图像增广

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。

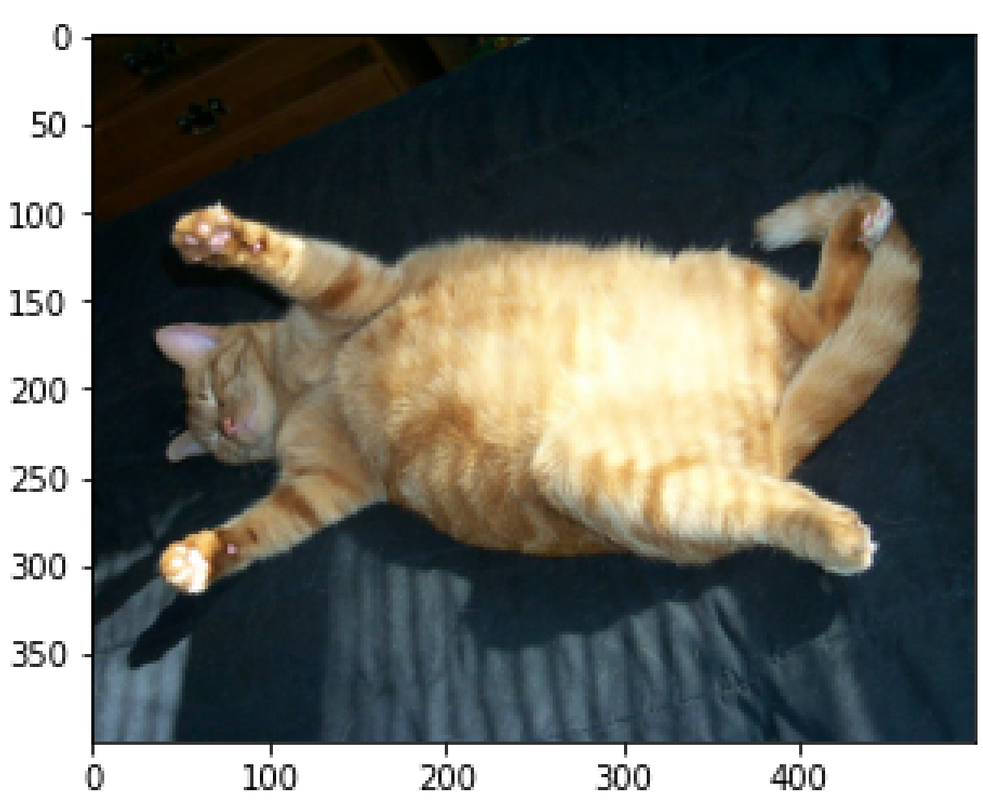
此外，应用图像增广的原因是，随机改变训练样本可以减少模型对某些属性的依赖，从而提高模型的泛化能力。

In [1]:
```
%matplotlib inline
import os
import torch
import torchvision
from torch import nn
from PIL import Image
from d2l import torch as d2l
from matplotlib import pyplot as plt
```

常用的图像增广方法

使用下面这个尺寸为400 × 500的图像作为示例，对常用图像增广方法进行探索。

In [2]:
```
figsize=(3.5, 2.5)
img = Image.open('./figs/cat1.jpg')
plt.imshow(img);
```



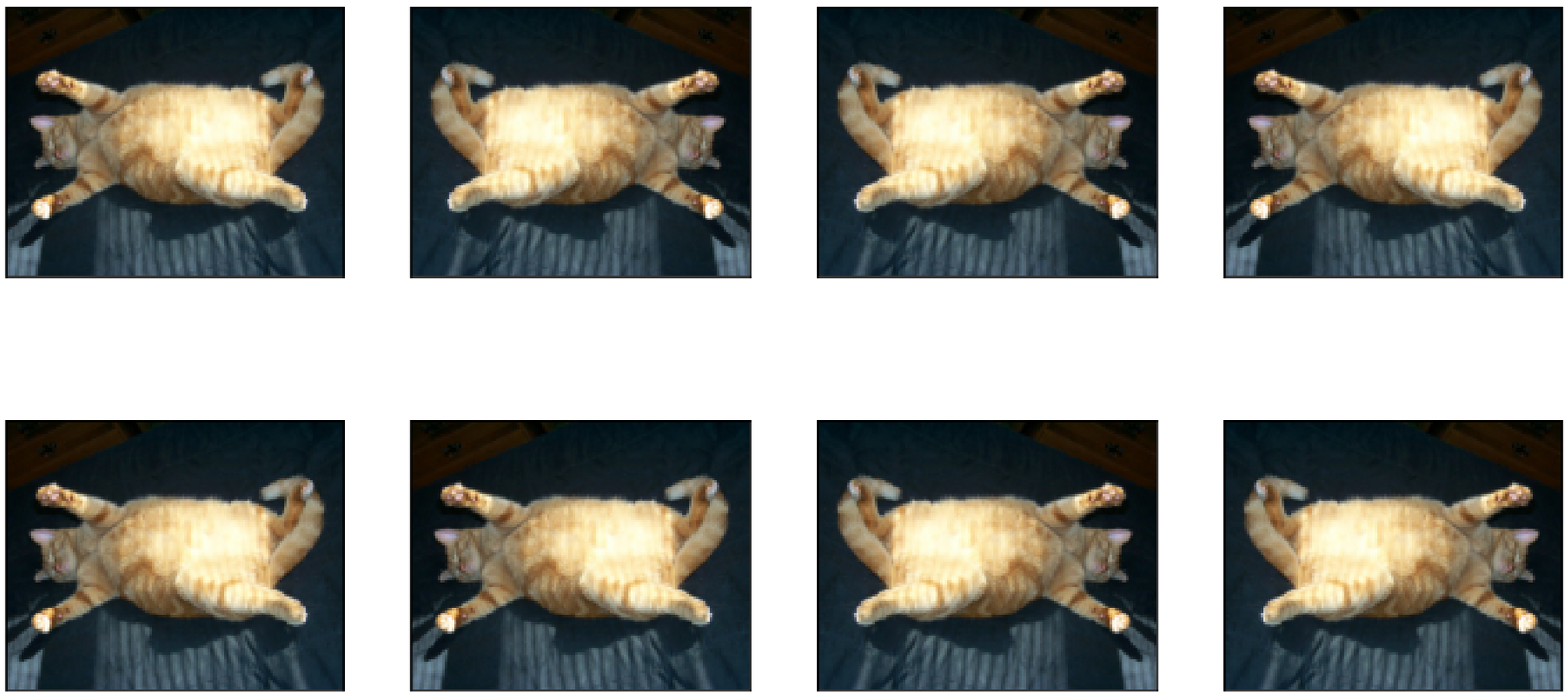
In [3]:
```
def apply(img, aug, num_rows=2, num_cols=4, scale=3):
    Y = [aug(img) for _ in range(num_rows * num_cols)]
    d2l.show_images(Y, num_rows, num_cols, scale=scale)
```

