第 2 章

PyTorch 基础编程

本章首先介绍 PyTorch 的基础编程，主要包括 PyTorch 张量操作和自动求导功能。然后介绍如何使用 PyTorch 的 Dataset 类和 DataLoader 类来读取数据集文件；由于大部分数据集都是以文件形式存放，熟悉并掌握各种格式的文件读写操作是 PyTorch 编程必备的常规技能。最后介绍 torchvision 和 torchtext 工具示例，以便能够快速上手计算机视觉和自然语言处理的工作。

2.1 张量数据操作

PyTorch 所使用的数据结构是基于计算图和基于张量的，因此理解张量的定义和基本操作非常重要。

2.1.1 张量简介

Tensor(张量)是 PyTorch 的核心数据结构。张量的维度(dimension)也称为阶，注意这里的阶与矩阵的阶无关。零阶张量称为标量(scalar)，1 阶张量称为向量(vector)，2 阶张量称为矩阵(matrix)，3 阶以上的就直接称为张量。

Tensor 类的 shape(形状)属性和 size(大小)函数返回张量的具体维度分量。PyTorch 中很多与张量相关的函数都有 dim 控制参数，指定对张量的哪一个维度进行操作。

(1) 标量 Tensor 的维度为 0，一般用作超参数、参数和损失函数。

(2) 向量 Tensor 的维度为 1，一般用作神经网络的偏置，以及神经网络各层的输入。要注意的是，虽然向量只有一个维度，但这个维度可以有多个分量，就像一维数组可以有多个元素一样，不要混淆维度和分量的概念。

(3) 矩阵 Tensor 的维度为 2，一般用作神经网络的批量输入和输出，以及线性神经元的权重参数。

(4) 三维张量通常用于图片表示，一张彩色图片的 RGB 这 3 个通道占一维，宽和高各占一维，假如图片的宽高都是 32 个像素，就构成一个 3×32×32 的三维张量。三维张量也可用于表示自然语言。例如，10 句话，每句话都有 20 个单词，将每个单词表示为 300 分量的向量，将得到一个 10×20×300 的三维张量。

(5) 四维张量通常在卷积神经网络中表示图像。一次输入到神经网络进行训练的样本数目称为批大小(batch size)，以 CIFAR-10 数据集为例，如果批大小为 64，RGB 图像通道数是 3，每张图高和宽都为 32 像素，所以这个批次的数据组成一个四维张量，其形状为[64, 3, 32, 32]，通常表示为[batch_size, channel, height, width]。

2.1.2 张量操作

本小节介绍 PyTorch 的张量操作，这是 PyTorch 的编程基础，深度学习时需要频繁地对

张量数据进行操作。

1. 创建张量

创建张量的完整程序可参见 tensor_create.py。

1) 判断是否为 PyTorch 张量

torch.is_tensor 函数可判断输入参数是否为 PyTorch 张量。

代码 2.1 判断是否为 PyTorch 张量

```
#  创建 Python 列表
a = [1.0, 2.0, 3.0]
# 是否为 PyTorch 张量
print("a = [1.0, 2.0, 3.0]\na 是否为 PyTorch 张量:", torch.is_tensor(a))
print("a[0]: ", a[0])
```

运行结果如下：

```
a = [1.0, 2.0, 3.0]
a 是否为 PyTorch 张量: False
a[0]:  1.0
```

2) 从 Python 列表中创建张量

torch.tensor 函数可直接从 Python 列表中创建 PyTorch 张量。

代码 2.2 从 Python 列表中创建张量

```
# 从 Python 列表中创建 PyTorch 张量
b = torch.tensor([1.0, 2.0, 3.0, 4.0, 5.0, 6.0])
print("b = torch.tensor([1.0, 2.0, 3.0, 4.0, 5.0, 6.0])")
# 是否是 PyTorch 张量
print("b 是否是 PyTorch 张量:", torch.is_tensor(b))
print("b 中的数据:", b)
```

运行结果如下：

```
b = torch.tensor([1.0, 2.0, 3.0, 4.0, 5.0, 6.0])
b 是否是 PyTorch 张量: True
b 中的数据: tensor([1., 2., 3., 4., 5., 6.])
```

下面的例子创建两个维度的 PyTorch 张量。

代码 2.3 从 Python 列表中创建张量的另一个例子

```
# 从 Python 列表中创建 PyTorch 张量
c = torch.tensor([[1.0, 4.0], [2.0, 1.0], [3.0, 5.0]])
print("c = torch.tensor([[1.0, 4.0], [2.0, 1.0], [3.0, 5.0]])\nc 中的数据:", c)
```

运行结果如下：

```
c = torch.tensor([[1.0, 4.0], [2.0, 1.0], [3.0, 5.0]])
c 中的数据: tensor([[1., 4.],
        [2., 1.],
        [3., 5.]])
```

3) 创建初始化为全 0 或全 1 的张量

torch.ones 函数和 torch.zeros 函数分别用于创建全 1 和全 0 的张量。代码中的 torch.numel 函数用于计算张量中的元素个数。

代码 2.4　创建初始化为全 0 或全 1 的张量

```
# 创建 3 个元素的张量，初始化为 1
d = torch.ones(3)
print("d = torch.ones(3)\nd 中的数据:", d)

# 创建 3*3 张量，初始化为 0
e = torch.zeros(3, 3)
print("e = torch.zeros(3, 3)\ne 中的数据:", e)
print("e 中的元素个数:", torch.numel(e))
```

运行结果如下：

```
d = torch.ones(3)
d 中的数据: tensor([1., 1., 1.])
e = torch.zeros(3, 3)
e 中的数据: tensor([[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]])
e 中的元素个数: 9
```

4) 创建随机数张量

torch.randn 函数用于创建正态分布的随机数张量，torch.rand 函数用于创建取值在[0,1)范围内均匀分布的随机数张量。

代码 2.5　创建随机数张量

```
# 创建 1*2*3 张量。n 表示正态分布，从均值为 0 方差为 1 的标准正态分布中抽取一组随机数
f = torch.randn(1, 2, 3)
print("f = torch.randn(1, 2, 3)\n 正态分布:")
print("f 中的数据:", f)
print("f 中的元素个数:", torch.numel(f))

# 创建 1*2*3 张量。从[0,1)范围内均匀分布中抽取一组随机数
g = torch.rand(1, 2, 3)
print("g = torch.rand(1, 2, 3) \n 均匀分布:")
```

```
print("g 中的数据:", g)
print("g 中的元素个数:", torch.numel(g))
```

运行结果如下：

```
f = torch.randn(1, 2, 3)
正态分布:
f 中的数据: tensor([[[ 1.1537, -0.0282,  2.2642],
        [-1.1006,  1.1332, -0.3923]]])
f 中的元素个数: 6
g = torch.rand(1, 2, 3)
均匀分布:
g 中的数据: tensor([[[0.5068, 0.2015, 0.4386],
        [0.7781, 0.9347, 0.4478]]])
g 中的元素个数: 6
```

注意：由于输出是随机数，可能每次运行的实际输出不同于以上结果。

5) 创建二维对角矩阵张量

torch.eye 函数用于创建二维对角矩阵张量，并不要求矩阵必须是方阵。

代码 2.6 创建二维对角矩阵张量

```
# 创建二维对角矩阵张量
print(f"torch.eye(3):\n{torch.eye(3)}")
print(f"torch.eye(4):\n{torch.eye(4)}")
print(f"torch.eye(3, 4):\n{torch.eye(3, 4)}")
```

运行结果如下：

```
torch.eye(3):
tensor([[1., 0., 0.],
        [0., 1., 0.],
        [0., 0., 1.]])
torch.eye(4):
tensor([[1., 0., 0., 0.],
        [0., 1., 0., 0.],
        [0., 0., 1., 0.],
        [0., 0., 0., 1.]])
torch.eye(3, 4):
tensor([[1., 0., 0., 0.],
        [0., 1., 0., 0.],
        [0., 0., 1., 0.]])
```

6) 创建一维序列张量

PyTorch 常用 torch.arange 函数和 torch.linspace 函数来创建一维序列张量。

torch.arange 函数用输入参数 start(默认值为 0)指定起始值，end 指定结束值，step(可选，

默认值为 1)指定步长。注意，输出序列不会包含 end 值，因此序列取值为[start, end)区间。

代码 2.7　创建一维序列张量

```
# 在[start, end)区间内创建一维序列张量，step 为步长
# 注意 torch.range 已经弃用，最好只用 torch.arange
print(f"torch.arange(1, 4):\n{torch.arange(1, 4)}")
print(f"torch.arange(0, 3, step=0.5):\n{torch.arange(0, 3, step=0.5)}")
```

运行结果如下：

```
torch.arange(1, 4):
tensor([1, 2, 3])
torch.arange(0, 3, step=0.5):
tensor([0.0000, 0.5000, 1.0000, 1.5000, 2.0000, 2.5000])
```

torch.linspace 函数生成等分间隔的序列，由输入参数 start 指定起始值，end 指定结束值，steps(可选，默认值为 100)指定在 start 和 end 之间的数据个数。注意，输出序列会包含 end 值。

下面的代码生成 1～4 之间的等分间隔序列，第一条语句给出 steps 参数值为 5，第二条语句不明确给出 steps 参数名，使用参数值 100。注意，稍旧的 PyTorch 版本可以不指定 steps 参数，默认值为 100，但新版本的 PyTorch 要求必须指定 steps 参数。

代码 2.8　生成等分间隔的序列

```
# 生成等分间隔的序列
print(f"torch.linspace(1, 4, steps=5):\n{torch.linspace(1, 4, steps=5)}")
print(f"torch.linspace(1, 4, 100):\n{torch.linspace(1, 4, 100)}")
```

