NCUSCC 2024秋季考核——Pytorch试题实验报告

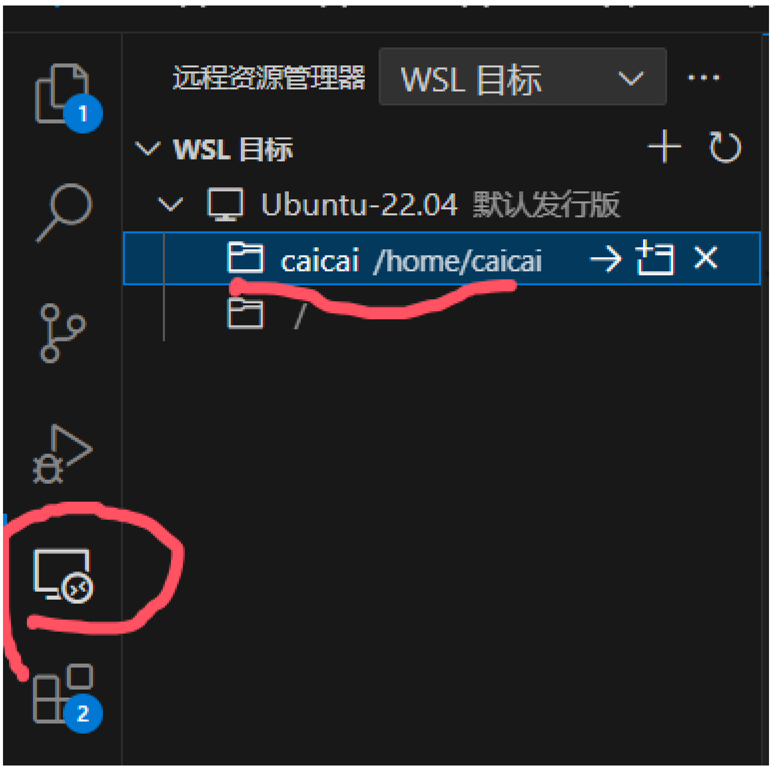
作者：杨许玮

开始时间：24/10/13

完成时间：24/10/21

实验前准备

- 实验中有要求：报告必须采用 LaTeX 或 Markdown 格式撰写。
  那我们不妨准备环境能够边实验边写报告。
  - 步骤：于官网中下载vs code，下载完成后我们可以安装Markdown All in One扩展来以Markdown 格式撰写。
  - 好处：通过该扩展，我们能够实现实时预览来检查我们的报告撰写方便修改。
- 选择我们实验的虚拟环境
  - 选择：通过WSL2来搭建虚拟环境
  - 原因：通过VM安装的系统的硬件都为虚拟化的硬件，我们难以直连我们的显卡。故使用WSL2来搭建虚拟环境（Ubuntu 22.04 LTS 操作系统），以此我们能够实现win10和Linux可以无缝切换，十分方便，省去很多不必要的麻烦。并且WSL2搭建虚拟环境也非常的便捷快速。
- 利用vscode连接WSL便于编辑python代码
  - 步骤：下载Remote Development扩展（该扩展为几个扩展打包在一起的，它包含了Remote-WSL，Remote-SSH，Remote-Container），此时左边为出现个屏幕的图案，我们可以以此连接WSL，如图：



  - 好处：使代码调试更便捷美观。

实验环境的搭建

以WSL2搭建虚拟环境

1. 启用适用于 Linux 的 Windows 子系统

需要先启用"适用于 Linux 的 Windows 子系统"可选功能，然后才能在 Windows 上安装 Linux 分发。
以管理员身份打开 PowerShell（"开始"菜单 >"PowerShell" >单击右键 >"以管理员身份运行"），然后输入以下命令：

```
dism.exe /online /enable-feature /eaturename:Microsoft-Windows-Subsystem-Linux
/all /norestart
```

2. 启用虚拟机功能

安装 WSL 2 之前，必须启用"虚拟机平台"可选功能。 计算机需要虚拟化功能才能使用此功能。
以管理员身份打开 PowerShell 并运行：

```
dism.exe /online /enable-feature /featurename:VirtualMachinePlatform /all
/norestart
```

重新启动计算机，以完成 WSL 安装并更新到 WSL 2。

3. 下载 Linux 内核更新包

首先我们需要确定自己计算机的类型， 如果不确定自己计算机的类型，请打开命令提示符或 PowerShell，并输入：

```
systeminfo | find "System Type"。
```

- 如果是x64计算机，则下载[适用于 x64 计算机的 WSL2 Linux 内核更新包]
- 如果是ARM64计算机，则下载[ARM64包]

然后运行上一步中下载的更新包。

4. 将 WSL 2 设置为默认版本

打开 PowerShell，然后在安装新的 Linux 发行版时运行以下命令，将 WSL 2 设置为默认版本：

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

5. 安装所选的 Linux 分发

根据要求我们需要安装Ubuntu 22.04 LTS，我们可以打开 Microsoft Store在分发版的页面中，选择"获取"。

首次启动新安装的 Linux 分发版时，将打开一个控制台窗口，系统会要求你等待一分钟或两分钟，以便文件解压缩并存储到电脑上。 未来的所有启动时间应不到一秒。
然后，需要为新的 Linux 分发版创建用户帐户和密码。

