目标检测

1. 目标检测数据集

目标检测领域没有像 MNIST 和 Fashion-MNIST 那样的小数据集。

为了快速测试目标检测模型，D2L 收集并标记了一个小型数据集。

首先，拍摄一组香蕉的照片，并生成1000张不同角度和大小的香蕉图像。然后，在一些背景图片的随机位置上放一张香蕉的图像。最后，在图片上为这些香蕉标记了边界框。

下载数据集

包含所有图像和CSV标签文件的香蕉检测数据集可以直接从互联网下载。

In [1]:
```
%matplotlib inline
import os
import torch
import torchvision
import pandas as pd
from torch import nn
from d2l import torch as d2l
from torch.nn import functional as F
import warnings
warnings.filterwarnings("ignore")
```

In [2]:
```
d2l.DATA_HUB['banana-detection'] = (
    d2l.DATA_URL + 'banana-detection.zip',
    '5de26c8fce5ccdea9f91267273464dc968d20d72')
```

读取数据集

通过`read_data_bananas`函数，可以**读取香蕉检测数据集。** 该数据集包括一个的CSV文件，内含目标类别标签和位于左上角和右下角的真实边界框坐标。

In [3]:
```
def read_data_bananas(is_train=True):
    """读取香蕉检测数据集中的图像和标签"""
    data_dir = d2l.download_extract('banana-detection')
    csv_fname = os.path.join(data_dir, 'bananas_train' if is_train
                             else 'bananas_val', 'label.csv')
    csv_data = pd.read_csv(csv_fname)
    csv_data = csv_data.set_index('img_name')
    images, targets = [], []
    for img_name, target in csv_data.iterrows():
        images.append(torchvision.io.read_image(
            os.path.join(data_dir, 'bananas_train' if is_train else
                         'bananas_val', 'images', f'{img_name}')))
        # 这里的target包含（类别，左上角x，左上角y，右下角x，右下角y），
        # 其中所有图像都具有相同的香蕉类（索引为0）
        targets.append(list(target))
    return images, torch.tensor(targets).unsqueeze(1) / 256
```

通过使用`read_data_bananas`函数读取图像和标签，以下`BananasDataset`类别将**创建一个自定义`Dataset`实例**来加载香蕉检测数据集。

