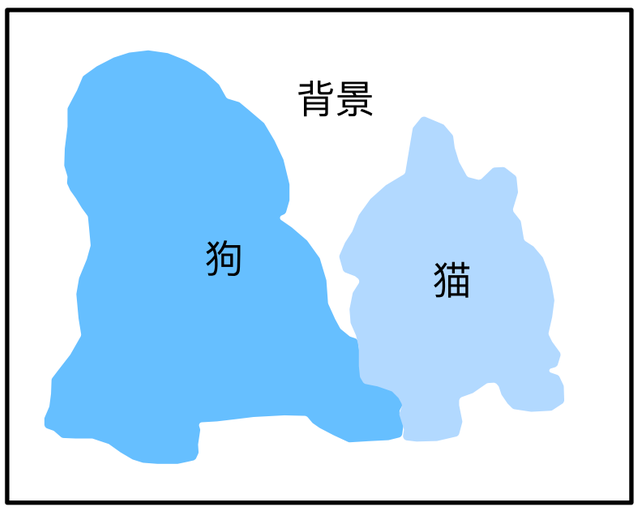
1. 语义分割和数据集

在目标检测问题中，标注和预测图像中的目标使用方形边界框。 本节将探讨*语义分割*（semantic segmentation）问题，它重点关注于如何将图像分割成属于不同语义类别的区域。

与目标检测不同，语义分割可以识别并理解图像中每一个像素的内容：其语义区域的标注和预测是像素级的。 下图展示了语义分割中图像有关狗、猫和背景的标签。 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。



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）。 这里将它们同语义分割简单区分一下。

- *图像分割*将图像划分为若干组成区域，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。以上图中的图像作为输入，图像分割可能会将狗分为两个区域：一个覆盖以黑色为主的嘴和眼睛，另一个覆盖以黄色为主的其余部分身体。
- *实例分割*也叫*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。例如，如果图像中有两条狗，则实例分割需要区分像素属于的两条狗中的哪一条。

Pascal VOC2012 语义分割数据集

最重要的语义分割数据集之一是[Pascal VOC2012 (http://host.robots.ox.ac.uk/pascal/VOC/voc2012/)](http://host.robots.ox.ac.uk/pascal/VOC/voc2012/)。

In [1]:
```
%matplotlib inline
import os
import torch
import torchvision
from torch import nn
from d2l import torch as d2l
from torch.nn import functional as F
```

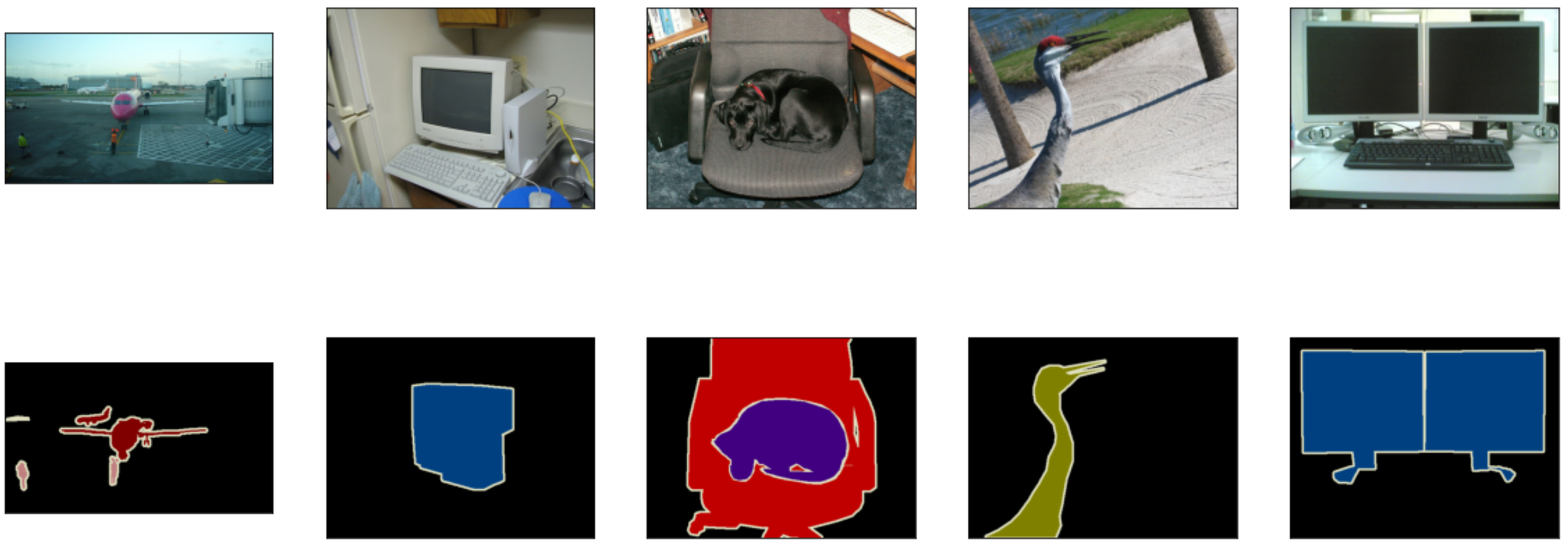
数据集的tar文件大约为2GB，所以下载可能需要一段时间。 提取出的数据集位于 ../data/VOCdevkit/VOC2012 。

In [2]:
```
d2l.DATA_HUB['voc2012'] = (d2l.DATA_URL + 'VOCtrainval_11-May-2012.tar',
                           '4e443f8a2eca6b1dac8a6c57641b67dd40621a49')

voc_dir = d2l.download_extract('voc2012', 'VOCdevkit/VOC2012')
```

In [3]:
```
def read_voc_images(voc_dir, is_train=True):
    """读取所有VOC图像并标注"""
    txt_fname = os.path.join(voc_dir, 'ImageSets', 'Segmentation',
                             'train.txt' if is_train else 'val.txt')
    mode = torchvision.io.image.ImageReadMode.RGB
    with open(txt_fname, 'r') as f:
        images = f.read().split()
    features, labels = [], []
    for i, fname in enumerate(images):
        features.append(torchvision.io.read_image(os.path.join(
            voc_dir, 'JPEGImages', f'{fname}.jpg')))
        labels.append(torchvision.io.read_image(os.path.join(
            voc_dir, 'SegmentationClass' ,f'{fname}.png'), mode))
    return features, labels

train_features, train_labels = read_voc_images(voc_dir, True)
```

