COSE474-2024F: Deep Learning

0.1. Installation

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

```
Collecting d2l==1.0.3
  Downloading d2l-1.0.3-py3-none-any.whl.metadata (556 bytes)
Collecting jupyter==1.0.0 (from d2l==1.0.3)
  Downloading jupyter-1.0.0-py2.py3-none-any.whl.metadata (995 bytes)
Collecting numpy==1.23.5 (from d2l==1.0.3)
  Downloading numpy-1.23.5-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_
Collecting matplotlib==3.7.2 (from d2l==1.0.3)
  Downloading matplotlib-3.7.2-cp310-cp310-manylinux_2_17_x86_64.manylinux2
Collecting matplotlib-inline==0.1.6 (from d2l==1.0.3)
  Downloading matplotlib_inline-0.1.6-py3-none-any.whl.metadata (2.8 kB)
Collecting requests==2.31.0 (from d2l==1.0.3)
  Downloading requests-2.31.0-py3-none-any.whl.metadata (4.6 kB)
Collecting pandas==2.0.3 (from d2l==1.0.3)
  Downloading pandas-2.0.3-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_
Collecting scipy==1.10.1 (from d2l==1.0.3)
  Downloading scipy-1.10.1-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_
                                        58.9/58.9 kB 3.6 MB/s eta 0:0
Requirement already satisfied: notebook in /usr/local/lib/python3.10/dist-p
Collecting qtconsole (from jupyter==1.0.0->d2l==1.0.3)
  Downloading qtconsole-5.6.0-py3-none-any.whl.metadata (5.0 kB)
Requirement already satisfied: jupyter-console in /usr/local/lib/python3.10
Requirement already satisfied: nbconvert in /usr/local/lib/python3.10/dist-
Requirement already satisfied: ipykernel in /usr/local/lib/python3.10/dist-
Requirement already satisfied: ipywidgets in /usr/local/lib/python3.10/dist
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.1
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/di
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.10/d
Collecting pyparsing<3.1,>=2.3.1 (from matplotlib==3.7.2->d2l==1.0.3)
  Downloading pyparsing-3.0.9-py3-none-any.whl.metadata (4.2 kB)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/pytho
Requirement already satisfied: traitlets in /usr/local/lib/python3.10/dist-
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/di
Requirement already satisfied: tzdata>=2022.1 in /usr/local/lib/python3.10/
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/p
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/di
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-p
Requirement already satisfied: ipython-genutils in /usr/local/lib/python3.1
Requirement already satisfied: ipython>=5.0.0 in /usr/local/lib/python3.10/
Requirement already satisfied: jupyter-client in /usr/local/lib/python3.10/
Requirement already satisfied: tornado>=4.2 in /usr/local/lib/python3.10/di
Requirement already satisfied: widgetsnbextension~=3.6.0 in /usr/local/lib/
Requirement already satisfied: jupyterlab-widgets>=1.0.0 in /usr/local/lib/
Requirement already satisfied: prompt-toolkit!=3.0.0,!=3.0.1,<3.1.0,>=2.0.0
Requirement already satisfied: pygments in /usr/local/lib/python3.10/dist-p
Requirement already satisfied: lxml in /usr/local/lib/python3.10/dist-packa
Requirement already satisfied: beautifulsoup4 in /usr/local/lib/python3.10/
Requirement already satisfied: bleach in /usr/local/lib/python3.10/dist-pac
Requirement already satisfied: defusedxml in /usr/local/lib/python3.10/dist
Requirement already satisfied: entrypoints>=0.2.2 in /usr/local/lib/python3
Requirement already satisfied: jinja2>=3.0 in /usr/local/lib/python3.10/dis
Requirement already satisfied: jupyter-core>=4.7 in /usr/local/lib/python3.
Requirement already satisfied: jupyterlab-pygments in /usr/local/lib/python
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10
Requirement already satisfied: mistune<2,>=0.8.1 in /usr/local/lib/python3.
Requirement already satisfied: nbclient>=0.5.0 in /usr/local/lib/python3.10
Requirement already satisfied: nbformat>=5.1 in /usr/local/lib/python3.10/d
Requirement already satisfied: pandocfilters>=1.4.1 in /usr/local/lib/pytho
Requirement already satisfied: tinycss2 in /usr/local/lib/python3.10/dist-p
Requirement already satisfied: pyzmq<25,>=17 in /usr/local/lib/python3.10/d
Requirement already satisfied: argon2-cffi in /usr/local/lib/python3.10/dis
Requirement already satisfied: nest-asyncio>=1.5 in /usr/local/lib/python3.
Requirement already satisfied: Send2Trash>=1.8.0 in /usr/local/lib/python3.
Requirement already satisfied: terminado>=0.8.3 in /usr/local/lib/python3.1
Requirement already satisfied: prometheus-client in /usr/local/lib/python3.
Requirement already satisfied: nbclassic>=0.4.7 in /usr/local/lib/python3.1
Collecting qtpy>=2.4.0 (from qtconsole->jupyter==1.0.0->d2l==1.0.3)
  Downloading QtPy-2.4.1-py3-none-any.whl.metadata (12 kB)
Requirement already satisfied: setuptools>=18.5 in /usr/local/lib/python3.1
Collecting jedi>=0.16 (from ipython>=5.0.0->ipykernel->jupyter==1.0.0->d2l=
  Using cached jedi-0.19.1-py2.py3-none-any.whl.metadata (22 kB)
Requirement already satisfied: decorator in /usr/local/lib/python3.10/dist-
Requirement already satisfied: pickleshare in /usr/local/lib/python3.10/dis
Requirement already satisfied: backcall in /usr/local/lib/python3.10/dist-p
Requirement already satisfied: pexpect>4.3 in /usr/local/lib/python3.10/dis
Requirement already satisfied: platformdirs>=2.5 in /usr/local/lib/python3.
Requirement already satisfied: notebook-shim>=0.2.3 in /usr/local/lib/pytho
Requirement already satisfied: fastjsonschema>=2.15 in /usr/local/lib/pytho
Requirement already satisfied: jsonschema>=2.6 in /usr/local/lib/python3.10
Requirement already satisfied: wcwidth in /usr/local/lib/python3.10/dist-pa
Requirement already satisfied: ptyprocess in /usr/local/lib/python3.10/dist
Requirement already satisfied: argon2-cffi-bindings in /usr/local/lib/pytho
Requirement already satisfied: soupsieve>1.2 in /usr/local/lib/python3.10/d
Requirement already satisfied: webencodings in /usr/local/lib/python3.10/di
Requirement already satisfied: parso<0.9.0,>=0.8.3 in /usr/local/lib/python
Requirement already satisfied: attrs>=22.2.0 in /usr/local/lib/python3.10/d
Requirement already satisfied: jsonschema-specifications>=2023.03.6 in /usr
Requirement already satisfied: referencing>=0.28.4 in /usr/local/lib/python
Requirement already satisfied: rpds-py>=0.7.1 in /usr/local/lib/python3.10/
Requirement already satisfied: jupyter-server<3,>=1.8 in /usr/local/lib/pyt
Requirement already satisfied: cffi>=1.0.1 in /usr/local/lib/python3.10/dis
Requirement already satisfied: pycparser in /usr/local/lib/python3.10/dist-
Requirement already satisfied: anyio<4,>=3.1.0 in /usr/local/lib/python3.10
Requirement already satisfied: websocket-client in /usr/local/lib/python3.1
Requirement already satisfied: sniffio>=1.1 in /usr/local/lib/python3.10/di
Requirement already satisfied: exceptiongroup in /usr/local/lib/python3.10/
Downloading d2l-1.0.3-py3-none-any.whl (111 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 111.7/111.7 kB 8.1 MB/s eta 0:0
Downloading jupyter-1.0.0-py2.py3-none-any.whl (2.7 kB)
Downloading matplotlib-3.7.2-cp310-cp310-manylinux_2_17_x86_64.manylinux201
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 11.6/11.6 MB 72.6 MB/s eta 0:00
Downloading matplotlib_inline-0.1.6-py3-none-any.whl (9.4 kB)
Downloading numpy-1.23.5-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x8
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 17.1/17.1 MB 72.8 MB/s eta 0:00
Downloading pandas-2.0.3-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x8
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 12.3/12.3 MB 36.3 MB/s eta 0:00
Downloading requests-2.31.0-py3-none-any.whl (62 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 62.6/62.6 kB 5.8 MB/s eta 0:00:
Downloading scipy-1.10.1-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x8
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 34.4/34.4 MB 22.4 MB/s eta 0:00
Downloading pyparsing-3.0.9-py3-none-any.whl (98 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 98.3/98.3 kB 8.7 MB/s eta 0:00:
Downloading qtconsole-5.6.0-py3-none-any.whl (124 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 124.7/124.7 kB 11.1 MB/s eta 0:
Downloading QtPy-2.4.1-py3-none-any.whl (93 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 93.5/93.5 kB 8.5 MB/s eta 0:00:
Using cached jedi-0.19.1-py2.py3-none-any.whl (1.6 MB)
Installing collected packages: requests, qtpy, pyparsing, numpy, matplotlib
  Attempting uninstall: requests
    Found existing installation: requests 2.32.3
    Uninstalling requests-2.32.3:
      Successfully uninstalled requests-2.32.3
  Attempting uninstall: pyparsing
    Found existing installation: pyparsing 3.1.4
    Uninstalling pyparsing-3.1.4:
      Successfully uninstalled pyparsing-3.1.4
  Attempting uninstall: numpy
    Found existing installation: numpy 1.26.4
    Uninstalling numpy-1.26.4:
      Successfully uninstalled numpy-1.26.4
  Attempting uninstall: matplotlib-inline
    Found existing installation: matplotlib-inline 0.1.7
    Uninstalling matplotlib-inline-0.1.7:
      Successfully uninstalled matplotlib-inline-0.1.7
  Attempting uninstall: scipy
    Found existing installation: scipy 1.13.1
    Uninstalling scipy-1.13.1:
      Successfully uninstalled scipy-1.13.1
  Attempting uninstall: pandas
    Found existing installation: pandas 2.2.2
    Uninstalling pandas-2.2.2:
      Successfully uninstalled pandas-2.2.2
  Attempting uninstall: matplotlib
    Found existing installation: matplotlib 3.7.1
    Uninstalling matplotlib-3.7.1:
      Successfully uninstalled matplotlib-3.7.1
ERROR: pip's dependency resolver does not currently take into account all t
albucore 0.0.16 requires numpy>=1.24, but you have numpy 1.23.5 which is in
albumentations 1.4.15 requires numpy>=1.24.4, but you have numpy 1.23.5 whi
bigframes 1.21.0 requires numpy>=1.24.0, but you have numpy 1.23.5 which is
chex 0.1.87 requires numpy>=1.24.1, but you have numpy 1.23.5 which is inco
google-colab 1.0.0 requires pandas==2.2.2, but you have pandas 2.0.3 which
google-colab 1.0.0 requires requests==2.32.3, but you have requests 2.31.0
jax 0.4.33 requires numpy>=1.24, but you have numpy 1.23.5 which is incompa
jaxlib 0.4.33 requires numpy>=1.24, but you have numpy 1.23.5 which is inco
mizani 0.11.4 requires pandas>=2.1.0, but you have pandas 2.0.3 which is in
plotnine 0.13.6 requires pandas<3.0.0,>=2.1.0, but you have pandas 2.0.3 wh
xarray 2024.9.0 requires numpy>=1.24, but you have numpy 1.23.5 which is in
xarray 2024.9.0 requires pandas>=2.1, but you have pandas 2.0.3 which is in
Successfully installed d2l-1.0.3 jedi-0.19.1 jupyter-1.0.0 matplotlib-3.7.2
```

