Artificial Intelligence Experimental Manual

人工智能实验课程手册（下册）

作者：Yunlong Yu

组织：浙江大学信电学院

时间：July 10, 2023

浙江大学

目录

第 1 章 现代卷积神经网络

内容提要

上一章我们介绍了卷积神经网络的基本原理，本章将介绍现代的卷积神经网络架构，许多现代卷积神经网络的研究都是建立在这一章的基础上的。在本章中的每一个模型都曾一度占据主导地位，其中许多模型都是 ImageNet 竞赛的优胜者。ImageNet 竞赛自 2010 年以来，一直是计算机视觉中监督学习进展的指向标。

- AlexNet。它是第一个在大规模视觉竞赛中击败传统计算机视觉模型的大型神经网络；
- 使用重复块的网络（VGG）。它利用许多重复的神经网络块；
- 网络中的网络（NiN）。它重复使用由卷积层和 1×1 卷积层（用来代替全连接层）来构建深层网络；
- 含并行连结的网络（GoogLeNet）。它使用并行连结的网络，通过不同窗口大小的卷积层和最大汇聚层来并行抽取信息；
- 残差网络（ResNet）。它通过残差块构建跨层的数据通道，是计算机视觉中最流行的体系架构；
- 稠密连接网络（DenseNet）。它的计算成本很高，但给我们带来了更好的效果。

虽然深度神经网络的概念非常简单——将神经网络堆叠在一起。但由于不同的网络架构和超参数选择，这些神经网络的性能会发生很大变化。本章介绍的神经网络是将人类直觉和相关数学见解结合后，经过大量研究试错后的结晶。我们会按时间顺序介绍这些模型，在追寻历史的脉络的同时，帮助培养对该领域发展的直觉。这将有助于研究开发自己的架构。例如，本章介绍的批量规范化（batch normalization）和残差网络（ResNet）为设计和训练深度神经网络提供了重要思想指导。

1.1 深度卷积神经网络（AlexNet）

在 LeNet 提出后，卷积神经网络在计算机视觉和机器学习领域中很有名气。但卷积神经网络并没有主导这些领域。这是因为虽然 LeNet 在小数据集上取得了很好的效果，但是在更大、更真实的数据集上训练卷积神经网络的性能和可行性还有待研究。事实上，在上世纪 90 年代初到 2012 年之间的大部分时间里，神经网络往往被其他机器学习方法超越，如支持向量机（support vector machines）。

在计算机视觉中，直接将神经网络与其他机器学习方法进行比较也许不公平。这是因为，卷积神经网络的输入是由原始像素值或是经过简单预处理（例如居中、缩放）的像素值组成的。但在使用传统机器学习方法时，从业者永远不会将原始像素作为输入。在传统机器学习方法中，计算机视觉流水线是由经过人的手工精心设计的特征流水线组成的。对于这些传统方法，大部分的进展都来自于对特征有了更聪明的想法，并且学习到的算法往往归于事后的解释。

虽然上世纪 90 年代就有了一些神经网络加速卡，但仅靠它们还不足以开发出有大量参数的深层多通道多层卷积神经网络。此外，当时的数据集仍然相对较小。除了这些障碍，训练神经网络的一些关键技巧仍然缺失，包括启发式参数初始化、随机梯度下降的变体、非挤压激活函数和有效的正则化技术。

因此，与训练端到端（从像素到分类结果）系统不同，经典机器学习的流水线看起来更像下面这样：

1. 获取一个有趣的数据集。在早期，收集这些数据集需要昂贵的传感器（在当时最先进的图像也就 100 万像素）。
2. 根据光学、几何学、其他知识以及偶然的发现，手工对特征数据集进行预处理。
3. 通过标准的特征提取算法，如 SIFT（尺度不变特征变换）和 SURF（加速鲁棒特征）或其他手动调整的流水线来输入数据。
4. 将提取的特征送入最喜欢的分类器中（例如线性模型或其它核方法），以训练分类器。

当人们和机器学习研究人员交谈时，会发现机器学习研究人员相信机器学习既重要又美丽：优雅的理论去

证明各种模型的性质。机器学习是一个正在蓬勃发展、严谨且非常有用的领域。然而，当人们和计算机视觉研究人员交谈，会听到一个完全不同的故事。计算机视觉研究人员会告诉一个诡异事实——推动领域进步的是数据特征，而不是学习算法。计算机视觉研究人员相信，从对最终模型精度的影响来说，更大或更干净的数据集、或是稍微改进的特征提取，比任何学习算法带来的进步要大得多。

1.1.1 学习表征

另一种预测这个领域发展的方法———观察图像特征的提取方法。在 2012 年前，图像特征都是机械地计算出来的。事实上，设计一套新的特征函数、改进结果，并撰写论文是盛极一时的潮流。SIFT、SURF 、HOG（定向梯度直方图）、bags of visual words88 和类似的特征提取方法占据了主导地位。

另一组研究人员，包括 Yann LeCun、Geoff Hinton、Yoshua Bengio、Andrew Ng、Shun ichi Amari 和 Juergen Schmidhuber，想法则与众不同：他们认为特征本身应该被学习。此外，他们还认为，在合理地复杂性前提下，特征应该由多个共同学习的神经网络层组成，每个层都有可学习的参数。在机器视觉中，最底层可能检测边缘、颜色和纹理。事实上，Alex Krizhevsky、Ilya Sutskever 和 Geoff Hinton 提出了一种新的卷积神经网络变体 AlexNet。在 2012 年 ImageNet 挑战赛中取得了轰动一时的成绩。AlexNet 以 Alex Krizhevsky 的名字命名，他是论文的第一作者。

有趣的是，在网络的最底层，模型学习到了一些类似于传统滤波器的特征抽取器。图1.1是从 AlexNet 论文复制的，描述了底层图像特征。

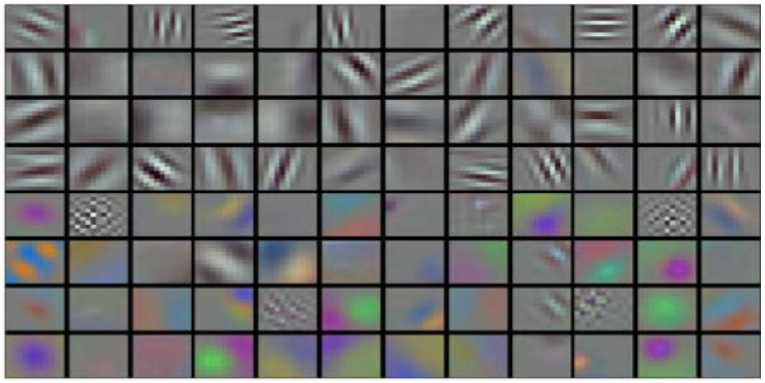


图 1.1: AlexNet 第一层学习到的特征抽取器。

AlexNet 的更高层建立在这些底层表示的基础上，以表示更大的特征，如眼睛、鼻子、草叶等等。而更高的层可以检测整个物体，如人、飞机、狗或飞盘。最终的隐藏神经元可以学习图像的综合表示，从而使属于不同类别的数据易于区分。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尝试都未有突破。深度卷积神经网络的突破出现在 2012 年。突破可归因于两个关键因素。

