1 TensorFlow2 基础

1.1 实验介绍

1.1.1 关于本实验

本实验主要是 TensorFlow 2 的张量操作，通过对张量的一系列操作介绍，可以使学员对 TensorFlow 2 的基本语法有所了解。，包括张量的创建、切片、索引、张量维度变化、张量的算术运算、张量排序中介绍 TensorFlow 2 的语法。

1.1.2 实验目的

掌握张量的创建方法。

掌握张量的切片与索引方法。

掌握张量维度变化的语法。

掌握张量的算术运算操作。

掌握张量的排序方法。

通过代码，深入了解 Eager Execution 和 AutoGraph。

1.2 实验步骤

1.2.1 tensor 介绍

TensorFlow 中，tensor 通常分为：常量 tensor 与变量 tensor：

- 常量 tensor 定义后值和维度不可变，变量定义后值可变而维度不可变。
- 在神经网络中，变量 tensor 一般可作为储存权重和其他信息的矩阵，是可训练的数据类型。而常量 tensor 可作为储存超参数或其他结构信息的变量

1.2.1.1 创建 tensor

1.2.1.1.1 创建常量 tensor

常量 tensor 的创建方式比较多，常见的有一下几种方式：

- tf.constant()：创建常量 tensor；
- tf.zeros(), tf.zeros_like(), tf.ones(),tf.ones_like()：创建全零或者全一的常量 tensor；
- tf.fill()：创建自定义数值的 tensor；
- tf.random：创建已知分布的 tensor；
- 从 numpy，list 对象创建，再利用 tf.convert_to_tensor 转换为类型。

步骤 1 tf.constant()

tf.constant(value, dtype=None, shape=None, name='Const', verify_shape=False)：

- value：值；
- dtype：数据类型；
- shape：张量形状；
- name：常量名称；
- verify_shape：布尔值，用于验证值的形状，默认 False。verify_shape 为 True 的话表示检查 value 的形状与 shape 是否相符，如果不符会报错。

代码：

```
const_a = tf.constant([[1, 2, 3, 4]],shape=[2,2], dtype=tf.float32) #  创建 2x2 矩阵,值 1,2,3,4
const_a
```

输出：

```
<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[1., 2.],
       [3., 4.]], dtype=float32)>
```

代码：

```
#查看常见属性
print("常量 const_a 的数值为：", const_a.numpy())
print("常量 const_a 的数据类型为：", const_a.dtype)
print("常量 const_a 的形状为：", const_a.shape)
print("常量 const_a 将被产生的设备名称为：", const_a.device)
```

输出：

```
常量 const_a 的数值为： [[1. 2.]
 [3. 4.]]
常量 const_a 的数据类型为： <dtype: 'float32'>
常量 const_a 的形状为： (2, 2)
常量 const_a 将被产生的设备名称为： /job:localhost/replica:0/task:0/device:CPU:0
```

步骤 2 tf.zeros(), tf.zeros_like(), tf.ones(),tf.ones_like()

因为 tf.ones()，tf.ones_like()与 tf.zeros()，tf.zeros_like()的用法相似，因此下面只演示前者的使用方法。

创建一个值为 0 的常量。

tf.zeros(shape, dtype=tf.float32, name=None):

- shape：张量形状;
- dtype：类型;
- name：名称。

代码：

```
zeros_b = tf.zeros(shape=[2, 3], dtype=tf.int32) # 创建 2x3 矩阵，元素值均为 0
```

根据输入张量创建一个值为 0 的张量，形状和输入张量相同。

tf.zeros_like(input_tensor, dtype=None, name=None, optimize=True):

- input_tensor：张量;
- dtype：类型;
- name：名称;
- optimize：优化。

代码：

```
zeros_like_c = tf.zeros_like(const_a)
#查看生成数据
zeros_like_c.numpy()
```

输出：

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

步骤 3 tf.fill()

创建一个张量，用一个具体值充满张量。

tf.fill(dims, value, name=None)：

- dims：张量形状，同上述 shape；
- vlaue：张量数值；
- name：名称。

代码：

```
fill_d = tf.fill([3,3], 8) # 2x3 矩阵，元素值均为为 8
#查看数据
fill_d.numpy()
```

输出

```
array([[8, 8, 8],
       [8, 8, 8],
       [8, 8, 8]], dtype=int32)
```

步骤 4 tf.random

用于产生具体分布的张量。该模块中常用的方法包括：tf.random.uniform()，tf.random.normal()和 tf.random.shuffle()等。下面演示 tf.random.normal()的用法。

创建一个符合正态分布的张量。

tf.random.normal(shape, mean=0.0, stddev=1.0, dtype=tf.float32,seed=None, name=None):

- shape：数据形状；
- mean：高斯分布均值；
- stddev：高斯分布标准差；
- dtype：数据类型；
- seed：随机种子
- name：名称。

代码：

```
random_e = tf.random.normal([5,5],mean=0,stddev=1.0, seed = 1)
#查看创建数据
random_e.numpy()
```

输出：

```
array([[-0.8521641 ,  2.0672443 , -0.94127315,  1.7840577 ,  2.9919195 ],
       [-0.8644102 ,  0.41812655, -0.85865736,  1.0617154 ,  1.0575105 ],
       [ 0.22457163, -0.02204755,  0.5084496 , -0.09113179, -1.3036906 ],
       [-1.1108295 , -0.24195422,  2.8516252 , -0.7503834 ,  0.1267275 ],
       [ 0.9460202 ,  0.12648873, -2.6540542 ,  0.0853276 ,  0.01731399]],
      dtype=float32)
```

步骤 5 从 numpy，list 对象创建，再利用 tf.convert_to_tensor 转换为类型。

将给定制转换为张量。可利用这个函数将 python 的数据类型转换成 TensorFlow 可用的 tensor 数据类型。

tf.convert_to_tensor(value,dtype=None,dtype_hint=None,name=None):

- value：需转换数值；
- dtype：张量数据类型；
- dtype_hint：返回张量的可选元素类型，当 dtype 为 None 时使用。在某些情况下，调用者在 tf.convert_to_tensor 时可能没有考虑到 dtype，因此 dtype_hint 可以用作为首选项。

代码：

```
#创建一个列表
list_f = [1,2,3,4,5,6]
#查看数据类型
type(list_f)
```

输出：

```
list
```

代码：

```
tensor_f = tf.convert_to_tensor(list_f, dtype=tf.float32)
tensor_f
```

输出：

```
<tf.Tensor: shape=(6,), dtype=float32, numpy=array([1., 2., 3., 4., 5., 6.], dtype=float32)>
```