In [4]:
```
class BananasDataset(torch.utils.data.Dataset):
    """一个用于加载香蕉检测数据集的自定义数据集"""
    def __init__(self, is_train):
        self.features, self.labels = read_data_bananas(is_train)
        print('read ' + str(len(self.features)) + (f' training examples' if
              is_train else f' validation examples'))

    def __getitem__(self, idx):
        return (self.features[idx].float(), self.labels[idx])

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

定义 `load_data_bananas` 函数，来**为训练集和测试集返回两个数据加载器实例。**对于测试集，无须按随机顺序读取它。

In [5]:
```
def load_data_bananas(batch_size):
    """加载香蕉检测数据集"""
    train_iter = torch.utils.data.DataLoader(BananasDataset(is_train=True),
                                             batch_size, shuffle=True)
    val_iter = torch.utils.data.DataLoader(BananasDataset(is_train=False),
                                           batch_size)
    return train_iter, val_iter
```

尝试读取一个小批量，并打印其中的图像和标签的形状。

图像的小批量的形状为（批量大小、通道数、高度、宽度），看起来很眼熟：它与我们之前图像分类任务中的相同。 标签的小批量的形状为（批量大小，m，5），其中m是数据集的任何图像中边界框可能出现的最大数量。

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。 通常来说，图像可能拥有不同数量个边界框；因此，在达到m之前，边界框少于m的图像将被非法边界框填充。

这样，每个边界框的标签将被长度为5的数组表示。 数组中的第一个元素是边界框中对象的类别，其中-1表示用于填充的非法边界框。 数组的其余四个元素是边界框左上角和右下角的（x，y）坐标值（值域在0到1之间）。

对于香蕉数据集而言，由于每张图像上只有一个边界框，因此m = 1。

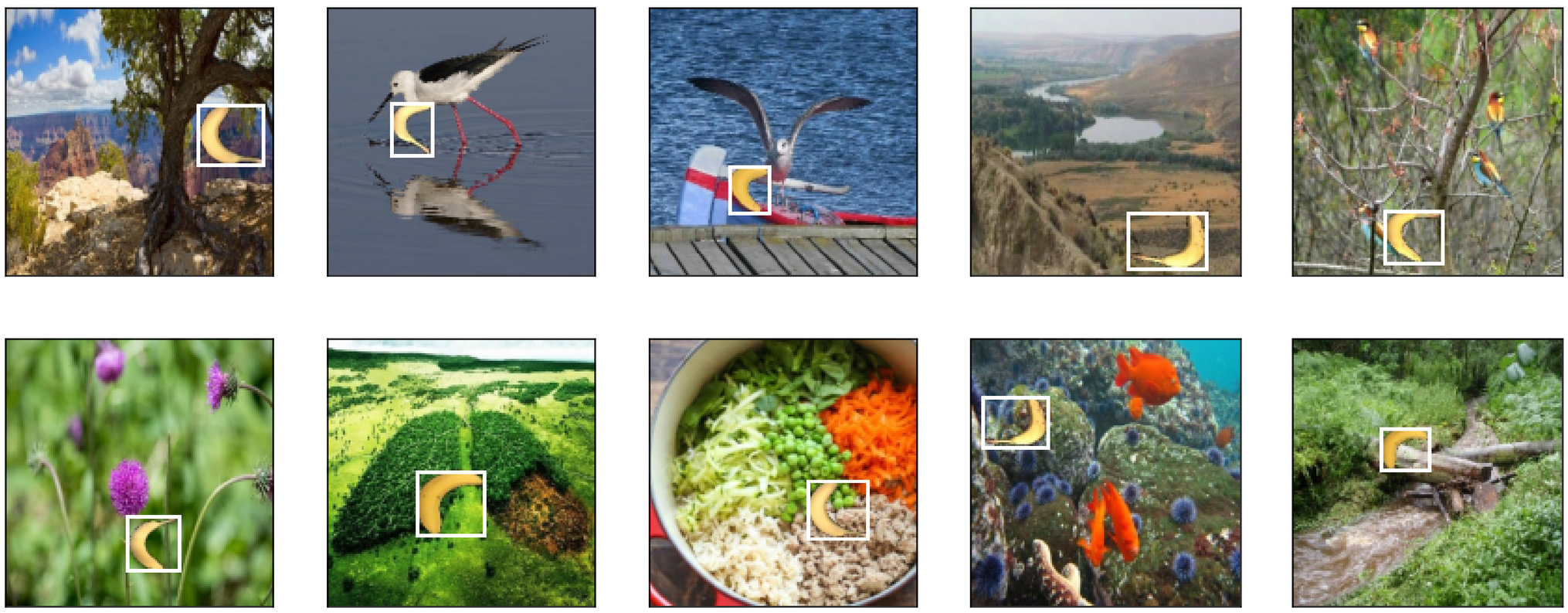
In [6]:
```
batch_size, edge_size = 32, 256
train_iter, _ = load_data_bananas(batch_size)
batch = next(iter(train_iter))
batch[0].shape, batch[1].shape
```

Out[6]:
```
read 1000 training examples
read 100 validation examples
(torch.Size([32, 3, 256, 256]), torch.Size([32, 1, 5]))
```

展示10幅带有真实边界框的图像。

可以看到在所有这些图像中香蕉的旋转角度、大小和位置都有所不同。 当然，这只是一个简单的人工数据集，实践中真实世界的数据集通常要复杂得多。

In [7]:
```
imgs = (batch[0][0:10].permute(0, 2, 3, 1)) / 255
axes = d2l.show_images(imgs, 2, 5, scale=3)
for ax, label in zip(axes, batch[1][0:10]):
    d2l.show_bboxes(ax, [label[0][1:5] * edge_size], colors=['w'])
```



2. 单发多框检测（SSD）

单发多框检测（SSD） Liu.Anguelov.Erhan.ea.2016。 该模型简单、快速且被广泛使用。尽管这只是其中一种目标检测模型，但其中的一些设计原则和实现细节也适用于其他模型。

模型

下图描述了单发多框检测模型的设计。 此模型主要由基础网络组成，其后是几个多尺度特征块。 基本网络用于从输入图像中提取特征，因此它可以使用深度卷积神经网络。

SSD 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。

回想一下在之前的多尺度目标检测中，通过深度神经网络分层表示图像的多尺度目标检测的设计。 如下图，由于接近顶部的多尺度特征图较小，但具有较大的感受野，它们适合检测较少但较大的物体。 简而言之，通过多尺度特征块，单发多框检测生成不同大小的锚框，并通过预测边界框的类别和偏移量来检测大小不同的目标，因此这是一个多尺度目标检测模型。

单发多框检测模型主要由一个基础网络块和若干多尺度特征块串联而成。

类别预测层

设目标类别的数量为 q。这样一来，锚框有 q + 1 个类别，其中0类是背景。 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。 单发多框检测采用卷积层的通道来输出类别预测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 ≤ j ≤ 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。

在下面定义了这样一个类别预测层，通过参数 `num_anchors` 和 `num_classes` 分别指定了 a 和 q。 该图层使用填充为 1 的 3 × 3 的卷积层。此卷积层的输入和输出的宽度和高度保持不变。

In [8]:
```
def cls_predictor(num_inputs, num_anchors, num_classes):
    return nn.Conv2d(num_inputs, num_anchors * (num_classes + 1),
                     kernel_size=3, padding=1)
```

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。

In [9]:
```
def bbox_predictor(num_inputs, num_anchors):
    return nn.Conv2d(num_inputs, num_anchors * 4, kernel_size=3, padding=1)
```

连结多尺度的预测

单发多框检测使用多尺度特征图来生成锚框并预测其类别和偏移量。 在不同的尺度下，特征图的形状或以同一单元为中心的锚框的数量可能会有所不同。 因此，不同尺度下预测输出的形状可能会有所不同。

在以下示例中，同一个小批量构建两个不同比例（`Y1`和`Y2`）的特征图，其中`Y2`的高度和宽度是`Y1`的一半。 以类别预测为例，假设`Y1`和`Y2`的每个单元分别生成了5个和3个锚框。 进一步假设目标类别的数量为10，对于特征图`Y1`和`Y2`，类别预测输出中的通道数分别为$5 \times (10 + 1) = 55$和$3 \times (10 + 1) = 33$，其中任一输出的形状是（批量大小，通道数，高度，宽度）。

In [10]:
```
def forward(x, block):
    return block(x)

