2024夏季学期《计算机系统综合实践Ⅰ》
实验报告

姓名	苏梓凯
学号	22090021016

2024 年 9 月 4 日

1 实验练习内容

1.1 调试及性能分析

学习如何使用 Python 的调试工具和性能分析工具来优化代码。

学习使用打印语句和日志来调试代码，了解调试器的使用方法，如Python中的pdb

1.2 元编程演示实验

理解和应用 Python 的元编程技术，包括装饰器、元类和动态代码生成。

探讨Python元编程的基本概念，包括元类（Metaclasses）、装饰器工厂、动态属性和方法、代码生成和协程的元编程应用。

1.3 PyTorch 编程

学习使用 PyTorch 进行深度学习模型的构建、训练和评估。

学习PyTorch的基础知识，包括张量、自动求导、并行计算、数据读入、模型构建、损失函数、训练和评估、可视化和优化器

2　实验结果展示

2.1 调试及性能分析

1. 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

使用journalctl | grep ls

```
ouc@islouc-vm:~/Desktop$ journalctl | grep sudo
9月 25 14:41:06 islouc-vm sudo[2085]: pam_unix(sudo:auth): authentication failur
e; logname= uid=1000 euid=0 tty=/dev/pts/0 ruser=ouc rhost=  user=ouc
9月 25 14:41:10 islouc-vm sudo[2085]:      ouc : TTY=pts/0 ; PWD=/home/ouc/Deskt
op ; USER=root ; COMMAND=/usr/bin/apt install open-vm-tools
9月 25 14:41:10 islouc-vm sudo[2085]: pam_unix(sudo:session): session opened for
 user root by (uid=0)
9月 25 14:41:10 islouc-vm sudo[2085]: pam_unix(sudo:session): session closed for
 user root
9月 25 14:41:18 islouc-vm sudo[2136]:      ouc : TTY=pts/0 ; PWD=/home/ouc/Deskt
op ; USER=root ; COMMAND=/usr/bin/apt install open-vm-tools open-vm-tools-deskto
p
9月 25 14:41:18 islouc-vm sudo[2136]: pam_unix(sudo:session): session opened for
 user root by (uid=0)
9月 25 14:41:19 islouc-vm sudo[2136]: pam_unix(sudo:session): session closed for
 user root
```

2. 使用gdb调试器进行调试分析

```python
# 示例代码：简单的数学计算程序
import pdb
def calculate_discount(price, discount_rate):
    # 计算折扣后的价格
    discounted_price = price * (1 - discount_rate)
    return discounted_price

def main():
    # 假设商品原价和折扣率
    original_price = 100.0
    discount = 0.20  # 20% 的折扣

    # 计算折扣后的价格
    final_price = calculate_discount(original_price, discount)
    pdb.set_trace()
    # 打印结果
```

3. 使用打印调试法调试程序

```python
# 打印调试信息
print("The original price is: ${:.2f}".format(original_price))
print("The discount is: ${:.2f}".format(discount * 100))
print("The final price is: ${:.2f}".format(final_price))
```

4. 第三方日志系统

```
ouc@islouc-vm:~/Desktop/lab$ logger "Hello Logs"
ouc@islouc-vm:~/Desktop/lab$ journalctl --since "1m ago" | grep Hello
9月 14 23:25:46 islouc-vm ouc[4256]: Hello Logs
```

5. 使用python的time模块调试分析

```
py > test.py > ...
1   import time, random
2   n = random.randint(1, 10) * 100
3
4   # 获取当前时间
5   start = time.time()
6
7   # 执行一些操作
8   print("Sleeping for {} ms".format(n))
9   time.sleep(n/1000)
10
11  # 比较当前时间和起始时间
12  print(time.time() - start)

问题  输出  调试控制台  终端  端口

PS E:\py> & C:/Users/LENOVO/AppData/Local/Programs/Python/Python311/python.exe e:/py/test.py
Sleeping for 200 ms
0.2004718780517578
```

