SDM5013:
Deep Learning and Reinforcement Learning

Zhiyun Lin

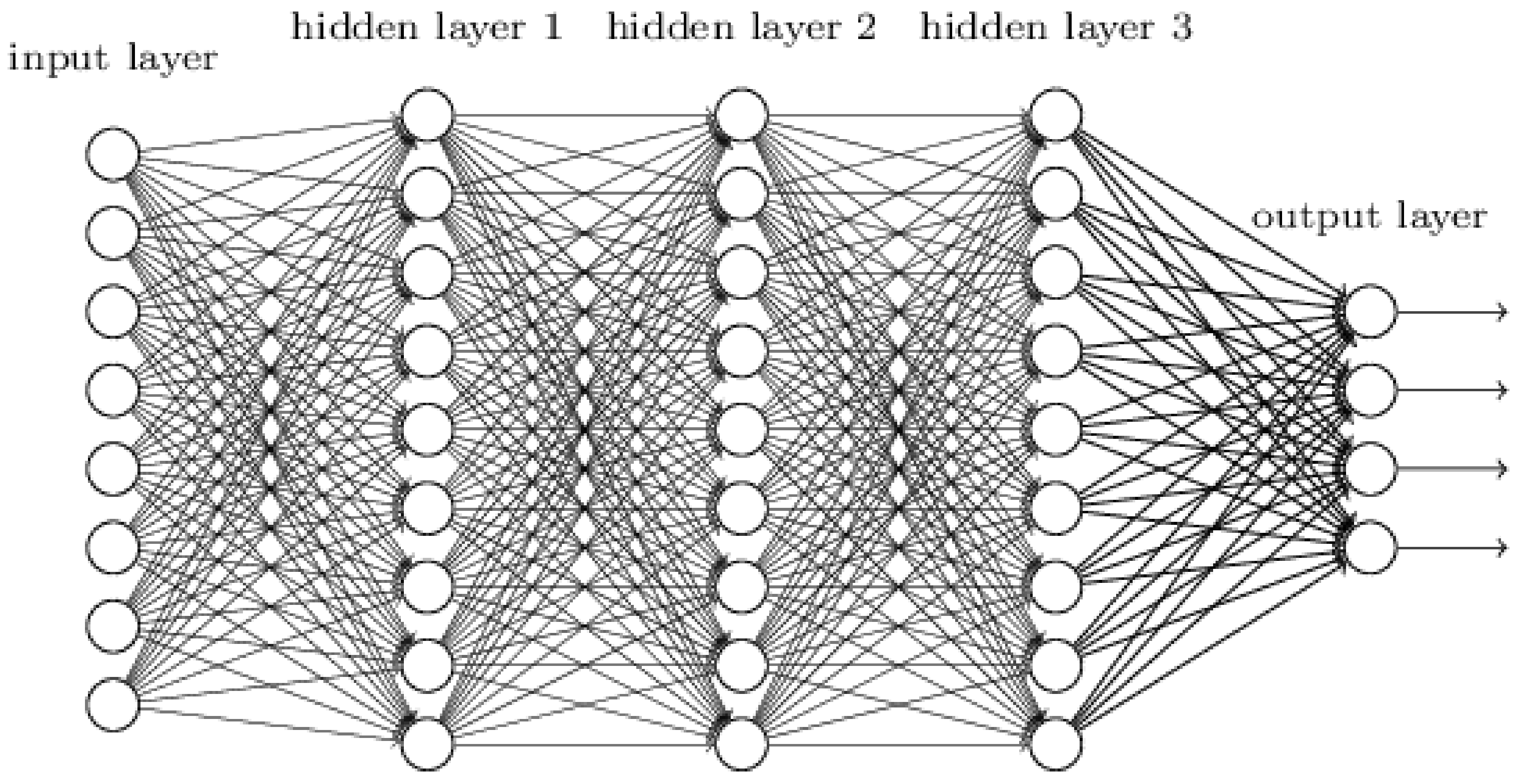
南方科技大学 SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY | SDIM 设计智造学院 School of System Design and Intelligent Manufacturing

Today

- CNN

- Modern Convolutional Neural Networks
  - AlexNet
  - VGG
  - NiN
  - GoogleNet
  - ResNet
  - DenseNet

From fully connected network to others



- We know it is good to learn a small model.
- From this fully connected model, do we really need all the edges?
- Can some of these be shared?

Why CNN for images

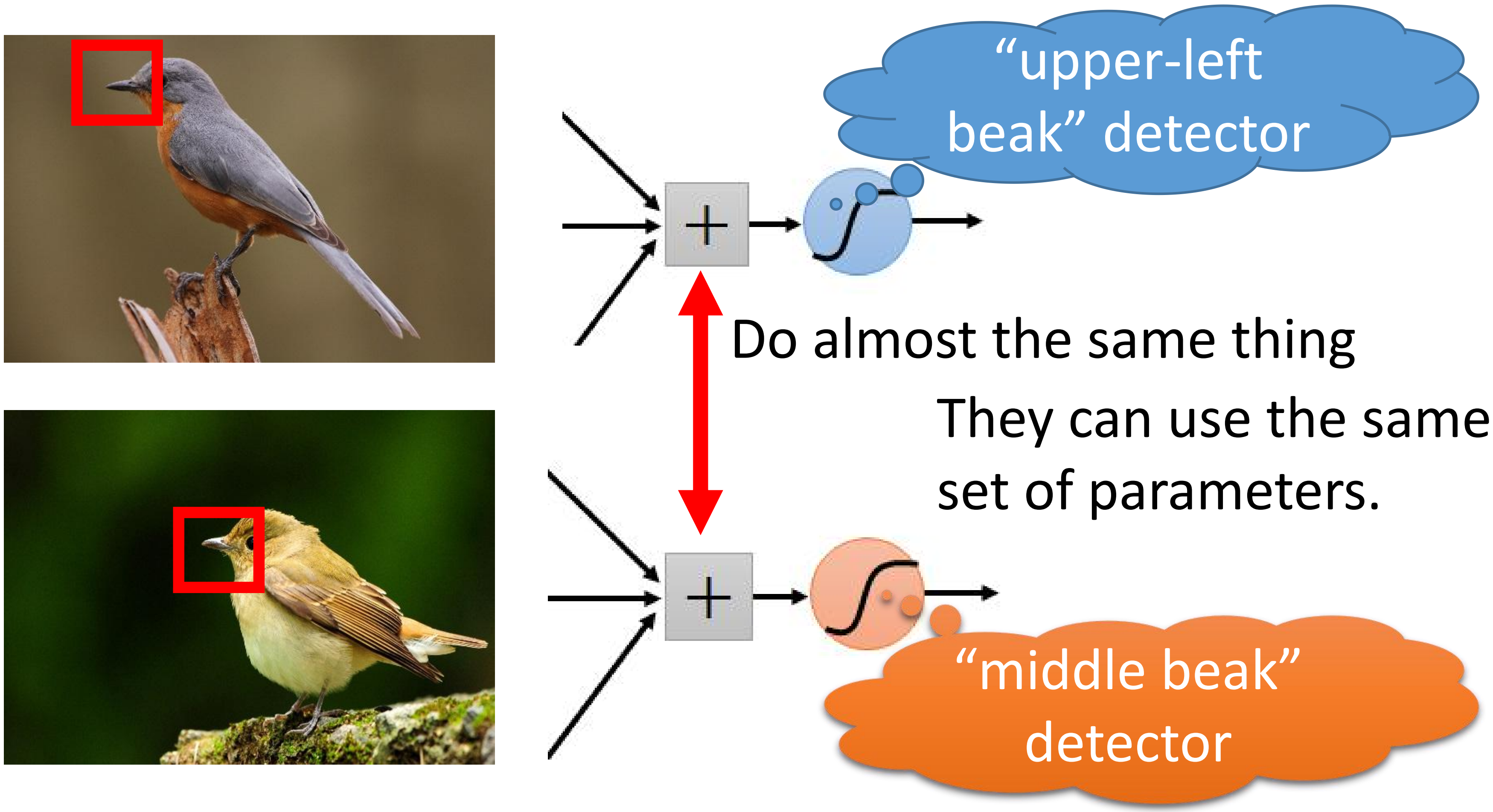
- Some patterns are much smaller than the whole image

A neuron does not have to see the whole image to discover the pattern.

Connecting to small region with less parameters

Why CNN for images

- The same patterns appear in different regions.



Intuitions for CNN

- 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**.

- 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.

CNN vs. Vision Transformer (ViT)

Why CNN for images

- Subsampling the pixels will not change the object

bird

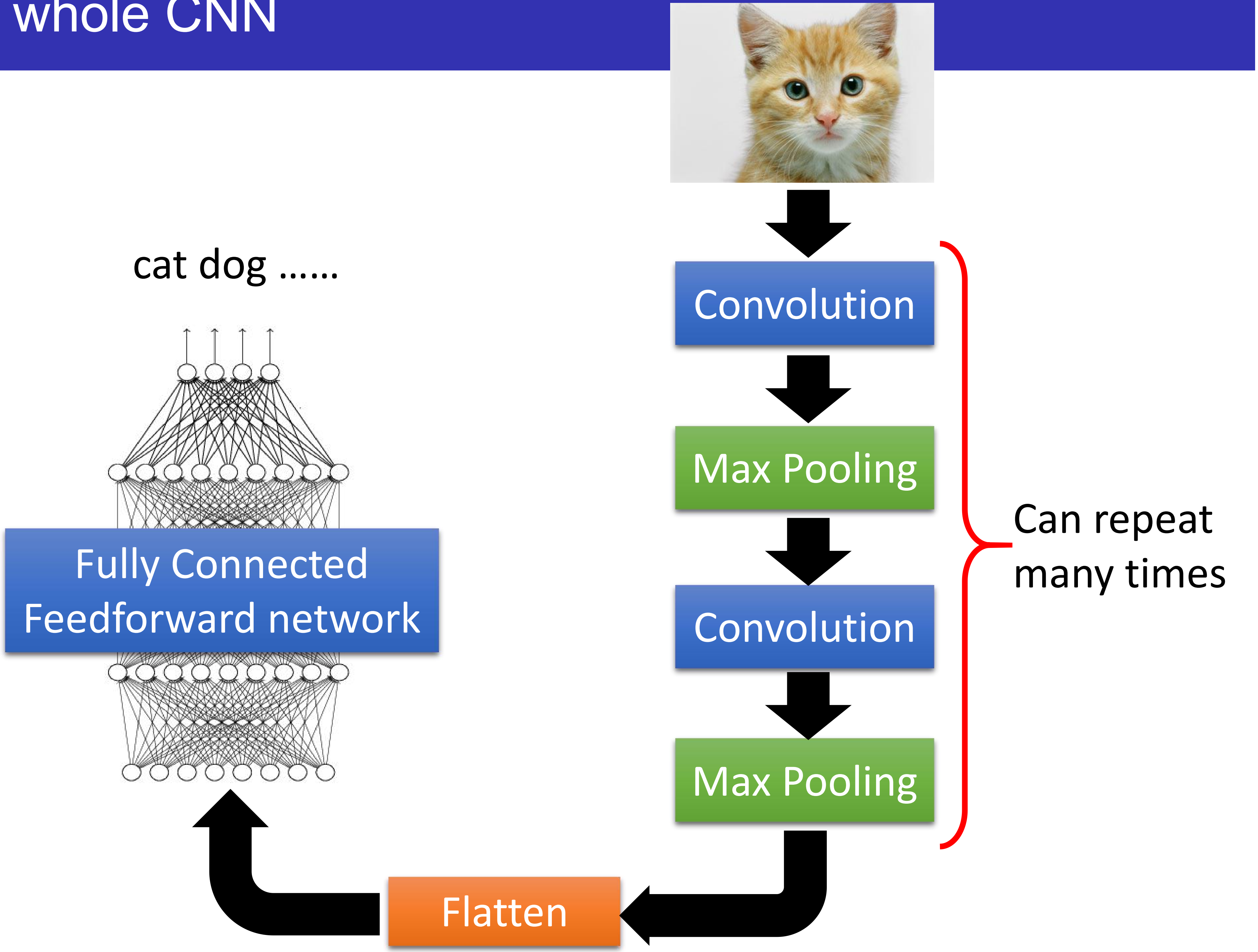
subsampling

bird

We can subsample the pixels to make image smaller

→ Less parameters for the network to process the image

The whole CNN



The whole CNN

Convolution

Max Pooling

Convolution

Max Pooling

Can repeat many times

Flatten

Property 1

- Some patterns are much smaller than the whole image

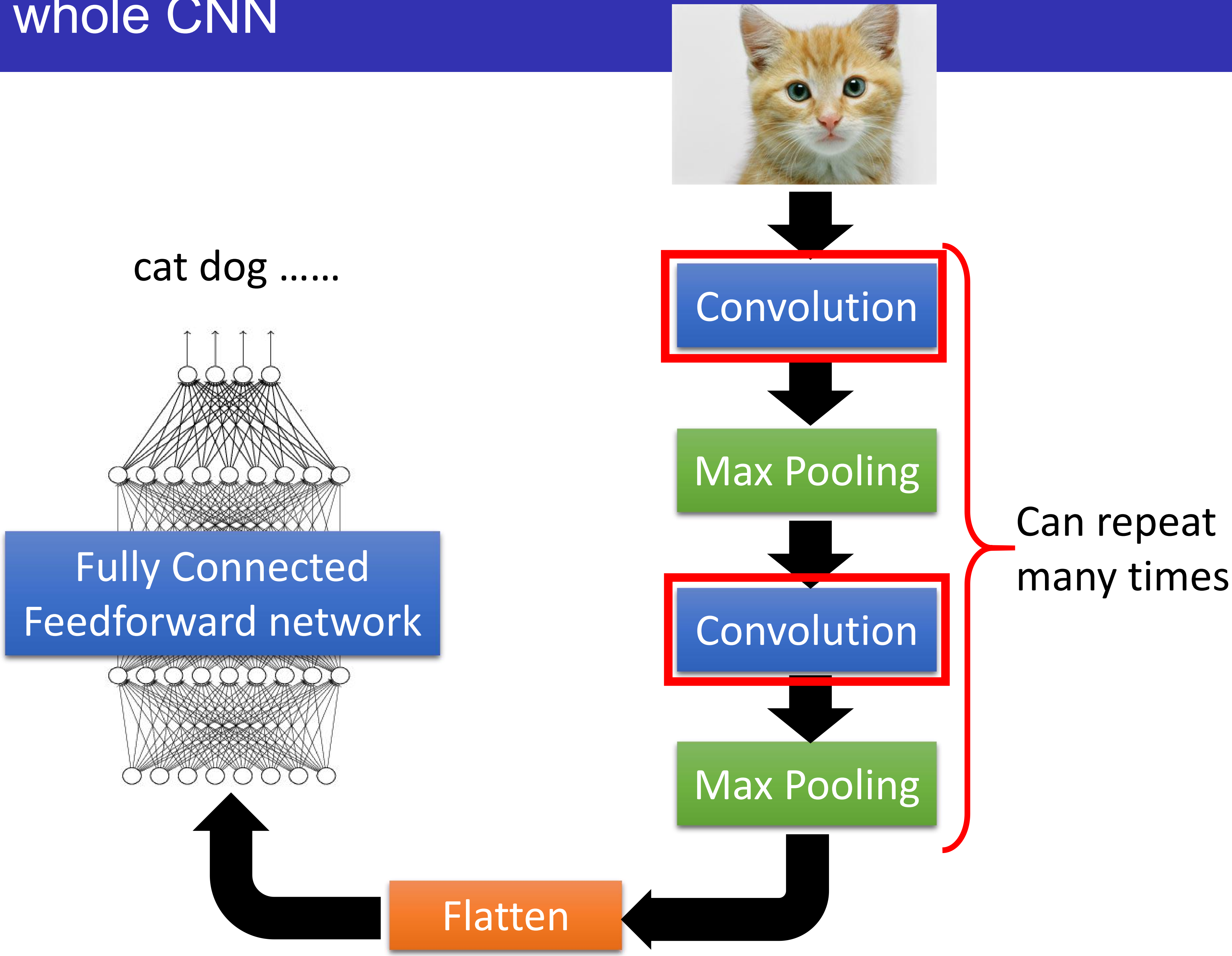
Property 2

- The same patterns appear in different regions.

