同济大学

TONGJI UNIVERSITY

毕业实训

课题名称	面向分布式机器学习的梯度压缩方法研究
副 标 题	
学　　院	电子与信息工程学院
专　　业	计算机科学与技术
学生姓名	罗劲桐
学　　号	1951443
指导教师	杨恺
日　　期	2022 年　12 月　24 日

摘 要

随着用于机器学习训练的数据量不断增长，使用分布式集群进行训练的需求显著增加，大规模分布式训练需要大量的通信带宽来进行梯度交换，这限制了多节点训练的可扩展性，并且需要昂贵的高带宽网络基础设施。因此，主流的分布式框架，例如 horovod 和 BytePS 使用了梯度压缩方法作为大幅降低通信带宽的手段。梯度压缩减少了通信带宽，在某些情况下，它可以使训练更具可扩展性和效率，而不会显着降低收敛速度或准确性。

本文中，主要调研了当下主要的梯度压缩方法，并对方法在梯度压缩的方面进行了分类。梯度压缩的改进方向通常基于以下几个方面：压缩方向，分布式学习架构，通信压缩方式，误差补偿。通过分析当下主要的梯度压缩方法，我确定了进行梯度压缩需要改进的方向，并引出了下述的实验。

本文分析了 Polar Code 作为梯度压缩方法的可能性和 Polar Code 作为有损压缩算法的最优性，并使用开源信道编码的 Polar Code 库实现了信源压缩的梯度压缩算法。通过实验，验证了 Polar Code 作为有损压缩算法的最优性。最后，通过实验指出，Polar Code 可以作为梯度压缩的方法，并分析了 Polar Code 和其他稀疏梯度压缩方法可以共同使用的可能性。

本文最后使用了 BytePS 作为训练的分布式框架，用于比较不同梯度压缩算法和引入 Polar Code 梯度压缩。其中， 分析了 BytePS 作为 Parameter Server 架构的分布式训练框架，使用 Polar Code 作为梯度压缩方法的合理性。

关键词：梯度压缩，分布式训练，极化码

ABSTRACT

As the amount of data used for machine learning training continues to grow, the need to use distributed clusters for training has increased significantly. Large-scale distributed training requires a large amount of communication bandwidth for gradient exchange, which limits the scalability of multi-node training, and Requires expensive high-bandwidth network infrastructure. Therefore, mainstream distributed frameworks such as horovod and BytePS use the gradient compression method as a means to greatly reduce communication bandwidth. Gradient compression reduces communication bandwidth, and in some cases, it can make training more scalable and efficient without significantly reducing convergence speed or accuracy.

In this paper, the current main gradient compression methods are mainly investigated, and the methods are classified in terms of gradient compression. The improvement direction of gradient compression is usually based on the following aspects: compression direction, distributed learning architecture, communication compression method, and error compensation. By analyzing the current main gradient compression methods, I determined the direction of improvement in gradient compression, and led to the following experiments.

This paper analyzes the possibility of Polar Code as a gradient compression method and the optimality of Polar Code as a lossy compression algorithm, and uses the Polar Code library of open source channel coding to realize the gradient compression algorithm of source compression. Through experiments, the optimality of Polar Code as a lossy compression algorithm is verified. Finally, it is pointed out through experiments that Polar Code can be used as a gradient compression method, and the possibility that Polar Code and other sparse gradient compression methods can be used together is analyzed.

At the end of this article, BytePS is used as a distributed framework for training, which is used to compare different gradient compression algorithms and introduce Polar Code gradient compression. Among them, the rationality of using BytePS as a distributed training framework of Parameter Server architecture and using Polar Code as a gradient compression method is analyzed.

Key words： gradient compression, distributed training, polar codes

装 订 线

目 录

装 订 线

1 引言

1.1 课题背景

1.1.1 研究背景

大规模分布式训练需要大量的通信带宽来进行梯度交换，这限制了多节点训练的可扩展性，并且需要昂贵的高带宽网络基础设施。如果在移动设备上进行分布式训练（联邦学习），由于梯度传输会遇到更高的延迟、更低的吞吐量和间歇性的不良连接，导致训练速率严重受到影响。

现有的分布式训练加速方法中，通信架构是一个优化通信效率的重要因素。其中，通信架构通过两种方法做出提升。其一是提高通信的有效带宽，主要使用分布式通信架构进行同步、异步训练梯度通信管理，优化通信调度；而另一种减少通信开销的方法是减少传输的数据量，也就是梯度压缩。

1.1.1.1 提高通信的有效带宽

提高通信的有效带宽的通信架构主要分为两种调度方式。参数服务器(Parameter Server)和 All-Reduce 两种架构。其中 All-Reduce 以 horovod 为代表，使用于在实验室或者高性能计算集群等同构集群环境中，在较高通信稳定性的环境中效率较高。而参数服务器(Parameter Server)以 BytePS 为主要代表，利用 RDMA、NVIDIA 的集体通信库（NCCL）实现了高效的参数服务器通信。

1.1.1.2 减少通信的开销

减少通信开销的方法主要为梯度压缩。在分布式训练过程中，使用梯度压缩以缓解数据并行分布式优化中的通信瓶颈。在实验中，我们发现分布式 SGD 中 99.9%的梯度交换是冗余的，因此可以使用深度梯度压缩(DGC)以大大降低通信带宽。为了在压缩过程中保持准确性，DGC 使用梯度缓存和热身训练等方法，在不提高通信开销情况下，提升训练效率和准确率。

为了缓解通信瓶颈，梯度压缩的改进方向通常基于以下几个方面：压缩方向，分布式学习架构，通信压缩方式，误差补偿。

压缩方向包括单向梯度压缩和双向梯度压缩。单向梯度压缩仅在梯度上传时进行梯度压缩，有较好的压缩算法兼容性和通用性。双向梯度压缩在参数服务器，对聚合的随机梯度进行再次压缩，可以有效降低传输带宽，提高模型收敛速率。

分布式学习架构分为集中式 Param server，非集中式 All-Reduce。Paraeter Server 框架中 server 节点可以和其他 server 节点通信，共同维持所有参数的更新。而 worker 节点之间没有通信，只和对应的 server 有通信。在 All-Reduce 架构中将每个节点上数据切分为 N 份，然后经过 N-1 轮的 Reduce-Scatter 过程。然后进行 N-1 轮的 All Gather 过程，将每一个节点上的一部分的完整信息传递到所有节点上。

通信压缩方式分为量化和稀疏化两种方式。其中量化梯度压缩通过降低每个梯度的内存大小以降低传输带宽。而稀疏化梯度压缩通过稀疏矩阵或仅保留关键梯度的方式减少传递的梯度，从而降低传输带宽。理论上量化和稀疏化两种方式是可以同时应用的。

误差补偿是修正梯度压缩导致的训练精度下降的一个必要方法。当梯度压缩比、稀疏性极高时，稀疏更新会极大地损害收敛。我们发现误差补偿的方法可以积累局部梯度，可以缓解这个问题。

