

# 《系统开发工具基础》实验报告



题目: 第四周学习报告

学号: 23020007139

姓名: 闫枳冰

专业: 计算机科学与技术

日期: 2024 年 9 月 15 日

# 目录

<b>1 实验目的</b>	<b>1</b>
1.1 练习记录	1
<b>2 实例</b>	<b>1</b>
2.1 实例一：使用 Linux 上的 journalctl 命令来获取最近一天中超级用户的登录信息及其所执行的指令。	1
2.2 实例二：安装 shellcheck 并尝试对下面的脚本进行检查。这段代码有什么问题吗？	1
2.3 实例三：请使用 cProfile 和 line_profiler 来比较插入排序和快速排序的性能。两种算法的瓶颈分别在哪里？然后使用 memory_profiler 来检查内存消耗，为什么插入排序更好一些？然后再看看原地排序版本的快排。	3
2.4 实例四：监听的端口已经被其他进程占用了。通过进程的 PID 查找相应的进程，并停止	3
2.5 实例五：执行 taskset -cpu-list 0,2 stress -c 3 并可视化。stress 占用了 3 个 CPU 吗？为什么没有？	4
2.6 实例六：将代码拷贝到文件中使其变为一个可执行的程序。首先安装 pycallgraph 和 graphviz。并使用 pycallgraph graphviz ./fib.py 来执行代码并查看 pycallgraph.png 这个文件。	6
2.7 实例七：构建系统	6
2.8 实例八：在上面的 makefile 中为 paper.pdf 实现一个 clean 目标。您需要将构建目标设置为 phony。	8
2.9 实例九：请编写一个 pre-commit 钩子，它会在提交前执行 make paper.pdf 并在出现构建失败的情况拒绝您的提交。	8
2.10 实例十：窗口管理器	9
2.11 实例十一：VPN	9
2.12 实例十二:Markdown	10
2.13 实例十三：Pytorch 定义网络	10
2.14 实例十四:Numpy 数组转换为 Tensor	10
2.15 实例十五：Tensors 可以通过.to 方法转换到不同的设备上	10
2.16 实例十六: 输出梯度	11
2.17 实例十七：损失函数	11
2.18 实例十八: 调试器	12
2.19 实例十九：反向传播	12
2.20 实例二十: 更新权重	14
<b>3 实验结果</b>	<b>14</b>
<b>4 解题感悟</b>	<b>14</b>

# 1 实验目的

1. 了解学习调试及性能分析。
2. 学习元编程演示实验。
3. 学习了解 PyTorch 编程。

## 1.1 练习记录

链接：<https://github.com/yyy0202/-remote-repo/tree/main/%E7%AC%AC%E5%9B%9B%E8%8A%82%E8%>

