西安電子科技大學

高性能智能计算实验

Relation-Network 基于单机多卡的并行加速

姓名： 刘佳玮

学号： 20031211496

学院： 计算机科学与技术学院

专业： 电子信息

2020 年 11 月 25 日

摘要

针对小样本问题，首先构建深层残差网络来实现 Relation Network 模型，并在 miniImageNet 数据集中实现小样本的分类，记录分类的准确率。以此为目标，进行并行加速实验的设计与实现。

首先给出实验环境，包括：1.CPU、GPU 和内存等硬件设备信息；2. 操作系统和开发工具等软件及版本，并简述了各个软件负责的功能；3. 实验所需的实验参数及取值。而后分别从数据并行的方法、并行加载数据的角度出发，阐述了本次实验中并行算法的设计思想，并对并行实现的原理进行了解析，包括多进程的工作流程和通信方法等。

而后针对并行加载数据进行对比实验，列出了在不同数量进程下的加速比与并行效率，对最终结果进行分析并得到了在进程数和处理器核心数近似相等的情况下，加速比效果较为良好的结论；针对 DataParallel 和 DistributedDataParallel 两种不同的并行方式，进行了对比实验后取得加速比，并结合对并行原理的解析，得到了 DistributedDataParallel 优于 DataParallel 的结论。之后更改实验参数，分析实验在不同参数下取得的结果并得到：并行方法的适用场合和实验参数如何取值等相关结论。

最后，在附录中列出了实验过程中遇到的相关问题和对应的解决方案，并给出实验所用的源程序。

关键词：深度学习，单机多卡，数据并行

目录

1 问题背景

在这一章节，主要介绍所需加速的神经网络、此类网络在实际问题中的应用和选用的数据集。

1.1 Relation-Network 介绍

自神经网络迅速发展以来，主要用于解决分类和回归问题，并且已经成功应用到图像、NLP 和语音等多个领域。在此期间，各种类型的网络层出不穷，如：LSTM，CNN 和 GCN 等。

但对于任何神经网络，其准确率和性能都会受到数据数量的影响。当训练数据很少时，网络的准确率必然会有所下降，而此类问题被称为小样本问题。针对小样本问题，提出了一种新的网络结构：Relation-Network，采用基于度量的方式来解决小样本问题 [1]。

首先，网络读入支持集中的多组图片，经过特征提取层 f_ϕ 得到图片的特征表示 E_1。而后读入查询集中的图片，查询集中的图片类别属于支持集多组图片中的某一类。经过同样的特征提取层（参数一致）f_ϕ 得到另外的特征表示 E_2。将查询集的特征表示 E_2 和支持集的特征表示 E_1 拼接到一起，再次经过另外的特征提取层 g_ϕ 得到综合的特征表示 E_3，最后经过分类层后得到特征向量 Relation Score。在 Relatoin Score 中，查询集所属的类别的维度得分较高，其他维度得分较低，以此为误差进行反向传播，模型结构示意图如图 1所示。

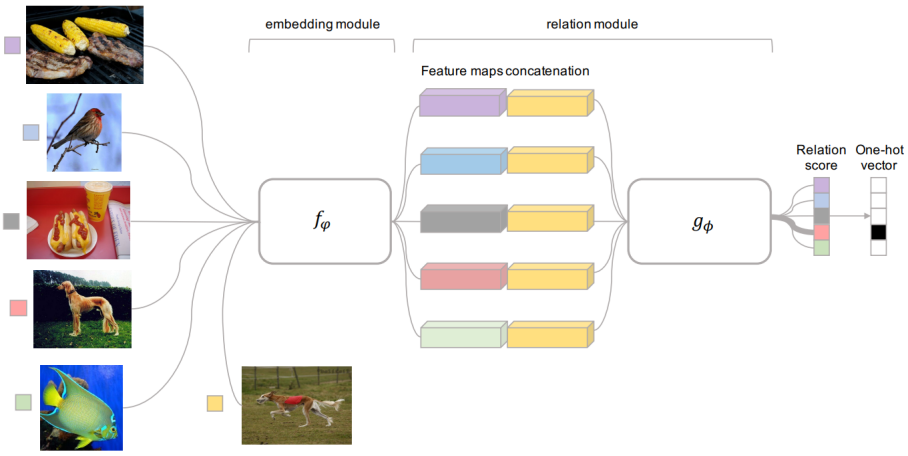

图 1: Relation-Network 网络结构示意图

而选用此网络进行实验的目的是，查询集与支持集的拼接过程难度较大且无法并行化，来观察在并行受限的情况下是否仍然具有较高的加速比。

1.2 数据集与选取方法

miniImageNet[1] 数据集含有 100 个类，每个类有 600 张三通道彩色图片，图片的尺寸是 84×84，共。因其类别数量较多，所以 miniImageNet 数据集被广泛应用在小样本分类问题中。miniImageNet 数据集中的图片实例如图 2所示。