1.1.2 研究意义

数据并行的梯度压缩方法是加速深度学习模型训练的最常用方法。对于许多传统方法，例如 1-bit 压缩和深度梯度压缩(DGC)等梯度压缩方法，虽然这些方法在某些情况下显示出比全精度 SGD 的速度更快，但我们发现，在快速的网络和高度优化的通信后端，甚至在通用硬件上，通信效率提升十分有限。我们结合三个观点来解决这些问题：i）线性压缩器运算符通过使用 all-reduce 来实现可扩展性；ii）压缩和解压算法需要满足一定的时间复杂度要求，以减少通信过程中的额外开销；iii）最小化压缩损失的梯度质量。特别是，我们注意到，目前提出的梯度压缩方案不是线性的，不能很好地做出训练规模和结点的扩展，不同压缩方式对系统架构的兼容性要求较高，并不能兼容各种不同的分布式训练通信架构。

1.1.3 研究目标

研究了有损梯度压缩方法，以缓解数据并行分布式优化中的通信瓶颈。我们考虑使用极化码（Polar Code）和低复杂度的连续编码算法对梯度进行二进制对称源的有损压缩。使用极化码的有损压缩实现了二进制对称源的率失真约束，即在大多数情况下接近香农极限。因此，研究的目标时利用极化码压缩，实现可以 i)快速压缩梯度，ii)普遍上兼容其他的通信架构和可同时套用其他梯度压缩算法，以及 iii)满足接近率失真函数，即在大多数情况下接近香农极限。基于此需求，研究目标时在现有的主要通信架构，参数服务器(Parameter Server)和 All-Reduce 两种架构，和其他梯度压缩方法进行比较和兼容使用，验证使用极化码（Polar Code）和低复杂度的连续编码算法的效果。

1.2 研究现状

1.2.1 压缩方向

大规模机器学习的标准方法是分布式随机梯度训练，它需要计算网络上多个节点的聚合随机梯度。通信是此类应用的主要瓶颈，因此提出了压缩随机梯度方法来减少通信，同时使用误差补偿可以与压缩相结合，以在每个节点压缩其局部随机梯度并将结果一次性广播到网络上的所有其他节点的方案中实现更好的收敛。考虑压缩梯度传播的方向和误差补偿梯度保存的位置，现在的压缩方向可以分为单向压缩和双向压缩。

1.2.1.1 单向压缩

在流行的分布式学习参数服务器模型下，工作节点需要将压缩的局部梯度发送到参数服务器，参数服务器执行聚合。由于执行了梯度聚合，参数服务器端的梯度稀疏性与工作节点

不同，因此参数服务器无法以同样的压缩比压缩梯度并保持原有的收敛速率。参数服务器广播未经压缩的聚合后的梯度，将其发送回工作节点。

- Sparsified Gradient

梯度稀疏化方法，每个节点按幅度对梯度进行排序，并且仅传递分量的一个子集，在本地累积其余部分，使用矩阵稀疏化操作可以大量减少需要的通信带宽。这种方法可以将每步的通信量减少多达三个数量级，同时保持模型的准确性。

该方法为局部纠错的 TopK 梯度稀疏化方法提供了理论上的依据。包括在凸和非凸的平滑目标下，分布式梯度更新的收敛约束。

- Natural Compression

自然压缩 Natural Compression 是一种新的、简单但在理论上和实践上都有效的压缩技术。应用于待压缩更新向量的所有条目，并通过随机四舍五入到最接近的（负数或正数）2 的幂，通过忽略尾数计算压缩梯度。自然压缩对分布式 SGD 等流行训练算法的收敛速度的影响可以忽略不计，但带来的通信节省是巨大的，导致整体理论运行时间缩短 3-4 倍。

- Q-SGD

量化 SGD(QSGD)具有收敛保证和良好实际性能的压缩方案。QSGD 允许平滑地权衡通信带宽和收敛时间：节点可以调整每次迭代发送的比特数，通过灵活的比特压缩每个梯度，设置量化后的离散取值（梯度级别），可以灵活的选择压缩比。QSGD 保证在异步情况下凸目标和非凸目标的收敛，并且可以扩展到随机方差减少技术。

1.2.1.2 双向压缩

单向压缩的分布式学习参数服务器模型，不进行参数服务器广播回工作节点的梯度压缩。为了加速反向梯度传输，节省通信成本，在发送回 worker 节点之前需要再次压缩，也有必要在参数服务器上进行错误补偿，因为只有参数服务器才能跟踪历史压缩 error。

- Double Squeeze

在工作节点和参数服务器上都有误差补偿压缩，全局梯度也会分步压缩。包括同步和异步的两种方法。整个参数更新可以分为前向与后向两步，前向是根节点收集叶节点的局部梯度，后向是根节点计算全局梯度并返回叶节点。

- Survey

统一的框架和 API，允许在流行的机器学习工具箱上一致和容易地实现压缩通信。DNN 训练压缩通信方法进行了全面调查和实现，对各种 DNN（卷积和循环神经网络）、数据集和系统配置进行了全面的定量评估。

1.2.2 分布式学习架构

1.2.2.1 集中式 Param server

Paraeter Server 框架中，每个 server 都只负责分到的部分参数，server 节点可以和其他 server 节点通信，server group 共同维持所有参数的更新。Scheduler 节点负责维护一些元数据的一致性，例如各个节点的状态，参数的分配情况。worker 节点之间没有通信，只和对

应的 server 有通信。

schedule 负责通知每个 worker 加载自己对应的数据，然后去 server 节点上拉取一个要更新的参数分片，用本地数据样本计算参数分片对应的变化量，然后同步给 server 节点；server 节点在收到本机负责的参数分片对应的所有 worker 的更新后，对参数分片做一次 update。

- DGC

梯度分类我们通过只发送重要的梯度（稀疏更新）来减少通信带宽。只有大于阈值的梯度才被发送。其余的梯度进行局部积累，变得足够大后，可以被传送。因此，我们立即发送大梯度，但最终随着时间的推移发送所有的梯度。

为了在压缩过程中保持准确性，DGC 采用了四种方法：动量校正、局部梯度裁剪、动量因子掩蔽和热身训练。

- Parallel SGD

所有 worker 在同一个数据池上独立运行 SGD，并每隔一段时间对模型进行平均，将模型平均作为一种减少方差的机制。

- Local SGD

降低通信频率，通过在不同的 worker 上独立并行运行 SGD，并且偶尔对序列进行一次平均，在 worker 和 batch 数量实现了线性加速。通信轮数最多可以减少 T0.5 倍，其中 T 表示总步数。Local SGD 适用于异步实现，也可用于深度学习模型的大规模训练。

- NEOLITHIC

在单向还是双向使用无偏或收缩压缩器，我们都为算法建立了一个收敛下界。为了缩小下限和现有上限之间的差距，我们进一步提出了一种算法 NEOLITHIC，它在温和的条件下几乎达到了我们的下限（达到对数因子）。

1.2.2.2 非集中式 Reduce

PS 架构在模型稠密，需要大量交换信息的情况下，Server 节点很容易成为瓶颈，限制了其作用，也因此有了将 All Reduce 这一类通信方法应用到机器学习领域的尝试。

