

- ✓ COSE474-2024F: Deep Learning HW2

- 0. Install

```
1 !pip install d2l==1.0.3
2 import torch
3 from torch import nn
4 from torch.nn import functional as F
5 from d2l import torch as d2l
```

 숨겨진 출력 표시

7.1. From Fully Connected Layers to Convolutions

7.2. Convolutions for Images

```
1 def corr2d(X, K):
2     """Compute 2D cross-correlation."""
3     h, w = K.shape
4     Y = torch.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1))
5     for i in range(Y.shape[0]):
6         for j in range(Y.shape[1]):
7             Y[i, j] = (X[i:i+h, j:j+w] * K).sum()
8     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.]])
```

```
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
```

```
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)
```

[illegible]

```
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 # Learning rate
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    # Update the kernel
13    conv2d.weight.data[:] -= lr * conv2d.weight.grad
14    if (i + 1) % 2 == 0:
15        print(f'epoch {i + 1}, loss {l.sum():.3f}')
```

↗ epoch 2, loss 1.581
epoch 4, loss 0.295
epoch 6, loss 0.062
epoch 8, loss 0.015
epoch 10, loss 0.005

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

↗ tensor([[0.9998, -0.9877]])

7.3. Padding and Stride

```
1 def comp_conv2d(conv2d, X):
2     # (1, 1) = (batch size, the number of channels)
3     X = X.reshape((1, 1) + X.shape)
4     Y = conv2d(X)
5     return Y.reshape(Y.shape[2:])
6
7 conv2d = nn.LazyConv2d(1, kernel_size=3, padding=1)
8 X = torch.rand(size=(8, 8))
9 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])

```
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 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.]])

```
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.]]]])
```

```
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 def pool2d(X, pool_size, mode='max'): # 디폴트는 맥스 풀링
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.]])
```

```
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.]]]]])
```

```
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.]],

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

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

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

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

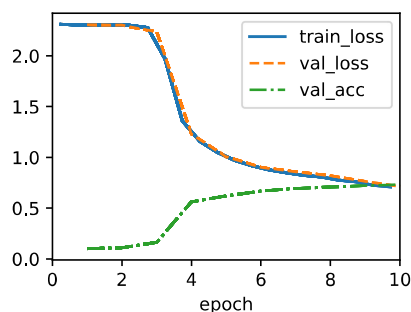
7.6. Convolutional Neural Networks (LeNet)

```
1 def init_cnn(module):
2     if type(module) == nn.Linear or type(module) == nn.Conv2d:
3         nn.init.xavier_uniform_(module.weight)
4
5 class LeNet(d2l.Classifier):
6     def __init__(self, lr=0.1, num_classes=10):
7         super().__init__()
8         self.save_hyperparameters()
9         self.net = nn.Sequential(
10             nn.LazyConv2d(6, kernel_size=5, padding=2), nn.Sigmoid(),
11             nn.AvgPool2d(kernel_size=2, stride=2),
12             nn.LazyConv2d(16, kernel_size=5), nn.Sigmoid(),
13             nn.AvgPool2d(kernel_size=2, stride=2),
14             nn.Flatten(),
15             nn.LazyLinear(120), nn.Sigmoid(),
16             nn.LazyLinear(84), nn.Sigmoid(),
17             nn.LazyLinear(num_classes))
```

```
1 @d2l.add_to_class(d2l.Classifier)
2 def layer_summary(self, X_shape):
3     X = torch.randn(*X_shape)
4     for layer in self.net:
5         X = layer(X)
6         print(layer.__class__.__name__, 'output shape: %d' % X.shape)
7
8 model = LeNet()
9 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])
```

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



8.2. Networks Using Blocks (VGG)

```

1 def vgg_block(num_convs, out_channels):
2     layers = []
3     for _ in range(num_convs):
4         layers.append(nn.LazyConv2d(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)

```

```

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.LazyLinear(4096), nn.ReLU(), nn.Dropout(0.5),
11            nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(0.5),
12            nn.LazyLinear(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))

```



```

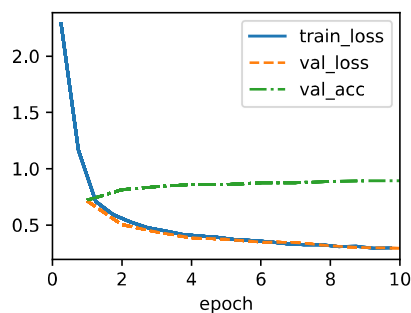
Sequential output shape:      torch.Size([1, 64, 112, 112])
Sequential output shape:      torch.Size([1, 128, 56, 56])
Sequential output shape:      torch.Size([1, 256, 28, 28])
Sequential output shape:      torch.Size([1, 512, 14, 14])
Sequential output shape:      torch.Size([1, 512, 7, 7])
Flatten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

```

```

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

(- 8.6.1 - 8.6.4)

```

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])

```

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))

