手写数字识别卷积网络

1　环境配置

1.1　程序运行环境

Python3.7+Anaconda+Pytorch+Cuda

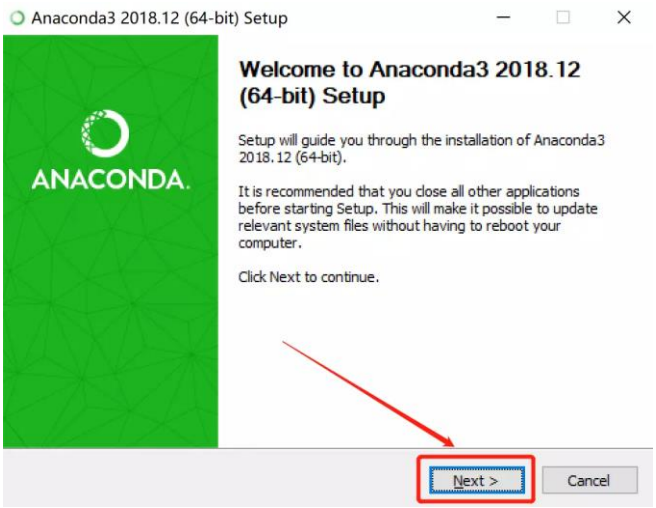
1.2　Anaconda 环境配置

第一步：下载安装 Anaconda

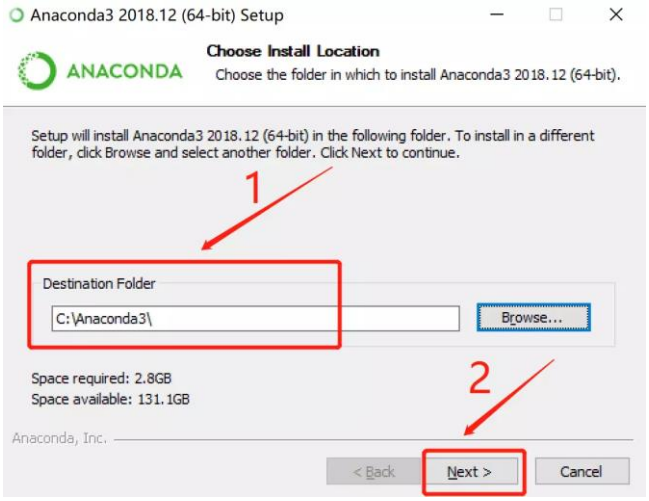
下载地址：https://www.anaconda.com/download/

Anaconda 是跨平台的，有 Windows、macOS、Linux 版本，根据自己的操作系统选择相应的版本下载。这里选择 Python3.7 版本的进行下载。

