

# 系统基础开发工具

## 实验二

姓 名: 陈 佳 玲

学 号: 23100021002

专 业: 23 级环境工程

2025 年 9 月 22 日

## 目录

<b>1</b>	<b>实验目的</b>	<b>3</b>
<b>2</b>	<b>实验内容</b>	<b>3</b>
2.1	调试与性能分析 . . . . .	3
2.1.1	获取超级用户登录信息及所执行的指令 . . . . .	3
2.1.2	打印调试法与日志 . . . . .	3
2.1.3	python 调试器——pdb . . . . .	9
2.1.4	性能分析——计时 . . . . .	9
2.2	元编程 . . . . .	10
2.2.1	构建系统 . . . . .	10
2.2.2	类装饰器 . . . . .	12
2.3	Pytorch . . . . .	12
2.3.1	PyTorch 张量 . . . . .	12
2.3.2	张量形状操作 . . . . .	14
2.3.3	张量运算的三种实现方法 . . . . .	15
2.3.4	自动求导 . . . . .	16
2.3.5	Tensor 与 Numpy 数组的相互转换 . . . . .	16
2.3.6	自动梯度计算 . . . . .	18
2.3.7	自定义简单神经网络 . . . . .	19
2.3.8	常用损失函数的使用 . . . . .	21
2.3.9	反向传播流程实践 . . . . .	23
2.3.10	优化器与权重更新 . . . . .	25

2.3.11	CIFAR10 数据加载和预处理 . . . . .	26
2.3.12	卷积操作 . . . . .	27
2.3.13	卷积神经网络简单实现 . . . . .	28
2.3.14	测试代码以及可视化结果 . . . . .	28
<b>3</b>	<b>心得体会</b>	<b>29</b>
<b>4</b>	<b>实验代码查看链接</b>	<b>29</b>

## 1 实验目的

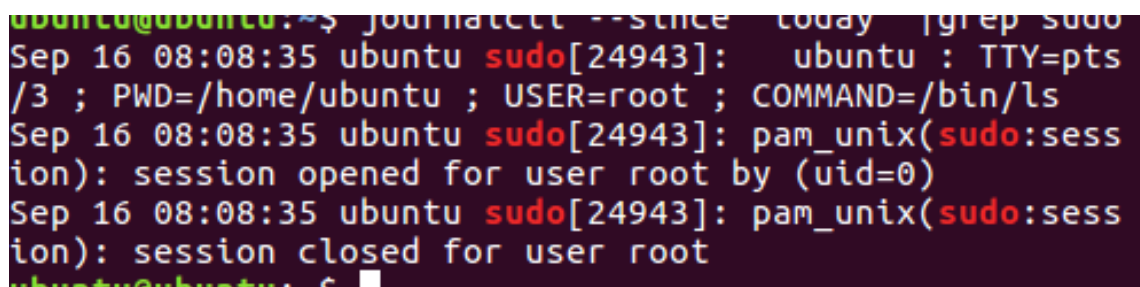
1. 掌握 Python 程序调试的基本方法和工具，学习性能分析和优化的技术
2. 理解元编程的概念和应用场景，掌握 Python 中代码生成和动态执行的技巧
3. 掌握 PyTorch 框架的基本操作和核心概念，理解张量运算和自动求导机制，学会构建和训练神经网络模型

## 2 实验内容

### 2.1 调试与性能分析

#### 2.1.1 获取超级用户登录信息及所执行的指令

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



```
ubuntu@ubuntu:~$ journalctl --since today |grep sudo
Sep 16 08:08:35 ubuntu sudo[24943]:    ubuntu : TTY=pts
/3 ; PWD=/home/ubuntu ; USER=root ; COMMAND=/bin/ls
Sep 16 08:08:35 ubuntu sudo[24943]: pam_unix(sudo:session): session opened for user root by (uid=0)
Sep 16 08:08:35 ubuntu sudo[24943]: pam_unix(sudo:session): session closed for user root
ubuntu@ubuntu:~$
```

图 1: 获取超级用户登录信息

#### 2.1.2 打印调试法与日志

创建一个日志文件 `logger.py` 执行不同的指令使得输出不同的结果信息。

```

import logging
import sys

class CustomFormatter(logging.Formatter):
    """Logging Formatter to add colors and count warning / errors"""

    grey = "\x1b[38;21m"
    yellow = "\x1b[33;21m"
    red = "\x1b[31;21m"
    bold_red = "\x1b[31;1m"
    reset = "\x1b[0m"
    format = "%(asctime)s - %(name)s - %(levelname)s - %(message)s (%(filename)s:%(lineno)
d)"

    FORMATS = {
        logging.DEBUG: grey + format + reset,
        logging.INFO: grey + format + reset,
        logging.WARNING: yellow + format + reset,
        logging.ERROR: red + format + reset,
        logging.CRITICAL: bold_red + format + reset
    }

    def format(self, record):
        log_fmt = self.FORMATS.get(record.levelno)
        formatter = logging.Formatter(log_fmt)
        return formatter.format(record)

# create logger with 'spam_application'
logger = logging.getLogger("Sample")

# create console handler with a higher log level
ch = logging.StreamHandler()
ch.setLevel(logging.DEBUG)

if len(sys.argv) > 1:
    if sys.argv[1] == 'log':
        ch.setFormatter(logging.Formatter('%(asctime)s : %(levelname)s : %(name)s :
%(message)s'))
    elif sys.argv[1] == 'color':
        ch.setFormatter(CustomFormatter())

if len(sys.argv) > 2:
    logger.setLevel(logging.__getattribute__(sys.argv[2]))
else:
    logger.setLevel(logging.DEBUG)

logger.addHandler(ch)

