COSE474-2024F: Deep Learning HW!

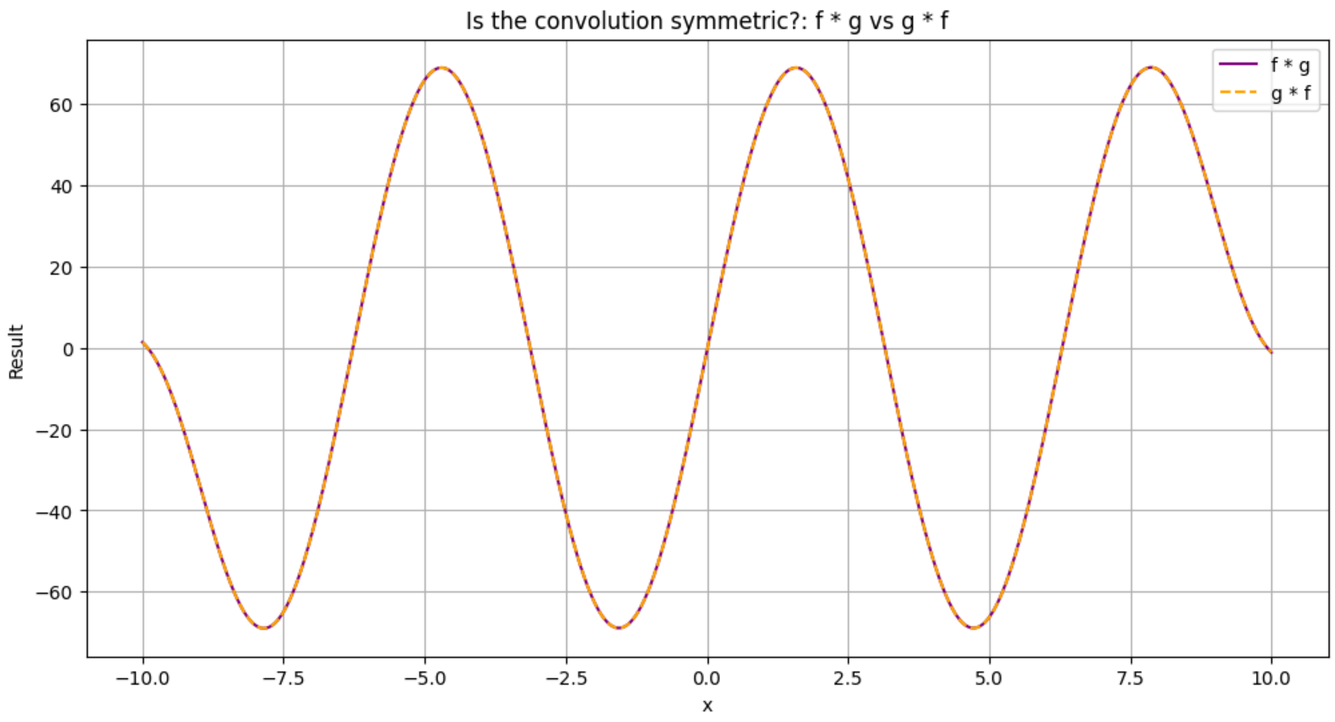
7.1 From Fully Connected Layers to Convolutions

```
pip install d2l==1.0.3
```

```
Requirement already satisfied: d2l==1.0.3 in /usr/local/lib/python3.10/dist-packages (1.0.3)
Requirement already satisfied: jupyter==1.0.0 in /usr/local/lib/python3.10/dist-packages (frc
Requirement already satisfied: numpy==1.23.5 in /usr/local/lib/python3.10/dist-packages (from
Requirement already satisfied: matplotlib==3.7.2 in /usr/local/lib/python3.10/dist-packages (
Requirement already satisfied: matplotlib-inline==0.1.6 in /usr/local/lib/python3.10/dist-pac
Requirement already satisfied: requests==2.31.0 in /usr/local/lib/python3.10/dist-packages (f
Requirement already satisfied: pandas==2.0.3 in /usr/local/lib/python3.10/dist-packages (from
Requirement already satisfied: scipy==1.10.1 in /usr/local/lib/python3.10/dist-packages (from
Requirement already satisfied: notebook in /usr/local/lib/python3.10/dist-packages (from jupy
Requirement already satisfied: qtconsole in /usr/local/lib/python3.10/dist-packages (from jup
Requirement already satisfied: jupyter-console in /usr/local/lib/python3.10/dist-packages (fr
Requirement already satisfied: nbconvert in /usr/local/lib/python3.10/dist-packages (from jup
Requirement already satisfied: ipykernel in /usr/local/lib/python3.10/dist-packages (from jup
Requirement already satisfied: ipywidgets in /usr/local/lib/python3.10/dist-packages (from ju
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.10/dist-packages (f
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-packages (from
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.10/dist-packages (
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.10/dist-packages (
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (fr
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.10/dist-packages (from
Requirement already satisfied: pyparsing<3.1,>=2.3.1 in /usr/local/lib/python3.10/dist-packag
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.10/dist-package
Requirement already satisfied: traitlets in /usr/local/lib/python3.10/dist-packages (from mat
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from
Requirement already satisfied: tzdata>=2022.1 in /usr/local/lib/python3.10/dist-packages (frc
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-pac