运行结果如下：

```
torch.linspace(1, 4, steps=5):
tensor([1.0000, 1.7500, 2.5000, 3.2500, 4.0000])
torch.linspace(1, 4, 100):
tensor([1.0000, 1.0303, 1.0606, 1.0909, 1.1212, 1.1515, 1.1818, 1.2121, 1.2424,
        1.2727, 1.3030, 1.3333, 1.3636, 1.3939, 1.4242, 1.4545, 1.4848, 1.5152,
        1.5455, 1.5758, 1.6061, 1.6364, 1.6667, 1.6970, 1.7273, 1.7576, 1.7879,
        1.8182, 1.8485, 1.8788, 1.9091, 1.9394, 1.9697, 2.0000, 2.0303, 2.0606,
        2.0909, 2.1212, 2.1515, 2.1818, 2.2121, 2.2424, 2.2727, 2.3030, 2.3333,
        2.3636, 2.3939, 2.4242, 2.4545, 2.4848, 2.5152, 2.5455, 2.5758, 2.6061,
        2.6364, 2.6667, 2.6970, 2.7273, 2.7576, 2.7879, 2.8182, 2.8485, 2.8788,
        2.9091, 2.9394, 2.9697, 3.0000, 3.0303, 3.0606, 3.0909, 3.1212, 3.1515,
        3.1818, 3.2121, 3.2424, 3.2727, 3.3030, 3.3333, 3.3636, 3.3939, 3.4242,
        3.4545, 3.4848, 3.5152, 3.5455, 3.5758, 3.6061, 3.6364, 3.6667, 3.6970,
        3.7273, 3.7576, 3.7879, 3.8182, 3.8485, 3.8788, 3.9091, 3.9394, 3.9697,
        4.0000])
```

torch.randperm 函数生成 0～$n-1$ 的整数的随机排列，常用于数据集样本的随机置乱，也就是打乱样本的顺序。

代码 2.9　创建取值为指定范围的随机置乱的张量

```
# 创建 0~n-1 之间随机置乱的张量
print(f"torch.randperm(5):\n{torch.randperm(5)}")
```

运行结果如下：

```
torch.randperm(5):
tensor([4, 2, 3, 1, 0])
```

7)　Tensor 与 Numpy 数组相互转换

torch.from_numpy 函数将 Numpy 数组转换为 Tensor，tensor.numpy 函数则将 Tensor 转换为 Numpy 数组。

代码 2.10　Tensor 与 Numpy 数组相互转换

```
# Numpy 数组转换为 Tensor
h = np.array([1, 2, 3, 4, 5, 6]).reshape(2, 3)
h2 = torch.from_numpy(h)
print(f"h2 = torch.from_numpy(h):\nh: {h}\nh2: {h2}")

# Tensor 转换为 Numpy 数组
h3 = h2.numpy()
print(f"h3 = h2.numpy():\nh2: {h2}\nh3: {h3}")
```

运行结果如下：

```
h2 = torch.from_numpy(h):
h: [[1 2 3]
 [4 5 6]]
h2: tensor([[1, 2, 3],
       [4, 5, 6]], dtype=torch.int32)
h3 = h2.numpy():
h2: tensor([[1, 2, 3],
       [4, 5, 6]], dtype=torch.int32)
h3: [[1 2 3]
 [4 5 6]]
```

需要注意的是，numpy 函数只能将 CPU 中的张量转换为 Numpy 数组，无法将 GPU 中的张量直接转换为 Numpy 数组。正确做法是，先将 GPU 中的张量存放到 CPU 中再进行转换。

2. 张量数据类型

使用特定函数(如 tensor、zeros 和 ones 等)来构建张量时，可以用 dtype 参数来设定数据

类型。PyTorch 定义的 dtype 参数与对应的 CPU 和 GPU 张量如表 2.1 所示。

表 2.1 张量数据类型[①]

数据类型	dtype	CPU 张量	GPU 张量
32 位浮点数	torch.float32 或 torch.float	torch.FloatTensor	torch.cuda.FloatTensor
64 位浮点数	torch.float64 或 torch.double	torch.DoubleTensor	torch.cuda.DoubleTensor
16 位浮点数 1(1 个符号位，5 个指数位，10 个有效位)	torch.float16 或 torch.half	torch.HalfTensor	torch.cuda.HalfTensor
16 位浮点数 2(1 个符号位，8 个指数位，7 个有效位)	torch.bfloat16	torch.BFloat16Tensor	torch.cuda.BFloat16Tensor
32 位复数	torch.complex32		
64 位复数	torch.complex64		
128 位复数	torch.complex128 或 torch.cdouble		
8 位无符号整数	torch.uint8	torch.ByteTensor	torch.cuda.ByteTensor
8 位有符号整数	torch.int8	torch.CharTensor	torch.cuda.CharTensor
16 位有符号整数	torch.int16 或 torch.short	torch.ShortTensor	torch.cuda.ShortTensor
32 位有符号整数	torch.int32 或 torch.int	torch.IntTensor	torch.cuda.IntTensor
64 位有符号整数	torch.int64 或 torch.long	torch.LongTensor	torch.cuda.LongTensor
布尔型	torch.bool	torch.BoolTensor	torch.cuda.BoolTensor

下面的代码定义了两个张量，使用 dtype 参数分别指定数据类型为 32 位浮点数和布尔型。

代码 2.11 使用 dtype 参数指定数据类型

```
# Tensor 数据类型
print(f"torch.tensor([1, 2], dtype=torch.float): {torch.tensor([1, 2],
dtype=torch.float)}")
print(f"torch.tensor([0, 1], dtype=torch.bool): {torch.tensor([0, 1],
dtype=torch.bool)}")
```

① 来源：https://pytorch.org/docs/stable/tensors.html#torch-tensor.

运行结果如下：

```
torch.tensor([1, 2], dtype=torch.float): tensor([1., 2.])
torch.tensor([0, 1], dtype=torch.bool): tensor([False,  True])
```

可以使用 type_as 函数转换数据类型。下面的代码将使用 torch.FloatTensor 函数定义的张量转换为整型。

代码 2.12　使用 type_as 函数转换数据类型

```
# Tensor 数据类型转换
i = torch.FloatTensor([1, 2, 3])
i2 = i.type_as(torch.IntTensor())
print(f"i2 = i.type_as(torch.IntTensor()):\ni: {i}\ni2: {i2}")
```

运行结果如下：

```
i2 = i.type_as(torch.IntTensor()):
i: tensor([1., 2., 3.])
i2: tensor([1, 2, 3], dtype=torch.int32)
```

3. 张量索引、切片、拼接及形状变换

索引、切片、拼接及形状变换都是张量的常用操作，完整程序可参见 tensor_indexing_etc.py。

1)　索引

torch.index_select 函数用于在参数 dim 指定的维度上，按参数 index 进行索引。

代码 2.13 在二维张量的第 0 维(行)上，索引第 0 行和第 2 行。

代码 2.13　索引操作

```
# torch.index_select()在维度 dim 上按 index 索引数据
a = torch.tensor([1.0, 2.0, 3.0, 4.0, 5.0, 6.0]).reshape(3, 2)
a2 = torch.index_select(a, 0, torch.LongTensor([0, 2]))
print(f"a2 = torch.index_select(a, 0, torch.LongTensor([0, 2])):\na: \n{a}\na2: \n{a2}")
```

运行结果如下：

```
a2 = torch.index_select(a, 0, torch.LongTensor([0, 2])):
a:
tensor([[1., 2.],
        [3., 4.],
        [5., 6.]])
a2:
tensor([[1., 2.],
        [5., 6.]])
```

PyTorch 还可以使用 Numpy 的索引方式，如代码 2.14 所示。

代码 2.14　使用 Numpy 的索引方式

```
# 使用 Numpy 的索引方式
print(f"a[[0, 2], :]: \n{a[[0, 2], :]}")
print(f"a[0:2, [-1]]: \n{a[0:2, [-1]]}")
```

运行结果如下：

```
a[[0, 2], :]:
tensor([[1., 2.],
        [5., 6.]])
a[0:2, [-1]]:
tensor([[2.],
        [4.]])
```

torch.masked_select 是一种更灵活的索引方式，根据掩码 mask 中的二元索引值，取出张量中的指定项。例如，代码 2.15 中所列为取出二维张量 a 中的第 0 行第 1 列、第 1 行第 0 列和第 2 行第 1 列，得到一个一维张量，然后选择取值范围为[2.0, 4.0]的元素。

代码 2.15　masked_select 索引

```
# torch.masked_select()按 mask 中取值为 True 进行筛选
mask = torch.BoolTensor([[0, 1], [1, 0], [0, 1]])
print(f"torch.masked_select(a, mask): \n{torch.masked_select(a, mask)}")
# 值在 2.0~4.0 之间为 1，否则为 0
mask = a.ge(2.0) & a.le(4.0)
print(mask)
print(f"torch.masked_select(a, mask): \n{torch.masked_select(a, mask)}")
```

运行结果如下：

```
torch.masked_select(a, mask):
tensor([2., 3., 6.])
tensor([[False,  True],
        [ True,  True],
        [False, False]])
torch.masked_select(a, mask):
tensor([2., 3., 4.])
```

2)　切片

torch.chunk 函数将张量在指定维度 dim 上分割为特定数量的块(chunks)。在代码 2.16 中，首先按默认维度 dim=0 平均切分为两块，然后按维度 dim=1 平均切分为两块。

代码 2.16 chunk 函数

```
# torch.chunk()将张量按维度 dim 进行平均切分。若不能均分，则最后一份小于其他份
b = torch.arange(10).reshape(5, 2)
print(f"torch.chunk(b, 2): \n{torch.chunk(b, 2)}")
print(f"torch.chunk(b, 2, dim=1): \n{torch.chunk(b, 2, dim=1)}")
```

运行结果如下：

```
torch.chunk(b, 2):
(tensor([[0, 1],
        [2, 3],
        [4, 5]]), tensor([[6, 7],
        [8, 9]]))
torch.chunk(b, 2, dim=1):
(tensor([[0],
        [2],
        [4],
        [6],
        [8]]), tensor([[1],
        [3],
        [5],
        [7],
        [9]]))
```

torch.split 函数将张量分割为指定形状的块。代码 2.17 首先将张量按块大小为 2 进行切分，然后按照列表[1, 4]进行切分，即第一份的行数为 1，第二份的行数为 4。

代码 2.17 split 函数

```
# torch.split()将张量按维度 dim 进行切分。当 split_size_or_sections 为 int 时，表示块的大小；为 list 时，按 len(split_size_or_sections)切分
b = torch.arange(10).reshape(5, 2)
print(f"torch.split(b, 2): \n{torch.split(b, 2)}")
print(f"torch.split(b, [1, 4]): \n{torch.split(b, [1, 4])}")
```

