```
pip install d2l
```

숨겨진 출력 표시

7.

7.1 From Fully Connected Layers to Convolutions

Is MLP appropriate for every dataset?

1) Tabular Dataset

- Consists of rows (=examples), columns (=features)
- Patterns we seek == interaction among the feature
- Do "not" assume any structure of a feature a priori. Just le the MLP do it.

=> MLP is effective for estimating the interaction between feature, without any prior knowledge

2) Image Dataset

- Millions of megapixel : For MLP, each pixel is treated as feature
- However, the network should be able to detect "objects", regardless of there location (invariant to the location of pixel)

=> If using MLP without any prior structural assumption, requires many parameters

=> Also, MLP suffers from capturing the "representation" of Image dataset

CNN exploits Structural Prior for Dataset

- Natural image has some well-known structure, and human perceptual system is based on processing this information
- What if applying this to Network?

7.1.1 Invariance

Where's Waldo : Invariance required in Object Detection

Find Waldo among many people!

- What Waldo Looks like does not depend upon where Waldo is located
- In other words, the image context is spatial invariant

=> We should sweep the image with a Waldo detector to each patch of image

=> Then, the patch with the highest similarity score indicates the location of Waldo

Rules for DL approach

1) Earliest Layer should respond similarly to similar patch. regardless of location : translation invariance

2) Earliest Layer should focus on local regions. Consider only "neighboring pixel set" : Locality principle

3) Subsequent layer should be able to capture longer-range features of the image : Hierarchical property

=> Translation invariance, Locality principle, Hierarchical property!

7.1.2 Constraining the MLP

Let's see how 3 principles are carried by mathematics!

1) Defining the MLP for image dataset

Let $[X]_{i.j}$ and $[H]_{i,j}$ each denotes pixel, and hidden representation in location (i, j). Then, we can define a layer that transforms from image to hidden representation, by using a fourth-order weight tensors W and second order bias tensor U.

$$\begin{aligned} H_{i,j} &= U_{i,j} + \sum_k \sum_l W_{i,j,k,l} X_{k,l} \\ &= U_{i,j} + \sum_a \sum_b V_{i,j,a,b} X_{i+a,j+b} \end{aligned}$$

In list line, V is a re-indexed matrix : $V_{i,j,a,b} = W_{i,j,i+a,j+b}$ and a, b can be positive and negative. Then, the layer conducts an operation as follows :

" $H_{i,j}$ is computed by summing over pixels in x centered around (i, j) and weighted by $V_{i,j,a,b}$"

2) Translation Invariance

Using the above definition, consider the translation invariance!

- Shift in the input X should lead to a shift in the hidden representation H.
- This is possible only when $V_{i,j,a,b} = V_{a,b}$, (i.e. weight matrix remains same regardless of the center of matrix multiplication, (i, j)), and U is constant.

Thus,

$$H_{i,j} = u + \sum_a \sum_b V_{a,b} X_{i+a,j+b}$$

=> This is a definition of convolution operation

=> Convolution operation is equivalent to linear transform

=> Less number of parameters required!

3) Locality

We should not have to look very far away from location (i, j). Thus, contrain a, b into small range. We should have "small window", thus let the value of V as zero for the range $|a| > k$ or $|b| > k$. Then,

$$H_{i,j} = u + \sum_{a=-k}^{k} \sum_{b=-k}^{k} V_{a,b} X_{i+a,j+b}$$

Now our layers are translation invariant, based on locality principle, with fairly less number of parameters.

- But, this is kind of inductive bias. Does this bias aggres with reality? If so, we are having well-generalizable, sample-efficient models.
- Also, we should make our layer able to focus on wide-range at last. This are achieved by interleaving nonlinearlities and convolutional layers repeatedly.

7.1.3 Convolutions

1) Convolution for continuous values

$$(f * g)(x) = \int f(z) g(x - z) dz$$

2) Convolution for discrete values

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

3) Convolution for discrete values, 2d

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

7.1.4 Channels

Images are usually multi-channels : 3 if RGB. Thus, we should be able to define convolution for multi-channel cases.

- "channels" are "versions" of representation. R version, G version, B version
- In input images, channels are stacked over the first dim. (channel, x, y)
- make the filter V also be stacked!

$$H_{i,j,d} = \sum_{a=-k}^{k} \sum_{b=-k}^{k} \sum_c V_{a,b,c,d} X_{i+a,j+b,c}$$

=> Despite of multi-channel property, convolution can be still represented as linear transformation

=> 1 filter blends the information of every channel and outputs one channel representation map

=> k filter results k channel representation map

Remaining problems

- We should combine all the hidden representation to a single output : "whether there is a Waldo anywhere in the image?"
- We should decide the method for computing things efficiently : "How to combine multiple layers? What is appropriate activation functions? How can we make reasonable design choices for effective networks?

7.2 Convolutions for Images

How does convolution works in practice? Build the code!

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

7.2.1 The Cross-Correlation Operation

Convolutional layers' operations are more accurately described as cross-correlations.

