7.1

In [1]:

```
import torch
from torch import nn
from d2l import torch as d2l
```

In [2]:

```
def corr2d(X, K):
    h, w = K.shape
    Y = torch.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            Y[i, j] = (X[i:i + h, j:j + w] * K).sum()
    return Y
```

In [3]:

```
X = torch.tensor([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
K = torch.tensor([[0.0, 1.0], [2.0, 3.0]])
corr2d(X, K)
```

Out[3]:

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

In [4]:

```
class Conv2D(nn.Module):
    def __init__(self, kernel_size):
        super().__init__()
        self.weight = nn.Parameter(torch.rand(kernel_size))
        self.bias = nn.Parameter(torch.zeros(1))

    def forward(self, x):
        return corr2d(x, self.weight) + self.bias
```

In [5]:

```
X = torch.ones((6, 8))
X[:, 2:6] = 0
X
```

Out[5]:

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

In [6]:

```
K = torch.tensor([[1.0, -1.0]])
```

In [7]:

```
Y = corr2d(X, K)
Y
```

Out[7]:

```
tensor([[ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.]])
```

In [8]:

```
corr2d(X.t(), K)
```

Out[8]:

```
tensor([[0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]])
```

In [9]:

```
conv2d = nn.LazyConv2d(1, kernel_size=(1, 2), bias=False)


X = X.reshape((1, 1, 6, 8))
Y = Y.reshape((1, 1, 6, 7))
lr = 3e-2

for i in range(10):
    Y_hat = conv2d(X)
    l = (Y_hat - Y) ** 2
    conv2d.zero_grad()
    l.sum().backward()
    conv2d.weight.data[:] -= lr * conv2d.weight.grad
    if (i + 1) % 2 == 0:
        print(f'epoch {i + 1}, loss {l.sum():.3f}')
```

```
epoch 2, loss 6.495
epoch 4, loss 1.915
epoch 6, loss 0.659
epoch 8, loss 0.249
epoch 10, loss 0.098
```

```
C:\Users\papib\miniconda3\envs\d2l\lib\site-packages\torch\nn\modules\lazy.py:18
0: UserWarning: Lazy modules are a new feature under heavy development so changes
to the API or functionality can happen at any moment.
  warnings.warn('Lazy modules are a new feature under heavy development '
```

In [10]:

```
conv2d.weight.data.reshape((1, 2))
```

Out[10]:

```
tensor([[ 0.9589, -1.0229]])
```

7.3

In [11]:

```
def comp_conv2d(conv2d, X):

    X = X.reshape((1, 1) + X.shape)
    Y = conv2d(X)
    return Y.reshape(Y.shape[2:])

conv2d = nn.Conv2d(1, 1, kernel_size=3, padding=1)
X = torch.rand(size=(8, 8))
comp_conv2d(conv2d, X).shape
```

Out[11]:

```
torch.Size([8, 8])
```

In [12]:

```
conv2d = nn.Conv2d(1, 1, kernel_size=(5, 3), padding=(2, 1))
comp_conv2d(conv2d, X).shape
```

```
Out[12]: torch.Size([8, 8])
```

```
In [13]: conv2d = nn.Conv2d(1, 1, kernel_size=3, padding=1, stride=2)
         comp_conv2d(conv2d, X).shape
```

```
Out[13]: torch.Size([4, 4])
```

```
In [14]: conv2d = nn.Conv2d(1, 1, kernel_size=(3, 5), padding=(0, 1), stride=(3, 4))
         comp_conv2d(conv2d, X).shape
```

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

7.4

```
In [15]: def corr2d_multi_in(X, K):
             return sum(d2l.corr2d(x, k) for x, k in zip(X, K))
```

```
In [16]: X = torch.tensor([[[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]],
                           [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]]])
         K = torch.tensor([[[0.0, 1.0], [2.0, 3.0]], [[1.0, 2.0], [3.0, 4.0]]])

         corr2d_multi_in(X, K)
```

```
Out[16]: tensor([[ 56.,  72.],
                 [104., 120.]])
```

```
In [17]: def corr2d_multi_in_out(X, K):

             return torch.stack([corr2d_multi_in(X, k) for k in K], 0)
```

```
In [18]: K = torch.stack((K, K + 1, K + 2), 0)
         K.shape
```

```
Out[18]: torch.Size([3, 2, 2, 2])
```

```
In [19]: corr2d_multi_in_out(X, K)
```

```
Out[19]: tensor([[[ 56.,  72.],
                  [104., 120.]],

                 [[ 76., 100.],
                  [148., 172.]],

                 [[ 96., 128.],
                  [192., 224.]]])
```

```
In [20]: def corr2d_multi_in_out_1x1(X, K):
             c_i, h, w = X.shape
             c_o = K.shape[0]
             X = X.reshape((c_i, h * w))
             K = K.reshape((c_o, c_i))

             Y = torch.matmul(K, X)
             return Y.reshape((c_o, h, w))
```

