T05460301-01

Deep Learning (Fall 2023)

Ikbeom Jang

ijang@hufs.ac.kr

CES HUFS

Convolutional Neural Networks

- Padding and Stride
- Multiple Input and Multiple Output Channels
- Pooling
- CNN (LeNet)
- Modern CNNs

Padding and Stride

Padding and Stride

- In Fig. 7.2.1, the input is 3x3 and the convolution kernel is 2x2, yielding an output representation with dimension 2×2.
- Assuming that the input shape is $n_h \times n_w$ and the convolution kernel shape is $k_h \times k_w$, the output shape will be $(n_h - k_h + 1) \times (n_w - k_w + 1)$
- We can only shift the convolution kernel so far until it runs out of pixels to apply the convolution to.
- **Padding** and **strided convolutions** offer more control over the size of the output.
- As motivation, note that since kernels generally have width and height greater than 1, **after applying many successive convolutions**, we tend to wind up with **outputs that are considerably smaller than our input**.

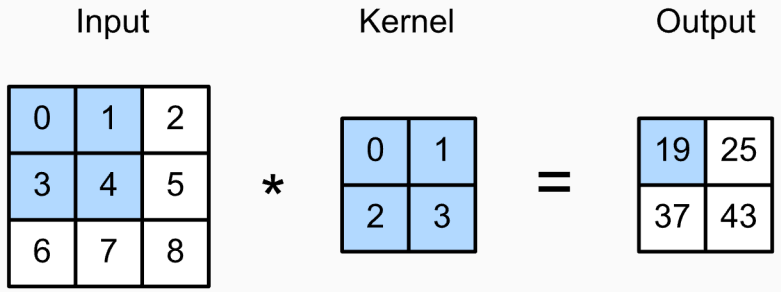


Fig. 7.2.1 Two-dimensional cross-correlation operation. The shaded portions are the first output element as well as the input and kernel tensor elements used for the output computation:

$$0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$$

Padding and Stride

- If we start with a 240×240 pixel image, ten layers of 5×5 convolutions **reduce the image** to 200×200 pixels, slicing off 30% of the image.
  - It obliterates any **interesting information on the boundaries** of the original image.
- **Padding** is the most popular tool for handling this issue.

- In other cases, we may want to reduce the dimensionality drastically, e.g., if we find the original input resolution to be unwieldy.
- **Strided convolutions** are a popular technique that can help in these instances.

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

* unwieldy: 버거운, 통제하기 힘든

Padding

- Consider Fig. 7.3.1 that depicts the pixel utilization as a function of the convolution kernel size and the position within the image. The pixels in the corners are hardly used at all.
- Since we typically use small kernels, for any given convolution we might only lose a few pixels but this can add up as we apply many successive convolutional layers.
- 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**.

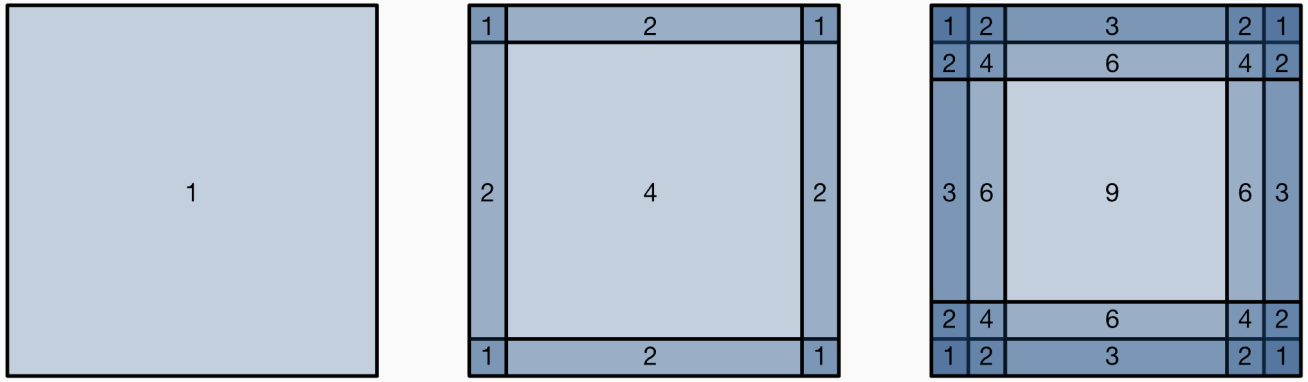


Fig. 7.3.1 Pixel utilization for convolutions of size 1×1, 2×2, and 3×3 respectively.

Padding

- Typically, we **set** the values of the **extra pixels to zero**.
- In Fig. 7.3.2, we pad a 3×3 input, increasing its size to 5×5.
- The corresponding output then increases to a 4×4 matrix.
- The shaded portions are the first output element as well as the input and kernel tensor elements used for the output computation: 0×0+0×1+0×2+0×3=0.

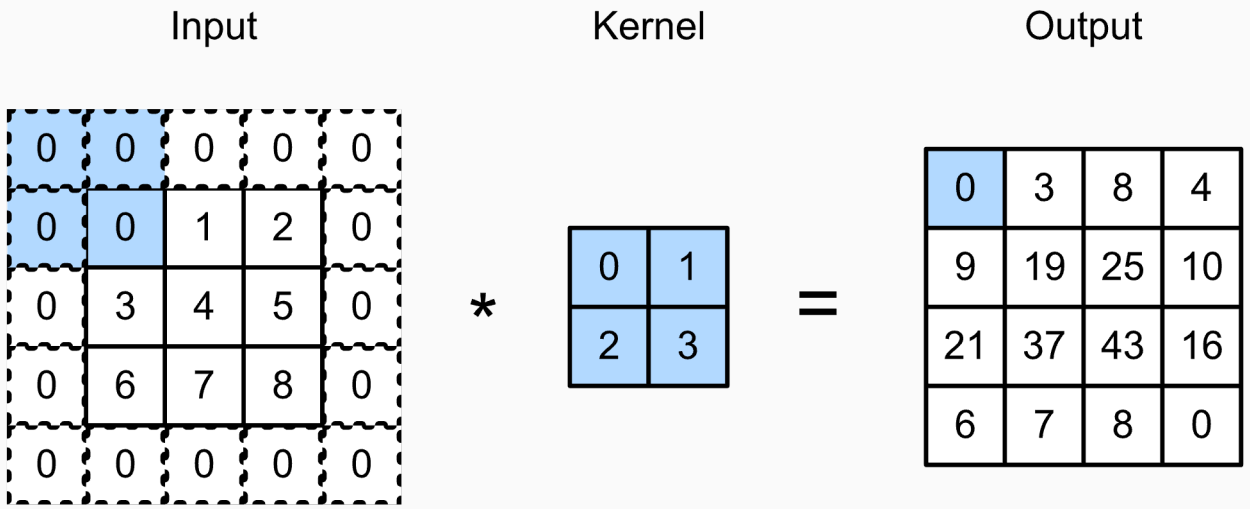


Fig. 7.3.2 Two-dimensional cross-correlation with padding.

Padding

- 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 **$(n_h-k_h+p_h+1)\times(n_w-k_w+p_w+1)$**.
  - This means that the height and width of the output will increase by p_h and p_w, respectively.

- In many cases, **we set $p_h=k_h-1$ and $p_w=k_w-1$** to give the **input and output the same height and width**.
  - This will make it easier to predict the output shape of each layer when constructing the network.

