

NCUSCC 2024秋季考核Pytorch试题实验报告

笔记本: My Notebook

创建时间: 2024/10/12 22:56

更新时间: 2024/10/21 23:51

作者: xb5zomlf

URL: <https://docs.qq.com/aio/DWnlER3BvTXpRYXFa?p=GXQfTG3fHHluFPKcdQNlin>

NCUSCC 2024秋季考核 Pytorch试题实验报告

目录

1.实验环境搭建

- 虚拟机搭建
- GPU驱动装置
- PyTorch安装

2.数据集准备及训练模型搭建

- CIFAR-10 数据集的下载、处理、与格式转换
- PyTorch实现深度学习模型
- 训练模型与性能验证

3.模型优化与加速

- GPU & CPU对比
- 调整 batch size
- 混合精度训练
- 调整workers
- 使用cudnn

- 调整输入通道
- 其他负优化的行为(批量一体化、调整学习率.....)

4.不同优化方式性能可视化

5.实验过程中遇到的问题及解决方案

- 虚拟机平台选择
- GPU驱动装置与CUDAToolkit下载方式
- PyTorch安装须知
- CIFAR的正确下载方法
- 负优化的原因

6.参考资料&特别鸣谢

I .实验环境搭建

①.虚拟机搭建

VMware? ✕

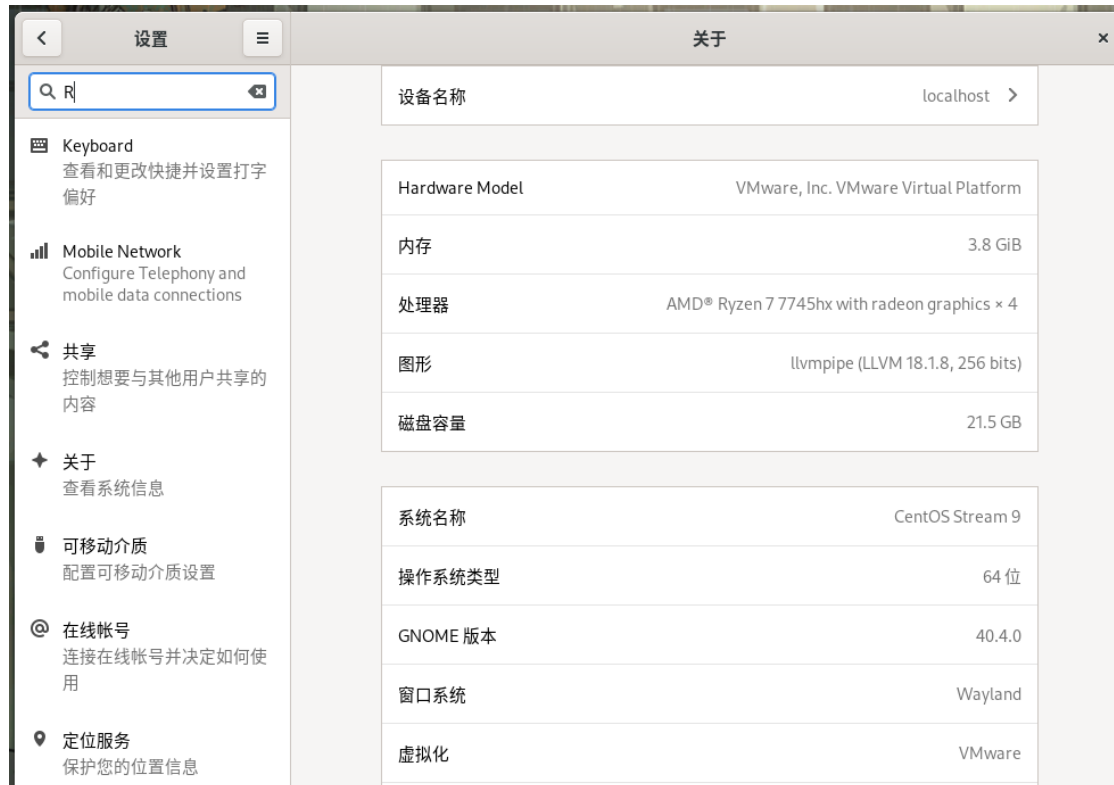
WSL2! ✓

首先，我们要知道此次考核的深度学习要求我们使用NVIDIA驱动，从而实现训练模型在GPU上的运行，因此我们虚拟机的GPU也应为NVIDIA，方便后面CUDA的使用以及模型的训练。

VMware的致命缺陷

与WSL不同，VMware等传统虚拟机软件创建的是一个完全隔离的虚拟环境，每个虚拟机都需要有自己的操作系统和驱动程序。通俗来讲，也就是VMware上的Linux系统几乎完全独立于我们的主机，通常使用虚拟化的硬件，这里面也就包括虚拟GPU，这些虚拟硬件需要在虚拟机内部模拟，性能通常不如直接访问主机硬件的WSL2。想要做到VMware直通主系统，是十分复杂且困难的，因此，本次试题我采用WSL搭建虚拟机环境，接下来我