# 2 实例

## 2.1 实例一：使用 Linux 上的 journalctl 命令来获取最近一天中超级用户的登录信息及其所执行的指令。

如图所示1, 输入 journalctl 命令，输出用户的 · 登录信息。

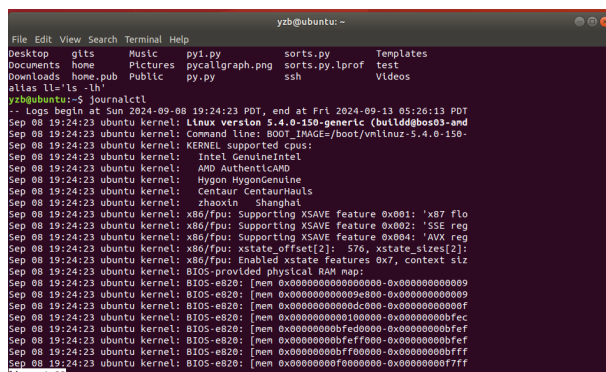


图 1: journalctl

## 2.2 实例二：安装 shellcheck 并尝试对下面的脚本进行检查。这段代码有什么问题吗？

代码如图2。

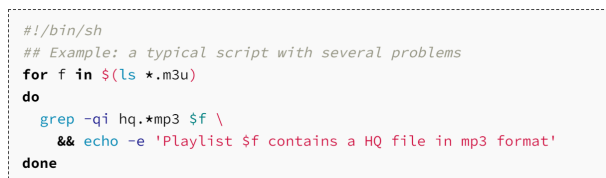


图 2: code

1. 首先输入命令 `sudo apt update` 更新软件包列表。
2. 输入命令 `sudo apt install shellcheck` 来安装 shellcheck。如图3
3. 输入 `shellcheck --version` 来验证 shellcheck 安装成功。如图4。
4. 将脚本文件写入 `test.sh`。如图5。
5. 输入命令 `shellcheck your_script.sh` 来使用 shellcheck。如图6 输出警告信息。

```

yzb@ubuntu:~$ sudo apt update
[sudo] password for yzb:
Hit:1 http://us.archive.ubuntu.com/ubuntu bionic InRelease
Hit:2 http://security.ubuntu.com/ubuntu bionic-security InRelease
Hit:3 http://us.archive.ubuntu.com/ubuntu bionic-updates InRelease
Hit:4 http://us.archive.ubuntu.com/ubuntu bionic-backports InRelease
Reading package lists... Done
Building dependency tree
Reading state information... Done
7 packages can be upgraded. Run 'apt list --upgradable' to see them.
yzb@ubuntu:~$ sudo apt install shellcheck
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  shellcheck
0 upgraded, 1 newly installed, 0 to remove and 7 not upgraded.
Need to get 1,841 kB of archives.
After this operation, 15.5 MB of additional disk space will be used.
Get:1 http://us.archive.ubuntu.com/ubuntu bionic/universe amd64 shellcheck amd64 0.4.6-1 [1,841 kB]
Fetched 1,841 kB in 2s (842 kB/s)
Selecting previously unselected package shellcheck.
(Reading database ... 163700 files and directories currently installed.)
Preparing to unpack .../shellcheck_0.4.6-1_amd64.deb ...
Unpacking shellcheck (0.4.6-1) ...
Setting up shellcheck (0.4.6-1) ...
Processing triggers for man-db (2.8.3-2ubuntu0.1) ...

```

图 3: shellcheck

```

yzb@ubuntu:~$ shellcheck --version
Shellcheck - shell script analysis tool
version: 0.4.6
license: GNU General Public License, version 3
website: http://www.shellcheck.net

```

图 4: shellcheck

```

File Edit View Search Terminal Help
yzb@ubuntu:~$ #!/bin/sh
## Example: a typical script with several problems
for f in $(ls *.mp3)
do
  grep -q1 hq.*mp3 $f \
  && echo -e 'Playlist $f contains a HQ file in mp3 format'
done

```

图 5: shellcheck

```

yzb@ubuntu:~$ vi test.sh
yzb@ubuntu:~$ shellcheck test.sh
In test.sh line 3:
for f in $(ls *.mp3)
^-- SC2045: Iterating over ls output is fragile. Use globs.
^-- SC2035: Use ./glob or --glob so names with dashes won't become options.

In test.sh line 5:
grep -q1 hq.*mp3 $f \
^-- SC2062: Quote the grep pattern so the shell won't interpret it.
^-- SC2086: Double quote to prevent globbing and word splitting.

In test.sh line 6:
&& echo -e 'Playlist $f contains a HQ file in mp3 format'
^-- SC2039: In POSIX sh, echo flags are undefined.
^-- SC2016: Expressions don't expand in single quotes, use double quotes for that.
yzb@ubuntu:~$

```

图 6: shellcheck3

SC2045: Iterating over ls output is fragile. Use globs. 指出使用 ls 命令的输出作为 for 循环的迭代源是不稳定的。原因是在文件名包含空格、换行符或特殊字符时，ls 的输出会被这些字符分割成多个“单词”，从而导致 for 循环无法正确处理这些文件名。

SC2035: Use ./glob or --glob so names with dashes won't become options. 是对 SC2045 的补充，特别提到了如果文件名以破折号 (-) 开头，它可能会被 shell 错误地解释为命令行选项。

SC2062: Quote the grep pattern so the shell won't interpret it. 指出 grep 的模式字符串没有被引号括起来。

