Adaembed：大规模推荐模型的自适应嵌入

杨洋洋[1)]

[1)](华中科技大学计算机学院 武汉市 中国 430074)

摘 要 目前，深度学习推荐模型广泛采用庞大的嵌入表以表示分类稀疏特征。尽管通过增加嵌入行的数量以涵盖更多特征实例可以提升模型的准确率，但这会导致部署成本的增加和模型执行效率的下降。现有研究主要专注于优化模型执行速度，例如平衡嵌入片段、加速嵌入检索、嵌入压缩和灵活的资源分配等方面。在这一背景下，本文提出了一种自适应修剪嵌入的补充系统——Adaembed。该系统能够识别重要的嵌入，并修剪不重要的嵌入，从而缩减嵌入表的规模。在工业环境中的评估结果表明，AdaEmbed 能够节省部署所需的嵌入大小达 35-60%，并将模型执行速度提升了 11-34%，同时显著提高了准确性。

关键词 深度学习推荐模型；自适应修剪；内存管理；虚拟物理散列索引

AdaEmbed: Adaptive Embedding for Large-Scale Recommendation Models

YangYang Yang[1)]

[1)](Huazhong University of Science and Technology, Wuhan, China,430074)

Abstract Deep learning recommendation models widely utilize extensive embedding tables to represent sparse categorical features currently. While increasing the number of embedding rows to encompass more feature instances can enhance the model's accuracy, it escalates deployment costs and reduces the efficiency of model execution. Existing research primarily focuses on optimizing model execution speed, such as balancing embedding partitions, accelerating embedding retrieval, embedding compression, and flexible resource scaling. In this context, this paper introduces a supplementary system for adaptively pruning embeddings—Adaembed. This system can identify crucial embeddings and trim insignificant ones, thereby reducing the size of the embedding table. Evaluation in industrial environments demonstrates that AdaEmbed achieves savings of 35-60% in required embedding size for deployment and improves model execution speed by 11-34%, concurrently achieving substantial accuracy enhancements.

Key words DLRM；Adaptive Pruning；Memory Management；Virtual Physical Hash Index

1 引言

深度学习推荐模型（DLRM）在许多在线服务中扮演着重要角色，例如视频推荐、广告展示以及电商服务。相较于传统的机器学习方法，DLRM 的输入数据包含了连续且密集的特征如时间戳，以及分类稀疏的特征，比如视频类型。对于每个稀疏特征，通常会关联一个嵌入表，其中该特征的每个实例都由一个可训练的嵌入行所表示。在模型的前向传播和反向传播过程中，模型会读取并更新所访问到的嵌入权重行。

由于 DLRM 的准确性通常随着嵌入规模的增加而提高，因此现代 DLRM 模型的嵌入大小持续增长，规模高达 TB 级别和数十亿的嵌入。然而，DLRM 对于在线训练和推理具有严格的吞吐量和延迟要求，而庞大的嵌入规模给计算、通信和内存优化带来了挑战。为了满足所需的模型吞吐量需求，在模型的实际部署中通常需要使用数百个 GPU 进行嵌入的训练。

在本文中，引入了一个自适应修剪系统 AdaEmbed 来在训练期间自动优化每个特征的嵌入，以提高模型的准确性。对于给定的嵌入规模，AdaEmbed 能够灵活地识别和保留在训练过程中对模型准确性影响更大的嵌入。AdaEmbed 是对现有 DLRM 工作的补充和支持，只需进行少量的代码更改。

2 背景和动机

2.1 DLRM

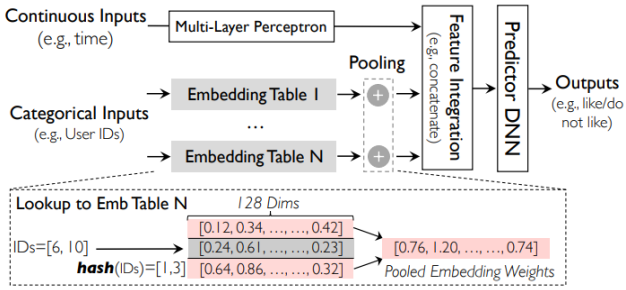

图 1 DLRM 的组成结构

如图 1 所示，DLRM 由多层感知器 MLP（Multi-Layer Perceptron）的集合组成。MLP 被完全连接在一起，用于捕获连续的密集特征，如时间戳，以及一系列嵌入表，用以将各种分类稀疏特征映射到密集表示，例如用户和视频 ID。DLRM 可能包含多达数千个稀疏特征：每个特征通常都与一个嵌入表相关联，而每个表可能含有数百万行。每个嵌入行都是一个多维权重向量，对应于特定的特征实例，例如“用户 ID”的特定用户 ID。

DLRM 与传统的计算机视觉（CV）和自然语言处理（NLP）模型有所不同，因为它们需要在按时间顺序组织的大量数据上进行训练，以跟上最新的推荐趋势。因此，训练数据的分布在训练过程中会发生变化。在模型计算的前向传播过程中，每个输入样本都包含一组嵌入 ID，用于提取相应的嵌入权重向量。为了降低计算复杂度，样本的嵌入权重将使用逐元素池化操作按表进行汇总，通常沿着每个向量维度进行求和或取最大值。汇总后的小批量样本的嵌入权重将与它们的稠密特征的中间输出一起打包，形成用于更深层次的批量输入。在反向传播过程中，访问的嵌入权重将根据梯度进行更新。

