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

Show hidden output

COSE474-2024F: Deep Learning

8.2. Networks Using Blocks (VGG)

While AlexNet offered empirical evidence that deep CNNs can achieve good results, it did not provide a general template to guide subsequent researchers in designing new networks. In the following sections, we will introduce several heuristic concepts commonly used to design deep networks.

Progress in this field mirrors that of VLSI (very large scale integration) in chip design where engineers moved from placing transistors to logical elements to logic blocks (Mead, 1980). Similarly, the design of neural network architectures has grown progressively more abstract, with researchers moving from thinking in terms of individual neurons to whole layers, and now to blocks, repeating patterns of layers. A decade later, this has now progressed to researchers using entire trained models to repurpose them for different, albeit related, tasks. Such large pretrained models are typically called foundation models (Bommasani et al., 2021).

Back to network design. The idea of using blocks first emerged from the Visual Geometry Group (VGG) at Oxford University, in their eponymously-named VGG network (Simonyan and Zisserman, 2014). It is easy to implement these repeated structures in code with any modern deep learning framework by using loops and subroutines.

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

8.2.1. VGG Blocks

The basic building block of 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 max-pooling to reduce the resolution. One of the problems with this approach is that the spatial resolution decreases quite rapidly. In particular, this imposes a hard limit of $\log_2 d$ convolutional layers on the network before all dimensions (d) are used up. For instance, in the case of ImageNet, it would be impossible to have more than 8 convolutional layers in this way.

The key idea of Simonyan and Zisserman (2014) was to use multiple convolutions in between downsampling via max-pooling in the form of a block. They were primarily interested in whether deep or wide networks perform better. For instance, the successive application of two 3×3 convolutions touches the same pixels as a single 5×5 convolution does. At the same time, the latter uses approximately as many parameters ($25 \cdot c^2$) as three 3×3 convolutions do ($3 \cdot 9 \cdot c^2$). In a rather detailed analysis they showed that deep and narrow networks significantly outperform their shallow counterparts. This set deep learning on a quest for ever deeper networks with over 100 layers for typical applications. Stacking 3×3 convolutions has become a gold standard in later deep networks (a design decision only to be revisited recently by Liu et al. (2022)). Consequently, fast implementations for small convolutions have become a staple on GPUs (Lavin and Gray, 2016).

Back to VGG: a VGG block consists of a *sequence* of convolutions with 3×3 kernels with padding of 1 (keeping height and width) followed by a 2×2 max-pooling layer with stride of 2 (halving height and width after each block). In the code below, we define a function called `vgg_block` to implement one VGG block.

The function below takes two arguments, corresponding to the number of convolutional layers `num_convs` and the number of output channels `num_channels`.

```
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

Like AlexNet and LeNet, the VGG Network can be partitioned into two parts: the first consisting mostly of convolutional and pooling layers and the second consisting of fully connected layers that are identical to those in AlexNet. The key difference is that the convolutional layers are grouped in nonlinear transformations that leave the dimensonality unchanged, followed by a resolution-reduction step, as depicted in Fig. 8.2.1.

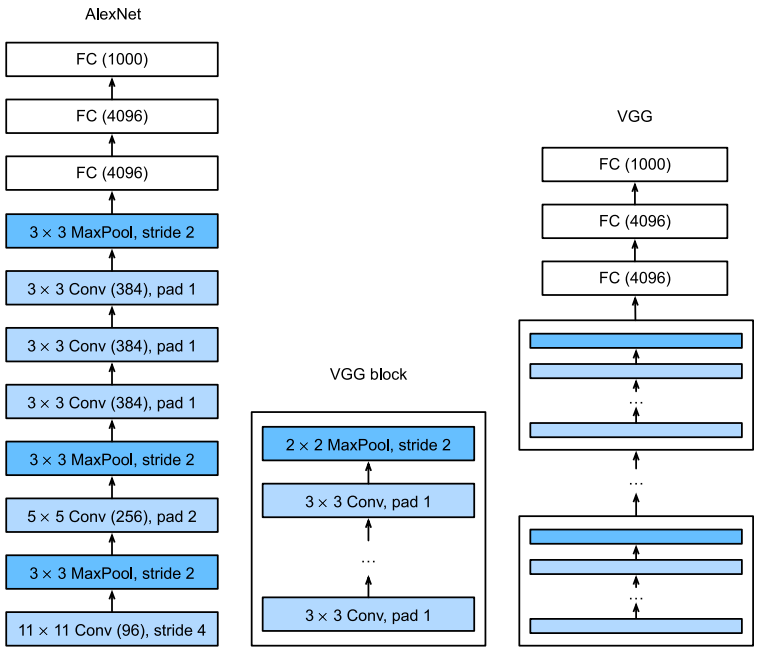

Fig. 8.2.1 From AlexNet to VGG. The key difference is that VGG consists of blocks of layers, whereas AlexNet's layers are all designed individually.

The convolutional part of the network connects several VGG blocks from Fig. 8.2.1 (also defined in the `vgg_block` function) in succession. This grouping of convolutions is a pattern that has remained almost unchanged over the past decade, although the specific choice of operations has undergone considerable modifications. The variable `arch` consists of a list of tuples (one per block), where each contains two values: the number of convolutional layers and the number of output channels, which are precisely the arguments required to call the `vgg_block` function. As such, VGG defines a *family* of networks rather than just a specific manifestation. To build a specific network we simply iterate over `arch` to compose the blocks.