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from pyth
Requirement already satisfied: ipython-genutils in /usr/local/lib/python3.10/dist-packages (f
Requirement already satisfied: ipython>=5.0.0 in /usr/local/lib/python3.10/dist-packages (frc
Requirement already satisfied: jupyter-client in /usr/local/lib/python3.10/dist-packages (frc
Requirement already satisfied: tornado>=4.2 in /usr/local/lib/python3.10/dist-packages (from
Requirement already satisfied: widgetsnbextension~=3.6.0 in /usr/local/lib/python3.10/dist-pa
Requirement already satisfied: jupyterlab-widgets>=1.0.0 in /usr/local/lib/python3.10/dist-pa
Requirement already satisfied: prompt-toolkit!=3.0.0,!=3.0.1,<3.1.0,>=2.0.0 in /usr/local/lib
Requirement already satisfied: pygments in /usr/local/lib/python3.10/dist-packages (from jupy
Requirement already satisfied: lxml in /usr/local/lib/python3.10/dist-packages (from nbconver
Requirement already satisfied: beautifulsoup4 in /usr/local/lib/python3.10/dist-packages (frc
Requirement already satisfied: bleach in /usr/local/lib/python3.10/dist-packages (from nbconv
Requirement already satisfied: defusedxml in /usr/local/lib/python3.10/dist-packages (from nb
Requirement already satisfied: entrypoints>=0.2.2 in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: jinja2>=3.0 in /usr/local/lib/python3.10/dist-packages (from r
Requirement already satisfied: jupyter-core>=4.7 in /usr/local/lib/python3.10/dist-packages (
Requirement already satisfied: jupyterlab-pygments in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-packages (fr
Requirement already satisfied: mistune<2,>=0.8.1 in /usr/local/lib/python3.10/dist-packages (
Requirement already satisfied: nbclient>=0.5.0 in /usr/local/lib/python3.10/dist-packages (fr
Requirement already satisfied: nbformat>=5.1 in /usr/local/lib/python3.10/dist-packages (from
Requirement already satisfied: pandocfilters>=1.4.1 in /usr/local/lib/python3.10/dist-package
Requirement already satisfied: tinycss2 in /usr/local/lib/python3.10/dist-packages (from nbcc
Requirement already satisfied: pyzmq<25,>=17 in /usr/local/lib/python3.10/dist-packages (from
Requirement already satisfied: argon2-cffi in /usr/local/lib/python3.10/dist-packages (from r
Requirement already satisfied: nest-asyncio>=1.5 in /usr/local/lib/python3.10/dist-packages (
Requirement already satisfied: Send2Trash>=1.8.0 in /usr/local/lib/python3.10/dist-packages (
```