```

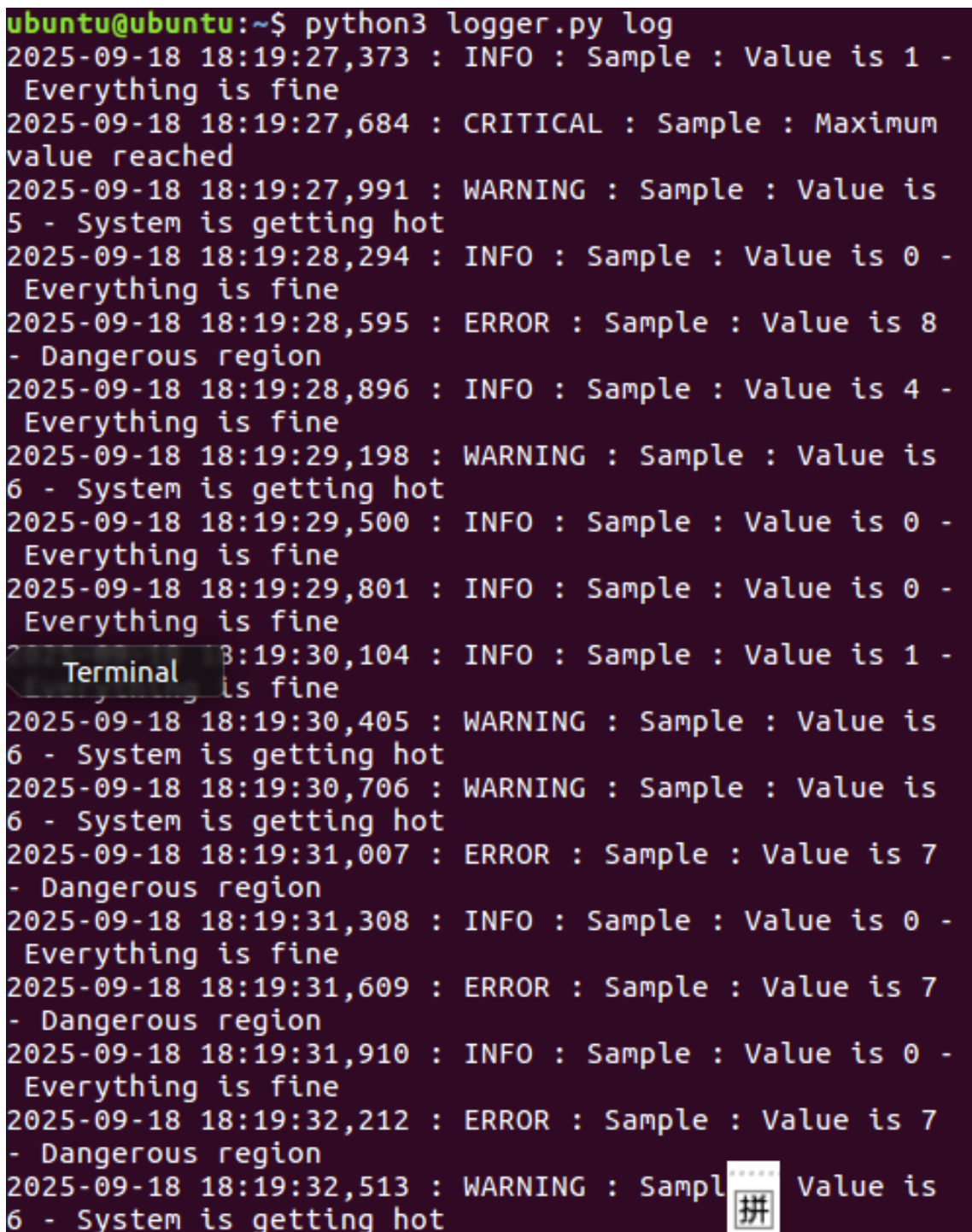
图 2: logger.py

1. 直接使用 print() 语句输出的内容，没有经过任何日志格式化或级别过滤。

```
ubuntu@ubuntu:~$ vim logger.py
ubuntu@ubuntu:~$ python3 logger.py
Value is 1 - Everything is fine
Value is 3 - Everything is fine
Value is 8 - Dangerous region
Value is 8 - Dangerous region
Value is 8 - Dangerous region
Value is 1 - Everything is fine
Value is 2 - Everything is fine
Value is 3 - Everything is fine
Value is 2 - Everything is fine
Value is 1 - Everything is fine
Value is 8 - Dangerous region
Value is 4 - Everything is fine
Maximum value reached
Maximum value reached
Value is 2 - Everything is fine
Value is 4 - Everything is fine
Value is 0 - Everything is fine
Maximum value reached
Value is 5 - System is getting hot
Value is 3 - Everything is fine
Value is 3 - Everything is fine
Value is 0 - Everything is fine
Value is 3 - Everything is fine
Maximum value reached
Value is 8 - Dangerous region
Value is 0 - Everything is fine
Maximum value reached
Value is 4 - Everything is fine
Value is 5 - System is getting hot
Value is 3 - Everything is fine
Value is 4 - Everything is fine
Value is 3 - Everything is fine
Value is 0 - Everything is fine
Maximum value reached
Value is 1 - Everything is fine
Value is 6 - System is getting hot
Maximum value reached
Value is 7 - Dangerous region
```

图 3: 无格式化

2. 仅经过日志格式化 (Log formatted output) 的输出。通常会包含时间戳、日志级别 (如 INFO、ERROR)、模块名等信息, 结构更清晰, 易于后续分析。



```
ubuntu@ubuntu:~$ python3 logger.py log
2025-09-18 18:19:27,373 : INFO : Sample : Value is 1 -
Everything is fine
2025-09-18 18:19:27,684 : CRITICAL : Sample : Maximum
value reached
2025-09-18 18:19:27,991 : WARNING : Sample : Value is
5 - System is getting hot
2025-09-18 18:19:28,294 : INFO : Sample : Value is 0 -
Everything is fine
2025-09-18 18:19:28,595 : ERROR : Sample : Value is 8
- Dangerous region
2025-09-18 18:19:28,896 : INFO : Sample : Value is 4 -
Everything is fine
2025-09-18 18:19:29,198 : WARNING : Sample : Value is
6 - System is getting hot
2025-09-18 18:19:29,500 : INFO : Sample : Value is 0 -
Everything is fine
2025-09-18 18:19:29,801 : INFO : Sample : Value is 0 -
Everything is fine
2025-09-18 18:19:30,104 : INFO : Sample : Value is 1 -
Everything is fine
2025-09-18 18:19:30,405 : WARNING : Sample : Value is
6 - System is getting hot
2025-09-18 18:19:30,706 : WARNING : Sample : Value is
6 - System is getting hot
2025-09-18 18:19:31,007 : ERROR : Sample : Value is 7
- Dangerous region
2025-09-18 18:19:31,308 : INFO : Sample : Value is 0 -
Everything is fine
2025-09-18 18:19:31,609 : ERROR : Sample : Value is 7
- Dangerous region
2025-09-18 18:19:31,910 : INFO : Sample : Value is 0 -
Everything is fine
2025-09-18 18:19:32,212 : ERROR : Sample : Value is 7
- Dangerous region
2025-09-18 18:19:32,513 : WARNING : Sample : Value is
6 - System is getting hot
```