将讲解WSL搭建 Ubuntu 22.04 LTS 操作系统 的过程

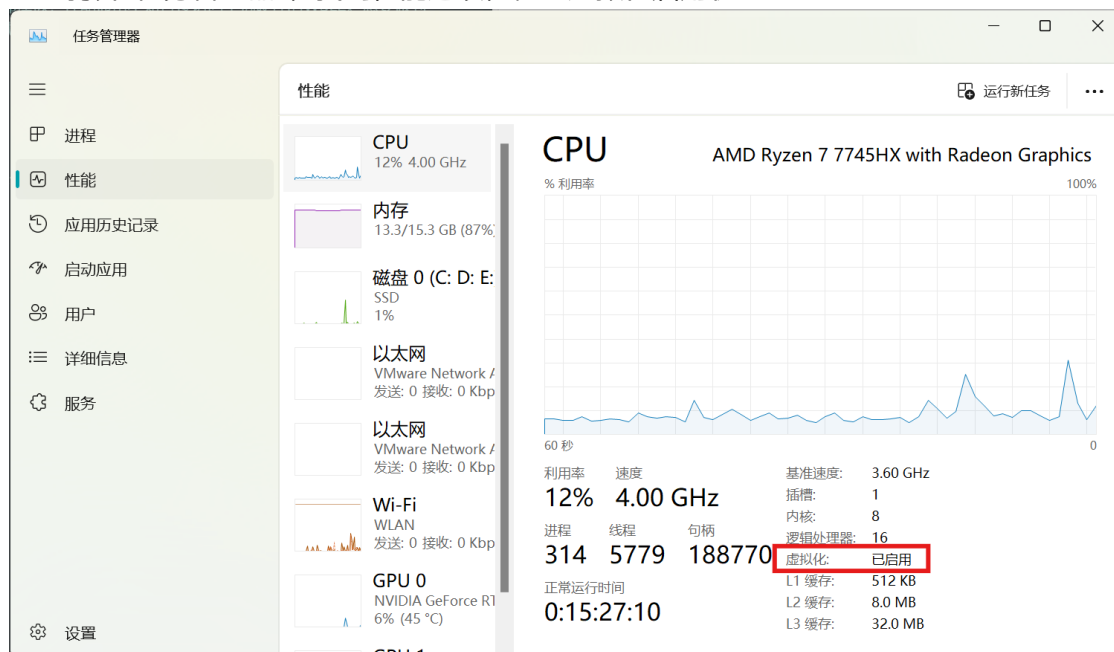


(此时我VMware上的乌班图已经删掉了，故用CentOS 9 为例)
可以看到，此时的GPU是一个虚拟的

WSL2搭建Ubuntu 22.04 LTS

1.检查虚拟化设置

打开"任务管理器", 找到性能选项, 检查虚拟化启用状态



确保虚拟化设置正确，方便后面搭建乌班图虚拟机

2.安装wsl2

以管理员身份打开Windows PowerShell 并执行以下指令

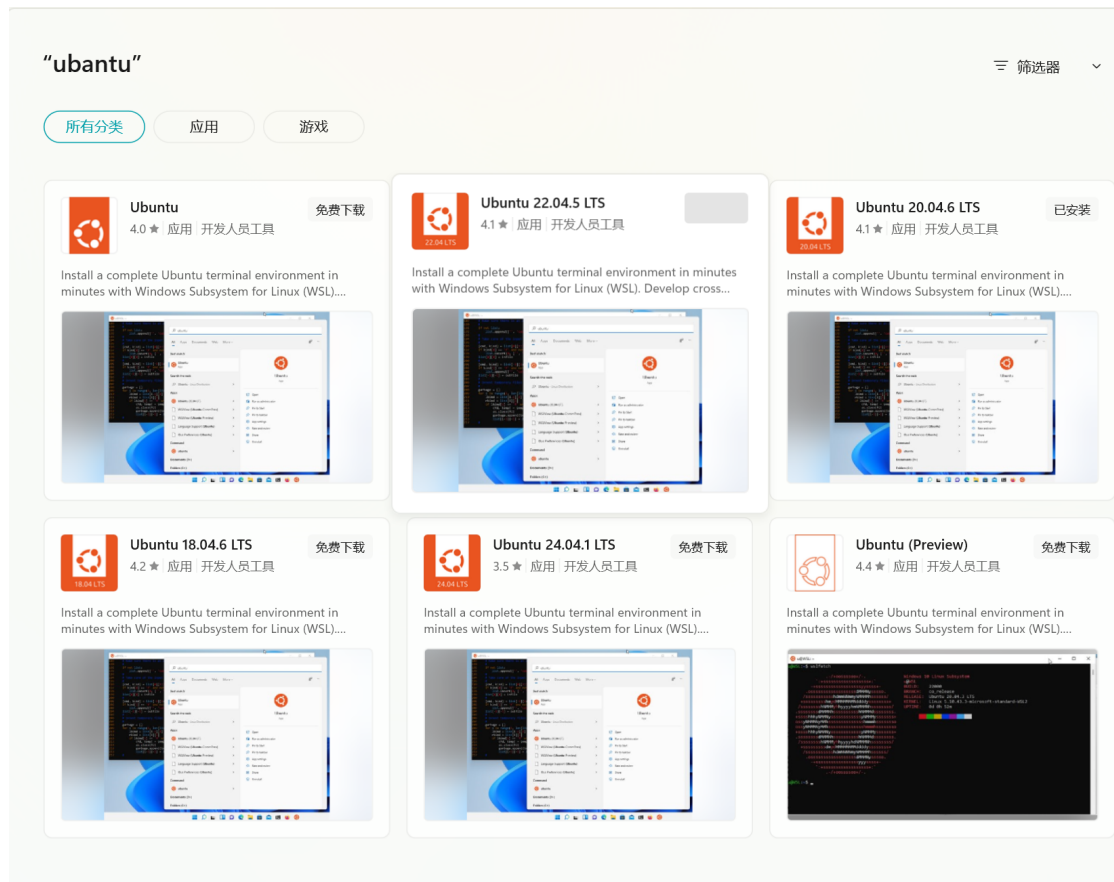
```
wsl --set-default-version 2
```

```
wsl --update
```

完成后重启系统

3.搭建 Ubuntu 22.04 LTS

搭建乌班图虚拟机有很多种方法，包括通过命令行下载安装，以及在Windows自带的应用商城中下载等，为了节约时间，这里直接使用第二种方法。



选择需要的版本下载后直接打开,设置初始账号密码

为了方便后续的操作，我在命令行中输入 `sudo -i` 进入root账户

②.NVIDIA 驱动和 CUDA Toolkit的安装

在这个部分正式开始之前，我们先要弄明白，我们为什么要安装NVIDIA驱动和CUDA Toolkit，它们对于深度学习作用是什么？不能知其然而不知其所以然。

NVIDIA的作用

- 1.硬件兼容性：NVIDIA 驱动确保了操作系统能够识别和正确使用 NVIDIA 的 GPU 硬件。没有正确的驱动程序，GPU 可能无法在系统中被识别，或者无法正常工作。
- 2.性能优化：NVIDIA 定期更新驱动程序，以优化 GPU 的性能。这些更新可能包括对新硬件的支持、对旧硬件的性能改进、以及对深度学习框架的优化。
- 3.CUDA 支持：NVIDIA 驱动通常包含对 CUDA (Compute Unified Device Architecture) 的支持，这是 NVIDIA 提供的一个并行计算平台和编程模型。深度学习框架如 TensorFlow 和 PyTorch 依赖于 CUDA 来利用 NVIDIA GPU 的并行处理能力。（换句话说，这两者存在双向联动）

CUDA Toolkit的作用

- 1.加速计算：深度学习涉及大量的矩阵运算和数据并行处理，这些计算在 GPU 上能比在 CPU 上更快地执行。CUDA Toolkit 提供了一套 API，允许开发者直接在 NVIDIA 的 GPU 上执行计算，显著提高了深度学习模型训练和推理的速度。
- 2.GPU 编程：CUDA Toolkit 提供了一套完整的工具集，包括编译器、库和调试工具，使得开发者能够为 NVIDIA GPU 编写并优化程序。这使得深度学习框架能够在 GPU 上实现高效的并行运算。
- 3.灵活性和控制：CUDA Toolkit 提供了较低级别的控制，允许开发者自定义 GPU 上的并行计算，从而为特定的深度学习任务优化性能。
- 4.研究和开发：在深度学习的研究领域，快速迭代和实验是非常重要的。CUDA Toolkit 提供的 GPU 加速能力使得我们得以快速训练和测试新的模型架构和算法。

1.安装Nvidia驱动

在命令行中输入 `nvidia-smi` 查询自己的显卡驱动版本，这里我的版本已经是新版，故跳过更新步骤，稍后我会在第五个大板块讲解更新步骤

```
8188 MiB, 115 MiB, 7843 MiB
olddove@localhost: ~$ free -h
              total        used          free      shared  buff/cache   available
Mem:           7.4Gi        1.2Gi        4.8Gi          71Mi        1.4Gi        5.8Gi
Swap:          2.0Gi           0B          2.0Gi
olddove@localhost: ~$ nvidia-smi
Sat Oct 12 15:36:07 2024
+-----+
| NVIDIA-SMI 565.51.01                  Driver Version: 565.90          CUDA Version: 12.7          |
+-----+-----+
| GPU   Name                               Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf              Pwr:Usage/Cap |      Memory-Usage | GPU-Util  Compute M. |
|=====+=====+
|  0  NVIDIA GeForce RTX 4060 ...      On          | 00000000:01:00:00 Off |         0%      N/A |
| N/A   49C   P8               2W / 80W | 214MiB / 8188MiB |             Default  |
|=====+=====+
+-----+
| Processes:                               |
| GPU   GI    CI        PID   Type   Process name          | GPU Memory |
| ID   ID                               |            | Usage           |
|=====+=====+
|  0   N/A   N/A         65365   G     /code                  |      N/A   |
+-----+
```