到此我们便完成了使用WSL2安装Ubuntu 22.04 LTS。

安装NVIDIA 驱动和 CUDA Toolkit

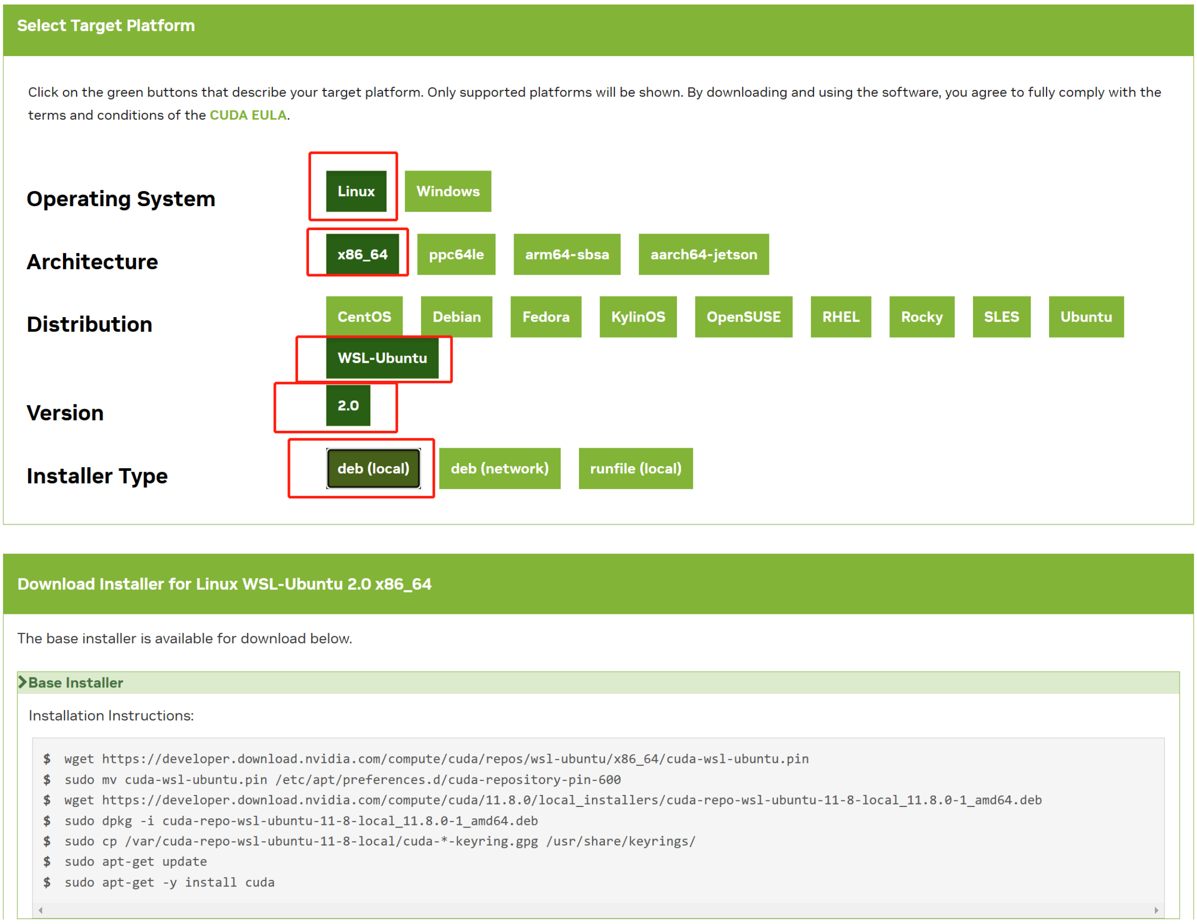
NVIDIA 驱动的安装

由于windows会自动为wsl安装nvidia驱动。
我们也可以通过在命令行中输入 nvidia-smi 查询自己的显卡驱动版本，如图：

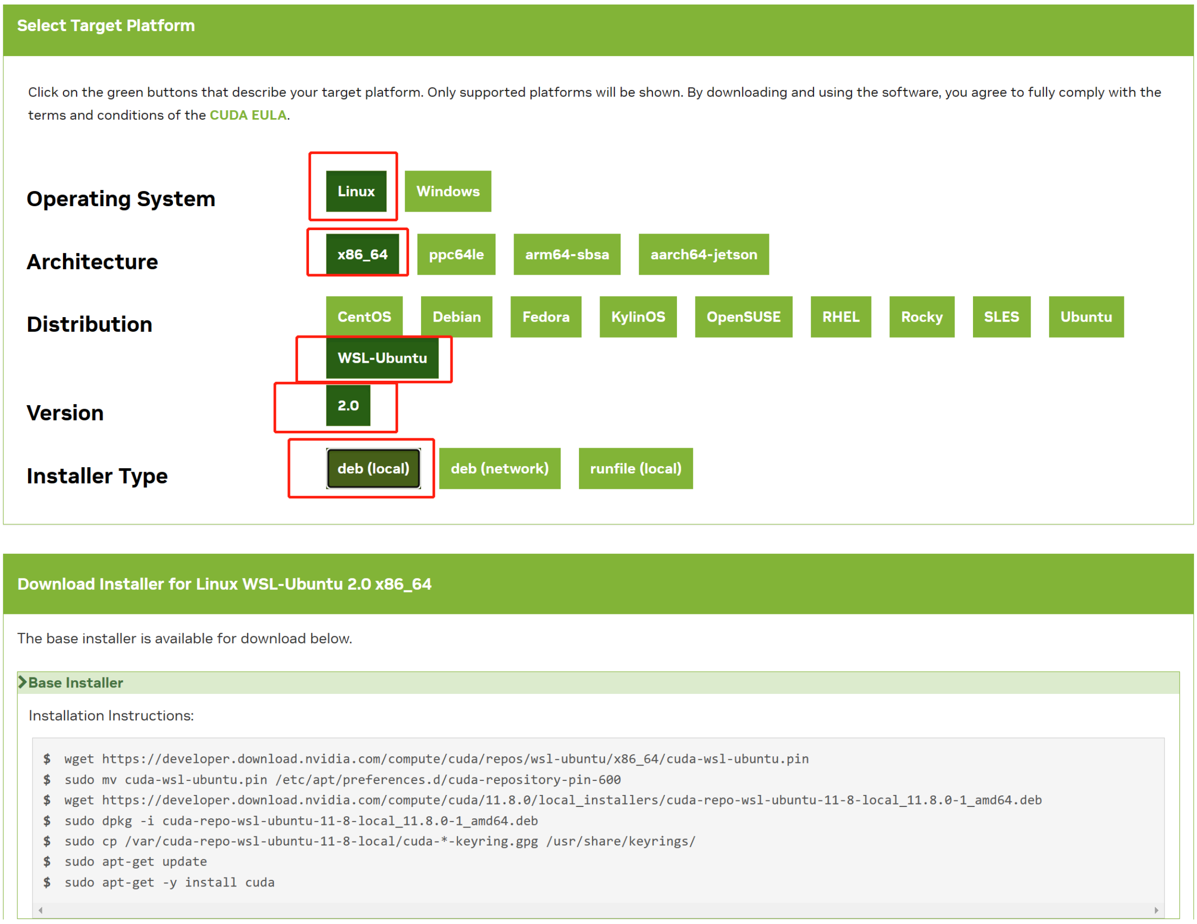
```
caicai@DESKTOP-26UFLTD:~$ nvidia-smi
Mon Oct 14 16:21:20 2024
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 530.56                 Driver Version: 532.10       CUDA Version: 12.1     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                  Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf            Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  NVIDIA GeForce RTX 4060 L...    On | 00000000:01:00.0  On |                  N/A |
| N/A   52C    P3              10W /  N/A |    628MiB /  8188MiB |     12%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|  No running processes found                                                           |
+---------------------------------------------------------------------------------------+
```

CUDA Toolkit的安装

可直接去官网下载所需版本，版本的选择上建议选较早的版本，因为较早的版本较为稳定bug较少，这里我选择安装11.8版本的。



一定要选择WSL版本的



选择好后将nvidia官方给的命令，一条条复制到wsl2的ternimal中即可。

安装结束之后执行nvcc -v，会提示没有nvcc可执行，这并不是因为我们cudatoolkit没安装好，而是因为环境变量还没配置好。
接下来我们将进行cuda环境变量配置：

1. 首先我们先在wsl2的ternimal中输入

```
sudo nano ~/.bashrc
```

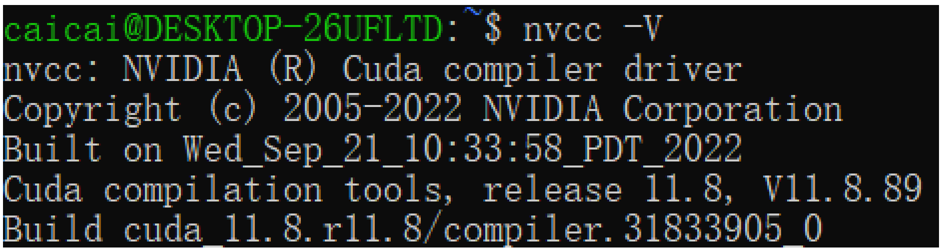
2. 将以下内容添加进文件最后

```
export PATH=/usr/local/cuda-11.8/bin${PATH:+:${PATH}}
export LD_LIBRARY_PATH=/usr/local/cuda-
11.8/lib64${LD_LIBRARY_PATH:+:${LD_LIBRARY_PATH}}
```

