Representing Schema Structure with Graph Neural Networks for Text-to-SQL Parsing

王天乐

摘要

将自然语言问题翻译成 SQL 查询以回答数据库中的问题时，现代语义解析模型很难泛化到未知的数据库模式中。泛化挑战在于如何以语义解析器可访问的方式对数据库关系进行编码，以及在给定查询中对数据库列及其提及项之间的关系进行建模。现在的研究在很大程度上忽略了数据库(DB) 模式的结构，这要么是因为 DB 非常简单，要么是因为它在训练和测试时都被观察到了。在最近发布的 text-to-sql 数据集 SPIDER 中，在测试时会给出新的和复杂的数据库，因此数据库模式的结构可以为预测的 SQL 查询提供信息。在本文中，作者提出了一个编码器-解码器语义解析器，其中 DB 模式的结构使用图神经网络进行编码，此表示稍后在编码和解码时使用。

关键词：embedding；text-to-sql；GNN

1 引言

SQL 语言是当前使用的关系数据库的主要查询语言。自然语言到 SQL 的映射可视为语义解析问题 (Andreas, Vlachos et al., 2013)。语义解析是长期存在且在自然语言处理（NLP）中被广泛研究的问题。因此，它引起了学术界和业界的广泛关注，特别是将自然语言转换为 SQL 查询。当今时代，从金融、电子商务到医疗领域，大量数据都存储在关系型数据库中。因此，使用自然语言查询数据库有许多应用场景。如自助式仪表盘和动态分析，可以通过自然语言来获取与业务最相关的信息。与将自然语言转成 sql 相关的任务还有代码生成和模式生成（code generation and schema generation）。这些任务可以总结为，将自然语言翻译成完整应用程序的一般任务。

如图-1 所示 (Sun, Tang et al., 2018)，用户提出自然语言问题：“what 's the total number of songs originally performed by anna nalick?”，输入到 Text to sql 解析器中，解析器输出 SQL 语句“h = h ”，执行模块再在数据库中执行 sql，返回执行结果：1。

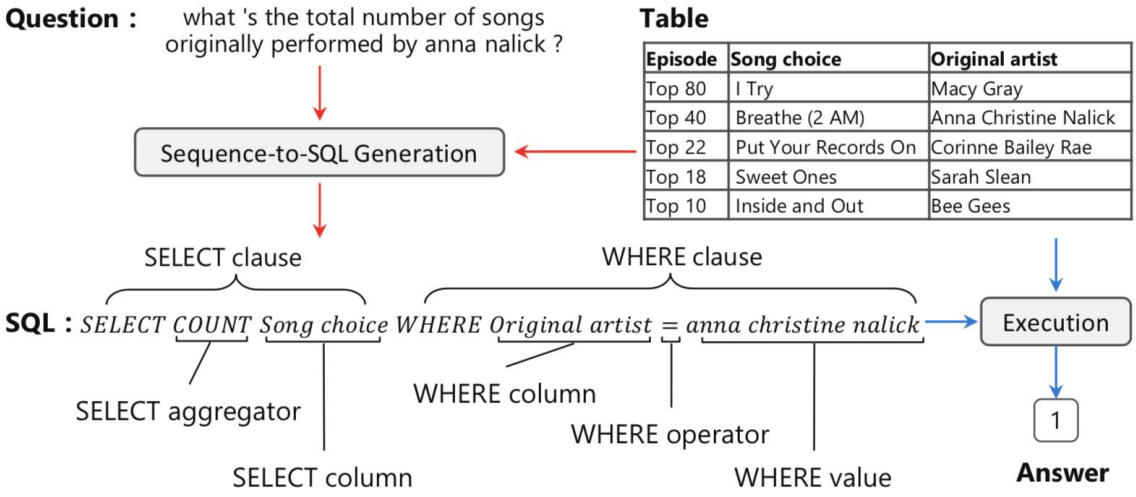

图 1: Text-To-SQL 问题示例

现阶段将文本解析为 SQL 的工作要么涉及只包含一个表的简单数据库，要么有一个在训练和测试时都被观察到的数据库。因此，模式结构建模很少受到关注。最近，Yu 等人发布了 SPIDER，这是一个文本到 SQL 的数据集，在测试阶段时，针对训练时未出现的复杂数据库执行问题。在这种零样本设置中，数据库模式结构的信息表示很重要。考虑图 2 中的问题: 尽管它们的语言结构相似，但在第一个查询中，必须进行“join”操作，因为信息分布在三个表中，而在另一个查询中，不需要“join”。

