第 3 章 深度学习框架

深度学习框架提供了一整套封装完善且使用方便、简单的 API，可以供人们快速地实现各种算法模型，同时它也提供了算法模型训练、评价、调优的方法，是深度学习领域工作、研究的神兵利器。

目前市场占有率最高的框架主要有 TensorFlow 和 PyTorch，它们的核心都是基于张量和计算图进行操作的。TensorFlow 各个版本的 API 管理相对烦琐，学习较困难，而 PyTorch 要容易得多，但是作为一个深度学习工程师，无论 TensorFlow 还是 PyTorch 都需要掌握，因为两者都可能会在工作中被用到。尤其 TensorFlow 在工业界中占有非常重要的地位，这是无法避开的，因此本章将重点梳理 TensorFlow 的 API，同时也将介绍 PyTorch，通过阅读本章可以实现 TensorFlow 和 PyTorch 相关 API 的无缝衔接，帮助读者开启实战深度学习之旅！

3.1 基本概念

在深度学习框架中有 4 种数据类型，分别如下：

(1) 标量，也就是常量。

(2) 向量，一维数据。

(3) 矩阵，二维数组。

(4) 张量，三维及以上的数据。

但是在广义上它们都被称为张量，又被称为 Tensor。无论是 TensorFlow 还是 PyTorch 或者其他框架的核心都是基于张量的数据操作。

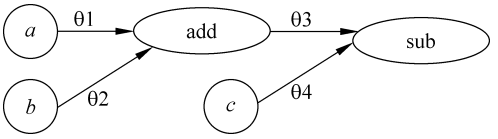


图 3-1 计算图

另一个比较重要的概念是计算图，即将算法模型的计算过程图形化地展示出来，可以方便地查看各个变量之间的关系及张量的流向，如图 3-1 所示。

a、b 表示数据的输入，add 表示数据的运算方法，又称为算子，其实现的计算为 $y=(a\times\theta1+b\times\theta2)\times\theta3-c\times\theta4$。深度学习框架会自动生成计算图，如图 3-2 所示。

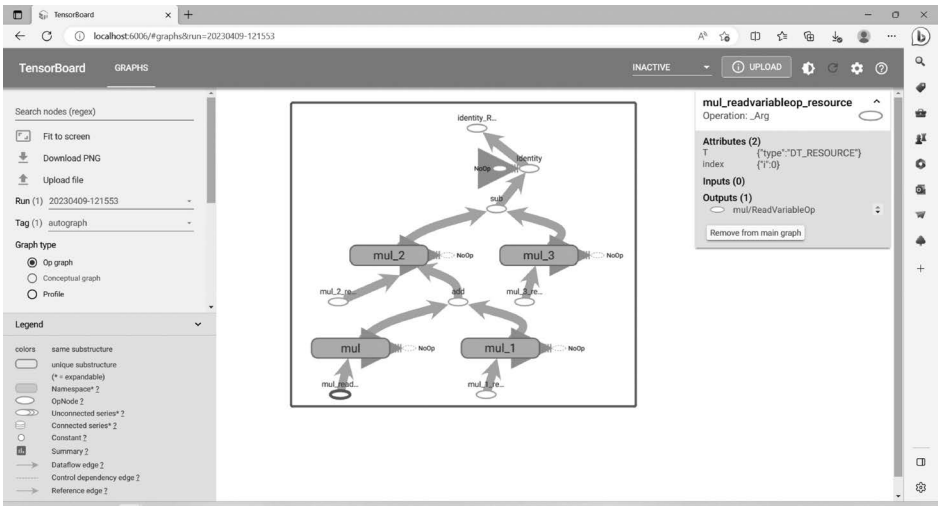


图 3-2　自动生成计算图

TensorFlow 2.x 中提供了动态图和静态图，而 PyTorch 提供的是动态图。所谓静态图是指先构建计算图，然后进行运算，虽然运行效率高，但是调试不方便，也不够灵活。动态图则是运算与计算图的构建同时进行，相对灵活且容易调试，但是运行效率有所下降。

例如静态图中需要先定义规则 $y=(a\times\theta1+b\times\theta2)\times\theta3-c\times\theta4$，然后将结构存储起来，然后把 $\theta1=1,\theta2=2,\theta3=3,\theta4=4$，放到计算图中计算。动态图则是 $y=(a\times1+b\times2)\times3-c\times4$，同时构建计算规则并填充数据，其区别可以类比为，建造房子时先搭建整个房子的框架，再砌砖，这就是静态图。边搭框架边砌砖，这就是动态图。

总结

TensorFlow 2.x 中提供了动态图和静态图，而 PyTorch 提供的是动态图。

练习

运行代码“第 3 章/TensorFlowAPI/计算图的生成.py”在 TensorBoard 中查看计算图的结构。

3.2　环境搭建

深度学习框架的运行环境分为 CPU 和 GPU(显卡)版本，安装 GPU 的 TensorFlow 运行环境，需要满足以下条件：

(1) 显卡支持 CUDA 架构。CUDA 是一种由 NVIDIA 推出的通用并行计算架构，该架构使 GPU 能够解决复杂的计算问题。

本机是否支持 CUDA 架构，可以在设备管理器中查看显卡的型号，如图 3-3 所示。

CUDA 支持的显卡型号及算力，可以在网站 https://developer.nvidia.com/zh-cn/cuda-gpus#compute 中可看，如图 3-4 所示。

图 3-3 设备管理器

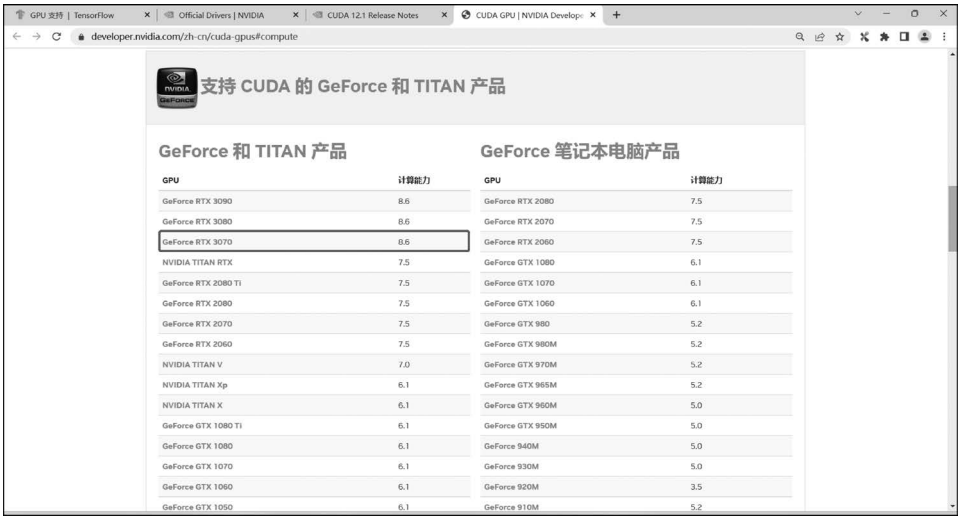


图 3-4 CUDA 支持显卡型号

一般来说集成显卡都不支持，独立显卡最少要在 1GB 以上才支持 GPU 的深度学习框架的运行。

(2) CUDA Toolkit 的版本与本机计算机的显卡驱动应保持一致。CUDA Toolkit 是 CUDA 开发工具包，主要包含了 CUDA-C、CUDA-C++的编译器、科学库、示例程序等。

通常在安装显卡驱动时会默认安装 CUDA Driver，但是不会安装 CUDA Toolkit。因为只安装 CUDA Driver 就可以正常办公了，而 CUDA Toolkit 通常是为了开发工作的需要才会被安装。

检查 CUDA Toolkit 与显卡驱动版本是否匹配，可以在网站 https://docs.nvidia.com/cuda/cuda-toolkit-release-notes/index.html#title-resolved-issues 中进行，如图 3-5 所示。

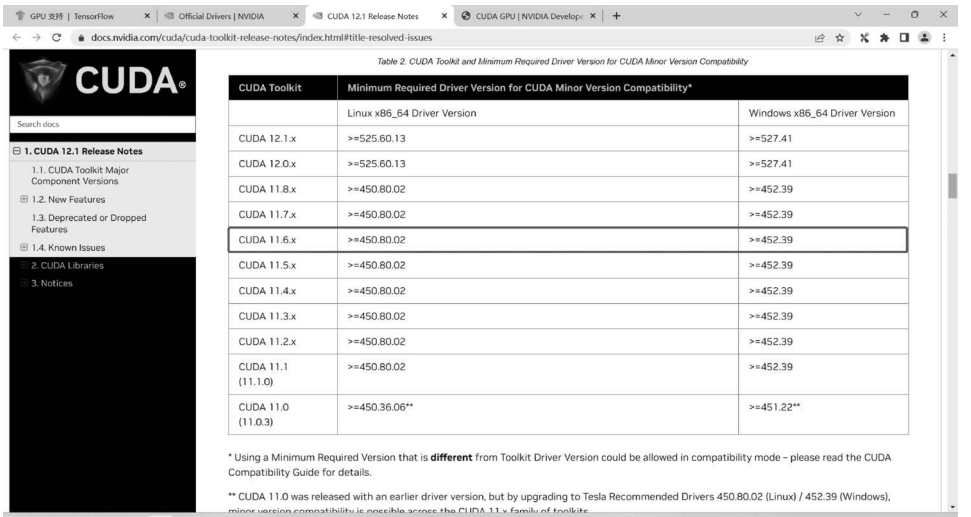


图 3-5　显卡驱动与 CUDA Toolkit 版本对应

在计算机中打开 NVIDIA 控制面板可以查看当前显卡驱动版本，如图 3-6 所示。

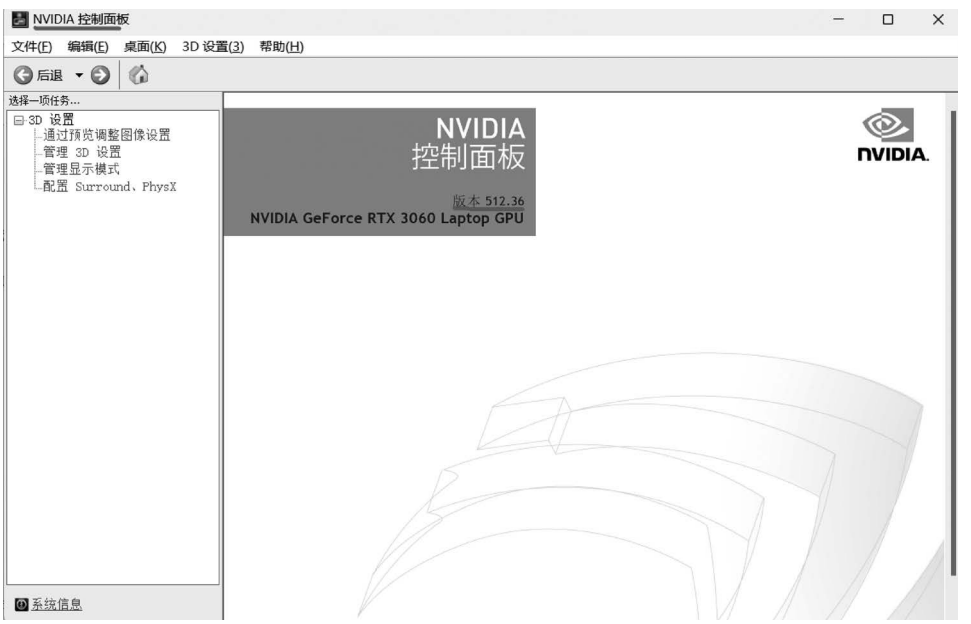


图 3-6　本机显卡驱动版本

如果驱动版本太低，则可以在网站 https://www.nvidia.com/download/index.aspx?lang=en-us 中下载驱动并进行升级，如图 3-7 所示。

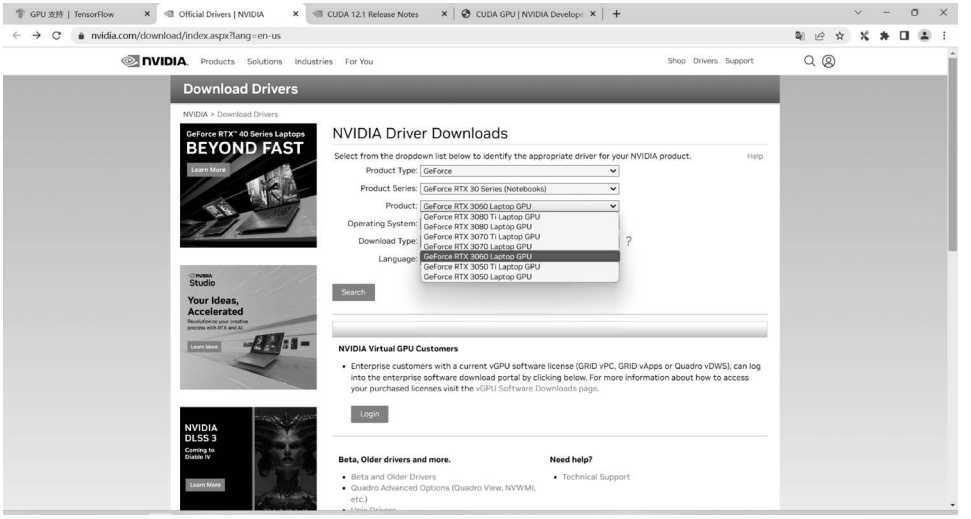


图 3-7　选择显卡型号并下载驱动

（3）CUDA Toolkit、cuDNN 版本应保持一致。cuDNN 是 NVIDIA 针对深度神经网络中的基础操作而设计的基于 GPU 的加速库，其依赖 CUDA Toolkit 工具包。

其依赖关系可以查看网站 https://developer.nvidia.com/rdp/cudnn-archive，如图 3-8 所示。

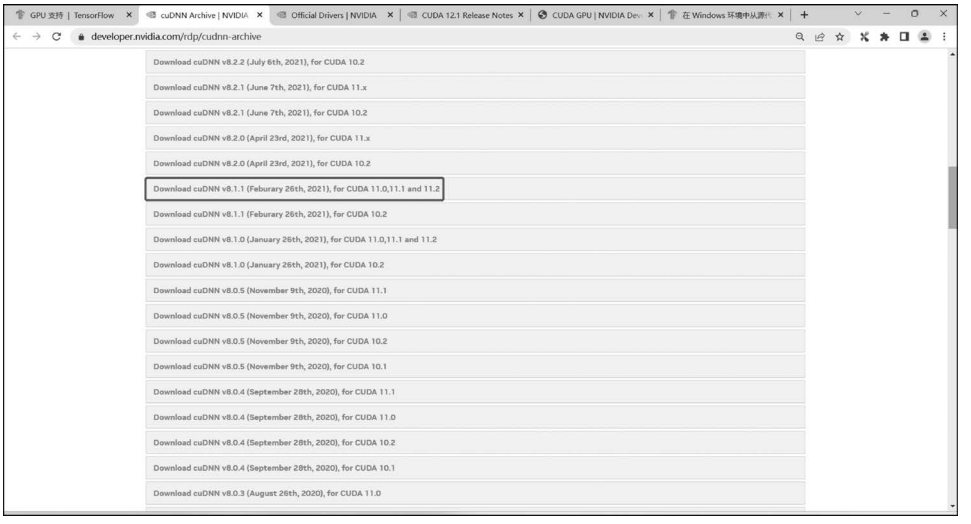
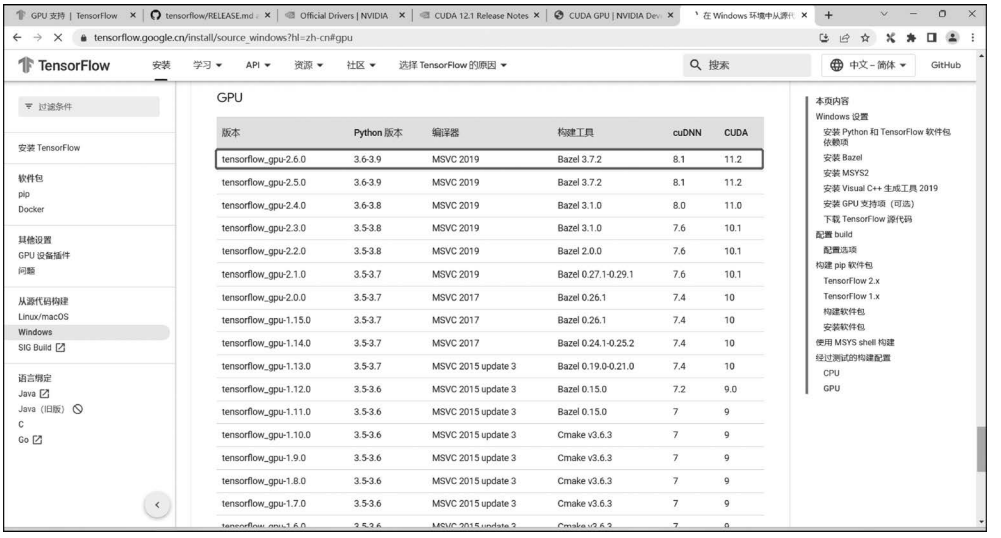


图 3-8　cuDNN 与 CUDA Toolkit 的对应关系

（4）Python、TensorFlow、CUDA Toolkit、cuDNN 版本应保持一致。根据 cuDNN、CUDA 的版本就可以在网站中选择 Python、TensorFlow 的版本进行安装，网站的网址为

https://tensorflow.google.cn/install/source_windows?hl=zh-cn#gpu，如图 3-9 所示。



版本	Python 版本	编译器	构建工具	cuDNN	CUDA
tensorflow_gpu-2.6.0	3.6-3.9	MSVC 2019	Bazel 3.7.2	8.1	11.2
tensorflow_gpu-2.5.0	3.6-3.9	MSVC 2019	Bazel 3.7.2	8.1	11.2
tensorflow_gpu-2.4.0	3.6-3.8	MSVC 2019	Bazel 3.1.0	8.0	11.0
tensorflow_gpu-2.3.0	3.5-3.8	MSVC 2019	Bazel 3.1.0	7.6	10.1
tensorflow_gpu-2.2.0	3.5-3.8	MSVC 2019	Bazel 2.0.0	7.6	10.1
tensorflow_gpu-2.1.0	3.5-3.7	MSVC 2019	Bazel 0.27.1-0.29.1	7.6	10.1
tensorflow_gpu-2.0.0	3.5-3.7	MSVC 2017	Bazel 0.26.1	7.4	10
tensorflow_gpu-1.15.0	3.5-3.7	MSVC 2017	Bazel 0.26.1	7.4	10
tensorflow_gpu-1.14.0	3.5-3.7	MSVC 2017	Bazel 0.24.1-0.25.2	7.4	10
tensorflow_gpu-1.13.0	3.5-3.7	MSVC 2015 update 3	Bazel 0.19.0-0.21.0	7.4	10
tensorflow_gpu-1.12.0	3.5-3.6	MSVC 2015 update 3	Bazel 0.15.0	7.2	9.0
tensorflow_gpu-1.11.0	3.5-3.6	MSVC 2015 update 3	Bazel 0.15.0	7	9
tensorflow_gpu-1.10.0	3.5-3.6	MSVC 2015 update 3	Cmake v3.6.3	7	9
tensorflow_gpu-1.9.0	3.5-3.6	MSVC 2015 update 3	Cmake v3.6.3	7	9
tensorflow_gpu-1.8.0	3.5-3.6	MSVC 2015 update 3	Cmake v3.6.3	7	9
tensorflow_gpu-1.7.0	3.5-3.6	MSVC 2015 update 3	Cmake v3.6.3	7	9

图 3-9　选择 TensorFlow、Python 版本进行安装

手动配置 TensorFlow 的 GPU 版本比较复杂，推荐使用 conda 命令进行安装，conda 会根据本机的 GPU 型号和驱动版本自动安装 CUDA Toolkit 和 cuDNN。进入“1.10 包的管理”节由 conda 命令创建的 StudyDNN 环境后，执行命令 conda install tensorflow-gpu=2.6.0，如图 3-10 所示。