3. 保存退出后（Ctrl+x再按 y 最后按回车），更新一下环境变量，输入：

```
source ~/.bashrc
```

4. 这时候在执行 nvcc -V 就能够显示cuda版本了。如图：

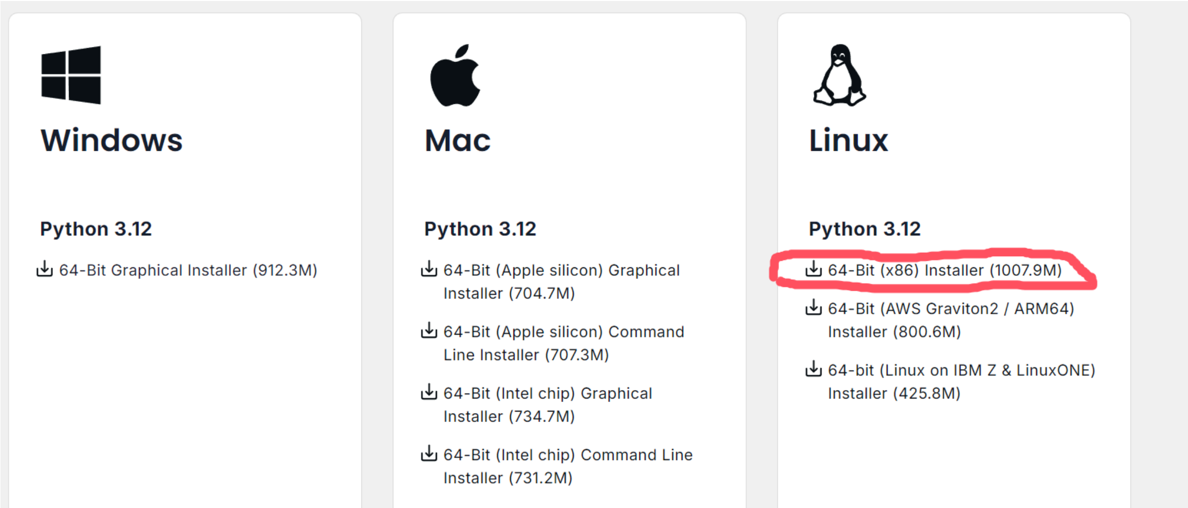


到此我们便完成了NVIDIA 驱动和 CUDA Toolkit的安装

wsl安装anaconda并配置环境以及PyTorch的安装

- 安装anaconda

anaconda3官方下载,然后提交自己的邮箱，进入下载页面。选择linux版本，鼠标放在其上方右键，复制链接。
如图：



回到Ubuntu的terminal，输入：

```
wget https://repo.anaconda.com/archive/Anaconda3-2024.06-1-Linux-x86_64.sh
```

我的版本是2024.06，请根据自己当时复制的链接进行修改。
运行以上代码，将会下载anaconda3到wsl ubuntu中。

之后按下sh A然后按Tab键，系统会自动补齐下面内容。
如：

```
sh Anaconda3-2024.06-1-Linux-x86_64.sh
```

接下来就是安装过程，只需要根据提示按回车或者输入yes即可。

- conda配置环境
  用conda创建虚拟环境

```
 conda create --name cu118py310 python=3.10  #--name 后面是创建环境的名字，按自己的习惯命名，python=XX，输入自己想用的版本号
```

例如我输入：

```
 conda create --name cc python=3.10
```

当左侧出现(base)则说明成功了，如：

当我们输入

```
 conda info --env
```

我们则可以看到所有python环境，前面有个'*'的代表当前环境

```
(base) caicai@DESKTOP-26UFLTD:~$ conda info --env
# conda environments:
#
base                  *  /home/caicai/anaconda3
cc                       /home/caicai/anaconda3/envs/cc
```

比如我这个现在仍是base环境
我们需要通过

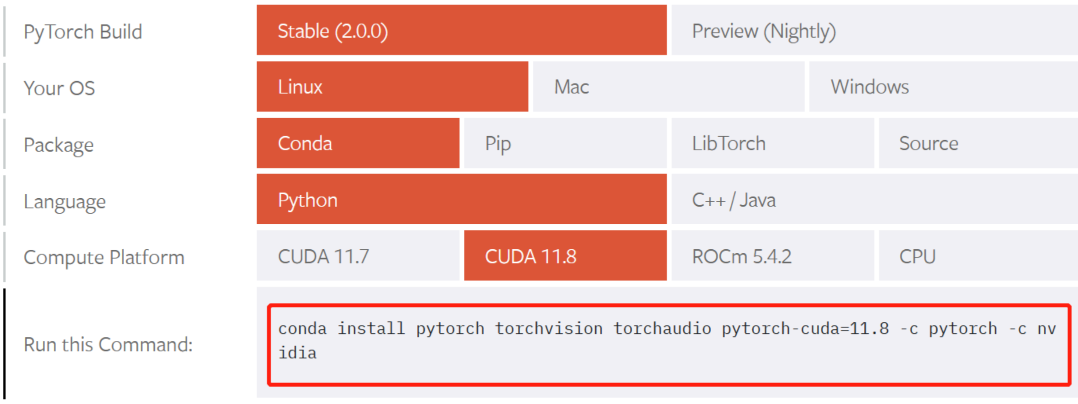
```
conda activate cc #（cc替换为我们创建的环境的名字） 激活刚刚创建的环境
```

然后我们可以明显看到我们切换到了cc环境，如图：

```
(cc) caicai@DESKTOP-26UFLTD:~$
```

- 配置pytorch

  前往[pytorch官网]，选择需要的环境（注意这里选择linux OS），复制conda命令,在terminal中粘贴，回车，安装环境：



  我们还需要检验一下PyTorch是否安装成功，我们可以通过torch.cuda.is_available() 验证，首先输入python进入python环境再通过import torch导入torch库，回车后执行print(torch.cuda.is_available())再回车，如若输出True则代表我们成功了
  以上操作如图所示：

```
(cc) caicai@DESKTOP-26UFLTD:~$ python
Python 3.10.15 (main, Oct  3 2024, 07:27:34) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>> print(torch.cuda.is_available())
True
>>> _
```

CIFAR-10 数据集的下载与处理

我们可以先与WSL的终端中使用：

```
code .
```

来进入vscode界面便于运行接下来的python代码

然后利用搜索引擎查找CIFAR-10 数据集的下载以及处理的代码，我们可以找到如下：

[参考]

```python
import torchvision
import torchvision.transforms as transforms

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

# 加载数据集
trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True,
transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4, shuffle=True)

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

通过以上便可实现该数据集的下载与处理，以及成功转变为 PyTorch DataLoader 格式，确保数据集可以高效加载到 GPU 进行训练。

实现深度学习模型

根据考核要求：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）。并在 GPU 上训练该模型，并验证其性能。

为了验证模型的性能，我们可以以以下标准进行测量：

1. 准确率（Accuracy）：
   这是衡量模型性能的最基本指标，表示模型正确预测的样本数量占总样本数量的比例。准确率越高，说明模型的预测能力越强。
2. 混淆矩阵（Confusion Matrix）：
   混淆矩阵是一个表格，用于显示模型预测的结果与实际标签之间的关系。它可以帮助我们了解模型在各个类别上的表现，特别是哪些类别容易被混淆。
3. 精确率（Precision）：精确率是指模型预测为正类别中实际为正类别的比例。对于每个类别，精确率告诉我们模型预测的准确性。
4. 召回率（Recall）：召回率是指在所有实际为正类别的样本中，模型正确预测为正类别的比例。召回率越高，说明模型漏掉的正样本越少。
5. F1分数（F1 Score）：F1分数是精确率和召回率的调和平均数，它综合考虑了精确率和召回率的平衡。F1分数越高，说明模型在精确率和召回率之间取得了较好的平衡。
6. 处理所有测试数据所需的时间（Training Time）：用来检测模型的运行快慢
   我们不妨通过以上标准来评估模型的性能，包括模型的整体效果、在不同类别上的表现、以及模型的稳定性和可靠性。通过这些指标，我们可以对模型的优缺点有一个清晰的认识，并据此进行模型的优化和调整。
7. 平均内存使用量（Average Memory Usage）：反应模型的内存效率

我们的Python环境中没有安装scikit-learn库。这个库提供了许多用于评估模型性能的工具。我们可以通过以下命令安装：