2.安装CUDA Toolkit

在安装CUDA Toolkit之前，我们应该先下载gcc，以防止后面下载时报错临时安装浪费时间，因为CUDA Toolkit的安装刚需gcc环境。

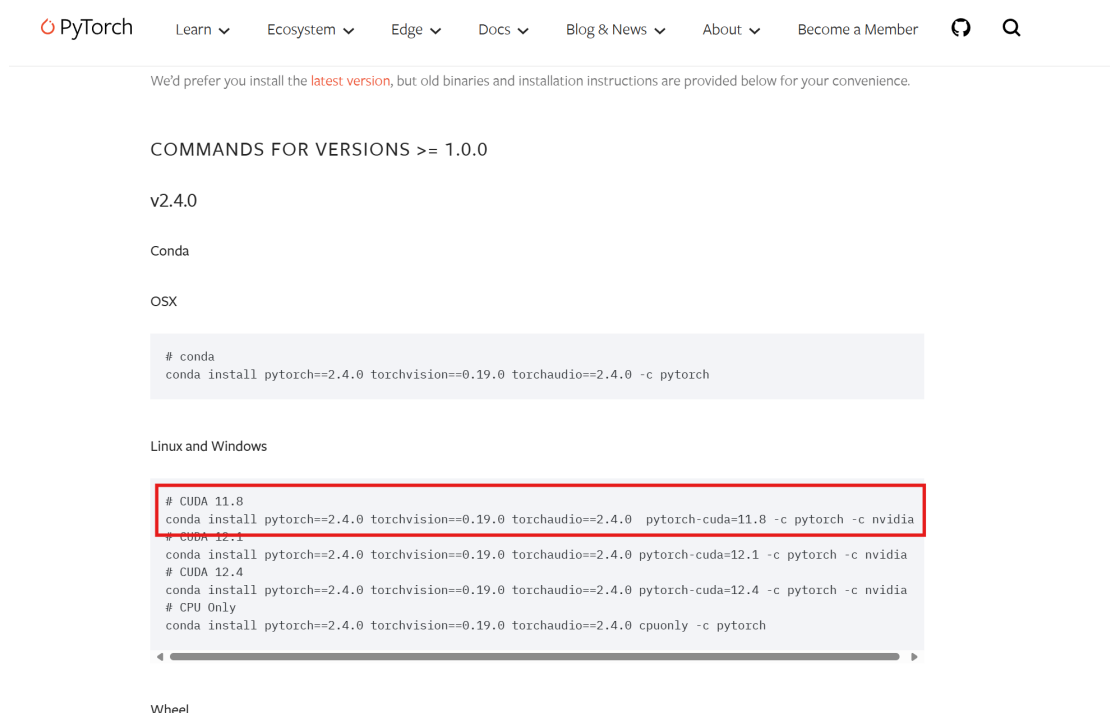
```
sudo apt install gcc
```

我们也可以同时下载一些配套的基础环境

```
sudo apt install build-essentia
```

安装CUDA的顺序有两种，一种是决定pytorch的版本以此来选择cuda版本，第二种是先决定cuda的版本在以此来选择pytorch的版本，在这里我先是第二种顺序进行操作，但最终出现了巨大失误导致必须重装系统，这里我将以第一种顺序讲解安装过程，稍后我将会在第五个大板块中讲解出错原因

打开 [PyTorch](#)



打开上方网站，我们可以查询到各版本的PyTorch支持的CUDA版本

这里我选择安装较早的11.8版本，以避免版本过新BUG多和版本过老功能缺失的问题

由网站可知支持PyTorch11.8版本的CUDA版本为11.8，这就是我们要下载的版本

决定好我们要下载的CUDA版本后 打开 [CUDA](#)

找到我要下载的版本后，根据我的系统版本，如图选择

Select Target Platform

Click on the green buttons that describe your target platform. Only supported platforms will be shown. By downloading and using the software, you agree to fully comply with the terms and conditions of the [CUDA EULA](#).

Operating System	Linux	Windows							
Architecture	x86_64	ppc64le	arm64-sbsa	aarch64-jetson					
Distribution	CentOS	Debian	Fedora	KylinOS	OpenSUSE	RHEL	Rocky	SLES	Ubuntu
Version	18.04	20.04	22.04						
Installer Type	deb (local)	deb (network)	runfile (local)						

Download Installer for Linux Ubuntu 22.04 x86_64

The base installer is available for download below.

> Base Installer

Installation Instructions:

```
$ wget https://developer.download.nvidia.com/compute/cuda/11.8.0/local_installers/cuda_11.8.0_520.61.05_linux.run
$ sudo sh cuda_11.8.0_520.61.05_linux.run
```

接下来，在Ubuntu命令行输入网页提供给我们的代码，进入CUDA的安装程序

```
End User License Agreement
-----

The CUDA Toolkit End User License Agreement applies to the
NVIDIA CUDA Toolkit, the NVIDIA CUDA Samples, the NVIDIA
Display Driver, NVIDIA Nsight tools (Visual Studio Edition),
and the associated documentation on CUDA APIs, programming
model and development tools. If you do not agree with the
terms and conditions of the license agreement, then do not
download or use the software.

Last updated: Mar 24, 2021.

Preface
-----

The Software License Agreement in Chapter 1 and the Supplement
in Chapter 2 contain license terms and conditions that govern
the use of NVIDIA software. By accepting this agreement, you

Do you accept the above EULA? (accept/decline/quit):
█
```

(当时忘记截图了，暂时用一下别人的图)

输入accept

接下来一直回车，等待安装完毕即可

接下来，我们进行环境配置

输入`vim ~/.bashrc` 编辑该文件

将下列配置写进去

```
export
LD_LIBRARY_PATH=/usr/local/cuda/lib64:/usr/local/cuda/extras/CUPTI/lib64
```

```
export CUDA_HOME=/usr/local/cuda/bin  
export PATH=$PATH:$LD_LIBRARY_PATH:$CUDA_HOME
```

保存并退出，输入 `source ~/.bashrc` 完成配置更新
输入 `nvcc -V` 检查配置是否正确

```
olddove@localhost:~$ nvcc -V  
nvcc: NVIDIA (R) Cuda compiler driver  
Copyright (c) 2005-2022 NVIDIA Corporation  
Built on Wed_Sep_21_10:33:58_PDT_2022  
Cuda compilation tools, release 11.8, V11.8.89  
Build cuda_11.8.r11.8/compiler.31833905_0  
olddove@localhost:~$
```

如图，我们可以知道我们成功下载了CUDA11.8版本，可喜可贺

!!!! 注：这几部一定要在root账户下完成，否则可能出现下载下来的版本与安装包不匹配的情况

③.Anaconda与PyTorch的安装

1.conda环境的搭建

在讲解conda环境搭建过程之前，我要先讲解一下为什么我们要搭建conda环境来下载PyTorch，而不是直接使用pip3进行安装。

1.环境隔离：Anaconda 提供了环境管理功能，允许我们为不同的项目创建隔离的环境。如果不使用 Anaconda，我们可能需要需要手动管理 Python 环境，以防止不同项目之间的依赖冲突。更通俗的说，使用pip3下载可能会对我们执行不同的项目造成影响。

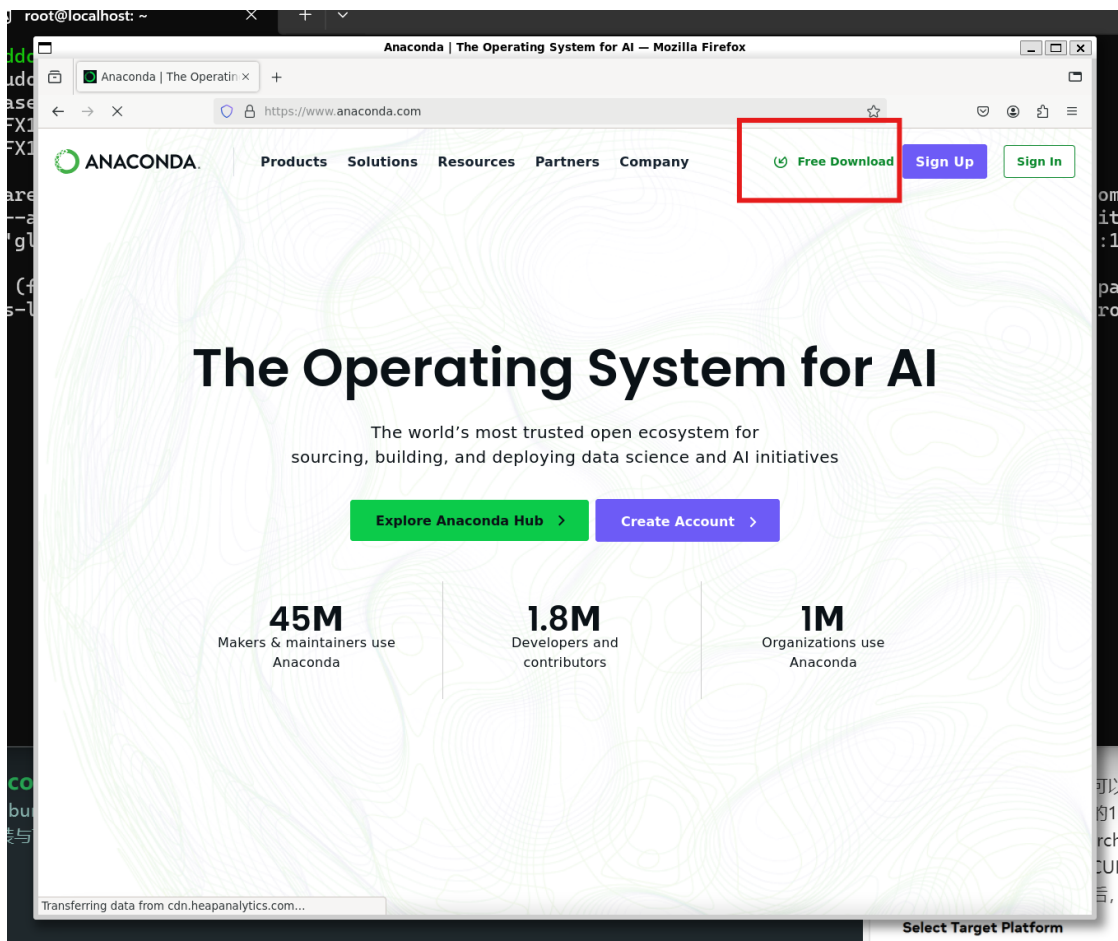
2.缺少高级功能：Anaconda 提供了一些高级功能，如环境导出导入、包搜索等，这些在不使用 Anaconda 的情况下可能不可用。

