7. Convolution

7.1. From Fully Connected Layers to Convolutions

7.1.1 Invariance

"Spatial invariance"

1. translation invariance: network should respond similarly to the **same patch**, regardless of where it appears in the image.
2. locality principle: network should focus on local regions, without regard for the contents of the image in distant regions
3. deeper layers should be able to capture longer-range features of the image, in a way similar to higher level vision in nature.

7.1.2 Constraining the MLP

- Two-dimensional images $\mathbf{X}$ as inputs and their immediate hidden representations $\mathbf{H}$ (in the same shape = pixel size)
- bias $\mathbf{U}$, fourth-order weight tensors **W**

$$[\mathbf{H}]_{i,j} = [\mathbf{U}]_{i,j} + \sum_k \sum_l [\mathsf{W}]_{i,j,k,l} [\mathbf{X}]_{k,l} = [\mathbf{U}]_{i,j} + \sum_a \sum_b [\mathsf{V}]_{i,j,a,b} [\mathbf{X}]_{i+a,j+b}.$$

- $[\mathbf{H}]_{i,j}$ to $[\mathbf{X}]_{k,l}$ ($k = i + a$, $l = j + b$)
- **W** -> **V**: one-to-one correspondence btw coefficients in both fourth-order tensors
- The indices a and b run over both positive & negative offsets so that covering the entire image
- ex. 1000x1000 pixels -> run over 1000x1000 pixels: 1000^4 parameters

7.1.2.1. Translation Invariance

- Shift in the input $\mathbf{X}$ should simply lead to a shift in the hidden representation $\mathbf{H}$.
- **V** and **U** don't depend on (i,j)

$[\mathbf{H}]_{i,j} = u + \sum_a \sum_b [\mathbf{V}]_{a,b} [\mathbf{X}]_{i+a,j+b}.$

- $[V]_{a,b}$ needs fewer coefficients than $[\mathsf{V}]_{i,j,a,b}$
- ex. 1000x1000 pixels -> $4 * 10^6$

7.1.2.2. Locality

- We don't have to look very far away from location to get relevant information to assess what is going on at (i,j)
- Outside the location $|a| > \Delta$ or $|b| > \Delta$, we should set $[V]_{a,b} = 0$

$[\mathbf{H}]_{i,j} = u + \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} [\mathbf{V}]_{a,b} [\mathbf{X}]_{i+a,j+b}$.

- number of parameters from $4 * 10^6$ to $4\Delta^2$

Convolutional Layers

- Convolutional neural networks (CNNs) are a special family of neural networks that contain convolutional layers
- $\mathbf{V}$ (=weight tensors) is referred to as a convolution kernel, a filter, or simply the layer's weights that are learnable parameters

7.1.3. Convolutions

Why Convolution?

- definition: $f, g : \mathbb{R}^d \to \mathbb{R}$ is defined as
- $(f * g)(\mathbf{x}) = \int f(\mathbf{z}) g(\mathbf{x} - \mathbf{z}) d\mathbf{z}$.
- $(f * g)(i, j) = \sum_a \sum_b f(a, b) g(i - a, j - b)$.

7.1.4. Channels

- 3 Channels in RGB image
- To support multiple channels in both inputs(**X**) and hidden representation(**H**), add a fourth coordinate to **V**: $[* * V * *]_a, b, c, d$
- d is output channels in H

$[\mathsf{H}]_{i,j,d} = \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} \sum_c [\mathsf{V}]_{a,b,c,d} [\mathsf{X}]_{i+a,j+b,c}$

7.1.5. Sumary and Discussion

1. Translation invariance / Locality
2. How to reduce the number of parameters in a function class without limiting its expressive power?
   - Shift in X is same in H (1-1)
   - Get information from near area (1-2)
3. Adding channels of image: manipulating dimensionality is important.

7.2. Convolutions for Images

7.2.1. The Cross-Correlation Operation

- cross-correlation
- input * kernel window (convolution window) = output
- output size formula (Given input size $n_h \times n_w$ and kernel $k_h \times k_w$)
  $(n_h - k_h + 1) \times (n_w - k_w + 1)$
- need enough space to "shift" -> padding

In [21]:
```
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l
```

In [13]:
```
def corr2d(X, K):
    """Compute 2D cross-correlation"""
    '''
    X: input tensor
    K: kernel tensor
    Y: output tensor
    '''
    h, w = K.shape # shape of kernel
    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 [14]:
```
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[14]:
```
tensor([[19., 25.],
        [37., 43.]])
```

7.2.2. Convolutional Layers

A convolutional layer cross-correlates the ***input*** and ***kernel*** and adds a ***scalar bias*** to produce an output.

- **kernel** and **scalar bias** are two parameters of a convolutional layer
- Typically initialize the kernels randomly, just as a fully connected layer.