```
conda install scikit-learn
```

然后便可以继续运行，
以下直接给出注有详细注释的代码：

```
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
from sklearn.metrics import confusion_matrix, precision_recall_fscore_support

# 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=64, shuffle=True, num_workers=4)

testset = torchvision.datasets.CIFAR10(root=data_save_pth, train=False,
download=True, transform=transform)
testloader = DataLoader(testset, batch_size=64, shuffle=False, num_workers=4)

# 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__()
        # 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

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(10):  # 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
    all_labels = []
    all_preds = []
    all_probs = []
    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()
                all_labels.extend(labels.cpu().numpy())
                all_preds.extend(predicted.cpu().numpy())
                all_probs.extend(outputs.cpu().numpy())

    accuracy = 100 * correct / total
    cm = confusion_matrix(all_labels, all_preds)
    precision, recall, f1, _ = precision_recall_fscore_support(all_labels,
all_preds, average='weighted')

    return elapsed_time, accuracy, cm, precision, recall, f1,
psutil.Process(os.getpid()).memory_info().rss / (1024 * 1024)

# 训练和评估在CPU上
cpu_time, cpu_accuracy, cpu_cm, cpu_precision, cpu_recall, cpu_f1,
cpu_memory_usage = train_and_evaluate(torch.device("cpu"))
print(f'CPU - Training Time: {cpu_time:.2f} seconds, Accuracy:
{cpu_accuracy:.2f}%, Precision: {cpu_precision:.2f}, Recall: {cpu_recall:.2f}, F1
Score: {cpu_f1:.2f}, Memory Usage: {cpu_memory_usage:.2f} MB')
print(f'Confusion Matrix: \n{cpu_cm}')

# 训练和评估在GPU上
if torch.cuda.is_available():
    gpu_time, gpu_accuracy, gpu_cm, gpu_precision, gpu_recall, gpu_f1,
gpu_memory_usage = train_and_evaluate(torch.device("cuda"))
    print(f'GPU - Training Time: {gpu_time:.2f} seconds, Accuracy:
{gpu_accuracy:.2f}%, Precision: {gpu_precision:.2f}, Recall: {gpu_recall:.2f}, F1
Score: {gpu_f1:.2f}, Memory Usage: {gpu_memory_usage:.2f} MB')
    print(f'Confusion Matrix: \n{gpu_cm}')
else:
    print("CUDA is not available. Cannot compare with GPU.")
```

我们尝试将将模型分别在CPU和GPU加速下进行比较 ，并且设置了batch size=64，workers=4。
运行，我们可以得到我们想要的结果，如我所运行的一次结果：

```
Files already downloaded and verified
Files already downloaded and verified
Training set size: 50000
Test set size: 10000
Finished Training
Training took 83.57 seconds on cpu
CPU - Training Time: 83.57 seconds, Accuracy: 62.94%, Precision: 0.64, Recall: 0.63, F1 Score: 0.62, Memory Usage: 783.43 MB
Confusion Matrix:
[[780  12  34  10  14   4  14   3 110  19]
 [ 55 701  12   4   7   2  10   3  95 111]
 [117   6 430  47 203  54  62  34  34  13]
 [ 52   6  81 377 164 108 101  40  50  21]
 [ 48   2  64  42 709  16  47  44  26   2]
 [ 31   3  92 160 107 456  45  67  30   9]
 [ 23   4  36  44 132  11 713   8  20   9]
 [ 44   0  28  39 134  45  17 658   8  27]
 [ 93  24  10   7   7   3   6   2 831  17]
 [ 76 113  11  12  15   4  11  19 100 639]]
Finished Training
Training took 21.83 seconds on cuda
GPU - Training Time: 21.83 seconds, Accuracy: 63.35%, Precision: 0.64, Recall: 0.63, F1 Score: 0.63, Memory Usage: 4660.57 MB
Confusion Matrix:
[[805  17  27  10  14   7   8   6  50  56]
 [ 42 703  10   7   3   3   5   2  32 193]
 [116  11 446  62 172  70  41  37  23  22]
 [ 51   9  61 386 119 200  65  34  30  45]
 [ 55   2  71  40 661  41  43  54  17  16]
 [ 30   3  71 147  88 542  31  53  16  19]
 [ 21  10  52  60 110  26 666  11  10  34]
 [ 34   2  24  40 118  70  10 649   3  50]
 [145  50   9  12   7   4   1   4 719  49]
 [ 57  98   6   8   9   8   9   8  39 758]]
```

在GPU加速下，速度明显更快，但两者的精度都维持在63%上下。

使用 GPU 加速，优化模型训练速度

- ## 调整 batch size

我们定义了一个列表batch_sizes，它包含了我们想要测试的不同批量大小。对于每个批量大小，我们创建了新的DataLoader实例，并记录了训练模型所需的总时间。然后，我们计算了训练速度（以样本/秒为单位），并在测试集上评估了模型的精度。给出代码：

```
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)
testset = torchvision.datasets.CIFAR10(root=data_save_pth, train=False,
download=True, transform=transform)

# 定义CNN模型
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, 3, stride=1, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(32, 64, 3, stride=1, padding=1)
        self.fc1 = nn.Linear(64 * 8 * 8, 256)
        self.fc2 = nn.Linear(256, 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, 64 * 8 * 8)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

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

```python
    criterion = nn.CrossEntropyLoss().to(device)
    trainloader = DataLoader(trainset, batch_size=batch_size, shuffle=True,
num_workers=2)
    testloader = DataLoader(testset, batch_size=batch_size, shuffle=False,
num_workers=2)

    start_time = time.time()

    for epoch in range(10):  # 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 % 100 == 99:  # 每100个小批量打印一次
                print(f'[{epoch + 1}, {i + 1}] loss: {running_loss / 100:.3f}')
                running_loss = 0.0

    print('Finished Training')

    end_time = time.time()
    elapsed_time = end_time - start_time
    print(f'Training Time (seconds) for batch_size {batch_size}:
{elapsed_time:.2f}')

    # 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()

    accuracy = 100 * correct / total if total > 0 else 0
    print(f'Accuracy (%) for batch_size {batch_size}: {accuracy:.2f}')

    memory_usage = psutil.Process(os.getpid()).memory_info().rss / (1024 * 1024)
    print(f'Memory Usage (MB) for batch_size {batch_size}: {memory_usage:.2f}')

    return elapsed_time, accuracy, memory_usage

# 训练和评估在GPU上
if torch.cuda.is_available():
    batch_sizes = [16, 32, 64, 128]  # 不同的batch_size进行比较
    for batch_size in batch_sizes:
```

```
        train_time, accuracy, memory_usage =
train_and_evaluate(torch.device("cuda"), batch_size)
else:
    print("CUDA is not available. Cannot train on GPU.")
```

