第六章 GNN在NL2SQL任务的研究

6.1 图神经网络概述

***Graph neural networks: A review of methods and applications***

近年来，得益于深度神经网络提取潜在表示的有效性，从图像分类和视频处理到语音识别和自然语言理解，深度学习已经彻底改变了许多机器学习任务。然而，这些任务中的数据仅限于欧几里德空间。现实生活中存在越来越多的应用，其中数据从非欧几里德域生成并且表示为对象之间具有复杂关系和相互依赖性的图。图形数据的复杂性给现有的机器学习算法带来了巨大的挑战。

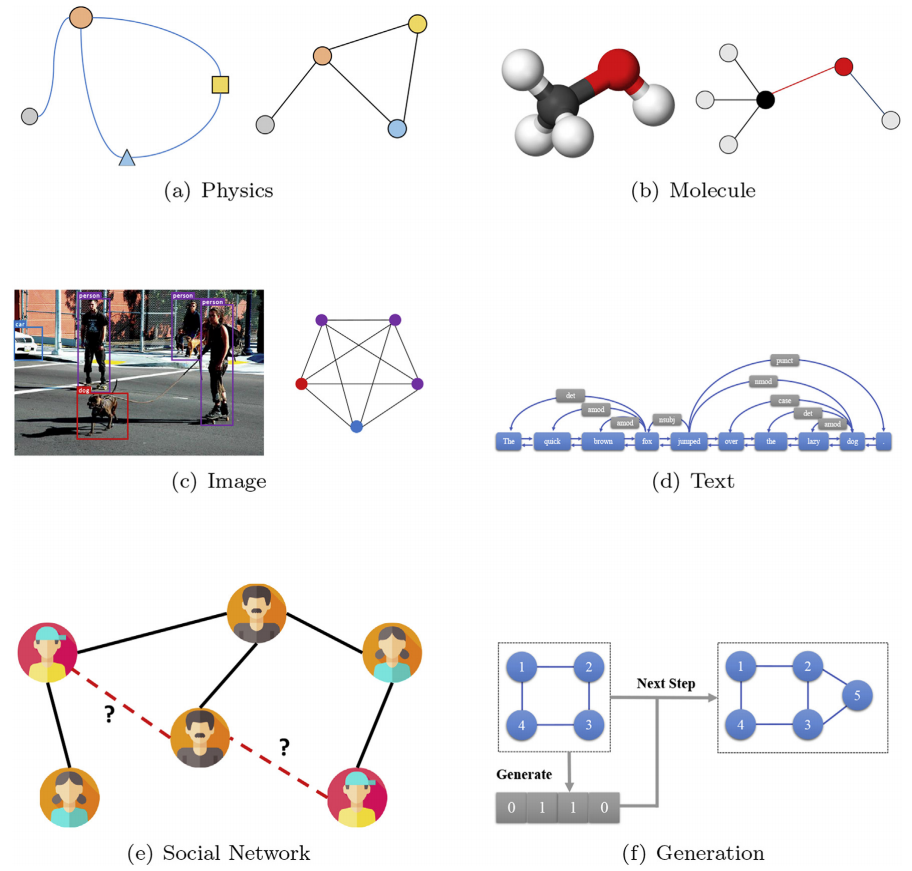


图6.1 GNN应用场景

图是对对象及其相互关系的一种简洁抽象的直观数学表达，具有相互关系的数据—图结构数据在众多领域（图6.1）普遍存在并得到广泛应用。随着大量数据的涌现，传统的图算法在解决一些深层次的重要问题，如节点分类和链路预测等方面有很大的局限性。图神经网络提供了一种利用非欧几里得空间数据的广义形式，通过利用图数据之间的关系来解决这个问题。一个图可以可视化为节点和边的聚合，而没有任何顺序。数据驱动的体系结构倾向于遵循一个固定的神经网络，试图在特征空间中找到该模式。这些策略已成功地应用于解决许多问题，显示了令人信服的可靠性能。例如预测化学分子的特性、文本的关系提取、图形图像的结构推理、社交网络的链路预测和节点聚类、缺失信息的网络补全和药物的相互作用预测等。

6.2 图神经网络的分类

图嵌入的目的是将网络节点表示为低维向量表示，该向量既保留了网络拓扑结构，又保留了节点内容信息，以便后续的图分析任务。网络嵌入主要包括矩阵分解、随机游走等非深度挖掘方法，和自编码器等深度学习方法。图神经网络旨在以端到端的方式解决图相关的任务，因此主要通过图自动编码器框架来解决网络嵌入问题。

图神经网络(GNN)主要分为递归图神经网络(RecGNN)、卷积图神经网络(Con-vGNN)、图自编码器(GAE)和时空图神经网络(STGNN)四个类别，下面我们简要介绍每个类别。

递归图神经网络(RecGNNs)是图神经网络的先驱。RecGNNs的目标是学习具有递归神经结构的节点表示。他们假设图中的一个节点不断地与它的邻居交换信息/消息，直到一个稳定的平衡为止。RecGNNs在概念上很重要，并启发了后来对卷积图神经网络的研究。特别是，基于空间的卷积图神经网络继承了消息传递的思想。

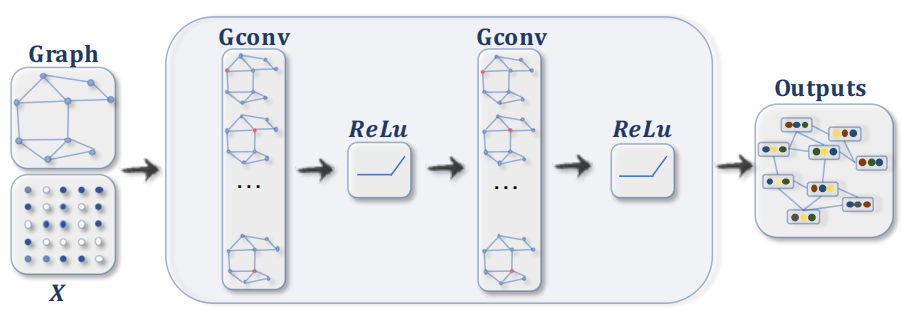


图6.2 一种具有多个图卷积层的ConvGNN

卷积图神经网络(ConvGNNs)将卷积的操作从网格数据推广到图形数据。其主要思想是通过聚合节点自己的特征和邻居的特征来生成节点的表示。与RecGNNs不同，ConvGNNs堆叠多个图卷积层来提取高级节点表示。ConvGNNs在构建许多其他复杂的GNN模型中发挥着核心作用。图6.2是一种具有多个图卷积层的ConvGNN，图卷积层通过聚合来自相邻节点的特征信息来封装每个节点的隐藏表示。在特征聚合后，对得到的输出应用非线性转换。通过叠加多层，每个节点的最终隐藏表示都会接收来自另一个邻域的消息。图6.3展示了一个用于图分类的ConvGNN。图卷积层之后是一个池化层，将图合并成子图，以便粗化图上的节点表示表示更高的图级表示。一个读出层通过取子图的隐藏表示的和/平均值来总结最终的图表示。