All Reduce 算法将集群内各个节点同时作为 server 和 worker 使用。每一个节点都只接收来自部分节点的信息，且都只广播信息给邻居节点。第一部分，对于 N 个节点的集群，将每个节点上数据切分为 N 份，然后经过 N-1 轮的 Reduce-Scatter 过程。每一轮中，每个节点向附近的邻居节点发送将自己的一个 chunk，并接收邻居发来的 chunk，并累加，经过这样的步骤，每一个节点都拥有一部分数据的最终结果。第二部分，进行 N-1 轮的 All Gather 过程，将每一个节点上的一部分的完整信息传递到所有节点上，每一个节点上就拥有了所有数据的完整信息。

- Distributed Subgradient

解决非平滑 convex 上的优化。每个 agent 根据从他的近邻收到的估计值和他使用次梯度方法获得的 cost 函数的局部信息。该方法涉及每个 agent 最小化目标函数，同时通过时变拓扑在本地与网络中的其他 agent 交换信息。

- Diffusion adaptation

自适应扩散 diffuse 机制。不需要在节点上使用循环路径，自适应网络 cost 函数随时间变化。扩散自适应允许节点实时协作和扩散信息；它还有助于通过持续学习过程减轻随机梯度噪声和测量噪声的影响。

- D2

在 Reduce 分布式训练框架中，由于每个 worker 只能访问一组有限标签的数据，不同 worker 回获得概率分布不同的训练样本。D2 的分布式框架对不同 worker 的数据差异不敏感。通过添加降低方差的组件，每个 worker 都会存储上一轮迭代的随机梯度和局部模型。

1.2.3 通信压缩方式

1.2.3.1 量化

量化梯度压缩通过降低每个梯度的内存大小以降低传输带宽。

- Sign-SGD

在深度神经网络的 SGD 训练中，如果量化误差在 mini-batch（误差反馈）之间传播，则可以只使用 1 位梯度，在不损失或几乎不损失准确性的情况下积极量化梯度。

- EF-SIGNSGD

有偏的梯度压缩方案（如 SIGNSGD 只包含梯度方向）会导致算法可能无法普及，甚至无法收敛。使用错误反馈，即将压缩运算符产生的错误合并到下一步中，可以克服这些问题。SIGNSGD 的上述问题可以通过以下方式解决：i)用梯度的法线缩放有符号的向量，以确保梯度的大小不会被遗忘；ii)本地存储实际梯度和压缩梯度之间的差异，iii)将其添加到下一步，以便正确的方向不会被遗忘。

- Q-SGD

QSGD 允许用户平滑地权衡通信带宽和收敛时间：节点可以调整每次迭代发送的比特数，代价可能是更高的方差。通过灵活的比特压缩每个梯度，可以适配合适的压缩比，保证更高的训练精度。

1.2.3.2 稀疏化

稀疏化梯度压缩通过稀疏矩阵或仅保留关键梯度的方式减少传递的梯度。

- Synchronous Distributed Optimization

随机丢弃随机梯度向量的坐标并适当放大剩余坐标以确保稀疏梯度无偏。为了有效地解决最优稀疏化问题，提出了几种简单快速的近似解算法，并从理论上保证了稀疏性。

- Parsified SGD

传输完整模型或梯度的最大元素的一定数量。具有 k-稀疏化或压缩（例如 top-k 或 random-k）的随机梯度下降（SGD），并表明该方案在配备误差补偿，可以在稀疏化梯度矩阵同时，保持训练精度。

- Power SGD

POWERSGD 使用低秩矩阵分解，计算梯度的低秩近似值。该近似值的计算量很轻，避免了任何昂贵的单值分解。为了提高高效近似的质量，我们通过重复使用前一个操作优化步

骤中的近似，来热启动动力迭代。

Power SGD 的优势在于：不同 workers 之间的高效聚合；用 all-reduce 协议来实现，而不需要集合操作，all-reduce 通信比参数服务器避免了双重压缩；由于 POWERSGD 方案是有偏差的，Error-feedback 版本通过引入压缩后动量来扩展原始算法

1.2.4 误差补偿

- EF-SIGNSGD

用梯度的法线缩放有符号的向量，表示梯度的大小；本地存储实际梯度和压缩梯度之间的差异，并将其添加到下一步，表示梯度方向。

- Sparsified SGD with Memory

具有 k-稀疏化或压缩（例如 top-k 或 random-k）的随机梯度下降（SGD），通过应用了内存向量 m 来缓存被压缩的数据，并应用于下一次的迭代，作为误差补偿，可以保持训练精度。

- EF21 with Bells & Whistles

现有的 EF 理论依赖于非常强的假设（例如，有界梯度），并提供悲观的收敛率。EF21 消除了 EF 的理论缺陷，同时在实践中效果更好。EF21 含有六个实用扩展，所有扩展都得到了强大的收敛理论的支持：部分参与、随机近似、方差减少、近端设置、动量和双向压缩。在结合 EF 的情况下（例如，双向压缩），EF21 训练速率要高得多。

装 订 线

2 研究内容与方法

2.1 Polar Codes 信源压缩算法

2.1.1Polar Code 简介

极化码在设计时并没有考虑最小距离特性，而是利用了信道联合（Channel Combination）与信道分裂（Channel Splitting）的过程来选择具体的编码方案，而且在译码时也是采用概率算法，这一点比较符合概率编码的思想。

对于长度为 $N=2^n$（n 为任意正整数）的极化码，它利用信道 W 的 N 个独立副本，进行信道联合和信道分裂，得到新的 N 个分裂之后的信道$\{W_N^{(1)},W_N^{(2)},...,W_N^{(N)}\}$。随着码长 N 的增加，分裂之后的信道将向两个极端发展：其中一部分分裂信道会趋近于完美信道，即信道容量趋近于 1 的无噪声信道；而另一部分分裂信道会趋近于完全噪声信道，即信道容量趋近于 0 的信道。假设原信道 W 的二进制输入对称容量记作 I(W)，那么当码长 N 趋近于无穷大时，信道容量趋近于 1 的分裂信道比例约为 $K=N\times I(W)$，而信道容量趋近于 0 的比例约为 $N\times(1-I(W))$。对于信道容量为 1 的可靠信道，可以直接放置消息比特而不采用任何编码，即相当于编码速率为 R=1；而对于信道容量为 0 的不可靠信道，可以放置发送端和接收端都事先已知的冻结比特，即相当于编码速率为 R=0。那么当码长$N 时，极化码的可达编码速率时，极化码的可达编码速率 R=NI(W)/N=I(W)$，即在理论上，极化码可以被证明是可达信道容量的。

在极化码编码时，首先要区分出 N 个分裂信道的可靠程度，即哪些属于可靠信道，哪些属于不可靠信道。对各个极化信道的可靠性进行度量常用的有三种方法：巴氏参数（Bhattacharyya Parameter）法、密度进化（Density Evolution，DE）法和高斯近似（Gaussian Approximation）法。