运行结果如下：

```
torch.split(b, 2):
(tensor([[0, 1],
        [2, 3]]), tensor([[4, 5],
        [6, 7]]), tensor([[8, 9]]))
torch.split(b, [1, 4]):
(tensor([[0, 1]]), tensor([[2, 3],
        [4, 5],
        [6, 7],
        [8, 9]]))
```

3) 拼接

torch.cat 函数将张量按维度 dim 进行拼接，但不扩展张量的维度。代码 2.18 首先将张量按维度 dim=0 拼接两次，然后按维度 dim=1 拼接 3 次。

代码 2.18 cat 函数

```
# torch.cat()将张量按维度 dim 进行拼接，不扩展张量的维度
c = torch.arange(6).reshape(2, 3)
print(f"torch.cat([c, c], dim=0): \n{torch.cat([c, c], dim=0)}")
print(f"torch.cat([c, c, c], dim=1): \n{torch.cat([c, c, c], dim=1)}")
```

运行结果如下：

```
torch.cat([c, c], dim=0):
tensor([[0, 1, 2],
        [3, 4, 5],
        [0, 1, 2],
        [3, 4, 5]])
torch.cat([c, c, c], dim=1):
tensor([[0, 1, 2, 0, 1, 2, 0, 1, 2],
        [3, 4, 5, 3, 4, 5, 3, 4, 5]])
```

torch.stack 函数在指定的新维度 dim 上进行拼接，会扩展张量的维度。代码 2.19 首先在扩展的维度 0 上拼接两次，然后在扩展的维度 2 上进行拼接，第二种方式的输出已经很难看出原张量的模样了。

代码 2.19 stack 函数

```
# torch.stack()在新维度 dim 上进行拼接，会扩展张量的维度
c = torch.arange(6).reshape(2, 3)
c_s0 = torch.stack([c, c], dim=0)  # 在扩展的维度 0 上进行拼接
print(f"torch.stack([c, c], dim=0): \n{c_s0}\n 形状：\n{c_s0.shape}")
c_s2 = torch.stack([c, c], dim=2)  # 在扩展的维度 2 上进行拼接
print(f"torch.stack([c, c], dim=2): \n{c_s2}\n 形状：\n{c_s2.shape}")
```

运行结果如下：

```
torch.stack([c, c], dim=0):
tensor([[[0, 1, 2],
         [3, 4, 5]],
        [[0, 1, 2],
         [3, 4, 5]]])
形状：
torch.Size([2, 2, 3])
torch.stack([c, c], dim=2):
tensor([[[0, 0],
         [1, 1],
```

```
        [2, 2]],
       [[3, 3],
        [4, 4],
        [5, 5]]])
形状:
torch.Size([2, 3, 2])
```

4) 形状变换

torch.reshape 函数可以变换张量形状。代码 2.20 将原张量变换成形状为(2, 4)的张量。

代码 2.20 reshape 函数

```
# torch.reshape()变换张量形状
d = torch.randperm(8)
print(f"d:\n{d}\ntorch.reshape(d, (2, 4)):\n{torch.reshape(d, (2, 4))}")
```

运行结果如下：

```
d:
tensor([4, 3, 0, 2, 6, 1, 5, 7])
torch.reshape(d, (2, 4)):
tensor([[4, 3, 0, 2],
        [6, 1, 5, 7]])
```

torch.transpose 函数可以交换张量的两个维度。代码 2.21 交换张量的维度 0 和维度 1，对于本例的二维张量，实际上就是进行了矩阵转置运算。

代码 2.21 transpose 函数

```
# torch.transpose()交换张量的两个维度
d = torch.randn((2, 3))
print(f"d:\n{d}\ntorch.transpose(d, 0, 1):\n{torch.transpose(d, 0, 1)}")
```

运行结果如下：

```
d:
tensor([[ 0.7058, -0.8687,  2.1313],
        [-1.2686, -0.2223, -0.4850]])
torch.transpose(d, 0, 1):
tensor([[ 0.7058, -1.2686],
        [-0.8687, -0.2223],
        [ 2.1313, -0.4850]])
```

torch.squeeze 函数可以压缩指定 dim 且长度为 1 的维度。如果 dim 取默认值 None，则压缩全部长度为 1 的维度。代码 2.22 首先压缩张量中长度为 1 的全部维度，然后压缩指定 dim 为 0 的维度，由于指定维度长度不为 1，因此没有效果，最后压缩指定 dim 为 1 的维度。

代码 2.22 squeeze 函数

```
# torch.squeeze()压缩指定dim且长度为1的维度。如果dim=None，则压缩全部长度为1的维度
d = torch.zeros((2, 1, 2, 1, 2))
d_s = torch.squeeze(d)
print(f"d.size():\n{d.size()}\nd_s.size():\n{d_s.size()}")
d_s0 = torch.squeeze(d, 0)
print(f"d.size():\n{d.size()}\nd_s0.size():\n{d_s0.size()}")
d_s1 = torch.squeeze(d, 1)
print(f"d.size():\n{d.size()}\nd_s1.size():\n{d_s1.size()}")
```

运行结果如下：

```
d.size():
torch.Size([2, 1, 2, 1, 2])
d_s.size():
torch.Size([2, 2, 2])
d.size():
torch.Size([2, 1, 2, 1, 2])
d_s0.size():
torch.Size([2, 1, 2, 1, 2])
d.size():
torch.Size([2, 1, 2, 1, 2])
d_s1.size():
torch.Size([2, 2, 1, 2])
```

torch.unsqueeze 函数扩展指定 dim 的维度，其长度是 1。代码 2.23 首先扩展指定 dim 为 0 的维度，然后扩展指定 dim 为 1 的维度。

代码 2.23 unsqueeze 函数

```
# torch.unsqueeze()扩展指定dim的维度，其长度是1
d = torch.arange(4)
print(f"d:\n{d}\ntorch.unsqueeze(d, 0): \n{torch.unsqueeze(d, 0)}")
print(f"d:\n{d}\ntorch.unsqueeze(d, 1): \n{torch.unsqueeze(d, 1)}")
```

运行结果如下：

```
d:
tensor([0, 1, 2, 3])
torch.unsqueeze(d, 0):
tensor([[0, 1, 2, 3]])
d:
tensor([0, 1, 2, 3])
torch.unsqueeze(d, 1):
tensor([[0],
        [1],
        [2],
        [3]])
```

4. 张量存储

展示张量存储概念的完整程序可参见 tensor_storage.py。

PyTorch 张量通常是连续内存块上的视图，张量的多维元素在连续的内存块中分配，由 torch.Storage 实例管理，不管张量有几维，都按一维数组进行存储。例如，代码 2.24 中的张量 points 是一个描述二维坐标系中 3 个点的二维张量，调用 tensor.storage 函数可返回其分配在连续内存中的一维数组。

代码 2.24　张量的存储

```
# 二维张量的存储是一维的连续内存
points = torch.tensor([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]])
print(f"points: \n{points}")
print(f"points.storage(): \n{points.storage()}")
```

运行结果如下：

```
points:
tensor([[1., 2.],
        [3., 4.],
        [5., 6.]])
points.storage():
 1.0
 2.0
 3.0
 4.0
 5.0
 6.0
[torch.FloatStorage of size 6]
```

存储实例是一维的，可以使用索引来访问对应元素，如代码 2.25 所示。

代码 2.25　使用索引访问张量存储

```
# 使用索引访问张量存储
points_storage = points.storage()
print(f"points_storage[0]: \n{points_storage[0]}")
print(f"points.storage()[0]: \n{points.storage()[0]}")
```

运行结果如下：

```
points_storage[0]:
1.0
points.storage()[0]:
1.0
```

由于张量就是存储实例的视图，更改存储实例同样影响到对应张量，如代码 2.26 所示。

代码 2.26　更改存储实例同时更改对应张量

```
# 更改存储实例同时更改对应张量
points = torch.tensor([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]])
points_storage = points.storage()
points_storage[0] = 111.0
print("更改存储实例后，张量也被更改\n", points)
```

运行结果如下：

```
更改存储实例后，张量也被更改
 tensor([[111.,   2.],
         [  3.,   4.],
         [  5.,   6.]])
```

同一个存储可以为多个张量所索引，各张量表现为存储的不同视图。索引具有 3 个属性，即大小(size)、存储偏移(storage offset)和步长(stride)。其中，大小是一个表示张量在每个维度上有多少个元素的元组，存储偏移是存储与张量中的第一个元素对应的索引，步长是为了沿存储的每个维度获取下一个元素而需要跳过的元素数量。

代码 2.27 演示张量索引的 3 个属性。对于描述 3 个点的二维张量 points，second_point 索引张量 points 的第二个点。

代码 2.27　张量索引的 3 个属性

```
# 张量索引的 3 个属性
points = torch.tensor([[1.0, 4.0], [2.0, 5.0], [3.0, 6.0]])
second_point = points[1]
print(f"second_point.storage_offset():\n{second_point.storage_offset()}")
print(f"second_point.size():\n{second_point.size()}")
print(f"second_point.shape:\n{second_point.shape}")
print(f"points.stride():\n{points.stride()}")
print(f"second_point.stride():\n{second_point.stride()}")
```

可以看到，second_point 的存储偏移为 2，这是因为第二个点在存储中跳过了第一个点，该点有两个元素。第二个点的大小(size)和形状(shape)都是 2，因为每个点都有两个元素。张量 points 的步长为(2, 1)，表明步长 stride[0]和 stride[1]分别为 2 和 1，如果要用下标 i 和 j 来访问该二维张量，实际就是访问存储中下标为 storage_offset + stride[0] * i + stride[1] * j 的元素。子张量 second_point 的步长为(1,)，表明子张量减少了一个维度。运行结果如下：

```
second_point.storage_offset():
2
second_point.size():
torch.Size([2])
second_point.shape:
```

```
torch.Size([2])
points.stride():
(2, 1)
second_point.stride():
(1,)
```

代码 2.28 演示更改子张量会影响原张量。如果想避免这一影响，可以采用克隆子张量的方法，即使用 second_point = points[1].clone()。

