关系抽取报告

张荐科（学号：1120200784）

目录

一、项目概述

1.1 任务目标

本项目的目标是通过深度神经网络模型完成知识图谱构建中的“**关系抽取**”任务。旨在从一段文本中抽取出 $(subject, relation, object)$ 即**（主体，关系，客体）**这样的三元组。

通过学习 2017 年发表于 ACL 的论文中提出的融合拷贝机制的端到端神经网络模型，在复现模型的基础上结合近年来新类型编码器和解码器方法，在 $WEBNLG$ 数据集上进行训练测试，以达到较好的关系抽取效果。项目完成的主要工作包括以下内容:

- 调查关系抽取任务处理难点与当前瓶颈
- 阅读论文总结基本模型架构和常用的编解码方式
- 代码实现
  - （数据集预处理）：由于使用的 $WEBNLG$ 是广泛在工业界应用的成熟数据集，项目中直接使用预处理好的数据集
  - $Pytorch$ 搭建神经网络模型：主要包括编码器和解码器两部分，后者设计关系三元组的解码方式设计，二者基本架构均为双向 LSTM
  - 评估函数：编写针对 RE 任务的 $Precision, Recall$ 和 $F1-measure$ 三种评估函数用于效果的检测评价
  - 迭代训练：设置合适的循环完成神经网络的训练，设计接口用于验证和测试
- 模型调整与比较：尝试不同优化前后的模型在数据关系抽取，进行结果比较，探索更好地模型模块
- 分析总结：整理模型对于关系抽取的提取方法，总结根据问题设计模型的思路和解决瓶颈问题的方法，思考模型可以进一步提升的方向

1.2 任务难点与方法分类

与命名实体识别抽取一个实体相比，关系抽取一次性抽取了一个三元组，包括两个实体和一个关系，这种目标的变化导致 RE 任务不再是一个简单的多分类问题，而是一个需要在分类基础上设计的元组编解码问题。这一基础要求了关系抽取任务的神经网络模型要显著复杂于命名实体识别。

另外，在关系抽取任务中，需要考虑实体-关系分离的问题。$Pipeline$ 方法是将两个任务分离开来分别进行，先进行实体识别，再进行关系分类，这样会存在误差传播的情

况，为了解决这个问题，人们设计各种能够一次性输出三元组的 $Joint$ 方法避免实体识别的误差无法消除的问题。

- $Pipeline$ 方法：先从文本中抽取全部实体 $(e_1, \cdots, e_n)$，然后针对全部可能的实体对 (e_i, e_j) 判定其之间的关系类别。
- $Joint$ 方法：通过修改标注方法和模型结构直接输出文本中包含的 (e_i, r_k, e_j) 三元组。

其中，基于上面两种方法模型又可以根据三元组的输出顺序分为以下两种：

- 一阶段模型：模型一次性输出全部实体、关系组合的模型
- 二阶段模型：先抽取头**实体**$(head)$、**关系**$(relation)$、**尾实体**$(tail)$ 中的一个或两个，然后基于此抽取另外的部分
  - $head \rightarrow relation + tail$
  - $relation \rightarrow head + tail$
  - $head + tail \rightarrow relation$

最后是关系抽取的重叠问题，主要包括以下三类：

- $Single\ Entity\ Overlap\ (SEO)$——单一实体重叠：两个三元组之间有一个实体重叠
- $Entity\ Pair\ Overlap\ (EPO)$——实体对重叠：即一个实体对之间存在着多种关系
- $Normal$——常规：两个实体见仅有一种联系并且均不与其他实体产生联系

关系三元组的表示不能避免语言中不同关系共用同个实体的问题，导致了模型不能逐次选取实体进行简单分类，而应该基于序列连续输出实体和关系，保证关系抽取中可以出现实体或者实体对的重复。

二、任务模型设计

2.1 模型概述

考虑到准确率和本地训练的需要，关系抽取的神经网络模型采用传统的编码器和解码器结构，主体采用融合了“复制”机制的序列到序列 $(Seq-to-Seq)$ 的端到端模型，可

以联合从不同重叠类别的句子中提取关系。（模型主要参考论文：**Extracting Relational Facts by an End-to-End Neural Model with Copy Mechanism [1]**）

