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

Show hidden output

7.1. From Fully Connected Layers to Convolutions

Tabular Data - MLP

High-Dimensional Data - CNN

7.1.1 Invariance

Invariance - system's ability to recognize an object or patter regardless of changes in its position.

An effective model for object detection should recognize objects regardless of its position. This is similar to 'Waldo', where his appearance does not depend on his location. And CNNs use this idea of **spatial invariance**.

Key principles for designing NN:

- Translation invariance: The NN should recognize an object in different positions.
- Locality: The initial layers should focus on small, local areas of the image instead of focusing on the entire image.
- Deeper Layers Capture More: The deeper it is into the network, it should learn to recognize more complex patterns that may cover larger areas of the image.

7.1.2 Constraining the MLP

MLP: X as an two-dimensional input, H as an hidden representation after processing the input. Each pixel in the image - $[X]_{i,j}$ and $[H]_{i,j}$. Weight Matrices -> Fourth-Order Tensors to account for the spacial strucure

V by re-indexing the sums, focusing on local neighborhoods around each pixel:

$[H]_{i,j} = [U]_{i,j} + \sum_a \sum_b [V]_{i,j,a,b} [X]_{i+a,j+b}$

Locality Principle: The principle of locality suggests that to assess the value of a hidden representation $[H]_{i,j}$ at a specific pixel location (i, j), you should only need to consider nearby pixels in the input image $[X]$.

Threshold (Δ): We define a neighborhood size Δ which determines how far from (i, j) we look. For offsets a and b: If $|a|$ > Δ or $|b|$ > Δ, the weights are $[V]_{a,b}$ = 0. This means that if the pixels are too far away, they won't contribute to the calculation of $[H]_{i,j}$.

Hidden Representation Calculation with locality:

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

By constraining the weight to small area, the number of parameters are reduced. It leads to the *convolutional layer* or *filter*,denoted as $[V]$, that slides over the input image and performs the weighted sum operation.

7.1.3 Convolutions

Convolution is a way to combine two functions to create a third function that expresses how the shape of one is modified by the other.

Convolution for two-dimensional data(like images):

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

7.1.4 Channels

Images consist of three channels—red, green, and blue—so they are not just two-dimensional objects but third-order tensors (height, width, and channels). In CNNs, filters (kernels) are applied to these channels, requiring the convolutional filter to account for all three channels.

The output of the convolution is also a third-order tensor, with each location having multiple hidden representations (channels/feature maps), which stack together to form grids. These channels allow the network to specialize in detecting different features, such as edges or textures, across multiple layers.

Convolution at a specific location in a multi-channel convolutional layer:

$$H_{i,j,d} = \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} \sum_{c} V_{a,b,c,d} X_{i+a,j+b,c}$$

Discussions/Takeaways/Exercises

CNNs were introduces to address the problems of fully connected layers when processing high dimensional data like images. It uses translation invariance and locality principle to reduce number of parameters, and adds additional channels.

7.2. Convolutions for Images

```
import torch
from torch import nn
from d2l import torch as d2l
from torch.nn import functional as F
```

7.2.1 CNNs uses cross-correlation, not exactly a convolution. In a two-dimensional cross-correlation operation, the kernel slides over the image (input tensor) and computes elementwise multiplication between the kernel and the input region, summing the results to produce one value for the output tensor. This process is repeated as the kernel moves across the image, forming the entire output tensor.

```
def corr2d(X, K):
    """Compute 2D cross-correlation."""
    h, w = K.shape
    Y = torch.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            Y[i, j] = (X[i:i + h, j:j + w] * K).sum()
    return Y


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

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

7.2.2 Convolutional layer = input.kernel + scalar bias

Two parameters of a convolutional layer: kernel and scalar bias

- define the weights and the bias
- define the forward propagation method

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

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

7.2.3 Using convolutional layer to detect the edge of an object in an image by finding the location of the pixel change.