```
class VGG(d2l.Classifier):
    def __init__(self, arch, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        conv_blks = []
        for (num_convs, out_channels) in arch:
            conv_blks.append(vgg_block(num_convs, out_channels))
        self.net = nn.Sequential(
            *conv_blks, nn.Flatten(),
            nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(0.5),
            nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(0.5),
            nn.LazyLinear(num_classes))
        self.net.apply(d2l.init_cnn)
```

The original VGG network had five convolutional blocks, among which the first two have one convolutional layer each and the latter three contain two convolutional layers each. The first block has 64 output channels and each subsequent block doubles the number of output channels, until that number reaches 512. Since this network uses eight convolutional layers and three fully connected layers, it is often called VGG-11.

```
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])
```

As you can see, we halve height and width at each block, finally reaching a height and width of 7 before flattening the representations for processing by the fully connected part of the network. Simonyan and Zisserman (2014) described several other variants of VGG. In fact, it has become the norm to propose families of networks with different speed–accuracy trade-off when introducing a new architecture.

8.2.3. Training

Since VGG-11 is computationally more demanding than AlexNet we construct a network with a smaller number of channels. This is more than sufficient for training on Fashion-MNIST. The model training process is similar to that of AlexNet in Section 8.1. Again observe the close match between validation and training loss, suggesting only a small amount of overfitting.

```
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)
```

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz to ../data/FashionMNIST/raw/train-imag
100%|██████████| 26421880/26421880 [00:00<00:00, 115385709.48it/s]
Extracting ../data/FashionMNIST/raw/train-images-idx3-ubyte.gz to ../data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz to ../data/FashionMNIST/raw/train-labe
100%|██████████| 29515/29515 [00:00<00:00, 6555196.32it/s]Extracting ../data/FashionMNIST/raw/train-labels-idx1-ubyte.gz to ../data/Fash

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz to ../data/FashionMNIST/raw/t10k-images

100%|██████████| 4422102/4422102 [00:00<00:00, 62695259.25it/s]
Extracting ../data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to ../data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz to ../data/FashionMNIST/raw/t10k-labels
100%|██████████| 5148/5148 [00:00<00:00, 17469479.77it/s]
Extracting ../data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to ../data/FashionMNIST/raw

/usr/local/lib/python3.10/dist-packages/torch/utils/data/dataloader.py:557: UserWarning: This DataLoader will create 4 worker processes
  warnings.warn(_create_warning_msg(
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-5-8d3dfa6be93b> in <cell line: 5>()
      3 data = d2l.FashionMNIST(batch_size=128, resize=(224, 224))
      4 model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
----> 5 trainer.fit(model, data)

                                 15 frames
/usr/local/lib/python3.10/dist-packages/torch/nn/functional.py in relu(input, inplace)
   1498         result = torch.relu_(input)
   1499     else:
-> 1500         result = torch.relu(input)
   1501     return result
   1502

KeyboardInterrupt:
```

8.2.4. Discussion

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

```
Start coding or generate with AI.
```