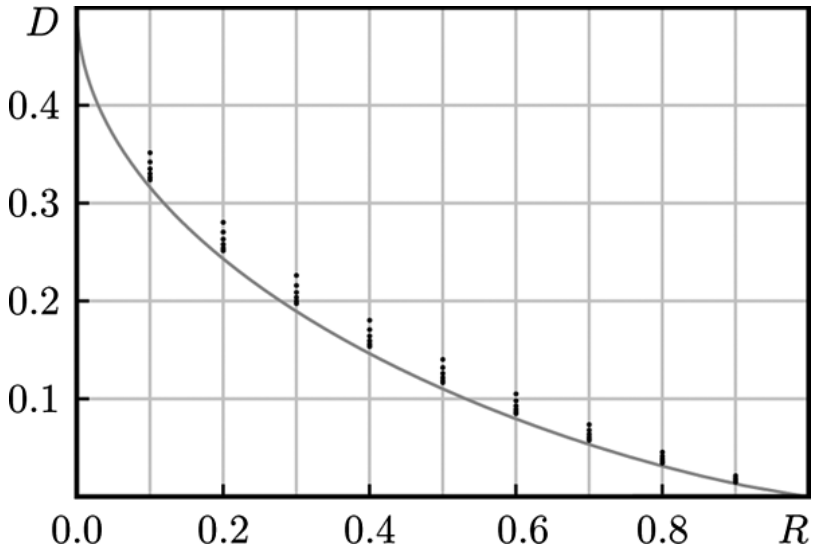
在大多数研究场景下，信道编码的传输信道模型均为 BAWGNC 信道。在 BAWGNC 信道下，可以将密度进化中的对数似然比（Likelihood Rate，LLR）的概率密度函数用一族方差为均值 2 倍的高斯分布来近似，从而简化成了对一维均值的计算，大大降低计算量，这种对 DE 的简化计算即为高斯近似。

2.1.2 使用 Polar Code 作为梯度压缩算法

2.1.2.1 简介

主要使用两个方法：极化码和低复杂度的连续编码算法。

率失真函数(RateDistortionFunction)：



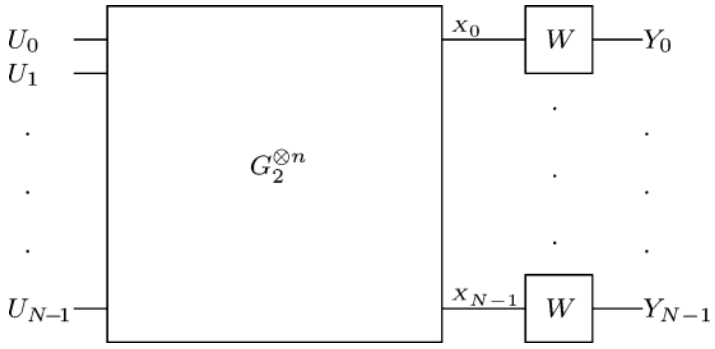
n=9,11,13,15,17 和 19 的算法的速率失真性能。随着区块长度的增加，各点越来越接近速率失真界限。其中，码的长度 N 总是 2 的幂，即 $N=2^n$。

2.1.2.2 极化码 Polar code 作为信道编码算法

当码长持续增加时，部分信道将趋向于容量近于 1 的完美信道（无误码），另一部分信道趋向于容量接近于 0 的纯噪声信道，选择在容量接近于 1 的信道上直接传输信息以逼近信道容量。

Polar Code 在容量趋于 1 的个子信道上传输消息比特 K 位，在其余子信道上传输冻结比特 N-K 位（即收发双方已知的固定比特，通常设置为全零）。码长 N 位。

U_F^C 是消息比特（$|U_F^C|=K$），U_F 是冻结比特（$|U_F|=N-K$ ），U^N 是传输的信息（$|U^N|=N$）。变换矩阵 G^N 与信息 U^N 相乘，所产生的向量 X^N（全噪通道 X_F，和无噪通道 X_F^C ）通过信道 W（BSS）传输，收到的字是 Y^N，通过 SCL 解码，得到 U_F^C 。



2.1.2.3 串行抵消算法 Successive Cancelation 作为信道解码的算法

对于有损源压缩，源编码的编码（解码）任务与信道编码的解码（编码）任务，这两种操作都能以 O(NlogN)的复杂度实现。

SCL 串行抵消算法中，U_i^N 按 0 到 N-1 的顺序进行解码。通过计算似然比（Likelihood Ratio，LR）L_N，确定 SCL 解码第 i 位的解码结果。

算法如下：

对于范围 0 到 N−1 中的每个 i：

1.如果 i∈F，则设 u_i=0。

2.如果 i∈F^c，则计算

$$L_N^{(i)}(\bar{y}, \hat{u}_0^{i-1}) = \frac{W_N^{(i)}(\bar{y}, \hat{u}_0^{i-1} \mid u_i = 0)}{W_N^{(i)}(\bar{y}, \hat{u}_0^{i-1} \mid u_i = 1)}$$

根据上述计算结果，u_i 的解码结果为：

$$\hat{u}_i = \begin{cases} 0, & \text{if } L_N^{(i)} > 1, \\ 1, & \text{if } L_N^{(i)} \leq 1. \end{cases}$$

2.1.2.4 极化码 Polar code 压缩原理

使用（原解码）SCL 算法作为编码算法，（原编码）变换矩阵 G^N，作为解码方法。

编码和 SCL 有略微不同：Y^N 是信源（$|Y^N|$=N），执行以下算法得到 U^N 为码字（$|U^N|$=N）。码字包括 U_F^C 是压缩后的信息（$|U_F^C|$=K），U_F 是冻结比特双方已知（$|U_F|$= N-K ）默认为 0。

对于范围 0 到 N−1 中的每个 i：

1.如果 i∈F，则设 u_i=0。

2.如果 i∈F^c，则计算

$$L_N^{(i)}(\bar{y}, \hat{u}_0^{i-1}) = \frac{W_N^{(i)}(\bar{y}, \hat{u}_0^{i-1} \mid u_i = 0)}{W_N^{(i)}(\bar{y}, \hat{u}_0^{i-1} \mid u_i = 1)}$$

根据上述计算结果，u_i 的解码结果根据概率设置为：

$$\hat{u}_i = \begin{cases} 0 & \text{w.p. } \frac{L_N^{(i)}}{1+L_N^{(i)}} \\ 1 & \text{w.p. } \frac{1}{1+L_N^{(i)}}. \end{cases}$$

2.1.2.5 解码方法

变换矩阵 G^N 与信息 U^N（拼接后）相乘，所产生的向量 X^N 即为结果

2.1.3 修改 Polar code

Encode、Decode 方法调换；修改 Encode 方法，使用 randomized rounding 代替 hard decision；修改 LR 生成的时机，需要在编码时生成（原在 AWGN 传输时）。

