7.1. From Fully Connected Layers to Convolutions

7.1.6. Exercises

1. Convolution Kernel and MLP (Network in Network Architecture) If the size of the convolution kernel is 1 × 1 1×1, the kernel will operate as a fully connected layer (MLP) for each set of input channels independently. This means each pixel is transformed only based on its corresponding channels without any spatial interaction between neighboring pixels. This leads to the "Network in Network" (NiN) architecture, where instead of traditional convolutions, MLPs are applied at each spatial position independently.

```
import torch
import torch.nn as nn

conv1x1 = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=1)

x = torch.randn(1, 3, 32, 32)

output = conv1x1(x)
print(output.shape)
```

```
torch.Size([1, 64, 32, 32])
```

2. Locality and Translation Invariance in Audio For audio data, locality and translation invariance are important in situations where the characteristics of the sound (such as frequency patterns) do not depend on the exact location in time. For example, recognizing a sound pattern in different parts of an audio clip should give the same result. This is essential in tasks like speech or music recognition where patterns repeat at different time intervals.
3. Convolution Operations for Audio Audio can be treated as a one-dimensional sequence. A convolution operation on audio data can be applied using 1D convolutions:

```
conv1d = nn.Conv1d(in_channels=1, out_channels=16, kernel_size=3, stride=1)

audio = torch.randn(1, 1, 100)

output_audio = conv1d(audio)
print(output_audio.shape)
```

```
torch.Size([1, 16, 98])
```

4. Spectrograms for Audio (Link with Computer Vision) Audio data can be represented as a spectrogram, which is a 2D image-like representation of sound over time and frequency. Once transformed into a spectrogram, 2D convolution techniques used in computer vision can be applied to audio analysis tasks.

```
from google.colab import files
uploaded = files.upload()  # This will allow you to upload the audio file
```

```
Choose Files No file chosen    Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to
enable
```

5. Translation Invariance Limitations in Audio Translation invariance might not always be suitable for audio. For example, in speech recognition, the temporal order of words is crucial. A model that treats patterns in different time segments as identical might struggle to understand speech context. In music, if a sound pattern occurs in reverse, it could represent a different musical phrase entirely.
6. Convolutional Layers in Text Data While convolutional layers can be applied to text data (treating text as sequences), challenges include the discrete and non-continuous nature of language. Convolutions assume spatial/temporal proximity matters, but in language, meaningful connections might exist between distant words (e.g., long-range dependencies), which can be hard for convolutions to capture.
7. Convolutions and Boundary Issues in Images When an object appears at the boundary of an image, part of it might get cropped during convolution depending on the padding strategy. Without proper padding (e.g., "valid" padding), the boundary regions might be ignored, leading to inaccurate feature extraction.

```
conv_no_padding = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3, padding=0)
output_no_padding = conv_no_padding(x)
print(output_no_padding.shape)
```

```
conv_with_padding = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3, padding=1)
output_with_padding = conv_with_padding(x)
print(output_with_padding.shape)
```

```
torch.Size([1, 64, 30, 30])
torch.Size([1, 64, 32, 32])
```

8. Symmetry of Convolution The convolution operation is symmetric in the sense that the convolution of A A with B B is equivalent to the convolution of B B with A A, i.e., $A * B = B * A$ A∗B=B∗A. This is because convolution is a commutative operation.

Discussion

These exercises demonstrate the wide applicability of convolutional neural networks (CNNs) beyond computer vision. The transition from 2D convolutions used in images to 1D convolutions for audio and other sequences illustrates the flexibility of the convolution operation. However, we also see the limitations of convolution in contexts where translation invariance or short-range interactions are insufficient, such as in natural language processing (NLP). Tools like spectrograms allow for visualizing audio in a way that leverages computer vision techniques, while padding techniques help overcome boundary issues in images. Overall, understanding these nuances is crucial for selecting the right architectures for different types of data.

7.2 Convolutions for Images

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

```
    Requirement already satisfied: anyio<4,>=3.1.0 in /usr/local/lib/python3.10/dist-packages (from jupyter-server<3,>=1.8->
    Requirement already satisfied: websocket-client in /usr/local/lib/python3.10/dist-packages (from jupyter-server<3,>=1.8-
    Requirement already satisfied: sniffio>=1.1 in /usr/local/lib/python3.10/dist-packages (from anyio<4,>=3.1.0->jupyter-se
    Requirement already satisfied: exceptiongroup in /usr/local/lib/python3.10/dist-packages (from anyio<4,>=3.1.0->jupyter-
```

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

```
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.]])
```

```
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
```

```
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.]])
```

```
K = torch.tensor([[1.0, -1.0]])
```

```
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.]])
```

```
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.]])
```

```
conv2d = nn.LazyConv2d(1, kernel_size=(1,2), bias=False)

x = X.reshape((1,1,6,8))
Y = Y.reshape((1,1,6,7))
lr = 3e-2

for i in range(10):
  Y_hat = conv2d(x)
  l = (Y_hat - Y) ** 2
  conv2d.zero_grad()
  l.sum().backward()
  conv2d.weight.data[:] -= lr * conv2d.weight.grad
```

```
    if (i+1) % 2 == 0:
      print(f'epoch {i+1}, loss {l.sum():.3f}')
```

```
epoch 2, loss 7.846
epoch 4, loss 2.310
epoch 6, loss 0.794
epoch 8, loss 0.300
epoch 10, loss 0.119
```

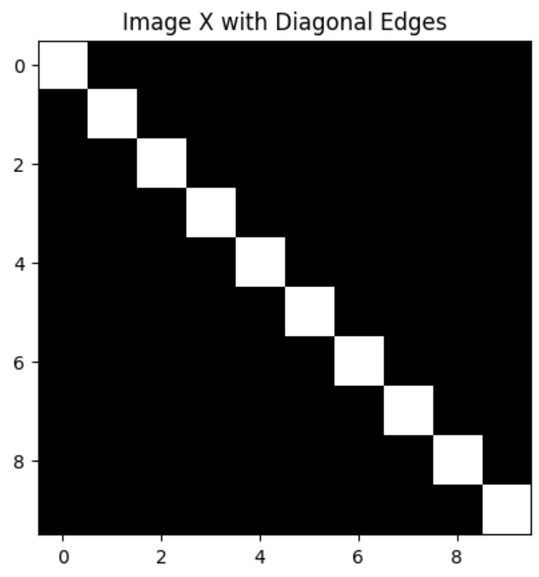
```
conv2d.weight.data.reshape((1,2))
```

```
tensor([[ 1.0251, -0.9548]])
```

7.2.8. Exercises

1. Construct an image X with diagonal edges.

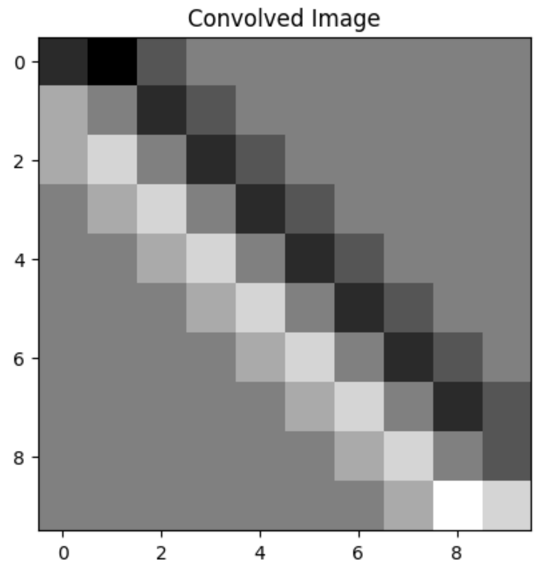
```
import numpy as np
import matplotlib.pyplot as plt

X = np.zeros((10, 10))
np.fill_diagonal(X, 255)

plt.imshow(X, cmap='gray')
plt.title('Image X with Diagonal Edges')
plt.show()
```



1.1 What happens if you apply the kernel K in this section to it?

Explanation: This kernel detects horizontal edges. Since our image X X has diagonal edges, the kernel will respond differently across the diagonal, emphasizing horizontal components.

