Improving Multi-hop Knowledge Base Question Answering by Learning Intermediate Supervision Signals

一、论文：

1. 摘要（Abstract）：

- **目标：**在知识图谱上找到距离问句中实体若干跳之外的答案实体。
  - multi-hop KBQA：不仅答案要正确，推理的关系路径是否正确也十分重要。
- **挑战：**传统KBQA问句集只有问题和最终答案对，缺少中间的监督信息。
- **创新：**提出了一个新颖的teacher-student方法，student网络目标是找到问题的正确答案，teacher网络试图学习中间的监督信号，以此来增强学生网络推理的过程。
  - 主要创新点在于teacher网络，利用了前向和后向推理来增强对中间实体分布的学习，将双向推理结合后，teacher网络可以产生更加可靠的中间监督信号。

2. 简介（Introduction）：

- **为什么推理的关系路径也很重要?**
  - 有时候错误的路径也能找到正确的答案，如下图：
    - 红色路径：正确的关系路径
    - 蓝色路径：伪路径
    - 绿色节点：话题（问题）实体（topic entity）
    - 红色节点：最终答案实体（answer entity）

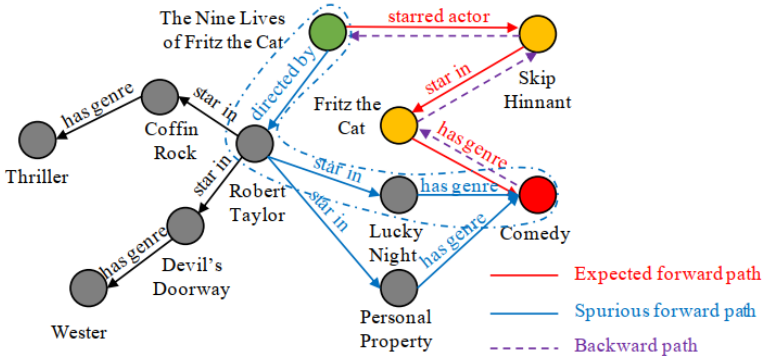


Figure 1: A spurious case from MetaQA-3hop dataset. We use green, red, yellow and grey circles to denote the topic entities, correct answer, intermediate entities and irrelevant entities respectively.

  - 原因：训练过程中缺乏中间监督信息。
- **强化学习的问题?**
  - 解决方式：
    - 设置一个policy-based agent来逐渐扩张推理链，直到找到target entity。
    - 它的状态通常被定义为query 和当前实体组成的tuple，动作被定义为在KB上通过当前实体的出边进行游走。
  - 问题：奖励函数的解决方法需要expert experience，要么依然缺乏中间步骤的监督信息
- **本文的解决方案?**
  1. 定义前向推理和后向推理的双向BFS来解决标注数据的问题：

    - 前向推理：从topic entity到answer entity的推理
    - 后向推理：从answer entity到topic entity的推理
  2. **前向推理路径上的实体应该和后向推理路径上的实体分布应该相近。**
  3. teacher-student方法来实现这一学习的过程：
    - student network：用一个Neural State Machine(NSM)来实现，用来找到最终的答案实体。
    - teacher network：提供前向推理和后向推理的相关性来增强对中间实体分布的学习。

3. 相关工作（Related Work）：

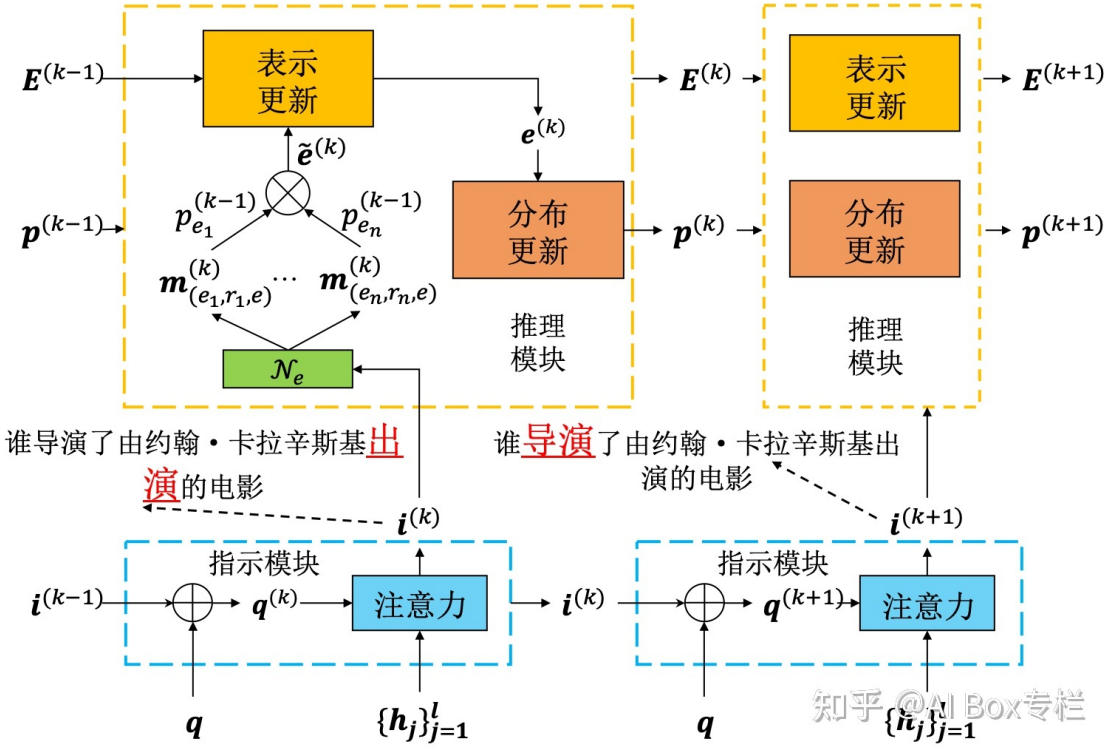
- **Knowledge Base Question Answering（KBQA）：**知识图谱问答问题分为两类基于语义分析的方法和基于检索的方法，但是传统的方式效果都不好：
  - 基于语义分析的方法（semantic parsing based method）：尝试将人类语言问题翻译成机器能够理解的逻辑性问题。
  - 基于检索的方法（retrieval based method）：基于检索的方法根据问题中传达的信息直接从知识库中检索答案。
- **Multi-hop Reasoning：**多条推理近几年是图像和自然语言处理的热点问题，尝试使用Attention、RNN、强化学习和图卷积来解决。
- **Teacher-student Framework：**知识蒸馏中常用teacher-student 框架，一个复杂的高性能的模型作为teacher，一个简单的轻量的模型作为student。
  - teacher预测的结果被作为软标签，student的目标就是学习软标签。
  - **主要用来做模型压缩。**

