中国海洋大学信息科学与工程学部
系统开发工具基础
2025 年夏季学期

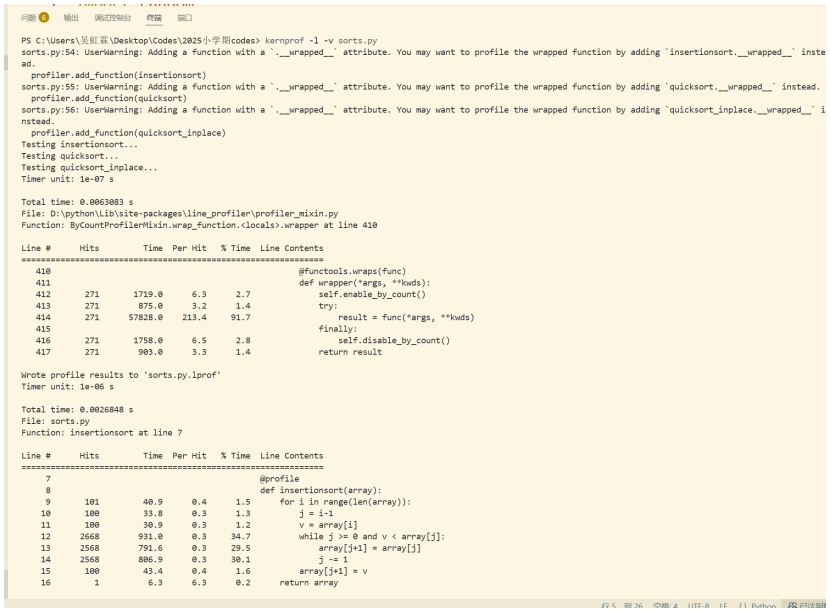
实验报告四 4

24 计算机科学与技术吴虹霖　学号: 24020007135　2025 年 9 月 19 日

1　一、程序性能与调试

实例 1.1 **line profiler 比较插入快速排序性能**

操作: 使用 kernprof 命令运行 line profiler，分析插入排序和快速排序的性能差异。



```
问题 6  输出  调试控制台  终端  端口

PS C:\Users\吴虹霖\Desktop\Codes\2025小学期codes> kernprof -l -v sorts.py
sorts.py:54: UserWarning: Adding a function with a `__wrapped__` attribute. You may want to profile the wrapped function by adding `insertionsort.__wrapped__` inste
ad.
  profiler.add_function(insertionsort)
sorts.py:55: UserWarning: Adding a function with a `__wrapped__` attribute. You may want to profile the wrapped function by adding `quicksort.__wrapped__` instead.
  profiler.add_function(quicksort)
sorts.py:56: UserWarning: Adding a function with a `__wrapped__` attribute. You may want to profile the wrapped function by adding `quicksort_inplace.__wrapped__` i
nstead.
  profiler.add_function(quicksort_inplace)
Testing insertionsort...
Testing quicksort...
Testing quicksort_inplace...
Timer unit: 1e-07 s

Total time: 0.0063083 s
File: D:\python\Lib\site-packages\line_profiler\profiler_mixin.py
Function: ByCountProfilerMixin.wrap_function.<locals>.wrapper at line 410

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   410                                           @functools.wraps(func)
   411                                           def wrapper(*args, **kwds):
   412       271      1719.0      6.3      2.7              self.enable_by_count()
   413       271       875.0      3.2      1.4              try:
   414       271     57828.0    213.4     91.7                  result = func(*args, **kwds)
   415                                               finally:
   416       271      1758.0      6.5      2.8                  self.disable_by_count()
   417       271       903.0      3.3      1.4              return result

Wrote profile results to 'sorts.py.lprof'
Timer unit: 1e-06 s

Total time: 0.0026848 s
File: sorts.py
Function: insertionsort at line 7

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     7                                           @profile
     8                                           def insertionsort(array):
     9       101         40.9      0.4      1.5      for i in range(len(array)):
    10       100         33.8      0.3      1.3          j = i-1
    11       100         30.9      0.3      1.2          v = array[i]
    12      2668        931.0      0.3     34.7          while j >= 0 and v < array[j]:
    13      2568        791.6      0.3     29.5              array[j+1] = array[j]
    14      2568        806.9      0.3     30.1              j -= 1
    15       100         43.4      0.4      1.6          array[j+1] = v
    16         1          6.3      6.3      0.2      return array
```

