第 5 章

CHAPTER 5

TensorFlow 进阶

人工智能将是谷歌的最终版本。它将成为终极搜索引擎，可以理解网络上的一切信息。它会准确地理解你想要什么，给你需要的东西。——拉里·佩奇

在介绍完张量的基本操作后，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、数据限幅等，并通过 MNIST 数据集的测试实战，来加深用户对 TensorFlow 张量操作的理解。

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 为[4,35,8]；同样的方式，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 2 份成绩册，便可得到学校所有班级的成绩册，记为张量 $\boldsymbol{C}$，shape 应为[10,35,8]，其中，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。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。

(1) **拼接**：在 TensorFlow 中，可以通过 tf.concat(tensors,axis)函数拼接张量，其中参数 tensors 保存了所有需要合并的张量 List，axis 参数指定需要合并的维度索引。回到上面的例子，在班级维度上合并成绩册，这里班级维度索引号为 0，即 axis=0，合并张量 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 的代码如下：

```
In [1]:
```

```
a = tf.random.normal([4,35,8])                    # 模拟成绩册 A
b = tf.random.normal([6,35,8])                    # 模拟成绩册 B
tf.concat([a,b],axis = 0)                         # 拼接合并成绩册
Out[1]:
< tf.Tensor: id = 13,shape = (10,35,8),dtype = float32,numpy =
array([[[ 1.95299834e - 01,6.87859178e - 01, - 5.80048323e - 01,...,
          1.29430830e + 00,2.56610274e - 01, - 1.27798581e + 00],
        [ 4.29753691e - 01,9.11329567e - 01, - 4.47975427e - 01,...,
```

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 = tf.random.normal([10,35,4])
b = tf.random.normal([10,35,4])
tf.concat([a,b],axis = 2)                         # 在科目维度上拼接
Out[2]:
< tf.Tensor: id = 28,shape = (10,35,8),dtype = float32,numpy =
array([[[ - 5.13509691e - 01, - 1.79707789e + 00,6.50747120e - 01,...,
          2.58447856e - 01,8.47878829e - 02,4.13468748e - 01],
        [ - 1.17108583e + 00,1.93961406e + 00,1.27830813e - 02,...,
```

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

```
In [3]:
a = tf.random.normal([4,32,8])
b = tf.random.normal([6,35,8])
tf.concat([a,b],axis = 0)                         # 非法拼接,其他维度长度不相同
Out[3]:
InvalidArgumentError: ConcatOp : Dimensions of inputs should match: shape[0] = [4,32,8] vs.
shape[1] = [6,35,8] [Op:ConcatV2] name: concat
```

(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 tf.stack 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 2 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]。

使用 tf.stack(tensors, axis)可以堆叠方式合并多个张量，通过 tensors 列表表示，参数

axis 指定新维度插入的位置，axis 的用法与 tf.expand_dims 的一致，当 axis≥0 时，在 axis 之前插入新维度；当 axis<0 时，在 axis 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，axis 参数对应的插入位置设置如图 5.1 所示。

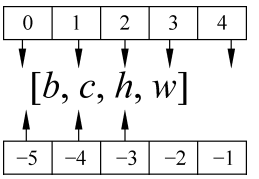

图 5.1 stack 插入维度位置

堆叠方式合并这 2 个班级成绩册，班级维度插入在 axis=0 位置，代码如下：

```
In [4]:
a = tf.random.normal([35,8])
b = tf.random.normal([35,8])
tf.stack([a,b],axis = 0)                    # 堆叠合并为 2 个班级，班级维度插入在最前
Out[4]:
<tf.Tensor: id = 55,shape = (2,35,8),dtype = float32,numpy =
array([[[ 3.68728966e - 01, - 8.54765773e - 01, - 4.77824420e - 01,
          - 3.83714020e - 01, - 1.73216307e + 00,2.03872994e - 02,
           2.63810277e + 00, - 1.12998331e + 00],...
```

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：

```
In [5]:
a = tf.random.normal([35,8])
b = tf.random.normal([35,8])
tf.stack([a,b],axis = - 1)                  # 在末尾插入班级维度
Out[5]:
<tf.Tensor: id = 69,shape = (35,8,2),dtype = float32,numpy =
array([[[ 0.3456724 , - 1.7037214 ],
        [ 0.41140947, - 1.1554345 ],
        [ 1.8998919 ,0.56994915],...
```

此时班级的维度在 axis=2 轴上面，理解时也需要按最新的维度顺序代表的视图去理解数据。若选择使用 tf.concat 拼接合并上述成绩单，则可以合并为：

```
In [6]:
a = tf.random.normal([35,8])
b = tf.random.normal([35,8])
tf.concat([a,b],axis = 0)                   # 拼接方式合并，没有 2 个班级的概念
Out[6]:
<tf.Tensor: id = 108,shape = (70,8),dtype = float32,numpy =
array([[ - 0.5516891 , - 1.5031327 , - 0.35369992,0.31304857,0.13965549,
          0.6696881 , - 0.50115544,0.15550546],
       [ 0.8622069 ,1.0188094 ,0.18977325,0.6353301 ,0.05809061,...
```

可以看到，tf.concat 也可以顺利合并数据，但是在理解时，需要按前 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 tf.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。

tf.stack 也需要满足张量堆叠合并条件，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，例如：

```
In [7]:
a = tf.random.normal([35,4])
b = tf.random.normal([35,8])
tf.stack([a,b],axis = -1)                    # 非法堆叠操作，张量 shape 不相同
Out[7]:
InvalidArgumentError: Shapes of all inputs must match: values[0].shape = [35,4] != values[1].
shape = [35,8] [Op:Pack] name: stack
```

上述操作尝试合并 shape 为 [35,4] 和 [35,8] 的 2 个张量，由于两者形状不一致，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，将一个张量分拆为多个张量。继续考虑成绩册的例子，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切割为 10 个张量，每个张量保存了对应班级的成绩册数据。

通过 tf.split(x, num_or_size_splits, axis)可以完成张量的分割操作，参数意义如下：

- x 参数：待分割张量。
- num_or_size_splits 参数：切割方案。当 num_or_size_splits 为单个数值时，如 10，表示等长切割为 10 份；当 num_or_size_splits 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。
- axis 参数：指定分割的维度索引号。

现在将总成绩册张量切割为 10 份，代码如下：