由于大量的稀疏特征实例，嵌入权重可以占据高达几 TB 空间。因此 DLRM 表现出比传统 ML 模型更大的内存强度。因此，实际的 DLRM 部署使用稀疏特征层的模型并行性和 MLP 的数据并行性的组合。前者在工作者之间分配不同的嵌入分区以避免复制它们，而后者支持密集特征输入的并发处理。即便如此，模型部署通常需要数百个 GPU 来实现所需的模型吞吐量。

2.2 DLRM部署挑战

DLRM 的部署遵循着“在保证更好准确性的同时尽可能追求更快速度”的原则。DLRM 模型的执行速度和准确性分别通过每秒查询（QPS）吞吐量和归一化熵（NE）损失来衡量。更高的 QPS 和更小的 NE 代表着更优秀的性能表现，任何相对>0.02%的 NE 增益都被视为显著的提升。然而，针对这两个方面的优化可能会引发新的问题。

大规模的嵌入尺寸改善 NE 指标

为适应稀疏特征及其实例的更多嵌入行，DLRM 的嵌入大小不断增长。由于 DLRM 通常在数月的数据上进行训练，并随着时间的推移进行重新训练，因此实例集的大小最终将远远超过嵌入大小。为了限制嵌入大小，现有设计通常对原始实例 ID 执行散列，然后使用散列 ID 访问其嵌入行。直观地说，使用更多的嵌入行意味着考虑更多的实例，从而为更好的 NE 提供更好的数据覆盖。

图 2 嵌入尺寸对 NE 的影响

图 2 显示了在训练的不同时间点上嵌入大小对归一化熵（NE）回归的影响。NE 回归表示使用较小嵌入大小相对于完整尺寸模型时准确性的下降。注意到：（1）使用较小的嵌入大小会严重损害 NE。例如，将嵌入行数减少 75%（即 0.25×模型）导致第 2 天 NE 回归约为 0.02%；（2）随着训练的进行，随着产生更多实例，这种 NE 回归会逐渐扩大。

大规模的嵌入尺寸降低 QPS

然而，增加嵌入量可能会导致模型执行速度变

慢，并在执行阶段消耗更多的机器资源：如果无法在高带宽 GPU 中存储所有嵌入，那么嵌入的访问速度就会变慢。此外，由于可能需要通过网络传输更多嵌入，通信时间也会增加。根据图 3 的显示，在相同的资源设置下，0.5×模型相比完整模型实现了 1.4×的 QPS 加速。即使使用最先进的 DLRM 优化方法，即缓存和预取将来批次中将要访问的嵌入，也无法消除每秒查询（QPS）的下降。更重要的是，它们可能不足以进行在线训练和模型服务，因为无法提前知道输入的数据情况。

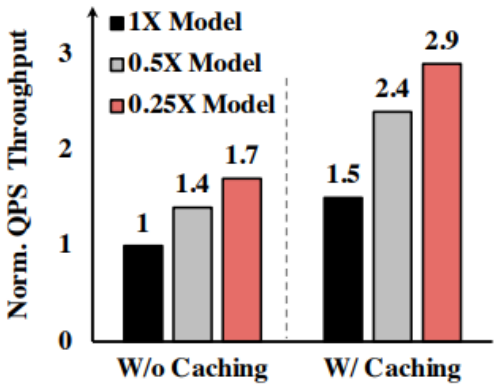

图 3 嵌入大小对 QPS 影响

2.3 研究现状

针对 DLRM 部署问题有着许多方面的研究与改进。

[1]提出了 Neo：一个软硬件协同设计的系统，用于大规模 DLRM 的高性能分布式训练。在软件方面，Neo 配备了许多新颖的软件技术，包括 4D 并行，高性能嵌入内核，混合内核融合和分层内存管理。在硬件方面，可扩展的 ZionEX 平台允许扩展到具有数千个节点的完整数据中心，从而使数据中心规模的 AI 训练集群能够继续满足深度学习模型不断增长的需求。使用 16 个 ZionEX 节点对 128 个 GPU 进行的评估表明，Neo 在训练生产中部署的 12 万亿参数 DLRM 模型方面的性能比现有系统高出 40 倍。

针对在低延迟的情况下更新模型的问题，[2]设计了 Ekko：一种能够实现低延迟模型更新的新型动态链接库。它的设计思想是允许模型更新立即传播到所有推理簇，从而绕过长延迟的模型检查点、验证和广播。Ekko 基于 P2P 模型更新算法，可以协调数十亿的模型更新，以有效地传播到地理分布式数据中心的副本。此外它还具有 SLO 保护机制，保护模型状态不受网络拥塞和有害模型在线更新的影响。

针对深度学习推荐模型推理中 CPU 端 DRAM 访问与 GPU 计算之间的速度鸿沟，以及现有的 GPU 端 cache 策略的低效性，[3]提出了 Fleche：一种整体的高速缓存方案，详细设计了高效的 GPU 驻留嵌入式缓存。与现有技术相比，Fleche 算法显著提高了嵌入层的吞吐量，并获得了端到端推理吞吐量的加速。

[4]提出了一种嵌入压缩方法 TT-Rec：通过将嵌入表分解为多个小矩阵的乘积的方式进行压缩，以计算换取空间。TT-Rec 的核心是使用基本的参数化方法帮助控制对计算设施过度的需求上，将巨大的嵌入表替换为一系列的矩阵乘积操作。在内存占用，模型精度和耗时三个维度上对 TT-Rec 方法进行了评估，证明 TT-Rec 方法能够在 Kaggle 数据上压缩模型 4 倍和 221 倍，而对应的 loss 精度仅损失 0.03%和 0.3%。而在 Terabtye 数据集上，能够实现 112 倍的模型压缩，并且和非压缩方法的 baseline 相比，没有 loss 精度损失和训练时间的明显增加。

