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 14, 2023

浙江大学

目录

第 1 章 计算机视觉

内容提要

近年来，深度学习一直是提高计算机视觉系统性能的变革力量。无论是医疗诊断、自动驾驶，还是智能滤波器、摄像头监控，许多计算机视觉领域的应用都与我们当前和未来的生活密切相关。可以说，最先进的计算机视觉应用与深度学习几乎是不可分割的。

1.1 图像增广

大型数据集是成功应用深度神经网络的先决条件。图像增广在对训练图像进行一系列的随机变化之后，生成相似但不同的训练样本，从而扩大了训练集的规模。此外，应用图像增广的原因是，随机改变训练样本可以减少模型对某些属性的依赖，从而提高模型的泛化能力。例如，我们可以以不同的方式裁剪图像，使感兴趣的对象出现在不同的位置，减少模型对于对象出现位置的依赖。我们还可以调整亮度、颜色等因素来降低模型对颜色的敏感度。可以说，图像增广技术对于 AlexNet 的成功是必不可少的。本节将讨论这项广泛应用于计算机视觉的技术。

```
%matplotlib inline
import torch
import torchvision
from torch import nn
from d2l import torch as d2l
```

1.1.1 常用的图像增广方法

在对常用图像增广方法的探索时，我们将使用下面这个尺寸为 400×500 的图像作为示例。

```
d2l.set_figsize()
img = d2l.Image.open('../img/cat1.jpg')
d2l.plt.imshow(img);
```

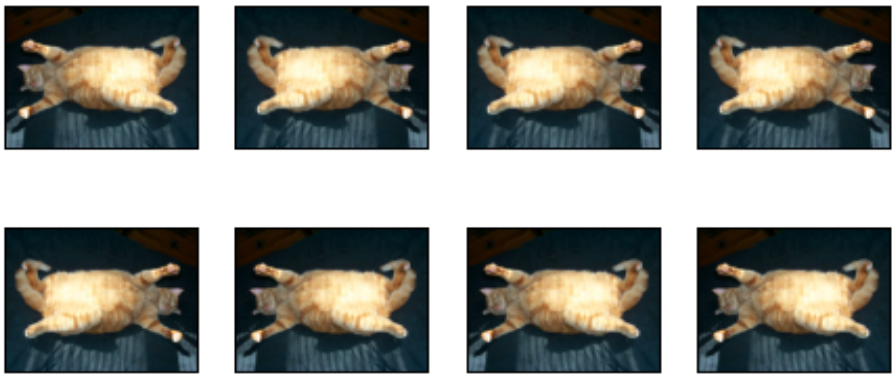
大多数图像增广方法都具有一定的随机性。为了便于观察图像增广的效果，我们下面定义辅助函数 apply。此函数在输入图像 img 上多次运行图像增广方法 aug 并显示所有结果。

```
def apply(img, aug, num_rows=2, num_cols=4, scale=1.5):
    Y = [aug(img) for _ in range(num_rows * num_cols)]
    d2l.show_images(Y, num_rows, num_cols, scale=scale)
```

翻转和裁剪

左右翻转图像通常不会改变对象的类别。这是最早且最广泛使用的图像增广方法之一。接下来，我们使用 transforms 模块来创建 RandomFlipLeftRight 实例，这样就各有 50% 的几率使图像向左或向右翻转。

```
apply(img, torchvision.transforms.RandomHorizontalFlip())
```



上下翻转图像不如左右图像翻转那样常用。但是，至少对于这个示例图像，上下翻转不会妨碍识别。接下来，我们创建一个 RandomFlipTopBottom 实例，使图像各有 50% 的几率向上或向下翻转。

```
apply(img, torchvision.transforms.RandomVerticalFlip())
```

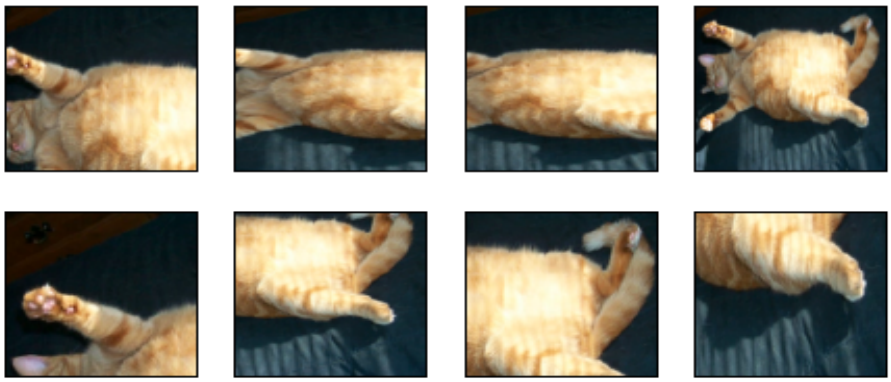
在我们使用的示例图像中，猫位于图像的中间，但并非所有图像都是这样。我们可以通过对图像进行随机裁剪，使物体以不同的比例出现在图像的不同位置。这也可以降低模型对目标位置的敏感性。

下面的代码将随机裁剪一个面积为原始面积 10% 到 100% 的区域，该区域的宽高比从 0.5 ~ 2 之间随机取值。然后，区域的宽度和高度都被缩放到 200 像素。在本章中，a 和 b 之间的随机数指的是在区间 [a, b] 中通过均匀采样获得的连续值。

```
shape_aug = torchvision.transforms.RandomResizedCrop(
(200, 200), scale=(0.1, 1), ratio=(0.5, 2))
apply(img, shape_aug)
```