- 指定输入信息长度 N，压缩后码长 K，信噪比 SNR（用于计算 LLR 初始值）：

  1.Compress 压缩传输的 message（N->K）

  2.Decompress 解压缩 message（K->N）

```
1.  class PolarCompress:
        def __init__(self, N, K, SNR):
```

```
            self.N = N
            self.K = K
            self.SNR = SNR

            # initialise polar code
            self.myPC = PolarCode(self.N, self.K)
            self.myPC.construction_type = 'bb'

            # mothercode construction
            Construct(self.myPC, SNR)
            # print(myPC, "\n\n")

        def compress(self, message):
            # set message
            AWGN_pure(self.myPC, self.SNR, message)
            # print("The message is:", message)

            # encode message
            Compress(self.myPC)
            # print("The coded message is:", myPC.message_received)

            return self.myPC.message_received

        def decompress(self, message_received):
            # transmit the codeword
            self.myPC.set_message(message_received)

            # decode the received codeword
            Decompress(self.myPC)
            # print("The decoded message is:", myPC.get_codeword())
            # print("Precision:", numpy.count_nonzero(my_message == myPC.get_codeword())
    / my_message.shape[0])
            # print("Diffs:", numpy.where(my_message != myPC.get_codeword()))

            return self.myPC.get_codeword()
```

- 信噪比 SNR->Eb_No

  Modulation+get_likelihoods 计算 LLR 初始值，用户压缩编码时计算 LLR

```
2.  def __init__(self, myPC, Eb_No, message, plot_noise=False):
        """
        Parameters
        ----------
        myPC: `PolarCode`
            a polar code object created using the `PolarCode` class
        Eb_No: float
            the design SNR in decibels
        plot_noise: bool
            a flag to view the modeled noise

        """

        self.myPC = myPC
        self.Es = myPC.get_normalised_SNR(Eb_No)
```

```
        self.No = 1
        self.plot_noise = plot_noise

        tx = self.modulation(message)
        # rx = tx + self.noise(self.myPC.N)
        self.myPC.likelihoods = np.array(self.get_likelihoods(tx), dtype=np.float64)

        # change shortened/punctured bit LLRs
        if self.myPC.punct_flag:
            if self.myPC.punct_type == 'shorten':
                self.myPC.likelihoods[self.myPC.source_set_lookup == 0] = np.inf
            elif self.myPC.punct_type == 'punct':
                self.myPC.likelihoods[self.myPC.source_set_lookup == 0] = 0
```

- Random decision 函数

  按照概率分配 0、1

  OverflowError：Y 较大的情况，可以直接分配 0。

```
3.  def random_decision(y):
        """
            Random decision of a log-likelihood.
        """
        rand = random.uniform(0, 1)
        try:
            if rand <= 1.0 / (math.exp(y) + 1):
                return 1
            else:
                return 0
        except OverflowError:
            return 0
```

2.1.4 冻结组件

需要指定集合 F，即冻结组件的集合。

首先，我们估计所有 i∈{0,...,N-1}的 $W_N^{(i)}$ ，并按照 $W_N^{(i)}$的递减顺序对指数 i 进行排序。

集合 F 由第一个 RN 指数组成，即由 RN 最大 $W_N^{(i)}$组成。

Let W be the $\mathrm{BSC}(D)$, i.e.

$$W(0 \mid 1) = W(1 \mid 0) = D,$$
$$W(0 \mid 0) = W(1 \mid 1) = 1 - D.$$

一般的，发送方和接收方通过默认冻结组件为全 0，减少 Polar Code 初始化所需信息。

2.2 BytePS 通信流程

2.2.1Worker

Worker 节点执行主要的计算，如读取数据和计算梯度。它通过 push 和 pull 与 Server 节点进行通信。它将计算出的梯度推送给服务器，或者从服务器中拉出最近的模型。

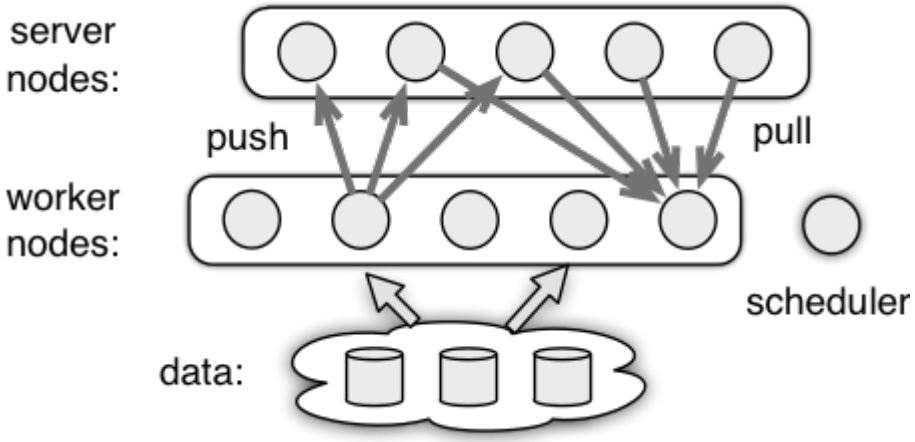
2.2.2Server

Server 节点维护和更新模型权重。每个节点只维护模型的一部分。

2.2.3Scheduler

Scheduler 节点监控其他节点的活跃度。它也可以用来向其他节点发送控制信号，并收集它们的进度。通过指定 scheduler 和 server 的 IP+port，通过 TCP 链接+传输。

基本的调度原则是：神经网络的前几层的通信具有更高的优先级，可以抢占后几层的通信。分割大层，合并小层。



2.2.4 分布式通信流程

以以下启动方式为例：

```
4.  python dist_launcher.py --worker-hostfile worker_hosts --server-hostfile
    server_hosts \
5.          --scheduler-ip 10.0.0.1 --scheduler-port 12345 \
6.          --username root --env ENV1:1 --env ENV2:2 \
7.          'echo this is $DMLC_ROLE; python byteps/launcher/launch.py YOUR_COMMAND'
```

通过指定 worker_hosts、server_hosts 、scheduler-ip+scheduler-port 可以显式的指定所有端口，不需要发现的过程了

然后每个结点包括多个（4 个）线程，使用 TCP/RDMA 与其他结点通信，ZMQ 进程内通信在节点内通信

Worker 需要绑定计算单元+计算，PropagatingThread 实现

Server+scheduler 不使用计算单元，byteps.server 实现

Scheduler 通信时具体使用 ps/Postoffice.cc 实现调度算法

2.3 配置环境和复现

2.3.1 Byteps

```
1. git clone --recursive https://github.com/bytedance/byteps
2. cd byteps
```

```
3. python3 setup.py install
```

需要 nccl，但使用 pytorch 的 nccl 也可以运行

```
1. export NVIDIA_VISIBLE_DEVICES=0,1,2,3  # gpus list
2. export DMLC_WORKER_ID=0 # your worker id
3. export DMLC_NUM_WORKER=1 # one worker
4. export DMLC_ROLE=worker
5. bpslaunch python example/pytorch/benchmark_byteps.py --model resnet50 --num-iters 1000000
```

