

NCUSCC考核Pytorch（老登版）报告

- 报告人：石佳
 - 开始时间：2024/10/13
 - 完成时间：2024/10/28
 - 服务器配置：
 - MIRROR:pytorch 2.1.2 python 3.10 (ubuntu22.04) cuda 11.8
 - GPU:RTX 2080Tix2(22GB)*1
 - CPU:12 vCPU Intel(R) Xeon(R) Platinum 8336C CPU @2.30GHZ
-

目录

1.实验环境搭建

- 服务器的搭建与使用

2.数据集准备及训练模型

- 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
- 调整输入通道和输出单元

4.优化方式性能可视化

5.问题中遇到的问题及解决方案

- 选择服务器的原因
- CIFAR的正确下载方法
- 其他负优化的行为(批量一体化、调整学习率.....) 及其原因

6.资料&特别鸣谢

I . 实验环境搭建

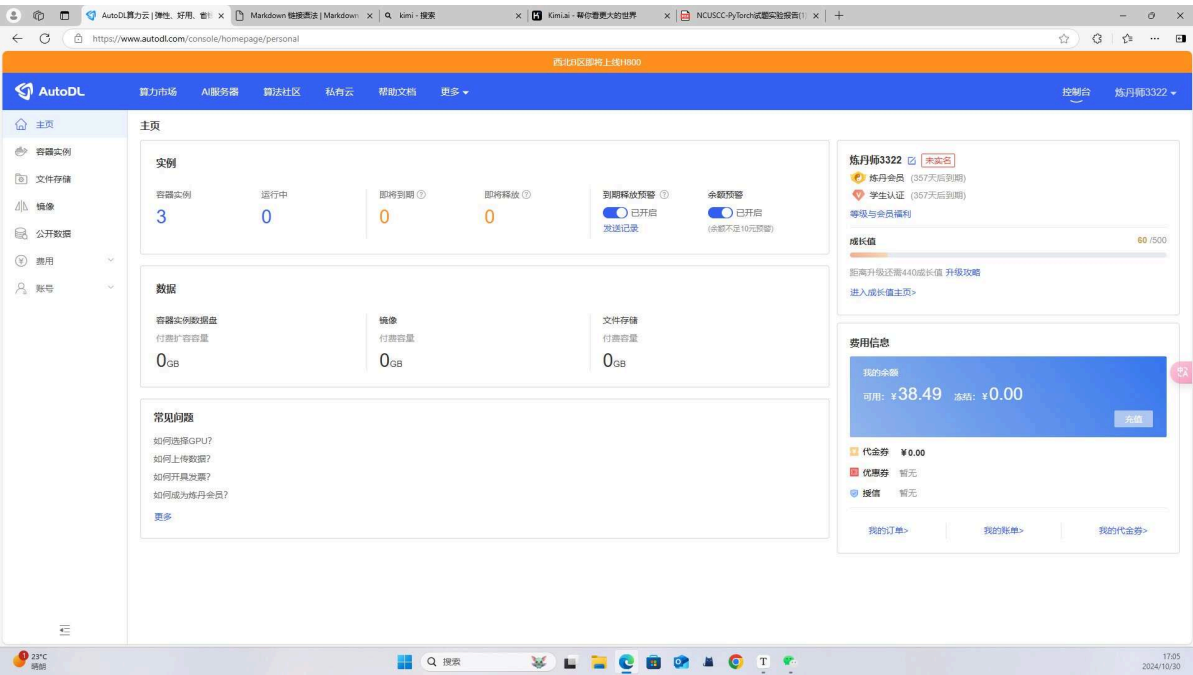
由于我的这台ROG电脑极其傻逼（这个事情帆哥也是知道的）因此在配置环境十天之后，不出预料的失败了，于是乎，最后选择了服务器进行完成这次的考核（srds有点胜之不武，但是因为还要完成其他事情只能这样子做了）。