改变颜色

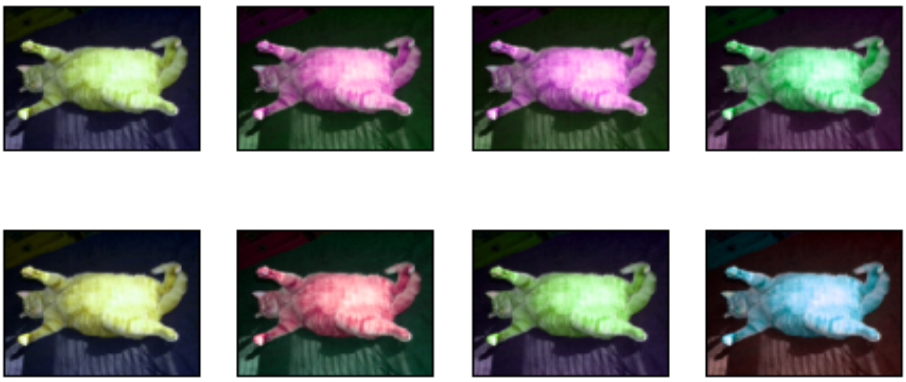
另一种增广方法是改变颜色。我们可以改变图像颜色的四个方面：亮度、对比度、饱和度和色调。在下面的示例中，我们随机更改图像的亮度，随机值为原始图像的 50%（1- 0.5）到 150%（1 + 0.5）之间。



```
apply(img, torchvision.transforms.ColorJitter(
brightness=0.5, contrast=0, saturation=0, hue=0))
```

同样，我们可以随机更改图像的色调。

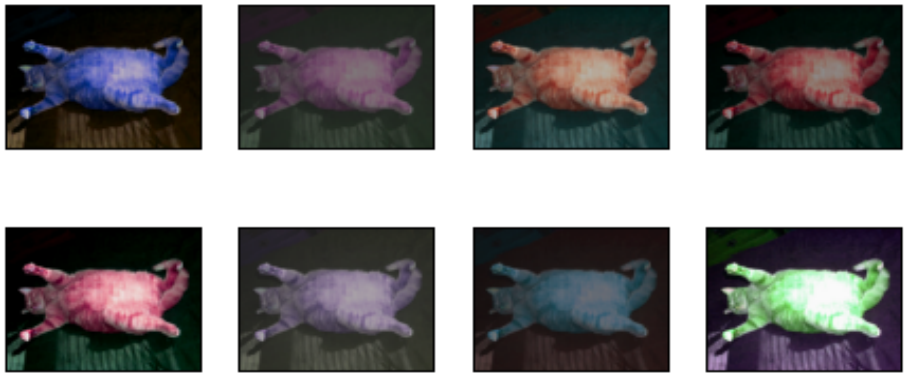
```
apply(img, torchvision.transforms.ColorJitter(
brightness=0, contrast=0, saturation=0, hue=0.5))
```



我们还可以创建一个 RandomColorJitter 实例，并设置如何同时随机更改图像的亮度（brightness）、对比度（contrast）、饱和度（saturation）和色调（hue）。

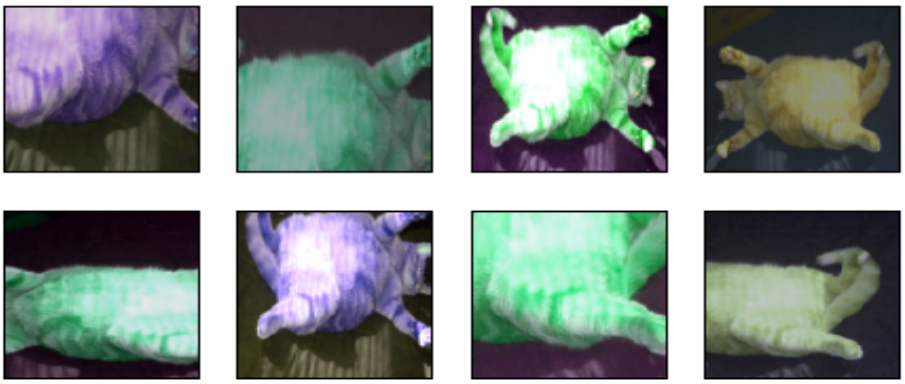
```
color_aug = torchvision.transforms.ColorJitter(
brightness=0.5, contrast=0.5, saturation=0.5, hue=0.5)
apply(img, color_aug)
```

结合多种图像增广方法



在实践中，我们将结合多种图像增广方法。比如，我们可以通过使用一个 Compose 实例来综合上面定义的不同的图像增广方法，并将它们应用到每个图像。

```
augs = torchvision.transforms.Compose([
torchvision.transforms.RandomHorizontalFlip(), color_aug, shape_aug])
apply(img, augs)
```



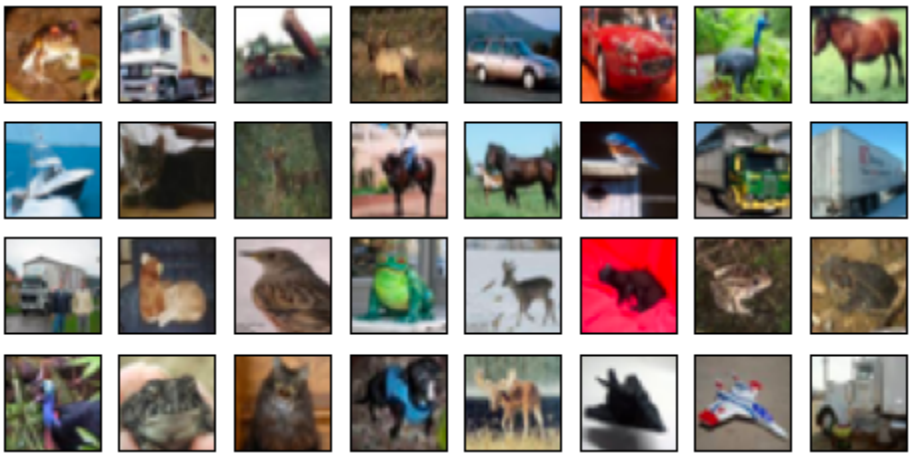
1.1.2 使用图像增广进行训练

让我们使用图像增广来训练模型。这里，我们使用 CIFAR-10 数据集，而不是我们之前使用的 Fashion-MNIST 数据集。这是因为 Fashion-MNIST 数据集中对象的位置和大小已被规范化，而 CIFAR-10 数据集中对象的颜色和大小差异更明显。CIFAR-10 数据集中的前 32 个训练图像如下所示。

```
all_images = torchvision.datasets.CIFAR10(train=True, root="../data", download=True)
d2l.show_images([all_images[i][0] for i in range(32)], 4, 8, scale=0.8);
```