7.1.6 Exercise 6

```python
import numpy as np
import matplotlib.pyplot as plt

def f(x):
    return np.exp(-x**2)

def g(x):
    return np.sin(x)

def conv(f, g, x):
    return np.convolve(f(x), g(x), mode='same')

x = np.linspace(-10, 10, 1000)

conv_of_fg = conv(f, g, x)
conv_of_gf = conv(g, f, x)

plt.figure(figsize=(12, 6))
plt.plot(x, conv_of_fg, label='f * g', color='purple')
plt.plot(x, conv_of_gf, label='g * f', color='orange', linestyle='dashed')
plt.title('Is the convolution symmetric?: f * g vs g * f')
plt.xlabel('x')
plt.ylabel('Result')
plt.legend()
plt.grid()
plt.show()
```



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 4.637
epoch 4, loss 1.238
epoch 6, loss 0.396
epoch 8, loss 0.144
epoch 10, loss 0.056
```

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

```
tensor([[ 1.0153, -0.9675]])
```

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])
```

7.4 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
```

7.5 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.]]]])
```

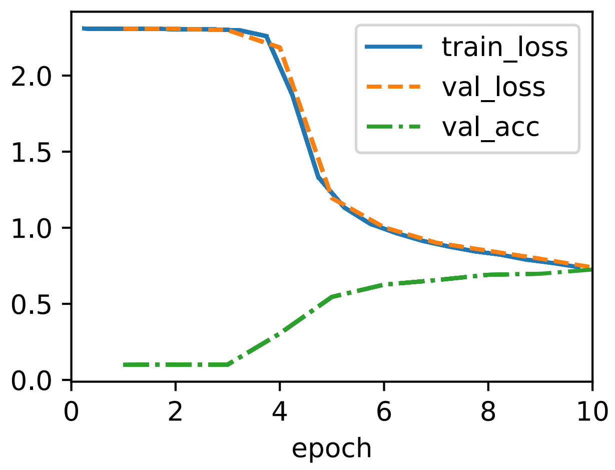
7.6 Convolutional Neural Networks (LeNet)

```
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:₩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])
```

```
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)
```

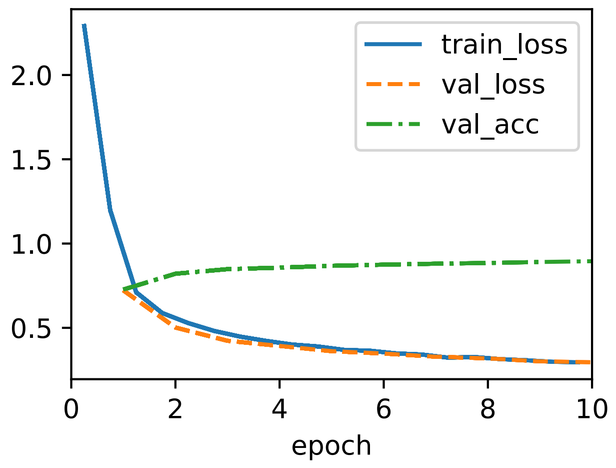


8.2 Networks Using Blocks (VGG)

```
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.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(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)


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)
```



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.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)


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

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

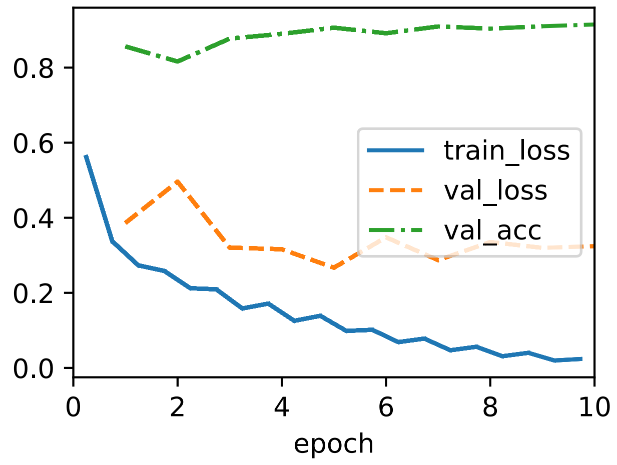
```
blk = Residual(6, use_1x1conv=True, strides=2)
blk(X).shape
```

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

```
class ResNet(d2l.Classifier):
    def b1(self):
        return nn.Sequential(
            nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),
            nn.LazyBatchNorm2d(), 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.LazyLinear(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)
```



Discussions & Exercises

7.1 From Fully Connected Layers to Convolutions

7.1.2. Constraining the MLP

- Let $[\mathbf{X}]_{i,j}$ and $[\mathbf{H}]_{i,j}$ denote the pixel at location (i, j) in the input image and hidden representation, respectively. Consequently, to have each of the hidden units receive input from each of the input pixels, we would switch from using weight matrices (as we did previously in MLPs) to representing our parameters as fourth-order weight tensors W. Suppose that $\mathbf{U}$ contains biases, we could formally express the fully connected layer as

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

7.1.2.2 Locality

- As motivated above, 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

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

- Convolutional neural networks (CNNs) are a special family of neural networks that contain convolutional layers. In the deep learning research community, V is referred to as a convolution kernel, a filter, or simply the layer’s weights that are learnable parameters.