7.1. From Fully Connected Layers to Convolutions

The passage discusses the challenges of using traditional machine learning models, such as multi-layer perceptrons (MLPs), for high-dimensional data like images. While MLPs work well for tabular data, they become impractical for large datasets such as one-megapixel images due to the vast number of parameters required (up to billions). This makes training difficult without substantial computational resources. However, convolutional neural networks (CNNs) offer a solution by leveraging the inherent structure in image data, making it possible to efficiently train models for tasks like distinguishing cats from dogs.

7.1.1. Invariance

The passage discusses the importance of detecting objects in images, regardless of their exact location. This spatial invariance is key in object recognition tasks, where an object like a pig should be recognizable whether it's at the top or bottom of an image. Drawing from examples like "Where's Waldo," the passage illustrates that identifying an object should not depend on its position but on its features. Convolutional neural networks (CNNs) leverage this principle of *spatial invariance* by using fewer parameters to detect objects across various regions. It outlines three guiding principles for designing neural networks for computer vision:

1. **Translation Invariance:** The network should recognize the same patch anywhere in the image.
2. **Locality:** Early layers should focus on local image regions, without considering distant regions.
3. **Long-range Feature Capture:** Deeper layers should capture broader features as the network progresses.

These principles lead to more efficient and effective object detection models.

7.1.2. Constraining the MLP

The challenge of using fully connected layers in a multi-layer perceptron (MLP) to process images while preserving spatial structure. It explains how both the input image and hidden representations can be treated as two-dimensional matrices. To connect each pixel in the input image to the hidden units, the model uses fourth-order weight tensors, denoted by W and V, which assign weights to patches of pixels around each position.

The equation provided shows how each hidden unit's value is computed by summing over weighted contributions from surrounding pixels in the input image. However, this approach would require an enormous number of parameters (e.g., 10^{12} for a single layer mapping a 1-megapixel image), which is computationally infeasible with current technology. This highlights the limitations of fully connected layers for high-dimensional data and motivates more efficient approaches like convolutional layers in CNNs.

The convolutional neural networks (CNNs) use the principle of *translation invariance* to drastically reduce the number of parameters required for image processing. Initially, using a fully connected layer to process an image would demand an infeasible number of parameters, but by simplifying the structure of the model, the number of parameters can be reduced.

By assuming that the convolutional weights (denoted $\mathbf{V}$) are independent of the location in the image, the model becomes translation invariant, where a shift in the input image results in a shift in the hidden representation. This results in the introduction of *convolutions*—a process that sums the weighted contributions of nearby pixels. This reduces the parameters from 10^{12} to a more manageable 4×10^6.

The second principle, *locality*, further reduces parameters by limiting the range of pixels considered for each hidden unit. This leads to the formulation of a *convolutional layer*, where weights (filters or kernels) are only applied to a small, local region around each pixel. This cuts down the parameters by another magnitude, from millions to just a few hundred, without sacrificing the model's ability to capture relevant information.