Property 3

- Subsampling the pixels will not change the object

The whole CNN



CNN—Convolution

Those are the network parameters to be learned.

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

Matrix

-1	1	-1
-1	1	-1
-1	1	-1

Filter 2

Matrix

⋮

Property 1

Each filter detects a small pattern (3 x 3).

CNN—Convolution

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

stride=1

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image

3 -1

CNN—Convolution

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

If stride=2

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image

3 -3

We set stride=1 below

CNN—Convolution

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

stride=1

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image

3	-1	-3	-1
-3	1	0	-3
-3	-3	0	1
3	-2	-2	-1

Property 2

CNN—Convolution

-1	1	-1
-1	1	-1
-1	1	-1

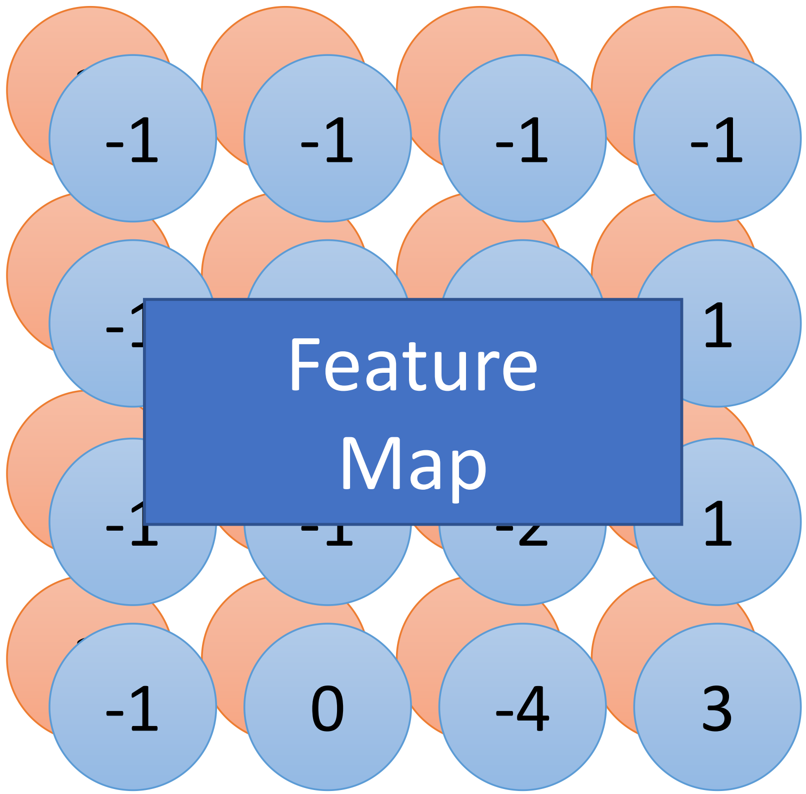
Filter 2

stride=1

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image

Do the same process for every filter



4 x 4 image

CNN—Colorful images

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

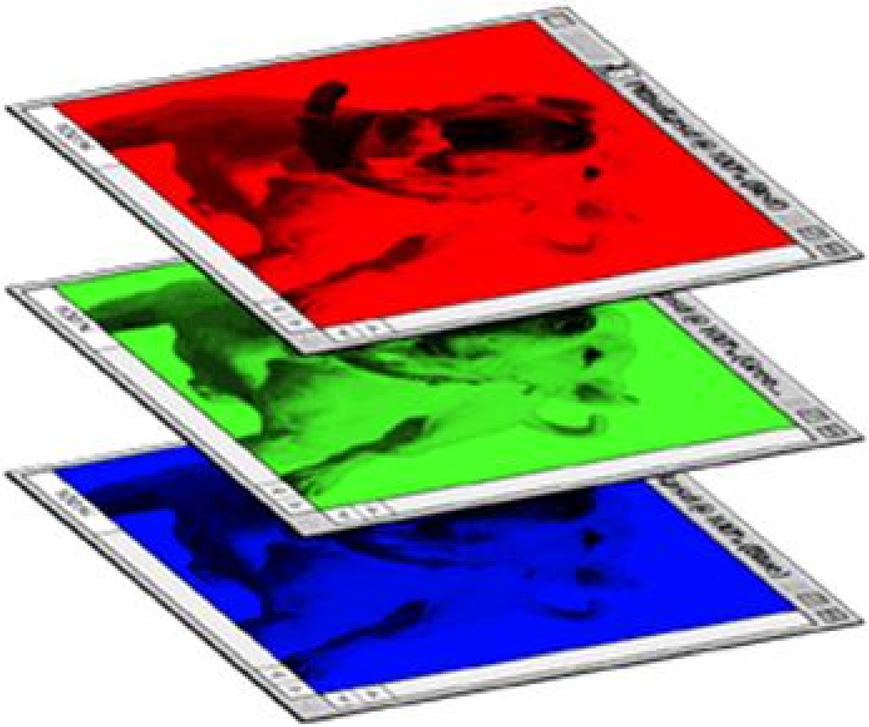
Filter 1

1	-1	-1
-1	1	-1
-1	-1	1

Filter 2

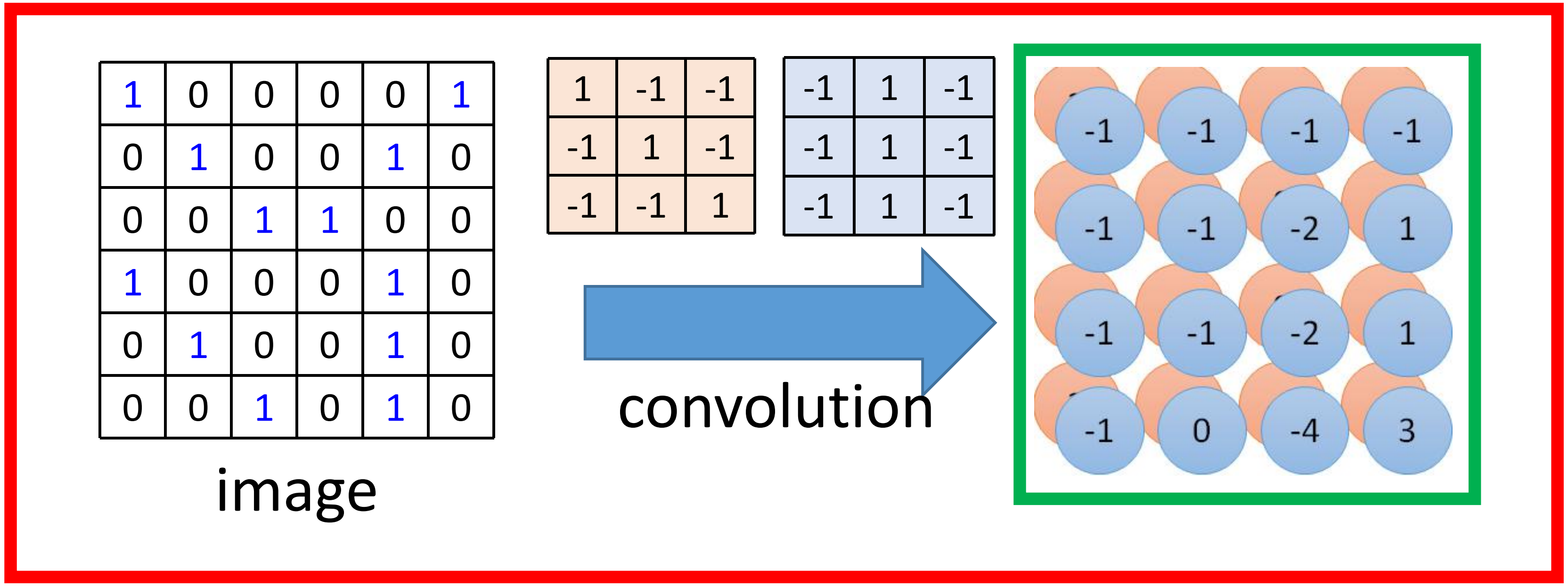
-1	1	-1
-1	1	-1
-1	1	-1

Colorful image

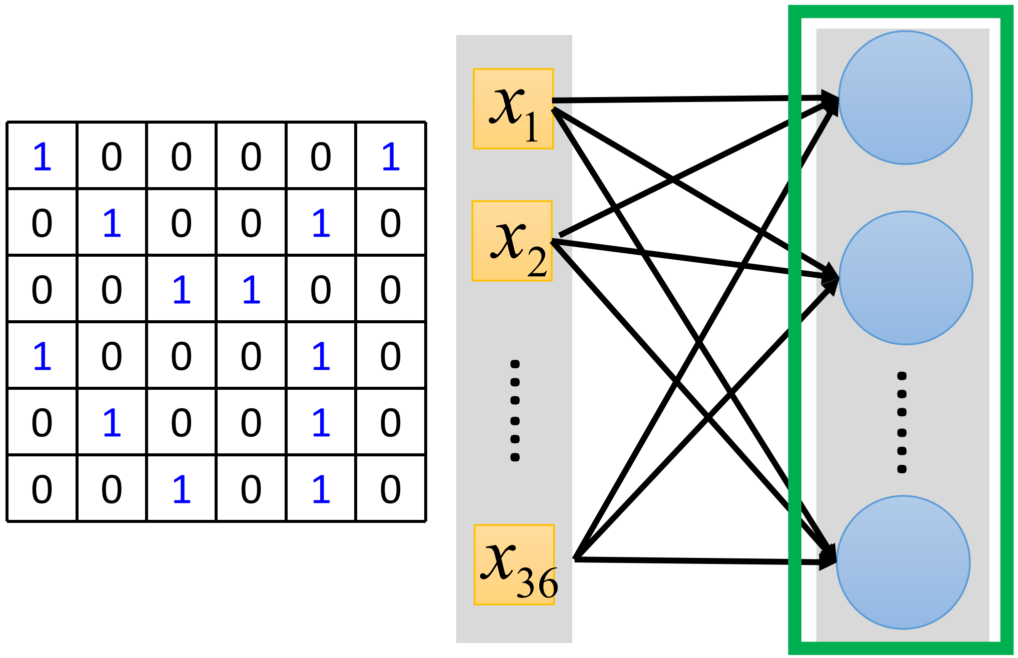


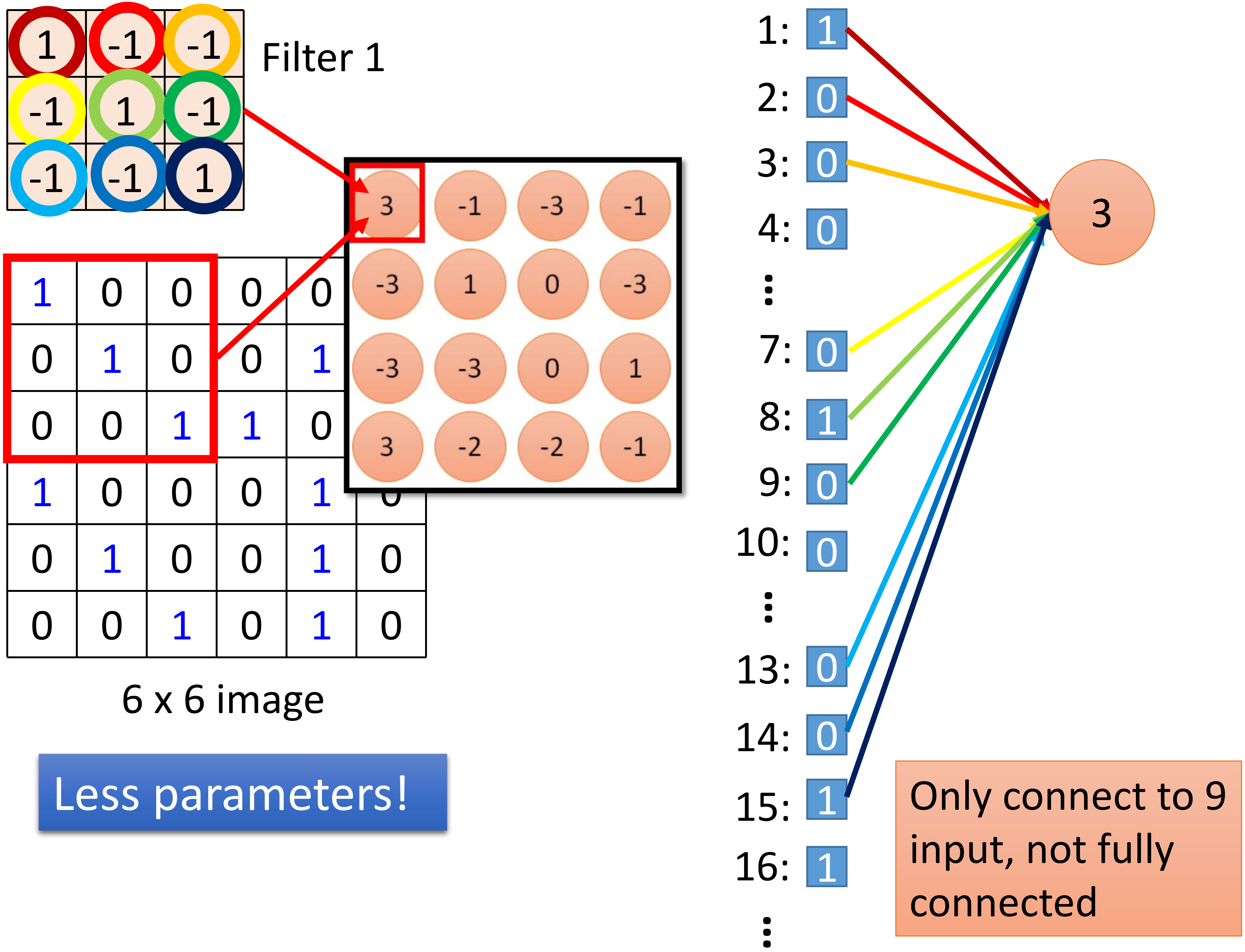
1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