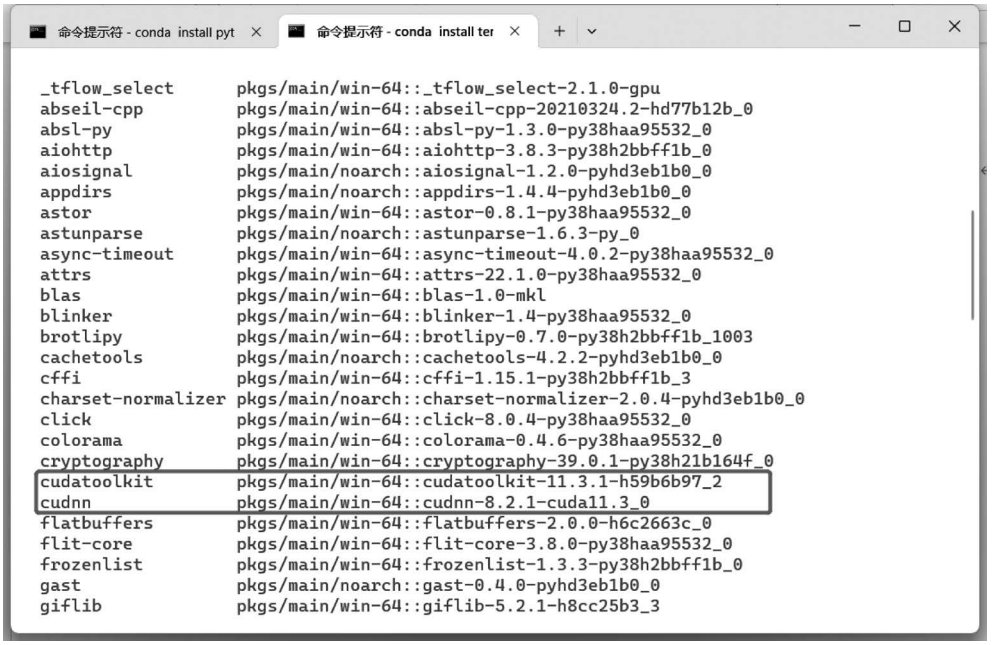


图 3-10　安装 TensorFlow 2.6.0

然后在命令行中执行以下代码测试是否成功地安装了 GPU 版本，如图 3-11 所示。

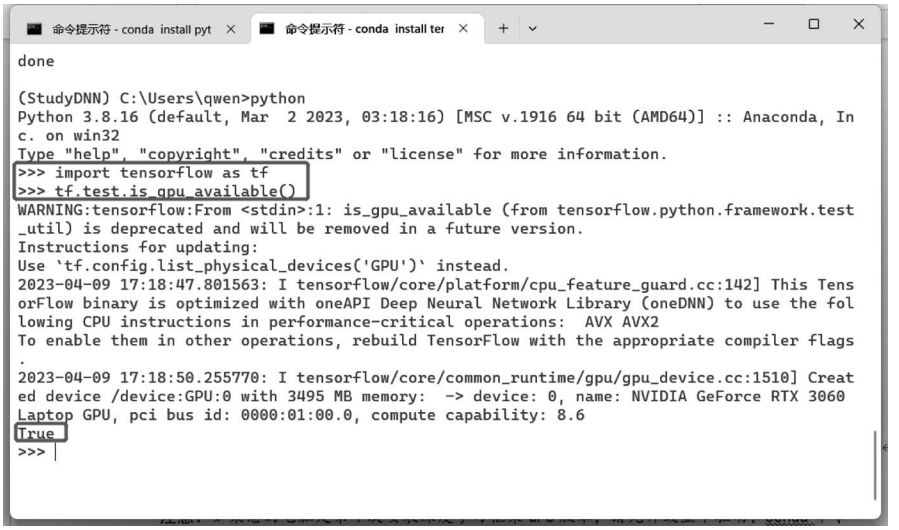


图 3-11　TensorFlow 2.6.0 安装成功 GPU 版本

注意：如果你的计算机是第 1 次安装深度学习框架 GPU 版本，则应先升级显卡驱动；conda 命令可以对相关依赖包进行下载并安装，而 pip 默认下载最新版本，pip 安装的库有可能出现不兼容问题，因此推荐使用 conda 命令安装。

创建一个 PyTorch 的 GPU 运行环境，需要先创建一个 conda 环境，执行的命令如图 3-12 所示。

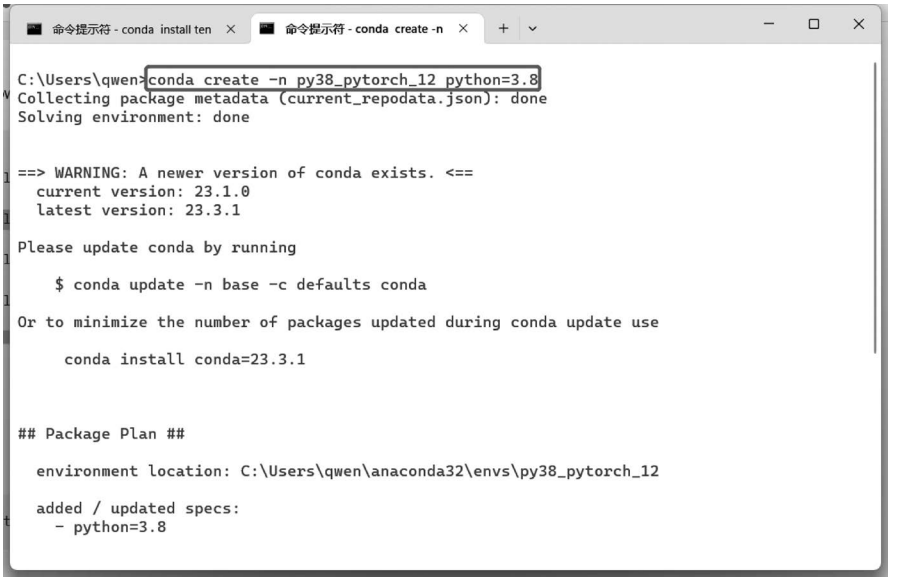


图 3-12　conda 创建环境

然后进入网站 https://pytorch.org/get-started/previous-versions/选择一个版本对应的命令来执行，如图 3-13 所示。

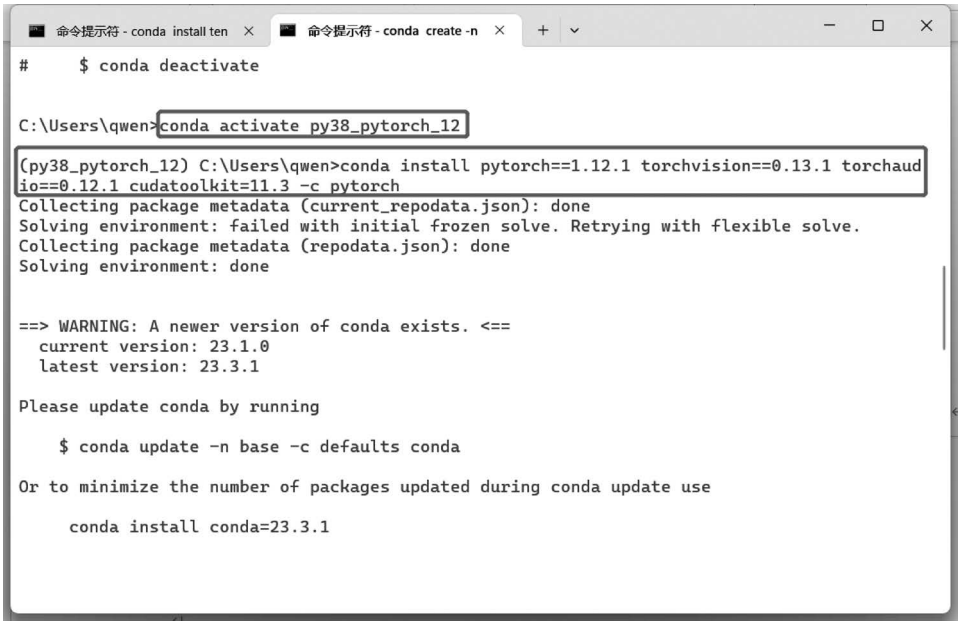


图 3-13　conda 创建 PyTorch 的 GPU 环境

同样 conda 会自动安装 CUDA Toolkit，如图 3-14 所示。

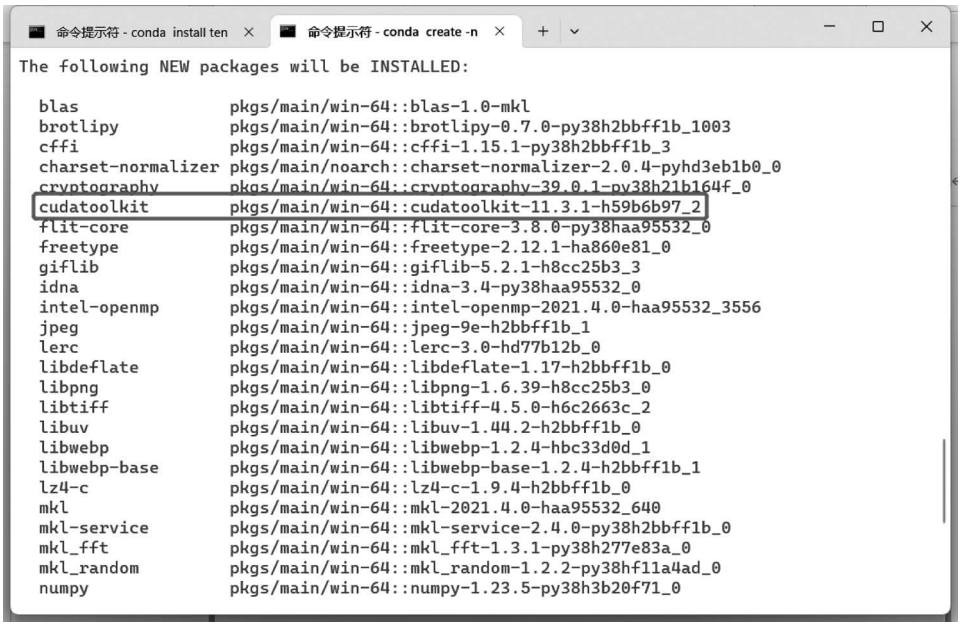


图 3-14　conda 安装 PyTorch 的 CUDA 库

然后在命令行中执行以下代码测试是否成功地安装了 GPU 版本，如图 3-15 所示。

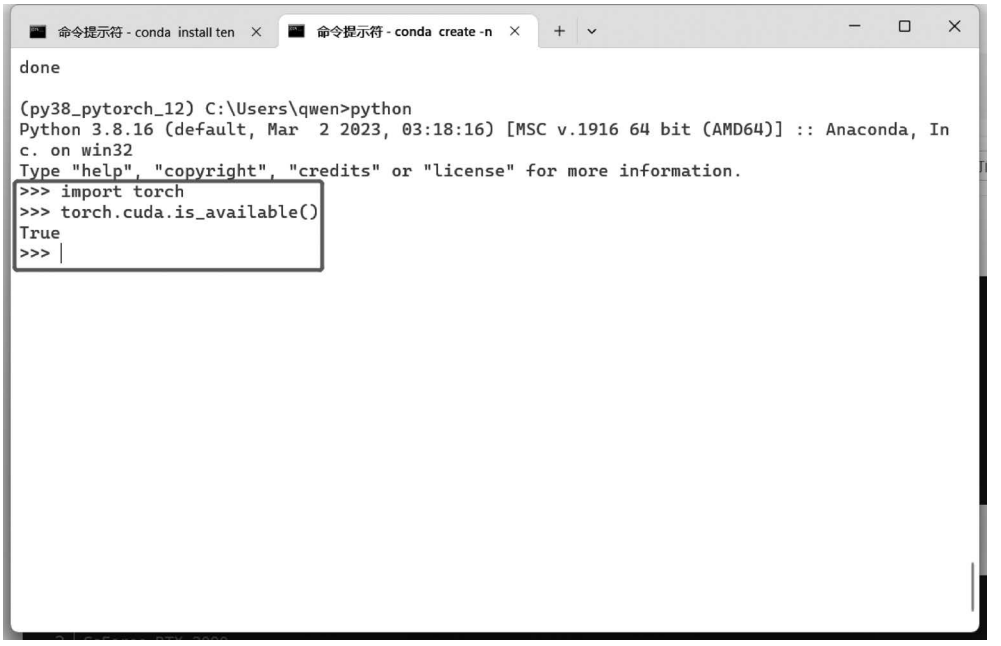


图 3-15 PyTorch 安装成功 GPU 环境

注意：不要将 TensorFlow 和 PyTorch 安装在同一环境中，否则会冲突；笔者当前计算机显卡驱动版本为 512.36，对应 cudatoolkit=11.3，所以只能安装 PyTorch 11.2，读者应根据自己的计算机选择相应版本。

总结

CUDA 环境的安装可使用 conda 命令一键安装。安装的版本需要跟自己的显卡型号相匹配。

练习

在自己的计算机中搭建 TensorFlow 和 PyTorch 运行环境。

3.3 TensorFlow 基础函数

3.3.1 TensorFlow 初始类型

TensorFlow 主要的数据类型有两种，一种是 ResourceVariable 变量类型，其初始后的值可以通过 assign()函数进行修改；另一种是 EagerTensor 类型，其值域定义后不可修改，代码如下：

```
#第 3 章/TensorFlowAPI/基本函数.py
import tensorflow as tf
import numpy as np

#(1) 创建标量
scalar = tf.constant([[1., 2]], dtype=tf.float32)
print('类型:', type(scalar))
#(2) variable 是一个初始的变量值,随着代码的推进会变化
theta = tf.Variable([[1.], [2.]], dtype=tf.float32)
print('类型:', type(theta))
#(3) 将 NumPy 转换为 Tensor
rnd_numpy = np.random.randn(4, 3)
np_tensor = tf.convert_to_tensor(rnd_numpy)
print("将其他对象转换成 Tensor:", type(np_tensor))
#(4) 将张量转换成其他数据类型
type_conversion = tf.cast(theta, dtype=tf.int8)
print("Tensor 数据类型转换: ", type_conversion)

#EagerTensor 对象不能修改,而 ResourceVariable 对象可以修改
theta = theta.assign([[5.], [6.]])
print('ResourceVariable 对象修改后的值:', theta)
#以下语句会报错,因为 EagerTensor 对象不能修改
#EagerTensor' object has no attribute 'assign'
scalar.assign([[11., 22]])
```

运行结果如下：

```
类型: <class 'Tensorflow.python.framework.ops.EagerTensor'>
类型: <class 'Tensorflow.python.ops.resource_variable_ops.ResourceVariable'>

将其他对象转换成 Tensor: <class 'Tensorflow.python.framework.ops.EagerTensor'>
Tensor 数据类型转换: tf.Tensor(
[[1]
[2]], shape=(2, 1), dtype=int8)
ResourceVariable 对象修改后的值: <tf.Variable 'UnreadVariable' shape=(2, 1)
dtype=float32, NumPy=
array([[5.],
       [6.]], dtype=float32)>
Traceback (most recent call last):
  File "D:/DLAI/TensorFlowAPI/基本函数.py", line 24, in <module>
    scalar.assign([[11., 22]])
  File "C:\Users\qwen\anaconda32\envs\py38_tf26\lib\site-packages\TensorFlow\
python\framework\ops.py", line 401, in __getattr__
    self.__getattribute__(name)
AttributeError: 'Tensorflow.python.framework.ops.EagerTensor' object has no
attribute 'assign'
```

tf.Variable()创建的是 ResourceVariable 类型，tf.constant()、tf.convert_to_tensor()、tf.cast()创建的是 EagerTensor 类型，scalar.assign([[11.,22]])意图对 EagerTensor 类型进行修改，所以会报'Tensorflow.python.framework.ops.EagerTensor' object has no attribute

'assign'错误。

tf.constant()用于创建标量；tf.convert_to_tensor()用于将其他数据类型(例如 NDArray)转换为 EagerTensor；tf.cast()用于将默认的 float32 位数据类型转换为 int 类型。

3.3.2 TensorFlow 指定设备

运行 TensorFlow 可以指定 CPU 或者 GPU 运行(GPU 需要显卡支持)，其语句主要用 tf.device(device_name)来指定，一般来说如果安装的是 TensorFlow-gpu，则会默认用 GPU 来执行，如果 GPU 不存在，则会调 CPU 来运行，代码如下：

```
#第 3 章/TensorFlowAPI/指定运行设备.py
import tensorflow as tf
import time

#在此 cpu 作用域语句下面计算执行时间
with tf.device("cpu"):
    t1 = time.time()
    cpu = tf.constant(1)
    print(time.time() - t1)
#在此 gpu 作用域语句下面计算执行时间
with tf.device('gpu'):
    t2 = time.time()
gpu = tf.constant(1)
print(time.time() - t1)
print(cpu.device, gpu.device, sep='\n')
```

运行结果如下：

```
0.0
0.03126859664916992
/job:localhost/replica:0/task:0/device:CPU:0
/job:localhost/replica:0/task:0/device:GPU:0
```

3.3.3 TensorFlow 数学运算

TensorFlow 已封装好一些基本数学运算函数，例如加、减、乘、除、内积、点乘等，其运算过程与 NumPy 中保持一致，代码如下：

```
#第 3 章/TensorFlowAPI/数学运算.py
import tensorflow as tf

data1 = tf.constant([[2., 2.]], dtype=tf.float32)
data2 = tf.constant([[4., 4.]], dtype=tf.float32)
#加法
print(tf.add(data1, data2))
#减法
print(tf.subtract(data1, data2))
#点乘
print(tf.multiply(data1, data2))
```

```
#除法
print(tf.divide(data1, data2))
#内积
print(tf.matmul(data1, tf.transpose(data2)))
#调用 tf.math 下面的函数
print(tf.math.sqrt(data1))
```

运行结果如下：

```
tf.Tensor([[6. 6.]], shape=(1, 2), dtype=float32)
tf.Tensor([[-2. -2.]], shape=(1, 2), dtype=float32)
tf.Tensor([[8. 8.]], shape=(1, 2), dtype=float32)
tf.Tensor([[0.5 0.5]], shape=(1, 2), dtype=float32)
tf.Tensor([[16.]], shape=(1, 1), dtype=float32)
tf.Tensor([[1.4142135 1.4142135]], shape=(1, 2), dtype=float32)
```

tf.math 类下提供了众多常用的数学函数，如图 3-16 所示。

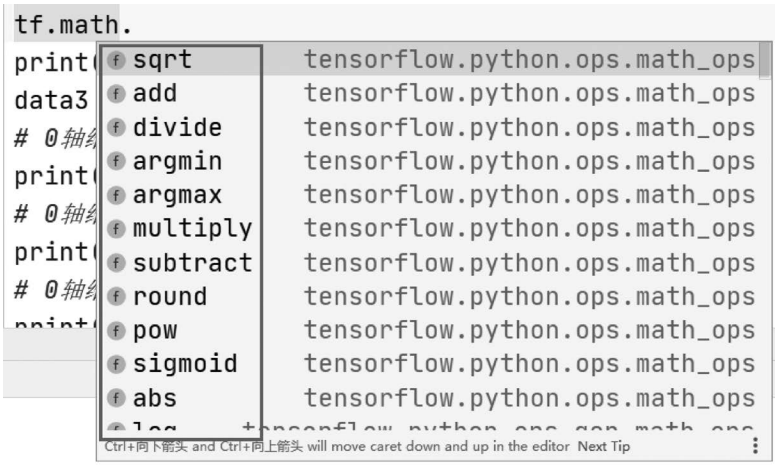


图 3-16　TensorFlow 常见数学函数

根据 axis 轴的位置进行与统计函数相关的操作，代码如下：

```
#第 3 章/TensorFlowAPI/数学运算.py
data3 = tf.convert_to_tensor(
    [[1.1228833, -0.94511896, -1.1491745],
     [0.50430834, -0.1323103, -0.509263]]
)
#1 轴维度最小值
print(tf.reduce_min(data3, axis=1))
#0 轴维度平均值
print(tf.reduce_mean(data3, axis=0))
#0 轴维度求和
print(tf.reduce_sum(data3, axis=0))
#1 轴维度最大
print(tf.reduce_max(data3, axis=1))
```

