哈爾濱工業大學

课程研究报告

课程名称：模式识别与机器学习

报告题目：分布式机器学习

所在院系：未来技术学院

所在专业：数据科学与大数据技术

学生姓名：刘俊杉

学生学号：2021112078

选课时间：2023 年秋季学期

评阅成绩：

分布式机器学习

一、研究目的和意义

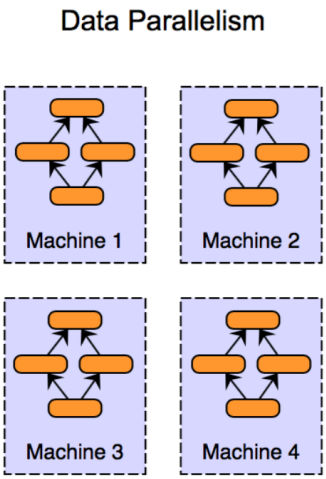
大数据、大模型为人工智能的飞速发展奠定了坚实的物质基础，也提出了新的技术挑战。近年来，越来越多的学者开始深入研究分布式机器学习，从而可以更高效地利用大数据训练更准确的大模型。分布式机器学习涉及如何分配训练任务，调配计算资源，协调各个模块以达到训练速度与精度的平衡。一个分布式机器学习系统通常会包含以下模块：数据和模型划分模块、单机优化模块、通信模块、模型和数据聚合模块等。机器学习虽然依赖于数据，但是它的目的是从数据中学习出规律或模型，而不是精准地对原始数据进行存储或者索引。机器学习对于数据细小差别具有很强鲁棒性。机器学习关心的是在测试集上的期待精度而不是训练集上的经验精度，所以如果分布式实现中引入一些微小噪声，很可能会增加学习过程的泛化能力，带来更好的测试结果。

二、大数据与大模型的挑战

随着互联网的发展，我们进入了一个前所未有的大数据时代。在大数据浪潮的推动下，带有标签训练数据的规模也取得了飞速增长。现在人们通常会用上千万的图像数据和语音数据训练模型。大规模的训练数据的出现为训练大模型提供了物质基础，大规模的机器学习模型具有超强的表达能力，可以帮助人们解决很多难度非常大的学习问题。另一方面，它们也有自己的弊端：非常容易过拟合。这就不可避免地导致大数据和大模型的双重挑战，从而对计算能力和存储容量都提出了新的要求。计算复杂度高，导致单机训练可能会消耗无法忍受的时长，因而不得不使用并行度更高的处理器或计算机集群来完成训练任务，存储容量大，导致单机无法满足要求，不得不使用分布式存储。

在浮点运算、并行计算等方面，GPU 可以提供数十倍乃至于上百倍于 CPU 的性能。通过使用 GPU，计算的并行度大大提升，很多计算任务都能得到大幅加速，这也正好解决了大规模机器学习的痛点。

之所以需要分布式机器学习，大体有三种情形：一是计算量太大，二是训练数据太多，三是模型规模太大。对于计算量太大的情景，需要将数据进行划分，并分配到多个工作节点上进行训练，这样每个工作节点的局部数据都在容限之内。每个工作节点会根据局部数据训练出一个子模型，并且会按照一定规律和其他工作节点进行通信，以保证最终可以有效整合来自各个工作节点的训练数据并得到全局的机器学习模型。对于模型规模太大的情形，则需要对模型进行划分，并且分配到不同节点上工作。与数据并行不同，在模型并行框架下各个子模型之间依赖性非常强，因为某个子模型的输出可能是另一个子模型的输入，如果不进行中间计算结果的通信，则无法完成整个模型训练。因此模型对通信要求较高。



图一：分布式机器学习架构

三、分布式机器学习框架

3.1 单机优化

完成数据或模型划分之后，每个工作节点只需要根据分配给自己的局部训练数据和子模型来进行训练。从这个意义上讲，除去各工作节点之间相互的通信以外，在每个工作节点自身的视野里，其实基本就是一个传统的单机机器学习任务：根据属于自己的训练数据，计算经验风险，然后利用某种优化算法通过最小化经验风险来学习模型的参数。

3.2 数据与模型并行

3.2.1 计算并行模式

如果所有的工作节点共享一块公共内存，并且数据和模型能全部存储于这块共享内存中，那么我们就不需要对数据和模型进行划分。这时，每个工作节点对数据有完全的访问权限，可以并行地执行相应的优化算法。

