13 Computer Vision

Many applications in the area of computer vision are closely related to our daily lives, now and in the future, whether medical diagnostics, driverless vehicles, camera monitoring, or smart filters. In recent years, deep learning technology has greatly enhanced computer vision systems' performance. It can be said that the most advanced computer vision applications are nearly inseparable from deep learning.

We have introduced deep learning models commonly used in the area of computer vision in the chapter "Convolutional Neural Networks" and have practiced simple image classification tasks. In this chapter, we will introduce image augmentation and fine tuning methods and apply them to image classification. Then, we will explore various methods of object detection. After that, we will learn how to use fully convolutional networks to perform semantic segmentation on images. Then, we explain how to use style transfer technology to generate images that look like the cover of this book. Finally, we will perform practice exercises on two important computer vision datasets to review the content of this chapter and the previous chapters.

13.1 Image Augmentation

We mentioned that large-scale datasets are prerequisites for the successful application of deep neural networks in Section 7.1. Image augmentation technology expands the scale of training datasets by making a series of random changes to the training images to produce similar, but different, training examples. Another way to explain image augmentation is that randomly changing training examples can reduce a model's dependence on certain properties, thereby improving its capability for generalization. For example, we can crop the images in different ways, so that the objects of interest appear in different positions, reducing the model's dependence on the position where objects appear. We can also adjust the brightness, color, and other factors to reduce model's sensitivity to color. It can be said that image augmentation technology contributed greatly to the success of AlexNet. In this section, we will discuss this technology, which is widely used in computer vision.

First, import the packages or modules required for the experiment in this section.

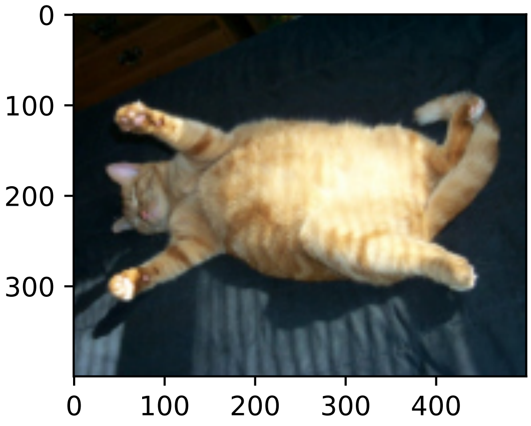
```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import autograd, gluon, image, init, np, npx
from mxnet.gluon import nn

npx.set_np()
```

13.1.1 Common Image Augmentation Method

In this experiment, we will use an image with a shape of 400×500 as an example.

```
d2l.set_figsize()
img = image.imread('../img/cat1.jpg')
d2l.plt.imshow(img.asnumpy());
```



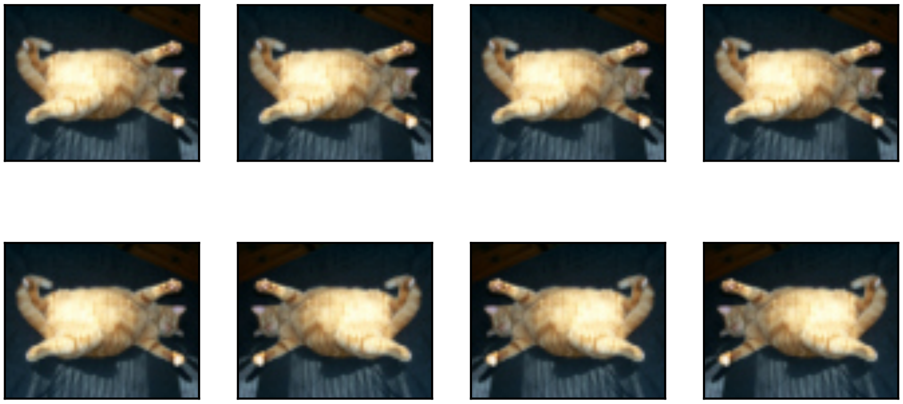
Most image augmentation methods have a certain degree of randomness. To make it easier for us to observe the effect of image augmentation, we next define the auxiliary function `apply`. This function runs the image augmentation method `aug` multiple times on the input image `img` and shows all results.

```
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)
```

Flipping and Cropping

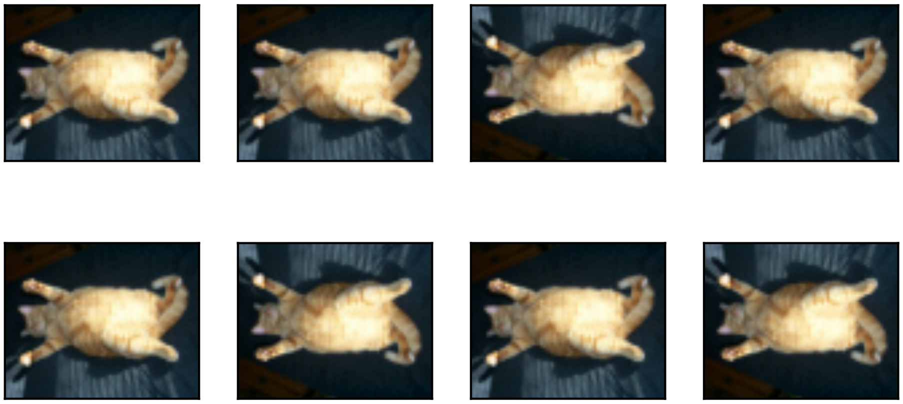
Flipping the image left and right usually does not change the category of the object. This is one of the earliest and most widely used methods of image augmentation. Next, we use the `transforms` module to create the `RandomFlipLeftRight` instance, which introduces a 50% chance that the image is flipped left and right.

```
apply(img, gluon.data.vision.transforms.RandomFlipLeftRight())
```



Flipping up and down is not as commonly used as flipping left and right. However, at least for this example image, flipping up and down does not hinder recognition. Next, we create a `RandomFlipTopBottom` instance for a 50% chance of flipping the image up and down.

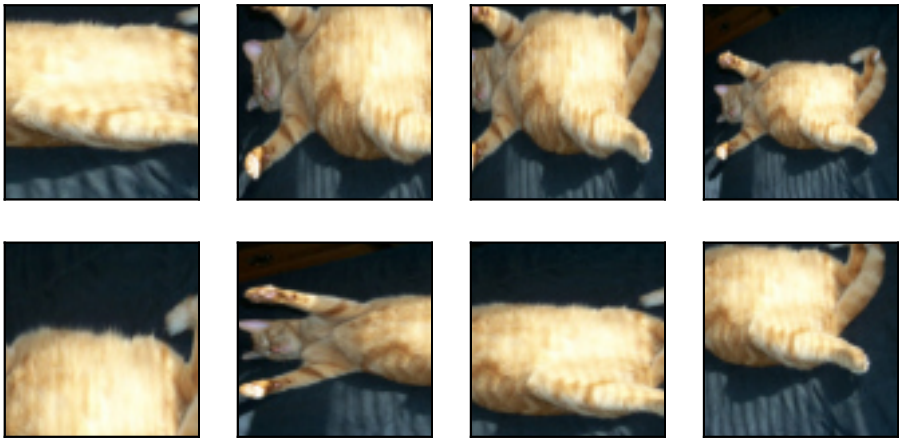
```python
apply(img, gluon.data.vision.transforms.RandomFlipTopBottom())
```



In the example image we used, the cat is in the middle of the image, but this may not be the case for all images. In Section 6.5, we explained that the pooling layer can reduce the sensitivity of the convolutional layer to the target location. In addition, we can make objects appear at different positions in the image in different proportions by randomly cropping the image. This can also reduce the sensitivity of the model to the target position.

In the following code, we randomly crop a region with an area of 10% to 100% of the original area, and the ratio of width to height of the region is randomly selected from between 0.5 and 2. Then, the width and height of the region are both scaled to 200 pixels. Unless otherwise stated, the random number between a and b in this section refers to a continuous value obtained by uniform sampling in the interval $[a, b]$.

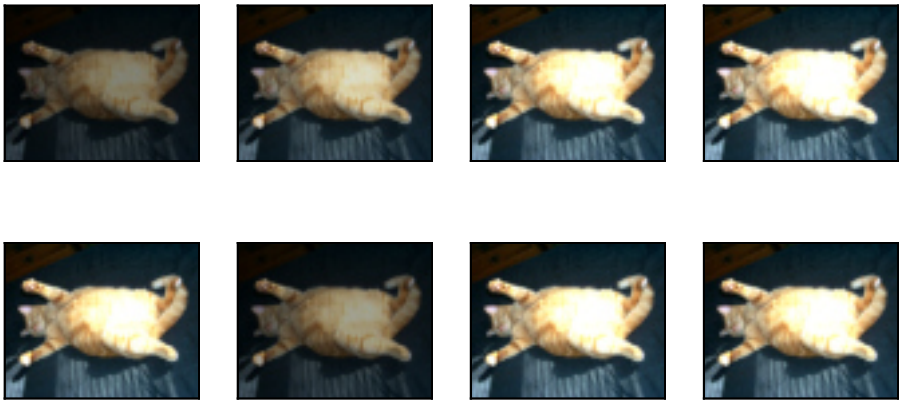
```python
shape_aug = gluon.data.vision.transforms.RandomResizedCrop(
    (200, 200), scale=(0.1, 1), ratio=(0.5, 2))
apply(img, shape_aug)
```



Changing the Color

Another augmentation method is changing colors. We can change four aspects of the image color: brightness, contrast, saturation, and hue. In the example below, we randomly change the brightness of the image to a value between 50% $(1-0.5)$ and 150% $(1+0.5)$ of the original image.

```
apply(img, gluon.data.vision.transforms.RandomBrightness(0.5))
```

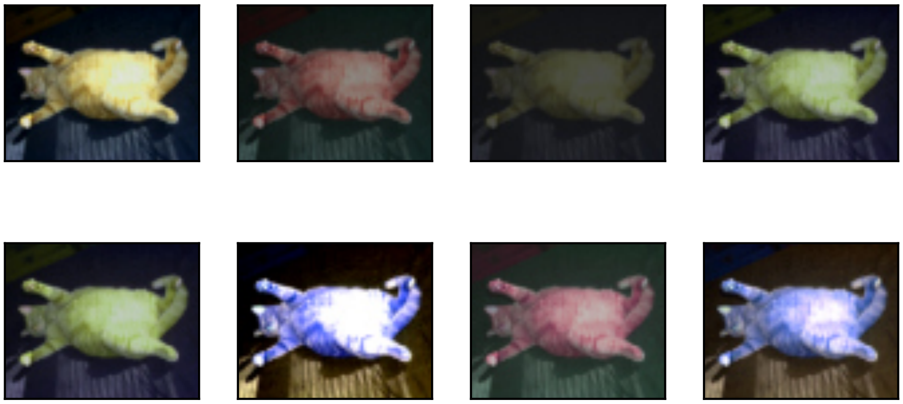


Similarly, we can randomly change the hue of the image.

```
apply(img, gluon.data.vision.transforms.RandomHue(0.5))
```

We can also create a RandomColorJitter instance and set how to randomly change the brightness, contrast, saturation, and hue of the image at the same time.

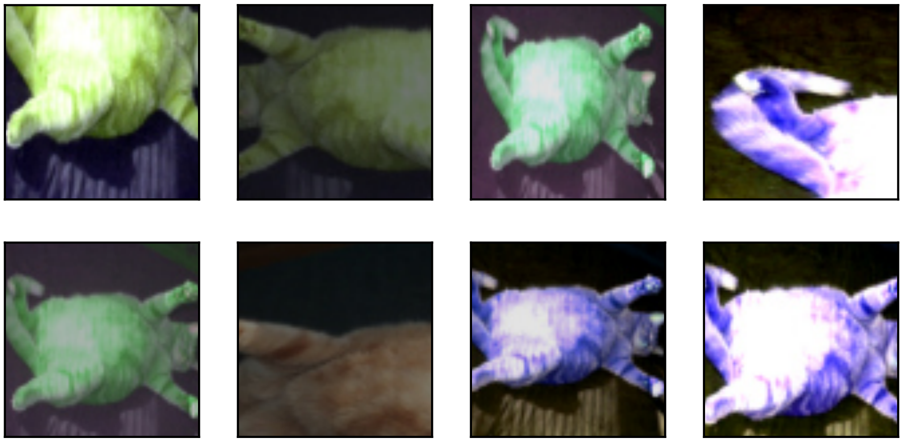
```
color_aug = gluon.data.vision.transforms.RandomColorJitter(
    brightness=0.5, contrast=0.5, saturation=0.5, hue=0.5)
apply(img, color_aug)
```



Overlying Multiple Image Augmentation Methods

In practice, we will overlay multiple image augmentation methods. We can overlay the different image augmentation methods defined above and apply them to each image by using a Compose instance.

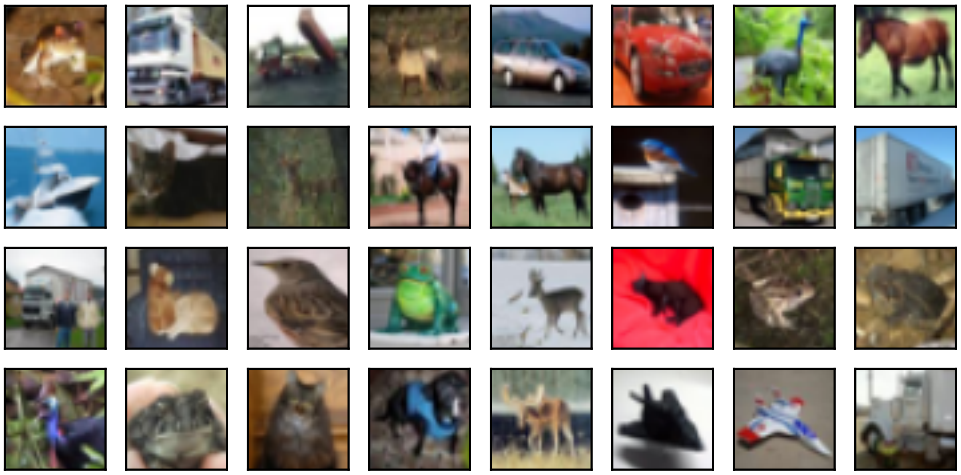
```
augs = gluon.data.vision.transforms.Compose([
    gluon.data.vision.transforms.RandomFlipLeftRight(), color_aug, shape_aug])
apply(img, augs)
```



13.1.2 Using an Image Augmentation Training Model

Next, we will look at how to apply image augmentation in actual training. Here, we use the CIFAR-10 dataset, instead of the Fashion-MNIST dataset we have been using. This is because the position and size of the objects in the Fashion-MNIST dataset have been normalized, and the differences in color and size of the objects in CIFAR-10 dataset are more significant. The first 32 training images in the CIFAR-10 dataset are shown below.

```
d2l.show_images(gluon.data.vision.CIFAR10(
    train=True)[0:32][0], 4, 8, scale=0.8);
```



In order to obtain definitive results during prediction, we usually only apply image augmentation to the training example, and do not use image augmentation with random operations during prediction. Here, we only use the simplest random left-right flipping method. In addition, we use a ToTensor instance to convert minibatch images into the format required by MXNet, i.e., 32-bit floating point numbers with the shape of (batch size, number of channels, height, width) and value range between 0 and 1.

```
train_augs = gluon.data.vision.transforms.Compose([
    gluon.data.vision.transforms.RandomFlipLeftRight(),
    gluon.data.vision.transforms.ToTensor()])

test_augs = gluon.data.vision.transforms.Compose([
    gluon.data.vision.transforms.ToTensor()])
```

Next, we define an auxiliary function to make it easier to read the image and apply image augmentation. The `transform_first` function provided by Gluon's dataset applies image augmentation to the first element of each training example (image and label), i.e., the element at the top of the image. For detailed descriptions of `DataLoader`, refer to Section 3.5.

```
def load_cifar10(is_train, augs, batch_size):
    return gluon.data.DataLoader(
        gluon.data.vision.CIFAR10(train=is_train).transform_first(augs),
        batch_size=batch_size, shuffle=is_train,
        num_workers=d2l.get_dataloader_workers())
```

Using a Multi-GPU Training Model

We train the ResNet-18 model described in Section 7.6 on the CIFAR-10 dataset. We will also apply the methods described in Section 12.6 and use a multi-GPU training model.

Next, we define the training function to train and evaluate the model using multiple GPUs.

```
#@save
def train_batch_ch13(net, features, labels, loss, trainer, devices,
                     split_f=d2l.split_batch):
    X_shards, y_shards = split_f(features, labels, devices)
    with autograd.record():
        pred_shards = [net(X_shard) for X_shard in X_shards]
        ls = [loss(pred_shard, y_shard) for pred_shard, y_shard
              in zip(pred_shards, y_shards)]
    for l in ls:
        l.backward()
    # The True flag allows parameters with stale gradients, which is useful
    # later (e.g., in fine-tuning BERT)
    trainer.step(labels.shape[0], ignore_stale_grad=True)
    train_loss_sum = sum([float(l.sum()) for l in ls])
    train_acc_sum = sum(d2l.accuracy(pred_shard, y_shard)
                        for pred_shard, y_shard in zip(pred_shards, y_shards))
    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(), split_f=d2l.split_batch):
    num_batches, timer = len(train_iter), d2l.Timer()
    animator = d2l.Animator(xlabel='epoch', xlim=[0, num_epochs], ylim=[0, 1],
                            legend=['train loss', 'train acc', 'test acc'])
    for epoch in range(num_epochs):
        # Store training_loss, training_accuracy, num_examples, num_features
        metric = d2l.Accumulator(4)
```

(continues on next page)

(continued from previous page)

```
        for i, (features, labels) in enumerate(train_iter):
            timer.start()
            l, acc = train_batch_ch13(
                net, features, labels, loss, trainer, devices, split_f)
            metric.add(l, acc, labels.shape[0], labels.size)
            timer.stop()
            if (i + 1) % (num_batches // 5) == 0:
                animator.add(epoch + i / num_batches,
                             (metric[0] / metric[2], metric[1] / metric[3],
                              None))
        test_acc = d2l.evaluate_accuracy_gpus(net, test_iter, split_f)
        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)}')
```

Now, we can define the train_with_data_aug function to use image augmentation to train the model. This function obtains all available GPUs and uses Adam as the optimization algorithm for training. It then applies image augmentation to the training dataset, and finally calls the train function just defined to train and evaluate the model.

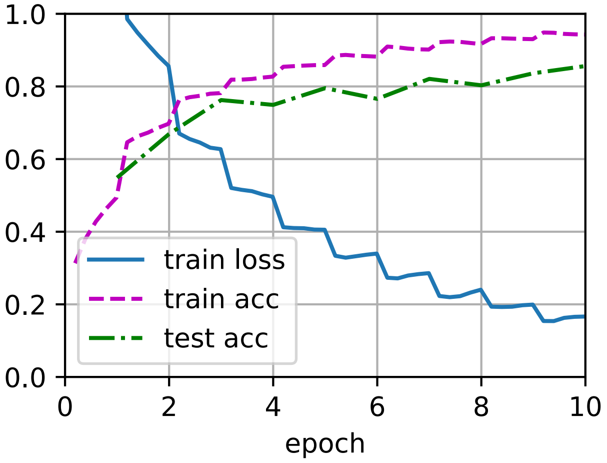
```
batch_size, devices, net = 256, d2l.try_all_gpus(), d2l.resnet18(10)
net.initialize(init=init.Xavier(), ctx=devices)

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 = gluon.loss.SoftmaxCrossEntropyLoss()
    trainer = gluon.Trainer(net.collect_params(), 'adam',
                            {'learning_rate': lr})
    train_ch13(net, train_iter, test_iter, loss, trainer, 10, devices)
```

Now we train the model using image augmentation of random flipping left and right.

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

```
loss 0.166, train acc 0.943, test acc 0.856
3897.5 examples/sec on [gpu(0), gpu(1)]
```



Summary

- Image augmentation generates random images based on existing training data to cope with overfitting.
- In order to obtain definitive results during prediction, we usually only apply image augmentation to the training example, and do not use image augmentation with random operations during prediction.
- We can obtain classes related to image augmentation from Gluon's `transforms` module.

Exercises

1. Train the model without using image augmentation: `train_with_data_aug(no_aug, no_aug)`. Compare training and testing accuracy when using and not using image augmentation. Can this comparative experiment support the argument that image augmentation can mitigate overfitting? Why?
2. Add different image augmentation methods in model training based on the CIFAR-10 dataset. Observe the implementation results.
3. With reference to the MXNet documentation, what other image augmentation methods are provided in Gluon's `transforms` module?

Discussions[186]

[186] https://discuss.d2l.ai/t/367

13.2 Fine-Tuning

In earlier chapters, we discussed how to train models on the Fashion-MNIST training dataset, which only has 60,000 images. We also described ImageNet, the most widely used large-scale image dataset in the academic world, with more than 10 million images and objects of over 1000 categories. However, the size of datasets that we often deal with is usually larger than the first, but smaller than the second.

Assume we want to identify different kinds of chairs in images and then push the purchase link to the user. One possible method is to first find a hundred common chairs, take one thousand different images with different angles for each chair, and then train a classification model on the collected image dataset. Although this dataset may be larger than Fashion-MNIST, the number of examples is still less than one tenth of ImageNet. This may result in the overfitting of the complicated model applicable to ImageNet on this dataset. At the same time, because of the limited amount of data, the accuracy of the final trained model may not meet the practical requirements.

In order to deal with the above problems, an obvious solution is to collect more data. However, collecting and labeling data can consume a lot of time and money. For example, in order to collect the ImageNet datasets, researchers have spent millions of dollars of research funding. Although, recently, data collection costs have dropped significantly, the costs still cannot be ignored.

Another solution is to apply transfer learning to migrate the knowledge learned from the source dataset to the target dataset. For example, although the images in ImageNet are mostly unrelated to chairs, models trained on this dataset can extract more general image features that can help identify edges, textures, shapes, and object composition. These similar features may be equally effective for recognizing a chair.

In this section, we introduce a common technique in transfer learning: fine tuning. As shown in Fig. 13.2.1, fine tuning consists of the following four steps:

1. Pre-train a neural network model, i.e., the source model, on a source dataset (e.g., the ImageNet dataset).
2. Create a new neural network model, i.e., the target model. This replicates all model designs and their parameters on the source model, except the output layer. We assume that these model parameters contain the knowledge learned from the source dataset and this knowledge will be equally applicable to the target dataset. We also assume that the output layer of the source model is closely related to the labels of the source dataset and is therefore not used in the target model.
3. Add an output layer whose output size is the number of target dataset categories to the target model, and randomly initialize the model parameters of this layer.
4. Train the target model on a target dataset, such as a chair dataset. We will train the output layer from scratch, while the parameters of all remaining layers are fine-tuned based on the parameters of the source model.

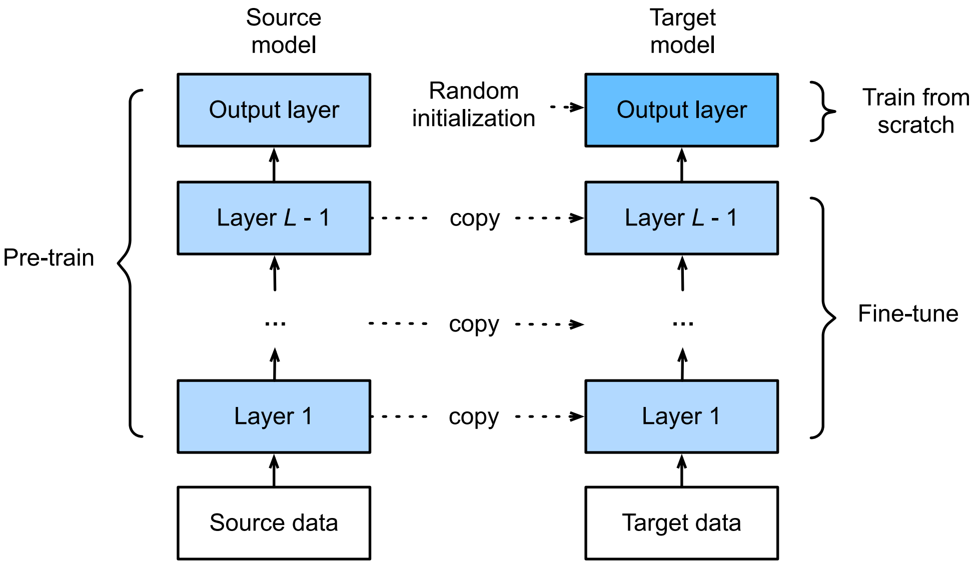


Fig. 13.2.1: Fine tuning.

13.2.1 Hot Dog Recognition

Next, we will use a specific example for practice: hot dog recognition. We will fine-tune the ResNet model trained on the ImageNet dataset based on a small dataset. This small dataset contains thousands of images, some of which contain hot dogs. We will use the model obtained by fine tuning to identify whether an image contains a hot dog.

First, import the packages and modules required for the experiment. Gluon's `model_zoo` package provides a common pre-trained model. If you want to get more pre-trained models for computer vision, you can use the GluonCV Toolkit[187].

