# 第 3 章 深度学习编程环境操作基础

初学者一定希望立刻上手进行深度学习的编程，实践网络模型的训练、测试和部署。工欲善其事，必先利其器。在此之前，我们需要在自己的计算机上搭建深度学习的开发环境，学习和掌握基本的系统操作命令和开发工具的使用方法，才能如鱼得水地进行深度学习实践。本章内容包括 Linux 操作系统、Python 编程语言开发工具、常用 Python 库、TensorFlow 和 PyTorch 深度学习框架及 SE5 平台的安装和操作，引导零基础读者快速入门深度学习的开发环境。有基础的读者可跳过本章。

## 3.1 Linux 入门

Linux 操作系统发行版本众多，常用的包括 Ubuntu、CentOS、Debian、Deepin、Fedora 等系统。Linux 系统的操作模式分为桌面模式和终端模式。桌面模式是图形化界面，与 MS Windows 系统类似，简单易学。终端模式即 Terminal，通过输入命令行来执行任务。

本书推荐使用 Ubuntu 操作系统。由于 Ubuntu 系统持续更新，安装方法也有所不同，因此本书仅简述 Ubuntu 的安装步骤，读者可自行网上搜索最新安装教程。

### 3.1.1 Linux 系统的安装简介

在 PC 上可选择 3 种方式中的一种进行 Linux 操作系统的安装：

(1) Ubuntu 单操作系统。

(2) Windows 和 Ubuntu 双操作系统，在开机时选择需要进入的操作系统。

(3) 在 Windows 系统中建立 Ubuntu 虚拟机。

访问 Ubuntu 官网 https://ubuntu.com/download/desktop，下载 Ubuntu 桌面版镜像文件。将镜像文件制作成自启动 U 盘。安装前请注意：

(1) 备份 PC 内的资料。

(2) 确保网络连接正常。

PC 开机，按下 F2 键进入 BIOS，选择使用 U 盘启动。按照系统提示进行操作。

(1) 选择安装方式：Normal installation(正常安装)。

(2) 选择安装类型：Install Ubuntu alongside Windows Boot Manager(双操作系统并存，在开机时选择进入哪个操作系统)；

Erase disk and install Ubuntu(删除现 PC 所有文件，只安装 Ubuntu 系统)；

Something else(用户自己创建或调整分区安装 Ubuntu)。

建议初学者选择 Windows 和 Ubuntu 双系统安装类型。

(3) 按照安装界面提示继续完成 Ubuntu 的安装。

安装完成后,重新开机,按照 GNU GRUB 提示选择进入 Windows 或 Ubuntu 操作系统。

虚拟机安装步骤如下所示。

(1) 访问 VMware 官网 https://www.vmware.com/cn/products/workstation-pro/workstation-pro-evaluation.html,下载安装包,安装 VMware 软件。注意:VMware 为商业软件,需购买许可证。免费版有 30 天使用期限。

(2) 运行虚拟机,选择【打开虚拟机】,进入虚拟机目录,选择 Ubuntu64 位.vmx 扩展名的文件打开。在【设备】菜单中配置虚拟机的物理内存(一般为 8GB 及以上)、硬盘等。

(3) 单击【继续运行此虚拟机】按钮开启 Ubuntu 虚拟机。Ubuntu 桌面图形界面进入可以使用状态。

除以上 3 种 Ubuntu 安装方式外,Windows 10 也有自带的 Ubuntu 系统,感兴趣的读者可以自行安装尝试,本书不再赘述。

### 3.1.2　Linux 系统的常用命令

在 Linux 系统中单击终端图标或使用快捷键 Ctrl+Alt+T 进入终端模式。在终端模式下,显示命令行的提示符及含义如图 3.1 所示。

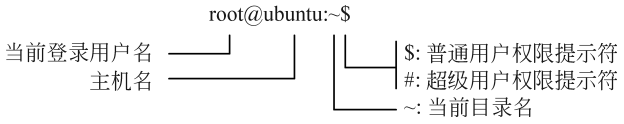


**图 3.1　命令行提示符含义**

Linux 下基本命令的使用格式为:命令 [-选项] [参数]。但并非所有命令都严格遵循这种格式。

(1)创建、切换和显示目录相关命令,如表 3.1 所示。

**表 3.1　创建、切换和显示目录相关命令**

| 命　令 | 命 令 格 式 | 功　能 |
|---|---|---|
| mkdir | mkdir [目录名] | make directories:创建单级目录 |
| | mkdir -p [目录名] | 递归创建多级目录 |
| cd | cd [目录名] | change directory:进入指定目录 |
| | cd.. | 切换到上一级目录 |
| | cd / | 切换到根目录 |
| | cd ~ | 切换到当前用户目录 |
| ls | ls | list:默认方式显示当前目录文件/文件夹列表 |
| | ls -a | 显示所有文件,包括隐藏文件 |
| | ls -l | 显示文件的详细信息 |
| | ls -lh | 以人性化方式显示文件大小 |
| | ls [目录名] | 显示指定目录的内容 |
| tree | tree | 以树状图方式列出目录的内容 |
| pwd | pwd | print working directory:显示当前工作目录的完整路径 |

创建、切换和显示目录命令操作实例如表 3.2 所示。

**表 3.2 创建、切换和显示目录命令操作实例**

| 命 令 | 解 释 |
| --- | --- |
| ngit@ubuntu:~ $ mkdir linux_learn | 创建目录 linux_learn |
| ngit@ubuntu:~ $ mkdir -p linux_learn/myfile/file | 递归创建多级目录 |
| ngit@ubuntu:~ $ cd linux_learn/ | 进入目录 linux_learn |
| ngit@ubuntu:~/linux_learn $ mkdir python | 在当前路径下创建目录 python |
| ngit@ubuntu:~/linux_learn $ ls<br>myfile python | 显示目录文件 |
| ngit@ubuntu:~/linux_learn $ tree<br>├── myfile<br>│ └── file<br>└── python<br>3 directories, 0 files | 以树状图列出目录的内容 |

