# 第 5 章 模型训练与复用

经过前面几个章节内容的介绍我们已经逐步迈入了深度学习的大门。所谓工欲善其事必先利其器，因此在接下来的这章内容中将会逐一对深度学习模型在训练过程中将会用到的一些辅助技能进行介绍，包括如何有效地对模型参数进行管理、怎么从本地文件中载入参数、如何保证模型训练过程的可追溯、模型的持久化与迁移方法及模型的多 GPU 训练和预处理结果缓存等内容。

## 5.1 参数及日志管理

在深度学习模型的实现过程中由于会频繁调整整个模型的超参数，例如需要突然新增 1 个丢弃率参数或者模型的控制参数等，而且这样的操作经常是跨多个函数或模块的，如果依旧采用参数名来传递参数，就会变得十分复杂。如果模型参数数量较多，则通过参数名来传递参数也会显得代码十分臃肿。

同时，由于在深度学习模型中通常会有较多的超参数，模型在训练过程中也会输出相应的评估结果、损失值，甚至是部分权重参数结果等，为了使整个模型的训练过程可追溯，因此就需要有效地将这些信息给保存下来，以便不时之需。

### 5.1.1 参数传递

例如对于某个深度学习模型来讲，其训练部分的函数实现过程如下：

```
1  def train(train_file_path = os.path.join('data', 'train.txt'),
2            val_file_path = os.path.join('data', 'val.txt'),
3            test_file_path = os.path.join('data', 'test.txt'),
4            split_sep = '_!_', is_sample_shuffle = True, batch_size = 16,
5            learning_rate = 3.5e - 5, max_sen_len = None, num_labels = 3,epochs = 5):
6       dataset = get_dataset(train_file_path, val_file_path,max_sen_len,
7                             test_file_path, split_sep, is_sample_shuffle)
8       model = get_model(max_sen_len, num_labels)
```

从上述代码可以看出，第 1～5 行定义了很多需要用到的参数，并且在第 6～8 行中分别将这些参数传入了对应的函数中。这样看起来似乎没有问题，但是此时如果需要将一个参

数添加到 get_model()函数中，例如加入丢弃率来提高模型的泛化能力，并且 get_model()函数在不同模块都要用到丢弃率这个参数，如果直接采用新加参数的方式，则难免涉及诸多地方的修改。

因此，对于模型参数有效管理的一种高效做法就是在所有地方均传入一个实例化的类对象，通过类对象访问类成员变量的方式来获取相应的参数值，这样在增删模型参数时只需在原始类对象实例化的地方修改一次就能实现。首先，需要定义一个配置类，代码如下：

```
class ModelConfig(object):
    def __init__(self,train_file_path = os.path.join('data', 'train.txt'),
                 val_file_path = os.path.join('data', 'val.txt'),
                 test_file_path = os.path.join('data', 'test.txt'),
                 split_sep = '_!_',is_sample_shuffle = True,
                 batch_size = 16,learning_rate = 3.5e - 5,
                 max_sen_len = None,num_labels = 3,epochs = 5):
        self.train_file_path = train_file_path
        self.val_file_path = val_file_path
        self.test_file_path = test_file_path
        self.split_sep = split_sep
        self.is_sample_shuffle = is_sample_shuffle
        self.batch_size = batch_size
        self.learning_rate = learning_rate
        self.max_sen_len = max_sen_len
        self.num_labels = num_labels
        self.epochs = epochs
```

在上述代码中定义了模型所需要用到的参数，并且可以通过以下方式进行访问，示例代码如下：

```
if __name__ == '__main__':
    config = ModelConfig(epochs = 10)
    print(f"epochs = {config.epochs}")
#epochs = 10
```

对于上面 train()函数中的示例可以改写为如下形式：

```
def train(config):
    dataset = get_dataset(config)
    model = get_mode(config)
```

通过这样的管理方式，即使后续需要在模型中新增参数也只需在类 ModelConfig 中新增 1 个成员变量，然后在需要的地方以 config.para_name 的方式来获取。

### 5.1.2　参数载入

在上述示例中，我们介绍了如何通过定义一个 ModelConfig 来管理模型参数，但是在一些场景中还需要从本地载入一个模型参数文件。例如在后面介绍 BERT 模型时就需要从本地载入一个名为 config.json 的参数文件，形式如下：

```
{
    "attention_probs_DropOut_prob": 0.1,
    "hidden_act": "gelu",
```

```
    "hidden_DropOut_prob": 0.1,
    "hidden_size": 768,
    "initializer_range": 0.02,
    "intermediate_size": 3072,
}
```

对于使用存放在本地文件中的参数，一种最直观的方式就是直接将这些参数手动添加到 ModelConfig 类的成员变量中。当然，通常来讲一种更常见的做法是在 ModelConfig 类中实现一种方法，以此来加载这些本地参数，代码如下：

```
1  @classmethod
2  def from_json_file(cls, json_file):
3      with open(json_file, 'r') as reader:
4          text = reader.read()
5      model_config = cls()
6      for (key, value) in dict(json.loads(text)).items():
7          model_config.__dict__[key] = value
8      return model_config
```

在上述代码中，第 1 行@classmethod 表示申明 from_json_file()方法作为类 ModelConfig 的一个类方法，其作用是在不实例化一个 ModelConfig 类对象之前同样可以调用类 ModelConfig 中的方法，即后续可以通过 ModelConfig.from_json_file()的形式进行调用，这一点在后续载入 BERT 预训练模型时也会遇到。第 3～4 行用于打开配置文件。第 6～7 行用于遍历文件中的每个参数并加入类 ModelConfig 的成员变量中，其中 dict(json.loads(text))表示将文本内容转换为 dict 对象。

最后，通过以下方式便可加载参数和访问相关参数：

```
1  if __name__ == '__main__':
2      config = ModelConfig.from_json_file("./config.json")
3      print(config.hidden_DropOut_prob)
4      print(config.hidden_size)
5  #0.1
6  #768
```

以上完整的示例代码可以参见 Code/Chapter05/C01_ConfigManage/E03_LoadConfig.py 文件。

### 5.1.3 定义日志函数

在模型开发中，可以借助 logging 这个 Python 模块来完成上述功能(如果没有通过 pip install logging 命令安装)。同时，为了满足日志信息也能在控制端输出等功能，需要基于 logging 再改进一下，代码如下：

```
1  import logging
2  import os,sys
3  def logger_init(log_file_name = 'monitor',log_level = logging.DEBUG,
4                  log_dir = './logs/',only_file = False):
5      if not os.path.exists(log_dir):
6          os.makedirs(log_dir)
7      log_path = os.path.join(log_dir, log_file_name + '_' +
```

```
                                    str(datetime.now())[:10] + '.txt')
8       formatter = '[%(asctime)s] - %(levelname)s:
                     [%(filename)s][%(lineno)s] %(message)s'
9       datefmt = "%Y-%d-%m %H:%M:%S'"
10      if only_file:
11          logging.basicConfig(filename = log_path, level = log_level,
12                              format = formatter, datefmt = datefmt)
13      else:
14          logging.basicConfig(level = log_level, format = formatter,
                    datefmt = datefmt, handlers = [logging.FileHandler(log_path),
15                  logging.StreamHandler(sys.stdout)])
```

在上述代码中，第 3 行中 log_file_name 用于指定日志文件名的前缀；log_level 用于指定日志的输出等级，常见的有 3 种，即 WARNING、INFO 和 DEBUG，其重要性降序排列（重要性越高输出内容越少）；log_dir 用于指定日志的保存目录；only_file 用于指定是否输出到日志文件。第 5～6 行用于判断日志目录是否存在，如果不存在，则创建。第 7 行用于构建最终日志保存的路径，并且同时在文件名后面加上当天日期。第 8～9 行用于定义日志信息的输出格式，其中 lineno 表示打印语句所在的行号。第 10～15 行根据条件判断日志输出方式。最后，在 logs 文件中将会生成一个类似名为 monitor_2023-03-03.txt 的日志文件。

### 5.1.4　日志输出示例

在完成上述工作后便可以在任意模块或者文件中使用 logging 来记录日志，下面是一个具体的示例。首先在 classA.py 文件中新建一个名为 ClassA 的类，代码如下：

```
1  import logging
2  class ClassA(object):
3      def __init__(self):
4          logging.info(f"我在{__name__}中!")
5          logging.debug(f"我在文件{__file__}中,这是一条 Debug 信息!")
6          logging.warning(f"我在文件{__file__}中,这是一条 Warning 信息!")
```

在上述代码中，第 4 行__name__表示取当前模块的名称，即 classA。第 5 行__file__表示所在文件的绝对路径。

接着在 classB.py 文件中新建一个名为 ClassB 的类，代码如下：

```
1  class ClassB(object):
2      def __init__(self):
3          logging.info(f"我在{__name__}中!")
4          logging.debug(f"我在文件{__file__}中,这是一条 Debug 信息!")
```

最后在 main.py 文件中调用这两个类，并输出相应的日志信息，代码如下：

```
1  from classA import ClassA
2  from classB import ClassB
3  from log_manage import logger_init
4  import logging
5
6  def log_test():
7      a = ClassA()
8      b = ClassB()
```

```
    logging.info(f"我在{__name__}中!")

if __name__ == '__main__':
    logger_init(log_file_name = 'monitor', log_level = logging.INFO,
                log_dir = './logs',only_file = False)
    log_test()
```

在运行完上述代码后,日志文件 monitor_2023_03_04.txt 和终端里就会输出如下所示的日志信息:

```
我在 classA 中!
我在文件 DeepLearningWithMe/Code/Chapter05/C02_LogManage/classA.py 文件中,这是一条 Warning 信息!
我在 classB 中!
我在__main__中!
```

可以发现,classA 和 classB 这两个模块中的日志信息都被打印出来了,而且也都满足了跨模块日志打印的需求,但是可以发现,logging.debug 这样的信息并没有打印出来,其原因就在于通过 logger_init()函数初始化时指定的日志输出等级为 logging.INFO,这就意味着不会输出调试信息。当然,只需将 log_level 指定为 logging.DEBUG 便可输出所有信息。

### 5.1.5 打印模型参数

在介绍完日志的打印输出方法后,进一步只需在上面 ModelConfig 类的定义中加入如下几行代码便可以在模型训练时打印相关的模型信息:

```
class ModelConfig(object):
    def __init__(self, ):
        .....
        logging.info("#<------------------------>")
        for key, value in self.__dict__.items():
            logging.info(f"#{key} = {value}")

if __name__ == '__main__':
    logger_init(log_file_name = 'monitor', log_level = logging.DEBUG,
                log_dir = './logs', only_file = False)
    config = ModelConfig()
```

在上述代码中,第 4～6 行用于遍历类中所有的成员变量(模型参数)并打印输出。最后,在控制台和日志文件中便会输出类似如下的信息:

```
#<------------------------>
#batch_size = 16
#learning_rate = 3.5e-05
#num_labels = 3
#epochs = 5
```

以上完整的示例代码可以参见 Code/Chapter05/C02_LogManage 文件夹。

### 5.1.6 小结

本节首先介绍了在编写代码模型的过程中参数管理的重要性和必要性,并介绍了如何

定义一个类配置类并通过类成员的方式来管理和获取参数，然后详细介绍了如何载入本地文件中的参数值并添加到配置类中进行使用；接着进一步介绍了如何基于 logging 模块来定义一个初始化函数；最后详细展示了如何使用 logging 在各个模块中将相关信息打印到同一个日志文件中。在实际使用过程中只需在需要输出日志信息的地方通过函数 logging.info()进行打印，然后在主函数运行的地方调用 logger_init()函数来初始化即可完成日志信息的输出或打印。

## 5.2　TensorBoard 可视化

在网络模型的训练过程中一般需要通过观察模型损失值或准确率的变化趋势来确定模型的优化方向，例如学习率的动态调整、惩罚项系数等。同时，对于图像处理方向来讲可能还希望能够可视化模型的特征图或者样本分类类别在空间中的分布情况等。虽然这些结果也可以在网络训练结果后取对应的变量并通过 Matplotlib 进行可视化，但是我们更希望在模型的训练过程中就能对其各种状态进行可视化。

因此，对于上述需求可以借助谷歌开源的 TensorBoard 工具来实现。在接下来的内容中将会详细地介绍如何在 PyTorch 中通过 TensorBoard 来对各类变量及指标进行可视化[1]。

### 5.2.1　安装与启动

如果需要在 PyTorch 中使用 TensorBoard，则除了需要安装 TensorBoard 工具本身之外，还需要安装 TensorFlow。因为 TensorBoard 中的部分可视化功能在使用中会依赖 TensorFlow 框架，例如 add_embedding()函数。

对于 TensorFlow 和 TensorBoard 的安装，只需执行安装 TensorFlow 的命令便可以同时完成两者的安装：

```
pip install tensorflow
```

同时，由于只是借助于 TensorBoard 来进行可视化，因此在安装 TensorFlow 时不用区分是 GPU 还是 CPU 版本，两者都可以，也就是说假如某台主机上装了 GPU 版本的 PyTorch，而不管装的是 GPU 版本还是 CPU 版的 TensorFlow，TensorBoard 都可以正常使用。

在安装成功后可以通过如下命令进行测试：

```
TensorBoard --logdir=runs
```

会出现如下提示：

```
TensorBoard 1.15.0 at http://localhost:6006/ (Press 快捷键 Ctrl+C to quit)
```

此时便可以通过 http://127.0.0.1:6006 链接来访问 TensorBoard 的可视化页面，如图 5-1 所示。

如果发现打不开这个地址，则可以尝试通过如下命令来启动，然后通过 http://127.0.