Convolution vs. Fully Connected

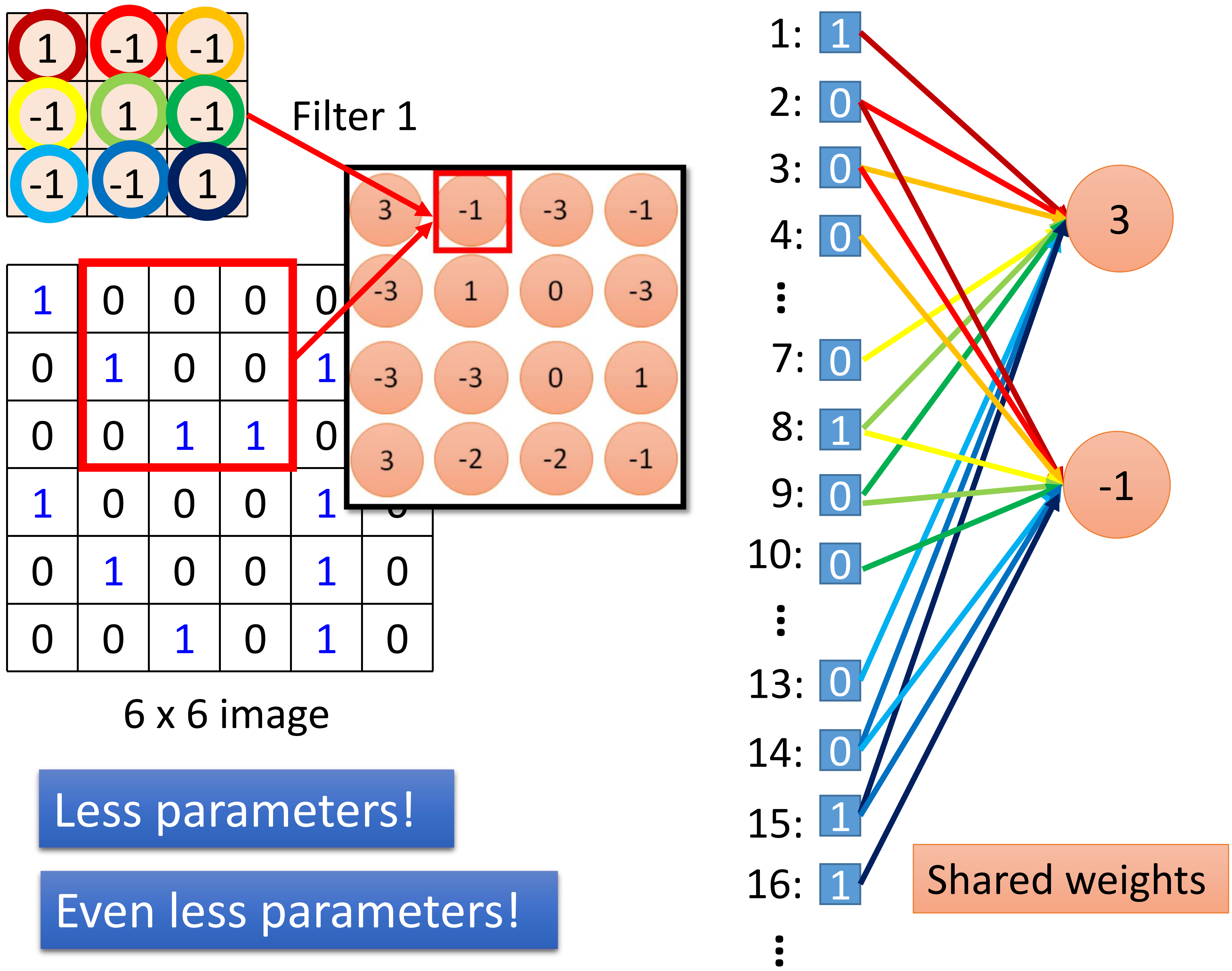


Fully-connected



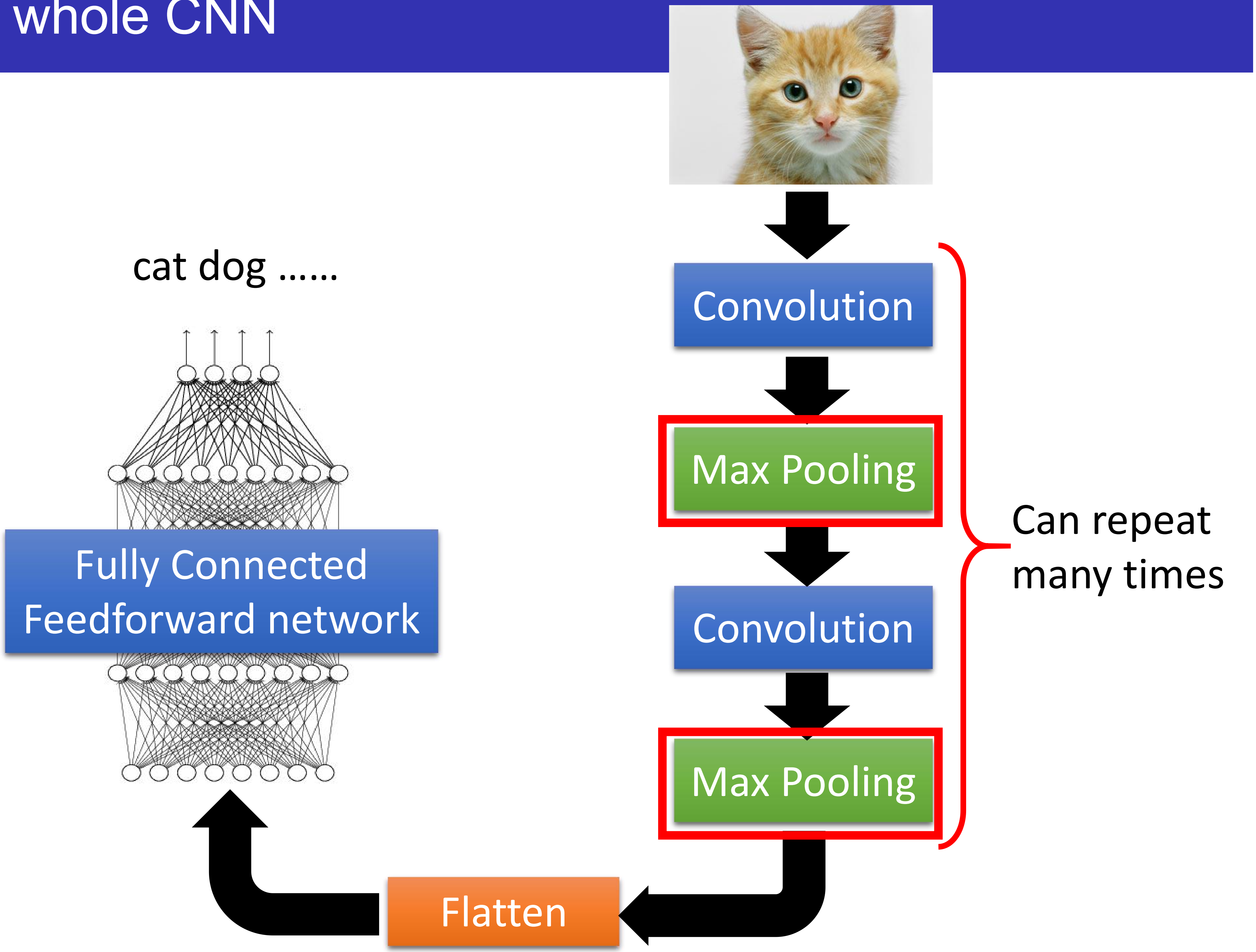


Filter 1
1 -1 -1
-1 1 -1
-1 -1 1
3 -1 -3 -1
-3 1 0 -3
-3 -3 0 1
3 -2 -2 -1
1 0 0 0 0 1
0 1 0 0 1 0
0 0 1 1 0 0
1 0 0 0 1 0
0 1 0 0 1 0
0 0 1 0 1 0
6 x 6 image
Less parameters!
1: 1
2: 0
3: 0
4: 0
7: 0
8: 1
9: 0
10: 0
13: 0
14: 0
15: 1
16: 1
3
Only connect to 9 input, not fully connected



1 -1 -1
-1 1 -1
-1 -1 1
Filter 1
3 -1 -3 -1
-3 1 0 -3
-3 -3 0 1
3 -2 -2 -1
1 0 0 0 0 1
0 1 0 0 1 0
0 0 1 1 0 0
1 0 0 0 1 0
0 1 0 0 1 0
0 0 1 0 1 0
6 x 6 image
Less parameters!
Even less parameters!
1: 1
2: 0
3: 0
4: 0
7: 0
8: 1
9: 0
10: 0
13: 0
14: 0
15: 1
16: 1
3
-1
Shared weights