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import gluon, init, np, npx
from mxnet.gluon import nn
import os

npx.set_np()
```

Obtaining the Dataset

The hot dog dataset we use was taken from online images and contains $1,400$ positive images containing hot dogs and the same number of negative images containing other foods. $1,000$ images of various classes are used for training and the rest are used for testing.

We first download the compressed dataset and get two folders `hotdog/train` and `hotdog/test`. Both folders have `hotdog` and `not-hotdog` category subfolders, each of which has corresponding image files.

[187] https://gluon-cv.mxnet.io

```
#@save
d2l.DATA_HUB['hotdog'] = (d2l.DATA_URL+'hotdog.zip',
                          'fba480ffa8aa7e0febbb511d181409f899b9baa5')

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

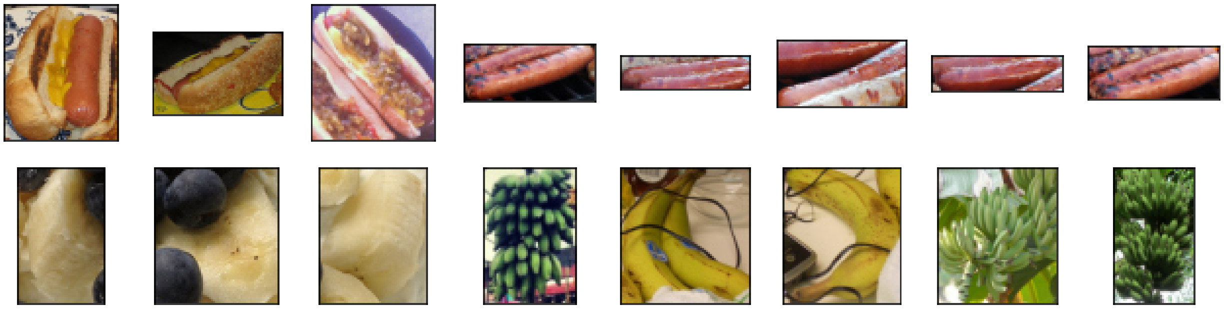
```
Downloading ../data/hotdog.zip from http://d2l-data.s3-accelerate.amazonaws.com/hotdog.zip...
```

We create two ImageFolderDataset instances to read all the image files in the training dataset and testing dataset, respectively.

```
train_imgs = gluon.data.vision.ImageFolderDataset(
    os.path.join(data_dir, 'train'))
test_imgs = gluon.data.vision.ImageFolderDataset(
    os.path.join(data_dir, 'test'))
```

The first 8 positive examples and the last 8 negative images are shown below. As you can see, the images vary in size and aspect ratio.

```
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);
```



During training, we first crop a random area with random size and random aspect ratio from the image and then scale the area to an input with a height and width of 224 pixels. During testing, we scale the height and width of images to 256 pixels, and then crop the center area with height and width of 224 pixels to use as the input. In addition, we normalize the values of the three RGB (red, green, and blue) color channels. The average of all values of the channel is subtracted from each value and then the result is divided by the standard deviation of all values of the channel to produce the output.

```
# We specify the mean and variance of the three RGB channels to normalize the
# image channel
normalize = gluon.data.vision.transforms.Normalize(
    [0.485, 0.456, 0.406], [0.229, 0.224, 0.225])

train_augs = gluon.data.vision.transforms.Compose([
    gluon.data.vision.transforms.RandomResizedCrop(224),
    gluon.data.vision.transforms.RandomFlipLeftRight(),
    gluon.data.vision.transforms.ToTensor(),
```

(continues on next page)

(continued from previous page)

```
    normalize])

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

Defining and Initializing the Model

We use ResNet-18, which was pre-trained on the ImageNet dataset, as the source model. Here, we specify `pretrained=True` to automatically download and load the pre-trained model parameters. The first time they are used, the model parameters need to be downloaded from the Internet.

```
pretrained_net = gluon.model_zoo.vision.resnet18_v2(pretrained=True)
```

The pre-trained source model instance contains two member variables: `features` and `output`. The former contains all layers of the model, except the output layer, and the latter is the output layer of the model. The main purpose of this division is to facilitate the fine tuning of the model parameters of all layers except the output layer. The member variable `output` of source model is given below. As a fully connected layer, it transforms ResNet's final global average pooling layer output into 1000 class output on the ImageNet dataset.

```
pretrained_net.output
```

```
Dense(512 -> 1000, linear)
```

We then build a new neural network to use as the target model. It is defined in the same way as the pre-trained source model, but the final number of outputs is equal to the number of categories in the target dataset. In the code below, the model parameters in the member variable `features` of the target model instance `finetune_net` are initialized to model parameters of the corresponding layer of the source model. Because the model parameters in `features` are obtained by pre-training on the ImageNet dataset, it is good enough. Therefore, we generally only need to use small learning rates to "fine-tune" these parameters. In contrast, model parameters in the member variable `output` are randomly initialized and generally require a larger learning rate to learn from scratch. Assume the learning rate in the `Trainer` instance is η and use a learning rate of 10η to update the model parameters in the member variable `output`.

```
finetune_net = gluon.model_zoo.vision.resnet18_v2(classes=2)
finetune_net.features = pretrained_net.features
finetune_net.output.initialize(init.Xavier())
# The model parameters in output will be updated using a learning rate ten
# times greater
finetune_net.output.collect_params().setattr('lr_mult', 10)
```

Fine Tuning the Model

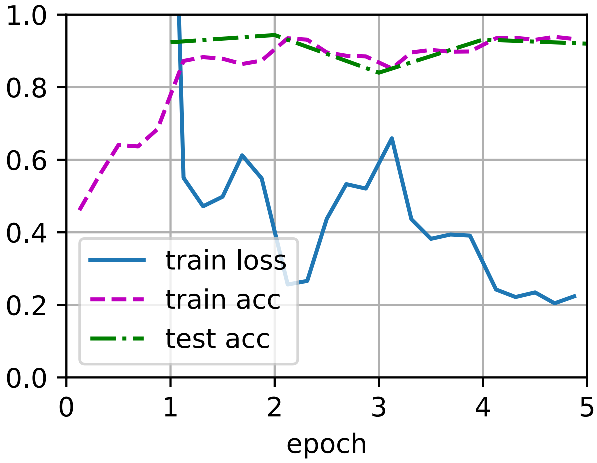
We first define a training function `train_fine_tuning` that uses fine tuning so it can be called multiple times.

```
def train_fine_tuning(net, learning_rate, batch_size=128, num_epochs=5):
    train_iter = gluon.data.DataLoader(
        train_imgs.transform_first(train_augs), batch_size, shuffle=True)
    test_iter = gluon.data.DataLoader(
        test_imgs.transform_first(test_augs), batch_size)
    devices = d2l.try_all_gpus()
    net.collect_params().reset_ctx(devices)
    net.hybridize()
    loss = gluon.loss.SoftmaxCrossEntropyLoss()
    trainer = gluon.Trainer(net.collect_params(), 'sgd', {
        'learning_rate': learning_rate, 'wd': 0.001})
    d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs,
                   devices)
```

We set the learning rate in the `Trainer` instance to a smaller value, such as 0.01, in order to fine-tune the model parameters obtained in pretraining. Based on the previous settings, we will train the output layer parameters of the target model from scratch using a learning rate ten times greater.

```
train_fine_tuning(finetune_net, 0.01)
```

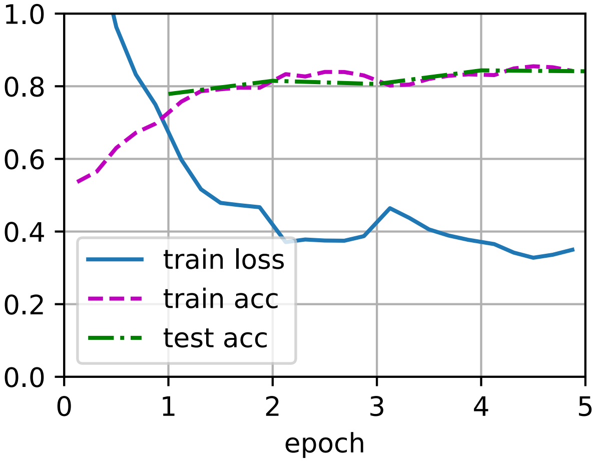
```
loss 0.230, train acc 0.931, test acc 0.920
352.8 examples/sec on [gpu(0), gpu(1)]
```



For comparison, we define an identical model, but initialize all of its model parameters to random values. Since the entire model needs to be trained from scratch, we can use a larger learning rate.

```
scratch_net = gluon.model_zoo.vision.resnet18_v2(classes=2)
scratch_net.initialize(init=init.Xavier())
train_fine_tuning(scratch_net, 0.1)
```

```
loss 0.350, train acc 0.842, test acc 0.841
352.7 examples/sec on [gpu(0), gpu(1)]
```



As you can see, the fine-tuned model tends to achieve higher precision in the same epoch because the initial values of the parameters are better.

Summary

- Transfer learning migrates the knowledge learned from the source dataset to the target dataset. Fine tuning is a common technique for transfer learning.
- The target model replicates all model designs and their parameters on the source model, except the output layer, and fine-tunes these parameters based on the target dataset. In contrast, the output layer of the target model needs to be trained from scratch.
- Generally, fine tuning parameters use a smaller learning rate, while training the output layer from scratch can use a larger learning rate.

Exercises

1. Keep increasing the learning rate of `finetune_net`. How does the precision of the model change?
2. Further tune the hyperparameters of `finetune_net` and `scratch_net` in the comparative experiment. Do they still have different precisions?
3. Set the parameters in `finetune_net.features` to the parameters of the source model and do not update them during training. What will happen? You can use the following code.

```
finetune_net.features.collect_params().setattr('grad_req', 'null')
```

4. In fact, there is also a "hotdog" class in the `ImageNet` dataset. Its corresponding weight parameter at the output layer can be obtained by using the following code. How can we use this parameter?

```
weight = pretrained_net.output.weight
hotdog_w = np.split(weight.data(), 1000, axis=0)[713]
hotdog_w.shape
```

```
(1, 512)
```

Discussions[188]

13.3 Object Detection and Bounding Boxes

In the previous section, we introduced many models for image classification. In image classification tasks, we assume that there is only one main target in the image and we only focus on how to identify the target category. However, in many situations, there are multiple targets in the image that we are interested in. We not only want to classify them, but also want to obtain their specific positions in the image. In computer vision, we refer to such tasks as object detection (or object recognition).

Object detection is widely used in many fields. For example, in self-driving technology, we need to plan routes by identifying the locations of vehicles, pedestrians, roads, and obstacles in the captured video image. Robots often perform this type of task to detect targets of interest. Systems in the security field need to detect abnormal targets, such as intruders or bombs.

In the next few sections, we will introduce multiple deep learning models used for object detection. Before that, we should discuss the concept of target location. First, import the packages and modules required for the experiment.

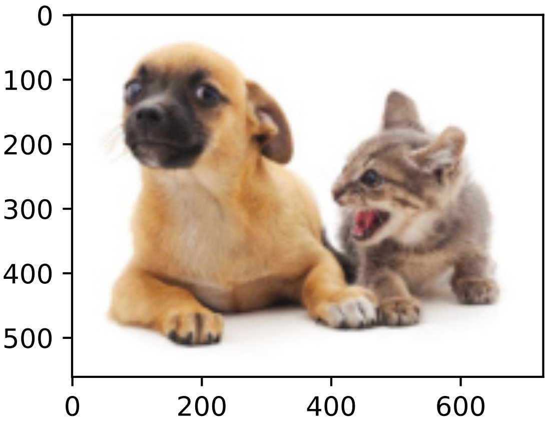
```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import image, npx

npx.set_np()
```

Next, we will load the sample images that will be used in this section. We can see there is a dog on the left side of the image and a cat on the right. They are the two main targets in this image.

```
d2l.set_figsize()
img = image.imread('../img/catdog.jpg').asnumpy()
d2l.plt.imshow(img);
```

[188] https://discuss.d2l.ai/t/368



13.3.1 Bounding Box

In object detection, we usually use a bounding box to describe the target location. The bounding box is a rectangular box that can be determined by the x and y axis coordinates in the upper-left corner and the x and y axis coordinates in the lower-right corner of the rectangle. We will define the bounding boxes of the dog and the cat in the image based on the coordinate information in the above image. The origin of the coordinates in the above image is the upper left corner of the image, and to the right and down are the positive directions of the x axis and the y axis, respectively.

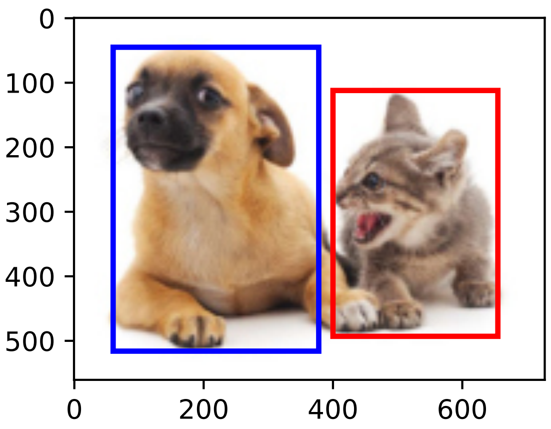
```
# bbox is the abbreviation for bounding box
dog_bbox, cat_bbox = [60, 45, 378, 516], [400, 112, 655, 493]
```

We can draw the bounding box in the image to check if it is accurate. Before drawing the box, we will define a helper function `bbox_to_rect`. It represents the bounding box in the bounding box format of `matplotlib`.

```
#@save
def bbox_to_rect(bbox, color):
    """Convert bounding box to matplotlib format."""
    # Convert the bounding box (top-left x, top-left y, bottom-right x,
    # bottom-right y) format to matplotlib format: ((upper-left x,
    # upper-left y), width, height)
    return d2l.plt.Rectangle(
        xy=(bbox[0], bbox[1]), width=bbox[2]-bbox[0], height=bbox[3]-bbox[1],
        fill=False, edgecolor=color, linewidth=2)
```

After loading the bounding box on the image, we can see that the main outline of the target is basically inside the box.

```
fig = d2l.plt.imshow(img)
fig.axes.add_patch(bbox_to_rect(dog_bbox, 'blue'))
fig.axes.add_patch(bbox_to_rect(cat_bbox, 'red'));
```



Summary

- In object detection, we not only need to identify all the objects of interest in the image, but also their positions. The positions are generally represented by a rectangular bounding box.

Exercises

1. Find some images and try to label a bounding box that contains the target. Compare the difference between the time it takes to label the bounding box and label the category.

Discussions[189]

13.4 Anchor Boxes

Object detection algorithms usually sample a large number of regions in the input image, determine whether these regions contain objects of interest, and adjust the edges of the regions so as to predict the ground-truth bounding box of the target more accurately. Different models may use different region sampling methods. Here, we introduce one such method: it generates multiple bounding boxes with different sizes and aspect ratios while centering on each pixel. These bounding boxes are called anchor boxes. We will practice object detection based on anchor boxes in the following sections.

First, import the packages or modules required for this section. Here, we have modified the printing accuracy of NumPy. Because printing tensors actually calls the print function of NumPy, the floating-point numbers in tensors printed in this section are more concise.

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import gluon, image, np, npx

np.set_printoptions(2)
npx.set_np()
```

[189] https://discuss.d2l.ai/t/369

13.4.1 Generating Multiple Anchor Boxes

Assume that the input image has a height of h and width of w. We generate anchor boxes with different shapes centered on each pixel of the image. Assume the size is $s \in (0, 1]$, the aspect ratio is $r > 0$, and the width and height of the anchor box are $ws\sqrt{r}$ and $hs/\sqrt{r}$, respectively. When the center position is given, an anchor box with known width and height is determined.

Below we set a set of sizes $s_1, \ldots, s_n$ and a set of aspect ratios $r_1, \ldots, r_m$. If we use a combination of all sizes and aspect ratios with each pixel as the center, the input image will have a total of $whnm$ anchor boxes. Although these anchor boxes may cover all ground-truth bounding boxes, the computational complexity is often excessive. Therefore, we are usually only interested in a combination containing s_1 or r_1 sizes and aspect ratios, that is:

$$(s_1, r_1), (s_1, r_2), \ldots, (s_1, r_m), (s_2, r_1), (s_3, r_1), \ldots, (s_n, r_1). \tag{13.4.1}$$

That is, the number of anchor boxes centered on the same pixel is $n + m - 1$. For the entire input image, we will generate a total of $wh(n + m - 1)$ anchor boxes.

The above method of generating anchor boxes has been implemented in the `multibox_prior` function. We specify the input, a set of sizes, and a set of aspect ratios, and this function will return all the anchor boxes entered.

```
img = image.imread('../img/catdog.jpg').asnumpy()
h, w = img.shape[0:2]

print(h, w)
X = np.random.uniform(size=(1, 3, h, w))  # Construct input data
Y = npx.multibox_prior(X, sizes=[0.75, 0.5, 0.25], ratios=[1, 2, 0.5])
Y.shape
```

```
561 728
```

```
(1, 2042040, 4)
```

We can see that the shape of the returned anchor box variable `y` is (batch size, number of anchor boxes, 4). After changing the shape of the anchor box variable `y` to (image height, image width, number of anchor boxes centered on the same pixel, 4), we can obtain all the anchor boxes centered on a specified pixel position. In the following example, we access the first anchor box centered on (250, 250). It has four elements: the x, y axis coordinates in the upper-left corner and the x, y axis coordinates in the lower-right corner of the anchor box. The coordinate values of the x and y axis are divided by the width and height of the image, respectively, so the value range is between 0 and 1.

```
boxes = Y.reshape(h, w, 5, 4)
boxes[250, 250, 0, :]
```

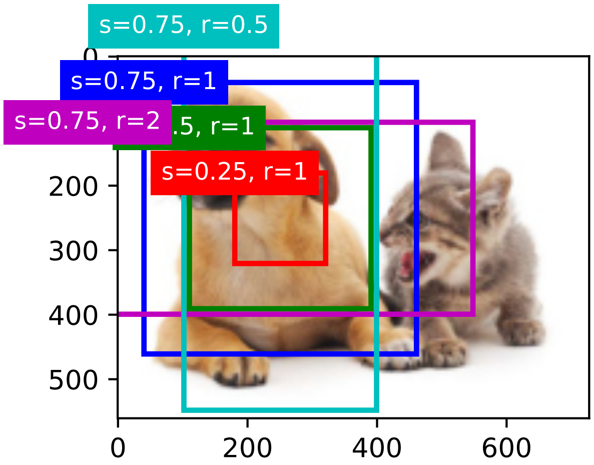
```
array([0.06, 0.07, 0.63, 0.82])
```

In order to describe all anchor boxes centered on one pixel in the image, we first define the `show_bboxes` function to draw multiple bounding boxes on the image.

```
#@save
def show_bboxes(axes, bboxes, labels=None, colors=None):
    """Show bounding boxes."""
    def _make_list(obj, default_values=None):
        if obj is None:
            obj = default_values
        elif not isinstance(obj, (list, tuple)):
            obj = [obj]
        return obj
    labels = _make_list(labels)
    colors = _make_list(colors, ['b', 'g', 'r', 'm', 'c'])
    for i, bbox in enumerate(bboxes):
        color = colors[i % len(colors)]
        rect = d2l.bbox_to_rect(bbox.asnumpy(), color)
        axes.add_patch(rect)
        if labels and len(labels) > i:
            text_color = 'k' if color == 'w' else 'w'
            axes.text(rect.xy[0], rect.xy[1], labels[i],
                      va='center', ha='center', fontsize=9, color=text_color,
                      bbox=dict(facecolor=color, lw=0))
```

As we just saw, the coordinate values of the x and y axis in the variable boxes have been divided by the width and height of the image, respectively. When drawing images, we need to restore the original coordinate values of the anchor boxes and therefore define the variable bbox_scale. Now, we can draw all the anchor boxes centered on (250, 250) in the image. As you can see, the blue anchor box with a size of 0.75 and an aspect ratio of 1 covers the dog in the image well.

```
d2l.set_figsize()
bbox_scale = np.array((w, h, w, h))
fig = d2l.plt.imshow(img)
show_bboxes(fig.axes, boxes[250, 250, :, :] * bbox_scale,
            ['s=0.75, r=1', 's=0.5, r=1', 's=0.25, r=1', 's=0.75, r=2',
             's=0.75, r=0.5'])
```



13.4.2 Intersection over Union

We just mentioned that the anchor box covers the dog in the image well. If the ground-truth bounding box of the target is known, how can "well" here be quantified? An intuitive method is to measure the similarity between anchor boxes and the ground-truth bounding box. We know that the Jaccard index can measure the similarity between two sets. Given sets $\mathcal{A}$ and $\mathcal{B}$, their Jaccard index is the size of their intersection divided by the size of their union:

$$J(\mathcal{A},\mathcal{B}) = \frac{\left|\mathcal{A} \cap \mathcal{B}\right|}{\left| \mathcal{A} \cup \mathcal{B}\right|}. \tag{13.4.2}$$

In fact, we can consider the pixel area of a bounding box as a collection of pixels. In this way, we can measure the similarity of the two bounding boxes by the Jaccard index of their pixel sets. When we measure the similarity of two bounding boxes, we usually refer the Jaccard index as intersection over union (IoU), which is the ratio of the intersecting area to the union area of the two bounding boxes, as shown in Fig. 13.4.1. The value range of IoU is between 0 and 1: 0 means that there are no overlapping pixels between the two bounding boxes, while 1 indicates that the two bounding boxes are equal.

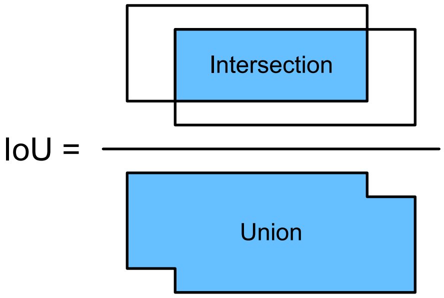


Fig. 13.4.1: IoU is the ratio of the intersecting area to the union area of two bounding boxes.

For the remainder of this section, we will use IoU to measure the similarity between anchor boxes and ground-truth bounding boxes, and between different anchor boxes.

13.4.3 Labeling Training Set Anchor Boxes

In the training set, we consider each anchor box as a training example. In order to train the object detection model, we need to mark two types of labels for each anchor box: first, the category of the target contained in the anchor box (category) and, second, the offset of the ground-truth bounding box relative to the anchor box (offset). In object detection, we first generate multiple anchor boxes, predict the categories and offsets for each anchor box, adjust the anchor box position according to the predicted offset to obtain the bounding boxes to be used for prediction, and finally filter out the prediction bounding boxes that need to be output.

We know that, in the object detection training set, each image is labelled with the location of the ground-truth bounding box and the category of the target contained. After the anchor boxes are generated, we primarily label anchor boxes based on the location and category information of the ground-truth bounding boxes similar to the anchor boxes. So how do we assign ground-truth bounding boxes to anchor boxes similar to them?

Assume that the anchor boxes in the image are $A_1, A_2, \ldots, A_{n_a}$ and the ground-truth bounding boxes are $B_1, B_2, \ldots, B_{n_b}$ and $n_a \geq n_b$. Define matrix $\mathbf{X} \in \mathbb{R}^{n_a \times n_b}$, where element x_{ij} in the i^th row and j^{th} column is the IoU of the anchor box A_i to the ground-truth bounding box B_j. First, we find the largest element in the matrix **X** and record the row index and column index of the element as i_1, j_1. We assign the ground-truth bounding box B_{j_1} to the anchor box A_{i_1}. Obviously, anchor box A_{i_1} and ground-truth bounding box B_{j_1} have the highest similarity among all the "anchor box–ground-truth bounding box" pairings. Next, discard all elements in the i_1th row and the j_1th column in the matrix **X**. Find the largest remaining element in the matrix **X** and record the row index and column index of the element as i_2, j_2. We assign ground-truth bounding box B_{j_2} to anchor box A_{i_2} and then discard all elements in the i_2th row and the j_2th column in the matrix **X**. At this point, elements in two rows and two columns in the matrix **X** have been discarded.

We proceed until all elements in the n_b column in the matrix **X** are discarded. At this time, we have assigned a ground-truth bounding box to each of the n_b anchor boxes. Next, we only traverse the remaining $n_a - n_b$ anchor boxes. Given anchor box A_i, find the bounding box B_j with the largest IoU with A_i according to the i^{th} row of the matrix **X**, and only assign ground-truth bounding box B_j to anchor box A_i when the IoU is greater than the predetermined threshold.

As shown in Fig. 13.4.2 (left), assuming that the maximum value in the matrix **X** is x_{23}, we will assign ground-truth bounding box B_3 to anchor box A_2. Then, we discard all the elements in row 2 and column 3 of the matrix, find the largest element x_{71} of the remaining shaded area, and assign ground-truth bounding box B_1 to anchor box A_7. Then, as shown in :numref:`fig_anchor_label` (middle), discard all the elements in row 7 and column 1 of the matrix, find the largest element x_{54} of the remaining shaded area, and assign ground-truth bounding box B_4 to anchor box A_5. Finally, as shown in :numref:`fig_anchor_label` (right), discard all the elements in row 5 and column 4 of the matrix, find the largest element x_{92} of the remaining shaded area, and assign ground-truth bounding box B_2 to anchor box A_9. After that, we only need to traverse the remaining anchor boxes of A_1, A_3, A_4, A_6, A_8 and determine whether to assign ground-truth bounding boxes to the remaining anchor boxes according to the threshold.

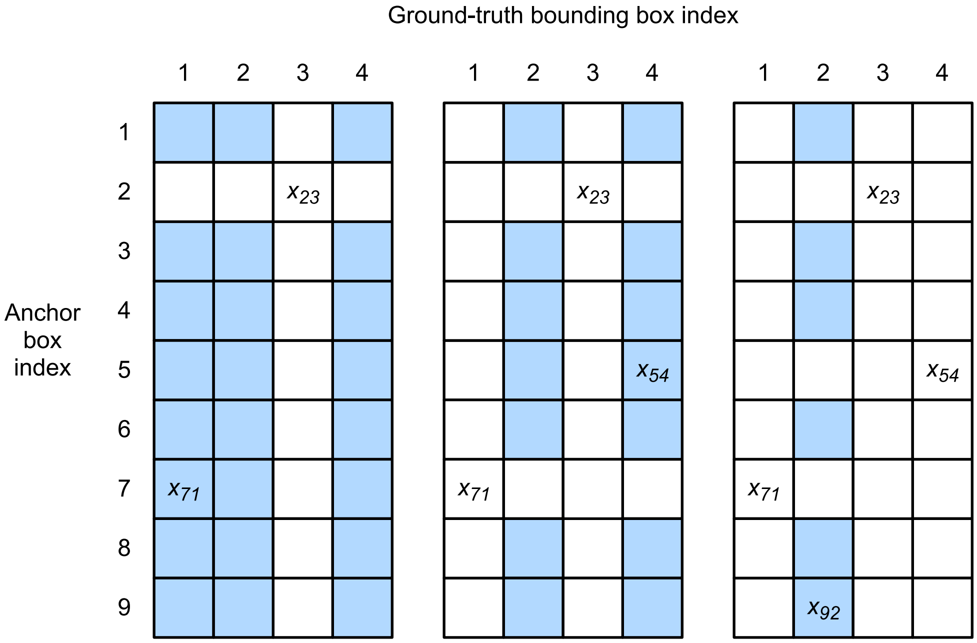


Fig. 13.4.2: Assign ground-truth bounding boxes to anchor boxes.

Now we can label the categories and offsets of the anchor boxes. If an anchor box A is assigned ground-truth bounding box B, the category of the anchor box A is set to the category of B. And the offset of the anchor box A is set according to the relative position of the central coordinates of

B and A and the relative sizes of the two boxes. Because the positions and sizes of various boxes in the dataset may vary, these relative positions and relative sizes usually require some special transformations to make the offset distribution more uniform and easier to fit. Assume the center coordinates of anchor box A and its assigned ground-truth bounding box B are $(x_a, y_a), (x_b, y_b)$, the widths of A and B are w_a, w_b, and their heights are h_a, h_b, respectively. In this case, a common technique is to label the offset of A as

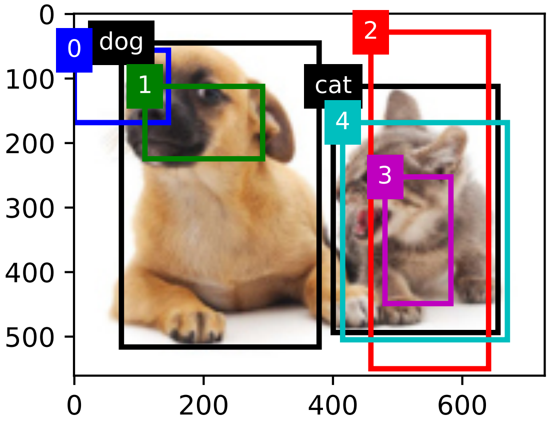
$$\left(\frac{\frac{x_b - x_a}{w_a} - \mu_x}{\sigma_x}, \frac{\frac{y_b - y_a}{h_a} - \mu_y}{\sigma_y}, \frac{\log \frac{w_b}{w_a} - \mu_w}{\sigma_w}, \frac{\log \frac{h_b}{h_a} - \mu_h}{\sigma_h} \right), \tag{13.4.3}$$

The default values of the constant are $\mu_x = \mu_y = \mu_w = \mu_h = 0, \sigma_x = \sigma_y = 0.1, and \sigma_w = \sigma_h = 0.2.$ If an anchor box is not assigned a ground-truth bounding box, we only need to set the category of the anchor box to background. Anchor boxes whose category is background are often referred to as negative anchor boxes, and the rest are referred to as positive anchor boxes.

