7.1 From Fully Connected Layers to Convolutions

7.2 Convolutions for Images

```
import torch
from torch import nn
!pip install d2l
from d2l import torch as d2l
```

```
Requirement already satisfied: ipython-genutils in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: ipython>=5.0.0 in /usr/local/lib/python3.10/dist-packages (f
Requirement already satisfied: jupyter-client in /usr/local/lib/python3.10/dist-packages (f
Requirement already satisfied: tornado>=4.2 in /usr/local/lib/python3.10/dist-packages (fro
Requirement already satisfied: widgetsnbextension~=3.6.0 in /usr/local/lib/python3.10/dist-
Requirement already satisfied: jupyterlab-widgets>=1.0.0 in /usr/local/lib/python3.10/dist-
Requirement already satisfied: prompt-toolkit!=3.0.0,!=3.0.1,<3.1.0,>=2.0.0 in /usr/local/l
Requirement already satisfied: pygments in /usr/local/lib/python3.10/dist-packages (from ju
Requirement already satisfied: lxml in /usr/local/lib/python3.10/dist-packages (from nbconv
Requirement already satisfied: beautifulsoup4 in /usr/local/lib/python3.10/dist-packages (f
Requirement already satisfied: bleach in /usr/local/lib/python3.10/dist-packages (from nbco
Requirement already satisfied: defusedxml in /usr/local/lib/python3.10/dist-packages (from
Requirement already satisfied: entrypoints>=0.2.2 in /usr/local/lib/python3.10/dist-package
Requirement already satisfied: jinja2>=3.0 in /usr/local/lib/python3.10/dist-packages (from
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-packag
```

```
Requirement already satisfied: attrs>=22.2.0 in /usr/local/lib/python3.10/dist-packages (fr
Requirement already satisfied: jsonschema-specifications>=2023.03.6 in /usr/local/lib/pytho
Requirement already satisfied: referencing>=0.28.4 in /usr/local/lib/python3.10/dist-packag
Requirement already satisfied: rpds-py>=0.7.1 in /usr/local/lib/python3.10/dist-packages (f
Requirement already satisfied: jupyter-server<3,>=1.8 in /usr/local/lib/python3.10/dist-pac
Requirement already satisfied: cffi>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from
Requirement already satisfied: pycparser in /usr/local/lib/python3.10/dist-packages (from c
Requirement already satisfied: anyio<4,>=3.1.0 in /usr/local/lib/python3.10/dist-packages (
Requirement already satisfied: websocket-client in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: sniffio>=1.1 in /usr/local/lib/python3.10/dist-packages (fro
Requirement already satisfied: exceptiongroup in /usr/local/lib/python3.10/dist-packages (f
```

```
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.]])
```

```
# Construct a two-dimensional convolutional layer with 1 output channel and a
# kernel of shape (1, 2). For the sake of simplicity, we ignore the bias here
conv2d = nn.LazyConv2d(1, kernel_size=(1, 2), bias=False)

# The two-dimensional convolutional layer uses four-dimensional input and
# output in the format of (example, channel, height, width), where the batch
# size (number of examples in the batch) and the number of channels are both 1
X = X.reshape((1, 1, 6, 8))
Y = Y.reshape((1, 1, 6, 7))
lr = 3e-2  # Learning rate

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

```
epoch 2, loss 3.749
epoch 4, loss 0.986
epoch 6, loss 0.312
epoch 8, loss 0.112
epoch 10, loss 0.043
```

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

```
tensor([[ 1.0132, -0.9711]])
```

7.3 Padding and Stride

```
import torch
from torch import nn


# We define a helper function to calculate convolutions. It initializes the
# convolutional layer weights and performs corresponding dimensionality
# elevations and reductions on the input and output
def comp_conv2d(conv2d, X):
    # (1, 1) indicates that batch size and the number of channels are both 1
    X = X.reshape((1, 1) + X.shape)
    Y = conv2d(X)
    # Strip the first two dimensions: examples and channels
    return Y.reshape(Y.shape[2:])

