https://d2l.ai/chapter_preliminaries/index.html

7.1. From Fully Connected Layers to Convolutions

7.2. Convolutions for Images

7.3. Padding and Stride

7.4. Multiple Input and Multiple Output Channels

7.5. Pooling

7.6. Convolutional Neural Networks (LeNet)

8.2. Networks Using Blocks (VGG)

8.6. Residual Networks (ResNet)

- 8.6.1 - 8.6.4

7.1 From Fully Connected Layers to Convolutions

이미지를 다룰 때 FC Layer는 Parameter의 수가 너무 많아진다.
Convolution을 사용하면 훨씬 적은 Parameter로 이미지를 처리할 수 있다.

또, 이미지를 처리할 때 Translation invariant라는 조건이 필요하다.
예를 들어 사과 사진을 상하좌우로 이동시켜도 똑같은 사과로 인식해야 한다.
수학적으로는 Input X의 Shifting이 Hidden H의 Shifting으로 대응되어야 한다.
이 때 Convolution은 이런 성질을 만족할 수 있다.

다음으로는 Locality라는 성질이 있다.
Input image X의 특정 위치의 정보는, Hidden H의 그 위치에 대응된다.
즉 이미지 전체를 볼 필요가 없고 부분부분 인식하여 Parameter 수를 크게 줄인다.

7.2 Convolutions for Images

CNN을 실제 코드로 작성해 본다.
Image X와 Kernel K의 연산을 corr2d(X,K)로 정의한다.
함수 이름이 corr인 이유는 사실 엄밀히는 Convolution이 아니기 때문이다.
순서를 뒤집어서 곱하지 않기 때문에 Cross Correlation 연산에 해당한다.

```
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


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
# Image
```

```
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]])
# Kernel detecting vertical line


Y = corr2d(X, K)
Y
# Convolution result
```

```
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.]])
```

```
# 2D convolutional layer with 1 output channel
conv2d = nn.LazyConv2d(1, kernel_size=(1, 2), bias=False)

# 4D Input
```

```
# output = (example, channel, height, width)
# Channel, Batch size = 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 10.604
epoch 4, loss 3.047
epoch 6, loss 1.031
epoch 8, loss 0.386
epoch 10, loss 0.152
```

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

```
tensor([[ 0.9483, -1.0276]])
```

7.3 Padding and Stride

앞에서는 Padding과 Stride등의 Hyperparameter들을 설정하지 않았다.
Padding은 Convolution 후 테두리 부분이 사라지고 사이즈가 줄어드는데,
이때 테두리 부분을 어떻게 처리할 지에 관련된 내용이다.
Stride는 Kernel이 움직일 때 몇 칸씩 움직이는지를 나타낸다.

```
import torch
from torch import nn


# helper function to calculate convolutions
def comp_conv2d(conv2d, X):
    X = X.reshape((1, 1) + X.shape) # (batch, channel) = (1, 1)
    Y = conv2d(X)
    return Y.reshape(Y.shape[2:]) # Drop examples and channels

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
# Size is same because of padding
```

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

```
conv2d = nn.LazyConv2d(1, kernel_size=3, padding=1, stride=2)
comp_conv2d(conv2d, X).shape
# Stride increased => Output size decreased
```

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

```
conv2d = nn.LazyConv2d(1, kernel_size=(3, 5), padding=(0, 1), stride=(3, 4))
comp_conv2d(conv2d, X).shape
# Stride increased => Output size decreased
```

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

7.4 Multiple Input and Multiple Output Channels

현실의 Image는 3개의 Channel을 가지고 있다. (RGB)
그리고 필터 역시 여러개를 사용하여 여러 Output을 낼 수 있다.
예를 들어 7.2에서 했던 것처럼 수직선을 인식하는 필터만이 있다고 하자.
그렇다면 가로, 대각선 등 여러 정보들을 놓치게 될 것이다.

그래서 실제로는 F개의 필터를 사용한다.
하나의 필터는 Input channel들과의 Convolution 값을 더하게 되고,
최종적으로는 F개 Channel의 Output이 나오게 한다.