图 1: line profiler 比较插入快速排序性能

实例 1.2 **cprofile 比较插入快速排序性能**

操作: 使用 cProfile 运行性能分析，获取函数调用统计信息。

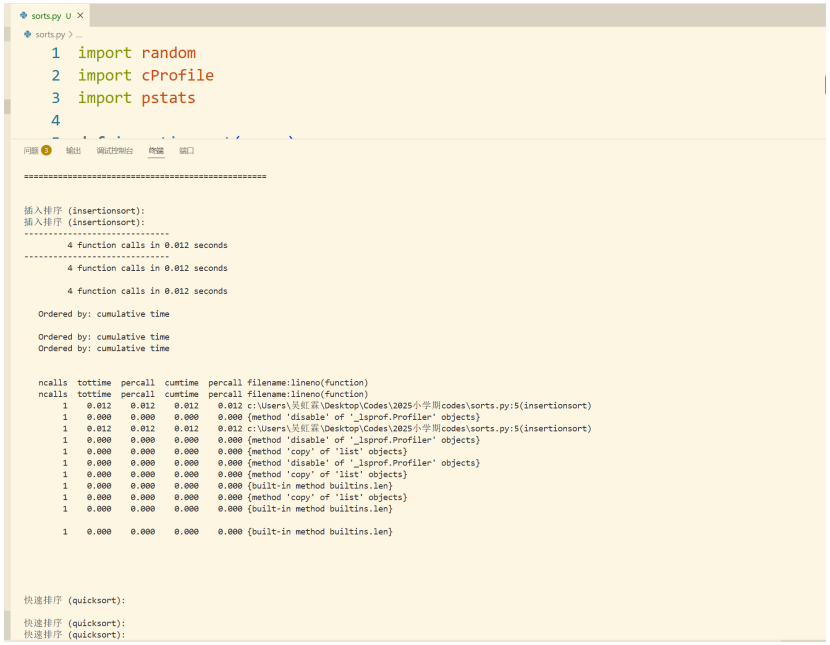


图 2: shellcheck 检查脚本

实例 1.3 **shellcheck 检查脚本**

操作: 运行 shellcheck 命令检查脚本，获取改进建议。

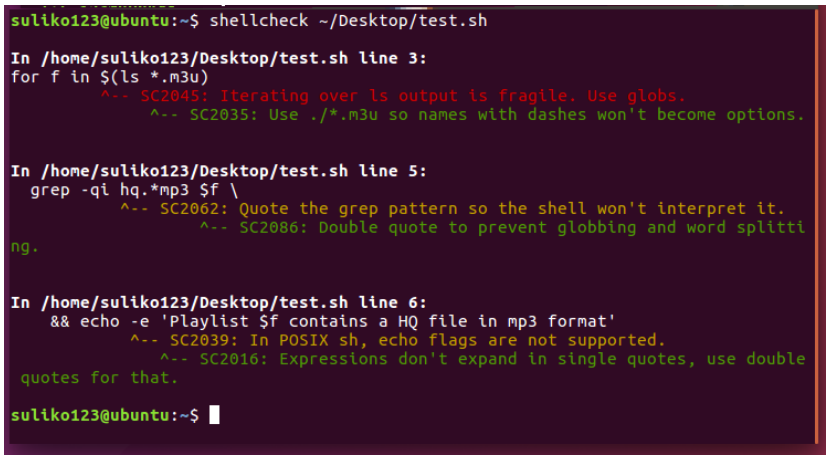


图 3: shellcheck 检查脚本

实例 1.4 **snakeviz 可视化程序性能**

操作: 运行 cProfile 生成性能数据，使用 snakeviz 进行可视化分析。

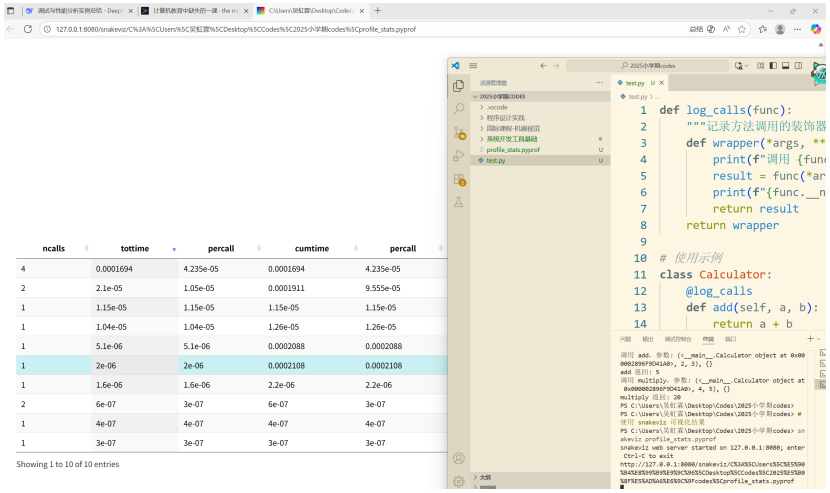


图 4: snakeviz 可视化程序性能

实例 1.5 **调试查看日志**

操作: 使用 journalctl 命令查看特定时间范围或特定用户的系统日志。

```
suliko123@ubuntu:~$ journalctl --since "1 day ago" _UID=0
-- Logs begin at Thu 2025-09-18 05:23:55 PDT, end at Thu 2025-09-18 05:24:32 PDT
Sep 18 05:23:55 ubuntu systemd-journald[342]: Runtime journal (/run/log/journal/
Sep 18 05:23:55 ubuntu systemd-journald[342]: Journal started
Sep 18 05:23:55 ubuntu systemd-modules-load[339]: Inserted module 'lp'
Sep 18 05:23:55 ubuntu systemd-modules-load[339]: Inserted module 'ppdev'
Sep 18 05:23:55 ubuntu systemd-modules-load[339]: Inserted module 'parport_pc'
Sep 18 05:23:55 ubuntu systemd-modules-load[339]: Module 'fuse' is builtin
Sep 18 05:23:55 ubuntu systemd[1]: Starting Flush Journal to Persistent Storage.
