的标量。

```
1 In [19]:  
2     torch.norm(X.float()).item()  
3 Out [19]:  
4     22.494443893432617
```

- 范数

$$L_1 \text{范数: } \|X\|^1 = |x_1| + |x_2| + \cdots + |x_n|$$
$$L_2 \text{范数: } \|X\|^2 = \sqrt{|x_1|^2 + |x_2|^2 + \cdots + |x_n|^2}$$

### 1.1.3 广播机制

前面我们看到如何对两个形状相同的 `Tensor` 做按元素运算。当对两个形状不同的 `Tensor` 按元素运算时，可能会触发广播（broadcasting）机制；先适当复制元素使这两个 `Tensor` 形状相同后再按元素运算。

先定义两个 `Tensor`。

```
1 In [20]:  
2     A = torch.arange(0, 3).view(3, 1)  
3     B = torch.arange(0, 2).view(1, 2)  
4     A, B  
5 Out [20]:  
6     (tensor([[0],  
7                 [1],  
8                 [2]]),  
9      tensor([[0, 1]))
```

由于A和B分别是3行1列和1行2列的矩阵，如果要计算A+B，那么A中第一列的3个元素被广播（复制）到了第二列，而B中第一行的2个元素被广播（复制）到了第二行和第三行。如此，就可以对2个3行2列的矩阵按元素相加。

```
1 In [21]:  
2     A + B  
3 Out [21]:  
4     tensor([[0, 1],  
5                 [1, 2],  
6                 [2, 3]])
```

### 1.1.4 索引

在 `Tensor` 中，索引（index）代表了元素的位置。`Tensor` 的索引从0开始逐一递增。例如，一个3行2列的矩阵的行索引分别为0、1和2，列索引分别为0和1。

在下面的例子中，我们指定了 `Tensor` 的行索引截取范围 `[1:3]`。根据[左闭右开指定范围的惯例](#)，它截取了矩阵X中行索引为1和2的两行。

```
1 In [22]:  
2     X[1:3]  
3 Out [22]:  
4     tensor([[ 4,  5,  6,  7],  
5                 [ 8,  9, 10, 11]])
```

我们可以指定 `Tensor` 中需要访问的单个元素的位置，如矩阵中行和列的索引，并未该元素重新赋值。

```
1 In [23]:  
2     X[1, 2] = 9  
3     X  
4 Out [23]:  
5     tensor([[ 0,  1,  2,  3],  
6                 [ 4,  5,  9,  7],  
7                 [ 8,  9, 10, 11]])
```

当然，我们也可以截取一部分元素，并为它们重新赋值。在下面的例子中，我们为行索引为1的每一系列元素重新赋值。

```
1 In [24]:  
2     X[1:2, :] = 12  
3     X  
4 Out [24]:  
5     tensor([[ 0,  1,  2,  3],  
6             [12, 12, 12, 12],  
7             [ 8,  9, 10, 11]])
```

## 1.1.5 运算的内存开销

在前面的例子里我们对每个操作新开内存来存储运算结果。举个例子，即使像 $Y=X+Y$ 这样的运算，我们也会新开内存，然后将 $Y$ 指向新内存。为了演示这一点，我们可以使用Python自带的`id`函数；如果两个实例的ID一致，那么它们所对应的内存地址相同；反之则不同。

```
1 In [25]:  
2     before = id(Y)  
3     Y = Y + X  
4     id(Y) == before  
5 Out [25]:  
6     False
```

如果想指定结果到特定内存，我们可以使用前面介绍的索引来进行替换操作。在下面的例子中，我们先通过`zeros_like()`创建和 $Y$ 形状相同且元素为0的`Tensor`，记为 $Z$ 。接下来，我们把 $X+Y$ 的结果通过`[:]写进 $Z$ 对应的内存中。`

```
1 In [26]:  
2     Z = torch.zeros_like(Y)  
3     before = id(Z)  
4     Z[:] = X + Y  
5     id(Z) == before  
6 Out [26]:  
7     True
```

实际上，上例中我们还是为 $X+Y$ 开了临时内存来存储计算结果，再复制到 $Z$ 对应的内存。如果想避免这个临时内存开销，我们可以使用运算符全名函数中的`out`参数。

```
1 In [27]:  
2     torch.add(X, Y, out=Z)  
3     id(Z) == before  
4 Out [27]:  
5     True
```

如果 $X$ 的值在之后的程序中不会复用，我们也可以用`X[:] = X+Y`或者`X += Y`来减少运算的内存开销。

```
1 In [28]:  
2     before = id(x)  
3     x += y  
4     id(x) == before  
5 Out [28]:  
6     True
```

## 1.1.6 Tensor和NumPy相互变换

我们可以通过 `numpy()` 和 `from_numpy()` 函数令数据在 Tensor 和 NumPy 格式之间相互变换。下面将 NumPy 实例转换成 Tensor 实例。

```
1 In [29]:  
2     import numpy as np  
3     P = np.ones((2, 3))  
4     D = torch.from_numpy(P)  
5     D  
6 Out [29]:  
7     tensor([[1., 1., 1.],  
8             [1., 1., 1.]])
```

再将 Tensor 实例转换成 NumPy 实例。

```
1 In [30]:  
2     D.numpy()  
3 Out [30]:  
4     array([[1., 1., 1.],  
5             [1., 1., 1.]])
```

### 小结:

- Tensor 是 PyTorch 中存储和变换数据的主要工具。
- 可以轻松地对 Tensor 创建、运算、指定索引，并与 NumPy 之间相互变换。

## 1.2 自动求梯度

在深度学习中，我们经常需要对函数求梯度（gradient）。本节将介绍如何使用 PyTorch 提供的 `autograd` 模块来自动求梯度。如果对本节中的数学概念（如梯度）不是很熟悉，可以参阅附录 A。

```
1 In [1]:  
2     from torch import autograd
```

### 1.2.1 简单例子

我们先看一个简单例子：对函数  $y = 2x^T x$  求关于列向量  $x$  的梯度。我们先创建变量  $x$ ，并赋初值。

```
1 In [2]:  
2     x = torch.arange(4).float()  
3     x  
4 Out [2]:  
5     tensor([0., 1., 2., 3.])
```

为了求有关变量 $x$ 的梯度，我们需要将其属性 `requires_grad_` 设置为 `True`，它将开始追踪 (track) 在其上的所有操作（这样就可以利用链式法则进行梯度传播了）。

```
1 In [3]:  
2     x.requires_grad_(True)  
3 Out [3]:  
4     tensor([0., 1., 2., 3.], requires_grad=True)
```

接下来我们构建表达式：

```
1 In [4]:  
2     y = 2*torch.dot(x, x.t())
```

由于 $x$ 的形状为  $(1, 4)$ ， $y$ 是一个标量。接下来我们可以通过调用 `backward()` 函数自动求梯度。

```
1 In [5]:  
2     y.backward()
```

函数 $y = 2x^T x$ 关于 $x$ 的梯度应为 $4x$ 。现在我们来验证一下求出来的梯度是否正确。

```
1 In [6]:  
2     print(x.grad)  
3 Out [6]:  
4     tensor([ 0.,  4.,  8., 12.])
```

## 1.3 查阅文档

受篇幅所限，本书无法对所有用到的PyTorch函数和类一一详细介绍。读者可以查阅相关文档来做更深入的了解。

### 1.3.1 查找模块里所有函数和类

当我们想知道一个模块里面提供了哪些可以调用的函数和类的时候，可以使用 `dir` 函数。下面我们将打印 `torch.random` 模块中所有的成员或属性。

```
1 In [1]:  
2     print(dir(torch.random))  
3 Out [1]:  
4    ['__builtins__', '__cached__', '__doc__', '__file__',  
5      '__loader__', '__name__', '__package__', '__spec__',  
6      '_fork_rng_warned_already', 'contextlib', 'default_generator',  
7      'fork_rng', 'get_rng_state', 'initial_seed', 'manual_seed',  
8      'set_rng_state', 'warnings']
```

通常我们可以忽略掉由 `_` 开头和结尾的函数（Python的特别对象）或者由 `_` 开头的函数（一般为内置函数），通过其余成员的名字我们大致猜测出这个模块提供了各种随机数的生成方法。

### 1.3.2 查找特定函数和类的使用

想了解某个函数或者类的具体用法时，可以使用 `help` 函数。让我们以 `Tensor` 中的 `ones_like` 函数为例，查阅它的用法。

```
1 In [2]:  
2     help(torch.ones_like)  
3 Out [2]:  
4 Help on built-in function ones_like:  
5  
6 ones_like(...)  
7     ones_like(input, dtype=None, layout=None, device=None,  
8                 requires_grad=False) -> Tensor  
9  
10    Returns a tensor filled with the scalar value `1`, with the same size as  
11    :attr:`input`. ``torch.ones_like(input)`` is equivalent to  
12    ``torch.ones(input.size(), dtype=input.dtype, layout=input.layout,  
13                 device=input.device)``.  
14  
15    .. warning::  
16        As of 0.4, this function does not support an :attr:`out` keyword.  
17        As an alternative, the old ``torch.ones_like(input, out=output)``  
18        is equivalent to ``torch.ones(input.size(), out=output)``.  
19  
20    Args:  
21        input (Tensor): the size of :attr:`input` will determine size of  
22            the output tensor  
23        dtype (:class:`torch.dtype`, optional): the desired data type of  
24            returned Tensor.  
25            Default: if ``None``, defaults to the dtype of :attr:`input`.  
26        layout (:class:`torch.layout`, optional): the desired layout of  
27            returned tensor.  
28            Default: if ``None``, defaults to the layout of :attr:`input`.  
29        device (:class:`torch.device`, optional): the desired device of  
30            returned tensor.  
31            Default: if ``None``, defaults to the device of :attr:`input`.  
32        requires_grad (bool, optional): If autograd should record operations  
33            on the returned tensor. Default: ``False``.  
34  
35    Example::  
36        >>> input = torch.empty(2, 3)  
37        >>> torch.ones_like(input)  
38        tensor([[ 1.,  1.,  1.],  
39                  [ 1.,  1.,  1.]])
```

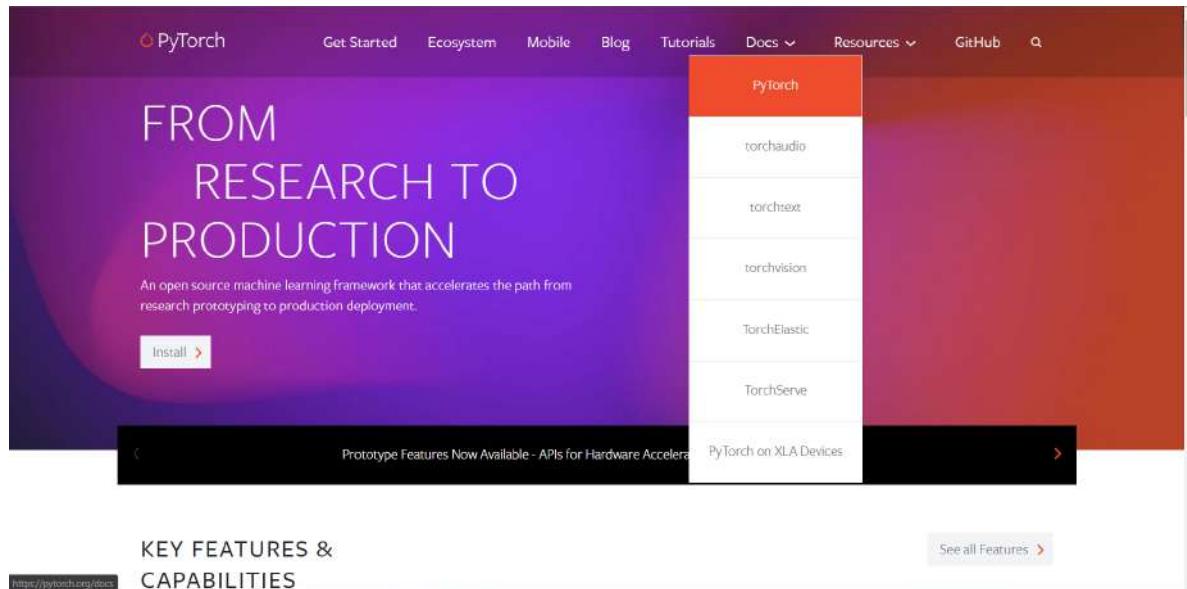
从文档信息我们了解到，`ones_like` 函数会创建和输入 `Tensor` 形状相同且元素为1的新 `Tensor`。我们可以验证一下。

```
1 In [3]:  
2     x = torch.tensor([[0, 0, 0], [2, 2, 2]])  
3     y = torch.ones_like(x)  
4     y  
5 Out [3]:  
6     tensor([[1, 1, 1],  
7             [1, 1, 1]])
```

在Jupyter Notebook里，我们可以使用`? 来将文档显示在另外一个窗口中。例如使用torch.ones_like? 将得到与help(torch.ones_like)一样的内容。`

### 1.3.3 在PyTorch网站上查阅

读者也可以在PyTorch的网站上查阅相关文档。访问PyTorch网站[PyTorch](#)，点击网页顶部下拉菜单“Docs”可查阅PyTorch的API文档。此外，也可以在网页右上方的搜索框中直接搜索函数或类名称。



此外，我们也可以访问[PyTorch中文文档](#)、[PyTorch官方教程中文版](#)这两个优质的中文资源。

#### 小结：

- 遇到不熟悉的PyTorch API时，可以主动查阅它的相关文档。
- 查阅PyTorch文档可以使用`dir`和`help`函数，或访问PyTorch官方网站。

## 1.4 本章附录

### ☆ `torch.arange()` 的用法

`torch.arange(start=0, end, step=1, out=None, layout=torch.strided, device=None, requires_grad=False) → Tensor`。该函数返回大小为  $\lceil \frac{end-start}{step} \rceil$  的一维张量，张量的取值介于  $[start, end)$ ，且取值之间的间隔为 `step`。需要注意的是，非整数 `step` 在与 `end` 比较时，会出现浮点舍入误差；为了避免不一致，我们建议在这种情况下在 `end` 上加一个很小的 `epsilon` 值。

$$out_{i+1} = out_i + step$$

参数	类型	说明	默认值
start	Number	取值的下限	0
end	Number	取值的上限	
step	Number	相邻两值之间的差值	1
out	Tensor (可选参数)	输出张量名称	
dtype	torch.dtype (可选参数)	返回张量数值类型, 若取值为None那选用全局默认值	None
device	torch.device (可选参数)	返回张量的存储位置, 若取值为None那选用全局默认位置, 也可以指定CPU、CUDA	None
requires_grad	bool (可选参数)	返回的张量是否能够进行梯度计算	False

### ☆ torch.numel() 的用法

torch.numel(input) → int。返回输入张量中元素的个数 (number of elements)。

参数	类型	说明	默认值
input	Tensor	输入张量	

### ☆ torch.randn() 的用法

torch.randn(\*sizes, out=None, dtype=None, layout=torch.strided, device=None, requires\_grad=False) → Tensor。返回一个形状为 size 的张量, 其值取于标准正态分布 (均值为0, 方差为1)。

$$\text{out}_i \sim \mathcal{N}(0, 1)$$

参数	类型	说明	默认值
size	(int...)	一个定义输出张量形状的整数序列。可以是一个数值, 也可以是一个集合, 如列表或元组。	

### ☆ torch.exp() 的用法

torch.exp(input, out=None) → Tensor。对输入张量的每一个元素取以 e 为底的指数。

$$y_i = e^{x_i}$$

参数	类型	说明	默认值
input	Tensor	待计算的张量	
out	Tensor (可选参数)	计算结果的存储位置	None

### ☆ torch.sum() 的用法

torch.sum(input, dtype=None) → Tensor。返回输入张量中所有元素的和。

### ☆ `torch.norm()` 的用法

`torch.norm(input, p, dim, out=None, keepdim=False) → Tensor`。返回输入张量给定维 `dim` 上每行的`p` 范数。

参数	类型	说明	默认值
<code>p</code>	<code>int, float</code>	范数计算中的幂指数值	2
<code>dim</code>	<code>int</code>	缩减的维度	<code>None</code>
<code>keepdim</code>	<code>bool</code>	是否保持输出的维度	<code>False</code>

`dim=0`是对0维度上的一个向量求范数，返回结果数量等于其列的个数，也就是说有多少个0维度的向量，将得到多少个范数。`dim=1`同理。当`keepdim=False`时，输出比输入少一个维度（就是指定的`dim`求范数的维度）。而`keepdim=True`时，输出与输入维度相同，仅仅是输出在求范数的维度上元素个数变为1。这也是为什么有时我们把参数中的`dim`称为缩减的维度，因为`norm`运算之后，此维度或者消失或者元素个数变为1。

### ☆ `item()` 的用法

`item() → number`。将张量转为标准Python数值类型，这个函数仅适用于仅有一个元素的张量，其它情况下应该使用`tolist()`。

### ☆ `torch.dot()` 的用法

`torch.dot(tensor1, tensor2) → Tensor`。计算两个张量的点积（内积）。这个函数不会使用广播机制。

### ☆ `backward()` 的用法

参见：[pytorch中backward\(\)函数详解](#)

本章代码详见GitHub链接[wzy6642](#)。

## 第2章 深度学习基础

从本章开始，我们将学习深度学习的奥秘。作为机器学习的一类，深度学习通常基于神经网络模型逐级表示越来越抽象的概念或模式。我们先从线性回归和softmax回归这两种单层神经网络入手，简要介绍机器学习中的基本概念。然后，我们由单层神经网络延伸到多层神经网络，并通过多层次感知机引入深度学习模型。在观察和了解了模型的过拟合现象后，我们将介绍深度学习中应对过拟合的常用方法——权重衰减和丢弃法。接着，为了进一步理解深度学习模型训练的本质，我们将详细解释正向传播和反向传播。掌握这两个概念后，我们能更好地认识深度学习中的数值稳定性和初始化的一些问题。最后，我们通过一个深度学习应用案例对本章内容学以致用。

在本章的前几节，我们先介绍单层神经网络——线性回归和softmax回归。

### 2.1 线性回归

线性回归输出是一个连续值，因此适用于回归问题。回归问题在实际中很常见，如预测房屋价格、气温、销售额等连续值的问题。与回归问题不同，分类问题中模型的最终输出是一个离散值。我们所说的图像分类、垃圾邮件识别、疾病检测等输出为离散值的问题都属于分类问题的范畴。**softmax回归则适用于分类问题。**

由于线性回归和softmax回归都是单层神经网络，它们涉及的概念和技术同样适用于大多数的深度学习模型。我们首先以线性回归为例，介绍大多数深度学习模型的基本要素和表示方法。

## 2.1.1 线性回归的基本要素

我们以一个简单的房屋价格预测作为例子来解释线性回归的基本要素。这个应用的目标是预测一栋房子的售出价格（元）。我们知道这个价格取决于很多因素，如房屋状况、地段、市场行情等。为了简单起见，这里我们假设价格只取决于房屋状况的两个因素，即面积（平方米）和房龄（年）。接下来我们希望探索价格与这两个因素的具体关系。

### 1. 模型

设房屋的面积为 $x_1$ ，房龄为 $x_2$ ，售出价格为 $y$ 。我们需要建立基于输入 $x_1$ 和 $x_2$ 来计算输出 $y$ 的表达式，也就是模型（model）。顾名思义，线性回归假设输出与各个输入之间是线性关系：

$$\hat{y} = x_1 w_1 + x_2 w_2 + b$$

其中 $w_1$ 和 $w_2$ 是权重（weight）， $b$ 是偏差（bias），且均为标量。它们是线性回归模型的参数（parameter）。模型输出 $\hat{y}$ 是线性回归对真实价格 $y$ 的预测或估计。我们通常允许它们之间有一定误差。

### 2. 模型训练

接下来我们需要通过数据来寻找特定的模型参数值，使模型在数据上的误差尽可能小。这个过程叫作模型训练（model training）。下面我们介绍模型训练所涉及的3个要素。

### 3. 训练数据

我们通常收集一系列的真实数据，例如多栋房屋的真实售出价格和它们对应的面积和房龄。我们希望在这个数据上面寻找模型参数来使模型的预测价格与真实价格的误差最小。在机器学习术语里，该数据集被称为训练数据集（training data set）或训练集（training set），一栋房屋被称为一个样本（sample），其真实售出价格叫作标签（label），用来预测标签的两个因素叫作特征（feature）。特征用来表征样本的特点。

假设我们采集的样本数为 $n$ ，索引为 $i$ 的样本的特征为 $x_1^{(i)}$ 和 $x_2^{(i)}$ ，标签为 $y^{(i)}$ 。对于索引为 $i$ 的房屋，线性回归模型的房屋价格预测表达式为：

$$\hat{y}^{(i)} = x_1^{(i)} w_1 + x_2^{(i)} w_2 + b$$

### 4. 损失函数

在模型训练中，我们需要衡量价格预测值与真实值之间的误差。通常我们会选取一个非负数作为误差，且数值越小表示误差越小。一个常用的选择是平方函数。它在评估索引为 $i$ 的样本误差的表达式为：

$$\ell^{(i)}(w_1, w_2, b) = \frac{1}{2} (\hat{y}^{(i)} - y^{(i)})^2$$

其中常数 $\frac{1}{2}$ 使对平方项求导后的常数系数为1，这样在形式上稍微简单一些。显然，误差越小表示预测价格与真实价格越相近，且当二者相等时误差为0。给定训练数据集，这个误差只与模型参数相关，因此我们将它记为以模型参数为参数的函数。在机器学习里，将衡量误差的函数称为损失函数（loss function）。这里使用的平方误差函数也称为平方损失（square loss）。

通常，我们用训练数据集中所有样本误差的平均来衡量模型预测的质量，即

$$\ell(w_1, w_2, b) = \frac{1}{n} \sum_{i=1}^n \ell^{(i)}(w_1, w_2, b) = \frac{1}{n} \sum_{i=1}^n \frac{1}{2} (x_1^{(i)} w_1 + x_2^{(i)} w_2 + b - y^{(i)})^2$$

在模型训练中，我们希望找出一组模型参数，记为 $w_1^*, w_2^*, b^*$ ，来使训练样本平均损失最小：

$$w_1^*, w_2^*, b^* = \arg \min_{w_1, w_2, b} \ell(w_1, w_2, b)$$

### 5. 优化算法

当模型和损失函数形式较为简单时，上面的误差最小化问题的解可以直接用公式表达出来。这类解叫作**解析解**（analytical solution）。本节使用的线性回归和平方误差刚好属于这个范畴。然而，大多数深度学习模型并没有解析解，只能通过优化算法有限次迭代模型参数来尽可能降低损失函数的值。这类解叫作**数值解**（numerical solution）。

在求数值解的优化算法中，**小批量随机梯度下降**（mini-batch stochastic gradient descent）在深度学习中被广泛使用。它的算法很简单：先选取一组模型参数的初始值，如随机选取；接下来对参数进行多次迭代，使每次迭代都可能降低损失函数的值。在每次迭代中，先随机均匀采样一个由固定数目训练数据样本所组成的小批量（mini-batch） $\mathcal{B}$ ，然后求小批量中数据样本的平均损失有关模型参数的导数（梯度），最后用此结果与预先设定的一个正数的乘积作为模型参数在本次迭代的减小量。

在训练本节讨论的线性回归模型的过程中，模型的每个参数将作如下迭代：

$$\begin{aligned} w_1 &\leftarrow w_1 - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \frac{\partial \ell^{(i)}(w_1, w_2, b)}{\partial w_1} = w_1 - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} x_1^{(i)} \left( x_1^{(i)} w_1 + x_2^{(i)} w_2 + b - y^{(i)} \right) \\ w_2 &\leftarrow w_2 - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \frac{\partial \ell^{(i)}(w_1, w_2, b)}{\partial w_2} = w_2 - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} x_2^{(i)} \left( x_1^{(i)} w_1 + x_2^{(i)} w_2 + b - y^{(i)} \right) \\ b &\leftarrow b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \frac{\partial \ell^{(i)}(w_1, w_2, b)}{\partial b} = b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \left( x_1^{(i)} w_1 + x_2^{(i)} w_2 + b - y^{(i)} \right) \end{aligned}$$

在上式中， $|\mathcal{B}|$  代表每个小批量中的样本个数（批量大小，batch size）， $\eta$  称作学习率（learning rate）并取正数。需要强调的是，这里的批量大小和学习率的值是人为设定的，并不是通过模型训练学出的，因此叫作超参数（hyperparameter）。我们通常所说的“调参”指的正是调节超参数，例如通过反复试错来找到超参数合适的值。在少数情况下，超参数也可以通过模型训练学出。本书对此类情况不做讨论。

#### 小贴士：梯度累积（accumulate gradients）

所谓梯度累积，其实很简单，我们梯度下降所用的梯度，实际上是多个样本算出来的梯度的平均值，以 batch\_size=128 为例，你可以一次性算出 128 个样本的梯度然后平均，我也可以每次算 16 个样本的平均梯度，然后缓存累加起来，算够了 8 次之后，然后把总梯度除以 8，然后才执行参数更新。当然，必须累积到了 8 次之后，用 8 次的平均梯度才去更新参数，不能每算 16 个就去更新一次，不然就是 batch\_size=16 了。

## 6. 模型预测

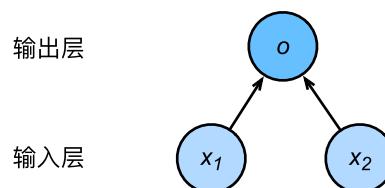
模型训练完成后，我们将模型参数  $w_1, w_2, b$  在优化算法停止时的值分别记作  $\hat{w}_1, \hat{w}_2, \hat{b}$ 。注意，这里我们得到的并不一定是最小化损失函数的最优解  $w_1^*, w_2^*, b^*$ ，而是对最优解的一个近似。然后，我们就可以使用学出的线性回归模型  $x_1 \hat{w}_1 + x_2 \hat{w}_2 + \hat{b}$  来估算训练数据集以外任意一栋面积（平方米）为  $x_1$ 、房龄（年）为  $x_2$  的房屋的价格了。这里的估算也叫作模型预测、模型推断或模型测试。

### 2.1.2 线性回归的表示方法

我们已经阐述了线性回归的模型表达式、训练和预测。下面我们解释线性回归与神经网络的联系，以及线性回归的矢量计算表达式。

#### 1. 神经网络图

在深度学习中，我们可以使用神经网络图直观地表现模型结构。为了更清晰地展示线性回归作为神经网络的结构，下图使用神经网络图表示本节中介绍的线性回归模型。神经网络图隐去了模型参数权重和偏差。



在上图所示的神经网络中，输入分别为  $x_1$  和  $x_2$ ，因此输入层的输入个数为2。输入个数也叫特征数或特征向量维度。图中网络的输出为  $o$ ，输出层的输出个数为1。需要注意的是，我们直接将图中神经网络的输出  $o$  作为线性回归的输出，即  $\hat{y} = o$ 。由于输入层并不涉及计算，按照惯例，图中所示的神经网络的层数为1。所以，线性回归是一个单层神经网络。输出层中负责计算  $o$  的单元又叫神经元。在线性回归中， $o$  的计算依赖于  $x_1$  和  $x_2$ 。也就是说，输出层中的神经元和输入层中各个输入完全连接。因此，这里的输出层又叫全连接层 (fully-connected layer) 或稠密层 (dense layer)。

## 2. 矢量计算表达式

在模型训练或预测时，我们常常会同时处理多个数据样本并用到矢量计算。在介绍线性回归的矢量计算表达式之前，让我们先考虑对两个向量相加的两种方法。

下面先定义两个1000维的向量。

```
1 In [1]:
2     import torch
3     a = torch.ones(1000)
4     b = torch.ones(1000)
```

向量相加的一种方法是，将这两个向量按元素逐一做标量加法。

```
1 In [2]:
2     from time import time
3     start = time()
4     c = torch.zeros(1000)
5     for i in range(1000):
6         c[i] = a[i] + b[i]
7     time() - start
8 Out [2]:
9     0.030505657196044922
```

向量相加的另一种方法是，将这两个向量直接做矢量加法。

```
1 In [3]:
2     start = time()
3     d = a + b
4     time() - start
5 Out [3]:
6     0.00029587745666503906
```

结果很明显，后者比前者更省时。因此，我们应该尽可能采用矢量计算，以提升计算效率。

让我们再次回到本节的房价预测问题。如果我们对训练数据集里的3个房屋样本（索引分别为1、2和3）逐一预测价格，将得到

$$\begin{aligned}\hat{y}^{(1)} &= x_1^{(1)}w_1 + x_2^{(1)}w_2 + b \\ \hat{y}^{(2)} &= x_1^{(2)}w_1 + x_2^{(2)}w_2 + b \\ \hat{y}^{(3)} &= x_1^{(3)}w_1 + x_2^{(3)}w_2 + b\end{aligned}$$

现在，我们将上面3个等式转化成矢量计算。设

$$\hat{\mathbf{y}} = \begin{bmatrix} \hat{y}^{(1)} \\ \hat{y}^{(2)} \\ \hat{y}^{(3)} \end{bmatrix}, \quad \mathbf{X} = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} \\ x_1^{(2)} & x_2^{(2)} \\ x_1^{(3)} & x_2^{(3)} \end{bmatrix}, \quad \mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$$

对3个房屋样本预测价格的矢量计算表达式为  $\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b$ , 其中的加法运算使用了广播机制。例如:

```
1 In [4]:  
2     a = torch.ones(3)  
3     b = 10  
4     a + b  
5 Out [4]:  
6     tensor([11., 11., 11.])
```

广义上讲, 当数据样本数为  $n$ , 特征数为  $d$  时, 线性回归的矢量计算表达式为

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b$$

其中模型输出  $\hat{\mathbf{y}} \in \mathbb{R}^{n \times 1}$  批量数据样本特征  $\mathbf{X} \in \mathbb{R}^{n \times d}$ , 权重  $\mathbf{w} \in \mathbb{R}^{d \times 1}$ , 偏差  $b \in \mathbb{R}$ 。相应地, 批量数据样本标签  $\mathbf{y} \in \mathbb{R}^{n \times 1}$ 。设模型参数  $\boldsymbol{\theta} = [w_1, w_2, b]^\top$ , 我们可以重写损失函数为

$$\ell(\boldsymbol{\theta}) = \frac{1}{2n} (\hat{\mathbf{y}} - \mathbf{y})^\top (\hat{\mathbf{y}} - \mathbf{y})$$

小批量随机梯度下降的迭代步骤将相应地改写为

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla_{\boldsymbol{\theta}} \ell^{(i)}(\boldsymbol{\theta})$$

其中梯度是损失有关3个为标量的模型参数的偏导数组成的向量:

$$\nabla_{\boldsymbol{\theta}} \ell^{(i)}(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\partial \ell^{(i)}(w_1, w_2, b)}{\partial w_1} \\ \frac{\partial \ell^{(i)}(w_1, w_2, b)}{\partial w_2} \\ \frac{\partial \ell^{(i)}(w_1, w_2, b)}{\partial b} \end{bmatrix} = \begin{bmatrix} x_1^{(i)}(x_1^{(i)}w_1 + x_2^{(i)}w_2 + b - y^{(i)}) \\ x_2^{(i)}(x_1^{(i)}w_1 + x_2^{(i)}w_2 + b - y^{(i)}) \\ x_1^{(i)}w_1 + x_2^{(i)}w_2 + b - y^{(i)} \end{bmatrix} = \begin{bmatrix} x_1^{(i)} \\ x_2^{(i)} \\ 1 \end{bmatrix} (\hat{y}^{(i)} - y^{(i)})$$

### 小结:

- 和大多数深度学习模型一样, 对于线性回归这样一种单层神经网络, 它的基本要素包括模型、训练数据、损失函数和优化算法。
- 既可以用神经网络图表示线性回归, 又可以用矢量计算表示该模型。
- 应该尽可能采用矢量计算, 以提升计算效率。

使用NumPy计算两向量相加:

```
1 In [5]:  
2     import numpy as np  
3     a = np.ones(1000)  
4     b = np.ones(1000)  
5     start = time()  
6     d = a + b  
7     time() - start  
8 Out [5]:  
9     0.00017452239990234375
```

## 2.2 线性回归从零开始实现

在了解了线性回归的背景知识之后, 现在我们可以动手实现它了。尽管强大的深度学习框架可以减少大量重复性工作, 但若过于依赖它提供的便利, 会导致我们很难深入理解深度学习是如何工作的。因此, 本节将介绍如何只利用 `Tensor` 和 `autograd` 来实现一个线性回归的训练。

首先，导入本节中实验所需的包或模块，其中的matplotlib包可用于作图。

```
1 In [1]:  
2     import matplotlib.pyplot as plt  
3     import random
```

## 2.2.1 生成数据集

我们构造一个简单的人工训练数据集，它可以使我们能够直观比较学到的参数和真实的模型参数的区别。设训练数据集样本数为1000，输入个数（特征数）为2。给定随机生成的批量样本特征  $\mathbf{X} \in \mathbb{R}^{1000 \times 2}$ ，我们使用线性回归模型真实权重  $\mathbf{w} = [2, -3.4]^\top$  和偏差  $b = 4.2$ ，以及一个随机噪声项  $\epsilon$  来生成标签

$$\mathbf{y} = \mathbf{X}\mathbf{w} + b + \epsilon$$

其中噪声项  $\epsilon$  服从均值为0、标准差为0.01的[正态分布](#)。噪声代表了数据集中无意义的干扰。下面，让我们生成数据集。

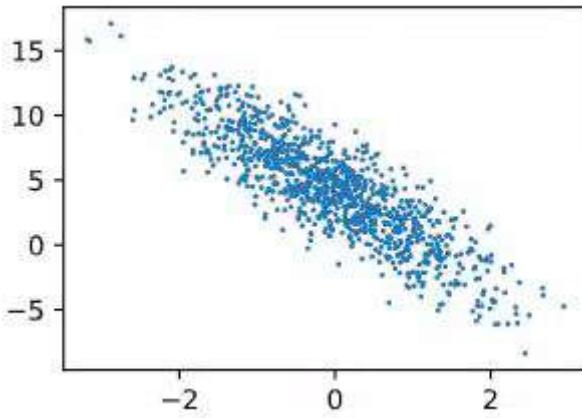
```
1 In [2]:  
2     num_inputs = 2  
3     num_examples = 1000  
4     true_w = [2, -3.4]  
5     true_b = 4.2  
6     features = torch.randn(num_examples, num_inputs, dtype=torch.float32)  
7     labels = true_w[0] * features[:, 0] + true_w[1] * features[:, 1] + true_b  
8     labels += torch.tensor(np.random.normal(0, 0.01, size=labels.size()),  
9                           dtype=torch.float32)
```

注意，`features` 的每一行是一个长度为2的向量，而 `labels` 的每一行是一个长度为1的向量（标量）。

```
1 In [3]:  
2     features[0], labels[0]  
3 Out [3]:  
4     (tensor([1.9959, 1.6476]), tensor(2.5870))
```

通过生成第二个特征 `features[:, 1]` 和标签 `labels` 的散点图，可以更直观地观察两者间的线性关系。

```
1 In [4]:  
2     from IPython import display  
3     def use_svg_display():  
4         # 用矢量图显示  
5         display.set_matplotlib_formats('svg')  
6  
7     def set_figsize(figsize=(3.5, 2.5)):  
8         use_svg_display()  
9         # 设置图的尺寸  
10        plt.rcParams['figure.figsize'] = figsize  
11  
12    set_figsize()  
13    # 加分号只显示图  
14    plt.scatter(features[:, 1].numpy(), labels.numpy(), 1);
```



我们将上面的 `plt` 作图函数以及 `use_svg_display` 函数和 `set_figsize` 函数定义在 `d2lzh` 包里。以后在作图时，我们在作图前只需要调用 `d2lzh.set_figsize()` 即可打印矢量图并设置图的尺寸。

## 2.2.2 读取数据集

在训练模型的时候，我们需要遍历数据集并不断读取小批量数据样本。这里我们定义一个函数：它每次返回 `batch_size`（批量大小）个随机样本的特征和标签。`yield` 用法参见[博客](#)。

```

1 In [5]:
2     def data_iter(batch_size, features, labels):
3         num_examples = len(features)
4         indices = list(range(num_examples))
5         # 样本的读取顺序是随机的
6         random.shuffle(indices)
7         for i in range(0, num_examples, batch_size):
8             # 最后一次可能不足一个batch
9             j = torch.LongTensor(indices[i: min(i+batch_size,
10                                         num_examples)])
11             # index_select函数根据索引返回对应元素
12             yield features.index_select(0, j), labels.index_select(0, j)

```

让我们读取第一个小批量数据样本并打印。每个批量的特征形状为 $(10, 2)$ ，分别对应批量大小和输入个数；标签形状为批量大小。

```

1 In [6]:
2     batch_size = 10
3     for x, y in data_iter(batch_size, features, labels):
4         print(x, y)
5         break
6 Out [6]:
7     tensor([[-0.3435,  1.8129],
8             [-1.1850, -2.3661],
9             [-0.2579,  0.5165],
10            [ 0.0646, -0.2483],
11            [-2.0696, -1.6340],
12            [ 1.5402,  1.5464],
13            [ 0.1918,  0.8717],
14            [ 0.7798, -0.2045],
15            [-2.1754, -0.3097],
16            [-0.7769,  2.0223]])
17     tensor([-2.6513,  9.8691,  1.9491,  5.1673,  5.6162,
18             2.0323,  1.6198,  6.4336,  0.9028, -4.2275])

```

## 2.2.3 初始化模型参数

我们将权重初始化成均值为0、标准差为0.01的正态随机数，偏差则初始化成0。

```
1 In [7]:  
2     w = torch.tensor(np.random.normal(0, 0.01, (num_inputs, 1)),  
3                         dtype=torch.float32)  
4     b = torch.zeros(1, dtype=torch.float32)
```

之后的模型训练中，需要对这些参数求梯度来迭代参数的值，因此我们要让它们的 `requires_grad=True`。

```
1 In [8]:  
2     w.requires_grad_(requires_grad=True)  
3     b.requires_grad_(requires_grad=True)  
4 Out [8]:  
5     tensor([0.], requires_grad=True)
```

## 2.2.4 定义模型

下面是线性回归的矢量计算表达式的实现。我们使用 `mm()` 函数做矩阵乘法。

```
1 In [9]:  
2     def linreg(x, w, b):  
3         return torch.mm(x, w) + b
```

## 2.2.5 定义损失函数

我们使用上一节描述的平方损失来定义线性回归的损失函数。在实现中，我们需要把真实值 `y` 变形成预测值 `y_hat` 的形状。以下函数返回的结果也将和 `y_hat` 的形状相同。

```
1 In [10]:  
2     def squared_loss(y_hat, y):  
3         return (y_hat - y.view(y_hat.size())) ** 2 / 2
```

## 2.2.6 定义优化算法

以下的 `sgd` 函数实现了上一节中介绍的小批量随机梯度下降算法。它通过不断迭代模型参数来优化损失函数。这里自动求梯度模块计算得来的梯度是一个批量样本的梯度和。我们将它除以批量大小来得到平均值。

```
1 In [11]:  
2     def sgd(params, lr, batch_size):  
3         for param in params:  
4             param.data -= lr * param.grad / batch_size
```

在训练中，我们将多次迭代模型参数。在每次迭代中，我们根据当前读取的小批量数据样本（特征 `x` 和标签 `y`），通过调用反向函数 `backward` 计算小批量随机梯度，并调用优化算法 `sgd` 迭代模型参数。由于我们之前设批量大小 `batch_size` 为10，每个小批量的损失 `l` 的形状为(10, 1)。回忆一下自动求梯度一节。由于变量 `l` 并不是一个标量，所以我们可以调用 `.sum()` 将其求和得到一个标量，再运行 `l.backward()` 得到该变量有关模型参数的梯度。注意在每次更新完参数后不要忘了将参数的梯度清零。（如果不清零，PyTorch默认会对梯度进行累加）

在一个迭代周期 (epoch) 中，我们将完整遍历一遍 `data_iter` 函数，并对训练数据集中所有样本都使用一次（假设样本数能够被批量大小整除）。这里的迭代周期个数 `num_epochs` 和学习率 `lr` 都是超参数，分别设3和0.03。在实践中，大多超参数都需要通过反复试错来不断调节。虽然迭代周期数设得越大模型可能越有效，但是训练时间可能过长。而有关学习率对模型的影响，我们会在后面“优化算法”一章中详细介绍。

```
1 In [12]:  
2     lr = 0.03  
3     num_epochs = 3  
4     net = linreg  
5     loss = squared_loss  
6     # 训练模型一共需要num_epochs个迭代周期  
7     for epoch in range(num_epochs):  
8         # 在每一个迭代周期中，会使用训练集中所有样本一次（假设样本数能够被批量大小整除）。  
9         # x和y分别是小批量样本的特征和标签  
10        for x, y in data_iter(batch_size, features, labels):  
11            # l是有关小批量x和y的损失  
12            l = loss(net(x, w, b), y).sum()  
13            # 小批量损失对模型参数求梯度  
14            l.backward()  
15            # 使用小批量随机梯度下降迭代模型参数  
16            sgd([w, b], lr, batch_size)  
17            # 梯度清零  
18            w.grad.data.zero_()  
19            b.grad.data.zero_()  
20            train_l = loss(net(features, w, b), labels)  
21            print('epoch %d, loss %f' % (epoch+1, train_l.mean().item()))  
22 Out [12]:  
23 epoch 1, loss 0.026483  
24 epoch 2, loss 0.000097  
25 epoch 3, loss 0.000050
```

训练完成后，我们可以比较学到的参数和用来生成训练集的真实参数。它们应该很接近。

```
1 In [13]:  
2     true_w, w  
3 Out [13]:  
4     ([2, -3.4],  
5      tensor([[ 1.9994],  
6              [-3.4004]], requires_grad=True))
```

```
1 In [14]:  
2     true_b, b  
3 Out [14]:  
4     (4.2, tensor([4.1995], requires_grad=True))
```

## 小结：

- 可以看出，仅使用 `Tensor` 和 `autograd` 模块就可以很容易地实现一个模型。接下来，本书会在此基础上描述更多深度学习模型，并介绍怎样使用更简洁的代码（见下一节）来实现它们。

## 2.3 线性回归的简洁实现

随着深度学习框架的发展，开发深度学习应用变得越来越便利。实践中，我们通常可以用比上一节更简洁的代码来实现同样的模型。在本节中，我们将介绍如何使用PyTorch更方便地实现线性回归的训练。

## 2.3.1 生成数据集

我们生成与上一节中相同的数据集。其中 `features` 是训练数据特征，`labels` 是标签。

```
1 In [1]:  
2     num_inputs = 2  
3     num_examples = 1000  
4     true_w = [2, -3.4]  
5     true_b = 4.2  
6     features = torch.tensor(  
7         np.random.normal(0, 1, (num_examples, num_inputs)),  
8         dtype=torch.float32)  
9     labels = true_w[0] * features[:, 0] + true_w[1] * features[:, 1] +  
10        true_b  
11    labels += torch.tensor(np.random.normal(0, 0.01, size=labels.size()),  
12                           dtype=torch.float32)
```

## 2.3.2 读取数据集

PyTorch提供了 `data` 包来读取数据。由于 `data` 常用作变量名，我们将导入的 `data` 模块用 `Data` 代替。在每一次迭代中，我们将随机读取包含10个数据样本的小批量。

```
1 In [2]:  
2     import torch.utils.data as Data  
3     batch_size = 10  
4     # 将训练数据的特征和标签组合  
5     dataset = Data.TensorDataset(features, labels)  
6     # 随机读取小批量  
7     data_iter = Data.DataLoader(dataset, batch_size, shuffle=True)
```

这里 `data_iter` 的使用跟上一节中的一样。让我们读取并打印第一个小批量数据样本。

```
1 In [3]:  
2     for x, y in data_iter:  
3         print(x, y)  
4         break  
5 Out [3]:  
6     tensor([[-0.3566, -0.6247],  
7             [-0.2553,  0.6302],  
8             [-0.1231, -1.1588],  
9             [ 1.3664,  1.4850],  
10            [ 0.2306,  0.9412],  
11            [-0.1985,  1.0253],  
12            [-0.6620, -1.4253],  
13            [ 1.1332, -0.4332],  
14            [ 1.0766,  1.1685],  
15            [-0.0485, -0.1464]]),  
16     tensor([5.6135, 1.5645, 7.8991, 1.8619, 1.4556,  
17             0.3204, 7.7312, 7.9381, 2.3910, 4.6097])
```

### 2.3.3 定义模型

在上一节从零开始的实现中，我们需要定义模型参数，并使用它们一步步描述模型是怎样计算的。当模型结构变得更复杂时，这些步骤将变得更繁琐。其实，PyTorch提供了大量预定义的层，这使我们只需关注使用哪些层来构造模型。下面将介绍如何使用PyTorch更简洁地定义线性回归。

首先，导入`torch.nn`模块。实际上，“nn”是neural networks（神经网络）的缩写。顾名思义，该模块定义了大量神经网络的层。之前我们已经用过了`autograd`，而`nn`就是利用`autograd`来定义模型。`nn`的核心数据结构是`Module`，它是一个抽象概念，既可以表示神经网络中的某个层（layer），也可以表示一个包含很多层的神经网络。在实际使用中，最常见的做法是继承`nn.Module`，撰写自己的网络/层。一个`nn.Module`实例应该包含一些层以及返回输出的前向传播（forward）方法。下面先来看看如何用`nn.Module`实现一个线性回归模型。

```
1 In [4]:  
2     from torch import nn  
3     class LinearNet(nn.Module):  
4         def __init__(self, n_feature):  
5             super(LinearNet, self).__init__()  
6             self.linear = nn.Linear(n_feature, 1)  
7         # forward定义前向传播  
8         def forward(self, x):  
9             y = self.linear(x)  
10            return y  
11     net = LinearNet(num_inputs)  
12     # 打印网络结构  
13     print(net)  
14 Out [4]:  
15     LinearNet(  
16         (linear): Linear(in_features=2, out_features=1, bias=True)  
17     )
```

事实上我们还可以用`nn.Sequential`来更加方便地搭建网络，`Sequential`是一个有序的容器，网络层将按照在传入`Sequential`的顺序依次被添加到计算图中。当给定输入数据时，容器中的每一层将依次计算并将输出作为下一层的输入。

```
1 In [5]:  
2     net = nn.Sequential()  
3     net.add_module('linear', nn.Linear(num_inputs, 1))  
4     print(net)  
5 Out [5]:  
6     Sequential(  
7         (linear): Linear(in_features=2, out_features=1, bias=True)  
8     )
```

作为一个单层神经网络，线性回归输入层中的神经元和输入层中各个输入完全连接。因此，线性回归的输出层又叫全连接层。

我们可以通过`net.parameters()`来查看模型所有的可学习参数，此函数将返回一个生成器。

```
1 In [6]:  
2     for param in net.parameters():  
3         print(param)  
4 Out [6]:  
5     Parameter containing:  
6     tensor([[ 0.1518, -0.0460]], requires_grad=True)  
7     Parameter containing:  
8     tensor([-0.2482], requires_grad=True)
```

#### 小贴士：单样本转换为min\_batch

`torch.nn` 仅支持输入一个batch的样本不支持单个样本输入，如果只有单个样本，可使用 `input.unsqueeze(0)` 来添加一维。

### 2.3.4 初始化模型参数

在使用 `net` 前，我们需要初始化模型参数，如线性回归模型中的权重和偏差。PyTorch在 `init` 模块中提供了多种参数初始化方法。这里的 `init` 是 `initializer` 的缩写形式。我们通过 `init.normal_` 将权重参数每个元素初始化为随机采样于均值为0、标准差为0.01的正态分布。偏差参数默认会初始化为零。

```
1 In [7]:  
2     from torch.nn import init  
3     init.normal_(net[0].weight, mean=0, std=0.01)  
4     init.constant_(net[0].bias, val=0)
```

如果这里的 `net` 是用2.3.3节一开始的代码自定义的，那么上面代码会报错，`net[0].weight` 应改为 `net.linear.weight`，`bias` 亦然。因为 `net[0]` 这样根据下标访问子模块的写法只有当 `net` 是个 `ModuleList` 或者 `Sequential` 实例时才可以。

### 2.3.5 定义损失函数

PyTorch在 `nn` 模块中提供了各种损失函数，这些损失函数可看作是一种特殊的层，PyTorch也将这些损失函数实现为 `nn.Module` 的子类。我们现在使用它提供的均方误差损失作为模型的损失函数。

```
1 In [8]:  
2     loss = nn.MSELoss()
```

### 2.3.6 定义优化算法

同样，我们也无须自己实现小批量随机梯度下降算法。`torch.optim` 模块提供了很多常用的优化算法比如SGD、Adam和RMSProp等。下面我们创建一个用于优化 `net` 所有参数的优化器实例，并指定学习率为0.03的小批量随机梯度下降（SGD）为优化算法。

```
1 In [9]:  
2     import torch.optim as optim  
3     optimizer = optim.SGD(net.parameters(), lr=0.03)  
4     print(optimizer)  
5 Out [9]:  
6     SGD (br/>7         Parameter Group 0  
8             dampening: 0  
9             lr: 0.03  
10            momentum: 0  
11            nesterov: False  
12            weight_decay: 0  
13     )
```

## 2.3.7 训练模型

在使用PyTorch训练模型时，我们通过调用 `optim` 实例的 `step` 函数来迭代模型参数。

```
1 In [10]:  
2     num_epochs = 3  
3     for epoch in range(1, num_epochs+1):  
4         for x, y in data_iter:  
5             # 前向传播  
6             output = net(x)  
7             # 计算损失  
8             l = loss(output, y.view(-1, 1))  
9             # 梯度清零  
10            optimizer.zero_grad()  
11            # 反向传播  
12            l.backward()  
13            # 更新参数  
14            optimizer.step()  
15            print('epoch %d, loss: %f' % (epoch, l.item()))  
16 Out [10]:  
17     epoch 1, loss: 0.000167  
18     epoch 2, loss: 0.000150  
19     epoch 3, loss: 0.000136
```

下面我们分别比较学到的模型参数和真实的模型参数。我们从 `net` 获得需要的层，并访问其权重 (`weight`) 和偏差 (`bias`)。学到的参数和真实的参数很接近。

```
1 In [11]:  
2     dense = net[0]  
3     print(true_w, dense.weight)  
4     print(true_b, dense.bias)  
5 Out [11]:  
6     [2, -3.4] Parameter containing:  
7     tensor([[ 2.0008, -3.3996]], requires_grad=True)  
8     4.2 Parameter containing:  
9     tensor([4.1997], requires_grad=True)
```

## 小结:

- 使用PyTorch可以更简洁地实现模型。

- `torch.utils.data` 模块提供了有关数据处理的工具，`torch.nn` 模块定义了大量神经网络的层，`torch.nn.init` 模块定义了各种初始化方法，`torch.optim` 模块提供了很多常用的优化算法。

## 2.4 softmax回归

前几节介绍的线性回归模型适用于输出为连续值的情景。在另一类情景中，模型输出可以是一个像图像类别这样的离散值。对于这样的离散值预测问题，我们可以使用诸如softmax回归在内的分类模型。和线性回归不同，softmax回归的输出单元从一个变成了多个，且引入了softmax运算使输出更适合离散值的预测和训练。本节以softmax回归模型为例，介绍神经网络中的分类模型。

### 2.4.1 分类问题

让我们考虑一个简单的图像分类问题，其输入图像的高和宽均为2像素，且色彩为灰度。这样每个像素值都可以用一个标量表示。我们将图像中的4像素分别记为 $x_1, x_2, x_3, x_4$ 。假设训练数据集中图像的真实标签为狗、猫或鸡（假设可以用4像素表示出这3种动物），这些标签分别对应离散值 $y_1, y_2, y_3$ 。

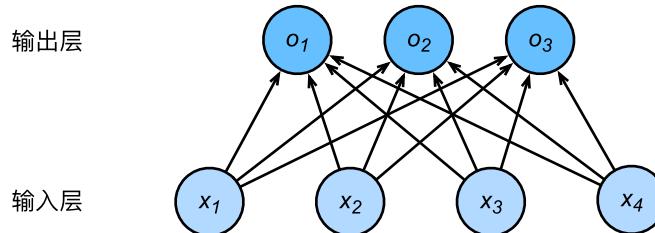
我们通常使用离散的数值来表示类别，例如 $y_1 = 1, y_2 = 2, y_3 = 3$ 。如此，一张图像的标签为1、2和3这3个数值中的一个。虽然我们仍然可以使用回归模型来进行建模，并将预测值就近定点化到1、2和3这3个离散值之一，但这种连续值到离散值的转化通常会影响到分类质量。因此我们一般使用更加适合离散值输出的模型来解决分类问题。

### 2.4.2 softmax回归模型

softmax回归跟线性回归一样将输入特征与权重做线性叠加。与线性回归的一个主要不同在于，softmax回归的输出值个数等于标签里的类别数。因为一共有4种特征和3种输出动物类别，所以权重包含12个标量（带下标的 $w$ ）、偏差包含3个标量（带下标的 $b$ ），且对每个输入计算 $o_1, o_2, o_3$ 这3个输出：

$$\begin{aligned} o_1 &= x_1 w_{11} + x_2 w_{21} + x_3 w_{31} + x_4 w_{41} + b_1 \\ o_2 &= x_1 w_{12} + x_2 w_{22} + x_3 w_{32} + x_4 w_{42} + b_2 \\ o_3 &= x_1 w_{13} + x_2 w_{23} + x_3 w_{33} + x_4 w_{43} + b_3 \end{aligned}$$

下图用神经网络图描绘了上面的计算。softmax回归同线性回归一样，也是一个单层神经网络。由于每个输出 $o_1, o_2, o_3$ 的计算都要依赖于所有的输入 $x_1, x_2, x_3, x_4$ ，softmax回归的输出层也是一个全连接层。



#### softmax运算

既然分类问题需要得到离散的预测输出，一个简单的办法是将输出值 $o_i$ 当作预测类别是 $i$ 的置信度，并将值最大的输出所对应的类作为预测输出，即输出  $\arg \max_i o_i$ 。例如，如果 $o_1, o_2, o_3$ 分别为0.1, 10, 0.1，由于 $o_2$ 最大，那么预测类别为2，其代表猫。

然而，直接使用输出层的输出有两个问题。一方面，由于输出层的输出值的范围不确定，我们难以直观上判断这些值的意义。例如，刚才举的例子中的输出值10表示“很置信”图像类别为猫，因为该输出值是其他两类的输出值的100倍。但如果 $o_1 = o_3 = 10^3$ ，那么输出值10却又表示图像类别为猫的概率很低。另一方面，由于真实标签是离散值，这些离散值与不确定范围的输出值之间的误差难以衡量。

softmax运算符解决了以上两个问题。它通过下式将输出值转换成值为正且和为1的概率分布：

$$\hat{y}_1, \hat{y}_2, \hat{y}_3 = \text{softmax}(o_1, o_2, o_3)$$

其中

$$\hat{y}_1 = \frac{\exp(o_1)}{\sum_{i=1}^3 \exp(o_i)}, \quad \hat{y}_2 = \frac{\exp(o_2)}{\sum_{i=1}^3 \exp(o_i)}, \quad \hat{y}_3 = \frac{\exp(o_3)}{\sum_{i=1}^3 \exp(o_i)}$$

容易看出  $\hat{y}_1 + \hat{y}_2 + \hat{y}_3 = 1$  且  $0 \leq \hat{y}_1, \hat{y}_2, \hat{y}_3 \leq 1$ , 因此  $\hat{y}_1, \hat{y}_2, \hat{y}_3$  是一个合法的概率分布。这时候, 如果  $\hat{y}_2 = 0.8$ , 不管  $\hat{y}_1$  和  $\hat{y}_3$  的值是多少, 我们都知道图像类别为猫的概率是 80%。此外, 我们注意到

$$\arg \max_i o_i = \arg \max_i \hat{y}_i$$

因此 softmax 运算不改变预测类别输出。

### 2.4.3 单样本分类的矢量计算表达式

为了提高计算效率, 我们可以将单样本分类通过矢量计算来表达。在上面的图像分类问题中, 假设 softmax 回归的权重和偏差参数分别为

$$\mathbf{W} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{41} & w_{42} & w_{43} \end{bmatrix}, \quad \mathbf{b} = [b_1 \quad b_2 \quad b_3]$$

设高和宽分别为 2 个像素的图像样本  $i$  的特征为

$$\mathbf{x}^{(i)} = [x_1^{(i)} \quad x_2^{(i)} \quad x_3^{(i)} \quad x_4^{(i)}]$$

输出层的输出为

$$\mathbf{o}^{(i)} = [o_1^{(i)} \quad o_2^{(i)} \quad o_3^{(i)}]$$

预测为狗、猫或鸡的概率分布为

$$\hat{\mathbf{y}}^{(i)} = [\hat{y}_1^{(i)} \quad \hat{y}_2^{(i)} \quad \hat{y}_3^{(i)}]$$

softmax 回归对样本  $i$  分类的矢量计算表达式为

$$\begin{aligned} \mathbf{o}^{(i)} &= \mathbf{x}^{(i)} \mathbf{W} + \mathbf{b} \\ \hat{\mathbf{y}}^{(i)} &= \text{softmax}(\mathbf{o}^{(i)}) \end{aligned}$$

### 2.4.4 小批量样本分类的矢量计算表达式

为了进一步提升计算效率, 我们通常对小批量数据做矢量计算。广义上讲, 给定一个小批量样本, 其批量大小为  $n$ , 输入个数 (特征数) 为  $d$ , 输出个数 (类别数) 为  $q$ 。设批量特征为  $\mathbf{X} \in \mathbb{R}^{n \times d}$ 。假设 softmax 回归的权重和偏差参数分别为  $\mathbf{W} \in \mathbb{R}^{d \times q}$  和  $\mathbf{b} \in \mathbb{R}^{1 \times q}$ 。softmax 回归的矢量计算表达式为

$$\begin{aligned} \mathbf{O} &= \mathbf{X} \mathbf{W} + \mathbf{b} \\ \hat{\mathbf{Y}} &= \text{softmax}(\mathbf{O}) \end{aligned}$$

其中的加法运算使用了广播机制,  $\mathbf{O}, \hat{\mathbf{Y}} \in \mathbb{R}^{n \times q}$  且这两个矩阵的第  $i$  行分别为样本  $i$  的输出  $\mathbf{o}^{(i)}$  和概率分布  $\hat{\mathbf{y}}^{(i)}$ 。例如批大小为 2, 那么  $\mathbf{X}$  的维度为  $2 \times 4$ , 权重矩阵的维度为  $4 \times 3$ , 那么矩阵相乘得到的结果的维度为  $2 \times 3$ 。此时偏置维度为  $1 \times 3$ , 需要将其复制一行广播为  $2 \times 3$ , 然后与前面计算结果相加得到维度为  $2 \times 3$  的矩阵。其中 2 表示 2 个样本, 3 表示类别个数。

## 2.4.5 交叉熵损失函数

前面提到，使用softmax运算后可以更方便地与离散标签计算误差。我们已经知道，softmax运算将输出转换成一个合法的类别预测分布。实际上，真实标签也可以用类别分布表达：对于样本*i*，我们构造向量 $\mathbf{y}^{(i)} \in \mathbb{R}^q$ ，使其第 $y_j^{(i)}$ （样本*i*类别的离散数值）个元素为1，其余为0。这样我们的训练目标可以设为使预测概率分布 $\hat{\mathbf{y}}^{(i)}$ 尽可能接近真实的标签概率分布 $\mathbf{y}^{(i)}$ 。

我们可以像线性回归那样使用平方损失函数 $\|\hat{\mathbf{y}}^{(i)} - \mathbf{y}^{(i)}\|^2 / 2$ 。然而，想要预测分类结果正确，我们其实并不需要预测概率完全等于标签概率。例如，在图像分类的例子中，如果 $y^{(i)} = 3$ ，那么我们只需要 $\hat{y}_3^{(i)}$ 比其他两个预测值 $\hat{y}_1^{(i)}$ 和 $\hat{y}_2^{(i)}$ 大就行了。即使 $\hat{y}_3^{(i)}$ 值为0.6，不管其他两个预测值为多少，类别预测均正确。而平方损失则过于严格，例如 $\hat{y}_1^{(i)} = \hat{y}_2^{(i)} = 0.2$ 比 $\hat{y}_1^{(i)} = 0, \hat{y}_2^{(i)} = 0.4$ 的损失要小很多，虽然两者都有同样正确的分类预测结果。

改善上述问题的一个方法是使用更适合衡量两个概率分布差异的测量函数。其中，**交叉熵** (cross entropy) 是一个常用的衡量方法：

$$H(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}) = - \sum_{j=1}^q y_j^{(i)} \log \hat{y}_j^{(i)}$$

其中带下标的 $y_j^{(i)}$ 是向量 $\mathbf{y}^{(i)}$ 中非0即1的元素，需要注意将它与样本*i*类别的离散数值，即不带下标的 $y^{(i)}$ 区分。在上式中，我们知道向量 $\mathbf{y}^{(i)}$ 中只有第 $y^{(i)}$ 个元素 $y_{y^{(i)}}^{(i)}$ 为1，其余全为0，于是 $H(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}) = -\log \hat{y}_{y^{(i)}}^{(i)}$ 。也就是说，交叉熵只关心对正确类别的预测概率，因为只要其值足够大，就可以确保分类结果正确。当然，遇到一个样本有多个标签时，例如图像里含有不止一个物体时，我们并不能做这一步简化。但即便对于这种情况，交叉熵同样只关心对图像中出现的物体类别的预测概率。

假设训练数据集的样本数为*n*，交叉熵损失函数定义为

$$\ell(\Theta) = \frac{1}{n} \sum_{i=1}^n H(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)})$$

其中 $\Theta$ 代表模型参数。同样地，如果每个样本只有一个标签，那么交叉熵损失可以简写成 $\ell(\Theta) = -(1/n) \sum_{i=1}^n \log \hat{y}_{y^{(i)}}^{(i)}$ 。从另一个角度来看，我们知道最小化 $\ell(\Theta)$ 等价于最大化 $\exp(-n\ell(\Theta)) = \prod_{i=1}^n \hat{y}_{y^{(i)}}^{(i)}$ ，即**最小化交叉熵损失函数等价于最大化训练数据集所有标签类别的联合预测概率**。

## 2.4.6 模型预测及评价

在训练好softmax回归模型后，给定任一样本特征，就可以预测每个输出类别的概率。通常，我们把预测概率最大的类别作为输出类别。如果它与真实类别（标签）一致，说明这次预测是正确的。在2.6节的实验中，我们将使用准确率（accuracy）来评价模型的表现。**它等于正确预测数量与总预测数量之比**。

**小结：**

- softmax回归适用于分类问题。它使用softmax运算输出类别的概率分布。
- softmax回归是一个单层神经网络，输出个数等于分类问题中的类别个数。
- 交叉熵适合衡量两个概率分布的差异。

## 2.5 图像分类数据集 (Fashion-MNIST)

在介绍softmax回归的实现前我们先引入一个多类图像分类数据集。它将在后面的章节中被多次使用，以方便我们观察比较算法之间在模型精度和计算效率上的区别。图像分类数据集中最常用的是手写数字识别数据集MNIST。但大部分模型在MNIST上的分类精度都超过了95%。为了更直观地观察算法之间的差异，我们将使用一个图像内容更加复杂的Fashion-MNIST数据集。

## 2.5.1 获取数据集

首先导入本节需要的包或模块。

```
1 In [1]:  
2     import torchvision  
3     import torchvision.transforms as transforms
```

下面，我们通过torchvision的 `torchvision.datasets` 来下载这个数据集。第一次调用时会自动从网上获取数据。我们通过参数 `train` 来指定获取训练数据集或测试数据集（testing data set）。测试数据集也叫测试集（testing set），只用来评价模型的表现，并不用来训练模型。

另外我们还指定了参数 `transform = transforms.ToTensor()` 使所有数据转换为 `Tensor`，如果不进行转换则返回的是PIL图片。`transforms.ToTensor()` 将尺寸为  $(H \times W \times C)$  且数据位于  $[0, 255]$  的 PIL图片或者数据类型为 `np.uint8` 的NumPy数组转换为尺寸为  $(C \times H \times W)$  且数据类型为 `torch.float32` 且位于  $[0.0, 1.0]$  的 `Tensor`。

注意：由于像素值为0到255的整数，所以刚好是uint8所能表示的范围，包括 `transforms.ToTensor()` 在内的一些关于图片的函数就默认输入的是uint8型，若不是，可能不会报错但可能得不到想要的结果。所以，**如果用像素值（0-255整数）表示图片数据，那么一律将其类型设置成 uint8，避免不必要的bug。**

```
1 In [2]:  
2     mnist_train = torchvision.datasets.FashionMNIST(  
3         root='data/FashionMNIST', train=True, download=True,  
4         transform=transforms.ToTensor())  
5     mnist_test = torchvision.datasets.FashionMNIST(  
6         root='data/FashionMNIST', train=False, download=True,  
7         transform=transforms.ToTensor())
```

上面的 `mnist_train` 和 `mnist_test` 都是 `torch.utils.data.Dataset` 的子类，所以我们可以用 `len()` 来获取该数据集的大小，还可以用下标来获取具体的一个样本。训练集中和测试集中的每个类别的图像数分别为6,000和1,000。因为有10个类别，所以训练集和测试集的样本数分别为60,000和10,000。

```
1 In [3]:  
2     len(mnist_train), len(mnist_test)  
3 Out [3]:  
4     (60000, 10000)
```

我们可以通过方括号 `[]` 来访问任意一个样本，下面获取第一个样本的图像和标签。

```
1 In [4]:  
2     feature, label = mnist_train[0]
```

变量 `feature` 对应高和宽均为28像素的图像。由于我们使用了 `transforms.ToTensor()`，所以每个像素的数值为  $[0.0, 1.0]$  的32位浮点数。需要注意的是，`feature` 的尺寸是  $(C \times H \times W)$  的，而不是  $(H \times W \times C)$ 。第一维是通道数，因为数据集中是灰度图像，所以通道数为1。后面两维分别是图像的高和宽。为了表述简洁，我们将高和宽分别为  $h$  和  $w$  像素的图像的形状记为  $h \times w$  或  $(h, w)$ 。

```
1 In [5]:  
2     feature.shape, feature.dtype  
3 Out [5]:  
4     (torch.Size([1, 28, 28]), torch.float32)
```

图像的标签使用 int 型数据表示。

```
1 In [6]:  
2     label, type(label)  
3 Out [6]:  
4     (9, int)
```

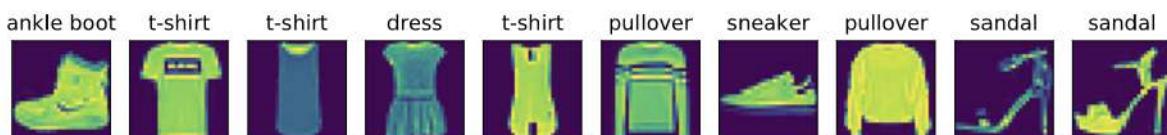
Fashion-MNIST 中一共包括了 10 个类别，分别为 t-shirt (T 恤)、trouser (裤子)、pullover (套衫)、dress (连衣裙)、coat (外套)、sandal (凉鞋)、shirt (衬衫)、sneaker (运动鞋)、bag (包) 和 ankle boot (短靴)。以下函数可以将数值标签转成相应的文本标签。

下面定义一个可以在一行里画出多张图像和对应标签的函数。

```
1 In [7]:  
2     import d2lzh as d2l  
3     def show_fashion_mnist(images, labels):  
4         d2l.use_svg_display()  
5         # 这里的_表示我们忽略(不使用)的变量  
6         _, figs = d2l.plt.subplots(1, len(images), figsize=(12, 12))  
7         for f, img, lbl in zip(figs, images, labels):  
8             f.imshow(img.view((28, 28)).numpy())  
9             f.set_title(lbl)  
10            f.axes.get_xaxis().set_visible(False)  
11            f.axes.get_yaxis().set_visible(False)
```

现在，我们看一下训练数据集中前 10 个样本的图像内容和文本标签。

```
1 In [8]:  
2     def get_fashion_mnist_labels(labels):  
3         text_labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',  
4                         'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']  
5         return [text_labels[int(i)] for i in labels]  
6     x, y = [], []  
7     for i in range(10):  
8         x.append(mnist_train[i][0])  
9         y.append(mnist_train[i][1])  
10    show_fashion_mnist(x, get_fashion_mnist_labels(y))
```



## 2.5.2 读取小批量

我们将在训练数据集上训练模型，并将训练好的模型在测试数据集上评价模型的表现。虽然我们可以像 2.2 节中那样通过 `yield` 来定义读取小批量数据样本的函数，但是为了代码简洁，这里我们直接创建 `DataLoader` 实例。该实例每次读取一个样本数为 `batch_size` 的小批量数据，这里的批量大小 `batch_size` 是一个超参数。

在实践中，数据读取经常是训练的性能瓶颈，特别当模型较简单或者计算硬件性能较高时。PyTorch的 `DataLoader` 中一个很方便的功能是允许使用多进程来加速数据读取。这里我们通过参数 `num_workers` 来设置4个进程读取数据。

```
1 In [9]:  
2     import sys  
3     batch_size = 256  
4     if sys.platform.startswith('win'):  
5         # 0表示不用额外的进程来加速读取数据  
6         num_workers = 0  
7     else:  
8         num_workers = 4  
9     train_iter = torch.utils.data.DataLoader(  
10        mnist_train, batch_size=batch_size, shuffle=True,  
11        num_workers=num_workers)  
12     test_iter = torch.utils.data.DataLoader(  
13        mnist_test, batch_size=batch_size, shuffle=False,  
14        num_workers=num_workers)
```

我们将获取并读取Fashion-MNIST数据集的逻辑封装在 `d2lzh.load_data_fashion_mnist` 函数中供后面章节调用。该函数将返回 `train_iter` 和 `test_iter` 两个变量。随着本书内容的不断深入，我们会进一步改进该函数。它的完整实现将在4.6节中描述。

最后我们查看读取一遍训练数据需要的时间。

```
1 In [10]:  
2     start = time()  
3     for x, y in train_iter:  
4         continue  
5     print('%.2f sec' % (time() - start))  
6 Out [10]:  
7     2.48 sec
```

## 小结:

- Fashion-MNIST是一个10类服饰分类数据集，之后章节里将使用它来检验不同算法的表现。
- 我们将高和宽分别为 $h$ 和 $w$ 像素的图像的形状记为 $h \times w$ 或 $(h, w)$ 。

## 2.6 softmax回归的从零开始实现

这一节我们来动手实现softmax回归。首先导入本节实现所需的包或模块。

```
1 In [1]:  
2     import d2lzh as d2l
```

### 2.6.1 读取数据集

我们将使用Fashion-MNIST数据集，并设置批量大小为256。

```
1 In [2]:  
2     batch_size = 256  
3     train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

## 2.6.2 初始化模型参数

跟线性回归中的例子一样，我们将使用向量表示每个样本。已知每个样本输入是高和宽均为28像素的图像。模型的输入向量的长度是  $28 \times 28 = 784$ ：该向量的每个元素对应图像中每个像素。由于图像有10个类别，单层神经网络输出层的输出个数为10，因此softmax回归的权重和偏差参数分别为  $784 \times 10$  和  $1 \times 10$  的矩阵。

```
1 In [3]:  
2     num_inputs = 784  
3     num_outputs = 10  
4     w = torch.tensor(np.random.normal(0, 0.01, (num_inputs, num_outputs)),  
5                         dtype=torch.float32)  
6     b = torch.zeros(num_outputs, dtype=torch.float32)
```

同之前一样，我们需要模型参数梯度。

```
1 In [4]:  
2     w.requires_grad_(requires_grad=True)  
3     b.requires_grad_(requires_grad=True)
```

## 2.6.3 实现softmax运算

在介绍如何定义softmax回归之前，我们先描述一下对如何对多维 `Tensor` 按维度操作。在下面的例子中，给定一个 `Tensor` 矩阵 `x`。我们可以只对其中同一列 (`dim=0`) 或同一行 (`dim=1`) 的元素求和，并在结果中保留行和列这两个维度 (`keepdim=True`)。

```
1 In [5]:  
2     x = torch.tensor([[1, 2, 3], [4, 5, 6]])  
3     print(x.sum(dim=0, keepdim=True))  
4     print(x.sum(dim=1, keepdim=True))  
5 Out [5]:  
6     tensor([5, 7, 9])  
7     tensor([[ 6],  
8             [15]])
```

下面我们就可以定义前面小节里介绍的softmax运算。在下面的函数中，矩阵 `x` 的行数是样本数，列数是输出个数。为了表达样本预测各个输出的概率，softmax运算会先通过 `exp` 函数对每个元素做指数运算，再对 `exp` 矩阵同行元素求和，最后令矩阵每行各元素与该行元素之和相除。这样一来，最终得到的矩阵每行元素和为1且非负。因此，该矩阵每行都是合法的概率分布。softmax运算的输出矩阵中的任意一行元素代表了一个样本在各个输出类别上的预测概率。

```
1 In [6]:  
2     def softmax(x):  
3         x_exp = torch.exp(x)  
4         partition = x_exp.sum(dim=1, keepdim=True)  
5         # 这里应用了广播机制  
6         return x_exp / partition
```

可以看到，对于随机输入，我们将每个元素变成了非负数，且每一行和为1。

```
1 In [7]:  
2     x = torch.tensor(np.random.normal(0, 1, (2, 5)), dtype=torch.float32)  
3     x_prob = softmax(x)  
4     x_prob, x_prob.sum(dim=1)  
5 Out [7]:  
6     (tensor([[0.1302, 0.2112, 0.1268, 0.2356, 0.2963],  
7             [0.1290, 0.1621, 0.0497, 0.1127, 0.5465]]),  
8     tensor([1., 1.])))
```

## 2.6.4 定义模型

有了softmax运算，我们可以定义上节描述的softmax回归模型了。这里通过 view 函数将每张原始图像改成长度为 num\_inputs 的向量。

```
1 In [8]:  
2     def net(x):  
3         return softmax(torch.mm(x.view((-1, num_inputs)), w) + b)
```

## 2.6.5 定义损失函数

上一节中，我们介绍了softmax回归使用的交叉熵损失函数。为了得到标签的预测概率，我们可以使用 gather 函数。在下面的例子中，变量 y\_hat 是2个样本在3个类别的预测概率，变量 y 是这2个样本的标签类别。通过使用 gather 函数，我们得到了2个样本的标签的预测概率。与2.4节数学表述中标签类别离散值从1开始逐一递增不同，在代码中，标签类别的离散值是从0开始逐一递增的。

```
1 In [9]:  
2     y_hat = torch.tensor([[0.1, 0.3, 0.6], [0.3, 0.2, 0.5]])  
3     y = torch.LongTensor([0, 2])  
4     torch.gather(input=y_hat, dim=1, index=y.view(-1, 1))  
5 Out [9]:  
6     tensor([[0.1000],  
7             [0.5000]])
```

下面实现了2.4节（softmax回归）中介绍的交叉熵损失函数。

```
1 In [10]:  
2     def cross_entropy(y_hat, y):  
3         return -torch.log(y_hat.gather(1, y.view(-1, 1)))
```

## 2.6.6 计算分类准确率

给定一个类别的预测概率分布 y\_hat，我们把预测概率最大的类别作为输出类别。如果它与真实类别 y 一致，说明这次预测是正确的。分类准确率即正确预测数量与总预测数量之比。

为了演示准确率的计算，下面定义准确率 accuracy 函数。其中 y\_hat.argmax(dim=1) 返回矩阵 y\_hat 每行中最大元素的索引，且返回结果与变量 y 形状相同。相等条件判断式 (y\_hat.argmax(dim=1) == y) 是一个类型为 ByteTensor 的 Tensor，我们用 float() 将其转换为值为0（相等为假）或1（相等为真）的浮点型 Tensor。

```
1 In [11]:  
2     def accuracy(y_hat, y):  
3         return (y_hat.argmax(dim=1) == y).float().mean().item()
```

让我们继续使用在演示 `gather` 函数时定义的变量 `y_hat` 和 `y`，并将它们分别作为预测概率分布和标签。可以看到，第一个样本预测类别为2（该行最大元素0.6在本行的索引为2），与真实标签0不一致；第二个样本预测类别为2（该行最大元素0.5在本行的索引为2），与真实标签2一致。因此，这两个样本上的分类准确率为0.5。

```
1 In [12]:  
2     accuracy(y_hat, y)  
3 Out [12]:  
4     0.5
```

类似地，我们可以评价模型 `net` 在数据集 `data_iter` 上的准确率。

```
1 In [13]:  
2     def evaluate_accuracy(data_iter, net):  
3         acc_sum, n = 0.0, 0  
4         # 特征和标签  
5         for X, y in data_iter:  
6             acc_sum += (net(X).argmax(dim=1) == y).float().sum().item()  
7             # 样本个数  
8             n += y.shape[0]  
9         return acc_sum / n
```

因为我们随机初始化了模型 `net`，所以这个随机模型的准确率应该接近于类别个数10的倒数即0.1。

```
1 In [14]:  
2     print(evaluate_accuracy(test_iter, net))  
3 Out [14]:  
4     0.0481
```

## 2.6.7 训练模型

训练softmax回归的实现跟2.2节介绍的线性回归中的实现非常相似。我们同样使用小批量随机梯度下降来优化模型的损失函数。在训练模型时，迭代周期数 `num_epochs` 和学习率 `lr` 都是可以调的超参数。改变它们的值可能会得到分类更准确的模型。

```
1 In [15]:  
2     num_epochs, lr = 5, 0.1  
3     def train_ch3(net, train_iter, test_iter, loss, num_epochs, batch_size,  
4                     params=None, lr=None, optimizer=None):  
5         for epoch in range(num_epochs):  
6             train_l_sum, train_acc_sum, n = 0.0, 0.0, 0  
7             for X, y in train_iter:  
8                 y_hat = net(X)  
9                 l = loss(y_hat, y).sum()  
10  
11                 # 梯度清零  
12                 if optimizer is not None:  
13                     optimizer.zero_grad()  
14                 elif params is not None and params[0].grad is not None:  
15                     for param in params:  
16                         param.grad.data.zero_()  
17  
18                 l.backward()
```

```

19         if optimizer is None:
20             d2l.sgd(params, lr, batch_size)
21         else:
22             optimizer.step()
23
24         train_l_sum += l.item()
25         train_acc_sum += (y_hat.argmax(dim=1) == y).sum().item()
26         n += y.shape[0]
27     test_acc = evaluate_accuracy(test_iter, net)
28     print('epoch %d, loss %.4f, train acc %.3f, test acc %.3f' %
29           (epoch+1, train_l_sum/n, train_acc_sum/n, test_acc))
30     train_ch3(net, train_iter, test_iter, cross_entropy, num_epochs,
31               batch_size, [w, b], lr)
32
33 Out [15]:
34 epoch 1, loss 0.7881, train acc 0.748, test acc 0.791
35 epoch 2, loss 0.5708, train acc 0.813, test acc 0.808
36 epoch 3, loss 0.5259, train acc 0.825, test acc 0.820
37 epoch 4, loss 0.5011, train acc 0.832, test acc 0.823
38 epoch 5, loss 0.4859, train acc 0.836, test acc 0.826

```

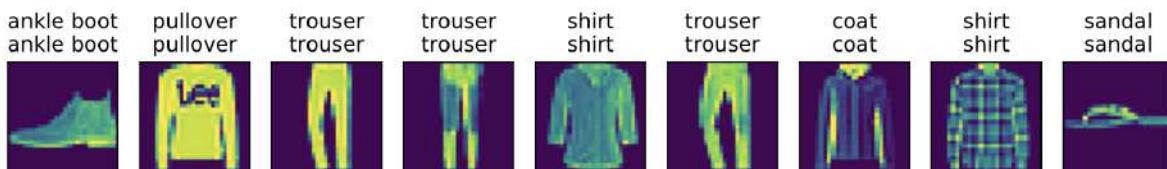
## 2.6.8 预测

训练完成后，现在就可以演示如何对图像进行分类了。给定一系列图像（第三行图像输出），我们比较一下它们的真实标签（第一行文本输出）和模型预测结果（第二行文本输出）。

```

1 In [16]:
2     X, y = iter(test_iter).next()
3     true_labels = d2l.get_fashion_mnist_labels(y.numpy())
4     pred_labels = d2l.get_fashion_mnist_labels(net(X).argmax(dim=1).numpy())
5     titles = [true + '\n' + pred for true, pred in
6               zip(true_labels, pred_labels)]
7     d2l.show_fashion_mnist(X[:9], titles[:9])

```



**小结：**

- 可以使用softmax回归做多类别分类。与训练线性回归相比，你会发现训练softmax回归的步骤和它非常相似：获取并读取数据、定义模型和损失函数并使用优化算法训练模型。事实上，绝大多数深度学习模型的训练都有着类似的步骤。

## 2.7 softmax回归的简洁实现

我们在2.3节（线性回归的简洁实现）中已经了解了使用Pytorch实现模型的便利。下面，让我们再次使用Pytorch来实现一个softmax回归模型。首先导入所需的包或模块。

```

1 In [1]:
2     import torch
3     from torch import nn
4     from torch.nn import init
5     import d2lzh as d2l

```

## 2.7.1 读取数据集

我们仍然使用Fashion-MNIST数据集和上一节中设置的批量大小。

```
1 In [2]:  
2     batch_size = 256  
3     train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

## 2.7.2 定义和初始化模型

在2.4节（softmax回归）中提到，softmax回归的输出层是一个全连接层，所以我们用一个线性模块就可以了。因为前面我们数据返回的每个batch样本  $x$  的形状为 $(batch\_size, 1, 28, 28)$ ，所以我们要先用 `view()` 将  $x$  的形状转换成 $(batch\_size, 784)$ 才送入全连接层。

```
1 In [3]:  
2     num_inputs = 784  
3     num_outputs = 10  
4     class LinearNet(nn.Module):  
5         def __init__(self, num_inputs, num_outputs):  
6             super(LinearNet, self).__init__()  
7             self.linear = nn.Linear(num_inputs, num_outputs)  
8             # x shape: (batch, 1, 28, 28)  
9             def forward(self, x):  
10                 y = self.linear(x.view(x.shape[0], -1))  
11                 return y  
12     net = LinearNet(num_inputs, num_outputs)
```

我们将对  $x$  的形状转换的这个功能自定义一个 `FlattenLayer` 并记录在 `d2lzh` 中方便后面使用。

```
1 In [4]:  
2     class FlattenLayer(nn.Module):  
3         def __init__(self):  
4             super(FlattenLayer, self).__init__()  
5             # x shape: (batch, *, *, ...)  
6             def forward(self, x):  
7                 return x.view(x.shape[0], -1)
```

这样我们就可以更方便地定义我们的模型：

```
1 In [5]:  
2     from collections import OrderedDict  
3     net = nn.Sequential(  
4         OrderedDict([  
5             ('flatten', FlattenLayer()),  
6             ('linear', nn.Linear(num_inputs, num_outputs))  
7         ])  
8     )
```

然后，我们使用均值为0、标准差为0.01的正态分布随机初始化模型的权重参数。

```
1 In [6]:  
2     init.normal_(net.linear.weight, mean=0, std=0.01)  
3     init.constant_(net.linear.bias, val=0)
```

## 2.7.3 softmax和交叉熵损失函数

如果做了上一节的练习，那么你可能意识到了分开定义softmax运算和交叉熵损失函数可能会造成数值不稳定。因此，PyTorch提供了一个包括softmax运算和交叉熵损失计算的函数。它的数值稳定性更好。

```
1 In [7]:  
2     loss = nn.CrossEntropyLoss()
```

## 2.7.4 定义优化算法

我们使用学习率为0.1的小批量随机梯度下降作为优化算法。

```
1 In [8]:  
2     optimizer = torch.optim.SGD(net.parameters(), lr=0.1)
```

## 2.7.5 训练模型

接下来，我们使用上一节中定义的训练函数来训练模型。

```
1 In [9]:  
2     num_epochs = 5  
3     d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs,  
4                     batch_size, None, None, optimizer)  
5 Out [9]:  
6     epoch 1, loss 0.0031, train acc 0.748, test acc 0.787  
7     epoch 2, loss 0.0022, train acc 0.813, test acc 0.803  
8     epoch 3, loss 0.0021, train acc 0.826, test acc 0.810  
9     epoch 4, loss 0.0020, train acc 0.832, test acc 0.821  
10    epoch 5, loss 0.0019, train acc 0.837, test acc 0.819
```

### 小结:

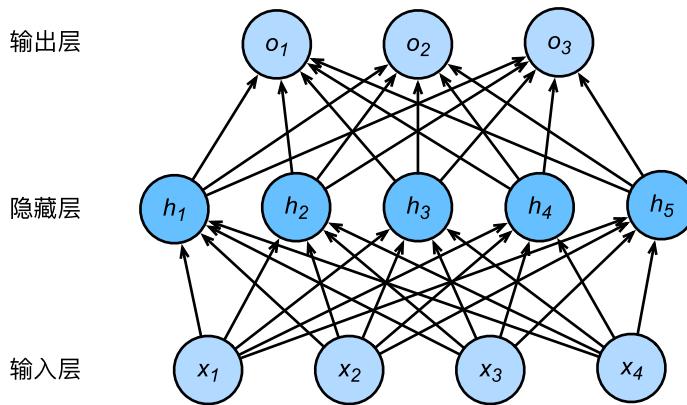
- PyTorch提供的函数往往具有更好的数值稳定性。
- 可以使用PyTorch更简洁地实现softmax回归。

## 2.8 多层感知机

我们已经介绍了包括线性回归和softmax回归在内的单层神经网络。然而深度学习主要关注多层模型。在本节中，我们将以多层感知机（multilayer perceptron, MLP）为例，介绍多层神经网络的概念。

### 2.8.1 隐藏层

多层感知机在单层神经网络的基础上引入了一到多个隐藏层（hidden layer）。隐藏层位于输入层和输出层之间。下图展示了一个多层感知机的神经网络图，它含有一个隐藏层，该层中有5个隐藏单元。



在上图所示的多层感知机中，输入和输出个数分别为4和3，中间的隐藏层中包含了5个隐藏单元（hidden unit）。由于输入层不涉及计算，上图中的多层感知机的层数为2。由上图可见，隐藏层中的神经元和输入层中各个输入完全连接，输出层中的神经元和隐藏层中的各个神经元也完全连接。因此，多层感知机中的隐藏层和输出层都是全连接层。

具体来说，给定一个小批量样本  $\mathbf{X} \in \mathbb{R}^{n \times d}$ ，其批量大小为  $n$ ，输入（特征）个数为  $d$ 。假设多层感知机只有一个隐藏层，其中隐藏单元个数为  $h$ 。记隐藏层的输出（也称为隐藏层变量或隐藏变量）为  $\mathbf{H}$ ，有  $\mathbf{H} \in \mathbb{R}^{n \times h}$ 。因为隐藏层和输出层均是全连接层，可以设隐藏层的权重参数和偏差参数分别为  $\mathbf{W}_h \in \mathbb{R}^{d \times h}$  和  $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$ ，输出层的权重和偏差参数分别为  $\mathbf{W}_o \in \mathbb{R}^{h \times q}$  和  $\mathbf{b}_o \in \mathbb{R}^{1 \times q}$ 。

我们先来看一种含单隐藏层的多层感知机的设计。其输出  $\mathbf{O} \in \mathbb{R}^{n \times q}$  的计算为

$$\begin{aligned}\mathbf{H} &= \mathbf{X}\mathbf{W}_h + \mathbf{b}_h \\ \mathbf{O} &= \mathbf{H}\mathbf{W}_o + \mathbf{b}_o\end{aligned}$$

也就是将隐藏层的输出直接作为输出层的输入。如果将以上两个式子联立起来，可以得到

$$\mathbf{O} = (\mathbf{X}\mathbf{W}_h + \mathbf{b}_h)\mathbf{W}_o + \mathbf{b}_o = \mathbf{X}\mathbf{W}_h\mathbf{W}_o + \mathbf{b}_h\mathbf{W}_o + \mathbf{b}_o$$

从联立后的式子可以看出，虽然神经网络引入了隐藏层，却依然等价于一个单层神经网络：其中输出层权重参数为  $\mathbf{W}_h\mathbf{W}_o$ ，偏差参数为  $\mathbf{b}_h\mathbf{W}_o + \mathbf{b}_o$ 。**不难发现，即便再添加更多的隐藏层，以上设计依然只能与仅含输出层的单层神经网络等价。**

## 2.8.2 激活函数

上述问题的根源在于全连接层只是对数据做**仿射变换**（affine transformation），而多个仿射变换的叠加仍然是一个仿射变换。解决问题的一个方法是引入非线性变换，例如对隐藏变量使用按元素运算的非线性函数进行变换，然后再作为下一个全连接层的输入。这个非线性函数被称为**激活函数**（activation function）。下面我们介绍几个常用的激活函数。

### 1. ReLU函数

ReLU (rectified linear unit) 函数提供了一个很简单的非线性变换。给定元素  $x$ ，该函数定义为

$$\text{ReLU}(x) = \max(x, 0)$$

可以看出，ReLU函数只保留正数元素，并将负数元素清零。为了直观地观察这一非线性变换，我们先定义一个绘图函数 `xyplot`。`detach()` 的使用见[博客](#)。

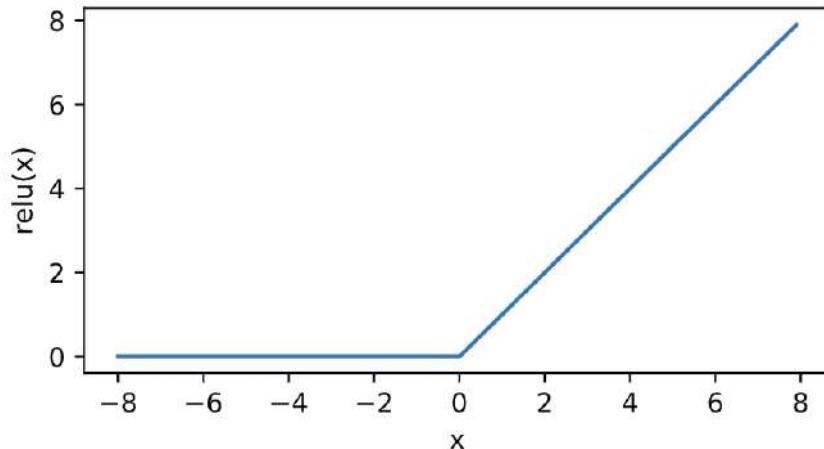
```

1 In [1]:
2     def xyplot(x_vals, y_vals, name):
3         d2l.set_figsize(figsize=(5, 2.5))
4         d2l.plt.plot(x_vals.detach().numpy(), y_vals.detach().numpy())
5         d2l.plt.xlabel('x')
6         d2l.plt.ylabel(name+'(x)')

```

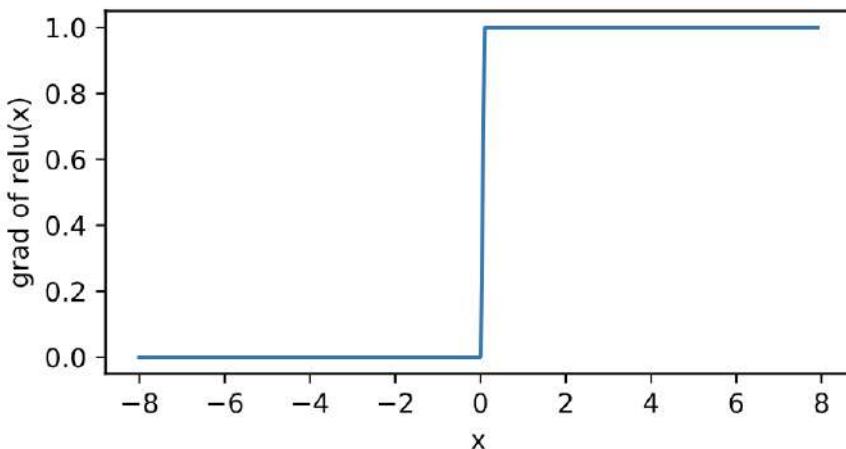
我们接下来通过Tensor提供的`relu`函数来绘制ReLU函数。可以看到，该激活函数是一个两段线性函数。

```
1 In [2]:  
2     x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)  
3     y = x.relu()  
4     xyplot(x, y, 'relu')
```



显然，当输入为负数时，ReLU函数的导数为0；当输入为正数时，ReLU函数的导数为1。尽管输入为0时ReLU函数不可导（左导数为0，右导数为1，左右导数不相等故而该点不可导），但是我们可以取此处的导数为0。下面绘制ReLU函数的导数。

```
1 In [3]:  
2     y.sum().backward()  
3     xyplot(x, x.grad, 'grad of relu')
```



## 2. sigmoid函数

sigmoid函数可以将元素的值变换到0和1之间：

$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}$$

sigmoid函数在早期的神经网络中较为普遍，但它目前逐渐被更简单的ReLU函数取代。在后面“循环神经网络”一章中我们会介绍如何利用它值域在0到1之间这一特性来控制信息在神经网络中的流动。下面绘制了sigmoid函数。当输入接近0时，sigmoid函数接近线性变换

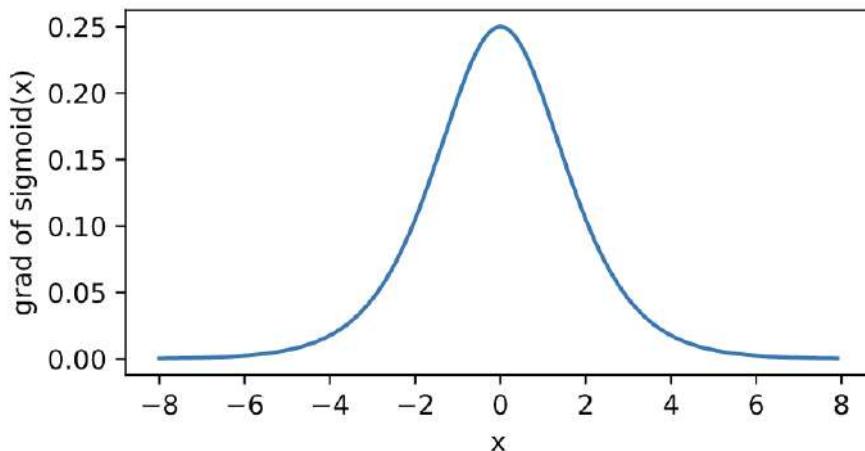
```
1 In [4]:  
2     y = x.sigmoid()  
3     xyplot(x, y, 'sigmoid')
```

依据链式法则，sigmoid函数的导数为

$$\text{sigmoid}'(x) = \text{sigmoid}(x)(1 - \text{sigmoid}(x))$$

下面绘制了sigmoid函数的导数。当输入为0时，sigmoid函数的导数达到最大值0.25；当输入越偏离0时，sigmoid函数的导数越接近0。

```
1 In [5]:  
2     x.grad.zero_()  
3     y.sum().backward()  
4     xyplot(x, x.grad, 'grad of sigmoid')
```



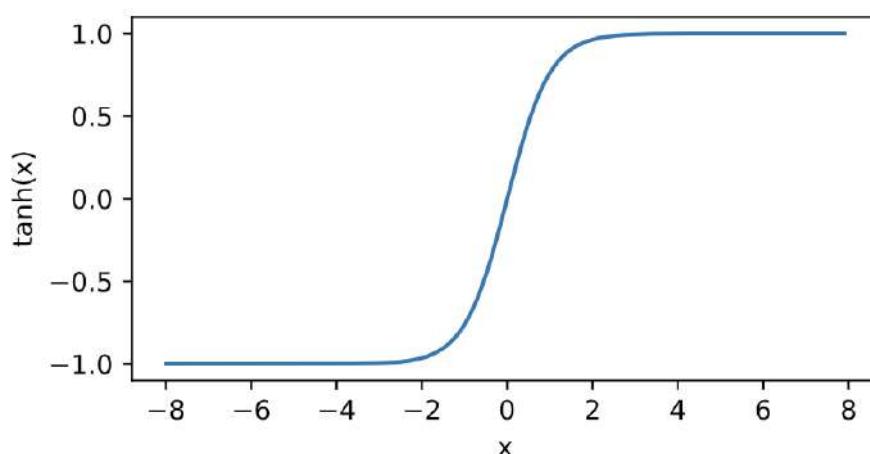
### 3. tanh函数

tanh（双曲正切）函数可以将元素的值变换到-1和1之间：

$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}$$

我们接着绘制tanh函数。当输入接近0时，tanh函数接近线性变换。虽然该函数的形状和sigmoid函数的形状很像，但tanh函数在坐标系的原点上对称。

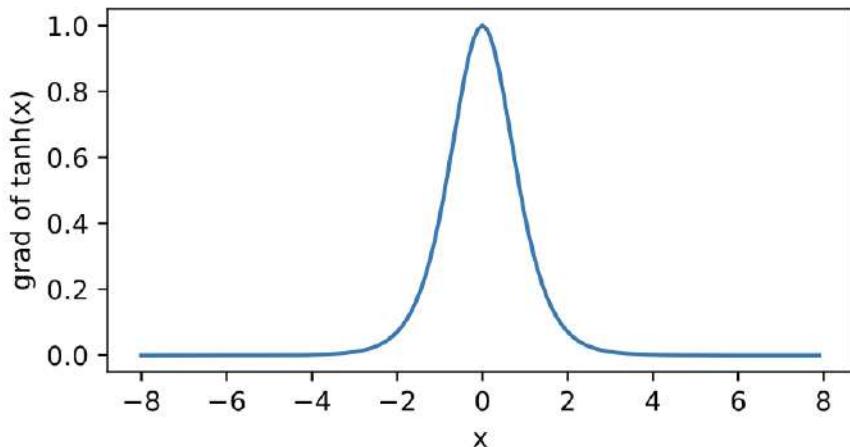
```
1 In [6]:  
2     y = x.tanh()  
3     xyplot(x, y, 'tanh')
```



依据链式法则， $\tanh$ 函数的导数为

$$\tanh'(x) = 1 - \tanh^2(x)$$

下面绘制了 $\tanh$ 函数的导数。当输入为0时， $\tanh$ 函数的导数达到最大值1；当输入越偏离0时， $\tanh$ 函数的导数越接近0。



### 2.8.3 多层感知机

多层感知机就是含有至少一个隐藏层的由全连接层组成的神经网络，且每个隐藏层的输出通过激活函数进行变换。多层感知机的层数和各隐藏层中隐藏单元个数都是超参数。以单隐藏层为例并沿用本节之前定义的符号，多层感知机按以下方式计算输出：

$$\begin{aligned}\mathbf{H} &= \phi(\mathbf{XW}_h + \mathbf{b}_h) \\ \mathbf{O} &= \mathbf{HW}_o + \mathbf{b}_o\end{aligned}$$

其中 $\phi$ 表示激活函数。在分类问题中，我们可以对输出 $\mathbf{O}$ 做softmax运算，并使用softmax回归中的交叉熵损失函数。在回归问题中，我们将输出层的输出个数设为1，并将输出 $\mathbf{O}$ 直接提供给线性回归中使用的平方损失函数。

**小结：**

- 多层感知机在输出层与输入层之间加入了一个或多个全连接隐藏层，并通过激活函数对隐藏层输出进行变换。
- 常用的激活函数包括ReLU函数、sigmoid函数和tanh函数。

## 2.9 多层感知机的从零开始实现

我们已经从上一节里了解了多层感知机的原理。下面，我们一起来动手实现一个多层感知机。

### 2.9.1 读取数据

这里继续使用Fashion-MNIST数据集。我们将使用多层感知机对图像进行分类。

```
1 In [1]:  
2     batch_size = 256  
3     train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

## 2.9.2 定义模型参数

我们在2.5节里已经介绍了，Fashion-MNIST数据集中图像形状为 $28 \times 28$ ，类别数为10。本节中我们依然使用长度为 $28 \times 28 = 784$ 的向量表示每一张图像。因此，输入个数为784，输出个数为10。实验中，我们设超参数隐藏单元个数为256。

```
1 In [2]:  
2     num_inputs, num_outputs, num_hiddens = 784, 10, 256  
3     w1 = torch.tensor(np.random.normal(0, 0.01, (num_inputs, num_hiddens)),  
4                         dtype=torch.float)  
5     b1 = torch.zeros(num_hiddens, dtype=torch.float)  
6     w2 = torch.tensor(np.random.normal(0, 0.01, (num_hiddens, num_outputs)),  
7                         dtype=torch.float)  
8     b2 = torch.zeros(num_outputs, dtype=torch.float)  
9     params = [w1, b1, w2, b2]  
10    for param in params:  
11        param.requires_grad_(requires_grad=True)
```

## 2.9.3 定义激活函数

这里我们使用基础的 max 函数来实现ReLU，而非直接调用 relu 函数。

```
1 In [3]:  
2     def relu(x):  
3         return torch.max(input=x, other=torch.tensor(0.0))
```

## 2.9.4 定义模型

同softmax回归一样，我们通过 view 函数将每张原始图像改成长度为 num\_inputs 的向量。然后我们实现上一节中多层感知机的计算表达式。

```
1 In [4]:  
2     def net(x):  
3         x = x.view((-1, num_inputs))  
4         h = relu(torch.matmul(x, w1) + b1)  
5         return torch.matmul(h, w2) + b2
```

## 2.9.5 定义损失函数

为了得到更好的数值稳定性，我们直接使用PyTorch提供的包括softmax运算和交叉熵损失计算的函数。

```
1 In [5]:  
2     loss = torch.nn.CrossEntropyLoss()
```

## 2.9.6 训练模型

训练多层感知机的步骤和2.6节中训练softmax回归的步骤没什么区别。我们直接调用 d2lzh 包中的 train\_ch3 函数，它的实现已经在2.6节里介绍过。我们在这里设超参数迭代周期数为5，学习率为 100.0。

由于原书的mxnet中的 softmaxCrossEntropyLoss 在反向传播的时候相对于沿batch维求和了，而 PyTorch默认的是求平均，所以用PyTorch计算得到的loss比mxnet小很多（大概是maxnet计算得到的 $1/batch\_size$ 这个量级），所以反向传播得到的梯度也小很多，所以为了得到差不多的学习效果，我们把学习率调得成原书的约batch\_size倍，原书的学习率为0.5，这里设置成100.0。(之所以这么大，应该是因为d2lzh里面的sgd函数在更新的时候除以了batch\_size，其实PyTorch在计算loss的时候已经除过一次了，sgd这里应该不用除了)。

```
1 In [6]:  
2     num_epochs, lr = 5, 100.0  
3     d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs,  
4                     batch_size, params, lr)  
5 Out [6]:  
6     epoch 1, loss 0.0031, train acc 0.709, test acc 0.761  
7     epoch 2, loss 0.0019, train acc 0.823, test acc 0.743  
8     epoch 3, loss 0.0017, train acc 0.844, test acc 0.793  
9     epoch 4, loss 0.0015, train acc 0.855, test acc 0.830  
10    epoch 5, loss 0.0015, train acc 0.862, test acc 0.848
```

## 小结:

- 可以通过手动定义模型及其参数来实现简单的多层感知机。
- 当多层感知机的层数较多时，本节的实现方法会显得较繁琐，例如在定义模型参数的时候。

## 2.10 多层感知机的简洁实现

下面我们使用PyTorch来实现上一节中的多层感知机。

### 2.10.1 定义模型

和softmax回归唯一的不同在于，我们多加了一个全连接层作为隐藏层。它的隐藏单元个数为256，并使用ReLU函数作为激活函数。

```
1 In [1]:  
2     num_inputs, num_outputs, num_hiddens = 784, 10, 256  
3     net = nn.Sequential(  
4         d2l.FlattenLayer(),  
5         nn.Linear(num_inputs, num_hiddens),  
6         nn.ReLU(),  
7         nn.Linear(num_hiddens, num_outputs)  
8     )  
9     for params in net.parameters():  
10        init.normal_(params, mean=0, std=0.01)
```

### 2.10.2 训练模型

我们使用与2.7节中训练softmax回归几乎相同的步骤来读取数据并训练模型。

```
1 In [2]:  
2     batch_size = 256  
3     train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)  
4     loss = torch.nn.CrossEntropyLoss()  
5     optimizer = torch.optim.SGD(net.parameters(), lr=0.5)  
6     num_epochs = 5  
7     d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs,  
8                   batch_size, None, None, optimizer)
```

```
9  out [2]:  
10     epoch 1, loss 0.0031, train acc 0.701, test acc 0.788  
11     epoch 2, loss 0.0019, train acc 0.819, test acc 0.818  
12     epoch 3, loss 0.0017, train acc 0.842, test acc 0.826  
13     epoch 4, loss 0.0015, train acc 0.856, test acc 0.848  
14     epoch 5, loss 0.0014, train acc 0.863, test acc 0.826
```

## 小结:

- 通过PyTorch可以更简洁地实现多层感知机。

## 2.11 模型选择、欠拟合和过拟合

在前几节基于Fashion-MNIST数据集的实验中，我们评价了机器学习模型在训练数据集和测试数据集上的表现。如果你改变过实验中的模型结构或者超参数，你也许发现了：当模型在训练数据集上更准确时，它在测试数据集上却不一定更准确。这是为什么呢？

### 2.11.1 训练误差和泛化误差

在解释上述现象之前，我们需要区分训练误差（training error）和泛化误差（generalization error）。通俗来讲，前者指模型在训练数据集上表现出的误差，后者指模型在任意一个测试数据样本上表现出的误差的期望，并常常通过测试数据集上的误差来近似。计算训练误差和泛化误差可以使用之前介绍过的损失函数，例如线性回归用到的平方损失函数和softmax回归用到的交叉熵损失函数。

让我们以高考为例来直观地解释训练误差和泛化误差这两个概念。训练误差可以认为是做往年高考试题（训练题）时的错误率，泛化误差则可以通过真正参加高考（测试题）时的答题错误率来近似。假设训练题和测试题都随机采样于一个未知的依照相同考纲的巨大试题库。如果让一名未学习中学知识的小学生去答题，那么测试题和训练题的答题错误率可能很相近。但如果换成一名反复练习训练题的高三备考生答题，即使在训练题上做到了错误率为0，也不代表真实的高考成绩会如此。

在机器学习里，我们通常假设训练数据集（训练题）和测试数据集（测试题）里的每一个样本都是从同一个概率分布中相互独立地生成的。基于该独立同分布假设，给定任意一个机器学习模型（含参数），它的训练误差的期望和泛化误差都是一样的。例如，如果我们将模型参数设成随机值（小学生），那么训练误差和泛化误差会非常相近。但我们从前面几节中已经了解到，模型的参数是通过在训练数据集上训练模型而学习出的，参数的选择依据了最小化训练误差（高三备考生）。所以，训练误差的期望小于或等于泛化误差。也就是说，一般情况下，由训练数据集学到的模型参数会使模型在训练数据集上的表现优于或等于在测试数据集上的表现。由于无法从训练误差估计泛化误差，一味地降低训练误差并不意味着泛化误差一定会降低。

机器学习模型应关注降低泛化误差。

### 2.11.2 模型选择

在机器学习中，通常需要评估若干候选模型的表现并从中选择模型。这一过程称为模型选择（model selection）。可供选择的候选模型可以是有着不同超参数的同类模型。以多层感知机为例，我们可以选择隐藏层的个数，以及每个隐藏层中隐藏单元个数和激活函数。为了得到有效的模型，我们通常要在模型选择上下一番功夫。下面，我们来描述模型选择中经常使用的验证数据集（validation data set）。

#### 1. 验证数据集

从严格意义上讲，测试集只能在所有超参数和模型参数选定后使用一次。不可以使用测试数据选择模型，如调参。由于无法从训练误差估计泛化误差，因此也不应只依赖训练数据选择模型。鉴于此，我们可以预留一部分在训练数据集和测试数据集以外的数据来进行模型选择。这部分数据被称为验证数据集，简称验证集（validation set）。例如，我们可以从给定的训练集中随机选取一小部分作为验证集，而将剩余部分作为真正的训练集。

然而在实际应用中，由于数据不容易获取，测试数据极少只使用一次就丢弃。因此，实践中验证数据集和测试数据集的界限可能比较模糊。从严格意义上讲，除非明确说明，否则本书中实验所使用的测试集应为验证集，实验报告的测试结果（如测试准确率）应为验证结果（如验证准确率）。

## 2. K折交叉验证

由于验证数据集不参与模型训练，当训练数据不够用时，预留大量的验证数据显得太奢侈。一种改善的方法是  $K$  折交叉验证 ( $K$ -fold cross-validation)。在  $K$  折交叉验证中，我们把原始训练数据集分割成  $K$  个不重合的子数据集，然后我们做  $K$  次模型训练和验证。每一次，我们使用一个子数据集验证模型，并使用其他  $K - 1$  个子数据集来训练模型。在这  $K$  次训练和验证中，每次用来验证模型的子数据集都不同。最后，我们对这  $K$  次训练误差和验证误差分别求平均。

## 2.11.3 欠拟合和过拟合

接下来，我们将探究模型训练中经常出现的两类典型问题：一类是模型无法得到较低的训练误差，我们将这一现象称作欠拟合 (underfitting)；另一类是模型的训练误差远小于它在测试数据集上的误差，我们称该现象为过拟合 (overfitting)。在实践中，我们要尽可能同时应对欠拟合和过拟合。虽然有很多因素可能导致这两种拟合问题，在这里我们重点讨论两个因素：**模型复杂度**和**训练数据集大小**。

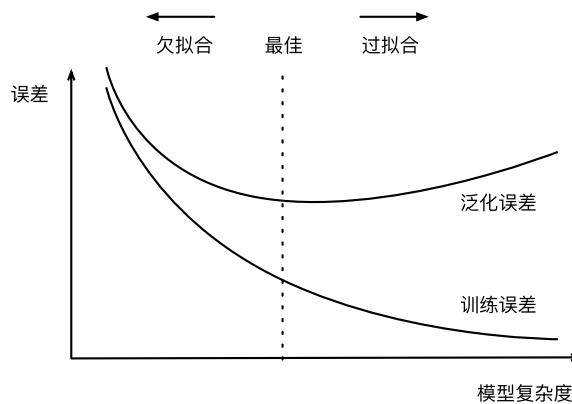
### 1. 模型复杂度

为了解释模型复杂度，我们以多项式函数拟合为例。给定一个由标量数据特征  $x$  和对应的标量标签  $y$  组成的训练数据集，多项式函数拟合的目标是找一个  $K$  阶多项式函数

$$\hat{y} = b + \sum_{k=1}^K x^k w_k$$

来近似  $y$ 。在上式中， $w_k$  是模型的权重参数， $b$  是偏差参数。与线性回归相同，多项式函数拟合也使用平方损失函数。特别地，一阶多项式函数拟合又叫线性函数拟合。

因为高阶多项式函数模型参数更多，模型函数的选择空间更大，所以高阶多项式函数比低阶多项式函数的复杂度更高。因此，高阶多项式函数比低阶多项式函数更容易在相同的训练数据集上得到更低的训练误差。给定训练数据集，模型复杂度和误差之间的关系通常如下图所示。给定训练数据集，如果模型的复杂度过低，很容易出现欠拟合；如果模型复杂度过高，很容易出现过拟合。应对欠拟合和过拟合的一个办法是针对数据集选择合适复杂度的模型。



### 2. 训练数据集大小

影响欠拟合和过拟合的另一个重要因素是训练数据集的大小。一般来说，如果训练数据集中样本数过少，特别是比模型参数数量（按元素计）更少时，过拟合更容易发生。此外，泛化误差不会随训练数据集里样本数量增加而增大。因此，在计算资源允许的范围之内，我们通常希望训练数据集大一些，特别是在模型复杂度较高时，例如层数较多的深度学习模型。

## 2.11.4 多项式函数拟合实验

为了理解模型复杂度和训练数据集大小对欠拟合和过拟合的影响，下面我们以多项式函数拟合为例来实验。

### 1. 生成数据集

我们将生成一个人工数据集。在训练数据集和测试数据集中，给定样本特征 $x$ ，我们使用如下的三阶多项式函数来生成该样本的标签：

$$y = 1.2x - 3.4x^2 + 5.6x^3 + 5 + \epsilon$$

其中噪声项 $\epsilon$ 服从均值为0、标准差为0.01的正态分布。训练数据集和测试数据集的样本数都设为100。

```
1 In [1]:  
2     n_train, n_test, true_w, true_b = 100, 100, [1.2, -3.4, 5.6], 5  
3     features = torch.randn((n_train+n_test, 1))  
4     poly_features = torch.cat((features,  
5                                torch.pow(features, 2),  
6                                torch.pow(features, 3)), 1)  
7     labels = (true_w[0]*poly_features[:, 0]+  
8                true_w[1]*poly_features[:, 1]+  
9                true_w[2]*poly_features[:, 2]+  
10               true_b)  
11    labels += torch.tensor(np.random.normal(0, 0.01, size=labels.size()),  
12                           dtype=torch.float)
```

看一看生成的数据集的前两个样本。

```
1 In [2]:  
2     features[:2], poly_features[:2], labels[:2]  
3 Out [2]:  
4     (tensor([[1.5090],  
5                [0.4379]]),  
6      tensor([[1.5090, 2.2772, 3.4364],  
7                [0.4379, 0.1918, 0.0840]]),  
8      tensor([18.3045, 5.3469]))
```

### 2. 定义、训练和测试模型

我们先定义作图函数 `semilogy`，其中  $y$  轴使用了对数尺度。

```
1 In [3]:  
2     def semilogy(x_vals, y_vals, x_label, y_label,  
3                     x2_vals=None, y2_vals=None, legend=None,  
4                     figsize=(3.5, 2.5)):  
5         d2l.set_figsize(figsize)  
6         d2l.plt.xlabel(x_label)  
7         d2l.plt.ylabel(y_label)  
8         # 对数坐标  
9         d2l.plt.semilogy(x_vals, y_vals)  
10        # 绘制测试集上的曲线  
11        if x2_vals and y2_vals:  
12            d2l.plt.semilogy(x2_vals, y2_vals, linestyle=':')13            d2l.plt.legend(legend)
```

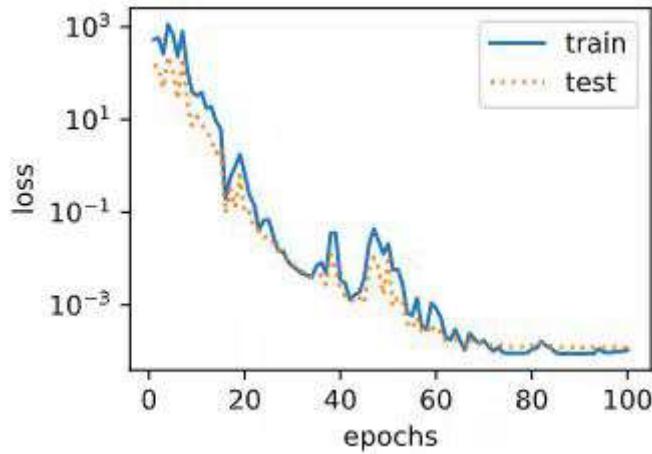
和线性回归一样，多项式函数拟合也使用平方损失函数。因为我们将尝试使用不同复杂度的模型来拟合生成的数据集，所以我们把模型定义部分放在 `fit_and_plot` 函数中。多项式函数拟合的训练和测试步骤与2.6节（softmax回归的从零开始实现）介绍的softmax回归中的相关步骤类似。

```
1 In [4]:  
2     num_epochs, loss = 100, torch.nn.MSELoss()  
3     def fit_and_plot(train_features, test_features, train_labels,  
4                         test_labels):  
5         net = torch.nn.Linear(train_features.shape[-1], 1)  
6         batch_size = min(10, train_labels.shape[0])  
7         dataset = torch.utils.data.TensorDataset(train_features,  
8                                         train_labels)  
9         train_iter = torch.utils.data.DataLoader(dataset, batch_size,  
10                                         shuffle=True)  
11         optimizer = torch.optim.SGD(net.parameters(), lr=0.01)  
12         train_ls, test_ls = [], []  
13         for _ in range(num_epochs):  
14             for x, y in train_iter:  
15                 l = loss(net(x), y.view(-1, 1))  
16                 optimizer.zero_grad()  
17                 l.backward()  
18                 optimizer.step()  
19                 train_labels = train_labels.view(-1, 1)  
20                 test_labels = test_labels.view(-1, 1)  
21                 train_ls.append(loss(net(train_features), train_labels).item())  
22                 test_ls.append(loss(net(test_features), test_labels).item())  
23             print('final epoch: train loss', train_ls[-1],  
24                  'test loss', test_ls[-1])  
25             semilogy(range(1, num_epochs+1), train_ls, 'epochs', 'loss',  
26                     range(1, num_epochs+1), test_ls, ['train', 'test'])  
27             print('weight:', net.weight.data, '\nbias:', net.bias.data)
```

### 3. 三阶多项式函数拟合（正常）

我们先使用与数据生成函数同阶的三阶多项式函数拟合。实验表明，这个模型的训练误差和在测试数据集的误差都较低。训练出的模型参数也接近真实值： $w_1 = 1.2, w_2 = -3.4, w_3 = 5.6, b = 5$ 。

```
1 In [5]:  
2     fit_and_plot(poly_features[:n_train, :],  
3                     poly_features[n_train:, :],  
4                     labels[:n_train],  
5                     labels[n_train:])  
6 Out [5]:  
7     final epoch: train loss 0.00010534885950619355  
8                     test loss 0.00011935166548937559  
9     weight: tensor([[ 1.1978, -3.4011,  5.6006]])  
10    bias: tensor([4.9995])
```



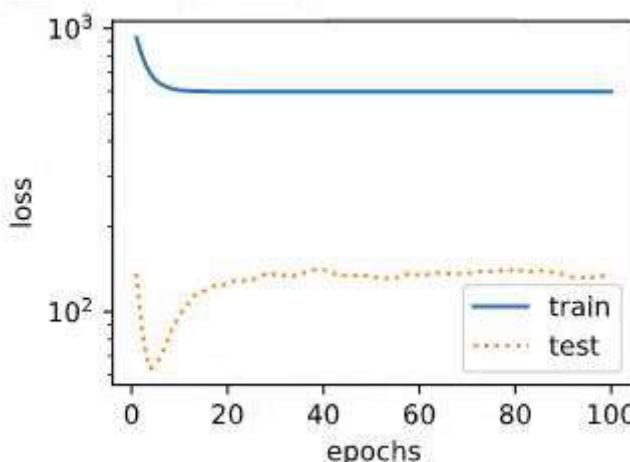
#### 4. 线性函数拟合（欠拟合）

我们再试试线性函数拟合。很明显，该模型的训练误差在迭代早期下降后便很难继续降低。在完成最后一次迭代周期后，训练误差依旧很高。线性模型在非线性模型（如三阶多项式函数）生成的数据集上容易欠拟合。

```

1 In [6]:
2     fit_and_plot(features[:n_train, :],
3                     features[n_train:, :],
4                     labels[:n_train],
5                     labels[n_train:])
6 Out [6]:
7     final epoch: train loss 598.3652954101562 test loss 135.366943359375
8     weight: tensor([[22.6452]])
9     bias: tensor([0.2046])

```



#### 5. 训练样本不足（过拟合）

事实上，即便使用与数据生成模型同阶的三阶多项式函数模型，如果训练样本不足，该模型依然容易过拟合。让我们只使用两个样本来训练模型。显然，训练样本过少了，甚至少于模型参数的数量。这使模型显得过于复杂，以至于容易被训练数据中的噪声影响。在迭代过程中，尽管训练误差较低，但是测试数据集上的误差却很高。这是典型的过拟合现象。

```

1 In [7]:
2     fit_and_plot(poly_features[:, :],
3                   poly_features[n_train:, :],
4                   labels[:, :],
5                   labels[n_train:])
6 Out [7]:
7     final epoch: train loss 0.6954999566078186 test loss 40.358909606933594
8     weight: tensor([[2.0820, 1.9556, 2.3756]])
9     bias: tensor([2.6901])

```

我们将在接下来的两个小节继续讨论过拟合问题以及应对过拟合的方法。

### 小结:

- 由于无法从训练误差估计泛化误差，一味地降低训练误差并不意味着泛化误差一定会降低。机器学习模型应关注降低泛化误差。
- 可以使用验证数据集来进行模型选择。
- 欠拟合指模型无法得到较低的训练误差，过拟合指模型的训练误差远小于它在测试数据集上的误差。
- 应选择复杂度合适的模型并避免使用过少的训练样本。

## 2.12 权重衰减

上一节中我们观察了过拟合现象，即模型的训练误差远小于它在测试集上的误差。虽然增大训练数据集可能会减轻过拟合，但是获取额外的训练数据往往代价高昂。本节介绍应对过拟合问题的常用方法：权重衰减（weight decay）。

### 2.12.1 方法

权重衰减等价于  $L_2$  范数正则化（regularization）。**正则化通过为模型损失函数添加惩罚项使学出的模型参数值较小，是应对过拟合的常用手段。（动态可视化）** 我们先描述  $L_2$  范数正则化，再解释它为何又称权重衰减。

越复杂的模型，越是会尝试对所有的样本进行拟合，甚至包括一些异常样本点，这就容易造成在较小的区间里预测值产生较大的波动，这种较大的波动也反映了在这个区间里的导数很大，而只有较大的参数值才能产生较大的导数。因此复杂的模型，其参数值会比较大。而更小的参数值下面，我们以高维线性回归为例来引入一个过拟合问题，并使用权重衰减来应对过拟合。设数据样本特征的维度为  $p$ 。对于训练数据集和测试数据集中特征为  $x_1, x_2, \dots, x_p$  的任一样本，我们使用如下的线性函数来生成该样本的标签：意味着模型的复杂度更低，对训练数据的拟合刚刚好（奥卡姆剃刀），不会过分拟合训练数据，从而使得不会过拟合，以提高模型的泛化能力。

$L_2$  范数正则化在模型原损失函数基础上添加  $L_2$  范数惩罚项，从而得到训练所需要最小化的函数。  
 **$L_2$  范数惩罚项指的是模型权重参数每个元素的平方和与一个正的常数的乘积。** 以 2.1 节（线性回归）中的线性回归损失函数

$$\ell(w_1, w_2, b) = \frac{1}{n} \sum_{i=1}^n \frac{1}{2} \left( x_1^{(i)} w_1 + x_2^{(i)} w_2 + b - y^{(i)} \right)^2$$

为例，其中  $w_1, w_2$  是权重参数， $b$  是偏差参数，样本  $i$  的输入为  $x_1^{(i)}, x_2^{(i)}$ ，标签为  $y^{(i)}$ ，样本数为  $n$ 。将权重参数用向量  $\mathbf{w} = [w_1, w_2]$  表示，带有  $L_2$  范数惩罚项的新损失函数为

$$\ell(w_1, w_2, b) + \frac{\lambda}{2n} \|\mathbf{w}\|^2$$

其中超参数 $\lambda > 0$ 。当权重参数均为0时，惩罚项最小。**当 $\lambda$ 较大时，惩罚项在损失函数中的比重较大，这通常会使学到的权重参数的元素较接近0。**当 $\lambda$ 设为0时，惩罚项完全不起作用。上式中 $L_2$ 范数平方 $\|\mathbf{w}\|^2$ 展开后得到 $w_1^2 + w_2^2$ 。有了 $L_2$ 范数惩罚项后，在小批量随机梯度下降中，我们将线性回归一节中权重 $w_1$ 和 $w_2$ 的迭代方式更改为

$$w_1 \leftarrow \left(1 - \frac{\eta\lambda}{|\mathcal{B}|}\right) w_1 - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} x_1^{(i)} \left(x_1^{(i)} w_1 + x_2^{(i)} w_2 + b - y^{(i)}\right)$$

$$w_2 \leftarrow \left(1 - \frac{\eta\lambda}{|\mathcal{B}|}\right) w_2 - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} x_2^{(i)} \left(x_1^{(i)} w_1 + x_2^{(i)} w_2 + b - y^{(i)}\right)$$

可见， $L_2$ 范数正则化令权重 $w_1$ 和 $w_2$ 先自乘小于1的数，再减去不含惩罚项的梯度。因此， $L_2$ 范数正则化又叫权重衰减。**权重衰减通过惩罚绝对值较大的模型参数为需要学习的模型增加了限制，这可能对过拟合有效。实际场景中，我们有时也在惩罚项中添加偏差元素的平方和。**

## 2.12.2 高维线性回归实验

下面，我们以高维线性回归为例来引入一个过拟合问题，并使用权重衰减来应对过拟合。设数据样本特征的维度为 $p$ 。对于训练数据集和测试数据集中特征为 $x_1, x_2, \dots, x_p$ 的任一样本，我们使用如下的线性函数来生成该样本的标签：

$$y = 0.05 + \sum_{i=1}^p 0.01x_i + \epsilon$$

其中噪声项 $\epsilon$ 服从均值为0、标准差为0.01的正态分布。为了较容易地观察过拟合，我们考虑高维线性回归问题，如设维度 $p = 200$ ；同时，我们特意把训练数据集的样本数设低，如20。

```

1 In [1]:
2     n_train, n_test, num_inputs = 20, 100, 200
3     true_w, true_b = torch.ones(num_inputs, 1) * 0.01, 0.05
4     features = torch.randn((n_train+n_test, num_inputs))
5     labels = torch.matmul(features, true_w) + true_b
6     labels += torch.tensor(np.random.normal(0, 0.01, size=labels.size())),
7                         dtype=torch.float)
8     train_features = features[:n_train, :]
9     test_features = features[n_train:, :]
10    train_labels, test_labels = labels[:n_train], labels[n_train:]
```

## 2.12.3 从零开始实现

下面先介绍从零开始实现权重衰减的方法。我们通过在目标函数后添加 $L_2$ 范数惩罚项来实现权重衰减。

### 1. 初始化模型参数

首先，定义随机初始化模型参数的函数。该函数为每个参数都附上梯度。

```

1 In [2]:
2     def init_params():
3         w = torch.randn((num_inputs, 1), requires_grad=True)
4         b = torch.zeros(1, requires_grad=True)
5         return [w, b]
```

### 2. 定义 $L_2$ 范数惩罚项

下面定义 $L_2$ 范数惩罚项。这里只惩罚模型的权重参数。

```
1 In [3]:  
2     def l2_penalty(w):  
3         return (w**2).sum() / 2
```

### 3. 定义训练和测试

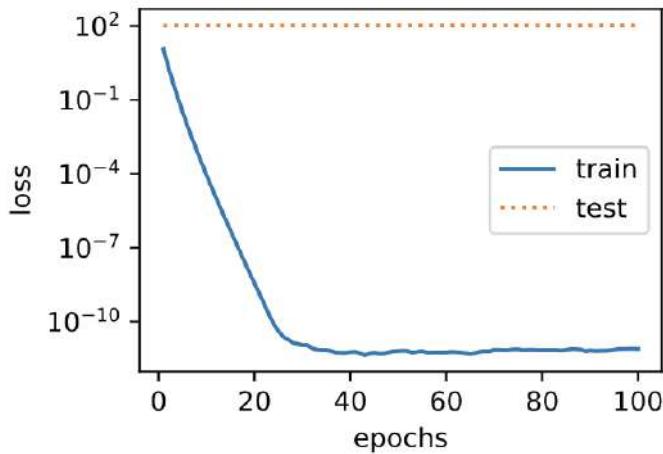
下面定义如何在训练数据集和测试数据集上分别训练和测试模型。与前面几节中不同的是，这里在计算最终的损失函数时添加了 $L_2$ 范数惩罚项。