服务器平台的选择

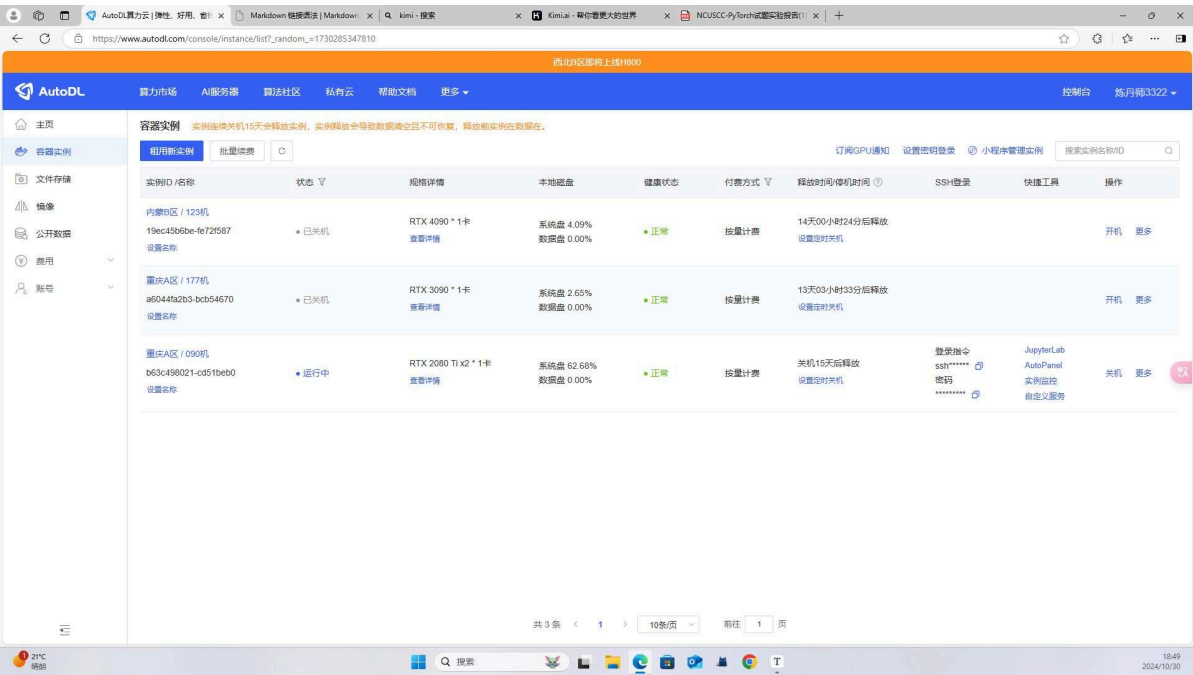
对比了华为云，阿里云等多家服务器平台之后，我最终选择了autol平台，原因有以下两点：

