# 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_x86_64.whl.metadata (2.3 kB)
Collecting matplotlib==3.7.2 (from d2l==1.0.3)
  Downloading matplotlib-3.7.2-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (5.6 kB)
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_x86_64.whl.metadata (18 kB)
Collecting scipy==1.10.1 (from d2l==1.0.3)
  Downloading scipy-1.10.1-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (58 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 58.9/58.9 kB 2.3 MB/s eta 0:00:00
Requirement already satisfied: notebook in /usr/local/lib/python3.10/dist-packages (from jupyter==1.0.0->d2l==1.0.
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/dist-packages (from jupyter==1.0.0->d2
Requirement already satisfied: nbconvert in /usr/local/lib/python3.10/dist-packages (from jupyter==1.0.0->d2l==1.0
Requirement already satisfied: ipykernel in /usr/local/lib/python3.10/dist-packages (from jupyter==1.0.0->d2l==1.0
Requirement already satisfied: ipywidgets in /usr/local/lib/python3.10/dist-packages (from jupyter==1.0.0->d2l==1.
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib==3.7.2
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-packages (from matplotlib==3.7.2->d2
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib==3.7.
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib==3.7.
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib==3.7.2-
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib==3.7.2->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/python3.10/dist-packages (from matplotlib==3
Requirement already satisfied: traitlets in /usr/local/lib/python3.10/dist-packages (from matplotlib-inline==0.1.6
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas==2.0.3->d2l==1
Requirement already satisfied: tzdata>=2022.1 in /usr/local/lib/python3.10/dist-packages (from pandas==2.0.3->d2l=
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests=
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests==2.31.0->d2l
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests==2.31.
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests==2.31.
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.7->mat
Requirement already satisfied: ipython-genutils in /usr/local/lib/python3.10/dist-packages (from ipykernel->jupyte
Requirement already satisfied: ipython>=5.0.0 in /usr/local/lib/python3.10/dist-packages (from ipykernel->jupyter=
Requirement already satisfied: jupyter-client in /usr/local/lib/python3.10/dist-packages (from ipykernel->jupyter=
Requirement already satisfied: tornado>=4.2 in /usr/local/lib/python3.10/dist-packages (from ipykernel->jupyter==1
Requirement already satisfied: widgetsnbextension~=3.6.0 in /usr/local/lib/python3.10/dist-packages (from ipywidge
Requirement already satisfied: jupyterlab-widgets>=1.0.0 in /usr/local/lib/python3.10/dist-packages (from ipywidge
Requirement already satisfied: prompt-toolkit!=3.0.0,!=3.0.1,<3.1.0,>=2.0.0 in /usr/local/lib/python3.10/dist-pack
Requirement already satisfied: pygments in /usr/local/lib/python3.10/dist-packages (from jupyter-console->jupyter=
Requirement already satisfied: lxml in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1.0.0->d2
Requirement already satisfied: beautifulsoup4 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter=
Requirement already satisfied: bleach in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1.0.0->
Requirement already satisfied: defusedxml in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1.0
Requirement already satisfied: entrypoints>=0.2.2 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupy
Requirement already satisfied: jinja2>=3.0 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1.
Requirement already satisfied: jupyter-core>=4.7 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyt
Requirement already satisfied: jupyterlab-pygments in /usr/local/lib/python3.10/dist-packages (from nbconvert->jup
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter
Requirement already satisfied: mistune<2,>=0.8.1 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyt
Requirement already satisfied: nbclient>=0.5.0 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter
Requirement already satisfied: nbformat>=5.1 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==
Requirement already satisfied: pandocfilters>=1.4.1 in /usr/local/lib/python3.10/dist-packages (from nbconvert->ju
Requirement already satisfied: tinycss2 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1.0.0
Requirement already satisfied: pyzmq<25,>=17 in /usr/local/lib/python3.10/dist-packages (from notebook->jupyter==1
Requirement already satisfied: argon2-cffi in /usr/local/lib/python3.10/dist-packages (from notebook->jupyter==1.0
Requirement already satisfied: nest-asyncio>=1.5 in /usr/local/lib/python3.10/dist-packages (from notebook->jupyte
Requirement already satisfied: Send2Trash>=1.8.0 in /usr/local/lib/python3.10/dist-packages (from notebook->jupyte
Requirement already satisfied: terminado>=0.8.3 in /usr/local/lib/python3.10/dist-packages (from notebook->jupyter
Requirement already satisfied: prometheus-client in /usr/local/lib/python3.10/dist-packages (from notebook->jupyte
Requirement already satisfied: nbclassic>=0.4.7 in /usr/local/lib/python3.10/dist-packages (from notebook->jupyter
```