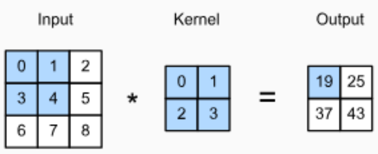


Fig. 7.2.1 Two-dimensional cross-correlation operation. The shaded portions are the first output element as well as the input and kernel tensor elements used for the output computation: $0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$.

output size can be calculated as follows

row : $n_h - k_h + 1$

col : $n_w - k_w + 1$

total : $(n_h - k_h + 1) \times (n_w - k_w + 1)$

```
def corr2d(X, K):
    """Compute 2D cross-correlation."""
    h, w = K.shape
    Y = torch.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            Y[i, j] = (X[i:i + h, j:j + w] * K).sum()
    return Y

X = torch.tensor([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
K = torch.tensor([[0.0, 1.0], [2.0, 3.0]])
corr2d(X, K)
```

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

7.2.2 Convolutional Layers

cross - correlate the input and kernel, then add a 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

How can we detect the edge of an object? We can detect the location of "pixel change"

1) Construct an "image" of 6×8 pixels

- Middel 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.]])
```

2) Then, construct a kernel K

- height 1, width 2 : $[1, -1]$
- At location (i, j), compute $x_{i,j} - x_{(i+1),j}$: If the horizontally adjacent elements are the same, the output is 0. Otherwise nonzero.
- Equivalent to finite difference operator.

```
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.]])
```

3) If the image transformed?

=> The kernel fails to detect edges

=> The kernel is only for vertical edges

```
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

We can also "learn" the kernel. Check this by inspecting input X and output$Y$.

- 1st : construct a conv layer and init kernel as random
- 2nd : Use squared error to compare Y with the output of the conv layer
- 3rd : gradient update for kernel

```
# 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 12.519
epoch 4, loss 4.126
epoch 6, loss 1.522
epoch 8, loss 0.595
epoch 10, loss 0.239
```

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

```
tensor([[ 1.0396, -0.9393]])
```

7.2.5 Cross-Correlation and Convolution

Convolution is a reverse Cross-Correlation. The index is treated in reverse order.

=> refer to the cross-correlation operation as a convolution, though they are not identical strictly.

=> Use the term "element" to refer to an entry of any tensor representing a convolution kernel.

7.2.6 Feature Map and Receptive Field

1) Feature Map

=> Output of the convolutional layer is a feature map

=> It represents a "score" based on similarity with kernel

2) Receptive field

=> Receptive field of element x refers to all the elements that may affect the calculation.

=> all the element which are bound in computation of x

- One layer example

=> if 3x3 image, 2x2 window, 2x2 result : receptive field of one element in feature map is 4

- Two layer example

=>if 3x3 image, 2x2 window, 2x2 result, 2x2 window, 1 result : receptive field of one element in final feature is 9

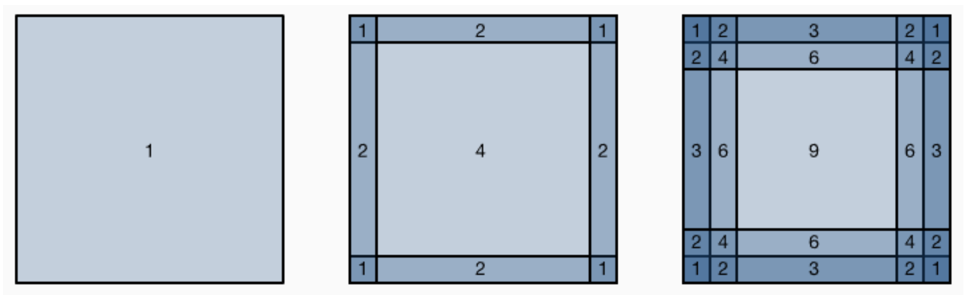
We can build deeper network for enlarging receptive field

7.3 Padding and Stride

Padding "reserves" the original resolution, while strided convolution "shrinks" the original resolution!

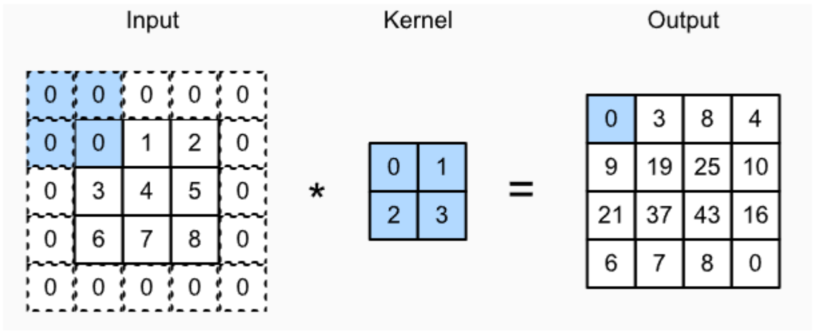
7.3.1 Padding

Pixel Utilization per various size of kernel



- if 1x1 : all pixels are used once
- if 2x2 : 1, 2, 4
- if 3x3 : 1, 2, 3, 2, 4, 6, 3, 6, 9

Increase the original input size, buy adding several zeros!



calculating output size when padded

$$(n_h - k_h + p_h + 1) \times (n_w - k_w + p_w + 1)$$

property & practical tips

- we can set $p_h = k_h - 1, p_w = k_w - 1$ to maintain the size.
- If k_h, k_w odd, pad $p_h/2, p_w/2$ zeros.
- If k_h, k_w even, pad zeros whose number is the upper bound of $p_h/2$ to the top, lower bound of $p_h/2$ to the bottom.
- CNN conventionally use odd kernel sizes, to preserve dimensionality by symmetric padding. Thus we can consider the output $Y[i,j]$ is calculated from the window centered at $X[i,j]$

```
# 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])
```

padding = 1 means $p_h = 2$ so we are adding 1 padding for both top and bottom.

