Deep Learning HW 2

Chapter 7 Convolutional Neural Networks

7.1 From Fully Connected Layers to Convolutions

Some Exercises of 7.1 in the d2l book

1. if the size of convolution kernel is 1 by 1 (delta = 0), the result from convolutional layer is weighted sum of each kernel per each image pixel by definition.
2. There could be some cases where location contains important information, such as distance from the ground.
3. Text is a sequential data and understanding context is an important task. convolutional layer doesn't consider that point.
4. Convolutional layer usually skips the edge pixels or uses zero-padding.
5. by changing of variable in integral to t = x-z , two operation result in same.

7.2. Convolutions for Images

In [1]:
```python
import torch
from torch import nn
from d2l import torch as d2l
```

In [2]:
```python
def corr2d(X, K):
    """Compute 2D cross-correlation."""
    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]:
```python
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]:
```python
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 % 2 == 1:
        print(f"epoch{i + 1}, loss {l.sum():.3f}")
```

```
epoch2, loss 17.034
epoch4, loss 5.666
epoch6, loss 2.101
epoch8, loss 0.824
epoch10, loss 0.331
```

```
c:\Users\John\.conda\envs\d2l\lib\site-packages\torch\nn\modules\lazy.py:180: Use
rWarning: Lazy modules are a new feature under heavy development so changes to th
e 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([[ 1.0469, -0.9288]])
```

Some Exercises of 7.2 in the d2l book

In [11]:
```
#ex 1
X = torch.zeros((7, 7))
for i in range(7):
    X[i, i] = 1
Y = corr2d(X, K)
print(corr2d(X, K), corr2d(X.T, K), corr2d(X, K.T))
```

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

In [12]:
```
#ex 3

class Conv2D_book(nn.Module):
    def __init__(self, kernel_size):
        super().__init__()
        self.weight = nn.Parameter(torch.rand(kernel_size), requires_grad=True)
        self.bias = nn.Parameter(torch.zeros(1), requires_grad=True)

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


X = torch.ones((6, 8))
X[:, 2:6] = 0
model = Conv2D_book((1, 2))
Y = corr2d(X, K)
lr = 1e-1

optimizer = torch.optim.Adam(model.parameters(), lr=lr)
for i in range(40):
    Y_hat = model(X)
    l = ((Y_hat - Y) ** 2).sum()
    l.backward()
    optimizer.step()
    optimizer.zero_grad()
    if i % 5 == 0:
        print(f"epoch{i}, loss {l:.3f}")
print(model.weight, K)
```

```
epoch0, loss 34.585
epoch5, loss 17.732
epoch10, loss 8.502
epoch15, loss 3.350
epoch20, loss 0.708
epoch25, loss 0.218
epoch30, loss 0.274
epoch35, loss 0.522
Parameter containing:
tensor([[ 1.1348, -1.2021]], requires_grad=True) tensor([[ 1., -1.]])
```

7.3 Padding and Stride

In [13]:
```
import torch
from torch import nn
```

In [14]:
```
def comp_conv2d(conv2d, X):
    X = X.reshape((1, 1) + X.shape)
    Y = conv2d(X)
    return Y.reshape(Y.shape[2:])

conv2d = nn.LazyConv2d(1, kernel_size=3, padding=1)
X = torch.rand(size=(8, 8))
comp_conv2d(conv2d, X).shape
```

Out[14]:
```
torch.Size([8, 8])
```

In [15]:
```
conv2d = nn.LazyConv2d(1, kernel_size=(5, 3), padding=(2, 1))
comp_conv2d(conv2d, X).shape
```

Out[15]:
```
torch.Size([8, 8])
```

In [16]:
```
conv2d = nn.LazyConv2d(1, kernel_size=3, padding=1, stride=2)
comp_conv2d(conv2d, X).shape
```

Out[16]:
```
torch.Size([4, 4])
```

In [17]:
```
conv2d = nn.LazyConv2d(1, kernel_size=(3, 5), padding=(0, 1), stride=(3, 4))
comp_conv2d(conv2d, X).shape
```

Out[17]:
```
torch.Size([2, 2])
```

Some Exercises of 7.3 in the d2l book

1. (8-3)/3 + 1, (8-5+1)/4 + 1 = 2, 2
2. It reduces the image size.
3. It can be used to upsampling.