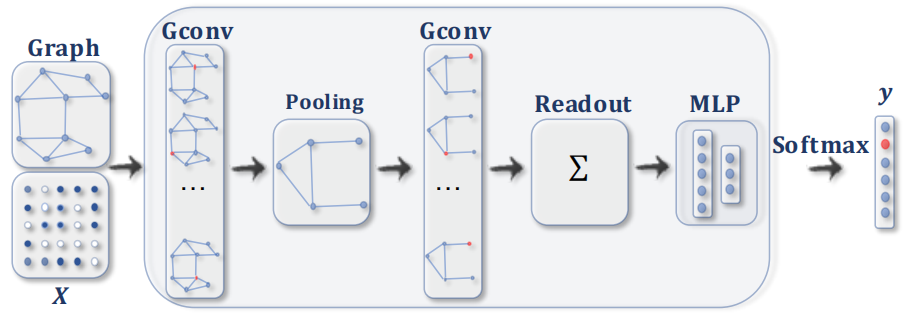


图6.3 一种具有图形分类的池化层和输出层的ConvGNN

图自编码器(GAEs)是一种无监督的学习框架，它将节点/图编码为一个潜在的向量空间，并从编码的信息中重建图数据。GAEs用于学习网络嵌入和图生成分布。对于网络嵌入，GAEs通过重构图邻接矩阵等图结构信息来学习潜在节点的表示。对于图的生成，一些方法一步一步地生成图的节点和边，而其他方法则一次输出图。图6.4中编码器使用图卷积层为每个节点获得一个网络嵌入。解码器计算给定的网络嵌入的成对距离。在应用非线性激活函数后，解码器重建图邻接矩阵。通过最小化实邻接矩阵和重建的邻接矩阵之间的差异来训练网络。

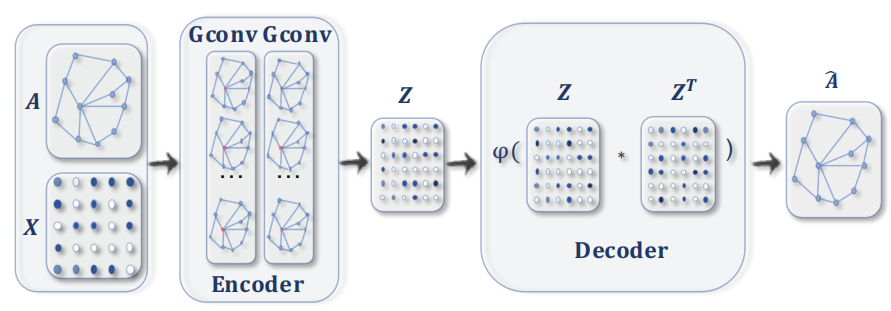


图6.4 一种用于网络嵌入的GAE

时空图神经网络(STGNNs)的目标是从时空图中学习隐藏模式，这在交通速度预测、驾驶员机动预期和人类动作识别等各种应用中越来越重要。STGNNs的关键思想是同时考虑空间依赖性和时间依赖性。目前的许多方法集成图卷积来捕获RNNs或CNNs的空间依赖来建模时间依赖。图6.5一种用于时空图预测的STGNN。一个图卷积层之后是一个1D-CNN层。图卷积层在A和X(t)上运行以捕获空间依赖关系，而1D-CNN层沿着时间轴在X上滑动以捕获时间依赖关系。输出层是线性变换，对每个节点生成预测，例如在下一个时间步的未来值。

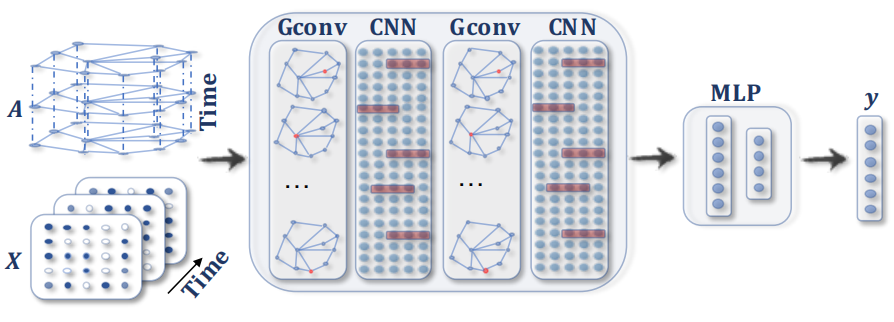


图6.5 一种用于时空图预测的STGNN

6.3 NL2SQL中的GNN

在NL2SQL任务中，SQL语句蕴含着数据表的结构信息。然而，一方面由于以往的数据集中的数据库非常简单，另一方面，在训练和测试时都能观察到数据库本身，导致对SQL解析语言的研究在很大程度上忽略了数据库模式的结构。BenBogin等研究人员首次提出使用GNN作为编码器-解码器的语义解析器，用图神经网络对数据库模式的结构进行编码，并在编码和解码时同时使用这种表示。数据库模式S包括：