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，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。

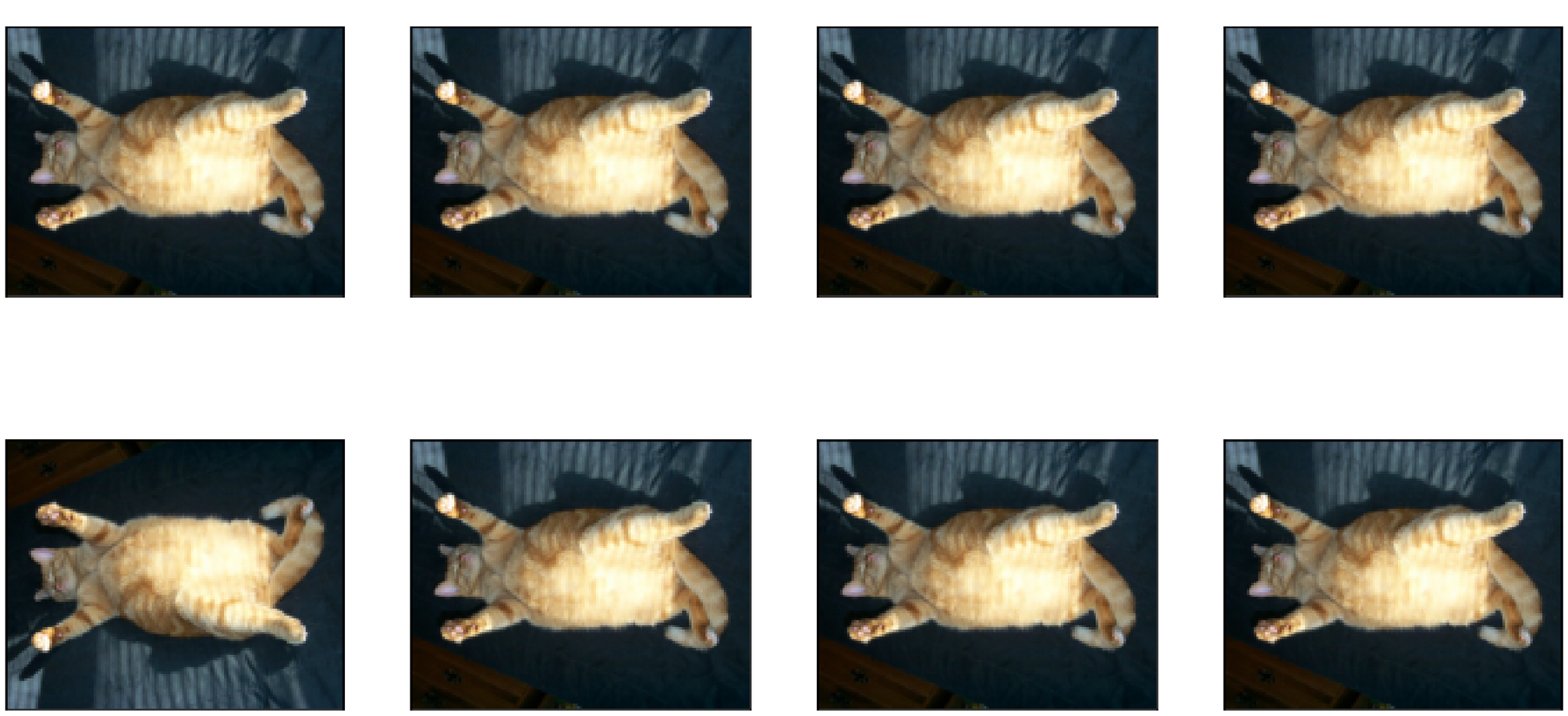
In [4]:
```
apply(img, torchvision.transforms.RandomHorizontalFlip())
```



