



## 实验报告四

课程：系统开发工具基础

姓名：陈培诺

学号：23160001003

时间：2024年9月15日

### 目录

<b>1</b>	<b>调试与性能分析</b>	<b>3</b>
1.1	实例1: pdb调试器 . . . . .	3
1.2	实例2: 打印调试 . . . . .	3
1.3	实例3: 日志调试 . . . . .	4
1.4	实例4: 计时 . . . . .	4
1.5	实例5: shellcheck分析程序错误原因 . . . . .	5
<b>2</b>	<b>元编程</b>	<b>6</b>
2.1	实例6: makefile . . . . .	6
<b>3</b>	<b>pytorch</b>	<b>6</b>
3.1	实例7: 创建张量 . . . . .	6
3.2	实例8: 张量的运算 . . . . .	6

3.3	实例9: 随机生成矩阵 . . . . .	7
3.4	实例10: 全零矩阵 . . . . .	8
3.5	实例11: 张量的属性 . . . . .	9
3.6	实例12: 张量的操作 . . . . .	9
3.7	实例13: 张量转变为numpy数组 . . . . .	10
3.8	实例14: 构建神经网络 . . . . .	10
3.9	实例15: Dataloader介绍 . . . . .	11
3.10	实例16: 张量的合并 . . . . .	12
3.11	实例17: 张量广播运算 . . . . .	12
<b>4</b>	<b>大杂烩</b>	<b>13</b>
4.1	实例18: API . . . . .	13
4.2	实例19: VPN . . . . .	13
4.3	实例20: 修改键位映射 . . . . .	14
<b>5</b>	<b>git仓库链接</b>	<b>14</b>
<b>6</b>	<b>学习心得</b>	<b>14</b>

# 1 调试与性能分析

## 1.1 实例1: pdb调试器

- l- 显示当前行附近的 11 行或继续执行之前的显示
- s- 执行当前行，并在第一个可能的地方停止
- n- 继续执行直到当前函数的下一条语句或者 return 语句
- b- 设置断点（基于传入的参数）
- p- 在当前上下文对表达式求值并打印结果
- r- 继续执行直到当前函数返回
- q- 退出调试器

```
import pdb
pdb.set_trace()
def factorial(n): 2 usages
    if n == 0 or n == 1:
        return 1
    else:
        return n*factorial(n-1)
sum = 0
sum += factorial(i)
print("1+2!+3!+...+10!的结果为:{}".format(sum))
```

图 1: 实例1: pdb调试器调用

## 1.2 实例2: 打印调试

调试最简单的方式就是打印输出，而print函数就可以输出各种类型变量，配合着格式化输出，我们可以打印出程序运行过程中各个变量的状态值。

例如下面这个例子

```


if __name__ == '__main__':
     a=1
    print(f'值:{a}')
```

图 2: 实例2: 打印调试

使用这种方式的好处是我们不需要引入其它包，我们只需要使用简单的print就可以调试我们的程序，当然，它的缺点也很明显，有时候为了调试一些变量，我们不得不写很多print语句，而且有时候为了更优雅地显示数据，我们不得不写很多代码。

### 1.3 实例3: 日志调试

```

import logging

logging.basicConfig(level=logging.DEBUG, format='%(asctime)s - %(levelname)s - %(message)s')

# 最低级别。用于小细节。通常只有在诊断问题时，你才会关心这些消息
logging.debug('Some debugging details.')
# 用于记录程序中一般事件的信息，或确认一切工作正常
logging.info('The logging module is working.')
# 用于表示可能的问题，它不会阻止程序的工作，但将来可能会
logging.warning('An error message is about to be logged.')
# 用于记录错误，它导致程序做某事失败
logging.error('An error has occurred.')
# 最高级别。用于表示致命的错误，它导致或将要导致程序完全停止工作
logging.critical('The program is unable to recover!')
```

图 3: 实例3: 日志调试

### 1.4 实例4: 计时

调用time内的函数

```
2024-09-16 17:25:39,532 - DEBUG - Some debugging details.
2024-09-16 17:25:39,533 - INFO - The logging module is working.
2024-09-16 17:25:39,533 - WARNING - An error message is about to be logged.
2024-09-16 17:25:39,533 - ERROR - An error has occurred.
2024-09-16 17:25:39,533 - CRITICAL - The program is unable to recover!
```