```
1 In [4]:  
2     batch_size, num_epochs, lr = 1, 100, 0.003  
3     net, loss = d2l.linreg, d2l.squared_loss  
4     dataset = torch.utils.data.TensorDataset(train_features, train_labels)  
5     train_iter = torch.utils.data.DataLoader(dataset, batch_size,  
6                                              shuffle=True)  
7     def fit_and_plot(lambd):  
8         w, b = init_params()  
9         train_ls, test_ls = [], []  
10        for _ in range(num_epochs):  
11            for X, y in train_iter:  
12                # 添加L2范数惩罚项  
13                l = loss(net(X, w, b), y) + lambd * l2_penalty(w)  
14                l = l.sum()  
15                if w.grad is not None:  
16                    w.grad.data.zero_()  
17                    b.grad.data.zero_()  
18                    l.backward()  
19                    d2l.sgd([w, b], lr, batch_size)  
20                    train_ls.append(loss(net(train_features, w, b),  
21                                         train_labels).mean().item())  
22                    test_ls.append(loss(net(test_features, w, b),  
23                                         test_labels).mean().item())  
24                    d2l.semilogy(range(1, num_epochs+1), train_ls, 'epochs', 'loss',  
25                                range(1, num_epochs+1), test_ls, ['train', 'test'])  
26                    print('L2 norm of w: ', w.norm().item())
```

### 4. 观察过拟合

接下来，让我们训练并测试高维线性回归模型。当 $\lambda$ 设为0时，我们没有使用权重衰减。结果训练误差远小于测试集上的误差。这是典型的过拟合现象。

```
1 In [5]:  
2     fit_and_plot(lambd=0)  
3 Out [5]:  
4     L2 norm of w:  12.592584609985352
```



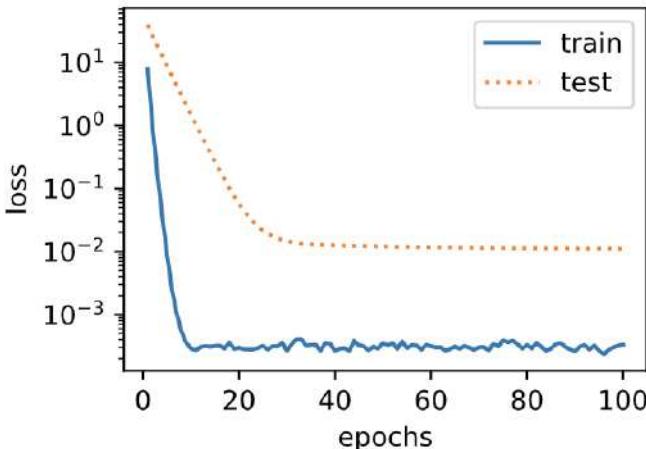
## 5. 使用权重衰减

下面我们使用权重衰减。可以看出，训练误差虽然有所提高，但测试集上的误差有所下降。过拟合现象得到一定程度的缓解。另外，权重参数的 $L_2$ 范数比不使用权重衰减时的更小，此时的权重参数更接近0。

```

1 In [6]:
2     fit_and_plot(lambd=3)
3 Out [6]:
4     L2 norm of w:  0.03535861149430275

```



## 2.12.4 简洁实现

这里我们直接在构造优化器实例时通过 `weight_decay` 参数来指定权重衰减超参数。默认下，PyTorch会对权重和偏差同时衰减。我们可以分别对权重和偏差构造优化器实例，从而只对权重衰减。

```

1 In [7]:
2     def fit_and_plot_pytorch(wd):
3         net = nn.Linear(num_inputs, 1)
4         nn.init.normal_(net.weight, mean=0, std=1)
5         nn.init.normal_(net.bias, mean=0, std=1)
6         # 对权重参数衰减
7         optimizer_w = torch.optim.SGD(params=[net.weight], lr=lr,
8                                         weight_decay=wd)
9         optimizer_b = torch.optim.SGD(params=[net.bias], lr=lr)
10        train_ls, test_ls = [], []
11        for _ in range(num_epochs):
12            for x, y in train_iter:
13                l = loss(net(x), y).mean()

```

```

14         optimizer_w.zero_grad()
15         optimizer_b.zero_grad()
16         l.backward()
17         # 对两个optimizer实例分别调用step函数，从而分别更新权重和偏差
18         optimizer_w.step()
19         optimizer_b.step()
20         train_ls.append(loss(net(train_features),
21                               train_labels).mean().item())
22         test_ls.append(loss(net(test_features),
23                             test_labels).mean().item())
24         d2l.semilogy(range(1, num_epochs+1), train_ls, 'epochs', 'loss',
25                     range(1, num_epochs+1), test_ls, ['train', 'test'])
26         print('L2 norm of w: ', net.weight.data.norm().item())

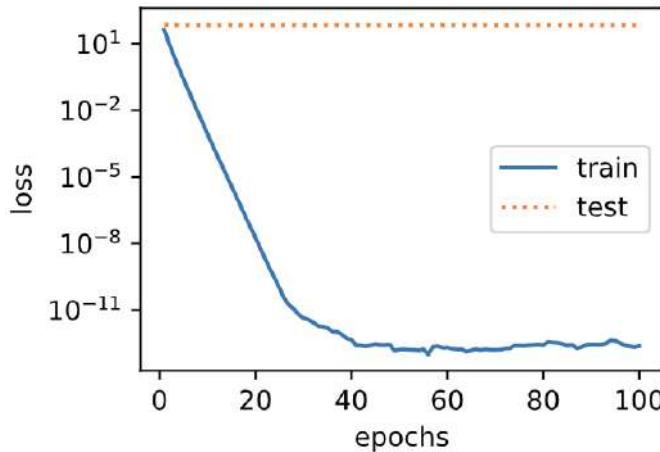
```

与从零开始实现权重衰减的实验现象类似，使用权重衰减可以在一定程度上缓解过拟合问题。

```

1 In [8]:
2     fit_and_plot_pytorch(0)
3 Out [8]:
4     L2 norm of w:  14.631454467773438

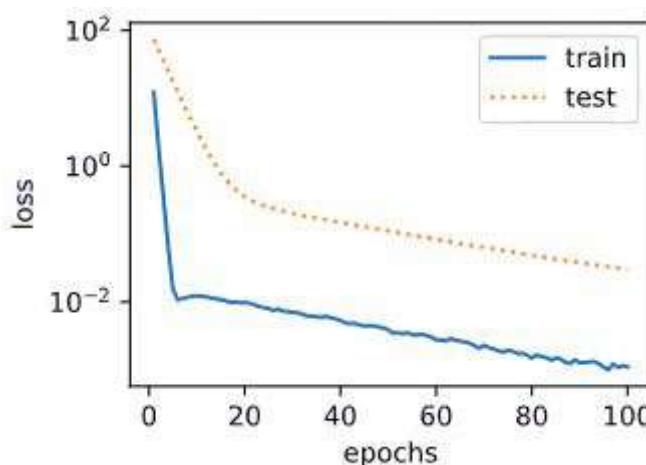
```



```

1 In [9]:
2     fit_and_plot_pytorch(3)
3 Out [9]:
4     L2 norm of w:  0.063807912170887

```



## 小结：

- 正则化通过为模型损失函数添加惩罚项使学出的模型参数值较小，是应对过拟合的常用手段。

- 权重衰减等价于  $L_2$  范数正则化，通常会使学到的权重参数的元素较接近0。
- 权重衰减可以通过优化器中的 `weight_decay` 超参数来指定。
- 可以定义多个优化器实例对不同的模型参数使用不同的迭代方法。

## 2.13 丢弃法

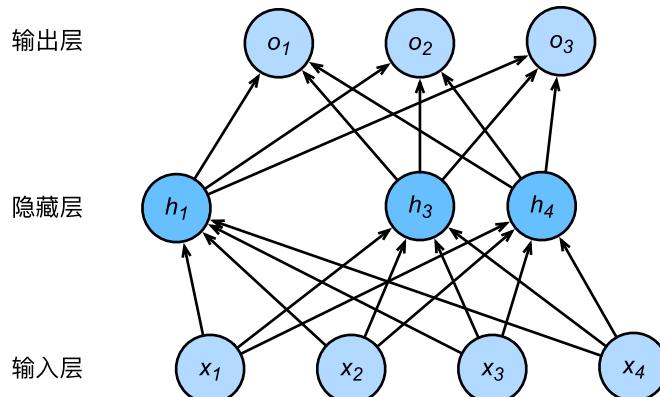
除了前一节介绍的权重衰减以外，深度学习模型常常使用丢弃法（dropout）来应对过拟合问题。丢弃法有一些不同的变体。本节中提到的丢弃法特指倒置丢弃法（inverted dropout）。

### 2.13.1 方法

回忆一下，2.8节（多层感知机）的图中描述了一个单隐藏层的多层感知机。其中输入个数为4，隐藏单元个数为5，且隐藏单元  $h_i$  ( $i = 1, \dots, 5$ ) 的计算表达式为

$$h_i = \phi(x_1 w_{1i} + x_2 w_{2i} + x_3 w_{3i} + x_4 w_{4i} + b_i)$$

这里  $\phi$  是激活函数， $x_1, \dots, x_4$  是输入，隐藏单元  $i$  的权重参数为  $w_{1i}, \dots, w_{4i}$ ，偏差参数为  $b_i$ 。当对该隐藏层使用丢弃法时，该层的隐藏单元将有一定概率被丢弃掉。设丢弃概率为  $p$ ，那么  $h_i$  有  $p$  的概率会被清零，有  $1 - p$  的概率会除以  $1 - p$  做拉伸。丢弃概率是丢弃法的超参数。我们这里说一下为什么要除以  $1 - p$ ：在使用丢弃法之前，某一个神经元输出的期望值为  $h_i$ ，如果我们以概率  $p$  对其值进行清零，那么该神经元输出值的期望为  $0 \times p + 1 \times (1 - p) \times h_i = (1 - p) \times h_i$ ，为了保证神经元输出激活值的期望值与不使用 dropout 时一致，我们需要对神经元输出值除以  $(1 - p)$ 。让我们对2.8节中图的隐藏层使用丢弃法，一种可能的结果如下图所示，其中  $h_2$  和  $h_5$  被清零。这时输出值的计算不再依赖  $h_2$  和  $h_5$ ，在反向传播时，与这两个隐藏单元相关的权重的梯度均为0。由于在训练中隐藏层神经元的丢弃是随机的，即  $h_1, \dots, h_5$  都有可能被清零，输出层的计算无法过度依赖  $h_1, \dots, h_5$  中的任一个，从而在训练模型时起到正则化的作用，并可以用来应对过拟合。在测试模型时，我们为了拿到更加确定性的结果，一般不使用丢弃法。



### 2.13.2 从零开始实现

根据丢弃法的定义，我们可以很容易地实现它。下面的 `dropout` 函数将以 `drop_prob` 的概率丢弃  $x$  中的元素。

```

1 | In [1]: 
2 |     def dropout(x, drop_prob):
3 |         x = x.float()
4 |         assert 0 <= drop_prob <= 1
5 |         keep_prob = 1 - drop_prob
6 |         # 这种情况下把全部元素都丢弃
7 |         if keep_prob == 0:
8 |             return torch.zeros_like(x)
9 |         mask = (torch.rand(x.shape) < keep_prob).float()
10 |        return mask * x / keep_prob

```

我们运行几个例子来测试一下 dropout 函数。其中丢弃概率分别为0、0.5和1。

```
1 In [2]:  
2     x = torch.arange(16).view(2, 8)  
3     dropout(x, 0)  
4 Out [2]:  
5     tensor([[ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.],  
6             [ 8.,  9., 10., 11., 12., 13., 14., 15.]])
```

```
1 In [3]:  
2     dropout(x, 0.5)  
3 Out [3]:  
4     tensor([[ 0.,  2.,  0.,  6.,  0.,  0., 12., 14.],  
5             [16.,  0., 20., 22.,  0.,  0., 28., 30.]])
```

```
1 In [4]:  
2     dropout(x, 1)  
3 Out [4]:  
4     tensor([[0., 0., 0., 0., 0., 0., 0., 0.],  
5             [0., 0., 0., 0., 0., 0., 0., 0.]])
```

## 1. 定义模型参数

实验中，我们依然使用2.5节中介绍的Fashion-MNIST数据集。我们将定义一个包含两个隐藏层的多层感知机，其中两个隐藏层的输出个数都是256。

```
1 In [5]:  
2     num_inputs, num_outputs, num_hiddens1, num_hiddens2 = 784, 10, 256, 256  
3     w1 = torch.tensor(  
4         np.random.normal(0, 0.01, size=(num_inputs, num_hiddens1)),  
5         dtype=torch.float,  
6         requires_grad=True)  
7     b1 = torch.zeros(num_hiddens1, requires_grad=True)  
8     w2 = torch.tensor(  
9         np.random.normal(0, 0.01, size=(num_hiddens1, num_hiddens2)),  
10        dtype=torch.float,  
11        requires_grad=True)  
12     b2 = torch.zeros(num_hiddens2, requires_grad=True)  
13     w3 = torch.tensor(  
14         np.random.normal(0, 0.01, size=(num_hiddens2, num_outputs)),  
15         dtype=torch.float,  
16         requires_grad=True)  
17     b3 = torch.zeros(num_outputs, requires_grad=True)  
18     params = [w1, b1, w2, b2, w3, b3]
```

## 2. 定义模型

下面定义的模型将全连接层和激活函数ReLU串起来，并对每个激活函数的输出使用丢弃法。我们可以分别设置各个层的丢弃概率。**通常的建议是把靠近输入层的丢弃概率设得小一点**。在这个实验中，我们把第一个隐藏层的丢弃概率设为0.2，把第二个隐藏层的丢弃概率设为0.5。我们可以通过参数 `is_training` 来判断运行模式为训练还是测试，并只需在训练模式下使用丢弃法。

```
1 In [6]:  
2     drop_prob1, drop_prob2 = 0.2, 0.5
```

```

3  def net(x, is_training=True):
4      x = x.view(-1, num_inputs)
5      H1 = (torch.matmul(x, w1) + b1).relu()
6      # 只在模型训练时使用丢弃法
7      if is_training:
8          # 在第一层全连接后添加丢弃层
9          H1 = dropout(H1, drop_prob1)
10     H2 = (torch.matmul(H1, w2) + b2).relu()
11     if is_training:
12         # 在第二层全连接后添加丢弃层
13         H2 = dropout(H2, drop_prob2)
14     return torch.matmul(H2, w3) + b3

```

我们在对模型评估的时候不应该进行丢弃，所以我们修改一下 d2lzh 中的 evaluate\_accuracy 函数：

```

1  def evaluate_accuracy(data_iter, net):
2      """
3          function:
4              计算多分类模型预测结果的准确率
5
6          Parameters:
7              data_iter - 样本划分为最小批的结果
8              net - 定义的网络
9
10         Returns:
11             准确率计算结果
12
13         Modify:
14             2020-12-03
15             """
16     acc_sum, n = 0.0, 0
17     for x, y in data_iter:
18         if isinstance(net, torch.nn.Module):
19             # 评估模式，这会关闭dropout
20             net.eval()
21             acc_sum += (net(x).argmax(dim=1) == y).float().sum().item()
22             # 改回训练模式
23             net.train()
24         else:
25             if 'is_training' in net.__code__.co_varnames:
26                 # 将is_training设置成False
27                 acc_sum += (net(x, is_training=False).argmax(dim=1)
28                             == y).float().sum().item()
29             else:
30                 acc_sum += (net(x).argmax(dim=1) == y).float().sum().item()
31             n += y.shape[0]
32     return acc_sum / n

```

### 3. 训练和测试模型

这部分与之前多层感知机的训练和测试类似。这里的学习率设置的很大，原因同2.9.6节。

```
1 In [7]:  
2     num_epochs, lr, batch_size = 5, 100.0, 256  
3     loss = torch.nn.CrossEntropyLoss()  
4     train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)  
5     d2l.train_ch3(net, train_iter, test_iter, loss,  
6                   num_epochs, batch_size, params, lr)  
7 Out [7]:  
8     epoch 1, loss 0.0045, train acc 0.546, test acc 0.768  
9     epoch 2, loss 0.0023, train acc 0.786, test acc 0.813  
10    epoch 3, loss 0.0019, train acc 0.821, test acc 0.818  
11    epoch 4, loss 0.0017, train acc 0.839, test acc 0.824  
12    epoch 5, loss 0.0016, train acc 0.850, test acc 0.804
```

## 2.13.3 简洁实现

在PyTorch中，我们只需要在全连接层后添加 `Dropout` 层并指定丢弃概率。在训练模型时，`Dropout` 层将以指定的丢弃概率随机丢弃上一层的输出元素；在测试模型时（即 `model.eval()` 后），`Dropout` 层并不发挥作用。

```
1 In [8]:  
2     net = nn.Sequential(  
3         d2l.FlattenLayer(),  
4         nn.Linear(num_inputs, num_hiddens1),  
5         nn.ReLU(),  
6         nn.Dropout(drop_prob1),  
7         nn.Linear(num_hiddens1, num_hiddens2),  
8         nn.ReLU(),  
9         nn.Dropout(drop_prob2),  
10        nn.Linear(num_hiddens2, num_outputs)  
11    )  
12    for param in net.parameters():  
13        nn.init.normal_(param, mean=0, std=0.01)
```

下面训练并测试模型。

```
1 In [9]:  
2     optimizer = torch.optim.SGD(net.parameters(), lr=0.5)  
3     d2l.train_ch3(net, train_iter, test_iter, loss,  
4                   num_epochs, batch_size, None, None, optimizer)  
5 Out [9]:  
6     epoch 1, loss 0.0044, train acc 0.566, test acc 0.770  
7     epoch 2, loss 0.0022, train acc 0.789, test acc 0.727  
8     epoch 3, loss 0.0019, train acc 0.821, test acc 0.831  
9     epoch 4, loss 0.0017, train acc 0.838, test acc 0.841  
10    epoch 5, loss 0.0016, train acc 0.846, test acc 0.823
```

### 小结：

- 我们可以通过使用丢弃法应对过拟合。
- 丢弃法只在训练模型时使用。

## 2.14 正向传播、反向传播和计算图

前面几节里我们使用了小批量随机梯度下降的优化算法来训练模型。在实现中，我们只提供了模型的正向传播（forward propagation）的计算，即对输入计算模型输出，然后通过 autograd 模块来调用系统自动生成的 backward 函数计算梯度。基于反向传播（back-propagation）算法的自动求梯度极大简化了深度学习模型训练算法的实现。本节我们将使用数学和计算图（computational graph）两个方式来描述正向传播和反向传播。具体来说，我们将以带  $L_2$  范数正则化的含单隐藏层的多层感知机为样例模型解释正向传播和反向传播。

## 2.14.1 正向传播

正向传播是指对神经网络沿着从输入层到输出层的顺序，依次计算并存储模型的中间变量（包括输出）。为简单起见，假设输入是一个特征为  $x \in \mathbb{R}^d$  的样本，且不考虑偏差项，那么中间变量

$$z = \mathbf{W}^{(1)}x$$

其中  $\mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$  是隐藏层的权重参数。把中间变量  $z \in \mathbb{R}^h$  输入按元素运算的激活函数  $\phi$  后，将得到向量长度为  $h$  的隐藏层变量

$$h = \phi(z)$$

隐藏层变量  $h$  也是一个中间变量。假设输出层参数只有权重  $\mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$ ，可以得到向量长度为  $q$  的输出层变量

$$o = \mathbf{W}^{(2)}h$$

假设损失函数为  $\ell$ ，且样本标签为  $y$ ，可以计算出单个数据样本的损失项

$$L = \ell(o, y)$$

根据  $L_2$  范数正则化的定义，给定超参数  $\lambda$ ，正则化项即

$$s = \frac{\lambda}{2} \left( \|\mathbf{W}^{(1)}\|_F^2 + \|\mathbf{W}^{(2)}\|_F^2 \right)$$

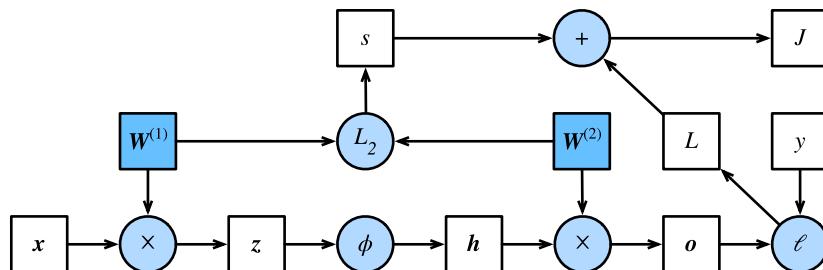
其中矩阵的 Frobenius 范数等价于将矩阵变平为向量后计算  $L_2$  范数。最终，模型在给定的数据样本上带正则化的损失为

$$J = L + s$$

我们将  $J$  称为有关给定数据样本的目标函数，并在以下的讨论中简称目标函数。

## 2.14.2 正向传播的计算图

我们通常绘制计算图（computational graph）来可视化运算符和变量在计算中的依赖关系。下图绘制了本节中样例模型正向传播的计算图，其中左下角是输入，右上角是输出。可以看到，图中箭头方向大多是向右和向上，其中方框代表变量，圆圈代表运算符，箭头表示从输入到输出之间的依赖关系。



### 2.14.3 反向传播

反向传播 (back-propagation) 指的是计算神经网络参数梯度的方法。总的来说，反向传播依据微积分中的链式法则，沿着从输出层到输入层的顺序，依次计算并存储目标函数有关神经网络各层的中间变量以及参数的梯度。对输入或输出  $X, Y, Z$  为任意形状张量的函数  $Y = f(X)$  和  $Z = g(Y)$ ，通过链式法则，我们有

$$\frac{\partial Z}{\partial X} = \text{prod} \left( \frac{\partial Z}{\partial Y}, \frac{\partial Y}{\partial X} \right)$$

其中  $\text{prod}$  运算符将根据两个输入的形状，在必要的操作（如转置和互换输入位置）后对两个输入做乘法。

回顾一下本节中样例模型，它的参数是  $W^{(1)}$  和  $W^{(2)}$ ，因此反向传播的目标是计算  $\partial J / \partial W^{(1)}$  和  $\partial J / \partial W^{(2)}$ 。我们将应用链式法则依次计算各中间变量和参数的梯度，其计算次序与前向传播中相应中间变量的计算次序恰恰相反。首先，分别计算目标函数  $J = L + s$  有关损失项  $L$  和正则项  $s$  的梯度

$$\frac{\partial J}{\partial L} = 1, \quad \frac{\partial J}{\partial s} = 1$$

其次，依据链式法则计算目标函数有关输出层变量的梯度  $\partial J / \partial o \in \mathbb{R}^q$ ：

$$\frac{\partial J}{\partial o} = \text{prod} \left( \frac{\partial J}{\partial L}, \frac{\partial L}{\partial o} \right) = \frac{\partial L}{\partial o}$$

接下来，计算正则项有关两个参数的梯度：

$$\frac{\partial s}{\partial W^{(1)}} = \lambda W^{(1)}, \quad \frac{\partial s}{\partial W^{(2)}} = \lambda W^{(2)}$$

现在，我们可以计算最靠近输出层的模型参数的梯度  $\partial J / \partial W^{(2)} \in \mathbb{R}^{q \times h}$ 。依据链式法则，得到

$$\frac{\partial J}{\partial W^{(2)}} = \text{prod} \left( \frac{\partial J}{\partial o}, \frac{\partial o}{\partial W^{(2)}} \right) + \text{prod} \left( \frac{\partial J}{\partial s}, \frac{\partial s}{\partial W^{(2)}} \right) = \frac{\partial J}{\partial o} h^\top + \lambda W^{(2)}$$

沿着输出层向隐藏层继续反向传播，隐藏层变量的梯度  $\partial J / \partial h \in \mathbb{R}^h$  可以这样计算：

$$\frac{\partial J}{\partial h} = \text{prod} \left( \frac{\partial J}{\partial o}, \frac{\partial o}{\partial h} \right) = W^{(2)\top} \frac{\partial J}{\partial o}$$

由于激活函数  $\phi$  是按元素运算的，中间变量  $z$  的梯度  $\partial J / \partial z \in \mathbb{R}^h$  的计算需要使用按元素乘法符  $\odot$ ：

$$\frac{\partial J}{\partial z} = \text{prod} \left( \frac{\partial J}{\partial h}, \frac{\partial h}{\partial z} \right) = \frac{\partial J}{\partial h} \odot \phi'(z)$$

最终，我们可以得到最靠近输入层的模型参数的梯度  $\partial J / \partial W^{(1)} \in \mathbb{R}^{h \times d}$ 。依据链式法则，得到

$$\frac{\partial J}{\partial W^{(1)}} = \text{prod} \left( \frac{\partial J}{\partial z}, \frac{\partial z}{\partial W^{(1)}} \right) + \text{prod} \left( \frac{\partial J}{\partial s}, \frac{\partial s}{\partial W^{(1)}} \right) = \frac{\partial J}{\partial z} x^\top + \lambda W^{(1)}$$

### 2.14.4 训练深度学习模型

在训练深度学习模型时，正向传播和反向传播之间相互依赖。下面我们仍然以本节中的样例模型分别阐述它们之间的依赖关系。

一方面，正向传播的计算可能依赖于模型参数的当前值，而这些模型参数是在反向传播的梯度计算后通过优化算法迭代的。例如，计算正则化项  $s = (\lambda/2) (\|W^{(1)}\|_F^2 + \|W^{(2)}\|_F^2)$  依赖模型参数  $W^{(1)}$  和  $W^{(2)}$  的当前值，而这些当前值是优化算法最近一次根据反向传播算出梯度后迭代得到的。

另一方面，反向传播的梯度计算可能依赖于各变量的当前值，而这些变量的当前值是通过正向传播计算得到的。举例来说，参数梯度 $\partial J / \partial \mathbf{W}^{(2)} = (\partial J / \partial o) \mathbf{h}^\top + \lambda \mathbf{W}^{(2)}$ 的计算需要依赖隐藏层变量的当前值 $\mathbf{h}$ 。这个当前值是通过从输入层到输出层的正向传播计算并存储得到的。

因此，在模型参数初始化完成后，我们交替地进行正向传播和反向传播，并根据反向传播计算的梯度迭代模型参数。既然我们在反向传播中使用了正向传播中计算得到的中间变量来避免重复计算，那么这个复用也导致正向传播结束后不能立即释放中间变量内存。这也是训练要比预测占用更多内存的一个重要原因。另外需要指出的是，这些中间变量的个数大体上与网络层数线性相关，每个变量的大小跟批量大小和输入个数也是线性相关的，它们是导致较深的神经网络使用较大批量训练时更容易超内存的主要原因。

**小结：**

- 正向传播沿着从输入层到输出层的顺序，依次计算并存储神经网络的中间变量。
- 反向传播沿着从输出层到输入层的顺序，依次计算并存储神经网络中间变量和参数的梯度。
- 在训练深度学习模型时，正向传播和反向传播相互依赖。

## 2.15 数值稳定性和模型初始化

理解了正向传播与反向传播以后，我们来讨论一下深度学习模型的数值稳定性问题以及模型参数的初始化方法。深度模型有关数值稳定性的典型问题是衰减 (vanishing) 和爆炸 (explosion)。

### 2.15.1 衰减和爆炸

当神经网络的层数较多时，模型的数值稳定性容易变差。假设一个层数为 $L$ 的多层感知机的第 $l$ 层 $\mathbf{H}^{(l)}$ 的权重参数为 $\mathbf{W}^{(l)}$ ，输出层 $\mathbf{H}^{(L)}$ 的权重参数为 $\mathbf{W}^{(L)}$ 。为了便于讨论，不考虑偏差参数，且设所有隐藏层的激活函数为恒等映射 (identity mapping)  $\phi(x) = x$ 。给定输入 $\mathbf{X}$ ，多层感知机的第 $l$ 层的输出 $\mathbf{H}^{(l)} = \mathbf{X}\mathbf{W}^{(1)}\mathbf{W}^{(2)} \dots \mathbf{W}^{(l)}$ 。此时，如果层数 $l$ 较大， $\mathbf{H}^{(l)}$ 的计算可能会出现衰减或爆炸。举个例子，假设输入和所有层的权重参数都是标量，如权重参数为0.2和5，多层感知机的第30层输出为输入 $\mathbf{X}$ 分别与 $0.2^{30} \approx 1 \times 10^{-21}$ （衰减）和 $5^{30} \approx 9 \times 10^{20}$ （爆炸）的乘积。类似地，当层数较多时，梯度的计算也更容易出现衰减或爆炸。

随着内容的不断深入，我们会在后面的章节进一步介绍深度学习的数值稳定性问题以及解决方法。

### 2.15.2 随机初始化模型参数

在神经网络中，通常需要随机初始化模型参数。下面我们来解释这样做的原因。

回顾2.8节（多层感知机）图中描述的多层感知机。为了方便解释，假设输出层只保留一个输出单元 $o_1$ （删去 $o_2$ 和 $o_3$ 以及指向它们的箭头），且隐藏层使用相同的激活函数。如果将每个隐藏单元的参数都初始化为相等的值，那么在正向传播时每个隐藏单元将根据相同的输入计算出相同的值，并传递至输出层。在反向传播中，每个隐藏单元的参数梯度值相等。因此，这些参数在使用基于梯度的优化算法迭代后值依然相等。之后的迭代也是如此。在这种情况下，无论隐藏单元有多少，隐藏层本质上只有1个隐藏单元在发挥作用。因此，正如在前面的实验中所做的那样，我们通常将神经网络的模型参数，特别是权重参数，进行随机初始化。

#### 1. PyTorch的默认随机初始化

随机初始化模型参数的方法有很多。在2.3节（线性回归的简洁实现）中，我们使用`torch.nn.init.normal_()`使模型`net`的权重参数采用正态分布的随机初始化方式。不过，PyTorch中`nn.Module`的模块参数都采取了较为合理的初始化策略（不同类型的layer具体采样的哪一种初始化方法的可参考[源代码](#)），因此一般不用我们考虑。

#### 2. Xavier随机初始化

还有一种比较常用的随机初始化方法叫作Xavier随机初始化。假设某全连接层的输入个数为 $a$ ，输出个数为 $b$ ，Xavier随机初始化将使该层中权重参数的每个元素都随机采样于均匀分布

$$U\left(-\sqrt{\frac{6}{a+b}}, \sqrt{\frac{6}{a+b}}\right)$$

它的设计主要考虑到，模型参数初始化后，每层输出的方差不该受该层输入个数影响，且每层梯度的方差也不该受该层输出个数影响。

### 小结：

- 深度模型有关数值稳定性的典型问题是衰减和爆炸。当神经网络的层数较多时，模型的数值稳定性容易变差。
- 我们通常需要随机初始化神经网络的模型参数，如权重参数。

## 2.16 实战Kaggle比赛：房价预测

作为深度学习基础篇章的总结，我们将对本章内容学以致用。下面，让我们动手实战一个Kaggle比赛：房价预测。本节将提供未经调优的数据的预处理、模型的设计和超参数的选择。我们希望读者通过动手操作、仔细观察实验现象、认真分析实验结果并不断调整方法，得到令自己满意的结果。

### 2.16.1 Kaggle比赛

[Kaggle](#)是一个著名的供机器学习爱好者交流的平台。下图展示了Kaggle网站的首页。为了便于提交结果，需要注册Kaggle账号。

The screenshot shows the 'All Competitions' page on the Kaggle website. The page has a sidebar with icons for search, dashboard, and user profile. The main area displays a list of active competitions:

Competition Name	Description	Prize
Riiid! Answer Correctness Prediction	Track knowledge states of 1M+ students in the wild.	\$100,000
Jane Street Market Prediction	Test your model against future real market data.	\$100,000
NFL Big Data Bowl 2021	Help evaluate defensive performance on passing plays.	\$100,000
NFL 1st and Future - Impact Detection	Detect helmet impacts in videos of NFL plays.	\$75,000
HuBMAP: Hacking the Kidney	Identify glomeruli in human kidney tissue images.	\$60,000
2020 Kaggle ML & DS Survey	The most comprehensive dataset available on the state of ML and data science.	\$30,000
Cassava Leaf Disease Classification	Identify the type of disease present on a Cassava Leaf image.	\$18,000

我们可以在房价预测比赛的网页上了解比赛信息和参赛者成绩，也可以下载数据集并提交自己的预测结果。该比赛的网页地址是 <https://www.kaggle.com/c/house-prices-advanced-regression-techniques>。

The screenshot shows the Kaggle competition page for 'House Prices: Advanced Regression Techniques'. At the top, it says 'Getting Started Prediction Competition'. The main title is 'House Prices: Advanced Regression Techniques' with the subtitle 'Predict sales prices and practice feature engineering, RFs, and gradient boosting'. Below that, it shows 'Kaggle · 4,904 teams · Ongoing'. A navigation bar at the bottom includes 'Overview', 'Data', 'Notebooks', 'Discussion', 'Leaderboard', 'Rules', and a 'Join Competition' button.

Overview

Description

Start here if...

Evaluation

You have some experience with R or Python and machine learning basics. This is a perfect competition for data science students who have completed an online course in machine learning and are looking to expand their skill set before trying a featured competition.

Tutorials

Frequently Asked Questions

Competition Description

上图展示了房价预测比赛的网页信息。比赛数据集可通过点击“Data”标签获取

## 2.16.2 读取数据集

比赛数据分为训练数据集和测试数据集。两个数据集都包括每栋房子的特征，如街道类型、建造年份、房顶类型、地下室状况等特征值。这些特征值有连续的数字、离散的标签甚至是缺失值“na”。只有训练数据集包括了每栋房子的价格，也就是标签。我们可以访问比赛网页，点击上图中的“Data”标签，并下载这些数据集。

我们将通过 pandas 库读入并处理数据。在导入本节需要的包前请确保已安装 pandas 库，否则请参考下面的代码注释。

```
1 In [1]:  
2     # 如果没有安装pandas，则反注释下面一行  
3     # ! pip install pandas  
4     import pandas as pd
```

假设解压后的数据位于 `data/` 目录，它包括两个csv文件。下面使用 pandas 读取这两个文件。

```
1 In [2]:  
2     train_data = pd.read_csv('data/train.csv')  
3     test_data = pd.read_csv('data/test.csv')
```

训练数据集包括1460个样本、80个特征和1个标签。

```
1 In [3]:  
2     train_data.shape  
3 Out [3]:  
4     (1460, 81)
```

测试数据集包括1459个样本和80个特征。我们需要将测试数据集中每个样本的标签预测出来。

```
1 In [4]:  
2     test_data.shape  
3 Out [4]:  
4     (1459, 80)
```

让我们来查看前4个样本的前4个特征、后2个特征和标签 (SalePrice) :

```
1 In [5]:  
2     train_data.iloc[0:4, [0, 1, 2, 3, -3, -2, -1]]
```

	<b>Id</b>	<b>MSSubClass</b>	<b>MSZoning</b>	<b>LotFrontage</b>	<b>SaleType</b>	<b>SaleCondition</b>	<b>SalePrice</b>
<b>0</b>	1	60	RL	65.0	WD	Normal	208500
<b>1</b>	2	20	RL	80.0	WD	Normal	181500
<b>2</b>	3	60	RL	68.0	WD	Normal	223500
<b>3</b>	4	70	RL	60.0	WD	Abnorml	140000

可以看到第一个特征是Id，它能帮助模型记住每个训练样本，但难以推广到测试样本，所以我们不使用它来训练。我们将所有的训练数据和测试数据的79个特征按样本连结。

```
1 In [6]:  
2     all_features = pd.concat((train_data.iloc[:, 1:-1],  
3                                 test_data.iloc[:, 1:]))
```

## 2.16.3 预处理数据集

我们对连续数值的特征做标准化 (standardization)：设该特征在整个数据集上的均值为 $\mu$ ，标准差为 $\sigma$ 。那么，我们可以将该特征的每个值先减去 $\mu$ 再除以 $\sigma$ 得到标准化后的每个特征值。对于缺失的特征值，我们将其替换成该特征的均值。

```
1 In [7]:  
2     numeric_features = all_features.dtypes[all_features.dtypes !=  
3                                         'object'].index  
4     all_features[numeric_features] = all_features[numeric_features].apply(  
5         lambda x: (x - x.mean()) / (x.std()))  
6     # 标准化后，每个特征的均值变为0，所以可以直接用0来替换缺失值  
7     all_features[numeric_features] = all_features[numeric_features].fillna(0)
```

接下来将离散数值转成[指示特征](#)（独热编码）。举个例子，假设特征MSZoning里面有两个不同的离散值RL和RM，那么这一步转换将去掉MSZoning特征，并新加两个特征MSZoning\_RL和MSZoning\_RM，其值为0或1。如果一个样本原来在MSZoning里的值为RL，那么有MSZoning\_RL=1且MSZoning\_RM=0。

```
1 In [8]:  
2     # dummy_na=True将缺失值也当作合法的特征值并为其创建指示特征  
3     all_features = pd.get_dummies(all_features, dummy_na=True)  
4     all_features.shape  
5 Out [8]:  
6     (2919, 331)
```

可以看到这一步转换将特征数从79增加到了331。

最后，通过values属性得到NumPy格式的数据，并转成Tensor方便后面的训练。

```

1 In [9]:
2     n_train = train_data.shape[0]
3     train_features = torch.tensor(all_features[:n_train].values,
4                                     dtype=torch.float)
5     test_features = torch.tensor(all_features[n_train:]).values,
6                                     dtype=torch.float)
7     train_labels = torch.tensor(train_data.SalePrice.values,
8                                     dtype=torch.float).view(-1, 1)

```

## 2.16.4 训练模型

我们使用一个基本的线性回归模型和平方损失函数来训练模型。

```

1 In [10]:
2     loss = torch.nn.MSELoss()
3     def get_net(feature_num):
4         net = nn.Linear(feature_num, 1)
5         for param in net.parameters():
6             nn.init.normal_(param, mean=0, std=0.01)
7         return net

```

下面定义比赛用来评价模型的对数均方根误差。给定预测值 $\hat{y}_1, \dots, \hat{y}_n$ 和对应的真实标签 $y_1, \dots, y_n$ , 它的定义为

$$\sqrt{\frac{1}{n} \sum_{i=1}^n (\log(y_i) - \log(\hat{y}_i))^2}$$

对数均方根误差的实现如下。

```

1 In [11]:
2     def log_rmse(net, features, labels):
3         with torch.no_grad():
4             # 将小于1的值设成1, 使得取对数时数值更稳定
5             clipped_preds = torch.max(net(features), torch.tensor(1.0))
6             rmse = torch.sqrt(loss(clipped_preds.log(), labels.log()))
7             return rmse.item()

```

下面的训练函数跟本章中前几节的不同在于使用了Adam优化算法。相对之前使用的小批量随机梯度下降, 它对学习率相对不那么敏感。我们将在之后的“优化算法”一章里详细介绍它。

```

1 In [12]:
2     def train(net, train_features, train_labels, test_features, test_labels,
3               num_epochs, learning_rate, weight_decay, batch_size):
4         train_ls, test_ls = [], []
5         dataset = torch.utils.data.TensorDataset(train_features,
6                                               train_labels)
6         train_iter = torch.utils.data.DataLoader(dataset, batch_size,
7                                               shuffle=True)
8
9         # 这里使用了Adam优化器
10        optimizer = torch.optim.Adam(params=net.parameters(),
11                                      lr=learning_rate,
12                                      weight_decay=weight_decay)
13
14        net = net.float()
15        for epoch in range(num_epochs):
16            for X, y in train_iter:

```

```
16         l = loss(net(x.float()), y.float())
17         optimizer.zero_grad()
18         l.backward()
19         optimizer.step()
20     train_ls.append(log_rmse(net, train_features, train_labels))
21     if test_labels is not None:
22         test_ls.append(log_rmse(net, test_features, test_labels))
23 return train_ls, test_ls
```

## 2.16.5 k折交叉验证

我们在2.11节（模型选择、欠拟合和过拟合）中介绍了 $K$ 折交叉验证。它将被用来选择模型设计并调节超参数。下面实现了一个函数，它返回第*i*折交叉验证时所需要的训练和验证数据。`slice` 的用法见[博客](#)。

```
In [13]:  
1     def get_k_fold_data(k, i, x, y):  
2         assert k > 1  
3         fold_size = x.shape[0] // k  
4         x_train, y_train = None, None  
5         for j in range(k):  
6             idx = slice(j * fold_size, (j+1) * fold_size)  
7             x_part, y_part = x[idx, :], y[idx]  
8             if j==i:  
9                 x_valid, y_valid = x_part, y_part  
10            elif x_train is None:  
11                x_train, y_train = x_part, y_part  
12            else:  
13                x_train = torch.cat((x_train, x_part), dim=0)  
14                y_train = torch.cat((y_train, y_part), dim=0)  
15  
16    return x_train, y_train, x_valid, y_valid
```

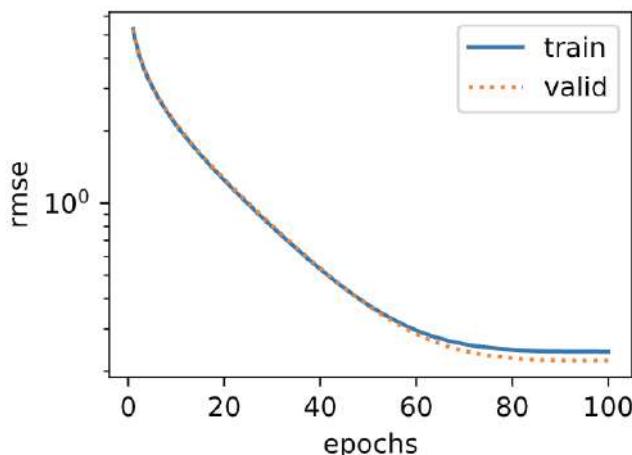
在  $K$  折交叉验证中我们训练  $K$  次并返回训练和验证的平均误差。

```
1 In [14]:  
2     def k_fold(k, x_train, y_train, num_epochs,  
3                 learning_rate, weight_decay, batch_size):  
4         train_l_sum, valid_l_sum = 0, 0  
5         for i in range(k):  
6             data = get_k_fold_data(k, i, x_train, y_train)  
7             net = get_net(x_train.shape[1])  
8             train_ls, valid_ls = train(net, *data, num_epochs,  
9                               learning_rate,  
10                              weight_decay, batch_size)  
11             train_l_sum += train_ls[-1]  
12             valid_l_sum += valid_ls[-1]  
13             if i == 0:  
14                 d2l.semilogy(range(1, num_epochs+1), train_ls, 'epochs',  
15                               'rmse', range(1, num_epochs+1),  
16                               valid_ls, ['train', 'valid'])  
17                 print('fold %d, train rmse %f, valid rmse %f' %  
18                               (i, train_ls[-1], valid_ls[-1]))  
19             return train_l_sum / k, valid_l_sum / k
```

## 2.16.6 模型选择

我们使用一组未经调优的超参数并计算交叉验证误差。可以改动这些超参数来尽可能减小平均测试误差。

```
1 In [15]:  
2     k, num_epochs, lr, weight_decay, batch_size = 5, 100, 5, 0, 64  
3     train_l, valid_l = k_fold(k, train_features, train_labels,  
4                               num_epochs, lr, weight_decay, batch_size)  
5     print('%d-fold validation: avg train rmse %f, avg valid rmse %f' %  
6           (k, train_l, valid_l))  
7 Out [15]:  
8     fold 0, train rmse 0.170489, valid rmse 0.157037  
9     fold 1, train rmse 0.162431, valid rmse 0.190838  
10    fold 2, train rmse 0.164008, valid rmse 0.168377  
11    fold 3, train rmse 0.168448, valid rmse 0.154738  
12    fold 4, train rmse 0.163813, valid rmse 0.183044  
13    5-fold validation: avg train rmse 0.165838, avg valid rmse 0.170807
```



有时候你会发现一组参数的训练误差可以达到很低，但是在 $K$ 折交叉验证上的误差可能反而较高。这种现象很可能是由过拟合造成的。因此，当训练误差降低时，我们要观察 $K$ 折交叉验证上的误差是否也相应降低。

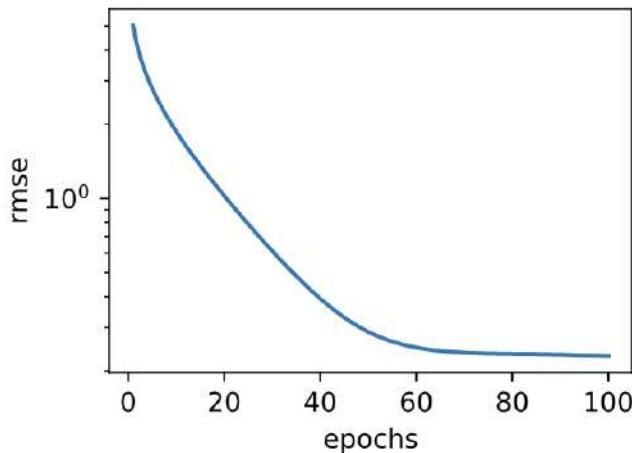
## 2.16.7 预测并在Kaggle提交结果

下面定义预测函数。在预测之前，我们会使用完整的训练数据集来重新训练模型，并将预测结果存成提交所需要的格式。

```
1 In [16]:  
2     def train_and_pred(train_features, test_features, train_labels,  
3                          test_data, num_epochs, lr, weight_decay,  
4                          batch_size):  
5         net = get_net(train_features.shape[1])  
6         train_ls, _ = train(net, train_features, train_labels, None, None,  
7                           num_epochs, lr, weight_decay, batch_size)  
8         d2l.semilogy(range(1, num_epochs+1), train_ls, 'epochs', 'rmse')  
9         print('train rmse %f' % train_ls[-1])  
10        preds = net(test_features).detach().numpy()  
11        test_data['SalePrice'] = pd.Series(preds.reshape(1, -1)[0])  
12        submission = pd.concat([test_data['Id'], test_data['SalePrice']],  
13                               axis=1)  
14        submission.to_csv('submission.csv', index=False)
```

设计好模型并调好超参数之后，下一步就是对测试数据集上的房屋样本做价格预测。如果我们得到与交叉验证时差不多的训练误差，那么这个结果很可能是理想的，可以在Kaggle上提交结果。

```
1 In [17]:  
2     train_and_pred(train_features, test_features,  
3                     train_labels, test_data, num_epochs,  
4                     lr, weight_decay, batch_size)  
5 Out [17]:  
6     train rmse 0.162403
```



上述代码执行完之后会生成一个submission.csv文件。这个文件是符合Kaggle比赛要求的提交格式的。这时，我们可以在Kaggle上提交我们预测得出的结果，并且查看与测试数据集上真实房价（标签）的误差。具体来说有以下几个步骤：登录Kaggle网站，访问房价预测比赛网页，并点击右侧“Submit Predictions”或“Late Submission”按钮；然后，点击页面下方“Upload Submission File”图标所在的虚线框选择需要提交的预测结果文件；最后，点击页面最下方的“Make Submission”按钮就可以查看结果了，如下图所示。

Step 1  
Upload submission file

Step 2  
Describe submission

B I | % “ <> | ⌂ ⌂ H ⌂ | ⌂ C | M Styling with Markdown supported

File Format  
Your submission should be in CSV format.  
You can upload this in a zip/gz/rar/7z archive, if you prefer.

Number of Predictions  
We expect the solution file to have 1469 prediction rows. This file should have a header row. Please see sample submission file on the [data page](#).

Briefly describe your submission.

Make Submission

## 小结：

- 通常需要对真实数据做预处理。
- 可以使用 $K$ 折交叉验证来选择模型并调节超参数。

## 2.17 本章附录

### ☆ torch.LongTensor() 的用法

`torch.LongTensor()` 将创建一个张量，其元素类型为 `64-bit integer (signed)`。使用方法同 `torch.tensor()`。

### ☆ torch.index\_select() 的用法

`torch.index_select(input, dim, index, out=None) → Tensor`。该函数会返回一个新的张量，此张量会沿着 `dim` 方向取 `input` 中符合 `index` 索引的数值，其中 `index` 的数值类型为 `LongTensor`。

参数	类型	说明	默认值
index	LongTensor	包含索引的一维张量	

```
1 | x = torch.randn(3, 4)
2 | x
3 | >>> tensor([[ 0.4343, -1.6746, -0.9908, -0.8505],
4 |               [ 0.0448,  1.1774, -0.0102, -0.4750],
5 |               [ 0.1494, -0.0502, -0.4513, -1.8651]])
6 | indices = torch.LongTensor([0, 2])
7 | torch.index_select(x, 0, indices)
8 | >>> tensor([[ 0.4343, -1.6746, -0.9908, -0.8505],
9 |               [ 0.1494, -0.0502, -0.4513, -1.8651]])
10 | torch.index_select(x, 1, indices)
11 | >>> tensor([[ 0.4343, -0.9908],
12 |               [ 0.0448, -0.0102],
13 |               [ 0.1494, -0.4513]])
```

### ☆ torch.mm() 的用法

`torch.mm(mat1, mat2, out=None) → Tensor`。对矩阵 `mat1` 和矩阵 `mat2` 执行矩阵乘法。如果 `mat1` 的维度为  $(n \times m)$ , `mat2` 的维度为  $(m \times p)$ , 那么 `out` 的维度为  $(n \times p)$ 。注意：这个函数不会进行广播，函数 `torch.matmul()` 能够进行广播机制的矩阵乘法。

```
1 | mat1 = torch.randn(2, 3)
2 | mat2 = torch.randn(3, 3)
3 | torch.mm(mat1, mat2)
4 | >>> tensor([[-0.3330, -0.2014, -0.0380],
5 |               [ 0.4439,  0.5871, -1.6853]])
```

### ☆ torch.utils.data.TensorDataset(\*tensors) 的用法

`TensorDataset` 可以用来对 `tensor` 进行打包，就好像 `python` 中的 `zip` 功能。该类通过每一个 `tensor` 的第一个维度进行索引。因此，该类中的 `tensor` 第一个维度必须相等。

```
1 | from torch.utils.data import TensorDataset
2 | a = torch.tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9],
3 |                   [1, 2, 3], [4, 5, 6], [7, 8, 9],
4 |                   [1, 2, 3], [4, 5, 6], [7, 8, 9],
5 |                   [1, 2, 3], [4, 5, 6], [7, 8, 9]])
6 | b = torch.tensor([44, 55, 66, 44, 55, 66,
7 |                   44, 55, 66, 44, 55, 66])
8 | train_ids = TensorDataset(a, b)
```

```

9 # 切片输出
10 print(train_ids[0:2])
11 >>> (tensor([[1, 2, 3],
12             [4, 5, 6]]), tensor([44, 55]))
13 # 循环取数据
14 for x_train, y_label in train_ids:
15     print(x_train, y_label)
16 >>>
17 tensor([1, 2, 3]) tensor(44)
18 tensor([4, 5, 6]) tensor(55)
19 tensor([7, 8, 9]) tensor(66)
20 tensor([1, 2, 3]) tensor(44)
21 tensor([4, 5, 6]) tensor(55)
22 tensor([7, 8, 9]) tensor(66)
23 tensor([1, 2, 3]) tensor(44)
24 tensor([4, 5, 6]) tensor(55)
25 tensor([7, 8, 9]) tensor(66)
26 tensor([1, 2, 3]) tensor(44)
27 tensor([4, 5, 6]) tensor(55)
28 tensor([7, 8, 9]) tensor(66)

```

## ☆ `torch.utils.data.DataLoader()` 的用法

`torch.utils.data.DataLoader(dataset, batch_size=1, shuffle=False, sampler=None, batch_sampler=None, num_workers=0, collate_fn=<function default_collate>, pin_memory=False, drop_last=False, timeout=0, worker_init_fn=None)`。此函数为数据加载器。它会利用单进程或多进程的方法结合采样算法对数据集进行采样，并将采样结果利用迭代器作为返回。参数说明见[博客](#)。

```

1 from torch.utils.data import DataLoader
2 # train_ids 沿用上例
3 train_loader = DataLoader(dataset=train_ids, batch_size=4, shuffle=True)
4 # 注意enumerate返回值有两个，一个是序号，一个是数据（包含训练数据和标签）
5 for i, data in enumerate(train_loader, 1):
6     x_data, label = data
7     print(' batch:{0} x_data:{1}  label: {2}'.format(i, x_data, label))
8 >>>
9 batch:1 x_data:tensor([[4, 5, 6,
10                         [1, 2, 3],
11                         [1, 2, 3],
12                         [1, 2, 3]])  label: tensor([55, 44, 44, 44])
13 batch:2 x_data:tensor([[7, 8, 9],
14                         [4, 5, 6],
15                         [1, 2, 3],
16                         [7, 8, 9]])  label: tensor([66, 55, 44, 66])
17 batch:3 x_data:tensor([[7, 8, 9],
18                         [4, 5, 6],
19                         [7, 8, 9],
20                         [4, 5, 6]])  label: tensor([66, 55, 66, 55])

```

## ☆ `Linear` 的用法

`torch.nn.Linear(in_features, out_features, bias=True)`。对输入数据做线性变换： $y = xA^T + b$ 。其中 `in_features` 表示输入特征的维度，`out_features` 表示输出特征的维度，`bias` 当取 `False` 时，模型不会添加偏置，默认值为 `True`。这一个层能够训练得到的参数有：`Linear.weight`——模型的权重，其维度为 `(out_features, in_features)`，其默认初始化方式是  $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ ，其中

$k = \frac{1}{\text{in\_features}}$ 。`Linear.bias`——模型的偏置，其维度为(`out_features`)，若`bias`取值为`True`，那么偏置的初始化方式为 $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ ，其中 $k = \frac{1}{\text{in\_features}}$ 。

```
1 import torch.nn as nn
2 import torch
3 m = nn.Linear(20, 30)
4 input = torch.randn(128, 20)
5 output = m(input)
6 print(output.size())
7 >>> torch.Size([128, 30])
```

## ☆ `torch.unsqueeze()` 的用法

`torch.unsqueeze(input, dim, out=None) → Tensor`。该函数会对输入张量在指定位置添加一维，除了指定位置之外，其余位置元素和原始张量保持一致。`dim`的取值范围是`[input.dim() - 1, input.dim() + 1]`。当`dim`为负数时，此时的插入位置为`dim = dim + input.dim() + 1`。

```
1 x = torch.tensor([1, 2, 3, 4])
2 x.size()
3 >>> torch.Size([4])
4 y = torch.unsqueeze(x, 0)
5 y
6 >>> tensor([[1, 2, 3, 4]])
7 y.size()
8 >>> torch.Size([1, 4])
9 z = torch.unsqueeze(x, 1)
10 z
11 >>> tensor([[1,
12             [2],
13             [3],
14             [4]]])
15 z.size()
16 >>> torch.Size([4, 1])
```

## ☆ `torch.nn.init.normal_()` 的用法

`torch.nn.init.normal_(tensor, mean=0.0, std=1.0)`。利用取自正态分布 $\mathcal{N}(mean, std)$ 的值填充张量中的元素。

```
1 w = torch.empty(3, 5)
2 nn.init.normal_(w)
3 >>> tensor([[-1.7806, -0.0610,  0.4522,  0.7378, -0.3459],
4             [ 0.9821, -1.2052, -1.3663, -0.4654,  0.1913],
5             [-1.7945,  0.2947, -0.1777, -1.6039, -0.7060]])
```

## ☆ `torch.nn.init.constant_()` 的用法

`torch.nn.init.constant_(tensor, val)`。利用值`val`填充张量。

```
1 w = torch.empty(3, 5)
2 nn.init.constant_(w, 0.3)
3 >>> tensor([[0.3000, 0.3000, 0.3000, 0.3000, 0.3000],
4             [0.3000, 0.3000, 0.3000, 0.3000, 0.3000],
5             [0.3000, 0.3000, 0.3000, 0.3000, 0.3000]])
```

## ☆ torch.nn.MSELoss() 的用法

求两个张量之间的均方差或和方差，当参数 `reduction` 为 `mean` 时计算均方差MSE（默认值），当其为 `sum` 时计算和方差SSE。

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (X_{obs,i} - X_{model,i})^2$$
$$\text{SSE} = \sum_{i=1}^n (X_{obs,i} - X_{model,i})^2$$

```
1 loss = nn.MSELoss()
2 target = torch.Tensor([10, 6])
3 res = torch.Tensor([5, 3])
4 print(loss(res, target))
5 >>> tensor(17.)
6 loss = nn.MSELoss(reduction='sum')
7 print(loss(res, target))
8 >>> tensor(34.)
```

## ☆ gather 的用法

`gather`是`scatter`的逆过程，从一个张量收集数据，到另一个张量，我们先看一个例子有个直观感受。

```
1 x = torch.tensor([[1, 2], [3, 4]])
2 torch.gather(input=x, dim=1, index=torch.tensor([[0, 0], [1, 0]]))
3 >>> tensor([[1, 1],
4             [4, 3]])
```

可以猜测到收集过程，根据`index`和`dim`将`x`中的数据挑选出来，放置到`y`中，满足下面的公式：`y[i, j] = x[i, index[i, j]]`，因此有`y[0, 0] = x[0, index[0, 0]] = x[0, 0] = 1`，`y[1, 0] = x[1, index[1, 0]] = x[1, 1] = 4`。特别注意一下，`index`的类型必须是`LongTensor`类型的。

## ☆ torch.max() 的用法

`torch.max(input, other, out=None) → Tensor`。对 `input` 中的每个元素与 `other` 中的对应元素比较大小，返回二值之中的较大值，`input` 和 `other` 的维度不必一致，但是它们必须满足广播机制。

$$\text{out}_i = \max(\text{tensor}_i, \text{other}_i)$$

```
1 a = torch.randn(4)
2 a
3 >>> tensor([-0.8041,  0.1231,  0.2058,  3.2035])
4 b = torch.randn(4)
5 b
6 >>> tensor([-0.8540,  0.5659,  1.3647,  1.0258])
7 torch.max(a, b)
8 >>> tensor([-0.8041,  0.5659,  1.3647,  3.2035])
```

## ☆ torch.pow() 的用法

`torch.pow(input, exponent, out=None) → Tensor`。对 `input` 中的每个元素计算其 `exponent` 次幂。`exponent` 可以是一个 `float` 数值，也可以是一个与 `input` 同维的张量。当 `exponent` 是一个标量时，计算方式如下：

$$\text{out}_i = x_i^{\text{exponent}}$$

当 `exponent` 是张量时，计算方式如下：

$$\text{out}_i = x_i^{\text{exponent}_i}$$

```
1 a = torch.randn(4)
2 a
3 >>> tensor([ 0.3070,  0.5302, -0.8638,  0.0226])
4 torch.pow(a, 2)
5 >>> tensor([9.4246e-02, 2.8113e-01, 7.4621e-01, 5.1005e-04])
6 exp = torch.arange(1., 5.)
7 a = torch.arange(1., 5.)
8 a
9 >>> tensor([1., 2., 3., 4.])
10 exp
11 >>> tensor([1., 2., 3., 4.])
12 torch.pow(a, exp)
13 >>> tensor([ 1.,   4.,  27., 256.])
```

本章代码详见GitHub链接[wzy6642](#)。

## 第3章 深度学习计算

第2章介绍了包括多层感知机在内的简单深度学习模型的原理和实现。本章我们将简要概括深度学习计算的各个重要组成部分，如模型构造、参数的访问和初始化等，自定义层，读取、存储和使用GPU。通过本章的学习，我们将能够深入了解模型实现和计算的各个细节，并为在之后章节实现更复杂模型打下坚实的基础。

### 3.1 模型构造

让我们回顾一下在2.10节（多层感知机的简洁实现）中含单隐藏层的多层感知机的实现方法。我们首先构造 `sequential` 实例，然后依次添加两个全连接层。其中第一层的输出大小为256，即隐藏层单元个数是256；第二层的输出大小为10，即输出层单元个数是10。我们在上一章的其他节中也使用了 `sequential` 类构造模型。这里我们介绍另外一种基于 `Module` 类的模型构造方法：它让模型构造更加灵活。

#### 3.1.1 继承Module类来构造模型

`Module` 类是 `nn` 模块里提供的一个模型构造类，是所有神经网络模块的基类，我们可以继承它来定义我们想要的模型。下面继承 `Module` 类构造本节开头提到的多层感知机。这里定义的 `MLP` 类重载了 `Module` 类的 `__init__` 函数和 `forward` 函数。它们分别用于创建模型参数和定义前向计算。前向计算也即正向传播。

```
1 In [1]:
2     import torch
3     from torch import nn
4     class MLP(nn.Module):
5         # 声明带有模型参数的层，这里声明了两个全连接层
6         def __init__(self, **kwargs):
7             # 调用MLP父类Module的构造函数来进行初始化
8             # 这样在构造实例时还可以指定其他函数参数
9             # 如3.2节将介绍的模型参数params
10            super(MLP, self).__init__(**kwargs)
```

```

11         self.hidden = nn.Linear(784, 256)
12         self.act = nn.ReLU()
13         self.output = nn.Linear(256, 10)
14     # 定义模型的前向计算，即如何根据输入x计算返回所需要的模型输出
15     def forward(self, x):
16         a = self.act(self.hidden(x))
17         return self.output(a)

```

以上的 `MLP` 类中无须定义反向传播函数。系统将通过自动求梯度而自动生成反向传播所需的 `backward` 函数。

我们可以实例化 `MLP` 类得到模型变量 `net`。下面的代码初始化 `net` 并传入输入数据 `x` 做一次前向计算。其中，`net(x)` 会调用 `MLP` 继承自 `Module` 类的 `__call__` 函数，这个函数将调用 `MLP` 类定义的 `forward` 函数来完成前向计算。

```

1 In [2]:
2     x = torch.randn(2, 784)
3     net = MLP()
4     print(net)
5     net(x)
6 Out [2]:
7     MLP(
8         (hidden): Linear(in_features=784, out_features=256, bias=True)
9         (act): ReLU()
10        (output): Linear(in_features=256, out_features=10, bias=True)
11    )
12    tensor([[-0.2132, -0.0707,  0.5831, -0.1266, -0.1045,
13            0.4470, -0.0326, -0.3240,  0.0637, -0.0768],
14           [ 0.0022,   0.1763,   0.1568, -0.0503,   0.3620,
15             0.3018,   0.2725, -0.0484,  0.0322, -0.4748]],
16           grad_fn=<AddmmBackward>)

```

注意，这里并没有将 `Module` 类命名为 `Layer`（层）或者 `Model`（模型）之类的名字，这是因为该类是一个可供自由组建的部件。它的子类既可以是一个层（如PyTorch提供的 `Linear` 类），又可以是一个模型（如这里定义的 `MLP` 类），或者是模型的一个部分。我们下面通过两个例子来展示它的灵活性。

### 3.1.2 Sequential类继承自Module类

我们刚刚提到，`Module` 类是一个通用的部件。事实上，PyTorch还实现了继承自 `Module` 的可以方便构建模型的类：如 `sequential`、`ModuleList` 和 `ModuleDict` 等等。当模型的前向计算为简单串联各个层的计算时，`Sequential` 类可以通过更加简单的方式定义模型。这正是 `Sequential` 类的目的：它可以接收一个子模块的有序字典（`OrderedDict`）或者一系列子模块作为参数来逐一添加 `Module` 的实例，而模型的前向计算就是将这些实例按添加的顺序逐一计算。

下面我们实现一个与 `Sequential` 类有相同功能的 `MySequential` 类。这或许可以帮助读者更加清晰地理解 `Sequential` 类的工作机制。

```

1 In [3]:
2     class MySequential(nn.Module):
3         from collections import OrderedDict
4         def __init__(self, *args):
5             super(MySequential, self).__init__()
6             # 如果传入的是一个OrderedDict
7             if len(args)==1 and isinstance(args[0], OrderedDict):
8                 for key, module in args[0].items():

```

```

9          # add_module方法会将module添加
10         # 进self._modules(一个orderedDict)
11         self.add_module(key, module)
12     # 传入的是一些Module
13     else:
14         for idx, module in enumerate(args):
15             self.add_module(str(idx), module)
16     def forward(self, input):
17         # self._modules返回一个orderedDict, 保证会按照成员添加时的顺序遍历成员
18         for module in self._modules.values():
19             input = module(input)
20     return input

```

我们用 `MySequential` 类来实现前面描述的 `MLP` 类，并使用随机初始化的模型做一次前向计算。

```

1 In [4]:
2     net = MySequential(
3         nn.Linear(784, 256),
4         nn.ReLU(),
5         nn.Linear(256, 10)
6     )
7     print(net)
8     net(x)
9 Out [4]:
10    MySequential(
11        (0): Linear(in_features=784, out_features=256, bias=True)
12        (1): ReLU()
13        (2): Linear(in_features=256, out_features=10, bias=True)
14    )
15    tensor([[ 0.2152,  0.1581,  0.0771, -0.1201, -0.3036,
16              0.3782, -0.2487, -0.2290, -0.1366,  0.1810],
17             [-0.0970,  0.1910,  0.0158,  0.0511, -0.3612,
18              0.3619, -0.1819,  0.0847,  0.1825,  0.1605]], grad_fn=<AddmmBackward>)

```

可以观察到这里 `MySequential` 类的使用跟2.10节（多层感知机的简洁实现）中 `sequential` 类的使用没什么区别。

### 3.1.3 构造复杂的模型

虽然上面介绍的这些类可以使模型构造更加简单，且不需要定义 `forward` 函数，但直接继承 `Module` 类可以极大地拓展模型构造的灵活性。下面我们构造一个稍微复杂点的网络 `FancyMLP`。在这个网络中，我们通过 `rand` 函数创建训练中不被迭代的参数，即常数参数。在前向计算中，除了使用创建的常数参数外，我们还使用 `Tensor` 的函数和Python的控制流，并多次调用相同的层。

```

1 In [5]:
2     class FancyMLP(nn.Module):
3         def __init__(self, **kwargs):
4             super(FancyMLP, self).__init__(**kwargs)
5             # 不可训练参数(常数参数)
6             self.rand_weight = torch.rand((20, 20), requires_grad=False)
7             self.linear = nn.Linear(20, 20)
8         def forward(self, x):
9             x = self.linear(x)
10            # 使用创建的常数参数, 以及nn.functional中的relu函数和mm函数

```

```

11         x = nn.functional.relu(torch.mm(x, self.rand_weight.data) + 1)
12     # 复用全连接层，等价于两个全连接层共享参数
13     x = self.linear(x)
14     # 控制流，这里我们需要调用item函数来返回标量进行比较
15     while x.norm().item() > 1:
16         x /= 2
17     if x.norm().item() < 0.8:
18         x *= 10
19     return x.sum()

```

在这个 FancyMLP 模型中，我们使用了常数权重 `rand_weight`（注意它不是可训练模型参数）、做了矩阵乘法操作（`torch.mm`）并重复使用了相同的 `Linear` 层。下面我们来测试该模型的前向计算。

```

1 In [6]:
2     x = torch.rand(2, 20)
3     net = FancyMLP()
4     print(net)
5     net(x)
6 Out [6]:
7     FancyMLP(
8         (linear): Linear(in_features=20, out_features=20, bias=True)
9     )
10    tensor(0.3003, grad_fn=<SumBackward0>)

```

因为 `FancyMLP` 和 `Sequential` 类都是 `Module` 类的子类，所以我们可以嵌套调用它们。

```

1 In [7]:
2     class NestMLP(nn.Module):
3         def __init__(self, **kwargs):
4             super(NestMLP, self).__init__(**kwargs)
5             self.net = nn.Sequential(nn.Linear(40, 30), nn.ReLU())
6         def forward(self, x):
7             return self.net(x)
8     net = nn.Sequential(NestMLP(), nn.Linear(30, 20), FancyMLP())
9     x = torch.rand(2, 40)
10    print(net)
11    net(x)
12 Out [7]:
13    Sequential(
14        (0): NestMLP(
15            (net): Sequential(
16                (0): Linear(in_features=40, out_features=30, bias=True)
17                (1): ReLU()
18            )
19        )
20        (1): Linear(in_features=30, out_features=20, bias=True)
21        (2): FancyMLP(
22            (linear): Linear(in_features=20, out_features=20, bias=True)
23        )
24    )
25    tensor(0.2910, grad_fn=<SumBackward0>)

```

## 小结：

- 可以通过继承 `Module` 类来构造模型。
- `Sequential`、`ModuleList`、`ModuleDict` 类都继承自 `Module` 类。

- 与 `Sequential` 不同，`ModuleList` 和 `ModuleDict` 并没有定义一个完整的网络，它们只是将不同的模块存放在一起，需要自己定义 `forward` 函数。
- 虽然 `Sequential` 等类可以使模型构造更加简单，但直接继承 `Module` 类可以极大地拓展模型构造的灵活性。

## 3.2 模型参数的访问、初始化和共享

在2.3节（线性回归的简洁实现）中，我们通过 `init` 模块来初始化模型的参数。我们也介绍了访问模型参数的简单方法。本节将深入讲解如何访问和初始化模型参数，以及如何在多个层之间共享同一份模型参数。

我们先定义一个与上一节中相同的含单隐藏层的多层感知机。我们依然使用默认方式初始化它的参数，并做一次前向计算。与之前不同的是，在这里我们从 `nn` 中导入了 `init` 模块，它包含了多种模型初始化方法。

```

1 In [1]:
2     net = nn.Sequential(nn.Linear(20, 256),
3                         nn.ReLU(),
4                         nn.Linear(256, 10))
5 # PyTorch默认已经初始化
6 print(net)
7 X = torch.rand(2, 20)
8 # 前向计算
9 Y = net(X)
10 Out [1]:
11 Sequential(
12     (0): Linear(in_features=20, out_features=256, bias=True)
13     (1): ReLU()
14     (2): Linear(in_features=256, out_features=10, bias=True)
15 )

```

### 3.2.1 访问模型参数

对于使用 `Sequential` 类构造的神经网络，我们可以通过方括号 `[]` 来访问网络的任一层。回忆一下上一节中提到的 `Sequential` 类与 `Module` 类的继承关系。对于 `Sequential` 实例中含模型参数的层，我们可以通过 `Module` 类的 `parameters()` 或者 `named_parameters()` 方法来访问所有参数（以迭代器的形式返回）。索引0表示隐藏层为 `Sequential` 实例最先添加的层。

```

1 In [2]:
2     for name, param in net[0].named_parameters():
3         print(name, param.size(), type(param))
4 Out [2]:
5     weight torch.Size([256, 20]) <class 'torch.nn.parameter.Parameter'>
6     bias torch.Size([256]) <class 'torch.nn.parameter.Parameter'>

```

因为这里是单层的所以没有了层数索引的前缀。另外返回的 `param` 的类型为 `torch.nn.parameter.Parameter`，其实这是 `Tensor` 的子类，和 `Tensor` 不同的是如果一个 `Tensor` 是 `Parameter`，那么它会自动被添加到模型的参数列表里。不难发现权重的形状为 `(256, 20)`，为了访问特定参数，我们既可以通过特定名字访问字典里的元素，也可以直接使用 `data` 方法。下面两种代码是等价的，但通常后者的代码可读性更好。

```
1 In [3]:  
2     weight_0 = list(net[0].parameters())[0]  
3     type(weight_0.data[0][0]), weight_0.data.size()  
4 Out [3]:  
5     (torch.Tensor, torch.size([256, 20]))  
6 In [4]:  
7     param = dict(net[0].named_parameters())  
8     type(param['weight']), param['weight'].size()  
9 Out [4]:  
10    (torch.nn.parameter.Parameter, torch.size([256, 20]))
```

因为 `Parameter` 是 `Tensor`，即 `Tensor` 拥有的属性它都有，它包含参数和梯度的数值，可以分别通过 `data` 方法和 `grad` 方法来访问。因为我们随机初始化了权重，所以权重参数是一个由随机数组成的形状为 `(256, 20)` 的 `Tensor`。

```
1 In [5]:  
2     weight_0.data  
3 Out [5]:  
4     tensor([[ 0.2155, -0.0453, -0.1902, ..., 0.1393, -0.0315, 0.0522],  
5             [ 0.1597, -0.1855, 0.1597, ..., 0.1955, 0.0225, -0.2209],  
6             [ 0.1106, -0.0571, 0.2116, ..., 0.2041, 0.0074, 0.0948],  
7             ...,  
8             [ 0.1125, 0.0480, -0.0999, ..., 0.0338, -0.0411, -0.0576],  
9             [ 0.0360, 0.2011, 0.0974, ..., 0.0782, 0.0011, 0.0547],  
10            [-0.1841, -0.0220, -0.0778, ..., 0.0269, -0.1520, 0.1373]])
```

权重梯度的形状和权重的形状一样。因为我们还没有进行反向传播计算，所以梯度的值为 `None`。

```
1 In [6]:  
2     print(weight_0.grad)  
3 Out [6]:  
4     None
```

类似地，我们可以访问其他层的参数，如输出层的偏差值。

```
1 In [7]:  
2     bias_1 = list(net[2].parameters())[1]  
3     bias_1.data  
4 Out [7]:  
5     tensor([-0.0276, -0.0006, 0.0290, -0.0006, -0.0475,  
6             -0.0322, -0.0338, 0.0268, 0.0351, 0.0163])
```

最后，我们可以使用 `print()` 函数获取网络结构。

```
1 In [8]:  
2     print(net)  
3 Out [8]:  
4     Sequential(  
5         (0): Linear(in_features=20, out_features=256, bias=True)  
6         (1): ReLU()  
7         (2): Linear(in_features=256, out_features=10, bias=True)  
8     )
```

### 3.2.2 初始化模型参数

我们在2.15节（数值稳定性和模型初始化）中提到了PyTorch中`nn.Module`的模块参数都采取了较为合理的初始化策略（不同类型的layer具体采样的哪一种初始化方法的可参考[源代码](#)）。但我们经常需要使用其他方法来初始化权重。PyTorch的`init`模块里提供了多种预设的初始化方法。在下面的例子中，我们将权重参数初始化成均值为0、标准差为0.01的正态分布随机数，并依然将偏差参数清零。

```
1 In [9]:  
2     from torch.nn import init  
3     for name, param in net.named_parameters():  
4         if 'weight' in name:  
5             init.normal_(param, mean=0, std=0.01)  
6             print(name, param.data)  
7 Out [9]:  
8     0.weight  
9     tensor([[-0.0107, ..., 0.0128, -0.0077],  
10            [-0.0064, ..., -0.0040, 0.0045],  
11            ...,  
12            [-0.0098, ..., -0.0052, -0.0151]])  
13     2.weight  
14     tensor([[ 6.7065e-03, ..., 1.0689e-03],  
15            [-3.9546e-03, ..., -6.4671e-03],  
16            ...,  
17            [ 6.8653e-03, ..., 1.0306e-02]])
```

下面使用常数来初始化权重参数。

```
1 In [10]:  
2     for name, param in net.named_parameters():  
3         if 'bias' in name:  
4             init.constant_(param, val=0)  
5             print(name, param.data)  
6 Out [10]:  
7     0.bias tensor([0., 0., ..., 0.])  
8     2.bias tensor([0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

我们也可以对某个特定参数进行初始化，下例中我们对第一层的权重使用Xavier随机初始化，对第二层的权重使用常数42初始化。

```
1 In [11]:  
2     def xavier(m):  
3         if type(m)==nn.Linear:  
4             torch.nn.init.xavier_uniform_(m.weight)  
5     def init_42(m):  
6         if type(m)==nn.Linear:  
7             torch.nn.init.constant_(m.weight, 42)  
8     net[0].apply(xavier)  
9     net[2].apply(init_42)  
10    print(net[0].weight.data[0])  
11    print(net[2].weight.data)  
12 Out [11]:  
13    tensor([ 0.1153,  0.0312,  0.1251, -0.0104,  0.1391,  
14           -0.1438, -0.1064,  0.1205, -0.0367,  0.1021,  
15           -0.0378,  0.1016,  0.0179, -0.0893, -0.1073,  
16           -0.0574,  0.1020,  0.1451,  0.0722,  0.1322])
```

```
17     tensor([[42.,  ..., 42., 42., 42.],  
18             ...,  
19             [42.,  ..., 42., 42., 42.],  
20             [42.,  ..., 42., 42., 42.]])
```

### 3.2.3 自定义初始化方法

有时候我们需要的初始化方法并没有在 `init` 模块中提供。这时，可以实现一个初始化方法，从而能够像使用其他初始化方法那样使用它。在这之前我们先来看看PyTorch是怎么实现这些初始化方法的，例如 `torch.nn.init.normal_`:

```
1 def normal_(tensor, mean=0, std=1):  
2     with torch.no_grad():  
3         return tensor.normal_(mean, std)
```

可以看到这就是一个改变 `Tensor` 值的函数，而且这个过程是不记录梯度的。类似的我们来实现一个自定义的初始化方法。在下面的例子里，我们令权重有一半概率初始化为0，有另一半概率初始化为  $[-10, -5]$  和  $[5, 10]$  两个区间里均匀分布的随机数。

```
1 In [12]:  
2     def init_weight_(tensor):  
3         with torch.no_grad():  
4             tensor.uniform_(-10, 10)  
5             tensor *= (tensor.abs() >= 5).float()  
6         for name, param in net.named_parameters():  
7             if 'weight' in name:  
8                 init_weight_(param)  
9                 print(name, param.data)  
10    Out [12]:  
11        0.weight  
12        tensor([[ 8.9102,  ..., -8.8676,  9.9697, -0.0000],  
13                  [ 8.0217,  ...,  0.0000, -9.8473, -7.5096],  
14                  ...,  
15                  [-7.2460,  ...,  0.0000,  5.0712,  9.8235]])  
16        2.weight  
17        tensor([[-0.0000,  ...,  9.1993, -0.0000, -0.0000],  
18                  [-0.0000,  ..., -7.4760,  9.8242,  0.0000],  
19                  ...,  
20                  [ 0.0000,  ..., -8.9268, -8.4430, -0.0000]])
```

此外，我们还可以通过改变这些参数的 `data` 来改写模型参数值同时不会影响梯度。例如，在下例中我们将隐藏层偏置参数在现有的基础上加1:

```
1 In [13]:  
2     for name, param in net.named_parameters():  
3         if 'bias' in name:  
4             param.data += 1  
5             print(name, param.data)  
6     Out [13]:  
7        0.bias tensor([1., 1., ..., 1., 1.])  
8        2.bias tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

### 3.2.4 共享模型参数

在有些情况下，我们希望在多个层之间共享模型参数。3.1.3节提到了如何共享模型参数：Module类的forward函数里多次调用同一个层。此外，如果我们传入Sequential的模块是同一个Module实例的话参数也是共享的，下面来看一个例子：

```
1 In [14]:  
2     linear = nn.Linear(1, 1, bias=False)  
3     net = nn.Sequential(linear, linear)  
4     print(net)  
5     for name, param in net.named_parameters():  
6         init.constant_(param, val=3)  
7         print(name, param.data)  
8 Out [14]:  
9     Sequential(  
10         (0): Linear(in_features=1, out_features=1, bias=False)  
11         (1): Linear(in_features=1, out_features=1, bias=False)  
12     )  
13     0.weight tensor([[3.]])
```

在内存中，这两个线性层其实一个对象：

```
1 In [15]:  
2     print(id(net[0]) == id(net[1]))  
3     print(id(net[0].weight) == id(net[1].weight))  
4 Out [15]:  
5     True  
6     True
```

因为模型参数里包含了梯度，所以在反向传播计算时，这些共享的参数的梯度是累加的：

```
1 In [16]:  
2     x = torch.ones(1, 1)  
3     y = net(x).sum()  
4     print(y)  
5     y.backward()  
6     # 单次梯度是3，两次所以就是6  
7     print(net[0].weight.grad)  
8 Out [16]:  
9     tensor(9., grad_fn=<SumBackward0>)  
10    tensor([[6.]])
```

**小结：**

- 有多种方法来访问、初始化和共享模型参数。
- 可以自定义初始化方法。

## 3.3 自定义层

深度学习的一个魅力在于神经网络中各式各样的层，例如全连接层和后面章节中将要介绍的卷积层、池化层与循环层。虽然PyTorch提供了大量常用的层，但有时候我们依然希望自定义层。本节将介绍如何使用Module来自定义层，从而可以被重复调用。

### 3.3.1 不含模型参数的自定义层

我们先介绍如何定义一个不含模型参数的自定义层。事实上，这和3.1节（模型构造）中介绍的使用 `Module` 类构造模型类似。下面的 `CenteredLayer` 类通过继承 `Module` 类自定义了一个将输入减掉均值后输出的层，并将层的计算定义在了 `forward` 函数里。这个层里不含模型参数。

```
1 In [1]:  
2     class CenteredLayer(nn.Module):  
3         def __init__(self, **kwargs):  
4             super(CenteredLayer, self).__init__(**kwargs)  
5         def forward(self, x):  
6             return x - x.mean()
```

我们可以实例化这个层，然后做前向计算。

```
1 In [2]:  
2     layer = CenteredLayer()  
3     layer(torch.tensor([1, 2, 3, 4, 5], dtype=torch.float))  
4 Out [2]:  
5     tensor([-2., -1.,  0.,  1.,  2.])
```

我们也可以用它来构造更复杂的模型。

```
1 In [3]:  
2     net = nn.Sequential(nn.Linear(8, 128), CenteredLayer())
```

下面打印自定义层各个输出的均值。因为均值是浮点数，所以它的值是一个很接近0的数。

```
1 In [4]:  
2     y = net(torch.rand(4, 8))  
3     y.mean().item()  
4 Out [4]:  
5     -2.0954757928848267e-09
```

### 3.3.2 含模型参数的自定义层

我们还可以自定义含模型参数的自定义层。其中的模型参数可以通过训练学出。

在3.2节（模型参数的访问、初始化和共享）中介绍了 `Parameter` 类其实是 `Tensor` 的子类，如果一个 `Tensor` 是 `Parameter`，那么它会自动被添加到模型的参数列表里。所以在自定义含模型参数的层时，我们应该将参数定义成 `Parameter`，除了像3.2.1节那样直接定义成 `Parameter` 类外，还可以使用 `ParameterList` 和 `ParameterDict` 分别定义参数的列表和字典。

`ParameterList` 接收一个 `Parameter` 实例的列表作为输入然后得到一个参数列表，使用的时候可以用索引来访问某个参数，另外也可以使用 `append` 和 `extend` 在列表后面新增参数。

```
1 In [5]:  
2     class MyDense(nn.Module):  
3         def __init__(self):  
4             super(MyDense, self).__init__()  
5             self.params = nn.ParameterList(  
6                 [nn.Parameter(torch.randn(4, 4)) for i in range(3)])  
7             self.params.append(nn.Parameter(torch.randn(4, 1)))  
8         def forward(self, x):
```

```

9         for i in range(len(self.params)):
10            x = torch.mm(x, self.params[i])
11        return x
12    net = MyDense()
13    print(net)
14 Out [5]:
15    MyDense(
16      (params): ParameterList(
17        (0): Parameter containing: [torch.FloatTensor of size 4x4]
18        (1): Parameter containing: [torch.FloatTensor of size 4x4]
19        (2): Parameter containing: [torch.FloatTensor of size 4x4]
20        (3): Parameter containing: [torch.FloatTensor of size 4x1]
21      )
22    )

```

而 `ParameterDict` 接收一个 `Parameter` 实例的字典作为输入然后得到一个参数字典，然后可以按照字典的规则使用了。例如使用 `update()` 新增参数，使用 `keys()` 返回所有键值，使用 `items()` 返回所有键值对等等，可参考[官方文档](#)。

```

1 In [6]:
2   class MyDictDense(nn.Module):
3     def __init__(self):
4       super(MyDictDense, self).__init__()
5       self.params = nn.ParameterDict({
6         'linear1': nn.Parameter(torch.randn(4, 4)),
7         'linear2': nn.Parameter(torch.randn(4, 1))
8       })
9       # 新增
10      self.params.update(
11        {'linear3': nn.Parameter(torch.randn(4, 2))})
12    def forward(self, x, choice='linear1'):
13      return torch.mm(x, self.params[choice])
14    net = MyDictDense()
15    print(net)
16 Out [6]:
17    MyDictDense(
18      (params): ParameterDict(
19        (linear1): Parameter containing: [torch.FloatTensor of size 4x4]
20        (linear2): Parameter containing: [torch.FloatTensor of size 4x1]
21        (linear3): Parameter containing: [torch.FloatTensor of size 4x2]
22      )
23    )

```

这样就可以根据传入的键值来进行不同的前向传播：

```

1 In [7]:
2   x = torch.ones(1, 4)
3   print(net(x, 'linear1'))
4   print(net(x, 'linear2'))
5   print(net(x, 'linear3'))
6 Out [7]:
7   tensor([[2.1038, 3.1801, 1.4371, 0.5740]], grad_fn=<MmBackward>)
8   tensor([[1.9707]], grad_fn=<MmBackward>)
9   tensor([[2.9602, 2.0829]], grad_fn=<MmBackward>)

```

我们也可以使用自定义层构造模型。它和PyTorch的其他层在使用上很类似。

```
1 In [8]:  
2     net = nn.Sequential(  
3         MyDictDense(),  
4         MyDense()  
5     )  
6     print(net)  
7     print(net(x))  
8 Out [8]:  
9     Sequential(  
10         (0): MyDictDense(  
11             (params): ParameterDict(  
12                 (linear1): Parameter containing: [torch.FloatTensor of size 4x4]  
13                 (linear2): Parameter containing: [torch.FloatTensor of size 4x1]  
14                 (linear3): Parameter containing: [torch.FloatTensor of size 4x2]  
15             )  
16         )  
17         (1): MyDense(  
18             (params): ParameterList(  
19                 (0): Parameter containing: [torch.FloatTensor of size 4x4]  
20                 (1): Parameter containing: [torch.FloatTensor of size 4x4]  
21                 (2): Parameter containing: [torch.FloatTensor of size 4x4]  
22                 (3): Parameter containing: [torch.FloatTensor of size 4x1]  
23             )  
24         )  
25     )  
26     tensor([[2.2536]], grad_fn=<MmBackward>)
```

## 小结:

- 可以通过 `Module` 类自定义神经网络中的层，从而可以被重复调用。

## 3.4 读取和存储

到目前为止，我们介绍了如何处理数据以及如何构建、训练和测试深度学习模型。然而在实际中，我们有时需要把训练好的模型部署到很多不同的设备。在这种情况下，我们可以把内存中训练好的模型参数存储在硬盘上供后续读取使用。

### 3.4.1 读写Tensor

我们可以直接使用 `save` 函数和 `load` 函数分别存储和读取 `Tensor`。下面的例子创建了 `Tensor` 变量 `x`，并将其存在文件名同为 `x.pt` 的文件里。

```
1 In [1]:  
2     x = torch.ones(3)  
3     torch.save(x, 'x.pt')
```

然后我们将数据从存储的文件读回内存。

```
1 In [2]:  
2     x2 = torch.load('x.pt')  
3     x2  
4 Out [2]:  
5     tensor([1., 1., 1.])
```

我们还可以存储一个 `Tensor` 列表并读回内存。

```
1 In [3]:  
2     y = torch.zeros(4)  
3     torch.save([x, y], 'xy.pt')  
4     xy_list = torch.load('xy.pt')  
5     xy_list  
6 Out [3]:  
7     [tensor([1., 1., 1.]), tensor([0., 0., 0., 0.])]
```

我们甚至可以存储并读取一个从字符串映射到 `Tensor` 的字典。

```
1 In [4]:  
2     torch.save({'x': x, 'y': y}, 'xy_dict.pt')  
3     xy = torch.load('xy_dict.pt')  
4     xy  
5 Out [4]:  
6     {'x': tensor([1., 1., 1.]), 'y': tensor([0., 0., 0., 0.])}
```

### 3.4.2 读写模型的参数

除 `Tensor` 以外，我们还可以读写 `Module` 模型的参数。PyTorch提供了 `save` 函数和 `load` 函数来读写模型。为了演示方便，我们先创建一个多层感知机，并将其初始化。

```
1 In [5]:  
2     class MLP(nn.Module):  
3         def __init__(self):  
4             super(MLP, self).__init__()  
5             self.hidden = nn.Linear(3, 2)  
6             self.act = nn.ReLU()  
7             self.output = nn.Linear(2, 1)  
8         def forward(self, x):  
9             a = self.act(self.hidden(x))  
10            return self.output(a)  
11 net = MLP()
```

下面把该模型存成文件，文件名为model.bin。

```
1 In [6]:  
2     torch.save(net, 'model.bin')
```

接下来，我们从文件加载整个模型。

```
1 In [7]:  
2     net2 = torch.load('model.bin')
```

因为这 `net` 和 `net2` 都有同样的模型参数，那么对同一个输入 `x` 的计算结果将会是一样的。我们来验证一下。

```
1 In [8]:  
2     Y2 = net2(x)  
3     Y = net(x)  
4     Y2 == Y  
5 Out [8]:  
6     tensor([1], dtype=torch.uint8)
```

## 小结:

- 通过 `save` 函数和 `load` 函数可以很方便地读写 `Tensor`。
- 通过 `save` 函数和 `load` 函数可以很方便地读写模型。

## 3.5 GPU计算

到目前为止，我们一直在使用CPU计算。对复杂的神经网络和大规模的数据来说，使用CPU来计算可能不够高效。在本节中，我们将介绍如何使用单块NVIDIA GPU来计算。所以需要确保已经安装好了PyTorch GPU版本。准备工作都完成后，下面就可以通过 `watch -n 10 nvidia-smi` 命令来查看显卡信息了。

```
Every 10.0s: nvidia-smi

Tue Dec 8 22:37:55 2020
+-----+
| NVIDIA-SMI 450.57      Driver Version: 450.57      CUDA Version: 11.0 |
+-----+
| GPU  Name        Persistence-M | Bus-Id      Disp.A  | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap | Memory-Usage | GPU-Util  Compute M. |
|          %          %          /    /          /   /   |          %          MIG M. |
|-----+
|  0  TITAN V           Off     00000000:02:00.0 Off |                  N/A | |
| 31%   45C    P8    27W / 250W | 311MiB / 12065MiB | 0%      Default |
|                               |                  N/A |
+-----+
|  1  TITAN V           Off     00000000:02:00.0 Off |                  N/A | |
| 40%   56C    P8    31W / 250W | 309MiB / 12066MiB | 0%      Default |
|                               |                  N/A |
+-----+
+-----+
| Processes:                               GPU Memory |
| GPU  GI CI PID  Type  Process name        Usage  |
| ID  ID   |                               |
|-----+
|  0  N/A N/A 30609    C  ...ares/anaconda3/bin/python  307MiB |
|  1  N/A N/A 24274    C  ...ares/anaconda3/bin/python  305MiB |
+-----+
```

### 3.5.1 计算设备

PyTorch可以指定用来存储和计算的设备，如使用内存的CPU或者使用显存的GPU。在默认情况下，PyTorch会将数据创建在内存，然后利用CPU来计算。在PyTorch中，`torch.device('cpu')` 表示所有的物理CPU和内存。这意味着，PyTorch的计算会尽量使用所有的CPU核。但 `torch.cuda.device('cuda')` 只代表一块GPU和相应的显存。如果有多块GPU，我们用 `torch.cuda.device('cuda:i')` 来表示第  $i$  块GPU及相应的显存 ( $i$  从0开始) 且 `torch.cuda.device('cuda')` 和 `torch.cuda.device('cuda:0')` 等价。

```
1 In [1]:
2     torch.device('cpu'), torch.cuda.device('cuda'),
3     torch.cuda.device('cuda:1')
4 Out [1]:
5     (device(type='cpu'),
6      <torch.cuda.device at 0x7f2cba460e10>,
7      <torch.cuda.device at 0x7f2d792a1710>)
```

### 3.5.2 Tensor的GPU计算

默认情况下，`Tensor` 会被存在内存上。因此，之前我们每次打印 `Tensor.device` 的时候都会看到 CPU这个标识。

```
1 In [2]:  
2     x = torch.tensor([1, 2, 3])  
3     x.device  
4 Out [2]:  
5     device(type='cpu')
```

## 1. GPU上的存储

我们有多种方法将Tensor存储在显存上。例如，我们可以在创建Tensor的时候通过 `device` 参数指定存储设备。下面我们将Tensor变量a创建在 `gpu(0)` 上。注意，在打印a时，设备信息变成了 `cuda:0`。创建在显存上的Tensor只消耗同一块显卡的显存。我们可以通过 `watch -n 10 nvidia-smi` 命令查看显存的使用情况。通常，我们需要确保不创建超过显存上限的数据。

```
1 In [3]:  
2     a = torch.tensor([1, 2, 3], device=torch.device('cuda'))  
3     a  
4 Out [3]:  
5     tensor([1, 2, 3], device='cuda:0')
```

假设至少有2块GPU，下面代码将会在 `gpu(1)` 上创建随机数组。

```
1 In [4]:  
2     B = torch.rand(2, 3, device=torch.device('cuda:1'))  
3     B  
4 Out [4]:  
5     tensor([[0.0251, 0.4416, 0.1046],  
6             [0.5686, 0.7297, 0.8858]], device='cuda:1')
```

除了在创建时指定，我们也可以通过 `cuda` 函数在设备之间传输数据。下面我们将内存上的Tensor变量x复制到 `gpu(1)` 上。

```
1 In [5]:  
2     Z = x.cuda(1)  
3     print(x)  
4     print(Z)  
5 Out [5]:  
6     tensor([1, 2, 3])  
7     tensor([1, 2, 3], device='cuda:1')
```

需要注意的是，存储在不同位置中的数据是不可以直接进行计算的。即存放在CPU上的数据不可以直接与存放在GPU上的数据进行运算，位于不同GPU上的数据也是不能直接进行计算的。

```
1 In [6]:  
2     Z + x  
3 Out [6]:  
4     RuntimeError: expected backend CUDA and dtype Long  
5         but got backend CPU and dtype Long
```

## 2. GPU上的计算

PyTorch的计算会在数据的 `device` 属性所指定的设备上执行。为了使用GPU计算，我们只需要实现将数据存到显存上。计算结果会自动保存在同一块显卡的显存上。

```
1 In [7]:  
2     torch.exp((z+2).float())*x.float().cuda(1)  
3 Out [7]:  
4     tensor([ 20.0855, 109.1963, 445.2395], device='cuda:1')
```

注意，PyTorch要求计算的所有输入数据都在内存或同一块显卡的显存上。这样设计的原因是CPU和不同的GPU之间的数据交互通常比较耗时。因此，PyTorch希望用户确切地指明计算的输入数据都在内存或同一块显卡的显存上。例如，如果将内存上的Tensor变量`x`和显存上的Tensor变量`y`做运算，会出现错误信息。当我们打印Tensor或将Tensor转换成NumPy格式时，如果数据不在内存里，PyTorch会将它先复制到内存，从而造成额外的传输开销。

### 3.5.3 模型的GPU计算

同Tensor类似，PyTorch模型也可以通过`.cuda`转换到GPU上。我们可以通过检查模型的参数的`device`属性来查看存放模型的设备。

```
1 In [8]:  
2     net = nn.Linear(3, 1)  
3     net.cuda()  
4     net.weight.data.device  
5 Out [8]:  
6     device(type='cuda', index=0)
```

当输入是显存上的Tensor时，PyTorch会在同一块显卡的显存上计算结果。

```
1 In [9]:  
2     net(z.view(1, 3).float().cuda())  
3 Out [9]:  
4     tensor([[1.4004]], device='cuda:0', grad_fn=<AddmmBackward>)
```

下面我们确认一下模型参数存储在同一块显卡的显存上。

```
1 In [10]:  
2     net.weight.data  
3 Out [10]:  
4     tensor([[ 0.2655, -0.1501,  0.5458]], device='cuda:0')
```

## 小结：

- PyTorch可以指定用来存储和计算的设备，如使用内存的CPU或者使用显存的GPU。在默认情况下，PyTorch会将数据创建在内存，然后利用CPU来计算。
- PyTorch要求计算的所有输入数据都在内存或同一块显卡的显存上。

## 3.6 本章附录

### ☆ OrderedDict的用法

通常来讲，python中的字典是无序的，因为它是按照hash来存储的。但是python中有个模块`collections`，里面自带了一个子类`orderedDict`，实现了对字典对象中元素的排序。`orderdDict`是对字典类型的补充，它记住了字典元素添加的顺序。

```
1 import collections
2 a = collections.OrderedDict()
3 a['a'] = 'A'
4 a['1'] = '1'
5 for k, v in a.items():
6     print(k, v)
>>> a A
8     1 1
```

## ☆ add\_module 的用法

add\_module(name, module)。为当前的模型添加子模型，被添加的模型可以通过 name 属性获取。

```
1 import torch.nn as nn
2 net = nn.Sequential(nn.Linear(1, 1))
3 net.add_module('hidden', nn.Linear(1, 1))
4 print(net)
5 >>>
6 Sequential(
7     (0): Linear(in_features=1, out_features=1, bias=True)
8     (hidden): Linear(in_features=1, out_features=1, bias=True)
9 )
10 net.hidden.bias.data
11 >>> tensor([-0.1389])
12 net._modules
13 >>>
14 OrderedDict([('0', Linear(in_features=1, out_features=1, bias=True)),
15             ('hidden', Linear(in_features=1, out_features=1, bias=True))])
```

## ☆ torch.rand() 的用法

torch.randn(\*sizes, out=None, dtype=None, layout=torch.strided, device=None, requires\_grad=False) → Tensor。返回一个形状为 size 的张量，其值取于 [0, 1] 之间的均匀分布。

```
1 torch.rand(4)
2 >>> tensor([0.9735, 0.9183, 0.5477, 0.6056])
```

## ☆ model.named\_parameters() 的用法

迭代打印 model.named\_parameters() 将会打印每一次迭代元素的名字和参数。

```
1 for name, param in model.named_parameters():
2     print(name, param)
```

## ☆ model.parameters() 的用法

迭代打印 model.parameters() 将会打印每一次迭代元素的参数而不会打印名字。

```
1 for param in model.parameters():
2     print(param)
```

# 第4章 卷积神经网络

本章将介绍卷积神经网络。它是近年来深度学习能在计算机视觉领域取得突破性成果的基石。它也逐渐在被其他诸如自然语言处理、推荐系统和语音识别等领域广泛使用。我们将先描述卷积神经网络中卷积层和池化层的工作原理，并解释填充、步幅、输入通道和输出通道的含义。在掌握了这些基础知识以后，我们将探究数个具有代表性的深度卷积神经网络的设计思路。这些模型包括最早提出的 AlexNet，以及后来的使用重复元素的网络（VGG）、网络中的网络（NiN）、含并行连结的网络（GoogLeNet）、残差网络（ResNet）和稠密连接网络（DenseNet）。它们中有不少在过去几年的 ImageNet 比赛（一个著名的计算机视觉竞赛）中大放异彩。虽然深度模型看上去只是具有很多层的神经网络，然而获得有效的深度模型不容易。有幸的是，本章阐述的**批量归一化**和**残差网络**为训练和设计深度模型提供了两类重要思路。

## 4.1 二维卷积层

卷积神经网络（convolutional neural network）是含有卷积层（convolutional layer）的神经网络。本章中介绍的卷积神经网络均使用最常见的二维卷积层。它有高和宽两个空间维度，常用来处理图像数据。本节中，我们将介绍简单形式的二维卷积层的工作原理。

### 4.1.1 二维互相关运算

虽然卷积层得名于卷积（convolution）运算，但我们通常在卷积层中使用更加直观的互相关（cross-correlation）运算。在二维卷积层中，一个二维输入数组和一个二维核（kernel）数组通过互相关运算输出一个二维数组。我们用一个具体例子来解释二维互相关运算的含义。如下图所示，输入是一个高和宽均为3的二维数组。我们将该数组的形状记为 $3 \times 3$ 或 $(3, 3)$ 。核数组的高和宽分别为2。该数组在卷积计算中又称**卷积核**或过滤器（filter）。卷积核窗口（又称卷积窗口）的形状取决于卷积核的高和宽，即 $2 \times 2$ 。下图中的阴影部分为第一个输出元素及其计算所使用的输入和核数组元素：

$$0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19.$$

输入	核	输出																	
<table border="1"><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>3</td><td>4</td><td>5</td></tr><tr><td>6</td><td>7</td><td>8</td></tr></table>	0	1	2	3	4	5	6	7	8	<table border="1"><tr><td>0</td><td>1</td></tr><tr><td>2</td><td>3</td></tr></table>	0	1	2	3	<table border="1"><tr><td>19</td><td>25</td></tr><tr><td>37</td><td>43</td></tr></table>	19	25	37	43
0	1	2																	
3	4	5																	
6	7	8																	
0	1																		
2	3																		
19	25																		
37	43																		

在二维互相关运算中，卷积窗口从输入数组的最左上方开始，按从左往右、从上往下的顺序，依次在输入数组上滑动。当卷积窗口滑动到某一位置时，窗口中的输入子数组与核数组按元素相乘并求和，得到输出数组中相应位置的元素。上图中的输出数组高和宽分别为2，其中的4个元素由二维互相关运算得出：

$$\begin{aligned}0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 &= 19 \\1 \times 0 + 2 \times 1 + 4 \times 2 + 5 \times 3 &= 25 \\3 \times 0 + 4 \times 1 + 6 \times 2 + 7 \times 3 &= 37 \\4 \times 0 + 5 \times 1 + 7 \times 2 + 8 \times 3 &= 43\end{aligned}$$

下面我们将上述过程实现在 `corr2d` 函数里。它接受输入数组 `x` 与核数组 `K`，并输出数组 `Y`。

```

1 In [1]:
2     import torch
3     from torch import nn
4     def corr2d(X, K):
5         # 行、列值
6         h, w = K.shape
7         # 卷积结果的存放位置
8         Y = torch.zeros((X.shape[0]-h+1, X.shape[1]-w+1))
9         for i in range(Y.shape[0]):
10            for j in range(Y.shape[1]):
11                Y[i, j] = (X[i:i+h, j:j+w] * K).sum()
12
13     return Y

```

我们可以构造上图中的输入数组 `X`、核数组 `K` 来验证二维互相关运算的输出。

```

1 In [2]:
2     X = torch.tensor([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
3     K = torch.tensor([[0, 1], [2, 3]])
4     corr2d(X, K)
5 Out [2]:
6     tensor([[19., 25.],
7             [37., 43.]])

```

## 4.1.2 二维卷积层

二维卷积层将输入和卷积核做互相关运算，并加上一个标量偏差来得到输出。卷积层的模型参数包括了卷积核和标量偏差。在训练模型的时候，通常我们先对卷积核随机初始化，然后不断迭代卷积核和偏差。

下面基于 `corr2d` 函数来实现一个自定义的二维卷积层。在构造函数 `__init__` 里我们声明 `weight` 和 `bias` 这两个模型参数。前向计算函数 `forward` 则是直接调用 `corr2d` 函数再加上偏差。

```

1 In [3]:
2     class Conv2D(nn.Module):
3         def __init__(self, kernel_size):
4             super(Conv2D, self).__init__()
5             self.weight = nn.Parameter(torch.randn(kernel_size))
6             self.bias = nn.Parameter(torch.randn(1))
7         def forward(self, x):
8             return corr2d(x, self.weight) + self.bias

```

卷积窗口形状为  $p \times q$  的卷积层称为  $p \times q$  卷积层。同样， $p \times q$  卷积或  $p \times q$  卷积核说明卷积核的高和宽分别为  $p$ （行）和  $q$ （列）。

## 4.1.3 图像中物体边缘检测

下面我们来看一个卷积层的简单应用：检测图像中物体的边缘，即找到像素变化的位置。首先我们构造一张  $6 \times 8$  的图像（即高和宽分别为 6 像素和 8 像素的图像）。它中间 4 列为黑（0），其余为白（1）。

```
1 In [4]:  
2     X = torch.ones(6, 8)  
3     X[:, 2:6] = 0  
4     X  
5 Out [4]:  
6     tensor([[1., 1., 0., 0., 0., 1., 1.],  
7             [1., 1., 0., 0., 0., 1., 1.],  
8             [1., 1., 0., 0., 0., 1., 1.],  
9             [1., 1., 0., 0., 0., 1., 1.],  
10            [1., 1., 0., 0., 0., 1., 1.],  
11            [1., 1., 0., 0., 0., 1., 1.]])
```

然后我们构造一个高和宽分别为1和2的卷积核  $K$ 。当它与输入做互相关运算时，如果横向相邻元素相同，输出为0；否则输出为非0。

```
1 In [5]:  
2     K = torch.tensor([[1, -1]])
```

下面将输入  $X$  和我们设计的卷积核  $K$  做互相关运算。可以看出，我们将从白到黑的边缘和从黑到白的边缘分别检测成了1和-1。其余部分的输出全是0。

```
1 In [6]:  
2     Y = corr2d(X, K.float())  
3     Y  
4 Out [6]:  
5     tensor([[ 0.,  1.,  0.,  0., -1.,  0.],  
6             [ 0.,  1.,  0.,  0., -1.,  0.],  
7             [ 0.,  1.,  0.,  0., -1.,  0.],  
8             [ 0.,  1.,  0.,  0., -1.,  0.],  
9             [ 0.,  1.,  0.,  0., -1.,  0.],  
10            [ 0.,  1.,  0.,  0., -1.,  0.]])
```

由此，我们可以看出，**卷积层可通过重复使用卷积核有效地表征局部空间**。

#### 4.1.4 通过数据学习核数组

最后我们来看一个例子，它使用物体边缘检测中的输入数据  $X$  和输出数据  $Y$  来学习我们构造的核数组  $K$ 。我们首先构造一个卷积层，其卷积核将被初始化成随机数组。接下来在每一次迭代中，我们使用平方误差来比较  $Y$  和卷积层的输出，然后计算梯度来更新权重。

```
1 In [7]:  
2     conv2d = Conv2D(kernel_size=(1, 2))  
3     step = 30  
4     lr = 0.01  
5     for i in range(step):  
6         Y_hat = conv2d(X)  
7         l = ((Y_hat - Y)**2).sum()  
8         l.backward()  
9         # 梯度下降  
10        conv2d.weight.data -= lr * conv2d.weight.grad  
11        conv2d.bias.data -= lr * conv2d.bias.grad  
12        # 梯度清零  
13        conv2d.weight.grad.fill_(0)  
14        conv2d.bias.grad.fill_(0)
```

```

15     if (i+1)%5 == 0:
16         print('Step %d, loss %.3f' % (i+1, l.item()))
17
18 Out [7]:
19     Step 5, loss 1.473
20     Step 10, loss 0.282
21     Step 15, loss 0.064
22     Step 20, loss 0.016
23     Step 25, loss 0.004
24     Step 30, loss 0.001

```

可以看到，30次迭代后误差已经降到了一个比较小的值。现在来看一下学习到的卷积核的参数。

```

1 In [8]:
2     print('weight: ', conv2d.weight.data)
3     print('bias: ', conv2d.bias.data)
4
5 Out [8]:
6     weight: tensor([[ 0.9918, -0.9908]])
7     bias: tensor([-0.0006])

```

可以看到，学到的卷积核的权重参数与我们之前定义的核数组 `k` 较接近，而偏置参数接近0。

## 4.1.5 互相关运算和卷积运算

实际上，卷积运算与互相关运算类似。**为了得到卷积运算的输出，我们只需将核数组左右翻转并上下翻转，再与输入数组做互相关运算。**可见，卷积运算和互相关运算虽然类似，但如果它们使用相同的核数组，对于同一个输入，输出往往并不相同。

那么，你也许会好奇卷积层为何能使用互相关运算替代卷积运算。其实，在深度学习中核数组都是学出来的：卷积层无论使用互相关运算或卷积运算都不影响模型预测时的输出。为了解释这一点，假设卷积层使用互相关运算学出上图中的核数组。设其他条件不变，使用卷积运算学出的核数组即上图中的核数组按上下、左右翻转。也就是说，上图中的输入与学出的已翻转的核数组再做卷积运算时，依然得到上图中的输出。为了与大多数深度学习文献一致，如无特别说明，本书中提到的卷积运算均指互相关运算。

## 4.1.6 特征图和感受野

二维卷积层输出的二维数组可以看作是输入在空间维度（宽和高）上某一级的表征，也叫**特征图**（feature map）。**影响元素 $x$ 的前向计算的所有可能输入区域（可能大于输入的实际尺寸）叫做 $x$ 的感受野（receptive field）。**以上图为例，输入中阴影部分的四个元素是输出中阴影部分元素的感受野。我们将上图中形状为 $2 \times 2$ 的输出记为 $Y$ ，并考虑一个更深的卷积神经网络：将 $Y$ 与另一个形状为 $2 \times 2$ 的核数组做互相关运算，输出单个元素 $z$ 。那么， $z$ 在 $Y$ 上的感受野包括 $Y$ 的全部四个元素，在输入上的感受野包括其中全部9个元素。可见，**我们可以通过更深的卷积神经网络使特征图中单个元素的感受野变得更加广阔，从而捕捉输入上更大尺寸的特征。**

我们常使用“元素”一词来描述数组或矩阵中的成员。在神经网络的术语中，这些元素也可称为“单元”。当含义明确时，本书不对这两个术语做严格区分。

**小结：**

- 二维卷积层的核心计算是二维互相关运算。在最简单的形式下，它对二维输入数据和卷积核做互相关运算然后加上偏差。
- 我们可以设计卷积核来检测图像中的边缘。
- 我们可以通过数据来学习卷积核。

## 4.2 填充和步幅

在上一节的例子里，我们使用高和宽为3的输入与高和宽为2的卷积核得到高和宽为2的输出。一般来说，假设输入形状是 $n_h \times n_w$ ，卷积核窗口形状是 $k_h \times k_w$ ，那么输出形状将会是

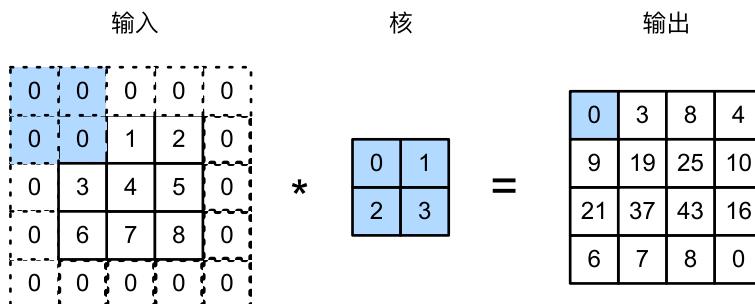
$$(n_h - k_h + 1) \times (n_w - k_w + 1)$$

所以卷积层的输出形状由输入形状和卷积核窗口形状决定。本节我们将介绍卷积层的两个超参数，即填充和步幅。它们可以对给定形状的输入和卷积核改变输出形状。

## 4.2.1 填充

填充 (padding) 是指在输入高和宽的两侧填充元素（通常是0元素）。下图里我们在原输入高和宽的两侧分别添加了值为0的元素，使得输入高和宽从3变成了5，并导致输出高和宽由2增加到4。下图中的阴影部分为第一个输出元素及其计算所使用的输入和核数组元素：

$$0 \times 0 + 0 \times 1 + 0 \times 2 + 0 \times 3 = 0.$$



一般来说，如果在高的两侧一共填充 $p_h$ 行，在宽的两侧一共填充 $p_w$ 列，那么输出形状将会是

$$(n_h - k_h + p_h + 1) \times (n_w - k_w + p_w + 1)$$

也就是说，输出的高和宽会分别增加 $p_h$ 和 $p_w$ 。

在很多情况下，我们会设置 $p_h = k_h - 1$ 和 $p_w = k_w - 1$ 来使输入和输出具有相同的高和宽。这样会方便在构造网络时推测每个层的输出形状。假设这里 $k_h$ 是奇数，我们会在高的两侧分别填充 $p_h/2$ 行。如果 $k_h$ 是偶数，一种可能是在输入的顶端一侧填充 $\lceil p_h/2 \rceil$ 行（向上取整），而在底端一侧填充 $\lfloor p_h/2 \rfloor$ 行（向下取整）。在宽的两侧填充同理。

卷积神经网络经常使用奇数高宽的卷积核，如1、3、5和7，所以两端上的填充个数相等。对任意的二维数组 $x$ ，设它的第 $i$ 行第 $j$ 列的元素为 $x[i, j]$ 。当两端上的填充个数相等，并使输入和输出具有相同的高和宽时，我们就知道输出 $Y[i, j]$ 是由输入以 $x[i, j]$ 为中心的窗口同卷积核进行互相关计算得到的。

下面的例子里我们创建一个高和宽为3的二维卷积层，然后设输入高和宽两侧的填充数分别为1。给定一个高和宽为8的输入，我们发现输出的高和宽也是8。

```

1 In [1]: 
2 # 定义一个函数来计算卷积层。它对输入和输出做相应的升维和降维
3 def comp_conv2d(conv2d, X):
4     # (1, 1)代表批量大小和通道数（“多输入通道和多输出通道”一节将介绍）均为1
5     # (1, 1)+(8, 8)=(1, 1, 8, 8)
6     X = X.view((1, 1)+X.shape)
7     Y = conv2d(X)
8     # 排除不关心的前两维：批量和通道
9     return Y.view(Y.shape[2:])
10 # 注意这里是两侧分别填充1行或列，所以在两侧一共填充2行或列
11 conv2d = nn.Conv2d(in_channels=1, out_channels=1,
12                   kernel_size=3, padding=1)
13 X = torch.rand(8, 8)
14 comp_conv2d(conv2d, X).shape

```

```

15  Out [1]:
16      torch.size([8, 8])

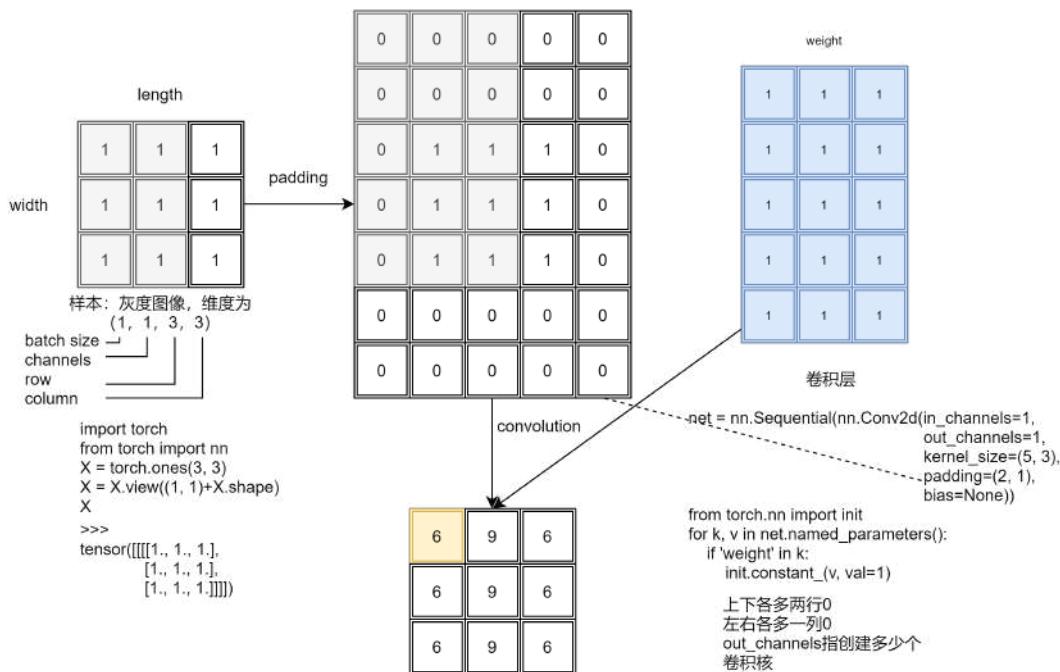
```

当卷积核的高和宽不同时，我们也可以通过设置高和宽上不同的填充数使输出和输入具有相同的高和宽。

```

1 In [2]:
2     # 使用高为5、宽为3的卷积核。在高和宽两侧的填充数分别为2和1
3     conv2d = nn.Conv2d(in_channels=1, out_channels=1,
4                         kernel_size=(5, 3), padding=(2, 1))
5     comp_conv2d(conv2d, X).shape
6 Out [2]:
7     torch.size([8, 8])

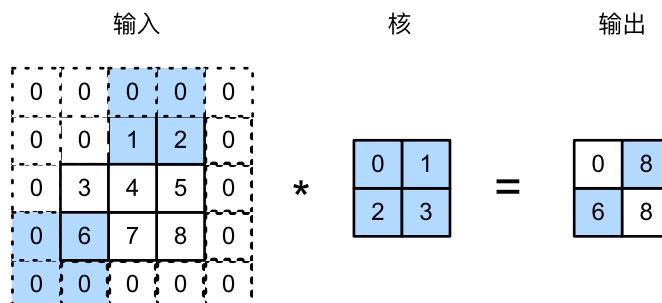
```



## 4.2.2 步幅

在上一节里我们介绍了二维互相关运算。卷积窗口从输入数组的最左上方开始，按从左往右、从上往下的顺序，依次在输入数组上滑动。我们将每次滑动的行数和列数称为步幅（stride）。

目前我们看到的例子里，在高和宽两个方向上步幅均为1。我们也可以使用更大步幅。下图展示了在高上步幅为3、在宽上步幅为2的二维互相关运算。可以看到，输出第一列第二个元素时，卷积窗口向下滑动了3行，而在输出第一行第二个元素时卷积窗口向右滑动了2列。当卷积窗口在输入上再向右滑动2列时，由于输入元素无法填满窗口，无结果输出。下图中的阴影部分为输出元素及其计算所使用的输入和核数组元素：

$$0 \times 0 + 0 \times 1 + 1 \times 2 + 2 \times 3 = 8, 0 \times 0 + 6 \times 1 + 0 \times 2 + 0 \times 3 = 6.$$


一般来说，当高上步幅为 $s_h$ ，宽上步幅为 $s_w$ 时，输出形状为

$$\lfloor (n_h - k_h + p_h + s_h) / s_h \rfloor \times \lfloor (n_w - k_w + p_w + s_w) / s_w \rfloor$$

其中， $n_h$  为输入张量的行数、 $k_h$  为卷积核的行数、 $p_h$  为两侧填充行数之和。如果设置  $p_h = k_h - 1$  和  $p_w = k_w - 1$ ，那么输出形状将简化为  $\lfloor (n_h + s_h - 1) / s_h \rfloor \times \lfloor (n_w + s_w - 1) / s_w \rfloor$ 。更进一步，如果输入的高和宽能分别被高和宽上的步幅整除，那么输出形状将是  $(n_h / s_h) \times (n_w / s_w)$ ，这里是因为后一部分计算结果小于 1，根据向下取整会被舍弃。

下面我们令高和宽上的步幅均为 2，从而使输入的高和宽减半。

```

1 In [3]:
2     conv2d = nn.Conv2d(1, 1, kernel_size=3, padding=1, stride=2)
3     comp_conv2d(conv2d, X).shape
4 Out [3]:
5     torch.Size([4, 4])

```

接下来是一个稍微复杂点儿的例子。

```

1 In [4]:
2     conv2d = nn.Conv2d(1, 1, kernel_size=(3, 5),
3                         padding=(0, 1), stride=(3, 4))
4     comp_conv2d(conv2d, X).shape
5 Out [4]:
6     torch.Size([2, 2])

```

为了表述简洁，当输入的高和宽两侧的填充数分别为  $p_h$  和  $p_w$  时，我们称填充为  $(p_h, p_w)$ 。特别地，当  $p_h = p_w = p$  时，填充为  $p$ 。当在高和宽上的步幅分别为  $s_h$  和  $s_w$  时，我们称步幅为  $(s_h, s_w)$ 。特别地，当  $s_h = s_w = s$  时，步幅为  $s$ 。在默认情况下，填充为 0，步幅为 1。

### 小结：

- 填充可以增加输出的高和宽。这常用来使输出与输入具有相同的高和宽。
- 步幅可以减小输出的高和宽，例如输出的高和宽仅为输入的高和宽的  $1/n$  ( $n$  为大于 1 的整数)。

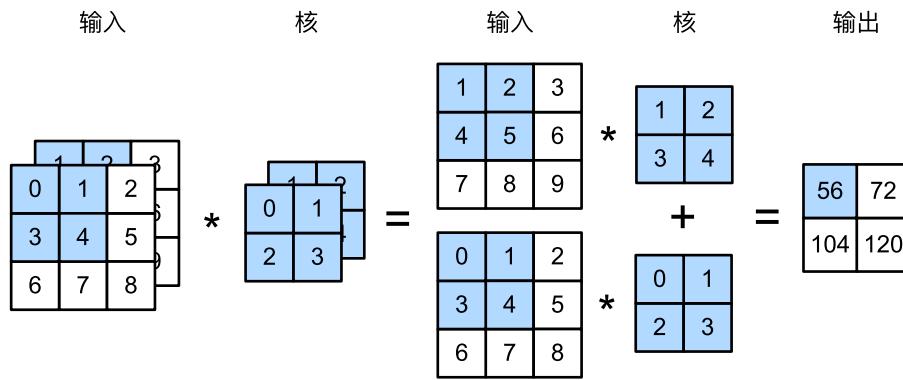
## 4.3 多输入通道和多输出通道

前面两节里我们用到的输入和输出都是二维数组，但真实数据的维度经常更高。例如，彩色图像在高和宽 2 个维度外还有 RGB（红、绿、蓝）3 个颜色通道。假设彩色图像的高和宽分别是  $h$  和  $w$ （像素），那么它可以表示为一个  $3 \times h \times w$  的多维数组。我们将大小为 3 的这一维称为通道（channel）维。本节我们将介绍含多个输入通道或多个输出通道的卷积核。

### 4.3.1 多输入通道

当输入数据含多个通道时，我们需要构造一个输入通道数与输入数据的通道数相同的卷积核，从而能够与含多通道的输入数据做互相关运算。假设输入数据的通道数为  $c_i$ ，那么卷积核的输入通道数同样为  $c_i$ 。设卷积核窗口形状为  $k_h \times k_w$ 。当  $c_i = 1$  时，我们知道卷积核只包含一个形状为  $k_h \times k_w$  的二维数组。当  $c_i > 1$  时，我们将会为每个输入通道各分配一个形状为  $k_h \times k_w$  的核数组。把这  $c_i$  个数组在输入通道维上连结，即得到一个形状为  $c_i \times k_h \times k_w$  的卷积核。由于输入和卷积核各有  $c_i$  个通道，我们可以在各个通道上对输入的二维数组和卷积核的二维核数组做互相关运算，再将这  $c_i$  个互相关运算的二维输出按通道相加，得到一个二维数组。这就是含多个通道的输入数据与多输入通道的卷积核做二维互相关运算的输出。

下图展示了含 2 个输入通道的二维互相关计算的例子。在每个通道上，二维输入数组与二维核数组做互相关运算，再按通道相加即得到输出。下图中阴影部分为第一个输出元素及其计算所使用的输入和核数组元素： $(1 \times 1 + 2 \times 2 + 4 \times 3 + 5 \times 4) + (0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3) = 56$ 。



接下来我们实现含多个输入通道的互相关运算。我们只需要对每个通道做互相关运算，然后将各个通道的计算结果进行累加。

```

1 In [1]:
2     import d2lzh as d2l
3     def corr2d_multi_in(x, K):
4         # 沿着X和K的第0维（通道维）分别计算再相加
5         res = d2l.corr2d(x[0, :, :], K[0, :, :])
6         for i in range(1, x.shape[0]):
7             res += d2l.corr2d(x[i, :, :], K[i, :, :])
8         return res

```

我们可以构造上图中的输入数组 `x`、核数组 `K` 来验证互相关运算的输出。

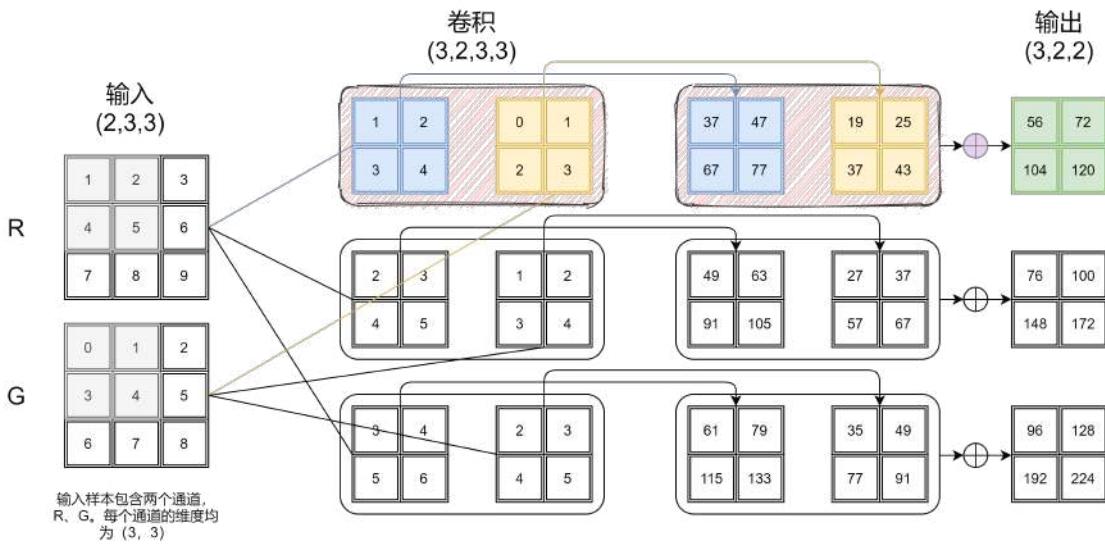
```

1 In [2]:
2     X = torch.tensor([
3         [[0, 1, 2], [3, 4, 5], [6, 7, 8]],
4         [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
5     ], dtype=torch.float)
6     K = torch.tensor([
7         [[0, 1], [2, 3]],
8         [[1, 2], [3, 4]]
9     ], dtype=torch.float)
10    corr2d_multi_in(X, K)
11 Out [2]:
12    tensor([[ 56.,  72.],
13            [104., 120.]])

```

### 4.3.2 多输出通道

当输入通道有多个时，因为我们对各个通道的结果做了累加，所以不论输入通道数是多少，输出通道数总是为1。设卷积核输入通道数和输出通道数分别为 $c_i$ 和 $c_o$ ，高和宽分别为 $k_h$ 和 $k_w$ 。如果希望得到含多个通道的输出，我们可以为每个输出通道分别创建形状为 $c_i \times k_h \times k_w$ 的核数组。将它们在输出通道维上连结，卷积核的形状即 $c_o \times c_i \times k_h \times k_w$ 。**在做互相关运算时，每个输出通道上的结果由卷积核在该输出通道上的核数组与整个输入数组计算而来。**



下面我们实现一个互相关运算函数来计算多个通道的输出。

```

1 In [3]:
2     def corr2d_multi_in_out(x, K):
3         # 对K的第0维遍历，每次同输入x做互相关计算。所有结果使用stack函数合并在一起
4         return torch.stack([corr2d_multi_in(x, k) for k in K])

```

我们将核数组 `K` 同 `K+1` (`K` 中每个元素加一) 和 `K+2` 连结在一起构造一个输出通道数为3的卷积核。