```
In [8]:
x = tf.random.normal([10,35,8])
# 等长切割为 10 份
result = tf.split(x,num_or_size_splits = 10,axis = 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]，例如：

```
In [9]: result[0]                            # 查看第一个班级的成绩册张量
Out[9]: <tf.Tensor: id = 136,shape = (1,35,8),dtype = float32,numpy =
array([[[ -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。

来进行不等长的切割，例如，将数据切割为 4 份，每份长度分别为[4,2,2,2]，实现如下：

```
In [10]: x = tf.random.normal([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 = tf.split(x,num_or_size_splits = [4,2,2,2] ,axis = 0)
len(result)
Out[10]: 4
```

查看第一个张量的 shape，根据切割方案，它应该包含了 4 个班级的成绩册，shape 应为 [4,35,8]，验证一下：

```
In [10]: result[0]
Out[10]: <tf.Tensor: id = 155,shape = (4,35,8),dtype = float32,numpy =
array([[[ - 6.95693314e - 01,3.01393479e - 01,1.33964568e - 01,...,
```

特别地，如果希望在某个维度上全部按长度为 1 的方式分割，还可以使用 tf.unstack(x, axis)函数。这种方式是 tf.split 的一种特殊情况，切割长度固定为 1，只需要指定切割维度的索引号即可。例如，将总成绩册张量在班级维度进行 unstack 操作：

```
In [11]: x = tf.random.normal([10,35,8])
result = tf.unstack(x,axis = 0)                    # unstack 是长度为 1 的张量
len(result)                                        # 返回 10 个张量的列表
Out[11]: 10
```

查看切割后的张量的形状：

```
In [12]: result[0]                                 # 第一个班级
Out[12]: <tf.Tensor: id = 166,shape = (35,8),dtype = float32,numpy =
array([[ - 0.2034383 ,1.1851563 ,0.25327438, - 0.10160723,2.094969 ,
         - 0.8571669 , - 0.48985648,0.55798006],...
```

可以看到，通过 tf.unstack 切割后，shape 变为 [35,8]，即班级维度消失了，这也是与 tf.split 区别之处。

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、最值位置、均值、范数等信息。由于张量通常较大，直接观察数据很难获得有用信息，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。

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。常用的向量范数如下：

- L1 范数，定义为向量 $\boldsymbol{x}$ 的所有元素绝对值之和：

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

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

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

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

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

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

在 TensorFlow 中，可以通过 tf.norm(x, ord)求解张量的 L1、L2、∞等范数，其中参数 ord 指定为 1、2 时计算 L1、L2 范数，指定为 np.inf 时计算∞范数，例如：

```
In [13]: x = tf.ones([2,2])
tf.norm(x,ord = 1)                              # 计算 L1 范数
Out[13]: <tf.Tensor: id = 183,shape = (),dtype = float32,numpy = 4.0>
In [14]: tf.norm(x,ord = 2)                     # 计算 L2 范数
Out[14]: <tf.Tensor: id = 189,shape = (),dtype = float32,numpy = 2.0>
In [15]: import numpy as np
tf.norm(x,ord = np.inf)                         # 计算∞范数
Out[15]: <tf.Tensor: id = 194,shape = (),dtype = float32,numpy = 1.0>
```

5.2.2 最值、均值、和

通过 tf.reduce_max、tf.reduce_min、tf.reduce_mean、tf.reduce_sum 函数可以求解张量在某个维度上的最大、最小、均值、和，也可以求全局最大、最小、均值和信息。

考虑 shape 为[4,10]的张量，其中，第一个维度代表样本数量，第二个维度代表当前样本分别属于 10 个类别的概率，需要求出每个样本的概率最大值为，可以通过 tf.reduce_max 函数实现：

```
In [16]: x = tf.random.normal([4,10])          # 模型生成概率
tf.reduce_max(x,axis = 1)                       # 统计概率维度上的最大值
Out[16]:<tf.Tensor: id = 203,shape = (4,),dtype = float32,numpy
 = array([1.2410722 ,0.88495886,1.4170984 ,0.9550192 ],dtype = float32)>
```

返回长度为 4 的向量，分别代表每个样本的最大概率值。同样求出每个样本概率的最小值，实现如下：

```
In [17]: tf.reduce_min(x,axis = 1)              # 统计概率维度上的最小值
Out[17]:<tf.Tensor: id = 206,shape = (4,),dtype = float32,numpy
 = array([-0.27862206,-2.4480672 ,-1.9983795 ,-1.5287997 ],dtype = float32)>
```

求出每个样本概率的均值，实现如下：

```
In [18]: tf.reduce_mean(x,axis = 1)            # 统计概率维度上的均值
Out[18]:< tf.Tensor: id = 209,shape = (4,),dtype = float32,numpy
 = array([ 0.39526337, - 0.17684573, - 0.148988 , - 0.43544054],dtype = float32)>
```

当不指定 axis 参数时，tf.reduce_ * 函数会求解出全局元素的最大、最小、均值、和等数据，例如：

```
In [19]:x = tf.random.normal([4,10])
# 统计全局的最大、最小、均值、和，返回的张量均为标量
tf.reduce_max(x),tf.reduce_min(x),tf.reduce_mean(x)
Out [19]: (< tf.Tensor: id = 218,shape = (),dtype = float32,numpy = 1.8653786 >,
< tf.Tensor: id = 220,shape = (),dtype = float32,numpy = - 1.9751656 >,
< tf.Tensor: id = 222,shape = (),dtype = float32,numpy = 0.014772797 >)
```

在求解误差函数时，通过 TensorFlow 的 MSE 误差函数可以求得每个样本的误差，需要计算样本的平均误差，此时可以通过 tf.reduce_mean 在样本数维度上计算均值，实现如下：

```
In [20]:
out = tf.random.normal([4,10])                 # 模拟网络预测输出
y = tf.constant([1,2,2,0])                     # 模拟真实标签
y = tf.one_hot(y,depth = 10)                   # One - hot 编码
loss = keras.losses.mse(y,out)                 # 计算每个样本的误差
loss = tf.reduce_mean(loss)                    # 平均误差，在样本数维度上取均值
loss # 误差标量
Out[20]:
< tf.Tensor: id = 241,shape = (),dtype = float32,numpy = 1.1921183 >
```

与均值函数相似的是求和函数 tf.reduce_sum(x, axis)，它可以求解张量在 axis 轴上所有特征的和：