4. 相关定义（Preliminary）：

- 使用三元组(e_1, r, e_2)表征知识图谱（头—关系—尾）
- 其中实体和关系向量：$e \in R^d$和$r \in R^d$，d表示embedding维数
- k和(k)表示第k步推理
- 其中所有的三元组包含：
  - 正三元组(e_1, r, e_2)
  - 逆三元组(e_2, r^{-1}, e_1)

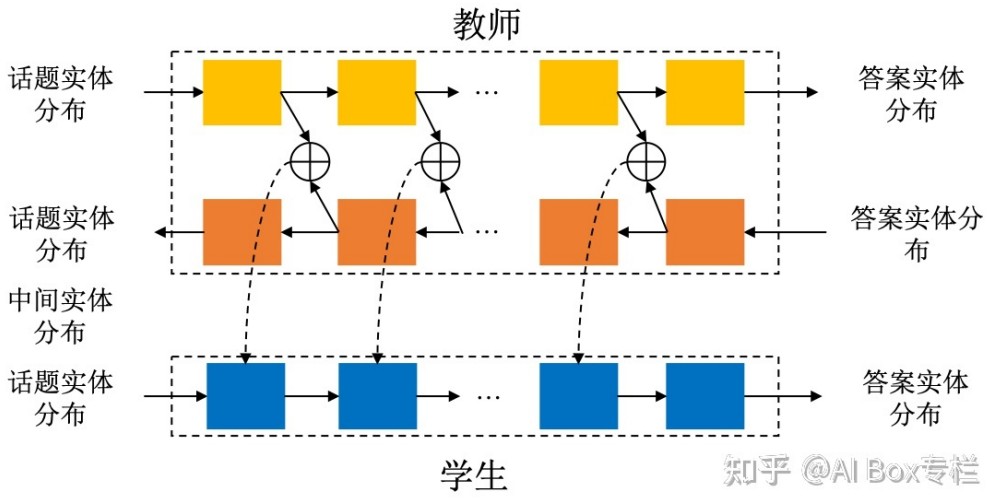
5. 模型设计（Approach）：

- **Neural State Machine for Multi-hop KBQA（NSM）：**指令模块向推理模块发送指令向量，推理模块根据指令向量推断实体的分布，学习实体的表示。



- **指令模块（Instruction Component）**：负责将问题转换成推理指令
  - 输入：
    - 问题矩阵query embedding（用GloVe包获取的word vector）
    - 上一个推理步骤的指示矩阵
  - 执行：
    1. 首先进行LSTM编码器获取输入问题矩阵query embedding的隐藏状态输出$\{h_j\}_{j=1}^{l}$，$h_j \in R^d$，**最后一个隐藏状态**h_l作为问题的特征表征矩阵，令其等于q（$q = h_l$），
       - 这里h_j表示第j层的隐藏状态
       - l表示输入query的长度
    2. $i^{(k)}$代表第k次推理步骤的指示矩阵：
       - 这里定义第k次步骤的权重参数和偏置$W^{(k)}$，W_α，$b^{(k)}$和b_α
         - $W^{(k)} \in R^{d \times 2d}$
         - $W_\alpha \in R^{d \times d}$
         - $b^{(k)}, b_\alpha \in R^d$
       - $q^{(k)} = W^{(k)}[i^{(k-1)}; q] + b^{(k)}$
         - 首先凭借上一次（第k-1次）的指令矩阵和最后一层隐藏层的句子表征$q = h_l$
         - 通过一个线性层转换，将因为拼接的2倍embedding维数降至原始的embedding大小d
         - 最终得到中间变量$q^{(k)}$
       - $\alpha_j^k = softmax_j(W_\alpha(q^{(k)} \odot h_j) + b_\alpha)$
         - 将j个不同的隐藏层和$q^{(k)}$点对点乘法（广播）
         - 经过一个线性层处理
         - 最后加softmax，计算attention系数，用于推理的不同步骤关注问题的不同部分（不同步骤关注的是哪个j）
    3. $i^{(k)} = \sum_{j=1}^{l} \alpha_j^{(k)} h_j$，$i^{(k)} \in R^d$：
       - 最后通过attention系数$\alpha_j^{(k)}$和每句话隐藏层输出h_j乘积并累加
       - 得到指示矩阵$i^{(k)} \in R^d$

