成　绩	
评卷人	

姓　名	刘静
学　号	2021123408

华 中 师 范 大 学

研 究 生 课 程 论 文

论文题目 基于 PyTorch 的 CBOW 模型实现

完成时间 2021.12.24

课程名称 数据工程

专　　业 电子信息

年　　级 2021 级

注：研究生须在规定期限内完成课程论文，并用 A4 页面打印，加此封面装订成册后，送交评审教师。教师应及时评定成绩，并至迟在下学期开学后两周内将此课程论文及成绩报告单一并交本单位研究生秘书存档。

1 概述

1.1 模型介绍

给定一段文本，CBOW 模型的基本思想是根据上下文对目标词进行预测。例如，对于文本…W_{t-2} W_{t-1} W_t W_{t+1} W_{t+2}…，CBOW 模型的任务是根据一定窗口大小内的上下文 C_t(若取窗口大小为 5,则 C_t={W_{t-2} W_{t-1} W_{t+1} W_{t+2}}对 t 时刻的词 W_t 进行预测。与神经网络语言模型不同，CBOW 模型不考虑上下文中单词的位置或者顺序，因此模型的输人实际上是一个“词袋”而非序列，这也是模型取名为“Continuous Bag-of-Words”的原因。但是，这并不意味着位置信息毫无用处。相关研究表明，融人相对位置信息之后所得到的词向量在语法相关的自然语言处理任务(如词性标注、依存句法分析)上表现更好。

CBOW 模型可以表示成图 1-1 所示的前馈神经网络结构。与一般的前馈神经网络相比，CBOW 模型的隐含层只是执行对词向量层取平均的操作，而没有线性变换以及非线性激活的过程。所以，也可以认为 CBOW 模型是没有隐含层的，这也是 CBOW 模型具有高训练效率的主要原因。