The whole CNN



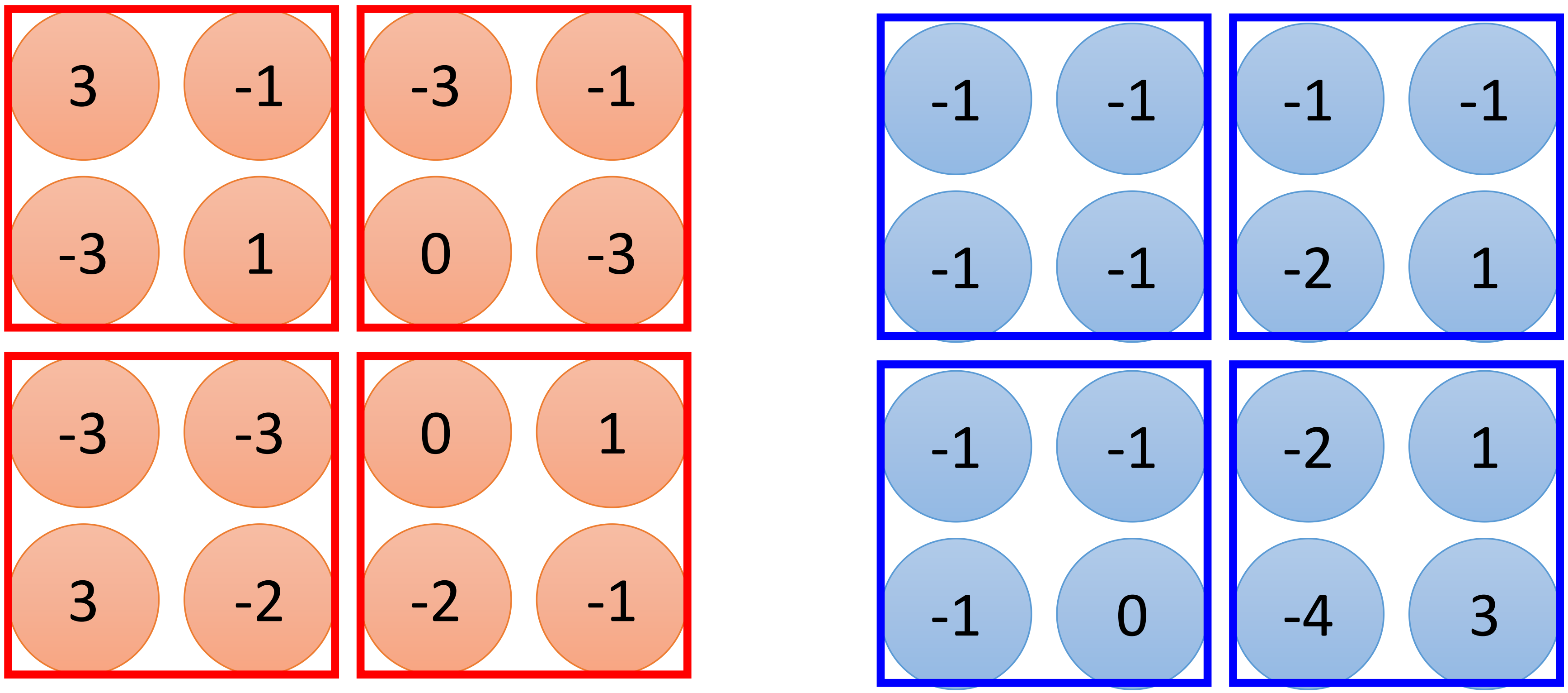
CNN—Max Pooling

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

-1	1	-1
-1	1	-1
-1	1	-1

Filter 2

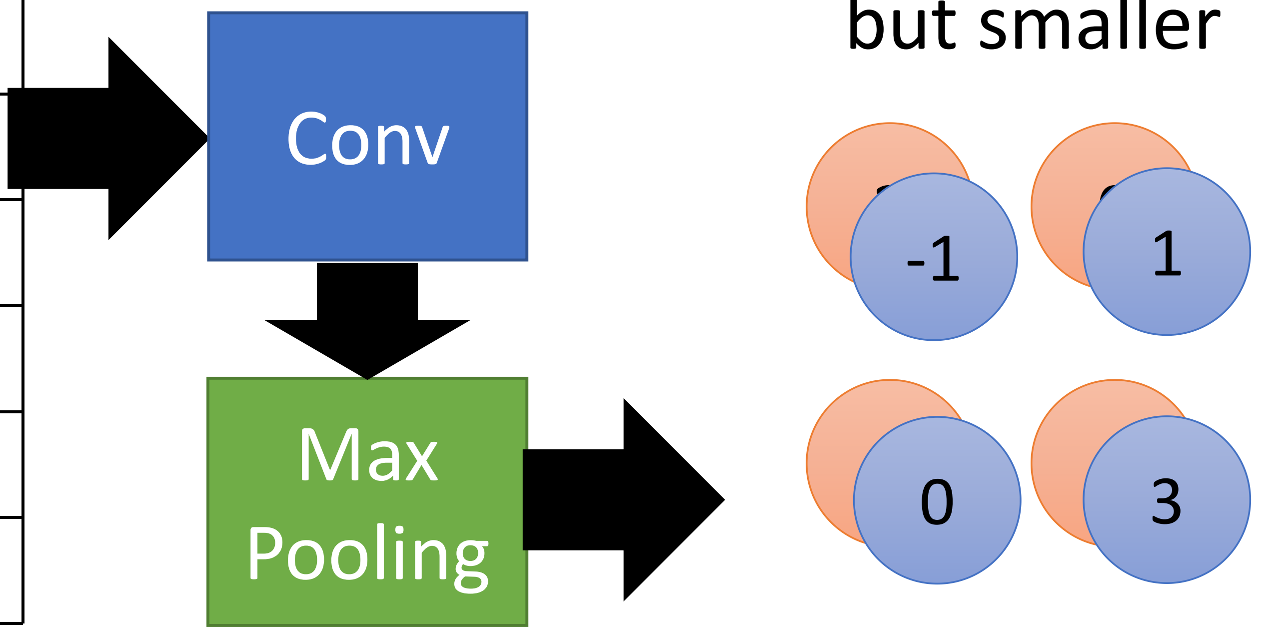


CNN—Max Pooling

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image

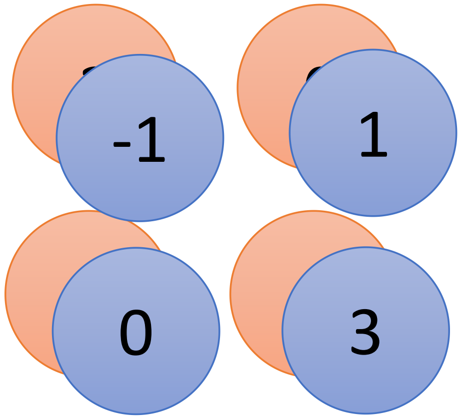
New image
but smaller



2 x 2 image

Each filter
is a channel

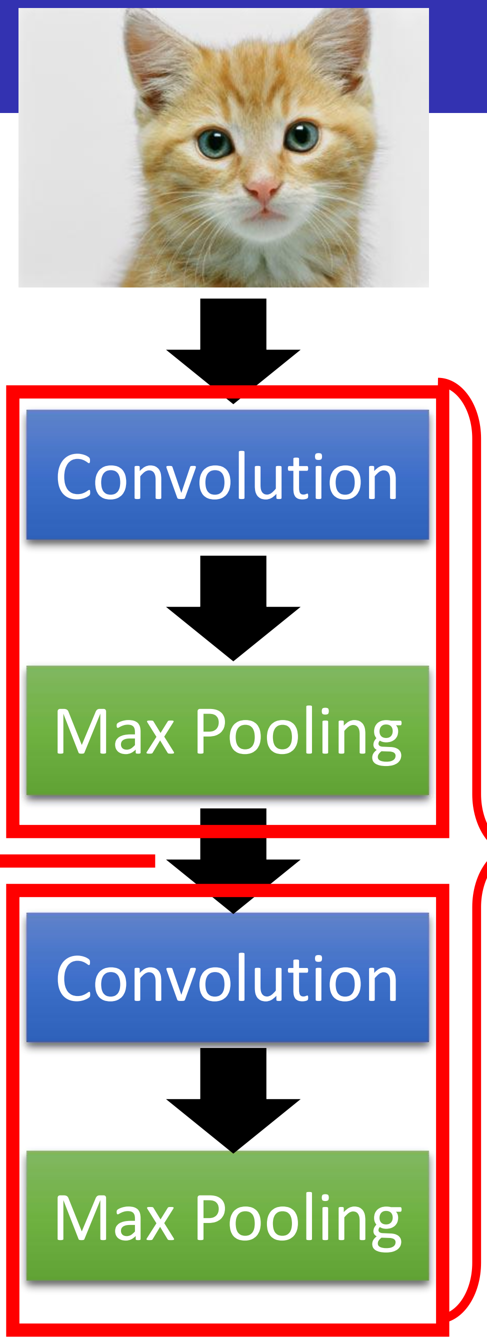
The whole CNN



A new image

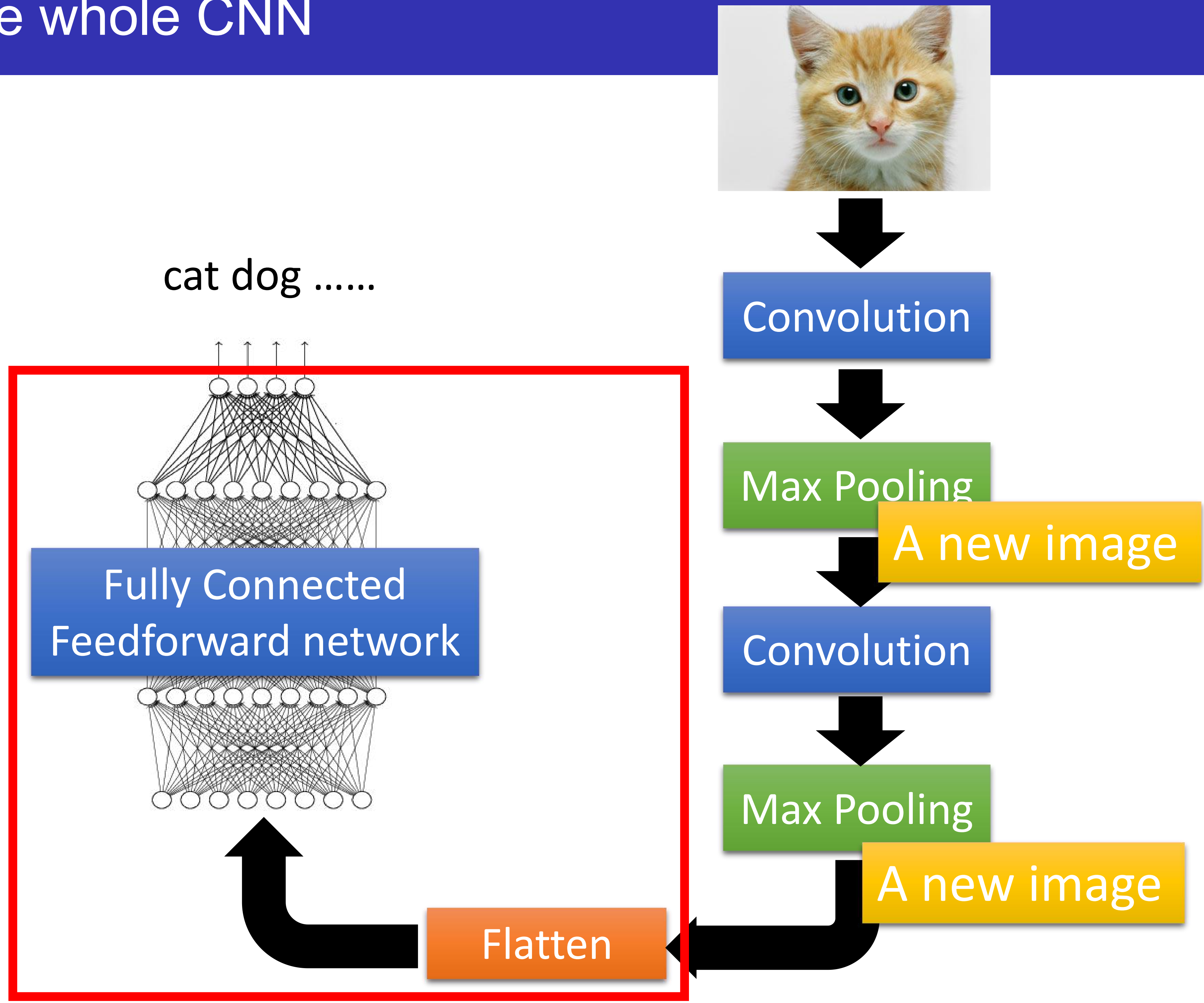
Smaller than the original image

The number of the channel is the number of filters

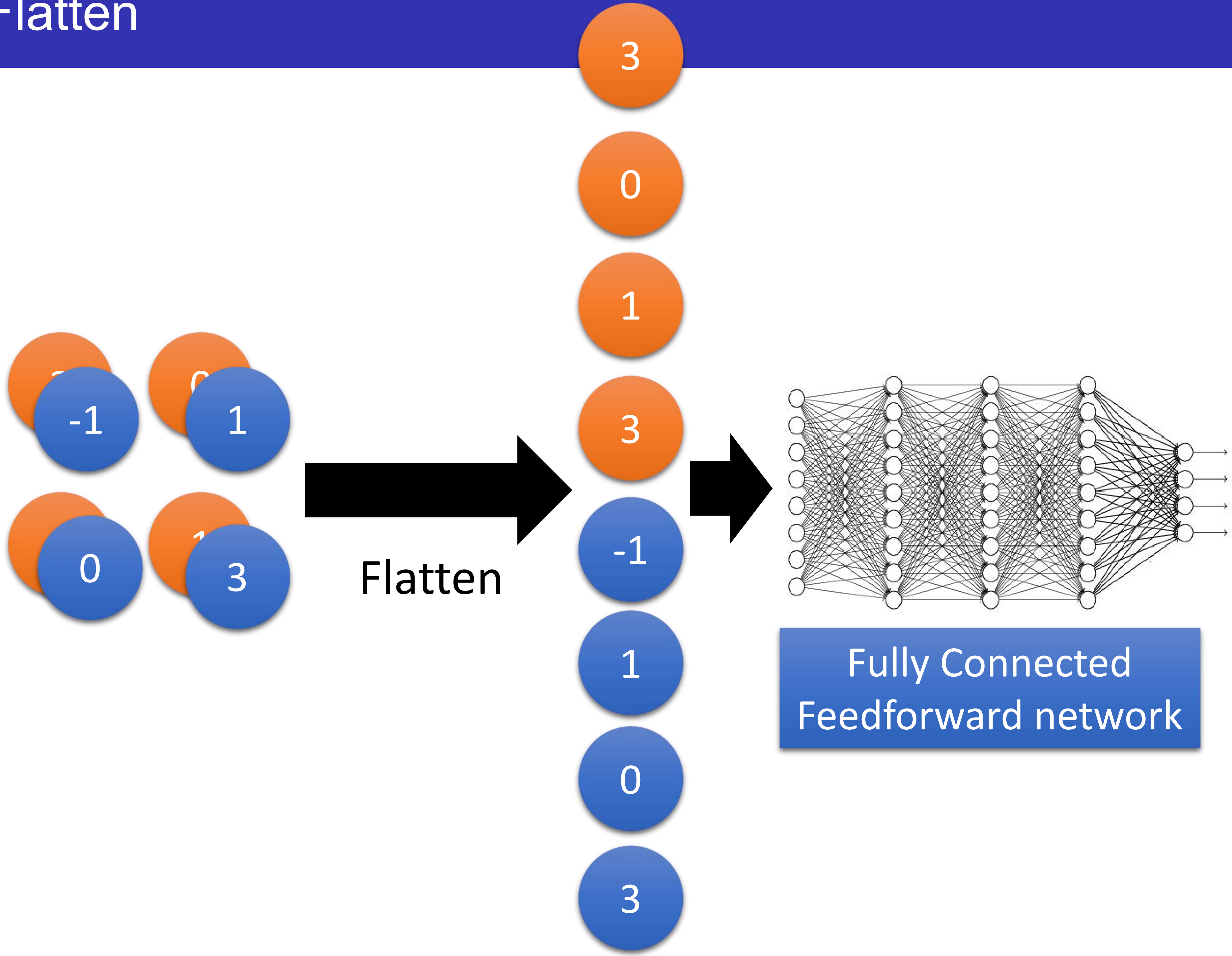


Can repeat many times

The whole CNN



Flatten


-1
1
0
3
Flatten
3
0
1
3
-1
1
0
3
Fully Connected
Feedforward network

Convolution vs. Cross-Correlation Operation

Convolution in 1D (continuous-time)

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

Convolution in 1D (discrete-time)

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

Convolution in 2D (discrete-time)

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

Convolution vs. Cross-Correlation Operation

1	-1	-1
-1	2	-1
-1	-1	3

Filter 1

stride=1

1	0	0	0	0	1
0	2	0	0	1	0
0	0	3	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image

14

$1*1+(-1)*0+(-1)*0+$
$(-1)*0+2*2+(-1)*0+$
$(-1)*0+(-1)*0+3*3$
$= 14$

Cross-Correlation Operation

Convolution vs. Cross-Correlation Operation

1	-1	-1
-1	2	-1
-1	-1	3

Filter 1

stride=1

1	0	0	0	0	1
0	2	0	0	1	0
0	0	3	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image

10

1*3+(-1)*0+(-1)*0+
(-1)*0+2*2+(-1)*0+
(-1)*0+(-1)*0+3*1
= 10

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

Convolution

Convolution vs. Cross-Correlation Operation

Convolution

10

Filter 1

1	-1	-1
-1	2	-1
-1	-1	3

stride=1

1	0	0	0	0	1
0	2	0	0	1	0
0	0	3	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image

Flip the kernel both horizontally and vertically

10

Filter 2

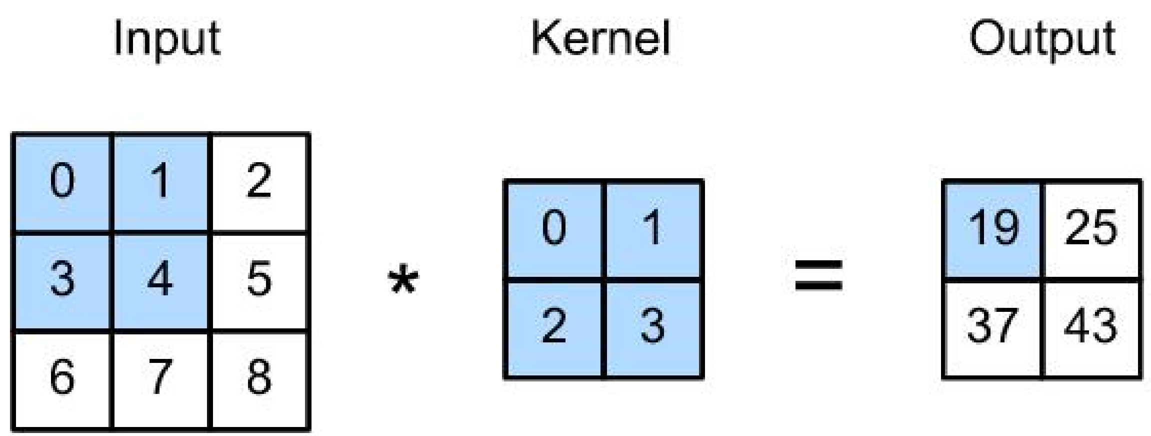
3	-1	-1
-1	2	-1
-1	-1	1

Cross-Correlation Operation

$$1*3+(-1)*0+(-1)*0+$$
$$(-1)*0+2*2+(-1)*0+$$
$$(-1)*0+(-1)*0+3*1$$
$$= 10$$

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.

Cross-Correlation Operation



output computation: $0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19.$

- Along each axis, the output size is slightly smaller than the input size.

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

the input size $n_h \times n_w$

convolution kernel $k_h \times k_w$

Cross-Correlation Operation

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

[2]:
```
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
```

[3]:
```
X = torch.tensor([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
K = torch.tensor([[0.0, 1.0], [2.0, 3.0]])
corr2d(X, K)
```

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

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.