运行，得到结果：

1. 当batch_size=16时：

```
Training Time (seconds) for batch_size 16: 63.72
Accuracy (%) for batch_size 16: 71.73
Memory Usage (MB) for batch_size 16: 4576.65
```

2. 当batch_size=32时：

```
Training Time (seconds) for batch_size 32: 33.94
Accuracy (%) for batch_size 32: 69.47
Memory Usage (MB) for batch_size 32: 4579.46
```

3. 当batch_size=64时：

```
Training Time (seconds) for batch_size 64: 25.62
Accuracy (%) for batch_size 64: 62.45
Memory Usage (MB) for batch_size 64: 4579.64
```

4. 当batch_size=128时：

```
Training Time (seconds) for batch_size 128: 23.20
Accuracy (%) for batch_size 128: 56.87
Memory Usage (MB) for batch_size 128: 4580.03
```

不难看出，随着batch_size的不断增高训练用时越来越短，精度逐渐降低，而内存的占用几乎没有明显的变化

- ## 使用混合精度训练（torch.cuda.amp）

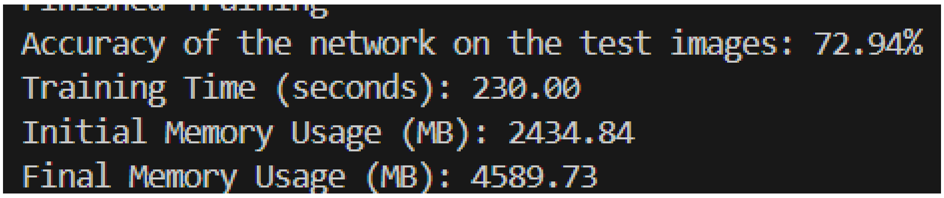
  先给出不使用该优化的代码：

```
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
```

```python
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, 32, 3, stride=1, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(32, 64, 3, stride=1, padding=1)
        self.fc1 = nn.Linear(64 * 8 * 8, 256)
        self.fc2 = nn.Linear(256, 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, 64 * 8 * 8)
        x = F.relu(self.fc1(x))
        x = self.fc2(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(10):  # 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')
```

```
    print('Finished Training')

    end_time = time.time()
    elapsed_time = end_time - start_time

    # 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()

    accuracy = 100 * correct / total if total > 0 else 0

    print(f'Training Time (seconds): {elapsed_time:.2f}')
    print(f'Accuracy (%): {accuracy:.2f}')
    print(f'Memory Usage (MB): {psutil.Process(os.getpid()).memory_info().rss / (1024 * 1024):.2f}')

    return elapsed_time, accuracy, psutil.Process(os.getpid()).memory_info().rss / (1024 * 1024)

# 训练和评估在GPU上
if torch.cuda.is_available():
    train_and_evaluate(torch.device("cuda"))
else:
    print("CUDA is not available. Cannot train on GPU.")
```

结果如下：


Accuracy of the network on the test images: 72.94%
Training Time (seconds): 230.00
Initial Memory Usage (MB): 2434.84
Final Memory Usage (MB): 4589.73

使用了混合精度训练（torch.cuda.amp）时，代码：

```
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 time
import psutil
import os

# 数据集放置路径
data_save_pth = "./data"

# 数据转换
```

```python
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, 32, 3, stride=1, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(32, 64, 3, stride=1, padding=1)
        self.fc1 = nn.Linear(64 * 8 * 8, 256)
        self.fc2 = nn.Linear(256, 10)

    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool(torch.relu(self.conv2(x)))
        x = x.view(-1, 64 * 8 * 8)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# 实例化模型、优化器、损失函数和GradScaler
model = Net().cuda()
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
criterion = nn.CrossEntropyLoss().cuda()
scaler = torch.amp.GradScaler()  # 使用新的API

# 训练和评估模型
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
model.train()

start_time = time.time()
memory_usage_start = psutil.Process(os.getpid()).memory_info().rss / (1024 *
1024)  # 记录初始内存使用

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

        optimizer.zero_grad()
```

```
        # 使用autocast进行混合精度计算
        with torch.amp.autocast(device_type='cuda'):
            outputs = model(inputs)
            loss = criterion(outputs, labels)

        # 使用GradScaler来缩放梯度
        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()

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

end_time = time.time()
elapsed_time = end_time - start_time
print('Finished Training')

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

accuracy = 100 * correct / total if total > 0 else 0
print(f'Accuracy of the network on the test images: {accuracy:.2f}%')
print(f'Training Time (seconds): {elapsed_time:.2f}')
print(f'Initial Memory Usage (MB): {memory_usage_start:.2f}')
print(f'Final Memory Usage (MB): {psutil.Process(os.getpid()).memory_info().rss / (1024 * 1024):.2f}')
```

```
Accuracy of the network on the test images: 71.56%
Training Time (seconds): 276.97
Initial Memory Usage (MB): 730.50
Final Memory Usage (MB): 1120.71
```

比较可得，使用混合精度训练使得用时和内存占用显著降低，但是精度反而下降了点。经过调查得到原因如下:

1. 数值稳定性问题：FP16的数值范围比FP32小，可能会导致数值下溢（underflow）或精度损失。这可能会在训练过程中引入舍入误差，影响模型的收敛和最终性能

   。

2. 梯度溢出或下溢：在FP16下，梯度的数值可能变得非常小，以至于在FP16格式下无法有效表示，这可能导致梯度被置为零，从而影响模型的学习

   。

3. 模型对精度敏感：某些模型或任务对数值精度非常敏感。在这些情况下，混合精度训练可能会因为精度损失而导致模型性能下降
。
4. 硬件兼容性：不是所有的硬件都支持FP16运算。在没有专门Tensor Core的GPU上，使用FP16可能不会带来预期的性能提升，甚至可能影响模型的精度
。
5. 调试和分析困难：使用混合精度训练可能会使得模型的调试和性能分析更加复杂，因为需要跟踪哪些操作是在FP16下执行的，哪些是在FP32下执行的。这种复杂性可能导致错误的配置或实现，从而影响模型的性能
。
6. 模型泛化能力：在某些情况下，混合精度训练可能会影响模型的泛化能力，尤其是在模型对精度非常敏感的情况下。因此，可能需要对模型进行微调以确保其性能和精度的稳定性
。

为了优化混合精度训练并减少精度损失，我们可以采取一些措施，比如使用 torch.amp.autocast 和 torch.amp.GradScaler 来实现自动混合精度训练，autocast 会自动选择使用 FP16 或 FP32 进行计算，以提高性能并减少内存使用，通过 GradScaler 的动态缩放，可以减少由于 FP16 精度限制导致的数值稳定性问题。代码如下：

```
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
from torch.amp import autocast, GradScaler  # 正确的导入方式
import time
import psutil
import os

# 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
```

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, 3, stride=1, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(32, 64, 3, stride=1, padding=1)
        self.fc1 = nn.Linear(64 * 8 * 8, 256)
        self.fc2 = nn.Linear(256, 10)