1. 表名集合
2. 涉及表的所有列名集合
3. 外键到主键的映射集合

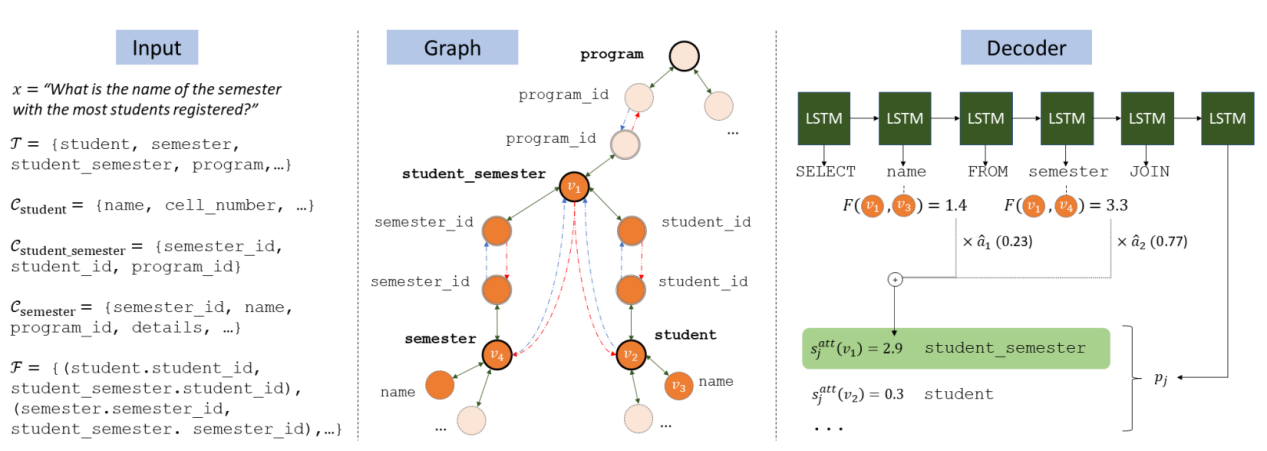


图6.6 NL2SQL模型结构

6.3.1 利用GNN构建NL2SQL生成框架

GNN模型结构见图6.6，模型包括以下步骤：首先从模式中匹配问句的可能模式项的集合，根据该集合构造图，利用GNN学习图表示，然后将句子嵌入和图嵌入送入Encoder得到模式的隐藏层的表示，最后Decoder进行解码。我们将详细说明每一部分。

1. 将模式转换为图形

为了将模式S转换为图6.6左，首先将图节点定义为模式项目V。然后添加三种类型的边：对于表t中的每列ct，我们将边（ct、t）和（t、ct）添加到绿色边集合。对于每个外主键列对（ct1，ct2），我们在添加边（ct1，ct2）和（t1，t2）到单项边集合，（ct2，ct1）和（t2，t1）到单向边虚线集合。这些边的类型被图神经网络用于捕获列和表相互关联的不同方式。

1. 根据问题条件的相关性对图形剪枝

每个问题都涉及到模式的不同部分，因此，我们的图表示应该根据这个问题而改变。例如，表student semester和表program是不相关的。为了对这种情况进行剪枝，通过单词和数据库常量之间的局部相似性函数来解析查询中涉及到得表名和列名，需要计算任何单词匹配的数据库常量的最大概率，得到概率最大的数据库常量。

1. GNN自编码表示

使用门控GNNs学习综合相关性分数和全局模式结构的节点表示，每个节点v都有一个基于相关性分数条件的初始嵌入：。然后，我们应用GNN递归计算L步。在每个步骤中，每个节点根据上一步中其邻居的表示来重新计算其表示：

然后使用标准GRU更新计算h(vl)如下：

1. 编码器和解码器使用模式表示

Encoder部分为双向LSTM，将模式项的加权平均拼接到每个单词上，这样每个单词都可以用围绕其链接到的模式项的图形结构进行扩充。Decoder部分为基于语法的LSTM。

6.3.2 GNN-based改进：Global GNN

上述的语义解析器主要通过单词和数据库常量之间的局部相似性函数获取text相关的数据库模式，由于该函数在计算时是自回归的，每次得到一个结果，并未考虑结果之间的全局依赖关系，根据局部相似性选择数据库常量的例子很困难。考虑图6.7中的示例，在解码SELECT之后，解码器现在必须选择数据库常量，假设它的注意力集中在单词“name”上，并且仅给出局部相似性，那么在词汇相关的数据库常量（singer.name和song.name）之间的选择会含糊不清。然而，如果我们对数据库常数和问题进行全局推理，我们可以组合额外的线索。首先，随后的词汇“nation”属于表singer的列country，因此选择singer表的可能性更大。其次，“name”下一次出现在“Hey”旁，这个短语是song.name的一个值。假设单词和数据库常量之间是一对一映射，那么第二个“name”属于song.name，第一个“name”属于singer.name。

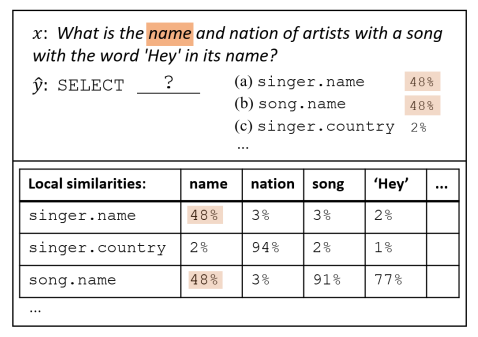


图6.7 局部相似性结果

Global GNN的模型结构见下图6.8，其全局推理能力主要体现在两个方面的改进：Gating GCN模块与Re-ranking GCN模块。第一个是为了解决上述局部推理的模糊问题，通过训练一个Gating GCN模块得到每个单词对不同数据库常量的选择概率，来为Encoder动态选择可能的数据库常量；第二个重排序是为了考虑全局单词的候选常量，综合整个query的上下文信息来进行排序。

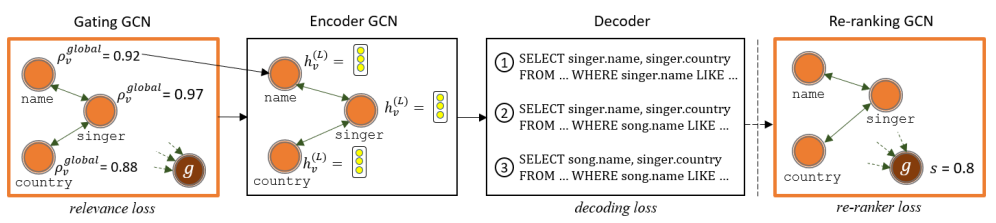


图6.8 Global GNN模型结构

6.3.2 重排序模块

全局门控编码模块为预测数据库常数的正确子集提供了一个更准确的模型。然而，解析仍然是自回归的方式，每次只取Top 1的结果同样陷入了局部最优解。为了从全局推理的角度选择数据库常量，解码器训练了一个新的判别模型对输出的每个beam中Top K个结果进行重新排序，从而可以对整个候选查询进行全局推理。