1x1 Kernel 역시 사용되는 경우가 있다.
무의미한 연산처럼 보이지만, 이를 통해 Channel 개수를 조정할 수 있다.
또한 Activation처럼, 단순해 보이지만 모델의 복잡도를 올리는 역할도 한다.

```
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))
```

```
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) # Matrix multiplication in the FC layer
    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

Image 관련 자주 등장하는 예시는 개와 고양이 사진을 구분하는 모델이다.
그렇다면 CNN Layer를 거친 뒤 마지막에는 MLP를 통해 Classification을 한다.
이 때, Fully Connected가 가능하도록 사이즈를 줄여야 한다.
그러면서도 이미지의 전체적인 정보의 손실은 최소화해야 한다.

이 때 사용되는 것이 Pooling이다.
이미지의 각 영역에서 영역을 대표하는 하나의 값만 남긴다.
그 하나의 값은 최댓값(Max pooling), 평균(Average pooling) 등이 있다.

또, Pooling 시에도 영역 설정을 Kernel이 움직이는 것으로 생각하면,
Stride, Padding 값을 같은 원리로 설정할 수 있다.
그리고 여러 Channnel에 대한 Pooling을 적용할 경우,
Convolution 때처럼 더하는 것이 아니라 따로따로 Pooling을 진행한다.

```
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]])


print(pool2d(X, (2, 2)))
# Max pooling

print(pool2d(X, (2, 2), 'avg'))
# Average pooling
```

```
tensor([[4., 5.],
        [7., 8.]])
tensor([[2., 3.],
        [5., 6.]])
```

```
# ------ Padding, Stride example ------


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.]]]])
```

```
# ------ Multiple channel example ------


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)

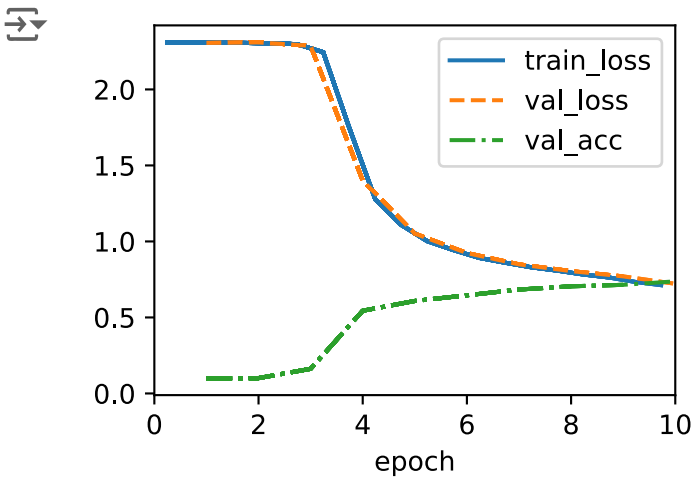
LeNet은 초창기의 CNN 모델이다.
Image에 Convolution -> Pooling -> Convolution -> Pooling -> ...
과 같이 반복되는 Layer가 있고, 끝에는 Full Connected Layer가 존재한다.
현재는 더 나은 모델들도 많지만, LeNet의 이 구조는 이후 모델들의 토대가 되었다.

```
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)

2014년에 제시된 VGG라는 모델을 사용해본다.
VGG Block은 (3x3 Conv + padding 1) 여러 번 -> 2x2 Max Pooling으로 구성된다.
이러한 Block을 여러 개 쌓은 것이 VGG 모델이다.
이 모델은 Layer가 Deep할수록 확실히 성능이 나음을 보였으며,
이 모델 이후 kernel은 3x3이 정석과 같이 자리잡았다.

```
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=8, 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)

Layer가 Deep할수록 좋지만, 너무 깊어지면 Input에서의 정보를 제대로 유지하지 못할 수 있다.
Residual block이라는 방법으로 이 문제를 해결한 ResNet에 대해 알아보자.

Input x에 대해 두 레이어 f1, f2가 있다고 하자.
원래대로라면 x -> f1(x) -> f2(f1(x))와 같이 되었을 것이다.
그러나 ResNet은 f2에도 이전 Input인 x를 전달해주는 Shortcut이 존재한다.
즉 f2 (f1(x) + x)와 같이 되어, 정보가 손실되지 않고 더 잘 전달된다.

이 방법을 도입하고 Layer의 수가 100개를 넘는 등 성능이 비약적으로 향상되었다.

```
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
# Normal case example
```

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

```
blk = Residual(6, use_1x1conv=True, strides=2)
blk(X).shape
# 1x1 conv example => Resize channels
```

```
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])
```