



预训练与搜索：利用预训练神经成本模型实现高效嵌入表分片

Daochen Zha [1] **Louis Feng** [2] **Liang Luo** [2] **Bhargav Bhushanam** [2] **Zirui Liu** [1] **Yusuo Hu** [2] **Jade Nie** [2] **Yuzhen Huang** [2] **Yuandong Tian** [2] **Arun Kejariwal** [2] **Xia Hu** [1]

arXiv:2305.01868v1 [cs.LG] 3 May 2023

摘要

在分布式训练中，将大型机器学习模型分片到多个设备上以平衡成本至关重要。这项任务颇具挑战，因为分区属于 NP难问题，而且难以准确、高效地估算成本。在这项工作中，我们探索了一种“预训练，再搜索”高效分片范式。其核心思路是预训练一个通用且一次训练即可适用所有场景的神经网络，以预测所有可能分片的成本，从而构建高效的分片模拟器。基于这个预训练成本模型，我们随后进行在线搜索，以针对任意特定分片任务找出最佳分片方案。我们在深度学习推荐模型（深度学习推荐模型）中实践了这一思路，并提出了用于嵌入表分片的 NeuroShard。NeuroShard 在增强表上预训练神经成本模型，以覆盖各种分片场景。随后，它分别使用束搜索和贪心网格搜索，找出最佳的按列分片方案和按表分片方案。实验表明，在基准分片数据集上，NeuroShard 的表现显著且始终优于最先进水平，改进幅度最高达 23.8%。在采用多 TB 嵌入表的超大型生产级深度学习推荐模型中部署时，NeuroShard 在嵌入成本方面比最先进水平降低 11.6%，相当于端到端训练吞吐量提升 6.6%。为促进系统机器学习领域未来对“预训练，再搜索”范式的研究，我们已在以下地址开源代码：https://github.com/daochenzha/neuroshard

1 引言

深度学习推荐模型（DLRMs）是最重要的机器学习应用之一（Zhang 等人，2019年；Cheng 等人，2016；Naumov 等人，2019年；Tan 等人，2023b；Zhou 等人，2023年）。例如，DLRMs 占 Meta 训练需求的 50% 以上和推理需求的 80%（Naumov 等人，2020年；Gupta 等人，2020年）。DLRMs 面临的一项挑战是如何处理稀疏的类别特征。例如，YouTube 推荐模型中的单个类别特征包含数千万个视频 ID（Covington 等人，2016）。为了处理类别特征，现代 DLRMs 使用嵌入表，这些表是将类别索引映射到向量的哈希表。

遗憾的是，在生产规模的深度学习推荐模型中，嵌入表往往是存储和效率方面的瓶颈。一方面，嵌入表可能非常庞大。例如，Meta 深度学习推荐模型中的嵌入表需要数太字节内存（Acun 等人，2021；Mudigere 等人，2022年）。因此，现代分布式训练系统通常必须采用模型并行性，将表分区并放置在不同设备上，例如 GPU 和 CPU，用于深度学习——推荐模型（Acun 等人，2021；Covington 等人，2016；Liu 等人，2017年；Gomez-Uribe 和 Hunt，2015年；Lian 等人，2022年）。另一方面，嵌入表通常会带来巨大的计算和通信成本。例如，有报告指出，在 Meta 的一个深度学习推荐模型中，嵌入表占总计算量的 48%，占总通信成本的 65%（Zha 等人，2022b）。

图 1 左侧展示了深度学习推荐模型典型分布式训练工作流中嵌入表的计算和通信成本[1]（Naumov 等人，2019年）。该工作流结合了模型并行（即将嵌入表分区并放置到多个 GPU 上）和数据并行（即复制全连接层并对训练数据进行分区）。在前向传播中，每个 GPU 使用其稀疏特征向其他 GPU 发起查询，以从其他 GPU 上的表中查找嵌入（前向计算），并通过全互联通信获取嵌入（前向通信）。在反向传播中，梯度通过另一次全互联通信发送回各 GPU（反向通信），并应用于嵌入（反向计算）。

[1]莱斯大学计算机科学系，美国 [2]Meta Platforms, Inc.，美国。通信作者：Daochen Zha<daochen.zha@rice.edu>。

论文集 6^{th} *MLSys* 会议，美国佛罗里达州迈阿密，2023年。
版权所有 2023年，作者所有。

[1]本研究聚焦于图形处理器设备之间的分片。未来我们将研究 CPU 分片或 CPU-GPU 混合分片场景。

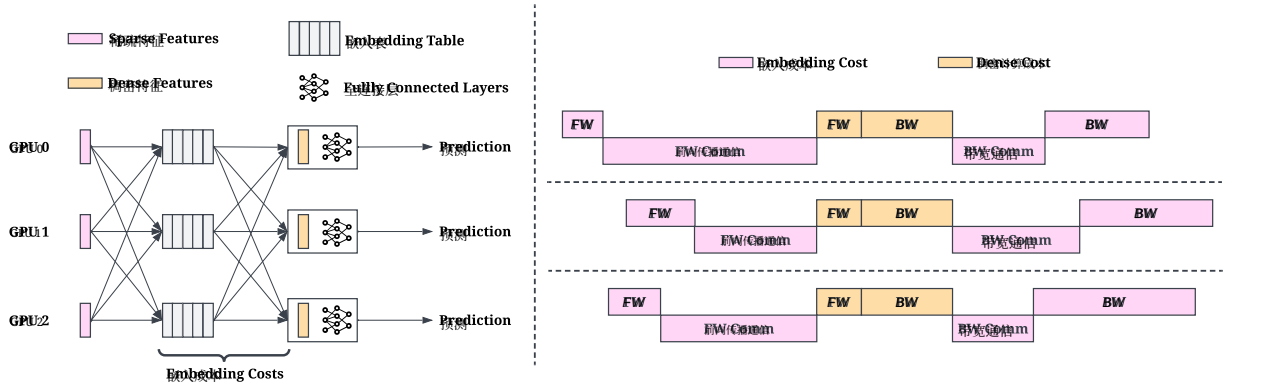


图*1.* 左：深度学习推荐模型（Naumov 等人，2019年）在三块 GPU 上的分布式训练工作流示意图。右：全同步模式下典型的 GPU 跟踪记录（Mudigere 等人，2022年）。对于每块 GPU，顶部和底部线程分别用于计算和通信，且两者可以重叠。FW 和 BW 分别表示前向和反向计算。FW Comm 和 BW Comm 分别表示前向和反向通信。我们仅展示主要成本；其他成本通常可以忽略，或因重叠而被隐藏。由于前一训练迭代中嵌入反向操作（粉色 BW）的结束时间各不相同，因此各 GPU 上的嵌入前向操作（粉色 FW）通常不会同时开始。

一个会显著影响嵌入成本的重要设计因素是嵌入表分片，即对嵌入表进行分区和放置的策略。如果分区不当，嵌入表很容易造成不均衡，使某些设备的计算和通信成本显著高于其他设备，从而在同步训练环境中导致拖尾效应（Mudigere 等人，2022年）。尽管已有一些基于启发式的分片策略被提出（Acun 等人，2021；Lui 等人，2021），但它们依赖过度简化的成本函数，因此分片性能往往不尽如人意。

近年来，强化学习（RL）在嵌入表分片方面展现出良好前景（Zha 等人，2022a；b）。其思路是将分片转化为一个优化问题，旨在确定能够使总体成本最小化的分片方案。这些方法将分片建模为马尔可夫决策过程（MDP），其状态和奖励是由神经网络估算的计算与通信成本。然后，它们训练另一个策略网络来求解该 MDP，从而使总体嵌入成本最小化。这些基于学习的方法相较于启发式分片策略取得了显著改进（Zha 等人，2022a；b）。

尽管基于强化学习的方法取得了成功，但很难将其部署。**1)**它们只考虑按表分片，即将表视为分片中的最小单位，并专注于如何将每个表分配到设备上。然而，一个表很可能极其庞大或成本高昂，从而成为内存和计算瓶颈。采用这些方法可能导致内存溢出错误或不理想的负载均衡。**2)** 策略网络通常只在极少量的分片任务上进行训练，因此，为了处理未见过的任务，需要频繁地重新训练。**3)** 强化学习中的策略因方差很高而出了名地不稳定（Henderson 等人，2018年；Zha 等人，2019年；2021d；Lai 等人，2020a）；也就是说，即使我们在同一个马尔可夫决策过程上通过多次独立运行训练同一个策略，其中一些运行可能表现良好，而其他运行则可能失败。然而，在生产环境中，我们通常需要稳定的分片解决方案。

受大语言模型中“预训练、提示和预测”近期成功的启发（Liu 等，2023年；Brown 等，2020年；Touvron 等，2023年；Chuang 等，2023年；Tang 等，2023年），我们探索了一种用于高效分片的“预训练和搜索”范式，并提出了 NeuroShard，如图2所示。为处理超大规模或高成本表，我们将按列分片纳入优化过程，在该过程中，一个表可以被划分为两个更小的表，每个表的列数都是原表的一半。不像以随机方式训练策略网络的基于强化学习的方法，我们在增强数据上预训练通用神经成本模型，以覆盖全面的分片场景。训练完成后，成本模型充当模拟器，能够高效估计任意分片方案的嵌入成本。基于预训练成本模型，NeuroShard 分别利用束搜索和贪心网格搜索确定最佳的按列分片方案和按表分片方案。NeuroShard 不仅超越了最先进水平，而且还可以轻松部署到生产环境中。总而言之，我们做出了以下贡献。

- 我们对嵌入表的计算成本和通信成本进行了全面分析。我们观察到：**1)** 按列对表进行分区会增加总体成本，因此按列分片需要在总体成本与均衡性之间进行权衡；**2)** 多表计算成本与单表成本之和呈非线性相关；**3)** 通信成本主要由表维度决定。

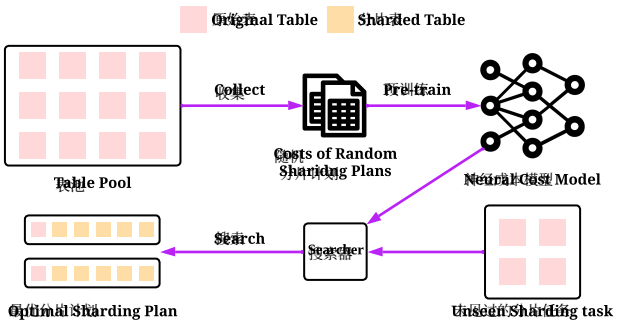


图 *2*。NeuroShard 通过“预训练，再搜索”找出最优分片方案。

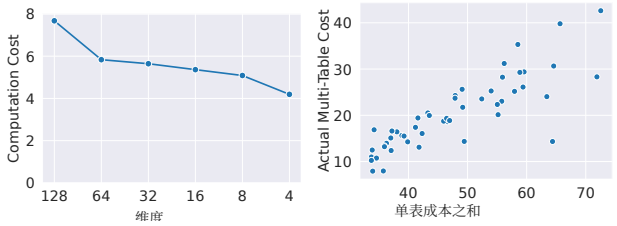


图*3*。左：不同维度下的计算成本。我们观察到其他表也呈现类似模式（见附录 A.1）。右：单表成本之和与实际多表成本的对比。

- 受这些观察启发，我们设计了 NeuroShard，这是一种“预训练并搜索”框架，可基于预训练的成本模型搜索按列分片和按表分片方案。不同于成本模型覆盖范围有限的基于强化学习的方法，我们的成本模型经过面向一次性全覆盖目标的训练，即训练一个适用于所有分片任务的通用成本模型。此外，我们的成本模型还能够直接支持按列分片，因为它是在包含各种表维度的扩展表池上训练的。

- 在基准分片数据集上，NeuroShard 相较于最先进水平最多提升 23.8%，且不会发生内存溢出（Naumov 等人，2019年）[2]。

- NeuroShard 已部署于一款超大型生产级深度学习推荐模型中，可将多 TB 的嵌入表分片到数百个 GPU 上。NeuroShard 将嵌入成本降低了 11.6%，从而使端到端训练吞吐量提升了 6.6%。鉴于该生产级深度学习推荐模型已经过充分优化，这一加速效果意义重大。