在这种并行模式下，当使用随机优化算法时，对数据的生成方式通常有两种不同的假定："在线数据生成"和"离线数据生成"。在线数据生成假定每个工作节点访问到的数据是按照真实分布即时生成出来的；离线数据生成则假定按照真实分布事先生成了一个离线数据集，之后每个工作节点再依据均匀分布从该离线数据集中重复采样得到训练所需的数据。一般来说，在线数据生成的假设对理解计算并行算法的理论性质有帮助，不过，实际中训练数据一般是事先离线生成的。

Algorithm 1: SSGD 算法

Input: 函数 $f(\cdot)$，全局参数 w^0，工作节点数 K，小批量规模 b，迭代轮数 T

Output: 终止全局参数 w^T

for $t = 0, 1, \cdots, T-1$ **do**
  Master 执行：
  广播当前模型 w^t;
  for $k = 1, 2, \cdots, K$ *in parallel* **do**
    Worker k 执行：
    接收当前模型 w^t;
    从局部数据中随机采小批量数据 $\mathcal{I}_k^t$;
    计算 $\mathcal{I}_k^t$ 上的随机梯度 $\nabla f_{\mathcal{I}_k^t}(w^t) = \sum_{i \in \mathcal{I}_k^t} \nabla f_i(w^t)$;
  end
  Master 执行：
  同步通信获得 $\sum_{k=1}^{K} \nabla f_{\mathcal{I}_k^t}(w^t)$;
  更新全局参数 $w^{t+1} = w^t - \eta^t \cdot \frac{1}{bK} \sum_{k=1}^{K} \nabla f_{\mathcal{I}_k^t}(w^t)$;
end
return w^T

图二：并行梯度下降法

3.2.2 数据并行模式

如果工作节点没有共享的公共内存，只有容量受限的本地内存，而训练数据的规模很大，无法存储于本地内存，就需要对数据集进行划分，分配到各个工作节点上，然后工作节点依据各自分配的局部数据对模型进行训练。

如果训练数据的样本量比较大，需要对数据按照样本进行划分。经典的数据样本划分方法有如下两种。

第一种是基于随机采样的方法。把原始训练数据集作为采样的数据源，通过有放回的方式进行随机采样，然后按照每个工作节点的内存容量为其分配相应数目的训练样本。随机采样方法可以保证每台机器上的局部训练数据与原始训练数据是独立同分布的，因此在训练的效果上有理论保证。但随机采样也有它的弊端：首先，因为训练数据较大，实现全局采样的计算复杂度比较高；其次，如果随机采样的次数小于数据样本的数目，可能有些训练样本会一直未被选出，导致辛苦标注的训练样本并没有得到充分的利用。

第二种是基于置乱切分的方法。该方法将训练数据进行随机置乱，然后按照工作节点的个数将打乱后的数据顺序划分成相应的小份，随后将这些小份数据分配到各个工作节点上。每个工作节点在进行模型训练的过程中，只利用分配给自己的局部数据，并且会定期地将局部数据再打乱一次。到一定阶段还可能再重新进行全局的数据打乱和重新分配。这样做的主要的目的是，让各个工作节点上的训练样本更加独立并具有更加一致的分布，以满足机器学习算法中训练数据要独立同分布的假设。置乱切分方法相比于随机采样方法，虽然数据的分布与原始数据分布略有偏差，但是其计算复杂度比全局随机采样要小很多，而且置乱切分能保留每一个样本，直观上对样本的利用更充分。

3.3 分布式机器学习算法

表一：常见的分布式机器学习算法及特点

分布式机器学习算法	对应单机优化	划分	通信	聚合
同步 SGD	SGD	数据样本划分	同步通信	全部模型加和
模型平均（MA）	不限	数据样本划分	同步通信	全部模型加和
BMUF	SGD	数据样本划分	同步通信	全部模型加和
ADMM	不限	数据样本划分	同步通信	全部模型加和
弹性 SGD（EASGD）	SGD	数据样本划分	同步通信	全部模型加和
异步 SGD	SGD	数据样本划分	异步通信	部分模型加和
Hogwild!	SGD	数据样本划分	异步通信	部分模型加和
Cyladies	SGD	数据样本划分	异步通信	部分模型加和
AdaDelay	SGD	数据样本划分	异步通信	部分模型加和
AdaptiveRevision	SGD	数据样本划分	异步通信	部分模型加和
带延迟补偿的异步 SGD	SGD	数据样本划分	异步通信	部分模型加和
DistBelief	SGD	数据样本划分;模型划分	异步通信	部分模型加和
AlexNet	SGD	模型划分	同步通信	数据聚合