# 1 row and column is padded on either side, so a total of 2 rows or columns
# are added
conv2d = nn.LazyConv2d(1, kernel_size=3, padding=1)
X = torch.rand(size=(8, 8))
comp_conv2d(conv2d, X).shape
```

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

```
# We use a convolution kernel with height 5 and width 3. The padding on either
# side of the height and width are 2 and 1, respectively
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):
    # Iterate through the 0th dimension (channel) of K first, then add them up
    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):
    # Iterate through the 0th dimension of K, and each time, perform
    # cross-correlation operations with input X. All of the results are
    # stacked together
    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))
    # Matrix multiplication in the fully connected layer
    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)
# Pooling has no model parameters, hence it needs no initialization
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
```

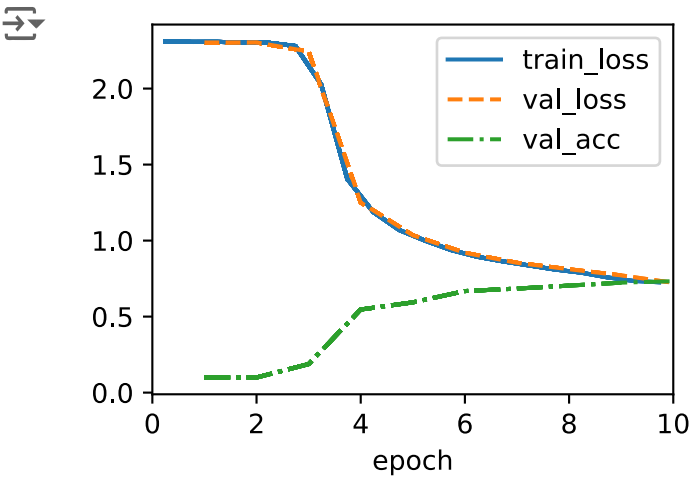
```
def init_cnn(module):
    """Initialize weights for CNNs."""
    if type(module) == nn.Linear or type(module) == nn.Conv2d:
        nn.init.xavier_uniform_(module.weight)

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)

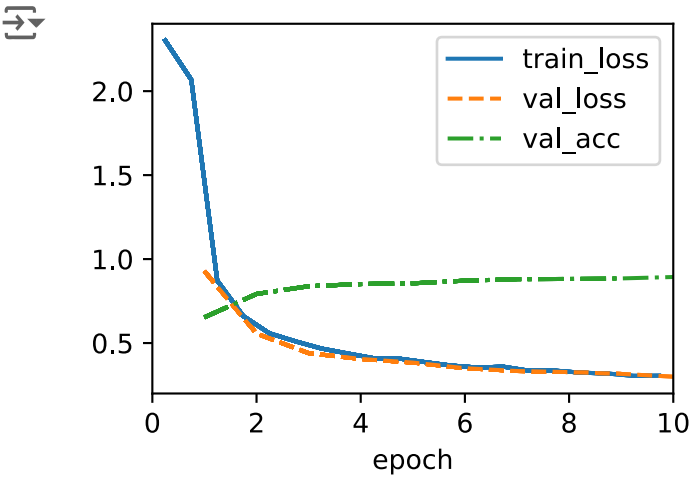
```
import torch
from torch import nn
from d2l import torch as d2l


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
```

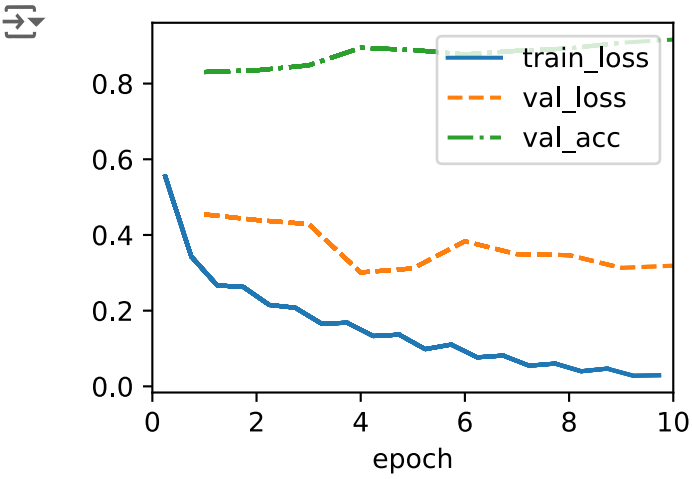
```
torch.Size([4, 6, 3, 3])
```

```
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)
```



