COSE474 2024 Fall : Deep Learning HW2

0.1 Installation

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

7.1. From Fully Connected Layers to Convolutions

- Sometimes, we truly lack the knowledge to be able to guide the construction of fancier architectures. In these cases, an MLP may be the best that we can do. However, for high-dimensional perceptual data, such structureless networks can grow unwieldy.
- while we might be able to get away with one hundred thousand pixels, our hidden layer of size 1000 grossly underestimates the number of hidden units that it takes to learn good representations of images, so a practical system will still require billions of parameters. Moreover, learning a classifier by fitting so many parameters might require collecting an enormous dataset.
- Convolutional neural networks (CNNs) are one creative way that machine learning has embraced for exploiting some of the known structure in natural images.

7.1.1. Invariance

- What Waldo looks like does not depend upon where Waldo is located. We could sweep the image with a Waldo detector that could assign a score to each patch, indicating the likelihood that the patch contains Waldo. In fact, many object detection and segmentation algorithms are based on this approach (Long et al., 2015). CNNs systematize this idea of spatial invariance, exploiting it to learn useful representations with fewer parameters.

1. In the earliest layers, our network should respond similarly to the same patch, regardless of where it appears in the image. This principle is called translation invariance (or translation equivariance).
2. The earliest layers of the network should focus on local regions, without regard for the contents of the image in distant regions. This is the locality principle. Eventually, these local representations can be aggregated to make predictions at the whole image level.
3. As we proceed, deeper layers should be able to capture longer-range features of the image, in a way similar to higher level vision in nature.

7.1.2. Constraining the MLP

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

- For any given location (i, j) in the hidden representation $[\mathbf{H}]_{i,j}$, we compute its value by summing over pixels in x, centered around (i, j) and weighted by $[\mathsf{V}]_{i,j,a,b}$. Before we carry on, let's consider the total number of parameters required for a single layer in this parametrization: a 1000 $\times$ 1000 image (1 megapixel) is mapped to a 1000 $\times$ 1000 hidden representation. This requires 10^{12} parameters, far beyond what computers currently can handle.

7.1.2.1. Translation Invariance

- This implies that a shift in the input X should simply lead to a shift in the hidden representation H.

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

- We are effectively weighting pixels at (i+a, j+b) in the vicinity of location (i, j) with coefficients $[\mathbf{V}]_{a,b}$ to obtain the value $[\mathbf{H}]_{i,j}$. Note that $[\mathbf{V}]_{a,b}$ needs many fewer coefficients than $[\mathsf{V}]_{i,j,a,b}$ since it no longer depends on the location within the image.

7.1.2.2. Locality

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

- 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.
- we now typically need just a few hundred, without altering the dimensionality of either the inputs or the hidden representations. The price paid for this drastic reduction in parameters is that our features are now translation invariant and that our layer can only incorporate local information, when determining the value of each hidden activation.
- This dramatic reduction in parameters brings us to our last desideratum, namely that deeper layers should represent larger and more complex aspects of an image. This can be achieved by interleaving nonlinearities and convolutional layers repeatedly.

7.1.3. Convolutions

- *Convolution* between two functions, say $f, g : \mathbb{R}^d \to \mathbb{R}$ is defined as

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

- the integral turns into a sum

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

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

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

7.1.4. Channels

- 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:

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

where d indexes the output channels in the hidden representations H.

7.2. Convolutions for Images

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

7.2.1. The Cross-Correlation Operation

- 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

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

```
def corr2d(X, K):  #@save
    """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
```

"@save" 주석은 허용되지 않습니다. 허용되는 값은 다음과 같습니다. [@param, @title, @markdown]

```
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.]])
```

7.2.2. Convolutional Layers

- 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.

```
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

- Let's take a moment to parse a simple application of a convolutional layer: detecting the edge of an object in an image by finding the location of the pixel change.
- "image" of 6 X 8 pixels. The middle four columns are black (0) and the rest are white (1).

```
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.]])
```

- Next, we construct a kernel K with a height of 1 and a width of 2.
- When we perform the cross-correlation operation with the input, if the horizontally adjacent elements are the same, the output is 0. Otherwise, the output is nonzero.

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

- we detect 1 for the edge from white to black and -1 for the edge from black to white. All other outputs take value 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.]])
```

- Apply the kernel to the transposed image. It vanishes.

```
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.]])
```

7.2.4. Learning a Kernel

- Now let's see whether we can learn the kernel that generated Y from X by looking at the input–output pairs only.
- Construct a convolutional layer and initialize its kernel as a random tensor.
- We will use the squared error to compare Y with the output of the convolutional layer

```
# 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 8.289
epoch 4, loss 1.552
epoch 6, loss 0.326
epoch 8, loss 0.082
epoch 10, loss 0.025
```

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

```
tensor([[ 0.9998, -0.9715]])
```

7.2.5. Cross-Correlation and Convolution

- It is noteworthy that since kernels are learned from data in deep learning, the outputs of convolutional layers remain unaffected no matter such layers perform either the strict convolution operations or the cross-correlation operations.
- In keeping with standard terminology in deep learning literature, we will continue to refer to the cross-correlation operation as a convolution even though, strictly-speaking, it is slightly different. Furthermore, we use the term element to refer to an entry (or component) of any tensor representing a layer representation or a convolution kernel.

7.2.6. Feature Map and Receptive Field

- The convolutional layer output is sometimes called a feature map, as it can be regarded as the learned representations (features) in the spatial dimensions (e.g., width and height) to the subsequent layer.
- 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). Suffice it to say, convolutions have proven to be an incredibly powerful tool for computer vision, both in biology and in code. As such, it is not surprising (in hindsight) that they heralded the recent success in deep learning.

7.3. Padding and Stride

- we will explore a number of techniques, including padding and strided convolutions, that offer more control over the size of the output.

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

7.3.1. Padding

- One tricky issue when applying convolutional layers is that we tend to lose pixels on the perimeter of our image.
- 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.
- 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
$$(n_\mathrm{h} - k_\mathrm{h} + p_\mathrm{h} + 1) \times (n_\mathrm{w} - k_\mathrm{w} + p_\mathrm{w} + 1).$$
- CNNs commonly use convolution kernels with odd height and width values, such as 1, 3, 5, or 7.

```
# 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])
```

7.3.2. Stride

- Sometimes, either for computational efficiency or because we wish to downsample, we move our window more than one element at a time, skipping the intermediate locations. This is particularly useful if the convolution kernel is large since it captures a large area of the underlying image.
- In general, when the stride for the height is s_h and the stride for the width is s_w, the output shape is
$$\lfloor (n_\mathrm{h} - k_\mathrm{h} + p_\mathrm{h} + s_\mathrm{h})/s_\mathrm{h} \rfloor \times \lfloor (n_\mathrm{w} - k_\mathrm{w} + p_\mathrm{w} + s_\mathrm{w})/s_\mathrm{w} \rfloor.$$

```
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])
```

7.3.3. Summary and Discussion

- So far all padding that we discussed simply extended images with zeros. This has significant computational benefit since it is trivial to accomplish. Moreover, operators can be engineered to take advantage of this padding implicitly without the need to allocate additional memory. At the same time, it allows CNNs to encode implicit position information within an image, simply by learning where the "whitespace" is.