[5]引入了一个名为 Kraken 的持续学习系统。Kraken 的关键在于一个稀疏感知的训练系统，通过一个特殊的参数服务实现，能够同时结合数据并行和模型并行来训练推荐模型。这个参数服务支持特征自动准入和过期机制来更有效的管理在在线学习期间稀疏嵌入的生命周期，能够在有限的内存占用下提升模型性能。此外，Kraken 的在线学习系统能够将稀疏嵌入的存储与模型计算解耦，进一步极大提升网络和磁盘利用率。

[6]设计了一种用于大规模深度学习广告系统的分布式 GPU 分层参数服务器。提出了一个分层的工作流程，利用 GPU 高带宽内存，CPU 主内存和 SSD 作为 3 层分层存储。所有的神经网络训练计算都包含在 GPU 中。在真实数据上的大量实验证实了该系统的有效性和可扩展性。4 节点层次化 GPU 参数服务器训练模型的速度比 MPI 集群中 150 节点内存分布式参数服务器快 2 倍以上。此外，该系统的性价比是 4-9 倍优于 MPI 集群解决方案。

主要的工作是致力于给定的嵌入优化深度学习推荐模型的执行速度，如平衡嵌入分片，加速嵌入检索，嵌入压缩，弹性资源扩容等。

2.4 Motivate & challenges

与现有的研究不同，提出一种动态优化方案：通过在模型训练期间优化每个特征的嵌入，从根本上减少相同精度所需要的嵌入大小。改进的策略主

要源于以下的观察：

(1) DLRM 的输入具有时间性：DLRM 与传统的计算机视觉和自然语言处理模型不同，它需要对按照时间顺序输入的数据进行训练，以跟上最近的推荐趋势。随着时间变化，一些很久之前输入的嵌入实例变得不那么重要。

(2) 嵌入具有异质性特征：现有的 DLRM 系统通常单独处理每个特征的嵌入表。由于数据分布随时间而变化，这可能导致单个表的利用率不足或者过载。这两种情况都会损害模型的精度。

基于这种观察，在模型训练期间优化每个特征的嵌入，从根本上减少相同精度所需要的嵌入大小的思路就有实现的可能。

然而实际应用中的 DLRM 通常包含数百个稀疏特征和高达数十亿的嵌入行。它们在数百个 GPU 上运行，处理非静态的模型输入，以达到所需的模型执行速度。这些因素导致了在实际训练中对嵌入行进行修剪所面临的以下挑战：：

Q1：[剪哪些] 在上亿级别的嵌入中找出重要的嵌入是非平凡的。并且，在训练过程中非平稳的数据分布导致不同嵌入的访问频率出现时空变化，嵌入的权重也随着训练迭代发生变化，导致嵌入的重要性在训练过程中不断变化。

Q2::：[何时剪] 对 DLRM 进行剪枝可能导致百万级别的嵌入项以及数十亿的嵌入权重需要重新分配内存，而每个训练迭代只需要几百毫秒。因此，虽然过于频繁的剪枝有助于提升模型精度，但是会百倍降低模型训练速度。

Q3：[怎么剪] 现有的 DLRM 系统往往依赖静态的、固定大小的嵌入存储 ，因此难以实现动态的大规模剪枝。

3 Adaembed

AdaEmbed 是一个自动化的训练中剪枝系统，它可以在规模上自适应地优化每个特征的嵌入，以提高模型的精度。与现有的模型修剪工作不同，AdaEmbed 在训练进行时自动识别并保留给定嵌入大小的重要嵌入以提高性能，而不是关注传统模型在训练完成时修剪模型大小。

3.1 Overview

Adaembed 是对现有深度学习推荐模型的补充，结构如图 4，红色部分是 Adaembed 的组成部分。它包含一个中央协调处理器和一系列的分布式 agent 代理。Agent 代理包含 Memory Manager 和 Emb Monitor。

AdaEmb Coordinator：从 agent 收集嵌入的信息，并决定全局剪枝的时机，并协调 agent 完成剪枝。

Emb Monitor：追踪嵌入的重要性，并且定期向 coordinator 汇报嵌入的重要性分析结果。

Memory Manager：管理内存，接收 coordinator 的剪枝指令，执行对本地嵌入权重的剪枝。

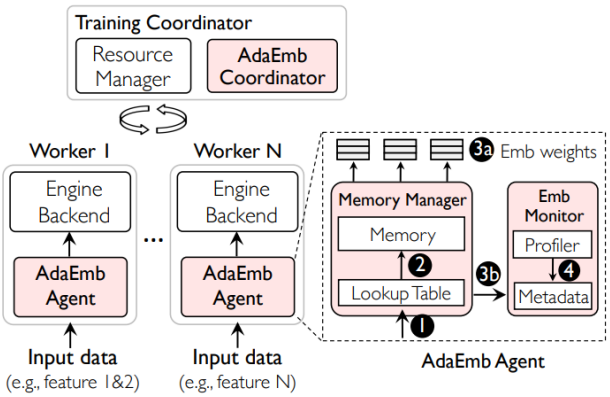

图 4 Adaembed 的结构及执行流程

图 5 的例子说明了 AdaEmbed 的接口，它仅仅通过几行代码支持现有的 DLRM 系统。

```
1  import AdaEmbed
2
3  def dlrm_model_training():
4    # Wrap existing embedding modules
5    emb_agent = AdaEmbed.create_agent(
6      emb_tables=model.embs, pruning_config=config)
7
8    for _ in range(num_iterations):
9      input_ids = get_next_data_batch()
10
11     # Look up physical embedding address
12     emb_physical_ids = emb_agent.look_up(input_ids)
13     feedback = model.train_step(emb_physical_ids)
14
15     # Update embedding importance with feedback
16     emb_agent.update_importance(input_ids, feedback)
```