3.3.1 MA 算法

Algorithm 1 FederatedAveraging

Server executes:
initialize w_0
for each round $t = 1, 2, \dots$ **do**
  $S_t = (\text{random set of } \max(C \cdot K, 1) \text{ clients})$
  for each client $k \in S_t$ **in parallel do**
    $w_{t+1}^k \leftarrow \text{ClientUpdate}(k, w_t)$
  $w_{t+1} \leftarrow \sum_{t=1}^{K} \frac{n_k}{n} w_{t+1}^k$

ClientUpdate(k, w): *// Executed on client k*
for each local epoch i from 1 to E **do**
  batches $\leftarrow$ (data $\mathcal{P}_k$ split into batches of size B)
  for batch b in batches **do**
    $w \leftarrow w - \eta \nabla \ell(w; b)$
return w to server

图三：MA 算法流程

MA 方法按照通信间隔的不同，可以分为下面两种情况：

1、 只在所有工作节点完成本地训练之后，做一次模型平均。这种情况所需的通信量极少，本地模型在迭代过程中没有任何交互，可以完全独立地并行计算，通信只在模型训练的最后发生一次。这类算法在强凸问题下的收敛性是有保证的，但对非凸问题不一定适用（比如神经网络），因为本地模型可能落到了不同的局部凸子域，对参数的平均无法保证最终模型的性能。

2、 在本地完成一定轮数的迭代之后，就做一次模型平均，然后用这次平均的模型的结果作为接下来的训练的起点，继续进行迭代，循环往复。相比于只在最终做一次模型平均，中间的多次平均控制了各个工作节点模型之间的差异，降低了它们落到不同的局部凸子域的可能性，从而保证了最终模型的精度。

在 MA 算法中，不论梯度的本地更新流程是什么策略，在聚合平均的时候都只是将来自各个工作节点的模型进行简单平均。如果把每次平均之间的本地更新称作一个数据块的话，那么模型平均可以看作基于数据块的全局模型更新流程。在单机优化算法中，常常会加

入冲量以有效地利用历史更新信息来减少随机梯度下降中梯度噪声的影响。类似地，我们也可以考虑在 MA 算法中对每次全局模型的更新引入冲量的概念。一种称为块模型更新过滤(Block-wise Model Update Filte-ring，BMUF)的算法基于数据块的冲量思想对 MA 进行了改进，其有效性在相关文献中被验证。BMUF 算法实际上是想利用全局的冲量，使历史上本地迭代对全局模型更新的影响有一定的延续性，从而达到加速模型优化进程的作用。

Algorithm 1 BMUF-Adam Algorithm

Input:
number of workers N, sync period τ, block momentum η, and initial parameter $\overline{\boldsymbol{\theta}}_0$

Input:
stochastic objective function $f(\boldsymbol{\theta})$, step size α, exponential decay rates for the moment estimates $[\beta_1, \beta_2]$, small scalar ϵ, and worker id i

Init: $\boldsymbol{\Delta}_0 \leftarrow \mathbf{0}$, $\rho_0 \leftarrow 0$, $n \leftarrow 0$, $\boldsymbol{m}_0^{(\text{init})} \leftarrow \mathbf{0}$, $\boldsymbol{v}_0^{(\text{init})} \leftarrow \mathbf{0}$