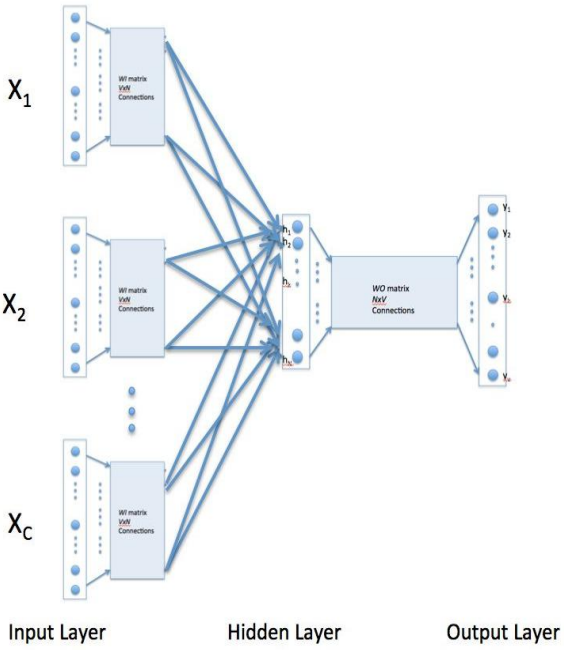

图 1-1CBOW 模型网络图

1.2 任务介绍

本文通过使用连续词袋模型将文本用词向量进行表示，如图 1-2，并能够通过上下文来预测中心词，同时将生成的词向量降维表示出来，如图 1-3。并使用 CPU、GPU、多 GPU 对连续词袋模型进行效率分析，比较单 CPU 下，不同的核心数对模型效率的影响，不同型号的 GPU 和多块 GPU 下的运行时间对比。

```
"mounting":[-1.5256633  -0.75024503 -0.653953   -1.609377   -0.10029095 -0.60908645
 -0.97972333 -1.6090509  -0.7120977   0.30361912 -0.77729297 -0.25161684
 -0.22244887  1.6870447   0.2284488   0.46762598 -0.6967907  -1.160759
  0.69947     0.19895515  0.86557657  0.24445455 -0.6628839   0.80737215
  1.1017736  -0.1757897  -2.2456117  -1.4463071   0.06120311 -0.61758256
 -0.79810053 -0.13158883  1.8794868  -0.07216933  0.15786035 -0.773529
  0.19914018  0.04573301  0.15303013 -0.47559166 -0.11100266  0.29288125
 -0.15794492 -0.0287563   2.3570426  -1.0372722   1.5749129  -0.62968713
 -0.9275109   0.5450214   0.06626818 -0.4370827   0.762685    0.44159746
  1.1650484   2.0154314   0.13727811  0.9386483  -0.18607156 -0.6445348
  1.5394049  -0.86946464 -3.3314412  -0.7479015  -0.02563655 -1.0232844
 -0.5961082  -1.0056671  -0.21071379 -0.00761244  1.6734102   0.01043226
 -0.7040515  -0.18529747 -0.99623156 -0.8312991  -0.461171   -0.56001395
  0.3955271  -0.9822183  -0.50637716  0.09981017 -0.65385664  0.731825
 -1.4342958  -0.50091994  0.17172273 -0.1597909   0.25445345 -0.50199413
 -1.0411772   0.7323627  -1.0483025  -0.47077322  0.2909607   1.9908239
  0.6616346   1.1900604   0.8166158  -0.91343355]
"problematic":[ 1.3851231  -0.81382656 -0.92755073  1.1119175   1.3352106   0.6043528
 -0.1034507  -0.1512433  -2.102148   -0.6200653  -1.4781808  -1.1333437
  0.87374485 -0.5602999   1.2858806   0.81683916  0.20534344  0.3050872
  0.5357779  -0.43107408  2.558101   -0.23335445 -0.01346042  1.8606136
 -1.9803381   1.7985694   0.10181426  0.34000573  0.71244556 -1.7764769
  0.35391122  1.1996859  -0.30302432 -1.7617657   0.6349164  -0.80443126
 -1.6110338  -1.871546    0.5430905   0.66064954  2.2951314   0.674784
  1.713353   -1.7942044  -1.36322    -0.98324627  1.5112929   0.641761
  0.4729262  -0.42869228  0.5512986  -1.5474108   0.5181322   0.106617
  0.26924065  1.3246522   1.7460485   1.8549794  -0.7062389   2.556992
```