7.1.3 Convolutions

- In mathematics, the convolution between two functions (Rudin, 1973), say $f, g : \mathbb{R}^d \to \mathbb{R}$ is defined as

$$(f * g)(\mathbf{x}) = \int f(\mathbf{z}) g(\mathbf{x} - \mathbf{z}) d\mathbf{z}.$$

7.1.4 Channels

- We could think of the hidden representations as comprising a number of two-dimensional grids stacked on top of each other. As in the inputs, these are sometimes called channels.

They are also sometimes called feature maps, as each provides a spatialized set of learned features for the subsequent layer. Intuitively, you might imagine that at lower layers that are closer to inputs, some channels could become specialized to recognize edges while others could recognize textures.

- To support multiple channels in both inputs (X) and hidden representations (H), we can add a fourth coordinate to V: $[\mathsf{V}]_{a,b,c,d}$. Putting everything together we have:

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

7.2 Convolutions for Images

7.2.1 The Cross-Correlation Operation

- Note that along each axis, the output size is slightly smaller than the input size. Because the kernel has width and height greater than 1, we can only properly compute the cross-correlation for locations where the kernel fits wholly within the image, the output size is given by the input size $n_\textrm{h} \times n_\textrm{w}$ minus the size of the convolution kernel $k_\textrm{h} \times k_\textrm{w}$ via

$$(n_\textrm{h}-k_\textrm{h}+1) \times (n_\textrm{w}-k_\textrm{w}+1).$$

7.2.1 corr2d 메소드

- `def corr2d(X, K):    #@save`
    - X는 input tensor이고, K는 kernel tensor로 2D cross-correlation 연산에 사용된다.

- `h, w = K.shape`
    - K.shape: 커널 K의 shape인 (높이, 너비)를 return 한다.
    - h, w = K.shpae: 슬라이드 연산을 통해 K의 높이와 너비 값을 각각 변수 h와 w에 저장한다.

- `Y = torch.zero((X.shape[0] - h + 1, X.shape[1] - w +1))`
    - 위의 $(n_\textrm{h}-k_\textrm{h}+1) \times (n_\textrm{w}-k_\textrm{w}+1).$에 적용해 output tensor의 크기를 계산한다.

  - torch.zero(): output tensor를 0으로 초기화한다.

- `for i in range(Y.shape[0]):`
  `for j in range(Y.shape[1])):`

  - range(Y.shape[0]): output tensor의 높이만큼 반복한다.
  - range(Y.shape[1]): output tensor의 너비만큼 반복한다.

- `Y[i, j] = (X[i:i + h, j:j + w] * K).sum()`

  - X[i:i + h, j:j + w]: K의 크기(h, w)만큼 잘라서 X의 부분 영역을 가져온다.
  - X[i:i + h, j:j + w] * K: X의 부분 영역과 K(커널)을 element-wise로 곱한다.
  - .sum(): X의 부분 영역과 K를 element-wise로 곱한 값을 모두 더한 결과를 반환한다.
  - Y[i, j] : 반환 값을 Y(output tensor)의 i행, j열 위치에 저장한다.

7.2.2 Conv2D 클래스

- A convolutional layer cross-correlates the input and kernel and adds a scalar bias to produce an output.
- The two parameters of a convolutional layer are the kernel and the scalar bias. When training models based on convolutional layers, we typically initialize the kernels randomly, just as we would with a fully connected layer.

- `class Conv2D(nn.Module):`

  - PyTorch의 nn.Module 클래스를 상속받아 Conv2D라는 클래스를 정의한다.

- `def __init__(self, kernel_size):`

  - init 메소드는 클래스의 초기화 메소드이다. kernel_size는 커널 사이즈에 대한 변수로 커널의 (높이, 너비)를 의미한다.

- `super().__init__():`

  - 부모 클래스인 nn.Module의 초기화 메소드를 호출한다.

- `self.weight = nn.Parameter(torch.rand(kernel_size)):`

    - self.weight는 convolutional layer의 parameter인 kernel을 의미한다.
    - torch.rand(kernel_size): training model이 convolutional layer에 기초할 때는 kernel을 랜덤하게 초기화하므로 torch.rand()를 호출하여 kernel_size 크기의 랜덤 값을 가지는 텐서를 생성한다.
    - nn.Parameter(): kernel은 convolutional layer의 learnable parameter이므로 이 파라미터가 학습 과정에서 업데이트될 수 있도록 한다.