```
In [21]:out = tf.random.normal([4,10])
tf.reduce_sum(out,axis = - 1)                  # 求最后一个维度的和
Out[21]:< tf.Tensor: id = 303,shape = (4,),dtype = float32,numpy = array([ - 0.588144 , 2.
2382064,2.1582587,4.962141 ],dtype = float32)>
```

除了希望获取张量的最值信息，还希望获得最值所在的位置索引号，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，一般把这个位置索引号作为预测类别。考虑 10 分类问题，得到神经网络的输出张量 out，shape 为 [2,10]，代表了 2 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，例如：

```
In [22]:out = tf.random.normal([2,10])
out = tf.nn.softmax(out,axis = 1)              # 通过 softmax 函数转换为概率值
```

```
out
Out[22]:< tf.Tensor: id = 257,shape = (2,10),dtype = float32,numpy =
array([[0.18773547,0.1510464 ,0.09431915,0.13652141,0.06579739,
        0.02033597,0.06067333,0.0666793 ,0.14594753,0.07094406],
       [0.5092072 ,0.03887136,0.0390687 ,0.01911005,0.03850609,
        0.03442522,0.08060656,0.10171875,0.08244187,0.05604421]],
      dtype = float32)>
```

以第一个样本为例，可以看到，它概率最大的索引为 $i=0$，最大概率值为 0.1877。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，在预测时考虑第一个样本应该最有可能属于类别 0。这就是需要求解最大值的索引号的一个典型应用。

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

```
In [23]:pred = tf.argmax(out,axis = 1)        # 选取概率最大的位置
pred
Out[23]:< tf.Tensor: id = 262,shape = (2,),dtype = int64,numpy = array([0,0],dtype = int64)>
```

可以看到，这 2 个样本概率最大值都出现在索引 0 上，因此最有可能都是类别 0，可以将类别 0 作为这 2 个样本的预测类别。

5.3 张量比较

为了计算分类任务的准确率等指标，一般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 tf.argmax 获取预测类别，实现如下：

```
In [24]:out = tf.random.normal([100,10])
out = tf.nn.softmax(out,axis = 1)             # 输出转换为概率
pred = tf.argmax(out,axis = 1)                # 计算预测值
Out[24]:< tf.Tensor: id = 272,shape = (100,),dtype = int64,numpy =
array([0,6,4,3,6,8,6,3,7,9,5,7,3,7,1,5,6,1,2,9,0,6,
       5,4,9,5,6,4,6,0,8,4,7,3,4,7,4,1,2,4,9,4,...
```

变量 pred 保存了这 100 个样本的预测类别值，与这 100 个样本的真实标签比较，例如：

```
In [25]:                                      # 模型生成真实标签
y = tf.random.uniform([100],dtype = tf.int64,maxval = 10)
Out[25]:< tf.Tensor: id = 281,shape = (100,),dtype = int64,numpy =
array([0,9,8,4,9,7,2,7,6,7,3,4,2,6,5,0,9,4,5,8,4,2,
       5,5,5,3,8,5,2,0,3,6,0,7,1,1,7,0,6,1,2,1,3,...
```

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 tf.equal(a, b)(或 tf.math.equal(a, b),两者等价)函数可以比较这 2 个张量是否相等,例如:

```
In [26]:out = tf.equal(pred,y)                    # 预测值与真实值比较,返回布尔类型的张量
Out[26]:<tf.Tensor: id=288,shape=(100,),dtype=bool,numpy=
array([False,False,False,False,True,False,False,False,False,
       False,False,False,False,False,True,False,False,True,...
```

tf.equal()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 1 的个数,就可以得到比较结果中 True 元素的个数:

```
In [27]:out = tf.cast(out,dtype=tf.float32)       # 布尔型转 int 型
correct = tf.reduce_sum(out)                      # 统计 True 的个数
Out[27]:<tf.Tensor: id=293,shape=(),dtype=float32,numpy=12.0>
```

可以看到,随机产生的预测数据中预测正确的个数是 12,因此它的准确度是

$$\text{accuracy}=\frac{12}{100}=12\%$$

这也是随机预测模型的正常水平。

除了比较相等的 tf.equal(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	函 数	比较逻辑
tf.math.greater	$a>b$	tf.math.less_equal	$a\leqslant b$
tf.math.less	$a<b$	tf.math.not_equal	$a\neq b$
tf.math.greater_equal	$a\geqslant b$	tf.math.is_nan	$a=\text{nan}$

5.4 填充与复制

5.4.1 填充

对于图片数据的高和宽、序列信号的长度,维度长度可能各不相同。为了方便网络的并行计算,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,并不适合于此处。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 0,使得填充后的长度满足系统要求。那么这种操作就称为填充(Padding)。

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 I,2 代表 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]

此时这两个句子可堆叠合并shape为[2,6]的张量。

填充操作可以通过tf.pad(x, paddings)函数实现，参数paddings是包含了多个[LeftPadding,RightPadding]的嵌套方案List，如[[0,0],[2,1],[1,2]]表示第一个维度不填充，第二个维度左边(起始处)填充两个单元，右边(结束处)填充一个单元，第三个维度左边填充一个单元，右边填充两个单元。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paddings方案为[[0,2]]：

```
In [28]:a = tf.constant([1,2,3,4,5,6])                    # 第一个句子
b = tf.constant([7,8,1,6])                                # 第二个句子
b = tf.pad(b,[[0,2]])                                     # 句子末尾填充2个0
b                                                         # 填充后的结果
Out[28]:<tf.Tensor: id=3,shape=(6,),dtype=int32,numpy=array([7,8,1,6,0,0])>
```

填充后句子张量形状一致，再将这两个句子stack在一起，代码如下：

```
In [29]:tf.stack([a,b],axis=0)                            # 堆叠合并,创建句子数维度
Out[29]:<tf.Tensor: id=5,shape=(2,6),dtype=int32,numpy=
array([[1,2,3,4,5,6],
       [7,8,1,6,0,0]])>
```

在自然语言处理中，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，部分句子长度较长，如超过100个单词。为了能够保存在同一张量中，一般会选取能够覆盖大部分句子长度的阈值，如80个单词。对小于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，截断超过规定长度的部分单词。以IMDB数据集的加载为例，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：

