딥러닝 HW 2

2020320131

컴퓨터학과 박수빈

7.1. From Fully Connected Layers to Convolutions

네트워크는 초기 레이어에서 이미지의 어느 특정한 부분을 고려하지 않아야 하고, 동일한 패치에 유사하게 응답해야 한다. CNN은 이 공간 불변성을 더 적은 매개변수로 유용한 표현을 학습하는 데에 활용한다. 매개변수를 줄임으로써 이미지의 복잡도를 낮추고 보다 효율적인 처리를 할 수 있다.

7.2. Convolutions for Images

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

```
[2]: def corr2d(X, K):
         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
```

```
[3]: 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)
```

```
[3]: tensor([[19., 25.],
             [37., 43.]])
```

```
[4]: 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
```

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

```
[5]: tensor([[1., 1., 0., 0., 0., 0., 1., 1.],
             [1., 1., 0., 0., 0., 0., 1., 1.],
             [1., 1., 0., 0., 0., 0., 1., 1.],
             [1., 1., 0., 0., 0., 0., 1., 1.],
             [1., 1., 0., 0., 0., 0., 1., 1.],
             [1., 1., 0., 0., 0., 0., 1., 1.]])
```

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

Input과 kernel 간의 행렬 연산을 수행하는 코드이다.

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

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

```
[8]: corr2d(X.t(), K)
```

```
[8]: tensor([[0., 0., 0., 0., 0.],
             [0., 0., 0., 0., 0.],
             [0., 0., 0., 0., 0.],
             [0., 0., 0., 0., 0.],
             [0., 0., 0., 0., 0.],
             [0., 0., 0., 0., 0.],
             [0., 0., 0., 0., 0.],
             [0., 0., 0., 0., 0.]])
```

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

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

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

```
epoch 2, loss 5.892
epoch 4, loss 1.333
epoch 6, loss 0.365
epoch 8, loss 0.119
epoch 10, loss 0.044
```

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

```
[10]: tensor([[ 1.0097, -0.9683]])
```

반복할수록 loss 값이 줄어드는 것을 확인할 수 있다.

7.3. Padding and Stride

```
[1]: import torch
     from torch import nn
```

```
[2]: 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
```

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

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

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

[4]:
```
conv2d = nn.LazyConv2d(1, kernel_size=3, padding=1, stride=2)
comp_conv2d(conv2d, X).shape
```

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

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

[5]:
```
torch.Size([2, 2])
```

Padding을 함으로써 input의 사이즈를 유지할 수 있고, stride를 이용해 output의 해상도를 조절할 수 있다.

7.4. Multiple Input and Multiple Output Channels

[1]:
```
import torch
from d2l import torch as d2l
```

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

[3]:
```
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)
```

[3]:
```
tensor([[ 56.,  72.],
        [104., 120.]])
```

[4]:
```
def corr2d_multi_in_out(X, K):
    return torch.stack([corr2d_multi_in(X, k) for k in K], 0)
```

[5]:
```
K = torch.stack((K, K + 1, K + 2), 0)
K.shape
```

[5]:
```
torch.Size([3, 2, 2, 2])
```

[6]:
```
corr2d_multi_in_out(X, K)
```

[6]:
```
tensor([[[ 56.,  72.],
         [104., 120.]],

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

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

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

[8]:
```
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
```

Input과 output data가 multiple channels를 가져도 계산할 수 있다.

7.5. Pooling

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

[2]:
```
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
```

[3]:
```
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))
```

[3]:
```
tensor([[4., 5.],
        [7., 8.]])
```

[4]:
```
pool2d(X, (2, 2), 'avg')
```

[4]:
```
tensor([[2., 3.],
        [5., 6.]])
```

[5]:
```
X = torch.arange(16, dtype=torch.float32).reshape((1, 1, 4, 4))
X
```

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

[6]:
```
pool2d = nn.MaxPool2d(3)
pool2d(X)
```

[6]:
```
tensor([[[[10.]]]])
```

[7]:
```
pool2d = nn.MaxPool2d(3, padding=1, stride=2)
pool2d(X)
```

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

[8]:
```
pool2d = nn.MaxPool2d((2, 3), stride=(2, 3), padding=(0, 1))
pool2d(X)
```

[8]:
```
tensor([[[[ 5.,  7.],
          [13., 15.]]]])
```

[9]:
```
X = torch.cat((X, X + 1), 1)
X
```

[9]:
```
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.]]]])
```

[10]:
```
pool2d = nn.MaxPool2d(3, padding=1, stride=2)
pool2d(X)
```

[10]:
```
tensor([[[[ 5.,  7.],
          [13., 15.]],

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

Pooling layer는 convolutional layer와 비슷하지만 파라미터를 포함하지 않는다. Pooling window에서 보통 최댓값이나 평균값을 계산하는데 이를 각각 maximum pooling, average pooling이라고 하며, maximum pooling이 선호되는 경우가 더 많다.