3.系统资源：手动管理 Python 环境和库可能会导致系统资源的混乱，特别是当多个项目需要不同版本的相同库时。

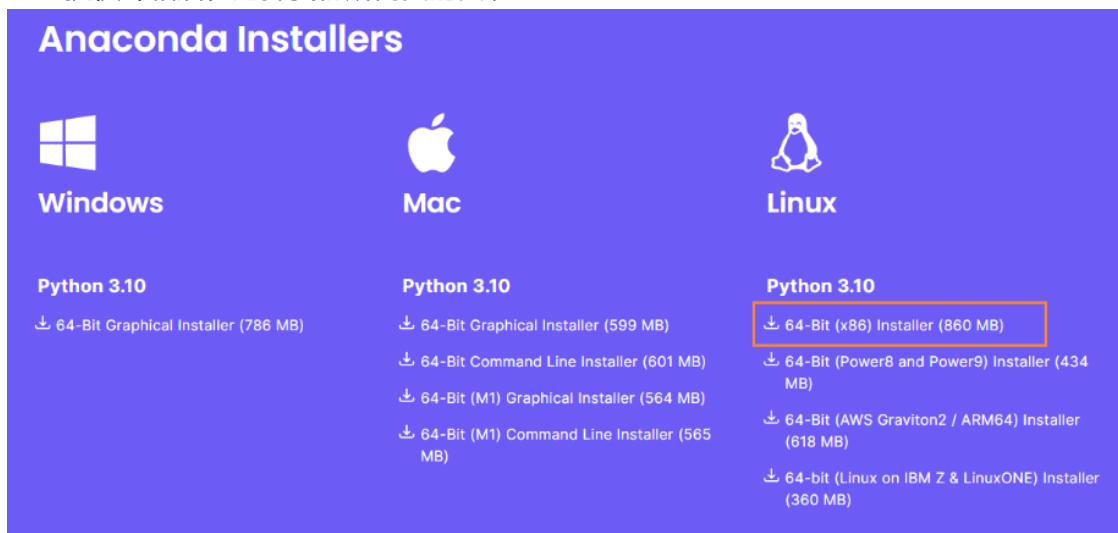
4.更新和维护：如果不使用 Anaconda，那么更新 PyTorch 或其依赖库可能更加困难，因为我们需要手动管理每个组件的更新。

接下来，我将讲解如何搭建conda环境

在Ubuntu虚拟机自带的firefox浏览器中，进入 [Anaconda](#) 进行Anaconda的安装与下载



提供邮箱后，选择我们所需要的安装包



然后，在Ubuntu命令行输入 `sudo bash /root/Anaconda3-2023.03-1-Linux-x86_64.sh` 进行Anaconda的安装

输入之后，一直回车直到出现要求你输入yes或no的指令时，输入yes，不出意外的话，会出现这个界面

```
Anaconda3 will now be installed into this location:  
/root/anaconda3
```

- Press ENTER to confirm the location
- Press CTRL-C to abort the installation
- Or specify a different location below

```
[/root/anaconda3] >>> █
```

这个地方，我建议不要修改路径，否则可能对后续的操作带来麻烦，直接回车即可

下面这一步非常重要!!! 千万不要手快了按回车，这里我们应该输入yes，再按回车去初始化anaconda，要不然需要自己去配置路径，这样非常麻烦

```
Downloading and Extracting Packages
```

```
Preparing transaction: done
```

```
Executing transaction: /
```

```
Installed package of scikit-learn can be accelerated using scikit-learn-intelex.  
More details are available here: https://intel.github.io/scikit-learn-intelex
```

```
For example:
```

```
< $ conda install scikit-learn-intelex  
$ python -m sklearnx my_application.py
```

```
ne
```

```
installation finished.
```

```
Do you wish the installer to initialize Anaconda3
```

```
by running conda init? [yes|no]
```

```
[no] >>> █
```

接下来，静候Anaconda安装完成即可

接下来，我们执行 `conda create -n dl python=3.8` 创建python3.8且名为dl的环境之后，出现这样的界面，输入yes即可

```
## Package Plan ##
```

```
environment location: /root/anaconda3/envs/dl
```

```
added / updated specs:
```

```
- python=3.8
```

```
The following packages will be downloaded:
```

package	build	
ca-certificates-2023.05.30	h06a4308_0	120 KB
libffi-3.4.4	h6a678d5_0	142 KB
openssl-3.0.8	h7f8727e_0	5.2 MB
pip-23.1.2	py38h06a4308_0	2.6 MB
python-3.8.16	h955ad1f_4	23.9 MB
setuptools-67.8.0	py38h06a4308_0	1.0 MB
sqlite-3.41.2	h5eeel8b_0	1.2 MB
wheel-0.38.4	py38h06a4308_0	63 KB
xz-5.4.2	h5eeel8b_0	642 KB
Total:		34.8 MB

```
The following NEW packages will be INSTALLED:
```

_libgcc_mutex	pkgs/main/linux-64::_libgcc_mutex-0.1-main
_openmp_mutex	pkgs/main/linux-64::_openmp_mutex-5.1-1_gnu
ca-certificates	pkgs/main/linux-64::ca-certificates-2023.05.30-h06a4308_0
ld_impl_linux-64	pkgs/main/linux-64::ld_impl_linux-64-2.38-h1181459_1
libffi	pkgs/main/linux-64::libffi-3.4.4-h6a678d5_0
libgcc-ng	pkgs/main/linux-64::libgcc-ng-11.2.0-h1234567_1
libgomp	pkgs/main/linux-64::libgomp-11.2.0-h1234567_1
libstdcxx-ng	pkgs/main/linux-64::libstdcxx-ng-11.2.0-h1234567_1
ncurses	pkgs/main/linux-64::ncurses-6.4-h6a678d5_0
openssl	pkgs/main/linux-64::openssl-3.0.8-h7f8727e_0
pip	pkgs/main/linux-64::pip-23.1.2-py38h06a4308_0
python	pkgs/main/linux-64::python-3.8.16-h955ad1f_4
readline	pkgs/main/linux-64::readline-8.2-h5eeel8b_0
setuptools	pkgs/main/linux-64::setuptools-67.8.0-py38h06a4308_0
sqlite	pkgs/main/linux-64::sqlite-3.41.2-h5eeel8b_0
tk	pkgs/main/linux-64::tk-8.6.12-h1ccaba5_0
wheel	pkgs/main/linux-64::wheel-0.38.4-py38h06a4308_0
xz	pkgs/main/linux-64::xz-5.4.2-h5eeel8b_0
zlib	pkgs/main/linux-64::zlib-1.2.13-h5eeel8b_0

这样，我们的conda环境就创建好了

2.PyTorch的安装

重新进入 [PyTorch](#)，找到我们之前决定的Pytorch版本，复制网页提供的命令并安装

```
#CUDA 11.8
```

```
conda install pytorch==2.4.0 torchvision==0.19.0 torchaudio==2.4.0 pytorch-  
cuda=11.8 -c pytorch -c nvidia
```

跟着安装提示进行即可，此时我们的PyTorch就安装完成了，接下来，我们检验一下PyTorch是否安装成功

通过 `python` 命令，进入python，导入torch库 `import torch` 并回车，然后执行 `print(torch.cuda.is_available())` 并回车

```
root@localhost: ~  
olddove@localhost:~$ python  
Command 'python' not found, did you mean:  
  
  command 'python3' from deb python3  
  command 'python' from deb python-is-python3  
  
olddove@localhost:~$ sudo -i  
[sudo] password for olddove:  
(base) root@localhost:~# conda activate dl  
(dl) root@localhost:~# python  
Python 3.8.20 (default, Oct 3 2024, 15:24:27)  
[GCC 11.2.0] :: Anaconda, Inc. on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> import torch  
>>> print(torch.cuda.is_available())  
True  
>>> |
```

