7 | Modern Convolutional Neural Networks

Now that we understand the basics of wiring together convolutional neural networks, we will take you through a tour of modern deep learning. In this chapter, each section will correspond to a significant neural network architecture that was at some point (or currently) the base model upon which an enormous amount of research and projects were built. Each of these networks was at briefly a dominant architecture and many were at one point winners or runners-up in the famous ImageNet competition, which has served as a barometer of progress on supervised learning in computer vision since 2010.

These models include AlexNet, the first large-scale network deployed to beat conventional computer vision methods on a large-scale vision challenge; the VGG network, which makes use of a number of repeating blocks of elements; the network in network (NiN) which convolves whole neural networks patch-wise over inputs; the GoogLeNet, which makes use of networks with parallel concatenations (GoogLeNet); residual networks (ResNet) which are currently the most popular go-to architecture today, and densely connected networks (DenseNet), which are expensive to compute but have set some recent benchmarks.

7.1 Deep Convolutional Neural Networks (AlexNet)

Although convolutional neural networks were well known in the computer vision and machine learning communities following the introduction of LeNet, they did not immediately dominate the field. Although LeNet achieved good results on early small datasets, the performance and feasability of training convolutional networks on larger, more realistic datasets had yet to be established In fact, for much of the intervening time between the early 1990s and the watershed results of 2012, neural networks were often surpassed by other machine learning methods, such as support vector machines.

For computer vision, this comparison is perhaps not fair. That is although the inputs to convolutional networks consist of raw or lightly-processed (e.g., by centering) pixel values, practitioners would never feed raw pixels into traditional models. Instead, typical computer vision pipelines consisted of manually engineering feature extraction pipelines. Rather than *learn the features*, the features were *crafted*. Most of the progress came from having more clever ideas for features, and the learning algorithm was often relegated to an afterthought.

Although some neural network accelerators were available in the 1990s, they were not yet sufficiently powerful to make deep multichannel, multilayer convolutional neural networks with a large number of parameters. Moreover, datasets were still relatively small. Added to these obstacles, key tricks for training neural networks including parameter initialization heuristics, clever

variants of stochastic gradient descent, non-squashing activation functions, and effective regularization techniques were still missing.

Thus, rather than training *end-to-end* (pixel to classification) systems, classical pipelines looked more like this:

1. Obtain an interesting dataset. In early days, these datasets required expensive sensors (at the time, 1 megapixel images were state of the art).
2. Preprocess the dataset with hand-crafted features based on some knowledge of optics, geometry, other analytic tools, and occasionally on the serendipitous discoveries of lucky graduate students.
3. Feed the data through a standard set of feature extractors such as SIFT[105], the Scale-Invariant Feature Transform, or SURF[106], the Speeded-Up Robust Features, or any number of other hand-tuned pipelines.
4. Dummp the resulting representations into your favorite classifier, likely a linear model or kernel method, to learn a classifier.

If you spoke to machine learning researchers, they believed that machine learning was both important and beautiful. Elegant theories proved the properties of various classifiers. The field of machine learning was thriving, rigorous and eminently useful. However, if you spoke to a computer vision researcher, you'd hear a very different story. The dirty truth of image recognition, they'd tell you, is that features, not learning algorithms, drove progress. Computer vision researchers justifiably believed that a slightly bigger or cleaner dataset or a slightly improved feature-extraction pipeline mattered far more to the final accuracy than any learning algorithm.

7.1.1 Learning Feature Representation

Another way to cast the state of affairs is that the most important part of the pipeline was the representation. And up until 2012 the representation was calculated mechanically. In fact, engineering a new set of feature functions, improving results, and writing up the method was a prominent genre of paper. SIFT[107], SURF[108], HOG[109], Bags of visual words[110] and similar feature extractors ruled the roost.

Another group of researchers, including Yann LeCun, Geoff Hinton, Yoshua Bengio, Andrew Ng, Shun-ichi Amari, and Juergen Schmidhuber, had different plans. They believed that features themselves ought to be learned. Moreover, they believed that to be reasonably complex, the features ought to be hierarchically composed with multiple jointly learned layers, each with learnable parameters. In the case of an image, the lowest layers might come to detect edges, colors, and textures. Indeed, (Krizhevsky et al., 2012) proposed a new variant of a convolutional neural network which achieved excellent performance in the ImageNet challenge.

Interestingly in the lowest layers of the network, the model learned feature extractors that resembled some traditional filters. Fig. 7.1.1 is reproduced from this paper and describes lower-level image descriptors.

[105] https://en.wikipedia.org/wiki/Scale-invariant_feature_transform
[106] https://en.wikipedia.org/wiki/Speeded_up_robust_features
[107] https://en.wikipedia.org/wiki/Scale-invariant_feature_transform
[108] https://en.wikipedia.org/wiki/Speeded_up_robust_features
[109] https://en.wikipedia.org/wiki/Histogram_of_oriented_gradients
[110] https://en.wikipedia.org/wiki/Bag-of-words_model_in_computer_vision

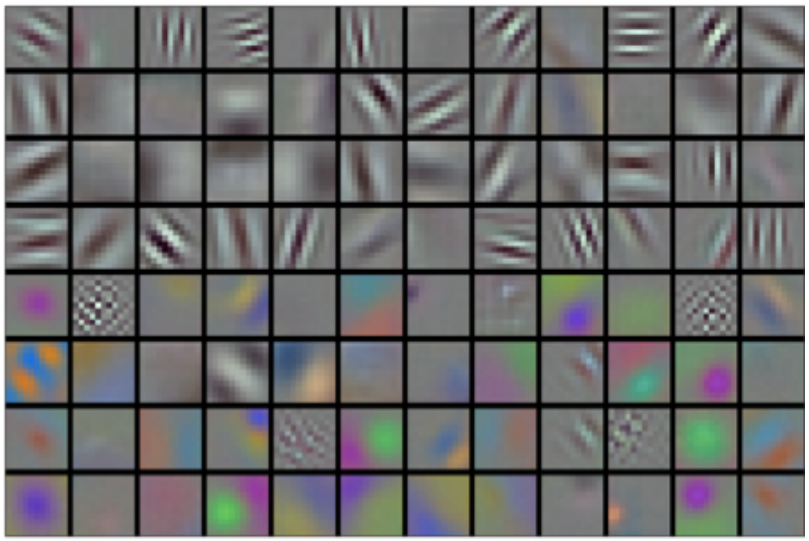


Fig. 7.1.1: Image filters learned by the first layer of AlexNet

Higher layers in the network might build upon these representations to represent larger structures, like eyes, noses, blades of grass, etc. Even higher layers might represent whole objects like people, airplanes, dogs, or frisbees. Ultimately, the final hidden state learns a compact representation of the image that summarizes its contents such that data belonging to different categories be separated easily.

While the ultimate breakthrough for many-layered convolutional networks came in 2012, a core group of researchers had dedicated themselves to this idea, attempting to learn hierarchical representations of visual data for many years. The ultimate breakthrough in 2012 can be attributed to two key factors.

Missing Ingredient - Data

Deep models with many layers require large amounts of data in order to enter the regime where they signficantly outperform traditional methods based on convex optimizations (e.g., linear and kernel methods). However, given the limited storage capacity of computers, the relative expense of sensors, and the comparatively tighter research budgets in the 1990s, most research relied on tiny datasets. Numerous papers addressed the UCI collection of datasets, many of which contained only hundreds or (a few) thousands of images captured in unnatural settings with low resolution.

In 2009, the ImageNet dataset was released, challenging researchers to learn models from 1 million examples, 1,000 each from 1,000 distinct categories of objects. The researchers, led by Fei-Fei Li, who introduced this dataset leveraged Google Image Search to prefilter large candidate sets for each category and employed the Amazon Mechanical Turk crowdsourcing pipeline to confirm for each image whether it belonged to the associated category. This scale was unprecedented. The associated competition, dubbed the ImageNet Challenge pushed computer vision and machine learning research forward, challenging researchers to identify which models performed best at a greater scale than academics had previously considered.

Missing Ingredient - Hardware

Deep learning models are voracious consumers of compute cycles. Training can take hundreds of epochs, and each iteration requires passing data through many layers of computationally-expensive linear algebra operations. This is one of the main reasons why in the 90s and early 2000s, simple algorithms based on the more-efficiently optimized convex objectives were preferred.

Graphical processing units (GPUs) proved to be a game changer in make deep learning feasible. These chips had long been developed for accelerating graphics processing to benefit computer games. In particular, they were optimized for high throughput 4x4 matrix-vector products, which are needed for many computer graphics tasks. Fortunately, this math is strikingly similar to that required to calculate convolutional layers. Around that time, NVIDIA and ATI had begun optimizing GPUs for general compute operations, going as far as to market them as General Purpose GPUs (GPGPU).

To provide some intuition, consider the cores of a modern microprocessor (CPU). Each of the cores is fairly powerful running at a high clock frequency and sporting large caches (up to several MB of L3). Each core is well-suited to executing a wide range of instructions, with branch predictors, a deep pipeline, and other bells and whistles that enable it to run a large variety of programs. This apparent strength, however, is also its Achilles heel: general purpose cores are very expensive to build. They require lots of chip area, a sophisticated support structure (memory interfaces, caching logic between cores, high speed interconnects, etc.), and they are comparatively bad at any single task. Modern laptops have up to 4 cores, and even high end servers rarely exceed 64 cores, simply because it is not cost effective.

By comparison, GPUs consist of 100-1000 small processing elements (the details differ somewhat between NVIDIA, ATI, ARM and other chip vendors), often grouped into larger groups (NVIDIA calls them warps). While each core is relatively weak, sometimes even running at sub-1GHz clock frequency, it is the total number of such cores that makes GPUs orders of magnitude faster than CPUs. For instance, NVIDIA's latest Volta generation offers up to 120 TFlops per chip for specialized instructions (and up to 24 TFlops for more general purpose ones), while floating point performance of CPUs has not exceeded 1 TFlop to date. The reason for why this is possible is actually quite simple: first, power consumption tends to grow *quadratically* with clock frequency. Hence, for the power budget of a CPU core that runs 4x faster (a typical number), you can use 16 GPU cores at 1/4 the speed, which yields 16 x 1/4 = 4x the performance. Furthermore, GPU cores are much simpler (in fact, for a long time they weren't even *able* to execute general purpose code), which makes them more energy efficient. Last, many operations in deep learning require high memory bandwidth. Again, GPUs shine here with buses that are at least 10x as wide as many CPUs.

Back to 2012. A major breakthrough came when Alex Krizhevsky and Ilya Sutskever implemented a deep convolutional neural network that could run on GPU hardware. They realized that the computational bottlenecks in CNNs (convolutions and matrix multiplications) are all operations that could be parallelized in hardware. Using two NIVIDA GTX 580s with 3GB of memory, they implemented fast convolutions. The code cuda-convnet[111] was good enough that for several years it was the industry standard and powered the first couple years of the deep learning boom.

[111] https://code.google.com/archive/p/cuda-convnet/

7.1.2 AlexNet

AlexNet was introduced in 2012, named after Alex Krizhevsky, the first author of the breakthrough ImageNet classification paper (Krizhevsky et al., 2012). AlexNet, which employed an 8-layer convolutional neural network, won the ImageNet Large Scale Visual Recognition Challenge 2012 by a phenomenally large margin. This network proved, for the first time, that the features obtained by learning can transcend manually-design features, breaking the previous paradigm in computer vision. The architectures of AlexNet and LeNet are *very similar*, as Fig. 7.1.2 illustrates. Note that we provide a slightly streamlined version of AlexNet removing some of the design quirks that were needed in 2012 to make the model fit on two small GPUs.