- `self.bias = nn.Parameter(torch.zeros(1)):`

    - self.bias는 convolutional layer의 parameter인 bias를 의미한다.
    - torch.zeros(1): bias는 scalar bias 즉, 스칼라 값이므로 크기가 1인 tensor를 만들고 0으로 초기화한다.
    - nn.Parameter(): bias는 convolutional layer의 learnable parameter이므로 이 파라미터가 학습 과정에서 업데이트될 수 있도록 한다.

7.3 Padding and Stride

7.3.1 Padding

- Since we typically use small kernels, for any given convolution we might only lose a few pixels but this can add up as we apply many successive convolutional layers.
- One straightforward solution to this problem is to add extra pixels of filler around the boundary of our input image, thus increasing the effective size of the image. Typically, we set the values of the extra pixels to zero.
- In general, if we add a total of p_h rows of padding (roughly half on top and half on bottom) and a total of p_w columns of padding (roughly half on the left and half on the right), the output shape will be

$$(n_\mathrm{h} - k_\mathrm{h} + p_\mathrm{h} + 1) \times (n_\mathrm{w} - k_\mathrm{w} + p_\mathrm{w} + 1)$$

.

- In many cases, we will want to set $p_\mathrm{h} = k_\mathrm{h} - 1$ and $p_\mathrm{w} = k_\mathrm{w} - 1$ to give the input and output the same height and width. This will make it easier to predict the output shape of each layer when constructing the network. Assuming that k_h is odd here, we will pad $p_\mathrm{h}/2$ rows on both sides of the height. If k_h is even, one possibility is to pad $\lceil p_\mathrm{h}/2\rceil$ rows on the top of the input and $\lfloor p_\mathrm{h}/2\rfloor$ rows on the bottom. We will pad both sides of the width in the same way.

7.3.1 코드 분석 (comp_conv2d 메소드 포함)

- `X = X.reshape((1, 1) + X.shape):`
    - X.shape을 하면 X의 (height, width)가 변환된다.
    - 2차원이었던 input tensor인 X를 (batch size, the number of channels, height, width)로 reshape하여 4차원으로 확장한다.
- `return Y.reshape(Y.shape[2:])`
    - output tensor인 Y에서 처음 두 차원 batch size와 the number of channels는 제외하고, height와 width만 반환한다.
- `conv2d = nn.LazyConv2d(1, kernel_size=3, padding=1)`
    - kernel_size=3: 3X3 크기의 커널
    - padding=1: input tensor의 row, column 양쪽에 1씩 패딩 적용

7.3.2 Stride

- We refer to the number of rows and columns traversed per slide as stride.
- So far, we have used strides of 1, both for height and width. Sometimes, we may want to use a larger stride.
- In general, when the stride for the height is s_h and the stride for the width is s_w, the output shape is

$$\lfloor(n_\mathrm{h}-k_\mathrm{h}+p_\mathrm{h}+s_\mathrm{h})/s_\mathrm{h}\rfloor \times \lfloor(n_\mathrm{w}-k_\mathrm{w}+p_\mathrm{w}+s_\mathrm{w})/s_\mathrm{w}\rfloor.$$

- If we set $p_\mathrm{h}=k_\mathrm{h}-1$ and $p_\mathrm{w}=k_\mathrm{w}-1$, then the output shape can be simplified to $\lfloor(n_\mathrm{h}+s_\mathrm{h}-1)/s_\mathrm{h}\rfloor \times \lfloor(n_\mathrm{w}+s_\mathrm{w}-1)/s_\mathrm{w}\rfloor$. Going a step further, if the input height and width are divisible by the strides on the height and width, then the output shape will be $(n_\mathrm{h}/s_\mathrm{h}) \times (n_\mathrm{w}/s_\mathrm{w})$. .

7.4 Multiple Input and Multiple Output Channels

7.4.1 Multiple Input Channels

- When the input data contains multiple channels, we need to construct a convolution kernel with the same number of input channels as the input data, so that it can perform cross-correlation with the input data.
- Assuming that the number of channels for the input data is c_i, the number of input channels of the convolution kernel also needs to be c_i. If our convolution kernel's window shape is $k_\mathrm{h} \times k_\mathrm{w}$, then, when $c_\mathrm{i} = 1$, we can think of our convolution kernel as just a two-dimensional tensor of shape $k_\mathrm{h} \times k_\mathrm{w}$. However, 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. Concatenating these c_i tensors together yields a convolution kernel of shape $c_\mathrm{i} \times k_\mathrm{h} \times k_\mathrm{w}$.
- Since the input and convolution kernel each have c_i channels, we can perform a cross-correlation operation on the two-dimensional tensor of the input and the two-dimensional tensor of the convolution kernel for each channel, adding the c_i results together (summing over the channels) to yield a two-dimensional tensor. This is the result of a two-dimensional cross-correlation between a multi-channel input and a multi-input-channel convolution kernel.