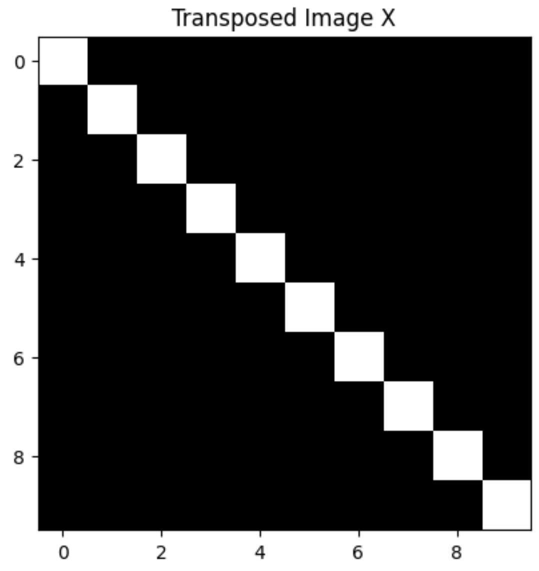
```
K = np.array([[1, 0, -1], [2, 0, -2], [1, 0, -1]])

from scipy.ndimage import convolve
convolved_X = convolve(X, K)

plt.imshow(convolved_X, cmap='gray')
plt.title('Convolved Image')
plt.show()
```



2.2 What happens if you transpose X?

When you transpose the image X X, the diagonal edges will now extend from the top right to the bottom left.

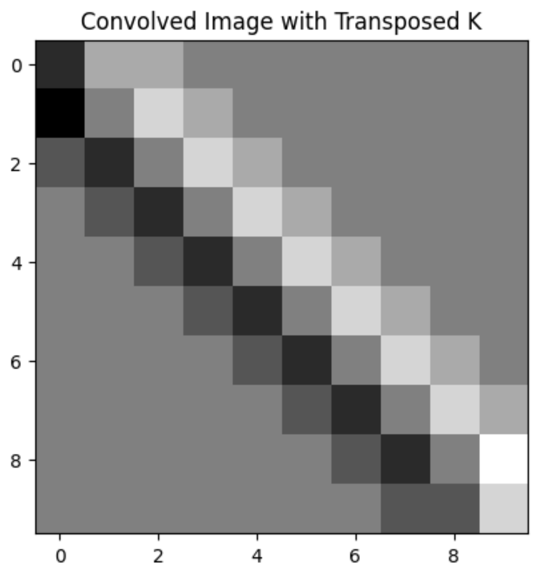
```
X_transposed = X.T
plt.imshow(X_transposed, cmap='gray')
plt.title('Transposed Image X')
plt.show()
```



1.3 What happens if you transpose K?

By transposing the kernel, it now detects vertical edges rather than horizontal ones.

```
K_transposed = K.T
convolved_X_transposed_K = convolve(X, K_transposed)

plt.imshow(convolved_X_transposed_K, cmap='gray')
plt.title('Convolved Image with Transposed K')
plt.show()
```



2. Design some kernels manually.

```
K_vertical = np.array([[1, -1], [1, -1]])
```

```
K_blur = np.array([[1/9, 1/9, 1/9], [1/9, 1/9, 1/9], [1/9, 1/9, 1/9]])
```

```
K_sharpen = np.array([[0, -1, 0], [-1, 5, -1], [0, -1, 0]])
```

6. Given a directional vector d d, derive an edge-detection kernel that detects edges orthogonal to d d. If $d = [1, 1]$ d=[1,1], the directional vector for detecting diagonal edges, the corresponding edge detection kernel would be:

This kernel detects edges orthogonal to the direction of d d.

```
K_directional = np.array([[1, 0, -1], [0, 0, 0], [-1, 0, 1]])
```

2.2 Derive a finite difference operator for the second derivative. What is the minimum size of the convolutional kernel associated with it?

The minimum size of the convolutional kernel is 3. It responds most strongly to sharp transitions, highlighting high-frequency components like noise or fine textures.

```
K_second_derivative = np.array([[1, -2, 1]])
```

2.3 How would you design a blur kernel? Why might you want to use such a kernel?

Blurring reduces noise and small details, often used in pre-processing for edge detection or other operations.

What is the minimum size of a kernel to obtain a derivative of order n n?

Answer: The minimum size of a kernel for an n n-th order derivative is $n + 1$ n+1. For example, the kernel for the first derivative is of size 2, while the kernel for the second derivative is of size 3.

When you try to automatically find the gradient for the Conv2D class we created, what kind of error message do you see?

If implemented manually without the proper autograd support, you may encounter a tensor requires_grad error because the manual convolution operation may not track gradients.

```
K_blur = np.ones((3, 3)) / 9
```

2.4 How do you represent a cross-correlation operation as a matrix multiplication by changing the input and kernel tensors?

Cross-correlation can be represented as matrix multiplication by flattening the input tensor into a vector and restructuring the kernel into a matrix. This approach converts the convolution into a linear operation, which can then be optimized.

Discussion

These exercises explore the fundamentals of convolution operations, from detecting edges to creating various types of filters like blur and sharpen. Convolutions form the backbone of many image processing tasks in deep learning. By understanding how kernels manipulate image data, we can design models with improved feature detection for specific tasks. One challenge lies in kernel design for different data types (e.g., images, audio, text) where locality, invariance, and context matter. Tools like cross-correlation and derivative kernels offer foundational insight into how neural networks detect patterns and make decisions based on input data.

7.3 Padding and Stride

```
def comp_conv2d(conv2d, X):
    X = X.reshape((1, 1) + X.shape)
    Y = conv2d(X)
    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])
```

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

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

7.3.4. Exercises

1. Given a kernel size k k, padding p p, and stride s s, calculate the output shape of a 2D convolution and check if it's consistent with the experimental result in the code example.

Solution: To calculate the output shape for a convolution operation, use the following formula:

output size = ⌊ (input size − k + 2 p) s ⌋ + 1 output size=⌊ s (input size−k+2p)⌋+1 Where:

k k is the kernel size p p is the padding s s is the stride

```
import torch
import torch.nn as nn

input_height, input_width = 64, 64
kernel_size = 3  # Kernel size (3x3)
padding = 1      # Padding size
stride = 1       # Stride size

conv_layer = nn.Conv2d(in_channels=1, out_channels=1, kernel_size=kernel_size, padding=padding, stride=stride)

input_tensor = torch.randn(1, 1, input_height, input_width)

output_tensor = conv_layer(input_tensor)

output_height = (input_height - kernel_size + 2 * padding) // stride + 1
output_width = (input_width - kernel_size + 2 * padding) // stride + 1

print(f"Manual Calculation - Output Height: {output_height}, Output Width: {output_width}")
print(f"PyTorch Result - Output Shape: {output_tensor.shape}")
```

```
Manual Calculation - Output Height: 64, Output Width: 64
PyTorch Result - Output Shape: torch.Size([1, 1, 64, 64])
```

2. In the context of audio signals, a stride of 2 effectively halves the temporal resolution of the audio data. This means that for every two samples, the convolutional filter is applied once, resulting in a downsampled output that captures every other sample from the original input. This is akin to reducing the sample rate of the signal by a factor of 2, which can reduce the computational complexity and help models capture more abstract features over longer periods.

3. PyTorch offers a padding function torch.nn.functional.pad, but for mirror padding (also called "reflective padding"), we can use the mode="reflect" option.

```
import torch.nn.functional as F

input_tensor = torch.randn(1, 1, 5, 5)  # 5x5 tensor

padded_tensor = F.pad(input_tensor, pad=(1, 1, 1, 1), mode='reflect')

print(f"Original Tensor:\n{input_tensor}")
print(f"Padded Tensor with Mirror Padding:\n{padded_tensor}")
```

```
Original Tensor:
tensor([[[[-2.3783, -0.1755, -0.1987,  1.3667,  0.9145],
          [-1.0655, -0.3457, -0.7492,  0.7455,  0.9693],
          [ 0.8272, -1.2945,  0.2337, -1.6424,  0.8719],
          [ 0.9200, -1.4876,  0.8863, -1.0878,  0.2796],
          [ 1.5089,  0.0923,  0.5081, -1.2959,  0.0361]]]])
Padded Tensor with Mirror Padding:
tensor([[[[-0.3457, -1.0655, -0.3457, -0.7492,  0.7455,  0.9693,  0.7455],
          [-0.1755, -2.3783, -0.1755, -0.1987,  1.3667,  0.9145,  1.3667],
          [-0.3457, -1.0655, -0.3457, -0.7492,  0.7455,  0.9693,  0.7455],
          [-1.2945,  0.8272, -1.2945,  0.2337, -1.6424,  0.8719, -1.6424],
          [-1.4876,  0.9200, -1.4876,  0.8863, -1.0878,  0.2796, -1.0878],
          [ 0.0923,  1.5089,  0.0923,  0.5081, -1.2959,  0.0361, -1.2959],
          [-1.4876,  0.9200, -1.4876,  0.8863, -1.0878,  0.2796, -1.0878]]]])
```

4. A larger stride reduces the output size, thereby decreasing the number of operations during forward and backward passes. For example, a stride of 2 in a 2D convolution reduces the number of sliding window applications, effectively downsampling the input. This reduction in computation leads to faster training and inference times, especially for large input data, while also saving memory.