下面**绘制前5个输入图像及其标签。** 在标签图像中，白色和黑色分别表示边框和背景，而其他颜色则对应不同的类别。

In [4]:
```
n = 5
imgs = train_features[0:n] + train_labels[0:n]
imgs = [img.permute(1,2,0) for img in imgs]
d2l.show_images(imgs, 2, n, scale = 4);
```



列举RGB颜色值和类名。

In [5]:
```
VOC_COLORMAP = [[0, 0, 0], [128, 0, 0], [0, 128, 0], [128, 128, 0],
                [0, 0, 128], [128, 0, 128], [0, 128, 128], [128, 128, 128],
                [64, 0, 0], [192, 0, 0], [64, 128, 0], [192, 128, 0],
                [64, 0, 128], [192, 0, 128], [64, 128, 128], [192, 128, 128],
                [0, 64, 0], [128, 64, 0], [0, 192, 0], [128, 192, 0],
                [0, 64, 128]]

VOC_CLASSES = ['background', 'aeroplane', 'bicycle', 'bird', 'boat',
               'bottle', 'bus', 'car', 'cat', 'chair', 'cow',
               'diningtable', 'dog', 'horse', 'motorbike', 'person',
               'potted plant', 'sheep', 'sofa', 'train', 'tv/monitor']
```

可以通过上面定义的两个常量，方便地**查找标签中每个像素的类索引。**

定义 voc_colormap2label 函数来构建从上述RGB颜色值到类别索引的映射，而 voc_label_indices 函数将RGB值映射到在Pascal VOC2012数据集中的类别索引。

In [6]:
```
def voc_colormap2label():
    """构建从RGB到VOC类别索引的映射"""
    colormap2label = torch.zeros(256 ** 3, dtype=torch.long) # 所有颜色的空间
    for i, colormap in enumerate(VOC_COLORMAP):  # 对 VOC_COLORMAP里所有颜色 在颜色空间
        colormap2label[
            (colormap[0] * 256 + colormap[1]) * 256 + colormap[2]] = i
    return colormap2label

def voc_label_indices(colormap, colormap2label):
    """将VOC标签中的RGB值映射到它们的类别索引"""
    colormap = colormap.permute(1, 2, 0).numpy().astype('int32')
    idx = ((colormap[:, :, 0] * 256 + colormap[:, :, 1]) * 256 + colormap[:, :, 2])
    return colormap2label[idx]
```

例如，在第一张样本图像中，飞机头部区域的类别索引为1，而背景索引为0。

In [7]:
```
y = voc_label_indices(train_labels[0], voc_colormap2label())
y[105:115, 130:140], VOC_CLASSES[1]
```

Out[7]:
```
(tensor([[0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
         [0, 0, 0, 0, 0, 0, 0, 1, 1, 1],
         [0, 0, 0, 0, 0, 0, 1, 1, 1, 1],
         [0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
         [0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
         [0, 0, 0, 0, 1, 1, 1, 1, 1, 1],
         [0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
         [0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
         [0, 0, 0, 0, 0, 0, 1, 1, 1, 1],
         [0, 0, 0, 0, 0, 0, 0, 0, 1, 1]]),
 'aeroplane')
```

预处理数据

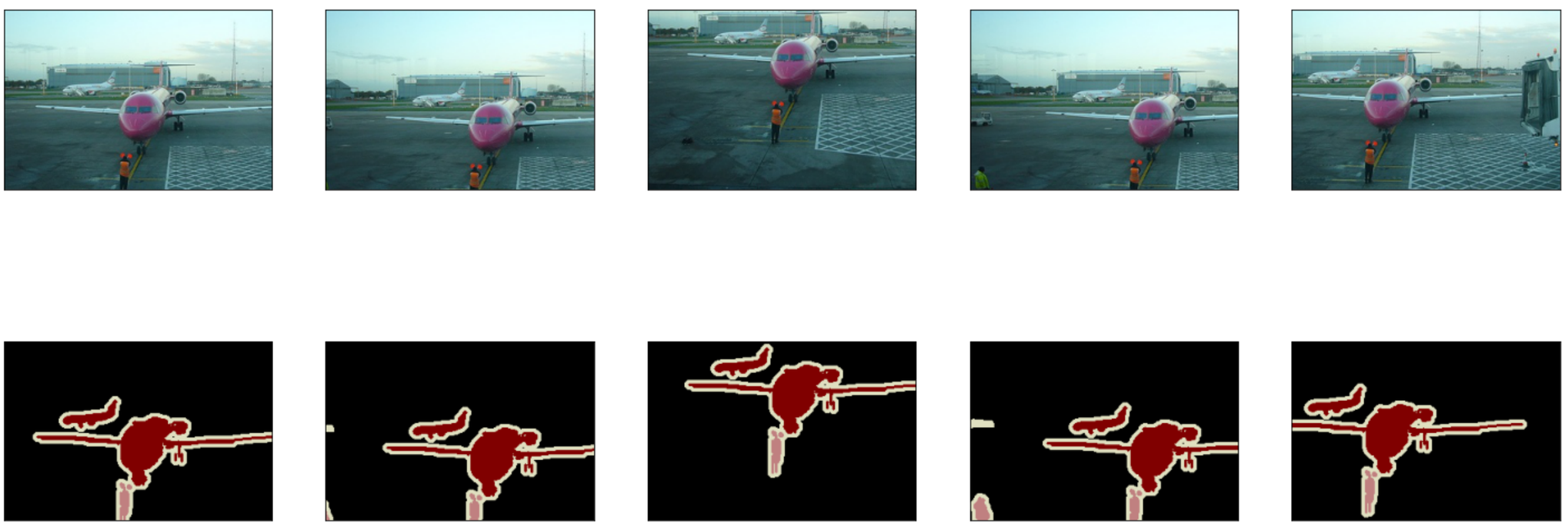
在之前的 CNN 实验通过再缩放图像使其符合模型的输入形状。

然而在语义分割中，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。 这样的映射可能不够精确，尤其在不同语义的分割区域。