Discussion

7.1 discussion cnn에서 사용하는 두가지 원칙들을 살펴보았다. 첫 번째는 변환 불변성이다. 이는 이미지 내에서 객체가 어디에 있든지 관계없이 네트워크가 동일하게 반응해야 한다는 원칙이다. 두 번째는 국부성이다. 네트워크의 초기 층은 이미지의 local한 영역에만 집중해야 하며, 이미지의 먼 영역은 고려하지 않아야 한다. MLP를 2차원 이미지 입력과 은닉 표현을 행렬로 표현하여 생각해볼 수 있다. 이때, 네트워크의 가중치를 4차원 텐서로 변환하면, 매개변수의 수가 너무 커질 수 있다. 하지만 첫 번째 원칙인 변환 불변성을 적용하면 가중치가 이미지 내 특정 위치에 의존하지 않으므로 매개변수 수가 대폭 줄어든다. 이 과정을 통해 합성곱 연산이 도입되며, 이는 픽셀을 가중치로 가중하여 결과 값을 얻는 방식이다.두 번째 원칙인 국부성을 적용하면, 국소적인 범위 내의 정보만 사용하게 되므로 매개변수의 수가 또 한 번 줄어든다. 이를 통해 CNN의 기본적인 구조가 형성된다. 결국 이 두 원칙은 둘 다 매개변수의 수를 줄여주는 역할을 한다. 이 두 원칙을 바탕으로 cnn은 고차원의 이미지를 처리하는 데 필요한 매개변수 수를 크게 줄이면서도, 이미지 내의 유용한 패턴을 학습할 수 있는 효율적인 구조를 제공한다. CNN의 설계가 단순히 이미지 데이터를 처리하는 것에서 그치지 않고, 컴퓨팅 자원을 절약하면서도 중요한 정보를 학습할 수 있는 구조라는 점이 인상 깊었다.. CNN의 성공은 이 두 원칙을 효과적으로 활용함으로써 가능했고, 이는 복잡한 데이터를 효율적으로 다루기 위한 신경망 설계의 진보라고 할 수 있다. 더 생각해보면, 이러한 원칙은 이미지 외에도 다양한 데이터 형식에 적용될 수 있는 가능성을 엿볼 수 있다. 변환 불변성과 국부성을 잘 이해하면, 다른 유형의 문제에 대한 더 나은 신경망 설계로 이어질 수 있다고 생각한다.

7.2 discussion 교차상관 연산은 2차원 입력 텐서와 커널 텐서를 결합하여 출력 텐서를 생성하는 과정이다. 이 연산은 입력 텐서의 좌상단에서 시작하여 커널을 입력 텐서 위에서 좌우, 상하로 이동시키면서 수행된다. 커널이 특정 위치에 도달하면, 해당 위치의 입력 텐서의 부분 집합과 커널을 원소별로 곱한 후 그 합을 구해 출력 텐서의 해당 위치에 저장한다. 출력 텐서의 크기는 입력 텐서보다 작아지는데, 이는 커널이 입력의 경계 밖으로 나갈 수 없기 때문이다. 커널의 크기만큼 입력 텐서의 크기가 줄어들어 출력 텐서가 계산된다. 만약 입력 텐서의 크기를 유지하려면, 입력의 경계에 0을 추가하는 패딩을 사용한다. 이 연산은 주로 이미지에서 패턴을 감지하거나 필터링하는 데 사용되며, 합성곱 층의 기본 연산으로 활용된다. 단순한 반복문으로 구현 가능하고 적절한 필터를 직접 설계하는 대신 데이터를 통해 학습할 수 있다는 점이 꽤 흥미로웠다.