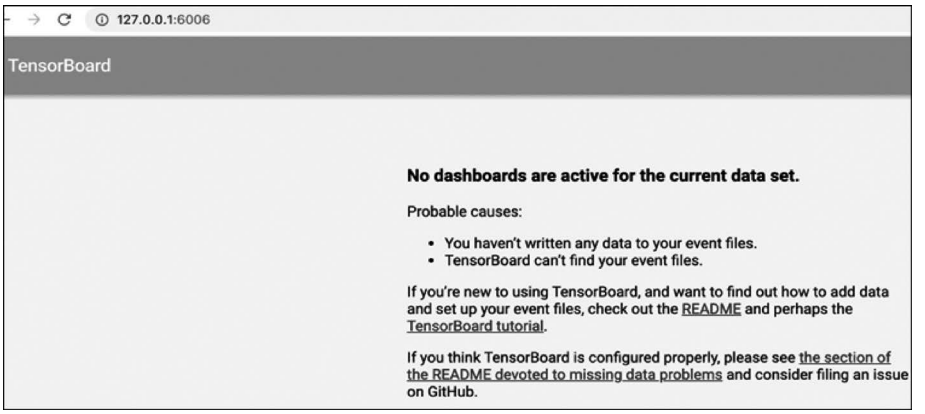


图 5-1 TensorBoard 启动成功界面图

0.1:6006 链接来访问：

```
TensorBoard --logdir=runs --host 0.0.0.0
```

其中,--logdir 用来指定可视化文件的目录地址,后续会详细介绍。

## 5.2.2 连接与访问

上面我们介绍了如何在本地安装与启动 TensorBoard,而更常见的一种场景便是在远程主机上运行代码,但需要在本地计算机上查看可视化运行结果。如果需要实现这种功能,则通常来讲有两种方法,下面分别进行介绍。

**1. 通过 IP 直接访问**

在通过 IP 直接访问的方案中,不管是在类似于腾讯云或阿里云上租用的主机还是实验室中的专用主机,在完成 TensorBoard 安装并启动后在自己计算机上都可以通过地址 http://IP:6006 来访问,但需要注意的是,上面的 IP 对于公网主机(如腾讯云)来讲指的是主机的公网 IP,对于实验室或学校的主机来讲指的则是局域网的内网 IP。同时,如果在远程主机上启动 TensorBoard 后发现在本地并不能打开,则可以通过以下方式来排查。

(1) 公网主机：在后台的安全策略里面查看 6006 这个端口有没有被打开,如果没有,则需要打开；查看 IP 是否为公网 IP,在主机的后台管理页面可以看到。

(2) 局域网主机：查看本地计算机是否和主机处于同一网段；查看主机的 6006 端口是否被打开,如果没有,则可以参考如下命令打开。

```
1  firewall-cmd --zone=public --list-ports #查看已开放端口
2  firewall-cmd --zone=public --add-port=6006/tcp --permanent #开放 6006 端口
3  firewall-cmd --zone=public --remove-port=6006/tcp --permanent #关闭 6006 端口
4  firewall-cmd --reload #配置立即生效
```

**2. 端口转发访问**

当然,除了可以通过 IP 直接访问外,还可以借助 SSH 反向隧道技术进行访问,例如服务器只开了 22 端口而且你没有权限打开其他端口的情况。在这种情况下可以通过下面两

种方式进行远程连接：

（1）命令行终端：如果你的命令行终端支持 SSH 命令（例如较新的 Windows 10 的 CMD 或者 Linux 等），则可以直接通过下面这条命令进行连接：

```
ssh -L 16006:127.0.0.1:6006 username@ip
```

这条命令的含义就是将服务器上的 6006 端口的信息通过 SSH 转发到本地的 16006 端口，其中 16006 是本地的任意端口（无限制），只要不和本地应用有冲突就行，后面则是对应的用户名和 IP。

上述命令连接成功并在远程主机上启动 TensorBoard 后，在本地通过浏览器打开地址 http://127.0.0.1:16006 即可访问。

（2）XShell 工具：如果你的计算机终端不支持 SSH 命令，则可以通过 XShell 工具来实现 SSH 反向代理访问。首先需要安装好 XShell 工具，然后在安装完成后按照如下步骤进行配置。

第 1 步：新建连接。单击“新建”按钮，如图 5-2 所示。

然后配置主机信息，如图 5-3 所示。


图 5-2　新建连接（1）

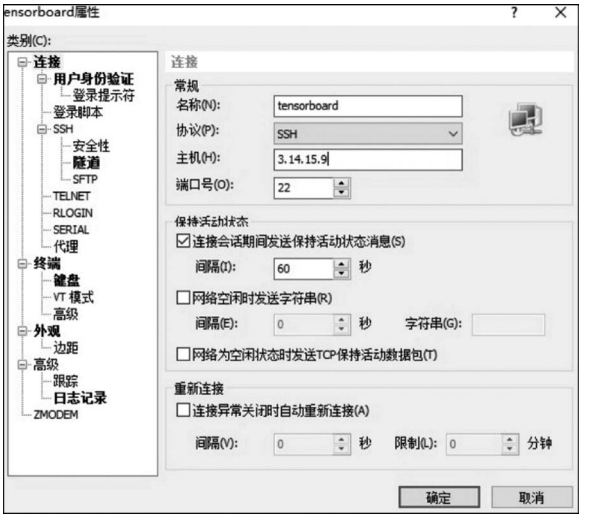


图 5-3　新建连接（2）

第 2 步：配置代理。选择侧边栏的“隧道”，并单击右侧的“添加”按钮，如图 5-4 所示。

接着进行端口代理配置，如图 5-5 所示。

配置完成后，单击“确定”按钮，如图 5-6 所示。

完成上述两步配置之后，再双击刚刚新建的这个连接，输入用户名和密码之后即可登录到主机并对相应的端口进行监听与转发。最后同样只需先在当前远程主机上启动 TensorBoard，然后在本地浏览器中通过地址 http://127.0.0.1：16006 进行访问。


图 5-4 配置代理(1)

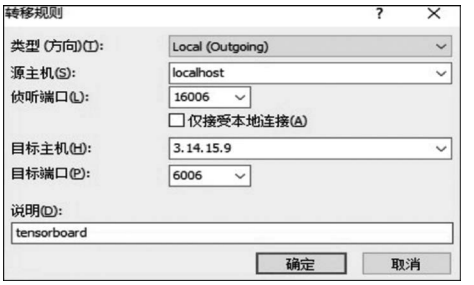


图 5-5 配置代理(2)


图 5-6 配置代理(3)

## 5.2.3 TensorBoard 使用场景

在完成 TensorBoard 的安装和调试后,下面将逐一通过实际示例来介绍如何使用 TensorBoard 提供的不同可视化模块。下面先通过一个简单的标量可视化示例来完整地介绍 TensorBoard 的使用方法。

### 1. add_scalar 方法

这种方法通常用来可视化网络训练时的各类标量参数,例如损失、学习率和准确率等。如下便是 add_scalar 方法的使用示例:

```
1  from torch.utils.TensorBoard import SummaryWriter
2  if __name__ == '__main__':
```

```
writer = SummaryWriter(log_dir = "runs/result_1", flush_secs = 120)
for n_iter in range(100):
    writer.add_scalar(tag = 'Loss/train',
                      scalar_value = np.random.random(),
                      global_step = n_iter)
    writer.add_scalar('Loss/test', np.random.random(), n_iter)
writer.close()
```

在上述代码中，第 1 行用来导入相关的可视化模块。第 3 行用于实例化一个可视化类对象，log_dir 用于指定可视化数据的保存路径，flush_secs 表示指定多少秒将数据写入本地一次（默认为 120s）。第 5～7 行则利用 add_scalar 方法来对相关标量进行可视化，其中 tag 表示对应的标签信息。

在上述代码运行之前，先进入该代码文件所在的目录，然后运行如下命令来启动 TensorBoard：

```
TensorBoard --logdir = runs
TensorBoard 1.15.0 at http://localhost:6006/ (Press 快捷键 Ctrl + C to quit)
```

可以看出，logdir 后面的参数就是上面代码第 3 行里的参数。同时，根据提示在浏览器中打开上述链接便可以看到如图 5-1 所示的界面。在运行上述程序后便会在当前目录中生成如图 5-7 所示的文件夹，其中 result_1 便是前面所指定的子目录，而以 events.out 开始的文件则是生成的可视化数据文件。

当程序运行时 TensorBoard 会加载如图 5-7 所示的文件并在网页端进行渲染，如图 5-8 所示。


图 5-7　可视化数据文件图

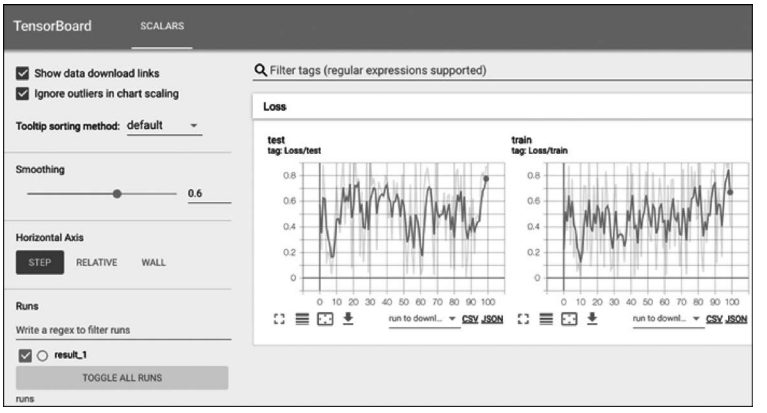


图 5-8　TensorBoard 可视化结果图

图 5-8 为 TensorBoard 的可视化结果图，其中右边的 Loss 标签就是上面第 5 行代码中指定 Loss/train 参数的前缀部分，也就是说如果想把若干幅图放到一个标签下，就要保持其前缀一致。例如这里的 Loss/train 和 Loss/test 这两幅图都将被放在 Loss 这个标签下。同时，在勾选左上角的 Show data download links 后，还能单击图片下方的按钮来分别下载 SVG 向量图、原始图片的 CSV 或 JSON 数据。

在图 5-8 的左边部分，Smoothing 参数用来调整右侧可视化结果的平滑度，Horizontal Axis 用来切换不同的显示模式，Runs 下面用来勾选需要可视化的结果。例如后续在初始化 SummaryWriter()时指定 log_dir="runs/result_2"，那么在 result_1 的下方便会再出现一个 result_2 的选项，这时可以选择多个结果同时可视化展示。

### 2. add_graph 方法

从名字可以看出 add_graph 方法用于可视化模型的网络结构图，其用法示例如下：

```
1  import torchvision
2  def add_graph(writer):
3      img = torch.rand([1, 3, 64, 64], dtype=torch.float32)
4      model = torchvision.models.AlexNet(num_classes=10)
5      writer.add_graph(model, input_to_model=img)
```

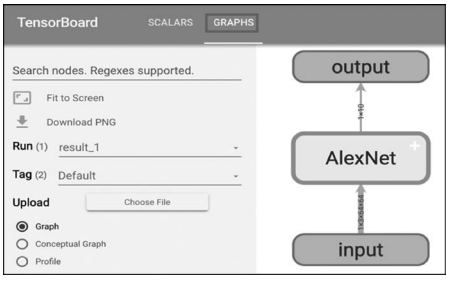


图 5-9 add_graph 可视化结果图

为了示例简洁，我们这里又把 SummaryWriter()中的 add_graph()方法写成了一个函数。在上述代码中，第 4 行用于返回一个网络模型。第 5 行用于对网络结构图进行可视化，其中 input_to_model 参数为模型所接收的输入，这类似于 TensorFlow 中的 fed_dict 参数。

上述代码运行完成后，便可以在网页端看到可视化结果，如图 5-9 所示。

右侧网络结构中的每个模块都可以通过双击进行展开，而左边则是相关模式的切换。

### 3. add_scalars 方法

这种方法与 add_scalar 方法的差别在于 add_scalars 在一张图中可以绘制多条曲线，只需以字典的形式传入参数，如下为 add_scalars 方法的使用示例。

```
1  def add_scalars(writer):
2      r = 5
3      for i in range(100):
4          scalar_dict = {'xsinx':i*np.sin(i / r), 'xcosx':i*np.cos(i / r)}
5          writer.add_scalars(main_tag='scalars1/P1',
6                             tag_scalar_dict=scalar_dict, global_step=i)
7        writer.add_scalars('scalars1/P2', {'xsinx': i*np.sin(i / (2 * r)),
8                           'xcosx': i * np.cos(i / (2 * r))}, i)
9        writer.add_scalars('scalars2/Q1', {'xsinx': i*np.sin((2*i) / r),
10                          'xcosx': i * np.cos((2 * i) / r)}, i)
11       writer.add_scalars('scalars2/Q2', {'xsinx': i*np.sin(i / (0.5*r)),
12                           'xcosx': i * np.cos(i / (0.5 * r))}, i)
```