为了避免这个问题，将图像裁剪为固定尺寸，而不是再缩放。 具体来说，**使用图像增广中的随机裁剪，裁剪输入图像和标签的相同区域。**

In [8]:
```
def voc_rand_crop(feature, label, height, width):
    """随机裁剪特征和标签图像"""
    rect = torchvision.transforms.RandomCrop.get_params(
        feature, (height, width))
    feature = torchvision.transforms.functional.crop(feature, *rect)
    label = torchvision.transforms.functional.crop(label, *rect)
    return feature, label
```

In [9]:
```
imgs = []
for _ in range(n):
    imgs += voc_rand_crop(train_features[0], train_labels[0], 200, 300)

imgs = [img.permute(1, 2, 0) for img in imgs]
d2l.show_images(imgs[::2] + imgs[1::2], 2, n, scale=5);
```



自定义语义分割数据集类

通过继承高级API提供的 `Dataset` 类，自定义一个语义分割数据集类 `VOCSegDataset` 。

通过实现 `__getitem__` 函数，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，这些样本可以通过自定义的 `filter` 函数移除掉。

此外，定义了 `normalize_image` 函数，从而对输入图像的RGB三个通道的值分别做标准化。

In [10]:
```python
class VOCSegDataset(torch.utils.data.Dataset):
    """一个用于加载VOC数据集的自定义数据集"""

    def __init__(self, is_train, crop_size, voc_dir):
        self.transform = torchvision.transforms.Normalize(
            mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
        self.crop_size = crop_size
        features, labels = read_voc_images(voc_dir, is_train=is_train)
        self.features = [self.normalize_image(feature)
                         for feature in self.filter(features)]
        self.labels = self.filter(labels)
        self.colormap2label = voc_colormap2label()
        print('read ' + str(len(self.features)) + ' examples')

    def normalize_image(self, img):
        return self.transform(img.float() / 255)

    def filter(self, imgs):
        return [img for img in imgs if (
            img.shape[1] >= self.crop_size[0] and
            img.shape[2] >= self.crop_size[1])]

    def __getitem__(self, idx):
        feature, label = voc_rand_crop(self.features[idx], self.labels[idx],
                                       *self.crop_size)
        return (feature, voc_label_indices(label, self.colormap2label))

    def __len__(self):
        return len(self.features)
```

读取数据集

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320×480， 下面可以查看训练集和测试集所保留的样本个数。

In [11]:
```python
crop_size = (320, 480)
voc_train = VOCSegDataset(True, crop_size, voc_dir)
voc_test = VOCSegDataset(False, crop_size, voc_dir)
```

```
read 1114 examples
read 1078 examples
```

设批量大小为64，定义训练集的迭代器。

打印第一个小批量的形状会发现：与图像分类或目标检测不同，这里的标签是一个三维数组。

In [12]:
```python
batch_size = 64
train_iter = torch.utils.data.DataLoader(voc_train, batch_size, shuffle=True,
                                         drop_last=True,
                                         num_workers=d2l.get_dataloader_workers())
for X, Y in train_iter:
    print(X.shape)
    print(Y.shape)
    break
```

```
torch.Size([64, 3, 320, 480])
torch.Size([64, 320, 480])
```

In [13]:
```python
def load_data_voc(batch_size, crop_size):
    """加载VOC语义分割数据集"""
    voc_dir = d2l.download_extract('voc2012', os.path.join(
        'VOCdevkit', 'VOC2012'))
    num_workers = d2l.get_dataloader_workers()
    train_iter = torch.utils.data.DataLoader(
        VOCSegDataset(True, crop_size, voc_dir), batch_size,
        shuffle=True, drop_last=True, num_workers=num_workers)
    test_iter = torch.utils.data.DataLoader(
        VOCSegDataset(False, crop_size, voc_dir), batch_size,
        drop_last=True, num_workers=num_workers)
    return train_iter, test_iter
```

2. 转置卷积（反卷积）

到目前为止所见到的卷积神经网络层，例如卷积层和汇聚层，通常会减少下采样输入图像的空间维度（高和宽）。

然而如果输入和输出图像的空间维度相同，在以像素级分类的语义分割中将会很方便。 例如，输出像素所处的通道维可以保有输入像素在同一位置上的分类结果。

为了实现这一点，尤其是在空间维度被卷积神经网络层缩小后，我们可以使用另一种类型的卷积神经网络层，它可以增加上采样中间层特征图的空间维度。

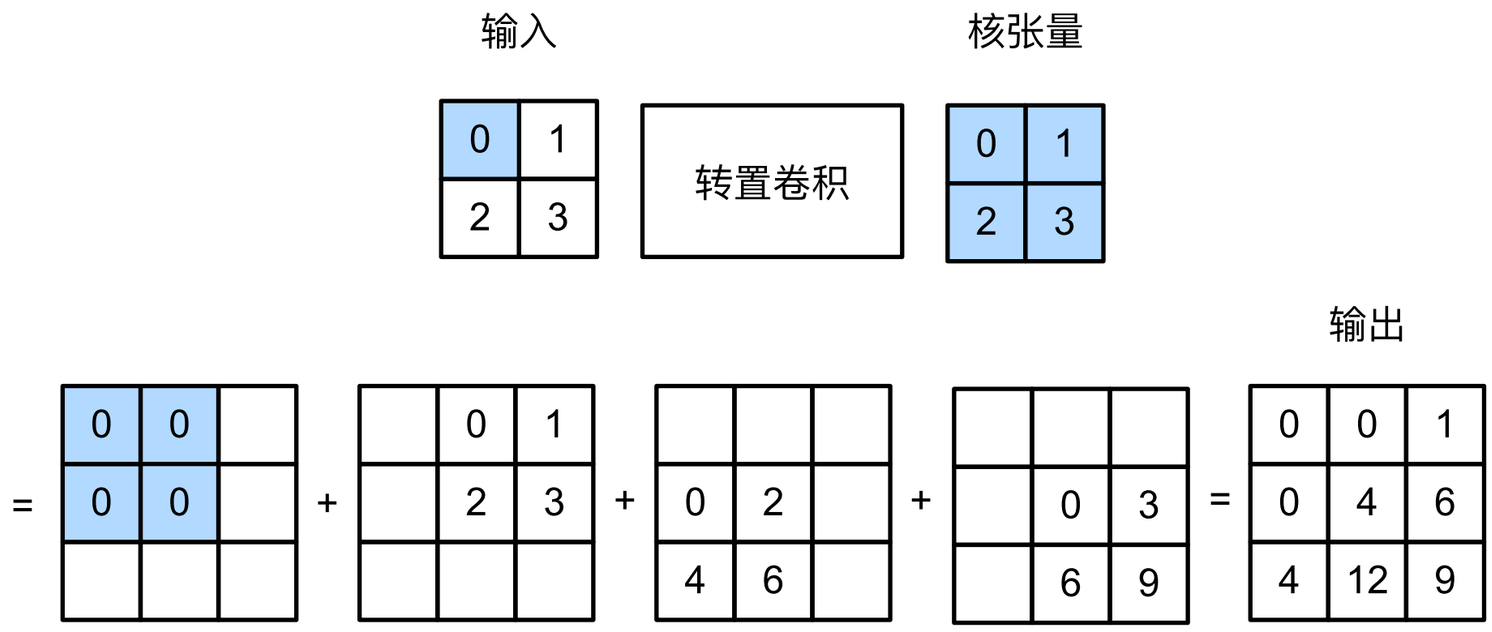
在本节中将介绍 *转置卷积*（transposed convolution）， 用于逆转下采样导致的空间尺寸减小。