6. 根据给出的排序算法的实现。请使用 cProfile 和 line_profiler 来比较插入排序和快速排序的性能。

```
PS C:\Users\LENOVO\Desktop> kernprof -l -v sorts.py
Timer unit: 1e-07 s

Total time: 0.169072 s
File: sorts.py
Function: insertionsort at line 20

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    20                                           def insertionsort(array):
    21
    22     25204      51018.0      2.0      3.0      for i in range(len(array)):
    23     24204      55187.0      2.3      3.3          j = i-1
    24     24204      55491.0      2.3      3.3          v = array[i]
    25    216702     582387.0      2.7     34.4          while j >= 0 and v < array[j]:
    26    192498     481103.0      2.5     28.5              array[j+1] = array[j]
    27    192498     402394.0      2.1     23.8              j -= 1
    28     24204      61128.0      2.5      3.6          array[j+1] = v
    29      1000       2010.0      2.0      0.1      return array

Total time: 0.0940361 s
File: sorts.py
Function: quicksort at line 32

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    32                                           def quicksort(array):
    33     33826      97681.0      2.9     10.4      if len(array) <= 1:
    34     17413      31890.0      1.8      3.4          return array
    35     16413      38381.0      2.3      4.1      pivot = array[0]
    36     16413     355007.0     21.6     37.8      left = [i for i in array[1:] if i < pivot]
    37     16413     354457.0     21.6     37.7      right = [i for i in array[1:] if i >= pivot]
    38     16413      62945.0      3.8      6.7      return quicksort(left) + [pivot] + quicksort(right)
```

```
PS C:\Users\LENOVO\Desktop> python -m memory_profiler sorts.py
Timer unit: 1e-07 s

Total time: 0.186993 s
File: sorts.py
Function: insertionsort at line 20

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    20                                           def insertionsort(array):
    21
    22     26740      54241.0      2.0      2.9      for i in range(len(array)):
    23     25740      62112.0      2.4      3.3          j = i-1
    24     25740      58755.0      2.3      3.1          v = array[i]
    25    235497     649297.0      2.8     34.7          while j >= 0 and v < array[j]:
    26    209757     534829.0      2.5     28.6              array[j+1] = array[j]
    27    209757     442258.0      2.1     23.7              j -= 1
    28     25740      66423.0      2.6      3.6          array[j+1] = v
    29      1000       2015.0      2.0      0.1      return array

Total time: 0.0924416 s
File: sorts.py
Function: quicksort at line 32

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    32                                           def quicksort(array):
    33     33646      96492.0      2.9     10.4      if len(array) <= 1:
    34     17323      31445.0      1.8      3.4          return array
    35     16323      38282.0      2.3      4.1      pivot = array[0]
    36     16323     349550.0     21.4     37.8      left = [i for i in array[1:] if i < pivot]
    37     16323     346793.0     21.2     37.5      right = [i for i in array[1:] if i >= pivot]
    38     16323      61854.0      3.8      6.7      return quicksort(left) + [pivot] + quicksort(right)
```

7.使用Valgrind这样的工具来检查内存泄漏问题

```python
@profile
def my_func():
    a = [1] * (10 ** 6)
    b = [2] * (2 * 10 ** 7)
    del b
    return a

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

```
PS C:\Users\LENOVO\Desktop> python -m memory_profiler exe.py
Filename: exe.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
     1   23.738 MiB   23.738 MiB           1   @profile
     2                                         def my_func():
     3   31.375 MiB    7.637 MiB           1       a = [1] * (10 ** 6)
     4  183.965 MiB  152.590 MiB           1       b = [2] * (2 * 10 ** 7)
     5   31.375 MiB -152.590 MiB           1       del b
     6   31.375 MiB    0.000 MiB           1       return a
```

2.2 元编程演示实验

1. 使用exec动态执行代码操作

exec("print('Hello, World!')")