在上述代码中一共画了 4 幅图，分别对应代码中的 4 个 add_scalars 方法；同时在每幅图里面都对应了两条曲线，即 add_scalars 方法里的 tag_scalar_dict 参数，并且这里一共用了两个标签来进行分隔，即 scalars1 和 scalars2，最后可视化的结果如图 5-10 所示。

### 4. add_histogram 方法

直方图的示例用法比较简单，示例代码如下：

```
1  def add_histogram(writer):
2      for i in range(10):
3          x = np.random.random(1000)
4          writer.add_histogram('distribution centers/p1', x + i, i)
5          writer.add_histogram('distribution centers/p2', x + i * 2, i)
```

上述代码运行结束后可视化结果如图 5-11 所示。

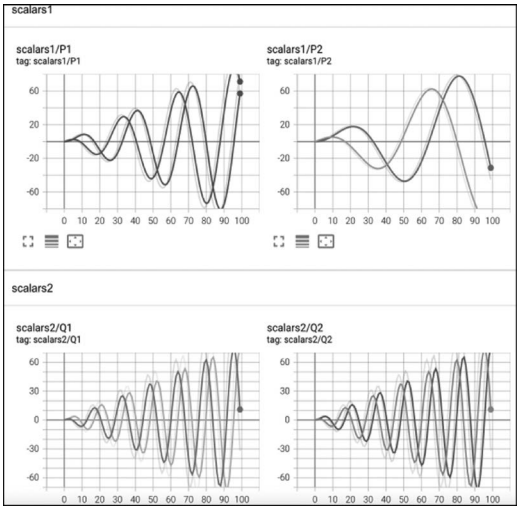


图 5-10　add_scalars 可视化结果图

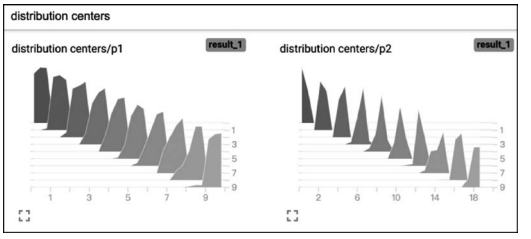


图 5-11　add_histogram 可视化结果图

### 5. add_image 方法

add_image 方法用来可视化相应的像素矩阵，例如本地图片或者网络中的特征图等，代码如下：

```
1  def add_image(writer):
2      from PIL import Image
3      img1 = np.random.randn(1, 100, 100)
4      writer.add_image('../img/imag1', img1)
5      img2 = np.random.randn(100, 100, 3)
6      writer.add_image('../img/imag2', img2, dataformats = 'HWC')
7      img = Image.open('./dufu.png')
8      img_array = np.array(img)
9      writer.add_image(tag = 'local/dufu', img_tensor = img_array,
                        dataformats = 'HWC')
```

在上述代码中，第 3～4 行用于生成一个形状为$[C,H,W]$的三维矩阵并进行可视化。第 5～6 行用于生成形状为$[H,W,C]$的三维矩阵并可视化，同时需要在 add_image 中指定矩阵的维度信息，因此可以看出 add_image 方法接受的默认格式为$[C,H,W]$。第 7～9 行用于先从本地读取一张图片，然后对其进行可视化。最后，可视化的结果如图 5-12 所示。

### 6. add_images 方法

从名字可以看出，该方法用于一次性可视化多张像素图，代码如下：

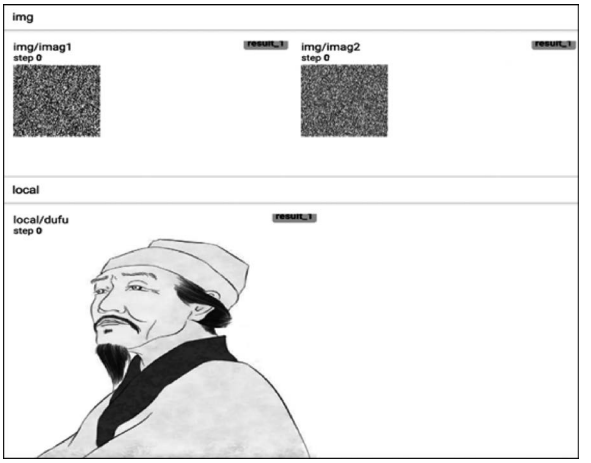


图 5-12　add_image 可视化结果图

```
1  def add_images(writer):
2      img1 = np.random.randn(8, 100, 100, 1)
3      writer.add_images('imgs/imags1', img1, dataformats = 'NHWC')
4      img2 = np.zeros((16, 3, 100, 100))
5      for i in range(16):
6          img2[i, 0] = np.arange(0,10000).reshape(100,100)/10000/16 * i
7          img2[i, 1] = (1 - np.arange(0,10000).reshape(100,100)/10000)/16 * i
8      writer.add_images('imgs/imags2', img2) #默认形状为(N, 3, H, W)
```

在上述代码中，第 2～3 行用于生成 8 张通道数为 1 的像素图并进行可视化。第 4～8 行用于生成 16 张通道数为 3 的像素图并进行可视化。最后可视化的结果如图 5-13 所示。

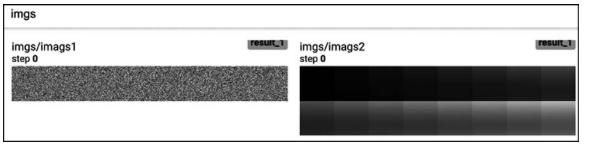


图 5-13　add_images 可视化结果图

### 7. add_figure 方法

这种方法用来将 Matplotlib 包中的 figure 对象可视化到 TensorBoard 的网页端，用于展示一些较为复杂的图片，其示例用法如下：

```
1  def add_figure(writer):
2      fig = plt.figure(figsize = (5, 4))
3      ax = fig.add_axes([0.12, 0.1, 0.85, 0.8])
4      xx = np.arange( - 5, 5, 0.01)
5      ax.plot(xx, np.sin(xx), label = "sin(x)")
6      ax.legend()
7      fig.suptitle('Sin(x) figure\n\n', fontweight = "bold")
8      writer.add_figure("figure", fig, 4)
```

在上述代码中，第 2～7 行根据 Matplotlib 包绘制相应的图像，其中第 3 行用来指定图片的坐标信息，分别表示[left，bottom，width，height]。第 8 行将其在 TensorBoard 中进行

可视化。最后可视化的结果如图 5-14 所示。

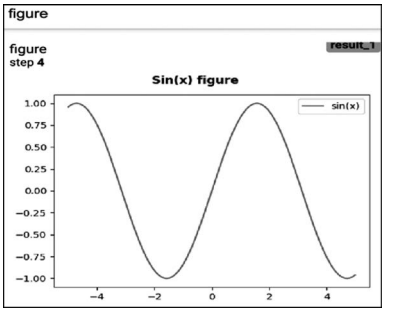


图 5-14　add_figure 可视化结果图

如果需要一次在 TensorBoard 中可视化一组图像，则可以通过以下方式进行实现：

```
1  def add_figures(writer, images, labels):
2      text_labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
3                     'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
4      labels = [text_labels[int(i)] for i in labels]
5      fit, ax = plt.subplots(len(images) //5, 5,
6                             figsize=(10, 2 * len(images) //5))
7      for i, axi in enumerate(ax.flat):
8          image, label = images[i].reshape([28, 28]).NumPy(), labels[i]
9          axi.imshow(image)
10         axi.set_title(label)
11         axi.set(xticks=[], yticks=[])
12     writer.add_figure("figures", fit)
```

在上述代码中，我们选择 FashionMNIST 数据集进行可视化。第 5～6 行代码用来生成一个包含若干子图的画布。第 7～11 行分别用来画出每幅子图，其中第 11 行用来去掉横纵坐标信息。第 12 行将其在 TensorBoard 中进行展示。最终可视化后的结果如图 5-15 所示。

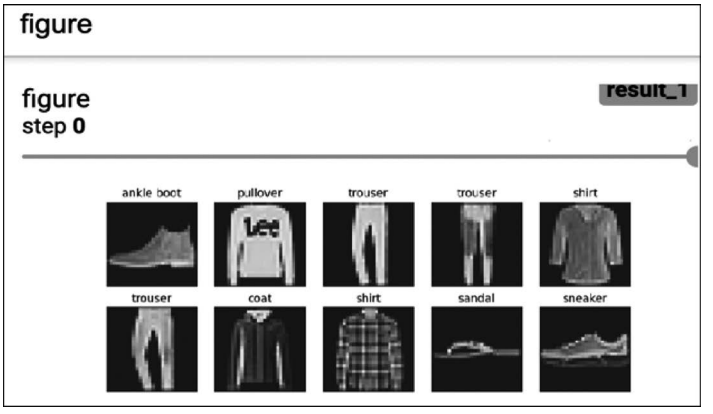


图 5-15　add_figure 可视化结果图

### 8. add_embedding 方法

这种方法的作用是在三维空间中对高维向量进行可视化，在默认情况下对高维向量以

PCA 方法进行降维处理。add_embedding()方法主要有 3 个比较重要的参数 mat、metadata 和 label_img，下面依次来进行介绍。

mat：用来指定可视化结果中每个点的坐标，形状为($N$,$D$)，不能为空，例如当对词向量可视化时 mat 就是词向量矩阵，当对图片分类时 mat 可以是分类层的输出结果；

metadata：用来指定每个点对应的标签信息，是一个包含 $N$ 个元素的字符串列表，如果为空，则默认为['1','2',…,'N']；

label_img：用来指定每个点对应的可视化信息，形状为($N$,$C$,$H$,$W$)，可以为空，例如当对图片分类时 label_img 就是每张真实图片的可视化结果。

进一步，可以通过如下代码来进行三维空间的高维向量可视化：

```
1  def add_embedding(writer):
2      import tensorflow as tf
3      import TensorBoard as tb
4      tf.io.gfile = tb.compat.tensorflow_stub.io.gfile
5      import keyword
6      meta = []
7      while len(meta) < 100:
8          meta = meta + keyword.kwlist
9      meta = meta[:100]
10     for i, v in enumerate(meta):
11         meta[i] = v + str(i)
12     label_img = torch.rand(100, 3, 10, 32)
13     for i in range(100):
14         label_img[i] *= i / 100.0
15     data_points = torch.randn(100, 5) #随机生成 100 个点
16     writer.add_embedding(mat = data_points, metadata = meta,
17                          label_img = label_img, global_step = 1)
```

在上述代码中，第 2～4 行用于解决 TensorFlow 1.x 版本的兼容性问题。第 6～11 行用于随机生成 100 个字符串标签信息。第 12～14 行用于生成标签对应的图片。第 15 行用于随机生成需要可视化的高维向量。上述代码运行结束后便会得到如图 5-16 所示的结果。

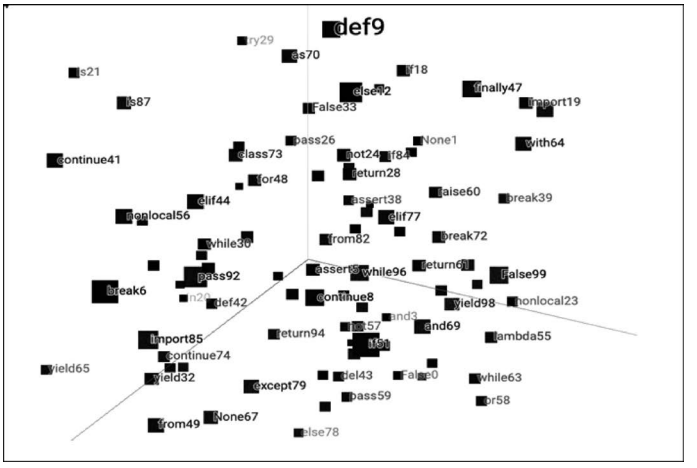


图 5-16　add_embedding 可视化结果图

如图 5-16 所示，字符串就是在上面的代码中对应的 metadata 参数，黑色方块就是对应的 label_img 参数，而方块背后的点（图中看不到）就是对应的 mat 参数。这里只是用了随机数据生成了上面这张图，在 5.2.4 节中将会用一个实际的例子来进行展示。

以上完整的示例代码可以参见 Code/Chapter05/C03_TensorBoardUsage/main.py 文件。

## 5.2.4　使用实例

本节直接使用在 4.4 节中介绍的 LeNet5 网络模型进行介绍。同时，在 4.4 节中对于整个模型的实现和训练部分的内容已经详细地进行了介绍，因此接下来的内容将只对可视化部分改动过的地方进行讲解，完整的示例代码可参见 Code/Chapter05/C03_TensorBoardUsage/main.py 文件。