1.2.1.1.2 创建变量 tensor

TensorFlow 中，变量通过 tf.Variable 类进行操作。tf.Variable 表示张量，其值可以通过在其上运行算术运算更改。可读取和修改变量值。

代码：

```
# 创建变量，只需提供初始值
var_1 = tf.Variable(tf.ones([2,3]))
var_1
```

输出：

```
<tf.Variable 'Variable:0' shape=(2, 3) dtype=float32, numpy=
array([[1., 1., 1.],
       [1., 1., 1.]], dtype=float32)>
```

代码：

```
#变量数值读取
print("变量 var_1 的数值：",var_1.read_value())
#变量赋值
var_value_1=[[1,2,3],[4,5,6]]
var_1.assign(var_value_1)
print("变量 var_1 赋值后的数值：",var_1.read_value())
```

输出：

```
变量 var_1 的数值： tf.Tensor(
[[1. 1. 1.]
 [1. 1. 1.]], shape=(2, 3), dtype=float32)
变量 var_1 赋值后的数值： tf.Tensor(
[[1. 2. 3.]
 [4. 5. 6.]], shape=(2, 3), dtype=float32)
```

代码：

```
#变量加法
var_1.assign_add(tf.ones([2,3]))
var_1
```

输出：

```
<tf.Variable 'Variable:0' shape=(2, 3) dtype=float32, numpy=
array([[2., 3., 4.],
       [5., 6., 7.]], dtype=float32)>
```

1.2.1.2 tensor 切片与索引

1.2.1.2.1 切片

切片的方式主要有：

- [start: end]：从 tensor 的开始位置到结束位置的数据切片；
- [start :end :step]或者[::step]：从 tensor 的开始位置到结束位置每隔 step 的数据切片；
- [::-1]:负数表示倒序切片；
- ‘...’：任意长。

代码：

```
#创建一个 4 维 tensor。tensor 包含 4 张图片，每张图片的大小为 100*100*3
tensor_h = tf.random.normal([4,100,100,3])
tensor_h
```

输出：

```
<tf.Tensor: shape=(4, 100, 100, 3), dtype=float32, numpy=
array([[[[ 1.68444023e-01, -7.46562362e-01, -4.34964240e-01],
         [-4.69263226e-01,  6.26460612e-01,  1.21065331e+00],
         [ 7.21675277e-01,  4.61057723e-01, -9.20868576e-01],
         ...,
```

代码：

```
#取出第一张图片
tensor_h[0,:,:,:]
```

输出：

```
<tf.Tensor: shape=(100, 100, 3), dtype=float32, numpy=
array([[[ 1.68444023e-01, -7.46562362e-01, -4.34964240e-01],
        [-4.69263226e-01,  6.26460612e-01,  1.21065331e+00],
        [ 7.21675277e-01,  4.61057723e-01, -9.20868576e-01],
        ...,
```

代码：

```
#每两张图片取出一张的切片
tensor_h[::2,...]
```

输出：

```
<tf.Tensor: shape=(2, 100, 100, 3), dtype=float32, numpy=
array([[[[ 1.68444023e-01, -7.46562362e-01, -4.34964240e-01],
         [-4.69263226e-01,  6.26460612e-01,  1.21065331e+00],
         [ 7.21675277e-01,  4.61057723e-01, -9.20868576e-01],
         ...,
```

代码：

```
#倒序切片
tensor_h[::-1]
```

输出：

```
<tf.Tensor: shape=(4, 100, 100, 3), dtype=float32, numpy=
array([[[[-1.70684665e-01,  1.52386248e+00, -1.91677585e-01],
         [-1.78917408e+00, -7.48436213e-01,  6.10363662e-01],
         [ 7.64770031e-01,  6.06725179e-02,  1.32704067e+00],
         ...,
```

1.2.1.2.2 索引

索引的基本格式：a[d1][d2][d3]

代码：

```
#取出第一张图片第二个通道中在[20,40]位置的像素点
tensor_h[0][19][39][1]
```

输出：

```
<tf.Tensor: shape=(), dtype=float32, numpy=0.38231283>
```

如果要提取的索引不连续的话，在 TensorFlow 中，常见的用法为 tf.gather 和 tf.gather_nd。

在某一维度进行索引。

tf.gather(params, indices,axis=None)：

- params：输入张量；
- indices：取出数据的索引；
- axis：所取数据所在维度。

代码：

```
#取出 tensor_h（[4,100,100,3]）中，第 1，2，4 张图像。
indices = [0,1,3]
tf.gather(tensor_h,axis=0,indices=indices,batch_dims=1)
```

输出：

```
<tf.Tensor: shape=(3, 100, 100, 3), dtype=float32, numpy=
array([[[[ 1.68444023e-01, -7.46562362e-01, -4.34964240e-01],
         [-4.69263226e-01,  6.26460612e-01,  1.21065331e+00],
         [ 7.21675277e-01,  4.61057723e-01, -9.20868576e-01],
         ...,
```

tf.gather_nd 允许在多维上进行索引: tf.gather_nd(params,indices)：

- params：输入张量；
- indices：取出数据的索引，一般为多维列表。

代码：

```
#取出 tensot_h([4,100,100,3])中，第一张图像第一个维度中[1,1]的像素点；第二张图片第一像素点中[2,2]的像素点
indices = [[0,1,1,0],[1,2,2,0]]
tf.gather_nd(tensor_h,indices=indices)
```

输出：

```
<tf.Tensor: shape=(2,), dtype=float32, numpy=array([0.5705869, 0.9735735], dtype=float32)>
```

1.2.1.3 张量的维度变化

1.2.1.3.1 维度查看

代码：

```
const_d_1 = tf.constant([[1, 2, 3, 4]],shape=[2,2], dtype=tf.float32)
#查看维度常用的三种方式
print(const_d_1.shape)
print(const_d_1.get_shape())
print(tf.shape(const_d_1))#输出为张量，其数值表示的是所查看张量维度大小
```

输出：

```
(2, 2)
(2, 2)
tf.Tensor([2 2], shape=(2,), dtype=int32)
```

可以看出.shape 和.get_shape()都是返回 TensorShape 类型对象，而 tf.shape(x)返回的是 Tensor 类型对象。

1.2.1.3.2 维度重组

tf.reshape(tensor,shape,name=None)：

- tensor：输入张量；
- shape：重组后张量的维度。

代码：

```
reshape_1 = tf.constant([[1,2,3],[4,5,6]])
print(reshape_1)
tf.reshape(reshape_1, (3,2))
```

输出：

```
<tf.Tensor: shape=(3, 2), dtype=int32, numpy=
array([[1, 2],
       [3, 4],
       [5, 6]], dtype=int32)>
```