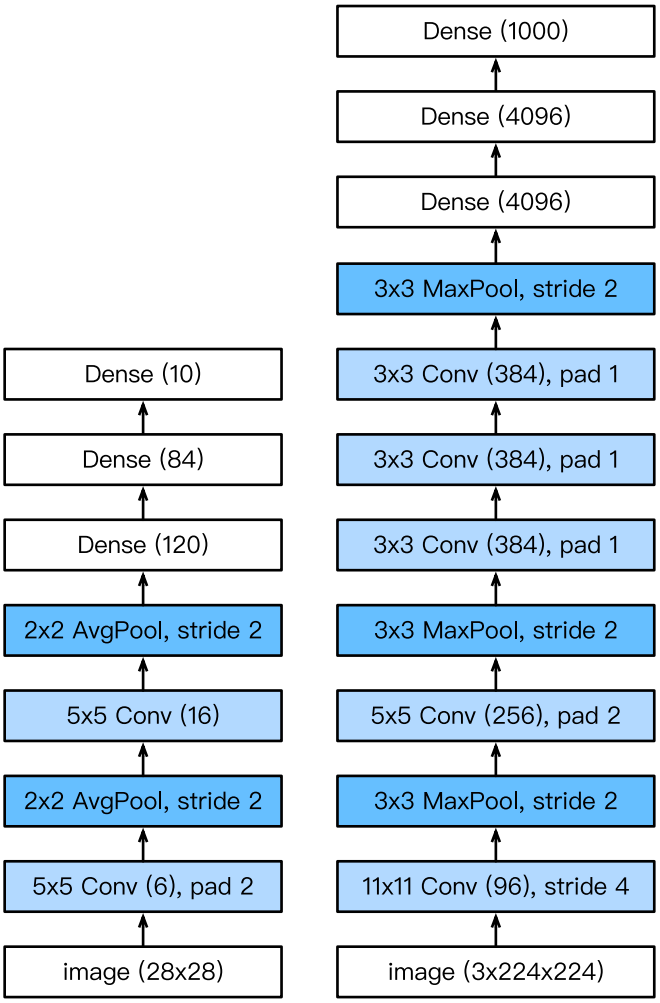


Fig. 7.1.2: LeNet (left) and AlexNet (right)

The design philosophies of AlexNet and LeNet are very similar, but there are also significant differences. First, AlexNet is much deeper than the comparatively small LeNet5. AlexNet consists of eight layers: five convolutional layers, two fully-connected hidden layers, and one fully-connected output layer. Second, AlexNet used the ReLU instead of the sigmoid as its activation function. Let's delve into the details below.

Architecture

In AlexNet's first layer, the convolution window shape is 11×11. Since most images in ImageNet are more than ten times higher and wider than the MNIST images, objects in ImageNet data tend to occupy more pixels. Consequently, a larger convolution window is needed to capture the object. The convolution window shape in the second layer is reduced to 5×5, followed by 3×3. In addition, after the first, second, and fifth convolutional layers, the network adds maximum pooling layers with a window shape of 3×3 and a stride of 2. Moreover, AlexNet has ten times more convolution channels than LeNet.

After the last convolutional layer are two fully-connected layers with 4096 outputs. These two huge fully-connected layers produce model parameters of nearly 1 GB. Due to the limited memory in early GPUs, the original AlexNet used a dual data stream design, so that each of their two GPUs could be responsible for storing and computing only its half of the model. Fortunately, GPU memory is comparatively abundant now, so we rarely need to break up models across GPUs these days (our version of the AlexNet model deviates from the original paper in this aspect).

Activation Functions

Second, AlexNet changed the sigmoid activation function to a simpler ReLU activation function. On the one hand, the computation of the ReLU activation function is simpler. For example, it does not have the exponentiation operation found in the sigmoid activation function. On the other hand, the ReLU activation function makes model training easier when using different parameter initialization methods. This is because, when the output of the sigmoid activation function is very close to 0 or 1, the gradient of these regions is almost 0, so that back propagation cannot continue to update some of the model parameters. In contrast, the gradient of the ReLU activation function in the positive interval is always 1. Therefore, if the model parameters are not properly initialized, the sigmoid function may obtain a gradient of almost 0 in the positive interval, so that the model cannot be effectively trained.

Capacity Control and Preprocessing

AlexNet controls the model complexity of the fully-connected layer by dropout (Section 4.6), while LeNet only uses weight decay. To augment the data even further, the training loop of AlexNet added a great deal of image augmentation, such as flipping, clipping, and color changes. This makes the model more robust and the larger sample size effectively reduces overfitting. We will discuss data augmentation in greater detail in Section 13.1.

```
import d2l
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()

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
net.add(nn.Conv2D(96, kernel_size=11, strides=4, activation='relu'),
        nn.MaxPool2D(pool_size=3, strides=2),
        # Make the convolution window smaller, set padding to 2 for consistent
```

(continues on next page)

(continued from previous page)

```
        # height and width across the input and output, and increase the
        # number of output channels
        nn.Conv2D(256, kernel_size=5, padding=2, activation='relu'),
        nn.MaxPool2D(pool_size=3, strides=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(384, kernel_size=3, padding=1, activation='relu'),
        nn.Conv2D(384, kernel_size=3, padding=1, activation='relu'),
        nn.Conv2D(256, kernel_size=3, padding=1, activation='relu'),
        nn.MaxPool2D(pool_size=3, strides=2),
        # 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.Dense(4096, activation="relu"), nn.Dropout(0.5),
        nn.Dense(4096, activation="relu"), nn.Dropout(0.5),
        # Output layer. Since we are using Fashion-MNIST, the number of
        # classes is 10, instead of 1000 as in the paper
        nn.Dense(10))
```

We construct a single-channel data instance with both height and width of 224 to observe the output shape of each layer. It matches our diagram above.

```
X = np.random.uniform(size=(1, 1, 224, 224))
net.initialize()
for layer in net:
    X = layer(X)
    print(layer.name, 'output shape:\t', X.shape)
```

```
conv0 output shape:  (1, 96, 54, 54)
pool0 output shape:  (1, 96, 26, 26)
conv1 output shape:  (1, 256, 26, 26)
pool1 output shape:  (1, 256, 12, 12)
conv2 output shape:  (1, 384, 12, 12)
conv3 output shape:  (1, 384, 12, 12)
conv4 output shape:  (1, 256, 12, 12)
pool2 output shape:  (1, 256, 5, 5)
dense0 output shape:         (1, 4096)
dropout0 output shape:       (1, 4096)
dense1 output shape:         (1, 4096)
dropout1 output shape:       (1, 4096)
dense2 output shape:         (1, 10)
```

7.1.3 Reading the Dataset

Although AlexNet uses ImageNet in the paper, we use Fashion-MNIST here since training an ImageNet model to convergence could take hours or days even on a modern GPU. One of the problems with applying AlexNet directly on Fashion-MNIST is that our images are lower resolution (28×28 pixels) than ImageNet images. To make things work, we upsample them to 244×244 (generally not a smart practice, but we do it here to be faithful to the AlexNet architecture). We perform this resizing with the `resize` argument in `load_data_fashion_mnist`.

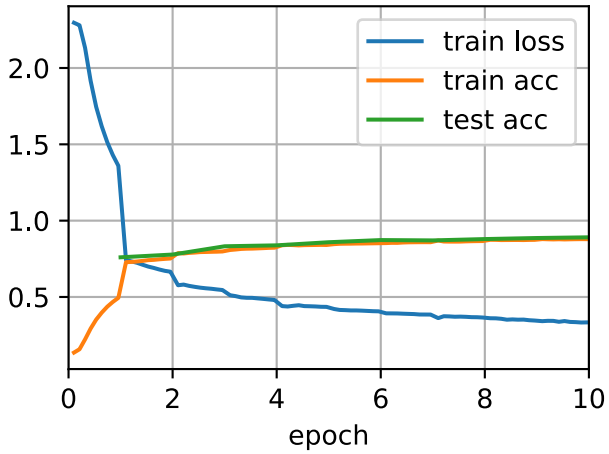
```
batch_size = 128
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)
```

7.1.4 Training

Now, we can start training AlexNet. Compared to LeNet in the previous section, the main change here is the use of a smaller learning rate and much slower training due to the deeper and wider network, the higher image resolution and the more costly convolutions.

```
lr, num_epochs = 0.01, 10
d2l.train_ch5(net, train_iter, test_iter, num_epochs, lr)
```

```
loss 0.333, train acc 0.879, test acc 0.891
4206.3 exampes/sec on gpu(0)
```



Summary

- AlexNet has a similar structure to that of LeNet, but uses more convolutional layers and a larger parameter space to fit the large-scale dataset ImageNet.
- Today AlexNet has been surpassed by much more effective architectures but it is a key step from shallow to deep networks that are used nowadays.
- Although it seems that there are only a few more lines in AlexNet's implementation than in LeNet, it took the academic community many years to embrace this conceptual change and

take advantage of its excellent experimental results. This was also due to the lack of efficient computational tools.

- Dropout, ReLU and preprocessing were the other key steps in achieving excellent performance in computer vision tasks.

Exercises

1. Try increasing the number of epochs. Compared with LeNet, how are the results different? Why?
2. AlexNet may be too complex for the Fashion-MNIST dataset.
   - Try to simplify the model to make the training faster, while ensuring that the accuracy does not drop significantly.
   - Can you design a better model that works directly on 28×28 images.
3. Modify the batch size, and observe the changes in accuracy and GPU memory.
4. Rooflines:
   - What is the dominant part for the memory footprint of AlexNet?
   - What is the dominant part for computation in AlexNet?
   - How about memory bandwidth when computing the results?
5. Apply dropout and ReLU to LeNet5. Does it improve? How about preprocessing?

7.2 Networks Using Blocks (VGG)

While AlexNet proved that deep convolutional neural networks can achieve good results, it did not offer 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 in chip design where engineers went from placing transistors to logical elements to logic blocks. Similarly, the design of neural network architectures had 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[113] (VGG) at Oxford University. In their eponymously-named VGG network, It is easy to implement these repeated structures in code with any modern deep learning framework by using loops and subroutines.

[113] http://www.robots.ox.ac.uk/~vgg/

7.2.1 VGG Blocks

The basic building block of classic convolutional networks is a sequence of the following layers: (i) a convolutional layer (with padding to maintain the resolution), (ii) a nonlinearity such as a ReLU, One VGG block consists of a sequence of convolutional layers, followed by a max pooling layer for spatial downsampling. In the original VGG paper (Simonyan & Zisserman, 2014), the authors employed convolutions with 3×3 kernels and 2×2 max pooling with stride of 2 (halving the resolution after each block). In the code below, we define a function called `vgg_block` to implement one VGG block. The function takes two arguments corresponding to the number of convolutional layers `num_convs` and the number of output channels `num_channels`.

```
import d2l
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()

def vgg_block(num_convs, num_channels):
    blk = nn.Sequential()
    for _ in range(num_convs):
        blk.add(nn.Conv2D(num_channels, kernel_size=3,
                          padding=1, activation='relu'))
    blk.add(nn.MaxPool2D(pool_size=2, strides=2))
    return blk
```

7.2.2 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 a second consisting of fully-connected layers. The convolutional portion of the net connects several `vgg_block` modules in succession. In Fig. 7.2.1, the variable `conv_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 requires to call the `vgg_block` function. The fully-connected module is identical to that covered in AlexNet.