2 理解嵌入成本

本节分析嵌入成本，以此说明分片算法设计的动机。图 1右侧展示了一个典型训练迭代的跟踪记录，其中主要包括嵌入的计算/通信以及全连接层的计算。请注意，图中未显示其他一些成本，例如嵌入查找前稀疏特征的通信，以及全连接层的权重同步等。这些成本通常可以忽略，或者由于计算与通信重叠而被隐藏。

我们首先通过跟踪记录示例说明为什么负载不均衡会导致更高的嵌入成本。在 GPU 1 的跟踪记录中，嵌入反向通信和计算耗时超过其他 GPU，导致它在下一次迭代中的嵌入前向计算比其他 GPU 晚开始。GPU 1 的嵌入前向计算耗时又再次超过其他 GPU。因此，上述延迟最终不断累积，迫使 GPU 1 比其他 GPU 晚得多地开始嵌入前向通信。这种负载不均衡导致 GPU 0 和 GPU 2 出现明显的空闲时间。

为减少累积延迟，我们需要平衡所有 GPU 上嵌入相关的计算成本和通信成本。下文将分别分析这些成本。所有结果均使用 2080Ti GPU 和 FBGEMM 的现代嵌入实现收集（Khudia 等人，2021），该实现将多个表查找融合为单个操作。附录 A提供了所有分析实验的更多详情。

2.1 计算成本分析

计算成本主要由表格配置和查找索引决定。一些重要因素被用于量化计算成本（Zha 等人，2022年a）。**1) 维度：**表格的列数。维度越大，通常意味着计算成本越高，因为这会直接增加内存带宽的使用量。**2) 哈希大小：**表格的行数。它会通过影响缓存/预取行为，间接影响计算成本。**3) 池化因子：**一次查找中的嵌入索引数量。我们通常会计算一批索引的平均池化因子。平均池化因子越高，计算成本通常也越高，因为它决定了查找的工作负载。**4) 索引分布：**查找的访问模式和分布，即某些索引的访问频率可能高于其他索引。它会通过影响缓存有效性间接影响成本。此外，一批数据中访问的唯一嵌入数量也会影响缓存。访问的索引越少，通常成本越低。

[2]https://github.com/facebookresearch/ dlrm 数据集_

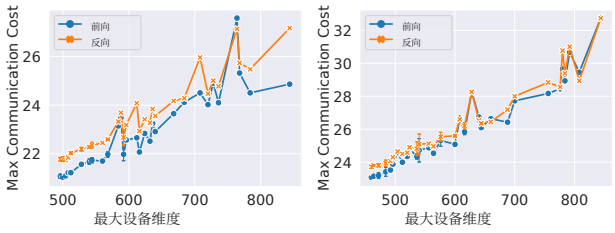


图4。4 个 GPU（左）和 8 个 GPU（右）中，最大前向/反向通信成本与最大设备维度的关系。

前述分析表明，实际计算成本与上述因素之间存在复杂的非线性关联（Zha 等人，2022a），这使得成本估算十分困难。我们在此进行两项分析实验，以加深理解，并为按列分片和按表分片的算法设计提供依据。

首先，我们研究维度的影响。我们从名为 DLRM（Naumov 等人，2019年）[2]的基准分片数据集中随机选择一个表。图3 左侧展示了其在不同维度下的计算成本（前向 +反向）。如 $\{128, 64, 32, 16, 8, 4\}$预期，维度越大，计算成本越高。然而，我们还观察到以下现象。

观察1 将一个表按列分成两半时，每个分片的计算成本都大于原表计算成本的一半。

例如，维度为 64 时的成本远大于维度为 128 时成本的一半。这可以通过并行性和算子融合来解释；与单独执行每个操作相比，融合后的嵌入表操作能够在 CUDA内核 中实现更好的优化。这也表明了按列分片中的一种权衡：虽然将表划分为更小的表可以改善负载均衡，但可能会增加总体计算成本。因此，按列分片算法需要在负载均衡与总体计算成本之间取得平衡。

接下来，我们研究能否用各单表成本之和来估算多表操作的计算成本。这一点很重要，因为如果这一假设成立，我们就可以借助混合整数线性规划等工具来均衡计算成本（Sethi 等人，2022年）。我们从 DLRM 数据集中随机抽取 50 个表子集，每个子集包含 10 个表。然后，我们在图 3右侧绘制它们之间的关系。

观察2 多表计算成本与各单表成本之和之间呈非线性关系。

显然，按表分片算法必须考虑多表成本的非线性。

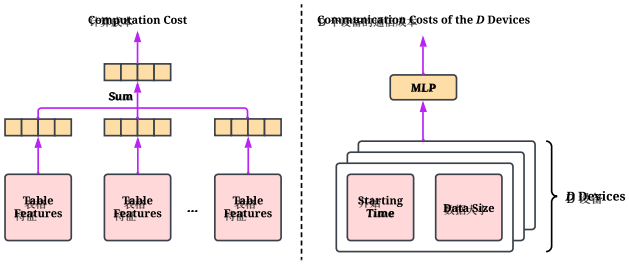


图 5. 计算成本模型（左）和通信成本模型（右）的神经网络架构。

2.2 通信成本分析

在分布式训练中，每对 GPU 都需要在前向和反向传播过程中进行通信。因此，每个 GPU 的通信延迟可能各不相同。我们的目标是尽量降低所有 GPU 中的最大通信成本，因为速度最慢的 GPU 会成为瓶颈。直观来说，通信成本取决于各 GPU 上待发送数据的大小，而 GPU 的数据量可通过批大小与设备维度的乘积来估算。设备维度定义为该设备上各表维度之和。由于所有 GPU 的批大小都相同，设备维度便成为决定通信均衡的关键因素。

我们通过实验研究最大设备维度与最大前向/反向通信成本之间的关系。我们从 DLRM 数据集中随机抽取一个表子集（4 个图形处理器时抽取 40 个表，8 个图形处理器时抽取 80 个表），并为每个表从 $\{128, 64, 32, 16, 8, 4\}$中随机选择一个维度。然后，我们在最大设备维度各不相同的情况下，将这些表分片到 4 个或 8 个图形处理器上。我们在图4中对 50 种分配方案进行了基准测试。

观察3 所有 *GPU* 中的最大前向/反向通信成本与所有 *GPU* 中的最大设备维度呈正相关。

上述观察促使我们采用一种平衡通信成本的替代方法，即最小化所有 GPU 中的最大设备维度。平衡设备维度要容易得多，因为设备维度就是各表维度之和。

3 NeuroShard 框架

受上述三个观察结果的启发，我们提出了 NeuroShard，这是一种基于预训练神经成本模型和在线搜索的嵌入表分片框架。图6展示了其工作流程。核心思路是预训练神经网络，以预测计算成本和通信成本，并将其用作分片模拟器，从而快速估算任意分片任务的嵌入成本以及

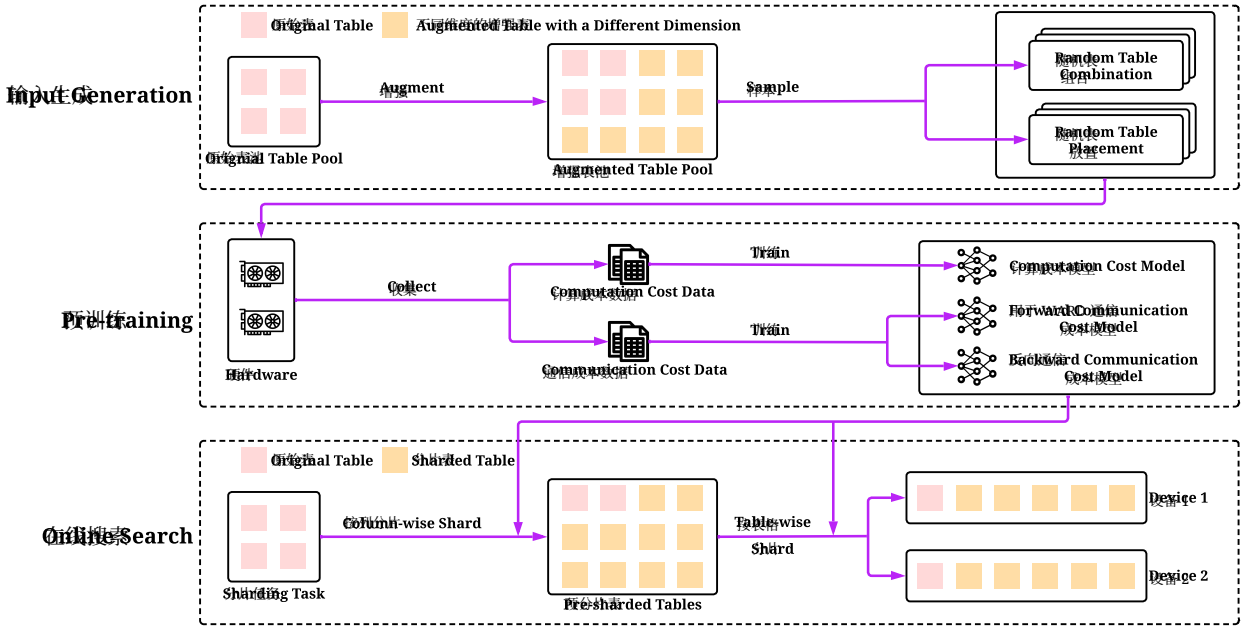


图6。NeuroShard 的整体工作流程。我们首先基于增强表生成随机输入（上排）。然后运行微基准测试以收集成本，并预训练三个神经网络来预测成本（中排）。最后，基于预训练成本模型执行在线搜索，以进行嵌入表分片（下排）。

任意分片方案。然后，我们在预训练成本模型上执行在线搜索，无需实际运行图形处理器即可确定最佳分片方案。接下来，我们将介绍如何生成合成输入以收集训练数据（第3.1节），以及如何训练神经网络进行成本预测（第3.2节）。成本模型预训练完成后，我们将介绍如何执行在线搜索，以优化按列分片和按表分片（第3.3节）。

3.1 生成合成输入

预训练主要是一种以数据为中心的过程（Zha 等人，2023b；a），其中高质量的成本数据发挥着至关重要的作用。本小节将介绍如何生成表输入，使其能够覆盖不同的表组合和放置方式，以对计算成本和通信成本进行基准测试，其中包括表增强、随机表组合生成和随机表放置生成。需要注意的是，为了实现最佳覆盖范围，生成策略应考虑模型训练所使用的基础设施以及模型中的嵌入表。例如，如果模型包含大量嵌入表，但只使用数量非常有限的 GPU，则生成的输入应覆盖包含大量表的表组合。下面我们主要讨论如何实现良好覆盖范围的高层次策略。我们将在第4节中，以 DLRM 数据集为例介绍该策略的具体实现。

表格增强。在深度学习推荐模型中，每个嵌入表都对应一个稀疏特征，这些特征通常由用户或物品收集。在实际应用中，我们通常会有一个嵌入表池，机器学习工程师会在其中进行特征选择，即从表池中选取一部分嵌入表用于模型训练。在此过程中，表格的维度可能会调整。此外，按列分片也会生成维度各异的新表。因此，生成的表格输入应能够涵盖不同维度的表。为此，我们会进行表格增强。具体而言，我们会为每个表生成维度各异的增强表。例如，假设原始表的维度为 64，我们可以生成 5 个维度分别为 128、32、16、8 和 4 的增强表，以适应模型设计和按列分片中可能进行的维度调整。表格增强会生成一个增强表池，用于数据生成。附录 B.1 总结了详细的增强流程。

随机表组合生成。 这些表组合将用于对计算成本进行基准测试。生成的组合应覆盖 GPU 上不同数量的表。为此，我们首先在特定范围内均匀采样表的数量 T，然后从增强表池中随机选择一个包含 T 个表的子集。附录B.2 总结了详细的生成过程。