输出True，说明我们成功安装了PyTorch环境，同时也说明你的pytorch是可以正常调用GPU去进行计算的

以上，我们就完成了Anaconda与PyTorch的安装，接下来，我们将实现数据集准备及训练模型搭建

II.数据集准备及训练模型搭建

①.CIFAR-10 数据集的下载与处理

在完成这一步之前，我们应该下载一个能够运行python程序的编辑器，以方便代码的书写，保存和运行，这里我选择了vscode。使用vscode可以选择在Windows主系统上进行远程连接，或者直接在虚拟机内下载，这里我为了节约时间，选择了后者

！！！！注意：在使用VSCode时，检查一下我们的Python环境，我们选择的Python环境一定是我们刚刚创建好的dl环境内的，因为只有在这个环境中我们安装了PyTorch，否则的话torch无法正常使用，我之前就中了这个坑

CIFAR-10数据集的下载和处理我找到的有两种方式，一种是使用非官方的，他人提供的，一种是官方的，在之前我并没有找到官方的方式，故使用了第一种，但是遇到了严重的test set size = 0 问题导致运行精度无法测算，后来找到了官方的处理方法才得以解决，稍后我会在第五个大板块中讲解。下面是官方加载数据的指令

```

import torchvision
import torchvision.transforms as transforms

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=4,
shuffle=False, num_workers=2)

```

这样，我们就完成了CIFAR-10数据集的下载与处理，并将其转换为 PyTorch DataLoader 格式，确保数据集可以高效加载到 GPU 进行训练。

②.实现深度学习模型

在考核中，试题要求我们使用 PyTorch 实现一个基础的卷积神经网络（CNN），模型结构可以包括卷积层、池化层和全连接层，不调用现成的模型库（如 torchvision.models）。因此，我使用了以下指令来实现深度学习模型的构建

```

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

class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        # 卷积层 1
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3,
stride=1, padding=1)
        # 池化层 1
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        # 卷积层 2
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64,

```

```

kernel_size=3, stride=1, padding=1)
    # 全连接层 1
    self.fc1 = nn.Linear(in_features=64 * 8 * 8, out_features=128)
    # 全连接层 2
    self.fc2 = nn.Linear(in_features=128, out_features=10)

    def forward(self, x):
        # 第一个卷积层 + 激活函数 + 池化层
        x = self.pool(F.relu(self.conv1(x)))
        # 第二个卷积层 + 激活函数 + 池化层
        x = self.pool(F.relu(self.conv2(x)))
        # 展平特征图
        x = x.view(-1, 64 * 8 * 8)
        # 第一个全连接层 + 激活函数
        x = F.relu(self.fc1(x))
        # 第二个全连接层
        x = self.fc2(x)
        return x

```

```

# 实例化模型并移动到 GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = SimpleCNN().to(device)

```

接下来，使用以下指令，确保模型训练在GPU上进行

```

# 实例化模型并移动到 GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = SimpleCNN().to(device)

```

接着，我们来设置损失函数和优化器，对于分类问题，交叉熵损失函数是一个常见选择。这里我们选择Adam 优化器

```

import torch.optim as optim

# 损失函数
criterion = nn.CrossEntropyLoss()
# 优化器
optimizer = optim.Adam(model.parameters(), lr=0.001)

```

然后，我们开始编写训练循环来训练模型。在每个 epoch 中，遍历训练数据集，计算损失，执行反向传播，并更新模型的权重。

```

num_epochs = 10 # 根据需要调整 epoch 数量

for epoch in range(num_epochs):
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data[0].to(device), data[1].to(device)

        optimizer.zero_grad() # 清零梯度

        outputs = model(inputs) # 前向传播
        loss = criterion(outputs, labels)
        loss.backward() # 反向传播
        optimizer.step() # 更新权重

        running_loss += loss.item()
        if i % 2000 == 1999: # 每 2000 个批次打印一次
            print(f'[{epoch + 1}, {i + 1}] loss: {running_loss / 2000:.3f}')
            running_loss = 0.0

print('Finished Training')

```

最终，我们来计算运行的精度与loss，来量化我们模型的性能

```

correct = 0
total = 0
with torch.no_grad(): # 在评估模式下，不计算梯度
    for data in testloader:
        images, labels = data[0].to(device), data[1].to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'Accuracy of the network on the 10000 test images: {100 * correct /
total:.2f}%')

```

最后，再加上时间计算，内存占用情况等代码，再将这些代码进行整合，我们得到了一个初始的，能够计算运算时间，内存占用情况，loss，精度，并能够调整batch size，worker和学习率的训练模型

```

import os
import torch
import torchvision

```



```

import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import time
import psutil

# 数据集放置路径
data_save_pth = "./data"

# 数据转换
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

# 使用官方方式加载数据集
trainset = torchvision.datasets.CIFAR10(root=data_save_pth, train=True,
download=True, transform=transform)
trainloader = DataLoader(trainset, batch_size=4, shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root=data_save_pth, train=False,
download=True, transform=transform)
testloader = DataLoader(testset, batch_size=4, shuffle=False, num_workers=2)

# 检查数据集大小
print(f'Training set size: {len(trainset)}')
print(f'Test set size: {len(testset)}')

# 定义CNN模型
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)

```



```

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

# 初始化网络和优化器
net = Net()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
criterion = nn.CrossEntropyLoss()

# 将模型移动到GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
net.to(device)
criterion.to(device)

start_time = time.time()

# 训练循环
for epoch in range(10): # 这里使用2个epoch作为示例
    running_loss = 0.0
    for i, (inputs, labels) in enumerate(trainloader, 0):
        inputs, labels = inputs.to(device), labels.to(device)

        optimizer.zero_grad()

        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        if i % 2000 == 1999: # 每2000个小批量打印一次
            print(f'[{epoch + 1}, {i + 1}] loss: {running_loss / 2000:.3f}')
            running_loss = 0.0
            print(f'Memory Usage:
{psutil.Process(os.getpid()).memory_info().rss / (1024 * 1024):.2f} MB') #
打印内存使用情况

print('Finished Training')

end_time = time.time()

# 计算运行时间
elapsed_time = end_time - start_time

```

```

print(f'Training took {elapsed_time:.2f} seconds')

# 测试模型性能
# 测试模型性能
correct = 0
total = 0
with torch.no_grad():
    for images, labels in testloader:
        images = images.to(device)
        labels = labels.to(device) # 确保标签也在 GPU 上
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

if total > 0:
    print(f'Accuracy of the network on the test images: {100 * correct /
total:.2f}%')
else:
    print('No test data found.')

```

在这里，第一个卷积层的输出通道数为6，第二个卷积层的输出通道数为16。第一个全连接层有120个输出单元，第二个全连接层有84个输出单元，最后一个全连接层有10个输出单元。这其实是相对来说没那么优质的模型，稍后我会在优化方式这一板块中详细讲解

以10次循环为例，我们可以得到以下结果

```

[1, 2000] loss: 2.214
Memory Usage: 984.87 MB
[1, 4000] loss: 1.896
Memory Usage: 984.88 MB
[1, 6000] loss: 1.683
Memory Usage: 984.93 MB
[1, 8000] loss: 1.584
Memory Usage: 984.93 MB
[1, 10000] loss: 1.530
Memory Usage: 984.93 MB
[1, 12000] loss: 1.460
Memory Usage: 984.93 MB
[2, 2000] loss: 1.412
Memory Usage: 985.41 MB
..... (省略中间部分)
Memory Usage: 987.01 MB
[10, 8000] loss: 0.873

```

```
Memory Usage: 987.01 MB
[10, 10000] loss: 0.876
Memory Usage: 987.01 MB
[10, 12000] loss: 0.916
Memory Usage: 987.01 MB
Finished Training
Training took 403.63 seconds
Accuracy of the network on the test images: 61.46%
```

由此可知，我们已经成功实现了模型的搭建与训练，总共运行时间为403.43s，loss在逐渐减少，精度为61.16%，内存平均占用986MB，之后我又运行了几次，算出平均的时间，精度，内存占用分别为404.15s，994.33MB，61.02%，接下来，我将尝试不同的优化策略，寻找最高效的优化方法