```

1 In [4]:
2     K = torch.stack([K, K+1, K+2])
3     K.shape
4 Out [4]:
5     torch.Size([3, 2, 2, 2])

```

下面我们对输入数组 `x` 与核数组 `K` 做互相关运算。此时的输出含有3个通道。其中第一个通道的结果与之前输入数组 `x` 与多输入通道、单输出通道核的计算结果一致。

```

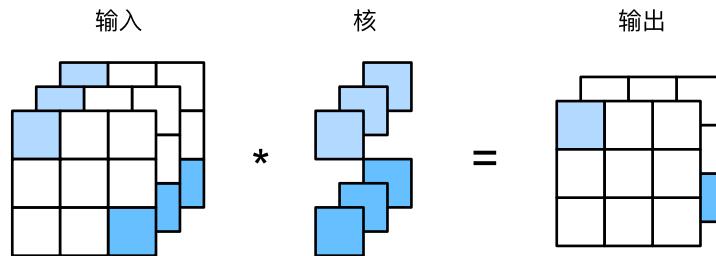
1 In [5]:
2     corr2d_multi_in_out(x, K)
3 Out [5]:
4     tensor([[ [ 56.,  72.],
5             [104., 120.] ],
6
6             [[ 76., 100.],
7             [148., 172.] ],
8
8             [[ 96., 128.],
9             [192., 224.] ]])

```

### 4.3.3 $1 \times 1$ 卷积层

最后我们讨论卷积窗口形状为  $1 \times 1$  ( $k_h = k_w = 1$ ) 的多通道卷积层。我们通常称之为  $1 \times 1$  卷积层，并将其中的卷积运算称为  $1 \times 1$  卷积。因为使用了最小窗口， $1 \times 1$  卷积失去了卷积层可以识别高和宽维度上相邻元素构成的模式的功能。**实际上， $1 \times 1$  卷积的主要计算发生在通道维上。**下图展示了使用输入通道数为3、输出通道数为2的  $1 \times 1$  卷积核的互相关计算。值得注意的是，输入和输出具有相同的高和宽。输出中的每个元素来自输入中在高和宽上相同位置的元素在不同通道之间的按权重累加。假设

我们将通道维当作特征维，将高和宽维度上的元素当成数据样本，那么 $1 \times 1$ 卷积层的作用与全连接层等价。



下面我们使用全连接层中的矩阵乘法来实现 $1 \times 1$ 卷积。这里需要在矩阵乘法运算前后对数据形状做一些调整。

```
1 In [6]:  
2     def corr2d_multi_in_out_1x1(x, K):  
3         # 通道、高、宽  
4         c_i, h, w = x.shape  
5         # 输出通道数目  
6         c_o = K.shape[0]  
7         # 通道、高*宽  
8         # 一个通道是一个特征  
9         x = x.view(c_i, h*w)  
10        K = K.view(c_o, c_i)  
11        # 全连接层的矩阵乘法  
12        Y = torch.mm(K, x)  
13        return Y.view(c_o, h, w)
```

经验证，做 $1 \times 1$ 卷积时，以上函数与之前实现的互相关运算函数 `corr2d_multi_in_out` 等价。

```
1 In [7]:  
2     X = torch.rand(3, 3, 3)  
3     K = torch.rand(2, 3, 1, 1)  
4     Y1 = corr2d_multi_in_out_1x1(X, K)  
5     Y2 = corr2d_multi_in_out(X, K)  
6     (Y1-Y2).norm().item() < 1e-6  
7 Out [7]:  
8     True
```

在之后的模型里我们将会看到 $1 \times 1$ 卷积层被当作保持高和宽维度形状不变的全连接层使用。于是，我们可以通过调整网络层之间的通道数来控制模型复杂度。

### 小结：

- 使用多通道可以拓展卷积层的模型参数。
- 假设将通道维当作特征维，将高和宽维度上的元素当成数据样本，那么 $1 \times 1$ 卷积层的作用与全连接层等价。
- $1 \times 1$ 卷积层通常用来调整网络层之间的通道数，并控制模型复杂度。

## 4.4 池化层

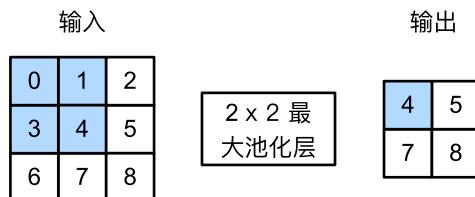
回忆一下，在4.1节（二维卷积层）里介绍的图像物体边缘检测应用中，我们构造卷积核从而精确地找到了像素变化的位置。设任意二维数组 `x` 的 `i` 行 `j` 列的元素为 `x[i, j]`。如果我们构造的卷积核输出 `Y[i, j]=1`，那么说明输入中 `x[i, j]` 和 `x[i, j+1]` 数值不一样。这可能意味着物体边缘通过这两个元素之间。但实际图像里，我们感兴趣的物体不会总出现在固定位置：即使我们连续拍摄同一个物体也

极有可能出现像素位置上的偏移。这会导致同一个边缘对应的输出可能出现在卷积输出  $Y$  中的不同位置，进而对后面的模式识别造成不便。

在本节中我们介绍池化（pooling）层，它的提出是为了缓解卷积层对位置的过度敏感性。

#### 4.4.1 二维最大池化层和平均池化层

同卷积层一样，池化层每次对输入数据的一个固定形状窗口（又称池化窗口）中的元素计算输出。不同于卷积层里计算输入和核的互相关性，池化层直接计算池化窗口内元素的最大值或者平均值。该运算也分别叫做最大池化或平均池化。在二维最大池化中，池化窗口从输入数组的最左上方开始，按从左往右、从上往下的顺序，依次在输入数组上滑动。当池化窗口滑动到某一位置时，窗口中的输入子数组的最大值即输出数组中相应位置的元素。



上图展示了池化窗口形状为  $2 \times 2$  的最大池化，阴影部分为第一个输出元素及其计算所使用的输入元素。输出数组的高和宽分别为 2，其中的 4 个元素由取最大值运算  $\max$  得出：

$$\begin{aligned}\max(0, 1, 3, 4) &= 4 \\ \max(1, 2, 4, 5) &= 5 \\ \max(3, 4, 6, 7) &= 7 \\ \max(4, 5, 7, 8) &= 8\end{aligned}$$

二维平均池化的工作原理与二维最大池化类似，但将最大运算符替换成平均运算符。池化窗口形状为  $p \times q$  的池化层称为  $p \times q$  池化层，其中的池化运算叫作  $p \times q$  池化。

让我们再次回到本节开始提到的物体边缘检测的例子。现在我们将卷积层的输出作为  $2 \times 2$  最大池化的输入。设该卷积层输入是  $x$ 、池化层输出为  $y$ 。无论是  $x[i, j]$  和  $x[i, j+1]$  值不同，还是  $x[i, j+1]$  和  $x[i, j+2]$  不同，池化层输出均有  $y[i, j]=1$ 。也就是说，使用  $2 \times 2$  最大池化层时，只要卷积层识别的模式在高和宽上移动不超过一个元素，我们依然可以将它检测出来。

下面把池化层的前向计算实现在 `pool2d` 函数里。它跟 4.1 节（二维卷积层）里 `corr2d` 函数非常类似，唯一的区别在计算输出  $y$  上。

```
1 In [1]:  
2     def pool2d(x, pool_size, mode='max'):  
3         x = x.float()  
4         p_h, p_w = pool_size  
5         # 存储池化计算结果  
6         Y = torch.zeros(X.shape[0]-p_h+1, X.shape[1]-p_w+1)  
7         for i in range(Y.shape[0]):  
8             for j in range(Y.shape[1]):  
9                 if mode=='max':  
10                     Y[i, j] = x[i: i+p_h, j: j+p_w].max()  
11                 elif mode=='avg':  
12                     Y[i, j] = x[i: i+p_h, j: j+p_w].mean()  
13         return Y
```

我们可以构造上图中的输入数组  $x$  来验证二维最大池化层的输出。

```
1 In [2]:  
2     x = torch.tensor([  
3         [0, 1, 2],  
4         [3, 4, 5],  
5         [6, 7, 8]  
6     ])  
7     pool2d(x, (2, 2))  
8 Out [2]:  
9     tensor([[4., 5.],  
10            [7., 8.]])
```

同时我们试验一下平均池化层。

```
1 In [3]:  
2     pool2d(x, (2, 2), mode='avg')  
3 Out [3]:  
4     tensor([[2., 3.],  
5             [5., 6.]])
```

## 4.4.2 填充和步幅

同卷积层一样，池化层也可以在输入的高和宽两侧的填充并调整窗口的移动步幅来改变输出形状。池化层填充和步幅与卷积层填充和步幅的工作机制一样。我们将通过 nn 模块里的二维最大池化层 `MaxPool2d` 来演示池化层填充和步幅的工作机制。我们先构造一个形状为(1, 1, 4, 4)的输入数据，前两个维度分别是批量和通道。

```
1 In [4]:  
2     x = torch.arange(16, dtype=torch.float).view((1, 1, 4, 4))  
3     x  
4 Out [4]:  
5     tensor([[[[ 0.,  1.,  2.,  3.],  
6                 [ 4.,  5.,  6.,  7.],  
7                 [ 8.,  9., 10., 11.],  
8                 [12., 13., 14., 15.]]]])
```

默认情况下，`MaxPool2d` 实例里步幅和池化窗口形状相同。下面使用形状为(3, 3)的池化窗口，默认获得形状为(3, 3)的步幅。

```
1 In [5]:  
2     pool2d = nn.MaxPool2d(3)  
3     pool2d(x)  
4 Out [5]:  
5     tensor([[[[10.]]]])
```

我们可以手动指定步幅和填充。

```
1 In [6]:  
2     pool2d = nn.MaxPool2d(3, padding=1, stride=2)  
3     pool2d(x)  
4 Out [6]:  
5     tensor([[[[ 5.,  7.],  
6                 [13., 15.]]]])
```

当然，我们也可以指定非正方形的池化窗口，并分别指定高和宽上的填充和步幅。

```
1 In [7]:  
2     # PyTorch要求padding应该比池化窗口的一半小  
3     pool2d = nn.MaxPool2d((2, 4), padding=(1, 2), stride=(2, 3))  
4     pool2d(x)  
5 Out [7]:  
6     tensor([[[[ 1.,  3.],  
7             [ 9., 11.],  
8             [13., 15.]]]])
```

### 4.4.3 多通道

在处理多通道输入数据时，池化层对每个输入通道分别池化，而不是像卷积层那样将各通道的输出按通道相加。这意味着池化层的输出通道数与输入通道数相等。下面将数组`x`和`x+1`在通道维上连结来构造通道数为2的输入。

```
1 In [8]:  
2     x = torch.cat((x, x+1), dim=1)  
3     x  
4 Out [8]:  
5     tensor([[[[ 0.,  1.,  2.,  3.],  
6                 [ 4.,  5.,  6.,  7.],  
7                 [ 8.,  9., 10., 11.],  
8                 [12., 13., 14., 15.]],  
9  
10                [[ 1.,  2.,  3.,  4.],  
11                  [ 5.,  6.,  7.,  8.],  
12                  [ 9., 10., 11., 12.],  
13                  [13., 14., 15., 16.]]]))
```

池化后，我们发现输出通道数仍然是2。

```
1 In [9]:  
2     pool2d = nn.MaxPool2d(3, padding=1, stride=2)  
3     pool2d(x)  
4 Out [9]:  
5     tensor([[[[ 5.,  7.],  
6                 [13., 15.]],  
7  
8                 [[ 6.,  8.],  
9                 [14., 16.]]]])
```

#### 小结：

- 最大池化和平均池化分别取池化窗口中输入元素的最大值和平均值作为输出。
- 池化层的一个主要作用是缓解卷积层对位置的过度敏感性。
- 可以指定池化层的填充和步幅。
- 池化层的输出通道数跟输入通道数相同。

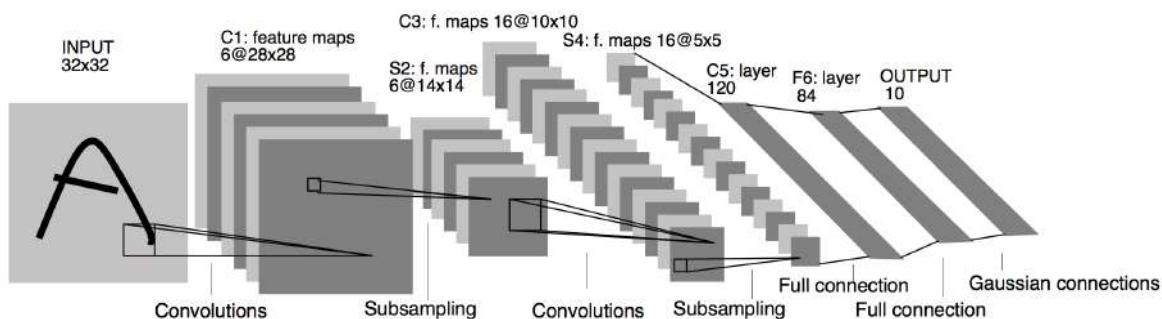
## 4.5 卷积神经网络（LeNet）

在2.9节（多层感知机的从零开始实现）一节里我们构造了一个含单隐藏层的多层感知机模型来对Fashion-MNIST数据集中的图像进行分类。每张图像高和宽均是28像素。我们将图像中的像素逐行展开，得到长度为784的向量，并输入进全连接层中。然而，这种分类方法有一定的局限性。

1. 图像在同一列邻近的像素在这个向量中可能相距较远。它们构成的模式可能难以被模型识别。
2. 对于大尺寸的输入图像，使用全连接层容易导致模型过大。假设输入是高和宽均为1,000像素的彩色照片（含3个通道）。即使全连接层输出个数仍是256，该层权重参数的形状也是 $3,000,000 \times 256$ ：它占用了大约3 GB的内存或显存。这会带来过于复杂的模型和过高的存储开销。

卷积层尝试解决这两个问题。一方面，卷积层保留输入形状，使图像的像素在高和宽两个方向上的相关性均可能被有效识别；另一方面，卷积层通过滑动窗口将同一卷积核与不同位置的输入重复计算，从而避免参数尺寸过大。

卷积神经网络就是含卷积层的网络。本节里我们将介绍一个早期用来识别手写数字图像的卷积神经网络：LeNet。这个名字来源于LeNet论文的第一作者Yann LeCun。LeNet展示了通过梯度下降训练卷积神经网络可以达到手写数字识别在当时最先进的结果。这个奠基性的工作第一次将卷积神经网络推上舞台，为世人所知。



## 4.5.1 LeNet模型

LeNet分为卷积层块和全连接层块两个部分。下面我们分别介绍这两个模块。

卷积层块里的基本单位是卷积层后接最大池化层：卷积层用来识别图像里的空间模式，如线条和物体局部，之后的最大池化层则用来降低卷积层对位置的敏感性。卷积层块由两个这样的基本单位重复堆叠构成。在卷积层块中，每个卷积层都使用 $5 \times 5$ 的窗口，并在输出上使用sigmoid激活函数。第一个卷积层输出通道数为6，第二个卷积层输出通道数则增加到16。**这是因为第二个卷积层比第一个卷积层的输入的高和宽要小，所以增加输出通道使两个卷积层的参数尺寸类似。**卷积层块的两个最大池化层的窗口形状均为 $2 \times 2$ ，且步幅为2。由于池化窗口与步幅形状相同，池化窗口在输入上每次滑动所覆盖的区域互不重叠。

卷积层块的输出形状为(批量大小, 通道, 高, 宽)。当卷积层块的输出传入全连接层块时，全连接层块会将小批量中每个样本变平 (flatten)。也就是说，全连接层的输入形状将变成二维，其中第一维是小批量中的样本，第二维是每个样本变平后的向量表示，且向量长度为通道、高和宽的乘积。全连接层块含3个全连接层。它们的输出个数分别是120、84和10，其中10为输出的类别个数。

下面我们通过 `Sequential` 类来实现LeNet模型。

```

1 In [1]:
2     from torch import nn, optim
3     device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
4     class LeNet(nn.Module):
5         def __init__(self):
6             super(LeNet, self).__init__()
7             self.conv = nn.Sequential(
8                 # in_channels, out_channels, kernel_size
9                 nn.Conv2d(1, 6, 5),
10                nn.Sigmoid(),
11                # kernel_size, stride
12                nn.MaxPool2d(2, 2),
13                nn.Conv2d(6, 16, 5),

```

```

14         nn.Sigmoid(),
15         nn.MaxPool2d(2, 2)
16     )
17     self.fc = nn.Sequential(
18         nn.Linear(16*4*4, 120),
19         nn.Sigmoid(),
20         nn.Linear(120, 84),
21         nn.Sigmoid(),
22         nn.Linear(84, 10)
23     )
24     def forward(self, img):
25         feature = self.conv(img)
26         output = self.fc(feature.view(img.shape[0], -1))
27         return output

```

接下来查看每个层的形状。

```

1 In [2]:
2     net = LeNet()
3     print(net)
4 Out [2]:
5     LeNet(
6         (conv): Sequential(
7             (0): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
8             (1): Sigmoid()
9             (2): MaxPool2d(kernel_size=2, stride=2, padding=0,
10                           dilation=1, ceil_mode=False)
11             (3): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
12             (4): Sigmoid()
13             (5): MaxPool2d(kernel_size=2, stride=2, padding=0,
14                           dilation=1, ceil_mode=False)
15         )
16         (fc): Sequential(
17             (0): Linear(in_features=256, out_features=120, bias=True)
18             (1): Sigmoid()
19             (2): Linear(in_features=120, out_features=84, bias=True)
20             (3): Sigmoid()
21             (4): Linear(in_features=84, out_features=10, bias=True)
22         )
23     )

```

可以看到，在卷积层块中输入的高和宽在逐层减小。卷积层由于使用高和宽均为5的卷积核，从而将高和宽分别减小4，而池化层则将高和宽减半，但通道数则从1增加到16。全连接层则逐层减少输出个数，直到变成图像的类别数10。

## 4.5.2 训练模型

下面我们来实验LeNet模型。实验中，我们仍然使用Fashion-MNIST作为训练数据集。

```

1 In [3]:
2     batch_size = 256
3     train_iter, test_iter = d2l.load_data_fashion_mnist(
4         batch_size=batch_size
5     )

```

因为卷积神经网络计算比多层感知机要复杂，建议使用GPU来加速计算。因此，我们对2.6节（softmax回归的从零开始实现）中描述的 evaluate\_accuracy 函数略作修改，使其支持GPU计算。

```
1 In [4]:  
2 def evaluate_accuracy(data_iter, net, device=None):  
3     """  
4     function:  
5         计算多分类模型预测结果的准确率  
6  
7     Parameters:  
8         data_iter - 样本划分为最小批的结果  
9         net - 定义的网络  
10        device - 指定计算在GPU或者CPU上进行  
11  
12    Returns:  
13        准确率计算结果  
14  
15    Modify:  
16        2020-11-30  
17        2020-12-03 增加模型训练模型和推理模式的判别  
18        2020-12-10 增加指定运行计算位置的方法  
19        """  
20    if device is None and isinstance(net, torch.nn.Module):  
21        # 如果没指定device就使用net的device  
22        device = list(net.parameters())[0].device  
23    acc_sum, n = 0.0, 0  
24    for x, y in data_iter:  
25        if isinstance(net, torch.nn.Module):  
26            # 评估模式，这会关闭dropout  
27            net.eval()  
28            # .cpu()保证可以进行数值加减  
29            acc_sum += (net(x.to(device)).argmax(dim=1) ==  
30                         y.to(device)).float().sum().cpu().item()  
31            # 改回训练模式  
32            net.train()  
33            # 自定义的模型，2.13节之后不会用到，不考虑GPU  
34        else:  
35            if 'is_training' in net.__code__.co_varnames:  
36                # 将is_training设置成False  
37                acc_sum += (net(x, is_training=False).argmax(dim=1) ==  
38                             y).float().sum().item()  
39            else:  
40                acc_sum += (net(x).argmax(dim=1) == y).float().sum().item()  
41            n += y.shape[0]  
42    return acc_sum / n
```

我们同样对2.6节中定义的 train\_ch3 函数略作修改，确保计算使用的数据和模型同在内存或显存上。

```
1 In [5]:  
2 def train_ch5(net, train_iter, test_iter, batch_size,  
3                 optimizer, device, num_epochs):  
4     """  
5     function:  
6         利用softmax回归模型对图像进行分类识别  
7
```

```

8  Parameters:
9      net - 定义的网络
10     train_iter - 训练集样本划分为最小批的结果
11     test_iter - 测试集样本划分为最小批的结果
12     num_epochs - 迭代次数
13     batch_size - 最小批大小
14     optimizer - 优化器
15     device - 指定计算在GPU或者CPU上进行
16
17 Returns:
18
19 Modify:
20     2020-12-10
21     """
22     # 将模型加载到指定运算器中
23     net = net.to(device)
24     print("training on ", device)
25     loss = torch.nn.CrossEntropyLoss()
26     for epoch in range(num_epochs):
27         train_l_sum, train_acc_sum, n, batch_count = 0.0, 0.0, 0, 0
28         start = time.time()
29         for X, y in train_iter:
30             X = X.to(device)
31             y = y.to(device)
32             y_hat = net(X)
33             l = loss(y_hat, y)
34             # 梯度清零
35             optimizer.zero_grad()
36             l.backward()
37             optimizer.step()
38             train_l_sum += l.cpu().item()
39             train_acc_sum += (y_hat.argmax(dim=1) == y).sum().cpu().item()
40             n += y.shape[0]
41             batch_count += 1
42         test_acc = evaluate_accuracy(test_iter, net)
43         print('epoch %d, loss %.4f, train acc %.3f, test acc %.3f, \
44               time %.1f sec' % (epoch+1, train_l_sum/batch_count,
45                               train_acc_sum/n, test_acc,
46                               time.time()-start))

```

学习率采用0.001，训练算法使用Adam算法，损失函数使用交叉熵损失函数。

```

1 In [6]:
2     lr, num_epochs = 0.001, 5
3     optimizer = torch.optim.Adam(net.parameters(), lr=lr)
4     d2l.train_ch5(net, train_iter, test_iter, batch_size,
5                   optimizer, device, num_epochs)
6 Out [6]:
7     training on cuda
8     epoch 1, loss 1.9234, train acc 0.288, test acc 0.585, time 4.8 sec
9     epoch 2, loss 0.9566, train acc 0.630, test acc 0.672, time 4.2 sec
10    epoch 3, loss 0.7686, train acc 0.713, test acc 0.722, time 4.2 sec
11    epoch 4, loss 0.6820, train acc 0.737, test acc 0.744, time 4.4 sec
12    epoch 5, loss 0.6315, train acc 0.753, test acc 0.749, time 4.7 sec

```

**小结:**

- 卷积神经网络就是含卷积层的网络。
- LeNet交替使用卷积层和最大池化层后接全连接层来进行图像分类。

## 4.6 深度卷积神经网络 (AlexNet)

在LeNet提出后的将近20年里，神经网络一度被其他机器学习方法超越，如支持向量机。虽然LeNet可以在早期的小数据集上取得好的成绩，但是在更大的真实数据集上的表现并不尽如人意。一方面，神经网络计算复杂。虽然20世纪90年代也有过一些针对神经网络的加速硬件，但并没有像之后GPU那样大量普及。因此，训练一个多通道、多层和有大量参数的卷积神经网络在当年很难完成。另一方面，当年研究者还没有大量深入研究参数初始化和非凸优化算法等诸多领域，导致复杂的神经网络的训练通常较困难。

我们在上一节看到，神经网络可以直接基于图像的原始像素进行分类。这种称为端到端 (end-to-end) 的方法节省了很多中间步骤。然而，在很长一段时间里更流行的是研究者通过勤劳与智慧所设计并生成的手工特征。这类图像分类研究的主要流程是：

1. 获取图像数据集；
2. 使用已有的特征提取函数生成图像的特征；
3. 使用机器学习模型对图像的特征分类。

当时认为的机器学习部分仅限最后这一步。如果那时候跟机器学习研究者交谈，他们会认为机器学习既重要又优美。优雅的定理证明了许多分类器的性质。机器学习领域生机勃勃、严谨而且极其有用。然而，如果跟计算机视觉研究者交谈，则是另外一幅景象。他们会告诉你图像识别里“不可告人”的现实是：计算机视觉流程中真正重要的是数据和特征。也就是说，使用较干净的数据集和较有效的特征甚至比机器学习模型的选择对图像分类结果的影响更大。

### 4.6.1 学习特征表示

既然特征如此重要，它该如何表示呢？

我们已经提到，在相当长的时间里，特征都是基于各式各样手工设计的函数从数据中提取的。事实上，不少研究者通过提出新的特征提取函数不断改进图像分类结果。这一度为计算机视觉的发展做出了重要贡献。

然而，另一些研究者则持异议。他们认为特征本身也应该由学习得来。他们还相信，为了表征足够复杂的输入，**特征本身应该分级表示**。持这一想法的研究者相信，多层神经网络可能可以学得数据的多级表征，并逐级表示越来越抽象的概念或模式。以图像分类为例，并回忆4.1节（二维卷积层）中物体边缘检测的例子。在多层神经网络中，图像的第一级的表示可以是在特定的位置和角度是否出现边缘；而第二级的表示说不定能够将这些边缘组合出有趣的模式，如花纹；在第三级的表示中，也许上一级的花纹能进一步汇合成对应物体特定部位的模式。这样逐级表示下去，最终，模型能够较容易根据最后一级的表示完成分类任务。需要强调的是，输入的逐级表示由多层模型中的参数决定，而这些参数都是学出来的。

尽管一直有一群执着的研究者不断钻研，试图学习视觉数据的逐级表征，然而很长一段时间里这些野心都未能实现。这其中有很多因素值得我们一一分析。

#### 1. 缺失要素一：数据

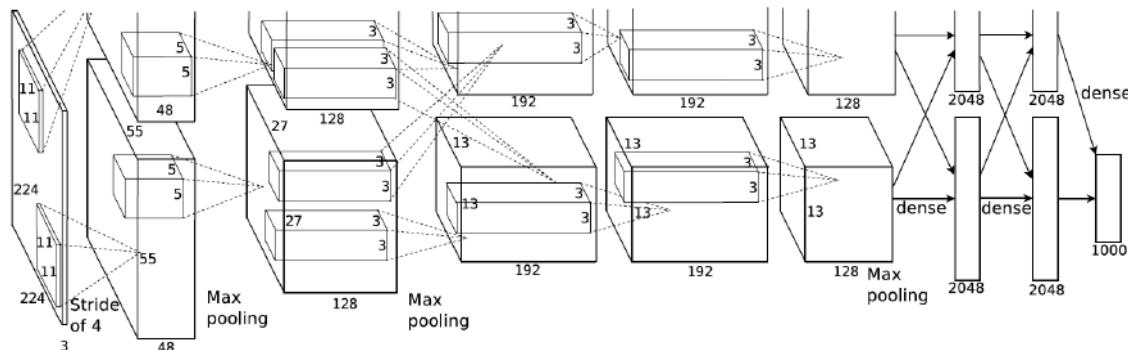
包含许多特征的深度模型需要大量的有标签的数据才能表现得比其他经典方法更好。限于早期计算机有限的存储和90年代有限的研究预算，大部分研究只基于小的公开数据集。例如，不少研究论文基于加州大学欧文分校 (UCI) 提供的若干个公开数据集，其中许多数据集只有几百至几千张图像。这一状况在2010年前后兴起的大数据浪潮中得到改善。特别是，2009年诞生的ImageNet数据集包含了1,000大类物体，每类有多达数千张不同的图像。这一规模是当时其他公开数据集无法与之相提并论的。ImageNet数据集同时推动计算机视觉和机器学习研究进入新的阶段，使此前的传统方法不再有优势。

#### 2. 缺失要素一：硬件

深度学习对计算资源要求很高。早期的硬件计算能力有限，这使训练较复杂的神经网络变得很困难。然而，通用GPU的到来改变了这一格局。很久以来，GPU都是为图像处理和计算机游戏设计的，尤其是针对大吞吐量的矩阵和向量乘法从而服务于基本的图形变换。值得庆幸的是，这其中的数学表达与深度网络中的卷积层的表达类似。通用GPU这个概念在2001年开始兴起，涌现出诸如OpenCL和CUDA之类的编程框架。这使得GPU也在2010年前后开始被机器学习社区使用。

## 4.6.2 AlexNet

2012年，AlexNet横空出世。这个模型的名字来源于论文第一作者的姓名Alex Krizhevsky。AlexNet使用了8层卷积神经网络，并以很大的优势赢得了ImageNet 2012图像识别挑战赛。它首次证明了学习到的特征可以超越手工设计的特征，从而一举打破计算机视觉研究的前状。



AlexNet与LeNet的设计理念非常相似，但也有显著的区别。

第一，与相对较小的LeNet相比，AlexNet包含8层变换，其中有5层卷积和2层全连接隐藏层，以及1个全连接输出层。下面我们来详细描述这些层的设计。

AlexNet第一层中的卷积窗口形状是 $11 \times 11$ 。因为ImageNet中绝大多数图像的高和宽均比MNIST图像的高和宽大10倍以上，ImageNet图像的物体占用更多的像素，所以需要更大的卷积窗口来捕获物体。第二层中的卷积窗口形状减小到 $5 \times 5$ ，之后全采用 $3 \times 3$ 。此外，第一、第二和第五个卷积层之后都使用了窗口形状为 $3 \times 3$ 、步幅为2的最大池化层。而且，AlexNet使用的卷积通道数也大于LeNet中的卷积通道数数十倍。

紧接着最后一个卷积层的是两个输出个数为4096的全连接层。这两个巨大的全连接层带来将近1 GB的模型参数。由于早期显存的限制，最早的AlexNet使用双数据流的设计使一个GPU只需要处理一半模型。幸运的是，显存在过去几年得到了长足的发展，因此通常我们不再需要这样的特别设计了。

第二，AlexNet将sigmoid激活函数改成了更加简单的ReLU激活函数。一方面，ReLU激活函数的计算更简单，例如它并没有sigmoid激活函数中的求幂运算。另一方面，**ReLU激活函数在不同的参数初始化方法下使模型更容易训练**。这是由于当sigmoid激活函数输出极接近0或1时，这些区域的梯度几乎为0，从而造成反向传播无法继续更新部分模型参数；而ReLU激活函数在正区间的梯度恒为1。因此，若模型参数初始化不当，sigmoid函数可能在正区间得到几乎为0的梯度，从而令模型无法得到有效训练。

第三，AlexNet通过丢弃法（参见2.13节）来控制全连接层的模型复杂度。而LeNet并没有使用丢弃法。

第四，AlexNet引入了大量的图像增广，如翻转、裁剪和颜色变化，从而进一步扩大数据集来缓解过拟合。我们将在后面的8.1节（图像增广）详细介绍这种方法。

下面我们实现稍微简化过的AlexNet。

```
1 In [1]:  
2     device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')  
3     class AlexNet(nn.Module):  
4         def __init__(self):  
5             super(AlexNet, self).__init__()
```

```

6         self.conv = nn.Sequential(
7             # in_channels, out_channels, kernel_size, stride
8             nn.Conv2d(1, 96, 11, 4),
9             nn.ReLU(),
10            # kernel_size, stride
11            nn.MaxPool2d(3, 2),
12            # 减小卷积窗口，使用填充为2来使得输入与输出的高和宽一致,
13            # 且增大输出通道数
14            nn.Conv2d(96, 256, 5, 1, 2),
15            nn.ReLU(),
16            nn.MaxPool2d(3, 2),
17            # 连续3个卷积层，且使用更小的卷积窗口。除了最后的卷积层外,
18            # 进一步增大了输出通道数。
19            # 前两个卷积层后不使用池化层来减小输入的高和宽
20            nn.Conv2d(256, 384, 3, 1, 1),
21            nn.ReLU(),
22            nn.Conv2d(384, 384, 3, 1, 1),
23            nn.ReLU(),
24            nn.Conv2d(384, 256, 3, 1, 1),
25            nn.ReLU(),
26            nn.MaxPool2d(3, 2)
27        )
28        # 这里全连接层的输出个数比LeNet中的大数倍。使用丢弃层来缓解过拟合
29        self.fc = nn.Sequential(
30            nn.Linear(256*5*5, 4096),
31            nn.ReLU(),
32            nn.Dropout(0.5),
33            nn.Linear(4096, 4096),
34            nn.ReLU(),
35            nn.Dropout(0.5),
36            # 输出层。由于这里使用Fashion-MNIST
37            # 所以用类别数为10，而非论文中的1000
38            nn.Linear(4096, 10)
39        )
40    def forward(self, img):
41        feature = self.conv(img)
42        output = self.fc(feature.view(img.shape[0], -1))
43        return output

```

打印看看网络结构。

```

1 In [2]:
2     net = AlexNet()
3     print(net)
4 Out [2]:
5 AlexNet(
6     (conv): Sequential(
7         (0): Conv2d(1, 96, kernel_size=(11, 11), stride=(4, 4))
8         (1): ReLU()
9         (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
10                     ceil_mode=False)
11         (3): Conv2d(96, 256, kernel_size=(5, 5), stride=(1, 1),
12                     padding=(2, 2))
13         (4): ReLU()
14         (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
15                     ceil_mode=False)
16         (6): Conv2d(256, 384, kernel_size=(3, 3), stride=(1, 1),

```

```

17         padding=(1, 1))
18     (7): ReLU()
19     (8): Conv2d(384, 384, kernel_size=(3, 3), stride=(1, 1),
20                 padding=(1, 1))
21     (9): ReLU()
22     (10): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1),
23                  padding=(1, 1))
24     (11): ReLU()
25     (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
26                       ceil_mode=False)
27 )
28 (fc): Sequential(
29     (0): Linear(in_features=6400, out_features=4096, bias=True)
30     (1): ReLU()
31     (2): Dropout(p=0.5)
32     (3): Linear(in_features=4096, out_features=4096, bias=True)
33     (4): ReLU()
34     (5): Dropout(p=0.5)
35     (6): Linear(in_features=4096, out_features=10, bias=True)
36   )
37 )

```

### 4.6.3 读取数据集

虽然论文中AlexNet使用ImageNet数据集，但因为ImageNet数据集训练时间较长，我们仍用前面的Fashion-MNIST数据集来演示AlexNet。读取数据的时候我们额外做了一步将图像高和宽扩大到AlexNet使用的图像高和宽——224。这个可以通过 `torchvision.transforms.Resize` 实例来实现。也就是说，我们在 `totensor` 实例前使用 `Resize` 实例，然后使用 `Compose` 实例来将这两个变换串联以方便调用。

```

1 In [3]:
2 def load_data_fashion_mnist(batch_size, resize=None):
3     """
4     function:
5         将fashion mnist数据集划分为小批量样本
6
7     Parameters:
8         batch_size - 小批量样本的大小(int)
9         resize - 对图像的维度进行扩大
10
11    Returns:
12        train_iter - 训练集样本划分为最小批的结果
13        test_iter - 测试集样本划分为最小批的结果
14
15    Modify:
16        2020-11-26
17        2020-12-10 添加图像维度变化
18    """
19
20    # 存储图像处理流程
21    trans = []
22    if resize:
23        trans.append(transforms.Resize(size=resize))
24    trans.append(transforms.ToTensor())
25    transform = transforms.Compose(trans)
26    mnist_train = torchvision.datasets.FashionMNIST(
27        root='data/FashionMNIST',
28        train=True, transform=transform, download=True)
29    mnist_test = torchvision.datasets.FashionMNIST(
30        root='data/FashionMNIST',
31        train=False, transform=transform, download=True)
32    return batchify.DataLoaderIter(
33        (mnist_train, mnist_test), batch_size=batch_size, shuffle=True)

```

```

1      train=True,
2      download=True,
3      transform=transform)
4  mnist_train = torchvision.datasets.FashionMNIST(root='data/FashionMNIST',
5                                              train=True,
6                                              download=True,
7                                              transform=transform)
8
9  if sys.platform.startswith('win'):
10     # 0表示不用额外的进程来加速读取数据
11     num_workers = 0
12
13 else:
14     num_workers = 4
15
16 train_iter = torch.utils.data.DataLoader(mnist_train,
17                                         batch_size=batch_size,
18                                         shuffle=True,
19                                         num_workers=num_workers)
20
21 test_iter = torch.utils.data.DataLoader(mnist_test,
22                                         batch_size=batch_size,
23                                         shuffle=False,
24                                         num_workers=num_workers)
25
26
27 return train_iter, test_iter
28
29
30 batch_size = 256
31
32 train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)

```

## 4.6.4 训练模型

这时候我们可以开始训练AlexNet了。相对于LeNet，由于图片尺寸变大了而且模型变大了，所以需要更大的显存，也需要更长的训练时间了。

```

1 In [4]:
2     lr, num_epochs = 0.001, 5
3     optimizer = torch.optim.Adam(net.parameters(), lr=lr)
4     d2l.train_ch5(net, train_iter, test_iter, batch_size, optimizer,
5                   device, num_epochs)
6 Out [4]:
7     training on  cuda
8     epoch 1, loss 0.6961, train acc 0.732, test acc 0.841, time 26.5 sec
9     epoch 2, loss 0.3328, train acc 0.874, test acc 0.884, time 26.5 sec
10    epoch 3, loss 0.2833, train acc 0.893, test acc 0.894, time 26.1 sec
11    epoch 4, loss 0.2503, train acc 0.907, test acc 0.903, time 26.5 sec
12    epoch 5, loss 0.2291, train acc 0.915, test acc 0.906, time 26.6 sec

```

### 小结:

- AlexNet跟LeNet结构类似，但使用了更多的卷积层和更大的参数空间来拟合大规模数据集ImageNet。它是浅层神经网络和深度神经网络的分界线。
- 虽然看上去AlexNet的实现比LeNet的实现也就多了几行代码而已，但这个观念上的转变和真正优秀实验结果的产生令学术界付出了很多年。

## 4.7 使用重复元素的网络 (VGG)

AlexNet在LeNet的基础上增加了3个卷积层。但AlexNet作者对它们的卷积窗口、输出通道数和构造顺序均做了大量的调整。虽然AlexNet指明了深度卷积神经网络可以取得出色的结果，但并没有提供简单的规则以指导后来的研究者如何设计新的网络。我们将在本章的后续几节里介绍几种不同的深度网络设计思路。

本节介绍VGG，它的名字来源于论文作者所在的实验室Visual Geometry Group。VGG提出了可以通过重复使用简单的基础块来构建深度模型的思路。

## 4.7.1 VGG块

VGG块的组成规律是：连续使用数个相同的填充为1、窗口形状为 $3 \times 3$ 的卷积层后接上一个步幅为2、窗口形状为 $2 \times 2$ 的最大池化层。卷积层保持输入的高和宽不变，而池化层则对其减半。我们使用`vgg_block`函数来实现这个基础的VGG块，它可以指定卷积层的数量和输入输出通道数。

对于给定的感受野（与输出有关的输入图片的局部大小），采用堆积的小卷积核优于采用大的卷积核，因为可以增加网络深度来保证学习更复杂的模式，而且代价还比较小（参数更少）。例如，在VGG中，使用了3个 $3 \times 3$ 卷积核来代替 $7 \times 7$ 卷积核，使用了2个 $3 \times 3$ 卷积核来代替 $5 \times 5$ 卷积核，这样做的主要目的是在保证具有相同感知野的条件下，提升了网络的深度，在一定程度上提升了神经网络的效果。

```
1 In [1]:  
2     def vgg_block(num_convs, in_channels, out_channels):  
3         blk = []  
4         for i in range(num_convs):  
5             if i==0:  
6                 blk.append(nn.Conv2d(in_channels, out_channels,  
7                                     kernel_size=3, padding=1))  
8             else:  
9                 blk.append(nn.Conv2d(out_channels, out_channels,  
10                           kernel_size=3, padding=1))  
11            blk.append(nn.ReLU())  
12            # 这里会使宽高减半  
13            blk.append(nn.MaxPool2d(kernel_size=2, stride=2))  
14        return nn.Sequential(*blk)
```

## 4.7.2 VGG网络

与AlexNet和LeNet一样，VGG网络由卷积层模块后接全连接层模块构成。卷积层模块串联数个`vgg_block`，其超参数由变量`conv_arch`定义。该变量指定了每个VGG块里卷积层个数和输入输出通道数。全连接模块则跟AlexNet中的一样。

现在我们构造一个VGG网络。它有5个卷积块，前2块使用单卷积层，而后3块使用双卷积层。第一块的输入输出通道分别是1（因为下面要使用的Fashion-MNIST数据的通道数为1）和64，之后每次对输出通道数翻倍，直到变为512。因为这个网络使用了8个卷积层和3个全连接层，所以经常被称为VGG-11。

```
1 In [2]:  
2     conv_arch = ((1, 1, 64), (1, 64, 128), (2, 128, 256),  
3                     (2, 256, 512), (2, 512, 512))  
4     # 经过5个vgg_block，宽高会减半5次，变成 $224/2^{**5}=224/32=7$   
5     # c * h * w  
6     fc_features = 512*7*7  
7     fc_hidden_units = 4096
```

下面我们实现VGG-11。

```
1 In [3]:  
2     def vgg(conv_arch, fc_features, fc_hidden_units=4096):  
3         net = nn.Sequential()  
4         # 卷积层部分  
5         for i, (num_convs, in_channels,
```

```

6         out_channels) in enumerate(conv_arch):
7             # 每经过一个vgg_block都会使宽高减半
8             net.add_module('vgg_block_' + str(i+1),
9                             vgg_block(num_convs, in_channels, out_channels))
10            # 全连接部分
11            net.add_module('fc', nn.Sequential(
12                d2l.FlattenLayer(),
13                nn.Linear(fc_features, fc_hidden_units),
14                nn.ReLU(),
15                nn.Dropout(0.5),
16                nn.Linear(fc_hidden_units, fc_hidden_units),
17                nn.ReLU(),
18                nn.Dropout(0.5),
19                nn.Linear(fc_hidden_units, 10)
20            ))
21
22    return net

```

下面构造一个高和宽均为224的单通道数据样本来观察每一层的输出形状。

```

1 In [4]:
2     net = vgg(conv_arch, fc_features, fc_hidden_units)
3     x = torch.rand(1, 1, 224, 224)
4     # named_children获取一级子模块及其名字
5     # named_modules会返回所有子模块，包括子模块的子模块
6     for name, blk in net.named_children():
7         x = blk(x)
8         print(name, 'output shape: ', x.shape)
9 Out [4]:
10    vgg_block_1 output shape:  torch.Size([1, 64, 112, 112])
11    vgg_block_2 output shape:  torch.Size([1, 128, 56, 56])
12    vgg_block_3 output shape:  torch.Size([1, 256, 28, 28])
13    vgg_block_4 output shape:  torch.Size([1, 512, 14, 14])
14    vgg_block_5 output shape:  torch.Size([1, 512, 7, 7])
15    fc output shape:  torch.Size([1, 10])

```

可以看到，每次我们将输入的高和宽减半，直到最终高和宽变成7后传入全连接层。与此同时，输出通道数每次翻倍，直到变成512。因为每个卷积层的窗口大小一样，所以每层的模型参数尺寸和计算复杂度与输入高、输入宽、输入通道数和输出通道数的乘积成正比。**VGG这种高和宽减半以及通道翻倍的设计使得多数卷积层都有相同的模型参数尺寸和计算复杂度。**

### 4.7.3 训练模型

因为VGG-11计算上比AlexNet更加复杂，出于测试的目的我们构造一个通道数更小，或者说更窄的网络在Fashion-MNIST数据集上进行训练。

```

1 In [5]:
2     ratio = 4
3     small_conv_arch = ((1, 1, 64//ratio),
4                         (1, 64//ratio, 128//ratio),
5                         (2, 128//ratio, 256//ratio),
6                         (2, 256//ratio, 512//ratio),
7                         (2, 512//ratio, 512//ratio))
8
9     net = vgg(small_conv_arch, fc_features//ratio, fc_hidden_units//ratio)

```

模型训练过程与上一节的AlexNet中的类似。

```

1 In [6]:
2     batch_size = 256
3     train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size,
4                                                     resize=224)
5     lr, num_epochs = 0.001, 5
6     optimizer = torch.optim.Adam(net.parameters(), lr=lr)
7     d2l.train_ch5(net, train_iter, test_iter, batch_size, optimizer,
8                   device, num_epochs)
9 Out [6]:
10    training on cuda
11    epoch 1, loss 0.7833, train acc 0.702, test acc 0.839, time 44.3 sec
12    epoch 2, loss 0.3814, train acc 0.860, test acc 0.873, time 44.5 sec
13    epoch 3, loss 0.3125, train acc 0.885, test acc 0.895, time 44.5 sec
14    epoch 4, loss 0.2730, train acc 0.899, test acc 0.904, time 44.6 sec
15    epoch 5, loss 0.2428, train acc 0.911, test acc 0.912, time 44.5 sec

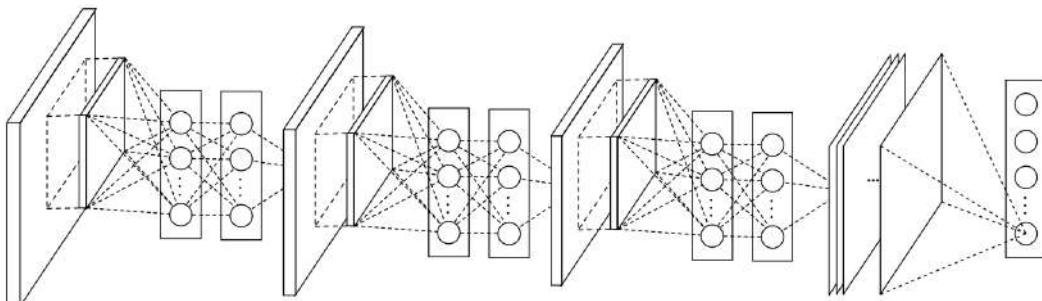
```

**小结:**

- VGG-11通过5个可以重复使用的卷积块来构造网络。根据每块里卷积层个数和输出通道数的不同可以定义出不同的VGG模型。

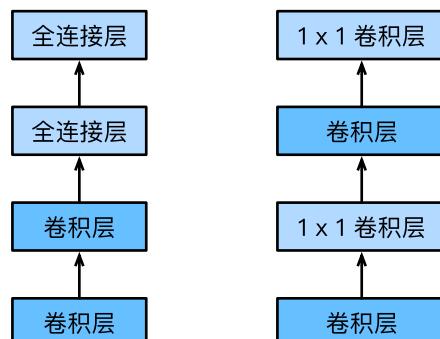
## 4.8 网络中的网络 (NiN)

前几节介绍的LeNet、AlexNet和VGG在设计上的共同之处是：先以由卷积层构成的模块充分抽取空间特征，再以由全连接层构成的模块来输出分类结果。其中，AlexNet和VGG对LeNet的改进主要在于如何对这两个模块加宽（增加通道数）和加深。本节我们介绍网络中的网络（NiN）。它提出了另外一个重要思路，即串联多个由卷积层和“全连接”层构成的小网络来构建一个深层网络。



### 4.8.1 NiN块

我们知道，卷积层的输入和输出通常是四维数组（样本，通道，高，宽），而全连接层的输入和输出则通常是二维数组（样本，特征）。如果想在全连接层后再接上卷积层，则需要将全连接层的输出变换为四维。回忆在4.3节（多输入通道和多输出通道）里介绍的 $1 \times 1$ 卷积层。它可以看成全连接层，其中空间维度（高和宽）上的每个元素相当于样本，通道相当于特征。因此，**NiN使用 $1 \times 1$ 卷积层来替代全连接层，从而使空间信息能够自然传递到后面的层中去**。下图对比了NiN同AlexNet和VGG等网络在结构上的主要区别。



左图是AlexNet和VGG的网络结构局部，右图是NiN的网络结构局部

NiN块是NiN中的基础块。它由一个卷积层加两个充当全连接层的 $1 \times 1$ 卷积层串联而成。其中第一个卷积层的超参数可以自行设置，而第二和第三个卷积层的超参数一般是固定的。

```
1 In [1]:  
2     device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')  
3     def nin_block(in_channels, out_channels, kernel_size, stride, padding):  
4         blk = nn.Sequential(  
5             nn.Conv2d(in_channels, out_channels, kernel_size, stride,  
6                     padding),  
7             nn.ReLU(),  
8             # 相当于全连接层  
9             nn.Conv2d(out_channels, out_channels, kernel_size=1),  
10            nn.ReLU(),  
11            nn.Conv2d(out_channels, out_channels, kernel_size=1),  
12            nn.ReLU()  
13        )  
14        return blk
```

## 4.8.2 NiN模型

NiN是在AlexNet问世不久后提出的。它们的卷积层设定有类似之处。NiN使用卷积窗口形状分别为 $11 \times 11$ 、 $5 \times 5$ 和 $3 \times 3$ 的卷积层，相应的输出通道数也与AlexNet中的一致。每个NiN块后接一个步幅为2、窗口形状为 $3 \times 3$ 的最大池化层。

除使用NiN块以外，NiN还有一个设计与AlexNet显著不同：NiN去掉了AlexNet最后的3个全连接层，取而代之地，NiN使用了输出通道数等于标签类别数的NiN块，然后使用全局平均池化层对每个通道中所有元素求平均并直接用于分类。**这里的全局平均池化层即窗口形状等于输入空间维形状的平均池化层。NiN的这个设计的好处是可以显著减小模型参数尺寸，从而缓解过拟合。**然而，该设计有时会造成获得有效模型的训练时间的增加。

```
1 In [2]:  
2     import torch.nn.functional as F  
3     class GlobalAvgPool2d(nn.Module):  
4         # 全局平均池化层可通过将池化窗口形状设置成输入的高和宽实现  
5         def __init__(self):  
6             super(GlobalAvgPool2d, self).__init__()  
7         def forward(self, x):  
8             return F.avg_pool2d(x, kernel_size=x.size()[2:])  
9     net = nn.Sequential(  
10        nin_block(1, 96, kernel_size=11, stride=4, padding=0),  
11        nn.MaxPool2d(kernel_size=3, stride=2),  
12        nin_block(96, 256, kernel_size=5, stride=1, padding=2),  
13        nn.MaxPool2d(kernel_size=3, stride=2),  
14        nin_block(256, 384, kernel_size=3, stride=1, padding=1),  
15        nn.MaxPool2d(kernel_size=3, stride=2),  
16        nn.Dropout(0.5),  
17        # 标签类别数是10  
18        nin_block(384, 10, kernel_size=3, stride=1, padding=1),  
19        GlobalAvgPool2d(),  
20        # 将四维的输出转成二维的输出，其形状为(批量大小, 10)  
21        d2l.FlattenLayer()  
22    )
```

我们构建一个数据样本来查看每一层的输出形状。

```

1 In [3]:
2     x = torch.rand(1, 1, 224, 224)
3     for name, blk in net.named_children():
4         x = blk(x)
5         print(name, 'output shape: ', x.shape)
6 Out [3]:
7     0 output shape:  torch.Size([1, 96, 54, 54])
8     1 output shape:  torch.Size([1, 96, 26, 26])
9     2 output shape:  torch.Size([1, 256, 26, 26])
10    3 output shape:  torch.Size([1, 256, 12, 12])
11    4 output shape:  torch.Size([1, 384, 12, 12])
12    5 output shape:  torch.Size([1, 384, 5, 5])
13    6 output shape:  torch.Size([1, 384, 5, 5])
14    7 output shape:  torch.Size([1, 10, 5, 5])
15    8 output shape:  torch.Size([1, 10, 1, 1])
16    9 output shape:  torch.Size([1, 10])

```

### 4.8.3 训练模型

我们依然使用Fashion-MNIST数据集来训练模型。NiN的训练与AlexNet和VGG的类似，但这里使用的学习率更大。

```

1 In [4]:
2     batch_size = 256
3     train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size,
4                                                       resize=224)
5     lr, num_epochs = 0.002, 5
6     optimizer = torch.optim.Adam(net.parameters(), lr=lr)
7     d2l.train_ch5(net, train_iter, test_iter, batch_size,
8                   optimizer, device, num_epochs)
9 Out [4]:
10    training on  cuda
11    epoch 1, loss 1.2753, train acc 0.524, test acc 0.726, time 31.9 sec
12    epoch 2, loss 0.6357, train acc 0.767, test acc 0.805, time 32.0 sec
13    epoch 3, loss 0.5100, train acc 0.815, test acc 0.832, time 32.2 sec
14    epoch 4, loss 0.4403, train acc 0.840, test acc 0.843, time 32.0 sec
15    epoch 5, loss 0.3973, train acc 0.854, test acc 0.856, time 32.0 sec

```

#### 小结:

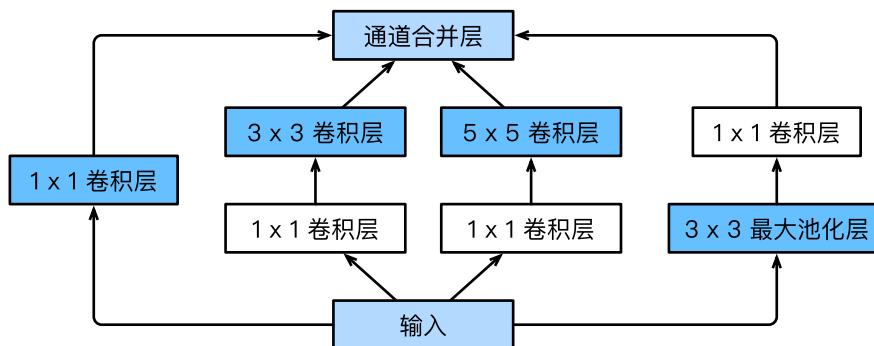
- NiN重复使用由卷积层和代替全连接层的 $1 \times 1$ 卷积层构成的NiN块来构建深层网络。
- NiN去除了容易造成过拟合的全连接输出层，而是将其替换成输出通道数等于标签类别数的NiN块和全局平均池化层。
- NiN的以上设计思想影响了后面一系列卷积神经网络的设计。

## 4.9 合并行连结的网络 (GoogLeNet)

在2014年的ImageNet图像识别挑战赛中，一个名叫GoogLeNet的网络结构大放异彩。它虽然在名字上向LeNet致敬，但在网络结构上已经很难看到LeNet的影子。GoogLeNet吸收了NiN中网络串联网络的思想，并在此基础上做了很大改进。在随后的几年里，研究人员对GoogLeNet进行了数次改进，本节将介绍这个模型系列的第一个版本。

## 4.9.1 Inception块

GoogLeNet中的基础卷积块叫作Inception块，得名于同名电影《盗梦空间》（Inception）。与上一节介绍的NiN块相比，这个基础块在结构上更加复杂，如下图所示。

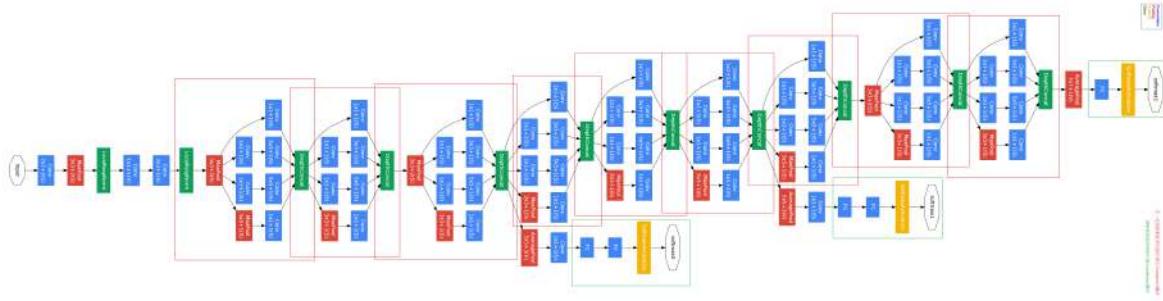


由上图可以看出，Inception块里有4条并行的线路。前3条线路使用窗口大小分别是 $1 \times 1$ 、 $3 \times 3$ 和 $5 \times 5$ 的卷积层来抽取不同空间尺寸下的信息，**其中中间2个线路会对输入先做 $1 \times 1$ 卷积来减少输入通道数，以降低模型复杂度**。第四条线路则使用 $3 \times 3$ 最大池化层，后接 $1 \times 1$ 卷积层来改变通道数。4条线路都使用了合适的填充来使**输入与输出的高和宽一致**。最后我们将每条线路的输出在通道维上连结，并输入接下来的层中去。**采用不同大小的卷积核意味着不同大小的感受野，最后拼接意味着不同尺度特征的融合**。之所以卷积核大小采用1、3和5，主要是为了方便对齐。设定卷积步长stride=1之后，只要分别设定padding=0、1、2，那么卷积之后便可以得到相同维度的特征，然后这些特征就可以直接拼接在一起了。而很多地方都表明pooling挺有效，所以Inception里面也嵌入了。

Inception块中可以自定义的超参数是每个层的输出通道数，我们以此来控制模型复杂度。

```
1 In [1]:
2     device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
3     class Inception(nn.Module):
4         # c1-c4为每条路线里的层的输出通道数
5         def __init__(self, in_c, c1, c2, c3, c4):
6             super(Inception, self).__init__()
7             # 线路1, 单1*1卷积层
8             self.p1_1 = nn.Conv2d(in_c, c1, kernel_size=1)
9             # 线路2, 1*1卷积层后接3*3卷积层
10            self.p2_1 = nn.Conv2d(in_c, c2[0], kernel_size=1)
11            self.p2_2 = nn.Conv2d(c2[0], c2[1], kernel_size=3, padding=1)
12            # 线路3, 1*1卷积层后接5*5卷积层
13            self.p3_1 = nn.Conv2d(in_c, c3[0], kernel_size=1)
14            self.p3_2 = nn.Conv2d(c3[0], c3[1], kernel_size=5, padding=2)
15            # 线路4, 3*3最大池化层后接1*1卷积层
16            self.p4_1 = nn.MaxPool2d(kernel_size=3, stride=1, padding=1)
17            self.p4_2 = nn.Conv2d(in_c, c4, kernel_size=1)
18         def forward(self, x):
19             p1 = F.relu(self.p1_1(x))
20             p2 = F.relu(self.p2_2(self.p2_1(x)))
21             p3 = F.relu(self.p3_2(self.p3_1(x)))
22             p4 = F.relu(self.p4_2(self.p4_1(x)))
23             # 在通道维上连结输出
24             return torch.cat((p1, p2, p3, p4), dim=1)
```

## 4.9.2 GoogLeNet模型



GoogLeNet跟VGG一样，在主体卷积部分中使用5个模块（block），每个模块之间使用步幅为2的 $3 \times 3$ 最大池化层来减小输出高宽。第一模块使用一个64通道的 $7 \times 7$ 卷积层。

```
1 In [2]:  
2     b1 = nn.Sequential(  
3         nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3),  
4         nn.ReLU(),  
5         nn.MaxPool2d(kernel_size=3, stride=2, padding=1)  
6     )
```

第二模块使用2个卷积层：首先是64通道的 $1 \times 1$ 卷积层，然后是将通道增大3倍的 $3 \times 3$ 卷积层。它对应Inception块中的第二条线路。

```
1 In [3]:  
2     b2 = nn.Sequential(  
3         nn.Conv2d(64, 64, kernel_size=1),  
4         nn.ReLU(),  
5         nn.Conv2d(64, 192, kernel_size=3, padding=1),  
6         nn.ReLU(),  
7         nn.MaxPool2d(kernel_size=3, stride=2, padding=1)  
8     )
```

第三模块串联2个完整的Inception块。第一个Inception块的输出通道数为 $64 + 128 + 32 + 32 = 256$ ，其中4条线路的输出通道数比例为 $64 : 128 : 32 : 32 = 2 : 4 : 1 : 1$ 。其中第二、第三条线路先分别将输入通道数减小至 $96/192 = 1/2$ 和 $16/192 = 1/12$ 后，再接上第二层卷积层。第二个Inception块输出通道数增至 $128 + 192 + 96 + 64 = 480$ ，每条线路的输出通道数之比为 $128 : 192 : 96 : 64 = 4 : 6 : 3 : 2$ 。其中第二、第三条线路先分别将输入通道数减小至 $128/256 = 1/2$ 和 $32/256 = 1/8$ 。

```
1 In [4]:  
2     b3 = nn.Sequential(  
3         Inception(192, 64, (96, 128), (16, 32), 32),  
4         Inception(256, 128, (128, 192), (32, 96), 64),  
5         nn.MaxPool2d(kernel_size=3, stride=2, padding=1)  
6     )
```

第四模块更加复杂。它串联了5个Inception块，其输出通道数分别是 $192 + 208 + 48 + 64 = 512$ 、 $160 + 224 + 64 + 64 = 512$ 、 $128 + 256 + 64 + 64 = 512$ 、 $112 + 288 + 64 + 64 = 528$ 和 $256 + 320 + 128 + 128 = 832$ 。这些线路的通道数分配和第三模块中的类似，首先含 $3 \times 3$ 卷积层的第二条线路输出最多通道，其次是仅含 $1 \times 1$ 卷积层的第一条线路，之后是含 $5 \times 5$ 卷积层的第三条线路和含 $3 \times 3$ 最大池化层的第四条线路。其中第二、第三条线路都会先按比例减小通道数。这些比例在各个Inception块中都略有不同。

```
1 In [5]:  
2     b4 = nn.Sequential(  
3         Inception(480, 192, (96, 208), (16, 48), 64),  
4         Inception(512, 160, (112, 224), (24, 64), 64),  
5         Inception(512, 128, (128, 256), (24, 64), 64),  
6         Inception(512, 112, (144, 288), (32, 64), 64),  
7         Inception(528, 256, (160, 320), (32, 128), 128),  
8         nn.MaxPool2d(kernel_size=3, stride=2, padding=1)  
9     )
```

第五模块有输出通道数为 $256 + 320 + 128 + 128 = 832$ 和 $384 + 384 + 128 + 128 = 1024$ 的两个Inception块。其中每条线路的通道数的分配思路和第三、第四模块中的一致，只是在具体数值上有所不同。需要注意的是，第五模块的后面紧跟输出层，该模块同NiN一样使用全局平均池化层来将每个通道的高和宽变成1。最后我们将输出变成二维数组后接上一个输出个数为标签类别数的全连接层。

```
1 In [6]:  
2     b5 = nn.Sequential(  
3         Inception(832, 256, (160, 320), (32, 128), 128),  
4         Inception(832, 384, (192, 384), (48, 128), 128),  
5         GlobalAvgPool2d()  
6     )  
7     net = nn.Sequential(  
8         b1, b2, b3, b4, b5,  
9         d21.FlattenLayer(),  
10        nn.Linear(1024, 10)  
11    )
```

GoogLeNet模型的计算复杂，而且不如VGG那样便于修改通道数。本节里我们将输入的高和宽从224降到96来简化计算。下面演示各个模块之间的输出的形状变化。

```
1 In [7]:  
2     x = torch.rand(1, 1, 96, 96)  
3     for blk in net.children():  
4         x = blk(x)  
5         print('output shape: ', x.shape)  
6 Out [7]:  
7     output shape:  torch.Size([1, 64, 24, 24])  
8     output shape:  torch.Size([1, 192, 12, 12])  
9     output shape:  torch.Size([1, 480, 6, 6])  
10    output shape:  torch.Size([1, 832, 3, 3])  
11    output shape:  torch.Size([1, 1024, 1, 1])  
12    output shape:  torch.Size([1, 1024])  
13    output shape:  torch.Size([1, 10])
```

### 4.9.3 训练模型

我们使用高和宽均为96像素的图像来训练GoogLeNet模型。训练使用的图像依然来自Fashion-MNIST数据集。

```
1 In [8]:  
2     batch_size = 256  
3     train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size,  
4                                                       resize=96)  
5     lr, num_epochs = 0.001, 5
```

```

6     optimizer = torch.optim.Adam(net.parameters(), lr=lr)
7     d2l.train_ch5(net, train_iter, test_iter, batch_size, optimizer,
8                     device, num_epochs)
9
10    out [8]:
11        training on cuda
12        epoch 1, loss 2.4547, train acc 0.099, test acc 0.100, time 23.0 sec
13        epoch 2, loss 1.6381, train acc 0.344, test acc 0.496, time 23.1 sec
14        epoch 3, loss 0.7035, train acc 0.733, test acc 0.726, time 23.1 sec
15        epoch 4, loss 0.5071, train acc 0.815, test acc 0.824, time 23.1 sec
16        epoch 5, loss 0.4271, train acc 0.842, test acc 0.826, time 23.1 sec

```

## 小结:

- Inception块相当于一个有4条线路的子网络。它通过不同窗口形状的卷积层和最大池化层来并行抽取信息，并使用 $1 \times 1$ 卷积层减少通道数从而降低模型复杂度。
- GoogLeNet将多个设计精细的Inception块和其他层串联起来。其中Inception块的通道数分配之比是在ImageNet数据集上通过大量的实验得来的。
- GoogLeNet和它的后继者们一度是ImageNet上最高效的模型之一：在类似的测试精度下，它们的计算复杂度往往更低。

## 4.10 批量归一化

本节我们介绍批量归一化 (batch normalization) 层，它能让较深的神经网络的训练变得更加容易。在2.16节（实战Kaggle比赛：预测房价）里，我们对输入数据做了标准化处理：处理后的任意一个特征在数据集中所有样本上的均值为0、标准差为1。**标准化处理输入数据使各个特征的分布相近：这往往更容易训练出有效的模型。**

通常来说，数据标准化预处理对于浅层模型就足够有效了。随着模型训练的进行，当每层中参数更新时，靠近输出层的输出较难出现剧烈变化。但对深层神经网络来说，即使输入数据已做标准化，训练中模型参数的更新依然很容易造成靠近输出层输出的剧烈变化。这种计算数值的不稳定性通常令我们难以训练出有效的深度模型。

批量归一化的提出正是为了应对深度模型训练的挑战。**在模型训练时，批量归一化利用小批量上的均值和标准差，不断调整神经网络中间输出，从而使整个神经网络在各层的中间输出的数值更稳定。批量归一化和下一节将要介绍的残差网络为训练和设计深度模型提供了两类重要思路。**

### 4.10.1 批量归一化层

对全连接层和卷积层做批量归一化的方法稍有不同。下面我们将分别介绍这两种情况下的批量归一化。

#### 1. 对全连接层做批量归一化

我们先考虑如何对全连接层做批量归一化。通常，我们将批量归一化层置于全连接层中的仿射变换和激活函数之间。设全连接层的输入为 $\mathbf{u}$ ，权重参数和偏差参数分别为 $\mathbf{W}$ 和 $\mathbf{b}$ ，激活函数为 $\phi$ 。设批量归一化的运算符为BN。那么，使用批量归一化的全连接层的输出为

$$\phi(\text{BN}(\mathbf{x}))$$

其中批量归一化输入 $\mathbf{x}$ 由仿射变换

$$\mathbf{x} = \mathbf{W}\mathbf{u} + \mathbf{b}$$

得到。考虑一个由 $m$ 个样本组成的小批量，仿射变换的输出为一个新的小批量 $\mathcal{B} = \mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}$ 。它们正是批量归一化层的输入。对于小批量 $\mathcal{B}$ 中任意样本 $\mathbf{x}^{(i)} \in \mathbb{R}^d, 1 \leq i \leq m$ ，批量归一化层的输出同样是一个 $d$ 维向量

$$\mathbf{y}^{(i)} = \text{BN}(\mathbf{x}^{(i)})$$

并由以下几步求得。首先，对小批量 $\mathcal{B}$ 求均值和方差：

$$\begin{aligned}\boldsymbol{\mu}_{\mathcal{B}} &\leftarrow \frac{1}{m} \sum_{i=1}^m \mathbf{x}^{(i)} \\ \sigma_{\mathcal{B}}^2 &\leftarrow \frac{1}{m} \sum_{i=1}^m (\mathbf{x}^{(i)} - \boldsymbol{\mu}_{\mathcal{B}})^2\end{aligned}$$

其中的平方计算是按元素求平方。接下来，使用按元素开方和按元素除法对 $\mathbf{x}^{(i)}$ 标准化（Z-score标准化）：

$$\hat{\mathbf{x}}^{(i)} \leftarrow \frac{\mathbf{x}^{(i)} - \boldsymbol{\mu}_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$$

这里 $\epsilon > 0$ 是一个很小的常数，保证分母大于0。在上面标准化的基础上，批量归一化层引入了两个可以学习的模型参数，拉伸（scale）参数 $\gamma$ 和偏移（shift）参数 $\beta$ 。这两个参数和 $\mathbf{x}^{(i)}$ 形状相同，皆为 $d$ 维向量。它们与 $\hat{\mathbf{x}}^{(i)}$ 分别做按元素乘法（符号 $\odot$ ）和加法计算：

$$\mathbf{y}^{(i)} \leftarrow \gamma \odot \hat{\mathbf{x}}^{(i)} + \beta$$

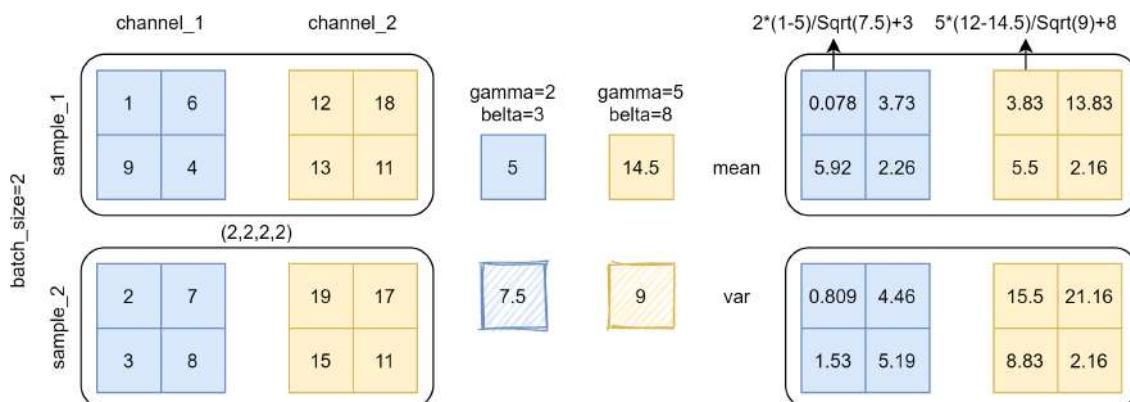
至此，我们得到了 $\mathbf{x}^{(i)}$ 的批量归一化的输出 $\mathbf{y}^{(i)}$ 。值得注意的是，可学习的拉伸和偏移参数保留了不对 $\mathbf{x}^{(i)}$ 做批量归一化的可能：此时只需学出 $\gamma = \sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}$ 和 $\beta = \boldsymbol{\mu}_{\mathcal{B}}$ 。我们可以对此这样理解：如果批量归一化无益，理论上，学出的模型可以不使用批量归一化。

## 2. 对卷积层做批量归一化

对卷积层来说，批量归一化发生在卷积计算之后、应用激活函数之前。**如果卷积计算输出多个通道，我们需要对这些通道的输出分别做批量归一化，且每个通道都拥有独立的拉伸和偏移参数，并均为标量。**设小批量中有 $m$ 个样本。在单个通道上，假设卷积计算输出的高和宽分别为 $p$ 和 $q$ 。我们需要对该通道中 $m \times p \times q$ 个元素同时做批量归一化。对这些元素做标准化计算时，我们使用相同的均值和方差，即该通道中 $m \times p \times q$ 个元素的均值和方差。

## 3. 预训练时的批量归一化

使用批量归一化训练时，我们可以将批量大小设得大一点，从而使批量内样本的均值和方差的计算都较为准确。将训练好的模型用于预测时，我们希望模型对于任意输入都有确定的输出。因此，单个样本的输出不应取决于批量归一化所需要的随机小批量中的均值和方差。一种常用的方法是**通过移动平均估算整个训练数据集的样本均值和方差，并在预测时使用它们得到确定的输出**。可见，和丢弃层一样，批量归一化层在训练模式和预测模式下的计算结果也是不一样的。



## 4.10.2 从零开始实现

下面我们通过Tensor来实现批量归一化层。

```
1 In [1]:  
2     device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')  
3     def batch_norm(is_training, x, gamma, beta, moving_mean,  
4                     moving_var, eps, momentum):  
5         # 判断当前模式是训练模式还是预测模式  
6         if not is_training:  
7             # 如果是在预测模式下, 直接使用传入的移动平均所得的均值和方差  
8             x_hat = (x - moving_mean) / torch.sqrt(moving_var + eps)  
9         else:  
10            # 前一层需要为全连接层或卷积层  
11            assert len(x.shape) in (2, 4)  
12            # 全连接层  
13            if len(x.shape) == 2:  
14                # 沿纵向求均值, (1, 特征个数)  
15                # 注意: 逐特征求均值  
16                mean = x.mean(dim=0)  
17                # 广播机制  
18                var = ((x - mean) ** 2).mean(dim=0)  
19            else:  
20                # 使用二维卷积层的情况, 计算通道维上 (axis=1) 的均值和方差。  
21                # 这里我们需要保持x的形状以便后面可以做广播运算  
22                mean = x.mean(dim=0, keepdim=True).  
23                mean(dim=2, keepdim=True).mean(dim=3, keepdim=True)  
24                var = ((x - mean) ** 2).mean(dim=0, keepdim=True).  
25                mean(dim=2, keepdim=True).mean(dim=3, keepdim=True)  
26                # 以上代码可以优化为  
27                # mean = x.mean(dim=(0, 2, 3), keepdim=True)  
28            # 训练模式下用当前的均值和方差做标准化  
29            x_hat = (x - mean) / torch.sqrt(var + eps)  
30            # 一阶指数平滑算法  
31            moving_mean = momentum * moving_mean + (1.0 - momentum) * mean  
32            moving_var = momentum * moving_var + (1.0 - momentum) * var  
33            # 拉伸和偏移  
34            Y = gamma * x_hat + beta  
35            return Y, moving_mean, moving_var
```

### • 一阶指数平滑算法

序列长度记为  $n$ , 参数记为  $\alpha$  (指数平滑算法只有一个参数)。我们定义时间序列:

$S = s_1 s_2 \cdots s_n$ , 同时定义拟合序列:  $T = t_1 t_2 \cdots t_n$ 。那么对于一阶指数平滑而言

$$t_i = \alpha s_i + (1 - \alpha)t_{i-1}$$

这里我们需要定义一下初始值, 一般来说, 我们可以令初值是前3个数据的平均值:  $t_0 = \frac{s_0 + s_1 + s_2}{3}$ 。我们通过这样的设置之后, 便可以利用初始值不断迭代出下一步的拟合值, 也就是通过  $t_0$  可以不断推到  $t_n$  的值。

接下来, 我们自定义一个 `BatchNorm` 层。它保存参与求梯度和迭代的拉伸参数 `gamma` 和偏移参数 `beta`, 同时也维护移动平均得到的均值和方差, 以便能够在模型预测时被使用。`BatchNorm` 实例所需指定的 `num_features` 参数对于全连接层来说应为输出个数, 对于卷积层来说则为输出通道数。该实例所需指定的 `num_dims` 参数对于全连接层和卷积层来说分别为2和4。

```

1 In [2]:
2     class BatchNorm(nn.Module):
3         def __init__(self, num_features, num_dims):
4             super(BatchNorm, self).__init__()
5             # 全连接层
6             if num_dims==2:
7                 shape = (1, num_features)
8             # 卷积层
9             else:
10                shape = (1, num_features, 1, 1)
11            # 参与求梯度和迭代的拉伸和偏移参数, 分别初始化成0和1
12            self.gamma = nn.Parameter(torch.ones(shape))
13            self.beta = nn.Parameter(torch.zeros(shape))
14            # 不参与求梯度和迭代的变量, 全在内存上初始化成0
15            self.moving_mean = torch.zeros(shape)
16            self.moving_var = torch.zeros(shape)
17        def forward(self, x):
18            # 如果x不在显存上, 将moving_mean和moving_var复制到x所在显存上
19            if self.moving_mean.device != x.device:
20                self.moving_mean = self.moving_mean.to(x.device)
21                self.moving_var = self.moving_var.to(x.device)
22            # 保存更新过的moving_mean和moving_var
23            # Module实例的traning属性默认为true, 调用.eval()后设成false
24            Y, self.moving_mean, self.moving_var = batch_norm(
25                self.training, x, self.gamma, self.beta, self.moving_mean,
26                self.moving_var, eps=1e-5, momentum=0.9)
27            return Y

```

### 4.10.3 使用批量归一化层的LeNet

下面我们修改4.5节（卷积神经网络（LeNet））介绍的LeNet模型，从而应用批量归一化层。我们在所有的卷积层或全连接层之后、激活层之前加入批量归一化层。

```

1 In [3]:
2     import d2lzh as d2l
3     net = nn.Sequential(
4         # in_channels, out_channels, kernel_size
5         nn.Conv2d(1, 6, 5),
6         BatchNorm(6, num_dims=4),
7         nn.Sigmoid(),
8         # kernel_size, stride
9         nn.MaxPool2d(2, 2),
10        nn.Conv2d(6, 16, 5),
11        BatchNorm(16, num_dims=4),
12        nn.Sigmoid(),
13        nn.MaxPool2d(2, 2),
14        d2l.FlattenLayer(),
15        nn.Linear(16*4*4, 120),
16        BatchNorm(120, num_dims=2),
17        nn.Sigmoid(),
18        nn.Linear(120, 84),
19        BatchNorm(84, num_dims=2),
20        nn.Sigmoid(),
21        nn.Linear(84, 10)
22    )

```

下面我们训练修改后的模型。

```
1 In [4]:  
2     batch_size = 256  
3     train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)  
4     lr, num_epochs = 0.001, 5  
5     optimizer = torch.optim.Adam(net.parameters(), lr=lr)  
6     d2l.train_ch5(net, train_iter, test_iter, batch_size,  
7                     optimizer, device, num_epochs)  
8 Out [4]:  
9     training on  cuda  
10    epoch 1, loss 1.0072, train acc 0.788, test acc 0.827, time 6.7 sec  
11    epoch 2, loss 0.4596, train acc 0.862, test acc 0.836, time 6.5 sec  
12    epoch 3, loss 0.3726, train acc 0.876, test acc 0.852, time 6.5 sec  
13    epoch 4, loss 0.3339, train acc 0.886, test acc 0.866, time 6.6 sec  
14    epoch 5, loss 0.3132, train acc 0.890, test acc 0.855, time 6.3 sec
```

最后我们查看第一个批量归一化层学习到的拉伸参数 `gamma` 和偏移参数 `beta`。

```
1 In [5]:  
2     net[1].gamma.view((-1,)), net[1].beta.view((-1,))  
3 Out [5]:  
4     (tensor([1.1173, 0.9368, 0.9685, 1.1043, 1.0498, 0.9036],  
5             device='cuda:0', grad_fn=<ViewBackward>),  
6     tensor([-0.2380, 0.0031, -0.4098, 0.3822, -0.6539, -0.1376],  
7             device='cuda:0', grad_fn=<ViewBackward>))
```

#### 4.10.4 简洁实现

与我们刚刚自己定义的 `BatchNorm` 类相比，Pytorch中 `nn` 模块定义的 `BatchNorm1d` 和 `BatchNorm2d` 类使用起来更加简单，二者分别用于全连接层和卷积层，都需要指定输入的 `num_features` 参数值。下面我们用PyTorch实现使用批量归一化的LeNet。

```
1 In [6]:  
2     net = nn.Sequential(  
3         # in_channels, out_channels, kernel_size  
4         nn.Conv2d(1, 6, 5),  
5         nn.BatchNorm2d(6),  
6         nn.Sigmoid(),  
7         # kernel_size, stride  
8         nn.MaxPool2d(2, 2),  
9         nn.Conv2d(6, 16, 5),  
10        nn.BatchNorm2d(16),  
11        nn.Sigmoid(),  
12        nn.MaxPool2d(2, 2),  
13        d2l.FlattenLayer(),  
14        nn.Linear(16*4*4, 120),  
15        nn.BatchNorm1d(120),  
16        nn.Sigmoid(),  
17        nn.Linear(120, 84),  
18        nn.BatchNorm1d(84),  
19        nn.Sigmoid(),  
20        nn.Linear(84, 10)  
21    )
```

使用同样的超参数进行训练。

```
1 In [7]:  
2     batch_size = 256  
3     train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)  
4     lr, num_epochs = 0.001, 5  
5     optimizer = torch.optim.Adam(net.parameters(), lr=lr)  
6     d2l.train_ch5(net, train_iter, test_iter, batch_size,  
7                     optimizer, device, num_epochs)  
8 Out [7]:  
9     training on  cuda  
10    epoch 1, loss 1.3550, train acc 0.765, test acc 0.806, time 5.2 sec  
11    epoch 2, loss 0.5887, train acc 0.857, test acc 0.760, time 5.9 sec  
12    epoch 3, loss 0.4131, train acc 0.875, test acc 0.810, time 6.6 sec  
13    epoch 4, loss 0.3566, train acc 0.883, test acc 0.837, time 6.0 sec  
14    epoch 5, loss 0.3254, train acc 0.891, test acc 0.821, time 5.4 sec
```

## 小结:

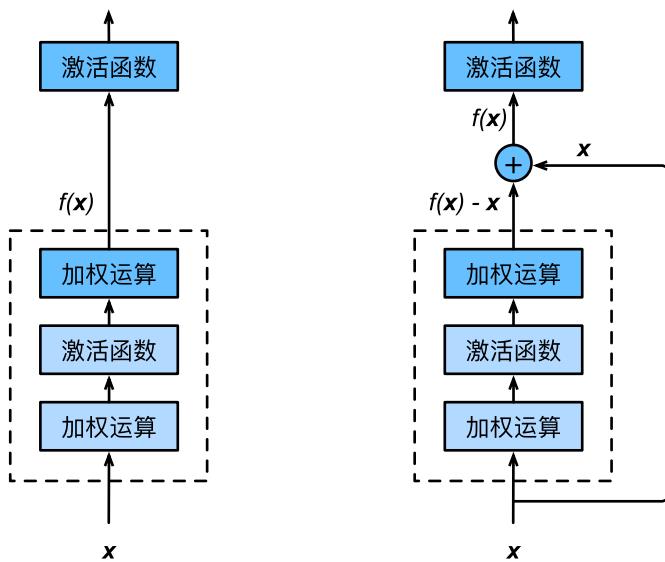
- 在模型训练时，批量归一化利用小批量上的均值和标准差，不断调整神经网络的中间输出，从而使整个神经网络在各层的中间输出的数值更稳定。
- 对全连接层和卷积层做批量归一化的方法稍有不同。
- 批量归一化层和丢弃层一样，在训练模式和预测模式的计算结果是不一样的。
- PyTorch提供了BatchNorm类方便使用。

## 4.11 残差网络 (ResNet)

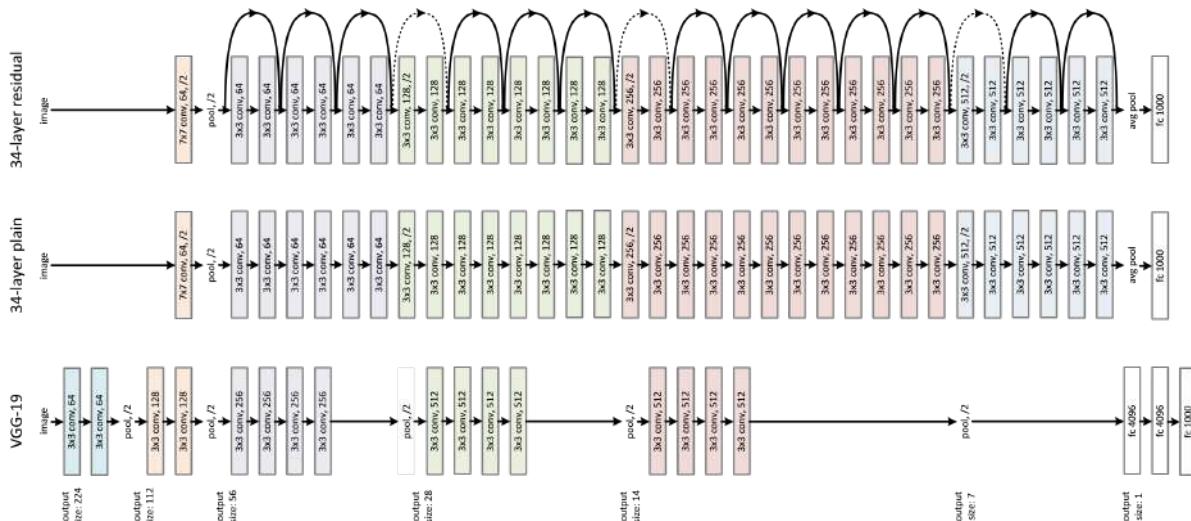
让我们先思考一个问题：对神经网络模型添加新的层，充分训练后的模型是否只可能更有效地降低训练误差？理论上，原模型解的空间只是新模型解的空间的子空间。也就是说，如果我们能将新添加的层训练成恒等映射  $f(x) = x$ ，新模型和原模型将同样有效。由于新模型可能得出更优的解来拟合训练数据集，因此添加层似乎更容易降低训练误差。然而在实践中，添加过多的层后训练误差往往不降反升。即使利用批量归一化带来的数值稳定性使训练深层模型更加容易，该问题仍然存在。针对这一问题，何恺明等人提出了残差网络（ResNet）。它在2015年的ImageNet图像识别挑战赛夺魁，并深刻影响了后来的深度神经网络的设计。

### 4.11.1 残差块

让我们聚焦于神经网络局部。如下图所示，设输入为  $x$ 。假设我们希望学出的理想映射为  $f(x)$ ，从而作为下图中激活函数的输入。左图虚线框中的部分需要直接拟合出该映射  $f(x)$ ，而右图虚线框中的部分则需要拟合出有关恒等映射的残差映射  $f(x) - x$ 。残差映射在实际中往往更容易优化。以本节开头提到的恒等映射作为我们希望学出的理想映射  $f(x)$ 。我们只需将下图内右图虚线框内上方的加权运算（如仿射）的权重和偏差参数学成0，那么  $f(x)$  即为恒等映射（相当于虚线框内的结果为0，此时  $f(x) = x$ ）。实际上，当理想映射  $f(x)$  极接近于恒等映射时，残差映射也易于捕捉恒等映射的细微波动。下图内的右图也是ResNet的基础块，即残差块（residual block）。在残差块中，输入可通过跨层的数据线路更快地向前传播。



设输入为 $x$ 。假设图中最上方激活函数输入的理想映射为 $f(x)$ 。左图虚线框中的部分需要直接拟合出该映射 $f(x)$ ，而右图虚线框中的部分需要拟合出有关恒等映射的残差映射 $f(x)-x$



ResNet沿用了VGG全 $3 \times 3$ 卷积层的设计。残差块里首先有2个有相同输出通道数的 $3 \times 3$ 卷积层。每个卷积层后接一个批量归一化层和ReLU激活函数。然后我们将输入跳过这两个卷积运算后直接加在最后的ReLU激活函数前。这样的设计要求两个卷积层的输出与输入形状一样，从而可以相加。**如果想改变通道数，就需要引入一个额外的 $1 \times 1$ 卷积层来将输入变换成为需要的形状后再做相加运算。**

残差块的实现如下。它可以设定输出通道数、是否使用额外的 $1 \times 1$ 卷积层来修改通道数以及卷积层的步幅。

```

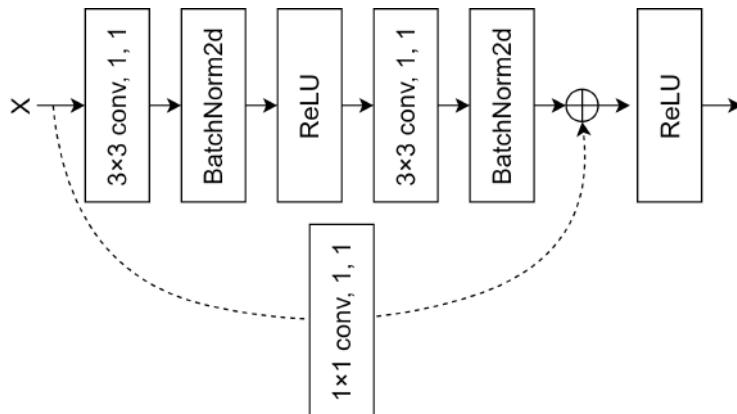
1 In [1]::
2     import torch.nn.functional as F
3     device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
4     class Residual(nn.Module):
5         # 输入通道数、输出通道数、是否使用1x1卷积核、步长
6         def __init__(self, in_channels, out_channels,
7                      use_1x1conv=False, stride=1):
8             super(Residual, self).__init__()
9             # 3x3搭配1步长，特征图大小不变
10            self.conv1 = nn.Conv2d(in_channels, out_channels,
11                                kernel_size=3, padding=1, stride=stride)
12            self.conv2 = nn.Conv2d(out_channels, out_channels,
13                                kernel_size=3, padding=1)
14            if use_1x1conv:
15                self.conv3 = nn.Conv2d(in_channels, out_channels,

```

```

16                                     kernel_size=1, stride=stride)
17
18     else:
19         self.conv3 = None
20
21     self.bn1 = nn.BatchNorm2d(out_channels)
22     self.bn2 = nn.BatchNorm2d(out_channels)
23
24     def forward(self, x):
25
26         Y = F.relu(self.bn1(self.conv1(x)))
27         Y = self.bn2(self.conv2(Y))
28
29         if self.conv3:
30             X = self.conv3(x)
31
32         return F.relu(Y+X)

```



下面我们来查看输入和输出形状一致的情况。

```

1 In [2]:
2     blk = Residual(3, 3)
3     X = torch.rand((4, 3, 6, 6))
4     blk(X).shape
5 Out [2]:
6     torch.Size([4, 3, 6, 6])

```

我们也可以在增加输出通道数的同时减半输出的高和宽。

```

1 In [3]:
2     blk = Residual(3, 6, use_1x1conv=True, stride=2)
3     X = torch.rand((4, 3, 6, 6))
4     blk(X).shape
5 Out [3]:
6     torch.Size([4, 6, 3, 3])

```

## 4.11.2 ResNet模型

ResNet的前两层跟之前介绍的GoogLeNet中的一样：在输出通道数为64、步幅为2的 $7 \times 7$ 卷积层后接步幅为2的 $3 \times 3$ 的最大池化层。不同之处在于ResNet每个卷积层后增加的批量归一化层。

```

1 In [4]:
2     net = nn.Sequential(
3         nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3),
4         nn.BatchNorm2d(64),
5         nn.ReLU(),
6         nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
7     )

```

GoogLeNet在后面接了4个由Inception块组成的模块。ResNet则使用4个由残差块组成的模块，每个模块使用若干个同样输出通道数的残差块。第一个模块的通道数同输入通道数一致。由于之前已经使用了步幅为2的最大池化层，所以无须减小高和宽。之后的每个模块在第一个残差块里将上一个模块的通道数翻倍，并将高和宽减半。

下面我们来实现这个模块。注意，这里对第一个模块做了特别处理。

```
1 In [5]:  
2     def resnet_block(in_channels, out_channels,  
3                         num_residuals, first_block=False):  
4         if first_block:  
5             # 第一个模块的通道数同输入通道数一致  
6             assert in_channels == out_channels  
7             blk = []  
8             for i in range(num_residuals):  
9                 if i==0 and not first_block:  
10                     blk.append(Residual(in_channels, out_channels,  
11                                         use_1x1conv=True, stride=2))  
12             else:  
13                 blk.append(Residual(out_channels, out_channels))  
14         return nn.Sequential(*blk)
```

接着我们为ResNet加入所有残差块。这里每个模块使用两个残差块。

```
1 In [6]:  
2     net.add_module('resnet_block1', resnet_block(64, 64, 2,  
3                                                 first_block=True))  
4     net.add_module('resnet_block2', resnet_block(64, 128, 2))  
5     net.add_module('resnet_block3', resnet_block(128, 256, 2))  
6     net.add_module('resnet_block4', resnet_block(256, 512, 2))
```

最后，与GoogLeNet一样，加入全局平均池化层后接上全连接层输出。

```
1 In [7]:  
2     # GlobalAvgPool2d的输出: (Batch, 512, 1, 1)  
3     net.add_module('global_avg_pool', d2l.GlobalAvgPool2d())  
4     net.add_module('fc', nn.Sequential(d2l.FlattenLayer(),  
5                                         nn.Linear(512, 10)))
```

这里每个模块里有4个卷积层（不计算 $1 \times 1$ 卷积层），加上最开始的卷积层和最后的全连接层，共计18层。这个模型通常也被称为ResNet-18。通过配置不同的通道数和模块里的残差块数可以得到不同的ResNet模型，例如更深的含152层的ResNet-152。虽然ResNet的主体架构跟GoogLeNet的类似，但ResNet结构更简单，修改也更方便。这些因素都导致了ResNet迅速被广泛使用。

在训练ResNet之前，我们来观察一下输入形状在ResNet不同模块之间的变化。

```
1 In [8]:  
2     X = torch.rand((1, 1, 224, 224))  
3     for name, layer in net.named_children():  
4         X = layer(X)  
5         print(name, ' output shape:\t', X.shape)  
6 Out [8]:  
7     0  output shape:      torch.Size([1, 64, 112, 112])  
8     1  output shape:      torch.Size([1, 64, 112, 112])  
9     2  output shape:      torch.Size([1, 64, 112, 112])
```

```

10      3 output shape:      torch.Size([1, 64, 56, 56])
11      resnet_block1 output shape:      torch.Size([1, 64, 56, 56])
12      resnet_block2 output shape:      torch.Size([1, 128, 28, 28])
13      resnet_block3 output shape:      torch.Size([1, 256, 14, 14])
14      resnet_block4 output shape:      torch.Size([1, 512, 7, 7])
15      global_avg_pool output shape:   torch.Size([1, 512, 1, 1])
16      fc    output shape:      torch.Size([1, 10])

```

### 4.11.3 训练模型

下面我们在Fashion-MNIST数据集上训练ResNet。

```

1 In [9]:
2     batch_size = 256
3     train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
4     lr, num_epochs = 0.001, 5
5     optimizer = torch.optim.Adam(net.parameters(), lr=lr)
6     d2l.train_ch5(net, train_iter, test_iter, batch_size,
7                   optimizer, device, num_epochs)
8 Out [9]:
9     training on cuda
10    epoch 1, loss 0.4296, train acc 0.843, test acc 0.848, time 10.8 sec
11    epoch 2, loss 0.2982, train acc 0.890, test acc 0.886, time 10.7 sec
12    epoch 3, loss 0.2583, train acc 0.904, test acc 0.873, time 10.8 sec
13    epoch 4, loss 0.2299, train acc 0.915, test acc 0.899, time 10.9 sec
14    epoch 5, loss 0.2084, train acc 0.923, test acc 0.903, time 10.9 sec

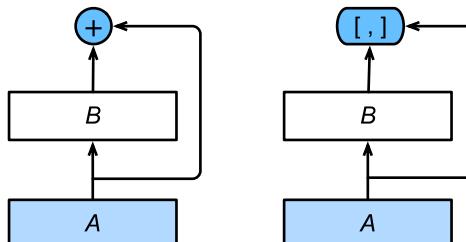
```

#### 小结:

- 残差块通过跨层的数据通道从而能够训练出有效的深度神经网络。
- ResNet深刻影响了后来的深度神经网络的设计。

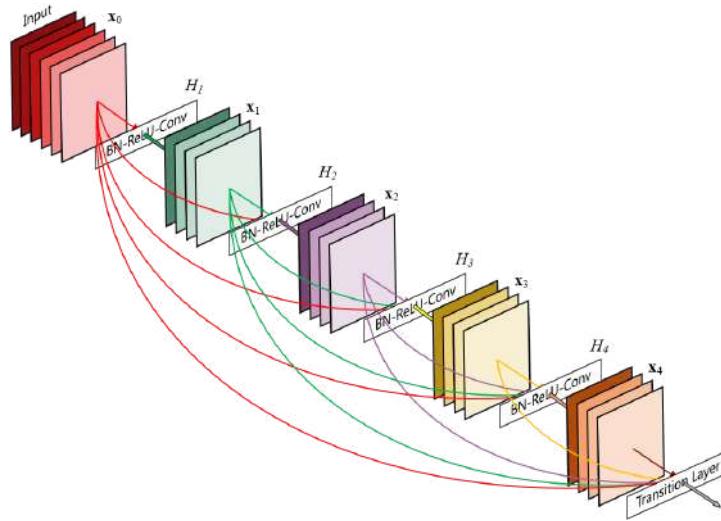
### 4.12 稠密连接网络 (DenseNet)

ResNet中的跨层连接设计引申出了数个后续工作。本节我们介绍其中的一个：稠密连接网络 (DenseNet)。它与ResNet的主要区别如下图所示。



上图中将部分前后相邻的运算抽象为模块 $A$ 和模块 $B$ 。与ResNet的主要区别在于，DenseNet里模块 $B$ 的输出不是像ResNet那样和模块 $A$ 的输出相加，而是在通道维上连接。这样模块 $A$ 的输出可以直接传入模块 $B$ 后面的层。在这个设计里，模块 $A$ 直接跟模块 $B$ 后面的所有层连接在了一起。这也是它被称为“稠密连接”的原因。

DenseNet的主要构建模块是稠密块 (dense block) 和过渡层 (transition layer)。前者定义了输入和输出是如何连结的，后者则用来控制通道数，使之不过大。



### 4.12.1 稠密块

DenseNet使用了ResNet改良版的“批量归一化、激活和卷积”结构，我们首先在 `conv_block` 函数里实现这个结构。

```

1 In [1]:
2     def conv_block(in_channels, out_channels):
3         blk = nn.Sequential(
4             nn.BatchNorm2d(in_channels),
5             nn.ReLU(),
6             nn.Conv2d(in_channels, out_channels,
7                     kernel_size=3, padding=1)
8         )
9         return blk

```

稠密块由多个 `conv_block` 组成，每块使用相同的输出通道数。但在前向计算时，我们将每块的输入和输出在通道维上连结。

```

1 In [2]:
2     class DenseBlock(nn.Module):
3         def __init__(self, num_convs, in_channels, out_channels):
4             super(DenseBlock, self).__init__()
5             net = []
6             for i in range(num_convs):
7                 in_c = in_channels + i*out_channels
8                 net.append(conv_block(in_c, out_channels))
9             self.net = nn.ModuleList(net)
10            # 计算输出通道数
11            self.out_channels = in_channels + num_convs*out_channels
12        def forward(self, x):
13            for blk in self.net:
14                Y = blk(x)
15                # 在通道维上将输入和输出连结
16                x = torch.cat((x, Y), dim=1)
17            return x

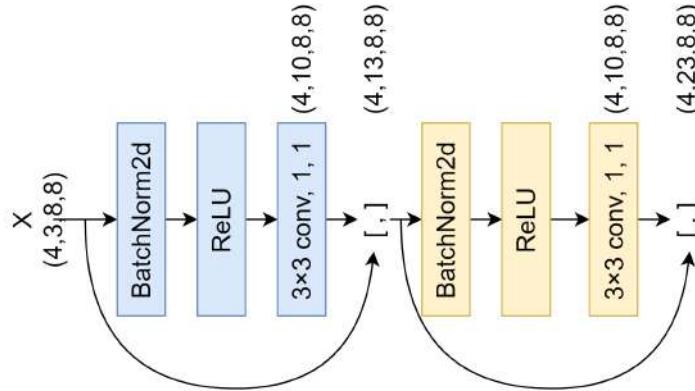
```

在下面的例子中，我们定义一个有2个输出通道数为10的卷积块。使用通道数为3的输入时，我们会得到通道数为 $3 + 2 \times 10 = 23$ 的输出。卷积块的通道数控制了输出通道数相对于输入通道数的增长，因此也被称为**增长率** (growth rate)。

```

1 In [3]:
2     blk = DenseBlock(2, 3, 10)
3     x = torch.rand(4, 3, 8, 8)
4     Y = blk(x)
5     Y.shape
6 Out [3]:
7     torch.size([4, 23, 8, 8])

```



## 4.12.2 过渡层

由于每个稠密块都会带来通道数的增加，使用过多则会带来过于复杂的模型。过渡层用来控制模型复杂度。**它通过 $1 \times 1$ 卷积层来减小通道数，并使用步幅为2的平均池化层减半高和宽，从而进一步降低模型复杂度。**

```

1 In [4]:
2     def transition_block(in_channels, out_channels):
3         blk = nn.Sequential(
4             nn.BatchNorm2d(in_channels),
5             nn.ReLU(),
6             nn.Conv2d(in_channels, out_channels, kernel_size=1),
7             nn.AvgPool2d(kernel_size=2, stride=2)
8         )
9         return blk

```

对上一个例子中稠密块的输出使用通道数为10的过渡层。此时输出的通道数减为10，高和宽均减半。

```

1 In [5]:
2     blk = transition_block(23, 10)
3     blk(Y).shape
4 Out [5]:
5     torch.size([4, 10, 4, 4])

```

## 4.12.3 DenseNet模型

我们来构造DenseNet模型。DenseNet首先使用同ResNet一样的单卷积层和最大池化层。

```
1 In [6]:  
2     net = nn.Sequential(  
3         nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3),  
4         nn.BatchNorm2d(64),  
5         nn.ReLU(),  
6         nn.MaxPool2d(kernel_size=3, stride=2, padding=1)  
7     )
```

类似于ResNet接下来使用的4个残差块，DenseNet使用的是4个稠密块。同ResNet一样，我们可以设置每个稠密块使用多少个卷积层。这里我们设成4，从而与上一节的ResNet-18保持一致。稠密块里的卷积层通道数（即增长率）设为32，所以每个稠密块将增加128个通道。

ResNet里通过步幅为2的残差块在每个模块之间减小高和宽。这里我们则使用过渡层来减半高和宽，并减半通道数。

```
1 In [7]:  
2     # num_channels为当前的通道数  
3     num_channels, growth_rate = 64, 32  
4     num_convs_in_dense_blocks = [4, 4, 4, 4]  
5     for i, num_convs in enumerate(num_convs_in_dense_blocks):  
6         DB = DenseBlock(num_convs, num_channels, growth_rate)  
7         net.add_module('DenseBlock_%d' % i, DB)  
8         # 上一个稠密块的输出通道数  
9         num_channels = DB.out_channels  
10        # 在稠密块之间加入通道数减半的过渡层  
11        if i != len(num_convs_in_dense_blocks)-1:  
12            net.add_module('transition_block_%d' % i,  
13                            transition_block(num_channels, num_channels//2))  
14            num_channels = num_channels // 2
```

同ResNet一样，最后接上全局池化层和全连接层来输出。

```
1 In [8]:  
2     net.add_module('BN', nn.BatchNorm2d(num_channels))  
3     net.add_module('relu', nn.ReLU())  
4     # GlobalAvgPool2d的输出: (Batch, num_channels, 1, 1)  
5     net.add_module('global_avg_pool', d2l.GlobalAvgPool2d())  
6     net.add_module('fc', nn.Sequential(  
7         d2l.FlattenLayer(),  
8         nn.Linear(num_channels, 10)  
9     ))
```

我们尝试打印每个子模块的输出维度确保网络无误：

```
1 In [9]:  
2     x = torch.rand((1, 1, 96, 96))  
3     for name, layer in net.named_children():  
4         x = layer(x)  
5         print(name, ' output shape:\t', x.shape)  
6 Out [9]:  
7     0  output shape:      torch.Size([1, 64, 48, 48])  
8     1  output shape:      torch.Size([1, 64, 48, 48])  
9     2  output shape:      torch.Size([1, 64, 48, 48])  
10    3  output shape:      torch.Size([1, 64, 24, 24])  
11    DenseBlock_0  output shape:  torch.Size([1, 192, 24, 24])
```

```
12     transition_block_0  output shape:  torch.Size([1, 96, 12, 12])
13     DenseBlock_1  output shape:  torch.Size([1, 224, 12, 12])
14     transition_block_1  output shape:  torch.Size([1, 112, 6, 6])
15     DenseBlock_2  output shape:  torch.Size([1, 240, 6, 6])
16     transition_block_2  output shape:  torch.Size([1, 120, 3, 3])
17     DenseBlock_3  output shape:  torch.Size([1, 248, 3, 3])
18     BN  output shape:  torch.Size([1, 248, 3, 3])
19     relu  output shape:  torch.Size([1, 248, 3, 3])
20     global_avg_pool  output shape:  torch.Size([1, 248, 1, 1])
21     fc  output shape:  torch.Size([1, 10])
```

## 4.12.4 训练模型

由于这里使用了比较深的网络，本节里我们将输入高和宽从224降到96来简化计算。

```
1 In [10]:
2     device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
3     batch_size = 256
4     train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size,
5                                     resize=96)
6     lr, num_epochs = 0.001, 5
7     optimizer = torch.optim.Adam(net.parameters(), lr=lr)
8     d2l.train_ch5(net, train_iter, test_iter, batch_size, optimizer,
9                     device, num_epochs)
10 Out [10]:
11     training on  cuda
12     epoch 1, loss 0.5218, train acc 0.828, test acc 0.833, time 12.7 sec
13     epoch 2, loss 0.3087, train acc 0.887, test acc 0.869, time 12.4 sec
14     epoch 3, loss 0.2656, train acc 0.902, test acc 0.890, time 12.5 sec
15     epoch 4, loss 0.2360, train acc 0.914, test acc 0.894, time 12.2 sec
16     epoch 5, loss 0.2186, train acc 0.919, test acc 0.871, time 12.3 sec
```

### 小结：

- 在跨层连接上，不同于ResNet中将输入与输出相加，DenseNet在通道维上连结输入与输出。
- DenseNet的主要构建模块是稠密块和过渡层。

## 4.13 本章附录

### ☆ `torch.nn.Parameter()` 的用法

`torch.nn.Parameter` 是继承自 `torch.Tensor` 的子类，其主要作用是作为 `nn.Module` 中的可训练参数使用。它与 `torch.Tensor` 的区别就是 `nn.Parameter` 会自动被认为是module的可训练参数，即加入到 `parameter()` 这个迭代器中去；而module中非 `nn.Parameter()` 的普通tensor是不在 `parameter` 中的。注意到，`nn.Parameter` 的对象的 `requires_grad` 属性的默认值是 `True`，即是可被训练的，这与 `torch.Tensor` 对象的默认值相反。在 `nn.Module` 类中，pytorch也是使用 `nn.Parameter` 来对每一个module的参数进行初始化的。

### ☆ `torch.cat()` 的用法

`torch.cat(tensors, dim=0, out=None) → Tensor`。将保存在 `tensors` 中的多个张量拼接在一起，`cat` 是 `concatenates` 的意思。使用 `torch.cat((A,B),dim)` 时，除拼接维数 `dim` 数值可不同外其余维数数值需相同（或为空），方能对齐。

```

1 A = torch.rand((2, 3))
2 A
3 >>> tensor([[0.0607, 0.4536, 0.6023],
4             [0.8500, 0.6533, 0.9541]])
5 torch.cat((A, A+1), dim=0)
6 >>> tensor([[0.0607, 0.4536, 0.6023],
7             [0.8500, 0.6533, 0.9541],
8             [1.0607, 1.4536, 1.6023],
9             [1.8500, 1.6533, 1.9541]])
10 torch.cat((A, A+1), dim=1)
11 >>> tensor([[0.0607, 0.4536, 0.6023, 1.0607, 1.4536, 1.6023],
12             [0.8500, 0.6533, 0.9541, 1.8500, 1.6533, 1.9541]])

```

### ☆ `torchvision.transforms.Resize()` 的用法

`torchvision.transforms.Resize(size, interpolation=2)`。将PIL图片转换为指定的形状。其中 `size` 参数可以为序列或 `int` 型数值，若 `size` 的取值形如 `(h, w)`，那么输出图像为维度就是这个值。如果 `size` 参数为一个 `int` 型数值，那么原图中较小的一边缩放到这个数值。例如，若 `height > width`，那么图片转换后的形状为 `(size * height / width, size)`，也即成比例缩放。`interpolation` 参数指定所用的插值算法，默认为 `PIL.Image.BILINEAR`。

### ☆ 序列解压

序列前添加 `*` 相当于解压，与 `zip` 的功能相反。

```

1 A = [1, 2, 3]
2 print(*A)
3 >>> 1 2 3

```

本章代码详见GitHub链接[wzy6642](#)。

## 第5章 循环神经网络

与之前介绍的多层感知机和能有效处理空间信息的卷积神经网络不同，[循环神经网络](#)是为更好地处理时序信息而设计的。它引入状态变量来存储过去的信息，并用其与当前的输入共同决定当前的输出。

循环神经网络常用于处理序列数据，如一段文字或声音、购物或观影的顺序，甚至是图像中的一行或一列像素。因此，循环神经网络有着极为广泛的实际应用，如语言模型、文本分类、机器翻译、语音识别、图像分析、手写识别和推荐系统。

因为本章中的应用是基于语言模型的，所以我们将先介绍语言模型的基本概念，并由此激发循环神经网络的设计灵感。接着，我们将描述循环神经网络中的梯度计算方法，从而探究循环神经网络训练可能存在的问题。对于其中的部分问题，我们可以使用本章稍后介绍的含门控的循环神经网络来解决。最后，我们将拓展循环神经网络的架构。

### 5.1 语言模型

语言模型（language model）是自然语言处理的重要技术。自然语言处理中最常见的数据是文本数据。我们可以把一段自然语言文本看作一段离散的时间序列。假设一段长度为  $T$  的文本中的词依次为  $w_1, w_2, \dots, w_T$ ，那么在离散的时间序列中， $w_t$  ( $1 \leq t \leq T$ ) 可看作在时间步（time step） $t$  的输出或标签。给定一个长度为  $T$  的词的序列  $w_1, w_2, \dots, w_T$ ，语言模型将计算该序列的概率：

$$P(w_1, w_2, \dots, w_T)$$

语言模型可用于提升语音识别和机器翻译的性能。例如，在语音识别中，给定一段“厨房里食油用完了”的语音，有可能会输出“厨房里食油用完了”和“厨房里石油用完了”这两个读音完全一样的文本序列。如果语言模型判断出前者的概率大于后者的概率，我们就可以根据相同读音的语音输出“厨房里食油用完了”的文本序列。在机器翻译中，如果对英文“you go first”逐词翻译成中文的话，可能得到“你走先”、“你先走”等排列方式的文本序列。如果语言模型判断出“你先走”的概率大于其他排列方式的文本序列的概率，我们就可以把“you go first”翻译成“你先走”。

## 5.1.1 语言模型的计算

既然语言模型很有用，那该如何计算它呢？假设序列 $w_1, w_2, \dots, w_T$ 中的每个词是**依次生成的**，我们有

$$P(w_1, w_2, \dots, w_T) = \prod_{t=1}^T P(w_t | w_1, \dots, w_{t-1})$$

例如，一段含有4个词的文本序列的概率

$$P(w_1, w_2, w_3, w_4) = P(w_1)P(w_2 | w_1)P(w_3 | w_1, w_2)P(w_4 | w_1, w_2, w_3)$$

为了计算语言模型，我们需要计算词的概率，以及一个词在给定前几个词的情况下条件概率，即语言模型参数。设训练数据集为一个大型文本语料库，如维基百科的所有条目。词的概率可以通过该词在训练数据集中的相对词频来计算。例如， $P(w_1)$ 可以计算为 $w_1$ 在训练数据集中的词频（词出现的次数）与训练数据集的总词数之比。因此，根据条件概率定义，一个词在给定前几个词的情况下条件概率也可以通过训练数据集中的相对词频计算。例如， $P(w_2 | w_1)$ 可以计算为 $w_1, w_2$ 两词相邻的频率与 $w_1$ 词频的比值，因为该比值即 $P(w_1, w_2)$ 与 $P(w_1)$ 之比；而 $P(w_3 | w_1, w_2)$ 同理可以计算为 $w_1, w_2$ 和 $w_3$ 三词相邻的频率与 $w_1$ 和 $w_2$ 两词相邻的频率的比值。以此类推。

## 5.1.2 n元语法

当序列长度增加时，计算和存储多个词共同出现的概率的复杂度会呈指数级增加。 $n$ 元语法通过马尔可夫假设（虽然并不一定成立）简化了语言模型的计算。**这里的马尔可夫假设是指一个词的出现只与前面n个词相关，即n阶马尔可夫链**（Markov chain of order  $n$ ）。如果 $n = 1$ ，那么有

$P(w_3 | w_1, w_2) = P(w_3 | w_2)$ 。如果基于 $n - 1$ 阶马尔可夫链，我们可以将语言模型改写为

$$P(w_1, w_2, \dots, w_T) \approx \prod_{t=1}^T P(w_t | w_{t-(n-1)}, \dots, w_{t-1})$$

以上也叫 **$n$ 元语法** ( $n$ -grams)。它是基于 $n - 1$ 阶马尔可夫链的概率语言模型。当 $n$ 分别为1、2和3时，我们将其分别称作一元语法 (unigram)、二元语法 (bigram) 和三元语法 (trigram)。例如，长度为4的序列 $w_1, w_2, w_3, w_4$ 在一元语法、二元语法和三元语法中的概率分别为

$$\begin{aligned} P(w_1, w_2, w_3, w_4) &= P(w_1)P(w_2)P(w_3)P(w_4) \\ P(w_1, w_2, w_3, w_4) &= P(w_1)P(w_2 | w_1)P(w_3 | w_1, w_2)P(w_4 | w_1, w_2, w_3) \\ P(w_1, w_2, w_3, w_4) &= P(w_1)P(w_2 | w_1)P(w_3 | w_1, w_2)P(w_4 | w_1, w_2, w_3) \end{aligned}$$

当 $n$ 较小时， $n$ 元语法往往并不准确。例如，在一元语法中，由三个词组成的句子“你走先”和“你先走”的概率是一样的。然而，当 $n$ 较大时， $n$ 元语法需要计算并存储大量的词频和多词相邻频率。

那么，有没有方法在语言模型中更好地平衡以上这两点呢？我们将在本章探究这样的方法。

**小结：**

- 语言模型是自然语言处理的重要技术。
- $N$ 元语法是基于 $n - 1$ 阶马尔可夫链的概率语言模型，其中 $n$ 权衡了计算复杂度和模型准确性。

## 5.2 循环神经网络

上一节介绍的 $n$ 元语法中，时间步 $t$ 的词 $w_t$ 基于前面所有词的条件概率只考虑了最近时间步的 $n - 1$ 个词。如果要考虑比 $t - (n - 1)$ 更早时间步的词对 $w_t$ 可能影响，我们需要增大 $n$ 。但这样模型参数的数量将随之呈指数级增长。

本节将介绍循环神经网络。它并非刚性地记忆所有固定长度的序列，而是通过隐藏状态来存储之前时间步的信息。首先我们回忆一下前面介绍过的多层感知机，然后描述如何添加隐藏状态来将它变成循环神经网络。

## 5.2.1 不含隐藏状态的神经网络

让我们考虑一个含单隐藏层的多层感知机。给定样本数为 $n$ 、输入个数（特征数或特征向量维度）为 $d$ 的小批量数据样本 $\mathbf{X} \in \mathbb{R}^{n \times d}$ 。设隐藏层的激活函数为 $\phi$ ，那么隐藏层的输出 $\mathbf{H} \in \mathbb{R}^{n \times h}$ 计算为

$$\mathbf{H} = \phi(\mathbf{X}\mathbf{W}_{xh} + \mathbf{b}_h)$$

其中隐藏层权重参数 $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$ ，隐藏层偏差参数 $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$ ， $h$ 为隐藏单元个数。上式相加的两项形状不同，因此将按照广播机制相加。把隐藏变量 $\mathbf{H}$ 作为输出层的输入，且设输出个数为 $q$ （如分类问题中的类别数），输出层的输出为

$$\mathbf{O} = \mathbf{H}\mathbf{W}_{hq} + \mathbf{b}_q$$

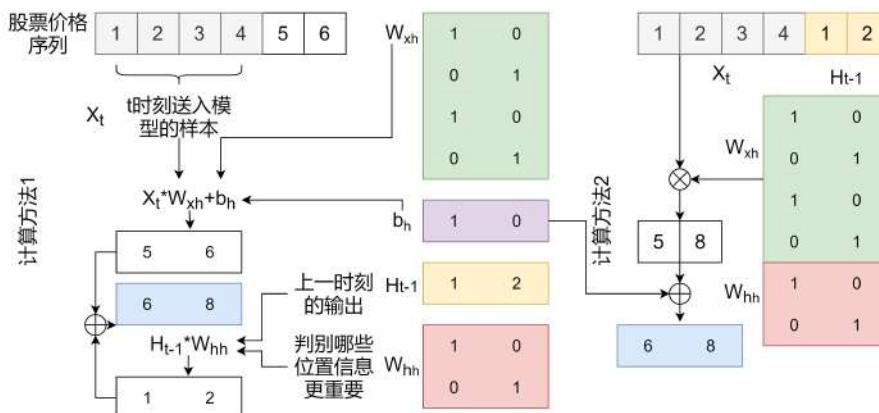
其中输出变量 $\mathbf{O} \in \mathbb{R}^{n \times q}$ ，输出层权重参数 $\mathbf{W}_{hq} \in \mathbb{R}^{h \times q}$ ，输出层偏差参数 $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$ 。如果是分类问题，我们可以使用softmax( $\mathbf{O}$ )来计算输出类别的概率分布。

## 5.2.2 含隐藏状态的循环神经网络

现在我们考虑输入数据存在时间相关性的情况。假设 $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ 是序列中时间步 $t$ 的小批量输入， $\mathbf{H}_t \in \mathbb{R}^{n \times h}$ 是该时间步的隐藏变量。与多层感知机不同的是，这里我们保存上一时间步的隐藏变量 $\mathbf{H}_{t-1}$ ，并引入一个新的权重参数 $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ ，该参数用来描述在当前时间步如何使用上一时间步的隐藏变量。具体来说，时间步 $t$ 的隐藏变量的计算由当前时间步的输入和上一时间步的隐藏变量共同决定：

$$\mathbf{H}_t = \phi(\mathbf{X}_t\mathbf{W}_{xh} + \mathbf{H}_{t-1}\mathbf{W}_{hh} + \mathbf{b}_h)$$

与多层感知机相比，我们在添加了 $\mathbf{H}_{t-1}\mathbf{W}_{hh}$ 一项。由上式中相邻时间步的隐藏变量 $\mathbf{H}_t$ 和 $\mathbf{H}_{t-1}$ 之间的关系可知，这里的隐藏变量能够捕捉截至当前时间步的序列的历史信息，就像是神经网络当前时间步的状态或记忆一样。因此，该隐藏变量也称为隐藏状态。由于隐藏状态在当前时间步的定义使用了上一时间步的隐藏状态，上式的计算是循环的。使用循环计算的网络即循环神经网络（recurrent neural network）。

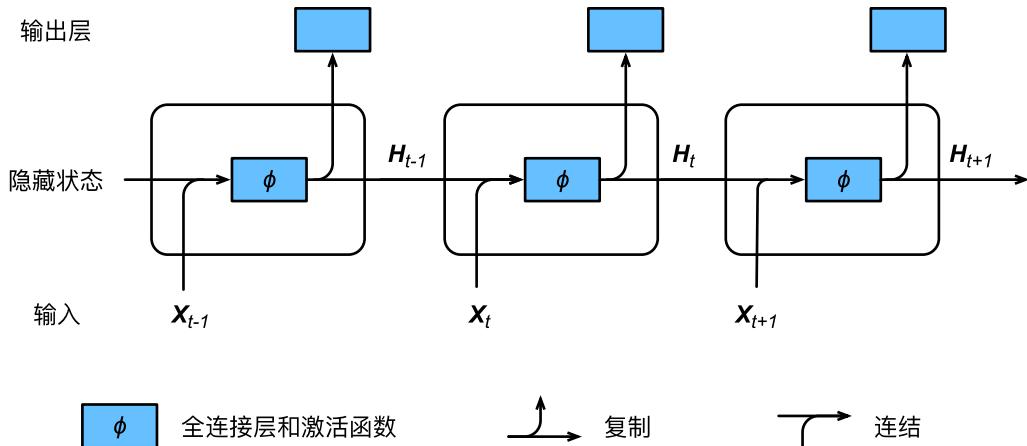


循环神经网络有很多不同的构造方法。含上式所定义的隐藏状态的循环神经网络是极为常见的一种。若无特别说明，本章中的循环神经网络均基于上式中隐藏状态的循环计算。在时间步 $t$ ，输出层的输出和多层感知机中的计算类似：

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hq} + \mathbf{b}_q$$

循环神经网络的参数包括隐藏层的权重  $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$ 、 $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$  和偏差  $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$ ，以及输出层的权重  $\mathbf{W}_{hq} \in \mathbb{R}^{h \times q}$  和偏差  $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$ 。值得一提的是，即便在不同时间步，循环神经网络也始终使用这些模型参数。因此，循环神经网络模型参数的数量不随时间步的增加而增长。

下图展示了循环神经网络在3个相邻时间步的计算逻辑。在时间步  $t$ ，隐藏状态的计算可以看成是将输入  $\mathbf{X}_t$  和前一时间步隐藏状态  $\mathbf{H}_{t-1}$  连结后输入一个激活函数为  $\phi$  的全连接层。该全连接层的输出就是当前时间步的隐藏状态  $\mathbf{H}_t$ ，且模型参数为  $\mathbf{W}_{xh}$  与  $\mathbf{W}_{hh}$  的连结，偏差为  $\mathbf{b}_h$ 。当前时间步  $t$  的隐藏状态  $\mathbf{H}_t$  将参与下一个时间步  $t+1$  的隐藏状态  $\mathbf{H}_{t+1}$  的计算，并输入到当前时间步的全连接输出层。



我们刚刚提到，隐藏状态中  $\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh}$  的计算等价于  $\mathbf{X}_t$  与  $\mathbf{H}_{t-1}$  连结后的矩阵乘以  $\mathbf{W}_{xh}$  与  $\mathbf{W}_{hh}$  连结后的矩阵。接下来，我们用一个具体的例子来验证这一点。首先，我们构造矩阵  $\mathbf{x}$ 、 $\mathbf{w\_xh}$ 、 $\mathbf{H}$  和  $\mathbf{w\_hh}$ ，它们的形状分别为  $(3, 1)$ 、 $(1, 4)$ 、 $(3, 4)$  和  $(4, 4)$ 。将  $\mathbf{x}$  与  $\mathbf{w\_xh}$ 、 $\mathbf{H}$  与  $\mathbf{w\_hh}$  分别相乘，再把两个乘法运算的结果相加，得到形状为  $(3, 4)$  的矩阵。

```

1 In [1]:
2     import torch
3     x, w_xh = torch.randn(3, 1), torch.randn(1, 4)
4     H, w_hh = torch.randn(3, 4), torch.randn(4, 4)
5     torch.matmul(x, w_xh)+torch.matmul(H, w_hh)
6 Out [1]:
7     tensor([[ 0.7436,  0.3531, -2.6245, -1.3189],
8             [-1.0150,  1.1306, -1.1868, -2.0577],
9             [ 0.8640, -0.5475, -0.3046, -0.5207]])

```

将矩阵  $\mathbf{x}$  和  $\mathbf{H}$  按列（维度1）连结，连结后的矩阵形状为  $(3, 5)$ 。可见，连结后矩阵在维度1的长度为矩阵  $\mathbf{x}$  和  $\mathbf{H}$  在维度1的长度之和  $(1 + 4)$ 。然后，将矩阵  $\mathbf{w\_xh}$  和  $\mathbf{w\_hh}$  按行（维度0）连结，连结后的矩阵形状为  $(5, 4)$ 。最后将两个连结后的矩阵相乘，得到与上面代码输出相同的形状为  $(3, 4)$  的矩阵。

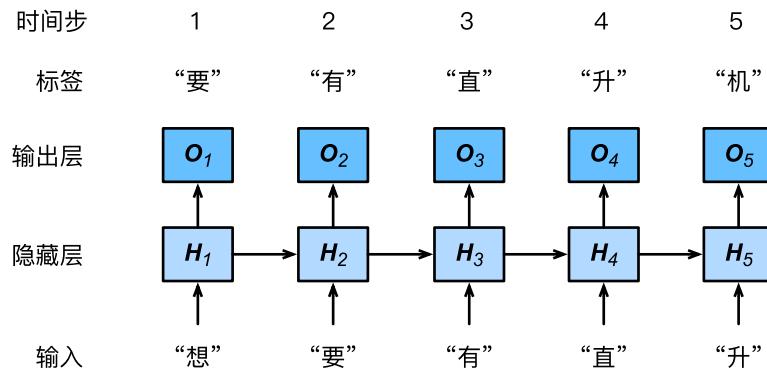
```

1 In [2]:
2     torch.matmul(torch.cat((x, H), dim=1),
3                  torch.cat((w_xh, w_hh), dim=0))
4 Out [2]:
5     tensor([[ 0.7436,  0.3531, -2.6245, -1.3189],
6             [-1.0150,  1.1306, -1.1868, -2.0577],
7             [ 0.8640, -0.5475, -0.3046, -0.5207]])

```

### 5.2.3 应用：基于字符级循环神经网络的语言模型

最后我们介绍如何应用循环神经网络来构建一个语言模型。设小批量中样本数为1，文本序列为“想”“要”“有”“直”“升”“机”。下图演示了如何使用循环神经网络基于当前和过去的字符来预测下一个字符。在训练时，我们对每个时间步的输出层输出使用softmax运算，然后使用交叉熵损失函数来计算它与标签的误差。在下图中，由于隐藏层中隐藏状态的循环计算，时间步3的输出 $O_3$ 取决于文本序列“想”“要”“有”。由于训练数据中该序列的下一个词为“直”，时间步3的损失将取决于该时间步基于序列“想”“要”“有”生成下一个词的概率分布与该时间步的标签“直”。



因为每个输入词是一个字符，因此这个模型被称为字符级循环神经网络（character-level recurrent neural network）。因为不同字符的个数远小于不同词的个数（对于英文尤其如此），所以字符级循环神经网络的计算通常更加简单。在接下来的几节里，我们将介绍它的具体实现。

#### 小结：

- 使用循环计算的网络即循环神经网络。
- 循环神经网络的隐藏状态可以捕捉截至当前时间步的序列的历史信息。
- 循环神经网络模型参数的数量不随时间步的增加而增长。
- 可以基于字符级循环神经网络来创建语言模型。

## 5.3 语言模型数据集（歌词）

本节将介绍如何预处理一个语言模型数据集，并将其转换成字符级循环神经网络所需要的输入格式。为此，我们收集了周杰伦从第一张专辑《Jay》到第十张专辑《跨时代》中的歌词，并在后面几节里应用循环神经网络来训练一个语言模型。当模型训练好后，我们就可以用这个模型来创作歌词。

### 5.3.1 读取数据集

首先读取这个数据集，看看前40个字符是什么样的。

```
1 In [1]:  
2     import zipfile  
3     with zipfile.ZipFile('data/jaychou_lyrics.txt.zip') as zin:  
4         with zin.open('jaychou_lyrics.txt') as f:  
5             corpus_chars = f.read().decode('utf-8')  
6             corpus_chars[:40]  
7 Out [1]:  
8     '想要有直升机\n想要和你飞到宇宙去\n想要和你融化在一起\nn  
9     融化在宇宙里\nn我每天每天每'
```

这个数据集有6万多个字符。为了打印方便，我们把换行符替换成空格，然后仅使用前1万个字符来训练模型。