基本操作

暂时忽略通道，从基本的转置卷积开始，设步幅为1且没有填充。

给定一个$n_h \times n_w$的输入张量和一个$k_h \times k_w$的卷积核。 以步幅为1滑动卷积核窗口，每行n_w次，每列n_h次，共产生$n_h n_w$个中间结果。 每个中间结果都是一个$(n_h + k_h - 1) \times (n_w + k_w - 1)$的张量，初始化为0。 为了计算每个中间张量，输入张量中的每个元素都要乘以卷积核，从而使所得的$k_h \times k_w$张量替换中间张量的一部分。 请注意，每个中间张量被替换部分的位置与输入张量中元素的位置相对应。 最后，所有中间结果相加以获得最终结果。

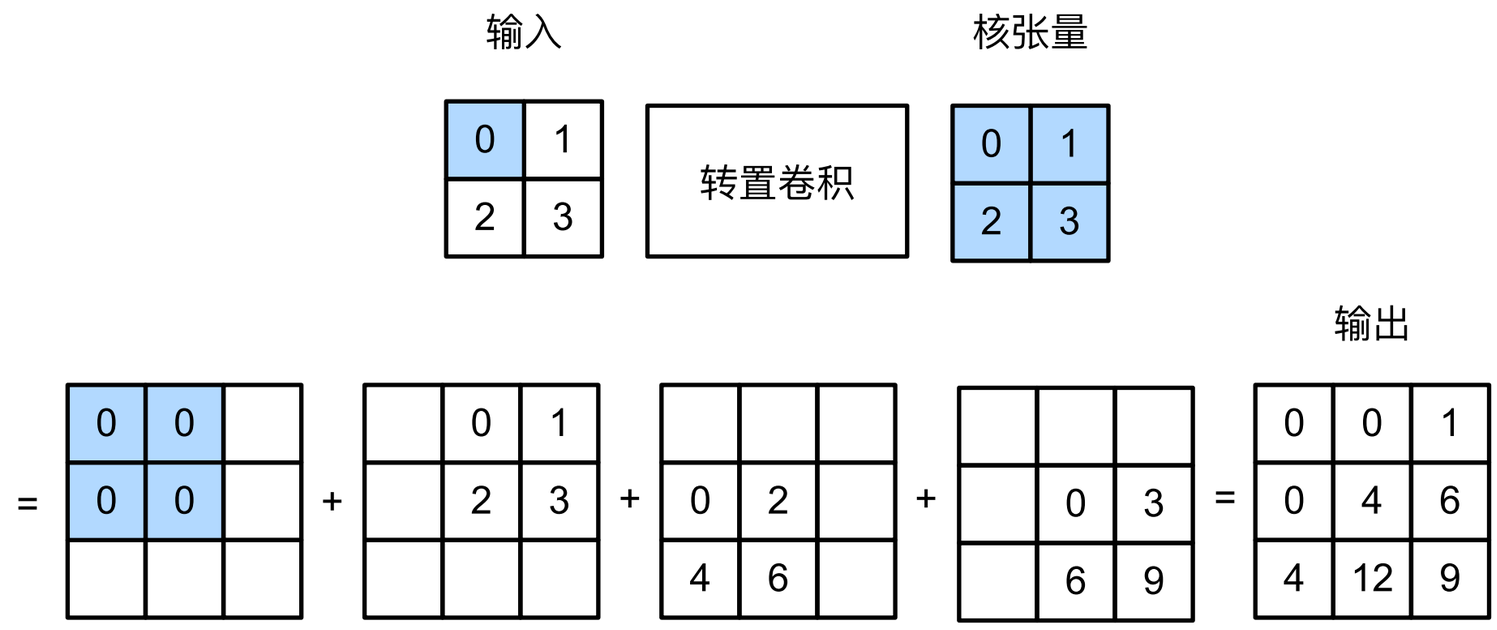
下图解释了如何为2×2的输入张量计算卷积核为2×2的转置卷积。



可以对输入矩阵 `X` 和卷积核矩阵 `K` **实现基本的转置卷积运算** `trans_conv` 。

In [14]:
```
def trans_conv(X, K):
    h, w = K.shape
    Y = torch.zeros((X.shape[0] + h - 1, X.shape[1] + w - 1))
    for i in range(X.shape[0]):
        for j in range(X.shape[1]):
            Y[i: i + h, j: j + w] += X[i, j] * K
    return Y
```

与通过卷积核“减少”输入元素的常规卷积相比，转置卷积通过卷积核“广播”输入元素，从而产生大于输入的输出。



已上图例子构建输入张量 X 和卷积核张量 K 从而**验证上述实现输出。** 此实现是基本的二维转置卷积运算。

In [15]:
```
X = torch.tensor([[0.0, 1.0], [2.0, 3.0]])
K = torch.tensor([[0.0, 1.0], [2.0, 3.0]])
trans_conv(X, K)
```

Out[15]:
```
tensor([[ 0.,  0.,  1.],
        [ 0.,  4.,  6.],
        [ 4., 12.,  9.]])
```

或者，当输入 X 和卷积核 K 都是四维张量时，可以**使用高级API获得相同的结果。**

In [16]:
```
X, K = X.reshape(1, 1, 2, 2), K.reshape(1, 1, 2, 2)
tconv = nn.ConvTranspose2d(1, 1, kernel_size=2, bias=False)
tconv.weight.data = K
tconv(X)
```