下载完成后会得到一个类似于 Anaconda2-5.2.0-Windows-x86_64.exe 的文件，双击打开进行安装即可。

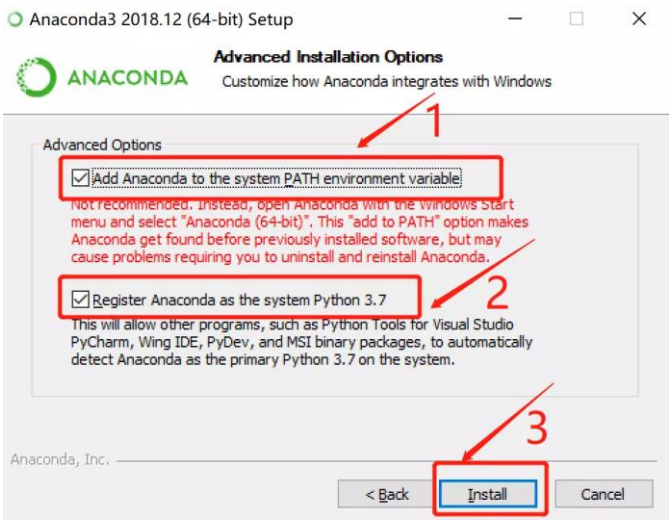


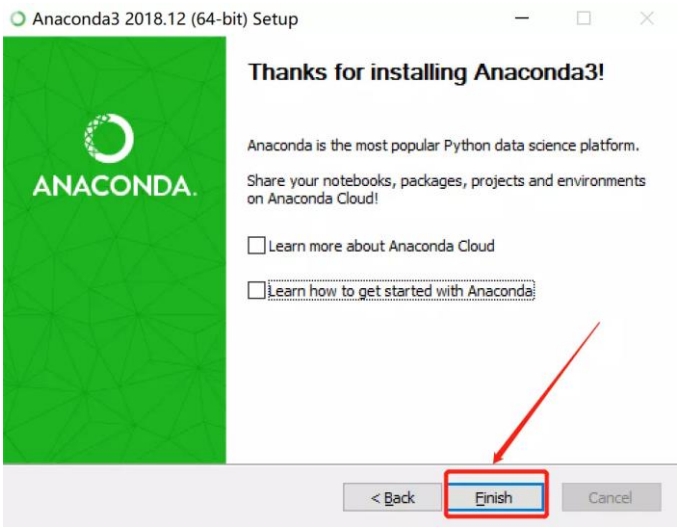
一路点击 Next，到这一步：



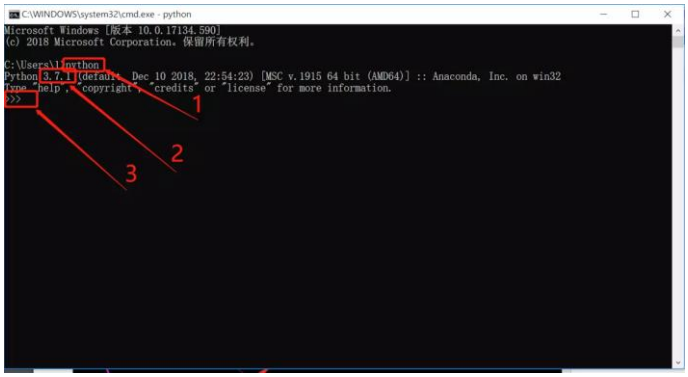
选择安装路径，也可以安装在默认路径下。

保持两个都在勾选状态，点击安装，等待安装完毕。





安装结束后，打开电脑的 CMD 命令行（使用 WIN 键+R， 然后输入 CMD，点击 OK），在其中输入 python，如果返回的信息为下图中的 2，并且提示符为下图中的 3，则安装顺利完成。



第二步：创建 Conda 虚拟环境

使用 conda create -n your_env_name python=X.X（2.7、3.7 等） anaconda 命令创建 python 版本为 X.X、名字为 your_env_name 的虚拟环境。your_env_name 文件可以在 Anaconda 安装目录 envs 文件下找到。

使用如下命令即可激活你的虚拟环境:

Linux: source activate your_env_name(虚拟环境名称)

Windows: activate your_env_name(虚拟环境名称)

关闭虚拟环境使用如下命令即可：

linux: source deactivate

Windows: deactivate

第三步：运行 Jupyter notebook

首先在虚拟环境命令行中运行命令：

conda install jupyter notebook

安装完成之后在命令行中切换到文件所在目录下，打开 Jupyter Notebook：

jupyter notebook

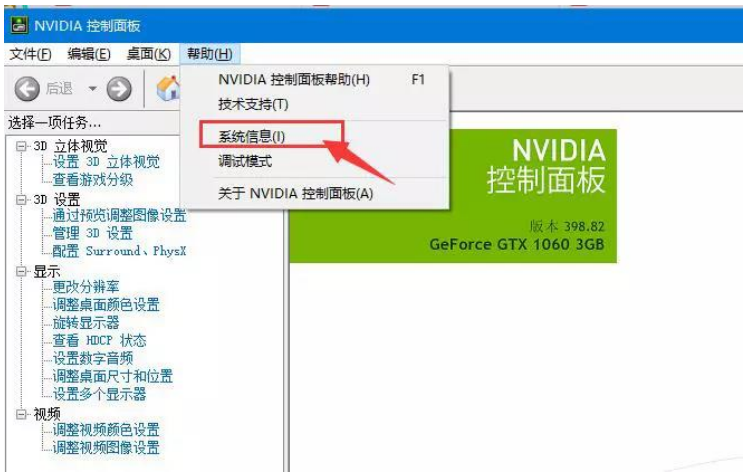
```
$ jupyter notebook
[I 08:58:24.417 NotebookApp] Serving notebooks from local directory: /Users/catherine
[I 08:58:24.417 NotebookApp] 0 active kernels
[I 08:58:24.417 NotebookApp] The Jupyter Notebook is running at: http://localhost:8888/
[I 08:58:24.417 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice t
```