Sep 18 05:23:55 ubuntu systemd[1]: Started udev Kernel Device Manager.
Sep 18 05:23:55 ubuntu systemd[1]: Started Load/Save Random Seed.
Sep 18 05:23:55 ubuntu systemd[1]: Mounted VMware vmblock fuse mount.
Sep 18 05:23:55 ubuntu systemd[1]: Started Flush Journal to Persistent Storage.
Sep 18 05:23:55 ubuntu loadkeys[336]: Loading /etc/console-setup/cached.kmap.gz
Sep 18 05:23:55 ubuntu systemd[1]: Started Set console keymap.
Sep 18 05:23:55 ubuntu systemd[1]: Reached target Local File Systems (Pre).
Sep 18 05:23:55 ubuntu systemd[1]: Reached target Local File Systems.
Sep 18 05:23:55 ubuntu systemd[1]: Started Service for virtual machines hosted o
Sep 18 05:23:55 ubuntu systemd[1]: Starting LSB: AppArmor initialization...
Sep 18 05:23:55 ubuntu systemd[1]: Starting Create Volatile Files and Directorie
Sep 18 05:23:55 ubuntu systemd[1]: Starting Set console font and keymap...
Sep 18 05:23:55 ubuntu systemd[1]: Starting Clean up any mess left by 0dns-up...
Sep 18 05:23:55 ubuntu systemd-tmpfiles[399]: [/usr/lib/tmpfiles.d/var.conf:14]
Sep 18 05:23:55 ubuntu systemd[1]: Starting Tell Plymouth To Write Out Runtime D
```