```
1 In [2]:  
2     # \r和\n均表示回车  
3     corpus_chars = corpus_chars.replace('\n', ' ').replace('\r', ' ')  
4     corpus_chars = corpus_chars[:10000]
```

## 5.3.2 建立字符索引

我们将每个字符映射成一个从0开始的连续整数，又称索引，来方便之后的数据处理。为了得到索引，我们将数据集里所有不同字符取出来，然后将其逐一映射到索引来构造词典。接着，打印 `vocab_size`，即词典中不同字符的个数，又称词典大小。

```
1 In [3]:  
2     idx_to_char = list(set(corpus_chars))  
3     char_to_idx = dict([(char, i) for i, char in enumerate(idx_to_char)])  
4     vocab_size = len(char_to_idx)  
5     vocab_size  
6 Out [3]:  
7     1027
```

之后，将训练数据集中每个字符转化为索引，并打印前20个字符及其对应的索引。

```
1 In [4]:  
2     corpus_indices = [char_to_idx[char] for char in corpus_chars]  
3     sample = corpus_indices[:20]  
4     print('chars:', ''.join([idx_to_char[idx] for idx in sample]))  
5     print('indices:', sample)  
6 Out [4]:  
7     chars: 想要有直升机 想要和你飞到宇宙去 想要和  
8     indices: [379, 473, 263, 727, 534, 542, 752, 379, 473, 368,  
9             765, 258, 806, 982, 54, 624, 752, 379, 473, 368]
```

我们将以上代码封装在 `d2lzh` 包里的 `load_data_jay_lyrics` 函数中，以方便后面章节调用。调用该函数后会依次得到 `corpus_indices`、`char_to_idx`、`idx_to_char` 和 `vocab_size` 这4个变量。

## 5.3.3 时序数据的采样

在训练中我们需要每次随机读取小批量样本和标签。与之前章节的实验数据不同的是，**时序数据的一个样本通常包含连续的字符**。假设时间步数为5，样本序列为5个字符，即“想”“要”“有”“直”“升”。**该样本的标签序列为这些字符分别在训练集中的下一个字符**，即“要”“有”“直”“升”“机”。我们有两种方式对时序数据进行采样，分别是随机采样和相邻采样。

### 1. 随机采样

下面的代码每次从数据里随机采样一个小批量。其中批量大小 `batch_size` 指每个小批量的样本数，`num_steps` 为每个样本所包含的时间步数。在随机采样中，每个样本是原始序列上任意截取的一段序列。相邻的两个随机小批量在原始序列上的位置不一定相毗邻。因此，我们无法用一个小批量最终时间步的隐藏状态来初始化下一个批量的隐藏状态。**在训练模型时，每次随机采样前都需要重新初始化隐藏状态。**

```
1 In [5]:  
2     import random  
3     def data_iter_random(corpus_indices, batch_size,
```

```

4                     num_steps, device=None):
5             # 减1是因为输出的索引x是相应输入的索引y加1
6             # 这里是计算一共生成多少个样本
7             num_examples = (len(corpus_indices) - 1) // num_steps
8             # 循环多少次可以遍历所有样本
9             epoch_size = num_examples // batch_size
10            # 每一条样本的索引
11            example_indices = list(range(num_examples))
12            # 随机打乱样本索引
13            random.shuffle(example_indices)
14            # 返回从pos开始的长为num_steps的序列
15            def _data(pos):
16                return corpus_indices[pos: pos+num_steps]
17            if device is None:
18                device = torch.device(
19                    'cuda' if torch.cuda.is_available() else 'cpu')
20            # 构建每一个epoch内的样本
21            for i in range(epoch_size):
22                # 每次读取batch_size个随机样本
23                i = i * batch_size
24                batch_indices = example_indices[i: i+batch_size]
25                X = [_data(j*num_steps) for j in batch_indices]
26                Y = [_data(j*num_steps+1) for j in batch_indices]
27                yield torch.tensor(X, dtype=torch.float32, device=device),
28                      torch.tensor(Y, dtype=torch.float32, device=device)

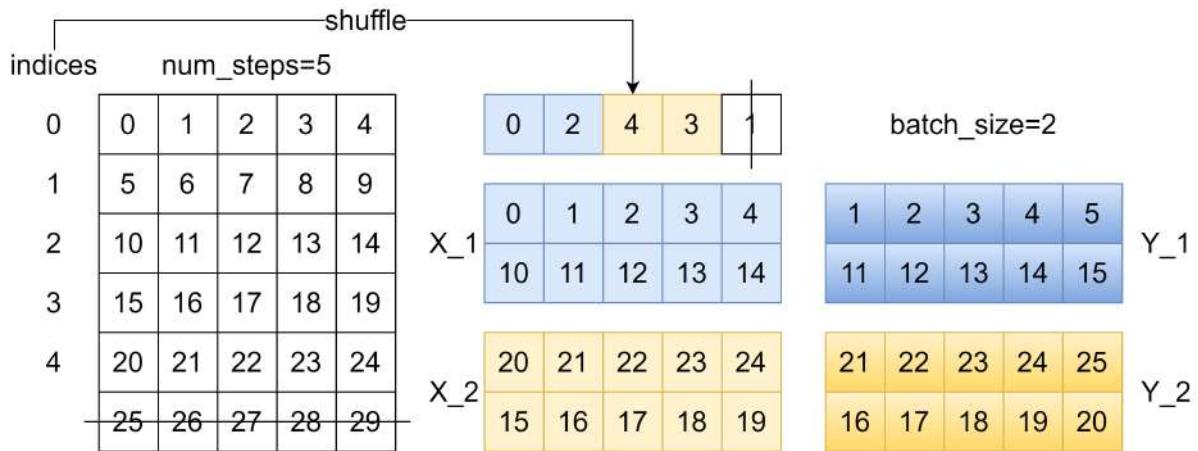
```

让我们输入一个从0到29的连续整数的人工序列。设批量大小和时间步数分别为2和6。打印随机采样每次读取的小批量样本的输入`X`和标签`Y`。可见，相邻的两个随机小批量在原始序列上的位置不一定相毗邻。

```

1 In [6]:
2     my_seq = list(range(30))
3     for X, Y in data_iter_random(my_seq, batch_size=2, num_steps=6):
4         print('X: ', X, '\nY:', Y, '\n')
5 Out [6]:
6     X: tensor([[12., 13., 14., 15., 16., 17.],
7                 [ 6.,  7.,  8.,  9., 10., 11.]], device='cuda:0')
8     Y: tensor([[13., 14., 15., 16., 17., 18.],
9                 [ 7.,  8.,  9., 10., 11., 12.]], device='cuda:0')
10
11    X: tensor([[18., 19., 20., 21., 22., 23.],
12                  [ 0.,  1.,  2.,  3.,  4.,  5.]], device='cuda:0')
13    Y: tensor([[19., 20., 21., 22., 23., 24.],
14                  [ 1.,  2.,  3.,  4.,  5.,  6.]], device='cuda:0')

```



## 2. 相邻采样

除对原始序列做随机采样之外，我们还可以令相邻的两个随机小批量在原始序列上的位置相毗邻。这时候，我们就可以用一个小批量最终时间步的隐藏状态来初始化下一个小批量的隐藏状态，从而使下一个批量的输出也取决于当前小批量的输入，并如此循环下去。这对实现循环神经网络造成了两方面影响：一方面，在训练模型时，我们只需在每一个迭代周期开始时初始化隐藏状态；另一方面，当多个相邻小批量通过传递隐藏状态串联起来时，模型参数的梯度计算将依赖所有串联起来的小批量序列。同一迭代周期中，随着迭代次数的增加，梯度的计算开销会越来越大。**为了使模型参数的梯度计算只依赖单次迭代读取的小批量序列，我们可以在每次读取小批量前将隐藏状态从计算图中分离出来。**我们将在下一节（循环神经网络的从零开始实现）的实现中了解这种处理方式。

```

1 In [7]:
2     def data_iter_consecutive(corpus_indices, batch_size,
3                               num_steps, device=None):
4         if device is None:
5             device = torch.device(
6                 'cuda' if torch.cuda.is_available() else 'cpu')
7         corpus_indices = torch.tensor(corpus_indices,
8                                         dtype=torch.float32, device=device)
9         # 原始数据长度
10        data_len = len(corpus_indices)
11        batch_len = data_len // batch_size
12        indices = corpus_indices[0: batch_size*batch_len].view(
13            batch_size, batch_len)
14        # 减1是因为输出的索引x是相应输入的索引y加1
15        epoch_size = (batch_len - 1) // num_steps
16        for i in range(epoch_size):
17            i = i * num_steps
18            x = indices[:, i: i + num_steps]
19            y = indices[:, i + 1: i + num_steps + 1]
20            yield x, y

```

同样的设置下，打印相邻采样每次读取的小批量样本的输入`x`和标签`y`。相邻的两个随机小批量在原始序列上的位置相毗邻。

```

1 In [8]:
2     for X, Y in data_iter_consecutive(my_seq, batch_size=2, num_steps=6):
3         print('X: ', X, '\nY:', Y, '\n')
4 Out [8]:
5     X: tensor([[ 0.,  1.,  2.,  3.,  4.,  5.],
6                 [15., 16., 17., 18., 19., 20.]], device='cuda:0')
7     Y: tensor([[ 1.,  2.,  3.,  4.,  5.,  6.],
8                 [16., 17., 18., 19., 20., 21.]], device='cuda:0')
9
10    X: tensor([[ 6.,  7.,  8.,  9., 10., 11.],
11                  [21., 22., 23., 24., 25., 26.]], device='cuda:0')
12    Y: tensor([[ 7.,  8.,  9., 10., 11., 12.],
13                  [22., 23., 24., 25., 26., 27.]], device='cuda:0')

```

	batch_len=15														
	num_steps=6						epoch_size=2								
batch_size=2	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29

小结:

- 时序数据采样方式包括随机采样和相邻采样。使用这两种方式的循环神经网络训练在实现上略有不同。

## 5.4 循环神经网络的从零开始实现

在本节中，我们将从零开始实现一个基于字符级循环神经网络的语言模型，并在周杰伦专辑歌词数据集上训练一个模型来进行歌词创作。首先，我们读取周杰伦专辑歌词数据集：

```

1 In [1]:
2     import d2lzh as d2l
3     device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
4     (corpus_indices, char_to_idx,
5      idx_to_char, vocab_size) = d2l.load_data_jay_lyrics()

```

### 5.4.1 one-hot向量

为了将词表示成向量输入到神经网络，一个简单的办法是使用one-hot向量。假设词典中不同字符的数量为 $N$ （即词典大小 `vocab_size`），每个字符已经同一个从0到 $N - 1$ 的连续整数值索引——对应。如果一个字符的索引是整数 $i$ ，那么我们创建一个全0的长为 $N$ 的向量，并将其位置为 $i$ 的元素设成1。该向量就是对原字符的one-hot向量。下面分别展示了索引为0和2的one-hot向量，向量长度等于词典大小。

```

1 In [2]:
2     import torch.nn.functional as F
3     F.one_hot(torch.tensor([0, 2]), vocab_size)
4 Out [2]:
5     tensor([[1, 0, 0, ..., 0, 0, 0],
6             [0, 0, 1, ..., 0, 0, 0]])

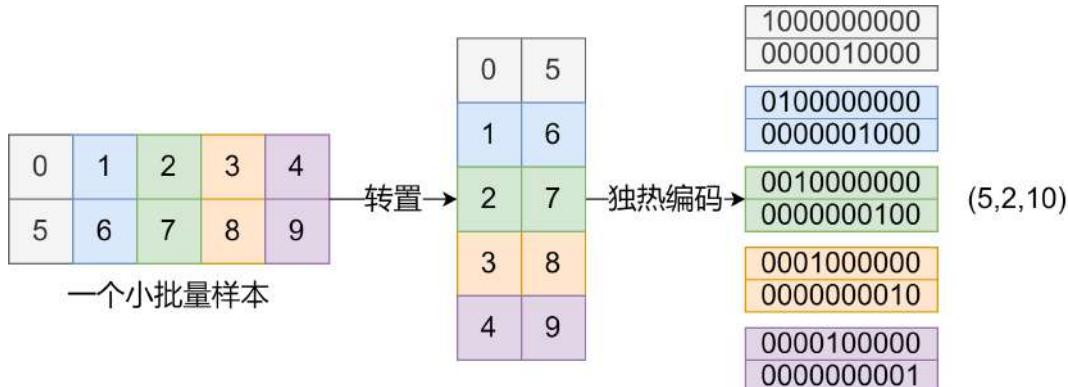
```

我们每次采样的小批量的形状是（批量大小，时间步数）。下面的函数将这样的小批量转换成数个可以输入进网络的形状为（批量大小，词典大小）的矩阵，矩阵个数等于时间步数。也就是说，时间步的输入为 $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ ，其中 $n$ 为批量大小， $d$ 为输入个数，即one-hot向量长度（词典大小）。

```

1 In [3]:
2     def to_onehot(x, size):
3         return F.one_hot(x.t(), size)
4     x = torch.arange(10).view(2, 5)
5     inputs = to_onehot(x, vocab_size)
6     len(inputs), inputs[0].shape
7 Out [3]:
8     (5, torch.Size([2, 1027]))

```



## 5.4.2 初始化模型参数

接下来，我们初始化模型参数（对照5.2.2节插图分析更易于理解）。隐藏单元个数 `num_hiddens` 是一个超参数。

```

1 In [4]:
2     num_inputs, num_hiddens, num_outputs = vocab_size, 256, vocab_size
3     print('will use', device)
4     import numpy as np
5     def get_params():
6         def _one(shape):
7             ts = torch.tensor(
8                 np.random.normal(0, 0.01, size=shape),
9                 device=device, dtype=torch.float32)
10            return torch.nn.Parameter(ts, requires_grad=True)
11        # 隐藏层参数
12        w_xh = _one((num_inputs, num_hiddens))
13        w_hh = _one((num_hiddens, num_hiddens))
14        b_h = torch.nn.Parameter(
15            torch.zeros(num_hiddens, device=device,
16                        requires_grad=True))
17        # 输出层参数
18        w_hq = _one((num_hiddens, num_outputs))
19        b_q = torch.nn.Parameter(
20            torch.zeros(num_outputs, device=device,
21                        requires_grad=True))
22        return torch.nn.ParameterList([w_xh, w_hh, b_h, w_hq, b_q])
23 Out [4]:
24     will use cuda

```

### 5.4.3 定义模型

我们根据循环神经网络的计算表达式实现该模型。首先定义 `init_rnn_state` 函数来返回初始化的隐藏状态。它返回由一个形状为（批量大小, 隐藏单元个数）的值为0的 Tensor 组成的元组。使用元组是为了更便于处理隐藏状态含有多个 Tensor 的情况。

```
1 In [5]:  
2     def init_rnn_state(batch_size, num_hiddens, device):  
3         return (torch.zeros((batch_size, num_hiddens), device=device), )
```

下面的 `rnn` 函数定义了在一个时间步里如何计算隐藏状态和输出。这里的激活函数使用了 `tanh` 函数。2.8节（多层感知机）中介绍过，当元素在实数域上均匀分布时，`tanh` 函数值的均值为0。

```
1 In [6]:  
2     def rnn(inputs, state, params):  
3         # inputs和outputs皆为num_steps个形状为(batch_size, vocab_size)的矩阵  
4         w_xh, w_hh, b_h, w_hq, b_q = params  
5         # 上一层传来的隐藏状态  
6         H, = state  
7         outputs = []  
8         for x in inputs:  
9             H = torch.tanh(torch.matmul(x.float(), w_xh) +  
10                         torch.matmul(H, w_hh) + b_h)  
11             Y = torch.matmul(H, w_hq) + b_q  
12             outputs.append(Y)  
13     return outputs, (H, )
```

做个简单的测试来观察输出结果的个数（时间步数），以及第一个时间步的输出层输出的形状和隐藏状态的形状。

```
1 In [7]:  
2     state = init_rnn_state(X.shape[0], num_hiddens, device)  
3     inputs = to_onehot(X.to(device), vocab_size)  
4     params = get_params()  
5     outputs, state_new = rnn(inputs, state, params)  
6     print(len(outputs), outputs[0].shape, state_new[0].shape)  
7 Out [7]:  
8     5 torch.size([2, 1027]) torch.size([2, 256])
```

### 5.4.4 定义预测函数

以下函数基于前缀 `prefix`（含有数个字符的字符串）来预测接下来的 `num_chars` 个字符。这个函数稍显复杂，其中我们将循环神经单元 `rnn` 设置成了函数参数，这样在后面小节介绍其他循环神经网络时能重复使用这个函数。

```
1 In [8]:  
2     def predict_rnn(prefix, num_chars, rnn, params,  
3                      init_rnn_state, num_hiddens, vocab_size,  
4                      device, idx_to_char, char_to_idx):  
5         # 初始化的隐藏状态  
6         state = init_rnn_state(1, num_hiddens, device)  
7         # 将输入的首字符传入到输出序列中  
8         output = [char_to_idx[prefix[0]]]  
9         for t in range(num_chars + len(prefix) - 1):
```

```

10     # 将上一时间步的输出作为当前时间步的输入
11     x = to_onehot(
12         torch.tensor([[output[-1]]], device=device),
13         vocab_size)
14     # 计算输出和更新隐藏状态
15     (Y, state) = rnn(x, state, params)
16     # 下一个时间步的输入是prefix里的字符或者当前的最佳预测字符
17     if t < len(prefix)-1:
18         output.append(char_to_idx[prefix[t+1]])
19     else:
20         output.append(int(Y[0].argmax(dim=1).item()))
21     return ''.join([idx_to_char[i] for i in output])

```

我们先测试一下 predict\_rnn 函数。我们将根据前缀“分开”创作长度为10个字符（不考虑前缀长度）的一段歌词。因为模型参数为随机值，所以预测结果也是随机的。

```

1 In [9]:
2     predict_rnn('分开', 10, rnn, params, init_rnn_state,
3                 num_hiddens, vocab_size, device,
4                 idx_to_char, char_to_idx)
5 Out [9]:
6     '分开兵环斑蜜已吗垂店连返'

```

## 5.4.5 裁剪梯度

循环神经网络中较容易出现梯度衰减或梯度爆炸。我们会在5.6节（通过时间反向传播）中解释原因。**为了应对梯度爆炸，我们可以裁剪梯度（clip gradient）。**假设我们把所有模型参数梯度的元素拼接成一个向量  $g$ ，并设裁剪的阈值是 $\theta$ 。裁剪后的梯度

$$\min\left(\frac{\theta}{\|g\|}, 1\right) g$$

的 $L_2$ 范数不超过 $\theta$ 。分析方式为：若  $\theta > \|g\|$ ，那么根据上式计算得到的张量为  $g$ ，此时计算其  $L_2$  范数为  $\|g\|$ 。若  $\theta \leq \|g\|$ ，那么根据上式计算得到的张量为  $\frac{\theta}{\|g\|} g$ ，此时计算其  $L_2$  范数为  $\theta$ 。

```

1 In [10]:
2     def grad_clipping(params, theta, device):
3         # 存储参数梯度值的平方和开根号
4         norm = torch.tensor([0.0], device=device)
5         for param in params:
6             norm += (param.grad.data**2).sum()
7         norm = norm.sqrt().item()
8         if norm > theta:
9             for param in params:
10                 param.grad.data *= (theta/norm)

```

## 5.4.6 困惑度

我们通常使用**困惑度**（perplexity）来评价语言模型的好坏。回忆一下2.4节（softmax回归）中交叉熵损失函数的定义。困惑度是对交叉熵损失函数做指数运算后得到的值。特别地，

- 最佳情况下，模型总是把标签类别的概率预测为1，此时困惑度为1；
- 最坏情况下，模型总是把标签类别的概率预测为0，此时困惑度为正无穷；
- 基线情况下，模型总是预测所有类别的概率都相同，此时困惑度为类别个数。

显然，任何一个有效模型的困惑度必须小于类别个数。在本例中，困惑度必须小于词典大小 `vocab_size`。

## 5.4.7 定义模型训练函数

跟之前章节的模型训练函数相比，这里的模型训练函数有以下几点不同：

- (1) 使用困惑度评价模型。
- (2) 在迭代模型参数前裁剪梯度。
- (3) 对时序数据采用不同采样方法将导致隐藏状态初始化的不同。相关讨论可参考5.3节。

另外，考虑到后面将介绍的其他循环神经网络，为了更通用，这里的函数实现更长一些。

```
1 In [11]:  
2     import time  
3     import math  
4     # rnn - 要使用的rnn模型  
5     # get_params - 初始化模型参数  
6     # init_rnn_state - 初始化隐藏状态  
7     # num_hiddens - 隐层神经元个数  
8     # vocab_size - 字典大小  
9     # device - 指定计算在GPU或者CPU上进行  
10    # corpus_indices - 编码之后的样本  
11    # idx_to_char - 索引到字符之间的映射  
12    # char_to_idx - 字符到索引之间的映射  
13    # is_random_iter - 是否采用随机采样  
14    # num_epochs - 训练轮次  
15    # num_steps - 每一批中每个样本的大小  
16    # lr - 学习率  
17    # clipping_theta - 梯度裁剪阈值  
18    # batch_size - 批大小  
19    # pred_period - 每多少次打印一下训练结果  
20    # pred_len - 生成样本长度  
21    # prefixes - 传入的生成引导内容  
22    def train_and_predict_rnn(rnn, get_params, init_rnn_state,  
23                                num_hiddens, vocab_size, device,  
24                                corpus_indices, idx_to_char,  
25                                char_to_idx, is_random_iter,  
26                                num_epochs, num_steps, lr, clipping_theta,  
27                                batch_size, pred_period, pred_len, prefixes):  
28        if is_random_iter:  
29            # 随机采样  
30            data_iter_fn = data_iter_random  
31        else:  
32            # 相邻采样  
33            data_iter_fn = data_iter_consecutive  
34        # 初始化模型参数  
35        params = get_params()  
36        # 交叉熵损失  
37        loss = torch.nn.CrossEntropyLoss()  
38        for epoch in range(num_epochs):  
39            if not is_random_iter:  
40                # 如使用相邻采样，在epoch开始时初始化隐藏状态  
41                state = init_rnn_state(batch_size, num_hiddens, device)  
42                # 损失之和、样本数、起始时间  
43                l_sum, n, start = 0.0, 0, time.time()
```

```

44     # 生成训练样本及label
45     data_iter = data_iter_fn(corpus_indices, batch_size,
46                               num_steps, device)
47     for x, y in data_iter:
48         # 如使用随机采样，在每个小批量更新前初始化隐藏状态
49         if is_random_iter:
50             state = init_rnn_state(batch_size,
51                                    num_hiddens, device)
52         else:
53             # 否则需要使用detach函数从计算图分离隐藏状态，这是为了
54             # 使模型参数的梯度计算只依赖一次迭代读取的小批量序列
55             # 梯度节流: https://www.cnblogs.com/catnofishing/p/13287322.html?tdsourcetag=s\_pctim\_aiomsg
56             for s in state:
57                 s.detach()
58
59     # 独热编码
60     inputs = to_onehot(x.long(), vocab_size)
61     # outputs有num_steps个形状为(batch_size, vocab_size)的矩阵
62     (outputs, state) = rnn(inputs, state, params)
63     # 拼接之后形状为(num_steps * batch_size, vocab_size)
64     outputs = torch.cat(outputs, dim=0)
65     # Y的形状是(batch_size, num_steps)，转置后再变成长度为
66     # batch * num_steps 的向量，这样跟输出的行一一对应
67     y = torch.transpose(y, 0, 1).contiguous().view(-1)
68     # 使用交叉熵损失计算平均分类误差
69     l = loss(outputs, y.long())
70     # 梯度清0
71     if params[0].grad is not None:
72         for param in params:
73             param.grad.data.zero_()
74     l.backward(retain_graph=True)
75     # 裁剪梯度
76     grad_clipping(params, clipping_theta, device)
77     # 因为误差已经取过均值，梯度不用再做平均
78     d2l.sgd(params, lr, 1)
79     l_sum += l.item() * y.shape[0]
80     n += y.shape[0]
81     if (epoch + 1) % pred_period == 0:
82         print('epoch %d, perplexity %f, time %.2f sec' %
83               (epoch+1, math.exp(l_sum/n), time.time()-start))
84     for prefix in prefixes:
85         print(' -', predict_rnn(prefix, pred_len, rnn, params,
86                                init_rnn_state, num_hiddens,
87                                vocab_size, device, idx_to_char,
88                                char_to_idx))

```

## 5.4.8 训练模型并创作歌词

现在我们可以训练模型了。首先，设置模型超参数。我们将根据前缀“分开”和“不分开”分别创作长度为50个字符（不考虑前缀长度）的一段歌词。我们每过50个迭代周期便根据当前训练的模型创作一段歌词。

```

1 In [12]:
2     num_epochs, num_steps, batch_size, lr = 250, 35, 32, 1e2
3     clipping_theta = 1e-2
4     pred_period, pred_len, prefixes = 50, 50, ['分开', '不分开']

```

下面采用随机采样训练模型并创作歌词。

```
1 In [13]:  
2     train_and_predict_rnn(rnn, get_params, init_rnn_state,  
3                             num_hiddens, vocab_size, device,  
4                             corpus_indices, idx_to_char, char_to_idx,  
5                             True, num_epochs, num_steps, lr,  
6                             clipping_theta, batch_size, pred_period,  
7                             pred_len, prefixes)  
8 Out [13]:  
9     epoch 50, perplexity 69.498287, time 0.32 sec  
10    - 分开 我想想的可爱女人 坏坏的让我疯狂的可爱女人 坏坏的让我疯狂的可爱女人  
11        坏坏的让我疯狂的可爱女人 坏  
12    - 不分开 我想想的可爱女人 坏坏的让我疯狂的可爱女人 坏坏的让我疯狂的可爱女人  
13        坏坏的让我疯狂的可爱女人 坏  
14    epoch 100, perplexity 10.113503, time 0.31 sec  
15    - 分开 一只令它心仪 哼哼哈兮 快使用双截棍 哼哼哈兮 快使用双截棍 哼哼哈兮  
16        快使用双截棍 哼哼哈兮 快  
17    - 不分开只 我不要再想 我不能再想 我不能再想 我不能再想 我不能再想  
18        我不能再想 我不能再想  
19    epoch 150, perplexity 3.004922, time 0.31 sec  
20    - 分开 还愿在美索球的太斑丘 印地安老斑年语边 我办不到 整颗心悬在半会  
21        我只能够斯坦看著 也物了一片荒芜  
22    - 不分开扫 我不能再想 我不 我不 我不能再想你 不知不觉 你已经离开我  
23        不知不觉 我该好好生活 我该好好生  
24    epoch 200, perplexity 1.593228, time 0.31 sec  
25    - 分开 装蟑在人蛛著你的可爱 让鹰盘旋死盯着腐肉 草原上两只敌对野牛  
26        想蝶著 融化共渡 一么令一片荒 真  
27    - 不分开吗 我叫你爸 你打我妈 这样对吗干嘛这样 何必让酒牵鼻的母斑鸠  
28        印地安老斑鸠 腿短毛不多 除天都没有  
29    epoch 250, perplexity 1.317237, time 0.26 sec  
30    - 分开 装蟑在人妥 你只能事忆能 景所拥 娘子她人在江南等我 泪不休  
31        语沉默 娘子她人在江南等我 泪不休  
32    - 不分开吗 然后将过去 慢慢温习 让我爱上你 那场悲剧 是你完美演出的一场戏  
33        宁愿心碎哭泣 再狠狠忘记 你爱
```

接下来采用相邻采样训练模型并创作歌词。

```
1 In [14]:  
2     train_and_predict_rnn(rnn, get_params, init_rnn_state,  
3                             num_hiddens, vocab_size, device,  
4                             corpus_indices, idx_to_char, char_to_idx,  
5                             False, num_epochs, num_steps, lr,  
6                             clipping_theta, batch_size, pred_period,  
7                             pred_len, prefixes)  
8 Out [13]:  
9     epoch 50, perplexity 61.255773, time 0.55 sec  
10    - 分开 我不要再想 我不要再想 我不要再想 我不要再想 我不要再想  
11        我不要再想 我不要再想 我不要再想 我  
12    - 不分开 你不了有 你谁了空 在谁在人 在谁的让 在果的让 在果的可  
13        在人的可 在人的可 在人的可 在人的可  
14    epoch 100, perplexity 6.411222, time 0.54 sec  
15    - 分开 我不要再样 我不懂再 在我村外的溪边河口默默等著我  
16        娘子依旧每日折一枝杨柳 你 那着 在小两外的溪  
17    - 不分开柳 你已经双 让有梦 如有箱中停留 你有在 瞎透了  
18        什么了有 沙头一碗热粥 配上几斤的牛肉 我说店  
19    epoch 150, perplexity 1.914138, time 0.55 sec
```

```
20      - 分开 我不懂 爱么我 三过了明 在故心中的溪边 默默等待
21          娘子 一什么酒 再来一碗热粥 配上几斤的牛肉
22      - 不分开柳 你已经离著的 内知哈 娘是她人在江南有我 泪不休
23          语沉默娘子她人在江南等我 爱不休 语沉默 一壶
24 epoch 200, perplexity 1.674830, time 0.55 sec
25      - 分开 她候的 旧时光 是属于那年 老师坊 旧皮箱 是属于那年
26          老师坊 旧皮箱 是属于 传满于中年的白墙
27      - 不分开觉 你已经离开你听抱 别会的钟我撒娇 从你睡著一直到老
28          就是开不了口让她知道 就是那么简单几句 我办
29 epoch 250, perplexity 6.651694, time 0.54 sec
30      - 分开 你已在 是时么 一九么那年每天 一壶现 一什么 一颗堂中旧一
31          还说骷依一中妈在 用著你 是给眼的风
32      - 不分开柳的美 一小走受是满下的暴息 你不定的想动 然黄里
33          娘点已的我不于人的温远 夕阳前可了酒下在天忆 你
```

## 小结:

- 可以用基于字符级循环神经网络的语言模型来生成文本序列，例如创作歌词。
- 当训练循环神经网络时，为了应对梯度爆炸，可以裁剪梯度。
- 困惑度是对交叉熵损失函数做指数运算后得到的值。

## 5.5 循环神经网络的简洁实现

本节将使用PyTorch来更简洁地实现基于循环神经网络的语言模型。首先，我们读取周杰伦专辑歌词数据集。

```
1 In [1]:
2     (corpus_indices, char_to_idx,
3      idx_to_char, vocab_size) = d2l.load_data_jay_lyrics()
```

### 5.5.1 定义模型

PyTorch中的`nn`模块提供了循环神经网络的实现。下面构造一个含单隐藏层、隐藏单元个数为256的循环神经网络层`rnn_layer`。

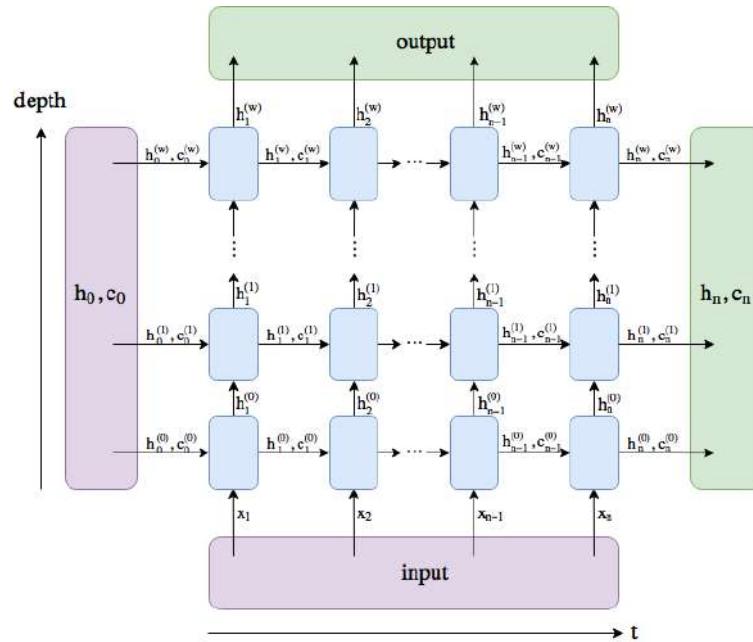
```
1 In [2]:
2     from torch import nn
3     num_hiddens = 256
4     rnn_layer = nn.RNN(input_size=vocab_size, hidden_size=num_hiddens)
```

接下来，我们使用Tensor初始化隐藏状态，他的形状为（隐藏层数、批量大小、隐藏单元个数）。

```
1 In [3]:
2     state = torch.zeros((1, 2, num_hiddens))
3     state.shape
4 Out [3]:
5     torch.Size([1, 2, 256])
```

与上一节中实现的循环神经网络不同，这里`rnn_layer`的输入形状为（**时间步数, 批量大小, 输入个数**）。其中输入个数即one-hot向量长度（词典大小）。此外，`rnn_layer`作为`nn.RNN`实例，在前向计算后会分别返回输出和隐藏状态`h`，其中输出指的是隐藏层在**各个时间步**上计算并输出的隐藏状态，它们通常作为后续输出层的输入。需要强调的是，该“输出”本身并不涉及输出层计算，形状为（时间步数，批量大小，隐藏单元个数）。而`nn.RNN`实例在前向计算返回的隐藏状态指的是隐藏层在**最后时间步**的隐

藏状态：当隐藏层有多层时，每一层的隐藏状态都会记录在该变量中；对于像长短期记忆（LSTM），隐藏状态是一个元组( $h, c$ )，即hidden state和cell state。我们会在本章的后面介绍长短期记忆和深度循环神经网络。关于循环神经网络（以LSTM为例）的输出，可以参考下图。



来看看我们的例子，输出形状为（时间步数, 批量大小, 隐藏单元个数），隐藏状态 $h$ 的形状为（层数, 批量大小, 隐藏单元个数）。

```

1 In [4]:
2     num_steps = 32
3     batch_size = 2
4     X = torch.rand(num_steps, batch_size, vocab_size)
5     Y, state_new = rnn_layer(X, state)
6     print(Y.shape, len(state_new), state_new[0].shape)
7 Out [4]:
8     torch.Size([32, 2, 256]) 1 torch.size([2, 256])

```

接下来我们继承 `Module` 类来定义一个完整的循环神经网络。它首先将输入数据使用one-hot向量表示后输入到 `rnn_layer` 中，然后使用全连接输出层得到输出。输出个数等于词典大小 `vocab_size`。

```

1 In [5]:
2 class RNNModel(nn.Module):
3     def __init__(self, rnn_layer, vocab_size):
4         super(RNNModel, self).__init__()
5         self.rnn = rnn_layer
6         # 若为双向循环网络需要*2
7         self.hidden_size = rnn_layer.hidden_size*(
8             2 if rnn_layer.bidirectional else 1)
9         self.vocab_size = vocab_size
10        self.dense = nn.Linear(self.hidden_size, vocab_size)
11        self.state = None
12    # inputs: (batch, seq_len)
13    def forward(self, inputs, state):
14        # 获取one-hot向量表示
15        # X是个list
16        X = d2l.to_onehot(inputs.long(), self.vocab_size)
17        Y, self.state = self.rnn(X.float(), state)
18        # 全连接层会首先将Y的形状变成(num_steps * batch_size, num_hiddens)

```

```
19     # 它的输出形状为(num_steps * batch_size, vocab_size)
20     output = self.dense(Y.view(-1, Y.shape[-1]))
21     return output, self.state
```

## 5.5.2 训练模型

同上一节一样，下面定义一个预测函数。这里的实现区别在于前向计算和初始化隐藏状态的函数接口。

```
1 In [6]:
2     def predict_rnn_pytorch(prefix, num_chars, model, vocab_size,
3                               device, idx_to_char, char_to_idx):
4         # 初始隐藏层状态可以不定义
5         state = None
6         # output会记录prefix加上输出
7         # 将输入的首字符传入到输出序列中
8         output = [char_to_idx[prefix[0]]]
9         for t in range(num_chars+1):
10             # 将上一时间步的输出作为当前时间步的输入
11             x = torch.tensor([output[-1]], device=device).view(1, 1)
12             if state is not None:
13                 if isinstance(state, tuple):
14                     state = (state[0].to(device), state[1].to(device))
15                 else:
16                     state = state.to(device)
17             (Y, state) = model(x, state)
18             if t < len(prefix) - 1:
19                 output.append(char_to_idx[prefix[t+1]])
20             else:
21                 output.append(int(Y.argmax(dim=1).item()))
22     return ''.join([idx_to_char[i] for i in output])
```

让我们使用权重为随机值的模型来预测一次。

```
1 In [7]:
2     model = RNNModel(rnn_layer, vocab_size).to(device)
3     predict_rnn_pytorch('分开', 10, model, vocab_size,
4                          device, idx_to_char, char_to_idx)
5 Out [7]:
6     '分开歌斑音觉涌鹿田斑去用'
```

接下来实现训练函数。算法同上一节的一样，但这里只使用了相邻采样来读取数据。

```
1 In [8]:
2     def train_and_predict_rnn_pytorch(model, num_hiddens, vocab_size,
3                                       device, corpus_indices,
4                                       idx_to_char, char_to_idx,
5                                       num_epochs, num_steps, lr,
6                                       clipping_theta, batch_size,
7                                       pred_period, pred_len, prefixes):
8         # 交叉熵损失
9         loss = torch.nn.CrossEntropyLoss()
10        optimizer = torch.optim.Adam(model.parameters(), lr=lr)
11        model.to(device)
12        state = None
```

```

13     for epoch in range(num_epochs):
14         # 损失之和、样本数、起始时间
15         l_sum, n, start = 0.0, 0, time.time()
16         # 相邻采样
17         data_iter = d2l.data_iter_consecutive(
18             corpus_indices, batch_size, num_steps, device)
19         for X, Y in data_iter:
20             if state is not None:
21                 # https://www.cnblogs.com/catnofishing/p/
22                 # 13287322.html?tdsourcetag=s_pctim_aiomsg
23                 if isinstance(state, tuple):
24                     state = (state[0].detach(), state[1].detach())
25                 else:
26                     state = state.detach()
27             # output: 形状为(num_steps * batch_size, vocab_size)
28             (output, state) = model(X, state)
29             # Y的形状是(batch_size, num_steps), 转置后再变成长度为
30             # batch * num_steps 的向量, 这样跟输出的行一一对应
31             y = torch.transpose(Y, 0, 1).contiguous().view(-1)
32             # 使用交叉熵损失计算平均分类误差
33             l = loss(output, y.long())
34             # 梯度清0
35             optimizer.zero_grad()
36             l.backward()
37             # 裁剪梯度
38             d2l.grad_clipping(model.parameters(), clipping_theta,
39                               device)
40             optimizer.step()
41             l_sum += l.item() * y.shape[0]
42             n += y.shape[0]
43         try:
44             perplexity = math.exp(l_sum / n)
45         except OverflowError:
46             perplexity = float('inf')
47         if (epoch + 1) % pred_period == 0:
48             print('epoch %d, perplexity %f, time %.2f sec' %
49                  (epoch+1, perplexity, time.time()-start))
50             for prefix in prefixes:
51                 print(' - ', predict_rnn_pytorch(prefix, pred_len,
52                                                 model, vocab_size,
53                                                 device, idx_to_char,
54                                                 char_to_idx))

```

使用和上一节实验中一样的超参数（除了学习率）来训练模型。

```

1 In [9]:
2     # 注意这里的學習率设置
3     num_epochs, batch_size, lr, clipping_theta = 250, 32, 1e-2, 1e-2
4     pred_period, pred_len, prefixes = 50, 50, ['分开', '不分开']
5     train_and_predict_rnn_pytorch(model, num_hiddens, vocab_size, device,
6                                     corpus_indices, idx_to_char, char_to_idx,
7                                     num_epochs, num_steps, lr, clipping_theta,
8                                     batch_size, pred_period, pred_len,
9                                     prefixes)
10 Out [9]:
11 epoch 50, perplexity 1.152438, time 0.07 sec
12     - 分开的我想就像 不要 我 在进黑色幽默 爱你的明的手不放开

```

```

13 爱可不可以简简单单没有伤害 你 靠着我的肩膀
14 - 不分开不要活 也有苦笑 没有你在的爸爸一句带酒 败给你的黑色幽默
15 不想太多 我想 我的肩膀 你 在我胸口睡
16 epoch 100, perplexity 1.146996, time 0.07 sec
17 - 分开的我后 快使用双截棍 哼哼哈兮 快使用双截棍 哼哼哈兮
18 快使用双截棍 哼哼哈兮 快使用双截棍 哼哼哈
19 - 不分开不美边 这样的梦 出现裂缝 隐隐作痛 随便个战 弓箭
20 太多的话去对医药箱说 别怪我 别怪我 三分球
21 epoch 150, perplexity 1.026819, time 0.07 sec
22 - 分开的我有我这辈子注定一个人演戏 最后再一个人慢慢的回忆
23 没有了过去 我将往事抽离 如果我遇见你是一场悲
24 - 不分开不要再这样的节奏 谁都无可奈何 没有你以后 我灵魂失控
25 黑云在降落 我被它拖着走 静静悄悄默默离开
26 epoch 200, perplexity 1.022801, time 0.07 sec
27 - 分开的我有我爱你 爱情来的太快就像龙卷风 离不开暴风圈来不及逃
28 我不能再想 我不能再想 我不 我不 我不
29 - 不分开不知天空 分手不这样牵着你的手不放开
30 爱可不可以简简单单没有伤害 你 靠着我的肩膀 你 在我胸口睡著
31 epoch 250, perplexity 1.024344, time 0.07 sec
32 - 分开的我后悔着对不起 一枚铜币 悲伤得很隐密 它在许愿池里轻轻叹息
33 太多的我爱你 让它喘不过气 已经 失
34 - 不分开 我有一双翅膀 二双翅膀 随时出发 偷偷出发 我一定带我妈走
35 从前的教育别人的家庭 别人的爸爸种

```

## 小结:

- PyTorch的 `nn` 模块提供了循环神经网络层的实现。
- PyTorch的 `nn.RNN` 实例在前向计算后会分别返回输出和隐藏状态。该前向计算并不涉及输出层计算。

## 5.6 通过时间反向传播

在前面两节中，如果不裁剪梯度，模型将无法正常训练。为了深刻理解这一现象，本节将介绍循环神经网络中梯度的计算和存储方法，即通过时间反向传播（back-propagation through time）。

我们在2.14节（正向传播、反向传播和计算图）中介绍了神经网络中梯度计算与存储的一般思路，并强调正向传播和反向传播相互依赖。正向传播在循环神经网络中比较直观，而通过时间反向传播其实是反向传播在循环神经网络中的具体应用。我们需要将循环神经网络按时间步展开，从而得到模型变量和参数之间的依赖关系，并依据链式法则应用反向传播计算并存储梯度。

### 5.6.1 定义模型

简单起见，我们考虑一个无偏差项的循环神经网络，且激活函数为恒等映射 ( $\phi(x) = x$ )。设时间步  $t$  的输入为单样本  $\mathbf{x}_t \in \mathbb{R}^d$ ，标签为  $y_t$ ，那么隐藏状态  $\mathbf{h}_t \in \mathbb{R}^h$  的计算表达式为

$$\mathbf{h}_t = \mathbf{W}_{hx} \mathbf{x}_t + \mathbf{W}_{hh} \mathbf{h}_{t-1}$$

其中  $\mathbf{W}_{hx} \in \mathbb{R}^{h \times d}$  和  $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$  是隐藏层权重参数。设输出层权重参数  $\mathbf{W}_{qh} \in \mathbb{R}^{q \times h}$ ，时间步  $t$  的输出层变量  $\mathbf{o}_t \in \mathbb{R}^q$  计算为

$$\mathbf{o}_t = \mathbf{W}_{qh} \mathbf{h}_t$$

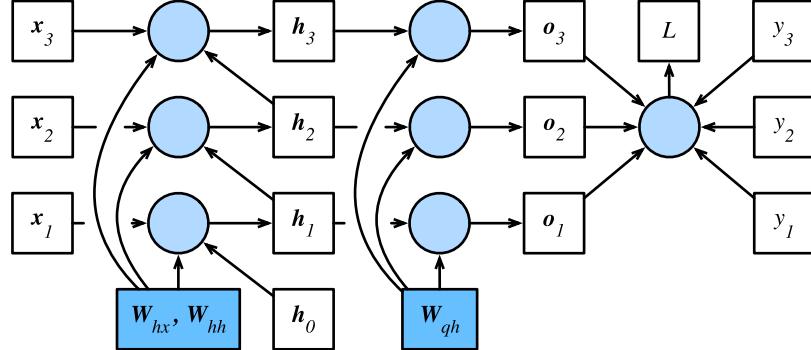
设时间步  $t$  的损失为  $\ell(\mathbf{o}_t, y_t)$ 。时间步数为  $T$  的损失函数  $L$  定义为

$$L = \frac{1}{T} \sum_{t=1}^T \ell(\mathbf{o}_t, y_t)$$

我们将 $L$ 称为有关给定时间步的数据样本的目标函数（平均每一步的损失值），并在本节后续讨论中简称为目标函数。

## 5.6.2 模型计算图

为了可视化循环神经网络中模型变量和参数在计算中的依赖关系，我们可以绘制模型计算图，如下图所示。例如，时间步3的隐藏状态 $\mathbf{h}_3$ 的计算依赖模型参数 $\mathbf{W}_{hx}$ 、 $\mathbf{W}_{hh}$ 、上一时间步隐藏状态 $\mathbf{h}_2$ 以及当前时间步输入 $\mathbf{x}_3$ 。



时间步数为3的循环神经网络模型计算中的依赖关系。方框代表变量（无阴影）或参数（有阴影），圆圈代表运算符

需要注意的是，整个循环网络只有最后一行这几个参数。

## 5.6.3 方法

刚刚提到，上图中的模型的参数是 $\mathbf{W}_{hx}$ 、 $\mathbf{W}_{hh}$ 和 $\mathbf{W}_{qh}$ 。与2.14节（正向传播、反向传播和计算图）中的类似，训练模型通常需要模型参数的梯度 $\partial L / \partial \mathbf{W}_{hx}$ 、 $\partial L / \partial \mathbf{W}_{hh}$ 和 $\partial L / \partial \mathbf{W}_{qh}$ 。根据上图中的依赖关系，我们可以按照其中箭头所指的反方向依次计算并存储梯度。为了表述方便，我们依然采用2.14节中表达链式法则的运算符prod。

首先，目标函数有关各时间步输出层变量的梯度 $\partial L / \partial \mathbf{o}_t \in \mathbb{R}^q$ 很容易计算：

$$\frac{\partial L}{\partial \mathbf{o}_t} = \frac{\partial \ell(\mathbf{o}_t, y_t)}{T \cdot \partial \mathbf{o}_t}$$

下面，我们可以计算目标函数有关模型参数 $\mathbf{W}_{qh}$ 的梯度 $\partial L / \partial \mathbf{W}_{qh} \in \mathbb{R}^{q \times h}$ 。根据上图， $L$ 通过 $\mathbf{o}_1, \dots, \mathbf{o}_T$ 依赖 $\mathbf{W}_{qh}$ 。依据链式法则，

$$\frac{\partial L}{\partial \mathbf{W}_{qh}} = \sum_{t=1}^T \text{prod} \left( \frac{\partial L}{\partial \mathbf{o}_t}, \frac{\partial \mathbf{o}_t}{\partial \mathbf{W}_{qh}} \right) = \sum_{t=1}^T \frac{\partial L}{\partial \mathbf{o}_t} \mathbf{h}_t^\top$$

其次，我们注意到隐藏状态之间也存在依赖关系。在上图中， $L$ 只通过 $\mathbf{o}_T$ 依赖最终时间步 $T$ 的隐藏状态 $\mathbf{h}_T$ 。因此，我们先计算目标函数有关最终时间步隐藏状态的梯度 $\partial L / \partial \mathbf{h}_T \in \mathbb{R}^h$ 。依据链式法则，我们得到

$$\frac{\partial L}{\partial \mathbf{h}_T} = \text{prod} \left( \frac{\partial L}{\partial \mathbf{o}_T}, \frac{\partial \mathbf{o}_T}{\partial \mathbf{h}_T} \right) = \mathbf{W}_{qh}^\top \frac{\partial L}{\partial \mathbf{o}_T}$$

接下来对于时间步 $t < T$ ，在上图中， $L$ 通过 $\mathbf{h}_{t+1}$ 和 $\mathbf{o}_t$ 依赖 $\mathbf{h}_t$ 。依据链式法则，目标函数有关时间步 $t < T$ 的隐藏状态的梯度 $\partial L / \partial \mathbf{h}_t \in \mathbb{R}^h$ 需要按照时间步从大到小依次计算：

$$\frac{\partial L}{\partial \mathbf{h}_t} = \text{prod} \left( \frac{\partial L}{\partial \mathbf{h}_{t+1}}, \frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t} \right) + \text{prod} \left( \frac{\partial L}{\partial \mathbf{o}_t}, \frac{\partial \mathbf{o}_t}{\partial \mathbf{h}_t} \right) = \mathbf{W}_{hh}^\top \frac{\partial L}{\partial \mathbf{h}_{t+1}} + \mathbf{W}_{qh}^\top \frac{\partial L}{\partial \mathbf{o}_t}$$

将上面的递归公式展开，对任意时间步 $1 \leq t \leq T$ ，我们可以得到目标函数有关隐藏状态梯度的通项公式

$$\frac{\partial L}{\partial \mathbf{h}_t} = \sum_{i=t}^T (\mathbf{W}_{hh}^\top)^{T-i} \mathbf{W}_{qh}^\top \frac{\partial L}{\partial \mathbf{o}_{T+t-i}}$$

我们举一个例子对上式进行验证，已知  $t$  时刻损失函数  $L$  对隐藏状态  $\mathbf{h}_t$  的偏导数为：

$$\frac{\partial L}{\partial \mathbf{h}_t} = \mathbf{W}_{hh}^\top \frac{\partial L}{\partial \mathbf{h}_{t+1}} + \mathbf{W}_{qh}^\top \frac{\partial L}{\partial \mathbf{o}_t} \quad (1)$$

那么，根据此递推公式可知  $t+1$  时刻损失函数  $L$  对隐藏状态  $\mathbf{h}_{t+1}$  的偏导数为：

$$\frac{\partial L}{\partial \mathbf{h}_{t+1}} = \mathbf{W}_{hh}^\top \frac{\partial L}{\partial \mathbf{h}_{t+2}} + \mathbf{W}_{qh}^\top \frac{\partial L}{\partial \mathbf{o}_{t+1}} \quad (2)$$

我们将 (2) 式代入 (1) 式有：

$$\frac{\partial L}{\partial \mathbf{h}_t} = \mathbf{W}_{hh}^\top \left( \mathbf{W}_{hh}^\top \frac{\partial L}{\partial \mathbf{h}_{t+2}} + \mathbf{W}_{qh}^\top \frac{\partial L}{\partial \mathbf{o}_{t+1}} \right) + \mathbf{W}_{qh}^\top \frac{\partial L}{\partial \mathbf{o}_t}$$

我们假设  $t+2 = T$ ，那么上式可以化简为：

$$\begin{aligned} \frac{\partial L}{\partial \mathbf{h}_t} &= \mathbf{W}_{hh}^\top \left( \mathbf{W}_{hh}^\top \mathbf{W}_{qh}^\top \frac{\partial L}{\partial \mathbf{o}_T} + \mathbf{W}_{qh}^\top \frac{\partial L}{\partial \mathbf{o}_{T-1}} \right) + \mathbf{W}_{qh}^\top \frac{\partial L}{\partial \mathbf{o}_{T-2}} \\ &= (\mathbf{W}_{hh}^\top)^2 \mathbf{W}_{qh}^\top \frac{\partial L}{\partial \mathbf{o}_T} + \mathbf{W}_{hh}^\top \mathbf{W}_{qh}^\top \frac{\partial L}{\partial \mathbf{o}_{T-1}} + \mathbf{W}_{qh}^\top \frac{\partial L}{\partial \mathbf{o}_{T-2}} \end{aligned}$$

这个式子与通项公式的计算结果完全一致。

由通项公式中的指数项可见，当时间步数  $T$  较大或者时间步  $t$  较小时，目标函数有关隐藏状态的梯度较容易出现衰减和爆炸（幂次高导致）。这也将影响其他包含  $\partial L / \partial \mathbf{h}_t$  项的梯度，例如隐藏层中模型参数的梯度  $\partial L / \partial \mathbf{W}_{hx} \in \mathbb{R}^{h \times d}$  和  $\partial L / \partial \mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ 。在上图中， $L$  通过  $\mathbf{h}_1, \dots, \mathbf{h}_T$  依赖这些模型参数。依据链式法则，我们有

$$\begin{aligned} \frac{\partial L}{\partial \mathbf{W}_{hx}} &= \sum_{t=1}^T \text{prod} \left( \frac{\partial L}{\partial \mathbf{h}_t}, \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{hx}} \right) = \sum_{t=1}^T \frac{\partial L}{\partial \mathbf{h}_t} \mathbf{x}_t^\top \\ \frac{\partial L}{\partial \mathbf{W}_{hh}} &= \sum_{t=1}^T \text{prod} \left( \frac{\partial L}{\partial \mathbf{h}_t}, \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{hh}} \right) = \sum_{t=1}^T \frac{\partial L}{\partial \mathbf{h}_t} \mathbf{h}_{t-1}^\top \end{aligned}$$

我们已在 2.14 节里解释过，每次迭代中，我们在依次计算完以上各个梯度后，会将它们存储起来，从而避免重复计算。例如，由于隐藏状态梯度  $\partial L / \partial \mathbf{h}_t$  被计算和存储，之后的模型参数梯度  $\partial L / \partial \mathbf{W}_{hx}$  和  $\partial L / \partial \mathbf{W}_{hh}$  的计算可以直接读取  $\partial L / \partial \mathbf{h}_t$  的值，而无须重复计算它们。此外，反向传播中的梯度计算可能会依赖变量的当前值。它们正是通过正向传播计算出来的。举例来说，参数梯度  $\partial L / \partial \mathbf{W}_{hh}$  的计算需要依赖隐藏状态在时间步  $t = 0, \dots, T-1$  的当前值  $\mathbf{h}_t$  ( $\mathbf{h}_0$  是初始化得到的)。这些值是通过从输入层到输出层的正向传播计算并存储得到的。

### 小结：

- 通过时间反向传播是反向传播在循环神经网络中的具体应用。
- 当总的时间步数较大或者当前时间步较小时，循环神经网络的梯度较容易出现衰减或爆炸。

## 5.7 门控循环单元 (GRU)

上一节介绍了循环神经网络中的梯度计算方法。我们发现，当时间步数较大或者时间步较小时，循环神经网络的梯度较容易出现衰减或爆炸。虽然裁剪梯度可以应对梯度爆炸，但无法解决梯度衰减的问题。通常由于这个原因，循环神经网络在实际中较难捕捉时间序列中时间步距离较大的依赖关系。

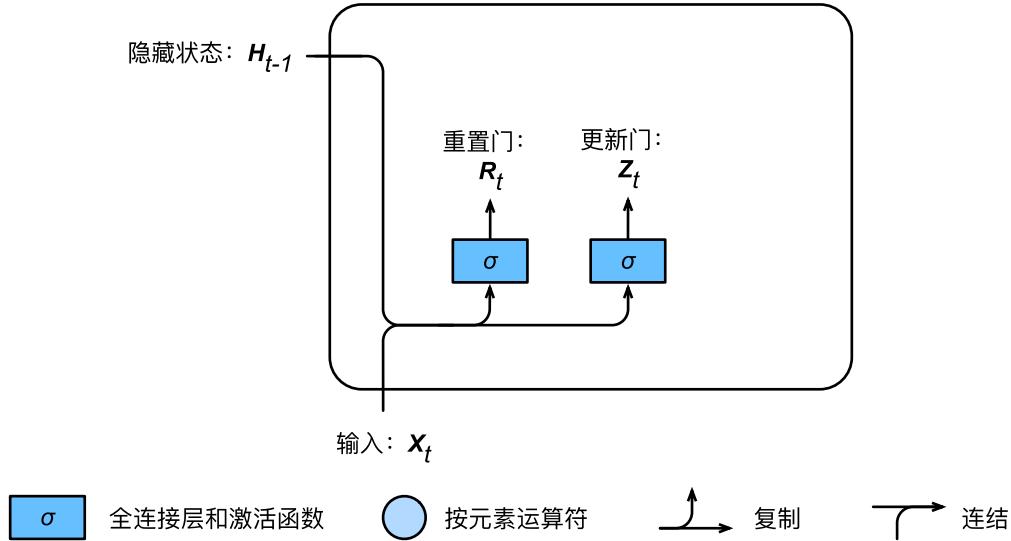
门控循环神经网络 (gated recurrent neural network) 的提出，正是为了更好地捕捉时间序列中时间步距离较大的依赖关系。它通过可以学习的门来控制信息的流动。其中，门控循环单元 (gated recurrent unit, GRU) 是一种常用的门控循环神经网络。另一种常用的门控循环神经网络则将在下一节中介绍。

## 5.7.1 门控循环单元

下面将介绍门控循环单元的设计。它引入了重置门 (reset gate) 和更新门 (update gate) 的概念，从而修改了循环神经网络中隐藏状态的计算方式。

### 1. 重置门和更新门

如下图所示，门控循环单元中的重置门和更新门的输入均为当前时间步输入  $X_t$  与上一时间步隐藏状态  $H_{t-1}$ ，输出由激活函数为 sigmoid 函数的全连接层计算得到。



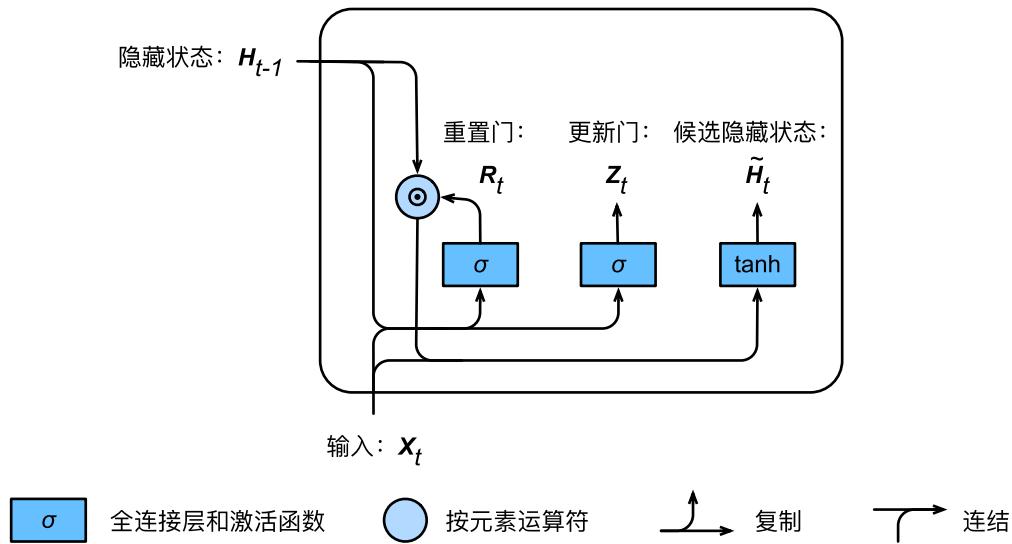
具体来说，假设隐藏单元个数为  $h$ ，给定时间步  $t$  的小批量输入  $X_t \in \mathbb{R}^{n \times d}$  (样本数为  $n$ ，输入个数为  $d$ ) 和上一时间步隐藏状态  $H_{t-1} \in \mathbb{R}^{n \times h}$ 。重置门  $R_t \in \mathbb{R}^{n \times h}$  和更新门  $Z_t \in \mathbb{R}^{n \times h}$  的计算如下：

$$\begin{aligned} R_t &= \sigma(X_t W_{xr} + H_{t-1} W_{hr} + b_r) \\ Z_t &= \sigma(X_t W_{xz} + H_{t-1} W_{hz} + b_z) \end{aligned}$$

其中  $W_{xr}, W_{xz} \in \mathbb{R}^{d \times h}$  和  $W_{hr}, W_{hz} \in \mathbb{R}^{h \times h}$  是权重参数， $b_r, b_z \in \mathbb{R}^{1 \times h}$  是偏差参数。2.8节 (多层感知机) 节中介绍过，sigmoid 函数可以将元素的值变换到 0 和 1 之间。因此，重置门  $R_t$  和更新门  $Z_t$  中每个元素的值域都是  $[0, 1]$ 。

### 2. 候选隐藏状态

接下来，门控循环单元将计算候选隐藏状态来辅助稍后的隐藏状态计算。如下图所示，我们将当前时间步重置门的输出与上一时间步隐藏状态做按元素乘法 (符号为  $\odot$ )。如果重置门中元素值接近 0，那么意味着重置对应隐藏状态元素为 0，即丢弃上一时间步的隐藏状态。如果元素值接近 1，那么表示保留上一时间步的隐藏状态。然后，将按元素乘法的结果与当前时间步的输入相加，再通过含激活函数  $\tanh$  的全连接层计算出候选隐藏状态，其所有元素的值域为  $[-1, 1]$ 。



具体来说，时间步 $t$ 的候选隐藏状态 $\tilde{H}_t \in \mathbb{R}^{n \times h}$ 的计算为

$$\tilde{H}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xh} + (\mathbf{R}_t \odot H_{t-1}) \mathbf{W}_{hh} + \mathbf{b}_h)$$

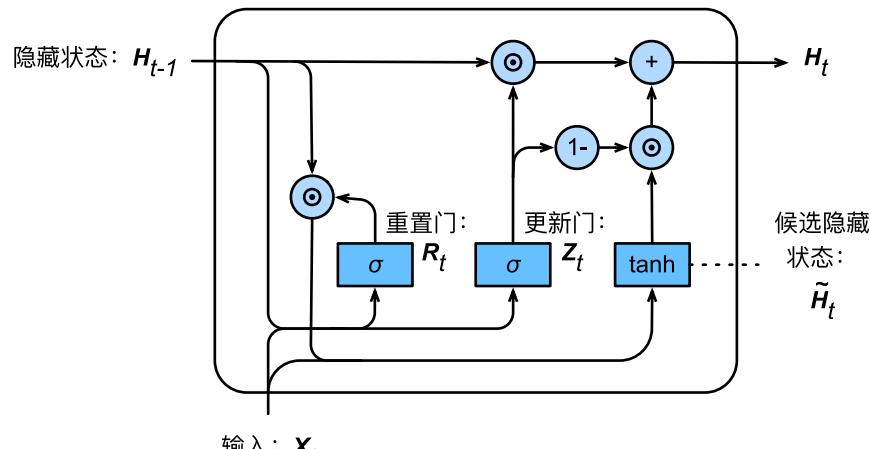
其中 $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$ 和 $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ 是权重参数， $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$ 是偏差参数。从上面这个公式可以看出，重置门控制了上一时间步的隐藏状态如何流入当前时间步的候选隐藏状态。而上一时间步的隐藏状态可能包含了时间序列截至上一时间步的全部历史信息。因此，重置门可以用来丢弃与预测无关的历史信息。

### 3. 隐藏状态

最后，时间步 $t$ 的隐藏状态 $H_t \in \mathbb{R}^{n \times h}$ 的计算使用当前时间步的更新门 $Z_t$ 来对上一时间步的隐藏状态 $H_{t-1}$ 和当前时间步的候选隐藏状态 $\tilde{H}_t$ 做组合：

$$H_t = Z_t \odot H_{t-1} + (1 - Z_t) \odot \tilde{H}_t$$

值得注意的是，更新门可以控制隐藏状态应该如何被包含当前时间步信息的候选隐藏状态所更新，如下图所示。假设更新门在时间步 $t'$ 到 $t$  ( $t' < t$ ) 之间一直近似1。那么，在时间步 $t'$ 到 $t$ 之间的输入信息几乎没有流入时间步 $t$ 的隐藏状态 $H_t$ 。实际上，这可以看作是较早时刻的隐藏状态 $H_{t'-1}$ 一直通过时间保存并传递至当前时间步 $t$ 。这个设计可以应对循环神经网络中的梯度衰减问题，并更好地捕捉时间序列中时间步距离较大的依赖关系。



我们对门控循环单元的设计稍作总结：

- 重置门有助于捕捉时间序列里**短期**的依赖关系；
- 更新门有助于捕捉时间序列里**长期**的依赖关系。

## 5.7.2 读取数据集

为了实现并展示门控循环单元，下面依然使用周杰伦歌词数据集来训练模型作词。这里除门控循环单元以外的实现已在5.2节（循环神经网络）中介绍过。以下为读取数据集部分。

```
1 In [1]:  
2     (corpus_indices, char_to_idx,  
3      idx_to_char, vocab_size) = d2l.load_data_jay_lyrics()
```

## 5.7.3 从零开始实现

我们先介绍如何从零开始实现门控循环单元。

### 1. 初始化模型参数

下面的代码对模型参数进行初始化。超参数 `num_hiddens` 定义了隐藏单元的个数。其中 `parameterList()` 就是一种和列表、元组之类一样的一种新的数据格式，用于保存神经网络权重及参数。

```
1 In [2]:  
2     num_inputs, num_hiddens, num_outputs = vocab_size, 256, vocab_size  
3     print('will use', device)  
4     def get_params():  
5         def _one(shape):  
6             ts = torch.tensor(  
7                 np.random.normal(0, 0.01, size=shape),  
8                 device=device, dtype=torch.float32)  
9             return torch.nn.Parameter(ts, requires_grad=True)  
10    def _three():  
11        return (_one((num_inputs, num_hiddens)),  
12                _one((num_hiddens, num_hiddens)),  
13                torch.nn.Parameter(  
14                    torch.zeros(  
15                        num_hiddens,  
16                        device=device,  
17                        dtype=torch.float32),  
18                        requires_grad=True))  
19        # 更新门参数  
20        w_xz, w_hz, b_z = _three()  
21        # 重置门参数  
22        w_xr, w_hr, b_r = _three()  
23        # 候选隐藏状态参数  
24        w_xh, w_hh, b_h = _three()  
25        # 输出层参数  
26        w_hq = _one((num_hiddens, num_outputs))  
27        b_q = torch.nn.Parameter(  
28            torch.zeros(num_outputs, device=device, dtype=torch.float32),  
29            requires_grad=True)  
30        return nn.ParameterList(  
31            [w_xz, w_hz, b_z,  
32             w_xr, w_hr, b_r,  
33             w_xh, w_hh, b_h,  
34             w_hq, b_q])  
35 Out [2]:  
36     will use cuda
```

## 2. 定义模型

下面的代码定义隐藏状态初始化函数 `init_gru_state`。同5.4节中定义的 `init_rnn_state` 函数一样，它返回由一个形状为（批量大小, 隐藏单元个数）的值为0的 `Tensor` 组成的元组。

```
1 In [3]:  
2     def init_gru_state(batch_size, num_hiddens, device):  
3         return (torch.zeros((batch_size, num_hiddens), device=device), )
```

下面根据门控循环单元的计算表达式定义模型。

```
1 In [4]:  
2     def gru(inputs, state, params):  
3         # 参数  
4         w_xz, w_hz, b_z, w_xr, w_hr, b_r,  
5         w_xh, w_hh, b_h, w_hq, b_q = params  
6         # 初始化的隐藏状态  
7         H, = state  
8         outputs = []  
9         for X in inputs:  
10             Z = torch.sigmoid(torch.matmul(X.float(), w_xz)  
11                         + torch.matmul(H, w_hz) + b_z)  
12             R = torch.sigmoid(torch.matmul(X.float(), w_xr)  
13                         + torch.matmul(H, w_hr) + b_r)  
14             H_tilda = torch.tanh(torch.matmul(X.float(), w_xh)  
15                         + torch.matmul(R*H, w_hh) + b_h)  
16             H = Z*H + (1-Z)*H_tilda  
17             Y = torch.matmul(H, w_hq) + b_q  
18             outputs.append(Y)  
19         return outputs, (H,)
```

## 3. 训练模型并创作歌词

我们在训练模型时只使用相邻采样。设置好超参数后，我们将训练模型并根据前缀“分开”和“不分开”分别创作长度为50个字符的一段歌词。

```
1 In [5]:  
2     num_epochs, num_steps, batch_size, lr = 160, 35, 32, 1e2  
3     clipping_theta = 1e-2  
4     pred_period, pred_len, prefixes = 40, 50, ['分开', '不分开']
```

我们每过40个迭代周期便根据当前训练的模型创作一段歌词。

```
1 In [6]:  
2     d2l.train_and_predict_rnn(gru, get_params, init_gru_state, num_hiddens,  
3                                 vocab_size, device, corpus_indices,  
4                                 idx_to_char, char_to_idx, False, num_epochs,  
5                                 num_steps, lr, clipping_theta, batch_size,  
6                                 pred_period, pred_len, prefixes)  
7 Out [6]:  
8     epoch 40, perplexity 149.917021, time 2.04 sec  
9     - 分开 我想你 我想你 你不我 你不的让我 你想我 你不我  
10    你不的我 你不的让我 你想我 你不我 你不的  
11    - 不分开 我想你 我想你 你不我 你不的让我 你想我 你不我  
12    你不的我 你不的让我 你想我 你不我 你不的
```

```

13 epoch 80, perplexity 32.286560, time 2.07 sec
14 - 分开 我想要这样 我不要再想 我不 我不 我不 我不
15     我不 我不 我不 我不 我不 我
16 - 不分开 我有你这样我 一场好觉 你爱我 别你的手 在人放人
17     温爱在人人 爱我有你的微笑 像龙放口 让我
18 epoch 120, perplexity 5.518456, time 2.04 sec
19 - 分开 我想要你的微笑每天都能看离 我知道这里很美但家乡的你更美
20     我想要这样坦堡 像这样的生色 让我开红
21 - 不分开 我已能这样打我妈妈 我不能再想 不知不觉 我不要再想
22     我不能再想 我不 我不 我不能 爱情走的太
23 epoch 160, perplexity 1.665924, time 2.02 sec
24 - 分开 小作 所有那里对着我进攻
25     古巴忆对王颁布了汉摩拉比典轻轻的脸窗 古录像一个世
26 - 不分开 你是一直 是你的那笑 喜的风美主义 平伤你的手
27     一阵莫名感动 我以带你 回我的外婆家 一起看着日落

```

## 5.7.4 简洁实现

在PyTorch中我们直接调用 `nn` 模块中的 `GRU` 类即可。

```

1 In [7]:
2     lr = 1e-2
3     gru_layer = nn.GRU(input_size=vocab_size, hidden_size=num_hiddens)
4     model = d2l.RNNModel(gru_layer, vocab_size).to(device)
5     d2l.train_and_predict_rnn_pytorch(model, num_hiddens, vocab_size,
6                                         device, corpus_indices,
7                                         idx_to_char, char_to_idx,
8                                         num_epochs, num_steps, lr,
9                                         clipping_theta, batch_size,
10                                        pred_period, pred_len, prefixes)
11 Out [7]:
12 epoch 40, perplexity 1.016270, time 0.07 sec
13 - 分开始乡情怯的我 相思寄红豆 相思寄红豆无能为力的在人海中漂泊心伤透
14     娘子她人在江南等我 泪不休 语沉默
15 - 不分开始乡相信命运 感谢地心引力 让我碰到你 漂亮的让我面红的可爱女人
16     温柔的让我心疼的可爱女人 透明的让
17 epoch 80, perplexity 1.009380, time 0.07 sec
18 - 分开始乡情怯的我 相思寄红豆 相思寄红豆无能为力的在人海中漂泊心伤透
19     娘子她人在江南等我 泪不休 语沉默
20 - 不分开始打呼 管家是一只会说法语举止优雅的猪 吸血前会念约翰福音做为弥补
21     拥有一双蓝色眼睛的凯萨琳公主 专
22 epoch 120, perplexity 1.009627, time 0.07 sec
23 - 分开始乡还为分手不投 又不会掩护我 选你这种队友 瞎透了我 说你说
24     分数怎么停留 一直在停留 谁让它停留
25 - 不分开始乡相信命运 感谢地心引力 让我碰到你 漂亮的让我面红的可爱女人
26     温柔的让我心疼的可爱女人 透明的让
27 epoch 160, perplexity 1.011384, time 0.06 sec
28 - 分开始乡还是你的脑袋有问题 随便说说 其实我早已经猜透看透不想多说
29     只是我怕眼泪撑不住 不懂 你的黑色幽
30 - 不分开始乡相信命运 感谢地心引力 让我碰到你 漂亮的让我面红的可爱女人
31     温柔的让我心疼的可爱女人 透明的让

```

### 小结:

- 门控循环神经网络可以更好地捕捉时间序列中时间步距离较大的依赖关系。

- 门控循环单元引入了门的概念，从而修改了循环神经网络中隐藏状态的计算方式。它包括重置门、更新门、候选隐藏状态和隐藏状态。
- 重置门有助于捕捉时间序列里短期的依赖关系。
- 更新门有助于捕捉时间序列里长期的依赖关系。

## 5.8 长短期记忆 (LSTM)

本节将介绍另一种常用的门控循环神经网络：[长短期记忆](#) (long short-term memory, LSTM)。它比门控循环单元的结构稍微复杂一点。

### 5.8.1 长短期记忆

LSTM 中引入了3个门，即**输入门** (input gate)、**遗忘门** (forget gate) 和**输出门** (output gate)，以及与隐藏状态形状相同的**记忆细胞** (某些文献把记忆细胞当成一种特殊的隐藏状态)，从而记录额外的信息。

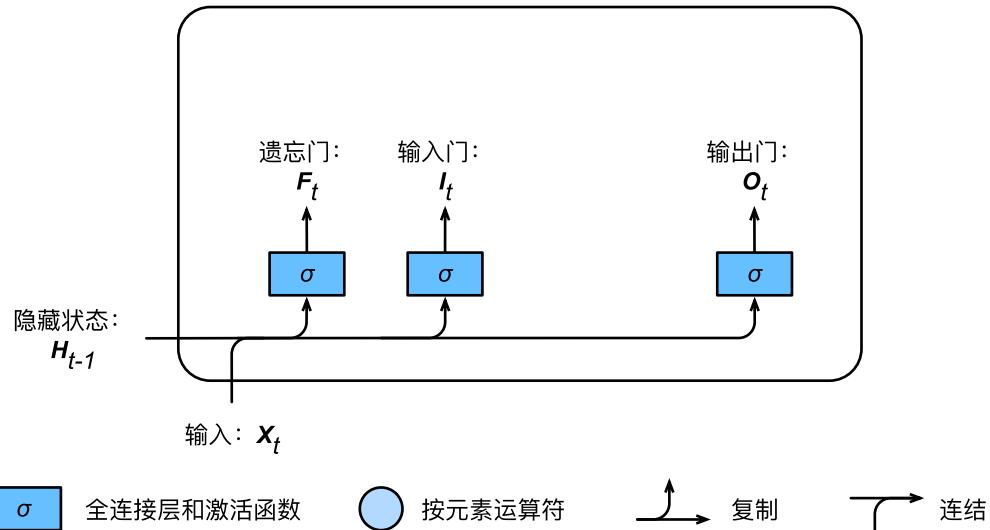
#### 1. 输入门、遗忘门和输出门

与门控循环单元中的重置门和更新门一样，如下图所示，长短期记忆的门的输入均为当前时间步输入 $\mathbf{X}_t$ 与上一时间步隐藏状态 $\mathbf{H}_{t-1}$ ，输出由激活函数为 sigmoid 函数的全连接层计算得到。如此一来，这3个门元素的值域均为 $[0, 1]$ 。

具体来说，假设隐藏单元个数为 $h$ ，给定时间步 $t$ 的小批量输入 $\mathbf{X}_t \in \mathbb{R}^{n \times d}$  (样本数为 $n$ ，输入个数为 $d$ ) 和上一时间步隐藏状态 $\mathbf{H}_{t-1} \in \mathbb{R}^{n \times h}$ 。时间步 $t$ 的输入门 $\mathbf{I}_t \in \mathbb{R}^{n \times h}$ 、遗忘门 $\mathbf{F}_t \in \mathbb{R}^{n \times h}$ 和输出门 $\mathbf{O}_t \in \mathbb{R}^{n \times h}$ 分别计算如下：

$$\begin{aligned}\mathbf{I}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xi} + \mathbf{H}_{t-1} \mathbf{W}_{hi} + \mathbf{b}_i) \\ \mathbf{F}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xf} + \mathbf{H}_{t-1} \mathbf{W}_{hf} + \mathbf{b}_f) \\ \mathbf{O}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xo} + \mathbf{H}_{t-1} \mathbf{W}_{ho} + \mathbf{b}_o)\end{aligned}$$

其中的 $\mathbf{W}_{xi}, \mathbf{W}_{xf}, \mathbf{W}_{xo} \in \mathbb{R}^{d \times h}$ 和 $\mathbf{W}_{hi}, \mathbf{W}_{hf}, \mathbf{W}_{ho} \in \mathbb{R}^{h \times h}$ 是权重参数， $\mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_o \in \mathbb{R}^{1 \times h}$ 是偏差参数。



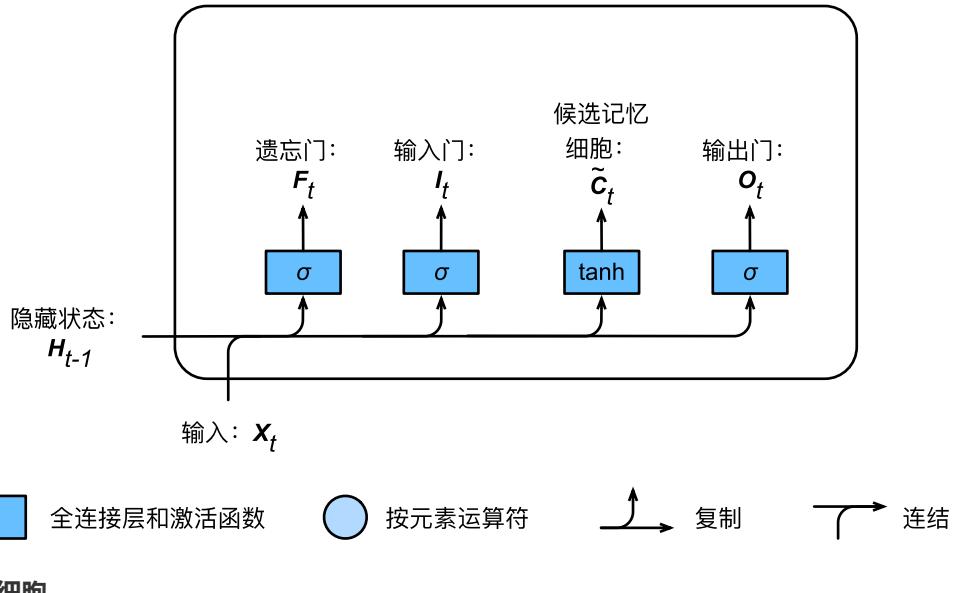
#### 2. 候选记忆细胞

接下来，长短期记忆需要计算候选记忆细胞 $\tilde{\mathbf{C}}_t$ 。它的计算与上面介绍的3个门类似，但使用了值域在 $[-1, 1]$ 的 tanh 函数作为激活函数，如下图所示。

具体来说，时间步 $t$ 的候选记忆细胞 $\tilde{\mathbf{C}}_t \in \mathbb{R}^{n \times h}$ 的计算为

$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xc} + \mathbf{H}_{t-1} \mathbf{W}_{hc} + \mathbf{b}_c)$$

其中 $\mathbf{W}_{xc} \in \mathbb{R}^{d \times h}$ 和 $\mathbf{W}_{hc} \in \mathbb{R}^{h \times h}$ 是权重参数， $\mathbf{b}_c \in \mathbb{R}^{1 \times h}$ 是偏差参数。

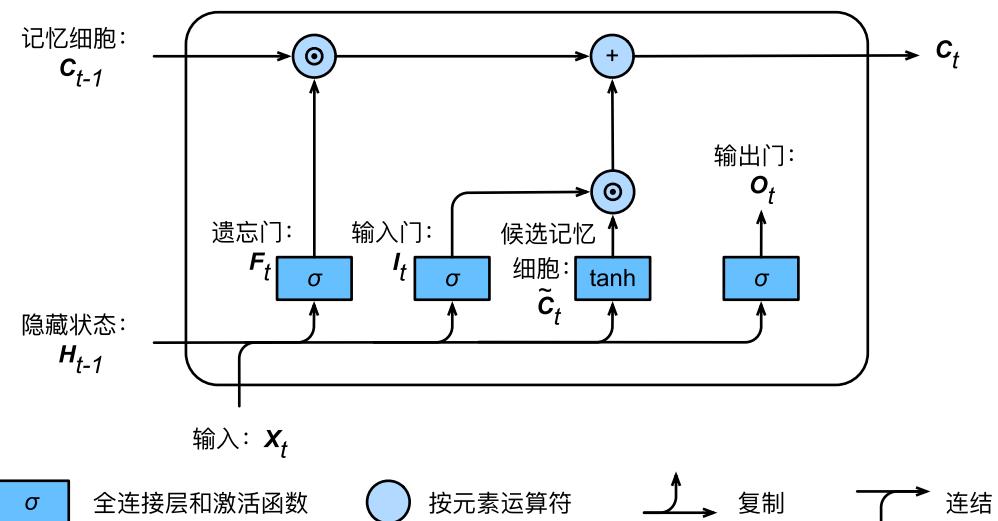


### 3. 记忆细胞

我们可以通过元素值域在 $[0, 1]$ 的输入门、遗忘门和输出门来控制隐藏状态中信息的流动，这一般也是通过使用按元素乘法（符号为 $\odot$ ）来实现的。当前时间步记忆细胞 $C_t \in \mathbb{R}^{n \times h}$ 的计算组合了上一时间步记忆细胞和当前时间步候选记忆细胞的信息，并通过遗忘门和输入门来控制信息的流动：

$$C_t = F_t \odot C_{t-1} + I_t \odot \tilde{C}_t$$

如下图所示，遗忘门控制上一时间步的记忆细胞 $C_{t-1}$ 中的信息是否传递到当前时间步，而输入门则控制当前时间步的输入 $X_t$ 通过候选记忆细胞 $\tilde{C}_t$ 如何流入当前时间步的记忆细胞。如果遗忘门一直近似1且输入门一直近似0，过去的记忆细胞将一直通过时间保存并传递至当前时间步。这个设计可以应对循环神经网络中的梯度衰减问题，并更好地捕捉时间序列中时间步距离较大的依赖关系

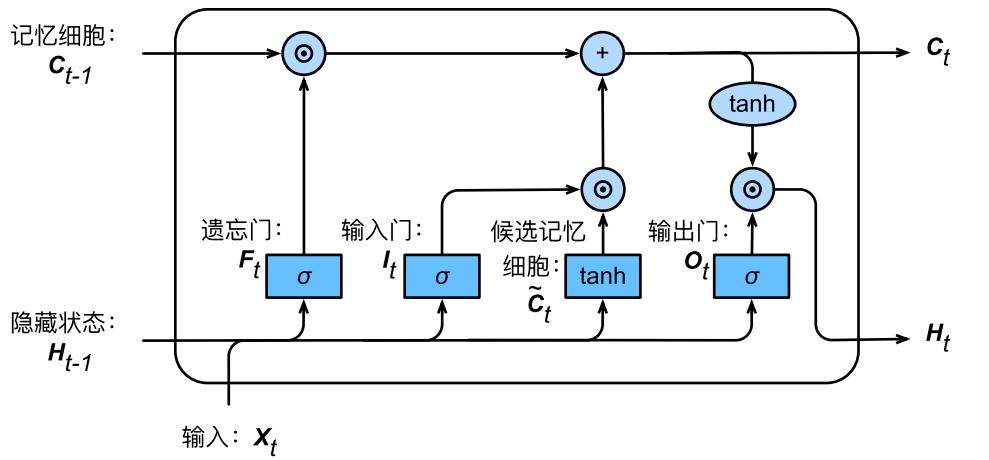


### 4. 隐藏状态

有了记忆细胞以后，接下来我们还可以通过输出门来控制从记忆细胞到隐藏状态 $H_t \in \mathbb{R}^{n \times h}$ 的信息的流动：

$$H_t = O_t \odot \tanh(C_t)$$

这里的  $\tanh$  函数确保隐藏状态元素值在-1到1之间。需要注意的是，当输出门近似1时，记忆细胞信息将传递到隐藏状态供输出层使用；当输出门近似0时，记忆细胞信息只自己保留。下图展示了长短期记忆中隐藏状态的计算。



## 5.8.2 读取数据集

下面我们开始实现并展示长短期记忆。和前几节中的实验一样，这里依然使用周杰伦歌词数据集来训练模型作词。

```

1 In [1]:
2     (corpus_indices, char_to_idx,
3      idx_to_char, vocab_size) = d2l.load_data_jay_lyrics()

```

## 5.8.3 从零开始实现

我们先介绍如何从零开始实现长短期记忆。

### 1. 初始化模型参数

下面的代码对模型参数进行初始化。超参数 `num_hiddens` 定义了隐藏单元的个数。

```

1 In [2]:
2     num_inputs, num_hiddens, num_outputs = vocab_size, 256, vocab_size
3     print('will use', device)
4     def get_params():
5         def _one(shape):
6             ts = torch.tensor(np.random.normal(0, 0.01, size=shape),
7                               device=device, dtype=torch.float32)
8             return torch.nn.Parameter(ts, requires_grad=True)
9         def _three():
10            return (_one((num_inputs, num_hiddens)),
11                    _one((num_hiddens, num_hiddens)),
12                    torch.nn.Parameter(
13                        torch.zeros(num_hiddens,
14                                    device=device,
15                                    dtype=torch.float32),
16                        requires_grad=True))
17
18         # 输入门参数
19         w_xi, w_hi, b_i = _three()
20         # 遗忘门参数
21         w_xf, w_hf, b_f = _three()
22         # 输出门参数
23         w_xo, w_ho, b_o = _three()
24         # 候选记忆细胞参数

```

```

24         w_xc, w_hc, b_c = _three()
25         # 输出层参数
26         w_hq = _one((num_hiddens, num_outputs))
27         b_q = torch.nn.Parameter(
28             torch.zeros(num_outputs, device=device,
29                         dtype=torch.float32),
30             requires_grad=True)
31         return nn.ParameterList([w_xi, w_hi, b_i,
32                                 w_xf, w_hf, b_f,
33                                 w_xo, w_ho, b_o,
34                                 w_xc, w_hc, b_c,
35                                 w_hq, b_q])
36 Out [2]:
37      will use cuda

```

## 2. 定义模型

在初始化函数中，长短期记忆的隐藏状态需要返回额外的形状为（批量大小, 隐藏单元个数）的值为0的记忆细胞。

```

1 In [3]:
2     def init_lstm_state(batch_size, num_hiddens, device):
3         return (torch.zeros((batch_size, num_hiddens), device=device),
4                 torch.zeros((batch_size, num_hiddens), device=device))

```

下面根据长短期记忆的计算表达式定义模型。需要注意的是，只有隐藏状态会传递到输出层，而记忆细胞不参与输出层的计算。

```

1 In [4]:
2     def lstm(inputs, state, params):
3         [w_xi, w_hi, b_i, w_xf, w_hf, b_f,
4          w_xo, w_ho, b_o, w_xc, w_hc, b_c, w_hq, b_q] = params
5         (H, C) = state
6         outputs = []
7         for x in inputs:
8             I = torch.sigmoid(
9                 torch.matmul(x.float(), w_xi) + torch.matmul(H, w_hi) + b_i)
10            F = torch.sigmoid(
11                torch.matmul(x.float(), w_xf) + torch.matmul(H, w_hf) + b_f)
12            O = torch.sigmoid(
13                torch.matmul(x.float(), w_xo) + torch.matmul(H, w_ho) + b_o)
14            C_tilda = torch.tanh(
15                torch.matmul(x.float(), w_xc) + torch.matmul(H, w_hc) + b_c)
16            C = F * C + I * C_tilda
17            H = O * C.tanh()
18            Y = torch.matmul(H, w_hq) + b_q
19            outputs.append(Y)
20        return outputs, (H, C)

```

## 3. 训练模型并创作歌词

同上一节一样，我们在训练模型时只使用相邻采样。设置好超参数后，我们将训练模型并根据前缀“分开”和“不分开”分别创作长度为50个字符的一段歌词。

```
1 In [5]:  
2     num_epochs, num_steps, batch_size, lr = 160, 35, 32, 1e-2  
3     clipping_theta = 1e-2  
4     pred_period, pred_len, prefixes = 40, 50, ['分开', '不分开']
```

我们每过40个迭代周期便根据当前训练的模型创作一段歌词。

```
1 In [6]:  
2     d2l.train_and_predict_rnn(lstm, get_params, init_lstm_state,  
3                                 num_hiddens, vocab_size, device,  
4                                 corpus_indices, idx_to_char,  
5                                 char_to_idx, False, num_epochs,  
6                                 num_steps, lr, clipping_theta,  
7                                 batch_size, pred_period, pred_len,  
8                                 prefixes)  
9 Out [6]:  
10    epoch 40, perplexity 213.158646, time 2.46 sec  
11    - 分开 我不的我 我不的我 我不的我 我不的我 我不的我 我不的我  
12        我不的我 我不的我 我不的我 我不的我  
13    - 不分开 我不的我 我不的我 我不的我 我不的我 我不的我 我不的我  
14        我不的我 我不的我 我不的我 我不的我  
15    epoch 80, perplexity 70.030242, time 2.48 sec  
16    - 分开 我想你你的你 我想想你 你不 我不要 我不要 我不要 我不要  
17        我不要 我不要 我不要 我不要 我  
18    - 不分开 我想你你 我不要 我不要 我不要 我不要 我不要 我不要 我不要  
19        我不要 我不要 我不要 我不要  
20    epoch 120, perplexity 17.365369, time 2.38 sec  
21    - 分开 我说你的你笑我 你样的风 我有你的爱你 你元人  
22        深埋在美索不达米亚 我想要你 爱样我 想样 没  
23    - 不分开 我知你这想堡堡 想想你的你笑着 像么  
24        有有你的肩着 我想想的你 我 我想你的你膀 没 在你  
25    epoch 160, perplexity 4.445391, time 2.43 sec  
26    - 分开 我说带的话笑 后悔在对不起 景色入秋 漫天了空凉我  
27        塞北的客栈人多 牧草有没有 我马儿有些瘦 天涯  
28    - 不分开 我已你这样笑着听 想是开 说说 一颗两颗 在上四碗  
29        在一上空的在老 还上上看看 让家 在什人 我开
```

## 5.8.4 简洁实现

在PyTorch中我们可以直接调用 `nn` 模块中的 `LSTM` 类。

```
1 In [7]:  
2     lr = 1e-2  
3     lstm_layer = nn.LSTM(input_size=vocab_size, hidden_size=num_hiddens)  
4     model = d2l.RNNModel(lstm_layer, vocab_size)  
5     d2l.train_and_predict_rnn_pytorch(model, num_hiddens, vocab_size,  
6                                         device, corpus_indices,  
7                                         idx_to_char, char_to_idx,  
8                                         num_epochs, num_steps, lr,  
9                                         clipping_theta, batch_size,  
10                                        pred_period, pred_len, prefixes)  
11 Out [7]:  
12    epoch 40, perplexity 1.020968, time 0.08 sec  
13    - 分开的太快就像龙卷风 离不开暴风圈来不及逃 我不能再想 我不能再想  
14        我不 我不 我不能 爱情走的太快就像  
15    - 不分开 我有轻功 飞檐走壁 为人耿直不屈 一身正气 他们儿子我习惯
```

```

16     从小就耳濡目染 什么刀枪跟棍棒 我都要
17 epoch 80, perplexity 1.014793, time 0.07 sec
18 - 分开的玩笑 想通 却又再考倒我 说散 你想很久了吧? 败给你的黑色幽默
19     说散 你想很久了吧? 我的认真败
20 - 不分开 我有多看着你 有话去对医药箱说 别怪我 别怪我 说你怎么面对我
21     甩开球我满腔的怒火 我想揍你已经很
22 epoch 120, perplexity 1.012057, time 0.07 sec
23 - 分开的玩笑 想通 却又再考倒我 说散 你想很久了吧? 败给你的黑色幽默
24     说散 你想很久了吧? 我的认真败
25 - 不分开 我有了空 是你完美演出的一场戏 宁愿心碎哭泣 再狠狠忘记
26     你爱过我的证据 让晶莹的泪滴 闪烁成回忆
27 epoch 160, perplexity 1.008711, time 0.07 sec
28 - 分开的快乐是你 想你想的都会笑 没有你在 我有多难熬
29     没有你在我有多难熬多烦恼 没有你烦 我有多烦恼
30 - 不分开 我有轻重 就是开不了口让她知道 就是那么简单几句
31     我办不到 整颗心悬在半空 我只能够远远看著 这些

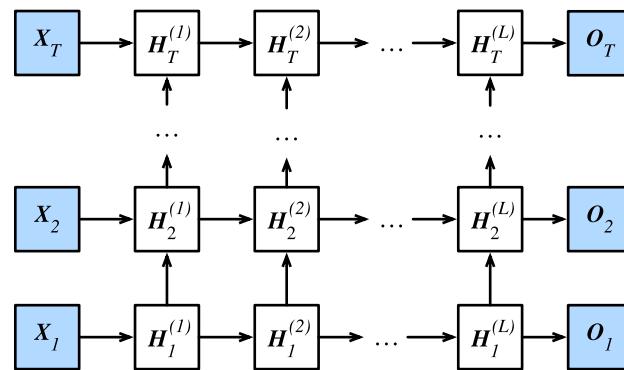
```

### 小结:

- 长短期记忆的隐藏层输出包括隐藏状态和记忆细胞。只有隐藏状态会传递到输出层。
- 长短期记忆的输入门、遗忘门和输出门可以控制信息的流动。
- 长短期记忆可以应对循环神经网络中的梯度衰减问题，并更好地捕捉时间序列中时间步距离较大的依赖关系。

## 5.9 深度循环神经网络

本章到目前为止介绍的循环神经网络只有一个单向的隐藏层，在深度学习应用里，我们通常会用到含有多个隐藏层的循环神经网络，也称作深度循环神经网络。下图演示了一个有 $L$ 个隐藏层的深度循环神经网络，**每个隐藏状态不断传递至当前层的下一时间步和当前时间步的下一层**。



具体来说，在时间步 $t$ 里，设小批量输入 $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ （样本数为 $n$ ，输入个数为 $d$ ），第 $\ell$ 隐藏层（ $\ell = 1, \dots, L$ ）的隐藏状态为 $\mathbf{H}_t^{(\ell)} \in \mathbb{R}^{n \times h}$ （隐藏单元个数为 $h$ ），输出层变量为 $\mathbf{O}_t \in \mathbb{R}^{n \times q}$ （输出个数为 $q$ ），且隐藏层的激活函数为 $\phi$ 。第1隐藏层的隐藏状态和之前的计算一样：

$$\mathbf{H}_t^{(1)} = \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(1)} + \mathbf{H}_{t-1}^{(1)} \mathbf{W}_{hh}^{(1)} + \mathbf{b}_h^{(1)})$$

其中权重 $\mathbf{W}_{xh}^{(1)} \in \mathbb{R}^{d \times h}$ 、 $\mathbf{W}_{hh}^{(1)} \in \mathbb{R}^{h \times h}$ 和偏差 $\mathbf{b}_h^{(1)} \in \mathbb{R}^{1 \times h}$ 分别为第1隐藏层的模型参数。

当 $1 < \ell \leq L$ 时，第 $\ell$ 隐藏层的隐藏状态的表达式为

$$\mathbf{H}_t^{(\ell)} = \phi(\mathbf{H}_t^{(\ell-1)} \mathbf{W}_{xh}^{(\ell)} + \mathbf{H}_{t-1}^{(\ell)} \mathbf{W}_{hh}^{(\ell)} + \mathbf{b}_h^{(\ell)})$$

其中权重 $\mathbf{W}_{xh}^{(\ell)} \in \mathbb{R}^{h \times h}$ 、 $\mathbf{W}_{hh}^{(\ell)} \in \mathbb{R}^{h \times h}$ 和偏差 $\mathbf{b}_h^{(\ell)} \in \mathbb{R}^{1 \times h}$ 分别为第 $\ell$ 隐藏层的模型参数。

最终，输出层的输出只需基于第 $L$ 隐藏层的隐藏状态：

$$\mathbf{O}_t = \mathbf{H}_t^{(L)} \mathbf{W}_{hq} + \mathbf{b}_q$$

其中权重  $\mathbf{W}_{hq} \in \mathbb{R}^{h \times q}$  和偏差  $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$  为输出层的模型参数。

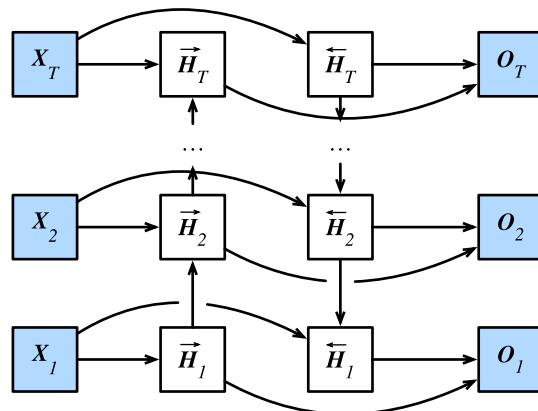
同多层感知机一样，隐藏层个数  $L$  和隐藏单元个数  $h$  都是超参数。此外，如果将隐藏状态的计算换成门控循环单元或者长短期记忆的计算，我们可以得到深度门控循环神经网络。

**小结：**

- 在深度循环神经网络中，隐藏状态的信息不断传递至当前层的下一时间步和当前时间步的下一层。

## 5.10 双向循环神经网络

之前介绍的循环神经网络模型都是假设当前时间步是由前面的较早时间步的序列决定的，因此它们都将信息通过隐藏状态从前往后传递。有时候，当前时间步也可能由后面时间步决定。例如，当我们写下一个句子时，可能会根据句子后面的词来修改句子前面的用词。双向循环神经网络通过增加**从后往前传递信息的隐藏层**来更灵活地处理这类信息。下图演示了一个含单隐藏层的双向循环神经网络的架构。



下面我们来介绍具体的定义。给定时间步  $t$  的小批量输入  $\mathbf{X}_t \in \mathbb{R}^{n \times d}$  (样本数为  $n$ , 输入个数为  $d$ ) 和隐藏层激活函数为  $\phi$ 。在双向循环神经网络的架构中，设该时间步正向隐藏状态为  $\vec{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$  (正向隐藏单元个数为  $h$ )，反向隐藏状态为  $\overleftarrow{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$  (反向隐藏单元个数为  $h$ )。我们可以分别计算正向隐藏状态和反向隐藏状态：

$$\begin{aligned}\vec{\mathbf{H}}_t &= \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(f)} + \vec{\mathbf{H}}_{t-1} \mathbf{W}_{hh}^{(f)} + \mathbf{b}_h^{(f)}) \\ \overleftarrow{\mathbf{H}}_t &= \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(b)} + \overleftarrow{\mathbf{H}}_{t+1} \mathbf{W}_{hh}^{(b)} + \mathbf{b}_h^{(b)})\end{aligned}$$

其中权重  $\mathbf{W}_{xh}^{(f)} \in \mathbb{R}^{d \times h}$ 、 $\mathbf{W}_{hh}^{(f)} \in \mathbb{R}^{h \times h}$ 、 $\mathbf{W}_{xh}^{(b)} \in \mathbb{R}^{d \times h}$ 、 $\mathbf{W}_{hh}^{(b)} \in \mathbb{R}^{h \times h}$  和偏差  $\mathbf{b}_h^{(f)} \in \mathbb{R}^{1 \times h}$ 、 $\mathbf{b}_h^{(b)} \in \mathbb{R}^{1 \times h}$  均为模型参数。

然后我们连结两个方向的隐藏状态  $\vec{\mathbf{H}}_t$  和  $\overleftarrow{\mathbf{H}}_t$  来得到隐藏状态  $\mathbf{H}_t \in \mathbb{R}^{n \times 2h}$ ，并将其输入到输出层。输出层计算输出  $\mathbf{O}_t \in \mathbb{R}^{n \times q}$  (输出个数为  $q$ )：

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hq} + \mathbf{b}_q$$

其中权重  $\mathbf{W}_{hq} \in \mathbb{R}^{2h \times q}$  和偏差  $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$  为输出层的模型参数。不同方向上的隐藏单元个数也可以不同。

**小结：**

- 双向循环神经网络在每个时间步的隐藏状态同时取决于该时间步之前和之后的子序列（包括当前时间步的输入）。

## 5.11 本章附录

☆ `zipFile` 对象

class zipfile.ZipFile(file, mode='r', compression=ZIP\_STORED, allowZip64=True)。作用为打开一个压缩文件，返回的也是一个类似文件的ZipFile对象，可以读写。参数 file 可以是一个文件地址字符串，mode 参数为 r 时，表示读取一个已经存在的文件；为 w 的时候表示覆盖或写入一个新文件；为 a 时表示在已有文件后追加；为 x 时表示新建文件并写入。compression 指明压缩格式。当文件大小超过 4GB 时，将使用 ZIP64 扩展（默认启用）。

### ☆ ZipFile.open() 的用法

ZipFile.open(name, mode='r', pwd=None, \*, force\_zip64=False)。该函数的作用是访问压缩包中的指定文件。pwd 是解压密码。r 模式为只读模式，提供 read()，readline()，readlines()，\_\_iter\_\_()，\_\_next\_\_() 等方法。w 模式为写入模式，支持 write() 方法。

### ☆ torch.transpose() 的用法

torch.transpose(input, dim0, dim1) → Tensor。在指定的两个维度之间对 input 进行转置。转置后的张量与原始张量共用存储空间，所以对一个张量进行修改会引起另一个张量的改变。

```
1 x = torch.randn(2, 3)
2 x
3 >>> tensor([[ 0.1118,  1.3875,  0.6013],
4             [ 0.1685,  0.1676, -0.0120]])
5 torch.transpose(x, 0, 1)
6 >>> tensor([[ 0.1118,  0.1685],
7             [ 1.3875,  0.1676],
8             [ 0.6013, -0.0120]])
```

### ☆ contiguous() 的用法

view 只能用在 contiguous 的 variable 上。如果在 view 之前用了 transpose，permute 等，需要用 contiguous() 来返回一个 contiguous copy。一种可能的解释是：有些 tensor 并不是占用一整块内存，而是由不同的数据块组成，而 tensor 的 view() 操作依赖于内存是整块的，这时只需要执行 contiguous() 这个函数，把 tensor 变成在内存中连续分布的形式。判断是否 contiguous 用 torch.Tensor.is\_contiguous() 函数。

```
1 x = torch.ones(10, 10)
2 x.is_contiguous()
3 >>> True
4 x.transpose(0, 1).is_contiguous()
5 >>> False
6 x.transpose(0, 1).contiguous().is_contiguous()
7 >>> True
```

### ☆ nn.RNN() 的用法

nn.RNN(input\_size, hidden\_size, num\_layers=1, nonlinearity=tanh, bias=True, batch\_first=False, dropout=0, bidirectional=False)。input\_size 输入特征的维度，一般 rnn 中输入的是词向量，那么 input\_size 就等于一个词向量的维度。hidden\_size 隐藏层神经元个数，或者也叫输出的维度（因为 rnn 输出为各个时间步上的隐藏状态）。num\_layers 网络的层数。nonlinearity 激活函数。bias 是否使用偏置。batch\_first 输入数据的形式，默认是 False，就是这样形式，(seq(num\_step), batch, input\_dim)，也就是将序列长度放在第一位，batch 放在第二位。dropout 是否应用 dropout，默认不使用，如若使用将其设置成一个 0-1 的数字即可。bidirectional 是否使用双向的 rnn，默认是 False。参考[博客](#)。

本章代码详见 GitHub 链接[wzy6642](#)。

# 第6章 优化算法

如果读者一直按照本书的顺序读到这里，那么一定已经使用了优化算法来训练深度学习模型。具体来说，在训练模型时，我们会使用优化算法不断迭代模型参数以降低模型损失函数的值。当迭代终止时，模型的训练随之终止，此时的模型参数就是模型通过训练所学习到的参数。

优化算法对于深度学习十分重要。一方面，训练一个复杂的深度学习模型可能需要数小时、数日，甚至数周时间，而优化算法的表现直接影响模型的训练效率；另一方面，理解各种优化算法的原理以及其中超参数的意义将有助于我们更有针对性地调参，从而使深度学习模型表现更好。

本章将详细介绍深度学习中常用的优化算法。

## 6.1 优化与深度学习

本节将讨论优化与深度学习的关系，以及优化在深度学习中的挑战。在一个深度学习问题中，我们通常会预先定义一个损失函数。有了损失函数以后，我们就可以使用优化算法试图将其最小化。在优化中，这样的损失函数通常被称作优化问题的目标函数（objective function）。依据惯例，优化算法通常只考虑最小化目标函数。其实，任何最大化问题都可以很容易地转化为最小化问题，只需令目标函数的相反数为新的目标函数即可。

### 6.1.1 优化与深度学习的关系

虽然优化为深度学习提供了最小化损失函数的方法，但本质上，优化与深度学习的目标是有区别的。在2.11节（模型选择、欠拟合和过拟合）中，我们区分了训练误差和泛化误差。由于优化算法的目标函数通常是一个基于训练数据集的损失函数，优化的目标在于降低训练误差。而深度学习的目标在于降低泛化误差。为了降低泛化误差，除了使用优化算法降低训练误差以外，还需要注意应对过拟合。

本章中，我们只关注优化算法在最小化目标函数上的表现，而不关注模型的泛化误差。

### 6.1.2 优化在深度学习中的挑战

我们在2.1节（线性回归）中对优化问题的解析解和数值解做了区分。深度学习中绝大多数目标函数都很复杂。因此，很多优化问题并不存在解析解，而需要使用基于数值方法的优化算法找到近似解，即数值解。本书中讨论的优化算法都是这类基于数值方法的算法。为了求得最小化目标函数的数值解，我们将通过优化算法有限次迭代模型参数来尽可能降低损失函数的值。

优化在深度学习中有很多挑战。下面描述了其中的两个挑战，即局部最小值和鞍点。为了更好地描述问题，我们先导入本节中实验需要的包或模块。

```
1 In [1]:  
2     import d2lzh as d2l  
3     import numpy as np  
4     # 三维画图  
5     from mpl_toolkits import mplot3d
```

#### 1. 局部最小值

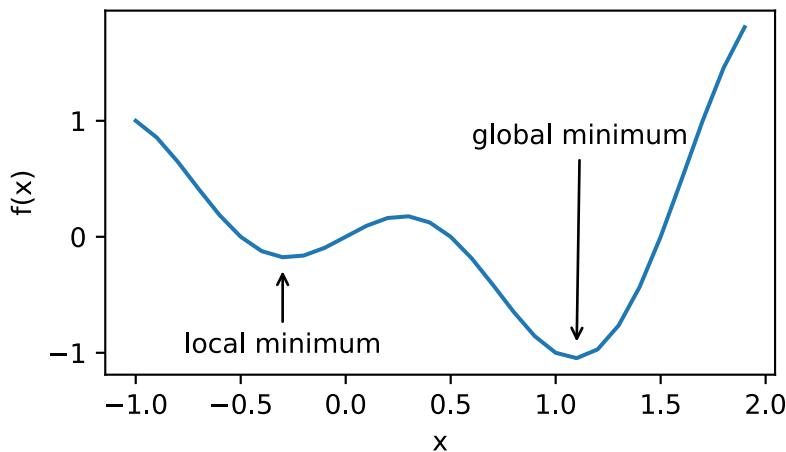
对于目标函数 $f(x)$ ，如果 $f(x)$ 在 $x$ 上的值比在 $x$ 邻近的其他点的值更小，那么 $f(x)$ 可能是一个局部最小值（local minimum）。如果 $f(x)$ 在 $x$ 上的值是目标函数在整个定义域上的最小值，那么 $f(x)$ 是全局最小值（global minimum）。

举个例子，给定函数

$$f(x) = x \cdot \cos(\pi x), \quad -1.0 \leq x \leq 2.0$$

我们可以大致找出该函数的局部最小值和全局最小值的位置。需要注意的是，图中箭头所指示的只是大致位置。

```
1 In [2]:  
2     def f(x):  
3         return x * np.cos(np.pi * x)  
4     d2l.set_figsize((4.5, 2.5))  
5     x = np.arange(-1.0, 2.0, 0.1)  
6     # 逗号表示只取返回列表中的第一个元素  
7     fig, = d2l.plt.plot(x, f(x))  
8     fig.axes.annotate('local minimum', xy=(-0.3, -0.25),  
9                         xytext=(-0.77, -1.0),  
10                        arrowprops=dict(arrowstyle='->'))  
11    fig.axes.annotate('global minimum', xy=(1.1, -0.95),  
12                         xytext=(0.6, 0.8),  
13                        arrowprops=dict(arrowstyle='->'))  
14    d2l.plt.xlabel('x')  
15    d2l.plt.ylabel('f(x)')
```



深度学习模型的目标函数可能有若干局部最优值。当一个优化问题的数值解在局部最优解附近时，由于目标函数有关解的梯度接近或变成零，最终迭代求得的数值解可能只令目标函数局部最小化而非全局最小化。

## 2. 鞍点

刚刚我们提到，梯度接近或变成零可能是由于当前解在局部最优解附近造成的。事实上，另一种可能性是当前解在鞍点 (saddle point) 附近。一个不是局部最小值的驻点（一阶导数为0的点）称为鞍点。数学含义是：目标函数在此点上的梯度（一阶导数）值为0，**但从该点出发的一个方向是函数的极大值点，而在另一个方向是函数的极小值点**。

举个例子，给定函数

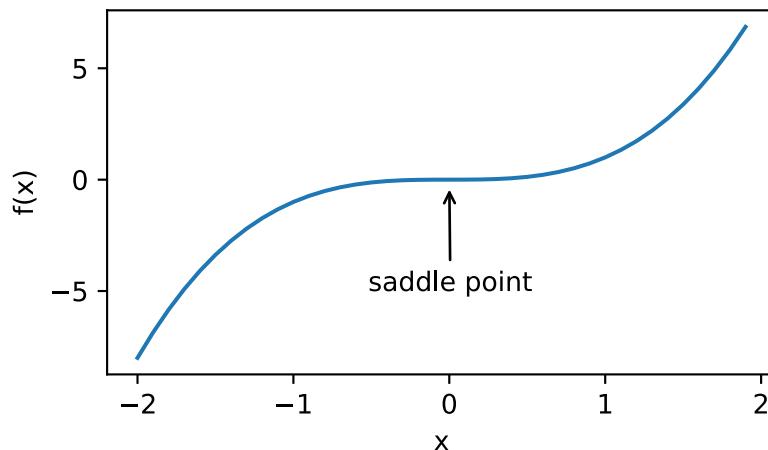
$$f(x) = x^3$$

我们可以找出该函数的鞍点位置。

```

1 In [3]:
2     x = np.arange(-2.0, 2.0, 0.1)
3     fig, = d2l.plt.plot(x, x**3)
4     # xy:箭头坐标, xytext:文字添加位置
5     fig.axes.annotate('saddle point', xy=(0, -0.2),
6                         xytext=(-0.52, -5.0),
7                         arrowprops=dict(arrowstyle='->'))
8     d2l.plt.xlabel('x')
9     d2l.plt.ylabel('f(x)')

```



再举个定义在二维空间的函数的例子，例如：

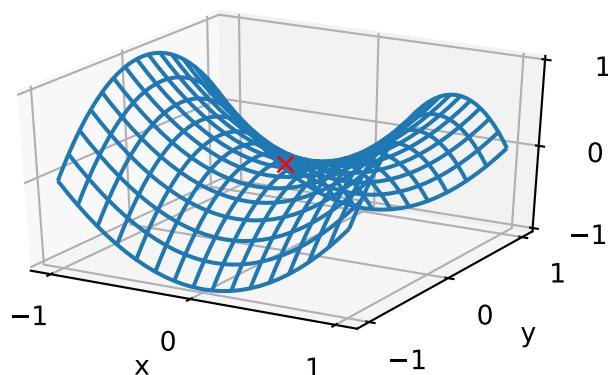
$$f(x, y) = x^2 - y^2$$

我们可以找出该函数的鞍点位置。也许你已经发现了，该函数看起来像一个马鞍，而鞍点恰好是马鞍上可坐区域的中心。

```

1 In [4]:
2     x, y = np.mgrid[-1:1:31j, -1:1:31j]
3     z = x**2 - y**2
4     ax = d2l.plt.figure().add_subplot(111, projection='3d')
5     ax.plot_wireframe(x, y, z, **{'rstride': 2, 'cstride': 2})
6     ax.plot([0], [0], [0], 'rx')
7     ticks = [-1, 0, 1]
8     d2l.plt.xticks(ticks)
9     d2l.plt.yticks(ticks)
10    ax.set_zticks(ticks)
11    d2l.plt.xlabel('x')
12    d2l.plt.ylabel('y')

```



在图的鞍点位置，目标函数在 $x$ 轴方向上是局部最小值，但在 $y$ 轴方向上是局部最大值。

假设一个函数的输入为 $k$ 维向量，输出为标量，那么它的海森矩阵（Hessian matrix）有 $k$ 个特征值。该函数在梯度为0的位置上可能是局部最小值、局部最大值或者鞍点：

- 当函数的海森矩阵在梯度为0的位置上的特征值全为正时，该函数得到局部最小值。
- 当函数的海森矩阵在梯度为0的位置上的特征值全为负时，该函数得到局部最大值。
- 当函数的海森矩阵在梯度为0的位置上的特征值有正有负时，该函数得到鞍点。

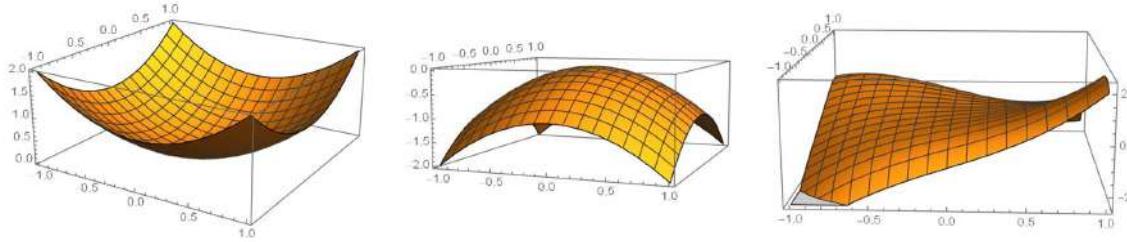
举例如下，我们分别构建以下3个函数进行说明：

$$f(x) = x^2 + y^2 \quad (1)$$

$$f(x) = -x^2 - y^2 \quad (2)$$

$$f(x) = x^3 - 2 \times x \times y - y^6 \quad (3)$$

我们依次对其函数图像进行绘制：



在数学中，海森矩阵(Hessian matrix) 是一个自变量为向量的实值函数的二阶偏导数组成的方块矩阵，假设有一个实数函数

$$f(x_1, x_2, \dots, x_n)$$

如果  $f$  所有的二阶偏导数都存在，那么  $f$  的海森矩阵第  $ij$  项，即：

$$H(f)_{ij}(x) = D_i D_j f(x)$$

其中  $x = (x_1, x_2, \dots, x_n)$ ，即：

$$H(f) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

根据上述知识，我们可以求得第一个函数的海森矩阵：

$$\frac{\partial f}{\partial x} = 2x, \frac{\partial^2 f}{\partial x \partial y} = 0, \frac{\partial f}{\partial y} = 2y, \frac{\partial^2 f}{\partial y \partial x} = 0, \frac{\partial^2 f}{\partial x^2} = 2, \frac{\partial^2 f}{\partial y^2} = 2$$

$$H(f) = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$$

我们令一阶导数为0可以解得  $x = 0, y = 0$ ，此时海森矩阵的两个特征值分别为 2, 2，均大于零，所以该函数在  $(0, 0)$  处取得局部最小值。其它两个函数的分析方法如上。

随机矩阵理论告诉我们，对于一个大的高斯随机矩阵来说，任一特征值是正或者是负的概率都是 0.5。那么，以上第一种情况的概率为  $0.5^k$ 。由于深度学习模型参数通常都是高维的 ( $k$ 很大)，目标函数的鞍点通常比局部最小值更常见。

在深度学习中，虽然找到目标函数的全局最优解很难，但这并非必要。我们将在本章接下来的几节中逐一介绍深度学习中常用的优化算法，它们在很多实际问题中都能够训练出十分有效的深度学习模型。

### 小结：

- 由于优化算法的目标函数通常是一个基于训练数据集的损失函数，优化的目标在于降低训练误差。
- 由于深度学习模型参数通常都是高维的，目标函数的鞍点通常比局部最小值更常见。

## 6.2 梯度下降和随机梯度下降

在本节中，我们将介绍梯度下降（gradient descent）的工作原理。虽然梯度下降在深度学习中很少被直接使用，但理解梯度的意义以及沿着梯度反方向更新自变量可能降低目标函数值的原因是学习后续优化算法的基础。随后，我们将引出随机梯度下降（stochastic gradient descent）。

### 6.2.1 一维梯度下降

我们先以简单的一维梯度下降为例，解释梯度下降算法可能降低目标函数值的原因。假设连续可导的函数  $f: \mathbb{R} \rightarrow \mathbb{R}$  的输入和输出都是标量。给定绝对值足够小的数  $\epsilon$ ，根据泰勒展开公式，我们得到以下的近似：

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!} (x-a)^n, \text{ 我们令 } x = x + \epsilon, a = x \text{ 可得 :}$$

$$f(x+\epsilon) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x)}{n!} (\epsilon)^n, \text{ 对其展开可得 :}$$

$$f(x+\epsilon) \approx f(x) + f'(x)\epsilon + O(\epsilon^2), \text{ 由于 } \epsilon \text{ 足够小, 所以可化简为 :}$$

$$f(x+\epsilon) \approx f(x) + f'(x)\epsilon$$

这里  $f'(x)$  是函数  $f$  在  $x$  处的梯度。一维函数的梯度是一个标量，也称导数。

接下来，找到一个常数  $\eta > 0$ ，使得  $|\eta f'(x)|$  足够小，那么可以将  $\epsilon$  替换为  $-\eta f'(x)$  并得到

$$f(x - \eta f'(x)) \approx f(x) - \eta f'(x)^2$$

如果导数  $f'(x) \neq 0$ ，那么  $\eta f'(x)^2 > 0$ ，所以

$$f(x - \eta f'(x)) \lesssim f(x)$$

这意味着，如果通过

$$x \leftarrow x - \eta f'(x)$$

来迭代  $x$ ，函数  $f(x)$  的值可能会降低。因此在梯度下降中，我们先选取一个初始值  $x$  和常数  $\eta > 0$ ，然后不断通过上式来迭代  $x$ ，直到达到停止条件，例如  $f'(x)^2$  的值已足够小或迭代次数已达到某个值。

下面我们以目标函数  $f(x) = x^2$  为例来看一看梯度下降是如何工作的。虽然我们知道最小化  $f(x)$  的解为  $x = 0$ ，这里依然使用这个简单函数来观察  $x$  是如何被迭代的。首先，导入本节实验所需的包或模块。

```
1 | In [1]:  
2 |     import math
```

接下来使用  $x = 10$  作为初始值，并设  $\eta = 0.2$ 。使用梯度下降对  $x$  迭代 10 次，可见最终  $x$  的值较接近最优解。

```

1 In [2]:
2     def gd(eta):
3         x = 10
4         results = [x]
5         for i in range(10):
6             # f(x) = x * x 的导数为f'(x) = 2 * x
7             x -= eta * 2 * x
8             results.append(x)
9             print('epoch 10, x:', x)
10            return results
11 res = gd(0.2)
12 Out [2]:
13 epoch 10, x: 0.06046617599999997

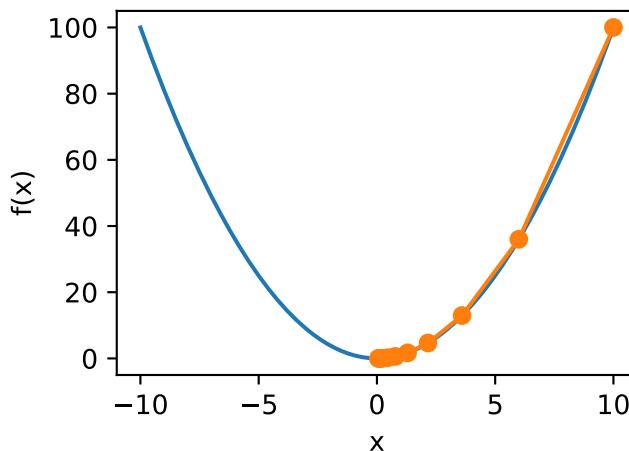
```

下面将绘制出自变量 $x$ 的迭代轨迹。

```

1 In [3]:
2     def show_trace(res):
3         n = max(abs(min(res)), abs(max(res)), 10)
4         # x的取值
5         f_line = np.arange(-n, n, 0.1)
6         d2l.set_figsize()
7         d2l.plt.plot(f_line, [x*x for x in f_line])
8         d2l.plt.plot(res, [x*x for x in res], '-o')
9         d2l.plt.xlabel('x')
10        d2l.plt.ylabel('f(x)')
11    show_trace(res)

```



## 6.2.2 学习率

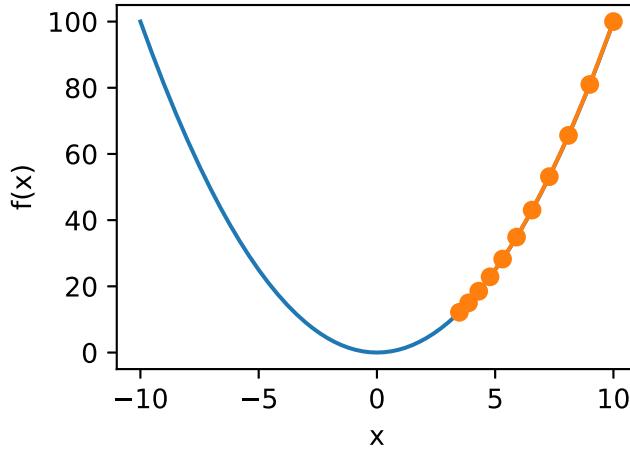
上述梯度下降算法中的正数 $\eta$ 通常叫作学习率。这是一个超参数，需要人工设定。如果使用过小的学习率，会导致 $x$ 更新缓慢从而需要更多的迭代才能得到较好的解。

下面展示使用学习率 $\eta = 0.05$ 时自变量 $x$ 的迭代轨迹。可见，同样迭代10次后，当学习率太小时，最终 $x$ 的值依然与最优解存在较大偏差。

```

1 In [4]:
2     show_trace(gd(0.05))
3 Out [4]:
4     epoch 10, x: 3.4867844009999995

```



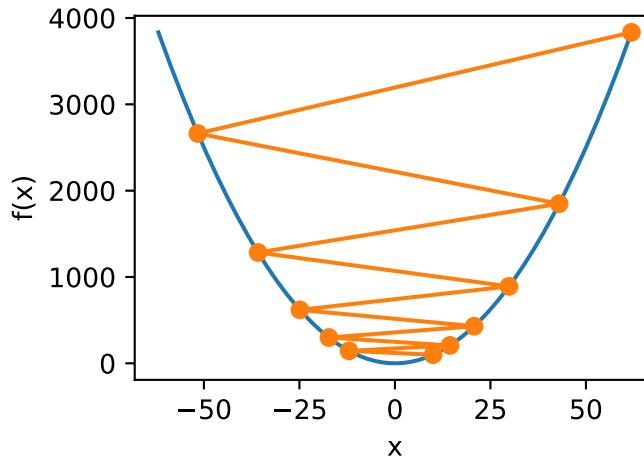
如果使用过大的学习率， $|\eta f'(x)|$ 可能会过大从而使前面提到的一阶泰勒展开公式不再成立：这时我们无法保证迭代 $x$ 会降低 $f(x)$ 的值。

举个例子，当设学习率 $\eta = 1.1$ 时，可以看到 $x$ 不断越过（overshoot）最优解 $x = 0$ 并逐渐发散。

```

1 In [5]:
2     show_trace(gd(1.1))
3 Out [5]:
4     epoch 10, x: 61.917364224000096

```



### 6.2.3 多维梯度下降

在了解了一维梯度下降之后，我们再考虑一种更广义的情况：目标函数的输入为向量，输出为标量。假设目标函数 $f: \mathbb{R}^d \rightarrow \mathbb{R}$ 的输入是一个 $d$ 维向量 $\mathbf{x} = [x_1, x_2, \dots, x_d]^\top$ 。目标函数 $f(\mathbf{x})$ 有关 $\mathbf{x}$ 的梯度是一个由 $d$ 个偏导数组成的向量：

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \left[ \frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \dots, \frac{\partial f(\mathbf{x})}{\partial x_d} \right]^\top$$

为表示简洁，我们用 $\nabla f(\mathbf{x})$ 代替 $\nabla_{\mathbf{x}} f(\mathbf{x})$ 。梯度中每个偏导数元素 $\partial f(\mathbf{x}) / \partial x_i$ 代表着 $f$ 在 $\mathbf{x}$ 有关输入 $x_i$ 的变化率。为了测量 $f$ 沿着单位向量 $\mathbf{u}$ （即 $\|\mathbf{u}\| = 1$ ）方向上的变化率，在多元微积分中，我们定义 $f$ 在 $\mathbf{x}$ 上沿着 $\mathbf{u}$ 方向的方向导数为

$$D_{\mathbf{u}} f(\mathbf{x}) = \lim_{h \rightarrow 0} \frac{f(\mathbf{x} + h\mathbf{u}) - f(\mathbf{x})}{h}$$

依据方向导数性质（参见本章附录），以上方向导数可以改写为

$$D_{\mathbf{u}} f(\mathbf{x}) = \nabla f(\mathbf{x}) \cdot \mathbf{u}$$

方向导数 $D_u f(\mathbf{x})$ 给出了 $f$ 在 $\mathbf{x}$ 上沿着所有可能方向的变化率。为了最小化 $f$ , 我们希望找到 $f$ 能被降低最快的方向。因此, 我们可以通过单位向量 $\mathbf{u}$ 来最小化方向导数 $D_u f(\mathbf{x})$ 。

由于 $D_u f(\mathbf{x}) = \|\nabla f(\mathbf{x})\| \cdot \|\mathbf{u}\| \cdot \cos(\theta) = \|\nabla f(\mathbf{x})\| \cdot \cos(\theta)$ , 其中 $\theta$ 为梯度 $\nabla f(\mathbf{x})$ 和单位向量 $\mathbf{u}$ 之间的夹角, 当 $\theta = \pi$ 时,  $\cos(\theta)$ 取得最小值 $-1$ 。因此, 当 $\mathbf{u}$ 在梯度方向 $\nabla f(\mathbf{x})$ 的相反方向时, 方向导数 $D_u f(\mathbf{x})$ 被最小化。因此, 我们可能通过梯度下降算法来不断降低目标函数 $f$ 的值:

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \nabla f(\mathbf{x})$$

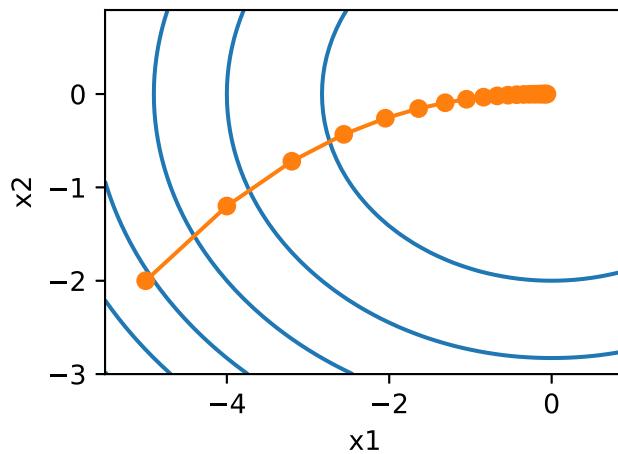
同样, 其中 $\eta$  (取正数) 称作学习率。

下面我们构造一个输入为二维向量 $\mathbf{x} = [x_1, x_2]^\top$ 和输出为标量的目标函数 $f(\mathbf{x}) = x_1^2 + 2x_2^2$ 。那么, 梯度 $\nabla f(\mathbf{x}) = [2x_1, 4x_2]^\top$ 。我们将观察梯度下降从初始位置 $[-5, -2]$ 开始对自变量 $\mathbf{x}$ 的迭代轨迹。我们先定义两个辅助函数, 第一个函数使用给定的自变量更新函数, 从初始位置 $[-5, -2]$ 开始迭代自变量 $\mathbf{x}$ 共20次, 第二个函数对自变量 $\mathbf{x}$ 的迭代轨迹进行可视化。