        - 输出：
            - 这个批次k的指示矩阵$i^{(k)}$
    - **推理模块（Reasoning Component）：**推理模块根据指令向量推断实体的分布，学习实体的表示
        - 输入：
            - 这个批次k的指示矩阵$i^{(k)}$
            - 上一步的实体分布（entity distribution）
            - 上一步的实体矩阵（entity embedding）
        - 操作：
            1. 首先通过单层图网络编码关系初始化实体表示（entity embedding）：
                - $e^{(0)} = \sigma(\sum_{<e_1,r,e>\in N_e} rW_T)$
                - r：表示节点周围邻居的关系矩阵
                - W_T：单层网络权重
                - 优点：
                    - 通过邻居关系刻画实体表示的方式能够为训练过程未涉及的实体提供较好的语义表示。
                    - 减少带噪实体对于邻居实体表示的影响。
            2. 得到中间变量匹配矩阵$m^{(k)}$（match vector），$m_{<e_1,r,e>} = \sigma(i^{(k)} \odot W_R r)$
                - $W_R r$：$W_R \in R^{d\times d}$的可学习参数和节点和关系矩阵r的单层线性网络
                - $\sigma(i^{(k)} \odot W_R r)$：$W_R r$和指示矩阵按位element-wise相乘再经过激活函数非线性话
                - 每次推理时，NSM模型依据指示向量与关系信息进行匹配（有点向邻居筛选）
            3. 计算得到候选的关系聚合实体$\tilde{e}^{(k)}$（relation-aggregated entity embedding），$\tilde{e}^{(k)} = \sum_{<e_1,r,e>\in N_e} p_{e_1}^{(k-1)} \cdot m_{<e_1,r,e>}^{(k)}$
                - N_e：实体e的邻居
                - $p^{(k-1)}$：第$k-1$步的实体分布（entity distribution）
                - $p_{e_1}^{(k-1)}$：第$k-1$步（上一批次）出发节点推理步骤的e_1的概率
                - 推理模块基于实体邻居N_e为实体e聚合匹配向量，并根据$k-1$步推理时邻域实体相关性进行加权聚合。
            4. 更新实体矩阵（entity embedding），$e^{(k)} = W^{(k)}([e^{(k-1)}; \tilde{e}^{(k)}]) + b^{(k)}$
                - 上一次$e^{(k-1)}$和$\tilde{e}^{(k)}$进行拼接
                - 经过单层线性层得到本批次的$e^{(k)}$（entity embedding）
            5. 更新实体相关性分布（entity distribution），$p^{(k)} = softmax({E^{(k)}}^T w)$
                - w：可学习参数
                - $E^{(k)}$：本批次实体矩阵的集合（entity embedding）
- **基于双向推理自动学习中间监督信号：**
    - **老师 — 学生模式：**
        - **思路：**为了让模型获得较强的多跳推理能力同时避免歧义推理，我们首先训练教师网络来学习可靠的中间推理过程。在教师网络收敛后，学生网络在多跳推理的中间步骤也能获得有效反馈。



- **启发：**受到老师网络双向搜索算法的启发，可以利用双向推理机制增强教师网络生成的中间监督信号。
  - 前向推理：除了之前相关研究多聚焦于从话题实体出发推理答案实体。
  - 后向推理：这里额外考虑了从答案实体出发反向推理话题实体。
  - 通过让这两类推理过程在多跳推理的中间步骤保持实体分布的一致性，教师网络能够学习到更加可靠的中间监督信号。
  - 假设一个n步的推理过程，我们同时获得了前向推理和后向推理中间实体分布$\{p_f^{(k)}, p_b^{(k)} | k = 1, \cdots, n-1\}$，其下标$f$和$b$表示前向后后向。
    - 目标使得前后向推理的腹部应保持一致：$p_f^{(k)} \approx p_b^{(n-k)}$

- **The Teacher Network：**
  - **平行推理（Parallel Reasoning）：**设置两个独立的NSM模块，一个前向推理，另一个反向推理。这两个网络不共享参数，相互独立。
  - **复合推理（Hybrid Reasoning）：**共享instruction component，循环进行前向推理和后向推理。
    前向推理的最后输出作为后向推理的最初输入。
    - 在对应的中间推理步骤，两模型输入相同指示向量
    - 两模型的推理模块串联成环（先前向再后向）
    - 除对齐约束外，该架构将后向推理过程的初始状态初始化为前向推理过程的最终输出
- **损失函数：**均采用KL散度和JS距离来衡量
  - 前向损失函数：$L_f = D_{KL}(p_f^{(n)}, p_f^*)$，$p_f^*$表示真实分布
  - 后向损失函数：$L_b = D_{KL}(p_b^{(n)}, p_b^*)$，$p_b^*$表示真实分布
  - 前后向分布损失函数（JS距离）：$L_c = \sum_{k=1}^{n-1} D_{JS}(p_f^{(k)}, p_b^{n-k})$
  - 损失函数组合：$L_t = L_f + \lambda_b L_b + \lambda_c L_c$
- **The Student Network：**
  - 在teacher模型训练完之后，可以获得一个中间实体的分布。将前向推理和后向推理的分布加起来作为中间的监督信号。
    - $p_t^k = \frac{1}{2}(p_f^{(k)} + p_b^{(n-k)}), k = 1, \cdots, n-1$
  - 在推理损失之外（L_1），加入距离Teacher Network的损失（L_2）。
    - $L_1 = D_{KL}(p_s^n, p_f^*)$
    - $L_2 = \sum_{k=1}^{n-1} D_{KL}(p_s^k, p_t^{(k)})$
    - $L_s = L_1 + \lambda L_2$