In [9]:
```
class Conv2D(nn.Module):
    def __init__(self, kernel_size):
        super().__init__()
        ## By making nn.Parameter, parameters are optimized during back-prop
```

```
        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
```

7.2.3. Object Edge Detection in Images

In [10]:
```
## black-and-white image with four black columns (0)
X = torch.ones((6, 8))
X[:, 2:6] = 0
X
```

Out[10]:
```
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 [11]:
```
## If horizontally adjacent elements are same, output is 0.
## Otherwise, output is nonzero (because 1*2 kernel)
## Finite Difference Operator (x_(i,j) - x_((i+1),j))
K = torch.tensor([[1.0, -1.0]])
```

In [12]:
```
Y = corr2d(X, K)
Y
```

Out[12]:
```
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 [13]:
```
## Transpose -> only vertical edges are detected (horizontal difference = ve
corr2d(X.t(), K)
```

Out[13]:
```
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.]])
```

7.2.4. Learning a Kernel

1. Construct convolutional layer
2. Initialize kernel as random tensor
3. Squared error to compare Y with output of convolutional layer ($\hat{Y}$)
4. Calculate the gradient to update the kernel

In [14]:
```
conv2d = nn.LazyConv2d(1, kernel_size=(1, 2), bias=False)

## shape = (batch size, channel, height, width)
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
    ## initialize gradient into zero every iter
    conv2d.zero_grad()
    l.sum().backward()
    ## update the kernel
    conv2d.weight.data[:] -= lr * conv2d.weight.grad
    if (i + 1) % 2 == 0:
        ## every 2nd iter
        print(f'epoch {i + 1}, loss {l.sum():.3f}')
```

```
epoch 2, loss 2.793
epoch 4, loss 0.576
epoch 6, loss 0.141
epoch 8, loss 0.042
epoch 10, loss 0.014
```

In [15]:
```
conv2d.weight.data.reshape((1, 2))
```

Out[15]:
```
tensor([[ 0.9805, -1.0036]])
```

7.2.5. Cross-Correlation and Convolution

Cross-Correlation vs. Convolution

- Convolution: kernel을 뒤집어서 계산함 -> learned kernel K' -> after flipped horizontally & vertically, same as K
- Cross-Correlation: kernel을 뒤집어서 계산하지 않음 (CNN에서 사용) -> learned kernel K

7.2.6. Feature Map and Receptive Field

- Convolutional layer output is sometimes called a **feature map**
- Learned representations (features) in the spatial dimensions (e.g., width and height) to the subsequent layer

7.2.7. Summary

In terms of convolutions themselves, they can be used for many purposes, for example detecting edges and lines, blurring images, or sharpening them. **We can simply learn**

suitable filters from data. This replaces feature engineering heuristics by evidence-based statistics.

7.3. Padding and Stride

padding and ***strided convolutions*** offer more control over the size of the output.

- padding: conserve original image size
- stride: reduce dimensionality dramatically

7.3.1. Padding

- With padding, output shape will be $(n_\text{h} - k_\text{h} + p_\text{h} + 1) \times (n_\text{w} - k_\text{w} + p_\text{w} + 1)$.
- To give input and output same height and width, $p_\text{h} = k_\text{h} - 1$ and $p_\text{w} = k_\text{w} - 1$.
- Assuming that k_h is **odd** here, we will pad $p_\text{h}/2$ rows on both sides of the height.
- If k_h is even, one possibility is to pad $\lceil p_\text{h}/2 \rceil$ rows on the top of the input and $\lfloor p_\text{h}/2 \rfloor$ rows on the bottom. We will pad both sides of the width in the same way.

In [16]:
```
## input: 8*8
## padding: 1 pixel on all sides (-> p=2)
## kernel: 3*3
## output: 8*8 (8-3+2+1=8)

def comp_conv2d(conv2d, X):
    # (1, 1) indicates that batch size and the number of channels are both 1
    X = X.reshape((1, 1) + X.shape)
    Y = conv2d(X)
    # Strip the first two dimensions: examples and channels
    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[16]:
```
torch.Size([8, 8])
```

In [17]:
```
## When height and width of convolution kernel are different, output and inp
## output: 8-5+4+1=8, 8-3+2+1=8
conv2d = nn.LazyConv2d(1, kernel_size=(5, 3), padding=(2, 1))
comp_conv2d(conv2d, X).shape
```

Out[17]:
```
torch.Size([8, 8])
```

7.3.2. Stride

- Either for computational efficiency or to downsample

- Useful if the convolution kernel is large since it captures a large area of the underlying image.
- When the stride for the height is s_h and the stride for the width is s_w,
- Output shape is $\lfloor (n_\mathrm{h} - k_\mathrm{h} + p_\mathrm{h} + s_\mathrm{h})/s_\mathrm{h} \rfloor \times \lfloor (n_\mathrm{w} - k_\mathrm{w} + p_\mathrm{w} + s_\mathrm{w})/s_\mathrm{w} \rfloor$.

In [18]:
```
## input: 8*8
## padding: 1 pixel on all sides (-> p=2)
## stride: 2
## kernel: 3*3
## output: 8*8 ((8-3+2+2)/2=4, (8-3+2+2)/2=4)