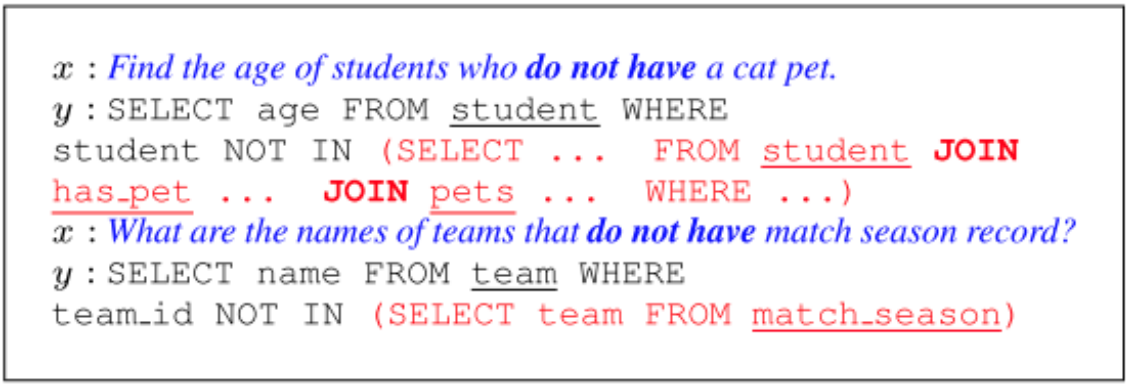

图 2: 来自 SPIDER 的示例显示了相似的问题如何有不同的 SQL 查询

在这项工作中，作者提出了一个使用模式结构的语义解析器。我们将数据库模式的结构表示为图，并使用图神经网络 (gnn) 为每个节点提供全局表示。作者将模式表示合并到编码器-解码器解析器中，该解析器旨在将问题解析为针对未见过的半结构化表的查询。在编码时，作者用与它相关的子图的表示来丰富每个问题词，在解码时，作者从模式中发出符号，这些符号通过图与先前解码的符号相关。

2 相关工作

从任务涉及的数据来看，用户输入为自然语言问题，可利用的数据有数据库、SQL 关键词，输出为 SQL 查询语句，本质上是一个符合语法、有逻辑结构的序列。所以，sql 的构成来自三部分：

- 自然语言问题：结合数据库，一般可以直接抽取出 sql 中需要的表名，列名，条件表达式，条件值；
- 数据库：结合自然语言问题，一般用于辅助识别 sql 中需要的表名，列名，条件表达式，条件值；
- SQL 关键词：作为 sql 查询语句的候选 token，用于生成 sql。

所有工作都在完成基于上述三部分数据来生成一个可在给定数据库中执行以获取正确结果的 sql。

2.1 基于模板和规则

早期研究多基于规则的方法（A.-M. Popescu et al., 2003）。因为 SQL 查询语句本身是一种有很强范式的编程语言，既然是语言就有一定的语法结构。根据 SQL 执行的复杂程度，可以将其分为简单 SQL 和复杂 SQL。简单 SQL 只涉及少数的 SQL 关键字和组成部分，典型特征是可以拆分为“SELECT”和“WHERE”两个片段。简单 SQL 语句都可以抽象成如图-3 的模板：

图 3: 简单 SQL 模板

基于该模板，针对用户输入的问题，可以设计一些匹配问题的正则表达式来抽取 SQL 模板中各个片段内容。如一种简单查询的模板设计如下：

- 问题：起购金额低于 1000 的理财有哪些？
- 表达式：([COLUMN]+?)([低于]+?) (d+?) 的 ([COLUMN]+?) 有哪些？
- SQL：SELECT 理财 FROM 理财产品 WHERE 起购金额 < 1000;

同理，针对其他更多的查询 sql，都可以设计配套的表达式去匹配用户的问题和抽取 sql 相应的成分。这是基于正则表达式来识别 sql 类型和抽取成分。然而，正则的弊端是需要列举各种表达范式，工作量大。其次，如果表达范式写的太严，容易导致漏匹配，相反，写的太宽泛易导致误匹配。为了解决上述问题，可以使用有监督的方法来替换正则，如使用神经网络对问题进行编码，识别 sql 类型可以采用分类的方法；抽取 sql 字段值可以采用序列标注的方法。该方法类似对话系统中的意图识别和槽位填充，往往一个模型可以同时实现分类和填槽。

本方法可快速实现一套覆盖主要问题的 text2sql 系统，可解释性强，对定义中的 sql 准确率高，定义外的 sql 可以拒绝识别。但由于每类 sql 都需要先定义，导致能覆盖的 sql 类型较少，如 select 有 1-n 列可以组合，where 条件组有 1-n 组可以组合，会导致 sql 类型数据激增，这时，该方法再覆盖就工作量巨大。为了生成的 sql 更灵活，样式更多，很多工作已经切换到端到端的神经网络模型。