6. 试验结果（Experiment）：

- 的NSM模型能够在大多数情况下较以往的方法可比或更好。
- 加入双向推理机制的教师网络后，NSM模型能够得到进一步提升。
- 神经状态机模型在稀疏的训练数据下仍然表现良好
- 教师模型在稀疏情况下带来的提升更加显著。

二、代码：

1. 数据结构：

- **entities.txt**：存放了**1441420**个实体编号
- **relations.txt**：存放了**6102**个关系
- **vocab_new.txt**：存放了**6718**个新词字典
- **train.dep**：存放了**2848**个问题的句法分析的信息
- **dev.dep**：存放了**250**个问题的句法分析的信息
- **test.dep**：存放了**1639**个问题的句法分析的信息

```
{
    "id": "WebQTrn-0", # 这条问句的id
    "dep": [  # 依存关系list
        [
            "what",  # 实体词
            "WP",  # 命名实体识别（代词）
            "0",  # 该词再句子中的编号
            "ROOT"  # 该词溯源的关系（根节点）
        ],
        [
            "is",
            "VBZ",
            "1",
            "cop"
        ],
        ...
    ],
    "question": "what is the name of justin bieber brother"  # 问答问题的问句
}
```

- **train_simple.json**：存放**2848**个问题的回答、问题、子图关系和实体等
- **dev_simple.json**：存放**250**个问题的回答、问题、子图关系和实体等
- **test_simple.json**：存放**1639**个问题的回答、问题、子图关系和实体等

```
{
    "id": "WebQTrn-0",  # 句子编号id
    "question": "what is the name of justin bieber brother",  # 问题
    "entities":  # 句子中的实体号（对应entities.txt）
    [
        15
    ],
    "answers":
    [
        {
            "kb_id": "m.0gxnnwq",  # 问题回答的id
            "text": "Jaxon Bieber"  # 问题的回答
```

```
        }
    ],
    "subgraph":  # 和问题节点近邻的关联和节点(最大路径)
    {
        "tuples":  # 放置关联的头-关系-尾的编号（entities.txt / relations.txt）
        [
            [
                0,
                0,
                1
            ],
            ...
        ],
        "entities":  # 放置实体编号（entities.txt）
        [
            27,
            121,
            ...
        ]
    }
}
```

2. 数据加载：

- 加载**entity**、**relation**和**vocab**三个文件：
  - **entity2id**：**1441420**个实体标签做key，序号做id，从0开始（`{'m.01vrt_c': 0, ...}`）
  - **word2id**：**6718**个单词标签做key，序号做id，从0开始（`{'m.01vrt_c': 0, ...}`）
  - **relation2id**：**6102**个关系标签做key，序号做id，从0开始（`{'freebase.valuenotation.is_reviewed': 0, ...}`）
  - **id2entity，id2word和id2relation**：和上面的key和value反过来
- 加载**sample**和**dep**文件：
  - **data**：存放**sample**文件中每一行（**line**）的数据
  - **max_facts**：存放**sample**文件中每行问题（**line**）中邻接关系组数（**tuples**）最大的数值乘以2（`len(line['subgraph']['tuples'])`）
  - **dep**：存放**dep**文件中每一行（**line**）的数据
  - **num_data**：行数（`len(self.data)`）
  - **batches**：以行数截至，从0开始的序号（`np.arange(self.num_data)`）
- 构建**全局和局部id映射**（**global2local_entity_maps**）
  - **next_id**：遍历数据（**data**）每一行（**sample**）的行索引号
  - **g2l**：遍历数据（**data**）每一行（**sample**）的**entities**和**subgraph.entities**（`sample['entities'], sample['subgraph']['entities']`），构建原始**entities**的**id / 行号**为**key**，当前**g2l**的**长度**为**value**的局部id值得字典（`g2l[entity_global_id] = len(g2l)`）
  - **total_local_entity**：遍历数据（**data**）每一行（**sample**），累加每一行**g2l**的长度（`total_local_entity += len(g2l)`）
  - **max_local_entity**：遍历数据（**data**）每一行（**sample**），取每一次遍历**g2l**中最大的长度（`max(self.max_local_entity, len(g2l))`）（**2000**）
  - **global2local_entity_maps**：遍历数据（**data**）每一行（**sample**），以行索引号（**next_id**）为key，本轮全局转局部id字典（**g2l**）为value的字典，这个是**返回值**
- 语法依存（**_prepare_dep**）处理：
  - **max_query_word**：遍历数据（**dep**）每一行（**line**），取每一行（也就是每个问句）的单词个数最大值（`max(max_count, len(word_list))`）（**14**）