7.3 discussion 패딩과 스트라이드는 합성곱 연산에서 입력 이미지의 크기와 출력 크기를 조절하는 중요한 기법이다. 합성곱 연산을 반복하면 입력 이미지의 경계 픽셀들이 손실되고 출력 크기가 줄어들게 되는데, 이를 해결하기 위해 패딩을 사용한다. 패딩은 입력 이미지의 가장자리에 0과 같은 값을 추가해 이미지 크기를 확장하는 방법으로, 커널이 입력의 경계를 넘어가지 않게 해 출력의 크기를 유지하거나 입력과 동일한 크기의 출력을 얻을 수 있게 한다. 일반적으로 대칭적인 패딩을 사용해 입력의 높이와 너비를 보존하는 방식으로 설정된다. 반면, 스트라이드는 커널이 입력 이미지 위를 이동하는 간격을 의미하며, 기본적으로 한 번에 한 칸씩 이동하지만, 더 큰 스트라이드를 설정하면 여러 칸을 건너뛰며 이동해 출력의 해상도를 줄일 수 있다. 스트라이드를 크게 설정하면 입력 크기보다 작은 출력 크기를 얻게 되어 해상도를 효과적으로 조절할 수 있다. 따라서 패딩은 출력 크기를 늘리기 위한 방법이고, 스트라이드는 출력 크기를 줄이는 데 사용된다. 이 두 기법을 적절히 활용하면 입력 이미지의 크기와 해상도를 효율적으로 관리할 수 있어, CNN에서 중요한 역할을 한다. 패딩과 스트라이드를 적절히 조절하면, 네트워크가 학습해야 할 매개변수의 양을 관리하

면서도 이미지의 해상도와 크기를 효과적으로 통제할 수 있다는 점에서 CNN의 성능을 최적화하는 중요한 기법인 것 같다. 이 두 기법을 어떻게 조정하느냐에 따라 모델이 특정 문제를 처리하는 방식이 달라질 수 있으므로, CNN 설계에 있어 유연성과 효율성을 극대화하는 수단으로 볼 수 있다고 생각이 들었다. stride 없이 output shape

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

stride 있이 output shape

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

7.4 discussion 만약 이미지 데이터가 여러 채널을 가지고 있다면 이를 처리하기 위해서는 다중 채널을 지원하는 합성곱 층이 필요하다. 입력 데이터에 여러 채널이 있을 때, 합성곱 커널 역시 동일한 수의 입력 채널을 가져야 한다. 각 채널마다 커널을 적용한 결과를 모두 더해 최종 출력을 계산한다. 다중 입력 채널의 합성곱 연산은 각 입력 채널에 대해 개별적으로 교차 상관 연산을 수행한 후 그 결과를 더하는 방식으로 이루어진다. 예를 들어, 두 개의 입력 채널이 있을 경우, 두 개의 2차원 입력 텐서에 대해 각각 커널을 적용하고 그 결과를 더해 최종 출력을 얻는다. 또, 출력 채널마다 별도의 커널을 생성해야 한다. 각 출력 채널은 모든 입력 채널에 대한 커널을 사용하여 교차 상관 연산을 수행한 결과로 얻어진다. 이를 통해 각 출력 채널은 입력 데이터에서 다른 특징을 학습할 수 있으며, 네트워크가 깊어질수록 채널 수가 증가하는 경향이 있다. 1x1 합성곱과 같은 경우 수업시간에도 배웠지만 인접한 픽셀 간의 상호작용을 고려하지 않고, 각 픽셀 위치에서 입력 채널을 통해 출력 채널을 계산하는 방식이다. 이는 각 픽셀에서의 정보 변환에 사용되며, 공간적 차원에서는 변환을 하지 않지만 채널 차원에서는 변환을 수행한다. 이는 복잡한 네트워크에서 종종 사용되는 중요한 연산이다. 다중 채널을 통해 CNN은 다양한 분석이 가능하다. 그러나 다중 채널을 처리하는 연산은 계산 비용이 매우 크다. 이를 최적화하기 위한 다양한 기법이 있다고 한다.