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，创建一个 RandomFlipTopBottom 实例，使图像各有50%的几率向上或向下翻转。

In [5]:
```
apply(img, torchvision.transforms.RandomVerticalFlip())
```

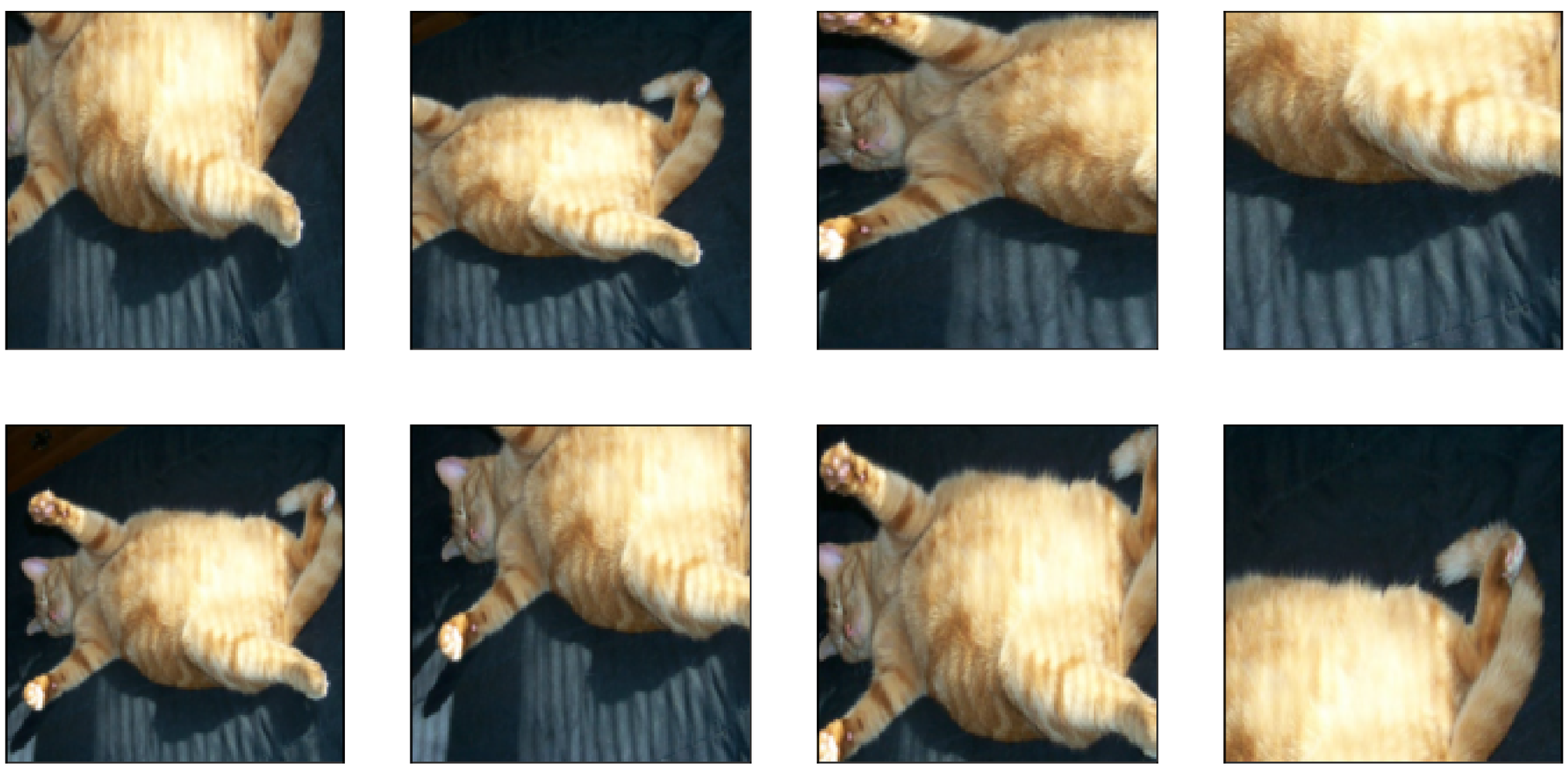


在我们使用的示例图像中，猫位于图像的中间，但并非所有图像都是这样。 在池化的学习中，我们解释了池化层可以降低卷积层对目标位置的敏感性。

另外，也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。

在下面的代码中，**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到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。

In [6]:
```
shape_aug = torchvision.transforms.RandomResizedCrop(
    (200, 200), scale=(0.1, 1), ratio=(0.5, 2))
apply(img, shape_aug)
```