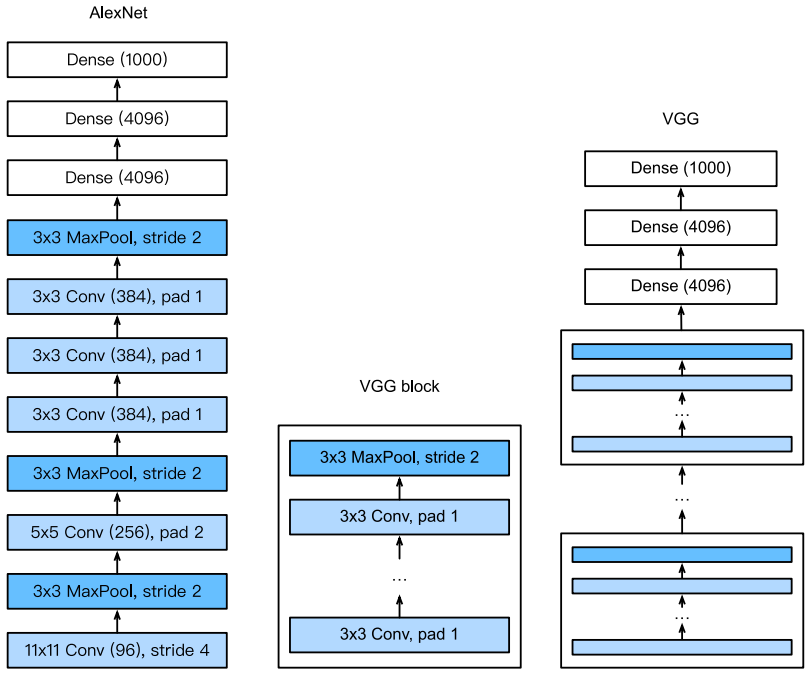


Fig. 7.2.1: Designing a network from building blocks

The original VGG network had 5 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 8 convolutional layers and 3 fully-connected layers, it is often called VGG-11.

```
conv_arch = ((1, 64), (1, 128), (2, 256), (2, 512), (2, 512))
```

The following code implements VGG-11. This is a simple matter of executing a for loop over `conv_arch`.

```
def vgg(conv_arch):
    net = nn.Sequential()
    # The convolutional layer part
    for (num_convs, num_channels) in conv_arch:
        net.add(vgg_block(num_convs, num_channels))
    # The fully connected layer part
    net.add(nn.Dense(4096, activation='relu'), nn.Dropout(0.5),
            nn.Dense(4096, activation='relu'), nn.Dropout(0.5),
            nn.Dense(10))
    return net

net = vgg(conv_arch)
```

Next, we will construct a single-channel data example with a height and width of 224 to observe the output shape of each layer.

```
net.initialize()
X = np.random.uniform(size=(1, 1, 224, 224))
for blk in net:
```

(continues on next page)

(continued from previous page)

```
    X = blk(X)
    print(blk.name, 'output shape:\t', X.shape)
```

```
sequential1 output shape:    (1, 64, 112, 112)
sequential2 output shape:    (1, 128, 56, 56)
sequential3 output shape:    (1, 256, 28, 28)
sequential4 output shape:    (1, 512, 14, 14)
sequential5 output shape:    (1, 512, 7, 7)
dense0 output shape:         (1, 4096)
dropout0 output shape:       (1, 4096)
dense1 output shape:         (1, 4096)
dropout1 output shape:       (1, 4096)
dense2 output shape:         (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 layer.

7.2.3 Model Training

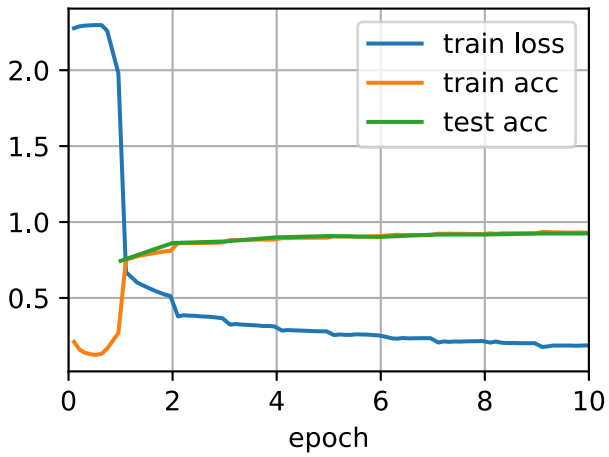
Since VGG-11 is more computationally-heavy than AlexNet we construct a network with a smaller number of channels. This is more than sufficient for training on Fashion-MNIST.

```
ratio = 4
small_conv_arch = [(pair[0], pair[1] // ratio) for pair in conv_arch]
net = vgg(small_conv_arch)
```

Apart from using a slightly larger learning rate, the model training process is similar to that of AlexNet in the last section.

```
lr, num_epochs, batch_size = 0.05, 10, 128,
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)
d2l.train_ch5(net, train_iter, test_iter, num_epochs, lr)
```

```
loss 0.186, train acc 0.931, test acc 0.925
1815.7 exampes/sec on gpu(0)
```



Summary

- VGG-11 constructs a network using reusable convolutional blocks. Different VGG models can be defined by the differences in the number of convolutional layers and output channels in each block.
- The use of blocks leads to very compact representations of the network definition. It allows for efficient design of complex networks.
- In their work Simonyan and Ziserman experimented with various architectures. In particular, they found that several layers of deep and narrow convolutions (i.e., 3×3) were more effective than fewer layers of wider convolutions.

Exercises

1. When printing out the dimensions of the layers we only saw 8 results rather than 11. Where did the remaining 3 layer informations go?
2. Compared with AlexNet, VGG is much slower in terms of computation, and it also needs more GPU memory. Try to analyze the reasons for this.
3. Try to change the height and width of the images in Fashion-MNIST from 224 to 96. What influence does this have on the experiments?
4. Refer to Table 1 in (Simonyan & Zisserman, 2014) to construct other common models, such as VGG-16 or VGG-19.

7.3 Network in Network (NiN)

LeNet, AlexNet, and VGG all share a common design pattern: extract features exploiting *spatial* structure via a sequence of convolutions 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. However, a careless use of dense layers might give up the spatial structure of the representation entirely, Network in Network (NiN) blocks offer an alternative. They were proposed in (Lin et al., 2013) based on a very simple insight—to use an MLP on the channels for each pixel separately.

7.3.1 NiN Blocks

Recall that the inputs and outputs of convolutional layers consist of four-dimensional arrays with axes corresponding to the batch, channel, height, and width. Also recall that the inputs and outputs of fully-connected layers are typically two-dimensional arrays corresponding to the batch, and features. The idea behind NiN is to apply a fully-connected layer at each pixel location (for each height and width). If we tie the weights across each spatial location, we could think of this as a 1×1 convolutional layer (as described in Section 6.4) or as a fully-connected layer acting independently on each pixel location. Another way to view this is to think of each element in the spatial dimension (height and width) as equivalent to an example and the channel as equivalent to a feature. Fig. 7.3.1 illustrates the main structural differences between NiN and AlexNet, VGG, and other networks.

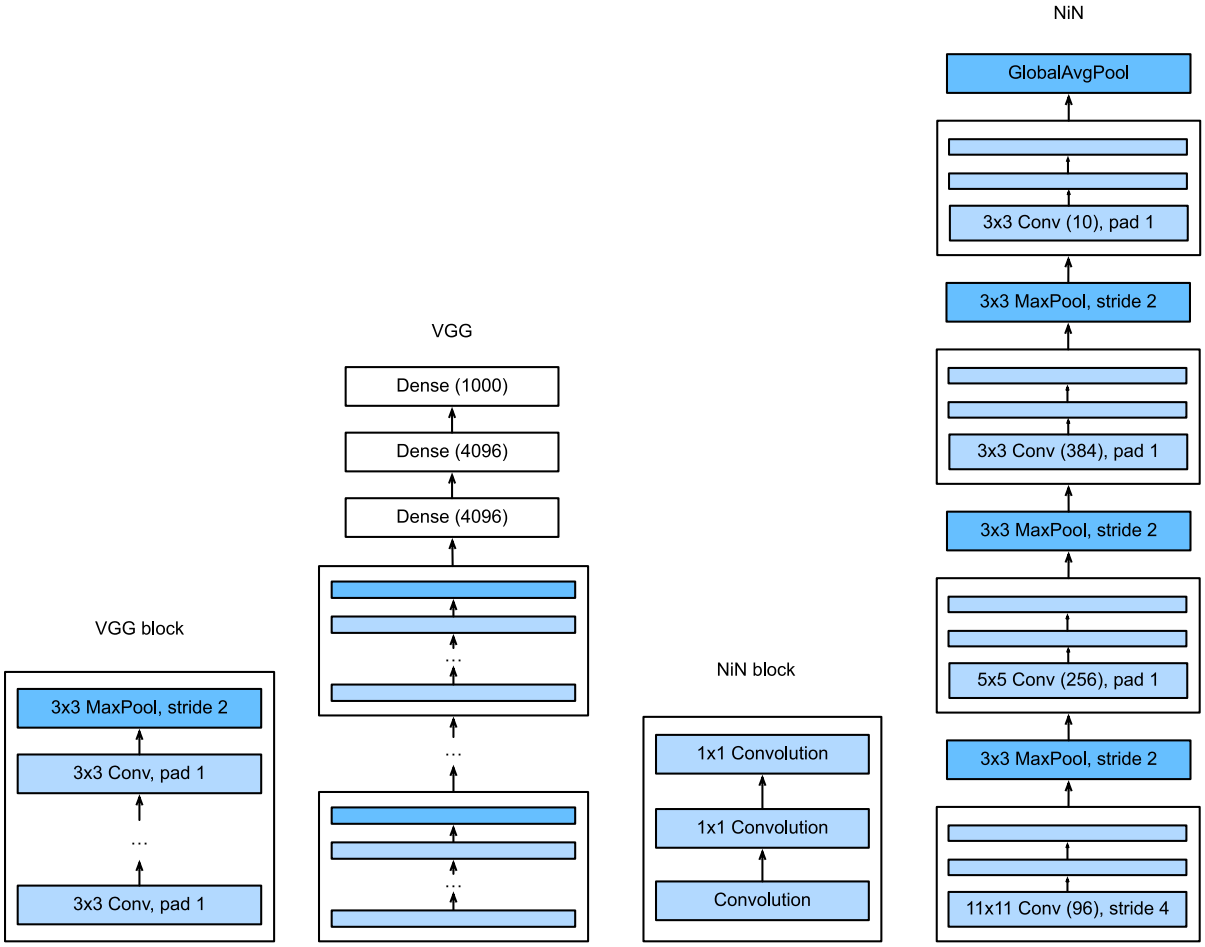


Fig. 7.3.1: The figure on the left shows the network structure of AlexNet and VGG, and the figure on the right shows the network structure of NiN.

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. The convolution width of the first layer is typically set by the user. The subsequent widths are fixed to 1×1.

```
import d2l
from mxnet import np, npx
from mxnet.gluon import nn
```

(continues on next page)

(continued from previous page)

```
npx.set_np()

def nin_block(num_channels, kernel_size, strides, padding):
    blk = nn.Sequential()
    blk.add(nn.Conv2D(num_channels, kernel_size, strides, padding,
                      activation='relu'),
            nn.Conv2D(num_channels, kernel_size=1, activation='relu'),
            nn.Conv2D(num_channels, kernel_size=1, activation='relu'))
    return blk
```

7.3.2 NiN Model

The original NiN network was proposed shortly after AlexNet and clearly draws some inspiration. NiN uses convolutional layers with window shapes of 11×11, 5×5, and 3×3, and the corresponding numbers of output channels are the same as in AlexNet. Each NiN block is followed by a maximum pooling layer with a stride of 2 and a window shape of 3×3.

Once significant difference between NiN and AlexNet is that NiN avoids dense connections altogether. Instead, NiN uses an NiN block with a number of output channels equal to the number of label classes, followed by a *global* average pooling layer, yielding a vector of logits[115]. 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.

```
net = nn.Sequential()
net.add(nin_block(96, kernel_size=11, strides=4, padding=0),
        nn.MaxPool2D(pool_size=3, strides=2),
        nin_block(256, kernel_size=5, strides=1, padding=2),
        nn.MaxPool2D(pool_size=3, strides=2),
        nin_block(384, kernel_size=3, strides=1, padding=1),
        nn.MaxPool2D(pool_size=3, strides=2),
        nn.Dropout(0.5),
        # There are 10 label classes
        nin_block(10, kernel_size=3, strides=1, padding=1),
        # The global average pooling layer automatically sets the window shape
        # to the height and width of the input
        nn.GlobalAvgPool2D(),
        # Transform the four-dimensional output into two-dimensional output
        # with a shape of (batch size, 10)
        nn.Flatten())
```

We create a data example to see the output shape of each block.