Out[16]:
```
tensor([[[[ 0.,  0.,  1.],
          [ 0.,  4.,  6.],
          [ 4., 12.,  9.]]]], grad_fn=<SlowConvTranspose2DBackward0>)
```

填充、步幅和多通道

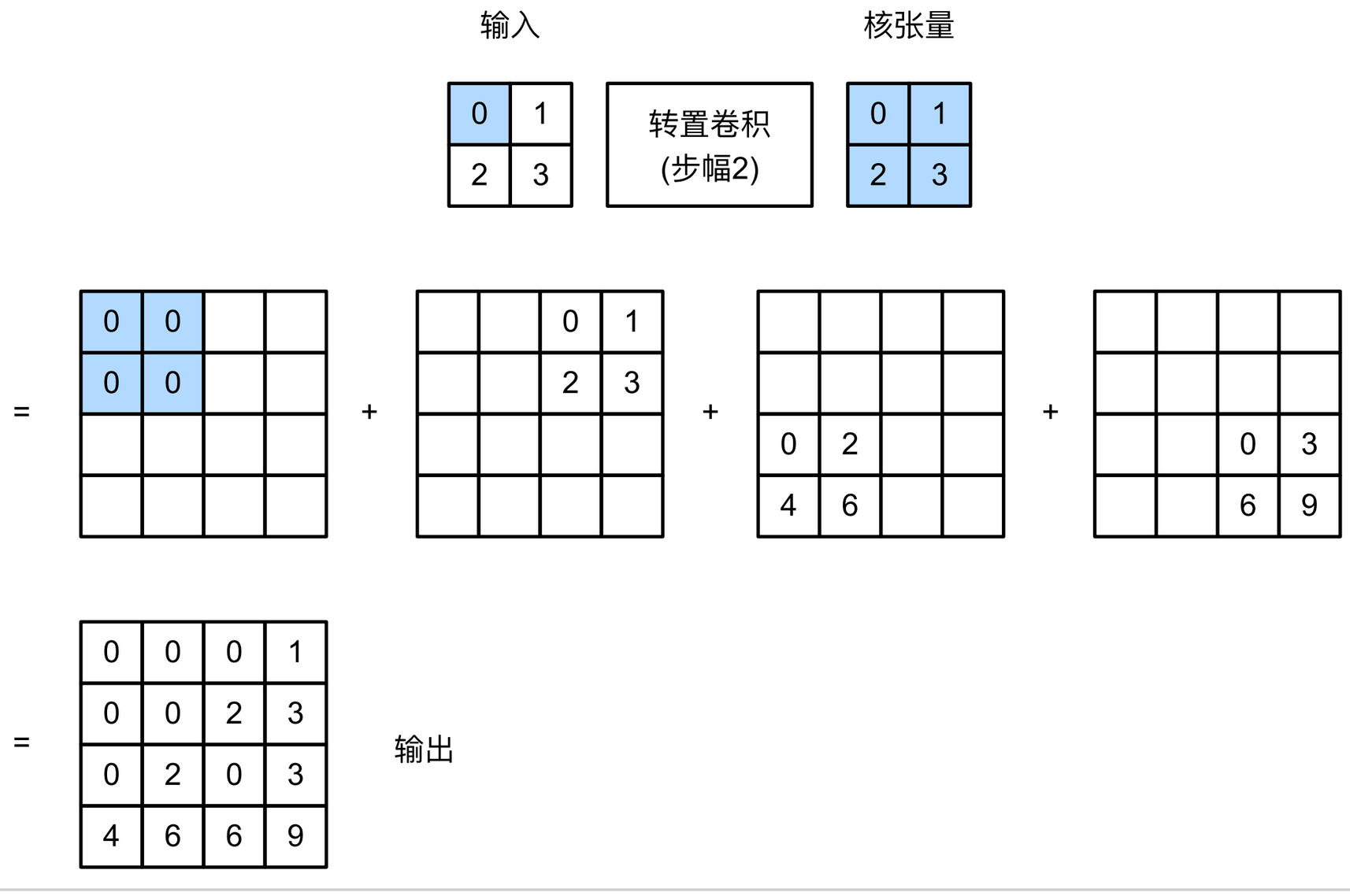
与常规卷积不同，在转置卷积中，填充被应用于输出（常规卷积将填充应用于输入）。 例如，当将高和宽两侧的填充数指定为1时，转置卷积的输出中将删除第一和最后的行与列。

In [17]:
```
tconv = nn.ConvTranspose2d(1, 1, kernel_size=2, padding=1, bias=False)
tconv.weight.data = K
tconv(X)
```

Out[17]:
```
tensor([[[[4.]]]], grad_fn=<SlowConvTranspose2DBackward0>)
```

在转置卷积中，步幅被指定为中间结果（输出），而不是输入。 使用与之前例子中相同输入和卷积核张

量，将步幅从1更改为2会增加中间张量的高和权重，输出张量在下图中。



以下代码可以验证该例子中步幅为2的转置卷积的输出。

In [18]:
```
tconv = nn.ConvTranspose2d(1, 1, kernel_size=2, stride=2, bias=False)
tconv.weight.data = K
tconv(X)
```

Out[18]:
```
tensor([[[[0., 0., 0., 1.],
          [0., 0., 2., 3.],
          [0., 2., 0., 3.],
          [4., 6., 6., 9.]]]], grad_fn=<SlowConvTranspose2DBackward0>)
```

对于多个输入和输出通道，转置卷积与常规卷积以相同方式运作。

假设输入有c_i个通道，且转置卷积为每个输入通道分配了一个$k_h \times k_w$的卷积核张量。 当指定多个输出通道时，每个输出通道将有一个$c_i \times k_h \times k_w$的卷积核。

同样，如果将X代入卷积层f来输出$Y = f(X)$，并创建一个与f具有相同的超参数、但输出通道数量是X中通道数的转置卷积层g，那么$g(Y)$的形状将与X相同。 下面的示例可以解释这一点。

In [19]:
```
X = torch.rand(size=(1, 10, 16, 16))
conv = nn.Conv2d(10, 20, kernel_size=5, padding=2, stride=3)
tconv = nn.ConvTranspose2d(20, 10, kernel_size=5, padding=2, stride=3)
tconv(conv(X)).shape == X.shape
```