```
[  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。

下面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。

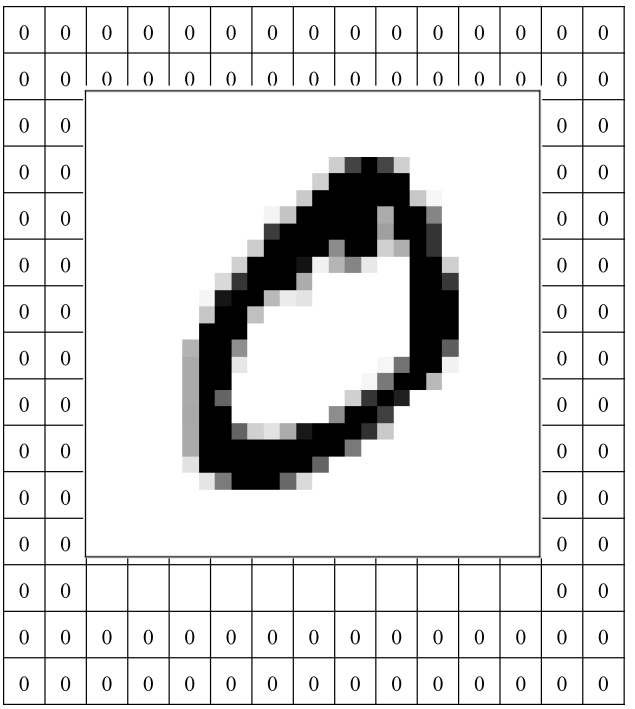

图 5.2 图片填充

上述填充方案可以表达为[[0,0],[2,2],[2,2],[0,0]],实现如下：

```
In [31]:
x = tf.random.normal([4,28,28,1])
# 图片上下、左右各填充 2 个单元
tf.pad(x,[[0,0],[2,2],[2,2],[0,0]])
Out[31]:
<tf.Tensor: id = 16, shape = (4,32,32,1), dtype = float32, numpy =
array([[[[ 0.         ],
         [ 0.         ],
         [ 0.         ],...
```

通过填充操作后，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 节，就介绍了通过 tf.tile()函数实现长度为 1 的维度复制的功能。tf.tile()函数除了可以对长度为 1 的维度进行复制若干份，还可以对任意长度的维度进行复制若干份，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，此处仅做简单回顾。

通过 tf.tile()函数可以在任意维度将数据重复复制多份，如 shape 为 [4,32,32,3] 的数据，复制方案为 multiples=[2,3,3,1]，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 = tf.random.normal([4,32,32,3])
tf.tile(x,[2,3,3,1])                              # 数据复制
Out[32]:<tf.Tensor: id = 25, shape = (8,96,96,3), dtype = float32, numpy =
array([[[[ 1.20957184e+00,2.82766962e+00,1.65782201e+00],
         [ 3.85402292e-01,2.00732923e+00,-2.79068202e-01],
         [-2.52583921e-01,7.82584965e-01,7.56870627e-01],...
```

5.5 数据限幅

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，限制元素的范围为 $x\in[0,+\infty)$即可。

在 TensorFlow 中，可以通过 tf.maximum(x, a)实现数据的下限幅，即 $x\in[a,+\infty)$；可以通过 tf.minimum(x, a)实现数据的上限幅，即 $x\in(-\infty,a]$，举例如下：

```
In [33]:x = tf.range(9)
tf.maximum(x,2)                                   # 下限幅到 2
Out[33]:<tf.Tensor: id = 48, shape = (9,), dtype = int32, numpy = array([2,2,2,3,4,5,6,7,8])>
In [34]:tf.minimum(x,7)                           # 上限幅到 7
Out[34]:<tf.Tensor: id = 41, shape = (9,), dtype = int32, numpy = array([0,1,2,3,4,5,6,7,7])>
```

基于 tf.maximum 函数，可以实现 ReLU 函数如下：

```
def relu(x):                                    # ReLU 函数
return tf.maximum(x,0.)                         # 下限幅为 0 即可
```

通过组合 tf.maximum(x, a)和 tf.minimum(x, b)可以实现同时对数据的上下边界限幅，即 $x \in [a,b]$，例如：

```
In [35]:x = tf.range(9)
tf.minimum(tf.maximum(x,2),7)                   # 限幅为 2～7
Out[35]:<tf.Tensor: id=57,shape=(9,),dtype=int32,numpy=array([2,2,2,3,4,5,6,7,7])>
```

更方便地，可以使用 tf.clip_by_value 函数实现上下限幅：

```
In [36]:x = tf.range(9)
tf.clip_by_value(x,2,7)                         # 限幅为 2～7
Out[36]:<tf.Tensor: id=66,shape=(9,),dtype=int32,numpy=array([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都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 tf.gather

tf.gather 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，保存成绩册的张量 shape 为 [4,35,8]。

```
x = tf.random.uniform([4,35,8],maxval=100,dtype=tf.int32)      # 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 axis=0，通过 tf.gather 函数收集数据，代码如下：

```
In [38]:tf.gather(x,[0,1],axis=0)               # 在班级维度收集第 1～2 号班级成绩册
Out[38]:<tf.Tensor: id=83,shape=(2,35,8),dtype=int32,numpy=
array([[[43,10,93,85,75,87,28,19],
        [52,17,44,88,82,54,16,65],
        [98,26,1,47,59,3,59,70],...
```

实际上，对于上述需求，通过切片 $x[:2]$可以更加方便地实现。但是对于不规则的索引方式，例如，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 tf.gather 则是针对于此需求设计的，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
tf.gather(x,[0,3,8,11,12,26],axis=1)
```

```
Out[39]:< tf.Tensor: id = 87,shape = (4,6,8),dtype = int32,numpy =
array([[[43,10,93,85,75,87,28,19],
        [74,11,25,64,84,89,79,85],...
```

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 axis＝2，实现如下：

```
In [40]:tf.gather(x,[2,4],axis = 2)              # 第 3,5 科目的成绩
Out[40]:< tf.Tensor: id = 91,shape = (4,35,2),dtype = int32,numpy =
array([[[93,75],
        [44,82],
        [ 1,59],...
```