```
X = np.random.uniform(size=(1, 1, 224, 224))
net.initialize()
for layer in net:
    X = layer(X)
    print(layer.name, 'output shape:\t', X.shape)
```

[115] https://en.wikipedia.org/wiki/Logit

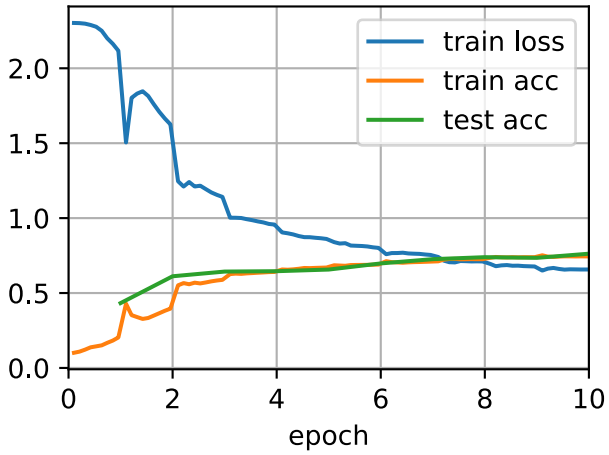
```
sequential1 output shape:    (1, 96, 54, 54)
pool0 output shape:  (1, 96, 26, 26)
sequential2 output shape:    (1, 256, 26, 26)
pool1 output shape:  (1, 256, 12, 12)
sequential3 output shape:    (1, 384, 12, 12)
pool2 output shape:  (1, 384, 5, 5)
dropout0 output shape:       (1, 384, 5, 5)
sequential4 output shape:    (1, 10, 5, 5)
pool3 output shape:  (1, 10, 1, 1)
flatten0 output shape:       (1, 10)
```

7.3.3 Data Acquisition and Training

As before we use Fashion-MNIST to train the model. NiN's training is similar to that for AlexNet and VGG, but it often uses a larger learning rate.

```
lr, num_epochs, batch_size = 0.1, 10, 128
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)
d2l.train_ch5(net, train_iter, test_iter, num_epochs, lr)
```

```
loss 0.657, train acc 0.744, test acc 0.762
3024.6 exampes/sec on gpu(0)
```



Summary

- NiN uses blocks consisting of a convolutional layer and multiple 1×1 convolutional layer. This can be used within the convolutional stack to allow for more per-pixel nonlinearity.
- NiN removes the fully connected layers and replaces them with global average pooling (i.e., summing over all locations) after reducing the number of channels to the desired number of outputs (e.g., 10 for Fashion-MNIST).
- Removing the dense layers reduces overfitting. NiN has dramatically fewer parameters.
- The NiN design influenced many subsequent convolutional neural networks designs.

Exercises

1. Tune the hyper-parameters to improve the classification accuracy.
2. Why are there two 1×1 convolutional layers in the NiN block? Remove one of them, and then observe and analyze the experimental phenomena.
3. Calculate the resource usage for NiN
    - What is the number of parameters?
    - What is the amount of computation?
    - What is the amount of memory needed during training?
    - What is the amount of memory needed during inference?
4. What are possible problems with reducing the $384 \times 5 \times 5$ representation to a $10 \times 5 \times 5$ representation in one step?

7.4 Networks with Parallel Concatenations (GoogLeNet)

In 2014, (Szegedy et al., 2015) won the ImageNet Challenge, proposing a structure that combined the strengths of the NiN and repeated blocks paradigms. One focus of the paper was to address the question of which sized convolutional kernels are best. After all, previous popular networks employed choices as small as 1×1 and as large as 11×11. One insight in this paper was that sometimes it can be advantageous to employ a combination of variously-sized kernels. In this section, we will introduce GoogLeNet, presenting a slightly simplified version of the original model—we omit a few ad hoc features that were added to stabilize training but are unnecessary now with better training algorithms available.

7.4.1 Inception Blocks

The basic convolutional block in GoogLeNet is called an Inception block, likely named due to a quote from the movie Inception ("We Need To Go Deeper"), which launched a viral meme.

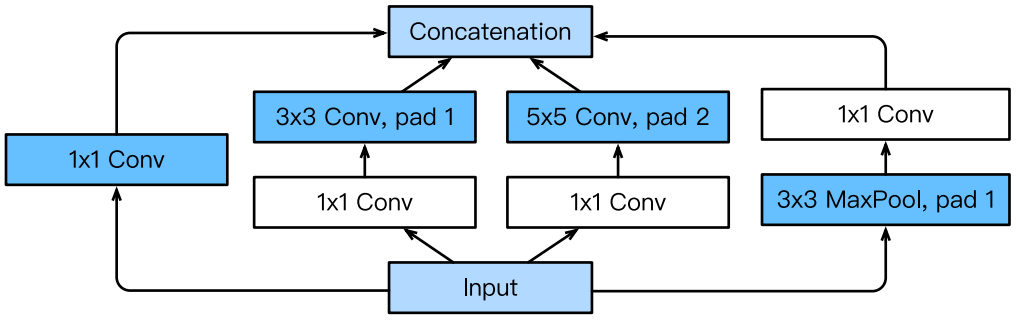


Fig. 7.4.1: Structure of the Inception block.

As depicted in the figure above, the inception block consists of four parallel paths. The first three paths use convolutional layers with window sizes of 1×1, 3×3, and 5×5 to extract information from different spatial sizes. The middle two paths perform a 1×1 convolution on the input to reduce the number of input channels, reducing the model's complexity. The fourth path uses a 3×3 maximum pooling layer, followed by a 1×1 convolutional layer to change the number of channels. The four paths all use appropriate padding to give the input and output the same height and width. Finally, the outputs along each path are concatenated along the channel dimension and comprise the block's output. The commonly-tuned parameters of the Inception block are the number of output channels per layer.

```
import d2l
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()

class Inception(nn.Block):
    # c1 - c4 are the number of output channels for each layer in the path
    def __init__(self, c1, c2, c3, c4, **kwargs):
        super(Inception, self).__init__(**kwargs)
        # Path 1 is a single 1 x 1 convolutional layer
        self.p1_1 = nn.Conv2D(c1, kernel_size=1, activation='relu')
        # Path 2 is a 1 x 1 convolutional layer followed by a 3 x 3
        # convolutional layer
        self.p2_1 = nn.Conv2D(c2[0], kernel_size=1, activation='relu')
        self.p2_2 = nn.Conv2D(c2[1], kernel_size=3, padding=1,
                              activation='relu')
        # Path 3 is a 1 x 1 convolutional layer followed by a 5 x 5
        # convolutional layer
        self.p3_1 = nn.Conv2D(c3[0], kernel_size=1, activation='relu')
        self.p3_2 = nn.Conv2D(c3[1], kernel_size=5, padding=2,
                              activation='relu')
        # Path 4 is a 3 x 3 maximum pooling layer followed by a 1 x 1
        # convolutional layer
        self.p4_1 = nn.MaxPool2D(pool_size=3, strides=1, padding=1)
        self.p4_2 = nn.Conv2D(c4, kernel_size=1, activation='relu')

    def forward(self, x):
        p1 = self.p1_1(x)
        p2 = self.p2_2(self.p2_1(x))
        p3 = self.p3_2(self.p3_1(x))
        p4 = self.p4_2(self.p4_1(x))
        # Concatenate the outputs on the channel dimension
        return np.concatenate((p1, p2, p3, p4), axis=1)
```

To gain some intuition for why this network works so well, consider the combination of the filters. They explore the image in varying ranges. This means that details at different extents can be recognized efficiently by different filters. At the same time, we can allocate different amounts of parameters for different ranges (e.g., more for short range but not ignore the long range entirely).

7.4.2 GoogLeNet Model

As shown in Fig. 7.4.2, GoogLeNet uses a stack of a total of 9 inception blocks and global average pooling to generate its estimates. Maximum pooling between inception blocks reduced the dimensionality. The first part is identical 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. The architecture is depicted below.

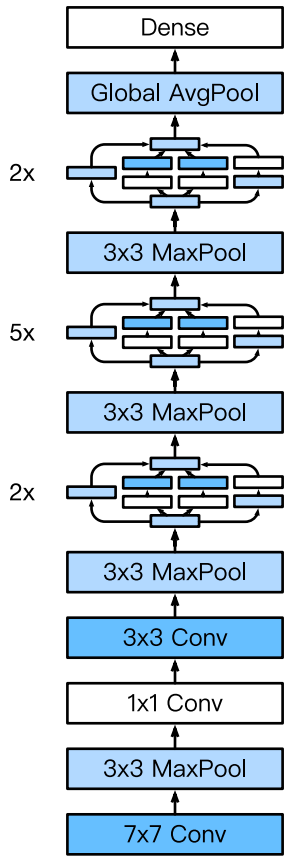


Fig. 7.4.2: Full GoogLeNet Model

We can now implement GoogLeNet piece by piece. The first component uses a 64-channel 7×7 convolutional layer.

```
b1 = nn.Sequential()
b1.add(nn.Conv2D(64, kernel_size=7, strides=2, padding=3, activation='relu'),
       nn.MaxPool2D(pool_size=3, strides=2, padding=1))
```

The second component uses two convolutional layers: first, a 64-channel 1×1 convolutional layer, then a 3×3 convolutional layer that triples the number of channels. This corresponds to the second path in the Inception block.

```
b2 = nn.Sequential()
b2.add(nn.Conv2D(64, kernel_size=1, activation='relu'),
       nn.Conv2D(192, kernel_size=3, padding=1, activation='relu'),
       nn.MaxPool2D(pool_size=3, strides=2, padding=1))
```

The third component connects two complete Inception blocks in series. The number of output channels of the first Inception block is $64 + 128 + 32 + 32 = 256$, and the ratio to the output channels of the four paths is $64 : 128 : 32 : 32 = 2 : 4 : 1 : 1$. The second and third paths first reduce the number of input channels to $96/192 = 1/2$ and $16/192 = 1/12$, respectively, and then connect the second convolutional layer. The number of output channels of the second Inception block is increased to $128 + 192 + 96 + 64 = 480$, and the ratio to the number of output channels per path is $128 : 192 : 96 : 64 = 4 : 6 : 3 : 2$. The second and third paths first reduce the number of input channels to $128/256 = 1/2$ and $32/256 = 1/8$, respectively.

```
b3 = nn.Sequential()
b3.add(Inception(64, (96, 128), (16, 32), 32),
       Inception(128, (128, 192), (32, 96), 64),
       nn.MaxPool2D(pool_size=3, strides=2, padding=1))
```

The fourth block is more complicated. It connects five Inception blocks in series, and they have $192+208+48+64 = 512$, $160+224+64+64 = 512$, $128+256+64+64 = 512$, $112+288+64+64 = 528$, and $256+320+128+128 = 832$ output channels, respectively. The number of channels assigned to these paths is similar to that in the third module: the second path with the 3×3 convolutional layer outputs the largest number of channels, followed by the first path with only the 1×1 convolutional layer, the third path with the 5×5 convolutional layer, and the fourth path with the 3×3 maximum pooling layer. The second and third paths will first reduce the number of channels according the ratio. These ratios are slightly different in different Inception blocks.

```
b4 = nn.Sequential()
b4.add(Inception(192, (96, 208), (16, 48), 64),
       Inception(160, (112, 224), (24, 64), 64),
       Inception(128, (128, 256), (24, 64), 64),
       Inception(112, (144, 288), (32, 64), 64),
       Inception(256, (160, 320), (32, 128), 128),
       nn.MaxPool2D(pool_size=3, strides=2, padding=1))
```

The fifth block has two Inception blocks with $256+320+128+128 = 832$ and $384+384+128+128 = 1024$ output channels. The number of channels assigned to each path is the same as that in the third and fourth modules, but differs in specific values. It should be noted that the fifth block is followed by the output layer. This block uses the global average pooling layer to change the height and width of each channel to 1, just as in NiN. Finally, we turn the output into a two-dimensional array followed by a fully-connected layer whose number of outputs is the number of label classes.

