Deep Learning Homework 2

7.1 From Fully Connected Layers to Convolutions

In [ ]:
```
# CNN is a way to make networks more efficient in discovering patterns and structures within the image to class
# parameters looked at. they look at local or nearby pixels to find patterns withing the image. they reduce imag
# the most useful structures in the image.
```

7.2 Convolutions for Images

In [2]:
```
import torch
from torch import nn
from d2l import torch as d2l
```

```
/Users/michaelvanhuet/Desktop/Computer Science/2024 T2 KU/Deep Learning/Coding/pytorch/lib/python3.9/site-packag
es/urllib3/__init__.py:35: NotOpenSSLWarning: urllib3 v2 only supports OpenSSL 1.1.1+, currently the 'ssl' modul
e is compiled with 'LibreSSL 2.8.3'. See: https://github.com/urllib3/urllib3/issues/3020
  warnings.warn(
```

In [4]:
```
# 7.2.1

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

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

Out[4]:
```
tensor([[19., 25.],
        [37., 43.]])
```

In [5]:
```
#7.2.2

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 [6]:
```
# 7.2.3

X = torch.ones((6, 8))
X[:, 2:6] = 0
X
```

Out[6]:
```
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.]])
```

In [7]:
```
K = torch.tensor([[1.0, -1.0]])

Y = corr2d(X, K)
Y
```

Out[7]:
```
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.]])
```

In [8]:
```
corr2d(X.t(), K)
```

Out[8]:
```
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.]])
```

In [9]:
```
# 7.2.4

# 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 12.593
epoch 4, loss 2.194
epoch 6, loss 0.401
epoch 8, loss 0.081
epoch 10, loss 0.019
```

In [10]:
```
conv2d.weight.data.reshape((1, 2))
```

Out[10]:
```
tensor([[ 0.9718, -0.9918]])
```

In [12]:
```
#Summary
# Using CNN makes calculations relatively uncomplicated and fast for more efficient networks.
```

7.3 Padding and Stride

In [13]:
```
# 7.3.1
# 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
```

Out[13]:
```
torch.Size([8, 8])
```

In [14]:
```
# 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
```

Out[14]:
```
torch.Size([8, 8])
```

In [15]:
```
# 7.3.2

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

Out[15]:
```
torch.Size([4, 4])
```

In [16]:
```
conv2d = nn.LazyConv2d(1, kernel_size=(3, 5), padding=(0, 1), stride=(3, 4))
```

```
comp_conv2d(conv2d, X).shape
```

Out[16]:
```
torch.Size([2, 2])
```

In [17]:
```
# Summary
# the use of padding helps prevent data loss at the edge of input images and ensures all information pixels are
# usually padding of 0 values are used. using odd sized kernels such as 3x3 or 5x5 are the best. stride is how r
# convolution window moves, greater strides result in downsizing of the output tensor
```

7.4 Multiple Input and Output Channels

In [18]:
```
# 7.4

# 7.4.1

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)
```

Out[18]:
```
tensor([[ 56.,  72.],
        [104., 120.]])
```

In [19]:
```
# 7.4.2

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)
```

In [20]:
```
K = torch.stack((K, K + 1, K + 2), 0)
K.shape
```

Out[20]:
```
torch.Size([3, 2, 2, 2])
```

In [21]:
```
corr2d_multi_in_out(X, K)
```

Out[21]:
```
tensor([[[ 56.,  72.],
         [104., 120.]],

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

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

In [26]:
```
# 7.4.3

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))
```

In [27]:
```
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
```

In [28]:
```
# summary
# using multiple dimensions allows the CNN to analyse features simultaneously
```

7.5 Pooling

In [29]:
```
# 7.5.1

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))
```

```python
    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))
```

Out[29]:
```
tensor([[4., 5.],
        [7., 8.]])
```

In [30]:
```python
pool2d(X, (2, 2), 'avg')
```

Out[30]:
```
tensor([[2., 3.],
        [5., 6.]])
```

In [31]:
```python
# 7.5.2

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

Out[31]:
```
tensor([[[[ 0.,  1.,  2.,  3.],
          [ 4.,  5.,  6.,  7.],
          [ 8.,  9., 10., 11.],
          [12., 13., 14., 15.]]]])
```

In [32]:
```python
pool2d = nn.MaxPool2d(3)
# Pooling has no model parameters, hence it needs no initialization
pool2d(X)
```

Out[32]:
```
tensor([[[[10.]]]])
```

In [33]:
```python
pool2d = nn.MaxPool2d(3, padding=1, stride=2)
pool2d(X)
```

Out[33]:
```
tensor([[[[ 5.,  7.],
          [13., 15.]]]])
```

In [34]:
```python
pool2d = nn.MaxPool2d((2, 3), stride=(2, 3), padding=(0, 1))
pool2d(X)
```

Out[34]:
```
tensor([[[[ 5.,  7.],
          [13., 15.]]]])
```

In [35]:
```python
#7.5.3