- Assuming that k_h is odd here, we will pad $p_h/2$ rows on both sides of the height.
- If k_h is even, one possibility is to pad $\lceil p_h/2 \rceil$ rows on the top of the input and $\lfloor p_h/2 \rfloor$ rows on the bottom.
- We will pad both sides of the width in the same way.

- CNNs commonly use convolution **kernels with odd height and width values**, such as **1, 3, 5, or 7**.
- Choosing odd kernel sizes has the benefit that we can **preserve the dimensionality** while padding with the same number of rows on top and bottom, and the same number of columns on left and right.
- Another benefit is that we know that the output **Y[i, j]** is **calculated by cross-correlation of the input and convolution kernel with the window centered on X[i, j]**.

Padding

- In the following example, we create a two-dimensional convolutional layer with a height and width of 3 and apply 1 pixel of padding on all sides.
- Given an input with a height and width of 8, we find that the height and width of the output is also 8.

```
# 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

- When the height and width of the convolution kernel are different, we can make the output and input have the same height and width by setting different padding numbers for height and width.

```
# 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])
```

Stride

- When computing the cross-correlation, we start with the convolution window at the upper-left corner of the input tensor, and then slide it over all locations both down and to the right.
- In the previous examples, we defaulted to **sliding one element at a time**.

- However, 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.

- We refer to the number of rows and columns traversed per slide as **stride**.
- So far, we have used strides of 1, both for height and width. Sometimes, we may want to use a larger stride.

Stride

- Fig. 7.3.3 shows a 2D cross-correlation operation with a **stride of 3 vertically and 2 horizontally**.

- The shaded portions are the **output**, **input** and **kernel** tensor elements used for the output computation:
  - 0×0+0×1+1×2+2×3=8
  - 0×0+6×1+0×2+0×3=6

- The convolution window **slides two columns to the right**, and then it **slides down three rows**.
  - When the convolution window continues to slide two columns to the right on the input, there is no output because the input element cannot fill the window (unless we add another column of padding).

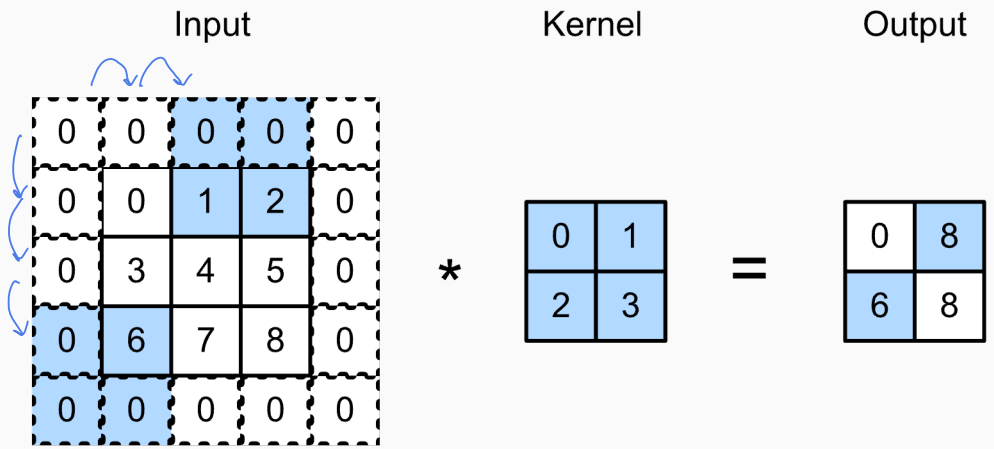


Fig. 7.3.3 Cross-correlation with strides of 3 and 2 for height and width, respectively.

Stride

In general, when the stride for the height is s_h and the stride for the width is s_w, the output shape is

$$\lfloor (n_\mathrm{h} - k_\mathrm{h} + p_\mathrm{h} + s_\mathrm{h})/s_\mathrm{h} \rfloor \times \lfloor (n_\mathrm{w} - k_\mathrm{w} + p_\mathrm{w} + s_\mathrm{w})/s_\mathrm{w} \rfloor. \tag{7.3.2}$$

If we set $p_\mathrm{h} = k_\mathrm{h} - 1$ and $p_\mathrm{w} = k_\mathrm{w} - 1$, then the output shape can be simplified to $\lfloor (n_\mathrm{h} + s_\mathrm{h} - 1)/s_\mathrm{h} \rfloor \times \lfloor (n_\mathrm{w} + s_\mathrm{w} - 1)/s_\mathrm{w} \rfloor$. Going a step further, if the input height and width are divisible by the strides on the height and width, then the output shape will be $(n_\mathrm{h}/s_\mathrm{h}) \times (n_\mathrm{w}/s_\mathrm{w})$.

Below, we set the strides on both the height and width to 2, thus halving the input height and width.

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

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

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

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

Multiple Input and Multiple Output Channels

Multiple Input and Multiple Output Channels

- We described the multiple channels that comprise each image and convolutional layers for multiple channels.
  - e.g., Color images have the standard RGB channels to indicate the amount of Red, Green and Blue
- 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 $3 \times h \times w$.
- We refer to this axis (a size of 3) as the **channel dimension**.

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

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.
- Assuming that the number of channels for the input data is c_i, the number of input channels of the convolution kernel also needs to be c_i.
  - If our convolution kernel's window shape is $k_h \times k_w$, then, when $c_i = 1$, we can think of our convolution kernel as just a two-dimensional tensor of shape $k_h \times k_w$.
  - However, when $c_i > 1$, we need a kernel that contains a tensor of shape $k_h \times k_w$ for every input channel.
  - Concatenating these c_i tensors together yields a convolution kernel of shape $c_i \times k_h \times k_w$.
- Since the input and convolution kernel each have c_i channels, we can perform a cross-correlation operation on the 2D tensor of the input and the 2D tensor of the convolution kernel for each channel.
- **We add the c_i results** together (**summing over the channels**) to yield a 2D tensor.

Multiple Input Channels

- Fig. 7.4.1 provides an example of a two-dimensional cross-correlation with two input channels.
- The shaded portions are the **first output, input**, and **kernel** tensor elements used for the output computation.
  - (1×1+2×2+4×3+5×4)+(0×0+1×1+3×2+4×3)=56.

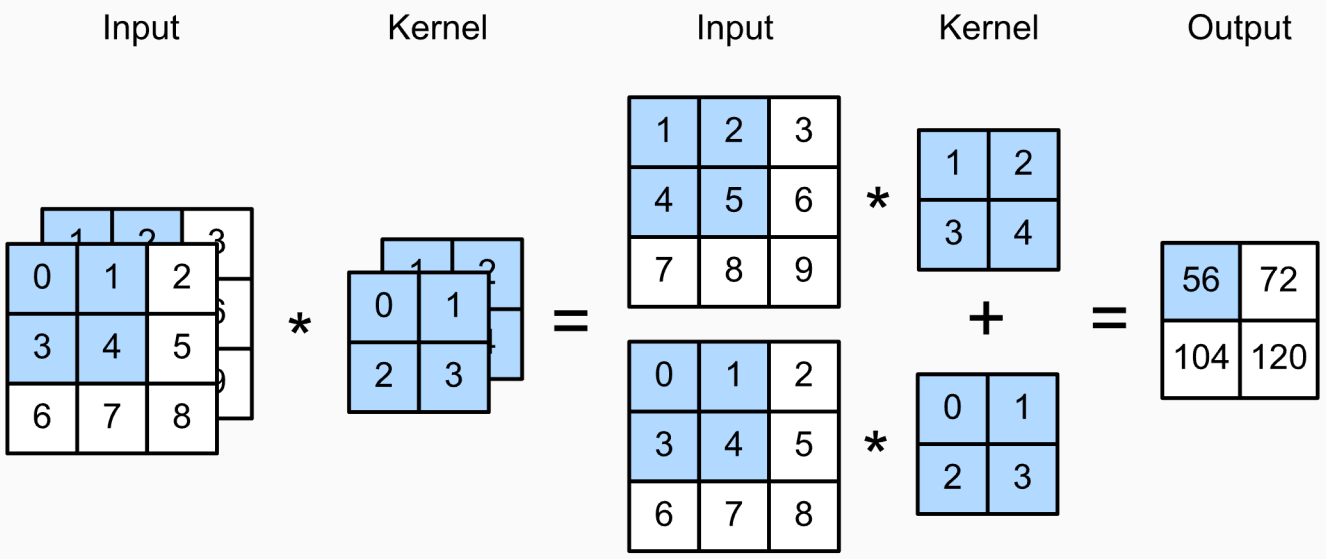


Fig. 7.4.1 Cross-correlation computation with two input channels.

Multiple Input Channels

- Let's implement cross-correlation operations with multiple input channels.
- Notice that all we are doing is performing a cross-correlation operation per channel and then adding up the results.