```
1 In [6]:  
2     def train_2d(trainer):  
3         # s1和s2是自变量状态, 本章后续几节会使用  
4         x1, x2, s1, s2 = -5, -2, 0, 0  
5         results = [(x1, x2)]  
6         for i in range(20):  
7             x1, x2, s1, s2 = trainer(x1, x2, s1, s2)  
8             results.append((x1, x2))  
9             print('epoch %d, x1 %f, x2 %f' % (i+1, x1, x2))  
10            return results  
11  
12    def show_trace_2d(f, results):  
13        d2l.plt.plot(*zip(*results), '-o', color='#ff7f0e')  
14        x1, x2 = np.meshgrid(  
15            np.arange(-5.5, 1.0, 0.1), np.arange(-3.0, 1.0, 0.1))  
16        # 这个函数主要对网格中每个点的值等于一系列值的时候做出一条条轮廓线  
17        # 类似于等高线  
18        d2l.plt.contour(x1, x2, f(x1, x2), colors='#1f77b4')  
19        d2l.plt.xlabel('x1')  
20        d2l.plt.ylabel('x2')
```

然后, 观察学习率为0.1时自变量的迭代轨迹。使用梯度下降对自变量 $\mathbf{x}$ 迭代20次后, 可见最终 $\mathbf{x}$ 的值较接近最优解 $[0, 0]$ 。

```
1 In [7]:  
2     eta = 0.1  
3     # 目标函数  
4     def f_2d(x1, x2):  
5         return x1**2+2*x2**2  
6     def gd_2d(x1, x2, s1, s2):  
7         return (x1-eta*2*x1, x2-eta*4*x2, 0, 0)  
8     show_trace_2d(f_2d, train_2d(gd_2d))  
9 Out [7]:  
10    epoch 20, x1 -0.057646, x2 -0.000073
```



### 6.2.4 随机梯度下降

在深度学习里，目标函数通常是训练数据集中有关各个样本的损失函数的平均。设 $f_i(\mathbf{x})$ 是有关索引为*i*的训练数据样本的损失函数，*n*是训练数据样本数， $\mathbf{x}$ 是模型的参数向量，那么目标函数定义为

$$f(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^n f_i(\mathbf{x})$$

目标函数在 $\mathbf{x}$ 处的梯度计算为

$$\nabla f(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^n \nabla f_i(\mathbf{x})$$

如果使用梯度下降，每次自变量迭代的计算开销为 $\mathcal{O}(n)$ ，它随着*n*线性增长。因此，当训练数据样本数很大时，梯度下降每次迭代的计算开销很高。

随机梯度下降 (stochastic gradient descent, SGD) 减少了每次迭代的计算开销。在随机梯度下降的每次迭代中，我们随机均匀采样一个样本索引 $i \in 1, \dots, n$ ，并计算梯度 $\nabla f_i(\mathbf{x})$ 来迭代 $\mathbf{x}$ ：

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \nabla f_i(\mathbf{x})$$

这里 $\eta$ 同样是学习率。可以看到每次迭代的计算开销从梯度下降的 $\mathcal{O}(n)$ 降到了常数 $\mathcal{O}(1)$ 。值得强调的是，随机梯度 $\nabla f_i(\mathbf{x})$ 是对梯度 $\nabla f(\mathbf{x})$ 的无偏估计：

$$E_i \nabla f_i(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^n \nabla f_i(\mathbf{x}) = \nabla f(\mathbf{x})$$

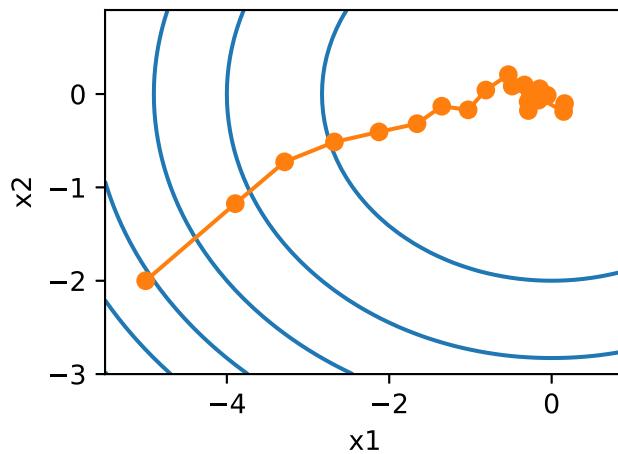
这意味着，平均来说，随机梯度是对梯度的一个良好的估计。

下面我们通过在梯度中添加均值为0的随机噪声来模拟随机梯度下降，以此来比较它与梯度下降的区别。

```

1 In [8]:
2     def sgd_2d(x1, x2, s1, s2):
3         return (x1-eta*(2*x1+np.random.normal(0.1)),
4                 x2-eta*(4*x2+np.random.normal(0.1)), 0, 0)
5     show_trace_2d(f_2d, train_2d(sgd_2d))
6 Out [8]:
7     epoch 20, x1 -0.150196, x2 0.029384

```



### 小结:

- 使用适当的学习率，沿着梯度反方向更新自变量可能降低目标函数值。梯度下降重复这一更新过程直到得到满足要求的解。
- 学习率过大或过小都有问题。一个合适的学习率通常是需要通过多次实验找到的。
- 当训练数据集的样本较多时，梯度下降每次迭代的计算开销较大，因而随机梯度下降通常更受青睐。

## 6.3 小批量随机梯度下降

在每一次迭代中，梯度下降使用整个训练数据集来计算梯度，因此它有时也被称为批量梯度下降 (batch gradient descent)。而随机梯度下降在每次迭代中只随机采样一个样本来计算梯度。正如我们在前几章中所看到的，我们还可以在每轮迭代中随机均匀采样多个样本来组成一个小批量，然后使用这个小批量来计算梯度。下面就来描述小批量随机梯度下降。

设目标函数  $f(\mathbf{x}) : \mathbb{R}^d \rightarrow \mathbb{R}$ 。在迭代开始前的时间步设为0。该时间步的自变量记为  $\mathbf{x}_0 \in \mathbb{R}^d$ ，通常由随机初始化得到。在接下来的每一个时间步  $t > 0$  中，小批量随机梯度下降随机均匀采样一个由训练数据样本索引组成的小批量  $\mathcal{B}_t$ 。我们可以通过重复采样 (sampling with replacement) 或者不重复采样 (sampling without replacement) 得到一个小批量中的各个样本。前者允许同一个小批量中出现重复的样本，后者则不允许如此，且更常见。对于这两者间的任一种方式，都可以使用

$$\mathbf{g}_t \leftarrow \nabla f_{\mathcal{B}_t}(\mathbf{x}_{t-1}) = \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \nabla f_i(\mathbf{x}_{t-1})$$

来计算时间步  $t$  的小批量  $\mathcal{B}_t$  上目标函数位于  $\mathbf{x}_{t-1}$  处的梯度  $\mathbf{g}_t$ 。这里  $|\mathcal{B}|$  代表批量大小，即小批量中样本的个数，是一个超参数。同随机梯度一样，重复采样所得的小批量随机梯度  $\mathbf{g}_t$  也是对梯度  $\nabla f(\mathbf{x}_{t-1})$  的无偏估计。给定学习率  $\eta_t$  (取正数)，小批量随机梯度下降对自变量的迭代如下：

$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \eta_t \mathbf{g}_t$$

基于随机采样得到的梯度的方差在迭代过程中无法减小，因此在实际中，(小批量) 随机梯度下降的学习率可以在迭代过程中自我衰减，例如  $\eta_t = \eta t^\alpha$  (通常  $\alpha = -1$  或者  $-0.5$ )、 $\eta_t = \eta \alpha^t$  (如  $\alpha = 0.95$ ) 或者每迭代若干次后将学习率衰减一次。如此一来，学习率和 (小批量) 随机梯度乘积的方差会减小。而梯度下降在迭代过程中一直使用目标函数的真实梯度，无须自我衰减学习率。

小批量随机梯度下降中每次迭代的计算开销为  $\mathcal{O}(|\mathcal{B}|)$ 。当批量大小为1时，该算法即为随机梯度下降；当批量大小等于训练数据样本数时，该算法即为梯度下降。当批量较小时，每次迭代中使用的样本少，这会导致并行处理和内存使用效率变低。这使得在计算同样数目样本的情况下比使用更大批量时所花时间更多。当批量较大时，每个小批量梯度里可能含有更多的冗余信息。为了得到较好的解，批量较大时比批量较小时需要计算的样本数目可能更多，例如增大迭代周期数。

## 6.3.1 读取数据集

本章里我们将使用一个来自NASA的测试不同飞机机翼噪音的数据集来比较各个优化算法。我们使用该数据集的前1,500个样本和5个特征，并使用标准化对数据进行预处理。

```
1 In [1]:  
2     import numpy as np  
3     import torch  
4     def get_data_ch7():  
5         # 数据读取以\t分隔  
6         # 从文本文件加载数据，并按指定处理缺失值。  
7         data = np.loadtxt('data/airfoil_self_noise.dat',  
8                           delimiter='\t')  
9         # 特征标准化处理  
10        data = (data - data.mean(axis=0)) / data.std(axis=0)  
11        # 最后一列是label  
12        # 前1500个样本(每个样本5个特征)  
13        return torch.tensor(data[:1500, :-1], dtype=torch.float32), \  
14                  torch.tensor(data[:1500, -1], dtype=torch.float32)  
15    features, labels = get_data_ch7()  
16    features.shape  
17 Out [1]:  
18    torch.Size([1500, 5])
```

## 6.3.2 从零开始实现

2.2节（线性回归的从零开始实现）中已经实现过小批量随机梯度下降算法。我们在这里将它的输入参数变得更加通用，主要是为了方便本章后面介绍的其他优化算法也可以使用同样的输入。具体来说，我们添加了一个状态输入 `states` 并将超参数放在字典 `hyperparams` 里。此外，我们将在训练函数里对各个小批量样本的损失求平均，因此优化算法里的梯度不需要除以批量大小。

```
1 In [2]:  
2     def sgd(params, states, hyperparams):  
3         for p in params:  
4             # 学习率衰减的参数更新  
5             p.data -= hyperparams['lr'] * p.grad.data
```

下面实现一个通用的训练函数，以方便本章后面介绍的其他优化算法使用。它初始化一个线性回归模型，然后可以使用小批量随机梯度下降以及后续小节介绍的其他算法来训练模型。

```
1 In [3]:  
2     import d2lzh as d2l  
3     import time  
4     def train_ch7(optimizer_fn, states, hyperparams,  
5                     features, labels, batch_size=10, num_epochs=2):  
6         # 初始化模型  
7         net, loss = d2l.linreg, d2l.squared_loss  
8         w = torch.nn.Parameter(  
9             torch.tensor(  
10                 np.random.normal(0, 0.01, size=(features.shape[1], 1)),  
11                 dtype=torch.float32),  
12                 requires_grad=True)  
13         b = torch.nn.Parameter(  
14             torch.zeros(1, dtype=torch.float32), requires_grad=True)  
15         def eval_loss():
```

```

16         return loss(net(features, w, b), labels).mean().item()
17 ls = [eval_loss()]
18 data_iter = torch.utils.data.DataLoader(
19     torch.utils.data.TensorDataset(features, labels),
20     batch_size,
21     shuffle=True)
22 for _ in range(num_epochs):
23     start = time.time()
24     for batch_i, (x, y) in enumerate(data_iter):
25         # 使用平均损失
26         l = loss(net(x, w, b), y).mean()
27         # 梯度清零
28         if w.grad is not None:
29             w.grad.data.zero_()
30             b.grad.data.zero_()
31         l.backward()
32         # 迭代模型参数
33         optimizer_fn([w, b], states, hyperparams)
34         # 每100个样本记录下当前训练误差
35         if (batch_i + 1) * batch_size % 100 == 0:
36             ls.append(eval_loss())
37         # 打印结果和作图
38         print('loss: %f, %f sec per epoch' % (ls[-1], time.time() - start))
39         d2l.set_figsize()
40         d2l.plt.plot(np.linspace(0, num_epochs, len(ls)), ls)
41         d2l.plt.xlabel('epoch')
42         d2l.plt.ylabel('loss')

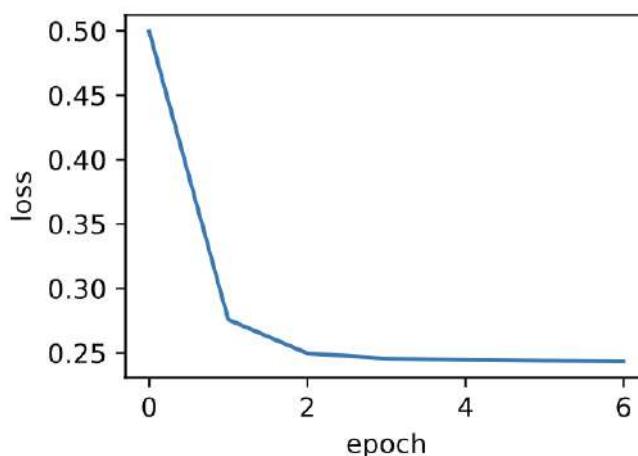
```

当批量大小为样本总数1,500时，优化使用的是梯度下降。梯度下降的1个迭代周期对模型参数只迭代1次。可以看到6次迭代后目标函数值（训练损失）的下降趋向了平稳。

```

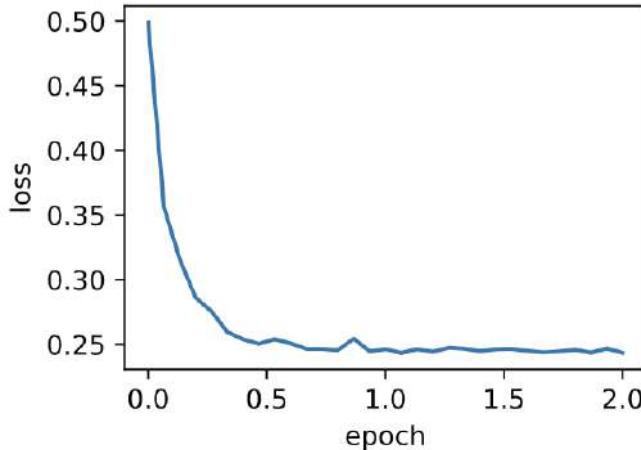
1 In [4]:
2     def train_sgd(lr, batch_size, num_epochs=2):
3         train_ch7(sgd, None, {'lr': lr}, features,
4                 labels, batch_size, num_epochs)
5     train_sgd(1, 1500, 6)
6 Out [4]:
7     loss: 0.247478, 0.051707 sec per epoch

```



当批量大小为1时，优化使用的是随机梯度下降。为了简化实现，有关（小批量）随机梯度下降的实验中，我们未对学习率进行自我衰减，而是直接采用较小的常数学习率。随机梯度下降中，每处理一个样本会更新一次自变量（模型参数），一个迭代周期里会对自变量进行1,500次更新。可以看到，目标函数值的下降在1个迭代周期后就变得较为平缓。

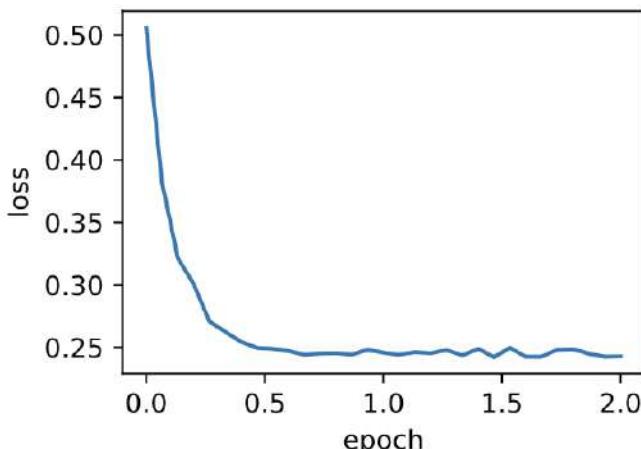
```
1 In [5]:  
2     train_sgd(0.005, 1)  
3 Out [5]:  
4     loss: 0.243556, 1.291605 sec per epoch
```



虽然随机梯度下降和梯度下降在一个迭代周期里都处理了1,500个样本，但实验中随机梯度下降的一个迭代周期耗时更多。这是因为随机梯度下降在一个迭代周期里做了更多次的自变量迭代，而且单样本的梯度计算难以有效利用矢量计算。

当批量大小为10时，优化使用的是小批量随机梯度下降。它在每个迭代周期的耗时介于梯度下降和随机梯度下降的耗时之间。

```
1 In [6]:  
2     train_sgd(0.05, 10)  
3 Out [6]:  
4     loss: 0.246715, 0.133914 sec per epoch
```



### 6.3.3 简洁实现

在PyTorch里可以通过创建 `optimizer` 实例来调用优化算法。这能让实现更简洁。下面实现一个通用的训练函数，它通过优化算法的函数 `optimizer_fn` 和超参数 `optimizer_hyperparams` 来创建 `optimizer` 实例。

```

1 In [7]:
2     import torch.nn as nn
3     def train_pytorch_ch7(optimizer_fn, optimizer_hyperparams,
4                           features, labels, batch_size=10, num_epochs=2):
5         # 初始化模型
6         net = nn.Sequential(
7             nn.Linear(features.shape[-1], 1)
8         )
9         loss = nn.MSELoss()
10        # 如果使用**前缀，输入的参数会被存放在字典中
11        optimizer = optimizer_fn(net.parameters(), **optimizer_hyperparams)
12        def eval_loss():
13            return loss(net(features).view(-1), labels).item() / 2
14        ls = [eval_loss()]
15        data_iter = torch.utils.data.DataLoader(
16            torch.utils.data.TensorDataset(features, labels),
17            batch_size, shuffle=True
18        )
19        for _ in range(num_epochs):
20            start = time.time()
21            for batch_i, (x, y) in enumerate(data_iter):
22                # 除以2是为了和train_ch7保持一致，因为squared_loss中除了2
23                l = loss(net(x).view(-1), y) / 2
24                # 梯度清零
25                optimizer.zero_grad()
26                l.backward()
27                optimizer.step()
28                if (batch_i + 1) * batch_size % 100 == 0:
29                    ls.append(eval_loss())
30        # 打印结果和作图
31        print('loss: %f, %f sec per epoch' % (ls[-1], time.time() - start))
32        d2l.set_figsize()
33        d2l.plt.plot(np.linspace(0, num_epochs, len(ls)), ls)
34        d2l.plt.xlabel('epoch')
35        d2l.plt.ylabel('loss')

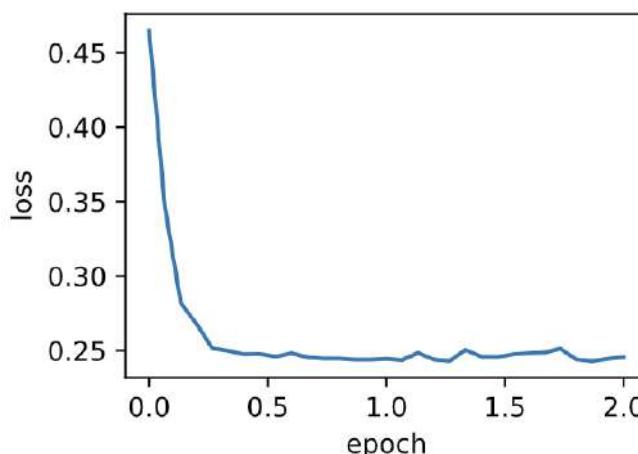
```

使用PyTorch重复上一个实验。

```

1 In [8]:
2     from torch import optim
3     train_pytorch_ch7(optim.SGD, {'lr': 0.05}, features, labels, 10)
4 Out [8]:
5     loss: 0.243574, 0.116980 sec per epoch

```



## 小结:

- 小批量随机梯度每次随机均匀采样一个小批量的训练样本来计算梯度。
- 在实际中，（小批量）随机梯度下降的学习率可以在迭代过程中自我衰减。
- 通常，小批量随机梯度在每个迭代周期的耗时介于梯度下降和随机梯度下降的耗时之间。

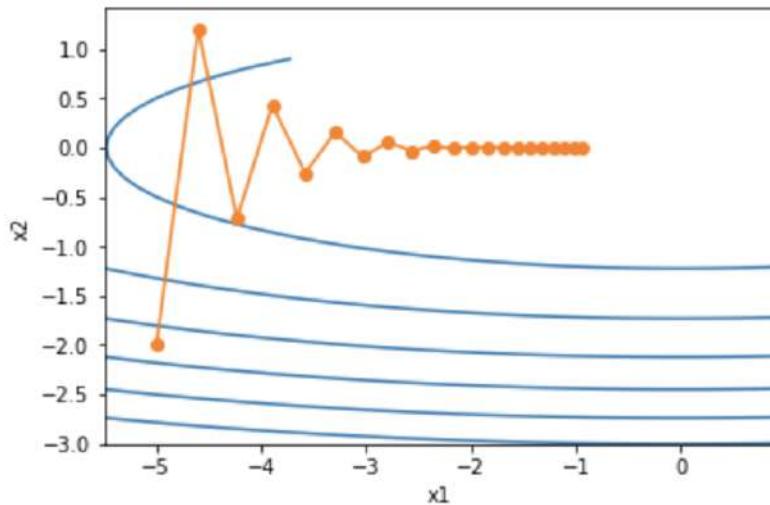
## 6.4 动量法

在6.2节（梯度下降和随机梯度下降）中我们提到，目标函数有关自变量的梯度代表了目标函数在自变量当前位置下降最快的方向。因此，梯度下降也叫作最陡下降（steepest descent）。在每次迭代中，梯度下降根据自变量当前位置，沿着当前位置的梯度更新自变量。然而，如果自变量的迭代方向仅仅取决于自变量当前位置，这可能会带来一些问题。

### 6.4.1 梯度下降的问题

让我们考虑一个输入和输出分别为二维向量  $\mathbf{x} = [x_1, x_2]^\top$  和标量的目标函数  $f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2$ 。与6.2节中不同，这里将  $x_1^2$  系数从1减小到了0.1。下面实现基于这个目标函数的梯度下降，并演示使用学习率为0.4时自变量的迭代轨迹。

```
1 In [1]:  
2     import d2lzh as d2l  
3     # 学习率  
4     eta = 0.4  
5     def f_2d(x1, x2):  
6         return 0.1 * x1 ** 2 + 2 * x2 ** 2  
7     def gd_2d(x1, x2, s1, s2):  
8         return (x1 - eta * 0.2 * x1, x2 - eta * 4 * x2, 0, 0)  
9     d2l.show_trace_2d(f_2d, d2l.train_2d(gd_2d))  
10    Out [1]:  
11        epoch 20, x1 -0.943467, x2 -0.000073
```



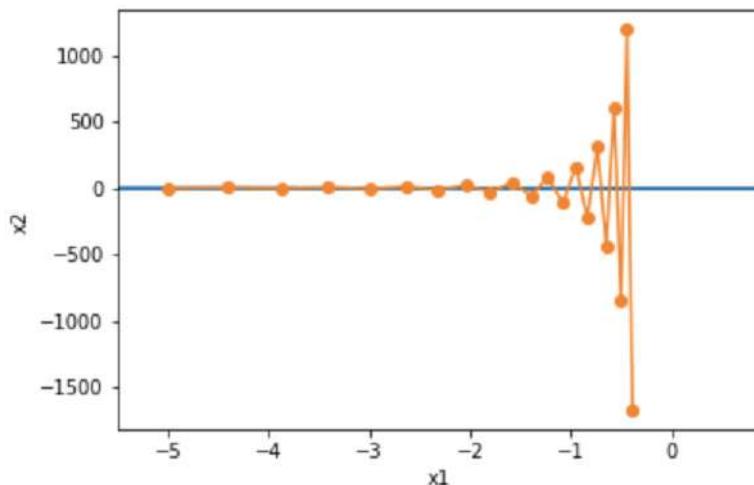
可以看到，同一位置上，目标函数在竖直方向 ( $x_2$  轴方向) 比在水平方向 ( $x_1$  轴方向) 的斜率的绝对值更大。因此，给定学习率，梯度下降迭代自变量时会使自变量在竖直方向比在水平方向移动幅度更大。那么，我们需要一个较小的学习率从而避免自变量在竖直方向上越过最优解。然而，这会造成自变量在水平方向上朝最优解移动变慢。

下面我们试着将学习率调得稍大一点，此时自变量在竖直方向不断越过最优解并逐渐发散。

```

1 In [2]:
2     eta = 0.6
3     d2l.show_trace_2d(f_2d, d2l.train_2d(gd_2d))
4 Out [2]:
5     epoch 20, x1 -0.387814, x2 -1673.365109

```



## 6.4.2 动量法

动量法的提出是为了解决梯度下降的上述问题。由于小批量随机梯度下降比梯度下降更为广义，本章后续讨论将沿用6.3节（小批量随机梯度下降）中时间步 $t$ 的小批量随机梯度 $\mathbf{g}_t$ 的定义。设时间步 $t$ 的自变量为 $\mathbf{x}_t$ ，学习率为 $\eta_t$ 。在时间步0，动量法创建速度变量 $\mathbf{v}_0$ ，并将其元素初始化成0。在时间步 $t > 0$ ，动量法对每次迭代的步骤做如下修改：

$$\begin{aligned}\mathbf{v}_t &\leftarrow \gamma \mathbf{v}_{t-1} + \eta_t \mathbf{g}_t \\ \mathbf{x}_t &\leftarrow \mathbf{x}_{t-1} - \mathbf{v}_t\end{aligned}$$

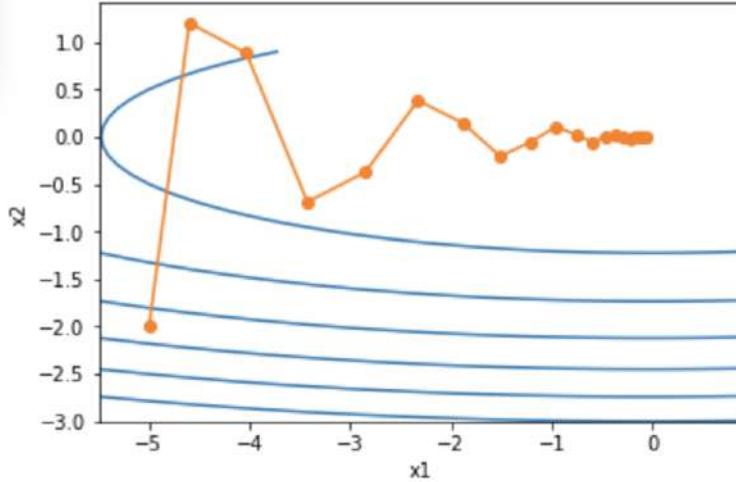
其中，动量超参数 $\gamma$ 满足 $0 \leq \gamma < 1$ 。当 $\gamma = 0$ 时，动量法等价于小批量随机梯度下降。

在解释动量法的数学原理前，让我们先从实验中观察梯度下降在使用动量法后的迭代轨迹。

```

1 In [3]:
2     def momentum_2d(x1, x2, v1, v2):
3         v1 = gamma * v1 + eta * 0.2 * x1
4         v2 = gamma * v2 + eta * 4 * x2
5         return x1-v1, x2-v2, v1, v2
6     eta, gamma = 0.4, 0.5
7     d2l.show_trace_2d(f_2d, d2l.train_2d(momentum_2d))
8 Out [3]:
9     epoch 20, x1 -0.062843, x2 0.001202

```

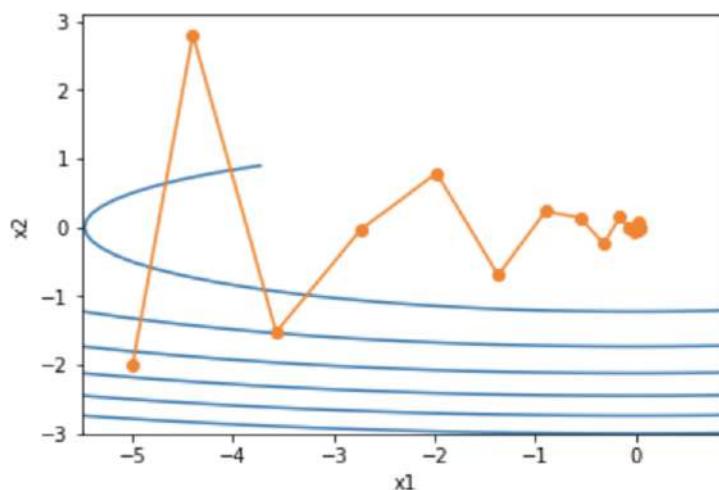


可以看到使用较小的学习率 $\eta = 0.4$ 和动量超参数 $\gamma = 0.5$ 时，动量法在竖直方向上的移动更加平滑，且在水平方向上更快逼近最优解。下面使用较大的学习率 $\eta = 0.6$ ，此时自变量也不再发散。

```

1 In [4]:
2     eta = 0.6
3     d2l.show_trace_2d(f_2d, d2l.train_2d(momentum_2d))
4 Out [4]:
5 epoch 20, x1 0.007188, x2 0.002553

```



## 1. 指数加权移动平均

为了从数学上理解动量法，让我们先解释一下指数加权移动平均 (exponentially weighted moving average)。给定超参数 $0 \leq \gamma < 1$ ，当前时间步 $t$ 的变量 $y_t$ 是上一时间步 $t-1$ 的变量 $y_{t-1}$ 和当前时间步另一变量 $x_t$ 的线性组合：

$$y_t = \gamma y_{t-1} + (1 - \gamma)x_t$$

我们可以对 $y_t$ 展开：

$$\begin{aligned} y_t &= (1 - \gamma)x_t + \gamma y_{t-1} \\ &= (1 - \gamma)x_t + (1 - \gamma) \cdot \gamma x_{t-1} + \gamma^2 y_{t-2} \\ &= (1 - \gamma)x_t + (1 - \gamma) \cdot \gamma x_{t-1} + (1 - \gamma) \cdot \gamma^2 x_{t-2} + \gamma^3 y_{t-3} \\ &\dots \end{aligned}$$

令 $n = 1/(1 - \gamma)$ ，那么 $(1 - 1/n)^n = \gamma^{1/(1-\gamma)}$ 。根据重要极限  $\lim_{x \rightarrow \infty} (1 + \frac{1}{x})^x = e$ ，我们有：

$$\lim_{n \rightarrow \infty} \left(1 - \frac{1}{n}\right)^n = \exp(-1) \approx 0.3679$$

所以当 $\gamma \rightarrow 1$ 时,  $\gamma^{1/(1-\gamma)} = \exp(-1)$ , 如 $0.95^{20} \approx \exp(-1)$ 。如果把 $\exp(-1)$ 当作一个比较小的数, 我们可以在近似中忽略所有含 $\gamma^{1/(1-\gamma)}$ 和比 $\gamma^{1/(1-\gamma)}$ 更高阶的系数的项。例如, 当 $\gamma = 0.95$ 时,

$$y_t \approx 0.05 \sum_{i=0}^{19} 0.95^i x_{t-i}$$

因此, 在实际中, 我们常常将 $y_t$ 看作是对最近 $1/(1 - \gamma)$ 个时间步的 $x_t$ 值的加权平均。例如, 当 $\gamma = 0.95$ 时,  $y_t$ 可以被看作对最近20个时间步的 $x_t$ 值的加权平均; 当 $\gamma = 0.9$ 时,  $y_t$ 可以看作是对最近10个时间步的 $x_t$ 值的加权平均。而且, 离当前时间步 $t$ 越近的 $x_t$ 值获得的权重越大(越接近1)。

## 2. 由指数加权移动平均理解动量法

现在, 我们对动量法的速度变量做变形:

$$\mathbf{v}_t \leftarrow \gamma \mathbf{v}_{t-1} + (1 - \gamma) \left( \frac{\eta_t}{1 - \gamma} \mathbf{g}_t \right)$$

由指数加权移动平均的形式可得, 速度变量 $\mathbf{v}_t$ 实际上对序列

$\{\eta_{t-i} \mathbf{g}_{t-i} / (1 - \gamma) : i = 0, \dots, 1/(1 - \gamma) - 1\}$ 做了指数加权移动平均。换句话说, 相比于小批量随机梯度下降, 动量法在每个时间步的自变量更新量近似于将最近 $1/(1 - \gamma)$ 个时间步的普通更新量(即学习率乘以梯度)做了指数加权移动平均后再除以 $1 - \gamma$ 。所以, 在动量法中, 自变量在各个方向上的移动幅度不仅取决于当前梯度, 还取决于过去的各个梯度在各个方向上是否一致(这里方向的描述可以参考本章附录中梯度的几何意义)。在本节之前示例的优化问题中, 所有梯度在水平方向上为正(向右), 而在竖直方向上时正(向上)时负(向下)。这样, 我们就可以使用较大的学习率, 从而使自变量向最优解更快移动。

### 6.4.3 从零开始实现

相对于小批量随机梯度下降, 动量法需要对每一个自变量维护一个同它一样形状的速度变量, 且超参数里多了动量超参数。实现中, 我们将速度变量用更广义的状态变量 states 表示。

```

1 In [5]:
2     import torch
3     features, labels = d2l.get_data_ch7()
4     # 初始化动量相关变量
5     def init_momentum_states():
6         v_w = torch.zeros((features.shape[1], 1), dtype=torch.float32)
7         v_b = torch.zeros(1, dtype=torch.float32)
8         return (v_w, v_b)
9     def sgd_momentum(params, states, hyperparams):
10        for p, v in zip(params, states):
11            v.data = hyperparams['momentum']*v.data + \
12                hyperparams['lr']*p.grad.data
13            p.data -= v.data

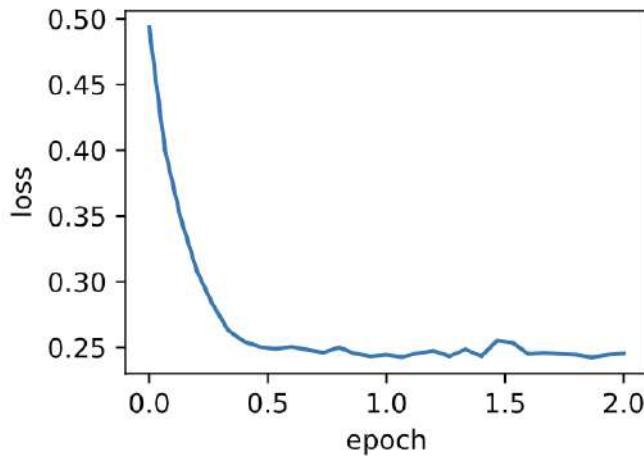
```

我们先将动量超参数 momentum 设 0.5, 这时可以看成是特殊的小批量随机梯度下降: 其小批量随机梯度为最近2个时间步的2倍小批量梯度的加权和。

```

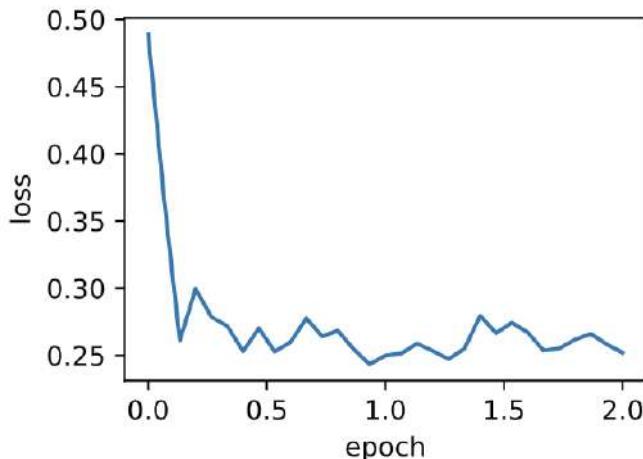
1 In [6]:
2     d2l.train_ch7(sgd_momentum, init_momentum_states(),
3                     {'lr': 0.02, 'momentum': 0.5}, features, labels)
4 Out [6]:
5     loss: 0.243169, 0.260499 sec per epoch

```



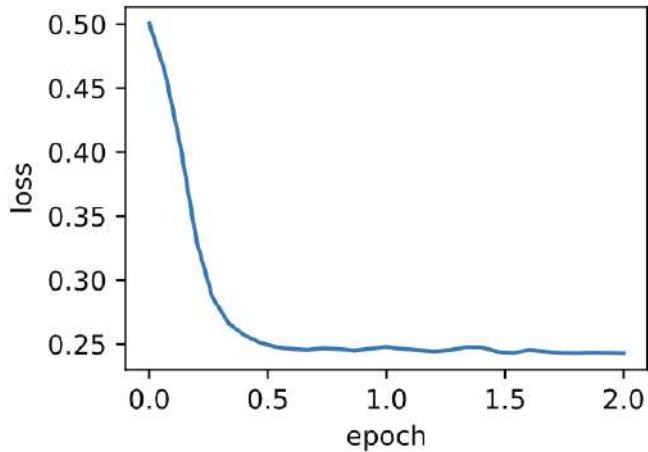
将动量超参数 `momentum` 增大到 0.9，这时依然可以看成是特殊的小批量随机梯度下降：其小批量随机梯度为最近 10 个时间步的 10 倍小批量梯度的加权和。我们先保持学习率 0.02 不变。

```
1 In [7]:  
2     d2l.train_ch7(sgd_momentum, init_momentum_states(),  
3                      {'lr': 0.02, 'momentum': 0.9}, features, labels)  
4 Out [7]:  
5     loss: 0.270185, 0.170542 sec per epoch
```



可见目标函数值在后期迭代过程中的变化不够平滑。直觉上，10倍小批量梯度比2倍小批量梯度大了5倍，我们可以试着将学习率减小到原来的1/5。此时目标函数值在下降了一段时间后变化更加平滑。

```
1 In [8]:  
2     d2l.train_ch7(sgd_momentum, init_momentum_states(),  
3                      {'lr': 0.004, 'momentum': 0.9}, features, labels)  
4 Out [8]:  
5     loss: 0.245188, 0.202992 sec per epoch
```



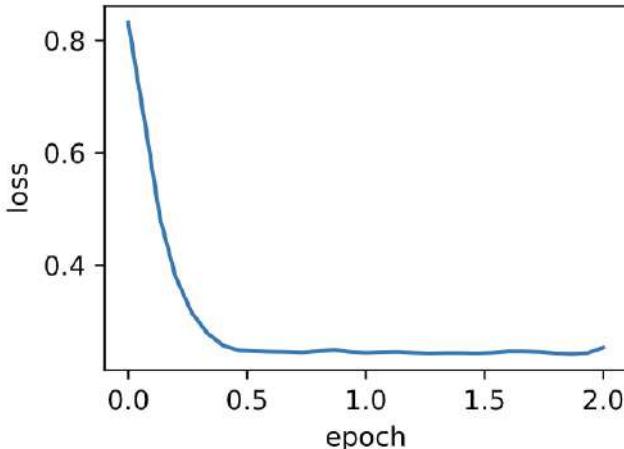
#### 6.4.4 简洁实现

在PyTorch中，只需要通过参数 `momentum` 来指定动量超参数即可使用动量法。

```

1 In [9]:
2     # 字典部分为优化器所要用到的参数
3     d2l.train_pytorch_ch7(torch.optim.SGD, {'lr': 0.004, 'momentum': 0.9},
4                           features, labels)
5 Out [9]:
6     loss: 0.244982, 0.120022 sec per epoch

```



**小结：**

- 动量法使用了指数加权移动平均的思想。它将过去时间步的梯度做了加权平均，且权重按时间步指数衰减。
- 动量法使得相邻时间步的自变量更新在方向上更加一致。

## 6.5 AdaGrad算法

在之前介绍过的优化算法中，目标函数自变量的每一个元素在相同时间步都使用同一个学习率来自我迭代。举个例子，假设目标函数为  $f$ ，自变量为一个二维向量  $[x_1, x_2]^\top$ ，该向量中每一个元素在迭代时都使用相同的学习率。例如，在学习率为  $\eta$  的梯度下降中，元素  $x_1$  和  $x_2$  都使用相同的学习率  $\eta$  来自我迭代：

$$x_1 \leftarrow x_1 - \eta \frac{\partial f}{\partial x_1}$$

$$x_2 \leftarrow x_2 - \eta \frac{\partial f}{\partial x_2}$$

在6.4节（动量法）里我们看到当 $x_1$ 和 $x_2$ 的梯度值有较大差别时，需要选择足够小的学习率使得自变量在梯度值较大的维度上不发散。但这样会导致自变量在梯度值较小的维度上迭代过慢。动量法依赖指数加权移动平均使得自变量的更新方向更加一致，从而降低发散的可能。本节我们介绍**AdaGrad算法**，它根据自变量在每个维度的梯度值的大小来调整各个维度上的学习率，从而避免统一的学习率难以适应所有维度的问题。

## 6.5.1 算法

AdaGrad算法会使用一个小批量随机梯度 $\mathbf{g}_t$ 按元素平方的累加变量 $\mathbf{s}_t$ 。在时间步0，AdaGrad将 $\mathbf{s}_0$ 中每个元素初始化为0。在时间步 $t$ ，首先将小批量随机梯度 $\mathbf{g}_t$ 按元素平方后累加到变量 $\mathbf{s}_t$ ：

$$\mathbf{s}_t \leftarrow \mathbf{s}_{t-1} + \mathbf{g}_t \odot \mathbf{g}_t$$

其中 $\odot$ 是按元素相乘。接着，我们将目标函数自变量中每个元素的学习率通过按元素运算重新调整一下：

$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t$$

其中 $\eta$ 是学习率， $\epsilon$ 是为了维持数值稳定性而添加的常数，如 $10^{-6}$ 。这里开方、除法和乘法的运算都是按元素运算的。这些按元素运算使得目标函数自变量中每个元素都分别拥有自己的学习率。

## 6.5.2 特点

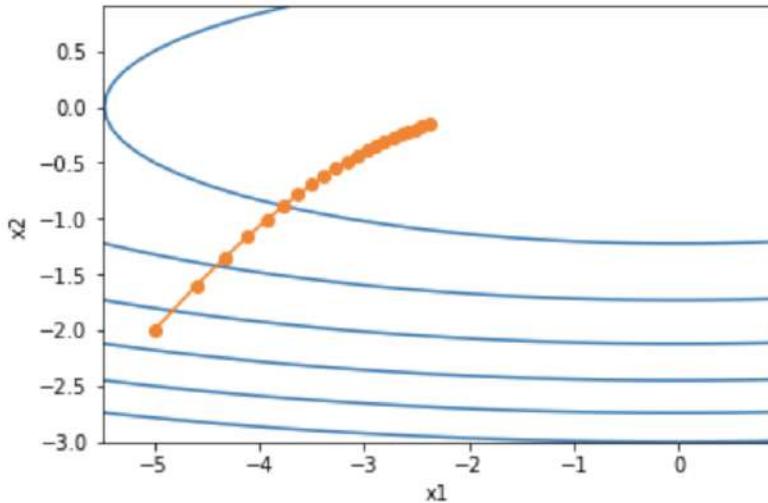
需要强调的是，小批量随机梯度按元素平方的累加变量 $\mathbf{s}_t$ 出现在学习率的分母项中。因此，如果目标函数有关自变量中某个元素的偏导数一直都较大，那么该元素的学习率将下降较快；反之，如果目标函数有关自变量中某个元素的偏导数一直都较小，那么该元素的学习率将下降较慢。然而，由于 $\mathbf{s}_t$ 一直在累加按元素平方的梯度，自变量中每个元素的学习率在迭代过程中一直在降低（或不变）。所以，**当学习率在迭代早期降得较快且当前解依然不佳时，AdaGrad算法在迭代后期由于学习率过小，可能较难找到一个有用的解。**

下面我们仍然以目标函数 $f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2$ 为例观察AdaGrad算法对自变量的迭代轨迹。我们实现AdaGrad算法并使用和上一节实验中相同的学习率0.4。可以看到，自变量的迭代轨迹较平滑。但由于 $\mathbf{s}_t$ 的累加效果使学习率不断衰减，自变量在迭代后期的移动幅度较小。

```

1 In [1]:
2     import math
3     def adagrad_2d(x1, x2, s1, s2):
4         # 前两项为自变量梯度
5         g1, g2, eps = 0.2*x1, 4*x2, 1e-6
6         s1 += g1**2
7         s2 += g2**2
8         x1 -= eta / math.sqrt(s1+eps)*g1
9         x2 -= eta / math.sqrt(s2+eps)*g2
10        return x1, x2, s1, s2
11    def f_2d(x1, x2):
12        return 0.1*x1**2+2*x2**2
13    eta = 0.4
14    d2l.show_trace_2d(f_2d, d2l.train_2d(adagrad_2d))
15 Out [1]:
16     epoch 20, x1 -2.382563, x2 -0.158591

```

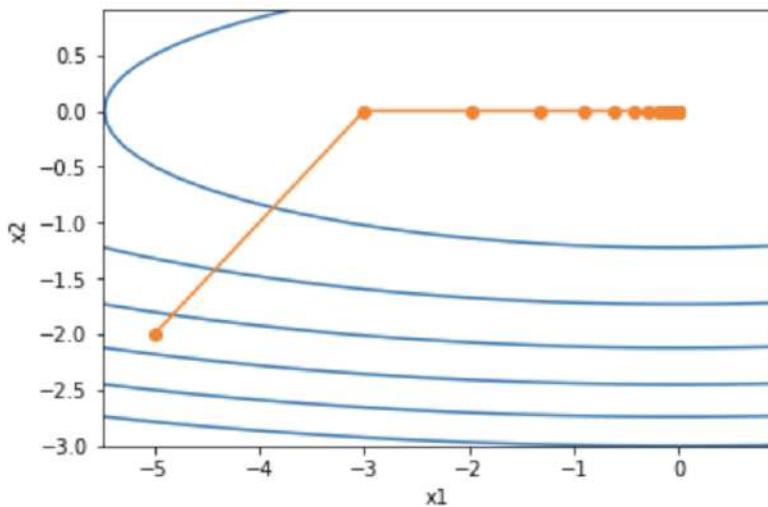


下面将学习率增大到2。可以看到自变量更为迅速地逼近了最优解。

```

1 In [2]:
2     eta = 2
3     d2l.show_trace_2d(f_2d, d2l.train_2d(adagrad_2d))
4 Out [2]:
5     epoch 20, x1 -0.002295, x2 -0.000000

```



### 6.5.3 从零开始实现

同动量法一样，AdaGrad算法需要对每个自变量维护同它一样形状的状态变量。我们根据AdaGrad算法中的公式实现该算法。

```

1 In [3]:
2     features, labels = d2l.get_data_ch7()
3     def init_adagrad_states():
4         s_w = torch.zeros((features.shape[1], 1), dtype=torch.float32)
5         s_b = torch.zeros(1, dtype=torch.float32)
6         return (s_w, s_b)
7     def adagrad(params, states, hyperparams):
8         eps = 1e-6
9         for p, s in zip(params, states):
10             s.data += (p.grad.data**2)
11             p.data -= hyperparams['lr'] * p.grad.data / torch.sqrt(s+eps)

```

与6.3节（小批量随机梯度下降）中的实验相比，这里使用更大的学习率来训练模型。

```

1 In [4]:
2     d2l.train_ch7(adagrad, init_adagrad_states(),
3                     {'lr': 0.1}, features, labels)
4 Out [4]:
5     loss: 0.242391, 0.163231 sec per epoch

```

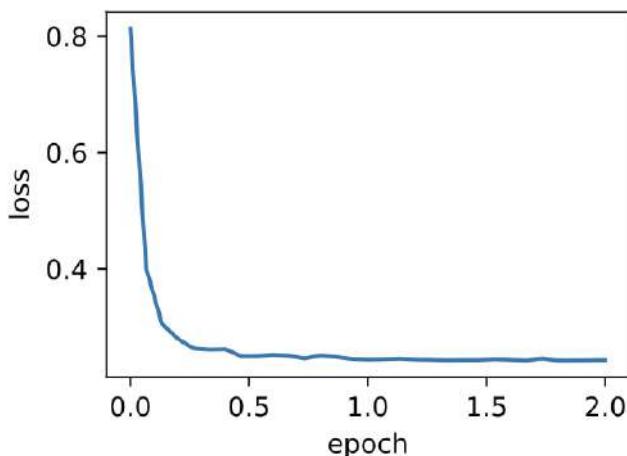
## 6.5.4 简洁实现

通过名称为 Adagrad 的优化器方法，我们便可使用PyTorch提供的AdaGrad算法来训练模型。

```

1 In [5]:
2     d2l.train_pytorch_ch7(torch.optim.Adagrad, {'lr': 0.1},
3                           features, labels)
4 Out [5]:
5     loss: 0.242369, 0.153083 sec per epoch

```



**小结：**

- AdaGrad算法在迭代过程中不断调整学习率，并让目标函数自变量中每个元素都分别拥有自己的学习率。
- 使用AdaGrad算法时，自变量中每个元素的学习率在迭代过程中一直在降低（或不变）。

## 6.6 RMSProp算法

我们在6.5节（AdaGrad算法）中提到，因为调整学习率时分母上的变量 $s_t$ 一直在累加按元素平方的小批量随机梯度，所以目标函数自变量每个元素的学习率在迭代过程中一直在降低（或不变）。因此，当学习率在迭代早期降得较快且当前解依然不佳时，AdaGrad算法在迭代后期由于学习率过小，可能较难找到一个有用的解。为了解决这一问题，RMSProp算法对AdaGrad算法做了一点小小的修改。该算法源自Coursera上的一门课程，即“机器学习的神经网络”。

### 6.6.1 算法

我们在6.4节（动量法）里介绍过指数加权移动平均。不同于AdaGrad算法里状态变量 $s_t$ 是截至时间步 $t$ 所有小批量随机梯度 $\mathbf{g}_t$ 按元素平方和，RMSProp算法将这些梯度按元素平方做**指数加权移动平均**。具体来说，给定超参数 $0 \leq \gamma < 1$ ，RMSProp算法在时间步 $t > 0$ 计算

$$\mathbf{s}_t \leftarrow \gamma \mathbf{s}_{t-1} + (1 - \gamma) \mathbf{g}_t \odot \mathbf{g}_t$$

和AdaGrad算法一样，RMSProp算法将目标函数自变量中每个元素的学习率通过按元素运算重新调整，然后更新自变量

$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \frac{\eta}{\sqrt{s_t + \epsilon}} \odot \mathbf{g}_t$$

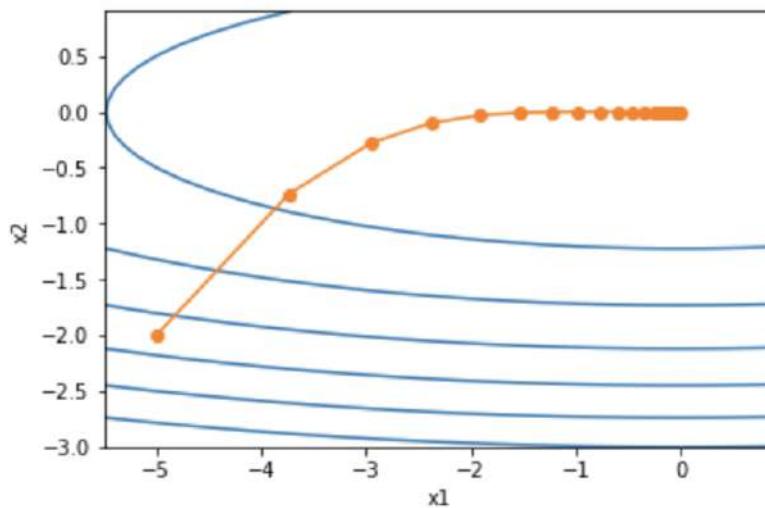
其中 $\eta$ 是学习率， $\epsilon$ 是为了维持数值稳定性而添加的常数，如 $10^{-6}$ 。因为RMSProp算法的状态变量 $s_t$ 是对平方项 $\mathbf{g}_t \odot \mathbf{g}_t$ 的指数加权移动平均，所以可以看作是最近 $1/(1-\gamma)$ 个时间步的小批量随机梯度平方项的加权平均。如此一来，自变量每个元素的学习率在迭代过程中就不再一直降低（或不变）。

照例，让我们先观察RMSProp算法对目标函数 $f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2$ 中自变量的迭代轨迹。回忆在6.5节（AdaGrad算法）使用的学习率为0.4的AdaGrad算法，自变量在迭代后期的移动幅度较小。但在同样的学习率下，RMSProp算法可以更快逼近最优解。

```

1 In [1]:
2     def rmsprop_2d(x1, x2, s1, s2):
3         g1, g2, eps = 0.2*x1, 4*x2, 1e-6
4         s1 = gamma*s1+(1-gamma)*g1**2
5         s2 = gamma*s2+(1-gamma)*g2**2
6         x1 -= eta / math.sqrt(s1+eps)*g1
7         x2 -= eta / math.sqrt(s2+eps)*g2
8         return x1, x2, s1, s2
9     def f_2d(x1, x2):
10        return 0.1*x1**2+2*x2**2
11    eta, gamma = 0.4, 0.9
12    d2l.show_trace_2d(f_2d, d2l.train_2d(rmsprop_2d))
13 Out [1]:
14 epoch 20, x1 -0.010599, x2 0.000000

```



## 6.6.2 从零开始实现

接下来按照RMSProp算法中的公式实现该算法。

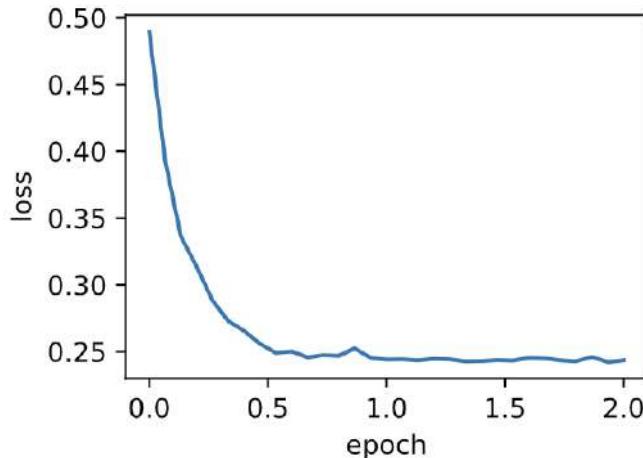
```

1 In [2]:
2     features, labels = d2l.get_data_ch7()
3     def init_rmsprop_states():
4         s_w = torch.zeros((features.shape[1], 1), dtype=torch.float32)
5         s_b = torch.zeros(1, dtype=torch.float32)
6         return (s_w, s_b)
7     def rmsprop(params, states, hyperparams):
8         gamma, eps = hyperparams['gamma'], 1e-6
9         for p, s in zip(params, states):
10            s.data = gamma * s.data + (1-gamma) * (p.grad.data)**2
11            p.data -= hyperparams['lr']*p.grad.data / torch.sqrt(s+eps)

```

我们将初始学习率设为0.01，并将超参数 $\gamma$ 设为0.9。此时，变量 $s_t$ 可看作是最近 $1/(1 - 0.9) = 10$ 个时间步的平方项 $g_t \odot g_t$ 的加权平均。

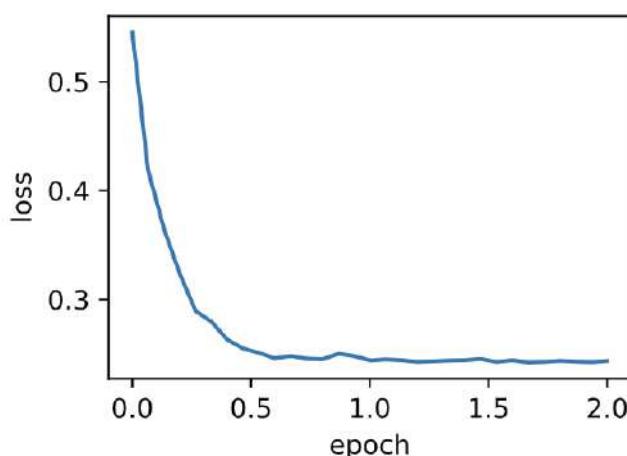
```
1 In [3]:  
2     d2l.train_ch7(rmsprop, init_rmsprop_states(),  
3                     {'lr': 0.01, 'gamma': 0.9}, features, labels)  
4 Out [3]:  
5     loss: 0.245565, 0.208567 sec per epoch
```



### 6.6.3 简洁实现

通过名称为 `RMSprop` 的优化器方法，我们便可使用PyTorch提供的RMSProp算法来训练模型。注意，超参数 $\gamma$ 通过 `alpha` 指定。

```
1 In [4]:  
2     d2l.train_pytorch_ch7(torch.optim.RMSprop,  
3                           {'lr': 0.01, 'alpha': 0.9}, features, labels)  
4 Out [4]:  
5     loss: 0.243149, 0.128298 sec per epoch
```



**小结：**

- RMSProp算法和AdaGrad算法的不同在于，RMSProp算法使用了小批量随机梯度按元素平方的指数加权移动平均来调整学习率。

## 6.7 AdaDelta算法

除了RMSProp算法以外，另一个常用优化算法AdaDelta算法也针对AdaGrad算法在迭代后期可能较难找到有用解的问题做了改进。有意思的是，**AdaDelta算法没有学习率这一超参数**。

## 6.7.1 算法

AdaDelta算法也像RMSProp算法一样，使用了小批量随机梯度 $\mathbf{g}_t$ 按元素平方的指数加权移动平均变量 $\mathbf{s}_t$ 。在时间步0，它的所有元素被初始化为0。给定超参数 $0 \leq \rho < 1$ （对应RMSProp算法中的 $\gamma$ ），在时间步 $t > 0$ ，同RMSProp算法一样计算

$$\mathbf{s}_t \leftarrow \rho \mathbf{s}_{t-1} + (1 - \rho) \mathbf{g}_t \odot \mathbf{g}_t$$

与RMSProp算法不同的是，AdaDelta算法还维护一个额外的状态变量 $\Delta \mathbf{x}_t$ ，其元素同样在时间步0时被初始化为0。我们用 $\mathbf{g}'_t$ 表示自变量的变化量，其计算过程依赖于 $\Delta \mathbf{x}_{t-1}$ ：

$$\mathbf{g}'_t \leftarrow \sqrt{\frac{\Delta \mathbf{x}_{t-1} + \epsilon}{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t$$

其中 $\epsilon$ 是为了维持数值稳定性而添加的常数，如 $10^{-5}$ 。接着更新自变量：

$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \mathbf{g}'_t$$

最后，我们使用 $\Delta \mathbf{x}_t$ 来记录自变量变化量 $\mathbf{g}'_t$ 按元素平方的指数加权移动平均：

$$\Delta \mathbf{x}_t \leftarrow \rho \Delta \mathbf{x}_{t-1} + (1 - \rho) \mathbf{g}'_t \odot \mathbf{g}'_t$$

可以看到，如不考虑 $\epsilon$ 的影响，**AdaDelta算法跟RMSProp算法的不同之处在于使用 $\sqrt{\Delta \mathbf{x}_{t-1}}$ 来替代学习率 $\eta$** 。

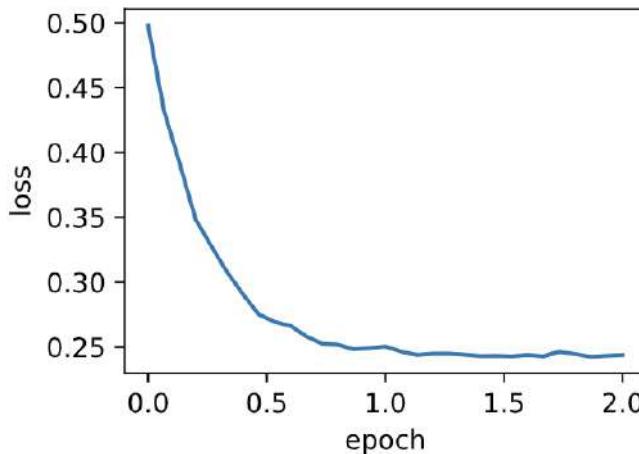
## 6.7.2 从零开始实现

AdaDelta算法需要对每个自变量维护两个状态变量，即 $\mathbf{s}_t$ 和 $\Delta \mathbf{x}_t$ 。我们按AdaDelta算法中的公式实现该算法。

```
1 In [1]:  
2     features, labels = d2l.get_data_ch7()  
3     def init_adadelta_states():  
4         s_w = torch.zeros((features.shape[-1], 1),  
5                            dtype=torch.float32)  
6         s_b = torch.zeros(1, dtype=torch.float32)  
7         delta_w = torch.zeros((features.shape[-1], 1),  
8                            dtype=torch.float32),  
9         delta_b = torch.zeros(1, dtype=torch.float32)  
10        return ((s_w, delta_w), (s_b, delta_b))  
11    def adadelta(params, states, hyperparams):  
12        rho, eps = hyperparams['rho'], 1e-5  
13        for p, (s, delta) in zip(params, states):  
14            s[:] = rho*s+(1-rho)*(p.grad.data**2)  
15            g = p.grad.data*torch.sqrt((delta+eps)/(s+eps))  
16            p.data -= g  
17            delta[:] = rho*delta+(1-rho)*g*g
```

使用超参数 $\rho = 0.9$ 来训练模型。

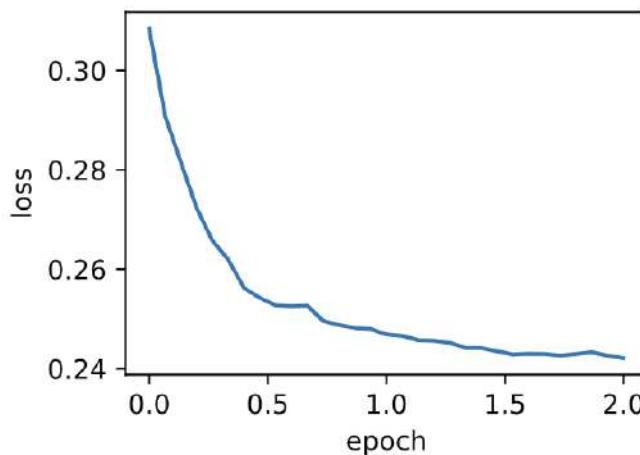
```
1 In [2]:  
2     d2l.train_ch7(adadelta, init_adadelta_states(),  
3                     {'rho': 0.9}, features, labels)  
4 Out [2]:  
5     loss: 0.244431, 0.208847 sec per epoch
```



### 6.7.3 简洁实现

通过名称为 Adadelta 的优化器方法，我们便可使用PyTorch提供的AdaDelta算法。它的超参数可以通过 rho 来指定。

```
1 In [3]:  
2     d2l.train_pytorch_ch7(torch.optim.Adadelta,  
3                           {'rho': 0.9}, features, labels)  
4 Out [3]:  
5     loss: 0.266846, 0.202872 sec per epoch
```



**小结：**

- AdaDelta算法没有学习率超参数，它通过使用有关自变量更新量平方的指数加权移动平均的项来替代RMSProp算法中的学习率。

## 6.8 Adam算法

Adam算法在RMSProp算法基础上对小批量随机梯度也做了**指数加权移动平均**，所以Adam算法可以看做是RMSProp算法与动量法的结合，下面我们来介绍这个算法。

## 6.8.1 算法

Adam算法使用了动量变量 $v_t$ 和RMSProp算法中小批量随机梯度按元素平方的指数加权移动平均变量 $s_t$ ，并在时间步0将它们中每个元素初始化为0。给定超参数 $0 \leq \beta_1 < 1$ （算法作者建议设为0.9），时间步 $t$ 的动量变量 $v_t$ 即小批量随机梯度 $g_t$ 的指数加权移动平均：

$$v_t \leftarrow \beta_1 v_{t-1} + (1 - \beta_1) g_t$$

和RMSProp算法中一样，给定超参数 $0 \leq \beta_2 < 1$ （算法作者建议设为0.999），将小批量随机梯度按元素平方后的项 $g_t \odot g_t$ 做指数加权移动平均得到 $s_t$ ：

$$s_t \leftarrow \beta_2 s_{t-1} + (1 - \beta_2) g_t \odot g_t$$

由于我们将 $v_0$ 和 $s_0$ 中的元素都初始化为0，在时间步 $t$ 我们得到 $v_t = (1 - \beta_1) \sum_{i=1}^t \beta_1^{t-i} g_i$ （参见6.4节）。将过去各时间步小批量随机梯度的权值相加，得到 $(1 - \beta_1) \sum_{i=1}^t \beta_1^{t-i} = 1 - \beta_1^t$ （等号右边求和公式展开，乘法分配律，前后两项错位相减）。需要注意的是，当 $t$ 较小时，过去各时间步小批量随机梯度权值之和会较小。例如，当 $\beta_1 = 0.9$ 时， $v_1 = 0.1g_1$ 。为了消除这样的影响，对于任意时间步 $t$ ，我们可以将 $v_t$ 再除以 $1 - \beta_1^t$ ，从而使过去各时间步小批量随机梯度权值之和为1。这也叫作**偏差修正**。在Adam算法中，我们对变量 $v_t$ 和 $s_t$ 均作偏差修正：

$$\hat{v}_t \leftarrow \frac{v_t}{1 - \beta_1^t}$$
$$\hat{s}_t \leftarrow \frac{s_t}{1 - \beta_2^t}$$

接下来，Adam算法使用以上偏差修正后的变量 $\hat{v}_t$ 和 $\hat{s}_t$ ，将模型参数中每个元素的学习率通过按元素运算重新调整：

$$g'_t \leftarrow \frac{\eta \hat{v}_t}{\sqrt{\hat{s}_t} + \epsilon}$$

其中 $\eta$ 是学习率， $\epsilon$ 是为了维持数值稳定性而添加的常数，如 $10^{-8}$ 。**和AdaGrad算法、RMSProp算法以及AdaDelta算法一样，目标函数自变量中每个元素都分别拥有自己的学习率。**最后，使用 $g'_t$ 迭代自变量：

$$x_t \leftarrow x_{t-1} - g'_t$$

## 6.8.2 从零开始实现

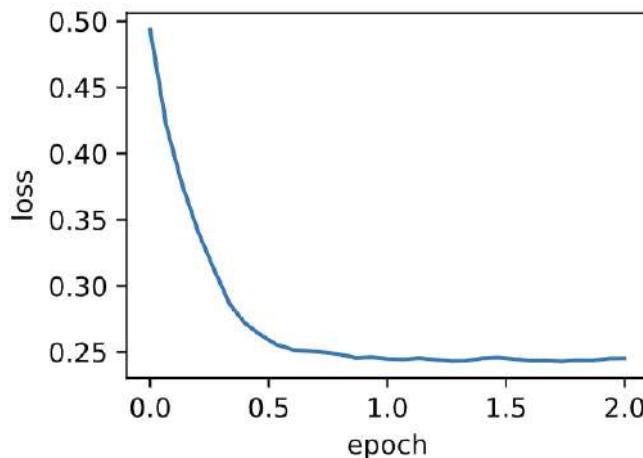
我们按照Adam算法中的公式实现该算法。其中时间步 $t$ 通过`hyperparams`参数传入`adam`函数。

```
1 In [1]:  
2     features, labels = d2l.get_data_ch7()  
3     def init_adam_states():  
4         v_w = torch.zeros((features.shape[1], 1),  
5                            dtype=torch.float32),  
6         v_b = torch.zeros(1, dtype=torch.float32)  
7         s_w = torch.zeros((features.shape[1], 1),  
8                            dtype=torch.float32),  
9         s_b = torch.zeros(1, dtype=torch.float32)  
10        return ((v_w, s_w), (v_b, s_b))  
11    def adam(params, states, hyperparams):  
12        beta1, beta2, eps = 0.9, 0.999, 1e-6  
13        for p, (v, s) in zip(params, states):  
14            v[:] = beta1*v + (1-beta1)*p.grad.data  
15            s[:] = beta2*s + (1-beta2)*p.grad.data**2  
16            v_bias_corr = v/(1-beta1**hyperparams['t'])
```

```
17         s_bias_corr = s/(1-beta2**hyperparams['t'])
18         p.data -= hyperparams['lr']*v_bias_corr/
19             torch.sqrt(s_bias_corr)+eps)
20         hyperparams['t'] += 1
```

使用学习率为0.01的Adam算法来训练模型。

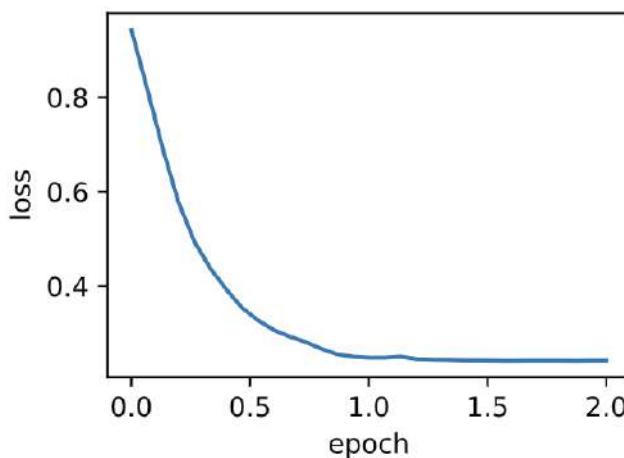
```
1 In [2]:
2     d2l.train_ch7(adam, init_adam_states(),
3                     {'lr': 0.01, 't': 1}, features, labels)
4 Out [2]:
5     loss: 0.243669, 0.228965 sec per epoch
```



### 6.8.3 简洁实现

通过名称为“Adam”的优化器实例，我们便可使用PyTorch提供的Adam算法。

```
1 In [3]:
2     d2l.train_pytorch_ch7(torch.optim.Adam, {'lr': 0.01},
3                           features, labels)
4 Out [3]:
5     loss: 0.243203, 0.139672 sec per epoch
```



#### 小结：

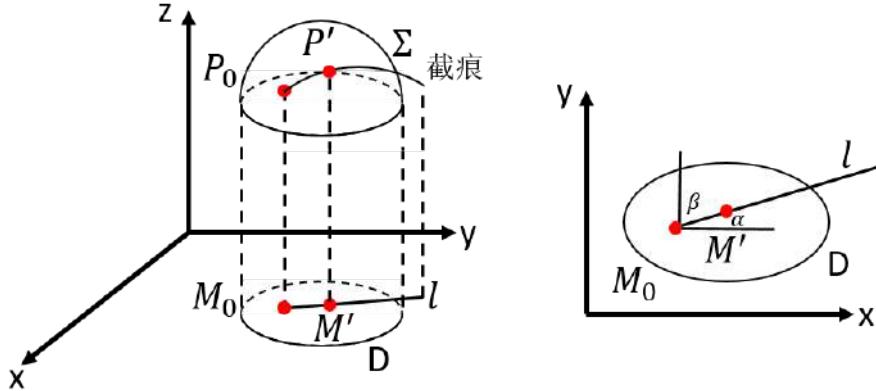
- Adam算法在RMSProp算法的基础上对小批量随机梯度也做了指数加权移动平均。
- Adam算法使用了偏差修正。

## 6.9 本章附录

### ☆ 方向导数与梯度

#### 1. 背景

我们定义曲面  $\Sigma : z = \phi(x, y)$ , 其中  $(x, y) \in D$ , 且定义点  $M_0 : (x_0, y_0) \in D$ 。为了方便理解, 做如下示意图:



#### 2. 方向导数

我们定义曲面  $\Sigma : z = f(x, y)$ , 其中  $(x, y) \in D$ , 且定义点  $M_0 : (x_0, y_0) \in D$ 。过  $M_0$  作射线  $l$ , 我们有  $M' : (x_0 + \Delta x, y_0 + \Delta y) \in l$ 。此时,  $\Delta z = f(x_0 + \Delta x, y_0 + \Delta y) - f(x_0, y_0)$ , 若其大于 0 说明在上升, 否则在下降。另外我们有  $\rho = |M_0 M'| = \sqrt{(\Delta x)^2 + (\Delta y)^2}$ 。那么高度差的平均变化率的计算公式如下,  $\frac{\Delta z}{\rho}$ , 那么瞬时变化率就是对其求极限  $\lim_{\rho \rightarrow 0} \frac{\Delta z}{\rho}$ , 我们称此极限为  $z = f(x, y)$  在  $M_0$  沿射线  $l$  的方向导数, 记为  $\frac{\partial z}{\partial l}|_{M_0}$ , 即  $\frac{\partial z}{\partial l}|_{M_0} \doteq \lim_{\rho \rightarrow 0} \frac{\Delta z}{\rho}$ 。

为了进一步计算方向导数, 我们首先给出向量模的定义: 设有向量  $\vec{a} = \{a_1, b_1, c_1\}$ , 那么其模为  $|\vec{a}| = \sqrt{a_1^2 + b_1^2 + c_1^2}$ , 如果  $\vec{a}$  的模为 1, 那么我们将称其为单位向量。如果  $\vec{a}$  的模不等于 0, 那我们可以将其进行单位化  $\vec{a}^o = \frac{\vec{a}}{|\vec{a}|} = \left\{ \frac{a_1}{|\vec{a}|}, \frac{b_1}{|\vec{a}|}, \frac{c_1}{|\vec{a}|} \right\}$ 。另外, 我们给出方向角与方向余弦的定义: 设有向量  $\vec{a} = \{a_1, b_1\}$ , 其与  $x$  轴正向的夹角为  $\alpha$ , 与  $y$  轴正向的夹角为  $\beta$ , 那么我们将角  $\alpha$  与角  $\beta$  称作方向角, 将  $\cos \alpha = \frac{a_1}{|\vec{a}|}$ 、 $\cos \beta = \frac{b_1}{|\vec{a}|}$  称作方向余弦, 不难发现  $\vec{a}^o = \{\cos \alpha, \cos \beta\}$ , 同理可推广至高维。

#### 3. 方向导数的计算

我们定义曲面  $\Sigma : z = f(x, y)$ , 其中  $(x, y) \in D$ , 且定义点  $M_0 : (x_0, y_0) \in D$ 。过  $M_0$  作射线  $l$ , 我们有  $M' : (x_0 + \Delta x, y_0 + \Delta y) \in l$ 。方向导数为:  $\frac{\partial z}{\partial l}|_{M_0} = \frac{\partial z}{\partial x}|_{M_0} \times \cos \alpha + \frac{\partial z}{\partial y}|_{M_0} \times \cos \beta$ 。

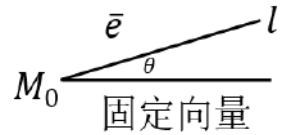
#### 4. 梯度

我们以三元为例:

$$\frac{\partial u}{\partial l}|_{M_0} = \frac{\partial u}{\partial x}|_{M_0} \times \cos \alpha + \frac{\partial u}{\partial y}|_{M_0} \times \cos \beta + \frac{\partial u}{\partial z}|_{M_0} \times \cos \gamma = \left\{ \frac{\partial u}{\partial x}|_{M_0}, \frac{\partial u}{\partial y}|_{M_0}, \frac{\partial u}{\partial z}|_{M_0} \right\} \cdot \{\cos \alpha, \cos \beta, \cos \gamma\}$$

$\{\frac{\partial u}{\partial x}|_{M_0}, \frac{\partial u}{\partial y}|_{M_0}, \frac{\partial u}{\partial z}|_{M_0}\}$  为一个固定向量, 我们将其称为  $u = f(x, y, z)$  在  $M_0$  处的梯度, 其取值与  $l$  无关, 若  $l$  的方向与梯度方向相同, 那么方向导数最大, 函数增长最快。 $\{\cos \alpha, \cos \beta, \cos \gamma\}$  是一个单位向量, 方向与  $l$  相同。我们根据向量点积的计算公式可知:

$$\frac{\partial u}{\partial l}|_{M_0} = \left| \left\{ \frac{\partial u}{\partial x}|_{M_0}, \frac{\partial u}{\partial y}|_{M_0}, \frac{\partial u}{\partial z}|_{M_0} \right\} \right| \times |\vec{e}| \times \cos \theta = \sqrt{\left( \frac{\partial u}{\partial x} \right)^2 + \left( \frac{\partial u}{\partial y} \right)^2 + \left( \frac{\partial u}{\partial z} \right)^2}|_{M_0} \times \cos \theta$$



而  $\sqrt{\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial u}{\partial z}\right)^2}|_{M_0}$  是一个常数，所以方向导数最终的结果与  $\theta$  有关。当  $\theta = 0$  时， $\frac{\partial u}{\partial l}|_{M_0}$  取最大值。

### ☆ np.mgrid() 的用法

功能：返回多维结构，常见的如2D图形，3D图形。其常见使用方法为：`np.mgrid[第1维, 第2维, 第3维, ...]`，其中第 n 维的书写形式为：`a:b:c`, c 表示步长，为实数表示间隔；区间长度为 [a,b)，左闭右开。或第 n 维的书写形式为：`a:b:cj`，cj 为复数表示点数，区间长度为 [a,b]，左闭右闭。举例说明：

```

1 # 生成1D数组
2 import numpy as np
3 # 在[-4,4]区间内取3个值
4 a = np.mgrid[-4:4:3j]
5 a
6 >>> array([-4., 0., 4.])
7 # 生成2D数组,生成的是3*2的矩阵
8 x, y = np.mgrid[1:3:3j, 4:5:2j]
9 x
10 # x向右广播
11 >>> array([[1., 1.],
12             [2., 2.],
13             [3., 3.]])
14 y
15 # y向下广播
16 >>> array([[4., 5.],
17             [4., 5.],
18             [4., 5.]])
19 # 最终结果
20 >>> [[(1,4),(1,5)],
21           [(2,4),(2,5)],
22           [(3,4),(3,5)]]
```

### ☆ numpy.meshgrid() 的用法

该函数用于生成**网格点坐标矩阵**，使用方法为：`X,Y = numpy.meshgrid(x, y)`，其中输入的 **x, y**，就是**网格点**的横纵坐标列向量（非矩阵），输出的 **X, Y**，就是**坐标矩阵**，也即把两个数组的笛卡尔积内的元素的第一二个坐标分别放入两个矩阵中。参考博客见[链接](#)。

```

1 X, Y = np.meshgrid(np.array([0, 1, 2]), np.array([0, 1]))
2 X
3 >>> array([[0, 1, 2],
4             [0, 1, 2]])
5 Y
6 >>> array([[0, 0, 0],
7             [1, 1, 1]])
8 # 也即一共可以表示平面坐标系中的6个点:
9 [(0, 0), (1, 0), (2, 0),
10  (0, 1), (1, 1), (2, 1)]
```

## ☆ zip() 的用法

`zip()` 函数用于将可迭代的对象作为参数，将对象中对应的元素打包成一个个元组，然后返回由这些元组组成的对象，这样做的好处是节约了不少的内存。我们可以使用 `list()` 转换来输出列表。如果各个迭代器的元素个数不一致，则返回列表长度与最短的对象相同，利用 \* 号操作符，可以将元组解压为列表。具体使用方法为：`zip([iterable, ...])`，其中 `iterable` 表示一个或多个迭代器。

```
1 a = [1, 2, 3]
2 b = [4, 5, 6]
3 c = [4, 5, 6, 7, 8]
4 zipped = zip(a, b)
5 # 返回一个对象
6 zipped
7 >>> <zip object at 0x000002522625F748>
8 # list() 转换为列表
9 list(zipped)
10 >>> [(1, 4), (2, 5), (3, 6)]
11 # 元素个数与最短的列表一致
12 list(zip(a, c))
13 >>> [(1, 4), (2, 5), (3, 6)]
14 # 与 zip 相反, zip(*) 可理解为解压, 返回二维矩阵式
15 a1, a2 = zip(*zip(a, b))
16 list(a1)
17 >>> [1, 2, 3]
18 list(a2)
19 >>> [4, 5, 6]
```

本章代码详见GitHub链接[wzy6642](#)。

# 第7章 计算性能

在深度学习中，数据集通常很大而且模型计算往往很复杂。因此，我们十分关注计算性能。本章将重点介绍影响计算性能的重要因子：命令式编程、符号式编程、[异步计算](#)、自动并行计算和多GPU计算。通过本章的学习，你将很可能进一步提升前几章已实现的模型的计算性能，例如，在不影响模型精度的前提下减少模型的训练时间。

## 7.1 命令式编程和符号式编程

本书到目前为止一直都在使用命令式编程，它使用编程语句改变程序状态。考虑下面这段简单的命令式程序。

```
1 In [1]:
2     def add(a, b):
3         return a+b
4     def fancy_func(a, b, c, d):
5         e = add(a, b)
6         f = add(c, d)
7         g = add(e, f)
8         return g
9     fancy_func(1, 2, 3, 4)
10 Out [1]:
11     10
```

和我们预期的一样，在运行语句 `e = add(a, b)` 时，Python会做加法运算并将结果存储在变量 `e` 中，从而令程序的状态发生改变。类似地，后面的两条语句 `f = add(c, d)` 和 `g = add(e, f)` 会依次做加法运算并存储变量。

虽然使用命令式编程很方便，但它的运行可能很慢。一方面，即使 `fancy_func` 函数中的 `add` 是被重复调用的函数，Python也会逐一执行这3条函数调用语句。另一方面，我们需要保存变量 `e` 和 `f` 的值直到 `fancy_func` 中所有语句执行结束。这是因为在执行 `e = add(a, b)` 和 `f = add(c, d)` 这2条语句之后我们并不知道变量 `e` 和 `f` 是否会被程序的其他部分使用。

与命令式编程不同，符号式编程通常在计算流程完全定义好后才被执行。多个深度学习框架，如 **Theano** 和 **TensorFlow**，都使用了符号式编程。通常，符号式编程的程序需要下面3个步骤：

1. 定义计算流程；
2. 把计算流程编译成可执行的程序；
3. 给定输入，调用编译好的程序执行。

下面我们用符号式编程重新实现本节开头给出的命令式编程代码。

```
1 In [2]:  
2     def add_str():  
3         return ''  
4     def add(a, b):  
5         return a+b  
6         ...  
7     def fancy_func_str():  
8         return ''  
9     def fancy_func(a, b, c, d):  
10        e = add(a, b)  
11        f = add(c, d)  
12        g = add(e, f)  
13        return g  
14        ...  
15    def evoke_str():  
16        return add_str() + fancy_func_str() + ''  
17    print(fancy_func(1, 2, 3, 4))  
18    ...  
19    # 计算流程  
20    prog = evoke_str()  
21    print(prog)  
22    # 编译  
23    y = compile(prog, '', 'exec')  
24    # 执行  
25    exec(y)  
26 Out [2]:  
27     def add(a, b):  
28         return a+b  
29  
30     def fancy_func(a, b, c, d):  
31        e = add(a, b)  
32        f = add(c, d)  
33        g = add(e, f)  
34        return g  
35  
36    print(fancy_func(1, 2, 3, 4))  
37  
38    10
```

以上定义的3个函数都仅以字符串的形式返回计算流程。最后，我们通过 `compile` 函数编译完整的计算流程并运行。由于在编译时系统能够完整地获取整个程序，因此有更多空间优化计算。例如，编译的时候可以将程序改写成 `print((1 + 2) + (3 + 4))`，甚至直接改写成 `print(10)`。这样不仅减少了函数调用，还节省了内存。

对比这两种编程方式，我们可以看到以下两点。

- 命令式编程更方便。当我们在Python里使用命令式编程时，大部分代码编写起来都很直观。同时，命令式编程更容易调试。这是因为我们可以很方便地获取并打印所有的中间变量值，或者使用Python的调试工具。
- 符号式编程更高效并更容易移植。一方面，在编译的时候系统容易做更多优化；另一方面，符号式编程可以将程序变成一个与Python无关的格式，从而可以使程序在非Python环境下运行，以避开Python解释器的性能问题。

大部分深度学习框架在命令式编程和符号式编程之间二选一。例如，**Theano**和**受其启发的后来者TensorFlow**使用了**符号式编程**，**Chainer**和它的追随者**PyTorch**使用了**命令式编程**，而**Gluon**则采用了**混合式编程的方式**。

## 7.2 自动并行计算

默认情况下，PyTorch中的GPU操作是异步的。当调用一个使用GPU的函数时，这些操作会在特定的设备上排队，但不一定会在稍后执行。这允许我们并行更多的计算，包括CPU或其他GPU上的操作。下面看一个简单的例子。

首先导入本节中实验所需的包或模块。注意，需要至少2块GPU才能运行本节实验。

```
1 In [1]:  
2     import time  
3     import torch  
4     assert torch.cuda.device_count() >= 2
```

我们先实现一个简单的计时类。

```
1 In [2]:  
2     class Benchmark():  
3         def __init__(self, prefix=None):  
4             # 打印的前置字符串  
5             self.prefix = prefix + ' ' if prefix else ''  
6         def __enter__(self):  
7             self.start = time.time()  
8         def __exit__(self, *args):  
9             print('%stime: %.4f sec' % (self.prefix,  
10                  time.time()-self.start))
```

再定义 `run` 函数，令它做20000次矩阵乘法。

```
1 In [3]:  
2     def run(x):  
3         for _ in range(20000):  
4             y = torch.mm(x, x)
```

接下来，分别在两块GPU上创建 `Tensor`。

```
1 In [4]:  
2     x_gpu1 = torch.rand(size=(100, 100), device='cuda:0')  
3     x_gpu2 = torch.rand(size=(100, 100), device='cuda:1')
```

然后，分别使用它们运行 run 函数并打印[运行所需时间](#)。

```
1 In [5]:  
2     with Benchmark('Run on GPU1.'):   
3         run(x_gpu1)  
4             # 同步操作  
5             torch.cuda.synchronize()  
6     with Benchmark('Run on GPU2.'):   
7         run(x_gpu2)  
8             torch.cuda.synchronize()  
9 Out [5]:  
10    Run on GPU1. time: 0.8804 sec  
11    Run on GPU2. time: 0.9097 sec
```

尝试系统能自动并行这两个任务：

```
1 In [6]:  
2     with Benchmark('Run on both GPU1 and GPU2 in parallel.'):   
3         run(x_gpu1)  
4         run(x_gpu2)  
5         torch.cuda.synchronize()  
6 Out [6]:  
7     Run on both GPU1 and GPU2 in parallel. time: 0.9803 sec
```

这两步计算之间没有依赖关系，因此系统可以选择并行执行它们。可以看到，当两个计算任务一起执行时，执行总时间小于它们分开执行的总和。这表明，PyTorch能有效地实现在不同设备上自动并行计算。

## 7.3 多GPU计算

本节中我们将展示如何使用多块GPU计算，例如，使用多块GPU训练同一个模型。正如所期望的那样，运行本节中的程序需要至少2块GPU。事实上，一台机器上安装多块GPU很常见，这是因为主板上通常会有多个PCIe插槽。如果正确安装了NVIDIA驱动，我们可以通过在命令行输入 nvidia-smi 命令来查看当前计算机上的全部GPU（或者在jupyter notebook中运行 !nvidia-smi）。

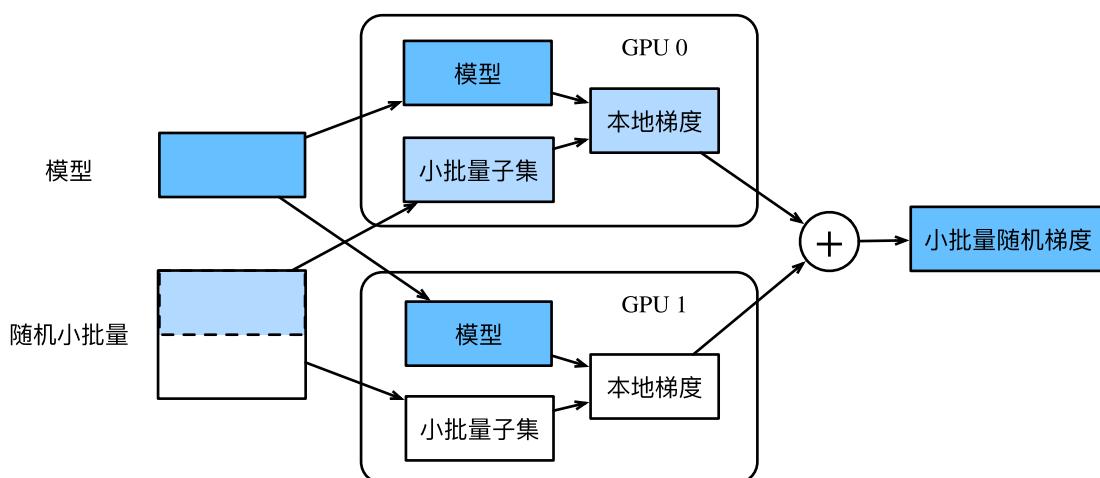
```
Every 10.0s: nvidia-smi  
:0 Dec 24 16:03:40 2020  
Thu Dec 24 16:04:00 2020  
+-----+  
| NVIDIA-SMI 450.57      Driver Version: 450.57      CUDA Version: 11.0 |  
+-----+  
| GPU  Name        Persistence-M| Bus-Id      Disp.A  Volatile Uncorr. ECC |  
| Fan  Temp  Perf  Pwr:Usage/Cap| Memory-Usage | GPU-Util  Compute M. |  
|          %          %          %          %          %          %          %          |  
+-----+  
| 0  TITAN V        Off        00000000:02:00.0 Off |           0%     Default      N/A |  
| 33%  47C   P8    28W / 250W |        4MiB / 12065MiB |           |  
+-----+  
| 1  TITAN V        Off        00000000:82:00.0 Off |           0%     Default      N/A |  
| 45%  60C   P8    32W / 250W |        4MiB / 12066MiB |           |  
+-----+  
+-----+  
| Processes:          PID  Type  Process name          GPU Memory |  
|          ID  ID          |          | Usage          |  
+-----+  
| No running processes found |          |          |  
+-----+
```

PyTorch中的大部分运算可以使用所有的CPU的全部计算资源，或者单块GPU的全部计算资源。但如果使用多块GPU训练模型，我们仍然需要实现相应的算法。这些算法中最常用的叫作数据并行。

### 7.3.1 数据并行

数据并行目前是深度学习里使用最广泛的将模型训练任务划分到多块GPU的方法。回忆一下我们在6.3节（小批量随机梯度下降）一节中介绍的使用优化算法训练模型的过程。下面我们就以小批量随机梯度下降为例来介绍数据并行是如何工作的。

假设一台机器上有 $k$ 块GPU。给定需要训练的模型，每块GPU及其相应的显存将分别独立维护一份完整的模型参数。在模型训练的任意一次迭代中，给定一个随机小批量，我们将该批量中的样本划分成 $k$ 份并分给每块显卡的显存一份。然后，每块GPU将根据相应显存所分到的小批量子集和所维护的模型参数分别计算模型参数的本地梯度。接下来，我们把 $k$ 块显卡的显存上的本地梯度相加，便得到当前的小批量随机梯度。之后，每块GPU都使用这个小批量随机梯度分别更新相应显存所维护的那一份完整的模型参数。下图描绘了使用2块GPU的数据并行下的小批量随机梯度的计算。



### 7.3.2 多GPU计算

先定义一个模型：

```
1 In [1]:  
2     net = torch.nn.Linear(10, 1).cuda()  
3     net  
4 Out [1]:  
5     Linear(in_features=10, out_features=1, bias=True)
```

要想使用PyTorch进行多GPU计算，最简单的方法是直接将模型传入`torch.nn.DataParallel`函数：

```
1 In [2]:  
2     net = torch.nn.DataParallel(net)  
3     net  
4 Out [2]:  
5     DataParallel(  
6         (module): Linear(in_features=10, out_features=1, bias=True)  
7     )
```

这时，默认所有存在的GPU都会被使用。

如果我们机子中有很多GPU，但我们只想使用0、3号显卡，那么我们可以用参数`device_ids`指定即可：`torch.nn.DataParallel(net, device_ids=[0, 3])`。

### 7.3.3 多GPU模型的保存与加载

我们现在来尝试一下按照3.5节（读取和存储）推荐的方式进行一下模型的保存与加载。保存模型：

```
1 In [3]:  
2     torch.save(net.state_dict(), '7.3_model.pt')
```

加载模型前我们一般要先进行一下模型定义，此时的 new\_net 并没有使用多GPU：

```
1 In [4]:  
2     new_net = torch.nn.Linear(10, 1)  
3     new_net.load_state_dict(torch.load('7.3_model.pt'))  
4 Out [4]:  
5     RuntimeError: Error(s) in loading state_dict for Linear:  
6         Missing key(s) in state_dict: "weight", "bias".  
7         Unexpected key(s) in state_dict: "module.weight", "module.bias".
```

事实上 DataParallel 也是一个 nn.Module，只是这个类其中有一个 module 就是传入的实际模型。因此当我们调用 DataParallel 后，模型结构变了（在外面加了一层而已，从 7.3.2 节两个输出可以对比看出来）。所以直接加载肯定会报错的，因为模型结构对不上。

所以正确的方法是保存的时候只保存 net.module：

```
1 In [5]:  
2     # 通用的存储、读取方式  
3     torch.save(net.module.state_dict(), '7.3_model.pt')  
4     new_net.load_state_dict(torch.load('7.3_model.pt'))  
5 Out [5]:  
6     IncompatibleKeys(missing_keys=[], unexpected_keys=[])
```

## 7.4 本章附录

### ☆ compile() 的用法

compile(source, filename, mode[, flags[, dont\_inherit]])。该函数的作用是将 source 编译为代码或 AST 对象。代码对象能够通过 exec 语句来执行或者 eval() 进行求值。

参数	说明
source	字符串或者AST (Abstract Syntax Trees) 对象 (抽象语法树)。
filename	代码文件名称，如果不是从文件读取代码则传递一些可辨认的值，例如 ''。
mode	指定编译代码的种类。可以指定为 exec, eval, single。

备注：exec 执行储存在字符串或文件中的 Python 语句，相比于 eval，exec 可以执行更复杂的 Python 代码。eval() 函数用来执行一个字符串表达式，并返回表达式的值。

```
1 str = 'for i in range(0, 10): print(i)'  
2 c = compile(str, '', 'exec')  
3 c  
4 >>> <code object <module> at 0x000002521F819150, file "", line 1>  
5 exec(c)  
6 >>>
```

```
7 0
8 1
9 2
10 3
11 4
12 5
13 6
14 7
15 8
16 9
17 str = '3*4+5'
18 a = compile(str, '', 'eval')
19 eval(a)
20 >>> 17
```

本章代码详见GitHub链接[wzy6642](#)。

## 第8章 计算机视觉

无论是医疗诊断、无人车、摄像监控，还是智能滤镜，计算机视觉领域的诸多应用都与我们当下和未来的生活息息相关。近年来，深度学习技术深刻推动了计算机视觉系统性能的提升。可以说，当下最先进的计算机视觉应用几乎离不开深度学习。鉴于此，本章将关注计算机视觉领域，并从中挑选时下在学术界和工业界具有影响力的方法与应用来展示深度学习的魅力。

我们在“卷积神经网络”一章中已经介绍了计算机视觉领域常使用的深度学习模型，并实践了简单的图像分类任务。在本章的开头，我们介绍两种有助于提升模型的泛化能力的方法，即**图像增广**和**微调**，并把它们应用于图像分类。由于深度神经网络能够对图像逐级有效地进行表征，这一特性被广泛地应用在目标检测、语义分割和样式迁移这些主流计算机视觉任务中，并取得了成功。围绕这一核心思想，首先，我们将描述目标检测的工作流程与各类方法。之后，我们将探究如何使用全卷积网络对图像做语义分割。接下来，我们再解释如何使用样式迁移技术生成像本书封面一样的图像。最后，我们在两个计算机视觉的重要数据集上实践本章和前几章的内容。

### 8.1 图像增广

在4.6节（深度卷积神经网络）里我们提到过，大规模数据集是成功应用深度神经网络的前提。图像增广（image augmentation）技术通过对训练图像做一系列随机改变，来产生相似但又不同的训练样本，从而扩大训练数据集的规模。**图像增广的另一种解释是，随机改变训练样本可以降低模型对某些属性的依赖，从而提高模型的泛化能力。**例如，我们可以对图像进行不同方式的裁剪，使感兴趣的物体出现在不同位置，从而减轻模型对物体出现位置的依赖性。我们也可以调整亮度、色彩等因素来降低模型对色彩的敏感度。可以说，在当年AlexNet的成功中，图像增广技术功不可没。本节我们将讨论这个在计算机视觉里被广泛使用的技术。

首先，导入实验所需的包或模块。

```
1 In [1]:
2     import torch
3     from torch import nn, optim
4     from torch.utils.data import Dataset, DataLoader
5     import torchvision
6     from PIL import Image
7     import d2lzh as d2l
8     device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

## 8.1.1 常用的图像增广方法

我们来读取一张形状为 $400 \times 500$ （高和宽分别为400像素和500像素）的图像作为实验的样例。

```
1 In [2]:  
2     d2l.set_figsize()  
3     img = Image.open('data/cat1.jpg')  
4     d2l.plt.imshow(img)  
5 Out [2]:  
6     <matplotlib.image.AxesImage at 0x7fac98fa5dd0>
```



下面定义绘图函数 `show_images`。

```
1 In [3]:  
2     def show_images(imgs, num_rows, num_cols, scale=2):  
3         figsize = (num_cols*scale, num_rows*scale)  
4         _, axes = d2l.plt.subplots(num_rows, num_cols, figsize=figsize)  
5         for i in range(num_rows):  
6             for j in range(num_cols):  
7                 axes[i][j].imshow(imgs[i*num_cols+j])  
8                 axes[i][j].axes.get_xaxis().set_visible(False)  
9                 axes[i][j].axes.get_yaxis().set_visible(False)  
10            return axes
```

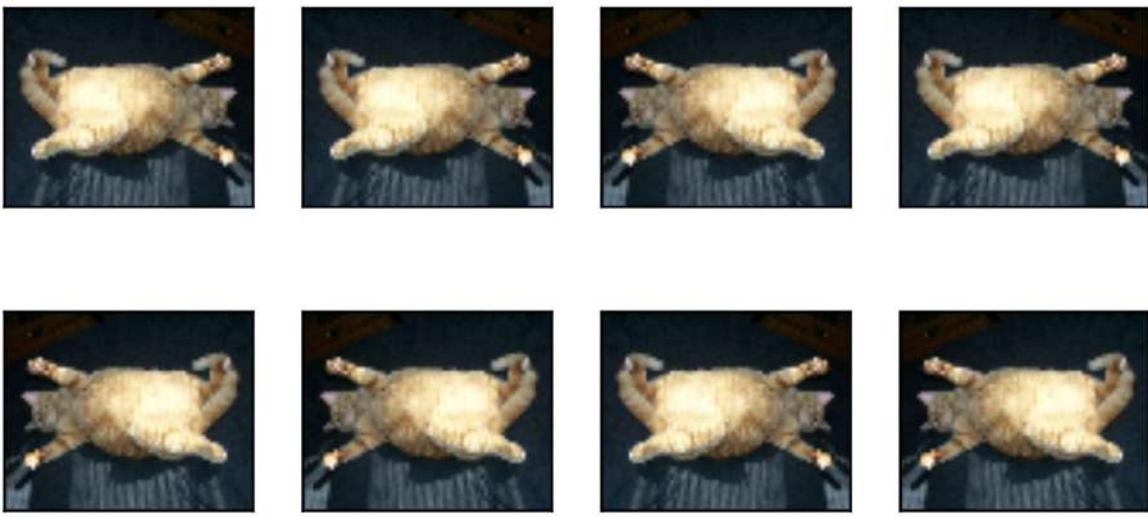
大部分图像增广方法都有一定的随机性。为了方便观察图像增广的效果，接下来我们定义一个辅助函数 `apply`。这个函数对输入图像 `img` 多次运行图像增广方法 `aug` 并展示所有的结果。

```
1 In [4]:  
2     def apply(img, aug, num_rows=2, num_cols=4, scale=1.5):  
3         Y = [aug(img) for _ in range(num_rows*num_cols)]  
4         show_images(Y, num_rows, num_cols, scale)
```

### 1. 翻转和裁剪

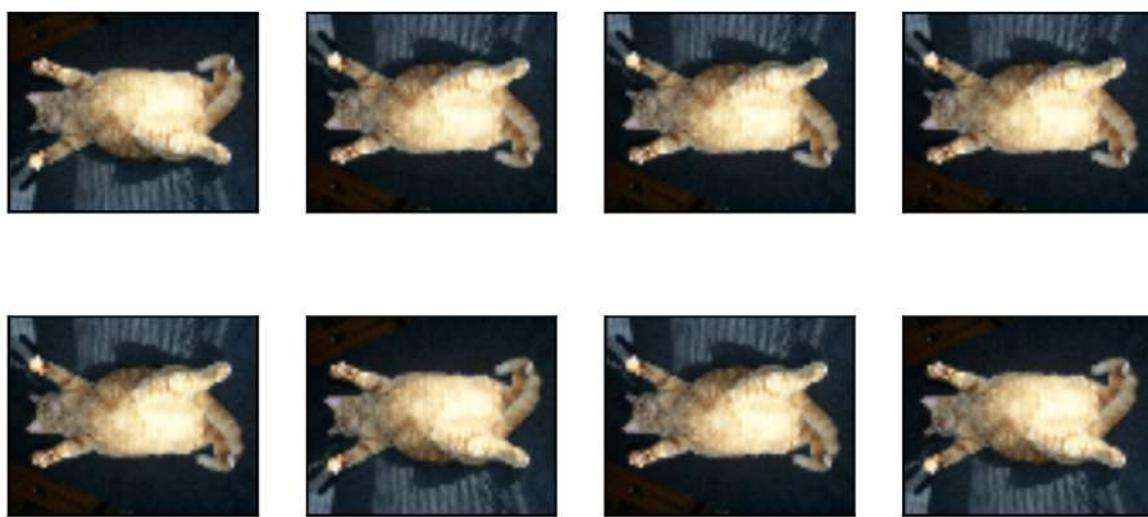
左右翻转图像通常不改变物体的类别。它是最早也是最广泛使用的一种图像增广方法。下面我们通过 `torchvision.transforms` 模块创建 `RandomHorizontalFlip` 实例来实现一半概率的图像水平（左右）翻转。

```
1 In [5]:  
2     apply(img, torchvision.transforms.RandomHorizontalFlip())
```



上下翻转不如左右翻转通用。但是至少对于样例图像，上下翻转不会造成识别障碍。下面我们创建 `RandomVerticalFlip` 实例来实现一半概率的图像垂直（上下）翻转。

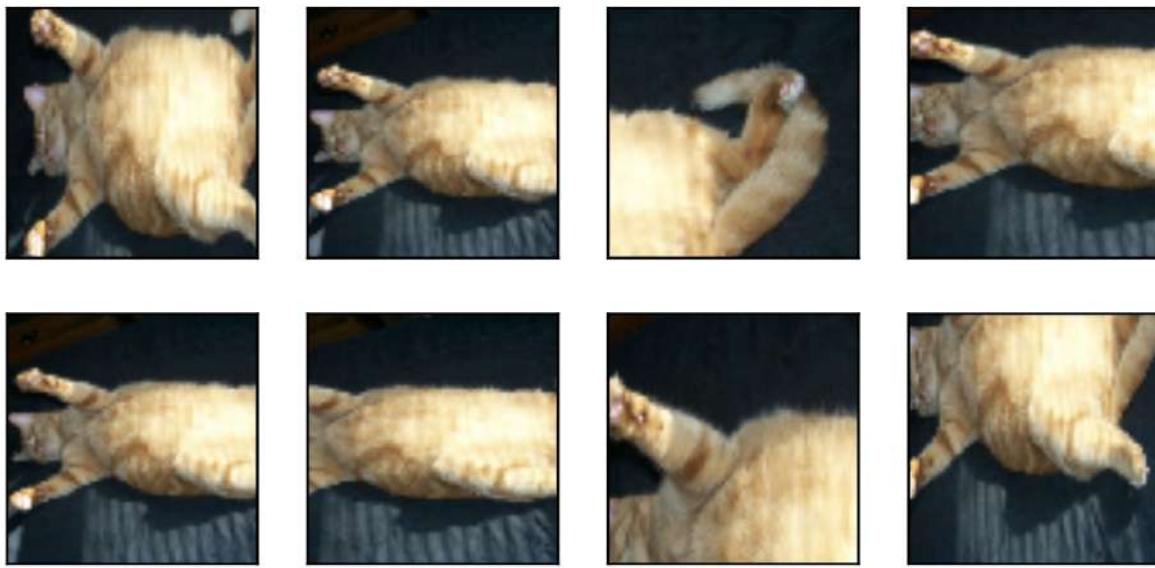
```
1 In [6]:  
2     apply(img, torchvision.transforms.RandomVerticalFlip())
```



在我们使用的样例图像里，猫在图像正中间，但一般情况下可能不是这样。在4.4节（池化层）里我们解释了池化层能降低卷积层对目标位置的敏感度。除此之外，我们还可以通过对图像随机裁剪来让物体以不同的比例出现在图像的不同位置，这同样能够**降低模型对目标位置的敏感性**。

在下面的代码里，我们每次随机裁剪出一块面积为原面积10% ~ 100%的区域，且该区域的宽和高之比随机取自0.5 ~ 2，然后再将该区域的宽和高分别缩放到200像素。若无特殊说明，本节中 $a$ 和 $b$ 之间的随机数指的是从区间 $[a, b]$ 中随机均匀采样所得到的连续值。

```
1 In [7]:  
2     shape_aug = torchvision.transforms.RandomResizedCrop(  
3         200, scale=(0.1, 1), ratio=(0.5, 2))  
4     apply(img, shape_aug)
```



## 2. 变化颜色

另一类增广方法是变化颜色。我们可以从4个方面改变图像的颜色：**亮度**（`brightness`）、**对比度**（`contrast`）、**饱和度**（`saturation`）和**色调**（`hue`）。在下面的例子里，我们将图像的亮度随机变化为原图亮度的50%（即 $1 - 0.5 \sim 150\%$ （即 $1 + 0.5$ ）。

```
1 In [8]:  
2     apply(img, torchvision.transforms.ColorJitter(brightness=0.5))
```



我们也可以随机变化图像的色调。

```
1 In [9]:  
2     apply(img, torchvision.transforms.ColorJitter(hue=0.5))
```



我们也可以同时设置如何随机变化图像的亮度（`brightness`）、对比度（`contrast`）、饱和度（`saturation`）和色调（`hue`）。

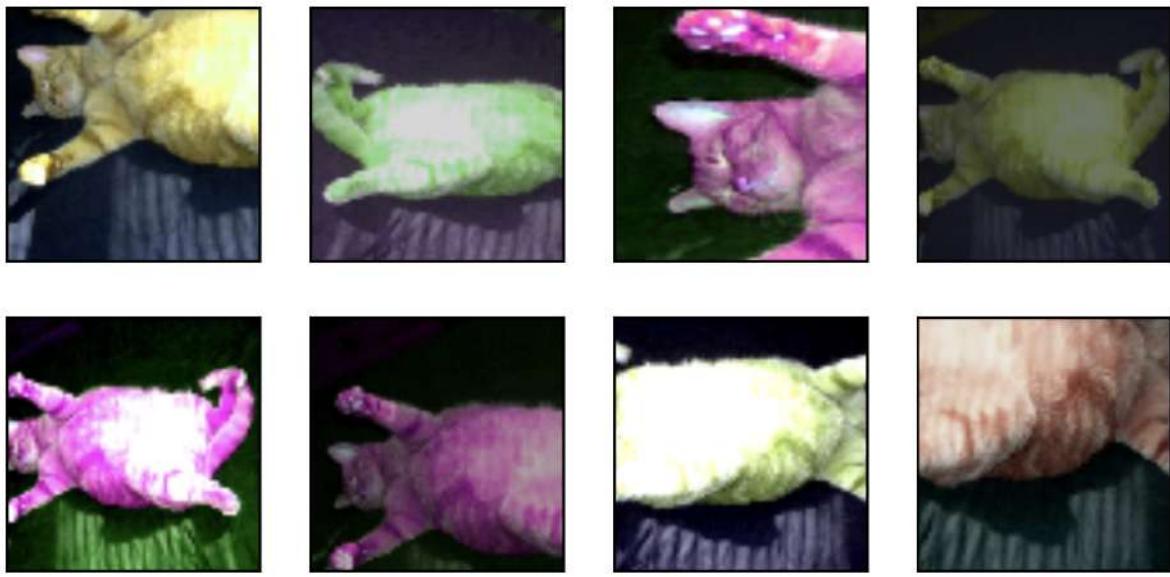
```
1 In [10]:  
2     color_aug = torchvision.transforms.ColorJitter(  
3         brightness=0.5, contrast=0.5,  
4         saturation=0.5, hue=0.5  
5     )  
6     apply(img, color_aug)
```



### 3. 叠加多个图像增广方法

实际应用中我们会将多个图像增广方法叠加使用。我们可以通过 `Compose` 实例将上面定义的多个图像增广方法叠加起来，再应用到每张图像之上。

```
1 In [11]:  
2     augs = torchvision.transforms.Compose([  
3         torchvision.transforms.RandomHorizontalFlip(),  
4         color_aug, shape_aug  
5     ])  
6     apply(img, augs)
```



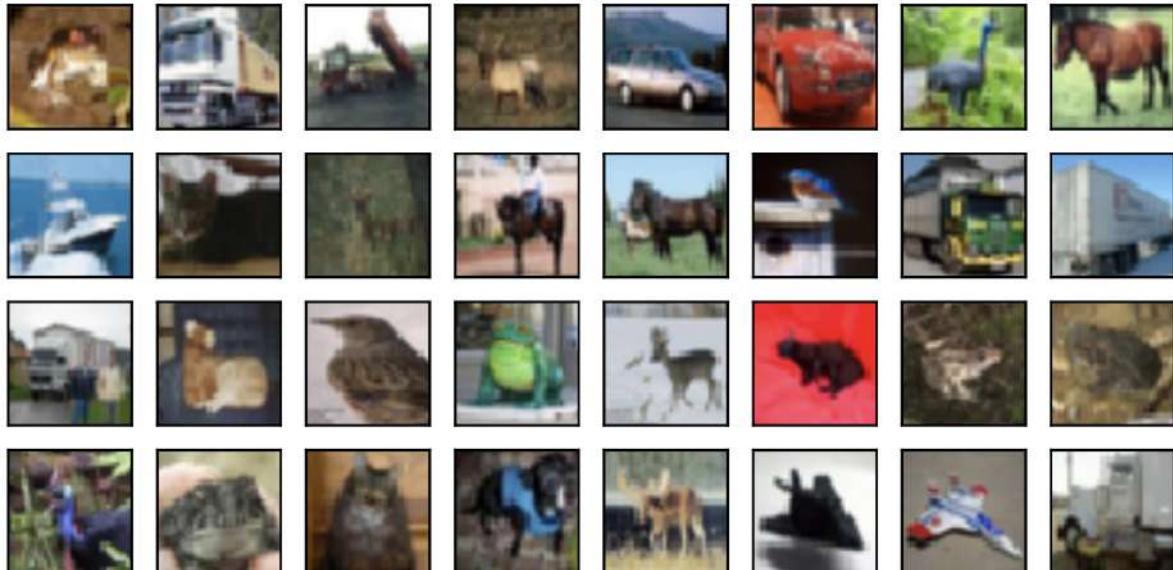
### 8.1.2 使用图像增广训练模型

下面我们来看一个将图像增广应用在实际训练中的例子。这里我们使用CIFAR-10数据集，而不是之前我们一直使用的Fashion-MNIST数据集。这是因为Fashion-MNIST数据集中物体的位置和尺寸都已经经过归一化处理，而CIFAR-10数据集中物体的颜色和大小区别更加显著。下面展示了CIFAR-10数据集中前32张训练图像。

```

1 In [12]:
2     all_imgs = torchvision.datasets.CIFAR10(
3         train=True, root='data/CIFAR-10', download=True)
4     # all_imgs的每一个元素都是(image, label)
5     show_images([all_imgs[i][0] for i in range(32)],
6                 4, 8, scale=0.8)

```



为了在预测时得到确定的结果，我们通常只将图像增广应用在训练样本上，而在预测时使用含随机操作的图像增广。在这里我们只使用最简单的随机左右翻转。此外，我们使用 `ToTensor` 将小批量图像转成PyTorch需要的格式，即形状为（批量大小, 通道数, 高, 宽）、值域在0到1之间且类型为32位浮点数。

```
1 In [13]:  
2     flip_aug = torchvision.transforms.Compose([  
3         torchvision.transforms.RandomHorizontalFlip(),  
4         torchvision.transforms.ToTensor()  
5     ])  
6     no_aug = torchvision.transforms.Compose([  
7         torchvision.transforms.ToTensor()  
8     ])
```

接下来我们定义一个辅助函数来方便读取图像并应用图像增广。有关 `DataLoader` 的详细介绍，可参考更早的2.5节图像分类数据集（Fashion-MNIST）。

```
1 In [14]:  
2     import sys  
3     num_workers = 0 if sys.platform.startswith('win32') else 4  
4     def load_cifar10(is_train, augs,  
5                         batch_size, root='data/CIFAR-10'):br/>6         dataset = torchvision.datasets.CIFAR10(  
7             root=root, train=is_train, transform=augs, download=True)  
8         return DataLoader(  
9             dataset, batch_size=batch_size,  
10            shuffle=is_train, num_workers=num_workers)
```

## 使用多GPU训练模型

我们在CIFAR-10数据集上训练4.11（残差网络（ResNet））一节介绍的ResNet-18模型。我们还将应用7.3.3（多GPU计算）一节中介绍的方法，使用多GPU训练模型。

我们先定义 `train` 函数使用GPU训练并评价模型。

```
1 In [15]:  
2     import time  
3     def train(train_iter, test_iter, net, loss,  
4               optimizer, device, num_epochs):  
5         # 多卡并行  
6         print("Let's use", torch.cuda.device_count(), "GPUs!")  
7         net = torch.nn.DataParallel(net, device_ids=[0, 1])  
8         net = net.cuda()  
9         print("training on", device)  
10        batch_count = 0  
11        for epoch in range(num_epochs):  
12            train_l_sum, train_acc_sum, n = 0.0, 0.0, 0  
13            start = time.time()  
14            for X, y in train_iter:  
15                X = X.to(device)  
16                y = y.to(device)  
17                y_hat = net(X)  
18                l = loss(y_hat, y)  
19                optimizer.zero_grad()  
20                l.backward()  
21                optimizer.step()  
22                train_l_sum += l.cpu().item()  
23                train_acc_sum += (y_hat.argmax(dim=1)==y).sum().cpu().item()  
24                n += y.shape[0]  
25                batch_count += 1
```

```

27     test_acc = d2l.evaluate_accuracy(test_iter, net)
28     print('epoch %d, loss %.4f, train acc %.3f, \
29           test acc %.3f, time %.1f sec' %
30           (epoch+1, train_l_sum/batch_count, train_acc_sum/n,
31            test_acc, time.time()-start))

```

然后就可以定义 `train_with_data_aug` 函数使用图像增广来训练模型了。该函数使用Adam算法作为训练使用的优化算法，然后将图像增广应用于训练数据集之上，最后调用刚才定义的 `train` 函数训练并评价模型。

```

1 In [16]:
2     def train_with_data_aug(train_augs, test_augs, lr=0.001):
3         batch_size, net = 256, d2l.resnet18(10)
4         optimizer = torch.optim.Adam(net.parameters(), lr=lr)
5         loss = torch.nn.CrossEntropyLoss()
6         train_iter = load_cifar10(True, train_augs, batch_size)
7         test_iter = load_cifar10(False, test_augs, batch_size)
8         train(train_iter, test_iter, net, loss, optimizer,
9               device, num_epochs=10)

```

下面使用随机左右翻转的图像增广来训练模型。

```

1 In [17]:
2     train_with_data_aug(flip_aug, no_aug)
3 Out [17]:
4     Files already downloaded and verified
5     Files already downloaded and verified
6     Let's use 2 GPUs!
7     training on cuda
8     epoch 1, loss 1.3540, train acc 0.509, test acc 0.457, time 23.0 sec
9     epoch 2, loss 0.5007, train acc 0.646, test acc 0.548, time 16.7 sec
10    epoch 3, loss 0.2827, train acc 0.702, test acc 0.608, time 16.6 sec
11    epoch 4, loss 0.1870, train acc 0.738, test acc 0.662, time 16.4 sec
12    epoch 5, loss 0.1347, train acc 0.765, test acc 0.694, time 16.2 sec
13    epoch 6, loss 0.1025, train acc 0.786, test acc 0.750, time 17.1 sec
14    epoch 7, loss 0.0795, train acc 0.807, test acc 0.751, time 16.3 sec
15    epoch 8, loss 0.0639, train acc 0.824, test acc 0.759, time 16.5 sec
16    epoch 9, loss 0.0519, train acc 0.837, test acc 0.696, time 16.5 sec
17    epoch 10, loss 0.0432, train acc 0.851, test acc 0.770, time 16.3 sec

```

## 小结:

- 图像增广基于现有训练数据生成随机图像从而应对过拟合。
- 为了在预测时得到确定的结果，通常只将图像增广应用在训练样本上，而不是在预测时使用含随机操作的图像增广。
- 可以从torchvision的 `transforms` 模块中获取有关图片增广的类。

## 8.2 微调

在前面的一些章节中，我们介绍了如何在只有6万张图像的Fashion-MNIST训练数据集上训练模型。我们还描述了学术界当下使用最广泛的大规模图像数据集ImageNet，它有超过1,000万的图像和1,000类的物体。然而，我们平常接触到数据集的规模通常在这两者之间。

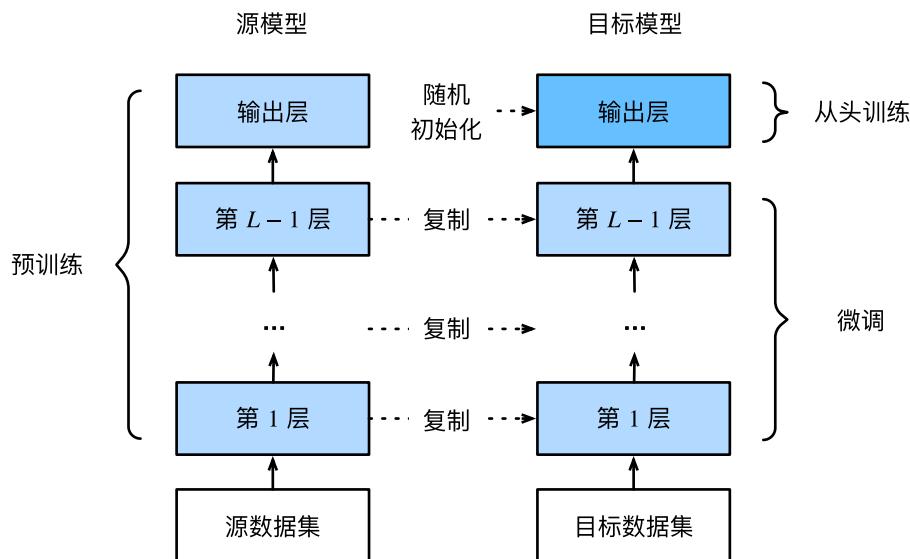
假设我们想从图像中识别出不同种类的椅子，然后将购买链接推荐给用户。一种可能的方法是先找出100种常见的椅子，为每种椅子拍摄1,000张不同角度的图像，然后在收集到的图像数据集上训练一个分类模型。这个椅子数据集虽然可能比Fashion-MNIST数据集要庞大，但样本数仍然不及ImageNet数据集中样本数的十分之一。这可能会导致适用于ImageNet数据集的复杂模型在这个椅子数据集上过拟合。同时，因为数据量有限，最终训练得到的模型的精度也可能达不到实用的要求。

为了应对上述问题，一个显而易见的解决办法是收集更多的数据。然而，收集和标注数据会花费大量的时间和资金。例如，为了收集ImageNet数据集，研究人员花费了数百万美元的研究经费。虽然目前的数据采集成本已降低了不少，但其成本仍然不可忽略。

另外一种解决办法是应用**迁移学习** (transfer learning)，**将从源数据集学到的知识迁移到目标数据集上**。例如，虽然ImageNet数据集的图像大多跟椅子无关，但在该数据集上训练的模型可以抽取较通用的图像特征，从而能够帮助识别边缘、纹理、形状和物体组成等。这些类似的特征对于识别椅子也可能同样有效。

本节我们介绍迁移学习中的一种常用技术：**微调** (fine tuning)。如下图所示，微调由以下4步构成。

1. 在源数据集（如ImageNet数据集）上预训练一个神经网络模型，即源模型。
2. 创建一个新的神经网络模型，即目标模型。它复制了源模型上**除了输出层外的所有模型设计及其参数**。我们假设这些模型参数包含了源数据集上学习到的知识，且这些知识同样适用于目标数据集。我们还假设源模型的输出层跟源数据集的标签紧密相关，因此在目标模型中不予采用。
3. 为目标模型添加一个输出大小为目标数据集类别个数的输出层，并**随机初始化**该层的模型参数。
4. 在目标数据集（如椅子数据集）上训练目标模型。**我们将从头训练输出层，而其余层的参数都是基于源模型的参数微调得到的。**



当目标数据集远小于源数据集时，微调有助于提升模型的泛化能力。

## 热狗识别

接下来我们来实践一个具体的例子：热狗识别。我们将基于一个小数据集对在ImageNet数据集上训练好的ResNet模型进行微调。该小数据集含有数千张包含热狗和不包含热狗的图像。我们将使用微调得到的模型来识别一张图像中是否包含热狗。

首先，导入实验所需的包或模块。torchvision的[models](#)包提供了常用的预训练模型。如果希望获取更多的预训练模型，可以使用[使用pretrained-models.pytorch仓库](#)。

```
1 In [1]:  
2     import torch  
3     from torch import nn, optim  
4     from torch.utils.data import DataLoader, Dataset  
5     import torchvision  
6     from torchvision.datasets import ImageFolder  
7     from torchvision import transforms  
8     from torchvision import models  
9     import os  
10    import sys  
11    import d2lzh as d2l  
12    device = torch.device(  
13        'cuda' if torch.cuda.is_available() else 'cpu')
```

## 1. 获取数据集

我们使用的热狗数据集（[点击下载](#)）是从网上抓取的，它含有1400张包含热狗的正类图像，和同样多包含其他食品的负类图像。各类的1000张图像被用于训练，其余则用于测试。

我们首先将压缩后的数据集下载到路径 `data_dir` 之下，然后在该路径将下载好的数据集解压，得到两个文件夹 `hotdog/train` 和 `hotdog/test`。这两个文件夹下面均有 `hotdog` 和 `not-hotdog` 两个类别文件夹，每个类别文件夹里面是图像文件。

```
1 In [2]:  
2     data_dir = 'data'  
3     # 列出指定路径下的文件夹名称  
4     os.listdir(os.path.join(data_dir, 'hotdog'))  
5 Out [2]:  
6     ['train', 'test']
```

我们创建两个 `ImageFolder` 实例来分别读取训练数据集和测试数据集中的所有图像文件。

```
1 In [3]:  
2     train_imgs = ImageFolder(  
3         os.path.join(data_dir, 'hotdog/train'))  
4     test_imgs = ImageFolder(  
5         os.path.join(data_dir, 'hotdog/test'))
```

下面画出前8张正类图像和最后8张负类图像。可以看到，它们的大小和高宽比各不相同。

```
1 In [4]:  
2     hotdogs = [train_imgs[i][0] for i in range(8)]  
3     not_hotdogs = [train_imgs[-i-1][0] for i in range(8)]  
4     d2l.show_images(hotdogs+not_hotdogs, 2, 8, scale=1.4)
```



在训练时，我们先从图像中裁剪出随机大小和随机高宽比的一块随机区域，然后将该区域缩放为高和宽均为224像素的输入。测试时，我们将图像的高和宽均缩放为256像素，然后从中裁剪出高和宽均为224像素的中心区域作为输入。此外，我们对RGB（红、绿、蓝）三个颜色通道的数值做标准化：**每个数值减去该通道所有数值的平均值，再除以该通道所有数值的标准差作为输出。**

注：在使用预训练模型时，一定要和预训练时作同样的预处理。如果你使用的是 torchvision 的 `models`，那就要求：All pre-trained models expect input images normalized in the same way, i.e. mini-batches of 3-channel RGB images of shape  $(3 \times H \times W)$ , where  $H$  and  $W$  are expected to be at least 224. The images have to be loaded in to a range of [0, 1] and then normalized using  $\text{mean} = [0.485, 0.456, 0.406]$  and  $\text{std} = [0.229, 0.224, 0.225]$ . 如果你使用的是 `pretrained-models.pytorch` 仓库，请**务必**阅读其 README，其中说明了如何预处理。

```
1 In [5]:  
2     # 指定RGB三个通道的均值和方差来将图像通道归一化  
3     normalize = transforms.Normalize(  
4         mean = [0.485, 0.456, 0.406],  
5         std = [0.229, 0.224, 0.225]  
6     )  
7     train_augs = transforms.Compose([  
8         # 统一大小  
9         transforms.RandomResizedCrop(size=224),  
10        # 水平翻转  
11        transforms.RandomHorizontalFlip(),  
12        # 值域0到1之间  
13        transforms.ToTensor(),  
14        # 标准化  
15        normalize  
16    ])  
17     test_augs = transforms.Compose([  
18         transforms.Resize(size=256),  
19         # 中心裁剪  
20         transforms.CenterCrop(size=224),  
21         transforms.ToTensor(),  
22         normalize  
23    ])
```

## 2. 定义和初始化模型

我们使用在ImageNet数据集上预训练的ResNet-18作为源模型。这里指定 `pretrained=True` 来自动下载并加载预训练的模型参数。在第一次使用时需要联网下载模型参数。

```
1 In [6]:  
2     pretrained_net = models.resnet18(pretrained=True)
```