图 5 Adaembed 支持现有的 DLRM 框架

与目前的 DLRM 部署类似，每个工作节点负责由模型并行性嵌入划分确定的稀疏特征的子集。工作节点负责处理输入的数据。不同的是，输入首先被转发到 AdaEmbed 代理以查找每个嵌入权重的物理地址（第 12 行）。然后这个物理地址被用来获取嵌入权重进行读取和写入操作。模型训练的其余部分与现有设计相同。在每个训练迭代之后，嵌入监视器利用训练反馈更新嵌入的重要性（第 16 行）。它定期对不同嵌入行的重要性进行采样，并将分析结果通知协调器。协调器确定如何根据总嵌入大小修剪嵌入，并指导内存管理器按比例接纳和修剪嵌

入。

3.2 Embedding Monitor

鉴于嵌入大小，用不太重要的嵌入行来换取更重要的嵌入行。这要求从其嵌入权重对模型准确性的贡献以及其物理大小方面来衡量，考虑每个嵌入行的重要性。然而，在训练期间确定最佳的修剪策略是具有挑战性的。首先，模型输出受到输入特征实例及其嵌入权重之间复杂相互作用的影响。即使在训练完成后拥有完整的模型信息，修剪仍然是机器学习中一个基本的尚未解决的问题。其次，在模型训练期间，由于模型输入和嵌入权重的分布存在大量时空变化，如图 6。一旦对一个嵌入的权重向量进行了修剪，在训练继续进行时很难评估其对模型准确性的影响。

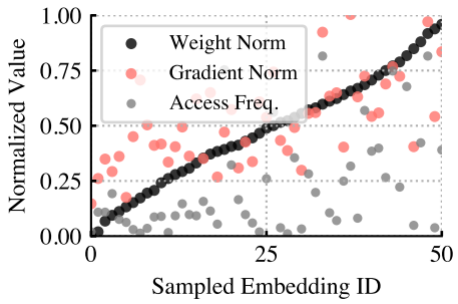

图 6 数据的属性具有异质性

给出的解决方案时，对于同一特征组内的嵌入：直观上，稀疏特征层的输出通常通过取输入嵌入权重的和或者最大值来导出。因此应该保留影响模型输入的嵌入，即频繁访问的并且更影响模型输出，即更大的权重。设置 *EI(i)*用于衡量一个嵌入的重要性。

$$EI(i)_t = freq_t(i) \times \|\nabla g_t(i)\|$$

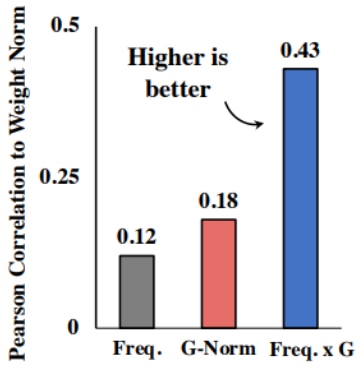

图 7 相关性分析

*EI(i)*由访问频率乘以梯度的 L2 范数得到用于表征重要性。虽然修剪嵌入后没有关于未来权重的信息，但观察到当训练收敛时，频率-梯度组合度量与最终嵌入权重之间存在很强的相关性，如图 7 所示。这是因为具有大梯度的频繁权重更新通常会导致更大的权重。

在具体实施时，为了避免小批量抽样的随机性带来的波动和不确定性，对 EI（i）的计算进行了修改：

$$EI(i)_t = freq_t(i) \times \|\nabla g_t(i)\| + EI(i)_{t-1}$$

在计算本轮 EI 时，需要加上上一轮的 EI。如果上一轮没有访问，则频率为 0，EI 不变。另外为了说明时间变化，每 T 次迭代使得 EI 缩小为 0.8 倍。

对于不同特征组间的嵌入，由于其重要性可能存在数量级的差异，因此采取归一化的方法避免特征之间的比较偏差，即使用嵌入在同一特征中的重要性 95 分位完成归一化。

$$EI(i)/EI_{95th}(feature(i))$$

计算 EI 的开销可以忽略不计，因为嵌入梯度在训练的方向传播期间已经生成。

3.3 AdaEmbed Coordinator

每次 DLRM 进行训练迭代时，都需要更新模型中数百万个嵌入行的重要性，其规模达到 TB 级别。在这种情况下，频繁地进行修剪可以提供更好的决策质量，即始终最大化重要嵌入的数量，以达到可能更高的模型准确度。然而，进行修剪可能需要清理和创建数十 GB 的嵌入权重，这可能需要数秒时间，并且会显着减缓亚秒级的训练迭代速度。