```
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.]])
```

Multiple Output Channels

- Regardless of the number of input channels, so far we always ended up with one output channel.

- In the most popular neural network architectures, **we increase the channel dimension as we go deeper in the neural network**, while **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.
- A naive interpretation would suggest that representations are learned independently per pixel or per channel.
- Instead, channels are optimized to be jointly useful.
- This means that rather than mapping a single channel to an edge detector, it may simply mean that some direction in channel space corresponds to detecting edges.

- c_i & c_o: the number of input and output channels
- k_h & k_w: the height and width of the kernel
- To get an **output with multiple channels**, we can create a **kernel** tensor of shape $\boldsymbol{c_i \times k_h \times k_w}$ for every output channel
- We **concatenate** them on the **output channel dimension** → shape of the convolution **kernel**: $\boldsymbol{c_o \times c_i \times k_h \times k_w}$
- The result on **each output channel** is **calculated** from the convolution **kernel corresponding to that output channel** and **takes input from all channels in the input** tensor.

Multiple Output Channels

- We implement a cross-correlation function to calculate the output of multiple channels.
- Then, construct a convolution kernel with 3 output channels by concatenating the kernel tensor for K with K+1 and K+2.

```
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)

corr2d_multi_in_out(X, K)
```

```
tensor([[[ 56., 72.], [104., 120.]],
        [[ 76., 100.], [148., 172.]],
        [[ 96., 128.], [192., 224.]]])
```

- The output contains three channels.
- The result of the first channel is consistent with the previous result of the multi-input single-output channel kernel.

1x1 Convolutional Layer

- At first, a 1×1 convolution, i.e., $k_h=k_w=1$, does not seem to make sense.
  - After all, a convolution correlates adjacent pixels.
- Nonetheless, they are often used in deep networks (Lin et al., 2013, Szegedy et al., 2017).
- It loses the ability of larger convolutional layers to recognize patterns consisting of interactions among adjacent elements in the height and width dimensions.
- The only **computation** of the 1×1 convolution **occurs on the channel dimension**.

- Fig. 7.4.2 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.

1x1 Convolutional Layer

- **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).

- Convolutional layers are typically followed by nonlinearities.
  - This ensures that 1×1 convolutions cannot simply be folded into other convolutions.

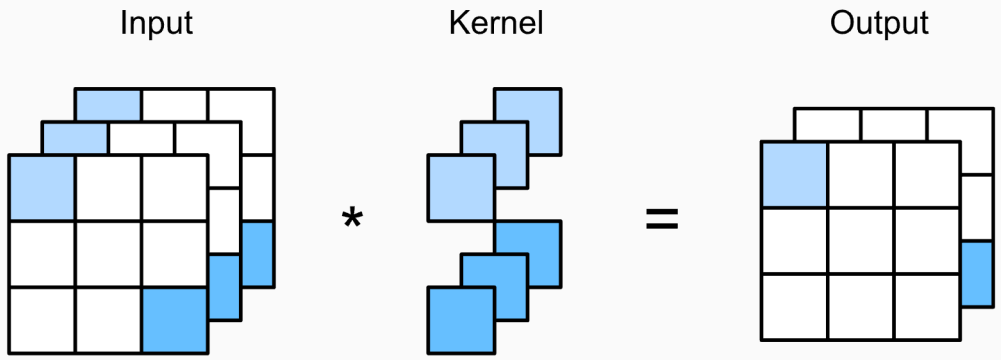


Fig. 7.4.2 The cross-correlation computation uses the 1×1 convolution kernel with three input channels and two output channels. The input and output have the same height and width.

1x1 Convolutional Layer

- We implement a 1×1 convolution using a fully connected layer.
- We just need to make some **adjustments to the data shape** before and after the matrix multiplication.
- When performing 1×1 convolutions, the below function is equivalent to the previously implemented cross-correlation function corr2d_multi_in_out.

```
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
```

Discussion

- Channels allow us to combine the best of both worlds:
  - MLPs allow for significant nonlinearities
  - Convolutions allow for localized analysis of features
- 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.

- This flexibility comes at a price.
- Given an image of size (h×w), the cost for computing a k×k convolution is $O(h \cdot w \cdot k^2)$.
- For c_i and c_o input and output channels, this increases to $O(h \cdot w \cdot k^2 \cdot c_i \cdot c_o)$.
- For a 256×256 pixel image with a 5×5 kernel and 128 input and output channels respectively this amounts to over 53 billion operations (we count multiplications and additions separately).

Pooling

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

Pooling

- In many cases our ultimate task asks some global question about the image
  - e.g., does it contain a cat?

- So, final layer units should be sensitive to the entire input.
  - By gradually aggregating information, yielding coarser and coarser maps, we accomplish this goal of ultimately learning a global representation.
  - The **deeper** we go in the **network**, the **larger the receptive field** (relative to the input) to which each hidden node is sensitive.
- **Reducing spatial resolution accelerates this process**, since the convolution kernels cover a larger effective area.

- Moreover, when detecting lower-level features, such as edges, we often want our representations to be somewhat **invariant to translation**.
  - In reality, objects hardly ever occur exactly at the same place.
  - Even with a tripod and a stationary object, vibration of the camera due to the movement of the shutter might shift everything by a pixel or so.

- This section introduces **pooling layers**, which serve the dual purposes of **1) mitigating the sensitivity of convolutional layers to location** and of **2) spatially downsampling representations**.

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
  - They compute a single output for each location traversed by the fixed-shape window (= pooling window).
- However, the pooling layer contains no parameters (there is no kernel).
- Instead, pooling operators are deterministic, typically calculating either the **maximum** or the **average** value of the elements **in the pooling window**.

- **Maximum pooling**:
  - average over adjacent pixels to obtain an image with better signal-to-noise ratio (SNR)
  - since we are combining the information from multiple adjacent pixels.
- **Average pooling:**
  - introduced in the context of cognitive neuroscience
  - to describe how information aggregation might be aggregated hierarchically (for the purpose of object recognition)
  - more commonly used

Pooling

- Pooling windows **slide** across the input tensor from **left to right** and **top to bottom**.
- At each location that the pooling window hits, it computes the maximum or average value of the input subtensor in the window.

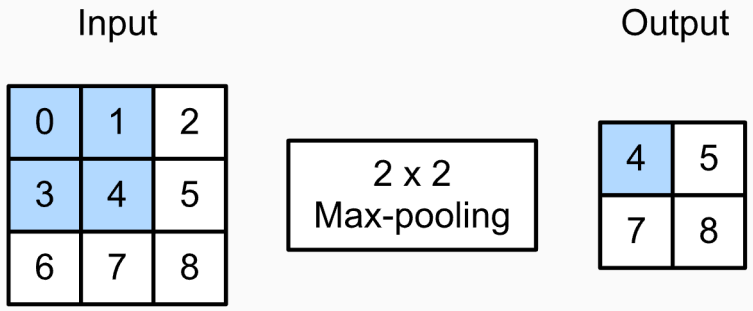


Fig. 7.5.1 **Max-pooling** with a pooling window shape of **2×2**.

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

Pooling

- Let's implement the **forward propagation of the pooling layer** in the *pool2d* function.
- This function is similar to the *corr2d* function.
- However, no kernel is needed, computing the output as either the maximum or the average of each region in the input.