7.4. Multiple Input and Multiple Output Channels

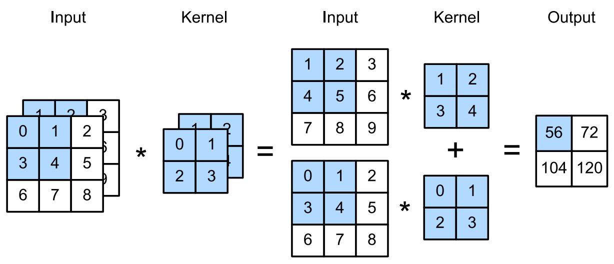
- When we add channels into the mix, our inputs and hidden representations both become three-dimensional tensors. For example, each RGB input image has shape $3 \times h \times w$.

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

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.
- Below image provides an example of a two-dimensional cross-correlation with two input channels. The shaded portions are the first output element as well as the input and kernel tensor elements used for the output computation:

$$(1 \times 1 + 2 \times 2 + 4 \times 3 + 5 \times 4) + (0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3) = 56$$



```
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.]])
```

7.4.2. Multiple Output Channels

- In the most popular neural network architectures, we actually increase the channel dimension as we go deeper in the neural network, typically downsampling to trade off spatial resolution for greater channel depth.
- Intuitively, you could think of each channel as responding to a different set of features. The reality is a bit more complicated than this.

```
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)
```

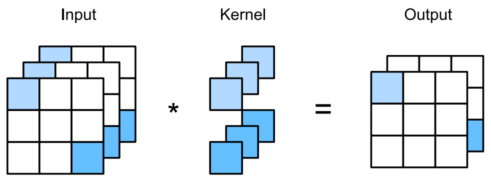
```
K = torch.stack((K, K + 1, K + 2), 0)
K.shape
```

```
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.]]])
```

7.4.3. 1×1 Convolutional Layer



- Upper image shows the cross-correlation computation using the 1×1 convolution kernel with 3 input channels and 2 output channels. Note that the inputs and outputs have the same height and width. Each element in the output is derived from a linear combination of elements at the same position in the input image. You could think of the 1×1 convolutional layer as constituting a fully connected layer applied at every single pixel location to transform the c_i corresponding input values into c_o output values. Because this is still a convolutional layer, the weights are tied across pixel location. Thus the 1×1 convolutional layer requires $c_o \times c_i$ weights (plus the bias). Also note that convolutional layers are typically followed by nonlinearities. This ensures that 1×1 convolutions cannot simply be folded into other convolutions.

```
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
```

7.4.4. Discussion

- Channels allow us to combine the best of both worlds: MLPs that allow for significant nonlinearities and convolutions that allow for localized analysis of features. In particular, channels allow the CNN to reason with multiple features, such as edge and shape detectors at the same time. They also offer a practical trade-off between the drastic parameter reduction arising from translation invariance and locality, and the need for expressive and diverse models in computer vision.
- Note, though, that this flexibility comes at a price.

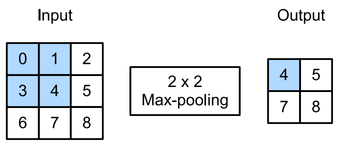
7.5. Pooling

- This section introduces pooling layers, which serve the dual purposes of mitigating the sensitivity of convolutional layers to location and of spatially downsampling representations.

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

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.
- However, unlike the cross-correlation computation of the inputs and kernels in the convolutional layer, the pooling layer contains no parameters . 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 and average pooling, respectively.



$$\max(0,1,3,4)=4,$$
$$\max(1,2,4,5)=5,$$
$$\max(3,4,6,7)=7,$$
$$\max(4,5,7,8)=8.$$

```
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.]])
```

7.5.2. Padding and Stride

- As with convolutional layers, pooling layers change the output shape. And as before, we can adjust the operation to achieve a desired output shape by padding the input and adjusting the stride.

```
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.]]]])
```

7.5.3. Multiple Channels

- When processing multi-channel input data, the pooling layer pools each input channel separately, rather than summing the inputs up over channels as in a convolutional layer. This means that the number of output channels for the pooling layer is the same as the number of input channels. Below, we will concatenate tensors X and X + 1 on the channel dimension to construct an input with two channels.

```
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.]]]])
```

7.5.4. Summary

- It does exactly what its name indicates, aggregate results over a window of values. All convolution semantics, such as strides and padding apply in the same way as they did previously.
- Note that pooling is indifferent to channels, i.e., it leaves the number of channels unchanged and it applies to each channel separately.
- Lastly, of the two popular pooling choices, max-pooling is preferable to average pooling, as it confers some degree of invariance to output.

7.6. Convolutional Neural Networks (LeNet)

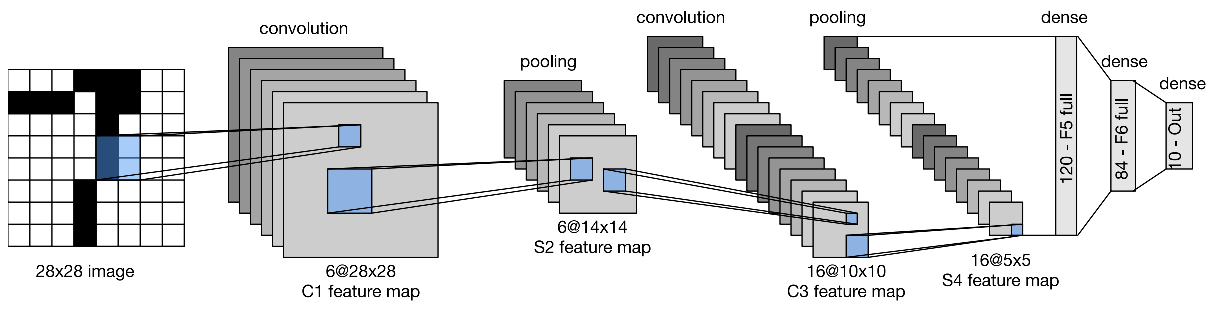
- we will introduce LeNet, among the first published CNNs to capture wide attention for its performance on computer vision tasks.
- At the time LeNet achieved outstanding results matching the performance of support vector machines, then a dominant approach in supervised learning, achieving an error rate of less than 1% per digit.
- To this day, some ATMs still run the code that Yann LeCun and his colleague Leon Bottou wrote in the 1990s!

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

7.6.1. LeNet

- At a high level, LeNet (LeNet-5) consists of two parts:

1. a convolutional encoder consisting of two convolutional layers
2. a dense block consisting of three fully connected layers.



- The basic units in each convolutional block are a convolutional layer, a sigmoid activation function, and a subsequent average pooling operation. Each convolutional layer uses a 5 X 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.
- The first convolutional layer has 6 output channels, while the second has 16. Each 2 X 2 pooling operation (stride 2) reduces dimensionality by a factor of 4 via spatial downsampling. The convolutional block emits an output with shape given by (batch size, number of channel, height, width).

