实践报告

高鹏宇
学号：2302007025

一、练习内容与结果

（一）程序调试和性能分析

1. 启动 PDB：

```
import pdb  # 导入 Python 调试器模块
pdb.set_trace()
for i in range(100):
    if i % 2 == 0:
        print(i)
```

结果:你可以在代码中需要调试的地方插入 `import pdb; pdb.set_trace()`，程序运行到这一行时会暂停，进入调试模式。

2. 常用 PDB 命令：

- n (next)：执行下一行代码，不会进入函数内部。
- s (step)：进入当前行调用的函数。
- c (continue)：继续执行程序，直到遇到下一个断点或程序结束。
- l (list)：显示当前代码行以及周围的代码。
- p (print)：打印变量的值。
- h (help)：显示帮助信息。
- b (break)：设置断点，可以在指定行或指定函数处设置断点。
- cl (clear)：清除断点。

- q (quit)：退出调试。

3. cProfile：

```
import cProfile
def my_function():
    for i in range(100000):
        pass
cProfile.run('my_function()')
```

结果：cProfile 是 Python 标准库中的一个性能分析工具，它提供了程序的详细性能分析报告。

4. timeit：

```
import timeit
def my_function():
    for i in range(100000):
        pass
timeit.timeit('my_function()', number=1000, globals=globals())
```

结果：timeit 模块用于测量小代码片段的执行时间，它通过多次运行代码来提供准确的时间估计。

5. line_profiler：

```
from line_profiler import LineProfiler
def my_function():
    a = 1
    b = a + 1
    for i in range(100000):
        b += i
lp = LineProfiler()
lp_wrapper = lp(my_function)
lp_wrapper()
```

```
lp.print_stats()
```

结果：line_profiler 是一个第三方库，它可以提供代码每一行的执行时间。

6. memory_profiler：

```
from memory_profiler import profile
@profile
def my_function():
    lst = [i for i in range(10000)]
    return lst
my_function()
```

结果：memory_profiler 用于分析内存使用情况，它可以显示每行代码的内存消耗。

（二）元编程

1. 动态函数创建：

```
def create_adder(n):
    def adder(x):
        return x + n
    return adder

add_five = create_adder(5)
print(add_five(10))
```

结果：使用高阶函数创建了一个加法函数 `add_five`，并输出 15。

2. 使用 `locals` 和 `globals` 动态修改变量：

```
def add_key_to_dict(key, value):
```

```
    globals()[key] = value

add_key_to_dict('new_var', 123)
print(new_var)
```

结果：在全局作用域中创建了一个名为 new_var 的变量，并赋值为 123。

3. 使用 type 动态创建类：

```
def create_class(name, bases, attrs):
    return type(name, bases, attrs)

class_attrs = {'x': 5, 'y': 10, '__add__': lambda self: self.x + self.y}
MyClass = create_class('MyClass', (object,), class_attrs)
obj = MyClass()
print(obj.x + obj.y)
```

结果：动态创建了一个名为 MyClass 的类，并输出 15。

4. 使用 setattr 动态设置属性：

```
class MyClass:
    pass

setattr(MyClass, 'new_method', lambda self: 'Hello, World!')
print(MyClass.new_method())
```

结果：为 MyClass 类动态添加了一个方法 new_method，并输出”Hello, World!”。

5. 使用 dir 动态获取对象属性：

```
class MyClass:
```

```
    def __init__(self):
        self.x = 5

my_obj = MyClass()
print(dir(my_obj))
```

结果：输出 MyClass 实例的所有属性和方法。

6. 使用 compile 动态执行代码：

```
code = "print('Hello, World!')"
compiled_code = compile(code, '<string>', 'exec')
exec(compiled_code)
```

结果：动态编译并执行了代码，输出”Hello, World!”。

7. 使用 inspect 模块检查对象：

```
import inspect

def my_function(x, y):
    return x + y
print(inspect.signature(my_function))
```