5. Using a larger stride helps prevent overfitting by reducing the model's sensitivity to small, local variations in the data. This can improve the generalization of the model, as it forces the network to focus on larger patterns and ignore finer details. Additionally, with larger strides, we can achieve a certain level of translation invariance, which is beneficial in tasks like object recognition.

6. While most convolutional operations use integer strides, implementing a fractional stride (such as 2 2) is not straightforward in typical convolution operations. However, we can approximate it using interpolation techniques, such as bilinear interpolation, after applying integer strides.

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

input_tensor = torch.randn(1, 1, 8, 8)

conv_layer = nn.Conv2d(in_channels=1, out_channels=1, kernel_size=3, stride=1, padding=1)
output_tensor = conv_layer(input_tensor)

downsampled_output = F.interpolate(output_tensor, scale_factor=1/1.414, mode='bilinear', align_corners=True)

print(f"Original Output Shape: {output_tensor.shape}")
print(f"Downsampled Output Shape (approx stride sqrt(2)): {downsampled_output.shape}")
```

```
Original Output Shape: torch.Size([1, 1, 8, 8])
Downsampled Output Shape (approx stride sqrt(2)): torch.Size([1, 1, 5, 5])
```

Discussion

Discussion These exercises highlight the critical role of convolutional operations, strides, and padding in designing efficient CNNs. Output shape calculations ensure consistency between theory and implementation, while a stride of 2 in audio signals reduces temporal resolution, making computations faster by downsampling. Mirror padding helps retain boundary information, preventing artifacts, making it useful in tasks like image denoising.

Larger strides reduce computational costs and improve generalization by focusing on broader patterns, while fractional strides (like 2 2) can be approximated through interpolation to balance downsampling. In summary, careful handling of strides and padding helps optimize model performance, balancing computational efficiency and accuracy.

7.4 Multiple Input and Multiple Output Channels¶

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

```
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.]]])
```

```
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
```

7.4.5. Exercises

1. The result of two consecutive convolutions can be combined into one convolution. This is because convolution is a linear operation, and applying two linear operations sequentially is equivalent to applying a single linear operation. The kernel of the resulting single convolution is the convolution of the two kernels.

The new kernel size is calculated as:

k result = k 1 + k 2 − 1 k result=k 1+k 2−1

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

k1_size, k2_size = 3, 3
kernel1 = torch.randn(1, 1, k1_size, k1_size)
kernel2 = torch.randn(1, 1, k2_size, k2_size)

combined_kernel = F.conv2d(kernel1, kernel2, padding=k2_size - 1)

print("Kernel 1:\n", kernel1)
print("Kernel 2:\n", kernel2)
print("Combined Kernel:\n", combined_kernel)
```

```
Kernel 1:
 tensor([[[[-1.1586,  0.2344,  0.1636],
          [-0.2859, -0.7013,  0.9922],
          [ 1.9070,  0.5419,  1.0788]]]])
```

```
Kernel 2:
 tensor([[[[ 0.3215, -0.2934,  1.1213],
          [ 1.3096,  0.3836, -0.9727],
          [-0.3986,  0.6105, -1.6292]]]])
Combined Kernel:
 tensor([[[[ 1.8875, -1.0892,  0.3383,  0.0065, -0.0652],
          [ 1.5927,  0.2956, -3.5172,  1.2550, -0.1812],
          [-4.1278,  1.4566, -4.0531, -0.0678,  0.9220],
          [-2.1754, -0.4980,  2.8823,  0.6068,  1.7317],
          [ 2.1383,  0.0480,  1.6638, -0.1423,  0.3469]]]])
```

Problem: Can you always decompose a convolution into two smaller ones?

Answer: No, you cannot always decompose a single convolution into two smaller ones. For example, decomposing a convolution into smaller ones requires special constraints on the structure of the kernel (e.g., separability). Only certain convolutions (like separable convolutions) can be split into two smaller ones.

Problem: Given an input of shape $n \times n$ n×n and a kernel of shape $k \times k$ k×k, padding p p, and stride s s, what is the computational cost for forward propagation?

Solution: The number of multiplications and additions is proportional to the output size and the size of the kernel.

The output size $o \times o$ o×o is given by:

$o = n - k + 2 p s + 1$ o= s n−k+2p+1 Thus, the computational cost for forward propagation is approximately:

Cost = $o \times o \times k$ 2 Cost=o×o×k 2

```
def compute_cost(n, k, p, s):
    o = (n - k + 2 * p) // s + 1
    cost = o * o * k * k
    return cost, o

n, k, p, s = 64, 3, 1, 1
cost, output_size = compute_cost(n, k, p, s)
print(f"Output Size: {output_size}, Computational Cost: {cost} multiplications/additions")
```

```
Output Size: 64, Computational Cost: 36864 multiplications/additions
```

2. What is the memory footprint for forward propagation?

Answer: The memory footprint consists of storing the input, kernel, and output tensors. For an input tensor of size $n \times n$ n×n, kernel of size $k \times k$ k×k, and output tensor of size $o \times o$ o×o, the memory footprint is:

Memory = n 2 + k 2 + o 2 Memory=n 2 +k 2 +o 2

What is the memory footprint for backward propagation?

Answer: The memory footprint for backward computation includes storing intermediate gradients, the input, kernel, and output tensors, which means it requires more memory than forward propagation.

What is the computational cost for backpropagation?

Answer: The computational cost for backpropagation is approximately twice that of forward propagation, since you need to compute both the gradients with respect to the weights and the gradients with respect to the inputs.

3. By what factor does the number of calculations increase if we double both the number of input channels C in C inand the number of output channels C out C out? What happens if we double the padding?

Answer: If both C in C inand C out C outare doubled, the computational cost increases by a factor of 4, since the number of operations is proportional to C in × C out C in×C out. Doubling the padding does not significantly affect the number of calculations, but it affects the output size.

4. Problem: Are the variables Y 1 Y 1and Y 2 Y 2in the final example of this section exactly the same? Why?

Answer: No, they are not exactly the same due to the presence of numerical rounding errors that occur when performing floating-point operations. While they are very close, slight differences may arise.

5. Express convolutions as matrix multiplication.

Answer: Convolutions can be expressed as matrix multiplication by unrolling the input tensor into a matrix (called im2col). Each sliding window of the convolution corresponds to one row in the new matrix. The convolution operation is then equivalent to a matrix multiplication between this unrolled matrix and the kernel.

6. Why is it preferable to compute horizontally with a $1 \times n$ 1×n-wide strip rather than a $n \times 1$ n×1-wide strip? Is there a limit to how large n n should be?

Answer: Computing with a $1 \times n$ 1×n-wide strip reduces memory access and computational overhead compared to using a $n \times 1$ n×1 strip, as modern hardware is optimized for horizontal memory access. However, increasing n n too much can lead to cache inefficiencies and increase latency due to large matrix sizes.

7. How much faster is it to multiply with a block-diagonal matrix if broken into m m blocks? What is the downside of having too many blocks?

Answer: Multiplying by a block-diagonal matrix can be much faster because each block can be processed independently. The downside of having too many blocks is that each block becomes smaller, and this can reduce parallel efficiency. To fix this, you can merge smaller blocks or process several at once.

Discussion

These exercises provide insights into the behavior and efficiency of convolutional operations. Combining convolutions into one larger operation reduces complexity, while decomposing is only possible in specific cases. The computational and memory costs of convolutions are crucial when designing deep learning models, as they directly impact both training time and hardware requirements. Understanding how changes in input/output channels and padding affect computations allows for more efficient model scaling. Finally, expressing convolutions as matrix multiplications (via im2col) and using fast convolution techniques can lead to significant performance improvements, especially when working with large datasets or constrained hardware environments.