SC2086: Double quote to prevent globbing and word splitting. 指出 \$f 没有被双引号括起来，这可能会导致 shell 在将文件名传递给 grep 之前对其进行单词拆分。

SC2039: In POSIX sh, echo flags are undefined. 指出 echo -e 中的 -e 选项在 POSIX 兼容的 shell（如 /bin/sh）中可能未定义。

SC2016: Expressions don't expand in single quotes, use double quotes for that. 指出在单引号

中的表达式不会被 shell 扩展在脚本中。

对代码进行修改后，不会报错，如图8和7。

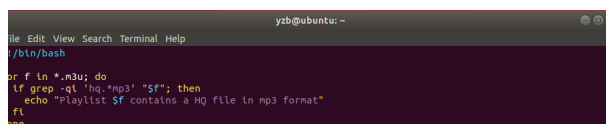


图 7: shellcheck

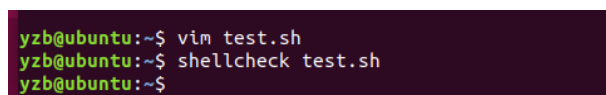


图 8: shellcheck

## 2.3 实例三：请使用 cProfile 和 line\_profiler 来比较插入排序和快速排序的性能。两种算法的瓶颈分别在哪里？然后使用 memory\_profiler 来检查内存消耗，为什么插入排序更好一些？然后再看看原地排序版本的快排。

首先使用命令 `pip install line_profiler` 安装 `line_profiler`。再在快排函数添加装饰器 `@profile`，执行 `kernprof -l -v sorts.py` 来对快排进行分析。同理对插入排序进行同样操作，如图9。在图中可以看到快速排序的耗时最长在于 `left` 和 `right` 的赋值，而插入排序的耗时最长在 `while` 循环。

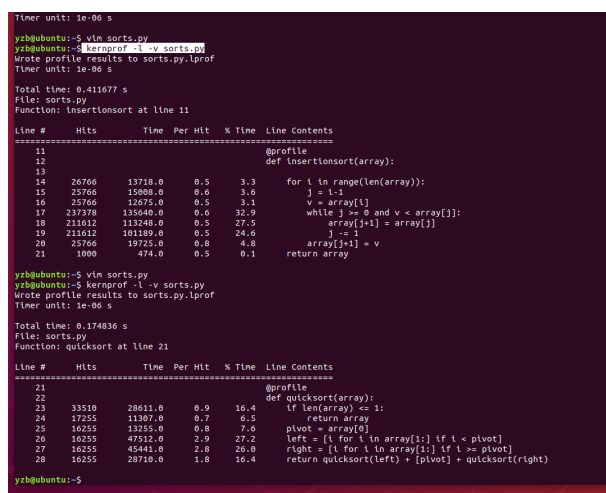


图 9: sort1

再使用 `pip install memory_profiler` 的命令安装 `memory_profiler`，再添加 `@profile` 装饰器，查看两者内存使用情况，可以得出快速排序比插入排序内存占用更大。如图10。

## 2.4 实例四：监听的端口已经被其他进程占用了。通过进程的 PID 查找相应的进程，并停止

首先执行 `python -m http.server 4444` 启动一个最简单的 web 服务器来监听 4444 端口。在另外一个终端中，执行 `lsof | grep LISTEN` 打印出所有监听端口的进程及相应的端口。找到对应的 PID 然后使用 `kill <PID>` 停止该进程。如图11。找到其 PID，输入 `kill PID`。结束进程12。

```
File Edit View Search Terminal Help
Desktop gits Music pycallgraph.png Templates
Documents none Pictures py.py test
Downloads home.pub Public ssh Videos
alias ll='ls -la'
yzb@ubuntu:~$ python -m memory_profiler sorts.py
#!/usr/bin/env python
# sorts.py
# Line # Mem usage Increment Line Contents
#-----
1 13.879 MiB 0.000 MiB import sys
2 13.879 MiB 0.000 MiB import random
3 13.879 MiB 0.000 MiB def sort(arr):
4 13.879 MiB 0.000 MiB     for i in range(len(arr)):
5 13.879 MiB 0.000 MiB         for j in range(i+1, len(arr)):
6 13.879 MiB 0.000 MiB             if arr[i] > arr[j]:
7 13.879 MiB 0.000 MiB                 arr[i], arr[j] = arr[j], arr[i]
8 13.879 MiB 0.000 MiB     return arr
9 13.879 MiB 0.000 MiB if __name__ == '__main__':
10 13.879 MiB 0.000 MiB     arr = [random.randint(0, 100) for i in range(1000)]
11 13.879 MiB 0.000 MiB     sort(arr)
12 13.879 MiB 0.000 MiB     sys.exit(0)
13 13.879 MiB 0.000 MiB
```

