

```
pip install d2l
```

 숨겨진 출력 표시

7.1. From Fully Connected Layers to Convolutions

7.1. Discussions

memo

- While previously, we might have required billions of parameters to represent just a single layer in an image-processing network, we now typically need just a few hundred, without altering the dimensionality of either the inputs or the hidden representations. The price paid for this drastic reduction in parameters is that our features are now translation invariant and that our layer can only incorporate local information, when determining the value of each hidden activation. All learning depends on imposing inductive bias. When that bias agrees with reality, we get sample-efficient models that generalize well to unseen data. But of course, if those biases do not agree with reality, e.g., if images turned out not to be translation invariant, our models might struggle even to fit our training data.

questions

CNN은 가변적인 크기의 이미지를 받을 수 있는가.


- 동적 패딩(Dynamic Padding): 이미지 크기가 다를 때, 가장 일반적인 방법은 가변적인 크기의 이미지를 처리하기 전에 동적 패딩을 적용하여 모든 입력 이미지를 동일한 크기로 만드는 것. 이를 통해 CNN이 입력을 일관되게 처리.
- Global Average Pooling: 마지막 합성곱 레이어 뒤에 Global Average Pooling 층을 추가하면, 다양한 크기의 입력 이미지에서 평균을 내어 고정된 크기의 출력 벡터를 생성.
- Fully Convolutional Networks (FCN): FCN은 모든 레이어가 합성곱 레이어로 구성된 신경망. 이 구조는 가변적인 입력 크기를 직접 처리할 수 있으며, 일반적으로 이미지 분할과 같은 작업에 사용됨. FCN은 마지막에 완전 연결층이 아닌, 합성곱 층을 사용하여 출력의 크기를 조정.

7.2. Convolutions for Images

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

```
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
```

```
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)
```

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

```
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
```

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

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

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

```
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.]])
```

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

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

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

```
↔ epoch 2, loss 3.635
   epoch 4, loss 0.638
   epoch 6, loss 0.118
   epoch 8, loss 0.025
   epoch 10, loss 0.006
```

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

```
↔ tensor([[ 0.9844, -0.9961]])
```

✓ 7.2. Discussions

takeaway messages

- LazyConv2d는 PyTorch의 내장 모듈. Lazy라는 이름에서 알 수 있듯이, LazyConv2d는 입력 텐서의 크기가 주어지지 않아도 정의할 수 있고, 실제로 데이터를 입력받을 때 가중치의 크기를 자동으로 설정
- LazyConv2d는 입력 텐서가 처음 들어올 때까지 weight를 초기화하지 않으며, 입력 데이터의 크기를 보고 필요한 크기에 맞게 가중치를 설정. 이를 통해 네트워크의 초기 설정이 더 유연해질 수 있다.

✓ 7.3. Padding and Stride

```
import torch
from torch import nn
```

```
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

```

↔ torch.Size([8, 8])

```

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

```

↔ torch.Size([8, 8])

```

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

```

↔ torch.Size([4, 4])

```

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

```

↔ torch.Size([2, 2])

✓ 7.3. Discussions

takeaway messages

- $\text{output_height} = ((\text{input_height} - \text{kernel_height} + 2 * \text{padding_height}) / \text{stride_height}) + 1$
- $\text{output_width} = ((\text{input_width} - \text{kernel_width} + 2 * \text{padding_width}) / \text{stride_width}) + 1$

✓ 7.4. Multiple Input and Multiple Output Channels

```

import torch
from d2l import torch as d2l

```

```

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

```

```

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)
```

↔ tensor([[56., 72.],
[104., 120.]])

```

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

```

```

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

```

↔ torch.Size([3, 2, 2, 2])

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

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

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

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

```

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))
```

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

7.4. Discussions

takeaway messages

- `torch.stack(tensors, dim)` 함수는 여러 개의 텐서를 새로운 차원을 추가하여 쌓는 함수.
이때 `dim` 인자는 새롭게 추가될 차원이 몇 번째 위치에 추가될지를 나타냄
`dim=0`은 새로운 차원을 가장 앞쪽에 추가하는 것을 의미
- `1x1 CNN`은 `MLP`의 완전 연결 계층과 동일하게 동작. `1x1` 커널을 사용하는 `CNN`은 공간적 정보를 전혀 고려하지 않고, 입력 채널들의 가중합을 학습하는 방식.