```
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/python3.10/dist-packages (from ipython>=5.0.0->i
Collecting jedi>=0.16 (from ipython>=5.0.0->ipykernel->jupyter==1.0.0->d2l==1.0.3)
  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-packages (from ipython>=5.0.0->ipykerne
Requirement already satisfied: pickleshare in /usr/local/lib/python3.10/dist-packages (from ipython>=5.0.0->ipyker
Requirement already satisfied: backcall in /usr/local/lib/python3.10/dist-packages (from ipython>=5.0.0->ipykernel
Requirement already satisfied: pexpect>4.3 in /usr/local/lib/python3.10/dist-packages (from ipython>=5.0.0->ipyker
Requirement already satisfied: platformdirs>=2.5 in /usr/local/lib/python3.10/dist-packages (from jupyter-core>=4.
Requirement already satisfied: notebook-shim>=0.2.3 in /usr/local/lib/python3.10/dist-packages (from nbclassic>=0.
Requirement already satisfied: fastjsonschema>=2.15 in /usr/local/lib/python3.10/dist-packages (from nbformat>=5.1
Requirement already satisfied: jsonschema>=2.6 in /usr/local/lib/python3.10/dist-packages (from nbformat>=5.1->nbc
Requirement already satisfied: wcwidth in /usr/local/lib/python3.10/dist-packages (from prompt-toolkit!=3.0.0,!=3.
Requirement already satisfied: ptyprocess in /usr/local/lib/python3.10/dist-packages (from terminado>=0.8.3->noteb
Requirement already satisfied: argon2-cffi-bindings in /usr/local/lib/python3.10/dist-packages (from argon2-cffi->
Requirement already satisfied: soupsieve>1.2 in /usr/local/lib/python3.10/dist-packages (from beautifulsoup4->nbco
Requirement already satisfied: webencodings in /usr/local/lib/python3.10/dist-packages (from bleach->nbconvert->ju
Requirement already satisfied: parso<0.9.0,>=0.8.3 in /usr/local/lib/python3.10/dist-packages (from jedi>=0.16->ip
Requirement already satisfied: attrs>=22.2.0 in /usr/local/lib/python3.10/dist-packages (from jsonschema>=2.6->nbf
Requirement already satisfied: jsonschema-specifications>=2023.03.6 in /usr/local/lib/python3.10/dist-packages (fr
Requirement already satisfied: referencing>=0.28.4 in /usr/local/lib/python3.10/dist-packages (from jsonschema>=2.
Requirement already satisfied: rpds-py>=0.7.1 in /usr/local/lib/python3.10/dist-packages (from jsonschema>=2.6->nb
Requirement already satisfied: jupyter-server<3,>=1.8 in /usr/local/lib/python3.10/dist-packages (from notebook-sh
Requirement already satisfied: cffi>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from argon2-cffi-bindings->
Requirement already satisfied: pycparser in /usr/local/lib/python3.10/dist-packages (from cffi>=1.0.1->argon2-cffi
Requirement already satisfied: anyio<4,>=3.1.0 in /usr/local/lib/python3.10/dist-packages (from jupyter-server<3,>
Requirement already satisfied: websocket-client in /usr/local/lib/python3.10/dist-packages (from jupyter-server<3,
Requirement already satisfied: sniffio>=1.1 in /usr/local/lib/python3.10/dist-packages (from anyio<4,>=3.1.0->jupy
Requirement already satisfied: exceptiongroup in /usr/local/lib/python3.10/dist-packages (from anyio<4,>=3.1.0->ju
Downloading d2l-1.0.3-py3-none-any.whl (111 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 111.7/111.7 kB 2.2 MB/s eta 0:00:00
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.manylinux2014_x86_64.whl (11.6 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 11.6/11.6 MB 70.2 MB/s eta 0:00:00
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_x86_64.whl (17.1 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 17.1/17.1 MB 72.4 MB/s eta 0:00:00
Downloading pandas-2.0.3-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (12.3 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 12.3/12.3 MB 78.2 MB/s eta 0:00:00
Downloading requests-2.31.0-py3-none-any.whl (62 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 62.6/62.6 kB 5.7 MB/s eta 0:00:00
Downloading scipy-1.10.1-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (34.4 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 34.4/34.4 MB 14.9 MB/s eta 0:00:00
Downloading pyparsing-3.0.9-py3-none-any.whl (98 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 98.3/98.3 kB 7.9 MB/s eta 0:00:00
Downloading qtconsole-5.6.0-py3-none-any.whl (124 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 124.7/124.7 kB 11.2 MB/s eta 0:00:00
Downloading QtPy-2.4.1-py3-none-any.whl (93 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 93.5/93.5 kB 7.5 MB/s eta 0:00:00
Using cached jedi-0.19.1-py2.py3-none-any.whl (1.6 MB)
Installing collected packages: requests, qtpy, pyparsing, numpy, matplotlib-inline, jedi, scipy, pandas, matplotli
  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:
```

```
    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 the packages that are installed. This be
albucore 0.0.16 requires numpy>=1.24, but you have numpy 1.23.5 which is incompatible.
albumentations 1.4.15 requires numpy>=1.24.4, but you have numpy 1.23.5 which is incompatible.
bigframes 1.21.0 requires numpy>=1.24.0, but you have numpy 1.23.5 which is incompatible.
chex 0.1.87 requires numpy>=1.24.1, but you have numpy 1.23.5 which is incompatible.
google-colab 1.0.0 requires pandas==2.2.2, but you have pandas 2.0.3 which is incompatible.
google-colab 1.0.0 requires requests==2.32.3, but you have requests 2.31.0 which is incompatible.
jax 0.4.33 requires numpy>=1.24, but you have numpy 1.23.5 which is incompatible.
jaxlib 0.4.33 requires numpy>=1.24, but you have numpy 1.23.5 which is incompatible.
mizani 0.11.4 requires pandas>=2.1.0, but you have pandas 2.0.3 which is incompatible.
plotnine 0.13.6 requires pandas<3.0.0,>=2.1.0, but you have pandas 2.0.3 which is incompatible.
xarray 2024.9.0 requires numpy>=1.24, but you have numpy 1.23.5 which is incompatible.
xarray 2024.9.0 requires pandas>=2.1, but you have pandas 2.0.3 which is incompatible.
Successfully installed d2l-1.0.3 jedi-0.19.1 jupyter-1.0.0 matplotlib-3.7.2 matplotlib-inline-0.1.6 numpy-1.23.5 p
```

# 7.1 From Fully Connected Layers to Convolutions

## 7.2 Convolutions for Images

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

```
def corr2d(X, K):  #@save
    """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
```

'@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.]])
```

```
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.]])
```

```
# 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 for-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 14.879
epoch 4, loss 4.442
epoch 6, loss 1.542
epoch 8, loss 0.585
epoch 10, loss 0.232
```

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

```
tensor([[ 0.9373, -1.0356]])
```

## 7.3 Padding and Stride

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

```
# 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])
```

```
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])
```

