Learned Index Benefits: Machine Learning Based Index Performance Estimation 论文学习记录

LIB，includes knowledge such like Database, Machine learning. From my personal view, I like ML most, actually I don’t like those fields intersecting with DB…… and in fact, this paper seems base on ML and most contents of it is about ML, but what interesting is that, it’s a course task from Introduction of Database (honor) in FDU……

I have a presentation about this paper with my friend, he is very cool and intelligent, good partner. it’s actually a f**king daunting task and I feel very bothered and annoyed…… because most of my classmates have started their happy summer holiday >>>…… whatever

以下是我对这篇论文的学习记录，恳请读者批评指正。

## 0. Abstract

关系型 DBMS 中，索引的选择至今仍是非常麻烦的事情。要找某个 workload 下最佳（近似最佳）的索引结构→精确且高效的量化每个候选索引配置的优劣是必不可少的步骤。显然遍历每个候选索引执行查询来判断优劣是愚蠢的，目前大多数 index tuner 都是依靠优化器的”what-if” API 来进行成本估计(cost estimation)。

（supplement：**index tuner**（索引调优器）用于优化数据库索引配置的工具。target 是根据特定 workload（具体表现为查询和更新操作）来选择最优的索引配置，以提高查询性能并减少执行时间。这东西没啥，工作方式就是分析 workload 生成候选索引，然后评估看行不行，行就用。目前很多 index tuner 评估就用的是”what-if” API，这东西相当于是 vivado 跑个仿真的感觉。它允许 index tuner 在不实际创建索引的情况下模拟和评估候选索引的效果。通过 **what-if API**，数据库优化器可以回答“如果有某个特定索引存在，查询计划和成本会是什么样”的问题。主要做**成本估算**（评估在特定索引存在的情况下，每个查询的执行成本。这是基于**数据库优化器的成本模型**，而不是实际执行查询）和**性能预测**（预测在不同的索引配置下整体性能变化））

然后论文说这种”**what-if” API 方式有两个显著问题**：一是容易出错，显然的，毕竟都是估计。二是为每个候选索引都生成查询计划和代价估计依旧很耗时耗算力。（所有论文都有的小对比 hhh）然后提出了他们做出来一个高效的端到端的基于机器学习的索引效益估计器。具体还介绍了一下细节：

**特征提取**(feature extraction) **编码技术**(encoding)——这就可以做到不用”what-if”而能对每个候选索引配置生成查询计划。

**注意力机制**(attention mechanism)——解决索引交互问题(index interaction problem)。

**迁移学习**(transfer learning)——提高这个估计器的学习能力，以适用新的数据库。

最后说他们跑实验结果非常好，准确度和效率都要超过”what-if”方法。

## 1. Introduction

这个找索引(index selection problem (ISP))的问题啊，其实是个 **NPC 问题**（计算理论刚考完不久复习一下：NP-Complete 也就是 NP 且 NP-hard，多项式时间可验证/多项式时间可被 nondeterministic TM 判定，所有 NP 问题可多项式时间归约到此问题。 抽象定义……也就是，多项式时间不可解，通俗说就是没法编程编出一个高效算法来解决这个问题。(NP≠P 的世界)一般这种问题是非常令人难绷的问题，但是总有 estimation，总能不停优化）

这里对”what-if” API 也只做了个引用，有个词说的比较关键：statistical information。这东西依靠一个统计得来的信息作为模型，或许整体不错，但出错或性能低确实是难免的。然后说到目前有些方法对这个 API 的 cost model 使用 ML 方法训练尝试优化，作者认为其实是没有前途的。其次，对每个候选索引调用”what-if”非常耗时，90%以上的总 index tuning 时间花在调用这个 API 上，作者认为这必然会有时间运行上瓶颈，不管再怎么优化（这一段引用了很多论文佐证）。其实逻辑上想一想确实，说的有道理。

然后，作者说道他们这个工作应该是第一个基于机器学习的量化索引效益的方法，主要要考虑四个方面：accuracy（毋庸置疑最重要） generalization（应该在未出现的 query 上也表现良好） efficiency adaptability（这模型也应该适应好新的数据集）

为此，提出了端到端的基于机器学习的索引效益估计器（**Learned Index Benefits (LIB)**）

一、 作者将索引效益估算问题表述为一个回归任务(regression task)，其中任务的目标被定义为在索引配置实现后的**规范化成本差异(normalized cost difference)**。

二、 为了从查询计划和索引配置中提取和表示特征，定义一个新概念——**索引可优化操作（index optimizable operations）**，并提出了一种新颖的特征化方法，该方法将特征表示为一组索引可优化操作，而不需要对每个索引配置进行“假设”调用以生成查询计划。

三、 作者提出了一种新颖的**特征化方法**，将特征表示为**一组索引可优化操作**，而不需要对每个索引配置进行”what-if”调用以生成查询计划。

四、 作者提出使用基于**注意力机制**的神经网络来学习**索引可优化操作之间的关联**，并处理**索引交互问题**。

五、 当 LIB 应用于不同数据库时，利用**迁移学习**来减少模型重新训练的需求：不是从头开始训练 LIB，而是将已在其他数据集上训练的 LIB 用作 pretrained model，并用新数据对整个模型进行微调(fine-tuning，居然还涉及到上个论文看的 fine tune 相关技术，如果模型过大还可以用 IPA 技术代替朴素 fine tune 哈哈)。

## 2. Problem Statement

本论文基本概念理解——

**工作负载(workload)**。工作负载 W = {$q1$, $q2$, ..., $qn$} 是一个包含 n 个查询的集合，这些查询针对关系数据库中的一个或多个表。查询 $qj$ ($qj \in W, 1 \leqslant j \leqslant n$) 由表中的一组列和用于连接或筛选的条件集所组成。

**二级索引(secondary index)**。是一种数据结构，包含表中的部分属性，并且有指向包含特定键值的所有记录的指针。

**索引配置(index configuration)**。索引配置 $ci$ 是一组索引。每个索引可以包含表中的一个或多个列。

**索引（配置）候选(index candidates)**。索引（配置）候选 C = {$c1$, $c2$, ..., $ck$} 是一组由索引选择算法评估的 k 个潜在索引配置。通常它们是基于**启发式规则(heuristic rules)**或查询相关性生成的。

**索引选择(index selection)**。是指在 C 中找到最优索引配置，以在某些约束条件下（例如有限的存储预算或索引选择的运行时间）提升数据库的查询性能。在索引选择过程中，每个索引配置的**收益(benefits)**量化了其对 workload 的影响，用于索引候选的比较。

DEFINITION1. **索引收益(index benefits)**。索引配置 $ci$ 在查询 $qj$ 上的索引收益定义为：“配置 $ci$ 后的执行成本”与“不使用索引时的执行成本”的差值（查询执行的成

本减少量(cost reduction)）公式形式化：

$$cr_{i,j} = Cost(q_j \mid \emptyset) - Cost(q_j \mid c_i)$$

cr 就是 cost reduction. Cost(qj | ci) - 有 ci 这一索引配置时的 qj 查询执行成本。

∅ 标识无索引使用时的查询执行。

作者将索引收益估计问题转为一个 regression problem，直接预测在特定索引配置下查询的 cost reduction（或是 benefits）。
作者首先反面说了一段，直观地想应该训练一个 learned cost model 来估计每个查询在索引配置下的执行成本，并通过计算两种成本之间的差异来计算索引收益。但这模型非常难构建，作者又引用了几篇论文，说因为查询执行时间可能从几毫秒到几千秒不等，还取决于硬件、数据库大小等条件，同时容易出错。**最后提出他的方法**：为了提供一个更好的衡量收益的指标，他提出
**DEFINITION2. 减少比率（reduction ratio）**其实就是归一化 cr。公式定义为：

$$rr_{i,j} = \frac{Cost(q_j \mid \emptyset) - Cost(q_j \mid c_i)}{Cost(q_j \mid \emptyset)} = \frac{cr_{i,j}}{Cost(q_j \mid \emptyset)}$$

rr（reduction ratio） 以 rr 作为目标输出时，未使用索引的常量查询成本（$Cost(qj \mid \emptyset)$）可以作为候选比较的参考。因此，能够训练一个准确的索引收益估计模型 LIB，该模型能够直接最小化比较索引配置时的错误。

