

✓ Ch.0 Installation

✓ 0.1. Installation

```
1 !pip3 install d2l==1.0.3
```

 숨겨진 출력 표시

✓ Ch.7 Convolutional Neural Networks

✓ 7.1. From Fully Connected Layers to Convolutional

(No Codes)


✓ 7.2. Convolutions for Images

```
1 import torch
2 from torch import nn
3 from d2l import torch as d2l
```

✓ 7.2.1. The Cross-Correlation Operation

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

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


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

✓ 7.2.2. Convolutional Layers

```
1 class Conv2D(nn.Module):
2     def __init__(self, kernel_size):
3         super().__init__()
4         self.weight=nn.Parameter(torch.rand(kernel_size))
5         self.bias=nn.Parameter(torch.zeros(1))
6
7     def forward(self, x):
8         return corr2d(x, self.weight)+self.bias
```

✓ 7.2.3. Object Edge Detection in Images

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

 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.]])

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

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

```
↩ 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.]])
```

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

```
↩ 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 conv2d=nn.LazyConv2d(1, kernel_size=(1, 2), bias=False)
2
3 X=X.reshape((1, 1, 6, 8))
4 Y=Y.reshape((1, 1, 6, 7))
5 lr=3e-2
6
7 for i in range(10):
8     Y_hat=conv2d(X)
9     l=(Y_hat - Y)**2
10    conv2d.zero_grad()
11    l.sum().backward()
12    conv2d.weight.data[:]=lr*conv2d.weight.grad
13    if (i+1)%2==0:
14        print(f'epoch {i+1}, loss {l.sum():.3f}')
```

```
↩ epoch 2, loss 9.955
epoch 4, loss 2.809
epoch 6, loss 0.938
epoch 8, loss 0.348
epoch 10, loss 0.137
```

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

```
↩ tensor([[ 0.9505, -1.0257]])
```

7.2.5. Cross-Correlation and Convolution

(No Codes)

7.2.6. Feature Map and Receptive Field

(No Codes)

7.3. Padding and Stride

```
1 import torch
2 from torch import nn
```

7.3.1. Padding

```
1 def comp_conv2d(conv2d, X):
2     X=X.reshape((1, 1)+X.shape)
3     Y=conv2d(X)
4     return Y.reshape(Y.shape[2:])
5
6 conv2d=nn.LazyConv2d(1, kernel_size=3, padding=1)
7 X=torch.rand(size=(8, 8))
8 comp_conv2d(conv2d, X).shape
```

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

```
1 conv2d=nn.LazyConv2d(1, kernel_size=(5, 3), padding=(2, 1))
2 comp_conv2d(conv2d, X).shape
```

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

7.3.2. Stride

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

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

```
1 conv2d=nn.LazyConv2d(1, kernel_size=(3, 5), padding=(0, 1), stride=(3, 4))
2 comp_conv2d(conv2d, X).shape
```

```
torch.Size([2, 2])
```

7.4. Multiple Input and Multiple Output Channels

```
1 import torch
2 from d2l import torch as d2l
```

7.4.1. Multiple Input Channels

```
1 def corr2d_multi_in(X, K):
2     return sum(d2l.corr2d(x, k) for x, k in zip(X, K))
```

```
1 X=torch.tensor([[[[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]],
2                  [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]]]])
3 K=torch.tensor([[[[0.0, 1.0], [2.0, 3.0]], [[1.0, 2.0], [3.0, 4.0]]]])
4
5 corr2d_multi_in(X, K)
```

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

7.4.2. Multiple Output Channels

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

```
1 K=torch.stack((K, K+1, K+2), 0)
2 K.shape
```

```
torch.Size([3, 2, 2, 2])
```

```
1 corr2d_multi_in_out(X, K)
```

```
tensor([[[[ 56.,  72.],
           [104., 120.]],

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

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

7.4.3. 1×1 Convolutional Layer

```
1 def corr2d_multi_in_out_1x1(X, K):
2     c_i, h, w=X.shape
3     c_o=K.shape[0]
4     X=X.reshape((c_i, h*w))
5     K=K.reshape((c_o, c_i))
6     Y=torch.matmul(K, X)
7     return Y.reshape((c_o, h, w))
```

```

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

```

7.5. Pooling

```

1 import torch
2 from torch import nn
3 from d2l import torch as d2l

```

7.5.1. Maximum Pooling and Average Pooling