## 7.4 Multiple Input and Multiple Output Channels

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

```
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.]])
```

```
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.]]])
```

```
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
```

## 7.5 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)
# 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.]]]])
```

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

## 7.6 Convolutional Neural Networks (LeNet)

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

```
def init_cnn(module): #@save
  """Initialize weights for CNNs."""
  if type(module) == nn.Linear or type(module) == nn.Conv2
    nn.init.xavier_uniform_(module.weight)

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

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

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

```
        nn.AvgPool2d(kernel_size=2, stride=2),
        nn.Flatten(),
        nn.LazyLinear(120), nn.Sigmoid(),
        nn.LazyLinear(84), nn.Sigmoid(),
        nn.LazyLinear(num_classes))
```

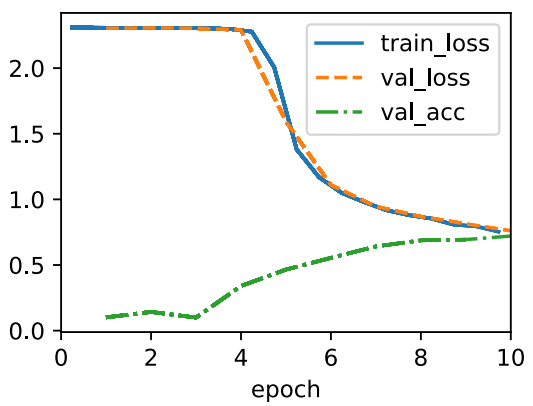
```
@d2l.add_to_class(d2l.Classifier) #@save
def layer_summary(self, X_shape):
  X = torch.randn(*X_shape)
  for layer in self.net:
    X = layer(X)
    print(layer.__class__.__name__, 'output shape:\t', X.sh

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

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

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

```
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)
```




## 8.2 Networks Using Blocks (VGG)

```
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())
```

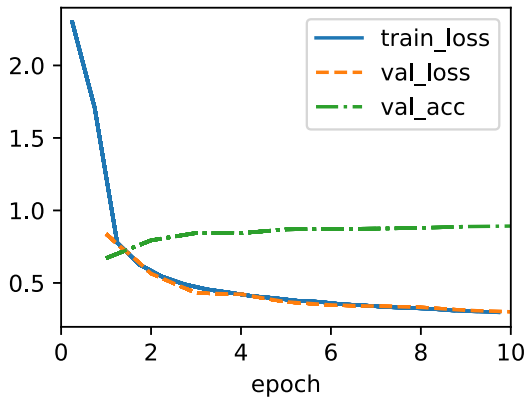
```
  layers.append(nn.MaxPool2d(kernel_size=2,stride=2))
  return nn.Sequential(*layers)
```

```
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)
```




## 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):  #@save
  """The Residual block of ResNet models."""
  def __init__(self, num_channels, use_1x1conv=False, stri
    super().__init__()
    self.conv1 = nn.LazyConv2d(num_channels, kernel_size=3
                               stride=strides)
    self.conv2 = nn.LazyConv2d(num_channels, kernel_size=3
    if use_1x1conv:
      self.conv3 = nn.LazyConv2d(num_channels, kernel_size
                                 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)
```

'@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])
```

```
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(),
```

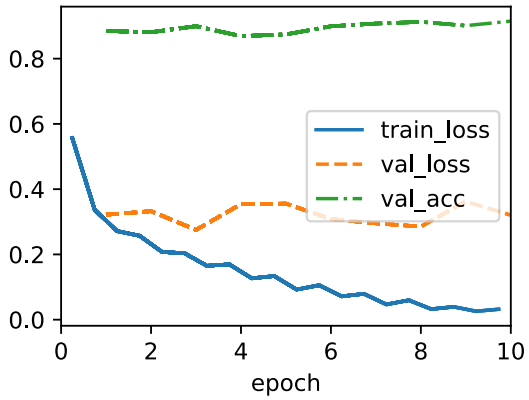
```
      nn.LazyLinear(num_classes)))
  self.net.apply(d2l.init_cnn)
```