    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool(torch.relu(self.conv2(x)))
        x = x.view(-1, 64 * 8 * 8)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# 实例化模型、优化器、损失函数和GradScaler
model = Net().cuda()
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
criterion = nn.CrossEntropyLoss().cuda()
scaler = GradScaler()  # 初始化GradScaler

# 训练和评估模型
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
model.train()

start_time = time.time()
memory_usage_start = psutil.Process(os.getpid()).memory_info().rss / (1024 *
1024)  # 记录初始内存使用

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

        optimizer.zero_grad()

        # 使用autocast进行混合精度计算
        with autocast(device_type='cuda'):  # 正确的autocast使用方式
            outputs = model(inputs)
            loss = criterion(outputs, labels)

        # 使用GradScaler来缩放梯度
        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()

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

end_time = time.time()
elapsed_time = end_time - start_time
print('Finished Training')
```

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

accuracy = 100 * correct / total if total > 0 else 0
print(f'Accuracy of the network on the test images: {accuracy:.2f}%')
print(f'Training Time (seconds): {elapsed_time:.2f}')
print(f'Initial Memory Usage (MB): {memory_usage_start:.2f}')
print(f'Final Memory Usage (MB): {psutil.Process(os.getpid()).memory_info().rss /
(1024 * 1024):.2f}')
```

- ## 使用混合精度训练（torch.cuda.amp）下，调整 batch size

结合以上两者，我们不妨直接给出测试代码：

```python
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
from torch.amp import autocast, GradScaler  # 正确的导入方式
import time
import psutil
import os

# 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)
testset = torchvision.datasets.CIFAR10(root=data_save_pth, train=False,
download=True, transform=transform)

# 定义CNN模型
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, 3, stride=1, padding=1)
```

```
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(32, 64, 3, stride=1, padding=1)
        self.fc1 = nn.Linear(64 * 8 * 8, 256)
        self.fc2 = nn.Linear(256, 10)

    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool(torch.relu(self.conv2(x)))
        x = x.view(-1, 64 * 8 * 8)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# 实例化模型、优化器和损失函数
model = Net().cuda()
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
criterion = nn.CrossEntropyLoss().cuda()
scaler = GradScaler()  # 初始化GradScaler

# 训练和评估模型
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

# 测试不同的batch size
batch_sizes = [32, 64, 128, 256]  # 不同的batch size
for batch_size in batch_sizes:
    trainloader = DataLoader(trainset, batch_size=batch_size, shuffle=True,
num_workers=2)
    testloader = DataLoader(testset, batch_size=batch_size, shuffle=False,
num_workers=2)

    model.train()
    start_time = time.time()
    memory_usage_start = psutil.Process(os.getpid()).memory_info().rss / (1024 *
1024)

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

            with autocast(device_type='cuda'):  # 正确的autocast使用方式
                outputs = model(inputs)
                loss = criterion(outputs, labels)

            scaler.scale(loss).backward()
            scaler.step(optimizer)
            scaler.update()

    end_time = time.time()
    elapsed_time = end_time - start_time
    print(f'Batch size: {batch_size}')
    print(f'Training Time (seconds): {elapsed_time:.2f}')
    print(f'Initial Memory Usage (MB): {memory_usage_start:.2f}')
    print(f'Final Memory Usage (MB):
{psutil.Process(os.getpid()).memory_info().rss / (1024 * 1024):.2f}')
```

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

accuracy = 100 * correct / total if total > 0 else 0
print(f'Accuracy of the network on the test images: {accuracy:.2f}%\n')
```

得到结果如下：

```
Batch size: 32
Training Time (seconds): 42.74
Initial Memory Usage (MB): 732.80
Final Memory Usage (MB): 1092.00
Accuracy of the network on the test images: 69.92%

Batch size: 64
Training Time (seconds): 26.61
Initial Memory Usage (MB): 1130.07
Final Memory Usage (MB): 1134.13
Accuracy of the network on the test images: 71.38%

Batch size: 128
Training Time (seconds): 24.12
Initial Memory Usage (MB): 1135.01
Final Memory Usage (MB): 1147.59
Accuracy of the network on the test images: 71.82%

Batch size: 256
Training Time (seconds): 27.74
Initial Memory Usage (MB): 1147.66
Final Memory Usage (MB): 1147.73
Accuracy of the network on the test images: 72.25%
```

可以看出：较大的 batch size 可以提高 GPU 的利用率，因为它允许更高效地进行并行计算；更大的 batch size 意味着每个 epoch 中的迭代次数减少，从而减少了训练所需的总迭代次数；同时较大的 batch size 会占用更多的显存。

- ## 使用学习率调度器（Learning Rate Scheduler）

  为了比较，我们先给出不使用学习率调度器的代码，并运行：

```python
import os
import torch
import torchvision
```

```python
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, 32, 3, stride=1, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(32, 64, 3, stride=1, padding=1)
        self.fc1 = nn.Linear(64 * 8 * 8, 256)
        self.fc2 = nn.Linear(256, 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, 64 * 8 * 8)
        x = F.relu(self.fc1(x))
        x = self.fc2(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(10):  # 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')

    print('Finished Training')

    end_time = time.time()
    elapsed_time = end_time - start_time

    # 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()

    accuracy = 100 * correct / total if total > 0 else 0

    print(f'Training Time (seconds): {elapsed_time:.2f}')
    print(f'Accuracy (%): {accuracy:.2f}')
    print(f'Memory Usage (MB): {psutil.Process(os.getpid()).memory_info().rss / (1024 * 1024):.2f}')

    return elapsed_time, accuracy, psutil.Process(os.getpid()).memory_info().rss / (1024 * 1024)

# 训练和评估在GPU上
if torch.cuda.is_available():
    train_and_evaluate(torch.device("cuda"))
else:
    print("CUDA is not available. Cannot train on GPU.")
```

得到结果如下：

```
Training Time (seconds): 283.13
Accuracy (%): 71.76
Memory Usage (MB): 4596.39
```

接下来我们再使用学习率调度器，直接贴出代码：

```python
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, 32, 3, stride=1, padding=1)
        self.pool  = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(32, 64, 3, stride=1, padding=1)
        self.fc1 = nn.Linear(64 * 8 * 8, 256)
        self.fc2 = nn.Linear(256, 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, 64 * 8 * 8)
        x = F.relu(self.fc1(x))
        x = self.fc2(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)
```

```python
    scaler = torch.amp.GradScaler()  # 初始化GradScaler

    start_time = time.time()

    for epoch in range(10):  # 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()

            with torch.amp.autocast(device_type="cuda"):  # 开启自动混合精度
                outputs = net(inputs)
                loss = criterion(outputs, labels)

            scaler.scale(loss).backward()  # 缩放损失以避免梯度下溢
            scaler.step(optimizer)  # 更新参数
            scaler.update()  # 更新缩放器

            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')

    print('Finished Training')

    end_time = time.time()
    elapsed_time = end_time - start_time

    # 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()

    accuracy = 100 * correct / total if total > 0 else 0

    print(f'Training Time (seconds): {elapsed_time:.2f}')
    print(f'Accuracy (%): {accuracy:.2f}')
    print(f'Memory Usage (MB): {psutil.Process(os.getpid()).memory_info().rss /
(1024 * 1024):.2f}')

    return elapsed_time, accuracy, psutil.Process(os.getpid()).memory_info().rss
/ (1024 * 1024)

# 训练和评估在GPU上
if torch.cuda.is_available():
    train_and_evaluate(torch.device("cuda"))
```

```
else:
    print("CUDA is not available. Cannot train on GPU.")
```

运行得到结果如下：