III.模型优化与加速

①.CPU & GPU

为了对比模型在CPU和GPU上的训练情况，我写了下面这个程序，以对比两者在时间，精度，以及内存占用上的对比。

```
import os
import torch
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import time
import psutil

# 数据集放置路径
data_save_path = "./data"

# 数据转换
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
```

```

# 使用官方方式加载数据集
trainset = torchvision.datasets.CIFAR10(root=data_save_pth, train=True,
download=True, transform=transform)
trainloader = DataLoader(trainset, batch_size=4, shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root=data_save_pth, train=False,
download=True, transform=transform)
testloader = DataLoader(testset, batch_size=4, shuffle=False, num_workers=2)

# 检查数据集大小
print(f'Training set size: {len(trainset)}')
print(f'Test set size: {len(testset)}')

# 定义CNN模型
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

def train_and_evaluate(device):
    net = Net().to(device)
    optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
    criterion = nn.CrossEntropyLoss().to(device)

    start_time = time.time()

    for epoch in range(30): # 使用10个epoch作为示例
        running_loss = 0.0
        for i, (inputs, labels) in enumerate(trainloader, 0):
            inputs, labels = inputs.to(device), labels.to(device)

```

```

optimizer.zero_grad()

outputs = net(inputs)
loss = criterion(outputs, labels)
loss.backward()
optimizer.step()

running_loss += loss.item()
if i % 2000 == 1999: # 每2000个小批量打印一次
    print(f' [{epoch + 1}, {i + 1}] loss: {running_loss /
2000:.3f}')

    running_loss = 0.0
    print(f'Memory Usage:
{psutil.Process(os.getpid()).memory_info().rss / (1024 * 1024):.2f} MB') #
打印内存使用情况

print('Finished Training')

end_time = time.time()
elapsed_time = end_time - start_time
print(f'Training took {elapsed_time:.2f} seconds on {device}')

# 测试模型性能
correct = 0
total = 0
with torch.no_grad():
    for images, labels in testloader:
        images = images.to(device)
        labels = labels.to(device)
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

if total > 0:
    print(f'Accuracy of the network on the test images: {100 * correct /
total:.2f}%')
else:
    print('No test data found.')

return elapsed_time

# 训练和评估在CPU上

```

```

cpu_time = train_and_evaluate(torch.device("cpu"))

# 训练和评估在GPU上
if torch.cuda.is_available():
    gpu_time = train_and_evaluate(torch.device("cuda"))
    print(f'Speedup on GPU compared to CPU: {cpu_time / gpu_time:.2f}x')
else:
    print("CUDA is not available. Cannot compare with GPU.")

```

以2次为例，我们得到以下结果

Memory Usage: 659.07 MB

Finished Training

Training took 57.07 seconds on cpu

Accuracy of the network on the test images: 54.07%

Memory Usage: 997.85 MB

Finished Training

Training took 84.73 seconds on cuda

Accuracy of the network on the test images: 53.79%

Speedup on GPU compared to CPU: 0.67x

可以看到在循环数为2的情况下，CPU的各方面表现得都比GPU好

那么10次呢？

在10次的情况下，CPU上运行模型的评价时间，精度，占用内存分别为293.87s，62.07%，662.47MB，仍然是比GPU有锈

如果我运算次数扩大到30次，或者更多呢？

在30次的情况下，我们可以发现GPU的精度开始超过CPU，那我们可以合理推测，在数据量逐渐增大的情况下，GPU的精度会逐渐优于CPU

关于GPU运算时间远超CPU，查询资料后，我有以下两个推论

- 数据传输时间：将数据从CPU传输到GPU需要时间，如果处理的数据量很小，这个传输时间可能会抵消GPU加速的优势。对于小规模的数据或模型，CPU可能会更快。
- 模型复杂度：如果模型过于简单，CPU可能迅速完成计算，而GPU还没来得及发挥其并行处理的优势。在这种情况下，可以尝试使用更复杂的模型或加深加宽现有的神经网络模型。

可以推测，CIFAR-10是一个并不算复杂的数据集，此时CPU的优势开始凸显，但随着循环次数的增加，数据量逐渐增大，GPU的优势就会逐渐显露，从而导致我们上面展现的结果

②.不同的优化方式

1.batch_size的调整

调整 Batch Size 是深度学习中优化模型性能的重要策略之一。Batch Size直接影响模型训练的稳定性、收敛速度和最终的泛化能力。这是我目前发现的最高效的优化方法

下面我将展示这种优化后的代码

```
import os
import torch
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import time
import psutil

# 数据集放置路径
data_save_pth = "./data"

# 数据转换
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

# 使用官方方式加载数据集
trainset = torchvision.datasets.CIFAR10(root=data_save_pth, train=True,
download=True, transform=transform)
trainloader = DataLoader(trainset, batch_size=8, shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root=data_save_pth, train=False,
download=True, transform=transform)
testloader = DataLoader(testset, batch_size=8, shuffle=False, num_workers=2)

#..... (省略后面重复部分
```

仅仅是增长了一倍的batch_size值，我们的运行速度就提高了整整一倍，并获得了4-5个百分点的精度提升。

这种方法的唯一缺点就是可能由于数据量的问题导致if i % 2000 == 1999: 这个条件错误，导致无法查看每2000个批量的loss值。

另外，我们最好不要将batch值设的过大，虽然速度会提升很多，但是精度也会随之迅速下降。

2.混合精度训练

相比其他的方式，混合精度似乎提升并不高，但是将这种优化方式和其他方式结合，往往能够得到 $1+1>2$ 的效果，下面我将展示只采用混合精度的代码。

```
#..... (省略重复部分)

# 定义CNN模型
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

# 初始化网络和优化器
net = Net()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
criterion = nn.CrossEntropyLoss()

# 将模型移动到GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
net.to(device)
criterion.to(device)

start_time = time.time()

# 导入自动混合精度的库
from torch.cuda.amp import GradScaler, autocast

# 初始化梯度缩放器
scaler = GradScaler()

.....

)
```


通过计算，我们可以得到通过混合精度运算优化后的平均时长，精度，和内存占用分别为526.15s，62.47%，1062.43MB，可以看到精度方面有着不小的提升。

3.调整workers

在DataLoader中设置num_workers参数，使用多线程或多进程同时加载数据，可以有效减少数据加载瓶颈

注：经过测试，发现调整workers和调整batch_size结合才是最高效的方法，下面是代码

```
import os
import torch
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import time
import psutil

# 数据集放置路径
data_save_pth = "./data"

# 数据转换
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

# 使用官方方式加载数据集
trainset = torchvision.datasets.CIFAR10(root=data_save_pth, train=True,
download=True, transform=transform)
trainloader = DataLoader(trainset, batch_size=16, shuffle=True,
num_workers=4)

testset = torchvision.datasets.CIFAR10(root=data_save_pth, train=False,
download=True, transform=transform)
testloader = DataLoader(testset, batch_size=16, shuffle=False, num_workers=4)

.....(省略重复部分
```

经过多次测试，发现将batch_size设为16，workers设为4才是最优解
数据大概为下
Memory Usage: 988.84 MB
Finished Training
Training took 111.21 seconds
Accuracy of the network on the test images: 64.09%

4.使用cudnn

本来一开始下载cuda后我就准备下载cudnn来着，但是由于找错了下载地址以及注册账号耽误了一段时间，后面才下载成功，接下来，我将讲解cudnn的下载以及如何使用它来优化我们的模型

[cudnn](#)

这个是可以直接打开找到下载指令的网址，无需登录或其他操作，只需要选择好我们系统的配置后将网址提供给我们的下载指令复制在虚拟机中运行即可

以我的系统配置为例，依次输入以下指令。

```
wget
https://developer.download.nvidia.com/compute/cudnn/9.5.0/local_installers/cudnn-
local-repo-ubuntu2204-9.5.0_1.0-1_amd64.deb

sudo dpkg -i cudnn-local-repo-ubuntu2204-9.5.0_1.0-1_amd64.deb

sudo cp /var/cudnn-local-repo-ubuntu2204-9.5.0/cudnn-*keyring.gpg
/usr/share/keyrings/

sudo apt-get update

sudo apt-get -y install cudnn
```

因为我们下载的CUDA版本为11.8，故输入适用于CUDA11的安装指令

```
sudo apt-get -y install cudnn-cuda-11
```

这样，我们的cudnn就下载好了，接下来，我将讲解如何通过cudnn实现模型优化
其实这一步十分简单，只需要在我们的模型训练循环开始前输入

```
torch.backends.cudnn.benchmark = True
```

即可启用cudnn加速了！

cudnn其实对于模型的精度并没有多少提升，但是它极大的提高了我们模型运行的速度，在调整batch和workers发挥的作用尤其明显。例如，在batch为32，workers为4的情况下，模型运行的平均时间由66.44s降到了51.26s，将近77%的时间缩减。而在batch和workers都为默认值的初始状态下，时间仅仅从404.15s降到了393.50s，提速效果也就没那么明显了，总的来说，cudnn可以显著提高batch和workers数量提高时模型的运行速度