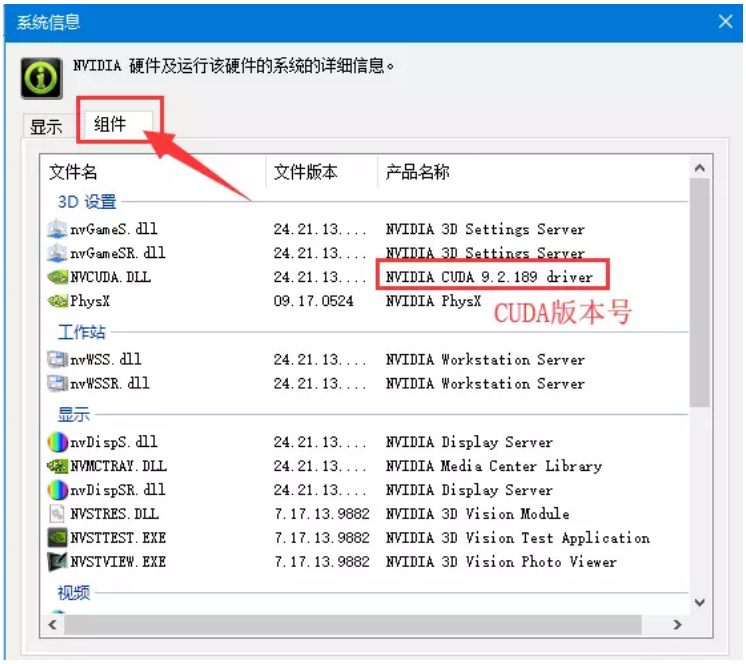
出现这种信息表示打开正常，复制链接到浏览器即可使用。

1.3 Cuda 环境配置

首先查看显卡对应的 cuda 版本，如果版本不对会无法使用。

打开控制面板，点击 NVIDIA 控制面板，选择进入导航栏的帮助中的系统信息，点击组件，可以看到显卡相关信息。





选择对应版本的 Cuda 之后进行安装：

下载地址：https://developer.nvidia.com/cuda-toolkit-archive

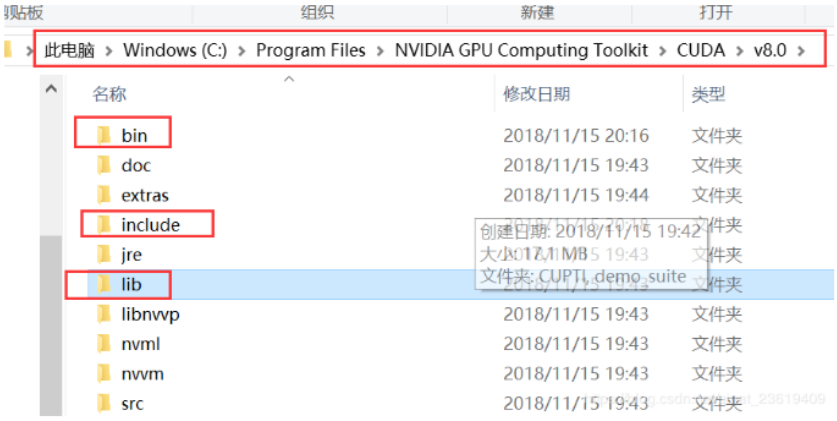
选择自定义安装，只需选择 CUDA 下面这 4 项就够了

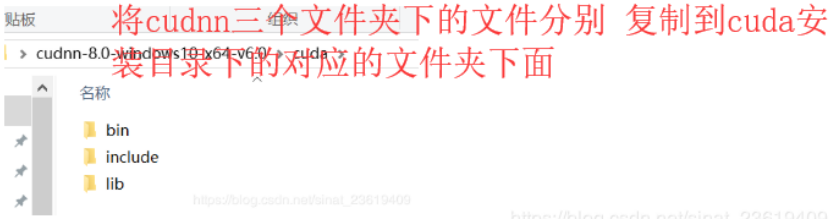
安装完成之后测试一下，命令行输入 nvcc -V，有显示即安装成功。

下载完 Cuda 之后接着要下载 cuDNN：

下载地址：https://developer.nvidia.com/rdp/cudnn-download

注意要下载与 Cuda 版本对应的 cuDNN。

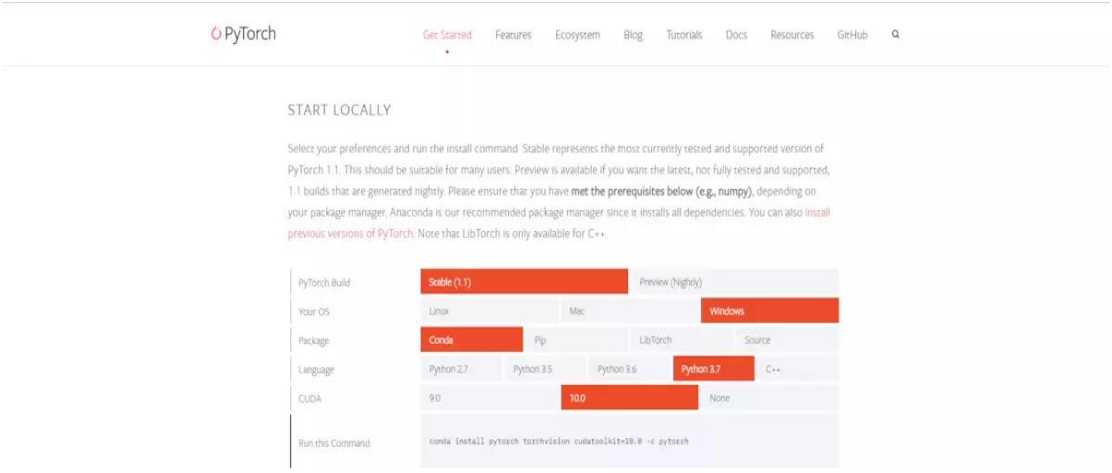




将cudnn三个文件夹下的文件分别 复制到cuda安装目录下的对应的文件夹下面

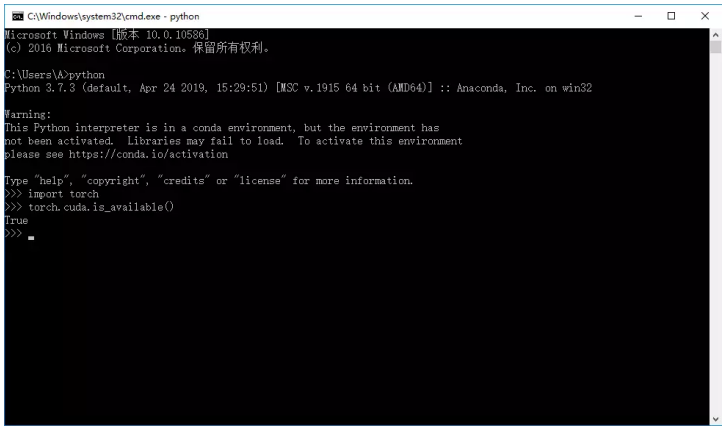
1.4 Pytorch 环境配置

下载地址：https://pytorch.org/get-started/previous-versions/



在 Conda 虚拟环境中运行网站给出的命令即可。

验证 pytorch 的 gpu 是否可用，返回结果为 True 则表示 GPU 可用。



2 程序运行指导

初次运行程序时可能会缺少一些库文件，使用 pip 安装相应的库即可。