(2) 复制、移动和删除文件或目录命令,如表 3.3 所示。

**表 3.3 复制、移动和删除文件或目录命令**

| 命令 | 命 令 格 式 | 功 能 |
| --- | --- | --- |
| cp | cp -i | copy: 复制文件或目录时,覆盖文件前提示用户是否覆盖 |
| | cp -r | 复制该目录下所有子目录和文件 |
| | cp -p | 保留文件属性不变 |
| mv | mv [原文件或目录] [目标文件或目录] | move: 移动、重命名文件/目录 |
| rm | rm -r [文件或目录] | remove: 删除文件或目录下的所有内容 |
| | rm -f [文件或目录] | 强制删除,忽略不存在的文件,无须提示 |

(3) 常用的压缩和解压缩命令,如表 3.4 所示。

**表 3.4 常用的压缩和解压缩命令**

| 命令 | 功 能 | 命 令 格 式 | 文 件 格 式 |
| --- | --- | --- | --- |
| zip | 压缩文件或目录 | zip [选项] [压缩后文件名] [文件或目录];<br>-r 压缩目录 | 压缩后文件格式: .zip |
| unzip | 解压.zip 压缩文件 | unzip [压缩文件] | |
| tar | gz 压缩 | tar -zcvf [file.tar.gz] [file] | 压缩成.tar.gz 格式 |
| | bz2 压缩 | tar -jcvf [file.tar.bz2] [file] | 压缩成.tar.bz2 格式 |
| | gz 格式解压缩 | tar -zxvf [file.tar.gz] | |
| | bz2 格式解压缩 | tar -jxvf [file.tar.bz2] | |

(4) 常用的权限和下载相关命令,如表 3.5 所示。

表 3.5　常用的权限和下载相关命令

| 命令 | 命 令 格 式 | 功　　能 |
|---|---|---|
| sudo | sudo ＋ 命令 | superuser do：以管理员权限执行命令，执行只有 root 权限才能执行的命令 |
| pip | pip install Python 包名<br>pip uninstall Python 包名 | package installer for python：Python 包管理，用于查找、下载、安装、升级、卸载 Python 包 |
| wget | wget http://... | world wide web get：从网络上自动下载文件 |

(5) 改变文件/目录权限的命令，如表 3.6 所示。

表 3.6　改变文件/目录权限的命令

| 不同角色的象征性表示 | 不同权限的符号表示 | 用 法 举 例 |
|---|---|---|
| u：user(用户) | r：读取权限 | (1)为用户提供执行权限：sudo chmod u＋x file<br>(2)删除用户的读写权限：sudo chmod u－rx file<br>(3)为所有人提供执行权限：sudo chmod a＋x file<br>(4)用逗号分隔多个权限集：sudo chmod u＋r,g＋x file |
| g：group(组) | w：写入权限 | |
| o：others(其他人) | x：执行权限 | |
| a：all(所有人，包括用户、组、其他人) | ＋：添加权限<br>－：删除权限 | |

### 3.1.3　Linux 的文本编辑器

gedit 是 Linux 下的一个纯文本编辑器，简单易用。在终端模式下输入命令：gedit test.py，便可新建并打开 test.py 文件。gedit 编辑器的工具栏包括新建、打开、保存、打印、撤销、复制、粘贴、搜索等功能，如图 3.2 所示。


图 3.2　gedit 编辑器工具栏

## 3.2　Python 入门

本节介绍使用 Anaconda 进行 Python 环境的安装、Python 解释器的使用及 PyCharm 软件的安装和使用，并示范在 PyCharm 集成开发环境下进行 Python 编程、调试和运行。

### 3.2.1　Python 环境的安装和使用

**1. Anaconda 安装**

Anaconda 是一款 Python 集成环境安装包。执行 Anaconda 后自动安装 Python、IPython 和众多的包和模块。Anaconda 的安装步骤如下所示。

(1) 下载 Anaconda。在终端模式下，进入 home 目录，执行 wget 下载命令：

```
ngit@ubuntu:~$ cd ~
ngit@ubuntu:~$ wget https://mirrors.tuna.tsinghua.edu.cn/anaconda/archive/
Anaconda3-5.1.0-Linux-x86_64.sh
```

输入 ls 可以查看到下载的 Anaconda3-5.1.0-Linux-x86_64.sh 脚本文件。

(2) 安装 Anaconda。在 home 目录下,执行该脚本文件进行安装:

```
ngit@ubuntu:~$ bash Anaconda3-5.1.0-Linux-x86_64.sh
```

在 Anaconda 安装过程中,按照安装提示,按 Enter 键或输入 yes 继续安装过程。安装完成后输入 ls 命令,显示如下:

```
ngit@ubuntu:~$ ls
anaconda3                                CNN-Saliency-Map  examples.desktop
Anaconda3-5.1.0-Linux-x86_64.sh  data  data         libevent-2.0.20-stable
```

(3) 配置环境变量。安装完成后,执行如下命令配置环境变量:

```
ngit@ubuntu:~$ echo 'export PATH="~/anaconda3/bin:$PATH"' >> ~/.bashrc
ngit@ubuntu:~$ source ~/.bashrc
```

**2. Python 解释器的使用**

在终端命令行中输入 python3 命令启动 Python 交互式编程,出现 Python 提示符>>>。在 Python 提示符后输入程序语句,按 Enter 键执行:

```
ngit@ubuntu:~$ python3
Python 3.6.4 |Anaconda, Inc.| (default, Jan 16 2018, 18:10:19)
[GCC 7.2.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>> 5+3
8
>>> print("hello world")
hello world
```

Python 解释器是以命令行的方式执行 Python 语句,非常不方便。本书推荐使用 PyCharm 软件作为 Python 的集成开发环境。PyCharm 可以进行代码的编辑、Python 环境的配置、程序运行和断点调试,以及远程连接服务器的方式调试运行代码(仅限专业版)。