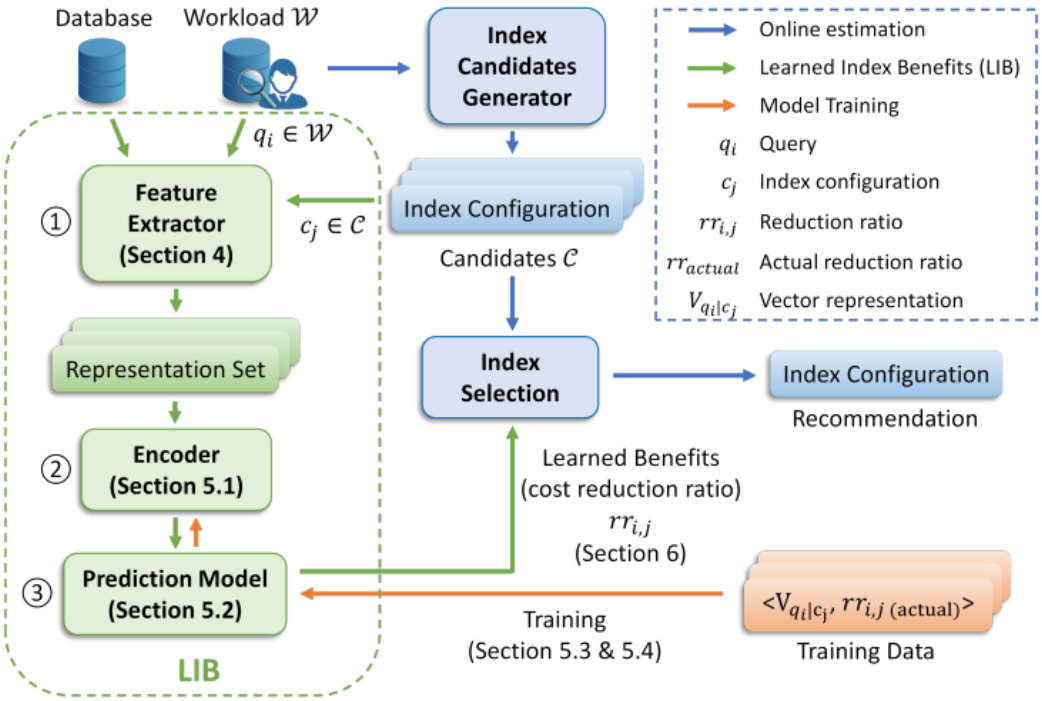
**核心问题陈述**：Index Performance (or Benefit) Estimation (IPE)
给定一个 workload W = {$q1, q2, \ldots, qn$} ——包含 $n$ 个查询操作，
一系列索引配置候选 C = {$c1, c2, \ldots, ck$ } ——包含 $k$ 个索引配置候选，
目标是估计成本的 reduction ratio（相当于是归一化的索引收益），这是一个嵌套循环，for 所有 ci 和 qj in C 和 W，计算 rr_i,j
总的来看，以表示查询 $qj$ 和索引配置 $ci$ 的**提取特征(extracted features)**作为**输入**，LIB 对每个 rr 进行估计，模型的目标形式化为一个 regret minimization problem，其中特定索引配置 $ci$ 对查询 $qj$ 的 regret 定义为索引配置的**实际成本 rr 与估计成本 rr** 之间的平方残差(squared residual)

$$\underset{q_j \in \mathcal{W},\, c_i \in C}{\text{minimize}} \quad \mathcal{R}_{q_j|c_i} = \left(rr_{i,j(actual)} - \widehat{rr}_{i,j(estimated)}\right)^2$$

Minimize 也就是使得估计成本 rr 尽量接近实际成本 rr。
理解：这一段的**目的就是为了训练个模型精确的估计索引的 benefits**，首先将 benefits 量化为 cr，一个 index 与无 index 时 cost 做差，但这样还是效率低，显然，对任意两个 ci cj 在 C 中要进行 benefits 比较，都要计算一遍，或者进行存储，有时间或空间

的额外开销。因此提出归一化 cr 也就是 rr，每个的效益同无 index 的效益比较，无 index 相当于一个基准。再形式化为真实 rr 与估计 rr 的平方残差，最终最小化他们的总和。（这个真实 rr 就相当于喂给模型的训练集）

3. Learning Framework Overview




这图非常之清晰

Feature extractor 就是用于将一个查询和索引配置的有用特征提取出来，表示为一系列向量作为输入。 简单理解：预处理、减少后续运行时间、向量化。

Encoder 聚合上面的一系列向量，无损表示为一个单独向量，使用注意力机制。

Prediction Model 就是 2 末尾讲的那些，它以 encoder 的向量作为输入，输出一个 rr 作为结果来衡量这个 index 在这个 query 下的 benefits。

简介：以上橙色部分训练模型是 **offline model training**，完全独立于其它工作流。蓝色部分是 **online estimation**，当一个 index tuner 被调用来给出推荐索引的时候，它先给出一系列候选的索引（同以前一样），其次，不用传统的”what-if”方法，而是嵌入这个 estimation 估计每个候选索引的 rr，根据给出的 rr 选择最好的候选索引。（LIB 与索引选择阶段也是完全独立的，它只是给出 rr，但工作流互不相干）

4. Feature Representation

对输入（一个 query 和一个 index configuration）提供一种特征表示，我们应当对**影响原始查询的索引成本减少的因素进行特征化**。作者首先说第一个问题，目前**特征化方法(featurization)**一是太过依赖于”what-if” API（前面也说到这东西存在瓶颈），第二个问题还是性能不行，无法很好地捕捉与 IPE 相关的信息。然后花了一段讨论几种现有的特征化方法，总之是为了衬托自己的新方法。新方法：

**为解决第一个问题**：仅关注那些将被索引配置改变的原始查询计划（即，没有任何索引配置的计划），而不是像以前的工作那样调用“what-if”以生成每个索引配置的计划。

举例： 图 2 展示了 TPC-DS 中的一个查询及其索引配置。

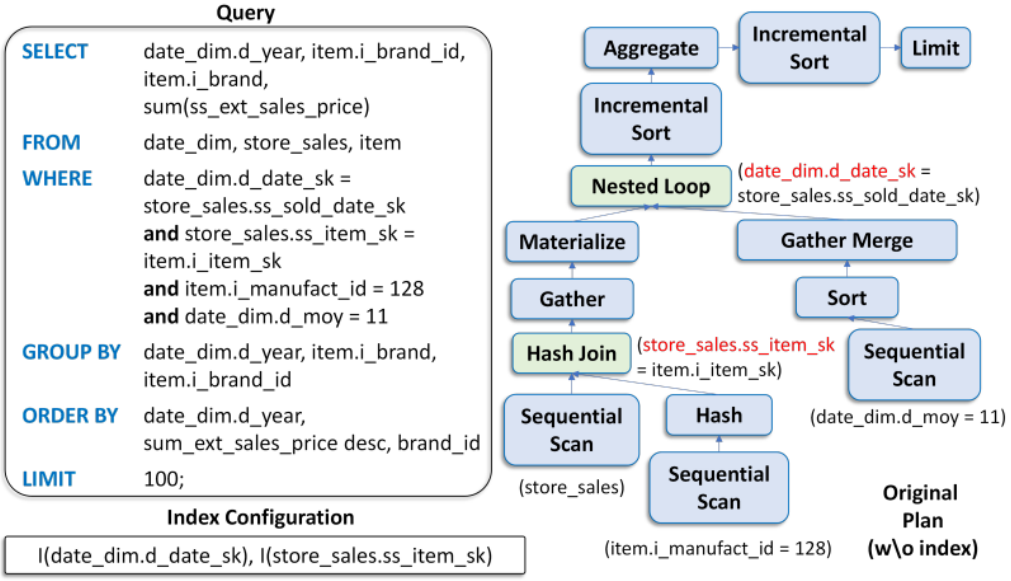



**Figure 2: Example of Feature Extraction**

我们不生成新的查询计划并研究新计划与原始计划之间的差异，而是关注查询显示出来的原始计划。为了学习索引配置对原始计划的缩减比率(rr)，我们可以特征化查询访问（即顺序扫描操作）和连接（即哈希连接和嵌套循环操作）数据的信息，以及索引配置，这些都可以为模型学习其影响提供见解。（这里相当于画了个查询的强语法树，把查询细分成了多个操作不同步骤，为了提取其中和索引配置有关的操作）

**为解决第二个问题**：将带有索引配置的原始查询计划转化为一组向量。（这里作者引用几篇论文说明了：索引通过改变操作内的数据检索方法来优化操作，索引带来的成本节约仅取决于扫描范围的缩小，其影响独立于操作在查询计划中的位置。）基于以上论文的论证，作者将**查询计划特征化为一组独立的操作**，而不包含任何计划结构信息。其次，因为**查询计划中并非所有操作都会受到特定索引配置的影响**，我们可以仅特征化相关操作的部分。（譬如 Sequential Scan 大家都要做，诸如此类，只会占空间和算力的操作，与具体选择什么索引配置无关，就可以舍弃掉）定义新概念——称为**索引可优化操作（Index Optimizable Operations, *O_IO*）**

DEFINITION3. 索引可优化操作（*O_IO*）。*O_IO*(those operation that is index optimizable) 是查询执行计划中的操作，其数据源包含由索引配置所 index 的列（索引都是建在某些属性上的）。*O_IO* 由以下三项关键信息定义：

O_IO = [OI, DS, IC]

在图 2 的例子中，由于嵌套循环（Nested Loop）和哈希连接（Hash Join）操作的数据源涉及索引列“date_dim.d_date_sk”和“store_sales.ss_item_sk”（索引是建在这两个属性上面的），它们被识别为 *O_IO*（这里就比较清晰了，对于其它属性，他们上面没有建索引，和无索引查询（原始查询）一样，有没有这个索引配置都无所谓，因此他们算是无关因素，特征化的时候不用考虑进来）。

*O_IO* 是索引可优化操作

***OI*(operation information)操作信息** -- 操作的类型将决定索引如何改变操作。例如，哈希连接这样的连接操作可以使用索引根据从另一张表选择的行来检索连接行，这替代了哈希函数的使用。扫描操作，它可以直接使用索引根据谓词访问数据来替代全表扫描。

其次，为了从优化器中学习，利用由工业级查询优化器生成的信息，如估计的基数，这提供了有关检索的元组数量的信息。这些信息对于 IPE 至关重要，因为访问的数据量越小，索引的有效性就越高。