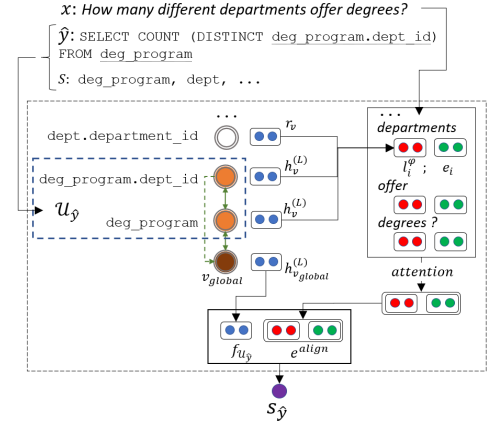


图6.9 解码器模块

6.3.3 AllenNLP开发框架

代码主要基于AllenNLP框架，AllenNLP是一个开源库，用于为自然语言处理构建深度学习模型，由艾伦人工智能研究所开发。它构建在PyTorch之上，旨在支持那些希望轻松构建高质量深度NLP模型的研究人员、工程师和学生等。它为现代NLP中的通用组件和模型提供了高级抽象和API。它还提供了一个可扩展的框架，便于运行和管理NLP实验。简而言之，AllenNLP是一个经过深思熟虑的抽象库，封装了在NLP研究中完成的通用数据和模型操作，其主要特性如下所示：

* 训练PyTorch模型的命令行工具
* 一组预先训练过的模型，你可以用它来进行预测
* 常用/最新NLP模型的可读参考实现集合
* 一个用于快速复现实验结果的实验框架
* 展示研究成果的一种方式
* 开源和社区驱动

AllenNLP被大量的组织和研究项目所使用。

基于AllenNLP的数据预处理，主要有两个模块，一是将训练数据处理成语法解析模型的输入，而是将模型输出的Top K的结果重新构成重排序模块的输入。每个模块只需要重写\_read方法读取数据，\_read\_examples\_file方法特殊处理和text\_to\_instance方法生成数据的格式，其他预处理过程，例如分词、构造词典、token转id，AllenNLP会通过初始化参数自动帮你完成。

6.3.4 Spider解析模型

1. @DatasetReader.register("spider")
2. class SpiderDatasetReader(DatasetReader):
3. def \_\_init\_\_(self,
4. lazy: bool = False,
5. question\_token\_indexers: Dict[str, TokenIndexer] = None,
6. keep\_if\_unparsable: bool = True,
7. tables\_file: str = None,
8. dataset\_path: str = 'dataset/database',
9. load\_cache: bool = True,
10. save\_cache: bool = True,
11. loading\_limit = -1):
12. super().\_\_init\_\_(lazy=lazy)
13. spacy\_tokenizer = SpacyWordSplitter(pos\_tags=True)
14. spacy\_tokenizer.spacy.tokenizer.add\_special\_case(u'id', [{ORTH: u'id', LEMMA: u'id'}])
15. self.\_tokenizer = WordTokenizer(spacy\_tokenizer)
16. self.\_utterance\_token\_indexers = question\_token\_indexers or {'tokens': SingleIdTokenIndexer()}
17. self.\_keep\_if\_unparsable = keep\_if\_unparsable
18. self.\_tables\_file = tables\_file
19. self.\_dataset\_path = dataset\_path
20. self.\_load\_cache = load\_cache
21. self.\_save\_cache = save\_cache
22. self.\_loading\_limit = loading\_limit

图 Spider数据处理模块

这里主要介绍\_read\_examples\_file方法，他涉及到如下数据模式的转化。

* + 跳过无法解析的示例
  + 跳过解析出错的示例
  + 将列名和表名对应 'name' -> 'singer@name'
  + 移除别名 'singer AS T1' -> 'singer'
  + 指代消解 'T1.name' -> 'singer@name'

1. def \_read\_examples\_file(self, file\_path: str):
2. cache\_dir = os.path.join('cache', file\_path.split("/")[-1])
3. if self.\_load\_cache:
4. logger.info(f'Trying to load cache from {cache\_dir}')
5. if self.\_save\_cache:
6. os.makedirs(cache\_dir, exist\_ok=True)
7. cnt = 0
8. with open(file\_path, "r") as data\_file:
9. json\_obj = json.load(data\_file)
10. for total\_cnt, ex in enumerate(json\_obj):
11. cache\_filename = f'instance-{total\_cnt}.pt'
12. cache\_filepath = os.path.join(cache\_dir, cache\_filename)
13. if self.\_loading\_limit == cnt:
14. break
15. if self.\_load\_cache:
16. try:
17. ins = dill.load(open(cache\_filepath, 'rb'))
18. ins = self.process\_instance(ins, total\_cnt)
19. if ins is None and not self.\_keep\_if\_unparsable:
20. continue
21. yield ins
22. cnt += 1
23. continue
24. except Exception as e:
25. pass
26. query\_tokens = None
27. if 'query\_toks' in ex:
28. ex = fix\_number\_value(ex)
29. query\_tokens = disambiguate\_items(ex['db\_id'], ex['query\_toks\_no\_value'],
30. self.\_tables\_file, allow\_aliases=False)
31. ins = self.text\_to\_instance(
32. utterance=ex['question'],
33. db\_id=ex['db\_id'],
34. sql=query\_tokens)
35. ins = self.process\_instance(ins, total\_cnt)
36. if ins is not None:
37. cnt += 1
38. if self.\_save\_cache:
39. dill.dump(ins, open(cache\_filepath, 'wb'))
40. if ins is not None:
41. yield ins

