Lecture 5: Deep Learning Computation, Convolutional Neural Networks-1

Reading: Chapter 5.1-5.2, 6.1-6.3 of *Dive Into Deep Learning*

Outline

- Deep Learning Computation: Model Construction and Parameter Management
- Convolutional Neural Networks: Convolution, Padding and Stride

Model Construction: Layers and Blocks

- **A single neuron**
    - takes some set of inputs;
    - generates a corresponding scalar output with a linear model and a activation function; and
    - has a set of associated parameters that can be learned to optimize some objective function of interest.
- **A layer**
    - Multiple neurons taking the same set of inputs;

- **MLP**
  - Multiple layers stacked together whethe the inputs of a layer is the outputs of its preceding layer
- **A block**
  - larger than an individual layer but smaller than the entire model.
  - Can be a single layer, a component consisting of multiple layers, or the entire model itself.
  - Multiple layers are combined into blocks, forming repeating patterns of larger models.
  - Benefit: compact code to implement complex neural networks.

Multiple layers are combined into blocks, forming repeating patterns of larger models.

- From a programing standpoint, a block is represented by a class.
  - Any subclass of it must define a forward propagation function that transforms its input into output and must store any necessary parameters.
  - Finally a block must possess a backpropagation function, for purposes of calculating gradients.

The following code generates a network with one fully-connected hidden layer with 256 units and ReLU activation, followed by a fully-connected output layer with 10 units (no activation function).

In [1]:

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

net = nn.Sequential(nn.Linear(20, 256), nn.ReLU(), nn.Linear(256, 10))
X = torch.rand(2, 20)
net(X)
```

Out[1]:

```
tensor([[ 0.0864, -0.0297,  0.0581,  0.0375, -0.1
816,  0.1452, -0.0222, -0.1160,
         -0.1659, -0.2437],
        [ 0.1028, -0.0994,  0.1481,  0.1704, -0.1
635,  0.1939, -0.0030,  0.0426,
         -0.1631, -0.2410]], grad_fn=<AddmmBackwa
rd0>)
```

Basic functionality of a block

1. Ingest input data as arguments to its forward propagation function.
2. Generate an output by having the forward propagation function return a value.
3. Calculate the gradient of its output with respect to its input, which can be accessed via its backpropagation function.
4. Store and provide access to those parameters necessary to execute the forward propagation computation.
5. Initialize model parameters as needed.

Summary for Model Construction

- Layers are blocks.
- Many layers can comprise a block.
- Many blocks can comprise a block.
- A block can contain code.
- Blocks take care of lots of housekeeping, including parameter initialization and backpropagation.
- Sequential concatenations of layers and blocks are handled by the `Sequential` block.

Parameter Management

- Accessing parameters for debugging, diagnostics, and visualizations.
- Parameter initialization.
- Sharing parameters across different model components.

In [2]:

```
import torch
from torch import nn

net = nn.Sequential(nn.Linear(4, 8), nn.ReLU(), nn.Linear(8, 1))
X = torch.rand(size=(2, 4))
net(X)
```

Out[2]:

```
tensor([[0.0048],
        [0.0381]], grad_fn=<AddmmBackward0>)
```

Parameter Access

- Access any layer by indexing into the model as though it were a list.
- Inspect the parameters of the second fully-connected layer as follows.

In [3]:

```
print(net[2].state_dict())
```

```
OrderedDict([('weight', tensor([[0.0865, 0.0056,
0.2186, 0.3380, 0.0305, 0.2811, 0.0094, 0.342
4]])), ('bias', tensor([-0.1197]))])
```

Targeted Parameters

- Extracts the bias from the second neural network layer and further accesses that parameter's value.

In [4]:

```
print(type(net[2].bias))
print(net[2].bias)
print(net[2].bias.data)
```

```
<class 'torch.nn.parameter.Parameter'>
Parameter containing:
tensor([-0.1197], requires_grad=True)
tensor([-0.1197])
```

- Access the gradient.

In [5]:

```
net[2].weight.grad == None
```

Out[5]:

```
True
```

All Parameters at Once

- Accessing the parameters of the first fully-connected layer vs. accessing all layers.

In [6]:

```
print(*[(name, param.shape) for name, param in net[0].named_parameters()])
print(*[(name, param.shape) for name, param in net.named_parameters()])
```

```
('weight', torch.Size([8, 4])) ('bias', torch.Size([8]))
('0.weight', torch.Size([8, 4])) ('0.bias', torch.Size([8])) ('2.weight', torch.Size([1, 8])) ('2.bias', torch.Size([1]))
```

- Accessing the parameters of the network

In [7]:

```
net.state_dict()['2.weight'].data
```

Out[7]:

```
tensor([[0.0865, 0.0056, 0.2186, 0.3380, 0.0305,
0.2811, 0.0094, 0.3424]])
```

Collecting Parameters from Nested Blocks

In [8]:

```
def block1():
    return nn.Sequential(nn.Linear(4, 8), nn.ReLU(),
                         nn.Linear(8, 4), nn.ReLU())

def block2():
    net = nn.Sequential()
    for i in range(4):
        # Nested here
        net.add_module(f'block {i}', block1())
    return net

rgnet = nn.Sequential(block2(), nn.Linear(4, 1))
rgnet(X)
```

Out[8]:

```
tensor([[0.0460],
        [0.0460]], grad_fn=<AddmmBackward0>)
```

In [9]:

```
print(rgnet)
```

```
Sequential(
  (0): Sequential(
    (block 0): Sequential(
      (0): Linear(in_features=4, out_features=8,
bias=True)
      (1): ReLU()
      (2): Linear(in_features=8, out_features=4,
bias=True)
      (3): ReLU()
    )
    (block 1): Sequential(
      (0): Linear(in_features=4, out_features=8,
bias=True)
      (1): ReLU()
      (2): Linear(in_features=8, out_features=4,
```

```
bias=True)
      (3): ReLU()
    )
    (block 2): Sequential(
      (0): Linear(in_features=4, out_features=8,
bias=True)
      (1): ReLU()
      (2): Linear(in_features=8, out_features=4,
bias=True)
      (3): ReLU()
    )
    (block 3): Sequential(
      (0): Linear(in_features=4, out_features=8,
bias=True)
      (1): ReLU()
      (2): Linear(in_features=8, out_features=4,
bias=True)
      (3): ReLU()
    )
```

```
  )
  (1): Linear(in_features=4, out_features=1, bias
=True)
)
```

- Access the first major block,
- within it the second sub-block, and
- within that the bias of the first layer.

In [10]:

```
rgnet[0][1][0].bias.data
```

Out[10]:

```
tensor([-0.1604, -0.4100, -0.1052, -0.0569, -0.1441, -0.3547, -0.4200,  0.2776])
```

Parameter Initialization

- PyTorch's `nn.init` module provides a variety of preset initialization methods.

Built-in Initialization

- initializes all weight parameters as Gaussian random variables
- with standard deviation 0.01,
- while bias parameters cleared to zero.

In [11]:

```python
def init_normal(m):
    if type(m) == nn.Linear:
        nn.init.normal_(m.weight, mean=0, std=0.01)
        nn.init.zeros_(m.bias)
net.apply(init_normal)
net[0].weight.data[0], net[0].bias.data[0]
```

Out[11]:

```
(tensor([ 0.0036,  0.0096, -0.0100, -0.0133]), te
nsor(0.))
```

- initialize all the parameters to a given constant value (say, 1).

In [12]:

```
def init_constant(m):
    if type(m) == nn.Linear:
        nn.init.constant_(m.weight, 1)
        nn.init.zeros_(m.bias)
net.apply(init_constant)
net[0].weight.data[0], net[0].bias.data[0]
```

Out[12]:

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

- Apply different initializers for certain blocks.

In [13]:

```
def xavier(m):
    if type(m) == nn.Linear:
        nn.init.xavier_uniform_(m.weight)
def init_42(m):
    if type(m) == nn.Linear:
        nn.init.constant_(m.weight, 42)

net[0].apply(xavier)
net[2].apply(init_42)
print(net[0].weight.data[0])
print(net[2].weight.data)
```

```
tensor([-0.0153, -0.4413, -0.2382,  0.4730])
tensor([[42., 42., 42., 42., 42., 42., 42., 4
2.]])
```

Tied Parameters (Shared weights)

- Share weights across multiple layers.
- The gradients of the shared layers are added together during backpropagation.

In [14]:

```
# We need to give the shared layer a name so that we can refer to its
# parameters
shared = nn.Linear(8, 8)
net = nn.Sequential(nn.Linear(4, 8), nn.ReLU(),
                    shared, nn.ReLU(),
                    shared, nn.ReLU(),
                    nn.Linear(8, 1))
net(X)
# Check whether the parameters are the same
print(net[2].weight.data[0] == net[4].weight.data[0])
net[2].weight.data[0, 0] = 100
# Make sure that they are actually the same object rather than just having the
# same value
print(net[2].weight.data[0] == net[4].weight.data[0])
```

```
tensor([True, True, True, True, True, True, True,
True])
```

```
tensor([True, True, True, True, True, True, True,
True])
```

Convolutional Neural Networks

- Some data (e.g. images) has special structures (e.g. two-dimensional grid of pixels).
- MLP discarded each image's spatial structure by flattening them into one-dimensional vectors
- MLPs networks are invariant to the order of the features.
- Convolutional neural networks (CNNs), a powerful family of neural networks to overcome this limitation.
- We will walk through the basic operations of CNNs.

From Fully-Connected Layers to Convolutions

- Tabular data: consist of rows corresponding to examples and columns corresponding to features.
    - Patterns: interactions among the features,
    - No assumption on the structure on how the features interact.
    - Lack knowledge to guide the construction of craftier architectures.
    - An MLP may be the best that we can do.
- CNN exploits some of the known structure in natural images.

Invariance

- Image classification
    - Varying location of the object in the image.
    - Object is just part of the image.
- Guidelines for the design of a neural network architecture suitable for computer vision:
    - *translation invariance*
    - *locality* principle: focus on local regions

Convolutions

- Constraining MLP
- Same linear transformation applied on every patch (of the image) with the same size of the convolution kernel.

Convolutions for Images

The Cross-Correlation Operation

Two-dimensional cross-correlation operation. The shaded portions are the first output element as well as the input and kernel tensor elements used for the output computation: $0\times0+1\times1+3\times2+4\times3=19$.

- Slide the convolution window across the input tensor, both from left to right and top to bottom.
- The input subtensor contained in that window and the kernel tensor are multiplied elementwise
- The resulting tensor is summed up yielding a single scalar value.

Output size

- Input size $n_h \times n_w$
- Convolution kernel $k_h \times k_w$
- The outsize is then via

$$(n_h-k_h+1) \times (n_w-k_w+1).$$

In [15]:

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

In [16]:

```
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
```

In [17]:

```
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[17]:

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

Convolutional Layers

- A convolutional layer cross-correlates the input and kernel and adds a scalar bias to produce an output.
- Parameters: the kernel and the scalar bias.
- Notation: $h \times w$ convolutional layer.

In [18]:

```
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
```

Object Edge Detection in Images

In [19]:

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

Out[19]:

```
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.]])
```

- Construct a kernel `K` ($[1,-1]$ with a height of 1 and a width of 2.
- When we perform the cross-correlation operation with the input,

In [20]:

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

In [21]:

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

Out[21]:

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

- Apply the kernel to the transposed image.

In [22]:

```
corr2d(X.t(), K)
```

Out[22]:

```
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.]])
```

Learning Kernels

- The kernels in CNNs are learnt similarly as the weights of MLPS.

Cross-Correlation and Convolution

- Convolution operations are usually implemented with cross-correlation
- For strict *convolution* operation,
    - flip the two-dimensional kernel tensor both horizontally and vertically, and
    - perform the *cross-correlation* operation with the input tensor.

Feature Map and Receptive Field

- *feature map*: the output of convolutional layer
    - learned representations (features)
    - *receptive field* of any element $x$of some layer: all the elements (from all the previous layers) that may affect the calculation of x during the forward propagation.
    - The *receptive field* of the elements in the late layers is getting larger than than those of the preceding layers.

Summary of Convolution

- The core computation of a two-dimensional convolutional layer is a two-dimensional cross-correlation operation.
- We can design a kernel to detect edges in images.
- We can learn the kernel's parameters from data.
- When any element in a feature map needs a larger receptive field to detect broader features on the input, a deeper network can be considered.

Padding and Stride

Padding

- Pad a 3×3 input with zeros increasing its size to 5×5.
- The corresponding output then increases to a 4×4 matrix.

Two-dimensional cross-correlation with padding.

- Add a total of p_h rows of padding (roughly half on top and half on bottom) and
- a total of p_w columns of padding (roughly half on the left and half on the right),
- the output shape will be

$$(n_h-k_h+p_h+1)\times(n_w-k_w+p_w+1).$$

In [23]:

```python
import torch
from torch import nn


# We define a convenience function to calculate the convolutional layer. This
# function initializes the convolutional layer weights and performs
# corresponding dimensionality elevations and reductions on the input and
# output
def comp_conv2d(conv2d, X):
    # Here (1, 1) indicates that the batch size and the number of channels
    # are both 1
    X = X.reshape((1, 1) + X.shape)
    Y = conv2d(X)
    # Exclude the first two dimensions that do not interest us: examples and
    # channels
    return Y.reshape(Y.shape[2:])
# Note that here 1 row or column is padded on either side, so a total of 2
# rows or columns are added
conv2d = nn.Conv2d(1, 1, kernel_size=3, padding=1)
X = torch.rand(size=(8, 8))
comp_conv2d(conv2d, X).shape
```

Out[23]:

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

In [24]:

```python
# Here, we use a convolution kernel with a height of 5 and a width of 3. The
# padding numbers on either side of the height and width are 2 and 1,
# respectively
conv2d = nn.Conv2d(1, 1, kernel_size=(5, 3), padding=(2, 1))
comp_conv2d(conv2d, X).shape
```

Out[24]:

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

Stride

- *Stride*: the number of rows and columns traversed per slide

Cross-correlation with strides of 3 and 2 for height and width, respectively.

In [25]:

```
conv2d = nn.Conv2d(1, 1, kernel_size=3, padding=1, stride=2)
comp_conv2d(conv2d, X).shape
```

Out[25]:

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

Next, we will look at (**a slightly more complicated example**).

In [26]:

```
conv2d = nn.Conv2d(1, 1, kernel_size=(3, 5), padding=(0, 1), stride=(3, 4))
comp_conv2d(conv2d, X).shape
```

Out[26]:

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

Summary of Padding and Stride

- Padding can increase the height and width of the output. This is often used to give the output the same height and width as the input.
- The stride can reduce the resolution of the output, for example reducing the height and width of the output to only $1/n$ of the height and width of the input (n is an integer greater than 1).
- Padding and stride can be used to adjust the dimensionality of the data effectively.

In [ ]:

```

```