***DS*(database statistics)数据库统计信息** -- 索引列的分布和统计信息在 IPE 中起着重要作用。例如，索引对列的效益与列中不同值的数量密切相关。对于具有少量不同值的列，使用索引检索数据可能不会很有效，因为仍然需要大量的扫描。

***IC*(index information)索引信息** -- 不同类型的索引（单属性或多属性）具有不同的性能。对于多属性索引，除了编码索引的类型外，还可以编码索引中列的顺序，因为索引在最前面（最左边）的列上的有效性最高，并且性能随着顺序增加而降低。

综上所述，作者将带有索引配置的原始查询计划特征化为一组索引可优化操作 {*O_IO*}，供 LIB 学习 IPE。

以上全是分析与定义，接下来开始详细说明特征提取(feature extraction)的步骤——

**对于 OI**，首先将索引可优化操作分类为**五种类型**（连接（join）、排序（sort）、分组（group）、范围扫描（scan_range）、等值扫描（scan_equal）。在这里，我们进一步将扫描操作分为范围扫描和等值扫描。这是因为具有范围谓词和等值谓词的扫描操作将具有不同的索引扫描范围，因此索引的有效性也不同。对每种操作类型进行**独热向量编码**(one-hot，即，向量中只有某一个维度值为 1，其余全是 0)。此外，我们还将每个索引可优化操作的估计基数作为特征维度，因为它与索引性能估计相关。最后对估计基数进行对数转换以减少其偏斜性和变异性。

**对于 DS**，我们**编码索引列的行数、NULL 值比例**以及**不同值的比例**。这些统计信息将影响索引的性能。对于 NULL 值比例和不同值比例，它们的值介于 0 和 1 之间，其中 1 表示所有值都是 NULL 或唯一的，0 表示相反。对于行数，也应用对数转换。

**对于 IC**，我们使用索引的类型和多属性索引中索引列的顺序作为特征属性。我们采用**独热编码**来表示索引的类型（单属性或多属性）。对于多属性索引，我们使用一个整数值的特征来指示索引中索引列的顺序。例如，对于多属性索引 $I(A, B, C)$，索引列 $B$ 的顺序是 2。

总结来说，每个 O_IO 的特征向量是如下形式——

**O_IO^i = [O_t, log(card), log(rows), dist_frac, NULL_frac, I_t, I_o] ∈ R^12**

O_IO^I 表示{O_IO}中的第 i 个向量， **O_t ∈ R^5** 表示操作类型(operation type)编码，由于采用独热编码，5 种类型，因此 5 维向量， log(card)表示 OI 的估计基数(cardinality)， log(rows), dist_frac, NULL_frac 就是数据库统计信息的**行数、不同值比例和 NULL 值比例**， I_t ∈ R^2（type）和 I_o（order）分别特征化索引类型和索引顺序。

举例说明——

以下图片形象说明了特征化的思路。为了将带有索引配置的查询表示为一组向量，**首先生成在没有使用索引的条件下查询执行计划**。对于同一查询的不同候选索引配置，我们可以重复使用先前生成的查询计划。然后，根据索引配置，我们识别出索引列。接下来，利用这些列，我们**在查询计划中找出索引可优化操作**。如上所定义，当操作的数据源或条件包含索引列时，我们将其标识为 *O_IO*。以上一张图片作为例子：操作嵌套循环（Nested Loop）和哈希连接（Hash Join）都是 *O_IO*。对于每个索引配置，我们可以通过**一次扫描**查询计划找出所有的 *O_IO*。之后，我们从数据库目录（例如 PostgreSQL 中的 pg_stats 。这个一般数据库系统会实现）中**提取每个索引列的数据库统计信息**。最后，我们按照上述讨论的方式表示每个特征维度，并将它们连接成每个 *O_IO* 的特征向量。(PS：对于一个索引可优化操作，可能会**有多个索引用于优化它**。在这种情况下，

作者将它们分别表示为单独的特征向量，其中每个特征向量表示一个索引对该操作的影响。然后，查询和索引配置被表示为一组特征向量，{$O\_IO$}。）

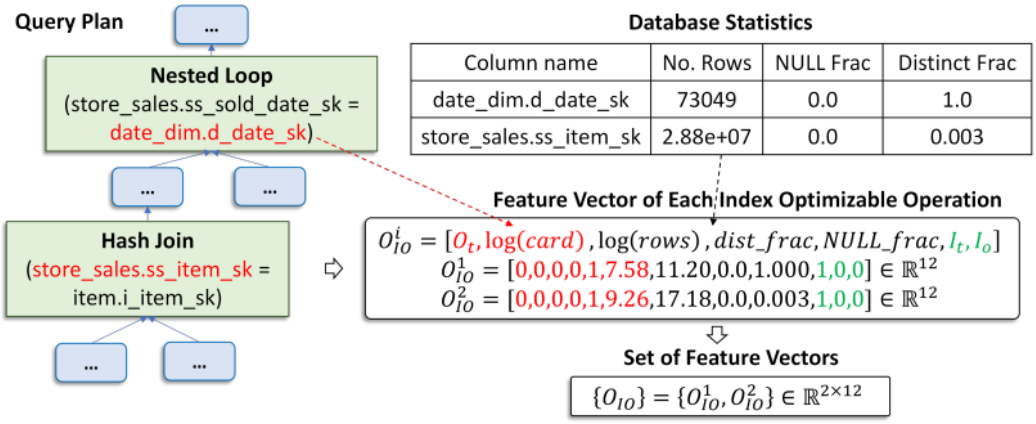



**Figure 3: Example of Proposed Featurization Method**

## 5. Learning Index Benefit Estimation Model

### 5.1 aggregation encoder

目的：将一系列向量（上面传来的{O_IO}）编码转成一个单独的向量。

这里的 encoder 主要包含以下三个器件：embedding（嵌入层），learned representation（学习表征），pooling（池化层）　　如图 4 所示——

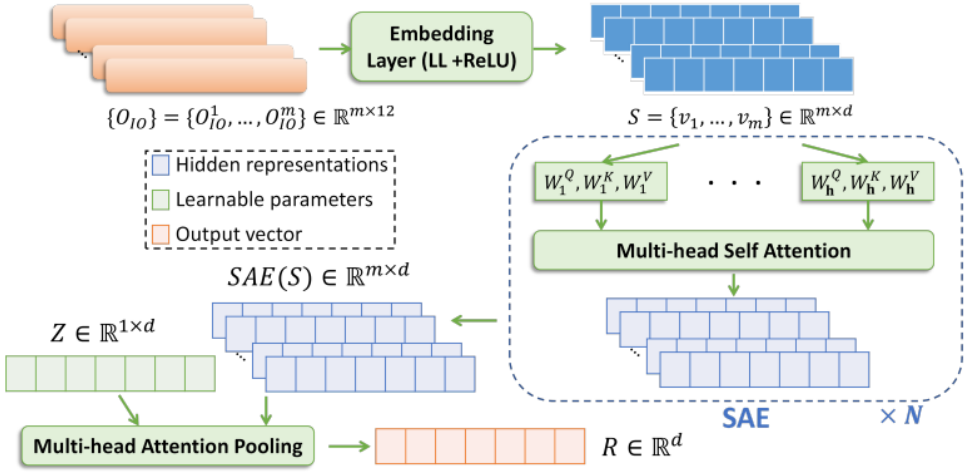



**Figure 4: Overview of Encoder**

**Embedding.** 将上一层特征提取器生成的稀疏向量 $O\_IO$ ∈ R^12 转换为稠密向量 $vi$ ∈ R^$d$，这里这个 d 是嵌入维度(embedding size)，不妨假设传入的{$O\_IO$}中包含 m 个 O_IO 向量，即 {$O\_IO$} ∈ R^$m$×12。考虑到向量值的变化幅度可能很大（显然的，先不说前五维是独热，后面即是经过 log 转化，数值差异也可能很大），采用一层**全连接的神经网络**和**ReLU 激活函数**，将{$O\_IO$} ∈ R^$m$×12 的向量嵌入到一组紧凑的新向量：S = {$v1$, ..., $vm$} ∈ R^$m$×d 中。（达到降维的效果，embedding 取值好基本不会损失数据的作用，同时能降维提高运算速度，这个 embedding size 是可学习的参数）

**Learned representation and pooling.** （to be honest, I feel a little confused about the term ”learned representation”, does it mean “representation learning”? 大致是：对原数据的有效信息进行筛选、提炼，最后形成一组特征值来表示原始数据，**表示学习**，用设计好的模型自动提取出有效特征。 It’s a useful and intelligent method, cause it can not only reduce the run-time cost, lower the storage usage and data complexity, but also can increase the quality of training data. 但我这里不知道他倒过来写是不是一个意思，他这词组我搜都没搜到啥论文这样叫的，但我感觉应该是一样的思路，其实他介绍的 feature extraction 也就是和表示学习一样的 idea） 作者首先提出：准确估计索引配置对查询的 rr 有