1.2.1.3.3 维度增加

tf.expand_dims(input,axis,name=None):

- input：输入张量；
- axis：在第 axis 维度后增加一个维度。在输入 D 尺寸的情况下，轴必须在[-(D + 1),D]（含）范围内。负数代表倒序。

代码：

```
#生成一个大小为 100*100*3 的张量来表示一张尺寸为 100*100 的三通道彩色图片
expand_sample_1 = tf.random.normal([100,100,3], seed=1)
print("原始数据尺寸：",expand_sample_1.shape)
print("在第一个维度前增加一个维度(axis=0)：",tf.expand_dims(expand_sample_1, axis=0).shape)
print("在第二个维度前增加一个维度(axis=1)：",tf.expand_dims(expand_sample_1, axis=1).shape)
print("在最后一个维度后增加一个维度(axis=-1)：",tf.expand_dims(expand_sample_1, axis=-1).shape)
```

输出：

```
原始数据尺寸： (100, 100, 3)
在第一个维度前增加一个维度(axis=0)： (1, 100, 100, 3)
在第二个维度前增加一个维度(axis=1)： (100, 1, 100, 3)
在最后一个维度后增加一个维度(axis=-1)： (100, 100, 3, 1)
```

1.2.1.3.4 维度减少

tf.squeeze(input,axis=None,name=None):

- input：输入张量；
- axis：axis=1，表示要删掉的为 1 的维度。

代码：

```
#生成一个大小为 100*100*3 的张量来表示一张尺寸为 100*100 的三通道彩色图片
squeeze_sample_1 = tf.random.normal([1,100,100,3])
print("原始数据尺寸：",squeeze_sample_1.shape)
squeezed_sample_1 = tf.squeeze(expand_sample_1)
print("维度压缩后的数据尺寸：",squeezed_sample_1.shape)
```

```
输出：
原始数据尺寸： (1, 100, 100, 3)
维度压缩后的数据尺寸： (100, 100, 3)
```

1.2.1.3.5 转置

tf.transpose(a,perm=None,conjugate=False,name='transpose')：

- a：输入张量；
- perm：张量的尺寸排列；一般用于高维数组的转置。
- conjugate：表示复数转置；
- name：名称。

代码：

```
#低维的转置问题比较简单，输入需转置张量调用 tf.transpose
trans_sample_1 = tf.constant([1,2,3,4,5,6],shape=[2,3])
print("原始数据尺寸：",trans_sample_1.shape)
transposed_sample_1 = tf.transpose(trans_sample_1)
print("转置后数据尺寸：",transposed_sample_1.shape)
```

输出：

```
原始数据尺寸： (2, 3)
转置后数据尺寸： (3, 2)
```

代码：

```
'''高维数据转置需要用到 perm 参数，perm 代表输入张量的维度排列。
对于一个三维张量来说，其原始的维度排列为[0,1,2]（perm）分别代表高维数据的长宽高。
通过改变 perm 中数值的排列，可以对数据的对应维度进行转置'''
#生成一个大小为$*100*200*3 的张量来表示 4 张尺寸为 100*200 的三通道彩色图片
trans_sample_2 = tf.random.normal([4,100,200,3])
print("原始数据尺寸：",trans_sample_2.shape)
#对 4 张图像的长宽进行对调。原始 perm 为[0,1,2,3]，现变为[0,2,1,3]
transposed_sample_2 = tf.transpose(trans_sample_2,[0,2,1,3])
print("转置后数据尺寸：",transposed_sample_2.shape)
```

输出：

```
原始数据尺寸： (4, 100, 200, 3)
转置后数据尺寸： (4, 200, 100, 3)
```

1.2.1.3.6 广播（broadcast_to）

利用把 broadcast_to 可以将小维度推广到大维度。

tf.broadcast_to(input,shape,name=None)：

- input：输入张量；
- shape：输出张量的尺寸。

代码：

```
broadcast_sample_1 = tf.constant([1,2,3,4,5,6])
print("原始数据：",broadcast_sample_1.numpy())
broadcasted_sample_1 = tf.broadcast_to(broadcast_sample_1,shape=[4,6])
print("广播后数据：",broadcasted_sample_1.numpy())
```

输出：

```
原始数据： [1 2 3 4 5 6]
广播后数据： [[1 2 3 4 5 6]
 [1 2 3 4 5 6]
 [1 2 3 4 5 6]
 [1 2 3 4 5 6]]
```

代码：

```
#运算时，当两个数组的形状不同时，与 numpyy 一样，TensorFlow 将自动触发广播机制。
a = tf.constant([[ 0, 0, 0],
                 [10,10,10],
                 [20,20,20],
                 [30,30,30]])
b = tf.constant([1,2,3])
print(a + b)
```

```
输出：
tf.Tensor(
[[ 1  2  3]
 [11 12 13]
 [21 22 23]
 [31 32 33]], shape=(4, 3), dtype=int32)
```

1.2.1.4 张量的算术运算

1.2.1.4.1 算术运算符

算术运算主要包括了：加(tf.add)、减(tf.subtract)、乘(tf.multiply)、除(tf.divide)、取对数（tf.math.log）和指数（tf.pow）等。 因为调用比较简单，下面只演示一个加法例子。

代码：

```
a = tf.constant([[3, 5], [4, 8]])
b = tf.constant([[1, 6], [2, 9]])
print(tf.add(a, b))
```

输出：

```
tf.Tensor(
[[ 4 11]
 [ 6 17]], shape=(2, 2), dtype=int32)
```

1.2.1.4.2 矩阵乘法运算

矩阵乘法运算的实现通过调用 tf.matmul。

代码：

```
tf.matmul(a,b)
```

输出：

```
<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[13, 63],
       [20, 96]], dtype=int32)>
```

1.2.1.4.3 张量的数据统计

张量的数据统计主要包括：

- tf.reduce_min/max/mean()：求解最小值最大值和均值函数；
- tf.argmax()/tf.argmin()：求最大最小值位置；
- tf.equal()：逐个元素判断两个张量是否相等；
- tf.unique()：除去张量中的重复元素。
- tf.nn.in_top_k(prediction, target, K):用于计算预测值和真是值是否相等，返回一个 bool 类型的张量。

下面演示 tf.argmax()的用法：

返回最大值所在的下标

- tf.argmax(input,axis)：
- input：输入张量；
- axis：按照 axis 维度，输出最大值。

代码：

```
argmax_sample_1 = tf.constant([[1,3,2],[2,5,8],[7,5,9]])
print("输入张量：",argmax_sample_1.numpy())
max_sample_1 = tf.argmax(argmax_sample_1, axis=0)
max_sample_2 = tf.argmax(argmax_sample_1, axis=1)
print("按列寻找最大值的位置：",max_sample_1.numpy())
print("按行寻找最大值的位置：",max_sample_2.numpy())
```