程序有两个版本，一个是.ipynb 文件，一个是.py 文件，均可以运行。ipynb 文件方便观察程序运行的每一步的结果，便于调试和理解。ipynb 文件需使用 Jupyter notebook 运行。

首先看 ipynb 文件的运行：

在 Jupyter Notebook 中打开文件，然后对于每一个 Cell，按住 shift+enter 运行。

In [1]:
```
import  torchvision,torch
import  torchvision.transforms as transforms
from torch.utils.data import Dataset, DataLoader
import matplotlib.pyplot as plt
import numpy as np
```

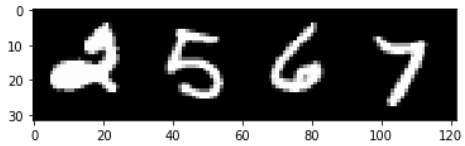
这个 cell 中加载了程序需要的一些第三方库文件。

In [2]:
```
trainset = torchvision.datasets.MNIST(root='./mnist/',train=True,
                                       download=True, transform= transforms.ToTensor())
trainloader = DataLoader(trainset, batch_size=4,shuffle=True, num_workers=4)

testset = torchvision.datasets.MNIST(root='./mnist/',train=False,
                                       download=True, transform= transforms.ToTensor())
testloader = DataLoader(testset, batch_size=4,shuffle=True, num_workers=4)
```