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

Out[35]:
```
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.]]]])
```

In [36]:
```python
pool2d = nn.MaxPool2d(3, padding=1, stride=2)
pool2d(X)
```

Out[36]:
```
tensor([[[[ 5.,  7.],
          [13., 15.]],

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

In [ ]:
```python
# Summary
# poolings helps mitigate the sensitivity of translation of pixels. max pooling is good because it retains valu
```

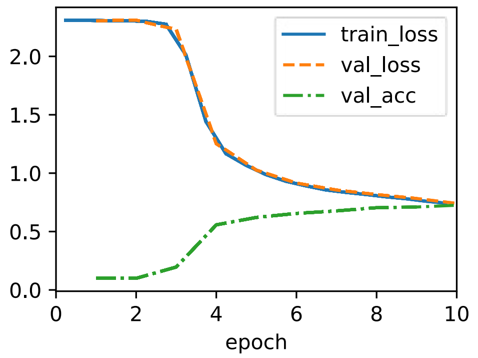
7.6 Convolutional Neural Networks (LeNet)

In [37]:
```python
# 7.6.1

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

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


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

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

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

In [38]:
```
# 7.6.2

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)
```



In [ ]:
```
# Summary
# by incorperating varius CNN techniques into the LeNet-5, an accuracy
# that previously could not be achieved is reached. this structure
# finds the most important patterns and sharp edges with the convolutional
# layers and then makes it smaller using pooling to find only the most important
# parts. this way only the most important aspects of the image is analysed
# which makes predictions faster and more accurate.
```

8.2 Networks Using Blocks (VGG)

In [39]:
```
# 8.2.1

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)
```

```python
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])
```

In [42]:

```python
# 8.2.3
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)
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In[42], line 6
      4 data = d2l.FashionMNIST(batch_size=128, resize=(224, 224))
      5 model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
----> 6 trainer.fit(model, data)

File ~/Desktop/Computer Science/2024 T2 KU/Deep Learning/Coding/pytorch/lib/python3.9/site-packages/d2l/torch.py
:285, in Trainer.fit(self, model, data)
    283 self.val_batch_idx = 0
    284 for self.epoch in range(self.max_epochs):
--> 285     self.fit_epoch()

File ~/Desktop/Computer Science/2024 T2 KU/Deep Learning/Coding/pytorch/lib/python3.9/site-packages/d2l/torch.py
:301, in Trainer.fit_epoch(self)
    299 self.optim.zero_grad()
    300 with torch.no_grad():
--> 301     loss.backward()
    302     if self.gradient_clip_val > 0:  # To be discussed later
    303         self.clip_gradients(self.gradient_clip_val, self.model)

File ~/Desktop/Computer Science/2024 T2 KU/Deep Learning/Coding/pytorch/lib/python3.9/site-packages/torch/_tenso
r.py:521, in Tensor.backward(self, gradient, retain_graph, create_graph, inputs)
    511 if has_torch_function_unary(self):
    512     return handle_torch_function(
    513         Tensor.backward,
    514         (self,),
   (...)
    519         inputs=inputs,
    520     )
--> 521 torch.autograd.backward(
    522     self, gradient, retain_graph, create_graph, inputs=inputs
    523 )

File ~/Desktop/Computer Science/2024 T2 KU/Deep Learning/Coding/pytorch/lib/python3.9/site-packages/torch/autogr
ad/__init__.py:289, in backward(tensors, grad_tensors, retain_graph, create_graph, grad_variables, inputs)
    284     retain_graph = create_graph
    286 # The reason we repeat the same comment below is that
    287 # some Python versions print out the first line of a multi-line function
    288 # calls in the traceback and some print out the last line
--> 289 _engine_run_backward(
    290     tensors,
    291     grad_tensors_,
    292     retain_graph,
    293     create_graph,
    294     inputs,
    295     allow_unreachable=True,
    296     accumulate_grad=True,
    297 )

File ~/Desktop/Computer Science/2024 T2 KU/Deep Learning/Coding/pytorch/lib/python3.9/site-packages/torch/autogr
ad/graph.py:769, in _engine_run_backward(t_outputs, *args, **kwargs)
    767     unregister_hooks = _register_logging_hooks_on_whole_graph(t_outputs)
    768 try:
--> 769     return Variable._execution_engine.run_backward(  # Calls into the C++ engine to run the backward pas
s
    770         t_outputs, *args, **kwargs
    771     )  # Calls into the C++ engine to run the backward pass
    772 finally:
    773     if attach_logging_hooks:

KeyboardInterrupt: 
```

