第 5 章

CHAPTER 5

PyTorch 进阶

> 人工智能将是谷歌的终极版本，即可以理解网络上所有内容的终极搜索引擎。它会准确地理解你想要什么，并给你正确的东西。
>
> ——拉里·佩奇

在介绍完张量的基本操作后，继续来进一步学习张量的进阶操作，如张量的合并与分割、范数统计、张量填充、张量限幅等。最后通过 MNIST 数据集的测试实战，来加深读者对 PyTorch 张量进阶操作的理解。

5.1 合并与分割

5.1.1 合并

合并是指将多个张量在某个维度上合并为一个张量。以某学校班级成绩册数据为例，设张量 $\boldsymbol{A}$ 保存了某学校 1～4 号班级的成绩册，每个班级 35 个学生，共 8 门科目成绩，则张量 $\boldsymbol{A}$ 的 shape 应为[4,35,8]；同理，张量 $\boldsymbol{B}$ 保存了其他 6 个班级的成绩册，shape 为[6,35,8]。通过合并这两份成绩册，便可得到该学校所有班级的成绩册数据，记为张量 $\boldsymbol{C}$，它的 shape 应为[10,35,8]，其中，数字 10 代表 10 个班级，35 代表 35 个学生，8 代表 8 门科目。这就是张量合并操作的意义所在。

张量的合并可以使用拼接(Concatenate)和堆叠(Stack)操作实现，拼接操作并不会产生新的维度，仅在现有的维度上合并，而堆叠会创建新维度并合并数据。选择使用拼接还是堆叠操作来合并张量，取决于具体的场景是否需要创建新维度。下面来介绍拼接操作和堆叠操作的典型应用场景和使用方法。

(1) 拼接：在 PyTorch 中，可以通过 torch.cat(tensors,dim)函数拼接张量，其中参数

tensors 保存了所有需要合并的张量 List，dim 参数指定需要合并的维度索引。回到上面的例子，需要在班级维度上合并成绩册，这里班级维度索引号为 0，即 dim＝0，合并张量 ***A*** 和 ***B*** 的代码如下。

```
In [1]:
a = torch.randn([4,35,8])                    #模拟成绩册 A,等价于 torch.randn(4,35,8)写法
b = torch.randn([6,35,8])                    #模拟成绩册 B
c = torch.cat([a,b],dim = 0)                 #拼接合并成绩册
c.shape
Out[1]:
torch.Size([10, 35, 8])
```

除了可以在班级维度上进行拼接合并，还可以在其他维度上拼接合并张量。考虑张量 ***A*** 保存了所有班级的所有学生的前 4 门科目成绩，shape 为[10,35,4]，张量 ***B*** 保存了剩下的 4 门科目成绩，shape 为[10,35,4]，则可以拼接合并 shape 为[10,35,8]的总成绩册张量，实现如下。

```
In [2]:
a = torch.randn([10,35,4])
b = torch.randn([10,35,4])
c = torch.cat([a,b], dim = 2)                #在科目维度上拼接
c.shape
Out[2]:
torch.Size([10, 35, 8])
```

从语法上来说，拼接合并操作可以在任意的维度上进行，唯一的约束是所有非合并维度的长度必须一致。比如 shape 为[4,32,8]和 shape 为[6,35,8]的张量不能直接在班级维度 dim＝0 上进行合并，因为学生数量维度 dim＝1 的长度并不一致，一个为 32，另一个为 35，代码如下。

```
In [3]:
a = torch.randn([4,32,8])
b = torch.randn([6,35,8])
torch.cat([a,b], dim = 0)                    #非法拼接,其他维度长度不相同
Out[3]:
RuntimeError: Sizes of tensors must match except in dimension 0. Got 32 and 35 in dimension 1
```

(2) 堆叠：拼接操作直接在现有维度上合并数据，并不会创建新的维度。如果在合并数据时，希望创建一个新的维度，则需要使用 stack 堆叠操作。考虑张量 ***A*** 保存了某个班级的成绩册，shape 为[35,8]，张量 ***B*** 保存了另一个班级的成绩册，shape 为[35,8]。合并这两个班级的数据时，则需要创建一个新维度，定义为班级维度，新维度可以选择放置在任意位置，一般根据大小维度的经验法则，将较大概念的班级维度放置在学生维度之前，则合并后的张量的新 shape 应为[2,35,8]，其中 2 代表两个班级。

使用 torch.stack(tensors,dim)可以以堆叠方式合并多个张量，通过 tensors 列表表示，参数 dim 指定新维度插入的位置，dim 的用法与 torch.unsqueeze 函数的一致，当 dim≥0

时，在 dim 之前插入；当 dim<0 时，在 dim 之后插入新维度。例如 shape 为[b,c,h,w]的张量，在不同位置通过 stack 操作插入新维度时，dim 参数对应的插入位置设置如图 5.1 所示。

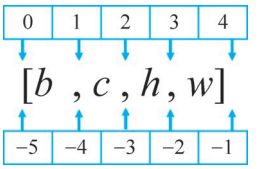

图 5.1 stack 插入维度位置示意图

通过堆叠方式合并这两个班级成绩册，班级维度插入在 dim=0 位置，代码如下。

```
In [4]:
a = torch.randn([35,8])
b = torch.randn([35,8])
c = torch.stack([a,b], dim = 0)          #堆叠合并为 2 个班级,班级维度插入在最前
c.shape
Out[4]:
torch.Size([2, 35, 8])
```

同样可以选择在其他位置插入新维度，例如，最末尾插入班级维度，代码如下。

```
In [5]:
a = torch.randn([35,8])
b = torch.randn([35,8])
c = torch.stack([a,b], dim = 2)          #堆叠合并为 2 个班级,班级维度插入在末尾
c.shape
Out[5]:
torch.Size([35, 8, 2])
```

此时班级的维度在 dim=2 轴上面，理解时也需要按着最新的维度顺序代表的视图去理解数据。对于这个例子，若选择使用 cat 拼接合并上述成绩单，则可以合并为

```
In [6]:
a = torch.randn([35,8])
b = torch.randn([35,8])
c = torch.cat([a,b], dim = 0)            #拼接方式合并,没有 2 个班级的概念
c.shape
Out[6]:
torch.Size([70, 8])
```

可以看到，cat 函数也可以顺利合并数据，但是在理解时，需要按照前 35 个学生来自第一个班级，后 35 个学生来自第二个班级的方式理解张量数据。对这个例子，明显通过 stack 方式创建新维度的方式更合理，得到的 shape 为[2,35,8]的张量也更容易理解。

Stack 操作也需要满足张量堆叠合并的条件，它需要所有待合并的张量 shape 完全一致才可合并。来看张量 shape 不一致时进行堆叠合并发生的错误，举例如下。

```
In [7]:
a = torch.randn([35,4])
b = torch.randn([35,8])
torch.stack([a,b], dim = -1)             #非法堆叠操作,张量 shape 不相同
Out[7]:
RuntimeError: stack expects each tensor to be equal size, but got [35, 4] at entry 0 and [35, 8]
at entry 1
```

上述操作尝试合并 shape 为[35,4]和[35,8]的两个张量，由于两者 shape 不一致，无法完成合并操作。

5.1.2 分割

合并操作的逆过程就是分割，即将一个张量拆分为多个张量。继续考虑上述成绩册的例子，通过合并操作可得到整个学校的成绩册张量，shape 为[10,35,8]，现在需要将数据在班级维度上切割为 10 个张量，每个张量保存了对应班级的成绩册数据，这样即可分别获得每个班级的成绩册数据。

通过 torch.split(x,split_size_or_sections,dim)可以完成张量的分割操作，参数意义定义如下。

(1) x 参数：待分割张量。

(2) split_size_or_sections 参数：切割方案。当 split_size_or_sections 为单个数值时，表示每份的长度，如 2，表示每份长度为 2，即等长切割为 10 份；当 split_size_or_sections 为 List 时，List 的每个元素表示每份的长度，如[2,4,2,2]表示切割为 4 份，每份的长度依次是 2、4、2、2。

(3) dim 参数：指定待分割的维度索引号。

现在将总成绩册张量切割为 10 份，代码如下。

```
In [8]:
x = torch.randn([10,35,8])
#等长切割为 10 份,每份长度为 1
result = torch.split(x, split_size_or_sections = 1, dim = 0)
len(result)                                    #返回的列表为 10 个张量的列表
Out[8]:10
```

可以查看切割后的某个张量的形状，它应是某个班级的所有成绩册数据，shape 为[35,8]，举例如下。

```
In [9]:result[0]                               #查看第一个班级的成绩册张量
Out[9]:#shape = (1, 35, 8)
tensor([[[ -1.7786729 , 0.2970506 , 0.02983334, 1.3970423 ,
1.315918 , -0.79110134, -0.8501629 , -1.5549672 ],
       [ 0.5398711 , 0.21478991, -0.08685189, 0.7730989 , …])
```

可以看到，切割后的班级 shape 为[1,35,8]，仍保留了班级维度，这一点需要注意。

现在来进行不等长的切割，例如，将数据切割为 4 份，每份长度分别设计为[4,2,2,2]，实现如下。

```
In [10]:
x = torch.randn([10,35,8])
#自定义长度的切割,切割为 4 份,返回 4 个张量的列表 result
result = torch.split(x, [4,2,2,2] , dim = 0)
len(result)
Out[10]:4
```