7.5. Pooling

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

```
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
```

```
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))
```

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

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

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

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

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

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

```
↔ tensor([[[[10.]]]])
```

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

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

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

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

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

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

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

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

✓ 7.5. Discussions

takeaway messages

- nn.MaxPool2d는 기본적으로 입력 텐서의 마지막 두 차원에 대해 풀링을 수행하도록 설계. 즉, 배치 차원(N)과 채널 차원(C)은 풀링에 영향을 주지 않고, 마지막 두 차원(높이와 너비)에 대해서만 풀링을 적용.
- nn.MaxPool2d의 파라미터 kernel_size 풀링 창 크기 기본: 필수

stride 풀링 창이 입력을 가로지르며 이동할 간격 기본: kernel_size

padding 입력 텐서의 가장자리에 추가되는 패딩 기본: 0

dilation 풀링 창 내부의 값들이 얼마나 떨어져 있는지를 결정 기본: 1

return_indices 최대값과 그 위치 인덱스를 함께 반환할지 여부 기본: False

ceil_mode 출력 크기를 ceil(올림) 방식으로 계산할지 여부 기본: False

✓ 7.6. Convolutional Neural Networks (LeNet)

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

```
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))
```

```
@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: %s' % X.shape)

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

```

```

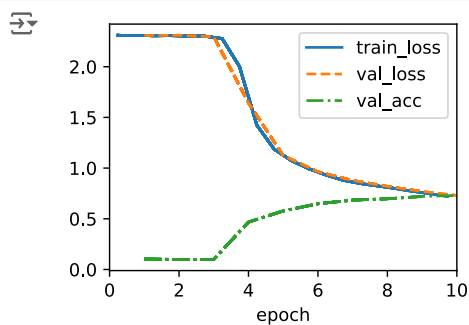
↳ Conv2d output shape:      torch.Size([1, 6, 28, 28])
  Sigmoid output shape:     torch.Size([1, 6, 28, 28])
  AvgPool2d output shape:   torch.Size([1, 6, 14, 14])
  Conv2d output shape:      torch.Size([1, 16, 10, 10])
  Sigmoid output shape:     torch.Size([1, 16, 10, 10])
  AvgPool2d output shape:   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])

```

```

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. Discussions

✓ self exercise

- 활성화 함수로 시그모이드가 아닌 ReLU 함수를 적용시켜 학습해 보았다.

```

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

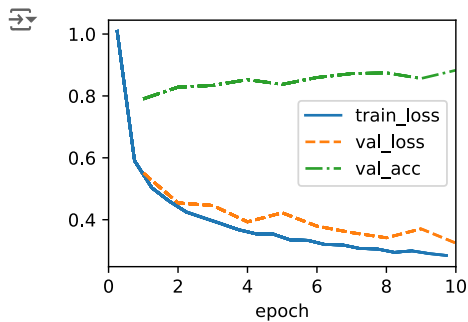
class LeNet(d2l.Classifier):
    """The LeNet-5 model."""
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.LazyConv2d(6, kernel_size=5, padding=2), nn.ReLU(),
            nn.AvgPool2d(kernel_size=2, stride=2),
            nn.LazyConv2d(16, kernel_size=5), nn.ReLU(),
            nn.AvgPool2d(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)

```



- 학습 결과 ReLU를 사용한 모델은 초기부터 손실값이 급격하게 줄어든고, 정확도가 빠르게 증가한다. 이는 ReLU가 단순하면서 효과적인 모델이며 기울기 소실 문제에 저항력을 가지고 있기 때문이라고 생각된다.
- 반면에 시그모이드를 적용한 모델은 손실값이 천천히 감소하다가 결국 특정값에 수렴되는데 이는 시그모이드가 기울기 감소 문제를 가지고 있기 때문에 학습이 비교적 느리고 정확하지 못한 것으로 생각된다.

✓ 8.2. Networks Using Blocks (VGG)

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

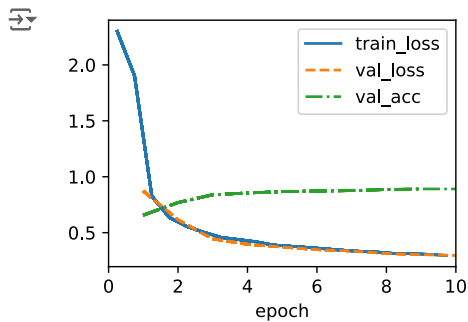
```
def vgg_block(num_convs, out_channels):
    layers = []
    for _ in range(num_convs):
        layers.append(nn.Conv2d(out_channels, kernel_size=3, padding=1))
        layers.append(nn.ReLU())
    layers.append(nn.MaxPool2d(kernel_size=2, stride=2))
    return nn.Sequential(*layers)
```