运行结果如下：

```
tf.Tensor([-1.1491745 -0.509263 ], shape=(2,), dtype=float32)
tf.Tensor([ 0.81359583 -0.53871465 -0.82921875], shape=(3,), dtype=float32)
tf.Tensor([ 1.6271917 -1.0774293 -1.6584375], shape=(3,), dtype=float32)
tf.Tensor([1.1228833  0.50430834], shape=(2,), dtype=float32)
```

代码中计算的维度参照 axis 的位置，由外到内是从 0 轴开始，由于 data3 是二维数据，所以当 axis＝0 时竖着算，当 axis＝1 时算最内层，其计算顺序如图 3-17 所示。

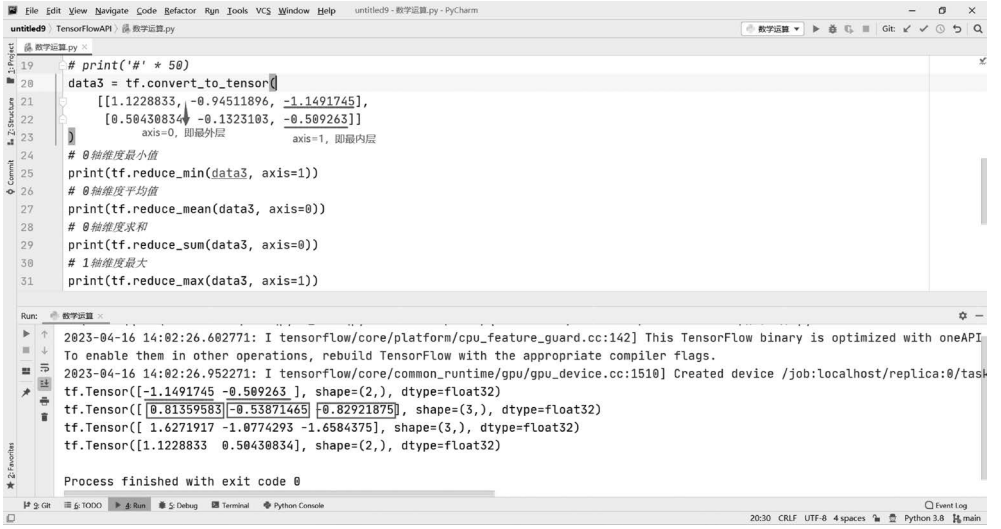


图 3-17　TensorFlow 中 axis 的计算顺序

另一个常用操作就是根据 axis 获取最小值、最大值、排序后的索引位置，代码如下：

```
#第 3 章/TensorFlowAPI/数学运算.py
data3 = tf.convert_to_tensor(
    [[1.1228833, -0.94511896, -1.1491745],
     [0.50430834, -0.1323103, -0.509263]]
)
#最大值的下标
print(tf.argmax(data3, axis=0))
#最小值的下标
print(tf.argmin(data3, axis=0))
#排序的下标
print(tf.argsort(data3, axis=1))
```

运行结果如下：

```
tf.Tensor([0 1 1], shape=(3,), dtype=int64)
tf.Tensor([1 0 0], shape=(3,), dtype=int64)
tf.Tensor(
[[2 1 0]
 [2 1 0]], shape=(2, 3), dtype=int32)
```

tf.argmax 得到的是索引位置，如果 axis 不同，则其索引值也不同，其计算过程如图 3-18 所示。

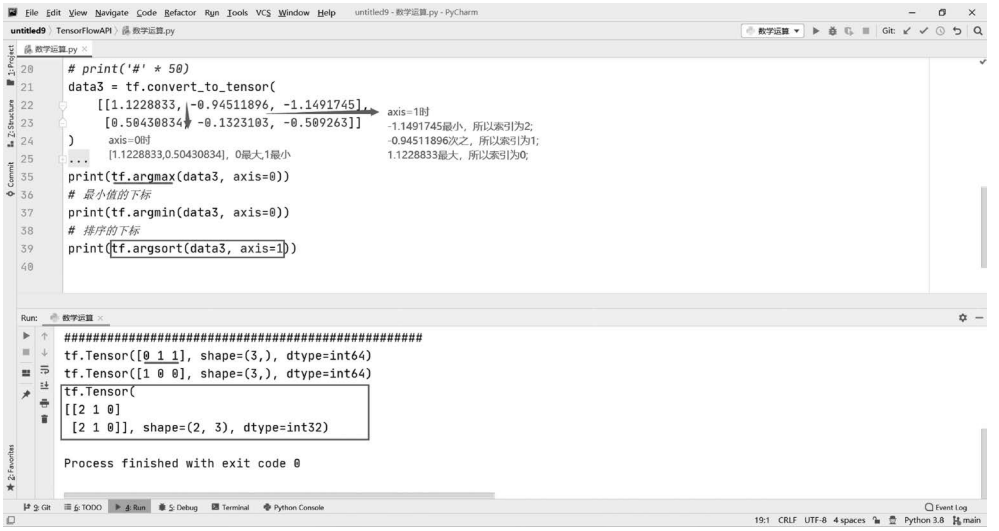


图 3-18　TensorFlow 中根据 axis 得到索引值

3.3.4　TensorFlow 维度变化

维度的操作主要包括维度转换、升维、降维、维度交换等，主要使用 tf.reshape()、tf.expand_dims()、tf.squeeze()、tf.transpose()等函数，无论张量的维度如何变化，其 size（总数）应该保持不变，代码如下：

```
#第 3 章/TensorFlowAPI/维度变换.py
import tensorflow as tf

#随机生成 4 维数据
rnd_shape4 = tf.random.normal([4, 32, 32, 3])
print("获取维度:", rnd_shape4.shape)
#数据维度升维、降维
#在 0 轴升一维，变成(1,4, 32, 32, 3)
shape5 = tf.expand_dims(rnd_shape4, axis=0)
print('升维:', shape5.shape)
#在 0 轴降一维，变成(4, 32, 32, 3)
shape5_to4 = tf.squeeze(shape5, axis=0)
print('降维:', shape5_to4.shape)
#在 1 轴升一维，变成(4, 1, 32, 32, 3)
shape6 = tf.expand_dims(rnd_shape4, axis=1)
print('升维:', shape6.shape)
#在 0 轴降一维，变成(4, 32, 32, 3)
shape6_1 = tf.squeeze(shape6, axis=1)
print('降维:', shape6_1.shape)
#改变维度
```

```
to_shape = tf.reshape(shape6_1, [1, 1, 4, 32, 32, 3])
print("改变维度:", to_shape.shape)

transpose_num = tf.random.normal([32, 28, 3])
#按指定的顺序转置,得到(28, 32, 3)
print("按指定的顺序交换: ", tf.transpose(transpose_num, perm=[1, 0, 2]).shape)
#以下语句会报错,因为 tf.squeeze 只能对 axis=0 位置是 1 的数组进行操作
error_shape = tf.squeeze(rnd_shape4, axis=0)
```

运行结果如下：

```
获取维度: (4, 32, 32, 3)
升维: (1, 4, 32, 32, 3)
降维: (4, 32, 32, 3)
升维: (4, 1, 32, 32, 3)
降维: (4, 32, 32, 3)
改变维度: (1, 1, 4, 32, 32, 3)
按指定的顺序交换: (28, 32, 3)
Tensorflow. python. framework. errors _ impl. InvalidArgumentError: Can not
squeeze dim[0], expected a dimension of 1, got 4 [Op:Squeeze]
```

属性 shape 为多维数组的形状，在进行张量操作时，需要特别注意张量的 shape 变化。代码中 tf. random. normal([4,32,32,3])初始了一个 shape 为 4×32×32×3 的张量，tf. expand_dims(rnd_shape4,axis=0)即在最外层插入 1 个维度就变成 shape 为 1×4×32×32×3 的张量，其 size 总数不变；tf. squeeze(shape5,axis=0)将最外层的维度去除，就变成了 4×32×32×3 的张量，其 size 总数也不变。tf. expand_dims(rnd_shape4,axis=1)即在 1 轴插入 1 个维度，所以变成了 4×1×32×32×3，总结规律可以发现，axis=?就在该位置插入 1 个维度，而 tf. squeeze 就减少 1 个维度。

tf. reshape()可以实现任意维度变化的操作，可以代替 tf. expand_dims()、tf. squeeze()实现维度的变化。tf. transpose()按指定的 axis 进行交换，交换后 size 保持不变，但是 shape 发生了变化。

注意：tf. squeeze 只能对 axis 所在位置维度为 1 的数组进行降维，所以 error_shape 变量时会报 InvalidArgumentError 错误。

3.3.5 TensorFlow 切片取值

TensorFlow 支持如 NumPy 一样的下标取值方法，代码如下：

```
#第 3 章/TensorFlowAPI/下标、切片操作.py
#初始张量
a = tf.ones([1, 5, 5, 3])
#先取 axis=0 的内容,得到 5*5*3;从 5*5*3 中再得到 axis=0 下标为 2 的内容,即 5*3;从 5*3
#中再得到 axis=0 下标为 2 的内容,即 shape=(3,)
print(a[0][2][2].shape)
```

运行结果如下：

```
(3,)
```

代码 a[0]，即从 1 个 5×5×3 中得到第 0 个内容，所以 shape 为 5×5×3；a[0][2]即从 5 个 5×3 中得到下标为 2 的内容，即 5×3；a[0][2][2]即从 5 个(3,)中得到下标为 2 的内容，即 shape=(3,)。

上面的代码 a[0][2][2]可以被修改成 a[0,2,2]的表达方式，代码如下：

```
#第 3 章/TensorFlowAPI/下标、切片操作.py
import tensorflow as tf

#初始张量
a = tf.ones([1, 5, 5, 3])
print(a[0].shape)                    #[5,5,3] 由外向内
print(a[0, 2].shape)                 #[5,3] 第 0 轴，第 1 轴里的第 3 个
print(a[0, 2, 2].shape)              #[3,]
```

运行结果如下：

```
(5, 5, 3)
(5, 3)
(3,)
```

根据结果可知，a[0][2][2]和 a[0,2,2]的内容是一样的。从 a[0,2,2]的表达式中可知每个 axis 可以用逗号隔开，有多少个逗号就有多少个维度。

多维度的切片与 NumPy 保持一致，代码如下：

```
#第 3 章/TensorFlowAPI/下标、切片操作.py
import tensorflow as tf

#初始张量
a = tf.ones([1, 5, 5, 3])
print("多维度的切片:", a[0, :, :, :].shape)                    #得到 5*5*3
print("多维度的切片:", a[:, :, :, 2].shape)                    #得到 1*5*5
print("多维度间隔取值:", a[:, 0:4:2, 0:4:2, :].shape)          #得到 1*2*2*3
print("...省略:", a[..., 2].shape)                             #得到 1*5*5
```

运行结果如下：

```
多维度的切片: (5, 5, 3)
多维度的切片: (1, 5, 5)
多维度间隔取值: (1, 2, 2, 3)
...省略: (1, 5, 5)
```

在每个 axis 中都可以使用切片的语法，a[0,:,:,:,:]表示 axis=0 中下标为 0 的所有的内容；a[:,:,:,:,2]表示 axis=3 中下标为 2 的内容，前面 axis 的内容保留，所以其 shape 为 1×5×5×5；a[:,0:4:2,0:4:2,:]表示 axis=0 中所有的内容，axis=1 每 2 取 1 个，所以 axis=1 时取下标 0、2 的内容，axis=2 也一样，axis=3 中所有的内容，所以其 shape 为 1×2×2×3；

a[...,0]中的...表示前面 axis=0,1,2 的内容全部保留,与 a[:,:,:,2]相同。

TensorFlow 中下标或者切片只能取值而不能进行修改,但是 NumPy 都可以,代码如下:

```
#第 3 章/TensorFlowAPI/下标、切片操作.py
import tensorflow as tf
import numpy as np

NumPy_a = np.ones([1, 5, 5, 3])
tensor_a = tf.ones([1, 5, 5, 3])
#NumPy 修改值
NumPy_a[..., 0] = 0
#TensorFlow 中不支持下标修改
tensor_a[..., 0] = 0
```

运行结果如下:

```
tensor_a[..., 0] = 0
TypeError: 'Tensorflow.python.framework.ops.EagerTensor' object does not
support item assignment
```

3.3.6 TensorFlow 中 gather 取值

函数 gather(params,indices,axis=None)提供了更灵活、强大的取值方法。params 要求传入张量,indices 可以传入多种形式的索引号的值,axis 为指定的轴位置,代码如下:

```
#第 3 章/TensorFlowAPI/gather 取值.py
import tensorflow as tf
params = tf.constant([10., 11., 12., 13., 14., 15.])
#取 params 索引下标为[2, 0, 2, 5]的内容
print(tf.gather(params, indices=[2, 0, 2, 5]))
#先取 params 索引下标[2, 0]的值,组成 1 个维度,然后取索引下标[2, 5]的值,再组成 1 个维度
#然后将两个维度合并成一个新的维度,变成 2*2 的张量
print(tf.gather(params, [[2, 0], [2, 5]]))
```

运行结果如下:

```
tf.Tensor([12. 10. 12. 15.], shape=(4,), dtype=float32)
tf.Tensor(
[[12. 10.]
 [12. 15.]], shape=(2, 2), dtype=float32)
```

代码中 params 只有一个维度,所以 axis=0。将 params 的内容变为 4×3,其 gather 取值的代码如下:

```
#第 3 章/TensorFlowAPI/gather 取值.py
import tensorflow as tf

params = tf.constant([[0, 1.0, 2.0],
                      [10.0, 11.0, 12.0],
                      [20.0, 21.0, 22.0],
                      [30.0, 31.0, 32.0]])
```

```
#没有指定 axis,默认为 0 轴,即下标索引为 3 和 1 的内容
print(tf.gather(params, indices=[3, 1]))
#axis=1,则取 axis=1 中下标索引为 2 和 1 的内容
print(tf.gather(params, indices=[2, 1], axis=1))
```

运行结果如下：

```
tf.Tensor(
[[30. 31. 32.]
 [10. 11. 12.]], shape=(2, 3), dtype=float32)
tf.Tensor(
[[ 2.  1.]
 [12. 11.]
 [22. 21.]
 [32. 31.]], shape=(4, 2), dtype=float32)
```

代码中 axis＝1 时会将索引下标为[2,1]的内容全部取走,所以得到的 shape＝(4,2),其取值过程如图 3-19 所示。

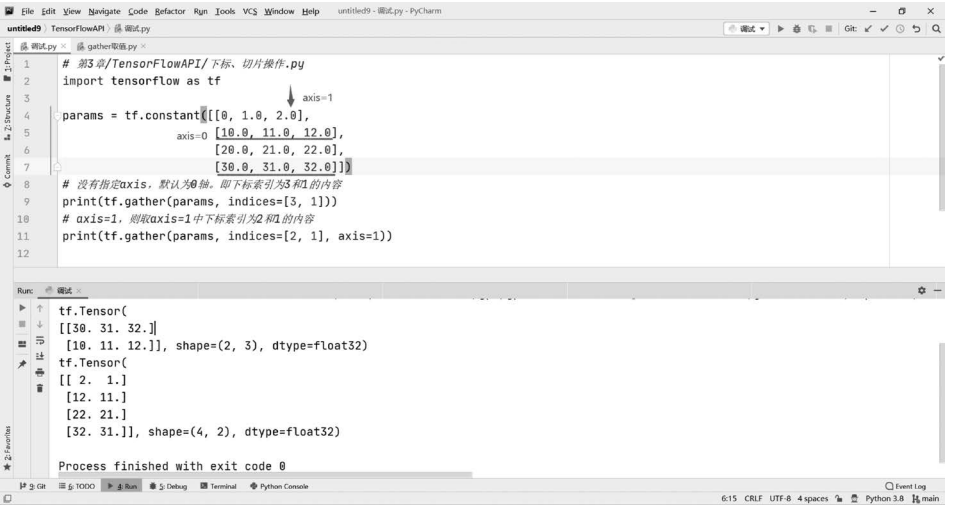


图 3-19　TensorFlow 中 gather 取值

同样 indices 可以为多维张量,代码如下：

```
#第 3 章/TensorFlowAPI/gather 取值.py
import tensorflow as tf

params = tf.constant([[0, 1.0, 2.0],
                      [10.0, 11.0, 12.0],
                      [20.0, 21.0, 22.0],
                      [30.0, 31.0, 32.0]])
#indices 是一个多维张量
indices = tf.constant([
    [2, 4],
    [0, 4]])
```

```
#当 indices 中的 4 超过 params 的索引时会自动补 0
#因为 axis=0,所以 0 轴补 3 个 0
print(tf.gather(params, indices, axis=0))
print('#'*20)
#因为 axis=1,所以 1 轴补 1 个 0
print(tf.gather(params, indices, axis=1))
```

运行结果如下：

```
tf.Tensor(
[[[20. 21. 22.]
  [ 0.  0.  0.]]

 [[ 0.  1.  2.]
  [ 0.  0.  0.]]], shape=(2, 2, 3), dtype=float32)
####################
tf.Tensor(
[[[ 2.  0.]
  [ 0.  0.]]

 [[12.  0.]
  [10.  0.]]

 [[22.  0.]
  [20.  0.]]

 [[32.  0.]
  [30.  0.]]], shape=(4, 2, 2), dtype=float32)
```

多维张量 gather 取值与此类似，代码如下：

```
#第 3 章/TensorFlowAPI/gather 取值.py
import tensorflow as tf

params = tf.random.normal([4, 35, 8])
#axis=0 就是第 1 个维度的变化
#[7, 9, 16]的下标不存在,会补 0,所以是[5,35,8]
result0 = tf.gather(params, axis=0, indices=[2, 3, 7, 9, 16])
#axis=1 就是第 2 个维度的变化,所以是[4,5,8]
result1 = tf.gather(params, axis=1, indices=[2, 3, 7, 9, 16])
#axis=2 就是最里面的维度,所以是[4,35,3]
result2 = tf.gather(params, axis=2, indices=[2, 3, 7])
#axis=2 就是第 3 个维度的变化,所以[4,35]会保留,另外[[2, 3, 7], [0, 1, 2]]即对[8]取了
#两次,所以应该变成[2,3],结合后变成[4,35,2,3]
result3 = tf.gather(params, axis=2, indices=[[2, 3, 7], [0, 1, 2]])
print(result0.shape, result1.shape, result2.shape, result3.shape)
```

运行结果如下：

```
(5, 35, 8) (4, 5, 8) (4, 35, 3) (4, 35, 2, 3)
```

代码 tf.gather(params，axis=0，indices=[2，3，7，9，16])中 indices 的数量超过了 4，TensorFlow 会自动补 0，如图 3-20 所示。

