

Python 调试和性能分析技巧及 PyTorch 代码示例报告

郭路通 23020007032

1 Python 调试和性能分析技巧

以下是一些具体的 Python 调试和性能分析技巧及其完整代码示例：

1. 使用打印语句进行调试:

```
def sum_numbers(a, b):  
    print(f"Adding {a} and {b}")  
    return a + b  
  
result = sum_numbers(3, 5)  
print(result)
```

打印中间结果以帮助调试。

2. 使用 pdb 进行交互式调试:

```
import pdb  
  
def divide(x, y):  
    result = x / y  
    return result  
  
def main():  
    try:  
        a = 10  
        b = 0  
        print(a, "/", b)  
        divide(a, b)  
    except ZeroDivisionError:  
        print("捕获到异常，正在启动调试器...")  
        pdb.post_mortem()  
  
if __name__ == "__main__":  
    main()
```

设置断点，使用 pdb 进行调试。

```

E:\python\python.exe E:\元编程\pdb_test.py
10 / 0
捕获到异常，正在启动调试器...
> e:\元编程\pdb_test.py(3)divide()
-> result = x / y
(Pdb) bt
  e:\元编程\pdb_test.py(11)main()
-> divide(a, b)
> e:\元编程\pdb_test.py(3)divide()
-> result = x / y
(Pdb) n

```

3. 计算运行时间:

```

import time

def time_decorator(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"{func.__name__} took {end_time - start_time}
              seconds to run")
        return result
    return wrapper

@time_decorator
def example_function(n):
    sum = 0
    for i in range(n):
        sum += i
    return sum

example_function(100000000)

```

测量代码片段的执行时间。

```

E:\python\python.exe E:\元编程\计算函数执行时间.py
example_function took 2.6461691856384277 seconds to run

```

4. 使用 cProfile 进行详细性能分析:

```

import cProfile

def fib(n):
    if n <= 1:
        return n

```

```

        else:
            return fib(n-1) + fib(n-2)

cProfile.run('fib(30)')

```

进行详细的性能分析。

```

E:\python\python.exe E:\元编程\cprofile_test.py
2692540 function calls (4 primitive calls) in 0.534 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      1   0.000    0.000    0.534    0.534 <string>:1(<module>)
2692537/1   0.534    0.000    0.534    0.534 cprofile_test.py:2(fib)
      1   0.000    0.000    0.534    0.534 {built-in method builtins.exec}
      1   0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}

```

5. 使用 timeit 计算函数的运行时间:

```

import timeit

def test_function():
    result = 0
    for i in range(1000):
        result += i

# 使用 timeit 测试函数执行时间
execution_time = timeit.timeit(test_function, number=100)

print(f"执行时间: {execution_time} 秒")

```

计算函数运行时间

```

E:\python\python.exe E:\元编程\timeit_test.py
执行时间: 0.002118400007020682 秒

```

2 Python 元编程示例

以下是一些具体的 Python 元编程示例及其完整代码示例:

1. 使用类装饰器动态修改类:

```

def add_greeting(cls):
    def say_hello(self):
        print("Hello!")

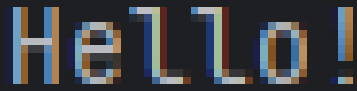
    cls.say_hello = say_hello
    return cls

@add_greeting
class MyClass:
    pass

```

```
obj = MyClass()
obj.say_hello()
```

动态为类添加方法。

A digital art style illustration of the word "Hello!" in a colorful, pixelated font on a dark background.

2. 使用元类动态修改类:

```
class Meta(type):
    def __new__(cls, name, bases, dct):
        dct['class_level'] = 'modified by metaclass'
        return super().__new__(cls, name, bases, dct)

class MyClass(metaclass=Meta):
    pass

print(MyClass.class_level)
```

在类创建时修改类属性。

A digital art style illustration of the Chinese characters "元类动态修改类" (Metaclass Dynamic Modification Class) in a colorful, pixelated font on a dark background.

3. 使用反射动态设置类属性:

```
class MyClass:
    pass

setattr(MyClass, 'dynamic_attribute', 'dynamic_value')

obj = MyClass()
print(obj.dynamic_attribute)
```

动态设置类属性。

A digital art style illustration of the text "dynamic_value" in a colorful, pixelated font on a dark background.

4. 使用装饰器修改函数行为:

```
def add_logging(func):
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__} with args: {args}, kwargs: {kwargs}")
```

```

        return func(*args, **kwargs)
    return wrapper

@add_logging
def greet(name):
    print(f"Hello, {name}!")

greet("Guolutong")

```

装饰器修改函数行为。

```

Calling greet with args: ('Guolutong',), kwargs: {}
Hello, Guolutong!