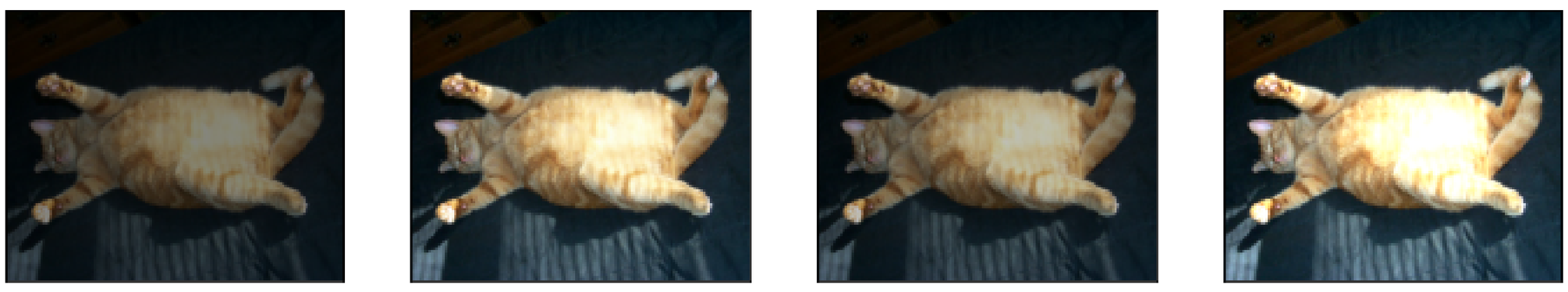


改变颜色

另一种增广方法是改变颜色。 可以改变图像颜色的四个方面：亮度、对比度、饱和度和色调。

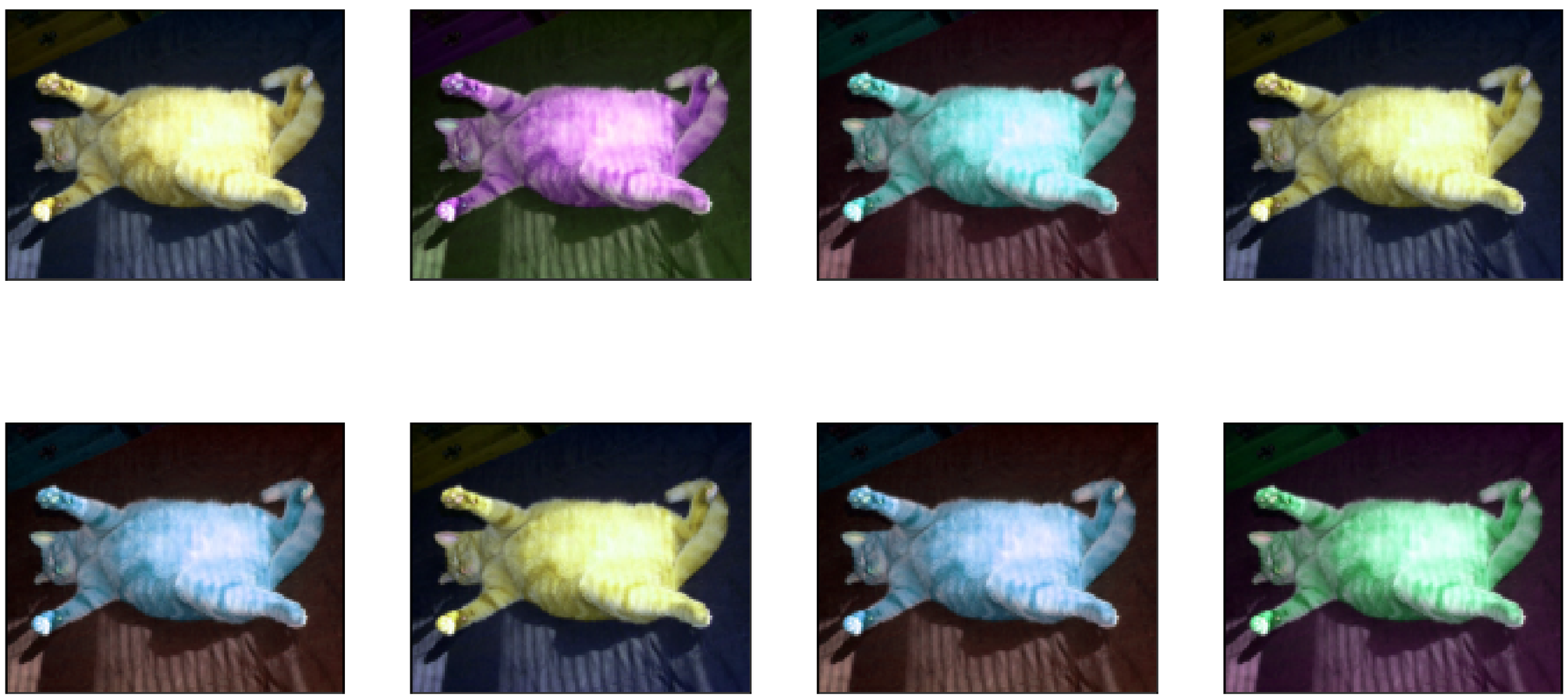
在下面的示例中**随机更改图像的亮度**，随机值为原始图像的50%（$1-0.5$）到150%（$1+0.5$）之间。