```
# construct an “image” of 6x8 pixels.
X = torch.ones((6, 8))
X[:, 2:6] = 0
X
```

```
tensor([[1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.]])
```

```
# edge detection kernel with height = 1 and width = 2
# kernel values are set to [1.0,-1.0], which means it computes the difference between horizontally adjacent pixels
# same -> 0
K = torch.tensor([[1.0, -1.0]])
```

```
# cross-correlation of the input and the kernel
Y = corr2d(X, K)
Y
```

```
tensor([[ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.]])
```

```
# apply the kernel to the transposed image
# detects only vertical edges
corr2d(X.t(), K)
```

```
tensor([[0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
```

```
[0., 0., 0., 0., 0.],
[0., 0., 0., 0., 0.],
[0., 0., 0., 0., 0.]])
```

7.2.4 Learning a Kernel

Developing a non-manual edge detectore unlike the previous using learned parameters.

Use the input-output pairs to train a convolutional layer that will learn the optimal kernel for edge detection through backpropagation.

```
# 2d convolutional layer with one output channel and a kernel size of (1, 2).
conv2d = nn.LazyConv2d(1, kernel_size=(1, 2), bias=False)

# The two-dimensional convolutional layer uses four-dimensional input and
# output in the format of (example, channel, height, width), where the batch
# size (number of examples in the batch) and the number of channels are both 1
X = X.reshape((1, 1, 6, 8))
Y = Y.reshape((1, 1, 6, 7))
lr = 3e-2  # Learning rate

for i in range(10):
    Y_hat = conv2d(X) #predicted output
    l = (Y_hat - Y) ** 2  #loss
    conv2d.zero_grad()
    l.sum().backward()
    # update the kernel using gradient descent
    conv2d.weight.data[:] -= lr * conv2d.weight.grad
    if (i + 1) % 2 == 0:
        print(f'epoch {i + 1}, loss {l.sum():.3f}')
```

```
epoch 2, loss 8.486
epoch 4, loss 2.214
epoch 6, loss 0.695
epoch 8, loss 0.249
epoch 10, loss 0.096
```

```
conv2d.weight.data.reshape((1, 2))
```

```
tensor([[ 0.9568, -1.0194]])
```

7.2.5 Cross-Correlation and Convolution

Cross-correlation is the process of sliding a kernel (filter) over an input tensor and computing a weighted sum of the input values that fall under the kernel to extract the features from input data, such as images.

Convolution involves flipping the kernel both horizontally and vertically before performing the operation with the input tensor.

Both can yield the same output . f a convolutional layer performs cross-correlation and learns a kernel denoted as K, then if the same layer were to perform strict convolution, the kernel K ' would be the result of flipping K both horizontally and vertically. K ' = flip (K)

7.2.6 Feature Map and Receptive Field

Feature Maps - outputs of convolutional layers representing learned features in spatial dimensions.

Receptive Fields - sets of input elements influencing the calculation of an output element, larger than the input size.

Discussions/Takeaways

Convolutions play an important role in image processing, particularly in CNNs, which efficiently capture spatial hierarchies in images. The cross-correlation operation, where an input tensor (image) interacts with a kernel tensor (filter) to produce an output tensor, known as the feature map.

The kernel slides over the input image, performing element-wise multiplication followed by a sum to generate a single scalar value for each position. This process reveals features like edges or textures by detecting local patterns in pixel intensity changes. The output size is reduced compared to the input size due to the finite dimensions of the kernel.

7.3. Padding and Stride

If the input shape is $n \times n$ and the convolution kernel shape is $k \times k$ -> the output shape will be $(n-k+1) \times (n-k+1)$

Padding handles the issue of spatial dimension reduction that occurs as a result of applying convolutional layers.