代码 2.28　更改子张量会影响原张量

```
# 更改子张量影响原张量
points = torch.tensor([[1.0, 4.0], [2.0, 5.0], [3.0, 6.0]])
second_point = points[1]
second_point[0] = 222.0
print(f"points:\n{points}")
```

运行结果如下：

```
points:
tensor([[  1.,   4.],
        [222.,   5.],
        [  3.,   6.]])
```

代码 2.29 验证了转置操作并不影响其存储，转置后的张量与原张量共享同一存储，因此，转置操作仅影响张量的形状和步长。

代码 2.29　验证转置操作并不影响其存储

```
# 验证转置操作并不影响其存储
points_t = points.t()
print(f"points_t:\n{points_t}")
print("转置前后的张量存储是否一致：", id(points.storage())== id(points_t.storage()))
print(f"points.stride():\n{points.stride()}")
print(f"points_t.stride():\n{points_t.stride()}")
```

运行结果如下：

```
points_t:
tensor([[1., 2., 3.],
        [4., 5., 6.]])
转置前后的张量存储是否一致： True
points.stride():
(2, 1)
points_t.stride():
(1, 2)
```

连续(contiguous)张量指的是沿着最右边的维开始存放在存储中的张量，由于不需要在

存储中进行跳跃访问，因此可以高效且有序地访问连续张量的元素。可以调用 tensor.is_contiguous 函数来检测张量是否连续，调用 tensor.contiguous 函数以返回在内存中连续的张量，如代码 2.30 所示。

代码 2.30　连续张量

```
# 张量是否连续
print(f"points.is_contiguous():\n{points.is_contiguous()}")
print(f"points_t.is_contiguous():\n{points_t.is_contiguous()}")
# 转化为连续
points_t_cont = points_t.contiguous()
print(f"points_t_cont.is_contiguous():\n{points_t_cont.is_contiguous()}")
print(f"points_t_cont.storage():\n{points_t_cont.storage()}")
print(f"points_t_cont.stride():\n{points_t_cont.stride()}")
```

运行结果如下：

```
points.is_contiguous():
True
points_t.is_contiguous():
False
points_t_cont.is_contiguous():
True
points_t_cont.storage():
 1.0
 2.0
 3.0
 4.0
 5.0
 6.0
[torch.FloatStorage of size 6]
points_t_cont.stride():
(3, 1)
```

5. 张量持久化

张量可以存储到文件中，称为张量的持久化，以便将来从文件中进行读取。完整程序可参见 tensor_serialize.py。

代码 2.31 首先定义一个要持久化的张量，然后调用 torch.save 函数将张量保存为文件，最后调用 torch.load 函数加载张量。

代码 2.31　张量持久化方法一

```
my_tensor = torch.randn((2, 3))
print("要保存的张量: \n", my_tensor)
```

```
# 将张量保存为文件方法一
torch.save(my_tensor, '../saved/my_tensor.pt')
# 加载张量方法一
my_tensor = torch.load('../saved/my_tensor.pt')
print("加载方法一的张量：\n", my_tensor)
```

代码 2.32 与代码 2.31 稍有不同，没有使用字符串的文件路径，而是使用通过 open 函数创建的 file 对象。

代码 2.32　张量持久化方法二

```
# 将张量保存为文件方法二
with open('../saved/my_tensor.pt', 'wb') as f:
    torch.save(my_tensor, f)
# 加载张量方法二
with open('../saved/my_tensor.pt', 'rb') as f:
    my_tensor = torch.load(f)
    print("加载方法二的张量：\n", my_tensor)
```

h5py 是 Python 语言用来操纵 HDF5 的模块，可以将 HDF5 文件视为能存储对象的容器，详情可参见网址 https://docs.h5py.org/en/latest/index.html。代码 2.33 展示如何将张量保存至 HDF5 文件以及从 HDF5 文件中加载张量。h5py 使用 File 对象的 create_dataset 函数来创建一个要保存的数据集，然后使用类似 Python 字典的方式读取所存储的张量。

代码 2.33　张量持久化方法三

```
# 保存到 HDF5 文件中
with h5py.File('../saved/my_tensor.hdf5', 'w') as f:
    data_set = f.create_dataset('my_train', data=my_tensor.numpy())

# 从 HDF5 文件中加载
with h5py.File('../saved/my_tensor.hdf5', 'r') as f:
    data_set = f['my_train']
    print(f"data_set.shape:\n{data_set.shape}\ndata_set.dtype:\n{data_set.dtype}")
    print(f"data_set[1:]:\n{data_set[1:]}")
    print(f"torch.from_numpy(data_set[1:]):\n{torch.from_numpy(data_set[1:])}")
```

2.1.3　广播机制

PyTorch 的很多操作都支持广播，也就是不用显式地复制数据，就可以将参与运算的张量自动扩展为同样的形状，以此来达到简化编程和实现高效算法的目的。

本小节以两个张量的相加运算为例，说明 PyTorch 的广播机制，其余数学运算同理。完整程序可参见 tensor_broadcast.py。

如果参与运算的两个张量的形状相同，显然不需要广播，如代码 2.34 所示。

代码 2.34　相同形状的张量运算

```
# 1 相同形状的张量运算
a1 = torch.arange(3)
a2 = torch.arange(4, 7)
print(f"{a1} + {a2} = \n{a1 + a2}")
```

运行结果如下：

```
tensor([0, 1, 2]) + tensor([4, 5, 6]) =
tensor([4, 6, 8])
```

由于标量与张量运算时形状不同，会触发广播机制，将标量自动扩展为与张量同样的形状，然后再进行运算，如代码 2.35 所示。

代码 2.35　标量与张量运算

```
# 2 标量与张量运算
a1 = torch.arange(3)
print(f"{a1} + 5 = \n{a1 + 5}")
print(f"{a1} + torch.tensor(5) = \n{a1 + torch.tensor(5)}")
```

运行结果如下：

```
tensor([0, 1, 2]) + 5 =
tensor([5, 6, 7])
tensor([0, 1, 2]) + torch.tensor(5) =
tensor([5, 6, 7])
```

代码 2.36 演示了相同维度但不同形状的张量运算。如果两个张量具有相同的维度，只要其中一个张量的分量为 1，就可以触发广播机制。

代码 2.36　相同维度但不同形状的张量运算

```
# 3 相同维度但不同形状的张量运算
a1 = torch.arange(12).reshape((3, 4))
a2 = torch.ones((1, 4))
print(f"{a1} + {a2} = \n{a1 + a2}")

a1 = torch.arange(12).reshape((3, 4))
a2 = torch.ones((3, 1))
print(f"{a1} + {a2} \n = {a1 + a2}")

a1 = torch.arange(3).reshape(3, 1)
a2 = torch.arange(4, 7).reshape(1, 3)
print(f"{a1} + {a2} = \n{a1 + a2}")
```

运行结果如下：

```
tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]]) + tensor([[1., 1., 1., 1.]]) =
tensor([[ 1.,  2.,  3.,  4.],
        [ 5.,  6.,  7.,  8.],
        [ 9., 10., 11., 12.]])
tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]]) + tensor([[1.],
        [1.],
        [1.]])
 = tensor([[ 1.,  2.,  3.,  4.],
          [ 5.,  6.,  7.,  8.],
          [ 9., 10., 11., 12.]])
tensor([[0],
        [1],
        [2]]) + tensor([[4, 5, 6]]) =
tensor([[4, 5, 6],
        [5, 6, 7],
        [6, 7, 8]])
```

两个不同维度的张量运算，其中一个张量不存在需要对齐的维度，同样会触发广播机制。代码 2.37 演示了这种情形。

代码 2.37　不同维度的张量运算

```
# 4 不同维度的张量运算
a1 = torch.arange(6).reshape(3, 2)
a2 = torch.arange(4, 6)
print(f"{a1} + {a2} = \n{a1 + a2}")
```

运行结果如下：

```
tensor([[0, 1],
        [2, 3],
        [4, 5]]) + tensor([4, 5]) =
tensor([[ 4,  6],
        [ 6,  8],
        [ 8, 10]])
```

总结一下，按照 PyTorch 广播机制文档 (https://pytorch.org/docs/stable/notes/broadcasting.html)，当两个张量满足下面的条件时，就会触发广播机制。

① 每个张量至少有一个维度。

② 迭代维度尺寸时，从尾部维度开始，每个对应维度的分量必须满足以下条件之一：

要么相等，要么其中一个张量的分量为 1，或者其中一个张量不存在该维度。

2.1.4 在 GPU 上使用 Tensor

深度学习对计算机的计算力要求很高，GPU 能够加速深度模型训练，拥有 GPU 肯定会在深度学习上体验更好。PyTorch 支持 GPU 加速，而且支持多 GPU 并行训练。

本小节介绍 PyTorch 使用 GPU 的方法，完整程序可参见 tensor_gpu.py。

代码 2.38 展示如何获取本机 GPU 配置的代码。

代码 2.38 获取本机 GPU 配置

```
# GPU 是否可用
print("torch.cuda.is_available(): ", torch.cuda.is_available())
# GPU 数量
print("torch.cuda.device_count(): ", torch.cuda.device_count())
# 当前 GPU 设备
print("torch.cuda.current_device(): ", torch.cuda.current_device())
# GPU 设备名
print("torch.cuda.get_device_name('cuda:0'): ", torch.cuda.get_device_name('cuda:0'))
```

新建张量时，可以使用 device 属性来指定期望的设备，device 的默认属性值为 None，如代码 2.39 所示。

代码 2.39 指定张量的设备

```
# 新建张量
ts = torch.tensor([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]])
# 新建 GPU 张量
ts_gpu = torch.tensor([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]], device='cuda')
print("torch.tensor([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]]): \n", ts)
print("torch.tensor([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]], device='cuda'): \n", ts_gpu)
```

创建张量以后，可以调用 tensor.cuda 函数返回其 GPU 张量的副本，也就是在 CUDA 内存中的张量对象；也可以调用 tensor.cpu 函数返回其 CPU 张量的副本。调用 tensor.to 函数可以将张量转换到由 device 属性指定的设备中。注意，不同设备的张量是不可以直接计算的，必须先将要计算的张量放到同一个设备中。另外，代码中要使用 cuda 表示 gpu，cuda:0 中冒号后面的 0 表示 GPU 的编号，实际指的是第 1 张 GPU 卡，以此类推。类似地，使用 cpu:0、cpu:1 指定多 CPU 环境下不同的 CPU。