**1. 载入数据集**

首先，需要构造训练模型时用到的数据集，这里将之前的 MNIST 数据集换成了 FashionMNIST，后者仅仅从 10 个数字变成了 10 种常见的衣物，其他数据（如图片大小、通道数等）均没有发生改变。FashionMNIST 数据集的载入方式如下：

```
1  from torchvision.datasets import FashionMNIST
2  text_labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
3                 'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
4
5  def load_dataset(batch_size=64):
6      mnist_train = FashionMNIST(root='~/Datasets/FashionMNIST',train=True,
7                                 download=True, transform=transforms.ToTensor())
8      mnist_test = FashionMNIST(root='~/Datasets/FashionMNIST',train=False,
9                                 download=True, transform=transforms.ToTensor())
10     train_iter = DataLoader(mnist_train, batch_size, shuffle=True)
11     test_iter = DataLoader(mnist_test, batch_size, shuffle=True)
12     return train_iter, test_iter
```

在上述代码中，第 2～3 行用于定义每个标签序号所对应的标签名，用于在使用 add_embedding 可视化预测结果时展示每个样本的标签名称。第 5～12 行则分别用于构造训练集和测试集的 DataLoader 对象。

**2. 初始化模型配置**

在 5.1 节内容中已经介绍过模型参数和日志打印这两部分内容，因此这里需要初始化相关模块，代码如下：

```
1  class ModelConfig(object):
2      def __init__(self,batch_size=64, epochs=3,learning_rate=0.01):
3          self.batch_size = batch_size
4          self.epochs = epochs
5          self.learning_rate = learning_rate
6          self.summary_writer_dir = "runs/LeNet5"
7          self.device = torch.device('cuda:0' if torch.cuda.is_available()
                                                else 'cpu')
8          logger_init(log_file_name='LeNet5', log_level=logging.INFO,
                       log_dir='log')
```

```
9           for key, value in self.__dict__.items():
10              logging.info(f"#{key} = {value}")
```

在上述代码中，第 2～5 行用于指定模型的相关超参数。第 6 行用于指定可视化文件的保存路径。第 7 行用于判断所使用的计算设备。第 8 行用于初始化日志打印模块。第 9～10 用于将模型参数输出到日志文件中。

**3. 定义评估方法**

由于模型在训练过程中需要返回预测结果相应的特征图，所以需要在之前 4.4 节实现的基础上添加一些返回值，代码如下：

```
1  def evaluate(data_iter, model, device):
2      model.eval()
3      all_logits, y_labels = [], []
4      images = []
5      with torch.no_grad():
6          acc_sum, n = 0.0, 0
7          for x, y in data_iter:
8              x, y = x.to(device), y.to(device)
9              logits = model(x)
10             acc_sum += (logits.argmax(1) == y).float().sum().item()
11             n += len(y)
12             all_logits.append(logits)
13             y_pred = logits.argmax(1).view(-1)
14             y_labels += (text_labels[i] for i in y_pred)
15             images.append(x)
16         all_logits, images = torch.cat(all_logits, dim=0),torch.cat(images, dim=0)
17         return acc_sum / n, all_logits, y_labels, images
```

在上述代码中，第 3～4 行里的 3 个变量分别用来保存预测值、真实文本标签和输入。第 7～11 行用于对每个小批量样本进行预测并计算相应的准确率。第 12～15 分别用来处理得到在使用 add_embedding 时所需要用到的变量。第 16 行是将所有预测值和输入值分别进行拼接。

**4. 定义训练过程**

由于这里新增了可视化部分的内容，所以需要在对之前 4.4 节实现的基础上增加部分内容，代码如下：

```
1  def train(config):
2      train_iter, test_iter = load_dataset(config.batch_size)
3      model = LeNet5()
4      optimizer = torch.optim.Adam([{"params": model.parameters(),
5                                     "initial_lr": config.learning_rate}])
6      num_training_steps = len(train_iter) * config.epochs
7      scheduler = get_cosine_schedule_with_warmup(optimizer,
                       num_warmup_steps=300,
8                      num_training_steps=num_training_steps,num_cycles=2)
9      writer = SummaryWriter(config.summary_writer_dir)
10     for epoch in range(config.epochs):
11         for i, (x, y) in enumerate(train_iter):
12             ......
```

```
            scheduler.step()
            if i % 50 == 0:
                acc = (logits.argmax(1) == y).float().mean()
                writer.add_scalar('Training/Accuracy', acc, scheduler.last_epoch)
                writer.add_scalar('Training/Loss', loss.item(), scheduler.last_epoch)
                writer.add_scalar('Training/Learning Rate',
                                  scheduler.get_last_lr()[0],
                                  scheduler.last_epoch)
        test_acc, all_logits, y_labels, label_img = evaluate(test_iter,
                                                    model, config.device)
        logging.info(f"Eps[{epoch+1}/{config.epochs}] Acc: {test_acc}")
        writer.add_scalar('Testing/Accuracy', test_acc,
                          scheduler.last_epoch)
        writer.add_embedding(mat=all_logits, metadata=y_labels,
                   label_img=label_img,global_step=scheduler.last_epoch)
    return model
```

在上述代码中，第 2 行用于得到训练集和测试集对应的迭代器。第 4～8 行分别用于定义优化器和学习率动态调整对象(这部分内容将在 6.1 节中进行介绍)，这里先学会如何用即可。第 13 行用于对学习率进行更新。第 16～18 行分别对训练集的准确率、损失值和学习率进行可视化。第 20 行则用于返回模型在测试集上预测的结果。第 22～24 行用于对模型在测试集上的准确率和预测结果进行可视化。

**5. 可视化展示**

在完成所有部分的编码工作后，便可以通过如下代码来运行整个模型：

```
if __name__ == '__main__':
    config = ModelConfig()
    model = train(config)
```

在程序运行开始后，便可以启动 TensorBoard 前端界面，此时能看到类似如图 5-17 所示的可视化结果。

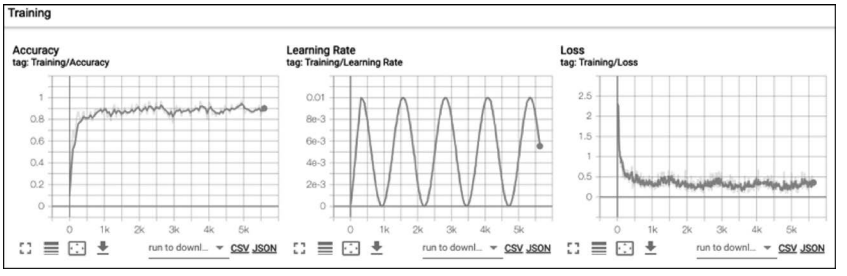


图 5-17　LeNet5 训练可视化结果图

如图 5-17 所示，这便是模型在训练过程中在训练集上的准确率、学习率和损失的变化结果。进一步可以展示出预测结果在空间中的分布情况，如图 5-18 所示。

如图 5-18 所示，这便是模型在测试集上的预测结果经过 add_embedding 方法可视化后的结果，其中每个小方块都表示一个原始样本，每种颜色代表一个类别。进一步，单击任意方块便可以查看该样本的相关信息，如图 5-19 所示。

图 5-18　LeNet5 模型预测标签可视化结果图(1)

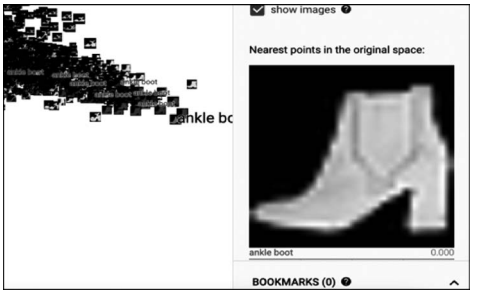


图 5-19　LeNet5 模型预测标签可视化结果图(2)

如图 5-19 所示,这便是 ankle boot 的可视化结果,并且可以发现只要单击其中的一个样本,与它类别相同的样本就会被标记出来。当然,该页面还有其他相应的功能,各位读者可以自行去探索,这里就不一一进行介绍了。

### 5.2.5　小结

本节首先详细介绍了如何在 PyTorch 框架下安装及启动 TensorBoard,包括远程连接和本地连接两种方式,然后详细介绍了 TensorBoard 中常用的 8 种可视化函数的使用方法及示例;最后以一个实际的 LeNet5 分类模型来展示了相关可视化函数的使用方法。

## 5.3　模型的保存与复用

在深度学习中通常训练一个可用的模型需要耗费极大的成本,因此在模型的训练过程中就需要对满足某些条件下的网络权重参数进行保存,然后在实际推理过程中直接载入这些权重参数来完成模型的推理过程。同时,另外一种场景便是模型已经在一批数据上训练完成且完成了本地持久化保存,但可能过了一段时间后又收集到了一批新的数据,因此这时就需要将之前的模型载入,以便在新数据上进行增量训练或者在整个数据上进行全量训练。

在 PyTorch 中可以通过 torch.save()和 torch.load()方法来完成上述场景中的主要步骤。下面将以之前介绍的 LeNet5 网络模型为例来分别进行介绍。不过在这之前先来看 PyTorch 中模型参数的保存形式。

### 5.3.1　查看模型参数

本节依旧以 4.4 节内容中介绍的 LeNet5 网络模型为例进行讲解。在定义完 LeNet5 网络模型并完成实例化操作后,网络中对应的权重参数也都完成了初始化的工作,即有了一个初始值。同时,可以通过以下代码来访问:

```
1  import sys
2  sys.path.append("../")
3  from Chapter04.C03_LeNet5.LeNet5 import LeNet5
4  if __name__ == '__main__':
5      model = LeNet5()
6      print("Model's state_dict:")
7      for (name, param) in model.state_dict().items():
8          print(name, param.size())
```

在上述代码中，第 1～2 行用于将 Chapter04 这个搜索路径加入系统路径中，否则第 3 行会提示 No module named 'Chapter04'。第 5 行用于实例化模型 LeNet5，即初始化整个模型。第 7～8 行用于遍历模型中的每个参数。同时，需要注意的是通过 model.state_dict()函数返回的是一个 Python 中的有序字段(OrderedDict)，即遍历输出的顺序就是元素插入字典时的顺序，例如这里插入的网络层。

上述代码运行结束后，其输出的结果如下：

```
1  Model's state_dict:
2  conv.0.weight torch.Size([6, 1, 5, 5])
3  conv.0.bias torch.Size([6])
4  conv.3.weight torch.Size([16, 6, 5, 5])
5  conv.3.bias torch.Size([16])
6  fc.1.weight torch.Size([120, 400])
7  fc.1.bias torch.Size([120])
8  fc.3.weight torch.Size([84, 120])
9  fc.3.bias torch.Size([84])
10 fc.5.weight torch.Size([10, 84])
11 fc.5.bias torch.Size([10])
```

在上述输出结果中，每行的前半部分表示参数的名称，如 conv.0.weight，后半部分表示该权重参数对应的形状。同时从输出结果可以看出，模型一共有 5 层权重参数，即 conv.0、conv.3、fc.1、fc.3 和 fc.5。

### 5.3.2　自定义参数前缀

在上面的输出结果中有两个地方值得注意：①参数名中的 fc 和 conv 前缀是根据定义 LeNet5 模型的 nn.Sequential()时的名字所确定的，即在 4.4.3 节中定模型时使用了两个 Sequential()实例对象，名称分别为 conv 和 fc；②参数名中的数字表示每个 Sequential()中网络层所在的位置。例如，如果将 LeNet5 网络结构定义成如下形式：

```
1  class LeNet5(nn.Module):
2      def __init__(self, ):
3          super(LeNet5, self).__init__()
4          self.LeNet5 = nn.Sequential(
5                  nn.Conv2d(1, 6, 5, padding=2), nn.ReLU(),
6                  nn.MaxPool2d(2, 2), nn.Conv2d(6, 16, 5), nn.ReLU(),
7                  nn.MaxPool2d(2, 2), nn.Flatten(),nn.Linear(16*5*5,120),
8                  nn.ReLU(),nn.Linear(120, 84),nn.ReLU(),nn.Linear(84, 10))
```

那么其参数名则为

```
print(model.state_dict().keys())
odict_keys(['LeNet5.0.weight', 'LeNet5.0.bias', 'LeNet5.3.weight',
 'LeNet5.3.bias', 'LeNet5.7.weight', 'LeNet5.7.bias', 'LeNet5.9.weight',
 'LeNet5.9.bias', 'LeNet5.11.weight', 'LeNet5.11.bias'])
```

