



优化方法及应用 课程报告

问题名称 DDP 变长序列动态批调度

成 员 杨逍宇 3220105453

日 期 2025 年 12 月 25 日

指导老师 何衍

摘要

在同步式分布式数据并行（DDP）训练中，单步耗时由最慢 GPU 决定。NLP 任务的变长序列与动态 padding 使各卡有效 token 数与显存峰值产生显著波动，形成“慢卡”瓶颈并拉长尾延迟。本文将“每步 per-GPU micro-batch 分配”形式化为小规模 min-max 整数优化问题，提出一种动态调度策略：在每步开始前预取候选样本，构造基于 padded tokens 的成本代理，通过枚举求解最优 per-rank batch size，并结合 DDP 梯度缩放保证全局平均梯度一致性。

在 4 卡 DDP + SST-2 + DistilBERT 的快速实验中，动态调度显著降低了跨卡 padded tokens 的方差，并在较高 global batch 规模下实现 step time 的 P95 改善与吞吐提升。该方法不依赖复杂外部框架，工程实现轻量，能较好体现“数学规划在工程实践中的落地价值”。

关键词：分布式训练 变长序列 min-max 整数优化 动态调度 PyTorch DDP

目录

摘要	1
一、 前言	3
1.1 问题背景	3
1.2 研究意义	3
1.3 相关基础知识与公开资料	3
1.4 研究内容与论文结构	3
二、 问题建模	4
2.1 建模假设与符号定义	4
2.2 成本代理的构造与合理性	4
2.3 min-max 整数优化模型	4
2.4 与 DDP 梯度等价性的关系	4
三、 求解方法	5
3.1 预取 buffer 与 exact 枚举	5
3.2 算法复杂度与可实现性	5
3.3 工程实现与工具链	5
3.4 关键工程要点	5
四、 案例分析	5
4.1 硬件与软件环境	5
4.2 实验设置	7
4.3 指标体系	7
4.4 结果与讨论	8
4.5 图表分析	8
4.6 失败设置与经验总结	10
五、 结论与亮点	11
参考文献	12

一、前言

1.1 问题背景

在 DDP 同步训练中，所有 GPU 需要在每一步结束时完成梯度同步，因此整体 step time 由最慢的 rank 决定 [1]。对于 NLP 任务，样本长度分布具有明显长尾特征；动态 padding 使不同 rank 的有效 token 数不一致，从而导致计算量、显存占用与通信等待呈现随机波动。固定 per-rank batch size 虽易实现，但在高并发训练下会放大“慢卡效应”，影响吞吐与稳定性。另一方面，Transformer 的注意力计算复杂度与序列长度呈平方相关 [2]，使得长度波动在计算代价上被进一步放大。

1.2 研究意义

该问题具有直接的工程价值：在保持模型精度不变的前提下，通过调度策略降低尾延迟、提升 GPU 利用率，能显著提高分布式训练的效率。更重要的是，该问题天然适合数学规划建模：每步决策变量为各卡样本数，目标为最小化最大开销，约束为全局 batch 与显存上限。这一过程完整呈现了“问题抽象—模型建立—算法求解—工程验证”的优化闭环，契合课程对数学规划应用与计算求解能力的要求。

1.3 相关基础知识与公开资料

本文参考并总结了官方文档与公开资料中的基础知识：

- **DDP 同步机制**：梯度 allreduce 的平均化会在不同 local batch 下产生偏差，需要额外的 loss 缩放以保持全局平均梯度一致 [1]。
- **变长序列与 padding**：序列长度决定 attention 计算代价与显存占用，padded tokens 可作为合理的代理成本 [2]。
- **数据与模型**：采用 GLUE/SST-2 数据集与 DistilBERT 模型，相关使用方式可见 Hugging Face 的 Datasets 与 Transformers 文档 [3–6]。

这些公开资料为实验搭建提供了可复现的工程基础，同时也为问题建模提供了清晰的物理含义。

1.4 研究内容与论文结构

本文的主要研究内容如下：

1. 将 per-step micro-batch 分配形式化为 min-max 整数规划，并引入可计算的成本代理；
2. 在 $G = 4$ 的场景下采用 exact 枚举求解，并加入方差作为 tie-break；
3. 结合 DDP 梯度缩放推导，保证动态 batch 下梯度等价；
4. 在 SST-2 上进行静态与动态对比实验，评估 step time、吞吐与跨卡均衡性；