图 1-2 词向量表示

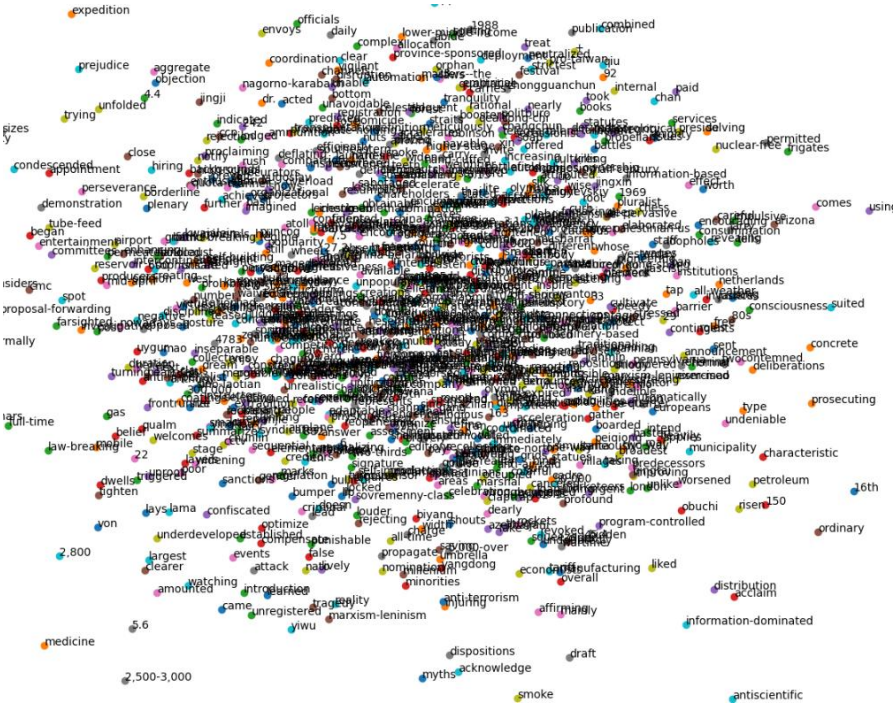

图 1-3 二维可视化图

2 实验方案

2.1 CPU运行

分别采用单核、2 核、3 核、4 核、5 核、6 核下的 CPU 进行对比，通过修改以下配置实现。

```
cpu_num = os.cpu_count() #  自动获取最大核心数目
# cpu_num = 4

os.environ ['OMP_NUM_THREADS'] = str(cpu_num)
os.environ ['OPENBLAS_NUM_THREADS'] = str(cpu_num)
os.environ ['MKL_NUM_THREADS'] = str(cpu_num)
os.environ ['VECLIB_MAXIMUM_THREADS'] = str(cpu_num)
os.environ ['NUMEXPR_NUM_THREADS'] = str(cpu_num)
torch.set_num_threads(cpu_num)
```

2.2 GPU运行

设置 device 参数，进行 CPU，GPU 之间的切换

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = CBOW(vocab_size, embedding_dim).to(device)
```

多 GPU 运行通过设置 DP 模式：

```
gpus=[0, 1]
if torch.cuda.device_count() > 1:
    model = nn.DataParallel(model, device_ids=gpus, output_device=gpus)
```

DDP 模式设置：