可以看出，参数名最前面的部分就是 Sequential()对象的名字，理解了这一点对于后续解析和载入一些预训练模型很有帮助。

除此之外，对于 PyTorch 中的优化器等，其同样有对应的 state_dict()方法来获取相关参数信息，示例代码如下：

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
print(optimizer.state_dict())
{'state': {}, 'param_groups': [{'initial_lr': 0.01, 'lr': 0.0,
'betas': (0.9, 0.999), 'eps': 1e-08, 'weight_decay': 0, 'amsgrad': False,
'maximize':False,'foreach':None,'capturable':False,'differentiable':False,
'fused': False,'params': [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]}]}
```

在介绍完模型参数的查看方法后，便可以介绍模型复用阶段的内容了。上述完整的示例代码可参见 Code/Chapter05/C04_ModelSaving/E01_CheckParams.py 文件。

### 5.3.3 保存训练模型

在 PyTorch 中对于模型的保存来讲非常容易，通常来讲通过如下两行代码便可以实现：

```
model_save_path = os.path.join(model_save_dir, 'model.pt')
torch.save(model.state_dict(), model_save_path)
```

在指定保存的模型名称时 PyTorch 官方建议的后缀为.pt 或者.pth(当然也不强制)。最后，只需在合适的地方加入第 2 行代码便可保存模型[2]。

同时，如果想要在训练过程中保存某个条件下的最优模型，则应该通过以下方式实现：

```
from copy import deepcopy
best_model_state = deepcopy(model.state_dict())
torch.save(best_model_state, model_save_path)
```

而不是通过以下方式实现：

```
best_model_state = model.state_dict()
torch.save(best_model_state, model_save_path)
```

因为后者 best_model_state 得到的只是 model.state_dict()的引用，它依旧会随着训练过程的变化而发生改变。

### 5.3.4 复用模型推理

在推理复用模型的过程中，首先需要完成网络的初始化工作，然后载入已有的模型参数，以此来覆盖网络中的权重参数，代码如下：

```
def inference(config, test_iter):
    test_data = test_iter.dataset
    model = LeNet5()
```

```
    model.eval()
    if os.path.exists(config.model_save_path):
        checkpoint = torch.load(config.model_save_path)
        model.load_state_dict(checkpoint)
    else:
        raise ValueError(f"模型{config.model_save_path}不存在!")
    y_true = test_data.targets[:5]
    imgs = test_data.data[:5].unsqueeze(1).to(torch.float32)
    with torch.no_grad():
        logits = model(imgs)
    y_pred = logits.argmax(1)
    print(f"真实标签为{y_true}")
    print(f"预测标签为{y_pred}")
```

在上述代码中，第 1 行传入的是模型配置参数和测试文件，即并没有像之前那样将模型作为参数传递进来。第 3 行用于实例化一个模型，此时模型中的权重参数都是随机初始化的。第 4 行用于将模型的状态切换至推理状态。第 5～7 行用于校验本地指定路径中是否已经存在模型文件，如果存在，则载入并用其重新初始化网络模型。第 10～16 行的介绍见 4.4.3 节内容。

### 5.3.5　复用模型训练

在介绍完模型的保存与复用之后，模型的追加训练过程就很简单了。在网络训练之前，只需按照 5.3.4 节中的方法重新初始化网络权重参数，然后按照正常的步骤训练模型即可，关键的示例代码如下：

```
def train(config):
    model = LeNet5()
    if os.path.exists(config.model_save_path):
        checkpoint = torch.load(config.model_save_path)
        model.load_state_dict(checkpoint)
    num_training_steps = len(train_iter) * config.epochs
    ...
    for epoch in range(config.epochs):
        for i, (x, y) in enumerate(train_iter):
            loss, logits = model(x, y)
            loss.backward()
            optimizer.step()     #执行梯度下降
            ...
        if test_acc > max_test_acc:
            max_test_acc = test_acc
            state_dict = deepcopy(model.state_dict())
            torch.save(state_dict, config.model_save_path)
    return model
```

在上述代码中，第 3～5 行用于判断本地是否有模型权重，如果有，则载入后重新初始化网络。第 14～17 行根据测试集上最大准确率的条件来将当前时刻的模型保存到本地，这样便完成了模型的追加训练。

最后，运行上述程序后便可以看到类似如下的结果输出：

```
# 载入模型 model.pt 进行追加训练...
Epochs[1/3] -- batch[0/938] -- Acc: 0.9219 -- loss: 0.322
Epochs[1/3] -- batch[50/938] -- Acc: 0.875 -- loss: 0.3906
Epochs[1/3] -- batch[100/938] -- Acc: 0.8906 -- loss: 0.3293
Epochs[1/3] -- batch[150/938] -- Acc: 0.9219 -- loss: 0.3178
......
```

从上述输出结果也可以看出，模型在追加训练时第 1 个批量样本上的准确率就已经达到了 0.922 左右。

除此之外也可以在保存参数时，将优化器参数、损失值等一同保存下来，然后在恢复模型时连同其他参数一起恢复，示例代码如下：

```
model_save_path = os.path.join(model_save_dir, 'model.pt')
torch.save({'epoch': epoch,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'loss': loss}, model_save_path)
```

载入方式如下：

```
checkpoint = torch.load(model_save_path)
model.load_state_dict(checkpoint['model_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
epoch = checkpoint['epoch']
loss = checkpoint['loss']
```

上述完整的示例代码可参见 Code/Chapter05/C04_ModelSaving/train.py 文件。

### 5.3.6 小结

本节首先介绍了模型复用的两种典型场景，然后介绍了如何查看 PyTorch 模型中的相关参数信息及自定义参数名前缀；最后详细介绍了如何保存模型、加载本地模型进行推理及追加训练等。在 5.4 节内容中，将会详细介绍如何载入本地模型进行迁移学习。

## 5.4 模型的迁移学习

前面几节内容详细介绍了 PyTorch 中模型的保存及载入推理和复用等过程。在有了前期这些基础知识后，接下来介绍关于模型迁移学习(Transfer Learning)部分的内容。

### 5.4.1 迁移学习

在深度神经网络中由于模型通常含有大量的可学习参数，所以在训练数据不充分的情况下模型极易出现过拟合或者泛化能力差的情况。另外，数据样本的标注又是一项既耗费时间又耗费财力的工作[6]，尤其是在一些需要业务专家介入的复杂任务标注中，因此，如何利用有限的数据来训练模型便成为热门的研究方向。受到人类学习的启发——人类在学习并解决一个新问题时，总是可以依赖先前所拥有的经验并迅速迁移到当前的场景中——研

究人员开始提出一种两段式的学习框架，即先在一个通用的大规模数据集上训练一个预训练模型(Pre-trained Model)，然后针对特定的任务场景再根据少量的标注数据对整个模型进行微调(Fine-tuning)，而这也被称为迁移学习。

在深度学习中迁移学习主要起源于图像处理领域，其背后的理念是如果一个模型是基于足够大且通用的数据集所训练的，则该模型将可以有效地充当视觉领域的通用模型，随后便可以直接将这些学习到的模型参数迁移到下游任务中而不必再从头开始训练整个模型[3]。

在图像处理领域中 ImageNet 是一个非常著名的大型通用数据集，它是由李飞飞团队于 2007 年所发起构建的一个项目，包含超过 1400 万张手动标注的图片，旨在为世界各地的研究人员提供用于训练大规模物体识别模型的图像数据[4]。自 2010 年以来，ImageNet 项目每年都举办一次大规模视觉识别挑战赛，挑战赛使用 1000 个类别的图片，用于正确分类和检测目标及场景[5]，如图 5-20 所示，这便是 ImageNet 数据集中的部分图像。

图 5-20　ImageNet 数据集示例图

根据 4.2.3 节内容可知，越靠近输出层其特征越抽象，越靠近输入层其特征越具体，因此假如现在有一个开源的图片分类模型 A，此模型是基于 ImageNet 数据集训练而来的，如果在某任务场景中需要训练另外一个 10 分类模型 B，用于汽车型号的分类，则可以直接取模型 A 中的前若干层(靠近输入层)网络作为特征提取器，然后在此基础上再加入一个新的全连接分类层来构造，从而得到模型 B，并以此完成整个 10 分类任务，此时将模型 A 称为预训练(Pre-trained)模型。同时，通常来讲还可以根据是否让预训练模型中的参数参与整个模型的训练这两种方式来完成模型的迁移学习任务[6]。

在接下来的内容中，我们将会通过一个实际的示例来对模型的迁移学习过程进行介绍。以下完整的示例代码可参见 Code/Chapter05/C05_ModelTrans/文件夹。

### 5.4.2 模型定义与比较

4.4 节详细介绍了 LeNet5 网络模型的原理及现实过程，并且同时根据 5.3 节的介绍也清楚了模型的保存与复用。现在假设有一个 LeNet6 网络模型，它是在 LeNet5 的基础上增加了一个全连接层，此时便可以通过迁移学习将 LeNet5 模型中的部分参数用于 LeNet6 模型中。具体地，LeNet6 模型结构的实现代码如下：

```
1  class LeNet6(nn.Module):
2      def __init__(self, ):
3          super(LeNet6, self).__init__()
4          self.conv = nn.Sequential(
5              nn.Conv2d(in_channels = 1, out_channels = 6,
6                        kernel_size = 5, padding = 2),
7              nn.ReLU(), nn.MaxPool2d(2, 2), nn.Conv2d(6, 16, 5),
8              nn.ReLU(),nn.MaxPool2d(2, 2))
9          self.fc = nn.Sequential(
10             nn.Flatten(),nn.Linear(16 * 5 * 5, 120),
11             nn.ReLU(),nn.Linear(120, 84),nn.ReLU(),
12             nn.Linear(84, 64),nn.ReLU(),nn.Linear(64, 10))
```

在上述代码中，第 1～11 行是 LeNet5 模型的前 4 层。第 12 行便是 LeNet6 模型中新加入的一个网络层。

在模型定义结束后，便可以输出模型中对应的参数信息。同时，为了完成后续模型的迁移过程，这里也将 LeNet5 保存在本地的权重参数载入并进行输出，以便两者进行对比，代码如下：

```
1  if __name__ == '__main__':
2      print("\n===== Model paras in LeNet6:")
3      model = LeNet6()
4      for (name, param) in model.state_dict().items():
5          print(name, param.size())
6
7      model_save_path = os.path.join('../C04_ModelSaving', 'LeNet5.pt')
8      print("\n===== Model paras in LeNet5:")
9      loaded_paras = torch.load(model_save_path)
10     for (name, param) in loaded_paras.items():
11         print(name, param.size())
```

在上述代码中，第 2～5 行用于输出 LeNet6 模型中各个权重参数的名称和形状信息。第 7～11 行用于载入 5.3 节中持久化保存到本地的 LeNet5 权重参数，并同时输出每个参数的名称和形状。

在上述代码运行结束后便可以得到如下的结果：

```
1  ===== Model paras in LeNet6:
2  conv.0.weight torch.Size([6, 1, 5, 5])
3  conv.0.bias torch.Size([6])
```

```
conv.3.weight torch.Size([16, 6, 5, 5])
conv.3.bias torch.Size([16])
fc.1.weight torch.Size([120, 400])
fc.1.bias torch.Size([120])
fc.3.weight torch.Size([84, 120])
fc.3.bias torch.Size([84])
fc.5.weight torch.Size([64, 84])
fc.5.bias torch.Size([64])
fc.7.weight torch.Size([10, 64])
fc.7.bias torch.Size([10])
 ===== Model paras in LeNet5:
conv.0.weight torch.Size([6, 1, 5, 5])
conv.0.bias torch.Size([6])
conv.3.weight torch.Size([16, 6, 5, 5])
conv.3.bias torch.Size([16])
fc.1.weight torch.Size([120, 400])
fc.1.bias torch.Size([120])
fc.3.weight torch.Size([84, 120])
fc.3.bias torch.Size([84])
fc.5.weight torch.Size([10, 84])
fc.5.bias torch.Size([10])
```

在上述结果中，第 1～13 行和第 14～24 行分别为两个模型的参数输出信息，其中第 2～9 行与第 15～22 行则是两个模型对应的相同部分(可复用)，区别在于前者是随机初始化的权重参数而后者是训练得到的权重参数。第 10～13 行便是 LeNet6 模型中所改动的部分。

在清楚了新旧模型的参数信息后，下面便可将 LeNet5 模型中需要的参数取出来并迁移到 LeNet6 模型中。

### 5.4.3　参数微调

在迁移学习中，最直观的一种方式就是让所有迁移过来的参数一同参与到整个模型的训练过程中，即参数的微调(Fine Tuning)，然后将训练完成的整个参数保存到本地，用于后续的推理过程。在对模型参数进行微调前，首先需要在类 LeNet6 中实现一种方法，以此来对 LeNet5 中的权重参数进行解析并将其用于 LeNet6 模型部分参数的初始化，代码如下：