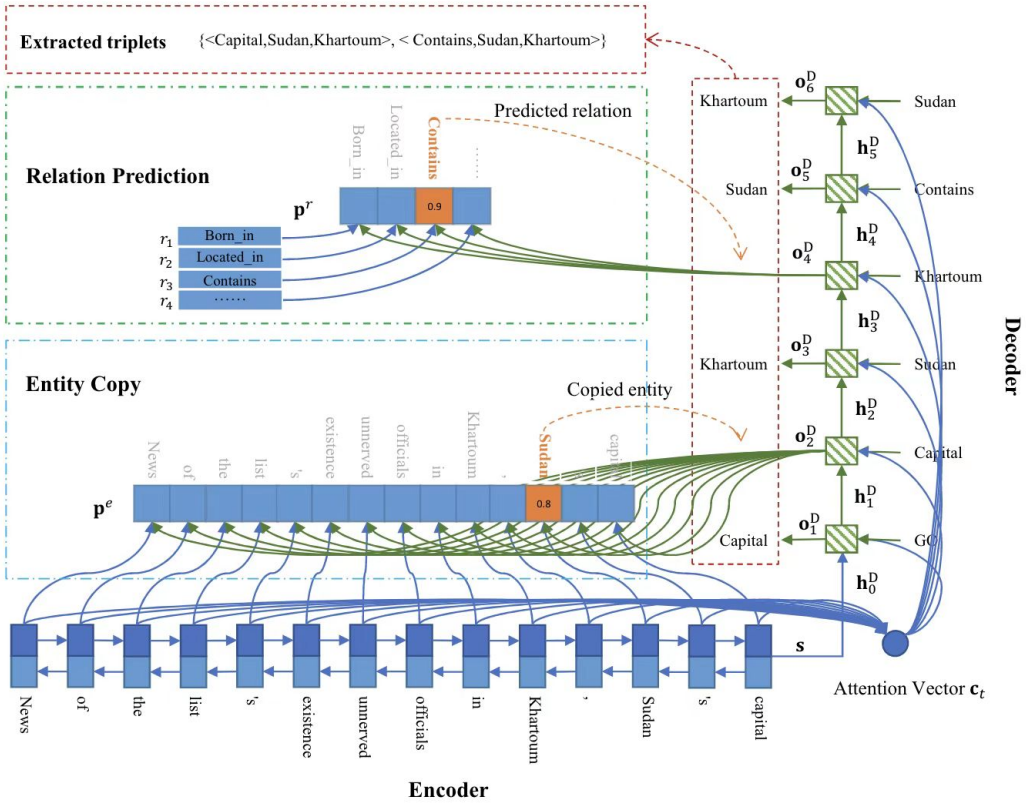

图 2-1 参考论文的关系抽取模型架构

编码器将自然语言句子转换为固定长度的语义向量。然后，解码器读入这个向量并直接生成三元组。这个过程中首先解码器需要生成关系。其次，通过采用**复制机制**，解码器从源语句中复制第一个实体（头实体）。最后，解码器从源语句复制第二个实体（尾部实体）。通过这种方式。另外模型的解码过程中采用两种不同的策略：仅使用一个统一解码器（**单解码器**）来生成所有三元组，或者使用多个分离的解码器（**多解码器**），其中每个解码器生成一个三元组。当一个实体需要参与不同的三元组时，它可以被复制多次，这使得模型具有处理多种重叠类型的句子的能力。模型主要步骤：

1. 编码器：将可变长度的句子编码为固定长度的向量表示
2. 解码器：
    - **One-Decoder-model**：用一个统一的解码器解码所有三元组
    - **Multi-Decoder**：用单独的解码器解码每个三元组

2.2 模型详细阐述

2.2.1 编码器模块

编码器部分是**图 1** 中最下部分蓝色方块组成的双向 LSTM 层，目的是将可变长度的句子编码为固定长度的向量加以表示。具体操作包括以下两个部分：

1. 编码一个句子 $s=[w_1,\cdots,w_n]$，其中 W_t 代表第 t 个单词，n 是原句子长度，首先将其转换为一个矩阵 $X=[x_1,\cdots,x_n]$，其中 x_t 是句子 s 中第 t 个单词的嵌入向量。即对一个句子 s，生成该句子的词嵌入矩阵：

   $$s=[News,of,the,list,\cdots,Sudan,'s,Capital]\Rightarrow X=[x_1,x_2,\cdots,x_n]$$