```
if __name__ == '__main__':
    exec("print('Hello, World!')")
```

```
问题　输出　调试控制台　终端　端口

PS E:\py> & C:/Users/LENOVO/AppData/Local/Programs/Python/Python311/python.exe c:/Users/LENOVO/Desktop/exe.py
Hello, World!
```

2. 使用eval计算字符串表达式操作

result = eval("1 + 2")

```
if __name__ == '__main__':
    result = eval("1 + 2")
    print(result)
```

```
问题　输出　调试控制台　终端　端口

PS E:\py> & C:/Users/LENOVO/AppData/Local/Programs/Python/Python311/python.exe c:/Users/LENOVO/Desktop/exe.py
3
```

2. 动态创建类：使用type动态创建一个类，并为其添加方法。

```
def my_method(self):
    return "Hello from my_method"

MyMeta = type('MyMeta', (object,), {'my_method': my_method})
obj = MyMeta()
print(obj.my_method())
```

```
问题　输出　调试控制台　终端　端口

PS E:\py> & C:/Users/LENOVO/AppData/Local/Programs/Python/Python311/python.exe c:/Users/LENOVO/Desktop/exe.py
Hello from my_method
```

4. 使用装饰器工厂：创建一个装饰器工厂，它可以根据传入的参数改变装饰器的行为。

```
C: > Users > LENOVO > Desktop > exe.py > ...
def my_decorator_factory(param):
    def decorator(func):
        def wrapper(*args, **kwargs):
            print(f"Decorator with param: {param}")
            return func(*args, **kwargs)
        return wrapper
    return decorator

@my_decorator_factory("test")
def my_function():
    print("Function executed")

my_function()
```

```
问题　输出　调试控制台　终端　端口

PS E:\py> & C:/Users/LENOVO/AppData/Local/Programs/Python/Python311/python.exe c:/Users/LENOVO/Desktop/exe.py
Decorator with param: test
Function executed
```

5. 使用描述符：创建一个描述符类，它能够在访问属性时进行额外的操作。

```
class MyDescriptor:
    def __init__(self, initial_value=None):
        self.value = initial_value

    def __get__(self, instance, owner):
        return self.value

    def __set__(self, instance, value):
        self.value = value

class MyClass:
    my_attr = MyDescriptor("Initial value")

instance = MyClass()
print(instance.my_attr)  # 输出: Initial value
instance.my_attr = "New value"
print(instance.my_attr)  # 输出: New value
```

```
问题　输出　调试控制台　终端　端口

PS E:\py> & C:/Users/LENOVO/AppData/Local/Programs/Python/Python311/python.exe c:/Users/LENOVO/Desktop/exe.py
Initial value
New value
PS E:\py>
```

6. 元类的高级用法：使用元类来控制类的创建过程，例如，自动注册类到某个注册表中。

```python
class_registry = {}

class Meta(type):
    def __new__(cls, name, bases, dct):
        cls = super().__new__(cls, name, bases, dct)
        class_registry[name] = cls
        return cls

class MyClass(metaclass=Meta):
    pass

print(class_registry['MyClass'])
```

```
问题　输出　调试控制台　终端　端口

PS E:\py> & C:/Users/LENOVO/AppData/Local/Programs/Python/Python311/python.exe c:/Users/LENOVO/Desktop/exe.py
<class '__main__.MyClass'>
PS E:\py>
```

7.动态属性和方法：使用__getattr__和__setattr__方法动态地处理属性访问。

```python
class DynamicClass:
    def __getattr__(self, item):
        return f"Attribute {item} not found"

    def __setattr__(self, key, value):
        print(f"Setting {key} to {value}")

obj = DynamicClass()
obj.new_attribute = "Hello"
print(obj.new_attribute)
```

```
问题　输出　调试控制台　终端　端口

PS E:\py> & C:/Users/LENOVO/AppData/Local/Programs/Python/Python311/python.exe c:/Users/LENOVO/Desktop/exe.py
Setting new_attribute to Hello
Attribute new_attribute not found
PS E:\py>
```

2.3 pytorch编程

1. 张量的基本操作：创建张量并进行基本的数学运算。