In [7]:
```
apply(img, torchvision.transforms.ColorJitter(
    brightness=0.5, contrast=0, saturation=0, hue=0))
```



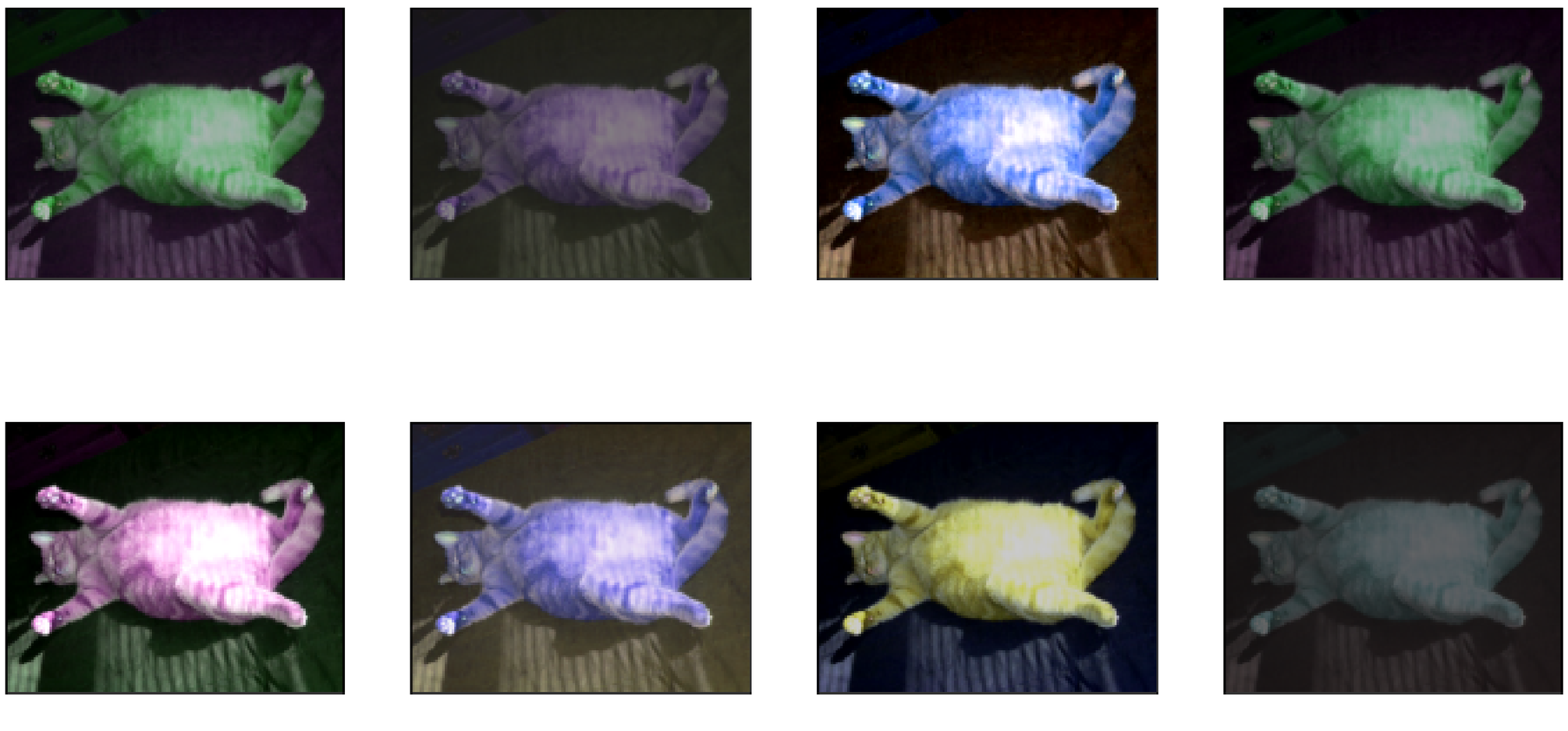
同样，可以**随机更改图像的色调。**

In [8]:
```
apply(img, torchvision.transforms.ColorJitter(
    brightness=0, contrast=0, saturation=0, hue=0.5))
```



