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

Show hidden output

COSE474-2024F: Deep Learning

7.6. Convolutional Neural Networks (LeNet)

We now have all the ingredients required to assemble a fully-functional CNN. In our earlier encounter with image data, we applied a linear model with softmax regression (Section 4.4) and an MLP (Section 5.2) to pictures of clothing in the Fashion-MNIST dataset. To make such data amenable we first flattened each image from a 28×28 matrix into a fixed-length 784-dimensional vector, and thereafter processed them in fully connected layers. Now that we have a handle on convolutional layers, we can retain the spatial structure in our images. As an additional benefit of replacing fully connected layers with convolutional layers, we will enjoy more parsimonious models that require far fewer parameters.

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

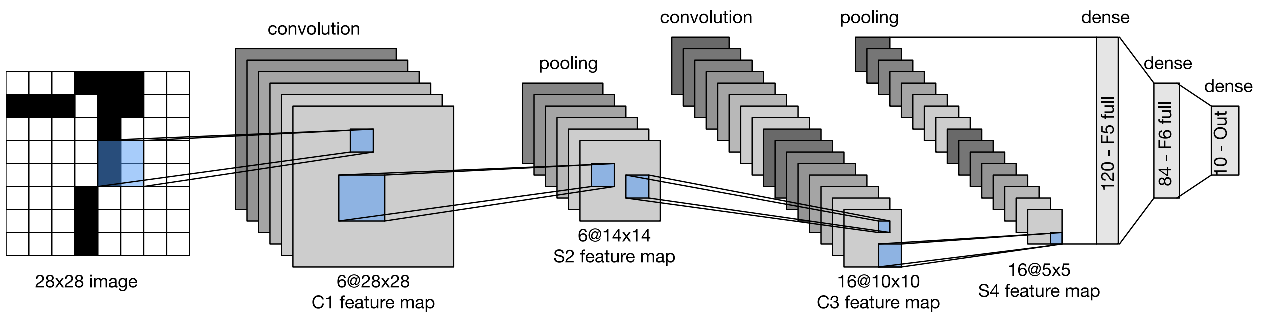
At the time LeNet achieved outstanding results matching the performance of support vector machines, then a dominant approach in supervised learning, achieving an error rate of less than 1% per digit. LeNet was eventually adapted to recognize digits for processing deposits in ATM machines. To this day, some ATMs still run the code that Yann LeCun and his colleague Leon Bottou wrote in the 1990s!

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

7.6.1. LeNet

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

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



The basic units in each convolutional block are a convolutional layer, a sigmoid activation function, and a subsequent average pooling operation. Note that while ReLUs and max-pooling work better, they had not yet been discovered. Each convolutional layer uses a 5×5 kernel and a sigmoid activation function. These layers map spatially arranged inputs to a number of two-dimensional feature maps, typically increasing the number of channels. The first convolutional layer has 6 output channels, while the second has 16. Each 2×2 pooling operation (stride 2) reduces dimensionality by a factor of 4 via spatial downsampling. The convolutional block emits an output with shape given by (batch size, number of channel, height, width).

In order to pass output from the convolutional block to the dense block, we must flatten each example in the minibatch. In other words, we take this four-dimensional input and transform it into the two-dimensional input expected by fully connected layers: as a reminder, the two-dimensional representation that we desire uses the first dimension to index examples in the minibatch and the second to give the flat vector representation of each example. LeNet's dense block has three fully connected layers, with 120, 84, and 10 outputs, respectively. Because we are still performing classification, the 10-dimensional output layer corresponds to the number of possible output classes.

While getting to the point where you truly understand what is going on inside LeNet may have taken a bit of work, we hope that the following code snippet will convince you that implementing such models with modern deep learning frameworks is remarkably simple. We need only to instantiate a `Sequential` block and chain together the appropriate layers, using Xavier initialization as introduced in Section 5.4.2.2.

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

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

"@save" is not an allowed annotation - allowed values include [@param, @title, @markdown].

"@save" is not an allowed annotation - allowed values include [@param, @title, @markdown].

We have taken some liberty in the reproduction of LeNet insofar as we have replaced the Gaussian activation layer by a softmax layer. This greatly simplifies the implementation, not least due to the fact that the Gaussian decoder is rarely used nowadays. Other than that, this network matches the original LeNet-5 architecture.

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

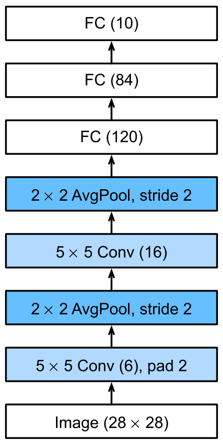


Fig. 7.6.2 Compressed notation for LeNet-5.

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

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

"@save" is not an allowed annotation - allowed values include [@param, @title, @markdown].

```
Conv2d output shape:     torch.Size([1, 6, 28, 28])
Sigmoid output shape:    torch.Size([1, 6, 28, 28])
AvgPool2d output shape:  torch.Size([1, 6, 14, 14])
```

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

Note that the height and width of the representation at each layer throughout the convolutional block is reduced (compared with the previous layer). The first convolutional layer uses two pixels of padding to compensate for the reduction in height and width that would otherwise result from using a 5×5 kernel. As an aside, the image size of 28×28 pixels in the original MNIST OCR dataset is a result of *trimming* two pixel rows (and columns) from the original scans that measured 32×32 pixels. This was done primarily to save space (a 30% reduction) at a time when megabytes mattered.

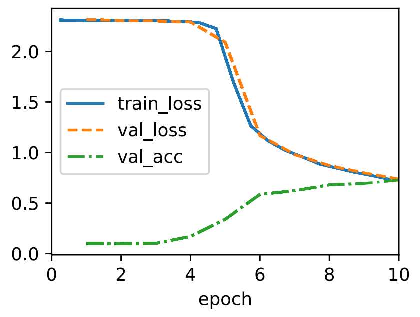
In contrast, the second convolutional layer forgoes padding, and thus the height and width are both reduced by four pixels. As we go up the stack of layers, the number of channels increases layer-over-layer from 1 in the input to 6 after the first convolutional layer and 16 after the second convolutional layer. However, each pooling layer halves the height and width. Finally, each fully connected layer reduces dimensionality, finally emitting an output whose dimension matches the number of classes.

7.6.2. Training

Now that we have implemented the model, let's run an experiment to see how the LeNet-5 model fares on Fashion-MNIST.

While CNNs have fewer parameters, they can still be more expensive to compute than similarly deep MLPs because each parameter participates in many more multiplications. If you have access to a GPU, this might be a good time to put it into action to speed up training. Note that the d2l.Trainer class takes care of all details. By default, it initializes the model parameters on the available devices. Just as with MLPs, our loss function is cross-entropy, and we minimize it via minibatch stochastic gradient descent.

```
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128)
model = LeNet(lr=0.1)
model.apply_init([next(iter(data.get_dataloader(True)))[0]], init_cnn)
trainer.fit(model, data)
```



7.6.3. Discussion

- It is worth comparing the error rates on Fashion-MNIST achievable with LeNet-5 both to the very best possible with MLPs (Section 5.2) and those with significantly more advanced architectures such as ResNet (Section 8.6).
- LeNet is much more similar to the latter than to the former.
- Greater amounts of computation enabled significantly more complex architectures.
- A second difference is the relative ease with which we were able to implement LeNet.