```
C: > Users > LENOVO > Desktop > exe.py > ...
  1  import torch
  2
  3  # 创建一个张量
  4  x = torch.tensor([1, 2, 3])
  5  y = torch.tensor([4, 5, 6])
  6
  7  # 张量加法
  8  z = x + y
  9  print(z)  # 输出: tensor([5, 7, 9])
 10
 11  # 张量点乘
 12  w = torch.tensor([1, 2, 3])
 13  result = torch.dot(x, w)
 14  print(result)

问题  输出  调试控制台  终端  端口

PS E:\py> & C:/Users/LENOVO/AppData/Local/Programs/Python/Python311/python.exe c:/Users/LENOVO/Desktop/exe.py
tensor([5, 7, 9])
tensor(14)
PS E:\py>
```

2. 使用CUDA张量：如果可用，将张量移动到GPU上进行计算。

```
17  if torch.cuda.is_available():
18      x = x.cuda()
19      y = y.cuda()
20      z = x + y
21      print(z)
```

3. 使用 PyTorch 的自动求导机制来计算梯度

```
C: > Users > LENOVO > Desktop > exe.py > ...
  1  import torch
  2
  3  # 创建一个需要梯度的张量
  4  x = torch.tensor([2.0], requires_grad=True)
  5
  6  # 定义一个函数 y = x^2
  7  y = x ** 2
  8
  9  # 计算y关于x的导数
 10  y.backward()
 11  print(x.grad)  # 输出: tensor(4.)

问题  输出  调试控制台  终端  端口

PS E:\py> & C:/Users/LENOVO/AppData/Local/Programs/Python/Python311/python.exe c:/Users/LENOVO/Desktop/exe.py
tensor([4.])
PS E:\py>
```

4. 使用 torch.nn 模块构建一个简单的全连接神经网络构建简单的神经网络

```
C: > Users > LENOVO > Desktop > exe.py > ...
1  import torch.nn as nn
2  import torch
3
4  class SimpleNet(nn.Module):
5      def __init__(self):
6          super(SimpleNet, self).__init__()
7          self.fc = nn.Linear(2, 1)
8
9      def forward(self, x):
10         return self.fc(x)
11
12 model = SimpleNet()
13 print(model)
14
15 # 创建输入数据
16 x = torch.tensor([[1.0, 2.0]], requires_grad=True)
17 y = model(x)
18 print(y)
```

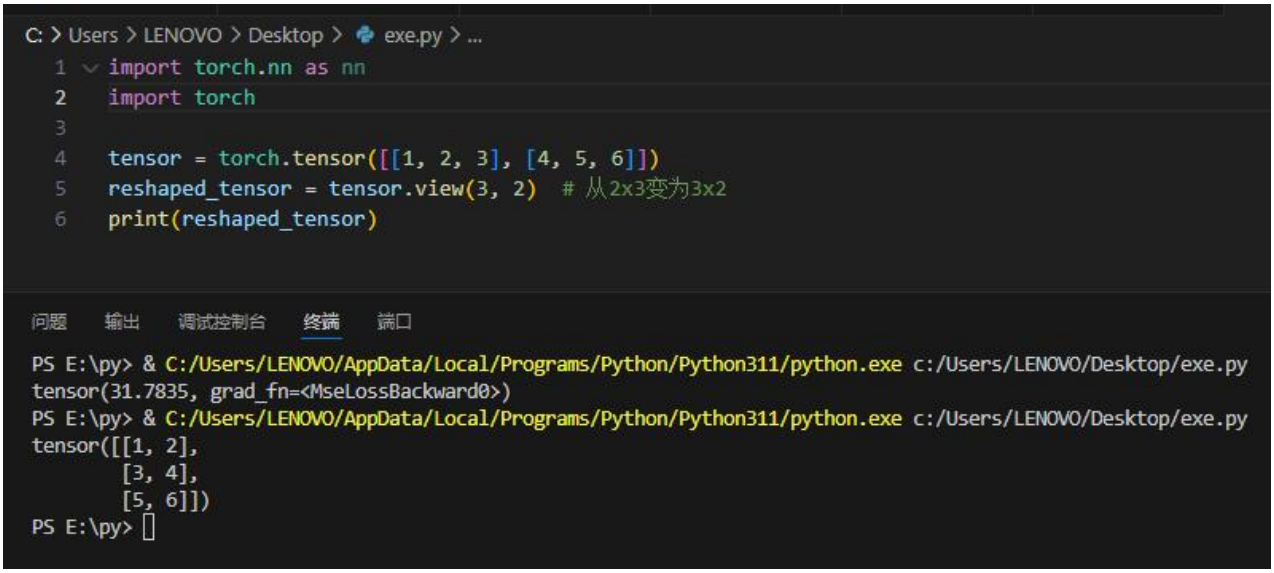
```
问题  输出  调试控制台  终端  端口
PS E:\py> & C:/Users/LENOVO/AppData/Local/Programs/Python/Python311/python.exe c:/Users/LENOVO/Desktop/exe.py
SimpleNet(
  (fc): Linear(in_features=2, out_features=1, bias=True)
)
tensor([[0.1647]], grad_fn=<AddmmBackward0>)
PS E:\py>
```

5. 使用PyTorch构建一个简单的线性回归模型。