```

1 def pool2d(X, pool_size, mode='max'):
2     p_h, p_w=pool_size
3     Y=torch.zeros((X.shape[0]-p_h+1, X.shape[1]-p_w+1))
4     for i in range(Y.shape[0]):
5         for j in range(Y.shape[1]):
6             if mode=='max':
7                 Y[i, j]=X[i:i+p_h, j:j+p_w].max()
8             elif mode=='avg':
9                 Y[i, j]=X[i:i+p_h, j:j+p_w].mean()
10    return Y

```

```

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

```

```

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

```

```

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

```

```

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

```

7.5.2. Padding and Stride

```

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

```

```

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

```

```

1 pool2d=nn.MaxPool2d(3)
2 pool2d(X)

```

```

↔ tensor([[[[10.]]]])

```

```

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

```

```

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

```

```

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

```

```

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

```

7.5.3. Multiple Channels

```

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

```

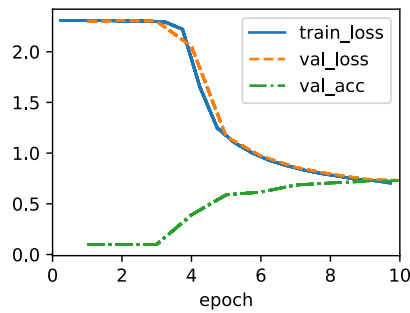
```

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

```

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

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



✓ Ch.8 Modern Convolutional Neural Networks

✓ 8.2. Networks Using Blocks (VGG)

```
1 import torch
2 from torch import nn
3 from d2l import torch as d2l
```

✓ 8.2.1. VGG Blocks

```
1 def vgg_block(num_convs, out_channels):
2     layers=[]
3     for _ in range(num_convs):
4         layers.append(nn.Conv2d(out_channels, kernel_size=3, padding=1))
5         layers.append(nn.ReLU())
6     layers.append(nn.MaxPool2d(kernel_size=2, stride=2))
7     return nn.Sequential(*layers)
```

✓ 8.2.2. VGG Network

```
1 class VGG(d2l.Classifier):
2     def __init__(self, arch, lr=0.1, num_classes=10):
3         super().__init__()
4         self.save_hyperparameters()
5         conv_blks=[]
6         for (num_convs, out_channels) in arch:
7             conv_blks.append(vgg_block(num_convs, out_channels))
8         self.net=nn.Sequential(
9             *conv_blks, nn.Flatten(),
10             nn.Linear(4096), nn.ReLU(), nn.Dropout(0.5),
11             nn.Linear(4096), nn.ReLU(), nn.Dropout(0.5),
12             nn.Linear(num_classes))
13         self.net.apply(d2l.init_cnn)

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



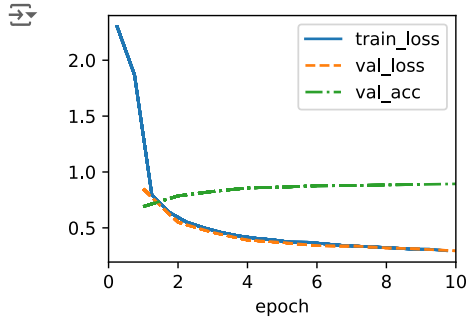
```
Sequential output shape: torch.Size([1, 64, 112, 112])
Sequential output shape: torch.Size([1, 128, 56, 56])
Sequential output shape: torch.Size([1, 256, 28, 28])
Sequential output shape: torch.Size([1, 512, 14, 14])
Sequential output shape: torch.Size([1, 512, 7, 7])
Flatten output shape: torch.Size([1,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

```

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

```



8.6. Residual Networks (ResNet) and ResNeXt

```

1 import torch
2 from torch import nn
3 from torch.nn import functional as F
4 from d2l import torch as d2l

```

8.6.1. Function Classes

(No Codes)

8.6.2. Residual Blocks

```

1 class Residual(nn.Module):
2     def __init__(self, num_channels, use_1x1conv=False, strides=1):
3         super().__init__()
4         self.conv1=nn.LazyConv2d(num_channels, kernel_size=3, padding=1,
5                                   stride=strides)
6         self.conv2=nn.LazyConv2d(num_channels, kernel_size=3, padding=1)
7         if use_1x1conv:
8             self.conv3=nn.LazyConv2d(num_channels, kernel_size=1,
9                                       stride=strides)
10        else:
11            self.conv3=None
12        self.bn1=nn.LazyBatchNorm2d()
13        self.bn2=nn.LazyBatchNorm2d()
14
15    def forward(self, X):
16        Y=F.relu(self.bn1(self.conv1(X)))
17        Y=self.bn2(self.conv2(Y))
18        if self.conv3:
19            X=self.conv3(X)
20        Y+=X
21        return F.relu(Y)

```