```
model = torch.nn.parallel.DistributedDataParallel(
        model,
        device_ids=gpus,
        output_device=gpus
        )
```

3 实验结果及分析

3.1 实验数据

实验数据由 10 万个英文句子组成，如图 3-1 所示。

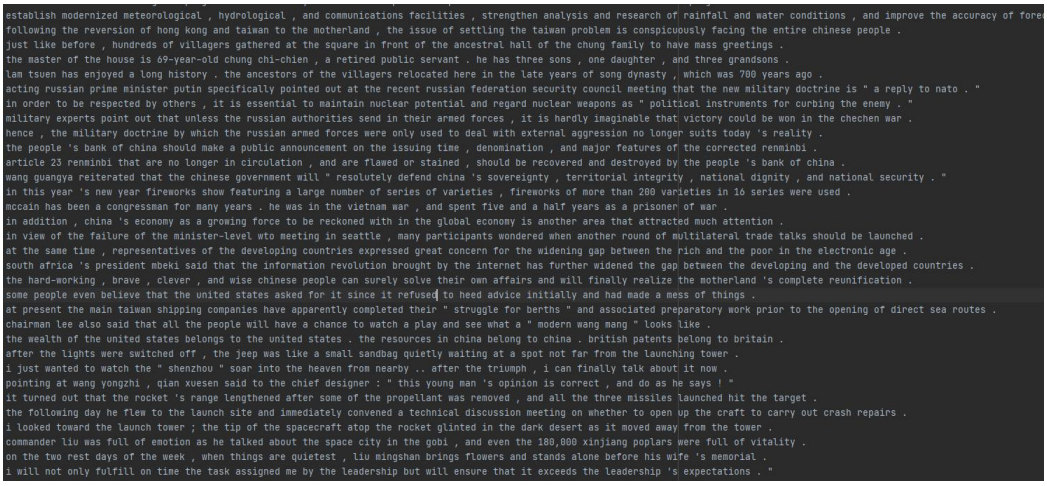

图 3-1 实验数据

3.2 CPU实现

通过使用不同核心数下的 CPU 来运行项目，并比较核心数对实验结果的影响。

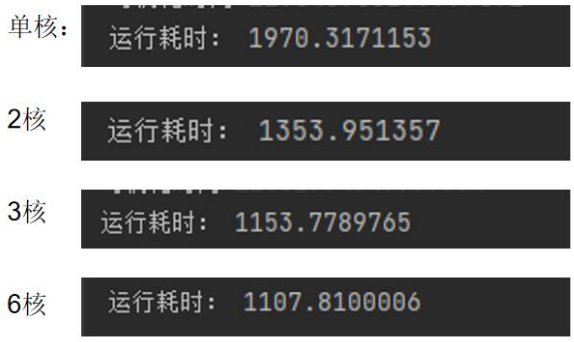

图 3-2 不同核心数下的运行时间

通过柱状图更加清晰的看出，CPU 核心数越多时，运行时间越短，但是并不是那么绝对的，通过图 3-2 中数据比较，当使用 3 个核心时，程序运行时间为 1153s，但是当核心数增加到 8 个时，程序运行时间是 1107s，并没有提升太多。

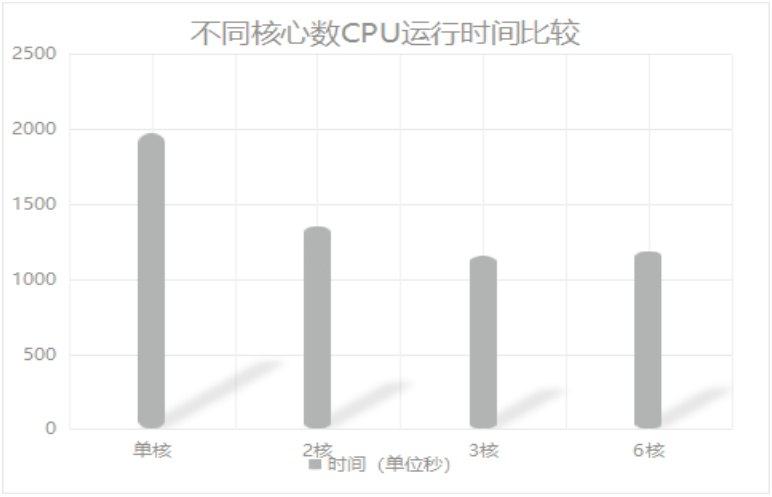

图 3-3 运行时间柱状图

CPU 利用率也是和核心数有关系，由于电脑配置是八个核心的 CPU，所以当核心数设置为 8 时，CPU 利用率时 100%，同时 CPU 利用率也是随着核心数增加而增加的。

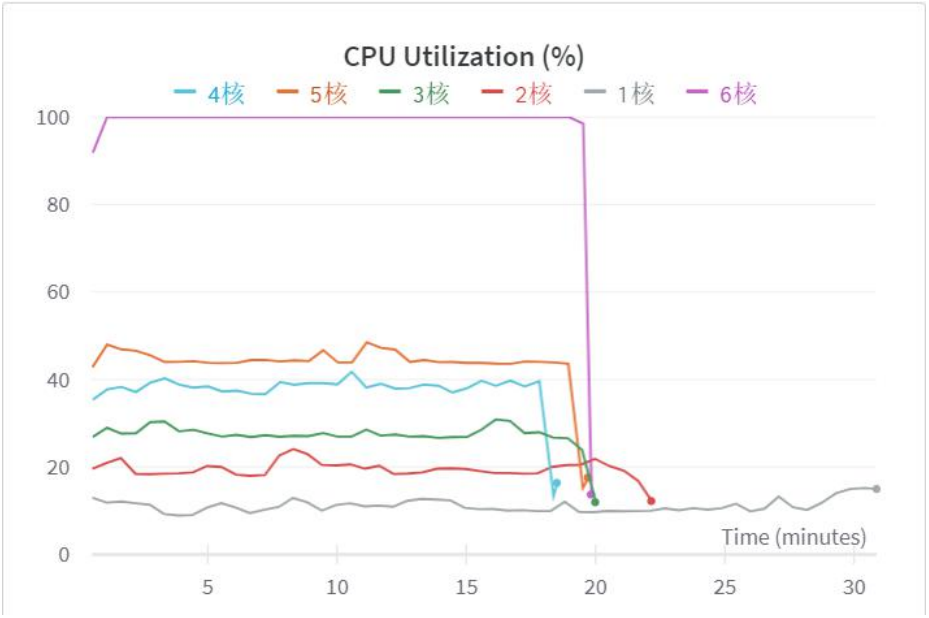

图 3-3 CPU 利用率

3.3 单GPU实现

通过对不同型号 GPU 的使用运行，来比较不同型号和配置的 GPU 对程序运行的影响，实验主要采用 Tesla K80、GeForce RTX 2080 Ti 和 GeForce RTX 3090 来进行比较。下图 3-4 为运行时间截图。

Tesla K80

```
训练测试时间 768.9027204755694
==================4.输出处理=======================
词向量已保存
输出处理时间 96.64797250274569
==================5.可视化阶段=======================
可视化时间 30.335869388654828
运行耗时： 895.8868684954941
Tesla K80
```

GeForce RTX 2080 Ti