Init: $\boldsymbol{\theta}_{0,i} \leftarrow \overline{\boldsymbol{\theta}}_0$, $\boldsymbol{m}_{0,i} \leftarrow \mathbf{0}$, $\boldsymbol{v}_{0,i} \leftarrow \mathbf{0}$, $t \leftarrow 0$, $k \leftarrow 0$

while $\overline{\boldsymbol{\theta}}_t$ not converged **do**
  $t \leftarrow t+1$ % number of local steps
  $k \leftarrow k+1$ % number of Adam steps w.r.t. moment buffers
  $\boldsymbol{g}_{t,i} \leftarrow \frac{\partial f_t(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_{t-1,i}}$
  $\boldsymbol{m}_{t,i} \leftarrow \beta_1 \boldsymbol{m}_{t-1,i} + (1-\beta_1)\boldsymbol{g}_{t,i}$
  $\boldsymbol{v}_{t,i} \leftarrow \beta_2 \boldsymbol{v}_{t-1,i} + (1-\beta_2)\boldsymbol{g}_{t,i} \odot \boldsymbol{g}_{t,i}$
  $\hat{\boldsymbol{m}}_{t,i} \leftarrow \boldsymbol{m}_{t,i}/(1-\beta_1^k)$, $\hat{\boldsymbol{v}}_{t,i} \leftarrow \boldsymbol{v}_{t,i}/(1-\beta_2^k)$
  $\boldsymbol{\theta}_{t,i} \leftarrow \boldsymbol{\theta}_{t-1,i} - \alpha \hat{\boldsymbol{m}}_{t,i}/(\epsilon + \sqrt{\hat{\boldsymbol{v}}_{t,i}})$
  if t divides τ **then**
    $n \leftarrow n+1$ % number of BMUF steps
    Get $\overline{\boldsymbol{\theta}}_t$, $\overline{\boldsymbol{m}}_t$, $\overline{\boldsymbol{v}}_t$ by all-reduce
    $\boldsymbol{\Delta}_n \leftarrow \eta \boldsymbol{\Delta}_{n-1} + [\overline{\boldsymbol{\theta}}_t - (\overline{\boldsymbol{\theta}}_{t-\tau} + \eta \boldsymbol{\Delta}_{n-1})]$
    (which equals to $\boldsymbol{\Delta}_n \leftarrow \overline{\boldsymbol{\theta}}_t - \overline{\boldsymbol{\theta}}_{t-\tau}$)
    $\boldsymbol{\theta}_{t,i} \leftarrow \overline{\boldsymbol{\theta}}_t + \eta \boldsymbol{\Delta}_n$, $\rho_n \leftarrow \eta \rho_{n-1} + \tau$
    $\boldsymbol{m}_t^{(\text{init})} = \frac{\beta_1^\tau(\beta_1^{\eta\rho_n}-1)}{1-\beta_1^\tau} \boldsymbol{m}_{t-\tau}^{(\text{init})} + \frac{1-\beta_1^{\tau+\eta\rho_n}}{1-\beta_1^\tau} \overline{\boldsymbol{m}}_t$
    $\boldsymbol{v}_t^{(\text{init})} = \frac{\beta_2^\tau(\beta_2^{\eta\rho_n}-1)}{1-\beta_2^\tau} \boldsymbol{v}_{t-\tau}^{(\text{init})} + \frac{1-\beta_2^{\tau+\eta\rho_n}}{1-\beta_2^\tau} \overline{\boldsymbol{v}}_t$
    $\boldsymbol{m}_{t,i} \leftarrow \boldsymbol{m}_t^{(\text{init})}$, $\boldsymbol{v}_{t,i} \leftarrow \boldsymbol{v}_t^{(\text{init})}$, $k \leftarrow k + \eta\rho_n$
  end if
end while
return $\overline{\boldsymbol{\theta}}_t$

图四：BMUF-Adam 算法

3.3.2 Cyclades 算法

Cyclades 之前 naive 的异步并行算法，ASGD 和 Hogwild! 的想法最直接：既然原来的全局同步造成了等待与低效，干脆取消这个环节，所有节点都可以无锁（无等待）地对全局模型进行访问和更新。

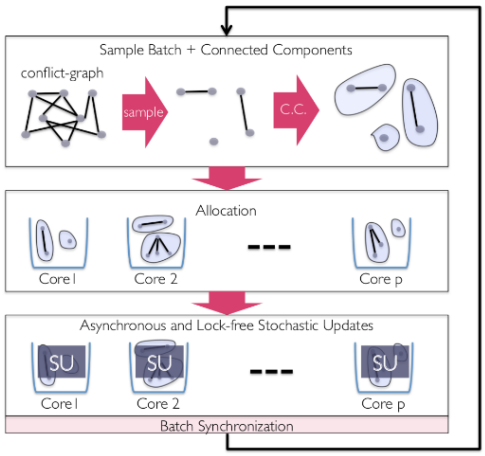
ASGD 是在多机场景下的算法实现。以 parameter server 架构为例，负责计算的 worker 一旦完成了当前节点的计算，可以直接把梯度更新 push 给储存模型参数的 server，server 则即刻更新参数，不用管别的 worker 的情况。