If using $(5,3)$ as a kernel, padding should be $p_h = 4, p_w = 2$ so $(2,1)$ would be passed!

```
# 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])
```

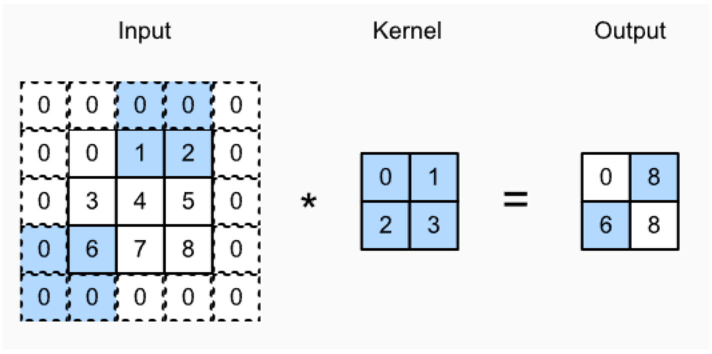
7.3.2 Stride

The step size was 1 in previous example. However, for several reasons such as

- Computational efficiency
- Downsampling process

the step size can be bigger, skipping the intermediate location. This is useful if the convolution kernel is large so that it captures a large area of the underlying image.

strides=(3,2) case



calculating output size when padded & striding

- add p_h, s_h additional space to the original input size n_h, then substract with k_h which indicates the "last move".
- Then, divide this into s_h.

$$\lfloor (n_h - k_h + p_h + s_h)/s_h \rfloor \times \lfloor (n_w - k_w + p_w + s_w)/s_w \rfloor$$

If (n_h, n_w) divisible, then

$$(n_h/s_h) \times (n_w/s_w)$$

```
conv2d = nn.LazyConv2d(1, kernel_size=3, padding=1, stride=2)
comp_conv2d(conv2d, X).shape
```

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

```
conv2d = nn.LazyConv2d(1, kernel_size=(3, 5), padding=(0, 1), stride=(3, 4))
comp_conv2d(conv2d, X).shape
```

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

7.4 Multiple Input and Multiple Output Channels

7.4.1 Multiple Input Channels

Multi-Channel convolution

1) Let input size is $(c, h, w)

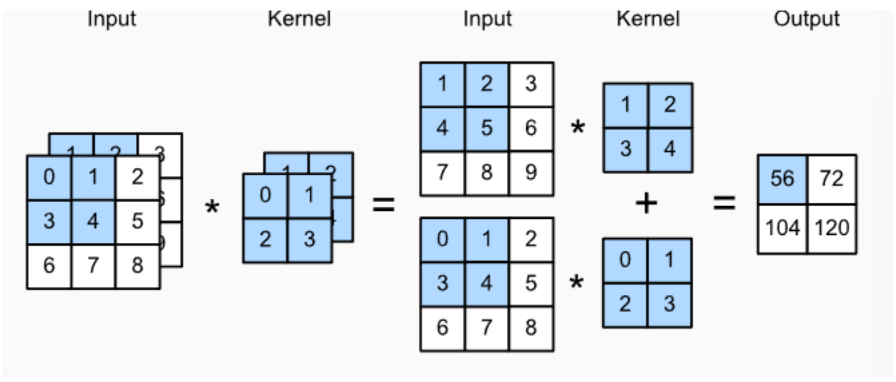
2) Prepare kernel whose channel size is same as the input : (c, k, k)

3) Cross-correlation

4) Add the result in a channel-wise manner (sum over the "channel score" for every elements in feature map"

5) Yield 2nd order tensor for image.

=> one filter yields one output channel, no matter what the size of input channel is. (summed over)



```
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

Multi channel output & shrinked spatial resolution

In the most popular neural network architectures, we actually increase the channel dimension, decrease the spatial resolution.

This can be interpreted as follows :

- We reduce the spatial resolution, since in the end the network should be able to catch the "invariant properties"
- For compensation, we greatly increase the channel depth. We interpret the image in many ways. The representations are learned independently per pixel or per channel.
- Thus, some channel might detect edge information. Other channel might detect the "spherical" shape. ETC.

Operation

- If input channel of image is c_i, then channel of each filter should be same.
- If c_o is the desired number of output channel, we should use c_o number of filter.

Thus,

$$c_o \times c_i \times k_h \times k_w$$