- 好上手，适合小白
- 价格便宜，适合学生党

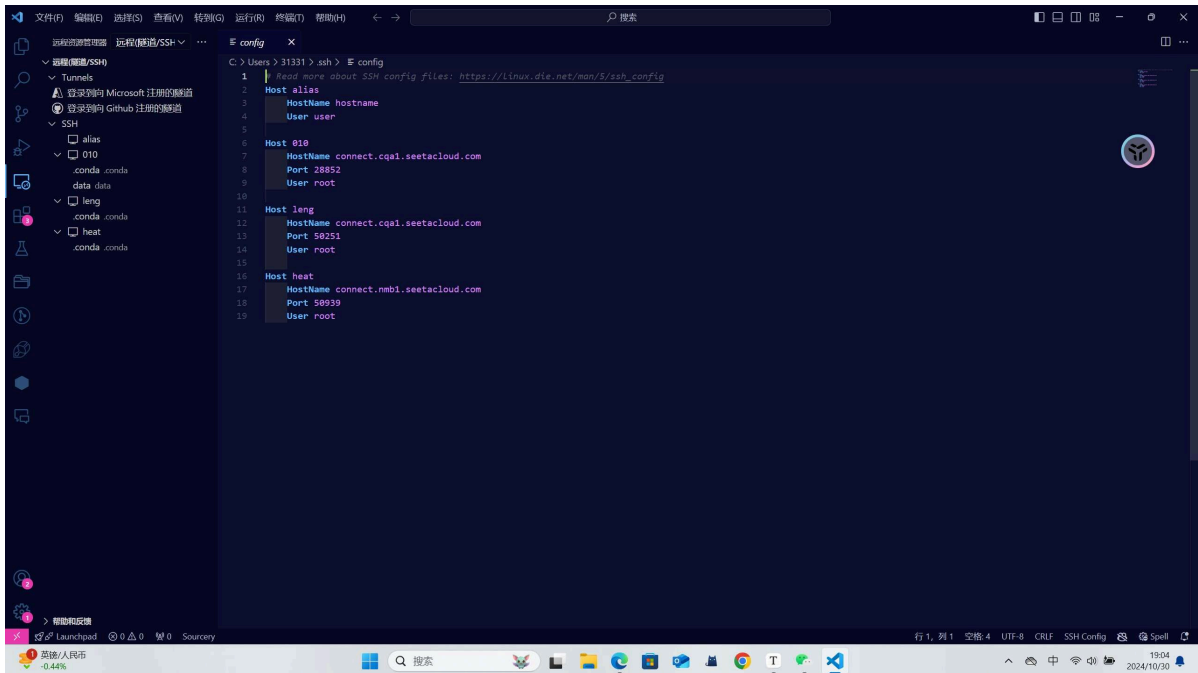
在注册完之后，我们会出现以下页面



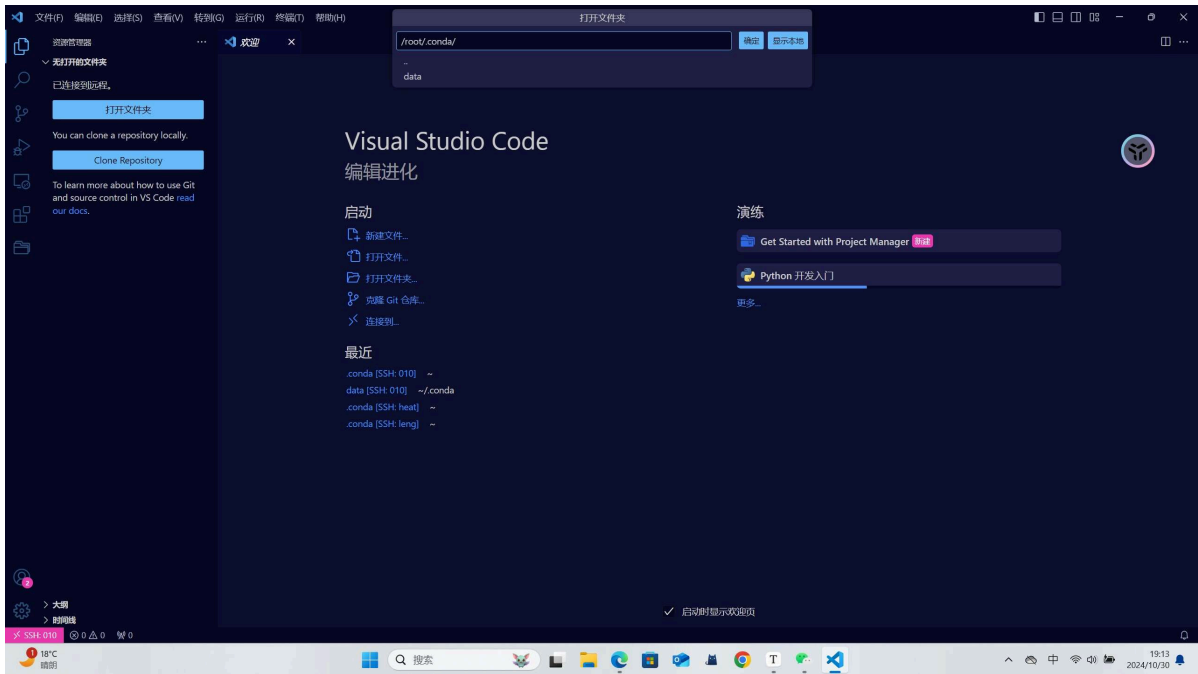
点击算力市场，根据个人情况选卡进行租借，这里我选择的是RTX 2080Ti的服务器，然后会出现以下页面