可以同时**随机更改图像的亮度（brightness）、对比度（contrast）、饱和度（saturation）和色调（hue）。**

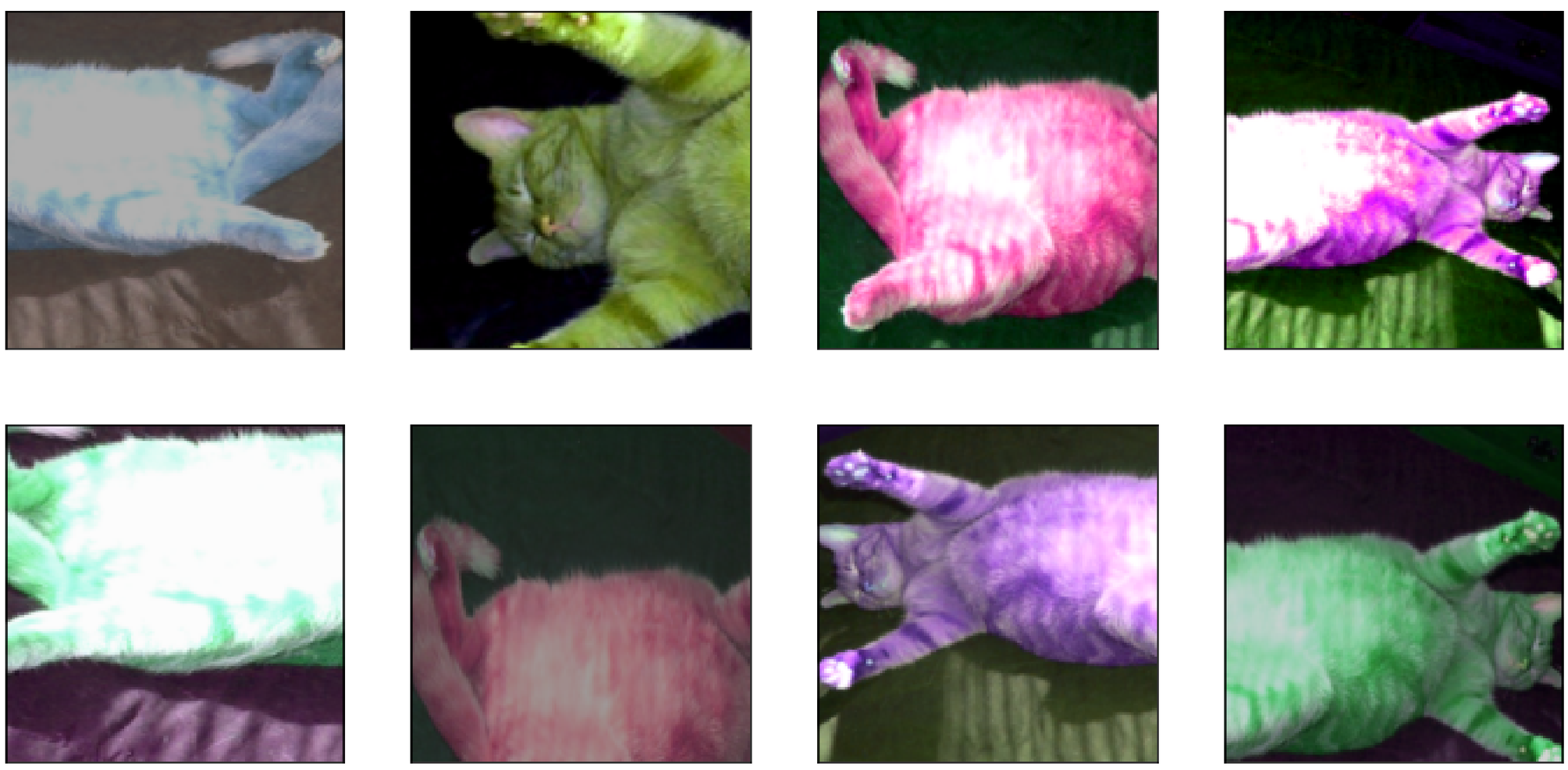
In [9]:
```
color_aug = torchvision.transforms.ColorJitter(
    brightness=0.5, contrast=0.5, saturation=0.5, hue=0.5)
apply(img, color_aug)
```



结合多种图像增广方法

在实践中，一般将结合多种图像增广方法。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。

In [10]:
```
augs = torchvision.transforms.Compose([
    torchvision.transforms.RandomHorizontalFlip(), color_aug, shape_aug])
apply(img, augs)
```