随机表放置生成。 这些表的放置方案将用于评测通信成本。生成的放置方案应涵盖设备维度的不同均衡度以及不同的通信

起始时间戳。 **1) 为了模拟不同的均衡度，我们采用一种带有随机性的贪心策略**。 具体而言，我们首先从表池中随机抽取一个子集，并按维度降序排列这些表。从维度最大的表开始，以 p 的概率，其中 p 对每次表放置均匀采样于 $[0,1]$ ，我们将当前表分配给迄今为止表维度总和最低的图形处理器；以 $(1-p)$ 的概率，则随机分配当前表。这里， p 可以间接控制均衡度。当 $p=1$时，该策略会在每一步贪心地平衡维度，因此分片方案能够很好地平衡各维度。当 $p=0$时，分片方案将是随机的，因此各维度很可能会失衡。由于 p 是随机选择的，我们可以覆盖均衡度各不相同的分片方案。 **2) 我们随机生成每个图形处理器的通信起始时间戳**。 回顾轨迹分析（图 1右侧）可知，先前操作造成的延迟会累积，使前向通信的开始时间明显晚于其他图形处理器。显然，在评测通信成本时，我们也需要考虑这些延迟，因为各图形处理器上的通信若非同时开始，其行为可能会有所不同。为了模拟这些延迟，我们在一定范围内为每个图形处理器随机选择一个起始时间戳。附录 B.3 总结了详细的生成流程。

3.2 NeuralCostModels 预训练

给定生成的输入，我们会运行微基准测试，例如 PARAM 基准测试[3], , 以收集实际的计算成本和通信成本，用于成本模型训练。附录C 提供了有关成本模型训练的更多细节。

图5 展示了计算成本模型和通信成本模型的神经网络架构。 **1）计算成本模型：** 该架构遵循（Zha 等人，2022年a）。具体而言，每个表格都由一些特征表示，包括维度、哈希大小、池化因子和索引分布。给定一个表格组合，我们使用共享的多层感知机处理所有表特征，以获得表表示。通过对所有表表示执行逐元素求和，我们获得表格组合的固定维度表示。最后，我们使用另一个多层感知机生成计算成本（前向 + 反向）。 **2 ）通信成本模型：** 我们使用多层感知机，根据起始时间戳和传输数据大小，预测所有 GPU 的通信成本。我们分别为前向和反向通信训练两个独立的模型。

神经成本模型的部署。在真实世界的深度学习-推荐系统中，索引分布可能会随时间发生变化。因此，我们可能需要重新训练/微调并重新部署成本模型，以应对潜在的变化。为确保重新部署过程顺利进行，我们通常需要严格的版本控制，以确保每个训练作业始终关联同一版本的成本模型。这对于检查点保存尤为重要，因为恢复训练时需要保持分片方案一致。我们发现，在生产环境中，每三个月重新训练一次就足够了。我们也可以定期抽取一批表索引来计算成本模型的预测误差，并在误差超过特定阈值时触发重新训练或微调。请注意，通常只有在索引分布发生变化时才需要重新训练。表维度发生变化时无需重新训练，因为表扩增已涵盖各种表维度。

[3] https://github.com/facebookresearch/ 参数

3.3 在线搜索

预训练成本模型可作为嵌入表分片的通用模拟器。通过汇总预测的计算、前向通信和反向通信成本，它们能够高效估算任意分片任务中任意分片方案的嵌入成本。本小节将介绍如何利用此模拟器，通过搜索来最小化嵌入成本。图7展示了整体概览。在外层循环中，我们通过束搜索来搜索按列分片方案。在内层循环中，我们通过贪心网格搜索，为按表分片找到最佳的最大维度约束。本文介绍的搜索算法主要受到第2节中的观察结果启发。

优化问题形式化。我们将按列分片方案记为 $\mathbf{c}=[c_1,c_2,...,c_m]$，其中$c_i$ 表示在第 i 步，将索引为 c_i 的表按列分成两半，并将生成的新表追加到表列表末尾。令 $\mathbf{t}=[t_1,t_2,...,t_T]$ 表示按表分片方案，即将 T 个表分配到 D 个 GPU 设备，其中 $t_i \in \{1,2,...,D\}$。$\mathbf{t}$ 取决于 $\mathbf{c}$，因为 $\mathbf{t}$ 是在按列分片后的表上进行操作。$\mathbf{c}$ 和 $\mathbf{t}$ 都必须满足一些约束。例如，FBGEMM（ Khudia 等，2021）中的嵌入操作要求维度必须能被 4 整除。对于 $\mathbf{c}$，分片方案必须满足 GPU 内存约束。我们分别将它们的合法方案空间记为 $\mathcal{C}$ 和 $\mathcal{T}$，其中 $\mathcal{T}$ 取决于所选的 $\mathbf{c}$。目标是

$$\arg\min_{\mathbf{c}\in\mathcal{C},\mathbf{t}\in\mathcal{T}} f(\mathbf{c},\mathbf{t}). \tag{1}$$

$f(\mathbf{c},\mathbf{t})$ 是模拟的嵌入成本。$\mathbf{t}$ 对$\mathbf{c}$ 的依赖关系，自然使得搜索$\mathbf{c}$ 成为外层循环，而搜索$\mathbf{t}$ 成为内层循环。

采用束搜索进行按列分片。 按列分片可以移除超大表和高成本表，从而实现更好的均衡。不过，根据观察1，按列分片

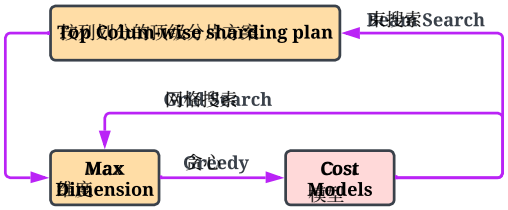


图 7. 搜索过程。外层循环使用束搜索，寻找最佳按列分片方案。内层循环通过网格搜索，重点处理贪心分配的最大维度约束。

按列分片会增加总体计算成本。因此，理想的分片方案应能在最少步骤下实现平衡。我们提出一种束搜索策略来缩小搜索空间。其核心思想是，我们通常只需对规模较大且计算成本较高的表进行按列分片。具体而言，在每次迭代 i 中，我们选出成本最高的前 N 个表，以及规模最大的前 N 个表作为候选表（去除重复项）。同样，我们只考虑上一步找到的前 K 个最佳按列分片方案。对于前 K 个方案中的每一个，我们都将每个候选表加入其中，以得到一个新的分片方案，并运行内层循环来计算成本。然后，我们再次选出前 K 个最佳新分片方案，用于下一步分片。我们执行 L 步分片，并输出成本最低的分片方案。 L、K 和 N 是用于平衡最优性与效率的超参数。

使用贪心网格搜索进行按表分片。给定一组按列分片的表，我们介绍如何在内层循环中执行按表分片。我们提出一种网格搜索策略，基于两个想法寻找计算成本与通信成本之间的最佳均衡。1）受观察2启发，我们提出一种贪心算法，以均衡多表计算成本。2）受观察3启发，我们将最大设备维度设为贪心算法的约束，以实现通信均衡，并通过网格搜索确定最佳最大设备维度。具体而言，在每一步中，我们执行以下操作：1）选择一个最大设备维度 $max\ dim$。2）根据成本模型预测的计算成本，按降序排列各表。3）从成本最高的表开始，逐个将表分配给当前设备成本最低的设备，同时满足内存和$max\ dim$ 约束；设备成本由成本模型预测。4）使用成本模型评估嵌入成本。我们按如下方式对 $max\ dim$ 进行网格搜索：给定起始值 M_s、结束值 M_e和总步数 M，以 $(M_e - M_s)/(M-1)$为步长，尝试 $[M_s, M_e]$ 中的所有值。根据经验，我们将 M_s 固定为各设备上的平均维度，并将 M_e 固定为 $1.5 * M_s$。 M是用于控制搜索粒度的超参数。

带缓存的实现。 搜索中成本最高的部分是预测计算成本。它需要

算法 1束搜索

输入： T 嵌入表，束搜索超参数 hyerperparameters（N、 K 和 L）
全局最佳按列分片方案$\mathbf{c}^* \leftarrow []$
全局最佳按表分片方案 $\mathbf{t}^* \leftarrow NULL$
最佳按列分片方案 $\mathcal{C}$ $_p \leftarrow \{[]\}$
初始化全局缓存 $global\ cache$
对于 $outer\ loop$ = 1、2、...、 L **执行**
  下一步中的按列分片方案 $\mathcal{C}'_p \leftarrow []$
  对于每个方案$\mathbf{c}_P$， 在 $\mathcal{C}_p$ **中执行**
    根据$\mathbf{c}_P$， 通过合并获取候选表——成本最高的前 N 个表以及前 N 个表移除重复项后大小最大的
    对于每个候选表格 t **执行**
      $col_plan \leftarrow \mathbf{c}_P$ 并在末尾追加 t
      $cost, \mathbf{t} \leftarrow$ GreedyGridSearch($global_cache$, col_plan)
      将 ($col\ plan$,$cost$) 附加到 $\mathcal{C}'_p$。
      如果观察到较低的 $cost$**那么**
        $\mathbf{c}^* \leftarrow col_plan$
        $\mathbf{t}^* \leftarrow \mathbf{t}$
      结束 if
    结束循环。
  结束循环
  $\mathcal{C}_p \leftarrow$ 成本最低的前 K 个方案，位于 $\mathcal{C}'_p$
结束循环
返回$\mathbf{c}^*$, $\mathbf{t}^*$

调用 $O(LKNMTD)$次（T是表的数量，而 D 是 GPU 的数量），其中每次调用都需要对成本模型进行一次前向传播。幸运的是，我们发现搜索过程中存在大量重复调用。这是因为，如果我们只对按列分片方案或 $max\ dim$做微小调整，那么在大多数步骤中，成本模型很可能会被要求预测同一组表的成本。因此，我们可以自然地使用一个生命周期内持续存在的哈希映射作为缓存来加速。实践中，缓存命中率可达 95% （见表 3 ）。我们在算法 1 中总结了束搜索，并在算法2中总结了贪心网格搜索。

4 实验

实验旨在回答以下研究问题。 **研究问题 1:** NeuroShard 与最先进的分片算法相比表现如何（第 4.1 节）？ **RQ2:** 神经成本模型的准确度如何（第4.2 节）？ **RQ3:** 每种搜索设计对性能有何贡献（第4.3 节）？ **RQ4:** 超参数如何影响性能（第 4.4 节）？ **RQ5:** NeuroShard 能否提升端到端训练吞吐量（第4.5 节）？

数据集。 公开的大型数据集（例如 Criteo、Avazu 和 KDD）通常无法匹配工业规模的数据，因此

表 *1.* NeuroShard 与基线方法在 4 个或 8 个 GPU、最大表维度为 4 至 128 时的嵌入表成本（100 个随机构造的分片任务的平均值，单位为毫秒）。排名第一和第二的结果分别以粗体和下划线标出。“-”表示该方法无法扩展，即 100 个任务中至少有一个遭遇内存爆炸。底行总结了 NeuroShard 相较于最强基线方法的改进。

类别	方法	4 个 GPU						8 个 GPU					
		4	8	16	32	64	128	4	8	16	32	64	128
随机	-	25.47	30.10	-	-	-	-	33.40	36.43	-	-	-	-
贪心	基于大小	24.45	29.60	30.86	37.80	41.59	-	31.75	34.30	37.70	46.07	54.57	-
	基于维度	23.51	28.46	29.76	35.98	38.71	-	27.54	32.20	34.78	42.35	47.54	-
	基于查找	18.69	24.34	26.83	34.62	38.69	-	21.83	26.66	30.59	<u>39.07</u>	47.47	-
	基于大小查找	18.38	24.18	26.81	33.94	-	-	21.27	26.55	30.34	39.28	48.17	-
强化学习	AutoShard	<u>17.99</u>	<u>22.08</u>	-	-	-	-	<u>20.79</u>	-	-	-	-	-
学习	DreamShard	18.78	22.59	25.08	30.74	-	-	21.40	25.30	26.90	-	-	-
规划	TorchRec	19.20	26.00	28.24	34.88	<u>38.13</u>	<u>47.22</u>	22.34	28.99	32.71	39.90	<u>47.43</u>	60.58
成本建模	NeuroShard	**17.74**	**21.75**	**23.11**	**28.86**	**31.55**	**39.99**	**20.68**	**23.23**	**25.64**	**32.30**	**38.30**	**49.10**
NeuroShard 的改进		+1.4%	+1.5%	+8.5%	+6.5%	+20.9%	+18.1%	+0.5%	+8.9%	+4.9%	+21.0%	+23.8%	+23.4%