7.4.2. Multiple Output Channels

- Denote by c_i and c_o the number of input and output channels, respectively, and by k_h and k_w the height and width of the kernel. To get an output with multiple channels, we can create a kernel tensor of shape $c_\mathrm{i} \times k_\mathrm{h} \times k_\mathrm{w}$ for every output channel. We concatenate them on the output channel dimension, so that the shape of the convolution kernel is $c_\mathrm{o} \times c_\mathrm{i} \times k_\mathrm{h} \times k_\mathrm{w}$.
- In cross-correlation operations, the result on each output channel is calculated from the convolution kernel corresponding to that output channel and takes input from all channels in the input tensor.

7.4.2 코드 분석

-
  ```
  def corr2d_multi_in_out(X, K):
  return torch.stack([corr2d_multi_in(X,k) for k in K], 0)
  ```
    - X: multiple input channel을 가진 input tensor로 크기는 (입력 채널 수, 높이, 너비)이다.

- K: multiple output channel을 가진 kernel tensor로 크기는 (출력 채널 수, 입력 채널 수, 커널 높이, 커널 너비)이다.
- for k in K: k는 output channel 하나에 해당하는 kernel tensor로 크기는 (입력 채널 수, 커널 높이, 커널 너비)이다.

7.5 Pooling

7.5.1 Maximum Pooling and Average 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).
- 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). Instead, pooling operators are deterministic, typically calculating either the maximum or the average value of the elements in the pooling window. These operations are called maximum pooling (max-pooling for short) and average pooling, respectively.
- Average pooling is essentially as old as CNNs. The idea is akin to downsampling an image. Rather than just taking the value of every second (or third) pixel for the lower resolution image, we can average over adjacent pixels to obtain an image with better signal-to-noise ratio since we are combining the information from multiple adjacent pixels.
- Max-pooling was introduced in Riesenhuber and Poggio (1999) in the context of cognitive neuroscience to describe how information aggregation might be aggregated hierarchically for the purpose of object recognition; there already was an earlier version in speech recognition (Yamaguchi et al., 1990). In almost all cases, max-pooling, as it is also referred to, is preferable to average pooling.

7.5.1 코드 분석

- `def pool2d(X, pool_size, mode='max'):`
  - pool_size: pooling window의 크기
  - mode: Average pooling 또는 Max-pooling 모드를 선택할 수 있다.

-
  ```
  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()
  ```
    - mode가 'max'이면, 슬라이싱한 영역의 최댓값을 Y[i, j] 위치에 저장한다.
    - mode가 'avg'이면, 슬라이싱한 영역의 평균값을 Y[i, j] 위치에 저장한다.

7.6 Convolutional Neural Networks (LeNet)

7.6.1 LeNet

- At a high level, LeNet (LeNet-5) consists of two parts: (i) a convolutional encoder consisting of two convolutional layers; and (ii) a dense block consisting of three fully connected layers. The architecture is summarized in Fig. 7.6.1.
- The basic units in each convolutional block are a convolutional layer, a sigmoid activation function, and a subsequent average pooling operation. Note that while ReLUs and max-pooling work better, they had not yet been discovered. Each convolutional layer uses a 5×5 kernel and a sigmoid activation function. These layers map spatially arranged inputs to a number of two-dimensional feature maps, typically increasing the number of channels.
- In order to pass output from the convolutional block to the dense block, we must flatten each example in the minibatch. In other words, we take this four-dimensional input and transform it into the two-dimensional input expected by fully connected layers: as a reminder, the two-dimensional representation that we desire uses the first dimension to index examples in the minibatch and the second to give the flat vector representation of each example.

7.6.2 Training

- While CNNs have fewer parameters, they can still be more expensive to compute than similarly deep MLPs because each parameter participates in many more multiplications. If you have access to a GPU, this might be a good time to put it into action to speed up training.
- Note that the d2l.Trainer class takes care of all details. By default, it initializes the model parameters on the available devices. Just as with MLPs, our loss function is cross-entropy, and we minimize it via minibatch stochastic gradient descent.