只有一个机器多卡的训练方式

2.3.2 Polar Code

```
1. pip install py-polar-codes
```

安装 python Polar Code 组件

```
1. myPC = PolarCode(256, 100)          # 初始化
2. Construct(myPC, design_SNR)                    # 构造
3. myPC.set_message(my_message)                   # 设置传输内容
4. Encode(myPC)               # 编码
5. AWGN(myPC, design_SNR)              # 传输
6. Decode(myPC)               # 解码
7. GUI()            # 开始测试压缩传输
```

2.3.3 训练流程

与 MXNet 不同，pytorch 构造方式。因为 pytorch 使通过优化器定义梯度计算和压缩的方式，所以需要使用优化器

```
1. # BytePS: wrap optimizer with DistributedOptimizer.
2. optimizer = bps.DistributedOptimizer(
    optimizer, named_parameters=model.named_parameters(),
    compression=compression,
    backward_passes_per_step=args.batches_per_pushpull)
```

batches-per-pushpull 表示每次梯度沟通的 epoch 数

```
1. parser.add_argument('--batches-per-pushpull', type=int, default=1,
                    help='number of batches processed locally before '
                         'executing pushpull across workers; it multiplies '
                         'total batch size.')
```

2.3.4 梯度压缩

当前 pytorch 只实现了一个 16 位压缩算法

在 byteps_push_pull 之前将本地梯度转换为低精度的数据格式，并在 byteps_push_pull 之后将汇聚的梯度恢复为原始精度。由于返回的压缩数据与输入位于相同的 context 中（例如相同的 GPU 设备），因此 BytePS 内核不需要进行任何更改。

```
1. class FP16Compressor(Compressor):
```

```
    """Compress all floating point gradients to 16-bit."""
    @staticmethod
    def compress(tensor):
        """Downcasts the tensor to 16-bit."""
        tensor_compressed = tensor
        if tensor.dtype.is_floating_point:
            # Only allow compression from other floating point types
            tensor_compressed = tensor.type(torch.float16)
        return tensor_compressed, tensor.dtype

    @staticmethod
    def decompress(tensor, ctx):
        """Upcasts the tensor to the initialization dtype."""
        tensor_decompressed = tensor
        dtype = ctx
        if dtype.is_floating_point:
            tensor_decompressed = tensor.type(dtype)
        return tensor_decompressed
```

2.3.5 BytePS 训练

2.3.5.1 Numactl 设置

numactl 工具可用于查看当前服务器的 NUMA 节点配置、状态，可通过该工具将进程绑定到指定 CPU core，由指定 CPU core 来运行对应进程。

当前单机模拟分布式的情况会导致绑定到同一组 CPU core，可能会影响训练效率。

解决方案：设置环境参数，绑定指定的 CPU，BYTEPS_VISIBLE_CPU_CORES=0-6,28-34

2.3.5.2 BytePs 单机训练

```
        CUDA_VISIBLE_DEVICES=4,5,6,7
        NVIDIA_VISIBLE_DEVICES=0,1,2,3
        DMLC_WORKER_ID=0
        DMLC_NUM_WORKER=1
        DMLC_ROLE=worker
        bpslaunch python3
    /root/github/gradient-compression/byteps/example/pytorch/benchmark_byteps.py
        --model resnet50
```

2.3.5.3 分布式训练

```
        DMLC_NUM_WORKER=1 DMLC_ROLE=scheduler DMLC_NUM_SERVER=1
    DMLC_PS_ROOT_URI=127.0.0.1 DMLC_PS_ROOT_PORT=1234 BYTEPS_FORCE_DISTRIBUTED=1 bpslaunch
    &
        DMLC_NUM_WORKER=1 DMLC_ROLE=server DMLC_NUM_SERVER=1 DMLC_PS_ROOT_URI=127.0.0.1
    DMLC_PS_ROOT_PORT=1234 BYTEPS_FORCE_DISTRIBUTED=1 bpslaunch &
        CUDA_VISIBLE_DEVICES=0,1 NVIDIA_VISIBLE_DEVICES=0,1 DMLC_WORKER_ID=0 \
        DMLC_NUM_WORKER=1 DMLC_ROLE=worker DMLC_NUM_SERVER=1 DMLC_PS_ROOT_URI=127.0.0.1
    DMLC_PS_ROOT_PORT=1234 BYTEPS_FORCE_DISTRIBUTED=1 \
```

```
19.      bpslaunch python3
    /root/github/gradient-compression/byteps/example/pytorch/train_mnist_byteps.py
    --fp16-pushpull &
```

2.3.5.4 新增压缩方法

- Sign-SGD

  只包含符号的 1 位压缩

```
20.      class OneBitCompressor(Compressor):
       """Compress all floating point gradients to 16-bit."""
       average = torch.as_tensor(1.0)
       sums = torch.as_tensor(0)

       @staticmethod
       def compress(tensor):
           """Downcasts the tensor to 16-bit."""
           tensor_compressed = tensor
           tensor_size = tensor.numel()
           tensor_average = tensor.abs().mean()
           if OneBitCompressor.sums > 50000:
               OneBitCompressor.sums = tensor_size
               OneBitCompressor.average = tensor_average
           else:
               OneBitCompressor.average = (OneBitCompressor.average *
   OneBitCompressor.sums + tensor_size * tensor_average
                                           ) / (OneBitCompressor.sums + tensor_size)
               OneBitCompressor.sums += tensor_size

           if tensor.dtype.is_floating_point:
               # Only allow compression from other floating point types
               tensor_compressed = tensor.type(torch.bool).type(torch.float16)
           return tensor_compressed, tensor.dtype

       @staticmethod
       def decompress(tensor, ctx):
           """Upcasts the tensor to the initialization dtype."""
           tensor_decompressed = tensor
           dtype = ctx
           if dtype.is_floating_point:
               tensor_decompressed = tensor.type(dtype)*OneBitCompressor.average
           return tensor_decompressed.type(dtype)
```

- Natural Compression

  自然对数取整作为压缩的结果

```
21.      class NaturalCompressor(Compressor):
       """Compress all floating point gradients to 16-bit."""

       @staticmethod
       def compress(tensor):
           """Downcasts the tensor to 16-bit."""
           tensor_compressed = tensor
           if tensor.dtype.is_floating_point:
               # Only allow compression from other floating point types
```

```
            tensor_compressed = torch.tensor(-tensor.abs().log() * tensor.sign(),
dtype=torch.int8)
        return tensor_compressed, tensor.dtype

    @staticmethod
    def decompress(tensor, ctx):
        """Upcasts the tensor to the initialization dtype."""
        tensor_decompressed = tensor
        dtype = ctx
        if dtype.is_floating_point:
            tensor_decompressed = (tensor.sign() * (-tensor.abs()).exp()).type(dtype)
        return tensor_decompressed
```

3 实验分析

3.1 评价标准

3.1.1BytePS

梯度压缩算法和分布式训练框架包含以下主要的评价标准，梯度稀疏率，通信带宽，梯度压缩时间复杂度，训练时长用于评价训练框架提高通信的有效带宽和减少通信的开销的能力，而模型精度和损失函数曲线用于评价训练框架保持梯度精度的能力。