查看第一个张量的 shape，根据上述的切割方案，它应该包含了 4 个班级的成绩册，shape 应为[4,35,8]，验证如下。

```
In [10]:
result[0]
Out[10]:
#torch.Size([4, 35, 8])
tensor([[[-6.95693314e-01, 3.01393479e-01, 1.33964568e-01, ...,]]])
```

除了 split 函数可以实现张量分割外，PyTorch 还提供了另一个函数 torch.chunk。它的用法与 torch.split 非常类似，区别在于 chunk 函数的参数 chunks 指定了切割份数，而 split 函数的参数 split_size_or_sections 则是每份长度，本质上两个函数是等价的。例如，将总成绩册张量在班级维度进行 chunk 操作，等分为 2 份，代码如下。

```
In [11]:
x = torch.randn([10,35,8])
a,b = torch.chunk(x, chunks=2, dim=0)          #等分为 2 份
a.shape, b.shape
Out[11]:
(torch.Size([5, 35, 8]), torch.Size([5, 35, 8]))
```

将总成绩册张量在班级维度进行 chunk 操作，等分为 10 份，代码如下。

```
In [12]:
x = torch.randn([10,35,8])
result = torch.chunk(x, chunks=10, dim=0)      #等分为 10 份
len(result), result[0].shape
Out[12]:
(10, torch.Size([1, 35, 8]))
```

可以看到，torch.chunk 函数完成的功能与 torch.split 完全一样。

此外，torch.unbind(x,dim)函数也具有分割张量的功能，它沿着 dim 维度按长度为 1 的方式将张量分割成长度为 1 的每份。例如，shape 为[10,35,8]的张量，沿着 dim=0 维度进行 unbind 切分，则获得 10 个 shape 为[35,8]的张量。


第 23 集
微课视频


第 24 集
微课视频

5.2 数据统计

在神经网络的计算过程中，经常需要统计数据的各种属性，如最值、最值元素所在位置、均值、范数等信息。由于张量维度通常较大，直接观察数据很难获得有用信息，因此通过获取这些张量的统计信息可以较轻松地推测张量数值的分布。下面介绍一些常用的张量统计函数。

5.2.1 向量范数

向量范数(Vector Norm)是表征向量“长度”的一种度量方法，它可以推广到张量上。

在神经网络中，常用来表示张量的权值大小、梯度大小等。常用的向量范数有

（1）L1 范数，定义为向量 $\boldsymbol{x}$ 的所有元素绝对值之和，即

$$\|\boldsymbol{x}\|_1 = \sum_i |x_i|$$

（2）L2 范数，定义为向量 $\boldsymbol{x}$ 的所有元素的平方和，再开根号，即

$$\|\boldsymbol{x}\|_2 = \sqrt{\sum_i |x_i|^2}$$

（3）∞范数，定义为向量 $\boldsymbol{x}$ 的所有元素绝对值的最大值，即

$$\|\boldsymbol{x}\|_\infty = \max_i(|x_i|)$$

对于矩阵和张量，同样可以利用向量范数的计算思想，等价于将矩阵和张量打平成向量后计算，统称为向量范数。

在 PyTorch 中，可以通过 torch.norm(x,p,dim=None)求解张量的 L1、L2、∞等范数，其中参数 p 指定为 1、2 时计算 L1、L2 范数，指定为 np.inf 时计算∞范数，举例如下。

```
In [13]:
x = torch.ones([2,2])
torch.norm(x, p=1)                    # 计算 L1 范数
Out[13]:
tensor(4.)
In[14]:
torch.norm(x, p=2)                    # 计算 L2 范数
Out[14]:
tensor(2.)
In [15]:
import numpy as np
torch.norm(x, p=np.inf)               # 计算∞范数
Out[15]:
tensor(1.)
```

在神经网络调试的过程中，通常需要在合适的地方查看张量的数值，直接打印张量并不合适。通常打印张量的范数即可大致推测张量的数值范数。例如，当梯度张量的 L2 范数较大时，可以推测梯度张量的部分元素或整体元素较大，容易出现梯度爆炸现象。

5.2.2 最值、均值、和

通过 torch.max(x,dim)、torch.min(x,dim)、torch.mean(x,dim)、torch.sum(x,dim)函数可以求解张量在某个 dim 维度上的最大值、最小值、均值以及和，也可以求全局最大值、最小值、均值以及和。不提供 dim 参数时即计算全局的最大值、最小值、均值以及和。

考虑 shape 为[4,10]的张量，其中，第一个维度 4 代表样本数量，第二个维度 10 代表了当前样本分别属于 10 个类别的概率。需要求出每个样本的概率最大值，则可以通过 max 函数实现，代码如下。

```
In [16]:
x = torch.randn([4,10])               # 模拟数据
```

```
x = torch.softmax(x, dim=1)                    #转换为[0,1]区间的概率
value, idx = torch.max(x, dim=1)               #统计概率维度上的最大值
value, idx                                     #打印最值和最值所在位置
Out[16]:
(tensor([0.2243, 0.4890, 0.2667, 0.2982]), tensor([7, 3, 8, 3]))
```

Torch.max 函数返回为元组(Tuple),包含了两个元素。第一个元素为长度为 4 的向量,代表了每个样本的最大概率值;第二个元素为长度为 4 的整型向量,代表了最大值元素出现的位置索引。

同样求出每个样本概率的最小值,实现如下。

```
In[17]:
value, idx = x.min(dim=1)                      #统计概率维度上的最小值
value, idx                                     #打印最值和最值所在位置
Out[17]:
(tensor([0.0102, 0.0176, 0.0091, 0.0215]), tensor([1, 4, 6, 8]))
```

求出每个样本的概率的均值,实现如下。

```
In[18]:
value= x.mean(dim=1)                           #统计概率维度上的均值
value                                          #打印均值
Out[18]:
tensor([0.1000, 0.1000, 0.1000, 0.1000])
```

当不指定 dim 参数时,上述函数会求解出全局元素的最大值、最小值、均值以及和等数据,举例如下。

```
In[19]:
x = torch.randn([4,10])
#统计全局的最大值、最小值、均值以及和,返回的张量均为标量
x.max(),x.min(),x.mean(),x.sum()
Out[19]:
(tensor(2.3259), tensor(-1.9073), tensor(0.1330), tensor(5.3181))
```

在求解神经网络的平均误差时,需要计算目标值与预测值的差的平方和,再计算样本上的平均误差。首先计算目标值与预测值的差的平方和,实现如下。

```
In[20]:
from torch.nn import functional as F
out = torch.randn([4,10])                      #模拟网络预测输出
y = torch.tensor([1,2,2,0])                    #模拟真实标签
y = F.one_hot(y, num_classes=10)               #One-hot 编码
loss = torch.square(y-out)                     #计算差的平方
loss = loss.sum(dim=1)                         #求各个样本的误差平方和
loss
Out[20]:
tensor([11.3666, 4.0145, 6.6415, 8.9286])
```

通过 sum 函数求和后,再计算样本维度的均值,即可得到平均误差。可以通过 mean 函数实现,它可以求解张量在 dim=0 轴上所有特征的平均值,实现如下。

```
In [21]:
loss = loss.mean(dim = 0)                    #计算样本维度上的平均误差
loss                                         #打印误差
Out[21]:
tensor(7.7378)
```

除了希望获取张量的最值信息，有时还需要获得最值元素所在的位置索引号，例如分类任务的标签预测，就需要知道概率最大值所在的位置索引号，并把这个位置索引号作为预测的类别。考虑 10 分类问题，可以得到神经网络的输出张量 out，其 shape 为[2，10]，代表了两个样本属于 10 个类别的概率，由于元素的位置索引代表了当前样本属于此类别的概率，预测时往往会选择概率值最大的元素所在的索引号作为样本类别的预测值，举例如下。

```
In [22]:
x = torch.randn([2,10])                      #模拟数据
x = torch.softmax(x, dim = 1)                #转换为[0, 1]区间的概率
idx = torch.argmax(x, dim = 1)               #统计概率维度上的最大值
x,idx                                        #打印网络输出 x 和预测类别值
Out[22]:
tensor([[0.0542, 0.0816, 0.0259, 0.0273, 0.0655, 0.1295, 0.0131, 0.3547, 0.0854,0.1628],
        [0.0461, 0.1030, 0.1025, 0.0075, 0.2999, 0.1521, 0.0217, 0.0116, 0.0791,0.1766]]),
tensor([7, 4])
```

以第一个样本为例，可以看到，它概率最大值的索引为 i=7，最大概率值为 0.3547。由于每个索引号上的概率值代表了样本属于此索引号的类别的概率，因此第一个样本属于 0 类的概率最大，在预测时第一个样本应最有可能属于类别 0。这就是需要求解最大值的索引号的一个典型应用。

通过 torch.argmax(x,dim)和 torch.argmin(x,dim)可以求解在 dim 轴上，x 的最大值、最小值所在的索引号，例如：

```
In [23]:
pred = x.argmin(dim = 1)                     #选取概率最小的位置
pred
Out[23]:
tensor([6, 3])
```