```
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])
```

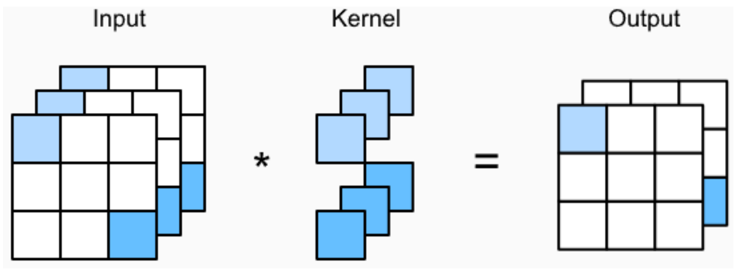
```
corr2d_multi_in_out(X, K)
```

```
tensor([[[ 56.,  72.],
         [104., 120.]],

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

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

7.4.3 1 x 1 Convolution Layer

1×1 convolution Layer

- lacks the ability of capturing the interactions among adjacent elements
- Only operates elements in channelwise manner
- Can be considered as constituting a fully connected layer applied at every single pixel location : transforms c_i corresponding input values into c_o output values.



```
def corr2d_multi_in_out_1x1(X, K):
    c_i, h, w = X.shape
    c_o = K.shape[0]
    X = X.reshape((c_i, h * w))
    K = K.reshape((c_o, c_i))
    # Matrix multiplication in the fully connected layer
    Y = torch.matmul(K, X)
    return Y.reshape((c_o, h, w))
```

```
X = torch.normal(0, 1, (3, 3, 3))
K = torch.normal(0, 1, (2, 3, 1, 1))
Y1 = corr2d_multi_in_out_1x1(X, K)
Y2 = corr2d_multi_in_out(X, K)
assert float(torch.abs(Y1 - Y2).sum()) < 1e-6
```

7.5 Pooling

Pooling also downsamples the image, by picking the value that can represent the property of local region : max, average

Consider the image, where a ballon appears in region A. Though the region is fixed, ballon's accurate location might differ slightly (e.g. shifted by one pixel).

- To make the network capable of detecting balloon, we can think of an operation that mapps each region to 0/1 flag. Then,

$$\text{region A} \to 1$$
$$\text{region B} \to 0$$

.

- This operation discards the accurate location, but yields the "property" of region.
- Convolution does this, but "Pooling" is another way to achieve this.

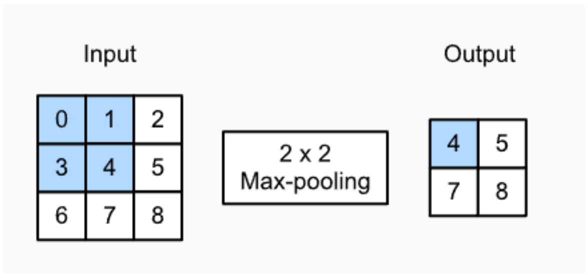
7.5.1 Maximum Pooling and Average Pooling

Pooling Layer has no parameters. It does an deterministic operation! We typically use maximum pooling or average pooling in CNN.

- Average Pooling : We average over adjacent pixels to obtain an image with better signal-to-noise ratio.
- Max Pooling : We pick the maximum value in the window.

max pooling example

- 2x2 kernel, step_size=1
- first element in the output tensor are fixed, though the order of 0, 1, 3, 4 changes : Invariance



Why both use pooling & conv?

Conv is about "detecting a representation with correlation operation". Since it is "learnable", we cannot control the operation.

Thus, use average pooling or max pooling as a "fixed window operation" for aggregating the information.

Output size

Same as conv.

$$(n_h - k_h + 1) \times (n_w - k_w + 1)$$

```
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

Size of output

$$\lfloor (n_h + p_h + s_h - k_h)/s_h \rfloor \times \lfloor (n_w + p_w + s_w - k_w)/s_w \rfloor$$

However, deep learning frameworks default to match pooling window size and stride (i.e. there aren't any overlapping areas, so that the images are divided to various pooling region)

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

If setting a pooling window of shape $(3, 3)$, we get a stride shape of $(3, 3)$ by default.

```
pool2d = nn.MaxPool2d(3)
# Pooling has no model parameters, hence it needs no initialization
pool2d(X)
```

```
tensor([[[[10.]]]])
```

We can specify the architecture, also

```
#specifying padding, stride
pool2d = nn.MaxPool2d(3, padding=1, stride=2)
pool2d(X)
```

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

```
#specifying the window size
pool2d = nn.MaxPool2d((2, 3), stride=(2, 3), padding=(0, 1))
pool2d(X)
```

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

7.5.3 Multiple Channels

While conv layer "sums" the result over channel so that the result is always one-channel feature map, Pooling layer pools each input channel separately.

=> For pooling layer, the channel doesn't shrink.

```
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)

Why CNN?

For classifying image, we should be able to align "image data point" in the space well. (Where we can easily determine the decision boundary)

1) Linear Classifier after flattening image pixel

- Requires lots of parameters, since image has tones of pixels
- Vulnerable to the translation. The location of datapoint differs drastically, though with only one-pixel shift.

2) CNN

CNN uses the "global representation" obtained from "local patches" to classify.

- Invariance
- Less number of parameter required
- "Learnable" feature extractor + Deterministic pooling operation

=> CNN also employs linear classifier, but extracts feature by learning before classifying. Thus, the "representation" of each image are well aligned in the space that enables partitioning.