Below we demonstrate a detailed example. We define ground-truth bounding boxes for the cat and dog in the read image, where the first element is category (0 for dog, 1 for cat) and the remaining four elements are the x, y axis coordinates at top-left corner and x, y axis coordinates at lower-right corner (the value range is between 0 and 1). Here, we construct five anchor boxes to be labeled by the coordinates of the upper-left corner and the lower-right corner, which are recorded as $A_0, \ldots, A_4$, respectively (the index in the program starts from 0). First, draw the positions of these anchor boxes and the ground-truth bounding boxes in the image.

```
ground_truth = np.array([[0, 0.1, 0.08, 0.52, 0.92],
                         [1, 0.55, 0.2, 0.9, 0.88]])
anchors = np.array([[0, 0.1, 0.2, 0.3], [0.15, 0.2, 0.4, 0.4],
                    [0.63, 0.05, 0.88, 0.98], [0.66, 0.45, 0.8, 0.8],
                    [0.57, 0.3, 0.92, 0.9]])

fig = d2l.plt.imshow(img)
show_bboxes(fig.axes, ground_truth[:, 1:] * bbox_scale, ['dog', 'cat'], 'k')
show_bboxes(fig.axes, anchors * bbox_scale, ['0', '1', '2', '3', '4']);
```



We can label categories and offsets for anchor boxes by using the `multibox_target` function. This function sets the background category to 0 and increments the integer index of the target category from zero by 1 (1 for dog and 2 for cat). We add example dimensions to the anchor boxes and ground-truth bounding boxes and construct random predicted results with a shape of (batch size, number of categories including background, number of anchor boxes) by using the `expand_dims` function.

```
labels = npx.multibox_target(np.expand_dims(anchors, axis=0),
                             np.expand_dims(ground_truth, axis=0),
                             np.zeros((1, 3, 5)))
```

There are three items in the returned result, all of which are in the tensor format. The third item is represented by the category labeled for the anchor box.

```
labels[2]
```

```
array([[0., 1., 2., 0., 2.]])
```

We analyze these labelled categories based on positions of anchor boxes and ground-truth bounding boxes in the image. First, in all "anchor box–ground-truth bounding box" pairs, the IoU of anchor box A_4 to the ground-truth bounding box of the cat is the largest, so the category of anchor box A_4 is labeled as cat. Without considering anchor box A_4 or the ground-truth bounding box of the cat, in the remaining "anchor box–ground-truth bounding box" pairs, the pair with the largest IoU is anchor box A_1 and the ground-truth bounding box of the dog, so the category of anchor box A_1 is labeled as dog. Next, traverse the remaining three unlabeled anchor boxes. The category of the ground-truth bounding box with the largest IoU with anchor box A_0 is dog, but the IoU is smaller than the threshold (the default is 0.5), so the category is labeled as background; the category of the ground-truth bounding box with the largest IoU with anchor box A_2 is cat and the IoU is greater than the threshold, so the category is labeled as cat; the category of the ground-truth bounding box with the largest IoU with anchor box A_3 is cat, but the IoU is smaller than the threshold, so the category is labeled as background.

The second item of the return value is a mask variable, with the shape of (batch size, four times the number of anchor boxes). The elements in the mask variable correspond one-to-one with the four offset values of each anchor box. Because we do not care about background detection, offsets of the negative class should not affect the target function. By multiplying by element, the 0 in the mask variable can filter out negative class offsets before calculating target function.

```
labels[1]
```

```
array([[0., 0., 0., 0., 1., 1., 1., 1., 1., 1., 1., 1., 0., 0., 0., 0.,
        1., 1., 1., 1.]])
```

The first item returned is the four offset values labeled for each anchor box, with the offsets of negative class anchor boxes labeled as 0.

```
labels[0]
```

```
array([[ 0.00e+00,  0.00e+00,  0.00e+00,  0.00e+00,  1.40e+00,  1.00e+01,
         2.59e+00,  7.18e+00, -1.20e+00,  2.69e-01,  1.68e+00, -1.57e+00,
         0.00e+00,  0.00e+00,  0.00e+00,  0.00e+00, -5.71e-01, -1.00e+00,
        -8.94e-07,  6.26e-01]])
```

13.4.4 Bounding Boxes for Prediction

During model prediction phase, we first generate multiple anchor boxes for the image and then predict categories and offsets for these anchor boxes one by one. Then, we obtain prediction bounding boxes based on anchor boxes and their predicted offsets. When there are many anchor boxes, many similar prediction bounding boxes may be output for the same target. To simplify the results, we can remove similar prediction bounding boxes. A commonly used method is called non-maximum suppression (NMS).

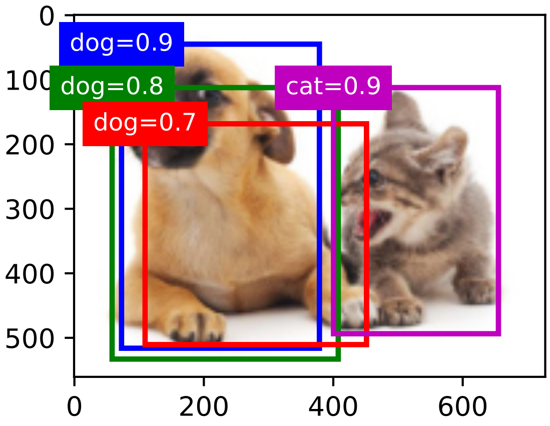
Let us take a look at how NMS works. For a prediction bounding box B, the model calculates the predicted probability for each category. Assume the largest predicted probability is p, the category corresponding to this probability is the predicted category of B. We also refer to p as the confidence level of prediction bounding box B. On the same image, we sort the prediction bounding boxes with predicted categories other than background by confidence level from high to low, and obtain the list L. Select the prediction bounding box B_1 with highest confidence level from L as a baseline and remove all non-benchmark prediction bounding boxes with an IoU with B_1 greater than a certain threshold from L. The threshold here is a preset hyperparameter. At this point, L retains the prediction bounding box with the highest confidence level and removes other prediction bounding boxes similar to it. Next, select the prediction bounding box B_2 with the second highest confidence level from L as a baseline, and remove all non-benchmark prediction bounding boxes with an IoU with B_2 greater than a certain threshold from L. Repeat this process until all prediction bounding boxes in L have been used as a baseline. At this time, the IoU of any pair of prediction bounding boxes in L is less than the threshold. Finally, output all prediction bounding boxes in the list L.

Next, we will look at a detailed example. First, construct four anchor boxes. For the sake of simplicity, we assume that predicted offsets are all 0. This means that the prediction bounding boxes are anchor boxes. Finally, we construct a predicted probability for each category.

```
anchors = np.array([[0.1, 0.08, 0.52, 0.92], [0.08, 0.2, 0.56, 0.95],
                    [0.15, 0.3, 0.62, 0.91], [0.55, 0.2, 0.9, 0.88]])
offset_preds = np.array([0] * anchors.size)
cls_probs = np.array([[0] * 4,  # Predicted probability for background
                      [0.9, 0.8, 0.7, 0.1],  # Predicted probability for dog
                      [0.1, 0.2, 0.3, 0.9]])  # Predicted probability for cat
```

Print prediction bounding boxes and their confidence levels on the image.

```
fig = d2l.plt.imshow(img)
show_bboxes(fig.axes, anchors * bbox_scale,
            ['dog=0.9', 'dog=0.8', 'dog=0.7', 'cat=0.9'])
```



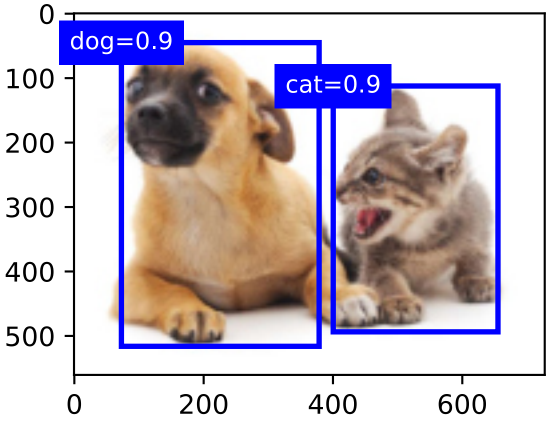
We use the `multibox_detection` function to perform NMS and set the threshold to 0.5. This adds an example dimension to the tensor input. We can see that the shape of the returned result is (batch size, number of anchor boxes, 6). The 6 elements of each row represent the output information for the same prediction bounding box. The first element is the predicted category index, which starts from 0 (0 is dog, 1 is cat). The value -1 indicates background or removal in NMS. The second element is the confidence level of prediction bounding box. The remaining four elements are the x, y axis coordinates of the upper-left corner and the x, y axis coordinates of the lower-right corner of the prediction bounding box (the value range is between 0 and 1).

```
output = npx.multibox_detection(
    np.expand_dims(cls_probs, axis=0),
    np.expand_dims(offset_preds, axis=0),
    np.expand_dims(anchors, axis=0),
    nms_threshold=0.5)
output
```

```
array([[[ 0.  ,  0.9 ,  0.1 ,  0.08,  0.52,  0.92],
        [ 1.  ,  0.9 ,  0.55,  0.2 ,  0.9 ,  0.88],
        [-1.  ,  0.8 ,  0.08,  0.2 ,  0.56,  0.95],
        [-1.  ,  0.7 ,  0.15,  0.3 ,  0.62,  0.91]]])
```

We remove the prediction bounding boxes of category -1 and visualize the results retained by NMS.

```
fig = d2l.plt.imshow(img)
for i in output[0].asnumpy():
    if i[0] == -1:
        continue
    label = ('dog=', 'cat=')[int(i[0])] + str(i[1])
    show_bboxes(fig.axes, [np.array(i[2:]) * bbox_scale], label)
```



In practice, we can remove prediction bounding boxes with lower confidence levels before performing NMS, thereby reducing the amount of computation for NMS. We can also filter the output of NMS, for example, by only retaining results with higher confidence levels as the final output.

Summary

- We generate multiple anchor boxes with different sizes and aspect ratios, centered on each pixel.
- IoU, also called Jaccard index, measures the similarity of two bounding boxes. It is the ratio of the intersecting area to the union area of two bounding boxes.
- In the training set, we mark two types of labels for each anchor box: one is the category of the target contained in the anchor box and the other is the offset of the ground-truth bounding box relative to the anchor box.
- When predicting, we can use non-maximum suppression (NMS) to remove similar prediction bounding boxes, thereby simplifying the results.

Exercises

1. Change the `sizes` and `ratios` values in the `multibox_prior` function and observe the changes to the generated anchor boxes.
2. Construct two bounding boxes with and IoU of 0.5, and observe their coincidence.
3. Verify the output of offset `labels[0]` by marking the anchor box offsets as defined in this section (the constant is the default value).
4. Modify the variable `anchors` in the "Labeling Training Set Anchor Boxes" and "Output Bounding Boxes for Prediction" sections. How do the results change?

Discussions[190]

[190] https://discuss.d2l.ai/t/370

13.5 Multiscale Object Detection

In Section 13.4, we generated multiple anchor boxes centered on each pixel of the input image. These anchor boxes are used to sample different regions of the input image. However, if anchor boxes are generated centered on each pixel of the image, soon there will be too many anchor boxes for us to compute. For example, we assume that the input image has a height and a width of 561 and 728 pixels respectively. If five different shapes of anchor boxes are generated centered on each pixel, over two million anchor boxes ($561 \times 728 \times 5$) need to be predicted and labeled on the image.

It is not difficult to reduce the number of anchor boxes. An easy way is to apply uniform sampling on a small portion of pixels from the input image and generate anchor boxes centered on the sampled pixels. In addition, we can generate anchor boxes of varied numbers and sizes on multiple scales. Notice that smaller objects are more likely to be positioned on the image than larger ones. Here, we will use a simple example: Objects with shapes of 1×1, 1×2, and 2×2 may have 4, 2, and 1 possible position(s) on an image with the shape 2×2. Therefore, when using smaller anchor boxes to detect smaller objects, we can sample more regions; when using larger anchor boxes to detect larger objects, we can sample fewer regions.

To demonstrate how to generate anchor boxes on multiple scales, let us read an image first. It has a height and width of 561×728 pixels.

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import image, np, npx

npx.set_np()

img = image.imread('../img/catdog.jpg')
h, w = img.shape[0:2]
h, w
```

```
(561, 728)
```

In Section 6.2, the 2D array output of the convolutional neural network (CNN) is called a feature map. We can determine the midpoints of anchor boxes uniformly sampled on any image by defining the shape of the feature map.

The function `display_anchors` is defined below. We are going to generate anchor boxes `anchors` centered on each unit (pixel) on the feature map `fmap`. Since the coordinates of axes x and y in anchor boxes `anchors` have been divided by the width and height of the feature map `fmap`, values between 0 and 1 can be used to represent relative positions of anchor boxes in the feature map. Since the midpoints of anchor boxes `anchors` overlap with all the units on feature map `fmap`, the relative spatial positions of the midpoints of the `anchors` on any image must have a uniform distribution. Specifically, when the width and height of the feature map are set to `fmap_w` and `fmap_h` respectively, the function will conduct uniform sampling for `fmap_h` rows and `fmap_w` columns of pixels and use them as midpoints to generate anchor boxes with size `s` (we assume that the length of list `s` is 1) and different aspect ratios (`ratios`).

```
def display_anchors(fmap_w, fmap_h, s):
    d2l.set_figsize()
    # The values from the first two dimensions will not affect the output
```

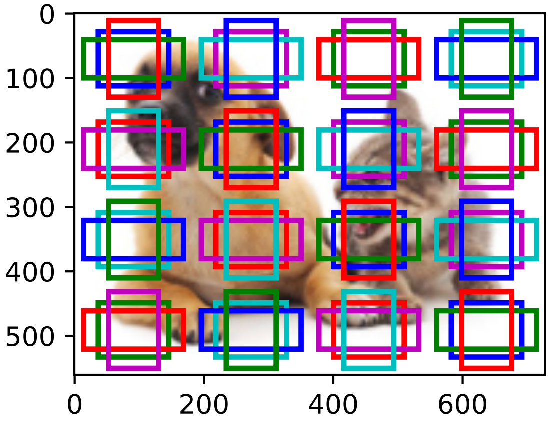
(continues on next page)

(continued from previous page)

```
    fmap = np.zeros((1, 10, fmap_w, fmap_h))
    anchors = npx.multibox_prior(fmap, sizes=s, ratios=[1, 2, 0.5])
    bbox_scale = np.array((w, h, w, h))
    d2l.show_bboxes(d2l.plt.imshow(img.asnumpy()).axes,
                    anchors[0] * bbox_scale)
```

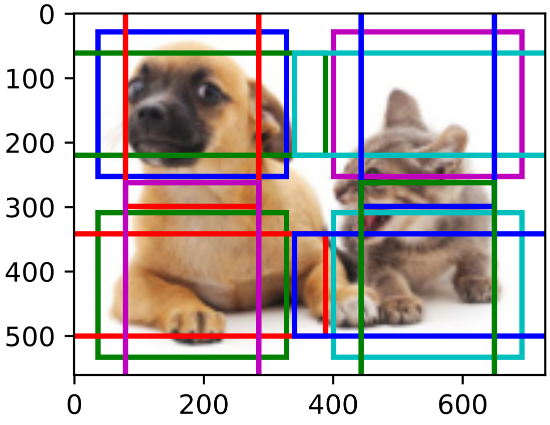
We will first focus on the detection of small objects. In order to make it easier to distinguish upon display, the anchor boxes with different midpoints here do not overlap. We assume that the size of the anchor boxes is 0.15 and the height and width of the feature map are 4. We can see that the midpoints of anchor boxes from the 4 rows and 4 columns on the image are uniformly distributed.

```
display_anchors(fmap_w=4, fmap_h=4, s=[0.15])
```



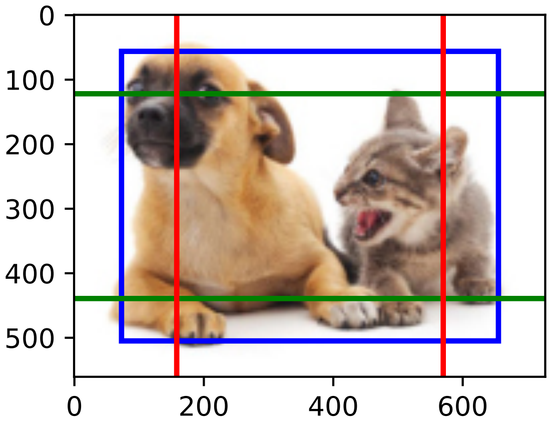
We are going to reduce the height and width of the feature map by half and use a larger anchor box to detect larger objects. When the size is set to 0.4, overlaps will occur between regions of some anchor boxes.

```
display_anchors(fmap_w=2, fmap_h=2, s=[0.4])
```



Finally, we are going to reduce the height and width of the feature map by half and increase the anchor box size to 0.8. Now the midpoint of the anchor box is the center of the image.

```
display_anchors(fmap_w=1, fmap_h=1, s=[0.8])
```



Since we have generated anchor boxes of different sizes on multiple scales, we will use them to detect objects of various sizes at different scales. Now we are going to introduce a method based on convolutional neural networks (CNNs).

At a certain scale, suppose we generate $h \times w$ sets of anchor boxes with different midpoints based on c_i feature maps with the shape $h \times w$ and the number of anchor boxes in each set is a. For example, for the first scale of the experiment, we generate 16 sets of anchor boxes with different midpoints based on 10 (number of channels) feature maps with a shape of 4×4, and each set contains 3 anchor boxes. Next, each anchor box is labeled with a category and offset based on the classification and position of the ground-truth bounding box. At the current scale, the object detection model needs to predict the category and offset of $h \times w$ sets of anchor boxes with different midpoints based on the input image.

We assume that the c_i feature maps are the intermediate output of the CNN based on the input image. Since each feature map has $h \times w$ different spatial positions, the same position will have c_i units. According to the definition of receptive field in the Section 6.2, the c_i units of the feature map at the same spatial position have the same receptive field on the input image. Thus, they represent the information of the input image in this same receptive field. Therefore, we can transform the c_i units of the feature map at the same spatial position into the categories and offsets of the a anchor boxes generated using that position as a midpoint. It is not hard to see that, in essence, we use the information of the input image in a certain receptive field to predict the category and offset of the anchor boxes close to the field on the input image.

When the feature maps of different layers have receptive fields of different sizes on the input image, they are used to detect objects of different sizes. For example, we can design a network to have a wider receptive field for each unit in the feature map that is closer to the output layer, to detect objects with larger sizes in the input image.

We will implement a multiscale object detection model in the following section.

Summary

- We can generate anchor boxes with different numbers and sizes on multiple scales to detect objects of different sizes on multiple scales.
- The shape of the feature map can be used to determine the midpoint of the anchor boxes that uniformly sample any image.
- We use the information for the input image from a certain receptive field to predict the category and offset of the anchor boxes close to that field on the image.

Exercises

1. Given an input image, assume $1 \times c_i \times h \times w$ to be the shape of the feature map while c_i, h, w are the number, height, and width of the feature map. What methods can you think of to convert this variable into the anchor box's category and offset? What is the shape of the output?

Discussions[191]

13.6 The Object Detection Dataset

There are no small datasets, like MNIST or Fashion-MNIST, in the object detection field. In order to quickly test models, we are going to assemble a small dataset. First, we generate 1000 banana images of different angles and sizes using free bananas from our office. Then, we collect a series of background images and place a banana image at a random position on each image. We use the im2rec tool[192] provided by MXNet to convert the images to binary RecordIO format[1]. This format can reduce the storage overhead of the dataset on the disk and improve the reading efficiency. If you want to learn more about how to read images, refer to the documentation for the GluonCV Toolkit[193].

13.6.1 Downloading the Dataset

The banana detection dataset in RecordIO format can be downloaded directly from the Internet.

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import gluon, image, np, npx
import os

npx.set_np()

#@save
d2l.DATA_HUB['bananas'] = (d2l.DATA_URL + 'bananas.zip',
                           'aadfd1c4c5d7178616799dd1801c9a234ccdaf19')
```

[191] https://discuss.d2l.ai/t/371
[192] https://github.com/apache/incubator-mxnet/blob/master/tools/im2rec.py
[193] https://gluon-cv.mxnet.io/

13.6.2 Reading the Dataset

We are going to read the object detection dataset by creating the instance ImageDetIter. The "Det" in the name refers to Detection. We will read the training dataset in random order. Since the format of the dataset is RecordIO, we need the image index file 'train.idx' to read random minibatches. In addition, for each image of the training set, we will use random cropping and require the cropped image to cover at least 95% of each object. Since the cropping is random, this requirement is not always satisfied. We preset the maximum number of random cropping attempts to 200. If none of them meets the requirement, the image will not be cropped. To ensure the certainty of the output, we will not randomly crop the images in the test dataset. We also do not need to read the test dataset in random order.

```
#@save
def load_data_bananas(batch_size, edge_size=256):
    """Load the bananas dataset."""
    data_dir = d2l.download_extract('bananas')
    train_iter = image.ImageDetIter(
        path_imgrec=os.path.join(data_dir, 'train.rec'),
        path_imgidx=os.path.join(data_dir, 'train.idx'),
        batch_size=batch_size,
        data_shape=(3, edge_size, edge_size),  # The shape of the output image
        shuffle=True,  # Read the dataset in random order
        rand_crop=1,  # The probability of random cropping is 1
        min_object_covered=0.95, max_attempts=200)
    val_iter = image.ImageDetIter(
        path_imgrec=os.path.join(data_dir, 'val.rec'), batch_size=batch_size,
        data_shape=(3, edge_size, edge_size), shuffle=False)
    return train_iter, val_iter
```

Below, we read a minibatch and print the shape of the image and label. The shape of the image is the same as in the previous experiment (batch size, number of channels, height, width). The shape of the label is (batch size, m, 5), where m is equal to the maximum number of bounding boxes contained in a single image in the dataset. Although computation for the minibatch is very efficient, it requires each image to contain the same number of bounding boxes so that they can be placed in the same batch. Since each image may have a different number of bounding boxes, we can add illegal bounding boxes to images that have less than m bounding boxes until each image contains m bounding boxes. Thus, we can read a minibatch of images each time. The label of each bounding box in the image is represented by an array of length 5. The first element in the array is the category of the object contained in the bounding box. When the value is -1, the bounding box is an illegal bounding box for filling purpose. The remaining four elements of the array represent the x, y axis coordinates of the upper-left corner of the bounding box and the x, y axis coordinates of the lower-right corner of the bounding box (the value range is between 0 and 1). The banana dataset here has only one bounding box per image, so $m = 1$.

```
batch_size, edge_size = 32, 256
train_iter, _ = load_data_bananas(batch_size, edge_size)
batch = train_iter.next()
batch.data[0].shape, batch.label[0].shape
```

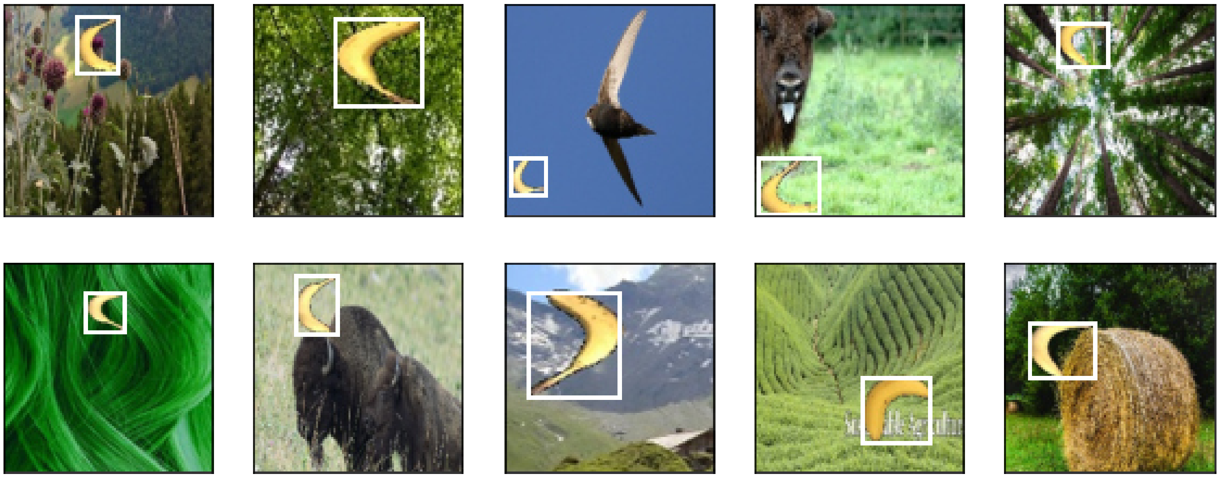
```
Downloading ../data/bananas.zip from http://d2l-data.s3-accelerate.amazonaws.com/bananas.zip.
↪..
```

```
((32, 3, 256, 256), (32, 1, 5))
```

13.6.3 Demonstration

We have ten images with bounding boxes on them. We can see that the angle, size, and position of banana are different in each image. Of course, this is a simple artificial dataset. In actual practice, the data are usually much more complicated.

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



Summary

- The banana detection dataset we synthesized can be used to test object detection models.
- The data reading for object detection is similar to that for image classification. However, after we introduce bounding boxes, the label shape and image augmentation (e.g., random cropping) are changed.

Exercises

1. Referring to the MXNet documentation, what are the parameters for the constructors of the `image.ImageDetIter` and `image.CreateDetAugmenter` classes? What is their significance?

Discussions[194]

[194] https://discuss.d2l.ai/t/372

13.7 Single Shot Multibox Detection (SSD)

In the previous few sections, we have introduced bounding boxes, anchor boxes, multiscale object detection, and datasets. Now, we will use this background knowledge to construct an object detection model: single shot multibox detection (SSD) (Liu et al., 2016). This quick and easy model is already widely used. Some of the design concepts and implementation details of this model are also applicable to other object detection models.