为了找到修剪开销和修剪效果之间的最佳平衡点，AdaEmbed 协调器决定何时进行修剪，以减少修剪的次数，并指示内存管理器在修剪嵌入权重时尽量降低每次修剪的成本。修剪关注的是个体嵌入的重要性排序，而不是它们的动态重要性。因此，AdaEmbed 协调器依赖于所有嵌入的重要性分布，并在重要性分布自上一次修剪以来发生显著变化时启动修剪。为了有效地收集来自数百个机器的重要性分布，每个本地代理都对其嵌入重要性值的小部分进行采样。然后，协调器可以估计有多少嵌入已超出修剪边界，即在重要性分布中处于第 X 个百分位数以上或以下的嵌入行数，自上次修剪以来发生了变化。

具体算法流程，如算法 1：嵌入监视器在每次训练迭代后更新已访问嵌入的重要性，并定期分析嵌入的重要性。分析结果将发送给协调器。如果重

要性分布发生显著变化，协调器将启动新的修剪轮，并将修剪决策通知给内存管理器。然后，每个工作节点上的内存管理器执行修剪，并按比例接纳新的嵌入权重。

Algorithm 1: Pseudo-code of AdaEmbed runtime

```
1:  weight_table ← EmbWeights()        ▷ Physical weight tables
2:  emb_meta ← Init(weight_table)      ▷ VHPI metadata
3:  pruning_start ← false              ▷ Enforce pruning or not
4:  Function UpdateEmbs (input_ids, feedback):
        /* Monitor: Update embedding importance
           asynchronously to model training.        */
5:      UpdateImport(input_ids, feedback)
6:      if pruning_start == true then
7:          EnforcePruning()           ▷ Stall training
8:          pruning_start ← false
9:  Function MonitorImportance (ProfilingInterval Δ):
        /* Coordinator: asynchronously inspect big changes on
           the importance distribution via profiling across
           workers.                                 */
10:     last_dist ← null
11:     while training == true do
12:         if mod(current_time, Δ) == 0 then
13:             cur_dist← ProfileImportance()
14:             pruning_start←Diff(last_dist, cur_dist) > p
15:             last_dist ← cur_dist
16: Function EnforcePruning():
        /* Memory manager: Identify embedding rows to admit
           and prune subject to the given embedding size.  */
17:     admit_emb, evict_emb ← IdentifyRecycleEmbs(
18:                 emb_meta, weight_table.size)
        /* Redistribute the lookup mapping from the embedding
           ID to the weight vector, whereby admitted embedding
           rows can recycle the weight vector of pruned ones. */
19:     RedistLookup(emb_meta, admit_emb, evict_emb)
        /* Reset embedding weights for admitted embeddings. */
20:     weight_table.ResetEmbs(admit_emb)
```

根据统计学原理，预设分位数 c=5%。为了满足偏差小于 1%，从 1 亿嵌入中抽样 5 百万个，这种开销对于 DLRM 是完全能够接受的。如图 8，抽样 5M 个的方案使得错误率和开销处于一个平衡点。

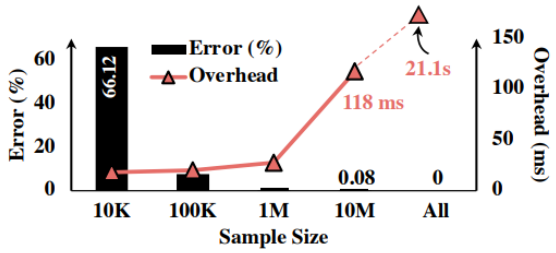

图 8 抽样结果分析

3.4 Memory Manager

由于嵌入权重的重新分配比每次训练迭代慢数百倍，减少所需的修剪轮数在实践中仍然远远不能实现可忽略的开销。此外现有的 DLRM 系统往往依赖静态的、固定大小的嵌入存储，因此为了避免密集的内存重新分配，AdaEmbed 的内存管理器采用虚拟散列物理索引（VHPI）设计，将嵌入的管理与其物理权重向量分离，从而 AdaEmbed 可以回收不同嵌入的权重向量，以便为各种现有的嵌入设计提供有效的修剪。

VHPI 主要由两部分组成，如图 9 所示：查找表(lookup table)：查找表存放每个嵌入项的元数据，包括嵌入的重要性（a float 32）和权重向量的物理地址（a int 64）。与权重向量相比，引入的内存可以忽略不记。权重表(weight table)：真正存储 embedding 权重向量，与现有的 DLRM 系统的嵌入表保持相同，但在内存管理器的协调下跨功能共享。

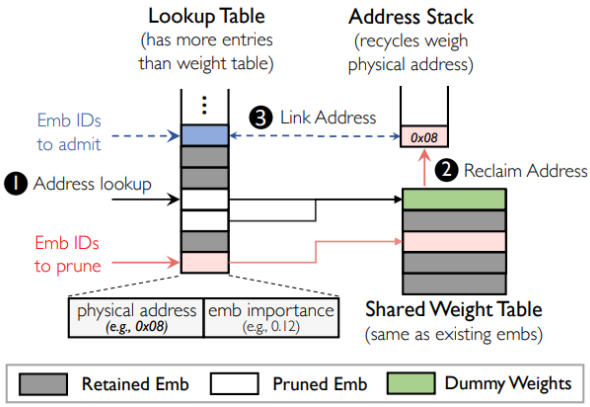

图 9 VHPI 的组成以及运行流程

被修剪的嵌入的权重向量不会被保留，而所有嵌入的元数据始终保留在查找表中。因此，查找表可以包含比权重表更多的条目即嵌入 ID。这能够自适应地确定嵌入和权重向量之间的关联，以便回收权重向量。此外，通过使查找表非常大以容纳许多嵌入条目，而不扩展权重表，可以改善模型准确性，减少哈希冲突。