**两个挑战：**

一、 **索引交互（IIA）问题**显著增加了 IPE 的复杂性，因为索引之间的交互严重影响配置的整体效益。(然后又引用论文说明这一点)。之后作者举了一个简单的例子说明 IIA：假设一个索引配置包含三个索引 $I1$、$I2$ 和 $I3$，当单独使用 $I1$ 和 $I2$ 时，它们的效益可能微不足道。然而，当 $I1$ 和 $I2$ 一起使用时，它们可能因为索引交互使得查询成本显著减少。这是 $I1$ 和 $I2$ 之间的正向交互。当然也有**负向交互**，这是我们需要尽力避免的。综上，捕捉这些索引交互对于模型更好地学习配置中的索引如何影响查询性能至关重要。然而，现有的方法通过**表征和计算每对索引之间的交互程度**需要大量的"what-if"调用，并且将所有交互对及其交互程度进行编码并不容易。

二、 由于原始查询计划和索引配置被特征化为一组索引可优化操作 {*O_IO*}，**每个*O_IO*对整体成本减少的贡献不一定相等**。因此，很难聚合所有*O_IO* 的影响以准确估计减少率。

**解决方案：**

一、 为了将复杂的索引交互（IIA）纳入 LIB，提出了**表示层(representation layer)**的方法，该层利用**自注意力机制(self-attention mechanism)**直接建模 IIA。(简单讲了一下自注意力机制：一种显式建模集合中元素之间高阶交互的机制。)为了捕捉每个 *O_IO* 之间复杂的 IIA 信息，我们采用 **Transformer 模型**的**编码器块(encoder block)**，但不使用**位置编码(positional encoding)**以学习 {*O_IO*} 的表示。给定来自嵌入层的潜在向量集合 S = {$v1$, ..., $vm$} ∈ R^m×d ，我们执行**堆叠的自注意力编码 (self-attention encoding---SAE)** 来学习表示。每个 SAE 可以形式化如下：

$$SAE(S) = LayerNorm(H + FF(H))$$

$$H = LayerNorm(S + Multihead(S, S, S))$$

$$Multihead(S, S, S) = concat(O_1, ..., O_h)W^O \in \mathbb{R}^{m \times d}$$

$$O_j = Att(SW_j^Q, SW_j^K, SW_j^V)$$

（XXX 这里很奇怪，原作者也没说清楚，这个 W^O 应该是个 d*d 的矩阵（前面 concat 是尾接的情况），但我不知道他在干嘛，d*d 也不是正则化）

沐神:“多头注意力的输出需要经过另一个线性转换，它对应着$h$个头连结后的结果”

花书因为维度不一样，要这个线性转换。这里我没搞懂本身维度相同，为什么还要转换？？？d*d 一个矩阵相乘我感觉在这里没有任何意义啊……（求指点）

$$\mathbf{W}_o \begin{bmatrix} \mathbf{h}_1 \\ \vdots \\ \mathbf{h}_h \end{bmatrix}$$

$$\mathbf{h}_i = f(\mathbf{W}_i^{(q)}\mathbf{q}, \mathbf{W}_i^{(k)}\mathbf{k}, \mathbf{W}_i^{(v)}\mathbf{v}) \in \mathbb{R}^{p_v},$$

其可学习参数是 $\mathbf{W}_o \in \mathbb{R}^{p_o \times hp_v}$：

$\mathbf{W}_i^{(q)} \in \mathbb{R}^{p_q \times d_q}$、$\mathbf{W}_i^{(k)} \in \mathbb{R}^{p_k \times d_k}$和 $\mathbf{W}_i^{(v)} \in \mathbb{R}^{p_v \times d_v}$；

以上来源花书讲的注意力机制（我还是信这个权威一点……）

这论文这里写错了吧？我没懂啊。按这样接法，上面 concat 那里出来要么是 hd*1 要么是 d*h，右乘矩阵不可能到 m*d 的情况。

*FF* is the feed forward neural network（前向传播神经网络）

*Att* is the attention encoding（注意力编码）

编码器包含 h 组可学习参数——$\{W^Q, W^K, W^V\}_{j=1}^{h}$

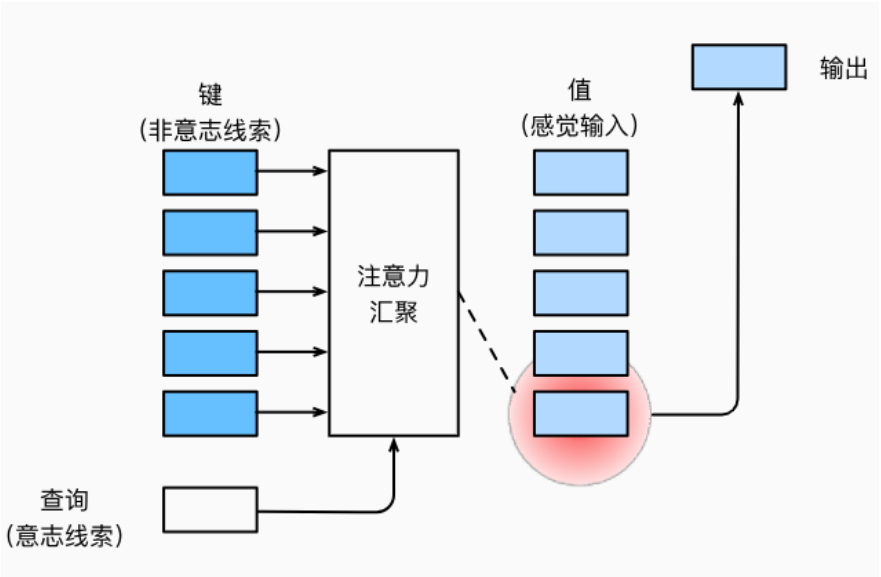
$h$ is the number of heads（注意力头的数量）

$$W^Q, W^K, W^V \in \mathbb{R}^{d \times d^M} \text{ and } d^M = d/h.$$

每个注意力头的输出最后连接（concatenate）在一起形成最终的 representation。SAE 块还使用了 residual connection 和 layer normalization 的技术。

（for techniques details: learning from “dive into deep learning” by Li Mu（沐神） ——10. 注意力机制 — 动手学深度学习 2.0.0 documentation (d2l.ai)）

**简述注意力机制**——通过注意力汇聚将查询（自主性提示）和键（非自主性提示）结合在一起，实现对值（感官输入）的选择倾向。




简单一个例子：平均汇聚层可以被视为输入的加权平均值，其中各输入的权重是一样的。 而实际上，**注意力汇聚得到的是加权平均的总和值，其中权重是在给定的查询和不同的键之间计算得出的。注意力汇聚有选择地聚合了值（感官输入）以生成最终的输出。**

**注意力汇聚**——Nadaraya-Watson kernel regression 核回归 举例：

$$f(x) = \sum_{i=1}^{n} \frac{K(x - x_i)}{\sum_{j=1}^{n} K(x - x_j)} y_i,$$

其中$x$是查询，$(xi, yi)$是键值对。 注意力汇聚是$yi$的加权平均。 将查询$x$和键$xi$之间的关系建模为 *注意力权重*（attention weight）$\alpha(x, xi)$

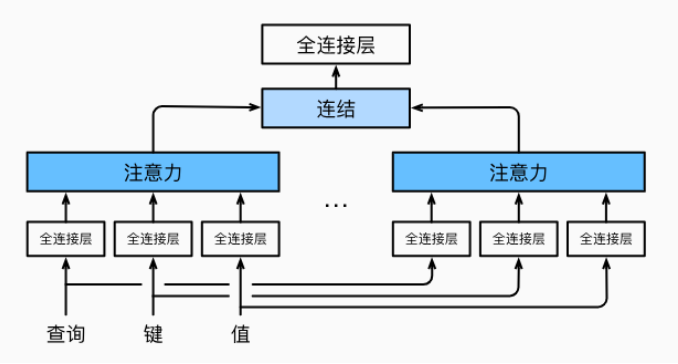
$$f(x) = \sum_{i=1}^{n} \alpha(x, x_i) y_i,$$

这个权重将被分配给每一个对应值$yi$。 对于任何查询，模型在所有键值对注意力权重都是一个有效的概率分布： 它们是非负的，并且总和为 1。（使用 softmax 函数即可）。以经典的高斯核举例（Gaussian kernel）：

$$K(u) = \frac{1}{\sqrt{2\pi}} \exp(-\frac{u^2}{2}).$$

$$\begin{aligned} f(x) &= \sum_{i=1}^{n} \alpha(x, x_i) y_i \\ &= \sum_{i=1}^{n} \frac{\exp\left(-\frac{1}{2}(x - x_i)^2\right)}{\sum_{j=1}^{n} \exp\left(-\frac{1}{2}(x - x_j)^2\right)} y_i \\ &= \sum_{i=1}^{n} \mathrm{softmax}\left(-\frac{1}{2}(x - x_i)^2\right) y_i. \end{aligned}$$

如果一个键$xi$越是接近给定的查询$x$， 那么分配给这个键对应值$yi$的注意力权重就会越大， 也就“获得了更多的注意力”。
多头注意力——当给定相同的查询、键和值的集合时， 我们希望模型可以基于相同的注意力机制学习到不同的行为， 然后将不同的行为作为知识组合起来， 捕获序列内各种范围的依赖关系




多头注意力：多个头连结然后线性变换。 给定查询$q \in R\hat{}dq$、 键$k \in R\hat{}dk$和值$v \in R\hat{}dv$，每个注意力头$hi$（$i$=1, …, $h$）的计算方法为：

$$\mathbf{h}_i = f(\mathbf{W}_i^{(q)}\mathbf{q}, \mathbf{W}_i^{(k)}\mathbf{k}, \mathbf{W}_i^{(v)}\mathbf{v}) \in \mathbb{R}^{p_v},$$