可以看到，概率最小值出现在索引 6 和索引 3 上，读者可自行验证这两个元素是否是最小值。

5.3 张量比较

为了计算分类任务的准确率等指标，通常需要将预测结果和真实标签比较，统计比较结果中正确的数量来计算准确率。考虑 100 个样本的预测结果，通过 torch.argmax 获取预测类别，实现如下。

```
In [24]:
```

```
out = torch.randn([100,10])
out = F.softmax(out, dim=1)          #输出转换为概率
pred = torch.argmax(out, dim=1)      #计算预测值
pred                                 #打印预测结果
Out[24]:
tensor([4, 4, 8, 2, 5, 8, 1, 8, 3, 5, 0, 4, 7, 5, 1, 8, 7, 0, 2, 5, 7, 7, 8, 5,
        7, 1, 1, 2, 4, 9, 1, 3, 9, 9, 1, 2, 9, 4, 1, 6, 1, 1, 4, 9, 5, 5, 8, 7,
        2, 4, 4, 0, 9, 1, 9, 5, 9, 2, 1, 6, 8, 6, 8, 0, 3, 9, 9, 0, 6, 6, 7, 8,...])
```

变量 pred 保存了这 100 个样本的预测类别值，需要与这 100 个样本的真实标签 y 进行比较，举例如下。

```
In[25]:
y = torch.randint(0, 10, [100])      #模拟生成真实标签
y
Out[25]:
tensor([1, 4, 8, 3, 0, 9, 2, 3, 8, 7, 3, 8, 0, 3, 1, 5, 0, 9, 7, 6, 0, 1, 0, 0,
        8, 7, 4, 2, 9, 5, 9, 0, 9, 7, 4, 5, 0, 8, 1, 0, 1, 8, 4, 9, 5, 4, 9, 3,
        6, 5, 3, 8, 0, 8, 4, 0, 6, 3, 4, 4, 4, 1, 0, 0, 2, 7, 7, 4, 4, 2, 6, 5,...])
```

即可获得代表每个样本是否预测正确的布尔类型张量。通过 torch.eq(a,b)函数可以比较这 2 个张量是否相等，举例如下。

```
In[26]:
res = torch.eq(y, pred)              #预测值与真实值比较，返回布尔类型的张量
res                                  #打印比较结果
Out[26]:
tensor([False, True, True, False, False, False, False, False, False, False,
        False, False, False, False, True, False, False, False, False, False,
        False, False,...])
```

torch.eq()函数返回布尔类型的张量比较结果，只需要统计张量中 True 元素的个数，即可知道预测正确的个数。为了达到这个目的，先将布尔类型转换为整型张量，即 True 对应为 1，False 对应为 0，再求和其中元素为 1 的个数，就可以得到比较结果中 True 元素的个数，代码如下。

```
In[27]:
res = res.int()                      #布尔型转 int 型
correct = res.sum()                  #统计 True 的个数
Out[27]:
tensor(13)
```

可以看到，此处随机产生的预测数据中预测正确的个数是 13，因此它的准确率是

$$\text{accuracy}=\frac{13}{100}=13\%$$

这也是随机预测模型的正常水平。

除了比较相等的 torch.eq(a,b)函数，其他的比较函数用法类似，如表 5.1 所示。

表 5.1 常用比较函数总结

函　　数	比 较 逻 辑
torch. gt	$a>b$
torch. lt	$a<b$
torch. ge	$a\geqslant b$
torch. le	$a\leqslant b$
torch. ne	$a\neq b$
torch. isnan	$a=\text{nan}$

此外，torch. equal(a,b)函数与 torch. eq(a,b)函数是不同的，它并不是逐元素的比较操作，而是比较两个张量是否完全相等，返回一个布尔标量。

5.4 填充与复制

本节介绍张量的填充操作和复制操作。

5.4.1 填充

对于图片数据的高和宽、序列信号的长度等，每个样本的维度长度可能各不相同。为了方便网络的并行运算，通常需要将不同长度的数据扩张为相同长度，之前介绍了通过复制的方式可以增加数据的长度，但是重复复制数据会破坏原有的数据结构，并且复制数据只能以倍数方式增加，并不适合于此处。为了统一不同样本数据的长度，通常的做法是，在需要补充长度的数据开始或结束处填充足够数量的特定数值，这些特定数值一般代表了无效意义，例如数字 0，使得填充后的长度满足模型要求。这种操作就叫作填充(Padding)操作。

考虑两个句子张量，每个单词使用数字编码方式表示，如 1 代表单词 I、2 代表单词 like 等。第一个句子为

“I like the weather today.”

假设句子数字编码为[1,2,3,4,5,6]，第二个句子为

“So do I.”

它的编码为[7,8,1,6]。为了能够将这两个句子保存在同一个张量中，需要将这两个句子的长度保持一致，也就是说，需要将第二个句子的长度扩充为 6。常见的填充方案是在句子末尾填充若干数量的 0，变成

[7,8,1,6,0,0]

此时这两个句子可通过堆叠操作合并成 shape 为[2,6]的张量。

填充操作可以通过 F. pad(x,pad)函数实现(F 代表 torch. nn. functional 模块，下文同)，参数 pad 是包含了多个[Left Padding,Right Padding]的嵌套方案 List，并且从最后一个维度开始制定，如[0,0,2,1,1,2]表示倒数第一个维度首部填充 0 个单元、尾部填充 0 个单元，倒数第二个维度首部填充两个单元、尾部填充一个单元，倒数第三个维度首部填充一

个单元、尾部填充两个单元。特别注意这里 pad 参数指定的填充格式是从最末维度开始的。

考虑上述两个句子的例子，需要在第二个句子的第一个维度的右边填充 2 个单元，则填充方案为[0,2]，代码如下。

```
In [28]:
a = torch.tensor([1,2,3,4,5,6])          #第一个句子
b = torch.tensor([7,8,1,6])              #第二个句子
b = F.pad(b, [0,2])                      #句子末尾填充 2 个 0
b                                        #填充后的结果
Out[28]:
tensor([7, 8, 1, 6, 0, 0])
```

填充后句子张量形状一致，再将这两个句子堆叠在一起，代码如下。

```
In [29]:
torch.stack([a,b], dim = 0)              #堆叠合并，创建句子数维度
Out[29]:
tensor([[1, 2, 3, 4, 5, 6],
        [7, 8, 1, 6, 0, 0]])>
```

在自然语言处理中，往往需要加载不同句子长度的数据集，有些句子长度较小，如仅 10 个单词，而部分句子长度较长，如超过 100 个单词。为了能够将多个句子保存在同一张量中，一般会选取能够覆盖大部分句子长度的阈值，如 80 个单词，来统一句子的长度。对于小于 80 个单词的句子，在末尾填充相应数量的 0；对大于 80 个单词的句子，则截断超过规定长度的部分单词。以 IMDB 数据集的加载为例，下面来演示如何将不等长的句子变换为等长结构，代码如下。

```
In [30]:
total_words = 10000                      #设定词汇量大小
max_review_len = 80                      #最大句子长度
embedding_len = 100                      #词向量长度
#加载 IMDB 数据集
(x_train, y_train), (x_test, y_test) = keras.datasets.imdb.load_data(num_words = total_words)
#将句子填充或截断到相同长度，设置为末尾填充和末尾截断方式
x_train = keras.preprocessing.sequence.pad_sequences(x_train, maxlen = max_review_len,
truncating = 'post',padding = 'post')
x_test = keras.preprocessing.sequence.pad_sequences(x_test, maxlen = max_review_len,
truncating = 'post',padding = 'post')
print(x_train.shape,x_test.shape)        #打印等长的句子张量形状
Out[30]:(25000, 80) (25000, 80)
```

上述代码中，将句子的最大长度 max_review_len 设置为 80 个单词，通过 keras.preprocessing.sequence.pad_sequences 函数可以快速完成句子的填充和截断工作，以其中某个句子为例，观察其变换后的向量内容为

```
[  1   778   128    74    12   630   163    15     4  1766  7982  1051     2    32
  85   156    45    40   148   139   121   664   665    10    10  1361   173     4
 749     2    16  3804     8     4   226    65    12    43   127    24     2    10
  10     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]
```

可以看到在句子末尾填充了若干数量的 0，使得句子的长度刚好为 80。实际上，也可以选择当句子长度不够时，在句子前面填充 0；句子长度过长时，截断句首的单词。经过处理后，所有的句子长度都变为 80，从而训练集可以统一保存在 shape 为[25000,80]的张量中，测试集可以保存在 shape 为[25000,80]的张量中。

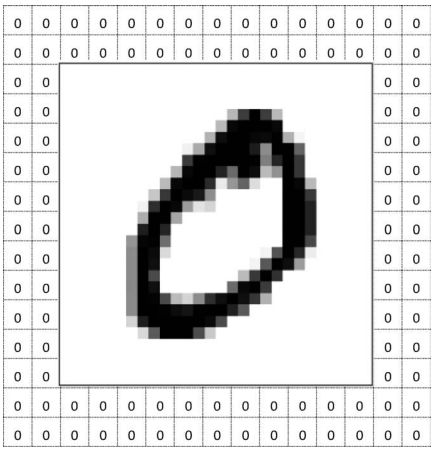

图 5.2　图片填充示意图

下面来介绍同时在多个维度进行填充的例子。考虑对图片的高宽维度进行填充。以 28×28 大小的图片数据为例，如果网络层所接受的数据高宽为 32×32，则必须将 28×28 大小填充到 32×32，可以选择在图片矩阵的上、下、左、右方向各填充 2 个单元，如图 5.2 所示。