```
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

- We construct the input tensor X in Fig. 7.5.1 to validate the output of the 2D max-pooling (and average pooling) layer.

```
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), 'max')
```

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

```
pool2d(X, (2, 2), 'avg')
```

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

Padding and Stride

- As with convolutional layers, pooling layers **change the output shape**.
- We can adjust the operation to achieve a desired output shape by **padding the input** and **adjusting the stride**.
- We can demonstrate the use of padding and strides in pooling layers via the built-in 2D max-pooling layer from DL framework.
- We first construct an input tensor X whose shape has four dimensions
  - number of examples (batch size) = 1
  - number of channels = 1

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

Padding and Stride

- DL frameworks default to matching pooling window sizes and stride.
- E.g., if we use 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.]]]])
```

- The stride and padding can be manually specified to override framework defaults.

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

Multiple Channels

- As we can see, the number of output channels is still two after pooling.

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

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

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

CNN (LeNet)

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

CNN

- We now have all the ingredients required to assemble a fully-functional CNN.

- We first applied a linear model with softmax regression (Section 4.4) and an MLP (Section 5.2) to pictures of clothing in the Fashion-MNIST dataset.
  - To make such data amenable we first flattened each image from a 28×28 matrix into a fixed-length 784-dimensional vector, and thereafter processed them in fully connected layers.
  - Now that we have a handle on convolutional layers, we can retain the spatial structure in our images.

- Additional benefit of **replacing fully connected layers with convolutional layers:** require **far fewer parameters**.

LeNet

- We will introduce **LeNet**, **among the first published CNNs** to capture wide attention for its performance on computer vision tasks.
  - The model was introduced by and named for Yann LeCun (a researcher at AT&T Bell Labs then) for the purpose of recognizing handwritten digits in images (LeCun et al., 1998).
  - This work represented the culmination of a decade of research developing the technology
  - LeCun's team published the first study to successfully train CNNs via backpropagation (LeCun et al., 1989).

- 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.

- LeNet was eventually adapted to **recognize digits for processing deposits in ATM machines**.

- To this day, **some ATMs still run the code that Yann LeCun and** (his colleague) **Leon Bottou wrote in the 1990s**!

LeNet

- We will introduce **LeNet**, **among the first published CNNs** to capture wide attention for its performance on computer vision tasks.
  - The model was introduced by and named for Yann LeCun (a researcher at AT&T Bell Labs then) for the purpose of recognizing handwritten digits in images (LeCun et al., 1998).
  - This work represented the culmination of a decade of research developing the technology
  - LeCun's team published the first study to successfully train CNNs via backpropagation (LeCun et al., 1989).

- 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.

- LeNet was eventually adapted to **recognize digits for processing deposits in ATM machines**.

- To this day, **some ATMs still run the code that Yann LeCun and** (his colleague) **Leon Bottou wrote in the 1990s**!

LeNet

- LeNet (LeNet-5) consists of two parts:
  - (i) a **convolutional encoder** consisting of two convolutional layers; and
  - (ii) a **dense block** consisting of three fully connected layers.
  - The architecture is summarized in Fig. 7.6.1.

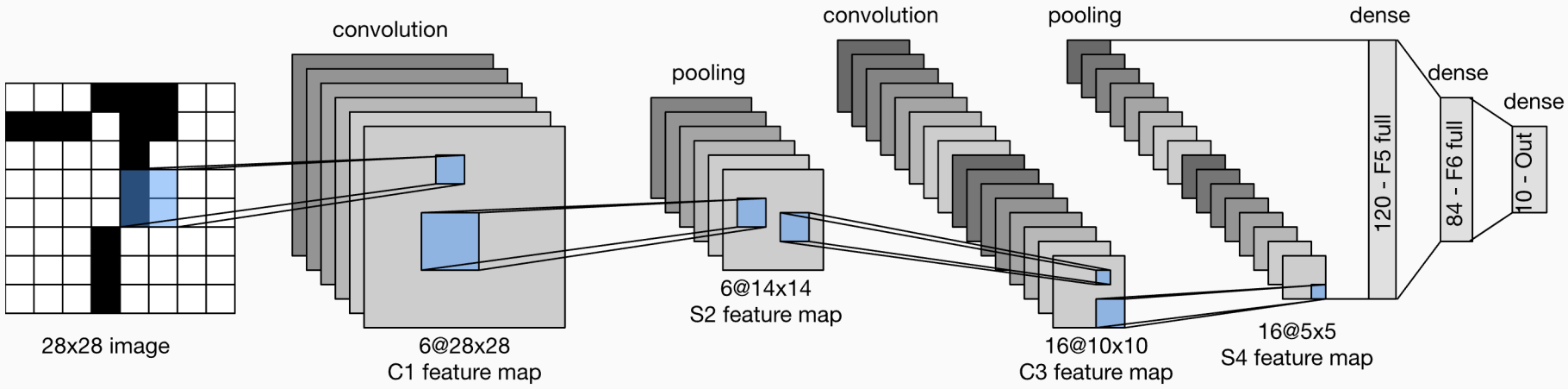


Fig. 7.6.1 Data flow in LeNet. The input is a handwritten digit, the output is a probability over 10 possible outcomes.

LeNet

- The basic units in each **convolutional block** are a convolutional layer, a sigmoid activation function, and a subsequent average pooling operation.
  - Note that while ReLUs and max-pooling work better, they had not yet been discovered at that time.

- Each **convolutional layer** uses a 5×5 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 **2×2 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).

LeNet

- To pass output from the convolutional block to the **dense block**, we **flatten each example** in the minibatch.
  - We take the 4D input and transform it into the 2D input expected by fully connected layers
  - 1st dimension: index examples in the minibatch
  - 2nd dimension: flat vector representation of each example

- LeNet's **dense block** has three fully connected layers, with 120, 84, and 10 outputs, respectively.
- Because we are still performing classification, the 10-dimensional output layer corresponds to the number of possible output classes.

- Although we learned various complicated components of CNNs, the following code snippet shows that implementing such models with modern DL frameworks is remarkably simple.
  - We need only to **instantiate a Sequential block** and **chain together the appropriate layers**, using Xavier initialization.

LeNet

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

- Note that we replaced the Gaussian activation layer (used in LeNet-5) by a softmax layer for simplicity.

LeNet

- By passing a single-channel (black and white) 28×28 image through the network and printing the output shape at each layer, we can inspect the model to ensure that its operations line up with what we expect from Fig. 7.6.2.

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

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

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

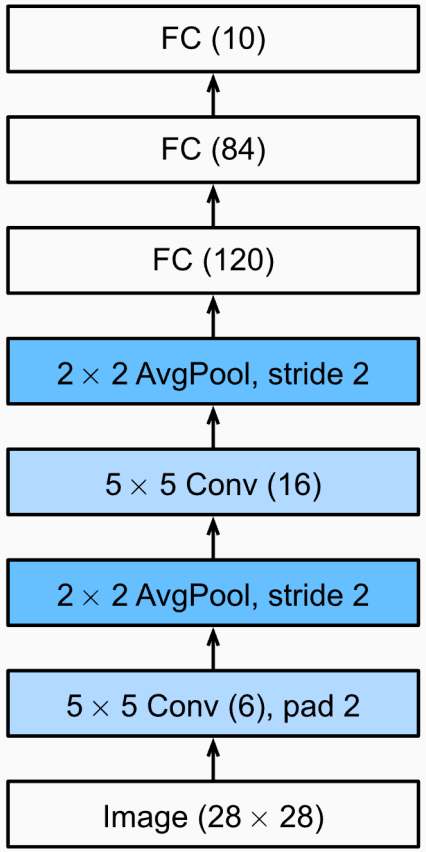


Fig. 7.6.2 Compressed notation for LeNet-5.

LeNet

- 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 5×5 kernel.

- ~~As an aside, the image size of 28×28 pixels in the original MNIST OCR dataset is a result of trimming two pixel rows and columns from the original scans that measured 32×32 pixels.~~
  - ~~This was done primarily to save space (a 30% reduction) at a time when megabytes mattered.~~

- In contrast, the second **convolutional layer** forgoes padding, and thus the height and width are both reduced by four pixels.
- As we go up the stack of layers, the number of channels increases layer-over-layer from 1 in the input to 6 after the first convolutional layer and 16 after the second convolutional layer.

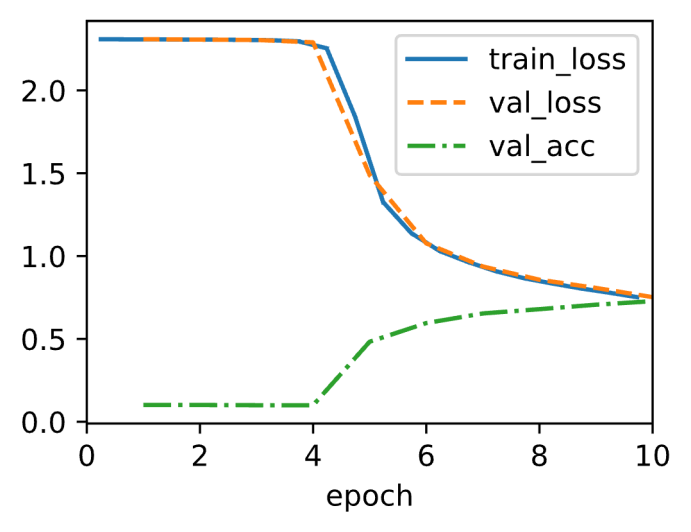
- However, each **pooling layer** halves the height and width.

- Finally, each **fully connected** layer reduces dimensionality, finally emitting an output whose dimension matches the number of classes.

Training

- 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**.
- If you have access to a **GPU**, this might be a good time to put it into action to **speed up training**.
- Note that the d2l.Trainer class takes care of all details.
  - By default, it initializes the model parameters on the **available devices**.
  - Just as with MLPs, our loss function is **cross-entropy**, and we minimize it via **minibatch stochastic gradient descent**.