```
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)
```




# Discussion and Exercises

## 7.1 Discussion

7.1. From Fully Connected Layers to Convolutions

- Processing high-resolution images using fully connected layers is computationally expensive, requiring millions or billions of parameters.
- Convolutional Neural Networks (CNNs) provide a more efficient approach by exploiting the inherent structure in images to reduce the number of parameters while preserving performance.

7.1.1. Invariance

- In image recognition, the location of objects should not affect their detection (translation invariance). CNNs detect patterns in any part of the images.
- Early CNN layers focus on local patterns, while deeper layers capture broader, more complex image features, leading to spatial invariance.

7.1.2. Constraining the MLP

- The spatial structure of input images and hidden representations can be utilized by representing parameters as fourth-order tensors. However, the parameter requirements for fully connected layers in image processing are prohibitively high, necessitating more efficient methods like convolutions.

7.1.2.1. Translation Invariance

- Translation invariance simplifies image processing by making network weights independent of pixel location, reducing parameter requirements.

7.1.2.2. Locality

- The locality principle implies that only nearby pixels are needed to understand the image content at any specific location. This further reduces the number of parameters and leads to the creation of convolutional layers.

7.1.3. Convolutions

- Convolution is the mathematical operation that underpins CNNs, measuring overlap between a filter and a portion of the input image.
- CNN layers apply filters across the image to extract important spatial features, reducing the number of parameters compared to fully connected layers.

7.1.4. Channels

- CNNs extend convolution operations to handle multiple channels by treating images as third-order tensors, allowing deeper layers to specialize in detecting more complex features.

## 7.1 Exercieses

1. When the size of the convolution kernel Δ is 0, the convolution operation does not involve any spatial interactions between neighboring pixels. This means that each output channel is computed as a weighted sum of the input channels, independent of spatial location. This effectively reduces the convolution to a multi-layer perceptron (MLP) applied independently to each spatial position of the feature map, across all channels. This leads to Network in Network (NIN) architectures (Lin et al., 2013), where 1x1 convolutions are used to increase the model's expressive power without increasing the spatial size of the data.
2. 1. Locality is crucial for detecting short-term patterns like phonemes or notes, which are essential in speech and music recognition. Translation invariance helps in recognizing the same sound pattern irrespective of its position in the time sequence.
   2. For a one-dimensional sequence like audio, the convolution operation is defined as:
      $$(f * g)(t) = \sum_i f(i) \cdot g(t - i)$$
      Here, $f(i)$ is the input signal and $g(i)$ is the convolution filter. The filter slides along the time axis of the audio sequence, and at each step, the dot product between the filter and a portion of the input sequence is computed.
   3. Audio can be processed similarly to images by converting the raw waveform into a spectrogram, representing the signal's frequency content over time. Convolutional layers, often used in computer vision, can be applied to this 2D spectrogram to detect features like pitches, rhythms, or phonemes.
3. Translation invariance may not be desirable when the location of features matters. For example, in face recognition, the relative positions of eyes, nose, and mouth are crucial. If these features are detected but in the wrong places, the translation invariance might lead to false positives or wrong interpretations.
4. Convolutional layers can be applied to text data for tasks like sentiment analysis or named entity recognition by sliding filters over word embeddings. However, some challenges that may arrise are issues with word order and variable lengths.

Convolutional layers are less sensitive to the order of words beyond a local context, which can be critical for understanding meaning in text. Text sequences vary in length, and padding/truncation may cause information loss or distortions.

5. When an object is at the boundary of an image, convolution operations may lose part of the object's information because the kernel cannot fully cover the boundary region. A simple solution for this is the use of padding to handle boundaries more smoothly.

6. To prove the symmetry of convolution:

$$(f * g)(t) = \sum_i f(i) \cdot g(t - i)$$

Switching $f$ and $g$, results in:

$$(g * f)(t) = \sum_i g(i) \cdot f(t - i)$$

Since summation is commutative, it complies with $(f * g)(t) = (g * f)(t)$. Hence, convolution is symmetric.

## 7.2 Discussion

7.2.1 The Cross-Correlation Operation:

- Convolutional layers are actually cross-correlation operations.
- A kernel tensor slides over an input tensor, multiplying and summing elements within a sliding window to produce the output tensor.
- Output size is typically smaller than the input due to the kernel's size, unless zero-padding is used.

7.2.2 Convolutional Layers:

- A convolutional layer performs cross-correlation between input and kernel tensors, adding a bias to the result.
- The process of forward propagation applies cross-correlation followed by adding bias to produce the output. The size of kernels defines the properties of the convolutional layer.

7.2.3 Object Edge Detection in Images:

- Convolutional layers can detect edges in images by capturing the pixel value differences between adjacent pixels.
- A kernel can be designed for edge detection, producing positive or negative values when it detects transitions from light to dark and vice versa.

7.2.4 Learning a Kernel:

- Instead of manually designing a kernel, we can train CNNs to learn the kernel that fits the desired output. The process involves training a convolutional layer, calculating the loss, and updating the kernel based on gradients.

7.2.5 Cross-Correlation and Convolution:

- Cross-correlation and strict convolution operations are nearly identical, differing only in kernel flipping.
- Despite this difference, the outputs of CNNs are unaffected as the learned kernel compensates for the operation type.

7.2.6 Feature Map and Receptive Field:

- The receptive field refers to the portion of the input that influences a particular output element. Deeper networks extend the receptive field, allowing neurons to detect broader patterns in images, similar to biological visual systems.

## 7.2 Exercises

1. An image $X$ with diagonal edges can be created by having the pixel values transition from one side of the diagonal to the other.

1. The kernel $K = \begin{bmatrix} 1 & -1 \end{bmatrix}$ is designed to detect horizontal edges by computing the difference between adjacent horizontal pixel values. When applied to an image with diagonal edges like $X$, the result will not show strong responses along the diagonal. It will instead detect any slight horizontal transitions along the edges of the diagonal. Since $K$ only detects horizontal changes, the output would have minimal detection in diagonal regions but would capture any slight horizontal edge changes.
2. If $X$ is transposed, the diagonal becomes the anti-diagonal. Since the kernel $K$ detects horizontal edges, it will again perform weakly on the anti-diagonal pattern, similar to how it behaves on the original diagonal.
3. Transposing $K$ makes it sensitive to vertical edges instead of horizontal ones. A transposed kernel would look like:
$$K^T = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

When applied to the diagonal image $X$, this kernel will not strongly detect diagonal edges but will detect vertical edges. As diagonal edges have both horizontal and vertical components, only a portion of the edge will be detected.

2. 1. To detect edges orthogonal to a directional vector $v = (v_1, v_2)$, a kernel that is sensitive to changes in the direction $(v_2, -v_1)$ is required. One way to create such a kernel is by using finite differences along this orthogonal direction. The kernel for detecting edges orthogonal to $v$ can be written as:
$$K = \begin{bmatrix} v_2 & -v_1 \end{bmatrix}$$
2. The finite difference operator for the second derivative can be derived by applying finite differences twice. For example, in one dimension, the second derivative can be approximated by the difference:
$$\frac{\partial^2 f}{\partial x^2} \approx f(x+1) - 2f(x) + f(x-1)$$
The corresponding convolutional kernel in 1D is:
$$K = \begin{bmatrix} 1 & -2 & 1 \end{bmatrix}$$
In 2D, the kernel would be:
$$K = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$
This kernel detects regions in the image where there are second-order changes, such as corners or other high-frequency components.
3. A simple blur kernel is the averaging kernel, where each pixel is replaced by the average of its neighboring pixels. This kernel smooths the image by averaging out local variations, which can be useful for reducing noise or simplifying an image before further processing.
4. The minimum size of a kernel to compute a derivative of order $d$ is $d + 1$. For example:

- The first derivative requires a kernel of size 2 (e.g., $[1, -1]$).
- The second derivative requires a kernel of size 3 (e.g., $[1, -2, 1]$).
- The third derivative would require a kernel of size 4, and so on.

3.

4. A cross-correlation operation can be represented as a matrix multiplication by reshaping the input tensor into a matrix where each row corresponds to the flattened receptive field of the convolution window, and the kernel is also reshaped into a column vector. The cross-correlation then becomes a matrix multiplication between the reshaped input and the reshaped kernel. This approach allows for more efficient computation using matrix operations.

## 7.3 Discussion

7.3. Padding and Stride

- When performing convolutions, the output size decreases due to kernel application, especially with small kernels.
- Padding helps preserve the original image size, while stride allows for controlled downsampling.

7.3.1. Padding

- Applying convolutions reduces the number of pixels used, especially on the image's edges. Successive convolution layers can result in significant pixel loss and reduction in output size.
- Padding adds extra pixels around the image to increase its size, often using zeros. This preserves the input size or controls the reduction of dimensionality.
- Convolution kernels are typically used in odd sizesto preserve dimensionality and symmetry in padding.

7.3.2. Stride

- Stride refers to the number of steps the convolution window moves in each direction. Larger strides result in more significant downsampling, reducing the output size.
- Increasing the stride reduces the number of output elements. This can be used to reduce image resolution or improve computational efficiency.

## 7.3 Exercises

1. The formula for the output shape of a convolutional layer is:

$$\text{Output Height} = \frac{\text{Input Height} - \text{Kernel Height} + 2 \times \text{Padding Height}}{\text{Stride Height}} + 1$$

$$\text{Output Width} = \frac{\text{Input Width} - \text{Kernel Width} + 2 \times \text{Padding Width}}{\text{Stride Width}} + 1$$

Given:

- Input shape: $(8, 8)$
- Kernel size: $(3, 5)$
- Padding: $(0, 1)$
- Stride: $(3, 4)$

  Height:

$$\text{Output Height} = \frac{8 - 3 + 2 \times 0}{3} + 1 = \frac{5}{3} + 1 \approx 2$$

  Width:

$$\text{Output Width} = \frac{8 - 5 + 2 \times 1}{4} + 1 = \frac{4}{4} + 1 = 2$$

  So, the output shape is $(2, 2)$, which is consistent with the experimental result.

2. In audio processing, a stride of 2 means that the convolution operation will downsample the audio signal by a factor of 2. Essentially, every other time step is skipped, reducing the temporal resolution of the output by half. This can be useful for compressing the data or for speeding up the computation by reducing the number of samples processed.

3.

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