5.修改输入通道和输出单元

在设计卷积神经网络（CNN）时，决定卷积层的通道数（即过滤器或内核的数量）是一个重要的架构决策，在之前的代码中，我们的第一个卷积层的输出通道数为6，第二个卷积层的输出通道数为16。第一个全连接层有120个输出单元，第二个全连接层有84个输出单元，最后一个全连接层有10个输出单元。在这里，我们将第一个卷积层的输出通道数改为32，第二个卷积层的输出通道数改为64，并且调整全连接层的输入特征数以匹配经过两次池化后的特征图大小。

```
# 定义CNN模型
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # 第一个卷积层，输入通道3（RGB图像），输出通道32，卷积核大小3x3
        self.conv1 = nn.Conv2d(3, 32, 3, stride=1, padding=1)
        # 池化层，窗口大小2x2，步长2
        self.pool = nn.MaxPool2d(2, 2)
        # 第二个卷积层，输入通道32，输出通道64，卷积核大小3x3
        self.conv2 = nn.Conv2d(32, 64, 3, stride=1, padding=1)
        # 第一个全连接层，输入特征数为64*8*8（因为经过两次池化后，特征图大小
        # 减半两次），输出特征数256
        self.fc1 = nn.Linear(64 * 8 * 8, 256)
        # 第二个全连接层，输入特征数256，输出特征数10（CIFAR-10数据集的类别
        # 数）
        self.fc2 = nn.Linear(256, 10)

    def forward(self, x):
        # 应用第一个卷积层和激活函数ReLU
        x = self.pool(F.relu(self.conv1(x)))
        # 应用第二个卷积层和激活函数ReLU
        x = self.pool(F.relu(self.conv2(x)))
        # 展平特征图，为全连接层准备
        x = x.view(-1, 64 * 8 * 8)
        # 应用第一个全连接层和激活函数ReLU
        x = F.relu(self.fc1(x))
        # 应用第二个全连接层
        x = self.fc2(x)
        return x
```

我们运行一次试一试

```
Memory Usage: 1009.16 MB
Finished Training
Training took 359.94 seconds
Accuracy of the network on the test images: 72.15%
```

可以明显的发现，我们的精度有了极大的提升。
那么如果是64，128的组合呢？

```
Memory Usage: 1077.75 MB
Finished Training
Training took 656.42 seconds
Accuracy of the network on the test images: 75.01%
```

可以看到，精度有了进一步提升，但是速度也随之变慢了许多

顺带一提，如果是在cpu上运行，精度变化幅度不大的同时，时间会来到412.14s，可见，在输入输出通道达到合适的数量的情况下，gpu的优势会逐渐凸显。输入通道数量正是影响模型复杂度的要素之一，这与我们之前的模型复杂度较低导致CPU性能高于GPU的推论是一致的！推论与实践相符的感觉真好啊。(误

另外，在输入通道提高后，由于初始精度基数的提高，我们可以进一步提高batch和workers的值，在精度变化在接受范围内的情况下，极大程度提高运行速度，如batch为62，workers为15时，精度为63.24%，运行时间降到29.66，虽然精度有所下降，但是速度得到巨大提升，适用于精度要求不高但是速度要求高的情况。

6.其他负优化行为

注：由于是负优化，下列测试都只进行了寥寥几次，代码不进行完整的展示

a.批量一体化

使用批量一体化后，精度反而下降，时间变长

```
Memory Usage: 1026.61 MB
Finished Training
Training took 489.16 seconds
Accuracy of the network on the test images: 51.08%
下面是实现批量一体化的代码部分
```

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.bn1 = nn.BatchNorm2d(6) # 添加批量归一化
```

```

self.pool = nn.MaxPool2d(2, 2)
self.conv2 = nn.Conv2d(6, 16, 5)
self.bn2 = nn.BatchNorm2d(16) # 添加批量归一化
self.fc1 = nn.Linear(16 * 5 * 5, 120)
self.bn3 = nn.BatchNorm1d(120) # 添加批量归一化
self.fc2 = nn.Linear(120, 84)
self.bn4 = nn.BatchNorm1d(84) # 添加批量归一化
self.fc3 = nn.Linear(84, 10)

```

ps:即使是修改输入通道后，这个方法仍然是负优化

b.调整学习率

同样的，使用调整学习率后，精度下降

```

[10, 12000] loss: 1.039
Memory Usage: 1000.86 MB
Finished Training
Training took 409.62 seconds
Accuracy of the network on the test images: 57.38%

```

ps: 在修改输入通道之后，这种方法虽然没有再负优化了，但是精度也没有进一步的提升

我会在第五个大板块讲解这些本应优化的策略负优化的原因

IV.不同优化方式性能可视化

为了使数据可视化，我先将测算出的数据放在一个csv文件当中

```

Optimization Technique, Training Time (seconds), Accuracy (%), Memory Usage (MB), Epochs
cpu, 293.87, 62.07, 662.47, 10
g-initial, 404.15, 61.02, 994.33, 10
g-torch.cuda.amp, 526.15, 62.47, 1062.43, 10
g-b32-w4, 66.44, 60.39, 977.11, 10
g-b16-w4, 113.21, 64.91, 988.84, 10
g-b16-w8, 109.48, 62.77, 994.29, 10
g-initial-cudnn, 393.50, 61.29, 1039.67, 10
g-b32-w4-cudnn, 51.26, 63.70, 1003.13, 10
g-b16-w4-cudnn, 104.69, 62.45, 995.61, 10
g-b16-w8-cudnn, 96.03, 62.52, 1070.30, 10

```

```
in-cpu, 412, 72.56, 779.27, 10
in-g-initial, 354.06, 71.90, 1007.84, 10
in-g-b32-w4-, 55.63, 69.70, 1038.15, 10
in-g-b16-w4, 91.81, 72.72, 1044.54, 10
in-g-b16-w8, 94.36, 71.84, 1047.11, 10
in-g-b64-w15, 29.66, 63.24, 1056.15, 10
in-g-b128-w15, 19.68, 56.12, 1038.43, 10
```

再通过使用pandas库读取csv文件中的内容，然后再使用 matplotlib.pyplot 函数制作柱状图

以下是生成柱状图的代码

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# 设置Seaborn的样式
sns.set(style="whitegrid")

# 读取CSV文件
df = pd.read_csv('training_data.csv')

# 设置图表大小
plt.figure(figsize=(25, 15)) # 调整图表大小以适应三个子图

# 定义颜色列表，确保颜色数量与数据类别数量匹配
colors =
['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b', '#ff7f0e', '#d62728',

# 绘制Training Time柱状图
plt.subplot(3, 1, 1)
sns.barplot(x='Optimization Technique', y='Training Time (seconds)', data=df,
palette=colors)
plt.title('Training Time by Optimization Technique', fontsize=18)
plt.xlabel('Optimization Technique', fontsize=16)
plt.ylabel('Training Time (seconds)', fontsize=16)
plt.grid(True) # 添加网格线

# 在柱状图上添加数据标签
for p in plt.gca().patches:
    plt.text(p.get_x() + p.get_width() / 2., p.get_height(),
             f'{p.get_height():.2f}',
```

```

        ha='center', va='bottom', fontsize=12, color='black')

# 绘制Accuracy柱状图
plt.subplot(3, 1, 2)
sns.barplot(x='Optimization Technique', y='Accuracy (%)', data=df,
            palette=colors)
plt.title('Accuracy by Optimization Technique', fontsize=18)
plt.xlabel('Optimization Technique', fontsize=16)
plt.ylabel('Accuracy (%)', fontsize=16)
plt.grid(True) # 添加网格线

# 在柱状图上添加数据标签
for p in plt.gca().patches:
    plt.text(p.get_x() + p.get_width() / 2., p.get_height(),
             f'{p.get_height():.2f}%',
             ha='center', va='bottom', fontsize=12, color='black')

# 绘制Memory Usage柱状图
plt.subplot(3, 1, 3)
sns.barplot(x='Optimization Technique', y='Memory Usage (MB)', data=df,
            palette=colors)
plt.title('Memory Usage by Optimization Technique', fontsize=18)
plt.xlabel('Optimization Technique', fontsize=16)
plt.ylabel('Memory Usage (MB)', fontsize=16)
plt.grid(True) # 添加网格线

# 在柱状图上添加数据标签
for p in plt.gca().patches:
    plt.text(p.get_x() + p.get_width() / 2., p.get_height(),
             f'{p.get_height():.2f} MB',
             ha='center', va='bottom', fontsize=12, color='black')

# 调整子图间距
plt.tight_layout()

# 保存图表, 设置透明度
plt.savefig('optimization_techniques_comparison.png', dpi=300,
            transparent=True)

# 显示图表
plt.show()