```
Training Time (seconds): 221.61
Accuracy (%): 75.25
Memory Usage (MB): 4579.92
```

不难看出，虽然用时变短，内存占用几乎没变化，但精确度确实存在一定的提高。

使用cuDNN

安装cuDNN，我们可以进入官网
按照所给命令在终端上输入即可，如我的：

```
 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
```

通过以上指令，我们成功安装了cuDNN，接下来我们可以使用cuDNN来训练模型。
以下为代码：

```
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 time

# 定义CNN模型
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(64 * 8 * 8, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool(torch.relu(self.conv2(x)))
        x = x.view(-1, 64 * 8 * 8)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
```

```python
        return x

# 数据集路径
data_dir = './data'

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

# 加载CIFAR-10数据集
train_dataset = torchvision.datasets.CIFAR10(root=data_dir, train=True,
                                             download=True, transform=transform)
test_dataset = torchvision.datasets.CIFAR10(root=data_dir, train=False,
                                           download=True, transform=transform)

# 创建数据加载器
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True,
num_workers=2)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False,
num_workers=2)

# 设置设备
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# 初始化模型、优化器和损失函数
model = SimpleCNN().to(device)
optimizer = optim.Adam(model.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()

# 启用cuDNN
torch.backends.cudnn.enabled = True
torch.backends.cudnn.benchmark = True

# 训练模型
num_epochs = 10
for epoch in range(num_epochs):
    model.train()
    start_time = time.time()
    total_loss = 0
    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    end_time = time.time()
    print(f'Epoch {epoch+1}, Loss: {total_loss / len(train_loader):.4f}, Time:
{end_time - start_time:.4f} seconds')

# 测试模型性能
model.eval()
correct = 0
```

```
total = 0
with torch.no_grad():
    for inputs, labels in test_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = model(inputs)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

accuracy = correct / total
print(f'Test Accuracy: {accuracy:.4f}')
```

运行得到以下结果：

```
Files already downloaded and verified
Files already downloaded and verified
Epoch 1, Loss: 1.3454, Time: 3.5468 seconds
Epoch 2, Loss: 0.9649, Time: 2.4095 seconds
Epoch 3, Loss: 0.8063, Time: 2.5018 seconds
Epoch 4, Loss: 0.6838, Time: 2.5234 seconds
Epoch 5, Loss: 0.5761, Time: 2.4362 seconds
Epoch 6, Loss: 0.4823, Time: 2.4228 seconds
Epoch 7, Loss: 0.3945, Time: 2.4565 seconds
Epoch 8, Loss: 0.3136, Time: 2.4530 seconds
Epoch 9, Loss: 0.2399, Time: 2.5430 seconds
Epoch 10, Loss: 0.1885, Time: 2.6930 seconds
Test Accuracy: 0.7165
```

性能提升并不明显，可能原因如下：

1. GPU 没有被充分利用，可能是因为批量大小太小，或者模型的并行化程度不够。
2. 模型相对较小，或者数据集不够大，那么 GPU 和 cuDNN 提供的加速效果可能不会非常明显。
3. 系统上同时运行了其他资源密集型的进程，它们可能会与你的训练任务竞争 CPU、内存或 I/O 资源，从而影响性能。

但当我们调整batch size和workers时可以发现，速度有了明显上升。所以，cuDNN的加速在速度方面有显著作用。

不同优化方法分别在CPU和GPU上测试的性能比较

- ## 调整 batch size下的比较

```
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
from torch.amp import autocast, GradScaler
import time

# 定义CNN模型
class SimpleCNN(nn.Module):
    def __init__(self):
```

```python
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(64 * 8 * 8, 512)
        self.fc2 = nn.Linear(512, 10)

    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool(torch.relu(self.conv2(x)))
        x = x.view(-1, 64 * 8 * 8)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# 数据集路径
data_dir = './data'

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

# 加载CIFAR-10数据集
train_dataset = torchvision.datasets.CIFAR10(root=data_dir, train=True,
                                             download=True, transform=transform)
test_dataset = torchvision.datasets.CIFAR10(root=data_dir, train=False,
                                            download=True, transform=transform)

# 创建数据加载器
def create_loader(dataset, batch_size, shuffle=True):
    return DataLoader(dataset, batch_size=batch_size, shuffle=shuffle,
num_workers=2)

# 初始化模型、优化器和损失函数
def init_model(device, batch_size):
    model = SimpleCNN().to(device)
    optimizer = optim.Adam(model.parameters(), lr=0.001)
    criterion = nn.CrossEntropyLoss()
    scaler = GradScaler()
    train_loader = create_loader(train_dataset, batch_size)
    test_loader = create_loader(test_dataset, batch_size, shuffle=False)
    return model, optimizer, criterion, scaler, train_loader, test_loader

# 训练模型
def train_model(model, device, train_loader, optimizer, criterion, scaler):
    model.train()
    total_loss = 0
    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()

        # 使用自动混合精度，指定device_type
        if device.type == 'cuda':
            with autocast(device_type='cuda'):
```

```
                outputs = model(inputs)
                loss = criterion(outputs, labels)
        else:
            outputs = model(inputs)
            loss = criterion(outputs, labels)

        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()
        total_loss += loss.item()
    return total_loss / len(train_loader)

# 测试模型性能
def test_model(model, device, test_loader):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in test_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    return correct / total

# 主函数
def main():
    batch_sizes = [32, 64, 128]  # 测试不同的批量大小
    for batch_size in batch_sizes:
        print(f"Testing with batch size: {batch_size}")

        # 在CPU上测试
        device = torch.device("cpu")
        model_cpu, optimizer_cpu, criterion_cpu, scaler_cpu, train_loader_cpu,
test_loader_cpu = init_model(device, batch_size)
        start_time_cpu = time.time()
        train_loss_cpu = train_model(model_cpu, device, train_loader_cpu,
optimizer_cpu, criterion_cpu, scaler_cpu)
        test_acc_cpu = test_model(model_cpu, device, test_loader_cpu)
        print(f"CPU - Loss: {train_loss_cpu:.4f}, Accuracy: {test_acc_cpu:.4f},
Time: {time.time() - start_time_cpu:.4f} seconds")

        # 在GPU上测试
        if torch.cuda.is_available():
            device = torch.device("cuda")
            model_gpu, optimizer_gpu, criterion_gpu, scaler_gpu,
train_loader_gpu, test_loader_gpu = init_model(device, batch_size)
            start_time_gpu = time.time()
            train_loss_gpu = train_model(model_gpu, device, train_loader_gpu,
optimizer_gpu, criterion_gpu, scaler_gpu)
            test_acc_gpu = test_model(model_gpu, device, test_loader_gpu)
            print(f"GPU - Loss: {train_loss_gpu:.4f}, Accuracy:
{test_acc_gpu:.4f}, Time: {time.time() - start_time_gpu:.4f} seconds")

if __name__ == "__main__":
```

```
    main()
```

运行得到结果：

```
Files already downloaded and verified
Files already downloaded and verified
Testing with batch size: 32
CPU - Loss: 1.2394, Accuracy: 0.6178, Time: 33.4057 seconds
GPU - Loss: 1.2152, Accuracy: 0.6636, Time: 10.3509 seconds
Testing with batch size: 64
CPU - Loss: 1.2986, Accuracy: 0.6335, Time: 23.8536 seconds
GPU - Loss: 1.2635, Accuracy: 0.6398, Time: 4.9362 seconds
Testing with batch size: 128
CPU - Loss: 1.3606, Accuracy: 0.6063, Time: 18.3343 seconds
GPU - Loss: 1.3639, Accuracy: 0.6132, Time: 4.4172 seconds
```

可以看出无论batch size多少，在GPU的加速下，精度和速度都优于CPU。
通过调查所知道的原因：

1. 并行处理能力：GPU设计用于同时处理大量数据，这使得它们在执行并行计算时非常高效。GPU拥有成百上千个核心，可以同时处理多个计算任务，而CPU通常只有几个核心。
2. 浮点运算性能：GPU在浮点运算方面通常比CPU更快，这对于深度学习、科学计算和图形渲染等任务至关重要。
3. 内存带宽：GPU通常具有更高的内存带宽，这意味着它们可以更快地读取和写入大量数据，这对于处理大型数据集和复杂模型非常有用。
   ......
   等。

使实验数据可视化

先将算出的数据放入csv文件中，如下：