Out[19]:
```
True
```

与矩阵变换的联系

转置卷积为何以矩阵变换命名呢？ 首先看看如何使用矩阵乘法来实现卷积。 在下面的示例中定义了一个3×3的输入 `X` 和2×2卷积核 `K` ， 然后使用 `corr2d` 函数计算卷积输出 `Y` 。

In [20]:
```
X = torch.arange(9.0).reshape(3, 3)
K = torch.tensor([[1.0, 2.0], [3.0, 4.0]])
Y = d2l.corr2d(X, K)
Y
```

Out[20]:
```
tensor([[27., 37.],
        [57., 67.]])
```

接下来将卷积核 K 重写为包含大量0的稀疏权重矩阵 W 。 权重矩阵的形状是（4，9），其中非0元素来自卷积核 K 。

In [21]:
```
def kernel2matrix(K):
    k, W = torch.zeros(5), torch.zeros((4, 9))
    k[:2], k[3:5] = K[0, :], K[1, :]
    W[0, :5], W[1, 1:6], W[2, 3:8], W[3, 4:] = k, k, k, k
    return W

W = kernel2matrix(K)
W
```

Out[21]:
```
tensor([[1., 2., 0., 3., 4., 0., 0., 0., 0.],
        [0., 1., 2., 0., 3., 4., 0., 0., 0.],
        [0., 0., 0., 1., 2., 0., 3., 4., 0.],
        [0., 0., 0., 0., 1., 2., 0., 3., 4.]])
```

逐行连结输入 X ，获得了一个长度为9的矢量。

然后， W 的矩阵乘法和向量化的 X 给出了一个长度为4的向量。

重塑它之后，可以获得与上面的原始卷积操作所得相同的结果 Y ：我们刚刚使用矩阵乘法实现了卷积。

In [22]:
```
Y == torch.matmul(W, X.reshape(-1)).reshape(2, 2)
```

Out[22]:
```
tensor([[True, True],
        [True, True]])
```

同样，可以使用矩阵乘法来实现转置卷积。

在下面的示例中，将上面的常规卷积2 × 2的输出 Y 作为转置卷积的输入。

想要通过矩阵相乘来实现它，只需要将权重矩阵 W 的形状转置为$(9,4)$。

In [23]:
```
Z = trans_conv(Y, K)
Z == torch.matmul(W.T, Y.reshape(-1)).reshape(3, 3)
```

Out[23]:
```
tensor([[True, True, True],
        [True, True, True],
        [True, True, True]])
```

抽象来看，给定输入向量$\mathbf{x}$和权重矩阵$\mathbf{W}$，卷积的前向传播函数可以通过将其输入与权重矩阵相乘并输出向量$\mathbf{y} = \mathbf{W}\mathbf{x}$来实现。 由于反向传播遵循链式法则和$\nabla_{\mathbf{x}}\mathbf{y} = \mathbf{W}^\top$，卷积的反向传播函数可以通过将其输入与转置的权重矩阵$\mathbf{W}^\top$相乘来实现。 因此，转置卷积层能够交换卷积层的正向传播函数和反向传播函数：它的正向传播和反向传播函数将输入向量分别与$\mathbf{W}^\top$和$\mathbf{W}$相乘。

3. 全卷积网络

语义分割是对图像中的每个像素分类。

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。

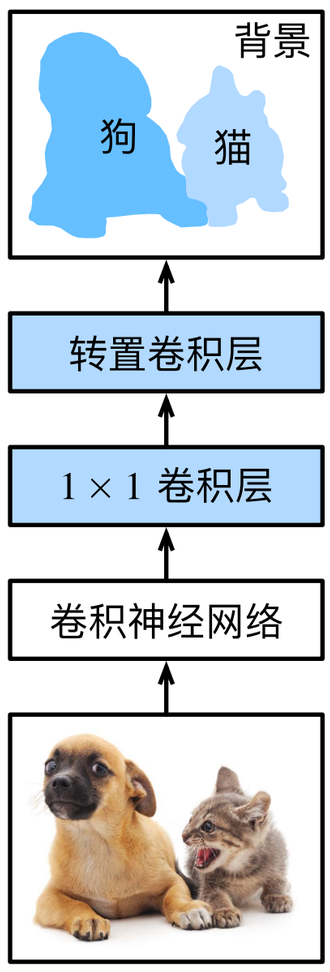
全卷积网络通过*转置卷积*（transposed convolution）将中间层特征图的高和宽变换回输入图像的尺寸。

因此，输出的类别预测与输入图像在像素级别上具有一一对应关系：通道维的输出即该位置对应像素的类别预测。

构造模型

下面介绍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1×1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。 因此，模型输出与输入图像的高和宽相同，且最终输出通道包含了该空间位置像素的类别预测。



下面，**使用在ImageNet数据集上预训练的ResNet-18模型来提取图像特征**，并将该网络记为`pretrained_net`。 ResNet-18模型的最后几层包括全局平均汇聚层和全连接层，然而全卷积网络中不需要它们。

In [24]:
```
pretrained_net = torchvision.models.resnet18(pretrained=True)
list(pretrained_net.children())[-3:]
```

Out[24]:
```
[Sequential(
   (0): BasicBlock(
     (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=Fal
se)
     (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
e)
     (relu): ReLU(inplace=True)
     (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fal
se)
     (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
e)
     (downsample): Sequential(
       (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
       (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
e)
     )
   )
   (1): BasicBlock(
     (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fal
se)
     (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
e)
     (relu): ReLU(inplace=True)
     (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fal
se)
     (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
e)
   )
 ),
 AdaptiveAvgPool2d(output_size=(1, 1)),
 Linear(in_features=512, out_features=1000, bias=True)]
```

接下来，**创建一个全卷积网络** `net` 。 它复制了ResNet-18中大部分的预训练层，除了最后的全局平均汇聚层和最接近输出的全连接层。

In [25]:
```
net = nn.Sequential(*list(pretrained_net.children())[:-2])
```

给定高度为320和宽度为480的输入， `net` 的前向传播将输入的高和宽减小至原来的1/32，即10和15。

In [26]:
```
X = torch.rand(size=(1, 3, 320, 480))
net(X).shape
```

Out[26]:
```
torch.Size([1, 512, 10, 15])
```

接下来，**使用1×1卷积层将输出通道数转换为Pascal VOC2012数据集的类数（21类）。**

最后，需要(**将特征图的高度和宽度增加32倍**)，从而将其变回输入图像的高和宽。

根据卷积层输出形状的计算方法： 由于 $(320-64+16\times 2+32)\times \frac{1}{32}=10$ 且 $(480-64+16\times 2+32)\times \frac{1}{32}=15$ ，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，填充为16。