图 5: 调试查看日志

2 二、元编程

实例 2.1 **type() 动态创建类**

操作: 使用 type() 函数传入类名、基类和属性字典来动态创建类。

图 6: type() 动态创建类

实例 2.2 **注册元类**

操作: 定义元类并在创建类时自动注册到全局注册表中。

图 7: type() 动态创建类

实例 2.3 **动态函数创建**

操作: 使用字符串拼接函数代码，通过 exec() 函数执行并创建函数对象。

图 8: 动态函数创建

实例 2.4 **动态属性访问**

操作: 重写特殊方法实现属性的动态获取和设置。

图 9: 动态属性访问

实例 2.5 **方法调用日志装饰器**

操作: 定义装饰器函数，包装目标方法并添加日志记录功能。

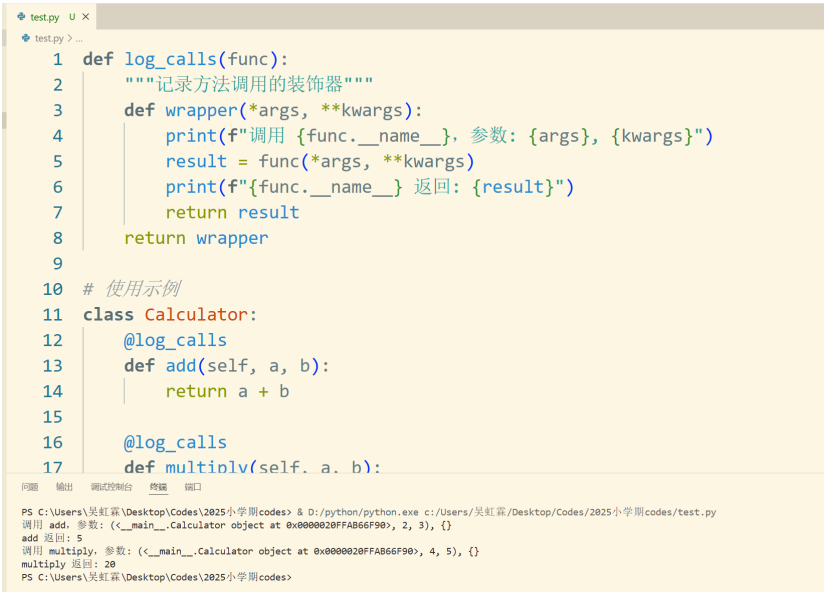


图 10: 方法调用日志装饰器

实例 2.6 **属性拦截**

操作： 重写 getattr 方法，对特定模式的属性访问进行特殊处理。

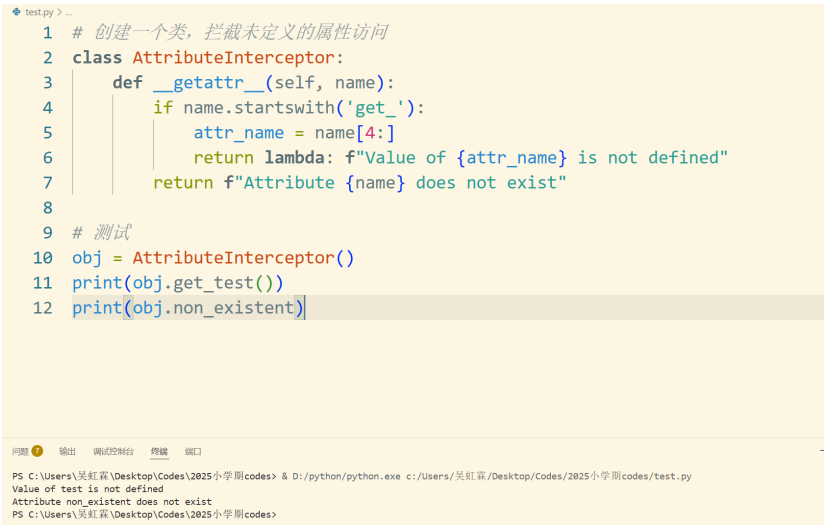


图 11: 属性拦截

实例 2.7 **原编程装饰器**

操作： 定义装饰器函数，在函数执行前后记录时间并计算耗时。

图 12: 原编程装饰器

3 三、PyTorch

实例 3.1 **PyTorch 张量操作**

操作: 使用 torch 模块创建张量，进行逐元素运算和属性访问。

图 13: PyTorch 张量操作

实例 3.2 **PyTorch 激活函数**

操作: 使用 torch.nn.functional 模块调用不同的激活函数处理张量。

```python
import torch
import torch.nn.functional as F

# 首先创建一个示例输入张量
input_tensor = torch.tensor([-1.0, 0.0, 1.0, 2.0])

# ReLU 激活
output_relu = F.relu(input_tensor)
print("ReLU output:", output_relu)

# Sigmoid 激活
output_sigmoid = torch.sigmoid(input_tensor)
print("Sigmoid output:", output_sigmoid)

# Tanh 激活
output_tanh = torch.tanh(input_tensor)
print("Tanh output:", output_tanh)
```

```
PS C:\Users\吴虹霖\Desktop\Codes\2025小学期codes> & D:/python/python.exe c:/Users/吴虹霖/Desktop/Codes/2025小学期codes/test.py
ReLU output: tensor([0., 0., 1., 2.])
Sigmoid output: tensor([0.2689, 0.5000, 0.7311, 0.8808])
Tanh output: tensor([-0.7616,  0.0000,  0.7616,  0.9640])
PS C:\Users\吴虹霖\Desktop\Codes\2025小学期codes>
```