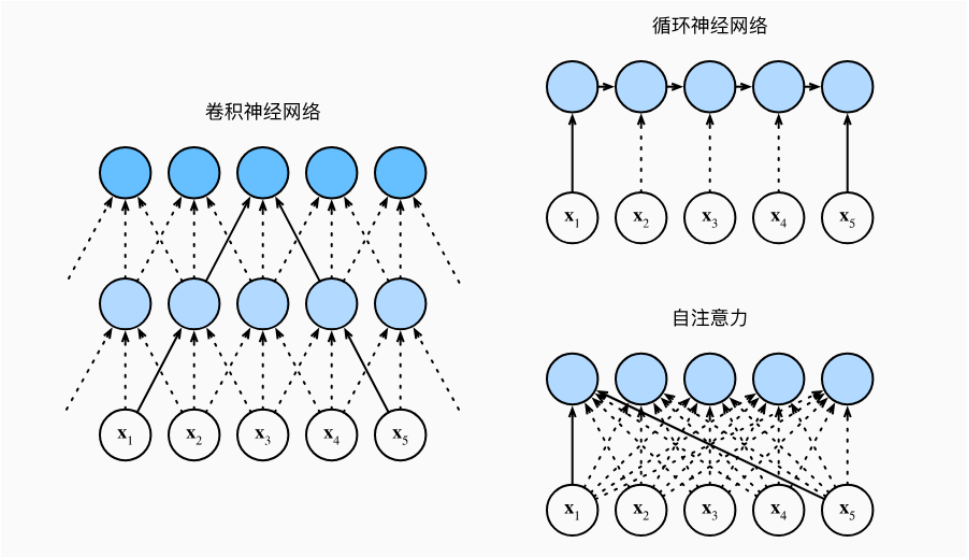
其中，可学习的参数包括 $Wi(q) \in R\hat{}pq \times dq$、 $Wi(k) \in R\hat{}pk \times dk$和 $Wi(v) \in R\hat{}pv \times dv$， 以及代表**注意力汇聚的函数$f$**。 （$f$可以是加性注意力或缩放点积注意力等）
基于这种设计，每个头都可能会关注输入的不同部分，可以表示比简单加权平均值更复杂的函数。
论文目前没有提到他们选取了哪个注意力机制，可能每个选取性能也差不太多。
**加性注意力：**

$$a(\mathbf{q}, \mathbf{k}) = \mathbf{w}_v^\top \tanh(\mathbf{W}_q\mathbf{q} + \mathbf{W}_k\mathbf{k}) \in \mathbb{R},$$

**缩放点积注意力：**

$$a(\mathbf{q}, \mathbf{k}) = \mathbf{q}^\top \mathbf{k} / \sqrt{d}.$$

应该是都行，我还猜测论文的 qkv 维度直接取的相同，功能上直观来讲，他目的是为了找到多个复杂的 IIA 的可能性，确实适合多头注意力机制，相当于给他不同的视野，肯定会比不使用的时候，可能看到一种索引交互情况就看不到别的了，这种要好上许多。
自注意力机制——同一组词元同时充当查询、键和值。 具体来说，每个查询都会关注所有的键－值对并生成一个注意力输出。 由于查询、键和值来自同一组输入，因此被称为 **自注意力（self-attention）**




Transformer 架构图——本论文用到了 encoder 部分。

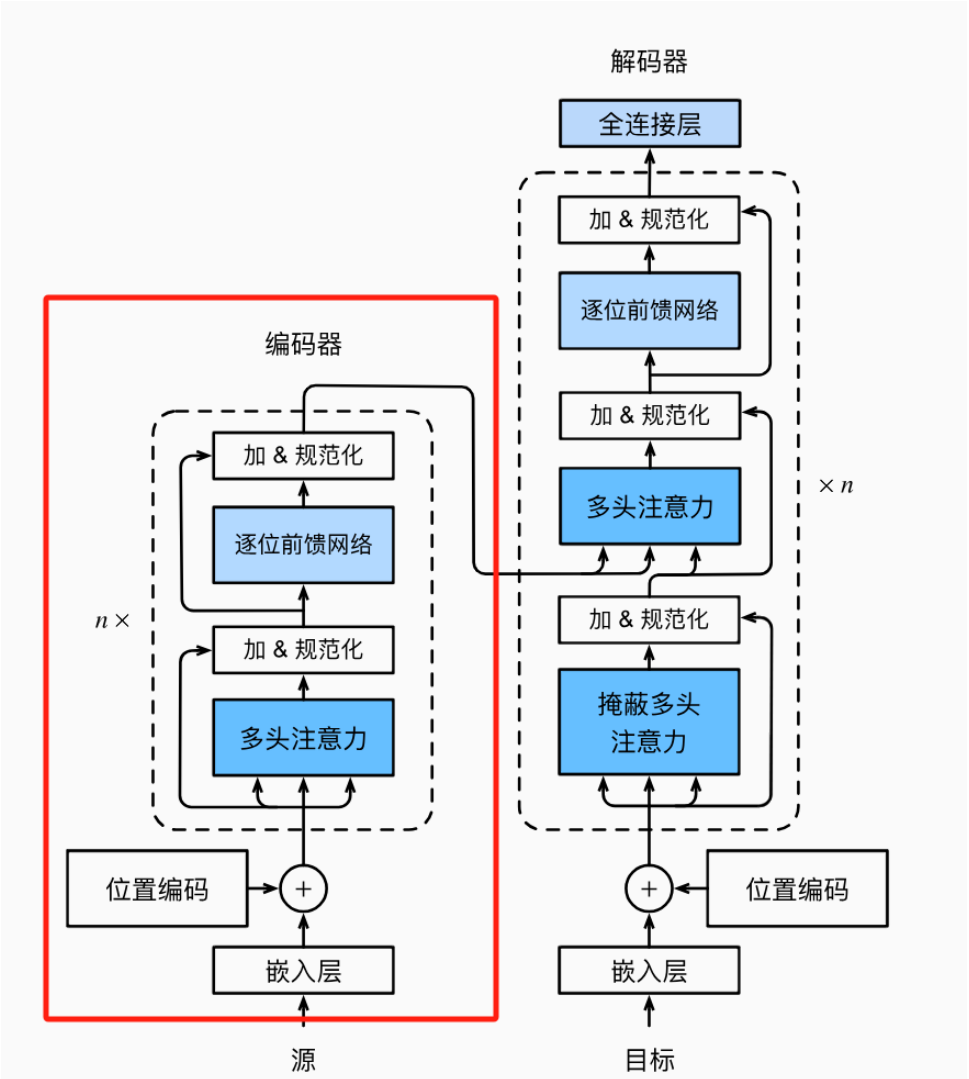



*图10.7.1* transformer架构

残差连接、层规范化，都有，除了位置编码就是拿了经典的 transformer 架构来用的。

层规范化：通过规范化输入数据，有助于减少内部协变量偏移，能加速训练过程并提高模型稳定性（也有助于避免过拟合等情况）。

二、 为解决第二个问题，应用**多头注意力池化（pooling by multihead attention (PMA)）**，来聚合 SAE 输出的一系列向量。引入一个可学习的向量 $Z \in R^{\wedge}1*d$ 作为 query，SAE(S)作为键和值，输出一个 $R \in R^{\wedge}d$ 的向量作为聚合后的最终结果。

$$R = LayerNorm(H' + FF(H'))$$

$$H' = Multihead(Z, SAE(S), SAE(S)) \in \mathbb{R}^{1\times d}$$

## 5.2 cost reduction ratio prediction model

成本降低预测模型：是一个两层全连接神经网络，使用 ReLU 作为激活器，Sigmoid 函数作为输出层。这东西就是用作概率输出，输出值限制在 0 和 1 之间且相加等于 1。对于导致查询性能下降的索引配置，我们将它们视为没有任何 benefits 的索引配置，输出概率极低，index tuner 就基本不太可能选择它们。输出层将来自 encoder 的向量表示 R（d*1 维度）作为输入并预测成本降低率 rr_hat。

## 5.3 training model

**训练数据集选取：**在 LIB 中，我们通过**执行带有实际索引配置的查询**，并使用实际成本信息作为**标签**来获取训练数据。首先，给定一个工作负载，我们采用一种 index tuning 算法（例如 DTA[7], Extend[36]）为工作负载生成一组索引候选集。然后，使用候选集的$k$组合（选取大小为 k 且元素互不相同的子集）来创建索引配置。对每个索引配置执行工作负载。同时，为了尽量减少硬件的影响，所有查询都在 warm-cache scenario 进行（即，每个查询执行 4 次，并报告后三次运行的平均成本作为执行成本）。接下来，我们计算成本减少率，并采用 LIB 中的特征提取器将查询和索引配置编码为一组向量表示。每个训练数据都存储为一个**<向量表示，成本减少率>**的元组。

**训练开销：**由于 LIB 采用 transformer 的 encoder 进行特征编码，因此 LIB 的训练时间复杂度随着每个数据中 Q_IO 的数量（即输入序列长度）呈**二次方增长**（因为使用的是自注意力机制，最后是个全连接层），并且随着训练数据的数量**线性增长**（输出多少次）。具体来说，给定一个包含 $n$ 个查询和 $m$ 个索引配置的工作负载，将会有总共$n \times m$个训练数据。对于每个数据，设 Q_IO 的数量为$k$（所有数据填充到相同数量）且每个操作的特征大小为$d$（嵌入维度）。LIB 在所有索引配置下评估整个工作负载的时间复杂度为 $O(nmdk^2)$。由于$k$（本文中最多为 40）相对整个查询计划中的所有操作数量较小，且$n$、$m$远大于$d$、$k$，作者后面 7.5 节中展示了 LIB 的训练时间是可控的。