However, this reduction in parameters comes with trade-offs: the model assumes that features are translation invariant and only uses local information. When these assumptions align with the real-world structure of images, the model becomes efficient and generalizes well. Deeper layers in CNNs, with nonlinearities and convolutions, can then capture more complex, large-scale features of the image, forming the foundation of modern computer vision architectures.

7.1.3. Convolutions

The passage explains why the operation in equation :eq: `eq_conv-layer` is called a convolution by connecting it to the mathematical definition of convolution. In mathematics, convolution measures the overlap between two functions, f and g, where one function is shifted and flipped before being integrated or summed over. The definition for continuous functions involves an integral, while for discrete functions, it becomes a sum.

For one-dimensional discrete functions, convolution is represented as:

$$(f * g)(i) = \sum_a f(a)g(i-a)$$

For two-dimensional tensors, the convolution becomes:

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

This closely resembles the form of $: eq : `eq_conv - layer`$, with the main difference being that in convolution, we use $(i-a, j-b)$, while in :eq: `eq_conv-layer`, we use $(i+a, j+b)$. Despite this difference, the operations are similar and can be aligned with slight changes in notation. The operation described in :eq: `eq_conv-layer` is technically a *cross-correlation*, but in practice, both are commonly referred to as convolutions in the context of deep learning.

7.1.4. Channels

This section expands on convolutional layers by introducing the concept of *channels* in images, which account for the red, green, and blue color layers in an image. Images are not simple two-dimensional objects but rather three-dimensional tensors with height, width, and channels (e.g., shape (1024 \times 1024 \times 3)).

To handle this, convolutional filters must also accommodate these channels. Instead of a two-dimensional filter $[\mathbf{V}]_{a,b}$, we now have $[\mathsf{V}]_{a,b,c}$, where c indexes the input channels (e.g., RGB layers).

In addition, hidden representations are also treated as third-order tensors, with multiple *channels* or *feature maps* that represent different learned features at each spatial location. For example, some channels may specialize in detecting edges, while others might focus on textures.

To support multiple channels in both the input image X and hidden layers H, the convolutional filter becomes a four-dimensional tensor $[\mathsf{V}]_{a,b,c,d}$, where d indexes the output channels in the hidden representation. The equation for the convolutional layer becomes:

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

This general form allows convolutional layers to process multi-channel inputs (such as colored images) and produce multi-channel hidden representations. The section also introduces the need to address efficient computation, combining layers, and using proper activation functions, all of which are discussed later in the chapter.

7.1.5. Summary and Discussion

In this section, the structure of convolutional neural networks (CNNs) is derived from first principles, emphasizing that CNNs are a logical choice for image processing based on reasonable assumptions. Two key principles are highlighted: **translation invariance**, meaning that all patches of an image are processed in the same way, and **locality**, which means that only local pixel neighborhoods influence hidden representations. These principles help reduce the computational complexity of image processing tasks, turning otherwise infeasible problems into manageable ones.

The introduction of channels, such as red, green, and blue, helps to regain some complexity lost by the restrictions imposed by locality and translation invariance. This can be extended to other types of data, like hyperspectral images in satellite imaging, which can have many more channels beyond just RGB. The section concludes by previewing methods to manipulate image dimensionality, shift from location-based to channel-based representations, and efficiently handle tasks involving large numbers of categories.

7.1.6. Exercises

1. Assume that the size of the convolution kernel is $\Delta = 0$. Show that in this case the convolution kernel implements an MLP independently for each set of channels. This leads to the Network in Network architectures :cite: `Lin.Chen.Yan.2013`.
2. Audio data is often represented as a one-dimensional sequence.
    1. When might you want to impose locality and translation invariance for audio?
    2. Derive the convolution operations for audio.
    3. Can you treat audio using the same tools as computer vision? Hint: use the spectrogram.
3. Why might translation invariance not be a good idea after all? Give an example.
4. Do you think that convolutional layers might also be applicable for text data? Which problems might you encounter with language?
5. What happens with convolutions when an object is at the boundary of an image?
6. Prove that the convolution is symmetric, i.e., $f * g = g * f$.

7.2. Convolutions for Images

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

7.2.1. The Cross-Correlation Operation

The cross-correlation operation is central to convolutional layers, and while it is often referred to as "convolution," it is more accurately described as cross-correlation. Here's a breakdown of the process with an example:

Cross-Correlation Example:

1. **Input Tensor:**
   - A two-dimensional tensor with dimensions 3×3, where the values are arranged in a grid:

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

2. **Kernel (Convolution Window):**
   - A 2×2 kernel (or filter) with weights as follows:

$$\begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}$$

3. **Sliding the Kernel:** The kernel slides over the input tensor. At each position, the elementwise product of the kernel and the corresponding part of the input tensor is summed to produce the output value. Here's how it works step-by-step:
   - For the first position (upper-left corner of the input):
$$\begin{bmatrix} 0 & 1 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix} = 0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$$
   - For the second position (shifted right by one):
$$\begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix} \times \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix} = 1 \times 0 + 2 \times 1 + 4 \times 2 + 5 \times 3 = 25$$
   - For the third position (shifted down by one):

$$\begin{bmatrix} 3 & 4 \\ 6 & 7 \end{bmatrix} \times \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix} = 3 \times 0 + 4 \times 1 + 6 \times 2 + 7 \times 3 = 37$$

- For the fourth position (shifted right and down):

$$\begin{bmatrix} 4 & 5 \\ 7 & 8 \end{bmatrix} \times \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix} = 4 \times 0 + 5 \times 1 + 7 \times 2 + 8 \times 3 = 43$$

Output Tensor:

The result of this cross-correlation operation is a 2×2 output tensor:

$$\begin{bmatrix} 19 & 25 \\ 37 & 43 \end{bmatrix}$$

Key Observations:

- The output tensor is smaller than the input tensor because the kernel needs enough space to fit within the input as it slides. The output size is determined by the formula:
$$(n_\text{h} - k_\text{h} + 1) \times (n_\text{w} - k_\text{w} + 1)$$
where n_h and n_w are the height and width of the input tensor, and k_h and k_w are the height and width of the kernel.

Next, to maintain the same size for the output as the input, we can apply padding, which we'll explore further. This keeps the output the same size as the input by adding zeros around the edges of the input tensor.

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

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

```
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 Layers

```
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
```

In $h \times w$ convolution or an $h \times w$ convolution kernel, the height and width of the convolution kernel are h and w, respectively. We also refer to a convolutional layer with an $h \times w$ convolution kernel simply as an $h \times w$ convolutional layer.

7.2.3. Object Edge Detection in Images

```
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.]])
```

7.2.4. Learning a Kernel

To learn a kernel that generates output Y from input X using a convolutional layer:

1. **Initialize**: Create a convolutional layer with a randomly initialized kernel.
2. **Loss Function**: Define a squared error loss to compare the output of the convolutional layer with the target output Y.
3. **Backpropagation**: Calculate the gradient of the loss with respect to the kernel.
4. **Update Kernel**: Adjust the kernel using gradient descent based on the calculated gradients.
5. **Training**: Repeat the process for multiple iterations to refine the kernel.

The goal is to approximate the desired edge-detecting behavior, similar to the original finite difference filter $K = [1, -1]$.