    - **next_id：**遍历数据（**dep**）每一行（**sample -> edge_list**）的行号
    - **read_tree**：遍历数据（**dep**）每一行（**sample -> edge_list**），返回**node_layer, parents, relations**
        - **parents**：遍历每个句子的每个词，取出每个词的**父节点-1**（`eval(edge_list[i][2]) - 1`）
        - **relations**：遍历每个句子的每个词，取出每个词和父节点的**关系类型**（`(edge_list[i][3]`）
        - **node_layer**：遍历每个句子的每个词，通过递归的方式获取该节点所在节点的**层级数**
    - **node2layer**：遍历数据（**dep**）每一行（**sample -> edge_list**），存放**read_tree**方法返回的**node_layer**
    - **dep_parents**：遍历数据（**dep**）每一行（**sample -> edge_list**），存放**read_tree**方法返回的**parents**
    - **dep_relations**：遍历数据（**dep**）每一行（**sample -> edge_list**），存放**read_tree**方法返回的**relations**
    - **query_texts**：
        - 初始化为新词总数的问句个数乘以问句最多单词数（`(self.num_data, self.max_query_word)`：**(2848, 14)**；`len(self.word2id)`：**6718**）
        - 遍历数据（**dep**）每一行（**sample -> edge_list**），以及遍历每一行的每个单词，如果该单词在**word2id**中，就在该位置存储该id，否则就存储`len(self.word2id)`作为padding（**6718**）
- 语句关系（**_prepare_data**）处理：
    - **next_id：**遍历数据（**data**）每一行（**sample**）的行号
    - **question_id**：遍历数据（**data**）每一行（**sample**），存储每个问题的id编号
    - **query_entities**：
        - 初始化全为0的，问句个数乘以**g2l**最大长度（**(num_word, max_local_entity)**：**(2949, 2000)**）
        - 遍历数据（**data**）每一行（**sample**），再遍历每个**entities**（`sample['entities']`），经过全局id转局部id（`g2l[global_entity]`），如果该处局部id存在，则这里赋值1（`self.query_entities[next_id, local_ent] = 1.0`）
    - **seed_list**：遍历数据（**data**）每一行（**sample**），再遍历每个**entities**（`sample['entities']`），存储此时的局部id号（`seed_list.append(local_ent)`）的**list**，最终每个**sample**存一次这个**list**（**共2848行**）（`(self.num_data, )`：**(2848,)**）
    - **tp_set**：数据（**data**）每一行（**sample**）时才初始化，再遍历每个**entities**（`sample['entities']`），存储此时的局部id号（`seed_list.append(local_ent)`）的**set**（也就是只包含上面**entities**的实体id，后面**subgraph**里的不考虑）
    - **candidate_entities**：
        - 初始化为（`(self.num_data, max_local_entity)`：**(2848, 2000)**），值为最大的全局id+1（**1441420**）
        - 遍历数据（**data**）每一行（**sample**），再遍历这个样本（**sample**）的个**g2l**，如果本次**sample**的**tp_set**里没有包含这个g2l里的本地（**local**）id，那么**candidate_entities**这里就赋值全局（**entity**）id
        - 这里看作时存放了**subgraph**的**entities**的全局id
    - **kb_adj_mats**：
        - 初始化为（`(self.num_data, )`：**(2848,)**），值为None

        - 遍历数据（**data**）每一行（**sample**），再遍历这个样本的**tuples**（`sample['subgraph']['tuples']`）
            - **head_list, rel_list, tail_list**：分别存放头尾的讯号以及转为**int**的关系**list**
        - **kb_adj_mats**存储每次遍历（**2848**）的**head_list, rel_list, tail_list**转成**np.array**的list
    - **seed_distribution**：
        - 初始化为（`(self.num_data, max_local_entity)`：**(2848, 2000)**），值为0
        - 遍历数据（**data**）每一行（**sample**）
            - 如果此次遍历的**tp_set**大于0，那么在**tp_set**存储**local_ent**序号的位置赋值**tp_set**长度的倒数（`1.0 / len(tp_set)`）
            - 否则在本次遍历**g2l**所有的本地序号位置赋值**g2l**长度的倒数（`1.0 / len(g2l)`）
    - **answer_lists**：
        - 初始化为（`(self.num_data, )`：**(2848,)**），值为None
        - 遍历数据（**data**）每一行（**sample**），再遍历每一个回答，**answer_list**添加回答编号在实体中的id（`self.entity2id[answer[keyword]]`），最后**answer_lists**每次循环添加**answer_list**。
    - **answer_dists**：
        - 初始化为（`(self.num_data, max_local_entity)`：**(2848, 2000)**），值为0
        - 遍历数据（**data**）每一行（**sample**），再遍历每一个回答，如果这次回答编号在实体中的id是属于这一批次的全局id（**g2l**），将**answer_dists**中的取局部id的位置设置为1（`self.answer_dists[next_id, g2l[answer_ent]] = 1.0`）