```
def comp_conv2d(conv2d, X):
    X = X.reshape((1, 1) + X.shape)
    Y = conv2d(X)
    # Strip the first two dimensions: examples and channels
    return Y.reshape(Y.shape[2:])

conv2d = nn.LazyConv2d(1, kernel_size=3, padding=1)
X = torch.rand(size=(8, 8))
comp_conv2d(conv2d, X).shape
```

```
torch.Size([8, 8])
```

```
conv2d = nn.LazyConv2d(1, kernel_size=(5, 3), padding=(2, 1))
comp_conv2d(conv2d, X).shape
```

```
torch.Size([8, 8])
```

Stride refers to how far the convolution window moves during each step when sliding across the input tensor.

Adjusting the stride is particularly useful when downsampling an image is necessary, as it allows reducing the spatial dimensions of the data while preserving important features.

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

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

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

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

Discussions/Takeaways

Padding and stride are key techniques used in CNNs to control the output size and manage the resolution of the data.

Padding ensures information preservation, while stride controls the spatial reduction of the output, giving CNNs flexibility in handling different input sizes and resolutions.

7.4. Multiple Input and Multiple Output Channels

If there are multiple input channels, all we have to do is perform cross-correlation operation per channel and then add up the results.

```
def corr2d_multi_in(X, K):
    return sum(d2l.corr2d(x, k) for x, k in zip(X, K))


X = torch.tensor([[[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]],
                  [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]]])
K = torch.tensor([[[0.0, 1.0], [2.0, 3.0]], [[1.0, 2.0], [3.0, 4.0]]])

corr2d_multi_in(X, K)
```

```
tensor([[ 56.,  72.],
        [104., 120.]])
```

```
def corr2d_multi_in_out(X, K):
    return torch.stack([corr2d_multi_in(X, k) for k in K], 0)
```

```
K = torch.stack((K, K + 1, K + 2), 0)
K.shape
```

```
torch.Size([3, 2, 2, 2])
```

```
corr2d_multi_in_out(X, K)
```

```
tensor([[[ 56.,  72.],
         [104., 120.]],

        [[ 76., 100.],
         [148., 172.]],

        [[ 96., 128.],
         [192., 224.]]])
```

For 1x1 convolution, it loses the ability of larger convolutional layers to recognize patterns consisting of interactions among adjacent elements in the height and width dimensions. The only computation of the 1×1 convolution occurs on the channel dimension.

1x1 convolution:

- no spatial interaction
- operates on channels
- transforms channels

```
def corr2d_multi_in_out_1x1(X, K):
    c_i, h, w = X.shape
    c_o = K.shape[0]
    X = X.reshape((c_i, h * w))
    K = K.reshape((c_o, c_i))
    Y = torch.matmul(K, X)
    return Y.reshape((c_o, h, w))
```

```
X = torch.normal(0, 1, (3, 3, 3))
K = torch.normal(0, 1, (2, 3, 1, 1))
Y1 = corr2d_multi_in_out_1x1(X, K)
Y2 = corr2d_multi_in_out(X, K)
assert float(torch.abs(Y1 - Y2).sum()) < 1e-6
```

Discussions/Takeaways

What about the impact of the 1x1 convolutions? 1x1 convolutions are frequently used to reduce the number of channels, by doing that required computation and memory is reduced.

After reducing the number of channels, 1x1 convolutions can be used again to increase the number of channels.

This process is referred to as **bottleneck layers**, where the number of channels is reduced and then increased again.This shows how much impact can 1x1 convolutions can have in controlling the network's dimensionality.

7.5. Pooling

Pooling layers serve the dual purposes of mitigating the sensitivity of convolutional layer to location and of spatially downsampling representations.

Pooling Operation: Pooling uses a fixed-size window (pooling window) that moves across the input according to its stride, computing a single value (either maximum or average) from the values within the window. There are no learned parameters like in convolutional layers.

In max pooling, the operation selects the maximum value from the window at each step. It helps in preserving strong features in an image by selecting the most prominent pixel values. Average pooling takes the average value from the window at each step,smoothing data by reducing noise but may lose critical information.

```
def pool2d(X, pool_size, mode='max'):
    p_h, p_w = pool_size
    Y = torch.zeros((X.shape[0] - p_h + 1, X.shape[1] - p_w + 1))
```

```
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            if mode == 'max':
                Y[i, j] = X[i: i + p_h, j: j + p_w].max()
            elif mode == 'avg':
                Y[i, j] = X[i: i + p_h, j: j + p_w].mean()
    return Y


X = torch.tensor([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
pool2d(X, (2, 2))
```

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

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

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

Pooling layers, like convolutional layers, change the output shape of the input data by reducing its spatial dimensions. The reduction depends on the pooling window size, padding, and stride.