```
@classmethod
def from_pretrained(cls, pretrained_model_dir = None):
    model = cls()
    pretrained_model_path = os.path.join(pretrained_model_dir,"LeNet5.pt")
    if not os.path.exists(pretrained_model_path):
        raise ValueError(f"< {pretrained_model_path} 中的模型不存在!>")
    loaded_paras = torch.load(pretrained_model_path)
    state_dict = deepcopy(model.state_dict())
    for key in state_dict:
        if key in loaded_paras and
            state_dict[key].size() == loaded_paras[key].size():
            logging.info(f"成功初始化参数: {key}")
            state_dict[key] = loaded_paras[key]
    model.load_state_dict(state_dict)
    return model
```

在上述代码中，第 2 行 pretrained_model_dir 用来指定预训练模型所在的目录。第 3 行用于实例化 LeNet6 模型。第 4～6 行用于构造预训练模型的路径并判断是否存在。第 7～8 行分别用于载入预训练模型和深度复制一份 LeNet6 模型中的参数，之所以称为深度复制是因为 model.state_dict()返回的是一个引用，无法直接修改里面的权重参数。第 9～13 行用于在 LeNet6 网络模型中遍历每个参数，并根据参数名和参数形状来判断 LeNet5 模型中是否有相同的参数，如果有，则对 LeNet6 网络模型中的参数进行替换。第 14～15 行用于对 LeNet6 中的部分权重参数进行重新初始化并返回。

这里值得一提的是，对于不同的迁移场景，第 10～11 行的判断条件并不一致，需要根据 5.4.2 节中的介绍进行分析。

在完成上述代码之后，便可以通过以下方式进行载入，并输出部分结果，以便进行对比，代码如下：

```
1  if __name__ == '__main__':
2      model_save_path = os.path.join('../C04_ModelSaving', 'LeNet5.pt')
3      print("\n===== Model paras in LeNet5:")
4      loaded_paras = torch.load(model_save_path)
5      print(f"LeNet5 模型中第 1 层权重参数(部分)为
6              {loaded_paras['conv.0.weight'][0, 0]}")
7      print("\n===== Load model from pretrained ")
8      model = LeNet6.from_pretrained('../C04_ModelSaving')
9      print(f"LeNet6 模型中第 1 层权重参数(部分)为
10           {model.state_dict()['conv.0.weight'][0, 0]}")
```

在上述代码中，第 2～6 用于载入本地 LeNet5 模型对应的参数并输出第 1 个卷积层对应的部分参数。第 7～10 行根据上面所实现的 from_pretrained 方法来完成权重参数的迁移过程。

在上述代码运行结束后，便可以看到类似如下的验证结果：

```
1  LeNet5 模型中第 1 层权重参数(部分)为
2  tensor([[-0.0538, -0.4352, 0.2128, -0.0808, 0.0599],
3          [ 0.1359, -0.4566, 0.0987, 0.1395, -0.0719],
4          [-0.1107, -0.2895, 0.3242, 0.3209, 0.1349],
5          [ 0.2209, -0.2949, 0.2101, 0.0179, 0.0596],
6          [-0.0431, -0.2913, -0.0029, 0.1416, 0.0864]])
7  LeNet6 模型中第 1 层权重参数(部分)为
8  tensor([[-0.0538, -0.4352, 0.2128, -0.0808, 0.0599],
9          [ 0.1359, -0.4566, 0.0987, 0.1395, -0.0719],
10         [-0.1107, -0.2895, 0.3242, 0.3209, 0.1349],
11         [ 0.2209, -0.2949, 0.2101, 0.0179, 0.0596],
12         [-0.0431, -0.2913, -0.0029, 0.1416, 0.0864]])
```

从上述输出结果可以看出，LeNet6 模型中第 1 个卷积层的权重参数已经变成了 LeNet5 中对应部分的参数。

最后，在训练初始化 LeNet6 模型时，只需像上面那样用 from_pretrained 方法来完成参数的迁移，其他部分的代码并没有发生任何改变。

虚线和实线分别表示是否将 LeNet5 模型中的参数迁移到 LeNet6 中，如图 5-21 所示。

从图中可以发现，在大约前 50 次小批量样本迭代过程中进行参数迁移的模型损失减小速度要明显快于没有进行迁移的模型，不过由于整个 LeNet6 模型比较小，所以大约在 100 次迭代后两者的损失变换便趋同了。

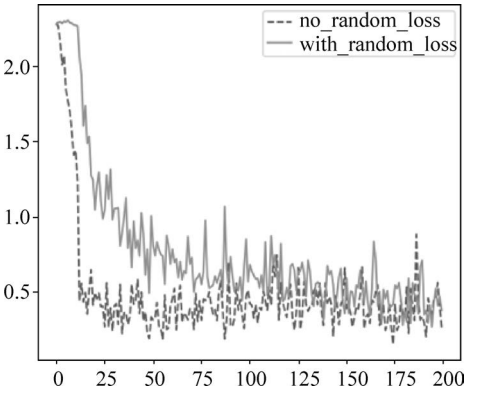


图 5-21　参数迁移与随机初始化损失对比图

### 5.4.4　参数冻结

除了将其他模型迁移过来的参数一同加入新模型中进行训练微调之外，还有一种做法就是对迁移部分的参数进行冻结(固定不变)，即不参与整个模型的训练过程，而仅仅将它作为一个固定的特征提取器。之所以选择这样做的一个主要原因是当被迁移过来的权重参数规模过大时，将会十分耗费整个模型的训练时间及可能陷入过拟合的状态。

为了使迁移过来的权重参数不参与整个模型的训练，只需在重新初始化模型参数时将需要冻结的参数的 requires_grad 属性设置为 False，即在训练过程中不再更新梯度，具体新增部分的代码如下：

```
@classmethod
def from_pretrained(cls, pretrained_model_dir = None, freeze = False):
    model = cls()
    frozen_list = []
    # ... 载入本地参数等
    for key in state_dict:
        if key in loaded_paras and
            state_dict[key].size() == loaded_paras[key].size():
            logging.info(f"成功初始化参数: {key}")
            state_dict[key] = loaded_paras[key]
            if freeze:
                frozen_list.append(key)
    if len(frozen_list) > 0:
        for (name, param) in model.named_parameters():
            if name in frozen_list:
                logging.info(f"冻结参数{name}")
                param.requires_grad = False
    model.load_state_dict(state_dict)
    return model
```

在上述代码中，第 1～10 行已经介绍过，此处就不再赘述了。第 11～12 行用于判断是否需要对参数进行冻结，并保存参数名。第 13～17 行用于先遍历模型中的每个参数，然后将需要冻结的参数的 requires_grad 属性设置为 False。最后，只需在通过 from_pretrained 方法对模型进行初始化时传入参数 freeze＝True 便可不让迁移部分的参数参与训练。

同时，在模型的训练过程中还可以在每个小批量迭代过程中通过以下一行代码来验证参数是否发生了改变：

```
print(f"第 1 层权重(部分)为{model.state_dict()['conv.0.weight'][0, 0]}")
```

当然在实际情况中也可以根据相应的判断条件来对需要的参数进行冻结。

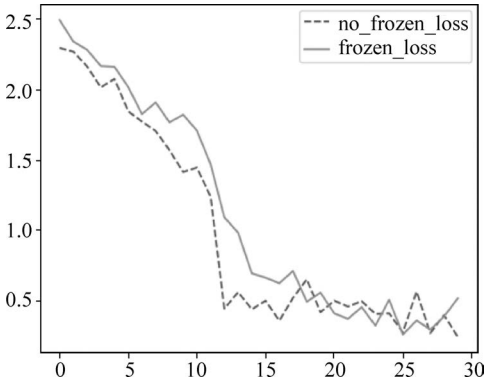

图 5-22　迁移参数是否冻结损失对比图

实线和虚线分别表示是否对迁移参数进行冻结，如图 5-22 所示。可以看出，在大约前 15 个小批量的迭代过程中，不进行参数冻结的模型在损失降低速度上会略快于进行参数冻结的模型，并且在大约 20 个小批量迭代后两者的变化速度趋同。当然，如果迁移部分的权重规模较大，则这两者将会有更加明显的区别。

总体来讲，对于是否应该让迁移部分的模型参数参与整个网络的训练过程大致可以分为以下 4 种情况[6]。

（1）当新场景中的数据集规模小于且类似于预训练模型中的数据集时不建议对迁移部分参数进行微调，因为此时新数据集规模较小，微调整个模型容易出现过拟合现象，所以更好的做法是将迁移部分的网络作为一个初步的特征抽取器，然后直接训练一个线性分类器，以此来完成后续任务。

（2）当新场景中的数据集规模大于且类似于预训练模型中的数据集时可以对迁移部分参数进行微调，因为此时拥有更大规模的相似数据集可以用来调整模型参数，并且也不易出现过拟合现象。

（3）当新场景中的数据集规模小于且不同于预训练模型中的数据集时不建议对迁移部分参数进行微调，因为新数据不同于源数据集，并且可能包含该数据特有的特征结构，所以更好的做法是将迁移部分的网络作为一个特征抽取器，然后构建一个简单的网络来完成后续任务。

（4）当新场景中的数据集规模大于且不同于预训练模型中的数据集时可以对迁移部分参数进行微调，因为此时拥有更大规模的数据集支持微调整个模型，并且通过迁移部分的参数来初始化新模型也有利于训练一个更好的模型参数。

### 5.4.5　小结

本节首先介绍了迁移学习的基本概念及其背后的思想，然后介绍了如何通过对比来分析预训练模型中参数结构和新模型中参数结构的差异，以此来实现参数的迁移过程；接着

进一步介绍了两种常见的模型参数迁移方式，即迁移部分的参数是否参与整个模型的微调过程；最后详细介绍了如何通过代码实现模型的迁移过程，并总结了是否让迁移参数参与模型微调的 4 种情况。

## 5.5　开源模型复用

在前面两节内容中我们陆续介绍了在 PyTorch 框架中模型保存和迁移的基本原理，在接下来的内容中将以 ResNet18 在 ImageNet 上训练得到的 1000 分类预训练模型为例，将其迁移到 CIFAR10 数据集上进行微调。总体上来讲，首先需要实例化一个 ResNet 模型，再用预训练模型对其初始化，然后将原始 ResNet 中的最后一个 1000 分类的分类层改为 CIFAR10 数据对应的 10 分类层；最后在 CIFAR10 数据集上完成整个模型的微调。以下完整的示例代码可以参见 Code/Chapter05/C06_PretrainedModel/文件。

### 5.5.1　ResNet 结构介绍

为了方便使用 PyTorch 官方开源的预训练模型，下面直接使用 PyTorch 框架中 ResNet 模型。同时，为了便于后续理解模型迁移，这里先简单介绍 PyTorch 中 ResNet 实现部分的代码。在 PyTorch 框架中，可以通过以下两行代码来实例化一个残差网络，以 ResNet18 为例，示例代码如下：

```
1  from torchvision.models import resnet18
2  model = resnet18()
```

其中函数 ResNet18 的实现过程如下：

```
1  def resnet18( * , weights = None, ):
2      weights = ResNet18_Weights.verify(weights)
3      return _resnet(BasicBlock, [2, 2, 2, 2], weights,...)
```

在上述代码中，第 2 行用于验证传入的预训练模型是否合法。第 3 行则根据残差结构的数量返回 ResNet18 模型。

ResNet 函数中的核心部分如下：

```
model = ResNet(block, layers, **kwargs)
```

在上述代码中返回的便是一个残差网络的实例化对象，而类 ResNet 中的网络结构的定义过程如下：

```
1  class ResNet(nn.Module):
2      def __init__(self,...):
3          super().__init__()
4          ...
5          self.layer1 = self._make_layer(block, 64, layers[0])
6          self.layer2 = self._make_layer(block, 128, layers[1], stride = 2)
7          self.layer3 = self._make_layer(block, 256, layers[2], stride = 2)
8          self.layer4 = self._make_layer(block, 512, layers[3], stride = 2)
```

```
9           self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
10          self.fc = nn.Linear(512 * block.expansion, num_classes)
```

在上述代码中，第5～9行便是相应的残差结构和全局平均池化层。第10行对应于最后的分类层，而有序将ResNet18迁移到CIFAR10数据集上需要修改的便是最后一个分类层。

### 5.5.2 迁移模型构造

在清楚了PyTorch中ResNet模型的基本实现结构之后，便可以对其进行修改以适应CIFAR10数据集，代码如下：

```
1  from torchvision.models import resnet18
2  from torchvision.models import ResNet18_Weights
3
4  class ResNet18(nn.Module):
5      def __init__(self, num_classes = 10, frozen = False):
6          super(ResNet18, self).__init__()
7          self.resnet18 = resnet18(weights = ResNet18_Weights.IMAGENET1K_V1)
8          if frozen:
9              for (name, param) in self.resnet18.named_parameters():
10                 param.requires_grad = False
11                 logging.info(f"冻结参数: {name}, {param.shape}")
12         self.resnet18.fc = nn.Linear(512, num_classes)
```

在上述代码中，第7行用于返回一个实例化的18层残差网络，同时指定了需要通过预训练模型来对其进行初始化。第8～11行用来判断是否需要对预训练部分的参数进行冻结，即不参与后续模型的训练过程，当然也可根据需要修改为对其中一部分参数进行冻结。第12行用于将原始残差网络的最后一层替换为符合新数据集的分类层。

其对应的前向传播的实现过程如下：