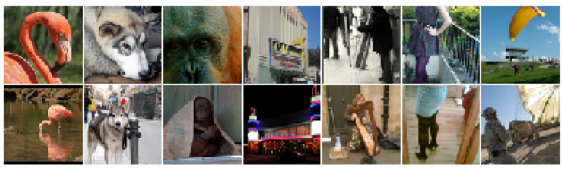

图 2: miniImageNet 数据集中的图片实例

按照小样本问题的标准，网络训练时使用的每组数据由支持集和查询集组成。n_way 表示一组支持集含有类的数量，k_query 表示查询集中含有的类的数量，k_shot 表示支持集与查询集中每个类有几个样本。

1.3 模型结构

因为数据集为彩色图片，所以采用以卷积为主要形式来处理图片，并借助残差块来搭建深层神经网络 [2]。最终实现的神经网络中的一个模块包括：卷积层、标准化层和激活函数，以及对输入进行下采样后得到的残差块。模块结构如图3所示：

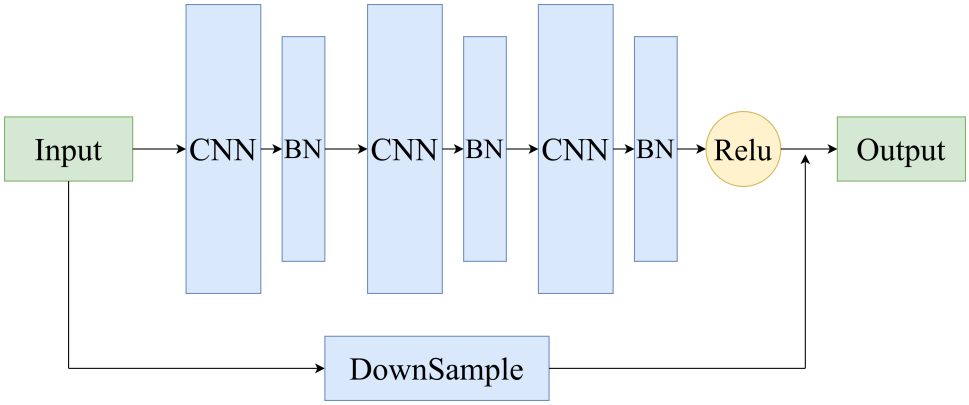

图 3: 神经网络中模块结构示意图，CNN 表示卷积层，BN 表示标准化层，DownSample 表示对输入进行下采样，Relu 表示激活函数，Input 表示模块的输入，Output 表示模块的输出

[1] https://drive.google.com/file/d/0B3Irx3uQNoBMQ1FlNXJsZUdYWEE/view

f_ϕ 有 13 组这样的模块，g_ϕ 有 7 组这样的模块，在网络的最后有一组全连接层，用于计算查询集的得分：Relation Score。

2 实验环境

实验的成功进行离不开软件件配置，本章分别介绍实验所依赖的的硬件设备和开发环境及版本，之后介绍本次实验所选用的参数。

2.1 硬件设备

1. CPU：2 个 Intel(R) Xeon(R) Gold 5115 CPU 2.40GHz，10 核心 20 线程。
2. GPU：4 路 Tesla P40，每路显存容量 22GB，共 88GB。
3. 内存：128GB。
4. 外存：520TB 可用，已用 15TB。

2.2 软件环境与版本

1. 运行程序的系统为 CentOS Linux release 7.3.1611。
2. 开发程序的系统为 Arch 5.9.6。
3. python：3.8.2：作为开发语言。
4. pytorch：1.6.0：作为模型实现的工具，借助其提供的 API 实现并行。
5. ssh：OpenSSH_8.3p1, OpenSSL 1.1.1h，实现远程登录功能，即本地登录到服务器。
6. scp：实现文件传输功能，将本地文件传输至服务器，或将服务器文件传输至本地。

在进入服务器后，不能在根目录下直接配置实验环境，否则第三方库之间的版本依赖、冲突等问题会对他人的程序造成一定影响。因此，首先使用 conda 创建虚拟环境，在 conda activate 进入虚拟环境后配置本次实验所需的依赖库。对于本实验，进入虚拟环境后 pip install torch torchvision 即可。

2.3 参数设置

1. n_way = 5, 5 个类为一组支持集。
2. k_query = 1，查询集只含有 1 个类。
3. k_shot = 1，支持集和查询集中每个类中只有一个样本。
4. batchsz = 30，每一组训练的数据中由 30 组支持集和查询集组成。