缺少的成分：数据

包含许多特征的深度模型需要大量的有标签数据，才能显著优于基于凸优化的传统方法（如线性方法和核方法）。然而，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在非自然环境下以低分辨率拍摄的图像。这一状况在 2010 年前后兴起的大数据浪潮中得到改善。2009 年，ImageNet 数据集发布，并发起 ImageNet 挑战赛：要求研究人员从 100 万个样本中训练模型，以区分 1000 个不同类别的对象。ImageNet 数据集由斯坦福教授李飞飞小组的研究人员开发，利用谷歌图像搜索（Google Image Search）对每一类图像进行预筛选，并利用亚马逊众包（Amazon Mechanical Turk）来标注每张图片的相关类别。这种规模是前所未有的。这项被称为 ImageNet 的挑战赛推动了计算机视觉和机器学习研究的发展，挑战研究人员确定哪些模型能够在更大的数据规模下表现最好。

硬件：GPU

1.1.2 AlexNet

2012 年，AlexNet 横空出世。它首次证明了学习到的特征可以超越手工设计的特征。它一举打破了计算机视觉研究的现状。AlexNet 使用了 8 层卷积神经网络，并以很大的优势赢得了 2012 年 ImageNet 图像识别挑战赛。

AlexNet 和 LeNet 的架构非常相似，如图1.2所示。注意，本手册在这里提供的是一个稍微精简版本的 AlexNet，去除了当年需要两个小型 GPU 同时运算的设计特点。

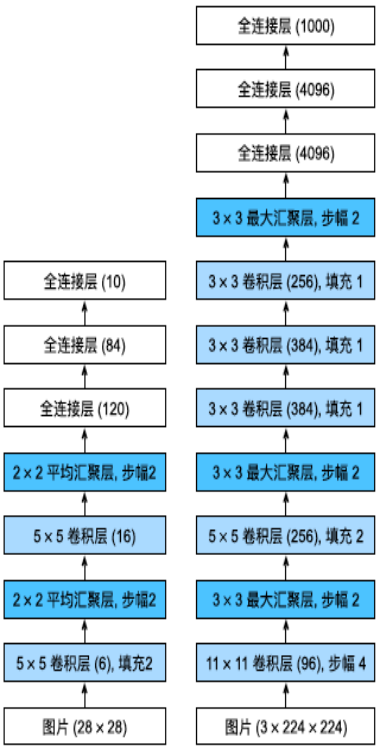


图 1.2: 从 LeNet（左）到 AlexNet（右）。

AlexNet 和 LeNet 的设计理念非常相似，但也存在显著差异。

1. AlexNet 比相对较小的 LeNet5 要深得多。AlexNet 由八层组成：五个卷积层、两个全连接隐藏层和一个全连接输出层。
2. AlexNet 使用 ReLU 而不是 sigmoid 作为其激活函数。

模型设计

在 AlexNet 的第一层，卷积窗口的形状是 11×11。由于 ImageNet 中大多数图像的宽和高比 MNIST 图像的多 10 倍以上，因此，需要一个更大的卷积窗口来捕获目标。第二层中的卷积窗口形状被缩减为 5×5，然后是 3×3。此外，在第一层、第二层和第五层卷积层之后，加入窗口形状为 3×3、步幅为 2 的最大汇聚层。而且，AlexNet 的卷积通道数目是 LeNet 的 10 倍。

在最后一个卷积层后有两个全连接层，分别有 4096 个输出。这两个巨大的全连接层拥有将近 1GB 的模型参数。由于早期 GPU 显存有限，原版的 AlexNet 采用了双数据流设计，使得每个 GPU 只负责存储和计算模型的一半参数。幸运的是，现在 GPU 显存相对充裕，所以现在很少需要跨 GPU 分解模型（因此，本书的 AlexNet 模型在这方面与原始论文稍有不同）。

激活函数

此外，AlexNet 将 sigmoid 激活函数改为更简单的 ReLU 激活函数。一方面，ReLU 激活函数的计算更简单，它不需要如 sigmoid 激活函数那般复杂的求幂运算。另一方面，当使用不同的参数初始化方法时，ReLU 激活函数使训练模型更加容易。当 sigmoid 激活函数的输出非常接近于 0 或 1 时，这些区域的梯度几乎为 0，因此反向传播无法继续更新一些模型参数。相反，ReLU 激活函数在正区间的梯度总是 1。因此，如果模型参数没有正确初始化，sigmoid 函数可能在正区间内得到几乎为 0 的梯度，从而使模型无法得到有效的训练。

容量控制和预处理 AlexNet 通过暂退法控制全连接层的模型复杂度，而 LeNet 只使用了权重衰减。为了进一步扩充数据，AlexNet 在训练时增加了大量的图像增强数据，如翻转、裁切和变色。这使得模型更健壮，更大的样本量有效地减少了过拟合。

```
import torch
from torch import nn
from d2l import torch as d2l
net = nn.Sequential(
    # 这里使用一个11*11的更大窗口来捕捉对象。
    # 同时，步幅为4，以减少输出的高度和宽度。
    # 另外，输出通道的数目远大于LeNet
    nn.Conv2d(1, 96, kernel_size=11, stride=4, padding=1), nn.ReLU(),
    nn.MaxPool2d(kernel_size=3, stride=2),
    # 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，且增大输出通道数
    nn.Conv2d(96, 256, kernel_size=5, padding=2), nn.ReLU(),
    nn.MaxPool2d(kernel_size=3, stride=2),
    # 使用三个连续的卷积层和较小的卷积窗口。
    # 除了最后的卷积层，输出通道的数量进一步增加。
    # 在前两个卷积层之后，汇聚层不用于减少输入的高度和宽度
    nn.Conv2d(256, 384, kernel_size=3, padding=1), nn.ReLU(),nn.Conv2d(384, 384, kernel_size=3,
        padding=1), nn.ReLU(),
    nn.Conv2d(384, 256, kernel_size=3, padding=1), nn.ReLU(),
    nn.MaxPool2d(kernel_size=3, stride=2),
    nn.Flatten(),
    # 这里，全连接层的输出数量是LeNet中的好几倍。使用dropout层来减轻过拟合
    nn.Linear(6400, 4096), nn.ReLU(),
    nn.Dropout(p=0.5),
    nn.Linear(4096, 4096), nn.ReLU(),
    nn.Dropout(p=0.5),
    # 最后是输出层。由于这里使用Fashion-MNIST，所以用类别数为10，而非论文中的1000
    nn.Linear(4096, 10))
```

我们构造一个高度和宽度都为 224 的单通道数据，来观察每一层输出的形状。它与图1.2中的 AlexNet 架构相匹配。

```
X = torch.randn(1, 1, 224, 224)
for layer in net:
    X=layer(X)
    print(layer.__class__.__name__,'output shape:\t',X.shape)
```

```
Conv2d output shape: torch.Size([1, 96, 54, 54])
ReLU output shape: torch.Size([1, 96, 54, 54])
MaxPool2d output shape: torch.Size([1, 96, 26, 26])
Conv2d output shape: torch.Size([1, 256, 26, 26])
ReLU output shape: torch.Size([1, 256, 26, 26])
MaxPool2d output shape: torch.Size([1, 256, 12, 12])
Conv2d output shape: torch.Size([1, 384, 12, 12])
ReLU output shape: torch.Size([1, 384, 12, 12])
Conv2d output shape: torch.Size([1, 384, 12, 12])
ReLU output shape: torch.Size([1, 384, 12, 12])
```

```
Conv2d output shape: torch.Size([1, 256, 12, 12])
ReLU output shape: torch.Size([1, 256, 12, 12])
MaxPool2d output shape: torch.Size([1, 256, 5, 5])
Flatten output shape: torch.Size([1, 6400])
Linear output shape: torch.Size([1, 4096])
ReLU output shape: torch.Size([1, 4096])
Dropout output shape: torch.Size([1, 4096])
Linear output shape: torch.Size([1, 4096])
ReLU output shape: torch.Size([1, 4096])
Dropout output shape: torch.Size([1, 4096])
Linear output shape: torch.Size([1, 10])
```