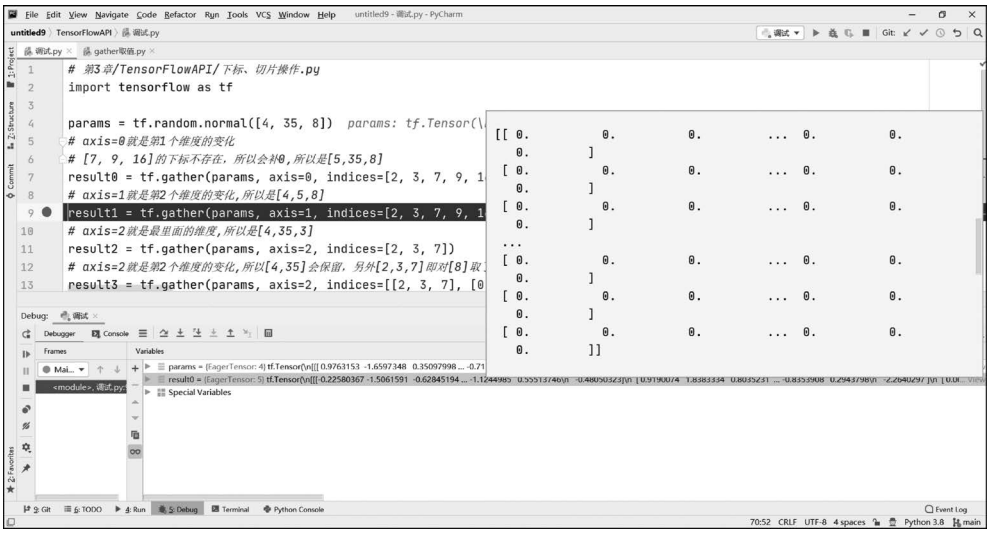


图 3-20　TensorFlow 中 gather 补 0

3.3.7　TensorFlow 中布尔取值

TensorFlow 提供了两种方式的布尔取值，一种是类似 NumPy 的风格，代码如下：

```
#第 3 章/TensorFlowAPI/布尔取值.py
import tensorflow as tf
import numpy as np

data = np.array([[1, 2], [3, 4], [5, 6]])
#获取 data 中>2 的数
mask = data > 2
print('mask:', mask)
print("输出满足条件的数：", data[mask], data[data > 2])
```

运行结果如下：

```
mask: [[False False]
 [ True  True]
 [ True  True]]
输出满足条件的数：[3 4 5 6] [3 4 5 6]
```

与 NumPy 一样 data[data>2]中的条件只能为简写表达式，其作用是将 data 中为 True 的内容取出来。

另一种方式是使用 tf.boolean_mask()函数来取值，代码如下：

```
#第 3 章/TensorFlowAPI/布尔取值.py
import tensorflow as tf
import numpy as np

tensor = tf.random.normal([4, 28, 28, 3])
```

```
mask = np.array([True, True, False, False])
# 由于 4 维中 axis=0,1 为 True,所以输出是(2,28,28,3)
print('多维度布尔取值：', tf.boolean_mask(tensor, mask).shape)

old = tf.random.normal([2, 3, 4])
print("原值：", old.shape)
#因为 axis=0 时,其 shape 为 2,所以 mask 需要两个数组的内容,正好对应
#当 axis=0 时,0 的下标为 True,所以这里是 2 *4
result1 = tf.boolean_mask(old, mask=[[True, False, False], [True, False, False]],
axis=0)
#当 axis=1 时,其 shape 为 3,所以需要有 3 个值
#当 axis=1 时,0 和 2 的下标都为 True,所以这里是 2*2*4
result2 = tf.boolean_mask(old, mask=[True, False, True], axis=1)
#当 axis=2 时,其 shape 为 4,所以需要有 4 个值
#当 axis=2 时,下标 0、2 都为 True,所以为 2,则加外层,所以是 2*3*2
result3 = tf.boolean_mask(old, mask=[True, False, True, False], axis=2)
print('布尔多维取值 axis=0：', result1.shape)
print('布尔多维取值 axis=1：', result2.shape)
print('布尔多维取值 axis=2：', result3.shape)
```

运行结果如下：

```
多维度布尔取值：(2, 28, 28, 3)
原值：(2, 3, 4)
布尔多维取值 axis=0：(2, 4)
布尔多维取值 axis=1：(2, 2, 4)
布尔多维取值 axis=2：(2, 3, 2)
```

代码中布尔值列表的长度需要跟 axis 中的 shape 数相等。例如 result1 中的 mask＝[[True，False，False]，[True，False，False]]跟 old 张量 shape＝[2，3，4]中的 2 相同。

3.3.8 TensorFlow 张量合并

张量合并，即将多个张量合并成一个新张量，但是除指定的 axis 的 shape 可以不相等，其他 shape 必须相等，代码如下：

```
#第 3 章/TensorFlowAPI/张量合并.py
import tensorflow as tf

#axis 所在 shape 可以不相等,但是其他 shape 必须相等
#从 0 外围的维度合并 Tensor,就是[6,35,8]
print(tf.concat([tf.ones([4, 35, 8]), tf.ones([2, 35, 8])], axis=0).shape)
#从 1 这个维度合并 Tensor,就是[4,35,8]
print(tf.concat([tf.ones([4, 32, 8]), tf.ones([4, 3, 8])], axis=1).shape)
#不能合并的情况,axis=1 这个维度没有对齐
try:
    df1 = tf.ones([4, 35, 8])
    df2 = tf.ones([3, 33, 8])
    print(tf.concat([df1, df2], axis=0).shape)
except Exception as e:
    print(e)
```

运行结果如下：

```
(6, 35, 8)
(4, 35, 8)
ConcatOp : Dimensions of inputs should match: shape[0] = [4,35,8] vs. shape[1] =
[3,33,8] [Op:ConcatV2] name: concat
```

3.3.9　TensorFlow 网格坐标

生成网格点坐标张量的方法 tf.meshgrid()，代码如下：

```
#第 3 章/TensorFlowAPI/meshgrid 生成网格.py
import tensorflow as tf
from matplotlib import pyplot as plt
x = tf.cast([[0, 1, 2], [0, 1, 2]], dtype=tf.float32)
y = tf.cast([[0, 0, 0], [1, 1, 1]], dtype=tf.float32)
#生成网络坐标点,即[0,0],[1,0],[2,0],[0,1],[1,1],[2,1]共 6 个坐标
points_x, points_y = tf.meshgrid(x, y)
#将坐标点画到图上
plt.plot(points_x, points_y,
         color='red',
         markersize=15,
         marker='.',
         linestyle='')
#将坐标值画到图上
for i in range(x.shape[0]):
   for j in range(y.shape[1]):
      xl = x[i, j].NumPy()
      yl = y[i, j].NumPy()
      plt.text(xl, yl + 0.01, f"{xl},{yl}")
plt.grid(True)
plt.xlabel('X')
plt.ylabel('Y')
plt.title("tf.meshgrid 生成网格坐标张量")
plt.show()
```

运行结果如图 3-21 所示。

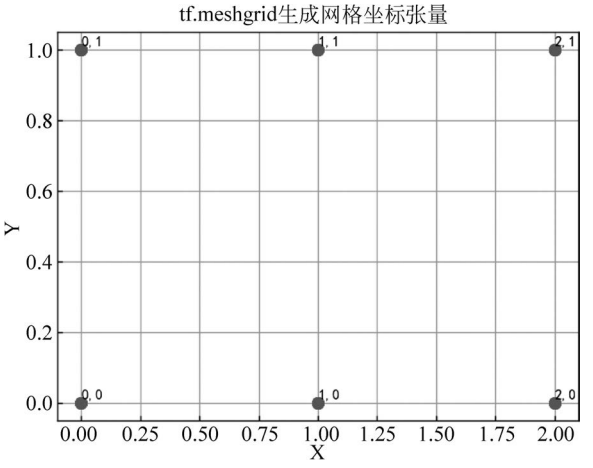


图 3-21　TensorFlow 中生成网格坐标

3.3.10 TensorFlow 自动求梯度

TensorFlow 提供了自动求梯度的功能，代码如下：

```
#第 3 章/TensorFlowAPI/自动求导.py
import tensorflow as tf

def g(z):
   return 1 / (1 + tf.exp(-z))

#输入 x 的值
ga = tf.convert_to_tensor([[2.], [3.], [1.]])
#初始权重值
init_w = {
    'w1': tf.Variable([[0.03, 0.02, 0.01], [0.013, 0.012, 0.001]]),
    'w2': tf.Variable([[0.03, 0.02, 0.01], [0.013, 0.012, 0.001]])
}
y = tf.convert_to_tensor([[1.], [0.]])

with tf.GradientTape() as tape:
    #存储每次反向传播的梯度
    lw = []
    lg = []
    #进行两次前向传播
    for i in range(1, 3):
        #获取权重
        w = init_w['w%s' % i]
        #线性计算
        z = w @ga
        #非线性计算
        out = g(z)
        #增加偏置项
        ga = tf.concat([out, tf.cast([[1.0]], dtype=tf.float32)], axis=0)
        lw.append(w)
        lg.append(out)
    #损失函数
    d = out - y
    loss = tf.reduce_sum(0.5 *tf.square(d))
    #自动求梯度
    gradients = tape.gradient(loss, lw)
    print('误差：', d)
    #print('激活：', lg)
    print('梯度：', gradients)
    #梯度下降
    for layer in range(len(gradients)):
       gradients[layer] = gradients[layer] - 0.1 *gradients[layer]
    #更新权重
    init_w['w1'].assign(gradients[-1])
    init_w['w2'].assign(gradients[0])
```

运行结果如下：

```
误差: tf.Tensor(
[[-0.4909289]
 [ 0.5035277]], shape=(2, 1), dtype=float32)
梯度: [<tf.Tensor: shape=(2, 3), dtype=float32, NumPy=
array([[-0.00101757, -0.00152636, -0.00050879],
       [-0.00047112, -0.00070667, -0.00023556]], dtype=float32)>, <tf.Tensor:
shape=(2, 3), dtype=float32, NumPy=
array([[-0.06529391, -0.06325722, -0.12268066],
       [ 0.0669831 ,  0.06489372,  0.12585449]], dtype=float32)>]
```

自动求梯度只需在 with tf.GradientTape()as tape 作用域下，并使用 tape.gradient(loss,lw)就可以根据 loss 求梯度。相对于“2.6.2 反向传播算法”节中实现的梯度计算，使用自动求梯度功能非常方便。

总结

TensorFlow 的基本 API 主要有维度变换、切片取值、自动求梯度等功能，其中自动求梯度功能对于网络的学习提供了极大的便利。

练习

调试本节所有代码，观察多维数组状态下张量的操作功能；本节相关 API 在 TensorFlow 模型搭建中有重要作用。

3.4　TensorFlow 中的 Keras 模型搭建

3.4.1　tf.keras 简介

Keras 是一个广泛受欢迎的神经网络框架，用 Python 编写而成，能够在 TensorFlow、PyTorch 中运行，其特点是支持快速神经网络模型搭建，同时网络模型具有很强的层次性，可以在 CPU 和 GPU 上运行。

TensorFlow 2.x 版本中的 Keras 主要在 tensorflow.keras 模块下，包含 Models、Layers、Optimizers 等子模块，每个子模块又封装了子方法，如图 3-22 所示。

使用 tensorflow.keras 可以方便、快速地构建神经网络模型，是深度学习代码实战中的重要组成部分。

3.4.2　基于 tf.keras.Sequential 模型搭建

Sequential 的参数是一个列表，列表里面存放着 layers 层下的各个算子，将各个算子按线型结构组装起来，模型就搭建完毕了。这个过程可以形象地类比成美食“串串”，Sequential 就是那一根“竹签”，将各种“美食”(算子)串联起来，就可以“涮串串”了，代码如下：

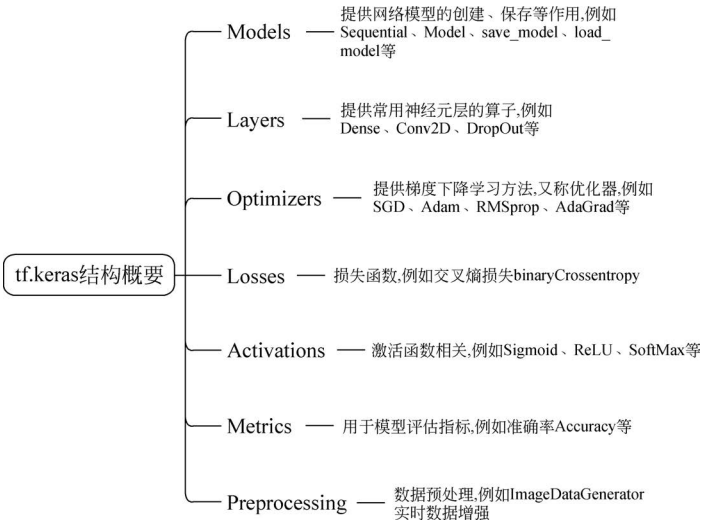


图 3-22　tf. keras 结构概要图

```
#第 3 章/TensorFlowAPI/基于 tf.keras.Sequential 模型搭建.py
import tensorflow as tf
from tensorflow.keras import Sequential, layers, Input

#输入数据的 shape
input_shape = (28, 28, 1)
model = Sequential(
    [
        Input(shape=input_shape),                  #输入数据的维度需要指定
        layers.Flatten(),                          #将多维数据打平成一维数据
        layers.Dense(784, activation='sigmoid'),   #784 个神经元
        layers.Dense(128, activation='sigmoid'),
        layers.Dense(10, activation='softmax')     #10 个类别
    ]
)
#描述模型的结构
model.summary()
```

运行结果如下：

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
dense (Dense)                (None, 784)               615440
_________________________________________________________________
dense_1 (Dense)              (None, 128)               100480
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1290
=================================================================
```

```
Total params: 717,210
Trainable params: 717,210
Non-trainable params: 0
_________________________________________________________________
```

Sequential 提供了网络模型组装方法，layers 模块下提供了各个算子（例如 Dense 类）设置神经元的个数（Dense 又称全连接层），其主要参数如下：

```
def __init__(self,
             units,                                    #神经元的个数
             activation=None,                          #激活函数
             use_bias=True,                            #是否支持偏置项，即 wx+b 中的 b
             kernel_initializer='glorot_uniform',
             **kwargs):
```

model.summary()会将各个网络层的 shape 及 param 参数自动计算出来。Input(shape=input_shape)用来表示输入数据的 shape。

在 model.summary()处断点，查看 model 对象的构成会发现 input、output 分别用于描述网络层的输入与输出，layers 属性用来描述神经网络层中的结构，trainable_variables、trainable_weights 属性会自动化初始神经网络权重中的 θ 值，如图 3-23 所示。

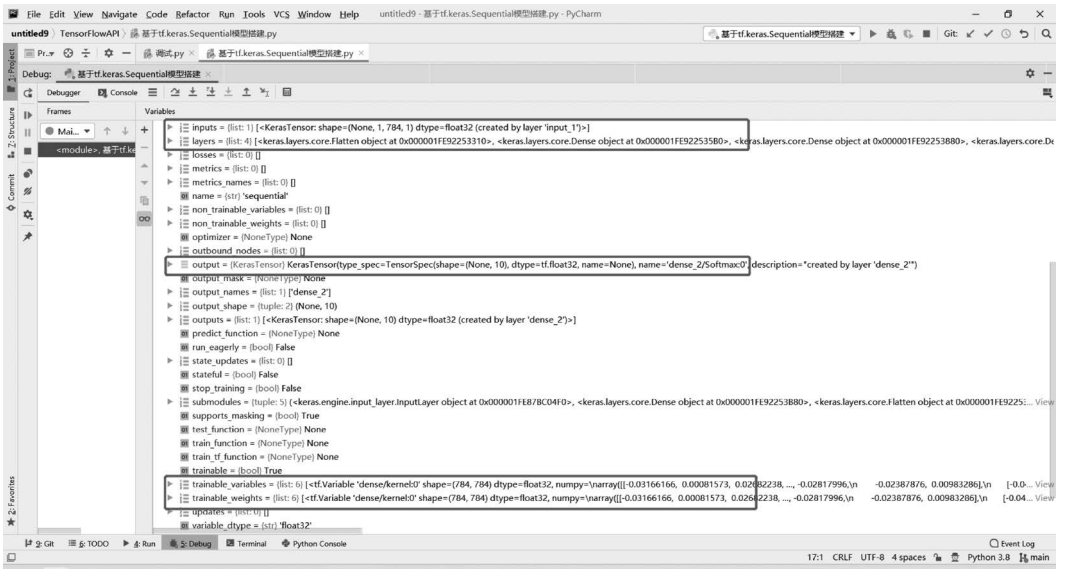


图 3-23　model 对象结构

3.4.3　继承 tf.keras.Model 类模型搭建

创建神经网络模型也可以通过自定义类来继承 tf.keras.Model 类，然后在__init__()构造方法中初始各个网络层，并在 call()构造函数中实现前向传播的逻辑，代码如下：

```
#第 3 章/TensorFlowAPI/继承 tf.keras.Modle 类模型搭建.py
import tensorflow as tf
from tensorflow.keras import layers


class MyModle(tf.keras.Model):
    def __init__(self):
        #继承 Model 类中的属性
        super(MyModel, self).__init__()
        #全连接
        self.flat = layers.Flatten()
        self.dense1 = layers.Dense(784, activation='sigmoid')
        self.dense2 = layers.Dense(128, activation='sigmoid')
        #输出
        self.out = layers.Dense(10, activation='softmax')

    #call()为类的实例方法,可以使类的实例对象成为可调用对象
    def call(self, inputs):
        #在 call()中实现前向传播
        x = self.flat(inputs)
        x = self.dense1(x)
        x = self.dense2(x)
        #返回预测结果
        return self.out(x)

    def get_config(self):
        #重载 get_config 方法
        base_config = super(MyModel, self).get_config()
        return dict(list(base_config.items()))


if __name__ == "__main__":
    model = MyModel()
    #继承 Model 类,需要通过 build 编译一下
    model.build(input_shape=(28, 28, 1))
    model.summary()
```

继承 Model 的方法,需要调用 model. build(input_shape)指明输入的 shape。

3.4.4 函数式模型搭建

另一种方法是直接描述输入,然后将返回值再传给下一个算子的输入,直到最后一个输出,然后将输入与输出放入 Model 类,代码如下:

```
#第 3 章/TensorFlowAPI/函数式模型搭建.py
from tensorflow.keras import Input, layers, Model
from tensorflow.keras.activations import sigmoid, softmax

#描述输入
inputs = Input(shape=(28, 28, 1))
#描述中间过程
```

```
x = layers.Flatten()(inputs)
x = layers.Dense(784, activation=sigmoid)(x)
x = layers.Dense(128, activation=sigmoid)(x)
outputs = layers.Dense(10, activation=softmax)(x)
#把输入和输出装载在 Model 这个类里面
#inputs 和 outpus 可以为列表,用来表示多输入或者多输出
model = Model(inputs=inputs, outputs=outputs)
model.summary()
```

激活函数不仅可以传字符串,也可以使用 tensorflow. keras. activations 模块下定义好的方法。只要描述了 inputs 和 outputs 模型 model 便会自动地找到相关层。

总结

TensorFlow 搭建模型可以使用 Sequential,也可以继承 Model 类,同时还支持函数式搭建。

练习

分别使用 3 种搭建模型的方式完成拥有 3 个隐藏层的模型搭建,并调试代码以观察每行变量中对象的变化。

3.5　TensorFlow 中模型的训练方法

3.5.1　使用 model. fit 训练模型

在 TensorFlow 中训练模型可以使用 model. fit()方法,其参数如下:

```
model.fit(
    x=None,                          #训练 x 的数据
    y=None,                          #训练 y 的数据
    batch_size=None,                 #每次训练的样本数
    epochs=1,                        #迭代次数
    verbose=1,                       #日志等级;0:为不在标准输出流输出日志信息;
                                     #1:显示进度条;2:每个 epoch 输出一行记录
    callbacks=None,                  #回调函数,即在训练过程中将被调用执行的函数
    validation_split=0.0,            #从 x 和 y 中设置验证集的比例
    validation_data=None,            #验证集的数据,validation_data 设置后会覆盖
                                     #validation_split
    shuffle=True,                    #是否打乱数据,通常为 True
    class_weight=None,               #不重要,一般默认
    sample_weight=None,              #不重要,一般默认
    initial_epoch=0,                 #开始的 epoch 次数设置
    steps_per_epoch=None,            #不重要,一般默认
    validation_steps=None,           #不重要,一般默认
    validation_freq=1,               #不重要,一般默认
    max_queue_size=10,               #不重要,一般默认
    workers=1,                       #进程数,Linux 可以设置多个,Windows 为 1
    use_multiprocessing=False        #是否多进程
)
```

一般设置只需传入 x、y、batch_size、epochs、callbacks 参数，代码如下：