**通用性：**为了提高 LIB 对未见过查询的通用性（这其实就是提高泛化能力，防止模型过拟合），采用了 **dropout**（每个层每次 forward 时候随机选一些神经元忽略掉，back-propagation 的时候也不对其更新权重。其实就是人为加一些噪声，有时候模型训练太专注，像人背书一样，但是没法很好的推广到未知数据的各种细节，造成过拟合。Dropout 正则化可有效避免这一点，可提高模型的泛化能力）和**早停法**（设定时间点定时跑 test set，最后回调到效果最好的一次，好处显然）进行正则化。在每个 SAE（self-attention estimation）和 PMA（pooling by multihead attention）的注意力编码层和前馈层之后，以一定概率（例如 0.2）应用 dropout。在模型训练过程中，我们在每 20 个训练周

期结束时监控模型在 test dataset 上的表现。训练结束后，我们回调到验证准确率最高的检查点。

5.4 adaption

为了使机器学习模型表现良好，需要足够数量的有效标记训练数据。然而，对于索引效益估计来说，收集大量数据用于模型训练既**昂贵又耗时**。当标注的训练数据不足时，LIB 对新数据模式的适应性是一个挑战。

Target: 增强 LIB 对新数据模式的适应性。

利用**迁移学习(transfer learning)**，就是把已学训练好的模型参数迁移到新的模型来帮助新模型训练。考虑到大部分数据或任务是存在相关性的，所以通过迁移学习我们可以将已经学到的模型参数（也可理解为模型学到的知识）通过某种方式来分享给新模型从而加快并优化模型的学习效率不用像大多数网络那样从零学习(某种方式：fine tune、IPA)。它可以减少对训练数据的需求，增加训练速度，甚至提高模型性能。

当 LIB 在具有不同数据模式的新数据集上实施时，并不用从头开始使用新收集的训练数据来训练模型，而是使用之前已经在具有大量样本的数据集上训练的模型作为预训练模型。然后我们 fine tune LIB 中的所有预训练参数以适应新数据模式。（其实 frozen 的技术，部分也行，可以更快且有效，好像在这论文之后这技术才出来？ 还有 IPA 的技术）Algorithm 1 展示了 LIB 上的迁移学习思路。

**Algorithm 1:** Transfer Learning Algorithm

**input** : Training Dataset: $(\{O_{IO}\}_k, rr_k)_{k=1}^{K}$,
Pre-trained Parameter Set: $\Theta_0$
**output**: Parameter Set for New Data Schema: $\Theta_1$

$\Theta_1 \leftarrow \Theta_0$;
**while** *training is true* **do**
  $\mathcal{B} \leftarrow$ A Random minibatch of data;
  $\widehat{rr}_q \leftarrow \mathrm{LIB}(\{O_{IO}\}_q|\Theta_1) \quad \forall\{O_{IO}\}_q \in \mathcal{B},\ q = 1, ..., |\mathcal{B}|$;
  $\mathcal{L} \leftarrow \sum(rr_q - \widehat{rr}_q)^2/|\mathcal{B}| \quad \forall rr_q \in \mathcal{B}, q = 1, ..., |\mathcal{B}|$;
  **for** $\theta \in \Theta_1$ **do**
    $g_\theta \leftarrow \nabla_\theta \mathcal{L}(\Theta_1)$;
    $\theta \leftarrow \theta + \Gamma(g_\theta)$;
  **end**
**end**
**return** $\Theta_1$;

上表是一种简单的迁移学习过程。作者说由于所有特征属性都是模式无关的，这使得轻量级的迁移学习技术能够满足我们的需求。

每次循环以当前的Θ1 使用 LIB 计算 rr_hat，与训练集真实 rr 做差的平方累加除以 minibatch 的大小赋给函数 L。

内层循环对Θ1 中的每个值 θ 做梯度下降，偏导 θ 计算下降最快的方向（直观来讲，目的是使 L 最小化，如何最小？→每个 θ 如何改变才能使它最小→梯度下降：计算 θ 如何改变使得 L 下降最快） 第二行的倒 L(g θ)，倒 L 应该就是个类似学习率的可学习参数，表述一个 step 多大。

6. Integration with Index Tuner

为了推荐一个索引配置以最小化给定 workload 的总执行成本，index tuner 需要执行索引效益估计以评估每个索引配置对 workload 的益处，从而搜索出理想的推荐配置。由于 LIB 输出的每个查询的索引效益是通过查询的初始执行成本归一化的，并且不同查询的执行成本不同，索引调优器不能直接使用成本减少率在整个工作负载的查询上进行索引调优。为了解决这个问题，我们将 LIB 输出的归一化比率乘以初始成本，将比率转换为**绝对成本减少**。转换公式如下：

$$\mathrm{cr}_{i,j} = \mathrm{rr}_{i,j} \times \mathrm{Cost}(q_j|\emptyset)$$

这东西他就是说，**我们论文 LIB 算的使 rr，而一般的 index tuner 还是拿 cr 做选择的，不兼容怎么办呢？乘个 cost 就转过去了。**

并行特性。LIB 的另一个理想特性是它容易并行化。由于使用 LIB 进行索引性能估计仅涉及矩阵乘法，因此可以用 GPU TPU 这些东西加速。

7. Experiments （总之结果很好，读者没啥必要看了，我论文要汇报，这里就翻译一下算了，本论文到此正文就完结了）

**实验参数设置：**

一、为衡量预测准确性，（引用了论文）使用 Q-Error 方法：

$$Q\_Error = max(\frac{rr_{actual}}{\widehat{rr}_{predicted}}, \frac{\widehat{rr}_{predicted}}{rr_{actual}})$$

二、工作负载(workload)：为了调查 LIB 在不同工作负载上的性能，我们使用了四个基准：TPC-H-10GB，TPC-DS-10GB，TPC-DS-50GB 和 IMDB-JOB。它们在**潜在索引的数量、查询数量和数据库大小方面具有不同的特征**。对于 TPC-H，我们使用了 10 的规模因子，而对于 TPC-DS，我们使用了 10 和 50 的规模因子。TPC 基准测试的查询是基于预定义模板使用查询生成工具生成的。更具体地，对于 TPC-H，我们使用 14 个模板生成 700 个查询。我们排除了一些执行成本比其他模板高出几个数量级的模板。这是因为它们主导了工作负载的成本，使得索引选择问题变得不那么复杂。来自被排除模板的索引加速查询总是优于其他查询的索引。对于 TPC-DS-10GB，出于相同的原因，我们使用 99 个模板中的 78 个生成 390 个查询，而对于 TPC-DS-50GB，我们使用 66 个模板生成 198 个查询。互联网电影数据库（IMDB）是现实世界的数据。我们使用 JOB Benchmark3，它包含 113 个查询。

**Table 1: Statistics about the workloads**

| Benchmark | DB size (GB) | # Tables | # Queries | # Cases |
|---|---|---|---|---|
| TPC-H | 10 | 8 | 700 | 3771 |
| TPC-DS-10 | 10 | 24 | 390 | 27584 |
| TPC-DS-50 | 50 | 24 | 198 | 4518 |
| IMDB-JOB | 9.3 | 20 | 113 | 2879 |

三、数据生成：使用第 5.3 节讨论的方法，我们采用 Microsoft SQL Server 的数据库

引擎调优顾问（DTA）的 Anytime 算法为上述工作负载生成索引配置候选。生成数据的详细信息如表 1 所示，其中**# Case 表示生成的实例数量**。**每个实例代表一个带有索引配置的查询**，形式为一个**<向量表示，成本减少率>**的元组（前面提到过的 training dataset）。TPC-H 基准测试的表数量相对较少，每个查询的潜在索引数量也有限。因此，TPC-H 的实例数量较少。对于 TPC-DS-50，由于每个查询的实际执行时间相比 TPC-DS-10 要大得多，因此采用的查询数量较少。将数据集划分为**训练集和测试集**，具体如下：(1) 索引配置：我们将所有数据点的联合随机分成五个不相交的集合。然后，我们进行 5 折交叉验证（**k 折交叉验证**：模型在验证数据中的评估常用的是交叉验证，又称循环验证。它将原始数据分成 K 组(K-Fold)，将每个子集数据分别做一次验证集，其余的 K-1 组子集数据作为训练集，这样会得到 K 个模型。这 K 个模型分别在验证集中评估结果，最后的误差 MSE(Mean Squared Error)加和平均就得到交叉验证误差。交叉验证有效利用了有限的数据，并且评估结果能够尽可能接近模型在测试集上的表现，可以做为模型优化的指标使用），取其中 4 个作为训练集，剩下的一个作为测试集。在这种训练-测试拆分中，测试集中的索引配置与训练集中的不同，以便在训练过程中未出现的新索引配置上进行推理。(2) 查询：数据集根据查询信息分为 5 个不相交的集合。我们首先将查询分成 5 个不相交的集合，然后**将每个查询集的相应实例放入一个集合中**。这**模拟了 LIB 用于未见查询的索引效益估计的设置**。通过使用这些拆分方法，我们模拟了训练集和测试集之间的两种分布变化类型。