1.1.3 读取数据集

尽管原文中 AlexNet 是在 ImageNet 上进行训练的，但本书在这里使用的是 Fashion-MNIST 数据集。因为即使在现代 GPU 上，训练 ImageNet 模型，同时使其收敛可能需要数小时或数天的时间。将 AlexNet 直接应用于 Fashion- MNIST 的一个问题是，Fashion-MNIST 图像的分辨率（28×28 像素）低于 ImageNet 图像。为了解决这个问题，我们将它们增加到 224×224（通常来讲这不是一个明智的做法，但在这里这样做是为了有效使用 AlexNet 架构）。这里需要使用 d2l.load_data_fashion_mnist 函数中的 resize 参数执行此调整。

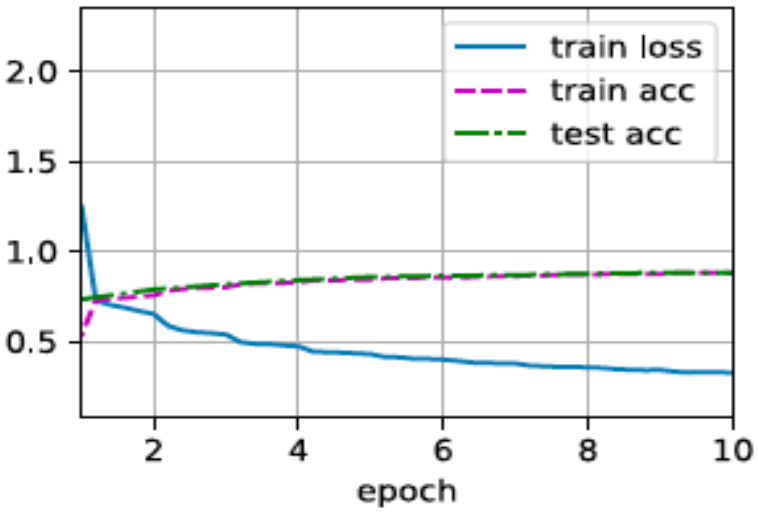
```
batch_size = 128
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)
```

1.1.4 训练 AlexNet

现在 AlexNet 可以开始被训练了。与 LeNet 相比，这里的主要变化是使用更小的学习速率训练，这是因为网络更深更广、图像分辨率更高，训练卷积神经网络就更昂贵。

```
lr, num_epochs = 0.01, 10
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.326, train acc 0.881, test acc 0.879
4187.6 examples/sec on cuda:0
```



1.2 使用块的网络（VGG）

虽然 AlexNet 证明深层神经网络卓有成效，但它没有提供一个通用的模板来指导后续的研究人员设计新的网络。在下面的几个章节中，我们将介绍一些常用于设计深层神经网络的启发式概念。

与芯片设计中工程师从放置晶体管到逻辑元件再到逻辑块的过程类似，神经网络架构的设计也逐渐变得更加抽象。研究人员开始从单个神经元的角度思考问题，发展到整个层，现在又转向块，重复层的模式。

使用块的想法首先出现在牛津大学的视觉几何组（visual geometry group）的 VGG 网络中。通过使用循环和子程序，可以很容易地在任何现代深度学习框架的代码中实现这些重复的架构。

1.2.1 VGG 块

经典卷积神经网络的基本组成部分是下面的这个序列：

1. 带填充以保持分辨率的卷积层；
2. 非线性激活函数，如 ReLU；
3. 汇聚层，如最大汇聚层。

而一个 VGG 块与之类似，由一系列卷积层组成，后面再加上用于空间下采样的最大汇聚层。在最初的 VGG 论文中，作者使用了带有 3×3 卷积核、填充为 1（保持高度和宽度）的卷积层，和带有 2×2 汇聚窗口、步幅为 2（每个块后的分辨率减半）的最大汇聚层。在下面的代码中，我们定义了一个名为 vgg_block 的函数来实现一个 VGG 块。

该函数有三个参数，分别对应于卷积层的数量 num_convs、输入通道的数量 in_channels 和输出通道的数量 out_channels.

```
import torch
from torch import nn
from d2l import torch as d2l

def vgg_block(num_convs, in_channels, out_channels):
    layers = []
    for _ in range(num_convs):
        layers.append(nn.Conv2d(in_channels, out_channels,
        kernel_size=3, padding=1))
        layers.append(nn.ReLU())
        in_channels = out_channels
    layers.append(nn.MaxPool2d(kernel_size=2,stride=2))
    return nn.Sequential(*layers)
```

1.2.2 VGG 网络

与 AlexNet、LeNet 一样，VGG 网络可以分为两部分：第一部分主要由卷积层和汇聚层组成，第二部分由全连接层组成。如图1.3中所示。

VGG 神经网络连接图1.3的几个 VGG 块（在 vgg_block 函数中定义）。其中有超参数变量 conv_arch。该变量指定了每个 VGG 块里卷积层个数和输出通道数。全连接模块则与 AlexNet 中的相同。

原始 VGG 网络有 5 个卷积块，其中前两个块各有一个卷积层，后三个块各包含两个卷积层。第一个模块有 64 个输出通道，每个后续模块将输出通道数量翻倍，直到该数字达到 512。由于该网络使用 8 个卷积层和 3 个全连接层，因此它通常被称为 VGG-11。

```
conv_arch = ((1, 64), (1, 128), (2, 256), (2, 512), (2, 512))
```

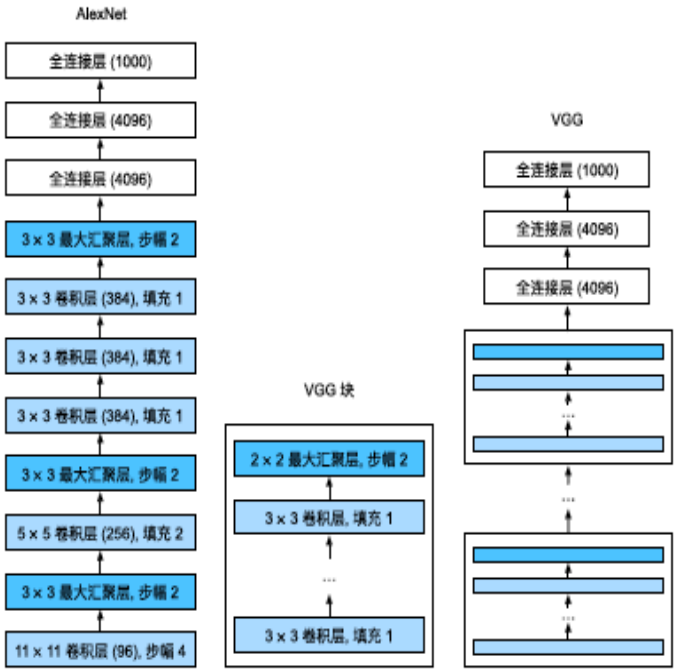


图 1.3: 从 AlexNet 到 VGG，它们本质上都是块设计。

下面的代码实现了 VGG-11。可以通过在 conv_arch 上执行 for 循环来简单实现。

```
def vgg(conv_arch):
    conv_blks = []
    in_channels = 1
    # 卷积层部分
    for (num_convs, out_channels) in conv_arch:
        conv_blks.append(vgg_block(num_convs, in_channels, out_channels))
        in_channels = out_channels

    return nn.Sequential(*conv_blks, nn.Flatten(),
    # 全连接层部分
    nn.Linear(out_channels * 7 * 7, 4096), nn.ReLU(), nn.Dropout(0.5),
    nn.Linear(4096, 4096), nn.ReLU(), nn.Dropout(0.5),
    nn.Linear(4096, 10))

net = vgg(conv_arch)
```