图 \_read\_examples\_file方法

Spider解析模型结构如下所示，AllenNLP提供了绝大部分常见的模块，只用配置参数即可。

1. @Model.register("spider")
2. class SpiderParser(SpiderBase):
3. def \_\_init\_\_(self,
4. vocab: Vocabulary,
5. encoder: Seq2SeqEncoder,
6. entity\_encoder: Seq2VecEncoder,
7. decoder\_beam\_search: BeamSearch,
8. question\_embedder: TextFieldEmbedder,
9. input\_attention: Attention,
10. past\_attention: Attention,
11. max\_decoding\_steps: int,
12. action\_embedding\_dim: int,
13. gnn: bool = True,
14. graph\_loss\_lambda: float = 0.5,
15. decoder\_use\_graph\_entities: bool = True,
16. decoder\_self\_attend: bool = True,
17. gnn\_timesteps: int = 2,
18. pruning\_gnn\_timesteps: int = 2,
19. parse\_sql\_on\_decoding: bool = True,
20. add\_action\_bias: bool = True,
21. use\_neighbor\_similarity\_for\_linking: bool = True,
22. dataset\_path: str = 'dataset',
23. training\_beam\_size: int = None,
24. decoder\_num\_layers: int = 1,
25. dropout: float = 0.0,
26. rule\_namespace: str = 'rule\_labels') -> None:

图 SpiderParser模型参数定义

解码器部分如下所示，根据是否使用self\_attention实现Pointer拷贝机制拷贝问题中出现过单词作为候选分为两个解码器进行对比实验。

1. if decoder\_self\_attend:
2. self.\_transition\_function = AttendPastSchemaItemsTransitionFunction(
3. encoder\_output\_dim=encoder\_output\_dim,
4. action\_embedding\_dim=action\_embedding\_dim,
5. input\_attention=input\_attention,
6. past\_attention=past\_attention,
7. predict\_start\_type\_separately=False,
8. add\_action\_bias=self.\_add\_action\_bias,
9. dropout=dropout,
10. num\_layers=self.\_decoder\_num\_layers)
11. else:
12. self.\_transition\_function = LinkingTransitionFunction(
13. encoder\_output\_dim=encoder\_output\_dim,
14. action\_embedding\_dim=action\_embedding\_dim,
15. input\_attention=input\_attention,
16. predict\_start\_type\_separately=False,
17. add\_action\_bias=self.\_add\_action\_bias,
18. dropout=dropout,
19. num\_layers=self.\_decoder\_num\_layers)

图 解码模块

6.3.5 Spider\_Rerank重排序模型

Spider\_rerank.py数据处理模块与Spider数据处理模块稍有不同，它的数据输入是候选集输出是候选集和相应部分的子图，如下所示：

1. @DatasetReader.register("spider\_rerank")
2. class SpiderRerankDatasetReader(SpiderDatasetReader):
3. def \_\_init\_\_(self,
4. sub\_graphs\_candidates\_path: str,
5. max\_candidates: int,
6. sub\_sample\_candidates: bool,
7. lazy: bool = False,
8. unique\_sub\_graphs: bool = True,
9. question\_token\_indexers: Dict[str, TokenIndexer] = None,
10. keep\_if\_unparsable: bool = True,
11. tables\_file: str = None):
12. super().\_\_init\_\_(lazy, question\_token\_indexers, keep\_if\_unparsable, tables\_file)
13. self.\_unique\_sub\_graphs = unique\_sub\_graphs
14. self.\_sub\_graphs\_candidates\_path = sub\_graphs\_candidates\_path
15. self.\_sub\_graphs\_candidates = []
16. self.\_max\_candidates = max\_candidates
17. self.\_sub\_sample\_candidates = sub\_sample\_candidates
18. self.\_query\_token\_indexers = {'query\_tokens': SingleIdTokenIndexer()}

图 Spider\_Rerank模型参数定义

Spider\_rerank.py的核心方法是重写process\_instance，用于从原始图谱中根据候选集抽取子图，返回子图，候选集以及它们的标签。

1. @overrides
2. def process\_instance(self, instance: Instance, index: int = None, candidates: List = None):
3. """
4. This function is called after the instance was loaded with the basic dataset reader, and adds the query
5. candidates data.
6. """
7. assert index is not None or candidates is not None
8. if instance is None:
9. return instance
10. fields = instance.fields
11. world: SpiderWorld = fields['world'].metadata
12. del fields['valid\_actions']
13. if 'action\_sequence' in fields:
14. del fields['action\_sequence']
15. *# Get the correct sub-graph (our supervision)*
16. if world.query is not None:
17. correct\_sub\_graph = set()
18. for token in world.query:
19. if token in world.entities\_names:
20. correct\_sub\_graph.add(token)
21. original\_candidates = candidates or self.\_sub\_graphs\_candidates[index]
22. if self.\_sub\_sample\_candidates:
23. shuffled\_candidates = list(original\_candidates)
24. *# make sure there is a correct candidate, so put them on top*
25. shuffled\_candidates = sorted(shuffled\_candidates, key=lambda x: (x['correct'], random()), reverse=True)
26. else:
27. shuffled\_candidates = original\_candidates
28. sub\_graphs = []
29. label\_fields = []
30. sub\_graphs\_label\_fields = []
31. entities\_names = [ent\_key\_to\_name(e) for e in world.db\_context.knowledge\_graph.entities
32. if e.startswith('column:') or e.startswith('table:')]
33. unique\_sub\_graphs = set()
34. kept\_candidates = []
35. *# go through the candidate queries and extract their sub-graph*
36. for candidate in shuffled\_candidates:
37. if self.\_sub\_sample\_candidates and len(kept\_candidates) == self.\_max\_candidates:
38. break
39. query\_tokens = candidate['query'].split()
40. sub\_graph = set()
41. candidate\_entities = []
42. for i, token in enumerate(query\_tokens):
43. ent\_id = -1
44. potential\_ent = token.replace('.', '@')
45. if potential\_ent in entities\_names:
46. sub\_graph.add(potential\_ent)
47. ent\_id = world.entities\_names[potential\_ent]
48. candidate\_entities.append(LabelField(ent\_id, skip\_indexing=True))
49. if not sub\_graph:
50. continue
51. if not self.\_unique\_sub\_graphs:
52. sub\_graphs.append(sub\_graph)
53. else:
54. sub\_graph\_hash = tuple(sorted(sub\_graph))
55. if sub\_graph\_hash in unique\_sub\_graphs:
56. continue
57. unique\_sub\_graphs.add(sub\_graph\_hash)
58. sub\_graphs.append(sub\_graph)
59. kept\_candidates.append(candidate)
60. if candidate['correct'] is not None:
61. label\_fields.append(LabelField(int(candidate['correct']), skip\_indexing=True))
62. sub\_graphs\_label\_fields.append(LabelField(int(sub\_graph == correct\_sub\_graph), skip\_indexing=True))
63. *# check if we should return all examples (even when no correct answers found)*
64. if not self.\_keep\_if\_unparsable:
65. if self.\_unique\_sub\_graphs and not any([sg == correct\_sub\_graph for sg in sub\_graphs]):
66. return None
67. if not self.\_unique\_sub\_graphs and not any([l.label for l in label\_fields]):
68. return None
69. sub\_graph\_candidates = []
70. for sub\_graph in sub\_graphs:
71. entities\_ids = []
72. for ent in sub\_graph:
73. ent\_id = world.entities\_names[ent]
74. entities\_ids.append(LabelField(ent\_id, skip\_indexing=True))
75. if not entities\_ids:
76. continue
77. sub\_graph\_candidates.append(ListField(entities\_ids))
78. *# we give both the subgraphs and the actual query candidates*
79. fields['sub\_graphs'] = ListField(sub\_graph\_candidates)
80. fields['candidates'] = MetadataField(kept\_candidates)
81. if sub\_graphs\_label\_fields:
82. fields['sub\_graphs\_labels'] = ListField(sub\_graphs\_label\_fields)
83. if label\_fields:
84. fields['candidates\_labels'] = ListField(label\_fields)
85. return Instance(fields)