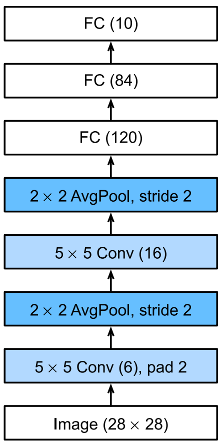
```
def init_cnn(module):  #@save
    """Initialize weights for CNNs."""
    if type(module) == nn.Linear or type(module) == nn.Conv2d:
        nn.init.xavier_uniform_(module.weight)

class LeNet(d2l.Classifier):  #@save
    """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(),
```

"@save" 주석은 허용되지 않습니다. 허용되는 값은 다음과 같습니다. [@param, @title, @markdown]

"@save" 주석은 허용되지 않습니다. 허용되는 값은 다음과 같습니다. [@param, @title, @markdown]

```
            nn.LazyLinear(84), nn.Sigmoid(),
            nn.LazyLinear(num_classes))
```

- Let's see what happens inside the network. By passing a single-channel (black and white) 28 X 28 image through the network and printing the output shape at each layer.



- Note that the height and width of the representation at each layer throughout the convolutional block is reduced (compared with the previous layer). The first convolutional layer uses two pixels of padding to compensate for the reduction in height and width that would otherwise result from using a 5 X 5 kernel.
- In contrast, the second convolutional layer forgoes padding, and thus the height and width are both reduced by four pixels.

```
@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))
```

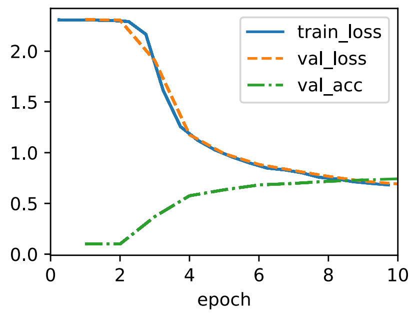
"@save" 주석은 허용되지 않습니다. 허용되는 값은 다음과 같습니다. [@param, @title, @markdown]

```
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

- Now that we have implemented the model, let's run an experiment to see how the LeNet-5 model fares on Fashion-MNIST.
- 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.

```
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)
```



7.6.3. Summary

- It is worth comparing the error rates on Fashion-MNIST achievable with LeNet-5 both to the very best possible with MLPs and those with significantly more advanced architectures such as ResNet. LeNet is much more similar to the latter than to the former.
- A second difference is the 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, and finally experimentation with models can now be accomplished in minutes.

8.2. Networks Using Blocks (VGG)

- Progress in this field mirrors that of VLSI (very large scale integration) in chip design where engineers moved from placing transistors to logical elements to logic blocks. Similarly, the design of neural network architectures has grown progressively more abstract, with researchers moving from thinking in terms of individual neurons to whole layers, and now to blocks, repeating patterns of layers.

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

8.2.1. VGG Blocks

- The basic building block of CNNs

1. A convolutional layer with padding to maintain the resolution
2. a nonlinearity such as a ReLU
3. 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.

- 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.
- Stacking 3 X 3 convolutions has become a gold standard in later deep networks.
- Back to VGG: a VGG block consists of a sequence of convolutions with 3 X 3 kernels with padding of 1 (keeping height and width) followed by a 2 X 2 max-pooling layer with stride of 2 (halving height and width after each block).

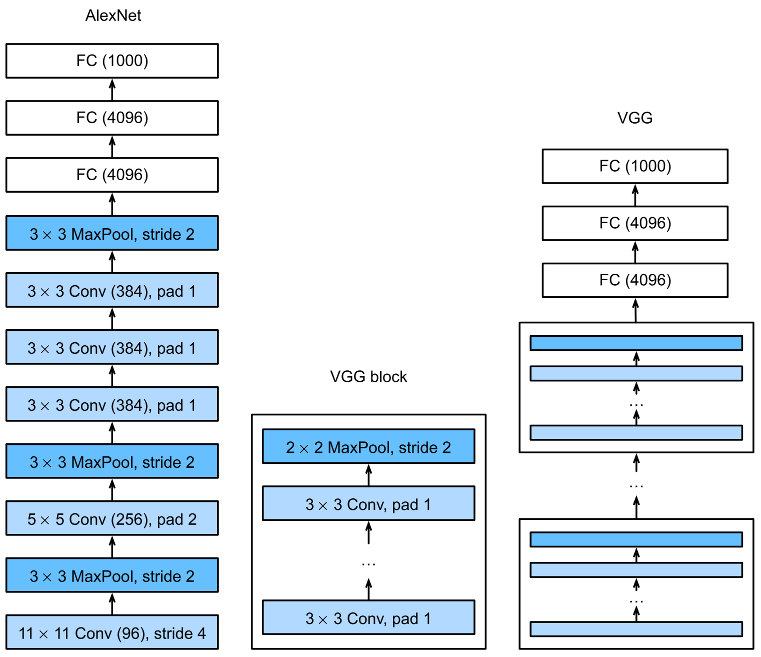
```
def vgg_block(num_convs, out_channels):
    layers = []
    for _ in range(num_convs):
        layers.append(nn.LazyConv2d(out_channels, kernel_size=3, padding=1))
        layers.append(nn.ReLU())
    layers.append(nn.MaxPool2d(kernel_size=2,stride=2))
    return nn.Sequential(*layers)
```

8.2.2. VGG Network

- The VGG Network can be partitioned into two parts

1. The first consisting mostly of convolutional and pooling layers
2. 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.



```python
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)
```

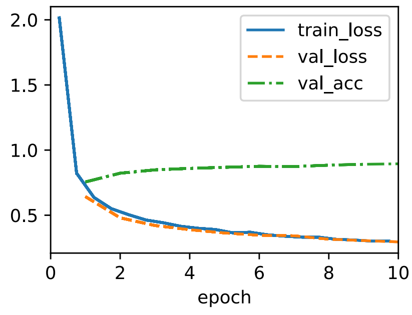
```python
VGG(arch=((1, 64), (1, 128), (2, 256), (2, 512), (2, 512))).layer_summary(
    (1, 1, 224, 224))
```

```
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

- Since VGG-11 is computationally more demanding than AlexNet we construct a network with a smaller number of channels. This is more than sufficient for training on Fashion-MNIST. The model training process is similar to that of AlexNet in Section 8.1.

```python
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)
```



8.2.4. Summary

- While AlexNet introduced many of the components of what make deep learning effective at scale, it is VGG that arguably introduced key properties such as blocks of multiple convolutions and a preference for deep and narrow networks. - It is also the first network that is actually an entire family of similarly parametrized models, giving the practitioner ample trade-off between complexity and speed.