Y1 = forward(torch.zeros((2, 8, 20, 20)), cls_predictor(8, 5, 10))
Y2 = forward(torch.zeros((2, 16, 10, 10)), cls_predictor(16, 3, 10))
Y1.shape, Y2.shape
```

Out[10]:
```
(torch.Size([2, 55, 20, 20]), torch.Size([2, 33, 10, 10]))
```

除了批量大小这一维度外，其他三个维度都具有不同的尺寸。 为了将这两个预测输出链接起来以提高计算效率，需要把这些张量转换为更一致的格式。

通道维包含中心相同的锚框的预测结果。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，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。

In [11]:
```
def flatten_pred(pred):
    return torch.flatten(pred.permute(0, 2, 3, 1), start_dim=1)

def concat_preds(preds):
    return torch.cat([flatten_pred(p) for p in preds], dim=1)
```

这样一来，尽管`Y1`和`Y2`在通道数、高度和宽度方面具有不同的大小，却仍然可以在同一个小批量的两个不同尺度上连接这两个预测输出。

In [12]:
```
concat_preds([Y1, Y2]).shape
```

Out[12]:
```
torch.Size([2, 25300])
```

高和宽减半块

为了在多个尺度下检测目标，下面定义了高和宽减半块 `down_sample_blk`，该模块将输入特征图的高度和宽度减半。 事实上，该块应用了 VGG 的模块设计。

更具体地说，每个高和宽减半块由两个填充为 1 的 3×3 的卷积层、以及步幅为 2 的 2×2 最大汇聚层组成。 其中，填充为 1 的 3×3 卷积层不改变特征图的形状。但是，其后的 2×2 的最大汇聚层将输入特征图的高度和宽度减少了一半。 对于此高和宽减半块的输入和输出特征图，因为 $1 \times 2 + (3 - 1) + (3 - 1) = 6$，所以输出中的每个单元在输入上都有一个 6×6 的感受野。因此，高和宽减半块会扩大每个单元在其输出特征图中的感受野。

In [13]:
```
def down_sample_blk(in_channels, out_channels):
    blk = []
    for _ in range(2):
        blk.append(nn.Conv2d(in_channels, out_channels,
                             kernel_size=3, padding=1))
        blk.append(nn.BatchNorm2d(out_channels))
        blk.append(nn.ReLU())
        in_channels = out_channels
    blk.append(nn.MaxPool2d(2))
    return nn.Sequential(*blk)
```

在以下示例中，构建的高和宽减半块会更改输入通道的数量，并将输入特征图的高度和宽度减半。

In [14]:
```
forward(torch.zeros((2, 3, 20, 20)), down_sample_blk(3, 10)).shape
```

Out[14]:
```
torch.Size([2, 10, 10, 10])
```

基本网络块

基本网络块用于从输入图像中抽取特征。 为了计算简洁，这里构造了一个小的基础网络，该网络串联3个高和宽减半块，并逐步将通道数翻倍。 给定输入图像的形状为256×256，此基本网络块输出的特征图形状为32×32（$256/2^3 = 32$）。

In [15]:
```
def base_net():
    blk = []
    num_filters = [3, 16, 32, 64]
    for i in range(len(num_filters) - 1):
        blk.append(down_sample_blk(num_filters[i], num_filters[i+1]))
    return nn.Sequential(*blk)