图 process\_instance方法

Spider\_rerank的模型结构比较简单，但是计算过程比较复杂，因为它要同时考虑全局子图的依赖关系及候选结果的得分。

1. @Model.register("spider\_reranker")
2. class SpiderReranker(SpiderBase):
3. def \_\_init\_\_(self,
4. vocab: Vocabulary,
5. encoder: Seq2SeqEncoder,
6. entity\_encoder: Seq2VecEncoder,
7. question\_embedder: TextFieldEmbedder,
8. action\_embedding\_dim: int,
9. attention: Attention,
10. gnn\_timesteps: int = 2,
11. dropout: float = 0.0,
12. rule\_namespace: str = 'rule\_labels') -> None:

图 Spider\_rerank模型参数

计算图如下所示，其主要内容为计算候选得分\_score\_candidates方法（略）和验证候选\_compute\_validation\_outputs方法（略）。

1. @overrides
2. def forward(self,  *# type: ignore*
3. utterance: Dict[str, torch.LongTensor],
4. world: List[SpiderWorld],
5. schema: Dict[str, torch.LongTensor],
6. sub\_graphs,
7. candidates,
8. sub\_graphs\_labels=None,
9. candidates\_labels=None
10. ) -> Dict[str, torch.Tensor]:
11. batch\_size = len(world)
12. num\_candidates = sub\_graphs.size(1)
13. candidates\_scores, debug\_info = self.\_score\_candidates(utterance, world, schema, sub\_graphs)
14. candidates\_scores = candidates\_scores.view(batch\_size, num\_candidates)
15. outputs: Dict[str, Any] = {}
16. if sub\_graphs\_labels is not None:
17. candidates\_labels = candidates\_labels.view(batch\_size, num\_candidates)
18. sub\_graphs\_labels = sub\_graphs\_labels.view(batch\_size, num\_candidates)
19. unanswerable\_mask = sub\_graphs\_labels.max(dim=1)[0] == 0
20. unanswerable\_mask2 = sub\_graphs\_labels.min(dim=1)[0] == 1
21. num\_unanswerable = unanswerable\_mask.sum() + unanswerable\_mask2.sum()
22. if batch\_size - num\_unanswerable > 0:
23. candidates\_mask = sub\_graphs.max(dim=-1)[0] != -1
24. normalized\_scores = masked\_softmax(candidates\_scores, candidates\_mask)
25. loss = -torch.log((normalized\_scores \* sub\_graphs\_labels.float()).sum(dim=1))
26. loss[unanswerable\_mask] = 0
27. loss[unanswerable\_mask2] = 0
28. loss = loss.sum() / (batch\_size - num\_unanswerable)
29. else:
30. loss = (candidates\_scores - candidates\_scores).sum()
31. outputs['loss'] = loss
32. self.\_compute\_validation\_outputs(world,sub\_graphs,candidates\_scores,
33. sub\_graphs\_labels,candidates,outputs)
34. return outputs

图 Spider\_rerank模型计算图

6.4 GNN-based改进：RAT-SQL

模式泛化具有挑战性，因为有三个相互关联的原因。首先，任何文本到SQL的解析模型都必须将模式编码为适合解码可能涉及给定列或表的SQL查询的表示。其次，这些表示应对有关架构的所有信息进行编码，如其列类型、外键关系和用于数据库连接的主键。最后，模型必须识别用于引用列和表的自然语言，由于spider数据集的训练集、测试集没有重叠部分，所以是零镜头的预测，这种情况下，模型对模式的泛化能力就至关重要了。后一个挑战被称为模式链接——将问题中的实体引用与预期的模式列或表对齐。

Global-GNN主要研究了模式编码的问题，而对模式链接的探索相对较少。见下图6.10中的示例。它说明了模式链接中的挑战：问题中的“model”指的是car\_names.model而不是model\_list.model，“car”实际上指的是cars\_data和car\_names而不是car\_makers。为了正确地解析列名、表名的引用，语义解析器必须同时考虑到已知的模式关系和问题的上下文。