5. 总结失败设置与实践经验，为后续改进提供方向。

论文结构安排如下：第二章建立数学模型；第三章给出求解方法与实现细节；第四章进行案例分析；第五章总结结论与亮点并讨论局限与展望。

二、问题建模

2.1 建模假设与符号定义

设总 GPU 数为 $G = 4$ ，第 t 步第 g 张 GPU 的 micro-batch 大小为 $b_{g,t}$ ，全局目标 batch 为 B 。设每卡最大可承受 batch 上限为 b_g^{\max} （由 OOM 探测得到）。DDP 同步训练的关键在于最慢 GPU 的开销决定整步耗时，因此自然对应于 min-max 目标。

2.2 成本代理的构造与合理性

每个 rank 在本步开始前预取 K 个样本，记前 k 个样本的最大长度为 $\max_len[k]$ ，定义

$$\text{padded_tokens}[k] = k \cdot \max_len[k].$$

考虑两种成本代理：

$$c_g(k) = \text{padded_tokens}[k], \quad \text{或} \quad c_g(k) = (\text{padded_tokens}[k])^2.$$

该代理具有单调性，并与注意力计算复杂度一致性较好 [2]。在工程实现中，计算成本与显存开销可用同一代理近似，从而降低建模复杂度。

2.3 min-max 整数优化模型

对每一步，求解以下问题：

$$\min_{b_1, \dots, b_G} \max_g c_g(b_g) \quad \text{s.t.} \quad \sum_g b_g = B, \quad 1 \leq b_g \leq b_g^{\max}, \quad b_g \in \mathbb{Z}.$$

当多个解具有相同最大成本时，进一步选择成本方差 $\text{Var}(c_g(b_g))$ 更小的方案，以增强均衡性。该模型可视为对“最慢 GPU 瓶颈”的直接优化，并与 DDP 同步机制相一致。

2.4 与 DDP 梯度等价性的关系

当各 rank 的 local batch 不同，若直接 allreduce 平均梯度会改变有效学习率。设每个 rank 使用 mean loss L_g ，全局 batch 为 $B = \sum_g b_g$ ，则应在 backward 前缩放：

$$\tilde{L}_g = L_g \cdot \frac{G \cdot b_g}{B}.$$

此时 allreduce 平均后的梯度与全局平均梯度等价，确保优化目标不受动态 batch 分配影响。

三、求解方法

3.1 预取 buffer 与 exact 枚举

由于 $G = 4$ 且 B 不大，本文采用 exact 枚举求解。具体流程：

1. 每个 rank 预取 K 个样本形成 buffer，计算 $c_g(k)$ ；
2. 通过 all-gather 将成本数组汇总到 rank0；
3. rank0 枚举所有满足 $\sum b_g = B$ 的组合，选择 min-max + 最小方差解；
4. 广播 b_g 给各 rank，按分配取样训练。

3.2 算法复杂度与可实现性

在 $G = 4$ 的场景下，枚举规模随 B 的增长近似为 $O(B^{G-1})$ ，对本文设定的 batch 范围而言完全可接受。相比启发式调度，exact 枚举具有确定性、可解释性强的优势，适合作为课程项目展示优化求解亮点。

3.3 工程实现与工具链

系统实现基于 PyTorch DDP 与 Hugging Face Transformers/Datasets[3, 4]，数据集为 GLUE/SST-2[5]，模型为 DistilBERT[6]。训练过程记录 per-step 指标并输出到 CSV，分析脚本基于 pandas 与 matplotlib 自动生成图表与汇总表。代码结构遵循模块化设计，调度算法与训练逻辑解耦，便于复现与扩展。

3.4 关键工程要点

- **OOM 保护**：通过 max_length 截断与启动时 batch 探测得到 b_g^{\max} 。
- **动态 loss 缩放**：按 $\tilde{L}_g = L_g \cdot G b_g / B$ 缩放 loss，保持全局平均梯度等价。
- **指标记录**：每步记录 step time、peak memory、padded tokens、吞吐等，便于后续统计分析。