接下来，我们将构建一个高度和宽度为 224 的单通道数据样本，以观察每个层输出的形状。

```
X = torch.randn(size=(1, 1, 224, 224))
for blk in net:
    X = blk(X)
    print(blk.__class__.__name__,'output shape:\t',X.shape)
```

```
Sequential output shape: torch.Size([1, 64, 112, 112])
Sequential output shape: torch.Size([1, 128, 56, 56])
Sequential output shape: torch.Size([1, 256, 28, 28])
Sequential output shape: torch.Size([1, 512, 14, 14])
Sequential output shape: torch.Size([1, 512, 7, 7])
```

```
Flatten output shape: torch.Size([1, 25088])
Linear output shape: torch.Size([1, 4096])
ReLU output shape: torch.Size([1, 4096])
Dropout output shape: torch.Size([1, 4096])
Linear output shape: torch.Size([1, 4096])
ReLU output shape: torch.Size([1, 4096])
Dropout output shape: torch.Size([1, 4096])
Linear output shape: torch.Size([1, 10])
```

正如从代码中所看到的，我们在每个块的高度和宽度减半，最终高度和宽度都为 7。最后再展平表示，送入全连接层处理。

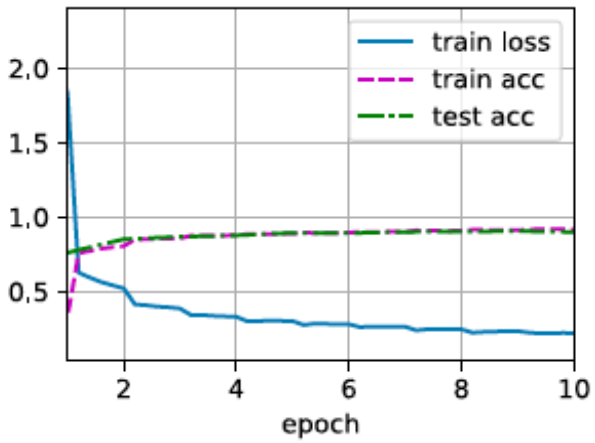
1.2.3 训练模型

由于 VGG-11 比 AlexNet 计算量更大，因此我们构建了一个通道数较少的网络，足够用于训练 Fashion-MNIST 数据集。

```
ratio = 4
small_conv_arch = [(pair[0], pair[1] // ratio) for pair in conv_arch]
net = vgg(small_conv_arch)

lr, num_epochs, batch_size = 0.05, 10, 128
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.220, train acc 0.918, test acc 0.900
2578.4 examples/sec on cuda:0
```



1.3 网络中的网络（NiN）

LeNet、AlexNet 和 VGG 都有一个共同的设计模式：通过一系列的卷积层与汇聚层来提取空间结构特征；然后通过全连接层对特征的表征进行处理。AlexNet 和 VGG 对 LeNet 的改进主要在于如何扩大和加深这两个模块。或者，可以想象在这个过程的早期使用全连接层。然而，如果使用了全连接层，可能会完全放弃表征的空间结构。网络中的网络（NiN）提供了一个非常简单的解决方案：在每个像素的通道上分别使用多层感知机

1.3.1 NiN 块

回想一下，卷积层的输入和输出由四维张量组成，张量的每个轴分别对应样本、通道、高度和宽度。另外，全连接层的输入和输出通常是分别对应于样本和特征的二维张量。NiN 的想法是在每个像素位置（针对每个高度和宽度）应用一个全连接层。如果我们将权重连接到每个空间位置，我们可以将其视为 1×1 卷积层，或作为在每个像素位置上独立作用的全连接层。从另一个𝔼度看，即将空间维度中的每个像素视为单个样本，将通道维度视为不同特征（feature）。

图1.4说明了 VGG 和 NiN 及它们的块之间主要架构差异。NiN 块以一个普通卷积层开始，后面是两个 1×1 的卷积层。这两个 1×1 卷积层充当带有 ReLU 激活函数的逐像素全连接层。第一层的卷积窗口形状通常由用户设置。随后的卷积窗口形状固定为 1×1。

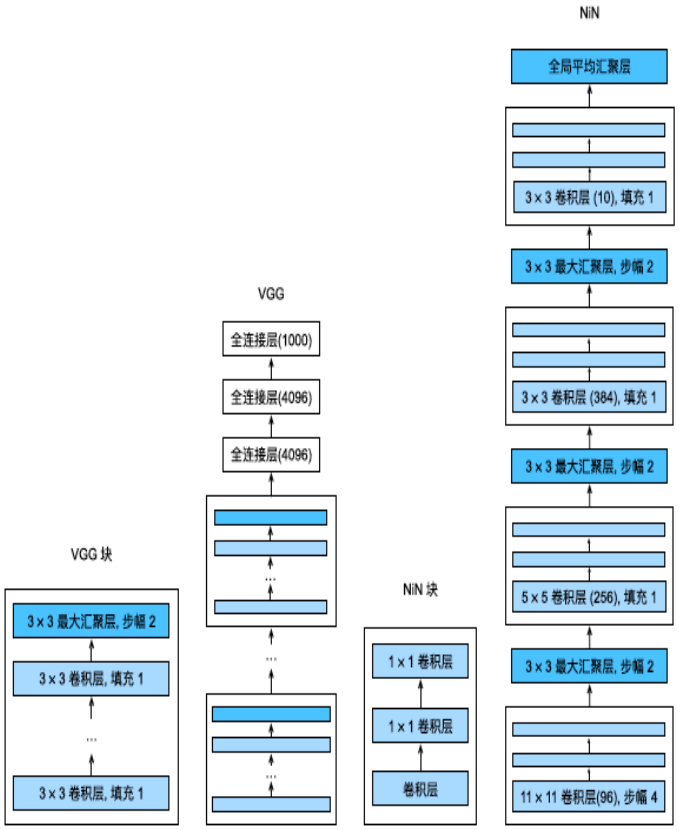


图 1.4: 对比 VGG 和 NiN 及它们的块之间主要架构差异。

```
import torch
from torch import nn
from d2l import torch as d2l

def nin_block(in_channels, out_channels, kernel_size, strides, padding):
    return nn.Sequential(
    nn.Conv2d(in_channels, out_channels, kernel_size, strides, padding),
    nn.ReLU(),
    nn.Conv2d(out_channels, out_channels, kernel_size=1), nn.ReLU(),
    nn.Conv2d(out_channels, out_channels, kernel_size=1), nn.ReLU())
```

1.3.2 NiN 模型

最初的 NiN 网络是在 AlexNet 后不久提出的，显然从中得到了一些启示。NiN 使用窗口形状为 11×11、5×5 和 3×3 的卷积层，输出通道数量与 AlexNet 中的相同。每个 NiN 块后有一个最大汇聚层，汇聚窗口形状为 3×3，步幅为 2。