7.4 Multiple Input and Multiple Output Channels

In [18]:
```
import torch
from d2l import torch as d2l
```

In [19]:
```
def corr2d_multi_in(X, K):
    return sum(d2l.corr2d(x, k) for x, k in zip(X, K))
```

In [20]:
```
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[20]:
```
tensor([[ 56.,  72.],
        [104., 120.]])
```

In [21]:
```
def corr2d_multi_in_out(X, K):
    return torch.stack([corr2d_multi_in(X, k) for k in K], 0)
```

In [22]:
```
K = torch.stack((K, K + 1, K + 2), 0)
K.shape
```

Out[22]:
```
torch.Size([3, 2, 2, 2])
```

In [23]:
```
corr2d_multi_in_out(X, K)
```

Out[23]:
```
tensor([[[ 56.,  72.],
         [104., 120.]],

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

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

In [24]:
```
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 [26]:
```
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
```

Some Exercises of 7.4 in the d2l book

1.It can represented by kernel size of k1 times k2. Decomposition can be done if the kernel size is not prime. 2. approximate memory usage of conv layer is c_i * c_o * k * k 3. 4 and 4.

7.5 Pooling

In [27]:
```
import torch
from torch import nn
from d2l import torch as d2l
```

In [28]:
```
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 [29]:
```
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[29]:
```
tensor([[4., 5.],
        [7., 8.]])
```

In [30]:
```
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[30]:
```
tensor([[4., 5.],
        [7., 8.]])
```

In [31]:
```
X = torch.arange(16, dtype=torch.float32).reshape((1, 1, 4, 4))
X
```

Out[31]:
```
tensor([[[[ 0.,  1.,  2.,  3.],
          [ 4.,  5.,  6.,  7.],
          [ 8.,  9., 10., 11.],
          [12., 13., 14., 15.]]]])
```

In [32]:
```
pool2d = nn.MaxPool2d(3)
pool2d(X)
```

Out[32]:
```
tensor([[[[10.]]]])
```

In [33]:
```
pool2d = nn.MaxPool2d(3, padding=1, stride=2)
pool2d(X)
```

Out[33]:
```
tensor([[[[ 5.,  7.],
          [13., 15.]]]])
```

In [34]:
```
pool2d = nn.MaxPool2d((2, 3), stride=(2, 3), padding=(0, 1))
pool2d(X)
```

Out[34]:
```
tensor([[[[ 5.,  7.],
          [13., 15.]]]])
```

In [35]:
```
X = torch.cat((X, X + 1), 1)
X
```

Out[35]:
```
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 [36]:
```
pool2d = nn.MaxPool2d(3, padding=1, stride=2)
pool2d(X)
```

Out[36]:
```
tensor([[[[ 5.,  7.],
          [13., 15.]],

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

Some Exercises of 7.5 in the d2l book

1. By set weights to one over kernel size ** 2.
2. Max pooling is one type of nonlinearity, and convolution can't introduce nonlinerity.
3. If some pixel value is high, then two pooling method will give very different result.
4. Maybe it normalizes the values, and it requires more computation.

7.6 Convolutional Neural Networks

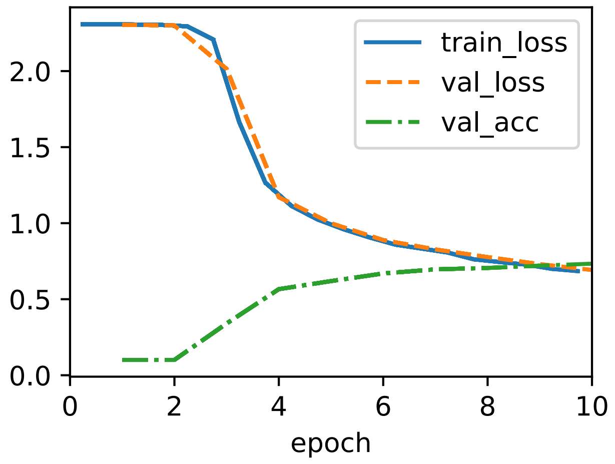
In [38]:
```
import torch
from torch import nn
from d2l import torch as d2l
```