```
b5 = nn.Sequential()
b5.add(Inception(256, (160, 320), (32, 128), 128),
       Inception(384, (192, 384), (48, 128), 128),
       nn.GlobalAvgPool2D())

net = nn.Sequential()
net.add(b1, b2, b3, b4, b5, nn.Dense(10))
```

The GoogLeNet model is computationally complex, so it is not as easy to modify the number of channels as in VGG. To have a reasonable training time on Fashion-MNIST, we reduce the input height and width from 224 to 96. This simplifies the computation. The changes in the shape of the output between the various modules is demonstrated below.

```
X = np.random.uniform(size=(1, 1, 96, 96))
net.initialize()
for layer in net:
    X = layer(X)
    print(layer.name, 'output shape:\t', X.shape)
```

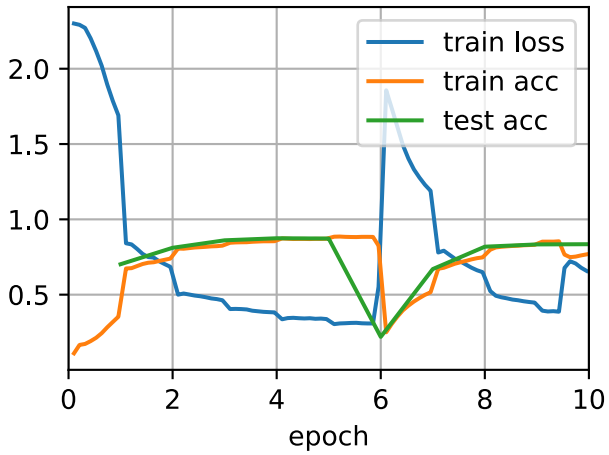
```
sequential0 output shape:    (1, 64, 24, 24)
sequential1 output shape:    (1, 192, 12, 12)
sequential2 output shape:    (1, 480, 6, 6)
sequential3 output shape:    (1, 832, 3, 3)
sequential4 output shape:    (1, 1024, 1, 1)
dense0 output shape:         (1, 10)
```

7.4.3 Data Acquisition and Training

As before, we train our model using the Fashion-MNIST dataset. We transform it to 96×96 pixel resolution before invoking the training procedure.

```
lr, num_epochs, batch_size = 0.1, 10, 128
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=96)
d2l.train_ch5(net, train_iter, test_iter, num_epochs, lr)
```

```
loss 0.648, train acc 0.770, test acc 0.835
2525.2 exampes/sec on gpu(0)
```



Summary

- The Inception block is equivalent to a subnetwork with four paths. It extracts information in parallel through convolutional layers of different window shapes and maximum pooling layers. 1×1 convolutions reduce channel dimensionality on a per-pixel level. Max-pooling reduces the resolution.
- GoogLeNet connects multiple well-designed Inception blocks with other layers in series. The ratio of the number of channels assigned in the Inception block is obtained through a large number of experiments on the ImageNet dataset.

- GoogLeNet, as well as its succeeding versions, was one of the most efficient models on ImageNet, providing similar test accuracy with lower computational complexity.

Exercises

1. There are several iterations of GoogLeNet. Try to implement and run them. Some of them include the following:
   - Add a batch normalization layer (Ioffe & Szegedy, 2015), as described later in Section 7.5.
   - Make adjustments to the Inception block (Szegedy et al., 2016).
   - Use "label smoothing" for model regularization (Szegedy et al., 2016).
   - Include it in the residual connection (Szegedy et al., 2017), as described later in Section 7.6.
2. What is the minimum image size for GoogLeNet to work?
3. Compare the model parameter sizes of AlexNet, VGG, and NiN with GoogLeNet. How do the latter two network architectures significantly reduce the model parameter size?
4. Why do we need a large range convolution initially?

7.5 Batch Normalization

Training deep neural nets is difficult. And getting them to converge in a reasonable amount of time can be tricky.
In this section, we describe batch normalization (BN) (Ioffe & Szegedy, 2015), a popular and effective technique that consistently accelerates the convergence of deep nets. Together with residual blocks—covered in Section 7.6—BN has made it possible for practitioners to routinely train networks with over 100 layers.

7.5.1 Training Deep Networks

To motivate batch normalization, let's review a few practical challenges that arise when training ML models and neural nets in particular.

1. Choices regarding data preprocessing often make an enormous difference in the final results. Recall our application of multilayer perceptrons to predicting house prices (Section 4.10). Our first step when working with real data was to standardize our input features to each have a mean of *zero* and variance of *one*. Intuitively, this standardization plays nicely with our optimizers because it puts the parameters are a-priori at a similar scale.
2. For a typical MLP or CNN, as we train, the activations in intermediate layers may take values with widely varying magnitudes—both along the layers from the input to the output, across

nodes in the same layer, and over time due to our updates to the model's parameters. The inventors of batch normalization postulated informally that this drift in the distribution of activations could hamper the convergence of the network. Intuitively, we might conjecture that if one layer has activation values that are 100x that of another layer, this might necessitate compensatory adjustments in the learning rates.

3. Deeper networks are complex and easily capable of overfitting. This means that regularization becomes more critical.

Batch normalization is applied to individual layers (optionally, to all of them) and works as follows: In each training iteration, for each layer, we first compute its activations as usual. Then, we normalize the activations of each node by subtracting its mean and dividing by its standard deviation estimating both quantities based on the statistics of the current the current minibatch. It is precisely due to this *normalization* based on *batch* statistics that *batch normalization* derives its name.

Note that if we tried to apply BN with minibatches of size 1, we would not be able to learn anything. That is because after subtracting the means, each hidden node would take value 0! As you might guess, since we are devoting a whole section to BN, with large enough minibatches, the approach proves effective and stable. One takeaway here is that when applying BN, the choice of minibatch size may be even more significant than without BN.

Formally, BN transforms the activations at a given layer $\mathbf{x}$ according to the following expression:

$$\mathrm{BN}(\mathbf{x}) = \gamma \odot \frac{\mathbf{x} - \hat{\mu}}{\hat{\sigma}} + \beta \tag{7.5.1}$$

Here, $\hat{\mu}$ is the minibatch sample mean and $\hat{\sigma}$ is the minibatch sample variance. After applying BN, the resulting minibatch of activations has zero mean and unit variance. Because the choice of unit variance (vs some other magic number) is an arbitrary choice, we commonly include coordinate-wise scaling coefficients γ and offsets β. Consequently, the activation magnitudes for intermediate layers cannot diverge during training because BN actively centers and rescales them back to a given mean and size (via μ and σ). One piece of practitioner's intuition/wisdom is that BN seems to allows for more aggressive learning rates.

Formally, denoting a particular minibatch by $\mathcal{B}$, we calculate $\hat{\mu}_\mathcal{B}$ and $\hat{\sigma}_\mathcal{B}$ as follows:

$$\hat{\mu}_\mathcal{B} \leftarrow \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} \mathbf{x} \text{ and } \hat{\sigma}^2_\mathcal{B} \leftarrow \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} (\mathbf{x} - \mu_\mathcal{B})^2 + \epsilon \tag{7.5.2}$$

Note that we add a small constant $\epsilon > 0$ to the variance estimate to ensure that we never attempt division by zero, even in cases where the empirical variance estimate might vanish. The estimates $\hat{\mu}_\mathcal{B}$ and $\hat{\sigma}_\mathcal{B}$ counteract the scaling issue by using noisy estimates of mean and variance. You might think that this noisiness should be a problem. As it turns out, this is actually beneficial.

This turns out to be a recurring theme in deep learning. For reasons that are not yet well-characterized theoretically, various sources of noise in optimization often lead to faster training and less overfitting. While traditional machine learning theorists might buckle at this characterization, this variation appears to act as a form of regularization. In some preliminary research, (Teye et al., 2018) and (Luo et al., 2018) relate the properties of BN to Bayesian Priors and penalties respectively. In particular, this sheds some light on the puzzle of why BN works best for moderate minibatches sizes in the 50–100 range.

Fixing a trained model, you might (rightly) think that we would prefer to use the entire dataset to estimate the mean and variance. Once training is complete, why would we want the same image to be classified differently, depending on the batch in which it happens to reside? During training, such exact calculation is infeasible because the activations for all data points change every time we update our model. However, once the model is trained, we can calculate the means and variances of each layer's activations based on the entire dataset. Indeed this is standard practice for models employing batch normalization and thus MXNet's BN layers function differently in *training mode* (normalizing by minibatch statistics) and in *prediction mode* (normalizing by dataset statistics).

We are now ready to take a look at how batch normalization works in practice.

7.5.2 Batch Normalization Layers

Batch normalization implementations for fully-connected layers and convolutional layers are slightly different. We discuss both cases below. Recall that one key differences between BN and other layers is that because BN operates on a full minibatch at a time, we cannot just ignore the batch dimension as we did before when introducing other layers.

Fully-Connected Layers

When applying BN to fully-connected layers, we usually insert BN after the affine transformation and before the nonlinear activation function. Denoting the input to the layer by $\mathbf{x}$, the linear transform (with weights θ) by $f_\theta(\cdot)$, the activation function by $\phi(\cdot)$, and the BN operation with parameters β and γ by $\text{BN}_{\beta,\gamma}$, we can express the computation of a BN-enabled, fully-connected layer $\mathbf{h}$ as follows:

$$\mathbf{h} = \phi(\text{BN}_{\beta,\gamma}(f_\theta(\mathbf{x}))) \tag{7.5.3}$$

Recall that mean and variance are computed on the *same* minibatch $\mathcal{B}$ on which the transformation is applied. Also recall that the scaling coefficient γ and the offset β are parameters that need to be learned jointly with the more familiar parameters θ.

Convolutional Layers

Similarly, with convolutional layers, we typically apply BN after the convolution and before the nonlinear activation function. When the convolution has multiple output channels, we need to carry out batch normalization for *each* of the outputs of these channels, and each channel has its own scale and shift parameters, both of which are scalars. Assume that our minibatches contain m each and that for each channel, the output of the convolution has height p and width q. For convolutional layers, we carry out each batch normalization over the $m \cdot p \cdot q$ elements per output channel simultaneously. Thus we collect the values over all spatial locations when computing the mean and variance and consequently (within a given channel) apply the same $\hat{\mu}$ and $\hat{\sigma}$ to normalize the values at each spatial location.

Batch Normalization During Prediction

As we mentioned earlier, BN typically behaves differently in training mode and prediction mode. First, the noise in μ and σ arising from estimating each on minibatches are no longer desirable once we have trained the model. Second, we might not have the luxury of computing per-batch normalization statistics, e.g., we might need to apply our model to make one prediction at a time.

Typically, after training, we use the entire dataset to compute stable estimates of the activation statistics and then fix them at prediction time. Consequently, BN behaves differently during training and at test time. Recall that dropout also exhibits this characteristic.

7.5.3 Implementation from Scratch

Below, we implement a batch normalization layer with `ndarrays` from scratch:

```
import d2l
from mxnet import autograd, np, npx, init
from mxnet.gluon import nn
npx.set_np()

def batch_norm(X, gamma, beta, moving_mean, moving_var, eps, momentum):
    # Use autograd to determine whether the current mode is training mode or
    # prediction mode
    if not autograd.is_training():
        # If it is the prediction mode, directly use the mean and variance
        # obtained from the incoming moving average
        X_hat = (X - moving_mean) / np.sqrt(moving_var + eps)
    else:
        assert len(X.shape) in (2, 4)
        if len(X.shape) == 2:
            # When using a fully connected layer, calculate the mean and
            # variance on the feature dimension
            mean = X.mean(axis=0)
            var = ((X - mean) ** 2).mean(axis=0)
        else:
            # When using a two-dimensional convolutional layer, calculate the
            # mean and variance on the channel dimension (axis=1). Here we
            # need to maintain the shape of X, so that the broadcast operation
            # can be carried out later
            mean = X.mean(axis=(0, 2, 3), keepdims=True)
            var = ((X - mean) ** 2).mean(axis=(0, 2, 3), keepdims=True)
        # In training mode, the current mean and variance are used for the
        # standardization
        X_hat = (X - mean) / np.sqrt(var + eps)
        # Update the mean and variance of the moving average
        moving_mean = momentum * moving_mean + (1.0 - momentum) * mean
        moving_var = momentum * moving_var + (1.0 - momentum) * var
    Y = gamma * X_hat + beta  # Scale and shift
    return Y, moving_mean, moving_var
```

We can now create a proper `BatchNorm` layer. Our layer will maintain poper parameters corresponding for scale `gamma` and shift `beta`, both of which will be updated in the course of training. Additionally, our layer will maintain a moving average of the means and variances for subsequent use during model prediction. The `num_features` parameter required by the `BatchNorm` instance is

the number of outputs for a fully-connected layer and the number of output channels for a convolutional layer. The `num_dims` parameter also required by this instance is 2 for a fully-connected layer and 4 for a convolutional layer.

Putting aside the algorithmic details, note the design pattern underlying our implementation of the layer. Typically, we define the math in a separate function, say `batch_norm`. We then integrate this functionality into a custom layer, whose code mostly addresses bookkeeping matters, such as moving data to the right device context, allocating and initializing any required variables, keeping track of running averages (here for mean and variance), etc. This pattern enables a clean separation of math from boilerplate code. Also note that for the sake of convenience we did not worry about automatically inferring the input shape here, thus our need to specify the number of features throughout. Do not worry, the Gluon `BatchNorm` layer will care of this for us.