四、神经网络设置：如前所述，数据向量大小为 12。LIB 中使用的嵌入大小为$d = 32$。编码器采用 6 层 8 头自注意力模块，dropout 率为 0.2。编码器中前馈线性层的隐藏维度设计为 128。输出层的隐藏维度和输出维度分别为 64 和 1。模型使用 Adam optimizer 进行训练，初始学习率为 0.001，训练 150 个周期。

五、具体方法：我们将 LIB 与 **PostgreSQL** 13.3 数据库的成本估计器进行比较，因为现有的 ISP 算法和 index tuner 是基于优化器的成本估计。我们还与一种最新的基于**分类器**的解决方案"AI-Meet-AI"进行比较。"AI-Meet-AI"使用分类算法在一对计划之间找到更好的计划，但它不支持 rr 预测。为了扩展"AI-Meet-AI"以估计成本减少率，我们将分类器组件替换为一个具有隐藏维度 128 的两层全连接神经网络（超参数通过标准交叉验证进行调优）和 Sigmoid 输出层，并将其称为**"AIMAI-R"**。此外，我们还研究了一种基于机器学习的成本估计器"end2end" 在 IPE 任务中的表现。我们按照中建议的超参数进行操作，并移除字符串嵌入部分，因为字符串谓词很少与二级索引相关。我们使用成本和基数的损失来训练模型，直到模型收敛。我们还将 LIB 与**两种操作符级成本模型**（即计划结构模型（简称 **Plan-Strut**）和 **MART**）进行比较。我们按照论文中建议的超参数来训练它们的模型。对于 MART，我们采用默认模型而不使用缩放函数，因为测试数据中非零*out_ratio*（在中定义的模型选择指标）不到 1%。

六、训练环境：所有评估均在具有 Intel i9-10900X CPU、64GB RAM 和 GeForce RTX 3080 的机器上进行。

**预测准确性实验：**我们评估了 LIB 在跨查询和索引配置估计成本减少比率(rr)任务中的准确性。与 PostgreSQL-13 优化器的成本估计相比，我们观察到**错误率减少了最多 91.2%**。此外，我们还观察到 LIB 不仅在平均上减少了错误，而且**在所有**误差分位数**上都减少了误差**。

**Table 2: Prediction accuracy (Q-Error) with different splitting methods**

| | | Splitting by Configuration | | | | | Splitting by Query | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Dataset | | PostgreSQL | AIMAI-R | Plan-Strut | MART | LIB | PostgreSQL | AIMAI-R | Plan-Strut | MART | LIB |
| TPC-DS-10 | Mean | 4.380 | 4.966 | 17.268 | 4.833 | **1.984** | 4.383 | 4.669 | 16.935 | 4.955 | **2.205** |
| | 90th | 10.488 | 4.788 | 53.661 | 13.696 | **3.260** | 10.485 | 4.907 | 44.762 | 14.039 | **3.935** |
| | 95th | 15.428 | 7.252 | 87.181 | 19.257 | **5.170** | 15.587 | 7.590 | 87.460 | 19.222 | **6.340** |
| TPC-H | Mean | 13.171 | 3.275 | 35.545 | 1.933 | **1.159** | 13.788 | 3.494 | 38.972 | 1.929 | **1.148** |
| | 90th | 52.581 | 5.797 | 99.020 | 2.505 | **1.324** | 55.543 | 6.865 | 99.020 | 2.560 | **1.310** |
| | 95th | 77.445 | 14.521 | 99.020 | 5.057 | **1.575** | 77.750 | 15.970 | 99.020 | 4.969 | **1.562** |
| TPC-DS-50 | Mean | 6.603 | 7.499 | 33.195 | 15.263 | **2.575** | 6.439 | 4.476 | 33.479 | 15.219 | **2.719** |
| | 90th | 16.623 | 8.524 | 93.195 | 52.150 | **4.388** | 15.374 | 8.309 | 89.063 | 49.917 | **5.101** |
| | 95th | 38.507 | 12.841 | 99.020 | 61.235 | **8.283** | 33.775 | 14.512 | 97.827 | 59.853 | **9.343** |
| IMDB-JOB | Mean | 5.654 | 7.478 | 9.939 | 7.716 | **3.655** | 5.518 | 8.279 | 11.226 | 9.874 | **4.363** |
| | 90th | 12.690 | 9.226 | 28.371 | 17.085 | **6.904** | 14.823 | 8.974 | 34.421 | 30.719 | **8.031** |
| | 95th | 20.699 | 12.596 | 40.206 | 36.995 | **9.513** | 24.281 | **10.396** | 49.345 | 46.546 | 11.120 |

中位数误差、百分之九十分数位误差、百分之九十五分数位误差 都减小

**未见配置的推理**（这是指模型在训练过程中**在未见过的索引配置上的性能表现**。换句话说，当给定一组**新的索引配置**时，模型是否能够准确地估算查询的成本缩减比率。测试这种情况是为了验证模型在面对新的、未见过的索引配置时能否依然保持高精度的预测能力。）、 **未见查询的推理**（这是指模型在训练过程中**未见过的查询**上的性能表现。也就是说，当给模型一组新的查询时，模型能否准确地估算这些查询在不同索引配置下的成本缩减比率。测试这种情况是为了验证模型能否泛化到新类型的查询，而不仅仅是它在训练过程中见过的查询。）

每种方法在估计查询上索引配置的成本减少率方面的准确性如表 2 所示。我们对每个集合进行了**五次实验，并报告了平均结果**。

表 2 中的结果显示，提出的 LIB 模型在所有方法中表现最佳。对于未见配置的推理，LIB 在平均 Q-Error 方面分别比 PostgreSQL、AIMAI-R、Plan-Strut 和 MART 提高了高达 91.2%、65.7%、96.7%和 83.1%。这表明，使用我们提出的基于集合的特征化方法和基于注意力的模型结构的基于 ML 的模型能够学习出一个准确的映射函数，将索引相关信息和成本减少率联系起来。我们还调查了所有方法在尾部（**第 90 和 95 百分位数**）的表现。LIB 在 TPC 基准测试中的所有误差分位数上表现优于其他方法。在 IMDB-JOB 上，虽然 LIB 在 95 百分位上不及 AIMAI-R，但在 90 百分位和平均误差上，LIB 分别比 AIMAI-R 提高了 10.5%和 47.3%。与 PostgreSQL 相比，LIB 在 95 百分位上可以减少高达 98.0%的错误。这进一步证明了 LIB 能够准确估计成本减少率。在 TPC-DS-50 上，PostgreSQL 的平均误差为 6.60，比 TPC-DS-10 上的 4.38 大 50%。这表明 PostgreSQL 成本估计器的性能受到数据库大小的影响。然而，LIB 的平均误差仅从 1.98 增加到 2.57，且在 TPC-DS-50 的 95 百分位误差更好。这表明 LIB 对数据库大小的变化具有鲁棒性。**这种观察的一个解释是，LIB 利用数据库统计作为特征属性之一，使得模型能够捕捉数据库大小的变化，因此性能更稳定。对于未见查询的推理（表 2 中的最后 5 列）**，LIB 的表现与**未见配置的推理**结果类似。平均误差的最大增加仅为 0.708，出现在 IMDB-JOB 上。这表明 LIB 能够泛化到未见的查询，并且对查询工作负载的变化具有鲁棒性。

**索引推荐改进：**作者将 LIB 集成到一个 index tuner 中，并量化了工作负载执行成本的端到端改进，展示了使用 LIB 增强 index tuner **可以提高索引推荐的效果**。

我们现在评估在将 LIB 集成到索引调优器中时，LIB 对端到端推荐质量的影响，**以查询执行成本为指标**。将 LIB 集成到使用 Anytime 算法的索引调优器中。我们对 75 个 TPC-DS-10GB 查询、66 个 TPC-DS-50GB 查询、70 个 TPC-H 查询和 22 个 IMDB-JOB 查询进行索引调优，**这些查询均未包含在 LIB 的训练数据中（即 LIB 在未见查询上进行索引调优）**。然后，我们评估使用相同算法但采用基于“what-if”的索引效益量化方法的调优器的

性能，这些方法分别通过 PostgreSQL13 的成本估算和 MART 的预测来区分不同的候选者。此外，我们将 LIB 与基于“what-if”的分类器“AI-Meet-AI”进行比较。我们执行所有候选项以找到最佳推荐。我们的目标是为工作负载中的一组查询推荐最佳索引。我们通过从每个基准的训练数据中随机抽取未见查询来构建 40 个查询工作负载（每个 TPC 基准包含 10 个查询，IMDB-JOB 包含 8 个查询）。然后，我们调用索引调优器，使用不同的索引性能估算方法为每个工作负载找到最佳索引。索引推荐的结果如表 3（TPC-DS）和表 4（TPC-H 和 IMDB-JOB）所示。我们报告了总的工作负载执行成本节省、平均工作负载改进以及改进分布（即每个改进范围内的工作负载数量）。