在不同的对比实验中，除需对比的参数外，其余实验参数需要保持一致，否则程序的执行时间会相差很远，无法得到准确的加速比。

3 并行思路与方法

在这一章，将按照并行加载数据、不同的并行方式这两个角度来阐述并行算法的思路与设计方法。并行训练模型可分为两种：数据并行和模型并行。数据并行是指：多个 GPU 使用相同的模型副本，但采用不同的数据进行训练 [3]；模型并行是指：当模型过大时，多个 GPU 使用相同的数据分别训练模型的不同部分。因实验选用的 GPU 足以容纳模型和数据，因此采用数据并行的形式来加速网络的执行。

3.1 并行加载数据

在加载数据时，可以使用多进程来加载数据。不使用多线程的目的是：python 的多线程因为 GIL（Global Interpreter Lock）的存在，只能在单核运行。不能发挥多核处理器的优势，因此只适合 I/O 型任务，不适合密集计算型任务。

首先以 torch.utils.data.Dataset 为父类，创建一个加载数据集的子类：MiniImagenet。在 MiniImagenet 中，设置 __getitem__ 方法来支持数据集的切片访问，即批量加载数据。

在训练阶段，实例化加载数据集的类，得到一个实例对象。在 torch.utils.data.DataLoader 中传入实例对象来加载数据，并设置其他参数来提升加载数据的速度和数量。设置 num_workers=8 可以使用 8 进程来加载数据，设置 batch_size=batchsz 对应 MiniImagenet 类的 __getitem__ 切片访问，即一次获取的训练数据的大小。

在创建 num_workers 个进程之后，多个进程共享一份数据集，将指定 batch（一批待加载的目标数据）分配给指定 worker（一个进程），worker 将对应 batch 的数据加载进内存。因此增大 num_workers 的数量，内存的占用率也会增加。在第一个 batch 的数据加载完成后，会等待主进程将该 batch 取走并汇总，而后此 worker 开始加载下一个 batch，不断迭代。

主进程采集完最后一个 worker 的数据后，此时需要返回并采集第一个 worker 加载的第二个 batch。如果第一个 worker 此时没有加载完，主线程将阻塞等待当前 worker 的数据加载完毕。最后所有数据加载完毕后，汇总到一起供用户使用。

当 num_worker 设置的很大时，优点时可以快速的寻找目标 batch；缺点是内存开销大，也加重了 CPU 维护进程的开销，如创建、初始化、通信、分配任务、接受任务反馈和销毁进程等 [4]。num_workers 的经验设置值是服务器的 CPU 核

心数，如果 CPU 性能强大且内存充足，可以设置为较大的数值。

3.2 DataParallel 并行方式

DataParallel 并行方式实现较为简单。首先将模型放到主 GPU 中，一般是 GPU-0；而后将模型复制得到副本，将模型副本放到另外的 n 张 GPU 中，将输入的 batchsz 大小的数据平均分为 n 份依次作为每个模型副本的输入。因此要求 batchsz 的大小要大于 GPU 的数量，否则数据无法分配，会导致有的模型副本得不到数据。

每个模型副本独立的进行前向计算，得到各自的 loss 值。在 n 个 GPU 完成计算后，将 loss 汇总到 GPU-0 中，由 GPU-0 中的模型进行反向传播和更新参数。更新好参数后，由 GPU-0 将参数分发至其他 n 个 GPU 卡，开始新一轮的计算。DataParallel 并行结构示意图如图 4所示：

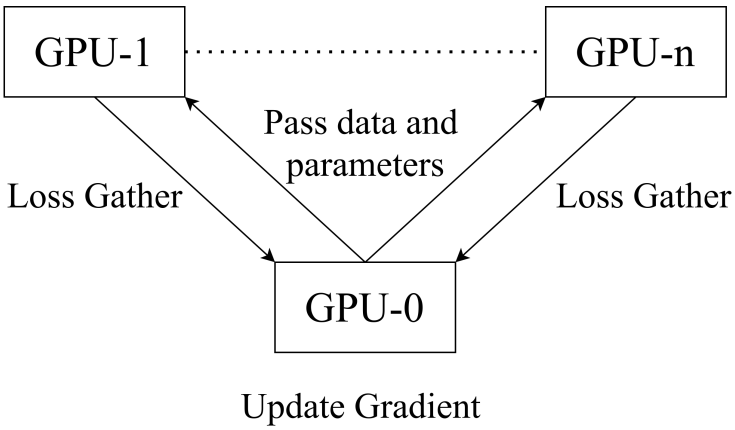

图 4: DataParallel 并行结构示意图

DataParallel 基于单进程多线程实现，因为 python 的多线程无法利用多核处理器，只能在单核上进行任务调度和参数更新的通信，所以此种方式并行效率较低，且 0 号卡负担较大。

3.3 DistributedDataParallel 并行方式

DistributedDataParallel 并行方式实现较为复杂，且相对于 DataParallel 需要更大的显存空间。DistributedDataParallel 使用的进程数量一般和 GPU 数量相等，对每个进程创建一个 DistributedDataParallel 实例，通过进程通信来同步梯度，通信方式为 AllReduce[5]。