输出：

```
输入张量： [[1 3 2]
 [2 5 8]
 [7 5 9]]
按列寻找最大值的位置： [2 1 2]
按行寻找最大值的位置： [1 2 2]
```

1.2.1.5 基于维度的算术操作

TensorFlow 中，tf.reduce_*一系列操作等都造成张量维度的减少。这一系列操作都可以对一个张量在维度上的元素进行操作，如按行求平均，求取张量中所有元素的乘积等。

常用的包括：tf.reduce_sum(加法)、tf.reduce_prod（乘法）、tf.reduce_min（最小）、tf.reduce_max（最大）、tf.reduce_mean（均值）、tf.reduce_all（逻辑和）、tf.reduce_any（逻辑或）和 tf.reduce_logsumexp（log(sum(exp))操作）等。

这些操作的使用方法都相似，下面只演示 tf.reduce_sum 的操作案例。

计算一个张量的各个维度上元素的总和

tf.reduce_sum(input_tensor, axis=None, keepdims=False,name=None)：

- input_tensor：输入张量；
- axis：指定需要计算的轴，如果不指定，则计算所有元素的均值;
- keepdims：是否降维度，设置为 True，输出的结果保持输入 tensor 的形状，设置为 False，输出结果会降低维度;
- name：操作名称。

代码：

```
reduce_sample_1 = tf.constant([1,2,3,4,5,6],shape=[2,3])
print("原始数据",reduce_sample_1.numpy())
print("计算张量中所有元素的和（axis=None）：",tf.reduce_sum(reduce_sample_1,axis=None).numpy())
print("按列计算，分别计算各列的和（axis=0）：",tf.reduce_sum(reduce_sample_1,axis=0).numpy())
print("按行计算，分别计算各列的和（axis=1）：",tf.reduce_sum(reduce_sample_1,axis=1).numpy())
```

输出：

```
原始数据 [[1 2 3]
 [4 5 6]]
计算张量中所有元素的和（axis=None）： 21
按行计算，取出行，分别计算各列的和（axis=0）： [5 7 9]
按列计算，取出列，分别计算各列的和（axis=1）： [6 15]
```

1.2.1.6 张量的拼接与分割

1.2.1.6.1 张量的拼接

TensorFlow 中，张量拼接的操作主要包括：

- tf.contact()：将向量按指定维连起来，其余维度不变。
- tf.stack() ：将一组 R 维张量变为 R+1 维张量，拼接前后维度变化。

tf.concat(values, axis, name='concat')：

- values：输入张量；
- axis：指定拼接维度；
- name：操作名称。

代码：

```
concat_sample_1 = tf.random.normal([4,100,100,3])
concat_sample_2 = tf.random.normal([40,100,100,3])
print("原始数据的尺寸分别为：",concat_sample_1.shape,concat_sample_2.shape)
concated_sample_1 = tf.concat([concat_sample_1,concat_sample_2],axis=0)
print("拼接后数据的尺寸：",concated_sample_1.shape)
```

输出：

```
原始数据的尺寸分别为： (4, 100, 100, 3) (40, 100, 100, 3)
拼接后数据的尺寸： (44, 100, 100, 3)
```

在原来矩阵基础上增加了一个维度，也是同样的道理，axis 决定维度增加的位置。

tf.stack(values, axis=0, name='stack')：

- values：输入张量；一组相同形状和数据类型的张量。
- axis：指定拼接维度；
- name：操作名称。

代码：

```
stack_sample_1 = tf.random.normal([100,100,3])
stack_sample_2 = tf.random.normal([100,100,3])
print("原始数据的尺寸分别为：",stack_sample_1.shape, stack_sample_2.shape)
#拼接后维度增加。axis=0，则在第一个维度前增加维度。
stacked_sample_1 = tf.stack([stack_sample_1, stack_sample_2],axis=0)
print("拼接后数据的尺寸：",stacked_sample_1.shape)
```

输出：

```
原始数据的尺寸分别为： (100, 100, 3) (100, 100, 3)
拼接后数据的尺寸： (2, 100, 100, 3)
```

1.2.1.6.2 张量的分割

TensorFlow 中，张量分割的操作主要包括：

- tf.unstack()：将张量按照特定维度分解。
- tf.split()：将张量按照特定维度划分为指定的分数。

与 tf.unstack()相比，tf.split()更佳灵活。

tf.unstack(value,num=None,axis=0,name='unstack')：

- value：输入张量；
- num：表示输出含有 num 个元素的列表，num 必须和指定维度内元素的个数相等。通常可以忽略不写这个参数。
- axis：指明根据数据的哪个维度进行分割；
- name：操作名称。

代码：

```
#按照第一个维度对数据进行分解，分解后的数据以列表形式输出。
tf.unstack(stacked_sample_1,axis=0)
```

输出：

```
[<tf.Tensor: shape=(100, 100, 3), dtype=float32, numpy=
 array([[[ 0.0665694 ,  0.7110351 ,  1.907618  ],
         [ 0.84416866,  1.5470593 , -0.5084871 ],
         [-1.9480026 , -0.9899087 , -0.09975405],
         ...,
```

tf.split(value, num_or_size_splits, axis=0)：

- value：输入张量；
- num_or_size_splits：准备切成几份
- axis：指明根据数据的哪个维度进行分割。

tf.split()的分割方式有两种：

1. 如果 num_or_size_splits 传入的是一个整数，那直接在 axis=D 这个维度上把张量平均切分成几个小张量。

2. 如果 num_or_size_splits 传入的是一个向量，则在 axis=D 这个维度上把张量按照向量的元素值切分成几个小张量。

代码：

```
import numpy as np
split_sample_1 = tf.random.normal([10,100,100,3])
print("原始数据的尺寸为：",split_sample_1.shape)
splited_sample_1 = tf.split(split_sample_1, num_or_size_splits=5,axis=0)
print("当 m_or_size_splits=10，分割后数据的尺寸为：",np.shape(splited_sample_1))
splited_sample_2 = tf.split(split_sample_1, num_or_size_splits=[3,5,2],axis=0)
print("当 num_or_size_splits=[3,5,2]，分割后数据的尺寸分别为：",
      np.shape(splited_sample_2[0]),
      np.shape(splited_sample_2[1]),
      np.shape(splited_sample_2[2]))
```