可以看到，tf.gather 非常适合索引号没有规则的场合，其中索引号可以乱序排列，此时收集的数据也是对应顺序，例如：

```
In [41]:a = tf.range(8)
a = tf.reshape(a,[4,2])                          # 生成张量 a
Out[41]:< tf.Tensor: id = 115,shape = (4,2),dtype = int32,numpy =
array([[0,1],
       [2,3],
       [4,5],
       [6,7]])>
In [42]:tf.gather(a,[3,1,0,2],axis = 0)          # 收集第 4,2,1,3 号元素
Out[42]:< tf.Tensor: id = 119,shape = (4,2),dtype = int32,numpy =
array([[6,7],
       [2,3],
       [0,1],
       [4,5]])>
```

将问题变得稍微复杂一点。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 tf.gather 实现。首先抽出第 [2,3] 班级，实现如下：

```
In [43]:
students = tf.gather(x,[1,2],axis = 0)           # 收集第 2,3 号班级
Out[43]:< tf.Tensor: id = 227,shape = (2,35,8),dtype = int32,numpy =
array([[[ 0,62,99,7,66,56,95,98],...
```

再从这 2 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
tf.gather(students,[2,3,5,26],axis = 1)          # 收集第 3,4,6,27 号同学
Out[44]:< tf.Tensor: id = 231,shape = (2,4,8),dtype = int32,numpy =
array([[[69,67,93,2,31,5,66,65],...
```

此时得到这 2 个班级 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。

可以通过笨方式，一个个地手动提取数据。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：

```
In [45]: x[1,1]                                    # 收集第 2 个班级的第 2 个同学
Out[45]:< tf.Tensor: id = 236,shape = (8,),dtype = int32,numpy = array([45,34,99,17,3,
1,43,86])>
```

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]: tf.stack([x[1,1],x[2,2],x[3,3]],axis = 0)
Out[46]:< tf.Tensor: id = 250,shape = (3,8),dtype = int32,numpy =
array([[45,34,99,17,3,1,43,86],
       [11,25,84,95,97,95,69,69],
       [ 0,89,52,29,76,7,2,98]])>
```

这种方法也能正确地得到 shape 为 [3,8] 的结果，其中 3 表示采样点的个数，8 表示每个采样点的特征数据的长度。但是它最大的问题在于用手动串行方式地执行采样，计算效率极低。可以采用 tf.gather_nd 来实现。

5.6.2 tf.gather_nd

通过 tf.gather_nd 函数，可以指定每次采样点的多维坐标来实现采样多个点的目的。回到上面的挑战，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。那么这 3 个采样点的索引坐标可以记为 [1,1]、[2,2]、[3,3]，将这个采样方案合并为一个 List 参数，即[[1,1],[2,2],[3,3]]，通过 tf.gather_nd 函数即可实现如下：

```
In [47]:                                           # 根据多维坐标收集数据
tf.gather_nd(x,[[1,1],[2,2],[3,3]])
Out[47]:< tf.Tensor: id = 256,shape = (3,8),dtype = int32,numpy =
array([[45,34,99,17,3,1,43,86],
       [11,25,84,95,97,95,69,69],
       [ 0,89,52,29,76,7,2,98]])>
```

可以看到，结果与串行采样方式完全一致，实现更加简洁，计算效率大大提升。

一般地，在使用 tf.gather_nd 采样多个样本时，例如希望采样 i 号班级，j 个学生，k 门科目的成绩，则可以表达为[…,[i,j,k],…]，外层的括号长度为采样样本的个数，内层列表包含了每个采样点的索引坐标，例如：

```
In [48]:                                           # 根据多维度坐标收集数据
tf.gather_nd(x,[[1,1,2],[2,2,3],[3,3,4]])
Out[48]:< tf.Tensor: id = 259,shape = (3,),dtype = int32,numpy = array([99,95,76])>
```

上述代码中，抽出了班级 1 的学生 1 的科目 2、班级 2 的学生 2 的科目 3、班级 3 的学生 3 的科目 4 的成绩，共有 3 个成绩数据，结果汇总为一个 shape 为 [3] 的张量。

5.6.3 tf.boolean_mask

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 和第 4 个班级的数据，通过 tf.boolean_mask(x, mask, axis)可以在 axis 轴上根据 mask 方案进行采样，实现为：

```
In [49]:                                          # 根据掩码方式采样班级,给出掩码和维度索引
tf.boolean_mask(x,mask = [True,False,False,True],axis = 0)
Out[49]:< tf.Tensor: id = 288,shape = (2,35,8),dtype = int32,numpy =
array([[[43,10,93,85,75,87,28,19],...
```

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 [50]:                                          # 根据掩码方式采样科目
tf.boolean_mask(x,mask = [True,False,False,True,True,False,False,True],axis = 2)
Out[50]:< tf.Tensor: id = 318,shape = (4,35,4),dtype = int32,numpy =
array([[[43,85,75,19],...
```

不难发现，这里的 tf.boolean_mask 的用法其实与 tf.gather 非常类似，只不过一个通过掩码方式采样，一个直接给出索引号采样。

现在来考虑与 tf.gather_nd 类似的多维掩码采样方式。为了方便演示，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，通过 tf.gather_nd 可以实现为：

```
In [51]:x = tf.random.uniform([2,3,8],maxval = 100,dtype = tf.int32)
tf.gather_nd(x,[[0,0],[0,1],[1,1],[1,2]])     # 多维坐标采集
Out[51]:< tf.Tensor: id = 325,shape = (4,8),dtype = int32,numpy =
array([[52,81,78,21,50,6,68,19],
       [53,70,62,12,7,68,36,84],
       [62,30,52,60,10,93,33,6],
       [97,92,59,87,86,49,47,11]])>
```

共采样 4 个学生的成绩，shape 为 [4,8]。

如果用掩码方式，可以表达为如表 5.2 所示，行为每个班级，列为每个学生，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

	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，代码实现如下：

```
In [52]:                                          # 多维掩码采样
tf.boolean_mask(x,[[True,True,False],[False,True,True]])
Out[52]:<tf.Tensor: id=354,shape=(4,8),dtype=int32,numpy=
array([[52,81,78,21,50,6,68,19],
       [53,70,62,12,7,68,36,84],
       [62,30,52,60,10,93,33,6],
       [97,92,59,87,86,49,47,11]])>
```

采样结果与 tf.gather_nd 完全一致。可见 tf.boolean_mask 既可以实现 tf.gather 方式的一维掩码采样，又可以实现 tf.gather_nd 方式的多维掩码采样。

上面的 3 个操作比较常用，尤其是 tf.gather 和 tf.gather_nd 出现的频率较高，必须掌握。下面再补充 3 个高阶操作。

5.6.4 tf.where

通过 tf.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，条件判定规则如下：

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

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，o_i 从 a_i 中复制数据；当对应位置的 cond_i 为 False，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 $\boldsymbol{B}$ 中对应位置提取元素 0，代码如下：