四、案例分析

4.1 硬件与软件环境

实验在单机 4 卡环境上完成，GPU 为 4 张 NVIDIA RTX 3090。图1 和图2 为实验前后的 nvidia-smi 截图，用于展示硬件配置与 GPU 占用情况。

Thu Dec 25 13:30:02 2025									

NVIDIA-SMI 570.124.04				Driver Version: 570.124.04			CUDA Version: 12.8		

GPU Name		Persistence-M		Bus-Id	Disp.A	Volatile Uncorr. ECC			
Fan	Temp	Perf	Pwr:Usage/Cap		Memory-Usage	GPU-Util	Compute M.		
						MIG M.			
=====									
0	NVIDIA GeForce RTX 3090		On	00000000:25:00.0	Off			N/A	
32%	43C	P2	167W / 350W	3853MiB / 24576MiB		93%	Default		
								N/A	

1	NVIDIA GeForce RTX 3090		On	00000000:41:00.0	Off			N/A	
33%	44C	P2	171W / 350W	3923MiB / 24576MiB		91%	Default		
								N/A	

2	NVIDIA GeForce RTX 3090		On	00000000:C1:00.0	Off			N/A	
31%	42C	P2	181W / 350W	3853MiB / 24576MiB		96%	Default		
								N/A	

3	NVIDIA GeForce RTX 3090		On	00000000:E1:00.0	Off			N/A	
30%	39C	P2	179W / 350W	3851MiB / 24576MiB		95%	Default		
								N/A	

Processes:									
GPU	GI	CI	PID	Type	Process name	GPU Memory			
	ID	ID				Usage			
=====									
0	N/A	N/A	950952	C	.../envs/term-project/bin/python	3844MiB			
1	N/A	N/A	950953	C	.../envs/term-project/bin/python	3914MiB			
2	N/A	N/A	950954	C	.../envs/term-project/bin/python	3844MiB			
3	N/A	N/A	950955	C	.../envs/term-project/bin/python	3842MiB			

图 1 实验环境 nvidia-smi 截图 (1)

```
Every 1.0s: nvidia-smi
```

Thu Dec 25 13:31:11 2025

NVIDIA-SMI 570.124.04		Driver Version: 570.124.04		CUDA Version: 12.8	
GPU	Name	Persistence-M	Bus-Id	Disp.A	Volatile Uncorr. ECC
Fan	Temp	Pwr:Usage/Cap	Memory-Usage	GPU-Util	Compute M.
	Perf				MIG M.
0	NVIDIA GeForce RTX 3090	On	00000000:25:00.0	Off	N/A
33%	46C P2	170W / 350W	3855MiB / 24576MiB	48%	Default
1	NVIDIA GeForce RTX 3090	On	00000000:41:00.0	Off	N/A
34%	46C P2	174W / 350W	3925MiB / 24576MiB	93%	Default
2	NVIDIA GeForce RTX 3090	On	00000000:C1:00.0	Off	N/A
32%	46C P2	185W / 350W	3855MiB / 24576MiB	90%	Default
3	NVIDIA GeForce RTX 3090	On	00000000:E1:00.0	Off	N/A
30%	42C P2	180W / 350W	3855MiB / 24576MiB	89%	Default

Processes:						
GPU	GI	CI	PID	Type	Process name	GPU Memory Usage
	ID	ID				
0	N/A	N/A	950952	C	.../envs/term-project/bin/python	3846MiB
1	N/A	N/A	950953	C	.../envs/term-project/bin/python	3916MiB
2	N/A	N/A	950954	C	.../envs/term-project/bin/python	3846MiB
3	N/A	N/A	950955	C	.../envs/term-project/bin/python	3846MiB

图 2 实验环境 nvidia-smi 截图 (2)

软件环境基于 Python + PyTorch DDP 与 Hugging Face 生态，训练与分析脚本均可在同一环境下运行。

4.2 实验设置

- 数据集: SST-2 (GLUE)
- 模型: distilbert-base-uncased
- 训练配置: 4 GPU, global batch $B = 64$, steps=800, max_length=256
- Baseline: static (每卡固定 16)
- 方法: dynamic (per-step min-max 调度, cost=tokens)
- buffer size: 64, 支持更稳定的成本估计