13.7.1 Model

Fig. 13.7.1 shows the design of an SSD model. The model's main components are a base network block and several multiscale feature blocks connected in a series. Here, the base network block is used to extract features of original images, and it generally takes the form of a deep convolutional neural network. The paper on SSDs chooses to place a truncated VGG before the classification layer (Liu et al., 2016), but this is now commonly replaced by ResNet. We can design the base network so that it outputs larger heights and widths. In this way, more anchor boxes are generated based on this feature map, allowing us to detect smaller objects. Next, each multiscale feature block reduces the height and width of the feature map provided by the previous layer (for example, it may reduce the sizes by half). The blocks then use each element in the feature map to expand the receptive field on the input image. In this way, the closer a multiscale feature block is to the top of Fig. 13.7.1 the smaller its output feature map, and the fewer the anchor boxes that are generated based on the feature map. In addition, the closer a feature block is to the top, the larger the receptive field of each element in the feature map and the better suited it is to detect larger objects. As the SSD generates different numbers of anchor boxes of different sizes based on the base network block and each multiscale feature block and then predicts the categories and offsets (i.e., predicted bounding boxes) of the anchor boxes in order to detect objects of different sizes, SSD is a multiscale object detection model.

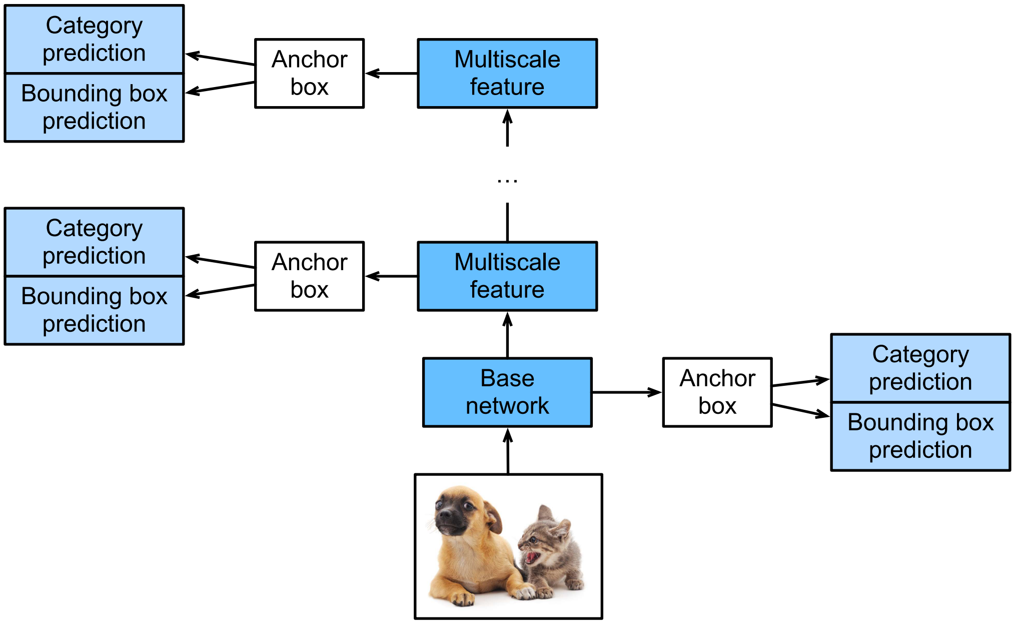


Fig. 13.7.1: The SSD is composed of a base network block and several multiscale feature blocks connected in a series.

Next, we will describe the implementation of the modules in Fig. 13.7.1. First, we need to discuss the implementation of category prediction and bounding box prediction.

Category Prediction Layer

Set the number of object categories to q. In this case, the number of anchor box categories is $q + 1$, with 0 indicating an anchor box that only contains background. For a certain scale, set the height and width of the feature map to h and w, respectively. If we use each element as the center to generate a anchor boxes, we need to classify a total of hwa anchor boxes. If we use a fully connected layer (FCN) for the output, this will likely result in an excessive number of model parameters. Recall how we used convolutional layer channels to output category predictions in Section 7.3. SSD uses the same method to reduce the model complexity.

Specifically, the category prediction layer uses a convolutional layer that maintains the input height and width. Thus, the output and input have a one-to-one correspondence to the spatial coordinates along the width and height of the feature map. Assuming that the output and input have the same spatial coordinates (x, y), the channel for the coordinates (x, y) on the output feature map contains the category predictions for all anchor boxes generated using the input feature map coordinates (x, y) as the center. Therefore, there are $a(q + 1)$ output channels, with the output channels indexed as $i(q+1)+j$ $(0 \leq j \leq q)$ representing the predictions of the category index j for the anchor box index i.

Now, we will define a category prediction layer of this type. After we specify the parameters a and q, it uses a 3×3 convolutional layer with a padding of 1. The heights and widths of the input and output of this convolutional layer remain unchanged.

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import autograd, gluon, image, init, np, npx
from mxnet.gluon import nn

npx.set_np()

def cls_predictor(num_anchors, num_classes):
    return nn.Conv2D(num_anchors * (num_classes + 1), kernel_size=3,
                     padding=1)
```

Bounding Box Prediction Layer

The design of the bounding box prediction layer is similar to that of the category prediction layer. The only difference is that, here, we need to predict 4 offsets for each anchor box, rather than $q+1$ categories.

```
def bbox_predictor(num_anchors):
    return nn.Conv2D(num_anchors * 4, kernel_size=3, padding=1)
```

Concatenating Predictions for Multiple Scales

As we mentioned, SSD uses feature maps based on multiple scales to generate anchor boxes and predict their categories and offsets. Because the shapes and number of anchor boxes centered on the same element differ for the feature maps of different scales, the prediction outputs at different scales may have different shapes.

In the following example, we use the same batch of data to construct feature maps of two different scales, `Y1` and `Y2`. Here, `Y2` has half the height and half the width of `Y1`. Using category prediction as an example, we assume that each element in the `Y1` and `Y2` feature maps generates five (Y1) or three (Y2) anchor boxes. When there are 10 object categories, the number of category prediction output channels is either $5 \times (10 + 1) = 55$ or $3 \times (10 + 1) = 33$. The format of the prediction output is (batch size, number of channels, height, width). As you can see, except for the batch size, the sizes of the other dimensions are different. Therefore, we must transform them into a consistent format and concatenate the predictions of the multiple scales to facilitate subsequent computation.

```
def forward(x, block):
    block.initialize()
    return block(x)

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

```
((2, 55, 20, 20), (2, 33, 10, 10))
```

The channel dimension contains the predictions for all anchor boxes with the same center. We first move the channel dimension to the final dimension. Because the batch size is the same for all scales, we can convert the prediction results to binary format (batch size, height × width × number of channels) to facilitate subsequent concatenation on the 1st dimension.

```
def flatten_pred(pred):
    return npx.batch_flatten(pred.transpose(0, 2, 3, 1))

def concat_preds(preds):
    return np.concatenate([flatten_pred(p) for p in preds], axis=1)
```

Thus, regardless of the different shapes of `Y1` and `Y2`, we can still concatenate the prediction results for the two different scales of the same batch.

```
concat_preds([Y1, Y2]).shape
```

```
(2, 25300)
```

Height and Width Downsample Block

For multiscale object detection, we define the following `down_sample_blk` block, which reduces the height and width by 50%. This block consists of two 3×3 convolutional layers with a padding of 1 and a 2×2 maximum pooling layer with a stride of 2 connected in a series. As we know, 3×3 convolutional layers with a padding of 1 do not change the shape of feature maps. However, the subsequent pooling layer directly reduces the size of the feature map by half. Because $1 \times 2 + (3 - 1) + (3 - 1) = 6$, each element in the output feature map has a receptive field on the input feature map of the shape 6×6. As you can see, the height and width downsample block enlarges the receptive field of each element in the output feature map.

```
def down_sample_blk(num_channels):
    blk = nn.Sequential()
    for _ in range(2):
        blk.add(nn.Conv2D(num_channels, kernel_size=3, padding=1),
                nn.BatchNorm(in_channels=num_channels),
                nn.Activation('relu'))
    blk.add(nn.MaxPool2D(2))
    return blk
```

By testing forward computation in the height and width downsample block, we can see that it changes the number of input channels and halves the height and width.

```
forward(np.zeros((2, 3, 20, 20)), down_sample_blk(10)).shape
```

```
(2, 10, 10, 10)
```

Base Network Block

The base network block is used to extract features from original images. To simplify the computation, we will construct a small base network. This network consists of three height and width downsample blocks connected in a series, so it doubles the number of channels at each step. When we input an original image with the shape 256×256, the base network block outputs a feature map with the shape 32×32.

```
def base_net():
    blk = nn.Sequential()
    for num_filters in [16, 32, 64]:
        blk.add(down_sample_blk(num_filters))
    return blk

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

```
(2, 64, 32, 32)
```

The Complete Model

The SSD model contains a total of five modules. Each module outputs a feature map used to generate anchor boxes and predict the categories and offsets of these anchor boxes. The first module is the base network block, modules two to four are height and width downsample blocks, and the fifth module is a global maximum pooling layer that reduces the height and width to 1. Therefore, modules two to five are all multiscale feature blocks shown in Fig. 13.7.1.

```
def get_blk(i):
    if i == 0:
        blk = base_net()
    elif i == 4:
        blk = nn.GlobalMaxPool2D()
    else:
        blk = down_sample_blk(128)
    return blk
```

Now, we will define the forward computation process for each module. In contrast to the previously-described convolutional neural networks, this module not only returns feature map `Y` output by convolutional computation, but also the anchor boxes of the current scale generated from `Y` and their predicted categories and offsets.

```
def blk_forward(X, blk, size, ratio, cls_predictor, bbox_predictor):
    Y = blk(X)
    anchors = npx.multibox_prior(Y, sizes=size, ratios=ratio)
    cls_preds = cls_predictor(Y)
    bbox_preds = bbox_predictor(Y)
    return (Y, anchors, cls_preds, bbox_preds)
```

As we mentioned, the closer a multiscale feature block is to the top in Fig. 13.7.1, the larger the objects it detects and the larger the anchor boxes it must generate. Here, we first divide the interval from 0.2 to 1.05 into five equal parts to determine the sizes of smaller anchor boxes at different scales: 0.2, 0.37, 0.54, etc. Then, according to $\sqrt{0.2 \times 0.37} = 0.272$, $\sqrt{0.37 \times 0.54} = 0.447$, and similar formulas, we determine the sizes of larger anchor boxes at the different scales.

```
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
```

Now, we can define the complete model, `TinySSD`.

```
class TinySSD(nn.Block):
    def __init__(self, num_classes, **kwargs):
        super(TinySSD, self).__init__(**kwargs)
        self.num_classes = num_classes
        for i in range(5):
            # The assignment statement is self.blk_i = get_blk(i)
            setattr(self, f'blk_{i}', get_blk(i))
            setattr(self, f'cls_{i}', cls_predictor(num_anchors, num_classes))
            setattr(self, f'bbox_{i}', bbox_predictor(num_anchors))

    def forward(self, X):
```

(continues on next page)

(continued from previous page)

```
        anchors, cls_preds, bbox_preds = [None] * 5, [None] * 5, [None] * 5
        for i in range(5):
            # getattr(self, 'blk_%d' % i) accesses 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}'))
        # In the reshape function, 0 indicates that the batch size remains
        # unchanged
        anchors = np.concatenate(anchors, axis=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
```

We now create an SSD model instance and use it to perform forward computation on image minibatch X, which has a height and width of 256 pixels. As we verified previously, the first module outputs a feature map with the shape 32×32. Because modules two to four are height and width downsample blocks, module five is a global pooling layer, and each element in the feature map is used as the center for 4 anchor boxes, a total of $(32^2 + 16^2 + 8^2 + 4^2 + 1) \times 4 = 5444$ anchor boxes are generated for each image at the five scales.

```
net = TinySSD(num_classes=1)
net.initialize()
X = np.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: (1, 5444, 4)
output class preds: (32, 5444, 2)
output bbox preds: (32, 21776)
```

13.7.2 Training

Now, we will explain, step by step, how to train the SSD model for object detection.

Data Reading and Initialization

We read the banana detection dataset we created in the previous section.

```
batch_size = 32
train_iter, _ = d2l.load_data_bananas(batch_size)
```

There is 1 category in the banana detection dataset. After defining the module, we need to initialize the model parameters and define the optimization algorithm.

```
device, net = d2l.try_gpu(), TinySSD(num_classes=1)
net.initialize(init=init.Xavier(), ctx=device)
trainer = gluon.Trainer(net.collect_params(), 'sgd',
                        {'learning_rate': 0.2, 'wd': 5e-4})
```

Defining Loss and Evaluation Functions

Object detection is subject to two types of losses. The first is anchor box category loss. For this, we can simply reuse the cross-entropy loss function we used in image classification. The second loss is positive anchor box offset loss. Offset prediction is a normalization problem. However, here, we do not use the squared loss introduced previously. Rather, we use the L_1 norm loss, which is the absolute value of the difference between the predicted value and the ground-truth value. The mask variable bbox_masks removes negative anchor boxes and padding anchor boxes from the loss calculation. Finally, we add the anchor box category and offset losses to find the final loss function for the model.

```
cls_loss = gluon.loss.SoftmaxCrossEntropyLoss()
bbox_loss = gluon.loss.L1Loss()

def calc_loss(cls_preds, cls_labels, bbox_preds, bbox_labels, bbox_masks):
    cls = cls_loss(cls_preds, cls_labels)
    bbox = bbox_loss(bbox_preds * bbox_masks, bbox_labels * bbox_masks)
    return cls + bbox
```

We can use the accuracy rate to evaluate the classification results. As we use the L_1 norm loss, we will use the average absolute error to evaluate the bounding box prediction results.

```
def cls_eval(cls_preds, cls_labels):
    # Because the category prediction results are placed in the final
    # dimension, argmax must specify this dimension
    return float((cls_preds.argmax(axis=-1).astype(
        cls_labels.dtype) == cls_labels).sum())

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

Training the Model

During model training, we must generate multiscale anchor boxes (anchors) in the model's forward computation process and predict the category (cls_preds) and offset (bbox_preds) for each anchor box. Afterwards, we label the category (cls_labels) and offset (bbox_labels) of each generated anchor box based on the label information Y. Finally, we calculate the loss function using the predicted and labeled category and offset values. To simplify the code, we do not evaluate the training dataset here.

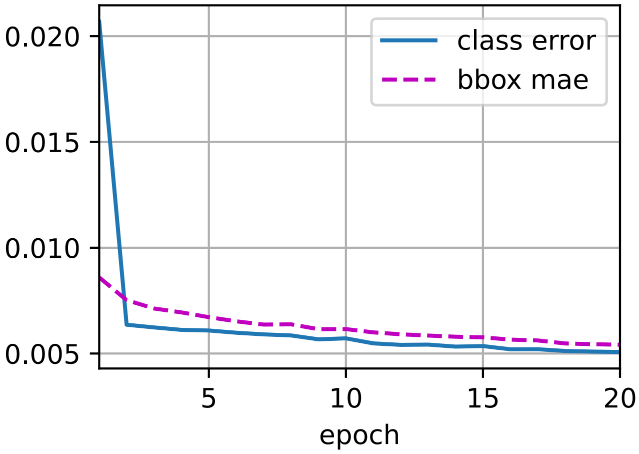
```
num_epochs, timer = 20, d2l.Timer()
animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                        legend=['class error', 'bbox mae'])
for epoch in range(num_epochs):
```

(continues on next page)

(continued from previous page)

```
    # accuracy_sum, mae_sum, num_examples, num_labels
    metric = d2l.Accumulator(4)
    train_iter.reset()  # Read data from the start.
    for batch in train_iter:
        timer.start()
        X = batch.data[0].as_in_ctx(device)
        Y = batch.label[0].as_in_ctx(device)
        with autograd.record():
            # Generate multiscale anchor boxes and predict the category and
            # offset of each
            anchors, cls_preds, bbox_preds = net(X)
            # Label the category and offset of each anchor box
            bbox_labels, bbox_masks, cls_labels = npx.multibox_target(
                anchors, Y, cls_preds.transpose(0, 2, 1))
            # Calculate the loss function using the predicted and labeled
            # category and offset values
            l = calc_loss(cls_preds, cls_labels, bbox_preds, bbox_labels,
                          bbox_masks)
        l.backward()
        trainer.step(batch_size)
        metric.add(cls_eval(cls_preds, cls_labels), cls_labels.size,
                   bbox_eval(bbox_preds, bbox_labels, bbox_masks),
                   bbox_labels.size)
    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'{train_iter.num_image/timer.stop():.1f} examples/sec on '
      f'{str(device)}')
```

```
class err 5.07e-03, bbox mae 5.42e-03
4496.9 examples/sec on gpu(0)
```



13.7.3 Prediction

In the prediction stage, we want to detect all objects of interest in the image. Below, we read the test image and transform its size. Then, we convert it to the four-dimensional format required by the convolutional layer.

```
img = image.imread('../img/banana.jpg')
feature = image.imresize(img, 256, 256).astype('float32')
X = np.expand_dims(feature.transpose(2, 0, 1), axis=0)
```

Using the `MultiBoxDetection` function, we predict the bounding boxes based on the anchor boxes and their predicted offsets. Then, we use non-maximum suppression to remove similar bounding boxes.

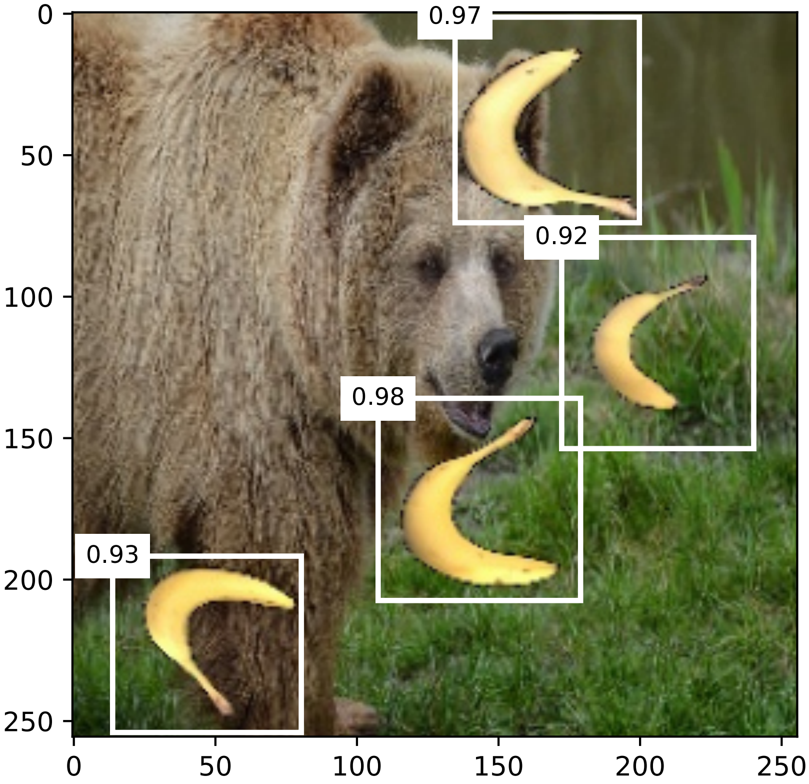
```
def predict(X):
    anchors, cls_preds, bbox_preds = net(X.as_in_ctx(device))
    cls_probs = npx.softmax(cls_preds).transpose(0, 2, 1)
    output = npx.multibox_detection(cls_probs, bbox_preds, anchors)
    idx = [i for i, row in enumerate(output[0]) if row[0] != -1]
    return output[0, idx]

output = predict(X)
```

Finally, we take all the bounding boxes with a confidence level of at least 0.3 and display them as the final output.

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

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



Summary

- SSD is a multiscale object detection model. This model generates different numbers of anchor boxes of different sizes based on the base network block and each multiscale feature block and predicts the categories and offsets of the anchor boxes to detect objects of different sizes.
- During SSD model training, the loss function is calculated using the predicted and labeled category and offset values.

Exercises

1. Due to space limitations, we have ignored some of the implementation details of SSD models in this experiment. Can you further improve the model in the following areas?

Loss Function

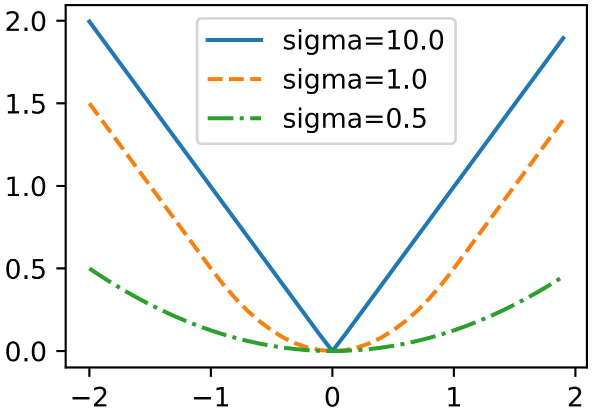
For the predicted offsets, replace L_1 norm loss with L_1 regularization loss. This loss function uses a square function around zero for greater smoothness. This is the regularized area controlled by the hyperparameter σ:

$$f(x) = \begin{cases} (\sigma x)^2/2, & \text{if } |x| < 1/\sigma^2 \\ |x| - 0.5/\sigma^2, & \text{otherwise} \end{cases} \tag{13.7.1}$$

When σ is large, this loss is similar to the L_1 norm loss. When the value is small, the loss function is smoother.

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

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



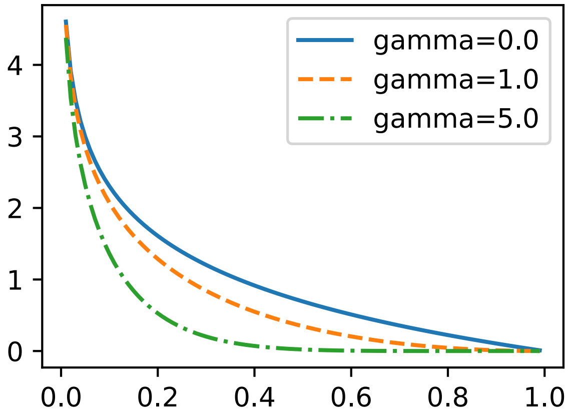
In the experiment, we used cross-entropy loss for category prediction. Now, assume that the prediction probability of the actual category j is p_j and the cross-entropy loss is $-\log p_j$. We can also use the focal loss (Lin et al., 2017). Given the positive hyperparameters γ and α, this loss is defined as:

$$-\alpha(1-p_j)^{\gamma}\log p_j. \tag{13.7.2}$$

As you can see, by increasing γ, we can effectively reduce the loss when the probability of predicting the correct category is high.

```
def focal_loss(gamma, x):
    return -(1 - x) ** gamma * np.log(x)

x = np.arange(0.01, 1, 0.01)
for l, gamma in zip(lines, [0, 1, 5]):
    y = d2l.plt.plot(x.asnumpy(), focal_loss(gamma, x).asnumpy(), l,
                     label='gamma=%.1f' % gamma)
d2l.plt.legend();
```



Training and Prediction

2. When an object is relatively large compared to the image, the model normally adopts a larger input image size.
3. This generally produces a large number of negative anchor boxes when labeling anchor box categories. We can sample the negative anchor boxes to better balance the data categories. To do this, we can set the `MultiBoxTarget` function's `negative_mining_ratio` parameter.
4. Assign hyperparameters with different weights to the anchor box category loss and positive anchor box offset loss in the loss function.
5. Refer to the SSD paper. What methods can be used to evaluate the precision of object detection models (Liu et al., 2016)?

Discussions[195]

13.8 Region-based CNNs (R-CNNs)

Region-based convolutional neural networks or regions with CNN features (R-CNNs) are a pioneering approach that applies deep models to object detection (Girshick et al., 2014). In this section, we will discuss R-CNNs and a series of improvements made to them: Fast R-CNN (Girshick, 2015), Faster R-CNN (Ren et al., 2015), and Mask R-CNN (He et al., 2017a). Due to space limitations, we will confine our discussion to the designs of these models.

13.8.1 R-CNNs

R-CNN models first select several proposed regions from an image (for example, anchor boxes are one type of selection method) and then label their categories and bounding boxes (e.g., offsets). Then, they use a CNN to perform forward computation to extract features from each proposed area. Afterwards, we use the features of each proposed region to predict their categories and bounding boxes. Fig. 13.8.1 shows an R-CNN model.

[195] https://discuss.d2l.ai/t/373

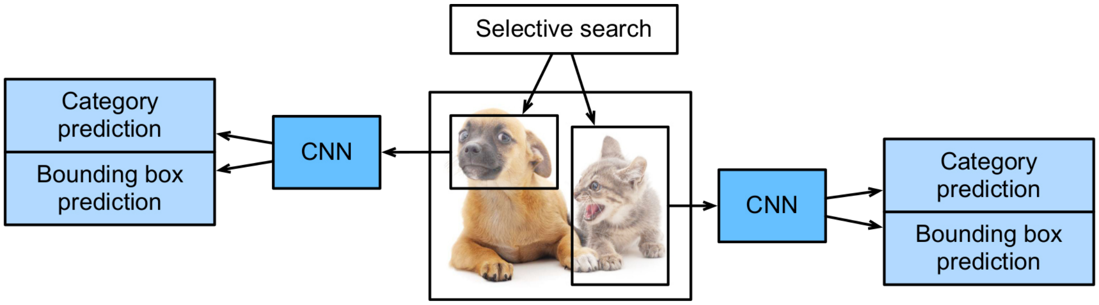


Fig. 13.8.1: R-CNN model.

Specifically, R-CNNs are composed of four main parts:

1. Selective search is performed on the input image to select multiple high-quality proposed regions (Uijlings et al., 2013). These proposed regions are generally selected on multiple scales and have different shapes and sizes. The category and ground-truth bounding box of each proposed region is labeled.
2. A pre-trained CNN is selected and placed, in truncated form, before the output layer. It transforms each proposed region into the input dimensions required by the network and uses forward computation to output the features extracted from the proposed regions.
3. The features and labeled category of each proposed region are combined as an example to train multiple support vector machines for object classification. Here, each support vector machine is used to determine whether an example belongs to a certain category.
4. The features and labeled bounding box of each proposed region are combined as an example to train a linear regression model for ground-truth bounding box prediction.

Although R-CNN models use pre-trained CNNs to effectively extract image features, the main downside is the slow speed. As you can imagine, we can select thousands of proposed regions from a single image, requiring thousands of forward computations from the CNN to perform object detection. This massive computing load means that R-CNNs are not widely used in actual applications.

13.8.2 Fast R-CNN

The main performance bottleneck of an R-CNN model is the need to independently extract features for each proposed region. As these regions have a high degree of overlap, independent feature extraction results in a high volume of repetitive computations. Fast R-CNN improves on the R-CNN by only performing CNN forward computation on the image as a whole.