2.2 基于 Seq2Seq 框架

Linking schema items 为了处理未见过的数据库模式项，作者提出学习单词 x_i 和具有类型 τ 的模式项 v 之间的相似得分 $s_{link}(v, x_i)s_{link}(v, x_i)$ 。该分数基于学习的词向量和交叉的特征。链接分数用于计算

$$p_{\text{link}}\left(v \mid x_i\right)=\frac{\exp\left(s_{\text{link}}\left(v, x_i\right)\right)}{\sum_{v' \in \mathcal{V}_\tau \cup\{\varnothing\}} \exp\left(s_{\text{link}}\left(v', x_i\right)\right)}$$

其中 V_τ 是所有类型为 τ 的模式项，$s_{link}(\varnothing, \cdot) = 0$ 表示不链接到任何模式项的单词。函数 $p_{link(\cdot)}$ 和 $s_{link(\cdot)}$ 将用于解码未见过的模式项。

Encoder 双向 LSTM 为每个问题词 x_i 提供上下文表示 h_i。重要的是，在时间步长 i 的编码器输入是 $[w_{x_i}; l_i]$：$l_i = \sum_\tau \sum_{v \in \mathcal{V}_\tau} p_{\text{link}}\left(v \mid x_i\right) \cdot r_v$ 和单词嵌入 x_i 的连接，其中 r_v 是模式项 v 的学习嵌入，其基于 v 的类型及它的模式邻居。因此，$p_{\text{link}}\left(v \mid x_i\right)$ 将每个单词 x_i 都增加了它应该链接到的模式项的信息。

Decoder 我们使用基于语法的 LSTM 解码器来关注输入问题（图 2）。在每个解码步骤中，使用一种语法规则扩展 τ 类型的非终结符。规则要么独立于模式并生成非终结符或 SQL 关键字，要么特定于模式并生成模式项。在每个解码步骤 j，解码 LSTM 将向量 g_i 作为输入，这是在前一步中解码的语法规则的嵌入，并输出向量 o_j。如果此规则是架构依赖性的，g_i 是学到的全局嵌入。如果是架构特定的，即，生成了架构项 v，那么 g_i 是其类型的学习嵌入 $\tau(v)$。输入单词上的注意力分布 a_j 以标准方式计算，其中每个单词的注意力分数为 $h_i^T o_j$。然后用于计算输入 $c_j = \sum_i a_j h_j$ 的加权平均值。现在通过以下方式计算语法规则的分布：

$$s_j^{\text{glob}} = \text{FF}\left([o_j; c_j]\right) \quad \in \mathbb{R}^{G_{\text{legal}}},$$

$$s_j^{\text{loc}} = S_{\text{link}}\, a_j \in \mathbb{R}^{\mathcal{V}_{\text{legal}}},$$

$$p_j = \text{softmax}\left(\left[s_j^{\text{glob}}; s_j^{\text{loc}}\right]\right)$$