图 14: PyTorch 激活函数

实例 3.3 PyTorch 简单的神经网络

操作： 继承 nn.Module 类，定义网络结构和前向传播过程。

```python
import torch
import torch.nn as nn

# 定义一个简单的神经网络模型
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        # 定义一个输入层到隐藏层的全连接层
        self.fc1 = nn.Linear(2, 2)  # 输入 2 个特征，输出 2 个特征
        # 定义一个隐藏层到输出层的全连接层
        self.fc2 = nn.Linear(2, 1)  # 输入 2 个特征，输出 1 个预测值

    def forward(self, x):
        # 前向传播过程
        x = torch.relu(self.fc1(x))  # 使用 ReLU 激活函数
        x = self.fc2(x)  # 输出层
        return x
```

```
PS C:\Users\吴虹霖\Desktop\Codes\2025小学期codes> & D:/python/python.exe c:/Users/吴虹霖/Desktop/Codes/2025小学期codes/test.py
SimpleNN(
  (fc1): Linear(in_features=2, out_features=2, bias=True)
  (fc2): Linear(in_features=2, out_features=1, bias=True)
)
PS C:\Users\吴虹霖\Desktop\Codes\2025小学期codes>
```

图 15: PyTorch 简单的神经网络

实例 3.4 PyTorch 损失函数

操作： 使用 nn 模块中的损失函数类，计算预测值与真实值之间的差异。

图 16: PyTorch 损失函数

实例 3.5 **PyTorch 优化器**

操作： 使用 optim 模块创建优化器，设置学习率和模型参数。

图 17: PyTorch 优化器

实例 3.6 **PyTorch 训练循环的基本结构**

操作： 设置训练循环，在每个 epoch 中执行完整的前向和反向传播过程。

```
test.py U ×
test.py > ...
optimizer = optim.SGD(model.parameters(), lr=learning_rate)

# 生成训练数据
X = torch.randn(10, 2)  # 10 个样本，每个样本有 2 个特征
Y = torch.randn(10, 1)  # 10 个目标标签

# 训练过程
for epoch in range(num_epochs):
    model.train()  # 设置模型为训练模式
    optimizer.zero_grad()  # 清除梯度
    output = model(X)  # 前向传播
    loss = criterion(output, Y)  # 计算损失
    loss.backward()  # 反向传播
    optimizer.step()  # 更新权重

    if (epoch + 1) % 10 == 0:  # 每 10 轮输出一次损失
        print(f'Epoch [{epoch + 1}/{num_epochs}], Loss: {loss.item():.4f}')
```

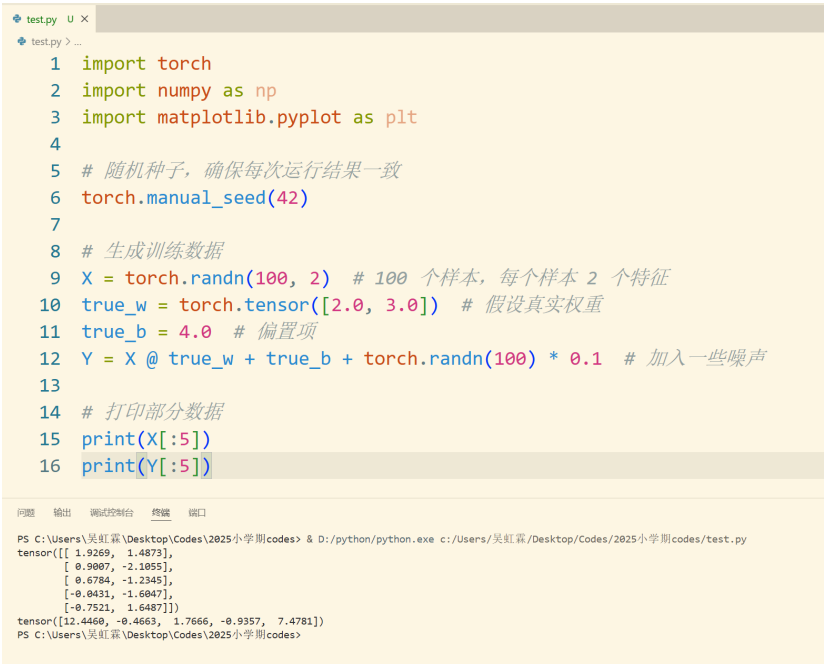
```
问题 输出 调试控制台 终端 端口 Python

PS C:\Users\吴虹霖\Desktop\Codes\2025小学期codes> & D:/python/python.exe c:/Users/吴虹霖/Desktop/Codes/2025小学期codes/test.py
Epoch [10/100], Loss: 0.7467
Epoch [20/100], Loss: 0.6489
Epoch [30/100], Loss: 0.5820
Epoch [40/100], Loss: 0.5358
Epoch [50/100], Loss: 0.5034
Epoch [60/100], Loss: 0.4803
Epoch [70/100], Loss: 0.4636
Epoch [80/100], Loss: 0.4512
Epoch [90/100], Loss: 0.4418
Epoch [100/100], Loss: 0.4346
Final Test Loss: 0.4340
PS C:\Users\吴虹霖\Desktop\Codes\2025小学期codes>
```