In [39]:
```
def init_cnn(module):
    """Initialize weights for CNNs."""
    if type(module) == nn.Linear or type(module) == nn.Conv2d:
        nn.init.xavier_uniform_(module.weight)

class LeNet(d2l.Classifier):
    """The LeNet-5 model."""
    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 [40]:
```
@d2l.add_to_class(d2l.Classifier)
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])
```

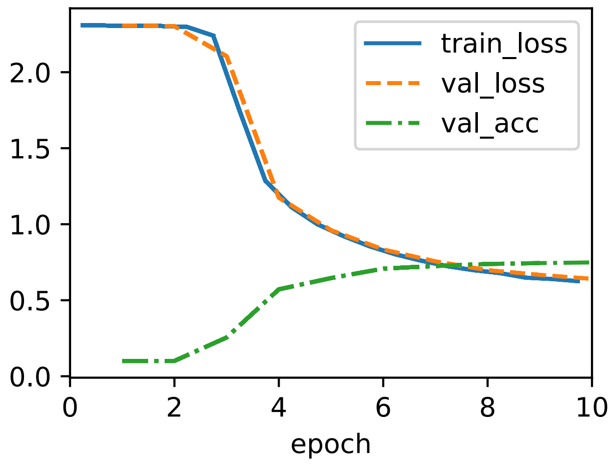
In [41]:
```
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128)
model = LeNet(lr=0.1)
model.apply_init([next(iter(data.get_dataloader(True)))[0]], init_cnn)
trainer.fit(model, data)
```

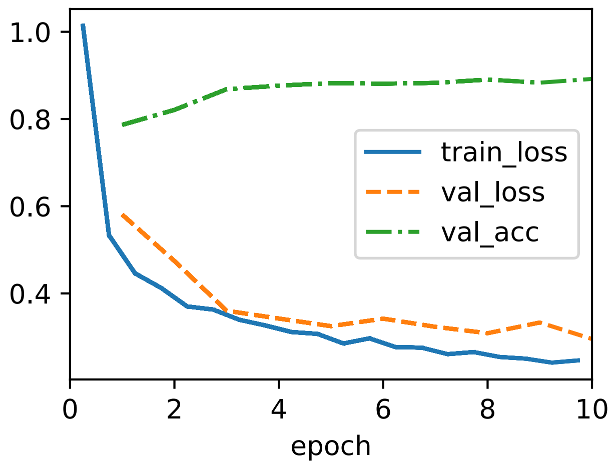


Some Exercises of 7.6 in the d2l book

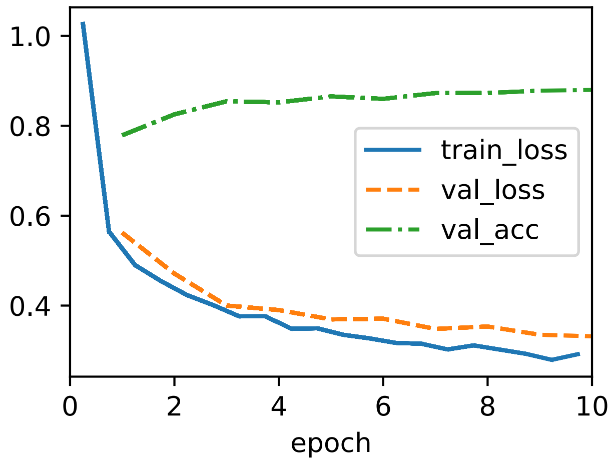
In [42]:
```
#ex1
class LeNet(d2l.Classifier):
    """The LeNet-5 model."""
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.LazyConv2d(6, kernel_size=5, padding=2), nn.Sigmoid(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.LazyConv2d(16, kernel_size=5), nn.Sigmoid(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Flatten(),
            nn.LazyLinear(120), nn.Sigmoid(),
            nn.LazyLinear(84), nn.Sigmoid(),
            nn.LazyLinear(num_classes))
```

```
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)

#Relu works much better, at least in terms of training time.
class LeNet(d2l.Classifier):
    """The LeNet-5 model."""
    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.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.LazyConv2d(16, kernel_size=5), nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Flatten(),
            nn.LazyLinear(120), nn.ReLU(),
            nn.LazyLinear(84), nn.ReLU(),
            nn.LazyLinear(num_classes))

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)
```

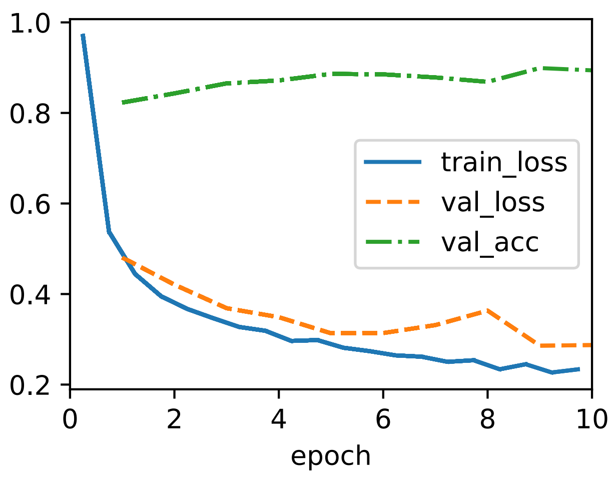




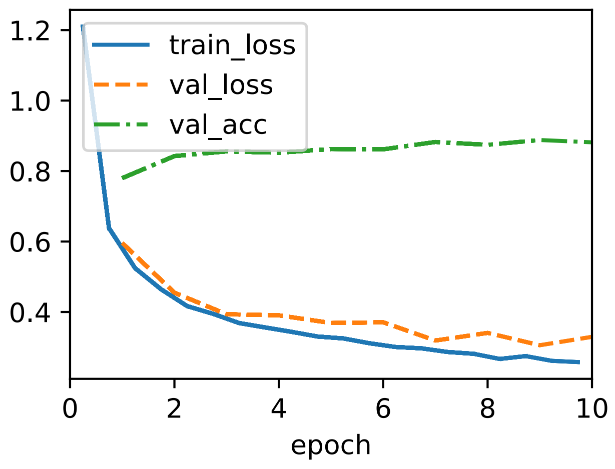
In [48]:
```
# ex 2
class LeNet(d2l.Classifier):
    """The LeNet-5 model."""
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.LazyConv2d(6, kernel_size=7, padding=2), nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.LazyConv2d(16, kernel_size=5), nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Flatten(),
            nn.LazyLinear(120), nn.ReLU(),
            nn.LazyLinear(84), nn.ReLU(),
            nn.LazyLinear(num_classes))