3. 模型初始化：

- **GNNModel**初始化：
    - **embedding_def**：**embedding layer**层级定义
        - 定义参数：
            - **word_dim**：每个单词词向量维度（**300**）
            - **kg_dim**：知识图谱向量维度（**100**）
            - **num_entity**：所有实体个数（**1441420**）
            - **num_relation**：所有关系个数（**6102**）
            - **num_word**：所有单词个数（**6718**）
        - **entity_embedding**：定义**padding**填充（**1441420**），个数为（**1441420 + 1**），维度是**100**的实体**embedding**
        - **relation_embedding**：定义个数为（**6102**），**2**倍知识图谱向量维度（**2 * kg_dim = 200**）的**embedding**
        - **word_embedding**：定义**padding**填充（**6718**），个数为（**6718 + 1**），维度是**300**的实体**embedding**
            - **word_embedding.weight：**从（`word_emb_300d.npy`）文件中加载的词向量（**(6718, 300)**）**padding**一个维度得到（**(6719, 300)**）**word_embedding**的初始化值
    - **share_module_def**：
        - **entity_linear**：定义输入是**kg_dim**（**100**）和输出维度是**entity_dim**（**100**）的单层线性网络
        - **relation_linear**：定义输入是**2 * kg_dim**（**200**）和输出维度是**entity_dim**（**100**）的单层线性网络
        - **lstm_drop**：**lstm**的**dropout**（**p=0.3**）
        - **linear_drop**：先行层的**dropout**（**p=0.2**）
        - **kld_loss**：定义**KL散度**损失函数

        - **bce_loss_logits**：定义二元交叉熵损失函数
        - **mse_loss**：定义均方根损失函数
    - **private_module_def**：
        - **GNNReasoning**：图神经网络推理模块
            - **softmax_d1**：对第一维度的**softmax**（`nn.Softmax(dim=1)`）
            - **score_func**：最后的**二分类**函数，输入时**entity_dim**（**100**），输出是**1**维度
            - **num_step**：定义了这里神经网路重复的层数（这里是**1**层）
                - **rel_linear**：每层中定义的输入和输出都是**entity_dim**（**100**）的单线性层
                - **e2e_linear**：每层中定义输入是**2 * entity_dim**（200），输出是**entity_dim**（**100**）的单线性层
        - **LSTMInstruction**：**LSTM**指示模块
            - **node_encoder**：定义节点编码器为输入**word_dim**（**300**），隐藏层大小**entity_dim**（**100**）的单向**LSTM**
            - **cq_linear**：定义输入维度是**2 * entity_dim**（**200**），输出维度是**entity_dim**（**100**）单层线性层
            - **ca_linear**：定义输入维度是**entity_dim**（**100**），输出维度是1的单层线性层
            - **num_step**：定义了这里神经网路重复的层数（这里是**1**层）
                - **question_linear**：每层中定义输入和输出都是**entity_dim**（**100**）的单层线性层
- **evaluator_nsm**评估指标初始化：该类存放了可调用的评估函数，后面模型训练，验证和测试时会使用
- **optim_def**优化器初始化：
    - **trainable**：过滤所有不需要求解的参数
    - **optim_student**：采用**Adam**的优化器（**lr=0.001**）

4. 模型训练：

- **evaluate**：训练前先通过一次评估
    - **test_batch_size**：批次大小（**20**）
    - **num_data**：验证集个数（**249**）
    - **batch**：一个批次的数据
        - **candidate_entities**：初始化是全局id最大值+1，这里看作时存放了**subgraph**的**entities**的全局id
        - **query_entities**：初始化全为0，局部id存在的位置标注1
        - **_build_fact_mat**：
            - **batch_heads**：循环每个批次从**kb_adj_mats**中取出并打乱存储的**head_list**（**86156**）
            - **batch_rels**：循环每个批次从**kb_adj_mats**中取出并打乱存储的**rel_list**（**86156**）
            - **batch_tails**：循环每个批次从**kb_adj_mats**中取出并打乱存储的**tail_list**（**86156**）
            - **batch_ids**：循环每个批次，构建每个批次的序列号id，用于存储头，尾和关系的所述批次（**86156**）
            - **fact_ids**：从0到结尾，按1递增的序号（**86156**）
            - **weight_list**：取每个**batch_heads**的元素个数的倒数（**86156**）

        - **head_count**：累计每个不同元素的个数（**16159**）
    - **q_input**：从存放问句单词id的**query_texts**抽取该批次的输入问题（**(20, 2000)**）
    - **seed_dist**：从存放序号个数倒数的分布**seed_distribution**收取该批次的分布（(20, 2000)）
    - **true_batch_id**：None
    - **answer_dists**：从存放回答**entity**的局部id和**subgraph**值相同的位置局部id置为1的**answer_dists**中取出该批次的**answer_dists**（**(20, 2000)**）
    - **answer_lists**：从存放回答编号在实体中的id的**answer_lists**取出该批次的**answer_lists**（**(20, 2000)**）
- **student**：**forward**前向过程
    - **deal_input**：将数据转为**torch.tensor**的格式
        - **query_text, kb_adj_mat, answer_dist, local_entity, query_entities, true_batch_id**：**batch**中的值转成**torch.tensor**的格式
        - **current_dist**：作为可以反向求导的参数**seed_dist**转化得到
        - **query_mask**：**query_text**中非**padding**的地方置为1，其他是0
    - **GNNModel**：
        - **init_reason**：**base**推理模块初始化的**init_reason**
            - **encode_question**：输入问题编码
                - **query_word_emb**：用初始化的**word_embedding**，将问题转化为**300**维度的**embedding**矩阵（**[20, 12, 300]**）
                - **node_encoder**：再经过**LSTM**抽取特征
                    - **lstm_drop**：之前的**word_embedding**经过（**p=0.3**）的**dropout**
                    - **init_hidden**：构建（**(1, batch_size, entity_dim)**：**(1, 20, 10)**）的全零的两个矩阵，作为**h_0**和**c_0**（**hidden state**和**cell state**的初始化值为0）
                        - **query_hidden_emb**：所有隐藏状态的输出作为**LSTM**抽取的**embedding**（**[20, 12, 100]**）
                        - **instruction_hidden**：最后一层**LSTM**隐藏状态输出（**[1, 20, 100]**）
                            - **query_node_emb**：维度转换（最后一个字代表整句话的特征的**embedding**）（`h_n.squeeze(dim=0).unsqueeze(dim=1)`）（**[20, 1, 100]**）
                        - **instruction_mem**：最后一层**LSTM**细胞状态输出（**[1, 20, 100]**）
                        - **query_mask**：**query_text**中非**padding**的地方置为1，其他是0（**[20, 12]**）
                        - **relational_ins**：初始化全为0的指示矩阵（$i^{(0)}$）（**[20, 1, 100]**）
            - **get_instruction**：指示模块操作（不断循环每个指示模块批次（**num_step**），这里是1）
                - **q_i**：将最后一层输出的**query_node_emb**值经过**dropout**和线性变换（**(100, 100)**）（**[20, 1, 100]**）
                - **cq**：先将关系**relational_ins**和**q_i**拼接（**[20, 1, 200]**），再经过线性层（**(200, 100)**）降成**100**（**[20, 1, 100]**）（$q^{(k)} = W^{(k)}[i^{(k-1)}; q] + b^{(k)}$）