8.6. Residual Networks (ResNet) and ResNeXt

- As we design ever deeper networks it becomes imperative to understand how adding layers can increase the complexity and expressiveness of the network.
- Even more important is the ability to design networks where adding layers makes networks strictly more expressive rather than just different.

```python
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l
```

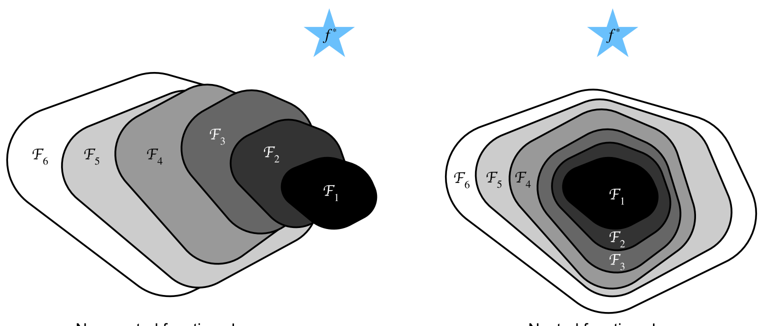
8.6.1. Function Classes

- The class of functions, denoted as $\mathcal{F}$, represents the functions a specific network architecture can reach with certain hyperparameters. Ideally, we aim to find the "truth" function f^*, but it is usually not part of $\mathcal{F}$. Instead, we search for $f^*_{\mathcal{F}}$, the best approximation of f^* within

$\mathcal{F}$, by solving an optimization problem:

$$f^*_\mathcal{F} \stackrel{\textrm{def}}{=} \mathop{\mathrm{argmin}}_f L(\mathbf{X}, \mathbf{y}, f) \textrm{ subject to } f\in\mathcal{F}.$$

- We know that regularization may control complexity of $\mathcal{F}$ and achieve consistency, so a larger size of training data generally leads to better $f^*_{\mathcal{F}}$. It is reasonable to assume that a more powerful architecture $\mathcal{F}'$ should result in a better outcome. However, if $\mathcal{F} \not\subseteq \mathcal{F}'$, there is no guarantee of improvement. For non-nested function classes, a larger function class does not always move closer to the "truth" function f^*. With nested function classes, we can avoid this issue.

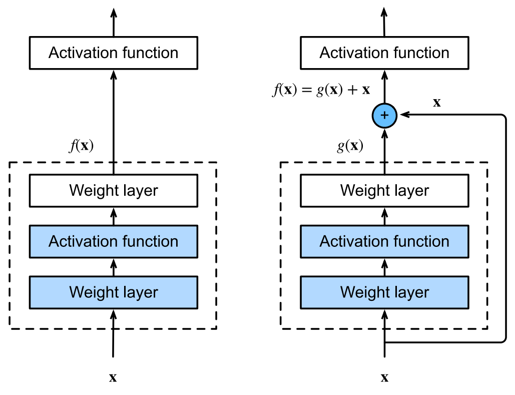


Non-nested function classes   Nested function classes

- Thus, only if larger function classes contain the smaller ones are we guaranteed that increasing them strictly increases the expressive power of the network. For deep neural networks, if we can train the newly-added layer into an identity function $f(\mathbf{x}) = \mathbf{x}$, the new model will be as effective as the original model.
- These considerations are rather profound but they led to a surprisingly simple solution, a residual block.

8.6.2. Residual Blocks

- Let's focus on a local part of a neural network, as depicted in below image.
- On the left, the portion within the dotted-line box must directly learn f(x).
- On the right, the portion within the dotted-line box needs to learn the residual mapping g(x) = f(x) - x, which is how the residual block derives its name.
- If the identity mapping f(x) = x is the desired underlying mapping, the residual mapping amounts to g(x) = 0 and it is thus easier to learn.



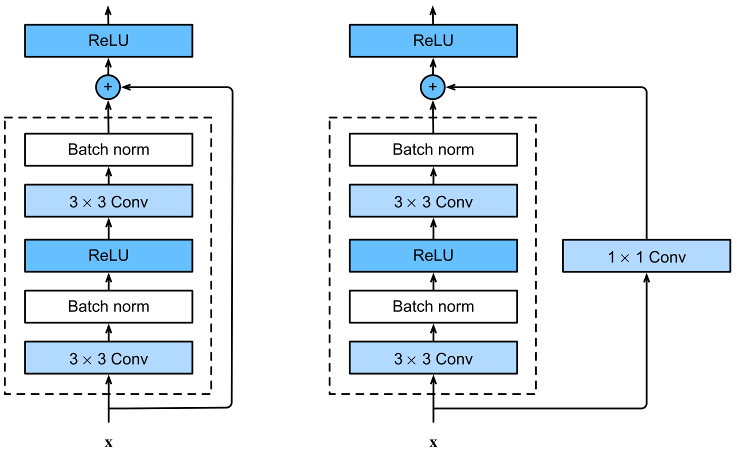
- ResNet has VGG's full 3 X 3 convolutional layer design. The residual block has two 3 X 3 convolutional layers with the same number of output channels. Each convolutional layer is followed by a batch normalization layer and a ReLU activation function. Then, we skip these two convolution operations and add the input directly before the final ReLU activation function.

```
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, paddin
                                   stride=strides)
        self.conv2 = nn.LazyConv2d(num_channels, kernel_size=3, paddin
        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)
```

"@save" 주석은 허용되지 않습니다. 허용되는 값은 다음과 같습니다. [@param, @title, @markdown]

- This code generates two types of networks

1. One where we add the input to the output before applying the ReLU nonlinearity whenever *use_1x1conv=False*
2. One where we adjust channels and resolution by means of a 1 X 1 convolution before adding.



- Now let's look at a situation where the input and output are of the same shape, where 1 X 1 convolution is not needed.
- We also have the option to halve the output height and width while increasing the number of output channels.

```
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])
```

8.6.3. ResNet Model

- The first two layers of ResNet are the same as those of the GoogLeNet we described before. 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.
- Then, we add all the modules to ResNet. Here, two residual blocks are used for each module. Lastly, just like GoogLeNet, we add a global average pooling layer, followed by the fully connected layer output.

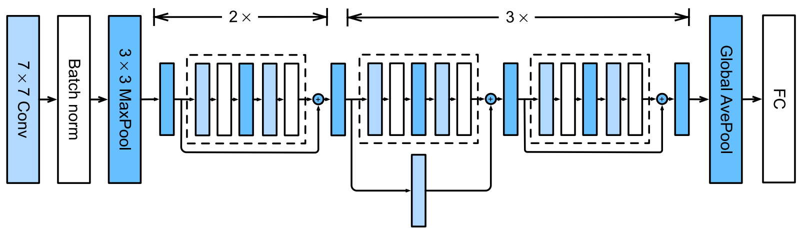
```
class ResNet(d2l.Classifier):
    def b1(self):
        return nn.Sequential(
            nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),
            nn.LazyBatchNorm2d(), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

```
@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)
```

- There are four convolutional layers in each module. Together with the first convolutional layer and the final fully connected layer, there are 18 layers in total.

- Therefore, this model is commonly known as ResNet-18. By configuring different numbers of channels and residual blocks in the module, we can create different ResNet models, such as the deeper 152-layer ResNet-152.



- The resolution decreases while the number of channels increases up until the point where a global average pooling layer aggregates all features.

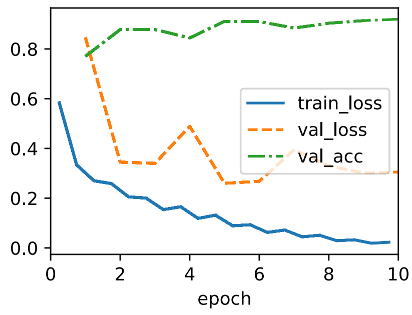
```
class ResNet18(ResNet):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__(((2, 64), (2, 128), (2, 256), (2, 512)),
                       lr, num_classes)