为了在预测过程中得到确切的结果，我们通常对训练样本只进行图像增广，且在预测过程中不使用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 实例将一批图像转换为深度学习框架所要求的格式，即形状为（批量大小，通道数，高度，宽度）的 32 位浮点数，取值范围为 0 ~ 1。

```
train_augs = torchvision.transforms.Compose([
torchvision.transforms.RandomHorizontalFlip(),
torchvision.transforms.ToTensor()])

test_augs = torchvision.transforms.Compose([
torchvision.transforms.ToTensor()])
```



接下来，我们定义一个辅助函数，以便于读取图像和应用图像增广。PyTorch 数据集提供的 transform 参数应用图像增广来转化图像.

```
def load_cifar10(is_train, augs, batch_size):
    dataset = torchvision.datasets.CIFAR10(root="../data", train=is_train,
    transform=augs, download=True)
    dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
    shuffle=is_train, num_workers=d2l.get_dataloader_workers())
    return dataloader
```

多 GPU 训练

```
#@save
def train_batch_ch13(net, X, y, loss, trainer, devices):
    """用多GPU进行小批量训练"""
    if isinstance(X, list):

        X = [x.to(devices[0]) for x in X]
    else:
        X = X.to(devices[0])
    y = y.to(devices[0])
    net.train()
    trainer.zero_grad()
    pred = net(X)
    l = loss(pred, y)
    l.sum().backward()
    trainer.step()
    train_loss_sum = l.sum()
    train_acc_sum = d2l.accuracy(pred, y)
    return train_loss_sum, train_acc_sum
```

```
#@save
def train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs,
devices=d2l.try_all_gpus()):
    """用多GPU进行模型训练"""
    timer, num_batches = d2l.Timer(), len(train_iter)
```

```
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs], ylim=[0, 1], legend=['train loss', '
        train acc', 'test acc'])
    net = nn.DataParallel(net, device_ids=devices).to(devices[0])
    for epoch in range(num_epochs):
        # 4个维度：储存训练损失，训练准确度，实例数，特点数
        metric = d2l.Accumulator(4)
        for i, (features, labels) in enumerate(train_iter):
            timer.start()
            l, acc = train_batch_ch13(
                    net, features, labels, loss, trainer, devices)
            metric.add(l, acc, labels.shape[0], labels.numel())
            timer.stop()
            if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
                animator.add(epoch + (i + 1) / num_batches, (metric[0] / metric[2], metric[1] / metric
                    [3],None))

        test_acc = d2l.evaluate_accuracy_gpu(net, test_iter)
        animator.add(epoch + 1, (None, None, test_acc))

    print(f'loss {metric[0] / metric[2]:.3f}, train acc 'f'{metric[1] / metric[3]:.3f}, test acc {test
        _acc:.3f}')
    print(f'{metric[2] * num_epochs / timer.sum():.1f} examples/sec on 'f'{str(devices)}')
```

现在，我们可以定义 train_with_data_aug 函数，使用图像增广来训练模型。该函数获取所有的 GPU，并使用 Adam 作为训练的优化算法，将图像增广应用于训练集，最后调用刚刚定义的用于训练和评估模型的 train_ch13 函数。

```
batch_size, devices, net = 256, d2l.try_all_gpus(), d2l.resnet18(10, 3)
def init_weights(m):
    if type(m) in [nn.Linear, nn.Conv2d]:
        nn.init.xavier_uniform_(m.weight)
net.apply(init_weights)
def train_with_data_aug(train_augs, test_augs, net, lr=0.001):
    train_iter = load_cifar10(True, train_augs, batch_size)
    test_iter = load_cifar10(False, test_augs, batch_size)
    loss = nn.CrossEntropyLoss(reduction="none")
    trainer = torch.optim.Adam(net.parameters(), lr=lr)
    train_ch13(net, train_iter, test_iter, loss, trainer, 10, devices)
```