```
class BatchNorm(nn.Block):
    def __init__(self, num_features, num_dims, **kwargs):
        super(BatchNorm, self).__init__(**kwargs)
        if num_dims == 2:
            shape = (1, num_features)
        else:
            shape = (1, num_features, 1, 1)
        # The scale parameter and the shift parameter involved in gradient
        # finding and iteration are initialized to 0 and 1 respectively
        self.gamma = self.params.get('gamma', shape=shape, init=init.One())
        self.beta = self.params.get('beta', shape=shape, init=init.Zero())
        # All the variables not involved in gradient finding and iteration are
        # initialized to 0 on the CPU
        self.moving_mean = np.zeros(shape)
        self.moving_var = np.zeros(shape)

    def forward(self, X):
        # If X is not on the CPU, copy moving_mean and moving_var to the
        # device where X is located
        if self.moving_mean.context != X.context:
            self.moving_mean = self.moving_mean.copyto(X.context)
            self.moving_var = self.moving_var.copyto(X.context)
        # Save the updated moving_mean and moving_var
        Y, self.moving_mean, self.moving_var = batch_norm(
            X, self.gamma.data(), self.beta.data(), self.moving_mean,
            self.moving_var, eps=1e-5, momentum=0.9)
        return Y
```

7.5.4 Using a Batch Normalization LeNet

To see how to apply `BatchNorm` in context, below we apply it to a traditional LeNet model (Section 6.6). Recall that BN is typically applied after the convolutional layers and fully-connected layers but before the corresponding activation functions.

```
net = nn.Sequential()
net.add(nn.Conv2D(6, kernel_size=5),
        BatchNorm(6, num_dims=4),
        nn.Activation('sigmoid'),
        nn.MaxPool2D(pool_size=2, strides=2),
        nn.Conv2D(16, kernel_size=5),
```

(continues on next page)

(continued from previous page)

```
        BatchNorm(16, num_dims=4),
        nn.Activation('sigmoid'),
        nn.MaxPool2D(pool_size=2, strides=2),
        nn.Dense(120),
        BatchNorm(120, num_dims=2),
        nn.Activation('sigmoid'),
        nn.Dense(84),
        BatchNorm(84, num_dims=2),
        nn.Activation('sigmoid'),
        nn.Dense(10))
```

As before, we will train our network on the Fashion-MNIST dataset. This code is virtually identical to that when we first trained LeNet (Section 6.6). The main difference is the considerably larger learning rate.

```
lr, num_epochs, batch_size = 1.0, 10, 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
d2l.train_ch5(net, train_iter, test_iter, num_epochs, lr)
```

Let's have a look at the scale parameter `gamma` and the shift parameter `beta` learned from the first batch normalization layer.

```
net[1].gamma.data().reshape(-1,), net[1].beta.data().reshape(-1,)
```

7.5.5 Concise Implementation

Compared with the `BatchNorm` class, which we just defined ourselves, the `BatchNorm` class defined by the `nn` model in Gluon is easier to use. In Gluon, we do not have to worry about `num_features` or `num_dims`. Instead, these parameter values will be inferred automatically via delayed initialization. Otherwise, the code looks virtually identical to the application our implementation above.

```
net = nn.Sequential()
net.add(nn.Conv2D(6, kernel_size=5),
        nn.BatchNorm(),
        nn.Activation('sigmoid'),
        nn.MaxPool2D(pool_size=2, strides=2),
        nn.Conv2D(16, kernel_size=5),
        nn.BatchNorm(),
        nn.Activation('sigmoid'),
        nn.MaxPool2D(pool_size=2, strides=2),
        nn.Dense(120),
        nn.BatchNorm(),
        nn.Activation('sigmoid'),
        nn.Dense(84),
        nn.BatchNorm(),
        nn.Activation('sigmoid'),
        nn.Dense(10))
```

Below, we use the same hyper-parameters to train out model. Note that as usual, the Gluon variant runs much faster because its code has been compiled to C++/CUDA while our custom implementation must be interpreted by Python.

```
d2l.train_ch5(net, train_iter, test_iter, num_epochs, lr)
```

7.5.6 Controversy

Intuitively, batch normalization is thought to make the optimization landscape smoother. However, we must be careful to distinguish between speculative intuitions and true explanations for the phenomena that we observe when training deep models. Recall that we do not even know why simpler deep neural networks (MLPs and conventional CNNs) generalize well in the first place. Even with dropout and L2 regularization, they remain so flexible that their ability to generalize to unseen data cannot be explained via conventional learning-theoretic generalization guarantees.

In the original paper proposing batch normalization, the authors, in addition to introducing a powerful and useful tool, offered an explanation for why it works: by reducing *internal covariate shift*. Presumably by *internal covariate shift* the authors meant something like the intuition expressed above—the notion that the distribution of activations changes over the course of training. However there were two problems with this explanation: (1) This drift is very different from *covariate shift*, rendering the name a misnomer. (2) The explanation offers an under-specified intuition but leaves the question of *why precisely this technique works* an open question wanting for a rigorous explanation. Throughout this book, we aim to convey the intuitions that practitioners use to guide their development of deep neural networks. However, we believe that it is important to separate these guiding intuitions from established scientific fact. Eventually, when you master this material and start writing your own research papers you will want to be clear to delineate between technical claims and hunches.

Following the success of batch normalization, its explanation in terms of *internal covariate shift* has repeatedly surfaced in debates in the technical literature and broader discourse about how to present machine learning research. In a memorable speech given while accepting a Test of Time Award at the 2017 NeurIPS conference, Ali Rahimi used *internal covariate shift* as a focal point in an argument likening the modern practice of deep learning to alchemy. Subsequently, the example was revisited in detail in a position paper outlining troubling trends in machine learning (Lipton & Steinhardt, 2018).
In the technical literature other authors ((Santurkar et al., 2018)) have proposed alternative explanations for the success of BN, some claiming that BN's success comes despite exhibiting behavior that is in some ways opposite to those claimed in the original paper.

We note that the *internal covariate shift* is no more worthy of criticism than any of thousands of similarly vague claims made every year in the technical ML literature. Likely, its resonance as a focal point of these debates owes to its broad recognizability to the target audience. Batch normalization has proven an indispensable method, applied in nearly all deployed image classifiers, earning the paper that introduced the technique tens of thousands of citations.

Summary

- During model training, batch normalization continuously adjusts the intermediate output of the neural network by utilizing the mean and standard deviation of the minibatch, so that the values of the intermediate output in each layer throughout the neural network are more stable.
- The batch normalization methods for fully connected layers and convolutional layers are slightly different.
- Like a dropout layer, batch normalization layers have different computation results in training mode and prediction mode.
- Batch Normalization has many beneficial side effects, primarily that of regularization. On the other hand, the original motivation of reducing covariate shift seems not to be a valid explanation.

Exercises

1. Can we remove the fully connected affine transformation before the batch normalization or the bias parameter in convolution computation?
    - Find an equivalent transformation that applies prior to the fully connected layer.
    - Is this reformulation effective. Why (not)?
2. Compare the learning rates for LeNet with and without batch normalization.
    - Plot the decrease in training and test error.
    - What about the region of convergence? How large can you make the learning rate?
3. Do we need Batch Normalization in every layer? Experiment with it?
4. Can you replace Dropout by Batch Normalization? How does the behavior change?
5. Fix the coefficients `beta` and `gamma` (add the parameter `grad_req='null'` at the time of construction to avoid calculating the gradient), and observe and analyze the results.
6. Review the Gluon documentation for `BatchNorm` to see the other applications for Batch Normalization.
7. Research ideas: think of other normalization transforms that you can apply? Can you apply the probability integral transform? How about a full rank covariance estimate?

7.6 Residual Networks (ResNet)

As we design increasingly deeper networks it becomes imperative to understand how adding layers can increase the complexity and expressiveness of the network. Even more important is the ability to design networks where adding layers makes networks strictly more expressive rather than just different. To make some progress we need a bit of theory.

7.6.1 Function Classes

Consider $\mathcal{F}$, the class of functions that a specific network architecture (together with learning rates and other hyperparameter settings) can reach. That is, for all $f \in \mathcal{F}$ there exists some set of parameters W that can be obtained through training on a suitable dataset. Let's assume that f^* is the function that we really would like to find. If it is in $\mathcal{F}$, we are in good shape but typically we will not be quite so lucky. Instead, we will try to find some $f^*_{\mathcal{F}}$ which is our best bet within $\mathcal{F}$. For instance, we might try finding it by solving the following optimization problem:

$$f^*_{\mathcal{F}} := \operatorname*{argmin}_f L(X, Y, f) \text{ subject to } f \in \mathcal{F}. \tag{7.6.1}$$

It is only reasonable to assume that if we design a different and more powerful architecture $\mathcal{F}'$ we should arrive at a better outcome. In other words, we would expect that $f^*_{\mathcal{F}'}$ is "better" than $f^*_{\mathcal{F}}$. However, if $\mathcal{F} \not\subseteq \mathcal{F}'$ there is no guarantee that this should even happen. In fact, $f^*_{\mathcal{F}'}$ might well be worse. This is a situation that we often encounter in practice—adding layers does not only make the network more expressive, it also changes it in sometimes not quite so predictable ways. Fig. 7.6.1illustrates this in slightly abstract terms.

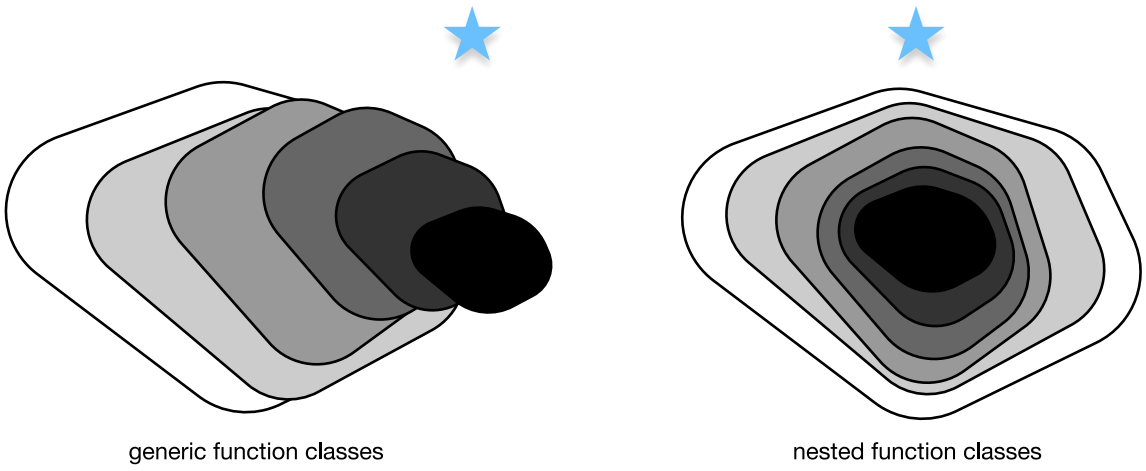


Fig. 7.6.1: Left: non-nested function classes. The distance may in fact increase as the complexity increases. Right: with nested function classes this does not happen.

Only if larger function classes contain the smaller ones are we guaranteed that increasing them strictly increases the expressive power of the network. This is the question that He et al, 2016 considered when working on very deep computer vision models. At the heart of ResNet is the idea that every additional layer should contain the identity function as one of its elements. This means that if we can train the newly-added layer into an identity mapping $f(\mathbf{x}) = \mathbf{x}$, the new model will be as effective as the original model. As the new model may get a better solution to fit the training dataset, the added layer might make it easier to reduce training errors. Even better,

the identity function rather than the null $f(\mathbf{x}) = 0$ should be the the simplest function within a layer.

These considerations are rather profound but they led to a surprisingly simple solution, a residual block. With it, (He et al., 2016a) won the ImageNet Visual Recognition Challenge in 2015. The design had a profound influence on how to build deep neural networks.

7.6.2 Residual Blocks

Let's focus on a local neural network, as depicted below. Denote the input by $\mathbf{x}$. We assume that the ideal mapping we want to obtain by learning is $f(\mathbf{x})$, to be used as the input to the activation function. The portion within the dotted-line box in the left image must directly fit the mapping $f(\mathbf{x})$. This can be tricky if we do not need that particular layer and we would much rather retain the input $\mathbf{x}$. The portion within the dotted-line box in the right image now only needs to parametrize the *deviation* from the identity, since we return $\mathbf{x} + f(\mathbf{x})$. In practice, the residual mapping is often easier to optimize. We only need to set $f(\mathbf{x}) = 0$. The right image in Fig. 7.6.2 illustrates the basic Residual Block of ResNet. Similar architectures were later proposed for sequence models which we will study later.

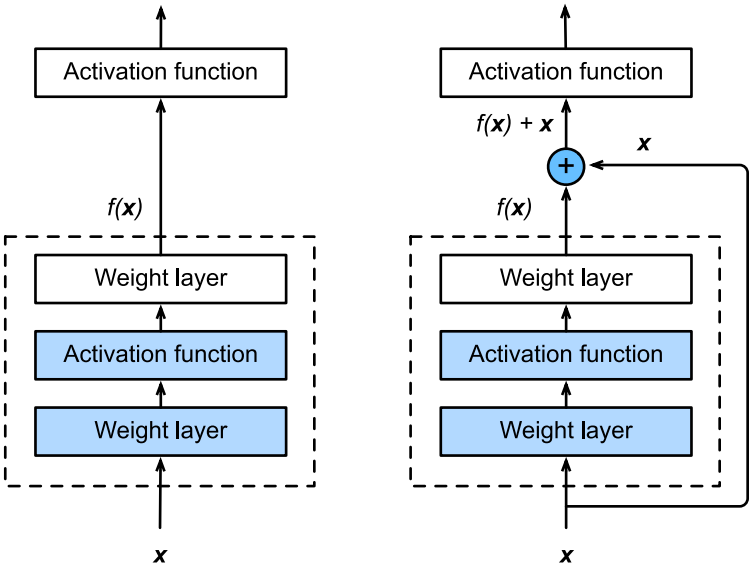


Fig. 7.6.2: The difference between a regular block (left) and a residual block (right). In the latter case, we can short-circuit the convolutions.

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. Then, we skip these two convolution operations and add the input directly before the final ReLU activation function. This kind of design requires that the output of the two convolutional layers be of the same shape as the input, so that they can be added together. If we want to change the number of channels or the the stride, we need to introduce an additional 1×1 convolutional layer to transform the input into the desired shape for the addition operation. Let's have a look at the code below.