4.3 指标体系

本文关注三类指标:

1. 时延类：每步最大 step time 的均值与 P95；
2. 吞吐类：samples/s 与 tokens/s 的均值与 P95；
3. 均衡性：跨卡的 step time、padded tokens 与 peak memory 标准差。

4.4 结果与讨论

表1 汇总了静态与动态调度的对比结果。动态调度在时延和吞吐上均有改善：mean max step time 降低 3.70%，P95 降低 3.82%；samples/s 平均提升 3.85%，P95 提升 4.16%。更显著的是，跨卡 padded tokens 标准差降低 71.60%，表明动态调度有效缓解了长度不均衡。需要注意的是，peak memory 标准差略有上升（约 4.12%），说明在某些步长分配更激进时，仍可能引入显存波动，这为后续调度策略改进提供了方向。

表 1 Static 与 Dynamic 对比结果

指标	Static	Dynamic	提升 (%)
Mean max step time (ms)	156.80	150.99	-3.70
P95 max step time (ms)	161.37	155.21	-3.82
Mean throughput (samples/s)	408.27	423.97	+3.85
P95 throughput (samples/s)	418.67	436.10	+4.16
Mean throughput (tokens/s)	14261.50	14456.85	+1.37
P95 throughput (tokens/s)	17276.87	17966.80	+3.99
Mean std(step time)	0.327	0.307	-6.10
Mean std(padded tokens)	112.06	31.83	-71.60
Mean std(peak mem)	1.422	1.481	+4.12

4.5 图表分析

图3 展示了每步最大 step time 的时间序列，动态调度在大多数步骤上保持更低的波动；图4 展示了 padded tokens 的跨卡标准差，动态方法在全程显著低于 static。图5 与图6 分别展示 step time 与 peak memory 的跨卡标准差，体现了动态调度在时延均衡性上的整体改善。图7 展示了 step time 分布，动态方法在尾部分布上更集中。

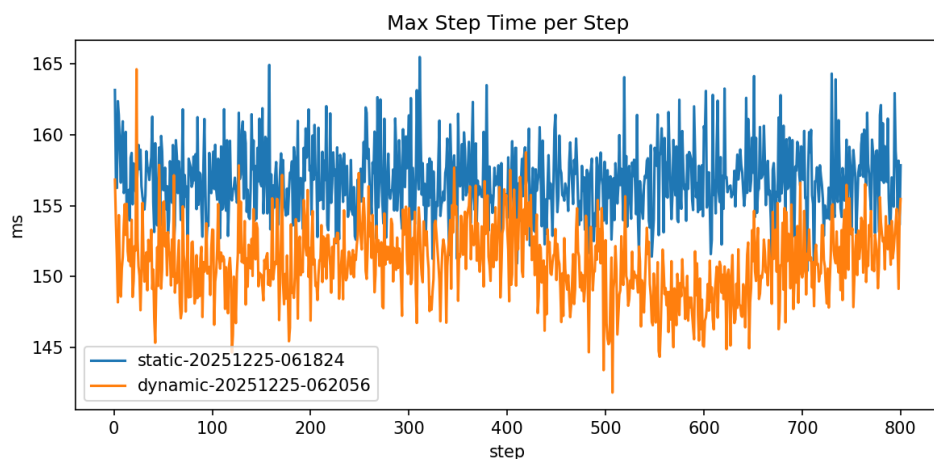


图3 每步最大 step time (static vs dynamic)