NiN 和 AlexNet 之间的一个显著区别是 NiN 完全取消了全连接层。相反，NiN 使用一个 NiN 块，其输出通道数等于标签类别的数量。最后放一个全局平均汇聚层（global average pooling layer），生成一个对数几率（logits）。NiN 设计的一个优点是，它显著减少了模型所需参数的数量。然而，在实践中，这种设计有时会增加训练模型的时间。

```
net = nn.Sequential(
    nin_block(1, 96, kernel_size=11, strides=4, padding=0),
    nn.MaxPool2d(3, stride=2),
    nin_block(96, 256, kernel_size=5, strides=1, padding=2),
    nn.MaxPool2d(3, stride=2),
    nin_block(256, 384, kernel_size=3, strides=1, padding=1),
    nn.MaxPool2d(3, stride=2),
    nn.Dropout(0.5),
    # 标签类别数是10
    nin_block(384, 10, kernel_size=3, strides=1, padding=1),
    nn.AdaptiveAvgPool2d((1, 1)),
    # 将四维的输出转成二维的输出，其形状为(批量大小,10)
    nn.Flatten())
```

我们创建一个数据样本来查看每个块的输出形状。

```
X = torch.rand(size=(1, 1, 224, 224))
for layer in net:
    X = layer(X)
    print(layer.__class__.__name__,'output shape:\t', X.shape)
```

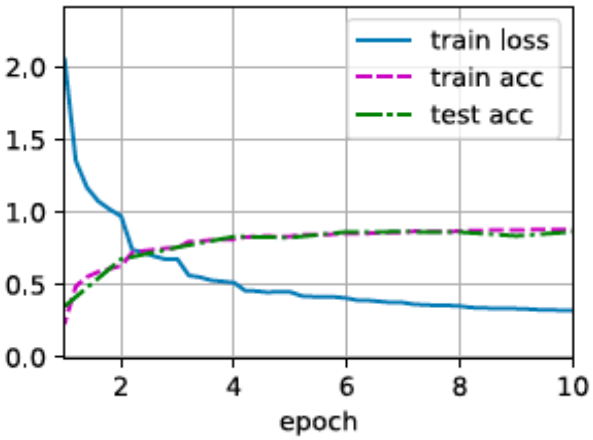
```
Sequential output shape: torch.Size([1, 96, 54, 54])
MaxPool2d output shape: torch.Size([1, 96, 26, 26])
Sequential output shape: torch.Size([1, 256, 26, 26])
MaxPool2d output shape: torch.Size([1, 256, 12, 12])
Sequential output shape: torch.Size([1, 384, 12, 12])
MaxPool2d output shape: torch.Size([1, 384, 5, 5])
Dropout output shape: torch.Size([1, 384, 5, 5])
Sequential output shape: torch.Size([1, 10, 5, 5])
AdaptiveAvgPool2d output shape: torch.Size([1, 10, 1, 1])
Flatten output shape: torch.Size([1, 10])
```

1.3.3 训练模型

和以前一样，我们使用 Fashion-MNIST 来训练模型。训练 NiN 与训练 AlexNet、VGG 时相似。

```
lr, num_epochs, batch_size = 0.1, 10, 128
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.322, train acc 0.881, test acc 0.865
3226.1 examples/sec on cuda:0
```



1.4 批量规范化

训练深层神经网络是十分困难的，特别是在较短的时间内使他们收敛更加棘手。本节将介绍批量规范化（batch normalization），这是一种流行且有效的技术，可持续加速深层网络的收敛速度。

1.4.1 训练深层网络

为什么需要批量规范化层呢？让我们来回顾一下训练神经网络时出现的一些实际挑战。

首先，数据预处理的方式通常会对最终结果产生巨大影响。回想一下我们应用多层感知机来预测房价的例子。使用真实数据时，我们的第一步是标准化输入特征，使其平均值为 0，方差为 1。直观地说，这种标准化可以很好地与我们的优化器配合使用，因为它可以将参数的量级进行统一。

第二，对于典型的多层感知机或卷积神经网络。当我们训练时，中间层中的变量（例如，多层感知机中的仿射变换输出）可能具有更广的变化范围：不论是沿着从输入到输出的层，跨同一层中的单元，或是随着时间的推移，模型参数的随着训练更新变幻莫测。批量规范化的发明者非正式地假设，这些变量分布中的这种偏移可能会阻碍网络的收敛。直观地说，我们可能会猜想，如果一个层的可变值是另一层的 100 倍，这可能需要对学习率进行补偿调整。

第三，更深层的网络很复杂，容易过拟合。这意味着正则化变得更加重要。

批量规范化应用于单个可选层（也可以应用到所有层），其原理如下：在每次训练迭代中，我们首先规范化输入，即通过减去其均值并除以其标准差，其中两者均基于当前小批量处理。接下来，我们应用比例系数和比例偏移。正是由于这个基于批量统计的标准化，才有了批量规范化的名称。

请注意，如果我们尝试使用大小为 1 的小批量应用批量规范化，我们将无法学到任何东西。这是因为在减去均值之后，每个隐藏单元将为 0。所以，只有使用足够大的小批量，批量规范化这种方法才是有效且稳定的。请注意，在应用批量规范化时，批量大小的选择可能比没有批量规范化时更重要。

从形式上来说，用 $x \in B$ 表示一个来自小批量 B 的输入，批量规范化 BN 根据以下表达式转换 x：

$$BN(x) = \gamma \odot \frac{x - \hat{\mu}_B}{\hat{\sigma}_B} + \beta \tag{1.1}$$

$\hat{\mu}_B$ 是小批量 B 的样本均值，$\hat{\sigma}_B$ 是小批量 B 的样本标准差。应用标准化后，生成的小批量的平均值为 0 和单位方差为 1。由于单位方差（与其他一些魔法数）是一个主观的选择，因此我们通常包含拉伸参数（scale）γ

和偏移参数（shift）β，它们的形状与 x 相同。请注意，γ 和 β 是需要与其他模型参数一起学习的参数。

由于在训练过程中，中间层的变化幅度不能过于剧烈，而批量规范化将每一层主动居中，并将它们重新调整为给定的平均值和大小（通过 $\hat{u}_B$ 和 $\hat{\sigma}_B$）。

从形式上来看，我们计算出 (7.5.1) 中的 $\hat{u}_B$ 和 $\hat{\sigma}_B$，如下所示：

$$\hat{u}_B = \frac{1}{|B|}\sum_{x\in B} x, \hat{\sigma}_B = \frac{1}{|B|}\sum_{x\in B}(x-\hat{u}_B)^2 + \epsilon \tag{1.2}$$

我们在方差估计值中添加一个小的常量 $\epsilon > 0$，以确保我们永远不会尝试除以零，即使在经验方差估计值可能消失的情况下也是如此。估计值 $\hat{u}_B$ 和 $\hat{\sigma}_B$ 通过使用平均值和方差的噪声（noise）估计来抵消缩放问题。乍看起来，这种噪声是一个问题，而事实上它是有益的。

现在，我们了解一下批量规范化在实践中是如何工作的。

1.4.2 批量规范化层

回想一下，批量规范化和其他层之间的一个关键区别是，由于批量规范化在完整的小批量上运行，因此我们不能像以前在引入其他层时那样忽略批量大小。我们在下面讨论这两种情况：全连接层和卷积层，他们的批量规范化实现略有不同。

全连接层

通常，我们将批量规范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 x，权重参数和偏置参数分别为 W 和 b，激活函数为 ϕ，批量规范化的运算符为 BN。那么，使用批量规范化的全连接层的输出的计算详情如下:

$$h = \phi(BN(Wx + b)) : \tag{1.3}$$

卷积层

同样，对于卷积层，我们可以在卷积层之后和非线性激活函数之前应用批量规范化。当卷积有多个输出通道时，我们需要对这些通道的“每个”输出执行批量规范化，每个通道都有自己的拉伸（scale）和偏移（shift）参数，这两个参数都是标量。假设我们的小批量包含 m 个样本，并且对于每个通道，卷积的输出具有高度 p 和宽度 q。那么对于卷积层，我们在每个输出通道的 $m \cdot p \cdot q$ 个元素上同时执行每个批量规范化。因此，在计算平均值和方差时，我们会收集所有空间位置的值，然后在给定通道内应用相同的均值和方差，以便在每个空间位置对值进行规范化。