### 3.2.2 PyCharm 集成开发环境的安装和使用

**1. PyCharm 环境搭建**

(1) 下载 PyCharm 安装包。

进入 JetBrains 官网 https://www.jetbrains.com/pycharm/download/,下载 Professional(付费专业版)或 Community(免费社区版)。

(2) 安装 PyCharm。

下载完成后,执行解压命令(由于版本号随时更新,读者在执行下列命令时,要先查看下

载的文件名，用下载的文件名替换掉该命令中的文件名）：

```
ngit@ubuntu:~$ tar -zxvf pycharm-community-2022.3.tar.gz
```

解压完成进入 bin 目录，输入命令：sh pycharm.sh，便可直接启动 PyCharm。

**2. 新建 Python 项目和设置解释器**

（1）进入【文件】菜单，选择【新建项目】。

（2）如图 3.3 所示，在“创建项目”对话框【位置】处输入新建项目的文件夹目录，【基础解释器】处选择 Python 解释器版本。单击【创建】按钮，完成新项目的创建。

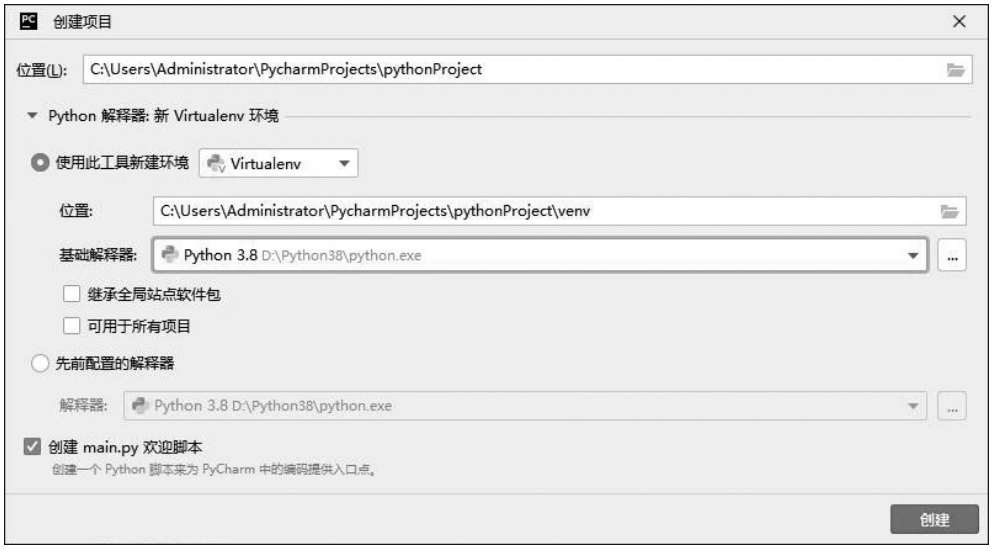


**图 3.3　“创建项目”对话框**

**3. 编写程序**

右击 PyCharm 左侧树状图的新建项目名称，选择【新建】→【Python 文件】，输入文件名，如 try，创建 try.py 文件，然后在右侧的编辑窗口中编写源代码。例程如下：

```
import cv2                                  #导入 OpenCV 库
img = cv2.imread("D:/dg.jpg", 1)            #读入图像
img_fr = cv2.flip(img,1)                    #翻转图像
pix = img[100, 200, 0]                      #获取原图像(100,200)处 B 通道的像素值
pix1 = img[100, 200]                        #获取原图像(100,200)处的像素值
print(pix, pix1)                            #打印两个像素值
img_fr = cv2.resize(img_fr,(200,200))       #缩放到 200×200
cv2.imshow('Original', img)                 #显示原始图像
cv2.imshow('Flip & Resize', img_fr)         #显示翻转和缩放后的图像
cv2.waitKey(0)                              #保持图像，按任意键退出
cv2.destroyAllWindows()                     #释放所有创建窗口
```

**注意**：在项目文件夹下事先放置一张图像，命名为 dg.jpg。# 符号为 Python 语言的注释符号。

**4. 安装运行库**

本例程使用了 OpenCV 库，因此在运行 Python 程序前先进行 OpenCV 库的安装。OpenCV 是 Open source Computer Vision 的缩写，它提供了 Python、MATLAB 等语言的接口，实现了高级图形用户接口、计算机视觉等算法。程序中读取图像、图像翻转、图像缩放和显示都要使用 OpenCV 的库函数。

OpenCV-Python 安装：单击 PyCharm 最下方状态栏的【终端】按钮进入终端模式，输入命令：

```
pip install opencv-python -i https://pypi.tuna.tsinghua.edu.cn/simple
```

**5. 运行程序**

将鼠标放在程序编辑窗口内右击，在弹出的菜单中选择【运行'try'(U)】。运行结果如图 3.4 所示，程序生成两个图像窗口，并在"运行"窗口显示像素值。


图 3.4 程序运行结果

**6. 断点调试程序**

断点调试能清楚地看到代码运行的过程及每一步的变量值，便于对代码问题进行跟踪。

(1) 添加断点：在代码区要进行调试的代码行左侧单击添加断点调试符号，如图 3.5 所示为"断点"所指向的位置。当删除断点时，只需再次单击断点处即可。


图 3.5 调试模式

(2) 断点调试状态下运行代码：在右上角工具栏区单击【调试'try'】图标按钮，或使用快捷键【Shift+F9】，或右击源文件或代码区内部，在弹出菜单中选择【调试'try'(D)】。