输出：

```
原始数据的尺寸为： (10, 100, 100, 3)
当 m_or_size_splits=10，分割后数据的尺寸为： (5, 2, 100, 100, 3)
当 num_or_size_splits=[3,5,2]，分割后数据的尺寸分别为： (3, 100, 100, 3) (5, 100, 100, 3) (2, 100, 100, 3)
```

1.2.1.7 张量排序

TensorFlow 中，张量排序的操作主要包括：

- tf.sort()：按照升序或者降序对张量进行排序，返回排序后的张量。
- tf.argsort()：按照升序或者降序对张量进行排序,但返回的是索引。
- tf.nn.top_k()：返回前 k 个最大值。

tf.sort/argsort(input, direction, axis)：

- input：输入张量；

- direction：排列顺序，可为 DESCENDING 降序或者 ASCENDING（升序）。默认为 ASCENDING（升序）；
- axis：按照 axis 维度进行排序。默认 axis=-1 最后一个维度。

代码：

```
sort_sample_1 = tf.random.shuffle(tf.range(10))
print("输入张量：",sort_sample_1.numpy())
sorted_sample_1 = tf.sort(sort_sample_1, direction="ASCENDING")
print("生序排列后的张量：",sorted_sample_1.numpy())
sorted_sample_2 = tf.argsort(sort_sample_1,direction="ASCENDING")
print("生序排列后，元素的索引：",sorted_sample_2.numpy())
```

输出：

```
输入张量： [1 8 7 9 6 5 4 2 3 0]
生序排列后的张量： [0 1 2 3 4 5 6 7 8 9]
生序排列后，元素的索引： [9 0 7 8 6 5 4 2 1 3]
```

tf.nn.top_k(input,K,sorted=TRUE):

- input：输入张量；
- K：需要输出的前 k 个值及其索引。
- sorted： sorted=TRUE 表示升序排列；sorted=FALSE 表示降序排列。

返回两个张量：

- values：也就是每一行的最大的 k 个数字
- indices：这里的下标是在输入的张量的最后一个维度的下标

代码：

```
values, index = tf.nn.top_k(sort_sample_1,5)
print("输入张量：",sort_sample_1.numpy())
print("升序排列后的前 5 个数值：", values.numpy())
print("升序排列后的前 5 个数值的索引：", index.numpy())
```

输出：

```
输入张量： [1 8 7 9 6 5 4 2 3 0]
升序排列后的前 5 个数值： [9 8 7 6 5]
升序排列后的前 5 个数值的索引： [3 1 2 4 5]
```

1.2.2 TensorFlow2 Eager Execution 模式

Eager Execution 介绍：

TensorFlow 的 Eager Execution 模式是一种命令式编程（imperative programming），这和原生 Python 是一致的，当你执行某个操作时，可以立即返回结果的。

Graph 模式介绍：

TensorFlow1.0 一直是采用 Graph 模式，即先构建一个计算图，然后需要开启 Session，喂进实际的数据才真正执行得到结果。

Eager Execution 模式下，我们可以更容易 debug 代码，但是代码的执行效率更低。

下面我们在 Eager Execution 和 Graph 模式下，用 TensorFlow 实现简单的乘法，来对比两个模式的区别。

代码：

```
x = tf.ones((2, 2), dtype=tf.dtypes.float32)
y = tf.constant([[1, 2],
                 [3, 4]], dtype=tf.dtypes.float32)
z = tf.matmul(x, y)
print(z)
```

输出：

```
tf.Tensor(
[[4. 6.]
 [4. 6.]], shape=(2, 2), dtype=float32)
```

代码：

```
#在 TensorFlow 2 版本中使用 1.X 版本的语法；可以使用 2.0 中的 v1 兼容包来沿用 1.x 代码，并在代码中关闭 eager 运算。
import TensorFlow.compat.v1 as tf
tf.disable_eager_execution()
#创建 graph，定义计算图
a = tf.ones((2, 2), dtype=tf.dtypes.float32)
b = tf.constant([[1, 2],
                 [3, 4]], dtype=tf.dtypes.float32)
c = tf.matmul(a, b)
#开启绘画，进行运算后，才能取出数据。
with tf.Session() as sess:
    print(sess.run(c))
```

输出：

```
[[4. 6.]
 [4. 6.]]
```

首先重启一下 kernel，使得 TensorFlow 恢复到 2.0 版本并打开 eager execution 模式。 Eager Execution 模式的另一个优点是可以使用 Python 原生功能，比如下面的条件判断：

代码：

```
import TensorFlow as tf
thre_1 = tf.random.uniform([], 0, 1)
x = tf.reshape(tf.range(0, 4), [2, 2])
print(thre_1)
if thre_1.numpy() > 0.5:
    y = tf.matmul(x, x)
else:
    y = tf.add(x, x)
```

输出：

```
tf.Tensor(0.11304152, shape=(), dtype=float32)
```

这种动态控制流主要得益于 eager 执行得到 Tensor 可以取出 numpy 值，这避免了使用 Graph 模式下的 tf.cond 和 tf.while 等算子。

1.2.3 TensorFlow2 AutoGraph

当使用 tf.function 装饰器注释函数时，可以像调用任何其他函数一样调用它。它将被编译成图，这意味着可以获得更高效地在在 GPU 或 TPU 上运行。此时函数变成了一个 TensorFlow 中的 operation。我们可以直接调用函数，输出返回值，但是函数内部是在 graph 模式下执行的，无法直接查看中间变量数值

代码：

```
@tf.function
def simple_nn_layer(w,x,b):
    print(b)
    return tf.nn.relu(tf.matmul(w, x)+b)


w = tf.random.uniform((3, 3))
x = tf.random.uniform((3, 3))
b = tf.constant(0.5, dtype='float32')

simple_nn_layer(w,x,b)
```

输出：

```
Tensor("b:0", shape=(), dtype=float32)
<tf.Tensor: shape=(3, 3), dtype=float32, numpy=
array([[1.4121541 , 1.1626956 , 1.2527422 ],
       [1.2903953 , 1.0956903 , 1.1309073 ],
       [1.1039395 , 0.92851776, 1.0752096 ]], dtype=float32)>
```

通过输出结果可知，无法直接查看函数内部 b 的数值，而返回值可以通过.numpy()查看。

通过相同的操作（执行一层 lstm 计算），比较 graph 和 eager execution 模式的性能。

代码：

```
#timeit 测量小段代码的执行时间
import timeit
#创建一个卷积层。
CNN_cell = tf.keras.layers.Conv2D(filters=100,kernel_size=2,strides=(1,1))

#利用@tf.function，将操作转化为 graph。
@tf.function
def CNN_fn(image):
    return CNN_cell(image)

image = tf.zeros([100, 200, 200, 3])

#比较两者的执行时间
CNN_cell(image)
CNN_fn(image)
#调用 timeit.timeit，测量代码执行 10 次的时间
print("eager execution 模式下做一层 CNN 卷积层运算的时间:", timeit.timeit(lambda: CNN_cell(image),
number=10))
print("graph 模式下做一层 CNN 卷积层运算的时间:", timeit.timeit(lambda: CNN_fn(image), number=10))
```