```
# Create a sample tensor
X = torch.rand(1, 1, 8, 8)

# Apply mirror padding using F.pad
padded_X = F.pad(X, (1, 1, 1, 1), mode='reflect')  # (left, right, top, bottom) padding

print(padded_X.shape)  # Should increase the size by 2 in both height and width
```

4. The computational benefits of a stride larger than 1 include:

- Fewer operations are required because the convolution window skips over parts of the input, reducing the total number of calculations.
- Memory consumption is also reduced by reducing the output size, which is especially helpful when working with large inputs or deep networks.

5. The statistical benefits of a stride larger than 1 include:

- A larger stride reduces the resolution of the feature map, potentially helping to eliminate redundant information and noise.
- With larger strides, the model becomes more invariant to small translations of the input, as it skips over minor changes in the input that might not be significant for classification or regression tasks.

6. A stride of 1/2 corresponds to up-sampling rather than down-sampling. This would involve inserting zeros between the elements of the input to effectively "stretch" it before applying the convolution operation. This can be useful in tasks such as super-resolution or upsampling in autoencoders and generative models. This is useful when the spatial resolution of an input needs to be increased, such as in image generation tasks or when reconstructing higher-resolution data from a lower-resolution input.

```
import torch.nn as nn

# Define a convolution layer
conv2d = nn.Conv2d(1, 1, kernel_size=3, padding=1)

# Upsample the input by a factor of 2 (which simulates a stride of 1/2)
upsample = nn.Upsample(scale_factor=2, mode='nearest')

# Apply upsampling and then convolution
X = torch.rand(1, 1, 8, 8)
upsampled_X = upsample(X)
output = conv2d(upsampled_X)