```
训练测试时间 501.5151565745473
==================4.输出处理=======================
词向量已保存
输出处理时间 63.37182877212763
==================5.可视化阶段=======================
可视化时间 18.469243749976158
运行耗时： 583.3566087186337
GeForce RTX 2080 Ti
```

GeForce RTX 3090

```
训练测试时间 287.94343576952815
==================4.输出处理=======================
词向量已保存
输出处理时间 56.06124426051974
==================5.可视化阶段=======================
可视化时间 21.715210931375623
运行耗时： 365.7200530786067
```

图 3-4 GPU 运行时间

接下来使用柱状图的形式对不同型号 GPU 的运行时间进行比较，可以清楚的看出，GPU 配置越高，程序运行时间越短，使用 K80 的训练时间为 768s，使用 2080 Ti 训练时间为 501s，3090 的训练时间为 287s，使用 3090 比 2080 Ti 快了大概一倍左右，而且使用 GPU 进行加速，加速时间也是 CPU 的几倍。

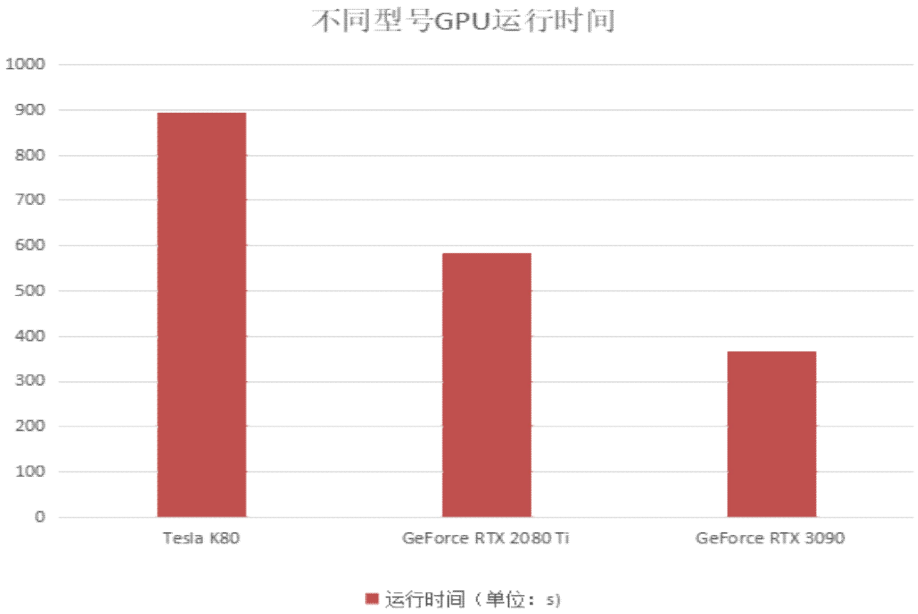

图 3-5 GPU 运行时间柱状图

使用 wandb 工具来使运行期间资源使用情况可视化，可以看到如图 3-6 所示的 GPU 利用率情况，不管是使用什么型号的 GPU，GPU 利用率都不高，而且 GPU 配置越高，利用率越低。

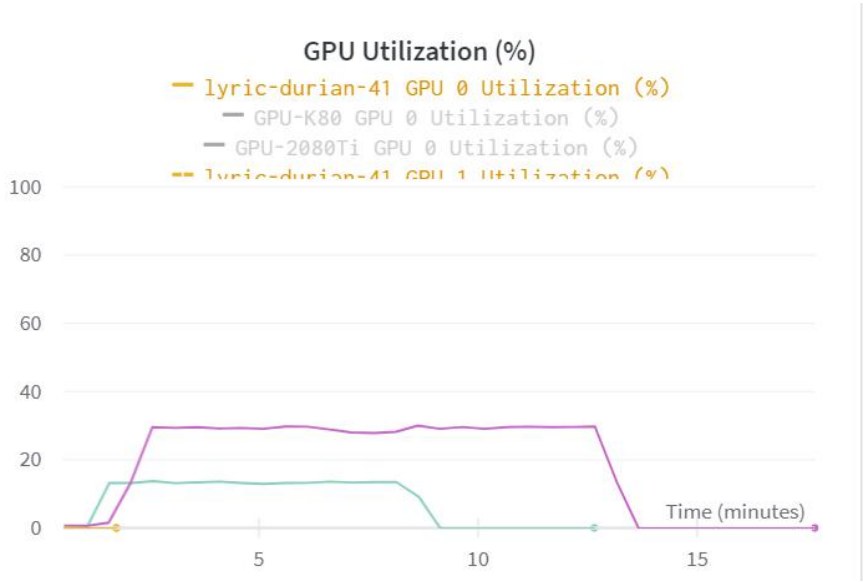

图 3-6 GPU 利用率

3.4 多GPU实现

多个 GPU 的实现方式主要有两个，一个是 DataParallel (DP)的方式，一个是 DistributedDataParallel (DDP)的方式，DP 的方式只能在单进程情况下使用，只能在单机运行，而 DDP 的方式可以在多进程的情况下使用，且在单机和分布式的训练中都可以使用。

首先采用 DP 的方式进行运行，图 3-7 是通过 nvidia-smi 查看服务器中 GPU 的使用情况，可以看到，在第 0 号 GPU 中，只有一个程序在运行，但 GPU 利用率才 22%，也没有充分利用起来。

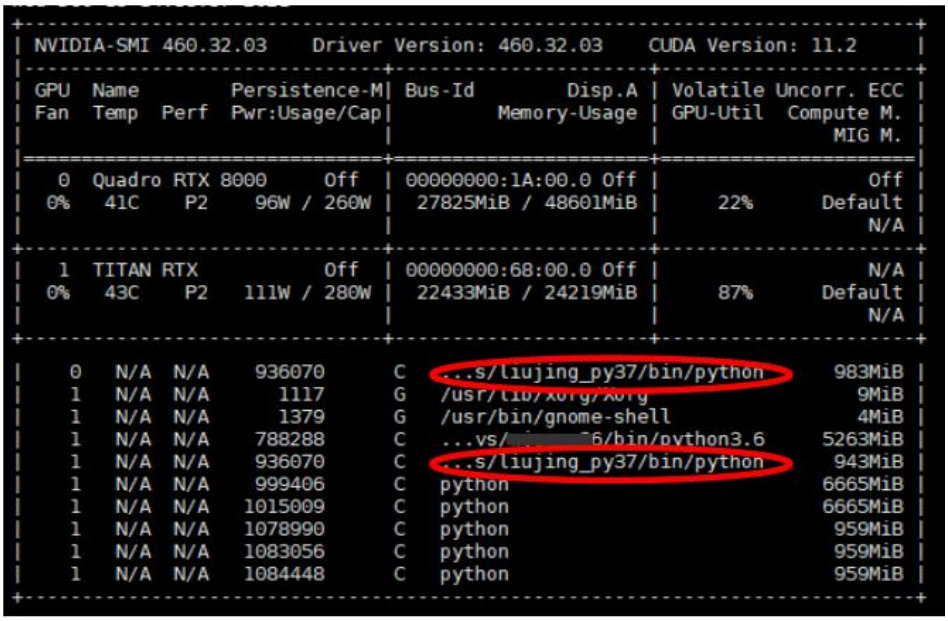

图 3-7 DP 方式下 GPU 运行情况