在任务初始化阶段，即并行程序执行之前首先要创建通信群组，通信群组中的每一个进程创建一个 DistributedDataParallel 实例。DistributedDataParallel 通过

rank0（进程 ID）将当前模型的结构和参数等状态通过广播通信的形式发送到进程组的所有进程中，确保其他进程中的模型副本和当前模型保持一致。而后，每个进程创建一个本地的 Reducer，用于反向传播阶段同步参数的梯度。因此相比于 DataParallel，DistributedDataParallel 需要更大的显存容量。为了提升通信效率，Reducer 将参数放到多个桶中，每次以桶为单位进行通信 [6]。

参数需要额外附加 unready 和 ready 两种状态，默认为 unready 状态。若参数在反向传播阶段计算了梯度，则参数由 unready 状态变为 ready 状态，表示这个参数可以同步到其他模型；否则保持 unready 状态。当模型完成一次训练后，参数由 ready 状态变为 unready 状态，为下一次训练做准备。

因为在反向传播阶段，DistributedDataParallel 只会等待 unready 状态的参数更新，所以在前向计算阶段首先要分析计算图，将不进行梯度更新的参数状态永久设为 ready 状态，防止这些参数影响反向传播。DistributedDataParallel 在获取输入数据后，将输入数据分发到每个 GPU 中，模型副本开始前向计算得到各自的 loss 值，而后开始反向传播。

DistributedDataParallel 在初始化阶段会为每一个可求导的参数申请一个自动求导的钩子，来发射同步梯度的信号。在反向传播阶段，当一个参数计算梯度后会为 ready 状态，对应梯度的钩子就会发射信号，DistributedDataParallel 会标记该参数的状态，表示该参数的梯度可以用于同步。当一个桶内的参数全部由 unready 状态变为 ready 状态后，Reducer 通过 AllReduce 的通信方式来更新桶内的参数，来计算所有模型副本中参数梯度的平均值。以 4 个 GPU 为例，其 AllReduce 通信方式如图 5所示。

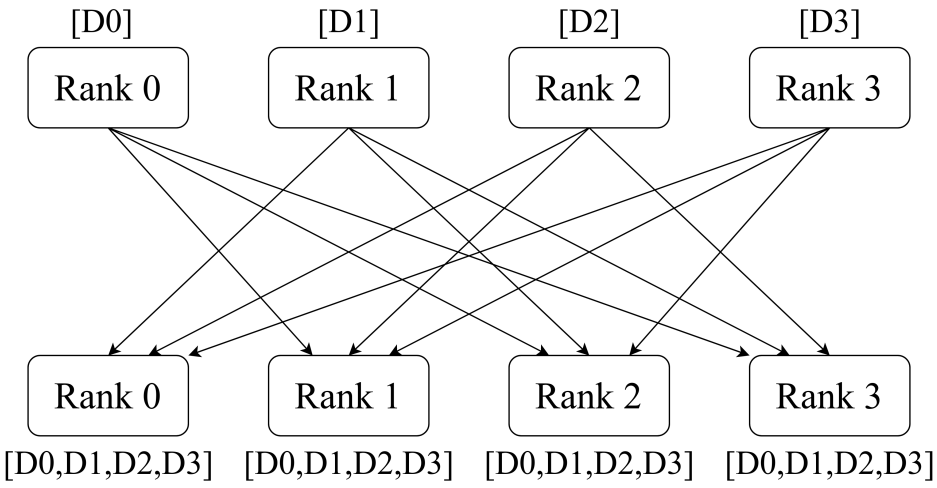

图 5: AllReduce 通信方法示意图

在图 5中，rank0 将自己模型副本的梯度 D0 发送给 rank1，rank2 和 rank3，其他进程同理。因此在 AllReduce 通信后，所有的模型副本的梯度值保持一致。

当所有的桶更新完毕后，Reducer 会阻塞等待 AllReduce 的所有操作结束，如重新将参数设为 unready 状态。因为所有模型副本的梯度值相同，所以在梯度更新阶段，只需更新本卡中的模型参数。因为所有的模型副本来自相同的初始状态，且具有相同的平均梯度值，所以在梯度更新后所的模型副本状态都会保持一致，以此来实现并行。

3.4 并行方式的对比

DataParallel 和 DistributedDataParallel 都属于数据并行的方法，在分析完各自的实现原理后，可以得到二者的对比。对比情况如表 1所示：

表 1: DataParallel 与 DistributedDataParallel 对比

	DataParallel	DistributedParallel
机器支持	只支持单机	可支持多机
模型支持	不支持拆分模型	可以将模型放到多张 GPU 中
通信方式	scatter input 和 gather output	scatter input 和 AllReduce
并行支持	只支持数据并行	支持数据并行和模型并行
进程与线程	单进程、多线程	多进程