结果：输出函数 my_function 的签名。

（三）PyTorch 视觉应用

1. 张量初始化：

   (a) 直接从数据创建张量：

```
data = [[1, 2], [3, 4]]
x_data = torch.tensor(data)
```

(b) 从 NumPy 数组创建张量：

```
import numpy as np
np_array = np.array(data)
x_np = torch.from_numpy(np_array)
```

(c) 从另一个张量创建新张量：

```
x_ones = torch.ones_like(x_data) # retains the properties of x_data
print(f"Ones Tensor: \n {x_ones} \n")
x_rand = torch.rand_like(x_data, dtype=torch.float) # overrides the datatyp
print(f"Random Tensor: \n {x_rand} \n")
```

结果：

```
Ones Tensor:
 tensor([[1, 1],
        [1, 1]])
Random Tensor:
 tensor([[0.3947, 0.4858],
        [0.9286, 0.8274]], dtype=torch.float32)
```

2. 张量属性：

```
import torch
tensor = torch.rand(3, 4)
print(f"Shape of tensor: {tensor.shape}")
print(f"Datatype of tensor: {tensor.dtype}")
print(f"Device tensor is stored on: {tensor.device}")
```

结果：

```
Shape of tensor: torch.Size([3, 4])
Datatype of tensor: torch.float32
```

```
Device tensor is stored on: cpu
```

3. NumPy 桥接：

   (a) Tensor 到 NumPy 数组：

```
t = torch.ones(5)
print(f"t: {t}")
n = t.numpy()
print(f"n: {n}")
```

   结果：

```
t: tensor([1., 1., 1., 1., 1.])
n: [1. 1. 1. 1. 1.]
```

   (b) NumPy 数组到 Tensor：

```
import numpy as np
n = np.ones(5)
t = torch.from_numpy(n)
np.add(n, 1, out=n)
print(f"t: {t}")
print(f"n: {n}")
```

   结果：

```
t: tensor([2., 2., 2., 2., 2.], dtype=torch.float64)
n: [2. 2. 2. 2. 2.]
```

4. 创建模型：

```
import torch.nn as nn
model = nn.Sequential(
    nn.Linear(20, 10),
```

```
    nn.ReLU(),
    nn.Linear(10, 2),
    nn.LogSoftmax(dim=1)
)
```

5. GPU 加速：

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
x = torch.tensor([1, 2, 3]).to(device)
print(x)
```

结果：

```
tensor([1, 2, 3], device='cuda:0')
```

6. 损失函数（Loss Function）：

```
criterion = nn.MSELoss()
output = model(x)
target = torch.tensor([1.0, 2.0, 3.0], device=device)
loss = criterion(output, target)
print(loss)
```

结果：

```
tensor(5., device='cuda:0')
```

7. 自动微分（Autograd）：

```
x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
y = x ** 2
y.backward()
```

```
print(x.grad)
```

结果：

```
tensor([2., 4., 6.])
```

8. 数据加载和预处理：

```
from torchvision import datasets, transforms
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])
train_dataset = datasets.MNIST(root='./data', train=True, download=True, transfo
train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=32,
```

9. 定义神经网络：

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

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(3, 5)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        return x

net = Net()
print(net)
```

结果：

```
Net((fc1): Linear(in_features=3, out_features=5, bias=True))
```

二、解题感悟

Python 的 pdb 调试器可以逐行执行代码，检查变量状态，这对于理解复杂逻辑和解决难以发现的 bug 非常有帮助。除此之外，我们可以通过使用 cProfile、timeit 或 line_profiler 等工具，可以准确地测量代码的运行时间和性能瓶颈，从而进行有针对性的优化。在学习过程中，我发现元编程是一种强大的编程范式，它可以让代码更加灵活。我们可以通过合理使用元编程可以极大地提高代码的灵活性和可维护性。PyTorch 是一个功能强大的深度学习框架，它提供了动态计算图和丰富的神经网络构建工具。在简单的学习后，我掌握了一些基本技能，如创建模型，修改模型等等。

总之，这些工具和技术是现代软件开发的重要组成部分。它们不仅可以帮助我们提高开发效率，还可以让我们编写更高质量、更高效的代码。不断学习和实践是我们提高这些技能的关键。

Github 地址：`https://github.com/Gao-py/class_homework`