图 4: 实例3: 日志调试

```
print('系统:', platform.system())

import time

T1 = time.perf_counter()

for i in range(100 * 100):
    pass

T2 = time.perf_counter()
print('程序运行时间:%s毫秒' % ((T2 - T1) * 1000))
```

图 5: 实例4: 计时

```
系统: Windows
程序运行时间:0.17110002227127552毫秒
```

图 6: 实例4: 计时

## 1.5 实例5: shellcheck分析程序错误原因

shellcheck+文件名即可得到程序错误原因

## 2 元编程

### 2.1 实例6: makefile

目标:即要生成的文件，如果目标文件的更新时间晚于依赖文件更新时间，则说明依赖文件没有改动，目标文件不需要重新编译。否则会进行重新编译并更新目标文件。默认情况下Makefile的第一个目标为终极目标。

依赖: 即目标文件由哪些文件生成。

命令: 即通过执行命令由依赖文件生成目标文件。注意每条命令之前必须有一个tab(此文档编辑器默认是空格，复制下来的代码需要把命令代码的缩进改为tab制表符)保持缩进，这是语法要求（会有一些编辑工具默认tab为4个空格，会造成Makefile语法错误）。

## 3 pytorch

### 3.1 实例7: 创建张量

```
import torch
a=torch.tensor([[1,2,3],[4,5,6],[7,8,9]])
print(a)
```

图 7: 实例7: 创建张量

### 3.2 实例8: 张量的运算

```
tensor([[1, 2, 3],  
        [4, 5, 6],  
        [7, 8, 9]])
```

图 8: 实例7: 创建张量

```
import torch  
a=torch.tensor([[1,2,3],[4,5,6],[7,8,9]])  
b=torch.tensor([[1,2,3],[4,5,6],[7,8,9]])  
print(a+b)
```

图 9: 实例8: 张量加法

```
tensor([[ 2,  4,  6],  
        [ 8, 10, 12],  
        [14, 16, 18]])
```

图 10: 实例8: 张量加法

### 3.3 实例9: 随机生成矩阵

随机生成一个3行3列矩阵

```
import torch
a=torch.rand(3,3)
print(a)
```

图 11: 实例9: 随机矩阵

```
tensor([[0.1576, 0.5610, 0.8697],
        [0.1688, 0.2930, 0.1287],
        [0.1939, 0.2799, 0.2043]])
```

图 12: 实例9: 随机矩阵

### 3.4 实例10: 全零矩阵

创建三行三列所有值为0的整型矩阵

```
import torch
a=torch.zeros(3,3,dtype=torch.int)
print(a)
```

图 13: 实例10: 0矩阵



```
tensor([[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]], dtype=torch.int32)
```

图 14: 实例10: 0矩阵

### 3.5 实例11: 张量的属性

张量的属性包括形状，数据类型和存储设备等

```
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}")

# Shape of tensor: torch.Size([3, 4])
# Datatype of tensor: torch.float32
# Device tensor is stored on: cpu
```

图 15: 实例11: 属性

### 3.6 实例12: 张量的操作

PyTorch中有100 多种张量运算，包括算术、线性代数、矩阵操作（转置、索引、切片）、采样等，而且这些操作中都可以在 GPU 上运行（通常以比 CPU 更高的速度）。

默认情况下，张量是在 CPU 上创建的。我们需要使用 `.to`方法明确地将张量移动到 GPU（在检查 GPU 可用性之后）。

### 3.7 实例13: 张量转变为numpy数组

```
import torch

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

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

图 16: 实例13

### 3.8 实例14: 构建神经网络

我们首先定义了一个Net类，这个类继承自nn.Module。然后在init方法中定义了网络的结构，在forward方法中定义了数据的流向。在网络的构建过程中，我们可以使用任何tensor操作。

需要注意的是，backward函数（用于计算梯度）会被autograd自动创建和实现。你只需要在nn.Module的子类中定义forward函数。

在创建好神经网络后，我们可以使用net.parameters()方法来返回网络的可学习参数。

```

class Net(nn.Module): 2 usages
    def __init__(self):
        super(Net, self).__init__()

        # 输入图像channel: 1, 输出channel: 6, 5x5卷积核
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5)
        self.conv2 = nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5)