```

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

```

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

```

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

```

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

8.6.3. ResNet Model

```

1 class ResNet(d2l.Classifier):
2     def b1(self):
3         return nn.Sequential(
4             nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),

```

```

5     nn.LazyBatchNorm2d(), nn.ReLU(),
6     nn.MaxPool2d(kernel_size=3, stride=2, padding=1))

1 @d2l.add_to_class(ResNet)
2 def block(self, num_residuals, num_channels, first_block=False):
3     blk=[]
4     for i in range(num_residuals):
5         if i==0 and not first_block:
6             blk.append(Residual(num_channels, use_1x1conv=True, strides=2))
7         else:
8             blk.append(Residual(num_channels))
9     return nn.Sequential(*blk)

1 @d2l.add_to_class(ResNet)
2 def __init__(self, arch, lr=0.1, num_classes=10):
3     super(ResNet, self).__init__()
4     self.save_hyperparameters()
5     self.net = nn.Sequential(self.b1())
6     for i, b in enumerate(arch):
7         self.net.add_module(f'b{i+2}', self.block(*b, first_block=(i==0)))
8     self.net.add_module('last', nn.Sequential(
9         nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
10        nn.LazyLinear(num_classes)))
11    self.net.apply(d2l.init_cnn)

1 class ResNet18(ResNet):
2     def __init__(self, lr=0.1, num_classes=10):
3         super().__init__(((2, 64), (2, 128), (2, 256), (2, 512)),
4                             lr, num_classes)
5
6 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])

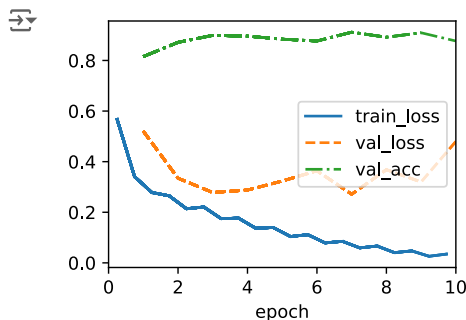
```

8.6.4. Training

```

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

```



Discussions and Exercises

Ch.7

Discussions / Exercises

7.1.

In the earliest layers, our network should respond similarly to the same patch, regardless of where it appears in the image. This principle is called **translation invariance** (or translation equivariance).

This implies that a shift in the input \mathbf{X} should simply lead to a shift in the hidden representation \mathbf{H} . This is only possible if \mathbf{V} and \mathbf{U} do not actually depend on (i, j) . As such, we have $[\mathbf{V}]_{i,j,a,b} = [\mathbf{V}]_{a,b}$ and \mathbf{U} is a constant, say u . As a result, we can simplify the definition for \mathbf{H} :

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

The earliest layers of the network should focus on local regions, without regard for the contents of the image in distant regions. This is the **locality** principle. Eventually, these local representations can be aggregated to make predictions at the whole image level.

we believe that we should not have to look very far away from location (i, j) in order to glean relevant information to assess what is going on at $[\mathbf{H}]_{i,j}$. This means that outside some range $|a| > \Delta$ or $|b| > \Delta$, we should set $[\mathbf{V}]_{a,b} = 0$. Equivalently, we can rewrite $[\mathbf{H}]_{i,j}$ as

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

This reduces the number of parameters from 4×10^6 to $4\Delta^2$, where Δ is typically smaller than 10. As such, we reduced the number of parameters by another four orders of magnitude.

✓ Ch.8

✓ Discussions / Exercises

✓ 8.6.

Consider \mathcal{F} , the class of functions that a specific network architecture (together with learning rates and other hyperparameter settings) can reach. That is, 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. Let's assume that f^* is the "truth" function that we really would like to find. If it is in \mathcal{F} , we are in good shape but typically we will not be quite so lucky. Instead, we will try to find some $f_{\mathcal{F}}^*$ which is our best bet within \mathcal{F} . For instance, given a dataset with features \mathbf{X} and labels \mathbf{y} , we might try finding it by solving the following optimization problem:

$$f_{\mathcal{F}}^* \stackrel{\text{def}}{=} \operatorname{argmin}_f L(\mathbf{X}, \mathbf{y}, f) \text{ subject to } f \in \mathcal{F}.$$

ResNet Model uses nested function classes that are desirable since they allow us to obtain strictly more powerful rather than also subtly different function classes when adding capacity. One way of accomplishing this is by letting additional layers to simply pass through the input to the output. Residual connections allow for this. As a consequence, this changes the inductive bias from simple functions being of the form $f(x)=0$ to simple functions looking like $f(x)=x$.