COSE474-2024F: Deep Learning HW2

7.1 From Fully Connected Layers to Convolutions

7.1.1 Invariance

The position of the patch of the image that is currently analyzed should not be treated differently depending on where the patch is located in the image. (translation invariance) Depending on how deep a layer is in the network it should be able to capture longer-range features.

7.1.4 Channels

One pixel in an image has 3 channels (R - Red, G - Green, B - Blue) Therefore it is also a good idea to incorporate multiple channels into the hidden representations of our input. These channels are also called feature maps.

7.2 Convolutions for Images

7.2.1 The Cross-Correlation Operation

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

```
/Users/matteo/PycharmProjects/pythonProject/.venv/lib/python3.9/site-packa
ges/urllib3/__init__.py:35: NotOpenSSLWarning: urllib3 v2 only supports Op
enSSL 1.1.1+, currently the 'ssl' module is compiled with 'LibreSSL 2.8.
3'. See: https://github.com/urllib3/urllib3/issues/3020
  warnings.warn(
```

In [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
```

In [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)
```

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

7.2.2 Convolutional Layers

In [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
```

7.2.3 Object Edge Detection in Images

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

Out[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.]])
```

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

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

Out[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.]])
```

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

Out[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.]])
```

7.2.4 Learning a Kernel

In [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 10.896
epoch 4, loss 2.404
epoch 6, loss 0.639
epoch 8, loss 0.204
epoch 10, loss 0.074
```

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

Out[10]:
```
tensor([[ 1.0116, -0.9581]])
```

7.2.5 Cross-Correlation and Convolution

In order to receive the output of a strict convolution operation from a two-dimensional convolution layer instead of a cross-correlation, the kernel tensor needs to be flipped both horizontally and vertically before performing the cross-correlation operation with the input tensor. The output of the convolution layers does not change depending on if a convolution or cross-correlation operation was performed.

7.2.6 Feature Map and Receptive Field

Feature Map: The output from a convolutional layer. Learned representations of the dimensions to the next layer.

Receptive Field: For any element x in a CNN, the receptive field refers to any other element in the CNN that has an influence on the calculation of x during forward propagation.

7.3 Padding and Stride

In [11]:
```
import torch
```

```
from torch import nn
```

7.3.1 Padding

One problem when applying convolutional layers, is that information of pixels at the edges often get lost. This can get mitigated by using a smaller kernel tensor, but after applying multiple convolutional layers, this effect can stack up. To solve this problem padding is added to the input tensor, by adding pixels with value 0 to the edges of the input (padding). The amount of padding depends on the size of the kernel, with the height of the kernel - 1 being added on top and bottom and the width of the kernel - 1 being added to either side.

In [12]:
```
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
```

Out[12]:
```
torch.Size([8, 8])
```

In [13]:
```
conv2d = nn.LazyConv2d(1, kernel_size=(5, 3), padding=(2, 1))
comp_conv2d(conv2d, X).shape
```

Out[13]:
```
torch.Size([8, 8])
```

7.3.2 Stride

To compute the cross-correlation, the convolution window is slided across the input from the top left of the input. The stride defines how many rows and columns the window gets slided. This can be useful to optimize performance or for down-scaling the output of the convolution layer.

In [14]:
```
conv2d = nn.LazyConv2d(1, kernel_size=3, padding=1, stride=2)
comp_conv2d(conv2d, X).shape
```

Out[14]:
```
torch.Size([4, 4])
```

In [15]:
```
conv2d = nn.LazyConv2d(1, kernel_size=(3, 5), padding=(0, 1), stride=(3,
comp_conv2d(conv2d, X).shape
```

Out[15]:
```
torch.Size([2, 2])
```

7.4 Multiple Input and Multiple Output Channels

In [16]:
```
import torch
from d2l import torch as d2l
```

7.4.1 Multiple Input Channels

If our input has multiple channels, the input tensor will be a third-degree tensor. Therefore, our kernel needs to have the same amount of channels as the input tensor and also be a third-degree tensor. To compute the output of the convolution layer, the cross-correlation is calculated for every channel of the input with the corresponding channel of the kernel. The results are then added together, yielding a second-degree tensor as the output of the convolution channel.

In [17]:
```
def corr2d_multi_in(X, K):
    return sum(d2l.corr2d(x, k) for x, k in zip(X, K))
```

In [18]:
```
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)
```

Out[18]:
```
tensor([[ 56.,  72.],
        [104., 120.]])
```

7.4.2 Multiple Output Channels

Having multiple channels in every layer is very important. If we want to have c_o output channels from c_i input channels, we need to construct c_o third-degree kernels with c_i channels each, creating a fourth-degree tensor as a kernel. The output is then a third-degree tensor with c_o channels.

In [19]:
```
def corr2d_multi_in_out(X, K):
    return torch.stack([corr2d_multi_in(X, k) for k in K], 0)
```

In [20]:
```
K = torch.stack((K, K + 1, K + 2), 0)
K.shape
```

Out[20]:
```
torch.Size([3, 2, 2, 2])
```

In [21]:
```
corr2d_multi_in_out(X, K)
```

Out[21]:
```
tensor([[[ 56.,  72.],
         [104., 120.]],