输出：

```
eager execution 模式下做一层 CNN 卷积层运算的时间: 18.26327505100926
graph 模式下做一层 CNN 卷积层运算的时间: 6.740775318001397
```

通过比较，我们可以发现 graph 模式下代码执行效率要高出许多。因此我们以后，可以多尝试用 @tf.function 功能，提高代码运行效率。

2 TensorFlow 2 常用模块介绍

2.1 实验介绍

本节将为大家介绍 TensorFlow 2 常用模块，主要包括：

- tf.data：实现对数据集的操作；

  包括读取从内存中直接读取数据集、读取 CSV 文件、读取 tfrecord 文件和数据增强等。

- tf.image：实现对图像处理的操作；

  包括图像亮度变换、饱和度变换、图像尺寸变换、图像旋转和边缘检测等操作。

- tf.gfile：实现对文件的操作；

  包括对文件的读写操作、文件重命名和文件夹操作等。

- tf.keras：用于构建和训练深度学习模型的高阶 API；
- tf.distributions 等等。

本节我们将重点聚焦到 tf.keras 模块，为后面深度学习建模打下基础。

2.2 实验目的

掌握 tf.keras 中常用的深度学习建模接口。

2.3 实验步骤

2.3.1 模型构建

2.3.1.1 模型堆叠（tf.keras.Sequential）

最常见的模型构建方法是层的堆叠，我们通常会使用 tf.keras.Sequential。

代码：

```
import TensorFlow.keras.layers as layers
model = tf.keras.Sequential()
model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

2.3.1.2 函数式模型构建

函数式模型主要利用 tf.keras.Input 和 tf.keras.Model 构建，比 tf.keras.Sequential 模型要复杂，但是效果很好，可以同时/分阶段输入变量，分阶段输出数据； 你的模型需要多于一个的输出，那么需要选择函数式模型。

模型堆叠（.Sequential）vs 函数式模型（Model）：

tf.keras.Sequential 模型是层的简单堆叠，无法表示任意模型。使用 Keras 的函数式模型可以构建复杂的模型拓扑，例如：

- 多输入模型；
- 多输出模型；
- 具有共享层的模型；
- 具有非序列数据流的模型（例如，残差连接）。

代码：

```
# 以上一层的输出作为下一层的输入
x = tf.keras.Input(shape=(32,))
h1 = layers.Dense(32, activation='relu')(x)
h2 = layers.Dense(32, activation='relu')(h1)
y = layers.Dense(10, activation='softmax')(h2)
model_sample_2 = tf.keras.models.Model(x, y)

#打印模型信息
model_sample_2.summary()
```

输出：

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 32)]              0
_________________________________________________________________
dense_3 (Dense)              (None, 32)                1056
_________________________________________________________________
dense_4 (Dense)              (None, 32)                1056
_________________________________________________________________
dense_5 (Dense)              (None, 10)                330
=================================================================
Total params: 2,442
Trainable params: 2,442
Non-trainable params: 0
_________________________________________________________________
```

2.3.1.3 网络层构建（tf.keras.layers）

tf.keras.layers 模块的主要作用为配置神经网络层。其中常用的类包括：

- tf.keras.layers.Dense：构建全连接层；
- tf.keras.layers.Conv2D：构建 2 维卷积层；
- tf.keras.layers.MaxPooling2D/AveragePooling2D：构建最大/平均池化层；
- tf.keras.layers.RNN：构建循环神经网络层；
- tf.keras.layers.LSTM/tf.keras.layers.LSTMCell：构建 LSTM 网络层/LSTM unit；
- tf.keras.layers.GRU/tf.keras.layers.GRUCell：构建 GRU unit/GRU 网络层；
- tf.keras.layers.Embedding 嵌入层将正整数（下标）转换为具有固定大小的向量，如 [[4],[20]]->[[0.25,0.1],[0.6,-0.2]]。Embedding 层只能作为模型的第一层；
- tf.keras.layers.Dropout：构建 dropout 层等。

下面主要讲解 tf.keras.layers.Dense、 tf.keras.layers.Conv2D、tf.keras.layers.MaxPooling2D/AveragePooling2D 和 tf.keras.layers.LSTM/tf.keras.layers.LSTMCell。

tf.keras.layers 中主要的网络配置参数如下：

- activation：设置层的激活函数。默认情况下，系统不会应用任何激活函数。
- kernel_initializer 和 bias_initializer：创建层权重（核和偏置）的初始化方案。默认为 "Glorot uniform" 初始化器。
- kernel_regularizer 和 bias_regularizer：应用层权重（核和偏置）的正则化方案，例如 L1 或 L2 正则化。默认情况下，系统不会应用正则化函数。

2.3.1.3.1 tf.keras.layers.Dense

tf.keras.layers.Dense 可配置的参数，主要有：

- units: 神经元个数；
- activation: 激活函数；
- use_bias: 是否使用偏置项。默认为使用；
- kernel_initializer: 创建层权重核的初始化方案；
- bias_initializer: 创建层权重偏置的初始化方案；
- kernel_regularizer: 应用层权重核的正则化方案；
- bias_regularizer: 应用层权重偏置的正则化方案；
- activity_regularizer：施加在输出上的正则项，为 Regularizer 对象；
- kernel_constraint: 施加在权重上的约束项；
- bias_constraint: 施加在权重上的约束项。

代码：

```
#创建包含 32 个神经元的全连接层，其中的激活函数设置为 sigmoid。
#activation 参数可以是函数名称字符串，如'sigmoid'；也可以是函数对象，如 tf.sigmoid。
layers.Dense(32, activation='sigmoid')
layers.Dense(32, activation=tf.sigmoid)

#设置 kernel_initializer 参数
layers.Dense(32, kernel_initializer=tf.keras.initializers.he_normal)
#设置 kernel_regularizer 为 L2 正则
layers.Dense(32, kernel_regularizer=tf.keras.regularizers.l2(0.01))
```

输出：

```
<TensorFlow.python.keras.layers.core.Dense at 0x130c519e8>
```

2.3.1.3.2 tf.keras.layers.Conv2D

tf.keras.layers.Conv2D 可配置的参数，主要有：

- filters：卷积核的数目（即输出的维度）；
- kernel_size：卷积核的宽度和长度；
- strides：卷积的步长。
- padding：补 0 策略。