2. 将 X 按序列输入双向 RNN，处理生成输出和隐藏状态层：

$$\text{输出层：}o_t^E \qquad \text{隐藏层：}h_t^E$$

$$o_t^E,h_t^E=f(x,h_{t-1}^E)$$

使用双向 RNN，因此结果是一个双向序列$\{\vec{o}_1,\vec{o}_2,\cdots\vec{o}_n\},\{\overleftarrow{o_n},o_{n-1}^{\leftarrow},\cdots,\overleftarrow{o_1}\}$

将它们联合起来作为最终输出：$\{o_1,o_2,\cdots,o_n\}$, 其中$o_t=(\vec{o}_t;o_{n+1-t}^{\leftarrow})$

将隐藏层的末端联合作为$decoder$的h_0层：$s=h_0^D=(\vec{h}_n;\overleftarrow{h_n})$

实际实现中为了方便用于解码器的输入，采用输出的两个序列元组取均值作为最终输出：

$$o_t=(\vec{o}_t+o_{n+1-t}^{\leftarrow})/2\;;s=(\vec{h}_n+\overleftarrow{h_n})/2$$

2.2.2 解码器

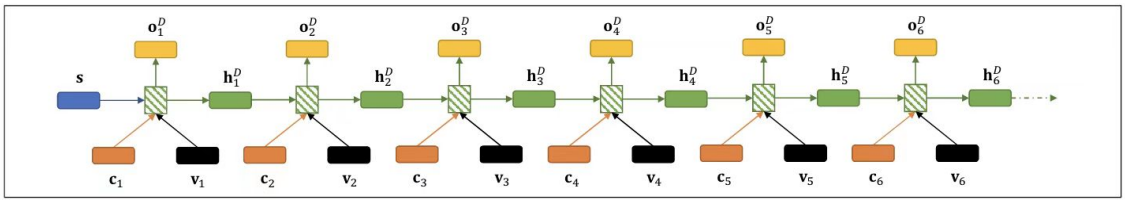

图 2-2 单解码器基本架构

解码器用来生成实体-关系三元组，生成顺序为：关系 → 第一个实体 → 第二个实体。解码器使用了时序模型架构，但是序列的每次计算步骤与通常的 RNN 并不完全相同，为了兼顾编码器的实体对编码和关系的关联，它采用了更加复杂的时序输入方式，其整体计算方法如下所述：

$$\text{输出层：}\ o_t^D \qquad \text{隐藏层：}\ h_t^D$$

$$o_t^D, h_t^D = g(u_t, h_{t-1}^D)$$

$$u_t = [v_t; c_t] \cdot W^u$$

$$\text{其中} c_t \text{是} attention - vector$$

$$v_t \text{是经过复制机制后产生的一个实体嵌入或关系向量}$$

c_t和v_t 的计算需要 t-1 的隐藏层和编码器的实体编码共同计算完成，具体过程如下：

- c_t **的计算**：根据 Encoder 的编码生成 Attention Vector

$$\begin{cases} 1. \quad \beta_i = selu(h_{t-1}^D; o_i^E) \\ 2. \quad \alpha = softmax(\beta) \\ 3. \quad c_t = \sum_{i=1}^{n} \alpha \times o_i^E \end{cases}$$

- v_t **的计算**：v_t 的生成是又 $t-1$ 时刻 o_t^D 决定的，根据不同的 t，o_t^D 的输出会进入两个不同的层，将输出作为用来计算 $t+1$ 次 v_{t+1}。对于 t 时刻 o_t^D 的输出:

$$\begin{cases} \text{使用} o_t^D \text{复制一个实体作为三元组第一个实体，} t = 2, 5, 8, \cdots \\ \text{使用} o_t^D \text{复制一个实体作为三元组第二个实体，} t = 3, 6, 9, \cdots \\ \text{使用} o_t^D \text{预测一个新的关系，} t = 1, 4, 7, \cdots \end{cases}$$

- **关系预测层** (t%3 = 1) 主要包括以下三个计算步骤：

(1) 计算以 o_t^D 为输入的 m 个关系的置信向量 q^r 以及终止表示符 q^{NA}：

$q^r = [q_1^r, q_2^r, \cdots, q_m^r]$，其中$q_r = selu(o_t^D \cdot W^r + b^r)$；$q^{NA} = selu(o_t^D \cdot W^{NA} + b^{NA})$