[4]:

```
class Conv2D(nn.Module):
    def __init__(self, kernel_size):
        super().__init__()
        self.weight = nn.Parameter(torch.rand(kernel_size))
        self.bias = nn.Parameter(torch.zeros(1))

    def forward(self, x):
        return corr2d(x, self.weight) + self.bias
```

Object Edge Detection in Images

- First, we construct an "image" of 6×8 pixels.

```
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.

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

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

```
[13]: 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.]])
```

Learning a Kernel

[9]:
```python
# 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.Conv2d(1, 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'batch {i + 1}, loss {l.sum():.3f}')
```

Learning a Kernel

```
batch 2, loss 10.007
batch 4, loss 2.165
batch 6, loss 0.562
batch 8, loss 0.176
batch 10, loss 0.063
```

Note that the error has dropped to a small value after 10 iterations.

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

```
[10]: tensor([[ 1.0099, -0.9607]])
```

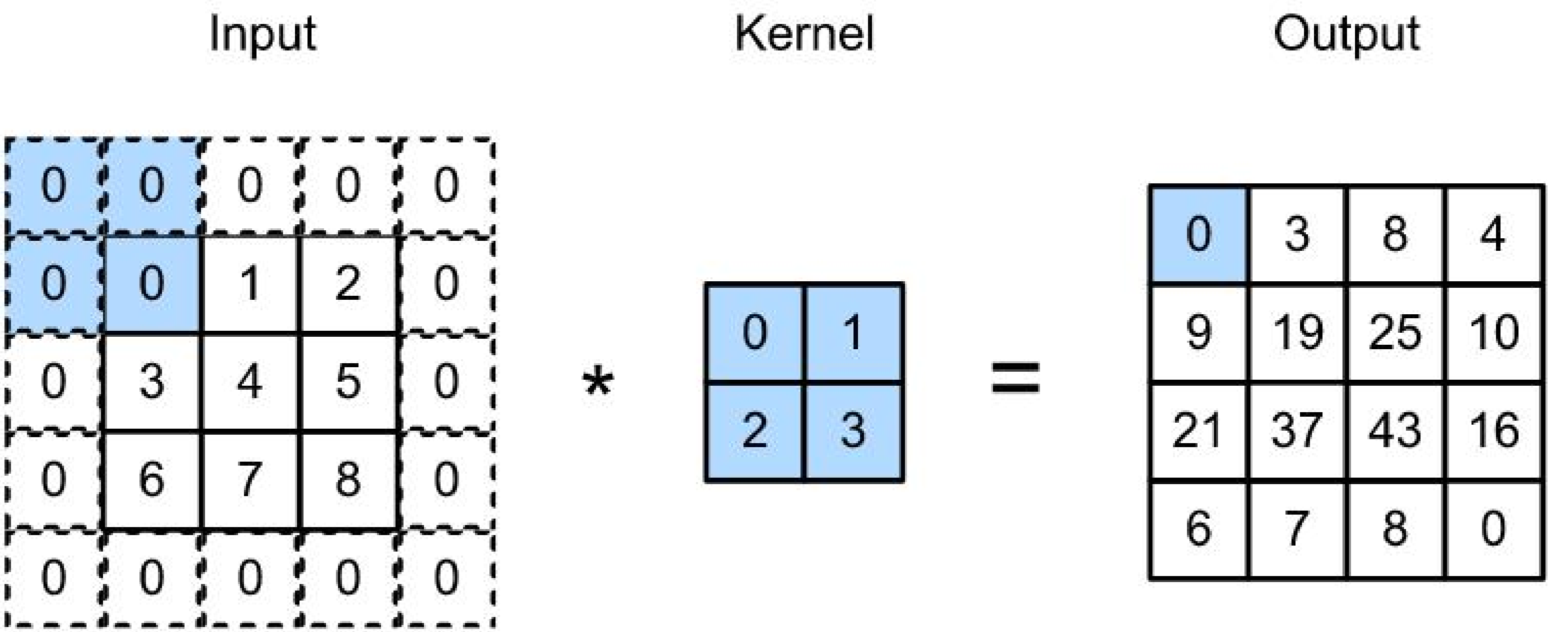
This is the kernel learned

This is the kernel we used

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

More on CNN: Padding

- Padding is the most popular tool for handling the shrinking of size.



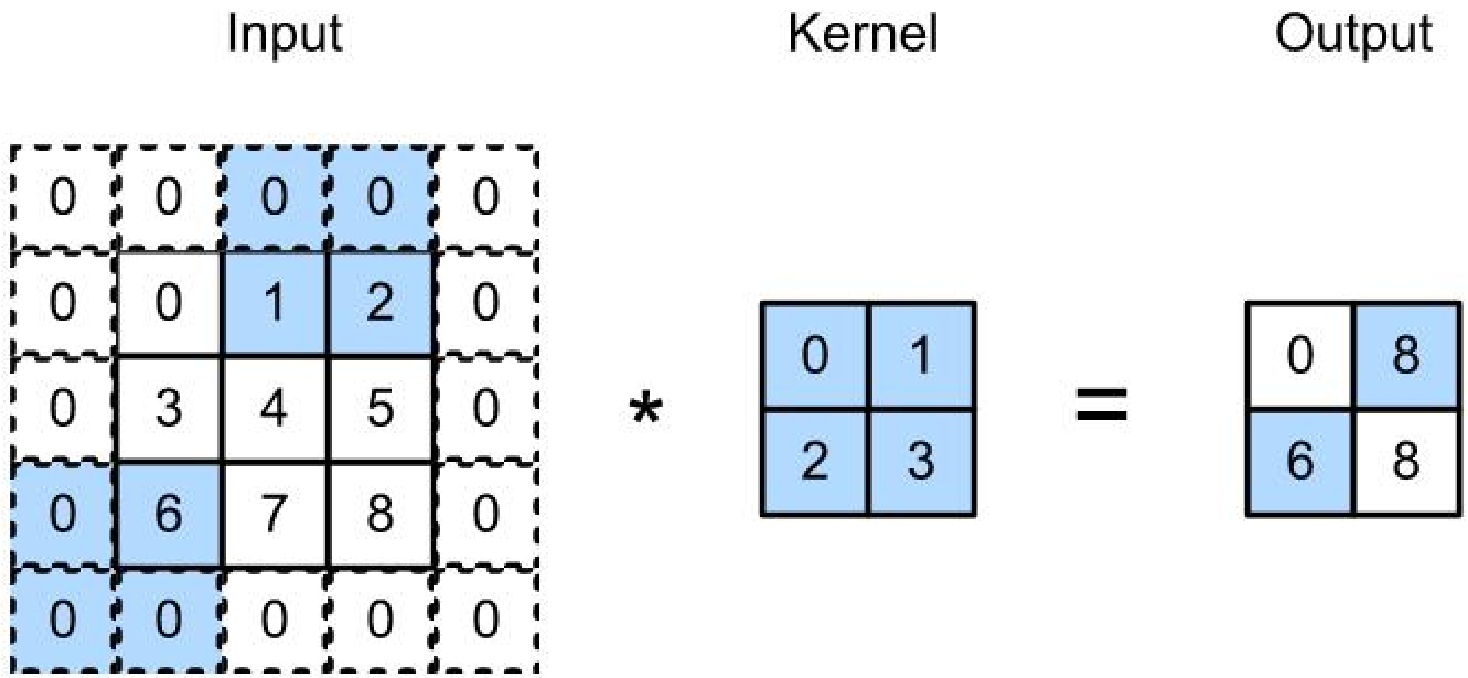
before: $$(n_h - k_h + 1) \times (n_w - k_w + 1)$$

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

More on CNN: 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.

- We refer to the number of rows and columns traversed per slide as **the stride**.

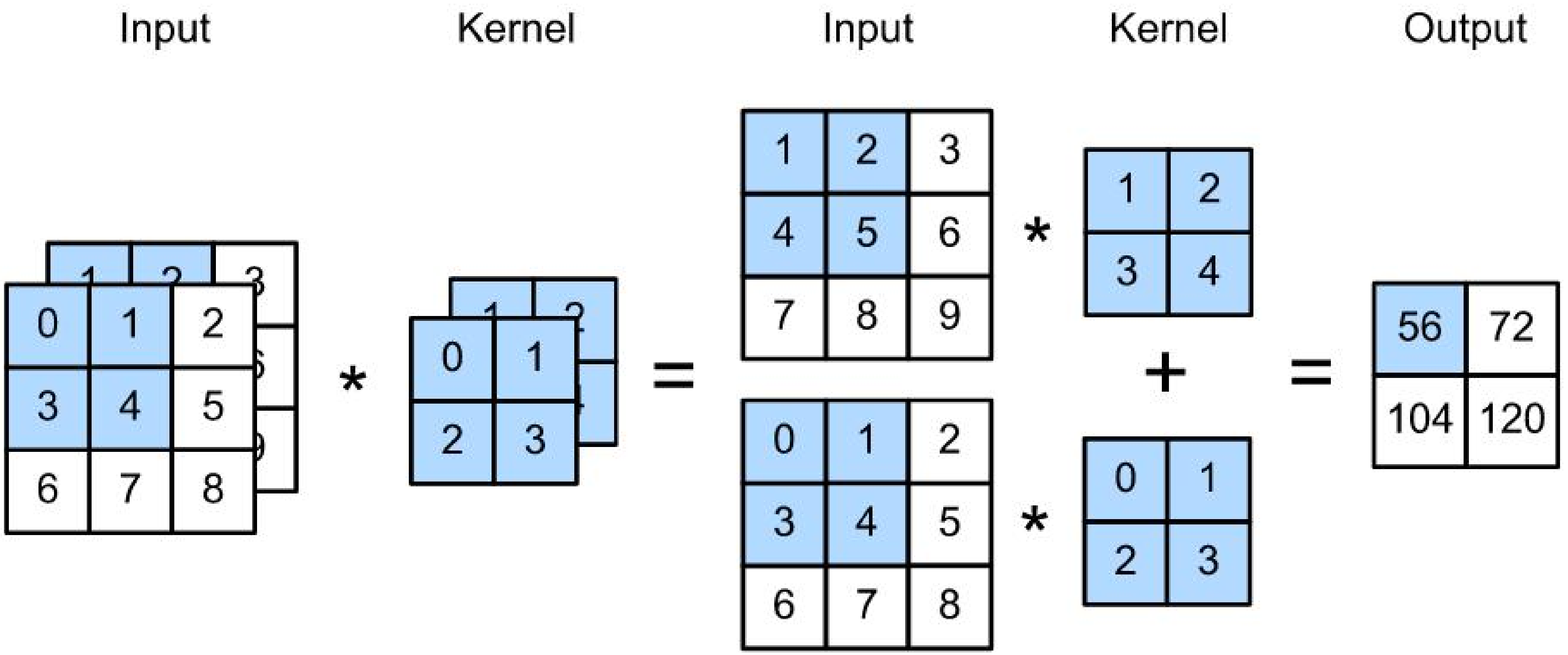


Cross-correlation with strides of 3 and 2 for height and width, respectively.

The output shape:

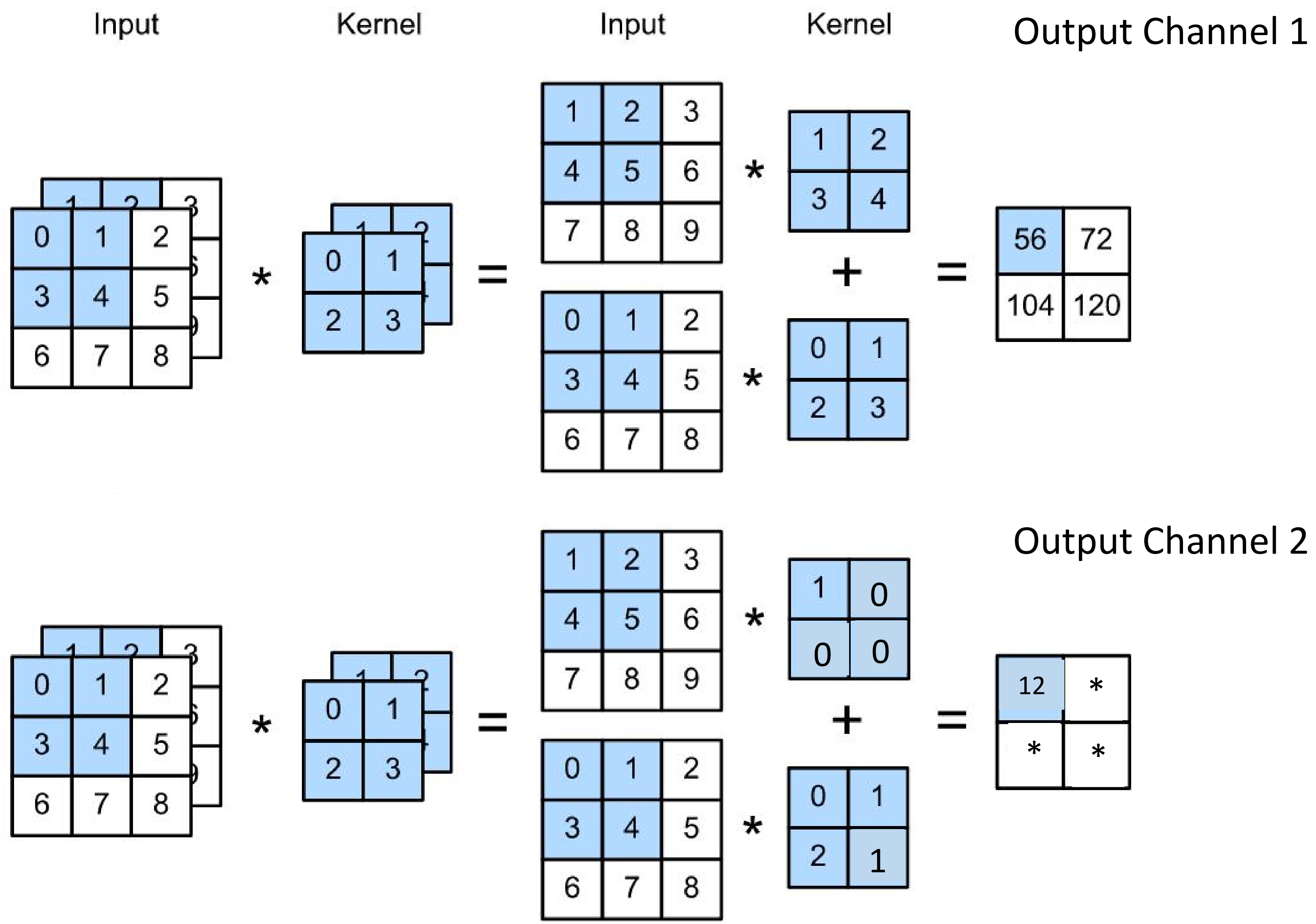
$$\lfloor (n_h - k_h + p_h + s_h)/s_h \rfloor \times \lfloor (n_w - k_w + p_w + s_w)/s_w \rfloor .$$

Multiple Input Channels



Cross-correlation computation with 2 input channels

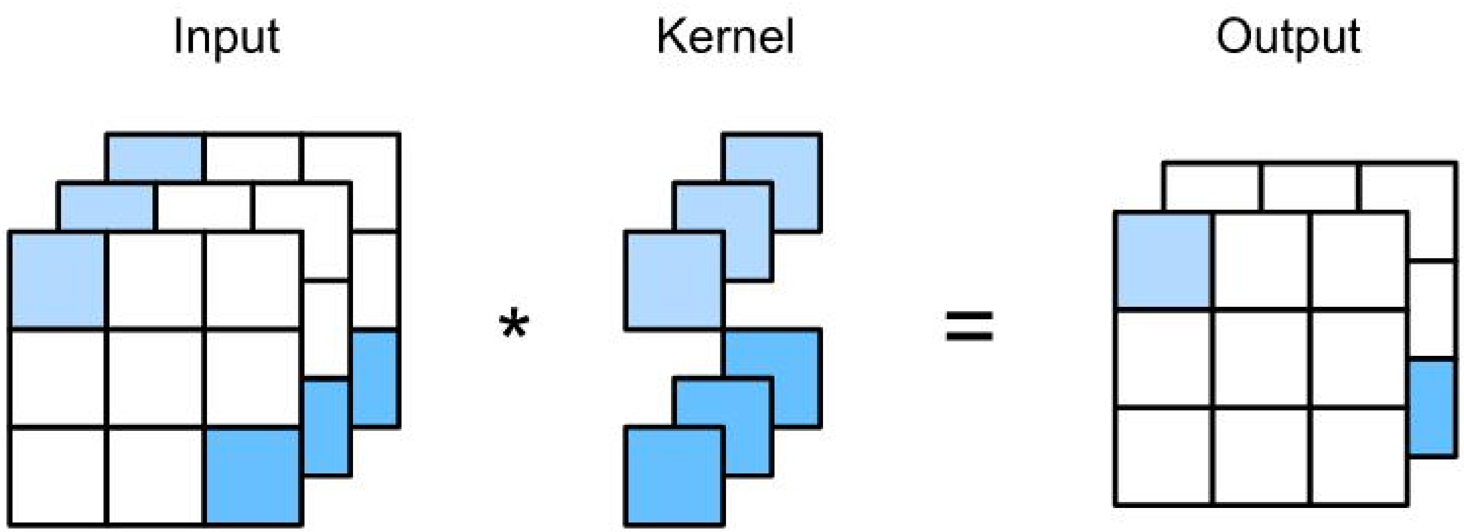
Multiple Input and Multiple Output Channels



the convolution kernel is $c_o \times c_i \times k_h \times k_w$.

1×1 Convolutional Layer

- At first, a 1×1 convolution, i.e., $k_h = k_w = 1$, does not seem to make much sense.
- Nonetheless, they are popular operations that are sometimes included in the designs of complex deep networks.

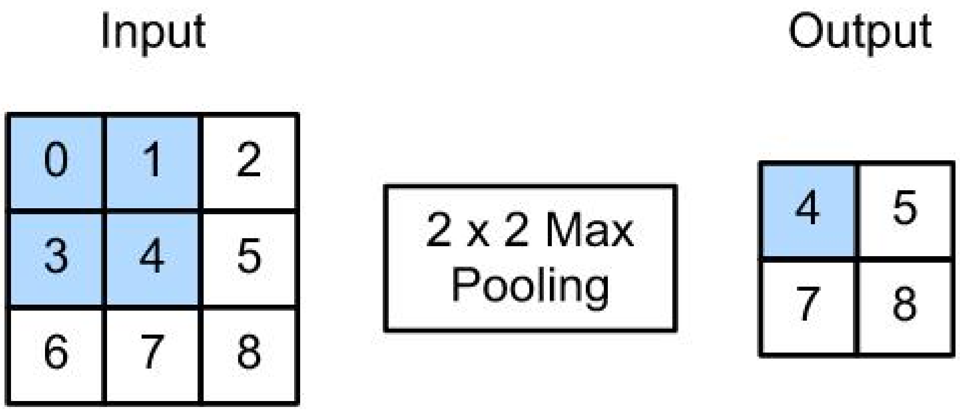


- The cross-correlation computation uses the 1×1 convolution kernel with 3 input channels and 2 output channels.
- The only computation of the 1×1 convolution occurs on the channel dimension.

Pooling 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 (sometimes known as the **pooling window**).

- Maximum Pooling and Average Pooling