不适合用于评估分片算法。更多讨论请参见附录 D。遵循先前的工作（Zha 等，2022a；2022b），我们使用 DLRM 数据集（Naumov 等，2019）[2]，，其中包含 856 个合成表，其索引分布与 Meta 生产环境中的工作负载相似。这 856 个表构成表池。我们尽可能多样化地构建合成分片任务，以测试分片算法在不同场景下的表现。我们考虑两组分片任务，目标分别是将表分片到 4 个和 8 个 GPU 上，并且每个 GPU 对嵌入表有 4 GB 的内存限制。对于每个表，我们从 $\{4, 8, ..., 2^j\}$ 中随机采样一个维度，其中$2 \leq j \leq 7$，而 2^j 指定了表可能的最大维度。更大的 2^j 会使分片任务更具挑战性，因为表将具有更加多样的维度和更大的规模。给定 GPU 数量 d 和最大维度 2^j，我们通过从表池中随机选择 T 个表来采样一个分片任务，其中，对于 4 个 GPU，$10 \leq T \leq 60$；对于 8 个 GPU，$20 \leq T \leq 120$。对于每一对 d 和 2^j，我们随机构建 100 个分片任务。我们在附录 D 中详细介绍了分片任务的生成过程。在第 4.1 节中，我们考虑上述所有d-2^j 对。对于其他实验，我们主要关注最大维度为 128 且使用 4 个 GPU 的情况。此外，我们还考虑了生产模型中的一个真实分片任务，其目标是将数百个表分片到 128 个 GPU 上（第 4.5 节）。

基线方法。 我们考虑了几类基线方法（详见附录E）：1）**随机**分片，2）**贪心**算法，用于平衡各种启发式成本（Acun 等人，2021；Lui 等人，2021），3）**强化学习**算法，提出于（Zha 等人，2022a；b），以及4）**TorchRec 提供的规划**算法[4]。

[4]https://github.com/pytorch/torchrec

评估协议。 对于每一对 d 和 2^j，我们应用每种分片算法，为 100 个分片任务生成分片方案，并从 GPU 收集实际嵌入成本。为了收集这些成本，我们在 GPU 上运行嵌入操作，以模拟计算和通信，并使用计时器测量每台设备所花费的时间。对于每个分片任务，我们报告各设备中的最大成本，因为最大的嵌入成本将成为瓶颈。如果某种算法生成的任一分片方案导致内存错误，则表示该算法无法扩展到 d 和 2^j 定义的设置，因此我们将性能记为“-”。如果所有分片方案均有效，我们报告 100 个任务的平均嵌入成本。对于真实生产环境中的分片任务，我们同时报告嵌入成本和端到端训练吞吐量提升。嵌入成本直接取自模型训练期间收集的跟踪记录。

实现细节。 生成合成输入时，我们通过增加维度 $\{4, 8, 16, 32, 64, 128\}$来扩充表池。我们随机选择 1 到 15 个表用于表组合生成，并选择 N_{place} 个表用于表放置生成，其中 $10 \leq N_{\text{place}} \leq 60$ 适用于 4 个图形处理器，$20 \leq N_{\text{place}} \leq 120$ 适用于 8 个图形处理器。我们为每个成本模型生成 100K 个样本。我们从 0 到 20 毫秒之间随机选择一个起始时间戳。对于在线搜索，我们设置 $N = 10$，$K = 3$，$L = 10$，以及 $M = 11$。所有实验均在配备八个 2080Ti GPU 的服务器上进行。更多细节请参见附录 E。

4.1 与最先进水平方法的比较

为回答 **研究问题 1**，我们在不同 GPU 数量和最大表维度下，将 NeuroShard 与基线方法进行比较，结果见表格1。我们得出以下观察结果。

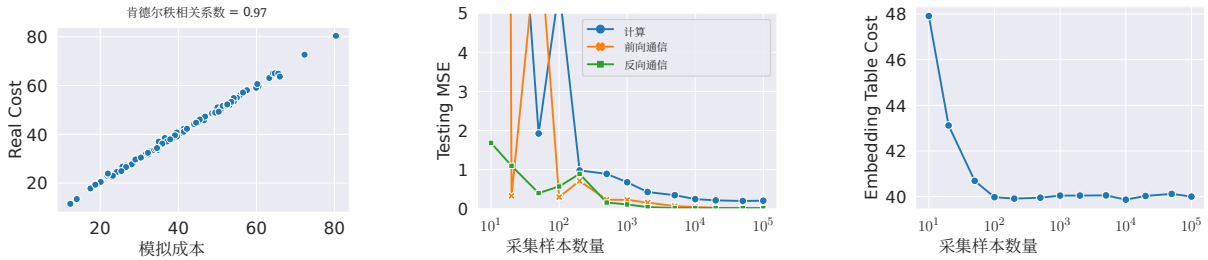


图 *8.* 左：神经成本模型估算的模拟成本与在 GPU 上测得的实际成本的散点图。中：神经成本模型的性能随训练所用样本数量的变化。右：对于最大维度为 128、使用 4 个 GPU 的分片任务，采用不同样本数量训练的神经成本模型得到的嵌入表成本。

1) NeuroShard 在所有分片任务中的表现均显著优于基线方法，改进幅度为+0.5% 至 +23.8%，这证明了 NeuroShard 的优越性。NeuroShard 相较于 AutoShard 和 DreamShard 的一项优势源于其按列分片。大型或高成本表会被划分，以避免内存爆炸或成为分片过程中的瓶颈。DreamShard 和 AutoShard 无法做到这一点。**2)**NeuroShard 能够成功扩展到表维度较高的分片任务。当表维度变大时，除 TorchRec 外，所有基线方法都容易失败。这是因为对于更大的表，要找到满足内存约束的分片方案会更加困难。与贪心方法和基于强化学习的方法不同，NeuroShard 会搜索最佳按列分片方案，从而能够划分大型表。令人意外的是，即使维度较小，基于强化学习的方法也会失败。一个可能的原因是，强化学习中的随机策略很难训练，且方差很高。**3)**基于学习的方法通常优于启发式成本函数。这是因为神经成本模型以数据驱动方式训练，因此能够提供更好的成本估计，从而提升分片性能。**4)** 尽管 TorchRec 的扩展性也很好，NeuroShard 的表现却好得多。这是因为 TorchRec 仍依赖启发式成本函数，导致其估计不准确。相比之下，NeuroShard 中的成本模型以数据驱动方式进行预训练，因此能够更准确地估计表的成本。

4.2 神经成本模型分析

为了研究**RQ2**，我们设计了实验，以了解成本模型的准确度，以及要实现良好的分片方案，成本模型需要达到怎样的准确度。首先，我们在表格2中报告了所有预训练神经网络成本模型在测试集上的均方误差（MSE）损失。我们观察到，最大的均方误差为 0.26，这表明预测误差在 0.6 毫秒以内（$0.6 \times 0.6 = 0.36 > 0.26$）。请注意，收集实际成本时可能会存在一定方差。因此，0.6 毫秒的误差已相当准确。图8 左侧绘制了 100 个随机分片方案的实际成本和模拟成本。结果再次验证成本模型的高精度。

接着，我们在图8的中部和右侧展示训练成本模型所需的样本数量。正如预期，样本越多，所有成本模型的准确度都越高。有趣的是，即使只有10^2 个样本，NeuroShard 也能取得非常出色的性能。这表明，我们只需要足够准确、但不必完全准确的成本模型。这在实际应用中很有优势，因为这意味着我们不需要很多样本。请注意，这一结果并不意味着我们可以使用更简单的模型。NeuroShard 当前的神经网络架构已经非常浅层。由于成本具有非线性，更简单的网络（即线性网络）可能无法奏效。

4.3 消融研究

为探究**RQ3**，我们在表3中报告了移除所提出的束搜索、贪心网格搜索和缓存机制之一后的结果。我们有以下两点观察。 **1**）移除束搜索或贪心网格搜索后，性能会显著下降，这表明必须同时利用二者进行联合搜索。**2** ）移除缓存机制后，分片耗时显著增加。这是因为缓存的命中率超过 95%，能够显著提升分片速度。

4.4 超参数分析

为理解 **RQ4**，我们分析 NeuroShard 中超参数的影响。回顾一下，在线搜索中有 4 个超参数用于平衡最优性与效率，即 N， K， L 和 M。我们在图 9 中展示了它们的影响。我们有两点观察。 **1**）上述四个超参数的取值越大，性能越好。这是因为取值越大，搜索迭代次数就越多。 **2**）取值越大，分片时间也越长。因此，我们应为每个超参数设定适当的取值（既不能太大，也不能太小），以平衡最优性与效率。

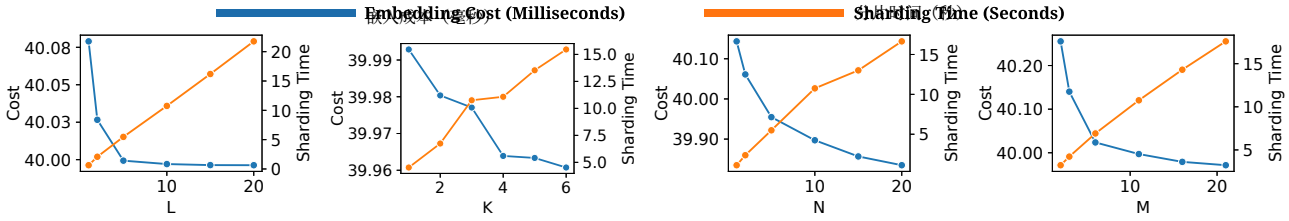


图 *9*。 NeuroShard 超参数的影响。

表格 *2*。 神经成本模型的测试均方误差。

	DLRM (4 个 GPU)	DLRM (8 个 GPU)	生产环境 (128 个 GPU)
计算	0.21	0.21	0.26
前向通信	0.02	0.05	0.05
反向通信	0.02	0.04	0.15

表 *3.* 在最大维度为 128、使用 4 个 GPU 的情况下进行的消融研究。无束搜索表示在按列分片时移除束搜索。无贪心网格搜索表示不对表格维度阈值进行网格搜索。无缓存则表示禁用计算成本的缓存机制。使用 8 个 GPU 的结果见附录 G。

	Cost (毫秒)	成功率	分片时间 (秒)	缓存命中率
不使用束搜索	-	87.0%	0.05	79.1%
无贪心网格搜索	42.90	100.0%	2.12	82.2%
无缓存	39.99	100.0%	95.87	0.0%
完整 NeuroShard	39.99	100.0%	10.75	95.4%

4.5 NeuroShard 在生产模型中的应用

为回答 **RQ5**，我们将 NeuroShard 部署到一个超大型生产级深度学习推荐模型中。我们采用了配备 RDMA 网络架构的最先进硬件平台，详情见（Mudigere 等人，2022年）。该模型有近千个需要数太字节内存的嵌入表。任务是将这些表分片到 128 个 GPU 上。我们在表 4 中比较了 NeuroShard 与基线方法的嵌入成本和训练吞吐量。由于我们发现不采用按列分片会出现内存溢出错误，因此除 TorchRec 外，我们先应用 NeuroShard 提出的按列分片方案，再运行各基线方法。与最先进水平的 DreamShard 相比，NeuroShard 将嵌入成本降低了 11.6%，从而使端到端训练吞吐量提升了 6.6%。请注意 >5% 在我们的生产模型中被视为非常显著，因为该模型已经过大量优化。此外，NeuroShard 在部署时分片表格的速度也显著更快；DreamShard 应用于该模型时需要进行强化学习训练，而 NeuroShard 无需进一步训练，因为它直接利用预训练成本模型。请注意，NeuroShard 使用部署前一个月收集的数据进行训练，因此可能