```
In [53]:
a = tf.ones([3,3])                                # 构造 a 为全 1 矩阵
b = tf.zeros([3,3])                               # 构造 b 为全 0 矩阵
# 构造采样条件
cond = tf.constant([[True,False,False],[False,True,False],[True,True,False]])
tf.where(cond,a,b)                                # 根据条件从 a,b 中采样
Out[53]:<tf.Tensor: id=384,shape=(3,3),dtype=float32,numpy=
```

```
array([[1.,0.,0.],
       [0.,1.,0.],
       [1.,1.,0.]],dtype=float32)>
```

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,tf.where会返回cond张量中所有True的元素的索引坐标。考虑如下cond张量:

```
In[54]: cond                                    # 构造的cond张量
Out[54]:<tf.Tensor: id=383,shape=(3,3),dtype=bool,numpy=
array([[ 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tf.where(cond)形式来获得这些元素的索引坐标,代码如下:

```
In[55]:tf.where(cond)                           # 获取cond中为True的元素索引
Out[55]:<tf.Tensor: id=387,shape=(4,2),dtype=int64,numpy=
array([[0,0],
       [1,1],
       [2,0],
       [2,1]],dtype=int64)>
```

那么这有什么用途?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:

```
In[56]:x = tf.random.normal([3,3])              # 构造a
Out[56]:<tf.Tensor: id=403,shape=(3,3),dtype=float32,numpy=
array([[-2.2946844 ,0.6708417 ,-0.5222212 ],
       [-0.6919401 ,-1.9418817 ,0.3559235 ],
       [-0.8005251 ,1.0603906 ,-0.68819374]],dtype=float32)>
```

通过比较运算,得到所有正数的掩码:

```
In[57]:mask=x>0                                 # 比较操作,等同于tf.math.greater()
mask
Out[57]:<tf.Tensor: id=405,shape=(3,3),dtype=bool,numpy=
array([[False,True,False],
       [False,False,True],
       [False,True,False]])>
```

通过tf.where提取此掩码处True元素的索引坐标:

```
In[58]:indices=tf.where(mask)                   # 提取所有大于0的元素索引
Out[58]:<tf.Tensor: id=407,shape=(3,2),dtype=int64,numpy=
array([[0,1],
```

```
        [1,2],
        [2,1]],dtype=int64)>
```

拿到索引后，通过 tf.gather_nd 即可恢复出所有正数的元素：

```
In [59]:tf.gather_nd(x,indices)                    # 提取正数的元素值
Out[59]:<tf.Tensor: id=410,shape=(3,),dtype=float32,numpy
 =array([0.6708417,0.3559235,1.0603906],dtype=float32)>
```

实际上，当得到掩码 mask 之后，也可以直接通过 tf.boolean_mask 获取所有正数的元素向量：

```
In [60]:tf.boolean_mask(x,mask)                    # 通过掩码提取正数的元素值
Out[60]:<tf.Tensor: id=439,shape=(3,),dtype=float32,numpy
 =array([0.6708417,0.3559235,1.0603906],dtype=float32)>
```

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，能够比较直观地感受到这个功能是有很大实际应用的，并且深刻地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我们的目的。

5.6.5 scatter_nd

通过 tf.scatter_nd(indices, updates, shape)函数可以高效地刷新张量的部分数据，但是这个函数只能在全 0 的白板张量上面执行刷新操作，因此可能需要结合其他操作来实现现有张量的数据刷新功能。

如图 5.3 所示，演示了一维张量白板的刷新运算原理。白板的形状通过 shape 参数表示，需要刷新的数据索引号通过 indices 表示，新数据为 updates。根据 indices 给出的索引位置将 updates 中新的数据依次写入白板中，并返回更新后的结果张量。

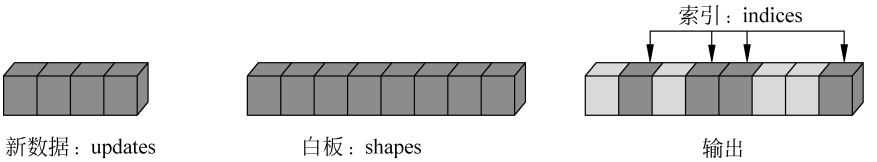

图 5.3 scatter_nd 更新数据

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
indices = tf.constant([[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
updates = tf.constant([4.4,3.3,1.1,7.7])
# 在长度为 8 的全 0 向量上根据 indices 写入 updates 数据
tf.scatter_nd(indices,updates,[8])
```

```
Out[61]:<tf.Tensor: id=467,shape=(8,),dtype=float32,numpy
 =array([0.,1.1,0.,3.3,4.4,0.,0.,7.7],dtype=float32)>
```

可以看到,在长度为 8 的白板上,写入了对应位置的数据,4 个位置的数据被刷新。

考虑三维张量的刷新例子,如图 5.4 所示,白板张量的 shape 为[4,4,4],共有 4 个通道的特征图,每个通道大小为 4×4,现有 2 个通道的新数据 updates 为[2,4,4],需要写入索引为 [1,3] 的通道上。

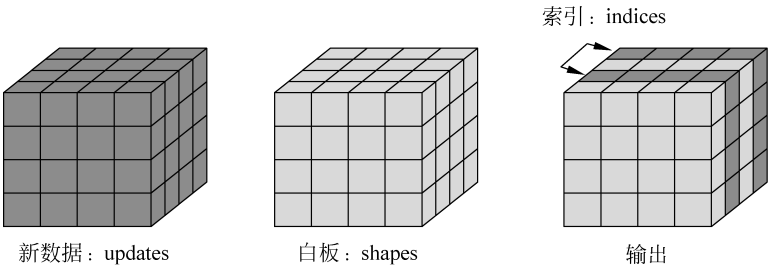

图 5.4　三维张量更新

将新的特征图写入现有白板张量,实现如下：