```
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)
```



Summary

- We have made significant progress in this chapter.
- We moved from the **MLPs of the 1980s** to the **CNNs of the 1990s and early 2000s.**
- The architectures proposed, e.g., in the form of LeNet-5, remain meaningful even to this day.
- It is worth comparing the error rates on Fashion-MNIST achievable with LeNet-5 to MLPs and ResNet.
  - LeNet is much more similar to the latter than to the former.
  - One of the primary differences is that **greater amounts of computation** enabled significantly **more complex architectures**.
  - A second difference is the relative **ease of implementation**.
  - 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.
  - It is this incredible **productivity boost** that has democratized DL model development tremendously.

Modern CNNs

Modern CNN

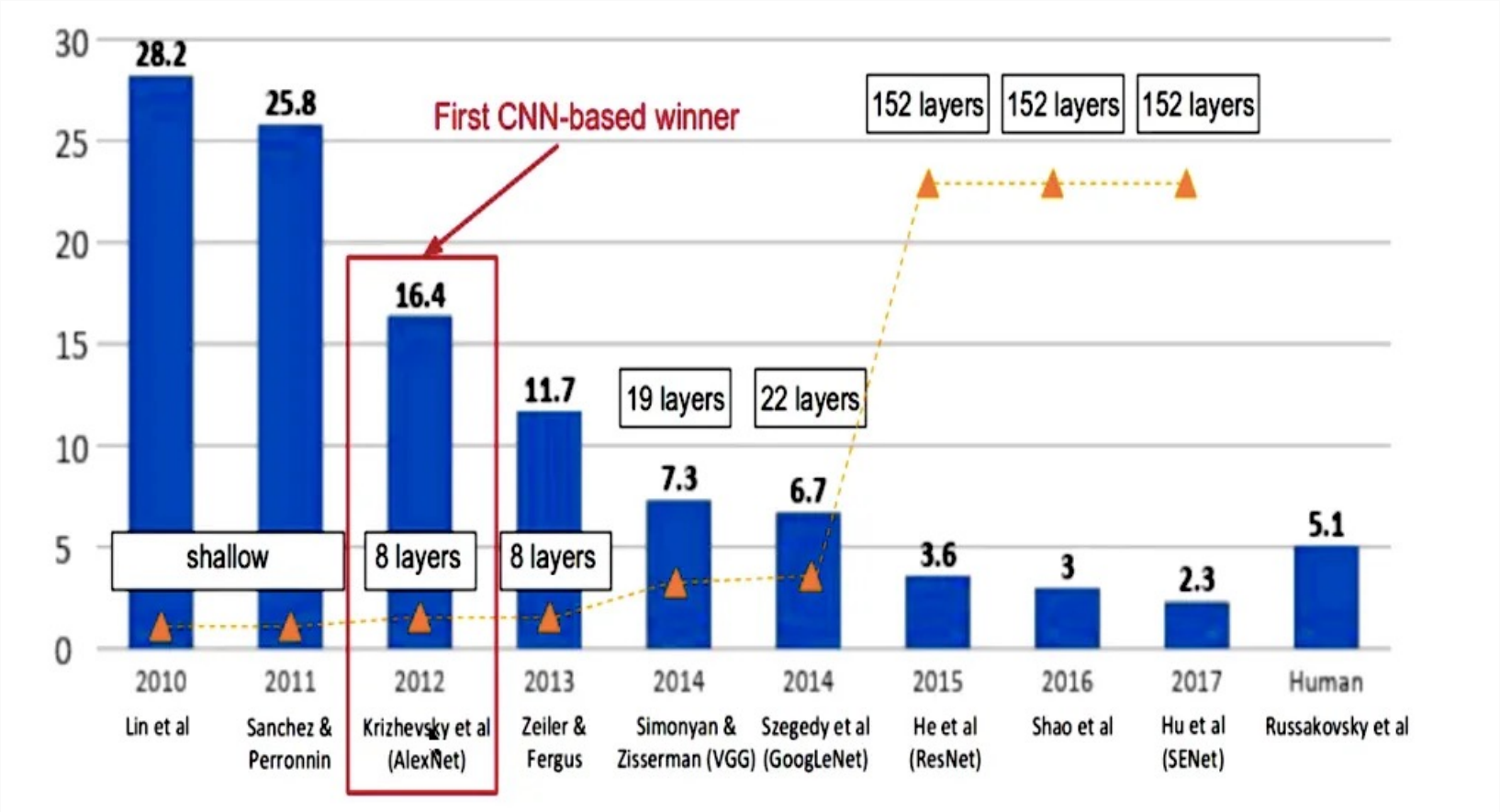


Fig. Evolution of CNN for object detection. Winners of ImageNet Classification Challenge.

Modern CNN