预测过程中的批量规范化

正如我们前面提到的，批量规范化在训练模式和预测模式下的行为通常不同。首先，将训练好的模型用于预测时，我们不再需要样本均值中的噪声以及在微批次上估计每个小批次产生的样本方差了。其次，例如，我们可能需要使用我们的模型对逐个样本进行预测。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，和暂退法一样，批量规范化层在训练模式和预测模式下的计算结果也是不一样的。

1.4.3 从零实现

下面，我们从头开始实现一个具有张量的批量规范化层。

```
import torch
from torch import nn
from d2l import torch as d2l
```

```
def batch_norm(X, gamma, beta, moving_mean, moving_var, eps, momentum):
# 通过is_grad_enabled来判断当前模式是训练模式还是预测模式
    if not torch.is_grad_enabled():
```

```
        # 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
        X_hat = (X - moving_mean) / torch.sqrt(moving_var + eps)
    else:
        assert len(X.shape) in (2, 4)
        if len(X.shape) == 2:
            # 使用全连接层的情况，计算特征维上的均值和方差
            mean = X.mean(dim=0)
            var = ((X - mean) ** 2).mean(dim=0)
        else:
            # 使用二维卷积层的情况，计算通道维上（axis=1）的均值和方差。
            # 这里我们需要保持X的形状以便后面可以做广播运算
            mean = X.mean(dim=(0, 2, 3), keepdim=True)
            var = ((X - mean) ** 2).mean(dim=(0, 2, 3), keepdim=True)
        # 训练模式下，用当前的均值和方差做标准化
        X_hat = (X - mean) / torch.sqrt(var + eps)
        # 更新移动平均的均值和方差
        moving_mean = momentum * moving_mean + (1.0 - momentum) * mean
        moving_var = momentum * moving_var + (1.0 - momentum) * var
    Y = gamma * X_hat + beta # 缩放和移位
    return Y, moving_mean.data, moving_var.data
```

我们现在可以创建一个正确的 BatchNorm 层。这个层将保持适当的参数：拉伸 gamma 和偏移 beta, 这两个参数将在训练过程中更新。此外，我们的层将保存均值和方差的移动平均值，以便在模型预测期间随后使用。撇开算法细节，注意我们实现层的基础设计模式。通常情况下，我们用一个单独的函数定义其数学原理，比如说 batch_norm。然后，我们将此功能集成到一个自定义层中，其代码主要处理数据移动到训练设备（如 GPU）、分配和初始化任何必需的变量、跟踪移动平均线（此处为均值和方差）等问题。为了方便起见，我们并不担心在这里自动推断输入形状，因此我们需要指定整个特征的数量。不用担心，深度学习框架中的批量规范化 API 将为我们解决上述问题，我们稍后将展示这一点。

```
class BatchNorm(nn.Module):
    # num_features：完全连接层的输出数量或卷积层的输出通道数。
    # num_dims：2表示完全连接层，4表示卷积层
    def __init__(self, num_features, num_dims):
        super().__init__()
        if num_dims == 2:
            shape = (1, num_features)
        else:
            shape = (1, num_features, 1, 1)

        # 参与求梯度和迭代的拉伸和偏移参数，分别初始化成1和0
        self.gamma = nn.Parameter(torch.ones(shape))
        self.beta = nn.Parameter(torch.zeros(shape))
        # 非模型参数的变量初始化为0和1
        self.moving_mean = torch.zeros(shape)
        self.moving_var = torch.ones(shape)


    def forward(self, X):
        # 如果X不在内存上，将moving_mean和moving_var
```

```
        # 复制到X所在显存上
        if self.moving_mean.device != X.device:
            self.moving_mean = self.moving_mean.to(X.device)
            self.moving_var = self.moving_var.to(X.device)
        # 保存更新过的moving_mean和moving_var
        Y, self.moving_mean, self.moving_var = batch_norm(X, self.gamma, self.beta, self.moving_mean,
             self.moving_var, eps=1e-5, momentum=0.9)
        return Y
```

1.4.4 使用批量规范化层的 LeNet

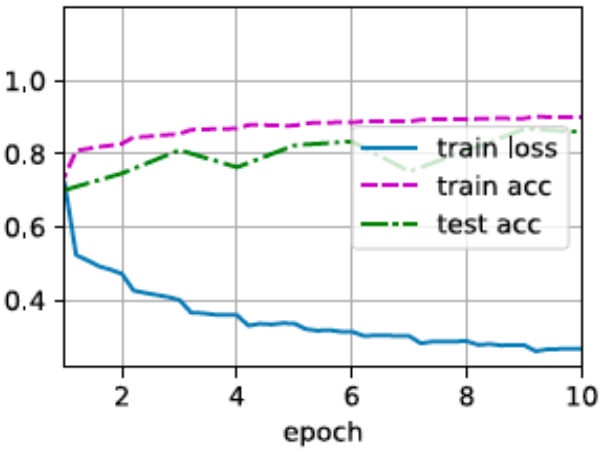
为了更好理解如何应用 BatchNorm，下面我们将其应用于 LeNet 模型。回想一下，批量规范化是在卷积层或全连接层之后、相应的激活函数之前应用的。

```
net = nn.Sequential(
    nn.Conv2d(1, 6, kernel_size=5), BatchNorm(6, num_dims=4), nn.Sigmoid(),
    nn.AvgPool2d(kernel_size=2, stride=2),
    nn.Conv2d(6, 16, kernel_size=5), BatchNorm(16, num_dims=4), nn.Sigmoid(),
    nn.AvgPool2d(kernel_size=2, stride=2), nn.Flatten(),
    nn.Linear(16*4*4, 120), BatchNorm(120, num_dims=2), nn.Sigmoid(),
    nn.Linear(120, 84), BatchNorm(84, num_dims=2), nn.Sigmoid(),
    nn.Linear(84, 10))
```

和以前一样，我们将在 Fashion-MNIST 数据集上训练网络。这个代码与我们第一次训练 LeNet 时几乎完全相同，主要区别在于学习率大得多。

```
lr, num_epochs, batch_size = 1.0, 10, 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.267, train acc 0.900, test acc 0.860
40878.1 examples/sec on cuda:0
```



让我们来看看从第一个批量规范化层中学到的拉伸参数 gamma 和偏移参数 beta。

```
net[1].gamma.reshape((-1,)), net[1].beta.reshape((-1,))
```

```
(tensor([2.2493, 1.6559, 2.7877, 2.4000, 4.1816, 3.5716], device='cuda:0',
grad_fn=<ReshapeAliasBackward0>),
tensor([-1.3887, -0.4102, -1.0248, 2.1779, -2.4067, -3.6746], device='cuda:0',
grad_fn=<ReshapeAliasBackward0>))
```