程序进入调试状态，如图 3.5 所示，可以在"变量窗口"查看变量的类型和计算结果。

(3) 单步调试：如图 3.5 的"单步调试"所示，从左到右的 5 个箭头图标对应单步调试方式，其含义如表 3.7 所示。

**表 3.7　单步调试命令及快捷按键**

| 调试命令 | 快捷按键 | 功　能 |
|---|---|---|
| Step Over | F8 | 单步执行代码，遇到子函数时，一步执行完子函数，不进入子函数内部单步执行语句 |
| Step Into | F7 | 单步执行代码，遇到子函数时，进入子函数内部单步执行语句 |
| Step Into My Code | Alt+Shift+F7 | 单步执行代码，只进入自己编写的子函数内部 |
| Step Out | Shift+F8 | 一步执行完子函数，回到调用子函数代码的下一行代码 |
| Run to Cursor | Alt +F9 | 运行到当前光标位置 |

### 3.2.3　常用 Python 库

Python 包含非常丰富的库函数，降低了学习和使用的门槛。Python 语言库分为 Python 标准库和 Python 第三方库，Python 标准库的相关信息及使用指南请查阅 https://docs.python.org/zh-cn/3/。Python 有很多功能强大的第三方库，如专门用于科学计算的函数库 Numpy，基于 Numpy 构建的高性能数据分析工具 Pandas，绘制图形的工具库 Matplotlib，机器视觉库 OpenCV 等。

**1. NumPy**

NumPy 是 Numerical Python 的缩写，是进行科学计算和数据分析的基础软件包，针对数组运算提供了大量的数学函数库，具有强大的数学与矩阵运算能力。同时，NumPy 集成了 C/C++ 和 FORTRAN 代码的工具，具有强大的线性代数、傅里叶变换和随机数等功能。

更多 NumPy 信息及使用教程可阅读 NumPy 官网：https://numpy.org/。

**2. Matplotlib**

Matplotlib 是 Python 的绘图库，是非常好的数据可视化工具。我们可以使用短短几行代码方便地绘制散点图、曲线图、直方图、柱状图，也可以对高维数据进行可视化观察，直观地感受数据，对分析和处理数据非常有帮助。

更多 Matplotlib 信息及使用教程可阅读 Matplotlib 官网：https://matplotlib.org/。

**3. OpenCV**

我们在上面例子中已经使用了 OpenCV 库，在本书的深度学习实践中，将多次使用 OpenCV 函数实现图像预处理和后处理功能。

### 3.2.4　Python 虚拟环境

不同的 Python 应用程序可能使用了不同版本的 Python 及第三方库，这意味着仅安装

一种 Python 版本或库可能无法满足每个应用程序的要求。例如，应用程序 A 需要特定模块的 1.0 版本，但应用程序 B 需要 2.0 版本，仅安装版本 1.0 或 2.0 将导致某一个应用程序无法运行。这个问题可以通过创建 Python 应用程序各自的虚拟环境来解决，虚拟环境允许为不同的项目安装配置各自的 Python 环境，而不是安装到整个系统，从而避免环境冲突。例如，应用程序 A 可以拥有自己的安装了 1.0 版本的虚拟环境，而应用程序 B 则拥有安装了 2.0 版本的另一个虚拟环境。如果应用程序 B 要求将某个库升级到 3.0 版本，也不会影响应用程序 A 的虚拟环境。

在本书的实践项目中使用了 3 个版本的深度学习框架。项目文件夹命名“XXX-TF1”，使用 TensorFlow 1.x 版本；项目文件夹命名“XXX-TF2”，使用 TensorFlow 2.x 版本；项目文件夹命名“XXX-Pytorch”，使用 PyTorch 深度学习框架。读者需要根据实践项目所使用的框架构建各自的虚拟环境。

Anaconda 可以用于虚拟环境的创建。如果使用 PyCharm 集成环境进行项目开发，则可以直接使用 PyCharm 的 Python 解释器配置虚拟环境。

**1. 使用 Anaconda 创建和使用 Python 虚拟环境**

（1）创建并安装 Python 虚拟环境：

```
conda create -n your_env_name python=3.6
```

your_env_name 是用户自定义的虚拟环境名称，python＝3.6 指定了本虚拟环境对应的 Python 版本，若不指定版本则选用当前环境变量中配置的 Python 版本。若环境变量中没有配置，则下载最新版本。创建的虚拟环境在 anaconda3/envs 目录下，即/anaconda3/envs/your_env_name。

（2）进入和退出虚拟环境：

```
conda activate your_env_name 或 source activate your_env_name        #进入
conda deactivate your_env_name 或 source deactivate your_env_name    #退出
```

（3）为虚拟环境安装、删除包：

```
conda install -n your_env_name [package]          #安装
conda remove --name your_env_name [package]       #删除
```

（4）移除和重建虚拟环境：

```
conda remove -n your_env_name --all               #移除
conda create -n your_env_name python=3.6          #重建
```

**2. 在 PyCharm 中配置虚拟环境**

（1）进入【文件】菜单，选择【设置】。如图 3.6 所示，在“设置”对话框中，先单击左侧的【Python 解释器】，右侧会显示出该项目的环境配置。

（2）单击对话框右侧的【Python 解释器】下拉框，选择指定的 Python 版本。

（3）单击【＋】或【－】添加或删除指定的软件包。


**图 3.6　PyCharm 虚拟环境配置**

## 3.3　TensorFlow 入门

TensorFlow 由 Google Brain 实验室开发和维护，是目前最优秀的深度学习框架之一。它的底层 API 基于 C/C++ 语言开发，为 Python、C++、Java、Go 等多种语言提供上层调用 API 接口，其中对 Python 的支持最全面、最完善。

相比 TensorFlow 1.x 版本，TensorFlow 2.x 版本显著提高了简洁易用性。首先，TensorFlow 2.x 版本重新整合了 1.x 中的许多 API，使 API 接口更加精简整洁。其次，TensorFlow 2.x 版本内置了深度学习框架 Keras，Keras 简单易用，仅用几行程序就可以构造出一个神经网络。此外，TensorFlow 2.x 版本还使用了动态图机制(eager execution)，使用户编写调试代码更加方便，也让新用户的学习更加容易。