图 10: sorts

```
File Edit View Search Terminal Help
Desktop gits Music pycallgraph.png Templates
Documents none Pictures py.py test
Downloads home.pub Public ssh Videos
alias ll='ls -la'
yzb@ubuntu:~$ python -m http.server 4444
/usr/bin/python: No module named http
yzb@ubuntu:~$ python3 -m http.server 4444
Serving HTTP on 0.0.0.0 port 4444 (http://0.0.0.0:4444/) ...
yzb@ubuntu:~$ lsof | grep LISTEN
python3 46430 tcp* 0.0.0.0:4444 (LISTEN)
yzb@ubuntu:~$
```

图 11: kill

```
yzb@ubuntu:~$ kill 46430
yzb@ubuntu:~$
```

```
Serving HTTP on 0.0.0.0 port 4444 (http://0.0.0.0:4444/) ...
Terminated
yzb@ubuntu:~$
```

图 12: kill1

## 2.5 实例五：执行 taskset -cpu-list 0,2 stress -c 3 并可视化。stress 占用了 3 个 CPU 吗？为什么没有？

先输入命令 `sudo apt-get install htop`，下载 htop，再输入 htop 可视化<sup>13</sup>。再执行 `taskset -cpu-list 0,2 stress -c 3` 并可视化。如图15

原因：stress 的三个工作线程都试图在 CPU 0 和 CPU 2 上运行，但由于只有两个核心可用，它们会在这两个核心之间共享资源。taskset 命令可以将任务绑定到指定 CPU 核心。