图 18: PyTorch 训练循环的基本结构

实例 3.7 **简单的线性关系的数据集**

操作： 使用 torch 生成随机数据，并添加线性关系和噪声。



```
test.py U ×
test.py > ...
1  import torch
2  import numpy as np
3  import matplotlib.pyplot as plt
4
5  # 随机种子，确保每次运行结果一致
6  torch.manual_seed(42)
7
8  # 生成训练数据
9  X = torch.randn(100, 2)  # 100 个样本，每个样本 2 个特征
10 true_w = torch.tensor([2.0, 3.0])  # 假设真实权重
11 true_b = 4.0  # 偏置项
12 Y = X @ true_w + true_b + torch.randn(100) * 0.1  # 加入一些噪声
13
14 # 打印部分数据
15 print(X[:5])
16 print(Y[:5])
```

```
问题 输出 调试控制台 终端 端口

PS C:\Users\吴虹霖\Desktop\Codes\2025小学期codes> & D:/python/python.exe c:/Users/吴虹霖/Desktop/Codes/2025小学期codes/test.py
tensor([[ 1.9269,  1.4873],
        [ 0.9007, -2.1055],
        [ 0.6784, -1.2345],
        [-0.0431, -1.6047],
        [-0.7521,  1.6487]])
tensor([12.4460, -0.4663,  1.7666, -0.9357,  7.4781])
PS C:\Users\吴虹霖\Desktop\Codes\2025小学期codes>
```

图 19: 简单的线性关系的数据集

实例 3.8 **自定义数据集类 MyDataset**

操作： 实现 $_{len}$ $_{getitem}$

图 20: 自定义数据集类 MyDataset

个人心得

通过本次实验，我深入学习了程序性能分析与调试、元编程以及 PyTorch 深度学习框架的使用。在程序性能与调试部分，掌握了使用 line profiler、cProfile 和 snakeviz 等工具分析代码性能的方法，以及使用 shellcheck 和 journalctl 进行脚本检查和系统调试的技巧。这些工具的使用大大提高了代码优化和问题排查的效率。

在元编程部分，我学习了 Python 中强大的元编程能力，包括使用 type() 动态创建类、使用装饰器增强函数功能、以及通过特殊方法实现动态属性访问等。这些技术使得代码更加灵活和可扩展，能够应对更复杂的编程场景。

在 PyTorch 部分，我从张量操作基础开始，逐步学习了神经网络构建、损失函数、优化器以及训练循环的实现。通过创建简单的线性数据集和自定义数据集类，我加深了对深度学习数据流程的理解。这些知识为后续进行更复杂的深度学习项目打下了坚实基础。

实验过程中，我遇到了一些挑战，如性能分析工具的输出解读、元编程概念的理解以及 PyTorch 模型训练中的参数调整等。通过查阅文档、调试代码和反复实验，我逐步解决了这些问题，增强了解决问题的能力。

本次实验不仅巩固了 Python 高级编程技巧，也为我后续从事系统开发和机器学习项目提供了重要支持。未来我将进一步探索性能优化技术、元编程高级应用以及 PyTorch 高级特性，提升开发效率和项目质量。