```
# Construct a two-dimensional convolutional layer with 1 output channel and a
# kernel of shape (1, 2). For the sake of simplicity, we ignore the bias here
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)
    l = (Y_hat - Y) ** 2
    conv2d.zero_grad()
    l.sum().backward()
    # Update the kernel
    conv2d.weight.data[:] -= lr * conv2d.weight.grad
    if (i + 1) % 2 == 0:
        print(f'epoch {i + 1}, loss {l.sum():.3f}')
```

```
epoch 2, loss 3.305
epoch 4, loss 0.899
epoch 6, loss 0.292
epoch 8, loss 0.107
epoch 10, loss 0.042
```

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

```
tensor([[ 1.0135, -0.9722]])
```

7.2.5. Cross-Correlation and Convolution

- **Convolution vs. Cross-Correlation**: Convolutional layers can perform strict convolution (where the kernel is flipped) or cross-correlation (without flipping). However, the output remains the same regardless of which operation is performed.
- **Kernel Transformation**: If a layer learns a kernel $\mathbf{K}$ using cross-correlation, the corresponding kernel for strict convolution $\mathbf{K}'$ will be $\mathbf{K}$ flipped both horizontally and vertically.
- **Output Consistency**: The output obtained from the strict convolution using $\mathbf{K}'$ will match the output obtained from cross-correlation with $\mathbf{K}$.
- **Terminology**: In deep learning literature, the term "convolution" is commonly used to refer to the cross-correlation operation, despite the technical differences.
- **Element Definition**: An "element" refers to an entry or component of any tensor that represents a layer's representation or a convolution kernel.

7.2.6. Feature Map and Receptive Field

- **Feature Map**: The output of a convolutional layer is referred to as a *feature map*, representing learned spatial features (width and height) for the subsequent layer.
- **Receptive Field**: The *receptive field* of an element (x) in a layer includes all elements from previous layers that influence the calculation of (x) during forward propagation. The receptive field can be larger than the input size.
- **Example of Receptive Field**: In a CNN with a (2 \times 2) convolution kernel, the receptive field of an output element includes all input elements that affect its value. For instance, if a deeper layer processes the output of an earlier layer, the receptive field expands to include more elements from the original input.
- **Neurophysiology Connection**: The concept of receptive fields is inspired by experiments on the visual cortex of animals, which demonstrated how lower levels respond to edges and shapes, paralleling the function of convolutional kernels.
- **Biological Relevance**: The relationship between biological vision systems and the features computed by deep networks supports the effectiveness of convolutions in computer vision tasks, as evidenced by studies.
- **Impact on Deep Learning**: Convolutions have significantly contributed to the success of deep learning in computer vision, drawing connections between biological processes and computational methods.

7.2.7. Summary

- **Core Computation**: The fundamental operation in a convolutional layer is the cross-correlation, which can be efficiently computed using a simple nested for-loop.
- **Matrix-Matrix Operations**: With multiple input and output channels, convolutions involve matrix-matrix operations, which enhance computational efficiency and locality.
- **Hardware Optimization**: The local nature of convolutions allows for significant hardware optimizations, enabling chip designers to focus on fast computation rather than memory, making advanced computer vision techniques more accessible.
- **Versatility of Convolutions**: Convolutions can be utilized for various tasks, including edge detection, image blurring, and sharpening.
- **Learning Filters**: Instead of manually designing filters, we can learn them from data, transitioning from feature engineering to evidence-based statistical learning.
- **Biological Relevance**: The filters learned in convolutional networks correspond to receptive fields and feature maps observed in biological systems, providing confidence in their effectiveness and relevance.

7.2.8. Exercises

1. Construct an image `X` with diagonal edges.
    1. What happens if you apply the kernel `K` in this section to it?
    2. What happens if you transpose `X`?
    3. What happens if you transpose `K`?
2. Design some kernels manually.
    1. Given a directional vector $\mathbf{v} = (v_1, v_2)$, derive an edge-detection kernel that detects edges orthogonal to $\mathbf{v}$, i.e., edges in the direction $(v_2, -v_1)$.
    2. Derive a finite difference operator for the second derivative. What is the minimum size of the convolutional kernel associated with it? Which structures in images respond most strongly to it?
    3. How would you design a blur kernel? Why might you want to use such a kernel?
    4. What is the minimum size of a kernel to obtain a derivative of order d?
3. When you try to automatically find the gradient for the `Conv2D` class we created, what kind of error message do you see?
4. How do you represent a cross-correlation operation as a matrix multiplication by changing the input and kernel tensors?

7.3. Padding and Stride

```
import torch
from torch import nn
```

7.3.1. Padding

- **Pixel Utilization Issue**: Convolutional layers often lead to the loss of pixels, especially around the perimeter of images. This is particularly evident in the corners, where pixels are minimally utilized.
- **Padding Solution**: To mitigate the loss of pixels, we can add extra rows and columns of pixels (padding) around the input image, typically setting these extra pixels to zero.
- **Output Shape Calculation**: When padding is applied, the output shape of the convolution is given by:

  $$(n_\mathrm{h} - k_\mathrm{h} + p_\mathrm{h} + 1) \times (n_\mathrm{w} - k_\mathrm{w} + p_\mathrm{w} + 1),$$

  where p_h and p_w are the padding values for height and width, respectively.
- **Preserving Dimensions**: By setting $p_\mathrm{h} = k_\mathrm{h} - 1$ and $p_\mathrm{w} = k_\mathrm{w} - 1$, the input and output dimensions can be kept the same, making it easier to predict output shapes when constructing networks.
- **Choice of Kernel Sizes**: Convolutional Neural Networks (CNNs) often use kernels with odd sizes (1, 3, 5, or 7). This choice facilitates dimensionality preservation while allowing equal padding on all sides.
- **Clerical Benefit**: Using odd kernel sizes with symmetrical padding results in the output dimensions matching the input dimensions. For any 2D tensor X, the output $Y[i, j]$ is computed by centering the convolution window on $X[i, j]$, thus maintaining spatial consistency.
- **Example Implementation**: A two-dimensional convolutional layer with a 3×3 kernel and 1 pixel of padding results in an output that maintains the same height and width as the 8×8 input.

```
# We define a helper function to calculate convolutions. It initializes the
# convolutional layer weights and performs corresponding dimensionality
# elevations and reductions on the input and output
def comp_conv2d(conv2d, X):
    # (1, 1) indicates that batch size and the number of channels are both 1
    X = X.reshape((1, 1) + X.shape)
    Y = conv2d(X)
    # Strip the first two dimensions: examples and channels
    return Y.reshape(Y.shape[2:])

# 1 row and column is padded on either side, so a total of 2 rows or columns
# are added
conv2d = nn.LazyConv2d(1, kernel_size=3, padding=1)
X = torch.rand(size=(8, 8))
comp_conv2d(conv2d, X).shape
```

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