■ padding=“valid”代表只进行有效的卷积，即对边界数据不处理。padding=“same”代表保留边界处的卷积结果，通常会导致输出 shape 与输入 shape 相同；

- activation：激活函数；
- data_format：数据格式，为“channels_first”或“channels_last”之一。以 128x128 的 RGB 图像为例，“channels_first”应将数据组织为（3,128,128），而“channels_last”应将数据组织为（128,128,3）。该参数的默认值是~/.keras/keras.json 中设置的值，若从未设置过，则为“channels_last”。

- 其他参数还包括：use_bias；kernel_initializer；bias_initializer；kernel_regularizer；bias_regularizer；activity_regularizer；kernel_constraints；bias_constraints。

代码：

```
layers.Conv2D(64,[1,1],2,padding='same',activation="relu")
```

输出：

```
<TensorFlow.python.keras.layers.convolutional.Conv2D at 0x106c510f0>
```

2.3.1.3.3 tf.keras.layers.MaxPooling2D/AveragePooling2D

tf.keras.layers.MaxPooling2D/AveragePooling2D 可配置的参数，主要有：

- pool_size：池化 kernel 的大小。如取矩阵（2，2）将使图片在两个维度上均变为原长的一半。为整数意为各个维度值都为该数字。
- strides：步长值。

- 其他参数还包括：padding；data_format。

代码：

```
layers.MaxPooling2D(pool_size=(2,2),strides=(2,1))
```

输出：

```
<TensorFlow.python.keras.layers.pooling.MaxPooling2D at 0x132ce1f98>
```

2.3.1.3.4 tf.keras.layers.LSTM/tf.keras.layers.LSTMCell

tf.keras.layers.LSTM/tf.keras.layers.LSTMCell 可配置的参数，主要有：

- units：输出维度；
- input_shape (timestep, input_dim),timestep 可以设置为 None,input_dim 为输入数据维度；
- activation：激活函数；
- recurrent_activation：为循环步施加的激活函数；
- return_sequences：=True 时，返回全部序列；=False 时，返回输出序列中的最后一个 cell 的输出；
- return_state：布尔值。除了输出之外是否返回最后一个状态；
- dropout：0~1 之间的浮点数，控制输入线性变换的神经元断开比例；

- recurrent_dropout：0~1 之间的浮点数，控制循环状态的线性变换的神经元断开比例。

代码：

```
import numpy as np
inputs = tf.keras.Input(shape=(3, 1))
lstm = layers.LSTM(1, return_sequences=True)(inputs)
model_lstm_1 = tf.keras.models.Model(inputs=inputs, outputs=lstm)

inputs = tf.keras.Input(shape=(3, 1))
lstm = layers.LSTM(1, return_sequences=False)(inputs)
model_lstm_2 = tf.keras.models.Model(inputs=inputs, outputs=lstm)



# t1, t2, t3 序列
data = [[[0.1],
  [0.2],
  [0.3]]]
print(data)
print("当 return_sequences=True 时的输出",model_lstm_1.predict(data))
print("当 return_sequences=False 时的输出",model_lstm_2.predict(data))
```

输出：

```
[[[0.1], [0.2], [0.3]]]
当 return_sequences=True 时的输出 [[[-0.0106758 ]
  [-0.02711176]
  [-0.04583194]]]
当 return_sequences=False 时的输出 [[0.05914127]]
```

LSTMcell 是 LSTM 层的实现单元。

- LSTM 是一个 LSTM 网络层
- LSTMCell 是一个单步的计算单元，即一个 LSTM UNIT。

```
#LSTM
tf.keras.layers.LSTM(16, return_sequences=True)

#LSTMCell
x = tf.keras.Input((None, 3))
y = layers.RNN(layers.LSTMCell(16))(x)
model_lstm_3= tf.keras.Model(x, y)
```

2.3.2 训练与评估

2.3.2.1 模型编译，确定训练流程。

构建好模型后，通过调用 compile 配置该模型的学习流程：

- compile(optimizer='rmsprop', loss=None, metrics=None, loss_weights=None):
- optimizer：优化器；
- loss：损失函数，对于二分类任务就是交叉熵，回归任务就是 mse 之类的；
- metrics：在训练和测试期间的模型评估标准。比如 metrics = ['accuracy']。 指定不同的评估标准，需要传递一个字典，如 metrics = {'output_a'：'accuracy'}。
- loss_weights: 如果的模型有多个任务输出，在优化全局 loss 的时候，需要给每个输出指定相应的权重。

代码：

```
model = tf.keras.Sequential()
model.add(layers.Dense(10, activation='softmax'))
#确定优化器（optimizer）、损失函数（loss）、模型评估方法（metrics）
model.compile(optimizer=tf.keras.optimizers.Adam(0.001),
              loss=tf.keras.losses.categorical_crossentropy,
              metrics=[tf.keras.metrics.categorical_accuracy])
```

2.3.2.2 模型训练

fit(x=None, y=None, batch_size=None, epochs=1, verbose=1, callbacks=None, validation_split=0.0, validation_data=None, shuffle=True, class_weight=None, sample_weight=None, initial_epoch=0, steps_per_epoch=None, validation_steps=None):

- x: 输入训练数据；
- y: 目标（标签）数据；
- batch_size: 每次梯度更新的样本数。如果未指定，默认为 32；
- epochs：训练模型迭代轮次；
- verbose：0, 1 或 2。日志显示模式。 0 = 不显示, 1 = 进度条, 2 = 每轮显示一行；
- callbacks：在训练时使用的回调函数；
- validation_split：验证集与训练数据的比例；
- validation_data：验证集；这个参数会覆盖 validation_split；
- shuffle： 是否在每轮迭代之前混洗数据。当 steps_per_epoch 非 None 时，这个参数无效；
- initial_epoch: 开始训练的轮次，常用于恢复之前的训练权重；
- steps_per_epoch：steps_per_epoch = 数据集大小/batch_size；
- validation_steps：只有在指定了 steps_per_epoch 时才有用。停止前要验证的总步数（批次样本）。

代码：

```
import numpy as np

train_x = np.random.random((1000, 36))
train_y = np.random.random((1000, 10))

val_x = np.random.random((200, 36))
val_y = np.random.random((200, 10))

model.fit(train_x, train_y, epochs=10, batch_size=100,
          validation_data=(val_x, val_y))
```