**Table 3: TPCDS-10GB & TPCDS-50GB Workload level tuning** (* denote methods based on "what-if" calls)

| Methods | Total Cost Saving (ms) | | Average Workload Improvement (%) | | Distribution of Improvement (# workloads) | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | Regression | | 0%-5% | | 5%-20% | | 20%-50% | | >50% | | Total |
| | 10GB | 50GB | 10GB | 50GB | 10GB | 50GB | 10GB | 50GB | 10GB | 50GB | 10GB | 50GB | 10GB | 50GB | |
| PostgreSQL* | 161,066 | 302,391 | 4.9 | 3.9 | 1 | 8 | 25 | 23 | 12 | 7 | 2 | 2 | 0 | 0 | 40 |
| MART* | 125,331 | 1,115,036 | 5.2 | 12.4 | 0 | 0 | 23 | 16 | 16 | 14 | 1 | 9 | 0 | 1 | 40 |
| AI Meets AI* | 1,488,759 | 425,664 | 8.1 | 5.0 | 0 | 0 | 27 | 31 | 11 | 6 | 0 | 3 | 2 | 0 | 40 |
| LIB | 3,071,577 | 1,175,872 | 15.5 | 12.5 | 0 | 0 | 24 | 14 | 8 | 16 | 2 | 9 | 6 | 1 | 40 |
| Optimal | 4,445,404 | 1,675,181 | 21.4 | 17.6 | 0 | 0 | 10 | 11 | 19 | 12 | 4 | 15 | 7 | 2 | 40 |

**Table 4: TPCH-10GB & IMDB-JOB Workload level tuning** (* denote methods based on "what-if" calls)

| Methods | Total Cost Saving (ms) | | Average Workload Improvement (%) | | Distribution of Improvement (# workloads) | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | Regression | | 0%-5% | | 5%-20% | | 20%-50% | | >50% | | Total |
| | TPCH | IMDB | TPCH | IMDB | TPCH | IMDB | TPCH | IMDB | TPCH | IMDB | TPCH | IMDB | TPCH | IMDB | |
| PostgreSQL* | 108,825 | 39,267 | 4.2 | 3.4 | 16 | 9 | 9 | 16 | 14 | 15 | 1 | 0 | 0 | 0 | 40 |
| MART* | 270,167 | 35,727 | 9.8 | 3.1 | 0 | 8 | 13 | 19 | 21 | 13 | 6 | 0 | 0 | 0 | 40 |
| AI Meets AI* | 176,930 | 19,578 | 6.5 | 1.6 | 0 | 12 | 26 | 23 | 10 | 5 | 4 | 0 | 0 | 0 | 40 |
| LIB | 277,209 | 42,321 | 10.0 | 3.6 | 0 | 8 | 12 | 15 | 22 | 17 | 6 | 0 | 0 | 0 | 40 |
| Optimal | 675,890 | 65,315 | 21.7 | 5.7 | 0 | 0 | 2 | 17 | 15 | 23 | 23 | 0 | 0 | 0 | 40 |

总成本节省 平均使得 workload 性能提升（百分比）

对于 TPC-DS-10GB，基于 LIB 的索引推荐在**40 个工作负载**上的**总成本节省**为 3,072 秒，是“AI-Meet-AI”的两倍多。基于 LIB 的索引推荐平均改进一个工作负载 15%，优于基于 PostgreSQL13、MART 和“AI-Meet-AI”的调优器推荐的索引。LIB 基于索引推荐比“AI-Meet-AI”有更大成本节省的一个原因是 LIB 能够更准确地量化每个索引配置对工作负载的效益，而“AI-Meet-AI”只能对每个单独查询的较优索引配置进行分类，并且“AI-Meet-AI”主要设计用于防止查询回归。从改进分布来看，我们观察到一个最优解可以**对 7 个工作负载实现超过 50%的改进**。相比之下，“AI-Meet-AI”只能对其中的 2 个实现超过 50%的改进，而 LIB 可以对其中的 6 个实现超过 50%的改进。在 TPC-DS-50GB 上，观察到类似的趋势，LIB 的总成本节省和平均工作负载改进（1,176 秒和 12.5%）是“AI-Meet-AI”（426 秒和 5.0%）和 PostgreSQL13（302 秒和 3.9%）的两倍多。当数据库大小增加时，PostgreSQL 会产生更多查询回归案例。然而，“AI-Meet-AI”和 LIB 在防止查询回归方面都被证明是有效的。对于 MART，TPC-DS-50GB 上的索引推荐质量优于 TPC-DS-10GB 上的推荐质量。一个解释是，当数据库大小增加时，不同索引配置之间的执行成本差异更大。尽管 MART 在 TPC-DS-50GB 上的预测准确性较低，但它能够区分各种索引配置下的查询性能。

对于 TPC-H，如表 4 所示，PostgreSQL 生成了更多**查询回归案例**（16 个）（当进行某些更改（如索引的添加或删除、查询优化器的调整等）后，查询的执行性能变得比之前更差的现象。这种性能下降可能表现为查询执行时间增加、资源消耗增多等。）然而，“AI-Meet-AI”和 LIB 在防止这些案例方面是有效的。对于 IMDB-JOB，由于复杂的属性相关性和数据分布偏斜，更难准确估算基数。如表 4 所示，所有方法都导致了更多的查询回归案例。然而，LIB 仍能优于其他方法并实现最大的成本节省。

**效率：**我们评估了在线索引效益预测的**运行时间**，以及在不同规模的工作负载上集成了

LIB 的两个索引选择算法的端到端索引调优时间。我们观察到 LIB 比基于"what-if"的方法更高效。使用 LIB 可以**减少最多 89%的端到端索引调优运行时间**。

我们报告了在索引调优期间，LIB、AIMAI-R、MART 和 PostgreSQL 在不同数量的 IPEs 下的在线推理时间。这里，每个 IPE 代表在工作负载中的一个查询对应的一个索引配置的成本缩减比率估计。对于 LIB、MART 和 AIMAI-R，推理时间包括数据编码和模型评估所需的时间。对于 PostgreSQL，我们测量其生成执行计划的“假设”调用所需的时间。我们将 IPE 的数量从 1 变化到 25,000，即索引选择过程中处理的估计数量，结果如图 6 所示。由于 **LIB 可以并行处理一批预测，因此其效率更高**。由于 AIMAI-R 和 MART 依赖于“what-if”调用生成每个索引配置的查询计划作为输入，它们的效率受限于“假设”调用，这进一步证明“假设”调用是 ISP 算法的瓶颈。对于 25,000 个 IPEs，LIB 相比其他方法可以减少高达 97.8%的运行时间。

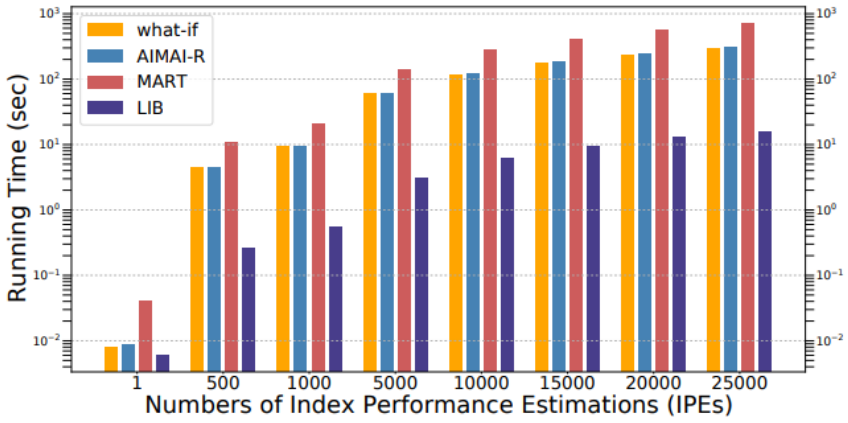
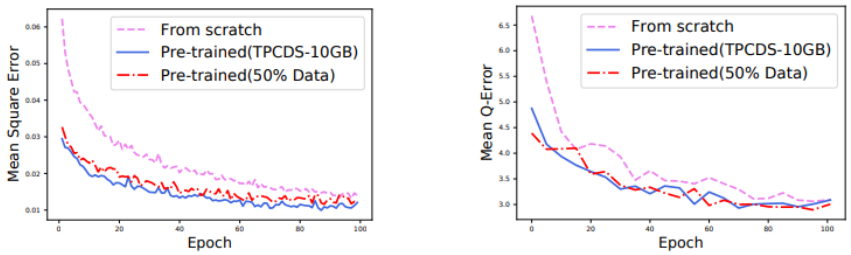



**Figure 6: Online Inference Running Time**

**适应不同数据模式：**我们评估了 LIB 对不同数据模式的适应性。我们发现 LIB 在提高模型学习性能方面是**有效**的。

我们评估迁移学习技术的有效性。我们在三种不同条件下训练 LIB 模型：(1) 仅使用新数据库的数据从头开始训练模型；(2) 预训练模型使用 TPC-DS-10GB 数据，然后使用新数据库的数据对整个模型进行微调；(3) 同条件 2，但仅使用新数据库的一半训练数据对整个模型进行微调。第三种条件设计用于评估当训练数据数量有限时模型的性能。对于所有的 LIB 预训练，我们训练模型 120 个周期。




**(a) TPCDS-50 Training** **(b) TPCDS-50 Testing**

**Figure 7: Learning curves of LIB**

图 7 显示了 LIB 在三种条件下的学习曲线。**使用迁移学习的 LIB 比从头开始训练的模型收敛更快。**此外，**使用 50%训练数据的模型表现出与使用全部训练数据的模型类似的学习行为**。这表明，利用迁移学习技术可以大大减少对训练数据的需求并缩短训练时间。

8. 相关工作和总结（略）