```
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.Linear(4096), nn.ReLU(), nn.Dropout(0.5),
            nn.Linear(4096), nn.ReLU(), nn.Dropout(0.5),
            nn.Linear(num_classes))
        self.net.apply(d2l.init_cnn)
```

```
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])
```

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

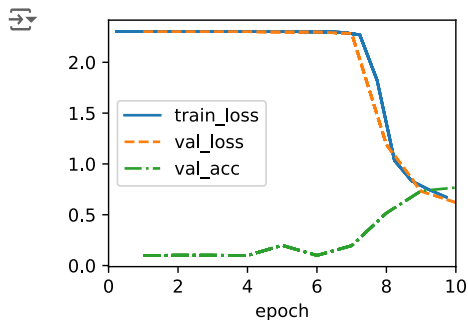


8.2. Discussions

self exercise

- VGG는 입력 크기를 224*224로 늘려 수행한다. 이는 FashionMNIST에 대해서는 낭비이다.

```
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=(56, 56))
model.apply_init([next(iter(data.get_dataloader(True)))[0]), d2l.init_cnn]
trainer.fit(model, data)
```



- 위는 기존 모델에서 입력 데이터의 크기만 줄인 모델이다. 입력 데이터의 크기를 줄이니 학습이 제대로 되지 않는 모습을 보인다.
- 이를 해결하기 위해 VGG의 성능을 저하시키지 않으면서 입력 크기를 줄일 수 있는 방법을 시행해보았다. 주요 수정사항은 입력될 이미지의 크기를 56*56으로 바꾼것이다. 이로 인해 입력될 이미지의 크기가 작으므로 컨볼루션 레이어의 수는 유지하되 풀링 계층을 줄여 마지막 풀링 계층의 output 이미지의 크기는 유지할 것이다. 그리고 마지막 linear 연결 계층의 뉴런 수를 적당한 수준으로 줄이고 줄어든 뉴런 수를 감안하여 dropout의 비율을 낮췄다.

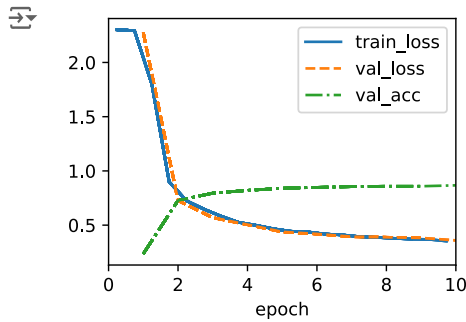
```
def vgg_block(num_convs, out_channels):
    layers = []
    for _ in range(num_convs):
        layers.append(nn.LazyConv2d(out_channels, kernel_size=3, padding=1))
        layers.append(nn.ReLU())
    layers.append(nn.MaxPool2d(kernel_size=2, stride=2))
    return nn.Sequential(*layers)
```

```
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(2048), nn.ReLU(), nn.Dropout(0.4),
            nn.LazyLinear(2048), nn.ReLU(), nn.Dropout(0.4),
            nn.LazyLinear(num_classes))
        self.net.apply(d2l.init_cnn)
```

```
model = VGG(arch=((2, 16), (2, 32), (2, 64), (2, 128)), lr=0.01)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(56, 56))
```



```
model.apply_init([next(iter(data.get_dataloader(True)))[0]), d2l.init_cnn)
trainer.fit(model, data)
```



- 기존 모델보다 비교적 나아진 모습을 보인다. 다만 loss의 수렴이 다소 큰 지점에서 발생하는 문제도 있고 LeNet에 비교해 비슷한 모습을 보이므로 작은 입력 크기에서 VGG를 구현하려면 조금더 적절한 모델 구조를 설계해야 할 것으로 보인다.

✓ 8.6. Residual Networks (ResNet) and ResNeXt

```
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l
```

```
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.Conv2d(num_channels, kernel_size=3, padding=1,
                                stride=strides)
        self.conv2 = nn.Conv2d(num_channels, kernel_size=3, padding=1)
        if use_1x1conv:
            self.conv3 = nn.Conv2d(num_channels, kernel_size=1,
                                    stride=strides)
        else:
            self.conv3 = None
        self.bn1 = nn.BatchNorm2d()
        self.bn2 = nn.BatchNorm2d()