Hogwild! 是在单机多线程场景下的算法实现。由于多个线程共享内存，按常理某个线程更新时应该给内存加锁以保证模型写入的一致性，但是 Hogwild! 为了提高吞吐率，选择不带锁的多线程的写入。

这种粗暴的做法肯定会导致冲突的出现：ASGD 中滞后的梯度更新，或者 Hogwild!的写入冲突都会造成优化过程中的抖动甚至收敛失败，但是在一定的约束和补偿下，两者的收敛性都可以得到保障。从另一个角度看，在 SGD 这类算法的优化过程中我们一般都会引入一定的随机变量，减少陷入 local optima 逃不出来的情况，而异步算法的噪声则恰好起到了这个作用。

Cyclades 是对 Hogwild! 算法的改进，核心目标是减少多线程之间的冲突。其基本假设为：不同的训练样本更新的是模型的不同局部的参数。如果我们能把训练数据首先进行一个预划分，让更新相似位置参数的数据分在同一组，且后续的训练过程中同一组数据放在一个线程里计算更新，那么即使多个线程同时写入更新，由于不同线程更新的参数并不重叠，也

就不会产生冲突了。



图五：Cyclades 算法流程图

如图五所示，Cyclades 算法的基本流程如下：

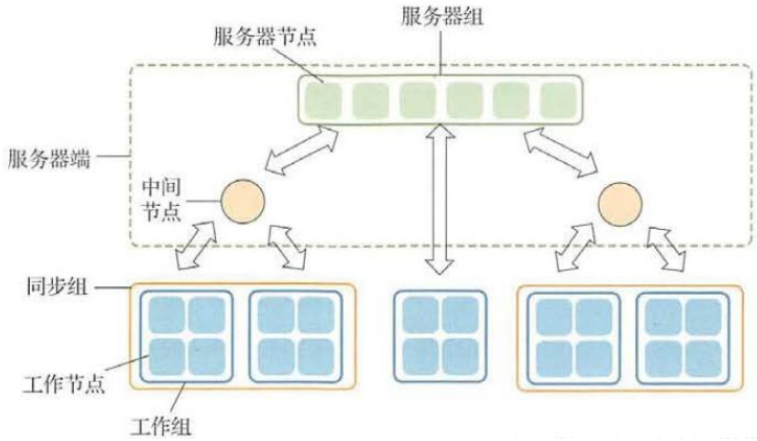
首先要对数据进行预处理。Cyclades 算法本质上是将样本进行分组，目的是使不同分组里的样本所对应的参数之间尽量不重叠。为了实现这个目的，首先构建样本和参数之间的二部图，进而得到样本之间的冲突关系图。然后对冲突关系图进行切分，得到相互不冲突的样本分组。接下来，按照分组情况将不同的样本分配到不同的核上进行计算。由于这些样本对应的参数区间基本互不重叠，因此不同核在读写对应参数时也基本不存在冲突。这样，该算法几乎可以完全复现单机串行版本的计算流程。

Cyclades 算法带来的好处是很直观的：它可以减少甚至完全消除无锁访问带来的冲突问题。同时由于计算任务被分割成小的分组，每个核都比较容易控制内存的访问，系统缓存的命中率也明显提高。因此，Cyclades 算法与 Hogwild！算法相比，无论是收敛速率还是加速性能都有更好的实验表现。