print(output.shape)
```

## 7.4 Discussion

7.4. Multiple Input and Multiple Output Channels

- The number of input channels should match the number of channels in the convolution kernel for accurate cross-correlation.

7.4.1. Multiple Input Channels

- Cross-correlation is applied independently for each channel, and the results are summed up to produce the output.

7.4.2. Multiple Output Channels

- Neural networks often have multiple output channels, especially in deeper layers, to capture different features.

- Each output channel is calculated using its specific convolution kernel across all input channels.

7.4.3. 1x1 Convolutional Layer

- Although 1x1 convolutions may seem unintuitive, they are powerful for transforming channel dimensions without affecting spatial resolution.
- 1x1 convolutions act like fully connected layers applied across channels at every pixel. They reduce the computational complexity of deeper layers and help with dimensionality reduction while preserving information.

## 7.4 Exercises

1.

1. Given two convolution kernels $k_1$ and $k_2$, the first convolution with $k_1$ is applied to the input, followed by applying the second convolution with $k_2$ to the result. This operation is linear, and since convolutions are commutative and associative, the combination of two convolutional layers without a non-linearity in between can be expressed as a single convolution.

   If the input is $I$, then:

   $$y = (I * k_1) * k_2 = I * (k_1 * k_2)$$

   Thus, the result is a convolution with a single kernel formed by convolving $k_1$ and $k_2$.

2. The dimensionality of the equivalent single convolution kernel will be the sum of the dimensions of $k_1$ and $k_2$, minus 1. Specifically, if $k_1$ is of size $(m_1 \times n_1)$ and $k_2$ is of size $(m_2 \times n_2)$, the resulting single convolution kernel will have size $((m_1 + m_2 - 1) \times (n_1 + n_2 - 1))$.

3. The converse is not always true. Not every convolution can be decomposed into two smaller convolutions. The decomposition depends on the structure of the original convolution kernel. For example, a kernel with non-separable elements cannot be easily broken down into smaller convolutions.

2. 1. Given an input of shape $c_i \times h \times w$ and a convolution kernel of shape $c_o \times c_i \times k_h \times k_w$, padding $(p_h, p_w)$, and stride $(s_h, s_w)$, the output shape will be:

   $$\left(\frac{h - k_h + 2p_h}{s_h} + 1\right) \times \left(\frac{w - k_w + 2p_w}{s_w} + 1\right)$$

   For each output element, the number of multiplications is:

   $$c_i \times k_h \times k_w$$

   Thus, the total number of multiplications and additions for the entire forward pass is:

   $$c_o \times \left(\frac{h - k_h + 2p_h}{s_h} + 1\right) \times \left(\frac{w - k_w + 2p_w}{s_w} + 1\right) \times c_i \times k_h \times k_w$$

   2. The memory footprint is the amount of memory required to store the input, output, and kernel. This includes:

- Input memory: $c_i \times h \times w$
- Kernel memory: $c_o \times c_i \times k_h \times k_w$
- Output memory: $c_o \times \left(\frac{h - k_h + 2p_h}{s_h} + 1\right) \times \left(\frac{w - k_w + 2p_w}{s_w} + 1\right)$

   3. For the backward pass, additional intermediate gradients need to be stored for both the input and the kernel. This adds the memory requirements for:

- Gradient with respect to the input: same size as input $c_i \times h \times w$
- Gradient with respect to the kernel: same size as kernel $c_o \times c_i \times k_h \times k_w$
- Gradient with respect to the output: same size as output

Thus, the total memory footprint for the backward computation is:

$$2 \times (\text{input memory} + \text{kernel memory} + \text{output memory})$$

4. During backpropagation, the computational cost is similar to forward propagation but needs to compute gradients with respect to both the input and the kernel. The cost of computing the gradient with respect to the input and kernel involves a convolution with flipped kernels and additional sums, leading to the same number of operations as the forward pass.Thus, the total cost of backpropagation is approximately twice the forward pass cost:

$$2 \times c_o \times \left( \frac{h - k_h + 2p_h}{s_h} + 1 \right) \times \left( \frac{w - k_w + 2p_w}{s_w} + 1 \right) \times c_i \times k_h \times k_w$$

3. Doubling both the number of input channels $c_i$ and the number of output channels $c_o$ increases the number of computations by a factor of 4. This is because the number of computations depends linearly on both $c_i$ and $c_o$, and if both are doubled, the total number of operations quadruples. Doubling the padding does not affect the number of multiplications directly but increases the size of the output, leading to more operations.
4. $Y1$ and $Y2$ are exactly the same. In the final example, the convolution with a $1 \times 1$ kernel is mathematically equivalent to a matrix multiplication, and the implemented matrix multiplication produces the same result as the convolution operation. The assertion in the code confirms that the absolute difference between $Y1$ and $Y2$ is less than $1 \times 10^{-6}$, which is effectively zero.
5. 
6. The reason the algorithm that reads a $k + \Delta$-wide strip and computes a$\Delta$-wide output strip is preferable to reading a $k$-wide strip and computing a single output value at a time is efficiency. By reading a wider strip, it can reuse data across adjacent output pixels, reducing redundant memory accesses and computations. However, increasing $\Delta$ too much can lead to inefficiencies due to cache misses and increased memory bandwidth usage. Thus, the choice of $\Delta$ should balance reuse of data and memory limitations, often optimized for the specific hardware.
7. 
    1. Multiplying a block-diagonal matrix with a vector is faster because each block can be processed independently, leading to a reduction in the number of operations. If the matrix is split into $b$ blocks, the computational cost is reduced to $1/b$ of the original matrix multiplication cost, assuming ideal parallelism.
    2. The downside of having many blocks is that block-diagonal matrices lose expressivity. They can only capture interactions within blocks, not between blocks. One way to partially mitigate this is by introducing a small number of off-diagonal elements to allow limited interaction between blocks, or by using hierarchical matrices that combine block structure with some level of inter-block connectivity.

## 7.5 Discussion

7.5 Pooling

- Pooling layers aggregate information and reduce spatial resolution, making representations sensitive to the entire input.
- As the network deepens, the receptive field of each hidden node expands, and spatial resolution decreases, enhancing feature extraction and global understanding.

7.5.1 Maximum Pooling and Average Pooling

- Pooling involves a window that slides over the input, either taking the maximum or average of values within the window.
- Max-pooling tends to be more effective than average pooling in retaining essential features, while average pooling offers a smoother downsampling.

7.5.2 Padding and Stride

- Default frameworks often match the pooling window size with the stride, but manual adjustments allow for more control over the operation.
- Rectangular pooling windows and varied stride lengths are possible, offering flexibility in output resolution.

7.5.3 Multiple Channels

- Pooling layers handle multiple channels separately, maintaining the same number of channels after pooling.
- Each channel is independently pooled, ensuring the spatial structure of features is preserved while downsampling.

## 7.5 Exercises

1. Average pooling can be implemented using a convolution by employing a convolutional layer without biases, where the kernel size corresponds to the pooling window size, and all elements of the kernel are set to $\frac{1}{\text{pool size}}$.