代码 2.40 CPU 与 GPU 张量转换

```
# 转换为 GPU 张量
ts_gpu = ts.to(device='cuda')
```

```
print("演示 to(device='cuda'): \n", ts_gpu)
# 另一种转换方式
ts_gpu = ts.to(device='cuda:0')
print("演示 to(device='cuda:0'): \n", ts_gpu)

# 张量运算
ts = 2 * ts
print("ts = 2 * ts: \n", ts)
# 可以这样转换
ts_gpu = 2 * ts.to(device='cuda')
print("2 * ts.to(device='cuda'): \n", ts_gpu)

# GPU 张量运算
ts_gpu = ts_gpu + 10
print("ts_gpu + 10: \n", ts_gpu)

# 转换为 CPU 张量
ts_cpu = ts_gpu.to(device='cpu')
print("演示 to(device='cpu'): \n", ts_cpu)

# 另一种转换方法
ts_gpu = ts.cuda()
print(ts_gpu)
ts_gpu = ts.cuda(0)
print(ts_gpu)
ts_cpu = ts_gpu.cpu()
print(ts_cpu)
```

为了避免因为程序的假设和计算机的实际配置不符而引发运行时错误，最好能够根据配置来选择是否使用 GPU 加速，代码 2.41 演示了这种方法。

代码 2.41　根据配置选择 GPU 加速

```
# 根据配置选择 GPU 加速
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

# 新建 GPU 或 CPU 张量
ts_unknown = torch.tensor([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]], device=device)
print("根据配置选择 GPU 加速: \n", ts_unknown)
```

由于本书的配套程序在单 GPU 的环境下已经能很好地运行，因此不考虑多 GPU 环境的编程，感兴趣的读者可参考 PyTorch 官网中的数据并行处理教程。网址为 https://pytorch.org/tutorials/beginner/blitz/data_parallel_tutorial.html。

2.2 自动求导

autograd 模块是 PyTorch 的核心功能，它为张量上的所有操作提供了自动求导机制。使用这个工具，就不再需要手工完成求导计算，简化了神经网络中复杂的优化计算。

2.2.1 自动求导概念

自动求导的对象是张量，将某张量的 requires_grad 属性设置为 True，就会追踪对该张量的全部操作。然后通过调用 backward 函数来自动计算梯度，并将梯度累加到张量的 grad 属性中。

一般而言，都会将模型的可训练参数的 requires_grad 属性设置为 True，在训练模型时需要梯度跟踪，但评估模型时不需要梯度跟踪。如果想暂时停止梯度跟踪，可以调用张量的 detach 函数隔离计算历史，阻止跟踪该张量的计算记录。停止梯度跟踪的另一种方法是将计算代码放到 with torch.no_grad()块中，这在评估模型性能时经常用到。

自动求导还有一个求导函数，它对完整的计算历史进行编码，存放在张量的 grad_fn 属性中，引用创建张量的 Function 对象。如果张量由用户手动创建，则其 grad_fn 属性为 None。

如果要计算张量的导数，可调用该张量的 backward 函数。如果张量自身是一个标量，则不需要为 backward 函数指定参数。但如果张量是一个向量、矩阵或更多维的张量，则需要指定一个 gradient 参数，以匹配张量的形状。

2.2.2 自动求导示例

本小节展示 PyTorch 的自动求导方法，详见 tensor_autograd.py。

1. 标量自动求导

此处以单变量线性回归为例，说明 PyTorch 如何进行自动求导。线性回归模型可表示为

$$y = wx + b \tag{2.1}$$

式中：x 为输入；y 为输出；w 和 b 都为模型参数。根据导数知识，有 $\frac{\partial y}{\partial w} = x$，$\frac{\partial y}{\partial b} = 1$。

代码 2.42 验证了单变量线性回归的自动求导。PyTorch 使用动态计算图，使用完毕后默认会将计算图删除，如果有多次使用该计算图的需求，可在 backward 函数中将 retain_graph

属性设置为 True。

代码 2.42 单变量线性回归的自动求导

```
# 先以常见的 y=wx+b 为例来进行说明
x = torch.tensor([2.0])
w = torch.tensor([1.0], requires_grad=True)
b = torch.tensor([0.0], requires_grad=True)

y = w * x + b
print(f"y.grad_fn: {y.grad_fn}")
# 反向传播
y.backward(retain_graph=True)
# 打印导数
print(f"x.grad: {x.grad}")
print(f"w.grad: {w.grad}")
print(f"b.grad: {b.grad}")
```

可以看到，grad_fn 属性存放张量的计算历史，AddBackward0 命名表示最后的计算是加运算。由于未指定 x 的 requires_grad 属性为 True，因此 x.grad 值为 None。w.grad 和 b.grad 分别表示 $\frac{\partial y}{\partial w}$ 和 $\frac{\partial y}{\partial b}$，值分别是 tensor([2.])和 tensor([1.])，符合其值应该等于 x 和 1 的预期。运行结果如下：

```
y.grad_fn: <AddBackward0 object at 0x000001A951D55148>
x.grad: None
w.grad: tensor([2.])
b.grad: tensor([1.])
```

注意到自动求导会累加梯度，所以要在反向传播之前将梯度清零，否则会得到错误的结果。代码 2.43 调用 grad.zero_函数分别将 w 和 b 的梯度清零。

代码 2.43 梯度清零

```
if w.grad is not None:
   w.grad.zero_()
if b.grad is not None:
   b.grad.zero_()
```

2. 矩阵自动求导

再来看一个矩阵的自动求导示例(代码 2.44)。首先创建一个 2×2 的矩阵，并调用 requires_grad_函数来设置 requires_grad 属性为 True 以启用梯度跟踪。

代码 2.44　矩阵的自动求导

```
# 再看自动求导的矩阵形式
x = torch.arange(4).view(2, 2).float()
x.requires_grad_(True)
print(f"x: {x}")
print(f"x.grad_fn: {x.grad_fn}")
```

由于张量 x 由手动创建，因此 grad_fn 为 None。运行结果如下：

```
x: tensor([[0., 1.],
       [2., 3.]], requires_grad=True)
x.grad_fn: None
```

代码 2.45 对张量 x 做一次加法运算。

代码 2.45　简单的加法运算

```
y = x + 1
print(f"y: {y}")
print(f"y.grad_fn: {y.grad_fn}")
```

由于张量 y 是由张量 x 计算而得，因此 grad_fn 为<AddBackward0>。运行结果如下：

```
y: tensor([[1., 2.],
       [3., 4.]], grad_fn=<AddBackward0>)
y.grad_fn: <AddBackward0 object at 0x0000015939305188>
```

is_leaf 属性测试张量是否为叶子节点，其值为 True 的 Tensor 就是手动创建的叶子节点，否则就不是。代码 2.46 测试了张量 x 和张量 y 是否为叶子节点。

代码 2.46　测试张量是否为叶子节点

```
print(f"x.is_leaf: {x.is_leaf}\ny.is_leaf: {y.is_leaf}")
```

由于张量 x 是手工创建，而张量 y 是计算而得的，因此两者的 is_leaf 属性分别为 True 和 False。运行结果如下：

```
x.is_leaf: True
y.is_leaf: False
```

再对张量 y 进行立方运算和平均运算操作，如代码 2.47 所示。

代码 2.47　立方运算和平均运算

```
z = y ** 3
out = z.mean()
print(f"z: {z}")
print(f"out: {out}")
```

运行结果如下：

```
z: tensor([[ 1.,  8.],
           [27., 64.]], grad_fn=<PowBackward0>)
out: 25.0
```

最后对 out 张量进行反向传播，由于 out 是一个标量，因此 out.backward() 与 out.backward(torch.tensor(1.)) 等价。

代码 2.48　反向传播

```
# 反向传播
out.backward()  # 等价于 out.backward(torch.tensor(1.))
print(f"x.grad: {x.grad}")
```

运行结果如下：

```
x.grad: tensor([[ 0.7500,  3.0000],
        [ 6.7500, 12.0000]])
```

下面解释 x.grad 的运行结果。为了方便，将 out 张量用 o 来表示。由于 $o=\frac{1}{4}\sum_i z_i$，且 $z_i=(x_i+1)^3$，所以 $\frac{\partial o}{\partial x_i}=\frac{3}{4}(x_i+1)^2$，将 x_i 分别取 1、2、3、4 并代入，即可得到 x.grad 的计算结果。

2.3　数据集 API

很多机器学习问题都需要读取数据集和进行简单的预处理，PyTorch 提供了 torch.utils.data 模块，能方便读取数据和预处理，其中 Dataset 类、IterableDataset 类和 DataLoader 类的应用最为广泛，本节详细介绍这 3 个类的使用方法。

2.3.1　自定义数据集类

PyTorch 的自定义数据集可使用 Dataset 类、IterableDataset 类来定义，前者用于实现 Map-style(映射风格)的数据集，后者用于实现 Iterable-style(迭代风格)的数据集。

1. Dataset 类

Dataset 类是一个映射风格的数据集，继承该类的子类实例需要实现__getitem__()函数和

__len__()函数，代表从索引(键值)到数据样本的映射。例如，如果使用 dataset[idx]来访问 Dataset 定义的图片数据集，就能从文件目录读到第 idx 张图片和对应的标签。

Dataset 是自定义数据集的抽象类，用户可以自己定义数据集类继承该抽象类，所有子类必须重写__getitem__()函数，以支持获取给定键值的数据样本。子类可以选择性地重写__len__()函数，它将通过 DataLoader 的一些 Sampler(样本抽样器)实现以及默认选项返回数据集的大小，详见后文。

代码 2.49 是鸢尾花数据集类的简单实现，IrisDataset 继承 Dataset，重写__getitem__()和__len__()函数。完整程序可参见 iris_dataset.py。

代码 2.49　鸢尾花数据集类

```
class IrisDataset(Dataset):
    """ 鸢尾花数据集 """
    def __init__(self):
        super(IrisDataset).__init__()
        data = np.loadtxt("../datasets/fisheriris.csv", delimiter=',', dtype=np.float32)
        self.x = torch.from_numpy(data[:, 0:-1])
        self.y = torch.from_numpy(data[:, [-1]])
        self.len = data.shape[0]

    def __getitem__(self, index):
        return self.x[index], self.y[index]

    def __len__(self):
        return self.len
```