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 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.

```
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)
```



Discussons

7.1. From Fully Connected Layers to Convolutions

- Sometimes, we truly lack the knowledge to be able to guide the construction of fancier architectures. In these cases, an MLP may be the best that we can do. However, for high-dimensional perceptual data, such structureless networks can grow unwieldy.
- while we might be able to get away with one hundred thousand pixels, our hidden layer of size 1000 grossly underestimates the number of hidden units that it takes to learn good representations of images, so a practical system will still require billions of parameters. Moreover, learning a classifier by fitting so many parameters might require collecting an enormous dataset.
- Convolutional neural networks (CNNs) are one creative way that machine learning has embraced for exploiting some of the known structure in natural images.

7.1.1. Invariance

- What Waldo looks like does not depend upon where Waldo is located. We could sweep the image with a Waldo detector that could assign a score to each patch, indicating the likelihood that the patch contains Waldo. In fact, many object detection and segmentation algorithms are based on this approach (Long et al., 2015). CNNs systematize this idea of spatial invariance, exploiting it to learn useful representations with fewer parameters.

1. In the earliest layers, our network should respond similarly to the same patch, regardless of where it appears in the image. This principle is called translation invariance (or translation equivariance).
2. The earliest layers of the network should focus on local regions, without regard for the contents of the image in distant regions. This is the locality principle. Eventually, these local representations can be aggregated to make predictions at the whole image level.
3. As we proceed, deeper layers should be able to capture longer-range features of the image, in a way similar to higher level vision in nature.

7.1.2. Constraining the MLP

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

- For any given location (i, j) in the hidden representation $[\mathbf{H}]_{i,j}$, we compute its value by summing over pixels in x, centered around (i, j) and weighted by $[\mathsf{V}]_{i,j,a,b}$. Before we carry on, let's consider the total number of parameters required for a single layer in this parametrization: a 1000 $\times$ 1000 image (1 megapixel) is mapped to a 1000 $\times$ 1000 hidden representation. This requires 10^{12} parameters, far beyond what computers currently can handle.

7.1.2.1. Translation Invariance

- This implies that a shift in the input X should simply lead to a shift in the hidden representation H.

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

- We are effectively weighting pixels at (i+a, j+b) in the vicinity of location (i, j) with coefficients $[\mathbf{V}]_{a,b}$ to obtain the value $[\mathbf{H}]_{i,j}$. Note that $[\mathbf{V}]_{a,b}$ needs many fewer coefficients than $[\mathsf{V}]_{i,j,a,b}$ since it no longer depends on the location within the image.

7.1.2.2. Locality

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

- 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.
- we now typically need just a few hundred, without altering the dimensionality of either the inputs or the hidden representations. The price paid for this drastic reduction in parameters is that our features are now translation invariant and that our layer can only incorporate local information, when determining the value of each hidden activation.
- This dramatic reduction in parameters brings us to our last desideratum, namely that deeper layers should represent larger and more complex aspects of an image. This can be achieved by interleaving nonlinearities and convolutional layers repeatedly.

7.1.3. Convolutions

- *Convolution* between two functions, say $f, g : \mathbb{R}^d \to \mathbb{R}$ is defined as

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

- the integral turns into a sum

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

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

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

7.1.4. Channels

- 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:

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

where d indexes the output channels in the hidden representations H.

7.2. Convolutions for Images

7.2.1. The Cross-Correlation Operation

- the output size is given by the input size $n_\mathrm{h} \times n_\mathrm{w}$ minus the size of the convolution kernel $k_\mathrm{h} \times k_\mathrm{w}$ via
$$(n_\mathrm{h} - k_\mathrm{h} + 1) \times (n_\mathrm{w} - k_\mathrm{w} + 1).$$

[ ] ↳ 숨겨진 셀 2개

7.2.2. Convolutional Layers

- 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.

[ ] ↳ 숨겨진 셀 1개

7.2.3. Object Edge Detection in Images

- Let's take a moment to parse a simple application of a convolutional layer: detecting the edge of an object in an image by finding the location of the pixel change.
- "image" of 6 X 8 pixels. The middle four columns are black (0) and the rest are white (1).

```
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.]])
```

- Next, we construct a kernel K with a height of 1 and a width of 2.
- When we perform the cross-correlation operation with the input, if the horizontally adjacent elements are the same, the output is 0. Otherwise, the output is nonzero.

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

- we detect 1 for the edge from white to black and -1 for the edge from black to white. All other outputs take value 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.]])
```

- Apply the kernel to the transposed image. It vanishes.