```
#第 3 章/TensorFlowAPI/使用 model.fit 训练模型.py
import tensorflow as tf
from tensorflow.keras import Input, layers, Model
from tensorflow.keras.activations import sigmoid, softmax
import numpy as np

#第 1 步：读取数据 ########################
num_classes = 10                    #类别数
input_shape = (28, 28, 1)
#自动下载手写数字识别的数据集
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
#归一化处理，将值放在 0~1
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
#增维，变成[batch,height,width,channel]
x_train = np.expand_dims(x_train, -1)
x_test = np.expand_dims(x_test, -1)
#将 y 转换为 one_hot 编码
y_train = tf.keras.utils.to_categorical(y_train, num_classes)
y_test = tf.keras.utils.to_categorical(y_test, num_classes)

#第 2 步：构建模型 ########################
#描述输入
inputs = Input(shape=input_shape)
#描述中间过程
x = layers.Flatten()(inputs)
x = layers.Dense(784, activation=sigmoid)(x)
x = layers.Dense(128, activation=sigmoid)(x)
outputs = layers.Dense(10, activation=softmax)(x)
#把输入和输出装载在 Model 这个类里面
#inputs 和 outpus 可以为列表，用来表示多输入或者多输出
model = Model(inputs=inputs, outputs=outputs)
model.summary()
#第 3 步：设置回调函数、学习率和损失函数
callbacks = [
    tf.keras.callbacks.ModelCheckpoint(
        filepath='mnist_weights',
        save_best_only=True,
        monitor='val_loss',
    ),
]
#设置学习率和损失函数
model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
    loss=tf.keras.losses.CategoricalCrossentropy(from_logits=False),
    metrics=['accuracy'],          #评价指定，此处为准确率
)
#第 4 步：训练 ########################
model.fit(
    x=x_train,
    y=y_train,
```

```
        batch_size=2,
        epochs=10,
        validation_split=0.1,
        callbacks=callbacks
    )
```

执行后就会开始进行训练，控制台会展示每个 epoch 训练后的损失、准确率的结果，如图 3-24 所示。

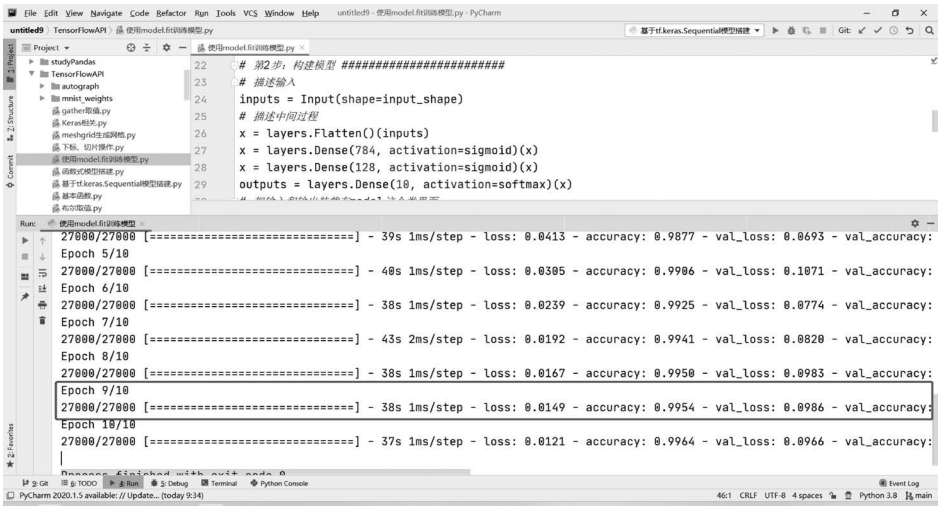


图 3-24　model.fit 训练结果

训练数据用的是手写数字识别数据集，其主要任务是能够识别手写[0,9]的数字，共计 10 个分类，都是灰度图，其尺寸为 28×28。tf.keras.datasets.mnist.load_data()会自动进行下载，共计 6 万张灰度图，如图 3-25 所示。

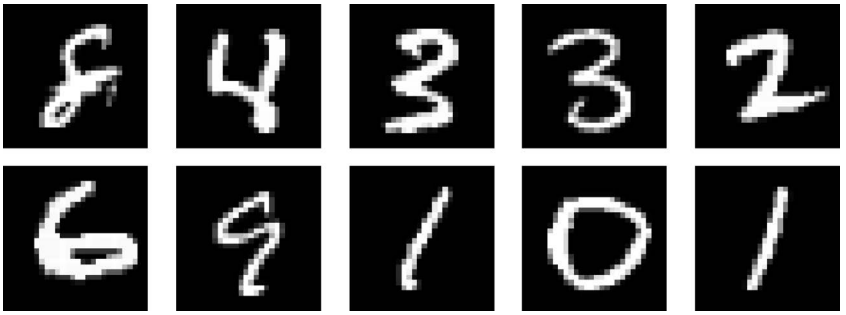


图 3-25　手写数字识别中的数据

因为一张图片，其值域在[0,255]，这里通过 x_train.astype('float32')/255 将其值域限定在[0,1]，可以在一定程度上缓解梯度爆炸。

由于 mnist.load_data()的数据 shape 是 60000×28×28，但是图片应该还有一个通道维度，所以需要通过 np.expand_dims(x_train,−1)变成 60000×28×28×1。

输出值 y 通过 tf. keras. utils. to_categorical(y_test,num_classes)变成[0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]的独热编码,这里表示数字 4。共计 10 个分类,当前图片为哪个分类,那么其位置中的值为 1,其他为 0。

回调函数 callbacks 是一个列表,tf. keras. callbacks. ModelCheckpoint()用来设置模型和权重参数保存等信息,主要参数如下:

```
keras.callbacks.ModelCheckpoint(
    filepath,                   #保存权重文件的位置
    monitor='val_loss',         #需要监视的值,val_accuracy、val_loss 或者 accuracy
    verbose=0,                  #信息展示方式:0 或者 1
    save_best_only=True,        #True 表示当前 epoch 的 monitor 比上一次好时保存
    save_weights_only=False,    #True 只存储权重,下次想用还得搭建模型。
                                #False 会将权重和模型一起保存
    mode='auto',                #save_best_only=True、monitor=val_accuracy,
                                #mode=max;val_loss 时为 min;auto 自动推断
    period=1                    #隔几个 epoch 进行保存
)
```

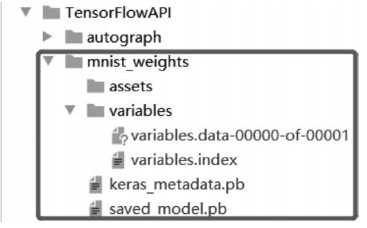


图 3-26 模型和权重文件的目录

在本例中设置 save_weights_only=False,训练完成后存在的文件如图 3-26 所示。

图 3-26 中 assets 目录用于存储计算图;variables 目录包含一个标准训练检查点的变量;saved_model. pb 文件用于存储模型;keras_metadata. pb 为 Keras 格式的权重。

当设置 save_weights_only=True 时,callbacks. ModelCheckpoint()中的 filepath 文件后缀名要为. h5 文件,再次训练得到如图 3-27 所示的内容。

图 3-27 只存储权重参数文件

可在 model. compile()函数中设置学习率和损失函数,其主要参数如下:

```
def compile(self,
            optimizer='rmsprop',    #梯度下降学习方法,俗称优化器,在
                                    #tf.keras.optimizers 模块下
            loss=None,              #损失函数设置,tf.keras.losses 模块下有多种
                                    #损失函数的封装。当然也可以自定义损失函数
            metrics=None,           #评价标准,自带 accuracy 和 mse
            run_eagerly=None,       #是否支持调试模型
)
```

学习率在 tf. keras. optimizers. Adam()中的 learning_rate 设置，CategoricalCrossentropy()为交叉熵损失。最后调用 model. fit()函数 TensorFlow 会自动根据反向传播算法进行训练、学习。

神经网络的学习训练的过程如下：

(1) 读取并处理数据。

(2) 构建神经网络模型。

(3) 回调函数、学习率、损失函数的设置。

(4) 调用 model. fit()进行训练。

注意：model. fit()中的 epochs 的大小与训练计算机的显存有关系，如果设置得太大，则会报 OOM 错误；调用 model. fit()后会自动进行反向传播算法的学习，但是这个过程在代码中是不可见的。

3.5.2　使用 model. train_on_batch 训练模型

训练方式使用 model. fit()虽然非常方便，但是不能在训练过程中调试代码，也不能在训练过程中进行自定义控制，因此 TensorFlow 还提供了 model. train_on_batch 的方法进行训练，代码如下：

```
#第 3 章/TensorFlowAPI/使用 model_train_on_batch 训练模型.py
#第 4 步：训练 ##########################
BATCH_SIZE = 6
epochs = 10
#按某个 batch_size 切割数据，并进行缓存
#训练集
ds_train = tf.data.Dataset.from_tensor_slices(
   (x_train, y_train)
).shuffle(buffer_size=1000).batch(BATCH_SIZE).prefetch(
   tf.data.experimental.AUTOTUNE
).cache()
#验证集
ds_val = tf.data.Dataset.from_tensor_slices(
   (x_test, y_test)
).shuffle(buffer_size=1000).batch(BATCH_SIZE).prefetch(
   tf.data.experimental.AUTOTUNE
).cache()

for epoch in tf.range(1, epochs + 1):
   #将模型的评价内容置为初始状态
   model.reset_metrics()
   #获取当前学习率
   new_lr = model.optimizer.lr
   #获取数据以进行训练
   for x, y in ds_train:
```

```
        #训练
        train_result = model.train_on_batch(x, y)
    #获取验证数据进行训练
    for val_x, val_y in ds_val:
        valid_result = model.train_on_batch(val_x, val_y)
    #操作每 5 个 epoch 学习率变为 2xnew_lr
    if epoch % 5 == 0:
        model.optimizer.lr.assign(new_lr *2)
        tf.print("当前学习率:", new_lr.NumPy())
    #每隔 10 个 epoch 保存一次权重文件
    if epoch % 10 == 0:
        model.save(f'train_on_batch{epoch}.h5')
    #打印训练中的结果
    tf.print("train:", dict(zip(model.metrics_names, train_result)))
    tf.print("valid:", dict(zip(model.metrics_names, valid_result)))
```

在 3.4.5 节的基础上更改了训练的方式，通过自定义 for epoch in tf.range(1，epochs＋1)来控制学习率每 5 个 epoch 变为 2xnew_lr，每 10 个 epoch 保存一次权重文件。model.train_on_batch(x，y)每次循环的单个数据，同样该函数也自动进行反向传播并进行训练、学习，学习的过程并没有用代码来体现。

运行后可以看到目录中存在 train_on_batch10.h5 这个权重文件。观察控制台的输出会发现当前学习率已从 0.001 变为 0.004，如图 3-28 所示。

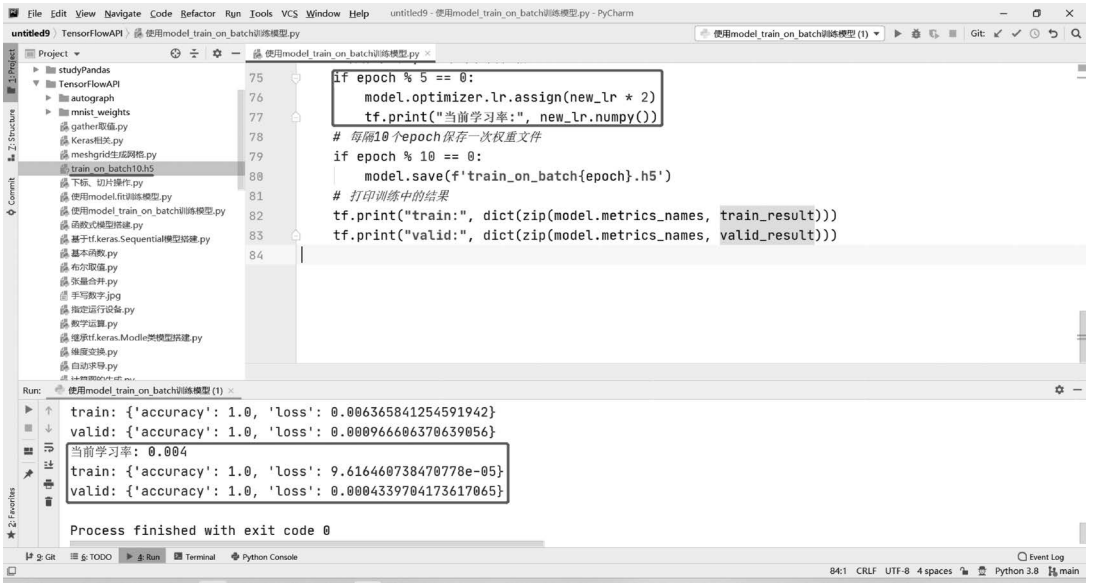


图 3-28 train_on_batch 训练模型

如果 model.save()不指定后缀名，则默认保存为 TensorFlow 的格式，如图 3-29 所示。

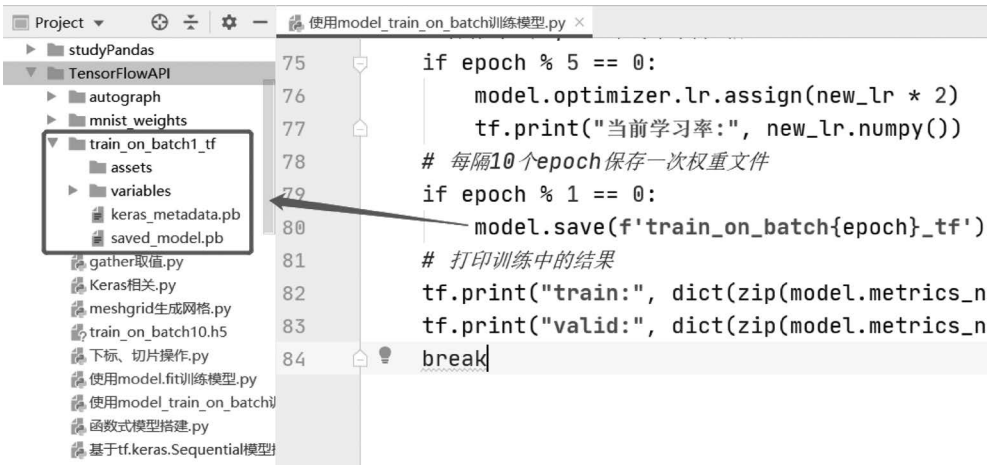


图 3-29　save 保存为 tf 格式的模型和参数

3.5.3　自定义模型训练

训练方式 model.fit()完全是一个黑箱操作，而 model.train_on_batch 虽然能够在一定程度上对训练过程进行控制，但是与反向传播学习的原理并不对应，同时其灵活程度有时不够，此时就可以使用完全自定义模型进行训练、学习，代码如下：

```
#第 3 章/TensorFlowAPI/完全自定义模型训练.py
#第 4 步：训练 ########################
def train_step(model, x, y):
    #model:模型
    #features:训练集 x
    #labels:训练集 y
    #tf.GradientTape()自动求梯度的作用域语句
    with tf.GradientTape() as tape:
        #前向传播
        predictions = model(x, training=True)
        #使用损失函数
        loss = loss_func(y, predictions)
    #根据损失函数自动求梯度
    gradients = tape.gradient(loss, model.trainable_variables)
    #将梯度更新到 model.trainable_variables 属性中，然后由 optimizers 进行指定优化器
    #的梯度下降
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))
    #更新评价指标
    train_loss.update_state(loss)
    train_metric.update_state(y, predictions)


#验证集
def valid_step(model, features, labels):
    #验证集不进行梯度下降更新学习
    predictions = model(features)
```

```
    batch_loss = loss_func(labels, predictions)
    valid_loss.update_state(batch_loss)
    valid_metric.update_state(labels, predictions)


BATCH_SIZE = 6
epochs = 10
#按某个 batch_size 切割数据,并进行缓存
#训练集
ds_train = tf.data.Dataset.from_tensor_slices(
    (x_train, y_train)
).shuffle(buffer_size=1000).batch(BATCH_SIZE).prefetch(
    tf.data.experimental.AUTOTUNE
).cache()
#验证集
ds_val = tf.data.Dataset.from_tensor_slices(
    (x_test, y_test)
).shuffle(buffer_size=1000).batch(BATCH_SIZE).prefetch(
    tf.data.experimental.AUTOTUNE
).cache()

for epoch in tf.range(1, epochs + 1):
    #训练集
     for x, y in ds_train:
        train_step(model, x, y)
    #验证集
    for val_x, val_y in ds_val:
        valid_step(model, val_x, val_y)
    logs = f'Epoch={epoch},' \
          f'Loss:{train_loss.result()},' \
          f'Accuracy:{train_metric.result()},' \
          f'Valid Loss:{valid_loss.result()},' \
          f'Valid Accuracy:{valid_metric.result()}'
    #每隔 10 个 epoch 保存一次权重文件
    if epoch % 5 == 0:
       model.save(f'完全自定义训练{epoch}')
    tf.print(logs)
    #将评价指标重置为 0
    train_loss.reset_states()
    valid_loss.reset_states()
    train_metric.reset_states()
    valid_metric.reset_states()
```

运行结果如图 3-30 所示。

在函数 train_step(model,x,y)中使用 tf.GradientTape()来控制 model(x,training=True)进行前向传播,并定义损失函数 loss_func(y,predictions),然后根据损失 loss 对象结合 tape.gradient(loss,model.trainable_variables)求梯度。

然后根据优化器的选择 optimizer.apply_gradients(zip(gradients,model.trainable_variables))更新梯度下降的权重参数。

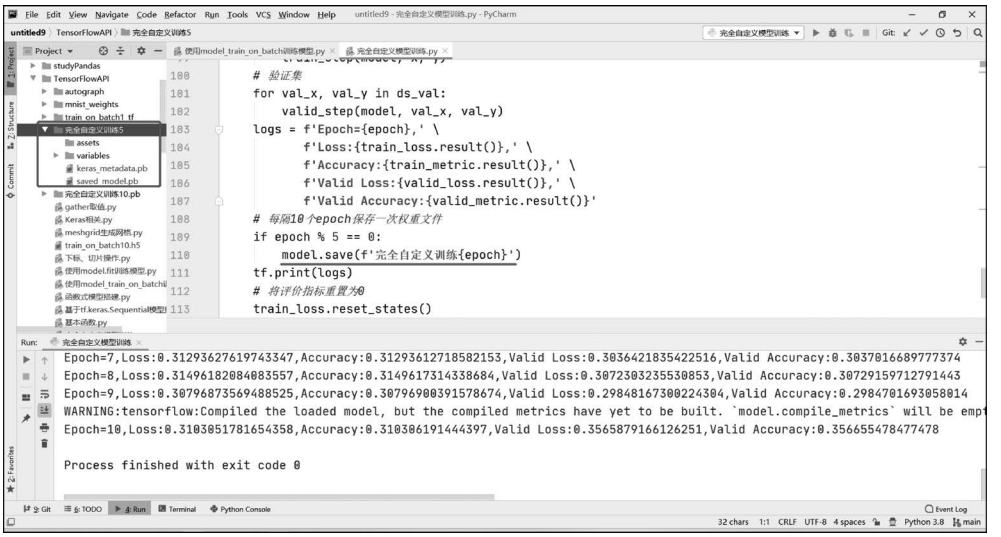


图 3-30　自定义训练结果

函数 valid_step(model,features,labels)是验证集,并不需要进行梯度下降的更新。整个训练、学习过程与神经网络反向传播算法的原理相对应,相对 model.fit()更容易理解,但构造过程稍复杂一些。

总结

在 TensorFlow 中搭建模型可使用 model.fit、model.train_on_batch 和自定义模型训练的方式,其形式灵活多变。

练习

分别使用 3 种搭建模型的方式及 3 种不同的训练方式进行组合,完成手写数字识别的训练。

3.6　TensorFlow 中 Metrics 评价指标

3.6.1　准确率

准确率是分类问题中最为原始的评价指标,准确率的定义是预测正确的结果占总样本的百分比,其公式如下:

$$\text{Accuracy}=\frac{\text{TP}+\text{TN}}{\text{TP}+\text{TN}+\text{FP}+\text{FN}} \tag{3-1}$$

公式中各评价指标的含义如下。

(1) TP(True Positive):真正例,被模型预测为正的正样本。

(2) FP(False Positive):假正例,被模型预测为正的负样本。

(3) FN(False Negative):假负例,被模型预测为负的正样本。

(4) TN(True Negative):真负例,被模型预测为负的负样本。

在 kearas.metrics 模块下框架已自带正例、负例的统计，只需将训练的代码修改如下：