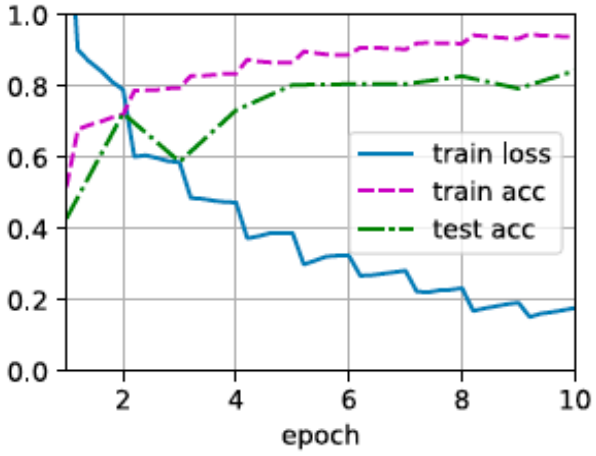
让我们使用基于随机左右翻转的图像增广来训练模型。

```
train_with_data_aug(train_augs, test_augs, net)
```

```
loss 0.178, train acc 0.938, test acc 0.844
5647.3 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```

1.2 微调

前面的一些章节介绍了如何在只有 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 1000 万的图像和 1000 类的物体。然而，我们平常接触到的数据集的规模通常在这两者之间。



假如我们想识别图片中不同类型的椅子，然后向用户推荐购买链接。一种可能的方法是首先识别 100 把普通椅子，为每把椅子拍摄 1000 张不同角度的图像，然后在收集的图像数据集上训练一个分类模型。尽管这个椅子数据集可能大于 Fashion-MNIST 数据集，但实例数量仍然不到 ImageNet 中的十分之一。适合 ImageNet 的复杂模型可能会在这个椅子数据集上过拟合。此外，由于训练样本数量有限，训练模型的准确性可能无法满足实际要求。

为了解决上述问题，一个显而易见的解决方案是收集更多的数据。但是，收集和标记数据可能需要大量的时间和金钱。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资金。尽管目前的数据收集成本已大幅降低，但这一成本仍不能忽视。

另一种解决方案是应用迁移学习（transfer learning）将从源数据集学到的知识迁移到目标数据集。例如，尽管 ImageNet 数据集中的大多数图像与椅子无关，但在此数据集上训练的模型可能会提取更通用的图像特征，这有助于识别边缘、纹理、形状和对象组合。这些类似的特征也可能有效地识别椅子。

1.2.1 步骤

本节将介绍迁移学习中的常见技巧: 微调（fine-tuning）。如图1.1所示，微调包括以下四个步骤。

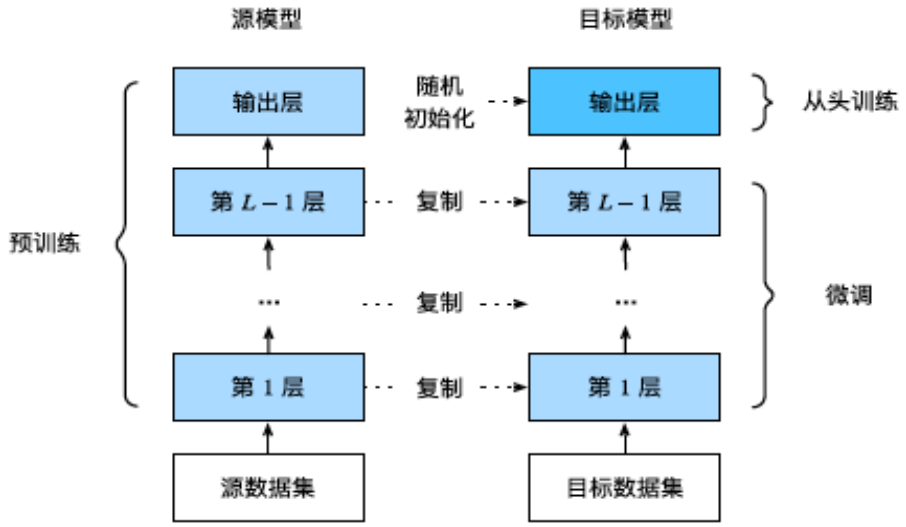


图 1.1: 微调架构。

1.2.2 热狗识别

让我们通过具体案例演示微调：热狗识别。我们将在一个小型数据集上微调 ResNet 模型。该模型已在 ImageNet 数据集上进行了预训练。这个小型数据集包含数千张包含热狗和不包含热狗的图像，我们将使用微调模型来识别图像中是否包含热狗。

```
%matplotlib inline
import os
import torch
import torchvision
from torch import nn
from d2l import torch as d2l
```

获取数据集

我们使用的热狗数据集来源于网络。该数据集包含 1400 张热狗的“正类”图像，以及包含尽可能多的其他食物的“负类”图像。含着两个类别的 1000 张图片用于训练，其余的则用于测试。

我们创建两个实例来分别读取训练和测试数据集中的所有图像文件。

```
train_imgs = torchvision.datasets.ImageFolder(os.path.join(data_dir, 'train'))
test_imgs = torchvision.datasets.ImageFolder(os.path.join(data_dir, 'test'))
```

下面显示了前 8 个正类样本图片和最后 8 张负类样本图片。正如所看到的，图像的大小和纵横比各有不同。

```
hotdogs = [train_imgs[i][0] for i in range(8)]
not_hotdogs = [train_imgs[-i - 1][0] for i in range(8)]
d2l.show_images(hotdogs + not_hotdogs, 2, 8, scale=1.4);
```

在训练期间，我们首先从图像中裁切随机大小和随机长宽比的区域，然后将该区域缩放为 224×224 输入图像。在测试过程中，我们将图像的高度和宽度都缩放到 256 像素，然后裁剪中央 224×224 区域作为输入。此外，对于 RGB（红、绿和蓝）颜色通道，我们分别标准化每个通道。具体而言，该通道的每个值减去该通道的平均值，然后将结果除以该通道的标准差。

```
# 使用RGB通道的均值和标准差，以标准化每个通道
normalize = torchvision.transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])

train_augs = torchvision.transforms.Compose([torchvision.transforms.RandomResizedCrop(224),
torchvision.transforms.RandomHorizontalFlip(),torchvision.transforms.ToTensor(),
normalize])

test_augs = torchvision.transforms.Compose([torchvision.transforms.Resize([256, 256]),
torchvision.transforms.CenterCrop(224),torchvision.transforms.ToTensor(),
normalize])
```

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。在这里，我们指定 pretrained=True 以自动下载预训练的模型参数。如果首次使用此模型，则需要连接互联网才能下载。

```
pretrained_net = torchvision.models.resnet18(pretrained=True)
```

预训练的源模型实例包含许多特征层和一个输出层 fc。此划分的主要目的是促进对除输出层以外所有层的模型参数进行微调。下面给出了源模型的成员变量 fc。