In [ ]:

```
# Summary
# VGG emphasises the use of repeated convolutional blocks and shows that deep
# networks generally perform much better than shallow ones.
```

8.6 Residual Networks (ResNet) and ResNeXt

In [43]:

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

In [44]:

```
class Residual(nn.Module):  #@save
    """The Residual block of ResNet models."""
    def __init__(self, num_channels, use_1x1conv=False, strides=1):
        super().__init__()
        self.conv1 = nn.LazyConv2d(num_channels, kernel_size=3, padding=1,
                                   stride=strides)
        self.conv2 = nn.LazyConv2d(num_channels, kernel_size=3, padding=1)
```

```python
        if use_1x1conv:
            self.conv3 = nn.LazyConv2d(num_channels, kernel_size=1,
                                       stride=strides)
        else:
            self.conv3 = None
        self.bn1 = nn.LazyBatchNorm2d()
        self.bn2 = nn.LazyBatchNorm2d()

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

In [45]:
```python
blk = Residual(3)
X = torch.randn(4, 3, 6, 6)
blk(X).shape
```

Out[45]: torch.Size([4, 3, 6, 6])

In [46]:
```python
blk = Residual(6, use_1x1conv=True, strides=2)
blk(X).shape
```

Out[46]: torch.Size([4, 6, 3, 3])

In [47]:
```python
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))
```

In [48]:
```python
@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)
```

In [49]:
```python
@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)
```

In [50]:
```python
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])
```

In [51]:
```python
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)
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In[51], line 5
      3 data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
      4 model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
----> 5 trainer.fit(model, data)

File ~/Desktop/Computer Science/2024 T2 KU/Deep Learning/Coding/pytorch/lib/python3.9/site-packages/d2l/torch.py
:285, in Trainer.fit(self, model, data)
    283 self.val_batch_idx = 0
    284 for self.epoch in range(self.max_epochs):
--> 285     self.fit_epoch()

File ~/Desktop/Computer Science/2024 T2 KU/Deep Learning/Coding/pytorch/lib/python3.9/site-packages/d2l/torch.py
:301, in Trainer.fit_epoch(self)
    299 self.optim.zero_grad()
    300 with torch.no_grad():
--> 301     loss.backward()
    302     if self.gradient_clip_val > 0:  # To be discussed later
    303         self.clip_gradients(self.gradient_clip_val, self.model)

File ~/Desktop/Computer Science/2024 T2 KU/Deep Learning/Coding/pytorch/lib/python3.9/site-packages/torch/_tenso
r.py:521, in Tensor.backward(self, gradient, retain_graph, create_graph, inputs)
    511 if has_torch_function_unary(self):
    512     return handle_torch_function(
    513         Tensor.backward,
    514         (self,),
   (...)
    519         inputs=inputs,
    520     )
--> 521 torch.autograd.backward(
    522     self, gradient, retain_graph, create_graph, inputs=inputs
    523 )

File ~/Desktop/Computer Science/2024 T2 KU/Deep Learning/Coding/pytorch/lib/python3.9/site-packages/torch/autogr
ad/__init__.py:289, in backward(tensors, grad_tensors, retain_graph, create_graph, grad_variables, inputs)
    284     retain_graph = create_graph
    286 # The reason we repeat the same comment below is that
    287 # some Python versions print out the first line of a multi-line function
    288 # calls in the traceback and some print out the last line
--> 289 _engine_run_backward(
    290     tensors,
    291     grad_tensors_,
    292     retain_graph,
    293     create_graph,
    294     inputs,
    295     allow_unreachable=True,
    296     accumulate_grad=True,
    297 )

File ~/Desktop/Computer Science/2024 T2 KU/Deep Learning/Coding/pytorch/lib/python3.9/site-packages/torch/autogr
ad/graph.py:769, in _engine_run_backward(t_outputs, *args, **kwargs)
    767     unregister_hooks = _register_logging_hooks_on_whole_graph(t_outputs)
    768 try:
--> 769     return Variable._execution_engine.run_backward(  # Calls into the C++ engine to run the backward pas
s
    770         t_outputs, *args, **kwargs
    771     )  # Calls into the C++ engine to run the backward pass
    772 finally:
    773     if attach_logging_hooks:

KeyboardInterrupt: 
```

In [ ]:

```
# summary
# ResNet introduces the concept of residual learning through skip connections, making it easier to train very d
# the learning of identity mappings. This architecture improves gradient flow, enhances expressiveness, and sim
# deep models, leading to robust performance across various applications. Its modular design and scalability ha
# architecture in deep learning
```