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)
```



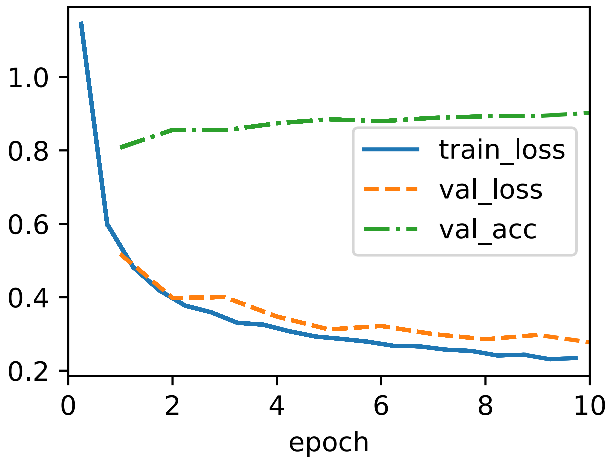
In [49]:
```
class LeNet(d2l.Classifier):
    """The LeNet-5 model."""
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.LazyConv2d(16, kernel_size=5, padding=2), nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.LazyConv2d(32, kernel_size=5), nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Flatten(),
            nn.LazyLinear(120), nn.ReLU(),
            nn.LazyLinear(84), nn.ReLU(),
            nn.LazyLinear(num_classes))

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)
```



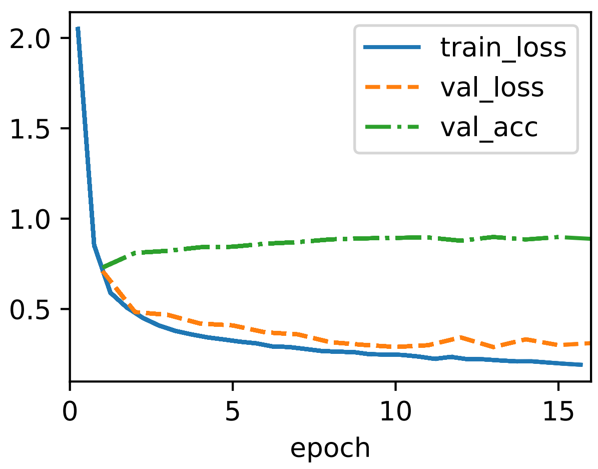
In [50]:
```python
class LeNet(d2l.Classifier):
    """The LeNet-5 model."""
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.LazyConv2d(16, kernel_size=3, padding=1), nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.LazyConv2d(32, kernel_size=3, padding=1), nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.LazyConv2d(64, kernel_size=3, padding=1), nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Flatten(),
            nn.LazyLinear(120), nn.ReLU(),
            nn.LazyLinear(84), nn.ReLU(),
            nn.LazyLinear(num_classes))

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)
```



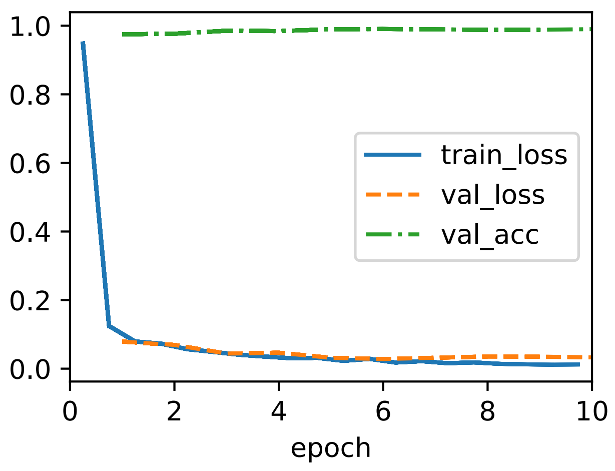
In [51]:
```python
class LeNet(d2l.Classifier):
    """The LeNet-5 model."""
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.LazyConv2d(16, kernel_size=3, padding=1), nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.LazyConv2d(32, kernel_size=3, padding=1), nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.LazyConv2d(64, kernel_size=3, padding=1), nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Flatten(),
            nn.LazyLinear(256), nn.ReLU(),
            nn.LazyLinear(64), nn.ReLU(),
            nn.LazyLinear(num_classes))