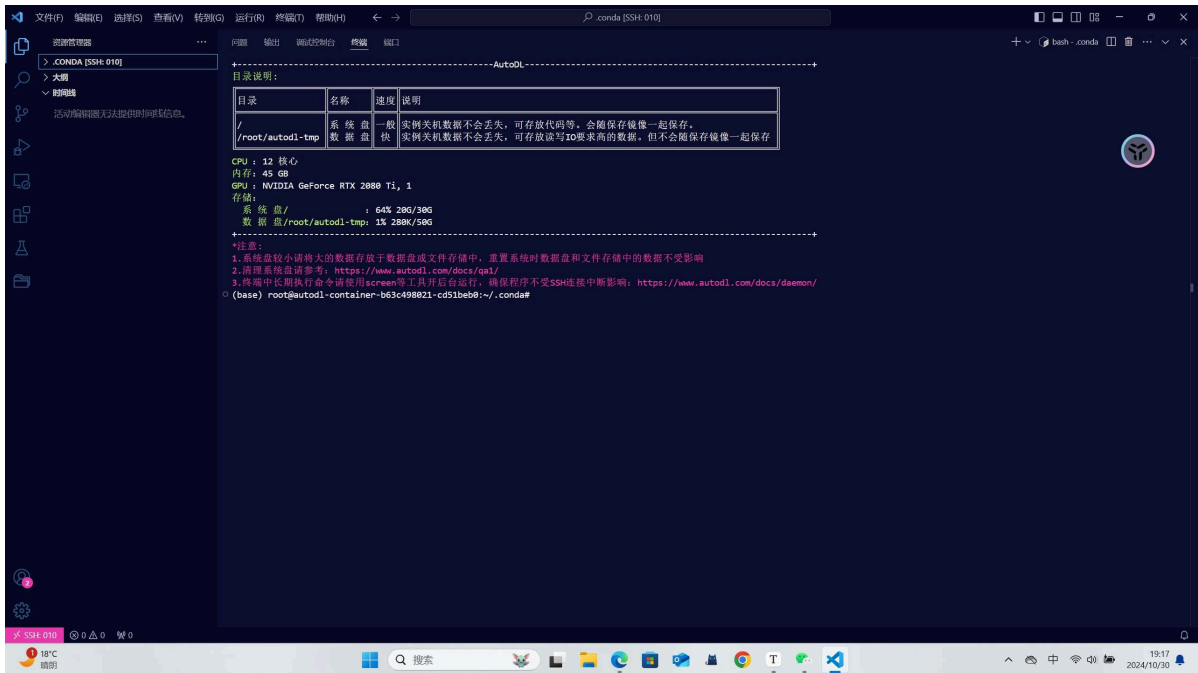
我们点击登录指令，选择ssh复制，然后打开vscode，打开远程资源管理器，点击ssh设置，选择ssh配置文件的第一个，如图所示：



去csdn或者一些论坛上找一下如何进行基础的配置之后，会出现以下页面



我们点击选择conda进行配置，当出现以下页面时，即说明我们已经配置好环境和服务器



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

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，下面是官方加载数据的指令

```
import torch
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)
```