图 4: 日志格式化

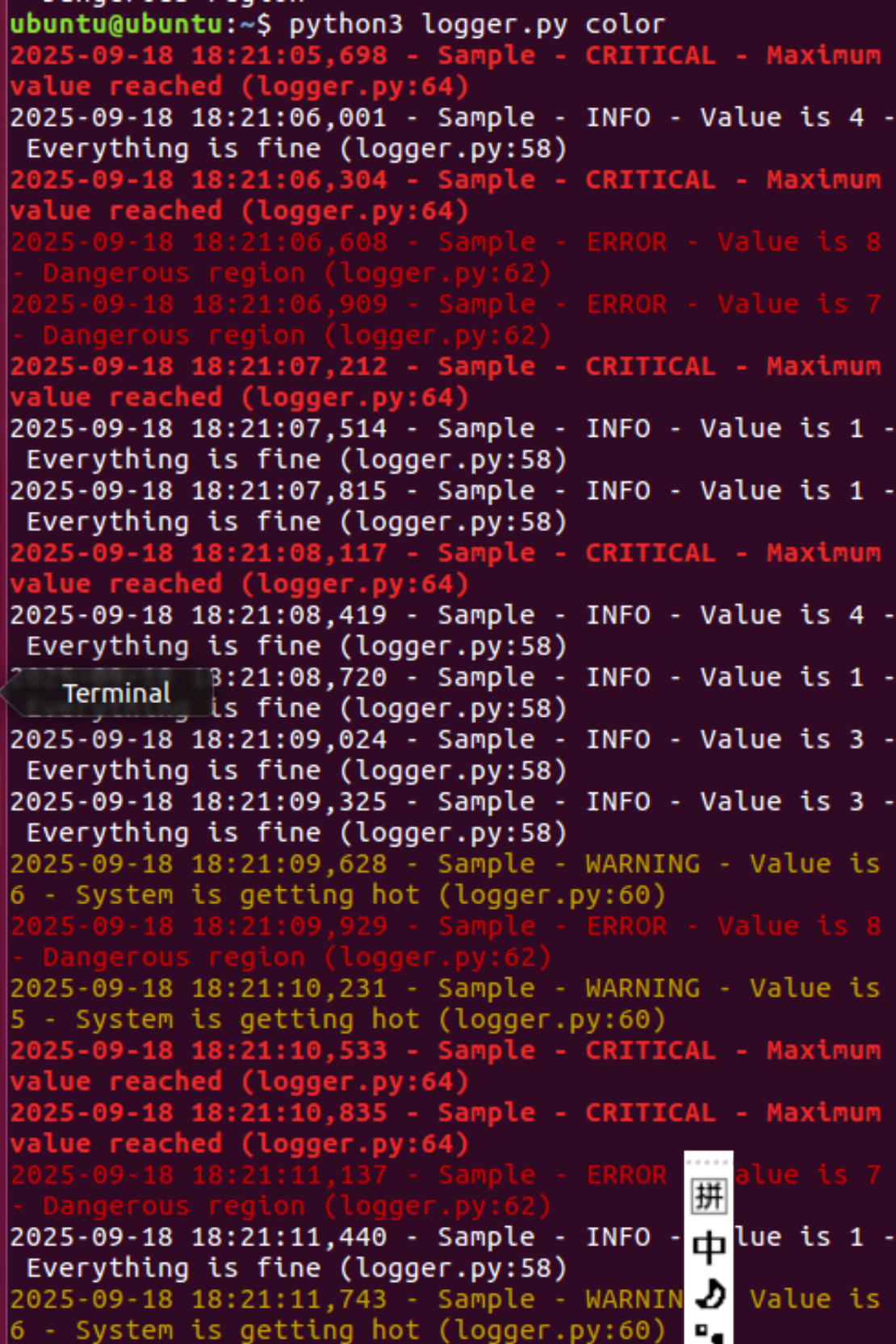
3. 仅输出日志级别为 ERROR 及以上 (如 ERROR、CRITICAL) 的信息。过滤掉较低级别的日志 (如 DEBUG、INFO、WARNING), 只显示错误和严重错误信息, 便于快速定位问题

```
ubuntu@ubuntu:~$ python3 logger.py log ERROR
2025-09-18 18:20:23,127 : ERROR : Sample : Value is 7
- Dangerous region
2025-09-18 18:20:23,733 : CRITICAL : Sample : Maximum
value reached
2025-09-18 18:20:25,543 : ERROR : Sample : Value is 7
- Dangerous region
2025-09-18 18:20:26,146 : CRITICAL : Sample : Maximum
value reached
2025-09-18 18:20:26,448 : ERROR : Sample : Value is 7
- Dangerous region
2025-09-18 18:20:28,253 : ERROR : Sample : Value is 8
- Dangerous region
2025-09-18 18:20:28,855 : ERROR : Sample : Value is 8
- Dangerous region
2025-09-18 18:20:29,156 : CRITICAL : Sample : Maximum
value reached
2025-09-18 18:20:29,457 : CRITICAL : Sample : Maximum
value reached
2025-09-18 18:20:30,058 : ERROR : Sample : Value is 7
- Dangerous region
2025-09-18 18:20:30,360 : ERROR : Sample : Value is 8
- Dangerous region
2025-09-18 18:20:30,661 : CRITICAL : Sample : Maximum
value reached
System Settings 2025-09-18 18:20:30,963 : CRITICAL : Sample : Maximum
value reached
2025-09-18 18:20:31,566 : ERROR : Sample : Value is 8
- Dangerous region
2025-09-18 18:20:32,168 : ERROR : Sample : Value is 7
- Dangerous region
2025-09-18 18:20:32,470 : ERROR : Sample : Value is 8
- Dangerous region
2025-09-18 18:20:33,074 : CRITICAL : Sample : Maximum
value reached
2025-09-18 18:20:34,579 : ERROR : Sample : Value is 7
- Dangerous region
2025-09-18 18:20:34,881 : ERROR : Sample : Value is 8
- Dangerous region
2025-09-18 18:20:36,986 : ERROR : Sample : Value is 7
- Dangerous region
2025-09-18 18:20:38,794 : CRITICAL : Sample : Maximum
```

图 5: 输出日志级别 ERROR 以上的

4. : 彩色格式化输出 (Color formatted output). 不同级别的日志会以不同颜色显示 (如 ERROR 用红色, WARNING 用黄色), 增强可读性, 适合在终端中实时查看





```
ubuntu@ubuntu:~$ python3 logger.py color
2025-09-18 18:21:05,698 - Sample - CRITICAL - Maximum
value reached (logger.py:64)
2025-09-18 18:21:06,001 - Sample - INFO - Value is 4 -
Everything is fine (logger.py:58)
2025-09-18 18:21:06,304 - Sample - CRITICAL - Maximum
value reached (logger.py:64)
2025-09-18 18:21:06,608 - Sample - ERROR - Value is 8
- Dangerous region (logger.py:62)
2025-09-18 18:21:06,909 - Sample - ERROR - Value is 7
- Dangerous region (logger.py:62)
2025-09-18 18:21:07,212 - Sample - CRITICAL - Maximum
value reached (logger.py:64)
2025-09-18 18:21:07,514 - Sample - INFO - Value is 1 -
Everything is fine (logger.py:58)
2025-09-18 18:21:07,815 - Sample - INFO - Value is 1 -
Everything is fine (logger.py:58)
2025-09-18 18:21:08,117 - Sample - CRITICAL - Maximum
value reached (logger.py:64)
2025-09-18 18:21:08,419 - Sample - INFO - Value is 4 -
Everything is fine (logger.py:58)
2025-09-18 18:21:08,720 - Sample - INFO - Value is 1 -
Everything is fine (logger.py:58)
2025-09-18 18:21:09,024 - Sample - INFO - Value is 3 -
Everything is fine (logger.py:58)
2025-09-18 18:21:09,325 - Sample - INFO - Value is 3 -
Everything is fine (logger.py:58)
2025-09-18 18:21:09,628 - Sample - WARNING - Value is
6 - System is getting hot (logger.py:60)
2025-09-18 18:21:09,929 - Sample - ERROR - Value is 8
- Dangerous region (logger.py:62)
2025-09-18 18:21:10,231 - Sample - WARNING - Value is
5 - System is getting hot (logger.py:60)
2025-09-18 18:21:10,533 - Sample - CRITICAL - Maximum
value reached (logger.py:64)
2025-09-18 18:21:10,835 - Sample - CRITICAL - Maximum
value reached (logger.py:64)
2025-09-18 18:21:11,137 - Sample - ERROR - Value is 7
- Dangerous region (logger.py:62)
2025-09-18 18:21:11,440 - Sample - INFO - Value is 1 -
Everything is fine (logger.py:58)
2025-09-18 18:21:11,743 - Sample - WARNING - Value is
6 - System is getting hot (logger.py:60)
```