Introduction to LeNet

LeNet is one of the first published CNNs for computer vision tasks. Yann Lecun successfully trained CNNs via backpropagaions.

They matched the performance of SVM, and finally dominating other approaches leading to an error rate of less than 1% per digit.

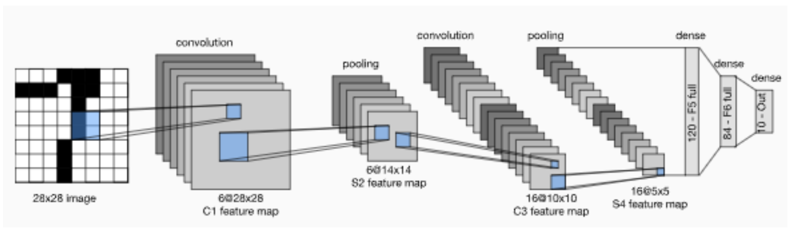
ATM machines use LeNet even nowadays!

7.6.1 LeNet

High level Explanation

1) LeNet consists of two parts

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

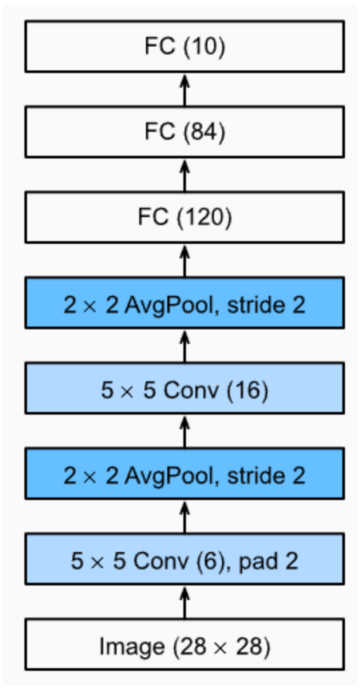


2) Basic Units of convolutional block are

- convolutional layer with 5x5 kerel
- sigmoid activation function (ReLU not discovered)
- 2x2 average pooling operation (Max-pooling not discovered)
- The numbers of channels are increased as a trade-off for spatial reduction : 6 output channels in first layer, 16 in second layer

3) Conv to dense

the feature map is flattened to be fed to fc layers

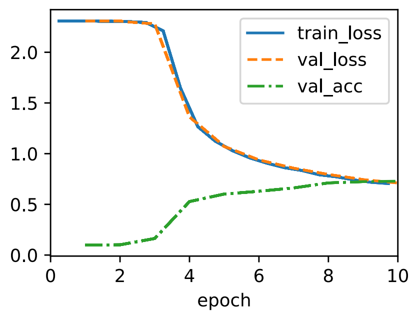


```
def init_cnn(module):
    """Initialize weights for CNNs."""
    if type(module) == nn.Linear or type(module) == nn.Conv2d:
        nn.init.xavier_uniform_(module.weight)

class LeNet(d2l.Classifier):
    """The LeNet-5 model."""
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.LazyConv2d(6, kernel_size=5, padding=2), nn.Sigmoid(),
            nn.AvgPool2d(kernel_size=2, stride=2),
            nn.LazyConv2d(16, kernel_size=5), nn.Sigmoid(),
            nn.AvgPool2d(kernel_size=2, stride=2),
            nn.Flatten(),
            nn.LazyLinear(120), nn.Sigmoid(),
            nn.LazyLinear(84), nn.Sigmoid(),
            nn.LazyLinear(num_classes))


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

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

```
Conv2d output shape:     torch.Size([1, 6, 28, 28])
Sigmoid output shape:    torch.Size([1, 6, 28, 28])
AvgPool2d output shape:  torch.Size([1, 6, 14, 14])
Conv2d output shape:     torch.Size([1, 16, 10, 10])
Sigmoid output shape:    torch.Size([1, 16, 10, 10])
AvgPool2d output shape:  torch.Size([1, 16, 5, 5])
Flatten output shape:    torch.Size([1, 400])
Linear output shape:     torch.Size([1, 120])
Sigmoid output shape:    torch.Size([1, 120])
Linear output shape:     torch.Size([1, 84])
Sigmoid output shape:    torch.Size([1, 84])
Linear output shape:     torch.Size([1, 10])
```

7.6.2 Training

```
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.

8.2 Networks Using Blocks (VGG)

The design of neural network architectures has grown progressively more abstract. Researchers now use the "common desigining guide" based on "block" heuristics!

- LeNet and AlexNet treated each layer individually
- VGG net considers "block" : there are recommended repetition in patterns of layers, so make them as "block"
- Nowadays, we consider "pretrained models". We fine-tune the large pretrained models called "foundation models"

8.2.1 VGG blocks

1) Basic Building Block of CNN : shallow is not good

Before VGG, basic building block of CNN was sequence of following

- (i) a conv layer with padding (resolution remained)
- (ii) a nonlinearity
- (iii) a pooling layer to reduce the resolution

=> The spatial resolution decreases quite rapidly.