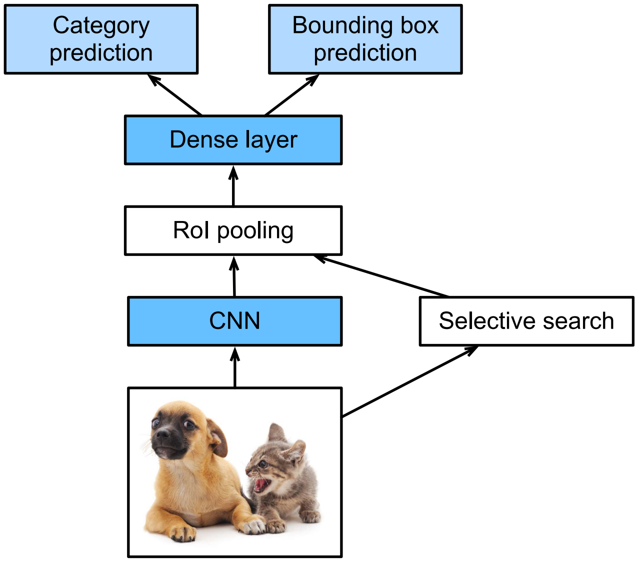


Fig. 13.8.2: Fast R-CNN model.

Fig. 13.8.2 shows a Fast R-CNN model. It is primary computation steps are described below:

1. Compared to an R-CNN model, a Fast R-CNN model uses the entire image as the CNN input for feature extraction, rather than each proposed region. Moreover, this network is generally trained to update the model parameters. As the input is an entire image, the CNN output shape is $1 \times c \times h_1 \times w_1$.
2. Assuming selective search generates n proposed regions, their different shapes indicate regions of interests (RoIs) of different shapes on the CNN output. Features of the same shapes must be extracted from these RoIs (here we assume that the height is h_2 and the width is w_2). Fast R-CNN introduces RoI pooling, which uses the CNN output and RoIs as input to output a concatenation of the features extracted from each proposed region with the shape $n \times c \times h_2 \times w_2$.
3. A fully connected layer is used to transform the output shape to $n \times d$, where d is determined by the model design.
4. During category prediction, the shape of the fully connected layer output is again transformed to $n \times q$ and we use softmax regression (q is the number of categories). During bounding box prediction, the shape of the fully connected layer output is again transformed to $n \times 4$. This means that we predict the category and bounding box for each proposed region.

The RoI pooling layer in Fast R-CNN is somewhat different from the pooling layers we have discussed before. In a normal pooling layer, we set the pooling window, padding, and stride to control the output shape. In an RoI pooling layer, we can directly specify the output shape of each region, such as specifying the height and width of each region as h_2, w_2. Assuming that the height and width of the RoI window are h and w, this window is divided into a grid of sub-windows with the shape $h_2 \times w_2$. The size of each sub-window is about $(h/h_2) \times (w/w_2)$. The sub-window height and width must always be integers and the largest element is used as the output for a given sub-window. This allows the RoI pooling layer to extract features of the same shape from RoIs of different shapes.

In Fig. 13.8.3, we select an 3×3 region as an RoI of the 4×4 input. For this RoI, we use a 2×2 RoI pooling layer to obtain a single 2×2 output. When we divide the region into four sub-windows, they respectively contain the elements 0, 1, 4, and 5 (5 is the largest); 2 and 6 (6 is the largest); 8

and 9 (9 is the largest); and 10.

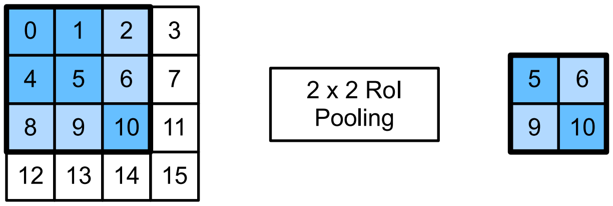


Fig. 13.8.3: 2×2 RoI pooling layer.

We use the ROIPooling function to demonstrate the RoI pooling layer computation. Assume that the CNN extracts the feature X with both a height and width of 4 and only a single channel.

```
from mxnet import np, npx

npx.set_np()

X = np.arange(16).reshape(1, 1, 4, 4)
X
```

```
array([[[[ 0.,  1.,  2.,  3.],
         [ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.],
         [12., 13., 14., 15.]]]])
```

Assume that the height and width of the image are both 40 pixels and that selective search generates two proposed regions on the image. Each region is expressed as five elements: the region's object category and the x, y coordinates of its upper-left and bottom-right corners.

```
rois = np.array([[0, 0, 0, 20, 20], [0, 0, 10, 30, 30]])
```

Because the height and width of X are $1/10$ of the height and width of the image, the coordinates of the two proposed regions are multiplied by 0.1 according to the spatial_scale, and then the RoIs are labeled on X as X[:, :, 0:3, 0:3] and X[:, :, 1:4, 0:4], respectively. Finally, we divide the two RoIs into a sub-window grid and extract features with a height and width of 2.

```
npx.roi_pooling(X, rois, pooled_size=(2, 2), spatial_scale=0.1)
```

```
array([[[[ 5.,  6.],
         [ 9., 10.]]],


       [[[ 9., 11.],
         [13., 15.]]]])
```

13.8.3 Faster R-CNN

In order to obtain precise object detection results, Fast R-CNN generally requires that many proposed regions be generated in selective search. Faster R-CNN replaces selective search with a region proposal network. This reduces the number of proposed regions generated, while ensuring precise object detection.

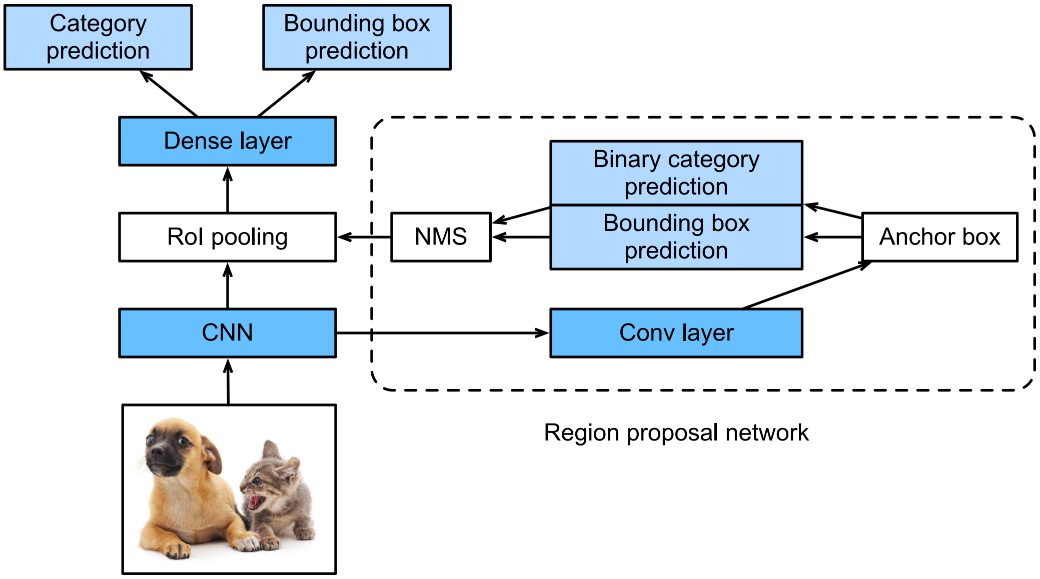


Fig. 13.8.4: Faster R-CNN model.

Fig. 13.8.4 shows a Faster R-CNN model. Compared to Fast R-CNN, Faster R-CNN only changes the method for generating proposed regions from selective search to region proposal network. The other parts of the model remain unchanged. The detailed region proposal network computation process is described below:

1. We use a 3×3 convolutional layer with a padding of 1 to transform the CNN output and set the number of output channels to c. This way, each element in the feature map the CNN extracts from the image is a new feature with a length of c.
2. We use each element in the feature map as a center to generate multiple anchor boxes of different sizes and aspect ratios and then label them.
3. We use the features of the elements of length c at the center on the anchor boxes to predict the binary category (object or background) and bounding box for their respective anchor boxes.
4. Then, we use non-maximum suppression to remove similar bounding box results that correspond to category predictions of "object". Finally, we output the predicted bounding boxes as the proposed regions required by the RoI pooling layer.

It is worth noting that, as a part of the Faster R-CNN model, the region proposal network is trained together with the rest of the model. In addition, the Faster R-CNN object functions include the category and bounding box predictions in object detection, as well as the binary category and bounding box predictions for the anchor boxes in the region proposal network. Finally, the region proposal network can learn how to generate high-quality proposed regions, which reduces the number of proposed regions while maintaining the precision of object detection.

13.8.4 Mask R-CNN

If training data is labeled with the pixel-level positions of each object in an image, a Mask R-CNN model can effectively use these detailed labels to further improve the precision of object detection.

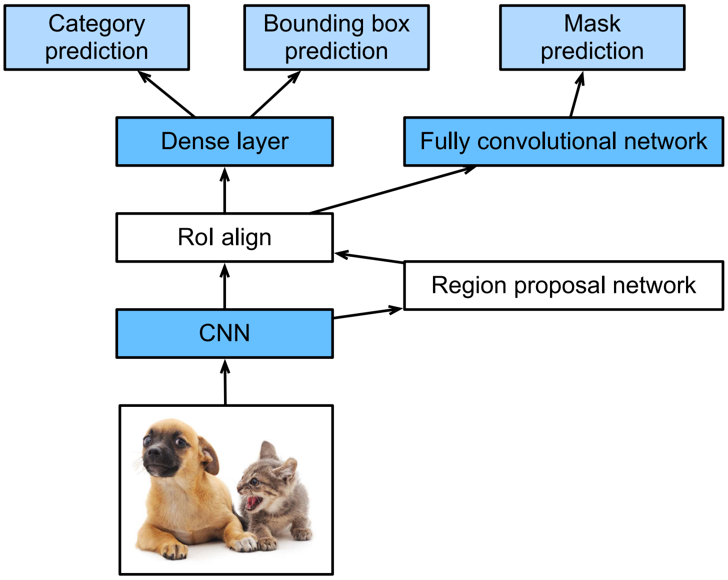


Fig. 13.8.5: Mask R-CNN model.

As shown in Fig. 13.8.5, Mask R-CNN is a modification to the Faster R-CNN model. Mask R-CNN models replace the RoI pooling layer with an RoI alignment layer. This allows the use of bilinear interpolation to retain spatial information on feature maps, making Mask R-CNN better suited for pixel-level predictions. The RoI alignment layer outputs feature maps of the same shape for all RoIs. This not only predicts the categories and bounding boxes of RoIs, but allows us to use an additional fully convolutional network to predict the pixel-level positions of objects. We will describe how to use fully convolutional networks to predict pixel-level semantics in images later in this chapter.

Summary

- An R-CNN model selects several proposed regions and uses a CNN to perform forward computation and extract the features from each proposed region. It then uses these features to predict the categories and bounding boxes of proposed regions.
- Fast R-CNN improves on the R-CNN by only performing CNN forward computation on the image as a whole. It introduces an RoI pooling layer to extract features of the same shape from RoIs of different shapes.
- Faster R-CNN replaces the selective search used in Fast R-CNN with a region proposal network. This reduces the number of proposed regions generated, while ensuring precise object detection.
- Mask R-CNN uses the same basic structure as Faster R-CNN, but adds a fully convolution layer to help locate objects at the pixel level and further improve the precision of object detection.

Exercises

1. Study the implementation of each model in the GluonCV toolkit[196] related to this section.

Discussions[197]

13.9 Semantic Segmentation and the Dataset

In our discussion of object detection issues in the previous sections, we only used rectangular bounding boxes to label and predict objects in images. In this section, we will look at semantic segmentation, which attempts to segment images into regions with different semantic categories. These semantic regions label and predict objects at the pixel level. Fig. 13.9.1 shows a semantically-segmented image, with areas labeled "dog", "cat", and "background". As you can see, compared to object detection, semantic segmentation labels areas with pixel-level borders, for significantly greater precision.

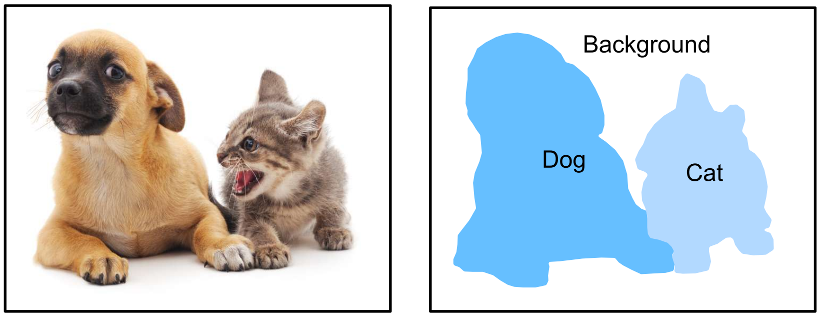


Fig. 13.9.1: Semantically-segmented image, with areas labeled "dog", "cat", and "background".

13.9.1 Image Segmentation and Instance Segmentation

In the computer vision field, there are two important methods related to semantic segmentation: image segmentation and instance segmentation. Here, we will distinguish these concepts from semantic segmentation as follows:

- Image segmentation divides an image into several constituent regions. This method generally uses the correlations between pixels in an image. During training, labels are not needed for image pixels. However, during prediction, this method cannot ensure that the segmented regions have the semantics we want. If we input the image in 9.10, image segmentation might divide the dog into two regions, one covering the dog's mouth and eyes where black is the prominent color and the other covering the rest of the dog where yellow is the prominent color.
- Instance segmentation is also called simultaneous detection and segmentation. This method attempts to identify the pixel-level regions of each object instance in an image. In contrast to semantic segmentation, instance segmentation not only distinguishes semantics, but also different object instances. If an image contains two dogs, instance segmentation will distinguish which pixels belong to which dog.

[196] https://github.com/dmlc/gluon-cv/
[197] https://discuss.d2l.ai/t/374

13.9.2 The Pascal VOC2012 Semantic Segmentation Dataset

In the semantic segmentation field, one important dataset is Pascal VOC2012[198]. To better understand this dataset, we must first import the package or module needed for the experiment.

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import gluon, image, np, npx
import os

npx.set_np()
```

The original site might be unstable, so we download the data from a mirror site. The archive is about 2 GB, so it will take some time to download. After you decompress the archive, the dataset is located in the `../data/VOCdevkit/VOC2012` path.

```
#@save
d2l.DATA_HUB['voc2012'] = (d2l.DATA_URL + 'VOCtrainval_11-May-2012.tar',
                           '4e443f8a2eca6b1dac8a6c57641b67dd40621a49')

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

Go to `../data/VOCdevkit/VOC2012` to see the different parts of the dataset. The `ImageSets/Segmentation` path contains text files that specify the training and testing examples. The `JPEGImages` and `SegmentationClass` paths contain the example input images and labels, respectively. These labels are also in image format, with the same dimensions as the input images to which they correspond. In the labels, pixels with the same color belong to the same semantic category. The `read_voc_images` function defined below reads all input images and labels to the memory.

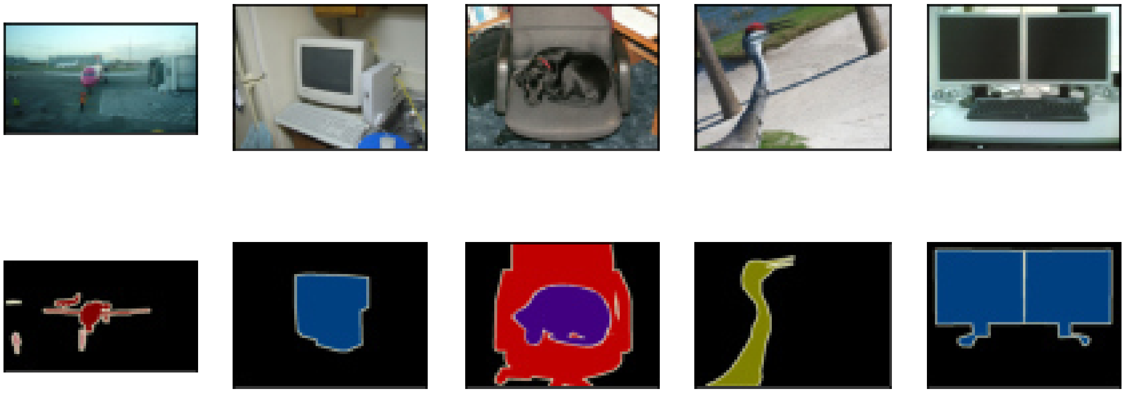
```
#@save
def read_voc_images(voc_dir, is_train=True):
    """Read all VOC feature and label images."""
    txt_fname = os.path.join(voc_dir, 'ImageSets', 'Segmentation',
                             'train.txt' if is_train else 'val.txt')
    with open(txt_fname, 'r') as f:
        images = f.read().split()
    features, labels = [], []
    for i, fname in enumerate(images):
        features.append(image.imread(os.path.join(
            voc_dir, 'JPEGImages', f'{fname}.jpg')))
        labels.append(image.imread(os.path.join(
            voc_dir, 'SegmentationClass', f'{fname}.png')))
    return features, labels

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

We draw the first five input images and their labels. In the label images, white represents borders and black represents the background. Other colors correspond to different categories.

```
n = 5
imgs = train_features[0:n] + train_labels[0:n]
d2l.show_images(imgs, 2, n);
```

[198] http://host.robots.ox.ac.uk/pascal/VOC/voc2012/



Next, we list each RGB color value in the labels and the categories they label.

```
#@save
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]]

#@save
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']
```

After defining the two constants above, we can easily find the category index for each pixel in the labels.

```
#@save
def build_colormap2label():
    """Build an RGB color to label mapping for segmentation."""
    colormap2label = np.zeros(256 ** 3)
    for i, colormap in enumerate(VOC_COLORMAP):
        colormap2label[(colormap[0]*256 + colormap[1])*256 + colormap[2]] = i
    return colormap2label

#@save
def voc_label_indices(colormap, colormap2label):
    """Map an RGB color to a label."""
    colormap = colormap.astype(np.int32)
    idx = ((colormap[:, :, 0] * 256 + colormap[:, :, 1]) * 256
           + colormap[:, :, 2])
    return colormap2label[idx]
```

For example, in the first example image, the category index for the front part of the airplane is 1 and the index for the background is 0.

```
y = voc_label_indices(train_labels[0], build_colormap2label())
y[105:115, 130:140], VOC_CLASSES[1]
```

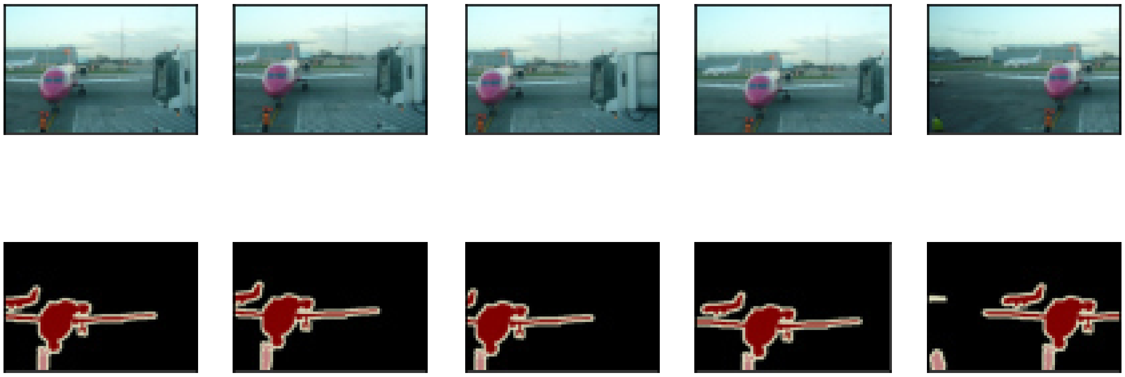
```
(array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 1.],
        [0., 0., 0., 0., 0., 0., 0., 1., 1., 1.],
        [0., 0., 0., 0., 0., 0., 1., 1., 1., 1.],
        [0., 0., 0., 0., 0., 1., 1., 1., 1., 1.],
        [0., 0., 0., 0., 0., 1., 1., 1., 1., 1.],
        [0., 0., 0., 0., 1., 1., 1., 1., 1., 1.],
        [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')
```

Data Preprocessing

In the preceding chapters, we scaled images to make them fit the input shape of the model. In semantic segmentation, this method would require us to re-map the predicted pixel categories back to the original-size input image. It would be very difficult to do this precisely, especially in segmented regions with different semantics. To avoid this problem, we crop the images to set dimensions and do not scale them. Specifically, we use the random cropping method used in image augmentation to crop the same region from input images and their labels.

```
#@save
def voc_rand_crop(feature, label, height, width):
    """Randomly crop for both feature and label images."""
    feature, rect = image.random_crop(feature, (width, height))
    label = image.fixed_crop(label, *rect)
    return feature, label

imgs = []
for _ in range(n):
    imgs += voc_rand_crop(train_features[0], train_labels[0], 200, 300)
d2l.show_images(imgs[::2] + imgs[1::2], 2, n);
```



Dataset Classes for Custom Semantic Segmentation

We use the inherited `Dataset` class provided by Gluon to customize the semantic segmentation dataset class `VOCSegDataset`. By implementing the `__getitem__` function, we can arbitrarily access the input image with the index `idx` and the category indexes for each of its pixels from the dataset. As some images in the dataset may be smaller than the output dimensions specified for random cropping, we must remove these example by using a custom `filter` function. In addition, we define the `normalize_image` function to normalize each of the three RGB channels of the input images.

```
#@save
class VOCSegDataset(gluon.data.Dataset):
    """A customized dataset to load VOC dataset."""

    def __init__(self, is_train, crop_size, voc_dir):
        self.rgb_mean = np.array([0.485, 0.456, 0.406])
        self.rgb_std = np.array([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 = build_colormap2label()
        print('read ' + str(len(self.features)) + ' examples')

    def normalize_image(self, img):
        return (img.astype('float32') / 255 - self.rgb_mean) / self.rgb_std

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

    def __getitem__(self, idx):
        feature, label = voc_rand_crop(self.features[idx], self.labels[idx],
                                       *self.crop_size)
        return (feature.transpose(2, 0, 1),
                voc_label_indices(label, self.colormap2label))

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

Reading the Dataset

Using the custom `VOCSegDataset` class, we create the training set and testing set instances. We assume the random cropping operation output images in the shape 320×480. Below, we can see the number of examples retained in the training and testing sets.

```
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
```

We set the batch size to 64 and define the iterators for the training and testing sets. Print the shape of the first minibatch. In contrast to image classification and object recognition, labels here are three-dimensional arrays.

```
batch_size = 64
train_iter = gluon.data.DataLoader(voc_train, batch_size, shuffle=True,
                                   last_batch='discard',
                                   num_workers=d2l.get_dataloader_workers())
for X, Y in train_iter:
    print(X.shape)
    print(Y.shape)
    break
```

```
(64, 3, 320, 480)
(64, 320, 480)
```

Putting All Things Together

Finally, we define a function `load_data_voc` that downloads and loads this dataset, and then returns the data iterators.

```
#@save
def load_data_voc(batch_size, crop_size):
    """Download and load the VOC2012 semantic dataset."""
    voc_dir = d2l.download_extract('voc2012', os.path.join(
        'VOCdevkit', 'VOC2012'))
    num_workers = d2l.get_dataloader_workers()
    train_iter = gluon.data.DataLoader(
        VOCSegDataset(True, crop_size, voc_dir), batch_size,
        shuffle=True, last_batch='discard', num_workers=num_workers)
    test_iter = gluon.data.DataLoader(
        VOCSegDataset(False, crop_size, voc_dir), batch_size,
        last_batch='discard', num_workers=num_workers)
    return train_iter, test_iter
```

Summary

- Semantic segmentation looks at how images can be segmented into regions with different semantic categories.
- In the semantic segmentation field, one important dataset is Pascal VOC2012.
- Because the input images and labels in semantic segmentation have a one-to-one correspondence at the pixel level, we randomly crop them to a fixed size, rather than scaling them.

Exercises

1. Recall the content we covered in Section 13.1. Which of the image augmentation methods used in image classification would be hard to use in semantic segmentation?

Discussions[199]

13.10 Transposed Convolution

The layers we introduced so far for convolutional neural networks, including convolutional layers (Section 6.2) and pooling layers (Section 6.5), often reduce the input width and height, or keep them unchanged. Applications such as semantic segmentation (Section 13.9) and generative adversarial networks (Section 17.2), however, require to predict values for each pixel and therefore needs to increase input width and height. Transposed convolution, also named fractionally-strided convolution (Dumoulin & Visin, 2016) or deconvolution (Long et al., 2015), serves this purpose.

```
from mxnet import np, npx, init
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()
```

13.10.1 Basic 2D Transposed Convolution

Let us consider a basic case that both input and output channels are 1, with 0 padding and 1 stride. Fig. 13.10.1 illustrates how transposed convolution with a 2×2 kernel is computed on the 2×2 input matrix.

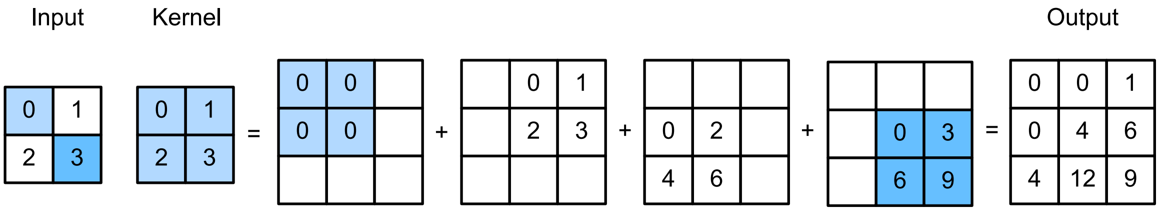


Fig. 13.10.1: Transposed convolution layer with a 2×2 kernel.

We can implement this operation by giving matrix kernel K and matrix input X.

```
def trans_conv(X, K):
    h, w = K.shape
    Y = np.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
```

[199] https://discuss.d2l.ai/t/375

Remember the convolution computes results by `Y[i, j] = (X[i: i + h, j: j + w] * K).sum()` (refer to `corr2d` in Section 6.2), which summarizes input values through the kernel. While the transposed convolution broadcasts input values through the kernel, which results in a larger output shape.