forward(torch.zeros((2, 3, 256, 256)), base_net()).shape
```

Out[15]:
```
torch.Size([2, 64, 32, 32])
```

完整的模型

完整的单发多框检测模型由五个模块组成。每个块生成的特征图既用于生成锚框，又用于预测这些锚框的类别和偏移量。

在这五个模块中，第一个是基本网络块，第二个到第四个是高和宽减半块，最后一个模块使用全局最大池将高度和宽度都降到1。从技术上讲，第二到第五个区块都是下图中的多尺度特征块。

 单发多框检测模型主要由一个基础网络块和若干多尺度特征块串联而成。

In [16]:
```
def get_blk(i):
    if i == 0:
        blk = base_net()
    elif i == 1:
```

```
        blk = down_sample_blk(64, 128)
    elif i == 4:
        blk = nn.AdaptiveMaxPool2d((1,1))
    else:
        blk = down_sample_blk(128, 128)
    return blk
```

现在**为每个块定义前向传播。** 与图像分类任务不同，此处的输出包括：CNN特征图`Y`；在当前尺度下根据`Y`生成的锚框；预测的这些锚框的类别和偏移量（基于`Y`）。

In [17]:
```
def blk_forward(X, blk, size, ratio, cls_predictor, bbox_predictor):
    Y = blk(X)
    anchors = d2l.multibox_prior(Y, sizes=size, ratios=ratio)
    cls_preds = cls_predictor(Y)
    bbox_preds = bbox_predictor(Y)
    return (Y, anchors, cls_preds, bbox_preds)
```

在 SSD 中，一个较接近顶部的多尺度特征块是用于检测较大目标的，因此需要生成更大的锚框。

在上面的前向传播中，在每个多尺度特征块上，通过调用的`multibox_prior`函数（在锚框章节定义，用来生成多个锚框）的`sizes`参数传递两个比例值的列表。

在下面，0.2和1.05之间的区间被均匀分成五个部分，以确定五个模块的在不同尺度下的较小值：0.2、0.37、0.54、0.71和0.88。 之后，他们较大的值由$\sqrt{0.2 \times 0.37} = 0.272$、$\sqrt{0.37 \times 0.54} = 0.447$等给出。

In [18]:
```
sizes = [[0.2, 0.272], [0.37, 0.447], [0.54, 0.619], [0.71, 0.79],
         [0.88, 0.961]]
ratios = [[1, 2, 0.5]] * 5
num_anchors = len(sizes[0]) + len(ratios[0]) - 1
```

现在，就可以按如下方式**定义完整的模型**`TinySSD`了。

In [19]:
```
class TinySSD(nn.Module):
    def __init__(self, num_classes, **kwargs):
        super(TinySSD, self).__init__(**kwargs)
        self.num_classes = num_classes
        idx_to_in_channels = [64, 128, 128, 128, 128]
        for i in range(5):
            # 即赋值语句self.blk_i=get_blk(i)
            setattr(self, f'blk_{i}', get_blk(i))
            setattr(self, f'cls_{i}', cls_predictor(idx_to_in_channels[i],
                                                    num_anchors, num_classes))
            setattr(self, f'bbox_{i}', bbox_predictor(idx_to_in_channels[i],
                                                      num_anchors))