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)
```



In [52]:
```
class LeNet(d2l.Classifier):
    """The LeNet-5 model."""
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.LazyConv2d(16, kernel_size=3, padding=1), nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.LazyConv2d(32, kernel_size=3, padding=1), nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.LazyConv2d(64, kernel_size=3, padding=1), nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Flatten(),
            nn.LazyLinear(256), nn.ReLU(),
            nn.LazyLinear(64), nn.ReLU(),
            nn.LazyLinear(num_classes))

trainer = d2l.Trainer(max_epochs=16, num_gpus=1)
data = d2l.FashionMNIST(batch_size=256)
model = LeNet(lr=0.3)
model.apply_init([next(iter(data.get_dataloader(True)))[0]], init_cnn)
trainer.fit(model, data)
```



In [53]:
```python
import torchvision
from torchvision import transforms
class MNIST(d2l.DataModule):
    """The MNIST dataset."""
    def __init__(self, batch_size=64, resize=(28, 28)):
        super().__init__()
        self.save_hyperparameters()
        trans = transforms.Compose([transforms.Resize(resize),
                                    transforms.ToTensor()])
        self.train = torchvision.datasets.MNIST(
            root=self.root, train=True, transform=trans, download=True)
        self.val = torchvision.datasets.MNIST(
            root=self.root, train=False, transform=trans, download=True)
    def get_dataloader(self, train):
        data = self.train if train else self.val
        return torch.utils.data.DataLoader(data, self.batch_size, shuffle=train,

class LeNet(d2l.Classifier):
    """The LeNet-5 model."""
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.LazyConv2d(16, kernel_size=3, padding=1), nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.LazyConv2d(32, kernel_size=3, padding=1), nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.LazyConv2d(64, kernel_size=3, padding=1), nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Flatten(),
            nn.LazyLinear(256), nn.ReLU(),
            nn.LazyLinear(64), nn.ReLU(),
            nn.LazyLinear(num_classes))

trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = MNIST(batch_size=128)
model = LeNet(lr=0.2)
model.apply_init([next(iter(data.get_dataloader(True)))[0]], init_cnn)
trainer.fit(model, data)
```



Chapter 7 discussion

7.1 The important property of fully connected layer is that for single node weighted sum of every previous node after passing activation function is calculated. It is okay and useful when the number of node is small. But for images, since usual image pixel is around one million, more effective architecture is required. Convolutional layer is one possible solution to that, since transitional invariance and locality is conceptually implemented.

7.2 Conceptually, cross-correlation is more close to the operation done between kernel and image in convolutional layer. Basic filer can be manually crafted, which can be used to derive feature map.

7.3 the zero padding size to recover original image after conv layer is (F-1)/2 where F is kernel size. Also, when applying stride s, the image size of h with padding p result in (h – k + p) / s + 1.

7.4 Multiple channel allows to process color channels in images without processing them into gray scale, with more flexibility. Also, it allows to capture multiple feature maps by using more filters.

8. Modern Convolutional Neural Networks

8.2 Networks Using Blocks (VGG)

In [1]:
```
import torch
from torch import nn
from d2l import torch as d2l
torch.cuda.is_available()
```

Out[1]:
```
True
```

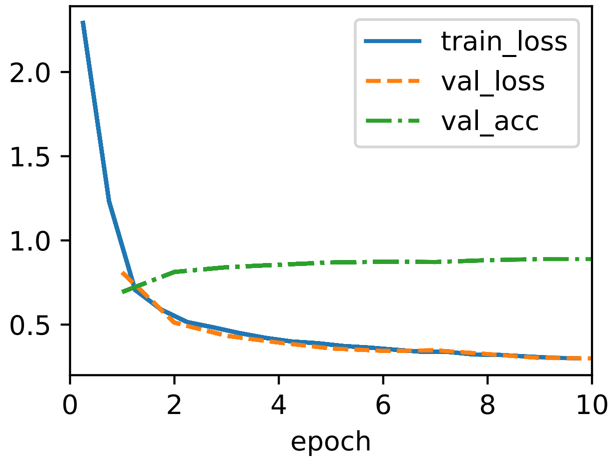
In [2]:
```
def vgg_block(num_convs, out_channels):
    layers = []
    for _ in range(num_convs):
        layers.append(nn.LazyConv2d(out_channels, kernel_size=3, padding=1))
        layers.append(nn.ReLU())
    layers.append(nn.MaxPool2d(kernel_size=2,stride=2))
    return nn.Sequential(*layers)
```

In [3]:
```
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 [4]:
```
VGG(arch=((1, 64), (1, 128), (2, 256), (2, 512), (2, 512))).layer_summary(
    (1, 1, 224, 224))
```

```
c:\ProgramData\anaconda3\envs\d2l\lib\site-packages\torch\nn\modules\lazy.py:180:
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 '
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])
```