=> This imposes a hard limit of $log_2 d$ convolutional layers on the network before all dimensions d are used up. (If32 x 32 images, pooling layer halves the dimension so we can only use $log232$ = 5 sequences of (i), (ii), (iii)

=> This leads to "loss of specific information". The information are pooled before obtaining rich representation.

=> Also, image is shrinked to the minimal before global feature is considered.

2) Multiple Convolution? : deep networks vs wide networks

Consider two design options : two 3x3 convolution vs single 5x5 convolution

- They touch the same pixels (same receptive field)
- The former uses $2 \times c \times (9+1)$ parameters
- The latter uses $1 \times c \times (25+1)$ parameters.

=> Deep and narrow networks outperform shallow networks

=> Thus, stacking $3x3$ convolution has become a gold standard

3) Stacking Conv : Large receptive field, high representational power

1. Pooling layers reduces the spatial resolution. If we use $\mathrm{Conv} \to \mathrm{Pool}$ architecture repeatedly, enrichment of representation is stopped. Global representation is not learned.
2. Thus we should think of using all the dimensions of images before shrinking.Instead of wide kernels, deep and narrow networks are parameter-efficient.

=> VGG net "stacks" the conv layer to enable large receptive field & high representation, before the image shrinks to the minimal!

=> although maintaining the spatial resolution, we can enlarge the receptive field!

```
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

Image goes through multiple repitition of conv & nonlinear pairs (which maintains resolution), and one pooling layer (which reduces spatial resolution).

VGG Network defines a "family" of networks rather than just a specific manifestation

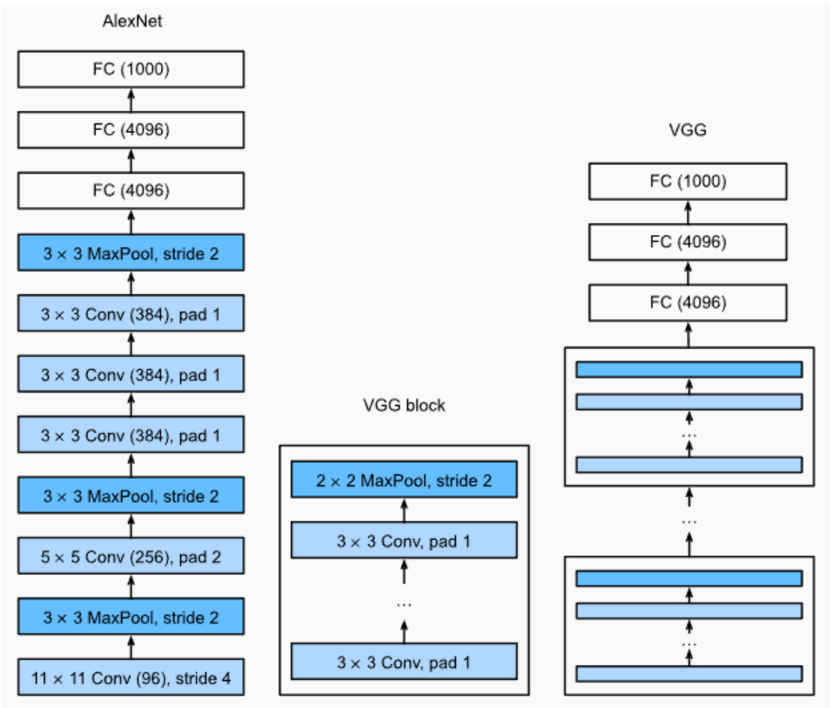


Fig. 8.2.1 From AlexNet to VGG. The key difference is that VGG consists of blocks of layers, whereas AlexNet's layers are all designed individually.

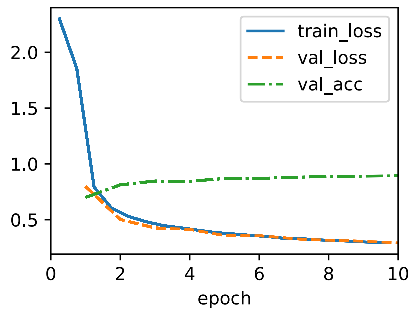
```
def vgg_block(num_convs, out_channels):
    layers = []
    for _ in range(num_convs):
        layers.append(nn.LazyConv2d(out_channels, kernel_size=3, padding=1))
        layers.append(nn.ReLU())
    layers.append(nn.MaxPool2d(kernel_size=2,stride=2))
    return nn.Sequential(*layers)

class VGG(d2l.Classifier):
    def __init__(self, arch, lr=0.1, num_classes=10):
        '''
        arch : list of tuples (num_convs, out_channels) -> each tuple is
               about one block
        conv_blks : list of nn.Sequential blocks
        '''
        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])
```

8.2.3 Training

```
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

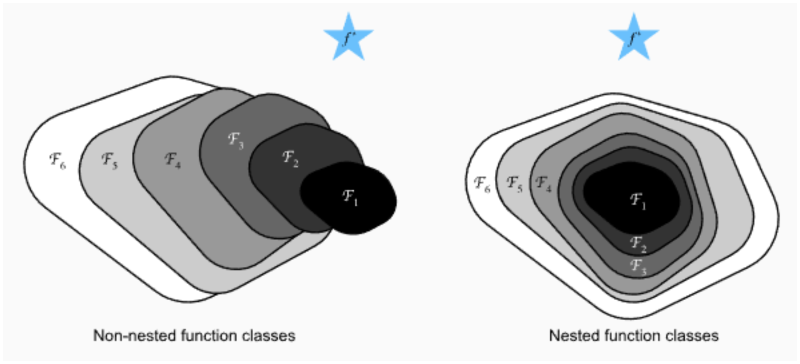
How can adding layers increase the complexity and expressiveness of the networks?