2. IterableDataset 类

IterableDataset 类是一个迭代风格的数据集，继承该类的子类实例需要实现__iter__()函数，该函数返回该数据集样本的迭代器。这种类型的数据集特别适合用于随机读取的代价很高甚至无法实现，且批大小取决于所获取数据的情形。例如，如果使用 iter(dataset)来访问这类数据集，就能返回从数据库、远程服务器读取的数据流，甚至返回实时生成的日志。

代码 2.50 是鸢尾花数据集类的迭代风格实现，IrisIterDataset 继承 IterableDataset 类，重写__iter__()函数。本例实现一个自定义的迭代器，因此还实现__next__()函数。注意，迭代风格的数据集不允许随机置乱，即不能设置 shuffle=True。完整程序可参见 iris_iter_dataset.py。

代码 2.50　迭代风格鸢尾花数据集类

```
class IrisIterDataset(IterableDataset):
    """ 鸢尾花数据集 """
    def __init__(self):
```

```
        super(IrisIterDataset).__init__()
        data = np.loadtxt("../datasets/fisheriris.csv", delimiter=',', dtype=np.float32)
        self.x = torch.from_numpy(data[:, 0:-1])
        self.y = torch.from_numpy(data[:, [-1]])
        self.len = data.shape[0]
        self.idx = 0

    def __iter__(self):
        self.idx = 0
        return self

    def __next__(self):
        if self.idx < self.len:
            # 此处应该从数据库或远程实时得到数据，这里只是示例，因此简单处理一下
            idx = self.idx
            self.idx += 1
            return self.x[idx], self.y[idx]
        else:
            raise StopIteration
```

2.3.2 DataLoader 类

实现自定义数据集之后，就可以返回数据集样本了。但这种直接通过索引来返回样本的方式比较原始，无法让数据集一次提供一个批次(batch)的数据，也无法对数据进行随机置乱和并行加速。为此，PyTorch 专门提供 DataLoader 类来实现这些功能。

DataLoader 类是一个数据加载器，它将数据集和样本抽样器组合在一起，并提供给定数据集上的可迭代对象，其构造函数如下：

```
DataLoader(dataset, batch_size=1, shuffle=False, sampler=None,
           batch_sampler=None, num_workers=0, collate_fn=None,
           pin_memory=False, drop_last=False, timeout=0,
           worker_init_fn=None, *, prefetch_factor=2,
           persistent_workers=False)
```

各参数含义如下。

- dataset：要加载数据的数据集对象。
- batch_size：批大小，每一批需要加载多少个样本，默认值为 1。
- shuffle：设置为 True 时将在每个 epoch 重新置乱样本顺序，默认值为 False。
- sampler：样本抽样器，定义从数据集中抽取样本的策略。
- batch_sampler：类似于 sampler，但一次返回一批的索引。使用该属性就不能同时

使用 batch_size、shuffle、sampler 和 drop_last 属性。

- num_workers：加载数据的子进程数。值为 0 表示数据由主进程加载，默认值为 0。
- collate_fn：将一系列样本合并，形成一个小批量的张量。在映射风格数据集的批加载时才可以使用。
- pin_memory：如果为 True，数据加载器先把张量复制到 CUDA 固定内存(pinned memory)中，然后再返回它们。
- drop_last：如果数据集大小不能被批大小整除，则在该属性值设置为 True 时，会丢弃最后一个不完整的批；否则，最后一批的样本数将少于其他批。默认值为 False。
- timeout：如果为正数，则表示从 workers 处收集批的超时值。该属性应该总是为非负值，默认值为 0。
- worker_init_fn：如果不为 None，则在每个 worker 的子进程中将 worker id 作为输入，默认值为 None。
- prefetch_factor：每个 worker 提前装载的样本数。2 表示在所有 workers 中预取总共 2 * num_workers 样本，默认值为 2。
- persistent_workers：如果为 True，则在使用数据集一次之后数据加载器不会关闭 worker 进程，这允许保持 workers 数据集实例为活动的，默认值为 False。

虽然 DataLoader 类的构造函数有许多参数，但一般应用只需要设置 dataset、batch_size 和 shuffle，不用过于纠结其他参数的用法。

DataLoader 是一个可迭代对象，自然可以如同迭代器那样使用。代码 2.51 是 DataLoader 的简单示例。先实例化数据集，然后调用 DataLoader 构造函数创建一个数据加载器对象，最后使用一个双重循环从加载器中读取数据。

代码 2.51　DataLoader 简单示例

```
# 实例化
iris = IrisDataset()
irir_loader = DataLoader(dataset=iris, batch_size=10, shuffle=True)
for epoch in range(2):
    for i, data in enumerate(irir_loader):
        # 从 irir_loader 中读取数据
        inputs, labels = data

        # 打印数据集
        print(f"轮次：{epoch}\t 输入数据形状：{inputs.data.size()}\t 标签形状：
                    {labels.data.size()}")
```

代码 2.52 是 DataLoader 的另一个示例，如果数据集继承 IterDataset，那么对应的 DataLoader 不允许随机置乱，即不能设置 shuffle=True。

代码 2.52 DataLoader 另一个示例

```
iris = IrisIterDataset()

# 继承 IterDataset 的 DataLoader 不允许 shuffle=True
irir_loader = DataLoader(dataset=iris, batch_size=10)
for epoch in range(2):
    for i, data in enumerate(irir_loader):
        # 从 irir_loader 中读取数据，一批 10 个样本
        inputs, labels = data

        # 打印数据集
        print(f"轮次：{epoch}\t 输入数据形状：{inputs.data.size()}\t 标签形状：
                    {labels.data.size()}")
```

2.4 torchvision 工具示例

计算机视觉(Computer Vision，CV)是深度学习的一个重要应用领域。PyTorch 提供现成的 torchvision 工具，帮助处理图像和视频。torchvision 包含一些常用的数据集、模型、转换函数等，学习和使用这些 API 有助于更快更好地在 CV 领域应用 PyTorch。

torchvision 的数据集都是 torch.utils.data.Dataset 的子类，都实现了__getitem__和__len__函数，可传递给支持并行处理的多进程 torch.utils.data.DataLoader 加载器进行数据加载。数据集包括 MNIST、Fashion-MNIST、CIFAR、CelebA 和 ImageNet 等工具，可直接下载相应的数据集到本地使用，免去查找数据集和编写对应数据集类的麻烦。

本节首先介绍如何手工编写简单的图像数据集，然后介绍部分常用 torchvision 工具的使用。

2.4.1 编写简单的图像数据集

本小节以 Kaggle 赛题猫狗大战(Dogs vs. Cats)为例，说明如何编写图像数据集。猫狗大战的 25000 张图片都存放在同一个目录下，文件命名为 cat.xxx.jpg 或 dog.xxx.jpg，其中的 xxx 为取值范围为 0～12499 的图片编号。需要根据文件名来判断是猫还是狗。

代码 2.53 定义了一个映射风格的猫狗数据集类。由于图片文件占内存较多，因此只保

存图片的完整路径，而不预先加载图片内容，只是在__getitem__()函数中才读取图片数据。注意，Image.open()函数打开的是一个图像对象，需要调用 np.array()函数转换成形状为(高 H, 宽 W, 通道 C)的 numpy.ndarray 对象，然后再调用 torch.from_numpy()函数转换为 PyTorch 张量。

代码 2.53　猫狗数据集类

```
class DogsCatsDataset(Dataset):
    """ 猫狗数据集定义 """

    def __init__(self, root):
        # root 为图片的路径
        imgs = os.listdir(root)
        # 仅保存图片完整路径
        self.imgs = [os.path.join(root, img) for img in imgs]

    def __getitem__(self, idx):
        img_path = self.imgs[idx]
        # 根据文件名确定标签。dog 标签 1，cat 标签 0
        label = 1 if 'dog' in img_path.split('/')[-1] else 0
        pil_img = Image.open(img_path)
        img_array = np.array(pil_img)
        # 形状：(H, W, C)，取值范围：[0, 255]，通道：RGB
        img = torch.from_numpy(img_array)
        return img, label

    def __len__(self):
        return len(self.imgs)
```

代码 2.54 为猫狗数据集测试代码。首先实例化猫狗数据集对象，然后读取第一张图片并显示。注意，Python 索引是从零开始的，因此这里所说的第一张实际上在前面还有一张。

代码 2.54　猫狗数据集测试代码

```
dataset = DogsCatsDataset("../datasets/kaggledogvscat/original_train")
img, label = dataset[1]
print("第一个样本")
print("图像形状：", img.shape)
plt.imshow(img)
plt.show()
```

第一张图片如图 2.1 所示。

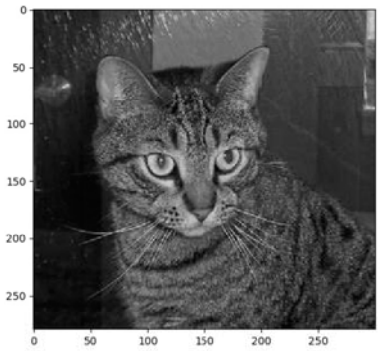

图 2.1 猫狗数据集第一张图片

2.4.2 Transforms 模块

2.4.1 小节定义的猫狗数据集类存在以下 3 个主要问题。

① 每张图片的宽、高不一样，需要进行处理。

② 图片每个像素的取值范围是 0～255，深度网络需要归一化到 0～1 范围。

③ 诸如 PIL 工具读取到的图片张量的形状是(H, W, C) ，但是深度网络接收的图片张量的形状为(B, C, H, W)，需要进行转换。其中，B 为一批样本中的图片数，H 和 W 分别为图片的高和宽，C 为图片的通道数。

对此，PyTorch torchvision 工具的 Transforms 模块提供对 PIL Image 对象的图像转换功能，该模块提供了常用预处理功能的开箱即用实现，如填充、裁剪、灰度模式、线性变换、将图像转换成 PyTorch 张量，以及一些实现数据增强的功能，如翻转、随机裁剪、颜色抖动。常用的转换操作如下。