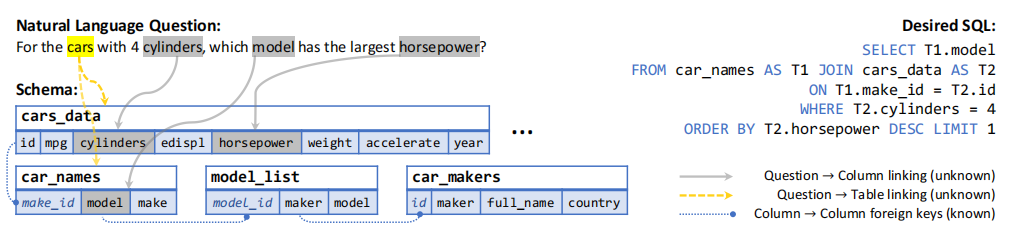


图6.10 来自复杂任务的挑战

RAT-SQL是对Global-GNN进一步的改进，Global-GNN主要是考虑从问题中匹配到表名和列名的联系，然后将这种联系拼接到问题的单词上，而RAT-SQL直接在图中考虑表名、列名和问题三种节点，而且利用字符串匹配的方法丰富了边的类型。GNN-based使用图神经网络，主要是用有向图来表示外键之间的关系，但是这种方法有两个缺点：首先，它没有将模式与问题结合起来编码，因此，在列和问题词被表示之后，对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，而全局信息对有效的表示结构关系非常重要。为了解决以上问题，RATSQL提出了一个统一的编码器，其使用relation-aware self-attention对显式关系（模式）和隐式关系（问题和模式之间的链接）进行编码，完善了模型的表示能力。

6.4.1 Relation-Aware Self-Attention

Relation-Aware Self-Attention是一种编码半结构化输入序列的模型，它联合编码序列中预先存在的关系结构和序列元素之间的“软”关系。其基本算子Self-Attention如下所示：

RAT-SQL提供了一种方法，如下所示，通过改变方程来描述自我注意层中的相对位置信息的方法，将已知关系添加到注意机制中。

这里的rij项编码了输入中两个元素xi和xj之间的已知关系，xi和xj之间本身是没有先后顺序关系的，这与原始的Self-Attention机制不符，而他们存在的依赖关系类似于一种相对位置信息编码，用它替换原始Self-Attention中的位置信息，可以得到Relation-Aware Self-Attention算子结构。

6.4.2 RAT-SQL的创新点

RAT-SQL中定义的边类型以及示例如下图所示，总共有十一类，但是在模式链接的过程中，会出现精确匹配，部分匹配和不匹配三种情况，因此总共会出现33类边。下面是一种匹配的模式样例。

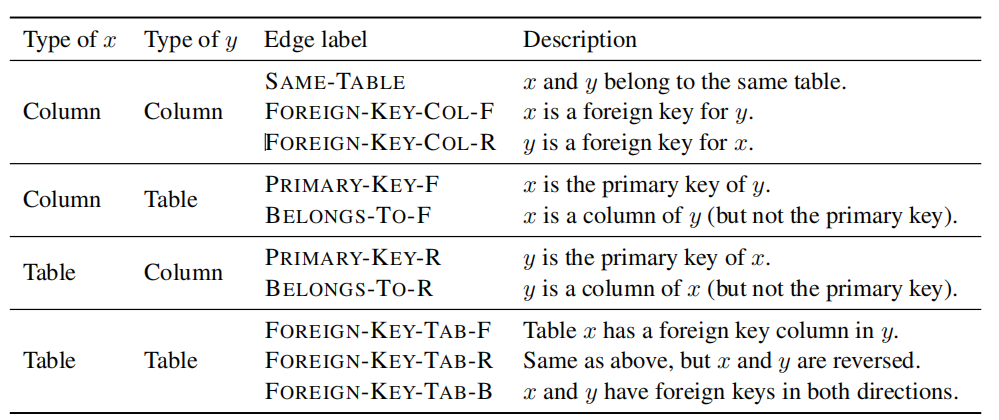


图6.11 边信息的描述

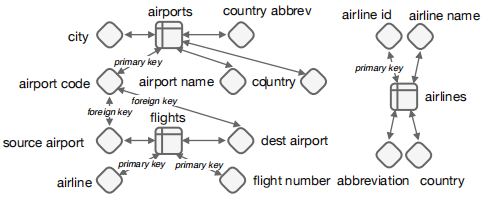


图6.11 模式示例

RAT-SQL的解码器主要目标是生成相关的AST树（CFG上下文无关的抽象语法树）。它使用深度优先遍历顺序生成SQL作为一个抽象语法树，使用LSTM输出解码器操作序列，它要么是语法树结构，要么是列名和表名。

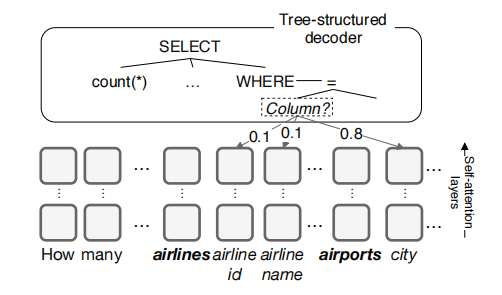
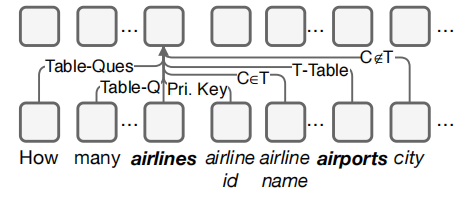


图6.11 解码器解码过程

6.4.3 核心代码实现

上述Relation-Aware Self-Attention相关公式实现如下：

1. def relative\_attention\_logits(query, key, relation):
2. qk\_matmul = torch.matmul(query, key.transpose(-2, -1))
3. q\_t = query.permute(0, 2, 1, 3)
4. r\_t = relation.transpose(-2, -1)
5. q\_tr\_t\_matmul = torch.matmul(q\_t, r\_t)
6. q\_tr\_tmatmul\_t = q\_tr\_t\_matmul.permute(0, 2, 1, 3)
7. return (qk\_matmul + q\_tr\_tmatmul\_t) / math.sqrt(query.shape[-1])
8. def relative\_attention\_values(weight, value, relation):
9. wv\_matmul = torch.matmul(weight, value)
10. w\_t = weight.permute(0, 2, 1, 3)
11. w\_tr\_matmul = torch.matmul(w\_t, relation)
12. w\_tr\_matmul\_t = w\_tr\_matmul.permute(0, 2, 1, 3)
13. return wv\_matmul + w\_tr\_matmul\_t
14. def attention\_with\_relations(query, key, value, relation\_k, relation\_v, mask=None, dropout=None):
15. d\_k = query.size(-1)
16. scores = relative\_attention\_logits(query, key, relation\_k)
17. if mask is not None:
18. scores = scores.masked\_fill(mask == 0, -1e9)
19. p\_attn\_orig = F.softmax(scores, dim = -1)
20. if dropout is not None:
21. p\_attn = dropout(p\_attn\_orig)
22. return relative\_attention\_values(p\_attn, value, relation\_v), p\_attn\_orig