How can adding layers makes "more expressive" rather than "different" function?

8.6.1 Function Classes

1) Does the "large model" always yield better results?

- Consider F, the class of functions that a specific network architecture can reach.
- for all $f \in F$, there exists some set of parameters that can be obtained through training on a suitable dataset.
- there exists a true function f^*, but we chose the best model f^*_F under our design space by optimization
- Then, the "size" of design space doesn't guarantee the closer distance to the truth function, as depicted in below:



=> F_6 is larger than F_3, but the distance from f^* might be far.

=> Thus, the enlarged functional space should hold some conditions : If $F_1 \in F_2 \in \cdots \in F_6$, we can guarantee that the "expressional power" has increased.

2) ResNet : How can we make "nested functional class" in DL architecture?

The newly-added layer should be able to be trained to become an identity function $f(x) = x$. Then, the new enlarged model may get a "better solution".

If trained to identity function, it yields same operation with the original function before adding layers.

Residual Network fulfills this requirements by Residual blocks (He et. al. 2016). ResNet won the ImageNet Large Scale Visual Recognition Challenge in 2015. Nowadays, RNN, Transformers, and even GNN employs this idea.

8.6.2 Residual Blocks

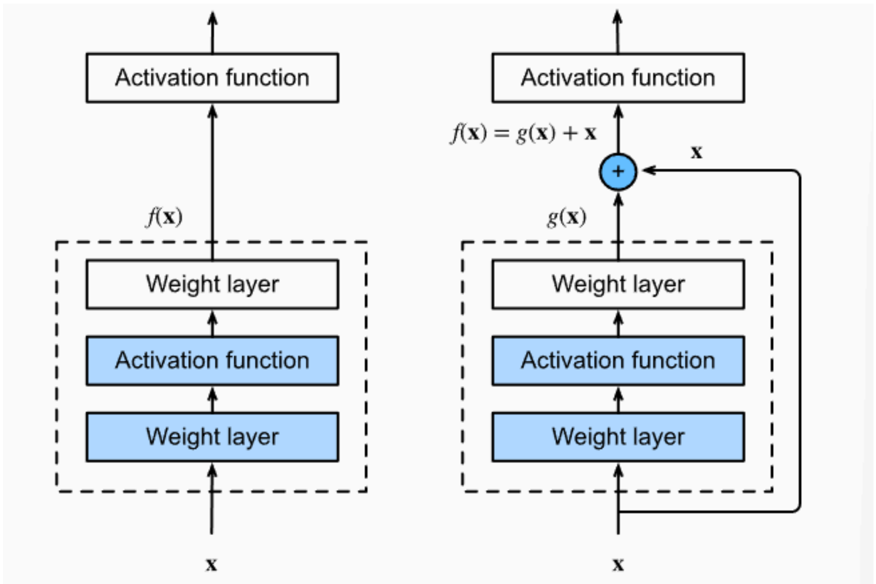
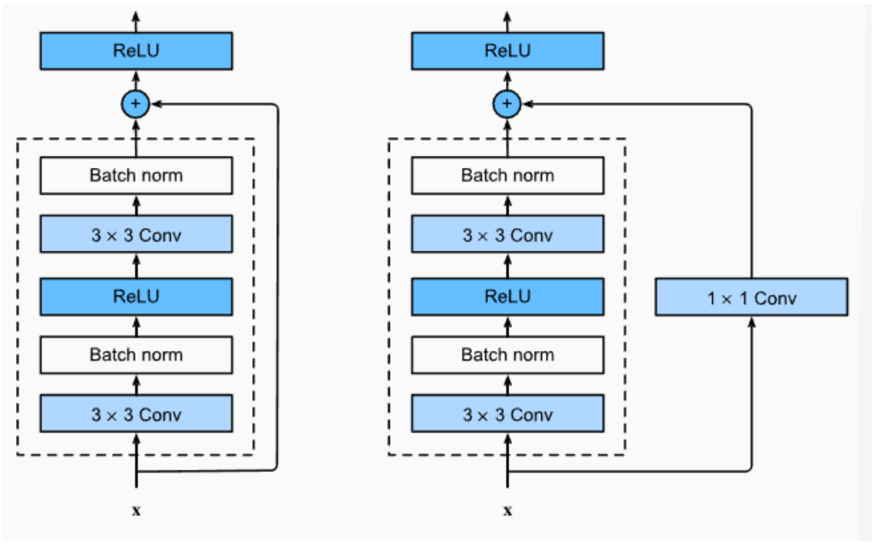


Fig. 8.6.2 In a regular block (left), the portion within the dotted-line box must directly learn the mapping $f(\mathbf{x})$. In a residual block (right), the portion within the dotted-line box needs to learn the residual mapping $g(\mathbf{x}) = f(\mathbf{x}) - \mathbf{x}$, making the identity mapping $f(\mathbf{x}) = \mathbf{x}$ easier to learn.