```
#第 3 章/TensorFlowAPI/神经网络的评价指标.py
#设置学习率和损失函数
METRICS = [
    tf.keras.metrics.TruePositives(name='TP'),
    tf.keras.metrics.FalsePositives(name='FP'),
    tf.keras.metrics.TrueNegatives(name='TN'),
    tf.keras.metrics.FalseNegatives(name='FN'),
]
model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
    loss=tf.keras.losses.CategoricalCrossentropy(from_logits=False),
    metrics=METRICS, #评价指定，此处为准确率
    run_eagerly=True
)


def plt_accuracy(history):
    h = history.history
    #history.history 得到的是字典
    #但是 v 不是一个 list，转换成 NDarray 格式以方便计算
    h = {k: np.array(v) for k, v in h.items()}
    train_accuracy = (h['TP'] + h['FP']) / (h['TP'] + h['FP'] + h['TN'] + h['FN'])
    val_accuracy = (h['val_TP'] + h['val_FP']) / (h['val_TP'] + h['val_FP'] +
h['val_TN'] + h['val_FN'])
    plt.plot(history.epoch, train_accuracy, label='训练准确率')
    plt.plot(history.epoch, val_accuracy, label='验证准确率')
    plt.xlabel('迭代次数')
plt.ylabel('准确率')
plt.title('迭代次数与准确率的变化')
    plt.legend(loc='best')
    plt.show()


#第 4 步：训练 ########################
history = model.fit(
    x=x_train,
    y=y_train,
    batch_size=6,
    epochs=3,
    validation_split=0.1,
    callbacks=callbacks
)
#对训练结果进行绘图
plt_accuracy(history)
```

训练结束 history 中将自带 METRICS 的结果，plt_accuracy()通过解析 history 中的内容，通过准确率公式计算后得到结果，如图 3-31 所示。

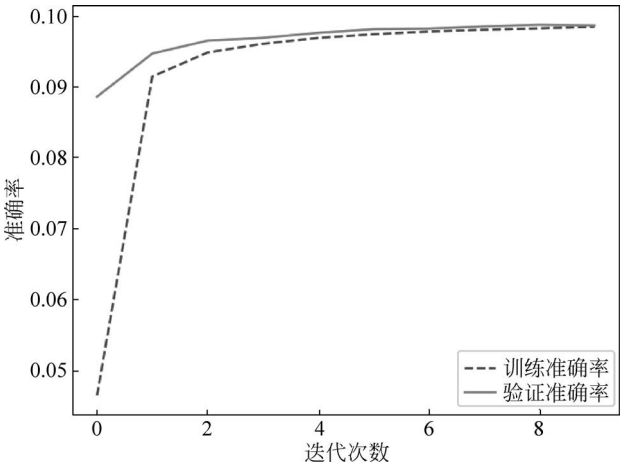


图 3-31　准确率的变化

准确率在类别数据量不平衡时，并不能客观地评价算法的优劣，例如测试集中有 100 个样本，其中 99 个是负样本，1 个是正样本，假设算法模型都能正确地预测出负样本，那么模型的准确率就是 99%，从数值来看效果不错，但事实上这个算法没有任何预测能力（因为正样本没有预测成功，模型偏向于预测负样本）。

3.6.2　精确率

精确率又叫查准率，在所有被预测为正的样本中实际为正样本的概率，其公式如下：

$$\text{Precision}=\frac{\text{TP}}{\text{TP}+\text{FP}} \tag{3-2}$$

精确率和准确率看上去有些类似，但它们是两个不同的概念。精确率代表对正样本结果的预测准确程度，而准确率代表整体的预测准确程度，包括正样本和负样本，代码如下：

```
#第 3 章/TensorFlowAPI/神经网络的评价指标.py
#精准率
def plt_precision(history):
    h = history.history
    #history.history 得到的是个字典
    #但是 v 不是一个 list,转换成 NDarray 格式以方便计算
    h = {k: np.array(v) for k, v in h.items()}
    train_accuracy = (h['TP']) / (h['TP'] + h['FP'])
    val_accuracy = (h['val_TP']) / (h['val_TP'] + h['val_FP'])
    plt.plot(history.epoch, train_accuracy, label='训练精准率')
    plt.plot(history.epoch, val_accuracy, label='验证精准率')
    plt.xlabel('迭代次数')
    plt.ylabel('精准率')
    plt.legend(loc='best')
    plt.title('迭代次数与精准率的变化')
    plt.show()
```

调用执行后的结果如图 3-32 所示。

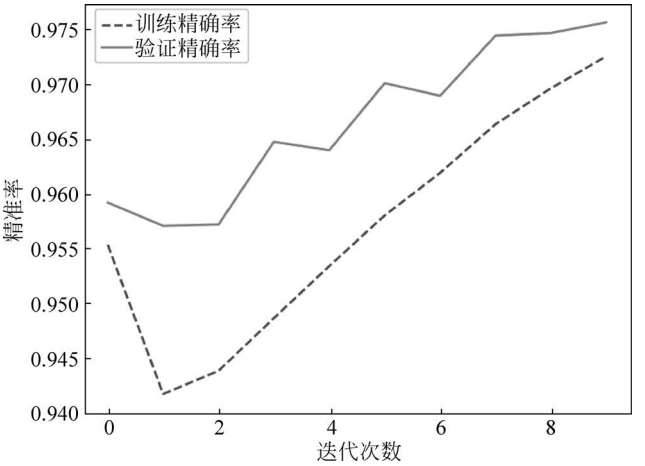


图 3-32 精确率的变化

3.6.3 召回率

召回率又叫查全率，它的含义是在实际为正的样本中被预测为正样本的概率，公式如下：

$$\text{Recall}=\frac{\text{TP}}{\text{TP}+\text{FN}} \tag{3-3}$$

代码如下：

```
#第 3 章/TensorFlowAPI/神经网络的评价指标.py
#召回率
def plt_recall(history):
    h = history.history
    #history.history 得到的是个字典
    #但是 v 不是一个 list,转换成 NDarray 格式以方便计算
    h = {k: np.array(v) for k, v in h.items()}
    train_accuracy = (h['TP']) / (h['TP'] + h['FN'])
    val_accuracy = (h['val_TP']) / (h['val_TP'] + h['val_FN'])
    plt.plot(history.epoch, train_accuracy, label='训练召回率')
    plt.plot(history.epoch, val_accuracy, label='验证召回率')
    plt.xlabel('迭代次数')
    plt.ylabel('召回率')
    plt.legend(loc='best')
    plt.title('迭代次数与召回率的变化')
    plt.show()
```

调用执行后得到的结果如图 3-33 所示。

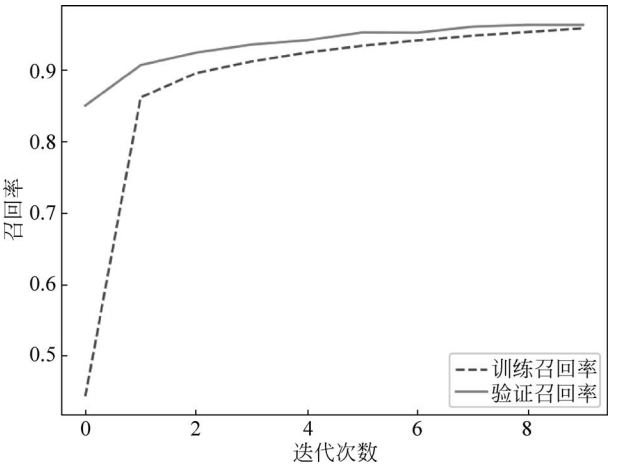


图 3-33　召回率的变化

3.6.4　P-R 曲线

在不同的应用场景中关注的指标不同。例如，在预测股票时更关心精准率，人们更关心那些升值的股票里真正升值的有多少，而在医疗病患中更关心的是召回率，即真的病患里我们预测错的情况应该越小越好。

精准率和召回率是一对此消彼长的关系。例如，在推荐系统中，我们想推送的内容应尽可能地让用户感兴趣，那只能推送我们把握较高的内容，这样就漏掉了一些用户感兴趣的内容，召回率就低了；如果想让用户感兴趣的内容都被推送，就只能推送全部内容，但是这样精准率就低了。在实际工程中，往往需要结合两个指标的结果，从而寻找到一个平衡点，使算法的性能达到最大化，这就是 P-R 曲线，代码如下：

```
#第 3 章/TensorFlowAPI/神经网络的评价指标.py
def plt_PR(history):
    h = history.history
    #history.history 得到的是个字典
    #但是 v 不是一个 list,转换成 NDarray 格式以方便计算
    h = {k: np.array(v) for k, v in h.items()}
    #精准率
    train_precision = (h['TP']) / (h['TP'] + h['FP'])
    val_precision = (h['val_TP']) / (h['val_TP'] + h['val_FP'])
    #召回率
    train_recall = (h['TP']) / (h['TP'] + h['FN'])
    val_recall = (h['val_TP']) / (h['val_TP'] + h['val_FN'])
    plt.plot(train_recall, train_precision, label='训练 P-R 曲线')
    plt.plot(val_recall, val_precision, label='验证 P-R 曲线')
    plt.xlabel('ReCall')
    plt.ylabel('Precision')
    plt.legend(loc='best')
    plt.title('P-R 曲线')
    plt.show()
```

调用执行后得到的结果如图 3-34 所示。

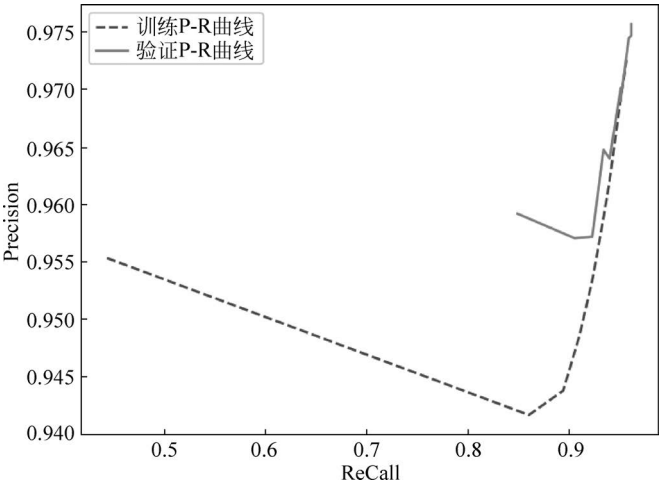


图 3-34 P-R 曲线变化

模型的精准率和召回率互相制约，P-R 曲线越向右上凸出，表示模型的性能越好；由于精准率和召回率更关注正样本的情况，当负样本比较多时 P-R 曲线的效果一般，此时使用 ROC 曲线更合适。

3.6.5 F1-Score

平衡精准率和召回率的另一个指标是 F1-Score，其公式如下：

$$F1=\frac{2\times P\times R}{P+R} \tag{3-4}$$

F1 值越高，代表模型的性能越好，代码如下：

```
#第 3 章/TensorFlowAPI/神经网络的评价指标.py
def plt_F1_Score(history):
    h = history.history
    #history.history 得到的是个字典
    #但是 v 不是一个 list,转换成 NDarray 格式以方便计算
    h = {k: np.array(v) for k, v in h.items()}
    #精准率
    train_precision = (h['TP']) / (h['TP'] + h['FP'])
    val_precision = (h['val_TP']) / (h['val_TP'] + h['val_FP'])
    #召回率
    train_recall = (h['TP']) / (h['TP'] + h['FN'])
    val_recall = (h['val_TP']) / (h['val_TP'] + h['val_FN'])

    f1_train = (2 * train_precision * train_recall) / (train_precision + train_
recall)
    f1_val = (2 * val_precision * val_recall) / (val_precision + val_recall)

    plt.plot(history.epoch, f1_train, label='训练集 F1-Score')
    plt.plot(history.epoch, f1_val, label='验证集 F1-Score')
```

```
    plt.xlabel('迭代次数')
    plt.ylabel('F1-Score')
    plt.legend(loc='best')
    plt.title('迭代次数 F1-Score 的变化')
    plt.show()
```

调用执行后得到的结果如图 3-35 所示。

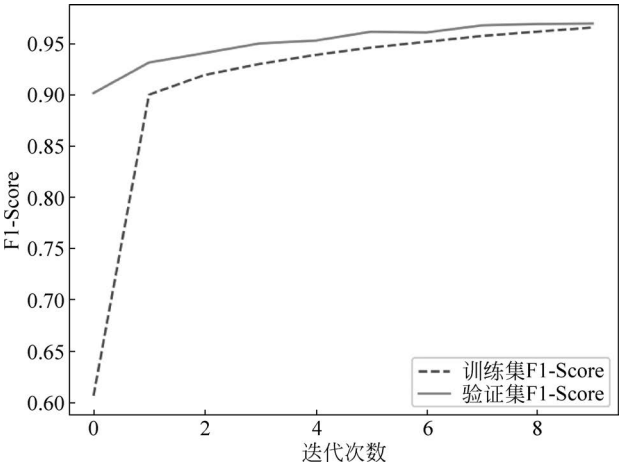


图 3-35　F1-Score 曲线变化

3.6.6　ROC 曲线

在实际数据集中经常会出现类别不平衡现象，即负样本比正样本多（或者相反），而测试数据中的正负样本的分布也可能随着时间的变化而变化，ROC 曲线和 AUC 曲线可以很好地消除样本类别不平衡对评估指标结果的影响。

ROC 曲线可以无视样本的不平衡，主要取决于灵敏度（sensitivity）和特异度（specificity），又称为真正率（TPR）和假正率（FPR）。

（1）真正率（True Positive Rate，TPR），又称为灵敏度。

（2）假负率（False Negative Rate，FNR）。

（3）假正率（False Positive Rate，FPR）。

（4）真负率（True Negative Rate，TNR），又称为特异度。

对应公式为

$$\begin{aligned} \mathrm{TPR} &= \frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FN}} \\ \mathrm{FNR} &= \frac{\mathrm{FN}}{\mathrm{TP}+\mathrm{FN}} \\ \mathrm{FPR} &= \frac{\mathrm{FP}}{\mathrm{TN}+\mathrm{FP}} \\ \mathrm{TNR} &= \frac{\mathrm{TN}}{\mathrm{TN}+\mathrm{FP}} \end{aligned} \tag{3-5}$$

从式(3-5)可知,灵敏度 TPR 与召回率一样,是正样本的召回率,特异度 TNR 是负样本的召回率,而假负率 FNR=1-TPR,假正率 FPR=1-TNR,这 4 个维度都是针对单一类别的预测结果而言的,所以对整体样本是否均衡并不敏感。例如,假设总样本中,90%是正样本,10%是负样本,在这种情况下如果使用准确率进行评价,则是不科学的,但是 TPR 和 TNR 是可以的,因为 TPR 只关注 90%正样本中有多少是被预测正确的,而与那 10%的负样本没有关系,同样 FPR 只注意 10%负样本中有多少是被预测错误的,也与 90%的正样本无关,这样就避免了样本不平衡的问题。

ROC 曲线主要由灵敏度 TPR 和假正率 FPR 构成,其中横坐标为假正率 FPR、纵坐标为灵敏度 TPR,代码如下:

```
#第 3 章/TensorFlowAPI/神经网络的评价指标.py
def plt_ROC(history):
    h = history.history
    #history.history 得到的是个字典
    #但是 v 不是一个 list,转换成 NDarray 格式以方便计算
    h = {k: np.array(v) for k, v in h.items()}
    train_FPR = (h['FP']) / (h['TN'] + h['FP'])
    train_TPR = (h['TP']) / (h['TP'] + h['FN'])

    val_FPR = (h['val_FP']) / (h['val_TN'] + h['val_FP'])
    val_TPR = (h['val_TP']) / (h['val_TP'] + h['val_FN'])

    plt.plot(train_FPR, train_TPR, label='训练集 ROC')
    plt.plot(val_FPR, val_TPR, label='验证集 ROC')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.legend(loc='best')
    plt.title('ROC 曲线')
    plt.show()
```

调用执行后得到的结果如图 3-36 所示。

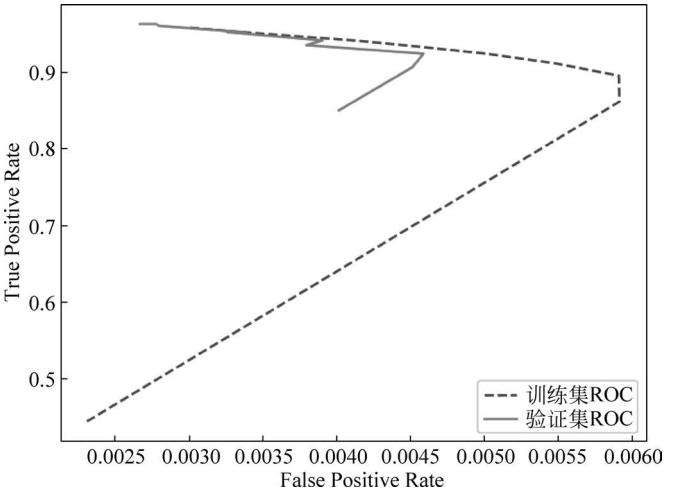


图 3-36 ROC 曲线变化

ROC 曲线中如果 FPR 越低 TPR 越高，则代表性能越高。

3.6.7　AUC 曲线

AUC(Area Under Curve)又称曲线下面积，是处于 ROC 曲线下方的面积的大小，ROC 曲线下方面积越大说明模型性能越好，因此 AUC 同理。如果是完美模型，则它的 AUC＝1，说明所有正例排在负例的前面，模型达到最优状态，代码如下：

```
#第 3 章/TensorFlowAPI/神经网络的评价指标.py
#增加评价指标
METRICS = [
    tf.keras.metrics.AUC(name='AUC')
]
def plt_AUC(history):
    h = history.history
    #history.history 得到的是个字典
    #但是 v 不是一个 list,转换成 NDArray 格式以方便计算
    h = {k: np.array(v) for k, v in h.items()}
    train_auc = h['AUC']
    val_auc = h['val_AUC']
    plt.plot(history.epoch, train_auc, label='训练集 AUC')
    plt.plot(history.epoch, val_auc, label='验证集 AUC')
    plt.xlabel('Epoch')
    plt.ylabel('AUC')
    plt.legend(loc='best')
    plt.title('Epoch 与 AUC 的变化')
    plt.show()
```

调用执行后得到的结果如图 3-37 所示。

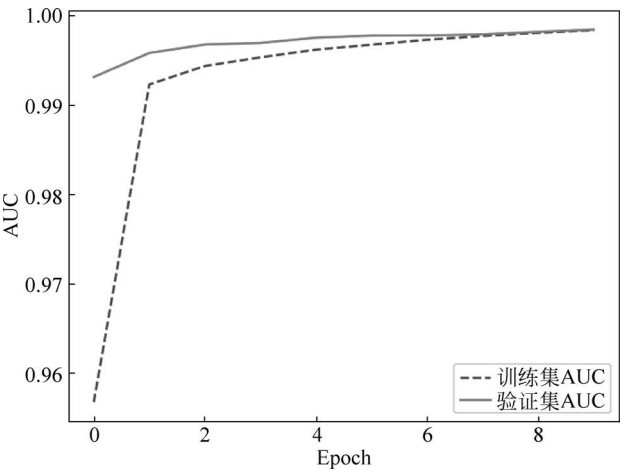


图 3-37　AUC 曲线变化

3.6.8　混淆矩阵

混淆矩阵(Confusion Matrix)又被称为错误矩阵，通过它可以直观地观察到算法的分

类效果，它的每列表示真值样本的分类概率，每行表示预测样本的分类概率，它反映了分类结果的混淆程度，代码如下：