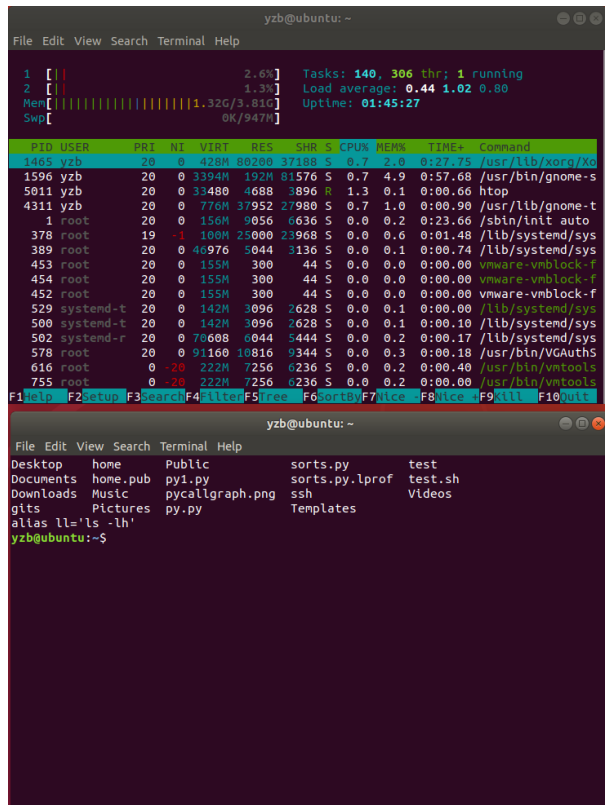


图 13: stress

输入命令 `stress -c 3`，如图14。

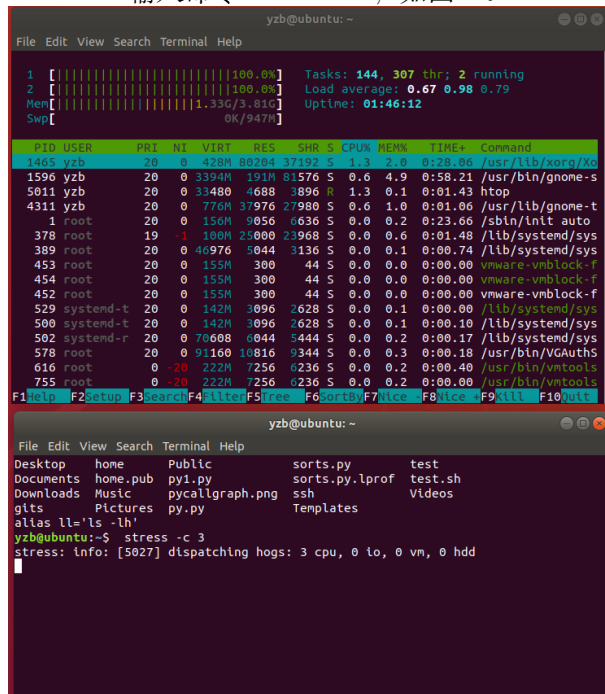


图 14: stress

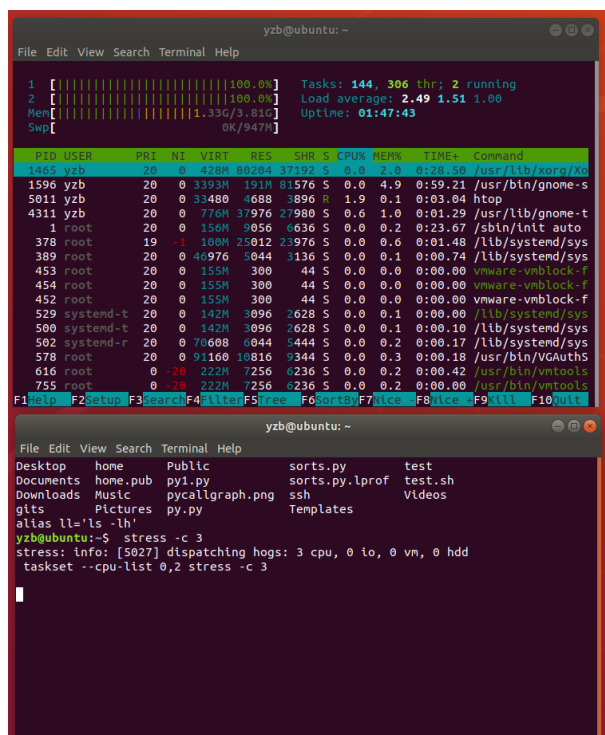


图 15: stress

2.6 实例六：将代码拷贝到文件中使其变为一个可执行的程序。首先安装 pycallgraph 和 graphviz。并使用 pycallgraph graphviz ./fib.py 来执行代码并查看 pycallgraph.png 这个文件。

代码如下：首先输入命令 pip install pycallgraph graphviz 来安装 pycallgraph 和 graphviz。再

```
#!/usr/bin/env python
def fib0(): return 0

def fib1(): return 1

s = """def fib(): return fib0() + fib1()"""

if __name__ == '__main__':
    for n in range(2, 10):
        exec(s.format(n, n-1, n-2))
        # from functools import lru_cache
        # for n in range(10):
        #     exec("fib{} = lru_cache(1)(fib{})".format(n, n))
    print(eval("fib9()"))
```

图 16: code

输入命令 pycallgraph graphviz - ./fib.py 生成一个名为 pycallgraph.png 的图片文件，再输入 eog pycallgraph.png，展示函数调用关系。如图所示17。

## 2.7 实例七：构建系统

首先建立名为 Makefile 的文件，在其中包含构建目标、相关依赖和规则。冒号左侧的是构建目标，冒号右侧的是构建它所需的依赖。缩进的部分是从依赖构建目标时需要用到的一段命令。18 然后创建 paper.tex 文件，ploy.py，data.dat 文件并在其中输入相关内容。输入命令 make 生成 PDF。





图 17: pycallgraph

```

paper.pdf: paper.tex plot-data.png
          pdflatex paper.tex

plot-%.png: %.dat plot.py
            ./plot.py -i $.dat -o $@