Verify the results in Fig. 13.10.1.

```
X = np.array([[0, 1], [2, 3]])
K = np.array([[0, 1], [2, 3]])
trans_conv(X, K)
```

```
array([[ 0.,  0.,  1.],
       [ 0.,  4.,  6.],
       [ 4., 12.,  9.]])
```

Or we can use `nn.Conv2DTranspose` to obtain the same results. As `nn.Conv2D`, both input and kernel should be 4-D tensors.

```
X, K = X.reshape(1, 1, 2, 2), K.reshape(1, 1, 2, 2)
tconv = nn.Conv2DTranspose(1, kernel_size=2)
tconv.initialize(init.Constant(K))
tconv(X)
```

```
array([[[[ 0.,  0.,  1.],
         [ 0.,  4.,  6.],
         [ 4., 12.,  9.]]]])
```

13.10.2 Padding, Strides, and Channels

We apply padding elements to the input in convolution, while they are applied to the output in transposed convolution. A 1×1 padding means we first compute the output as normal, then remove the first/last rows and columns.

```
tconv = nn.Conv2DTranspose(1, kernel_size=2, padding=1)
tconv.initialize(init.Constant(K))
tconv(X)
```

```
array([[[[4.]]]])
```

Similarly, strides are applied to outputs as well.

```
tconv = nn.Conv2DTranspose(1, kernel_size=2, strides=2)
tconv.initialize(init.Constant(K))
tconv(X)
```

```
array([[[[0., 0., 0., 1.],
         [0., 0., 2., 3.],
         [0., 2., 0., 3.],
         [4., 6., 6., 9.]]]])
```

The multi-channel extension of the transposed convolution is the same as the convolution. When the input has multiple channels, denoted by c_i, the transposed convolution assigns a $k_h \times k_w$ kernel matrix to each input channel. If the output has a channel size c_o, then we have a $c_i \times k_h \times k_w$ kernel for each output channel.

As a result, if we feed X into a convolutional layer f to compute $Y = f(X)$ and create a transposed convolution layer g with the same hyperparameters as f except for the output channel set to be the channel size of X, then $g(Y)$ should has the same shape as X. Let us verify this statement.

```
X = np.random.uniform(size=(1, 10, 16, 16))
conv = nn.Conv2D(20, kernel_size=5, padding=2, strides=3)
tconv = nn.Conv2DTranspose(10, kernel_size=5, padding=2, strides=3)
conv.initialize()
tconv.initialize()
tconv(conv(X)).shape == X.shape
```

```
True
```

13.10.3 Analogy to Matrix Transposition

The transposed convolution takes its name from the matrix transposition. In fact, convolution operations can also be achieved by matrix multiplication. In the example below, we define a $3\times$ input X with a 2×2 kernel K, and then use `corr2d` to compute the convolution output.

```
X = np.arange(9).reshape(3, 3)
K = np.array([[0, 1], [2, 3]])
Y = d2l.corr2d(X, K)
Y
```

```
array([[19., 25.],
       [37., 43.]])
```

Next, we rewrite convolution kernel K as a matrix W. Its shape will be $(4, 9)$, where the i^{th} row present applying the kernel to the input to generate the i^{th} output element.

```
def kernel2matrix(K):
    k, W = np.zeros(5), np.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
```

```
array([[0., 1., 0., 2., 3., 0., 0., 0., 0.],
       [0., 0., 1., 0., 2., 3., 0., 0., 0.],
       [0., 0., 0., 0., 1., 0., 2., 3., 0.],
       [0., 0., 0., 0., 0., 1., 0., 2., 3.]])
```

Then the convolution operator can be implemented by matrix multiplication with proper reshaping.

```
Y == np.dot(W, X.reshape(-1)).reshape(2, 2)
```

```
array([[ True,  True],
       [ True,  True]])
```

We can implement transposed convolution as a matrix multiplication as well by reusing `kernel2matrix`. To reuse the generated W, we construct a 2×2 input, so the corresponding weight matrix will have a shape $(9, 4)$, which is $W^\top$. Let us verify the results.

```
X = np.array([[0, 1], [2, 3]])
Y = trans_conv(X, K)
Y == np.dot(W.T, X.reshape(-1)).reshape(3, 3)
```

```
array([[ True,  True,  True],
       [ True,  True,  True],
       [ True,  True,  True]])
```

Summary

- Compared to convolutions that reduce inputs through kernels, transposed convolutions broadcast inputs.
- If a convolution layer reduces the input width and height by n_w and h_h time, respectively. Then a transposed convolution layer with the same kernel sizes, padding and strides will increase the input width and height by n_w and n_h, respectively.
- We can implement convolution operations by the matrix multiplication, the corresponding transposed convolutions can be done by transposed matrix multiplication.

Exercises

1. Is it efficient to use matrix multiplication to implement convolution operations? Why?

Discussions[200]

13.11 Fully Convolutional Networks (FCN)

We previously discussed semantic segmentation using each pixel in an image for category prediction. A fully convolutional network (FCN) (Long et al., 2015) uses a convolutional neural network to transform image pixels to pixel categories. Unlike the convolutional neural networks previously introduced, an FCN transforms the height and width of the intermediate layer feature map back to the size of input image through the transposed convolution layer, so that the predictions have a one-to-one correspondence with input image in spatial dimension (height and width). Given a position on the spatial dimension, the output of the channel dimension will be a category prediction of the pixel corresponding to the location.

[200] https://discuss.d2l.ai/t/376

We will first import the package or module needed for the experiment and then explain the transposed convolution layer.

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import gluon, image, init, np, npx
from mxnet.gluon import nn

npx.set_np()
```

13.11.1 Constructing a Model

Here, we demonstrate the most basic design of a fully convolutional network model. As shown in Fig. 13.11.1, the fully convolutional network first uses the convolutional neural network to extract image features, then transforms the number of channels into the number of categories through the 1×1 convolution layer, and finally transforms the height and width of the feature map to the size of the input image by using the transposed convolution layer Section 13.10. The model output has the same height and width as the input image and has a one-to-one correspondence in spatial positions. The final output channel contains the category prediction of the pixel of the corresponding spatial position.

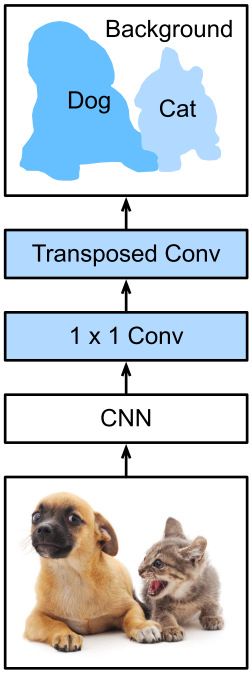


Fig. 13.11.1: Fully convolutional network.

Below, we use a ResNet-18 model pre-trained on the ImageNet dataset to extract image features and record the network instance as `pretrained_net`. As you can see, the last two layers of the model member variable `features` are the global maximum pooling layer `GlobalAvgPool2D` and example flattening layer `Flatten`. The `output` module contains the fully connected layer used for output. These layers are not required for a fully convolutional network.

```
pretrained_net = gluon.model_zoo.vision.resnet18_v2(pretrained=True)
pretrained_net.features[-4:], pretrained_net.output
```

```
(HybridSequential(
   (0): BatchNorm(axis=1, eps=1e-05, momentum=0.9, fix_gamma=False, use_global_stats=False,␣
↪in_channels=512)
   (1): Activation(relu)
   (2): GlobalAvgPool2D(size=(1, 1), stride=(1, 1), padding=(0, 0), ceil_mode=True, global_
↪pool=True, pool_type=avg, layout=NCHW)
   (3): Flatten
 ),
 Dense(512 -> 1000, linear))
```

Next, we create the fully convolutional network instance `net`. It duplicates all the neural layers except the last two layers of the instance member variable `features` of `pretrained_net` and the model parameters obtained after pre-training.

```
net = nn.HybridSequential()
for layer in pretrained_net.features[:-2]:
    net.add(layer)
```

Given an input of a height and width of 320 and 480 respectively, the forward computation of `net` will reduce the height and width of the input to $1/32$ of the original, i.e., 10 and 15.

```
X = np.random.uniform(size=(1, 3, 320, 480))
net(X).shape
```

```
(1, 512, 10, 15)
```

Next, we transform the number of output channels to the number of categories of Pascal VOC2012 (21) through the 1×1 convolution layer. Finally, we need to magnify the height and width of the feature map by a factor of 32 to change them back to the height and width of the input image. Recall the calculation method for the convolution layer output shape described in Section 6.3. Because $(320 - 64 + 16 \times 2 + 32)/32 = 10$ and $(480 - 64 + 16 \times 2 + 32)/32 = 15$, we construct a transposed convolution layer with a stride of 32 and set the height and width of the convolution kernel to 64 and the padding to 16. It is not difficult to see that, if the stride is s, the padding is $s/2$ (assuming $s/2$ is an integer), and the height and width of the convolution kernel are $2s$, the transposed convolution kernel will magnify both the height and width of the input by a factor of s.

```
num_classes = 21
net.add(nn.Conv2D(num_classes, kernel_size=1),
        nn.Conv2DTranspose(
            num_classes, kernel_size=64, padding=16, strides=32))
```

13.11.2 Initializing the Transposed Convolution Layer

We already know that the transposed convolution layer can magnify a feature map. In image processing, sometimes we need to magnify the image, i.e., upsampling. There are many methods for upsampling, and one common method is bilinear interpolation. Simply speaking, in order to get the pixel of the output image at the coordinates (x, y), the coordinates are first mapped to the coordinates of the input image (x', y'). This can be done based on the ratio of the size of three input to the size of the output. The mapped values x' and y' are usually real numbers. Then, we find the four pixels closest to the coordinate (x', y') on the input image. Finally, the pixels of the output image at coordinates (x, y) are calculated based on these four pixels on the input image and their relative distances to (x', y'). Upsampling by bilinear interpolation can be implemented by transposed convolution layer of the convolution kernel constructed using the following `bilinear_kernel` function. Due to space limitations, we only give the implementation of the `bilinear_kernel` function and will not discuss the principles of the algorithm.

```
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 = (np.arange(kernel_size).reshape(-1, 1),
          np.arange(kernel_size).reshape(1, -1))
    filt = (1 - np.abs(og[0] - center) / factor) * \
           (1 - np.abs(og[1] - center) / factor)
    weight = np.zeros((in_channels, out_channels, kernel_size, kernel_size))
    weight[range(in_channels), range(out_channels), :, :] = filt
    return np.array(weight)
```

Now, we will experiment with bilinear interpolation upsampling implemented by transposed convolution layers. Construct a transposed convolution layer that magnifies height and width of input by a factor of 2 and initialize its convolution kernel with the `bilinear_kernel` function.

```
conv_trans = nn.Conv2DTranspose(3, kernel_size=4, padding=1, strides=2)
conv_trans.initialize(init.Constant(bilinear_kernel(3, 3, 4)))
```

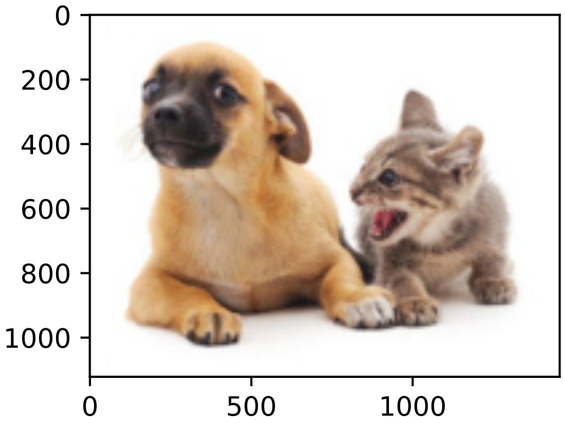
Read the image `X` and record the result of upsampling as `Y`. In order to print the image, we need to adjust the position of the channel dimension.

```
img = image.imread('../img/catdog.jpg')
X = np.expand_dims(img.astype('float32').transpose(2, 0, 1), axis=0) / 255
Y = conv_trans(X)
out_img = Y[0].transpose(1, 2, 0)
```

As you can see, the transposed convolution layer magnifies both the height and width of the image by a factor of 2. It is worth mentioning that, besides to the difference in coordinate scale, the image magnified by bilinear interpolation and original image printed in Section 13.3 look the same.

```
d2l.set_figsize()
print('input image shape:', img.shape)
d2l.plt.imshow(img.asnumpy());
print('output image shape:', out_img.shape)
d2l.plt.imshow(out_img.asnumpy());
```

```
input image shape: (561, 728, 3)
output image shape: (1122, 1456, 3)
```



In a fully convolutional network, we initialize the transposed convolution layer for upsampled bilinear interpolation. For a 1×1 convolution layer, we use Xavier for randomly initialization.

```
W = bilinear_kernel(num_classes, num_classes, 64)
net[-1].initialize(init.Constant(W))
net[-2].initialize(init=init.Xavier())
```

13.11.3 Reading the Dataset

We read the dataset using the method described in the previous section. Here, we specify shape of the randomly cropped output image as 320×480, so both the height and width are divisible by 32.

```
batch_size, crop_size = 32, (320, 480)
train_iter, test_iter = d2l.load_data_voc(batch_size, crop_size)
```

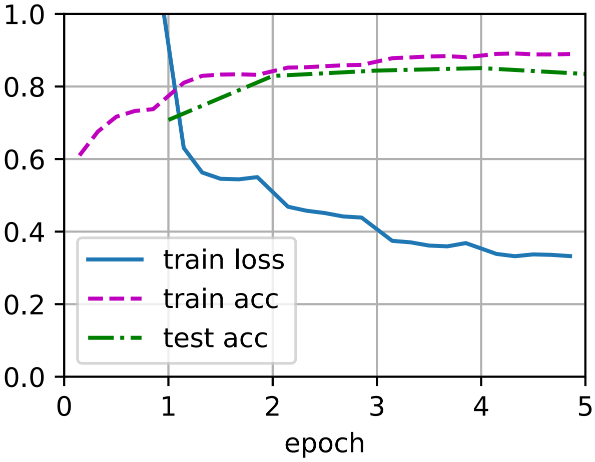
```
Downloading ../data/VOCtrainval_11-May-2012.tar from http://d2l-data.s3-accelerate.amazonaws.
↪com/VOCtrainval_11-May-2012.tar...
read 1114 examples
read 1078 examples
```

13.11.4 Training

Now we can start training the model. The loss function and accuracy calculation here are not substantially different from those used in image classification. Because we use the channel of the transposed convolution layer to predict pixel categories, the `axis=1` (channel dimension) option is specified in `SoftmaxCrossEntropyLoss`. In addition, the model calculates the accuracy based on whether the prediction category of each pixel is correct.

```
num_epochs, lr, wd, devices = 5, 0.1, 1e-3, d2l.try_all_gpus()
loss = gluon.loss.SoftmaxCrossEntropyLoss(axis=1)
net.collect_params().reset_ctx(devices)
trainer = gluon.Trainer(net.collect_params(), 'sgd',
                        {'learning_rate': lr, 'wd': wd})
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices)
```

```
loss 0.334, train acc 0.889, test acc 0.835
268.3 examples/sec on [gpu(0), gpu(1)]
```



13.11.5 Prediction

During predicting, we need to standardize the input image in each channel and transform them into the four-dimensional input format required by the convolutional neural network.

```
def predict(img):
    X = test_iter._dataset.normalize_image(img)
    X = np.expand_dims(X.transpose(2, 0, 1), axis=0)
    pred = net(X.as_in_ctx(devices[0])).argmax(axis=1)
    return pred.reshape(pred.shape[1], pred.shape[2])
```

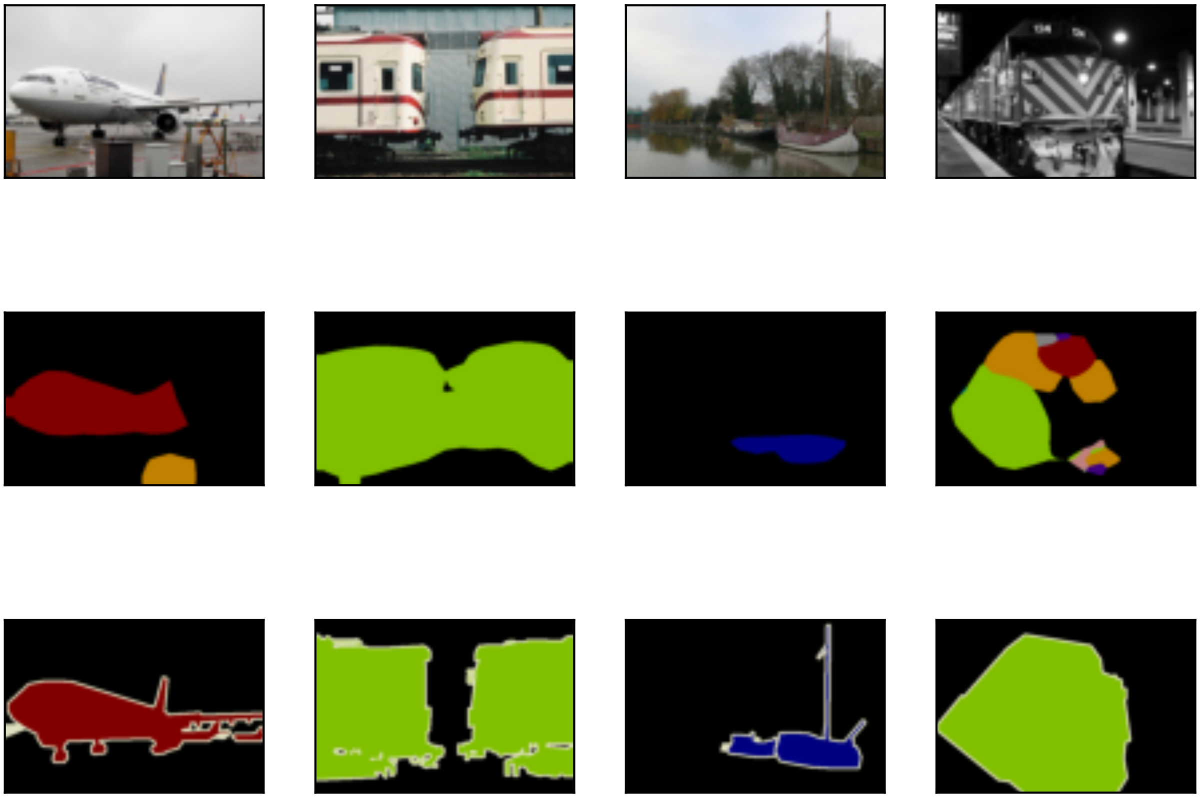
To visualize the predicted categories for each pixel, we map the predicted categories back to their labeled colors in the dataset.

```
def label2image(pred):
    colormap = np.array(d2l.VOC_COLORMAP, ctx=devices[0], dtype='uint8')
    X = pred.astype('int32')
    return colormap[X, :]
```

The size and shape of the images in the test dataset vary. Because the model uses a transposed convolution layer with a stride of 32, when the height or width of the input image is not divisible by 32, the height or width of the transposed convolution layer output deviates from the size of the input image. In order to solve this problem, we can crop multiple rectangular areas in the image with heights and widths as integer multiples of 32, and then perform forward computation on the pixels in these areas. When combined, these areas must completely cover the input image. When a pixel is covered by multiple areas, the average of the transposed convolution layer output in the forward computation of the different areas can be used as an input for the softmax operation to predict the category.

For the sake of simplicity, we only read a few large test images and crop an area with a shape of 320×480 from the top-left corner of the image. Only this area is used for prediction. For the input image, we print the cropped area first, then print the predicted result, and finally print the labeled category.

```
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, 480, 320)
    X = image.fixed_crop(test_images[i], *crop_rect)
    pred = label2image(predict(X))
    imgs += [X, pred, image.fixed_crop(test_labels[i], *crop_rect)]
d2l.show_images(imgs[::3] + imgs[1::3] + imgs[2::3], 3, n, scale=2);
```



Summary

- The fully convolutional network first uses the convolutional neural network to extract image features, then transforms the number of channels into the number of categories through the 1×1 convolution layer, and finally transforms the height and width of the feature map to the size of the input image by using the transposed convolution layer to output the category of each pixel.
- In a fully convolutional network, we initialize the transposed convolution layer for upsampled bilinear interpolation.

Exercises

1. If we use Xavier to randomly initialize the transposed convolution layer, what will happen to the result?
2. Can you further improve the accuracy of the model by tuning the hyperparameters?
3. Predict the categories of all pixels in the test image.
4. The outputs of some intermediate layers of the convolutional neural network are also used in the paper on fully convolutional networks[1]. Try to implement this idea.

Discussions[201]

13.12 Neural Style Transfer

If you use social sharing apps or happen to be an amateur photographer, you are familiar with filters. Filters can alter the color styles of photos to make the background sharper or people's faces whiter. However, a filter generally can only change one aspect of a photo. To create the ideal photo, you often need to try many different filter combinations. This process is as complex as tuning the hyperparameters of a model.

In this section, we will discuss how we can use convolution neural networks (CNNs) to automatically apply the style of one image to another image, an operation known as style transfer (Gatys et al., 2016). Here, we need two input images, one content image and one style image. We use a neural network to alter the content image so that its style mirrors that of the style image. In Fig. 13.12.1, the content image is a landscape photo the author took in Mount Rainier National Part near Seattle. The style image is an oil painting of oak trees in autumn. The output composite image retains the overall shapes of the objects in the content image, but applies the oil painting brushwork of the style image and makes the overall color more vivid.

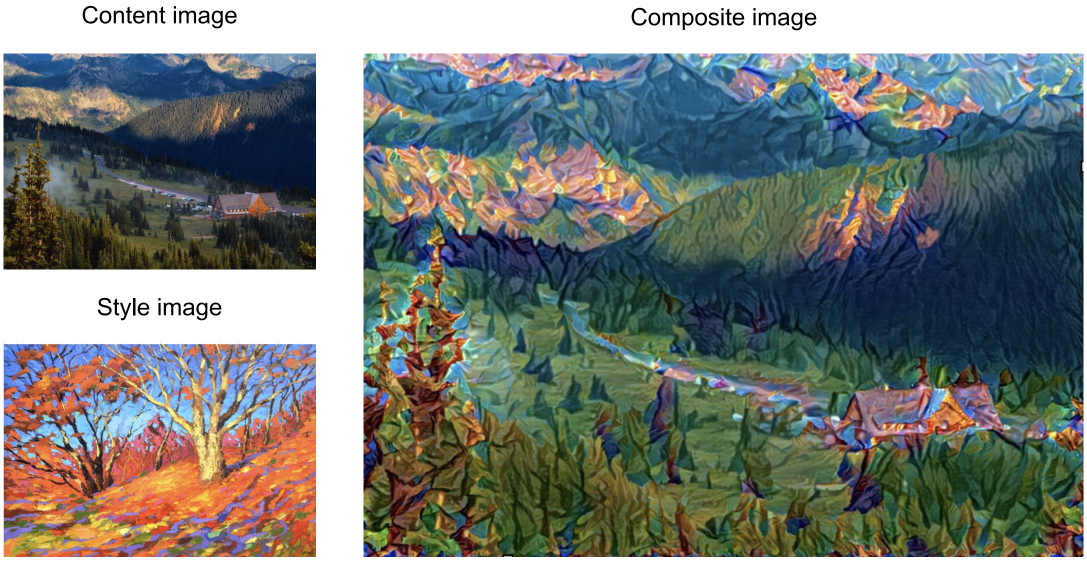


Fig. 13.12.1: Content and style input images and composite image produced by style transfer.

[201] https://discuss.d2l.ai/t/377

13.12.1 Technique

The CNN-based style transfer model is shown in Fig. 13.12.2. First, we initialize the composite image. For example, we can initialize it as the content image. This composite image is the only variable that needs to be updated in the style transfer process, i.e., the model parameter to be updated in style transfer. Then, we select a pre-trained CNN to extract image features. These model parameters do not need to be updated during training. The deep CNN uses multiple neural layers that successively extract image features. We can select the output of certain layers to use as content features or style features. If we use the structure in Fig. 13.12.2, the pre-trained neural network contains three convolutional layers. The second layer outputs the image content features, while the outputs of the first and third layers are used as style features. Next, we use forward propagation (in the direction of the solid lines) to compute the style transfer loss function and backward propagation (in the direction of the dotted lines) to update the model parameter, constantly updating the composite image. The loss functions used in style transfer generally have three parts: 1. Content loss is used to make the composite image approximate the content image as regards content features. 2. Style loss is used to make the composite image approximate the style image in terms of style features. 3. Total variation loss helps reduce the noise in the composite image. Finally, after we finish training the model, we output the style transfer model parameters to obtain the final composite image.

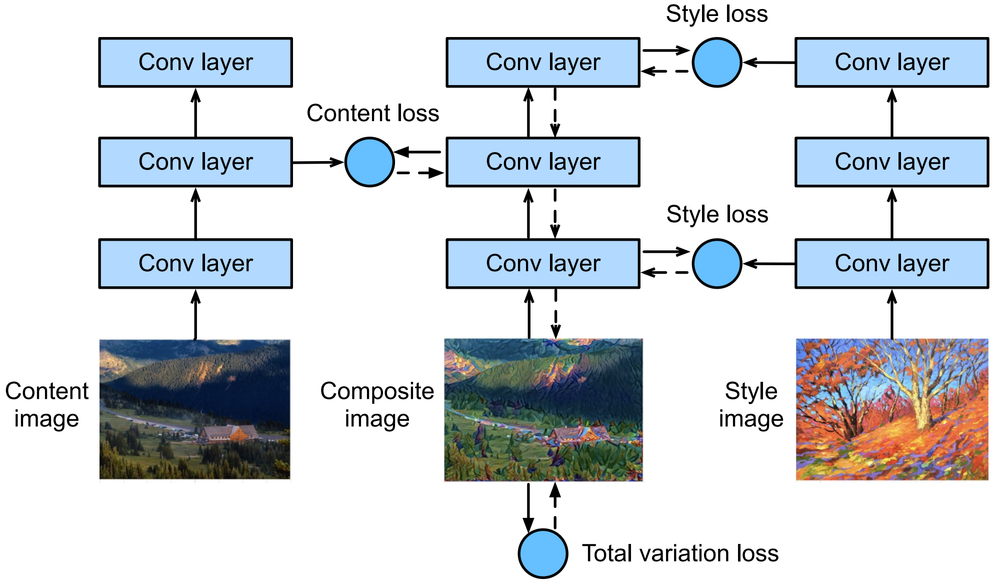


Fig. 13.12.2: CNN-based style transfer process. Solid lines show the direction of forward propagation and dotted lines show backward propagation.

Next, we will perform an experiment to help us better understand the technical details of style transfer.

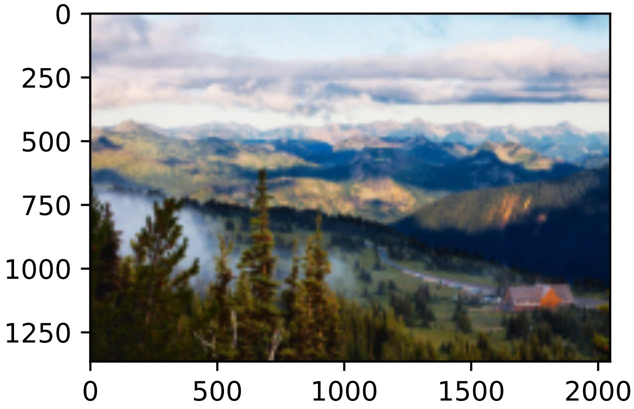
13.12.2 Reading the Content and Style Images

First, we read the content and style images. By printing out the image coordinate axes, we can see that they have different dimensions.

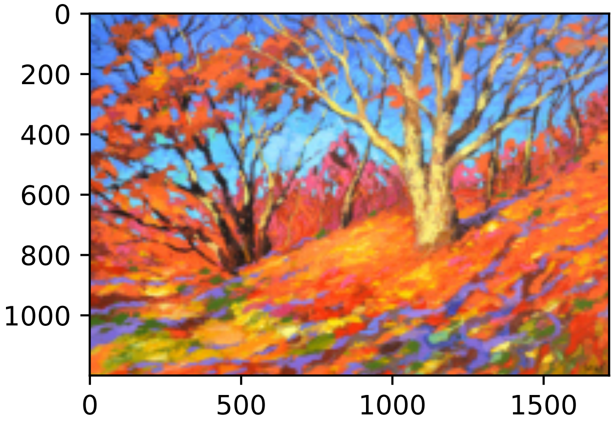
```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import autograd, gluon, image, init, np, npx
from mxnet.gluon import nn

npx.set_np()

d2l.set_figsize()
content_img = image.imread('../img/rainier.jpg')
d2l.plt.imshow(content_img.asnumpy());
```



```
style_img = image.imread('../img/autumn_oak.jpg')
d2l.plt.imshow(style_img.asnumpy());
```



13.12.3 Preprocessing and Postprocessing

Below, we define the functions for image preprocessing and postprocessing. The `preprocess` function normalizes each of the three RGB channels of the input images and transforms the results to a format that can be input to the CNN. The `postprocess` function restores the pixel values in the output image to their original values before normalization. Because the image printing function requires that each pixel has a floating point value from 0 to 1, we use the `clip` function to replace values smaller than 0 or greater than 1 with 0 or 1, respectively.

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

def preprocess(img, image_shape):
    img = image.imresize(img, *image_shape)
    img = (img.astype('float32') / 255 - rgb_mean) / rgb_std
    return np.expand_dims(img.transpose(2, 0, 1), axis=0)

def postprocess(img):
    img = img[0].as_in_ctx(rgb_std.ctx)
    return (img.transpose(1, 2, 0) * rgb_std + rgb_mean).clip(0, 1)
```

13.12.4 Extracting Features

We use the VGG-19 model pre-trained on the ImageNet dataset to extract image features[1].

```
pretrained_net = gluon.model_zoo.vision.vgg19(pretrained=True)
```

To extract image content and style features, we can select the outputs of certain layers in the VGG network. In general, the closer an output is to the input layer, the easier it is to extract image detail information. The farther away an output is, the easier it is to extract global information. To prevent the composite image from retaining too many details from the content image, we select a VGG network layer near the output layer to output the image content features. This layer is called the content layer. We also select the outputs of different layers from the VGG network for matching local and global styles. These are called the style layers. As we mentioned in Section 7.2, VGG networks have five convolutional blocks. In this experiment, we select the last convolutional layer of the fourth convolutional block as the content layer and the first layer of each block as style layers. We can obtain the indexes for these layers by printing the `pretrained_net` instance.