这个 cell 加载了 minist 手写数字的数据集。这个数据集已经下载好了，可以直接调用，如果没有下载的话运行这个 cell 会自动去下载。

In [5]:
```
classes = ('0', '1', '2', '3',
           '4', '5', '6', '7', '8', '9')
print('01',trainloader,'\n')
dataiter = iter(trainloader)
print('02',dataiter)
images, labels = dataiter.next()
print(images.shape,'\n')
print(labels)

def imshow(img):
    npimg = img.numpy()
    print(npimg.shape)
    plt.imshow(npimg.transpose((1,2,0)))

imshow(torchvision.utils.make_grid(images))
```

```
01 <torch.utils.data.dataloader.DataLoader object at 0x7f74f2fb6890>

02 <torch.utils.data.dataloader._MultiProcessingDataLoaderIter object at 0x7f74f2fc5150>
torch.Size([4, 1, 28, 28])

tensor([2, 5, 6, 7])
(3, 32, 122)
```



如图，这个 cell 是对数据加载的正确性进行一次检测，确保训练时数据不会出错。

In [5]:
```
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim


device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
#device = torch.device("cpu")
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 4 * 4, 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 * 4 * 4)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


print(device)
net = Net().to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

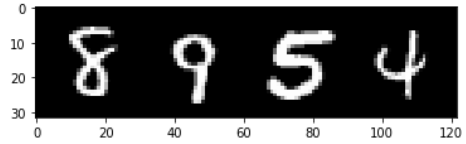
这个 cell 主要是进行网络的搭建，这里没有使用目前比较流行的图像处理的深度卷积网络模型，而是使用简单的两层卷积配合池化层和三个全连接层一起对图像进行处理。其中 device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")控制了这个模型是在 CPU 还是在 GPU 上进行训练。ipynb 文件需要手动调换训练的平台，py 文件可以直接输入参数来调整。

In [6]:
```
import time

start = time.time()
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data
        inputs = inputs.to(device)
        labels = labels.to(device)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
       # print(outputs.shape,'\n')
       # print(labels.shape,'\n')
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0


end = time.time()
print('Finished Training')
print('training time is %6.3f' % (end - start))
```

网络搭建完之后开始对网络进行训练，在 Minist 的训练集上进行两次迭代，同时记录训练的时间。

In [10]:
```
dataiter = iter(testloader)
images, labels = dataiter.next()

imshow(torchvision.utils.make_grid(images))
print('GroundTruth: ', ' '.join('%5s' % classes[labels[j]] for j in range(4)))


images = images.to(device)
labels = labels.to(device)
outputs = net(images)

_, predicted = torch.max(outputs, 1)

print('Predicted:   ', ' '.join('%5s' % classes[predicted[j]]
                              for j in range(4)))
```

```
(3, 32, 122)
GroundTruth:     8     9     5     4
Predicted:       8     9     5     4
```



对训练的结果进行一次测试。

In [12]:
```
class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))
with torch.no_grad():
    for data in testloader:
        images, labels = data
        images = images.to(device)
        labels = labels.to(device)
        outputs = net(images)
        _, predicted = torch.max(outputs, 1)
        c = (predicted == labels).squeeze()
        for i in range(4):
            label = labels[i]
            class_correct[label] += c[i].item()
            class_total[label] += 1


for i in range(10):
    print('Accuracy of %5s : %2d %%' % (
        classes[i], 100 * class_correct[i] / class_total[i]))
```

```
Accuracy of     0 : 99 %
Accuracy of     1 : 98 %
Accuracy of     2 : 99 %
Accuracy of     3 : 98 %
Accuracy of     4 : 96 %
Accuracy of     5 : 99 %
Accuracy of     6 : 96 %
Accuracy of     7 : 95 %
Accuracy of     8 : 97 %
Accuracy of     9 : 96 %
```

对每一类的分类的正确性做一次评估。

3 训练时间对比