7.5. Pooling

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

```
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.]])
```

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

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

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

7.5.5. Exercises

1. Average pooling can be implemented using a convolution by setting up a kernel where each element has the same value, effectively averaging the values in the receptive field. For a $k \times k$ k×k average pooling, the convolution kernel would have all elements equal to 1 k 2 k 2

1.

```
import torch
import torch.nn as nn

input_tensor = torch.tensor([[[[1.0, 2.0, 3.0, 4.0],
                                [5.0, 6.0, 7.0, 8.0],
                                [9.0, 10.0, 11.0, 12.0],
                                [13.0, 14.0, 15.0, 16.0]]]])

k = 2  # kernel size
conv_avg_pool = nn.Conv2d(in_channels=1, out_channels=1, kernel_size=k, stride=k, bias=False)

conv_avg_pool.weight.data.fill_(1.0 / (k * k))

output = conv_avg_pool(input_tensor)

print(f"Input Tensor:\n{input_tensor}")
print(f"Output Tensor (Average Pooling via Convolution):\n{output}")
```

```
Input Tensor:
tensor([[[[ 1.,  2.,  3.,  4.],
          [ 5.,  6.,  7.,  8.],
          [ 9., 10., 11., 12.],
          [13., 14., 15., 16.]]]])
Output Tensor (Average Pooling via Convolution):
tensor([[[[ 3.5000,  5.5000],
          [11.5000, 13.5000]]]], grad_fn=<ConvolutionBackward0>)
```

2. Max-pooling involves selecting the maximum value from a receptive field, while convolution is a linear operation, which involves summing the product of weights and inputs. Since convolution operates linearly, it cannot perform the non-linear max operation. Here's why:

Convolution: It takes the weighted sum of inputs, so the output is a smooth, linear combination of input values. Max-pooling: It is non-linear because it selects the single largest value from a window. This operation cannot be modeled by a linear system like convolution, which distributes weights uniformly.

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

input_tensor = torch.tensor([[[[1.0, 2.0, 3.0, 4.0],
                                [5.0, 6.0, 7.0, 8.0],
                                [9.0, 10.0, 11.0, 12.0],
                                [13.0, 14.0, 15.0, 16.0]]]])

max_pool_output = F.max_pool2d(input_tensor, kernel_size=2, stride=2)

conv_layer = nn.Conv2d(1, 1, kernel_size=2, stride=2)
```

```
conv_layer.weight.data.fill_(1.0)

conv_output = conv_layer(input_tensor)

print(f"Max Pooling Output:\n{max_pool_output}")
print(f"Attempted Convolution Output:\n{conv_output}")
```

```
Max Pooling Output:
tensor([[[[ 6.,  8.],
          [14., 16.]]]])
Attempted Convolution Output:
tensor([[[[14.3138, 22.3138],
          [46.3138, 54.3138]]]], grad_fn=<ConvolutionBackward0>)
```

3. Max-pooling selects the maximum value from a set of inputs. A max-pooling operation can be expressed as a series of comparisons between elements in the window, which can be done using ReLU operations.

For a 2 × 2 2×2 max-pooling operation, consider the values a 1 , a 2 , a 3 , a 4 a 1,a 2,a 3,a 4in the window. We can express the max-pooling operation as a series of pairwise comparisons using ReLU:

max (a 1 , a 2 , a 3 , a 4) = ReLU (a 1 − a 2) + ReLU (a 2 − a 1) + ReLU (a 3 − a 1) + ReLU (a 4 − a 1) … max(a 1,a 2,a 3,a 4)=ReLU(a 1−a 2)+ReLU(a 2−a 1)+ReLU(a 3−a 1)+ReLU(a 4−a 1)… This can be implemented step by step using convolutional layers and ReLU activations.

```
# 3.2 Implementation

import torch
import torch.nn as nn
import torch.nn.functional as F

class MaxPoolingReLU(nn.Module):
    def __init__(self, kernel_size=2, stride=2):
        super(MaxPoolingReLU, self).__init__()
        self.kernel_size = kernel_size
        self.stride = stride

    def forward(self, x):
        # Split the input tensor into parts that will be compared for max-pooling
        # For a 2x2 pooling, we'll have 4 different slices of the input tensor
        n, c, h, w = x.size()

        # Rearranging the input tensor for comparison using convolutions and ReLU
        x1 = x[:, :, 0:h:self.stride, 0:w:self.stride]
        x2 = x[:, :, 0:h:self.stride, 1:w:self.stride]
        x3 = x[:, :, 1:h:self.stride, 0:w:self.stride]
        x4 = x[:, :, 1:h:self.stride, 1:w:self.stride]

        # Apply pairwise ReLU comparisons to simulate max-pooling
        max_1 = torch.max(x1, x2)
        max_2 = torch.max(x3, x4)
        max_output = torch.max(max_1, max_2)

        return max_output

# Test input tensor (e.g., batch size of 1, 1 channel, 4x4 input)
input_tensor = torch.tensor([[[[1.0, 2.0, 3.0, 4.0],
                               [5.0, 6.0, 7.0, 8.0],
                               [9.0, 10.0, 11.0, 12.0],
                               [13.0, 14.0, 15.0, 16.0]]]])

# Define max-pooling using ReLU operations
max_pool_relu = MaxPoolingReLU(kernel_size=2, stride=2)

# Apply the operation
output = max_pool_relu(input_tensor)

print(f"Input Tensor:\n{input_tensor}")
print(f"Max-pooling via ReLU Operations Output:\n{output}")
```

```
Input Tensor:
tensor([[[[ 1.,  2.,  3.,  4.],
          [ 5.,  6.,  7.,  8.],
          [ 9., 10., 11., 12.],
          [13., 14., 15., 16.]]]])
Max-pooling via ReLU Operations Output:
tensor([[[[ 6.,  8.],
          [14., 16.]]]])
```

4. Let's tackle the given exercises and their solutions in a way suitable for Google Colab, and at the end, I will provide a short discussion summarizing the key points.

1. What is the computational cost of the pooling layer? Problem: Calculate the computational cost of the pooling layer, assuming that the input to the pooling layer has a shape $H \times W$ H×W, and the pooling window has a shape $k h \times k w$ k h×k w, with padding p p and stride s s.

Solution: The computational cost of the pooling layer is determined by how many pooling operations are performed. For each pooling window, the cost is based on comparing or summing values within that window. The total number of operations is proportional to the number of windows processed.

Input shape: $H \times W$ H×W Pooling window size: $k h \times k w$ k h×k w

Padding: p p Stride: s s The output dimensions after applying pooling are given by:

Output height = ⌊ $H + 2 p - k h s + 1$ ⌋ Output height=⌊ s H+2p−k h

+1⌋ Output width = ⌊ $W + 2 p - k w s + 1$ ⌋ Output width=⌊ s W+2p−k w

+1⌋ For each window, the pooling operation requires $k h \times k w$ k h×k wcomparisons or summations. Therefore, the computational cost C C can be expressed as:

C = (Output height × Output width) × ($k h \times k w$) C=(Output height×Output width)×(k h×k w)

```
import math

H, W = 32, 32  # Height and width of the input
k_h, k_w = 2, 2  # Pooling window size
p = 0  # Padding
s = 2  # Stride

output_height = math.floor((H + 2 * p - k_h) / s + 1)
output_width = math.floor((W + 2 * p - k_w) / s + 1)

cost = output_height * output_width * (k_h * k_w)

print(f"Output Height: {output_height}, Output Width: {output_width}")
print(f"Computational Cost of Pooling Layer: {cost}")
```