微调

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。

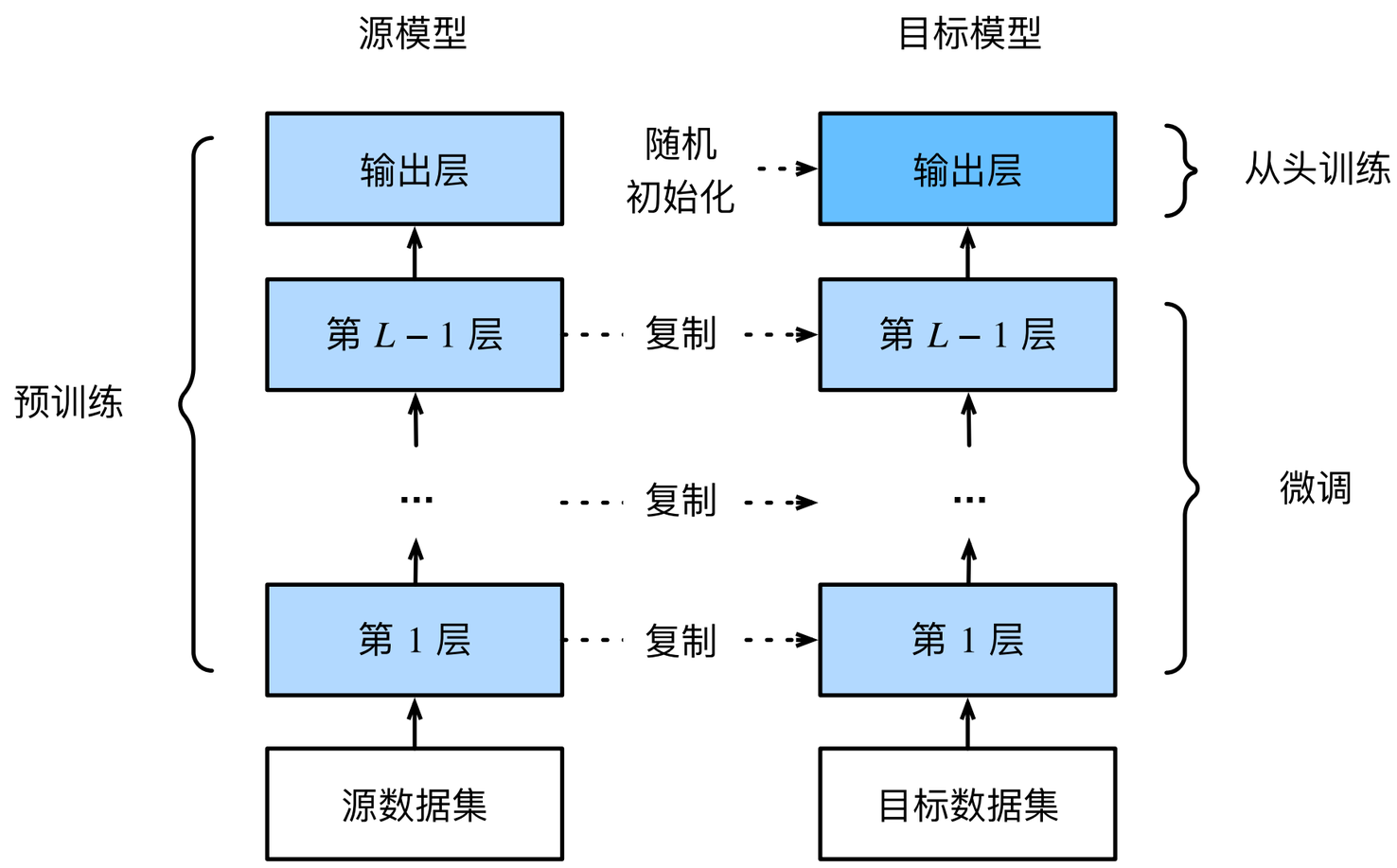
另一种解决方案是应用*迁移学习*（transfer learning）将从*源数据集*学到的知识迁移到*目标数据集*。 例如，尽管ImageNet数据集中的大多数图像与椅子无关，但在此数据集上训练的模型可能会提取更通用的图像特征，这有助于识别边缘、纹理、形状和对象组合。 这些类似的特征也可能有效地识别椅子。

步骤

在本节中，我们将介绍迁移学习中的常见技巧:*微调*（fine-tuning）。如下图所示，微调包括以下四个步骤：

1. 在源数据集（例如ImageNet数据集）上预训练神经网络模型，即*源模型*。
2. 创建一个新的神经网络模型，即*目标模型*。这将复制源模型上的所有模型设计及其参数（输出层除外）。我们假定这些模型参数包含从源数据集中学到的知识，这些知识也将适用于目标数据集。我们还假设源模型的输出层与源数据集的标签密切相关；因此不在目标模型中使用该层。
3. 向目标模型添加输出层，其输出数是目标数据集中的类别数。然后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。输出层将从头开始进行训练，而所有其他层的参数将根据源模型的参数进行微调。



当目标数据集比源数据集小得多时，微调有助于提高模型的泛化能力。

热狗识别

我们将在一个小型数据集——热狗识别，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。

获取数据集

这里使用的**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。

解压下载的数据集，获得了两个文件夹 `hotdog/train` 和 `hotdog/test` 。 这两个文件夹都有 `hotdog`（有热狗）和 `not-hotdog` （无热狗）两个子文件夹， 子文件夹内都包含相应类的图像。

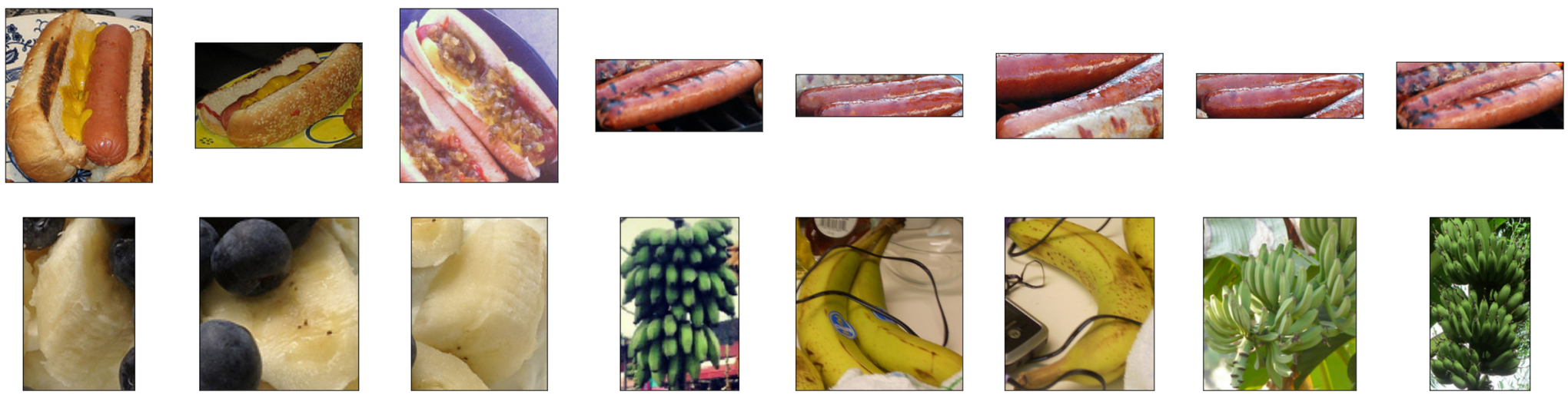
In [11]:
```
d21.DATA_HUB['hotdog'] = (d21.DATA_URL + 'hotdog.zip',
                          'fba480ffa8aa7e0febbb511d181409f899b9baa5')

data_dir = d2l.download_extract('hotdog')
```