TensorFlow 旨在为深度学习工业化提供更好的支持，在深度学习的算法训练、推理、跨平台部署等工业化全链路环节都有很好的工具支持，因此在工业化应用场景颇受用户欢迎。

有关 TensorFlow 的更多信息与指南，可阅读 TensorFlow 官网 https://tensorflow.google.cn/guide/basics。

### 3.3.1　TensorFlow 的安装

使用 Python 包管理器 pip 或 Anaconda 包管理器 conda 安装 TensorFlow。其中，conda 源的通用性更强但版本更新较慢，难以第一时间获得最新 TensorFlow 版本。

以下是使用国外镜像源指定安装 TensorFlow 2.10.0 版本的 pip 命令。在不指定版本的情况下，系统安装 pip 源中的最新版本：

```
ngit@ubuntu:~$ pip install tensorflow==2.10.0
```

也可以使用国内(清华)镜像源安装 TensorFlow，安装速度更快，命令如下：

```
ngit@ubuntu:~$ pip install tensorflow==2.10.0 -i https://pypi.tuna.tsinghua.
edu.cn/simple/
```

若 PC 配置了 GPU 硬件，并已安装 NVIDIA 的通用并行计算架构 CUDA 和深度神经网络的 GPU 加速库 CUDNN，则可以安装 GPU 版 TensorFlow：

```
ngit@ubuntu:~$ pip install tensorflow-gpu==2.10.0 -i https://pypi.tuna.
tsinghua.edu.cn/simple/
```

完成 TensorFlow 安装之后，进入 Python 环境，检查 TensorFlow 是否正确安装：

```
ngit@ubuntu:~$ python3
>>> import tensorflow
```

若没有报错，则表示 TensorFlow 安装成功。

### 3.3.2 TensorFlow 的基本操作

在 TensorFlow 中，张量分为常量和变量两大类，支持几乎所有的数据类型。例如，数值数据类型(tf.uint8、tf.int32、tf.float32 等)、布尔数据类型(tf.bool)和字符串数据类型(tf.string)等。此外，TensorFlow 还自创了一些数据类型，如用于张量类型转换的 tf.cast 等。

有关 TensorFlow 中张量的更多使用，请阅读 TensorFlow 官网教程 https://tensorflow.google.cn/guide/tensor。

**1. 常量的定义和使用**

在 TensorFlow 中，常量型张量可以用 tf.constant 来生成。

```
#1. 导入 TensorFlow 库
import tensorflow as tf
#2. 定义一个常量(零阶张量)，数据类型默认为 int32
tensor_0 = tf.constant(1)
print(tensor_0)
#3. 定义一个浮点型常量(一阶张量)，数据类型为 float32
tensor_1 = tf.constant([3.0, 1.0, 5.0, 2.0])
print(tensor_1)
#4. 定义一个常量(二阶张量)，数据类型设置为 float16
tensor_2 = tf.constant([[1, 2], [3, 4]], dtype=tf.float16)
print(tensor_2)
#5. 定义一个常量(三阶张量)，数据类型默认为 int32
tensor_3 = tf.constant([[[0, 1, 2],[3, 4, 5],[6, 7, 8]],[[9, 10, 11],[12, 13, 14],
[15, 16, 17]],[[18, 19, 20],[21, 22, 23],[24, 25, 26]]])
print(tensor_3)
```

输出结果如下。张量具有值(value)、维度属性(shape)和数据类型属性(dtype)。同时TensorFlow 还提供了许多修改数据维度的函数,如 reshape()、resize()等。

```
tf.Tensor(1, shape=(), dtype=int32)                                    #1. 零阶张量
tf.Tensor([3. 1. 5. 2.], shape=(4,), dtype=float32)                    #2. 一阶张量
tf.Tensor(                                                             #3. 二阶张量
      [[1. 2.]
      [3. 4.]], shape=(2, 2), dtype=float16)
tf.Tensor(                                                             #4. 三阶张量
      [[[ 0  1  2]
      [ 3  4  5]
      [ 6  7  8]]
      [[ 9 10 11]
      [12 13 14]
      [15 16 17]]
     [[18 19 20]
      [21 22 23]
      [24 25 26]]], shape=(3, 3, 3), dtype=int32)
Process finished with exit code 0
```

**2. 变量的定义和使用**

变量型张量用 tf.Variable()来生成。

```
import tensorflow as tf
#1. 使用 tf.Variable()定义一个变量 w1,设定初始值,数据类型及名称
w1 = tf.Variable(initial_value=[[3,2],[4,5]],dtype=tf.float32,name='w1')
#2. 查看该变量的数据类型及形状,并导出至 Numpy
print(w1.dtype)
print(w1.shape)
print(w1.numpy())
```

输出结果为:

```
<dtype: 'float32'>     #1. 数据类型为 32 位浮点数
(2, 2)                 #2. 矩阵的长和宽均为 2
[[3. 2.]
[4. 5.]]               #3. 数据导出至 Numpy
```

继续上面的程序,对变量进行运算:

```
#3. 定义一个常量 x,为 1 * 2 的矩阵
x =tf.constant(value=[[2,3]],dtype=tf.float32,name='x')
#4. 定义一个变量 b,由于设定初始值为 32 位浮点数,故无须指定数据类型
b = tf.Variable(initial_value=[1.2,3.5],name='b')
#5. TensorFlow 中包含许多操作,我们以矩阵间的运算为示例
result_mul = tf.matmul(x,w1)            #矩阵乘法运算
result_add = tf.add(result_mul,b)       #矩阵加法运算
#6. 实现最终结果 result_add = x * w1+b
print("result_matmul:{}".format(result_mul))
print("result_add:{}".format(result_add))
```

结果为：

```
result_matmul:[[18. 19.]]
result_add:[[19.2 22.5]]
```

如果需要对变量进行更新和迭代操作，可以通过循环来实现：

```
import tensorflow as tf
#1. 定义变量 epochs
epochs = tf.Variable(initial_value=0,dtype=tf.int32,name='epochs')
#2. 变量的更新
for i in range(5):
        epochs=epochs+1
print(format(epochs))
```