```
#第 3 章/TensorFlowAPI/神经网络的评价指标.py
from collections import deque

#自己实现的混淆矩阵函数，只保存最后 1 个 epoch 的结果
val_ll = deque([], maxlen=1)
def confusion_matrix_acc(y_true, y_pred):
    #取预测值下标的最大值的位置
    pre_label = tf.argmax(y_pred, axis=-1)
    #取真实值下标的最大值的位置
    true_lable = tf.argmax(y_true, axis=-1)
    #调用混淆矩阵函数
    matrix_mat = tf.math.confusion_matrix(true_lable, pre_label)
    #只保存最后一次预测与真实值的矩阵值
    val_ll.append(matrix_mat)
    #这个返回值没有使用，语法要求要有返回值，metrics 默认求平均
    return matrix_mat

#在 compile 中的 metrics 增加自定义评价函数 confusion_matrix_acc
model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
    loss=tf.keras.losses.CategoricalCrossentropy(from_logits=False),
    metrics=[METRICS, confusion_matrix_acc],     #评价指定，此处为准确率
    run_eagerly=True
)

#将矩阵中的数据转换成浮点数
def getMatrix(matrix):
    #求和
    per_sum = matrix.sum(axis=1)
    #得到每个单元格的概率
    ll = [matrix[i, :] / per_sum[i] for i in range(10)]
    #保留两位小数
    matrix = np.around(np.array(ll), 3)
    #如果有异常值，则赋为 0
    matrix[np.isnan(matrix)] = 0
    return matrix


#绘制混淆矩阵
def plt_confusion_matrix(class_num=10):
    #设置类别名称
    kind = [f"数字_{x}" for x in range(class_num)]
    #将分类概率转换为百分比
    data = getMatrix(val_ll[0].NumPy())
    plt.imshow(data, cmap=plt.cm.Blues)
    plt.title("手写数字识别混淆矩阵")              #title
    plt.xlabel("预测值")
    plt.ylabel("真实值")
    plt.yticks(range(class_num), kind)           #y 轴标签
```

```
    plt.xticks(range(class_num), kind, rotation=45)      #x 轴标签
    #数值处理
    for x in range(class_num):
        for y in range(class_num):
            value = float(data[y, x])
            color = 'red' if value else 'green'
            plt.text(x, y, value, verticalalignment='center',
horizontalalignment='center', color=color)
    #自动调整子图参数,使之填充整个图像区域
    plt.tight_layout()
    plt.colorbar()
    plt.savefig("手写数字识别的混淆矩阵.jpg")
    plt.show()
```

调用执行后得到的结果如图 3-38 所示。

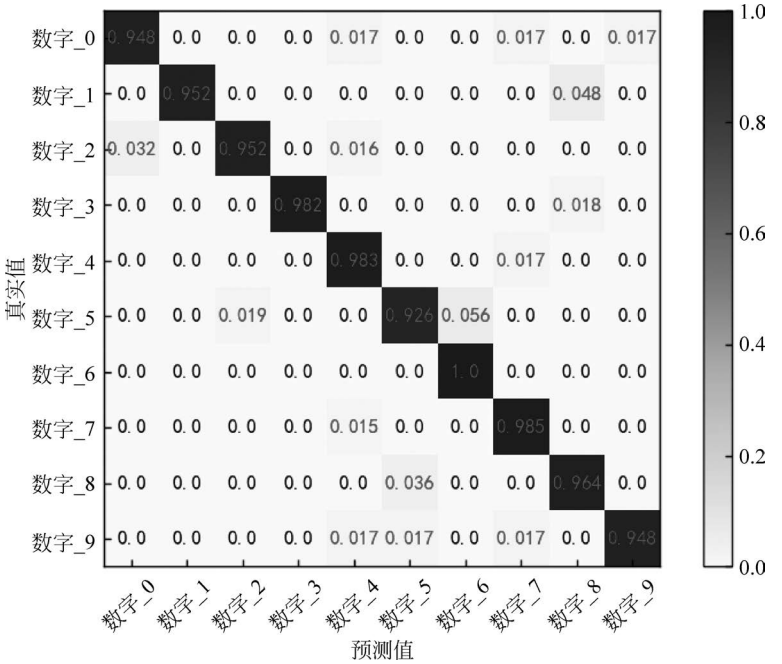


图 3-38　混淆矩阵

如图 3-38 中，当预测“数字_0”时，真实值“数字_0”的概率为 0.948，预测值为“数字_3”的概率是 0.02，也就是说有 0.052 的概率当前模型将“数字_0”预测为其他数字。代码中主要使用了 tf.math.confusion_matrix(true_lable,pre_label)来统计验证集中预测为正样本的数量，因为其返回的是一个整数，所以 getMatrix(matrix)将其转换成了概率。deque([],maxlen=1)是一个队列，因为 model.fit()在训练时，metrics 不能从 confusion_matrix_acc()函数得到返回值 matrix_mat(metrics 本身没有混淆矩阵这个指标)，所以这里使用队列来更新 matrix_mat 的值。

总结

评估模型的指标有准确率、精确率、召回率、P-R 曲线、F1-Score、混淆矩阵等指标。

练习

调试并重写评价指标的代码，观察模型评估指标与代码实现的关系。

3.7 TensorFlow 中的推理预测

推理预测调用 load_model()函数加载保存的权重文件，然后使用 cv2.imread()读入灰度图，并进行维度的变化，最后由 tf.argmax 取最大下标对应的类别，得到推理结果，代码如下：

```
#第 3 章/TensorFlowAPI/预测推理.py
import tensorflow as tf
import cv2
from tensorflow.keras.models import load_model

if __name__ == "__main__":
    model = load_model('完全自定义训练 400')
height,width = 28, 28
    #读预测图片
    img = cv2.imread('test.png', cv2.IMREAD_GRAYSCALE)
    #转换为模型输入的 shape
    img = cv2.resize(img, [height,width])
    #转换为 torch.tensor()类型
    img_tensor = tf.cast(img, dtype=tf.float32)
    #因为模型输入要为[1,28,28,1],所以升维
    img_tensor = tf.reshape(img_tensor, [1, height,width, 1])
    output = model.predict(img_tensor)
    #取预测结果最大值的下标
    index = tf.argmax(output, -1).NumPy()
    #预测类型
    kind = [f"数字_{x}" for x in range(10)]
    print(f"预测={kind[int(index+1)]},概率={round(float(output[:, int(index)]),
2)}")
```

运行结果如图 3-39 所示。

总结

模型搭建、模型训练、模型保存、模型推理预测是神经网络的 4 大阶段，推理阶段也是实际软件应用的主要内容。

练习

完成手写数字识别模型的推理代码。

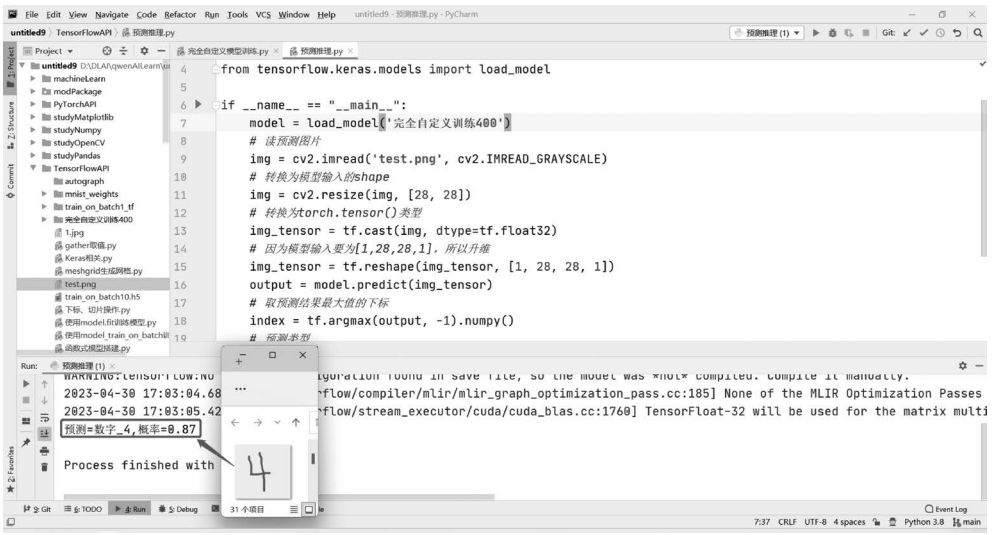


图 3-39　TensorFlow 预测结果

3.8　PyTorch 搭建神经网络

3.8.1　PyTorch 中将数据转换为张量

PyTorch 中将数据转换为张量的函数常见的有 torch.tensor()、torch.from_numpy()、torch.as_tenor()等，代码如下：

```
#第 3 章/PyTorchAPI/转换为张量.py
import torch
import numpy as np
from torch.autograd import Variable

#(1) 创建标量
scalar = torch.tensor([[1, 2], [3, 4]], dtype=torch.float32)
scalar[0, :] = 0
print(scalar)
#(2) 从 NumPy 转换为张量
np_tensor = torch.from_numpy(np.random.rand(2, 2))
print(np_tensor)
#(3) torch 随机生成指定维度的数据
rnd = torch.randn([2, 2])
print(rnd)
print('#'*30)
```

运行结果如下：

```
tensor([[0., 0.],
        [3., 4.]])
```

```
tensor([[0.5540, 0.9942],
        [0.3511, 0.1088]], dtype=torch.float64)
tensor([[ 1.6299, -2.0948],
        [-0.1387, -0.4610]])
```

代码 scalar[0, :]=0 会运行成功，而 TensorFlow 则无法运行。PyTorch 中的张量可以通过索引操作改变值的内容。

如果想定义一个变量，即专门用来存储权重参数并能够进行自动求导的张量，则需要使用 Variable()类，代码如下：

```
#第 3 章/PyTorchAPI/转换为张量.py
#(4) 创建权重参数变量
var_tensor = Variable(scalar, requires_grad=True)
#(5) 损失函数
v_out = torch.mean(var_tensor *var_tensor)
#(6) 反向传播
v_out.backward()
#(7) 获取梯度值
print(var_tensor.grad)
```

运行结果如下：

```
tensor([[0.0000, 0.0000],
        [1.5000, 2.0000]])
```

3.8.2 PyTorch 指定设备

指定设备使用 torch.device()传入 cpu 或者 cuda 可指定硬件来运行，代码如下：

```
#第 3 章/PyTorchAPI/指定设备.py
import torch

tensor1 = torch.tensor([[1, 2], [3, 4]])
tensor2 = torch.tensor([[3, 3], [5, 5]])
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
if device:
   print(tensor1 + tensor2)
```

运行结果如下：

```
tensor([[4, 5],
        [8, 9]])
```

3.8.3 PyTorch 数学运算

常见的数学操作已被 PyTorch 封装，代码如下：

```
#第 3 章/PyTorchAPI/数学运算.py
import torch
```

```
tensor1 = torch.tensor([[1., 2.], [3., 4.]])
tensor2 = torch.tensor([[3., 3.], [5., 5.]])

#矩阵内积
print(torch.matmul(tensor1, tensor2))
#矩阵点乘
print(torch.mul(tensor1, tensor2))
#矩阵相加
print(torch.add(tensor1, tensor2))
#矩阵相减
print(torch.sub(tensor1, tensor2))
#矩阵平方根
print(torch.sqrt(tensor1))
#矩阵求平均
print(torch.mean(tensor1))
```

运行结果如下：

```
tensor([[13., 13.],
        [29., 29.]])
tensor([[ 3., 6.],
        [15., 20.]])
tensor([[4., 5.],
        [8., 9.]])
tensor([[-2., -1.],
        [-2., -1.]])
tensor([[1.0000, 1.4142],
        [1.7321, 2.0000]])
tensor(2.5000)
```

3.8.4　PyTorch 维度变化

维度的变化主要使用 torch. reshape()、torch. unsqueeze()、torch. squeeze()、torch. transpose()函数，代码如下：

```
#第 3 章/PyTorchAPI/维度变化.py
import torch

tensor1 = torch.tensor([[1., 2.], [3., 4.]])
#由原来的 2*2,变换为[1,2,2]
print(torch.reshape(tensor1, [1, 2, 2]).shape)
#在原来的 2*2 的 axis=0 处增加 1 这个维度
new_tensor = torch.unsqueeze(tensor1, 0)
print(new_tensor.shape)
#在原来的 1*2*2 的 axis=0 处,减去 1 这个维度
print(torch.squeeze(new_tensor, 0).shape)
#使用 transpose 变换维度,按 axis 进行变换,由原来的 1*2*2 变成 2*1*2
print(torch.transpose(new_tensor, 1, 0).shape)
```

运行结果如下：

```
torch.Size([1, 2, 2])
torch.Size([1, 2, 2])
torch.Size([2, 2])
torch.Size([2, 1, 2])
```

3.8.5 PyTorch 切片取值

其切片取值的语法与 TensorFlow 保持一致，需要注意的是 PyTorch 中通过切片取值可以进行修改，代码如下：

```
#第 3 章/PyTorchAPI/切片取值.py
import torch

a = torch.randn([4, 28, 28, 3])
print(a[0, ...].shape)          #第 1 个[28,28,3]
print(a[0, 1, :, :].shape)      #[28,3]
print(a[:, :, :, 2].shape)      #[4,28,28],因为它取的是前面所有的值,而最里层的是
                                #第 0 个,所以是[4,28,28]
print(a[..., 2].shape)          #[4,28,28]
print(a[:, 0, :, :].shape)      #[4,28,3]
```

运行结果如下：

```
torch.Size([28, 28, 3])
torch.Size([28, 3])
torch.Size([4, 28, 28])
torch.Size([4, 28, 28])
torch.Size([4, 28, 3])
```

3.8.6 PyTorch 中 gather 取值

PyTorch 中的 gather 取值与 TensorFlow 类似，代码如下：

```
#第 3 章/PyTorchAPI/gather 取值.py
import torch

params = torch.tensor([[1, 2], [3, 4]])
#沿着 1 轴取值：[0,0]表示第 1 行取下标为 0,0 的值；[3,4]表示第 2 行取下标为 1,1 的值
print(torch.gather(params, 1, index=torch.tensor([[0, 0], [1, 1]])))
#沿着 0 轴取值
print(torch.gather(params, 0, index=torch.tensor([[1, 1], [0, 0]])))
```

运行结果如下：

```
tensor([[1, 1],
        [4, 4]])
tensor([[3, 4],
        [1, 2]])
```

3.8.7　PyTorch 中布尔取值

在 PyTorch 中通常使用切片的语法来过滤数据，当然也可以使用 torch.masked_select()函数，代码如下：

```
#第 3 章/PyTorchAPI/布尔取值.py
import torch

data = torch.tensor([[1, 2], [3, 4], [5, 6]])
#获取 data 中>2 的数
mask = data > 2
print('mask:', mask)
#通常直接使用 data[data > 2]来表示
print("输出满足条件的数：", data[mask], data[data > 2])
#按布尔掩码选择元素
print("输出满足条件的数：", torch.masked_select(data, mask, out=None))
```

运行结果如下：

```
mask: tensor([[False, False],
        [ True, True],
        [ True, True]])
输出满足条件的数：tensor([3, 4, 5, 6]) tensor([3, 4, 5, 6])
输出满足条件的数：tensor([3, 4, 5, 6])
```

3.8.8　PyTorch 张量合并

张量合并使用 torch.cat()函数来完成，代码如下：

```
#第 3 章/TensorFlowAPI/张量合并.py
import torch

t1 = torch.randn([4, 28, 28])
t2 = torch.randn([2, 28, 28])
#按 axis=0 进行合并
print(torch.cat([t1, t2], dim=0).shape)
```

运行结果如下：

```
torch.Size([6, 28, 28])
```

3.8.9　PyTorch 模型搭建

PyTorch 搭建神经网络模型主要集中在 torch.optim 模块和 torch.nn 模块下，其模块下的概要结构如图 3-40 所示。

PyTorch 模型创建继承自 torch.nn.Module 类，可以在__init__()初始构造函数中描述属性，然后在 forward()方法中实现网络的前向传播，其描述方法与 3.4.3 节类似，代码如下：

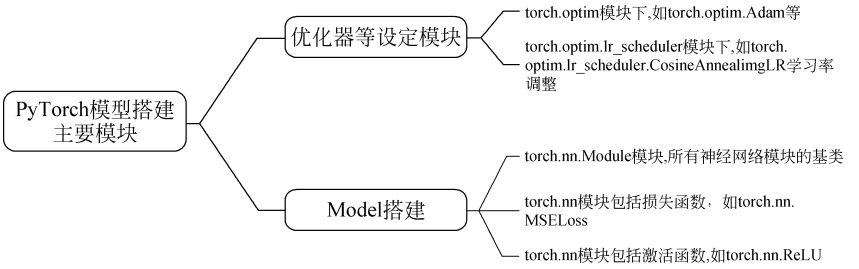


图 3-40　PyTorch 模型搭建主要模块

```
#第 3 章/PyTorchAPI/手写数字识别.py
import torch
#pip install thop
from thop import profile

#搭建模型
class MyModel(torch.nn.Module):
    def __init__(self):
        #继承 Model 类中的属性
        super(MyModel, self).__init__()
        #全连接
        self.flat = torch.nn.Flatten()
        #Linear(input_channel,output_channel)
        self.fc = torch.nn.Sequential(
            torch.nn.Linear(784, 784),
            torch.nn.Linear(784, 128)
        )
        #输出
        self.out = torch.nn.Linear(128, 10)

    def forward(self, input_x):
        x = self.flat(input_x)
        #第 1 个网络层的计算和输出
        x = torch.sigmoid(self.fc(x))
        return torch.sigmoid(self.out(x))

if __name__ == "__main__":
    model = MyModel()
    print("输出网络结构：", model)
    #PyTorch 输入的维度是 batch_size,channel,height,width
    input = torch.randn(1, 1, 28, 28)
    flops, params = profile(model, inputs=(input,))
    print('浮点计算量：', flops)
    print('参数量：', params)
```

运行结果如下：

```
输出网络结构：MyModel(
  (flat): Flatten(start_dim=1, end_dim=-1)
  (fc): Sequential(
```

```
      (0): Linear(in_features=784, out_features=784, bias=True)
      (1): Linear(in_features=784, out_features=128, bias=True)
    )
    (out): Linear(in_features=128, out_features=10, bias=True)
  )
  [INFO] Register count_linear() for <class 'torch.nn.modules.linear.Linear'>.
  [INFO] Register zero_ops() for <class 'torch.nn.modules.container.Sequential'>.
  浮点计算量：716288.0
  参数量：717210.0
```

torch.nn.Sequential()与 tf.keras.Sequential()一样，将神经网络各层“串起来”。获取模型的参数量使用了 thop 这个库。

注意：PyTorch 的输入维度是[batch_size,channel,height,width]，而 TensorFlow 的输入维度是[batch_size,height,width,channel]。

3.8.10　PyTorch 模型自定义训练

PyTorch 的自定义训练与 3.5.3 节类似，主要步骤如下：

(1) 读取训练数据和验证数据。

(2) 设置优化器、学习率等。

(3) 设置损失函数。

(4) 对训练集进行学习。

(5) 对验证集进行学习。

(6) 保存模型和参数。

使用手写数字识别模型 MyModel 进行训练，代码如下：

```
#第 3 章/PyTorchAPI/手写数字识别自定义训练.py
if __name__ == "__main__":
    model = MyModel()
    print("输出网络结构：", model)
    #训练过程
    #(1)加载数据集
    batch_size = 60
    learning_rate = 1e-5
    epochs = 10
    #手写数字识别的数据集会被自动下载到当前./data 目录
    #训练集
    train_loader = torch.utils.data.DataLoader(
        torchvision.datasets.MNIST(
            './data/',
            train=True,
            transform=torchvision.transforms.ToTensor(),
            download=True
        ),
        batch_size=batch_size,
```