表格 *4.*NeuroShard 与基线方法在生产模型上的嵌入成本和整体训练吞吐量提升。结果收集自一个配备 128 个 GPU 的训练集群。

分片算法	嵌入成本（毫秒）	训练吞吐量改进
随机	118.3	-
基于大小	107.6	+4.0%
基于维度	90.8	+13.9%
基于查找	102.4	+11.9%
基于大小查找	109.2	+12.8%
AutoShard	86.6	+32.4%
DreamShard	61.6	+45.3%
TorchRec	86.4	+34.6%
NeuroShard	55.2	+54.9%

是表索引分布发生了变化。结果表明，在生产环境中，至少一个月内无需重新训练 NeuroShard。

在这项实验中，整个流程（收集数据 + 训练 NeuroShard）在 128 个 GPU 上大约耗时一小时。与 6.6% 的吞吐量提升相比，这点开销微不足道，因为：1）训练推荐模型可能需要长达一周，因此 NeuroShard 每次运行都能节省数小时；2）如上所述，我们无需频繁重新训练 NeuroShard。

5 相关工作

深度学习推荐模型。深度学习推荐模型已广泛应用于许多推荐场景（Zhang 等人，2019；Cheng 等人，2016；Naumov 等，2019；He 等人，2017；Wang 等人，2020；Lin 等人，2019；Chuang 等人，2020；Chang 等人，2020；Zhou 等人，2021；2022；Tan 等人，2021a；b；2020；Liu 等人，2019；Tan 等人，2019；2023a）。为在超大规模数据和模型上训练深度学习推荐模型，研究人员开发了分布式训练解决方案（Acun 等人，2021；Covington 等人，2016；Zhou 等人，2019；Liu 等人，2017；Gomez-Uribe 和 Hunt，2015）。嵌入表分片是深度学习推荐模型分布式训练中的一个重要设计因素，但在文献中鲜有研究。NeuroShard 提供了一种可部署的嵌入表分片解决方案，以提升深度学习推荐模型的分布式训练效率。

嵌入表分片。近年来的几篇论文已经

算法 2 GreedyGridSearch

输入： T 嵌入表，D GPU 设备，预训练的神经成本模型，网格搜索超参数 $M, global_cache, col_plan$
使用 $col\ plan$ 生成 T' 按列分片的表 - 其中 $T' = T + |col_plan|$
根据计算成本，将 T' 表按降序排列 由计算成本模型预测
最佳按表分片方案 $\mathbf{t}^* \leftarrow NULL$
最佳计算成本 $cost^* \leftarrow Inf$
对于 $inner\ loop$ = 1、2、.....、M **执行**
  获取 $max\ dim$，基于 M
  初始化按表分片方案 $\mathbf{t} \leftarrow []$
  对于每个 T' 表格**执行**
    获取不会导致内存错的候选 GPU- ror，其中设备维度小于 $max\ dim$ -
    对于每个候选图形处理器**执行**
      如果图形处理器中的表位于 $global_cache$ **则**
        从 $global\ cache$ 获取 $cost$
      else
        使用计算成本模型获取 $cost$ 并将成本存储到 $global\ cache$
      结束如果
      将成本最低的图形处理器追加到 $\mathbf{t}$
      如果观察到较低的 $\mathbf{t}$ **则**
        $\mathbf{t}^* \leftarrow \mathbf{t}$，$cost^* \leftarrow cost$
      结束如果
    结束循环
  结束循环
结束循环
返回 $cost^*$, $\mathbf{t}^*$

研究了嵌入表分片。开创性工作依赖启发式成本函数和贪心分片策略（Acun 等人，2021；Lui 等人，2021）。Rec- Shard 将分片建模为一个混合整数线性规划优化问题（Sethi 等人，2022年）。然而，它没有考虑表成本的非线性。FlexShard（Sethi 等人，2023年）为序列式深度学习推荐模型提出了一种专门的分片策略。SurCo（Ferber 等人，2022年）通过学习线性代理成本来解决嵌入表分片问题。我们先前的工作使用强化学习（RL）和学习得到的成本模型来优化分片（Zha 等人，2022a；b）。尽管基于强化学习的方法取得了显著改进，但它们无法处理超大型或高成本表，训练成本高昂，并且由于强化学习中的随机策略而不稳定、方差较大。相比之下，NeuroShard 预训练成本模型以实现一次训练、多次使用，并联合搜索按列和按表的分片方案。在基准分片数据集上，NeuroShard 的表现优于基于强化学习的方法，并提升了生产级模型的端到端训练吞吐量。

嵌入表压缩。与此同时，研究人员也在研究如何压缩嵌入表（Zhang 等，2020年；Shi 等，2020年；Zhao 等，2020年；Joglekar 等，2020年；Liu 等，2021；Kang 等，2020年；2021；Pansare 等，2022年；Desai 等，2022年；Lan 等，2019年；Chen 等，2015年）。嵌入表分片是与这些方法正交的方向。这是因为压缩后通常仍需要执行表分片，因为压缩后的表可能仍然过大，无法容纳在单个图形处理器的内存中。因此，分片和压缩可以相互补充，共同提升效率。此外，嵌入压缩可能会导致精度下降，因为嵌入可能会丢失信息。而分片是一种无损优化，用于确定如何划分和放置表，因此可以在不损失任何精度的情况下提高效率。

设备放置。与嵌入表分片相关的另一项任务是神经网络中各项操作的设备放置。这一研究领域的现有工作要么采用强化学习（Mirhoseini 等人，2017年；2018年；Gao 等人，2018年b；Addanki 等人，2019年；Paliwal 等人，2019年；Gao 等人，2018年a；Goldie 和 Mirhoseini，2020年），要么采用成本建模（Lawler 等人，1993年；Jia 等人，2019年；2018年；Narayanan 等人，2019年；Tarnawski 等人，2020年）。我们的工作同样基于成本建模。与常规操作不同，嵌入操作的成本难以估算，因为成本不仅取决于操作本身，还取决于索引分布。我们的工作提出了一种基于学习的解决方案，利用预训练成本模型和在线搜索进行嵌入表分片。

6 结论与未来工作

在本项工作中，我们提出了 NeuroShard，这是一种基于预训练神经成本模型和在线搜索的嵌入表分片框架。我们开发了多种策略，用于训练通用且准确的成本模型，以估算嵌入成本。借助预训练成本模型，在线搜索无需实际执行图形处理器操作，即可将计算资源消耗降至最低。我们证明，NeuroShard 不仅在基准数据集上优于现有分片算法，还显著提升了超大规模生产级深度学习推荐模型的训练吞吐量。未来，我们将把 NeuroShard 扩展到按行分片，以实现大型表的分区。此外，我们还计划研究 CPU 分片或 CPU-GPU 混合分片场景。最后，我们将探索如何将“预训练与搜索”方法应用于其他系统问题。

致谢

莱斯大学的作者部分获得了美国国家科学基金会（# IIS- 2224843）的资助。文中观点仅代表作者本人，不应被视为代表任何资助机构。我们感谢匿名审稿人和 Ankur Mallick 在指导过程中提供的宝贵意见。

参考文献

Acun, B.、Murphy, M.、Wang, X.、Nie, J.、Wu, C.-J. 和 Hazelwood, K. 大规模深度学习推荐模型训练效率研究。发表于*HPCA*，2021。

Addanki, R., Venkatakrishnan, S. B., Gupta, S., Mao, H., 和 Alizadeh, M. Placeto：学习适用于分布式机器学习的通用设备放置算法。发表于*NeurIPS*，2019年。

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A. 等。语言模型是少样本学习者。发表于*NeurIPS*，2020年。

Chang, C.-Y., Chen, N., Chiang, W.-T., Lee, C.-H., Tseng, Y.-H., Wang, C.-J., Chen, H.-H., 和 Tsai, M.-F. 结合基于语义的省略消解进行对话式信息检索查询扩展。发表于*TREC*，2020年。

Chen, W., Wilson, J., Tyree, S., Weinberger, K. 和 Chen, Y. 使用哈希技巧压缩神经网络。发表于*ICML*，2015年。

Cheng, H.-T., Koc, L., Harmsen, J., Shaked, T., Chandra, T., Aradhye, H., Anderson, G., Corrado, G., Chai, W., Ispir, M., 等. 面向推荐系统的宽深度学习. 收录于*DLRS*研讨会，2016年。

Chuang, Y.-N., Chen, C.-M., Wang, C.-J., Tsai, M.-F., Fang, Y., 和 Lim, E.-P. TPR：面向推荐系统的文本感知偏好排序。发表于*CIKM*，2020年。

Chuang, Y.-N.、Tang, R.、Jiang, X. 和 Hu, X. SPEC：一种基于软提示的校准方法，用于缓解临床笔记摘要中的性能波动。*arXiv* 预印本 *arXiv:2303.13035*，2023年。

Covington, P.、Adams, J. 和 Sargin, E. 用于YouTube推荐的深度神经网络。在*RecSys*，2016年。

Desai, A.、Chou, L. 和 Shrivastava, A. 用于深度学习推荐系统中压缩嵌入表的随机偏移块嵌入（RoBE）。*MLSys*，2022年。

Ferber, A., Huang, T., Zha, D., Schubert, M., Steiner, B., Dilkina, B. 和 Tian, Y. SurCo：为组合非线性优化问题学习线性代理模型。*arXiv* 预印本 *arXiv:2210.12547*，2022年。

Gao, Y.、Chen, L. 和 Li, B. POST：通过交叉熵最小化和近端策略优化进行设备放置。发表于*NeurIPS*，2018年a。

Gao, Y.、Chen, L. 和 Li, B. Spotlight：优化深度神经网络训练的设备放置。发表于*ICML*，2018b。

Goldie, A. 和 Mirhoseini, A. 基于深度强化学习的放置优化。发表于*ISPD*，2020年。Gomez-Uribe, C. A. 和 Hunt, N. Netflix 推荐系统：算法、商业价值与创新。*ACM* 管理信息系统汇刊 *(TMIS)*，6(4):1–19，2015年。Gupta, U.、Wu, C.-J.、Wang, X.、Naumov, M.、Reagen, B.、Brooks, D.、Cottel, B.、Hazelwood, K.、Hempstead, M.、Jia, B. 等。Facebook 基于深度神经网络的个性化推荐的架构影响。发表于*HPCA*，2020年。

He, X., Liao, L., Zhang, H., Nie, L., Hu, X., and Chua, T.-S. 神经协同过滤。在*WWW*，2017年。

Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., 和 Meger, D. 有意义的深度强化学习。发表于*AAAI*，2018年。

Jia, Z., Lin, S., Qi, C.R., 和Aiken, A. 探索并行化卷积神经网络中的隐藏维度。发表于*ICML*，2018年。

Jia, Z., Zaharia, M., 和 Aiken, A. 超越深度神经网络的数据并行和模型并行。发表于*MLSys*，2019年。

Joglekar, M. R., Li, C., Chen, M., Xu, T., Wang, X., Adams, J. K., Khaitan, P., Liu, J., 和 Le, Q. V. 面向大规模推荐模型的神经网络输入搜索。发表于*KDD*，2020年。

Kang, W.-C., Cheng, D. Z., Chen, T., Yi, X., Lin, D., Hong, L., 和 Chi, E. H. 为推荐系统中的大词汇量类别特征学习多粒度量化嵌入。发表于*WWW*，2020年。

Kang, W.-C., Cheng, D. Z., Yao, T., Yi, X., Chen, T., Hong, L., 和 Chi, E. H. 学习在不使用嵌入表的情况下嵌入类别特征，以用于推荐。在 *KDD*，2021。

Khudia, D., Huang, J., Basu, P., Deng, S., Liu, H., Park, J., 和 Smelyanskiy, M. Fbgemm：实现高性能低精度深度学习推理。*arXiv* 预印本 *arXiv:2101.05615*，2021。