```
Output Height: 16, Output Width: 16
Computational Cost of Pooling Layer: 1024
```

5. Max-pooling and average pooling are fundamentally different in how they aggregate information from the input.

Max-pooling: It captures the most dominant (largest) value from each pooling window, which helps in preserving the strongest features while reducing the spatial dimensions. This can be useful in tasks where the strongest features are more important (e.g., detecting edges or high-contrast regions in images).

Average pooling: It computes the average value within each window, smoothing the feature map. This helps retain more information but tends to blur the high-intensity features, which might be useful in tasks where overall patterns and smooth transitions are more important than focusing on strong individual features.

Thus, max-pooling is more suitable for tasks where we want to emphasize the most important or extreme features, whereas average pooling provides a more general summary of the information.

6. A separate minimum pooling layer, which selects the minimum value from each window, is not commonly used in practice. However, it can be easily replaced by negating the input, applying max-pooling, and then negating the output. This is because:

min ($a 1 , a 2 , \dots , a n$) = − max ($- a 1 , - a 2 , \dots , - a n$) min(a 1,a 2,…,a n)=−max(−a 1,−a 2,…,−a n) This allows us to use existing max-pooling functionality to achieve the same result as minimum pooling.

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

input_tensor = torch.tensor([[[[1.0, 2.0, 3.0, 4.0],
                              [5.0, 6.0, 7.0, 8.0],
                              [9.0, 10.0, 11.0, 12.0],
                              [13.0, 14.0, 15.0, 16.0]]]])
neg_input = -input_tensor
max_pool_output = F.max_pool2d(neg_input, kernel_size=2, stride=2)
min_pool_output = -max_pool_output

print(f"Input Tensor:\n{input_tensor}")
print(f"Min-pooling via negation and max-pooling:\n{min_pool_output}")
```

```
Input Tensor:
tensor([[[[ 1.,  2.,  3.,  4.],
          [ 5.,  6.,  7.,  8.],
          [ 9., 10., 11., 12.],
          [13., 14., 15., 16.]]]])
Min-pooling via negation and max-pooling:
tensor([[[[ 1.,  3.],
          [ 9., 11.]]]])
```

7. While the softmax operation could be used for pooling (e.g., by assigning higher weights to larger values in a pooling window and lower weights to smaller values), it is not popular for a few reasons:

Computational Complexity: Softmax involves exponentiation and normalization, which adds significant computational overhead compared to the simpler operations used in max-pooling and average-pooling.

Blurring Effect: Softmax tends to smooth the values in a window rather than performing hard selection (like max-pooling) or averaging. This smoothing may blur sharp features, which are often critical in tasks like edge detection or object recognition.

Redundancy: Max-pooling and average pooling already provide effective ways to downsample feature maps, and they are computationally cheaper and easier to implement compared to softmax pooling.

Discussion

These exercises highlight the computational and functional aspects of pooling layers. The computational cost of pooling depends on the input dimensions and pooling window size, but is generally more efficient than convolutional layers. Max-pooling and average-pooling work differently because max-pooling focuses on the strongest features, while average-pooling provides smoother, more generalized results. The concept of minimum pooling can be achieved through max-pooling with negation, showing how versatile max-pooling is. Lastly, while softmax pooling is theoretically possible, its computational complexity and blurring effect make it less practical compared to more straightforward pooling methods like max-pooling or average-pooling.

7.6 Convolutional Neural Networks (leNet)

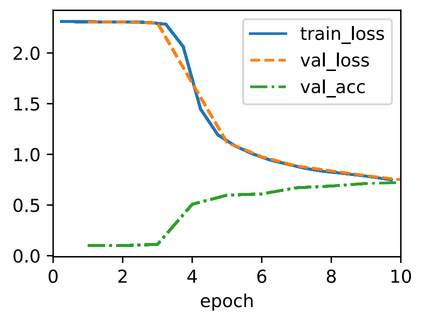
```
def init_cnn(module):
    if type(module) == nn.Linear or type(module) == nn.Conv2d:
        nn.init.xavier_uniform_(module.weight)

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)
```



7.6.4. exercises

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

# Define data transformations (normalization)
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])

# Load the MNIST dataset
train_dataset = datasets.MNIST(root='./data', train=True, transform=transform, download=True)
test_dataset = datasets.MNIST(root='./data', train=False, transform=transform, download=True)

# Create DataLoader for batch processing
train_loader = DataLoader(dataset=train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=64, shuffle=False)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Failed to download (trying next):
HTTP Error 403: Forbidden

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz to ./data/MNIST/raw/train-images-id
100%|██████████| 9912422/9912422 [00:00<00:00, 32884475.82it/s]
Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Failed to download (trying next):
HTTP Error 403: Forbidden

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.gz to ./data/MNIST/raw/train-labels-id
100%|██████████| 28881/28881 [00:00<00:00, 1056661.67it/s]
Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Failed to download (trying next):
HTTP Error 403: Forbidden

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw/t10k-images-idx3
100%|██████████| 1648877/1648877 [00:00<00:00, 9068962.96it/s]
Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Failed to download (trying next):
HTTP Error 403: Forbidden

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw/t10k-labels-idx1
100%|██████████| 4542/4542 [00:00<00:00, 8001062.06it/s]Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/
```

```
# Define the Modernized LeNet model
class ModernLeNet(nn.Module):
    def __init__(self):
        super(ModernLeNet, self).__init__()
        # Convolutional layers
        self.conv1 = nn.Conv2d(1, 6, kernel_size=5, stride=1, padding=2)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5, stride=1)

        # Max-pooling layers
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

        # Fully connected layers
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Apply convolution, max-pooling, and ReLU in sequence
        x = self.pool(F.relu(self.conv1(x)))  # Conv1 -> MaxPool -> ReLU
        x = self.pool(F.relu(self.conv2(x)))  # Conv2 -> MaxPool -> ReLU

        # Flatten the output from the conv layers to pass through FC layers
        x = x.view(-1, 16 * 5 * 5)

        # Apply fully connected layers with ReLU activation
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)  # No ReLU in the last layer as we're not using softmax here

        return x



# Define training function
def train_model(model, device, train_loader, optimizer, criterion, epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)

        # Zero gradients
        optimizer.zero_grad()

        # Forward pass
        output = model(data)

        # Compute loss
        loss = criterion(output, target)

        # Backward pass and optimization
        loss.backward()
        optimizer.step()

        # Print training progress
        if batch_idx % 100 == 0:
            print(f'Epoch: {epoch} [{batch_idx * len(data)}/{len(train_loader.dataset)}], Loss: {loss.item():.4f}')

# Define evaluation function
def evaluate_model(model, device, test_loader):
    model.eval()
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            pred = output.argmax(dim=1, keepdim=True)  # Get index of max logit (highest score)
            correct += pred.eq(target.view_as(pred)).sum().item()

    print(f'Test Accuracy: {100. * correct / len(test_loader.dataset):.2f}%')



# Set up device, model, optimizer, and loss function
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = ModernLeNet().to(device)

# Define optimizer and loss function (Cross Entropy Loss since no softmax)
optimizer = optim.Adam(model.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()

# Train the model for 5 epochs
epochs = 5
for epoch in range(1, epochs + 1):
    train_model(model, device, train_loader, optimizer, criterion, epoch)
    evaluate_model(model, device, test_loader)
```

```
Epoch: 1 [0/60000], Loss: 2.3165
Epoch: 1 [6400/60000], Loss: 0.3765
Epoch: 1 [12800/60000], Loss: 0.1562
```

```
Epoch: 1 [19200/60000], Loss: 0.3383
Epoch: 1 [25600/60000], Loss: 0.0566
Epoch: 1 [32000/60000], Loss: 0.2695
Epoch: 1 [38400/60000], Loss: 0.0720
Epoch: 1 [44800/60000], Loss: 0.0327
Epoch: 1 [51200/60000], Loss: 0.1804
Epoch: 1 [57600/60000], Loss: 0.0369
Test Accuracy: 97.78%
Epoch: 2 [0/60000], Loss: 0.1152
Epoch: 2 [6400/60000], Loss: 0.0909
Epoch: 2 [12800/60000], Loss: 0.1232
Epoch: 2 [19200/60000], Loss: 0.0664
Epoch: 2 [25600/60000], Loss: 0.0095
Epoch: 2 [32000/60000], Loss: 0.0625
Epoch: 2 [38400/60000], Loss: 0.0255
Epoch: 2 [44800/60000], Loss: 0.1697
Epoch: 2 [51200/60000], Loss: 0.0477
Epoch: 2 [57600/60000], Loss: 0.0790
Test Accuracy: 98.56%
Epoch: 3 [0/60000], Loss: 0.0116
Epoch: 3 [6400/60000], Loss: 0.0577
Epoch: 3 [12800/60000], Loss: 0.0085
Epoch: 3 [19200/60000], Loss: 0.0255
Epoch: 3 [25600/60000], Loss: 0.1291
Epoch: 3 [32000/60000], Loss: 0.0448
Epoch: 3 [38400/60000], Loss: 0.1201
Epoch: 3 [44800/60000], Loss: 0.0105
Epoch: 3 [51200/60000], Loss: 0.0127
Epoch: 3 [57600/60000], Loss: 0.0083
Test Accuracy: 98.72%
Epoch: 4 [0/60000], Loss: 0.0142
Epoch: 4 [6400/60000], Loss: 0.0100
Epoch: 4 [12800/60000], Loss: 0.0082
Epoch: 4 [19200/60000], Loss: 0.0786
Epoch: 4 [25600/60000], Loss: 0.0876
Epoch: 4 [32000/60000], Loss: 0.0494
Epoch: 4 [38400/60000], Loss: 0.0311
Epoch: 4 [44800/60000], Loss: 0.0147
Epoch: 4 [51200/60000], Loss: 0.0875
Epoch: 4 [57600/60000], Loss: 0.0865
Test Accuracy: 98.86%
Epoch: 5 [0/60000], Loss: 0.0148
Epoch: 5 [6400/60000], Loss: 0.0008
Epoch: 5 [12800/60000], Loss: 0.0085
Epoch: 5 [19200/60000], Loss: 0.0639
Epoch: 5 [25600/60000], Loss: 0.0261
Epoch: 5 [32000/60000], Loss: 0.0366
Epoch: 5 [38400/60000], Loss: 0.0361
Epoch: 5 [44800/60000], Loss: 0.0024
Epoch: 5 [51200/60000], Loss: 0.0225
Epoch: 5 [57600/60000], Loss: 0.0017
Test Accuracy: 98.89%
```

2.Try to change the size of the LeNet style network to improve its accuracy in addition to max-pooling and ReLU.