```
1  def forward(self, x, labels = None):
2      logits = self.resnet18(x)
3      if labels is not None:
4          loss_fct = nn.CrossEntropyLoss(reduction = 'mean')
5          loss = loss_fct(logits, labels)
6          return loss, logits
7      else:
8          return logits
```

然后可通过以下方式打印网络结构信息：

```
1  if __name__ == '__main__':
2      model = ResNet18(frozen = True)
3      x = torch.rand(1, 3, 96, 96)
4      out = model(x)
5      print(out)
6      for (name, param) in model.named_parameters():
7          print(f"name = {name,param.shape}
                   requires_grad = {param.requires_grad}")
```

在上述代码中，第2行用于实例化一个残差网络并且冻结相关的预训练参数。第5行用于输出前向传播最后的结果。第6～7行用于查看模型中的权重参数是否被冻结。

```
1  冻结参数: conv1.weight, torch.Size([64, 3, 7, 7])
2  冻结参数: bn1.weight, torch.Size([64])
3  冻结参数: bn1.bias, torch.Size([64])
4  ...
5  冻结参数: layer4.1.bn2.weight, torch.Size([512])
6  冻结参数: layer4.1.bn2.bias, torch.Size([512])
7  冻结参数: fc.weight, torch.Size([1000, 512])
8  冻结参数: fc.bias, torch.Size([1000])
9  tensor([[ -1.380,  -0.227, 0.492, 0.605, 1.078, 0.049, 1.057, 0.451,
10          0.2397,  -0.2712]], grad_fn = <AddmmBackward0>)
11 ...
12 name = ('resnet18.layer4.1.bn2.weight', torch.Size([512]))
           requires_grad = False
13 name = ('resnet18.layer4.1.bn2.bias', torch.Size([512]))
           requires_grad = False
14 name = ('resnet18.fc.weight', torch.Size([10, 512])) requires_grad = True
15 name = ('resnet18.fc.bias', torch.Size([10])) requires_grad = True
```

在上述输出结果中，第1～8行为原始ResNet18的参数信息，并且均已经被冻结。第9～15行为迁移后残差网络的相关输出信息，其中第10～11行用于前向传播输出结果，第11～15行是各层权重的名称、形状及是否被冻结等信息，从这里可以看出除了最后两层之外其余层的参数均不参与训练，并且最后一个分类层已经变成了10分类。

到此，对于迁移模型的网络结构实现就介绍完了，整个网络训练代码与4.9节中的代码相同，这里就不再赘述了，各位读者直接阅读代码即可。

### 5.5.3　结果对比

在完成模型的训练过程后，可以将原始ResNet18模型、迁移冻结后的ResNet18模型及进行微调的ResNet18模型这三者在CIFAR10上的结果进行一个简单的对比，如表5-1所示。

**表5-1　模型分类准确率对比**

| 模型名称 | 迭代轮数 | | | | |
|---|---|---|---|---|---|
| | 1轮 | 5轮 | 10轮 | 30轮 | 50轮 |
| ResNet18 | 0.6042 | 0.7869 | 0.8102 | 0.8393 | 0.8634 |
| ResNet18(冻结) | 0.7283 | 0.7461 | 0.7504 | 0.7496 | 0.7505 |
| ResNet18(微调) | 0.7589 | 0.8374 | 0.893 | 0.8984 | 0.9093 |

从表5-1中的结果可以看出，如果整个网络模型的权重都随机初始化，则虽然第1轮迭代结束后它在测试集上的准确率最差，但是随后却超越了冻结整个预训练参数只有分类层参与训练的模型。同时，在这3种情况中，对预训练模型一同进行微调时的效果最好，经过50轮迭代之后在测试集上的准确率达到了90%以上。

### 5.5.4　小结

本节首先介绍了PyTorch框架中ResNet残差网络的基本实现逻辑，然后详细介绍了如何基于预训练模型来完成ResNet18的迁移任务并对相关输出结果进行了分析；最后，对比了3种不同初始化方法或训练策略的残差模型在CIFAR10数据集上的分类准确率。

## 5.6 多 GPU 训练

在深度学习中一些大型的网络模型往往需要大量的计算资源才能进行训练，因为每层神经网络都需要对输入数据进行复杂的矩阵乘法和非线性变换操作。由于单个 GPU 的计算能力及显存有限，所以可能无法满足大规模深度神经网络的训练需要，因此需要使用多个 GPU 来加速网络的训练速度。在接下来的内容中，将会简单介绍几种多 GPU 模型训练的基本思想，并就其中一种最常见的方法进行详细讲解。

### 5.6.1 训练方式

从理论上来讲，实现模型多 GPU 训练的策略有模型并行、数据并行和混合并行 3 种，然而在实际情况中并不是每种都具有较高的可行性。

(1) 模型并行：将模型的不同层分配到不同的 GPU 上进行训练，每个 GPU 只处理部分层的计算，并将计算后的结果传递给下一个 GPU 进行处理。同时，在模型并行中每个 GPU 上的模型权重可能并不相同，但每个 GPU 的输入数据却都相同，因此不同 GPU 之间需要相互传递数据以进行计算。通常这种方法适用于模型较大且无法在单个 GPU 上容纳的情况，但是其存在需要更多的硬件资源、实现难度较大、通信开销较大等问题，所以实际使用较少。

(2) 数据并行：将输入网络的训练数据分成多个批次，每个批次在不同的 GPU 上进行并行计算。此时每个 GPU 上的模型权重都相同，只是处理的数据不同，每个 GPU 在训练完自己的批次数据后再将梯度更新汇总到主 GPU 上，从而实现模型参数的更新。这种方法的优点是简单、易实现、不容易出错，因此也是实现多 GPU 训练中使用最多的一种策略。

(3) 混合并行：一种同时使用数据并行和模型并行的技术。在混合并行中网络模型将会被拆分为多个子模型，并将每个子模型分配到不同的 GPU 上进行计算，然后将计算好的结果传递给下一个 GPU 进行处理，同时在每个 GPU 中也将使用数据并行技术进行处理。混合并行的优点在于它可以同时利用数据并行和模型并行的优势，因为数据并行可以处理大规模数据集，而模型并行可以扩展深度神经网络的规模，但混合并行也存在一些挑战，例如需要更多的硬件资源、实现难度较大、调试和优化复杂等。

以上便是 3 种策略的基本思想，但是需要注意的是多 GPU 并不是越多越好，过多数量的 GPU 可能会造成通信延迟和资源浪费，并极有可能出现多个 GPU 的训练速度反而比单个 GPU 更慢的情况。在实际使用中，需要根据具体的硬件条件和数据规模选择合适的多 GPU 训练策略。

下面对最常见的数据并行策略进行详细介绍。

### 5.6.2 数据并行

在使用数据并行策略实现多 GPU 训练时，首先会将整个小批量数据划分成多个小批次并分配到不同的 GPU 上，同时整个模型也将被复制到每个 GPU 上，然后在每个 GPU 上模型均各自独立地完成损失和梯度的计算，随后将每个 GPU 上计算得到的损失和梯度汇

聚到主 GPU 上，从而得到整个小批量数据样本的平均梯度，最后将该梯度分配到其他 GPU 中对各自模型参数进行更新以完成一次迭代训练过程[7]。

含有两个 GPU 的数据并行原理图如图 5-23 所示。例如此时每个小批量数据都含有 256 个样本，那么图示中每个 GPU 将会被分配 128 个样本进行后续的计算处理。同时，每个 GPU 上也都有着一模一样的网络模型，并且它们在各自获得 128 个样本后会分别计算损失和梯度，然后将两部分的梯度汇聚到主 GPU 上，从而得到 256 个样本的平均梯度，最后用该梯度通过梯度下降算法并行对每个 GPU 上的模型进行参数更新。

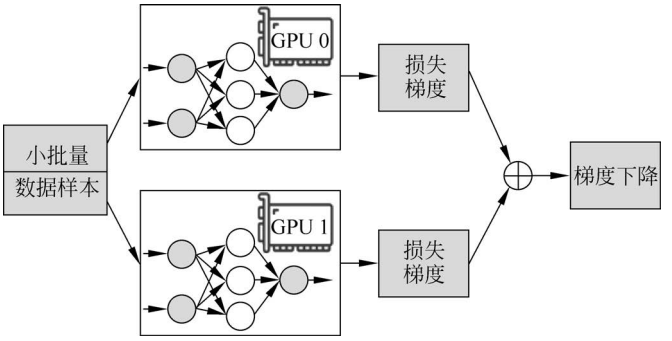


图 5-23　数据并行原理图

由此可以发现，对于数据并行这一多 GPU 训练策略来讲，本质上就相当于每个 GPU 各自完成了部分数据样本的训练过程，并且在整个前向传播和反向传播中每个 GPU 之间均是相互独立的，只有在进行整体损失和梯度的计算时才进行交互，因此基于数据并行的多 GPU 训练方法相对较容易实现，但在实践中该方法需要权衡计算资源、通信开销和同步效率等因素。

## 5.6.3　使用示例

本节以 4.9 节中介绍的 ResNet18 为例来介绍如何通过 PyTorch 框架实现网络模型的多 GPU 训练过程[8]。在这里首先需要清楚的是，对于是否使用多 GPU 进行模型训练与模型的定义与前向传播过程无关，也就是只需修改模型训练部分的代码。以下完整的示例代码可参见 Code/Chapter05/C07_MultiGPUs/train.py 文件。

### 1. 获取 GPU

首先，需要定义一个辅助函数来获取指定的 GPU 设备，代码如下：

```
1  def get_gpus(num = None):
2      gpu_nums = torch.cuda.device_count()
3      if isinstance(num, list):
4          devices = [torch.device(f'cuda:{i}')for i in num if i < gpu_nums]
5      else:
6          devices = [torch.device(f'cuda:{i}')for i in range(gpu_nums)][:num]
7      return devices if devices else [torch.device('cpu')]
```

在上述代码中，第 1 行 num 如果为 list，则返回 list 中对应编号的 GPU 设备；如果 num 为整数，则返回主机中前 num 个 GPU 设备。第 2 行用于得到当前主机上的 GPU 设备的个数。

第 3～4 行根据 num 为 list 的情况获取对应的 GPU 设备。第 5～6 行根据 num 为整数的情况获取对应的 GPU 设备。第 7 行则用于判断是否有 GPU 设备，如果没有，则返回 CPU 设备。

上述代码运行结束后的结果如下：

```
[device(type='cpu')] #无 GPU 时的情况
[device(type='gpu', index=0), device(type='gpu', index=1)] #有两块 GPU 设备
```

**2. 数据并行**

在得到相应的 GPU 设备之后便需要在训练代码中完成数据并行及相应代码的修改，其中修改处的代码如下：

```
def train(config):
    ...
    model = model.to(config.device[config.master_gpu_id]) #指定主 GPU
    model = nn.DataParallel(model, device_ids=config.device)
    for epoch in range(config.epochs):
        for i, (x, y) in enumerate(train_iter):
            x = x.to(config.device[config.master_gpu_id])
            y = y.to(config.device[config.master_gpu_id])
            loss, logits = model(x, y)
            loss.mean().backward()
            optimizer.step() #执行梯度下降
            if i % 50 == 0:
                acc = (logits.argmax(1) == y).float().mean()
                logging.info(f"Epochs[{epoch + 1}/{config.epochs}]—
                             batch[{i}/{len(train_iter)}]"
                            f"--Acc: {round(acc.item(), 4)}--loss:
                                    {round(loss.sum().item(), 4)}")
```

在上述代码中，第 3 行表示将模型放到指定的主 GPU 上，因为后续需要根据主 GPU 来完成每个 GPU 设备上计算得到的损失和梯度的汇聚。第 4 行表示 PyTorch 中实现数据并行的方式。第 7～10 行用于指定在主 GPU 设备上完成损失和梯度的汇聚，其中这里需要注意的是由于每个 GPU 设备上都会通过计算得到一个损失值，因此在第 10 行中需要指定为所有损失的均值(或总和)，以此来计算各个权重参数的梯度。第 15 行在输出模型的整体训练损失时需要指定为 loss.sum()或 loss.mean()的形式。

另一点需要注意的是，在使用多 GPU 进行模型训练时，小批量样本的数量一定要大于 GPU 设备的数量，不然无法使用多 GPU 进行训练。

**3. 模型训练**

在完成上述代码之后便可以开始训练模型了，然后将会得到类似如下的输出结果：

```
Epochs[1/60]--batch[0/196]--Acc: 0.1367--loss: 4.7522
Epochs[1/60]--batch[50/196]--Acc: 0.4961--loss: 2.7907
Epochs[1/60]--batch[150/196]--Acc: 0.5117--loss: 2.5251
...
Epochs[16/60]--Acc on test 0.8411
Epochs[17/60]--Acc on test 0.8186
Epochs[18/60]--Acc on test 0.8273
```

同时，在此过程中还可以通过命令 nvidim-smi 来查看此时 GPU 设备的工作情况，如下所示。

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 450.191.01 Driver Version: 450.191.01 CUDA Version: 11.0 |
|-------------------------------+----------------------+----------------------+
| GPU Name        Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan Temp Perf Pwr:Usage/Cap|         Memory-Usage | GPU-Util Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
| 0 Tesla T4            Off     |00000:18:00.0 Off |                    0|
| N/A 74C P0 77W / 70W          |1978MiB / 15109MiB |     95 %      Default |
|                               |                      |               N/A |
|-------------------------------+----------------------+----------------------+
| 1 Tesla T4            Off     |00000:18:00.0 Off |                    0 |
| N/A 74C P0 40W / 70W          |1920MiB / 15109MiB |     90 %      Default |
|                               |                      |           N/A |
+-------------------------------+----------------------+----------------------+
```