不管是使用的torchvision的 `models` 还是 `pretrained-models.pytorch` 仓库，默认都会将预训练好的模型参数下载到你的home目录下 `torch` 文件夹。你可以通过修改环境变量 `$TORCH_MODEL_ZOO` 来更改下载目录：`export TORCH_MODEL_ZOO="/local/pretrainedmodels` 另外我比较常使用的方法是，在其源码中找到下载地址直接浏览器输入地址下载，下载好后将其放到环境变量 `$TORCH_MODEL_ZOO` 所指文件夹即可，这样比较快。

下面打印源模型的成员变量 `fc`。作为一个全连接层，它将ResNet最终的全局平均池化层输出变换为ImageNet数据集上1000类的输出。

```
1 In [7]:  
2     print(pretrained_net.fc)  
3 Out [7]:  
4     Linear(in_features=512, out_features=1000, bias=True)
```

注: 如果你使用的是其他模型, 那可能没有成员变量 `fc` (比如 `models` 中的 VGG 预训练模型), 所以正确做法是查看对应模型源码中其定义部分, 这样既不会出错也能加深我们对模型的理解。  
`pretrained-models.pytorch` 仓库貌似统一了接口, 但是我还是建议使用时查看一下对应模型的源码。

可见此时 `pretrained_net` 最后的输出个数等于目标数据集的类别数 1000。所以我们应该将最后的 `fc` 层修改为我们需要的输出类别数:

```
1 In [8]:  
2     pretrained_net.fc = nn.Linear(512, 2)  
3     print(pretrained_net.fc)  
4 Out [8]:  
5     Linear(in_features=512, out_features=2, bias=True)
```

此时, `pretrained_net` 的 `fc` 层就被随机初始化了, 但是其他层依然保存着预训练得到的参数。由于是在很大的 ImageNet 数据集上预训练的, 所以参数已经足够好, 因此一般只需使用较小的学习率来微调这些参数, 而 `fc` 中的随机初始化参数一般需要更大的学习率从头训练。PyTorch 可以方便的对模型的不同部分设置不同的学习参数, 我们在下面代码中将 `fc` 的学习率设为已经预训练过的部分的 10 倍。

```
1 In [9]:  
2     output_params = list(map(id, pretrained_net.fc.parameters()))  
3     feature_params = filter(  
4         lambda p: id(p) not in output_params,  
5         pretrained_net.parameters())  
6     lr = 0.01  
7     optimizer = optim.SGD([  
8         {'params': feature_params},  
9         # output 中的模型参数将在迭代中使用 10 倍大的学习率  
10        {'params': pretrained_net.fc.parameters(), 'lr': lr*10}  
11    ], lr=lr, weight_decay=0.001)
```

### 3. 微调模型

我们先定义一个使用微调的训练函数 `train_fine_tuning` 以便多次调用。

```
1 In [10]:  
2     def train_fine_tuning(net, optimizer, batch_size=128, num_epochs=5):  
3         train_iter = DataLoader(  
4             ImageFolder(  
5                 os.path.join(data_dir, 'hotdog/train'),  
6                 transform=train_augs),  
7                 batch_size, shuffle=True)  
8         test_iter = DataLoader(  
9             ImageFolder(  
10                os.path.join(data_dir, 'hotdog/test'),  
11                transform=test_augs),  
12                batch_size)  
13         loss = torch.nn.CrossEntropyLoss()  
14         d2l.train(train_iter, test_iter, net, loss,  
15                   optimizer, device, num_epochs)
```

我们将 `models` 实例中的学习率设得小一点，如 0.01，以便微调预训练得到的模型参数。根据前面的设置，我们将以 10 倍的学习率从头训练目标模型的输出层参数。

```
1 In [11]:  
2     train_fine_tining(pretrained_net, optimizer)  
3 Out [11]:  
4     Let's use 2 GPUs!  
5     training on cuda  
6     epoch 1, loss 4.4752, train acc 0.653, test acc 0.865, time 42.6 sec  
7     epoch 2, loss 0.1845, train acc 0.906, test acc 0.932, time 38.6 sec  
8     epoch 3, loss 0.0978, train acc 0.917, test acc 0.936, time 33.3 sec  
9     epoch 4, loss 0.0526, train acc 0.931, test acc 0.946, time 35.8 sec  
10    epoch 5, loss 0.0498, train acc 0.918, test acc 0.938, time 38.8 sec
```

作为对比，我们定义一个相同的模型，但将它的所有模型参数都初始化为随机值。由于整个模型都需要从头训练，我们可以使用较大的学习率。

```
1 In [12]:  
2     scratch_net = models.resnet18(pretrained=False, num_classes=2)  
3     lr = 0.1  
4     optimizer = optim.SGD(  
5         scratch_net.parameters(), lr=lr, weight_decay=0.001)  
6     train_fine_tining(scratch_net, optimizer)  
7 Out [12]:  
8     Let's use 2 GPUs!  
9     training on cuda  
10    epoch 1, loss 2.8947, train acc 0.615, test acc 0.718, time 38.4 sec  
11    epoch 2, loss 0.3260, train acc 0.762, test acc 0.787, time 38.3 sec  
12    epoch 3, loss 0.1612, train acc 0.794, test acc 0.840, time 37.2 sec  
13    epoch 4, loss 0.1052, train acc 0.818, test acc 0.830, time 39.6 sec  
14    epoch 5, loss 0.0745, train acc 0.838, test acc 0.839, time 37.8 sec
```

可以看到，微调的模型因为参数初始值更好，往往在相同迭代周期下取得更高的精度。

## 小结：

- 迁移学习将从源数据集学到的知识迁移到目标数据集上。微调是迁移学习的一种常用技术。
- 目标模型复制了源模型上除了输出层外的所有模型设计及其参数，并基于目标数据集微调这些参数。而目标模型的输出层需要从头训练。
- 一般来说，微调参数会使用较小的学习率，而从头训练输出层可以使用较大的学习率。

## 8.3 目标检测和边界框

在前面的一些章节中，我们介绍了诸多用于图像分类的模型。在图像分类任务里，我们假设图像里只有一个主体目标，并关注如何识别该目标的类别。然而，很多时候图像里有多个我们感兴趣的目标，我们不仅想知道它们的类别，还想得到它们在图像中的具体位置。在计算机视觉里，我们将这类任务称为目标检测（object detection）或物体检测。

目标检测在多个领域中被广泛使用。例如，在无人驾驶里，我们需要通过识别拍摄到的视频图像里的车辆、行人、道路和障碍的位置来规划行进线路。机器人也常通过该任务来检测感兴趣的目标。安防领域则需要检测异常目标，如歹徒或者炸弹。

在接下来的几节里，我们将介绍目标检测里的多个深度学习模型。在此之前，让我们来了解目标位置这个概念。先导入实验所需的包或模块。

```
1 In [1]:  
2     import d2lzh as d2l  
3     from PIL import Image
```

下面加载本节将使用的示例图像。可以看到图像左边是一只狗，右边是一只猫。它们是这张图像里的两个主要目标。

```
1 In [2]:  
2     d2l.set_figsize()  
3     img = Image.open('data/catdog.jpg')  
4     # 加分号只显示图片  
5     d2l.plt.imshow(img);
```



## 边界框

在目标检测里，我们通常使用边界框（bounding box）来描述目标位置。边界框是一个矩形框，可以由矩形左上角的 $x$ 和 $y$ 轴坐标与右下角的 $x$ 和 $y$ 轴坐标确定。我们根据上面的图的坐标信息来定义图中狗和猫的边界框。图中的坐标原点在图像的左上角，原点往右和往下分别为 $x$ 轴和 $y$ 轴的正方向。

```
1 In [3]:  
2     # bbox是bounding box的缩写  
3     # [左上角x, 左上角y, 右下角x, 右下角y]  
4     dog_bbox, cat_bbox = [60, 45, 378, 516], [400, 114, 655, 493]
```

我们可以在图中将边界框画出来，以检查其是否准确。画之前，我们定义一个辅助函数 `bbox_to_rect`。它将边界框表示成matplotlib的边界框格式。

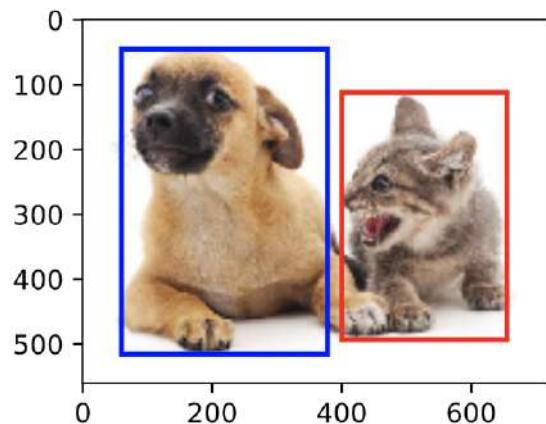
```
1 In [4]:  
2     def bbox_to_rect(bbox, color):  
3         # 将边界框(左上x, 左上y, 右下x, 右下y)格式  
4         # 转换成matplotlib格式: ((左上x, 左上y), 宽, 高)  
5         return d2l.plt.Rectangle(  
6             xy=(bbox[0], bbox[1]), width=bbox[2]-bbox[0],  
7             height=bbox[3]-bbox[1], fill=False,  
8             edgecolor=color, linewidth=2  
9         )
```

我们将边界框加载在图像上，可以看到目标的主要轮廓基本在框内。

```

1 In [5]:
2     fig = d2l=plt.imshow(img)
3     fig.axes.add_patch(bbox_to_rect(dog_bbox, 'blue'));
4     fig.axes.add_patch(bbox_to_rect(cat_bbox, 'red'));

```



### 小结:

- 在目标检测里不仅需要找出图像里面所有感兴趣的目标，而且要知道它们的位置。位置一般由矩形边界框来表示。

## 8.4 锚框

目标检测算法通常会在输入图像中采样大量的区域，然后判断这些区域中是否包含我们感兴趣的目标，并调整区域边缘从而更准确地预测目标的真实边界框（ground-truth bounding box）。不同的模型使用的区域采样方法可能不同。这里我们介绍其中的一种方法：它以每个像素为中心生成多个大小和宽高比（aspect ratio）不同的边界框。这些边界框被称为锚框（anchor box）。我们将在后面基于锚框实践目标检测。注：建议想学习用PyTorch做检测的童鞋阅读一下仓库[a-PyTorch-Tutorial-to-Object-Detection](#)。

先导入一下相关包。

```

1 In [1]:
2     import numpy as np
3     import math
4     import torch

```

### 8.4.1 生成多个锚框

假设输入图像高为 $h$ ，宽为 $w$ 。我们分别以图像的每个像素为中心生成不同形状的锚框。设缩放比为 $s$ （该数值描述基准锚框的大小信息，若取值为 $a$ ，那么基准锚框面积由原图宽和高分别缩小 $a$ 倍所得，也即 $a^2$ ）且宽高比为 $r > 0$ ，我们假设其宽为 $w'$ ，高为 $h'$ ，就有

$$\begin{cases} w' \times h' = s^2 \times w \times h \\ \frac{w'}{h'} = r \end{cases} \Rightarrow \begin{cases} h' = s \times \sqrt{\frac{wh}{r}} \\ w' = s \times \sqrt{rwh} \end{cases}$$

我们对图片的长宽分别做归一化处理，也即 $w = 1, h = 1$ ，此时：

$$h' = s / \sqrt{r}, w' = s \sqrt{r}$$

也即由 $s$ 和 $r$ 可以确定锚框的宽和高，当中心位置给定时，已知面积 $s$ 和宽高比 $r$ 的锚框是确定的。

下面我们分别设定好一组大小 $s_1, \dots, s_n$ 和一组宽高比 $r_1, \dots, r_m$ 。如果以每个像素为中心时使用所有的大小与宽高比的组合，输入图像将一共得到 $whnm$ 个锚框。虽然这些锚框可能覆盖了所有的真实边界框，但计算复杂度容易过高。因此，我们通常只对包含 $s_1$ 或 $r_1$ 的大小与宽高比的组合感兴趣，即

$$(s_1, r_1), (s_1, r_2), \dots, (s_1, r_m), (s_2, r_1), (s_3, r_1), \dots, (s_n, r_1)$$

也就是说，以相同像素为中心的锚框的数量为 $n + m - 1$ 。对于整个输入图像，我们将一共生成 $wh(n + m - 1)$ 个锚框。

以上生成锚框的方法实现在下面的 `MultiBoxPrior` 函数中。指定输入、一组大小和一组宽高比，该函数将返回输入的所有锚框。注：PyTorch官方在[torchvision.models.detection.rpn](#)里有一个 `AnchorGenerator`类可以用来生成anchor，但是和这里讲的不一样，感兴趣的可以去看看。

```
1 In [2]:  
2     d2l.set_figsize()  
3     img = Image.open('data/catdog.jpg')  
4     w, h = img.size  
5     print('w = %d, h = %d' % (w, h))  
6     def MultiBoxPrior(feature_map, sizes=[0.75, 0.5, 0.25],  
7                         ratios=[1, 2, 0.5]):  
8         # 记录锚框面积和宽高比的组合方式  
9         pairs = []  
10        # 包含s1或r1的组合方式  
11        for r in ratios:  
12            pairs.append([sizes[0], math.sqrt(r)])  
13        for s in sizes[1:]:  
14            pairs.append([s, math.sqrt(ratios[0])])  
15        pairs = np.array(pairs)  
16        # size * sqrt(ratio)  
17        # 没有w是因为图片长宽做了归一化  
18        ss1 = pairs[:, 0] * pairs[:, 1]  
19        # size / sqrt(ratio)  
20        ss2 = pairs[:, 0] / pairs[:, 1]  
21        # 锚框左上角、右下角坐标计算  
22        # 除以2是因为像素点在锚框中心位置  
23        base_anchors = np.stack([-ss1, -ss2, ss1, ss2], axis=1) / 2  
24        # 目标图像的高、宽  
25        h, w = feature_map.shape[-2:]  
26        # x轴  
27        shifts_x = np.arange(0, w) / w  
28        # y轴  
29        shifts_y = np.arange(0, h) / h  
30        # 生成图布内所有坐标点  
31        shift_x, shift_y = np.meshgrid(shifts_x, shifts_y)  
32        # 拉成1维，位置相同的元素组成画布内的一个点的表示  
33        shift_x = shift_x.reshape(-1)  
34        shift_y = shift_y.reshape(-1)  
35        shifts = np.stack((shift_x, shift_y, shift_x, shift_y), axis=1)  
36        # 每个像素点添加所有类型的锚框  
37        anchors = shifts.reshape((-1, 1, 4)) + \  
38                  base_anchors.reshape((1, -1, 4))  
39        return torch.tensor(anchors, dtype=torch.float32).view(1, -1, 4)  
40 Out [2]:  
41     w = 728, h = 561  
42     torch.size([1, 2042040, 4])
```

我们看到，返回锚框变量 `y` 的形状为 (1, 锚框个数, 4)。将锚框变量 `y` 的形状变为 (图像高, 图像宽, 以相同像素为中心的锚框个数, 4) 后，我们就可以通过指定像素位置来获取所有以该像素为中心的锚框了。下面的例子里我们访问以 (250, 250) 为中的第一个锚框。它有4个元素，分别是锚框左上角的 $x$ 和 $y$ 轴坐标和右下角的 $x$ 和 $y$ 轴坐标，其中 $x$ 和 $y$ 轴的坐标值分别已除以图像的宽和高，因此值域均为0和1之间。

```
1 In [3]:  
2     boxes = Y.reshape((h, w, 5, 4))  
3     boxes[250, 250, 0, :]  
4 Out [3]:  
5     tensor([-0.0316,  0.0706,  0.7184,  0.8206])
```

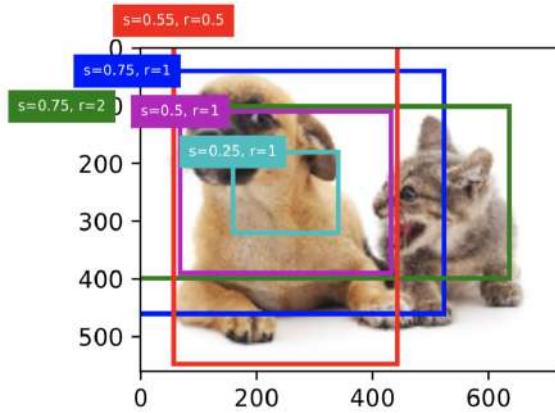
可以验证一下以上输出对不对：size和ratio分别为0.75和1，则（归一化后的）宽高均为0.75，所以输出是正确的 ( $0.75 = 0.7184 + 0.0316 = 0.8206 - 0.0706$ )。

为了描绘图像中以某个像素为中心的所有锚框，我们先定义 `show_bboxes` 函数以便在图像上画出多个边界框。

```
1 In [4]:  
2     def show_bboxes(axes, bboxes, labels=None, colors=None):  
3         # 获取锚框上的标签  
4         def _make_list(obj, default_values=None):  
5             if obj is None:  
6                 obj = default_values  
7             elif not isinstance(obj, (list, tuple)):  
8                 obj = [obj]  
9             return obj  
10    labels = _make_list(labels)  
11    # 框的颜色  
12    colors = _make_list(colors, ['b', 'g', 'r', 'm', 'c'])  
13    for i, bbox in enumerate(bboxes):  
14        color = colors[i % len(colors)]  
15        rect = d2l.bbox_to_rect(  
16            bbox.detach().cpu().numpy(), color)  
17        axes.add_patch(rect)  
18        if labels and len(labels) > i:  
19            text_color = 'k' if color=='w' else 'w'  
20            axes.text(rect.xy[0], rect.xy[1], labels[i],  
21                        va='center', ha='center', fontsize=6,  
22                        color=text_color,  
23                        bbox=dict(facecolor=color, lw=0))
```

刚刚我们看到，变量 `bboxes` 中 $x$ 和 $y$ 轴的坐标值分别已除以图像的宽和高。在绘图时，我们需要恢复锚框的原始坐标值，并因此定义了变量 `bbox_scale`。现在，我们可以画出图像中以(250, 250)为中心的所有锚框了。可以看到，大小为0.75且宽高比为1的锚框较好地覆盖了图像中的狗。

```
1 In [5]:  
2     d2l.set_figsize()  
3     fig = d2l.plt.imshow(img)  
4     bbox_scale = torch.tensor([[w, h, w, h]], dtype=torch.float32)  
5     show_bboxes(fig.axes, boxes[250, 250, :, :] * bbox_scale,  
6                  ['s=0.75, r=1', 's=0.75, r=2', 's=0.55, r=0.5',  
7                   's=0.5, r=1', 's=0.25, r=1'])
```



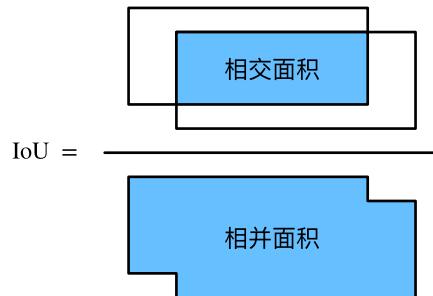
### 8.4.2 交并比

我们刚刚提到某个锚框较好地覆盖了图像中的狗。如果该目标的真实边界框已知，这里的“较好”该如何量化呢？一种直观的方法是衡量锚框和真实边界框之间的相似度。我们知道，Jaccard系数（Jaccard index）可以衡量两个集合的相似度。给定集合 $A$ 和 $B$ ，它们的Jaccard系数即二者交集大小除以二者并集大小：

$$J(\mathcal{A}, \mathcal{B}) = \frac{|\mathcal{A} \cap \mathcal{B}|}{|\mathcal{A} \cup \mathcal{B}|}$$

实际上，我们可以把边界框内的像素区域看成是像素的集合。如此一来，我们可以用两个边界框的像素集合的Jaccard系数衡量这两个边界框的相似度。当衡量两个边界框的相似度时，我们通常将Jaccard系数称为交并比 (Intersection over Union, IoU)，即两个边界框相交面积与相并面积之比，如下图所示。交并比的取值范围在0和1之间：0表示两个边界框无重合像素，1表示两个边界框相等。

在本节的剩余部分，我们将使用交并比来衡量锚框与真实边界框以及锚框与锚框之间的相似度。



```

20     # 右下角取最小
21     # (n1, n2, 2)
22     upper_bounds = torch.min(set_1[:, 2:].unsqueeze(1),
23                               set_2[:, 2:].unsqueeze(0))
24     # (n1, n2, 2)
25     intersection_dims = torch.clamp(
26         upper_bounds - lower_bounds, min=0)
27     # (n1, n2)
28     return intersection_dims[:, :, 0] * intersection_dims[:, :, 1]
29
30
31 def compute_jaccard(set_1, set_2):
32     """
33     计算anchor之间的Jaccard系数(IoU)
34     Args:
35         set_1: a tensor of dimensions (n1, 4),
36                 anchor表示成(xmin, ymin, xmax, ymax)
37         set_2: a tensor of dimensions (n2, 4),
38                 anchor表示成(xmin, ymin, xmax, ymax)
39     Returns:
40         Jaccard Overlap of each of the boxes in set 1 with
41         respect to each of the boxes in set 2, shape: (n1, n2)
42     """
43     # Find intersections
44     # (n1, n2)
45     intersection = compute_intersection(set_1, set_2)
46
47     # Find areas of each box in both sets
48     areas_set_1 = (set_1[:, 2] - set_1[:, 0]) * \
49                     (set_1[:, 3] - set_1[:, 1]) # (n1)
50     areas_set_2 = (set_2[:, 2] - set_2[:, 0]) * \
51                     (set_2[:, 3] - set_2[:, 1]) # (n2)
52
53     # Find the union
54     # PyTorch auto-broadcasts singleton dimensions
55     union = areas_set_1.unsqueeze(1) + \
56             areas_set_2.unsqueeze(0) - intersection # (n1, n2)
57
58     return intersection / union # (n1, n2)

```

### 8.4.3 标注训练集的锚框

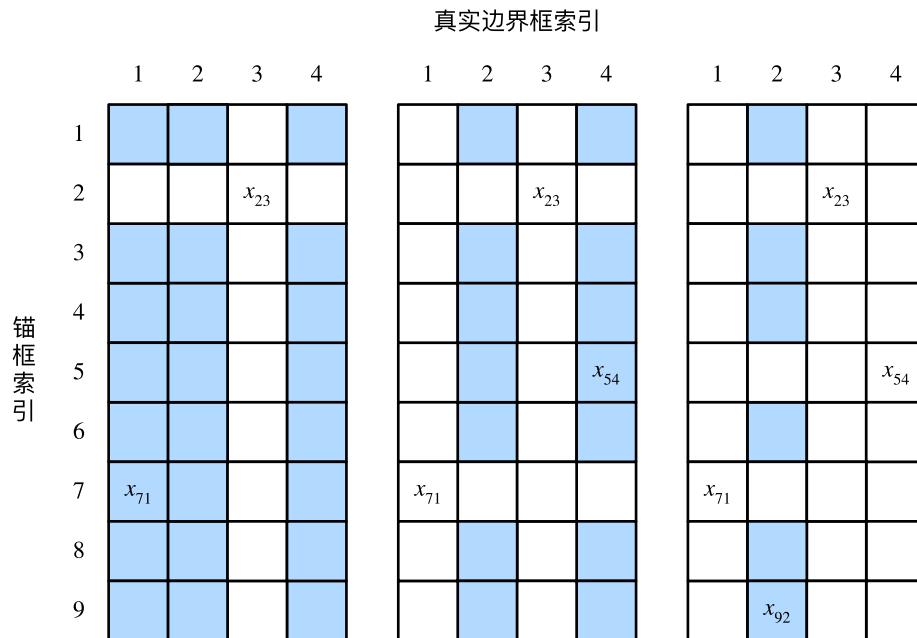
在训练集中，我们将每个锚框视为一个训练样本。为了训练目标检测模型，我们需要为每个锚框标注两类标签：一是锚框所含目标的类别，简称类别；二是真实边界框相对锚框的偏移量，简称偏移量（offset）。在目标检测时，我们首先生成多个锚框，然后为每个锚框预测类别以及偏移量，接着根据预测的偏移量调整锚框位置从而得到预测边界框，最后筛选需要输出的预测边界框。

我们知道，在目标检测的训练集中，每个图像已标注了真实边界框的位置以及所含目标的类别。在生成锚框之后，我们主要依据与锚框相似的真实边界框的位置和类别信息为锚框标注。那么，该如何为锚框分配与其相似的真实边界框呢？

假设图像中锚框分别为 $A_1, A_2, \dots, A_{n_a}$ ，真实边界框分别为 $B_1, B_2, \dots, B_{n_b}$ ，且 $n_a \geq n_b$ 。定义矩阵 $\mathbf{X} \in \mathbb{R}^{n_a \times n_b}$ ，其中第*i*行第*j*列的元素 $x_{ij}$ 为锚框 $A_i$ 与真实边界框 $B_j$ 的交并比。首先，我们找出矩阵 $\mathbf{X}$ 中最大元素，并将该元素的行索引与列索引分别记为 $i_1, j_1$ 。我们为锚框 $A_{i_1}$ 分配真实边界框 $B_{j_1}$ 。显然，锚框 $A_{i_1}$ 和真实边界框 $B_{j_1}$ 在所有的“锚框—真实边界框”的配对中相似度最高。接下来，将矩阵 $\mathbf{X}$ 中第 $i_1$ 行和第 $j_1$ 列上的所有元素丢弃。找出矩阵 $\mathbf{X}$ 中剩余的最大元素，并将该元素的行索引与列索引分别

记为 $i_2, j_2$ 。我们为锚框 $A_{i_2}$ 分配真实边界框 $B_{j_2}$ ，再将矩阵 $\mathbf{X}$ 中第 $i_2$ 行和第 $j_2$ 列上的所有元素丢弃。此时矩阵 $\mathbf{X}$ 中已有两行两列的元素被丢弃。依此类推，直到矩阵 $\mathbf{X}$ 中所有 $n_b$ 列元素全部被丢弃。这个时候，我们已为 $n_b$ 个锚框各分配了一个真实边界框。接下来，我们只遍历剩余的 $n_a - n_b$ 个锚框（因为有 $n_b$ 行元素被抛弃了）：给定其中的锚框 $A_i$ ，根据矩阵 $\mathbf{X}$ 的第 $i$ 行找到与 $A_i$ 交并比最大的真实边界框 $B_j$ ，且只有当该交并比大于预先设定的阈值时，才为锚框 $A_i$ 分配真实边界框 $B_j$ 。

如下图（左）所示，假设矩阵 $\mathbf{X}$ 中最大值为 $x_{23}$ ，我们将为锚框 $A_2$ 分配真实边界框 $B_3$ 。然后，丢弃矩阵中第2行和第3列的所有元素，找出剩余阴影部分的最大元素 $x_{71}$ ，为锚框 $A_7$ 分配真实边界框 $B_1$ 。接着如下图（中）所示，丢弃矩阵中第7行和第1列的所有元素，找出剩余阴影部分的最大元素 $x_{54}$ ，为锚框 $A_5$ 分配真实边界框 $B_4$ 。最后如下图（右）所示，丢弃矩阵中第5行和第4列的所有元素，找出剩余阴影部分的最大元素 $x_{92}$ ，为锚框 $A_9$ 分配真实边界框 $B_2$ 。之后，我们只需遍历除去 $A_2, A_5, A_7, A_9$ 的剩余锚框，并根据阈值判断是否为剩余锚框分配真实边界框。



现在我们可以标注锚框的类别和偏移量了。如果一个锚框 $A$ 被分配了真实边界框 $B$ ，将锚框 $A$ 的类别设为 $B$ 的类别，并根据 $B$ 和 $A$ 的中心坐标的相对位置以及两个框的相对大小为锚框 $A$ 标注偏移量。由于数据集中各个框的位置和大小各异，因此这些相对位置和相对大小通常需要一些特殊变换，才能使偏移量的分布更均匀从而更容易拟合。设锚框 $A$ 及其被分配的真实边界框 $B$ 的中心坐标分别为 $(x_a, y_a)$ 和 $(x_b, y_b)$ ， $A$ 和 $B$ 的宽分别为 $w_a$ 和 $w_b$ ，高分别为 $h_a$ 和 $h_b$ ，一个常用的技巧是将 $A$ 的偏移量标注为

$$\left( \frac{\frac{x_b - x_a}{w_a} - \mu_x}{\sigma_x}, \frac{\frac{y_b - y_a}{h_a} - \mu_y}{\sigma_y}, \frac{\log \frac{w_b}{w_a} - \mu_w}{\sigma_w}, \frac{\log \frac{h_b}{h_a} - \mu_h}{\sigma_h} \right)$$

其中常数的默认值为 $\mu_x = \mu_y = \mu_w = \mu_h = 0, \sigma_x = \sigma_y = 0.1, \sigma_w = \sigma_h = 0.2$ 。如果一个锚框没有被分配真实边界框，我们只需将该锚框的类别设为背景。**类别为背景的锚框通常被称为负类锚框，其余则被称为正类锚框。**

```

1 # 为与真实边框最相似的锚框分配label
2 def match_anchor_to_bbox(ground_truth, anchors,
3     device, iou_threshold=0.5):
4     num_anchors = anchors.shape[0]
5     num_gt_boxes = ground_truth.shape[0]
6     # 对于第i个锚框与第j个真实边框
7     # 我们用矩阵第i行, 第j列的元素表示其二者的IoU
8     jaccard = compute_jaccard(anchors, ground_truth)
9     # 初始化一个张量用于存储每一个锚框对应的真实边框
10    anchors_bbox_map = torch.full(
11        (num_anchors,), -1, dtype=torch.long, device=device)

```

```

12     # 根据阈值为锚框分配真实边框
13     max_ious, indices = torch.max(jaccard, dim=1)
14     anc_i = torch.nonzero(max_ious>=0.5).reshape(-1)
15     box_j = indices[max_ious>=0.5]
16     anchors_bbox_map[anc_i] = box_j
17     # 为每一个锚框匹配最大的iou对应的真实边框
18     anc_i = torch.argmax(jaccard, dim=0)
19     box_j = torch.arange(num_gt_boxes, device=device)
20     anchors_bbox_map[anc_i] = box_j
21     return anchors_bbox_map
22 # 将(左上角坐标, 右下角坐标)转换为(中心点坐标, 宽, 高)
23 def box_corner_to_center(boxes):
24     x1, y1, x2, y2 = boxes[:, 0], boxes[:, 1], boxes[:, 2], boxes[:, 3]
25     cx = (x1 + x2) / 2
26     cy = (y1 + y2) / 2
27     w = x2 - x1
28     h = y2 - y1
29     boxes = np.stack((cx, cy, w, h), axis=1)
30     return boxes
31 # 为锚框标注偏移量
32 def offset_boxes(anchors, assigned_bb, eps=1e-6):
33     c_anc = box_corner_to_center(anchors)
34     c_assigned_bb = box_corner_to_center(assigned_bb)
35     offset_xy = 10 * (c_assigned_bb[:, :2] - c_anc[:, :2]) / c_anc[:, 2:]
36     offset_wh = 5 * np.log(eps + c_assigned_bb[:, 2:] / c_anc[:, 2:])
37     offset = np.concatenate([offset_xy, offset_wh], axis=1)
38     return offset

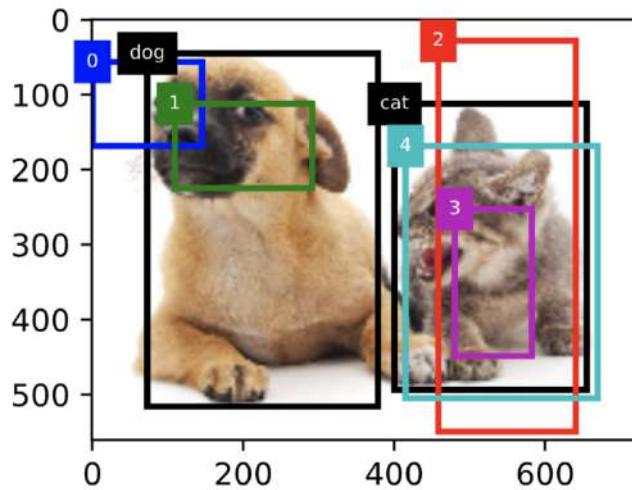
```

下面演示一个具体的例子。我们为读取的图像中的猫和狗定义真实边界框，其中第一个元素为类别（0为狗，1为猫），剩余4个元素分别为左上角的 $x$ 和 $y$ 轴坐标以及右下角的 $x$ 和 $y$ 轴坐标（值域在0到1之间）。这里通过左上角和右下角的坐标构造了5个需要标注的锚框，分别记为 $A_0, \dots, A_4$ （程序中索引从0开始）。先画出这些锚框与真实边界框在图像中的位置。

```

1 In [6]:
2     bbox_scale = torch.tensor(
3         (w, h, w, h), dtype=torch.float32)
4     # 第一列表示类别
5     ground_truth = torch.tensor([
6         [0, 0.1, 0.08, 0.52, 0.92],
7         [1, 0.55, 0.2, 0.9, 0.88]
8     ])
9     anchors = torch.tensor([
10        [0, 0.1, 0.2, 0.3],
11        [0.15, 0.2, 0.4, 0.4],
12        [0.63, 0.05, 0.88, 0.98],
13        [0.66, 0.45, 0.8, 0.8],
14        [0.57, 0.3, 0.92, 0.9]
15    ])
16     fig = d2l.plt.imshow(img)
17     show_bboxes(fig.axes, ground_truth[:, 1:]*bbox_scale,
18                 ['dog', 'cat'], 'k')
19     show_bboxes(fig.axes, anchors*bbox_scale,
20                 ['0', '1', '2', '3', '4']);

```



下面实现 `multibox_target` 函数来为锚框标注类别和偏移量。该函数将背景类别设为0，并令从零开始的目标类别的整数索引自加1（1为狗，2为猫）。

```

1 In [7]:
2 def multibox_target(anchors, labels):
3     batch_size, anchors = labels.shape[0], anchors.squeeze(0)
4     batch_offset, batch_mask, batch_class_labels = [], [], []
5     device, num_anchors = anchors.device, anchors.shape[0]
6     for i in range(batch_size):
7         # 取当前边框的坐标
8         label = labels[i, :, :]
9         # 第一列为标签
10        anchors_bbox_map = match_anchor_to_bbox(
11            label[:, 1:], anchors, device)
12        # 只保留匹配上的，未匹配的为-1被过滤了
13        bbox_mask = (
14            (anchors_bbox_map >= 0).float().unsqueeze(-1)
15            .repeat(1, 4))
16        # 初始化锚框标签
17        class_labels = torch.zeros(
18            num_anchors, dtype=torch.long, device=device)
19        # 初始化偏移量
20        assigned_bb = torch.zeros(
21            (num_anchors, 4), dtype=torch.float32, device=device)
22        # 为锚框标记类别
23        indices_true = torch.nonzero(anchors_bbox_map >= 0)
24        bb_idx = anchors_bbox_map[indices_true]
25        # 加1为了使原来的背景变为标签0
26        class_labels[indices_true] = label[bb_idx, 0].long() + 1
27        assigned_bb[indices_true] = label[bb_idx, 1:]
28        # 计算偏移量，没有配上就是4个0
29        offset = torch.from_numpy(
30            offset_boxes(anchors, assigned_bb)) * bbox_mask
31        batch_offset.append(offset.reshape(-1))
32        batch_mask.append(bbox_mask.reshape(-1))
33        batch_class_labels.append(class_labels)
34        bbox_offset = torch.stack(batch_offset)
35        bbox_mask = torch.stack(batch_mask)
36        class_labels = torch.stack(batch_class_labels)
37        return (bbox_offset, bbox_mask, class_labels)

```

返回的结果里有3项，均为 `Tensor`。第三项表示为锚框标注的类别。我们通过 `unsqueeze` 函数为锚框和真实边界框添加样本维。

```
1 In [8]:  
2     labels = multibox_target(anchors.unsqueeze(dim=0),  
3                               ground_truth.unsqueeze(dim=0))  
4     labels[2]  
5 Out [8]:  
6     tensor([[0, 1, 2, 0, 2]])
```

我们根据锚框与真实边界框在图像中的位置来分析这些标注的类别。首先，在所有的“锚框—真实边界框”的配对中，锚框  $A_4$  与猫的真实边界框的交并比最大，因此锚框  $A_4$  的类别标注为猫。不考虑锚框  $A_4$  或猫的真实边界框，在剩余的“锚框—真实边界框”的配对中，最大交并比的配对为锚框  $A_1$  和狗的真实边界框，因此锚框  $A_1$  的类别标注为狗。接下来遍历未标注的剩余3个锚框：与锚框  $A_0$  交并比最大的真实边界框的类别为狗，但交并比小于阈值（默认为0.5），因此类别标注为背景；与锚框  $A_2$  交并比最大的真实边界框的类别为猫，且交并比大于阈值，因此类别标注为猫；与锚框  $A_3$  交并比最大的真实边界框的类别为猫，但交并比小于阈值，因此类别标注为背景。

返回值的第二项为掩码（mask）变量，形状为（批量大小, 锚框个数的四倍）。掩码变量中的元素与每个锚框的4个偏移量一一对应。由于我们不关心对背景的检测，有关负类的偏移量不应影响目标函数。通过按元素乘法，掩码变量中的0可以在计算目标函数之前过滤掉负类的偏移量。

```
1 In [9]:  
2     labels[1]  
3 Out [9]:  
4     tensor([[0., 0., 0., 0., 1., 1., 1., 1., 1., 1.,  
5                 1., 0., 0., 0., 1., 1., 1., 1.]])
```

返回的第一项是为每个锚框标注的四个偏移量，其中负类锚框的偏移量标注为0。

```
1 In [10]:  
2     labels[0]  
3 Out [10]:  
4     tensor([[-0.0000e+00, -0.0000e+00, -0.0000e+00, -0.0000e+00,  1.4000e+00,  
5               1.0000e+01,  2.5940e+00,  7.1754e+00, -1.2000e+00,  2.6882e-01,  
6               1.6824e+00, -1.5655e+00, -0.0000e+00, -0.0000e+00, -0.0000e+00,  
7               -0.0000e+00, -5.7143e-01, -1.0000e+00,  4.1723e-06,  6.2582e-01  
8             ]])
```

## 8.4.4 输出预测边界框

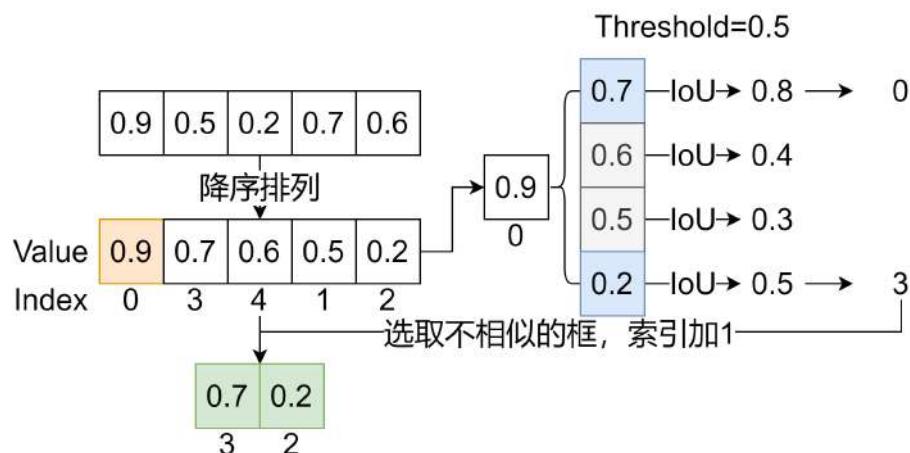
在模型预测阶段，我们先为图像生成多个锚框，并为这些锚框——预测类别和偏移量。随后，我们根据锚框及其预测偏移量得到预测边界框。当锚框数量较多时，同一个目标上可能会输出较多相似的预测边界框。为了使结果更加简洁，**我们可以移除相似的预测边界框**。常用的方法叫作**非极大值抑制**（non-maximum suppression, NMS）。

我们来描述一下非极大值抑制的工作原理。对于一个预测边界框  $B$ ，模型会计算各个类别的预测概率。设其中最大的预测概率为  $p$ ，该概率所对应的类别即  $B$  的预测类别。我们也将  $p$  称为预测边界框  $B$  的置信度。在同一图像上，我们将预测类别非背景的预测边界框按置信度从高到低排序，得到列表  $L$ 。**从  $L$  中选取置信度最高的预测边界框  $B_1$  作为基准，将所有与  $B_1$  的交并比大于某阈值的非基准预测边界框从  $L$  中移除**。这里的阈值是预先设定的超参数。此时， $L$  保留了置信度最高的预测边界框并移除了与其相似的其他预测边界框。接下来，从  $L$  中选取置信度第二高的预测边界框  $B_2$  作为基准，将所有与  $B_2$  的交并比大于某阈值的非基准预测边界框从  $L$  中移除。重复这一过程，直到  $L$  中所有的预测边界框都曾作为基准。此时  $L$  中任意一对预测边界框的交并比都小于阈值。最终，输出列表  $L$  中的所有预测边界框。

```

1 # 将(中心点坐标, 宽, 高)转换为(左上角坐标, 右下角坐标)
2 def box_center_to_corner(boxes):
3     cx, cy, w, h = boxes[:, 0], boxes[:, 1], boxes[:, 2], boxes[:, 3]
4     x1 = cx - 0.5 * w
5     y1 = cy - 0.5 * h
6     x2 = cx + 0.5 * w
7     y2 = cy + 0.5 * h
8     boxes = np.stack((x1, y1, x2, y2), axis=1)
9     return boxes
10 # 根据锚框和偏移量反推预测的边框
11 def offset_inverse(anchors, offset_preds):
12     c_anc = torch.from_numpy(box_center_to_corner(anchors))
13     c_pred_bb_xy = (offset_preds[:, :2] * c_anc[:, 2:] / 10) + c_anc[:, :2]
14     c_pred_bb_wh = np.exp(offset_preds[:, 2:] / 5) * c_anc[:, 2:]
15     c_pred_bb = np.concatenate([c_pred_bb_xy, c_pred_bb_wh], axis=1)
16     predicted_bb = box_center_to_corner(c_pred_bb)
17     return predicted_bb
18 # 利用非极大值抑制移除相似的预测边界框
19 def nms(boxes, scores, iou_threshold):
20     boxes = torch.tensor(boxes)
21     # 降序排列, 并返回索引
22     B = torch.argsort(scores, dim=-1, descending=True)
23     # 保存保留框的索引
24     keep = []
25     while B.numel() > 0:
26         # 最大值的索引
27         i = B[0]
28         keep.append(i)
29         # 终止条件, 1个框无法计算交并比
30         if B.numel() == 1:
31             break
32         # 当前框与剩余框的交并比
33         iou = compute_jaccard(boxes[i, :].reshape(-1, 4),
34                               boxes[B[1:], :].reshape(-1, 4)).reshape(-1)
35         # 筛选满足条件的框, 返回其索引
36         inds = torch.nonzero(iou <= iou_threshold).reshape(-1)
37         # 加1是因为当前框已经计算过, 所以整体索引加1才是剩余框的索引
38         B = B[inds + 1]
39     return torch.tensor(keep, device=boxes.device)

```



下面来看一个具体的例子。先构造4个锚框。简单起见，我们假设预测偏移量全是0：预测边界框即锚框。最后，我们构造每个类别的预测概率。

```

1 In [11]:
2     anchors = torch.tensor([
3         [0.1, 0.08, 0.52, 0.92],
4         [0.08, 0.2, 0.56, 0.95],
5         [0.15, 0.3, 0.62, 0.91],
6         [0.55, 0.2, 0.9, 0.88]
7     ])
8     offset_preds = torch.tensor([0.0] * (4 * len(anchors)))
9     cls_probs = torch.tensor([
10        [0., 0., 0.],           # 背景的预测概率
11        [0.9, 0.8, 0.7, 0.1],  # 狗的预测概率
12        [0.1, 0.2, 0.3, 0.9]   # 猫的预测概率
13    ])

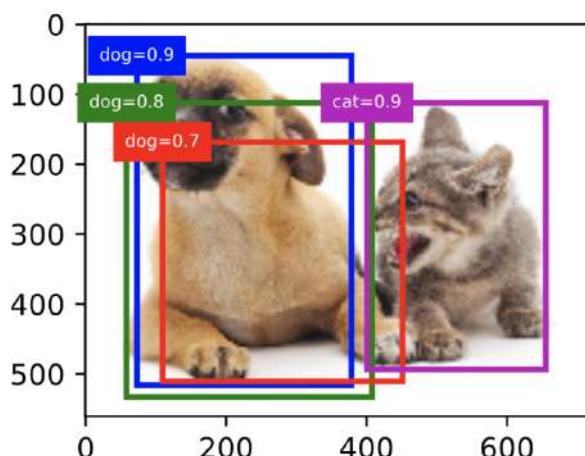
```

在图像上打印预测边界框和它们的置信度。

```

1 In [12]:
2     fig = d2l=plt.imshow(img)
3     show_bboxes(fig.axes, anchors*bbox_scale,
4                  ['dog=0.9', 'dog=0.8', 'dog=0.7', 'cat=0.9'])

```



下面我们实现 `multibox_detection` 函数来执行非极大值抑制。

```

1 def multibox_detection(cls_probs, offset_preds, anchors,
2                         nms_threshold=0.5, score_threshold=0.0099):
3     device, batch_size = cls_probs.device, cls_probs.shape[0]
4     anchors = anchors.squeeze(0)
5     num_classes, num_anchors = cls_probs.shape[1], cls_probs.shape[2]
6     out = []
7     for i in range(batch_size):
8         cls_prob, offset_pred = cls_probs[i], offset_preds[i].reshape(-1, 4)
9         # 每一列最大值及其索引，且过滤掉背景概率
10        conf, class_id = torch.max(cls_prob[1:], 0)
11        predicted_bb = offset_inverse(anchors, offset_pred)
12        keep = nms(predicted_bb, conf, 0.5)
13        # 将所有未保留的框设为背景类
14        all_idx = torch.arange(num_anchors, dtype=torch.long, device=device)
15        combined = torch.cat((keep, all_idx))
16        # 不重复元素（类别），不重复元素的个数（类别数）
17        uniques, counts = combined.unique(return_counts=True)
18        # 剔除掉只出现过1次的索引
19        non_keep = uniques[counts==1]

```

```

18     all_id_sorted = torch.cat((keep, non_keep))
19     # 设为背景
20     class_id[non_keep] = -1
21     class_id = class_id[all_id_sorted]
22     predicted_bb = torch.tensor(predicted_bb, device=device)
23     pred_info = torch.cat(
24         (class_id.unsqueeze(1).float(),
25          conf[all_id_sorted].unsqueeze(1),
26          predicted_bb[all_id_sorted]), dim=1
27     )
27     out.append(pred_info)
28
29 return torch.stack(out)

```

然后我们运行 `multibox_detection` 函数并设阈值为 0.5。这里为输入都增加了样本维。我们看到，返回的结果的形状为（批量大小，锚框个数, 6）。其中每一行的6个元素代表同一个预测边界框的输出信息。第一个元素是索引从0开始计数的预测类别（0为狗，1为猫），其中-1表示背景或在非极大值抑制中被移除。第二个元素是预测边界框的置信度。剩余的4个元素分别是预测边界框左上角的 $x$ 和 $y$ 轴坐标以及右下角的 $x$ 和 $y$ 轴坐标（值域在0到1之间）。

```

1 In [13]:
2     output = multibox_detection(
3         cls_probs.unsqueeze(dim=0),
4         offset_preds.unsqueeze(dim=0),
5         anchors.unsqueeze(dim=0),
6         nms_threshold=0.5
7     )
8     output
9 Out [13]:
10    tensor([[[ 0.0000,  0.9000,  0.1000,  0.0800,  0.5200,  0.9200],
11              [ 1.0000,  0.9000,  0.5500,  0.2000,  0.9000,  0.8800],
12              [-1.0000,  0.8000,  0.0800,  0.2000,  0.5600,  0.9500],
13              [-1.0000,  0.7000,  0.1500,  0.3000,  0.6200,  0.9100]]])

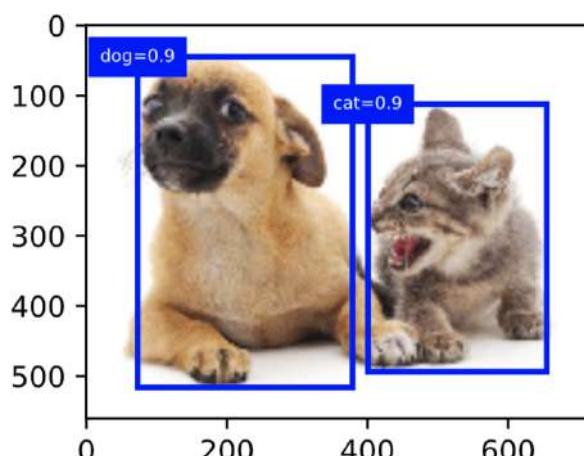
```

我们移除掉类别为-1的预测边界框，并可视化非极大值抑制保留的结果。

```

1 In [14]:
2     fig = d2l.plt.imshow(img)
3     for i in output[0].detach().cpu().numpy():
4         if i[0] == -1:
5             continue
6         label = ('dog=', 'cat=')[int(i[0])] + str(i[1])
7         show_bboxes(fig.axes, [torch.tensor(i[2:]) * bbox_scale], label)

```



实践中，我们可以在执行非极大值抑制前将置信度较低的预测边界框移除，从而减小非极大值抑制的计算量。我们还可以筛选非极大值抑制的输出，例如，只保留其中置信度较高的结果作为最终输出。

## 小结：

- 以每个像素为中心，生成多个大小和宽高比不同的锚框。
  - 交并比是两个边界框相交面积与相并面积之比。
  - 在训练集中，为每个锚框标注两类标签：一是锚框所含目标的类别；二是真实边界框相对锚框的偏移量。
  - 预测时，可以使用非极大值抑制来移除相似的预测边界框，从而令结果简洁。

## 8.5 多尺度目标检测

在8.4节（锚框）中，我们在实验中以输入图像的每个像素为中心生成多个锚框。这些锚框是对输入图像不同区域的采样。然而，如果以图像每个像素为中心都生成锚框，很容易生成过多锚框而造成计算量过大。举个例子，假设输入图像的高和宽分别为561像素和728像素，如果以每个像素为中心生成5个不同形状的锚框，那么一张图像上则需要标注并预测200多万个锚框 ( $561 \times 728 \times 5$ )。

减少锚框个数并不难。一种简单的方法是在输入图像中**均匀采样**一小部分像素，并以采样的像素为中心生成锚框。此外，在不同尺度下，我们可以生成不同数量和不同大小的锚框。值得注意的是，较小目标比较大目标在图像上出现位置的可能性更多。举个简单的例子：形状为 $1 \times 1$ 、 $1 \times 2$ 和 $2 \times 2$ 的目标在形状为 $2 \times 2$ 的图像上可能出现的位置分别有4、2和1种。因此，当使用较小锚框来检测较小目标时，我们可以采样较多的区域；而当使用较大锚框来检测较大目标时，我们可以采样较少的区域。

为了演示如何多尺度生成锚框，我们先读取一张图像。它的高和宽分别为561像素和728像素。

```
1 In [1]:  
2     from PIL import Image  
3     img = Image.open('data/catdog.jpg')  
4     w, h = img.size  
5     print(w, h)  
6 Out [1]:  
7     728 561
```

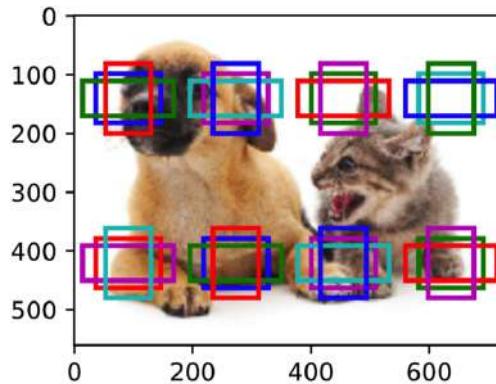
我们在4.1节（二维卷积层）中将卷积神经网络的二维数组输出称为特征图。我们可以通过定义特征图的形状来确定任一图像上均匀采样的锚框中心。

下面定义 `display_anchors` 函数。我们在特征图 `fmap` 上以每个单元（像素）为中心生成锚框 `anchors`。由于锚框 `anchors` 中  $x$  和  $y$  轴的坐标值分别已除以特征图 `fmap` 的宽和高，这些值域在 0 和 1 之间的值表达了锚框在特征图中的相对位置。由于锚框 `anchors` 的中心遍布特征图 `fmap` 上的所有单元，`anchors` 的中心在任一图像的空间相对位置一定是均匀分布的。具体来说，当特征图的宽和高分别设为 `fmap_w` 和 `fmap_h` 时，该函数将在任一图像上均匀采样 `fmap_h` 行 `fmap_w` 列个像素，并分别以它们为中心生成大小为 `s`（假设列表 `s` 长度为 1）的不同宽高比（`ratios`）的锚框。

```
13 |     bbox_scale = torch.tensor([[w, h, w, h]], dtype=torch.float32)
14 |     d2l.show_bboxes(d2l.plt.imshow(img).axes, anchors[0]*bbox_scale)
```

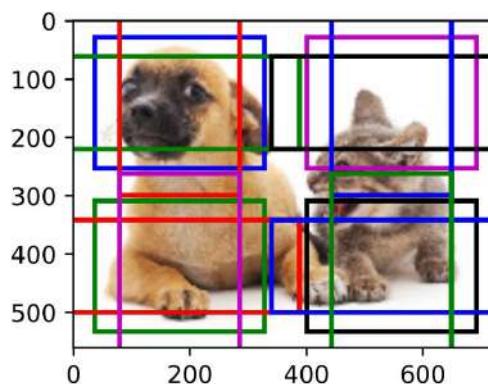
我们先关注小目标的检测。为了在显示时更容易分辨，这里令不同中心的锚框不重合：设锚框大小为0.15，特征图的高和宽分别为2和4。可以看出，图像上2行4列的锚框中心分布均匀。

```
1 | In [3]:
2 |     display_anchors(fmap_w=4, fmap_h=2, s=[0.15])
```



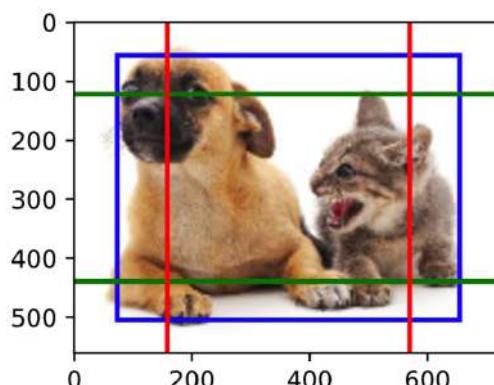
我们将特征图的高和宽分别减半，并用更大的锚框检测更大的目标。当锚框大小设0.4时，有些锚框的区域有重合。

```
1 | In [4]:
2 |     display_anchors(fmap_w=2, fmap_h=2, s=[0.4])
```



最后，我们将特征图的宽进一步减半至1，并将锚框大小增至0.8。此时锚框中心即图像中心。

```
1 | In [5]:
2 |     display_anchors(fmap_w=1, fmap_h=1, s=[0.8])
```



既然我们已在多个尺度上生成了不同大小的锚框，相应地，我们需要在不同尺度下检测不同大小的目标。下面我们将介绍一种基于卷积神经网络的方法。

在某个尺度下，假设我们依据 $c_i$ 张形状为 $h \times w$ 的特征图生成 $h \times w$ 组不同中心的锚框，且每组的锚框个数为 $a$ 。例如，在刚才实验的第一个尺度下，我们依据10（通道数）张形状为 $4 \times 2$ 的特征图生成了8组不同中心的锚框，且每组含3个锚框。接下来，依据真实边界框的类别和位置，每个锚框将被标注类别和偏移量。在当前的尺度下，目标检测模型需要根据输入图像预测 $h \times w$ 组不同中心的锚框的类别和偏移量。

假设这里的 $c_i$ 张特征图为卷积神经网络根据输入图像做前向计算所得的中间输出。既然每张特征图上都有 $h \times w$ 个不同的空间位置，那么相同空间位置可以看作含有 $c_i$ 个单元。根据4.1节（二维卷积层）中感受野的定义，**特征图在相同空间位置的 $c_i$ 个单元在输入图像上的感受野相同，并表征了同一感受野内的输入图像信息**。因此，我们可以将特征图在相同空间位置的 $c_i$ 个单元变换为以该位置为中心生成的 $a$ 个锚框的类别和偏移量。不难发现，本质上，**我们用输入图像在某个感受野区域内的信息来预测输入图像上与该区域位置相近的锚框的类别和偏移量**。

当不同层的特征图在输入图像上分别拥有不同大小的感受野时，它们将分别用来检测不同大小的目标。例如，我们可以通过设计网络，令较接近输出层的特征图中每个单元拥有更广阔的感受野，从而检测输入图像中更大尺寸的目标。

我们将在8.7节（单发多框检测（SSD））中具体实现一个多尺度目标检测的模型。

### 小结：

- 可以在多个尺度下生成不同数量和不同大小的锚框，从而在多个尺度下检测不同大小的目标。
- 特征图的形状能确定任一图像上均匀采样的锚框中心。
- 用输入图像在某个感受野区域内的信息来预测输入图像上与该区域相近的锚框的类别和偏移量。

## 8.6 目标检测数据集（皮卡丘）

在目标检测领域并没有类似MNIST或Fashion-MNIST那样的小数据集。为了快速测试模型，我们合成了一个小的数据集。我们首先使用一个开源的皮卡丘3D模型生成了1000张不同角度和大小的皮卡丘图像。然后我们收集了一系列背景图像，并在每张图的随机位置放置一张随机的皮卡丘图像。其中图片格式为`.png`，并用`json`文件存放对应的label信息。`pikachu`文件夹下的结构应如下所示。

```
1 --pikachu
2   --train
3     --images
4       --1.png
5       ...
6     --label.json
7   --val
8     --images
9       --1.png
10      ...
11     --label.json
```

### 8.6.1 读取数据集

先导入相关库。

```
1 In [1]:  
2     import os  
3     import json  
4     import numpy as np  
5     import torch  
6     import torchvision  
7     from PIL import Image  
8     import sys  
9     data_dir = 'data/pikachu'
```

我们先定义一个数据集类 `PikachuDetDataset`，数据集每个样本包含 `label` 和 `image`，其中 `label` 是一个  $m \times 5$  的向量，即  $m$  个边界框，每个边界框由 `[class, x_min, y_min, x_max, y_max]` 表示，这里的皮卡丘数据集中每个图像只有一个边界框，因此  $m=1$ 。`image` 是一个所有元素都位于  $[0.0, 1.0]$  的浮点 `tensor`，代表图片数据。

```
1 # 对皮卡丘数据进行读取，并保存为指定格式  
2 class PikachuDetDataset(torch.utils.data.Dataset):  
3     def __init__(self, data_dir, part, image_size=(256, 256)):  
4         assert part in ['train', 'val']  
5         # 图片尺寸  
6         self.image_size = image_size  
7         # 图片路径  
8         self.image_dir = os.path.join(data_dir, part, 'images')  
9         # 图片标签  
10        with open(os.path.join(data_dir, part, 'label.json')) as f:  
11            self.label = json.load(f)  
12        self.transform = torchvision.transforms.Compose([  
13            # 将 PIL 图片转换成位于[0.0, 1.0]的floatTensor, shape (C x H x W)  
14            torchvision.transforms.ToTensor()  
15        ])  
16    def __len__(self):  
17        # 样本个数  
18        return len(self.label)  
19    def __getitem__(self, index):  
20        # 构建样本  
21        image_path = str(index+1)+'.png'  
22        cls = self.label[image_path]['class']  
23        label = np.array(  
24            [cls]+self.label[image_path]['loc'],  
25            dtype='float32')[None, :]  
26        PIL_img = Image.open(  
27            os.path.join(self.image_dir, image_path)  
28        ).convert('RGB').resize(self.image_size)  
29        img = self.transform(PIL_img)  
30        sample = {  
31            # shape: (1, 5) [class, xmin, ymin, xmax, ymax]  
32            'label': label,  
33            # shape: (3, *image_size)  
34            'image': img  
35        }  
36        return sample
```

然后我们通过创建 `DataLoader` 实例来读取目标检测数据集。我们将以随机顺序读取训练数据集，按序读取测试数据集。

```

1 In [2]:
2     def load_data_pikachu(batch_size, edge_size=256,
3         data_dir='data/pikachu'):
4         # edge_size: 输出图像的宽和高
5         image_size = (edge_size, edge_size)
6         train_dataset = PiKachuDetDataset(
7             data_dir, 'train', image_size)
8         val_dataset = PiKachuDetDataset(
9             data_dir, 'val', image_size)
10        train_iter = torch.utils.data.DataLoader(
11            train_dataset, batch_size=batch_size,
12            shuffle=True, num_workers=4)
13        val_iter = torch.utils.data.DataLoader(
14            val_dataset, batch_size=batch_size,
15            shuffle=False, num_workers=4)
16        return train_iter, val_iter

```

下面我们读取一个小批量并打印图像和标签的形状。图像的形状和之前实验中的一样，依然是(批量大小, 通道数, 高, 宽)。而标签的形状则是(批量大小,  $m$ , 5)，其中 $m$ 等于数据集中单个图像最多含有的边界框个数。小批量计算虽然高效，但它要求每张图像含有相同数量的边界框，以便放在同一个批量中。由于每张图像含有的边界框个数可能不同，我们为边界框个数小于 $m$ 的图像填充非法边界框，直到每张图像均含有 $m$ 个边界框。这样，我们就可以每次读取小批量的图像了。图像中每个边界框的标签由长度为5的数组表示。数组中第一个元素是边界框所含目标的类别。当值为-1时，该边界框为填充用的非法边界框。数组的剩余4个元素分别表示边界框左上角的 $x$ 和 $y$ 轴坐标以及右下角的 $x$ 和 $y$ 轴坐标（值域在0到1之间）。这里的皮卡丘数据集中每个图像只有一个边界框，因此 $m = 1$ 。

```

1 In [3]:
2     batch_size, edge_size = 32, 256
3     train_iter, _ = load_data_pikachu(
4         batch_size, edge_size, data_dir)
5     batch = iter(train_iter).next()
6     print(batch['image'].shape, batch['label'].shape)
7 Out [3]:
8     torch.Size([32, 3, 256, 256]) torch.Size([32, 1, 5])

```

## 8.6.2 图示数据

我们画出10张图像和它们中的边界框。可以看到，皮卡丘的角度、大小和位置在每张图像中都不一样。当然，这是一个简单的人工数据集。实际中的数据通常会复杂得多。

```

1 In [4]:
2     imgs = batch['image'][:10].permute(0, 2, 3, 1)
3     # 获取框的坐标
4     bboxes = batch['label'][:10, 0, 1:]
5     axes = d2l.show_images(imgs, 2, 5).flatten()
6     for ax, bb in zip(axes, bboxes):
7         d2l.show_bboxes(ax, [bb*edge_size], colors=['w'])

```



### 小结:

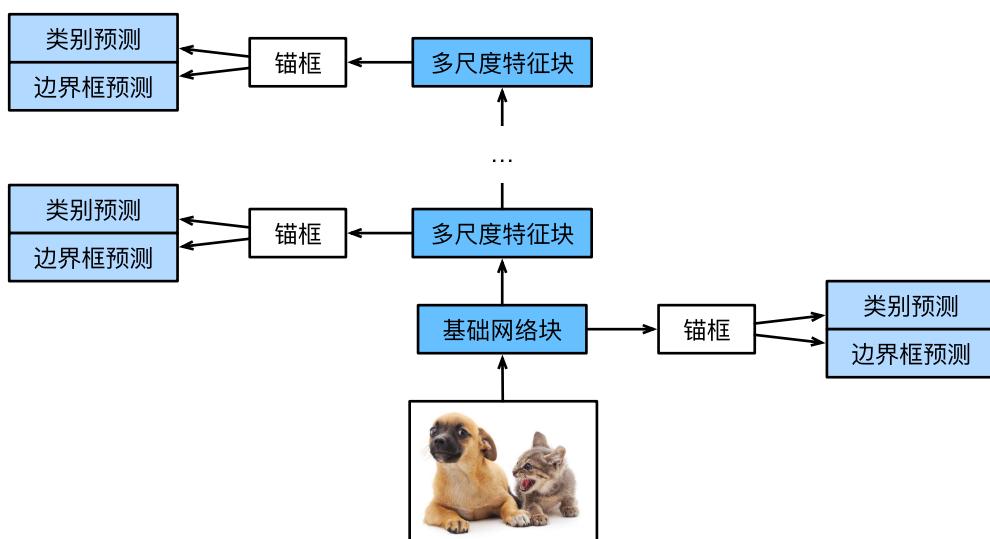
- 合成的皮卡丘数据集可用于测试目标检测模型。
- 目标检测的数据读取跟图像分类的类似。然而，在引入边界框后，标签形状和图像增广（如随机裁剪）发生了变化。

## 8.7 单发多框检测 (SSD)

我们在前几节分别介绍了边界框、锚框、多尺度目标检测和数据集，下面我们基于这些背景知识来构造一个目标检测模型：单发多框检测（single shot multibox detection, SSD）。它简单、快速，并得到了广泛应用。该模型的一些设计思想和实现细节常适用于其他目标检测模型。

### 8.7.1 定义模型

下图描述了单发多框检测模型的设计。它主要由一个基础网络块和若干个多尺度特征块串联而成。其中基础网络块用来从原始图像中抽取特征，因此一般会选择常用的深度卷积神经网络。单发多框检测论文中选用了在分类层之前截断的VGG，现在也常用ResNet替代。我们可以设计基础网络，使它输出的高和宽较大。这样一来，基于该特征图生成的锚框数量较多，可以用来检测尺寸较小的目标。接下来的每个多尺度特征块将上一层提供的特征图的高和宽缩小（如减半），并使特征图中每个单元在输入图像上的感受野变得更广阔。如此一来，下图中越靠近顶部的多尺度特征块输出的特征图越小，故而基于特征图生成的锚框也越少，加之特征图中每个单元感受野越大，因此更适合检测尺寸较大的目标。由于单发多框检测基于基础网络块和各个多尺度特征块生成不同数量和不同大小的锚框，并通过预测锚框的类别和偏移量（即预测边界框）检测不同大小的目标，因此单发多框检测是一个多尺度的目标检测模型。



接下来我们介绍如何实现图中的各个模块。我们先介绍如何实现类别预测和边界框预测。

## 1. 类别预测层

设目标的类别个数为 $q$ 。每个锚框的类别个数将是 $q + 1$ ，其中类别0表示锚框只包含背景。在某个尺度下，设特征图的高和宽分别为 $h$ 和 $w$ ，如果以其中每个单元为中心生成 $a$ 个锚框，那么我们需要对 $hwa$ 个锚框进行分类。如果使用全连接层作为输出，很容易导致模型参数过多。回忆4.8节介绍的**使用卷积层的通道来输出类别预测的方法**。单发多框检测采用同样的方法来降低模型复杂度。

具体来说，类别预测层使用一个保持输入高和宽的卷积层。这样一来，输出和输入在特征图宽和高上的空间坐标一一对应。考虑输出和输入同一空间坐标 $(x, y)$ ：输出特征图上 $(x, y)$ 坐标的通道里包含了以输入特征图 $(x, y)$ 坐标为中心生成的所有锚框的类别预测。因此输出通道数为 $a(q + 1)$ ，其中索引为 $i(q + 1) + j$  ( $0 \leq j \leq q$ ) 的通道代表了索引为 $i$ 的锚框有关类别索引为 $j$ 的预测。

下面我们定义一个这样的类别预测层：指定参数 $a$ 和 $q$ 后，它使用一个填充为1的 $3 \times 3$ 卷积层。该卷积层的输入和输出的高和宽保持不变。

```
1 In [1]:  
2     def cls_predictor(num_inputs, num_anchors, num_classes):  
3         return torch.nn.Conv2d(  
4             num_inputs, num_anchors*(num_classes+1),  
5             kernel_size=3, padding=1)
```

## 2. 边界框预测层

边界框预测层的设计与类别预测层的设计类似。唯一不同的是，这里需要为每个锚框预测4个偏移量，而不是 $q + 1$ 个类别。

```
1 In [2]:  
2     def bbox_predictor(num_inputs, num_anchors):  
3         return torch.nn.Conv2d(  
4             num_inputs, num_anchors*4,  
5             kernel_size=3, padding=1)
```

## 3. 连结多尺度的预测

前面提到，单发多框检测根据多个尺度下的特征图生成锚框并预测类别和偏移量。由于每个尺度上特征图的形状或以同一单元为中心生成的锚框个数都可能不同，因此不同尺度的预测输出形状可能不同。

在下面的例子中，我们对同一批量数据构造两个不同尺度下的特征图 $Y1$ 和 $Y2$ ，其中 $Y2$ 相对于 $Y1$ 来说高和宽分别减半。以类别预测为例，假设以 $Y1$ 和 $Y2$ 特征图中每个单元生成的锚框个数分别是5和3，当目标类别个数为10时，类别预测输出的通道数分别为 $5 \times (10 + 1) = 55$ 和 $3 \times (10 + 1) = 33$ 。预测输出的格式为（批量大小，通道数，高，宽）。可以看到，除了批量大小外，其他维度大小均不一样。我们需要将它们变形成统一的格式并将多尺度的预测连结，从而让后续计算更简单。

```
1 In [3]:  
2     def forward(x, block):  
3         return block(x)  
4     Y1 = forward(  
5         torch.zeros((2, 8, 20, 20)), cls_predictor(8, 5, 10))  
6     Y2 = forward(  
7         torch.zeros((2, 16, 10, 10)), cls_predictor(16, 3, 10))  
8     (Y1.shape, Y2.shape)  
9 Out [3]:  
10    (torch.Size([2, 55, 20, 20]), torch.Size([2, 33, 10, 10]))
```

**通道维包含中心相同的锚框的预测结果。**我们首先将通道维移到最后一维。因为不同尺度下批量大小仍保持不变，我们可以将预测结果转成二维的(批量大小, 高×宽×通道数)的格式，以方便之后在维度1上的连结。

```
1 In [4]:  
2     def flatten_pred(pred):  
3         return torch.flatten(pred.permute(0, 2, 3, 1), start_dim=1)  
4     def concat_preds(preds):  
5         return torch.cat([flatten_pred(p) for p in preds], dim=1)
```

这样一来，尽管`Y1`和`Y2`形状不同，我们仍然可以将这两个同一批量不同尺度的预测结果连结在一起。

```
1 In [5]:  
2     concat_preds([Y1, Y2]).shape  
3 Out [5]:  
4     torch.Size([2, 25300])
```

#### 4. 高和宽减半块

为了在多尺度检测目标，下面定义高和宽减半块`down_sample_blk`。**它串联了两个填充为1的 $3 \times 3$ 卷积层和步幅为2的 $2 \times 2$ 最大池化层。**我们知道，填充为1的 $3 \times 3$ 卷积层不改变特征图的形状，而后面的池化层则直接将特征图的高和宽减半。由于 $1 \times 2 + (3 - 1) + (3 - 1) = 6$ ，输出特征图中每个单元在输入特征图上的感受野形状为 $6 \times 6$ 。可以看出，高和宽减半块使输出特征图中每个单元的感受野变得更广阔。

```
1 In [6]:  
2     def down_sample_blk(in_channels, out_channels):  
3         blk = []  
4         for _ in range(2):  
5             blk.append(torch.nn.Conv2d(in_channels, out_channels,  
6                                     kernel_size=3, padding=1))  
7             blk.append(torch.nn.BatchNorm2d(out_channels))  
8             blk.append(torch.nn.ReLU())  
9             in_channels = out_channels  
10            blk.append(torch.nn.MaxPool2d(2))  
11            return torch.nn.Sequential(*blk)
```

测试高和宽减半块的前向计算。可以看到，它改变了输入的通道数，并将高和宽减半。

```
1 In [7]:  
2     forward(torch.zeros((2, 3, 20, 20)),  
3             down_sample_blk(3, 10)).shape  
4 Out [7]:  
5     torch.Size([2, 10, 10, 10])
```

#### 5. 基础网络块

基础网络块用来从原始图像中抽取特征。为了计算简洁，我们在这里构造一个小的基础网络。该网络串联3个高和宽减半块，并逐步将通道数翻倍。当输入的原始图像的形状为 $256 \times 256$ 时，基础网络块输出的特征图的形状为 $32 \times 32$ 。

```

1 In [8]:
2     def base_net():
3         blk = []
4         num_filters = [3, 16, 32, 64]
5         for i in range(len(num_filters)-1):
6             blk.append(
7                 down_sample_blk(num_filters[i],
8                                 num_filters[i+1]))
9         return torch.nn.Sequential(*blk)
10    forward(torch.zeros((2, 3, 256, 256)), base_net()).shape
11 Out [8]:
12    torch.size([2, 64, 32, 32])

```

## 6. 完整的模型

单发多框检测模型一共包含5个模块，**每个模块输出的特征图既用来生成锚框，又用来预测这些锚框的类别和偏移量。**第一模块为基础网络块，第二模块至第四模块为高和宽减半块，第五模块使用全局最大池化层将高和宽降到1。因此第二模块至第五模块均为上图中的多尺度特征块。

```

1 In [9]:
2     def get_blk(i):
3         if i==0:
4             blk = base_net()
5         elif i==1:
6             blk = down_sample_blk(64, 128)
7         elif i==4:
8             blk = torch.nn.AdaptiveMaxPool2d((1, 1))
9         else:
10            blk = down_sample_blk(128, 128)
11        return blk

```

接下来，我们定义每个模块如何进行前向计算。与之前介绍的卷积神经网络不同，这里不仅返回卷积计算输出的特征图`Y`，还返回根据`Y`生成的当前尺度的锚框，以及基于`Y`预测的锚框类别和偏移量。

```

1 In [10]:
2     def blk_forward(x, blk, size, ratio, cls_predictor,
3                      bbox_predictor):
4         Y = blk(x)
5         anchors = d2l.MultiBoxPrior(Y, sizes=size, ratios=ratio)
6         cls_preds = cls_predictor(Y)
7         bbox_preds = bbox_predictor(Y)
8         return (Y, anchors, cls_preds, bbox_preds)

```

我们提到，上图中较靠近顶部的多尺度特征块用来检测尺寸较大的目标，因此需要生成较大的锚框。我们在这里先将0.2到1.05之间均分5份，以确定不同尺度下锚框大小的较小值0.2、0.37、0.54等，再按 $\sqrt{0.2 \times 0.37} = 0.272$ 、 $\sqrt{0.37 \times 0.54} = 0.447$ 等来确定不同尺度下锚框大小的较大值。

```

1 In [11]:
2     sizes = [[0.2, 0.272], [0.37, 0.447], [0.54, 0.619],
3               [0.71, 0.79], [0.88, 0.961]]
4     ratios = [[1, 2, 0.5]] * 5
5     # 这里的计算参见8.4.1
6     num_anchors = len(sizes[0])+len(ratios[0])-1

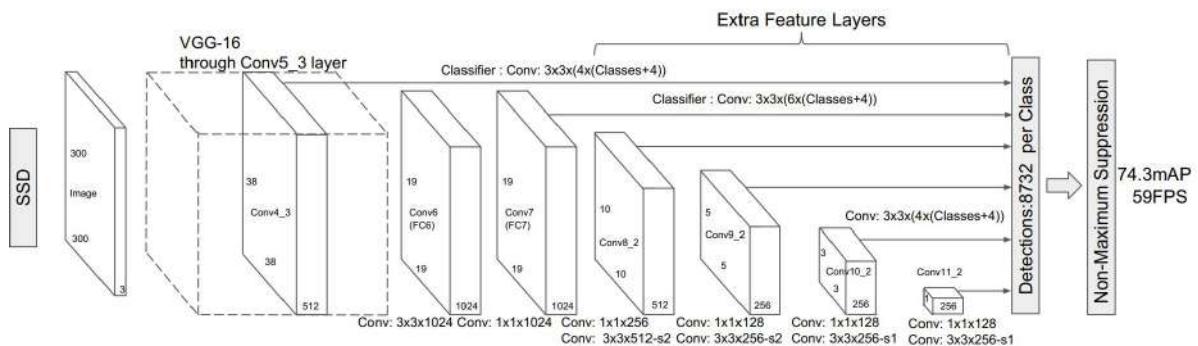
```

现在，我们就可以定义出完整的模型 TinySSD 了。

```
1 In [12]:  
2 class TinySSD(torch.nn.Module):  
3     def __init__(self, num_classes, **kwargs):  
4         super(TinySSD, self).__init__(**kwargs)  
5         self.num_classes = num_classes  
6         idx_to_in_channels = [64, 128, 128, 128, 128]  
7         for i in range(5):  
8             # The assignment statement is self.blk_i = get_blk(i)  
9             setattr(self, f'blk_{i}', get_blk(i))  
10            setattr(self, f'cls_{i}', cls_predictor(  
11                idx_to_in_channels[i], num_anchors, num_classes))  
12            setattr(self, f'bbox_{i}', bbox_predictor(  
13                idx_to_in_channels[i], num_anchors))  
14    def forward(self, X):  
15        anchors, cls_preds, bbox_preds = [None]*5, [None]*5, [None]*5  
16        for i in range(5):  
17            # getattr(self, 'blk_%d' % i) accesses self.blk_i  
18            x, anchors[i], cls_preds[i], bbox_preds[i] = blk_forward(  
19                X, getattr(self, f'blk_{i}'), sizes[i], ratios[i],  
20                getattr(self, f'cls_{i}'), getattr(self, f'bbox_{i}'))  
21        )  
22        # In the reshape function, 0 indicates that the batch size remains  
23        # unchanged  
24        anchors = torch.cat(anchors, dim=1)  
25        cls_preds = concat_preds(cls_preds)  
26        # batch_size, num_anchors, num_classes  
27        cls_preds = cls_preds.reshape(  
28            cls_preds.shape[0], -1, self.num_classes+1  
29        )  
30        bbox_preds = concat_preds(bbox_preds)  
31        return anchors, cls_preds, bbox_preds
```

我们创建单发多框检测模型实例并对一个高和宽均为256像素的小批量图像  $x$  做前向计算。我们在之前验证过，第一模块输出的特征图的形状为  $32 \times 32$ 。由于第二至第四模块为高和宽减半块、第五模块为全局池化层，并且以特征图每个单元为中心生成4个锚框，每个图像在5个尺度下生成的锚框总数为  $(32^2 + 16^2 + 8^2 + 4^2 + 1) \times 4 = 5444$ 。

```
1 In [13]:  
2 net = TinySSD(num_classes=1)  
3 x = torch.zeros((32, 3, 256, 256))  
4 anchors, cls_preds, bbox_preds = net(x)  
5 print('output anchors:', anchors.shape)  
6 print('output class preds:', cls_preds.shape)  
7 print('output bbox preds:', bbox_preds.shape)  
8 Out [13]:  
9 output anchors: torch.Size([1, 5444, 4])  
10 output class preds: torch.Size([32, 5444, 2])  
11 output bbox preds: torch.Size([32, 21776])
```



## 8.7.2 训练模型

下面我们描述如何一步步训练单发多框检测模型来进行目标检测。

### 1. 读取数据集和初始化

我们读取8.6节构造的皮卡丘数据集。

```
1 In [14]:  
2     batch_size = 32  
3     train_iter, _ = load_data_pikachu(batch_size)
```

在皮卡丘数据集中，目标的类别数为1。定义好模型以后，我们需要初始化模型参数并定义优化算法。

```
1 In [15]:  
2     device = torch.device(  
3         'cuda' if torch.cuda.is_available() else 'cpu')  
4     net = TinySSD(num_classes=1)  
5     trainer = torch.optim.SGD(  
6         net.parameters(), lr=0.2, weight_decay=5e-4)
```

### 2. 定义损失函数和评价函数

目标检测有两个损失：一是有关锚框类别的损失，我们可以重用之前图像分类问题里一直使用的交叉熵损失函数；二是有关正类锚框偏移量的损失。预测偏移量是一个回归问题，但这里不使用前面介绍过的平方损失，而使用 $L_1$ 范数损失，即预测值与真实值之间差的绝对值。掩码变量 bbox\_masks 令负类锚框和填充锚框不参与损失的计算。最后，我们将有关锚框类别和偏移量的损失相加得到模型的最终损失函数。

```
1 In [16]:  
2     cls_loss = torch.nn.CrossEntropyLoss(reduction='none')  
3     bbox_loss = torch.nn.L1Loss(reduction='none')  
4     def calc_loss(cls_preds, cls_labels, bbox_preds,  
5                   bbox_labels, bbox_masks):  
6         batch_size, num_classes = cls_preds.shape[0], cls_preds.shape[2]  
7         cls = cls_loss(cls_preds.reshape(-1, num_classes),  
8                         cls_labels.reshape(-1)  
9                         ).reshape(batch_size, -1).mean(dim=1)  
10        bbox = bbox_loss(bbox_preds*bbox_masks,  
11                           bbox_labels*bbox_masks).mean(dim=1)  
12        return cls+bbox
```

我们可以沿用准确率评价分类结果。因为使用了 $L_1$ 范数损失，我们用平均绝对误差评价边界框的预测结果。

```

1 In [17]:
2     def cls_eval(cls_preds, cls_labels):
3         # Because the category prediction results are placed in the final
4         # dimension, argmax must specify this dimension
5         return float(
6             (cls_preds.argmax(dim=-1).type(
7                 cls_labels.dtype
8             )==cls_labels).sum()
9         )
10    def bbox_eval(bbox_preds, bbox_labels, bbox_masks):
11        return float((torch.abs((bbox_labels-bbox_preds)*bbox_masks)).sum())

```

### 3. 训练模型

在训练模型时，我们需要在模型的前向计算过程中生成多尺度的锚框 anchors，并为每个锚框预测类别 `cls_preds` 和偏移量 `bbox_preds`。之后，我们根据标签信息 Y 为生成的每个锚框标注类别 `cls_labels` 和偏移量 `bbox_labels`。最后，我们根据类别和偏移量的预测和标注值计算损失函数。为了代码简洁，这里没有评价测试数据集。

```

1 In [18]:
2     import time
3     num_epochs = 30
4     net = net.to(device)
5     for epoch in range(num_epochs):
6         accuracy_sum = 0
7         mae_sum = 0
8         num_examples = 0
9         num_labels = 0
10        net.train()
11        for data in train_iter:
12            features = data['image']
13            target = data['label']
14            begin = time.time()
15            trainer.zero_grad()
16            X, Y = features.to(device), target.to(device)
17            anchors, cls_preds, bbox_preds = net(X)
18            bbox_labels, bbox_masks, cls_labels = multibox_target(
19                anchors, Y.cpu())
20            bbox_labels = bbox_labels.to(device)
21            bbox_masks = bbox_masks.to(device)
22            cls_labels = cls_labels.to(device)
23            l = calc_loss(cls_preds, cls_labels, bbox_preds,
24                           bbox_labels, bbox_masks)
25            l.mean().backward()
26            trainer.step()
27            accuracy_sum += cls_eval(cls_preds, cls_labels)
28            mae_sum += bbox_eval(bbox_preds, bbox_labels, bbox_masks)
29            num_examples += bbox_labels.numel()
30            num_labels += cls_labels.numel()
31            cls_err, bbox_mae = 1 - accuracy_sum / num_labels, mae_sum / num_examples
32            if (epoch+1) % 5 == 0:
33                print(f'epoch {epoch+1}, class err {cls_err:.2e}, \
34                      bbox mae {bbox_mae:.2e}, time {time.time()-begin:.1f} sec')
35 Out [18]:
36 epoch 5, class err 3.90e-04, bbox mae 5.44e-04, time 0.4 sec
37 epoch 10, class err 3.90e-04, bbox mae 5.21e-04, time 0.2 sec

```

```
38     epoch 15, class err 3.90e-04, bbox mae 4.99e-04, time 0.3 sec
39     epoch 20, class err 3.90e-04, bbox mae 4.74e-04, time 0.3 sec
40     epoch 25, class err 3.90e-04, bbox mae 4.45e-04, time 0.4 sec
41     epoch 30, class err 3.90e-04, bbox mae 4.11e-04, time 0.3 sec
```

### 8.7.3 预测目标

在预测阶段，我们希望能把图像里面所有我们感兴趣的目标检测出来。下面读取测试图像，将其变换尺寸，然后转成卷积层需要的四维格式。

```
1 In [19]:
2     from copy import deepcopy
3     img = Image.open('data/pikachu.jpg')
4     img = np.array(img, dtype=np.float32) / 255.0
5     X = torch.from_numpy(img)
6     # [batch_size, 3, image_height, image_width] ([B, C, H, W])
7     show_img = deepcopy(X*255.0)
8     X = X.permute(2, 0, 1).unsqueeze(0).float()
9     img = X.squeeze(0).permute(1, 2, 0).long()
10
11    show_img = show_img.permute(2, 0, 1).unsqueeze(0).float()
12    show_img = show_img.squeeze(0).permute(1, 2, 0).long()
```

我们通过 `multibox_detection` 函数根据锚框及其预测偏移量得到预测边界框，并通过非极大值抑制移除相似的预测边界框。

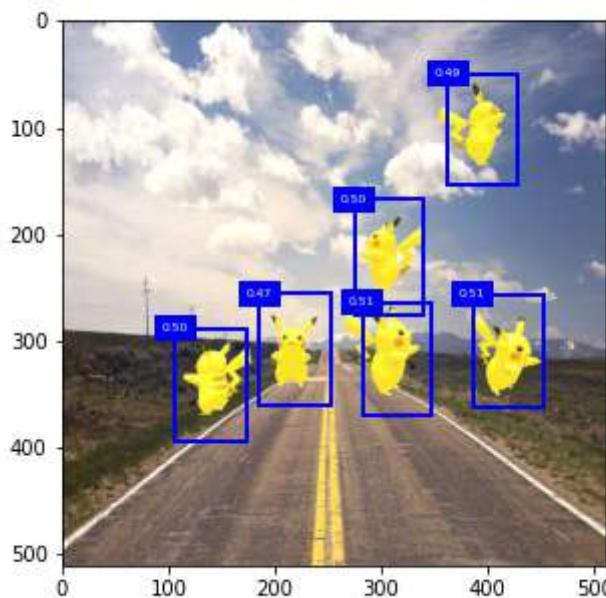
```
1 In [20]:
2     import torch.nn.functional as F
3     def predict(X):
4         net.eval()
5         anchors, cls_preds, bbox_preds = net(X.to(device))
6         cls_probs = F.softmax(cls_preds, dim=2).permute(0, 2, 1)
7         output = multibox_detection(cls_probs.detach().cpu(),
8                                     bbox_preds.detach().cpu(),
9                                     anchors.detach().cpu())
10        idx = [i for i, row in enumerate(output[0]) if row[0] != -1]
11        return output[0, idx]
12    output = predict(X)
```

最后，我们将置信度不低于0.46的边界框筛选为最终输出用以展示。

```

1 In [21]:
2     def display(img, output, threshold):
3         d2l.set_figsize((5, 5))
4         fig = d2l.plt.imshow(img)
5         for row in output:
6             score = float(row[1])
7             if score < threshold:
8                 continue
9             h, w = img.shape[:2]
10            bbox = [row[2:6] * torch.tensor(
11                (w, h, w, h), device=row.device).float()]
12            show_bboxes(fig.axes, bbox, '%.2f' % score)
13 display(show_img, output.cpu(), threshold=0.46)

```



### 小结:

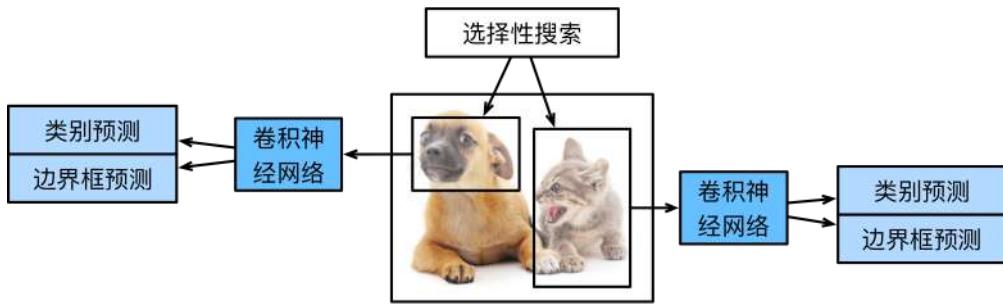
- 单发多框检测是一个多尺度的目标检测模型。该模型基于基础网络块和各个多尺度特征块生成不同数量和不同大小的锚框，并通过预测锚框的类别和偏移量检测不同大小的目标。
- 单发多框检测在训练中根据类别和偏移量的预测和标注值计算损失函数。

## 8.8 区域卷积神经网络 (R-CNN) 系列

区域卷积神经网络 (region-based CNN或regions with CNN features, R-CNN) 是将深度模型应用于目标检测的开创性工作之一。在本节中，我们将介绍R-CNN和它的一系列改进方法：快速的R-CNN (Fast R-CNN) 、更快的R-CNN (Faster R-CNN) 以及掩码R-CNN (Mask R-CNN) 。限于篇幅，这里只介绍这些模型的设计思路。

### 8.8.1 R-CNN

R-CNN首先对图像选取若干提议区域（如锚框也是一种选取方法）并标注它们的类别和边界框（如偏移量）。然后，用卷积神经网络对每个提议区域做前向计算抽取特征。之后，我们用每个提议区域的特征预测类别和边界框。下图描述了R-CNN模型。



具体来说，R-CNN主要由以下4步构成。

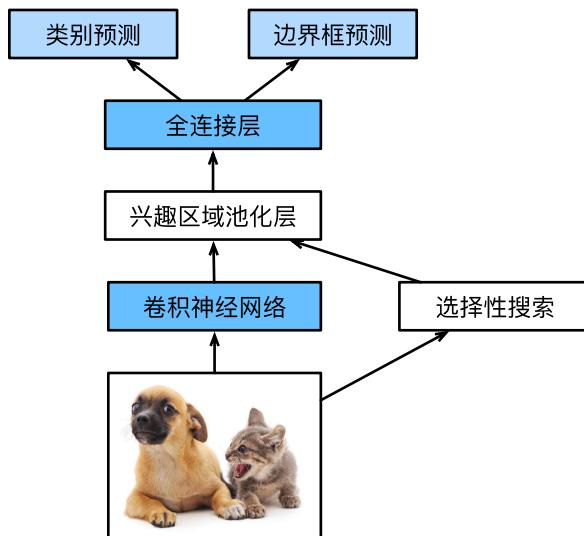
1. 对输入图像使用**选择性搜索** (selective search) 来选取多个高质量的提议区域。这些提议区域通常是在多个尺度下选取的，并具有不同的形状和大小。每个提议区域将被标注类别和真实边界框。
2. **选取一个预训练的卷积神经网络，并将其在输出层之前截断。** 将每个提议区域变形为网络需要的输入尺寸，并通过前向计算输出抽取的提议区域特征。
3. 将每个提议区域的特征连同其标注的类别作为一个样本，训练多个支持向量机对目标分类。其中每个支持向量机用来判断样本是否属于某一个类别。
4. 将每个提议区域的特征连同其标注的边界框作为一个样本，训练线性回归模型来预测真实边界框。

R-CNN虽然通过预训练的卷积神经网络有效抽取了图像特征，但它的主要缺点是速度慢。想象一下，我们可能从一张图像中选出上千个提议区域，对该图像做目标检测将导致上千次的卷积神经网络的前向计算。这个巨大的计算量令R-CNN难以在实际应用中被广泛采用。

## 8.8.2 Fast R-CNN

R-CNN的主要性能瓶颈在于需要对每个提议区域独立抽取特征。由于这些区域通常有大量重叠，独立的特征抽取会导致大量的重复计算。**Fast R-CNN对R-CNN的一个主要改进在于只对整个图像做卷积神经网络的前向计算。**

下图描述了Fast R-CNN模型。



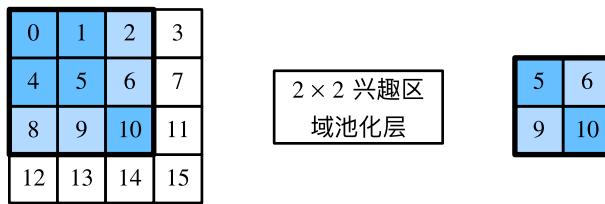
它的主要计算步骤如下。

1. 与R-CNN相比，Fast R-CNN用来提取特征的卷积神经网络的输入是整个图像，而不是各个提议区域。而且，这个网络通常会参与训练，即更新模型参数。设输入为一张图像，将卷积神经网络的输出的形状记为 $1 \times c \times h_1 \times w_1$ 。
2. 假设选择性搜索生成 $n$ 个提议区域。这些形状各异的提议区域在卷积神经网络的输出上分别标出形状各异的兴趣区域。**这些兴趣区域需要抽取出形状相同的特征（假设高和宽均分别指定为 $h_2$ 和 $w_2$ ）以便于连结后输出。**Fast R-CNN引入**兴趣区域池化** (Region of Interest Pooling, Roi池化) 层，将卷积神经网络的输出和提议区域作为输入，输出连结后的各个提议区域抽取的特征，形状为 $n \times c \times h_2 \times w_2$ 。
3. 通过全连接层将输出形状变换为 $n \times d$ ，其中超参数 $d$ 取决于模型设计。

4. 预测类别时，将全连接层的输出的形状再变换为 $n \times q$ 并使用softmax回归（ $q$ 为类别个数）。预测边界框时，将全连接层的输出的形状变换为 $n \times 4$ 。也就是说，我们为每个提议区域预测类别和边界框。

Fast R-CNN中提出的兴趣区域池化层跟我们在4.4节（池化层）中介绍过的池化层有所不同。在池化层中，我们通过设置池化窗口、填充和步幅来控制输出形状。而**兴趣区域池化层对每个区域的输出形状是可以直接指定的**，例如，指定每个区域输出的高和宽分别为 $h_2$ 和 $w_2$ 。假设某一兴趣区域窗口的高和宽分别为 $h$ 和 $w$ ，该窗口将被划分为形状为 $h_2 \times w_2$ 的子窗口网格，且每个子窗口的大小大约为 $(h/h_2) \times (w/w_2)$ 。任一子窗口的高和宽要取整，**其中的最大元素作为该子窗口的输出**。因此，兴趣区域池化层可从形状各异的兴趣区域中均抽取出形状相同的特征。

下图中，我们在 $4 \times 4$ 的输入上选取了左上角的 $3 \times 3$ 区域作为兴趣区域。对于该兴趣区域，我们通过 $2 \times 2$ 兴趣区域池化层得到一个 $2 \times 2$ 的输出。4个划分后的子窗口分别含有元素0、1、4、5（5最大），2、6（6最大），8、9（9最大），10。



我们使用 `roi_pool` 函数来演示兴趣区域池化层的计算。假设卷积神经网络抽取的特征  $x$  的高和宽均为4且只有单通道。

```
1 In [1]:  
2     import torch  
3     import torchvision  
4     x = torch.arange(16, dtype=torch.float).view(1, 1, 4, 4)  
5     x  
6 Out [1]:  
7     tensor([[[[ 0.,  1.,  2.,  3.],  
8             [ 4.,  5.,  6.,  7.],  
9             [ 8.,  9., 10., 11.],  
10            [12., 13., 14., 15.]]]])
```

假设图像的高和宽均为40像素。再假设选择性搜索在图像上生成了两个提议区域：每个区域由5个元素表示，分别为区域目标类别、左上角的 $x$ 和 $y$ 轴坐标以及右下角的 $x$ 和 $y$ 轴坐标。

```
1 In [2]:  
2     rois = torch.tensor([  
3         [0, 0, 0, 20, 20],  
4         [0, 0, 10, 30, 30]  
5     ], dtype=torch.float)
```

由于  $x$  的高和宽是图像的高和宽的 $1/10$ ，以上两个提议区域中的坐标先按 `spatial_scale` 自乘 0.1，然后在  $x$  上分别标出兴趣区域  $x[:, :, 0:3, 0:3]$  和  $x[:, :, 1:4, 0:4]$ 。最后对这两个兴趣区域分别划分子窗口网格并抽取高和宽为2的特征。

```

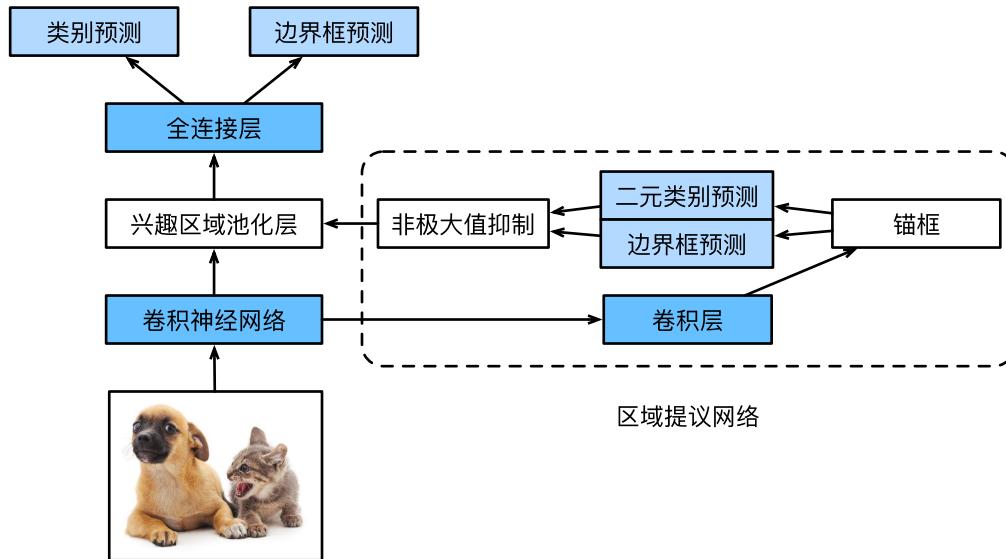
1 In [3]:
2     torchvision.ops.roi_pool(
3         x, rois, output_size=(2, 2), spatial_scale=0.1)
4 Out [3]:
5     tensor([[[[ 5.,  6.],
6                 [ 9., 10.]],
7
8                 [[ 9., 11.],
9                  [13., 15.]]]])

```

### 8.8.3 Faster R-CNN

Fast R-CNN通常需要在选择性搜索中生成较多的提议区域，以获得较精确的目标检测结果。Faster R-CNN提出将选择性搜索替换成**区域提议网络** (region proposal network)，从而减少提议区域的生成数量，并保证目标检测的精度。

下图描述了Faster R-CNN模型。与Fast R-CNN相比，只有生成提议区域的方法从选择性搜索变成了区域提议网络，而其他部分均保持不变。具体来说，区域提议网络的计算步骤如下。



1. 使用填充为1的 $3 \times 3$ 卷积层变换卷积神经网络的输出，并将输出通道数记为 $c$ 。这样，卷积神经网络为图像抽取的特征图中的每个单元均得到一个长度为 $c$ 的新特征。
2. 以特征图每个单元为中心，生成多个不同大小和宽高比的锚框并标注它们。
3. 用锚框中心单元长度为 $c$ 的特征分别预测该锚框的二元类别（含目标还是背景）和边界框。
4. 使用非极大值抑制，从预测类别为目标的预测边界框中移除相似的结果。最终输出的预测边界框即兴趣区域池化层所需要的提议区域。

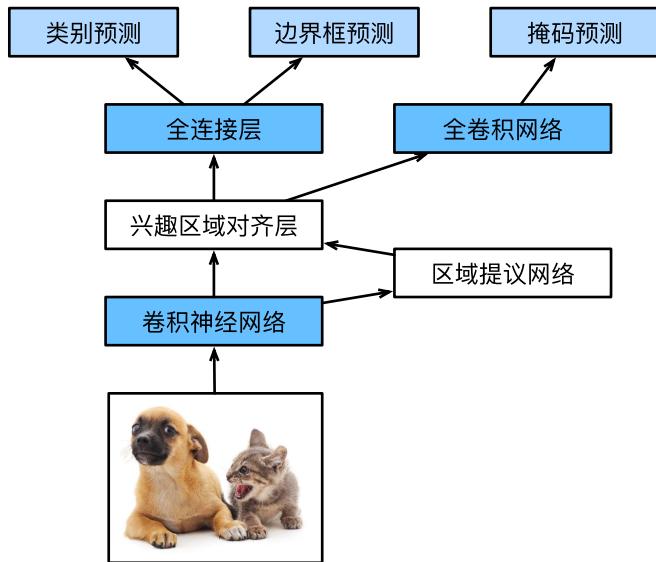
值得一提的是，区域提议网络作为Faster R-CNN的一部分，是和整个模型一起训练得到的。也就是说，Faster R-CNN的目标函数既包括目标检测中的类别和边界框预测，又包括区域提议网络中锚框的二元类别和边界框预测。最终，区域提议网络能够学习到如何生成高质量的提议区域，从而在减少提议区域数量的情况下也能保证目标检测的精度。

### 8.8.4 Mask R-CNN

如果训练数据还标注了每个目标在图像上的像素级位置，那么Mask R-CNN能有效利用这些详尽的标注信息进一步提升目标检测的精度。

如下图所示，Mask R-CNN在Faster R-CNN的基础上做了修改。Mask R-CNN将兴趣区域池化层替换成**兴趣区域对齐层**，即通过**双线性插值 (bilinear interpolation)** 来保留特征图上的空间信息，从而更适于像素级预测。兴趣区域对齐层的输出包含了所有兴趣区域的形状相同的特征图。它们既用来预测兴趣区域的类别和边界框，又通过额外的全卷积网络预测目标的像素级位置。我们将在8.10节（全

卷积网络) 介绍如何使用全卷积网络预测图像中像素级的语义。



### 小结:

- R-CNN对图像选取若干提议区域，然后用卷积神经网络对每个提议区域做前向计算抽取特征，再用这些特征预测提议区域的类别和边界框。
- Fast R-CNN对R-CNN的一个主要改进在于只对整个图像做卷积神经网络的前向计算。它引入了兴趣区域池化层，从而令兴趣区域能够抽取出形状相同的特征。
- Faster R-CNN将Fast R-CNN中的选择性搜索替换成区域提议网络，从而减少提议区域的生成数量，并保证目标检测的精度。
- Mask R-CNN在Faster R-CNN基础上引入一个全卷积网络，从而借助目标的像素级位置进一步提升目标检测的精度。

## 8.9 语义分割和数据集

在前几节讨论的目标检测问题中，我们一直使用方形边界框来标注和预测图像中的目标。本节将探讨语义分割 (semantic segmentation) 问题，它关注如何将图像分割成属于不同语义类别的区域。值得一提的是，这些语义区域的标注和预测都是像素级的。下图展示了语义分割中图像有关狗、猫和背景的标签。可以看到，与目标检测相比，语义分割标注的像素级的边框显然更加精细。



### 8.9.1 图像分割和实例分割

计算机视觉领域还有2个与语义分割相似的重要问题，即图像分割 (image segmentation) 和实例分割 (instance segmentation)。我们在这里将它们与语义分割简单区分一下。

- 图像分割将图像分割成若干组成区域。这类问题的方法通常利用图像中像素之间的相关性。它在训练时不需要有关图像像素的标签信息，在预测时也无法保证分割出的区域具有我们希望得到的语义。以上图的图像为输入，图像分割可能将狗分割成两个区域：一个覆盖以黑色为主的嘴巴和眼睛，而另一个覆盖以黄色为主的其余部分身体。
- 实例分割又叫同时检测并分割 (simultaneous detection and segmentation)。它研究如何识别图像中各个目标实例的像素级区域。与语义分割有所不同，实例分割不仅需要区分语义，还要区分

不同的目标实例。如果图像中有两只狗，实例分割需要区分像素属于这两只狗中的哪一只。

## 8.9.2 Pascal VOC2012语义分割数据集

语义分割的一个重要数据集叫作Pascal VOC2012。为了更好地了解这个数据集，我们先导入实验所需的包或模块。

```
1 In [1]:  
2     import time  
3     import torch  
4     import torch.nn.functional as F  
5     import torchvision  
6     import numpy as np  
7     from PIL import Image  
8     from tqdm import tqdm  
9     import sys  
10    import d2lzh as d2l
```

我们先下载这个数据集的压缩包到 `data` 路径下。压缩包大小是2 GB左右，下载需要一定时间。解压之后的数据集会放置在 `data/vocdevkit/voc2012` 路径下。

```
1 In [2]:  
2     ! ls data/vocdevkit/voc2012  
3 Out [2]:  
4     Annotations  ImageSets  JPEGImages  SegmentationClass  
5     SegmentationObject
```

进入 `data/vocdevkit/voc2012` 路径后，我们可以获取数据集的不同组成部分。其中 `ImageSets/Segmentation` 路径包含了指定训练和测试样本的文本文件，而 `JPEGImages` 和 `SegmentationClass` 路径下分别包含了样本的输入图像和标签。这里的标签也是图像格式，其尺寸和它所标注的输入图像的尺寸相同。标签中颜色相同的像素属于同一个语义类别。下面定义 `read_voc_images` 函数将输入图像和标签读进内存。

```
1 In [3]:  
2     def read_voc_images(root='data/vocdevkit/voc2012',  
3                         is_train=True, max_num=None):  
4         txt_fname = '%s/ImageSets/Segmentation/%s' % (  
5             root, 'train.txt' if is_train else 'val.txt'  
6         )  
7         with open(txt_fname, 'r') as f:  
8             images = f.read().split()  
9             if max_num is not None:  
10                 images = images[:min(max_num, len(images))]  
11             features, labels = [None]*len(images), [None]*len(images)  
12             for i, fname in tqdm(enumerate(images)):  
13                 features[i] = Image.open(  
14                     '%s/JPEGImages/%s.jpg' % (root, fname)).convert('RGB')  
15                 labels[i] = Image.open(  
16                     '%s/SegmentationClass/%s.png' % \  
17                     (root, fname)).convert('RGB')  
18             return features, labels  
19             voc_dir = 'data/vocdevkit/voc2012'  
20             train_features, train_labels = read_voc_images(voc_dir, max_num=100)
```

我们画出前5张输入图像和它们的标签。在标签图像中，白色和黑色分别代表边框和背景，而其他不同的颜色则对应不同的类别。

```
1 In [4]:  
2     n = 5  
3     imgs = train_features[:n]+train_labels[:n]  
4     d2l.show_images(imgs, 2, n);
```



接下来，我们列出标签中每个RGB颜色的值及其标注的类别。

```
1 In [5]:  
2     VOC_COLORMAP = [[0, 0, 0], [128, 0, 0], [0, 128, 0], [128, 128, 0],  
3                         [0, 0, 128], [128, 0, 128], [0, 128, 128],  
4                         [128, 128, 128], [64, 0, 0], [192, 0, 0],  
5                         [64, 128, 0], [192, 128, 0], [64, 0, 128],  
6                         [192, 0, 128], [64, 128, 128], [192, 128, 128],  
7                         [0, 64, 0], [128, 64, 0], [0, 192, 0], [128, 192, 0],  
8                         [0, 64, 128]]  
9     VOC_CLASSES = ['background', 'aeroplane', 'bicycle', 'bird', 'boat',  
10                      'bottle', 'bus', 'car', 'cat', 'chair', 'cow',  
11                      'diningtable', 'dog', 'horse', 'motorbike', 'person',  
12                      'potted plant', 'sheep', 'sofa', 'train', 'tv/monitor']
```

有了上面定义的两个常量以后，我们可以很容易地查找标签中每个像素的类别索引。

```
1 In [6]:  
2     colormap2label = torch.zeros(256**3, dtype=torch.uint8)  
3     for i, colormap in enumerate(VOC_COLORMAP):  
4         colormap2label[(colormap[0]*256+colormap[1])*256+colormap[2]] = i  
5     def voc_label_indices(colormap, colormap2label):  
6         """  
7             将colormap(PIL)转换为colormap2label(uint8)  
8         """  
9         colormap = np.array(colormap.convert('RGB')).astype('int32')  
10        idx = ((colormap[:, :, 0]*256+colormap[:, :, 1])*256+  
11                  colormap[:, :, 2])  
12        return colormap2label[idx]
```

例如，第一张样本图像中飞机头部区域的类别索引为1，而背景全是0。

```
1 In [7]:  
2     y = voc_label_indices(train_labels[0], colormap2label)  
3     y[105:115, 130:140], VOC_CLASSES[1]
```

```

4  out [7]:
5      (tensor([[0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
6                  [0, 0, 0, 0, 0, 0, 0, 1, 1, 1],
7                  [0, 0, 0, 0, 0, 0, 1, 1, 1, 1],
8                  [0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
9                  [0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
10                 [0, 0, 0, 0, 1, 1, 1, 1, 1, 1],
11                 [0, 0, 0, 0, 1, 1, 1, 1, 1, 1],
12                 [0, 0, 0, 0, 1, 1, 1, 1, 1, 1],
13                 [0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
14                 [0, 0, 0, 0, 0, 0, 0, 1, 1, 1]], dtype=torch.uint8),
15      'aeroplane')

```

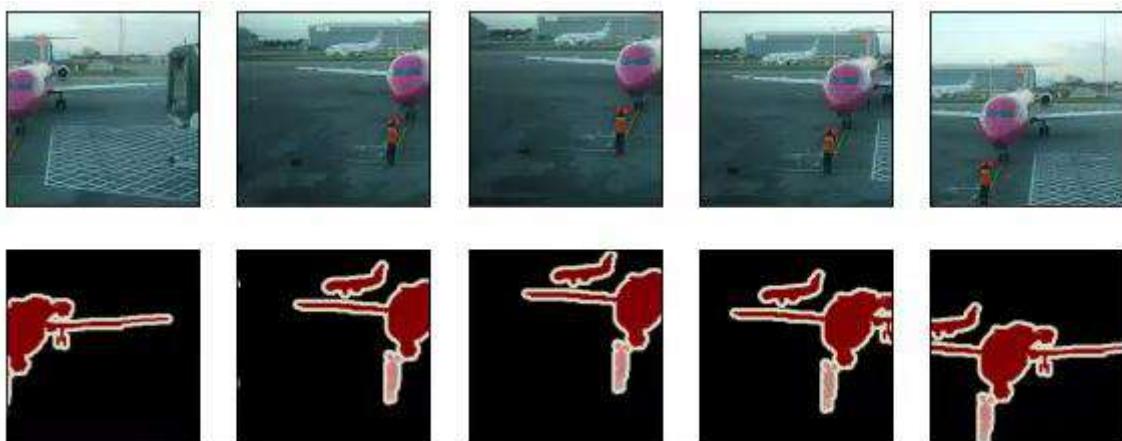
## 1. 预处理数据

在之前的章节中，我们通过缩放图像使其符合模型的输入形状。然而在语义分割里，这样做需要将预测的像素类别重新映射回原始尺寸的输入图像。这样的映射难以做到精确，尤其在不同语义的分割区域。为了避免这个问题，我们将图像裁剪成固定尺寸而不是缩放。具体来说，我们使用图像增广里的随机裁剪，并对输入图像和标签裁剪相同区域。

```

1 In [8]:
2     def voc_rand_crop(feature, label, height, width):
3         """
4             同时对图片和标签进行随机裁剪为固定大小
5         """
6         i, j, h, w = torchvision.transforms.RandomCrop.get_params(
7             feature, output_size=(height, width)
8         )
9         feature = torchvision.transforms.functional.crop(
10            feature, i, j, h, w)
11         label = torchvision.transforms.functional.crop(label, i, j, h, w)
12         return feature, label
13     imgs = []
14     for _ in range(n):
15         imgs += voc_rand_crop(train_features[0], train_labels[0], 200, 200)
16     d2l.show_images(imgs[::2]+imgs[1::2], 2, n);

```



## 2. 自定义语义分割数据集类

我们通过继承PyTorch提供的 `Dataset` 类自定义了一个语义分割数据集类 `vocSegDataset`。通过实现 `__getitem__` 函数，我们可以任意访问数据集中索引为 `idx` 的输入图像及其每个像素的类别索引。由于数据集中有些图像的尺寸可能小于随机裁剪所指定的输出尺寸，这些样本需要通过自定义的 `filter` 函数所移除。此外，我们还对输入图像的RGB三个通道的值分别做标准化。

```

1 class VOCSegDataset(torch.utils.data.Dataset):
2     def __init__(self, is_train, crop_size, voc_dir,
3                  colormap2label, max_num=None):
4         # crop_size - 随机裁剪后的大小
5         self.rgb_mean = np.array([0.485, 0.456, 0.406])
6         self.rgb_std = np.array([0.229, 0.224, 0.225])
7         self.tsf = torchvision.transforms.Compose([
8             torchvision.transforms.ToTensor(),
9             torchvision.transforms.Normalize(mean=self.rgb_mean,
10                                             std=self.rgb_std)
11         ])
12         self.crop_size = crop_size #(h, w)
13         features, labels = read_voc_images(root=voc_dir,
14                                           is_train=is_train,
15                                           max_num=max_num)
16         self.features = self.filter(features)
17         self.labels = self.filter(labels)
18         self.colormap2label = colormap2label
19         print('read ' + str(len(self.features)) + ' valid examples')
20     def filter(self, imgs):
21         return [img for img in imgs if (
22             img.size[1] >= self.crop_size[0] and
23             img.size[0] >= self.crop_size[1]
24         )]
25     def __getitem__(self, idx):
26         feature, label = voc_rand_crop(
27             self.features[idx], self.labels[idx], *self.crop_size)
28         # float32, uint8
29         return (self.tsf(feature),
30                 voc_label_indices(label, self.colormap2label))
31     def __len__(self):
32         return len(self.features)

```

### 3. 读取数据集

我们通过自定义的 `vocsegdataset` 类来分别创建训练集和测试集的实例。假设我们指定随机裁剪的输出图像的形状为  $320 \times 480$ 。下面我们可以查看训练集和测试集所保留的样本个数。

```

1 In [9]:
2     crop_size = (320, 480)
3     max_num = 100
4     voc_train = VOCSegDataset(True, crop_size, voc_dir,
5                               colormap2label, max_num)
5     voc_test = VOCSegDataset(False, crop_size, voc_dir,
6                               colormap2label, max_num)
7
8 Out [9]:
9     read 75 valid examples
10    read 77 valid examples

```

设批量大小为64，分别定义训练集和测试集的迭代器。

```
1 In [10]:  
2     batch_size = 64  
3     num_workers = 0 if sys.platform.startswith('win32') else 4  
4     train_iter = torch.utils.data.DataLoader(  
5         voc_train, batch_size, shuffle=True,  
6         drop_last=True, num_workers=num_workers)  
7     test_iter = torch.utils.data.DataLoader(  
8         voc_test, batch_size, drop_last=True,  
9         num_workers=num_workers)
```

打印第一个小批量的类型和形状。不同于图像分类和目标识别，这里的标签是一个三维数组。

```
1 In [11]:  
2     for X, Y in train_iter:  
3         print(X.dtype, X.shape)  
4         print(Y.dtype, Y.shape)  
5         break  
6 Out [11]:  
7     torch.float32 torch.size([64, 3, 320, 480])  
8     torch.uint8 torch.size([64, 320, 480])
```

## 小结:

- 语义分割关注如何将图像分割成属于不同语义类别的区域。
- 语义分割的一个重要数据集叫作Pascal VOC2012。
- 由于语义分割的输入图像和标签在像素上一一对应，所以将图像随机裁剪成固定尺寸而不是缩放。

## 8.10 全卷积网络 (FCN)

上一节介绍了，我们可以基于语义分割对图像中的每个像素进行类别预测。全卷积网络 (fully convolutional network, FCN) 采用卷积神经网络实现了从图像像素到像素类别的变换。与之前介绍的卷积神经网络有所不同，全卷积网络通过转置卷积 (transposed convolution) 层将中间层特征图的高和宽变换回输入图像的尺寸，从而令预测结果与输入图像在空间维（高和宽）上一一对应：**给定空间维上的位置，通道维的输出即该位置对应像素的类别预测。**

我们先导入实验所需的包或模块，然后解释什么是转置卷积层。

```
1 In [1]:  
2     import torch  
3     import torchvision  
4     from torch import nn  
5     from torch.nn import functional as F  
6     import d2lzh as d2l
```

### 8.10.1 转置卷积层

顾名思义，转置卷积层得名于矩阵的转置操作。事实上，卷积运算还可以通过矩阵乘法来实现。在下面的例子中，我们定义高和宽分别为4的输入  $x$ ，以及高和宽分别为3的卷积核  $k$ 。打印二维卷积运算的输出以及卷积核。可以看到，输出的高和宽分别为2。

```
1 In [2]:  
2     X = torch.arange(1, 17).view((1, 1, 4, 4)).float()  
3     K = torch.arange(1, 10).view((1, 1, 3, 3)).float()  
4     conv = nn.Conv2d(in_channels=1, out_channels=1,
```

```

5                 kernel_size=3, bias=False)
6         conv.load_state_dict({'weight': K})
7         conv(X), K
8 Out [2]:
9         (tensor([[[[348., 393.],
10            [528., 573.]]]],
11            grad_fn=<MkldnnConvolutionBackward>),
12            tensor([[[[1., 2., 3.],
13              [4., 5., 6.],
14              [7., 8., 9.]]]]))

```

下面我们将卷积核  $K$  改写成含有大量零元素的稀疏矩阵  $w$ ，即权重矩阵。权重矩阵的形状为(4, 16)，其中的非零元素来自卷积核  $K$  中的元素。将输入  $x$  逐行连结，得到长度为16的向量。然后将  $w$  与向量化的  $x$  做矩阵乘法，得到长度为4的向量。对其变形后，我们可以得到和上面卷积运算相同的结果。可见，我们在这个例子中使用矩阵乘法实现了卷积运算。

```

1 In [3]:
2     w, k = torch.zeros((4, 16)), torch.zeros(11)
3     k[:3], k[4:7], k[8:] = K[0, 0, 0, :], K[0, 0, 1, :], K[0, 0, 2, :]
4     w[0, 0:11], w[1, 1:12], w[2, 4:15], w[3, 5:16] = k, k, k, k
5     torch.mm(w, X.view((16, 1))).view((1, 1, 2, 2)), w
6 Out [3]:
7         (tensor([[[[348., 393.],
8            [528., 573.]]]]),
9            tensor([[1., 2., 3., 0., 4., 5., 6., 0., 7., 8., 9., 0., 0., 0.,
10               0., 1., 2., 3., 0., 4., 5., 6., 0., 7., 8., 9., 0., 0., 0.,
11               0., 0., 0., 0., 1., 2., 3., 0., 4., 5., 6., 0., 7., 8., 9.,
12               0., 0., 0., 0., 1., 2., 3., 0., 4., 5., 6., 0., 7., 8., 9.,
13               0.])))