结果为：

```
5
```

**3. 自动微分和梯度**

自动微分是深度学习算法（如神经网络训练的反向传播过程）的关键组成部分，TensorFlow 为自动微分提供了 tf.GradientTape API 函数，根据某个函数的输入变量来计算梯度。

```
#1. 导入 TensorFlow 库
import tensorflow as tf
#2. 定义需要计算梯度的变量 x
x = tf.Variable(3.0)
with tf.GradientTape() as tape:
  y = x * 2+5
#3. 计算 y=2x+5 在 x=3 时关于变量 x 的导数
dy_dx = tape.gradient(y, x)
print(dy_dx)
```

输出结果为：

```
tf.Tensor(2.0, shape=(), dtype=float32)
```

tf.GradientTape 可以应用于任何维度的张量。除此之外，GradientTape 的资源在调用一次 gradient 函数之后就会被释放，如果需要多次计算，需要将 persistent＝True 属性开启。

```
#1. 导入 TensorFlow 库
import tensorflow as tf
#2. 定义一个二维随机矩阵 w,以及一个二维全 0 矩阵 b
w = tf.Variable(tf.random.normal((3, 2)), name='w')  #定义一个服从正态分布的随机
                                                     #3 行 2 列矩阵
```

```
#定义一个全零变量,数据类型为 32 位浮点数
b = tf.Variable(tf.zeros(2, dtype=tf.float32), name='b')
x = [[1., 2., 3.]]
#3. 分别计算 loss 关于变量 w 和变量 b 的梯度
with tf.GradientTape(persistent=True) as tape:  #开启 persistent=True 属性
  y = tf.add(tf.matmul(x,w),b)
  loss = tf.reduce_mean(y* *2)  #tf.reduce_mean 用于计算张量沿着某一维度的平均值
[dl_dw, dl_db] = tape.gradient(loss, [w, b])
print(w)
print(dl_dw)
```

输出结果为：

```
<tf.Variable 'w:0' shape=(3, 2) dtype=float32, numpy=
array([[-1.6353815,  0.8802248],
       [ 1.5613909,  1.5711588],
       [ 0.8008925, -1.1357195]], dtype=float32)>
tf.Tensor(
    [[ 3.8900778   0.61538386]
     [ 7.7801557   1.2307677 ]
     [11.670234    1.8461516 ]], shape=(3, 2), dtype=float32)
```

### 3.3.3　使用 TensorFlow 实现手写数字识别

本节通过 TensorFlow 框架构建一个只包含两层全连接层的神经网络模型，实现了 MNIST 手写数字识别，使读者能够对 TensorFlow 的使用有一个大致的了解，并能够快速上手。读者可以从 www.tup.com.cn 下载本实验的完整源代码。

**1. 加载 MNIST 数据集**

本实践项目使用 TensorFlow 2.0 的 tf.keras 库进行算法设计。代码原型来自 TensorFlow 官网 https://tensorflow.google.cn/tutorials/quickstart/beginner。

tf.keras 提供了 MNIST 数据集的函数封装，通过下面的代码可完成数据集的加载和准备：

```
import tensorflow as tf
#1. 声明使用 MNIST 数据集,TensorFlow 内置 MNIST 数据集
mnist = tf.keras.datasets.mnist
#2. 训练和测试数据集加载
(x_train, y_train), (x_test, y_test) = mnist.load_data()
#3. 数据归一化
x_train, x_test = x_train / 255.0, x_test / 255.0
```

**2. 构建神经网络模型**

手写数字识别网络具有两个全连接层，输入层为 784 个神经元，对应输入的 784 维向量。隐藏层为 128 个神经元，与输入层构成第一个全连接层，输出 128 维特征向量。输出层为 10 个神经元，与隐藏层构成第二个全连接层，输出 0～9 的 10 个数字分类的预测值。网络模型可表示为

$$\boldsymbol{z}=\text{dropout}(\text{ReLU}(\boldsymbol{W}_1\boldsymbol{x}+\boldsymbol{b}_1))$$

$$\boldsymbol{y}=\boldsymbol{W}_2\boldsymbol{z}+\boldsymbol{b}_2$$

式中：$\boldsymbol{x}$ 为 784 维的输入向量，即一张展开后的 MNIST 图像；$\boldsymbol{W}_1$ 为第一个全连接层的权重矩阵，大小为 $784\times128$；$\boldsymbol{b}_1$ 为 128 维的偏置向量；$\boldsymbol{z}$ 为第一个全连接层的输出；ReLU 为激活函数，dropout 层防止网络过拟合；$\boldsymbol{W}_2$ 为第二个全连接层的权重矩阵，大小为 $128\times10$；$\boldsymbol{b}_2$ 为 10 维的偏置向量；$\boldsymbol{y}$ 为第二个全连接层的输出，是一个 10 维向量，向量的值表示该图像属于每类的预测值，最大预测值的索引为该手写字符的类别。

通过堆叠层来构建 tf.keras.Sequential 模型：

```
model = tf.keras.models.Sequential([
            #1. 将输入图像数据从二维数组[28,28]展开为 784 维向量
            tf.keras.layers.Flatten(input_shape=(28, 28)),
            #2. 全连接层,将 784 维向量映射为 128 维向量,使用 ReLU 函数激活
            tf.keras.layers.Dense(128, activation='relu'),
            #3. Dropout 层
            tf.keras.layers.Dropout(0.2),
            #4. 全连接层,将 128 维向量映射为 10 维向量输出,对应 10 个类别
            tf.keras.layers.Dense(10)
            ])
```

**3. 定义损失函数**

本例为多分类问题，使用交叉熵损失函数：

$$\text{LOSS}=-\frac{1}{n}\sum_{i=1}^{C}\hat{y}_i\log y_i \tag{3.1}$$

式中：$y_i$ 是模型预测结果向量 $y$ 的元素；$\hat{y}_i$ 是真实标签向量 $\hat{y}$ 的元素。实现代码如下：

```
#声明交叉熵损失函数
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
```