(2) 连接并取 softmax 得到概率向量 p^r：

$$p^r = [p_1^r, p_2^r, \cdots, p_{m+1}^r] : p^r = softmax([q^r; q^{NA}])$$

(3) 选择概率最大的并用对应的词嵌入向量作为 $t+1$ 输入的 v_{t+1}

- **第一个实体复制层**：$(t\%3=2)$

使用 linear+softmax 选择概率最大的作为实体 1：假设共 n 个词汇

令$q^e=[q_1^e,q_2^e,\cdots,q_m^e]$, 其中$q_i^e=selu([o_t^D;o_i^E]\cdot w^e)$

选择概率最大的并用对应的词嵌入向量作为 t+1 输入的v_{t+1}

- **第二个实体复制层**：$(t\%3=0)$

由于关系三元组中两个实体不能相同，所以在复制第二个实体的时候要把第一个实体对应的嵌入向量用 0 覆盖，避免选择了相同词语，除此之外，复制层与第一个实体复制层完全相同

2.2.3 多解码器

多解码器模型是提出的单解码器模型的扩展。主要区别在于解码三元组时，多解码器模型使用多个分开的解码器对三元组进行解码。

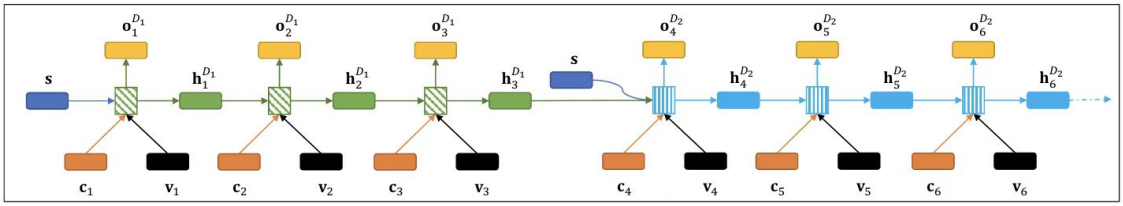

图 2-3 多解码器基本架构

图2-3中展示了两个解码器（绿色阴影框和蓝色阴影框），每个解码器生成一个关系三元组。本质上就是多个单解码器的串联（但是每个解码器只执行 3 次）。需要注意的是在除第一个解码器外的其他解码器的输入 h_0 不再是源句 s：

$$\hat{h}_{t-1}^{D_i}=\begin{cases}s, & i=1\\ \frac{1}{2}(s+h_{t-1}^{D_{i-1}}), & i>1\end{cases}$$

三、关系抽取实现

3.1 代码实现文件说明

代码实现中尝试使用更模块化的代码编写方式，尽量将关系抽取任务中设计的不同模块、参数配置、数据导入、模型存储等等部分合理地封装，方便后续进行调参或者扩展创新。代码文件中主要包括以下部分：

(1) 数据文件：*data* 文件夹

实验中采用的 $WEBNLG$ 数据集是提前预处理好的，包括以 $json$ 文件存储的五个切分好的样本数据文件，分别为：实体词-ID 字典；关系词-ID 字典；训练集、验集证、测试集的语句集及关系三元组标签向量。

(2) 训练参数保存：$saved\_model$ 文件夹

训练结束时将所得全部参数存储在该子文件夹下的 pkl 数据文件中，用于 test 模式下的关系抽取预测。

(3) 配置文件及配置参数提取：$config.json$ 和 $config.py$ 文件

为了更方便的更改训练配置，实验中将常用且可更改的训练配置参数统一在 $Json$ 文件中管理，同时在 $config.py$ 文件中设计了 $Config$ 类用于读取 $Json$ 文件中配置参数，并统一管理了全部涉及的超参数、文件路径等各类参数。调试时有需要可以在 $config$ 类中调整任意想要的参数。在模型超参数传递时可以直接传递一个 $Config$ 类，极大方便了参数管理和调整。

(4) 数据导入及预处理：$data\_preprocess.py$

该部分将已经编码后的 $WEBNLG$ 数据集进一步切分，生成可用于模型输入的数据张量，并将数据集中的三元组标签进行切分堆叠，与输入向量一一对应，方便误差函数的计算。