```
C: > Users > LENOVO > Desktop > exe.py > ...
1  import torch.nn as nn
2  import torch
3  # 定义线性回归模型
4  model = nn.Linear(in_features=1, out_features=1)
5
6  # 创建数据
7  x = torch.tensor([[1.0], [2.0], [3.0]])
8  y = torch.tensor([[2.0], [4.0], [6.0]])
9
10 # 前向传播
11 prediction = model(x)
12
13 # 计算损失
14 criterion = nn.MSELoss()
15 loss = criterion(prediction, y)
16 print(loss)
```

```
问题  输出  调试控制台  终端  端口
PS E:\py> & C:/Users/LENOVO/AppData/Local/Programs/Python/Python311/python.exe c:/Users/LENOVO/Desktop/exe.py
tensor(31.7835, grad_fn=<MseLossBackward0>)
PS E:\py>
```

6. 改变张量的形状。


C: > Users > LENOVO > Desktop > exe.py > ...
1 import torch.nn as nn
2 import torch
3
4 tensor = torch.tensor([[1, 2, 3], [4, 5, 6]])
5 reshaped_tensor = tensor.view(3, 2) # 从2x3变为3x2
6 print(reshaped_tensor)
问题 输出 调试控制台 终端 端口
PS E:\py> & C:/Users/LENOVO/AppData/Local/Programs/Python/Python311/python.exe c:/Users/LENOVO/Desktop/exe.py
tensor(31.7835, grad_fn=<MseLossBackward0>)
PS E:\py> & C:/Users/LENOVO/AppData/Local/Programs/Python/Python311/python.exe c:/Users/LENOVO/Desktop/exe.py
tensor([[1, 2],
[3, 4],
[5, 6]])
PS E:\py>

3 实验感悟

通过调试分析的实验，我深刻体会到了调试工具在开发过程中的重要性，它们帮助我快速定位和解决问题。通过逐步调试，我能够观察程序的执行流程和变量状态，也学会了如何使用性能分析工具来识别程序的性能瓶颈

通过元编程的实验，我更深入地理解了元编程允许我们在编译时进行代码生成和修改的能力，这是提高代码灵活性和复用性的强大工具。通过编写和使用模板元函数，我学会了如何在编译时处理类型和常量，以及如何通过模板特化来定制行为，这无异进一步提升了我的编程能力

通过pytorch实验，我完成了pytorch的基本入门，了解到了新的数据结构张量，学会了如何进行张量的创建、形状变换和基本运算，也了解到了pytorch的强大的功能,这使我受益匪浅

Github链接:https://github.com/ffffz1/labreport.git