```python
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
```

Pooling layers

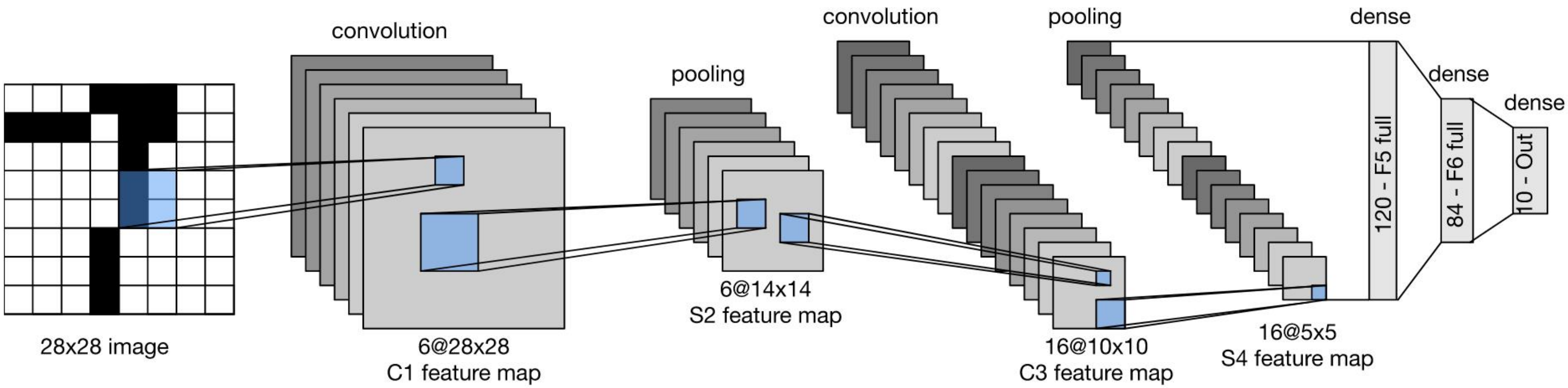
- As with convolutional layers, pooling layers can also change the output shape via padding and stride

- 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.

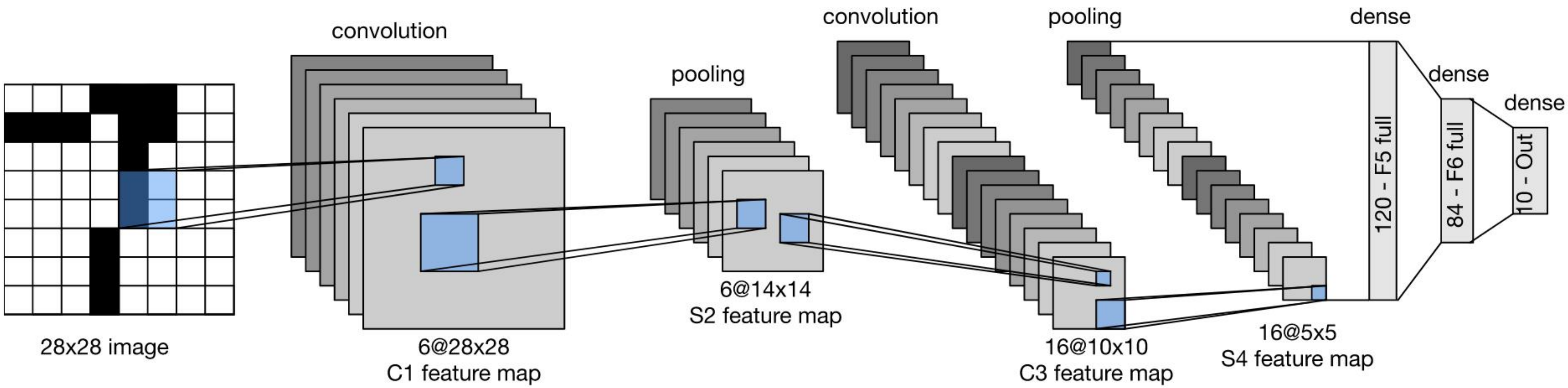
Convolutional Neural Networks (LeNet)

- The model was introduced by (and named for) Yann LeCun, for the purpose of recognizing handwritten digits.
- LeNet (LeNet-5) consists of two parts:
  - (i) a convolutional encoder consisting of two convolutional layers;
  - (ii) 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.
- LeNet's dense block has three fully-connected layers, with 120, 84, and 10 outputs, respectively.

Convolutional Neural Networks (LeNet)




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

net = nn.Sequential(nn.Conv2d(1, 6, kernel_size=5, padding=2), nn.Sigmoid(),
                    nn.AvgPool2d(kernel_size=2, stride=2),
                    nn.Conv2d(6, 16, kernel_size=5), nn.Sigmoid(),
                    nn.AvgPool2d(kernel_size=2, stride=2), nn.Flatten(),
                    nn.Linear(16 * 5 * 5, 120), nn.Sigmoid(),
                    nn.Linear(120, 84), nn.Sigmoid(), nn.Linear(84, 10))
```

Modern Convolutional Neural Networks

Now that we understand the basics of wiring together CNNs,

I will take you through a tour of modern CNN architectures.

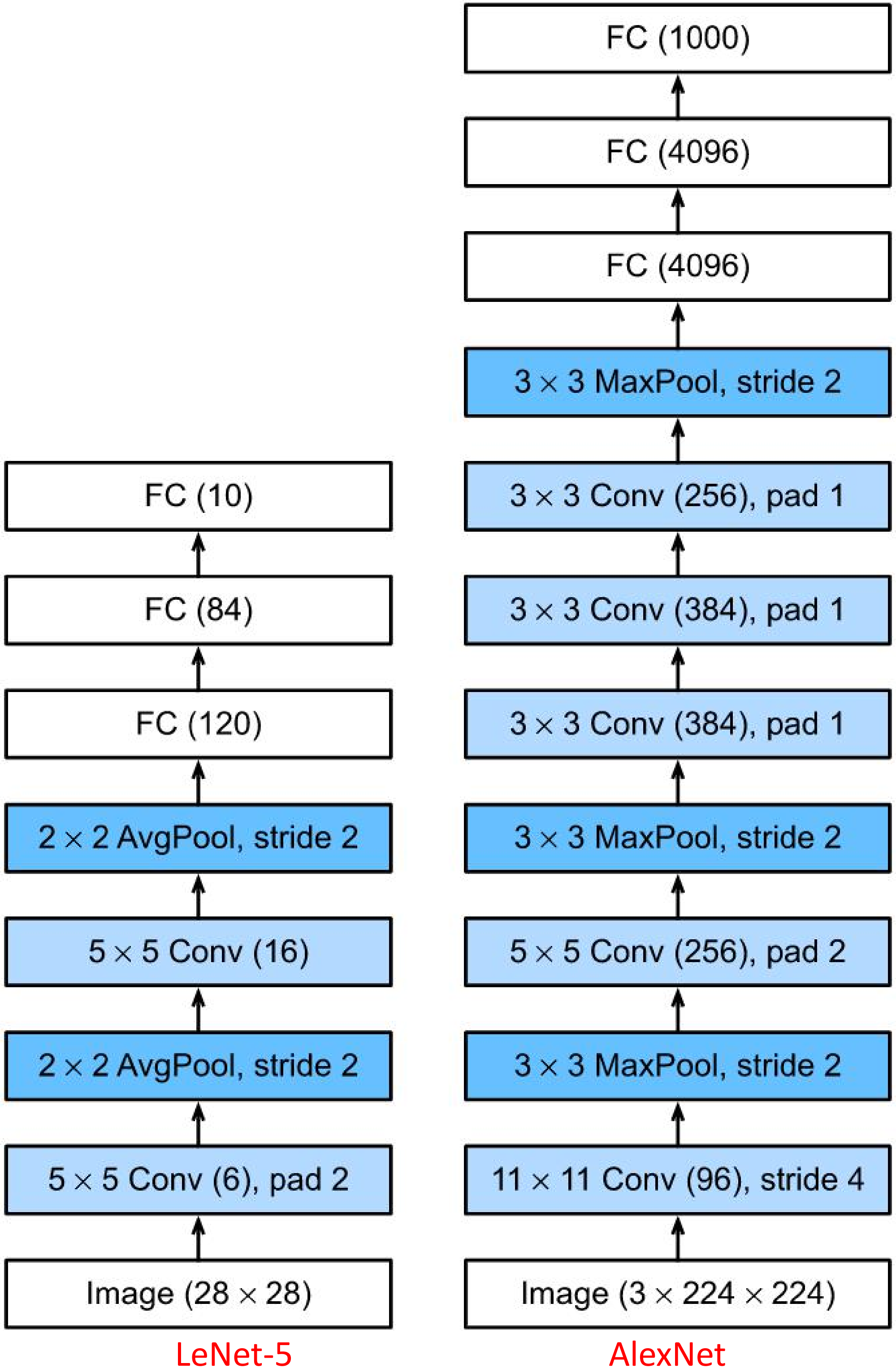
AlexNet

- AlexNet, that achieved excellent performance in the 2012 ImageNet challenge.

- The design philosophies of AlexNet and LeNet are very similar, but there are also significant differences.

- First, AlexNet is much deeper than LeNet5. AlexNet consists of eight layers:
  - ✓ five convolutional layers,
  - ✓ two fully-connected hidden layers,
  - ✓ one fully-connected output layer.

- Second, AlexNet used the ReLU instead of sigmoid as its activation function.
  - ✓ The computation of ReLU is simpler.
  - ✓ ReLU makes model training easier.



AlexNet

```python
net = nn.Sequential(
    # Here, we use a larger 11 x 11 window to capture objects. At the same
    # time, we use a stride of 4 to greatly reduce the height and width of the
    # output. Here, the number of output channels is much larger than that in
    # LeNet
    nn.Conv2d(1, 96, kernel_size=11, stride=4, padding=1), nn.ReLU(),
    nn.MaxPool2d(kernel_size=3, stride=2),
    # Make the convolution window smaller, set padding to 2 for consistent
    # height and width across the input and output, and increase the number of
    # output channels
    nn.Conv2d(96, 256, kernel_size=5, padding=2), nn.ReLU(),
    nn.MaxPool2d(kernel_size=3, stride=2),
    # Use three successive convolutional layers and a smaller convolution
    # window. Except for the final convolutional layer, the number of output
    # channels is further increased. Pooling layers are not used to reduce the
    # height and width of input after the first two convolutional layers
    nn.Conv2d(256, 384, kernel_size=3, padding=1), nn.ReLU(),
    nn.Conv2d(384, 384, kernel_size=3, padding=1), nn.ReLU(),
    nn.Conv2d(384, 256, kernel_size=3, padding=1), nn.ReLU(),
    nn.MaxPool2d(kernel_size=3, stride=2), nn.Flatten(),
    # Here, the number of outputs of the fully-connected layer is several
    # times larger than that in LeNet. Use the dropout layer to mitigate
    # overfitting
    nn.Linear(6400, 4096), nn.ReLU(), nn.Dropout(p=0.5),
    nn.Linear(4096, 4096), nn.ReLU(), nn.Dropout(p=0.5),
    # Output layer. Since we are using Fashion-MNIST, the number of classes is
    # 10, instead of 1000 as in the paper
    nn.Linear(4096, 10))
```

Networks Using Blocks (VGG)

- The design of NN 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.

- The idea of using blocks first emerged from the Visual Geometry Group (VGG) at Oxford University, named **VGG network**.

- The basic building block of classic CNNs is a sequence of the following:
  (i) a convolutional layer with padding to maintain the resolution,
  (ii) a nonlinearity such as a ReLU,
  (iii) a pooling layer such as a maximum pooling layer.