图 6: 彩色格式化输出



### 2.1.3 python 调试器——pdb

pdb 是 Python 自带的一个强大的交互式源代码调试器。它允许你在程序任意位置设置断点、单步执行、查看变量值、查看调用栈，甚至动态修改变量和执行代码，是排查复杂 Bug 的终极利器。

```

ubuntu@ubuntu:~$ vim test.py
ubuntu@ubuntu:~$ python -m pdb test.py
> /home/ubuntu/test.py(1)<module>()
-> def bubble_sort(arr):
(Pdb) l
1  -> def bubble_sort(arr):
2      n = len(arr)
3      for i in range(n):
4          for j in range(n):
5              if arr[j] > arr[j+1]:
6                  arr[j] = arr[j+1]
7                  arr[j+1] = arr[j]
8      arr
9
10     print(bubble_sort([4, 2, 1, 8, 7, 6]))
[EOF]
(Pdb) c
Traceback (most recent call last):
  File "/usr/lib/python2.7/pdb.py", line 1314, in main
    pdb._runscript(mainpyfile)
  File "/usr/lib/python2.7/pdb.py", line 1233, in _runscript
    self.run(statement)
  File "/usr/lib/python2.7/bdb.py", line 400, in run
    exec cmd in globals, locals
  File "<string>", line 1, in <module>
  File "test.py", line 1, in <module>
    def bubble_sort(arr):
  File "test.py", line 5, in bubble_sort
    if arr[j] > arr[j+1]:
IndexError: list index out of range
Uncaught exception. Entering post mortem debugging
Running 'cont' or 'step' will restart the program
> /home/ubuntu/test.py(5)bubble_sort()
-> if arr[j] > arr[j+1]:
(Pdb) p n
6
(Pdb) p j
5
(Pdb) p arr
[2, 1, 1, 7, 6, 6]
(Pdb) q
Post mortem debugger finished. The test.py will be restarted
> /home/ubuntu/test.py(1)<module>()
-> def bubble_sort(arr):
(Pdb) q
ubuntu@ubuntu:~$ vim test.py
ubuntu@ubuntu:~$ python test.py
[1, 1, 1, 6, 6, 6]
ubuntu@ubuntu:~$ vim test2.py

```

运行指令，调用pdb调试器

查看原错误代码

发现是由于数组越界问题导致运行报错

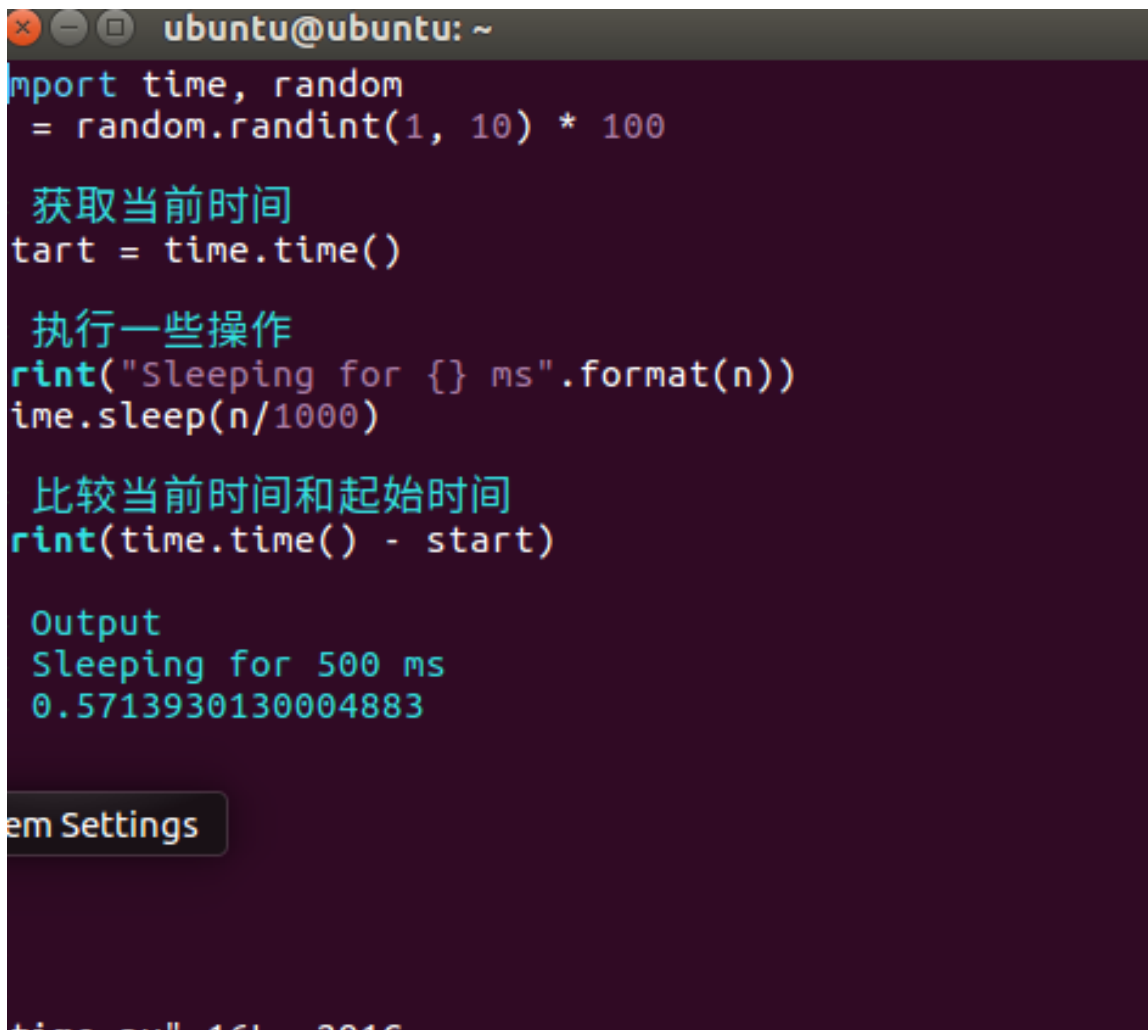
修改将j的范围改为0至n-i-1

修改后运行成功

图 7: 调试器 pdb 的操作

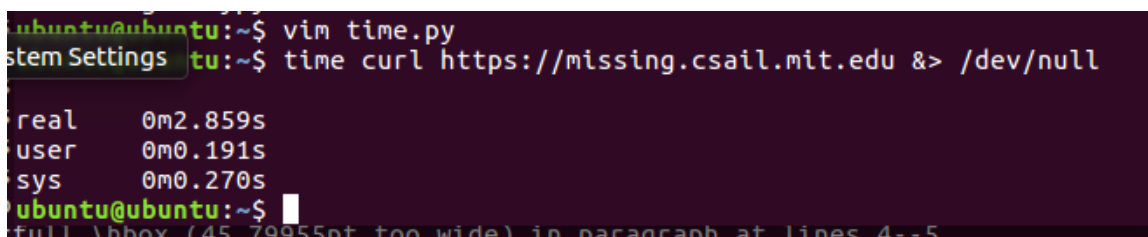
### 2.1.4 性能分析——计时

Python 的 time 模块：(1) 真实时间 Real - 从程序开始到结束流失掉的真实时间，包括其他进程的执行时间以及阻塞消耗的时间（例如等待 I/O 或网络）// (2) 用户时间 User - CPU 执行用户代码所花费的时间// (3) 系统时间 Sys - CPU 执行系统内核代码所花费的时间。