```
 Optimization Technique,Training Time (seconds),Accuracy (%),Memory Usage (MB)
C_b=32_w=4,98.02,67.59,698.39
G_b=32_w=4,35.66,68.02,4600.46
C_b=64_w=4,84.21,61.31,735.86
G_b=64_w=4,22.13,61.25,4607.96
C_b=128_w=4,84.51,53.02,752.80
G_b=128_w=4,17.03,53.39,4622.02
C_b=128_w=6,81.49,53.38,797.48
G_b=128_w=6,17.56,53.45,4612.20
C_b=64_w=6,85.63,61.39,742.11
G_b=64_w=6,22.11,61.89,4603.28
C_torch.cuda.amp,479.67,72.87,667.46
G_torch.cuda.amp,359.58,71.90,4559.08
G_cuDNN,234.44,71.34,4586.90
```

其中G代表GPU，C代表CPU，b代表batch_size,w代表workers。

接下来我们使用python环境使得数据可视化，我们将会用到Python环境中的pandas库，matplotlib库和seaborn库。如若未安装可使用以下命令来安装：

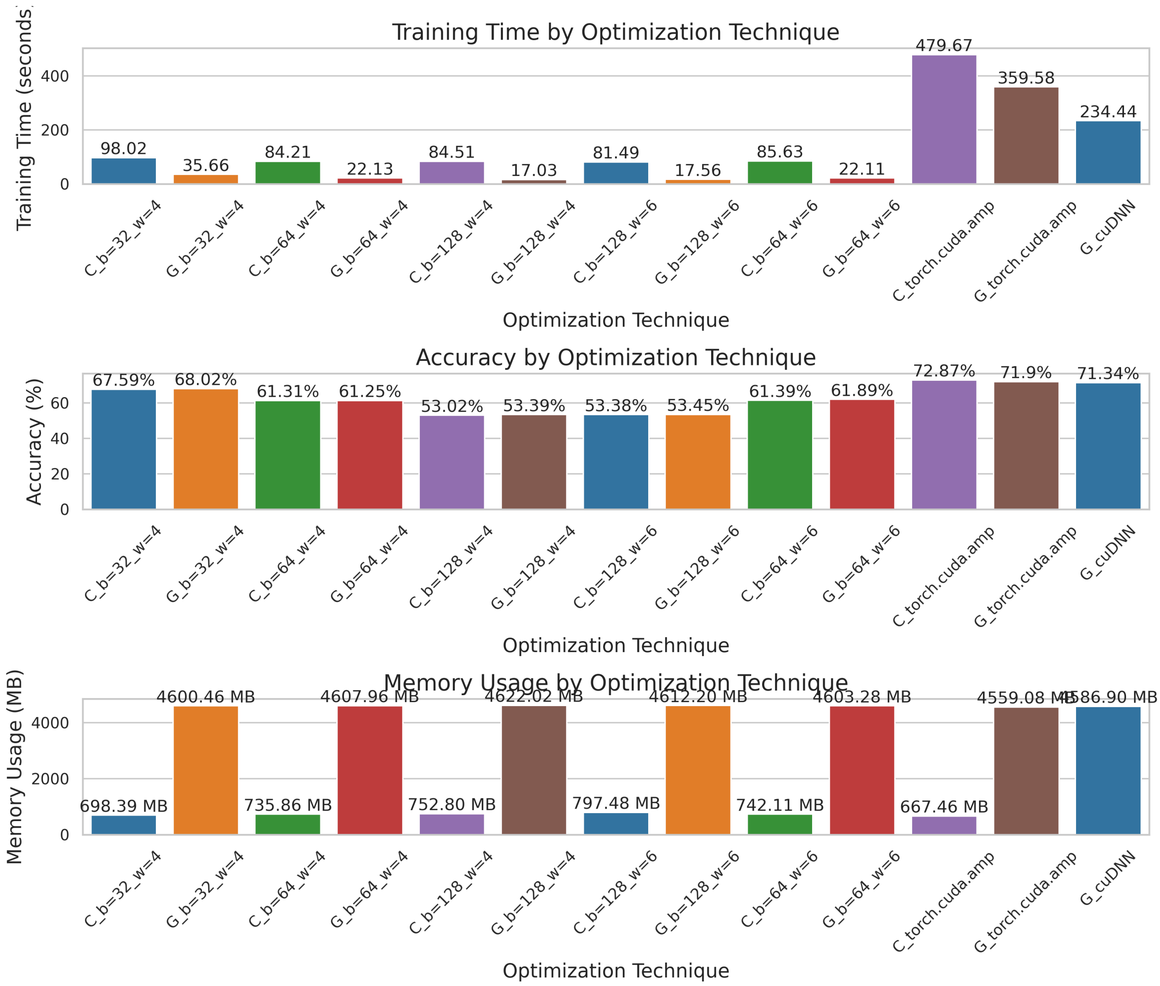
```
pip install pandas -i https://pypi.tuna.tsinghua.edu.cn/simple
pip install matplotlib -i https://pypi.tuna.tsinghua.edu.cn/simple
pip install -i https://pypi.tuna.tsinghua.edu.cn/simple seaborn
```

接下来通过代码来让电脑进行绘制：

```python
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('data.csv')

# 设置图表大小
plt.figure(figsize=(12, 10))

# 定义颜色列表
colors = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b']

# 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=16)
plt.xlabel('Optimization Technique', fontsize=14)
plt.ylabel('Training Time (seconds)', fontsize=14)
plt.xticks(rotation=45)  # 设置横轴标签旋转45度

# 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')

# 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=16)
plt.xlabel('Optimization Technique', fontsize=14)
plt.ylabel('Accuracy (%)', fontsize=14)
plt.xticks(rotation=45)  # 设置横轴标签旋转45度

# 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()}%',
             ha='center', va='bottom')

# 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=16)
plt.xlabel('Optimization Technique', fontsize=14)
plt.ylabel('Memory Usage (MB)', fontsize=14)
plt.xticks(rotation=45)  # 设置横轴标签旋转45度

# 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')

# 调整子图间距
plt.tight_layout()

# 保存图表
plt.savefig('optimization_techniques_comparison.png', dpi=300)

# 显示图表
plt.show()
```

得到结果如下：



实验中遇到的问题

所遇问题皆于以上提出并给出了解决方案。

特别鸣谢

老鸽

🌻

How

士+

老e

郑老师