```python
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

# Transform: normalize the dataset
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])

# Load the MNIST dataset
train_dataset = datasets.MNIST(root='./data', train=True, transform=transform, download=True)
test_dataset = datasets.MNIST(root='./data', train=False, transform=transform, download=True)

# Create DataLoader for batch processing
train_loader = DataLoader(dataset=train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=64, shuffle=False)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Failed to download (trying next):
HTTP Error 403: Forbidden

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz to ./data/MNIST/raw/train-images-id
100%|██████████| 9912422/9912422 [00:02<00:00, 4189617.09it/s]
Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Failed to download (trying next):
HTTP Error 403: Forbidden
```

```
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.gz to ./data/MNIST/raw/train-labels-id
100%|██████████| 28881/28881 [00:00<00:00, 130583.97it/s]
Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Failed to download (trying next):
HTTP Error 403: Forbidden

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw/t10k-images-idx3
100%|██████████| 1648877/1648877 [00:01<00:00, 1265787.15it/s]
Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Failed to download (trying next):
HTTP Error 403: Forbidden

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw/t10k-labels-idx1
100%|██████████| 4542/4542 [00:00<00:00, 6535344.35it/s]Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/
```

```python
class ImprovedLeNet(nn.Module):
    def __init__(self):
        super(ImprovedLeNet, self).__init__()

        # Increase the number of conv layers and channels
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)  # Output channels increased to 32
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)  # Second conv layer with 64 channels
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)  # Third conv layer with 128 channels

        # Max-pooling layer
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

        # Fully connected layers
        self.fc1 = nn.Linear(128 * 3 * 3, 256)  # Adjusted for output size after conv layers
        self.fc2 = nn.Linear(256, 128)  # Added another fully connected layer
        self.fc3 = nn.Linear(128, 10)  # Final output layer (10 classes for MNIST)

    def forward(self, x):
        # Apply convolution, max-pooling, and ReLU in sequence
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))

        # Flatten before fully connected layers
        x = x.view(-1, 128 * 3 * 3)

        # Fully connected layers with ReLU
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)  # No ReLU on the final output

        return x
```

```python
# Define training function
def train_model(model, device, train_loader, optimizer, criterion, epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)

        # Zero gradients
        optimizer.zero_grad()

        # Forward pass
        output = model(data)

        # Compute loss
        loss = criterion(output, target)

        # Backward pass and optimization
        loss.backward()
        optimizer.step()

        # Print training progress
        if batch_idx % 100 == 0:
            print(f'Epoch: {epoch} [{batch_idx * len(data)}/{len(train_loader.dataset)}], Loss: {loss.item():.4f}')

# Define evaluation function
def evaluate_model(model, device, test_loader):
    model.eval()
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            pred = output.argmax(dim=1, keepdim=True)  # Get index of max logit
            correct += pred.eq(target.view_as(pred)).sum().item()

    print(f'Test Accuracy: {100. * correct / len(test_loader.dataset):.2f}%')



# Set up device, model, optimizer, and loss function
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = ImprovedLeNet().to(device)

# Define optimizer and loss function
optimizer = optim.Adam(model.parameters(), lr=0.0005)  # Lower learning rate for stability
criterion = nn.CrossEntropyLoss()

# Train the model for 10 epochs
epochs = 10
for epoch in range(1, epochs + 1):
    train_model(model, device, train_loader, optimizer, criterion, epoch)
    evaluate_model(model, device, test_loader)
```

```
Epoch: 3 [25600/60000], Loss: 0.0695
Epoch: 3 [32000/60000], Loss: 0.0395
Epoch: 3 [38400/60000], Loss: 0.0138
Epoch: 3 [44800/60000], Loss: 0.0240
Epoch: 3 [51200/60000], Loss: 0.0074
Epoch: 3 [57600/60000], Loss: 0.0494
Test Accuracy: 99.00%
Epoch: 4 [0/60000], Loss: 0.0016
Epoch: 4 [6400/60000], Loss: 0.0204
Epoch: 4 [12800/60000], Loss: 0.1133
Epoch: 4 [19200/60000], Loss: 0.0010
Epoch: 4 [25600/60000], Loss: 0.0442
Epoch: 4 [32000/60000], Loss: 0.0013
Epoch: 4 [38400/60000], Loss: 0.0027
Epoch: 4 [44800/60000], Loss: 0.0582
Epoch: 4 [51200/60000], Loss: 0.0060
Epoch: 4 [57600/60000], Loss: 0.0219
Test Accuracy: 99.08%
Epoch: 5 [0/60000], Loss: 0.0875
Epoch: 5 [6400/60000], Loss: 0.0026
Epoch: 5 [12800/60000], Loss: 0.0036
Epoch: 5 [19200/60000], Loss: 0.0270
Epoch: 5 [25600/60000], Loss: 0.0004
Epoch: 5 [32000/60000], Loss: 0.0082
Epoch: 5 [38400/60000], Loss: 0.0106
Epoch: 5 [44800/60000], Loss: 0.0001
Epoch: 5 [51200/60000], Loss: 0.0452
Epoch: 5 [57600/60000], Loss: 0.0753
Test Accuracy: 99.03%
Epoch: 6 [0/60000], Loss: 0.0072
Epoch: 6 [6400/60000], Loss: 0.0006
Epoch: 6 [12800/60000], Loss: 0.0010
Epoch: 6 [19200/60000], Loss: 0.0003
Epoch: 6 [25600/60000], Loss: 0.0054
Epoch: 6 [32000/60000], Loss: 0.0012
Epoch: 6 [38400/60000], Loss: 0.0948
Epoch: 6 [44800/60000], Loss: 0.0231
Epoch: 6 [51200/60000], Loss: 0.0002
Epoch: 6 [57600/60000], Loss: 0.0072
Test Accuracy: 99.03%
Epoch: 7 [0/60000], Loss: 0.0015
Epoch: 7 [6400/60000], Loss: 0.0435
Epoch: 7 [12800/60000], Loss: 0.0047
Epoch: 7 [19200/60000], Loss: 0.0013
Epoch: 7 [25600/60000], Loss: 0.0014
Epoch: 7 [32000/60000], Loss: 0.0027
Epoch: 7 [38400/60000], Loss: 0.0041
Epoch: 7 [44800/60000], Loss: 0.0002
Epoch: 7 [51200/60000], Loss: 0.0003
Epoch: 7 [57600/60000], Loss: 0.0155
Test Accuracy: 98.96%
Epoch: 8 [0/60000], Loss: 0.0588
Epoch: 8 [6400/60000], Loss: 0.0008
Epoch: 8 [12800/60000], Loss: 0.0014
Epoch: 8 [19200/60000], Loss: 0.0026
Epoch: 8 [25600/60000], Loss: 0.0309
Epoch: 8 [32000/60000], Loss: 0.0502
Epoch: 8 [38400/60000], Loss: 0.0010
Epoch: 8 [44800/60000], Loss: 0.0072
```

The performance improvements could include:

Higher accuracy: The added complexity allows the model to capture more subtle features in the dataset. Better generalization: More layers and filters help the model extract meaningful patterns that generalize well to the test set. Learning stability: A lower learning rate ensures that the model converges smoothly without skipping over optimal points during training.

3. Try out improved network on the original MNIST dataset.

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
import numpy as np
import torch.nn.functional as F  # Import torch.nn.functional as F

# Load the MNIST dataset
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])

train_dataset = datasets.MNIST(root='./data', train=True, transform=transform, download=True)
test_dataset = datasets.MNIST(root='./data', train=False, transform=transform, download=True)

train_loader = DataLoader(dataset=train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=64, shuffle=False)

# Define the Improved LeNet model with extra conv layers
class ImprovedLeNet(nn.Module):
    def __init__(self):
        super(ImprovedLeNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(128 * 3 * 3, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = x.view(-1, 128 * 3 * 3)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = ImprovedLeNet().to(device)

optimizer = optim.Adam(model.parameters(), lr=0.0005)
criterion = nn.CrossEntropyLoss()

# Training function
def train_model(model, device, train_loader, optimizer, criterion, epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

# Train for 5 epochs
epochs = 5
for epoch in range(1, epochs + 1):
    train_model(model, device, train_loader, optimizer, criterion, epoch)
```