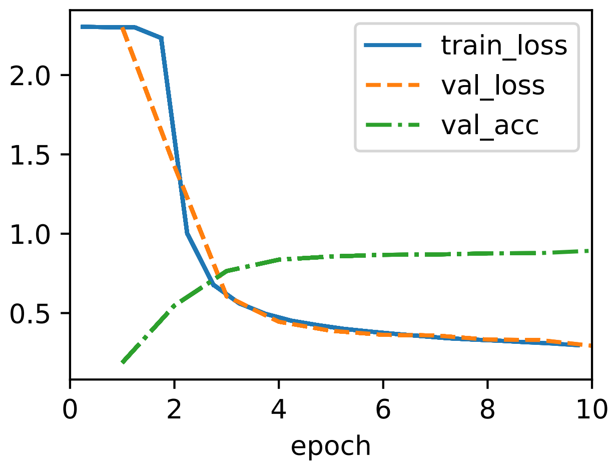
In [5]:
```
model = VGG(arch=((1, 16), (1, 32), (2, 64), (2, 128), (2, 128)), lr=0.01)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(224, 224))
model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
trainer.fit(model, data)
```



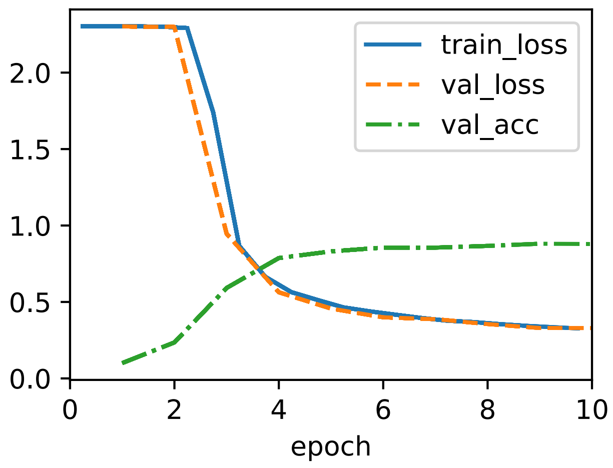
Some Exercises of 8.2 in the d2l book

2. Conv block is integrated into Sequential output.

In [6]:
```
#ex 3
class VGG16(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)

model = VGG16(arch=((2, 16), (2, 32), (3, 64), (3, 128), (3, 128)), lr=0.01)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(224, 224))
model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
trainer.fit(model, data)
```



In [7]:
```
class VGG19(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)

model = VGG19(arch=((2, 16), (2, 32), (4, 64), (4, 128), (4, 128)), lr=0.01)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(224, 224))
model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
trainer.fit(model, data)
```