```
X = torch.arange(16, dtype=torch.float32).reshape((1, 1, 4, 4))
X
```

```
tensor([[[[ 0.,  1.,  2.,  3.],
          [ 4.,  5.,  6.,  7.],
          [ 8.,  9., 10., 11.],
          [12., 13., 14., 15.]]]])
```

```
pool2d = nn.MaxPool2d(3)
pool2d(X)
```

```
tensor([[[[10.]]]])
```

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

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

```
pool2d = nn.MaxPool2d((2, 3), stride=(2, 3), padding=(0, 1))
pool2d(X)
```

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

When processing multi-channel input data, the pooling layer pools each input channel separately, rather than summing the inputs up over channels as in a convolutional layer.

the number of output channels = number of input channels

```
X = torch.cat((X, X + 1), 1)
X
```

```
tensor([[[[ 0.,  1.,  2.,  3.],
          [ 4.,  5.,  6.,  7.],
          [ 8.,  9., 10., 11.],
          [12., 13., 14., 15.]],

         [[ 1.,  2.,  3.,  4.],
          [ 5.,  6.,  7.,  8.],
          [ 9., 10., 11., 12.],
          [13., 14., 15., 16.]]]])
```

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

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

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

Discussions/Takeaways

Since the pooling values are determined manually, there should also be non-deterministic stochastic pooling. This approach can enhance the traditional pooling method, by introducing a degree of randomness which could help prevent overfitting.

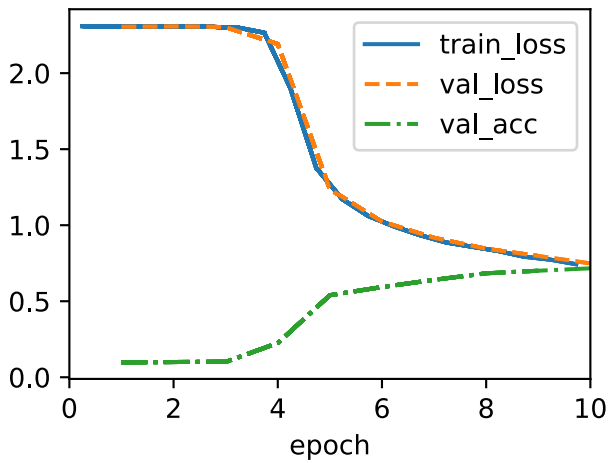
7.6. Convolutional Neural Networks (LeNet)

```
def init_cnn(module):
    if type(module) == nn.Linear or type(module) == nn.Conv2d:
        nn.init.xavier_uniform_(module.weight)

class LeNet(d2l.Classifier):
    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))


@d2l.add_to_class(d2l.Classifier)
def layer_summary(self, X_shape):
    X = torch.randn(*X_shape)
```

```
    for layer in self.net:
        X = layer(X)
        print(layer.__class__.__name__, 'output shape:\t', X.shape)

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

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

```
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)
```



Discussions/Takeaways

LeNet-5 consists of two main components:

- Convolutional Encoder: This section includes two convolutional layers that extract features from the input image. Each convolutional layer is followed by a sigmoid activation function and an average pooling layer, which reduces the dimensionality of the feature maps.
- Dense Block: This part comprises three fully connected layers that perform the final classification. The architecture allows for the input image's spatial structure to be preserved, making it more efficient than traditional fully connected networks.

Convolutional Layers: LeNet uses 5×5 kernels for convolution. The first layer outputs 6 channels, while the second outputs 16 channels. The pooling layers (2×2) downsample the feature maps, reducing their size and allowing the network to focus on the most salient features.

Activation Function: a sigmoid function.