Kulkarni, T. D., Narasimhan, K., Saeedi, A., 和 Tenenbaum, J. 分层深度强化学习：整合时间抽象与内在动机。*NeurIPS*，2016。

Kumar, A., Zhou, A., Tucker, G., and Levine, S. 用于离线强化学习的保守 Q 学习。*NeurIPS*，2020年。Lai, K.-H., Zha, D., Li, Y., and Hu, X. 双策略蒸馏。载于*IJCAI*，2020a。

Lai, K.-H., Zha, D., Zhou, K., and Hu, X. Policy-gnn：图神经网络的聚合优化。载于*KDD*，2020b。

Lan, Z., Chen, M., Goodman, S., Gimpel, K., Sharma, P., and Soricut, R. ALBERT：一种用于语言表示自监督学习的轻量级 BERT。发表于*ICLR*，2019年。

Lawler, E. L., Lenstra, J. K., Kan, A. H. R., and Shmoys, D. B. 排序与调度：算法与复杂性。运筹学与管理科学手册，4:445–522，1993年。

Li, Y., Chen, Z., Zha, D., Zhou, K., Jin, H., Chen, H., and Hu, X. 通过好奇心引导搜索和自我模仿学习实现自动异常检测。*IEEE* 神经网络与学习系统汇刊，33(6): 2365–2377，2021。Li, Y., Zha, D., Zhang, T., Yan, F. Y., Suh, G. E., and Delimitrou, C. 利用离线强化学习应对分布式系统中的亚稳态故障。发表于*ICLR*，2023年。Lian, X., Yuan, B., Zhu, X., Wang, Y., He, Y., Wu, H., Sun, L., Lyu, H., Liu, C., Dong, X., 等。Persia：一种开放式混合系统，可将深度学习推荐系统扩展至 100 万亿个参数。发表于*KDD*，2022年。Lin, S.-C., Chuang, Y.-N., Yang, S.-F., Tsai, M.-F., and Wang, C.-J. 负样本感知协同过滤。发表于*RecSys*，2019年。Liu, D. C., Rogers, S., Shiau, R., Kislyuk, D., Ma, K. C., Zhong, Z., Liu, J., and Jing, Y. Pinterest 上的相关 Pin：真实世界推荐系统的演进。发表于*WWW*，2017年。Liu, N., Tan, Q., Li, Y., Yang, H., Zhou, J., and Hu, X. 单个向量是否足够？探索网络嵌入中的节点多义性。发表于*KDD*，2019年。Liu, P., Yuan, W., Fu, J., Jiang, Z., Hayashi, H., and Neubig, G. 预训练、提示与预测：自然语言处理提示方法的系统综述。*ACM* 计算调查，55(9):1–35，2023年。Liu, S., Gao, C., Chen, Y., Ji n, D., and Li, Y. 面向推荐系统的可学习嵌入维度。发表于*ICLR*，2021。¨ ¨Lui, M., Yetim, Y., Ozkan, O., Zhao, Z., Tsai, S.-Y., Wu, C.-J., and Hempstead, M. 理解容量驱动的横向扩展神经推荐推理。发表于*2021*年 *IEEE* 系统与软件性能分析国际研讨会（*ISPASS*），第 162–171 页。IEEE，2021年。

Mirhoseini, A., Pham, H., Le, Q. V., Steiner, B., Larsen, R., Zhou, Y., Kumar, N., Norouzi, M., Bengio, S., and Dean, J. 使用强化学习进行设备放置优化。载于*ICML*，2017年。Mirhoseini, A., Goldie, A., Pham, H., Steiner, B., Le, Q. V., and Dean, J. 用于设备放置的层次模型。载于*ICLR*，2018年。Mudigere, D., Hao, Y., Huang, J., Jia, Z., Tulloch, A., Srid- haran, S., Liu, X., Ozdal, M., Nie, J., Park, J., 等。面向深度学习推荐模型快速、可扩展训练的软硬件协同设计。载于*ISCA*，2022年。Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Za- haria, M. Pipedream：用于深度神经网络训练的通用流水线并行。载于*SOSP*，2019年。Naumov, M., Mudigere, D., Shi, H.-J. M., Huang, J., Sun- daraman, N., Park, J., Wang, X., Gupta, U., Wu, C.-J., Azzolini, A. G., 等。面向个性化和推荐系统的深度学习推荐模型。*arXiv*预印本 *arXiv:1906.00091*，2019年。Naumov, M., Kim, J., Mudigere, D., Sridharan, S., Wang, X., Zhao, W., Yilmaz, S., Kim, C., Yuen, H., Ozdal, M., 等。Facebook数据中心的深度学习训练：横向扩展与纵向扩展系统的设计。*arXiv*预印本 *arXiv:2003.09518*，2020年。Oh, J., Guo, Y., Singh, S., and Lee, H. 自我模仿学习。载于*ICML*，2018年。Paliwal, A., Gimeno, F., Nair, V., Li, Y., Lubin, M., Kohli, P., and Vinyals, O. 用强化遗传算法学习优化计算图。载于*ICLR*，2019年。Pansare, N., Katukuri, J., Arora, A., Cipollone, F., Shaik, R., Tokgozoglu, N., and Venkataraman, C. 学习用于设备端推理的压缩嵌入。*MLSys*，2022年。Sethi, G., Acun, B., Agarwal, N., Kozyrakis, C., Trippel, C., and Wu, C.-J. Recshard：面向工业规模神经推荐的基于统计特征的内存优化。载于*ASPLOS*，2022年。Sethi, G., Bhattacharya, P., Choudhary, D., Wu, C.-J., and Kozyrakis, C. Flexshard：面向工业规模序列推荐模型的灵活分片。*arXiv*预印本 *arXiv:2301.02959*，2023年。Shi, H.-J. M., Mudigere, D., Naumov, M., and Yang, J. 使用互补分区的组合式嵌入，实现内存高效推荐系统。载于*KDD*，2020年。

Tan, Q., Liu, N., and Hu, X. 面向社交网络分析的深度表示学习。*Frontiers in bigData*，2:2，2019年。

Tan, Q., Liu, N., Zhao, X., Yang, H., Zhou, J., and Hu, X. 使用图神经网络为推荐系统学习哈希。在*WWW*，2020年。

Tan, Q., Zhang, J., Liu, N., Huang, X., Yang, H., Zhou, J., and Hu, X. 基于动态记忆的注意力网络，用于序列推荐。发表于*AAAI*，2021a。

Tan, Q., Zhang, J., Yao, J., Liu, N., Zhou, J., Yang, H., and Hu, X. 用于序列推荐的稀疏兴趣网络。发表于*WSDM*，2021b。

Tan, Q., Liu, N., Huang, X., Choi, S.-H., Li, L., Chen, R., and Hu, X. S2gae：自监督图自编码器是具有图掩码能力的通用学习器。发表于*WSDM*，2023a。Tan, Q., Zhang, X., Liu, N., Zha, D., Li, L., Chen, R., Choi, S.-H., and Hu, X. 自带视图：通过个性化子图选择实现链接预测的图神经网络。发表于*WSDM*，2023b。Tang, R., Chuang, Y.-N., and Hu, X. 检测大语言模型生成文本的科学方法。*arXiv preprint arXiv:2303.07205*，2023年。Tarnawski, J. M., Phanishayee, A., Devanur, N., Mahajan, D., and Nina Paravecino, F. 深度神经网络图算子设备放置的高效算法。发表于*NeurIPS*，2020年。Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozi`ere, B., Goyal, N., Hambro, E., Azhar, F., et al. Llama：开放且高效的基础语言模型。*arXiv preprint arXiv:2302.13971*，2023年。Wang, C.-J., Chuang, Y.-N., Chen, C.-M., and Tsai, M.-F. 面向个性化推荐的偏度排序优化。发表于*UAI*，2020年。Zha, D., Lai, K.-H., Zhou, K., and Hu, X. 经验回放优化。发表于*IJCAI*，2019年。Zha, D., Lai, K.-H., Wan, M., and Hu, X. Meta-aad：基于深度强化学习的主动异常检测。发表于*ICDM*，2020年。Zha, D., Lai, K.-H., Huang, S., Cao, Y., Reddy, K., Vargas, J., Nguyen, A., Wei, R., Guo, J., and Hu, X. Rlcard：一个用于纸牌游戏强化学习的平台。发表于*IJCAI*，2021a。Zha, D., Lai, K.-H., Zhou, K., and Hu, X. 通过自监督简化深度强化学习。*arXivpreprintarXiv:2106.05526*，2021b。

Zha, D., Ma, W., Yuan, L., Hu, X. 和 Liu, J. 对剧集排序：在程序生成环境中进行探索的一种简单方法。发表于*ICLR*，2021c。

Zha, D.、Xie, J.、Ma, W.、Zhang, S.、Lian, X.、Hu, X. 和 Liu, J. Douzero：通过自博弈深度强化学习掌握斗地主。发表于*ICML*，2021d。

Zha, D., Feng, L., Bhushanam, B., Choudhary, D., Nie, J., Tian, Y., Chae, J., Ma, Y., Kejariwal, A., 和 Hu, X. AutoShard：面向推荐系统的自动化嵌入表分片。载于*KDD*，2022a。

Zha, D., Feng, L., Tan, Q., Liu, Z., Lai, K.-H., Bhargav, B., Tian, Y., Kejariwal, A., 和 Hu, X. DreamShard：面向推荐系统的通用化嵌入表放置。载于*NeurIPS*，2022b。

Zha, D.、Lai, K.-H.、Tan, Q.、Ding, S.、Zou, N. 和 Hu, X. 通过深度分层强化学习实现自动化不平衡学习。载于*CIKM*，2022年。Zha, D.、Bhat, Z. P.、Lai, K.-H.、Yang, F. 和 Hu, X. 以数据为中心的人工智能：观点与挑战。载于*SDM*，2023a。Zha, D.、Bhat, Z. P.、Lai, K.-H.、Yang, F.、Jiang, Z.、Zhong, S. 和 Hu, X. 以数据为中心的人工智能：综述。*arXiv* 预印本 *arXiv:2303.10158*，2023b。Zhang, C.、Liu, Y.、Xie, Y.、Ktena, S. I.、Tejani, A.、Gupta, A.、Myana, P. K.、Dilipkumar, D.、Paul, S.、Ihara, I. 等。利用基于频率的双重哈希缩减推荐系统的模型规模。载于*RecSys*，2020年。Zhang, S.、Yao, L.、Sun, A. 和 Tay, Y. 基于深度学习的推荐系统：综述与新视角。*ACM* 计算调查 *(CSUR)*，52(1):1–38，2019年。Zhao, X.、Wang, C.、Chen, M.、Zheng, X.、Liu, X. 和 Tang, J. Autoemb：流式推荐中的自动化嵌入维度搜索。载于*SIGIR*，2020年。Zhou, G.、Mou, N.、Fan, Y.、Pi, Q.、Bian, W.、Zhou, C.、Zhu, X. 和 Gai, K. 用于点击率预测的深度兴趣演化网络。载于*AAAI*，2019年。Zhou, H.、Tan, Q.、Huang, X.、Zhou, K. 和 Wang, X. 用于基于会话推荐的时序增强图神经网络。载于*SIGIR*，2021年。Zhou, H.、Fan, J.、Huang, X.、Li, K. H.、Tang, Z. 和 Yu, D. 通过协同属性建模实现点击率预测的多兴趣精炼。载于*CIKM*，2022年。Zhou, H.、Zhou, S.、Duan, K.、Huang, X.、Tan, Q. 和 Yu, Z. 兴趣驱动的图结构学习用于基于会话的推荐。载于*PAKDD*，2023年。