In [21]:

```
X = torch.normal(0, 1, (3, 3, 3))
K = torch.normal(0, 1, (2, 3, 1, 1))
Y1 = corr2d_multi_in_out_1x1(X, K)
Y2 = corr2d_multi_in_out(X, K)
assert float(torch.abs(Y1 - Y2).sum()) < 1e-6
```

7.5

In [22]:

```
def pool2d(X, pool_size, mode='max'):
    p_h, p_w = pool_size
    Y = torch.zeros((X.shape[0] - p_h + 1, X.shape[1] - p_w + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            if mode == 'max':
                Y[i, j] = X[i: i + p_h, j: j + p_w].max()
            elif mode == 'avg':
                Y[i, j] = X[i: i + p_h, j: j + p_w].mean()
    return Y
```

In [23]:

```
X = torch.tensor([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
pool2d(X, (2, 2))
```

Out[23]:

```
tensor([[4., 5.],
        [7., 8.]])
```

In [24]:

```
pool2d(X, (2, 2), 'avg')
```

Out[24]:

```
tensor([[2., 3.],
        [5., 6.]])
```

In [25]:

```
X = torch.arange(16, dtype=torch.float32).reshape((1, 1, 4, 4))
X
```

Out[25]:

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

In [26]:

```
pool2d = nn.MaxPool2d(3)
pool2d(X)
```

Out[26]:

```
tensor([[[[10.]]]])
```

In [27]:

```
pool2d = nn.MaxPool2d(3, padding=1, stride=2)
pool2d(X)
```

Out[27]:

```
tensor([[[[ 5.,  7.],
          [13., 15.]]]])
```

In [28]:

```
pool2d = nn.MaxPool2d((2, 3), stride=(2, 3), padding=(0, 1))
pool2d(X)
```

Out[28]:

```
tensor([[[[ 5.,  7.],
          [13., 15.]]]])
```

In [29]:

```
X = torch.cat((X, X + 1), 1)
```

```
X
```

Out[29]:

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

         [[ 1.,  2.,  3.,  4.],
          [ 5.,  6.,  7.,  8.],
          [ 9., 10., 11., 12.],
          [13., 14., 15., 16.]]]])
```

In [30]:

```
pool2d = nn.MaxPool2d(3, padding=1, stride=2)
pool2d(X)
```

Out[30]:

```
tensor([[[[ 5.,  7.],
          [13., 15.]],

         [[ 6.,  8.],
          [14., 16.]]]])
```

7.6

In [31]:

```
def init_cnn(module):

    if type(module) == nn.Linear or type(module) == nn.Conv2d:
        nn.init.xavier_uniform_(module.weight)

class LeNet(d2l.Classifier):

    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.LazyConv2d(6, kernel_size=5, padding=2), nn.Sigmoid(),
            nn.AvgPool2d(kernel_size=2, stride=2),
            nn.LazyConv2d(16, kernel_size=5), nn.Sigmoid(),
            nn.AvgPool2d(kernel_size=2, stride=2),
            nn.Flatten(),
            nn.LazyLinear(120), nn.Sigmoid(),
            nn.LazyLinear(84), nn.Sigmoid(),
            nn.LazyLinear(num_classes))
```

In [32]:

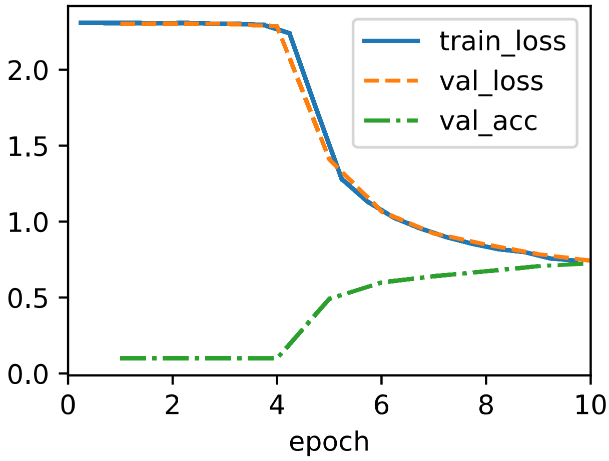
```
@d2l.add_to_class(d2l.Classifier)  #@save
def layer_summary(self, X_shape):
    X = torch.randn(*X_shape)
    for layer in self.net:
        X = layer(X)
        print(layer.__class__.__name__, 'output shape:\t', X.shape)

model = LeNet()
model.layer_summary((1, 1, 28, 28))
```

```
Conv2d output shape:     torch.Size([1, 6, 28, 28])
Sigmoid output shape:    torch.Size([1, 6, 28, 28])
AvgPool2d output shape:  torch.Size([1, 6, 14, 14])
Conv2d output shape:     torch.Size([1, 16, 10, 10])
Sigmoid output shape:    torch.Size([1, 16, 10, 10])
AvgPool2d output shape:  torch.Size([1, 16, 5, 5])
Flatten output shape:    torch.Size([1, 400])
Linear output shape:     torch.Size([1, 120])
Sigmoid output shape:    torch.Size([1, 120])
Linear output shape:     torch.Size([1, 84])
Sigmoid output shape:    torch.Size([1, 84])
Linear output shape:     torch.Size([1, 10])
```

In [33]:

```
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128)
model = LeNet(lr=0.1)
model.apply_init([next(iter(data.get_dataloader(True)))[0]], init_cnn)
trainer.fit(model, data)
```



8.2

In [34]:

```
def vgg_block(num_convs, out_channels):
    layers = []
    for _ in range(num_convs):
        layers.append(nn.LazyConv2d(out_channels, kernel_size=3, padding=1))
        layers.append(nn.ReLU())
    layers.append(nn.MaxPool2d(kernel_size=2,stride=2))
    return nn.Sequential(*layers)
```

In [35]:

```
class VGG(d2l.Classifier):
    def __init__(self, arch, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        conv_blks = []
        for (num_convs, out_channels) in arch:
            conv_blks.append(vgg_block(num_convs, out_channels))
        self.net = nn.Sequential(
            *conv_blks, nn.Flatten(),
            nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(0.5),
            nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(0.5),
            nn.LazyLinear(num_classes))
        self.net.apply(d2l.init_cnn)
```

In [36]:

```
VGG(arch=((1, 64), (1, 128), (2, 256), (2, 512), (2, 512))).layer_summary(
    (1, 1, 224, 224))
```

```
Sequential output shape:         torch.Size([1, 64, 112, 112])
Sequential output shape:         torch.Size([1, 128, 56, 56])
Sequential output shape:         torch.Size([1, 256, 28, 28])
Sequential output shape:         torch.Size([1, 512, 14, 14])
Sequential output shape:         torch.Size([1, 512, 7, 7])
Flatten output shape:    torch.Size([1, 25088])
Linear output shape:     torch.Size([1, 4096])
ReLU output shape:       torch.Size([1, 4096])
Dropout output shape:    torch.Size([1, 4096])
Linear output shape:     torch.Size([1, 4096])
ReLU output shape:       torch.Size([1, 4096])
Dropout output shape:    torch.Size([1, 4096])
Linear output shape:     torch.Size([1, 10])
```

In [37]:

```
model = VGG(arch=((1, 16), (1, 32), (2, 64), (2, 128), (2, 128)), lr=0.01)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(224, 224))
```

8.6

In [38]:

```
from torch.nn import functional as F
```

In [39]:

```
class Residual(nn.Module):  #@save
    """The Residual block of ResNet models."""
    def __init__(self, num_channels, use_1x1conv=False, strides=1):
        super().__init__()
        self.conv1 = nn.LazyConv2d(num_channels, kernel_size=3, padding=1,
                                   stride=strides)
        self.conv2 = nn.LazyConv2d(num_channels, kernel_size=3, padding=1)
        if use_1x1conv:
            self.conv3 = nn.LazyConv2d(num_channels, kernel_size=1,
                                       stride=strides)
        else:
            self.conv3 = None
        self.bn1 = nn.LazyBatchNorm2d()
        self.bn2 = nn.LazyBatchNorm2d()

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

In [40]:

```
blk = Residual(3)
X = torch.randn(4, 3, 6, 6)
blk(X).shape
```

Out[40]:

```
torch.Size([4, 3, 6, 6])
```

In [41]:

```
blk = Residual(6, use_1x1conv=True, strides=2)
blk(X).shape
```

```
Out[41]: torch.Size([4, 6, 3, 3])
```

```
In [42]: class ResNet(d2l.Classifier):
             def b1(self):
                 return nn.Sequential(
                     nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),
                     nn.LazyBatchNorm2d(), nn.ReLU(),
                     nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

```
In [43]: @d2l.add_to_class(ResNet)
         def block(self, num_residuals, num_channels, first_block=False):
             blk = []
             for i in range(num_residuals):
                 if i == 0 and not first_block:
                     blk.append(Residual(num_channels, use_1x1conv=True, strides=2))
                 else:
                     blk.append(Residual(num_channels))
             return nn.Sequential(*blk)
```

```
In [44]: @d2l.add_to_class(ResNet)
         def block(self, num_residuals, num_channels, first_block=False):
             blk = []
             for i in range(num_residuals):
                 if i == 0 and not first_block:
                     blk.append(Residual(num_channels, use_1x1conv=True, strides=2))
                 else:
                     blk.append(Residual(num_channels))
             return nn.Sequential(*blk)
```

```
In [50]: class ResNet18(ResNet):
             def __init__(self, lr=0.1, num_classes=10):
                 super().__init__(((2, 64), (2, 128), (2, 256), (2, 512)),
                                lr, num_classes)
```