so，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，在kimi的帮助下一次的修改后，用以下代码来实现深度学习模型的构建

```
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(2): # 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
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.')

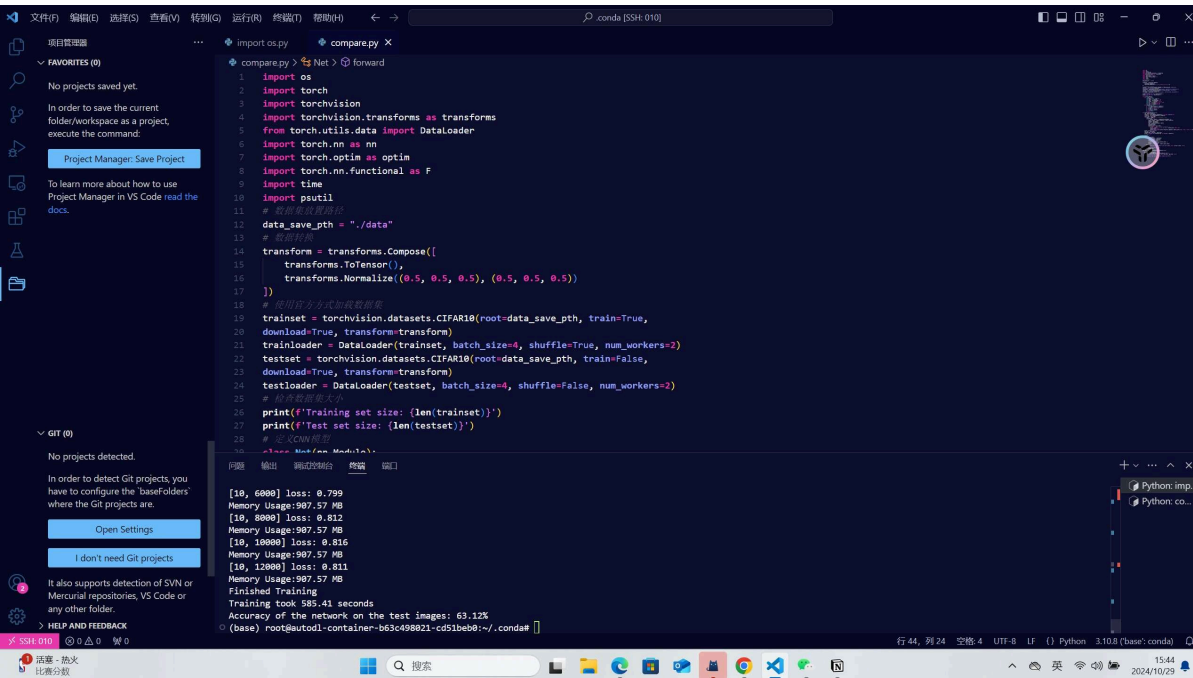
```

上面可以看到，利用kimi生成的CNN模型包含着基础的卷积层、池化层和全连接层，同时也加进来了损失函数和优化器。与此同时，我们的实例化、循环训练、精度计算和最后的时间计算也包含在内，并且该模型可以调整batch size, worker和学习率。

于是，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就这样被我们做好了。

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。

以10次循环为例，我们可以得到以下结果



```
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)}')
```

训练输出

Iteration	loss	Memory Usage
[10, 6000]	0.799	907.57 MB
[10, 8000]	0.812	907.57 MB
[10, 10000]	0.816	907.57 MB
[10, 12000]	0.811	907.57 MB

Finished Training
Training took 585.41 seconds
Accuracy of the network on the test images: 63.12%

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585.41s，loss在逐渐减少，精度为63.12%，内存平均占用907.57MB，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586.15s，907.33MB，63.02%，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，我调教kimi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_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_path,
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__()
        #卷积层 1
        self.conv1 = nn.Conv2d(3, 6, 5)
        # 池化层 1
        self.pool = nn.MaxPool2d(2, 2)
        #卷积层 1
        self.conv2 = nn.Conv2d(6, 16, 5)
        # 全连接层 1
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        # 全连接层 2
        self.fc2 = nn.Linear(120, 84)
        # 全连接层 3
        self.fc3 = nn.Linear(84, 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, 16 * 5 * 5)
        # 第一个全连接层 + 激活函数
        x = F.relu(self.fc1(x))
        # 第二个全连接层 + 激活函数
        x = F.relu(self.fc2(x))
        # 第三个全连接层
        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，我们得到以下结果以2次为例，我们得到以下结果

Memory Usage: 592.33 MB

Finished Training

Training took 81.85seconds on cpu

Accuracy of the network on the test images: 54.96%

Memory Usage: 917.85 MB

Finished Training

Training took 102.71 seconds on cuda

Accuracy of the network on the test images: 56.70%

Speedup on GPU compared to CPU: 0.80x

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

我们接着增大到十次

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, 63.07%, 692.47MB，仍然是比GPU优秀

如果我运算次数扩大到30次，even more?

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运算时间超过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（即多次循环）。

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。

下面是用这种方式优化后的代码:

```
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，下面是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()
```

#.....