        # 全连接层
        self.fc1 = nn.Linear(16 * 5 * 5, out_features=120)
        self.fc2 = nn.Linear(in_features=120, out_features=84)
        self.fc3 = nn.Linear(in_features=84, out_features=10)

    def forward(self, x):
        # 使用2x2窗口进行最大池化
        x = F.max_pool2d(F.relu(self.conv1(x)), kernel_size=(2, 2))
        # 如果窗口是方的, 只需要指定一个维度
        x = F.max_pool2d(F.relu(self.conv2(x)), kernel_size=2)

        x = x.view(-1, self.num_flat_features(x))

        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)

        return x

    def num_flat_features(self, x): 1 usage
        size = x.size()[1:] # 获取除了batch维度之外的其他维度
        num_features = 1
        for s in size:
            num_features *= s
        return num_features

```

图 17: 实例14

### 3.9 实例15: Dataloader介绍

DataLoader类提供了对数据集的并行加载, 可以有效地加载大量数据, 并提供了多种数据采样方式。常用的参数有:

- dataset: 加载的数据集 (Dataset对象)
- batch-size: batch大小

shuffle: 是否每个epoch时都打乱数据

num-workers: 使用多进程加载的进程数, 0表示不使用多进程

### 3.10 实例16: 张量的合并

```
import torch
x = torch.arange(12, dtype=torch.float32).reshape((3,4))
y = torch.tensor([[2.0,1,4,3],[1,2,3,4],[4,3,2,1]])
m = torch.cat( tensors: (x,y), dim=0) # 按行合并起来
n = torch.cat( tensors: (x,y), dim=1) # 按列合并起来
print(m)
print(n)
```

图 18: 实例16

```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.],
        [ 2.,  1.,  4.,  3.],
        [ 1.,  2.,  3.,  4.],
        [ 4.,  3.,  2.,  1.]])
tensor([[ 0.,  1.,  2.,  3.,  2.,  1.,  4.,  3.],
        [ 4.,  5.,  6.,  7.,  1.,  2.,  3.,  4.],
        [ 8.,  9., 10., 11.,  4.,  3.,  2.,  1.]])
```

图 19: 实例16

### 3.11 实例17: 张量广播运算

即使形状不同, 仍然可以通过调用广播机制(broadcasting mechanism)来执行按元素操作。就是说当矩阵的形式不符合相加的要求时会自动补齐, 并进行相加。

```
import torch
a = torch.arange(3).reshape((3,1))
b = torch.arange(2).reshape((1,2))
print(a)
print(b)
print(a+b) # a会复制出一个3*2的矩阵，b复制出一个3*2的矩阵，然后再相加，会得到一个3*2矩阵
```

图 20: 实例17

## 4 大杂烩

### 4.1 实例18: API

API 是应用程序编程接口,Pytorch使用Tensorboard主要用到了三个API:

**SummaryWriter:** 这个用来创建一个log文件，TensorBoard面板查看时，也是需要选择查看那个log文件。

**add-something:** 向log文件里面增添数据。例如可以通过add-scalar增添折线图数据，add-image可以增添图片。

**close:** 当训练结束后，我们可以通过close方法结束log写入

### 4.2 实例19: VPN

VPN指依靠ISP或其他NSP在公用网络基础设施之上构建的专用的安全数据通信网络，只不过这个专线网络是逻辑上的而不是物理的，所以称为虚拟专用网。

**虚拟:** 用户不再需要拥有实际的长途数据线路，而是使用公共网络资源建立自己的私有网络。

**专用:** 用户可以定制最符合自身需求的网络。

**核心技术:** 隧道技术

发出的流量看上去来源于 VPN 供应商的网络而不是你的“真实”地址，而你实际接入的网络只能看到加密的流量

### 4.3 实例20: 修改键位映射

推荐修改键位是因为键盘上很多不常用的功能却占用了比较方便的按键位置，例如CapsLock 可以在window控制面板将其修改

## 5 git仓库链接

<https://github.com/cpn-cyber/-.git>

## 6 学习心得

这是系统开发工具基础最后一节课，也是收获满满，学习了如pytorch等干货知识，在这四节课里，我大大提升了自己对计算机领域的认知，学习了许多重要的工具，在未来也要继续深入学习，为以后做相关开发打下基础，同时也感谢老师和助教在这四节课中的辛苦付出！