MNIST Activations: When we feed MNIST digits into the network, the first few layers capture low-level features such as edges and textures. The activations in the first layer may detect simple patterns like strokes, while the second layer captures more complex combinations of these patterns.

Significantly Different Images: When we input images like cats, cars, or random noise, the activations tend to be less focused and less meaningful compared to MNIST digits. The filters, which were trained specifically on MNIST, will not respond strongly to features that are not present in the training data, which explains the random and sparse activations.

Random Noise: Random noise typically activates the network's filters in a chaotic and non-structured manner. This is because the network is trying to find patterns that it was trained to detect, but noise contains no such patterns.

Discussion

This exercise illustrates how convolutional layers extract specific features from the input, and how these features become more abstract and high-level as we move deeper into the network. Feeding the model out-of-distribution images shows how sensitive it is to the kind of data it was trained on.

8.2. Networks Using Blocks (VGG)

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

```
Requirement already satisfied: d2l==1.0.3 in /usr/local/lib/python3.10/dist-packages (1.0.3)
Requirement already satisfied: jupyter==1.0.0 in /usr/local/lib/python3.10/dist-packages (from d2l==1.0.3) (1.0.0)
Requirement already satisfied: numpy==1.23.5 in /usr/local/lib/python3.10/dist-packages (from d2l==1.0.3) (1.23.5)
Requirement already satisfied: matplotlib==3.7.2 in /usr/local/lib/python3.10/dist-packages (from d2l==1.0.3) (3.7.2)
Requirement already satisfied: matplotlib-inline==0.1.6 in /usr/local/lib/python3.10/dist-packages (from d2l==1.0.3) (0.
Requirement already satisfied: requests==2.31.0 in /usr/local/lib/python3.10/dist-packages (from d2l==1.0.3) (2.31.0)
Requirement already satisfied: pandas==2.0.3 in /usr/local/lib/python3.10/dist-packages (from d2l==1.0.3) (2.0.3)
Requirement already satisfied: scipy==1.10.1 in /usr/local/lib/python3.10/dist-packages (from d2l==1.0.3) (1.10.1)
Requirement already satisfied: notebook in /usr/local/lib/python3.10/dist-packages (from jupyter==1.0.0->d2l==1.0.3) (6.
Requirement already satisfied: qtconsole in /usr/local/lib/python3.10/dist-packages (from jupyter==1.0.0->d2l==1.0.3) (5
Requirement already satisfied: jupyter-console in /usr/local/lib/python3.10/dist-packages (from jupyter==1.0.0->d2l==1.0
Requirement already satisfied: nbconvert in /usr/local/lib/python3.10/dist-packages (from jupyter==1.0.0->d2l==1.0.3) (6
Requirement already satisfied: ipykernel in /usr/local/lib/python3.10/dist-packages (from jupyter==1.0.0->d2l==1.0.3) (5
Requirement already satisfied: ipywidgets in /usr/local/lib/python3.10/dist-packages (from jupyter==1.0.0->d2l==1.0.3) (
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib==3.7.2->d2l=
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-packages (from matplotlib==3.7.2->d2l==1.0
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib==3.7.2->d2l
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib==3.7.2->d2l
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib==3.7.2->d2l==
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib==3.7.2->d2l==1.
Requirement already satisfied: pyparsing<3.1,>=2.3.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib==3.7.2-
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.10/dist-packages (from matplotlib==3.7.2->
Requirement already satisfied: traitlets in /usr/local/lib/python3.10/dist-packages (from matplotlib-inline==0.1.6->d2l=
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas==2.0.3->d2l==1.0.3)
Requirement already satisfied: tzdata>=2022.1 in /usr/local/lib/python3.10/dist-packages (from pandas==2.0.3->d2l==1.0.3
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests==2.31.
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests==2.31.0->d2l==1.0.
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests==2.31.0->d2l
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests==2.31.0->d2l
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.7->matplotli
Requirement already satisfied: ipython-genutils in /usr/local/lib/python3.10/dist-packages (from ipykernel->jupyter==1.0
Requirement already satisfied: ipython>=5.0.0 in /usr/local/lib/python3.10/dist-packages (from ipykernel->jupyter==1.0.0
Requirement already satisfied: jupyter-client in /usr/local/lib/python3.10/dist-packages (from ipykernel->jupyter==1.0.0
Requirement already satisfied: tornado>=4.2 in /usr/local/lib/python3.10/dist-packages (from ipykernel->jupyter==1.0.0->
Requirement already satisfied: widgetsnbextension~=3.6.0 in /usr/local/lib/python3.10/dist-packages (from ipywidgets->ju
Requirement already satisfied: jupyterlab-widgets>=1.0.0 in /usr/local/lib/python3.10/dist-packages (from ipywidgets->ju
Requirement already satisfied: prompt-toolkit!=3.0.0,!=3.0.1,<3.1.0,>=2.0.0 in /usr/local/lib/python3.10/dist-packages (
Requirement already satisfied: pygments in /usr/local/lib/python3.10/dist-packages (from jupyter-console->jupyter==1.0.0
Requirement already satisfied: lxml in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1.0.0->d2l==1.0
Requirement already satisfied: beautifulsoup4 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1.0.0
Requirement already satisfied: bleach in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1.0.0->d2l==1
Requirement already satisfied: defusedxml in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1.0.0->d2
Requirement already satisfied: entrypoints>=0.2.2 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1
Requirement already satisfied: jinja2>=3.0 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1.0.0->d
Requirement already satisfied: jupyter-core>=4.7 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1.
Requirement already satisfied: jupyterlab-pygments in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1.0.
Requirement already satisfied: mistune<2,>=0.8.1 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1.
Requirement already satisfied: nbclient>=0.5.0 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1.0.
Requirement already satisfied: nbformat>=5.1 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1.0.0-
Requirement already satisfied: pandocfilters>=1.4.1 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter=
Requirement already satisfied: tinycss2 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1.0.0->d2l=
Requirement already satisfied: pyzmq<25,>=17 in /usr/local/lib/python3.10/dist-packages (from notebook->jupyter==1.0.0->
Requirement already satisfied: argon2-cffi in /usr/local/lib/python3.10/dist-packages (from notebook->jupyter==1.0.0->d2
Requirement already satisfied: nest-asyncio>=1.5 in /usr/local/lib/python3.10/dist-packages (from notebook->jupyter==1.0
Requirement already satisfied: Send2Trash>=1.8.0 in /usr/local/lib/python3.10/dist-packages (from notebook->jupyter==1.0
Requirement already satisfied: terminado>=0.8.3 in /usr/local/lib/python3.10/dist-packages (from notebook->jupyter==1.0.
Requirement already satisfied: prometheus-client in /usr/local/lib/python3.10/dist-packages (from notebook->jupyter==1.0
```

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

```
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)
```