    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)
```

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

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

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

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

```
class ResNet(d2l.Classifier):
    def b1(self):
        return nn.Sequential(
            nn.Conv2d(64, kernel_size=7, stride=2, padding=3),
            nn.BatchNorm2d(), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

```
@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)
```

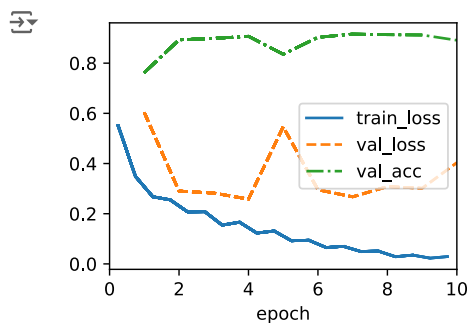
```
@d2l.add_to_class(ResNet)
def __init__(self, arch, lr=0.1, num_classes=10):
    super(ResNet, self).__init__()
    self.save_hyperparameters()
    self.net = nn.Sequential(self.b1())
    for i, b in enumerate(arch):
        self.net.add_module(f'b{i+2}', self.block(*b, first_block=(i==0)))
    self.net.add_module('last',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

```
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])
```

```
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)
```



✓ 8.6. Discussions

takeaway messages

- `nn.LazyBatchNorm2d()`: 배치 정규화(Batch Normalization, BN)는 딥 러닝 모델에서 신경망의 학습을 안정화하고 가속화하기 위해 사용되는 기법

self exercise

- ResNet은 배치정규화와 Skip-Connection을 통해 성능 개선을 이루었다. 배치정규화를 제외한 모델, Skip-Connection을 제거한 모델 두 모델을 구축해서 배치정규화와 Skip-Connection이 모델 성능에 얼마나 도움이 되는지 알아보고자 한다.

✓ 배치정규화가 제외된 모델

```
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.conv1(X))
        Y = self.conv2(Y)
        if self.conv3:
            X = self.conv3(X)
        Y += X
        return F.relu(Y)

```

```

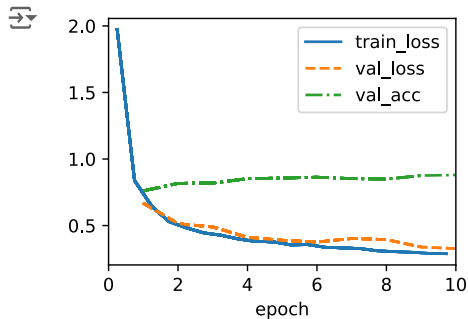
class ResNet(d2l.Classifier):
    def b1(self):
        return nn.Sequential(
            nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1))

```

```

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)

```



- 이 모델은 손실 함수도 빠르게 감소하고 정확도도 크게 떨어지지 않는다. 이는 기존 모델이 val_loss는 높고 계속 변동함을 보이면서 매우 높은 정확도를 보인것과 비교된다. 이는 기존 모델이 유연한 네트워크이며 fashion-MNIST와 같은 간단한 데이터에서는 데이터셋의 특수한 패턴에 맞춰졌기 때문으로 생각되며, 배치 정규화가 ResNet의 유연성 측면을 구성한다는 것을 알 수 있다.

✎ Skip-Connection이 제외된 모델

```

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()

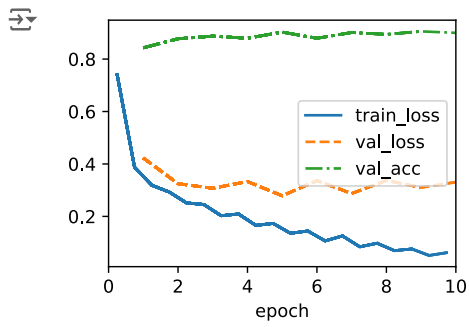
    def forward(self, X):
        Y = F.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3:
            X = self.conv3(X)
        return F.relu(Y)

```

```

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)

```



- 이 모델과 기존 모델은 거의 차이가 없는 것으로 보인다. 발견할 수 있는 사실은 기존 모델의 손실 함수가 아주 약간 빠르게 적합되고 val_loss의 변동성이 조금 크다는 점이다. 이러한 점에서 fashion-MNIST 데이터셋과 사용된 모델의 계층 수 수준에서는 skip-connection이 큰 필요가 없음을 보여주는 것으로 생각된다.