    def forward(self, X):
        anchors, cls_preds, bbox_preds = [None] * 5, [None] * 5, [None] * 5
        for i in range(5):
            # getattr(self,'blk_%d'%i)即访问self.blk_i
            X, anchors[i], cls_preds[i], bbox_preds[i] = blk_forward(
                X, getattr(self, f'blk_{i}'), sizes[i], ratios[i],
                getattr(self, f'cls_{i}'), getattr(self, f'bbox_{i}'))
        anchors = torch.cat(anchors, dim=1)
        cls_preds = concat_preds(cls_preds)
        cls_preds = cls_preds.reshape(
            cls_preds.shape[0], -1, self.num_classes + 1)
        bbox_preds = concat_preds(bbox_preds)
        return anchors, cls_preds, bbox_preds
```

创建一个模型实例，然后使用它对一个 256 × 256 像素的小批量图像`X`**执行前向传播。**

如本节前面部分所示，第一个模块输出特征图的形状为 32 × 32。

第二到第四个模块为高和宽减半块，第五个模块为全局汇聚层。 由于以特征图的每个单元为中心有4个锚框生成，因此在所有五个尺度下，每个图像总共生成
$(32^2 + 16^2 + 8^2 + 4^2 + 1) \times 4 = 5444$个锚框。

In [20]:
```
net = TinySSD(num_classes=1)
X = torch.zeros((32, 3, 256, 256))
anchors, cls_preds, bbox_preds = net(X)

print('output anchors:', anchors.shape)
print('output class preds:', cls_preds.shape)
print('output bbox preds:', bbox_preds.shape)
```

```
output anchors: torch.Size([1, 5444, 4])
output class preds: torch.Size([32, 5444, 2])
output bbox preds: torch.Size([32, 21776])
```

训练模型

现在描述如何训练用于目标检测的单发多框检测模型。

读取数据集和初始化

首先，**读取** 之前描述的(**香蕉检测数据集**)。

In [21]:
```
batch_size = 32
train_iter, _ = d2l.load_data_bananas(batch_size)
```

```
read 1000 training examples
read 100 validation examples
```

香蕉检测数据集中，目标的类别数为1。 定义好模型后，需要**初始化其参数并定义优化算法。**

In [22]:
```
device, net = d2l.try_gpu(), TinySSD(num_classes=1)
trainer = torch.optim.SGD(net.parameters(), lr=0.2, weight_decay=5e-4)
```

定义损失函数和评价函数

目标检测有两种类型的损失。 第一种有关锚框类别的损失：可以简单地复用之前图像分类问题里一直使用的交叉熵损失函数来计算； 第二种有关正类锚框偏移量的损失：预测偏移量是一个回归问题。

但是，对于这个回归问题，这里不使用平方损失，而是使用 L_1 范数损失，即预测值和真实值之差的绝对值。

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，将锚框类别和偏移量的损失相加，以获得模型的最终损失函数。

In [23]:
```
cls_loss = nn.CrossEntropyLoss(reduction='none')
bbox_loss = nn.L1Loss(reduction='none')

def calc_loss(cls_preds, cls_labels, bbox_preds, bbox_labels, bbox_masks):
    batch_size, num_classes = cls_preds.shape[0], cls_preds.shape[2]
    cls = cls_loss(cls_preds.reshape(-1, num_classes),
                   cls_labels.reshape(-1)).reshape(batch_size, -1).mean(dim=1)
    bbox = bbox_loss(bbox_preds * bbox_masks,
```

```
                       bbox_labels * bbox_masks).mean(dim=1)
    return cls + bbox
```

可以沿用准确率评价分类结果。 由于偏移量使用了 L_1 范数损失，我们使用***平均绝对误差***来评价边界框的预测结果。这些预测结果是从生成的锚框及其预测偏移量中获得的。

In [24]:
```
def cls_eval(cls_preds, cls_labels):
    # 由于类别预测结果放在最后一维，argmax需要指定最后一维。
    return float((cls_preds.argmax(dim=-1).type(
        cls_labels.dtype) == cls_labels).sum())