4 实验结果与分析

在这一章将列出并行加载数据、不同并行方式下实验的耗时结果。使用 python 的 time 库记录耗时，并对结果进行探讨和分析。数据集的规模是 60000 张图片，其余参数按照第二章第三节中进行设置，且在各个实验中始终保持一致。

在本次实验中，加速比的计算公式为：

$$S_p = \frac{T_1}{T_p} \tag{1}$$

p 表示处理器数量或进程数量，T_1 表示程序顺序执行的时间，T_p 表示当有 p 个处理器或进程时程序的执行时间。当 $S_p = p$ 时，便可称为线性加速比。当某一并行算法的加速比为线性加速比时，若将处理器数量加倍，执行速度也会加倍。

并行效率的计算公式为：

$$E_p = \frac{S_p}{p} = \frac{T_1}{pT_p} \tag{2}$$

并行效率用于表示在解决问题时，相较于在通信与同步上的花费，参与计算的处理器获取成分利用的程度。由此可见，拥有线性加速比的算法并行效率为 1，E_p 的值介于 $0 \sim 1$ 之间。

4.1 并行加载数据实验

为计算加速比，首先得到不使用多进程加载数据的时间为 0.00044 秒。依次更改进程数量，观察耗时与计算加速比，结果如表 2所示。

表 2: 多进程加载数据的耗时与加速比

进程数量	时间（单位：秒）	加速比	并行效率
2	0.00081	0.54	27%
8	0.00056	0.78	9.8%
16	0.00044	1.0	6.25%
20	0.00042	1.04	5.23%
32	0.00037	1.18	3.71%
50	0.00027	1.62	3.25%
60	0.00049	0.89	1.49%
64	0.00126	0.35	0.5%

如表 2所示，在 20 核心 40 线程的处理器中，进程数量和处理器线程数近似相等时会取得较为良好的加速比。

1. 当进程数量很少时，处理器空载，系统需要维护任务的调度和进程通信的开销，所以加速比小于 1，效果反而不如单进程；
2. 当进程数量和处理器线程数近似时，处理器满载而不是空载或负载，所以取得的加速比效果较为良好，但并行效率较低；
3. 当进程远远大于处理器线程数时，处理器负载，需要更大的内存和更多的处理器负担来维护任务调度和进程通信的开销，加速比反而下降，并行效率也显著降低。

所以要合理的设置任务中进程的数量，尽量和处理器的线程数相等。

4.2 不同并行方式加速比对比

按照第二章的实验参数，进行不同数据并行方法的实验：DataParallel 和 DistributedDataParallel 并行方法对比。因为模型参数过多，所以不进行 CPU 实验和

单个 GPU 实验。考虑到有多人在同一时刻使用服务器，防止影响他人正常工作，所以不进行多机实验，只进行单机多卡下不同并行方法的对比实验。

防止用户端持续等待程序结束，所以使用 nohup 挂起执行程序。并将结果重定向输出到 log 日志中，程序结束后即可查看执行时间。DataParallel 消耗时间 $t_1 = 137172$ 秒，DistributedDataParallel 消耗时间 $t_2 = 89856$ 秒。由此可见，DistributedDataParallel 方法优于 DataParallel 方法，加速比为 $t_1/t_2 = 1.53$。

此外，如果只实现了模型加速而忽略了模型的准确率，那么并行加速就失去了意义。即并行加速的同时需要考虑结果的准确性，因此需要对比两种并行加速方法的准确率。DataParallel 的准确率为 0.566，DistributedDataParallel 的准确率为 0.582，准确率结果在可接受的误差范围内。

4.3 更改实验参数后的结果与分析

考虑到 DistributedDataParallel 方法的通信模式会受到数据量大小 batchsz 的影响，所以考虑更改 batchsz 的大小进而分析实验结论。为防止浪费计算资源，以 batchsz 为 30 的条件下，DistributedDataParallel 执行时间为阈值。当程序执行时间大于阈值后会自动中断。

1. 将 batchsz 的大小由 30 改为 2，即一次训练的数据量是之前的 $\frac{1}{15}$。多次执行 DistributedDataParallel 程序，程序因执行时间大于阈值而被强行中断。由此可见，AllReduce 在数据量很小的情况下，每一次梯度更新带来的进程通信、任务调度与分配的开销占比会增大，而用于数据通信的开销占比却很小，导致并行效率没有明显提升。因此 AllReduce 不适合数据量较小情况下的通信。

2. 而后将 batchsz 的大小由 30 改为 60，即一次训练的数据量是之前的 2 倍。多次执行 DistributedDataParallel 程序，得到的程序执行时间为 73824 秒。由此可见，当数据量较大时数据通信的占比也会随之增大，并行效率相对之前也有所提升。所以 AllReduce 更适合数据量较大情况下的通信。

参考文献

[1] F. Sung, Y. Yang, L. Zhang, T. Xiang, P. H. S. Torr, and T. M. Hospedales, “Learning to compare: Relation network for few-shot learning,” in 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2018, pp. 1199–1208.