```

现在我们从矩阵乘法的角度来描述卷积运算。设输入向量为  $x$ ，权重矩阵为  $W$ ，卷积的前向计算函数的实现可以看作将函数输入乘以权重矩阵，并输出向量  $y = Wx$ 。我们知道，反向传播需要依据链式法则。由于  $\nabla_x y = W^\top$ ，卷积的反向传播函数的实现可以看作将函数输入乘以转置后的权重矩阵  $W^\top$ 。而转置卷积层正好交换了卷积层的前向计算函数与反向传播函数：转置卷积层的这两个函数可以看作将函数输入向量分别乘以  $W^\top$  和  $W$ 。

不难想象，转置卷积层可以用来交换卷积层输入和输出的形状。让我们继续用矩阵乘法描述卷积。设权重矩阵是形状为  $4 \times 16$  的矩阵，对于长度为16的输入向量，卷积前向计算输出长度为4的向量。假如输入向量的长度为4，转置权重矩阵的形状为  $16 \times 4$ ，那么转置卷积层将输出长度为16的向量。在模型设计中，**转置卷积层常用于将较小的特征图变换为更大的特征图**。在全卷积网络中，当输入是高和宽较小的特征图时，转置卷积层可以用来将高和宽放大到输入图像的尺寸。

我们来看一个例子。构造一个卷积层 `conv`，并设输入  $x$  的形状为(1, 3, 64, 64)。卷积输出  $Y$  的通道数增加到10，但高和宽分别缩小了一半。

```
1 In [4]:  
2     x = torch.zeros((1, 3, 64, 64))  
3     conv = torch.nn.Conv2d(in_channels=3, out_channels=10,  
4                           kernel_size=4, padding=1, stride=2)  
5     Y = conv(x)  
6     Y.shape  
7 Out [4]:  
8     torch.Size([1, 10, 32, 32])
```

下面我们通过创建 ConvTranspose2d 实例来构造转置卷积层 conv\_trans。这里我们设 conv\_trans 的卷积核形状、填充以及步幅与 conv 中的相同，并设输出通道数为3。当输入为卷积层 conv 的输出 Y 时，转置卷积层输出与卷积层输入的高和宽相同：转置卷积层将特征图的高和宽分别放大了2倍。

```
1 In [5]:  
2     conv_trans = torch.nn.ConvTranspose2d(in_channels=10, out_channels=3,  
3                                         kernel_size=4, padding=1, stride=2)  
4     conv_trans(Y).shape  
5 Out [5]:  
6     torch.Size([1, 3, 64, 64])
```

在有些文献中，转置卷积也被称为分数步长卷积 (fractionally-strided convolution)。

## 8.10.2 构造模型

我们在这里给出全卷积网络模型最基本的设计。如下图所示，全卷积网络先使用卷积神经网络抽取图像特征，然后通过 $1 \times 1$ 卷积层将通道数变换为类别个数，最后通过转置卷积层将特征图的高和宽变换为输入图像的尺寸。模型输出与输入图像的高和宽相同，并在空间位置上一一对应：最终输出的通道包含了该空间位置像素的类别预测。



下面我们使用一个基于ImageNet数据集预训练的ResNet-18模型来抽取图像特征，并将该网络实例记为 pretrained\_net。可以看到，该模型的最后两层分别是自适应平均池化层 AdaptiveAvgPool2d 和全连接层 Linear，全卷积网络不需要使用这些层。

```
1 In [6]:  
2     pretrained_net = torchvision.models.resnet18(pretrained=True)  
3     pretrained_net.layer4[1], pretrained_net.avgpool, pretrained_net.fc  
4 Out [6]:
```

```
5     (BasicBlock(
6         (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
7             padding=(1, 1), bias=False)
8         (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
9             track_running_stats=True)
10        (relu): ReLU(inplace)
11        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
12            padding=(1, 1), bias=False)
13        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
14            track_running_stats=True)
15    ),
16    AdaptiveAvgPool2d(output_size=(1, 1)),
17    Linear(in_features=512, out_features=1000, bias=True))
```

下面我们创建全卷积网络实例`net`。它复制了`pretrained\_net`实例的成员变量`features`里除去最后两层的所有层以及预训练得到的模型参数。

```
1 In [7]:
2     net = nn.Sequential(*list(pretrained_net.children())[:-2])
```

给定高和宽分别为320和480的输入，`net`的前向计算将输入的高和宽减小至原来的 $1/32$ ，即10和15。

```
1 In [8]:
2     x = torch.rand(size=(1, 3, 320, 480))
3     net(x).shape
4 Out [8]:
5     torch.Size([1, 512, 10, 15])
```

接下来，我们通过 $1 \times 1$ 卷积层将输出通道数变换为Pascal VOC2012数据集的类别个数21。最后，我们需要将特征图的高和宽放大32倍，从而变回输入图像的高和宽。回忆一下4.2节中描述的卷积层输出形状的计算方法。由于 $(320 - 64 + 16 \times 2 + 32)/32 = 10$ 且 $(480 - 64 + 16 \times 2 + 32)/32 = 15$ ，我们构造一个步幅为32的转置卷积层，并将卷积核的高和宽设为64、填充设为16。**不难发现，如果步幅为s、填充为s/2（假设s/2为整数）、卷积核的高和宽为2s，转置卷积核将输入的高和宽分别放大s倍。**

```
1 In [9]:
2     num_classes = 21
3     # 在原来网络的基础上添加层
4     net.add_module('final_conv', nn.Conv2d(512, num_classes, kernel_size=1))
5     net.add_module('transpose_conv', nn.ConvTranspose2d(
6         num_classes, num_classes, kernel_size=64, padding=16, stride=32))
```

### 8.10.3 初始化转置卷积层

我们已经知道，转置卷积层可以放大特征图。在图像处理中，我们有时需要将图像放大，即上采样(upsample)。上采样的方法有很多，常用的有**双线性插值**。简单来说，为了得到输出图像在坐标 $(x, y)$ 上的像素，先将该坐标映射到输入图像的坐标 $(x', y')$ ，例如，根据输入与输出的尺寸之比来映射。映射后的 $x'$ 和 $y'$ 通常是实数。然后，在输入图像上找到与坐标 $(x', y')$ 最近的4像素。最后，输出图像在坐标 $(x, y)$ 上的像素依据输入图像上这4像素及其与 $(x', y')$ 的相对距离来计算。双线性插值的上采样可以通过由以下`bilinear\_kernel`函数构造的卷积核的转置卷积层来实现。限于篇幅，我们只给出`bilinear\_kernel`函数的实现，不再讨论算法的原理。

```
1 In [10]:  
2     def bilinear_kernel(in_channels, out_channels, kernel_size):  
3         factor = (kernel_size+1)//2  
4         if kernel_size % 2 == 1:  
5             center = factor - 1  
6         else:  
7             center = factor - 0.5  
8         og = (torch.arange(kernel_size).reshape(-1, 1),  
9                torch.arange(kernel_size).reshape(1, -1))  
10        filt = (1-torch.abs(og[0] - center) / factor) * \  
11            (1-torch.abs(og[1] - center) / factor)  
12        weight = torch.zeros((in_channels, out_channels,  
13                               kernel_size, kernel_size))  
14        weight[range(in_channels), range(out_channels), :, :] = filt.float()  
15    return weight
```

我们来实验一下用转置卷积层实现的双线性插值的上采样。构造一个将输入的高和宽放大2倍的转置卷积层，并将其卷积核用 `bilinear_kernel` 函数初始化。

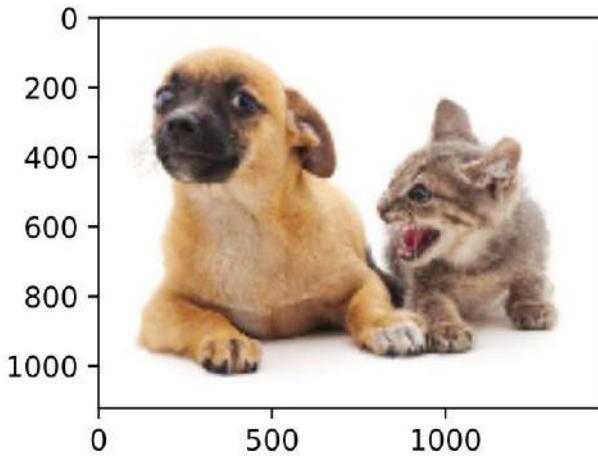
```
1 In [11]:  
2     conv_trans = nn.ConvTranspose2d(3, 3, kernel_size=4,  
3                                     padding=1, stride=2, bias=False)  
4     conv_trans.weight.data.copy_(bilinear_kernel(3, 3, 4));
```

读取图像 `x`，将上采样的结果记作 `Y`。为了打印图像，我们需要调整通道维的位置。

```
1 In [12]:  
2     img = torchvision.transforms.ToTensor()(  
3         d2l.Image.open('data/catdog.jpg'))  
4     X = img.unsqueeze(0)  
5     Y = conv_trans(X)  
6     out_img = Y[0].permute(1, 2, 0).detach()
```

可以看到，转置卷积层将图像的高和宽分别放大2倍。值得一提的是，除了坐标刻度不同，双线性插值放大的图像和8.3节中打印出的原图看上去没什么两样。

```
1 In [13]:  
2     d2l.set_figsize()  
3     print('input image shape:', img.permute(1, 2, 0).shape)  
4     d2l.plt.imshow(img.permute(1, 2, 0));  
5     print('output image shape:', out_img.shape)  
6     d2l.plt.imshow(out_img);  
7 Out [13]:  
8     input image shape: torch.Size([561, 728, 3])  
9     output image shape: torch.Size([1122, 1456, 3])
```



在全卷积网络中，我们将转置卷积层初始化为双线性插值的上采样。对于 $1 \times 1$ 卷积层，我们采用 Xavier随机初始化。

```

1 In [14]:
2     w = bilinear_kernel(num_classes, num_classes, 64)
3     net.transpose_conv.weight.data.copy_(w);
4     torch.nn.init.xavier_uniform_(net.final_conv.weight.data);
```

## 8.10.4 读取数据集

我们用上一节介绍的方法读取数据集。这里指定随机裁剪的输出图像的形状为 $320 \times 480$ ：高和宽都可以被32整除。

```

1 In [15]:
2     crop_size = (320, 480)
3     batch_size = 32
4     voc_train = VOCSegDataset(True, crop_size, voc_dir, colormap2label)
5     voc_test = VOCSegDataset(False, crop_size, voc_dir, colormap2label)
6     num_workers = 0 if sys.platform.startswith('win32') else 4
7     train_iter = torch.utils.data.DataLoader(
8         voc_train, batch_size, shuffle=True,
9         drop_last=True, num_workers=num_workers)
10    test_iter = torch.utils.data.DataLoader(
11        voc_test, batch_size, drop_last=True, num_workers=num_workers)
12 Out [15]:
13    read 1114 examples
14    read 1078 examples
```

## 8.10.5 训练模型

现在可以开始训练模型了。这里的损失函数和准确率计算与图像分类中的并没有本质上的不同。因为我们使用转置卷积层的通道来预测像素的类别，所以在 cross\_entropy 里指定了 reduction='none' 选项。此外，模型基于每个像素的预测类别是否正确来计算准确率。

```

1 In [16]:
2     device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
3     def loss(inputs, targets):
4         loss_calc = F.cross_entropy(
5             inputs, targets.long(), reduction='none').mean()
6         return loss_calc
7     num_epochs, lr, wd, devices = 10, 0.003, 1e-3, device
```

```

8   trainer = torch.optim.SGD(net.parameters(), lr=lr, weight_decay=wd)
9   # 这里的训练函数与d2l.train()一样，只是修改了结果打印的格式
10  d2l.train_ch8(train_iter, test_iter, net, loss,
11      trainer, device, num_epochs)
12 Out [16]:
13 Let's use 2 GPUs!
14 training on cuda
15 epoch 1, loss 0.4443, train acc 0.869, test acc 0.822, time 16.7 sec
16 epoch 2, loss 0.2201, train acc 0.870, test acc 0.826, time 16.4 sec
17 epoch 3, loss 0.1441, train acc 0.872, test acc 0.825, time 16.5 sec
18 epoch 4, loss 0.1043, train acc 0.876, test acc 0.827, time 16.7 sec
19 epoch 5, loss 0.0834, train acc 0.876, test acc 0.826, time 16.5 sec
20 epoch 6, loss 0.0699, train acc 0.876, test acc 0.829, time 16.7 sec
21 epoch 7, loss 0.0593, train acc 0.876, test acc 0.827, time 16.7 sec
22 epoch 8, loss 0.0502, train acc 0.879, test acc 0.829, time 16.4 sec
23 epoch 9, loss 0.0445, train acc 0.879, test acc 0.827, time 16.7 sec
24 epoch 10, loss 0.0402, train acc 0.878, test acc 0.828, time 16.7 sec

```

## 8.10.6 预测像素类别

在预测时，我们需要将输入图像在各个通道做标准化，并转成卷积神经网络所需要的四维输入格式。

```

1 In [17]:
2     def predict(img):
3         rgb_mean = np.array([0.485, 0.456, 0.406])
4         rgb_std = np.array([0.229, 0.224, 0.225])
5         tsf = torchvision.transforms.Compose([
6             torchvision.transforms.ToTensor(),
7             torchvision.transforms.Normalize(mean=rgb_mean,
8                                             std=rgb_std)
9         ])
10        # X = np.array(img)/255.0
11        # X = torch.from_numpy(X).unsqueeze(0).permute(0, 3, 1, 2).float()
12        X = tsf(img).unsqueeze(0)
13        pred = net(X.to(device)).argmax(dim=1)
14        return pred.reshape(pred.shape[1], pred.shape[2])

```

为了可视化每个像素的预测类别，我们将预测类别映射回它们在数据集中的标注颜色。

```

1 In [18]:
2     def label2image(pred):
3         colormap = torch.tensor(d2l.VOC_COLORMAP, device=device)
4         X = pred.long()
5         return colormap[X, :]

```

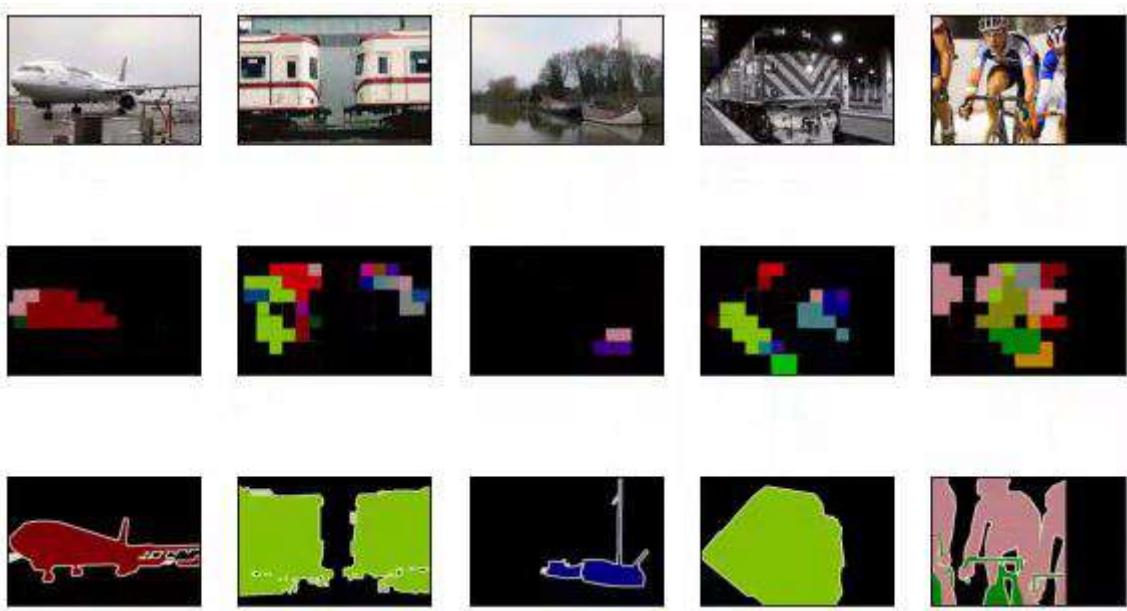
测试数据集中的图像大小和形状各异。由于模型使用了步幅为32的转置卷积层，当输入图像的高或宽无法被32整除时，转置卷积层输出的高或宽会与输入图像的尺寸有偏差。为了解决这个问题，我们可以在图像中截取多块高和宽为32的整数倍的矩形区域，并分别对这些区域中的像素做前向计算。这些区域的并集需要完整覆盖输入图像。当一个像素被多个区域所覆盖时，它在不同区域前向计算中转置卷积层输出的平均值可以作为softmax运算的输入，从而预测类别。

简单起见，我们只读取几张较大的测试图像，并从图像的左上角开始截取形状为 $320 \times 480$ 的区域：只有该区域用于预测。对于输入图像，我们先打印截取的区域，再打印预测结果，最后打印标注的类别。

```

1 In [19]:
2     test_images, test_labels = d2l.read_voc_images(is_train=False)
3     n, imgs = 5, []
4     for i in range(n):
5         crop_rect = (0, 0, 320, 480)
6         x = torchvision.transforms.functional.crop(
7             test_images[i], *crop_rect)
8         pred = label2image(predict(x))
9         imgs += [torch.from_numpy(np.array(x)),
10            pred.cpu(),
11            torchvision.transforms.functional.crop(
12                test_labels[i], *crop_rect)]
13 d2l.show_images(imgs[::3]+imgs[1::3]+imgs[2::3], 3, n, scale=2);

```



### 小结:

- 可以通过矩阵乘法来实现卷积运算。
- 全卷积网络先使用卷积神经网络抽取图像特征，然后通过 $1 \times 1$ 卷积层将通道数变换为类别个数，最后通过转置卷积层将特征图的高和宽变换为输入图像的尺寸，从而输出每个像素的类别。
- 在全卷积网络中，可以将转置卷积层初始化为双线性插值的上采样。

## 8.11 样式迁移

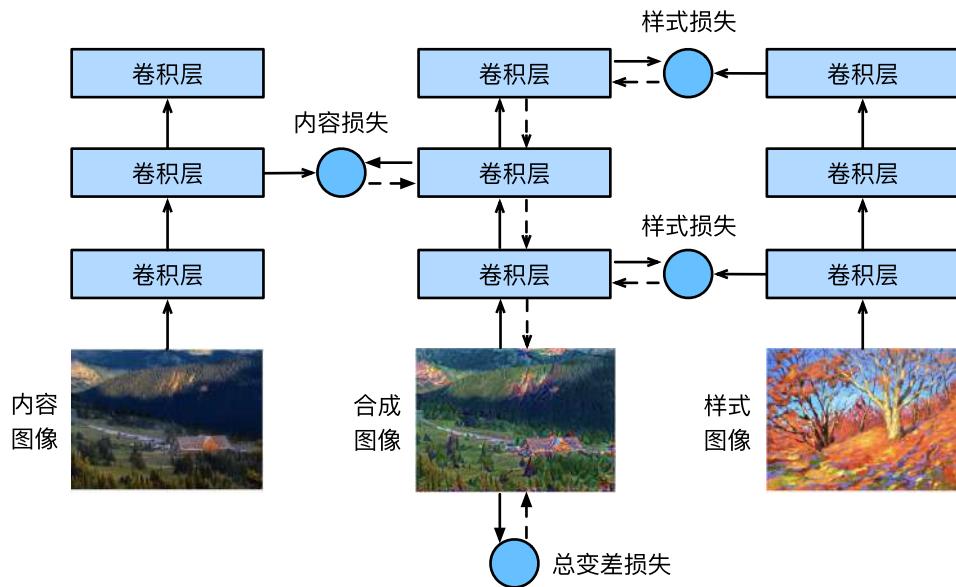
如果你是一位摄影爱好者，也许接触过滤镜。它能改变照片的颜色样式，从而使风景照更加锐利或者令人像更加美白。但一个滤镜通常只能改变照片的某个方面。如果要照片达到理想中的样式，经常需要尝试大量不同的组合，其复杂程度不亚于模型调参。

在本节中，我们将介绍如何使用卷积神经网络自动将某图像中的样式应用在另一图像之上，即样式迁移（style transfer）。这里我们需要两张输入图像，一张是内容图像，另一张是样式图像，我们将使用神经网络修改内容图像使其在样式上接近样式图像。下图中的内容图像为本书作者在西雅图郊区的雷尼尔山国家公园（Mount Rainier National Park）拍摄的风景照，而样式图像则是一副主题为秋天橡树的油画。最终输出的合成图像在保留了内容图像中物体主体形状的情况下应用了样式图像的油画笔触，同时也让整体颜色更加鲜艳。



### 8.11.1 方法

下图用一个例子来阐述基于卷积神经网络的样式迁移方法。首先，我们初始化合成图像，例如将其初始化成内容图像。该合成图像是样式迁移过程中唯一需要更新的变量，即样式迁移所需迭代的模型参数。然后，我们选择一个预训练的卷积神经网络来抽取图像的特征，其中的模型参数在训练中无须更新。深度卷积神经网络凭借多个层逐级抽取图像的特征。我们可以选择其中某些层的输出作为内容特征或样式特征。以下图为例，这里选取的预训练的神经网络含有3个卷积层，其中第二层输出图像的内容特征，而第一层和第三层的输出被作为图像的样式特征。接下来，我们通过正向传播（实线箭头方向）计算样式迁移的损失函数，并通过反向传播（虚线箭头方向）迭代模型参数，即不断更新合成图像。样式迁移常用的损失函数由3部分组成：**内容损失**（content loss）使合成图像与内容图像在内容特征上接近，**样式损失**（style loss）令合成图像与样式图像在样式特征上接近，而**总变差损失**（total variation loss）则有助于减少合成图像中的噪点。最后，当模型训练结束时，我们输出样式迁移的模型参数，即得到最终的合成图像。



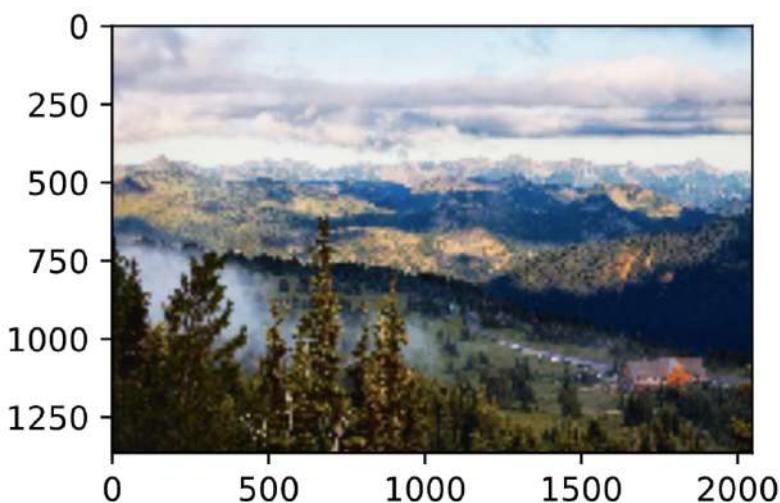
下面，我们通过实验来进一步了解样式迁移的技术细节。实验需要用到一些导入的包或模块。

```
1 In [1]:  
2     import time  
3     import torch  
4     import torch.nn.functional as F  
5     import torchvision  
6     import numpy as np  
7     from PIL import Image  
8     import sys  
9     import d2lzh as d2l  
10    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

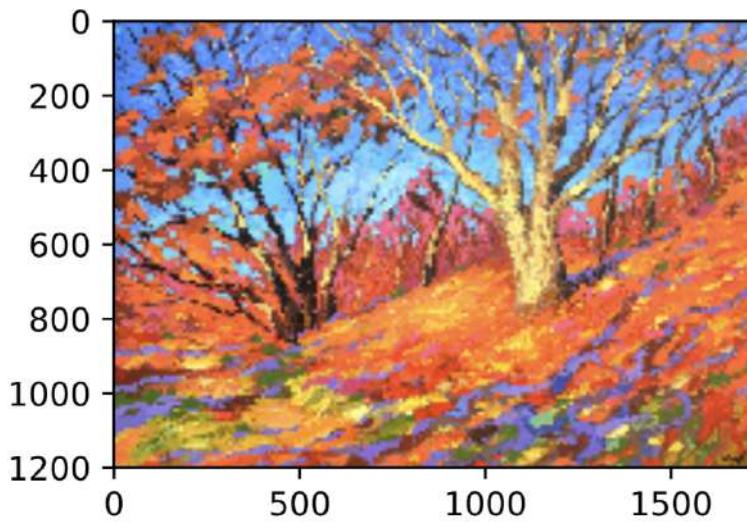
## 8.11.2 读取内容图像和样式图像

首先，我们分别读取内容图像和样式图像。从打印出的图像坐标轴可以看出，它们的尺寸并不一样。

```
1 In [2]:  
2     d2l.set_figsize()  
3     content_img = Image.open('data/rainier.jpg')  
4     d2l.plt.imshow(content_img);
```



```
1 In [3]:  
2     d2l.set_figsize()  
3     style_img = Image.open('data/autumn_oak.jpg')  
4     d2l.plt.imshow(style_img);
```



### 8.11.3 预处理和后处理图像

下面定义图像的预处理函数和后处理函数。预处理函数 `preprocess` 对先对更改输入图像的尺寸，然后再将PIL图片转成卷积神经网络接受的输入格式，再在RGB三个通道分别做标准化，由于预训练模型是在均值为[0.485, 0.456, 0.406]标准差为[0.229, 0.224, 0.225]的图片数据上预训练的，所以我们要将图片标准化保持相同的均值和标准差。后处理函数 `postprocess` 则将输出图像中的像素值还原回标准化之前的值。由于图像每个像素的浮点数值在0到1之间，我们使用 `clamp` 函数对小于0和大于1的值分别取0和1。

`torchvision.transforms` 模块有大量现成的转换方法，不过需要注意的是有的方法输入的是PIL图像，如 `Resize`；有的方法输入的是 `tensor`，如 `Normalize`；而还有的是用于二者转换，如 `ToTensor` 将PIL图像转换成 `tensor`。一定要注意这点，使用时看清[文档](#)。

```

1 In [4]:
2     rgb_mean = np.array([0.485, 0.456, 0.406])
3     rgb_std = np.array([0.229, 0.224, 0.225])
4     def preprocess(PIL_img, image_shape):
5         process = torchvision.transforms.Compose([
6             torchvision.transforms.Resize(image_shape),
7             torchvision.transforms.ToTensor(),
8             torchvision.transforms.Normalize(mean=rgb_mean,
9                                             std=rgb_std)
10        ])
11        # (batch_size, 3, H, W)
12        return process(PIL_img).unsqueeze(dim=0)
13    def postprocess(img_tensor):
14        inv_normalize = torchvision.transforms.Normalize(
15            mean = -rgb_mean/rgb_std,
16            std = 1/rgb_std
17        )
18        to_PIL_image = torchvision.transforms.ToPILImage()
19        return to_PIL_image(
20            inv_normalize(img_tensor[0].cpu()).clamp(0, 1))

```

## 8.11.4 抽取特征

我们使用基于ImageNet数据集预训练的VGG-19模型来抽取图像特征。

PyTorch官方在 `torchvision.models` 模块提供了一些常见的预训练好的计算机视觉模型，包括图片分类、语义分割、目标检测、实例分割、人关键点检测和视频分类等等。使用时要仔细阅读其[文档](#)，搞清楚如何使用，例如刚刚提到的对图片进行标准化等。

```
1 In [5]:  
2     pretrained_net = torchvision.models.vgg19(  
3         pretrained=True, progress=True)
```

为了抽取图像的内容特征和样式特征，我们可以选择VGG网络中某些层的输出。一般来说，越靠近输入层的输出越容易抽取图像的细节信息，反之则越容易抽取图像的全局信息。为了避免合成图像过多保留内容图像的细节，我们选择VGG较靠近输出的层，也称内容层，来输出图像的内容特征。我们还从VGG中选择不同层的输出来匹配局部和全局的样式，这些层也叫样式层。在4.7节（使用重复元素的网络（VGG））中我们曾介绍过，VGG网络使用了5个卷积块。**实验中，我们选择第四卷积块的第一个卷积层作为内容层，以及每个卷积块的第一个卷积层作为样式层。**这些层的索引可以通过打印 `pretrained_net` 实例来获取。

```
1 In [6]:  
2     pretrained_net  
3 Out [6]:  
4     VGG(  
5         features): Sequential(  
6             (0): Conv2d(3, 64, kernel_size=(3, 3),  
7                         stride=(1, 1), padding=(1, 1))  
8             (1): ReLU(inplace)  
9             (2): Conv2d(64, 64, kernel_size=(3, 3),  
10                      stride=(1, 1), padding=(1, 1))  
11            (3): ReLU(inplace)  
12            (4): MaxPool2d(kernel_size=2, stride=2,  
13                           padding=0, dilation=1, ceil_mode=False)  
14            (5): Conv2d(64, 128, kernel_size=(3, 3),  
15                           stride=(1, 1), padding=(1, 1))  
16            (6): ReLU(inplace)  
17            (7): Conv2d(128, 128, kernel_size=(3, 3),  
18                           stride=(1, 1), padding=(1, 1))  
19            (8): ReLU(inplace)  
20            (9): MaxPool2d(kernel_size=2, stride=2,  
21                           padding=0, dilation=1, ceil_mode=False)  
22            (10): Conv2d(128, 256, kernel_size=(3, 3),  
23                           stride=(1, 1), padding=(1, 1))  
24            (11): ReLU(inplace)  
25            (12): Conv2d(256, 256, kernel_size=(3, 3),  
26                           stride=(1, 1), padding=(1, 1))  
27            (13): ReLU(inplace)  
28            (14): Conv2d(256, 256, kernel_size=(3, 3),  
29                           stride=(1, 1), padding=(1, 1))  
30            (15): ReLU(inplace)  
31            (16): Conv2d(256, 256, kernel_size=(3, 3),  
32                           stride=(1, 1), padding=(1, 1))  
33            (17): ReLU(inplace)  
34            (18): MaxPool2d(kernel_size=2, stride=2,  
35                           padding=0, dilation=1, ceil_mode=False)  
36            (19): Conv2d(256, 512, kernel_size=(3, 3),
```

```

37         stride=(1, 1), padding=(1, 1))
38     (20): ReLU(inplace)
39     (21): Conv2d(512, 512, kernel_size=(3, 3),
40                  stride=(1, 1), padding=(1, 1))
41     (22): ReLU(inplace)
42     (23): Conv2d(512, 512, kernel_size=(3, 3),
43                  stride=(1, 1), padding=(1, 1))
44     (24): ReLU(inplace)
45     (25): Conv2d(512, 512, kernel_size=(3, 3),
46                  stride=(1, 1), padding=(1, 1))
47     (26): ReLU(inplace)
48     (27): MaxPool2d(kernel_size=2, stride=2,
49                       padding=0, dilation=1, ceil_mode=False)
50     (28): Conv2d(512, 512, kernel_size=(3, 3),
51                  stride=(1, 1), padding=(1, 1))
52     (29): ReLU(inplace)
53     (30): Conv2d(512, 512, kernel_size=(3, 3),
54                  stride=(1, 1), padding=(1, 1))
55     (31): ReLU(inplace)
56     (32): Conv2d(512, 512, kernel_size=(3, 3),
57                  stride=(1, 1), padding=(1, 1))
58     (33): ReLU(inplace)
59     (34): Conv2d(512, 512, kernel_size=(3, 3),
60                  stride=(1, 1), padding=(1, 1))
61     (35): ReLU(inplace)
62     (36): MaxPool2d(kernel_size=2, stride=2,
63                       padding=0, dilation=1, ceil_mode=False)
64   )
65   (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
66   (classifier): Sequential(
67     (0): Linear(in_features=25088, out_features=4096, bias=True)
68     (1): ReLU(inplace)
69     (2): Dropout(p=0.5)
70     (3): Linear(in_features=4096, out_features=4096, bias=True)
71     (4): ReLU(inplace)
72     (5): Dropout(p=0.5)
73     (6): Linear(in_features=4096, out_features=1000, bias=True)
74   )
75 )

```

```

1 In [7]:
2     style_layers, content_layers = [0, 5, 10, 19, 28], [25]

```

在抽取特征时，我们只需要用到VGG从输入层到最靠近输出层的内容层或样式层之间的所有层。下面构建一个新的网络 net，它只保留需要用到的VGG的所有层。我们将使用 net 来抽取特征。

```

1 In [8]:
2     net_list = []
3     for i in range(max(content_layers+style_layers)+1):
4         net_list.append(pretrained_net.features[i])
5     net = torch.nn.Sequential(*net_list)

```

给定输入 `x`，如果简单调用前向计算 `net(x)`，只能获得最后一层的输出。由于我们还需要中间层的输出，因此这里我们逐层计算，并保留内容层和样式层的输出。

```

1 In [9]:
2     def extract_features(x, content_layers, style_layers):
3         contents = []
4         styles = []
5         for i in range(len(net)):
6             x = net[i](x)
7             if i in style_layers:
8                 styles.append(x)
9             if i in content_layers:
10                contents.append(x)
11        return contents, styles

```

下面定义两个函数，其中`get_contents`函数对内容图像抽取内容特征，而`get_styles`函数则对样式图像抽取样式特征。因为在训练时无须改变预训练的VGG的模型参数，所以我们在训练开始之前就提取出内容图像的内容特征，以及样式图像的样式特征。由于合成图像是样式迁移所需迭代的模型参数，我们只能在训练过程中通过调用`extract_features`函数来抽取合成图像的内容特征和样式特征。

```

1 In [10]:
2     def get_contents(image_shape, device):
3         content_X = preprocess(content_img, image_shape).to(device)
4         contents_Y, _ = extract_features(
5             content_X, content_layers, style_layers)
6         return content_X, contents_Y
7     def get_styles(image_shape, device):
8         style_X = preprocess(style_img, image_shape).to(device)
9         _, styles_Y = extract_features(
10            style_X, content_layers, style_layers)
11        return style_X, styles_Y

```

## 8.11.5 定义损失函数

下面我们来描述样式迁移的损失函数。它由内容损失、样式损失和总变差损失3部分组成。

### 1. 内容损失

与线性回归中的损失函数类似，内容损失通过平方误差函数衡量合成图像与内容图像在内容特征上的差异。平方误差函数的两个输入均为`extract_features`函数计算所得到的内容层的输出。

```

1 In [11]:
2     def content_loss(Y_hat, Y):
3         return F.mse_loss(Y_hat, Y)

```

### 2. 样式损失

样式损失也一样通过平方误差函数衡量合成图像与样式图像在样式上的差异。为了表达样式层输出的样式，我们先通过`extract_features`函数计算样式层的输出。假设该输出的样本数为1，通道数为 $c$ ，高和宽分别为 $h$ 和 $w$ ，我们可以把输出变换为 $c$ 行 $hw$ 列的矩阵 $\mathbf{X}$ 。矩阵 $\mathbf{X}$ 可以看作是由 $c$ 个长度为 $hw$ 的向量 $\mathbf{x}_1, \dots, \mathbf{x}_c$ 组成的。其中向量 $\mathbf{x}_i$ 代表了通道 $i$ 上的样式特征。这些向量的**格拉姆矩阵**（Gram matrix） $\mathbf{X}\mathbf{X}^\top \in \mathbb{R}^{c \times c}$ 中 $i$ 行 $j$ 列的元素 $x_{ij}$ 即向量 $\mathbf{x}_i$ 与 $\mathbf{x}_j$ 的内积，它表达了通道 $i$ 和通道 $j$ 上样式特征的相关性。我们用这样的格拉姆矩阵表达样式层输出的样式。需要注意的是，当 $hw$ 的值较大时，格拉姆矩阵中的元素容易出现较大的值。此外，格拉姆矩阵的高和宽皆为通道数 $c$ 。为了让样式损失不受这些值的大小影响，下面定义的`gram`函数将格拉姆矩阵除以了矩阵中元素的个数，即 $chw$ 。

```

1 In [12]:
2     def gram(X):
3         num_channels, n = X.shape[1], X.shape[2]*X.shape[3]
4         X = X.view(num_channels, n)
5         return torch.matmul(X, X.t())/(num_channels*n)

```

自然地，样式损失的平方误差函数的两个格拉姆矩阵输入分别基于合成图像与样式图像的样式层输出。这里假设基于样式图像的格拉姆矩阵 `gram_Y` 已经预先计算好了。

```

1 In [13]:
2     def style_loss(Y_hat, gram_Y):
3         return F.mse_loss(gram(Y_hat), gram_Y)

```

### 3. 总变差损失

有时候，我们学到的合成图像里面有大量高频噪声，即有特别亮或者特别暗的颗粒像素。一种常用的降噪方法是总变差降噪 (total variation denoising)。假设  $x_{i,j}$  表示坐标为  $(i, j)$  的像素值，降低总变差损失

$$\sum_{i,j} |x_{i,j} - x_{i+1,j}| + |x_{i,j} - x_{i,j+1}|$$

能够尽可能使邻近的像素值相似。

```

1 In [14]:
2     def tv_loss(Y_hat):
3         return 0.5*(F.l1_loss(Y_hat[:, :, 1:, :], Y_hat[:, :, :-1, :])+
4                     F.l1_loss(Y_hat[:, :, :, 1:], Y_hat[:, :, :, :-1]))

```

### 4. 损失函数

样式迁移的损失函数即内容损失、样式损失和总变差损失的加权和。通过调节这些权值超参数，我们可以权衡合成图像在保留内容、迁移样式以及降噪三方面的相对重要性。

```

1 In [15]:
2     content_weight, style_weight, tv_weight = 1, 1e3, 10
3     def compute_loss(X, contents_Y_hat, styles_Y_hat,
4                       contents_Y, styles_Y_gram):
5         # 分别计算内容损失、样式损失和总变差损失
6         contents_l = [content_loss(Y_hat, Y)*content_weight
7                       for Y_hat, Y in zip(contents_Y_hat, contents_Y)]
7         styles_l = [style_loss(Y_hat, Y)*style_weight
8                       for Y_hat, Y in zip(styles_Y_hat, styles_Y_gram)]
9         tv_l = tv_loss(X)*tv_weight
10        # 对所有损失求和
11        l = sum(styles_l)+sum(contents_l)+tv_l
12        return contents_l, styles_l, tv_l, l
13

```

## 8.11.6 创建和初始化合成图像

在样式迁移中，合成图像是唯一需要更新的变量。因此，我们可以定义一个简单的模型 `GeneratedImage`，并将合成图像视为模型参数。模型的前向计算只需返回模型参数即可。

```
1 In [16]:  
2     class GeneratedImage(torch.nn.Module):  
3         def __init__(self, img_shape):  
4             super(GeneratedImage, self).__init__()  
5             self.weight = torch.nn.Parameter(  
6                 torch.rand(*img_shape))  
7         def forward(self):  
8             return self.weight
```

下面，我们定义 `get_inits` 函数。该函数创建了合成图像的模型实例，并将其初始化为图像 `x`。样式图像在各个样式层的格拉姆矩阵 `styles_Y_gram` 将在训练前预先计算好。

```
1 In [17]:  
2     def get_inits(x, device, lr, styles_Y):  
3         gen_img = GeneratedImage(x.shape).to(device)  
4         gen_img.weight.data = x.data  
5         optimizer = torch.optim.Adam(gen_img.parameters(), lr=lr)  
6         styles_Y_gram = [gram(Y) for Y in styles_Y]  
7         return gen_img(), styles_Y_gram, optimizer
```

## 8.11.7 训练模型

在训练模型时，我们不断抽取合成图像的内容特征和样式特征，并计算损失函数。

```
1 In [18]:  
2     def train(x, contents_Y, styles_Y, device, lr,  
3                max_epochs, lr_decay_epoch):  
4         print('training on ', device)  
5         x, styles_Y_gram, optimizer = get_inits(  
6             x, device, lr, styles_Y)  
7         scheduler = torch.optim.lr_scheduler.StepLR(  
8             optimizer, lr_decay_epoch, gamma=0.1)  
9         for i in range(max_epochs):  
10            start = time.time()  
11            contents_Y_hat, styles_Y_hat = extract_features(  
12                x, content_layers, style_layers)  
13            contents_l, styles_l, tv_l, l = compute_loss(  
14                x, contents_Y_hat, styles_Y_hat, contents_Y, styles_Y_gram)  
15            optimizer.zero_grad()  
16            l.backward(retain_graph=True)  
17            optimizer.step()  
18            scheduler.step()  
19            if i%50==0 and i!=0:  
20                print('epoch %3d, content loss %.2f, style loss %.2f, '\  
21                      'TV loss %.2f, %.2f sec' % (i, sum(contents_l).item(),  
22                                         sum(styles_l).item(),  
23                                         tv_l.item(),  
24                                         time.time()-start))  
25        return x.detach()
```

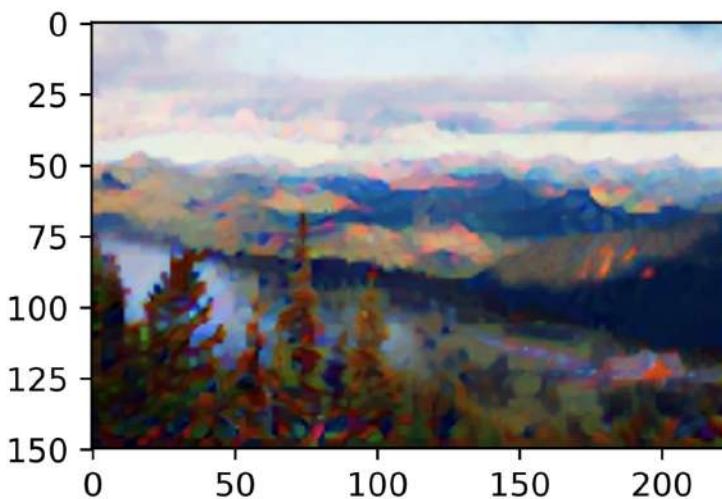
下面我们开始训练模型。首先将内容图像和样式图像的高和宽分别调整为150和225像素。合成图像将由内容图像来初始化。

```
1 In [19]:  
2     image_shape = (150, 225)
```

```
3     net = net.to(device)
4     content_X, contents_Y = get_contents(image_shape, device)
5     style_X, styles_Y = get_styles(image_shape, device)
6     output = train(content_X, contents_Y, styles_Y, device,
7                     0.01, 500, 200)
8 Out [19]:
9     training on cuda
10    epoch 50, content loss 0.24, style loss 1.11, TV loss 1.33, 0.03 sec
11    epoch 100, content loss 0.24, style loss 0.81, TV loss 1.20, 0.03 sec
12    epoch 150, content loss 0.24, style loss 0.72, TV loss 1.12, 0.03 sec
13    epoch 200, content loss 0.23, style loss 0.68, TV loss 1.06, 0.03 sec
14    epoch 250, content loss 0.23, style loss 0.68, TV loss 1.05, 0.03 sec
15    epoch 300, content loss 0.23, style loss 0.67, TV loss 1.04, 0.03 sec
16    epoch 350, content loss 0.23, style loss 0.67, TV loss 1.04, 0.03 sec
17    epoch 400, content loss 0.23, style loss 0.67, TV loss 1.03, 0.03 sec
18    epoch 450, content loss 0.23, style loss 0.67, TV loss 1.03, 0.03 sec
```

下面我们查看一下训练好的合成图像。可以看到下图中的合成图像保留了内容图像的风景和物体，并同时迁移了样式图像的色彩。因为图像尺寸较小，所以细节上依然比较模糊。

```
1 In [20]:
2     d2l.plt.imshow(postprocess(output));
```



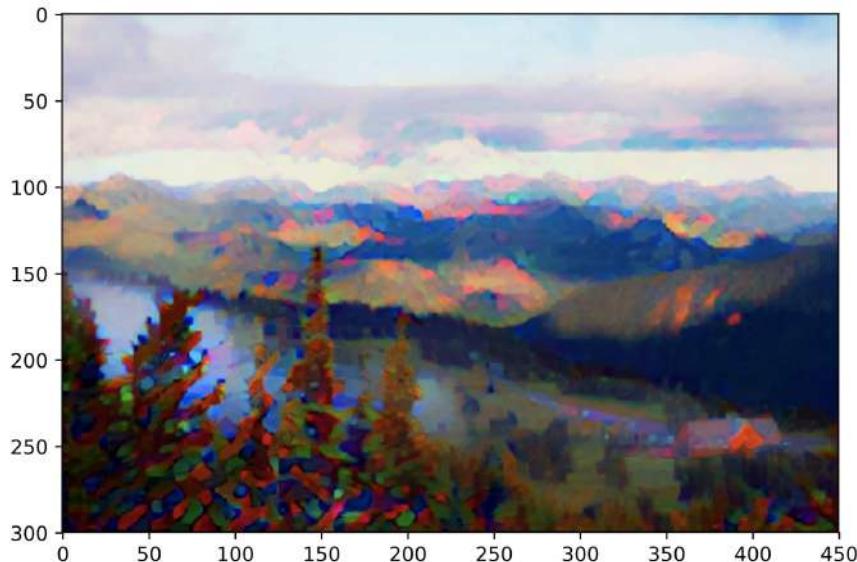
为了得到更加清晰的合成图像，下面我们在更大的 $300 \times 450$ 尺寸上训练。我们将上图的高和宽放大2倍，以初始化更大尺寸的合成图像。

```
1 In [21]:
2     image_shape = (300, 450)
3     _, content_Y = get_contents(image_shape, device)
4     _, style_Y = get_styles(image_shape, device)
5     X = preprocess(postprocess(output), image_shape).to(device)
6     big_output = train(X, content_Y, style_Y, device,
7                         0.01, 500, 200)
8 Out [21]:
9     training on cuda
10    epoch 50, content loss 0.34, style loss 0.63, TV loss 0.79, 0.15 sec
11    epoch 100, content loss 0.30, style loss 0.50, TV loss 0.74, 0.15 sec
12    epoch 150, content loss 0.29, style loss 0.45, TV loss 0.72, 0.14 sec
13    epoch 200, content loss 0.28, style loss 0.43, TV loss 0.70, 0.15 sec
14    epoch 250, content loss 0.28, style loss 0.43, TV loss 0.70, 0.15 sec
15    epoch 300, content loss 0.28, style loss 0.42, TV loss 0.69, 0.14 sec
```

```
16     epoch 350, content loss 0.28, style loss 0.42, TV loss 0.69, 0.14 sec
17     epoch 400, content loss 0.27, style loss 0.42, TV loss 0.69, 0.14 sec
18     epoch 450, content loss 0.27, style loss 0.42, TV loss 0.69, 0.15 sec
```

可以看到，由于图像尺寸更大，每一次迭代需要花费更多的时间。下面我们查看一下训练好的合成图像。

```
1 In [22]:
2     d2l plt.imshow(postprocess(big_output));
```



从训练得到的上图中可以看到，此时的合成图像因为尺寸更大，所以保留了更多的细节。合成图像里面不仅有大块的类似样式图像的油画色彩块，色彩块中甚至出现了细微的纹理。

### 小结：

- 样式迁移常用的损失函数由3部分组成：内容损失使合成图像与内容图像在内容特征上接近，样式损失令合成图像与样式图像在样式特征上接近，而总变差损失则有助于减少合成图像中的噪点。
- 可以通过预训练的卷积神经网络来抽取图像的特征，并通过最小化损失函数来不断更新合成图像。
- 用格拉姆矩阵表达样式层输出的样式。

## 8.12 实战Kaggle比赛：图像分类（CIFAR-10）

到目前为止，我们一直在用PyTorch的 `data` 包直接获取 `Tensor` 格式的图像数据集。然而，实际中的图像数据集往往是以图像文件的形式存在的。在本节中，我们将从原始的图像文件开始，一步步整理、读取并将其变换为 `Tensor` 格式。

我们曾在8.1节中实验过CIFAR-10数据集。它是计算机视觉领域的一个重要数据集。现在我们将应用前面所学的知识，动手实战CIFAR-10图像分类问题的Kaggle比赛。该比赛的网页地址是 <https://www.kaggle.com/c/cifar-10>。

下图展示了该比赛的网页信息。为了便于提交结果，请先在Kaggle网站上注册账号。

The screenshot shows the Kaggle interface for the CIFAR-10 competition. The top navigation bar includes icons for refresh, search, and user profile, followed by the title 'CIFAR-10 - Object Recognition in Images' and a subtitle 'Identify the subject of 60,000 labeled images'. Below this is a section from Kaggle stating 'Kaggle · 231 teams · 6 years ago'. The main menu at the top has tabs for 'Overview', 'Data', 'Notebooks', 'Discussion', 'Leaderboard', 'Rules', and '...'. To the right are links for 'My Submissions' and a prominent 'Late Submission' button. A sidebar on the left contains icons for file operations like copy, paste, and delete. The 'Overview' section is currently selected. It contains two tabs: 'Description' and 'Evaluation'. The 'Description' tab contains text about the CIFAR-10 dataset, mentioning it is a subset of the 80 million tiny images dataset, consists of 60,000 32x32 color images, and contains 10 object classes. It was collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Below the description is a code cell numbered 1 to 11, showing Python imports for collections, math, torch, torchvision, nn, os, pandas, shutil, d2lzh, and setting data\_dir to 'data/kaggle\_cifar10/'.

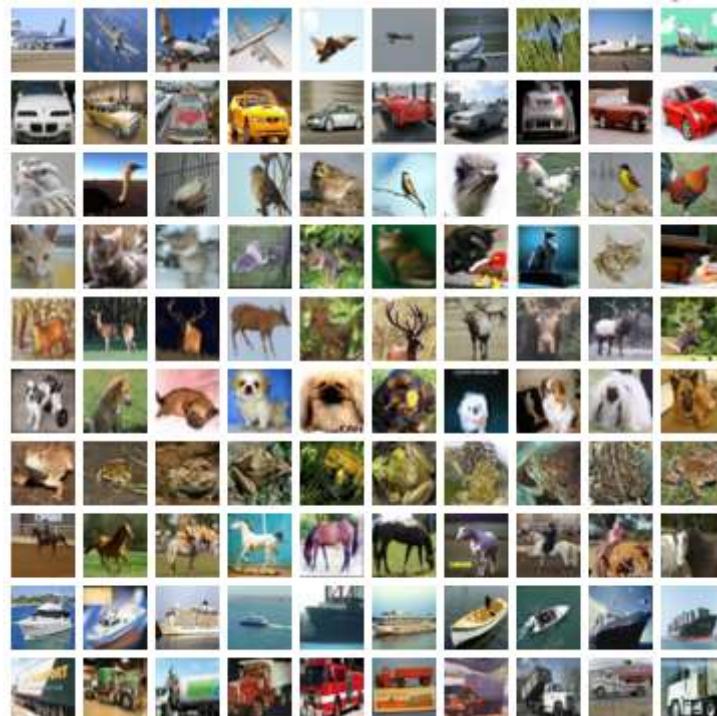
```

1 In [1]:
2 import collections
3 import math
4 import torch
5 import torchvision
6 from torch import nn
7 import os
8 import pandas as pd
9 import shutil
10 import d2lzh as d2l
11 data_dir = 'data/kaggle_cifar10/'

```

## 8.12.1 获取和整理数据集

比赛数据分为训练集和测试集。训练集包含5万张图像。测试集包含30万张图像，其中有1万张图像用来计分，其他29万张不计分的图像是为了防止人工标注测试集并提交标注结果。两个数据集中的图像格式都是png，高和宽均为32像素，并含有RGB三个通道（彩色）。图像一共涵盖10个类别，分别为飞机、汽车、鸟、猫、鹿、狗、青蛙、马、船和卡车。下图展示了数据集中部分飞机、汽车和鸟的图像。



## 1. 下载数据集

登录Kaggle后，可以点击CIFAR-10图像分类比赛网页上的“Data”标签，并分别下载训练数据集train.7z、测试数据集test.7z和训练数据集标签trainLabels.csv。

## 2. 解压数据集

下载完训练数据集train.7z和测试数据集test.7z后需要解压缩。解压缩后，将训练数据集、测试数据集以及训练数据集标签分别存放在以下3个路径：

- data/kaggle\_cifar10/train/[1-50000].png;
- data/kaggle\_cifar10/test/[1-300000].png;
- data/kaggle\_cifar10/trainLabels.csv。

为方便快速上手，我们提供了上述数据集的小规模采样，其中train\_tiny.zip包含100个训练样本，而test\_tiny.zip仅包含1个测试样本。它们解压后的文件夹名称分别为train\_tiny和test\_tiny。此外，将训练数据集标签的压缩文件解压，并得到trainLabels.csv。

## 3. 整理数据集

我们需要整理数据集，以方便训练和测试模型。我们首先从 csv 文件中读取表格，下述函数将返回一个字典，该字典的键为图片编号，值为图片的类别。

```
1 In [2]:  
2     def read_csv_labels(fname):  
3         """  
4             读取文件，返回图片名称到标签之间的映射  
5         """  
6         with open(fname, 'r') as f:  
7             # 跳过表头  
8             lines = f.readlines()[1:]  
9             tokens = [l.rstrip().split(',') for l in lines]  
10            return dict((name, label) for name, label in tokens))  
11    labels = read_csv_labels(  
12        os.path.join(data_dir, 'trainLabels.csv'))  
13    print('# training examples:', len(labels))  
14    print('# classes:', len(set(labels.values())))  
15 Out [2]:  
16    # training examples: 50000  
17    # classes: 10
```

我们接下来定义 reorg\_train\_valid 函数来从原始训练集中切分出验证集。该函数中的参数 valid\_ratio 是验证集样本数与原始训练集样本数之比。以 valid\_ratio=0.1 为例，由于原始训练集有50,000张图像，调参时将有45,000张图像用于训练并存放在路径 train\_valid\_test/train 下，而另外5,000张图像将作为验证集并存放在路径 train\_valid\_test/valid 下。经过整理后，同一类图像将被放在同一个文件夹下，便于稍后读取。

```
1 In [3]:  
2     def copyfile(filename, target_dir):  
3         """  
4             将文件复制到目标路径下  
5         """  
6         # 在路径不存在的情况下创建路径  
7         os.makedirs(target_dir, exist_ok=True)  
8         shutil.copy(filename, target_dir)  
9     def reorg_train_valid(data_dir, labels, valid_ratio):  
10        # 训练集中样本数最少的类别包含的样本数
```

```

11     n = collections.Counter(labels.values()).most_common()[-1][1]
12     # 验证集中每一类的样本数
13     n_valid_per_label = max(1, math.floor(n*valid_ratio))
14     label_count = {}
15     for train_file in os.listdir(
16         os.path.join(data_dir, 'train', 'train')):
17         # 为每个训练集样本匹配类别
18         label = labels[train_file.split('.')[0]]
19         fname = os.path.join(data_dir, 'train', 'train', train_file)
20         # 以类别名为文件夹名称保存数据
21         # 训练集与验证集合并
22         copyfile(fname, os.path.join(
23             data_dir, 'train_valid_test', 'train_valid', label))
24         if label not in label_count or \
25             label_count[label] < n_valid_per_label:
26             # 仅包含验证集
27             copyfile(fname, os.path.join(
28                 data_dir, 'train_valid_test', 'valid', label))
29             label_count[label] = label_count.get(label, 0) + 1
30         else:
31             # 仅包含训练集
32             copyfile(fname, os.path.join(
33                 data_dir, 'train_valid_test', 'train', label))
34     return n_valid_per_label

```

下面的 `reorg_test` 函数用来整理测试集，从而方便预测时的读取。

```

1 In [4]:
2     def reorg_test(data_dir):
3         # 测试集数据整理，类别名为unknown
4         for test_file in os.listdir(os.path.join(data_dir, 'test', 'test')):
5             copyfile(
6                 os.path.join(data_dir, 'test', 'test', test_file),
7                 os.path.join(data_dir, 'train_valid_test',
8                             'test', 'unknown'))

```

最后，我们用一个函数分别调用前面定义的 `read_csv_labels` 函数、`reorg_train_valid` 函数以及 `reorg_test` 函数。

```

1 In [5]:
2     def reorg_cifar10_data(data_dir, valid_ratio):
3         labels = read_csv_labels(
4             os.path.join(data_dir, 'trainLabels.csv'))
5         reorg_train_valid(data_dir, labels, valid_ratio)
6         reorg_test(data_dir)

```

实际训练和测试时应使用Kaggle比赛的完整数据集，并将批量大小 `batch_size` 设为一个较大的整数，如128。我们将10%的训练样本作为调参使用的验证集。

```

1 In [6]:
2     batch_size = 128
3     valid_ratio = 0.1
4     reorg_cifar10_data(data_dir, valid_ratio)

```

## 8.12.2 图像增广

为应对过拟合，我们使用图像增广。例如，加入`transforms.RandomFlipLeftRight()`即可随机对图像做镜面翻转，也可以通过`transforms.Normalize()`对彩色图像RGB三个通道分别做标准化。下面列举了其中的部分操作，你可以根据需求来决定是否使用或修改这些操作。

```
1 In [7]:  
2     transform_train = torchvision.transforms.Compose([  
3         # 将图像放大成高和宽各为40像素的正方形  
4         torchvision.transforms.Resize(40),  
5         # 随机对高和宽各为40像素的正方形图像裁剪出面积为原图像面积0.64~1倍的小正方形  
6         # 再放缩为高和宽各为32像素的正方形  
7         torchvision.transforms.RandomResizedCrop(  
8             32, scale=(0.64, 1.0), ratio=(1.0, 1.0)),  
9         torchvision.transforms.RandomHorizontalFlip(),  
10        torchvision.transforms.ToTensor(),  
11        # 对图像的每个通道做标准化  
12        torchvision.transforms.Normalize([0.4914, 0.4822, 0.4465],  
13                                         [0.2023, 0.1994, 0.2010])  
14    ])
```

测试时，为保证输出的确定性，我们仅对图像做标准化。

```
1 In [8]:  
2     transform_test = torchvision.transforms.Compose([  
3         torchvision.transforms.ToTensor(),  
4         torchvision.transforms.Normalize([0.4914, 0.4822, 0.4465],  
5                                         [0.2023, 0.1994, 0.2010])  
6     ])
```

## 8.12.3 读取数据集

接下来，可以通过创建`ImageFolder`实例来读取整理后的含原始图像文件的数据集，其中每个数据样本包括图像和标签。

```
1 In [9]:  
2     train_ds, train_valid_ds = [torchvision.datasets.ImageFolder(  
3         os.path.join(data_dir, 'train_valid_test', folder),  
4         transform=transform_train) for folder in ['train', 'train_valid']  
5     ]  
6     valid_ds, test_ds = [torchvision.datasets.ImageFolder(  
7         os.path.join(data_dir, 'train_valid_test', folder),  
8         transform=transform_test) for folder in ['valid', 'test']]  
9 ]
```

我们在`DataLoader`中指明定义好的图像增广操作。在训练时，我们仅用验证集评价模型，因此需要保证输出的确定性。在预测时，我们将在训练集和验证集的并集上训练模型，以充分利用所有标注的数据。

```
1 In [10]:  
2     train_iter, train_valid_iter = [  
3         torch.utils.data.DataLoader(  
4             dataset, batch_size, shuffle=True, drop_last=True)  
5             for dataset in (train_ds, train_valid_ds)  
6         ]  
7     valid_iter = torch.utils.data.DataLoader(  
8         valid_ds, batch_size, shuffle=False, drop_last=True)  
9     test_iter = torch.utils.data.DataLoader(  
10        test_ds, batch_size, shuffle=False, drop_last=False)
```

## 8.12.4 定义模型

我们使用 ResNet-18 模型对数据样本进行训练，CIFAR-10图像分类问题的类别个数为10。

```
1 In [11]:  
2     def get_net():  
3         num_classes = 10  
4         net = d2l.resnet18(num_classes, 3)  
5         return net  
6     net = get_net()  
7     loss = nn.CrossEntropyLoss()
```

## 8.12.5 定义训练函数

我们将根据模型在验证集上的表现来选择模型并调节超参数。这里我们沿用8.1.2节定义的训练函数 `train`。我们记录了每个迭代周期的训练时间，这有助于比较不同模型的时间开销。

## 8.12.6 训练模型

现在，我们可以训练并验证模型了。下面的超参数都是可以调节的，如增加迭代周期等。简单起见，这里仅训练5个迭代周期。

```
1 In [12]:  
2     device, num_epochs, lr, wd = 'cuda', 5, 0.1, 5e-4  
3     optimizer = torch.optim.SGD(  
4         net.parameters(), lr=lr, momentum=0.9, weight_decay=wd)  
5     d2l.train(train_iter, valid_iter, net, loss,  
6               optimizer, device, num_epochs)  
7 Out [12]:  
8     Let's use 2 GPUs!  
9     training on cuda  
10    epoch 1, loss 1.7174, train acc 0.378, test acc 0.463, time 74.0 sec  
11    epoch 2, loss 0.6811, train acc 0.509, test acc 0.564, time 67.1 sec  
12    epoch 3, loss 0.3951, train acc 0.580, test acc 0.588, time 67.5 sec  
13    epoch 4, loss 0.2688, train acc 0.623, test acc 0.616, time 68.1 sec  
14    epoch 5, loss 0.2009, train acc 0.649, test acc 0.642, time 67.5 sec
```

## 8.12.7 对测试集分类并在Kaggle提交结果

得到一组满意的模型设计和超参数后，我们对测试集进行分类。

```

1 In [13]:
2     preds = []
3     for x, _ in test_iter:
4         y_hat = net(x.to(device))
5         preds.extend(y_hat.argmax(dim=1).type(torch.int32).cpu().numpy())
6     sorted_ids = list(range(1, len(test_ds)+1))
7     sorted_ids.sort(key=lambda x: str(x))
8     df = pd.DataFrame({'id': sorted_ids, 'label': preds})
9     df['label'] = df['label'].apply(lambda x: train_valid_ds.classes[x])
10    df.to_csv('submission.csv', index=False)

```

执行完上述代码后，我们会得到一个submission.csv文件。这个文件符合Kaggle比赛要求的提交格式。提交结果的方法与2.16节中的类似。

### 小结：

- 可以通过创建 `ImageFolder` 实例来读取含原始图像文件的数据集。
- 可以应用卷积神经网络、图像增广来实战图像分类比赛。

## 8.13 实战Kaggle比赛：狗的品种识别（ImageNet Dogs）

我们将在本节动手实战Kaggle比赛中的狗的品种识别问题。该比赛的网页地址是 <https://www.kaggle.com/c/dog-breed-identification>。

在这个比赛中，将识别120类不同品种的狗。这个比赛的数据集实际上是著名的ImageNet的子集数据集。和上一节的CIFAR-10数据集中的图像不同，ImageNet数据集中的图像更高更宽，且尺寸不一。

下图展示了该比赛的网页信息。为了便于提交结果，请先在Kaggle网站上注册账号。

The screenshot shows the Kaggle interface for the 'Dog Breed Identification' competition. On the left, there's a sidebar with various icons. The main content area has a large header image of a dog's head. Below it, the title 'Dog Breed Identification' and subtitle 'Determine the breed of a dog in an image' are displayed. A 'Kaggle · 1,282 teams · 3 years ago' badge is present. The navigation bar includes 'Overview', 'Data', 'Notebooks', 'Discussion', 'Leaderboard', 'Rules', and 'Datasets'. A prominent 'Late Submission' button is located at the bottom right of the main content area. The 'Overview' section contains a 'Description' tab with text about identifying dog breeds and a 'Evaluation' tab with text about the competition rules. At the bottom, there are five small images of different dog breeds.

首先，导入比赛所需的包或模块。

```
1 In [1]:  
2     import d2lzh as d2l  
3     import torch  
4     import torchvision  
5     from torch import nn  
6     import os  
7     import numpy as np  
8     data_dir = 'data/kaggle_dogs/'  
9     device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

## 8.13.1 获取和整理数据集

比赛数据分为训练集和测试集。训练集包含了10,222张图像，测试集包含了10,357张图像。两个数据集中的图像格式都是JPEG。这些图像都含有RGB三个通道（彩色），高和宽的大小不一。训练集中狗的类别共有120种，如拉布拉多、贵宾、腊肠、萨摩耶、哈士奇、吉娃娃和约克夏等。

### 1. 下载数据集

登录Kaggle后，我们可以点击上图所示的狗的品种识别比赛网页上的“Data”标签，并分别下载训练数据集train.zip、测试数据集test.zip和训练数据集标签label.csv.zip。下载完成后，将它们分别存放在以下3个路径：

- data/kaggle\_dogs/train.zip；
- data/kaggle\_dogs/test.zip；
- data/kaggle\_dogs/labels.csv.zip。

### 2. 整理数据集

我们利用上一章定义的 reorg\_train\_valid 函数来从Kaggle比赛的完整原始训练集中切分出验证集。该函数中的参数 valid\_ratio 指验证集中每类狗的样本数与原始训练集中数量最少一类的狗的样本数（66）之比。经过整理后，同一类狗的图像将被放在同一个文件夹下，便于稍后读取。下面的 reorg\_dog\_data 函数用来读取训练数据标签、切分验证集并整理测试集。

```
1 In [2]:  
2     def reorg_dog_data(data_dir, valid_ratio):  
3         # 读取训练数据标签  
4         labels = d2l.read_csv_labels(os.path.join(data_dir, 'labels.csv'))  
5         # 整理训练数据集  
6         d2l.reorg_train_valid(data_dir, labels, valid_ratio)  
7         # 整理测试数据集  
8         d2l.reorg_test(data_dir)
```

在实际训练和测试时，我们应使用Kaggle比赛的完整数据集并调用 reorg\_dog\_data 函数来整理数据集。相应地，我们也需要将批量大小 batch\_size 设为一个较大的整数，如128。

```
1 In [3]:  
2     batch_size = 128  
3     valid_ratio = 0.1  
4     reorg_dog_data(data_dir, valid_ratio)
```

## 8.13.2 图像增广

本节比赛的图像尺寸比上一节中的更大。这里列举了更多可能有用的图像增广操作。

```
1 In [4]:  
2     transform_train = torchvision.transforms.Compose([  
3         # 随机对图像裁剪出面积为原图像面积0.08~1倍、  
4         # 且高和宽之比在3/4~4/3的图像  
5         # 再放缩为高和宽均为224像素的新图像  
6         torchvision.transforms.RandomResizedCrop(  
7             224, scale=(0.08, 1.0), ratio=(3.0/4.0, 4.0/3.0)),  
8         torchvision.transforms.RandomHorizontalFlip(),  
9         # 随机变化亮度、对比度和饱和度  
10        torchvision.transforms.ColorJitter(  
11            brightness=0.4, contrast=0.4, saturation=0.4),  
12        torchvision.transforms.ToTensor(),  
13        # 对图像的每个通道做标准化  
14        torchvision.transforms.Normalize(  
15            [0.485, 0.456, 0.406], [0.229, 0.224, 0.225])  
16    ])
```

测试时，我们只使用确定性的图像预处理操作。

```
1 In [5]:  
2     transform_test = torchvision.transforms.Compose([  
3         torchvision.transforms.Resize(256),  
4         # 将图像中央的高和宽均为224的正方形区域裁剪出来  
5         torchvision.transforms.CenterCrop(224),  
6         torchvision.transforms.ToTensor(),  
7         torchvision.transforms.Normalize(  
8             [0.485, 0.456, 0.406], [0.229, 0.224, 0.225])  
9     ])
```

## 8.13.3 读取数据集

和上一节一样，我们创建 `ImageFolder` 实例来读取整理后的含原始图像文件的数据集。

```
1 In [6]:  
2     train_ds, train_valid_ds = [torchvision.datasets.ImageFolder(  
3         os.path.join(data_dir, 'train_valid_test', folder),  
4         transform=transform_train) for folder in ['train', 'train_valid']]  
5     ]  
6     valid_ds, test_ds = [torchvision.datasets.ImageFolder(  
7         os.path.join(data_dir, 'train_valid_test', folder),  
8         transform=transform_test) for folder in ['valid', 'test']]  
9     ]
```

这里创建 `DataLoader` 实例的方法也与上一节中的相同。

```
1 In [7]:  
2     train_iter, train_valid_iter = [  
3         torch.utils.data.DataLoader(dataset, batch_size,  
4                                         shuffle=True, drop_last=True)  
5         for dataset in (train_ds, train_valid_ds)  
6     ]  
7     valid_iter = torch.utils.data.DataLoader(  
8         valid_ds, batch_size, shuffle=False, drop_last=True)  
9     test_iter = torch.utils.data.DataLoader(  
10        test_ds, batch_size, shuffle=False, drop_last=False)
```

## 8.13.4 定义模型

这个比赛的数据集属于ImageNet数据集的子集，因此我们可以使用8.2节中介绍的思路，选用在ImageNet完整数据集上预训练的模型来抽取图像特征，以作为自定义小规模输出网络的输入。torchvision提供了丰富的预训练模型，这里以预训练的ResNet-34模型为例。由于比赛数据集属于预训练数据集的子集，因此我们直接复用预训练模型在输出层的输入，即抽取的特征。然后，我们可以将原输出层替换成自定义的可训练的小规模输出网络，如两个串联的全连接层。与8.2节中的实验不同，**这里不再训练用于抽取特征的预训练模型**：这样既节省了训练时间，又省去了存储其模型参数的梯度的空间。

需要注意的是，我们在图像增广中使用了ImageNet数据集上RGB三个通道的均值和标准差做标准化，这和预训练模型所做的标准化是一致的。

```
1 In [8]:  
2     def get_net(devices):  
3         finetune_net = nn.Sequential()  
4         finetune_net.features = torchvision.models.resnet34(  
5             pretrained=True)  
6         # 定义新的输出网络  
7         # 120是输出的类别个数  
8         finetune_net.output_new = nn.Sequential(  
9             nn.Linear(1000, 256),  
10            nn.ReLU(),  
11            nn.Linear(256, 120))  
12         # 把模型参数分配到内存或显存上  
13         finetune_net = finetune_net.to(devices)  
14         for param in finetune_net.features.parameters():  
15             param.requires_grad = False  
16         return finetune_net
```

在计算损失时，我们先通过成员变量 `features` 来获取预训练模型输出层的输入，即抽取的特征。然后，将该特征作为自定义的小规模输出网络的输入，并计算输出。

```
1 In [9]:  
2     loss = nn.CrossEntropyLoss()  
3     def evaluate_loss(data_iter, net, devices):  
4         l_sum, n = 0.0, 0  
5         for features, labels in data_iter:  
6             features, labels = features.to(devices), labels.to(devices)  
7             outputs = net(features)  
8             l = loss(outputs, labels)  
9             l_sum += l.sum()  
10            n += labels.numel()  
11        return l_sum/n
```

## 8.13.5 定义训练函数

我们将根据模型在验证集上的表现来选择模型并调节超参数。这里我们沿用8.1.2节定义的训练函数 `train`。我们记录了每个迭代周期的训练时间，这有助于比较不同模型的时间开销。

## 8.13.6 训练模型

我们将依赖模型在验证集上的表现来选择模型并调节超参数。

```
1 In [10]:  
2     devices, num_epochs, lr, wd = device, 5, 0.01, 1e-4  
3     net = get_net(devices)  
4     optimizer = torch.optim.SGD(  
5         net.parameters(), lr=lr, momentum=0.9, weight_decay=wd)  
6     d2l.train(train_iter, valid_iter, net, loss, optimizer,  
7             device, num_epochs)  
8 Out [10]:  
9     Let's use 2 GPUs!  
10    training on cuda  
11    epoch 1, loss 2.8775, train acc 0.351, test acc 0.680, time 227.9 sec  
12    epoch 2, loss 0.7222, train acc 0.605, test acc 0.727, time 218.0 sec  
13    epoch 3, loss 0.4139, train acc 0.657, test acc 0.769, time 208.6 sec  
14    epoch 4, loss 0.2969, train acc 0.668, test acc 0.787, time 218.9 sec  
15    epoch 5, loss 0.2304, train acc 0.679, test acc 0.783, time 210.7 sec
```

## 8.13.7 对测试集分类并在Kaggle提交结果

得到一组满意的模型设计和超参数后，我们使用训练好的模型对测试集分类。注意，我们要用刚训练好的输出网络做预测。

```
1 In [11]:  
2     preds = []  
3     for data, label in test_iter:  
4         output = torch.nn.functional.softmax(  
5             net(data.to(device)), dim=0)  
6         preds.extend(output.cpu().detach().numpy())  
7     ids = sorted(os.listdir(  
8         os.path.join(data_dir, 'train_valid_test', 'test', 'unknown'))  
9     ))  
10    with open('submission.csv', 'w') as f:  
11        f.write('id,' + ','.join(train_valid_ds.classes) + '\n')  
12        for i, output in zip(ids, preds):  
13            f.write(i.split('.')[0] + ',' + ','.join(  
14                [str(num) for num in output]) + '\n'  
15            )
```

执行完上述代码后，会生成一个`submission.csv`文件。这个文件符合Kaggle比赛要求的提交格式。提交结果的方法与2.16节中的类似。

**小结：**

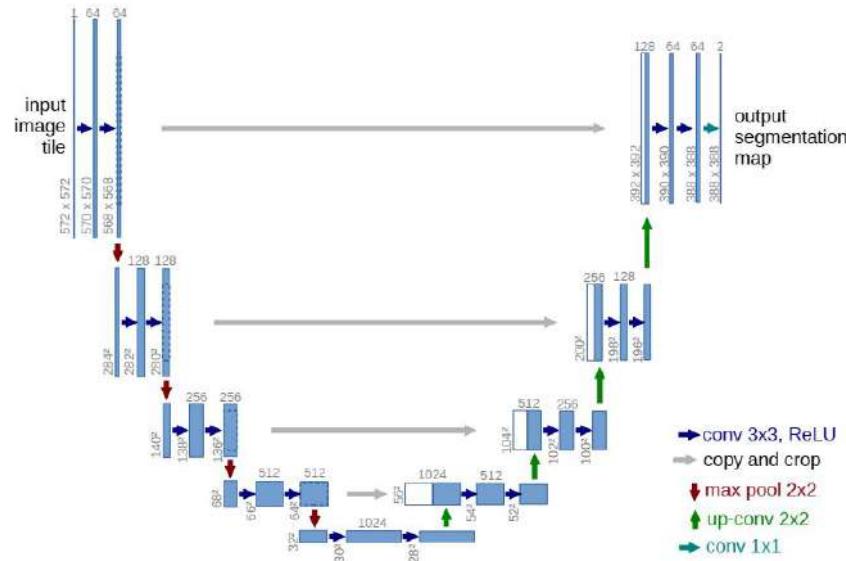
- 我们可以使用在ImageNet数据集上预训练的模型抽取特征，并仅训练自定义的小规模输出网络，从而以较小的计算和存储开销对ImageNet的子集数据集做分类。

## 8.14 语义分割网络（U-Net）

我们来介绍一个经典的语义分割网络U-net, 它于2015年提出, 最初应用在医疗影像分割任务上, 由于效果很好, 之后被广泛应用在各种分割任务中。至今已衍生出许多基于U-net的分割模型。U-net是典型的Encoder-Decoder结构, encoder进行特征提取, decoder进行上采样。由于数据的限制, U-net在训练阶段使用了大量的数据增强操作, 最后得到了不错的效果。

## 8.14.1 网络结构

U-net的网络结构如下所示。左边为encoder部分, 对输入进行下采样, 下采样通过最大池化实现; 右边为decoder部分, 对encoder的输出进行上采样, 恢复分辨率, 上采样通过Upsample实现; 中间为跳跃连接 (Skip-connect) , 进行特征融合。由于整个网络形似一个"U", 所以称为U-net。网络中除了最后的输出层, 其余所有卷积层均为 $3 \times 3$ 卷积。U-net结构简单稳定, 是典型的下采样+上采样的分割网络结构。尤其在数据集较小的时候, 推荐使用。

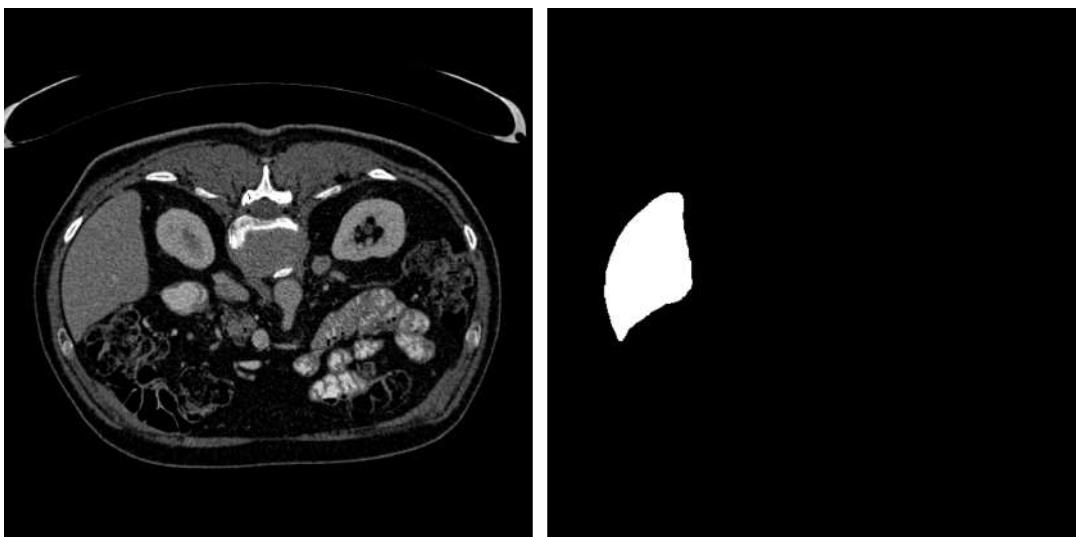


首先导入实验所需的包或模块。

```
In [1]:  
1 import numpy as np  
2 import torch  
3 from torch.utils.data import DataLoader  
4 from torch import autograd, optim  
5 from torchvision.transforms import transforms  
6 import torch.nn as nn  
7 import torch.utils.data as data  
8 import PIL.Image as Image  
9 import os  
10 import matplotlib.pyplot as plt  
11 device = torch.device(  
12     'cuda:1' if torch.cuda.is_available() else 'cpu')
```

## 8.14.2 读取数据集

本文用到的数据集为肝脏图, 该数据集对每一个肝脏医学影像图片中的肝脏进行了标注, 原图 (命名方式为000.png) 及标注结果 (命名方式为000\_mask.png) 如下图所示:



```
1 In [2]:  
2     # 读取数据的路径  
3     def make_dataset(root):  
4         imgs = []  
5         # 计算共有多少张原始图片  
6         n = len(os.listdir(root))//2  
7         for i in range(n):  
8             # 找到00i.png的路径  
9             img = os.path.join(root, '%03d.png'%i)  
10            # 找到00i_mask.png的路径  
11            mask = os.path.join(root, '%03d_mask.png'%i)  
12            # 添加至列表  
13            imgs.append((img, mask))  
14        return imgs
```

在数据增强方面，我们对原始图片转为 `Tensor` 格式并对每一个通道进行标准化处理，对于标签图片我们仅将其转为 `Tensor` 格式。

```
1 In [3]:  
2     x_transforms = transforms.Compose([  
3         transforms.ToTensor(),  
4         transforms.Normalize([0.5, 0.5, 0.5],  
5                             [0.5, 0.5, 0.5])  
6     ])  
7     # mask只需转为Tensor  
8     y_transforms = transforms.ToTensor()
```

为了方便于对图片进行预处理，我们定义 `LiverDataset` 类以读取图片及其标签图片，并对它们进行相应的预处理。

```
1 In [4]:  
2     class LiverDataset(data.Dataset):  
3         def __init__(self, root, transform=None,  
4                      target_transform=None):  
5             imgs = make_dataset(root)  
6             self.imgs = imgs  
7             self.transform = transform  
8             self.target_transform = target_transform  
9         def __getitem__(self, index):  
10             x_path, y_path = self.imgs[index]
```

```

11         img_x = Image.open(x_path)
12         img_y = Image.open(y_path)
13         if self.transform is not None:
14             img_x = self.transform(img_x)
15         if self.target_transform is not None:
16             img_y = self.target_transform(img_y)
17         return img_x, img_y
18     def __len__(self):
19         return len(self.imgs)
20     batch_size = 1
21     liver_dataset = LiverDataset('data/liver/train',
22                                 transform=x_transforms,
23                                 target_transform=y_transforms)
24     dataloaders = DataLoader(liver_dataset, batch_size=batch_size,
25                             shuffle=True, num_workers=4)

```

### 8.14.3 定义模型

我们按照本章所示网络结构对模型进行定义。

```

1 # U_Net模型中的双卷积网络结构
2 class DoubleConv(nn.Module):
3     def __init__(self, in_ch, out_ch):
4         super(DoubleConv, self).__init__()
5         self.conv = nn.Sequential(
6             # 此处包含padding, 为了使输出图像与输入图像大小相同
7             nn.Conv2d(in_ch, out_ch, kernel_size=3, padding=1),
8             nn.BatchNorm2d(out_ch),
9             nn.ReLU(inplace=True),
10            nn.Conv2d(out_ch, out_ch, kernel_size=3, padding=1),
11            nn.BatchNorm2d(out_ch),
12            nn.ReLU(inplace=True)
13        )
14     def forward(self, input):
15         return self.conv(input)
16 class Unet(nn.Module):
17     def __init__(self, in_ch, out_ch):
18         super(Unet, self).__init__()
19         # 特征图大小不变
20         self.conv1 = DoubleConv(in_ch, 64)
21         # 特征图大小长宽减半
22         self.pool1 = nn.MaxPool2d(2)
23         self.conv2 = DoubleConv(64, 128)
24         self.pool2 = nn.MaxPool2d(2)
25         self.conv3 = DoubleConv(128, 256)
26         self.pool3 = nn.MaxPool2d(2)
27         self.conv4 = DoubleConv(256, 512)
28         self.pool4 = nn.MaxPool2d(2)
29         self.conv5 = DoubleConv(512, 1024)
30         # 长宽翻倍, 通道数减半
31         self.up6 = nn.ConvTranspose2d(1024, 512, 2, stride=2)
32         self.conv6 = DoubleConv(1024, 512)
33         self.up7 = nn.ConvTranspose2d(512, 256, 2, stride=2)
34         self.conv7 = DoubleConv(512, 256)
35         self.up8 = nn.ConvTranspose2d(256, 128, 2, stride=2)
36         self.conv8 = DoubleConv(256, 128)
37         self.up9 = nn.ConvTranspose2d(128, 64, 2, stride=2)

```

```

38     self.conv9 = DoubleConv(128, 64)
39     self.conv10 = nn.Conv2d(64, out_ch, 1)
40
41     def forward(self, x):
42         c1 = self.conv1(x)
43         p1 = self.pool1(c1)
44         c2 = self.conv2(p1)
45         p2 = self.pool2(c2)
46         c3 = self.conv3(p2)
47         p3 = self.pool3(c3)
48         c4 = self.conv4(p3)
49         p4 = self.pool4(c4)
50         c5 = self.conv5(p4)
51         up_6 = self.up6(c5)
52         # 通道维拼接
53         merge6 = torch.cat([up_6, c4], dim=1)
54         c6 = self.conv6(merge6)
55         up_7 = self.up7(c6)
56         merge7 = torch.cat([up_7, c3], dim=1)
57         c7 = self.conv7(merge7)
58         up_8 = self.up8(c7)
59         merge8 = torch.cat([up_8, c2], dim=1)
60         c8 = self.conv8(merge8)
61         up_9 = self.up9(c8)
62         merge9 = torch.cat([up_9, c1], dim=1)
63         c9 = self.conv9(merge9)
64         c10 = self.conv10(c9)
65         out = nn.Sigmoid()(c10)
66         return out

```

## 8.14.4 定义训练函数

我们这里用到的损失函数为 `BCELoss`，该损失函数用于图片多标签分类，其具体计算方法见[博客](#)。

```

1 In [5]:
2 # 输入图像有3个通道, 标签图像有1个通道
3 net = Unet(3, 1).to(device)
4 loss = torch.nn.BCELoss()
5 optimizer = optim.Adam(net.parameters())
6 def train_model(model, loss, optimizer, dataloaders, num_epochs=20):
7     for epoch in range(num_epochs):
8         print('Epoch {}/{}'.format(epoch, num_epochs-1))
9         print('-'*10)
10        dt_size = len(dataloaders.dataset)
11        epoch_loss = 0
12        step = 0
13        for x, y in dataloaders:
14            step += 1
15            inputs = x.to(device)
16            labels = y.to(device)
17            optimizer.zero_grad()
18            outputs = model(inputs)
19            l = loss(outputs, labels)
20            l.backward()
21            optimizer.step()
22            epoch_loss += l.item()
23            if step % 200 == 0:

```

```

24         print('%d/%d, train_loss:%0.3f' %
25             (step, (dt_size-1)//dataloaders.batch_size+1,
26              1.item())))
27     print('epoch %d loss:%0.3f' % (epoch, epoch_loss))
28     return model

```

## 8.14.5 训练模型

```

1 In [6]:
2     model = train_model(net, loss, optimizer, dataloaders)
3 Out [6]:
4     Epoch 18/19
5     -----
6     200/400, train_loss:0.005
7     400/400, train_loss:0.002
8     epoch 18 loss:1.811
9     Epoch 19/19
10    -----
11    200/400, train_loss:0.002
12    400/400, train_loss:0.003
13    epoch 19 loss:1.715

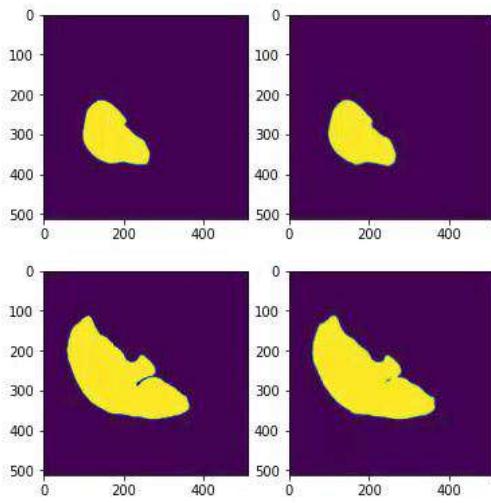
```

## 8.14.6 预测像素类别

```

1 In [7]:
2     liver_val = LiverDataset('data/liver/val', transform=x_transforms,
3                             target_transform=y_transforms)
4     liver_val = DataLoader(liver_val, batch_size=1)
5     model.eval()
6     with torch.no_grad():
7         for i, data in enumerate(dataloaders):
8             # 左边真实, 右边预测
9             x, z = data
10            y = model(x.to(device))
11            img_y = torch.squeeze(y.cpu()).numpy()
12            plt.subplot(1, 2, 1)
13            z = torch.squeeze(z).numpy()
14            plt.imshow(z)
15            plt.axis('on')
16            plt.subplot(1, 2, 2)
17            plt.imshow(img_y)
18            plt.axis('on')
19            plt.pause(0.01)
20            filename = 'data/liver/predict/' + 'new_%d.jpg'%i
21            Image.fromarray(
22                (img_y*255).astype('uint8')).convert('L').save(filename)

```



### 小结：

- U-net结构简单稳定，是典型的下采样+上采样的分割网络结构。尤其在数据集较小的时候，推荐使用。

## 8.15 本章附录

### ☆ `torchvision.transforms` 模块

- `torchvision.transforms.RandomHorizontalFlip()`：随机水平翻转给定的 `PIL.Image`，概率为0.5。即：一半的概率翻转，一半的概率不翻转。
- `torchvision.transforms.RandomVerticalFlip()`：以0.5概率竖直翻转给定的 `PIL` 图像。
- `torchvision.transforms.RandomResizedCrop()`：将给定图像随机裁剪为不同的大小和宽高比，然后缩放所裁剪得到的图像为指定的大小（有一个参数n）。
- `torchvision.transforms.ColorJitter()`：修改图像的亮度，对比度，饱和度和色度。
- `torchvision.transforms.Compose()`：串联多个图片变换的操作。
- `torchvision.transforms.ToTensor()`：将一个PIL图像转换为tensor。即， $(H \times W \times C)$ 范围在[0,255]的PIL图像转换为 $(C \times H \times W)$ 范围在[0,1]的`torch.tensor`。
- `torchvision.transforms.Normalize(mean, std)`：给定均值：`(R, G, B)` 方差：`(R, G, B)`，将会把 `Tensor` 正则化。即：`Normalized_image = (image - mean) / std`。
- `torchvision.transforms.CenterCrop(size)`：将给定的 `PIL.Image` 进行中心切割，得到给定的 `size`，`size` 可以是 `tuple`，`(target_height, target_width)`。`size` 也可以是一个 `Integer`，在这种情况下，切出来的图片的形状是正方形。
- `torchvision.transforms.Resize()`：把给定的图片resize到给定的尺寸。
- `torchvision.transforms.RandomCrop()`：以输入图的随机位置为中心做指定size的裁剪操作。
- `torchvision.transforms.ToPILImage()`：将`torch.tensor` 转换为PIL图像。

### ☆ `ImageFolder()` 的用法

`ImageFolder`假设所有的文件按文件夹保存，**每个文件夹下存储同一个类别的图片，文件夹名为类名**，其构造函数如下：`ImageFolder(root, transform=None, target_transform=None, loader=default_loader)`。`root`: 在`root`指定的路径下寻找图片、`transform`: 对`PIL Image`进行的转换操作，`transform`的输入是使用`loader`读取图片的返回对象、`target_transform`: 对label的转换、`loader`: 给定路径后如何读取图片，默认读取为RGB格式的`PIL Image`对象。`label`是按照文件夹名顺序排序后存成字典，即{类名:类序号(从0开始)}，一般来说最好直接将文件夹命名为从0开始的数字，这样会和`ImageFolder`实际的label一致，如果不是这种命名规范，建议看看`self.class_to_idx`属性以了解`label`和文件夹名的映射关系。

### ☆ `id()` 的用法

`id()` 函数返回对象的唯一标识符（对象的内存地址），标识符是一个整数。

```
1 | a = 'a'
2 | id(a)
3 | >>> 2486531128712
```

## ☆ torch.clamp() 的用法

`torch.clamp(input, min, max, out=None) → Tensor`。将输入 `input` 张量每个元素的范围限制到区间  $[min, max]$ ，返回结果到一个新张量，其计算方式如下：

$$y_i = \begin{cases} \min & \text{if } x_i < \min \\ x_i & \text{if } \min \leq x_i \leq \max \\ \max & \text{if } x_i > \max \end{cases}$$

## ☆ BCELoss() 的计算方法

在图片多标签分类时，如果3张图片分3类，会输出一个 $3 \times 3$ 的矩阵。

```
1 | import torch
2 | input = torch.tensor([[-0.4089, -1.2471, 0.5907],
3 |                      [-0.4897, -0.8267, -0.7349],
4 |                      [0.5241, -0.1246, -0.4751]])
```

先用Sigmoid函数将这些值放缩到0~1之间：

```
1 | m = torch.nn.Sigmoid()
2 | input = m(input)
3 | input
4 | >>> tensor([[0.3992, 0.2232, 0.6435],
5 |                  [0.3800, 0.3043, 0.3241],
6 |                  [0.6281, 0.4689, 0.3834]])
```

假设图片的真实标签是：

```
1 | target = torch.FloatTensor([[0, 1, 1],
2 |                               [0, 0, 1],
3 |                               [1, 0, 1]])
```

BCELoss的计算公式为：

$$-\frac{1}{n} \sum (y_n \times \ln x_n + (1 - y_n) \times \ln(1 - x_n))$$

其中  $y$  是真实标签， $x$  是模型输出的值。

所以对于第一行：

第一列： $0 \times \ln 0.3992 + (1 - 0) \times \ln(1 - 0.3992) = -0.5095$

第二列： $1 \times \ln 0.2232 + (1 - 1) \times \ln(1 - 0.2232) = -1.4997$

第三列： $1 \times \ln 0.6435 + (1 - 1) \times \ln(1 - 0.6435) = -0.4408$

对于第二行：

第一列： $0 \times \ln 0.3800 + (1 - 0) \times \ln(1 - 0.3800) = -0.4780$

第二列： $0 \times \ln 0.3044 + (1 - 0) \times \ln(1 - 0.3044) = -0.3630$

第三列： $1 \times \ln 0.3241 + (1 - 1) \times \ln(1 - 0.3241) = -1.1267$

对于第三行：

$$\text{第一列: } 1 \times \ln 0.6281 + (1 - 1) \times \ln(1 - 0.6281) = -0.4651$$

$$\text{第二列: } 0 \times \ln 0.4689 + (1 - 0) \times \ln(1 - 0.4689) = -0.6328$$

$$\text{第三列: } 1 \times \ln 0.3834 + (1 - 1) \times \ln(1 - 0.3834) = -0.9587$$

去掉负号求均值：

$$\frac{0.5095 + 1.4997 + 0.4408}{3} = 0.8167$$
$$\frac{0.4780 + 0.3630 + 1.1267}{3} = 0.6559$$
$$\frac{0.4651 + 0.6328 + 0.9587}{3} = 0.6855$$

再取个平均：

$$\frac{0.8167 + 0.6559 + 0.6855}{3} = 0.7194$$

下面我们用BCELoss来验证一下Loss是不是0.7194

```
1 loss = torch.nn.BCELoss()
2 loss(input, target)
3 >>> tensor(0.7193)
```

本章代码详见GitHub链接[wzy6642](#)。

## 第9章 自然语言处理

自然语言处理关注计算机与人类之间的自然语言交互。在实际中，我们常常使用自然语言处理技术，如“循环神经网络”一章中介绍的语言模型，来处理和分析大量的自然语言数据。本章中，根据输入与输出的不同形式，我们按“定长到定长”“不定长到定长”“不定长到不定长”的顺序，逐步展示在自然语言处理中如何表征并变换定长的词或类别以及不定长的句子或段落序列。

我们先介绍如何用向量表示词，并在语料库上训练词向量。之后，我们把在更大语料库上预训练的词向量应用于求近义词和类比词，即“定长到定长”。接着，在文本分类这种“不定长到定长”的任务中，我们进一步应用词向量来分析文本情感，并分别基于循环神经网络和卷积神经网络为表征时序数据提供两种思路。此外，自然语言处理任务中很多输出是不定长的，如任意长度的句子或段落。我们将描述应对这类问题的编码器—解码器模型、束搜索和注意力机制，并将它们应用于“不定长到不定长”的机器翻译任务中。

### 9.1 词嵌入 (word2vec)

自然语言是一套用来表达含义的复杂系统。在这套系统中，词是表义的基本单元。顾名思义，词向量是用来表示词的向量，也可被认为是词的特征向量或表征。把词映射为实数域向量的技术也叫词嵌入 (word embedding)。近年来，词嵌入已逐渐成为自然语言处理的基础知识。

#### 9.1.1 为何不采用one-hot向量

我们在5.4节（循环神经网络的从零开始实现）中使用one-hot向量表示词（字符为词）。回忆一下，假设词典中不同词的数量（词典大小）为 $N$ ，每个词可以和从0到 $N - 1$ 的连续整数一一对应。这些与词对应的整数叫作词的索引。假设一个词的索引为 $i$ ，为了得到该词的one-hot向量表示，我们创建一个全0的长为 $N$ 的向量，并将其第 $i$ 位设成1。这样一来，每个词就表示成了一个长度为 $N$ 的向量，可以直接被神经网络使用。

虽然one-hot词向量构造起来很容易，但通常并不是一个好选择。一个主要的原因是，one-hot词向量无法准确表达不同词之间的相似度，如我们常常使用的余弦相似度。对于向量 $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$ ，它们的余弦相似度是它们之间夹角的余弦值

$$\frac{\mathbf{x}^\top \mathbf{y}}{\|\mathbf{x}\| \|\mathbf{y}\|} \in [-1, 1]$$

由于任何两个不同词的one-hot向量的余弦相似度都为0，多个不同词之间的相似度难以通过one-hot向量准确地体现出来。

word2vec工具的提出正是为了解决上面这个问题。它将每个词表示成一个定长的向量，并使得这些向量能较好地表达不同词之间的相似和类比关系。[word2vec](#)工具包含了两个模型，即**跳字模型**（skip-gram）和**连续词袋模型**（continuous bag of words, CBOW）。接下来让我们分别介绍这两个模型以及它们的训练方法。

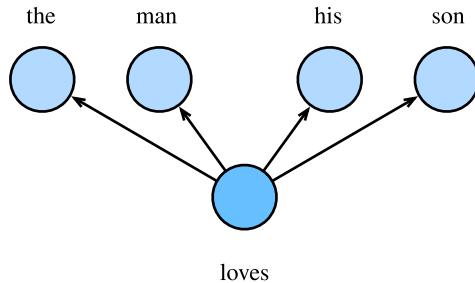
## 9.1.2 跳字模型

**跳字模型假设基于某个词来生成它在文本序列周围的词。**举个例子，假设文本序列是“the”“man”“loves”“his”“son”。以“loves”作为中心词，设背景窗口大小为2。如下图所示，跳字模型所关心的是，给定中心词“loves”，生成与它距离不超过2个词的背景词“the”“man”“his”“son”的条件概率，即

$$P(\text{"the"}, \text{"man"}, \text{"his"}, \text{"son"} \mid \text{"loves"})$$

假设给定中心词的情况下，背景词的生成是**相互独立的**，那么上式可以改写成

$$P(\text{"the"} \mid \text{"loves"}) \cdot P(\text{"man"} \mid \text{"loves"}) \cdot P(\text{"his"} \mid \text{"loves"}) \cdot P(\text{"son"} \mid \text{"loves"})$$



在跳字模型中，**每个词被表示成两个 $d$ 维向量**，用来计算条件概率。假设这个词在词典中索引为 $i$ ，当它为中心词时向量表示为 $\mathbf{v}_i \in \mathbb{R}^d$ ，而为背景词时向量表示为 $\mathbf{u}_i \in \mathbb{R}^d$ 。设中心词 $w_c$ 在词典中索引为 $c$ ，背景词 $w_o$ 在词典中索引为 $o$ ，**给定中心词生成背景词的条件概率可以通过对向量内积做softmax运算而得到：**

$$P(w_o \mid w_c) = \frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)}$$

其中词典索引集 $\mathcal{V} = 0, 1, \dots, |\mathcal{V}| - 1$ 。假设给定一个长度为 $T$ 的文本序列，设时间步 $t$ 的词为 $w^{(t)}$ 。假设给定中心词的情况下背景词的生成相互独立，当背景窗口大小为 $m$ 时，跳字模型的**似然函数**（参考本章附录）即给定任一中心词生成所有背景词的概率

$$\prod_{t=1}^T \prod_{-m \leq j \leq m, j \neq 0} P(w^{(t+j)} \mid w^{(t)})$$

这里小于1和大于 $T$ 的时间步可以忽略。

### 训练跳字模型

**跳字模型的参数是每个词所对应的中心词向量和背景词向量。**训练中我们通过最大化似然函数来学习模型参数，即最大似然估计。这等价于最小化以下损失函数：

$$-\sum_{t=1}^T \sum_{-m \leq j \leq m, j \neq 0} \log P(w^{(t+j)} | w^{(t)})$$

如果使用随机梯度下降，那么在每一次迭代里我们随机采样一个较短的子序列来计算有关孩子序列的损失，然后计算梯度来更新模型参数。梯度计算的关键是条件概率的对数有关中心词向量和背景词向量的梯度。根据定义，首先看到

$$\log P(w_o | w_c) = \mathbf{u}_o^\top \mathbf{v}_c - \log \left( \sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c) \right)$$

通过微分，我们可以得到上式中 $\mathbf{v}_c$ 的梯度

$$\begin{aligned} \frac{\partial \log P(w_o | w_c)}{\partial \mathbf{v}_c} &= \mathbf{u}_o - \frac{\sum_{j \in \mathcal{V}} \exp(\mathbf{u}_j^\top \mathbf{v}_c) \mathbf{u}_j}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)} \\ &= \mathbf{u}_o - \sum_{j \in \mathcal{V}} \left( \frac{\exp(\mathbf{u}_j^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)} \right) \mathbf{u}_j \\ &= \mathbf{u}_o - \sum_{j \in \mathcal{V}} P(w_j | w_c) \mathbf{u}_j \end{aligned}$$

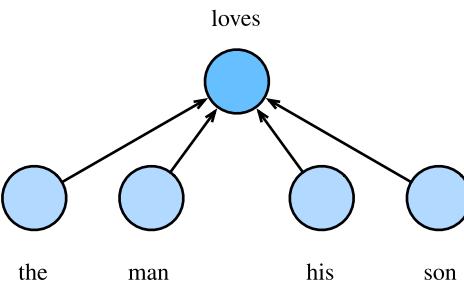
它的计算需要词典中所有词以 $w_c$ 为中心词的条件概率。有关其他词向量的梯度同理可得。

训练结束后，对于词典中的任一索引为*i*的词，我们均得到该词作为中心词和背景词的两组词向量 $\mathbf{v}_i$ 和 $\mathbf{u}_i$ 。在自然语言处理应用中，一般使用跳字模型的中心词向量作为词的表征向量。

### 9.1.3 连续词袋模型

连续词袋模型与跳字模型类似。与跳字模型最大的不同在于，连续词袋模型假设基于某中心词在文本序列前后的背景词来生成该中心词。在同样的文本序列“the”“man”“loves”“his”“son”里，以“loves”作为中心词，且背景窗口大小为2时，连续词袋模型关心的是，给定背景词“the”“man”“his”“son”生成中心词“loves”的条件概率（如下图所示），也就是

$$P(\text{"loves"} | \text{"the"}, \text{"man"}, \text{"his"}, \text{"son"})$$



因为连续词袋模型的背景词有多个，我们将这些背景词向量取平均，然后使用和跳字模型一样的方法来计算条件概率。设 $\mathbf{v}_i \in \mathbb{R}^d$ 和 $\mathbf{u}_i \in \mathbb{R}^d$ 分别表示词典中索引为*i*的词作为背景词和中心词的向量（注意符号的含义与跳字模型中的相反）。设中心词 $w_c$ 在词典中索引为 $c$ ，背景词 $w_{o_1}, \dots, w_{o_{2m}}$ 在词典中索引为 $o_1, \dots, o_{2m}$ ，那么给定背景词生成中心词的条件概率

$$P(w_c | w_{o_1}, \dots, w_{o_{2m}}) = \frac{\exp \left( \frac{1}{2m} \mathbf{u}_c^\top (\mathbf{v}_{o_1} + \dots + \mathbf{v}_{o_{2m}}) \right)}{\sum_{i \in \mathcal{V}} \exp \left( \frac{1}{2m} \mathbf{u}_i^\top (\mathbf{v}_{o_1} + \dots + \mathbf{v}_{o_{2m}}) \right)}$$

为了让符号更加简单，我们记 $\mathcal{W}_o = w_{o_1}, \dots, w_{o_{2m}}$ ，且 $\bar{\mathbf{v}}_o = (\mathbf{v}_{o_1} + \dots + \mathbf{v}_{o_{2m}}) / (2m)$ ，那么上式可以简写成

$$P(w_c | \mathcal{W}_o) = \frac{\exp(\mathbf{u}_c^\top \bar{\mathbf{v}}_o)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \bar{\mathbf{v}}_o)}$$

给定一个长度为 $T$ 的文本序列，设时间步 $t$ 的词为 $w^{(t)}$ ，背景窗口大小为 $m$ 。连续词袋模型的似然函数是由背景词生成任一中心词的概率

$$\prod_{t=1}^T P(w^{(t)} \mid w^{(t-m)}, \dots, w^{(t-1)}, w^{(t+1)}, \dots, w^{(t+m)})$$

### 训练连续词袋模型

训练连续词袋模型同训练跳字模型基本一致。连续词袋模型的最大似然估计等价于最小化损失函数

$$-\sum_{t=1}^T \log P(w^{(t)} \mid w^{(t-m)}, \dots, w^{(t-1)}, w^{(t+1)}, \dots, w^{(t+m)})$$

注意到

$$\log P(w_c \mid \mathcal{W}_o) = \mathbf{u}_c^\top \bar{\mathbf{v}}_o - \log \left( \sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \bar{\mathbf{v}}_o) \right)$$

通过微分，我们可以计算出上式中条件概率的对数有关任一背景词向量 $\mathbf{v}_{o_i}$  ( $i = 1, \dots, 2m$ ) 的梯度

$$\frac{\partial \log P(w_c \mid \mathcal{W}_o)}{\partial \mathbf{v}_{o_i}} = \frac{1}{2m} \left( \mathbf{u}_c - \sum_{j \in \mathcal{V}} \frac{\exp(\mathbf{u}_j^\top \bar{\mathbf{v}}_o) \mathbf{u}_j}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \bar{\mathbf{v}}_o)} \right) = \frac{1}{2m} \left( \mathbf{u}_c - \sum_{j \in \mathcal{V}} P(w_j \mid \mathcal{W}_o) \mathbf{u}_j \right)$$

有关其他词向量的梯度同理可得。同跳字模型不一样的一点在于，我们一般使用连续词袋模型的背景词向量作为词的表征向量。

**小结：**

- 词向量是用来表示词的向量。把词映射为实数域向量的技术也叫词嵌入。
- word2vec包含跳字模型和连续词袋模型。跳字模型假设基于中心词来生成背景词。连续词袋模型假设基于背景词来生成中心词。

## 9.2 近似训练

回忆上一节的内容。跳字模型的核心在于使用softmax运算得到给定中心词 $w_c$ 来生成背景词 $w_o$ 的条件概率

$$P(w_o \mid w_c) = \frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)}$$

该条件概率相应的对数损失

$$-\log P(w_o \mid w_c) = -\mathbf{u}_o^\top \mathbf{v}_c + \log \left( \sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c) \right)$$

由于softmax运算考虑了背景词可能是词典 $\mathcal{V}$ 中的任一词，以上损失包含了词典大小数目的项的累加。在上一节中我们看到，不论是跳字模型还是连续词袋模型，由于条件概率使用了softmax运算，每一步的梯度计算都包含词典大小数目的项的累加。对于含几十万或上百万词的较大词典，每次的梯度计算开销可能过大。为了降低该计算复杂度，本节将介绍两种近似训练方法，即负采样（negative sampling）或层序softmax（hierarchical softmax）。由于跳字模型和连续词袋模型类似，本节仅以跳字模型为例介绍这两种方法。

## 9.2.1 负采样

负采样修改了原来的目标函数。给定中心词 $w_c$ 的一个背景窗口，我们把背景词 $w_o$ 出现在该背景窗口看作一个事件，并将该事件的概率计算为

$$P(D = 1 \mid w_c, w_o) = \sigma(\mathbf{u}_o^\top \mathbf{v}_c)$$

其中的 $\sigma$ 函数与sigmoid激活函数的定义相同：

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

我们先考虑最大化文本序列中所有该事件的联合概率来训练词向量。具体来说，给定一个长度为 $T$ 的文本序列，设时间步 $t$ 的词为 $w^{(t)}$ 且背景窗口大小为 $m$ ，考虑最大化联合概率（似然函数）

$$\prod_{t=1}^T \prod_{-m \leq j \leq m, j \neq 0} P(D = 1 \mid w^{(t)}, w^{(t+j)})$$

然而，以上模型中包含的事件仅考虑了正类样本。这导致当所有词向量相等且值为无穷大时，以上的联合概率才被最大化为1。很明显，这样的词向量毫无意义。**负采样通过采样并添加负类样本使目标函数更有意义。**设背景词 $w_o$ 出现在中心词 $w_c$ 的一个背景窗口为事件 $P$ ，我们根据分布 $P(w)$ 采样 $K$ 个未出现在该背景窗口中的词，即噪声词。设噪声词 $w_k$  ( $k = 1, \dots, K$ ) 不出现在中心词 $w_c$ 的该背景窗口为事件 $N_k$ 。假设同时含有正类样本和负类样本的事件 $P, N_1, \dots, N_K$ 相互独立，负采样将以上需要最大化的仅考虑正类样本的联合概率改写为

$$\prod_{t=1}^T \prod_{-m \leq j \leq m, j \neq 0} P(w^{(t+j)} \mid w^{(t)})$$

其中条件概率被近似表示为

$$P(w^{(t+j)} \mid w^{(t)}) = P(D = 1 \mid w^{(t)}, w^{(t+j)}) \prod_{k=1, w_k \sim P(w)}^K P(D = 0 \mid w^{(t)}, w_k)$$

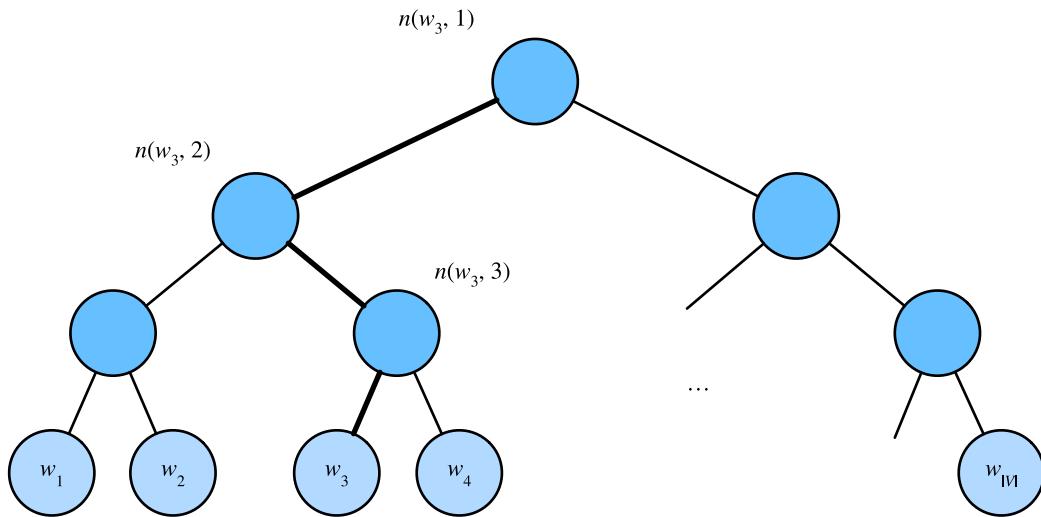
设文本序列中时间步 $t$ 的词 $w^{(t)}$ 在词典中的索引为 $i_t$ ，噪声词 $w_k$ 在词典中的索引为 $h_k$ 。有关以上条件概率的对数损失为（极大似然估计）

$$\begin{aligned} -\log P(w^{(t+j)} \mid w^{(t)}) &= -\log P(D = 1 \mid w^{(t)}, w^{(t+j)}) - \sum_{k=1, w_k \sim P(w)}^K \log P(D = 0 \mid w^{(t)}, w_k) \\ &= -\log \sigma(\mathbf{u}_{i_{t+j}}^\top \mathbf{v}_{i_t}) - \sum_{k=1, w_k \sim P(w)}^K \log(1 - \sigma(\mathbf{u}_{h_k}^\top \mathbf{v}_{i_t})) \\ &= -\log \sigma(\mathbf{u}_{i_{t+j}}^\top \mathbf{v}_{i_t}) - \sum_{k=1, w_k \sim P(w)}^K \log \sigma(-\mathbf{u}_{h_k}^\top \mathbf{v}_{i_t}) \end{aligned}$$

现在，训练中每一步的梯度计算开销不再与词典大小相关，而与 $K$ 线性相关。当 $K$ 取较小的常数时，负采样在每一步的梯度计算开销较小。

## 9.2.2 层序softmax

层序softmax是另一种近似训练法。它使用了二叉树这一数据结构，树的每个叶结点代表词典 $\mathcal{V}$ 中的每个词。



假设 $L(w)$ 为从二叉树的根结点到词 $w$ 的叶结点的路径（包括根结点和叶结点）上的结点数。设 $n(w, j)$ 为该路径上第 $j$ 个结点，并设该结点的背景词向量为 $\mathbf{u}_{n(w,j)}$ 。以上图为例， $L(w_3) = 4$ 。层序softmax将跳字模型中的条件概率近似表示为

$$P(w_o | w_c) = \prod_{j=1}^{L(w_o)-1} \sigma \left( \llbracket n(w_o, j+1) = \text{leftChild}(n(w_o, j)) \rrbracket \cdot \mathbf{u}_{n(w_o, j)}^\top \mathbf{v}_c \right)$$

其中 $\sigma$ 函数与2.8节（多层感知机）中sigmoid激活函数的定义相同， $\text{leftChild}(n)$ 是结点 $n$ 的左子结点：如果判断 $x$ 为真， $\llbracket x \rrbracket = 1$ ；反之 $\llbracket x \rrbracket = -1$ 。让我们计算上图中给定词 $w_c$ 生成词 $w_3$ 的条件概率。我们需要将 $w_c$ 的词向量 $\mathbf{v}_c$ 和根结点到 $w_3$ 路径上的非叶结点向量一一求内积。由于在二叉树中由根结点到叶结点 $w_3$ 的路径上需要向左、向右再向左地遍历（上图中加粗的路径），我们得到

$$P(w_3 | w_c) = \sigma(\mathbf{u}_{n(w_3, 1)}^\top \mathbf{v}_c) \cdot \sigma(-\mathbf{u}_{n(w_3, 2)}^\top \mathbf{v}_c) \cdot \sigma(\mathbf{u}_{n(w_3, 3)}^\top \mathbf{v}_c)$$

由于 $\sigma(x) + \sigma(-x) = 1$ ，给定中心词 $w_c$ 生成词典 $\mathcal{V}$ 中任一词的条件概率之和为1这一条件也将满足：

$$\sum_{w \in \mathcal{V}} P(w | w_c) = 1$$

此外，由于 $L(w_o) - 1$ 的数量级为 $\mathcal{O}(\log_2 |\mathcal{V}|)$ ，当词典 $\mathcal{V}$ 很大时，层序softmax在训练中每一步的梯度计算开销相较于使用近似训练时大幅降低。

### 小结：

- 负采样通过考虑同时含有正类样本和负类样本的相互独立事件来构造损失函数。其训练中每一步的梯度计算开销与采样的噪声词的个数线性相关。
- 层序softmax使用了二叉树，并根据根结点到叶结点的路径来构造损失函数。其训练中每一步的梯度计算开销与词典大小的对数相关。

## 9.3 word2vec的实现

本节是对前两节内容的实践。我们以9.1节（词嵌入word2vec）中的跳字模型和9.2节（近似训练）中的负采样为例，介绍在语料库上训练词嵌入模型的实现。我们还会介绍一些实现中的技巧，如二次采样（subsampling）。

首先导入实验所需的包或模块。

```
1 In [1]:  
2     import collections  
3     import math  
4     import random  
5     import sys  
6     import time  
7     import os  
8     import numpy as np  
9     import torch  
10    from torch import nn  
11    import torch.utils.data as Data  
12    import d2lzh as d2l
```

### 9.3.1 预处理数据集

PTB (Penn Tree Bank) 是一个常用的小型语料库。它采样自《华尔街日报》的文章，包括训练集、验证集和测试集。我们将在PTB训练集上训练词嵌入模型。该数据集的每一行作为一个句子。句子中的每个词由空格隔开。

```
1 In [2]:  
2     with open('data/ptb/ptb.train.txt', 'r') as f:  
3         lines = f.readlines()  
4         # st是sentence的缩写  
5         raw_dataset = [st.split() for st in lines]  
6         '# sentences: %d' % len(raw_dataset)  
7 Out [2]:  
8     '# sentences: 42068'
```

对于数据集的前3个句子，打印每个句子的词数和前5个词。这个数据集中句尾符为"<eos>"，生僻词全用"<unk>"表示，数字则被替换成"N"。

```
1 In [3]:  
2     for st in raw_dataset[:3]:  
3         print('# tokens:', len(st), st[:5])  
4 Out [3]:  
5     # tokens: 24 ['aer', 'banknote', 'berlitz', 'calloway', 'centrust']  
6     # tokens: 15 ['pierre', '<unk>', 'N', 'years', 'old']  
7     # tokens: 11 ['mr.', '<unk>', 'is', 'chairman', 'of']
```

#### 1. 建立词语索引

为了计算简单，我们只保留在数据集中至少出现5次的词。

```
1 In [4]:  
2     # tk是token的缩写  
3     counter = collections.Counter(  
4         [tk for st in raw_dataset for tk in st])  
5     counter = dict(filter(lambda x: x[1]>=5, counter.items()))
```

然后将词映射到整数索引。

```

1 In [5]:
2     idx_to_token = [tk for tk, _ in counter.items()]
3     token_to_idx = {tk: idx for idx, tk in enumerate(idx_to_token)}
4     dataset = [[token_to_idx[tk] for tk in st if tk in token_to_idx]
5                 for st in raw_dataset]
6     num_tokens = sum([len(st) for st in dataset])
7     '# tokens: %d' % num_tokens
8 Out [5]:
9     '# tokens: 887100'

```

## 2. 二次采样

文本数据中一般会出现一些高频词，如英文中的“the”“a”和“in”。通常来说，在一个背景窗口中，一个词（如“chip”）和较低频词（如“microprocessor”）同时出现比和较高频词（如“the”）同时出现对训练词嵌入模型更有益。因此，训练词嵌入模型时可以对词进行二次采样。具体来说，**数据集中每个被索引词 $w_i$ 将有一定概率被丢弃**，该丢弃概率为

$$P(w_i) = \max\left(1 - \sqrt{\frac{t}{f(w_i)}}, 0\right)$$

其中  $f(w_i)$  是数据集中词 $w_i$ 的个数与总词数之比，常数 $t$ 是一个超参数（实验中设为 $10^{-4}$ ）。可见，只有当 $f(w_i) > t$ 时，我们才有可能在二次采样中丢弃词 $w_i$ ，并且越高的词被丢弃的概率越大。

```

1 In [6]:
2     def discard(idx):
3         # 与均匀分布对比，确定该词是否被剔除
4         return random.uniform(0, 1) < 1-math.sqrt(
5             1e-4 / counter[idx_to_token[idx]] * num_tokens
6         )
7     subsampled_dataset = [[tk for tk in st if not discard(tk)]
8                           for st in dataset]
9     '# tokens: %d' % sum([len(st) for st in subsampled_dataset])
10 Out [6]:
11     '# tokens: 375413'

```

可以看到，二次采样后我们掉了一半左右的词。下面比较一个词在二次采样前后出现在数据集中的次数。可见高频词“the”的采样率不足1/20。

```

1 In [7]:
2     def compare_counts(token):
3         return '# %s: before=%d, after=%d' % (token, sum(
4             [st.count(token_to_idx[token]) for st in dataset]), sum(
5                 [st.count(token_to_idx[token]) for st in subsampled_dataset]))
6
7     compare_counts('the')
8 Out [7]:
9     '# the: before=50770, after=2131'

```

但低频词“join”则完整地保留了下来。

```

1 In [8]:
2     compare_counts('join')
3 Out [8]:
4     '# join: before=45, after=45'

```

### 3. 提取中心词和背景词

我们将与中心词距离不超过背景窗口大小的词作为它的背景词。下面定义函数提取出所有中心词和它们的背景词。它每次在整数1和 max\_window\_size (最大背景窗口) 之间随机均匀采样一个整数作为背景窗口大小。

```
1 In [9]:  
2     def get_centers_and_contexts(dataset, max_window_size):  
3         centers, contexts = [], []  
4         for st in dataset:  
5             # 每个句子至少有2个词才能组成一对“中心词-背景词”  
6             if len(st)<2:  
7                 continue  
8             # 只要句子长度大于等于2, 每个词都要做中心词  
9             centers += st  
10            for center_i in range(len(st)):  
11                window_size = random.randint(1, max_window_size)  
12                indices = list(range(max(0, center_i-window_size),  
13                                min(len(st), center_i+1+window_size)  
14                                ))  
15                # 将中心词排除在背景词外  
16                indices.remove(center_i)  
17                contexts.append([st[idx] for idx in indices])  
18            return centers, contexts
```

下面我们创建一个人工数据集，其中含有词数分别为7和3的两个句子。设最大背景窗口为2，打印所有中心词和它们的背景词。

```
1 In [10]:  
2     tiny_dataset = [list(range(7)), list(range(7, 10))]  
3     print('dataset', tiny_dataset)  
4     for center, context in zip(*get_centers_and_contexts(tiny_dataset, 2)):  
5         print('center', center, 'has contexts', context)  
6 Out [10]:  
7     dataset [[0, 1, 2, 3, 4, 5, 6], [7, 8, 9]]  
8     center 0 has contexts [1]  
9     center 1 has contexts [0, 2]  
10    center 2 has contexts [0, 1, 3, 4]  
11    center 3 has contexts [2, 4]  
12    center 4 has contexts [3, 5]  
13    center 5 has contexts [3, 4, 6]  
14    center 6 has contexts [4, 5]  
15    center 7 has contexts [8, 9]  
16    center 8 has contexts [7, 9]  
17    center 9 has contexts [7, 8]
```

实验中，我们设最大背景窗口大小为5。下面提取数据集中所有的中心词及其背景词。

```
1 In [11]:  
2     all_centers, all_contexts = get_centers_and_contexts(  
3         subsampled_dataset, 5)
```

## 9.3.2 负采样

我们使用负采样来进行近似训练。对于一对中心词和背景词，我们随机采样背景词个数 $K$ 倍个噪声词（实验中设 $K = 5$ ）。根据word2vec论文的建议，噪声词采样概率 $P(w)$ 设为 $w$ 词频与总词频之比的0.75次方。

```
1 In [12]:  
2     def get_negatives(all_contexts, sampling_weights, K):  
3         # all_contexts每个词的背景词列表  
4         # sampling_weights每个词词频的0.75次幂  
5         # K噪声词相比于背景词个数的倍数  
6         all_negatives, neg_candidates, i = [], [], 0  
7         # 词表中词的个数  
8         population = list(range(len(sampling_weights)))  
9         for contexts in all_contexts:  
10            negatives = []  
11            while len(negatives) < len(contexts)*K:  
12                if i==len(neg_candidates):  
13                    # 从population随机选取k次数据，返回一个列表  
14                    # 根据每个词的权重随机生成k个词的索引作为噪声词  
15                    # 为了高效计算，可以将k设的稍微大一点  
16                    i, neg_candidates = 0, random.choices(  
17                        population, sampling_weights, k=int(1e5)  
18                    )  
19                    neg, i = neg_candidates[i], i+1  
20                    # 噪声词不能是背景词  
21                    if neg not in set(contexts):  
22                        negatives.append(neg)  
23                    all_negatives.append(negatives)  
24            return all_negatives  
25 sampling_weights = [counter[w]**0.75 for w in idx_to_token]  
26 all_negatives = get_negatives(all_contexts, sampling_weights, 5)
```

## 9.3.3 读取数据集

我们从数据集中提取所有中心词 `all_centers`，以及每个中心词对应的背景词 `all_contexts` 和噪声词 `all_negatives`。我们先定义一个 `Dataset` 类。

```
1 In [13]:  
2     class MyDataset(torch.utils.data.Dataset):  
3         def __init__(self, centers, contexts, negatives):  
4             assert len(centers)==len(contexts)==len(negatives)  
5             self.centers = centers  
6             self.contexts = contexts  
7             self.negatives = negatives  
8         def __getitem__(self, index):  
9             return (self.centers[index],  
10                  self.contexts[index],  
11                  self.negatives[index])  
12         def __len__(self):  
13             return len(self.centers)
```

我们将通过随机小批量来读取它们。在一个小批量数据中，第 $i$ 个样本包括一个中心词以及它所对应的 $n_i$ 个背景词和 $m_i$ 个噪声词。由于每个样本的背景窗口大小可能不一样，其中背景词与噪声词个数之和 $n_i + m_i$ 也会不同。**在构造小批量时，我们将每个样本的背景词和噪声词连结在一起，并添加填充项0直至连结后的长度相同**，即长度均为 $\max_i n_i + m_i$ （`max_len`变量）。为了避免填充项对损失函数计算的影响，我们构造了掩码变量 `masks`，其每一个元素分别与连结后的背景词和噪声词 `contexts_negatives` 中的元素一一对应。当 `contexts_negatives` 变量中的某个元素为填充项时，相同位置的掩码变量 `masks` 中的元素取0，否则取1。为了区分正类和负类，我们还需要将 `contexts_negatives` 变量中的背景词和噪声词区分开来。依据掩码变量的构造思路，我们只需创建与 `contexts_negatives` 变量形状相同的标签变量 `labels`，并将与背景词（正类）对应的元素设1，其余清0。

下面我们实现这个小批量读取函数 `batchify`。它的小批量输入 `data` 是一个长度为批量大小的列表，其中每个元素分别包含中心词 `center`、背景词 `context` 和噪声词 `negative`。该函数返回的小批量数据符合我们需要的格式，例如，包含了掩码变量。

```
1 In [14]:  
2     def batchify(data):  
3         # 中心词、背景词、噪声词  
4         max_len = max(len(c)+len(n) for _, c, n in data)  
5         centers, contexts_negatives, masks, labels = [], [], [], []  
6         for center, context, negative in data:  
7             cur_len = len(context)+len(negative)  
8             centers += [center]  
9             contexts_negatives += [context+negative+[0]*(max_len-cur_len)]  
10            masks += [[1]*cur_len+[0]*(max_len-cur_len)]  
11            labels += [[1]*len(context)+[0]*(max_len-len(context))]  
12        return (torch.tensor(centers).view(-1, 1),  
13                torch.tensor(contexts_negatives),  
14                torch.tensor(masks),  
15                torch.tensor(labels))
```

我们用刚刚定义的 `batchify` 函数指定 `DataLoader` 实例中小批量的读取方式，然后打印读取的第一个批量中各个变量的形状。

```
1 In [15]:  
2     batch_size = 512  
3     num_workers = 4  
4     dataset = MyDataset(all_centers, all_contexts, all_negatives)  
5     data_iter = Data.DataLoader(dataset, batch_size, shuffle=True,  
6                                 collate_fn=batchify, num_workers=4)  
7     for batch in data_iter:  
8         for name, data in zip(['centers', 'contexts_negatives',  
9                               'masks', 'labels'], batch):  
10            print(name, 'shape:', data.shape)  
11            break  
12 Out [15]:  
13     centers shape: torch.Size([512, 1])  
14     contexts_negatives shape: torch.Size([512, 60])  
15     masks shape: torch.Size([512, 60])  
16     labels shape: torch.Size([512, 60])
```

## 9.3.4 跳字模型

我们将通过使用嵌入层和小批量乘法来实现跳字模型。它们也常常用于实现其他自然语言处理的应用。

### 1. 嵌入层

获取词嵌入的层称为嵌入层，在PyTorch中可以通过创建 `nn.Embedding` 实例得到。嵌入层的权重是一个矩阵，其行数为词典大小（`num_embeddings`），列数为每个词向量的维度（`embedding_dim`）。我们设词典大小为20，词向量的维度为4。

```
1 In [16]:  
2     embed = nn.Embedding(num_embeddings=20, embedding_dim=4)  
3     embed.weight  
4 Out [16]:  
5     Parameter containing:  
6     tensor([[-7.5014e-01, -6.7758e-01,  1.3905e-01, -3.2504e-01],  
7             [ 4.6432e-01, -1.2951e+00, -6.3469e-01,  1.7867e+00],  
8             [-1.4255e+00, -5.1740e-01, -1.8344e+00, -8.6178e-01],  
9             [-4.1812e-01,  9.8486e-01,  8.6274e-01, -1.8278e-01],  
10            [-4.8235e-02, -4.4251e-01,  1.8103e+00,  1.4304e-03],  
11            [ 6.0103e-01,  5.2687e-01, -7.9238e-01, -3.8206e-01],  
12            [-6.4648e-01,  6.1382e-01, -2.6217e-01,  1.8242e+00],  
13            [ 2.1424e-01, -1.2573e+00,  9.9863e-01,  4.7190e-01],  
14            [-1.4787e-01, -1.3340e+00,  9.4021e-03,  2.0213e-01],  
15            [-2.3015e-01, -5.4096e-01,  7.9691e-01,  5.0277e-01],  
16            [-2.4606e-01, -1.6335e+00, -9.0530e-03, -5.7450e-01],  
17            [ 1.7134e+00, -1.9645e+00, -1.8626e-01,  8.8031e-01],  
18            [ 1.7762e+00, -1.2961e-01, -1.2376e+00, -5.2555e-01],  
19            [ 1.7810e-01,  1.5218e+00, -2.2159e-01,  2.5675e-01],  
20            [ 1.5030e+00,  2.1536e-01,  1.1065e+00, -3.9987e-01],  
21            [ 3.3883e-01, -5.9900e-01, -4.5714e-01,  1.7813e+00],  
22            [-2.2987e-01, -1.6290e+00,  1.2975e+00, -5.6441e-01],  
23            [ 6.4336e-01,  4.5049e-02, -1.5265e+00,  5.7984e-01],  
24            [-3.3092e-01,  1.1003e-01,  9.1502e-01, -2.5245e-01],  
25            [ 9.8932e-01,  4.5280e-01, -1.1923e+00, -1.7245e+00]],  
26            requires_grad=True)
```

嵌入层的输入为词的索引。输入一个词的索引*i*，嵌入层返回权重矩阵的第*i*行作为它的词向量。下面我们将形状为(2, 3)的索引输入进嵌入层，由于词向量的维度为4，我们得到形状为(2, 3, 4)的词向量。

```
1 In [17]:  
2     x = torch.tensor([[1, 2, 3], [4, 5, 6]], dtype=torch.long)  
3     embed(x)  
4 Out [17]:  
5     tensor([[[ 4.6432e-01, -1.2951e+00, -6.3469e-01,  1.7867e+00],  
6             [-1.4255e+00, -5.1740e-01, -1.8344e+00, -8.6178e-01],  
7             [-4.1812e-01,  9.8486e-01,  8.6274e-01, -1.8278e-01]],  
8  
9             [[-4.8235e-02, -4.4251e-01,  1.8103e+00,  1.4304e-03],  
10            [ 6.0103e-01,  5.2687e-01, -7.9238e-01, -3.8206e-01],  
11            [-6.4648e-01,  6.1382e-01, -2.6217e-01,  1.8242e+00]]],  
12            grad_fn=<EmbeddingBackward>)
```

### 2. 小批量乘法

我们可以使用小批量乘法运算 `bmm` 对两个小批量中的矩阵一一做乘法。假设第一个小批量中包含  $n$  个形状为  $a \times b$  的矩阵  $\mathbf{X}_1, \dots, \mathbf{X}_n$ , 第二个小批量中包含  $n$  个形状为  $b \times c$  的矩阵  $\mathbf{Y}_1, \dots, \mathbf{Y}_n$ 。这两个小批量的矩阵乘法输出为  $n$  个形状为  $a \times c$  的矩阵  $\mathbf{X}_1\mathbf{Y}_1, \dots, \mathbf{X}_n\mathbf{Y}_n$ 。因此, 给定两个形状分别为  $(n, a, b)$  和  $(n, b, c)$  的 `Tensor`, 小批量乘法输出的形状为  $(n, a, c)$ 。

```
1 In [18]:  
2     X = torch.ones((2, 1, 4))  
3     Y = torch.ones((2, 4, 6))  
4     torch.bmm(X, Y).shape  
5 Out [18]:  
6     torch.Size([2, 1, 6])
```

### 3. 跳字模型前向计算

在前向计算中, 跳字模型的输入包含中心词索引 `center` 以及连结的背景词与噪声词索引 `contexts_and_negatives`。其中 `center` 变量的形状为(批量大小, 1), 而 `contexts_and_negatives` 变量的形状为(批量大小, `max_len`)。这两个变量先通过词嵌入层分别由词索引变换为词向量, 再通过小批量乘法得到形状为(批量大小, 1, `max_len`)的输出。输出中的每个元素是中心词向量与背景词向量或噪声词向量的内积。

```
1 In [19]:  
2     def skip_gram(center, contexts_and_negatives,  
3                     embed_v, embed_u):  
4         # 每一个词由背景词向量和中心词向量表示  
5         # 所以需要两个嵌入表示  
6         v = embed_v(center)  
7         u = embed_u(contexts_and_negatives)  
8         # batch, emb, num  
9         pred = torch.bmm(v, u.permute(0, 2, 1))  
10        return pred
```

## 9.3.5 训练模型

在训练词嵌入模型之前, 我们需要定义模型的损失函数。

### 1. 二元交叉熵损失函数

根据负采样中损失函数的定义, 我们可以使用二元交叉熵损失函数, 下面定义 `SigmoidBinaryCrossEntropyLoss`。

```
1 In [20]:  
2     class SigmoidBinaryCrossEntropyLoss(nn.Module):  
3         def __init__(self):  
4             super(SigmoidBinaryCrossEntropyLoss, self).__init__()  
5         def forward(self, inputs, targets, mask=None):  
6             """  
7                 input - Tensor shape: (batch_size, len)  
8                 target - Tensor of the same shape as input  
9             """  
10            inputs, targets = inputs.float(), targets.float()  
11            mask = mask.float()  
12            res = nn.functional.binary_cross_entropy_with_logits(  
13                inputs, targets, reduction='none', weight=mask)  
14            return res.mean(dim=1)  
15    loss = SigmoidBinaryCrossEntropyLoss()
```

值得一提的是，我们可以通过掩码变量指定小批量中参与损失函数计算的部分预测值和标签：当掩码为1时，相应位置的预测值和标签将参与损失函数的计算；当掩码为0时，相应位置的预测值和标签则不参与损失函数的计算。我们之前提到，**掩码变量可用于避免填充项对损失函数计算的影响**。

```
1 In [21]:  
2     pred = torch.tensor([[1.5, 0.3, -1, 2], [1.1, -0.6, 2.2, 0.4]])  
3     # 标签变量label中的1和0分别代表背景词和噪声词  
4     label = torch.tensor([[1, 0, 0, 0], [1, 1, 0, 0]])  
5     # 掩码变量  
6     mask = torch.tensor([[1, 1, 1, 1], [1, 1, 1, 0]])  
7     loss(pred, label, mask)*mask.shape[1]/mask.float().sum(dim=1)  
8 Out [21]:  
9     tensor([0.8740, 1.2100])
```

作为比较，下面将从零开始实现二元交叉熵损失函数的计算，并根据掩码变量 `mask` 计算掩码为1的预测值和标签的损失。

```
1 In [22]:  
2     def sigmd(x):  
3         return -math.log(1/(1+math.exp(-x)))  
4     # 1-sigmd(x)=sigmd(-x)  
5     # 背景词部分计算sigmd(x), 噪声词部分计算sigmd(-x)  
6     print('%.4f' % ((sigmd(1.5)+sigmd(-0.3)+sigmd(1)+sigmd(-2))/4))  
7     print('%.4f' % ((sigmd(1.1)+sigmd(-0.6)+sigmd(-2.2))/3))  
8 Out [22]:  
9     0.8740  
10    1.2100
```

## 2. 初始化模型参数

我们分别构造中心词和背景词的嵌入层，并将超参数词向量维度 `embed_size` 设置成100。

```
1 In [23]:  
2     embed_size = 100  
3     net = nn.Sequential(  
4         nn.Embedding(num_embeddings=len(idx_to_token),  
5                         embedding_dim=embed_size),  
6         nn.Embedding(num_embeddings=len(idx_to_token),  
7                         embedding_dim=embed_size)  
8     )
```

## 3. 定义训练函数

下面定义训练函数。由于填充项的存在，与之前的训练函数相比，损失函数的计算稍有不同。

```
1 In [24]:  
2     def train(net, lr, num_epochs):  
3         device = 'cuda'  
4         net = net.to(device)  
5         optimizer = torch.optim.Adam(net.parameters(), lr=lr)  
6         for epoch in range(num_epochs):  
7             start, l_sum, n = time.time(), 0.0, 0  
8             for batch in data_iter:  
9                 center, context_negative, mask, label = [  
10                     d.to(device) for d in batch]
```

```

11         pred = skip_gram(center, context_negative,
12                           net[0], net[1])
13         # 使用掩码变量mask来避免填充项对损失函数计算的影响
14         # 一个batch的平均loss
15         l = (
16             loss(pred.view(label.shape), label, mask) *
17             mask.shape[1]/mask.float().sum(dim=1)
18         ).mean()
19         optimizer.zero_grad()
20         l.backward()
21         optimizer.step()
22         l_sum += l.cpu().item()
23         n += 1
24     print('epoch %d, loss %.2f, time %.2f' % (epoch+1, l_sum/n,
25                                               time.time()-start))