A terminal window titled 'ubuntu@ubuntu: ~' with a dark purple background. It contains Python code for time analysis. The code includes comments in Chinese: '获取当前时间' (Get current time), '执行一些操作' (Execute some operations), and '比较当前时间和起始时间' (Compare current time and start time). The code uses the 'time' and 'random' modules to generate a random sleep duration and measure the time taken. The output shows 'Sleeping for 500 ms' and a time value of '0.5713930130004883'. A 'System Settings' button is visible at the bottom left.

```
ubuntu@ubuntu: ~  
import time, random  
n = random.randint(1, 10) * 100  
  
    获取当前时间  
start = time.time()  
  
    执行一些操作  
print("Sleeping for {} ms".format(n))  
time.sleep(n/1000)  
  
    比较当前时间和起始时间  
print(time.time() - start)  
  
Output  
Sleeping for 500 ms  
0.5713930130004883  
  
System Settings
```

图 8: 时间分析代码

A terminal window showing the execution of a time analysis command. The user runs 'time curl https://missing.csail.mit.edu &> /dev/null'. The output shows the real, user, and system time taken for the curl command to complete. The terminal has a dark purple background and a 'System Settings' button on the left.

```
ubuntu@ubuntu:~$ vim time.py  
System Settings ubuntu@ubuntu:~$ time curl https://missing.csail.mit.edu &> /dev/null  
  
real    0m2.859s  
user    0m0.191s  
sys     0m0.270s  
ubuntu@ubuntu:~$
```

图 9: 时间分析操作

## 2.2 元编程

### 2.2.1 构建系统

`make` 是最常用的构建系统之一，它通常被安装到了几乎所有基于 UNIX 的系统中。当执行 `make` 时，它会去参考当前目录下名为 `Makefile` 的文件。

```
cat paper.tex: command not found
ubuntu@ubuntu:~$ cat paper.tex
\documentclass{article}
\usepackage{graphicx}
\begin{document}
\includegraphics[scale=0.65]{plot-data.png}
\end{document}
ubuntu@ubuntu:~$ cat plot.py
#!/usr/bin/env python3
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('-i', type=argparse.FileType('r'))
parser.add_argument('-o')
args = parser.parse_args()

data = np.loadtxt(args.i)
plt.plot(data[:, 0], data[:, 1])
plt.savefig(args.o)
ubuntu@ubuntu:~$ cat data.dat
1 1
2 2
3 3
4 4
5 8
ubuntu@ubuntu:~$ make
make: 'paper.pdf' is up to date.
ubuntu@ubuntu:~$
```

图 10: make 构建系统简单操作

### 2.2.2 类装饰器

```
class ClassDecorator:
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        print("类装饰器前")
        result = self.func(*args, **kwargs)
        print("类装饰器后")
        return result

@ClassDecorator
def say_hi():
    print("Hi!")
```

图 11: 类装饰器

## 2.3 Pytorch

### 2.3.1 PyTorch 张量

张量是一个多维数组，可以是标量、向量、矩阵或更高维度的数据结构。在 PyTorch 中，张量是数据的核心表示形式，类似于 NumPy 的多维数组，但具有更强大的功能，例如支持 GPU 加速和自动梯度计算。张量支持多种数据类型（整型、浮点型、布尔型等）。张量可以存储在 CPU 或 GPU 中，GPU 张量可显著加速计算。

```
import torch

tensor = torch.tensor([1, 2, 3])
print(tensor)

tensor_2d = torch.tensor([
    [-9, 4, 2, 5, 7],
    [3, 0, 12, 8, 6],
    [1, 23, -6, 45, 2],
    [22, 3, -1, 72, 6]
])
print("2D Tensor (Matrix):\n", tensor_2d)
print("Shape:", tensor_2d.shape) # 形状

# 创建 3D 张量 (立方体)
tensor_3d = torch.stack([tensor_2d, tensor_2d + 10, tensor_2d - 5]) # 堆叠 3 个 2D 张量
print("3D Tensor (Cube):\n", tensor_3d)
print("Shape:", tensor_3d.shape) # 形状
```

图 12: 张量的声明与定义

```
(pytorch_env) PS C:\Users\71883> python test.py
tensor([1, 2, 3])
2D Tensor (Matrix):
  tensor([[ -9,  4,  2,  5,  7],
         [ 3,  0, 12,  8,  6],
         [ 1, 23, -6, 45,  2],
         [22,  3, -1, 72,  6]])
Shape: torch.Size([4, 5])
3D Tensor (Cube):
  tensor([[[[ -9,  4,  2,  5,  7],
            [ 3,  0, 12,  8,  6],
            [ 1, 23, -6, 45,  2],
            [22,  3, -1, 72,  6]]],
         [[[ 1, 14, 12, 15, 17],
            [13, 10, 22, 18, 16],
            [11, 33,  4, 55, 12],
            [32, 13,  9, 82, 16]]],
         [[[-14, -1, -3,  0,  2],
            [-2, -5,  7,  3,  1],
            [-4, 18, -11, 40, -3],
            [17, -2, -6, 67,  1]]]])
Shape: torch.Size([3, 4, 5])
```

图 13: 张量的创建结果

## 2.3.2 张量形状操作

```

# 改变形状
x = torch.randn(2, 3)
print(f"原始形状: {x.shape}")
print(f"原始张量:\n{x}")

y = x.view(3, 2)
print(f"view重塑后形状: {y.shape}")

z = x.reshape(3, 2)
print(f"reshape重塑后形状: {z.shape}")

flattened = x.flatten()
print(f"展平后: {flattened}, 形状: {flattened.shape}")

# 转置和维度操作
transposed = x.t()
print(f"转置后形状: {transposed.shape}")

permuted = x.permute(1, 0)
print(f"维度重排后形状: {permuted.shape}")

# 增加/减少维度
unsqueezeed = x.unsqueeze(0)
print(f"增加维度后形状: {unsqueezeed.shape}")

squeezed = unsqueezeed.squeeze(0)
print(f"去除维度后形状: {squeezed.shape}")