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。

In [12]:
```
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，其中**图像的大小和纵横比各有不同。**

In [13]:
```
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=4);
```



在训练期间，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，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，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。

In [14]:
```
# 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),
    torchvision.transforms.CenterCrop(224),
    torchvision.transforms.ToTensor(),
    normalize])
```

定义和初始化模型

使用在ImageNet数据集上预训练的ResNet-18作为源模型。 在这里，指定 `pretrained=True` 以自动下载预训练的模型参数。

如果是首次使用此模型，则需要连接互联网才能下载。

In [15]:
```
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` 。

In [16]:
```
pretrained_net.fc
```

Out[16]:
```
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，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，将成员变量 `output` 中参数的学习率设置为10η。

In [17]:
```
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

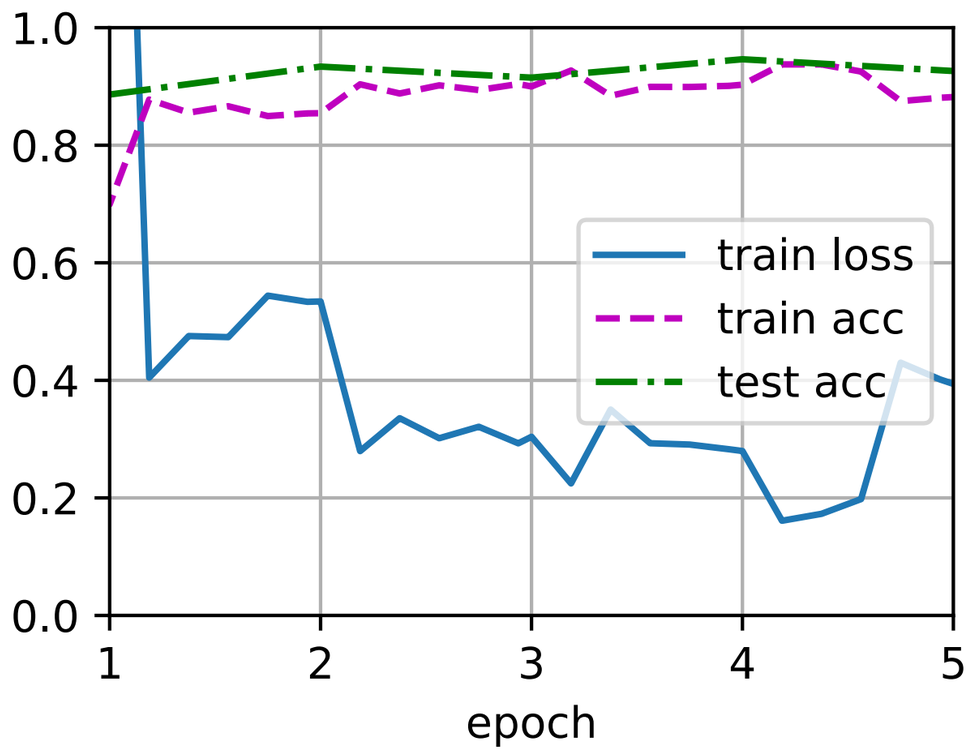
首先，定义一个使用微调的训练函数 `train_fine_tuning` 。

In [18]:
```
# 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)
```

通过*微调*预训练获得的模型参数需要**使用较小的学习率。**

In [19]:
```
1 train_fine_tuning(finetune_net, 5e-5)
```

```
loss 0.394, train acc 0.882, test acc 0.926
472.9 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1), device
(type='cuda', index=2), device(type='cuda', index=3)]
```



下面定义一个相同的模型，但是将其(**所有模型参数初始化为随机值**)来与初始化模型进行比较。

由于整个模型需要从头开始训练，因此需要使用更大的学习率。

In [20]:
```
1 scratch_net = torchvision.models.resnet18()
2 scratch_net.fc = nn.Linear(scratch_net.fc.in_features, 2)
3 train_fine_tuning(scratch_net, 5e-4, param_group=False)
```

```
loss 0.371, train acc 0.832, test acc 0.816
1224.7 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1), device
(type='cuda', index=2), device(type='cuda', index=3)]
```