```
pretrained_net.fc
```

```
Linear(in_features=512, out_features=1000, bias=True)
```

在 ResNet 的全局平均汇聚层后，全连接层转换为 ImageNet 数据集的 1000 个类输出。之后，我们构建一个新的神经网络作为目标模型。它的定义方式与预训练源模型的定义方式相同，只是最终层中的输出数量被设置为目标数据集中的类数（而不是 1000 个）。

在下面的代码中，目标模型 finetune_net 中成员变量 features 的参数被初始化为源模型相应层的模型参数。由于模型参数是在 ImageNet 数据集上预训练的，并且足够好，因此通常只需要较小的学习率即可微调这些参数。

成员变量 output 的参数是随机初始化的，通常需要更高的学习率才能从头开始训练。假设 Trainer 实例中的学习率为 η，我们将成员变量 output 中参数的学习率设置为 10η。

```
finetune_net = torchvision.models.resnet18(pretrained=True)
finetune_net.fc = nn.Linear(finetune_net.fc.in_features, 2)
nn.init.xavier_uniform_(finetune_net.fc.weight);
```

微调模型

首先，我们定义了一个训练函数 train_fine_tuning，该函数使用微调，因此可以多次调用。

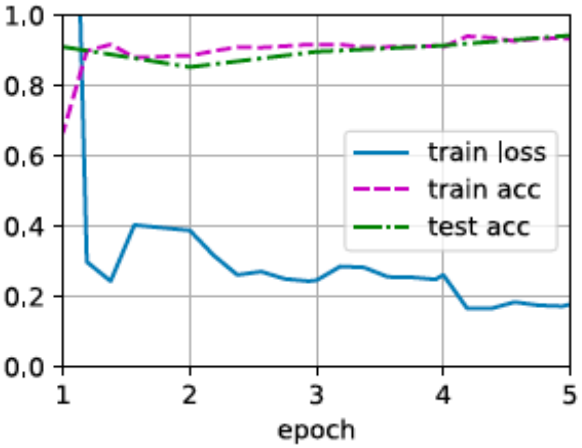
```
# 如果param_group=True，输出层中的模型参数将使用十倍的学习率
def train_fine_tuning(net, learning_rate, batch_size=128, num_epochs=5,
param_group=True):
    train_iter = torch.utils.data.DataLoader(torchvision.datasets.ImageFolder(
    os.path.join(data_dir, 'train'), transform=train_augs),
    batch_size=batch_size, shuffle=True)
    test_iter = torch.utils.data.DataLoader(torchvision.datasets.ImageFolder(
    os.path.join(data_dir, 'test'), transform=test_augs),
    batch_size=batch_size)
    devices = d2l.try_all_gpus()

    loss = nn.CrossEntropyLoss(reduction="none")

    if param_group:
        params_1x = [param for name, param in net.named_parameters() if name not in ["fc.weight", "fc.
            bias"]]
        trainer = torch.optim.SGD([{'params': params_1x},
            {'params': net.fc.parameters(),
            'lr': learning_rate * 10}],
            lr=learning_rate, weight_decay=0.001)
    else:
        trainer = torch.optim.SGD(net.parameters(), lr=learning_rate,
        weight_decay=0.001)
    d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs,
```

```
    devices)
```

我们使用较小的学习率，通过微调预训练获得的模型参数。

```
train_fine_tuning(finetune_net, 5e-5)
```

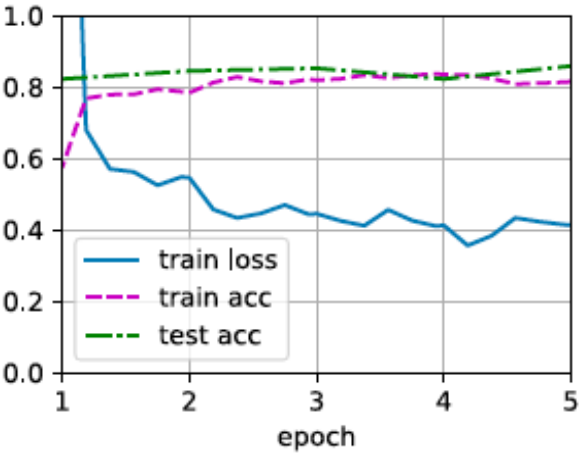
```
loss 0.177, train acc 0.932, test acc 0.943
968.4 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