```

图 18: rule

在过程中遇到报错19, 输入命令 `pip install matplotlib`, 安装 `matplotlib`。再输入 `make`, 20。构建成

```

yzb@ubuntu:~$ make
./plot.py -i data.dat -o plot-data.png
Traceback (most recent call last):
  File "./plot.py", line 2, in <module>
    import matplotlib
ImportError: No module named matplotlib
Makefile:5: recipe for target 'plot-data.png' failed
make: *** [plot-data.png] Error 1
yzb@ubuntu:~$ pip install matplotlib

```

图 19: error

功



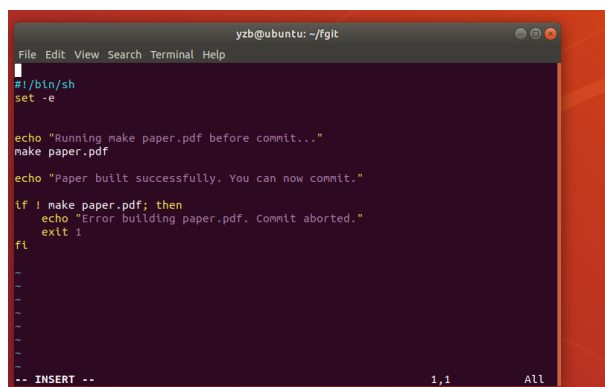


图 23: hook

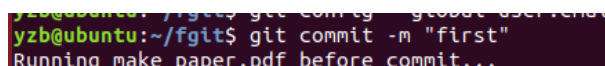


图 24: hook1

## 2.10 实例十：窗口管理器

平铺式管理器会把不同的窗口像贴瓷砖一样平铺在一起而不和其他窗口重叠。首先在虚拟机中下载 i325。再用 kill +PID 终端其他窗口管理器的使用。再输入 i3 使用 i3。

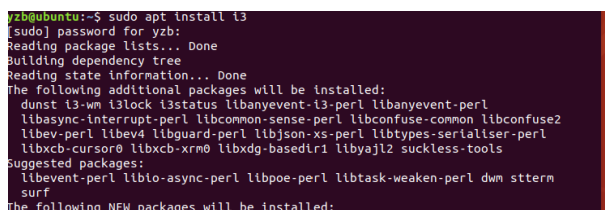


图 25: i3

## 2.11 实例十一：VPN

VPN 会在用户设备与目标服务器之间创建一个加密的隧道，用户的数据传输会被封装在加密的数据包中，通过这条隧道传输，从而保证数据的私密性和完整性。这种加密和隧道技术确保了数据在传输过程中不会被未经授权的第三方截获或篡改。<sup>26</sup>

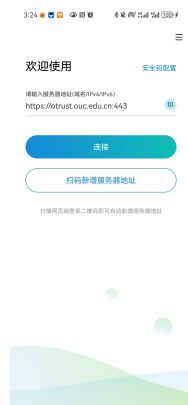


图 26: vpn

## 2.12 实例十二:Markdown

Markdown 是一种轻量级标记语言，它的设计初衷是让文档的编写和阅读变得更加简单和直观。了解 markdown 的初级用法。27



图 27: markdown

## 2.13 实例十三：Pytorch 定义网络

Net(nn.Module): 先定义了一个名为 Net 的类，它继承自 torch.nn.Module。在 \_\_init\_\_ 方法中，首先调用了 super(Net, self).\_\_init\_\_() 来执行父类 nn.Module 的初始化方法。接着，定义了网络中的各个层。forward 方法定义了数据通过网络的前向传播路径。最后，通过 net = Net() 实例化了一个 Net 对象，并打印出来以查看网络结构。如图所示28。