conv2d = nn.LazyConv2d(1, kernel_size=3, padding=1, stride=2)
comp_conv2d(conv2d, X).shape
```

Out[18]:
```
torch.Size([4, 4])
```

In [19]:
```
## output: (8-3+0+3)/3 = 2, (8-5+2+4)/4=2
conv2d = nn.LazyConv2d(1, kernel_size=(3, 5), padding=(0, 1), stride=(3, 4))
comp_conv2d(conv2d, X).shape
```

Out[19]:
```
torch.Size([2, 2])
```

7.3.3. Summary and Discussion

padding

So far all padding that we discussed simply extended images with zeros. This has significant computational benefit since it is trivial to accomplish. Moreover, operators can be engineered to take advantage of this padding implicitly without the need to allocate additional memory. At the same time, it allows CNNs to encode implicit position information within an image, simply by learning where the “whitespace” is.

7.4. Multiple Input and Multiple Output Channels

7.4.1. Multiple Input Channels

ex. RGB image -> input and hidden representation are both 3-D.

When $c_\mathrm{i} > 1$, we need a kernel that contains a tensor of shape $k_\mathrm{h} \times k_\mathrm{w}$ for every input channel.

- Perform a **cross-correlation** operation on 2D tensor of the **input** and 2D tensor of the **convolution kernel** for each channel
- Then add the c_i results together (summing over the channels) to yield 2D tensor (**output**)

- Input과 Kernel의 채널 개수가 같음. 각각 convolution 연산(사실은 cross correlation)한 다음, Output은 채널을 합침
- $(1 \times 1 + 2 \times 2 + 4 \times 3 + 5 \times 4) + (0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3) = 56$

In [20]:
```
def corr2d_multi_in(X, K):
    # Iterate through the 0th dimension (channel) of K first, then add them
    return sum(d2l.corr2d(x, k) for x, k in zip(X, K))
```

In [21]:
```
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[21]:
```
tensor([[ 56.,  72.],
        [104., 120.]])
```

7.4.2. Multiple Output Channels

- We increase the channel dimension as we go deeper in the neural network, typically **downsampling to trade off spatial resolution for greater channel depth**.
- Representations are learned independently **per pixel or per channel**. Instead, channels are optimized to be **jointly useful**.
- This means that ***rather than mapping a single channel to an edge detector,*** it may simply mean that **some direction in channel space corresponds to detecting edges.**

Dimension

- **Kernel tensor of shape $c_{\mathrm{i}} \times k_{\mathrm{h}} \times k_{\mathrm{w}}$ for *every* output channel. (per input channel)**
- **concat them on output channel dimension**
- **shape of convolutional kernel** $c_{\mathrm{o}} \times c_{\mathrm{i}} \times k_{\mathrm{h}} \times k_{\mathrm{w}}$ (output * input * kernel)
- Result on each output channel takes input **from all channels in the input tensor** and is calculated from the convolution kernel **corresponding to that output channel.**

In [22]:
```
def corr2d_multi_in_out(X, K):
    # Iterate through the 0th dimension of K, and each time, perform
    # cross-correlation operations with input X. All of the results are
    # stacked together
    return torch.stack([corr2d_multi_in(X, k) for k in K], 0)
```