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz to ../data/FashionMNIS
100%|██████████| 26421880/26421880 [00:01<00:00, 19591011.75it/s]
Extracting ../data/FashionMNIST/raw/train-images-idx3-ubyte.gz to ../data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz to ../data/FashionMNIS
100%|██████████| 29515/29515 [00:00<00:00, 311775.64it/s]
Extracting ../data/FashionMNIST/raw/train-labels-idx1-ubyte.gz to ../data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz to ../data/FashionMNIST
100%|██████████| 4422102/4422102 [00:00<00:00, 5304543.90it/s]
Extracting ../data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to ../data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz to ../data/FashionMNIST
100%|██████████| 5148/5148 [00:00<00:00, 16382607.73it/s]
Extracting ../data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to ../data/FashionMNIST/raw

/usr/local/lib/python3.10/dist-packages/torch/utils/data/dataloader.py:557: UserWarning: This DataLoader will create 4 w
  warnings.warn(_create_warning_msg(
```

8.6. Residual Networks (ResNet) and ResNeXt

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


class Residual(nn.Module):
    """The Residual block of ResNet models."""
    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))
```

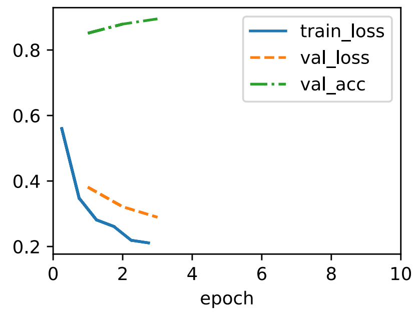
```
@d2l.add_to_class(ResNet)
def block(self, num_residuals, num_channels, first_block=False):
    blk = []
    for i in range(num_residuals):
        if i == 0 and not first_block:
            blk.append(Residual(num_channels, use_1x1conv=True, strides=2))
        else:
            blk.append(Residual(num_channels))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



8.6.7. Exercises

Exercise 1: Comparing Inception Block and Residual Block

In ResNet, the residual block takes the input, applies a series of convolutions, and adds the input back to the output of the convolutions using a shortcut (skip connection). This prevents the network from degrading when more layers are added.

```
import torch
from torch import nn

class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU()
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1)
        self.bn2 = nn.BatchNorm2d(out_channels)

        # Shortcut connection
        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride),
                nn.BatchNorm2d(out_channels)
            )

    def forward(self, x):
        out = self.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(x)
        out = self.relu(out)
        return out

# Test Residual Block
x = torch.rand(1, 3, 224, 224)
block = ResidualBlock(3, 64)
y = block(x)
y.shape
```

```
torch.Size([1, 64, 224, 224])
```

Inception Block Design

The Inception block, on the other hand, applies convolutions with different kernel sizes in parallel (e.g., 1x1, 3x3, 5x5). This block efficiently captures information at multiple scales. Unlike Residual blocks, the Inception block does not use shortcut connections.

```
class InceptionBlock(nn.Module):
    def __init__(self, in_channels):
        super(InceptionBlock, self).__init__()
        self.branch1 = nn.Conv2d(in_channels, 64, kernel_size=1)

        self.branch2 = nn.Sequential(
            nn.Conv2d(in_channels, 48, kernel_size=1),
            nn.Conv2d(48, 64, kernel_size=3, padding=1)
        )

        self.branch3 = nn.Sequential(
            nn.Conv2d(in_channels, 64, kernel_size=1),
            nn.Conv2d(64, 96, kernel_size=5, padding=2)
        )
        self.branch4 = nn.Sequential(
            nn.MaxPool2d(kernel_size=3, stride=1, padding=1),
            nn.Conv2d(in_channels, 32, kernel_size=1)
        )

    def forward(self, x):
        branch1 = self.branch1(x)
        branch2 = self.branch2(x)
        branch3 = self.branch3(x)
        branch4 = self.branch4(x)
        return torch.cat([branch1, branch2, branch3, branch4], dim=1)

# Test Inception Block
x = torch.rand(1, 3, 224, 224)
block = InceptionBlock(3)
y = block(x)
y.shape
```

```
torch.Size([1, 256, 224, 224])
```

Exercise 2: Implementing Variants of ResNet

```
class ResNet(nn.Module):
    def __init__(self, block, num_blocks, num_classes=1000):
        super(ResNet, self).__init__()
        self.in_channels = 64

        self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU()
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

        self.layer1 = self._make_layer(block, 64, num_blocks[0], stride=1)
        self.layer2 = self._make_layer(block, 128, num_blocks[1], stride=2)
        self.layer3 = self._make_layer(block, 256, num_blocks[2], stride=2)
        self.layer4 = self._make_layer(block, 512, num_blocks[3], stride=2)

        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512, num_classes)

    def _make_layer(self, block, out_channels, num_blocks, stride):
        layers = []
        layers.append(block(self.in_channels, out_channels, stride))
        self.in_channels = out_channels
        for _ in range(1, num_blocks):
            layers.append(block(out_channels, out_channels))
        return nn.Sequential(*layers)

    def forward(self, x):
        x = self.relu(self.bn1(self.conv1(x)))
        x = self.maxpool(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.fc(x)
        return x

# Define ResNet-18
def ResNet18():
    return ResNet(ResidualBlock, [2, 2, 2, 2])

# Test ResNet-18
model = ResNet18()
x = torch.rand(1, 3, 224, 224)
y = model(x)
y.shape
```

```
torch.Size([1, 1000])
```

Exercise 3: Implementing the Bottleneck Architecture

```
class Bottleneck(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super(Bottleneck, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.conv3 = nn.Conv2d(out_channels, out_channels * 4, kernel_size=1, stride=1, bias=False)
        self.bn3 = nn.BatchNorm2d(out_channels * 4)

        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels * 4:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels * 4, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(out_channels * 4)
            )

    def forward(self, x):
        out = torch.relu(self.bn1(self.conv1(x)))
        out = torch.relu(self.bn2(self.conv2(out)))
        out = self.bn3(self.conv3(out))
        out += self.shortcut(x)
        out = torch.relu(out)
        return out

# Test Bottleneck Block
x = torch.rand(1, 256, 56, 56)
block = Bottleneck(256, 64)
y = block(x)
y.shape
```

```
torch.Size([1, 256, 56, 56])
```

Exercise 4: Changing the Convolution Order

The original ResNet uses the "Convolution -> BatchNorm -> Activation" order. Later ResNet versions modified this to "BatchNorm -> Activation -> Convolution". Here's the modified structure:

```
class ModifiedResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super(ModifiedResidualBlock, self).__init__()
        self.bn1 = nn.BatchNorm2d(in_channels)
        self.relu = nn.ReLU()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1)

        self.bn2 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1)

        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride),
                nn.BatchNorm2d(out_channels)
            )

    def forward(self, x):
        out = self.relu(self.bn1(x))
        out = self.conv1(out)
        out = self.bn2(out)
        out = self.conv2(out)
        out += self.shortcut(x)
        out = self.relu(out)
        return out

# Test Modified Residual Block
x = torch.rand(1, 64, 56, 56)
block = ModifiedResidualBlock(64, 64)
y = block(x)
y.shape
```

```
import torch
from torch import nn

class ModifiedResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super(ModifiedResidualBlock, self).__init__()
        self.bn1 = nn.BatchNorm2d(in_channels)
        self.relu = nn.ReLU()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1)

        self.bn2 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1)

        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride),
                nn.BatchNorm2d(out_channels)
            )

    def forward(self, x):
        out = self.relu(self.bn1(x))
        out = self.conv1(out)
        out = self.bn2(out)
        out = self.conv2(out)
        out += self.shortcut(x)
        out = self.relu(out)
        return out

# Test Modified Residual Block
x = torch.rand(1, 64, 56, 56)
block = ModifiedResidualBlock(64, 64)
y = block(x)
y.shape
```

```
torch.Size([1, 64, 56, 56])
```

Discussion

The Inception and Residual blocks represent different approaches to neural network design. The Inception block captures multi-scale features through parallel convolutions, making it computationally heavier but potentially more expressive. In contrast, the Residual block relies on skip connections to mitigate vanishing gradients, which allows for very deep networks without degradation in performance.

When comparing accuracy and computational cost, Residual blocks are typically more efficient for extremely deep architectures, as evidenced by the success of ResNet. The bottleneck architecture further improves efficiency by reducing the number of channels during computation. The order of convolution, batch normalization, and activation functions affects model performance and convergence, with newer architectures favoring "batch normalization -> activation -> convolution" as seen in later ResNet versions.

Lastly, increasing complexity without bounds can lead to overfitting and computational inefficiency. Models need to balance expressiveness with generalizability, ensuring they perform well on both training and unseen data.