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

During feature extraction, we only need to use all the VGG layers from the input layer to the content or style layer nearest the output layer. Below, we build a new network, `net`, which only retains the layers in the VGG network we need to use. We then use `net` to extract features.

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

Given input `X`, if we simply call the forward computation `net(X)`, we can only obtain the output of the last layer. Because we also need the outputs of the intermediate layers, we need to perform layer-by-layer computation and retain the content and style layer outputs.

```
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
```

Next, we define two functions: The `get_contents` function obtains the content features extracted from the content image, while the `get_styles` function obtains the style features extracted from the style image. Because we do not need to change the parameters of the pre-trained VGG model during training, we can extract the content features from the content image and style features from the style image before the start of training. As the composite image is the model parameter that must be updated during style transfer, we can only call the `extract_features` function during training to extract the content and style features of the composite image.

```
def get_contents(image_shape, device):
    content_X = preprocess(content_img, image_shape).copyto(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).copyto(device)
    _, styles_Y = extract_features(style_X, content_layers, style_layers)
    return style_X, styles_Y
```

13.12.5 Defining the Loss Function

Next, we will look at the loss function used for style transfer. The loss function includes the content loss, style loss, and total variation loss.

Content Loss

Similar to the loss function used in linear regression, content loss uses a square error function to measure the difference in content features between the composite image and content image. The two inputs of the square error function are both content layer outputs obtained from the `extract_features` function.

```
def content_loss(Y_hat, Y):
    return np.square(Y_hat - Y).mean()
```

Style Loss

Style loss, similar to content loss, uses a square error function to measure the difference in style between the composite image and style image. To express the styles output by the style layers, we first use the `extract_features` function to compute the style layer output. Assuming that the output has 1 example, c channels, and a height and width of h and w, we can transform the output into the matrix $\mathbf{X}$, which has c rows and $h{\cdot}w$ columns. You can think of matrix $\mathbf{X}$ as the combination of the c vectors $\mathbf{x}_1, \ldots, \mathbf{x}_c$, which have a length of hw. Here, the vector $\mathbf{x}_i$ represents the style feature of channel i. In the Gram matrix of these vectors $\mathbf{X}\mathbf{X}^\top \in \mathbb{R}^{c\times c}$, element x_{ij} in row i column j is the inner product of vectors $\mathbf{x}_i$ and $\mathbf{x}_j$. It represents the correlation of the style features of channels i and j. We use this type of Gram matrix to represent the style output by the style layers. You must note that, when the $h{\cdot}w$ value is large, this often leads to large values in the Gram matrix. In addition, the height and width of the Gram matrix are both the number of channels c. To ensure that the style loss is not affected by the size of these values, we define the `gram` function below to divide the Gram matrix by the number of its elements, i.e., $c \cdot h \cdot w$.

```
def gram(X):
    num_channels, n = X.shape[1], X.size // X.shape[1]
    X = X.reshape(num_channels, n)
    return np.dot(X, X.T) / (num_channels * n)
```

Naturally, the two Gram matrix inputs of the square error function for style loss are taken from the composite image and style image style layer outputs. Here, we assume that the Gram matrix of the style image, `gram_Y`, has been computed in advance.

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

Total Variance Loss

Sometimes, the composite images we learn have a lot of high-frequency noise, particularly bright or dark pixels. One common noise reduction method is total variation denoising. We assume that $x_{i,j}$ represents the pixel value at the coordinate (i, j), so the total variance loss is:

$$\sum_{i,j} \left|x_{i,j} - x_{i+1,j}\right| + \left|x_{i,j} - x_{i,j+1}\right|. \tag{13.12.1}$$

We try to make the values of neighboring pixels as similar as possible.

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

The Loss Function

The loss function for style transfer is the weighted sum of the content loss, style loss, and total variance loss. By adjusting these weight hyperparameters, we can balance the retained content, transferred style, and noise reduction in the composite image according to their relative importance.

```
content_weight, style_weight, tv_weight = 1, 1e3, 10

def compute_loss(X, contents_Y_hat, styles_Y_hat, contents_Y, styles_Y_gram):
    # Calculate the content, style, and total variance losses respectively
    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
    # Add up all the losses
    l = sum(styles_l + contents_l + [tv_l])
    return contents_l, styles_l, tv_l, l
```

13.12.6 Creating and Initializing the Composite Image

In style transfer, the composite image is the only variable that needs to be updated. Therefore, we can define a simple model, `GeneratedImage`, and treat the composite image as a model parameter. In the model, forward computation only returns the model parameter.

```
class GeneratedImage(nn.Block):
    def __init__(self, img_shape, **kwargs):
        super(GeneratedImage, self).__init__(**kwargs)
        self.weight = self.params.get('weight', shape=img_shape)

    def forward(self):
        return self.weight.data()
```

Next, we define the `get_inits` function. This function creates a composite image model instance and initializes it to the image `X`. The Gram matrix for the various style layers of the style image, `styles_Y_gram`, is computed prior to training.

```
def get_inits(X, device, lr, styles_Y):
    gen_img = GeneratedImage(X.shape)
    gen_img.initialize(init.Constant(X), ctx=device, force_reinit=True)
    trainer = gluon.Trainer(gen_img.collect_params(), 'adam',
                            {'learning_rate': lr})
    styles_Y_gram = [gram(Y) for Y in styles_Y]
    return gen_img(), styles_Y_gram, trainer
```

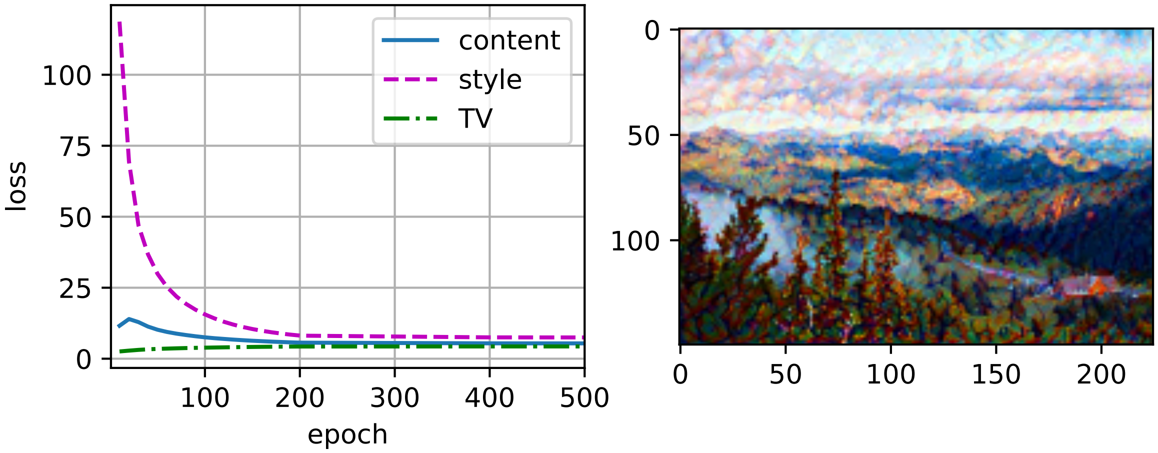
13.12.7 Training

During model training, we constantly extract the content and style features of the composite image and calculate the loss function. Recall our discussion of how synchronization functions force the front end to wait for computation results in Section 12.2. Because we only call the `asscalar` synchronization function every 50 epochs, the process may occupy a great deal of memory. Therefore, we call the `waitall` synchronization function during every epoch.

```
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)
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[1, num_epochs],
                            legend=['content', 'style', 'TV'],
                            ncols=2, figsize=(7, 2.5))
    for epoch in range(1, num_epochs+1):
        with autograd.record():
            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(1)
        npx.waitall()
        if epoch % lr_decay_epoch == 0:
            trainer.set_learning_rate(trainer.learning_rate * 0.1)
        if epoch % 10 == 0:
            animator.axes[1].imshow(postprocess(X).asnumpy())
            animator.add(epoch, [float(sum(contents_l)),
                                 float(sum(styles_l)),
                                 float(tv_l)])
    return X
```

Next, we start to train the model. First, we set the height and width of the content and style images to 150 by 225 pixels. We use the content image to initialize the composite image.

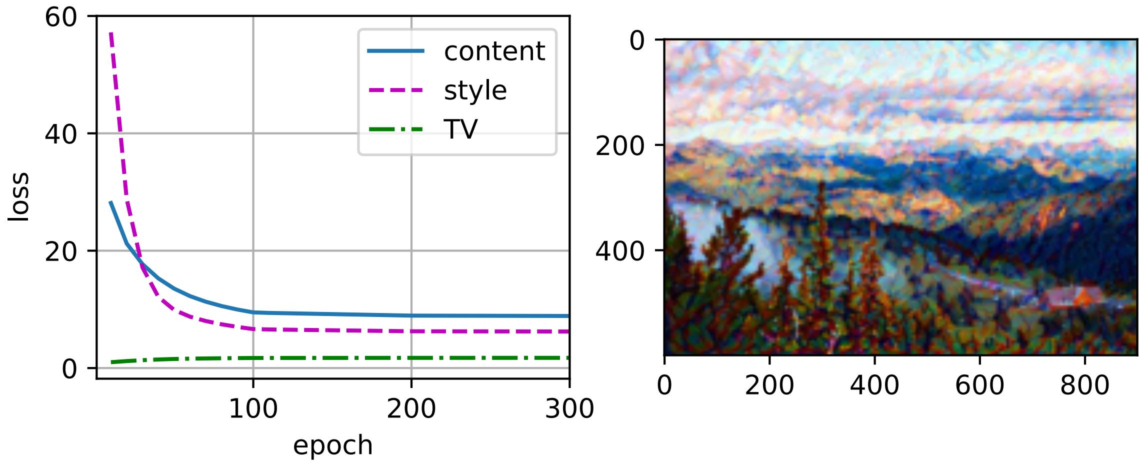
```
device, image_shape = d2l.try_gpu(), (225, 150)
net.collect_params().reset_ctx(device)
content_X, contents_Y = get_contents(image_shape, device)
_, styles_Y = get_styles(image_shape, device)
output = train(content_X, contents_Y, styles_Y, device, 0.01, 500, 200)
```



As you can see, the composite image retains the scenery and objects of the content image, while introducing the color of the style image. Because the image is relatively small, the details are a bit fuzzy.

To obtain a clearer composite image, we train the model using a larger image size: 900×600. We increase the height and width of the image used before by a factor of four and initialize a larger composite image.

```
image_shape = (900, 600)
_, content_Y = get_contents(image_shape, device)
_, style_Y = get_styles(image_shape, device)
X = preprocess(postprocess(output) * 255, image_shape)
output = train(X, content_Y, style_Y, device, 0.01, 300, 100)
d2l.plt.imsave('../img/neural-style.jpg', postprocess(output).asnumpy())
```



As you can see, each epoch takes more time due to the larger image size. As shown in Fig. 13.12.3, the composite image produced retains more detail due to its larger size. The composite image not only has large blocks of color like the style image, but these blocks even have the subtle texture of brush strokes.

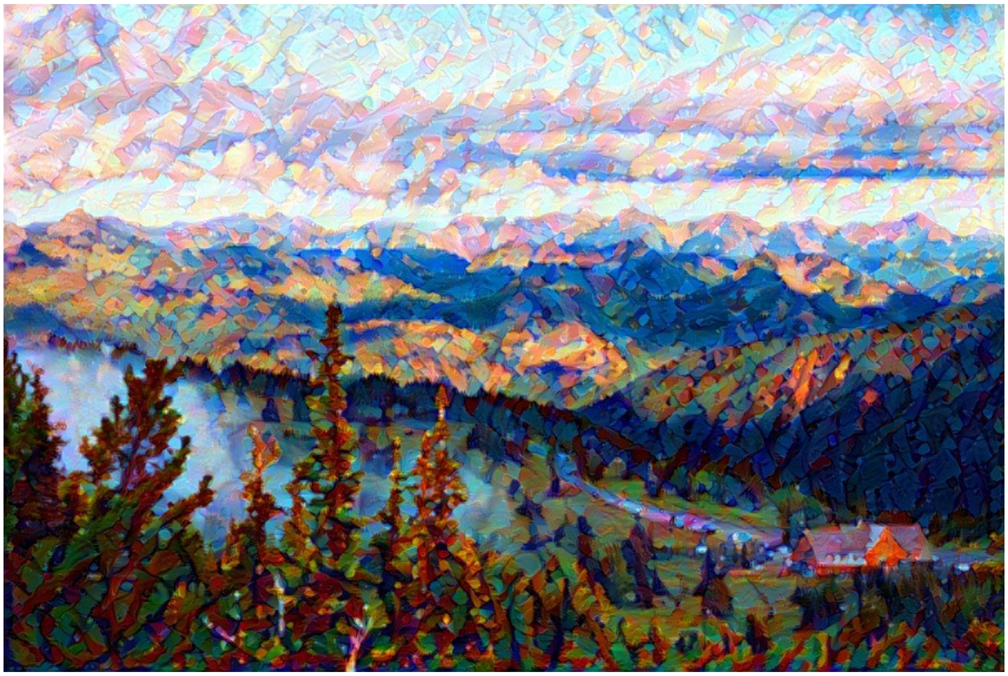


Fig. 13.12.3: 900×600 composite image.

Summary

- The loss functions used in style transfer generally have three parts: 1. Content loss is used to make the composite image approximate the content image as regards content features. 2. Style loss is used to make the composite image approximate the style image in terms of style features. 3. Total variation loss helps reduce the noise in the composite image.
- We can use a pre-trained CNN to extract image features and minimize the loss function to continuously update the composite image.
- We use a Gram matrix to represent the style output by the style layers.

Exercises

1. How does the output change when you select different content and style layers?
2. Adjust the weight hyperparameters in the loss function. Does the output retain more content or have less noise?
3. Use different content and style images. Can you create more interesting composite images?

Discussions[202]

[202] https://discuss.d2l.ai/t/378

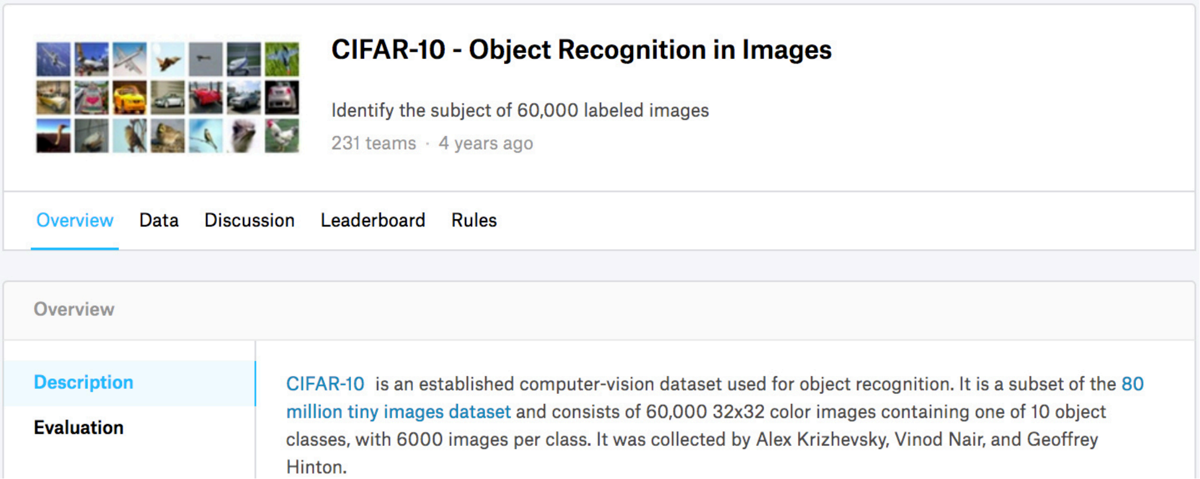
13.13 Image Classification (CIFAR-10) on Kaggle

So far, we have been using Gluon's `data` package to directly obtain image datasets in the tensor format. In practice, however, image datasets often exist in the format of image files. In this section, we will start with the original image files and organize, read, and convert the files to the tensor format step by step.

We performed an experiment on the CIFAR-10 dataset in Section 13.1. This is an important data set in the computer vision field. Now, we will apply the knowledge we learned in the previous sections in order to participate in the Kaggle competition, which addresses CIFAR-10 image classification problems. The competition's web address is

> https://www.kaggle.com/c/cifar-10

Fig. 13.13.1 shows the information on the competition's webpage. In order to submit the results, please register an account on the Kaggle website first.



CIFAR-10 - Object Recognition in Images

Identify the subject of 60,000 labeled images

231 teams · 4 years ago

CIFAR-10 is an established computer-vision dataset used for object recognition. It is a subset of the 80 million tiny images dataset and consists of 60,000 32x32 color images containing one of 10 object classes, with 6000 images per class. It was collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton.

Fig. 13.13.1: CIFAR-10 image classification competition webpage information. The dataset for the competition can be accessed by clicking the "Data" tab.

First, import the packages or modules required for the competition.

```python
import collections
from d2l import mxnet as d2l
import math
from mxnet import autograd, gluon, init, npx
from mxnet.gluon import nn
import os
import pandas as pd
import shutil
import time

npx.set_np()
```

13.13.1 Obtaining and Organizing the Dataset

The competition data is divided into a training set and testing set. The training set contains $50,000$ images. The testing set contains $300,000$ images, of which $10,000$ images are used for scoring, while the other $290,000$ non-scoring images are included to prevent the manual labeling of the testing set and the submission of labeling results. The image formats in both datasets are PNG, with heights and widths of 32 pixels and three color channels (RGB). The images cover 10 categories: planes, cars, birds, cats, deer, dogs, frogs, horses, boats, and trucks. The upper-left corner of Figure 9.16 shows some images of planes, cars, and birds in the dataset.

Downloading the Dataset

After logging in to Kaggle, we can click on the "Data" tab on the CIFAR-10 image classification competition webpage shown in Fig. 13.13.1 and download the dataset by clicking the "Download All" button. After unzipping the downloaded file in `../data`, and unzipping `train.7z` and `test.7z` inside it, you will find the entire dataset in the following paths:

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

Here folders `train` and `test` contain the training and testing images respectively, `trainLabels.csv` has labels for the training images, and `sample_submission.csv` is a sample of submission.

To make it easier to get started, we provide a small-scale sample of the dataset: it contains the first 1000 training images and 5 random testing images. To use the full dataset of the Kaggle competition, you need to set the following `demo` variable to `False`.

```python
#@save
d2l.DATA_HUB['cifar10_tiny'] = (d2l.DATA_URL + 'kaggle_cifar10_tiny.zip',
                                '2068874e4b9a9f0fb07ebe0ad2b29754449ccacd')

# If you use the full dataset downloaded for the Kaggle competition, set
# `demo` to False
demo = True

if demo:
    data_dir = d2l.download_extract('cifar10_tiny')
else:
    data_dir = '../data/cifar-10/'
```

```
Downloading ../data/kaggle_cifar10_tiny.zip from http://d2l-data.s3-accelerate.amazonaws.com/
↪kaggle_cifar10_tiny.zip...
```

Organizing the Dataset

We need to organize datasets to facilitate model training and testing. Let us first read the labels from the csv file. The following function returns a dictionary that maps the filename without extension to its label.

```
#@save
def read_csv_labels(fname):
    """Read fname to return a name to label dictionary."""
    with open(fname, 'r') as f:
        # Skip the file header line (column name)
        lines = f.readlines()[1:]
    tokens = [l.rstrip().split(',') for l in lines]
    return dict(((name, label) for name, label in tokens))

labels = read_csv_labels(os.path.join(data_dir, 'trainLabels.csv'))
print('# training examples:', len(labels))
print('# classes:', len(set(labels.values())))
```

```
# training examples: 1000
# classes: 10
```

Next, we define the `reorg_train_valid` function to segment the validation set from the original training set. The argument `valid_ratio` in this function is the ratio of the number of examples in the validation set to the number of examples in the original training set. In particular, let n be the number of images of the class with the least examples, and r be the ratio, then we will use max$(\lfloor nr \rfloor, 1)$ images for each class as the validation set. Let us use `valid_ratio=0.1` as an example. Since the original training set has $50,000$ images, there will be $45,000$ images used for training and stored in the path "`train_valid_test/train`" when tuning hyperparameters, while the other $5,000$ images will be stored as validation set in the path "`train_valid_test/valid`". After organizing the data, images of the same class will be placed under the same folder so that we can read them later.

```
#@save
def copyfile(filename, target_dir):
    """Copy a file into a target directory."""
    d2l.mkdir_if_not_exist(target_dir)
    shutil.copy(filename, target_dir)

#@save
def reorg_train_valid(data_dir, labels, valid_ratio):
    # The number of examples of the class with the least examples in the
    # training dataset
    n = collections.Counter(labels.values()).most_common()[-1][1]
    # The number of examples per class for the validation set
    n_valid_per_label = max(1, math.floor(n * valid_ratio))
    label_count = {}
    for train_file in os.listdir(os.path.join(data_dir, 'train')):
        label = labels[train_file.split('.')[0]]
        fname = os.path.join(data_dir, 'train', train_file)
        # Copy to train_valid_test/train_valid with a subfolder per class
        copyfile(fname, os.path.join(data_dir, 'train_valid_test',
                                     'train_valid', label))
```

(continues on next page)

(continued from previous page)

```
        if label not in label_count or label_count[label] < n_valid_per_label:
            # Copy to train_valid_test/valid
            copyfile(fname, os.path.join(data_dir, 'train_valid_test',
                                         'valid', label))
            label_count[label] = label_count.get(label, 0) + 1
        else:
            # Copy to train_valid_test/train
            copyfile(fname, os.path.join(data_dir, 'train_valid_test',
                                         'train', label))
    return n_valid_per_label
```

The reorg_test function below is used to organize the testing set to facilitate the reading during prediction.

```
#@save
def reorg_test(data_dir):
    for test_file in os.listdir(os.path.join(data_dir, 'test')):
        copyfile(os.path.join(data_dir, 'test', test_file),
                 os.path.join(data_dir, 'train_valid_test', 'test',
                              'unknown'))
```

Finally, we use a function to call the previously defined read_csv_labels, reorg_train_valid, and reorg_test functions.

```
def reorg_cifar10_data(data_dir, valid_ratio):
    labels = read_csv_labels(os.path.join(data_dir, 'trainLabels.csv'))
    reorg_train_valid(data_dir, labels, valid_ratio)
    reorg_test(data_dir)
```

We only set the batch size to 4 for the demo dataset. During actual training and testing, the complete dataset of the Kaggle competition should be used and batch_size should be set to a larger integer, such as 128. We use 10% of the training examples as the validation set for tuning hyperparameters.

```
batch_size = 4 if demo else 128
valid_ratio = 0.1
reorg_cifar10_data(data_dir, valid_ratio)
```

13.13.2 Image Augmentation

To cope with overfitting, we use image augmentation. For example, by adding transforms.RandomFlipLeftRight(), the images can be flipped at random. We can also perform normalization for the three RGB channels of color images using transforms.Normalize(). Below, we list some of these operations that you can choose to use or modify depending on requirements.

```
transform_train = gluon.data.vision.transforms.Compose([
    # Magnify the image to a square of 40 pixels in both height and width
    gluon.data.vision.transforms.Resize(40),
    # Randomly crop a square image of 40 pixels in both height and width to
    # produce a small square of 0.64 to 1 times the area of the original
```

(continues on next page)

(continued from previous page)

```
    # image, and then shrink it to a square of 32 pixels in both height and
    # width
    gluon.data.vision.transforms.RandomResizedCrop(32, scale=(0.64, 1.0),
                                                   ratio=(1.0, 1.0)),
    gluon.data.vision.transforms.RandomFlipLeftRight(),
    gluon.data.vision.transforms.ToTensor(),
    # Normalize each channel of the image
    gluon.data.vision.transforms.Normalize([0.4914, 0.4822, 0.4465],
                                           [0.2023, 0.1994, 0.2010])])
```

In order to ensure the certainty of the output during testing, we only perform normalization on the image.

```
transform_test = gluon.data.vision.transforms.Compose([
    gluon.data.vision.transforms.ToTensor(),
    gluon.data.vision.transforms.Normalize([0.4914, 0.4822, 0.4465],
                                           [0.2023, 0.1994, 0.2010])])
```

13.13.3 Reading the Dataset

Next, we can create the ImageFolderDataset instance to read the organized dataset containing the original image files, where each example includes the image and label.

```
train_ds, valid_ds, train_valid_ds, test_ds = [
    gluon.data.vision.ImageFolderDataset(
        os.path.join(data_dir, 'train_valid_test', folder))
    for folder in ['train', 'valid', 'train_valid', 'test']]
```

We specify the defined image augmentation operation in DataLoader. During training, we only use the validation set to evaluate the model, so we need to ensure the certainty of the output. During prediction, we will train the model on the combined training set and validation set to make full use of all labelled data.

```
train_iter, train_valid_iter = [gluon.data.DataLoader(
    dataset.transform_first(transform_train), batch_size, shuffle=True,
    last_batch='discard') for dataset in (train_ds, train_valid_ds)]

valid_iter = gluon.data.DataLoader(
    valid_ds.transform_first(transform_test), batch_size, shuffle=False,
    last_batch='discard')

test_iter = gluon.data.DataLoader(
    test_ds.transform_first(transform_test), batch_size, shuffle=False,
    last_batch='keep')
```

13.13.4 Defining the Model

Here, we build the residual blocks based on the HybridBlock class, which is slightly different than the implementation described in Section 7.6. This is done to improve execution efficiency.

```
class Residual(nn.HybridBlock):
    def __init__(self, num_channels, use_1x1conv=False, strides=1, **kwargs):
        super(Residual, self).__init__(**kwargs)
        self.conv1 = nn.Conv2D(num_channels, kernel_size=3, padding=1,
                               strides=strides)
        self.conv2 = nn.Conv2D(num_channels, kernel_size=3, padding=1)
        if use_1x1conv:
            self.conv3 = nn.Conv2D(num_channels, kernel_size=1,
                                   strides=strides)
        else:
            self.conv3 = None
        self.bn1 = nn.BatchNorm()
        self.bn2 = nn.BatchNorm()

    def hybrid_forward(self, F, X):
        Y = F.npx.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3:
            X = self.conv3(X)
        return F.npx.relu(Y + X)
```

Next, we define the ResNet-18 model.

```
def resnet18(num_classes):
    net = nn.HybridSequential()
    net.add(nn.Conv2D(64, kernel_size=3, strides=1, padding=1),
            nn.BatchNorm(), nn.Activation('relu'))

    def resnet_block(num_channels, num_residuals, first_block=False):
        blk = nn.HybridSequential()
        for i in range(num_residuals):
            if i == 0 and not first_block:
                blk.add(Residual(num_channels, use_1x1conv=True, strides=2))
            else:
                blk.add(Residual(num_channels))
        return blk

    net.add(resnet_block(64, 2, first_block=True),
            resnet_block(128, 2),
            resnet_block(256, 2),
            resnet_block(512, 2))
    net.add(nn.GlobalAvgPool2D(), nn.Dense(num_classes))
    return net
```

The CIFAR-10 image classification challenge uses 10 categories. We will perform Xavier random initialization on the model before training begins.

```
def get_net(devices):
    num_classes = 10
    net = resnet18(num_classes)
```

(continues on next page)

(continued from previous page)

```
    net.initialize(ctx=devices, init=init.Xavier())
    return net

loss = gluon.loss.SoftmaxCrossEntropyLoss()
```

13.13.5 Defining the Training Functions

We will select the model and tune hyperparameters according to the model's performance on the validation set. Next, we define the model training function `train`. We record the training time of each epoch, which helps us compare the time costs of different models.