输出：

```
Train on 1000 samples, validate on 200 samples
Epoch 1/10
1000/1000 [==============================] - 0s 488us/sample - loss: 12.6024 -
categorical_accuracy: 0.0960 - val_loss: 12.5787 - val_categorical_accuracy: 0.0850
Epoch 2/10
1000/1000 [==============================] - 0s 23us/sample - loss: 12.6007 -
categorical_accuracy: 0.0960 - val_loss: 12.5776 - val_categorical_accuracy: 0.0850
Epoch 3/10
1000/1000 [==============================] - 0s 31us/sample - loss: 12.6002 -
categorical_accuracy: 0.0960 - val_loss: 12.5771 - val_categorical_accuracy: 0.0850
...
Epoch 10/10
1000/1000 [==============================] - 0s 24us/sample - loss: 12.5972 -
categorical_accuracy: 0.0960 - val_loss: 12.5738 - val_categorical_accuracy: 0.0850

<TensorFlow.python.keras.callbacks.History at 0x130ab5518>
```

对于大型数据集可以使用 tf.data 构建训练输入。

代码：

```
dataset = tf.data.Dataset.from_tensor_slices((train_x, train_y))
dataset = dataset.batch(32)
dataset = dataset.repeat()
val_dataset = tf.data.Dataset.from_tensor_slices((val_x, val_y))
val_dataset = val_dataset.batch(32)
val_dataset = val_dataset.repeat()

model.fit(dataset, epochs=10, steps_per_epoch=30,
          validation_data=val_dataset, validation_steps=3)
```

输出：

```
Train for 30 steps, validate for 3 steps
Epoch 1/10
30/30 [==============================] - 0s 15ms/step - loss: 12.6243 - categorical_accuracy:
0.0948 - val_loss: 12.3128 - val_categorical_accuracy: 0.0833
...
30/30 [==============================] - 0s 2ms/step - loss: 12.5797 - categorical_accuracy:
0.0951 - val_loss: 12.3067 - val_categorical_accuracy: 0.0833
<TensorFlow.python.keras.callbacks.History at 0x132ab48d0>
```

2.3.2.3 回调函数

回调函数是传递给模型以自定义和扩展其在训练期间的行为的对象。我们可以编写自己的自定义回调，或使用 tf.keras.callbacks 中的内置函数，常用内置回调函数如下：

tf.keras.callbacks.ModelCheckpoint：定期保存模型。

tf.keras.callbacks.LearningRateScheduler：动态更改学习率。

tf.keras.callbacks.EarlyStopping：提前终止。

tf.keras.callbacks.TensorBoard：使用 TensorBoard。

代码：

```
#超参数设置
Epochs = 10

#定义一个学习率动态设置函数
def lr_Scheduler(epoch):
    if epoch > 0.9 * Epochs:
        lr = 0.0001
    elif epoch > 0.5 * Epochs:
        lr = 0.001
    elif epoch > 0.25 * Epochs:
        lr = 0.01
    else:
        lr = 0.1

    print(lr)
    return lr


callbacks = [
    #早停:
    tf.keras.callbacks.EarlyStopping(
        #不再提升的关注指标
        monitor='val_loss',
        #不再提升的阈值
        min_delta=1e-2,
        #不再提升的轮次
        patience=2),

    #定期保存模型:
     tf.keras.callbacks.ModelCheckpoint(
        #模型路径
        filepath='testmodel_{epoch}.h5',
        #是否保存最佳模型
        save_best_only=True,
        #监控指标
        monitor='val_loss'),

    #动态更改学习率
    tf.keras.callbacks.LearningRateScheduler(lr_Scheduler),

    #使用 TensorBoard
    tf.keras.callbacks.TensorBoard(log_dir='./logs')
```

输出：

```
Train on 1000 samples, validate on 200 samples
0
0.1
Epoch 1/10
1000/1000 [==============================] - 0s 155us/sample - loss: 12.7907 -
categorical_accuracy: 0.0920 - val_loss: 12.7285 - val_categorical_accuracy: 0.0750
1
0.1
Epoch 2/10
1000/1000 [==============================] - 0s 145us/sample - loss: 12.6756 -
categorical_accuracy: 0.0940 - val_loss: 12.8673 - val_categorical_accuracy: 0.0950
...
0.001
Epoch 10/10
1000/1000 [==============================] - 0s 134us/sample - loss: 12.3627 -
categorical_accuracy: 0.1020 - val_loss: 12.3451 - val_categorical_accuracy: 0.0900


<TensorFlow.python.keras.callbacks.History at 0x133d35438>
```

2.3.2.4 评估与预测

评估和预测函数：tf.keras.Model.evaluate 和 tf.keras.Model.predict 方法。

代码：

```
# 模型评估
test_x = np.random.random((1000, 36))
test_y = np.random.random((1000, 10))
model.evaluate(test_x, test_y, batch_size=32)
```

输出：

```
1000/1000 [==============================] - 0s 45us/sample - loss: 12.2881 -
categorical_accuracy: 0.0770
[12.288104843139648, 0.077]
```

代码：

```
# 模型预测
pre_x = np.random.random((10, 36))
result = model.predict(test_x,)
print(result)
```

输出：

```
[[0.04431767 0.24562006 0.05260926 ... 0.1016549  0.13826898 0.15511878]
 [0.06296062 0.12550288 0.07593573 ... 0.06219672 0.21190381 0.12361749]
 [0.07203944 0.19570401 0.11178136 ... 0.05625525 0.20609994 0.13041474]
 ...
 [0.09224506 0.09908539 0.13944311 ... 0.08630784 0.15009451 0.17172746]
 [0.08499582 0.17338121 0.0804626  ... 0.04409525 0.27150458 0.07133815]
 [0.05191234 0.11740112 0.08346355 ... 0.0842929  0.20141983 0.19982798]]
```

2.3.3 模型保存与恢复

2.3.3.1 保存和恢复整个模型

代码：

```
import numpy as np
# 模型保存
model.save('./model/the_save_model.h5')
# 导入模型
new_model = tf.keras.models.load_model('./model/the_save_model.h5')
new_prediction = new_model.predict(test_x)
#np.testing.assert_allclose: 判断两个对象的近似程度是否超出了指定的容差限。若是，则抛出异常。:
#atol:指定的容差限
np.testing.assert_allclose(result, new_prediction, atol=1e-6) # 预测结果一样
```

模型保存后可以在对应的文件夹中找到对应的权重文件。

2.3.3.2 只保存和加载网络权重

若权重名后有.h5 或.keras 后缀，则保存为 HDF5 格式文件，否则默认为 TensorFlow Checkpoint 格式文件。

代码：

```
model.save_weights('./model/model_weights')
model.save_weights('./model/model_weights.h5')
#权重加载
model.load_weights('./model/model_weights')
model.load_weights('./model/model_weights.h5')
```