def bbox_eval(bbox_preds, bbox_labels, bbox_masks):
    return float((torch.abs((bbox_labels - bbox_preds) * bbox_masks)).sum())
```

训练模型

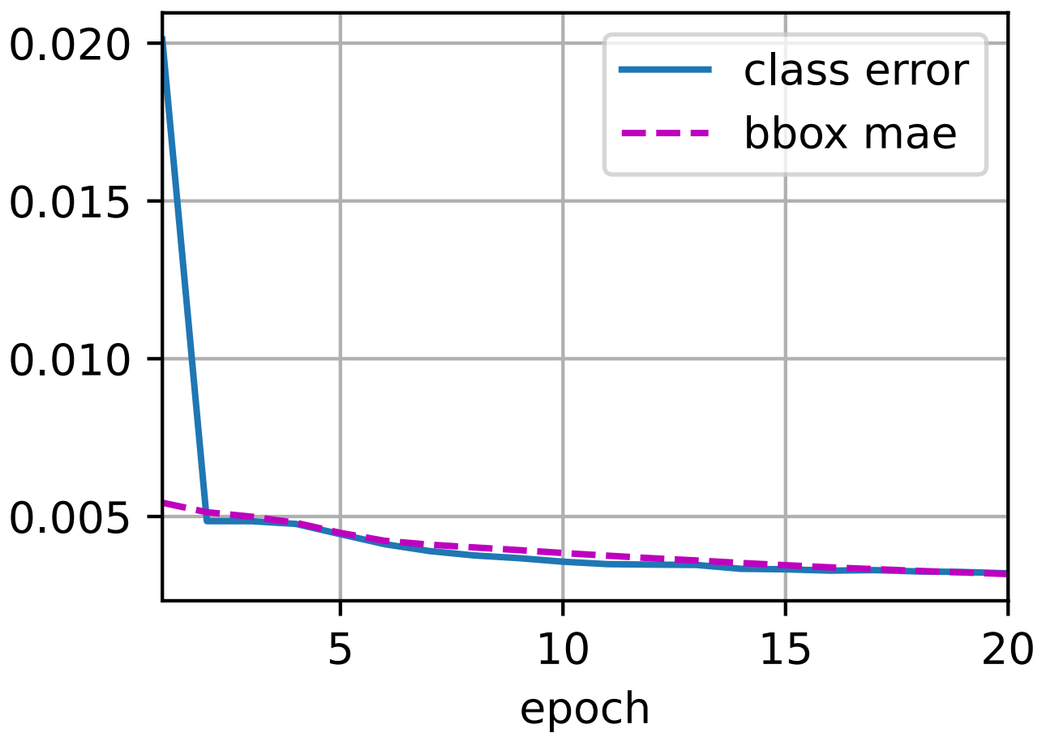
在训练模型时，需要在模型的前向传播过程中生成多尺度锚框（`anchors`），并预测其类别（`cls_preds`）和偏移量（`bbox_preds`）。

然后，根据标签信息`Y`为生成的锚框标记类别（`cls_labels`）和偏移量（`bbox_labels`）。

最后，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。

In [25]:
```
num_epochs, timer = 20, d2l.Timer()
animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                        legend=['class error', 'bbox mae'])
net = net.to(device)
for epoch in range(num_epochs):
    # 训练精确度的和，训练精确度的和中的示例数
    # 绝对误差的和，绝对误差的和中的示例数
    metric = d2l.Accumulator(4)
    net.train()
    for features, target in train_iter:
        timer.start()
        trainer.zero_grad()
        X, Y = features.to(device), target.to(device)
        # 生成多尺度的锚框，为每个锚框预测类别和偏移量
        anchors, cls_preds, bbox_preds = net(X)
        # 为每个锚框标注类别和偏移量
        bbox_labels, bbox_masks, cls_labels = d2l.multibox_target(anchors, Y)
        # 根据类别和偏移量的预测和标注值计算损失函数
        l = calc_loss(cls_preds, cls_labels, bbox_preds, bbox_labels,
                      bbox_masks)
        l.mean().backward()
        trainer.step()
        metric.add(cls_eval(cls_preds, cls_labels), cls_labels.numel(),
                   bbox_eval(bbox_preds, bbox_labels, bbox_masks),
                   bbox_labels.numel())
    cls_err, bbox_mae = 1 - metric[0] / metric[1], metric[2] / metric[3]
    animator.add(epoch + 1, (cls_err, bbox_mae))
print(f'class err {cls_err:.2e}, bbox mae {bbox_mae:.2e}')
print(f'{len(train_iter.dataset) / timer.stop():.1f} examples/sec on '
      f'{str(device)}')
```