上述填充方案可以表达为倒数第一个维度填充[2,2]，倒数第二个维度填充[2,2]，则填充参数设置为[2,2,2,2]，实现如下。

```
In [31]:
x = torch.randn([4,1, 28,28])          #28x28 大小灰度图片
#图片上下、左右各填充 2 个单元
x2 = F.pad(x, [2,2,2,2])
x2[0,0]                                #打印其中一个矩阵
Out[31]:
tensor([[ 0.0000, 0.0000, 0.0000, ..., 0.0000, 0.0000, 0.0000],
        [ 0.0000, 0.0000, 0.0000, ..., 0.0000, 0.0000, 0.0000],
        [ 0.0000, 0.0000, -2.3530, ..., 1.0238, 0.0000, 0.0000],
        ...,
        [ 0.0000, 0.0000, 0.2458, ..., -1.3867, 0.0000, 0.0000],
        [ 0.0000, 0.0000, 0.0000, ..., 0.0000, 0.0000, 0.0000],
        [ 0.0000, 0.0000, 0.0000, ..., 0.0000, 0.0000, 0.0000]])
```

通过填充操作后，所有高宽的两边均填充两个单位的 0，图片的大小变为 32×32，满足神经网络的输入要求。

5.4.2　复制

在“4.7 维度变换”一节，介绍了通过 torch.repeat(repeats)函数实现长度为 1 的维度复制的功能。torch.repeat 函数除了可以复制若干份长度为 1 的维度，还可以复制若干份任意长度的维度，进行复制时会根据原来的数据次序重复复制。由于前面已经介绍过，此处仅作简单回顾。

通过 torch.repeat(repeats)函数可以在任意维度将数据重复复制多份，如 shape 为[4,3,32,32]的数据，复制方案为 repeats =[2,1,3,3]，即通道数据不复制，高和宽方向分别复制 2 份，图片数再复制 1 份，实现如下。

```
In [32]:
x = torch.randn([4,3,32,32])
x2 = x.repeat([2,1,3,3])                    #数据复制
x2.shape
Out[32]:
torch.Size([8, 3, 96, 96])
```

5.5 数据限幅

现在来考虑怎么实现非线性激活函数 ReLU 的问题。它其实可以通过简单的数据限幅运算实现，只需要限制元素的范围 $x\in[0,+\infty)$即可。

在 PyTorch 中，可以通过 torch.max(x,a)实现数据的下限幅，即 $x\in[a,+\infty)$；可以通过 torch.min(x,a)实现数据的上限幅，即 $x\in(-\infty,a]$。举例如下。

```
In [33]:
x = torch.arange(9)
torch.max(x, torch.tensor(2))               #与 2 进行比较，取较大值
Out[33]:
tensor([2, 2, 2, 3, 4, 5, 6, 7, 8])
In [34]:
torch.min(x, torch.tensor(7))               #上限幅为 7
Out[34]:
tensor([0, 1, 2, 3, 4, 5, 6, 7, 7])
```

基于 torch.max 函数，可以实现 ReLU 函数如下。

```
def relu(x):                                #ReLU 函数
    return torch.max(x, torch.tensor(0))      #下限幅为 0 即可
x = torch.arange(-4, 4)                     #创建输入张量
print('x:', x)                              #打印输入 tensor([-4, -3, -2, -1, 0, 1, 2, 3])
print('relu:', relu(x))                     #打印 relu 输出 tensor([0, 0, 0, 0, 0, 1, 2, 3])
```

需要注意的是，torch.max(x,dim)函数调用为求解 dim 维度上的最大值，此时 dim 应为整型数字。torch.max(x,a)方式的 x 和 a 参数皆为张量，这两种调用方式需要特别注意区分。

通过组合 torch.max(x,a)和 torch.min(x,b)可以实现同时对数据的上下边界限幅，即 $x\in[a,b]$，举例如下。

```
In [35]:
x = torch.arange(9)
torch.min(torch.max(x, torch.tensor(2)), torch.tensor(7))    #限幅为 2～7
Out[35]:
tensor([2, 2, 2, 3, 4, 5, 6, 7, 7])
```

更方便地，可以使用 torch.clamp 函数实现张量的上下限幅，代码如下。

```
In [36]:
```

```
x = torch.arange(9)
torch.clamp(x, 2, 7)                    #限幅为 2～7
Out[36]:
tensor([2, 2, 2, 3, 4, 5, 6, 7, 7])
```

5.6 高级操作

上述介绍的操作函数大部分是常用并且容易理解的，接下来将介绍部分常用，但是稍复杂的功能函数。

5.6.1 索引采样

torch.index_select()函数可以实现根据索引号收集数据的目的。考虑班级成绩册的例子，假设共有 4 个班级，每个班级 35 个学生，8 门科目，则保存成绩册的张量 shape 为[4,35,8]。随机创建张量如下。

```
x = torch.randint(0,100,[4,35,8])      #成绩册张量
```

现在需要收集第 1～2 个班级的成绩册，可以给定需要收集班级的索引号：[0,1]，并指定班级的维度 dim=0，通过 torch.index_select()函数收集数据，代码如下。


第 25 集
微课视频

```
In [38]:
#选择班级维度的 0,1 号班级
out = torch.index_select(x, dim = 0, index = torch.tensor([0,1]))
out.shape
Out[38]:
torch.Size([2, 35, 8])
```

实际上，对于上述需求，通过切片 $x[:2]$操作可以更加方便地实现。但是对于不规则的索引方式，比如，需要抽查所有班级的第 1、4、9、12、13、27 号同学的成绩数据，则切片方式实现起来非常麻烦，而 torch.index_select 则是针对此需求设计的，使用起来更加方便，实现如下。

```
In [39]:
#收集第 1,4,9,12,13,27 号同学成绩
out = torch.index_select(x, dim = 1, index = torch.tensor([0,3,8,11,12,26]))
out.shape
Out[39]:
torch.Size([4, 6, 8])
```

如果需要收集所有同学的第 3 和第 5 门科目的成绩，则可以指定科目维度 dim=2，实现如下。

```
In [40]:
#第 3,5 科目的成绩
out = torch.index_select(x, dim = 2, index = torch.tensor([2, 4]))
```

```
out.shape
Out[40]:
torch.Size([4, 35, 2])
```

可以看到，torch.index_select 函数非常适合索引号没有规则的场合，并且索引号可以乱序排列，此时收集的数据也是对应顺序排列，举例如下。

```
In [41]:
a = torch.arange(8)
a = a.reshape([4,2])                    #生成张量 a
Out[41]:
tensor([[0, 1],
        [2, 3],
        [4, 5],
        [6, 7]])
In [42]:
#收集第 4,2,1,3 号元素
torch.index_select(a, dim = 0, index = torch.tensor([3,1,0,2]))
Out[42]:
tensor([[6, 7],
        [2, 3],
        [0, 1],
        [4, 5]])
```

现在将问题变得稍复杂一点。如果希望抽查第[2,3]班级的第[3,4,6,27]号同学的科目成绩，则可以通过组合多个 torch.index_select 函数实现。首先采样第[2,3]班级，实现如下。

```
In [43]:
x = torch.randint(0,100,[4,35,8])      #成绩册张量
#收集第 2,3 号班级
students = x.index_select(dim = 0, index = torch.tensor([1,2]))
students.shape
Out[43]:
torch.Size([2, 35, 8])
```

再从这两个班级的同学中提取对应学生成绩，代码如下：

```
In [44]:
#基于 students 张量继续收集指定编号学员
students.index_select(1, torch.tensor([2,3,5,26]))
Out[44]:
tensor([[[60, 15, 86, 81, 58, 0, 21, 60],
         [10, 4, 2, 75, 42, 66, 56, 25],
         [16, 60, 46, 78, 1, 9, 59, 51],
         [77, 6, 21, 33, 0, 72, 34, 36]],
        [[15, 25, 91, 33, 26, 98, 70, 50],
         [67, 43, 52, 33, 19, 3, 20, 49],
         [54, 6, 61, 98, 14, 99, 26, 14],
         [65, 82, 96, 15, 8, 65, 72, 12]]])
```

此时得到这两个班级的 4 个同学的成绩张量，shape 为[2,4,8]。

继续将问题进一步复杂化。这次希望抽查第 2 个班级的第 2 个同学的所有科目，第 3 个班级的第 3 个同学的所有科目，第 4 个班级的第 4 个同学的所有科目。那么怎么实现呢？

可以通过“笨方法”，一个一个地手动提取数据。首先提取第一个采样点的数据：$x[1,1]$，可得到 8 门科目的数据向量，代码如下。

```
In [45]:
x[1,1]                                  #收集第 2 个班级的第 2 个同学
Out[45]:
tensor([95, 54, 34, 29, 52, 21, 9, 91])
```