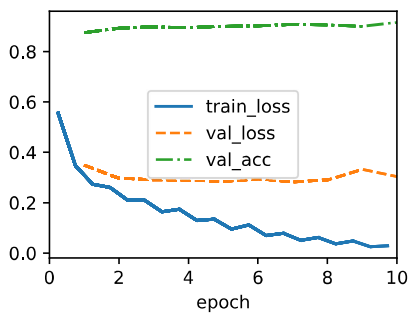
```

 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])

```

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_data_loader(True)))[0]], d2l.init_cnn)
5 trainer.fit(model, data)

```



Discussions & Exercises

7.1 note

- **translation invariance**: 가장 초기 레이어에서 우리의 신경망은 이미지의 어디에서 나타나는 동일한 patch에 대해 유사하게 반응해야 한다.
- **locality**: 신경망의 가장 초기 레이어는 멀리 떨어진 지역을 신경쓰지 않고 국소적으로 초점을 뒀야한다.
- 깊은 레이어에서는 먼 거리의 특징을 포착할 수 있게 되는데, 이것은 자연의 고차원적 시각과 유사하다.

- 기본적인 MLP

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

- 변환 불변성(translation invariance) 적용

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

($[V]_{i,j,a,b} = [V]_{a,b}$ 이고, U 는 상수인 모습. 이는 convolution과 같다.)

- 지역성(locality) 적용

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

(합의 범위를 지정해 매개변수 수를 줄임. 이를 이용해 합성곱 계층을 만든다.)

- 합성곱 수학적 개념

$$(f * g)(x) = \int f(z)g(x-z)dz$$

- 3차원 MLP: 채널(피쳐맵) 적용

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

7.2 note

- 이미지 사이즈가 $n_h \times n_w$ 이고, 커널 사이즈가 $k_h \times k_w$ 일 때, 커널을 적용한 결과 텐서의 사이즈는 $(n_h - k_h + 1) \times (n_w - k_w + 1)$ 이다.

7.3 note

- 패딩: 커널을 적용해 원래의 이미지 크기가 축소되는 것을 방지하고자 모서리에 적절한 크기로 0을 채워줌.
 - discussion: 왜 0으로 채우는 것일까?
 - 패딩 크기가 $p_h \times p_w$ 라면, 출력 크기는 $(n_h - k_h + p_h + 1) \times (n_w - k_w + p_w + 1)$ 이다.
- CNN에서는 일반적으로 커널의 높이 및 너비를 홀수로 하여 차원 유지를 용이하게 한다.
- stride: 커널을 적용할 때, 다운 샘플링 등을 목적으로 stride의 크기만큼 건너뛰면서 적용한다. 즉, stride는 해상도에 관여한다.
 - stride 크기가 $s_h \times s_w$ 라면, 출력 크기는 $[(n_h - k_h + p_h + s_h) / s_h] \times [(n_w - k_w + p_w + s_w) / s_w]$ 이다.

7.5 note

- pooling: 각 레이어에서 중요한 정보만을 취하기 위해 크기를 축소시키는 기법

- max pooling: pooling window에서 최댓값을 취함
- average pooling: pooling window에 포함된 셀들의 평균값을 취함
- pooling은 채널과 무관하여 채널의 수를 유지한다.

7.6 note

• LeNet

- 르넷은 (i) 두 개의 합성곱 레이어로 구성된 합성곱 인코더, (ii) 세 개의 완전 연결 레이어로 구성된 dense 블록으로 구성된다.
- (i) 합성곱 블록은 합성곱 레이어, 시그모이드 활성화 함수, 평균 풀링 연산으로 구성된다(일반적으로 시그모이드보다 ReLU가, 평균 풀링보다 최대 풀링이 더 뛰어난 성능을 보이지만 르넷-5 당시에는 이 사실을 알지 못했다).
- 각 합성곱 레이어는 여러 개의 2차원 피쳐맵에 매핑하여 채널 수를 늘림으로써 피쳐 포착을 용이하게 한다.
- (ii) dense 블록으로 전달하기 전, 피쳐맵은 평면화를 통해 1차원 벡터로 변환된다.

8.2 note

- VGG 네트워크는 레이어 단위로 운용되던 기존의 AlexNet과 달리 블록 단위로 합성곱 계층이 구성되고, 각 블록들은 동일한 크기의 필터를 갖는다. 이 덕에 네트워크의 깊이는 VGG가 더 깊은데도 불구하고 parameter 수가 균일하다.

8.6 note

- 정규화를 통해 \mathcal{F} 의 복잡성을 제어하고, 일관성을 확보할 수 있다.
- 함수 클래스를 중첩시키면 truth function에 가까워짐을 보장할 수 있다.
- ResNet의 핵심은 모든 추가적인 레이어가 '항등함수'를 더 쉽게 포함해야 한다는 아이디어이다. 이는 'residual block'을 통해 이루어졌다. 이 잔차 블록을 통해 기존의 입력을 항등함수로 그대로 전달해서 정보 손실을 낮춘다.
 - $y = F(x) + x$ 에서 F 는 일반적인 학습 함수, x 는 입력을 그대로 출력에 더해주는 역할을 한다.

더블클릭 또는 Enter 키를 눌러 수정

더블클릭 또는 Enter 키를 눌러 수정