① CenterCrop、RandomCrop、RandomSizedCrop、FiveCrop、TenCrop：裁剪图片尺寸。

② Grayscale：转换为灰度图像。

③ Pad：用给定的值填充给定图像。

④ RandomAffine、RandomApply、RandomGrayscale、RandomPerspective、GaussianBlur、RandomChoice、RandomOrder、RandomErasing：对图像施加随机变换。

⑤ RandomHorizontalFlip、RandomVerticalFlip：以给定概率水平或垂直翻转给定图像。

⑥ Resize：调整输入图像为给定的尺寸。

⑦ LinearTransformation：用一个方阵和一个离线计算的均值向量对张量图像进行

变换。

⑧ Normalize：用给定的均值和标准差来规范化张量图像。

⑨ Compose：将几个 Transforms 组合在一起。

⑩ ConvertImageDtype：将张量图像转换为给定的 dtype 类型，并相应缩放。不支持 PIL 图像。

⑪ ToPILImage：把 tensor 或 ndarray 数组转换为 PIL Image 对象。

⑫ ToTensor：将 PIL 图像对象或 ndarray 数组转换为 Tensor。注意会将像素取值[0, 255]归一化为[0, 1]范围，并且将原来的形状(H, W, C)转置为(C, H, W)。

⑬ Lambda：应用用户自定义的 lambda 函数进行转换。

此外，Transforms 还提供上述转换功能对应的转换函数，实现对转换管道的细粒度控制。这些函数位于 torchvision.transforms.functional 模块下，一般使用以下语句导入转换函数：

```
import torchvision.transforms.functional as TF
```

代码 2.55 是一个简单的 Transforms 示例。首先定义一个 my_transform 实例；然后对输入图片进行尺寸调整、中心裁剪和转换为 tensor 这 3 项操作；最后在数据集中应用该自定义转换对输入图像进行变换。完整代码可参见 dogs_cats_transforms.py。

代码 2.55　Transforms 示例

```
my_transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor()
])


class DogsCatsDataset(Dataset):
    """ 猫狗数据集定义 """

    def __init__(self, root):
        # root 为图片的路径
        imgs = os.listdir(root)
        # 仅保存图片完整路径
        self.imgs = [os.path.join(root, img) for img in imgs]
        self.transforms = my_transform

    def __getitem__(self, idx):
        img_path = self.imgs[idx]
        # 根据文件名确定标签。dog 标签 1，cat 标签 0
        label = 1 if 'dog' in img_path.split('/')[-1] else 0
        pil_img = Image.open(img_path)
```

```
        if self.transforms:
            img = self.transforms(pil_img)
        return img, label

    def __len__(self):
        return len(self.imgs)
```

2.4.3 Normalize 用法

在 Transforms 中通常使用 Normalize 来规范化张量图像，这时就需要计算数据集的均值和标准差。例如，以下代码的变换器对 CIFAR-10 数据集进行规范化，需要用到计算出来的均值(0.4914, 0.4822, 0.4465)和标准差(0.2023, 0.1994, 0.2010)：

```
# 数据变换
transform_train = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
    # RGB 归一化的均值和标准差
])
```

网上有一些代码直接给出类似的均值和标准差数值，可以直接使用。但是，了解如何计算出这些数值也很重要。cifar10_normalize.py 实现了计算 CIFAR-10 数据集的均值和标准差，代码 2.56 是核心函数。它首先定义一个数据加载器，并初始化存放均值和标准差的张量 mean 和 std；然后用两个 for 循环计算并累加每张图片的均值和标准差；最后求均值和标准差的平均值并返回。

代码 2.56 计算指定图像数据集的均值和标准差函数

```
def get_mu_sigma(train_dataset, dim=3):
    """ 计算指定图像数据集的均值和标准差 """
    print("计算指定图像数据集的均值和标准差")
    data_length = len(train_dataset)
    print("数据集大小：", data_length)
    train_loader = DataLoader(train_dataset, batch_size=1, shuffle=False)
    mean = torch.zeros(dim)
    std = torch.zeros(dim)
    for img, _ in train_loader:
        # 计算并累加每张图片的均值和标准差
        for d in range(dim):
            mean[d] += img[:, d, :, :].mean()
            std[d] += img[:, d, :, :].std()
    # 求平均
    mean.div_(data_length)
```

```
    std.div_(data_length)
    return mean, std
```

程序的输出结果如下：

```
计算指定图像数据集的均值和标准差
数据集大小： 50000
计算出来的均值和标准差： (tensor([0.4914, 0.4822, 0.4465]), tensor([0.2023, 0.1994,
0.2010]))
```

2.4.4 ImageFolder 用法

ImageFolder 是一种通用的数据加载器，它假定所有的图片文件都按类别分别存放，子目录名为类别名，每个子目录下存放对应类别的图片。本书 1.3.4 小节已经按照上述要求对猫狗数据集进行了预处理，处理后的目录结构如图 2.2 所示。

图 2.2 猫狗数据集预处理后的目录结构

ImageFolder 类的构造函数如下：

```
torchvision.datasets.ImageFolder(root, transform=None, target_transform=None,
loader=default_loader, is_valid_file=None)
```

各参数含义如下。

- root：根目录路径。
- transform：接收 PIL 图像并返回转换后版本的转换函数，如 transforms.RandomCrop。
- target_transform：接收目标标签并转换。
- loader：加载给定路径的图像。
- is_valid_file：检查给定路径的图片文件是否有效，用于检查文件是否已损坏。

dogs_cats_imagefolder.py 演示如何使用 ImageFolder 类来读取预处理后的猫狗数据集图片。

代码 2.57 首先定义一个简单的图像转换器 data_transform，对输入图像做 Resize 和 ToTensor 转换操作；然后使用 ImageFolder 类读取指定目录下的图像文件，这里使用 Python 字典来存放训练集、验证集和测试集数据；最后使用 DataLoader 加载 3 种数据集。一般而言，训练集需要随机置乱，但验证集和测试集不需要，因此使用 shuffle=(ds_type == "train") 来指定只对训练集置乱。

代码 2.57 读取数据集

```
batch_size = 16  # 批大小

# 猫狗数据集图片目录
imgs_dir = '../datasets/kaggledogvscat/small'

# 图像转换
data_transform = transforms.Compose([transforms.Resize([64, 64]),
                 transforms.ToTensor()])
# 数据集
dog_vs_cat_datasets = {ds_type: datasets.ImageFolder(root=os.path.join
                      (imgs_dir, ds_type), transform=data_transform[ds_type])
                       for ds_type in ["train", "validation", "test"]}
# 数据加载器
data_loader = {ds_type: DataLoader(dataset=dog_vs_cat_datasets[ds_type],
              batch_size=batch_size, shuffle=(ds_type == "train"))
              for ds_type in ["train", "validation", "test"]}
```

代码 2.58 加载训练集并打印结果。可以调用数据集对象的 class_to_idx、classes 和 imgs 属性，分别获取数据集类别索引、数据集类别和图片路径。

代码 2.58 加载训练集并打印结果

```
x_train, y_train = next(iter(data_loader["train"]))
print(f'训练集样本个数：{len(x_train)}')
print(f'训练集标签个数：{len(y_train)}')
print('训练集样本形状：', x_train.shape)
print('训练集标签形状：', y_train.shape)

index_classes = dog_vs_cat_datasets["train"].class_to_idx
print("数据集类别索引：")
# 输出为：{'cats': 0, 'dogs': 1}
print(index_classes)

class_names = dog_vs_cat_datasets["train"].classes
print("数据集类别：")
```

```
# 输出为：['cats', 'dogs']
print(class_names)

img_path = dog_vs_cat_datasets["train"].imgs
print("图片路径：")
# 输出为：[('../datasets/kaggledogvscat/small\\train\\cats\\cat.0.jpg', 0), ...]
print(img_path)
```

图 2.3 显示读取的一批训练数据集样本。由于已经进行随机置乱，因此图片中既有猫也有狗。

图 2.3　训练数据集部分样本

2.5　torchtext 工具示例

自然语言处理(NLP)是深度学习的一个重要应用领域。PyTorch 提供有现成的 torchtext 工具，帮助处理 NLP 任务。torchtext 包含一些数据处理实用程序和常用的数据集，避免了重新编程的麻烦。

torchtext 包含的数据集十分丰富，有 TextClassificationDataset、AG_NEWS 和 SogouNews 等文本分类数据集，有 WikiText-2、WikiText103 和 PennTreebank 等语言模型数据集，有 IWSLT2016 和 IWSLT2017 等机器翻译数据集，有 UDPOS 和 CoNLL2000Chunking 等序列标签数据集，还有 SQuAD 1.0 和 SQuAD 2.0 等问答数据集。

本节首先介绍如何手工编写简单的文本数据集，然后介绍如何使用 torchtext 工具来编写文本数据集。

2.5.1　编写文本预处理程序

文本预处理程序一般需要完成以下工作，即读入文本数据、拆分句子并分词、创建词典。其中，词典应包含两个部分：能够将单词映射为索引以便输入到模型中，即编码；以

及将索引映射回单词，即解码。

本小节以读取莎士比亚作品、分词并创建词典为例，说明如何编写一个实用的文本预处理程序，完整程序可参见 text_preprocess.py。

代码 2.59 是 3 个辅助函数：read_text 函数打开指定 zip 文件并返回所读出的各行；tokenize 函数将句子划分为单词；count_corpus 函数返回一个统计单词出现次数的字典。

代码 2.59　3 个辅助函数

```
def read_text(filename):
    """ 打开文件返回读出的各行 """
    with gzip.open(filename, 'rt') as f:
        lines = [line.strip().lower() for line in f]
    return lines


def tokenize(sentences):
    """ 将句子划分为单词 """
    return [sentence.split(' ') for sentence in sentences]


def count_corpus(sentences):
    """ 返回统计单词出现次数的字典 """
    tokens = [tk for st in sentences for tk in st]
    return collections.Counter(tokens)
```

代码 2.60 实现一个词典类。代码中的 min_freq 设定一个最小词频的阈值，只保留不小于该阈值的单词，use_special 为 True 则使用填充<pad>符号、序列开始<bos>符号、序列结束<eos>符号和未登录词<unk>符号，idx_to_token 是序号映射为单词的列表，token_to_idx 则是单词映射为索引的词典。