In [23]:
```
K = torch.stack((K, K + 1, K + 2), 0)
K.shape
```

Out[23]:
```
torch.Size([3, 2, 2, 2])
```

In [24]:
```
corr2d_multi_in_out(X, K)
```

Out[24]:
```
tensor([[[ 56.,  72.],
         [104., 120.]],

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

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

7.4.3. 1×1 Convolutional Layer

Inputs and outputs have the same height and width whenusing 1*1 convolution. Because this is still a convolutional layer, the weights are tied across pixel location. Thus the 1×1 convolutional layer requires $c_{\mathrm{o}} \times c_{\mathrm{i}}$ weights (plus the bias).

In [25]:
```
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))
    # Matrix multiplication in the fully connected layer
    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)
##  equivalent
assert float(torch.abs(Y1 - Y2).sum()) < 1e-6
```

7.4.4. Discussion

- MLP for convolution -> localized analysis
- CNN -> edge/shape detectors...
- Given an image of size $(h \times w)$, the cost for computing a $k \times k$ convolution is $\mathcal{O}(h \cdot w \cdot k^2)$.
- For c_{i} and c_{o}, input and output channels respectively, this increases to $\mathcal{O}(h \cdot w \cdot k^2 \cdot c_{\mathrm{i}} \cdot c_{\mathrm{o}})$.

7.5. Pooling

7.5.1. Maximum Pooling and Average Pooling

Why pooling?

- Dual purposes of **mitigating the sensitivity of convolutional layers to location** and of **spatially downsampling representations.**
- Because in reality objects hardly ever occur exactly at the same place.

What is pooling?

- Like convolutional layers, pooling operators consist of a fixed-shape window that is slid over all regions in the input according to its stride, computing a single output for each location traversed by the fixed-shape window (sometimes known as the pooling window).
- However, unlike the cross-correlation computation of the inputs and kernels in the convolutional layer, **the pooling layer contains no parameters (there is no kernel).**
- Calculating either the maximum or the average value of the elements in the pooling window. (max-pooling / average pooling)

Average Pooling: downsampling an image
Max Pooling: How information aggregation might be aggregated hierarchically for the purpose of object recognition;

In [27]:
```
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 [28]:
```
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[28]:
```
tensor([[4., 5.],
        [7., 8.]])
```

In [29]:
```
pool2d(X, (2, 2), 'avg')
```

Out[29]:
```
tensor([[2., 3.],
        [5., 6.]])
```

7.5.2. Padding and Stride

In [30]:
```
X = torch.arange(16, dtype=torch.float32).reshape((1, 1, 4, 4))
X
```

Out[30]:
```
tensor([[[[ 0.,  1.,  2.,  3.],
          [ 4.,  5.,  6.,  7.],
          [ 8.,  9., 10., 11.],
          [12., 13., 14., 15.]]]])
```

In [31]:
```
## pooling window of shape (3,3), we get a stride shape of (3, 3) by default
pool2d = nn.MaxPool2d(3)
# Pooling has no model parameters, hence it needs no initialization
pool2d(X)
```

Out[31]:
```
tensor([[[[10.]]]])
```

In [32]:
```
pool2d = nn.MaxPool2d(3, padding=1, stride=2)
pool2d(X)
```

Out[32]:
```
tensor([[[[ 5.,  7.],
          [13., 15.]]]])
```

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

Out[33]:
```
tensor([[[[ 5.,  6.,  7.,  7.],
          [ 9., 10., 11., 11.],
          [13., 14., 15., 15.],
          [13., 14., 15., 15.]]]])
```

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

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

7.5.3. Multiple Channels

Multi-channel input data: number of output channels for the pooling layer == the number of input channels.
(pooling layer pools each input channel separately, rather than summing the inputs up)

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.]]]])
```

7.5.4. Summary

Pooling = aggregate results over a window of values

- indifferent to channels, i.e., it leaves the number of channels unchanged and it applies to each channel separately.
- **max-pooling** is preferable to average pooling

7.6. Convolutional Neural Networks (LeNet)

7.6.1. LeNet

In [52]:
```
def init_cnn(module):  #@save
    """Initialize weights for CNNs."""
    if type(module) == nn.Linear or type(module) == nn.Conv2d:
        nn.init.xavier_uniform_(module.weight)
        ## Xavier 초기화
        ## Fill the input Tensor with values using a Xavier uniform distribu
        ## The method is described in Understanding the difficulty of traini

class LeNet(d2l.Classifier):  #@save
    """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))
```

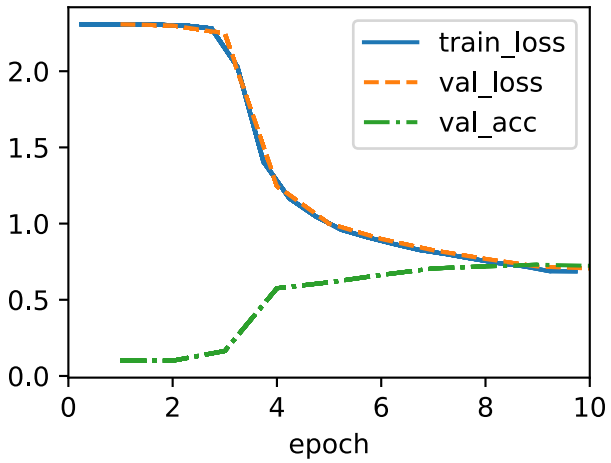
In [38]:
```
@d2l.add_to_class(d2l.Classifier)  #@save
def layer_summary(self, X_shape):
    X = torch.randn(*X_shape)
    for layer in self.net:
        X = layer(X)
        print(layer.__class__.__name__, 'output shape:\t', X.shape)

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])
```

7.6.2. Training

In [39]:
```
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)
```



7.6.3. Summary

Contribution of LeNet

- greater amounts of computation enabled significantly more complex architectures.
- the relative ease with which we were able to implement LeNet.