1.4.5 简明实现

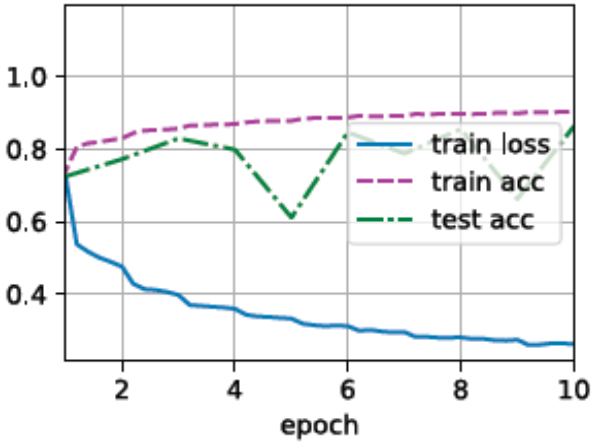
除了使用我们刚刚定义的 BatchNorm，我们也可以直接使用深度学习框架中定义的 BatchNorm。该代码看起来几乎与我们上面的代码相同。

```
net = nn.Sequential(
    nn.Conv2d(1, 6, kernel_size=5), nn.BatchNorm2d(6), nn.Sigmoid(),
    nn.AvgPool2d(kernel_size=2, stride=2),
    nn.Conv2d(6, 16, kernel_size=5), nn.BatchNorm2d(16), nn.Sigmoid(),
    nn.AvgPool2d(kernel_size=2, stride=2), nn.Flatten(),
    nn.Linear(256, 120), nn.BatchNorm1d(120), nn.Sigmoid(),
    nn.Linear(120, 84), nn.BatchNorm1d(84), nn.Sigmoid(),
    nn.Linear(84, 10))
```

下面，我们使用相同超参数来训练模型。请注意，通常高级 API 变体运行速度快得多，因为它的代码已编译为 C++ 或 CUDA，而我们的自定义代码由 Python 实现。

```
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.263, train acc 0.902, test acc 0.862
71480.6 examples/sec on cuda:0
```



1.5 残差网络（ResNet）

1.5.1 残差块

让我们聚焦于神经网络局部：如图图1.5所示，假设我们的原始输入为 x，而希望学出的理想映射为 $f(x)$（作为图1.5上方激活函数的输入）。图1.5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残差映射 $f(x)-x$。残差映射在现实中往往更容易优化。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 $f(x)$，我们只需将图1.5中右图虚线框内上方的加权运算（如仿射）的权重和偏置参数设成 0，那么 $f(x)$ 即为恒等映射。实际中，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。图1.5右图是 ResNet 的基础架构 残差块（residual block）。在残差块中，输入可通过跨层数据线路更快地向前传播。

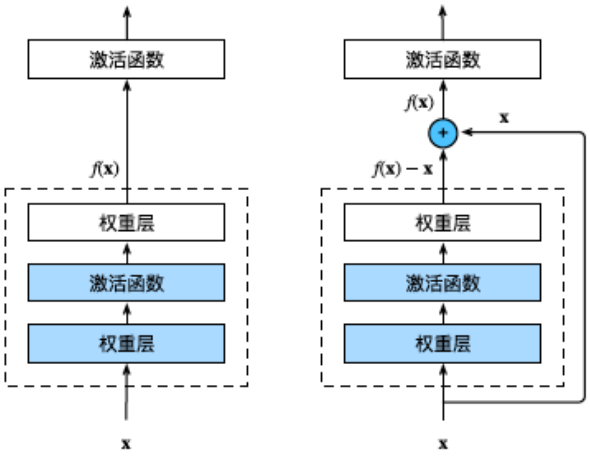


图 1.5: 一个正常块（左图）和一个残差块（右图）。

ResNet 沿用了 VGG 完整的 3 × 3 卷积层设计。残差块里首先有 2 个有相同输出通道数的 3 × 3 卷积层。每个卷积层后接一个批量规范化层和 ReLU 激活函数。然后我们通过跨层数据通路，跳过这 2 个卷积运算，将输入直接加在最后的 ReLU 激活函数前。这样的设计要求 2 个卷积层的输出与输入形状一样，从而使它们可以相加。如果想改变通道数，就需要引入一个额外的 1 × 1 卷积层来将输入变换成需要的形状后再做相加运算。残差块的实现如下：

```
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l


class Residual(nn.Module): #@save
    def __init__(self, input_channels, num_channels, use_1x1conv=False, strides=1):

        super().__init__()
        self.conv1 = nn.Conv2d(input_channels, num_channels,
        kernel_size=3, padding=1, stride=strides)
        self.conv2 = nn.Conv2d(num_channels, num_channels,
        kernel_size=3, padding=1)
        if use_1x1conv:
            self.conv3 = nn.Conv2d(input_channels, num_channels,
            kernel_size=1, stride=strides)
        else:
            self.conv3 = None
        self.bn1 = nn.BatchNorm2d(num_channels)
        self.bn2 = nn.BatchNorm2d(num_channels)
    def forward(self, X):
        Y = F.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
```

```
        if self.conv3:
            X = self.conv3(X)
        Y += X
        return F.relu(Y)
```

如图1.6所示，此代码生成两种类型的网络：一种是当 use_1x1conv=False 时，应用 ReLU 非线性函数之前，将输入添加到输出。另一种是当 use_1x1conv=True 时，添加通过 1 × 1 卷积调整通道和分辨率。

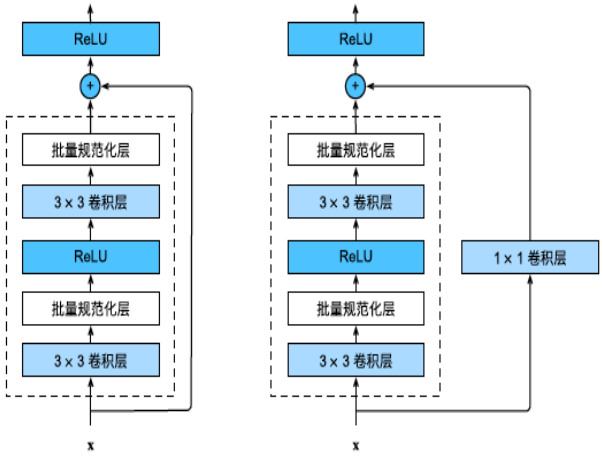


图 1.6: 包含以及不包含 1 × 1 卷积层的残差块。

下面我们来查看输入和输出形状一致的情况。

```
blk = Residual(3,3)
X = torch.rand(4, 3, 6, 6)
Y = blk(X)
Y.shape
```

```
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。

```
blk = Residual(3,6, use_1x1conv=True, strides=2)
blk(X).shape
```

```
torch.Size([4, 6, 3, 3])
```

1.5.2 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 × 7 卷积层后，接步幅为 2 的 3 × 3 的最大汇聚层。不同之处在于 ResNet 每个卷积层后增加了批量规范化层。

```
b1 = nn.Sequential(nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3),
    nn.BatchNorm2d(64), nn.ReLU(),
    nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

ResNetå 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 的最大汇聚层，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，我们对第一个模块做了特别处理。

```
def resnet_block(input_channels, num_channels, num_residuals,
first_block=False):
    blk = []
    for i in range(num_residuals):
        if i == 0 and not first_block:
            blk.append(Residual(input_channels, num_channels,
            use_1x1conv=True, strides=2))
        else:
            blk.append(Residual(num_channels, num_channels))
    return blk
```

接着在 ResNet 加入所有残差块，这里每个模块使用 2 个残差块。

```
b2 = nn.Sequential(*resnet_block(64, 64, 2, first_block=True))
b3 = nn.Sequential(*resnet_block(64, 128, 2))
b4 = nn.Sequential(*resnet_block(128, 256, 2))
b5 = nn.Sequential(*resnet_block(256, 512, 2))
```

最后，与 GoogLeNet 一样，在 ResNet 中加入全局平均汇聚层，以及全连接层输出。

```
net = nn.Sequential(b1, b2, b3, b4, b5,
    nn.AdaptiveAvgPool2d((1,1)),
    nn.Flatten(), nn.Linear(512, 10))
```