8.2 Networks Using Blocks (VGG)

8.2.1 VGG Blocks

- The basic building block of CNNs is a sequence of the following: (i) a convolutional layer with padding to maintain the resolution, (ii) a nonlinearity such as a ReLU, (iii) a pooling layer such as max-pooling to reduce the resolution.
- One of the problems with this approach is that the spatial resolution decreases quite rapidly. In particular, this imposes a hard limit of $\log_2 d$ convolutional layers on the network before all dimensions (d) are used up. For instance, in the case of ImageNet, it would be impossible to have more than 8 convolutional layers in this way.
- The key idea of Simonyan and Zisserman (2014) was to use multiple convolutions in between downsampling via max-pooling in the form of a block. They were primarily interested in whether deep or wide networks perform better. For instance, the successive application of two 3×3 convolutions touches the same pixels as a single 5×5 convolution does. At the same time, the latter uses approximately as many parameters ($25 \cdot c^2$) as three 3×3 convolutions do ($3 \cdot 9 \cdot c^2$). In a rather detailed analysis they showed that deep and narrow networks significantly outperform their shallow counterparts.
- Back to VGG: a VGG block consists of a sequence of convolutions with 3×3 kernels with padding of 1 (keeping height and width) followed by a 2×2 max-pooling layer with stride of 2 (halving height and width after each block)

8.2.1 코드 분석

-
  ```
  def vgg_block(num_convs, out_channels):
  ```
  - VGG 블록을 정의하는 함수
  - num_convs: convolutional layers의 수
  - out_channels: output channels의 수

-
  ```
  for _ in range(num_convs):
      layers.append(nn.LazyConv2d(out_channels, kernel_size=3, padding=1))
      layers.append(nn.ReLU())
  ```
  - num_convs만큼 for문이 반복된다.

  - 각 반복마다 Conv2D 계층과 ReLU 활성화 함수를 추가한다.

- `layers.append(nn.MaxPool2d(kernel_size=2, stride=2))`

  - Conv2D 계층과 ReLU 추가가 끝나면, MaxPool2d 계층을 추가한다.

- `return nn.Sequential(*layers)`

  - layers에 저장된 모든 계층들을 하나로 묶어 순차적으로 실행되도록 한다.

8.2.2 VGG Network

- Like AlexNet and LeNet, the VGG Network can be partitioned into two parts: the first consisting mostly of convolutional and pooling layers and the second consisting of fully connected layers that are identical to those in AlexNet. The key difference is that the convolutional layers are grouped in nonlinear transformations that leave the dimensonality unchanged, followed by a resolution-reduction step, as depicted in Fig. 8.2.1.
- The convolutional part of the network connects several VGG blocks from Fig. 8.2.1 (also defined in the vgg_block function) in succession. This grouping of convolutions is a pattern that has remained almost unchanged over the past decade, although the specific choice of operations has undergone considerable modifications.

8.2.2 코드 분석

- `conv_blks = []`

  - conv_blks는 VGG 블록들을 저장할 리스트이다.

-
  ```
  for (num_convs, out_channels) in arch:
      conv_blks.append(vgg_block(num_convs, out_channels))
  ```