```
# We use a convolution kernel with height 5 and width 3. The padding on either
# side of the height and width are 2 and 1, respectively
conv2d = nn.LazyConv2d(1, kernel_size=(5, 3), padding=(2, 1))
comp_conv2d(conv2d, X).shape
```

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

7.3.2. Stride

- **Cross-Correlation Process**: The convolution window starts at the upper-left corner of the input tensor and slides over the entire input, both down and to the right.
- **Strides**: The term *stride* refers to the number of rows and columns the convolution window moves per slide. While the default stride is 1, larger strides can be used for computational efficiency or to downsample the input.
- **Example of Stride Usage**: In an example with a stride of 3 vertically and 2 horizontally:
    - The window skips 3 rows when moving down and 2 columns when moving to the right.
    - If the window cannot fit the input element (without additional padding), no output is generated for that position.
- **Output Shape Calculation**: The output shape with specified strides is given by:

$$\lfloor(n_\textrm{h}-k_\textrm{h}+p_\textrm{h}+s_\textrm{h})/s_\textrm{h}\rfloor \times \lfloor(n_\textrm{w}-k_\textrm{w}+p_\textrm{w}+s_\textrm{w})/s_\textrm{w}\rfloor.$$

- **Simplified Output Shape**: By setting $p_\textrm{h}=k_\textrm{h}-1$ and $p_\textrm{w}=k_\textrm{w}-1$, the output shape can be simplified to:

$$\lfloor(n_\textrm{h}+s_\textrm{h}-1)/s_\textrm{h}\rfloor \times \lfloor(n_\textrm{w}+s_\textrm{w}-1)/s_\textrm{w}\rfloor.$$

- **Divisibility Condition**: If the input height and width are divisible by their respective strides, the output shape will be:

$$(n_\textrm{h}/s_\textrm{h}) \times (n_\textrm{w}/s_\textrm{w}).$$

- **Example Implementation**: When both height and width strides are set to 2, the result is a halved output height and width compared to the input.

```
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])
```

7.3.3. Summary and Discussion

- **Purpose of Padding**:
    - Padding is used to maintain the same height and width of the output as the input, preventing undesirable shrinkage.
    - It ensures that all pixels are utilized equally frequently.
- **Symmetric Padding**:
    - Padding is typically symmetric on both sides, denoted as $(p_\mathrm{h}, p_\mathrm{w})$.
    - Often, $p_\mathrm{h} = p_\mathrm{w} = p$ is used for simplicity.
- **Stride Convention**:
    - When horizontal and vertical strides match $(s_\mathrm{h} = s_\mathrm{w})$, it is referred to as stride s.
    - Strides can reduce the resolution of the output, resulting in dimensions of $1/n$ of the input for $n > 1$.
- **Default Values**:
    - The default padding is 0, and the default stride is 1.
- **Zero-Padding Benefits**:
    - Zero-padding is computationally efficient and can be implemented without allocating additional memory.
    - It allows CNNs to implicitly encode position information within an image by learning the locations of "whitespace."
- **Alternatives to Zero-Padding**:
    - While zero-padding is common, there are many alternatives. A review by Alsallakh et al. (2020) provides an overview of these options, though it lacks clear guidelines on when to use nonzero paddings unless artifacts arise.

7.3.4. Exercises

1. Given the final code example in this section with kernel size $(3, 5)$, padding $(0, 1)$, and stride $(3, 4)$, calculate the output shape to check if it is consistent with the experimental result.
2. For audio signals, what does a stride of 2 correspond to?
3. Implement mirror padding, i.e., padding where the border values are simply mirrored to extend tensors.
4. What are the computational benefits of a stride larger than 1?
5. What might be statistical benefits of a stride larger than 1?
6. How would you implement a stride of $\frac{1}{2}$? What does it correspond to? When would this be useful?

7.4. Multiple Input and Multiple Output Channels

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

7.4.1. Multiple Input Channels

- **Convolution Kernel Structure**:
  - The convolution kernel must have the same number of input channels as the input data. If the number of input channels is c_i, the kernel will also have c_i channels.
- **Kernel Shape**:
  - For a kernel window of size $k_\mathrm{h} \times k_\mathrm{w}$:
    - If $c_\mathrm{i} = 1$, the kernel is a 2D tensor of shape $k_\mathrm{h} \times k_\mathrm{w}$.
    - If $c_\mathrm{i} > 1$, the kernel has a shape of $c_\mathrm{i} \times k_\mathrm{h} \times k_\mathrm{w}$, containing a separate $k_\mathrm{h} \times k_\mathrm{w}$ tensor for each input channel.
- **Cross-Correlation Operation**:
  - The cross-correlation is performed for each input channel individually.
  - The results from each channel are summed together to produce a 2D tensor output.
- **Example Illustration**:
  - An example with two input channels shows the calculation of the first output element by summing the products of corresponding elements from the input and kernel tensors.
- **Implementation Insight**:
  - Implementing the cross-correlation for multiple input channels involves performing the operation per channel and aggregating the results, reinforcing understanding of the process.

```
def corr2d_multi_in(X, K):
    # Iterate through the 0th dimension (channel) of K first, then add them up
    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.]])
```

7.4.2. Multiple Output Channels

- **Need for Multiple Output Channels**:
  - In deep learning architectures, it is common to increase the number of output channels as we progress deeper into the network. This is done while typically downsampling the spatial resolution.
- **Channel Representation**:
  - Each output channel is thought to respond to different features, but channels are not learned independently. Instead, they are optimized to be collectively useful, meaning that certain channels might jointly represent features like edges or other patterns.
- **Kernel Tensor Dimensions**:
  - Let c_i be the number of input channels and c_o be the number of output channels. The kernel tensor for each output channel is structured as:
    - Shape: $c_\mathrm{o} \times c_\mathrm{i} \times k_\mathrm{h} \times k_\mathrm{w}$.
    - This structure allows each output channel's convolution to be influenced by all input channels.
- **Cross-Correlation for Multiple Channels**:
  - The output for each output channel is computed using the corresponding kernel tensor and inputs from all input channels, allowing for complex feature extraction.
- **Implementation**:
  - A specific implementation for calculating the output from multiple channels is provided, reinforcing the understanding of how multiple output channels are managed in convolutional operations.

```
def corr2d_multi_in_out(X, K):
    # Iterate through the 0th dimension of K, and each time, perform
    # cross-correlation operations with input X. All of the results are
    # stacked together
    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.]]])
```

7.4.3. 1×1 Convolutional Layer

Understanding 1×1 Convolutions

- **Definition and Purpose**:
  - A 1×1 convolution (where $k_h = k_w = 1$) might initially seem counterintuitive since it does not consider neighboring pixels, unlike larger convolutional kernels. However, they are widely used in complex deep learning architectures.
- **Operational Mechanics**:
  - The 1×1 convolution performs operations solely on the channel dimension while keeping the spatial dimensions (height and width) unchanged.
  - Each output pixel is computed as a linear combination of the input pixel values at the same position across all input channels.
- **Equivalent to Fully Connected Layer**:
  - You can conceptualize the 1×1 convolutional layer as applying a fully connected layer to each pixel position. This means it transforms c_i input channels into c_o output channels while maintaining the same spatial dimensions.
  - The weight sharing across the pixel locations is a crucial aspect of this convolution, leading to a total weight count of $c_o \times c_i$ (plus biases).
- **Post-Convolution Nonlinearities**:
  - Typically, nonlinear activation functions follow the 1×1 convolution. This means that the results cannot be directly integrated into other convolutions, preserving the distinctiveness of this layer type.
- **Implementation**:
  - To implement a 1×1 convolution using a fully connected layer, you may need to reshape the input and output data. This involves adjusting the dimensions appropriately before and after the matrix multiplication to align with the structure required for the convolution.