8.6. Residual Networks (ResNet)

In [8]:
```
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l
```

In [9]:
```
class Residual(nn.Module):
    """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 [10]:
```
blk = Residual(3)
X = torch.randn(4, 3, 6, 6)
blk(X).shape
```

Out[10]:
```
torch.Size([4, 3, 6, 6])
```

In [11]:
```
blk = Residual(6, use_1x1conv=True, strides=2)
blk(X).shape
```

Out[11]:
```
torch.Size([4, 6, 3, 3])
```

In [12]:
```
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))
```

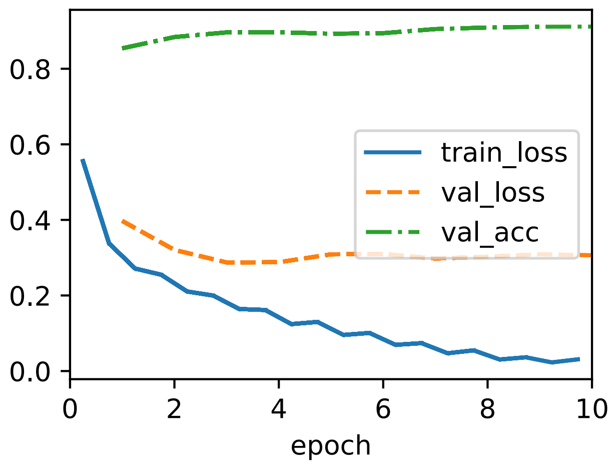
In [13]:
```
@d2l.add_to_class(ResNet)
def block(self, num_residuals, num_channels, first_block=False):
    blk = []
    for i in range(num_residuals):
        if i == 0 and not first_block:
            blk.append(Residual(num_channels, use_1x1conv=True, strides=2))
        else:
            blk.append(Residual(num_channels))
    return nn.Sequential(*blk)
```

In [14]:
```
@d2l.add_to_class(ResNet)
def __init__(self, arch, lr=0.1, num_classes=10):
    super(ResNet, self).__init__()
    self.save_hyperparameters()
    self.net = nn.Sequential(self.b1())
    for i, b in enumerate(arch):
        self.net.add_module(f'b{i+2}', self.block(*b, first_block=(i==0)))
    self.net.add_module('last', nn.Sequential(
        nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
        nn.LazyLinear(num_classes)))
    self.net.apply(d2l.init_cnn)
```

In [15]:
```
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)

ResNet18().layer_summary((1, 1, 96, 96))
```

```
Sequential output shape:         torch.Size([1, 64, 24, 24])
Sequential output shape:         torch.Size([1, 64, 24, 24])
Sequential output shape:         torch.Size([1, 128, 12, 12])
Sequential output shape:         torch.Size([1, 256, 6, 6])
Sequential output shape:         torch.Size([1, 512, 3, 3])
Sequential output shape:         torch.Size([1, 10])
```

In [16]:
```
model = ResNet18(lr=0.01)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
trainer.fit(model, data)
```

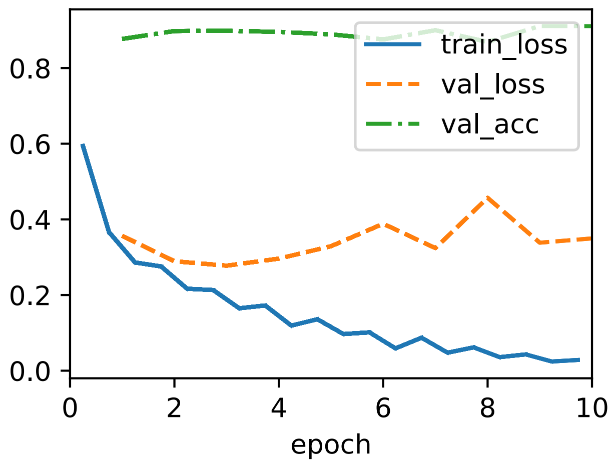


Some Exercises of 8.6 in the d2l book

5. Maybe because training large model has always not been a easy task.

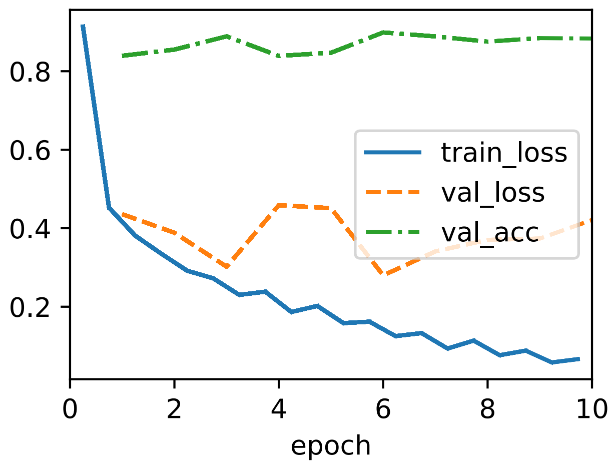
In [72]:
```
#ex1

class Residual(nn.Module):
    """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)

class Residual_bottleneck(nn.Module):
    """The Residual bottleneck 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.conv0_1 = nn.LazyConv2d(num_channels, kernel_size=1,
                                   stride=strides)
        self.conv0_2 = nn.LazyConv2d(num_channels * 4, kernel_size=1,
                                   stride=strides)
        if use_1x1conv:
            self.conv3 = nn.LazyConv2d(num_channels * 2, kernel_size=1,
                                       stride=strides)
        else:
            self.conv3 = None
        self.bn1 = nn.LazyBatchNorm2d()
        self.bn2 = nn.LazyBatchNorm2d()
        self.bn3 = nn.LazyBatchNorm2d()

    def forward(self, X):
        Y = F.relu(self.bn1(self.conv0_1(X)))
        Y = F.relu(self.bn2(self.conv1(X)))
        Y = self.bn3(self.conv0_2(X))
        if self.conv3:
            X = self.conv3(X)
        Y += X
        return F.relu(Y)