再串行提取第二个采样点的数据 $x[2,2]$，以及第三个采样点的数据 $x[3,3]$，最后通过 stack 方式合并采样结果，实现如下。

```
In [46]:
print('1:', torch.stack([x[1,1],x[2,2],x[3,3]], dim=0))    #方式一
idx = torch.tensor([[1,1],[2,2],[3,3]]).long()
print('2:', x[idx[:,0], idx[:,1]])                          #方式二
Out[46]:
#方式一
tensor([[95, 54, 34, 29, 52, 21, 9, 91],
        [15, 25, 91, 33, 26, 98, 70, 50],
        [71, 21, 84, 24, 88, 50, 84, 71]])
#方式二
tensor([[95, 54, 34, 29, 52, 21, 9, 91],
        [15, 25, 91, 33, 26, 98, 70, 50],
        [71, 21, 84, 24, 88, 50, 84, 71]])
```

方式一也能正确地得到 shape 为[3,8]的结果，其中 3 表示采样点的个数，8 表示每个采样点的数据长度，看上去似乎也不错。方式二更加简洁，将坐标信息表示在张量 idx 中，然后通过分别知道每个维度坐标信息来采样数据。方式二中的采样方式与 TensorFlow 框架中的 gather_nd 函数是类似的。

实际上，通过[]索引方式既可以实现 index_select 函数，又可以实现多维坐标索引方式，更值得推荐。

5.6.2 掩码采样

除了可以通过给定索引号的方式采样，还可以通过给定掩码(Mask)的方式进行采样。继续以 shape 为[4,35,8]的成绩册张量为例，这次以掩码方式进行数据提取。

考虑在班级维度上进行采样，对这 4 个班级的采样方案的掩码为

$$\text{mask}=[\text{True},\text{False},\text{False},\text{True}]$$

即采样第 1 个和第 4 个班级的数据，通过 x[mask]索引方式可以在指定轴上根据 mask 方案进行采样，实现为

```
In [47]:                                #根据掩码方式采样班级，给出掩码和维度索引
```

```
x = torch.randint(0, 9, [4,35,8])
x[mask].shape                          #选择班级维度,根据掩码
Out[47]:
torch.Size([2, 35, 8])
```

注意掩码的长度必须与对应维度的长度一致,如在班级维度上采样,则必须对这 4 个班级是否采样的掩码全部指定,掩码长度为 4。

如果对 8 门科目进行掩码采样,设掩码采样方案为

$$\text{mask}=[\text{True},\text{False},\text{False},\text{True},\text{True},\text{False},\text{False},\text{True}]$$

即采样第 1、4、5、8 门科目,则可以实现为

```
In[48]:                                #根据掩码方式采样科目
mask = torch.tensor([True,False,False,True,True,False,False,True])
x[:, :, mask].shape                    #选择科目维度,根据掩码
Out[48]:
torch.Size([4, 35, 4])
```

不难发现,这种通过 Mask 方式的用法其实与 index_select 非常类似,只不过一个通过掩码方式采样,另一个直接给出索引号采样。

现在来考虑多维掩码采样方式。为了方便演示,这里将班级数量减少到 2 个,学生的数量减少到 3 个,即一个班级只有 3 个学生,shape 为[2,3,8]。如果希望采样第 1 个班级的第 1～2 号学生,第 2 个班级的第 2～3 号学生,可以实现为

```
In[49]:
x = torch.randint(0, 9, [2,3,8])
idx = torch.tensor([[0,0], [0,1], [1,1], [1,2]])
x[idx[:,0], idx[:,1]]                  #多维坐标采样
Out[49]:
tensor([[3, 5, 8, 1, 1, 0, 1, 2],
        [0, 7, 0, 7, 4, 7, 6, 5],
        [2, 7, 7, 5, 4, 3, 3, 3],
        [2, 8, 6, 2, 8, 5, 3, 7]])
```

共采样 4 个学生的成绩,张量 shape 为[4,8]。

如果用掩码方式,可以表示为如表 5.2 所示的成绩册掩码采样方案,表中数据表达了对应位置的采样情况。

表 5.2 成绩册掩码采样方案

班级	学生 0	学生 1	学生 2
班级 0	True	True	False
班级 1	False	True	True

因此,通过这张表就能很好地表征利用掩码方式的采样方案。下面通过 x[mask]方式来实现多维掩码方式采样,代码实现如下。

```
In[50]:                                #多维掩码采样
mask = torch.tensor([[True,True,False],[False,True,True]])
```

```
x[mask]                                    #多维掩码方式
Out[50]:
tensor([[3, 5, 8, 1, 1, 0, 1, 2],
        [0, 7, 0, 7, 4, 7, 6, 5],
        [2, 7, 7, 5, 4, 3, 3, 3],
        [2, 8, 6, 2, 8, 5, 3, 7]])
```

采样结果与直接给出多维坐标的完全一致。实际上，掩码坐标与索引坐标之间可以进行等价转换，掩码坐标转索引坐标可以通过 idx＝mask. nonzero()方式实现，而索引坐标转掩码可以通过 mask[idx[:,0],idx[:,1]]＝True 类似的方式实现。

特别应该注意的是，PyTorch 中提供的掩码函数 torch. masked_select 反而使用方式比较单一，并不如上述掩码方式灵活。它需要指定每个维度的掩码坐标，再进行采样。接下来考虑采集张量中负数的问题，需要返回某张量中所有负数的元素。

首先创建随机输入张量，并通过 lt 小于函数来获取负数掩码，代码如下：

```
x = torch.randn([2,2])                     #创建张量
print('x:', x)
mask = torch.lt(x, 0)                      #获取负数坐标掩码
print('mask:', mask)
```

然后通过 masked_select 函数来选择所有负数元素并返回，代码如下：

```
torch.masked_select(x, mask)               #返回负数元素
```

正如上文所述，推荐大家直接用[]形式进行索引采样或者掩码采样，功能强大且表达简洁。

5.6.3 gather 采样函数

在多维坐标索引采样中，需要给出所有采样点的多维坐标信息。尤其是当需要采样某个维度上的部分数据，而其他维度全部采样时，直接使用多维坐标采样方式表示非常烦琐，此时可以通过 Gather 函数实现。

考虑这样一个具体问题，现需要随机采样每个同学的两门科目成绩。为了便于演示，设成绩张量 shape 为[2,3,4]，即共有两个班级，每个班级 3 位学生，4 门科目成绩。这里将采样方案的张量表达为

```
#随机采样方案
idx = torch.tensor([
    #第一个班级采样方案
    [[0,1],                                #班级 1,学生 1,采样第 1、2 门科目
    [1,2],                                 #班级 1,学生 2,采样第 2、3 门科目
    [2,3]],                                #班级 1,学生 3,采样第 3、4 门科目
    #第二个班级采样方案
    [[3,2],                                #班级 2,学生 1,采样第 4、3 门科目
    [2,1],                                 #班级 2,学生 2,采样第 3、2 门科目
    [1,0]],                                #班级 2,学生 3,采样第 2、1 门科目
])
```

采样方案张量的 shape 为[2,3,2],除了采样维度 dim=2 外,其他维度与成绩张量的长度保持一致。上述采样方法的实现如下。

```
In [51]:
x = torch.randint(0,9,[2,3,4])                    #随机生成成绩张量
print('x:', x)
out = torch.gather(x, dim = 2, index = idx)        #在科目维度上采集数据
print('out:', out)
Out[51]:
#成绩张量
x: tensor([[[5, 0, 7, 8],
            [8, 4, 5, 8],
            [4, 2, 2, 0]],
           [[8, 1, 7, 5],
            [0, 6, 7, 3],
            [2, 3, 2, 0]]])
#采样结果
out: tensor([[[5, 0],                              #班级 1,学生 1,采样第 1、2 门科目
              [4, 5],                              #班级 1,学生 2,采样第 2、3 门科目
              [2, 0]],                             #班级 1,学生 3,采样第 3、4 门科目

             [[5, 7],                              #班级 2,学生 1,采样第 4、3 门科目
              [7, 6],                              #班级 2,学生 2,采样第 3、2 门科目
              [3, 2]]])                            #班级 2,学生 3,采样第 2、1 门科目
```

可以看到,gather 的采样方式与 index_select 等的采样方式是完全不同的,读者需要仔细体会两者的区别。

5.6.4 where 采样函数

通过 torch.where(cond,a,b)操作可以根据 cond 条件的真假从参数 $\boldsymbol{A}$ 或 $\boldsymbol{B}$ 中读取数据,条件判定规则为

$$o_i = \begin{cases} a_i, & \text{cond}_i \text{ 为 True} \\ b_i, & \text{cond}_i \text{ 为 False} \end{cases}$$

其中,i 为张量的元素索引,返回的张量大小与 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 一致,当对应位置的 cond_i 为 True 时,o_i 从 a_i 中复制数据;当对应位置的 cond_i 为 False 时,o_i 从 b_i 中复制数据。考虑从 2 个全 1 和全 0 的 3×3 大小的张量 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 中提取数据,其中 cond_i 为 True 的位置从 $\boldsymbol{A}$ 中对应位置提取元素 1,cond_i 为 False 的位置从 $\boldsymbol{B}$ 对应位置提取元素 0,代码如下。

```
In [53]:
a = torch.ones([3,3])                              #构造 a 为全 1 矩阵
b = torch.zeros([3,3])                             #构造 b 为全 0 矩阵
#构造采样条件
cond = torch.tensor([[True,False,False],[False,True,False],[True,True,False]])
torch.where(cond,a,b)                              #根据条件从 a,b 中采样
Out[53]:
```