## 2.14 实例十四:Numpy 数组转换为 Tensor

转换的操作是调用 torch.from\_numpy(numpy\_array) 方法。29。

## 2.15 实例十五：Tensors 可以通过.to 方法转换到不同的设备上

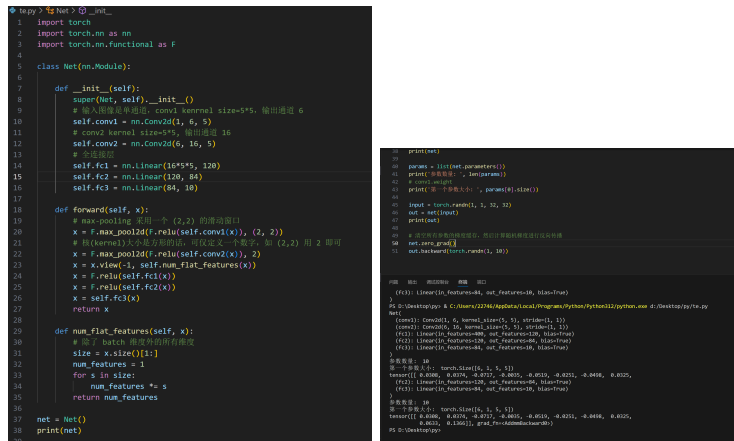


图 28: net

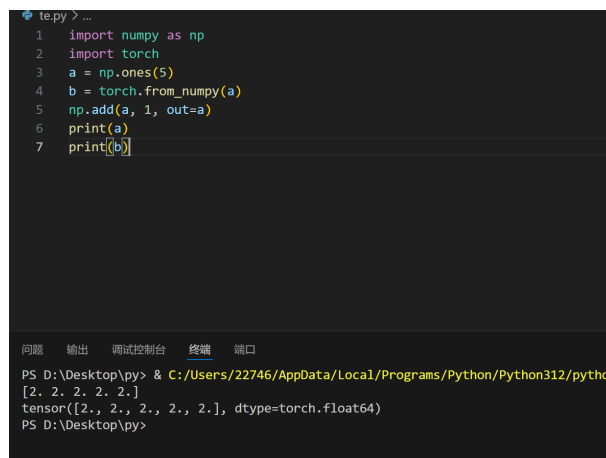


图 29: 转换



图 30: .to

## 2.16 实例十六: 输出梯度

首先, 定义了一个可训练的张量  $x$ , 其形状为  $2 \times 2$ , 并且设置 `requires_grad=True` 以便 PyTorch 能够追踪其上的所有操作, 从而计算梯度。然后, 再执行  $y = x + 2$ : 将  $x$  中的每个元素都加 2。 $z = y * y * 3$ : 将  $y$  中的每个元素先平方, 然后乘以 3。 $out = z.mean()$ : 计算  $z$  中所有元素的平均值。输出  $out$ 。当调用 `out.backward()` 时, PyTorch 会自动计算  $out$  关于  $x$  的梯度, 并将这个梯度存储在  $x.grad$  中。31

## 2.17 实例十七: 损失函数

先创建了一个随机生成的输入张量 `input`, 其形状为  $(1, 1, 32, 32)$ , 表示一个批次大小为 1, 具有 1 个通道, 高度和宽度均为 32 的图像。然后通过 `net` 来处理这个输入, 得到输出 `output`。再定

```
te.py > ...
1 import numpy as np
2 import torch
3 x = torch.ones(2, 2, requires_grad=True)
4 y=x+2
5 z = y * y * 3
6 out = z.mean()
7 out.backward()
8 # 输出梯度 d(out)/dx
9 print(x.grad)
```

```
PS D:\Desktop\py> & C:/Users/22746/AppData/Local/Programs/Python/Python312/python.exe d:/Desktop/py/te.py
[2. 2. 2. 2.]
tensor([2., 2., 2., 2.], dtype=torch.float64)
PS D:\Desktop\py> & C:/Users/22746/AppData/Local/Programs/Python/Python312/python.exe d:/Desktop/py/te.py
[2. 2. 2. 2.]
tensor([2., 2., 2., 2.], dtype=torch.float64)
PS D:\Desktop\py> & C:/Users/22746/AppData/Local/Programs/Python/Python312/python.exe d:/Desktop/py/te.py
tensor([[4.5000, 4.5000],
        [4.5000, 4.5000]])
PS D:\Desktop\py>
```

