

# task2 report

张艺洋

January 2026

## 1 简介

任务 2 使用和任务 1 一样的神经网络结构，在同样的设备上，使用 DP 和 DDP 方法进行多 GPU 数据并行的模型训练。本报告是一些实验结果和相关讨论。

## 2 网络结构

在任务 2 训练中我使用的网络结构如下，该结构任务 1 中的是完全一致的。

表 1: LeNet 网络结构

层类型	输出尺寸	参数	备注
Conv1 + BN1 + ReLU	$6 \times 28 \times 28$	$3 \times 6 \times 5 \times 5 + 6$	卷积核 $5 \times 5$ , BN
MaxPool1	$6 \times 14 \times 14$	-	$2 \times 2$ 池化
Conv2 + BN2 + ReLU	$16 \times 10 \times 10$	$6 \times 16 \times 5 \times 5 + 16$	卷积核 $5 \times 5$ , BN
MaxPool2	$16 \times 5 \times 5$	-	$2 \times 2$ 池化
Flatten	400	-	展平为向量
FC1 + BN3 + ReLU	120	$400 \times 120 + 120$	全连接, BN
FC2 + BN4 + ReLU	84	$120 \times 84 + 84$	全连接, BN
FC3	10	$84 \times 10 + 10$	输出层

## 3 实验结果

本实验使用的设备都是从 AutoDL 平台上面租用的 RTX 4090(24GB) 型号的 GPU，共 4 张卡并行进行训练。

超参数与任务 1 相同，即 batch\_size=64, num\_workers=4, epochs=10, lr=0.001, momentum=0.9，使用 SGD+momentum 优化器训练 10 个 epoch。

### 3.1 DP

根据 Pytorch 官方文档 [2] 进行 Data parallel 的训练。基本的代码和任务 1 相同，在生成一个 Lenet 模型之后，只需检测当前设备的 GPU 数量，并且使用 Pytorch 专门的接口给 model 设置数据并行即可，核心代码如下。

```
model = LeNet()
if torch.cuda.device_count() > 1:
    model = nn.DataParallel(model)
```

命令行写下 `python dp.py --mode train`(后面可以添加其他超参数, 具体请查看代码), 运行训练过程, 得到的平均每 epoch 训练时间为 25.3s, 训练的 loss 曲线如图1。

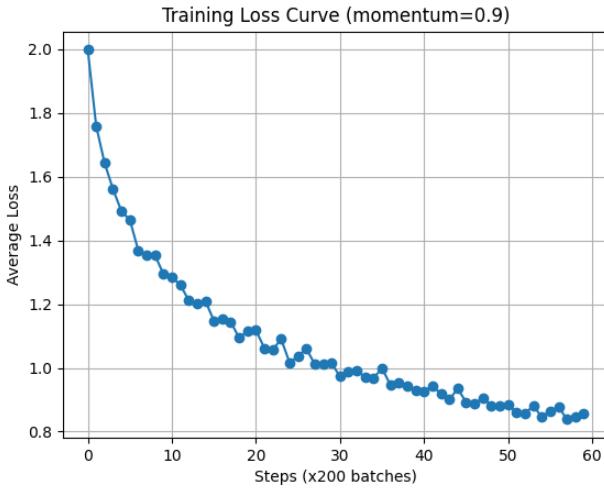


图 1: DP loss 曲线

在 Cifar10 的测试集上面运行, 模型的训练准确率为 66.80 %。

### 3.2 DDP

相比于 DP, DDP 有着更高的并行度。每个 GPU 都单独拿到一部分不重复数据并且计算自己的梯度, 随后通过 GPU 通信进行 all-reduce 求各个 GPU 的梯度平均, 最后对各个 GPU 上面的模型参数进行梯度下降。具体的代码可以参照 `ddp.py` 中的注释。

命令行写下 `torchrun --nproc_per_node=4 ddp.py --mode train --batch_size 64 --epochs 80`。由于实际训练因素我不得不把 epoch 调整到 80 个回合。平均每 epoch 训练时间为 1.80s, 训练的 loss 曲线如图2。