```

这是最终呈现的图片

注：这里g代表gpu，w代表worker的数量，b代表batch的数量，cudnn表示使用了cudnn，in表示修改了输入通道的值



V.实验过程中遇到的问题及解决方案

①.虚拟机平台选择

正如我之前所说，与WSL不同，VMware等传统虚拟机软件创建的是一个完全隔离的虚拟环境。使用的是虚拟的硬件，而VMvare等平台做到直通功能又十分艰难，我第一次搭建Anaconda环境时，就因此卡在了安装nvidia驱动这一步，后来经过查询csdn等平台才选择了使用WSL2，轻而易举地做到了NVIDIA驱动与CUDATOOLKIT的安装

②.GPU驱动装置与CUDATOOLKIT下载方式

a.NVIDIA驱动的更新

之前我说过，我的虚拟机中NVIDIA驱动已经是较新的版本，故跳过更新，实际上，我们的NVIDIA驱动版本至少要高于525，才能实现我们后面的操作，接下来，我将讲解如何更新NVIDIA驱动

驱动安装包传送门：[NVIDIA驱动](#)（请在虚拟机中进行这一步）

手动驱动搜索




GeForce 



GeForce RTX 40 Series (Notebooks) 

GeForce RTX 4060 Laptop GPU 

Linux 64-bit 

Chinese (Simplified) 

查找

按照我们的系统，选择合适的版本，下载安装包即可。

请记住你的安装路径，下载完成后，我们在命令行中输入 `sudo bash 安装包路径`，接下来，跟着安装引导走，无视所有警告即可。

安装完成后，在命令行中输入 `nvidia-smi` 查询自己的驱动状态，发现驱动版本已更新即可

b.CUDATOOLKIT下载方式

在这一步中，我犯了一个错误，即在windows系统上安装好适用于Linux系统的安装包后，使用cp指令将安装包复制到Linux系统上，事实上，这是不安全的，这可能会导致安装包被放在一个即存在又不存在的Downloads文件中，即使下载成功，也会遇到其他奇怪的问题，从而导致PyTorch下载失败，比如说，conda文件内可能出现错误的版本号以及莫名其妙的波浪号，我vim很多相关文件也没找到所谓的~在哪里，换源之后，又出现了所谓的多余的“2.7”，后面重装了一遍系统，使用Ubuntu自带的firefox浏览器进行下载后，才解决了这个问题

```
(base) olddove@localhost:~$ conda install pytorch torchvision torchaudio
cudatoolkit=11.5 -c pytorch Solving environment: failed
InvalidVersionSpecError: Invalid version spec: =2.7
```

```
(base) olddove@localhost:~$ conda install pytorch torchvision torchaudio -c
pytorch Solving environment: failed CondaValueError: Malformed version string
'~': invalid character(s).
```

这两个是我因为复制安装包而遇到的两个错误

③.PyTorch安装须知

在之前，我提到过PyTorch安装有两种顺序，一种是决定pytorch的版本以此来选择cuda版本，第二种是先决定cuda的版本在以此来选择pytorch的版本，在第一次安装PyTorch时，我选择了第二种，这是不适合新手的。在第一次安装过程中，由于使用了第二种方法，我遇到了许多版本不兼容问题，究其根本是对CUDA 的版本通常由硬件和驱动程序决定这个细节，导致选择一个与 PyTorch 兼容的 CUDA 版本这个过程变得复杂。

下面是第一种方法的几个优势

- PyTorch 官方网站提供了一个非常直观的安装指引，用户可以通过选择操作系统、Python 版本、CUDA 版本来获取相应的安装命令。这种方法简化了选择过程，使得新手可以更容易地找到适合自己环境的安装命令。
- PyTorch 通常每隔一段时间就会发布新版本，通过先选择 PyTorch 版本，可以更容易地保持我们的环境是最新的。
- 如果先选择了CUDA的一个较老版本，可能会导致一些功能的缺失，从而导致CUDA和PyTorch版本之间的不兼容

在安装cuda过程中，我还遇到了11.8版本的安装包下载后变为10.1版本的问题，登录root账户后，得到了解决

切记，不要在windows中下载安装包后复制到虚拟机内

④.CIFAR-10的正确下载方法

在之前，我说过CIFAR-10有官方和非官方两种下载方法，第一次由于不知道官方的下载方法，我在csdn中寻找了非官方的下载方法，指令如下

```
import numpy as np
import pickle
import os
from torchvision import datasets
from imageio import imwrite

# 数据集放置路径
data_save_path = "./data"
```

```
train_pth = os.path.join(data_save_pth, "train")
test_pth = os.path.join(data_save_pth, "test")


# 创建必要的目录
def create_dir(path):
    if not os.path.exists(path):
        os.makedirs(path)

create_dir(train_pth)
create_dir(test_pth)


# 解压路径
data_dir = os.path.join(data_save_pth, "cifar-10-batches-py")


# 数据集下载
def download_data():
    datasets.CIFAR10(root=data_save_pth, train=True, download=True)


# 解压缩数据
def unpickle(file):
    with open(file, "rb") as fo:
        return pickle.load(fo, encoding="bytes")


# 保存图像
def save_images(data, output_dir, offset):
    for i in range(0, 10000):
        img = np.reshape(data[b'data'][i], (3, 32, 32)).transpose(1, 2, 0)
        label = str(data[b'labels'][i])
        label_dir = os.path.join(output_dir, label)
        create_dir(label_dir)

        img_name = f'{label}_{i + offset}.png'
        img_path = os.path.join(label_dir, img_name)
        imwrite(img_path, img)

if __name__ == '__main__':
    download_data()
    for j in range(1, 6):
```

```
path = os.path.join(data_dir, f"data_batch_{j}")
data = unpickle(path)
print(f"{path} is loading...")
save_images(data, train_pth, (j - 1) * 10000)
print(f"{path} loaded")

test_data_path = os.path.join(data_dir, "test_batch")
test_data = unpickle(test_data_path)
save_images(test_data, test_pth, 0)
print("test_batch loaded")
```

那么，这样做会出现什么问题呢？

在使用非官方方式下载的过程中，我发现无论采取什么措施，模型训练时输出的 Test set size 永远为0，从而导致无法计算出我们模型的精度，试题也就无法继续进行下去，同时，每次运行模型时，这种加载方式都会消耗大量的时间去下载数据集，造成巨大的时间浪费。

为什么会出现这种情况，我也不清楚，推测有可能是图像文件格式错误或是测试集目录路径错误。

⑤.负优化的原因

在之前进行模型优化的过程中，我提到过某些应为优化的策略造成了负优化，这是为什么呢，接下来就是我的解释

1.批量一体化

a.批量大小选择：批量归一化是在小批量数据上计算均值和方差的，如果批量大小太小，可能会导致归一化效果不佳，从而影响模型的精度。也许是我设置的批量太小，导致优化效果不明显甚至倒退。

b. 模型结构问题：如果模型结构不适合特定的数据集，即使使用了批量归一化，也可能导致精度下降。也许是我设定的模型结构不适合CIFAR-10数据集，导致精度下降

2.调整学习率

不同的优化算法对学习率的敏感度不同。一些优化算法如Adam具有自适应学习率调整的能力，而SGD等则需要手动调整学习率。如果更换了优化算法而没有重新调整学习率，可能会影响模型性能。在我的模型中，我使用了Adam，这可能是调整学习率负优化的原因

VI.参考资料&特别鸣谢

参考资料

[在Windows中安装wsl2和ubuntu22.04](#)

[2023年最新Ubuntu安装pytorch教程](#)

[用PyTorch搭建卷积神经网络](#)

[\[数据集\] CIFAR-10 数据集介绍](#)

特别鸣谢

[Kimi](#)

wwb的茶饼 [玩微博森sei](#)

10+ 的打压 [冰雪女王](#)

rainbow rainbow提醒了我输入通道相关内容 [太南昌彩彩](#)

炒蒜的大家

老鸽 2024.10.17