为了进行比较，我们定义了一个相同的模型，但是将其所有模型参数初始化为随机值。由于整个模型需要从头开始训练，因此我们需要使用更大的学习率。

```
scratch_net = torchvision.models.resnet18()
scratch_net.fc = nn.Linear(scratch_net.fc.in_features, 2)
train_fine_tuning(scratch_net, 5e-4, param_group=False)
```

```
loss 0.413, train acc 0.815, test acc 0.859
1627.1 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



意料之中，微调模型往往表现更好，因为它的初始参数值更有效。

1.3 风格迁移

摄影爱好者也许接触过滤波器。它能改变照片的颜色风格，从而使风景照更加锐利或者令人像更加美白。但一个滤波器通常只能改变照片的某个方面。如果要照片达到理想中的风格，可能需要尝试大量不同的组合。这个过程的复杂程度不亚于模型调参。

本节将介绍如何使用卷积神经网络，自动将一个图像中的风格应用在另一图像之上，即风格迁移（style transfer）。这里我们需要两张输入图像：一张是内容图像，另一张是风格图像。我们将使用神经网络修改内容图像，使其在风格上接近风格图像。例如，图1.2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拍摄的风景照，而风格图像则是一幅主题为秋天橡树的油画。最终输出的合成图像应用了风格图像的油画笔触让整体颜色更加鲜艳，同时保留了内容图像中物体主体的形状。

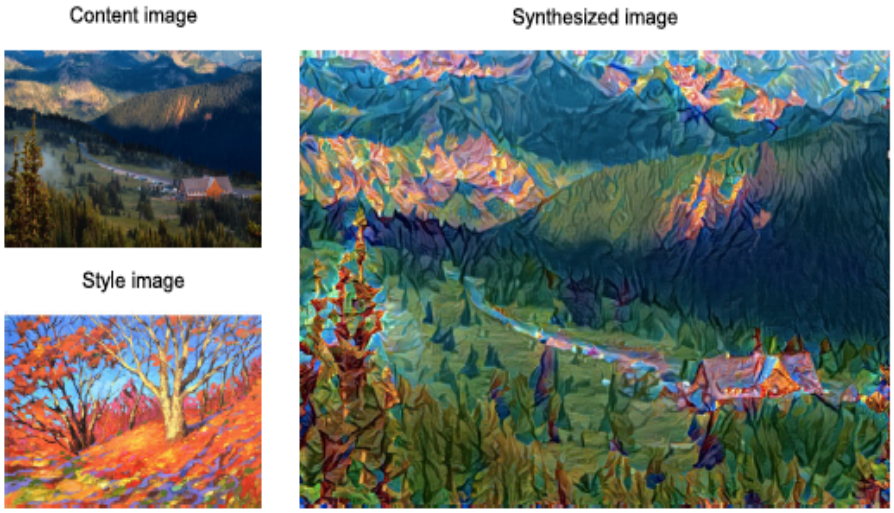


图 1.2: 输入内容图像和风格图像，输出风格迁移后的合成图像.

1.3.1 方法

图1.3用简单的例子阐述了基于卷积神经网络的风格迁移方法。首先，我们初始化合成图像，例如将其初始化为内容图像。该合成图像是风格迁移过程中唯一需要更新的变量，即风格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。这个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风格特征。以图1.3为例，这里选取的预训练的神经网络含有 3 个卷积层，其中第二层输出内容特征，第一层和第三层输出风格特征。

接下来，我们通过前向传播（实线箭头方向）计算风格迁移的损失函数，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。风格迁移常用的损失函数由 3 部分组成：

1. 内容损失使合成图像与内容图像在内容特征上接近；
2. 风格损失使合成图像与风格图像在风格特征上接近；
3. 全变分损失则有助于减少合成图像中的噪点。

最后，当模型训练结束时，我们输出风格迁移的模型参数，即得到最终的合成图像。在下面，我们将通过代码来进一步了解风格迁移的技术细节。

1.3.2 阅读内容和风格图像

首先，我们读取内容和风格图像。从打印出的图像坐标轴可以看出，它们的尺寸并不一样。

```
%matplotlib inline
```

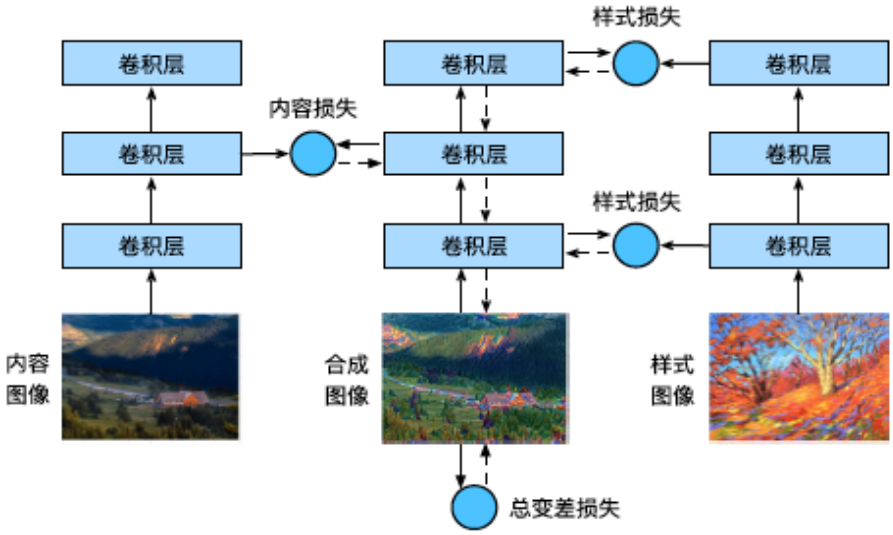
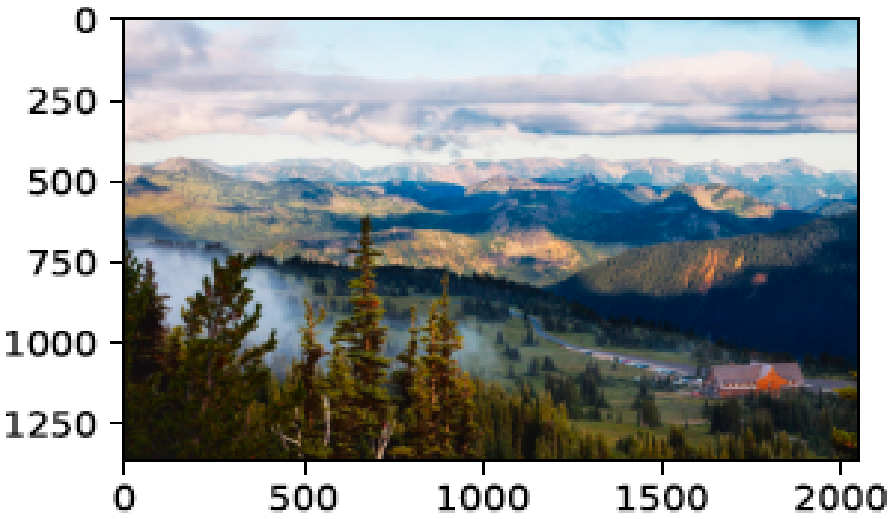


图 1.3: 基于卷积神经网络的风格迁移。实线箭头和虚线箭头分别表示前向传播和反向传播。

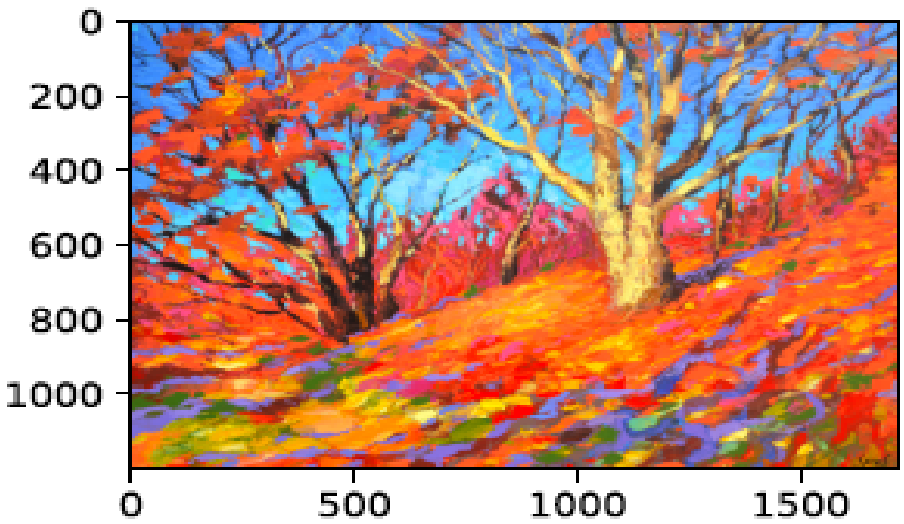
```
import torch
import torchvision
from torch import nn
from d2l import torch as d2l
d2l.set_figsize()
content_img = d2l.Image.open('../img/rainier.jpg')
d2l.plt.imshow(content_img);
```