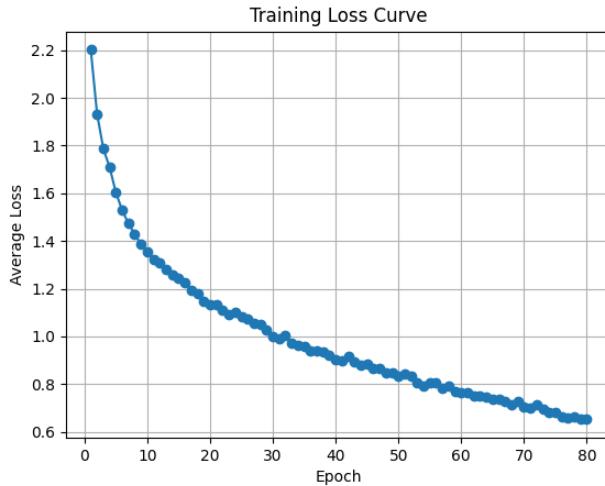


图 2: DDP loss 曲线

在 Cifar10 测试集上面运行, 模型的训练准确率为 64.16%。

## 4 讨论与对比分析

在同样的设备下，使用三种不同的方式进行训练，出现了不同的效果。下面我来简单分析一下。首先我认为这

表 2: 不同训练方式数据对比

训练方式	每 epoch 时间	准确率
单卡	3.29s	65.39%
4 卡 DP	25.3s	66.80%
4 卡 DDP	1.80s	64.16%

三种训练方式在最终的准确率上差别不大，很有可能是误差，因此可以视为三种训练方式的效果是相近的。我将主要讨论运行效率。

### 4.1 讨论 1

首先是最引人注目的 4 卡 DP 的运行时间。在实验所用的简单模型上，这个运行时间堪称和 CPU 有一拼了。

通过查阅相关文章 [1], [3]，可以发现 DP 的运行模式是存在较为严重的缺陷的：因为它本质上还是 CPU 的单进程模式，只不过是使用了 GPU 的多线程。因此在所有 GPU 中必定存在一个主 GPU0 进行数据的汇总和分发，这一点严重影响了 DP 的运行速度。

首先每个训练 epoch 开始前，GPU0 需要把自己上面的参数复制分发给其它 GPU，这种显存拷贝的过程带来的通信成本会严重影响模型的训练速度。

同时在每个 epoch 反向传播结束，optimizer 需要进行 step 更新的时候，GPU0 需要汇总所有 GPU 算出的梯度，然后在自己的参数上进行梯度下降。这个汇总的过程也会带来严重的通信成本，GPU0 必须等待所有 GPU 都算完之后才能开始自己的更新运算。

综上，DP 不仅面临着严重的通信成本过高问题，GPU0 由于同时肩负着分发数据、计算、汇总数据的任务，它的任务量过大了，因此每一轮训练时候的瓶颈也恰恰在 GPU0 上，只有等它计算完，才能开始下一轮训练。在 Lenet 这样的小模型上，由于计算成本相对较低，因此使用 DP 造成了本实验中严重的效率低下问题。这注定了 DP 终究只能是一个教学 demo 级别的并行方式。

### 4.2 讨论 2

说完 DP，不得不说下 DDP。本次实验中 DDP 还是跑出了预期的效果的，相比于单卡有 2 倍的提速，这在 Lenet 这种小模型上是可以接受的，毕竟因为通信问题注定无法达到 4 倍速的效果。

那么，同样是数据并行，为什么 DDP 的运行时间要远低于 DP 呢？

在 DDP 中，各个 GPU 的地位是平等的。Pytorch 对于 DDP 支持的 sampler 能够向各个 GPU 分发不重复的训练数据，每个 GPU 只需在自己的数据上进行前向计算和反向传播就可以了。

而在最终的参数更新时，DDP 并不需要像 DP 一样进行汇总，而是利用 NVLink 的 GPU 通信机制，使用 reduce 技术（这学期课上讲过，本身也是一种并行高效的求和方式），比如时间复杂度为  $O(g)$  的 ring-reduce 进行环形数据传播，边计算边通信，规避了 DP 的最大缺陷，使得 DDP 成为了主流的多机多卡数据并行方案。

### 4.3 讨论 3

当然在本次实验中我也有一点失误，就是我在 DDP 运行的时候不得不让它跑了 80 个 epoch。我一开始无法理解为什么 DDP 的收敛速度慢，后来才发现问题所在：由于 DDP 本质上使用的 batch\_size 是单卡的四倍，然而我没有调节学习率，因此导致 DDP 的学习速度较慢，没有很快收敛。但是这也正告诉了我，面对更大规模的数据时候，使用 DDP 可以实现更大 batch\_size 的训练，用空间换时间 (bushi)。

## 参考文献

- [1] Anthony Peng. Multi-gpu training in pytorch with code (part 2): Data parallel, 2023. URL <https://medium.com/polo-club-of-data-science/multi-gpu-training-in-pytorch-with-code-part-2-data-parallel-21b5b8df4ef>.
- [2] PyTorch Contributors. Optional: Data parallelism — pytorch tutorials, 2024. URL [https://docs.pytorch.org/tutorials/beginner/blitz/data\\_parallel\\_tutorial.html](https://docs.pytorch.org/tutorials/beginner/blitz/data_parallel_tutorial.html).
- [3] Suraj Subramanian. What is distributed data parallel (ddp), 2024. URL [https://docs.pytorch.org/tutorials/beginner/ddp\\_series\\_theory.html#why-you-should-prefer-ddp-over-dataparallel-dp](https://docs.pytorch.org/tutorials/beginner/ddp_series_theory.html#why-you-should-prefer-ddp-over-dataparallel-dp).