```
In[62]:                                          # 构造写入位置,即 2 个位置
indices = tf.constant([[1],[3]])
updates = tf.constant([                          # 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]]
])
# 在 shape 为[4,4,4]白板上根据 indices 写入 updates
tf.scatter_nd(indices,updates,[4,4,4])
Out[62]:<tf.Tensor: id=477,shape=(4,4,4),dtype=int32,numpy=
array([[[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。

5.6.6 meshgrid

通过 tf.meshgrid 函数可以方便地生成二维网格的采样点坐标，方便可视化等应用场合。考虑 2 个自变量 x 和 y 的 sinc 函数表达式为：

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

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 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。可以通过如下方式生成 10000 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，那么有没有通过 tf.meshgrid 函数即可以简洁、高效的方式生成网格坐标。

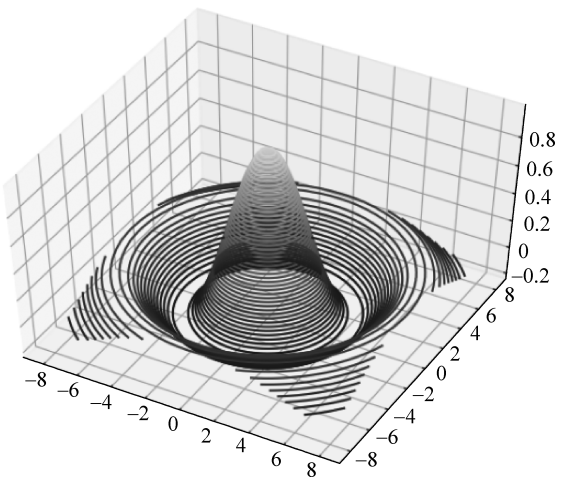

图 5.5 sinc 函数

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 tf.meshgrid(x, y)即可返回这 10000 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，tf.meshgrid 会返回在 axis=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 = tf.linspace( - 8.,8,100)                    # 设置 x 轴的采样点
y = tf.linspace( - 8.,8,100)                    # 设置 y 轴的采样点
x,y = tf.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]: (TensorShape([100,100]),TensorShape([100,100]))
```

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

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

通过 matplotlib 库即可绘制出函数在 $x \in [-8,8]$，$y \in [-8,8]$ 区间的三维(3D)曲面，如图 5.5 所示。代码如下：

```
import matplotlib
from matplotlib import pyplot as plt
# 导入三维坐标轴支持
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = Axes3D(fig)                                # 设置三维坐标轴
# 根据网格点绘制 sinc 函数三维曲面
ax.contour3D(x.numpy(),y.numpy(),z.numpy(),50)
plt.show()
```

5.7 经典数据集加载

到目前为止，已经学完张量的常用操作方法，已具备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 TensorFlow 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，会在后续章节中介绍。

在 TensorFlow 中，keras.datasets 模块提供了常用经典数据集的自动下载、管理、加载与转换功能，并且提供了 tf.data.Dataset 数据集对象，方便实现多线程(Multi-threading)、预处理(Preprocessing)、随机打散(Shuffle)和批训练(Training on Batch)等常用数据集的功能。

对于常用的经典数据集，介绍如下：

- Boston Housing：波士顿房价趋势数据集，用于回归模型训练与测试。
- CIFAR10/100：真实图片数据集，用于图片分类任务。
- MNIST/Fashion_MNIST：手写数字图片数据集，用于图片分类任务。
- IMDB：情感分类任务数据集，用于文本分类任务。

这些数据集在机器学习或深度学习的研究和学习中使用非常频繁。对于新提出的算法，一般优先在经典的数据集上面测试，再尝试迁移到更大规模、更复杂的数据集上。

通过 datasets.xxx.load_data()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。TensorFlow 会默认将数据缓存在用户目录下的.keras/datasets 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：

```
In [66]:
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import datasets                    # 导入经典数据集加载模块
# 加载 MNIST 数据集
(x,y),(x_test,y_test) = datasets.mnist.load_data()
print('x:',x.shape,'y:',y.shape,'x test:',x_test.shape,'y test:',y_test)
Out [66]:                                                # 返回数组的形状
x: (60000,28,28) y: (60000,) x test: (10000,28,28) y test: [7 2 1 ...4 5 6]
```

通过 load_data()函数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，第 1 个 tuple 保存了用于训练的数据 x 和 y 训练集对象；第 2 个 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。

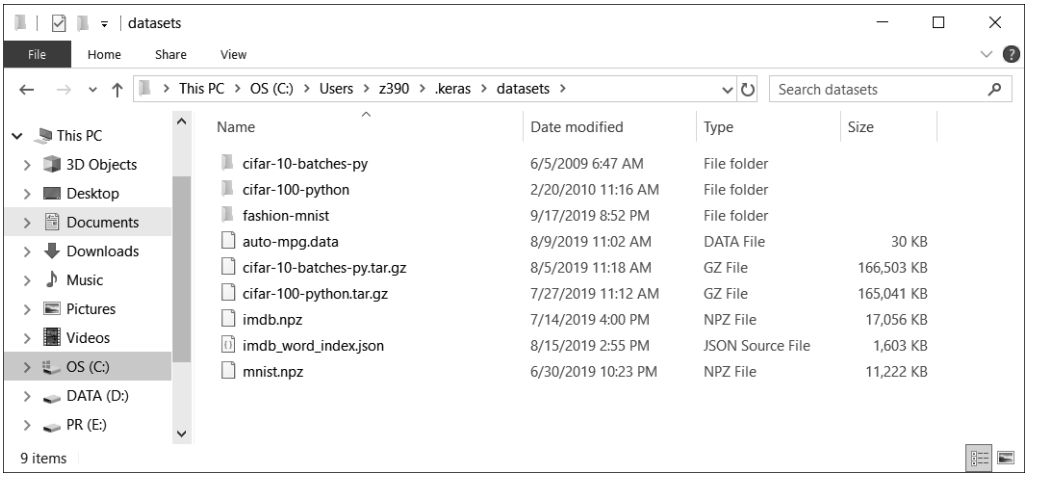

图 5.6 TensorFlow 缓存经典数据集的位置

数据加载进入内存后，需要转换成 Dataset 对象，才能利用 TensorFlow 提供的各种便捷功能。通过 Dataset.from_tensor_slices 可以将训练部分的数据图片 x 和标签 y 都转换成 Dataset 对象：