```

5. 使用闭包记住外部变量:

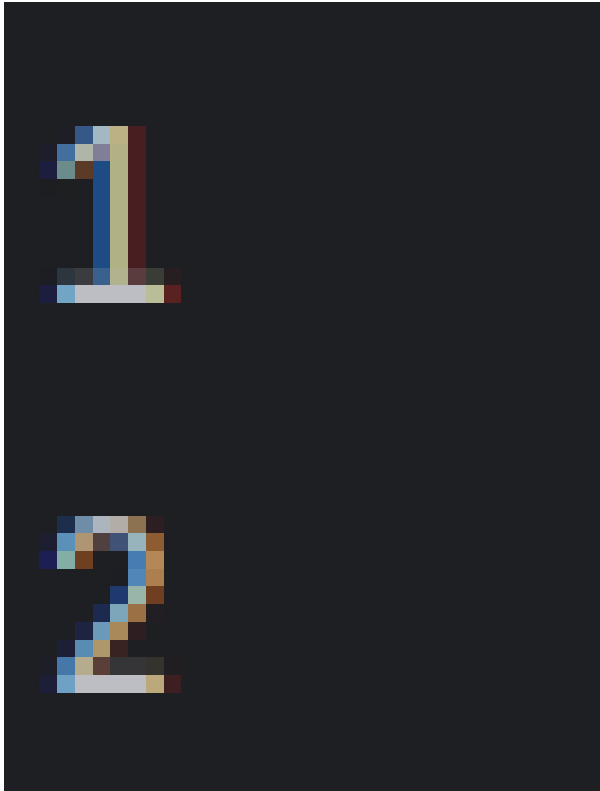
```

def make_counter():
    count = [0] # 使用列表来绕过局部变量不可变的问题
    def counter():
        count[0] += 1
        return count[0]
    return counter

counter = make_counter()
print(counter())
print(counter())

```

闭包记住外部变量。



6. 使用 `getattr` 和 `setattr` 动态获取和设置属性:

```
class DynamicAttr:
    def __getattr__(self, name):
        return f"Value for {name}"

    def __setattr__(self, name, value):
        object.__setattr__(self, name, value)

obj = DynamicAttr()
print(obj.some_attribute)
obj.some_attribute = 5
print(obj.some_attribute)
```

```
Value for some_attribute
5
```

7. 使用 `exec` 执行字符串形式的代码:

```
code = """
def hello_world():
    print("Hello, World!")
"""
exec(code)
hello_world()
```

此时代码会报错，但这其实可以编译运行。

```
Hello, World!
```

3 PyTorch 代码示例

以下是一些具体的 PyTorch 代码示例及其完整代码示例:

1. 创建张量:

```
import torch

x = torch.tensor([1.0, 2.0, 3.0])
print(x)
```

创建一个张量。

```
tensor([1., 2., 3.])
```

2. 张量加法:

```
import torch

x = torch.tensor([1.0, 2.0, 3.0])
print(x)
y = torch.tensor([4.0, 5.0, 6.0])
z = x + y
print(z)
```

两个张量相加。

```
tensor([1., 2., 3.])
tensor([5., 7., 9.])
```

3. 张量乘法:

```
import torch

x = torch.tensor([1.0, 2.0, 3.0])
y = torch.tensor([4.0, 5.0, 6.0])
w = torch.matmul(x, y)
print(w)
```

两个张量的矩阵乘法。

```
tensor(32.)
```

4. 训练模型:

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

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

# 加载训练数据
trainset = datasets.MNIST(root='./data', train=True, download=True,
    transform=transform)
trainloader = DataLoader(trainset, batch_size=4, shuffle=True)
```

```

# 定义网络
class SimpleNet(nn.Module):
    def __init__(self):
        super(SimpleNet, self).__init__()
        self.fc1 = nn.Linear(784, 128) # 784 is the size of the
            flattened image
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = x.view(x.size(0), -1) # Flatten the image
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

model = SimpleNet()

# 损失函数和优化器
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

# 训练模型
for epoch in range(20): # 迭代20个epoch
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        if (i + 1) % 100 == 0:
            print(f'Epoch [{epoch + 1}/20], Step [{i + 1}/600],
                Loss: {loss.item():.4f}')

torch.save(model.state_dict(), 'output')

```

训练了一个手写数字识别的模型, 下载了一个 data, 里面有 raw, 用于训练, 自己创建一个.pth 文件, 用于保存模型, 训练了一个可以预测手写数字的模型, 这里训练 20 圈

t10k-images-idx3-ubyte	2024/9/16 2:41	文件	7,657 KB
t10k-images-idx3-ubyte.gz	2024/9/16 2:41	压缩存档文件夹	1,611 KB
t10k-labels-idx1-ubyte	2024/9/16 2:41	文件	10 KB
t10k-labels-idx1-ubyte.gz	2024/9/16 2:41	压缩存档文件夹	5 KB
train-images-idx3-ubyte	2024/9/16 2:41	文件	45,938 KB
train-images-idx3-ubyte.gz	2024/9/16 2:41	压缩存档文件夹	9,681 KB
train-labels-idx1-ubyte	2024/9/16 2:41	文件	59 KB
train-labels-idx1-ubyte.gz	2024/9/16 2:41	压缩存档文件夹	29 KB

5. 使用模型:

```
import torch
from PIL import Image
import torchvision.transforms as transforms
import torch.nn as nn
class SimpleNet(nn.Module):
    def __init__(self):
        super(SimpleNet, self).__init__()
        self.fc1 = nn.Linear(784, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# 加载模型
model = SimpleNet()
model_state_dict = torch.load('E:\pytorch_learn\output\\num.pth')
model.eval()