AlexNet (paper) was the first winner of the ImageNet challenge and was based on a CNN, and since 2012, every year's challenge has been won by a CNN; significantly outperforming other deep and shallow (traditional) machine learning methods.

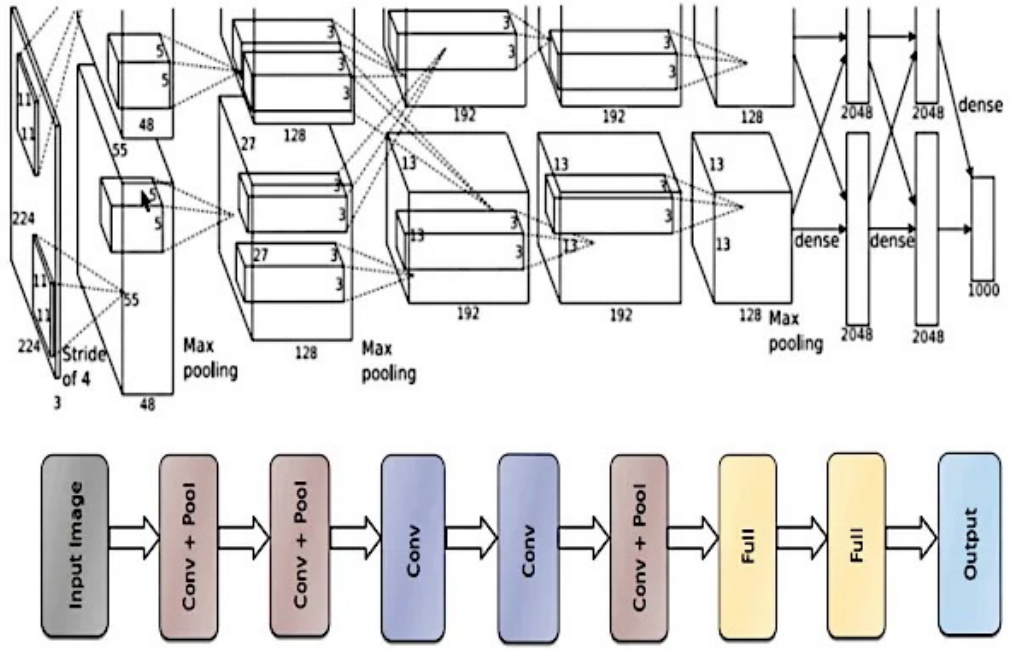


Fig. 4: The architecture of AlexNet

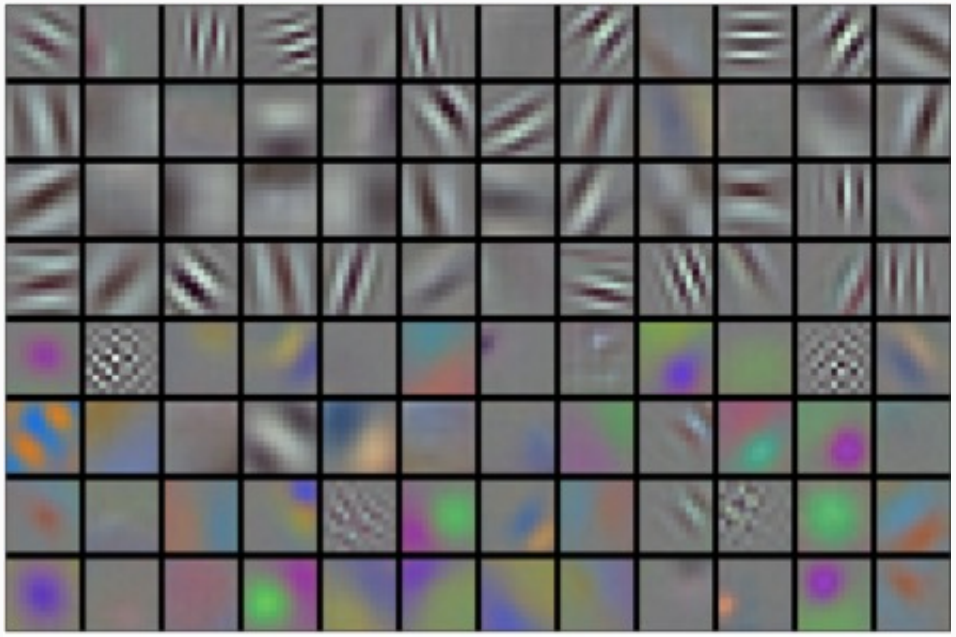


Fig. 8.1.1 Image filters learned by the first layer of AlexNet.

Modern CNN

- **AlexNet vs LeNet**
  - AlexNet is **much deeper** than LeNet-5.
    - 8 layers: 5 convolutional layers, 2 fully connected hidden layers, and 1 fully connected output layer.
  - AlexNet used the **ReLU** instead of the sigmoid as its activation function.

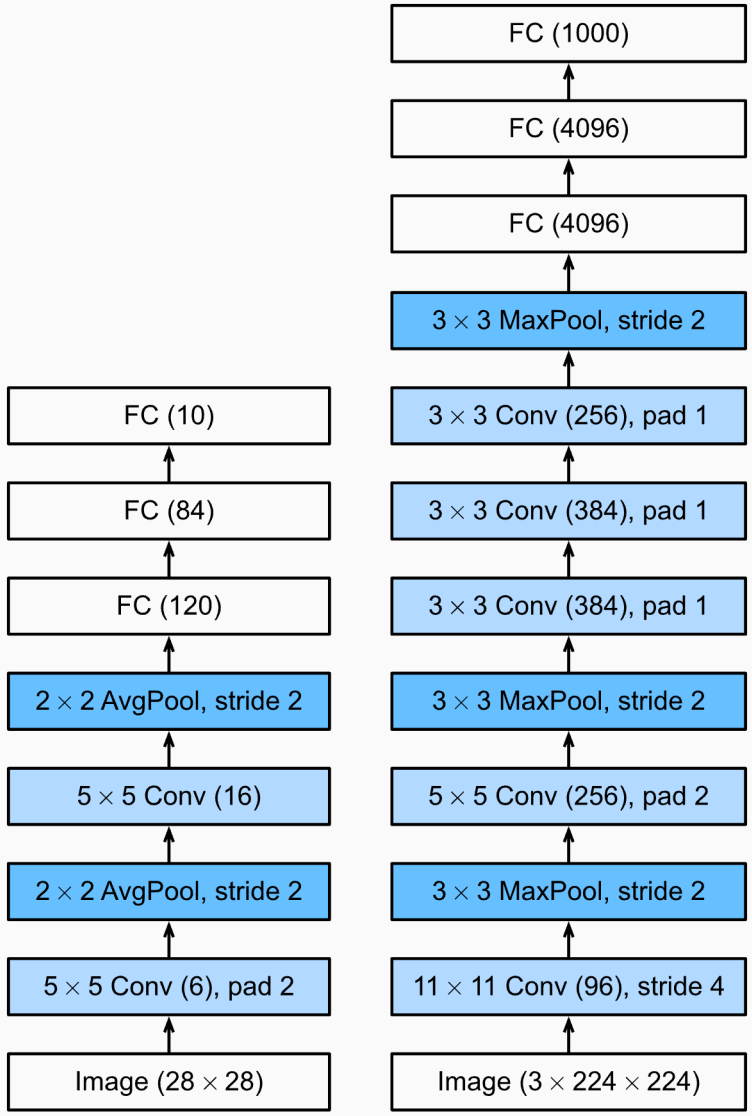


Fig. 8.1.2 From LeNet (left) to AlexNet (right).

Modern CNN

- **VGG**
  - The **idea of using blocks** first emerged from the Visual Geometry Group (VGG) at Oxford University.
  - It is **easy to implement these repeated structures** in code with any modern DL framework by using loops and subroutines.

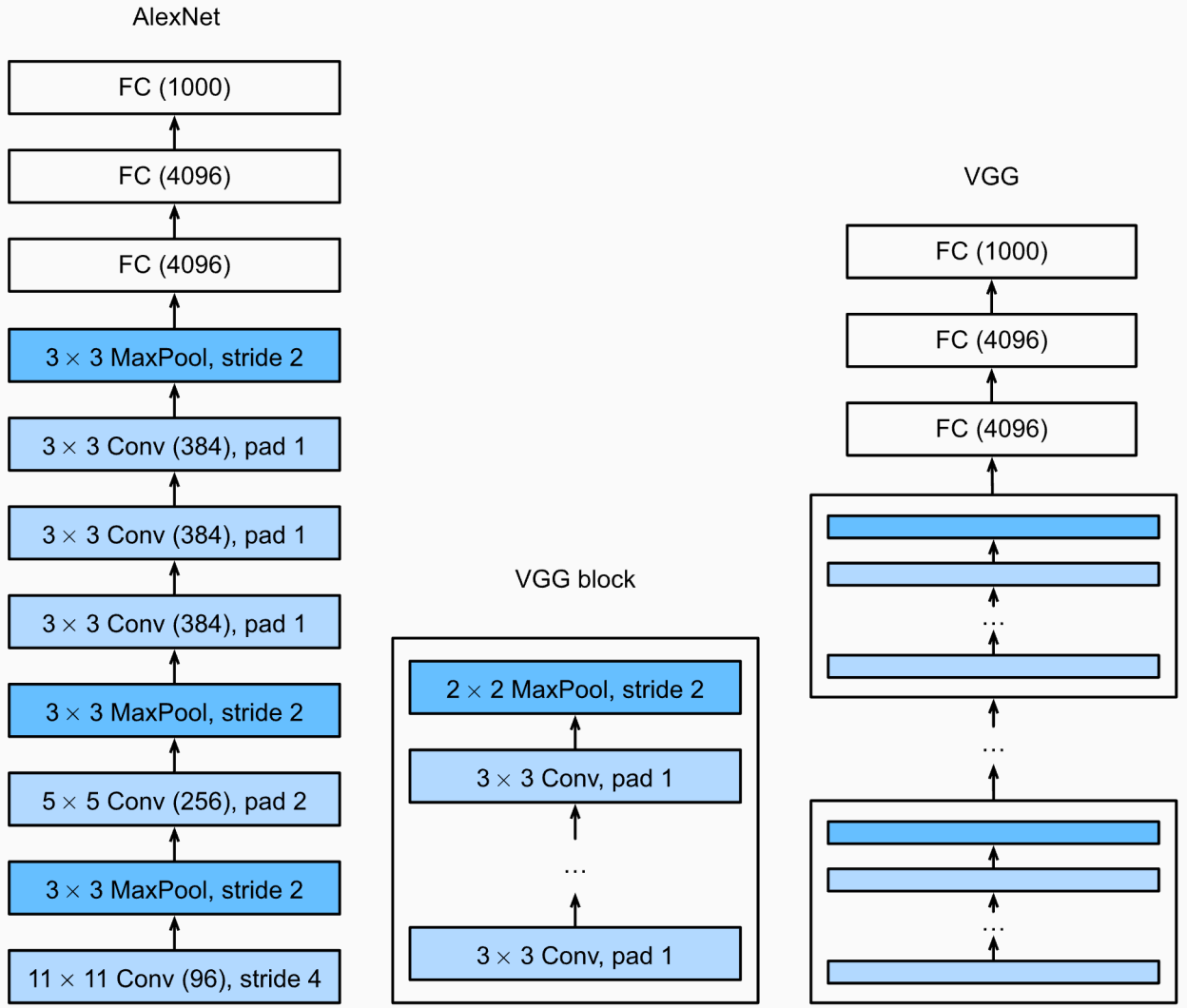