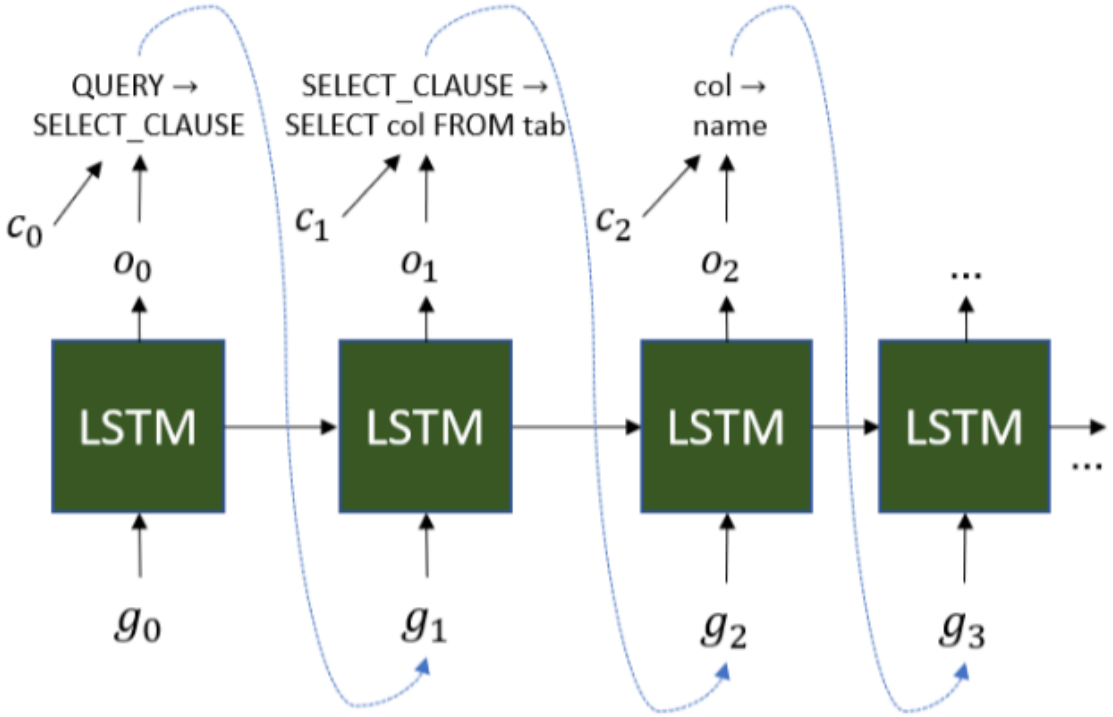

图 4: 解码器

其中 G_{legal}□$\mathcal{V}_{legal}$ 是可以在时间步长 j 分别为模式独立规则和模式特定规则选择的合法规则数（根据语法）。分数 s_j^{glob} 是用前馈网络计算的，分数 s_j^{loc} 是通过乘以矩阵 $S_{\text{link}} \in \mathbb{R}^{\mathcal{V}_{\text{legal}} \times |x|}$ 为所有合法模式项计算的，其中包含相关的链接分数 $s_{link}(v, x_i)$，注意力向量 a_j。因此，解码未见过的模式项是通过首先关注链接到模式项的问题词来完成的。

3 本文方法

3.1 本文方法概述

作者基于 Krishnamurthy 等人的解析器上，使用图神经网络对 DB 模式的结构进行编码。考虑一个有两列的表（如图 5），其中每一列都是另外两个表的外键。这样的表通常用于描述两个其他表之间的多对多关系，这会影响输出查询。

作者提出的模型有以下部分。(a) 该模式被转换为图形。(b) 该图根据输入问题进行了软修剪。(c) 图神经网络为了解全局模式结构的节点生成表示。(d) 编码器和解码器使用模式表示。我们现在将详细说明每个部分。