[2] C. Chen, O. Li, D. Tao, A. Barnett, C. Rudin, and J. K. Su, “This looks like that: Deep learning for interpretable image recognition,” in Advances in Neural Information Processing Systems, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019, pp. 8930–8941.

[3] L. Song, F. Chen, Y. Zhuo, X. Qian, H. Li, and Y. Chen, “Accpar: Tensor partitioning for heterogeneous deep learning accelerators,” in 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2020, pp. 342–355.

[4] K. Sangaiah, M. Lui, R. Kuttappa, B. Taskin, and M. Hempstead, “Snacknoc: Processing in the communication layer,” in 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2020, pp. 461–473.

[5] J. Dong, Z. Cao, T. Zhang, J. Ye, S. Wang, F. Feng, L. Zhao, X. Liu, L. Song, L. Peng, Y. Guo, X. Jiang, L. Tang, Y. Du, Y. Zhang, P. Pan, and Y. Xie, “Eflops: Algorithm and system co-design for a high performance distributed training platform,” in 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2020, pp. 610–622.

[6] G. Kadam, D. Zhang, and A. Jog, “Bcoal: Bucketing-based memory coalescing for efficient and secure gpus,” in 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2020, pp. 570–581.

附录 A：遇到问题及解决方案

数据无法序列化

在深度学习中，常常需要把训练好的模型或使用数据对象存储起来，这样在训练或预测时直接将模型读出，而不需要重新读取数据，这样就大大节约了时间。Python 提供的 pickle 库就很好地解决了这个问题，它可以序列化对象并保存到磁盘中，并在需要的时候读取出来。

在加载数据时，使用多进程加载数据，并设置 num_workers 来指定进程的数量。在这样形式下获取的数据可以在 DataParallel 方法中训练，并没有错误。但在 DistributedDataParallel 方法训练时，提示有数据无法序列化的原因。

在仔细阅读 DistributedDataParallel 和 DataLoader 的官方文档后发现了原因，因为 num_workers 会开启多进程读取数据，且序列化数据。而 DistributedDataParallel 训练数据时，也会序列化数据。导致在 DistributedDataParallel 训练时会有两次序列化数据，因此产生错误。所以，在 DistributedDataParallel 训练方式中，只需要将 num_workers 设定为 0 即可。

显存溢出错误

深度学习的网络参数和数据量往往较大，因此常常发生显存溢出错误，且因为 DistributedDataParallel 独特的并行机制，会需要更大的显存空间，所以需要一些设计技巧来规避显存溢出问题。

1. 为避免浪费服务器的内存，可以把模型和数据通过.cuda() 方法放到 CUDA 中在使用。

2. 通常在训练结束后想要绘制 loss 的图像，都是将 loss 添加到一个列表中保存。但在保存前可以使用 detach 方法将 loss 的数值与计算图解锁，然后仅保存其数值。否则会将计算图添加到列表中，造成内存的浪费。

3. 使用 numpy 操作数据，而不是列表。python 的列表存储了对象引用地址和对象本身，因此对象在内存中并不连续，通过引用地址来访问对象。numpy 在会开辟连续的内存空间来存储对象，因此只需要存储一个引用。相比之下，在内存空间、访问效率等两个角度而言，numpy 比列表更具优势。

4. DistributedDataParallel 需要更大的显存空间去存储参数状态、自动求导的钩子、用于通信的参数桶和 Reducer。所以在每一次训练过程中，不能加载过大的的训练数据。

数据泄漏

在 DistributedDataParallel 并行方式中，显示 rank1 无法获取 rank0 数据的错误，可以分析得到出错的原因是 DistributedDataParallel 的机制限制了多进程之间不能共享同一张卡。

在发现问题出错的原因后，定位到代码前后查找错误来源。在训练过后的验证阶段，为计算网络的准确率，额外声明了一个计算图外的变量，所以这个变量并没有准确的归属，可能在 0 号显卡，也可能在 1 号显卡。在 AllReduce 通信时，发现一张显卡有这个变量，而其余的显卡没有这个变量，所以导致此类错误。因此需要将计算图外的变量放到每一张显卡中来避免此类错误，而显卡使用进程编号 rank 标记，所以只需要对数据使用.to(rank) 方法即可。在采用上述方法后，错误消失。

无法迭代 0 维张量

在训练的开始时，报错信息为：无法迭代 0 维度的数据，定位到出错代码位置，打印相关张量后发现数据维度并不是 0。但解释器仍然提示无法迭代 0 维向量。因此扩大 debug 范围，观察出错代码的函数，发现函数需要传入 5 个参数，而在调用时少传入一个参数只传入了 4 个参数，所以产生无法迭代 0 维张量的错误。在补全参数后，此错误消失，网络可以被正确训练。因此解释器的报错提示并不一定准确，还需要对出错代码进行实际查验。