(5) 模型建立：$model.py$

使用 $pytorch$ 的神经网络库实现上文提到的 $Seq-Seq$ 关系抽取模型，基本架构中的 $LSTM$ 可以根据参数更改为 GRU 层。

(6) 评价函数：$evaluation.py$ 及 $data\_preprocess.py$

将模型提取的关系三元组与数据集中的正确标签进行格式转换并比较，计算对应的 precision、recall 和 F1-measure 值用于评价预测结果。

(7) 训练与测试：$train\_test.py$

完成模型在数据集上的正反向传播，存储训练出的模型参数，将参数再次导入后可将模型用于测试集进行关系抽取。

(8) 运行效果展示：$ResultShow.ipynb$

在 jupyternotebook 中展示报告中的中间结果、训练过程以及后续对比实验的数据结果。

3.2 数据集处理格式说明

在 data_preprocess.py 中封装了 Prepare 类和 Data 类，前者用于对数据集进行预处理 (prepare.process 函数)，后者用于将 Prepare 处理好的数据进行转换封装为自定义类

型 InputData，并为其生成相应的迭代器（使用 Data.next_batch 函数生成下一个小批量数据）。为了更好的理解后续模型的处理及各个参数矩阵形状，这里对输入数据 Data 的主要张量形状和意义进行说明。

表 3-1 输入数据说明

名称	意义	形状
input_sentence_length	输入语句真实长度列表	(句子数量，)
input_sentence_append_eos	输入语句编码张量	(句子数量，最大句子长度)
all_triples_id	三元组标签张量	(句子数量，3× 每句最多三元组数量)

任务中使用的 WebNLG 数据集经过处理后的数据张量形状如**下图**，数据集大小较小，可以用于本地进行模型的训练。预处理中将所有句子长度填充为 80，并且增加一列用作句子终止符。将同一句子中的多个三元组堆叠成一个向量，若句子中的三元组数量不及最大的 5 个，则不足的部分全部使用 0 填充，因此每个句子的三元组标签长度均为 $3 \times 5 = 15$。

```
-----train-----
Input sentence size:  (5019, 81)
Relation Triple label size:  (5019, 15)
Total num of triples:  35328
-----valid-----
Input sentence size:  (500, 81)
Relation Triple label size:  (500, 15)
Total num of triples:  3354
-----test-----
Input sentence size:  (703, 81)
Relation Triple label size:  (703, 15)
Total num of triples:  4773
```

图 3-1 数据集大小截图

3.3 模型搭建

根据2.2节提到的关系抽取模型，针对每一模块利用集成封装关系构建了整体的模型，实现中模型类的关系架构的设计如**下图**所示。其中 Encoder 是编码器类、Decoder 是解码器基类，根据需求派生出 One-Decoder 类和 Multi-Decoder 类。

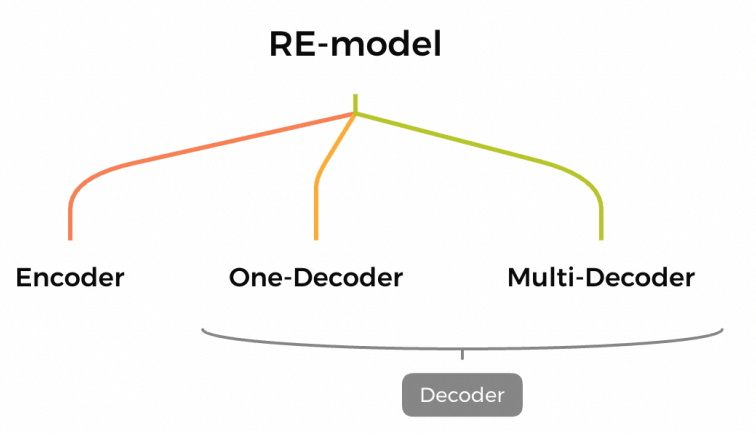

图 3-2 模型实现架构

(1)*Encoder* 部分中按照模型将输入语句编码向量转化为嵌入向量并通过双向 RNN（实现中提供 LSTM 和 GRU 两种选择）输出词编码 o^E 和末态隐藏层，前者用于实体拷贝输入和解码器组合输入，后者作为解码器的第一个隐藏层输入。其中涉及的主要神经网络模块包括以下几部分：