```

图 14: 改变张量形状

```

=== 张量形状操作 ===
原始形状: torch.Size([2, 3])
原始张量:
tensor([[ 0.0747, -0.8760, -0.4961],
        [-0.0040, -0.3056, -0.7150]])
view重塑后形状: torch.Size([3, 2])
reshape重塑后形状: torch.Size([3, 2])
展平后: tensor([ 0.0747, -0.8760, -0.4961, -0.0040, -0.3056, -0.7150]),
形状: torch.Size([6])
转置后形状: torch.Size([3, 2])
维度重排后形状: torch.Size([3, 2])
增加维度后形状: torch.Size([1, 2, 3])
去除维度后形状: torch.Size([2, 3])

```

图 15: 改变后结果



### 2.3.3 张量运算的三种实现方法

#### 1. 三种运算方式的区别

- (1) 运算符方式:  $a + b$  - 最直观, 创建新张量
- (2) 函数方式: `torch.add(a, b)` - 功能更丰富, 可指定输出张量
- (3) 原地操作: `a.add(b)` -

```
(pytorch_env) PS C:\Users\71883> python test.py
原始张量a:
tensor([[1., 2.],
        [3., 4.]])
原始张量b:
tensor([[5., 6.],
        [7., 8.]])

=== 方法1: 直接运算符 ===
a + b =
tensor([[ 6.,  8.],
        [10., 12.]])

=== 方法2: torch.add函数 ===
torch.add(a, b) =
tensor([[ 6.,  8.],
        [10., 12.]])

=== 方法3: 原地操作 ===
执行原地操作前a的id: 1706444490128
执行原地操作后a:
tensor([[ 6.,  8.],
        [10., 12.]])
执行原地操作后a的id: 1706444490128

=== 结果验证 ===
方法1和方法2结果是否相同: True
```

图 17: 运算结果

## 2.3.4 自动求导

```

# 单变量求导
x = torch.tensor(2.0, requires_grad=True)
print(f"x = {x}")

y = x**2 + 3*x + 1
print(f"y = x2 + 3x + 1 = {y}")

y.backward() # 反向传播
print(f"dy/dx = {x.grad}") # 2*2 + 3 = 7.0

# 多变量求导
x1 = torch.tensor(1.0, requires_grad=True)
x2 = torch.tensor(2.0, requires_grad=True)
print(f"\nx1 = {x1}, x2 = {x2}")

y = x1**2 + x2**3
print(f"y = x12 + x23 = {y}")

y.backward()
print(f"∂y/∂x1 = {x1.grad}") # 2*1 = 2.0
print(f"∂y/∂x2 = {x2.grad}") # 3*22 = 12.0

```

图 18: 自动求导方法

```

x = 2.0
y = x2 + 3x + 1 = 11.0
dy/dx = 7.0

x1 = 1.0, x2 = 2.0
y = x12 + x23 = 9.0
∂y/∂x1 = 2.0
∂y/∂x2 = 12.0

```

图 19: 运行效果图

## 2.3.5 Tensor 与 Numpy 数组的相互转换

PyTorch 张量和 NumPy 数组可以共享底层内存，必要条件是张量必须在 CPU 上，且数据类型兼容，能够避免数据拷贝，提高效率

```
import torch
import numpy as np

print("=== Tensor转Numpy ===")
torch_tensor = torch.tensor([[1, 2], [3, 4]])
print(f"原始Tensor:\n{torch_tensor}")

numpy_array = torch_tensor.numpy()
print(f"转换后的Numpy数组:\n{numpy_array}")

print("\n=== 内存共享测试 ===")
print("修改Tensor前Numpy数组的值:", numpy_array[0, 0])
torch_tensor[0, 0] = 100
print("修改Tensor后Numpy数组的值:", numpy_array[0, 0])

print("\n=== Numpy转Tensor ===")
np_array = np.array([[5, 6], [7, 8]])
print(f"原始Numpy数组:\n{np_array}")

converted_tensor = torch.from_numpy(np_array)
print(f"转换后的Tensor:\n{converted_tensor}")

print("修改Numpy数组前Tensor的值:", converted_tensor[0, 0])
np_array[0, 0] = 500
print("修改Numpy数组后Tensor的值:", converted_tensor[0, 0])
```

图 20: 转换代码示例

```
(pytorch_env) PS C:\Users\71883> python test.py
=== Tensor转Numpy ===
原始Tensor:
tensor([[1, 2],
        [3, 4]])
转换后的Numpy数组:
[[1 2]
 [3 4]]

=== 内存共享测试 ===
修改Tensor前Numpy数组的值: 1
修改Tensor后Numpy数组的值: 100

=== Numpy转Tensor ===
原始Numpy数组:
[[5 6]
 [7 8]]
转换后的Tensor:
tensor([[5, 6],
        [7, 8]])
修改Numpy数组前Tensor的值: tensor(5)
修改Numpy数组后Tensor的值: tensor(500)
```

图 21: 转换前后结果对比

### 2.3.6 自动梯度计算

使用直接调用 `backward()`，需要传入梯度参数后计算

```

import torch

# 创建需要梯度的张量
x = torch.tensor(2.0, requires_grad=True)
y = torch.tensor(3.0, requires_grad=True)

# 定义计算图
z = x**2 + y**3 + x*y
print(f"x = {x}, y = {y}")
print(f"z = x2 + y3 + x*y = {z}")

# 计算梯度
z.backward()

print(f"∂z/∂x = {x.grad}") # 2x + y = 2*2 + 3 = 7
print(f"∂z/∂y = {y.grad}") # 3y2 + x = 3*9 + 2 = 29

```

图 22: 梯度计算方法

```

(pytorch_env) PS C:\Users\71883> python test.py
x = 2.0, y = 3.0
z = x2 + y3 + x*y = 37.0
∂z/∂x = 7.0
∂z/∂y = 29.0

```

图 23: 计算梯度结果

### 2.3.7 自定义简单神经网络

`nn.Module` 是所有神经网络模块的基类，封装了参数管理、设备转移等功能。前向传播定义数据流动路径，支持复杂逻辑 (1) `nn.Module` 基类，采用模块化设计，是所有神经网络模块的基类，自动跟踪所有可训练参数并处理 CPU/GPU 设备转移，支持模型保存和加载。

(2) 采用前向传播设计，`forward` 方法定义数据流动路径。支持条件判断、循环等复杂逻辑

```

print("=== 自定义神经网络类 ===")

class SimpleNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, dropout_rate=0.2):
        """
        初始化网络层
        """
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.bn1 = nn.BatchNorm1d(hidden_size) # 批归一化
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.bn2 = nn.BatchNorm1d(hidden_size)
        self.fc3 = nn.Linear(hidden_size, output_size)
        self.dropout = nn.Dropout(dropout_rate)
        self.activation = nn.ReLU()

    def forward(self, x):
        """
        定义前向传播
        """
        x = self.activation(self.bn1(self.fc1(x)))
        x = self.dropout(x)
        x = self.activation(self.bn2(self.fc2(x)))
        x = self.dropout(x)
        x = self.fc3(x) # 输出层不使用激活函数
        return x

    def initialize_weights(self):
        """
        自定义权重初始化
        """
        for m in self.modules():
            if isinstance(m, nn.Linear):
                nn.init.xavier_uniform_(m.weight)
                nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.BatchNorm1d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)

# 创建网络实例
input_size = 10
hidden_size = 50