```
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))
    # Matrix multiplication in the fully connected layer
    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
```

7.4.4. Discussion

Summary of Channels in CNNs

- **Purpose of Channels:**
  - Combine benefits of Multi-Layer Perceptrons (MLPs) and convolutions.
  - Enable handling of significant nonlinearities and localized feature analysis.
  - Allow simultaneous reasoning with multiple features (e.g., edge and shape detectors).
- **Advantages:**
  - Provide a trade-off between parameter reduction (due to translation invariance and locality) and the need for expressive, diverse models in computer vision.
- **Cost Considerations:**
  - Computational cost for a $k \times k$ convolution on an image of size $(h \times w)$ is $\mathcal{O}(h \cdot w \cdot k^2)$.
  - With $c_\textrm{i}$ (input channels) and $c_\textrm{o}$ (output channels), the cost becomes $\mathcal{O}(h \cdot w \cdot k^2 \cdot c_\textrm{i} \cdot c_\textrm{o})$.
  - For a 256×256 image using a 5×5 kernel and 128 input/output channels, this results in over 53 billion operations.
- **Future Considerations:**
  - Strategies to reduce costs include using block-diagonal channel-wise operations.
  - Introduction of architectures like ResNeXt to optimize performance.

7.4.5. Exercises

1. Assume that we have two convolution kernels of size k_1 and k_2, respectively (with no nonlinearity in between).
    1. Prove that the result of the operation can be expressed by a single convolution.
    2. What is the dimensionality of the equivalent single convolution?
    3. Is the converse true, i.e., can you always decompose a convolution into two smaller ones?
2. Assume an input of shape $c_\textrm{i} \times h \times w$ and a convolution kernel of shape $c_\textrm{o} \times c_\textrm{i} \times k_\textrm{h} \times k_\textrm{w}$, padding of $(p_\textrm{h}, p_\textrm{w})$, and stride of $(s_\textrm{h}, s_\textrm{w})$.
    1. What is the computational cost (multiplications and additions) for the forward propagation?
    2. What is the memory footprint?
    3. What is the memory footprint for the backward computation?
    4. What is the computational cost for the backpropagation?
3. By what factor does the number of calculations increase if we double both the number of input channels $c_\textrm{i}$ and the number of output channels $c_\textrm{o}$? What happens if we double the padding?
4. Are the variables `Y1` and `Y2` in the final example of this section exactly the same? Why?
5. Express convolutions as a matrix multiplication, even when the convolution window is not 1×1.
6. Your task is to implement fast convolutions with a $k \times k$ kernel. One of the algorithm candidates is to scan horizontally across the source, reading a k-wide strip and computing the 1-wide output strip one value at a time. The alternative is to read a $k + \Delta$ wide strip and compute a Δ-wide output strip. Why is the latter preferable? Is there a limit to how large you should choose Δ?
7. Assume that we have a $c \times c$ matrix.
    1. How much faster is it to multiply with a block-diagonal matrix if the matrix is broken up into b blocks?
    2. What is the downside of having b blocks? How could you fix it, at least partly?

7.5. Pooling

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

7.5.1. Maximum Pooling and Average Pooling

- **Definition and Functionality:**
  - Pooling operators use a fixed-shape window that slides over the input, computing a single output for each position.
  - Unlike convolutional layers, pooling layers have no parameters (no kernel) and are deterministic.
  - Two main types of pooling:
    - **Max-Pooling:** Computes the maximum value within the pooling window.
    - **Average Pooling:** Computes the average value of the elements in the pooling window.
- **Purpose of Pooling:**
  - Aims to downsample an image and reduce dimensions while retaining important information.
  - Helps improve the signal-to-noise ratio by combining information from adjacent pixels.
- **Historical Context:**
  - Average pooling has been used since the inception of CNNs.
  - Max-pooling was introduced in cognitive neuroscience for object recognition and has earlier applications in speech recognition.
- **Operational Mechanics:**
  - The pooling window starts at the upper-left of the input tensor and slides across it horizontally and vertically.
  - At each position, it calculates either the maximum or average value of the input subtensor.
- **Example of Max-Pooling:**
  - For a 2×2 max-pooling operation, the output tensor is generated by taking the maximum value from each 2×2 region in the input.

- For example:
    - $\max(0, 1, 3, 4) = 4$
    - $\max(1, 2, 4, 5) = 5$
    - $\max(3, 4, 6, 7) = 7$
    - $\max(4, 5, 7, 8) = 8$

- **Pooling Layer and Edge Detection:**
    - The output of the convolutional layer is used as input for pooling (e.g., 2×2 max-pooling).
    - This maintains the ability to detect patterns even when they move slightly, confirming the effectiveness of pooling in feature detection.

- **Implementation:**
    - The forward propagation of the pooling layer can be implemented in the `pool2d` function, similar to the `corr2d` function but without requiring a kernel.

```
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.]])
```

7.5.2. 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)
# Pooling has no model parameters, hence it needs no initialization
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.]]]])
```

7.5.3. Multiple Channels

- **Separate Pooling:** Each input channel is pooled independently.
- **Output Channels:** Number of output channels equals the number of input channels.
- **Example:** Tensors (X) and (X + 1) can be concatenated along the channel dimension to create a two-channel input.

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

7.5.4. Summary

- **Operation Overview:**
  - Pooling aggregates values over a specified window size.
- **Pooling Mechanics:**
  - Retains the same convolution semantics (strides and padding) as convolutional layers.
  - Indifferent to channels: number of channels remains unchanged, and pooling is applied to each channel separately.
- **Pooling Choices:**
  - **Max-Pooling** is generally preferred over average pooling due to its ability to provide some degree of invariance in outputs.
  - A common pooling window size is (2 \times 2), which reduces the spatial resolution of the output by a factor of four.
- **Alternative Resolution Reduction Methods:**
  - **Stochastic Pooling:** Incorporates randomization into aggregation, potentially improving accuracy.
  - **Fractional Max-Pooling:** A variant that also combines randomization for resolution reduction.
- **Advanced Aggregation Techniques:**
  - Attention mechanisms offer refined methods for aggregating outputs based on the alignment between query and representation vectors.

7.5.5. Exercises

1. Implement average pooling through a convolution.
2. Prove that max-pooling cannot be implemented through a convolution alone.
3. Max-pooling can be accomplished using ReLU operations, i.e., $\mathrm{ReLU}(x) = \max(0, x)$.
    1. Express $\max(a, b)$ by using only ReLU operations.
    2. Use this to implement max-pooling by means of convolutions and ReLU layers.
    3. How many channels and layers do you need for a 2×2 convolution? How many for a 3×3 convolution?
4. What is the computational cost of the pooling layer? Assume that the input to the pooling layer is of size $c \times h \times w$, the pooling window has a shape of $p_\mathrm{h} \times p_\mathrm{w}$ with a padding of $(p_\mathrm{h}, p_\mathrm{w})$ and a stride of $(s_\mathrm{h}, s_\mathrm{w})$.
5. Why do you expect max-pooling and average pooling to work differently?
6. Do we need a separate minimum pooling layer? Can you replace it with another operation?
7. We could use the softmax operation for pooling. Why might it not be so popular?

7.6. Convolutional Neural Networks (LeNet)

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

7.6.1. LeNet

- **Architecture Overview:**
  - LeNet consists of two parts:
    1. **Convolutional Encoder:** Two convolutional layers.
    2. **Dense Block:** Three fully connected layers.
- **Convolutional Block:**
  - Each block contains: Convolutional layer with a 5×5 kernel.
    - Sigmoid activation function.
    - Average pooling operation (stride 2) for downsampling.
  - The first convolutional layer has 6 output channels, and the second has 16.
  - Reduces spatial dimensions by a factor of 4 after pooling.
- **Dense Block:**
  - Three fully connected layers with outputs of size 120, 84, and 10 (for 10-class classification).
  - Input to the dense block is flattened from four dimensions (batch size, channels, height, width) to two dimensions.
- **Historical Context:**
  - LeNet used sigmoid activations and average pooling as ReLU and max-pooling were not yet discovered.
- **Modern Implementation:**
  - Can be easily implemented with deep learning frameworks using a `Sequential` block and Xavier initialization.