```
import torch
import torch.nn as nn

# Implement average pooling via convolution
class AvgPoolConv(nn.Module):
    def __init__(self, pool_size):
        super(AvgPoolConv, self).__init__()
        self.pool_size = pool_size
        # Define the kernel to be the average
        self.kernel = torch.ones((1, 1, pool_size, pool_size)) / (pool_size * pool_size)

    def forward(self, x):
        return nn.functional.conv2d(x, self.kernel, stride=self.pool_size)

# Example
x = torch.tensor([[[[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]]]])
avg_pool_conv = AvgPoolConv(pool_size=2)
output = avg_pool_conv(x)
print(output)
```

2. Max-pooling selects the maximum value from a window, while convolution computes a weighted sum of the input elements, making the two operations fundamentally different. A convolution computes a linear combination of inputs, which cannot intrinsically select the maximum value. To compute the maximum, a non-linear function is necessary, which convolutional layers alone do not provide.
3. 1. The maximum of two values $a$ and $b$ can be expressed as:

$$\max(a, b) = \text{ReLU}(a - b) + b$$

This formula works because $\text{ReLU}(a - b)$ will be 0 if $b$ is greater than or equal to $a$, and will return $a - b$ otherwise, effectively selecting the larger value.

2. To implement 2x2 max-pooling using convolutions and ReLU operations:

- Use a convolutional layer to extract overlapping windows of size 2x2.
- Apply the formula for $\max(a, b)$ sequentially over each pair of values in the window.

3. For a 2x2 pooling window:

- 1 layer of convolution to extract the 4 values is needed.
- 3 layers of ReLU operations to compare the pairs are needed: $\max(a, b)$, $\max(c, d)$, and $\max(\max(a, b), \max(c, d))$.

For a 3x3 pooling window:

- 1 layer of convolution to extract the 9 values is needed.
- 8 ReLU comparisons to iteratively find the maximum of the 9 values are needed.

4. The computational cost of a pooling layer depends on the number of comparisons performed in each pooling window. For each window of size $p_h \times p_w$, the pooling operation requires $p_h \times p_w - 1$ comparisons to find the maximum or compute the average. Given an input of size $c \times h \times w$, the computational cost is:

$$\text{cost} = O(c \times \left\lceil \frac{h}{s_h} \right\rceil \times \left\lceil \frac{w}{s_w} \right\rceil \times (p_h \times p_w - 1))$$

Where:

- $c$ is the number of input channels,
- $s_h$ and $s_w$ are the stride sizes,
- $p_h$ and $p_w$ are the pooling window sizes.

5. Max-pooling captures the strongest feature in a window, while average pooling smoothens the values by taking the average. Max-pooling is better at preserving sharp features, while average pooling blurs the information, which might be desirable for some tasks.

6. A separate minimum pooling layer isn't strictly necessary, as it can be simulated by applying max-pooling to the negative of the input values:

$$\text{min-pooling}(x) = -\text{max-pooling}(-x)$$

Thus, minimum pooling can be replaced by applying max-pooling to the negated inputs.

7. Softmax is computationally more expensive than max or average pooling because it involves exponentiation and normalization. Additionally, softmax is typically used for probability distributions and might not provide the same level of translation invariance that max-pooling offers. The softmax function tends to highlight large values without completely suppressing smaller ones, which could dilute the pooling effect.

## 7.6 Discussion

7.6. Convolutional Neural Networks (LeNet)

- CNNs allow the retention of spatial structure in image data, unlike fully connected layers. Models become more efficient and require fewer parameters. LeNet, one of the first successful CNNs, was created by Yann LeCun to recognize handwritten digits.

7.6.1. LeNet Architecture

- LeNet is composed of a convolutional encoder with pooling and a dense block for final classification.
- It extracts features with a kernel and sigmoid activation, reduces spatial dimensionality via average pooling, and converts the 4D tensor output into a 2D tensor for the dense layers.

## 7.6 Exercises

1.

```
class LeNet_Modernized(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.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2),
        nn.LazyConv2d(16, kernel_size=5), nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2),
        nn.Flatten(),
        nn.LazyLinear(120), nn.ReLU(),
        nn.LazyLinear(84), nn.ReLU(),
        nn.LazyLinear(num_classes))
```

2.

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

def init_cnn(module):
    if isinstance(module, (nn.Linear, nn.Conv2d)):
        nn.init.xavier_uniform_(module.weight)

class LeNet_Modernized(d2l.Classifier):
    def __init__(self, lr=0.05, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.LazyConv2d(32, kernel_size=3, padding=1), nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.LazyConv2d(64, kernel_size=3, padding=1), nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.LazyConv2d(128, kernel_size=3, padding=1), nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Flatten(),
            nn.LazyLinear(256), nn.ReLU(),
            nn.LazyLinear(128), nn.ReLU(),
            nn.LazyLinear(num_classes))

        self.apply(init_cnn)
```

```
trainer = d2l.Trainer(max_epochs=20, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128)
model = LeNet_Modernized(lr=0.05)
model.apply_init([next(iter(data.get_dataloader(True)))[0]], init_cnn)
trainer.fit(model, data)
```

3.

```
trainer = d2l.Trainer(max_epochs=20, num_gpus=1)
data = d2l.MNIST(batch_size=128)
model = LeNet_Modernized(lr=0.05)
model.apply_init([next(iter(data.get_dataloader(True)))[0]], init_cnn)
trainer.fit(model, data)
```

4.

5.

## 8.2 Discussion

8.2 Networks Using Blocks (VGG)

- Introduced by the Visual Geometry Group (VGG) at Oxford University in 2014. It aims to find if deep or wide networks are better for performance.
- Pioneered the concept of building neural networks using blocks.

8.2.1 VGG Blocks

- Made up of basic building block with multiple convolutional layers followed by max-pooling for downsampling.
- Uses small 3x3 convolutional filters with padding to maintain spatial resolution.
- Stacking small convolutional filters provides better performance than using larger filters.
- Max-pooling halves the height and width of the image after each block.