内存管理器在运行时执行两个用于权重修剪的原语操作：

地址查找：它针对每个嵌入 ID 查找物理权重地址，以访问其嵌入权重。❶如果该嵌入行已被修剪，为了避免破坏现有设计（例如，由于修剪而导致的权重缺失），查找将返回一个共享的物理地址，指向包含常数零的权重向量。

权重分配：该部分执行修剪决策，以修剪和接纳嵌入。❷对于要修剪的嵌入行，VHPI 首先解除

链接并回收该嵌入权重的当前物理地址。然后，它将被修剪嵌入的查找条目地址设置为共享虚拟向量的地址，重定向未来的访问。❸为了接纳一个嵌入，VHPI 弹出一个可用的物理地址，并将该地址与查找条目链接，从而回收物理内存。与此同时，内存管理器将重置权重向量的值，以清理先前修剪的权重状态。

对于接纳的嵌入来说，重新设置权重值并不是一件简单的事，研究了四种流行的重置权重向量的策略图 10：（1）w/o set：继承被修剪的嵌入的权重而不对其进行重置；（2）权重恢复：将先前被修剪的权重逐出到额外的存储（例如磁盘），并在重新获得该嵌入时恢复权重；（3）原始初始化：像在训练开始时那样随机初始化嵌入权重；（4）零初始化：将嵌入权重重置为零。

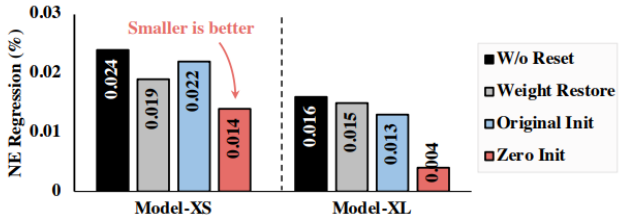

图 10 重设权重策略对比结果

文章主张将权重向量值重置为零，因为这样可以避免大量噪音，同时允许录入的嵌入从头开始学习。事实上，真实环境评估报告显示零初始化优于其他替代方法。

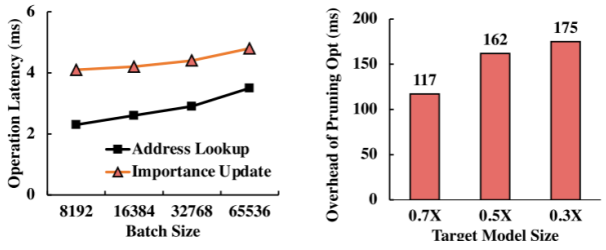

图 11 VHPI 操作的开销

图 11 说明了 VHPI 的操作的开销估计。在所有涉及到的操作中，每次迭代都会有地址查找和重要性更新，仅仅只有几毫秒的开销。虽然修剪嵌入需要花费几百毫秒，但是几百次迭代中可能只发生一次。总的来说，在大规模部署中，VHPI 涉及到的操作并不会引入端到端的开销延迟。

4 评估

实验设置：在评估中使用了来自工业 DLRM 系统的模型和数据。图 12 展示了模型统计数据。它们涵盖了不同规模和推荐任务，包括点击率预测和排名。我们将每个模型训练 14 天的数据，以获取模型的生命周期 NE，该指标表示整个训练过程中的累积模型准确性，然后在第 15 天的数据上对模型进行测试，以获取评估 NE。每天都有 TB 级别的数据输入。

每个模型的训练批处理大小为 65536，数十个 GPU 节点，每个 GPU 节点具有 8 个 A100 GPU，每个 GPU 的显存为 40 GB。这些 GPU 通过 200 Gbps 的 RoCE NICs 进行互连。

Model	# of Sparse Features (Approximate Value)	Raw Emb Size (Approximate Value)	# of GPUs	w/ Same Model NE		w/ Same Emb Size	
				Memory Saving	QPS Speedup	Avg. NE Gain (%)	QPS Overhead
Model-XS	1000	200 GB	32	≈ 35%	1.1×	0.015	0.4%
Model-S	600	350 GB	32	≈ 45%	1.2×	0.018	0.2%
Model-M	1000	1 TB	64	≈ 40%	1.2×	0.028	1.6%
Model-L	1000	1.1 TB	64	≈ 55%	1.3×	0.021	1.3%
Model-XL	800	1.5 TB	128	≈ 60%	1.3×	0.026	1.1%

图 12 评估结果

因此可以得出 AdaEmbed 减少了资源需求并提高了 QPS，在相同尺寸下使用 AdaEmbed 获得了更好的 NE，并且 AdaEmbed 引入的开销可以忽略不计。

如图 13 所示，给出了具有代表性的三个尺度的模型进行细化分析，其中 w/ AdaEmbed 表示使用 AdaEmbed 自适应修剪嵌入；w/o AdaEmbed 表示不适用 AdaEmbed 直接删去访问频率低的嵌入。评估结果说明了使用 AdaEmbed，模型可以在相同的嵌入大小下实现 0.011-0.077%更好的 NE。注意到：

（i）AdaEmbed 相对于基线，在各种目标嵌入大小和不同模型下，能够始终实现更好的 NE；

（ii）即使与完整模型相比，也可以通过较小的嵌入大小（例如，0.7 倍模型）获得 NE 增益。这是因为 AdaEmbed 能够自动学习更好的特征嵌入。同时，修剪不太重要的嵌入可以减少模型过拟合，从而提高模型的泛化能力；