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

class LeNet(d2l.Classifier):  #@save
    """The LeNet-5 model."""
    def __init__(self, lr=0.1, num_clas
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.LazyConv2d(6, kernel_siz
            nn.AvgPool2d(kernel_size=2,
            nn.LazyConv2d(16, kernel_si
            nn.AvgPool2d(kernel_size=2,
            nn.Flatten(),
            nn.LazyLinear(120), nn.Sigm
            nn.LazyLinear(84), nn.Sigmo
            nn.LazyLinear(num_classes))


@d2l.add_to_class(d2l.Classifier)  #@sa
def layer_summary(self, X_shape):
    X = torch.randn(*X_shape)
    for layer in self.net:
        X = layer(X)
        print(layer.__class__.__name__,

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

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

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

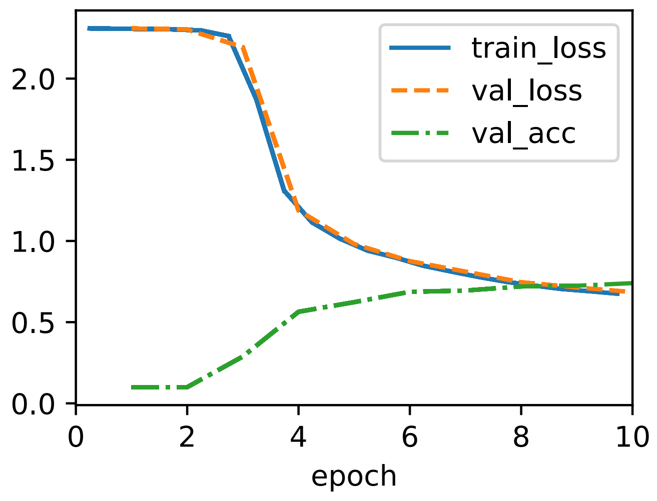
"@save" is not an allowed annotation - allowed values include [@param, @title, @markdown].

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

7.6.2. Training

- **Computational Cost:**
  - Despite fewer parameters, CNNs can be computationally expensive because each parameter is involved in more multiplications compared to MLPs.
- **GPU Utilization:**
  - Utilizing a GPU can significantly speed up training for CNNs.
- **Training Process:**
  - The `d2l.Trainer` class manages device initialization and model parameters.
  - The loss function used is cross-entropy, and the model is trained using minibatch stochastic gradient descent (SGD).

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



7.6.3. Summary

- **Historical Progress:**
    - Transitioned from MLPs in the 1980s to CNNs in the 1990s and 2000s.
    - LeNet-5, developed in this era, remains relevant and effective today.
- **Comparative Performance:**
    - LeNet-5 performs significantly better than MLPs on tasks like Fashion-MNIST and is more similar in approach to advanced architectures like ResNet.
- **Advancements in Computation:**
    - Increased computational power has enabled the development of more complex architectures, such as ResNet, compared to LeNet.
- **Ease of Implementation:**
    - What once took months of engineering with C++ and Lisp-based tools can now be implemented in minutes, reflecting the dramatic productivity boost in deep learning development.
- **Next Steps:**
    - Future chapters will explore more advanced architectures and the current state of deep learning techniques.

7.6.4. Exercises

1. Let's modernize LeNet. Implement and test the following changes:
    1. Replace average pooling with max-pooling.
    2. Replace the softmax layer with ReLU.
2. Try to change the size of the LeNet style network to improve its accuracy in addition to max-pooling and ReLU.
    1. Adjust the convolution window size.
    2. Adjust the number of output channels.
    3. Adjust the number of convolution layers.
    4. Adjust the number of fully connected layers.
    5. Adjust the learning rates and other training details (e.g., initialization and number of epochs).
3. Try out the improved network on the original MNIST dataset.
4. Display the activations of the first and second layer of LeNet for different inputs (e.g., sweaters and coats).
5. What happens to the activations when you feed significantly different images into the network (e.g., cats, cars, or even random noise)?

8.2. Networks Using Blocks (VGG)

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

8.2.1. VGG Blocks

- **Basic Structure of CNNs:**
  - A sequence of convolutional layers, nonlinear activation (ReLU), and pooling layers (e.g., max-pooling).
  - This approach quickly reduces spatial resolution, limiting the number of convolutional layers.
- **VGG Innovation:**
  - **Simonyan & Zisserman (2014)** proposed using multiple convolutional layers between pooling layers, forming a block structure.
  - Successive (3 \times 3) convolutions replace larger kernels (e.g., (5 \times 5)) for a similar receptive field but fewer parameters.
- **Deep vs. Shallow Networks:**
  - Deep and narrow networks were shown to outperform shallow networks, leading to deeper networks with over 100 layers becoming standard.
- **VGG Block Design:**
  - Each VGG block consists of multiple (3 \times 3) convolutions (with padding of 1 to maintain spatial resolution) followed by a (2 \times 2) max-pooling layer (stride 2, halving the height and width).
- **Impact on Deep Learning:**
  - Stacking small (3 \times 3) convolutions became a standard design choice in later deep networks.
- **Efficiency:**
  - Fast GPU implementations for small convolutions have become critical for practical applications.

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

8.2.2. VGG Network

- **Two-Part Structure:**
  - Similar to AlexNet and LeNet, the VGG network is divided into two parts:
    1. **Convolutional and Pooling Layers:** Responsible for feature extraction.
    2. **Fully Connected Layers:** Identical to those in AlexNet, used for classification.
- **Key Difference from AlexNet:**
  - VGG uses **blocks of convolutional layers** grouped into nonlinear transformations, followed by resolution reduction, unlike AlexNet's individually designed layers.
- **VGG Blocks:**
  - The network connects multiple VGG blocks (as defined in the `vgg_block` function) in succession, forming a deep architecture.
- **Family of Networks:**
  - VGG defines a **family of networks**, varying the number of convolutional layers and output channels in each block, allowing flexible architectures.
- **Long-lasting Design:**
  - This block-based convolutional structure has been a standard for over a decade, although specific operations have evolved over time.