然后是通过 DDP 的方式来运行，通过命令查看如图 3-8 所示，但是由于多人共同使用 GPU 所以无法确定该模型真正的 GPU 利用率，于是重新运行一次通过可视化工具来查看，从图 3-9 可以看到，GPU 0 的利用率为 9%左右，GPU 1 利用率为 13%，利用率变的更低，而且运行时间也没有比单 GPU 少，反而有所增加。

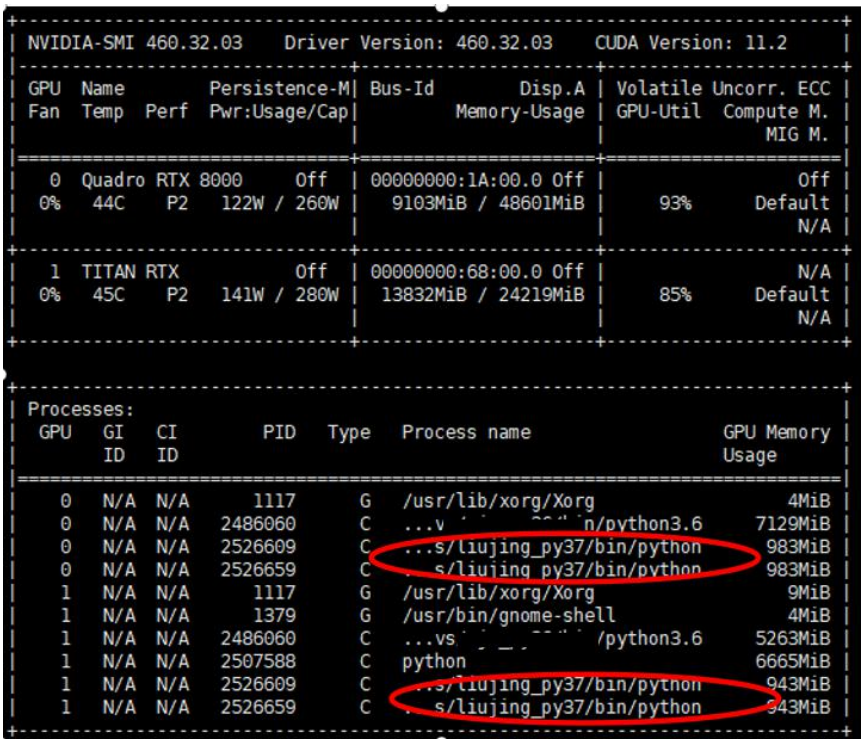

图 3-8 DDP 方式下 GPU 运行情况

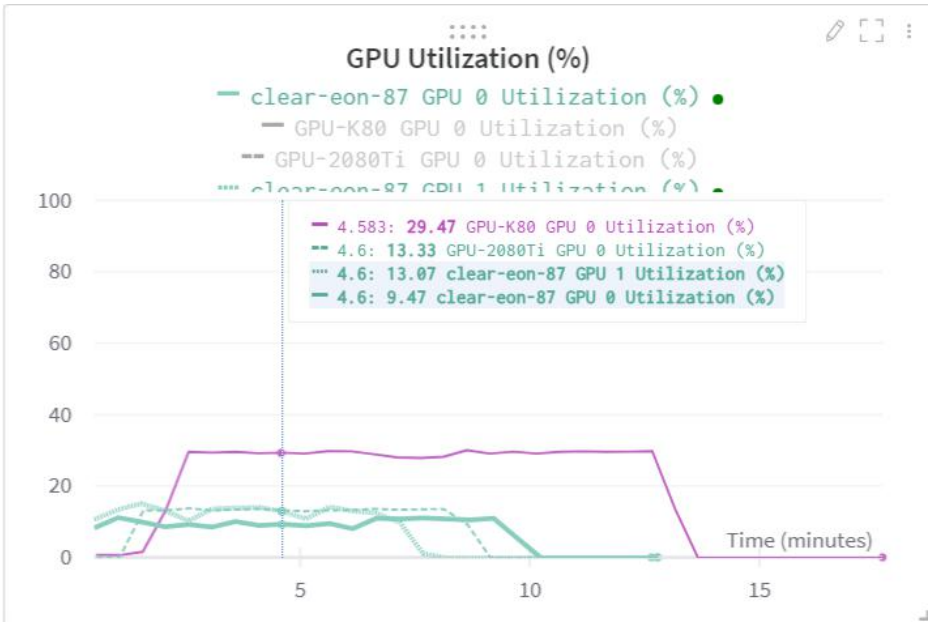

图 3-9 GPU 利用率

4 总　结

1、在程序运行中，一般使用 CPU 进行数据的读取和预处理，而使用 GPU 进行模型的正向传播和反向传播。由于 CPU 数据读取跟不上（读到内存+多线程+二进制文件），而 GPU 的处理速度太快，导致 GPU 的利用率不高。可以通过关闭日志文件，减少日志的 IO 操作频率，NVIDIA 提供了 DALI 库可以将数据处理转移到 GPU 上。

2、模型太小，数据集也不是很大的时候，更加推荐使用单 CPU 或者单 GPU，减少一些传输损失。CBOW 是一个只有一个隐藏层的全连接神经网络结构，操作较为简单，如果利用多 GPU 开销更大，效率也会更低。

3、在进行效率对比时，要保证唯一变量，尽量使用相同型号的机器，否则运行结果有可能造成很大的影响。