```

图 24: 自定义神经网络的主体代码



```

(pytorch_env) PS C:\Users\71883> python test.py
=== 自定义神经网络类 ===
网络结构:
SimpleNN(
  (fc1): Linear(in_features=10, out_features=50, bias=True)
  (bn1): BatchNorm1d(50, eps=1e-05, momentum=0.1, affine=True, track_run
ning_stats=True)
  (fc2): Linear(in_features=50, out_features=50, bias=True)
  (bn2): BatchNorm1d(50, eps=1e-05, momentum=0.1, affine=True, track_run
ning_stats=True)
  (fc3): Linear(in_features=50, out_features=3, bias=True)
  (dropout): Dropout(p=0.2, inplace=False)
  (activation): ReLU()
)

=== 网络参数分析 ===
fc1.weight: torch.Size([50, 10]) - 可训练: True
fc1.bias: torch.Size([50]) - 可训练: True
bn1.weight: torch.Size([50]) - 可训练: True
bn1.bias: torch.Size([50]) - 可训练: True
fc2.weight: torch.Size([50, 50]) - 可训练: True
fc2.bias: torch.Size([50]) - 可训练: True
bn2.weight: torch.Size([50]) - 可训练: True
bn2.bias: torch.Size([50]) - 可训练: True
fc3.weight: torch.Size([3, 50]) - 可训练: True
fc3.bias: torch.Size([3]) - 可训练: True
总参数量: 3453
可训练参数量: 3453

=== 前向传播测试 ===
输入形状: torch.Size([4, 10])
训练模式输出形状: torch.Size([4, 3])
评估模式输出形状: torch.Size([4, 3])

=== 模型保存和加载 ===
模型已保存
C:\Users\71883\test.py:90: FutureWarning: You are using `torch.load` wit
h `weights_only=False` (the current default value), which uses the defau
lt pickle module implicitly. It is possible to construct malicious pickl
e data which will execute arbitrary code during unpickling (See https://
github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for mo
re details). In a future release, the default value for `weights_only` w
ill be flipped to `True`. This limits the functions that could be execut
ed during unpickling. Arbitrary objects will no longer be allowed to be
loaded via this mode unless they are explicitly allowlisted by the user
via `torch.serialization.add_safe_globals`. We recommend you start setti
ng `weights_only=True` for any use case where you don't have full contro
l of the loaded file. Please open an issue on GitHub for any issues rela
ted to this experimental feature.
  new_model.load_state_dict(torch.load('simple_model.pth'))

```

图 25: 自定义神经网络

### 2.3.8 常用损失函数的使用

损失函数衡量模型预测与真实值的差异, MSE 适用于回归任务, 交叉熵适用于分类任务。理解不同损失函数的数学特性和适用场景对模型训练至关重要, 合适的损失函数能有效引导模型学习方向

```

import torch
import torch.nn as nn
import torch.nn.functional as F

print("=== 回归问题损失函数 ===")
# 均方误差损失
mse_loss = nn.MSELoss()
predictions_reg = torch.tensor([2.5, 0.5, 3.0, 1.2], dtype=torch.float32)
targets_reg = torch.tensor([3.0, 0.8, 2.8, 1.0], dtype=torch.float32)
mse = mse_loss(predictions_reg, targets_reg)
print(f"MSE Loss: {mse.item():.4f}")

# L1损失
l1_loss = nn.L1Loss()
l1 = l1_loss(predictions_reg, targets_reg)
print(f"L1 Loss: {l1.item():.4f}")

print("\n=== 分类问题损失函数 ===")
# 二分类交叉熵
bce_loss = nn.BCELoss()
predictions_binary = torch.tensor([0.8, 0.2, 0.6, 0.1], dtype=torch.float32)
targets_binary = torch.tensor([1.0, 0.0, 1.0, 0.0], dtype=torch.float32)
bce = bce_loss(predictions_binary, targets_binary)
print(f"BCE Loss: {bce.item():.4f}")

# 多分类交叉熵
ce_loss = nn.CrossEntropyLoss()
predictions_multi = torch.tensor([[2.0, 1.0, 0.1], [0.5, 2.0, 0.3], [0.2, 0.1, 3.0]])
dtype=torch.float32)
targets_multi = torch.tensor([0, 1, 2])
ce = ce_loss(predictions_multi, targets_multi)
print(f"CrossEntropy Loss: {ce.item():.4f}")

print("\n=== 损失函数特性分析 ===")
# 测试不同预测情况的损失
perfect_pred = torch.tensor([1.0, 0.0, 0.0]).unsqueeze(0)
bad_pred = torch.tensor([0.0, 1.0, 0.0]).unsqueeze(0)
target = torch.tensor([0])

perfect_loss = ce_loss(perfect_pred, target)
bad_loss = ce_loss(bad_pred, target)
print(f"完美预测损失: {perfect_loss.item():.4f}")
print(f"错误预测损失: {bad_loss.item():.4f}")

```

图 26: 损失函数使用方法

```
(pytorch_env) PS C:\Users\71883> python test.py
=== 回归问题损失函数 ===
MSE Loss: 0.1050
L1 Loss: 0.3000

=== 分类问题损失函数 ===
BCE Loss: 0.2656
CrossEntropy Loss: 0.2891

=== 损失函数特性分析 ===
完美预测损失: 0.5514
错误预测损失: 1.5514
```

图 27: 损失函数使用结果

### 2.3.9 反向传播流程实践

反向传播是基于链式法则的梯度计算方法，通过计算图从输出向输入逐层传播误差。梯度清零防止累加，数值梯度验证确保实现正确性。

```

# 创建简单模型
class SimpleModel(nn.Module):
    def __init__(self):
        super(SimpleModel, self).__init__()
        self.linear1 = nn.Linear(2, 3)
        self.linear2 = nn.Linear(3, 1)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.linear1(x))
        x = self.linear2(x)
        return x

model = SimpleModel()
criterion = nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

print("模型参数初始值:")
for name, param in model.named_parameters():
    print(f"{name}: {param.data}")

# 训练数据
inputs = torch.tensor([[1.0, 2.0], [2.0, 1.0]], dtype=torch.float32)
targets = torch.tensor([[3.0], [2.5]], dtype=torch.float32)

print("\n=== 单次训练迭代 ===")
# 训练模式
model.train()

# 前向传播
outputs = model(inputs)
print(f"模型输出: {outputs.detach()}")

loss = criterion(outputs, targets)
print(f"损失值: {loss.item():.4f}")

# 梯度清零
optimizer.zero_grad()
print("梯度已清零")

# 反向传播
loss.backward()
print("反向传播完成")