从输出信息可以看出，此时有两块 GPU 设备参与了模型的训练过程。

这里需要注意的一点是，多增加一倍的 GPU 数量并不意味着模型的训练速度会加快一倍，因为涉及 GPU 之间的通信和数据交互等，所以将同样数量的小批量数据从单卡放到多卡后，训练速度甚至可能出现变慢的情况。

### 5.6.4　小结

本节首先介绍了 GPU 模型训练中 3 种常见的训练策略的基本思想，包括模型并行、数据并行和混合并行；然后详细介绍了其中最常见的数据并行策略；最后，以 ResNet18 网络模型为例介绍了如何使用数据并行策略来完成模型的多 GPU 训练过程。

## 5.7　数据预处理缓存

随着任务场景和深度学习模型的复杂化，模型在训练过程中每次调试时都需要花费较长的时间来等待数据集预处理结果。一个简单直接的办法就是在模型每次载入数据集时都预先判断本地是否有对应的缓存文件，如果有，则直接载入，如果没有，则重新处理并进行缓存。同时，为了这段处理逻辑能够方便地迁移到其他类似情况中，因此需要将其定义成一个 Python 修饰器。

下面，先来简单介绍一个 Python 中修饰器的功能及用法。

### 5.7.1　修饰器介绍

关于什么是修饰器或装饰器(Decorator)这里就不从 Python 语法上来详细地进行解释了。修饰器的作用就是在正式执行某个功能函数之前，预先执行想要执行的某些逻辑。例如在进行数据预处理之前先判断是否有对应的缓存文件。下面，直接从用法的层面来逐步了解 Python 中的修饰器。

首先来看这样一个场景，假如之前已经定义了多个功能函数，但此时需要在日志文件中

同时输出每个函数的实际运行时间和其他相关信息。

打印出当前主程序正在调用哪个功能函数的信息，代码如下：

```
def add(a = 1, b = 2):
    time.sleep(2)
    r = a + b
    return r

def subtract(a = 1, b = 2):
    time.sleep(3)
    r = a - b
    return r
```

在上述代码中，time.sleep(2)是为了模拟运行所花费的时间。

进一步，对于上述两个函数，如果需要打印运行时间等相关信息，则可以通过如下类似方式实现：

```
1  def add(a = 1, b = 2):
2      print(f"正在执行函数 add() 。")
3      start_time = time.time()
4      time.sleep(2)
5      r = a + b
6      end_time = time.time()
7      print(f"一共耗时{(end_time - start_time):.3f}s")
8      return r
```

在上述代码中，第 2、第 3、第 6 和第 7 行便是需要打印输出的相关信息。虽然通过这样的方式也能解决问题，但是如果有大量的函数，并且每个函数都需要添加这么一段逻辑，则这种做法显然不可取。另外一种高效的方法则是使用 Python 中的修饰器。

假如现在已经定义好了一个名为 get_info 的修饰器，那么只需通过以下方式便可以打印上述相关信息，示例代码如下：

```
1  @get_info
2  def subtract(a = 1, b = 2):
3      time.sleep(3)
4      r = a - b
5      return r
6
7  if __name__ == '__main__':
8      subtract(3, 4)
9
10     #正在执行函数 subtract() 。
11     #一共耗时 3.002s
```

在上述代码中，第 1 行便调用了 get_info 修饰器。第 2～5 行是 subtract 函数原有的计算逻辑，并没有进行任何修改，所以此时只需在所有函数定义的地方使用 get_info 修饰器便可以实现运行时间计算的功能。

## 5.7.2 修饰器定义

在 Python 语法中，修饰器可以简单地分为包含参数和不包含参数两种。例如上面在

使用@get_info时便没有传入相关参数，如果包含参数，则使用方式类似@get_info(book_name="《跟我一起学深度学习》")。下面分别就这两种情况进行介绍。

**1. 不含参数的修饰器**

在使用修饰器之前，需要先定义一个完成目标功能的函数。对于5.7.1节中的例子来讲，示例代码如下：

```
1  def get_info(func):
2      def wrapper(*args, **kwargs):
3          print(f"正在执行函数 {func.__name__}() 。")
4          start_time = time.time()
5          result = func(*args, **kwargs)
6          end_time = time.time()
7          print(f"一共耗时{(end_time - start_time):.3f}s")
8          return result
9      return wrapper
```

在上述代码中，第3～4行和第6～7行便是为了实现目标功能所加入的逻辑。第5行则是原有功能函数的执行逻辑，例如5.7.1节中的add和subtract函数。

此时可以看出，get_info本质上就是定义了一个多层嵌套的函数，因此也可以通过函数调用的方式来使用，示例代码如下：

```
1  def subtract(a=1, b=2):
2      time.sleep(3)
3      r = a - b
4      return r
5
6  if __name__ == '__main__':
7      get_info(subtract)(7, 8)
```

虽然这样的方式也能实现同样的逻辑，但是使用起来不如修饰器简洁。

通过上述介绍可以发现，定义修饰器函数的大致格式如下：

```
1  def decorator(func):
2      def wrapper(*args, **kwargs):
3          #在这里添加需要预先执行的代码语句
4          result = func(*args, **kwargs)
5          #在这里添加需要事后执行的代码语句
6          return result
7      return wrapper
```

在上述代码中，第1行decorator为修饰器的名称，func为使用该修饰器的函数。第2行*args，**kwargs则为使用该修饰器函数的相关参数。第3行则是需要预先执行的计算逻辑。第4行则用于执行原有函数的计算逻辑。第5行是事后需要执行的计算逻辑。

同时，由于通过@符号来将decorator作为修饰器调用本质上只是一种快速简洁的方式，所以@decorator还等价于decorator(func)(*args，**kwargs)这样的调用方式，因此，通过后者我们还能够更加清晰地认识到整个修饰器的工作流程。

**2. 包含参数的修饰器**

所谓包含参数的修饰器指的是在调用修饰器时同时传入相关参数。例如在后续介绍数

据预处理结果缓存时，为了能够区分缓存结果的唯一性需要传入预处理时的相关参数，以此来构造一个缓存文件名，例如 top_k、max_len 或者 cut_words 这样的参数。因为对于不同的参数，构造的数据集并不一样。

对于需要传入用户参数的修饰器，其定义代码如下：

```
def get_info_with_para(name = None):
    print(f"name = {name}")
    def decorating_function(func):
        def wrapper( * args, ** kwargs):
            print(f"正在执行函数 {func.__name__}() 。")
            start_time = time.time()
            result = func( * args, ** kwargs)
            end_time = time.time()
            print(f"一共耗时{(end_time - start_time):.3f}s")
            return result
        return wrapper
    return decorating_function
```

在上述代码中，为了实现传入自定义参数我们在已有的两层函数之上又嵌套了一个函数。可以通过以下方式来使用：

```
@get_info_with_para(name = 'power function')
def power(num):
    time.sleep(3)
    r = num ** 2
    return r

name = power function
#正在执行函数 power() 。
#一共耗时 3.005s
```

上述完整的示例代码可参见 Code/Chapter05/C08_DataCache/decorator.py 文件。

### 5.7.3 定义数据集构造类

在介绍完修饰器的基本原理及用法之后再来看如何实现数据预处理结果缓存。整理逻辑依旧是本节内容伊始所提，载入数据集之前首先需要判断本地是否存在缓存，如果存在，则直接载入缓存，如果不存在，则调用函数进行数据预处理并进行缓存。

通常来讲，在构造训练集时可以通过定义一个类来完成，并且这个类至少包含 3 种方法：__init__、data_process 和 load_train_val_test_data，其中__init__用来初始化类中的相关参数（如 batch_size、max_len、file_ptah 等）；data_process 用来对数据集进行预处理并返回预处理后的结果；load_train_test_data 用来构造最后模型训练时的 DataLoader 迭代器，其定义代码如下：

```
class LoadData(object):
    FILE_PATH = './text_train.txt'

    def __init__(self):
```

```
5           self.max_len = 5
6           self.batch_size = 2
7
8       def data_process(self, file_path = None):
9           time.sleep(10)
10          logging.info("正在预处理数据……")
11          x = torch.randn((10, 5))
12          y = torch.randint(2, [10])
13          data = {"x": x, "y": y}
14          return data
15
16      def load_train_val_test_data(self):
17          data = self.data_process(file_path = self.FILE_PATH)
18          x, y = data['x'], data['y']
19          data_iter = TensorDataset(x, y)
20          data_iter = DataLoader(data_iter, batch_size = self.batch_size)
21          return data_iter
```

在上述代码中，第 4～6 行是初始化数据预处理的相关参数。第 8～14 行则用于模拟数据集的处理过程，这里直接随机生成，其中第 9 行用来模拟消耗的时间。第 16～21 行用来构造最后的迭代器。

## 5.7.4　定义缓存修饰器

在完成数据集构造类之后，只需按照 5.7.2 节中的语法完成缓存修饰器的实现，具体的示例代码如下：

```
1   def process_cache(unique_key = None):
2       if unique_key is None:
3           raise ValueError("unique_key 不能为空，请指定数据集构造类的成员变量")
4       def decorating_function(func):
5           def wrapper( * args, ** kwargs):
6               obj = args[0]
7               file_path = kwargs['file_path']
8               file_dir = f"{os.sep}".join(file_path.split(os.sep)[: - 1])
9               file_name = "".join(file_path.split(os.sep)[ -
                            1].split('.')[: - 1])
10              paras = f"cache_{file_name}_"
11              for k in unique_key:
12                  paras += f"{k}{obj.__dict__[k]}_" # 遍历对象中的所有参数
13              cache_path = os.path.join(file_dir, paras[: - 1] + '.pt')
14              start_time = time.time()
15              if not os.path.exists(cache_path):
16                  logging.info(f"缓存文件 {cache_path}不存在，重新处理并缓存。")
17                  data = func( * args, ** kwargs)
18                  with open(cache_path, 'wb') as f:
19                      torch.save(data, f)
20              else:
21                  logging.info(f"缓存文件 {cache_path}存在，直接载入缓存文件。")
22                  with open(cache_path, 'rb') as f:
23                      data = torch.load(f)
24              end_time = time.time()
25              logging.info(f"数据预处理共耗时{(end_time - start_time):.3f}s")
```

```
            return data
        return wrapper
    return decorating_function
```

在上述代码中，第 1 行 unique_key 用于区分同一原始数据但不同超参数所生成的缓存文件，如['top_k','cut_words','max_sen_len']等。第 6 行用于获取类对象，因为 data_process(self,file_path=None)中的第 1 个参数为 self。第 7 行用于获取方法 data_process 中 file_path 的取值。第 8～13 行根据文件名和传入的 unique_key 构造一个唯一的缓存文件名。第 15～19 行表示当本地不存在缓存文件时，根据第 17 行来对原始数据进行预处理并根据第 18～19 行将处理好的结果存放到本地。第 20～23 行用于直接从本地载入缓存文件。

在函数 process_cache 实现后，只需以修饰器@process_cache(['max_len'])的形式将其作用于 data_process 方法上，此时指定了用于区分不同缓存文件的参数名 max_len。

最后，在第 1 次使用上述数据集构造类时将会得到如下所示的输出信息：

```
#索引预处理缓存文件的参数为['max_len']
缓存文件 ./cache_text_train_max_len5.pt 不存在,重新处理并缓存。
正在预处理数据……
数据预处理一共耗时 10.006s
```

从上述结果可以看出，数据集处理完毕后将会生成一个名为 cache_text_train_max_len5.pt 的缓存文件，并且一共耗费了 10s 的时间。

当第 2 次载入同样的缓存文件时，则会得到如下所示的输出信息：

```
#索引预处理缓存文件的参数为['max_len']
缓存文件 ./cache_text_train_max_len5.pt 存在,直接载入缓存文件。
数据预处理一共耗时 0.002s
```

从上述结果可以看出，由于此时本地缓存文件存在，所以直接从本地载入了缓存文件，一共耗时不到 1s。

到此，对于如何利用 Python 修饰器来便捷地缓存数据预处理结果的内容就介绍完了，上述完整的示例代码可参见 Code/utils/tools.py 文件。

### 5.7.5 小结

本节首先从使用示例的角度来介绍了 Python 修饰器的用法及工作原理，即其本质上只是 Python 中所支持的一种快速简洁的函数调用方式，然后介绍了不含参数和含有参数两种修饰器的实现方法；最后通过一个实际的示例详细地介绍了如何从零实现一个可通用的数据预处理缓存修饰器。