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

可以看到，返回的张量中为 1 的位置全部来自张量 a，返回的张量中为 0 的位置全部来自张量 b。

当参数 a=b=None 时，即 a 和 b 参数不指定，torch. where 会返回 cond 张量中所有 True 的元素的索引坐标，此时 torch. where 等价于 torch. nonzero 函数。考虑如下 cond 张量，代码为

```
In [54]: cond                                    #构造的 cond 张量
Out[54]:
tensor([[ True, False, False],
        [False, True, False],
        [ True, True, False]])
```

其中 True 共出现 4 次，每个 True 元素位置处的索引分别为[0,0]、[1,1]、[2,0]、[2,1]，可以直接通过 torch. where(cond)形式来获得这些元素的索引坐标，代码如下。

```
In [55]:
#获取 cond 中为 True 的元素索引
print('where:', torch.where(cond))
print('nonzero:', cond.nonzero())
Out[55]:
where: (tensor([0, 1, 2, 2]), tensor([0, 1, 0, 1]))
nonzero: tensor([[0, 0],
                 [1, 1],
                 [2, 0],
                 [2, 1]])
```

可以看到，where 与 nonzero 函数是完全等价的，但是在结果的表达方式上略有不同，where 函数会将坐标信息拆开为元组表示。

那么这有什么用途呢？考虑一个场景，需要提取张量中所有正数的数据和索引。首先构造张量 a，并通过比较运算得到所有正数的位置掩码，代码如下。

```
In [56]:
x = torch.randn([3,3])                           #构造张量 a
Out[56]:
tensor([[ 0.6784, -0.3506, -0.1932],
        [ 0.2462,  0.3916,  0.8359],
        [ 0.8373, -0.5131,  1.0343]])
```

通过比较运算，得到所有正数的掩码，代码如下。

```
In [57]:mask = x>0                               #比较操作
mask
Out[57]:
tensor([[ True, False, False],
        [ True, True, True],
        [ True, False, True]])
```

通过 where 提取此掩码处 True 元素的索引坐标，代码如下。

```
In [58]:
idx = torch.where(mask)                          # 提取所有大于 0 的元素索引
Out[58]:
(tensor([0, 1, 1, 1, 2, 2]), tensor([0, 0, 1, 2, 0, 2]))
```

拿到索引后，通过多维坐标索引即可恢复出所有正数的元素，代码如下。

```
In [59]:
x[idx[0], idx[1]]                                # 多维坐标索引，提取正数的元素值
Out[59]:
tensor([0.6784, 0.2462, 0.3916, 0.8359, 0.8373, 1.0343])
```

实际上，当得到掩码 mask 之后，也可以直接通过多维掩码索引方式获取所有正数的元素向量，代码如下。

```
In [60]:
x[mask]                                          # 直接利用掩码进行多维索引
Out[60]:
tensor([0.6784, 0.2462, 0.3916, 0.8359, 0.8373, 1.0343])
```

结果也是一致的。

通过上述一系列的比较、索引号收集和掩码收集的操作组合，相信读者能够比较直观地感受到这个功能是有很大的实际应用价值的，深刻理解它们的本质有利于更加灵活地选用简便高效的方式实现算法逻辑。

5.6.5 scatter 写入函数

前面介绍了 gather 采样方式，它通过 idx 张量指定的采样坐标来从张量中读取数据，gather 函数的逆过程可以理解为根据 idx 张量指定的采样坐标来更新张量的部分数据，可以通过 scatter 函数或 scatter_函数来实现。在 PyTorch 中，函数名加下画线后缀一般表示原地更新(In-place Update)操作，类似的函数还有 fill_、add_等。

通过 x.scatter (dim,index,src)函数可以高效地写入张量的部分数据，特别适合需要根据坐标来更新张量的部分数据的场合。其中 dim 表示更新的维度，index 参数意义等价于 gather 函数的 index 参数，用于选定需要更新数据的坐标，而待写入的数据则用 src 张量表示。通常把 x 张量称为目标张量，src 张量称为源张量。

如图 5.3 所示，演示了一维张量的 scatter 操作过程。目标张量保存为张量 x，需要刷新的数据索引号通过 index 表示，新数据保存在源张量 src 中。根据 index 给出的索引位置将 src 中新的数据依次写入 x 中，并返回更新后的结果张量。

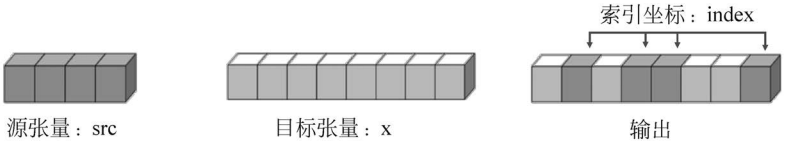

图 5.3 scatter 更新数据示意图

实现一个图 5.3 中向量的刷新实例，代码如下。

```
In [61]:
#构造需要刷新数据的位置参数，即为 4、3、1 和 7 号位置
idx = torch.tensor([4,3,1,7])
#构造需要写入的数据，4 号位写入 4.4，3 号位写入 3.3，以此类推
src = torch.tensor([4.4, 3.3, 1.1, 7.7])
#创建目标张量，为便于演示，创建为全 0 张量
x = torch.zeros([8])
#在长度为 8 的全 0 向量上根据索引位置(indices)写入刷新(updates)数据
x.scatter(dim = 0, index = idx, src = src)
Out[61]:
tensor([0.0000, 1.1000, 0.0000, 3.3000, 4.4000, 0.0000, 0.0000, 7.7000])
```

可以看到，在长度为 8 的目标张量 x 上，写入了对应位置的数据，4 个位置的数据被刷新。scatter 的索引方式可以视为 gather 的逆过程。

考虑三维张量的刷新例子，如图 5.4 所示，目标张量 x 的 shape 为[4,4,4]，同理数据设置为全 0，共有 4 个通道的特征图，每个通道大小为 4×4，现有 2 个通道的源张量 src，其 shape 为[2,4,4]，需要写入索引为[1,3]的通道上。

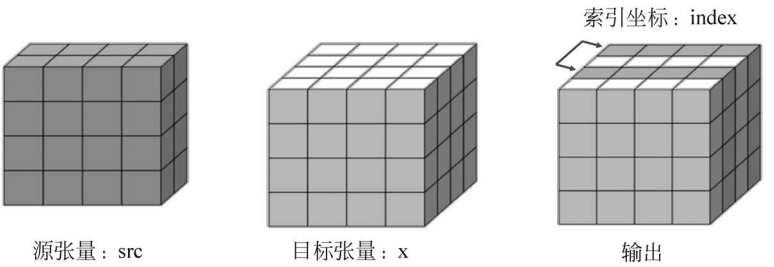

图 5.4　三维张量更新示意图

下面将新的特征图写入目标张量中，实现如下。

```
In [62]:                                          #构造写入位置，即 2 个位置
idx = torch.tensor([1, 3])
src = torch.tensor([                              #构造写入数据，即 2 个矩阵
    [[5,5,5,5],[6,6,6,6],[7,7,7,7],[8,8,8,8]],
    [[1,1,1,1],[2,2,2,2],[3,3,3,3],[4,4,4,4]]
]).float()
x = torch.zeros([4,4,4])
x[idx] = src                                      #这种单一维度的索引写入不需要 scatter 函数
x
Out[62]:
tensor([[[0, 0, 0, 0],
         [0, 0, 0, 0],
         [0, 0, 0, 0],
         [0, 0, 0, 0]],
        [[5, 5, 5, 5],                            #写入的新数据 1
         [6, 6, 6, 6],
         [7, 7, 7, 7],
```

```
        [8, 8, 8, 8]],
       [[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]],
       [[1, 1, 1, 1],                          #写入的新数据 2
        [2, 2, 2, 2],
        [3, 3, 3, 3],
        [4, 4, 4, 4]]])>
```

可以看到，数据被刷新到第 2 和第 4 个通道特征图上。上述实现非常简单，并不需要 scatter 这种复杂的索引方式。

下面将继续使用 5.6.3 节中的例子来更新部分科目成绩数据。考虑 shape 为[2,3,4]的成绩张量，即共有两个班级，每个班级 3 位学生，4 门科目成绩。继续使用坐标为 idx 的采样方案，并生成目标张量 x，代码如下。

```
#随机采样方案
idx = torch.tensor([
    #第一个班级采样方案
    [[0,1],                                  #班级 1,学生 1,采样第 1、2 门科目
    [1,2],                                   #班级 1,学生 2,采样第 2、3 门科目
    [2,3]],                                  #班级 1,学生 3,采样第 3、4 门科目
    #第二个班级采样方案
    [[3,2],                                  #班级 2,学生 1,采样第 4、3 门科目
    [2,1],                                   #班级 2,学生 2,采样第 3、2 门科目
    [1,0]],                                  #班级 2,学生 3,采样第 2、1 门科目
])
x = torch.randint(0,9,[2,3,4]).float()       #随机生成成绩张量
print('x:', x)
```

其中 x 成绩张量内容为

```
tensor([[[2., 5., 8., 0.],
         [4., 2., 2., 2.],
         [0., 7., 0., 7.]],
        [[4., 2., 8., 3.],
         [1., 5., 6., 2.],
         [5., 1., 8., 0.]]])
```

现需要从源张量 src 中更新目标张量的部分数据，这里将采样的科目成绩全部设置为 10，即

```
src = torch.full([2,3,2], 10)                #这里简单设置新更新的数据全为 10
```