**4. 训练模型**

训练前使用 model.compile 配置优化器、损失函数和评估指标。将优化器(optimizer)设置为 adam，将 loss 设置为前面声明的 loss_fn 函数，将 metrics 参数设置为 accuracy 来指定模型的评估指标：

```
model.compile(optimizer='adam',
              loss=loss_fn,
              metrics=['accuracy'])
```

使用 model.fit 方法训练迭代，优化模型参数并最小化损失：

```
model.fit(x_train, y_train, epochs=5)
```

训练过程的输出如下：

```
Epoch 1/5 1875/1875 [====] - 20s 8ms/step - loss: 0.2920 - accuracy: 0.9151
Epoch 2/5 1875/1875 [====] - 15s 8ms/step - loss: 0.1414 - accuracy: 0.9581
```

```
Epoch 3/5 1875/1875 [====] - 15s 8ms/step - loss: 0.1079 - accuracy: 0.9678
Epoch 4/5 1875/1875 [====] - 14s 8ms/step - loss: 0.0874 - accuracy: 0.9724
Epoch 5/5 1875/1875 [====] - 15s 8ms/step - loss: 0.0739 - accuracy: 0.9768
```

**5. 评估模型**

使用 model.evaluate 方法在测试集上评估模型的准确率。代码如下：

```
model.evaluate(x_test,  y_test, verbose=2)
```

模型的评估输出如下：

```
313/313 - 1s - loss: 0.0740 - accuracy: 0.9765 - 1s/epoch - 3ms/step
```

本例在 MNIST 数据集的准确率达到 97.65%。

**6. 保存和加载推理模型**

TensorFlow 支持多种保存模型的方式，这里介绍其中的两种保存方式。第一种为保存整个模型的结构信息和参数信息，代码如下：

```
model.save('model.h5')
#对应的模型加载方式
tf.keras.models.load_model('model.h5')
```

第二种为仅保存模型的参数，代码如下：

```
model.save_weights('model.h5')
#对应的模型加载方式
reinitialized_model.load_weights('model.h5')
```

## 3.4　PyTorch 入门

PyTorch 是 Facebook AI Research 用 Python 语言开发的深度学习框架。PyTorch 是个简洁且高效快速的框架，编程风格更贴近 Python 习惯，易于理解和上手，因此深受科研人员的欢迎。在 2022 年的 AI 顶级会议论文中，使用 PyTorch 作为训练框架的占比超过 80%。但使用 PyTorch 训练的模型，在进行跨平台部署等环节仍需要将模型转换为其他框架，其在工业界的应用便捷度不如 TensorFlow 训练的模型。

### 3.4.1　PyTorch 的安装

PyTorch 支持 CPU 和 GPU 两种硬件类型，默认安装方式是 CPU 方式。使用国外镜像源的安装方式如下：

```
ngit@ubuntu:~$ pip install torch torchvision
```

也可以使用国内(清华)镜像源安装 PyTorch，安装速度更快，安装命令如下：

```
ngit@ubuntu:~$ pip install torch torchvision -i https://pypi.tuna.tsinghua.
edu.cn/simple/
```

若 PC 上已经安装 GPU 硬件，并已安装 NVIDIA 的通用并行计算架构 CUDA 和深度神经网络的 GPU 加速库 CUDNN，则可以安装 GPU 版 PyTorch。以 CUDA 11.3 版本为例，安装方式如下：

```
ngit@ubuntu:~$ pip install torch torchvision --extra-index-url https://
download.pytorch.org/whl/cu113
```

完成 PyTorch 安装之后，进入 PyThon 环境，检查 PyTorch 是否正确安装：

```
ngit@ubuntu:~$ python3
>>> import torch
```

若没有报错，则表示 PyTorch 安装成功。

### 3.4.2 PyTorch 的基本操作

**1. 创建一个张量**

张量(Tensor)是 PyTorch 里的基本运算单位，类似于 Numpy 的 ndarray，与 ndarray 最大的区别在于 Tensor 能使用 GPU 加速，而 ndarray 只能用在 CPU 上。其中 Tensor 与 Numpy 之间转换方法为：

将 Tensor 转换成 Numpy，调用 numpy()进行转换；

将 Numpy 转换成 Tensor，调用 torch.from_numpy()进行转换。

```
#1. 导入 torch 库
import torch
#2. 构建一个随机初始化的矩阵
a = torch.rand(3, 2)
print(a)
#3. 将 tensor 转换成 numpy
numpy_a = a.numpy()
print(numpy_a)
#4. 将 numpy 转换成 tensor
b = torch.from_numpy(numpy_a)
print(b)
```

输出结果如下：

```
tensor([[0.2232, 0.7406],                    #1. 随机初始化矩阵 a
        [0.7256, 0.3646],
        [0.8887, 0.6180]])
[[0.22317886 0.740585  ]                     #2. 将 tensor a 转换成 numpy 格式
 [0.72564596 0.36462945]
 [0.88874906 0.61796373]]
```

```
tensor([[0.2232, 0.7406],                    #3. 将 numpy 格式为 tensor
        [0.7256, 0.3646],
        [0.8887, 0.6180]])
```

**2. 自动微分和梯度**

Autograd 包为张量上的所有操作提供了自动求导。如果设置 requires_grad 为 True，那么将会追踪所有对于该张量的操作。当完成计算后通过调用 backward，自动计算所有的梯度，这个张量的所有梯度将会自动积累到 grad 中。

```
#1. 导入 torch 库
import torch
#2. 创建一个张量 x,设置 requires_grad=True,跟踪张量 x 上的计算
x = torch.ones(2, 2, requires_grad=True)
#3. 张量计算
y = x + 2
z = y * y * 3
out = z.mean()
#4. 简单的反向传播可以直接计算。复杂的则需要指定输入的值
out.backward()
print(x.grad)       #反向传播计算出的梯度值 d(out)/dx
```