代码 2.60　词典类

```
class Vocab(object):
   """ 词典类 """

   def __init__(self, tokens, min_freq=0, use_special=False):
      counter = count_corpus(tokens)
      # 词频
      self.token_freqs = list(counter.items())
      # 索引映射为单词。这里的列表只记录单词，索引就是单词在列表中的下标
      self.idx_to_token = []
      # 如果 use_special 为 True，则 4 个特殊符号都保留；否则只使用<unk>符号
      if use_special:
         self.pad, self.bos, self.eos, self.unk = (0, 1, 2, 3)
         self.idx_to_token += ['<pad>', '<bos>', '<eos>', '<unk>']
```

```
        else:
            self.unk = 0
            self.idx_to_token += ['<unk>']

        # 将新词加入到 idx_to_token 列表中
        self.idx_to_token += [token for token, freq in self.token_freqs
                              if freq >= min_freq and token not in self.idx_to_token]
        # 单词映射为索引的词典
        self.token_to_idx = dict()
        for idx, token in enumerate(self.idx_to_token):
            self.token_to_idx[token] = idx

    def __len__(self):
        return len(self.idx_to_token)

    def __getitem__(self, tokens):
        if not isinstance(tokens, (list, tuple)):
            return self.token_to_idx.get(tokens, self.unk)
        return [self.__getitem__(token) for token in tokens]
```

在如代码 2.61 所示的主函数中，首先读取数据文本；然后进行分词和构建字典工作；最后输出单词和索引的几个示例。

代码 2.61　主函数

```
    # 读取数据文本
    lines = read_text("../datasets/shakespeare.txt.gz")
    print("句子总数： %d" % len(lines))

# 分词
    tokens = tokenize(lines)
    print("打印前 2 句：\n", tokens[0: 2])

    # 构建词典
    # vocab = Vocab(tokens, 0, True)
    vocab = Vocab(tokens)
    print("将字典中前 10 个词的单词映射为索引：")
    print(list(vocab.token_to_idx.items())[0: 10])

    for i in range(2):
        print("单词：", tokens[i])
        print("索引：", vocab[tokens[i]])
```

程序运行结果如下：

```
句子总数： 40000
打印前 2 句：
```

```
[['first', 'citizen:'], ['before', 'we', 'proceed', 'any', 'further,', 'hear', 'me',
'speak.']]
将字典中前 10 个词的单词映射为索引：
[('<unk>', 0), ('first', 1), ('citizen:', 2), ('before', 3), ('we', 4), ('proceed',
5), ('any', 6), ('further,', 7), ('hear', 8), ('me', 9)]
单词： ['first', 'citizen:']
索引： [1, 2]
单词： ['before', 'we', 'proceed', 'any', 'further,', 'hear', 'me', 'speak.']
索引： [3, 4, 5, 6, 7, 8, 9, 10]
```

上述分词方法只是简单利用 split 函数进行分词，可能只适合类似英语这样用空格分隔的句子，不适合中文。即便是英文，也有一些诸如“doesn’t”“Dr.”和“New York”这样的词不容易处理好，更好的方法是利用现成的分词工具，下面介绍使用 spaCy 工具进行分词。

2.5.2 使用 torchtext

本小节演示如何使用 spaCy 工具对 Multi30k 数据集进行分词。由于该数据集有德语和英语两种语言，因此需要对应两种语言的分词器，为了简化编程，使用一个名称为 tokenizer 的 Python 字典来存放这两种语言的分词器。完整代码可参见 multi30k_torchtext.py。

代码 2.62 首先定义源语言和目标语言；然后定义 yield_tokens 函数，其功能就是使用指定语言的分词器进行分词。

代码 2.62 输出符号列表的辅助函数

```
# 源语言和目标语言
src_lang = 'de'
tgt_lang = 'en'


def yield_tokens(data_iter, language, tokenizer):
    """ 输出符号列表的辅助函数 """
    language_index = {src_lang: 0, tgt_lang: 1}
    # 迭代输出符号列表
    for sentence in data_iter:
        yield tokenizer[language](sentence[language_index[language]])
```

代码 2.63 首先加载 Multi30k 训练集并构建源语言和目标语言的词典；然后设置未登录词为<unk>符号；最后打印训练集的前 5 个源语言和目标语言的句子。可以看到，本例主要使用 torchtext 工具进行分词和构建词典，比手工编写预处理代码更简洁和标准。

代码 2.63 主函数

```
# 分词器和词典
tokenizer = {}
vocab = {}

# 创建源语言和目标语言的分词器，需要预装 spaCy 模块
tokenizer[src_lang] = get_tokenizer('spacy', language='de_core_news_sm')
tokenizer[tgt_lang] = get_tokenizer('spacy', language='en_core_web_sm')

# 定义 4 种特殊标记及对应索引。Unk 为未知，pad 为填充，bos 为序列开始，eos 为序列结束
special_symbols = ['<unk>', '<pad>', '<bos>', '<eos>']
unk_idx, pad_idx, bos_idx, eos_idx = 0, 1, 2, 3

# 迭代源语言和目标语言以构建词典
for lang in [src_lang, tgt_lang]:
    # 训练数据迭代器
    train_iter = Multi30k(root='../datasets', split='train',
language_pair=(src_lang, tgt_lang))
    # 从迭代器中构建 torchtext 的 Vocab 对象
    vocab[lang] = build_vocab_from_iterator(yield_tokens(train_iter, lang, tokenizer),
                                            min_freq=1,
                                            specials=special_symbols,
                                            special_first=True)

# 设置未登录词为'<unk>'
for lang in [src_lang, tgt_lang]:
    vocab[lang].set_default_index(unk_idx)
    print(f"{lang}语言的词典长度：{len(vocab[lang])}")

    print()
    print("打印前 5 个源语言和目标语言的句子：")
    train_iter = Multi30k(root='../datasets', split='train',
                  language_pair=(src_lang, tgt_lang))
for i in range(5):
    pair = next(train_iter)
    print(pair)
    for lang in [src_lang, tgt_lang]:
        lang_idx = {src_lang: 0, tgt_lang: 1}
        tokens = tokenizer[lang](pair[lang_idx[lang]])
        # 打印分词后的单词
        print(tokens)
        # 打印单词的词典序号
        print([vocab[lang].get_stoi()[w] for w in tokens])
    print()
```

程序运行结果如下：

```
de 语言的词典长度：19215
en 语言的词典长度：10838

打印前 5 个源语言和目标语言的句子：
('Zwei junge weiße Männer sind im Freien in der Nähe vieler Büsche.\n', 'Two young,
White males are outside near many bushes.\n')
['Zwei', 'junge', 'weiße', 'Männer', 'sind', 'im', 'Freien', 'in', 'der', 'Nähe',
'vieler', 'Büsche', '.', '\n']
[22, 86, 258, 32, 88, 23, 95, 8, 17, 113, 7911, 3210, 5, 4]
['Two', 'young', ',', 'White', 'males', 'are', 'outside', 'near', 'many', 'bushes',
'.', '\n']
[20, 26, 16, 1170, 809, 18, 58, 85, 337, 1340, 6, 5]

('Mehrere Männer mit Schutzhelmen bedienen ein Antriebsradsystem.\n', 'Several men
in hard hats are operating a giant pulley system.\n')
['Mehrere', 'Männer', 'mit', 'Schutzhelmen', 'bedienen', 'ein',
'Antriebsradsystem', '.', '\n']
[85, 32, 11, 848, 2209, 16, 8269, 5, 4]
['Several', 'men', 'in', 'hard', 'hats', 'are', 'operating', 'a', 'giant', 'pulley',
'system', '.', '\n']
[166, 37, 8, 336, 288, 18, 1225, 4, 759, 4497, 2958, 6, 5]

('Ein kleines Mädchen klettert in ein Spielhaus aus Holz.\n', 'A little girl climbing
into a wooden playhouse.\n')
['Ein', 'kleines', 'Mädchen', 'klettert', 'in', 'ein', 'Spielhaus', 'aus', 'Holz',
'.', '\n']
[6, 70, 28, 220, 8, 16, 6770, 56, 509, 5, 4]
['A', 'little', 'girl', 'climbing', 'into', 'a', 'wooden', 'playhouse', '.', '\n']
[7, 62, 34, 233, 72, 4, 254, 4461, 6, 5]

('Ein Mann in einem blauen Hemd steht auf einer Leiter und putzt ein Fenster.\n',
'A man in a blue shirt is standing on a ladder cleaning a window.\n')
['Ein', 'Mann', 'in', 'einem', 'blauen', 'Hemd', 'steht', 'auf', 'einer', 'Leiter',
'und', 'putzt', 'ein', 'Fenster', '.', '\n']
[6, 13, 8, 7, 48, 42, 31, 12, 14, 544, 10, 699, 16, 249, 5, 4]
['A', 'man', 'in', 'a', 'blue', 'shirt', 'is', 'standing', 'on', 'a', 'ladder',
'cleaning', 'a', 'window', '.', '\n']
[7, 13, 8, 4, 31, 24, 11, 38, 10, 4, 590, 587, 4, 243, 6, 5]

('Zwei Männer stehen am Herd und bereiten Essen zu.\n', 'Two men are at the stove
preparing food.\n')
['Zwei', 'Männer', 'stehen', 'am', 'Herd', 'und', 'bereiten', 'Essen', 'zu', '.',
'\n']
[22, 32, 54, 57, 1351, 10, 410, 175, 30, 5, 4]
['Two', 'men', 'are', 'at', 'the', 'stove', 'preparing', 'food', '.', '\n']
[20, 37, 18, 21, 9, 1204, 376, 135, 6, 5]
```

习 题

2.1 试运行张量数据操作的程序，熟悉各种张量操作。

2.2 试总结 PyTorch 的广播机制。

2.3 简述 PyTorch 中自动求导的概念。

2.4 查阅 torchvision 文档，了解 Transforms 模块的功能。

2.5 尝试修改 cifar10_normalize.py 程序，使之能够计算 MNIST 或 Fashion-MNIST 数据集的均值和标准差。

2.6 查阅 torchtext 文档，了解相应 API。

2.7 尝试修改 multi30k_torchtext.py 程序，使之能够对其他一些翻译数据集进行处理。