附录 B：实验程序相关文件

使用 ssh 连接到服务器后并进入虚拟环境后，按照前文所述进行实验。因代码量较大，已达千行左右，可在 Github [2]获取完整程序。附录只对各个模块的核心程序进行展示。

并行加载数据方法的核心程序：

```
from torch.utils.data import DataLoader
from miniImagenet import MiniImagenet

# 加载测试数据集
mini = MiniImagenet('../mini-imagenet/', mode='train', n_way=n_way, k_shot=k_shot,
    k_query=k_query,
                                batchsz=10000, resize=224)
# pin_memory 快速的将数据转化为 GPU 可以处理的数据, num_workers 读取数据子线程的数量
db = DataLoader(mini, batchsz, shuffle=True,
                    num_workers=8, pin_memory=True)
mini_val = MiniImagenet('../mini-imagenet/', mode='val', n_way=n_way, k_shot=k_shot,
    k_query=k_query,
                            batchsz=200, resize=224)
db_val = DataLoader(mini_val, batchsz, shuffle=True,
                        num_workers=8, pin_memory=True)
```

[2]https://github.com/muyuuuu/Algorithm/tree/master/meta-learning/Metric-based/Relation-Netowrk

DistributedDataParallel 并行方法核心程序：

```
import torch.distributed as dist
import torch.multiprocessing as mp
from torch.nn.parallel import DistributedDataParallel as DDP

def setup(rank, world_size):
    os.environ['MASTER_ADDR'] = 'localhost'
    os.environ['MASTER_PORT'] = '12355'
    # initialize the process group
    dist.init_process_group("gloo", rank=rank, world_size=world_size)

def cleanup():
    dist.destroy_process_group()

def train(rank, world_size):
    print(f"Running basic DDP example on rank {rank}.")
    setup(rank, world_size)
    net = DDP(Compare(n_way, k_shot).to(rank), device_ids=[rank])
    # 训练阶段。遍历训练数据集中的每一个 batch
    for step, batch in enumerate(db):
        # 支持集合与查询集合
        support_x = Variable(batch[0]).cuda().to(rank)
        support_y = Variable(batch[1]).cuda().to(rank)
        query_x = Variable(batch[2]).cuda().to(rank)
        query_y = Variable(batch[3]).cuda().to(rank)
        # 开始训练
        net.train()
        # 计算 loss
        loss = net(support_x, support_y, query_x, query_y, rank)
        # Multi-GPU support
        loss = loss.mean()
        # 清空优化器之前的梯度
        optimizer.zero_grad()
        # 反向传播
        loss.backward()
        optimizer.step()
    cleanup()

def run_demo(demo_fn, world_size):
    mp.spawn(demo_fn,
             args=(world_size,),
             nprocs=world_size,
             join=True)

if __name__ == "__main__":
    run_demo(train, 3)
```

DataParallel 并行方法核心程序：

```
import torch
from torch.autograd import Variable
from torch import optim
from compare import Compare

net = torch.nn.DataParallel(Compare(n_way, k_shot)).cuda()
# 训练阶段。遍历训练数据集中的每一个 batch
for step, batch in enumerate(db):
    # 支持集合与查询集合
    support_x = Variable(batch[0]).cuda()
    support_y = Variable(batch[1]).cuda()
    query_x = Variable(batch[2]).cuda()
    query_y = Variable(batch[3]).cuda()
    # 开始训练
    net.train()
    # 计算 loss
    # print('computing loss....')
    loss = net(support_x, support_y, query_x, query_y)
    # Multi-GPU support
    loss = loss.mean()
    # 清空优化器之前的梯度
    optimizer.zero_grad()
    # 反向传播
    # print('backwarding ...')
    loss.backward()
    optimizer.step()
```