7.5 discussion Pooling 레이어는 합성곱 신경망에서 공간적으로 다운샘플링을 수행하고, 합성곱 층이 위치에 너무 민감해지는 문제를 완화하는 역할을 한다. 풀링 연산은 커널 없이 고정된 크기의 윈도우를 입력 텐서 위에서 슬라이딩하며, 해당 영역의 최대값을 선택하는 맥스 풀링과 평균값을 계산하는 평균 풀링 두 가지 방식이 있다. 이를 통해 각 위치의 특징을 요약하고, 불필요한 세부 정보를 제거해 학습 효율을 높인다. 맥스 풀링은 보통 평균 풀링보다 더 선호되며, 입력 이미지의 작은 변동에 덜 민감하게 만드는 효과가 있다. 풀링은 커널이 없는 고정 연산으로, 입력 텐서의 각 채널에 대해 별도로 수행되므로 채널 수는 변하지 않는다. 또한, 풀링은 패딩과 스트라이드를 적용하여 출력 크기를 조절할 수 있다. 예를 들어, 2x2 풀링 윈도우에 스트라이드를 2로 설정하면 출력의 크기는 입력의 절반이 된다. 풀링 레이어는 주로 맥스 풀링을 사용하여 공간 해상도를 감소시키면서도 중요한 특징을 보존하는 데 유리하다. 아까 다중 채널의 관점에서 이야기해보자면 다중 채널 처리에 있어 풀링은 각 채널을 독립적으로 처리하며, 입력 데이터가 여러 채널을 포함하는 경우에도 각 채널에 대해 동일하게 작동한다. 이는 다중 채널 입력에서 각 채널의 고유한 특징을 유지하면

서도 공간적인 다운샘플링을 수행할 수 있음을 의미한다. 예를 들어, RGB 이미지와 같은 다중 채널 데이터를 처리할 때, 각 채널의 특징을 개별적으로 처리하고 그 결과를 합치지 않기 때문에 채널 간의 정보 손실이 없다. 결과적으로 풀링은 다중 채널의 특징을 보존하며, 각 채널에서 얻은 중요한 정보를 유지하는 기능적 역할을 수행할 수 있다.

7.6 discussion LeNet은 손글씨를 인식하기 위해 설계된 cnn모델이라고 한다. LeNet의 기본 구조는 두 부분으로 나뉘는데, 첫 번째는 두 개의 합성곱 층으로 이루어진 컨볼루션 인코더이며, 두 번째는 세 개의 완전 연결 층으로 이루어진 밀집 블록이다. LeNet의 각 합성곱 블록은 합성곱 층, 시그모이드 활성화 함수, 그리고 평균 풀링 연산으로 구성된다. 각 합성곱 층은 5x5 크기의 커널을 사용하여 공간적으로 정렬된 입력을 여러 2차원 특징 맵으로 매핑한다. 첫 번째 합성곱 층은 6개의 출력 채널을 생성하며, 두 번째 합성곱 층은 16개의 출력 채널을 만든다. 평균 풀링은 2x2 크기의 윈도우로 적용되어, 공간적 다운샘플링을 통해 차원을 절반으로 줄인다. 이러한 합성곱 블록을 통해 얻은 출력은 네트워크의 완전 연결 층으로 전달되기 전에 평탄화되어 2차원 벡터로 변환된다. LeNet은 이미지의 공간적 구조를 보존하면서 학습하는 합성곱을 사용하는 것이 가장 큰 특징이라고 한다. 이는 과거의 mlp와는 다르게 위치정보를 그대로 가져갈 수 있다는 점에서 유리하다고 한다. 훈련시에는 주로 교차 엔트로피 손실 함수와 미니배치 확률적 경사 하강법을 사용한다고 한다. 어찌되었든 lenet은 딥러닝 모델의 시초격으로 lenet 이후의 딥러닝 연구는 비약적인 발전을 이루었다고 한다.