Fig. 8.2.1 From AlexNet to VGG. The key difference is that VGG consists of blocks of layers, whereas AlexNet's layers are all designed individually.

Modern CNN

- GoogLeNet won the ImageNet Challenge, combining the strengths of **NiN**, **repeated blocks**, and a **cocktail of convolution kernels**.
- It was arguably also the first network that exhibited a clear distinction among the **stem (data ingest), body (data processing)**, and **head (prediction)** in a CNN.
  - This design pattern has persisted ever since in the design of deep networks
  - The **stem** is given by the first two or three convolutions that operate on the image. They extract low-level features from the underlying images.
  - This is followed by a **body** of convolutional blocks.
  - Finally, the **head** maps the features obtained so far to the required classification, segmentation, detection, or tracking problem at hand.

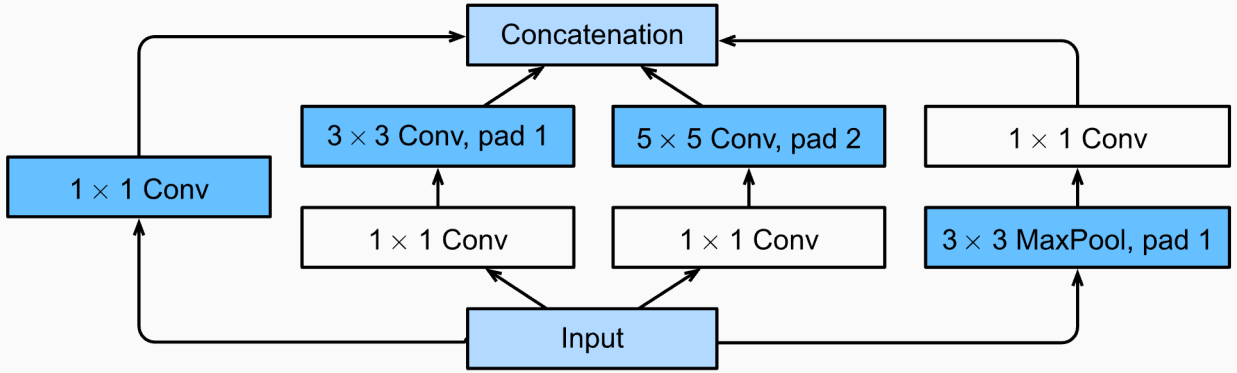


Fig. 8.4.1 Structure of the Inception block.

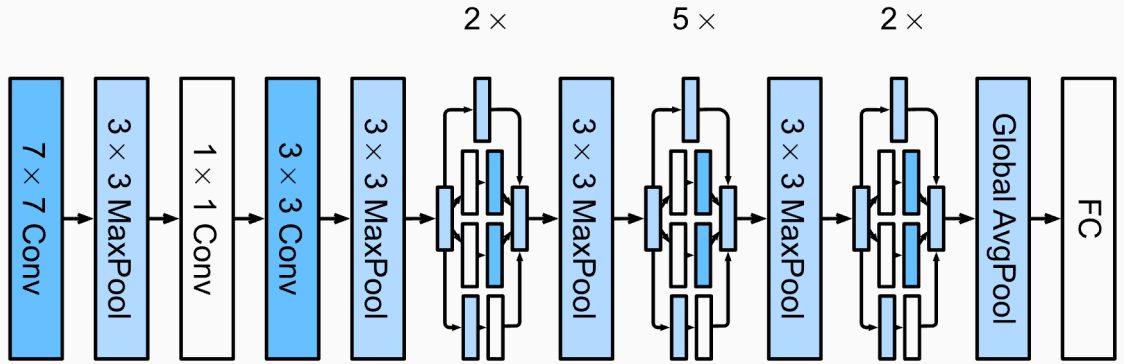


Fig. 8.4.2 The GoogLeNet architecture.

Modern CNN

- The key **contribution** in GoogLeNet was the design of the **network body**.