```

图 28: 反向传播简单实现

```
(pytorch_env) PS C:\Users\71883> python test.py
模型参数初始值:
linear1.weight: tensor([[ 0.5559, -0.6501],
                        [ 0.5543, -0.6883],
                        [-0.1213, -0.4465]])
linear1.bias: tensor([-0.0196, -0.3360,  0.3927])
linear2.weight: tensor([[ 0.4313, -0.2538, -0.5083]])
linear2.bias: tensor([0.5012])

=== 单次训练迭代 ===
模型输出: tensor([[0.5012],
                  [0.6704]])
损失值: 4.7958
梯度已清零
反向传播完成
```

图 29: 反向传播结果

### 2.3.10 优化器与权重更新

优化器根据梯度更新模型参数，SGD 简单但可能收敛慢，带动量的 SGD 加速收敛，Adam 自适应调整学习率。学习率调度动态调整学习率，权重衰减防止过拟合

```
def train_step(model, inputs, targets):
    # 前向传播
    outputs = model(inputs)
    loss = criterion(outputs, targets)

    # 反向传播
    optimizer.zero_grad() # 清空过往梯度
    loss.backward()        # 反向传播计算梯度

    # 查看梯度
    print("梯度示例:", model.fc.weight.grad[0][:3])

    # 权重更新
    optimizer.step()       # 执行优化步骤（更新权重）

    return loss.item()

# 5. 执行训练步骤
loss_value = train_step(model, inputs, targets)
print(f"损失值: {loss_value:.4f}")

# 6. 查看更新后的权重
print("更新后的权重示例:", model.fc.weight.data[0][:3])
```

图 30: 优化器简单使用

```
(pytorch_env) PS C:\Users\71883> python test.py
梯度示例: tensor([ 1.8105,  1.0852, -1.0333])
损失值: 1.1289
更新后的权重示例: tensor([ 0.0453, -0.3100, -0.1840])
```

图 31: 权重更新结果

### 2.3.11 CIFAR10 数据加载和预处理

下载 CIFAR10 数据集，定义数据变换，创建 DataLoader，实现批量数据加载。数据加载是模型训练的前提，DataLoader 提供批量加载、打乱、并行加载等功能。数据预处理包括归一化、数据增强等，对模型性能有重要影响

```
def main():
    # 数据预处理
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])

    # 加载数据集
    trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                            download=True, transform=transform)
    trainloader = torch.utils.data.DataLoader(trainset, batch_size=16,
                                              shuffle=True, num_workers=2)

    testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                            download=True, transform=transform)
    testloader = torch.utils.data.DataLoader(testset, batch_size=16,
                                             shuffle=False, num_workers=2)

    classes = ('plane', 'car', 'bird', 'cat', 'deer',
              'dog', 'frog', 'horse', 'ship', 'truck')

    def imshow(img):
        img = img / 2 + 0.5 # 反归一化
        npimg = img.numpy()
        plt.imshow(np.transpose(npimg, (1, 2, 0)))
        plt.axis('off') # 不显示坐标轴

    # 获取一个批量的数据
    dataiter = iter(trainloader)
    images, labels = next(dataiter)

    # 创建子图显示多张图片
    fig, axes = plt.subplots(2, 2, figsize=(8, 8))
    axes = axes.ravel()

    for i in range(4):
        img = images[i]
        img = img / 2 + 0.5 # 反归一化
        npimg = img.numpy()
        axes[i].imshow(np.transpose(npimg, (1, 2, 0)))
        axes[i].set_title(classes[labels[i]])
        axes[i].axis('off')

    plt.tight_layout()
    plt.show()

    # 打印信息
    print('显示图片的标签:', ' '.join('%5s' % classes[labels[j]] for j in range(4)))
    print(f'训练集样本数: {len(trainset)}')
    print(f'测试集样本数: {len(testset)}')
    print(f'图片尺寸: {images.shape[2]} x {images.shape[3]}')

    # 显示每个类别的样本数量
    train_targets = [target for _, target in trainset]
    class_counts = {classes[i]: train_targets.count(i) for i in range(10)}
    print("\n训练集各分类样本数量:")
    for cls, count in class_counts.items():
        print(f"{cls}: {count}")

if __name__ == '__main__':
    main()
```

图 32: 处理代码



(2) 删除和修改元组



图 33: 处理结果

2.3.12 卷积操作

```
输入图像形状: torch.Size([1, 1, 5, 5])  
卷积输出形状: torch.Size([1, 1, 5, 5])
```

图 34: 卷积操作

```
conv = nn.Conv2d(1, 1, kernel_size=3, padding=1)
image = torch.randn(1, 1, 5, 5) # (batch, channels, height, width)
conv_output = conv(image)
print(f"输入图像形状: {image.shape}")
print(f"卷积输出形状: {conv_output.shape}")
```

图 35: 卷积操作效果图

### 2.3.13 卷积神经网络简单实现

```
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        # 定义卷积层
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1) # 输入1通道, 输出32通道
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1) # 输入32通道, 输出64通道
        # 定义全连接层
        self.fc1 = nn.Linear(64 * 7 * 7, 128) # 展平后输入到全连接层
        self.fc2 = nn.Linear(128, 10) # 10 个类别

    def forward(self, x):
        x = F.relu(self.conv1(x)) # 第一层卷积 + ReLU
        x = F.max_pool2d(x, 2) # 最大池化
        x = F.relu(self.conv2(x)) # 第二层卷积 + ReLU
        x = F.max_pool2d(x, 2) # 最大池化
        x = x.view(-1, 64 * 7 * 7) # 展平
        x = F.relu(self.fc1(x)) # 全连接层 + ReLU
        x = self.fc2(x) # 最后一层输出
        return x

# 创建模型实例
model = SimpleCNN()
```

图 36: 简单实现

### 2.3.14 测试代码以及可视化结果

```
(pytorch_env) PS C:\Users\71883> python test.py
Epoch [1/5], Loss: 0.2589
Epoch [2/5], Loss: 0.0545
Epoch [3/5], Loss: 0.0390
Epoch [4/5], Loss: 0.0304
Epoch [5/5], Loss: 0.0229
Test Accuracy: 99.04%
```

图 37: 训练数据

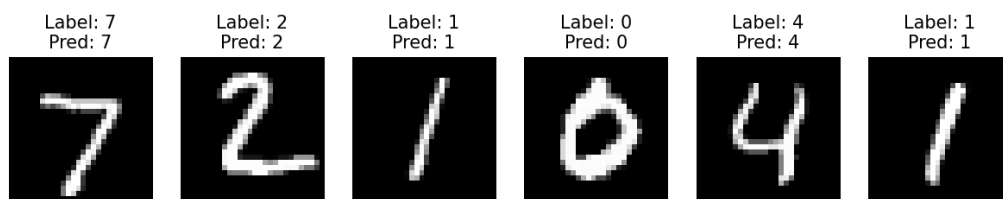


图 38: 可视化结果

### 3 心得体会

在本次的实验当中，从调试与性能分析当中学习基础调试工具，动手实践学习，感受如何系统化地定位和解决问题；在元编程的实践中，知道什么是元编程以及代码抽象；在 Pytorch 的部分，学习相关的基础语法知识，拓宽了视野，有助于以后的学习。

### 4 实验代码查看链接

本次报告相关练习、报告和代码均可以在 <https://github.com/chen2-spec/my-latex-report> 查看