可以看到如果步幅为s，填充为$s/2$（假设$s/2$是整数）且卷积核的高和宽为$2s$，转置卷积核会将输入的高和宽分别放大s倍。

In [27]:
```
num_classes = 21
net.add_module('final_conv', nn.Conv2d(512, num_classes, kernel_size=1))
net.add_module('transpose_conv', nn.ConvTranspose2d(num_classes, num_classes,
                                    kernel_size=64, padding=16, stride=32))
```

初始化转置卷积层

在图像处理中有时需要将图像放大，即*上采样*（upsampling）。 *双线性插值*（bilinear interpolation） 是常用的上采样方法之一，它也经常用于初始化转置卷积层。

双线性插值：假设给定输入图像，要计算上采样输出图像上的每个像素。

1. 将输出图像的坐标(x, y)映射到输入图像的坐标(x', y')上。 例如，根据输入与输出的尺寸之比来映射。 请注意，映射后的x'和y'是实数。
2. 在输入图像上找到离坐标(x', y')最近的4个像素。
3. 输出图像在坐标(x, y)上的像素依据输入图像上这4个像素及其与(x', y')的相对距离来计算。

双线性插值的上采样可以通过转置卷积层实现，内核由以下 `bilinear_kernel` 函数构造。 限于篇幅，只给出 `bilinear_kernel` 函数的实现，不讨论算法的原理。

In [28]:
```
def bilinear_kernel(in_channels, out_channels, kernel_size):
    factor = (kernel_size + 1) // 2
    if kernel_size % 2 == 1:
        center = factor - 1
    else:
        center = factor - 0.5
    og = (torch.arange(kernel_size).reshape(-1, 1),
          torch.arange(kernel_size).reshape(1, -1))
    filt = (1 - torch.abs(og[0] - center) / factor) * \
           (1 - torch.abs(og[1] - center) / factor)
    weight = torch.zeros((in_channels, out_channels,
                          kernel_size, kernel_size))
    weight[range(in_channels), range(out_channels), :, :] = filt
    return weight
```

用**双线性插值的上采样实验**，它由转置卷积层实现。

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。

In [29]:
```
conv_trans = nn.ConvTranspose2d(3, 3, kernel_size=4, padding=1, stride=2,
                                bias=False)
conv_trans.weight.data.copy_(bilinear_kernel(3, 3, 4));
```

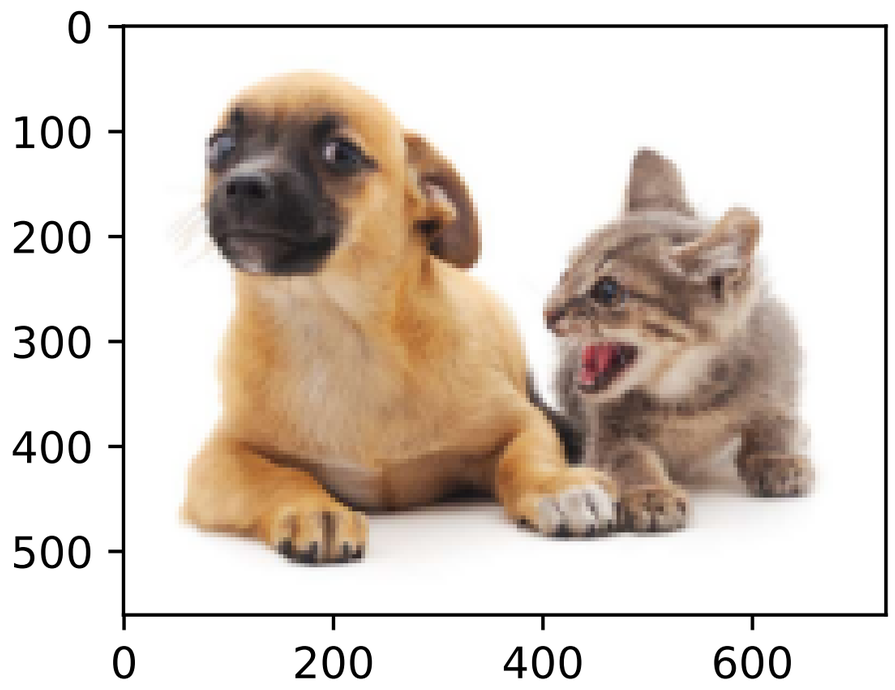
读取图像 `X` ，将上采样的结果记作 `Y` 。为了打印图像，需要调整通道维的位置。

In [30]:
```
img = torchvision.transforms.ToTensor()(d2l.Image.open('figs/catdog.jpg'))
X = img.unsqueeze(0)
Y = conv_trans(X)
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了2倍。 除了坐标刻度不同，双线性插值放大的图像和原图看上去没什么两样。

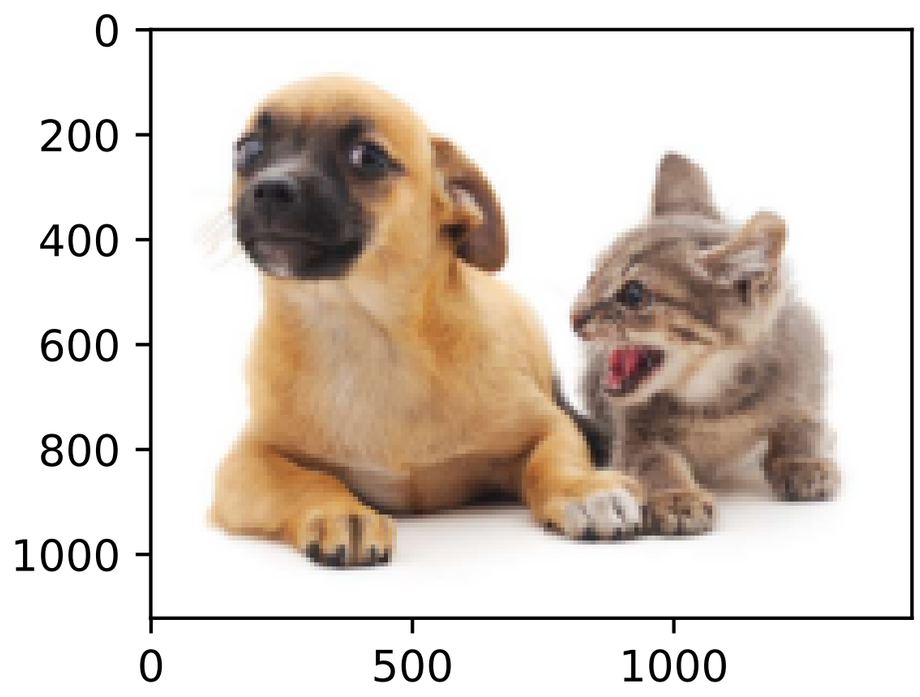
In [31]:
```
d2l.set_figsize()
print('input image shape:', img.permute(1, 2, 0).shape)
d2l.plt.imshow(img.permute(1, 2, 0));
```