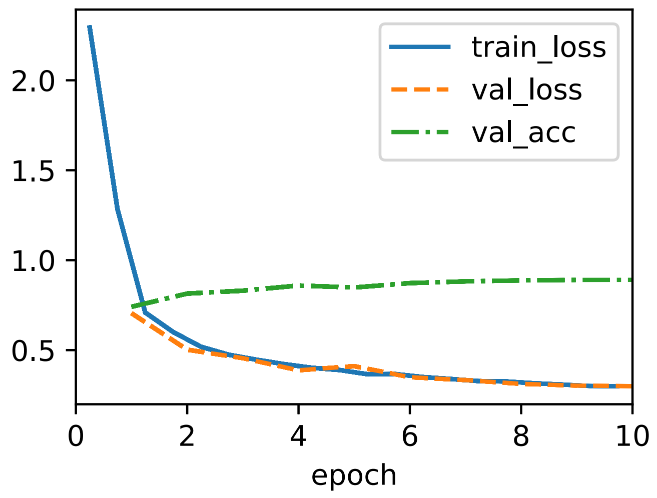
```
class VGG(d2l.Classifier):
    def __init__(self, arch, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        conv_blks = []
        for (num_convs, out_channels) in arch:
            conv_blks.append(vgg_block(num_convs, out_channels))
        self.net = 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])
```

8.2.3. Training

```
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)
```



8.2.4. Summary

- **VGG as a Modern CNN:**
  - VGG is considered the first truly modern CNN, introducing:
    - **Blocks of multiple convolutions** instead of isolated layers.
    - A preference for **deep and narrow** networks.
- **VGG's Flexibility:**
  - VGG defines an entire **family of networks** with similar architectures, offering trade-offs between complexity and computational speed.
- **Ease of Implementation:**
  - Modern deep learning frameworks enable building networks with simple Python code, replacing the need for cumbersome XML configurations.
- **Recent Developments:**
  - **ParNet (2021)** demonstrated competitive performance with shallow architectures using **parallel computations**, signaling potential future shifts in CNN architecture design.
- **Legacy and Influence:**
  - While innovations like ParNet are promising, the chapter follows the historical evolution of deep learning architectures, where VGG remains foundational.

8.2.5. Exercises

1. Compared with AlexNet, VGG is much slower in terms of computation, and it also needs more GPU memory.
    1. Compare the number of parameters needed for AlexNet and VGG.
    2. Compare the number of floating point operations used in the convolutional layers and in the fully connected layers.
    3. How could you reduce the computational cost created by the fully connected layers?
2. When displaying the dimensions associated with the various layers of the network, we only see the information associated with eight blocks (plus some auxiliary transforms), even though the network has 11 layers. Where did the remaining three layers go?
3. Use Table 1 in the VGG paper :cite: `Simonyan.Zisserman.2014` to construct other common models, such as VGG-16 or VGG-19.
4. Upsampling the resolution in Fashion-MNIST eight-fold from 28×28 to 224×224 dimensions is very wasteful. Try modifying the network architecture and resolution conversion, e.g., to 56 or to 84 dimensions for its input instead. Can you do so without reducing the accuracy of the network? Consult the VGG paper :cite: `Simonyan.Zisserman.2014` for ideas on adding more nonlinearities prior to downsampling.

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
```

8.6.1. Function Classes

- **Function Class**:
    - The set of functions, denoted as $\mathcal{F}$, represents the range of functions a network architecture can model through specific parameters (weights, biases, etc.).
    - The "truth" function, f^*, may or may not be in $\mathcal{F}$, and we aim to find the best approximation, $f^*_{\mathcal{F}}$, within $\mathcal{F}$ by minimizing a loss function.
- **Complexity and Regularization**:
    - Regularization helps control the complexity of $\mathcal{F}$, and larger training data generally leads to better $f^*_{\mathcal{F}}$.
    - A more powerful function class, $\mathcal{F}'$, may not always yield a better solution if $\mathcal{F} \nsubseteq \mathcal{F}'$.
- **Nested vs. Non-nested Classes**:
    - For **non-nested function classes** (e.g., $\mathcal{F}_3, \mathcal{F}_6$), increasing complexity may not always bring the model closer to f^*.
    - For **nested classes** (e.g., $\mathcal{F}_1 \subseteq \mathcal{F}_6$), increasing complexity improves expressiveness and ensures that the added layers can still perform as well as the original model.
- **ResNet and Residual Blocks**:
    - **ResNet (2016)**, by Kaiming He et al., addressed the issue of increasing depth by introducing **residual blocks**, where each new layer can be trained to represent the identity function, $f(\mathbf{x}) = \mathbf{x}$.
    - This innovation allowed the network to become deeper without sacrificing performance, leading to ResNet's victory in the **2015 ImageNet Challenge**.
    - Residual blocks have influenced many areas, including recurrent networks, Transformers, graph neural networks, and more.
- **Highway Networks**:
    - Residual networks are related to **highway networks** (2015), which share similar motivations but lack the identity function's elegant parameterization.

8.6.2. Residual Blocks

1. **Residual Block Concept**:
   - **Input**: Denoted as $\mathbf{x}$, the desired function $f(\mathbf{x})$ is the target of learning.
   - **Residual Mapping**: Instead of learning $f(\mathbf{x})$ directly, residual blocks learn the difference $g(\mathbf{x}) = f(\mathbf{x}) - \mathbf{x}$.
   - **Identity Mapping**: If $f(\mathbf{x}) = \mathbf{x}$, then $g(\mathbf{x}) = 0$, making it easier to learn. The weights and biases can simply be pushed to zero.
2. **Residual Connection**:
   - The **solid line** carrying $\mathbf{x}$ through the block without transformation is known as a **residual connection** or **shortcut connection**.
   - It allows faster propagation of information through layers and simplifies learning when the identity function is sufficient.
3. **Relation to Inception Blocks**:
   - The residual block is similar to the multi-branch **Inception block**, with one of the branches being the identity mapping.
4. **ResNet Architecture**:
   - **VGG-like Design**: ResNet uses a full 3×3 convolutional layer design similar to VGG, with two 3×3 convolutional layers in each residual block.
   - **Batch Normalization and ReLU**: Each convolutional layer is followed by batch normalization and ReLU activation.
   - **Skip Connection**: After the two convolutional layers, the input $\mathbf{x}$ is added to the output of the convolutional operations before applying the final ReLU activation.
5. **Shape Compatibility**:
   - The output of the two convolutional layers must have the same shape as the input for the addition to work.
   - If a change in the number of channels is required, a 1×1 convolutional layer is added to adjust the input to the desired shape.

```
class Residual(nn.Module):  #@save
    """The Residual block of ResNet moc
    def __init__(self, num_channels, us
        super().__init__()
        self.conv1 = nn.LazyConv2d(num_
                                   str
        self.conv2 = nn.LazyConv2d(num_
        if use_1x1conv:
            self.conv3 = nn.LazyConv2d(

        else:
            self.conv3 = None
        self.bn1 = nn.LazyBatchNorm2d()
        self.bn2 = nn.LazyBatchNorm2d()

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

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

```
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])
```

8.6.3. ResNet Model

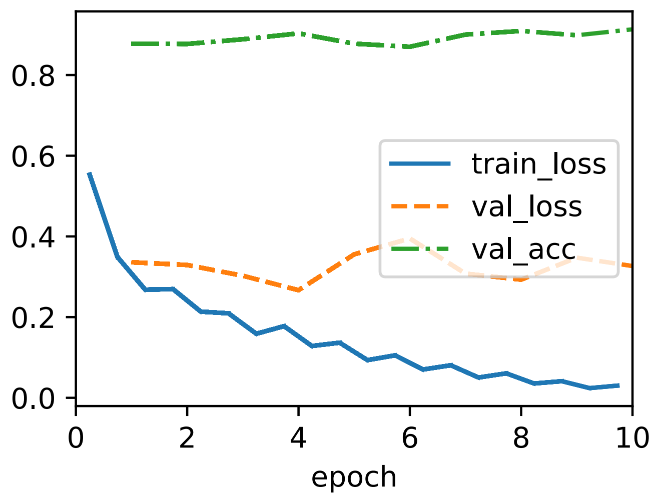
```
class ResNet(d2l.Classifier):
    def b1(self):
        return nn.Sequential(
            nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),
            nn.LazyBatchNorm2d(), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

```
@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])
```

8.6.4. Training

```
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)
```


epoch
train_loss
val_loss
val_acc