```

现在我们就可以使用负采样训练跳字模型了。

```

1 In [25]:
2     train(net, 0.01, 10)
3 Out [25]:
4     epoch 1, loss 1.96, time 14.15
5     epoch 2, loss 0.62, time 13.38
6     epoch 3, loss 0.45, time 13.23
7     epoch 4, loss 0.39, time 13.57
8     epoch 5, loss 0.37, time 13.39
9     epoch 6, loss 0.35, time 13.27
10    epoch 7, loss 0.34, time 13.31
11    epoch 8, loss 0.33, time 13.50
12    epoch 9, loss 0.32, time 13.92
13    epoch 10, loss 0.32, time 13.38

```

### 9.3.6 应用词嵌入模型

训练好词嵌入模型之后，我们可以根据两个词向量的余弦相似度表示词与词之间在语义上的相似度。可以看到，使用训练得到的词嵌入模型时，与词“chip”语义最接近的词大多与芯片有关。

```

1 In [26]:
2     def get_similar_tokens(query_tokens, k, embed):
3         w = embed.weight.data
4         x = w[token_to_idx[query_tokens]]
5         # 添加1e-9是为了数值稳定性
6         cos = torch.matmul(w, x)/(
7             torch.sum(w*w, dim=1)*(torch.sum(x*x)+1e-9)).sqrt()
8         _, topk = torch.topk(cos, k=k+1)
9         topk = topk.cpu().numpy()
10        # 除去输入词
11        for i in topk[1:]:
12            print('cosine sim=%3f: %s' % (cos[i], (idx_to_token[i])))
13    get_similar_tokens('chip', 3, net[0])
14 Out [26]:
15    cosine sim=0.486: bugs
16    cosine sim=0.484: computers
17    cosine sim=0.444: mips

```

**小结：**

- 可以使用PyTorch通过负采样训练跳字模型。
- 二次采样试图尽可能减轻高频词对训练词嵌入模型的影响。
- 可以将长度不同的样本填充至长度相同的小批量，并通过掩码变量区分非填充和填充，然后只令非填充参与损失函数的计算。

## 9.4 子词嵌入 (fastText)

英语单词通常有其内部结构和形成方式。例如，我们可以从“dog”“dogs”和“dogcatcher”的字面上推测它们的关系。这些词都有同一个词根“dog”，但使用不同的后缀来改变词的含义。而且，这个关联可以推广至其他词汇。例如，“dog”和“dogs”的关系如同“cat”和“cats”的关系，“boy”和“boyfriend”的关系如同“girl”和“girlfriend”的关系。这一特点并非为英语所独有。在法语和西班牙语中，很多动词根据场景不同有40多种不同的形态，而在芬兰语中，一个名词可能有15种以上的形态。事实上，构词学 (morphology) 作为语言学的一个重要分支，研究的正是词的内部结构和形成方式。

在word2vec中，我们并没有直接利用构词学中的信息。无论是在跳字模型还是连续词袋模型中，我们都将形态不同的单词用不同的向量来表示。例如，“dog”和“dogs”分别用两个不同的向量表示，而模型中并未直接表达这两个向量之间的关系。鉴于此，**fastText提出了子词嵌入 (subword embedding) 的方法，从而试图将构词信息引入word2vec中的跳字模型。**

在fastText中，每个中心词被表示成子词的集合。下面我们用单词“where”作为例子来了解子词是如何产生的。首先，我们在单词的首尾分别添加特殊字符“<”和“>”以区分作为前后缀的子词。然后，将单词当成一个由字符构成的序列来提取n元语法。例如，当n = 3时，我们得到所有长度为3的子词：“<wh”“whe”“her”“ere”“re>”以及特殊子词“<where>”。

在fastText中，对于一个词 $w$ ，我们将它所有长度在3 ~ 6的子词和特殊子词的并集记为 $\mathcal{G}_w$ 。那么词典则是所有词的子词集合的并集。假设词典中子词 $g$ 的向量为 $\mathbf{z}_g$ ，那么跳字模型中词 $w$ 的作为中心词的向量 $\mathbf{v}_w$ 则表示成

$$\mathbf{v}_w = \sum_{g \in \mathcal{G}_w} \mathbf{z}_g$$

fastText的其余部分同跳字模型一致，不在此重复。可以看到，与跳字模型相比，fastText中词典规模更大，造成模型参数更多，同时一个词的向量需要对所有子词向量求和，继而导致计算复杂度更高。**但与此同时，较生僻的复杂单词，甚至是词典中没有的单词，可能会从同它结构类似的其他词那里获取更好的词向量表示。**

**小结：**

- fastText提出了子词嵌入方法。它在word2vec中的跳字模型的基础上，将中心词向量表示成单词的子词向量之和。
- 子词嵌入利用构词上的规律，通常可以提升生僻词表示的质量。

## 9.5 全局向量的词嵌入 (GloVe)

让我们先回顾一下word2vec中的跳字模型。将跳字模型中使用softmax运算表达的条件概率 $P(w_j | w_i)$ 记作 $q_{ij}$ ，即

$$q_{ij} = \frac{\exp(\mathbf{u}_j^\top \mathbf{v}_i)}{\sum_{k \in \mathcal{V}} \exp(\mathbf{u}_k^\top \mathbf{v}_i)}$$

其中 $\mathbf{v}_i$ 和 $\mathbf{u}_i$ 分别是索引为 $i$ 的词 $w_i$ 作为中心词和背景词时的向量表示， $\mathcal{V} = 0, 1, \dots, |\mathcal{V}| - 1$ 为词典索引集。

对于词 $w_i$ ，它在数据集中可能多次出现。我们将每一次以它作为中心词的所有背景词全部汇总并保留重复元素，记作多重集 (multiset)  $\mathcal{C}_i$ 。**一个元素在多重集中的个数称为该元素的重数 (multiplicity)**。举例来说，假设词 $w_i$ 在数据集中出现2次：文本序列中以这2个 $w_i$ 作为中心词的背景窗口分别包含背景词索引2, 1, 5, 2和2, 3, 2, 1。那么多重集 $\mathcal{C}_i = \{1, 1, 2, 2, 2, 3, 5\}$ ，其中元素1的重数

为2，元素2的重数为4，元素3和5的重数均为1。将多重集 $\mathcal{C}_i$ 中元素 $j$ 的重数记作 $x_{ij}$ ：它表示了整个数据集中所有以 $w_i$ 为中心词的背景窗口中词 $w_j$ 的个数。那么，跳字模型的损失函数还可以用另一种方式表达（交叉熵）：

$$-\sum_{i \in \mathcal{V}} \sum_{j \in \mathcal{V}} x_{ij} \log q_{ij}$$

我们将数据集中所有以词 $w_i$ 为中心词的背景词的数量之和 $|\mathcal{C}_i|$ 记为 $x_i$ ，并将以 $w_i$ 为中心词生成背景词 $w_j$ 的条件概率 $x_{ij}/x_i$ 记作 $p_{ij}$ 。我们可以进一步改写跳字模型的损失函数为

$$-\sum_{i \in \mathcal{V}} x_i \sum_{j \in \mathcal{V}} p_{ij} \log q_{ij}$$

**上式中， $-\sum_{j \in \mathcal{V}} p_{ij} \log q_{ij}$ 计算的是以 $w_i$ 为中心词的背景词条件概率分布 $p_{ij}$ 和模型预测的条件概率分布 $q_{ij}$ 的交叉熵，且损失函数使用所有以词 $w_i$ 为中心词的背景词的数量之和来加权。最小化上式中的损失函数会令预测的条件概率分布尽可能接近真实的条件概率分布。**

然而，作为常用损失函数的一种，交叉熵损失函数有时并不是好的选择。一方面，正如我们在9.2节（近似训练）中所提到的，令模型预测 $q_{ij}$ 成为合法概率分布的代价是它在分母中基于整个词典的累加项。这很容易带来过大的计算开销。另一方面，词典中往往有大量生僻词，它们在数据集中出现的次数极少。而有关大量生僻词的条件概率分布在交叉熵损失函数中的最终预测往往并不准确。

### 9.5.1 GloVe模型

鉴于此，作为在word2vec之后提出的词嵌入模型，GloVe模型采用了**平方损失**，并基于该损失对跳字模型做了3点改动：

1. 使用非概率分布的变量 $p'_{ij} = x_{ij}$ 和 $q'_{ij} = \exp(\mathbf{u}_j^\top \mathbf{v}_i)$ ，并对它们取对数。因此，平方损失项是 $(\log p'_{ij} - \log q'_{ij})^2 = (\mathbf{u}_j^\top \mathbf{v}_i - \log x_{ij})^2$ 。
2. 为每个词 $w_i$ 增加两个为标量的模型参数：中心词偏差项 $b_i$ 和背景词偏差项 $c_i$ 。
3. 将每个损失项的权重替换成函数 $h(x_{ij})$ 。权重函数 $h(x)$ 是值域在[0, 1]的单调递增函数。

如此一来，GloVe模型的目标是最小化损失函数

$$\sum_{i \in \mathcal{V}} \sum_{j \in \mathcal{V}} h(x_{ij}) (\mathbf{u}_j^\top \mathbf{v}_i + b_i + c_j - \log x_{ij})^2$$

其中权重函数 $h(x)$ 的一个建议选择是：当 $x < c$ 时（如 $c = 100$ ），令 $h(x) = (x/c)^\alpha$ （如 $\alpha = 0.75$ ），反之令 $h(x) = 1$ 。因为 $h(0) = 0$ ，所以对于 $x_{ij} = 0$ 的平方损失项可以直接忽略。当使用小批量随机梯度下降来训练时，每个时间步我们随机采样小批量非零 $x_{ij}$ ，然后计算梯度来迭代模型参数。这些非零 $x_{ij}$ 是预先基于整个数据集计算得到的，包含了数据集的全局统计信息。因此，GloVe模型的命名取“全局向量”（Global Vectors）之意。

需要强调的是，如果词 $w_i$ 出现在词 $w_j$ 的背景窗口里，那么词 $w_j$ 也会出现在词 $w_i$ 的背景窗口里。也就是说， $x_{ij} = x_{ji}$ 。不同于word2vec中拟合的是非对称的条件概率 $p_{ij}$ ，GloVe模型拟合的是对称的 $\log x_{ij}$ 。因此，任意词的中心词向量和背景词向量在GloVe模型中是等价的。但由于初始化值的不同，同一个词最终学习到的两组词向量可能不同。**当学习得到所有词向量以后，GloVe模型使用中心词向量与背景词向量之和作为该词的最终词向量。**

### 9.5.2 从条件概率比值理解GloVe模型

我们还可以从另外一个角度来理解GloVe模型。沿用本节前面的符号， $P(w_j | w_i)$ 表示数据集中以 $w_i$ 为中心词生成背景词 $w_j$ 的条件概率，并记作 $p_{ij}$ 。作为源于某大型语料库的真实例子，以下列举了两组分别以“ice”（冰）和“steam”（蒸汽）为中心词的条件概率以及它们之间的比值：

$w_k$	$p_1 = P(w_k \mid \text{"ice"})$	$p_2 = P(w_k \mid \text{"steam"})$	$p_1/p_2$
"solid"	0.00019	0.000022	8.9
"gas"	0.000066	0.00078	0.085
"water"	0.003	0.0022	1.36
"fashion"	0.000017	0.000018	0.96

我们可以观察到以下现象。

- 对于与“ice”相关而与“steam”不相关的词 $w_k$ , 如 $w_k = \text{"solid"}$  (固体), 我们期望条件概率比值较大, 如上表最后一行中的值8.9;
- 对于与“ice”不相关而与“steam”相关的词 $w_k$ , 如 $w_k = \text{"gas"}$  (气体), 我们期望条件概率比值较小, 如上表最后一行中的值0.085;
- 对于与“ice”和“steam”都相关的词 $w_k$ , 如 $w_k = \text{"water"}$  (水), 我们期望条件概率比值接近1, 如上表最后一行中的值1.36;
- 对于与“ice”和“steam”都不相关的词 $w_k$ , 如 $w_k = \text{"fashion"}$  (时尚), 我们期望条件概率比值接近1, 如上表最后一行中的值0.96。

由此可见, 条件概率比值能比较直观地表达词与词之间的关系。我们可以构造一个词向量函数使它能有效拟合条件概率比值。我们知道, 任意一个这样的比值需要3个词 $w_i$ 、 $w_j$ 和 $w_k$ 。以 $w_i$ 作为中心词的条件概率比值为 $p_{ij}/p_{ik}$ 。我们可以找一个函数, 它使用词向量来拟合这个条件概率比值

$$f(\mathbf{u}_j, \mathbf{u}_k, \mathbf{v}_i) \approx \frac{p_{ij}}{p_{ik}}$$

这里函数 $f$ 可能的设计并不唯一, 我们只需考虑一种较为合理的可能性。注意到条件概率比值是一个标量, 我们可以将 $f$ 限制为一个标量函数:  $f(\mathbf{u}_j, \mathbf{u}_k, \mathbf{v}_i) = f((\mathbf{u}_j - \mathbf{u}_k)^\top \mathbf{v}_i)$ 。交换索引 $j$ 和 $k$ 后可以看到函数 $f$ 应该满足 $f(x)f(-x) = 1$ , 因此一种可能是 $f(x) = \exp(x)$ , 于是

$$f(\mathbf{u}_j, \mathbf{u}_k, \mathbf{v}_i) = \frac{\exp(\mathbf{u}_j^\top \mathbf{v}_i)}{\exp(\mathbf{u}_k^\top \mathbf{v}_i)} \approx \frac{p_{ij}}{p_{ik}}$$

满足最右边约等号的一种可能是 $\exp(\mathbf{u}_j^\top \mathbf{v}_i) \approx \alpha p_{ij}$ , 这里 $\alpha$ 是一个常数。考虑到 $p_{ij} = x_{ij}/x_i$ , 取对数后 $\mathbf{u}_j^\top \mathbf{v}_i \approx \log \alpha + \log x_{ij} - \log x_i$ 。我们使用额外的偏差项来拟合 $-\log \alpha + \log x_i$ , 例如, 中心词偏差项 $b_i$ 和背景词偏差项 $c_j$ :

$$\mathbf{u}_j^\top \mathbf{v}_i + b_i + c_j \approx \log(x_{ij})$$

对上式左右两边取平方误差并加权, 我们可以得到GloVe模型的损失函数。更加详细的计算过程参见本章附录。

### 小结:

- 在有些情况下, 交叉熵损失函数有劣势。GloVe模型采用了平方损失, 并通过词向量拟合预先基于整个数据集计算得到的全局统计信息。
- 任意词的中心词向量和背景词向量在GloVe模型中是等价的。

## 9.6 求近义词和类比词

在9.3节 (word2vec的实现) 中, 我们在小规模数据集上训练了一个word2vec词嵌入模型, 并通过词向量的余弦相似度搜索近义词。实际中, 在大规模语料上预训练的词向量常常可以应用到下游自然语言处理任务中。本节将演示如何用这些预训练的词向量来求近义词和类比词。我们还将在后面两节中继续应用预训练的词向量。

## 9.6.1 使用预训练的词向量

基于PyTorch的关于自然语言处理的常用包有官方的[torchtext](#)以及第三方的[pytorch-nlp](#)等等。你可以使用 pip 很方便地安装它们，例如执行以下命令行：

```
1 | ! pip install torchtext==0.4.0
```

本节我们使用torchtext进行练习。下面查看它目前提供的预训练词嵌入的名称。

```
1 | In [1]:  
2 |     import torch  
3 |     import torchtext.vocab as vocab  
4 |     vocab.pretrained_aliases.keys()  
5 | Out [1]:  
6 |     dict_keys(['charngram.100d', 'fasttext.en.300d',  
7 |                 'fasttext.simple.300d', 'glove.42B.300d',  
8 |                 'glove.840B.300d', 'glove.twitter.27B.25d',  
9 |                 'glove.twitter.27B.50d', 'glove.twitter.27B.100d',  
10 |                'glove.twitter.27B.200d', 'glove.6B.50d',  
11 |                'glove.6B.100d', 'glove.6B.200d', 'glove.6B.300d'])
```

下面查看该 glove 词嵌入提供了哪些预训练的模型。每个模型的词向量维度可能不同，或是在不同数据集上预训练得到的。

```
1 | In [2]:  
2 |     [key for key in vocab.pretrained_aliases.keys()  
3 |         if 'glove' in key]  
4 | Out [2]:  
5 |     ['glove.42B.300d',  
6 |      'glove.840B.300d',  
7 |      'glove.twitter.27B.25d',  
8 |      'glove.twitter.27B.50d',  
9 |      'glove.twitter.27B.100d',  
10 |     'glove.twitter.27B.200d',  
11 |     'glove.6B.50d',  
12 |     'glove.6B.100d',  
13 |     'glove.6B.200d',  
14 |     'glove.6B.300d']
```

预训练的GloVe模型的命名规范大致是“模型.（数据集.）数据集词数.词向量维度”。更多信息可以参考GloVe和fastText的项目网站。下面我们使用基于维基百科子集预训练的50维GloVe词向量。第一次创建预训练词向量实例时会自动下载相应的词向量到 cache 指定文件夹（默认为 `.vector_cache`），因此需要联网。

```
1 | In [3]:  
2 |     cache_dir = 'data/glove'  
3 |     glove = vocab.Glove(name='6B', dim=50, cache=cache_dir)
```

返回的实例主要有以下三个属性：

- `stoi`: 词到索引的字典；
- `itos`: 一个列表，索引到词的映射；
- `vectors`: 词向量。

打印词典大小。其中含有40万个词。

```
1 In [4]:  
2     print('一共包含%d个词。' % len(glove.stoi))  
3 Out [4]:  
4     一共包含400000个词。
```

我们可以通过词来获取它在词典中的索引，也可以通过索引获取词。

```
1 In [5]:  
2     glove.stoi['beautiful'], glove.itos[3366]  
3 Out [5]:  
4     (3366, 'beautiful')
```

## 9.6.2 应用预训练词向量

下面我们以GloVe模型为例，展示预训练词向量的应用。

### 1. 求近义词

这里重新实现 9.3 节 (word2vec的实现) 中介绍过的使用余弦相似度来搜索近义词的算法。为了在求类比词时重用其中的求  $k$  近邻 ( $k$ -nearest neighbors) 的逻辑，我们将这部分逻辑单独封装在 `knn` 函数中。

```
1 In [6]:  
2     def knn(w, x, k):  
3         # 添加的1e-9是为了数值稳定性  
4         cos = torch.matmul(w, x.view((-1,))) / (br/>5             (torch.sum(w*w, dim=1)+1e-9).sqrt() * torch.sum(x*x).sqrt())  
6         _, topk = torch.topk(cos, k=k)  
7         topk = topk.cpu().numpy()  
8         return topk, [cos[i].item() for i in topk]
```

然后，我们通过预训练词向量实例 `embed` 来搜索近义词。

```
1 In [7]:  
2     def get_similar_tokens(query_token, k, embed):  
3         topk, cos = knn(embed.vectors,  
4                           embed.vectors[embed.stoi[query_token]],  
5                           k+1)  
6         # 除去输入词  
7         for i, c in zip(topk[1:], cos[1:]):  
8             print('cosine sim=% .3f: %s' % (c, (embed.itos[i])))
```

已创建的预训练词向量实例 `glove_6b50d` 的词典中含40万个词，除去输入词，我们从中搜索与“chip”语义最相近的3个词。

```
1 In [8]:  
2     get_similar_tokens('chip', 3, glove)  
3 Out [8]:  
4     cosine sim=0.856: chips  
5     cosine sim=0.749: intel  
6     cosine sim=0.749: electronics
```

接下来查找“baby”和“beautiful”的近义词。

```
1 In [9]:  
2     get_similar_tokens('baby', 3, glove)  
3 Out [9]:  
4     cosine sim=0.839: babies  
5     cosine sim=0.800: boy  
6     cosine sim=0.792: girl  
7 In [10]:  
8     get_similar_tokens('beautiful', 3, glove)  
9 Out [10]:  
10    cosine sim=0.921: lovely  
11    cosine sim=0.893: gorgeous  
12    cosine sim=0.830: wonderful
```

## 2. 求类比词

除了求近义词以外，我们还可以使用预训练词向量求词与词之间的类比关系。例如，“man”（男人）：“woman”（女人） :: “son”（儿子）：“daughter”（女儿）是一个类比例子：“man”之于“woman”相当于“son”之于“daughter”。求类比词问题可以定义为：对于类比关系中的4个词  $a : b :: c : d$ ，给定前3个词  $a$ 、 $b$  和  $c$ ，求  $d$ 。设词  $w$  的词向量为  $\text{vec}(w)$ 。求类比词的思路是，搜索与  $\text{vec}(c) + \text{vec}(b) - \text{vec}(a)$  的结果向量最相似的词向量。

```
1 In [11]:  
2     def get_analogy(token_a, token_b, token_c, embed):  
3         vecs = [embed.vectors[embed.stoi[t]] for  
4                 t in [token_a, token_b, token_c]]  
5         x = vecs[1] - vecs[0] + vecs[2]  
6         topk, cos = knn(embed.vectors, x, 1)  
7         return embed.itos[topk[0]]
```

验证一下“男-女”类比。

```
1 In [12]:  
2     get_analogy('man', 'woman', 'son', glove)  
3 Out [12]:  
4     'daughter'
```

“首都-国家”类比：“beijing”（北京）之于“china”（中国）相当于“tokyo”（东京）之于什么？答案应该是“japan”（日本）。

```
1 In [13]:  
2     get_analogy('beijing', 'china', 'tokyo', glove)  
3 Out [13]:  
4     'japan'
```

“形容词-形容词最高级”类比：“bad”（坏的）之于“worst”（最坏的）相当于“big”（大的）之于什么？答案应该是“biggest”（最大的）。

```
1 In [14]:  
2     get_analogy('bad', 'worst', 'big', glove)  
3 Out [14]:  
4     'biggest'
```

“动词一般时-动词过去时”类比：“do”（做）之于“did”（做过）相当于“go”（去）之于什么？答案应该是“went”（去过）。

```
1 In [15]:  
2     get_analogy('do', 'did', 'go', glove)  
3 Out [15]:  
4     'went'
```

### 小结：

- 在大规模语料上预训练的词向量常常可以应用于下游自然语言处理任务中。
- 可以应用预训练的词向量求近义词和类比词。

## 9.7 文本情感分类：使用循环神经网络

文本分类是自然语言处理的一个常见任务，它把一段不定长的文本序列变换为文本的类别。本节关注它的一个子问题：使用文本情感分类来分析文本作者的情绪。这个问题也叫情感分析，并有着广泛的应用。例如，我们可以分析用户对产品的评论并统计用户的满意度，或者分析用户对市场行情的情绪并用以预测接下来的行情。

同搜索近义词和类比词一样，文本分类也属于词嵌入的下游应用。在本节中，我们将应用预训练的词向量和含多个隐藏层的双向循环神经网络，来判断一段不定长的文本序列中包含的是正面还是负面的情绪。

在实验开始前，导入所需的包或模块。

```
1 In [1]:  
2     import collections  
3     import os  
4     import random  
5     import tarfile  
6     import torch  
7     from torch import nn  
8     import torchtext.vocab as vocab  
9     import torch.utils.data as Data  
10    import sys  
11    import d2lzh as d2l  
12    os.environ['CUDA_VISIBLE_DEVICES'] = '1'  
13    device = torch.device('cuda' if torch.cuda.is_available()  
14                      else 'cpu')  
15    DATA_ROOT = 'data'
```

### 9.7.1 文本情感分类数据集

我们使用斯坦福的IMDb数据集 (Stanford's Large Movie Review Dataset) 作为文本情感分类的数据集。这个数据集分为训练和测试用的两个数据集，分别包含25,000条从IMDb下载的关于电影的评论。在每个数据集中，标签为“正面”和“负面”的评论数量相等。

#### 1. 读取数据集

首先[下载](#)这个数据集到 `DATA_ROOT` 路径下，然后解压。

```

1 In [2]:
2     fname = os.path.join(DATA_ROOT, 'aclImdb_v1.tar.gz')
3     if not os.path.exists(os.path.join(DATA_ROOT, 'aclImdb')):
4         print('从压缩包解压...')
5         with tarfile.open(fname, 'r') as f:
6             f.extractall(DATA_ROOT)
7 Out [2]:
8     从压缩包解压...

```

接下来，读取训练数据集和测试数据集。每个样本是一条评论及其对应的标签：1表示“正面”，0表示“负面”。

```

1 In [3]:
2     from tqdm import tqdm
3     def read_imdb(folder='train', data_root='data/aclImdb'):
4         data = []
5         for label in ['pos', 'neg']:
6             folder_name = os.path.join(data_root, folder, label)
7             # 读取路径下所有文件
8             for file in tqdm(os.listdir(folder_name)):
9                 with open(os.path.join(folder_name, file), 'rb') as f:
10                     review = f.read().decode('utf-8').replace(
11                         '\n', '')
12                     data.append([review, 1 if label=='pos' else 0])
13         random.shuffle(data)
14         return data
15 train_data, test_data = read_imdb('train'), read_imdb('test')

```

## 2. 预处理数据集

我们需要对每条评论做分词，从而得到分好词的评论。这里定义的 `get_tokenized_imdb` 函数使用最简单的方法：基于空格进行分词。

```

1 In [4]:
2     def get_tokenized_imdb(data):
3         """
4             data: list of [string, label]
5         """
6         def tokenizer(text):
7             return [tok.lower() for tok in text.split(' ')]
8         return [tokenizer(review) for review, _ in data]

```

现在，我们可以根据分好词的训练数据集来创建词典了。我们在这里过滤掉了出现次数少于5的词。

```

1 In [5]:
2     def get_vocab_imdb(data):
3         tokenized_data = get_tokenized_imdb(data)
4         counter = collections.Counter(
5             [tk for st in tokenized_data for tk in st])
6         return Vocab.Vocab(counter, min_freq=5)
7     vocab = get_vocab_imdb(train_data)
8     '# words in vocab:', len(vocab)
9 Out [5]:
10    '# words in vocab:', 46152

```

因为每条评论长度不一致所以不能直接组合成小批量，我们定义 `preprocess_imdb` 函数对每条评论进行分词，并通过词典转换成词索引，然后通过截断或者补0来将每条评论长度固定成500。

```
1 In [6]:  
2     def preprocess_imdb(data, vocab):  
3         # 将每条评论通过截断或者补0，使得长度变成500  
4         max_l = 500  
5         def pad(x):  
6             return x[:max_l] if len(x)>max_l else x+[0]*(max_l-len(x))  
7         tokenized_data = get_tokenized_imdb(data)  
8         features = torch.tensor(  
9             [pad([vocab.stoi[word] for word in words])  
10                for words in tokenized_data])  
11         labels = torch.tensor([score for _, score in data])  
12         return features, labels
```

### 3. 创建数据迭代器

现在，我们创建数据迭代器。每次迭代将返回一个小批量的数据。

```
1 In [7]:  
2     batch_size = 64  
3     train_set = Data.TensorDataset(*preprocess_imdb(train_data, vocab))  
4     test_set = Data.TensorDataset(*preprocess_imdb(test_data, vocab))  
5     train_iter = Data.DataLoader(train_set, batch_size, shuffle=True)  
6     test_iter = Data.DataLoader(test_set, batch_size)
```

打印第一个小批量数据的形状以及训练集中小批量的个数。

```
1 In [8]:  
2     for X, y in train_iter:  
3         print('X', X.shape, 'y', y.shape)  
4         break  
5     '#batches:', len(train_iter)  
6 Out[8]:  
7     X torch.Size([64, 500]) y torch.Size([64])  
8     '#batches:', 391
```

### 9.7.2 使用循环神经网络的模型

在这个模型中，每个词先通过嵌入层得到特征向量。然后，我们使用双向循环神经网络对特征序列进一步编码得到序列信息。最后，我们将编码的序列信息通过全连接层变换为输出。具体来说，我们可以将双向长短期记忆在最初时间步和最终时间步的隐藏状态连结，作为特征序列的表征传递给输出层分类。在下面实现的 BiRNN 类中，`Embedding` 实例即嵌入层，`LSTM` 实例即为序列编码的隐藏层，`Linear` 实例即生成分类结果的输出层。

```

10                                bidirectional=True)
11
12         # 初始时间步和最终时间步的隐藏状态作为全连接层输入
13         self.decoder = nn.Linear(4*num_hiddens, 2)
14
15     def forward(self, inputs):
16
17         # inputs的形状是（批量大小，词数）
18         # 因为LSTM需要将序列长度（seq_len）作为第一维
19         # 所以将输入转置后再提取词特征
20         # 输出形状为（词数、批量大小、词向量维度）
21         embeddings = self.embedding(inputs.permute(1, 0))
22
23         # rnn.LSTM只传入输入embeddings, 因此只返回最后一层的
24         # 隐藏层在各时间步的隐藏状态
25         # outputs形状是（词数，批量大小，2*隐层单元个数）
26         # 乘2是因为双向LSTM
27         outputs, _ = self.encoder(embeddings)
28
29         # 连结初始时间步和最终时间步的隐藏状态作为全连接层输入
30         # 它的形状为（批量大小，4*隐藏单元个数）
31         encoding = torch.cat((outputs[0], outputs[-1]), -1)
32
33         outs = self.decoder(encoding)
34
35     return outs

```

创建一个含两个隐藏层的双向循环神经网络。

```

1 In [10]:
2     embed_size, num_hiddens, num_layers = 100, 100, 2
3     net = BiRNN(vocab, embed_size, num_hiddens, num_layers)

```

## 1. 加载预训练的词向量

由于情感分类的训练数据集并不是很大，为应对过拟合，我们将直接使用在更大规模语料上预训练的词向量作为每个词的特征向量。这里，我们为词典 `vocab` 中的每个词加载100维的GloVe词向量。

```

1 In [11]:
2     glove_vocab = Vocab.Glove(
3         name='6B', dim=100, cache=os.path.join(DATA_ROOT, 'glove'))

```

然后，我们将用这些词向量作为评论中每个词的特征向量。注意，预训练词向量的维度需要与创建的模型中的嵌入层输出大小 `embed_size` 一致。此外，在训练中我们不再更新这些词向量。

```

1 In [12]:
2     def load_pretrained_embedding(words, pretrained_vocab):
3         """
4             从预训练好的vocab中提取出words对应的词向量
5         """
6
7         embed = torch.zeros(
8             len(words), pretrained_vocab.vectors[0].shape[0])
9         oov_count = 0
10        for i, word in enumerate(words):
11            try:
12                idx = pretrained_vocab.stoi[word]
13                embed[i, :] = pretrained_vocab.vectors[idx]
14            except KeyError:
15                oov_count += 1
16
17            # 没有索引的单词个数
18            if oov_count > 0:
19                print('There are %d oov words.' % oov_count)
20
21        return embed

```

```
19     net.embedding.weight.data.copy_(
20         load_pretrained_embedding(vocab.itos, glove_vocab))
21     # 直接加载预训练好的, 所以不需要更新
22     net.embedding.weight.requires_grad = False
23 Out [12]:
24     There are 21202 oov words.
```

## 2. 训练模型

这时候就可以开始训练模型了。

```
1 In [13]:
2     lr, num_epochs = 0.01, 5
3     # 要过滤掉不计算梯度的embedding参数
4     optimizer = torch.optim.Adam(
5         filter(lambda p: p.requires_grad, net.parameters()), lr=lr)
6     loss = nn.CrossEntropyLoss()
7     d2l.train(train_iter, test_iter, net, loss,
8             optimizer, device, num_epochs)
9 Out [13]:
10    Let's use 2 GPUs!
11    training on cuda
12    epoch 1, loss 0.6009, train acc 0.675, test acc 0.794, time 103.1 sec
13    epoch 2, loss 0.2098, train acc 0.811, test acc 0.838, time 96.3 sec
14    epoch 3, loss 0.1183, train acc 0.844, test acc 0.851, time 99.5 sec
15    epoch 4, loss 0.0772, train acc 0.870, test acc 0.858, time 98.7 sec
16    epoch 5, loss 0.0542, train acc 0.890, test acc 0.853, time 99.4 sec
```

最后, 定义预测函数。

```
1 In [14]:
2     def predict_sentiment(net, vocab, sentence):
3         """
4             sentence是词语的列表
5         """
6         device = list(net.parameters())[0].device
7         sentence = torch.tensor(
8             [vocab.stoi[word] for word in sentence], device=device)
9         label = torch.argmax(net(sentence.view((1, -1))), dim=1)
10        return 'positive' if label.item()==1 else 'negative'
```

下面使用训练好的模型对两个简单句子的情感进行分类。

```
1 In [15]:
2     predict_sentiment(net, vocab, ['this', 'movie', 'is',
3                                     'so', 'great'])
4 Out [15]:
5     'positive'
6 In [16]:
7     predict_sentiment(net, vocab, ['this', 'movie', 'is',
8                                     'so', 'bad'])
9 Out [16]:
10    'negative'
```

小结:

- 文本分类把一段不定长的文本序列变换为文本的类别。它属于词嵌入的下游应用。
- 可以应用预训练的词向量和循环神经网络对文本的情感进行分类。

## 9.8 文本情感分类：使用卷积神经网络 (textCNN)

在“卷积神经网络”一章中我们探究了如何使用二维卷积神经网络来处理二维图像数据。在之前的语言模型和文本分类任务中，我们将文本数据看作是只有一个维度的时间序列，并很自然地使用循环神经网络来表征这样的数据。其实，我们也可以将文本当作一维图像，从而可以用一维卷积神经网络来捕捉临近词之间的关联。本节将介绍将卷积神经网络应用到文本分析的开创性工作之一：textCNN。

首先导入实验所需的包和模块。

```

1 In [1]:
2     import os
3     import torch
4     from torch import nn
5     import torchtext.vocab as Vocab
6     import torch.nn.functional as F
7     import sys
8     import d2lzh as d2l
9     os.environ['CUDA_VISIBLE_DEVICES'] = '1'
10    device = torch.device(
11        'cuda' if torch.cuda.is_available() else 'cpu')
12    DATA_ROOT = 'data'
```

### 9.8.1 一维卷积层

在介绍模型前我们先来解释一维卷积层的工作原理。与二维卷积层一样，一维卷积层使用一维的互相关运算。在一维互相关运算中，卷积窗口从输入数组的最左方开始，按从左往右的顺序，依次在输入数组上滑动。当卷积窗口滑动到某一位置时，**窗口中的输入子数组与核数组按元素相乘并求和**，得到输出数组中相应位置的元素。如下图所示，输入是一个宽为7的一维数组，核数组的宽为2。可以看到输出的宽度为 $7 - 2 + 1 = 6$ ，且第一个元素是由输入的最左边的宽为2的子数组与核数组按元素相乘后再相加得到的： $0 \times 1 + 1 \times 2 = 2$ 。



下面我们将一维互相关运算实现在 `corr1d` 函数里。它接受输入数组 `X` 和核数组 `K`，并输出数组

`Y`。

```

1 In [2]:
2     def corr1d(X, K):
3         w = K.shape[0]
4         Y = torch.zeros(X.shape[0] - w + 1)
5         for i in range(Y.shape[0]):
6             Y[i] = (X[i:i+w]*K).sum()
7         return Y
```

让我们复现上图中一维互相关运算的结果。

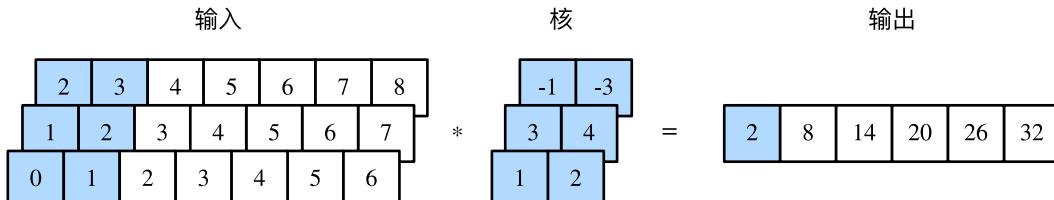
```

1 In [3]:
2     x = torch.tensor([0, 1, 2, 3, 4, 5, 6])
3     K = torch.tensor([1, 2])
4     corr1d(x, K)
5 Out [3]:
6     tensor([ 2.,  5.,  8., 11., 14., 17.])

```

多输入通道的一维互相关运算也与多输入通道的二维互相关运算类似：**在每个通道上，将核与相应的输入做一维互相关运算，并将通道之间的结果相加得到输出结果。**下图展示了含3个输入通道的一维互相关运算，其中阴影部分为第一个输出元素及其计算所使用的输入和核数组元素：

$$0 \times 1 + 1 \times 2 + 1 \times 3 + 2 \times 4 + 2 \times (-1) + 3 \times (-3) = 2.$$



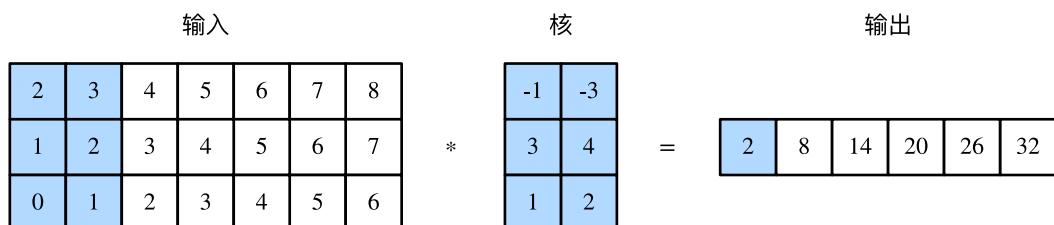
让我们复现上图中多输入通道的一维互相关运算的结果。

```

1 In [4]:
2     def corr1d_multi_in(x, K):
3         # 首先沿着X和K的第0维（通道维）遍历并计算一维互相关结果
4         # 然后将所有结果堆叠起来沿着第0维累加
5         return torch.stack(
6             [corr1d(x, k) for x, k in zip(x, K)]).sum(dim=0)
7     x = torch.tensor([
8         [0, 1, 2, 3, 4, 5, 6],
9         [1, 2, 3, 4, 5, 6, 7],
10        [2, 3, 4, 5, 6, 7, 8]
11    ])
12    K = torch.tensor([
13        [1, 2],
14        [3, 4],
15        [-1, -3]
16    ])
17    corr1d_multi_in(x, K)
18 Out [4]:
19     tensor([ 2.,  8., 14., 20., 26., 32.])

```

由二维互相关运算的定义可知，**多输入通道的一维互相关运算可以看作单输入通道的二维互相关运算**。如下图所示，我们也可以将上图中多输入通道的一维互相关运算以等价的单输入通道的二维互相关运算呈现。这里核的高等于输入的高。下图中的阴影部分为第一个输出元素及其计算所使用的输入和核数组元素： $2 \times (-1) + 3 \times (-3) + 1 \times 3 + 2 \times 4 + 0 \times 1 + 1 \times 2 = 2$ 。



上图中的输出都只有一个通道。我们在4.3节（多输入通道和多输出通道）一节中介绍了如何在二维卷积层中指定多个输出通道。类似地，我们也可以在一维卷积层指定多个输出通道，从而拓展卷积层中的模型参数。

## 9.8.2 时序最大池化层

类似地，我们有一维池化层。textCNN中使用的时序最大池化（max-over-time pooling）层实际上对应**一维全局最大池化层**：假设输入包含多个通道，各通道由不同时间步上的数值组成，各通道的输出即该通道所有时间步中最大的数值。因此，时序最大池化层的输入在各个通道上的时间步数可以不同。

为提升计算性能，我们常常将不同长度的时序样本组成一个小批量，并通过在较短序列后附加特殊字符（如0）令批量中各时序样本长度相同。这些人为添加的特殊字符当然是无意义的。**由于时序最大池化的主要目的是抓取时序中最重要的特征，它通常能使模型不受人为添加字符的影响。**

由于PyTorch没有自带全局的最大池化层，所以类似4.8节我们可以通过普通的池化来实现全局池化。

```
1 In [5]:  
2     class GlobalMaxPool1d(nn.Module):  
3         def __init__(self):  
4             super(GlobalMaxPool1d, self).__init__()  
5         def forward(self, x):  
6             # x shape: (batch_size, channel, seq_len)  
7             # return shape: (batch_size, channel, 1)  
8             return F.max_pool1d(x, kernel_size=x.shape[2])
```

## 9.8.3 读取和预处理IMDb数据集

我们依然使用和上一节中相同的IMDb数据集做情感分析。以下读取和预处理数据集的步骤与上一节中的相同。

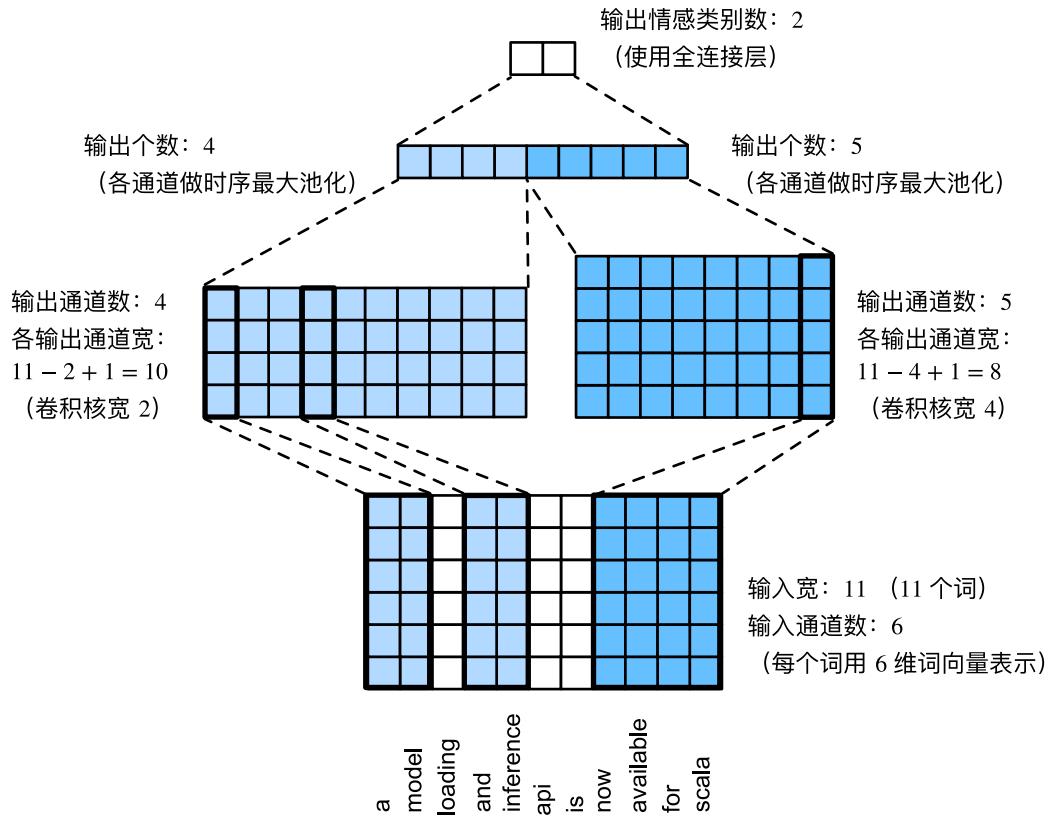
```
1 In [6]:  
2     import torch.utils.data as Data  
3     batch_size = 64  
4     train_data = d2l.read_imdb(  
5         'train',  
6         data_root=os.path.join(DATA_ROOT, 'aclImdb'))  
7     test_data = d2l.read_imdb(  
8         'test',  
9         data_root=os.path.join(DATA_ROOT, 'aclImdb'))  
10    vocab = d2l.get_vocab_imdb(train_data)  
11    train_set = Data.TensorDataset(  
12        *d2l.preprocess_imdb(train_data, vocab))  
13    test_set = Data.TensorDataset(  
14        *d2l.preprocess_imdb(test_data, vocab))  
15    train_iter = Data.DataLoader(train_set, batch_size,  
16                                shuffle=True)  
17    test_iter = Data.DataLoader(test_set, batch_size)
```

## 9.8.4 textCNN模型

textCNN模型主要使用了一维卷积层和时序最大池化层。假设输入的文本序列由 $n$ 个词组成，每个词用 $d$ 维的词向量表示。那么输入样本的宽为 $n$ ，高为1，输入通道数为 $d$ 。textCNN的计算主要分为以下几步。

1. 定义多个一维卷积核，并使用这些卷积核对输入分别做卷积计算。宽度不同的卷积核可能会捕捉到不同个数的相邻词的相关性。
2. 对输出的所有通道分别做时序最大池化，再将这些通道的池化输出值连结为向量。
3. 通过全连接层将连结后的向量变换为有关各类别的输出。这一步可以使用丢弃层应对过拟合。

下图用一个例子解释了textCNN的设计。这里的输入是一个有11个词的句子，每个词用6维词向量表示。因此输入序列的宽为11，输入通道数为6。给定2个一维卷积核，核宽分别为2和4，输出通道数分别设为4和5。因此，一维卷积计算后，4个输出通道的宽为 $11 - 2 + 1 = 10$ ，而其他5个通道的宽为 $11 - 4 + 1 = 8$ 。尽管每个通道的宽不同，我们依然可以对各个通道做时序最大池化，并将9个通道的池化输出连结成一个9维向量。最终，使用全连接将9维向量变换为2维输出，即正面情感和负面情感的预测。



下面我们来实现textCNN模型。与上一节相比，除了用一维卷积层替换循环神经网络外，这里我们还使用了两个嵌入层，一个的权重固定，另一个则参与训练。

```

1 class TextCNN(nn.Module):
2     def __init__(self, vocab, embed_size, kernel_sizes, num_channels):
3         super(TextCNN, self).__init__()
4         self.embedding = nn.Embedding(len(vocab), embed_size)
5         # 不参与训练的嵌入层
6         self.constant_embedding = nn.Embedding(len(vocab), embed_size)
7         self.dropout = nn.Dropout(0.5)
8         self.decoder = nn.Linear(sum(num_channels), 2)
9         # 时序最大化层没有权重，所以可以共用一个实例
10        self.pool = GlobalMaxPool1d()
11        # 创建多个一维卷积层
12        self.convs = nn.ModuleList()
13        for c, k in zip(num_channels, kernel_sizes):
14            self.convs.append(nn.Conv1d(in_channels=2*embed_size,
15                                      out_channels=c,
16                                      kernel_size=k))
17    def forward(self, inputs):
18        # 将两个形状是(批量大小,词数,词向量维度)的嵌入层的输出按词向量连结
19        # (batch, seq_len, 2*embed_size)
20        embeddings = torch.cat((self.embedding(inputs),
21                               self.constant_embedding(inputs)), dim=2)
22        # 根据Conv1D要求的输入格式，将词向量维，即一维卷积层的通道维
23        # 即词向量那一维变换到前一维

```

```

24     embeddings = embeddings.permute(0, 2, 1)
25     # 对于每个一维卷积，在时序最大池化后会得到一个形状为
26     # (批量大小,通道大小,1)的Tensor。使用flatten函数去掉最后一维,
27     # 然后在通道维上连结
28     encoding = torch.cat(
29         [self.pool(F.relu(conv(embeddings))).squeeze(-1)
30          for conv in self.convs],
31         dim=1
32     )
33     # 应用丢弃法后使用全连接层得到输出
34     outputs = self.decoder(self.dropout(encoding))
35     return outputs

```

创建一个TextCNN实例。它有3个卷积层，它们的核宽分别为3、4和5，输出通道数均为100。

```

1 In [7]:
2     embed_size = 100
3     kernel_size = [3, 4, 5]
4     nums_channels = [100, 100, 100]
5     net = TextCNN(vocab, embed_size, kernel_size, nums_channels)

```

## 1. 加载预训练的词向量

同上一节一样，加载预训练的100维GloVe词向量，并分别初始化嵌入层embedding和constant\_embedding，前者参与训练，而后者权重固定。

```

1 In [8]:
2     glove_vocab = Vocab.Glove(name='6B', dim=100,
3                               cache=os.path.join(DATA_ROOT, 'glove'))
4     net.embedding.weight.data.copy_(
5         d2l.load_pretrained_embedding(vocab.itos, glove_vocab))
6     net.constant_embedding.weight.data.copy_(
7         d2l.load_pretrained_embedding(vocab.itos, glove_vocab))
8     net.constant_embedding.weight.requires_grad = False
9 Out [8]:
10    There are 21202 oov words.

```

## 2. 训练模型

现在就可以训练模型了。

```

1 In [9]:
2     lr, num_epochs = 0.001, 5
3     optimizer = torch.optim.Adam(
4         filter(lambda p: p.requires_grad, net.parameters()), lr=lr)
5     loss = nn.CrossEntropyLoss()
6     d2l.train(train_iter, test_iter, net, loss, optimizer,
7                device, num_epochs)
8 Out [9]:
9     Let's use 2 GPUs!
10    training on cuda
11    epoch 1, loss 0.4848, train acc 0.755, test acc 0.809, time 26.7 sec
12    epoch 2, loss 0.1607, train acc 0.861, test acc 0.870, time 21.1 sec
13    epoch 3, loss 0.0685, train acc 0.919, test acc 0.876, time 21.3 sec
14    epoch 4, loss 0.0294, train acc 0.958, test acc 0.864, time 21.0 sec
15    epoch 5, loss 0.0127, train acc 0.978, test acc 0.863, time 21.1 sec

```

下面，我们使用训练好的模型对两个简单句子的情感进行分类。

```
1 In [10]:  
2     d2l.predict_sentiment(net, vocab, ['this', 'movie', 'is', 'so', 'great'])  
3 Out [10]:  
4     'positive'  
5 In [11]:  
6     d2l.predict_sentiment(net, vocab, ['this', 'movie', 'is', 'so', 'bad'])  
7 Out [11]:  
8     'negative'
```

### 小结：

- 可以使用一维卷积来表征时序数据。
- 多输入通道的一维互相关运算可以看作单输入通道的二维互相关运算。
- 时序最大池化层的输入在各个通道上的时间步数可以不同。
- textCNN主要使用了一维卷积层和时序最大池化层。

## 9.9 编码器-解码器 (seq2seq)

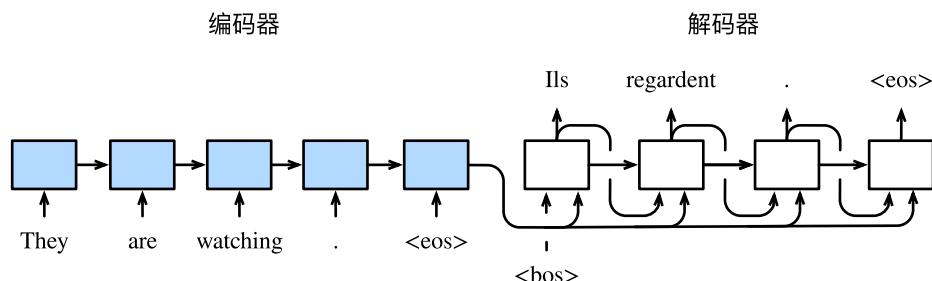
我们已经在前两节中表征并变换了不定长的输入序列。但在自然语言处理的很多应用中，输入和输出都可以是不定长序列。以机器翻译为例，输入可以是一段不定长的英语文本序列，输出可以是一段不定长的法语文本序列，例如

英语输入：“They”、“are”、“watching”、“.”

法语输出：“Ils”、“regardent”、“.”

当输入和输出都是不定长序列时，我们可以使用编码器—解码器（encoder-decoder）或者 seq2seq 模型。这两个模型本质上都用到了两个循环神经网络，分别叫做编码器和解码器。**编码器用来分析输入序列，解码器用来生成输出序列。**

下图描述了使用编码器—解码器将上述英语句子翻译成法语句子的一种方法。在训练数据集中，我们可以在每个句子后附上特殊符号“<eos>”（end of sequence）以表示序列的终止。编码器每个时间步的输入依次为英语句子中的单词、标点和特殊符号“<eos>”。**下图中使用了编码器在最终时间步的隐藏状态作为输入句子的表征或编码信息。解码器在各个时间步中使用输入句子的编码信息和上个时间步的输出以及隐藏状态作为输入。**我们希望解码器在各个时间步能正确依次输出翻译后的法语单词、标点和特殊符号“<eos>”。需要注意的是，解码器在最初时间步的输入用到了一个表示序列开始的特殊符号“<bos>”（beginning of sequence）。



接下来，我们分别介绍编码器和解码器的定义。

## 9.9.1 编码器

编码器的作用是把一个不定长的输入序列变换成一个定长的背景变量 $\mathbf{c}$ ，并在该背景变量中编码输入序列信息。常用的编码器是循环神经网络。

让我们考虑批量大小为1的时序数据样本。假设输入序列是 $x_1, \dots, x_T$ ，例如 $x_i$ 是输入句子中的第*i*个词。在时间步 $t$ ，循环神经网络将输入 $x_t$ 的特征向量 $\mathbf{x}_t$ 和上个时间步的隐藏状态 $\mathbf{h}_{t-1}$ 变换为当前时间步的隐藏状态 $\mathbf{h}_t$ 。我们可以用函数 $f$ 表达循环神经网络隐藏层的变换：

$$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1})$$

接下来，编码器通过自定义函数 $q$ 将各个时间步的隐藏状态变换为背景变量

$$\mathbf{c} = q(\mathbf{h}_1, \dots, \mathbf{h}_T)$$

例如，当选择 $q(\mathbf{h}_1, \dots, \mathbf{h}_T) = \mathbf{h}_T$ 时，背景变量是输入序列最终时间步的隐藏状态 $\mathbf{h}_T$ 。

以上描述的编码器是一个单向的循环神经网络，每个时间步的隐藏状态只取决于该时间步及之前的输入子序列。我们也可以使用双向循环神经网络构造编码器。在这种情况下，编码器每个时间步的隐藏状态同时取决于该时间步之前和之后的子序列（包括当前时间步的输入），并编码了整个序列的信息。

## 9.9.2 解码器

刚刚已经介绍，编码器输出的背景变量 $\mathbf{c}$ 编码了整个输入序列 $x_1, \dots, x_T$ 的信息。给定训练样本中的输出序列 $y_1, y_2, \dots, y_{T'}$ ，对每个时间步 $t'$ （符号与输入序列或编码器的时间步 $t$ 有区别），解码器输出 $y_{t'}$ 的条件概率将基于之前的输出序列 $y_1, \dots, y_{t'-1}$ 和背景变量 $\mathbf{c}$ ，即 $P(y_{t'} | y_1, \dots, y_{t'-1}, \mathbf{c})$ 。

为此，我们可以使用另一个循环神经网络作为解码器。在输出序列的时间步 $t'$ ，解码器将上一时间步的输出 $y_{t'-1}$ 以及背景变量 $\mathbf{c}$ 作为输入，并将它们与上一时间步的隐藏状态 $\mathbf{s}_{t'-1}$ 变换为当前时间步的隐藏状态 $\mathbf{s}_{t'}$ 。因此，我们可以用函数 $g$ 表达解码器隐藏层的变换：

$$\mathbf{s}_{t'} = g(y_{t'-1}, \mathbf{c}, \mathbf{s}_{t'-1})$$

有了解码器的隐藏状态后，我们可以使用自定义的输出层和softmax运算来计算 $P(y_{t'} | y_1, \dots, y_{t'-1}, \mathbf{c})$ ，例如，基于当前时间步的解码器隐藏状态 $\mathbf{s}_{t'}$ 、上一时间步的输出 $y_{t'-1}$ 以及背景变量 $\mathbf{c}$ 来计算当前时间步输出 $y_{t'}$ 的概率分布。

## 9.9.3 训练模型

根据最大似然估计，我们可以最大化输出序列基于输入序列的条件概率

$$\begin{aligned} P(y_1, \dots, y_{T'} | x_1, \dots, x_T) &= \prod_{t'=1}^{T'} P(y_{t'} | y_1, \dots, y_{t'-1}, x_1, \dots, x_T) \\ &= \prod_{t'=1}^{T'} P(y_{t'} | y_1, \dots, y_{t'-1}, \mathbf{c}) \end{aligned}$$

并得到该输出序列的损失

$$-\log P(y_1, \dots, y_{T'} | x_1, \dots, x_T) = -\sum_{t'=1}^{T'} \log P(y_{t'} | y_1, \dots, y_{t'-1}, \mathbf{c})$$

在模型训练中，所有输出序列损失的均值通常作为需要最小化的损失函数。在图10.8所描述的模型预测中，我们需要将解码器在上一个时间步的输出作为当前时间步的输入。与此不同，在训练中我们也可以将标签序列（训练集的真实输出序列）在上一个时间步的标签作为解码器在当前时间步的输入。这叫做强制教学（teacher forcing）。

小结：

- 编码器-解码器（seq2seq）可以输入并输出不定长的序列。
- 编码器-解码器使用了两个循环神经网络。
- 在编码器-解码器的训练中，可以采用强制教学。

## 9.10 束搜索

上一节介绍了如何训练输入和输出均为不定长序列的编码器-解码器。本节我们介绍如何使用编码器-解码器来预测不定长的序列。

上一节里已经提到，在准备训练数据集时，我们通常会在样本的输入序列和输出序列后面分别附上一个特殊符号“`<eos>`”表示序列的终止。我们在接下来的讨论中也将沿用上一节的全部数学符号。为了便于讨论，假设解码器的输出是一段文本序列。设输出文本词典 $\mathcal{Y}$ （包含特殊符号“`<eos>`”）的大小为 $|\mathcal{Y}|$ ，输出序列的最大长度为 $T'$ 。所有可能的输出序列一共有 $\mathcal{O}(|\mathcal{Y}|^{T'})$ 种。这些输出序列中所有特殊符号“`<eos>`”后面的子序列将被舍弃。

### 9.10.1 贪婪搜索

让我们先来看一个简单的解决方案：贪婪搜索（greedy search）。对于输出序列任一时间步 $t'$ ，我们从 $|\mathcal{Y}|$ 个词中搜索出条件概率最大的词

$$y_{t'} = \underset{y \in \mathcal{Y}}{\operatorname{argmax}} P(y | y_1, \dots, y_{t'-1}, c)$$

作为输出。一旦搜索出“`<eos>`”符号，或者输出序列长度已经达到了最大长度 $T'$ ，便完成输出。

我们在描述解码器时提到，基于输入序列生成输出序列的条件概率是 $\prod_{t'=1}^{T'} P(y_{t'} | y_1, \dots, y_{t'-1}, c)$ 。我们将该条件概率最大的输出序列称为最优输出序列。而**贪婪搜索的主要问题是不能保证得到最优输出序列。**

下面来看一个例子。假设输出词典里面有“`A`”“`B`”“`C`”和“`<eos>`”这4个词。下图中每个时间步下的4个数字分别代表了该时间步生成“`A`”“`B`”“`C`”和“`<eos>`”这4个词的条件概率。在每个时间步，贪婪搜索选取条件概率最大的词。因此，下图中将生成输出序列“`A`”“`B`”“`C`”“`<eos>`”。该输出序列的条件概率是 $0.5 \times 0.4 \times 0.4 \times 0.6 = 0.048$ 。

时间步	1	2	3	4
A	0.5	0.1	0.2	0.0
B	0.2	0.4	0.2	0.2
C	0.2	0.3	0.4	0.2
<code>&lt;eos&gt;</code>	0.1	0.2	0.2	0.6

接下来，观察下图演示的例子。与上图中不同，下图在时间步2中选取了条件概率第二大的词“`C`”。由于时间步3所基于的时间步1和2的输出子序列由上图中的“`A`”“`B`”变为了下图中的“`A`”“`C`”，下图中时间步3生成各个词的条件概率发生了变化。我们选取条件概率最大的词“`B`”。此时时间步4所基于的前3个时间步的输出子序列为“`A`”“`C`”“`B`”，与上图中的“`A`”“`B`”“`C`”不同。因此，下图中时间步4生成各个词的条件概率也与上图中的不同。我们发现，此时的输出序列“`A`”“`C`”“`B`”“`<eos>`”的条件概率是 $0.5 \times 0.3 \times 0.6 \times 0.6 = 0.054$ ，大于贪婪搜索得到的输出序列的条件概率。因此，贪婪搜索得到的输出序列“`A`”“`B`”“`C`”并非最优输出序列。

时间步	1	2	3	4
A	0.5	0.1	0.1	0.1
B	0.2	0.4	0.6	0.2
C	0.2	0.3	0.2	0.1
<code>&lt;eos&gt;</code>	0.1	0.2	0.1	0.6

## 9.10.2 穷举搜索

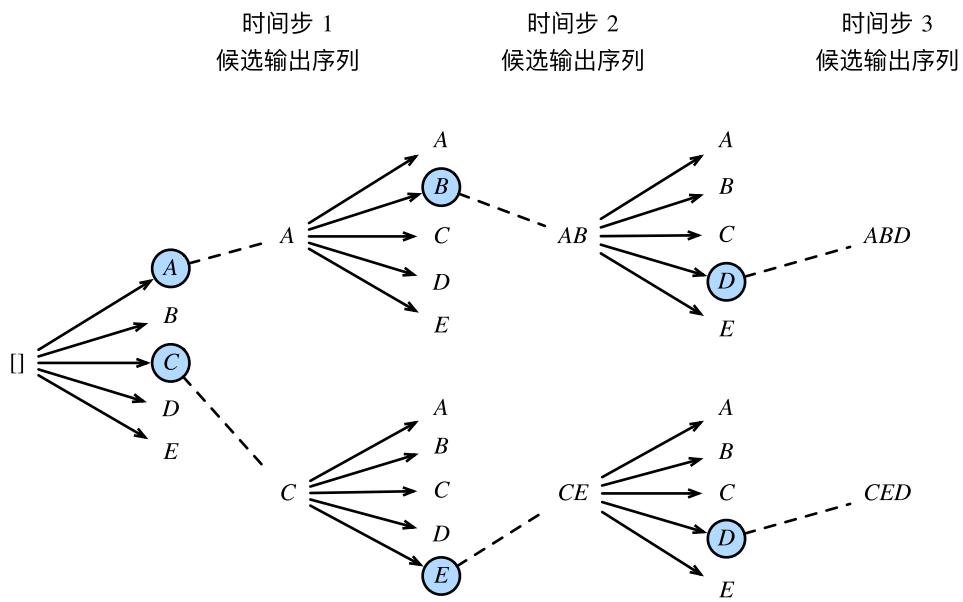
如果目标是得到最优输出序列，我们可以考虑穷举搜索（exhaustive search）：穷举所有可能的输出序列，输出条件概率最大的序列。

虽然穷举搜索可以得到最优输出序列，但它的计算开销 $\mathcal{O}(|\mathcal{Y}|^{T'})$ 很容易过大。例如，当 $|\mathcal{Y}| = 10000$ 且 $T' = 10$ 时，我们将评估 $10000^{10} = 10^{40}$ 个序列：这几乎不可能完成。而贪婪搜索的计算开销是 $\mathcal{O}(|\mathcal{Y}| T')$ ，通常显著小于穷举搜索的计算开销。例如，当 $|\mathcal{Y}| = 10000$ 且 $T' = 10$ 时，我们只需评估 $10000 \times 10 = 10^5$ 个序列。

## 9.10.3 束搜索

束搜索（beam search）是对贪婪搜索的一个改进算法。它有一个束宽（beam size）超参数。我们将它设为 $k$ 。在时间步1时，选取当前时间步条件概率最大的 $k$ 个词，分别组成 $k$ 个候选输出序列的首词。在之后的每个时间步，基于上个时间步的 $k$ 个候选输出序列，从 $k |\mathcal{Y}|$ 个可能的输出序列中选取条件概率最大的 $k$ 个，作为该时间步的候选输出序列。最终，我们从各个时间步的候选输出序列中筛选出包含特殊符号“”的序列，并将它们中所有特殊符号“”后面的子序列舍弃，得到最终候选输出序列的集合。

下图通过一个例子演示了束搜索的过程。假设输出序列的词典中只包含5个元素，即 $\mathcal{Y} = A, B, C, D, E$ ，且其中一个为特殊符号“”。设束搜索的束宽等于2，输出序列最大长度为3。在输出序列的时间步1时，假设条件概率 $P(y_1 | c)$ 最大的2个词为 $A$ 和 $C$ 。我们在时间步2时将对所有的 $y_2 \in \mathcal{Y}$ 都分别计算 $P(A, y_2 | c) = P(A | c)P(y_2 | A, c)$ 和 $P(C, y_2 | c) = P(C | c)P(y_2 | C, c)$ ，并从计算出的10个条件概率中取最大的2个，假设为 $P(A, B | c)$ 和 $P(C, E | c)$ 。那么，我们在时间步3时将对所有的 $y_3 \in \mathcal{Y}$ 都分别计算 $P(A, B, y_3 | c) = P(A, B | c)P(y_3 | A, B, c)$ 和 $P(C, E, y_3 | c) = P(C, E | c)P(y_3 | C, E, c)$ ，并从计算出的10个条件概率中取最大的2个，假设为 $P(A, B, D | c)$ 和 $P(C, E, D | c)$ 。如此一来，我们得到6个候选输出序列：（1） $A$ ；（2） $C$ ；（3） $A$ 、 $B$ ；（4） $C$ 、 $E$ ；（5） $A$ 、 $B$ 、 $D$ 和（6） $C$ 、 $E$ 、 $D$ 。接下来，我们将根据这6个序列得出最终候选输出序列的集合。



在最终候选输出序列的集合中，我们取以下分数最高的序列作为输出序列：

$$\frac{1}{L^\alpha} \log P(y_1, \dots, y_L) = \frac{1}{L^\alpha} \sum_{t'=1}^L \log P(y_{t'} | y_1, \dots, y_{t'-1}, c)$$

其中 $L$ 为最终候选序列长度， $\alpha$ 一般可选为0.75。分母上的 $L^\alpha$ 是为了惩罚较长序列在以上分数中较多的对数相加项。分析可知，束搜索的计算开销为 $\mathcal{O}(k |\mathcal{Y}| T')$ 。这介于贪婪搜索和穷举搜索的计算开销之间。此外，贪婪搜索可看作是束宽为1的束搜索。束搜索通过灵活的束宽 $k$ 来权衡计算开销和搜索质量。

## 小结：

- 预测不定长序列的方法包括贪婪搜索、穷举搜索和束搜索。
- 束搜索通过灵活的束宽来权衡计算开销和搜索质量。

## 9.11 注意力机制

在9.9节（编码器—解码器（seq2seq））里，解码器在各个时间步依赖相同的背景变量来获取输入序列信息。当编码器为循环神经网络时，背景变量来自它最终时间步的隐藏状态。

现在，让我们再次思考那一节提到的翻译例子：输入为英语序列“*They*”“*are*”“*watching*”“.”，输出为法语序列“*Ils*”“*regardent*”“.”。不难想到，解码器在生成输出序列中的每一个词时可能只需利用输入序列某一部分的信息。例如，在输出序列的时间步1，解码器可以主要依赖“*They*”“*are*”的信息来生成“*Ils*”，在时间步2则主要使用来自“*watching*”的编码信息生成“*regardent*”，最后在时间步3则直接映射句号“.”。这看上去就像是在解码器的每一时间步对输入序列中不同时间步的表征或编码信息分配不同的注意力一样。这也是注意力机制的由来。

仍然以循环神经网络为例，**注意力机制通过对编码器所有时间步的隐藏状态做加权平均来得到背景变量。解码器在每一时间步调整这些权重，即注意力权重，从而能够在不同时间步分别关注输入序列中的不同部分并编码进相应时间步的背景变量。**本节我们将讨论注意力机制是怎么工作的。

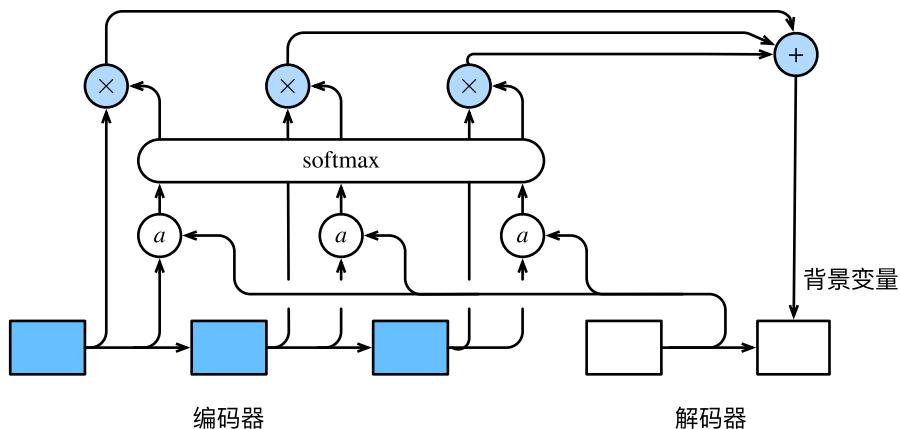
在9.9节（编码器—解码器（seq2seq））里我们区分了输入序列或编码器的索引 $t$ 与输出序列或解码器的索引 $t'$ 。该节中，解码器在时间步 $t'$ 的隐藏状态 $s_{t'} = g(y_{t'-1}, c, s_{t'-1})$ ，其中 $y_{t'-1}$ 是上一时间步 $t' - 1$ 的输出 $y_{t'-1}$ 的表征，且任一时间步 $t'$ 使用相同的背景变量 $c$ 。**但在注意力机制中，解码器的每一时间步将使用可变的背景变量。**记 $c_{t'}$ 是解码器在时间步 $t'$ 的背景变量，那么解码器在该时间步的隐藏状态可以改写为

$$s_{t'} = g(y_{t'-1}, c_{t'}, s_{t'-1})$$

这里的关键是如何计算背景变量 $c_{t'}$ 和如何利用它来更新隐藏状态 $s_{t'}$ 。下面将分别描述这两个关键点。

### 9.11.1 计算背景变量

我们先描述第一个关键点，即计算背景变量。下图描绘了注意力机制如何为解码器在时间步2计算背景变量。首先，函数 $a$ 根据解码器在时间步1的隐藏状态和编码器在各个时间步的隐藏状态计算softmax运算的输入。softmax运算输出概率分布并对编码器各个时间步的隐藏状态做加权平均，从而得到背景变量。



具体来说，令编码器在时间步 $t$ 的隐藏状态为 $h_t$ ，且总时间步数为 $T$ 。那么解码器在时间步 $t'$ 的背景变量为所有编码器隐藏状态的加权平均：

$$c_{t'} = \sum_{t=1}^T \alpha_{t't} h_t$$

其中给定 $t'$ 时，权重 $\alpha_{t't}$ 在 $t = 1, \dots, T$ 的值是一个概率分布。为了得到概率分布，我们可以使用softmax运算：

$$\alpha_{t't} = \frac{\exp(e_{t't})}{\sum_{k=1}^T \exp(e_{t'k})}, \quad t = 1, \dots, T$$

现在，我们需要定义如何计算上式中softmax运算的输入 $e_{t't}$ 。由于 $e_{t't}$ 同时取决于解码器的时间步 $t'$ 和编码器的时间步 $t$ ，我们不妨以解码器在时间步 $t' - 1$ 的隐藏状态 $s_{t'-1}$ 与编码器在时间步 $t$ 的隐藏状态 $h_t$ 为输入，并通过函数 $a$ 计算 $e_{t't}$ ：

$$e_{t't} = a(s_{t'-1}, h_t)$$

这里函数 $a$ 有多种选择，如果两个输入向量长度相同，一个简单的选择是计算它们的内积 $a(s, h) = s^\top h$ 。而最早提出注意力机制的论文则将输入连结后通过含单隐藏层的多层感知机变换：

$$a(s, h) = v^\top \tanh(W_s s + W_h h)$$

其中 $v$ 、 $W_s$ 、 $W_h$ 都是可以学习的模型参数。

### 矢量化计算

我们还可以对注意力机制采用更高效的矢量化计算。广义上，注意力机制的输入包括查询项以及一对对应的键项和值项，其中值项是需要加权平均的一组项。在加权平均中，值项的权重来自查询项以及与该值项对应的键项的计算。

在上面的例子中，**查询项为解码器的隐藏状态，键项和值项均为编码器的隐藏状态**。让我们考虑一个常见的简单情形，即编码器和解码器的隐藏单元个数均为 $h$ ，且函数 $a(s, h) = s^\top h$ 。假设我们希望根据解码器单个隐藏状态 $s_{t'-1} \in \mathbb{R}^h$ 和编码器所有隐藏状态 $h_t \in \mathbb{R}^h, t = 1, \dots, T$ 来计算背景向量 $c_{t'} \in \mathbb{R}^h$ 。我们可以将查询项矩阵 $Q \in \mathbb{R}^{1 \times h}$ 设为 $s_{t'-1}^\top$ ，并令键项矩阵 $K \in \mathbb{R}^{T \times h}$ 和值项矩阵 $V \in \mathbb{R}^{T \times h}$ 相同且第 $t$ 行均为 $h_t^\top$ 。此时，我们只需要通过矢量化计算

$$\text{softmax}(Q K^\top) V$$

即可算出转置后的背景向量 $c_{t'}^\top$ 。当查询项矩阵 $Q$ 的行数为 $n$ 时，上式将得到 $n$ 行的输出矩阵。输出矩阵与查询项矩阵在相同行上一一对应。

## 9.11.2 更新隐藏状态

现在我们描述第二个关键点，即更新隐藏状态。以门控循环单元为例，在解码器中我们可以对5.7节（门控循环单元（GRU））中门控循环单元的设计稍作修改，从而变换上一时间步 $t' - 1$ 的输出 $y_{t'-1}$ 、隐藏状态 $s_{t'-1}$ 和当前时间步 $t'$ 的含注意力机制的背景变量 $c_{t'}$ 。解码器在时间步 $t'$ 的隐藏状态为

$$s_{t'} = z_{t'} \odot s_{t'-1} + (1 - z_{t'}) \odot \tilde{s}_{t'}$$

其中的重置门、更新门和候选隐藏状态分别为

$$\begin{aligned} r_{t'} &= \sigma(W_{yr} y_{t'-1} + W_{sr} s_{t'-1} + W_{cr} c_{t'} + b_r) \\ z_{t'} &= \sigma(W_{yz} y_{t'-1} + W_{sz} s_{t'-1} + W_{cz} c_{t'} + b_z) \\ \tilde{s}_{t'} &= \tanh(W_{ys} y_{t'-1} + W_{ss} (s_{t'-1} \odot r_{t'}) + W_{cs} c_{t'} + b_s) \end{aligned}$$

其中含下标的 $W$ 和 $b$ 分别为门控循环单元的权重参数和偏差参数。

### 9.11.3 发展

本质上，注意力机制能够为表征中较有价值的部分分配较多的计算资源。这个有趣的想法自提出后得到了快速发展，特别是启发了依靠注意力机制来编码输入序列并解码出输出序列的变换器

(Transformer) 模型的设计。变换器抛弃了卷积神经网络和循环神经网络的架构。它在计算效率上比基于循环神经网络的编码器—解码器模型通常更具明显优势。含注意力机制的变换器的编码结构在后来的BERT预训练模型中得以应用并令后者大放异彩：微调后的模型在多达11项自然语言处理任务中取得了当时最先进的结果。不久后，同样是基于变换器设计的GPT-2模型于新收集的语料数据集预训练后，在7个未参与训练的语言模型数据集上均取得了当时最先进的结果。除了自然语言处理领域，注意力机制还被广泛用于图像分类、自动图像描述、唇语解读以及语音识别。

**小结：**

- 可以在解码器的每个时间步使用不同的背景变量，并对输入序列中不同时间步编码的信息分配不同的注意力。
- 广义上，注意力机制的输入包括查询项以及一一对应的键项和值项。
- 注意力机制可以采用更为高效的矢量化计算。

## 9.12 机器翻译

机器翻译是指将一段文本从一种语言自动翻译到另一种语言。因为一段文本序列在不同语言中的长度不一定相同，所以我们使用机器翻译为例来介绍编码器—解码器和注意力机制的应用。

### 9.12.1 读取和预处理数据集

我们先定义一些特殊符号。其中'<pad>' (padding) 符号用来添加在较短序列后，直到每个序列等长，而'<bos>'和'<eos>'符号分别表示序列的开始和结束。

```
1 In [1]:  
2     import collections  
3     import os  
4     import io  
5     import math  
6     import torch  
7     from torch import nn  
8     import torch.nn.functional as F  
9     import torchtext.vocab as Vocab  
10    import torch.utils.data as Data  
11    import sys  
12    import d2lzh as d2l  
13    PAD, BOS, EOS = '<pad>', '<bos>', '<eos>'  
14    os.environ['CUDA_VISIBLE_DEVICES'] = '1'  
15    device = torch.device(  
16        'cuda' if torch.cuda.is_available() else 'cpu')
```

接着定义两个辅助函数对后面读取的数据进行预处理。

```
1 In [2]:  
2     # 将一个序列中所有的词记录在all_tokens中以便之后构造词典  
3     # 然后在该序列后面添加PAD直到序列长度为max_seq_len  
4     # 然后将序列保存在all_seqs中  
5     def process_one_seq(seq_tokens, all_tokens, all_seqs, max_seq_len):  
6         all_tokens.extend(seq_tokens)  
7         seq_tokens += [EOS] + [PAD] * (max_seq_len - len(seq_tokens) - 1)  
8         all_seqs.append(seq_tokens)
```

```

9     # 使用所有的词构造词典。并将所有序列中的词变为索引后构造Tensor
10    def build_data(all_tokens, all_seqs):
11        vocab = Vocab(collections.Counter(all_tokens),
12                      specials=[PAD, BOS, EOS])
13        indices = [[vocab.stoi[w] for w in seq] for seq in all_seqs]
14        return vocab, torch.tensor(indices)

```

为了演示方便，我们在这里使用一个很小的法语—英语数据集。在这个数据集里，每一行是一对法语句子和它对应的英语句子，中间使用'\t'隔开。在读取数据时，我们在句末附上'<eos>'符号，并可能通过添加'<pad>'符号使每个序列的长度均为 max\_seq\_len。我们为法语词和英语词分别创建词典。法语词的索引和英语词的索引相互独立。

```

1 In [3]:
2     def read_data(max_seq_len):
3         # in和out分别是input和output的缩写
4         in_tokens, out_tokens, in_seqs, out_seqs = [], [], [], []
5         with io.open('data/fr-en-small.txt') as f:
6             lines = f.readlines()
7             for line in lines:
8                 in_seq, out_seq = line.rstrip().split('\t')
9                 in_seq_tokens = in_seq.split(' ')
10                out_seq_tokens = out_seq.split(' ')
11                if max(len(in_seq_tokens),
12                       len(out_seq_tokens)) > max_seq_len - 1:
13                    # 如果加上EOS后长于max_seq_len，则忽略掉此样本
14                    continue
15                process_one_seq(in_seq_tokens, in_tokens, in_seqs, max_seq_len)
16                process_one_seq(out_seq_tokens, out_tokens, out_seqs,
17                                max_seq_len)
18                in_vocab, in_data = build_data(in_tokens, in_seqs)
19                out_vocab, out_data = build_data(out_tokens, out_seqs)
20            return in_vocab, out_vocab, Data.TensorDataset(in_data, out_data)

```

将序列的最大长度设成7，然后查看读取到的第一个样本。该样本分别包含法语词索引序列和英语词索引序列。

```

1 In [4]:
2     max_seq_len = 7
3     in_vocab, out_vocab, dataset = read_data(max_seq_len)
4     dataset[0]
5 Out [4]:
6     (tensor([ 5,  4, 45,  3,  2,  0,  0]),
7      tensor([ 8,  4, 27,  3,  2,  0,  0]))

```

## 9.12.2 含注意力机制的编码器-解码器

我们将使用含注意力机制的编码器—解码器来将一段简短的法语翻译成英语。下面我们来介绍模型的实现。

### 1. 编码器

在编码器中，我们将输入语言的词索引通过词嵌入层得到词的表征，然后输入到一个多层次门控循环单元中。正如我们在5.5节（循环神经网络的简洁实现）中提到的，PyTorch的 `nn.GRU` 实例在前向计算后也会分别返回输出和最终时间步的多层次隐藏状态。其中的输出指的是最后一层的隐藏层在各个时间步的隐藏状态，并不涉及输出层计算。注意力机制将这些输出作为键项和值项。

```

1 In [5]:
2     class Encoder(nn.Module):
3         def __init__(self, vocab_size, embed_size, num_hiddens,
4                      num_layers, drop_prob=0, **kwargs):
5             super(Encoder, self).__init__(**kwargs)
6             self.embedding = nn.Embedding(vocab_size, embed_size)
7             self.rnn = nn.GRU(
8                 embed_size, num_hiddens, num_layers, dropout=drop_prob)
9             def forward(self, inputs, state):
10                 # 输入形状是(批量大小, 时间步数)
11                 # 将输出互换样本维和时间步维
12                 # (seq_len, batch_size, input_size)
13                 embedding = self.embedding(inputs.long()).permute(1, 0, 2)
14                 return self.rnn(embedding, state)
15             def begin_state(self):
16                 # 隐藏状态初始化为None时PyTorch会自动初始化为0
17                 return None

```

下面我们来创建一个批量大小为4、时间步数为7的小批量序列输入。设门控循环单元的隐藏层个数为2，隐藏单元个数为16。编码器对该输入执行前向计算后返回的输出形状为（时间步数, 批量大小, 隐藏单元个数）。门控循环单元在最终时间步的多层隐藏状态的形状为（隐藏层个数, 批量大小, 隐藏单元个数）。对于门控循环单元来说，`state`就是一个元素，即隐藏状态；如果使用长短期记忆，`state`是一个元组，包含两个元素即隐藏状态和记忆细胞。

```

1 In [6]:
2     encoder = Encoder(vocab_size=10, embed_size=8,
3                         num_hiddens=16, num_layers=2)
4     output, state = encoder(torch.zeros((4, 7)),
5                             state=encoder.begin_state())
6     # GRU的state是h, 而LSTM的是一个元组(h, c)
7     output.shape, state.shape
8 Out [6]:
9     (torch.Size([7, 4, 16]), torch.Size([2, 4, 16]))

```

## 2. 注意力机制

我们将实现9.11节（注意力机制）中定义的函数`a`：将输入连结后通过含单隐藏层的多层次感知机变换（注意力最早的样子）。其中隐藏层的输入是解码器的隐藏状态与编码器在所有时间步上隐藏状态的一一连结，且使用tanh函数作为激活函数。输出层的输出个数为1。两个`Linear`实例均不使用偏差。其中函数`a`定义里向量`v`的长度是一个超参数，即`attention_size`。

```

1 In [7]:
2     def attention_model(input_size, attention_size):
3         model = nn.Sequential(
4             nn.Linear(input_size, attention_size, bias=False),
5             nn.Tanh(),
6             nn.Linear(attention_size, 1, bias=False)
7         )
8         return model

```

注意力机制的输入包括查询项、键项和值项。设编码器和解码器的隐藏单元个数相同。这里的查询项为解码器在上一时间步的隐藏状态，形状为（批量大小, 隐藏单元个数）；键项和值项均为编码器在所有时间步的隐藏状态，形状为（时间步数, 批量大小, 隐藏单元个数）。注意力机制返回当前时间步的背景变量，形状为（批量大小, 隐藏单元个数）。

```

1 In [8]:
2     def attention_forward(model, enc_states, dec_state):
3         """
4             enc_states: (时间步数, 批量大小, 隐藏单元个数)
5             dec_state: (批量大小, 隐藏单元个数)
6         """
7             # 将解码器隐藏状态广播到和编码器隐藏状态形状相同后进行连结
8             dec_states = dec_state.unsqueeze(dim=0).expand_as(enc_states)
9             enc_and_dec_states = torch.cat((enc_states, dec_states), dim=2)
10            # 形状为(时间步数, 批量大小, 1)
11            e = model(enc_and_dec_states)
12            # 在时间步维度做softmax运算
13            alpha = F.softmax(e, dim=0)
14            # 返回背景变量
15            return (alpha*enc_states).sum(dim=0)

```

在下面的例子中，编码器的时间步数为10，批量大小为4，编码器和解码器的隐藏单元个数均为8。注意力机制返回一个小批量的背景向量，每个背景向量的长度等于编码器的隐藏单元个数。因此输出的形状为(4, 8)。

```

1 In [9]:
2     seq_len, batch_size, num_hiddens = 10, 4, 8
3     model = attention_model(2*num_hiddens, 10)
4     enc_states = torch.zeros((seq_len, batch_size, num_hiddens))
5     dec_state = torch.zeros((batch_size, num_hiddens))
6     attention_forward(model, enc_states, dec_state).shape
7 Out [9]:
8     torch.Size([4, 8])

```

### 3. 含注意力机制的解码器

我们直接将编码器在最终时间步的隐藏状态作为解码器的初始隐藏状态。这要求编码器和解码器的循环神经网络使用相同的隐藏层个数和隐藏单元个数。

在解码器的前向计算中，我们先通过刚刚介绍的注意力机制计算得到当前时间步的背景向量。由于解码器的输入来自输出语言的词索引，我们将输入通过词嵌入层得到表征，然后和背景向量在特征维连结。我们将连结后的结果与上一时间步的隐藏状态通过门控循环单元计算出当前时间步的输出与隐藏状态。最后，我们将输出通过全连接层变换为有关各个输出词的预测，形状为（批量大小，输出词典大小）。

```

1 In [10]:
2     class Decoder(nn.Module):
3         def __init__(self, vocab_size, embed_size, num_hiddens,
4                      num_layers, attention_size, drop_prob=0):
5             super(Decoder, self).__init__()
6             self.embedding = nn.Embedding(vocab_size, embed_size)
7             self.attention = attention_model(2*num_hiddens, attention_size)
8             # GRU的输入包含attention输出的c和实际输入
9             # 所以尺寸是num_hiddens+embed_size
10            self.rnn = nn.GRU(num_hiddens+embed_size, num_hiddens,
11                            num_layers, dropout=drop_prob)
12            self.out = nn.Linear(num_hiddens, vocab_size)
13        def forward(self, cur_input, state, enc_states):
14            """
15                cur_input shape: (batch, )
16                state shape: (num_layers, batch, num_hiddens)

```

```

17     """
18     # 使用注意力机制计算背景向量
19     c = attention_forward(self.attention, enc_states, state[-1])
20     # 将嵌入后的输入和背景向量在特征维连结
21     # (批量大小, num_hiddens+embed_size)
22     input_and_c = torch.cat((self.embedding(cur_input), c), dim=1)
23     # 为输入和背景向量的连结增加时间步维, 时间步个数为1
24     output, state = self.rnn(input_and_c.unsqueeze(0), state)
25     # 移除时间步维, 输出形状为(批量大小, 输出词典大小)
26     output = self.out(output).squeeze(dim=0)
27     return output, state
28 def begin_state(self, enc_state):
29     # 直接将编码器最终时间步的隐藏状态作为解码器的初始隐藏状态
30     return enc_state

```

### 9.12.3 训练模型

我们先实现 batch\_loss 函数计算一个小批量的损失。解码器在最初时间步的输入是特殊字符 `BOS`。之后，解码器在某时间步的输入为样本输出序列在上一时间步的词，即强制教学。此外，同9.3节（word2vec的实现）中的实现一样，我们在这里也使用掩码变量避免填充项对损失函数计算的影响。

```

1 In [11]:
2     def batch_loss(encoder, decoder, X, Y, loss):
3         batch_size = X.shape[0]
4         enc_state = encoder.begin_state()
5         enc_outputs, enc_state = encoder(X, enc_state)
6         # 初始化解码器的隐藏状态
7         dec_state = decoder.begin_state(enc_state)
8         # 解码器在最初时间步的输入是BOS
9         dec_input = torch.tensor([out_vocab.stoi[BOS]]*batch_size)
10        # 我们将使用掩码变量mask来忽略掉标签为填充项PAD的损失
11        mask, num_not_pad_tokens = torch.ones(batch_size,), 0
12        l = torch.tensor([0.0])
13        # Y shape: (batch, seq_len)
14        for y in Y.permute(1, 0):
15            dec_output, dec_state = decoder(
16                dec_input, dec_state, enc_outputs)
17            l = l + (mask*loss(dec_output, y)).sum()
18            # 使用强制教学
19            dec_input = y
20            num_not_pad_tokens += mask.sum().item()
21            # EOS后面全是PAD
22            # 下面一行保证一旦遇到EOS接下来的循环中mask就一直是0
23            mask = mask * (y != out_vocab.stoi[EOS]).float()
24        return l / num_not_pad_tokens

```

在训练函数中，我们需要同时迭代编码器和解码器的模型参数。

```

1 In [12]:
2     def train(encoder, decoder, dataset, lr, batch_size, num_epochs):
3         enc_optimizer = torch.optim.Adam(encoder.parameters(), lr=lr)
4         dec_optimizer = torch.optim.Adam(decoder.parameters(), lr=lr)
5         loss = nn.CrossEntropyLoss(reduction='none')
6         data_iter = Data.DataLoader(dataset, batch_size, shuffle=True)
7         for epoch in range(num_epochs):
8             l_sum = 0.0

```

```

9         for x, y in data_iter:
10            enc_optimizer.zero_grad()
11            dec_optimizer.zero_grad()
12            l = batch_loss(encoder, decoder, x, y, loss)
13            l.backward()
14            enc_optimizer.step()
15            dec_optimizer.step()
16            l_sum += l.item()
17        if (epoch+1) % 10 == 0:
18            print('epoch %d, loss %.3f' % (epoch+1,
19                                         l_sum/len(data_iter)))

```

接下来，创建模型实例并设置超参数。然后，我们就可以训练模型了。

```

1 In [13]:
2     embed_size, num_hiddens, num_layers = 64, 64, 2
3     attention_size = 10
4     drop_prob = 0.5
5     lr = 0.01
6     batch_size = 2
7     num_epochs = 50
8     encoder = Encoder(len(in_vocab), embed_size,
9                         num_hiddens, num_layers, drop_prob)
10    decoder = Decoder(len(out_vocab), embed_size,
11                        num_hiddens, num_layers, attention_size,
12                        drop_prob)
13    train(encoder, decoder, dataset, lr, batch_size, num_epochs)
14 Out [13]:
15    epoch 10, loss 0.462
16    epoch 20, loss 0.215
17    epoch 30, loss 0.114
18    epoch 40, loss 0.105
19    epoch 50, loss 0.039

```

## 9.12.4 预测不定长的序列

在9.10节（束搜索）中我们介绍了3种方法来生成解码器在每个时间步的输出。这里我们实现最简单的贪婪搜索。

```

1 In [14]:
2     def translate(encoder, decoder, input_seq, max_seq_len):
3         in_tokens = input_seq.split(' ')
4         in_tokens += [EOS]+[PAD]*(max_seq_len-len(in_tokens)-1)
5         # batch=1
6         enc_input = torch.tensor(
7             [[in_vocab.stoi[tk] for tk in in_tokens]]))
8         enc_state = encoder.begin_state()
9         enc_output, enc_state = encoder(enc_input, enc_state)
10        dec_input = torch.tensor([out_vocab.stoi[BOS]])
11        dec_state = decoder.begin_state(enc_state)
12        output_tokens = []
13        for _ in range(max_seq_len):
14            dec_output, dec_state = decoder(
15                dec_input, dec_state, enc_output)
16            pred = dec_output.argmax(dim=1)
17            pred_token = out_vocab.itos[int(pred.item())]

```

```
18     # 当任一时间步搜索出现EOS时，输出序列即完成
19     if pred_token == EOS:
20         break
21     else:
22         output_tokens.append(pred_token)
23         dec_input = pred
24
return output_tokens
```

简单测试一下模型。输入法语句子“ils regardent.”，翻译后的英语句子应该是“they are watching.”。

```
1 In [15]:  
2     input_seq = 'ils regardent .'  
3     translate(encoder, decoder, input_seq, max_seq_len)  
4 Out [15]:  
5     ['they', 'are', 'watching', '.']
```

### 9.12.5 评价翻译结果

评价机器翻译结果通常使用BLEU（Bilingual Evaluation Understudy）。对于模型预测序列中任意的子序列，BLEU考察这个子序列是否出现在标签序列中。具体计算过程详见本章附录。

具体来说，设词数为 $n$ 的子序列的精度为 $p_n$ 。它是预测序列与标签序列匹配词数为 $n$ 的子序列的数量与预测序列中词数为 $n$ 的子序列的数量之比。举个例子，假设标签序列为 $A, B, C, D, E, F$ ，预测序列为 $A, B, B, C, D$ ，那么 $p_1 = 4/5, p_2 = 3/4, p_3 = 1/3, p_4 = 0$ 。设 $len_{label}$ 和 $len_{pred}$ 分别为标签序列和预测序列的词数，那么，BLEU的定义为

$$\exp\left(\min\left(0, 1 - \frac{\text{len}_{\text{label}}}{\text{len}_{\text{pred}}}\right)\right) \prod_{n=1}^k p_n^{1/2^n}$$

其中 $k$ 是我们希望匹配的子序列的最大词数。可以看到当预测序列和标签序列完全一致时，BLEU为1。

因为匹配较长子序列比匹配较短子序列更难，BLEU对匹配较长子序列的精度赋予了更大权重。例如，当 $p_n$ 固定在0.5时，随着 $n$ 的增大， $0.5^{1/2} \approx 0.7, 0.5^{1/4} \approx 0.84, 0.5^{1/8} \approx 0.92, 0.5^{1/16} \approx 0.96$ 。另外，模型预测较短序列往往得到较高 $p_n$ 值。因此，**上式中连乘项前面的系数是为了惩罚较短的输出而设的。**举个例子，当 $k=2$ 时，假设标签序列为 $A, B, C, D, E, F$ ，而预测序列为 $A, B$ 。虽然 $p_1 = p_2 = 1$ ，但惩罚系数 $\exp(1 - 6/2) \approx 0.14$ ，因此BLEU也接近0.14。

下面来实现BLEU的计算。

```
1 In [16]:  
2     def bleu(pred_tokens, label_tokens, k):  
3         len_pred, len_label = len(pred_tokens), len(label_tokens)  
4         score = math.exp(min(0, 1 - len_label / len_pred))  
5         for n in range(1, k+1):  
6             num_matches, label_subs = 0, collections.defaultdict(int)  
7             for i in range(len_label-n+1):  
8                 label_subs[''.join(label_tokens[i:i+n])] += 1  
9             for i in range(len_pred-n+1):  
10                if label_subs[''.join(pred_tokens[i:i+n])] > 0:  
11                    num_matches += 1  
12                    label_subs[''.join(pred_tokens[i:i+n])] -= 1  
13                score *= math.pow(num_matches/(len_pred-n+1),  
14                                  math.pow(0.5, n))  
15        return score
```

接下来，定义一个辅助打印函数。

```
1 In [17]:  
2     def score(input_seq, label_seq, k):  
3         pred_tokens = translate(encoder, decoder,  
4                               input_seq, max_seq_len)  
5         label_tokens = label_seq.split(' ')  
6         print('bleu %.3f, predict: %s' % (  
7             bleu(pred_tokens, label_tokens, k),  
8             ' '.join(pred_tokens)))
```

预测正确则分数为1。

```
1 In [18]:  
2     score('ils regardent .', 'they are watching .', k=2)  
3 Out [18]:  
4     bleu 1.000, predict: they are watching .
```

测试一个不在训练集中的样本。

```
1 In [19]:  
2     score('ils sont canadienne .', 'they are canadian .',  
3             k=2)  
4 Out [19]:  
5     bleu 0.658, predict: they are arguing .
```

## 小结：

- 可以将编码器—解码器和注意力机制应用于机器翻译中。
- BLEU可以用来评价翻译结果。

## 9.13 本章附录

### ☆ 似然函数

#### 1. 统计推断的两大学派

在统计领域，有两种对立的思想学派：贝叶斯学派和经典学派（也称频率学派），他们之间最重要的区别就是如何看待被估计的未知参数。贝叶斯学派的观点是将其看成是已知分布的随机变量，而经典学派的观点是将其看成未知的待估计的常量。

##### 1.1 贝叶斯统计推断

具体来说，贝叶斯推断方法是将未知参数看做是一个随机变量，他具备某种先验分布。在已知观测数据  $x$  的基础上，可以利用贝叶斯公式来推导后验概率分布  $p_{\Theta|X}(\theta | x)$ ，这样就同时包含人的先验知识以及观测值  $x$  所能提供的关于  $\theta$  的新信息。贝叶斯统计推断的内容，我们这里不展开。

##### 1.2 经典统计推断

而经典统计方法是将未知参数  $\theta$  看作是一个常数，但是他是未知的，那么，这就需要去估计他了。经典统计的目标就是提出参数  $\theta$  的估计方法，并且保证其具有一定的性质。

##### 1.3 一个例子

我们举个简单的例子，比如我们要通过一个物理试验来测量某个粒子的质量，从经典学派的观点来看，虽然粒子的质量未知，但他本质上是一个确定的常数，不能将其看成是一个随机变量。而贝叶斯学派则截然不同，会将待估计的粒子质量看做是一个随机变量，并利用人们对该粒子的已有的认知给他一个先验分布，按照分布的概率模型，使其集中在某个指定的范围内。

这一讲，我们重点介绍经典统计推断当中的极大似然估计法。为了给大家一个直观的感觉，这里我先来两个例子。

## 2. 极大似然估计法的引例

第一个例子还是盒子摸球的例子。有两个盒子，一号盒子里面有100个球，其中99个是白球，1个是黑球；二号盒子里面也有100个球，其中99个是黑球，1个是白球。现在我告诉你，我从其中一个盒子中随机摸出来一个球，这个球是白球，那么你说，我更有可能是从哪个盒子里摸出的这个球？

显然，你会说是一号盒子。道理很简单，因为一号盒子当中，摸出白球的概率是0.99，而二号盒子摸出白球的概率是0.01。显然更有可能是一号盒子了。

第二个例子也是大家熟悉的丢硬币的例子。我有三个不均匀的硬币，其中第一个硬币抛出正面的概率是 $\frac{2}{5}$ ，第二个硬币抛出正面的概率是 $\frac{1}{2}$ ，第三个硬币抛出正面的概率是 $\frac{3}{5}$ ，这时我取其中一个硬币，抛了20次，其中正面向上的次数是13次，请问我最有可能是拿的哪一个硬币？

思考的过程也很简单，三枚硬币，抛掷20次，13次正面向上的概率分别是：

$$\begin{aligned} \text{第一枚: } & C_{20}^{13} \left(\frac{2}{5}\right)^{13} \left(1 - \frac{2}{5}\right)^{20-13} = 0.014563052125736147 \\ \text{第二枚: } & C_{20}^{13} \left(\frac{1}{2}\right)^{13} \left(1 - \frac{1}{2}\right)^{20-13} = 0.0739288330078125 \\ \text{第三枚: } & C_{20}^{13} \left(\frac{3}{5}\right)^{13} \left(1 - \frac{3}{5}\right)^{20-13} = 0.1658822656197132 \end{aligned}$$

```
1 from scipy.special import comb
2 import math
3 def get_possibility(n, head, p_head):
4     return comb(n, head)*math.pow(p_head, head)*math.pow(
5         (1-p_head), (n-head))
6 print(get_possibility(20, 13, 2/5))
7 >>> 0.014563052125736147
8 print(get_possibility(20, 13, 1/2))
9 >>> 0.0739288330078125
10 print(get_possibility(20, 13, 3/5))
11 >>> 0.1658822656197132
```

第三枚硬币抛掷出这种结果的概率最大，我更有可能拿的第三枚硬币？这种直观的认识是正确的，这种思维方式的背后正是我们要介绍的极大似然估计法，他就是这么的简单粗暴而有效。

## 3. 似然函数的由来

有了这个例子，下面我们开始介绍极大似然估计方法。我们重点要理解的是似然这个词，这个词听起来比较陌生。

我们首先看离散型的情形，随机变量  $X$  的概率分布已知，但是这个分布的参数是未知的，需要我们去估计，我们把他记作是  $\theta$ ，好比上面抛掷硬币的试验中，硬币正面朝上的概率是未知的，需要我们去估计，那么此时  $\theta$  就代表了这个待估计的正面向上的概率值。

随机变量  $X$  的取值  $x_i$  表示抛掷  $k$  次硬币，正面向上的次数，那么这个概率就表示为：

$P(\{X = x_i\}) = C_k^{x_i} \theta^{x_i} (1 - \theta)^{k-x_i}$ 。这里需要注意的是， $k$  和  $x_i$  都是指定的、已知的，而参数  $\theta$  是一个未知的参数。因此在这个大的背景下，抛掷  $k$  次，其中有  $x_i$  次向上的概率是一个关于未知参数  $\theta$  的函数，我们把他写作是  $P(\{X = x_i\}) = p(x_i; \theta)$ 。概括地说：概率质量函数PMF是一个关于待估参数  $\theta$  的函数。

那么此时，我们做  $n$  次这种实验，每次实验中，都是连续抛掷  $k$  次硬币，统计正面出现的次数，这样就能取得一系列的样本： $x_1, x_2, x_3, \dots, x_n$ ，这些样本的取值之间满足**相互独立**，那么这一串样本取得上述取值  $\{X_1 = x_1, X_2 = x_2, X_3 = x_3, \dots, X_n = x_n\}$  的联合概率为： $p(x_1; \theta) \cdot p(x_2; \theta) \cdot p(x_3; \theta) \cdots p(x_n; \theta)$ ，用连乘符号写起来就是  $\prod_{i=1}^n p(x_i; \theta)$ 。这是一个通用的表达式，实际上，你别看他表达式是长长的一串，实际上他的未知数就是一个  $\theta$ ，而其他的  $x_i$  都是已知的样本值，因此我们说  $\theta$  的取值，完全决定了这一连串样本取值的联合概率。由此，我们更换一个更加有针对性的写法：

$$L(\theta) = L(x_1, x_2, x_3, \dots, x_n; \theta) = \prod_{i=1}^n p(x_i; \theta)$$

那么， $L(\theta) = L(x_1, x_2, x_3, \dots, x_n; \theta)$  就是这一串已知样本值  $x_1, x_2, x_3, \dots, x_n$  的似然函数，他描述了取得这一串指定样本值的概率值，而这个概率值完全由未知参数  $\theta$  决定。这就是似然函数的由来。当然如果  $X$  是一个连续型的随机变量，我们只要相应的把离散型的概率质量函数替换成连续型的概率密度函数即可：

$$L(\theta) = L(x_1, x_2, x_3, \dots, x_n; \theta) = \prod_{i=1}^n f(x_i; \theta)$$

#### 4. 极大似然估计的思想

显然，似然函数  $L(\theta) = L(x_1, x_2, x_3, \dots, x_n; \theta)$  指的就是随机变量  $X$  取到指定的这一组样本值： $x_1, x_2, x_3, \dots, x_n$  时的概率的大小。当未知的待估计的参数  $\theta$  取不同的值时，计算出来的概率的值会发生变化。

例如，当  $\theta = \theta_0$  时，似然函数  $L(x_1, x_2, x_3, \dots, x_n; \theta)$  的取值为0或趋近于0，那么意味着，当  $\theta = \theta_0$  时，随机变量  $X$  取得这一组样本  $x_1, x_2, x_3, \dots, x_n$  的概率为0，即压根儿不可能得到这一组样本值，或可能性非常非常小，那么你肯定不会觉得参数  $\theta$  应该能够取  $\theta_0$  这个数。

那么当  $\theta$  取  $\theta_1$  和  $\theta_2$  两种不同的值时，似然函数的值

$L(x_1, x_2, x_3, \dots, x_n; \theta_1) > L(x_1, x_2, x_3, \dots, x_n; \theta_2)$ ，意味着，当  $\theta = \theta_1$  时，随机变量  $X$  取得这一组指定样本的概率要更大一些，换句话说， $\theta$  取  $\theta_1$  比取  $\theta_2$  有更大的可能获得这一组样本值  $x_1, x_2, x_3, \dots, x_n$ ，那么当你面对这一组已经获得的采样值，在  $\theta_1$  和  $\theta_2$  中二选一作为估计值的时候，倾向于选择使似然函数取值更大的估计值，就是再自然不过的了。这里就是盒子摸球试验中，我们选择一号盒子，丢硬币试验中，我们选择第三枚硬币的原因。

那么更进一步，跳出前面几个引导例子的限制，当我们的未知参数选择的余地更大时，比如我们的未知参数  $\theta$  是对一个概率值的估计，那么他的取值范围就是一个在  $[0, 1]$  之间取值的连续变量，如果是估计总体的方差，那么他的范围就是非负数，如果估计的是总体的均值，那么他的范围就是全体实数了。

此时我们要做的就是在未知参数  $\theta$  的取值范围  $\Theta$  中选取使得似然函数  $L(x_1, x_2, x_3, \dots, x_n; \theta)$  能够取得最大值的  $\hat{\theta}$ ，作为未知参数的估计值，由于  $\hat{\theta}$  使得似然函数取值达到最大，因此  $\hat{\theta}$  就是未知参数  $\theta$  的极大似然估计。换句话说，未知参数  $\theta$  取估计值  $\hat{\theta}$  时获取到这组已知样本  $x_1, x_2, x_3, \dots, x_n$  的可能性比取其他任何值时都要大，在这种思维框架下，我们有什么理由不用他呢？

#### 5. 极大似然估计值的计算

那么接下来，问题就到了如何求解这个极大似然估计值了。问题转换为一个求最值的问题：即：在给定概率模型和一组相互独立的观测样本  $x_1, x_2, x_3, \dots, x_n$  的基础上，求解使得似然函数  $L(\theta) = L(x_1, x_2, x_3, \dots, x_n; \theta) = \prod_{i=1}^n p(x_i; \theta)$  取得最大值的未知参数  $\theta$  的取值。当然，如果是连续型随机变量，则把似然函数替换成  $L(\theta) = L(x_1, x_2, x_3, \dots, x_n; \theta) = \prod_{i=1}^n f(x_i; \theta)$  即可。

那么下面问题就变得很直接了，对似然函数求导，使得导数为0的  $\theta$  的取值，就是我们要找的极大似然估计值  $\hat{\theta}$ 。

这个连乘的函数求导数比较复杂，由于函数  $g(x)$  和  $\ln(g(x))$  的单调性是保持一致的。因此我们可以选择把似然函数  $L(x)$  转化为  $\ln(L(x))$ ，这样连乘就变成了连加：

$$\ln(L(\theta)) = \ln\left(\prod_{i=1}^n p(x_i; \theta)\right) = \ln(p(x_1; \theta)) + \ln(p(x_2; \theta)) + \dots + \ln(p(x_n; \theta)) = \sum_{i=1}^n \ln(p(x_i; \theta))$$

此时再对他进行求导就变得容易了，如果方程有唯一解，且是极大值点，那么我们就求得了极大似然估计值。

如果有多个未知参数需要我们去估计呢？那也好办，用上偏导数就可以了：

$$\ln(L(\theta_1, \theta_2, \dots, \theta_k)) = \ln\left(\prod_{i=1}^n p(x_i; \theta_1, \theta_2, \dots, \theta_k)\right) = \sum_{i=1}^n \ln(p(x_i; \theta_1, \theta_2, \dots, \theta_k))$$

为了使得  $\ln(L)$  达到最大，我们对每一个待估计的未知参数  $\theta_i$ ，都去求偏导数，并建立方程组：

$$\begin{cases} \frac{\partial \ln L}{\partial \theta_1} = 0 \\ \frac{\partial \ln L}{\partial \theta_2} = 0 \\ \dots \\ \frac{\partial \ln L}{\partial \theta_k} = 0 \end{cases}$$

解得这个方程组就可以了。

## 6. 极大似然估计的例子

说了这么多的理论方法，最后我们还是来看看实际例子中的估计方法：

### 6.1 简单案例热身

第一个例子还是抛硬币的例子，我们的硬币正反面不规则，我们想要估计他正面向上的概率  $\theta$ ，我们连续抛掷10次，每一次抛掷的结果形成的样本序列如下：正，正，正，反，反，正，反，正，正，反。很显然，每次抛掷的过程是都是彼此独立的，并且  $X$  是一个伯努利随机变量。我们知道：

$p(\{x_i = \text{正}\}) = \theta$ ,  $p(\{x_i = \text{反}\}) = 1 - \theta$ 。那么这组观测数据的似然函数为：

$L(x_1, x_2, \dots, x_{10}; \theta) = \theta^3(1 - \theta)^2\theta(1 - \theta)\theta^2(1 - \theta) = \theta^6(1 - \theta)^4$ 。将其转换为对数似然函数：

$$\ln(L(x_1, x_2, \dots, x_{10}; \theta)) = \ln(\theta^6(1 - \theta)^4) = 6 \ln \theta + 4 \ln(1 - \theta)$$

然后对对数似然函数求导：

$$\ln'(L(x_1, x_2, \dots, x_{10}; \theta)) = (6 \ln \theta + 4 \ln(1 - \theta))' = \frac{6}{\theta} + \frac{4}{\theta - 1} = \frac{10\theta - 6}{\theta(\theta - 1)}$$

让对数似然函数的导数为0：

$$\ln'(L(x_1, x_2, \dots, x_{10}; \theta)) = \frac{10\theta - 6}{\theta(\theta - 1)} = 0$$

得到极大似然估计值  $\hat{\theta} = \frac{6}{10}$ 。

### 6.2 单参数极大似然估计

再看一个指数分布的例子：我们在前面学习过，在一个柜台前，相邻顾客的到达时间的时间间隔服从参数为  $\theta$  的指数分布，我们获取了一组上述时间间隔的样本  $x_1, x_2, x_3, \dots, x_n$ ，下面来运用极大似然估计的方法来估计未知参数  $\theta$ 。首先依据指数分布，得到： $f(x_i; \theta) = \theta e^{-\theta x_i}$ 。紧接着，得到对数似然函数：

$$\begin{aligned}\ln(L(x_1, x_2, \dots, x_n; \theta)) &= \ln(\prod_i^n \theta e^{-\theta x_i}) = \sum_{i=1}^n \ln \theta e^{-\theta x_i} = \sum_{i=1}^n \ln \theta + \sum_{i=1}^n \ln e^{-\theta x_i} \\ &= \sum_{i=1}^n \ln \theta - \sum_{i=1}^n \theta x_i = \sum_{i=1}^n \ln \theta - \theta \sum_{i=1}^n x_i = n \ln \theta - \theta \sum_{i=1}^n x_i\end{aligned}$$

此时，我们对这个对数似然函数求导，来获取未知参数的极大似然估计值  $\hat{\theta}$ ，也很简单：

$$\ln'(L(x_1, x_2, \dots, x_n; \theta)) = \left( n \ln \theta - \theta \sum_{i=1}^n x_i \right)' = \frac{n}{\theta} - \sum_{i=1}^n x_i = 0$$

则有： $\frac{n}{\theta} = \sum_{i=1}^n x_i$ ，那么未知参数  $\theta$  的极大似然估计值  $\hat{\theta} = \frac{n}{\sum_{i=1}^n x_i}$ 。

### 6.3 多参数极大似然估计

上面是单参数的极大似然估计的例子，那么下面我们再来看看多参数的例子，这里我们从一个服从参数为  $(\mu, \sigma^2)$  的正态分布当中获取一组采样值： $x_1, x_2, x_3, \dots, x_n$ ，通过这组采样值，我们来求得两个参数的极大似然估计值，首先写出似然函数：

$$L(x_1, x_2, x_3, \dots, x_n; \mu, \sigma^2) = \prod_{i=1}^n \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x_i-\mu)^2}{2\sigma^2}} = (2\pi\sigma^2)^{-\frac{n}{2}} e^{-\frac{1}{2\sigma^2} \sum_{i=1}^n (x_i-\mu)^2}$$

写成对数似然函数的形式：

$$\begin{aligned}\ln(L(x_1, x_2, x_3, \dots, x_n; \mu, \sigma^2)) &= \ln((2\pi\sigma^2)^{-\frac{n}{2}} e^{-\frac{1}{2\sigma^2} \sum_{i=1}^n (x_i-\mu)^2}) \\ &= \ln(2\pi\sigma^2)^{-\frac{n}{2}} + \ln(e^{-\frac{1}{2\sigma^2} \sum_{i=1}^n (x_i-\mu)^2}) = -\frac{n}{2} \ln 2\pi - \frac{n}{2} \ln(\sigma^2) - \frac{1}{2\sigma^2} \sum_{i=1}^n (x_i - \mu)^2\end{aligned}$$

由于这里有两个待估计的参数，我们分别对其进行求偏导， $\sigma^2$  我们把他看作是一个整体即可：

$$\begin{cases} \frac{\partial(-\frac{n}{2} \ln 2\pi - \frac{n}{2} \ln(\sigma^2) - \frac{1}{2\sigma^2} \sum_{i=1}^n (x_i - \mu)^2)}{\partial \mu} = 0 \\ \frac{\partial(-\frac{n}{2} \ln 2\pi - \frac{n}{2} \ln(\sigma^2) - \frac{1}{2\sigma^2} \sum_{i=1}^n (x_i - \mu)^2)}{\partial \sigma^2} = 0 \end{cases}$$

第一个求偏导数的式子中，我们可以得出： $2 \sum_{i=1}^n (x_i - \mu) = 0$  则有： $\sum_{i=1}^n \mu = \sum_{i=1}^n x_i$  即  $n\mu = \sum_{i=1}^n x_i \Rightarrow \hat{\mu} = \frac{1}{n} \sum_{i=1}^n x_i$ ，即得到了均值的极大似然估计值。

第二个求偏导的式子中，注意我们是把  $\sigma^2$  看作是一个整体，因此可以得出：

$-\frac{n}{2\sigma^2} + \frac{1}{2\sigma^4} \sum_{i=1}^n (x_i - \mu)^2 = 0$  得到  $\hat{\sigma}^2 = \frac{1}{n} \sum_{i=1}^n (x_i - \mu)^2$ ，而这里我们要带入  $\mu$  的极大似然估计值  $\hat{\mu}$ ，最终得到了方差的极大似然估计值： $\hat{\sigma}^2 = \frac{1}{n} \sum_{i=1}^n (x_i - \hat{\mu})^2$ 。

## ☆深入理解GloVe模型

### 1. 概述

- 模型目标：进行词的向量化表示，使得词向量尽可能多地蕴含语义和语法的信息。
- 输入：语料库。
- 输出：词向量。
- 方法概述：首先基于语料库构建词的共现矩阵，然后基于共现矩阵和GloVe模型学习词向量。

### 2. 统计共现矩阵

设共现矩阵为  $X$ ，其元素为  $X_{ij}$ 。 $X_{ij}$  的意义为：在整个语料库中，单词  $i$  和单词  $j$  共同出现在一个窗口中的次数。举例如下，设有语料库：i love you but you love him i am sad。这个小小的语料库只有1个句子，涉及到7个单词：i、love、you、but、him、am、sad。如果我们采用一个窗口宽度为5（左右长度都为2）的统计窗口，那么就有以下窗口内容：

窗口标号	中心词	窗口内容
0	i	i love you
1	love	i love you but
2	you	i love you but you
3	but	love you but you love
4	you	you but you love him
5	love	but you love him i
6	him	you love him i am
7	i	love him i am sad
8	am	him i am sad
9	sad	i am sad

窗口0、1长度小于5是因为中心词左侧内容少于2个，同理窗口8、9长度也小于5。以窗口5为例说明如何构造共现矩阵，中心词为love，语境词为but、you、him、i；则执行：

$$X_{\text{love}, \text{but}} = 1, X_{\text{love}, \text{you}} = 1, X_{\text{love}, \text{him}} = 1, X_{\text{love}, \text{i}} = 1$$

使用窗口将整个语料库遍历一遍，即可得到共现矩阵  $X$ 。

### 3. 使用GloVe模型训练词向量

#### 3.1 模型公式

先看模型，损失函数长这个样子：

$$J = \sum_{i,j}^N f(X_{i,j}) (v_i^T v_j + b_i + c_j - \log(X_{i,j}))^2$$

$v_i, v_j$  是单词  $i$  和单词  $j$  的词向量， $b_i, c_j$  是两个标量（作者定义的偏差项）， $f$  是权重函数（具体函数公式及功能在后文介绍）， $N$  是词汇表的大小（共现矩阵维度为  $N \times N$ ）。可以看到，GloVe模型没有使用神经网络的方法。

#### 3.2 模型怎么来的

那么作者为什么这么构造模型呢？首先定义几个符号：

$$X_i = \sum_{j=1}^N X_{i,j}$$

其实就是矩阵单词  $i$  那一行的和；

$$P_{i,k} = \frac{X_{i,k}}{X_i}$$

条件概率，表示单词  $k$  出现在单词  $i$  语境中的概率；

$$\text{ratio}_{i,j,k} = \frac{P_{i,k}}{P_{j,k}}$$

两个条件概率的比率。

作者的灵感是这样的，作者发现  $\text{ratio}_{i,j,k}$  这个指标是有规律的，规律统计在下表：

$\text{ratio}_{i,j,k}$ 的值	单词 $j, k$ 相关	单词 $j, k$ 不相关
单词 $i, k$ 相关	趋近1	很大
单词 $i, k$ 不相关	很小	趋近1

很简单的规律，但是有用。思想：假设我们已经得到了词向量，如果我们用词向量  $v_i, v_j, v_k$  通过某种函数计算  $\text{ratio}_{i,j,k}$ ，能够同样得到这样的规律的话，就意味着我们词向量与共现矩阵具有很好的一致性，也就说明我们的词向量中蕴含了共现矩阵中所蕴含的信息。

设用词向量  $v_i, v_j, v_k$  计算  $\text{ratio}_{i,j,k}$  的函数为  $g(v_i, v_j, v_k)$ （我们先不去管具体的函数形式），那么应该有：

$$\frac{P_{i,k}}{P_{j,k}} = \text{ratio}_{i,j,k} = g(v_i, v_j, v_k)$$

即：

$$\frac{P_{i,k}}{P_{j,k}} = g(v_i, v_j, v_k)$$

即二者应该尽可能地接近，很容易想到用二者的方差来作为损失函数：

$$J = \sum_{i,j,k}^N \left( \frac{P_{i,k}}{P_{j,k}} - g(v_i, v_j, v_k) \right)^2$$

但是仔细一看，模型中包含3个单词，这就意味着要在  $N \times N \times N$  的复杂度上进行计算，太复杂了，最好能再简单点。现在我们来仔细思考  $g(v_i, v_j, v_k)$ ，或许它能帮上忙。作者的想法是这样的：

1. 考虑单词  $i$  和单词  $j$  之间的关系，那  $g(v_i, v_j, v_k)$  中大概要有这么一项吧： $v_i - v_j$ 。嗯，合理，在线性空间中考察两个向量的相似性，不失线性地考察，那么  $v_i - v_j$  大概是个合理的选择；
2.  $\text{ratio}_{i,j,k}$  是个标量，那么  $g(v_i, v_j, v_k)$  最后应该是个标量啊，虽然其输入都是向量，那内积应该是合理的选择，于是应该有这么一项吧： $(v_i - v_j)^T v_k$ 。
3. 然后作者又往  $(v_i - v_j)^T v_k$  的外面套了一层指数运算  $\exp()$ ，得到最终的  $g(v_i, v_j, v_k) = \exp((v_i - v_j)^T v_k)$ 。

最关键的第3步，为什么套了一层  $\exp()$ ？

套上之后，我们的目标是让以下公式尽可能地成立：

$$\frac{P_{i,k}}{P_{j,k}} = g(v_i, v_j, v_k)$$

即：

$$\frac{P_{i,k}}{P_{j,k}} = \exp((v_i - v_j)^T v_k)$$

即：

$$\frac{P_{i,k}}{P_{j,k}} = \exp(v_i^T v_k - v_j^T v_k)$$

即：

$$\frac{P_{i,k}}{P_{j,k}} = \frac{\exp(v_i^T v_k)}{\exp(v_j^T v_k)}$$

然后就发现找到简化方法了：只需要让上式分子对应相等，分母对应相等，即： $P_{i,k} = \exp(v_i^T v_k)$  并且  $P_{j,k} = \exp(v_j^T v_k)$ 。分子分母形式相同，就可以把两者统一考虑了，即：

$$P_{i,j} = \exp(v_i^T v_j)$$

本来我们追求：

$$\frac{P_{i,k}}{P_{j,k}} = g(v_i, v_j, v_k)$$

现在只需要追求：

$$P_{i,j} = \exp(v_i^T v_j)$$

两边取个对数：

$$\log(P_{i,j}) = v_i^T v_j$$

那么损失函数就可以简化为：

$$J = \sum_{i,j}^N (\log(P_{i,j}) - v_i^T v_j)^2$$

现在只需要在  $N \times N$  的复杂度上进行计算，而不是  $N \times N \times N$ ，现在关于为什么第3步中，外面套一层  $\exp()$  就清楚了，**正是因为套了一层  $\exp()$ ，才使得差形式变成商形式，进而等式两边分子分母对应相等，进而简化模型。**

然而，出了点问题。仔细看这两个式子： $\log(P_{i,j}) = v_i^T v_j$  和  $\log(P_{j,i}) = v_j^T v_i$ 。 $\log(P_{i,j})$  不等于  $\log(P_{j,i})$  但是  $v_i^T v_j$  等于  $v_j^T v_i$ 。即等式左侧不具有对称性，但是右侧具有对称性。数学上出了问题，补救一下好了。

现将损失函数中的条件概率展开：

$$\log(P_{i,j}) = v_i^T v_j$$

即为：

$$\log(X_{i,j}) - \log(X_i) = v_i^T v_j$$

将其变为：

$$\log(X_{i,j}) = v_i^T v_j + b_i + c_j$$

即添了一个偏差项  $c_j$ ，并将  $\log(X_i)$  吸收到偏差项  $b_i$  中。

于是损失函数就变成了：

$$J = \sum_{i,j}^N (v_i^T v_j + b_i + c_j - \log(X_{i,j}))^2$$

然后基于出现频率越高的词对权重应该越大的原则，在损失函数中添加权重项，于是损失函数进一步完善：

$$J = \sum_{i,j}^N f(X_{i,j}) (v_i^T v_j + b_i + c_j - \log(X_{i,j}))^2$$

具体权重函数应该是怎么样的呢？首先应该是非减的，其次当词频过高时，权重不应过分增大，作者通过实验确定权重函数为：

$$f(x) = \begin{cases} (x/xmax)^{0.75}, & \text{if } x < xmax \\ 1, & \text{if } x \geq xmax \end{cases}$$

到此，整个模型就介绍完了。

### ☆ BLEU指标

**核心思想：**The primary programming task for a BLEU implementor is to compare n-grams of the candidate with the n-grams of the reference translation and count the number of matches. These matches are position independent. The more the matches, the better the candidate translation is.

**计算方式：** $c_i$  表示模型生成的译文， $s_i = s_{i1}, \dots, s_{im}$  表示  $m$  个参考译文， $h_k(c_i)$  表示元素  $w_k$  在译文  $c_i$  中出现的次数， $h_k(s_{ij})$  表示元素  $w_k$  在译文  $s_{ij}$  中出现的次数， $w_k$  表示句子中的第  $k$  个n-gram词组， $\max_{j \in m} h_k(s_{ij})$  表示元素  $w_k$  在各条参考译文中的最大出现次数。基于上述定义，我们给出各阶n-gram的精度计算公式：

$$P_n = \frac{\sum_i \sum_k \min(h_k(c_i), \max_{j \in m} h_k(s_{ij}))}{\sum_i \sum_k h_k(c_i)}$$

不难发现上述指标可能随着模型生成翻译句子长度变短，评价指标得到的评分越高，所以我们需要引入长度惩罚因子(Brevity Penalty, BP)，其中  $l_c$  表示模型生成译文的长度， $l_s$  表示标准译文的长度，当存在多个标准译文时，我们选用和  $l_c$  取值最接近的  $l_s$ ：

$$BP = \begin{cases} 1 & \text{if } l_c > l_s \\ e^{1 - \frac{l_s}{l_c}} & \text{if } l_c \leq l_s \end{cases}$$

我们将上述两部分进行组合，并对各阶n-gram进行综合度量，得到如下计算公式，其中  $w_n$  可以采用几何平均进行计算，这里的对数是以  $e$  为底的：

$$BLEU = BP \times e^{\sum_{n=1}^N w_n \log p_n}$$

### 算例分析：

模型生成译文：The cat the cat on the mat.

标准译文1：The cat is on the mat.

标准译文2：There is a cat on the mat.

1-gram：

生成：['The', 'cat', 'the', 'cat', 'on', 'the', 'mat']

标准1：['The', 'cat', 'is', 'on', 'the', 'mat']

标准2：['There', 'is', 'a', 'cat', 'on', 'the', 'mat']

单词分析：the: 2; cat: 1; on: 1; mat: 1

计算1-gram精度：上一行求和/生成译文中取一个单词的取法=5/7

2-gram：

生成：['The cat', 'cat the', 'cat on', 'on the', 'the mat']

标准1：['The cat', 'cat is', 'is on', 'on the', 'the mat']

标准2：['There is', 'is a', 'a cat', 'cat on', 'on the', 'the mat']

词组分析: 'The cat': 1; 'cat the': 0; 'cat on': 1; 'on the': 1; 'the mat': 1

计算2-gram精度: 上一行求和/生成译文中连续取两个单词的取法=4/6

3-gram:

生成: ['The cat the', 'cat the cat', 'the cat on', 'cat on the', 'on the mat']

标准1: ['The cat is', 'cat is on', 'is on the', 'on the mat']

标准2: ['There is a', 'is a cat', 'a cat on', 'cat on the', 'on the mat']

词组分析: 'The cat the': 0; 'cat the cat': 0; 'the cat on': 0; 'cat on the': 1; 'on the mat': 1

计算3-gram精度: 上一行求和/生成译文中连续取三个单词的取法=2/5

由于生成译文长度大于等于参考译文长度, 所以长度惩罚因子为1, 也即不惩罚。最终结合上述计算过程计算BLEU分数:

$$BLEU = e^{\left(\frac{1}{3} \log \frac{5}{7} + \frac{1}{3} \log \frac{4}{6} + \frac{1}{3} \log \frac{2}{5}\right)} = 0.5753695826647881$$

代码实现:

```
1 from nltk.translate.bleu_score import sentence_bleu
2 import math
3 reference = [['The', 'cat', 'is', 'on', 'the', 'mat'],
4               ['There', 'is', 'a', 'cat', 'on', 'the', 'mat']]
5 candidate = ['The', 'cat', 'the', 'cat', 'on', 'the', 'mat']
6 score = sentence_bleu(reference, candidate,
7                       weights=(1/3, 1/3, 1/3, 0))
8 >>> 0.5753695826647881
```

**注意事项:** 在计算过程中标点符号也会加入考虑。

优点: 自动化计算, 易于计算

缺点: 并不限制n-grams的顺序, 每一个n-gram都是同等重要, 主要考量语句的通顺, 对两句话之间的语义相似性关注较少。

本章代码详见GitHub链接[wzy6642](#)。