输出结果为：

```
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
tensor([[4.5000, 4.5000],
        [4.5000, 4.5000]])
```

如果不需要在某个计算过程中累积梯度，可以调用 detach 方法将其与计算历史记录分离，并禁止跟踪将来的计算记录。在模型评估等过程中，不需要对变量进行梯度计算，可在代码外层使用 with torch.no_grad()。

**3. 神经网络包 nn**

Autograd 实现了反向传播功能，但是直接用来写深度学习的代码在很多情况下还是稍显复杂，torch.nn 是专门为神经网络设计的模块化接口，构建于 Autograd 之上，可用来定义和运行神经网络，包含了很多有用的功能：神经网络层、损失函数、优化器等。

nn.Module 是 nn 中最重要的类之一，用于构建神经网络，包含网络各层定义及 forward 方法，调用 forward(input)方法，可返回前向传播的结果。

nn 包下有几种不同的损失函数。例如，nn.MSELoss 用来计算均方误差，nn.CrossEntropyLoss 用来计算交叉熵损失。

torch.optim 实现了深度学习绝大多数的优化方法，如 SGD、Nesterov-SGD、Adam、RMSProp 等。

### 3.4.3　使用 PyTorch 实现手写数字识别

本节将通过 PyTorch 框架构建与 3.3.3 节 TensorFlow 实验中相同的神经网络模型，实

现 MNIST 手写字体识别，使读者能够对 PyTorch 的使用有一个大致的了解，快速上手。读者可以从 www.tup.com.cn 下载本实验的完整源代码。

**1. 加载 MNIST 数据集**

使用 torchvision 加载 MNIST。

```
import torch
import torch.utils.data as data
import torchvision.transforms as transforms
import torchvision
from tqdm import tqdm
#1. 配置数据格式转换，把数据从 Image 转为 torch.Tensor
transform = transforms.Compose([transforms.ToTensor()])
#2. 下载并加载训练集
train_dataset = torchvision.datasets.MNIST(root = './mnist/', train = True,
transform=transform, download=True)
#3. 下载并加载测试集
test_dataset = torchvision.datasets.MNIST(root = './mnist/', train = False,
transform=transform, download=False)
#4. 将训练集放到数据加载器中，设置样本批量 batch_size=128，shuffle=True 随机组建训
练批次，drop_last=True 当数据集不是 batch_size 的整倍数时，抛弃最后一组数据
train_loader = data.DataLoader(train_dataset, batch_size=128, shuffle=True,
drop_last=True)
#5. 将测试集放到数据加载器中
test_loader = data.DataLoader(test_dataset, batch_size=128, shuffle=False,
drop_last=False)
```

**2. 构建神经网络模型**

在 PyTorch 中可以使用 nn.Sequential 作为网络模块的容器，快速构建模型，模型会按照放入的顺序进行模型计算。还可以通过继承 nn.Module 实现自己的模型类。使用 nn.Sequential 定义神经网络的代码如下：

```
import torch.nn as nn
model = nn.Sequential(
        #1. 将输入图像数据从二维数组[28,28]展开为 784 维向量
        nn.Flatten(),
        #2. 全连接层，将 784 维向量映射为 128 维向量，使用 ReLU 函数激活
        nn.Linear(784,128),
        nn.ReLU(),
        #3. Dropout 层
        nn.Dropout(0.2),
        #4. 全连接层，将 128 维向量映射为 10 维向量输出，对应 10 个类别
        nn.Linear(128,10)
)
```

**3. 定义损失函数**

使用与 TensorFlow 实验中相同的交叉熵损失函数，相应的实现代码如下：

```
from torch.optim import lr_scheduler
#定义交叉熵损失函数
criterion = torch.nn.CrossEntropyLoss()
```

**4. 训练模型**

使用 Adam 优化器，初始学习率为 0.01，每个 epoch 对学习率进行衰减更新。代码如下：

```
#1. 定义优化器,定义初始学习率 0.01
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
#2. 定义学习率调整策略,每个 epoch 更新学习率,lr_new=lr_last * gamma
scheduler = lr_scheduler.ExponentialLR(optimizer, gamma=0.5)
#3. 训练 5 个 epoch
for epoch in range(5):
    total_correct = 0
    for images, labels in tqdm(train_loader):
        optimizer.zero_grad()
        #3.1 正向传播
        outputs = model(images)
        #3.2 统计预测正确的样本数
        _, predicted = torch.max(outputs.data, dim=1)
        total_correct += torch.eq(predicted, labels).sum().item()
        #3.3 计算损失
        loss = criterion(outputs, labels)
        #3.4 反向传播
        loss.backward()
        #3.5 权重更新
        optimizer.step()
    print('epoch:{} loss:{}'.format(epoch,loss))
    print('训练集正确率: {}'.format(total_correct/60000))
    scheduler.step()
```

输出结果为：

```
epoch:0 loss:0.11254709959030151 训练集正确率: 0.9163666666666667
epoch:1 loss:0.03629797324538231 训练集正确率: 0.9612
epoch:2 loss:0.1316099762916565  训练集正确率: 0.97155
epoch:3 loss:0.0818803608417511  训练集正确率: 0.9776333333333334
epoch:4 loss:0.1215408593416214  训练集正确率: 0.97985
```

**5. 评估模型**

由于不需要计算梯度，所以测试网络的代码在 torch.no_grad()下完成。代码如下：

```
#测试
total_correct = 0
with torch.no_grad():
    for images, labels in tqdm(test_loader):
        #正向传播
        outputs = model(images)
        #统计预测正确的样本数
        _, predicted = torch.max(outputs.data, dim=1)
        total_correct += torch.eq(predicted, labels).sum().item()
#预测正确率=预测正确样本数/总测试样本数
print('测试集正确率: {}'.format(total_correct/10000))
```