四、同步和异步的对比融合

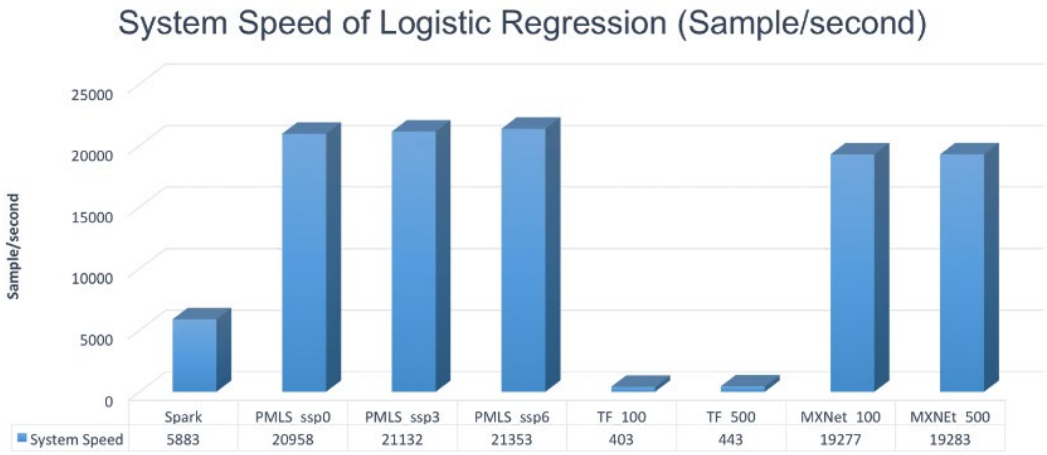
一般来看，SSGD 的进度要不 ASGD 高，与单机的 SGD 算法更加接近。DC-ASGD 算法由于采用了延迟补偿，精度有所提高。基于模型平均的算法 MA 和 BMUF 比其他算法精度要差一些，其中后者因为引入了冲量，收敛更快一些。每种算法在不同的应用场景之下各有优势。在通信代价不构成瓶颈的前提下，SSGD 是一种很好的选择。当通信成为短板时，可以使用异步算法。当异步算法的延迟很高时，可以使用 DC-ASGD 来缓解延迟问题。如果使用异步方法，但通信开销还是大于计算开销时，可以使用基于模型平均的方法。总之，没有最好的，只有最合适的。

同步和异步各有优缺点，如果能够把两者结合起来，取长补短，或许可以达到很好的效果。在上面对不同方法的对比中，我们发现不同算法有不同的应用场景。然而在实际中，我们的场景或许不是单一的，这种场景复杂的情况下，需要对场景进行分解，然后每个比较一致的场景下选用特定的算法。比如，对于机器数目很多、本地工作节点负载不均衡的集群。可以先根据每个节点的运算能力和通信能力进行聚类，将性能相近的节点分为不同的组。由于组内的各组运算速度差异较小，可以采用同步并行。组件的运算速度差异介意采用异步并行的方式。这种混合的方式既不会让运行速度慢的本地工作节点过度拖累全局训练速度，也不会引入过大的异步延迟进而影响收敛精度。下图展示了一个能够融合同步异步的原型系统架构。



图六：有分组的并行机器学习系统

五、实验结果对比图



图七：逻辑回归速度对比

该图展示了各个平台对逻辑斯蒂回归的计算速度。PMLS 和 MXNet 是最快的两个系统，TensorFlow 速度最慢，而 Spark 介于两者之间。对此，我们分析有多个导致原因。首先，相比于 Spark 和 TensorFlow，PMLS 是一个轻量级系统，它是采用高性能 C++ 编程语言实现的，而 Spark 是使用运行在 JVM 上的 Scala 语言编写的。其次，PMLS 中包含的抽象较少，而 TensorFlow 中则具有过多的抽象。抽象增加了系统的复杂度，并导致运行时开销。

图八：DNN 在各个平台上的速度

相比于单层神经网络，当发展到两层神经网络时，由于需要更多的迭代计算，Spark 的性能下降。我们将模型参数存储在 Spark 的驱动器中，因为驱动器可以容纳这些参数。但是如果我们使用 RDD 保持参数，并在每次迭代后做更新，结果可能会更差。

六、参考文献

[1] https://www.cnblogs.com/initzhang/p/13678262.html
[2] https://zhuanlan.zhihu.com/p/412045002
[3] https://zhuanlan.zhihu.com/p/365662727
[4] https://www.edu.cn/info/zhuan_jia_zhuan_lan/lx/202111/t20211118_2176663.shtml
[5] 分布式机器学习:交替方向乘子法在机器学习中的应用 Distributed machine learning :the alternative multiplier method in machine learning /雷大江著
[6]《分布式机器学习：算法、理论与实践》刘铁岩 陈薇 王太峰 高飞