```
class err 3.20e-03, bbox mae 3.18e-03
3140.6 examples/sec on cuda:0
```



预测目标

在预测阶段，希望能把图像里面所有感兴趣的目标检测出来。在下面的代码中，读取并调整测试图像的大小，然后将其转成卷积层需要的四维格式。

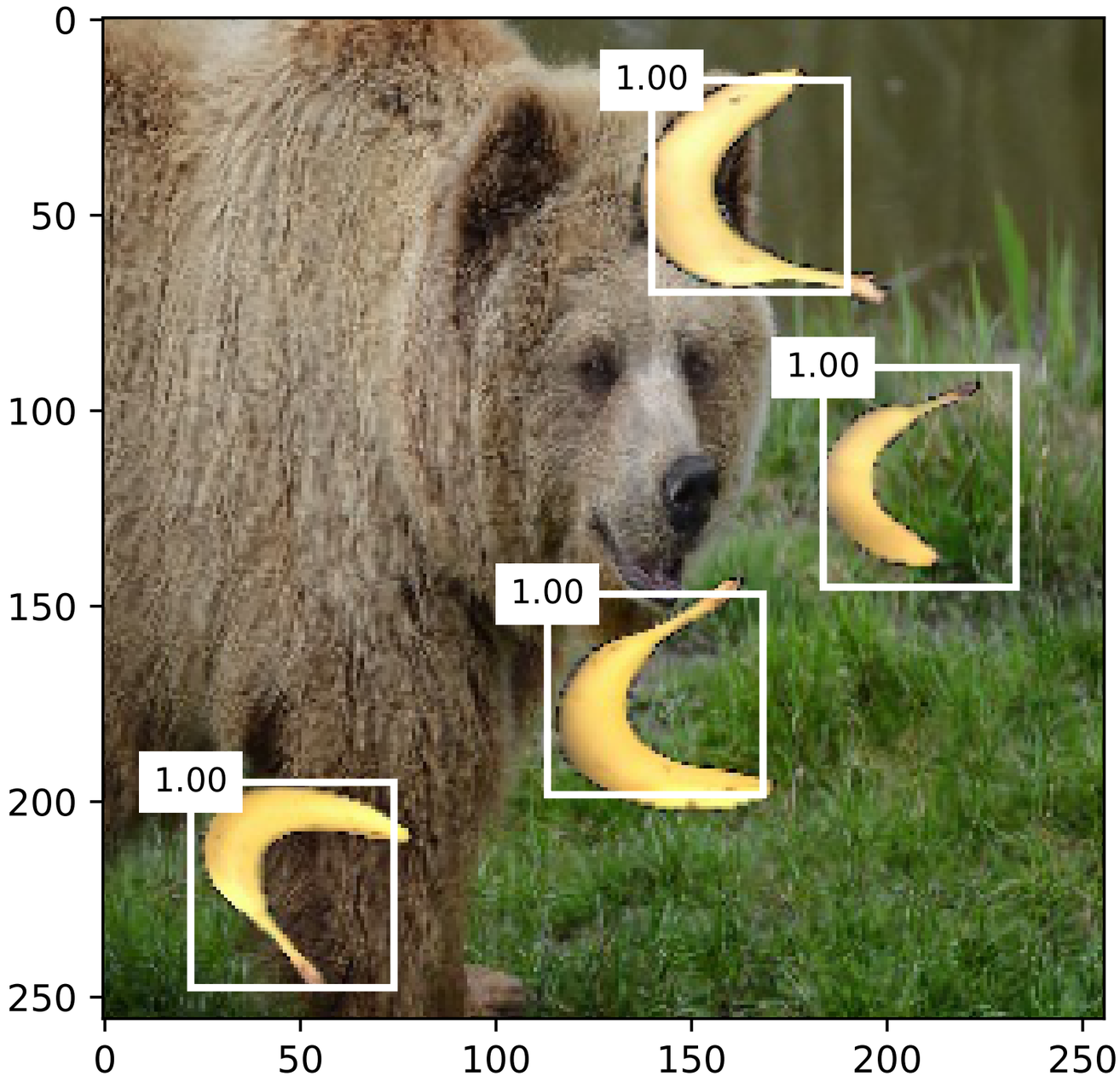
In [26]:
```
X = torchvision.io.read_image('./figs/banana.jpg').unsqueeze(0).float()
img = X.squeeze(0).permute(1, 2, 0).long()
```

使用下面的 `multibox_detection` 函数，可以根据锚框及其预测偏移量得到预测边界框。然后，通过非极大值抑制来移除相似的预测边界框。

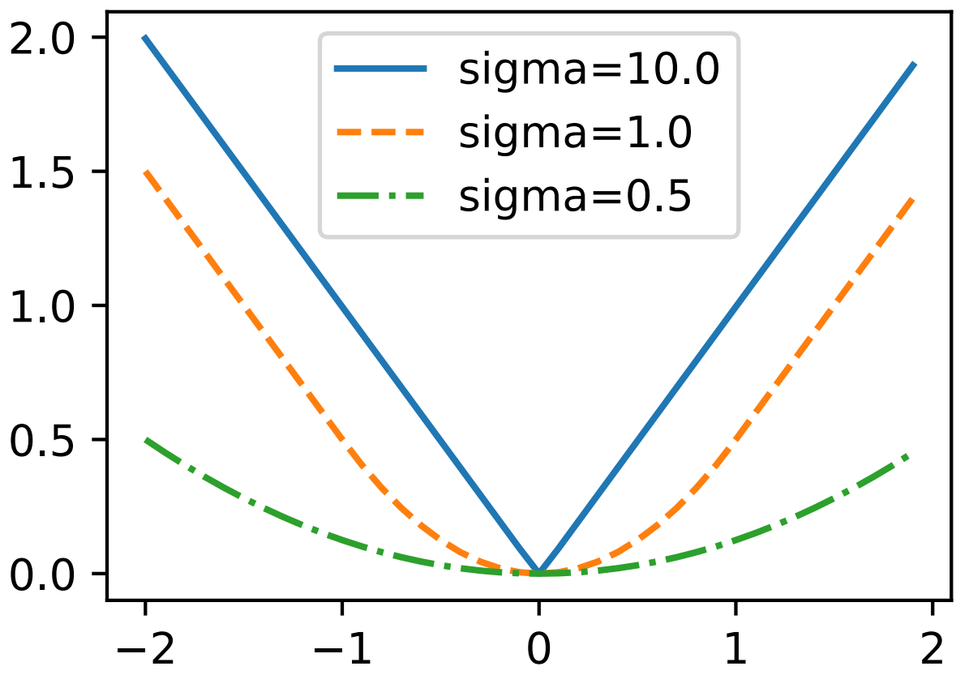
In [27]:
```
def predict(X):
    net.eval()
    anchors, cls_preds, bbox_preds = net(X.to(device))
    cls_probs = F.softmax(cls_preds, dim=2).permute(0, 2, 1)
    output = d2l.multibox_detection(cls_probs, bbox_preds, anchors)
    idx = [i for i, row in enumerate(output[0]) if row[0] != -1]
    return output[0, idx]