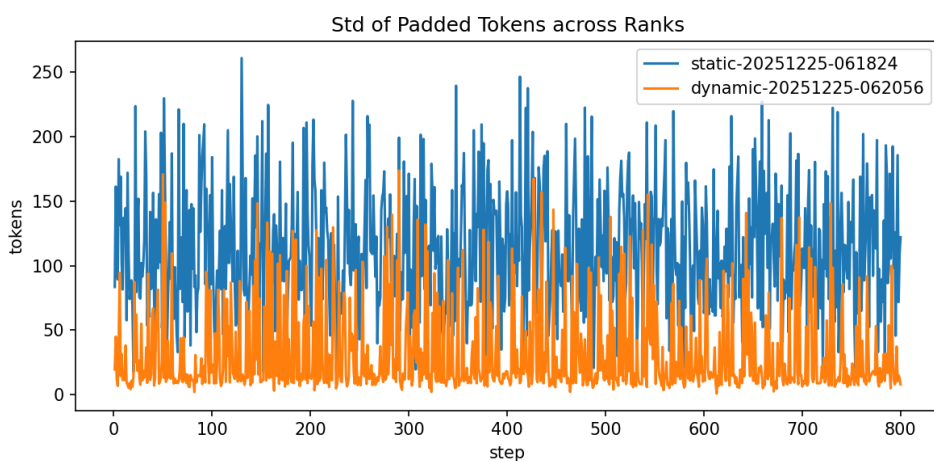


图4 跨卡 padded tokens 标准差

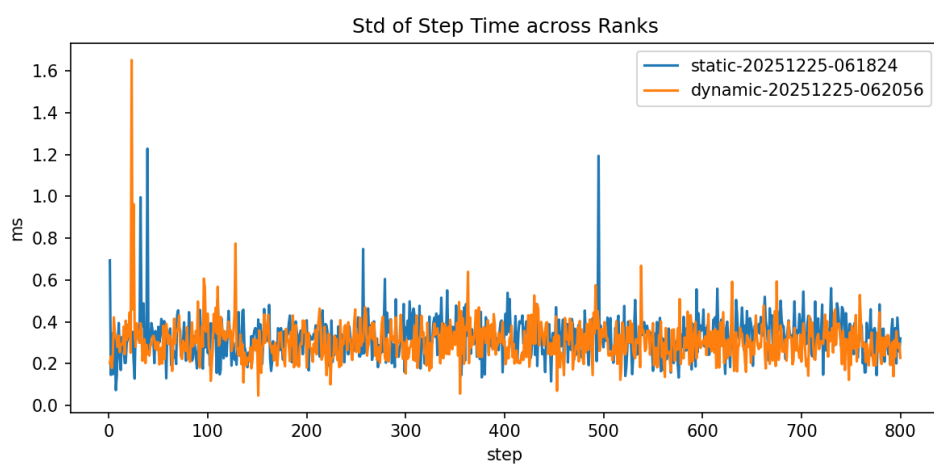


图5 跨卡 step time 标准差

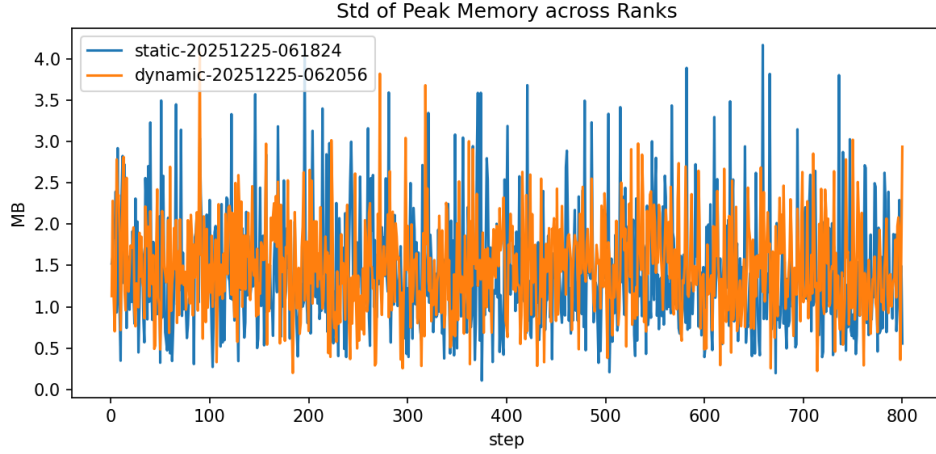


图 6 跨卡 peak memory 标准差

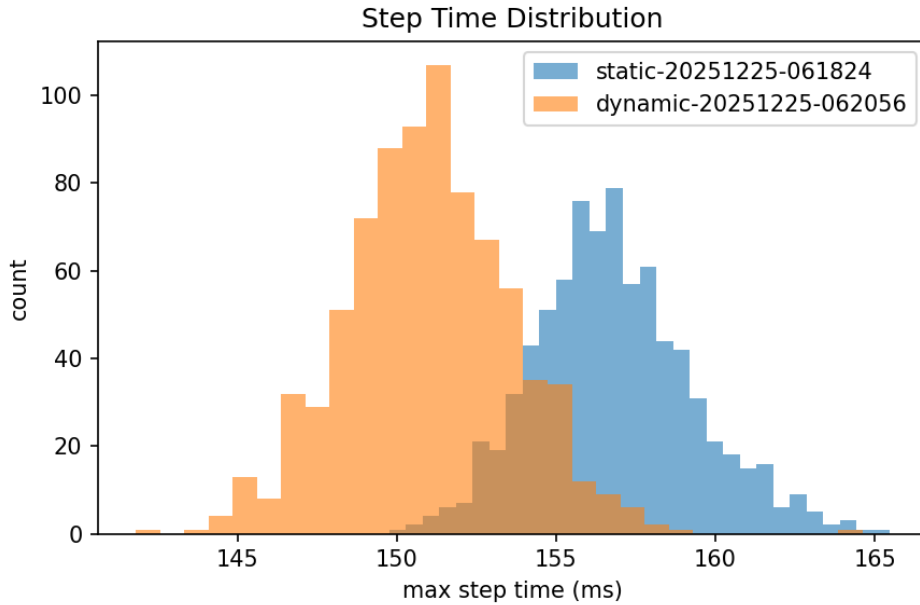


图 7 step time 分布 (static vs dynamic)

4.6 失败设置与经验总结

在更激进的设置（如 `cost=tokens_sq`）下，虽然 padded tokens 的均衡性更好，但 step time 反而变差，说明代理成本与真实耗时之间仍存在偏差；调度策略需在“均衡性”与“调度开销”之间权衡。此外，过小的 buffer size 会降低成本估计精度，导致分配波动。该部分的负结果为后续改进提供了实践依据。

五、结论与亮点

本文针对 DDP 变长序列训练的负载不均衡问题，完成了“建模—求解—实验验证”的闭环。实验结果表明，动态 per-GPU micro-batch 调度在保持训练稳定的同时显著改善了跨卡均衡性，并提升了 step time 与吞吐表现。

亮点与心得总结：

1. 以真实工程问题为载体，完整呈现数学规划建模与求解过程；
2. 给出 min-max 整数优化的 exact 枚举解，并引入方差 tie-break 强化均衡性；
3. 推导并实现 DDP 梯度缩放，使动态 batch 与全局平均梯度等价；
4. 记录失败配置，分析代理成本与真实耗时的偏差来源；
5. 总结 PyTorch DDP、Transformers、数据缓存与离线运行的工具使用经验。

局限与展望

当前工作仍存在局限：代理成本未显式考虑通信与 kernel 融合；枚举方法仅适用于小规模 GPU；模型与数据规模较小。未来可考虑：引入更精细的成本模型，研究近似优化或启发式调度，并拓展到更大规模训练。

参考文献

- [1] PyTorch Team. Distributeddataparallel —pytorch documentation[EB/OL]. 2024. <https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html>.
- [2] VASWANI A, SHAZEER N, PARMAR N, et al. Attention is all you need[EB/OL]. 2017. <https://arxiv.org/abs/1706.03762>.
- [3] Hugging Face. Hugging face datasets documentation[EB/OL]. 2024. <https://huggingface.co/docs/datasets>.
- [4] Hugging Face. Transformers documentation[EB/OL]. 2024. <https://huggingface.co/docs/transformers>.
- [5] WANG A, SINGH A, MICHAEL J, et al. Glue: A multi-task benchmark and analysis platform[EB/OL]. 2018. <https://gluebenchmark.com/>.
- [6] SANH V, DEBUT L, CHAUMOND J, et al. Distilbert: a distilled version of bert[EB/OL]. 2019. <https://arxiv.org/abs/1910.01108>.