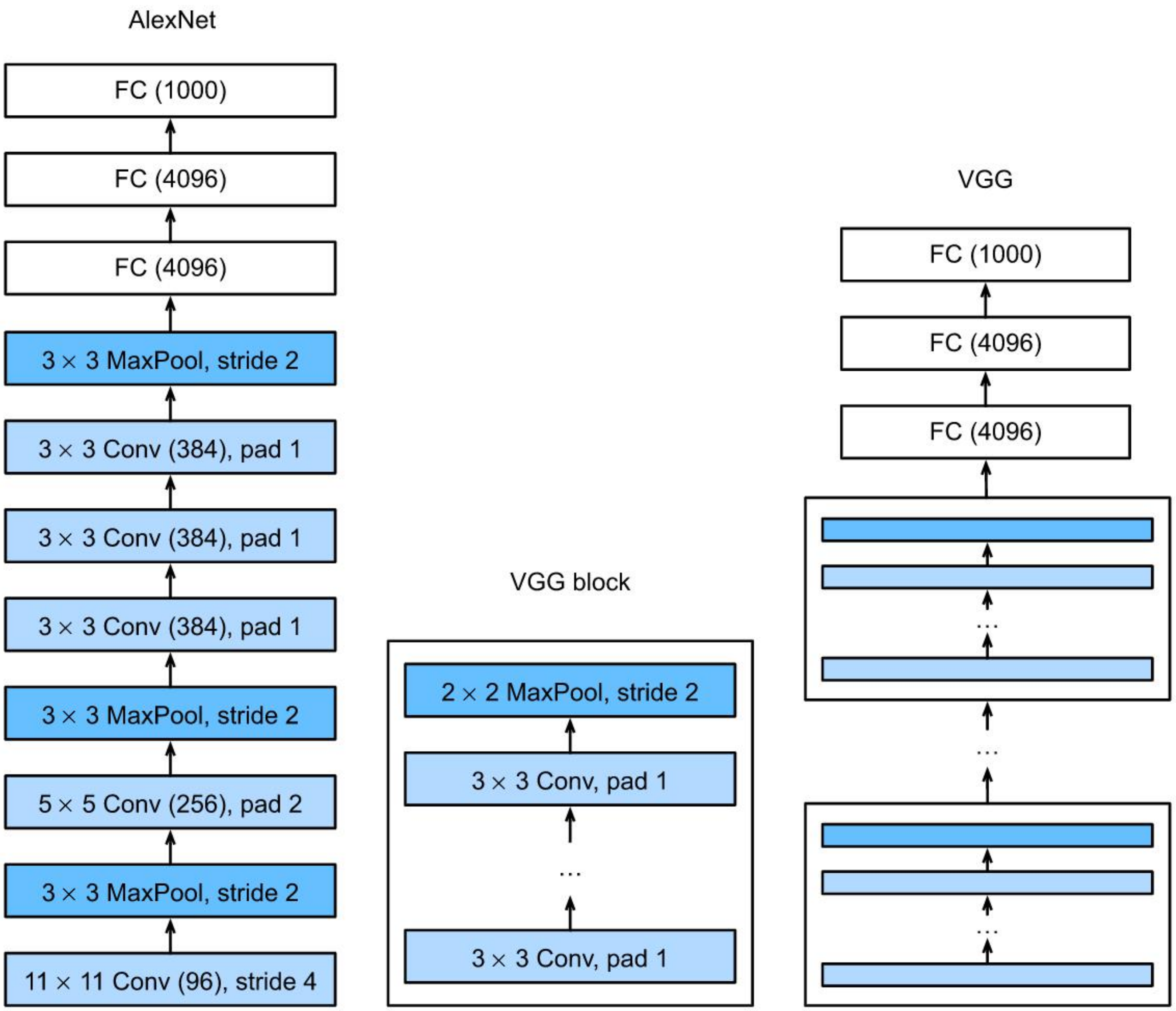
- One VGG block consists of a sequence of convolutional layers, followed by a maximum pooling layer for spatial downsampling.

Networks Using Blocks (VGG)

- One VGG block consists of a sequence of convolutional layers, followed by a maximum pooling layer for spatial downsampling.

```
def vgg_block(num_convs, in_channels, out_channels):
    layers = []
    for _ in range(num_convs):
        layers.append(
            nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1))
        layers.append(nn.ReLU())
        in_channels = out_channels
    layers.append(nn.MaxPool2d(kernel_size=2, stride=2))
    return nn.Sequential(*layers)
```

VGG Network



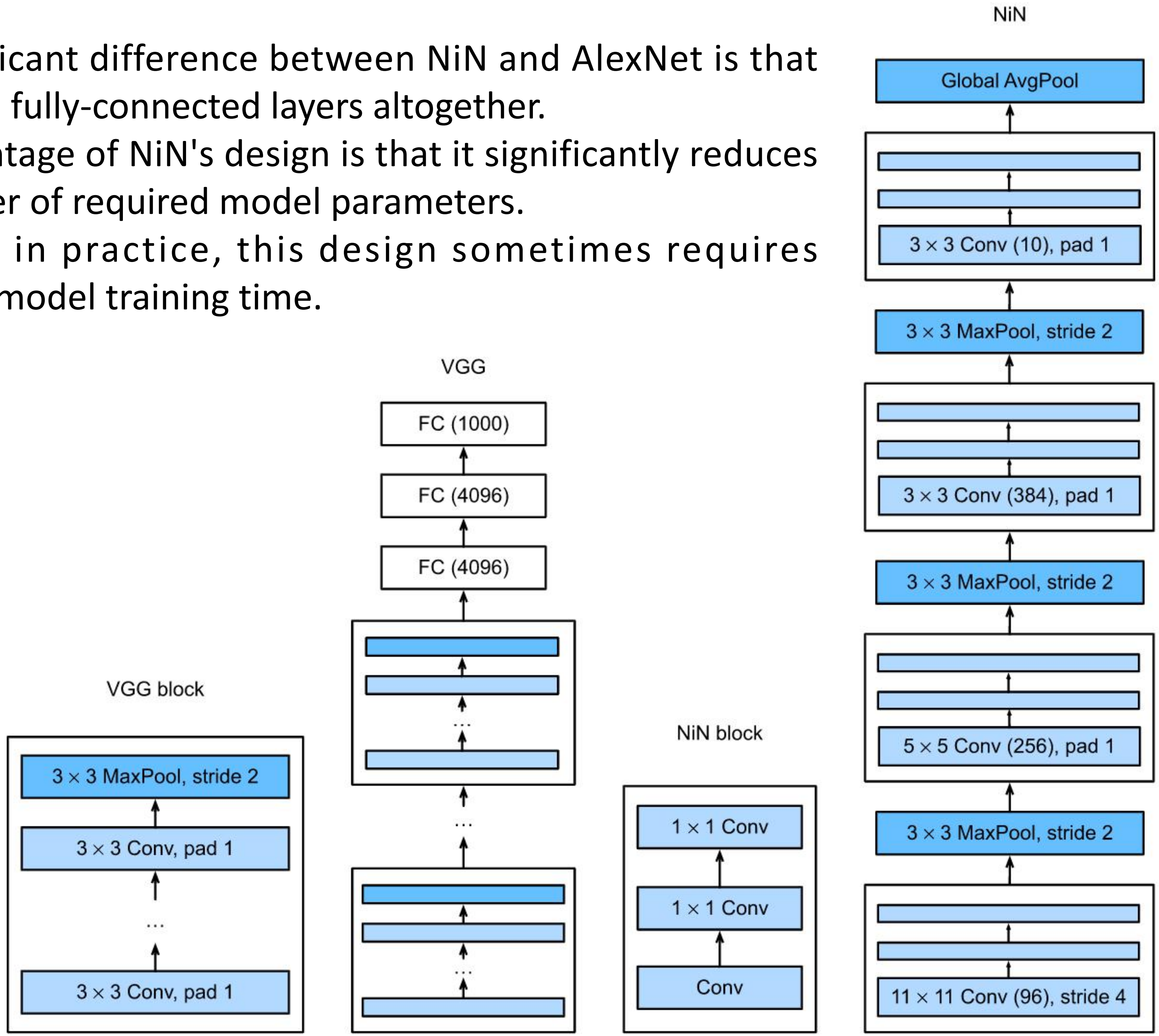
AlexNet
FC (1000)
FC (4096)
FC (4096)
3 × 3 MaxPool, stride 2
3 × 3 Conv (384), pad 1
3 × 3 Conv (384), pad 1
3 × 3 Conv (384), pad 1
3 × 3 MaxPool, stride 2
5 × 5 Conv (256), pad 2
3 × 3 MaxPool, stride 2
11 × 11 Conv (96), stride 4
VGG block
2 × 2 MaxPool, stride 2
3 × 3 Conv, pad 1
...
3 × 3 Conv, pad 1
VGG
FC (1000)
FC (4096)
FC (4096)
...

Network in Network (NiN)

- LeNet, AlexNet, and VGG all share a common design pattern:
  - ✓ Extract features exploiting spatial structure via a sequence of convolution and pooling layers and then post-process the representations via fully-connected layers.
  - ✓ The improvements upon LeNet by AlexNet and VGG mainly lie in how these later networks widen and deepen these two modules.
- Alternatively, one could imagine using fully-connected layers earlier in the process.
  - ✓ Network in network (NiN) blocks offer an alternative.
  - ✓ The NiN block consists of one convolutional layer followed by two 1×1 convolutional layers that act as per-pixel fully-connected layers with ReLU activations.

NiN

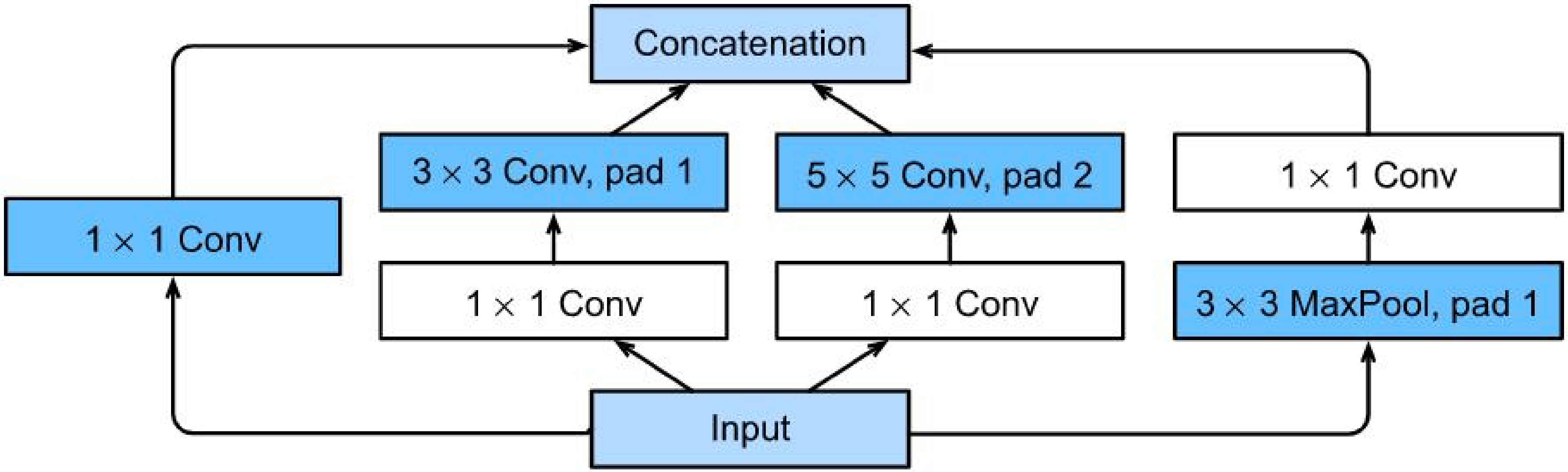
- One significant difference between NiN and AlexNet is that NiN avoids fully-connected layers altogether.
- One advantage of NiN's design is that it significantly reduces the number of required model parameters.
- However, in practice, this design sometimes requires increased model training time.



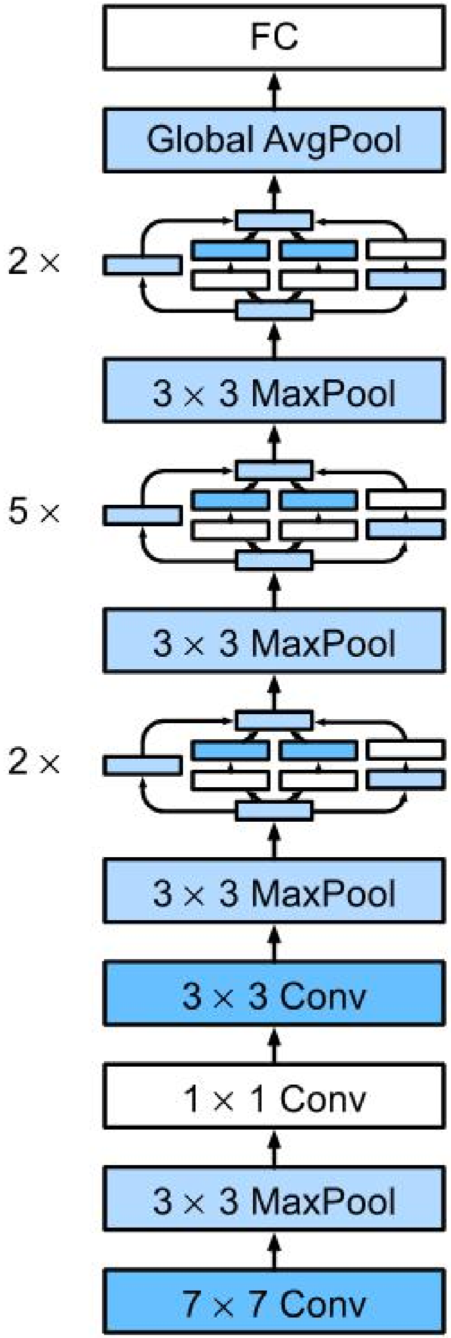
Networks with Parallel Concatenations (GoogLeNet)

- In 2014, GoogLeNet won the ImageNet Challenge, proposing a structure that combined the strengths of NiN and paradigms of repeated blocks