output = predict(X)
```

最后，筛选所有置信度不低于0.9的边界框，做为最终输出。

In [28]:
```
def display(img, output, threshold):
    d2l.set_figsize((5, 5))
    fig = d2l.plt.imshow(img)
    for row in output:
        score = float(row[1])
        if score < threshold:
            continue
        h, w = img.shape[0:2]
        bbox = [row[2:6] * torch.tensor((w, h, w, h), device=row.device)]
        d2l.show_bboxes(fig.axes, bbox, '%.2f' % score, 'w')

display(img, output.cpu(), threshold=0.9)
```



In [29]:
```
def smooth_l1(data, scalar):
    out = []
    for i in data:
        if abs(i) < 1 / (scalar ** 2):
            out.append(((scalar * i) ** 2) / 2)
        else:
            out.append(abs(i) - 0.5 / (scalar ** 2))
    return torch.tensor(out)

sigmas = [10, 1, 0.5]
lines = ['-', '--', '-.']
x = torch.arange(-2, 2, 0.1)
d2l.set_figsize()

for l, s in zip(lines, sigmas):
    y = smooth_l1(x, scalar=s)
    d2l.plt.plot(x, y, l, label='sigma=%.1f' % s)
d2l.plt.legend();
```



In [30]:
```
def focal_loss(gamma, x):
    return -(1 - x) ** gamma * torch.log(x)

x = torch.arange(0.01, 1, 0.01)
for l, gamma in zip(lines, [0, 1, 5]):
```

```
    y = d2l.plt.plot(x, focal_loss(gamma, x), l, label='gamma=%.1f' % gamma)
d2l.plt.legend();
```