

코딩을 시작하거나 AI로 코드를 생성하세요.

## ✓ COSE474-2024F: Deep Learning HW2

### 7.1 From Fully Connected Layers to Convolutions

**Constraining the MLP** consider an MLP with two-dimensional images  $\mathbf{X}$  as inputs

immediate hidden representations  $\mathbf{H}$  similarly represented as matrices (they are two-dimensional tensors in code), where both  $\mathbf{X}$  and  $\mathbf{H}$  have the same shape

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.

switch from using weight matrices to representing our parameters as fourth-order weight tensors  $\mathbf{W}$ . Suppose that  $\mathbf{U}$  contains biases, we could formally express the fully connected layer as

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

#### Translation Invariance

simplify the definition for  $\mathbf{H}$ :

$$[\mathbf{H}]_{i,j} = u + \sum_a \sum_b [\mathbf{V}]_{a,b} [\mathbf{X}]_{i+a,j+b}.$$

**Locality** invoke the second principle: locality.

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

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

#### Convolutions

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

That is, we measure the overlap between  $f$  and  $g$  when one function is "flipped" and shifted by  $\mathbf{x}$ . Whenever we have discrete objects, the integral turns into a sum. For instance, for vectors from the set of square-summable infinite-dimensional vectors with index running over  $\mathbb{Z}$  we obtain the following definition:

$$(f * g)(i) = \sum_a f(a)g(i - a).$$

For two-dimensional tensors, we have a corresponding sum with indices  $(a, b)$  for  $f$  and  $(i - a, j - b)$  for  $g$ , respectively:

$$(f * g)(i, j) = \sum_a \sum_b f(a, b)g(i - a, j - b).$$

#### Channels

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

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

where  $d$  indexes the output channels in the hidden representations  $\mathbf{H}$ . The subsequent convolutional layer will go on to take a third-order tensor,  $\mathbf{H}$ , as input.

## ✓ 7.2 Convolutions for Images

```
!pip install d2l
```

 숨겨진 출력 표시

```
import torch
from torch import nn
```

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

```
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
```

```
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 7.646
epoch 4, loss 1.286
epoch 6, loss 0.217
epoch 8, loss 0.037
epoch 10, loss 0.006
```

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

```
→ tensor([[ 0.9835, -0.9878]])
```

## ✓ 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])
```

## ✓ 7.3 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]]]),
                  [[1.0, 2.0], [3.0, 4.0]])]
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)

```
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)

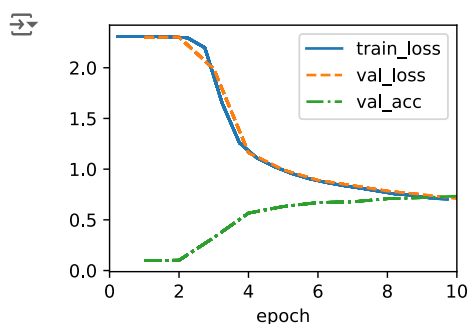
model = LeNet()
model.layer_summary((1, 1, 28, 28))
```

```

1) 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)

```
!pip install torch torchvision torchaudio --extra-index-url https://download.pytorch.org/whl/cu117
```


 숨겨진 출력 표시

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

```

## Discussions

### 7.2

#### Receptive fields

Receptive fields derive their name from neurophysiology. A series of experiments on a range of animals using different stimuli (Hubel and Wiesel, 1959, Hubel and Wiesel, 1962, Hubel and Wiesel, 1968) explored the response of what is called the visual cortex on said stimuli. By and large they found that lower levels respond to edges and related shapes. Later on, Field (1987) illustrated this effect on natural images with, what can only be called, convolutional kernels.

As it turns out, this relation even holds for the features computed by deeper layers of networks trained on image classification tasks, as demonstrated in, for example, Kuzovkin et al. (2018).

## 7.3

### Why padding with zero

- it is trivial to accomplish.
- operators can be engineered to take advantage of this padding implicitly without the need to allocate additional memory.
- it allows CNNs to encode implicit position information within an image, simply by learning where the “whitespace” is. There are many alternatives to zero-padding. Alsallakh et al. (2020) provided an extensive overview of those (albeit without a clear case for when to use nonzero paddings unless artifacts occur).

## 7.6

### Difference of LeNet compared to MLPs

- greater amounts of computation enabled significantly more complex architectures.
- relative ease with which we were able to implement LeNet. What used to be an engineering challenge worth months of C++ and assembly code, engineering to improve SN, an early Lisp-based deep learning tool (Bottou and Le Cun, 1988), and finally experimentation with models can now be accomplished in minutes. It is this incredible productivity boost that has democratized deep learning model development tremendously.

## 8.2

### VGG

More recently ParNet (Goyal et al., 2021) demonstrated that VGG is possible to achieve competitive performance using a much more shallow architecture through a large number of parallel computations.

## ✓ Exercises

### 7.1.3.

#### Reasons Translation Invariance Might Be Problematic

1. Loss of Spatial Information: In tasks where object position is crucial, such as in context-heavy scenes, translation invariance can obscure important details.
2. Varying Importance of Features: Some features are more significant based on their location, like traffic signs relative to the road.
3. Complex Patterns: Tasks like handwriting recognition require understanding spatial relationships, which translation invariance might overlook.

In handwriting recognition, the arrangement of letters affects meaning. For instance, "cat" and "act" consist of the same letters but have different meanings depending on their position. A translation-invariant model might treat them equally, missing contextual clues and spacing differences that are vital for accurate interpretation. Thus, translation invariance can hinder performance in scenarios where spatial relationships are essential.

### 7.3.2.

a stride of 2 in audio signals means that during processing, the algorithm skips every other sample, effectively reducing the data size and computational complexity while still capturing key features of the audio signal.

### 8.2.1

#### 1. AlexNet vs VGG

- AlexNet has approximately 60 million parameters. The architecture consists of five convolutional layers followed by three fully connected layers.
- VGG models, particularly VGG16, have about 138 million parameters, while VGG19 has around 143 million parameters. The increase in parameters comes from using more convolutional layers and smaller 3x3 filters.



## 2. convolutional layers vs fully connected layers

### (1) convolutional layers:

- AlexNet: The total FLOPs in the convolutional layers of AlexNet is estimated to be around 724 million for a single forward pass on a standard input image size (227x227).
- VGG: VGG models (like VGG16) have a higher FLOP count due to deeper architectures, with approximately 15.5 billion FLOPs for a single forward pass using the same input size.

### (2) fully connected layers:

- AlexNet: The fully connected layers add about 2.5 billion FLOPs.
- VGG: VGG's fully connected layers contribute around 12 billion FLOPs, significantly increasing the computational cost compared to AlexNet.