```
        shuffle=True,
    )
    #验证集
    val_loader = torch.utils.data.DataLoader(
        torchvision.datasets.MNIST(
            './data/',
            train=False,
            download=True,
            transform=torchvision.transforms.ToTensor(),
        ),
        batch_size=batch_size, shuffle=True)
    #(2)设置优化器、学习率
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
    #(3)设置损失函数
    criterion = torch.nn.CrossEntropyLoss()
    #(4)训练
    def model_train(data, label):
        #清空模型的梯度
        model.zero_grad()
        #前向传播
        outputs = model(data)
        #计算损失
        loss = criterion(outputs, label)
        #计算 acc
        pred = torch.argmax(outputs, dim=1)
        accuracy = torch.sum(pred == label) / label.shape[0]
        #反向传播
        loss.backward()
        #权重更新
        optimizer.step()
        #进度条显示 loss 和 acc
        desc = "train.[{}/{}] loss: {:.4f}, Acc: {:.2f}".format(
            epoch, epochs, loss.item(), accuracy.item()
        )
        return loss, accuracy, desc
    #验证
    @torch.no_grad()
    def model_val(val_data, val_label):
        #前向传播
        val_outputs = model(val_data)
        #计算损失
        loss = criterion(val_outputs, val_label)
        #计算前向传播结果与标签一致的数量
        val_pred = torch.argmax(val_outputs, dim=1)
        num_correct = torch.sum(val_pred == val_label)
        return loss, num_correct

    #根据指定的 epoch 进行学习
    for epoch in range(1, epochs + 1):
        #进度条
        process_bar = tqdm(train_loader, unit='step')
        #开始训练模式
        model.train(True)
```

```
        #因为 train_loader 是按 batch_size 划分的,所以都要进行学习
        for step, (data, label) in enumerate(process_bar):
            #调用训练函数
            loss, accuracy, desc = model_train(data, label)
            #输出日志
            process_bar.set_description(desc)
            #在训练的最后 1 组数据后面进行验证
            if step == len(process_bar) - 1:
                #用来计算总数
                total_loss, correct = 0, 0
                #根据验证集进行验证
                model.eval()
                for _, (val_data, val_label) in enumerate(val_loader):
                    #验证集前向传播
                    loss, num_correct = model_val(val_data, val_label)
                    total_loss += loss
                    correct += num_correct
                #计算总测试的平均 acc
                val_acc = correct / (batch_size * len(val_loader))
                #计算总测试的平均 loss
                val_Loss = total_loss / len(val_loader)
                #验证集的日志
                var_desc = " val.[{}/{}]loss: {:.4f}, Acc: {:.2f}".format(
                    epoch, epochs, val_Loss.item(), val_acc.item()
                )
                #显示训练集和验证集的日志
                process_bar.set_description(desc + var_desc)
        #进度条结束
        process_bar.close()
        #保存模型和权重
        torch.save(model, './torch_mnist.pt')
```

运行结果如图 3-41 所示。

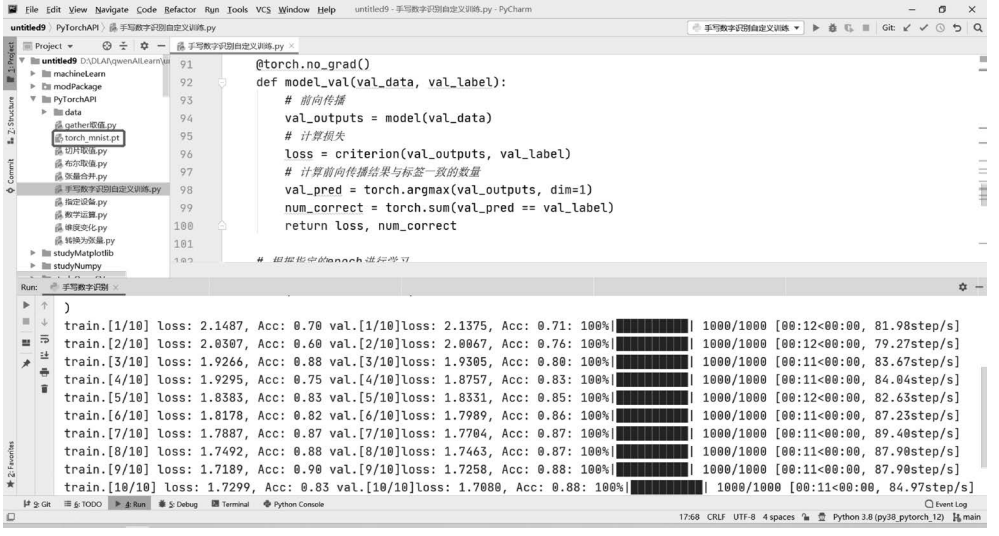


图 3-41　PyTorch 训练手写数字识别结果

仔细观察代码会发现与 TensorFlow 的训练代码类似，例如训练数据处理通过 torch.utils.data.DataLoader()来控制，优化器通过 torch.optim.Adam()设置，损失函数的设置通过 torch.nn.CrossEntropyLoss()实现。在模型训练函数 model_train(data,label)中，由 outputs＝model(data)进行前向传播，由 loss＝criterion(outputs,label)进行损失函数的计算，由 loss.backward()进行反向传播，由 optimizer.step()进行梯度下降后的权重更新，而当训练完毕时调用 torch.save(model,'./torch_mnist.pt')保存模型和权重。

3.8.11 PyTorch 调用 Keras 训练

PyTorch 也支持调用 Keras 模块进行训练，首先调用 torchkeras 库中的 KerasModel()类，然后使用 model.fit()就能进行训练了，代码如下：

```
#第 3 章/PyTorchAPI/手写数字识别 Keras 风格训练.py
#需要安装的包
#pip install torchkeras,wandb,accelerate
from torchkeras import KerasModel
if __name__ == "__main__":
    #(1)加载数据集
    batch_size = 60
    learning_rate = 1e-5
    epochs = 10
    #手写数字识别的数据集会被自动下载到当前./data 目录
    #训练集
    train_loader = torch.utils.data.DataLoader(
        torchvision.datasets.MNIST(
            './data/',
            train=True,
            transform=torchvision.transforms.ToTensor(),
            download=True
        ),
        batch_size=batch_size,
        shuffle=True,
    )
    #验证集
    val_loader = torch.utils.data.DataLoader(
        torchvision.datasets.MNIST(
            './data/',
            train=False,
            download=True,
            transform=torchvision.transforms.ToTensor(),
        ),
        batch_size=batch_size, shuffle=True)

    #(2) 调用集成函数,设置优化器、学习率等
    model = MyModel()
    model = KerasModel(
        net=model,
        loss_fn=torch.nn.CrossEntropyLoss(),
        optimizer=torch.optim.Adam(model.parameters(), lr=1e-5)
```

```
    )
    print(model)
    #使用 model.fit()函数进行训练
    history = model.fit(
        train_data=train_loader,
        val_data=val_loader,
        epochs=10,
        patience=3, #每隔 3 个 epoch 保存权重
        ckpt_path='./torch_mnist_keras.pt',
    )
```

运行结果如图 3-42 所示。

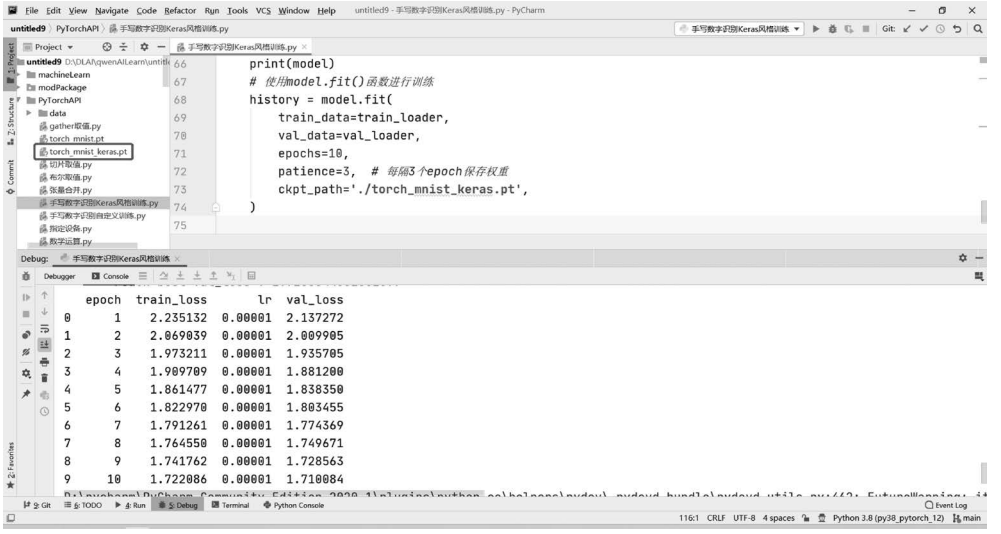


图 3-42　PyTorch 调用 Keras 训练手写数字识别结果

3.8.12　PyTorch 调用 TorchMetrics 评价指标

评价指标库 TorchMetrics 在分类模型中支持 20 种指标，如图 3-43 所示。

使用方法，代码如下：

```
#第 3 章/PyTorchAPI/手写数字识别评价指标.py
from torchmetrics import MetricCollection, Accuracy, Precision, Recall,
ConfusionMatrix
#绘制混淆矩阵
def plt_confusion_matrix(data, class_num=10):
    #设置类别名称
    kind = [f"数字_{x}" for x in range(class_num)]
    data = np.around(data.NumPy(), 2)
    plt.imshow(data, cmap=plt.cm.Blues)
    plt.title("手写数字识别混淆矩阵") #title
    plt.xlabel("预测值")
```

手写数字识别评价指标.py × __init__.py ×

```
)
from torchmetrics.classification import (  # noqa: E402
    AUROC,#AUC-ROC曲线
    ROC, # ROC曲线
    Accuracy, # 准确率
    AveragePrecision,
    CalibrationError,
    CohenKappa,
    ConfusionMatrix, # 混淆矩阵
    Dice,
    ExactMatch,
    F1Score, # F1-Score
    FBetaScore,
    HammingDistance,
    HingeLoss,
    JaccardIndex,
    MatthewsCorrCoef,
    Precision, # 精确率
    PrecisionRecallCurve, # P-R曲线
    Recall, # 召回率
    Specificity,
    StatScores,
)
```

图 3-43　TorchMetrics 评价指标

```
    plt.ylabel("真实值")
    plt.yticks(range(class_num), kind)                    #y 轴标签
    plt.xticks(range(class_num), kind, rotation=45)       #x 轴标签
    #数值处理
    for x in range(class_num):
        for y in range(class_num):
            value = data[y, x]
            color = 'red' if value else None
            plt.text(x, y, value, verticalalignment='center',
horizontalalignment='center', color=color)
    #自动调整子图参数,使之填充整个图像区域
    plt.tight_layout()
    plt.colorbar()
    plt.savefig("手写数字识别的混淆矩阵.jpg")
    plt.show()


if __name__ == "__main__":
    model = MyModel()
    #训练过程
    #(1)加载数据集
    batch_size = 60
    learning_rate = 1e-5
    epochs = 10
```

```
    #训练集
    train_loader = torch.utils.data.DataLoader(
        torchvision.datasets.MNIST(
            './data/',
            train=True,
            transform=torchvision.transforms.ToTensor(),
            download=True
        ),
        batch_size=batch_size,
        shuffle=True,
    )
    #验证集
    val_loader = torch.utils.data.DataLoader(
        torchvision.datasets.MNIST(
             './data/',
            train=False,
            download=True,
            transform=torchvision.transforms.ToTensor(),
        ),
        batch_size=batch_size, shuffle=True)
    #(2)设置优化器、学习率
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
    #(3)设置损失函数
    criterion = torch.nn.CrossEntropyLoss()
    #定义评价指标
    metrics = {
        'acc': Accuracy(num_classes=10, task='multiclass'),
        'prec': Precision(num_classes=10, task='multiclass',
average='macro'),
        'recall': Recall(num_classes=10, task='multiclass',
average='macro'),
        'matrix': ConfusionMatrix(num_classes=10, task='multiclass',
normalize='true')
    }
    train_collection = MetricCollection(metrics)
    val_collection = MetricCollection(metrics)
    #(4)训练
    def model_train(data, label):
        #清空模型的梯度
        model.zero_grad()
        #前向传播
        outputs = model(data)
        #计算损失
        loss = criterion(outputs, label)
        train_collection.forward(outputs, label)
        #反向传播
        loss.backward()
        #权重更新
        optimizer.step()
        return loss
```

```
    #验证
    @torch.no_grad()
    def model_val(val_data, val_label):
        #前向传播
        val_outputs = model(val_data)
        #计算损失
        loss = criterion(val_outputs, val_label)
        val_collection.forward(val_outputs, val_label)
        return loss


    #根据指定的 epoch 进行学习
     for epoch in range(1, epochs + 1):
        #进度条
        process_bar = tqdm(train_loader, unit='step')
        #开始训练模式
        model.train(True)
        #因为 train_loader 是按 batch_size 划分的,所以都要进行学习
        for step, (data, label) in enumerate(process_bar):
            #调用训练函数
            loss = model_train(data, label)
            #在训练的最后 1 组数据后面进行验证
            if step == len(process_bar) - 1:
                #根据验证集进行验证
                model.eval()
                for _, (val_data, val_label) in enumerate(val_loader):
                    #验证集前向传播
                    loss = model_val(val_data, val_label)
        train_metrics = train_collection.compute()
        val_metrics = val_collection.compute()
        train_desc = f"train: acc={train_metrics['acc']},precision={train_
metrics['prec']},recall=={train_metrics['recall']}"
        val_desc = f" val:acc={val_metrics['acc']},precision={val_metrics['prec']},
recall=={val_metrics['recall']}"
        process_bar.close()
        #保存模型和权重
        torch.save(model, './torch_mnist.pt')
    #展示最后 1 次 epoch 的混淆矩阵图
    plt_confusion_matrix(val_metrics['matrix'])
```

运行结果如图 3-44 所示。

代码中通过 from torchmetrics import MetricCollection，Accuracy 来引入模型评价指标，然后由 train_collection＝MetricCollection(metrics)来收集每次训练和验证中的指标值，而在 model_train(data,label)训练函数中只需增加 train_collection.forward(outputs,label)就可以完成指标的收集工作，训练完成后由 train_collection.compute()完成所有数据的计算并得到相关指标的值，然后就可以根据需要对数据进行可视化操作，例如代码中混淆矩阵的可视化。

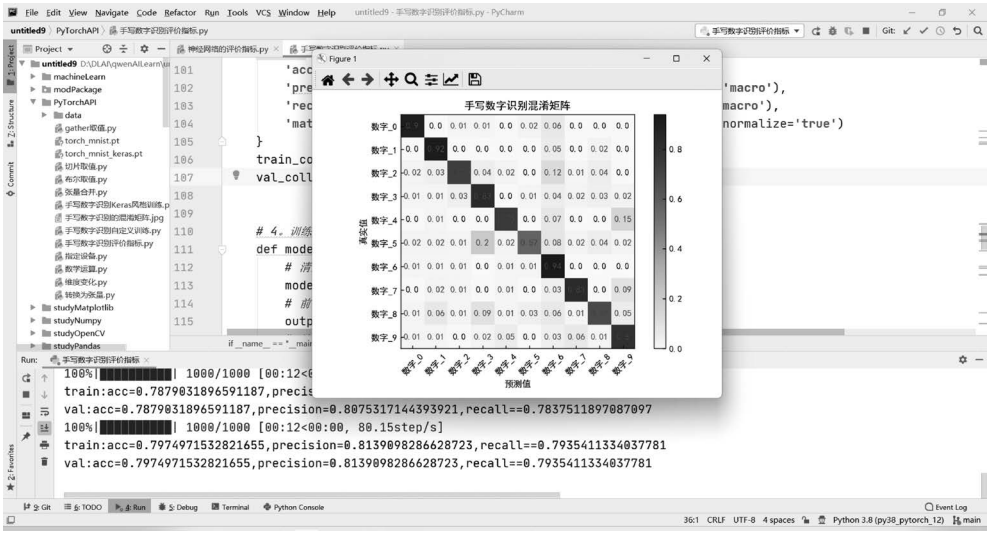


图 3-44　增加 TorchMetrics 评价指标运行结果

3.8.13　PyTorch 中推理预测

预测推理代码只需由 torch.load()导入模型权重文件，然后由 cv2.imread()读取图片，转换为模型后输入 shape，因为模型的输入维度为[batch_size，channel，height，width]，所以需要由 img_tensor.unsqueeze(0)进行升维，然后由 model(img_tensor)进行前向传播，再根据输出的概率通过 torch.max(output，1)排序，得到概率最大值，代码如下：

```
#第 3 章/PyTorchAPI/预测推理.py
import torch
import cv2

if __name__ == "__main__":
    #加载模型文件
    model = torch.load("torch_mnist.pt")
    #读预测图片
    img = cv2.imread('test.png', cv2.IMREAD_GRAYSCALE)
    #转换为模型输入的 shape
    img = cv2.resize(img, [28, 28])
    #转换为 torch.tensor()类型
    img_tensor = torch.tensor(img, dtype=torch.float32)
    #因为模型输入为[1,28,28,1],所以要升维
    img_tensor = img_tensor.unsqueeze(0)
    #前向传播
    output = model(img_tensor)
    #取预测结果最大值的下标
    max_value, pre = torch.max(output, 1)
    #预测类型
    kind = [f"数字_{x}" for x in range(10)]
    print(f"预测={kind[pre+1]},概率={round(max_value.item(), 2)}")
```

运行结果如图 3-45 所示。

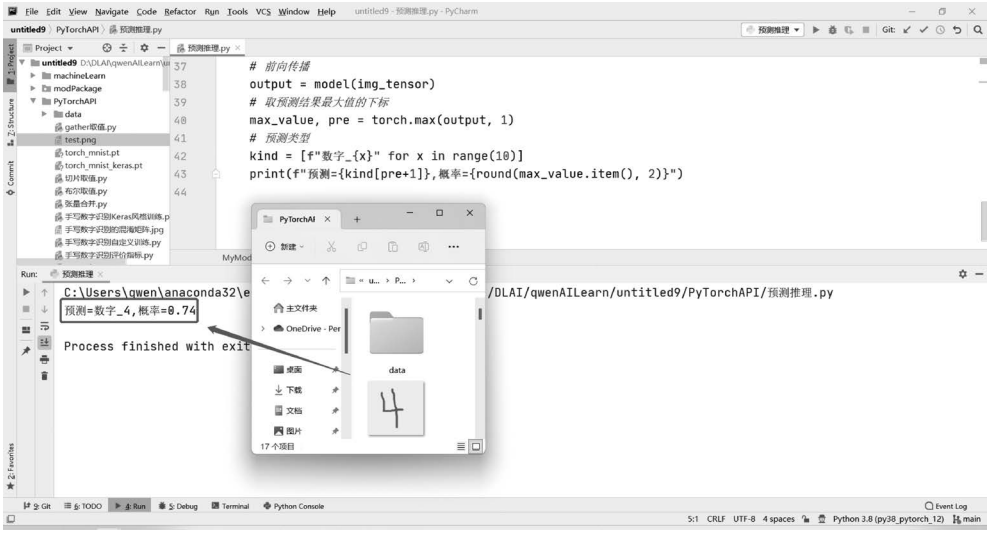


图 3-45　PyTorch 手写数字推理结果

总结

PyTorch 的主要 API 也由张量操作、模型搭建、模型训练、TorchMetrics 模型评估、模型推理构成，相对 TensorFlow 来讲 API 管理更加规范，调试起来较为方便。

练习

完成由 PyTorch 构建手写数字识别的模型，并进行训练、推理工作。