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

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

7.4.3 1×1 Convolutional Layer

While a 1×1 convolution window is not able to pick up on patterns across multiple adjacent pixels, it is useful when trying to find patterns between the different channels. An input with c_i channels is transformed into an output with c_o channels while preserving its size.

In [22]:
```
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))
```

In [23]:
```
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

In [24]:
```
import torch
from torch import nn
from d2l import torch as d2l
```

7.5.1 Maximum Pooling and Average Pooling

Similar to Convolution, in Pooling a window is slid over the input. However, the Pooling layer does not contain any parameters and calculates a deterministic output, often the maximum (Maximum Pooling) or average (Average Pooling) in the pooling window, to reduce the size of the output.

In [25]:
```
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
```

In [26]:
```
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))
```

Out[26]:
```
tensor([[4., 5.],
        [7., 8.]])
```

In [27]:
```
pool2d(X, (2, 2), 'avg')
```

Out[27]:
```
tensor([[2., 3.],
        [5., 6.]])
```

7.5.2 Padding and Stride

As with convolution layers, the padding of the input and the stride of the window can be adjusted in Pooling layers. For Pooling the function `nn.MaxPool2d()` can be used to perform Max-Pooling.

In [28]:
```
X = torch.arange(16, dtype=torch.float32).reshape((1, 1, 4, 4))
X
```

Out[28]:
```
tensor([[[[ 0.,  1.,  2.,  3.],
          [ 4.,  5.,  6.,  7.],
          [ 8.,  9., 10., 11.],
          [12., 13., 14., 15.]]]])
```

In [29]:
```
pool2d = nn.MaxPool2d(3)
pool2d(X)
```

Out[29]:
```
tensor([[[[10.]]]])
```

In [30]:
```
pool2d = nn.MaxPool2d(3, padding=1, stride=2)
pool2d(X)
```

Out[30]:
```
tensor([[[[ 5.,  7.],
          [13., 15.]]]])
```

In [31]:
```
pool2d = nn.MaxPool2d((2, 3), stride=(2, 3), padding=(0, 1))
pool2d(X)
```

Out[31]:
```
tensor([[[[ 5.,  7.],
          [13., 15.]]]])
```

7.5.3 Multiple Channels

When pooling over multiple channels, the output of each channel is not aggregated with the other channels. This means that the output has just as many channels as the input tensor.

In [32]:
```
X = torch.cat((X, X + 1), 1)
X
```

Out[32]:
```
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.]]]])
```

In [33]:
```
pool2d = nn.MaxPool2d(3, padding=1, stride=2)
pool2d(X)
```

Out[33]:
```
tensor([[[[ 5.,  7.],
          [13., 15.]],

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

7.6 Convolutional Neural Networks (LeNet)

In [34]:
```
import torch
from torch import nn
from d2l import torch as d2l
```

7.6.1 LeNet

The LeNet consists of two parts:

1. Convolutional encoder consisting of two convolutional layers
2. Dense Block of three fully connected layers

In [35]:
```
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))
```

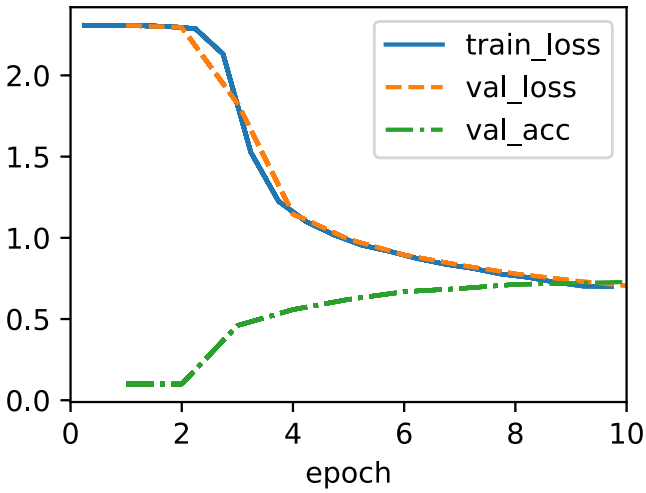
In [36]:
```
@d2l.add_to_class(d2l.Classifier)  #@save
def layer_summary(self, X_shape):
    X = torch.randn(*X_shape)
    for layer in self.net:
        X = layer(X)
        print(layer.__class__.__name__, 'output shape:\t', X.shape)

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])
```

7.6.2 Training

After implementing the model, we now inspect its performance on the Fashion-MNIST

In [37]:
```
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)

In [43]:
```
import torch
from torch import nn
from d2l import torch as d2l
```

8.2.1 VGG Blocks

A VGG Block consists of a sequence of convolutions with 3×3 kernels and padding of 1, followed by a 2×2 Max-Pooling Layer with a stride of 2, shrinking the size of the input by half after each block.

In [44]:
```
def vgg_block(num_convs, out_channels):
    layers = []
    for _ in range(num_convs):
        layers.append(nn.LazyConv2d(out_channels, kernel_size=3, padding=
        layers.append(nn.ReLU())
    layers.append(nn.MaxPool2d(kernel_size=2,stride=2))
    return nn.Sequential(*layers)
```

8.2.2 VGG Network

The VGG Network can be split into two parts:

1. Several VGG blocks
2. Fully connected layers identical to AlexNet

In [45]:
```
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)
```

In [46]:
```
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])
```

8.2.3 Training

The VGG-Network is now trained on Fashion-MNIST.

In [ ]:
```
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

In [51]:
```
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l
```

8.6.1 Function Classes

Given F, the class of functions that a given network architecture can reach, we want to find f_F^*, the closest representation of the "true" function f^* that is contained in F. Even though it might make sense to increase the size of the function class as much

as possible, this can sometimes have a negative effect on the quality of the best possible function in the class. Unless $F \subseteq F'$, we cannot guarantee, that $f_{F'}^*$ is better than f_F^*.

8.6.2 Residual Blocks

Instead of having to learn the underlying mapping $f(x)$ directly as a regular block would, a residual block has to learn the residual mapping $g(x) = f(x) - x$. If the underlying mapping is the identity function $f(x) = x$, using a residual block makes this a lot easier. With residual blocks, inputs can forward propagate faster through the residual connections across layers.

In [52]:
```
class Residual(nn.Module):  #@save
    """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)
```

In [53]:
```
blk = Residual(3)
X = torch.randn(4, 3, 6, 6)
blk(X).shape
```

Out[53]:
```
torch.Size([4, 3, 6, 6])
```

In [54]:
```
blk = Residual(3)
X = torch.randn(4, 3, 6, 6)
blk(X).shape
```

Out[54]:
```
torch.Size([4, 3, 6, 6])
```

8.6.3 ResNet Model

The Following Code describes the Implementation of the ResNet.

In [55]:
```
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))
```

In [56]:
```
@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)
```

In [57]:
```
@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)
```

In [58]:
```
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])
```

8.6.4 Training

The Model is now trained on the Fashion-MNIST.

In [ ]:
```
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)
```