模式链接，分别计算匹配的列和表，每个列或表分为部分匹配，完全匹配和不匹配三种关系，通过内部函数partial\_match和exact\_match进行计算。

值链接，通过向数据库检索特殊的字段，数字、日期等，利用外部知识作为模式链接的背景知识。

1. def compute\_schema\_linking(question, column, table):
2. def partial\_match(x\_list, y\_list):
3. x\_str = " ".join(x\_list)
4. y\_str = " ".join(y\_list)
5. if x\_str in STOPWORDS or x\_str in PUNKS:
6. return False
7. if re.match(rf"\b{re.escape(x\_str)}\b", y\_str):
8. assert x\_str in y\_str
9. return True
10. else:
11. return False
12. def exact\_match(x\_list, y\_list):
13. x\_str = " ".join(x\_list)
14. y\_str = " ".join(y\_list)
15. if x\_str == y\_str:
16. return True
17. else:
18. return False
19. q\_col\_match, q\_tab\_match = dict(), dict()
20. col\_id2list = dict()
21. for col\_id, col\_item in enumerate(column):
22. if col\_id == 0:
23. continue
24. col\_id2list[col\_id] = col\_item
25. tab\_id2list = dict()
26. for tab\_id, tab\_item in enumerate(table):
27. tab\_id2list[tab\_id] = tab\_item
28. n = 5
29. while n > 0:
30. for i in range(len(question) - n + 1):
31. n\_gram\_list = question[i:i + n]
32. n\_gram = " ".join(n\_gram\_list)
33. if len(n\_gram.strip()) == 0:
34. continue
35. for col\_id in col\_id2list:
36. if exact\_match(n\_gram\_list, col\_id2list[col\_id]):
37. for q\_id in range(i, i + n):
38. q\_col\_match[f"{q\_id},{col\_id}"] = "CEM"
39. for tab\_id in tab\_id2list:
40. if exact\_match(n\_gram\_list, tab\_id2list[tab\_id]):
41. for q\_id in range(i, i + n):
42. q\_tab\_match[f"{q\_id},{tab\_id}"] = "TEM"
43. for col\_id in col\_id2list:
44. if partial\_match(n\_gram\_list, col\_id2list[col\_id]):
45. for q\_id in range(i, i + n):
46. if f"{q\_id},{col\_id}" not in q\_col\_match:
47. q\_col\_match[f"{q\_id},{col\_id}"] = "CPM"
48. for tab\_id in tab\_id2list:
49. if partial\_match(n\_gram\_list, tab\_id2list[tab\_id]):
50. for q\_id in range(i, i + n):
51. if f"{q\_id},{tab\_id}" not in q\_tab\_match:
52. q\_tab\_match[f"{q\_id},{tab\_id}"] = "TPM"
53. n -= 1
54. return {"q\_col\_match": q\_col\_match, "q\_tab\_match": q\_tab\_match}

图 模式链接

1. def compute\_cell\_value\_linking(tokens, schema):
2. def isnumber(word):
3. try:
4. float(word)
5. return True
6. except:
7. return False
8. def db\_word\_match(word, column, table, db\_conn):
9. cursor = db\_conn.cursor()
10. p\_str = f"select {column} from {table} where {column} like '{word}" \
11. f" %' or {column} like '% {word}' or " \
12. f"{column} like '% {word} %' or {column} like '{word}'"
13. try:
14. cursor.execute(p\_str)
15. p\_res = cursor.fetchall()
16. if len(p\_res) == 0:
17. return False
18. else:
19. return p\_res
20. except Exception as e:
21. return False
22. num\_date\_match = {}
23. cell\_match = {}
24. for q\_id, word in enumerate(tokens):
25. if len(word.strip()) == 0:
26. continue
27. if word in STOPWORDS or word in PUNKS:
28. continue
29. num\_flag = isnumber(word)
30. CELL\_MATCH\_FLAG = "CELLMATCH"
31. for col\_id, column in enumerate(schema.columns):
32. if col\_id == 0:
33. assert column.orig\_name == "\*"
34. continue
35. *# word is number*
36. if num\_flag:
37. if column.type in ["number", "time"]:  *# TODO fine-grained date*
38. num\_date\_match[f"{q\_id},{col\_id}"] = column.type.upper()
39. else:
40. ret = db\_word\_match(word, column.orig\_name, column.table.orig\_name, schema.connection)
41. if ret:
42. *# print(word, ret)*
43. cell\_match[f"{q\_id},{col\_id}"] = CELL\_MATCH\_FLAG
44. cv\_link = {"num\_date\_match": num\_date\_match, "cell\_match": cell\_match}
45. return cv\_link

图 值链接

6.5 本章小结

本章我们探讨了GNN在NL2SQL任务上的相关研究，首先我们介绍了GNN的相关内容，然后介绍了Global GNN，其首次将GNN引入NL2SQL模式表示，最后我们介绍了一个统一的问题、列、表的统一编码器RAT-SQL。

Global GNN提出通过全局决策提高语义解析泛化能力的尝试，其中的关键难点在于选择相关的数据库常量集。其贡献主要体现在两个方面。其一，使用一个全局的门控GCN和整个DB模式来选择DB常量。其二，同过对选定的数据库常量集进行全局评分，对生成语义解析器的输出进行重新排序。这两个贡献可以以最小的代价融入到编码-解码框架中，提高模型的整体性能。

RAT-SQL提出了一个统一的框架来解决模式编码和链接的问题，利用关系感知的自注意力机制，它根据模式之间的对齐以及模式关系共同学习模式和问题表示，提供了一种在同一编码器架构中结合预定义的模式和自动选择相应模式关系的方法。只要输入有某种预定义的结构，这种表示学习在NL2SQL之外的任务中也是有益的。