- It solved the problem of **selecting convolution kernels** in an ingenious way.
- While other works tried to identify which convolution, ranging from 1×1 to 11×11 would be best, it simply **concatenated multi-branch convolutions**.

Modern CNN

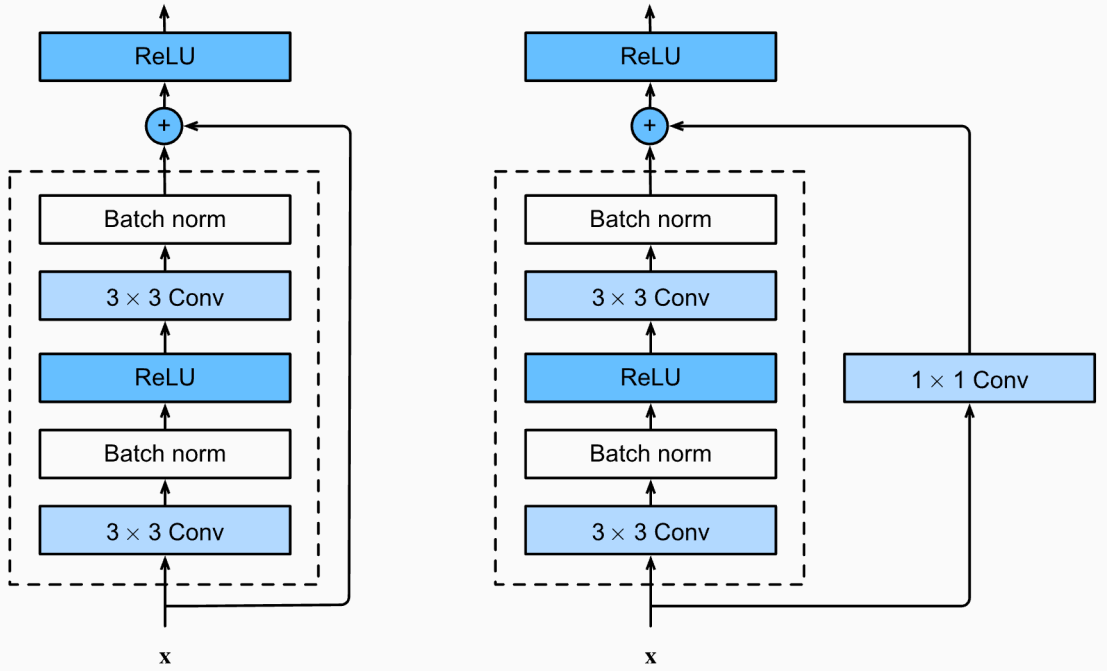


Fig. 8.6.3 ResNet block with and without 1×1 convolution, which transforms the input into the desired shape for the addition operation.

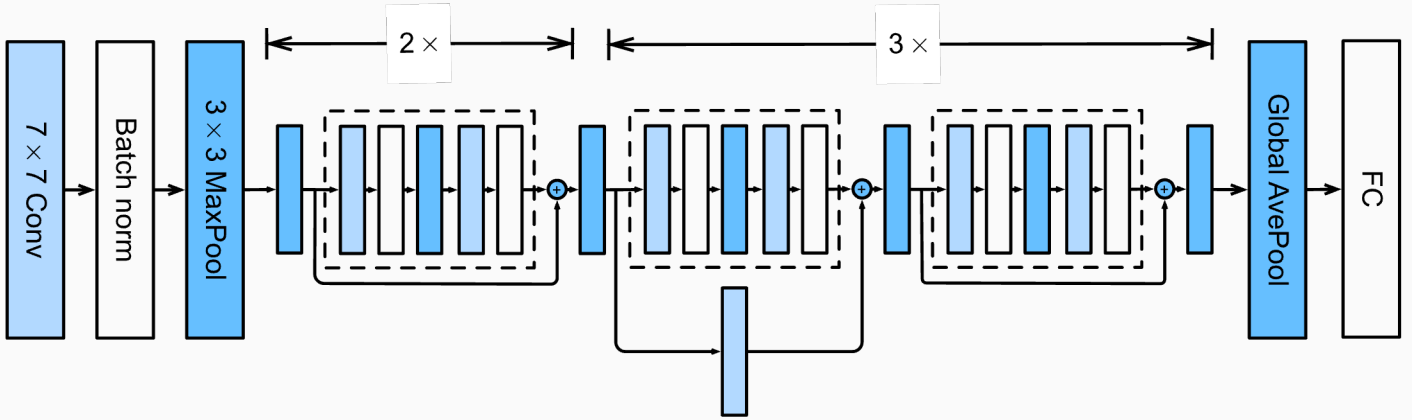


Fig. 8.6.4 The ResNet-18 architecture.

Latest Model: Stable Diffusion

```
[ ]  images = model.text_to_image("FPS game scene, battle royale mode, unreal engine", batch_size=3, num_steps=25)

     def plot_images(images):
         plt.figure(figsize=(20, 20))
         for i in range(len(images)):
             ax = plt.subplot(1, len(images), i+1)
             plt.imshow(images[i])
             plt.axis("off")

     plot_images(images)
```

```
25/25 [==============================] - 62s 2s/step
```

Code: https://colab.research.google.com/drive/1RDgQplNHbos9o6ohcV0Wlm0qU3-nazRy?usp=drive_link