A 分析性实验的详细信息

本节将详细介绍2节中进行的三项分析性实验。

A.1 维度的影响

在本实验中，我们旨在研究维度如何影响表的计算成本。在 DLRM 数据集（Naumov 等人，2019年）中，我们拥有表的表索引和哈希大小，但未指定维度。因此，我们从集合$\{128, 64, 32, 8, 4\}$中改变维度，以研究维度的影响。请注意，对于上述五种情况，我们使用相同的哈希大小和相同的表索引，唯一的区别是维度。我们尝试了多个表，并观察到类似的模式。一些其他随机选定的表的结果如图10所示。

A.2 多表成本与单表成本之和

本分析实验旨在复现（Zha 等人，2022年）中的结果，以了解多表成本与单表成本总和之间的关系。我们随机抽取 50 个数据点来绘制散点图，其中每个数据点由 DLRM 数据集（Naumov 等人，2019年）中 856 张表里的 10 张表组成。对于每个数据点，我们在单个图形处理器上测量两种成本：

- **多表成本：** 我们直接在图形处理器上运行融合操作（前向和反向传播）。首先运行该操作 10 次，以预热硬件。然后再运行 100 次，并将中位数成本作为多表成本。
- **单表成本之和：** 我们按照相同的流程收集每个单表的成本，然后将单表成本相加。

我们发现，单表成本之和通常大于多表成本，因为融合操作速度更快。

A.3 通信成本分析

这项分析实验旨在帮助理解最大设备维度与最大前向/反向通信成本之间的关系。为模拟最大设备维度不同的场景，我们采用第 3.1 节所述的相同随机表放置生成策略，并通过贪心策略平衡各维度，同时加入随机性以覆盖不同场景。完整的生成流程总结在算法 5 中。对于每个数据点（即一个放置方案），我们在单台服务器上运行前向全互联通信和反向通信，以收集各 GPU 的成本。与计算成本类似，我们首先运行通信 10 次，以预热硬件。随后再运行通信 100 次，并将中位数成本作为通信成本。请注意，每个 GPU 测得的本地成本各不相同。我们只关注最大成本，因为它是瓶颈。

B 合成输入生成的详细信息

本节将进一步介绍我们如何生成合成数据，以便在基准测试中实现良好的覆盖范围。

B.1 表增强

关键思想是为单个表生成不同的维度。给定一个维度列表，对于表池中的每个表，我们将该表与每个维度关联，作为增强表。算法 3 概述了增强过程。

算法 3表增强

输入： T嵌入表，维度集合$\mathcal{D}$
增强表池$\mathcal{P} \leftarrow \{\}$
对于 每个T表**执行**
  对于$\mathcal{D}$**中的每个维度，执行**
    增强后的表格$aug\ table \leftarrow$ 所选的具有所选维度的表格
    将 $aug\ table$ 添加到 $\mathcal{P}$
  结束循环
结束循环
返回$\mathcal{P}$

B.2 随机表组合生成

我们在算法中总结了随机数据表组合的生成过程4。

算法4随机表组合生成

输入： 增强表池 $\mathcal{P}$，最小表格数量为 table T_{min}，表格 T 的最大数量 $_{\text{max}}$，表格数量我们旨在生成的组合 N_{com}
表组合 $\mathcal{T}_{\text{com}} \leftarrow \{\}$
对于 $i = 0,\ 1, ..., N_{\text{com}}$ **执行**
  均匀采样表格数量 T in $[T_{\text{min}}, T_{\text{max}}]$
  从 $\mathcal{P}$ 中随机抽取 T 个表格并追加此项。组合到 $\mathcal{T}_{\text{com}}$
结束循环
返回 $\mathcal{T}_{\text{com}}$

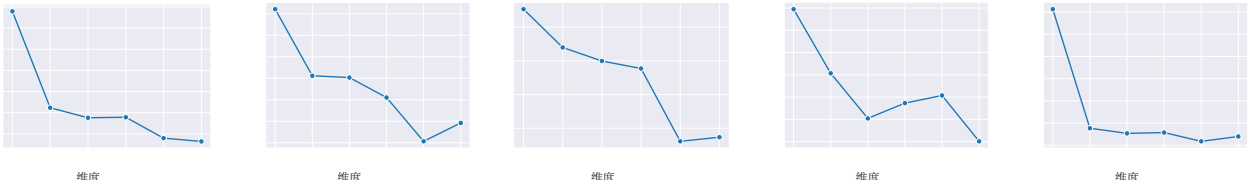


图 *10.* 其他一些随机选取的表格相对于维度的计算成本。

B.3 随机表放置生成

我们在算法中总结了表放置生成过程5。

算法 5随机表放置生成 1: **输入：** 增强表池 $\mathcal{P}$，表的最小数量 T_{min}，表的最大数量 T_{max}，目标生成的表放置数量 N_{place}，GPU 设备数量 D。 2: 表放置 $\mathcal{T}_{\text{place}} \leftarrow \{\}$ 3: **对于** $i = 0,\ 1,\ \ldots N_{\text{place}}$ **执行**4: 均匀采样表的数量 T ，范围为$[T_{\text{min}}, T_{\text{max}}]$5: 从 $\mathcal{P}$中随机采样 T 张表 6: 根据表维度按降序对这 T 张表进行排序 7: 均匀采样应用贪心策略的概率 $p \in [0, 1]$8: **对于** 这 T 张表中的每一张**执行**9: 随机采样 $p' \in [0, 1]$10: 获取不会导致内存错误的候选 GPU 设备。 11: **如果** $p' \leq p$**则**12: 将当前表分配给设备维度最低的候选 GPU 13: **否则**14: 将当前表随机分配给候选 GPU 中的一个 15: **结束如果**16: **结束循环**17: 将该放置追加到 $\mathcal{T}_{\text{place}}$18: **结束循环**19: 返回 $\mathcal{T}_{\text{place}}$

C 成本模型训练详情

本节将进一步介绍用于训练神经成本模型的神经网络架构和损失函数。

神经网络架构。 对于计算成本模型，我们使用一个大小为 128-32 的多层感知机来处理表特征，并使用另一个大小为 32-64 的多层感知机。对于通信成本模型，我们使用一个大小为 128-64-32-16。

损失函数。 我们使用均方误差（MSE）来更新计算成本模型和通信成本模型。具体而言，设$\mathbf{x}$ 为特征（可以是表特征或通信特征），y 为真实值，g 为成本模型（可以是计算成本模型或通信模型）。设 N_{sample} 为样本数量。那么损失为

$$\mathcal{L} = \sum_{i=1}^{N_{\text{sample}}} \text{MSE}(\mathbf{x}_i, y_i), \tag{2}$$

其中，MSE$(\cdot, \cdot)$ 表示均方误差损失。实际中，我们可以使用小批量训练。

D 数据集

我们使用用于评估嵌入表分片的基准数据集（Nau mov 等人，2019年）。该数据集可公开获取，网址为 https://github.com/facebookresearch/dlrm_ 数据集。它包含 856 个合成表，其索引分布与 Meta 生产环境中的工作负载相似。先前的研究已对该数据集的统计信息进行了全面总结。详情请参阅（Zha 等人，2022a；b）中的附录。

我们在表格 5中总结了实验中使用的 12 个分片任务。所有分片任务都要求每个图形处理器的内存为 4 GB。

公开数据集的讨论。表格6比较了深度学习推荐模型数据集与若干大型公开数据集的规模。与这些公开数据集相比，深度学习推荐模型具有显著更多的表、更大的平均哈希大小以及更大的平均池化因子。无论我们如何进行分片，这些数据集上的嵌入成本始终都非常小。例如，深度学习推荐模型的表数量至少是 Criteo 的 30 倍，平均哈希大小至少是其 200 倍，平均池化因子则是其 15 倍。因此，鉴于深度学习推荐模型数据集的嵌入成本处于 17 ms 至 40 ms 的范围内（本文中的表格 1），Criteo 的嵌入成本可能仅为 1 ms 左右，甚至更低。因此，没有必要对 Criteo 数据集进行分片。这些数据集并不适合用于评估嵌入表分片算法。

表5。 实验中生成的分片任务。

GPU 数量	表数量范围	表维度范围
4	10-60	4
4	10-60	4, 8
4	10-60	4, 8, 16
4	10-60	4, 8, 16, 32
4	10-60	4, 8, 16, 64
4	10-60	4, 8, 16, 64, 128
8	20-120	4
8	20-120	4, 8
8	20-120	4, 8, 16
8	20-120	4, 8, 16, 32
8	20-120	4, 8, 16, 64
8	20-120	4, 8, 16, 64, 128

表6. 部分热门公开推荐数据集与工业级DLRM数据集的嵌入表特征统计比较。

数据集		表数量	平均哈希大小	平均池化因子	Link
公开	Criteo	26	17,839	1	https://www.kaggle.com/c/criteo-display-ad-challenge
	Avazu	23	67,152	1	https://www.kaggle.com/c/avazu-ctr-prediction/data
	KDD	10	601,908	1	https://www.kaggle.com/c/kddcup2012-track2/data
工业规模	DLRM	856	4,107,458	15	https://github.com/facebookresearch/dlrm_数据集

E 基线方法

在本节中，我们将介绍本次实验中使用的所有基线方法的具体细节。

E.1 贪心算法

贪心分片算法已被用于以往的分布式推荐系统研究中（Acun 等人，2021；Lui 等人，2021）。其核心思想是在每一步中，使用贪心算法将表格分配给当前成本最低的设备，以实现成本均衡，其中成本通过不同方式进行估算。具体而言，贪心算法由以下两个步骤组成。

- **设计成本函数：**我们为每个表格分配一个成本，用于量化其在设备上的预期运行时间。成本是我们希望实现均衡的目标。
- **贪心分配：** 目标是平衡每个设备上的成本总和。为此，我们首先根据成本函数定义的成本，按降序对嵌入表进行排序。如果采用贪心方式分配表格，排序可以更容易地实现平衡。然后，我们从成本最高的表格开始分配。在每一步中，我们都会做出贪心决策，将当前表格分配给目前成本总和最低的设备。这种经过排序优化的贪心策略可以使每个设备上的成本总和大致相近。

四种基线方法在成本函数的设计方式上有所不同。这会显著影响负载均衡，因为成本函数决定了我们的优化目标。我们将使用的成本函数总结如下：

- **基于大小：** 我们使用表格大小作为成本函数。其理念是，平衡表格大小可以降低发生内存溢出错误的风险。此外，表格大小也与维度呈正相关，因此也能反映工作负载。
- **基于维度：** 我们使用表维度作为成本函数。表维度是表示成本的重要特征，因为它可以决定计算和通信工作负载。因此，平衡维度总和（即设备维度）是很自然的做法。
- **基于查找：** 我们使用表维度与平均池化因子的乘积作为成本函数。其直觉是，表维度和池化因子可以决定嵌入查找中的计算工作负载。
- **基于大小查找：** 我们使用表维度、平均池化因子和表格大小的乘积作为成本函数。这是一种更全面的成本函数，同时考虑了查找成本和表格大小。

E.2 强化学习算法

我们纳入了两种用于嵌入表分片的最先进强化学习算法（Zha 等人，2022年a；b）。它们的思路相似，但在优化目标和训练方法上有所不同。我们总结如下。

- **AutoShard** (Zha 等人，2022a)：它训练一个 LSTM 控制器来执行分片，以均衡计算成本。其目标是均衡度，定义为最低成本除以最高成本。代码可在以下地址获取：https://github.com/daochenzha/autoshard

- **DreamShard** (Zha 等人，2022b)：它在 AutoShard 的基础上进一步均衡通信。它还将成本模型扩展到通信成本。此外，它引入了一个估计的马尔可夫决策过程，使训练和推理速度大幅提升。代码可在以下地址获取：https://github.com/daochenzha/dreamshard

E.3 规划算法

与强化学习并行，规划算法通过搜索确定分片方案。TorchRec 提供了一种基于规划的分片策略。为进行公平比较，我们允许 TorchRec 搜索按列分片和按表分片方案。然而，TorchRec 仍依赖启发式成本，因此我们没有看到 TorchRec 相较于贪心算法有明显改进。代码可在以下地址公开获取：https://github.com/pytorch/torchrec。