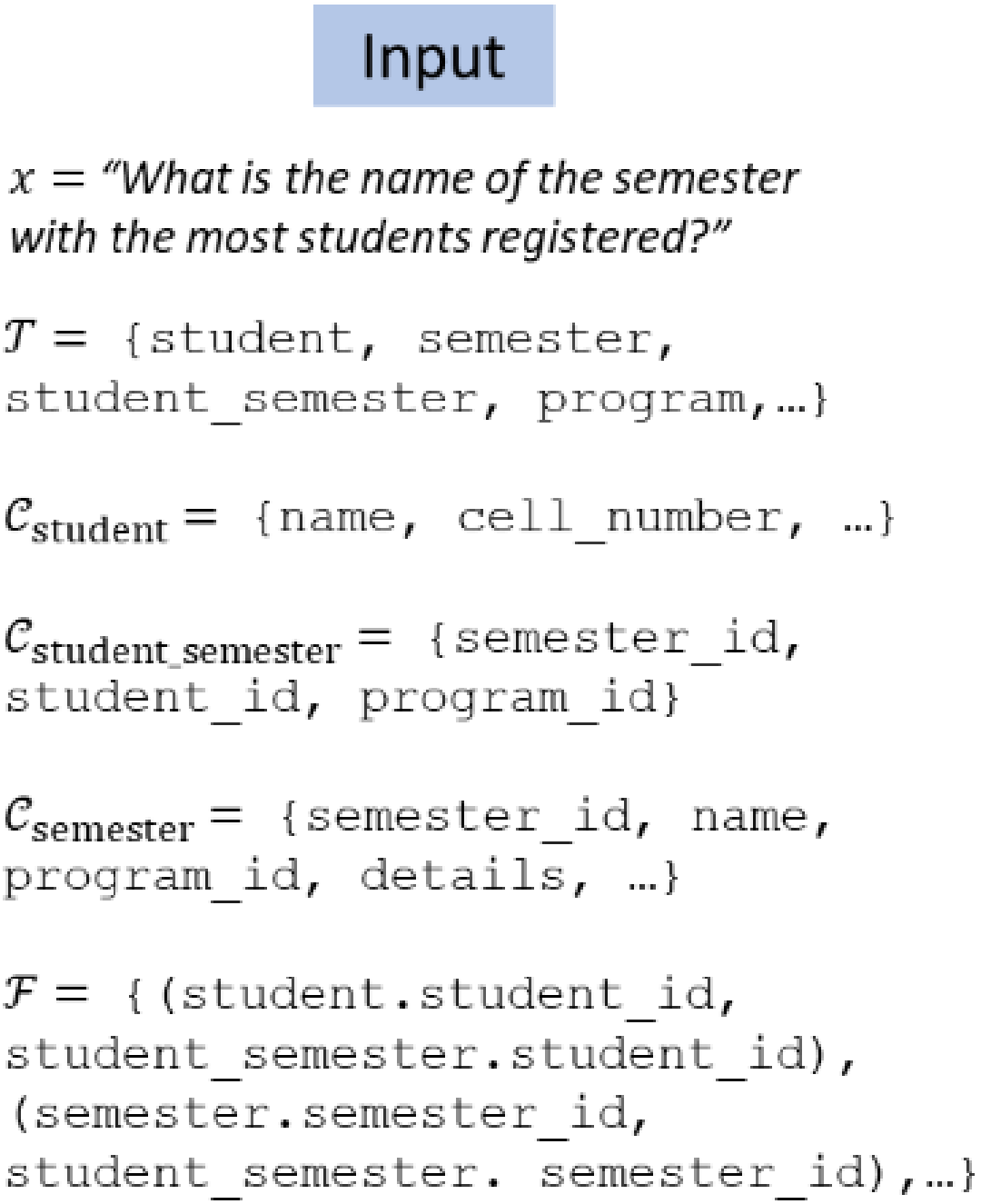

图 5: 数据库模式和问题

3.2 Schema-to-graph

为了将模式 S 转换为图，作者将图节点定义为模式项 V。作者添加三种类型的边：对于表 t 中的每一列 c_t，我们将边 (c_t, t) 和 (t, c_t) 添加到边集 $\mathcal{E}_{\leftrightarrow}$（绿色边）。对于每个外键列对 $(c_{t1}, c_{t2}) \in \mathcal{F}$，我们将边 (c_{t1}, c_{t2}) 和 (t_1, t_2) 添加到边集 $\mathcal{E}_{\leftrightarrow}$ 并将边 (c_{t1}, c_{t2}) 和 (t_1, t_2) 添加到 $\mathcal{E}_{\leftrightarrow}$（虚线边）（如图 5）。边缘类型由图形神经网络用于捕获列和表彼此相关的不同方式。