（iii）生命周期 NE 增益比评估 NE 增益更明显，因为前者更接近在线部署即实时数据上的重新训练，在这种情况下，AdaEmbed 能够适应最新的数据分布。

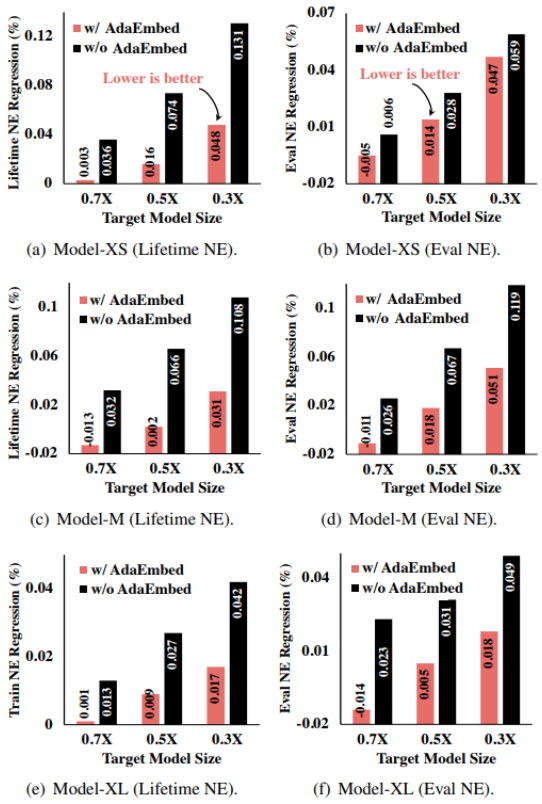

图 13 三个代表性模型的评估结果

性能分解对照实验：分别设计分为两个变体：（1）AdaEmbed w/o Norm：在组修剪中禁用重要性归一化;（2） AdaEmbed w/o Group：完全禁用组修剪。评估结果如图 14，归一化和组修剪都有助于更好的 NE。因为组修剪允许更大的灵活性，可以使用共享的巨大权重表调整每个特征的嵌入大小，此外重要性归一化有助于通过在全局比较嵌入重要性时优先考虑每个特征的重要嵌入来减少特征间的异质性。

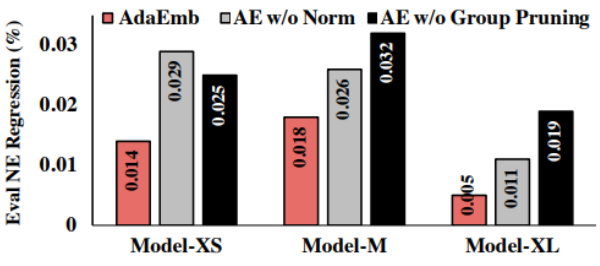

图 14 AdaEmbed 设计的性能分解实验

修剪时机对照实验：结果如图 15 所示。每分钟修剪一次导致过于频繁的训练噪声影响了瞬时嵌入重要性，而每天修剪一次未能及时接收重要的嵌入导致效果不佳，而 AdaEmbed 的选择性修剪通过依赖于运行时的整体重要性分析来实现更好的性能。

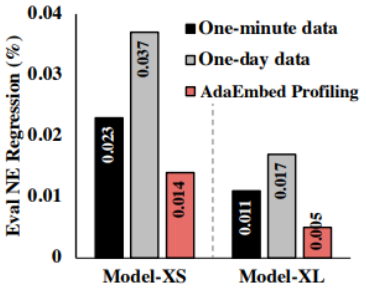

图 15 修剪间隔的对照实验

不同数据集影响实验：图 16 报告了模型-S 在三个不同数据集上的 NE 性能。每次训练覆盖了 10 天的训练数据，并报告了第 11 天数据的评估 NE。尽管随着数据分布在不同日期之间的变化，NE 增益略有不同，但 AdaEmbed 在保持 NE 不降的情况下始终实现了 50%的内存节省。

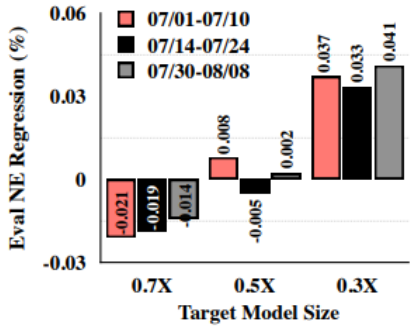

图 16 不同数据对照实验

重要性标准对照实验：在训练 10 天的数据中使用不同的嵌入重要性设计考虑使用频率、梯度以及它们的组合作为嵌入的重要性度量。如图 17 注意到频率-梯度组合优于其他替代方案。因为频率-梯度的组合与最终嵌入权重有更强的相关性。与此相反，访问频率和梯度仅分别考虑数据分布和模型特性，而 DLRM 的准确性取决于这两个方面。

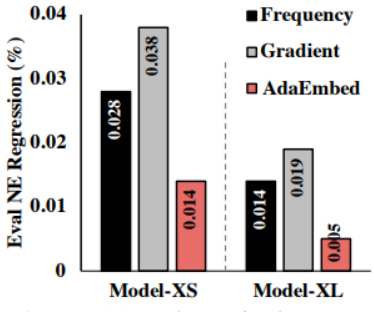

图 17 重要性衡量标准对照

最后，将 AdaEmbed 与其类似的训练后修剪（PTP）进行了比较。在模型训练完成后，PTP 通过修剪重要性设计衡量的不太重要的嵌入来减小嵌入大小。实际上，在实际应用中部署 PTP 通常是不切实际的，并且无法在模型训练期间实现内存节省和 QPS 的改善。图 18 显示了在相同的嵌入大小下，AdaEmbed 可以实现比 PTP 更好的 NE，因为在训练中的设计可以根据运行时的模型性能进行调整，并持续优化嵌入。