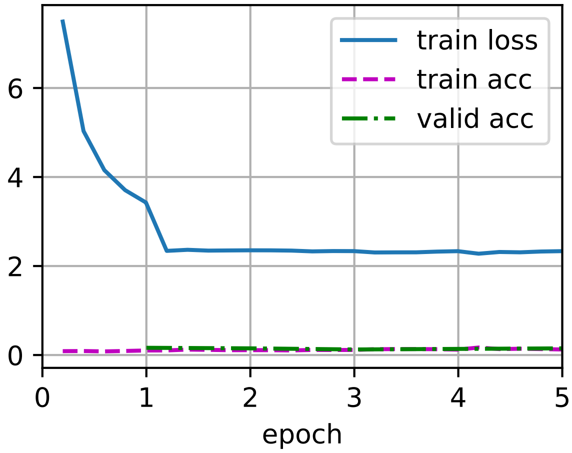
```
def train(net, train_iter, valid_iter, num_epochs, lr, wd, devices, lr_period,
          lr_decay):
    trainer = gluon.Trainer(net.collect_params(), 'sgd',
                            {'learning_rate': lr, 'momentum': 0.9, 'wd': wd})
    num_batches, timer = len(train_iter), d2l.Timer()
    animator = d2l.Animator(xlabel='epoch', xlim=[0, num_epochs],
                            legend=['train loss', 'train acc', 'valid acc'])
    for epoch in range(num_epochs):
        metric = d2l.Accumulator(3)
        if epoch > 0 and epoch % lr_period == 0:
            trainer.set_learning_rate(trainer.learning_rate * lr_decay)
        for i, (features, labels) in enumerate(train_iter):
            timer.start()
            l, acc = d2l.train_batch_ch13(
                net, features, labels.astype('float32'), loss, trainer,
                devices, d2l.split_batch)
            metric.add(l, acc, labels.shape[0])
            timer.stop()
            if (i + 1) % (num_batches // 5) == 0:
                animator.add(epoch + i / num_batches,
                             (metric[0] / metric[2], metric[1] / metric[2],
                              None))
        if valid_iter is not None:
            valid_acc = d2l.evaluate_accuracy_gpus(net, valid_iter, d2l.split_batch)
            animator.add(epoch + 1, (None, None, valid_acc))
    if valid_iter is not None:
        print(f'loss {metric[0] / metric[2]:.3f}, '
              f'train acc {metric[1] / metric[2]:.3f}, '
              f'valid acc {valid_acc:.3f}')
    else:
        print(f'loss {metric[0] / metric[2]:.3f}, '
              f'train acc {metric[1] / metric[2]:.3f}')
    print(f'{metric[2] * num_epochs / timer.sum():.1f} examples/sec '
          f'on {str(devices)}')
```

13.13.6 Training and Validating the Model

Now, we can train and validate the model. The following hyperparameters can be tuned. For example, we can increase the number of epochs. Because `lr_period` and `lr_decay` are set to 80 and 0.1 respectively, the learning rate of the optimization algorithm will be multiplied by 0.1 after every 80 epochs. For simplicity, we only train one epoch here.

```
devices, num_epochs, lr, wd = d2l.try_all_gpus(), 5, 0.1, 5e-4
lr_period, lr_decay, net = 50, 0.1, get_net(devices)
net.hybridize()
train(net, train_iter, valid_iter, num_epochs, lr, wd, devices, lr_period,
      lr_decay)
```

```
loss 2.334, train acc 0.125, valid acc 0.150
94.3 examples/sec on [gpu(0), gpu(1)]
```



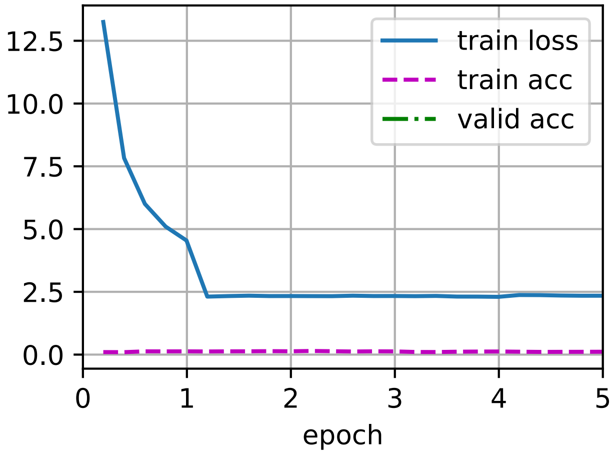
13.13.7 Classifying the Testing Set and Submitting Results on Kaggle

After obtaining a satisfactory model design and hyperparameters, we use all training datasets (including validation sets) to retrain the model and classify the testing set.

```
net, preds = get_net(devices), []
net.hybridize()
train(net, train_valid_iter, None, num_epochs, lr, wd, devices, lr_period,
      lr_decay)

for X, _ in test_iter:
    y_hat = net(X.as_in_ctx(devices[0]))
    preds.extend(y_hat.argmax(axis=1).astype(int).asnumpy())
sorted_ids = list(range(1, len(test_ds) + 1))
sorted_ids.sort(key=lambda x: str(x))
df = pd.DataFrame({'id': sorted_ids, 'label': preds})
df['label'] = df['label'].apply(lambda x: train_valid_ds.synsets[x])
df.to_csv('submission.csv', index=False)
```

```
loss 2.345, train acc 0.114
97.3 examples/sec on [gpu(0), gpu(1)]
```



After executing the above code, we will get a "submission.csv" file. The format of this file is consistent with the Kaggle competition requirements. The method for submitting results is similar to method in Section 4.10.

Summary

- We can create an `ImageFolderDataset` instance to read the dataset containing the original image files.
- We can use convolutional neural networks, image augmentation, and hybrid programming to take part in an image classification competition.

Exercises

1. Use the complete CIFAR-10 dataset for the Kaggle competition. Change the `batch_size` and number of epochs `num_epochs` to 128 and 100, respectively. See what accuracy and ranking you can achieve in this competition.
2. What accuracy can you achieve when not using image augmentation?
3. Scan the QR code to access the relevant discussions and exchange ideas about the methods used and the results obtained with the community. Can you come up with any better techniques?

Discussions[203]

[203] https://discuss.d2l.ai/t/379

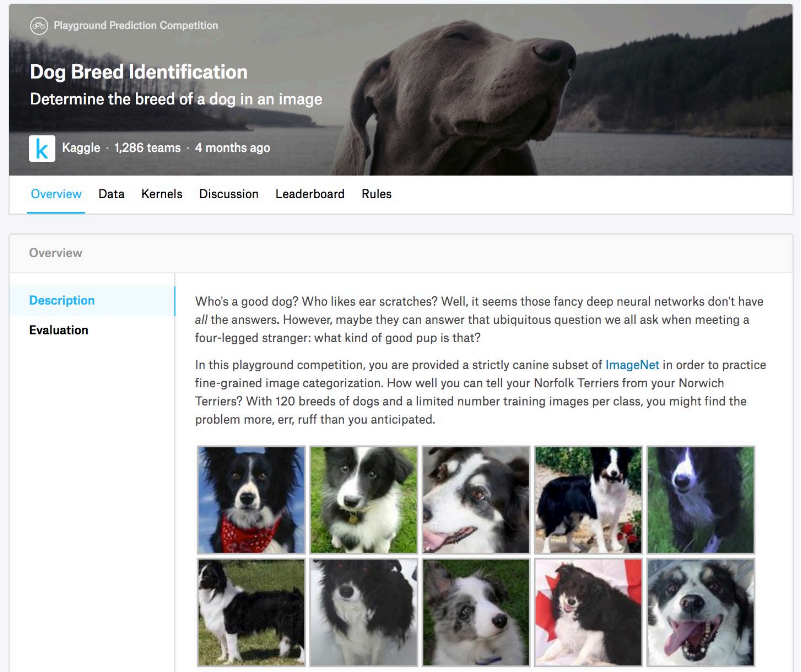
13.14 Dog Breed Identification (ImageNet Dogs) on Kaggle

In this section, we will tackle the dog breed identification challenge in the Kaggle Competition. The competition's web address is

> https://www.kaggle.com/c/dog-breed-identification

In this competition, we attempt to identify 120 different breeds of dogs. The dataset used in this competition is actually a subset of the famous ImageNet dataset. Different from the images in the CIFAR-10 dataset used in the previous section, the images in the ImageNet dataset are higher and wider and their dimensions are inconsistent.

Fig. 13.14.1 shows the information on the competition's webpage. In order to submit the results, please register an account on the Kaggle website first.


Playground Prediction Competition
Dog Breed Identification
Determine the breed of a dog in an image
Kaggle · 1,286 teams · 4 months ago
Overview
Data
Kernels
Discussion
Leaderboard
Rules
Overview
Description
Evaluation

Who's a good dog? Who likes ear scratches? Well, it seems those fancy deep neural networks don't have *all* the answers. However, maybe they can answer that ubiquitous question we all ask when meeting a four-legged stranger: what kind of good pup is that?

In this playground competition, you are provided a strictly canine subset of ImageNet in order to practice fine-grained image categorization. How well you can tell your Norfolk Terriers from your Norwich Terriers? With 120 breeds of dogs and a limited number training images per class, you might find the problem more, err, ruff than you anticipated.

Fig. 13.14.1: Dog breed identification competition website. The dataset for the competition can be accessed by clicking the "Data" tab.

First, import the packages or modules required for the competition.

```
import collections
from d2l import mxnet as d2l
import math
from mxnet import autograd, gluon, init, npx
from mxnet.gluon import nn
import os
import time

npx.set_np()
```

13.14.1 Obtaining and Organizing the Dataset

The competition data is divided into a training set and testing set. The training set contains $10,222$ images and the testing set contains $10,357$ images. The images in both sets are in JPEG format. These images contain three RGB channels (color) and they have different heights and widths. There are 120 breeds of dogs in the training set, including Labradors, Poodles, Dachshunds, Samoyeds, Huskies, Chihuahuas, and Yorkshire Terriers.

Downloading the Dataset

After logging in to Kaggle, we can click on the "Data" tab on the dog breed identification competition webpage shown in Fig. 13.14.1 and download the dataset by clicking the "Download All" button. After unzipping the downloaded file in `../data`, you will find the entire dataset in the following paths:

- ../data/dog-breed-identification/labels.csv
- ../data/dog-breed-identification/sample_submission.csv
- ../data/dog-breed-identification/train
- ../data/dog-breed-identification/test

You may have noticed that the above structure is quite similar to that of the CIFAR-10 competition in Section 13.13, where folders `train/` and `test/` contain training and testing dog images respectively, and `labels.csv` has the labels for the training images.

Similarly, to make it easier to get started, we provide a small-scale sample of the dataset mentioned above, "train_valid_test_tiny.zip". If you are going to use the full dataset for the Kaggle competition, you will also need to change the `demo` variable below to `False`.

```
#@save
d2l.DATA_HUB['dog_tiny'] = (d2l.DATA_URL + 'kaggle_dog_tiny.zip',
                            '0cb91d09b814ecdc07b50f31f8dcad3e81d6a86d')

# If you use the full dataset downloaded for the Kaggle competition, change
# the variable below to False
demo = True
if demo:
    data_dir = d2l.download_extract('dog_tiny')
else:
    data_dir = os.path.join('..', 'data', 'dog-breed-identification')
```

```
Downloading ../data/kaggle_dog_tiny.zip from http://d2l-data.s3-accelerate.amazonaws.com/
↪kaggle_dog_tiny.zip...
```

Organizing the Dataset

We can organize the dataset similarly to what we did in Section 13.13, namely separating a validation set from the training set, and moving images into subfolders grouped by labels.

The reorg_dog_data function below is used to read the training data labels, segment the validation set, and organize the training set.

```
def reorg_dog_data(data_dir, valid_ratio):
    labels = d2l.read_csv_labels(os.path.join(data_dir, 'labels.csv'))
    d2l.reorg_train_valid(data_dir, labels, valid_ratio)
    d2l.reorg_test(data_dir)


batch_size = 4 if demo else 128
valid_ratio = 0.1
reorg_dog_data(data_dir, valid_ratio)
```

13.14.2 Image Augmentation

The size of the images in this section are larger than the images in the previous section. Here are some more image augmentation operations that might be useful.

```
transform_train = gluon.data.vision.transforms.Compose([
    # Randomly crop the image to obtain an image with an area of 0.08 to 1 of
    # the original area and height to width ratio between 3/4 and 4/3. Then,
    # scale the image to create a new image with a height and width of 224
    # pixels each
    gluon.data.vision.transforms.RandomResizedCrop(224, scale=(0.08, 1.0),
                                                   ratio=(3.0/4.0, 4.0/3.0)),
    gluon.data.vision.transforms.RandomFlipLeftRight(),
    # Randomly change the brightness, contrast, and saturation
    gluon.data.vision.transforms.RandomColorJitter(brightness=0.4,
                                                   contrast=0.4,
                                                   saturation=0.4),
    # Add random noise
    gluon.data.vision.transforms.RandomLighting(0.1),
    gluon.data.vision.transforms.ToTensor(),
    # Standardize each channel of the image
    gluon.data.vision.transforms.Normalize([0.485, 0.456, 0.406],
                                           [0.229, 0.224, 0.225])])
```

During testing, we only use definite image preprocessing operations.

```
transform_test = gluon.data.vision.transforms.Compose([
    gluon.data.vision.transforms.Resize(256),
    # Crop a square of 224 by 224 from the center of the image
    gluon.data.vision.transforms.CenterCrop(224),
    gluon.data.vision.transforms.ToTensor(),
    gluon.data.vision.transforms.Normalize([0.485, 0.456, 0.406],
                                           [0.229, 0.224, 0.225])])
```

13.14.3 Reading the Dataset

As in the previous section, we can create an ImageFolderDataset instance to read the dataset containing the original image files.

```
train_ds, valid_ds, train_valid_ds, test_ds = [
    gluon.data.vision.ImageFolderDataset(
        os.path.join(data_dir, 'train_valid_test', folder))
    for folder in ('train', 'valid', 'train_valid', 'test')]
```

Here, we create DataLoader instances, just like in Section 13.13.

```
train_iter, train_valid_iter = [gluon.data.DataLoader(
    dataset.transform_first(transform_train), batch_size, shuffle=True,
    last_batch='discard') for dataset in (train_ds, train_valid_ds)]

valid_iter = gluon.data.DataLoader(
    valid_ds.transform_first(transform_test), batch_size, shuffle=False,
    last_batch='discard')

test_iter = gluon.data.DataLoader(
    test_ds.transform_first(transform_test), batch_size, shuffle=False,
    last_batch='keep')
```

13.14.4 Defining the Model

The dataset for this competition is a subset of the ImageNet data set. Therefore, we can use the approach discussed in Section 13.2 to select a model pre-trained on the entire ImageNet dataset and use it to extract image features to be input in the custom small-scale output network. Gluon provides a wide range of pre-trained models. Here, we will use the pre-trained ResNet-34 model. Because the competition dataset is a subset of the pre-training dataset, we simply reuse the input of the pre-trained model's output layer, i.e., the extracted features. Then, we can replace the original output layer with a small custom output network that can be trained, such as two fully connected layers in a series. Different from the experiment in Section 13.2, here, we do not retrain the pre-trained model used for feature extraction. This reduces the training time and the memory required to store model parameter gradients.

You must note that, during image augmentation, we use the mean values and standard deviations of the three RGB channels for the entire ImageNet dataset for normalization. This is consistent with the normalization of the pre-trained model.

```
def get_net(devices):
    finetune_net = gluon.model_zoo.vision.resnet34_v2(pretrained=True)
    # Define a new output network
    finetune_net.output_new = nn.HybridSequential(prefix='')
    finetune_net.output_new.add(nn.Dense(256, activation='relu'))
    # There are 120 output categories
    finetune_net.output_new.add(nn.Dense(120))
    # Initialize the output network
    finetune_net.output_new.initialize(init.Xavier(), ctx=devices)
    # Distribute the model parameters to the CPUs or GPUs used for computation
    finetune_net.collect_params().reset_ctx(devices)
    return finetune_net
```

When calculating the loss, we first use the member variable features to obtain the input of the pre-trained model's output layer, i.e., the extracted feature. Then, we use this feature as the input for our small custom output network and compute the output.

```
loss = gluon.loss.SoftmaxCrossEntropyLoss()

def evaluate_loss(data_iter, net, devices):
    l_sum, n = 0.0, 0
    for features, labels in data_iter:
        X_shards, y_shards = d2l.split_batch(features, labels, devices)
        output_features = [net.features(X_shard) for X_shard in X_shards]
        outputs = [net.output_new(feature) for feature in output_features]
        ls = [loss(output, y_shard).sum() for output, y_shard
              in zip(outputs, y_shards)]
        l_sum += sum([float(l.sum()) for l in ls])
        n += labels.size
    return l_sum / n
```

13.14.5 Defining the Training Functions

We will select the model and tune hyperparameters according to the model's performance on the validation set. The model training function train only trains the small custom output network.

```
def train(net, train_iter, valid_iter, num_epochs, lr, wd, devices, lr_period,
          lr_decay):
    # Only train the small custom output network
    trainer = gluon.Trainer(net.output_new.collect_params(), 'sgd',
                            {'learning_rate': lr, 'momentum': 0.9, 'wd': wd})
    num_batches, timer = len(train_iter), d2l.Timer()
    animator = d2l.Animator(xlabel='epoch', xlim=[0, num_epochs],
                            legend=['train loss', 'valid loss'])
    for epoch in range(num_epochs):
        metric = d2l.Accumulator(2)
        if epoch > 0 and epoch % lr_period == 0:
            trainer.set_learning_rate(trainer.learning_rate * lr_decay)
        for i, (features, labels) in enumerate(train_iter):
            timer.start()
            X_shards, y_shards = d2l.split_batch(features, labels, devices)
            output_features = [net.features(X_shard) for X_shard in X_shards]
            with autograd.record():
                outputs = [net.output_new(feature) for feature in output_features]
                ls = [loss(output, y_shard).sum() for output, y_shard
                      in zip(outputs, y_shards)]
            for l in ls:
                l.backward()
            trainer.step(batch_size)
            metric.add(sum([float(l.sum()) for l in ls]), labels.shape[0])
            timer.stop()
            if (i + 1) % (num_batches // 5) == 0:
                animator.add(epoch + i / num_batches,
                             (metric[0] / metric[1], None))
        if valid_iter is not None:
            valid_loss = evaluate_loss(valid_iter, net, devices)
            animator.add(epoch + 1, (None, valid_loss))
```

(continues on next page)

(continued from previous page)

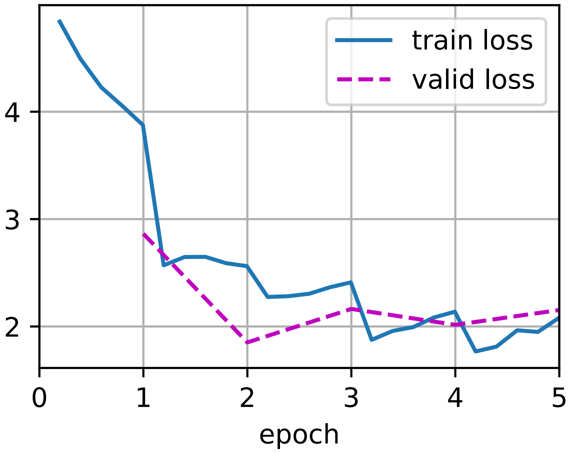
```
    if valid_iter is not None:
        print(f'train loss {metric[0] / metric[1]:.3f}, '
              f'valid loss {valid_loss:.3f}')
    else:
        print(f'train loss {metric[0] / metric[1]:.3f}')
    print(f'{metric[1] * num_epochs / timer.sum():.1f} examples/sec '
          f'on {str(devices)}')
```

13.14.6 Training and Validating the Model

Now, we can train and validate the model. The following hyperparameters can be tuned. For example, we can increase the number of epochs. Because `lr_period` and `lr_decay` are set to 10 and 0.1 respectively, the learning rate of the optimization algorithm will be multiplied by 0.1 after every 10 epochs.

```
devices, num_epochs, lr, wd = d2l.try_all_gpus(), 5, 0.01, 1e-4
lr_period, lr_decay, net = 10, 0.1, get_net(devices)
net.hybridize()
train(net, train_iter, valid_iter, num_epochs, lr, wd, devices, lr_period,
      lr_decay)
```

```
train loss 2.080, valid loss 2.153
87.2 examples/sec on [gpu(0), gpu(1)]
```



13.14.7 Classifying the Testing Set and Submitting Results on Kaggle

After obtaining a satisfactory model design and hyperparameters, we use all training datasets (including validation sets) to retrain the model and then classify the testing set. Note that predictions are made by the output network we just trained.

```
net = get_net(devices)
net.hybridize()
train(net, train_valid_iter, None, num_epochs, lr, wd, devices, lr_period,
```

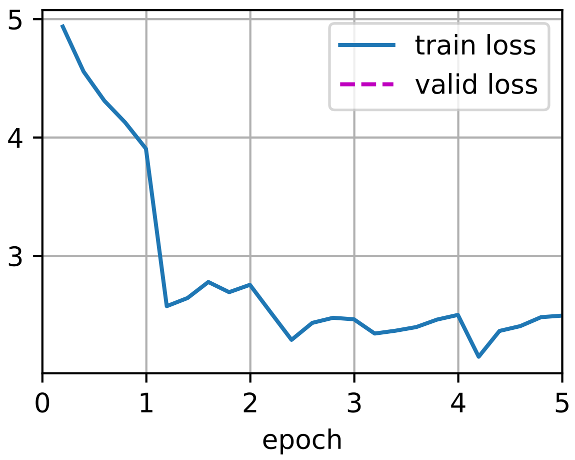
(continues on next page)

(continued from previous page)

```
      lr_decay)

preds = []
for data, label in test_iter:
    output_features = net.features(data.as_in_ctx(devices[0]))
    output = npx.softmax(net.output_new(output_features))
    preds.extend(output.asnumpy())
ids = sorted(os.listdir(
    os.path.join(data_dir, 'train_valid_test', 'test', 'unknown')))
with open('submission.csv', 'w') as f:
    f.write('id,' + ','.join(train_valid_ds.synsets) + '\n')
    for i, output in zip(ids, preds):
        f.write(i.split('.')[0] + ',' + ','.join(
            [str(num) for num in output]) + '\n')
```

```
train loss 2.495
89.8 examples/sec on [gpu(0), gpu(1)]
```



After executing the above code, we will generate a "submission.csv" file. The format of this file is consistent with the Kaggle competition requirements. The method for submitting results is similar to method in Section 4.10.

Summary

- We can use a model pre-trained on the ImageNet dataset to extract features and only train a small custom output network. This will allow us to classify a subset of the ImageNet dataset with lower computing and storage overhead.

Exercises

1. When using the entire Kaggle dataset, what kind of results do you get when you increase the `batch_size` (batch size) and `num_epochs` (number of epochs)?
2. Do you get better results if you use a deeper pre-trained model?
3. Scan the QR code to access the relevant discussions and exchange ideas about the methods used and the results obtained with the community. Can you come up with any better techniques?

Discussions[204]

[204] https://discuss.d2l.ai/t/380