8. Modern Convolutional Neural Networks

8.2. Networks Using Blocks (VGG)

8.2.1. VGG Blocks

- CNN basic building block
  1. Convolutional layer ***with padding to maintain the resolution***
  2. Nonlinearity such as a ReLU
  3. Pooling layer such as max-pooling to ***reduce the resolution***
- **Problem of CNN**
  - **Spatial resolution decreases quite rapidly**
  - 1 Convolutional Layer, 1 Max pooling
- **Multiple convolutions in between downsampling via max-pooling**
  - Use less parameters / Better performance (3x3 kernels)

In [4]:
```
def vgg_block(num_convs, out_channels):
    layers = []
    for _ in range(num_convs):
        # out channels -> c (diff with number of conv)
        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)
```

8.2.2. VGG Network

- VGG Network (VGG-11)
  - Convolutional and pooling layers (VGG Blocks)
  - Fully connected layers that are identical to those in AlexNet

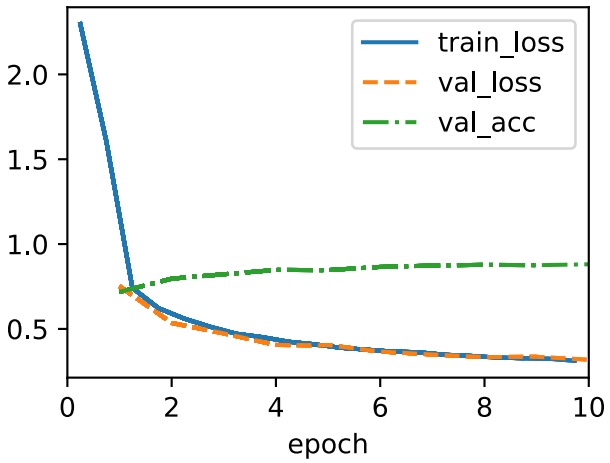
In [9]:
```
class VGG(d2l.Classifier):
    def __init__(self, arch, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        conv_blks = []
        ## vgg blocks
        for (num_convs, out_channels) in arch:
            conv_blks.append(vgg_block(num_convs, out_channels))
        ## vgg blocks + fully connected layer
        self.net = nn.Sequential(
            *conv_blks, nn.Flatten(),
            ## Fully connected layers expect input in a one-dimensional form
            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 [6]:
```
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])
```

8.2.3. Training

In [7]:
```
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)
```



8.2.4. Summary

VGG introduced key properties such as blocks of multiple convolutions and a preference for **deep and narrow networks.**

8.6. Residual Networks (ResNet) and ResNeXt¶

8.6.1. Function Classes

1. Definition
   - $\mathcal{F}$ = **class of functions that a specific network architecture can reach**

- For all $f \in \mathcal{F}$, there exists some set of parameters (e.g., weights and biases) that can be obtained through training on a suitable dataset.
- f^* is "Truth" function that we would like to find, and $f^*_{\mathcal{F}}$ **is our best bet within** $\mathcal{F}$ - Given a dataset with features $\mathbf{X}$ and labels $\mathbf{y}$, we might try finding it by solving the following optimization problem:
- $f^*_{\mathcal{F}} \stackrel{\mathrm{def}}{=} \mathrm{argmin}_f \, L(\mathbf{X}, \mathbf{y}, f) \text{ subject to } f \in \mathcal{F}.$

2. Nested vs. Non-nested Function
    - Regularization (Morozov, 1984, Tikhonov and Arsenin, 1977) may control complexity of $\mathcal{F}$ and achieve consistency, so larger size of training data needed.
    - It is only valid when $f^*_{\mathcal{F}'}$ is better than $f^*_{\mathcal{F}}$ ($\mathcal{F}$ -> $\mathcal{F}'$) ,*but it can be WORSE!*
    - For non-nested function classes, a larger (indicated by area) function class does not guarantee we will get closer to the “truth” function (f^*). This does not happen in nested function classes.
    - Thus, only if larger function classes contain the smaller ones are we guaranteed that increasing them strictly increases the expressive power of the network.
    - For deep neural networks, if we can train the newly-added layer into an identity function $f(\mathbf{x}) = \mathbf{x}$, the new model will be as effective as the original model.

3. ResNet
    - Residual network (ResNet) is the idea that **every additional layer should more easily contain the identity function as one of its elements.**
    - Residual block

8.6.2. Residual Blocks

- $f(\mathbf{x})$: **desired underlying mapping** we want to obtain
- On the left, the portion within the dotted-line box must directly learn $f(\mathbf{x})$
- On the right, the portion within the dotted-line box needs to learn the **residual mapping** $g(\mathbf{x}) = f(\mathbf{x}) - \mathbf{x}$
- If **identity mapping** $f(\mathbf{x}) = \mathbf{x}$ is desired underlying mapping, residual mapping is $g(\mathbf{x}) = 0$ thus easier to learn
- Because we only need to push the weights and biases of the upper weight layer
- With Residual blocks, inputs can forward propagate faster through the residual connections across layers.
- In fact, the residual block can be thought of as a special case of the multi-branch Inception block: it has two branches one of which is the identity mapping.
- ResNet has VGG’s full 3x3 convolutional layer design
- Each convolutional layer is followed by a batch normalization layer and a ReLU activation function.

- Then, we skip these two convolution operations and add the input directly before the final ReLU activation function.
- Required that the **output of the two convolutional layers has to be of the same shape as the input,** so that they can be added together.

In [22]:
```
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)