        - **ca**：将**cq**（**[20, 1, 100]**）和**query_node_emb**（**[20, 12, 100]**）进行**element-wise**乘积（广播到**12**维），再进行**dropout**，最后加入线性层二分类得到（**[20, 12, 1]**）
        - **attn_weight**：**ca**的**padding**位置加上一个极小值，再接一个**softmax**求出**attention**系数（**[20, 12, 1]**）（$\alpha_j^k = softmax_j(W_\alpha(q^{(k)} \odot h_j) + b_\alpha)$）
        - **relational_ins**：**attention**系数（**attn_weight**）和**query_node_emb**点对点相乘（**element-wise**）再对第一维（问题长度**12**）进行求和更新指示矩阵（**[20, 100]**）（$i^{(k)} = \sum_{j=1}^{l} \alpha_j^{(k)} h_j$）
    - **get_rel_feature**：获取关系特征
        - **rel_features**：**relation_embedding**的权重（`self.relation_embedding.weight`）（**[6102, 200]**）
        - **relation_linear**：经过线性层将**rel_features**转换成100维度（**[6102, 100]**）
    - **get_ent_init**：
        - **local_entity_emb**：给实体全局id编码（**[20, 2000]**）（**[20, 2000, 100]**）
        - **entity_linear**：再经过一层输入输出都是**100**维度的线性层（**[20, 2000, 100]**）
    - **init_reason**：**gnn**的**init_reason**
        - **local_entity_mask**：每行**entities**非0的位置赋值1（**[20, 2000]**）
        - **edge_list**：**kb_adj_mat**（长度为**6**的**list**，每个元素**86156**维，详见**_build_fact_mat**返回的内容）（取出的是**batch_heads, batch_rels, batch_tails, batch_ids, fact_ids, weight_list**）
        - **x2y**中间变量：
            - **fact2head**：**batch_heads, fact_ids**拼接得到（**[2, 86156]**）
            - **fact2tail**：**batch_tails, fact_ids**拼接得到（**[2, 86156]**）
            - **head2fact**：**fact_ids, batch_heads**拼接得到（**[2, 86156]**）
            - **tail2fact**：**fact_ids, batch_tails**拼接得到（**[2, 86156]**）
            - **rel2fact**：（(`batch_rels + batch_ids * num_relation`, `fact_ids`)）**relation**和**fact_ids**拼接得到（**[2, 86156]**）
            - **val_one**：全是1长度是关系对总数的**list**（**[86156]**）
        - **fact2head_mat**：构建（`(batch_size * max_local_entity, num_fact)`：**(40000, 86156)**）**indices**是**fact2head**（**fact_id**对应**head_id**）的稀疏矩阵
        - **head2fact_mat**：构建（`(num_fact, batch_size * max_local_entity)`：**(86156, 40000)**）**indices**是**head2fact**（**head_id**对**fact_id**）的稀疏矩阵
        - **fact2tail_mat**：构建（`(batch_size * max_local_entity, num_fact)`：**(40000, 86156)**）**indices**是**fact2tail**（**fact_id**对应**tail_id**）的稀疏矩阵
        - **tail2fact_mat**：构建（`(num_fact, batch_size * max_local_entity)`：**(86156, 40000)**）**indices**是**tail2fact**（**tail_id**对应**fact_id**）的稀疏矩阵
        - **rel2fact_mat**：构建（`(batch_size * max_local_entity, num_fact)`：**(40000, 86156)**）**indices**是**rel2fact**（**rel_id**对应**fact_id**）的稀疏矩阵
- **GNNReasoning**：推理模块操作