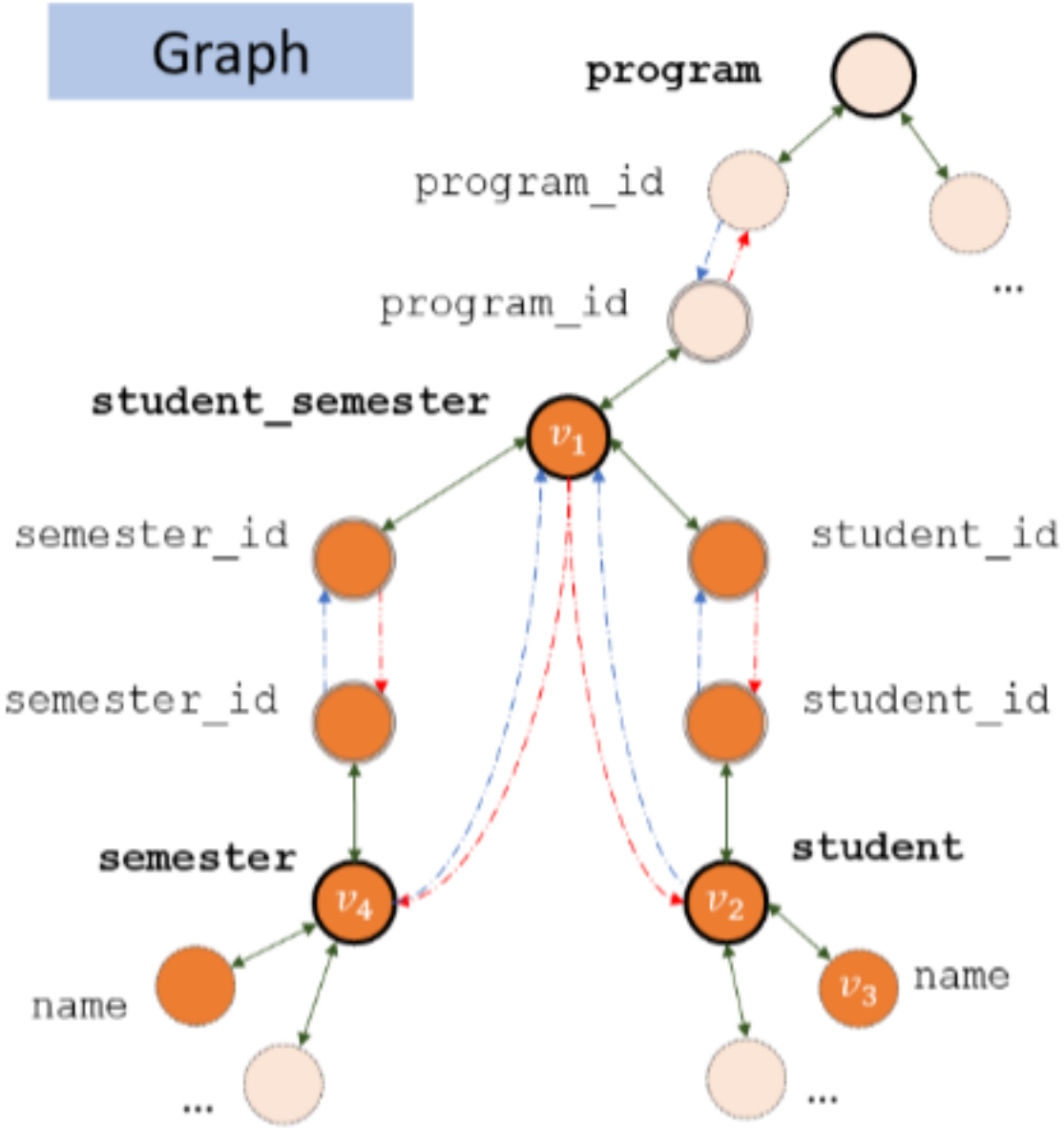

图 6: 数据库模式和问题

3.3 Question-conditioned relevance

每一个问题涉及图式的不同部分，因此，我们的表征应该根据问题的不同而改变。例如，在图 6 中，表 student_semester 和 program 表之间的关系是无关紧要的。为了对此进行建模，作者使用之前方法提出的分布 $p_{link}(\cdot)$，并定义一个 schema 项 v：$\rho_v = max_i p_{link}(v|x_i)$——任何字 x_i 的最大概率 v。接下来作者使用这个分数来创建一个问题条件图表示。图 6 以深橙色显示相关的架构项目，以浅橙色显示不相关的项目。

3.4 Neural graph representation

为了学习考虑其相关性得分和全局模式结构的节点表示，作者使用门控 GNN。每个节点 v 都被赋予一个以相关性分数为条件的初始嵌入：$h_v^{(0)} = r_v \cdot \rho_v$。然后作者将 GNN 递归应用于 L 步骤。在每一步中，每个节点都会根据上一步中其邻居的表示重新计算其表示：

$$a_v^{(l)} = \sum_{\text{type} \in \{\rightarrow, \leftrightarrow\}} \sum_{(u,v) \in \mathcal{E}_{\text{typ}}} W_{\text{type}} h_u^{l-1} + b_{\text{type}}$$

然后 $h_v^{(l)}$ 使用标准 GRU 更新计算如下：

$$h_v^{(l)} = \text{GRU}\left(h_v^{(l-1)}, a_v^{(l)}\right)$$

作者用 $\varphi_v = h_v^{(L)}$ 表示 L 步骤后每个模式项的最终表示。我们现在展示解析器如何使用这种表示。

3.5 Encoder

在之前中，模式项 l_i 的加权平均值连接到每个单词 x_i。为了适应数据库模式表示，作者计算 $l_i^{\varphi} = \sum_{\tau} \sum_{v \in \mathcal{V}_\tau} \varphi_v p_{\text{link}}(v \mid x_i)$，这与 l_i 相同，除了使用 φ_v 代替 r_v。作者将 l_i^{φ} 连接到编码器 h_i 的输出，这样每个单词都用它链接到的模式项周围的图形结构来增强。

3.6 Decoder

如前所述，当模式项 v 被解码时，下一个时间步的输入是其类型 $\tau(v)$。第一个变化是用 φ_v 替换 $\tau(v)$，它了解 v 周围的结构。第二个变化是链接到模式的自我注意机制，我们接下来将对此进行描述。作者将 l_i^{φ} 连接到编码器 h_i 的输出，这样每个单词都用它链接到的模式项周围的图形结构来增强。在对模式项进行评分时，其分数应取决于其与先前解码的模式项的关系。例如，在图 7 中，一旦表 semester 被解码，它很可能会连接到相关表。作者通过自我注意机制捕捉这种直觉。