        ## True: adjust channels and resolution by means of 1x1 convolution
        ## False: add the input to the output before applying ReLU
        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 [24]:
```
## When input and output are of the same shape
blk = Residual(3)
X = torch.randn(4, 3, 6, 6)
blk(X).shape
```

Out[24]:
```
torch.Size([4, 3, 6, 6])
```

In [26]:
```
## increase output channel and
## half h and w by stride=2
blk = Residual(6, use_1x1conv=True, strides=2)
blk(X).shape
```

Out[26]:
```
torch.Size([4, 6, 3, 3])
```

8.6.3. ResNet Model

First two layers of ResNet

- convolutional layer with 64 output channels and a stride of 2
- max-pooling layer with a stride of 2.

- The difference with GoogleNet is the **batch normalization layer added after each convolutional layer in ResNet.**

Modules

- ResNet uses four modules made up of residual blocks, each of which uses **several residual blocks with the same number of output channels.**
- The number of channels in the first module is the same as the number of input channels.
- **Since a max-pooling layer with a stride of 2 has already been used, it is not necessary to reduce the height and width.**
- In the first residual block for each of the subsequent modules, the number of channels is doubled compared with that of the previous module, and the height and width are halved.

In [32]:
```
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 [33]:
```
@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 [34]:
```
@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)
```

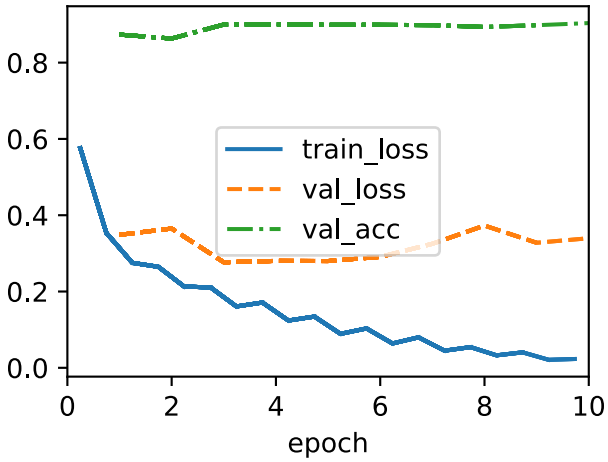
In [35]:
```
class ResNet18(ResNet):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__(((2, 64), (2, 128), (2, 256), (2, 512)),
                         lr, num_classes)

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])
```

8.6.4. Training

In [36]:
```
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)
```



#. Exercise / Discussion Points

7.

7.1

1. Assume that the size of the convolution kernel is $\Delta = 0$. Show that in this case **the convolution kernel implements an MLP independently for each set of channels.** This leads to the Network in Network architectures (Lin et al., 2013)
   - $H_{i,j,d} = \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} \sum_c V_{a,b,c,d} X_{i+a,j+b,c} = \sum_c V_{c,d} X_{i,j,c}$

7.2

1. Construct an image X with diagonal edges.

- What happens if you apply the kernel K in this section to it?
- What happens if you transpose X?

- What happens if you transpose K?

In [15]:
```
## X is 5x5 diagonal matrix
X = torch.tensor([[1.0, 1.0, 1.0, 1.0, 1.0],
                  [0.0, 1.0, 1.0, 1.0, 1.0],
                  [0.0, 0.0, 1.0, 1.0, 1.0],
                  [0.0, 0.0, 0.0, 1.0, 1.0],
                  [0.0, 0.0, 0.0, 0.0, 1.0]])
K = torch.tensor([[1.0, -1.0]])
Y = corr2d(X, K)
Y
```

Out[15]:
```
tensor([[ 0.,  0.,  0.,  0.],
        [-1.,  0.,  0.,  0.],
        [ 0., -1.,  0.,  0.],
        [ 0.,  0., -1.,  0.],
        [ 0.,  0.,  0., -1.]])
```

In [35]:
```
X_t = X.transpose(0, 1)
Y_xt = corr2d(X_t, K)
Y_xt
```

Out[35]:
```
tensor([[1., 0., 0., 0.],
        [0., 1., 0., 0.],
        [0., 0., 1., 0.],
        [0., 0., 0., 1.],
        [0., 0., 0., 0.]])
```

In [36]:
```
K_t = K.transpose(0, 1)
Y_kt = corr2d(X, K_t)
Y_kt
```

Out[36]:
```
tensor([[1., 0., 0., 0., 0.],
        [0., 1., 0., 0., 0.],
        [0., 0., 1., 0., 0.],
        [0., 0., 0., 1., 0.]])
```