```
input image shape: torch.Size([561, 728, 3])
```



In [32]:
```
print('output image shape:', out_img.shape)
d2l.plt.imshow(out_img);
```

```
output image shape: torch.Size([1122, 1456, 3])
```



在全卷积网络中，**用双线性插值的上采样初始化转置卷积层。对于1×1卷积层，使用Xavier初始化参数。**

In [33]:
```
W = bilinear_kernel(num_classes, num_classes, 64)
net.transpose_conv.weight.data.copy_(W);
```

读取数据集

用之前介绍的语义分割读取数据集。 指定随机裁剪的输出图像的形状为320×480：高和宽都可以被32整除。

In [34]:
```
batch_size, crop_size = 32, (320, 480)
train_iter, test_iter = d2l.load_data_voc(batch_size, crop_size)
```

```
read 1114 examples
read 1078 examples
```

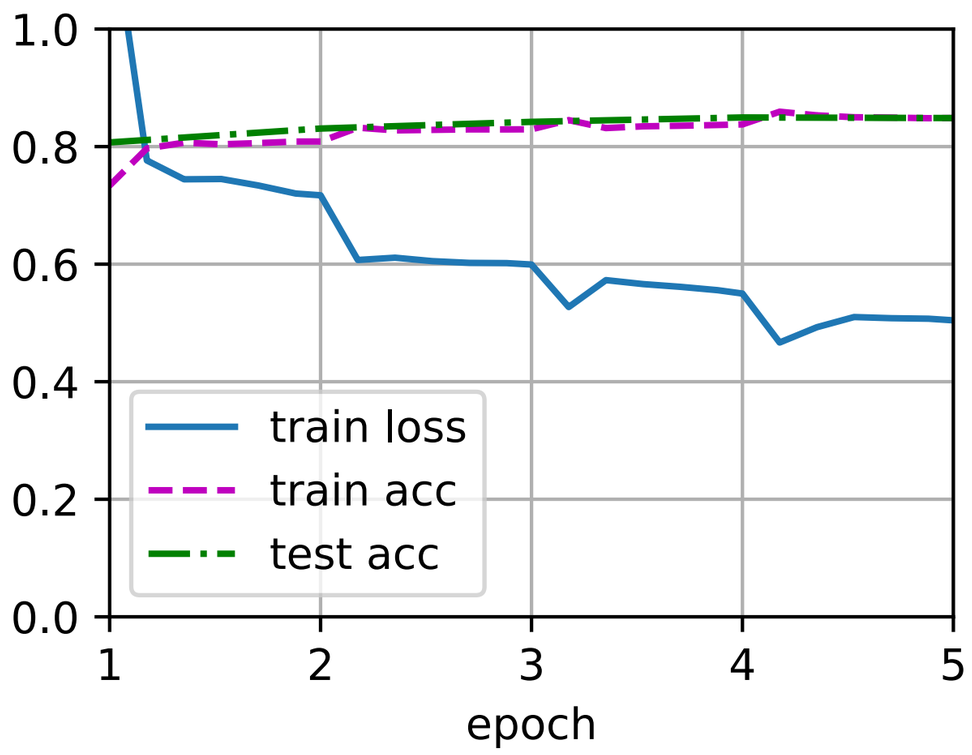
训练

现在可以训练全卷积网络了。 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，因为使用转置卷积层的通道来预测像素的类别，所以需要在损失计算中指定通道维。

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。

In [35]:
```
def loss(inputs, targets):
    return F.cross_entropy(inputs, targets, reduction='none').mean(1).mean(1)

num_epochs, lr, wd, devices = 5, 0.001, 1e-3, d2l.try_all_gpus()
trainer = torch.optim.SGD(net.parameters(), lr=lr, weight_decay=wd)
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices)
```

```
loss 0.504, train acc 0.849, test acc 0.849
25.9 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1), device(t
ype='cuda', index=2), device(type='cuda', index=3)]
```



预测

在预测时，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。

In [36]:
```
def predict(img):
    X = test_iter.dataset.normalize_image(img).unsqueeze(0)
    pred = net(X.to(devices[0])).argmax(dim=1)
    return pred.reshape(pred.shape[1], pred.shape[2])
```

为了给每个像素**可视化预测的类别**，我们将预测类别映射回它们在数据集中的标注颜色。

In [37]:
```
def label2image(pred):
    colormap = torch.tensor(d2l.VOC_COLORMAP, device=devices[0])
    X = pred.long()
    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，因此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，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传播。 请注意，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，它在不同区域前向传播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。

为简单起见，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320×480的区域用于预测。

对于这些测试图像逐一打印它们截取的区域，再打印预测结果，最后打印标注的类别。

In [38]:

```
voc_dir = d2l.download_extract('voc2012', 'VOCdevkit/VOC2012')
test_images, test_labels = d2l.read_voc_images(voc_dir, False)
n, imgs = 4, []
for i in range(n):
    crop_rect = (0, 0, 320, 480)
    X = torchvision.transforms.functional.crop(test_images[i], *crop_rect)
    pred = label2image(predict(X))
    imgs += [X.permute(1,2,0), pred.cpu(),
             torchvision.transforms.functional.crop(
                 test_labels[i], *crop_rect).permute(1,2,0)]
d2l.show_images(imgs[::3] + imgs[1::3] + imgs[2::3], 3, n, scale=2);
```