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))
    def block(self, num_residuals, num_channels, first_block=False, bottleneck=F
        blk = []
        for i in range(num_residuals):
            if bottleneck:
                if i == 0 and not first_block:
                    blk.append(Residual_bottleneck(num_channels, use_1x1conv=Tru
                else:
                    blk.append(Residual_bottleneck(num_channels))
            else:
                if i == 0 and not first_block:
                    blk.append(Residual(num_channels, use_1x1conv=True, strides=
                else:
                    blk.append(Residual(num_channels))
        return nn.Sequential(*blk)
    def __init__(self, arch, lr=0.1, num_classes=10, bottleneck=False):
        super(ResNet, self).__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(self.b1())
        for i, b in enumerate(arch):
            self.net.add_module(f'b{i+2}', self.block(*b, first_block=(i==0), bc
        self.net.add_module('last', nn.Sequential(
            nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
            nn.LazyLinear(num_classes)))
        self.net.apply(d2l.init_cnn)


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)

class ResNet34(ResNet):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__(((3, 64), (4, 128), (6, 256), (3, 512)),
                       lr, num_classes)
```

In [73]:
```
model = ResNet34(lr=0.01)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
trainer.fit(model, data)
```



In [76]:
```
class Residual(nn.Module):
    """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)

class Residual_bottleneck(nn.Module):
    """The Residual bottleneck block of ResNet models."""
    def __init__(self, num_channels, use_1x1conv=False, use_1x1conv_channel=Fals
        super().__init__()
        self.conv0_1 = nn.LazyConv2d(num_channels, kernel_size=1)
        self.conv1 = nn.LazyConv2d(num_channels, kernel_size=3, padding=1,
                                   stride=strides)
        self.conv0_2 = nn.LazyConv2d(num_channels * 4, kernel_size=1)

        self.conv3 = None
        if use_1x1conv:
            self.conv3 = nn.LazyConv2d(num_channels * 4, kernel_size=1,
                                       stride=strides)
        if use_1x1conv_channel:
            self.conv3 = nn.LazyConv2d(num_channels * 4, kernel_size=1,
                                       stride=strides)



        self.bn1 = nn.LazyBatchNorm2d()
        self.bn2 = nn.LazyBatchNorm2d()
        self.bn3 = nn.LazyBatchNorm2d()

    def forward(self, X):
        X.clone()
        Y = F.relu(self.bn1(self.conv0_1(X)))
        Y = F.relu(self.bn2(self.conv1(Y)))
        Y = self.bn3(self.conv0_2(Y))
        if self.conv3:
            X = self.conv3(X)
        Y += X
        return F.relu(Y)

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))
    def block(self, num_residuals, num_channels, first_block=False, bottleneck=F
        blk = []
        for i in range(num_residuals):
            if bottleneck:
                if i == 0 and not first_block:
                    blk.append(Residual_bottleneck(num_channels, use_1x1conv=Tru
                else:
                    blk.append(Residual_bottleneck(num_channels, use_1x1conv_cha
            else:
                if i == 0 and not first_block:
                    blk.append(Residual(num_channels, use_1x1conv=True, strides=
                else:
                    blk.append(Residual(num_channels, use_1x1conv_channel=True))
        return nn.Sequential(*blk)
    def __init__(self, arch, lr=0.1, num_classes=10, bottleneck=False):
        super(ResNet, self).__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(self.b1())
        for i, b in enumerate(arch):
            self.net.add_module(f'b{i+2}', self.block(*b, first_block=(i==0), bc
        self.net.add_module('last', nn.Sequential(
            nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
            nn.LazyLinear(num_classes)))
        self.net.apply(d2l.init_cnn)


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)

class ResNet34(ResNet):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__(((2, 64), (2, 128), (2, 256), (2, 512)),
                       lr, num_classes)

class ResNet50(ResNet):
    def __init__(self, lr=0.1, num_classes=10, bottleneck=True):
        super().__init__(((2, 64), (2, 128), (2, 256), (2, 512)),
                       lr, num_classes, bottleneck=bottleneck)
```

In [77]:
```
model = ResNet50(lr=0.01)
#print(model.layer_summary)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
trainer.fit(model, data)
```



Chapter 8 discussion

8.2 VGG network tries to expand the network by adding more layer and build a block. Also, it uses downsamping by two and extend the channel size by two thereby preserving computational amount but increase complexity.

8.6 ResNet tried to solve the problem by using neural network to learn not a entire mapping but residue of more complex function with identity mapping. Since gradient starts by one, training deeper model became possible.

```
In [ ]:
```