每个模块有 4 个卷积层（不包括恒等映射的 1×1 卷积层）。加上第一个 7×7 卷积层和最后一个全连接层，共有 18 层。因此，这种模型通常被称为 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。图1.7描述了完整的 ResNet-18。

在训练 ResNet 之前，让我们观察一下 ResNet 中不同模块的输入形状是如何变化的。在之前所有架构中，分辨率降低，通道数量增加，直到全局平均汇聚层聚集所有特征。

```
X = torch.rand(size=(1, 1, 224, 224))
for layer in net:
    X = layer(X)
    print(layer.__class__.__name__,'output shape:\t', X.shape)
```

```
Sequential output shape: torch.Size([1, 64, 56, 56])
Sequential output shape: torch.Size([1, 64, 56, 56])
Sequential output shape: torch.Size([1, 128, 28, 28])
Sequential output shape: torch.Size([1, 256, 14, 14])
Sequential output shape: torch.Size([1, 512, 7, 7])
AdaptiveAvgPool2d output shape: torch.Size([1, 512, 1, 1])
Flatten output shape: torch.Size([1, 512])
Linear output shape: torch.Size([1, 10])
```

1.5.3 训练模型

同之前一样，我们在 Fashion-MNIST 数据集上训练 ResNet。

```
lr, num_epochs, batch_size = 0.05, 10, 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=96)
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

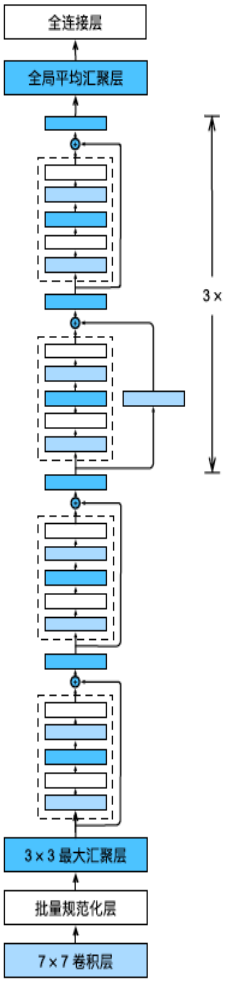


图 1.7: ResNet-18 架构。

```
loss 0.008, train acc 0.999, test acc 0.898
4650.1 examples/sec on cuda:0
```

1.6 作业：实战 Kaggle 比赛图像分类 (CIFAR-10)

CIFAR-10 是计算机视觉领域中的一个重要的数据集。本章作业将运用我们在前几节中学到的知识来参加 CIFAR-10 图像分类问题的 Kaggle 竞赛，比赛的网址是 https://www.kaggle.com/c/cifar-10。

首先，导入竞赛所需的包和模块。

```
import collections
import math
import os
import shutil
import pandas as pd
import torch
import torchvision
from torch import nn
from d2l import torch as d2l
```

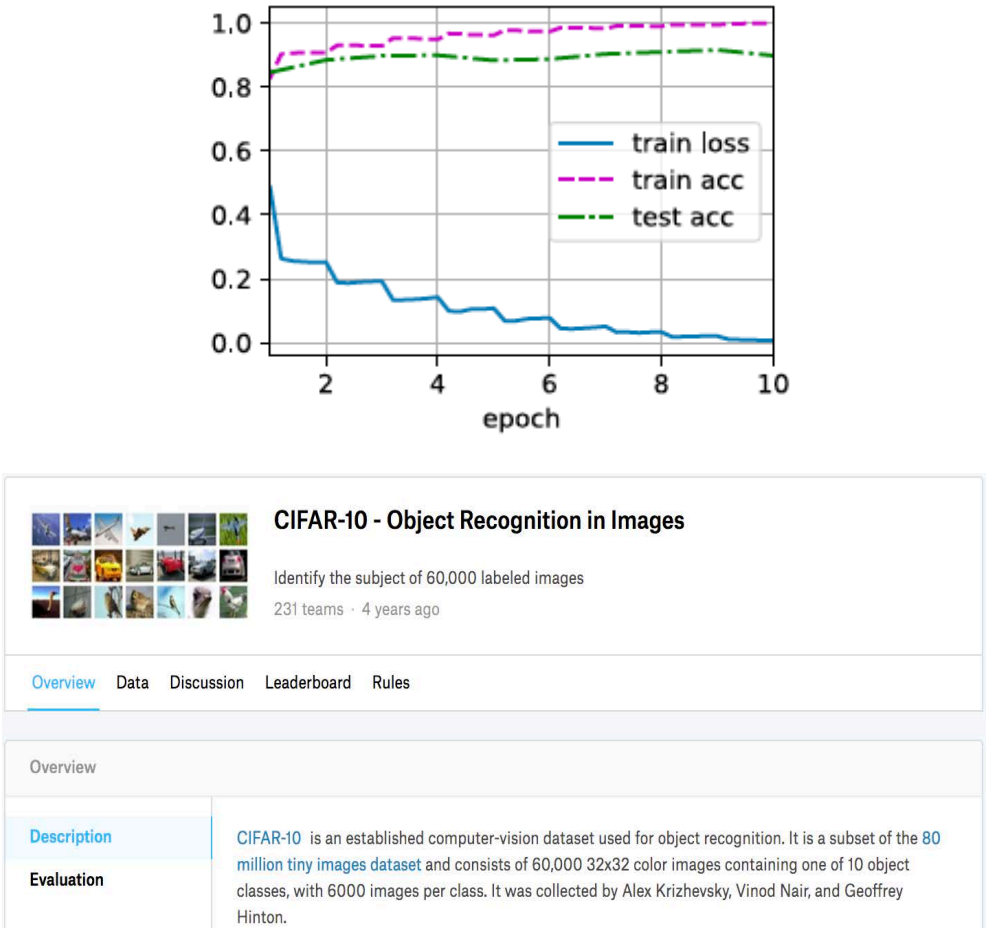


图 1.8: CIFAR-10 图像分类竞赛页面上的信息。竞赛用的数据集可通过点击“Data”选项卡获取。

1.6.1 获取并组织数据集

比赛数据集分为训练集和测试集，其中训练集包含 50000 张、测试集包含 300000 张图像。在测试集中，10000 张图像将被用于评估，而剩下的 290000 张图像将不会被进行评估，包含它们只是为了防止手动标记测试集并提交标记结果。两个数据集中的图像都是 png 格式，高度和宽度均为 32 像素并有三个颜色通道（RGB）。这些图片共涵盖 10 个类别：飞机、汽车、鸟类、猫、鹿、狗、青蛙、马、船和卡车。图1.8的左上角显示了数据集中飞机、汽车和鸟类的一些图像

下载数据集

登录 Kaggle 后，我们可以点击图1.8中显示的 CIFAR-10 图像分类竞赛网页上的“Data”选项卡，然后单击“Download All”按钮下载数据集。在../data 中解压下载的文件并在其中解压缩 train.7z 和 test.7z 后，在以下路径中可以找到整个数据集：

- ../data/cifar-10/train/[1-50000].png
- ../data/cifar-10/test/[1-300000].png
- ../data/cifar-10/trainLabels.csv
- ../data/cifar-10/sampleSubmission.csv

train 和 test 文件夹分别包含训练和测试图像，trainLabels.csv 含有训练图像的标签，sample_submission. csv 是提交文件的范例。

整理数据集

读取数据集

定义模型

定义训练函数
训练和验证模型
在 Kaggle 上对测试集进行分类