```
style_img = d2l.Image.open('../img/autumn-oak.jpg')
d2l.plt.imshow(style_img);
```

1.3.3 预处理和后处理

下面，定义图像的预处理函数和后处理函数。预处理函数 preprocess 对输入图像在 RGB 三个通道分别做标准化，并将结果变换成卷积神经网络接受的输入格式。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。由于图像打印函数要求每个像素的浮点数值在 0 ~ 1 之间，我们对小于 0 和大于 1 的值分别取 0 和 1。




```
rgb_mean = torch.tensor([0.485, 0.456, 0.406])
rgb_std = torch.tensor([0.229, 0.224, 0.225])

def preprocess(img, image_shape):
    transforms = torchvision.transforms.Compose([
        torchvision.transforms.Resize(image_shape),
        torchvision.transforms.ToTensor(),
        torchvision.transforms.Normalize(mean=rgb_mean, std=rgb_std)])
    return transforms(img).unsqueeze(0)

def postprocess(img):
    img = img[0].to(rgb_std.device)
    img = torch.clamp(img.permute(1, 2, 0) * rgb_std + rgb_mean, 0, 1)
    return torchvision.transforms.ToPILImage()(img.permute(2, 0, 1))
```

1.3.4 抽取图像特征

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

```
pretrained_net = torchvision.models.vgg19(pretrained=True)
```

为了抽取图像的内容特征和🄴格特征，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，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，即内容层，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风格，这些图层也称为风格层。VGG 网络使用了 5 个卷积块。实验中，我们选择第四卷积块的最后一个卷积层作为内容层，选择每个卷积块的第一个卷积层作为🄴格层。这些层的索引可以通过打印 pretrained_net 实例获取。

```
style_layers, content_layers = [0, 5, 10, 19, 28], [25]
```

使用 VGG 层抽取特征时，我们只需要用到从输入层到最靠近输出层的内容层或风格层之间的所有层。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。

```
net = nn.Sequential(*[pretrained_net.features[i] for i in range(max(content_layers + style_layers) +
    1)])
```

给定输入 X，如果我们简单地调用前向传播 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，并保留内容层和风格层的输出。

```
def extract_features(X, content_layers, style_layers):
    contents = []
    styles = []
    for i in range(len(net)):
        X = net[i](X)
        if i in style_layers:
            styles.append(X)
        if i in content_layers:
            contents.append(X)
    return contents, styles
```

下面定义两个函数：get_contents 函数对内容图像抽取内容特征；get_styles 函数对风格图像抽取风格特征。因为在训练时无须改变预训练的 VGG 的模型参数，所以我们可以在训练开始之前就提取出内容特征和风格特征。由于合成图像是风格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风格特征。

```
def get_contents(image_shape, device):
    content_X = preprocess(content_img, image_shape).to(device)
    contents_Y, _ = extract_features(content_X, content_layers, style_layers)
    return content_X, contents_Y

def get_styles(image_shape, device):
    style_X = preprocess(style_img, image_shape).to(device)
    _, styles_Y = extract_features(style_X, content_layers, style_layers)
    return style_X, styles_Y
```

1.3.5 定义损失函数

下面我们来描述F格迁移的损失函数。它由内容损失、风格损失和全变分损失 3 部分组成。

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。

```
def content_loss(Y_hat, Y):
    # 我们从动态计算梯度的树中分离目标：
    # 这是一个规定的值，而不是一个变量。
    return torch.square(Y_hat - Y.detach()).mean()
```

风格损失

风格损失与内容损失类似，也通过平方误差函数衡量合成图像与风格图像在风格上的差异。为了表达风格层输出的风格，我们先通过 extract_features 函数计算风格层的输出。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，我们可以将此输出转换为矩阵 X，其有 c 行和 h_w 列。这个矩阵可以被看作由 c 个长度为 h_w 的向量 $x_1, ..., x_c$ 组合而成的。其中向量 x_i 代表了通道 ě 上的风格特征。

在这些向量的格拉姆矩阵 $XX \in \mathbb{R}^{c \times c}$ 中，i 行 j 列的元素 x_{ij} 即向量 x_i 和 x_j 的内积。它表达了通道 i 和通道 j 上风格特征的相关性。我们用这样的格拉姆矩阵来表达风格层输出的风格。需要注意的是，当 h_w 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。为了让风格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。

```
def gram(X):
    num_channels, n = X.shape[1], X.numel() // X.shape[1]
    X = X.reshape((num_channels, n))
    return torch.matmul(X, X.T) / (num_channels * n)
```

自然地，风格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风格图像的风格层输出。这里假设基于风格图像的格拉姆矩阵 gram_Y 已经预先计算好了。