# 图像预处理
transform = transforms.Compose([
    transforms.Grayscale(), # 转换为灰度图
    transforms.Resize((28, 28)), # 调整图像大小为28x28
    transforms.ToTensor(), # 转换为Tensor
    transforms.Normalize((0.5,), (0.5,)) # 归一化
])

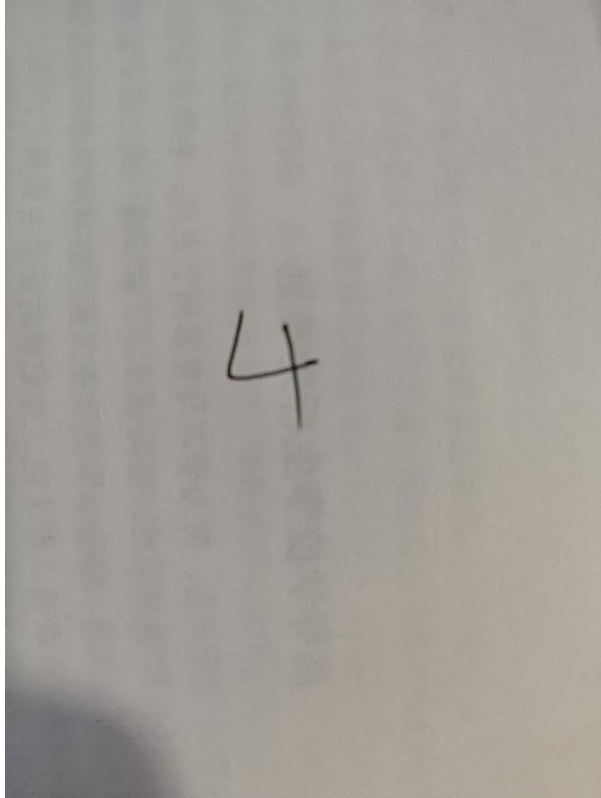
# 加载图像
image_path = "E:\pytorch_learn\input\\6c004d86e8eab8dd654677cf1edd9b9.jpg"
image = Image.open(image_path).convert('L') # 转换为灰度图

# 预处理图像
tensor = transform(image).unsqueeze(0) # 增加一个批次维度
```

```
# 预测
with torch.no_grad():
    output = model(tensor)
    _, predicted = torch.max(output, 1)

# 显示结果
print(f'Predicted digit: {predicted.item()}')
```

使用该模型来预测手写数字，给它一张图片，他能预测数字，结果并不算理想



Predicted digit: 4

6. 使用 GPU:

```
import torch

# 检查是否有可用的CUDA设备
if torch.cuda.is_available():
    device = torch.device("cuda:0")
    print("Using CUDA")
else:
    device = torch.device("cpu")
    print("Using CPU")

# 将张量移动到GPU
tensor = torch.tensor([1.0, 2.0]).to(device)
print(tensor)
```

将模型和数据移到 GPU 上。

7. 自动求导:

```
import torch

# 创建一个张量, requires_grad 设置为 True
x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)

# 进行一些操作
y = x ** 2

# 计算梯度
y.backward(torch.tensor([1.0, 1.0, 1.0]))

# 打印梯度
print(x.grad)
```

输出该张量梯度

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

8. 使用 pytorch 进行积分:

```
import torch

# 定义一个函数
def f(t):
    return t * torch.sin(t)

# 定义一个范围
t = torch.linspace(0., 5., 100)

# 近似积分
integral_approximation = torch.trapz(f(t), t)
print(integral_approximation)
```

用了梯形法则。

```
tensor(-2.3771)
```

9. 使用 pytorch 进行随机采样:

```
import torch

# 创建一个张量
```

```
x = torch.rand(5, 3)

# 随机选择一个元素
index = torch.randint(low=0, high=x.shape[0], size=(1,))
selected_element = x[index]
print(selected_element)
```

随机采样。

```
tensor([[0.8289, 0.1371, 0.6951]])
```

4 感悟

1. 通过学习的调试和性能分析技巧，我发现这些工具对于找出代码中的错误和提高程序效率非常有帮助。尤其是 ‘pdb’ 和 ‘cProfile’，可以让我找到问题出现的地方。
2. 元编程的概念让我意识到 Python 的灵活性，通过动态生成或修改类和函数，可以实现更高效的编程模式。例如，使用类装饰器可以轻松地为类添加新的功能，还有一些能够将字符串变成代码的操作，都让我眼前一亮。
3. PyTorch 的使用让我感受到了深度学习的强大之处。从简单的张量操作到复杂的模型训练，PyTorch 提供了一套完整的工具链，使得训练变得更加容易，曾经我也训练过一些语言模型，对这个十分感兴趣，但用的都是别人现成的代码，这次是自己的代码，虽然过程十分曲折，但还是学习到很多新知识
4. 我希望在未来的学习和工作中，能够更加熟练地运用这些工具和技术，提高自己的工作效率。

5 GitHub 链接

<https://github.com/gleati/work>