        - **fact_rel**：将这个循环批次（**batch**）的所有的关系从**rel_features**的特征表中取出（**[86156, 100]**）
        - **fact_query**：将这个循环批次（**batch**）的所有的问题从**instruction**（**relational_ins**）的特征表中取出（**[86156, 100]**）
        - **fact_val**：**fact_query**和经过输入输出都是**100**维度的线性层（**rel_linear**）的**fact_rel**元素乘（**element-wise**），最后接一个**ReLU**得到（**[86156, 100]**）（可以看作句子特征**fact_query**和关系特征**fact_rel**进行相乘）（$\sigma(i^{(k)} \odot W_R r)$）
        - **fact_prior**：**head2fact_mat**和**curr_dist**当前的分布矩阵乘，得到所有关系头实体的权重（分布）（**[86156, 1]**）(图神经网络1)
        - **possible_tail**：（**[20, 2000]**）
            - 首先**fact2tail_mat**和**fact_prior**矩阵乘（**[40000, 1]**）(图神经网络2)
            - 将大于极小值的数置为1，并转换格式（**[20, 2000]**）
        - **fact_val**：**fact_prior * fact_val**
        - **neighbor_rep**：**fact2tail_mat**稀疏矩阵和求出的**fact_val**相乘，并形状转换后的结果（**[20, 2000, 100]**）（**fact2tail @ ((head2fact @ curr_dist) * fact_val)**）（$\tilde{e}^{(k)} = \sum_{<e_1,r,e>\in N_e} p_{e_1}^{(k-1)} \cdot m_{<e_1,r,e>}^{(k)}$）(图神经网络3)
    - **local_entity_emb**：更新实体矩阵（$e^{(k)} = W^{(k)}([e^{(k-1)}; \tilde{e}^{(k)}]) + b^{(k)}$）
        - 首先将**local_entity_emb**和**neighbor_rep**进行拼接（第二维）（**[20, 2000, 200]**）
        - 然后经过**dropout**处理
        - 最后通过输入是**200**输出是**100**的线性变化接**ReLU**
    - **current_dist**：更新实体相关分布（$p^{(k)} = softmax(E^{(k)^T} w)$）
        - **score_tp**：
            - 更新后的**local_entity_emb**经过一层线性分类层得到分数（$E^{(k)^T} w$）
            - 然后取**score_tp**的非**mask**值为极小值
        - **current_dist**：对**score_tp**的第一维度求**softmax**更新当前的分布（**[20, 2000]**）
    - **dist_history**：用于记录每批次算得的**current_dist**分布
  - **case_valid**：计算有回答的问题**mask**
    - **answer_number**：对**answer_dist**的第一维度求和，得到回答的个数
    - **case_valid**：**answer_number**大于零的部分设置为**1**（**[20, 1]**）
  - **pred_dist**：取**dist_history**即最后一批次的预测分布
  - **calc_loss_label**：计算损失值
    - **get_loss_new**：计算返回**answer_dist**和**pred_dist**的分布**KL**散度损失值（**[20, 2000]**）
    - **tp_loss**：**get_loss_new**和**case_valid**点对点相乘，过滤没有回答的分布损失值
    - **cur_loss**：所有损失求和除以批次
  - **pred**：**pred_dist**的最后一维度（批次）进行求最大值，得到每一批次回答的序号（**[20]**）
- **train**：训练
  - 类似于验证过程，不过还有**f1**和**hits**等其他评估指标

5. 补充：

- KL散度损失函数：https://www.cnblogs.com/bincoding/p/14362557.html
  - 相比交叉熵只关注最终答案的置信度，KL散度加入了信息熵，还关注了本身的分布 $\sum plog\frac{p}{q} = \sum plogp - plogq$
    - 交叉熵：$-\sum plogq$，中间只要label是0，损失就是零，所以只关注分对的概率
    - 这里采用散度，它的目的是可以实现多分类
  - `nn.KLDivLoss`：https://blog.csdn.net/qq_36533552/article/details/104034759

```
target = torch.FloatTensor([0.1, 0.2, 0.7])
pred = torch.FloatTensor([0.5, 0.2, 0.3])
loss_fun = nn.KLDivLoss(reduction='sum')  # reduction可选 none, sum,
mean, batchmean
loss = loss_fun(pred.log(), target)
print(loss)  # tensor(0.4322)

#上面的计算过程等价于下面
a = (0.1 * np.log(1/5) + 0.2 * np.log(1) + 0.7 * np.log(7/3))
print(a)  # 0.43216
```

```
answer_len = torch.sum(answer_dist, dim=1, keepdim=True)
answer_len[answer_len == 0] = 1.0
answer_prob = answer_dist.div(answer_len)  # 转成分布概率
log_prob = torch.log(pred_dist + 1e-8)
loss = self.kld_loss(log_prob, answer_prob)
```

- 损失：
  - 有负样本时通过**label_valid**过滤掉样本中没有答案的部分（不能乱回答，只计算有答案的损失）（得到的**tp_loss * label_valid**）
  - 最后求损失的平均值
- 预测：只需要问题和相应问题的实体（没有**subgraph**的部分）
- 评估：
  - 单分类：最大的预测值
  - 多分类：F1、hits、precision、recall
  - 输入：
    - anwswer：label标签（entity）
    - candidates：subgraph里的entity
    - seed_entities：entities里的entites
    - prob：预测的结果（概率）
  - 操作：
    - 种子实体（seed）、padding实体、过低的概率不要
      - 选出candidates非padding，概率非特别低的candidates的概率
    - 上一步求出的进行置信度排序（大到小）
    - 从高到低遍历置信度（概率值）—— 支持预测一个或多个实体回答
      - 累加概率值
      - 如果该概率在回答中，正确的数加1
      - 如果累加的概率值大于阈值break