```
[1,  2000] loss: 1.577
[1,  4000] loss: 0.338
[1,  6000] loss: 0.209
[1,  8000] loss: 0.167
[1, 10000] loss: 0.141
[1, 12000] loss: 0.116
[1, 14000] loss: 0.102
[2,  2000] loss: 0.087
[2,  4000] loss: 0.085
[2,  6000] loss: 0.081
[2,  8000] loss: 0.068
[2, 10000] loss: 0.081
[2, 12000] loss: 0.068
[2, 14000] loss: 0.071
Finished Training
training time is 53.128
```

```
[1,  2000] loss: 1.880
[1,  4000] loss: 0.311
[1,  6000] loss: 0.194
[1,  8000] loss: 0.145
[1, 10000] loss: 0.128
[1, 12000] loss: 0.116
[1, 14000] loss: 0.104
[2,  2000] loss: 0.084
[2,  4000] loss: 0.084
[2,  6000] loss: 0.089
[2,  8000] loss: 0.073
[2, 10000] loss: 0.078
[2, 12000] loss: 0.071
[2, 14000] loss: 0.064
Finished Training
training time is 119.825
```

左边是使用 GPU 训练的结果，右边是 CPU 的结果，可见即使是在一个小网络上，GPU 的速度比 CPU 也有明显的加快。几次的训练结果显示 GPU 上面的训练时间只有 CPU 上的一半。

4　空间占用对比

GPU:

内存消耗:

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
14541 liujinh+  20   0 42.009g 2.135g  98388 S  14.6  1.7   0:02.79 ZMQbg/1
14539 liujinh+  20   0 42.009g 2.135g  98388 S  12.6  1.7   0:02.70 ZMQbg/1
14538 liujinh+  20   0 42.009g 2.135g  98408 S  12.3  1.7   0:02.74 ZMQbg/1
14540 liujinh+  20   0 42.009g 2.135g  98388 R  11.6  1.7   0:02.57 ZMQbg/1
```

显存消耗:

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 410.78       Driver Version: 410.78       CUDA Version: 10.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX 108...  Off  | 00000000:17:00.0 Off |                  N/A |
| 35%   59C    P2    59W / 250W |   1160MiB / 11178MiB |     18%      Default |
+-------------------------------+----------------------+----------------------+
|   1  GeForce GTX 108...  Off  | 00000000:65:00.0 Off |                  N/A |
| 23%   38C    P8     8W / 250W |     12MiB / 11178MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   2  GeForce GTX 108...  Off  | 00000000:66:00.0  On |                  N/A |
| 23%   35C    P8    10W / 250W |    141MiB / 11175MiB |     23%      Default |
+-------------------------------+----------------------+----------------------+
|   3  GeForce GTX 108...  Off  | 00000000:B3:00.0 Off |                  N/A |
| 23%   28C    P8     7W / 250W |     12MiB / 11178MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0     14603      C   ...e/liujinhui/.conda/envs/test/bin/python   617MiB |
|    2      1225      G   /usr/lib/xorg/Xorg                            88MiB |
|    2      2169      G   compiz                                        39MiB |
+-----------------------------------------------------------------------------+
```

CPU:

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 2815 yangdong  30  10  742380   9624   4368 R  12.3  0.0   3617:08 update-manager
21848 liujinh+  20   0 2832564 168612  19688 S   8.3  0.1   0:03.22 ZMQbg/1
21845 liujinh+  20   0 2832528 168596  19688 S   8.0  0.1   0:03.24 ZMQbg/1
21846 liujinh+  20   0 2832540 168564  19648 R   8.0  0.1   0:03.24 ZMQbg/1
21847 liujinh+  20   0 2832552 168608  19688 S   8.0  0.1   0:03.23 ZMQbg/1
 1225 root      20   0  377820  28476  23120 S   7.0  0.0   1881:08 Xorg
```

可以观察到：使用 GPU 进行训练的时候，内存的消耗为 2.13G 比使用 CPU 训练的 1.33G 要大，同时还伴随着显存的消耗。

5　结论

神经网络，尤其是卷积神经网络在 GPU 上进行训练可以极大的缩短训练时间，提高效率，但同时也对资源有更大的消耗。在具体任务时，可以使用其他方法优化代码从而降低内存和显存的消耗。在资源紧张的时候，小型任务可以使用 CPU 进行训练。