当梯度达到 99.9%稀疏度，只有绝对值最大的 0.1%梯度发送到参数服务器。一般使用 top-k selection 算法找到 threshold。梯度压缩能极大减少网络开销，但会影响收敛，导致模型精度降低。因此梯度压缩既要评价训练加速的速率，同时要评价最终的模型精确度，达到不损失精度。

一般的，梯度压缩算法使用稀疏矩阵的 top-k 和 random-k 稀疏矩阵压缩算法压缩梯度，梯度稀疏率一般大于 99%。梯度压缩时间复杂度主要影响梯度在节点内留存的时间，一般梯度压缩时间复杂度小于 O(NlogN)时，对训练加速影响较小，和通信的开销相比可以忽略，因此梯度压缩时间复杂度一般都在 O(NlogN)以内。训练时长是梯度压缩的终极目标，是最好的体现梯度压缩算法效果的评价标准。

另外，模型精度是训练中进行模型 evaluation 时的精度，一般的，经过梯度压缩的模型精度在前几个 epoch 会落后于无梯度压缩的模型精度，但在训练的中后期，两者的模型精度应该趋同。模型精度也是评价评价梯度压缩算法保持模型精度的最重要的指标。损失函数和模型精度作用类似，但损失函数在每个 epoch 都有结果，可以更精确的度量模型精确度的变化。

3.1.2Polar Code

极化码 Polar Code 在连续解码策略下实现了任意对称二进制输入离散无记忆信道的容量。同时 Polar Code 可以实现有损源压缩的等效结果，即这种组合实现了二进制对称源 BSS 的率失真界限。

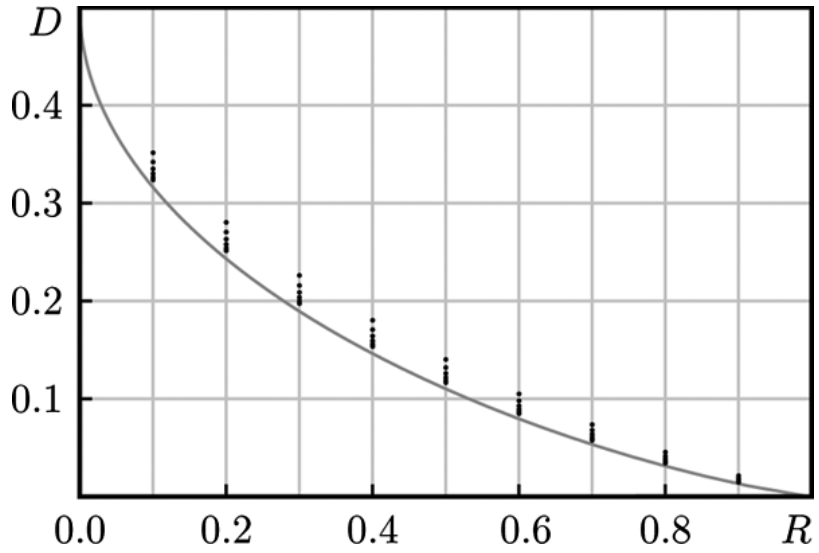
考虑二进制对称源(BSS)Y。让 d(·,·)表示汉明失真函数

$$\mathrm{d}(0,0)=\mathrm{d}(1,1)=0,\mathrm{d}(0,1)=1.$$

众所周知，为了以平均失真 D 压缩信源 Y，速率 R 必须至少为 R(D)=1−h2(D)，其中 h2(·)是二元熵函数。香农对这种率失真界限的证明是基于随机编码论证的，因此也适用于梯度压缩的梯度假设，即梯度也是随机 0-1 编码，不具备规律性。

因此在信道编码中，速率 R 被选择为严格小于 1−h2(D)，而在信源压缩（即 Polar Code 压缩编码）中，它被选择为严格大于这个量。且随着块长度的增加，这些点接近率失真界限。因此，Polar Code 压缩算法评价标准即在不同的传输速率 R 下，被压缩的信息 Y 最终解码后的结果，应接近率失真函数，且随着块长度的增加，这些点应更接近率失真界限。

率失真函数(Rate Distortion Function)：



3.2 实验结果与分析

3.2.1 BytePS

3.2.1.1 训练设置

batch-size=64

Epochs=100

Lr=0.01

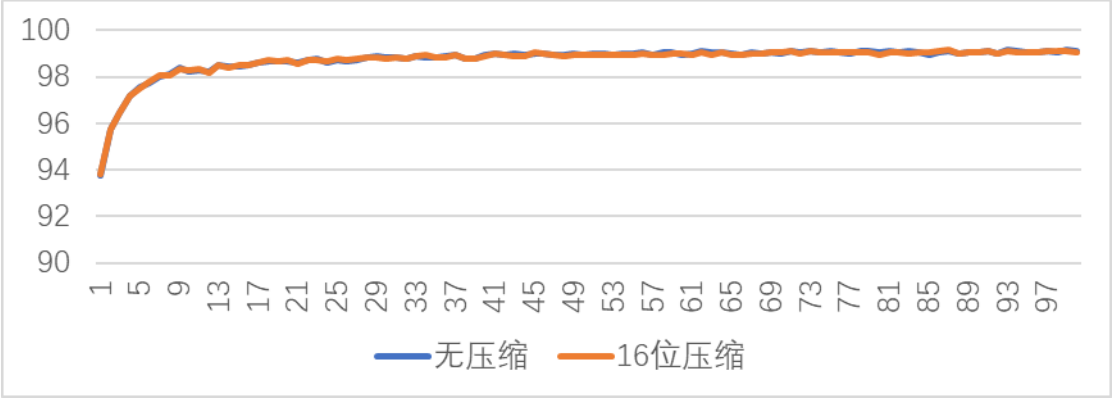
Momentum=0.5

MNIST 数据集

GPU*2

3.2.1.2 训练结果

方法	压缩时间复杂度	时间	Averageloss（NLLloss）	Accuracy
无压缩	O(1)	34min	0.0296	99.11%
16 位压缩	O(N)	25min	0.0298	99.03%
1bit	O(N)	20min	0.23096	95.8%



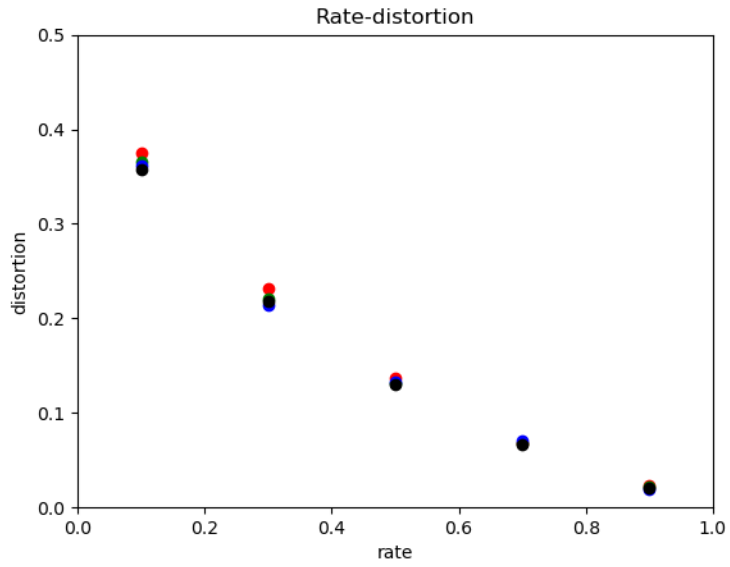
所有压缩算法的实验基于同样的训练配置，环境一致。从训练结果中可以看出，两种压缩方法都满足时间复杂度要求。