Relation Network 核心程序：

```
# 残差块
class Bottleneck(nn.Module):
    expansion = 4

    def __init__(self, inplanes, planes, stride=1, downsample=None):
        super(Bottleneck, self).__init__()
        self.conv1 = nn.Conv2d(inplanes, planes, kernel_size=1, bias=False)
        self.bn1 = nn.BatchNorm2d(planes)
        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3, stride=stride, padding
            =1, bias=False)
        self.bn2 = nn.BatchNorm2d(planes)
        self.conv3 = nn.Conv2d(planes, planes * 4, kernel_size=1, bias=False)
        self.bn3 = nn.BatchNorm2d(planes * 4)
        self.relu = nn.ReLU(inplace=True)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x):
        residual = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)
        out = self.relu(out)

        out = self.conv3(out)
        out = self.bn3(out)

        if self.downsample is not None:
            residual = self.downsample(x)

        out += residual
        out = self.relu(out)
        return out

# 训练：开始拼接
def forward(self, support_x, support_y, query_x, query_y, rank, train=True):
    """
    """
    # [多少组数组，每个类的样本数量，3个通道，图片高度，图片宽度]
    batchsz, setsz, c_, h, w = support_x.size()
    # 每组数据中 query 样本的数量
    querysz = query_x.size(1)
    # c 是 batchsize，d 是图片高度
    c, d = self.c, self.d

    # view 重构张量的维度
    # batchsz * setsz 是输入的 batchsize 大小
    # [b, setsz, c_, h, w] => [b*setsz, c_, h, w] => [b*setsz, c, d, d] => [b, setsz,
        c, d, d]
    support_xf = self.repnet(support_x.view(batchsz * setsz, c_, h, w)).view(batchsz,
        setsz, c, d, d)
    # [b, querysz, c_, h, w] => [b*querysz, c_, h, w] => [b*querysz, c, d, d] => [b,
        querysz, c, d, d]
    query_xf = self.repnet(query_x.view(batchsz * querysz, c_, h, w)).view(batchsz,
        querysz, c, d, d)

    # 按照图片的三个通道，将查询集和测试集拼接起来
    # squeeze：删除维度是 1 的数据
    # unsqueeze：在指定位置插入一维数据
    # expand 在指定维度进行扩展
    # 第一个维度，扩展查询集这么多个。[b, setsz, c, d, d] => [b, 1, setsz, c, d, d] =>
        [b, querysz, setsz, c, d, d]
    support_xf = support_xf.unsqueeze(1).expand(-1, querysz, -1, -1, -1, -1)
    # 第二个维度，扩展支持集这么多个。[b, querysz, c, d, d] => [b, querysz, 1, c, d, d
        ] => [b, querysz, setsz, c, d, d]
    query_xf = query_xf.unsqueeze(2).expand(-1, -1, setsz, -1, -1, -1)
    # cat: [b, querysz, setsz, c, d, d] => [b, querysz, setsz, 2c, d, d]
    comb = torch.cat([support_xf, query_xf], dim=3)

    # print('comb size is {}'.format(comb.size()))

    # print('c = {}'.format(c))
```

```
    comb = self.layer5(self.layer4(comb.view(batchsz * querysz * setsz, 2 * c, d, d)))
    # print('layer5 sz:', comb.size())
    comb = F.avg_pool2d(comb, 3)
    # print('avg sz:', comb.size())
    # push to Linear layer
    # [b * querysz * setsz, 256] => [b * querysz * setsz, 1] => [b, querysz, setsz, 1]
         => [b, querysz, setsz]
    score = self.fc(comb.view(batchsz * querysz * setsz, -1)).view(batchsz, querysz,
        setsz, 1).squeeze(3)

    # build its label
    # [b, setsz] => [b, 1, setsz] => [b, querysz, setsz]
    support_yf = support_y.unsqueeze(1).expand(batchsz, querysz, setsz)
    # [b, querysz] => [b, querysz, 1] => [b, querysz, setsz]
    query_yf = query_y.unsqueeze(2).expand(batchsz, querysz, setsz)
    # 标签相等，表示查询集属于这个类。和score进行对比
    # eq: [b, querysz, setsz] => [b, querysz, setsz] and convert byte tensor to float
        tensor
    label = torch.eq(support_yf, query_yf).float()

    # score: [b, querysz, setsz]
    # label: [b, querysz, setsz]
    if train:
        loss = torch.pow(label - score, 2).sum() / batchsz
        return loss

    else:
        # [b, querysz, setsz]
        rn_score_np = score.cpu().data.numpy()
        pred = []
        # [b, setsz]
        support_y_np = support_y.cpu().data.numpy()
        for i, batch in enumerate(rn_score_np):
            for _, query in enumerate(batch):
                # query: [setsz]
                sim = []  # [n_way]
                for way in range(self.n_way):
                    sim.append(np.sum(query[way * self.k_shot: (way + 1) * self.k_shot
                        ]))
                idx = np.array(sim).argmax()
                pred.append(support_y_np[i, idx * self.k_shot])
        # pred: [b, querysz]
        pred = Variable(torch.from_numpy(np.array(pred).reshape((batchsz, querysz)))).
            cuda().to(rank)

        correct = torch.eq(pred, query_y).sum()
        return pred, correct
```

特征提取层中的一组模块：

```
(conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
(bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
(bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
(bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
    )
(relu): ReLU(inplace=True)
(downsample): Sequential(
    (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
        True)
)
```

Relation Network 最后用于计算 Relatoin Score 的全连接层：

```
(fc): Sequential(
    (0): Linear(in_features=256, out_features=64, bias=True)
    (1): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
        True)
    (2): ReLU(inplace=True)
    (3): Linear(in_features=64, out_features=1, bias=True)
    (4): Sigmoid()
)
```