Flattening: After passing through the convolutional and pooling layers, the output is flattened into a two-dimensional shape to be fed into the fully connected layers. This step transforms the multi-dimensional output into a format suitable for classification.

Why was the sigmoid function switched later:Models like LeNet relied on sigmoid activation functions. While they worked reasonably well at the time, they had some significant drawbacks. One major issue was the vanishing gradient problem. This happens when gradients (the values that help the model learn) become very small as they propagate back through the network. When this occurs, the model struggles to learn,

especially in deeper networks, making training slow or even impossible. Instead of sigmoid function, functions like ReLU and Leaky ReLU were proposed

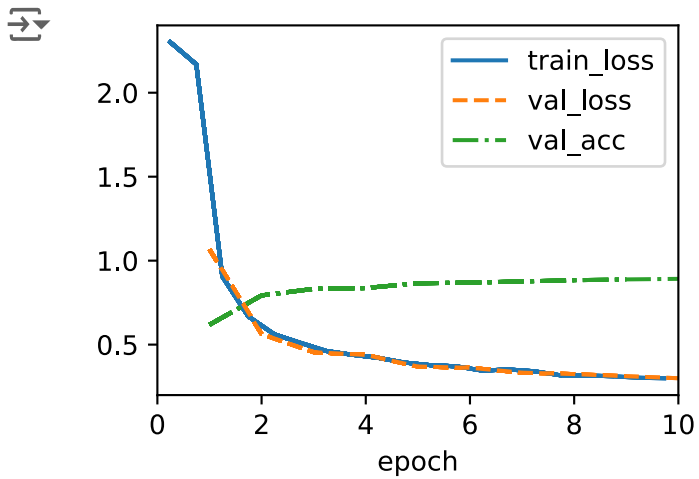
8.2. Networks Using Blocks (VGG)

```
def vgg_block(num_convs, out_channels):
    layers = []
    for _ in range(num_convs):
        layers.append(nn.LazyConv2d(out_channels, kernel_size=3, padding=1))
        layers.append(nn.ReLU())
    layers.append(nn.MaxPool2d(kernel_size=2, stride=2))
    return nn.Sequential(*layers)


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


VGG(arch=((1, 64), (1, 128), (2, 256), (2, 512), (2, 512))).layer_summary(
    (1, 1, 224, 224))
```

```
Sequential output shape:         torch.Size([1, 64, 112, 112])
Sequential output shape:         torch.Size([1, 128, 56, 56])
Sequential output shape:         torch.Size([1, 256, 28, 28])
```

```
Sequential output shape:         torch.Size([1, 512, 14, 14])
Sequential output shape:         torch.Size([1, 512, 7, 7])
Flatten output shape:    torch.Size([1, 25088])
Linear output shape:     torch.Size([1, 4096])
ReLU output shape:       torch.Size([1, 4096])
Dropout output shape:    torch.Size([1, 4096])
Linear output shape:     torch.Size([1, 4096])
ReLU output shape:       torch.Size([1, 4096])
Dropout output shape:    torch.Size([1, 4096])
Linear output shape:     torch.Size([1, 10])
```

```
model = VGG(arch=((1, 16), (1, 32), (2, 64), (2, 128), (2, 128)), lr=0.01)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(224, 224))
model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
trainer.fit(model, data)
```



Discussions/Takeaways

Early networks were effective but did not provide a structured designed template.

So, Visual Geometry Group (VGG),consistinf of multiple convolutional layer and a max-pooling layer, was introduced. Deep and narrow networks generally outperform shallow ones, leading to very deep networks.

VGG networks are composed of two parts:

- convolutional layers (made of blocks)
- fully connectted layers

Because VGG uses smaller convolutional filter, it brings several advantages. It allows the network work with finer and more detailed features. And also it allows for layered approach in the recognition, where ones output is passed to the next layer as an input.