其中 16 位压缩能够有效降低模型的通信带宽，减少训练所需时间。同时，从训练模型精度曲线和损失函数的结果，16 位压缩和无压缩基本达到了一致的训练精度，是可行的梯度压缩方式。

1bit 压缩减少训练所需时间比 16 位压缩更加明显，因为有更低的通信传输信息量。但从训练模型精度曲线和损失函数的结果，1bit 压缩损失的模型精度较多，在保持模型精度的方面效果较差。

3.2.2Polar Code

以下实验中，信息块长度选择了 256,512,1024,2048 几个长度，分别标为颜色红、绿、蓝、黑。



N\R	0.1	0.3	0.5	0.7	0.9

256	0.37369792	0.24088542	0.14453125	0.07942708	0.02604167
512	0.36653646	0.21614583	0.13151042	0.07291667	0.02213542
1024	0.3531901	0.21386719	0.13151042	0.06445312	0.02018229
2048	0.35579427	0.21809896	0.12906901	0.06852214	0.02001953

从以上实验结果可以分析得出，Polar Code 梯度压缩是符合论文中评价标准的。整体曲线接近于率失真函数，随着块长度的增加，这些点应更接近率失真界限。

4 结论与展望

通过本次毕业实训的研究和学习，我调研了当今使用较多的梯度压缩算法，并分析了梯度压缩算法重要的特性。通过分析这些特性，我确定了进行梯度压缩需要改进的方向，并尝试使用极化码和 BytePS 框架作为梯度压缩实验的主要工具。由于 Polar Code 作为有损压缩算法的最优性，Polar Code 可以作为梯度压缩方法，并且能和其他梯度压缩方法良好适配，体现了 Polar Code 的兼容性和研究前景。最后，本文使用了 BytePS 作为训练的分布式框架，用于比较不同梯度压缩算法和引入 Polar Code 梯度压缩。

在之后的毕业设计中，我会完成 Polar Code 的梯度压缩代码，并使其与大量其他梯度压缩算法可兼容使用。然后，在 BytePS 作为训练的分布式框架，比较 Polar Code 梯度压缩方法和其他方法，并将 Polar Code 和其他梯度压缩方法共同使用，进行比较实验和改进。

参考文献

[1] D. Alistarh, T. Hoefler, M. Johansson, S. Khirirat, N. Konstantinov, and C. Renggli. The convergenceof sparsified gradient methods. InAdvances in Neural Information Processing Systems, 2018.

[2] S. Horvath, C.-Y. Ho, L. Horvath, A. N. Sahu, M. Canini, and P. Richt´arik. Natural compression fordistributed deep learning.ArXiv, abs/1905.10988, 2019.

[3] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic. Qsgd: Communication-efficient sgd viagradient quantization and encoding. InAdvances in Neural Information Processing Systems, 2017.

[4] H. Tang, X. Lian, T. Zhang, and J. Liu. Doublesqueeze: Parallel stochastic gradient descent withdouble-pass error-compensated compression.ArXiv, abs/1905.05957, 2019.

[5] H. Xu, C.-Y. Ho, A. M. Abdelmoniem, A. Dutta, E. H. Bergou, K. Karatsenidis, M. Canini, andP. Kalnis. Compressed communication for distributed deep learning: Survey and quantitative evaluation.Technical report, 2020

[6] Lin, Y., Han, S., Mao, H., Wang, Y., & Dally, W.J. (2018). Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. ArXiv, abs/1712.01887.

[7] Zhang, J., Sa, C.D., Mitliagkas, I., & Ré, C. (2016). Parallel SGD: When does averaging help? ArXiv, abs/1606.07365.

[8] Stich, S.U. (2019). Local SGD Converges Fast and Communicates Little. ArXiv, abs/1805.09767.

[9] Huang X, Chen Y, Yin W, et al. Lower Bounds and Nearly Optimal Algorithms in Distributed Learning with Communication Compression[J]. arXiv preprint arXiv:2206.03665, 2022.

[10] A. Nedic and A. Ozdaglar, "Distributed Subgradient Methods for Multi-Agent Optimization," in IEEE Transactions on Automatic Control, vol. 54, no. 1, pp. 48-61, Jan. 2009, doi: 10.1109/TAC.2008.2009515.

[11] J. Chen and A. H. Sayed, "Diffusion Adaptation Strategies for Distributed Optimization and Learning Over Networks," in IEEE Transactions on Signal Processing, vol. 60, no. 8, pp. 4289-4305, Aug. 2012, doi: 10.1109/TSP.2012.2198470.

[12] Tang H, Lian X, Yan M, et al. D $^ 2$: Decentralized Training over Decentralized Data[J]. arXiv preprint arXiv:1803.07068, 2018.

[13] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu. 1-bit stochastic gradient descent and its application todata-parallel distributed training of speech dnns. InINTERSPEECH, 2014.

[14] Karimireddy S P, Rebjock Q, Stich S U, et al. Error Feedback Fixes SignSGD and other Gradient Compression Schemes[J]. arXiv preprint arXiv:1901.09847, 2019.

[15] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic. Qsgd: Communication-efficient sgd viagradient quantization and encoding. InAdvances in Neural Information Processing Systems, 2017.

[16] J. Wangni, J. Wang, J. Liu, and T. Zhang. Gradient sparsification for communication-efficient dis-tributed optimization. InAdvances in Neural Information Processing Systems, 2018.

[17] S. U. Stich, J.-B. Cordonnier, and M. Jaggi. Sparsified sgd with memory. InAdvances in NeuralInformation Processing Systems, 2018.

[18] Vogels T, Karimireddy S P, Jaggi M. PowerSGD: Practical low-rank gradient compression for distributed optimization[J]. Advances in Neural Information Processing Systems, 2019, 32.
[19] Karimireddy S P, Rebjock Q, Stich S U, et al. Error Feedback Fixes SignSGD and other Gradient Compression Schemes[J]. arXiv preprint arXiv:1901.09847, 2019.
[20] S. U. Stich, J.-B. Cordonnier, and M. Jaggi. Sparsified sgd with memory. InAdvances in NeuralInformation Processing Systems, 2018.
[21] I. Fatkhullin, I. Sokolov, E. A. Gorbunov, Z. Li, and P. Richt´arik. Ef21 with bells & whistles: Practicalalgorithmic extensions of modern error feedback.ArXiv, abs/2110.03294, 2021.