```
import d2l
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()

# Saved in the d2l package for later use
class Residual(nn.Block):
    def __init__(self, num_channels, use_1x1conv=False, strides=1, **kwargs):
        super(Residual, self).__init__(**kwargs)
        self.conv1 = nn.Conv2D(num_channels, kernel_size=3, padding=1,
                               strides=strides)
        self.conv2 = nn.Conv2D(num_channels, kernel_size=3, padding=1)
        if use_1x1conv:
            self.conv3 = nn.Conv2D(num_channels, kernel_size=1,
                                   strides=strides)
        else:
            self.conv3 = None
        self.bn1 = nn.BatchNorm()
        self.bn2 = nn.BatchNorm()

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

This code generates two types of networks: one where we add the input to the output before applying the ReLU nonlinearity, and whenever `use_1x1conv=True`, one where we adjust channels and resolution by means of a 1×1 convolution before adding. Fig. 7.6.3 illustrates this:

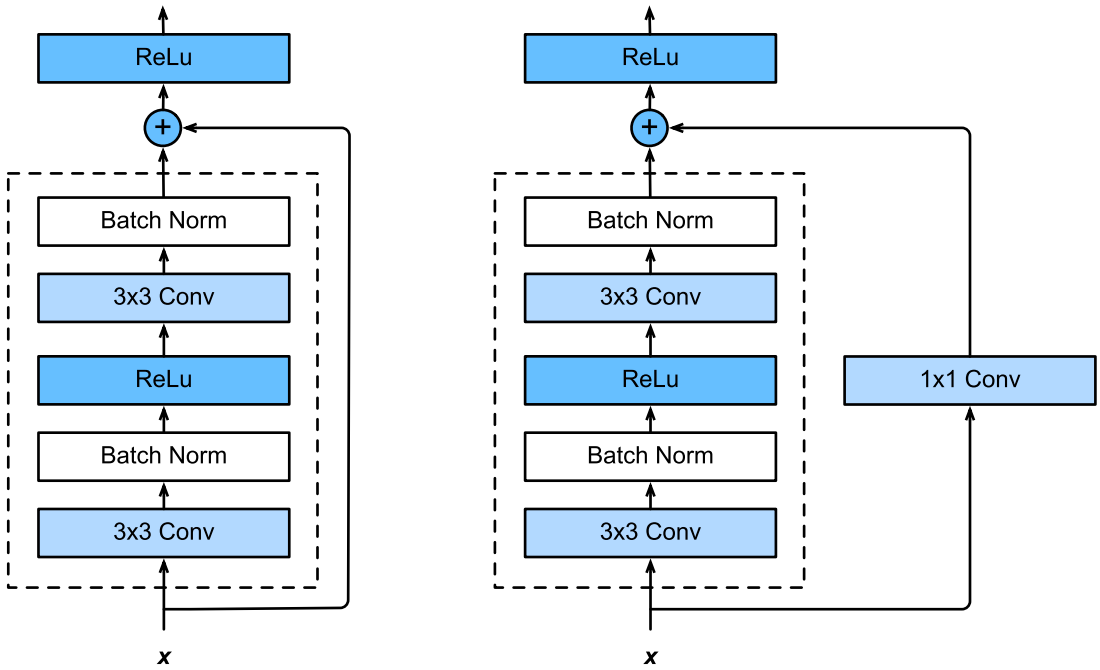


Fig. 7.6.3: Left: regular ResNet block; Right: ResNet block with 1x1 convolution

Now let's look at a situation where the input and output are of the same shape.

```
blk = Residual(3)
blk.initialize()
X = np.random.uniform(size=(4, 3, 6, 6))
blk(X).shape
```

```
(4, 3, 6, 6)
```

We also have the option to halve the output height and width while increasing the number of output channels.

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

```
(4, 6, 3, 3)
```

7.6.3 ResNet Model

The first two layers of ResNet are the same as those of the GoogLeNet we described before: the 7×7 convolutional layer with 64 output channels and a stride of 2 is followed by the 3×3 maximum pooling layer with a stride of 2. The difference is the batch normalization layer added after each convolutional layer in ResNet.

```
net = nn.Sequential()
net.add(nn.Conv2D(64, kernel_size=7, strides=2, padding=3),
        nn.BatchNorm(), nn.Activation('relu'),
        nn.MaxPool2D(pool_size=3, strides=2, padding=1))
```

GoogLeNet uses four blocks made up of Inception blocks. However, ResNet uses four modules made up of residual blocks, each of which uses several residual blocks with the same number of output channels. The number of channels in the first module is the same as the number of input channels. Since a maximum pooling layer with a stride of 2 has already been used, it is not necessary to reduce the height and width. In the first residual block for each of the subsequent modules, the number of channels is doubled compared with that of the previous module, and the height and width are halved.

Now, we implement this module. Note that special processing has been performed on the first module.

```
def resnet_block(num_channels, num_residuals, first_block=False):
    blk = nn.Sequential()
    for i in range(num_residuals):
        if i == 0 and not first_block:
            blk.add(Residual(num_channels, use_1x1conv=True, strides=2))
        else:
            blk.add(Residual(num_channels))
    return blk
```

Then, we add all the residual blocks to ResNet. Here, two residual blocks are used for each module.

```
net.add(resnet_block(64, 2, first_block=True),
        resnet_block(128, 2),
        resnet_block(256, 2),
        resnet_block(512, 2))
```

Finally, just like GoogLeNet, we add a global average pooling layer, followed by the fully connected layer output.

```
net.add(nn.GlobalAvgPool2D(), nn.Dense(10))
```

There are 4 convolutional layers in each module (excluding the 1×1 convolutional layer). Together with the first convolutional layer and the final fully connected layer, there are 18 layers in total. Therefore, this model is commonly known as ResNet-18. By configuring different numbers of channels and residual blocks in the module, we can create different ResNet models, such as the deeper 152-layer ResNet-152. Although the main architecture of ResNet is similar to that of GoogLeNet, ResNet's structure is simpler and easier to modify. All these factors have resulted in the rapid and widespread use of ResNet. Fig. 7.6.4 is a diagram of the full ResNet-18.

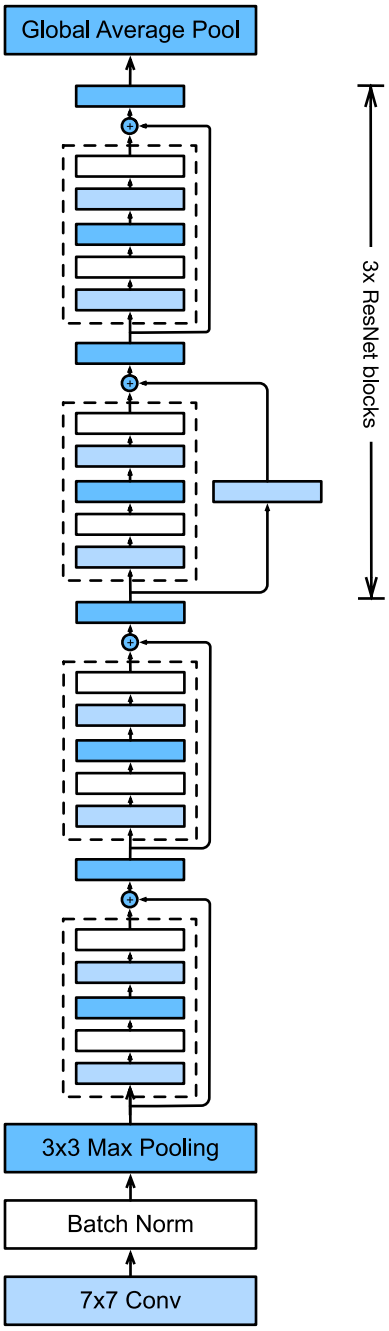


Fig. 7.6.4: ResNet 18

Before training ResNet, let's observe how the input shape changes between different modules in ResNet. As in all previous architectures, the resolution decreases while the number of channels increases up until the point where a global average pooling layer aggregates all features.

```
X = np.random.uniform(size=(1, 1, 224, 224))
net.initialize()
for layer in net:
    X = layer(X)
    print(layer.name, 'output shape:\t', X.shape)
```