8.2.2 VGG Network

- Like AlexNet and LeNet, VGG can be divided into two parts: convolutional layers and fully connected layers.
- The convolutional layers are grouped into blocks of multiple convolutions, followed by downsampling.
- VGG-11 is composed of five convolutional blocks, with the number of output channels doubling after each block. The fully connected part has three layers, making it similar to AlexNet.
- VGG introduced the idea of network families, where different architectures offer varying trade-offs between speed and accuracy.

## 8.2 Exercises

1. 1. AlexNet consists of five convolutional layers and three fully connected layers having a total number of parameters of around 60 million. Meanwhile VGG uses smaller convolutional kernels 3x3 but stacks more layers. VGG-16 has 13 convolutional layers and 3 fully connected layers with a total number of parameters of approximately 138 million.
   2. For convolutional layers:
   - VGG has more convolutional layers than AlexNet. The computational cost of convolutional layers is proportional to the number of filters, filter size, and input feature map size.
   - In AlexNet, the convolutional layers contribute to fewer FLOPs due to fewer layers and larger kernels.
   - VGG uses many more smaller 3x3 kernels, which means a higher number of FLOPs in its convolutional layers compared to AlexNet.

     For fully connected layers:
   - In AlexNet, the fully connected layers contribute significantly to the total FLOPs because the FC layers connect to a large flattened feature map.
   - While VGG also has large fully connected layers, the contribution of FLOPs here is still significant, but the convolutional layers dominate overall.

   3. To reduce the computational cost of the fully connected layers:
   - Decrease the input image resolution or increase the stride of convolutional and pooling layers to downsample more aggressively.
   - Instead of 4096 units in each fully connected layer, you can reduce the number of units to something smaller .
   - Instead of flattening the feature map, you can apply global average pooling before the fully connected layer. This approach reduces the number of inputs to the fully connected layers significantly.

## 8.2 Exercises

1. 1. AlexNet consists of five convolutional layers and three fully connected layers having a total number of parameters of around 60 million. Meanwhile VGG uses smaller convolutional kernels 3x3 but stacks more layers. VGG-16 has 13 convolutional layers and 3 fully connected layers with a total number of parameters of approximately 138 million.

   2. For convolutional layers:

   - VGG has more convolutional layers than AlexNet. The computational cost of convolutional layers is proportional to the number of filters, filter size, and input feature map size.
   - In AlexNet, the convolutional layers contribute to fewer FLOPs due to fewer layers and larger kernels.
   - VGG uses many more smaller 3x3 kernels, which means a higher number of FLOPs in its convolutional layers compared to AlexNet.

     For fully connected layers:

   - In AlexNet, the fully connected layers contribute significantly to the total FLOPs because the FC layers connect to a large flattened feature map.
   - While VGG also has large fully connected layers, the contribution of FLOPs here is still significant, but the convolutional layers dominate overall.

   3. To reduce the computational cost of the fully connected layers:

   - Decrease the input image resolution or increase the stride of convolutional and pooling layers to downsample more aggressively.
   - Instead of 4096 units in each fully connected layer, you can reduce the number of units to something smaller .
   - Instead of flattening the feature map, you can apply global average pooling before the fully connected layer. This approach reduces the number of inputs to the fully connected layers significantly.

2. The remaining three layers are part of the fully connected layers. The architecture of VGG, specifically VGG-11, contains:

- 8 convolutional layers (organized into 5 blocks),
- 3 fully connected layers at the end.

  Convolutional blocks consist of convolution layers followed by max pooling. The fully connected layers are not part of these blocks but are still counted in the total number of layers.

3.

- VGG-16:

    - The architecture consists of 13 convolutional layers and 3 fully connected layers.
    - Configuration:
        - (Conv, 64), (Conv, 64)
        - (Conv, 128), (Conv, 128)
        - (Conv, 256), (Conv, 256), (Conv, 256)
        - (Conv, 512), (Conv, 512), (Conv, 512)
        - (Conv, 512), (Conv, 512), (Conv, 512)
        - (FC, 4096), (FC, 4096), (FC, 10)
- VGG-19:
    - The architecture consists of 16 convolutional layers and 3 fully connected layers.
    - Configuration:
        - (Conv, 64), (Conv, 64)
        - (Conv, 128), (Conv, 128)
        - (Conv, 256), (Conv, 256), (Conv, 256), (Conv, 256)
        - (Conv, 512), (Conv, 512), (Conv, 512), (Conv, 512)
        - (Conv, 512), (Conv, 512), (Conv, 512), (Conv, 512)
        - (FC, 4096), (FC, 4096), (FC, 10)

4.

```
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=(56, 56))
model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
trainer.fit(model, data)
```

## 8.6 Discussion

8.6.1. Function Classes

- Regularization helps control model complexity and improve consistency, especially with more training data. The goal is to find a model close to the "truth" function.
- For deep networks to improve performance, newly added layers must be trained to either enhance model performance or be as effective as existing layers.
- He et al. (2016) introduced residual networks (ResNet), allowing deeper networks by ensuring additional layers can default to identity functions. This improves model training without worsening performance.

8.6.2. Residual Blocks

- A regular block learns a direct mapping, while a residual block learns the residual between input and output, simplifying the learning process.

- Residual connections allow faster propagation of inputs through layers, bypassing complex transformations, and making deep networks more trainable.
- ResNet uses two 3x3 convolutional layers, each followed by batch normalization and ReLU, and adds the input to the final output before the ReLU activation. For mismatched input-output dimensions, a 1x1 convolution is used.

8.6.3. ResNet Model

- The first two layers of ResNet are similar to GoogLeNet but with added batch normalization layers. It uses residual blocks instead of inception modules.
- The first block in each module reduces the height and width while doubling the number of channels, making ResNet highly scalable.