```
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.]])
```

7.2.4. Learning a Kernel

- Now let's see whether we can learn the kernel that generated Y from X by looking at the input–output pairs only.
- Construct a convolutional layer and initialize its kernel as a random tensor.
- We will use the squared error to compare Y with the output of the convolutional layer

[ ] ↳ 숨겨진 셀 2개

7.2.5. Cross-Correlation and Convolution

- It is noteworthy that since kernels are learned from data in deep learning, the outputs of convolutional layers remain unaffected no matter such layers perform either the strict convolution operations or the cross-correlation operations.
- In keeping with standard terminology in deep learning literature, we will continue to refer to the cross-correlation operation as a convolution even though, strictly-speaking, it is slightly different. Furthermore, we use the term element to refer to an entry (or component)

of any tensor representing a layer representation or a convolution kernel.

7.2.6. Feature Map and Receptive Field

- The convolutional layer output is sometimes called a feature map, as it can be regarded as the learned representations (features) in the spatial dimensions (e.g., width and height) to the subsequent layer.
- 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). Suffice it to say, convolutions have proven to be an incredibly powerful tool for computer vision, both in biology and in code. As such, it is not surprising (in hindsight) that they heralded the recent success in deep learning.

7.3. Padding and Stride

- we will explore a number of techniques, including padding and strided convolutions, that offer more control over the size of the output.

7.3.1. Padding

- One tricky issue when applying convolutional layers is that we tend to lose pixels on the perimeter of our image.
- 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.
- 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
$$(n_\mathrm{h}-k_\mathrm{h}+p_\mathrm{h}+1)\times(n_\mathrm{w}-k_\mathrm{w}+p_\mathrm{w}+1).$$
- CNNs commonly use convolution kernels with odd height and width values, such as 1, 3, 5, or 7.

[ ] ↳ 숨겨진 셀 2개

7.3.2. Stride

- Sometimes, either for computational efficiency or because we wish to downsample, we move our window more than one element at a time, skipping the intermediate locations. This is particularly useful if the convolution kernel is large since it captures a large area of the underlying image.
- In general, when the stride for the height is s_h and the stride for the width is s_w, the output shape is
$$\lfloor(n_\mathrm{h}-k_\mathrm{h}+p_\mathrm{h}+s_\mathrm{h})/s_\mathrm{h}\rfloor\times\lfloor(n_\mathrm{w}-k_\mathrm{w}+p_\mathrm{w}+s_\mathrm{w})/s_\mathrm{w}\rfloor.$$

[ ] ↳ 숨겨진 셀 2개

7.3.3. Summary and Discussion

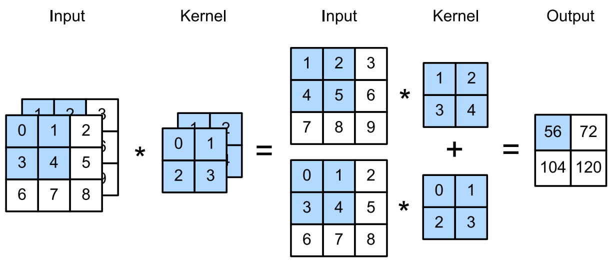
- So far all padding that we discussed simply extended images with zeros. This has significant computational benefit since it is trivial to accomplish. Moreover, operators can be engineered to take advantage of this padding implicitly without the need to allocate additional memory. At the same time, it allows CNNs to encode implicit position information within an image, simply by learning where the "whitespace" is.

7.4. Multiple Input and Multiple Output Channels

- When we add channels into the mix, our inputs and hidden representations both become three-dimensional tensors. For example, each RGB input image has shape $3\times h\times w$.

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.
- Below image provides an example of a two-dimensional cross-correlation with two input channels. The shaded portions are the first output element as well as the input and kernel tensor elements used for the output computation:
$$(1\times1+2\times2+4\times3+5\times4)+(0\times0+1\times1+3\times2+4\times3)=56$$



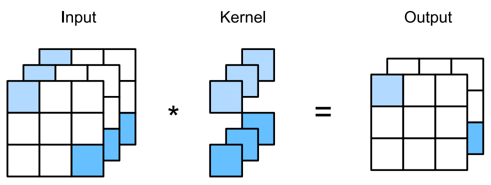
[ ] ↳ 숨겨진 셀 2개

7.4.2. Multiple Output Channels

- In the most popular neural network architectures, we actually increase the channel dimension as we go deeper in the neural network, typically downsampling to trade off spatial resolution for greater channel depth.
- Intuitively, you could think of each channel as responding to a different set of features. The reality is a bit more complicated than this.

[ ] ↳ 숨겨진 셀 3개

7.4.3. 1×1 Convolutional Layer



- Upper image shows the cross-correlation computation using the 1×1 convolution kernel with 3 input channels and 2 output channels. Note that the inputs and outputs have the same height and width. Each element in the output is derived from a linear combination of elements at the same position in the input image. You could think of the 1×1 convolutional layer as constituting a fully connected layer applied at every single pixel location to transform the c_i corresponding input values into c_o output values. Because this is still a convolutional layer, the weights are tied across pixel location. Thus the 1×1 convolutional layer requires $c_o \times c_i$ weights (plus the bias). Also note that convolutional layers are typically followed by nonlinearities. This ensures that 1×1 convolutions cannot simply be folded into other convolutions.

[ ] ↳ 숨겨진 셀 2개

7.4.4. Discussion

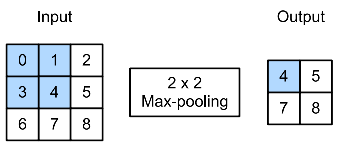
- Channels allow us to combine the best of both worlds: MLPs that allow for significant nonlinearities and convolutions that allow for localized analysis of features. In particular, channels allow the CNN to reason with multiple features, such as edge and shape detectors at the same time. They also offer a practical trade-off between the drastic parameter reduction arising from translation invariance and locality, and the need for expressive and diverse models in computer vision.
- Note, though, that this flexibility comes at a price.

7.5. Pooling

- This section introduces pooling layers, which serve the dual purposes of mitigating the sensitivity of convolutional layers to location and of spatially downsampling representations.

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.
- However, unlike the cross-correlation computation of the inputs and kernels in the convolutional layer, the pooling layer contains no parameters . 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 and average pooling, respectively.



$$\max(0,1,3,4) = 4,$$
$$\max(1,2,4,5) = 5,$$
$$\max(3,4,6,7) = 7,$$
$$\max(4,5,7,8) = 8.$$

[ ] ↳ 숨겨진 셀 3개

7.5.2. Padding and Stride

- As with convolutional layers, pooling layers change the output shape. And as before, we can adjust the operation to achieve a desired output shape by padding the input and adjusting the stride.

[ ] ↳ 숨겨진 셀 4개

7.5.3. Multiple Channels

- When processing multi-channel input data, the pooling layer pools each input channel separately, rather than summing the inputs up over channels as in a convolutional layer. This means that the number of output channels for the pooling layer is the same as the number of input channels. Below, we will concatenate tensors X and X + 1 on the channel dimension to construct an input with two channels.

[ ] ↳ 숨겨진 셀 2개

7.5.4. Summary

- It does exactly what its name indicates, aggregate results over a window of values. All convolution semantics, such as strides and padding apply in the same way as they did previously.
- Note that pooling is indifferent to channels, i.e., it leaves the number of channels unchanged and it applies to each channel separately.
- Lastly, of the two popular pooling choices, max-pooling is preferable to average pooling, as it confers some degree of invariance to output.

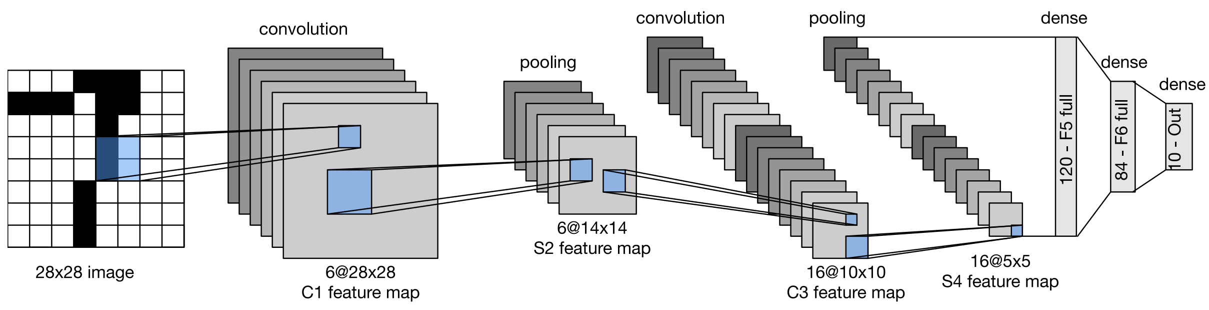
7.6. Convolutional Neural Networks (LeNet)

- we will introduce LeNet, among the first published CNNs to capture wide attention for its performance on computer vision tasks.
- At the time LeNet achieved outstanding results matching the performance of support vector machines, then a dominant approach in supervised learning, achieving an error rate of less than 1% per digit.
- To this day, some ATMs still run the code that Yann LeCun and his colleague Leon Bottou wrote in the 1990s!

7.6.1. LeNet

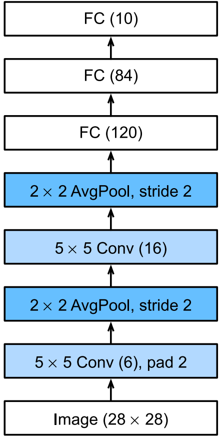
- At a high level, LeNet (LeNet-5) consists of two parts:

1. a convolutional encoder consisting of two convolutional layers
2. a dense block consisting of three fully connected layers.



- The basic units in each convolutional block are a convolutional layer, a sigmoid activation function, and a subsequent average pooling operation. Each convolutional layer uses a 5 X 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.
- The first convolutional layer has 6 output channels, while the second has 16. Each 2 X 2 pooling operation (stride 2) reduces dimensionality by a factor of 4 via spatial downsampling. The convolutional block emits an output with shape given by (batch size, number of channel, height, width).

- Let’s see what happens inside the network. By passing a single-channel (black and white) 28 X 28 image through the network and printing the output shape at each layer.

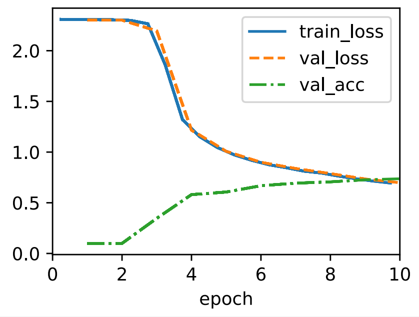


- Note that the height and width of the representation at each layer throughout the convolutional block is reduced (compared with the previous layer). The first convolutional layer uses two pixels of padding to compensate for the reduction in height and width that would otherwise result from using a 5 X 5 kernel.
- In contrast, the second convolutional layer forgoes padding, and thus the height and width are both reduced by four pixels.

7.6.2. Training

- Now that we have implemented the model, let's run an experiment to see how the LeNet-5 model fares on Fashion-MNIST.
- 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.