  - The variable arch consists of a list of tuples (one per block), where each contains two values: the number of convolutional layers and the number of output channels, which are precisely the arguments required to call the vgg_block function.
  - arch 리스트에 있는 각 블록의 num_convs와 out_channels를 매개변수로 vgg_block 함수를 사용하여 VGG 블록을 생성하고, 이를 conv_blks 리스트에 append한다.

-
  ```
  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.net은 모델의 전체 네트워크를 의미한다.
    - *conv_blks: 앞서 생성한 VGG 블록들
    - nn.Flatten(): fully connected layer로 전달하기 위해 representations을 flatten한다.
    - nn.LazyLinear(4096): 4096개의 뉴런을 가진 fully connected layer이다.
    - nn.ReLU(): 활성화 함수로 비선형성을 추가한다.
    - nn.Dropout(0.5): 드롭아웃 계층을 추가하여 과적합을 방지한다.
    - nn.LazyLinear(num_classes): 분류할 class의 수만큼 output을 생성하는 fully connected layer이다.

8.6 Residual Networks (ResNet) and ResNeXt

8.6.1 Function Classes

- 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:
$$f^*_\mathcal{F} \stackrel{\text{def}}{=} \underset{f}{\operatorname{argmin}}\, L(\mathbf{X}, \mathbf{y}, f) \text{ subject to } f \in \mathcal{F}.$$
- This is the question that He et al. (2016) considered when working on very deep computer vision models. At the heart of their proposed residual network (ResNet) is the idea that every additional layer should more easily contain the identity function as one of its elements. These considerations are rather profound but they led to a surprisingly simple solution, a residual block. With it, ResNet won the ImageNet Large Scale Visual Recognition Challenge in 2015.

8.6.2 Residual Blocks

- We assume that $f(\mathbf{x})$, the desired underlying mapping we want to obtain by learning, is to be used as input to the activation function on the top. 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 $g(\mathbf{x}) = f(\mathbf{x}) - \mathbf{x}$, which is how the residual block derives its name.
- If the identity mapping $f(\mathbf{x}) = \mathbf{x}$ is the desired underlying mapping, the residual mapping amounts to $g(\mathbf{x}) = 0$ and it is thus easier to learn: we only need to push the weights and biases of the upper weight layer (e.g., fully connected layer and convolutional layer) within the dotted-line box to zero. The right figure illustrates the residual block of ResNet, where the solid line carrying the layer input $\mathbf{x}$ to the addition operator is called a residual connection (or shortcut connection). 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.

8.6.3. ResNet Model

- The first two layers of ResNet are the same as those of the GoogLeNet we described before: the 7×7 convolutional layer with 64 output channels and a stride of 2 is followed by the 3×3 max-pooling layer with a stride of 2. The difference is the batch normalization layer added after each convolutional layer in ResNet.
- GoogLeNet uses four modules made up of Inception blocks. However, 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.

8.6.3 코드 분석 - ResNet 클래스

- `class ResNet(d21.Classifier):`
    - ResNet의 첫 번째 합성곱 블록을 정의하고 있다.
    - d21.Classifier를 상속받는다.

-
  ```
  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))
```

- nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3): 2D 합성곱 레이어이다. 커널 크기는 7x7이며, 출력 채널 수는 64개이다.
- nn.LazyBatchNorm2d(): Batch Normalization layer이다.
- nn.ReLU(): 활성화 함수로 음수는 0으로 양수는 값을 그대로 유지하는 비선형 함수이다.
- nn.MaxPool2d(kernel_size=3, stride=2, padding=1): Max Pooling layer이다. 커널 크기는 3x3이며, stride의 값은 2로 설정했기 때문에 2칸씩 이동하며 pooling을 수행한다.

8.6.3 코드 분석 - block 메소드

- `@d2l.add_to_class(ResNet)`
  - block 메소드를 ResNet 클래스에 동적으로 추가하는 역할을 한다.

- `def block(self, num_residuals, num_channels, first_block=False):`
  - num_resideuals: ResNet 블록에 추가할 residual block의 수를 의미한다.
  - first_block=False: first_block은 해당 블록이 첫번째 블록인지를 판단하기 위해 사용된다, 기본값은 False이다.

- `blk = []`
  - 리스트의 형태로 residual block들을 저장할 공간을 정의해준다.

-
```
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)
```
  - num_residuals 값만큼 for문을 반복하며 blk.append()을 이용해서 blk 리스트에 생성한 residual block들을 추가해준다.
  - if i == 0 and not first_block: 첫 번째 residual block일 경우

  - else: 첫 번째 residual block이 아닐 경우
  - use_1x1conv는 input의 크기와 output의 크기가 다른 경우 1x1 합성곱을 하기 위해 사용된다. 첫 번째 residual block일 경우 이를 True로, 첫 번째 residual block이 아닐 경우에는 이를 False로 설정해주었다.
  - nn.Sequential(*blk): 생성한 residual block들을 blk 리스트에 모두 추가했으면, nn.Sequential()을 이용해서 residual block들을 하나로 묶어 return한다.

8.6.4 Training

- We train ResNet on the Fashion-MNIST dataset, just like before. ResNet is quite a powerful and flexible architecture. The plot capturing training and validation loss illustrates a significant gap between both graphs, with the training loss being considerably lower. For a network of this flexibility, more training data would offer distinct benefit in closing the gap and improving accuracy.

8.6.4 코드 분석

- `model = ResNet18(lr=0.01)`
  - lr은 learning rate를 의미하며, 학습률을 0.01로 설정한다.
- `trainer = d2l.Trainer(max_epochs=10, num_gpus=1)`
  - max_epochs는 최대 학습 에폭 수를 의미한다. 10으로 설정했기 때문에, 10번 반복해서 학습한다.
  - num_gpus는 학습에 사용할 GPU의 수를 의미한다.