通过 scatter 进行更新，代码如下。

```
x.scatter(dim = 2, index = idx, src = src)
```

返回张量如下。

```
tensor([[[10., 10.,  8.,  0.],
         [ 4., 10., 10.,  2.],
         [ 0.,  7., 10., 10.]],
```

```
[[ 4.,  2., 10., 10.],
 [ 1., 10., 10.,  2.],
 [10., 10.,  8.,  0.]]])
```

可以看到，idx 坐标所选定的科目成绩更新为 10，结果符合预期。这个例子比较好地展示了 scatter 函数的强大之处。

5.6.6 meshgrid 网格函数

算法中对特征位置进行编码(Positional Encoding)时，或者可视化 3D 函数时，通常需要生成一组网格点的坐标张量。在 PyTorch 中，可以通过 torch.meshgrid 函数方便地生成二维网格的采样点坐标。考虑 2 个自变量 x 和 y 的 sinc 函数表达式为

$$z=\frac{\sin(x^2+y^2)}{x^2+y^2}$$

如果需要绘制在 $x\in[-8,8]$，$y\in[-8,8]$区间的 sinc 函数的 3D 曲面，如图 5.5 所示，则首先需要生成 x 轴和 y 轴的网格点坐标集合$\{(x,y)\}$，这样才能通过 sinc 函数的表达式计算函数在每个(x,y)位置的输出值 z。

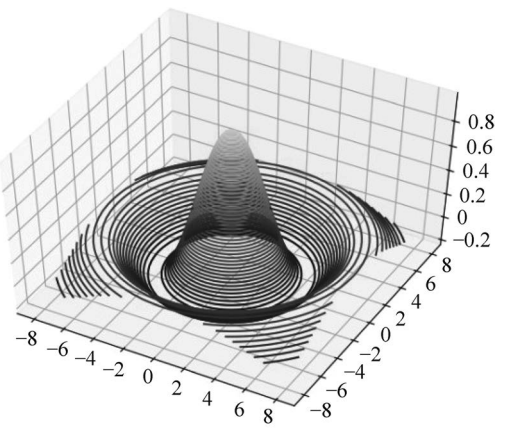

图 5.5 sinc 函数

可以通过如下方式生成 1 万个坐标采样点：

```
points = []                                  # 保存所有点的坐标列表
for x in range(-8,8,100):                    # 循环生成 x 坐标，100 个采样点
    for y in range(-8,8,100):                # 循环生成 y 坐标，100 个采样点
    z = sinc(x,y)                            # 计算每个点(x,y)处的 sinc 函数值
        points.append([x,y,z])               # 保存采样点
```

很明显这种串行采样方式效率极低，那么有没有简洁高效的方式生成网格坐标呢？答案是肯定的，meshgrid 函数即可实现。

通过在 x 轴上进行采样 100 个数据点，y 轴上采样 100 个数据点，然后利用 torch.meshgrid(x,y)即可返回这 1 万个数据点的张量数据，保存在 shape 为[100,100,2]的张量

中。为了方便计算，torch.meshgrid 会返回在 dim=2 维度切割后的 2 个张量 $\boldsymbol{A}$ 和 $\boldsymbol{B}$，其中张量 $\boldsymbol{A}$ 包含了所有点的 x 坐标，$\boldsymbol{B}$ 包含了所有点的 y 坐标，shape 都为[100,100]，实现如下。

```
In [63]:
x = torch.linspace(-8.,8,100)                    #设置 x 轴的采样点
y = torch.linspace(-8.,8,100)                    #设置 y 轴的采样点
x,y = torch.meshgrid(x,y)                        #生成网格点,并内部拆分后返回
x.shape,y.shape                                  #打印拆分后的所有点的 x,y 坐标张量 shape
Out[63]:
torch.Size([100, 100]), torch.Size([100, 100])
```

利用生成的网格点坐标张量 $\boldsymbol{A}$ 和 $\boldsymbol{B}$，sinc 函数在 PyTorch 中实现如下。

```
z = torch.sqrt(x**2+y**2)
z = torch.sin(z)/z                               #sinc 函数实现
```

通过 matplotlib 库即可绘制出函数在 $x\in[-8,8]$，$y\in[-8,8]$ 区间的 3D 曲面，如图 5.5 所示。代码如下。

```
import matplotlib
from matplotlib import pyplot as plt
#导入 3D 坐标轴支持
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = Axes3D(fig)                                 #设置 3D 坐标轴
#根据网格点绘制 sinc 函数 3D 曲面
ax.contour3D(x.numpy(), y.numpy(), z.numpy(), 50)
plt.show()
```

5.7 经典数据集加载

到这里为止，张量的常用操作方法已经介绍完，用户能够实现大部分深度网络的技术储备。最后，将以一个完整的张量方式实现的分类网络模型实战收尾本章。在进入实战之前，先正式介绍对于常用的经典数据集，如何利用 PyTorch 生态链提供的工具便捷地加载数据集。对于自定义的数据集的加载，将会在后续章节介绍。

PyTorch 发布至今，受到越来越多的开发者的青睐，目前在很多国际会议论文中的使用率已经远超 TensorFlow。使用的人越多，生态系统相应地越加完善，目前涌现出如 torchvision、torchtext、torchaudio、transformers、deepspeed、accelerate 等一系列优秀的面向各个行业应用的第三方库，极大地方便了开发人员的使用。在图片和视频相关应用中，torchvision 是使用最多的库之一，它提供了经典数据集的加载以及常见网络模型、图片的增强变换、可视化等快捷功能，本书视觉相关任务均采用 torchvision 库完成。

torchvision 库提供了常用的经典数据集的自动下载、管理、加载与转换功能，配合 PyTorch 的 DataLoader 类，可以方便地实现多线程(Multi-threading)、数据变换(Transformation)、

随机打散(Shuffle)和批训练(Training on Batch)等常用数据处理逻辑。

对于常用的经典图片数据集,介绍如下。

① MNIST、Fashion MNIST 等:手写数字图片数据集;

② CIFAR10/100:小规模图片数据集;

③ ImageNet:大规模图片数据集;

④ VOC:图片分割数据集;

⑤ Kinetics-400:视频动作理解数据集。

这些数据集在机器学习或深度学习的研究和学习中使用地非常频繁。对于新提出的算法,通常优先在经典的数据集上面进行测试和验证,再尝试迁移到更大规模、更复杂的数据集上。torchvision 均对这些常见数据集的加载提供了便捷支持,对于如 MNIST、CIFAR 这种小型数据集,可以直接在线下载、自动加载;对于如 ImageNet、Kinetics-400 这种大型数据集,用户需要自行下载数据集文件,并在 torchvision 中指定路径。

通过 torchvision.datasets.xxx 函数即可实现经典数据集的自动加载,其中 xxx 代表具体的数据集名称,如"CIFAR10""MNIST"等。torchvision 会默认将数据缓存在用户指定的文件夹中,如图 5.6 所示,用户不需要关心数据集是如何保存的。如果当前数据集不在缓存中,则会自动从网络下载、解压和加载数据集;如果已经在缓存中,则自动完成加载。例如,自动加载 MNIST 数据集,代码如下。

```
In [66]:
import  torch
import  torch.nn as nn
import  torch.nn.functional as F
import  torch.optim as optim
#常用数据集加载类,数据变换类
from torchvision import datasets, transforms

batch_size = 200
learning_rate = 0.01
epochs = 10

#导入 MNIST 数据集
train_db = datasets.MNIST('./data',
                          train = True,          #训练集
                          download = True,       #没有则下载
                      #数据变换
                      transform = transforms.Compose([
                        transforms.ToTensor(),
                        transforms.Normalize((0.1307,), (0.3081,))
                    ])
                    )
print('数据集大小:', len(train_db))
x,y = train_db[0]
print("样本:", x.shape, y)
Out [66]:#返回数组的形状
```

```
数据集大小: 60000
样本:torch.Size([1, 28, 28]) 5
```

通过对 train_db 对象进行迭代会返回相应格式的数据，对于图片数据集 MNIST、CIFAR10 等，会返回 2 个 tuple，tuple 保存了用于训练的数据 x 和 y 训练集对象；对测试 test_db 进行迭代，tuple 则保存了用于测试的数据 x_test 和 y_test 测试集对象，所有的数据都用 Numpy 数组容器保存。