2. Design some kernels manually.

- Given a directional vector $\mathbf{v} = (v_1, v_2)$, derive an edge-detection kernel that detects edges orthogonal to $\mathbf{v}$, i.e., edges in the direction $(v_2, -v_1)$.
  - $\begin{bmatrix} v_2 & -v_2 \\ -v_1 & v_1 \end{bmatrix}$
- Derive a finite difference operator for the second derivative. What is the minimum size of the convolutional kernel associated with it? Which structures in images respond most strongly to it?
  - (1, -2, 1). Minimum size 3, Image with curved edge
- How would you design a blur kernel? Why might you want to use such a kernel?
  - Mean filter, to reduce noise
- What is the minimum size of a kernel to obtain a derivative of order d?
  - $d + 1$

7.3

1. Given the final code example in this section with kernel size $(3, 5)$, padding $(0, 1)$, and stride $(3, 4)$, calculate the output shape to check if it is consistent with the experimental result.
   - output: (2, 2) ((8 - 3 + 0 + 3) / 3 = 2, (8 - 5 + 1 + 4) / 4 = 2)
2. For audio signals, what does a stride of 2 correspond to?
   - Downsample audio quality
3. What are the computational benefits of a stride larger than 1?
   - Faster speed
4. What might be statistical benefits of a stride larger than 1?
   - Increased receptive field -> consider larger portion of input
5. How would you implement a stride of 1/2? What does it correspond to? When would this be useful?
   - Upscaling, it can be useful when image size is too small

7.4

1. Assume that we have two convolution kernels of size $K_1$and K_2, respectively (with no nonlinearity in between).
   A. Prove that the result of the operation can be expressed by a single convolution.
      - $k = k_1 * k_2$ (association)
   B. What is the dimensionality of the equivalent single convolution?
      - $k_1 + k_2 - 1$
   C. Is the converse true, i.e., can you always decompose a convolution into two smaller ones?
      - No
2. By what factor does the number of calculations increase if we double both the number of input channels c_i and the number of output channels c_o? What happens if we double the padding?
   - Number of total caculation will be $k_h \times k_w \times c_i \times c_o \times h_o \times w_o$
   - If double both c_i and c_o, 4 times increase
   - If double padding, output dimension (h_o and w_o) increases (not linearly) so increase in computation