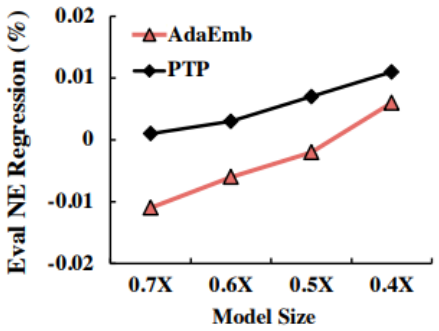

图 18 训练中修剪与训练后剪枝对比

5 总结

现代 DLRM 发展迅速，在实际部署中面临着 QPS 和 NE 不能同时兼得的问题，现有的研究思路是致力于给定嵌入进行优化执行速度，而本文介绍了 AdaEmbed，一种用于提高 DLRM 准确性的训练中嵌入修剪系统。AdaEmbed 识别对模型准确性更重要的嵌入行，然后自适应性地修剪不太重要的嵌入，以控制总体嵌入大小的规模。评估表明，AdaEmbed 可以通过自动学习使用更好的每个特征嵌入来减少手动工作，在部署中节省了 35-60%的嵌入大小，并在模型准确性和模型执行速度上实现了明显的改进。

参 考 文 献

[1] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie (Amy) Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K. Ardestani, Xiaodong Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, KR Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair, Bharath Muthiah, Mahmoud khorashadi, Pallab Bhattacharya, Petr Lapukhov, Maxim Naumov, Ajit Mathews, Lin Qiao, Mikhail Smelyanskiy, Bill Jia, and Vijay Rao. Software-hardware co-design for fast and scalable train-ing of deep learning recommendation models. ISCA, 2022.

[2] Chijun Sima, Yao Fu, Man-Kit Sit, Liyi Guo, Xuri Gong, Feng Lin, Junyu Wu, Yongsheng Li, Haidong Rong, Pierre-Louis Aublin, and Luo Mai. Ekko: A Large-Scale deep learning recommender system with Low-Latency model update. In OSDI, 2022.

[3] Minhui Xie, Youyou Lu, Jiazhen Lin, Qing Wang, Jian Gao, Kai Ren, and Jiwu Shu. Fleche: An efficient gpu embedding cache for personalized recommendations. EuroSys, 2022.

[4] Chunxing Yin, Bilge Acun, Carole-Jean Wu, and Xing Liu. TT-Rec: Tensor train compression for deep learning recommendation models. In MLSys, 2021.

[5] Minhui Xie, Kai Ren, Youyou Lu, Guangxu Yang, Qingxing Xu, Bihai Wu, Jiazhen Lin, Hongbo Ao, Wanhong Xu, and Jiwu Shu. Kraken: Memory-efficient continual learning for large-scale real-time recommendations. In SC, 2020

[6] Weijie Zhao, Deping Xie, Ronglai Jia, Yulei Qian, Ruiquan Ding, Mingming Sun, and Ping Li. Distributed hierarchical gpu parameter server for massive scale deep learning ads systems. MLSys, 2020.

附录 汇报记录

问题 1：你觉得在设计这个系统中的三个挑战中，哪一个更困难，如何实现？

回答： 我认为对于现有的 DLRM 系统，实现动态大规模剪枝是一项具有挑战性的任务。这主要是因为现有的 DLRM 系统通常采用静态、固定大小的嵌入存储结构，难以进行动态剪枝。要实现剪枝，就必修修改 DLRM 系统的存储结构，以实现对嵌入的动态存储。这种改动不仅需要满足动态剪枝的要求，而且避免引入大量密集的内存分配操作，因为这种操作可能会对 DLRM 系统造成严重影响，比如效率变低。

为了解决这一挑战，文章提出了巧妙的虚拟物理散列索引结构。该结构在进行嵌入操作时，避免了对内存的大规模直接修改。具体而言，通过对地址表（lookup table）进行操作，即地址查找和修剪操作，来避免直接对内存进行大量修改。文章介绍了 VHPI（地址查找和重要性更新）的作用，这两种操作发生在每次迭代中，但仅需要数毫秒的时间。另外，修剪策略无需重新分配内存，只需链接或断开 lookup table 与 weight table 的链接即可。虽然每次修剪需要处理数亿的嵌入，但引入 VHPI 使得这些操作仅需要数百毫秒的时间。此外，修剪并非每轮都发生，而是每隔几百轮进行一次，这种开销对于现代 DLRM 系统是可以接受的。

问题 2：在重要性的衡量中，为什么选择访问频率与梯度 L2 范数的乘积形式，这是基于什么考量？

回答：文章好像没有给出为什么选择这两个指标的乘积形式来衡量重要性。文章只给出了二者乘积组合形式与重要性的相关系数为 0.43，单独频率是 0.12，单独梯度是 0.18。文章认为 0.43 已经可以视为一种在中等和强相关范围的相关。

我个人猜测选择乘积形式，不选其他形式的原因是相对于线性组合，乘积形式更能放大相关性；相对于非线性组合（指数，幂函数等形式），乘积的形式运算更加方便引入的开销更小一些。当然这只是个人猜测，具体结果需要根据实验来判断。