8.2 discussion 딥러닝 연구자들은 개별 뉴런에서 계층 단위로, 그리고 반복적인 블록 단위로 생각하기 시작했다고 한다. 또한 대규모의 사전에 학습된 모델, 일명 파운데이션 모델을 사용하기 시작했다고 한다. 그리고 VGG 네트워크는 이러한 블록 개념을 처음으로 도입한 모델 중 하나라고 한다. VGG 블록의 기본 구성 요소는 1.패딩을 적용한 합성곱 층, 2. ReLU 비선형성, 3. 풀링 층으로 이루어진다. 이 구성 요소를 통해 공간 해상도를 빠르게 줄일 수 있다. VGG 블록에서는 연속적인 3x3합성곱 연산을 사용하여 동일한 픽셀 영역을 처리하지만, 더 깊고 좁은 네트워크가 얕고 넓은 네트워크보다 훨씬 우수한 성능을 보인다는 사실이 발견되었다. 전체적인 구성으로는 합성곱과 풀링 층이 주를 이루는 앞부분과, 완전 연결 층으로 이루어진 뒷부분으로 나뉜다. 여러 합성곱 층을 그룹화하여 비선형 변환을 적용한 후, 해상도를 줄이는 방식을 가진다. VGG는 여러 블록으로 이루어져 있으며, 각 블록은 동일한 구조를 갖지만 출력 채널 수가 다르다. 네트워크가 깊어질수록 출력 채널 수가 점차 증가한다.VGG-11은 계산 비용이 높기 때문에, Fashion-MNIST와 같은 데이터셋에 대해 훈련할 때는 더 적은 수의 채널을 사용하는 것이 일반적이다.교차 엔트로피 손실 함수를 사용하고, 확률적 경사 하강법을 통해 손실을 최소화한다. GPU를 사용하면 훈련 속도를 크게 높일 수 있다고 한다. 결국 이 모델은 cnn의 발전 방향성을 제시했고 진화 과정에서 필수적인 역할을 담당했다는 것을 알 수 있었다.

8.6 discussion 네트워크에 레이어를 추가할 때, 네트워크가 더 강력한 표현력을 가지는지, 또는 단순히 다른 표현을 학습하는지를 이해하는 것이 필요하다고 한다. 이를 위해 우리는 함수 클래스를 사용한다고 한다. 우리가 찾고자 하는 함수가 함수 클래스 안에 존재하면 좋은 결과를 얻을 수 있지만, 그렇지 않다면 최적화 문제를 통해 근사한 함수를 찾아야 한다고 하고, 문제는 바로 함수 클래스가 클수록 반드시 더 좋은 결과를 보장하지 않는다는 점이다. 만약 함수 클래스가 중첩되지 않는

다면, 더 복잡한 네트워크를 사용해도 더 나은 결과를 얻지 못할 수 있다. 하지만 함수 클래스가 중첩된 경우에는 네트워크의 표현력을 안정적으로 증가시킬 수 있다. 이를 해결하기 위해 resnet이 등장했다고 한다. ResNet은 추가된 레이어가 항상 항등 함수를 포함할 수 있도록 설계되었다. 이는 새로운 레이어가 기존 레이어보다 더 나은 해를 찾을 수 있도록 돕는다. 이를 실현하기 위해 residual block이 사용되며, 이는 입력을 직접 출력에 더하는 residual connection을 통해 네트워크의 학습을 용이하게 만든다. residual block은 입력을 그대로 출력에 더해줌으로써, 네트워크가 항등 함수를 쉽게 학습할 수 있도록 한다. 일반적인 블록에서는 출력이 입력을 직접 학습해야 하지만, residual 블록에서는 잔차 함수만 학습하면 되므로 학습이 더 쉬워진다고 한다.residual block은 두 개의 3x3 합성곱 층과 배치 정규화, ReLU 활성화 함수로 구성되며, 입력을 그대로 출력에 더해주는 구조로 이루어져 있다. 그리고 resnet은 다시 residual block 여러개를 쌓아 만든 네트워크이다. 첫 두개의 층 이후 네트워크는 잔차 블록으로 구성된다. 각 블록은 동일한 채널 수를 가지며, 다음 블록으로 넘어가면서 채널 수가 두 배로 증가하고 공간 해상도는 절반으로 줄어든다. 가장 결정적으로 단순히 레이어를 추가한다고 해서 더 나은 성능이 보장되지 않는다는 사실은 우리가 모델을 설계할 때 얼마나 신중해야 하는지를 잘 보여주는 것 같다.

.

.

.