```
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)
```



7.6.3. Summary

- It is worth comparing the error rates on Fashion-MNIST achievable with LeNet-5 both to the very best possible with MLPs and those with significantly more advanced architectures such as ResNet. LeNet is much more similar to the latter than to the former.
- A second difference is the 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, and finally experimentation with models can now be accomplished in minutes.

8.2. Networks Using Blocks (VGG)

- Progress in this field mirrors that of VLSI (very large scale integration) in chip design where engineers moved from placing transistors to logical elements to logic blocks. Similarly, the design of neural network architectures has grown progressively more abstract, with researchers moving from thinking in terms of individual neurons to whole layers, and now to blocks, repeating patterns of layers.

8.2.1. VGG Blocks

- The basic building block of CNNs

1. A convolutional layer with padding to maintain the resolution
2. a nonlinearity such as a ReLU
3. 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.

- 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.
- Stacking 3 X 3 convolutions has become a gold standard in later deep networks.

- Back to VGG: a VGG block consists of a sequence of convolutions with 3 X 3 kernels with padding of 1 (keeping height and width) followed by a 2 X 2 max-pooling layer with stride of 2 (halving height and width after each block).

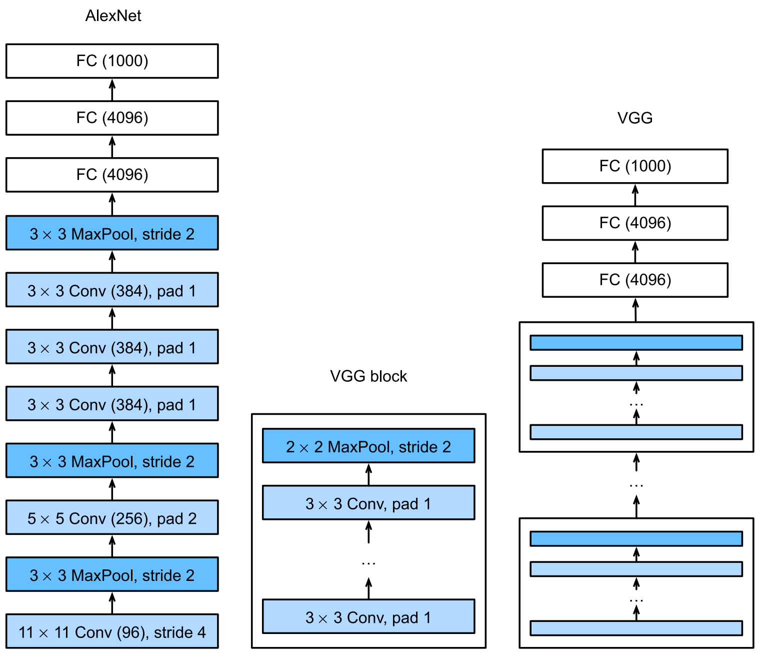
[ ] ↳ 숨겨진 셀 1개

8.2.2. VGG Network

- The VGG Network can be partitioned into two parts

1. The first consisting mostly of convolutional and pooling layers
2. 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.

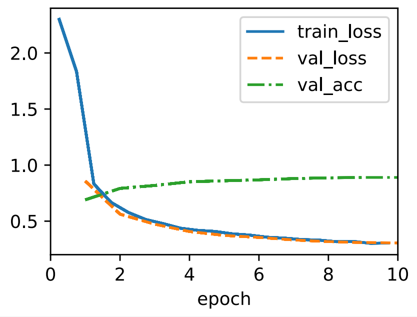


[ ] ↳ 숨겨진 셀 2개

8.2.3. Training

- Since VGG-11 is computationally more demanding than AlexNet we construct a network with a smaller number of channels. This is more than sufficient for training on Fashion-MNIST. The model training process is similar to that of AlexNet in Section 8.1.