- Without residual connection, it is still possible to achieve identity mapping. However, the stochastic property of learning prevents this.
- With residual connection, we can make identity mapping by just setting all the weights to zero in the dotted box. (i.e. $g(x) = 0$
- This is special case of multi-branch inception block : two branches, where one of which is the identity mapping

convolutional layer design with BN?

ResNet follows VGG's full 3 x 3 convolutional layer design.



The skip connections & output of the network is integrated before entering the activation function. We need to introduce an additional 1 x 1 convolutional layer to transform the input into the desired shape.

```
class Residual(nn.Module):
    """The Residual block of ResNet models."""
    def __init__(self, num_channels, use_1x1conv=False, strides=1):
        super().__init__()
        self.conv1 = nn.LazyConv2d(num_channels, kernel_size=3, padding=1,
                                   stride=strides)
        self.conv2 = nn.LazyConv2d(num_channels, kernel_size=3, padding=1)
        if use_1x1conv:
            self.conv3 = nn.LazyConv2d(num_channels, kernel_size=1,
                                       stride=strides)
        else:
            self.conv3 = None
        self.bn1 = nn.LazyBatchNorm2d()
        self.bn2 = nn.LazyBatchNorm2d()

    def forward(self, X):
        Y = F.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3:
            X = self.conv3(X)
        Y += X
        return F.relu(Y)
```

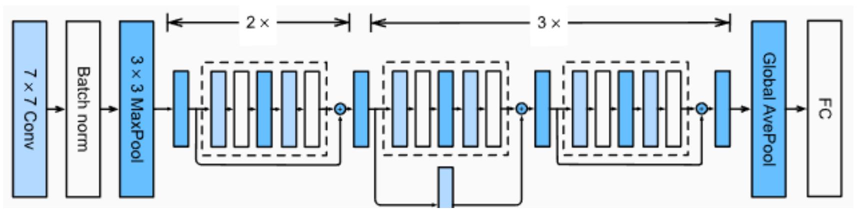
```
from torch.nn import functional as F

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

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

```
# additional option : stride = 2 for halving & setting channel to 6

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

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

8.6.3 ResNet Model



ResNet uses four modules made up of residual blocks. Each of them uses several residual blocks.

ResNet-18 model has 18 layers :

- four convolutional layers in each module excluding the 1x1 convolutional layer
- 7x7 convolutional layer at first
- final fully connected layer

1) Def residual block

conv1 -> bn -> relu -> conv2 -> bn -> add (optional conv3) -> relu

```
class Residual(nn.Module):
    """The Residual block of ResNet models."""
    def __init__(self, num_channels, use_1x1conv=False, strides=1):
        super().__init__()
        self.conv1 = nn.LazyConv2d(num_channels, kernel_size=3, padding=1,
                                   stride=strides)
        self.conv2 = nn.LazyConv2d(num_channels, kernel_size=3, padding=1)
        if use_1x1conv:
            self.conv3 = nn.LazyConv2d(num_channels, kernel_size=1,
                                       stride=strides)
        else:
            self.conv3 = None
        self.bn1 = nn.LazyBatchNorm2d()
        self.bn2 = nn.LazyBatchNorm2d()

    def forward(self, X):
        Y = F.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3:
            X = self.conv3(X)
        Y += X
        return F.relu(Y)
```

2) First two layers

7x7 conv, 64 c, 2 s -> bn -> relu -> 3x3 max-pooling, 2 s

```
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))
```

3) 4 modules made up of residual blocks

each module..

- appends residual blocks as given in arch
- First module doesn't need to halve the input channel, since max-pooling layer already halved it.
- Subsequent modules double the channels, with halving the width and height by stride=2
- here, two residual blocks for each module

```
@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)
```

4) Final

Add the Average Pool layer & fully connected layer after whole. This aggregates the global information and predicts the possibility for each classes.

```
@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)
```

5) Checking the flow of input

(2,64), (2,128), (2, 256), (2, 512) each means the different types of module

- first module : 2 resblock, 64 channel, first_block = True thus no halving
- second module : 2 resblock, 128 channel, halved
- third module : 2 resblock, 256 channel, halved
- fourth module : 2 resblock, 512 channel, halved

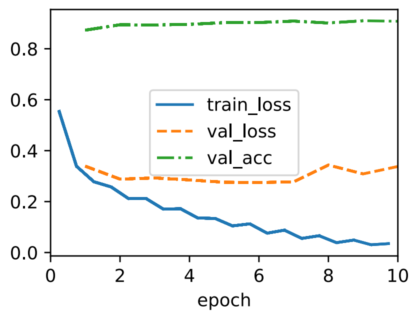
=> 1x1 conv is needed only except first module : first module doesn't halve the input, thus the channel are remained same. However, subsequent module halves the input and doubles the channel. For adding, the input channel should be modified with 1x1 conv.

```
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

```
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)
```



low val loss, train loss & substantially high val_acc : No overfitting, high accuracy achieved!