7.5

1. Implement average pooling through a convolution

In [127…
```
def pool2d(X, pool_size, mode):
    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 [128…
```
X = torch.arange(16, dtype=torch.float32).reshape(4, 4)
X
```

Out[128…
```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.],
        [12., 13., 14., 15.]])
```

In [130…
```
pool2d(X, (2, 2), 'avg')
```

Out[130…
```
tensor([[ 2.5000,  3.5000,  4.5000],
        [ 6.5000,  7.5000,  8.5000],
        [10.5000, 11.5000, 12.5000]])
```

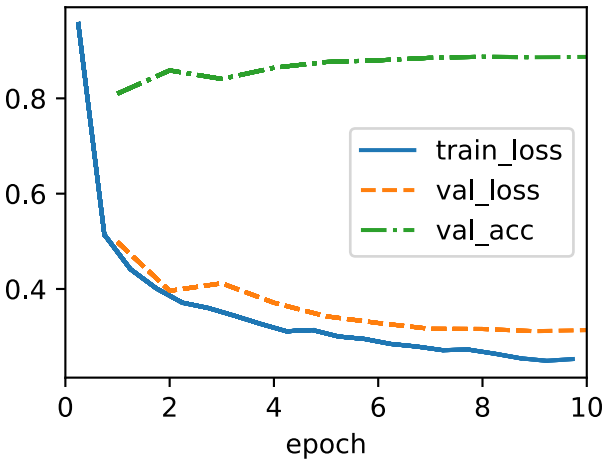
2. Prove that max-pooling cannot be implemented through a convolution alone.
   - Because max operation is not linear (convolution is linear)
3. Max-pooling can be accomplished using ReLU operations, i.e., $\mathrm{ReLU}(x) = \max(0, x)$.
   A. Express $\max(a, b)$ by using only ReLU operations.
      - $ReLU(a-b) = max(0, a-b), max(a,b) = max(0, a-b) + b = ReLU(a$
   B. How many channels and layers do you need for a 2x2 convolution? How many for a 3x3 convolution?
      - n×n convolution needs n^2 channels and layer.
4. What is the computational cost of the pooling layer?
   - $H_o = \lfloor (H + 2p - k)/s + 1 \rfloor, W_o = \lfloor (W + 2p - k)/s + 1 \rfloor$
   - $O(c \times h_o \times w_o)$
5. Why do you expect max-pooling and average pooling to work differently?
   - Max pooling -> Not differentiable, Feature sensitivity
   - Average pooling -> Differentiable, Noise Robustness
6. Do we need a separate minimum pooling layer? Can you replace it with another operation?
   - No usually, min pooling most likely result in zero activations.
   - Can be replaced by ReLU activations.
7. We could use the softmax operation for pooling. Why might it not be so popular?
   - Computational cost

7.6

1. Let's modernize LeNet. Implement and test the following changes:
   A. Replace average pooling with max-pooling.
   B. Replace the softmax layer with ReLU.

In [141…
```
class LeNet_modern(d2l.Classifier):  #@save
    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_modern(lr=0.1)
model.apply_init([next(iter(data.get_dataloader(True)))[0]], init_cnn)
trainer.fit(model, data)
```



8.

8.2.

1. Compared with AlexNet, VGG is much slower in terms of computation, and it also needs more GPU memory.
   A. Compare the number of parameters needed for AlexNet and VGG.

- AlexNet: 60 million / VGG: 138~144 million

B. Compare the number of floating point operations(FLOPs) used in the convolutional layers and in the fully connected layers.

- Convolutional Layers:
  - AlexNet: Uses larger filters and a stride of 4 in the first layer, thus reducing the computational cost.
  - VGG: More convolutional layers but uses small 3x3 filters. Despite the smaller size, the deep stack increases the FLOPs dramatically.
- Fully Connected Layers:
  - Both architectures have significant FLOPs in the fully connected layers due to the large number of connections.

C. How could you reduce the computational cost created by the fully connected layers?

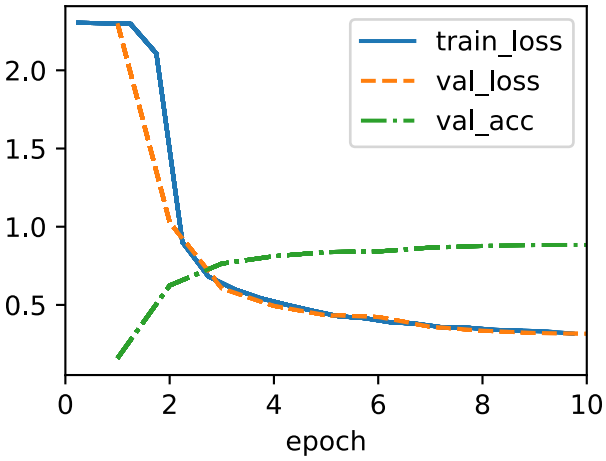
- Global Average Pooling instead of FC layers

2. When displaying the dimensions associated with the various layers of the network, we only see the information associated with eight blocks (plus some auxiliary transforms), even though the network has 11 layers. Where did the remaining three layers go?
   - 3 FC layers
3. Use Table 1 in the VGG paper (Simonyan and Zisserman, 2014) to construct other common models, such as VGG-16 or VGG-19.

In [8]:
```
VGG(arch=((2, 64), (2, 128), (3, 256), (3, 512), (3, 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 [10]:
```
model = VGG(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)
```



8.6.

ResNet

- x는 점점 출력값 H(x)에 근접하게 되어 추가 학습량 F(x)는 점점 작아져서 최종적으로 0에 근접하는 최소값으로 수렴되어야 할 것이다.
- 다시 말해, H(x)=F(x)+x에서 추가 학습량에 해당하는 F(x)=H(x)−x가 최소값(0)이 되도록 학습이 진행이 된다.
- 그리고, H(x)=F(x)+x 이므로 네트워크 구조 또한 크게 변경할 필요가 없는데, 단순히 입력에서 출력으로 바로 연결되는 shortcut만 추가하면 되기 때문이다. 또한, 입력과 같은 x가 그대로 출력에 연결되기에 파라미터 수에 영향이 없으며, 덧셈이 늘어나는 것을 제외하면 shortcut 연결을 통한 연산량 증가는 없다.
- 여기서 H(x) - x를 잔차(residual) 라고 한다. 즉, 잔차를 최소로 해주는 것이므로 ResNet이란 이름이 붙게 된다.

BottleNeck

- 채널 * 높이 * 너비
- Parameters: Kernel Size x Kernel Size x Input Channel x Output Channel
- Feature Map = output channel 개수
- 1x1 convolution을 통해 "채널" 수 "줄이면" 속도 증가, 그러나 정보손실
- 3x3 convolution을 통해 특성을 추출, 그러나 parameter의 수 증가하여 연산량 증가 (속도저하)
- 1x1 convolution을 통해 "채널" 수 "늘리면" feature map의 특성이 많아져 학습이 잘 됨, 그러나 정보손실

1. What are the major differences between the Inception block in Fig. 7.4.1 and the residual block? After removing some paths in the Inception block, how are they

related to each other?

- Inception uses multiple paths while ResNet uses one single path with X