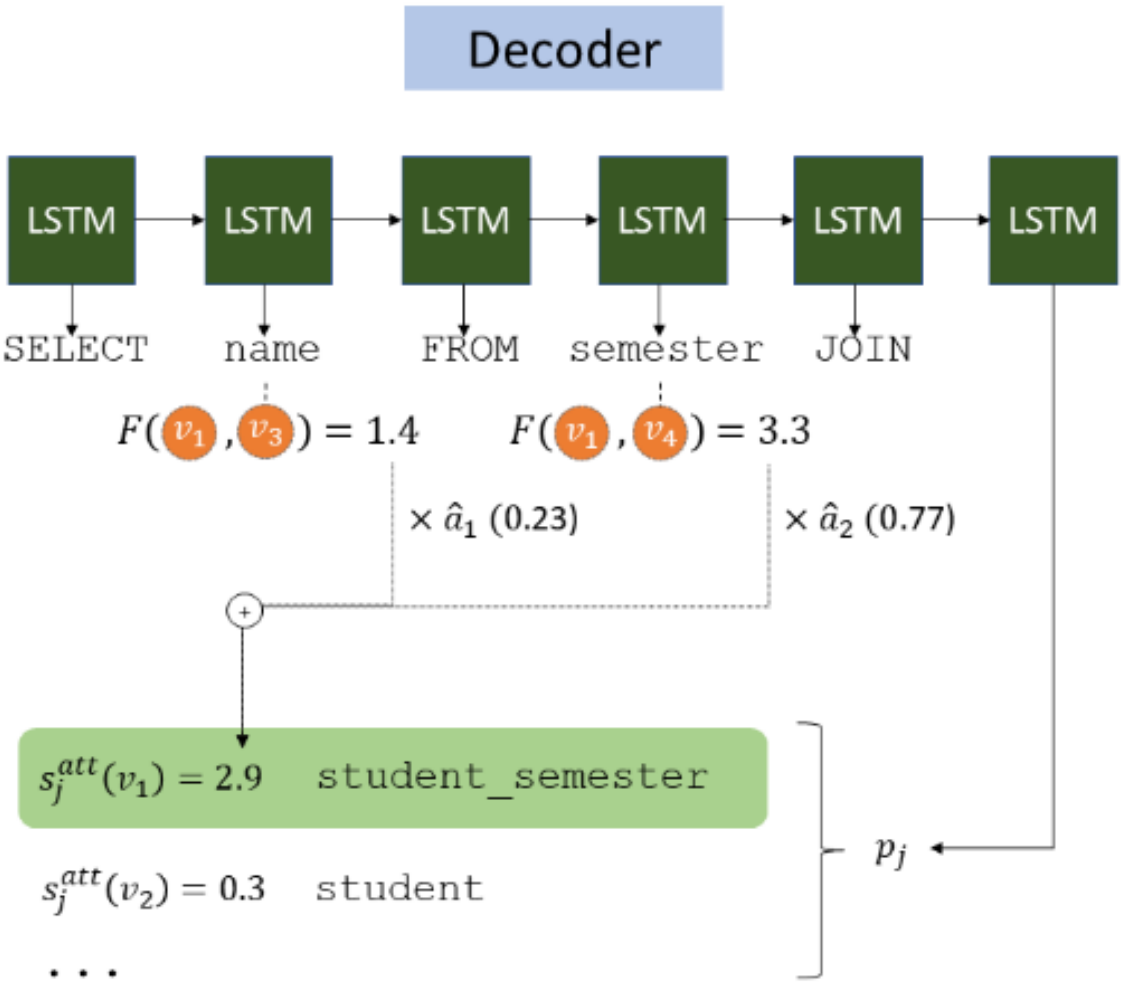

图 7: 解码器

对于每个解码步骤 j，作者用 u_j 表示解码器的隐藏状态，用 $\hat{J} = \left(i_1, \ldots, i_{|\hat{J}|}\right)$ 表示 j 之前模式项已被解码的时间步列表。作者定义矩阵 $\hat{U} \in \mathbb{R}^{d \times |J|} = \left[u_{i_1}, \ldots, u_{i_{\hat{J}_J}}\right]$，它连接了所有这些时间步长的

隐藏状态。作者现在计算这些时间步长的自注意力分布，并根据此分布对模式项进行评分：

$$\hat{a}_j = \text{softmax}\left(\hat{U}^T u_j\right) \quad \in \mathbb{R}^{|\hat{J}|}$$

$$s_j^{\text{att}} = \hat{a}_j S^{\text{att}}$$

$$p_j = \text{softmax}\left(\left[s_j^{\text{glob}}; s_j^{\text{loc}} + s_j^{\text{att}}\right]\right)$$

其中所述矩阵 $S^{\text{att}} \in \mathbb{R}^{|\hat{J}| \times \mathcal{V}_{\text{legal}}}$ 计算先前解码的架构项与根据语法合法的架构项之间的相似性：$S^{att} = F\left(\varphi_{v_1}\right)^{\top} F\left(\varphi_{v_2}\right)$，其中 $F(\cdot)$ 是前馈网络。因此，如果大量关注其具有高度相似性的架构项，则架构项的分数增加。

4 复现细节

4.1 与已有开源代码对比

在开源代码的基础上进行了改进，我主要跟根据作者给出的框架结构，在 GNN 模型中添加了 schema linking 模块，主要修改了训练模型中的对问题词和数据库模型的关联处理。但是实现的效果比较差，还需要进一步的改进。

4.2 实验环境搭建

1. 安装适合 CUDA 版本的 pytorch 1.0.1.post2

pip install https://download.pytorch.org/whl/cu100/torch-1.0.1.post2-cp37-cp37m-linux_x86_64.whl

2. 安装其余所需的软件包

pip install -r requirements.txt

3. 运行此命令以安装 NLTK punkt。

python -c ”import nltk; nltk.download(’punkt’)”