7.6. Convolutional Neural Networks (LeNet)

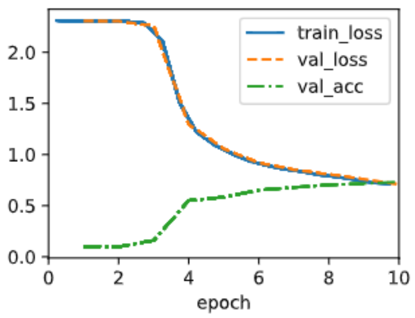
[1]:
```
import torch
from torch import nn
from d2l import torch as d2l
```

[2]:
```
def init_cnn(module):
    if type(module) == nn.Linear or type(module) == nn.Conv2d:
        nn.init.xavier_uniform(module.weight)

class LeNet(d2l.Classifier):
    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))
```

[3]:
```
@d2l.add_to_class(d2l.Classifier)
def layer_summary(self, X_shape):
    X = torch.randn(*X_shape)
    for layer in self.net:
        X = layer(X)
        print(layer.__class__.__name__, 'output shape:\t', X.shape)

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

```
Conv2d output shape:     torch.Size([1, 6, 28, 28])
Sigmoid output shape:    torch.Size([1, 6, 28, 28])
AvgPool2d output shape:  torch.Size([1, 6, 14, 14])
Conv2d output shape:     torch.Size([1, 16, 10, 10])
Sigmoid output shape:    torch.Size([1, 16, 10, 10])
AvgPool2d output shape:  torch.Size([1, 16, 5, 5])
Flatten output shape:    torch.Size([1, 400])
Linear output shape:     torch.Size([1, 120])
Sigmoid output shape:    torch.Size([1, 120])
Linear output shape:     torch.Size([1, 84])
Sigmoid output shape:    torch.Size([1, 84])
Linear output shape:     torch.Size([1, 10])
```

[4]:
```
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)
```



LeNet은 두 개의 convolutional layers로 구성된 convolutional encoder와 세 개의 fully connected layers로 구성된 dense block, 총 두 부분으로 이루어져 있다. Convolutional block의 결과를 dense block으로 넘기기 위해서는 minibatch의 각 예시를 평면화 해야 한다.

8.2. Networks Using Blocks (VGG)

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

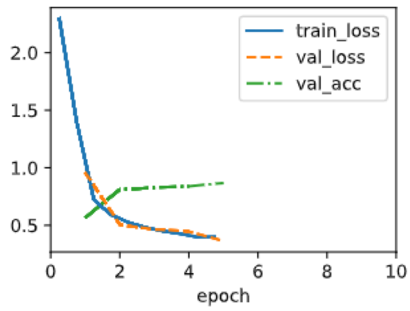
[2]:
```
def vgg_block(num_convs, out_channels):
    layers = []
    for _ in range(num_convs):
        layers.append(nn.LazyConv2d(out_channels, kernel_size=3, padding=1))
        layers.append(nn.ReLU())
    layers.append(nn.MaxPool2d(kernel_size=2,stride=2))
    return nn.Sequential(*layers)
```

[3]:
```
class VGG(d2l.Classifier):
    def __init__(self, arch, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        conv_blks = []
        for (num_convs, out_channels) in arch:
            conv_blks.append(vgg_block(num_convs, out_channels))
        self.net = nn.Sequential(
            *conv_blks, nn.Flatten(),
            nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(0.5),
            nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(0.5),
            nn.LazyLinear(num_classes))
        self.net.apply(d2l.init_cnn)
```

[4]:
```
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_dataloader(True)))[0]], d2l.init_cnn)
trainer.fit(model, data)
```



(그래프가 그려지는 데 시간이 너무 많이 소요되어 다 그려지지 못한 채로 첨부한 점 양해 부탁드립니다.)

VGG block은 padding 1과 3*3 kernels, stride 2와 2*2 max-pooling layer로 이루어진 convolutions의 연속으로 구성된다. VGG 네트워크는 convolutional pooling layers와 fully connected layers, 두 부분으로 구성된다.

8.6. Residual Networks (ResNet) and ResNeXt

[1]:
```
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l
```

[2]:
```
class Residual(nn.Module):
    def __init__(self, num_channels, use_1x1conv=False, strides=1):
        super().__init__()
        self.conv1 = nn.LazyConv2d(num_channels, kernel_size=3, padding=1,
                                   stride=strides)
        self.conv2 = nn.LazyConv2d(num_channels, kernel_size=3, padding=1)
        if use_1x1conv:
            self.conv3 = nn.LazyConv2d(num_channels, kernel_size=1,
                                       stride=strides)
        else:
            self.conv3 = None
        self.bn1 = nn.LazyBatchNorm2d()
        self.bn2 = nn.LazyBatchNorm2d()

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

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

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

[4]:
```
blk = Residual(6, use_1x1conv=True, strides=2)
blk(X).shape
```

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

[5]:
```
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))
```

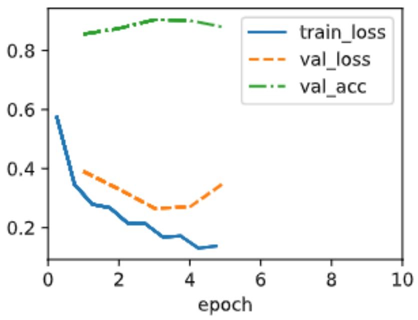
[6]:
```
@d2l.add_to_class(ResNet)
def block(self, num_residuals, num_channels, first_block=False):
    blk = []
    for i in range(num_residuals):
        if i == 0 and not first_block:
            blk.append(Residual(num_channels, use_1x1conv=True, strides=2))
        else:
            blk.append(Residual(num_channels))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

```
[7]: @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)
```

```
[8]: 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)
```



(그래프가 그려지는 데 시간이 너무 많이 소요되어 다 그려지지 못한 채로 첨부한 점 양해 부탁드립니다.)

ResNet은 VGG의 full convolutional layer design을 갖는다. Residual block은 같은 수의 output channel을 가진 두 개의 convolutional layers를 가지고, 그 뒤에는 batch normalization layer와 ReLU 활성화 함수가 이어진다.