- The basic convolutional block in GoogLeNet is called an Inception block

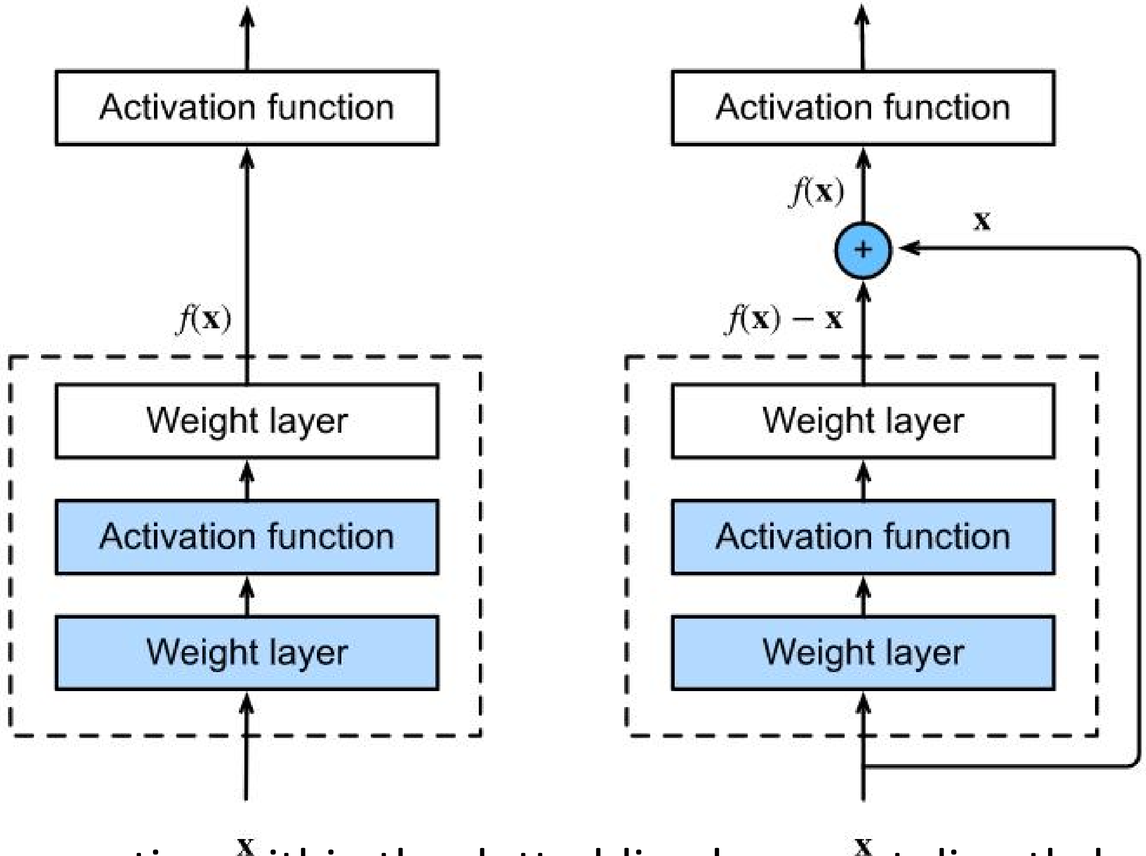


GoogLeNet Model



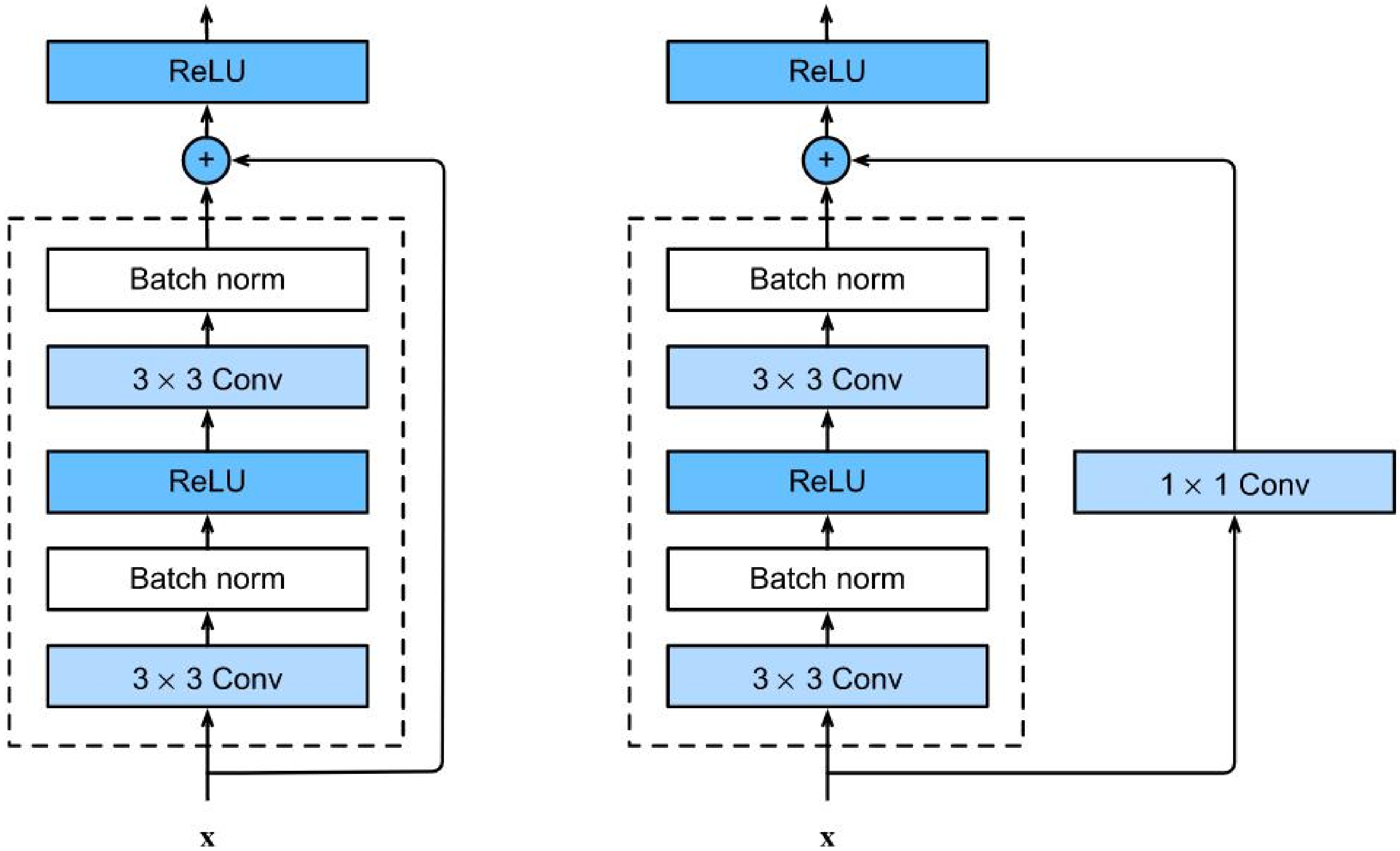
- GoogLeNet uses a stack of a total of 9 inception blocks and global average pooling to generate its estimates.
- Maximum pooling between inception blocks reduces the dimensionality.
- The first module is similar to AlexNet and LeNet.
- The stack of blocks is inherited from VGG and the global average pooling avoids a stack of fully-connected layers at the end.

Residual Networks (ResNet)



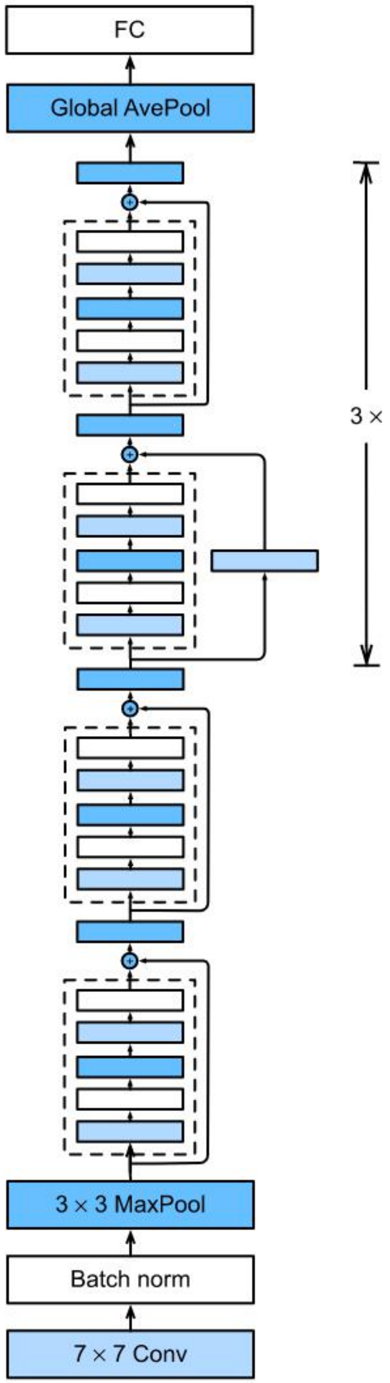
- On the left, the portion within the dotted-line box must directly learn the mapping $f(x)$.
- On the right, needs to learn the residual mapping $f(x)-x$, which is how the residual block derives its name.
- If the identity mapping $f(x)=x$ is the desired underlying mapping, the residual mapping is easier to learn.
- The right figure illustrates the residual block of ResNet, where the solid line carrying the layer input x to the addition operator is called a residual connection.
- With residual blocks, inputs can forward propagate faster through the residual connections across layers.

Residual Networks (ResNet)

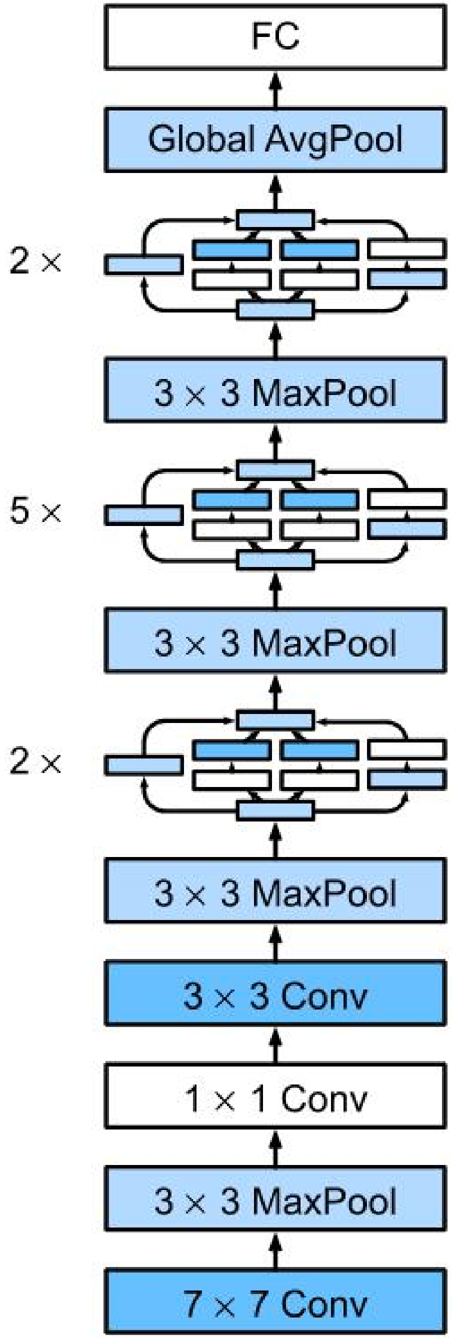


- ResNet follows VGG's full 3×3 convolutional layer design.
- The residual block has two 3×3 convolutional layers with the same number of output channels.
- Each convolutional layer is followed by a batch normalization layer and a ReLU activation function.

Residual Networks (ResNet)

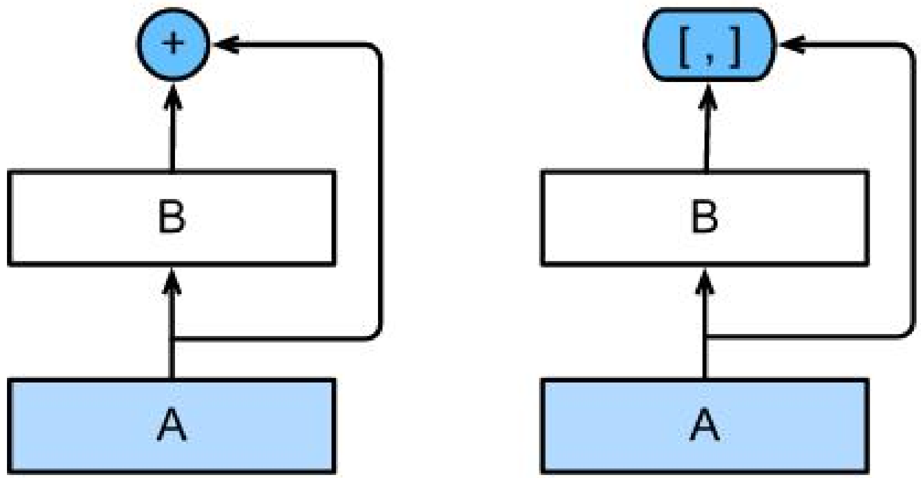


- The first two layers of ResNet are the same as those of the GoogLeNet
- The difference is the batch normalization layer added after each convolutional layer in ResNet.
- GoogLeNet uses four modules made up of Inception blocks. However, ResNet uses four modules made up of residual blocks



Densely Connected Networks (DenseNet)

- ResNet decomposes functions into $f(\mathbf{x}) = \mathbf{x} + g(\mathbf{x}).$

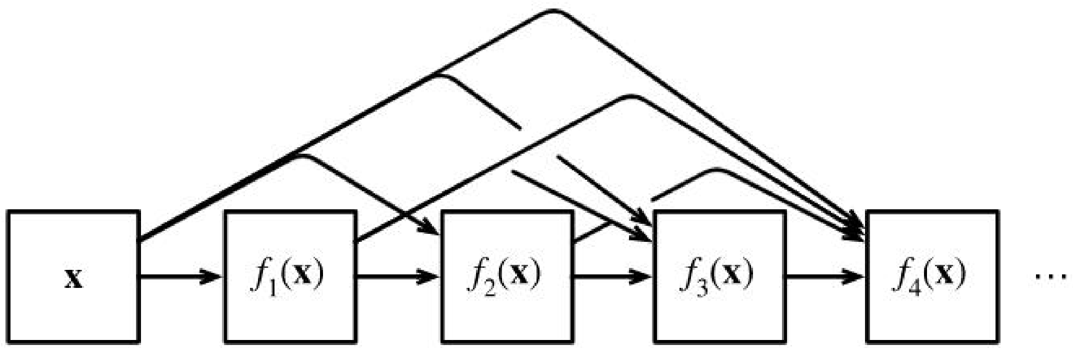


The main difference between ResNet (left) and DenseNet (right) in cross-layer connections: use of addition and use of concatenation.

$$\mathbf{x} \to \left[\mathbf{x}, f_1(\mathbf{x}), f_2([\mathbf{x}, f_1(\mathbf{x})]), f_3([\mathbf{x}, f_1(\mathbf{x}), f_2([\mathbf{x}, f_1(\mathbf{x})])]), \ldots\right].$$

In the end, all these functions are combined in MLP to reduce the number of features again.

Densely Connected Networks (DenseNet)



- The name DenseNet arises from the fact that the dependency graph between variables becomes quite dense. The last layer of such a chain is densely connected to all previous layers.

- The main components that compose a DenseNet are dense blocks and transition layers:
  - ✓ A dense block consists of multiple convolution blocks, each using the same number of output channels. In the forward propagation, however, we concatenate the input and output of each convolution block on the channel dimension.
  - ✓ A transition layer is used to control the complexity of the model. Since each dense block will increase the number of channels, adding too many of them will lead to an excessively complex model.