4. 从官方 Spider 数据集网站下载数据集

5. 编辑配置文件 train_configs/defaults.jsonnet 以更新数据集的位置：

local dataset_path = ”dataset/”;

4.3 创新点

该作者使用图神经网络，主要是用有向图来表示外键之间的关系，但是这种方法有一个缺点：即作者没有将 schema 与 question 结合起来编码，因此，在列和问题词被表示之后（这里的表示主要是 embedding 词向量和 schema 的一些内容），对 schema linking 模式链接的推理会变得很困难。

我的工作是主要是在 GNN 模型中添加了 schema linking 模块。针对每个表、列节点，使用 glove 的 embedding 来获得词向量，并用 BiLSTM 来处理词向量，并用另外独立的 BiLSTM 来获取 question 的 embedding。输入 X 是所有节点表示的集合

$$X = (c_1^{init}, \cdots, c_{|C|^{init}}, t_1^{init}, \cdots, t_\tau^{init}, q_1^{init}, \cdots, q_{|Q|}^{init})$$

这里的 encoder 会堆叠多层的 Relation-aware self-attention 层模型对已有的关系信息会有偏向性，优化模型对其他关系的预测能力，并使用 relation type 来处理 schema 的 embedding 和 linking 工作

5 实验结果分析

作者使用完整模型 (GNN) 在测试集中获得了 39.4% 的准确性，大大高于先前的最新技术 (SYNTAXSQLNET)，后者的准确性为 19.7%。将 GNN 从解析器 (NO GNN) 中移除，结果是 Krishnamurthy 等人 (2017) 的解析器中添加了 SQL 语法，获得了 33.8% 的准确性，这表明编码模式结构的重要性。表 1 显示了对比结果。第一列描述了对整个数据集的准确性，接下来的两列显示了对只涉及一个表 (SINGLE) 的查询和涉及多个表 (MULTI) 的查询进行分区时的准确性。

Method	Acc.	SINGLE	MULTI
SQLNET	0.9%	13.6%	3.3%
SYNTAXSQLNET	18.9%	23.1%	7.0%
NO GNN	34.9%	52.3%	14.6%
GNN	40.7%	52.2%	26.8%
- NO SELF ATTEND	38.7%	54.5%	20.3%
- ONLY SELF ATTEND	35.9%	47.1%	23.0%
- NO REL.	37.0%	50.4%	21.5%
GNN ORACLE REL.	54.3%	63.5%	43.7%

表 1: 公共数据集上各个模型结果指标对比

下面的表 2 描述了再公共数据集上 Transtab 和 XGBoost 的迁移学习结果 arouc 指标对比。实验中迁移学习使用的方式是将一个数据集分为重合度为 50% 的两个数据集 s1 和 s2，在 s1 上预训练，在 s2 上进行微调，然后再 s2 上测试性能。我们同样可以看到除了 adult 数据集，transtab 的表现都比 xgboost 好。

GNN 显著优于以前发布的 SQLNET 和 SYNTAXSQLNET，并将 NO GNN 的性能从 34.9% 提高到 40.7%。重要的是，使用模式结构可以特别地提高具有多个表的问题的性能，从 14.6% 提高到 26.8%。

我们去除模型中主要的新组件，以评估它们的影响。首先，我们去掉了自我注意组件 (NO SELF ATTEND)。我们观察到性能下降了 2 点，其中 SINGLE 略有改善，MULTI 下降了 6.5 点。其次，为了验证改进不仅仅是由于自我注意，我们取消了 GNN 的所有其他用途。也就是说，我们使用了一个与 NO GNN 相同的模型，除了它可以通过自我注意 (ONLY SELF ATTEND) 访问 GNN 表示。我们观察到性能大幅下降到 35.9%，这表明所有组件都很重要。最后，通过设置所有模式项的 $_v = 1$(NO REL.) 来消除相关性得分。事实上，准确率下降到 37.0%。

为了评估具有完美关联分数的上限性能，我们运行了一个 oracle 实验，其中我们为 gold 查询中的所有模式项设置 ρv = 1，为所有其他模式项设置 $_v = 0$(GNN oracle REL.)。我们看到，一个完美的相关性得分大大提高了绩效，达到 54.3%。

6 总结与展望

在本文中，作者提出了一个编码器-解码器语义解析器，其中 DB 模式的结构使用图神经网络进行编码，此表示稍后在编码和解码时使

当然模型也存在一定的不足，首先，它没有将 schema 与 question 结合起来编码，因此，在列和问题词被表示之后（这里的表示主要是 embedding 词向量和 schema 的一些内容），对 schema linking 模式链接的推理会变得很困难；其次，它将模式编码期间的信息传播限制在预定义的外键关系图中，这就没有考虑全局信息，之前的研究中已经证明全局信息对有效的表示结构关系非常重要。