```
train_db = tf.data.Dataset.from_tensor_slices((x,y))      # 构建 Dataset 对象
```

将数据转换成 Dataset 对象后，一般需要再添加一系列的数据集标准处理步骤，如随机打散、预处理、批训练等。

5.7.1 随机打散

通过 Dataset.shuffle(buffer_size)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：

```
train_db = train_db.shuffle(10000)        # 随机打散样本，不会打乱样本与标签映射关系
```

其中，buffer_size 参数指定缓冲池的大小，一般设置为一个较大的常数即可。调用 Dataset 提供的这些工具函数会返回新的 Dataset 对象，可以通过

$$db = db.step1().step2().step3.()$$

方式按序完成所有的数据处理步骤，实现起来非常方便。

5.7.2 批训练

为了利用显卡的并行计算能力，一般在网络的计算过程中会同时计算多个样本，把这种训练方式称为批训练，其中一个批中样本的数量称为 Batch Size。为了一次能够从 Dataset 中产生 Batch Size 数量的样本，需要设置 Dataset 为批训练方式，实现如下：

```
train_db = train_db.batch(128)            # 设置批训练，batch size 为 128
```

其中 128 为 Batch Size 参数，即一次并行计算 128 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。

5.7.3 预处理

从 keras.datasets 中加载的数据集的格式大部分情况都不能直接满足模型的输入要求，因此需要根据用户的逻辑自行实现预处理步骤。Dataset 对象通过提供 map(func)工具函数，可以非常方便地调用用户自定义的预处理逻辑，它实现在 func 函数中。例如，下面代码调用名为 preprocess 的函数完成每个样本的预处理。

```
# 预处理函数实现在 preprocess 函数中，传入函数名即可
train_db = train_db.map(preprocess)
```

考虑 MNIST 手写数字图片，从 keras.datasets 中经 batch()后加载的图片 x shape 为 $[b,28,28]$，像素使用 0～255 的整型表示；标签 shape 为 $[b]$，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。

根据 5.8 节的实战设定，将 MNIST 图片数据映射到 $x \in [0,1]$ 区间，视图调整为 $[b, 28\times28]$；对于标签数据，选择在预处理函数中进行 One-hot 编码。preprocess 函数实现如下：

```
def preprocess(x,y):                                    # 自定义的预处理函数
    # 调用此函数时会自动传入 x,y 对象,shape 为[b,28×28],[b]
    # 标准化到 0～1
    x = tf.cast(x,dtype=tf.float32) / 255.
    x = tf.reshape(x,[-1,28*28])                        # 打平
    y = tf.cast(y,dtype=tf.int32)                       # 转成整型张量
    y = tf.one_hot(y,depth=10)                          # One-hot 编码
    # 返回的 x,y 将替换传入的 x,y 参数,从而实现数据的预处理功能
    return x,y
```

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，称为一个 step；通过多个 step 来完成整个训练集的一次迭代，称为一个 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...
```

此外，也可以通过设置 Dataset 对象，使得数据集对象内部遍历多次才会退出，实现如下：

```
train_db = train_db.repeat(20)                          # 数据集迭代 20 遍才终止
```

上述代码使得 for x,y in train_db 循环迭代 20 个 Epoch 才会退出。不管使用上述哪种方式，都能取得一样的效果。由于第 4 章已经完成了前向计算实战，此处略过。

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 step % 100 == 0:
    print(step,'loss:',float(loss))
```

由于 loss 为 TensorFlow 的张量类型，因此可以通过 float()函数将标量转换为标准的 Python 浮点数。在若干个 Step 或者若干个 Epoch 训练后，可以进行一次测试(验证)，以获得模型的当前性能，例如：

```
if step % 500 == 0:                          # 每 500 个 batch 后进行一次测试(验证)
    # evaluate/test
```

现在利用学习到的 TensorFlow 张量操作函数，完成准确度的计算实战。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：

```
for x,y in test_db:                          # 对测验集迭代一遍
    h1 = x @ w1 + b1                         # 第一层
    h1 = tf.nn.relu(h1)                      # 激活函数
    h2 = h1 @ w2 + b2                        # 第二层
    h2 = tf.nn.relu(h2)                      # 激活函数
    out = h2 @ w3 + b3                       # 输出层
```

预测值 out 的 shape 为$[b,10]$，分别代表了样本属于每个类别的概率，根据 tf.argmax 函数选出概率最大值出现的索引号，即样本最有可能的类别号：

```
pred = tf.argmax(out,axis = 1)               # 选取概率最大的类别
```

由于标注 y 已经在预处理中完成了 One-hot 编码，这在测试时其实是不需要的，因此通过 tf.argmax 可以得到数字编码的标注 y：

```
y = tf.argmax(y,axis = 1)                    # One - hot 编码逆过程
```

通过 tf.equal 可以比较这两者的结果是否相等：

```
correct = tf.equal(pred,y)                   # 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：

```
total_correct += tf.reduce_sum(tf.cast(correct,dtype
= tf.int32)).numpy()                         # 统计预测正确的样本个数
```

预测正确的数量除以总测试数量即可得到准确率，并打印出来，实现如下：

```
# 计算准确率
print(step,'Evaluate Acc:',total_correct/total)
```

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。如果使用复杂的神经网络模型，增加数据增强环节，精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。

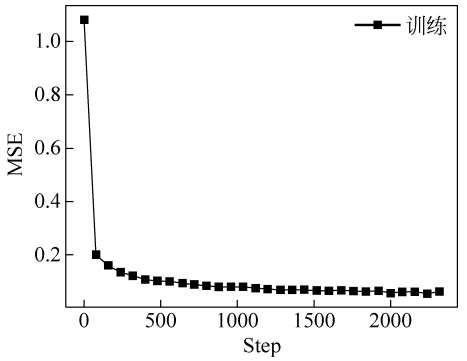

图 5.7 MNIST 训练误差曲线

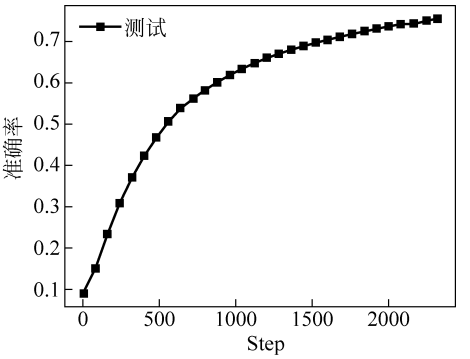

图 5.8 MNIST 测试准确率曲线