```
def style_loss(Y_hat, gram_Y):
    return torch.square(gram(Y_hat) - gram_Y.detach()).mean()
```

全变分损失

有时候，我们学到的合成图像里面有大量高频噪点，即有特别亮或者特别暗的颗粒像素。一种常用的去噪方法是全变分去噪（total variation denoising）：假设 $x_{i,j}$ 表示坐标 (i, j) 处的像素值，降低全变分损失

$$\sum_{i,j} |x_{i,j} - x_{i+1,j}| + |x_{i,j} - x_{i,j+1}| \tag{1.1}$$

能够尽可能使邻近的像素值相似。

```
def tv_loss(Y_hat):
    return 0.5 * (torch.abs(Y_hat[:, :, 1:, :] - Y_hat[:, :, :-1, :]).mean() +
    torch.abs(Y_hat[:, :, :, 1:] - Y_hat[:, :, :, :-1]).mean())
```

损失函数

风格转移的损失函数是内容损失、风格损失和总变化损失的加权和。通过调节这些权重超参数，我们可以权衡合成图像在保留内容、迁移风格以及去噪三方面的相对重要性。

```
content_weight, style_weight, tv_weight = 1, 1e3, 10
def compute_loss(X, contents_Y_hat, styles_Y_hat, contents_Y, styles_Y_gram):
    # 分别计算内容损失、风格损失和全变分损失
    contents_l = [content_loss(Y_hat, Y) * content_weight for Y_hat, Y in zip(
    contents_Y_hat, contents_Y)]
    styles_l = [style_loss(Y_hat, Y) * style_weight for Y_hat, Y in zip(
    styles_Y_hat, styles_Y_gram)]
    tv_l = tv_loss(X) * tv_weight
    # 对所有损失求和
    l = sum(10 * styles_l + contents_l + [tv_l])
    return contents_l, styles_l, tv_l, l
```

1.3.6 初始化合成图像

在风格迁移中，合成的图像是训练期间唯一需要更新的变量。因此，我们可以定义一个简单的模型 SynthesizedImage，并将合成的图像视为模型参数。模型的前向传播只需返回模型参数即可。

```
class SynthesizedImage(nn.Module):
    def __init__(self, img_shape, **kwargs):
```

```
        super(SynthesizedImage, self).__init__(**kwargs)
        self.weight = nn.Parameter(torch.rand(*img_shape))
    def forward(self):
        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。风格图像在各个风格层的格拉姆矩阵 styles_Y_gram 将在训练前预先计算好。

```
def get_inits(X, device, lr, styles_Y):
    gen_img = SynthesizedImage(X.shape).to(device)
    gen_img.weight.data.copy_(X.data)
    trainer = torch.optim.Adam(gen_img.parameters(), lr=lr)
    styles_Y_gram = [gram(Y) for Y in styles_Y]
    return gen_img(), styles_Y_gram, trainer
```

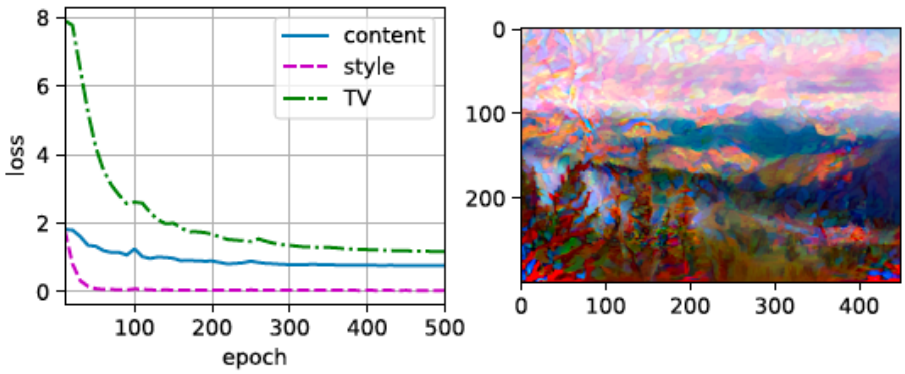
1.3.7 训练模型

在训练模型进行风格迁移时，我们不断抽取合成图像的内容特征和风格特征，然后计算损失函数。下面定义了训练循环。

```
def train(X, contents_Y, styles_Y, device, lr, num_epochs, lr_decay_epoch):
    X, styles_Y_gram, trainer = get_inits(X, device, lr, styles_Y)
    scheduler = torch.optim.lr_scheduler.StepLR(trainer, lr_decay_epoch, 0.8)
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
    xlim=[10, num_epochs],
    legend=['content', 'style', 'TV'],
    ncols=2, figsize=(7, 2.5))
    for epoch in range(num_epochs):
        trainer.zero_grad()
        contents_Y_hat, styles_Y_hat = extract_features(
        X, content_layers, style_layers)
        contents_l, styles_l, tv_l, l = compute_loss(
        X, contents_Y_hat, styles_Y_hat, contents_Y, styles_Y_gram)
        l.backward()
        trainer.step()
        scheduler.step()
        if (epoch + 1) % 10 == 0:
            animator.axes[1].imshow(postprocess(X))
            animator.add(epoch + 1, [float(sum(contents_l)),
            float(sum(styles_l)), float(tv_l)])
    return X
```

现在我们训练模型：首先将内容图像和风格图像的高和宽分别调整为 300 和 450 像素，用内容图像来初始化合成图像。

```
device, image_shape = d2l.try_gpu(), (300, 450)
net = net.to(device)
content_X, contents_Y = get_contents(image_shape, device)
_, styles_Y = get_styles(image_shape, device)
output = train(content_X, contents_Y, styles_Y, device, 0.3, 500, 50)
```


content
style
TV
loss
epoch
0
2
4
6
8
100
200
300
400
500
0
100
200
300
400