图 31: 梯度

义伪标签、调整 target 的大小、使用均方误差损失 (nn.MSELoss()) 来计算 output 和 target 之间的损失。32

```
43 input = torch.randn(1, 1, 32, 32)
44 output = net(input)
45 # 定义伪标签
46 target = torch.randn(10)
47 # 调整大小, 使得和 output 一样的 size
48 target = target.view(1, -1)
49 criterion = nn.MSELoss()
50
51 loss = criterion(output, target)
52 print(loss)
```

```
(int, int), int)
* (Tensor input, Tensor weight, Tensor bias = None, tuple of ints st
ride = 1, str padding = "valid", tuple of ints dilation = 1, int grou
ps = 1)
didn't match because some of the arguments have invalid types:
(built_function_or_method, Parameter, Parameter, tuple of (int, int
), tuple of (int, int), tuple of (int, int), int)
PS D:\Desktop\py> & C:/Users/22746/AppData/Local/Programs/Python/Pyth
on312/python.exe d:/Desktop/py/te.py
参数数量: 10
第一个参数大小: torch.Size([6, 1, 5, 5])
tensor(0.5425, grad_fn=<MseLossBackward0>)
PS D:\Desktop\py>
```

图 32: 损失函数

## 2.18 实例十八: 调试器

题目: 用 pdb 来修复下面的 Python 代码。33 通过命令 import pdb 导入 pdb 模块。在想要暂停执行的代码行前添加 pdb.set\_trace()。运行程序, 并使用 pdb 的命令来检查变量和执行流程。输入 l (list) 来查看当前行的代码, n (next) 来执行下一行代码, p arr (print) 来打印 arr 的值等。34.

## 2.19 实例十九: 反向传播

调用 loss.backward(), 先需要清空当前梯度缓存, 即.zero\_grad() 方法, 否则之前的梯度会累加到当前的梯度, 这样会影响权值参数的更新。

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(n):
            if arr[j] > arr[j+1]:
                arr[j] = arr[j+1]
                arr[j+1] = arr[j]
        return arr

print(bubble_sort([4, 2, 1, 8, 7, 6]))
```

图 33: code



图 34: 调试

以 conv1 层的偏置参数 bias 在反向传播前后的结果为例：35

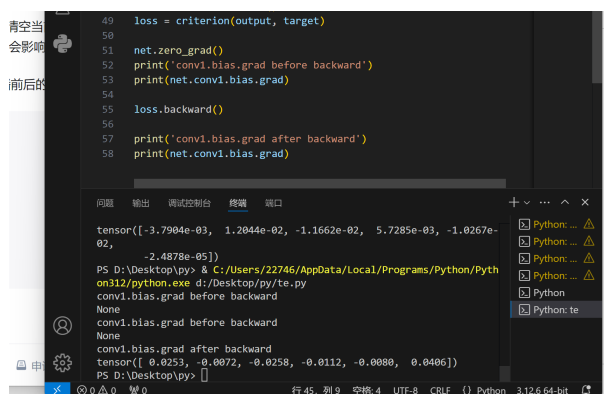


图 35: 反向

## 2.20 实例二十: 更新权重

采用随机梯度下降 (Stochastic Gradient Descent, SGD) 方法的最简单的更新权重规则:  $\text{weight} = \text{weight} - \text{learning\_rate} * \text{gradient}$ 。36

```
learning_rate = 0.01
for f in net.parameters():
    f.data.sub_([f.grad.data * learning_rate])
```

图 36: 权重

## 3 实验结果

1. 了解学习调试及性能分析。
2. 学习元编程演示实验。
3. 学习了解 PyTorch 编程。
4. 学习了“大杂烩”之中的一些内容。

链接: <https://github.com/yyy0202/-remote-repo/tree/main>

## 4 解题感悟

调试是软件开发过程中不可或缺的一环,它涉及到查找并修正代码中的错误。性能分析则关注于优化代码执行速度、减少资源消耗等,以确保软件运行的高效性。此次学习中我初步掌握了一些相关工具。同时我也通过编写简单的装饰器来实践元编程,理解其如何修改函数或类的行为。在大杂烩中,我了解到了 vpn、markdown、窗口管理等知识。同时我也初步学习了 pytorch,通过这些学习,我认识到自己需要学习的知识还有很多,需要不断努力。