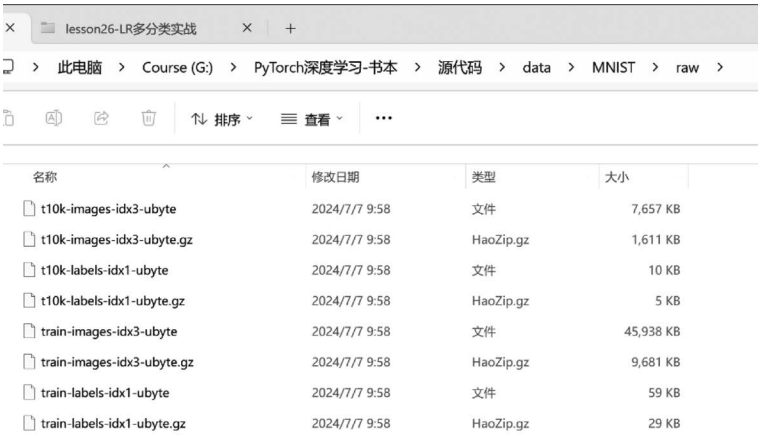

图 5.6 PyTorch 缓存经典数据集的位置

数据集类只有单个样本的读取能力，但是没有批量读取等便捷功能。一般需要再添加一系列的数据集标准处理步骤，如预处理、随机打散、批训练等。

5.7.1 预处理

从 torchvision.datasets 中加载的数据集的格式一般是初始的样本数据，大部分情况都不能直接满足模型的输入要求，因此需要根据模型的逻辑自行实现预处理步骤。PyTorch 提供了 transform 函数作为输入参数，可以接受并调用用户自定义的预处理逻辑，并且常用的预处理逻辑已经在 torchvision 中内置支持了，可以直接调用 torchvision.transforms 来构建常用的预处理逻辑组合。例如，下方代码显示的是调用 ToTensor 和 Normalize 进行预处理的过程。

```
#数据变换
transform = transforms.Compose([
    #将 Numpy 数据转换为 PyTorch 张量
    transforms.ToTensor(),
    #对张量进行归一化
    transforms.Normalize((0.1307,), (0.3081,))
])
```

考虑 MNIST 手写数字图片，从 datasets 中经.batch()后加载的图片 x 的 shape 为[1,28,28]，像素使用区间[−1,1]表示；标签 shape 为[1]，即采样数字编码方式。实际的

神经网络输入，一般需要将图片数据标准化到[0,1]或[−1,1]等 0 附近区间，同时根据网络的设置，需要将 shape 为[28,28]的输入视图调整为合法的格式；对于标签信息，可以选择在预处理时进行 One-hot 编码，也可以在计算误差时进行 One-hot 编码。

Transforms 也支持自定义的预处理函数：transforms.Lambda，将 MNIST 图片中的 ToTensor 和 Normalize 删除，全部自行实现在 myimgproc 函数中。myimgproc 函数实现如下。

```
def myimgproc(img):
    img = np.array(img)
    #转换为 float Tensor
    img = torch.tensor(img).float()
    #裁剪到区间[-1,1]
    img = torch.clip(img, -1.0, 1.0)
    #转换为想要的格式
    img = img.reshape([1, 28, 28])

    #自行实现的归一化逻辑
    img = (img - 0.1307) / 0.3081

    return img

#自定义数据变换
transform = transforms.Compose([
    #将 Numpy 数据转换为 PyTorch 张量
    #transforms.ToTensor(),
    #自行实现的预处理逻辑
    transforms.Lambda(myimgproc),
    #对张量进行归一化,在这里自行实现
    #transforms.Normalize((0.1307,), (0.3081,)),
])
```

5.7.2 随机打散

通过 DataLoader 类工具可以设置 Dataset 对象随机打散数据之间的顺序，防止每次训练时数据按固定顺序产生，从而使得模型尝试“记忆”住标签信息，代码实现如下。

```
#随机打散样本,不会打乱样本与标签的映射关系
#构建 DataLoader 对象
train_loader = torch.utils.data.DataLoader(
    train_db,
    batch_size = batch_size,                    #设置批大小
    shuffle = True,                             #设置随机打散
    )
#同样的方法构建 test loader
test_loader = torch.utils.data.DataLoader(
    datasets.MNIST('./data', train = False, transform = transforms.Compose([
        transforms.ToTensor(),
```

```
        transforms.Normalize((0.1307,), (0.3081,))
    ])),
    batch_size = batch_size, shuffle = True)
```

5.7.3 批训练

为了利用显卡的并行算力，一般在网络的计算过程中会同时计算多个样本，把这种训练方式叫作批训练，通过对 DataLoader 类进行迭代，即可实现批训练，实现如下。

```
#迭代 DataLoader
batchx, batchy = next(iter(train_loader))
print('批加载:', batchx.shape)
print('批加载:', batchy)
批加载:torch.Size([32, 1, 28, 28])
批加载:tensor([2, 0, 2, 1, 7, 6, 7, 8, 2, 8, 3, 8, 7, 4, 1, 9, 5, 0, 4, 0, 3, 1, 0, 6, 6, 5, 6, 0,
               6, 1, 8, 1])
```

其中 32 为 Batch Size 参数，即一次并行计算 32 个样本的数据。Batch Size 一般根据用户的 GPU 显存资源来设置，当显存不足时，可以适量减少 Batch Size 来减少算法的显存使用量。

5.7.4 循环训练

对于 Dataset 对象，在使用时可以通过

```
for step, (x,y) in enumerate(train_db):          #迭代数据集对象,带 step 参数
```

或

```
for x,y in train_db:                             #迭代数据集对象
```

方式进行迭代，每次返回的 x 和 y 对象即为批量样本和标签。当对 train_db 的所有样本完成一次迭代后，for 循环终止退出。这样完成一个 batch 的数据训练，叫作一个 step；通过多个 step 来完成整个训练集的一次迭代，叫作一个 epoch。在实际训练时，通常需要对数据集迭代多个 epoch 才能取得较好地训练效果。例如，固定训练 20 个 epoch，实现如下：

```
for epoch in range(20):                          #训练 epoch 数
for step, (x,y) in enumerate(train_db):          #迭代 step 数
#training...
```

5.8 MNIST 测试实战

前面已经介绍并实现了前向传播和数据集的加载部分，现在来完成剩下的分类任务逻辑。在训练的过程中，通过间隔数个 step 后打印误差数据，可以有效监督模型的训练进度，代码如下。

```
##间隔 100 个 step 打印一次训练误差
```

```
if batch_idx % 100 == 0:
    print('Train epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
        epoch, batch_idx * len(data), len(train_loader.dataset),
               100. * batch_idx / len(train_loader), loss.item()))
```

由于 loss 为张量类型，因此可以通过 item()函数将标量转换为标准的 Python 浮点数。在若干 step 或者若干 epoch 训练后，可以进行一次测试(验证)，以获得模型的当前性能。

现在利用学习到的 PyTorch 张量操作函数，完成模型的计算实战。首先考虑一个 batch 的样本 x，通过前向计算可以获得网络的预测值，代码如下。

```
def forward(x):
    #前向计算函数

    #第一层+激活函数
    x = x@w1.t() + b1
    x = F.relu(x)
    #第二层+激活函数
    x = x@w2.t() + b2
    x = F.relu(x)
    #第三层+激活函数
    x = x@w3.t() + b3
    #x = F.relu(x)
    return x
```

预测值 out 的 shape 为$[b,10]$，分别代表样本属于每个类别的概率，根据 argmax 或者 max 函数选出概率最大值出现的索引号，也即样本最有可能的类别号，代码如下。

```
for data, target in test_loader:
    data = data.view(-1, 28 * 28)
    logits = forward(data)

    _, pred = logits.data.max(1)                    #选取概率最大的类别
```

通过 eq 函数可以比较这两者的结果是否相等，代码如下。

```
correct += pred.eq(target.data).sum()               #比较预测值与真实值
```

并求和比较结果中所有 True(转换为 1)的数量，即为预测正确的数量。预测正确的数量除以总测试数量即可得到准确度，并打印出来，实现如下。

```
print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
    test_loss, correct, len(test_loader.dataset),
    100. * correct / len(test_loader.dataset)))
```

通过简单的 3 层神经网络，训练固定的 20 个 epoch 后，在测试集上获得了 87.25%的准确率。如果使用大规模预训练技术，即更复杂的神经网络模型，增加数据增强环节，细调网络超参数等技巧，可以获得更高的模型性能。模型的训练误差曲线如图 5.7 所示，测试准确率曲线如图 5.8 所示。

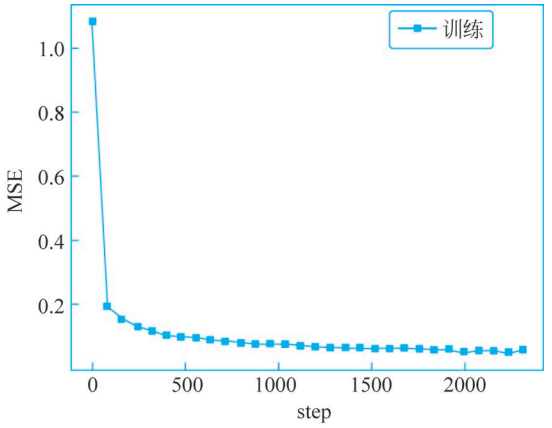

图 5.7 MNIST 训练误差曲线

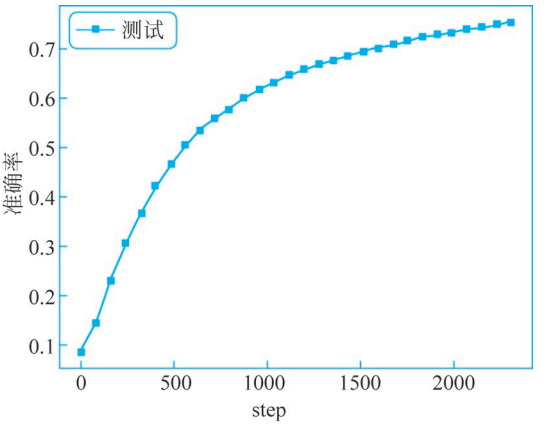

图 5.8 MNIST 测试准确率曲线