8.6. Residual Networks (ResNet) and ResNeXt

```
class Residual(nn.Module):
    def __init__(self, num_channels, use_1x1conv=False, strides=1):
        super().__init__()
        self.conv1 = nn.LazyConv2d(num_channels, kernel_size=3, padding=1,
                                   stride=strides)
        self.conv2 = nn.LazyConv2d(num_channels, kernel_size=3, padding=1)
        if use_1x1conv:
            self.conv3 = nn.LazyConv2d(num_channels, kernel_size=1,
                                       stride=strides)
        else:
            self.conv3 = None
        self.bn1 = nn.LazyBatchNorm2d()
        self.bn2 = nn.LazyBatchNorm2d()

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

```
blk = Residual(3)
X = torch.randn(4, 3, 6, 6)
blk(X).shape
```

```
torch.Size([4, 3, 6, 6])
```

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

```
torch.Size([4, 6, 3, 3])
```

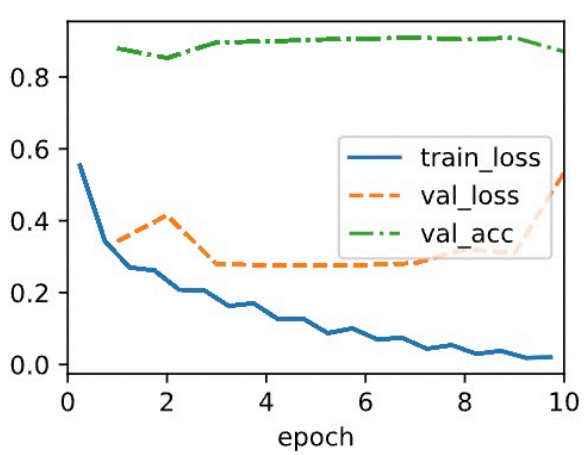
```
class ResNet(d2l.Classifier):
    def b1(self):
        return nn.Sequential(
            nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),
            nn.LazyBatchNorm2d(), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1))


@d2l.add_to_class(ResNet)
def block(self, num_residuals, num_channels, first_block=False):
    blk = []
    for i in range(num_residuals):
        if i == 0 and not first_block:
            blk.append(Residual(num_channels, use_1x1conv=True, strides=2))
        else:
            blk.append(Residual(num_channels))
    return nn.Sequential(*blk)


@d2l.add_to_class(ResNet)
def __init__(self, arch, lr=0.1, num_classes=10):
    super(ResNet, self).__init__()
    self.save_hyperparameters()
    self.net = nn.Sequential(self.b1())
    for i, b in enumerate(arch):
```

```
        self.net.add_module(f'b{i+2}', self.block(*b, first_block=(i==0)))
    self.net.add_module('last', nn.Sequential(
        nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
        nn.LazyLinear(num_classes)))
    self.net.apply(d2l.init_cnn)


class ResNet18(ResNet):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__(((2, 64), (2, 128), (2, 256), (2, 512)),
                         lr, num_classes)

ResNet18().layer_summary((1, 1, 96, 96))
```

```
Sequential output shape:         torch.Size([1, 64, 24, 24])
Sequential output shape:         torch.Size([1, 64, 24, 24])
Sequential output shape:         torch.Size([1, 128, 12, 12])
Sequential output shape:         torch.Size([1, 256, 6, 6])
Sequential output shape:         torch.Size([1, 512, 3, 3])
Sequential output shape:         torch.Size([1, 10])
```

```
model = ResNet18(lr=0.01)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
trainer.fit(model, data)
```

```
trainer.fit(model, data)
```



Takeaways/Discussions

Simply adding layers to a neural network doesn't guarantee improved expressiveness unless the architecture is designed to incorporate identity mappings effectively.

The concept of learning residual mappings allows for easier training of deep networks, reducing the complexity of the optimization problem.

ResNet's design promotes ease of modification and widespread applicability.

By introducing **grouped convolution**s, ResNeXt maintains efficiency while expanding the network's capacity to learn complex representations without excessive computational costs.

The performance of deep learning models like ResNet is significantly influenced by the amount of training data available, emphasizing the need for robust datasets in training.

What are some potential downsides or limitations of using deep residual networks in practice, and how might we address them?

Computational cost might increase as it requires substantial amount of resources for training.