```
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)
```



8.2.4. Summary

- While AlexNet introduced many of the components of what make deep learning effective at scale, it is VGG that arguably introduced key properties such as blocks of multiple convolutions and a preference for deep and narrow networks. - It is also the first network that is actually an entire family of similarly parametrized models, giving the practitioner ample trade-off between complexity and speed.

8.6. Residual Networks (ResNet) and ResNeXt

- As we design ever deeper networks it becomes imperative to understand how adding layers can increase the complexity and expressiveness of the network.

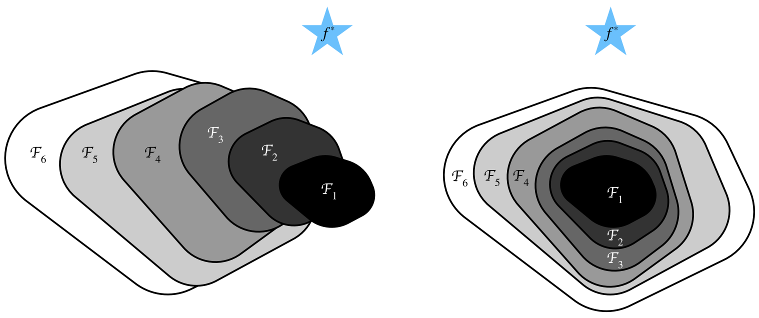
- Even more important is the ability to design networks where adding layers makes networks strictly more expressive rather than just different.

8.6.1. Function Classes

- The class of functions, denoted as $\mathcal{F}$, represents the functions a specific network architecture can reach with certain hyperparameters. Ideally, we aim to find the "truth" function f^*, but it is usually not part of $\mathcal{F}$. Instead, we search for $f^*_{\mathcal{F}}$, the best approximation of f^* within $\mathcal{F}$, by solving an optimization problem:

$$f^*_{\mathcal{F}} \stackrel{\mathrm{def}}{=} \underset{f}{\mathrm{argmin}}\, L(\mathbf{X}, \mathbf{y}, f) \text{ subject to } f \in \mathcal{F}.$$

- We know that regularization may control complexity of $\mathcal{F}$ and achieve consistency, so a larger size of training data generally leads to better $f^*_{\mathcal{F}}$. It is reasonable to assume that a more powerful architecture $\mathcal{F}'$ should result in a better outcome. However, if $\mathcal{F} \not\subseteq \mathcal{F}'$, there is no guarantee of improvement. For non-nested function classes, a larger function class does not always move closer to the "truth" function f^*. With nested function classes, we can avoid this issue.

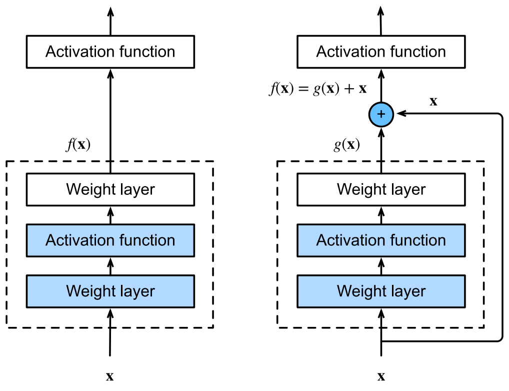


Non-nested function classes Nested function classes

- Thus, only if larger function classes contain the smaller ones are we guaranteed that increasing them strictly increases the expressive power of the network. For deep neural networks, if we can train the newly-added layer into an identity function $f(\mathbf{x}) = \mathbf{x}$, the new model will be as effective as the original model.
- These considerations are rather profound but they led to a surprisingly simple solution, a residual block.

8.6.2. Residual Blocks

- Let's focus on a local part of a neural network, as depicted in below image.
- On the left, the portion within the dotted-line box must directly learn f(x).
- On the right, the portion within the dotted-line box needs to learn the residual mapping g(x) = f(x) - x, which is how the residual block derives its name.
- If the identity mapping f(x) = x is the desired underlying mapping, the residual mapping amounts to g(x) = 0 and it is thus easier to learn.



- ResNet has VGG's full 3 X 3 convolutional layer design. The residual block has two 3 X 3 convolutional layers with the same number of output channels. Each convolutional layer is followed by a batch normalization layer and a ReLU activation function. Then, we skip these two convolution operations and add the input directly before the final ReLU activation function.

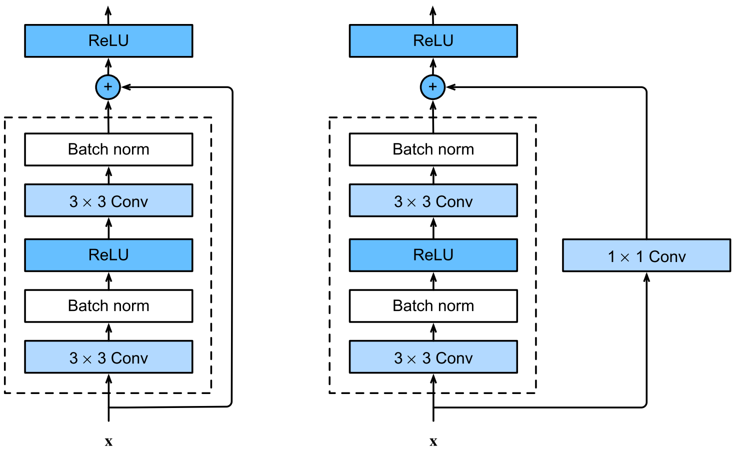
```
class Residual(nn.Module):  #@save
    """The Residual block of ResNet models."""
    def __init__(self, num_channels, use_1x1conv=False, strides=1):
        super().__init__()
        self.conv1 = nn.LazyConv2d(num_channels, kernel_size=3, paddin
                                   stride=strides)
        self.conv2 = nn.LazyConv2d(num_channels, kernel_size=3, paddin
        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()

```

"@save" 주석은 허용되지 않습니다. 허용되는 값은 다음과 같습니다.
[@param, @title, @markdown]

```
    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)
```

- This code generates two types of networks

1. One where we add the input to the output before applying the ReLU nonlinearity whenever *use_1x1conv=False*
2. One where we adjust channels and resolution by means of a 1 X 1 convolution before adding.



- Now let's look at a situation where the input and output are of the same shape, where 1 X 1 convolution is not needed.
- We also have the option to halve the output height and width while increasing the number of output channels.

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

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

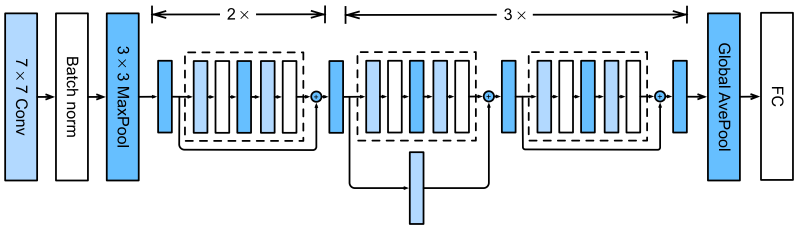
```
blk = Residual(6, use_1x1conv=True, strides=2)
blk(X).shape
```

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

8.6.3. ResNet Model

- The first two layers of ResNet are the same as those of the GoogLeNet we described before. 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.
- Then, we add all the modules to ResNet. Here, two residual blocks are used for each module. Lastly, just like GoogLeNet, we add a global average pooling layer, followed by the fully connected layer output.

- There are four convolutional layers in each module. Together with the first convolutional layer and the final fully connected layer, there are 18 layers in total.
- Therefore, this model is commonly known as ResNet-18. By configuring different numbers of channels and residual blocks in the module, we can create different ResNet models, such as the deeper 152-layer ResNet-152.

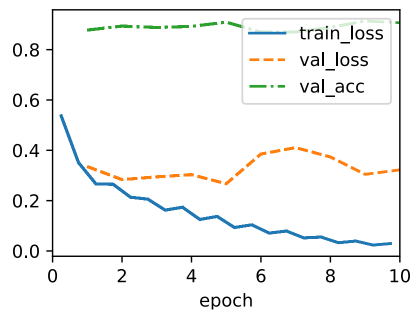


- The resolution decreases while the number of channels increases up until the point where a global average pooling layer aggregates all features.

8.6.4. Training

- 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.

```
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)
```


epoch
train_loss
val_loss
val_acc