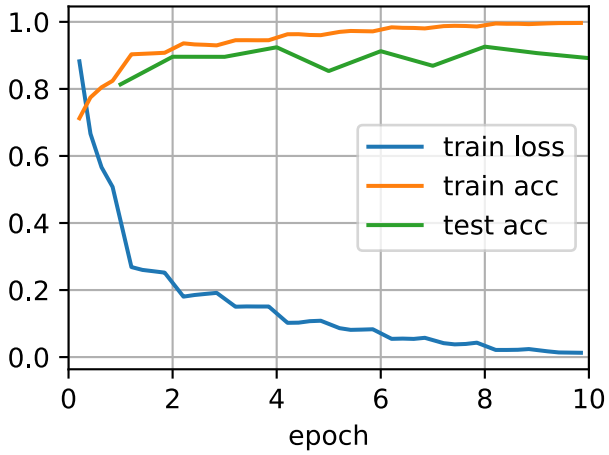
```
conv5 output shape:  (1, 64, 112, 112)
batchnorm4 output shape:     (1, 64, 112, 112)
relu0 output shape:  (1, 64, 112, 112)
pool0 output shape:  (1, 64, 56, 56)
sequential1 output shape:    (1, 64, 56, 56)
sequential2 output shape:    (1, 128, 28, 28)
sequential3 output shape:    (1, 256, 14, 14)
sequential4 output shape:    (1, 512, 7, 7)
pool1 output shape:  (1, 512, 1, 1)
dense0 output shape:         (1, 10)
```

7.6.4 Data Acquisition and Training

We train ResNet on the Fashion-MNIST dataset, just like before. The only thing that has changed is the learning rate that decreased again, due to the more complex architecture.

```
lr, num_epochs, batch_size = 0.05, 10, 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=96)
d2l.train_ch5(net, train_iter, test_iter, num_epochs, lr)
```

```
loss 0.013, train acc 0.997, test acc 0.892
4931.7 exampes/sec on gpu(0)
```



Summary

- Residual blocks allow for a parametrization relative to the identity function $f(\mathbf{x}) = \mathbf{x}$.
- Adding residual blocks increases the function complexity in a well-defined manner.
- We can train an effective deep neural network by having residual blocks pass through cross-layer data channels.
- ResNet had a major influence on the design of subsequent deep neural networks, both for convolutional and sequential nature.

Exercises

1. Refer to Table 1 in the (He et al., 2016a) to implement different variants.
2. For deeper networks, ResNet introduces a "bottleneck" architecture to reduce model complexity. Try to implement it.
3. In subsequent versions of ResNet, the author changed the "convolution, batch normalization, and activation" architecture to the "batch normalization, activation, and convolution" architecture. Make this improvement yourself. See Figure 1 in (He et al., 2016b) for details.
4. Prove that if $\mathbf{x}$ is generated by a ReLU, the ResNet block does indeed include the identity function.
5. Why cannot we just increase the complexity of functions without bound, even if the function classes are nested?

7.7 Densely Connected Networks (DenseNet)

ResNet significantly changed the view of how to parametrize the functions in deep networks. DenseNet is to some extent the logical extension of this. To understand how to arrive at it, let's take a small detour to theory. Recall the Taylor expansion for functions. For scalars it can be written as

$$f(x) = f(0) + f'(x)x + \frac{1}{2}f''(x)x^2 + \frac{1}{6}f'''(x)x^3 + o(x^3). \tag{7.7.1}$$

7.7.1 Function Decomposition

The key point is that it decomposes the function into increasingly higher order terms. In a similar vein, ResNet decomposes functions into

$$f(\mathbf{x}) = \mathbf{x} + g(\mathbf{x}). \tag{7.7.2}$$

That is, ResNet decomposes f into a simple linear term and a more complex nonlinear one. What if we want to go beyond two terms? A solution was proposed by (Huang et al., 2017) in the form of DenseNet, an architecture that reported record performance on the ImageNet dataset.

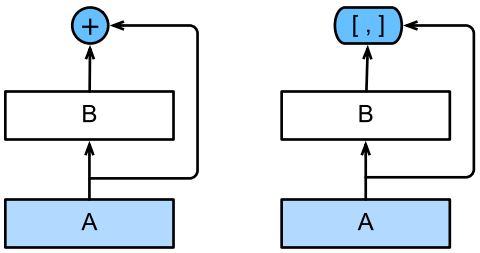


Fig. 7.7.1: The main difference between ResNet (left) and DenseNet (right) in cross-layer connections: use of addition and use of concatenation.

As shown in Fig. 7.7.1, the key difference between ResNet and DenseNet is that in the latter case outputs are *concatenated* rather than added. As a result we perform a mapping from $\mathbf{x}$ to its values after applying an increasingly complex sequence of functions.

$$\mathbf{x} \to [\mathbf{x}, f_1(\mathbf{x}), f_2(\mathbf{x}, f_1(\mathbf{x})), f_3(\mathbf{x}, f_1(\mathbf{x}), f_2(\mathbf{x}, f_1(\mathbf{x})), \ldots] . \tag{7.7.3}$$

In the end, all these functions are combined in an MLP to reduce the number of features again. In terms of implementation this is quite simple—rather than adding terms, we concatenate them. 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. The former defines how the inputs and outputs are concatenated, while the latter controls the number of channels so that it is not too large. The dense connections are shown in Fig. 7.7.2.

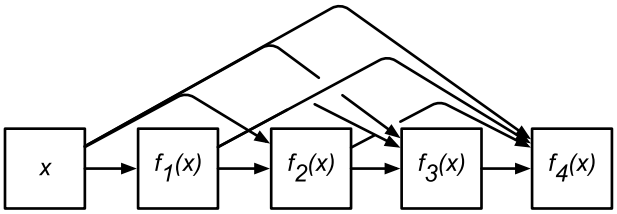


Fig. 7.7.2: Dense connections in DenseNet

7.7.2 Dense Blocks

DenseNet uses the modified "batch normalization, activation, and convolution" architecture of ResNet (see the exercise in Section 7.6). First, we implement this architecture in the `conv_block` function.

```
import d2l
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()

def conv_block(num_channels):
    blk = nn.Sequential()
    blk.add(nn.BatchNorm(),
            nn.Activation('relu'),
            nn.Conv2D(num_channels, kernel_size=3, padding=1))
    return blk
```

A dense block consists of multiple `conv_block` units, each using the same number of output channels. In the forward computation, however, we concatenate the input and output of each block on the channel dimension.

```
class DenseBlock(nn.Block):
    def __init__(self, num_convs, num_channels, **kwargs):
        super(DenseBlock, self).__init__(**kwargs)
        self.net = nn.Sequential()
        for _ in range(num_convs):
            self.net.add(conv_block(num_channels))
```

(continues on next page)

(continued from previous page)

```
    def forward(self, X):
        for blk in self.net:
            Y = blk(X)
            # Concatenate the input and output of each block on the channel
            # dimension
            X = np.concatenate((X, Y), axis=1)
        return X
```

In the following example, we define a convolution block with two blocks of 10 output channels. When using an input with 3 channels, we will get an output with the $3+2\times 10=23$ channels. The number of convolution block channels controls the increase in the number of output channels relative to the number of input channels. This is also referred to as the growth rate.

```
blk = DenseBlock(2, 10)
blk.initialize()
X = np.random.uniform(size=(4, 3, 8, 8))
Y = blk(X)
Y.shape
```

```
(4, 23, 8, 8)
```

7.7.3 Transition Layers

Since each dense block will increase the number of channels, adding too many of them will lead to an excessively complex model. A transition layer is used to control the complexity of the model. It reduces the number of channels by using the 1×1 convolutional layer and halves the height and width of the average pooling layer with a stride of 2, further reducing the complexity of the model.

```
def transition_block(num_channels):
    blk = nn.Sequential()
    blk.add(nn.BatchNorm(), nn.Activation('relu'),
            nn.Conv2D(num_channels, kernel_size=1),
            nn.AvgPool2D(pool_size=2, strides=2))
    return blk
```

Apply a transition layer with 10 channels to the output of the dense block in the previous example. This reduces the number of output channels to 10, and halves the height and width.

```
blk = transition_block(10)
blk.initialize()
blk(Y).shape
```

```
(4, 10, 4, 4)
```

7.7.4 DenseNet Model

Next, we will construct a DenseNet model. DenseNet first uses the same single convolutional layer and maximum pooling layer as ResNet.

```
net = nn.Sequential()
net.add(nn.Conv2D(64, kernel_size=7, strides=2, padding=3),
        nn.BatchNorm(), nn.Activation('relu'),
        nn.MaxPool2D(pool_size=3, strides=2, padding=1))
```

Then, similar to the four residual blocks that ResNet uses, DenseNet uses four dense blocks. Similar to ResNet, we can set the number of convolutional layers used in each dense block. Here, we set it to 4, consistent with the ResNet-18 in the previous section. Furthermore, we set the number of channels (i.e., growth rate) for the convolutional layers in the dense block to 32, so 128 channels will be added to each dense block.

In ResNet, the height and width are reduced between each module by a residual block with a stride of 2. Here, we use the transition layer to halve the height and width and halve the number of channels.

```
# Num_channels: the current number of channels
num_channels, growth_rate = 64, 32
num_convs_in_dense_blocks = [4, 4, 4, 4]

for i, num_convs in enumerate(num_convs_in_dense_blocks):
    net.add(DenseBlock(num_convs, growth_rate))
    # This is the number of output channels in the previous dense block
    num_channels += num_convs * growth_rate
    # A transition layer that haves the number of channels is added between
    # the dense blocks
    if i != len(num_convs_in_dense_blocks) - 1:
        num_channels //= 2
        net.add(transition_block(num_channels))
```

Similar to ResNet, a global pooling layer and fully connected layer are connected at the end to produce the output.

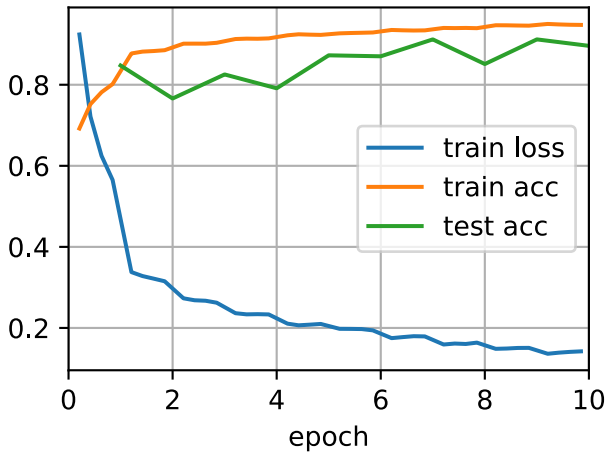
```
net.add(nn.BatchNorm(),
        nn.Activation('relu'),
        nn.GlobalAvgPool2D(),
        nn.Dense(10))
```

7.7.5 Data Acquisition and Training

Since we are using a deeper network here, in this section, we will reduce the input height and width from 224 to 96 to simplify the computation.

```
lr, num_epochs, batch_size = 0.1, 10, 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=96)
d2l.train_ch5(net, train_iter, test_iter, num_epochs, lr)
```

```
loss 0.144, train acc 0.947, test acc 0.896
5628.8 exampes/sec on gpu(0)
```



Summary

- In terms of cross-layer connections, unlike ResNet, where inputs and outputs are added together, DenseNet concatenates inputs and outputs on the channel dimension.
- The main units that compose DenseNet are dense blocks and transition layers.
- We need to keep the dimensionality under control when composing the network by adding transition layers that shrink the number of channels again.

Exercises

1. Why do we use average pooling rather than max-pooling in the transition layer?
2. One of the advantages mentioned in the DenseNet paper is that its model parameters are smaller than those of ResNet. Why is this the case?
3. One problem for which DenseNet has been criticized is its high memory consumption.
   - Is this really the case? Try to change the input shape to 224×224 to see the actual (GPU) memory consumption.
   - Can you think of an alternative means of reducing the memory consumption? How would you need to change the framework?
4. Implement the various DenseNet versions presented in Table 1 of (Huang et al., 2017).
5. Why do we not need to concatenate terms if we are just interested in $\mathbf{x}$ and $f(\mathbf{x})$ for ResNet? Why do we need this for more than two layers in DenseNet?
6. Design a DenseNet for fully connected networks and apply it to the Housing Price prediction task.