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结合是最高效的方法，以下是代码

```
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32，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.调整输入通道和输出单元

在设计卷积神经网络（CNN）时，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
```

Memory Usage: 1039.16 MB

Finished Training

Training took 339.94 seconds

Accuracy of the network on the test images: 73.15%

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: 1097.75 MB

Finished Training

Training took 606.42 seconds

Accuracy of the network on the test images: 79.01%

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

并且，我们可以发现如果是在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的猜测是一致的

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。

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

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('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()

```

值得注意的是，最开始的服务器一般是没有自带pandas和seaborn库的，因为我们是在conda环境里，所以可以直接使用conda下载

接下来是最终呈现的样子

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



此图可以验证我们之前关于性能优化的猜测

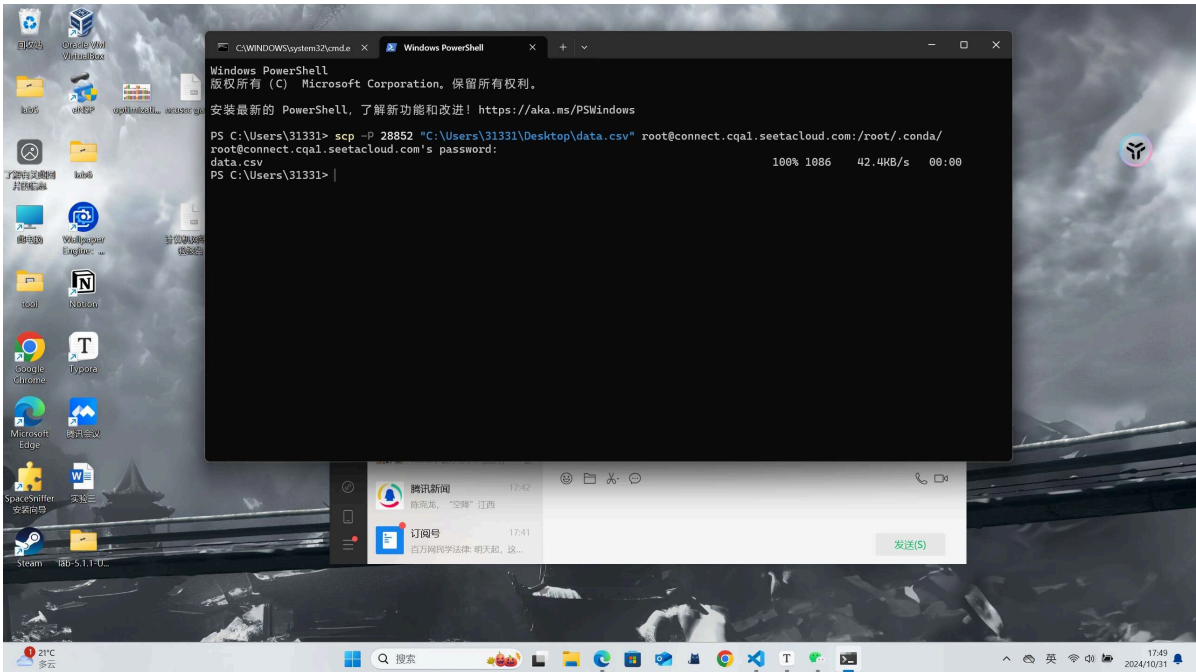
V.实验过程中遇到的问题及解决方案

①.服务器平台的选择

正如我之前所说，被虚拟机环境硬控了十天还是失败了，而且ROG这边官方也不出面解决问题，再加上我还要一生一芯和嵌入式比赛的ddl，于是乎只能选择服务器进行开发。我一开始是选择了阿里云，但是太贵了，只能作罢选择相对便宜的autodl，在最开始也不知道服务器到底怎么使用，也是去网上找了教程才学会的基础的使用。

②.CSV上传到服务器

和用虚拟机的其他同学不一样，因为我们的操作都在服务器上运行的，所以我们需要传到服务器上，最开始找了很多办法都没有解决，直到我问kimi，只需要拿到自己的csv文件地址和服务器端口就可以了，接下来是解决的方法演示



(我的背景真好看)

VI.参考资料&特别鸣谢

参考资料

特别鸣谢

[老鸽](#)和[彩彩](#)的无私技术的帮助（小登太强了）

[塔菲老师](#)关于环境问题的耐心解决（感动哭了）

小登们的鼓励与压力

[kimi](#)

[向日葵](#)和[太阳王子](#)关于typora的问题的回答

炒蒜的大家