F 超参数与配置

本节列出 NeuroShard 的所有超参数。我们还列出了硬件/软件配置

- **生成合成输入：** 我们通过维度 $\{4, 8, 16, 32, 64, 128\}$增强表池。我们随机选择 1 到 15 个表进行表组合生成，并选择 $N_{\text{放置}}$ 表进行表放置生成，其中 $10 \leq N_{\text{放置}} \leq 60$ 适用于 4 个 GPU，$20 \leq N_{\text{放置}} \leq 120$适用于 8 个 GPU。通信数据的起始时间戳从 0 到 20 毫秒中抽样。我们为每个成本模型生成 100K 个样本。

- **训练神经成本模型：**我们使用 80% 的数据进行训练，10% 的数据进行验证，10% 的数据进行测试。批大小设为 512。我们使用 Adam 优化器，学习率设为 0.001，其他配置采用默认值。我们训练 1000 个训练轮次，并保存在验证数据上表现最佳的模型。

- **在线搜索：** 我们设置 $N = 10$、$K = 3$、$L = 10$ 和 $M = 11$。

- **软件：** 我们使用 PyTorch 1.9.1 和 FBGEMM 0.0.1

- **硬件：** 我们在一台配备 48 个 Intel(R) Xeon(R) Silver 4116 CPU @2.10GHz 处理器、188 GB 内存和八个 NVIDIA GeForce RTX 2080 Ti GPU 的服务器上进行实验。

G 额外消融实验结果

结果如表格 7所示。

H 使用强化学习的搜索补充讨论

在这项工作中，我们使用束搜索和贪心网格搜索来进行搜索。然而，通过使用强化学习（Zha 等人，2020年；Lai 等人，2020年b）训练元策略（Zha 等人，2021a），并将其迁移到不同任务中，这一搜索过程还可以得到进一步加速。下面，我们重点介绍几种正在尝试的潜在策略。1）分层强化学习：其思想是将分片任务分解为若干子任务（Kulkarni 等人，2016；Zha 等人，2022年c）。这可以自然地应用于嵌入表分片问题，因为其中存在两个层级的搜索：按列分片和按表分片。2）自我模仿学习：其思想是选择高奖励样本，并使用监督损失来鼓励策略复现良好行为（Oh 等人，2018；Zha 等人，2021年c；b；Li 等人，2021）。由于我们通常拥有大量分片系统日志，这可能会有所帮助。具体而言，可以从系统日志中选择良好的分片方案，并使用监督损失训练策略。2）离线强化学习：其思想是基于离线数据学习最优策略（Kumar 等人，2020年；Li 等人，2023年）。这同样可以应用于离线分片日志。

请注意，强化学习元策略也可以与搜索结合，以引导搜索过程。

I 研究材料

本节面向有兴趣复现我们结果的读者。接下来，我们将介绍如何下载数据集、如何安装 NeuroShard 及其依赖项、如何运行 NeuroShard，以及如何从硬件收集延迟数据。

I.1 研究材料检查清单（元信息）

- **算法：**预训练模型、束搜索、网格搜索。

表 7. 最大维度为 128、使用 8 个 GPU 时的消融研究。无束搜索表示在按列分片中移除束搜索。无贪心网格搜索表示不对表格维度阈值进行网格搜索。无缓存表示停用计算成本的缓存机制。

	Cost（毫秒）	成功率	分片时间（秒）	缓存命中率
不使用束搜索	-	63.0%	0.09	75.4%
不使用贪心网格搜索	55.97	100.0%	4.09	79.8%
无缓存	49.10	100.0%	164.21	0.0%
完整 NeuroShard	49.10	100.0%	20.79	93.0%

- **程序：** 使用 PyTorch 和 Python 实现。
- **编译：** PyTorch 1.8.0 和 Python 3.8.0。
- **数据集：** Meta 开源的合成嵌入数据。
- **运行环境：** Ubuntu 18.04.6 LTS。
- **硬件：** 八块 NVIDIA GeForce RTX 2080 Ti GPU .
- **指标：** 延迟（毫秒） .
- **输出：** 命令行输出 .
- **实验：** 生成分片方案并收集延迟 .
- **所需磁盘空间（大约）？：** 20 GB。
- **准备工作流程所需时间（大约）？：** 2 小时。
- **完成实验所需时间（大约）？：** 20+小时。
- **是否公开提供?：** 是。
- **代码许可证（若公开提供）：** MIT。
- **已归档（提供 DOI）？：** https://zenodo.org/badge/latestdoi/556106683

I.2 描述

I.2.1 交付方式

该工件以 ZIP 文件形式提供，下载地址为 https://zenodo.org/badge/latestdoi/556106683。源代码也已在 GitHub 上公开发布，地址为 https://github.com/daochenzha/neuroshard，并附有 README 文件，其中提供了逐步运行代码的说明。

I.2.2 硬件依赖项

我们的结果收集自一台配有八块 NVIDIA GeForce RTX 2080 Ti GPU 的服务器。其他类型的 GPU 也可以使用，但由于收集到的延迟取决于 GPU，结果可能会有所不同。

I.2.3 软件依赖项

运行代码需要 Python 3.8.0 或更高版本（我们使用的是 3.8.0）。此外，要使用 GPU，必须安装适当版本的 CUDA。最后，还必须安装 FBGEMM，这是一款开源的嵌入表操作工具。

I.2.4 数据集

合成数据集可从以下地址公开获取 https://github.com/facebookresearch/dlrm_datasets.git。运行以下命令即可下载该数据集：Git LFS：

git lfs install --skip-smudge git clone \ https://github.com/facebookresearch/dlrm_datasets.git cd dlrm datasets_ git lfs pull \ --include=embedding_bag/2021/fbgemm_t856_bs65536.pt.gz gzip -d embedding_bag/2021/fbgemm_t856_bs65536.pt.gz

运行上述命令后，将在 embedding bag/2021/ 中生成一个名为 fbgemm t856 bs65536.pt 的文件，其大小为 4.0 GB。该文件包含用于嵌入查找的合成索引，其索引分布与 Meta 生产环境中的分布相似。

I.3 安装

首先，我们需要安装 Python 3.8。0+ 然后安装 CUDA。其次，按照以下步骤安装 PyTorch 1.8.0：

pip3 install PyTorch==1.8.0

第三，我们需要安装开源的 FBGEMM，可从以下地址获取：https://github.com/pytorch/FBGEMM/tree/main/fbgemm_图形处理器。

请注意，除 A100 或 V100 GPU 外，FBGEMM 需要按照说明手动构建，预计耗时 0.5 到 1 小时。最后，克隆 NeuroShard 代码并安装：

git clone \ https://github.com/anonymoussubmition/neuroshard.git c d neuroshard pip3 install -e .

I.4 实验工作流程

实验分为四个主要步骤：

1. 我们使用脚本处理原始合成数据，并随机构建一些分片任务以进行评估。

2. 我们在硬件上进行微基准测试，以收集计算和通信成本。这些成本数据将保存在文件中。

3. 基于收集到的成本数据，我们预训练成本模型。具体而言，我们训练了三个模型，包括计算成本模型、前向通信成本模型和反向通信成本模型。

4. 使用预训练成本模型，我们可以在 NeuroShard 中运行嵌入表分片的在线搜索。我们也可以运行基线启发式分片算法。性能可以通过使用成本模型（模拟）或从硬件收集成本来评估。

现在我们逐一介绍上述步骤。首先，我们需要确保当前目录位于 neuroshard/，并将 fbgemm t856 bs65536.pt 文件放在当前目录中。**在步骤 1 中**，运行以下命令：

```
python3 tools/gen_dlrm_data.py \ --data
fbgemm_t856_bs65536.pt
```

此处，--data 指定原始数据的路径。预期输出为：

```
正在处理 DLRM 数据.....正在生成表格
配置.....
```

处理后的数据默认保存在 data/dlrm 数据集目录中。之后，使用以下命令生成分片任务：

```
python3 tools/gen_tasks.py --max_dim 128
```

这里，--max dim 指定最大表维度（表 1 的第二行）。我们使用 128 作为示例来展示如何运行代码。预期输出为：

```
已生成 100 个分片任务!
```

分片任务默认保存在 data/tasks/4 gpus 中。**在步骤 2 中**，我们通过微基准测试收集成本数据。首先收集计算成本数据：

```
python3 收集_计算_成本_数据.py --数据_大小 10
```

--data size 用于指定要生成的数据样本数量。这里我们使用较小的数量，以便程序运行得更快。若要复现结果，应将 --data size 设置为 100K。最后两行的预期结果应与以下内容相似（第一行中的具体数值可能有所不同）：

```
2997,436,2332,3728,535 3.4415176584173777 设备 0 已完成!
```

同样地，我们可以通过运行以下脚本来收集通信成本数据：

```
python3 收集_comm_成本_data.py --data_size 10
```

此处我们还将 --data 的大小设得较小，以便更快运行。要复现结果，应将其设为 100K。预期输出的最后一行应为：

```
评估器子进程已终止!
```

运行上述两个脚本后，预计成本数据将存储在 data/cost data/ 中。**在步骤 3 中**，我们根据收集到的成本数据预训练神经成本模型。运行以下命令来训练计算成本模型：

```
python train_compute_成本_model.py --epochs 3
```

--epochs 表示训练轮数。此处我们将其设为 3，以加快训练速度。要复现结果，需将 --epochs 设为 1000。预期输出的最后一行是（具体数值可能有所不同）：

```
最终结果，训练集均方误差：8.861547689180117，验证集均方误差：
8.398209571838379，测试集均方误差：8.432512283325195
```

然后，我们对训练通信成本模型也进行类似的操作。我们可以运行以下命令：

```
python3 train_comm_cost_model.py --epochs 3
```

同样，我们将 --epochs 设置为 3，以加快训练速度；若要复现结果，则需要将其设置为 1000。预期输出的最后一行如下（具体数值可能有所不同）：

```
最终结果，训练均方误差：31.95864486694336，验证均方误差
42.04985046386719，测试均方误差：23.29010772705078
```

训练后，models/ 中应存储三个模型。**在步骤 4 中**，我们通过模拟或真实硬件评估不同的分片算法。要获得模拟结果，我们可以使用以下命令运行 NeuroShard 和启发式基线方法：

```
python3 eval_模拟器.py --alg neuroshard python3 eval_模拟器.
py --alg random python3 eval_模拟器.py --alg dim_greedy
python3 eval_模拟器.py --alg lookup_greedy python3 eval_模拟器.
py --alg size_greedy python3 eval_模拟器.py --alg size_lookup_
greedy
```

同样，我们在真实硬件上评估成本 with

python3 eval.py --alg neuroshard python3 eval.py --alg 随机 python3 eval.py --alg dim_贪心 python3 eval.py --alg lookup_贪心 python3 eval.py --alg size_贪心 python3 eval.py --alg size_lookup_贪心

I.5 评估与预期结果

对于步骤 4 中的每个脚本，最终结果都会打印在终端中。成本模型需要使用大小为 100K 的 --data 和 3 个 --epochs 进行训练，才能得到与下方类似的结果。NeuroShard 的模拟结果预期输出为：

平均值：39.0441405081749 有效 100 / 100

第一行是平均延迟。第二行表示 NeuroShard 在所有任务上均成功，即没有发生内存错误。如果运行大小查找贪心算法，结果将是：

平均值：47.95177095494372 有效 94 / 100

它仅在 100 个任务中的 94 个任务上成功。对于真实硬件上的结果，NeuroShard 的预期输出为：

平均值：39.98647058823529 有效 100 / 100

大小查找贪心算法的预期结果为：

平均值：48.010588235294115 有效 94 / 100

模拟结果与实际成本一致。上述过程仅提供了表 1 中的一列结果。要获得完整结果，我们需要自定义 gen tasks.py 中的 --max dim、--T 范围和 --max mem，以及 collect comm cost data.py、eval.py 和 eval simulator.py 中对应的参数。

I.6 注释

成本高度取决于所使用的 GPU，不同版本的 PyTorch 可能会产生不同的结果。此外，结果本身可能存在方差，因此在另一台机器上获得的结果可能与上面显示的结果不同。