- words_embedding：$nn.Embedding(words\_num, embedded\_dim)$

  词向量嵌入，将输入语句的每个词语编号按照字典对应的方式映射为 $embedded\_dim$ 大小的特征向量，反向传播中需要进行参数更新

- Encoder.RNN：

  $nn.LSTM(embedded\_dim, hidden\_dim, bidirectional = True)$

  编码器的双向 RNN 单元层，需要将双向 RNN 的两次输出进行对应位置拼接作为总体的词编码输出，末态的隐藏层作为解码器的首个隐藏层，体现句子整体的特征

(2)*Decoder* 基类中完成了2.2.2节中的注意力机制的嵌入向量计算，提供了解码器 RNN 的单步计算过程，将输出 o^D 作为实体复制/关系预测层的输入并给出当前预测的三元组元素和对应的得分 logits。其中设计的模块比较繁杂，主要包含以下部分:

- Relation_embedding：$nn.Embedding(relation\_num, embedded\_dim)$

  关系词向量嵌入，将关系编号按照字典对应的方式映射为特征向量，在关系预测时作为输入

- Decoder.RNN：$nn.LSTM(embedded\_dim, hidden\_dim)$

  解码器的单向 RNN 单元层，需要将注意力机制产生的词编码与上一层解码结果拼接作为输入，得到 o^D，再作为实体/关系预测分类器的输入进行三元组抽取。

- Attention_layer：将编码器的输出 o^E 使用线性层转化为权重矩阵，再用此权重矩阵与 o^E 相乘作为解码器序列的输入，实现自注意力机制的编码，体现编码序列和输出三元组的语义性。

$$[decoder\_hidden, encoder\_output] \rightarrow nn.Linear(2 \times hidden\_dim, 1) \Rightarrow atten\_weight$$

$$[encoder\_output] \times atten\_weight \Rightarrow decoder\_input$$

- 实体复制：使用拼接和线性层的 softmax 分类选择下一个实体:

$$[decoder\_output, encoder\_outputs] \rightarrow$$

$$selu + nn.Linear(2 \times hidden\_dim, 100) + selu + nn.Linear(100, 1) + softmax$$

- 关系预测：直接使用线性层和 softmax 分类选择最优关系:

$$[decoder\_output] \rightarrow nn.Linear(hidden\_dim, relation\_num) + softmax$$

(3)$One/Multi - Decoder$ 在 Decoder 的基础上完成整个解码器 RNN 序列前向传播，区别在于 RNN 中是否使用统一的权重。One-Decoder 中只采用一个 RNN 单元，即每次抽取实体的 RNN 权重是共享的，而 Multi-Decoder 则在每次输出关系三元组后更换一个 RNN 单元，不同单元间参数独立不共享。具体实现只需依照2.2节中给出的设计编写即可。

(4)$RE - model$ 完成从句子序列到关系三元组抽取的全过程，按照配置参数将 Encoder 和 Decoder 输入输出进行拼接，得到最终的关系三元组序列每个元素的得分 logits。

上述实现细节可在 model.py 文件中查看，其中给出了详细的注释和输入输出说明

3.4 评价函数

关系三元组抽取效果仍然采用自然语言处理领域中常用的三个评价指标，即查准率 ($precision$)、查全率 ($Recall$)、综合评价指标 ($F1 - measure$)。

$$Precision = \frac{TP}{TP + FP}; \quad Recall = \frac{TP}{TP + FN};$$
$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

其中 TP 表示检测出来正确的标签,FP 表示检测出来但错误的标签,

FN 表示样本中未被检测出的标签

为了更加全面的评价一个三元组的标签，需要设计多个标签对比的函数。在 evaluation.py 中实现了五种评价指标的计算函数，分别是针对实体、关系、非重叠三元组、实体重叠三元组、关系重叠三元组。为了实现这些不同组合的标签比较，任务中同时实现

了很多三元组变换函数和抽取的预测标签和正确标签张量的变换函数，在 evaluation.py 和 *data_preprocess.py* 中进行了实现。

另外，为了评价模型对于关系交叠的处理效果，需要对预测的三元组按着交叠形式进行分类：第一类记为 **Normal**，即不存在与这个三元组有元素重叠的其他三元组；第二类记为 **Multi-Label**，即两个实体有多个关系与之组成关系三元组；第三类 **Over-Lapping** 表示其他三元组中存在只有一个实体与其相同的三元组，并且不为第二类 Multi-Label 的三元组。

3.5 训练及测试

在 *train_test.py* 中实现了训练器 *SupervisedTrainer* 和测试器 *Evaluator*，分别用于模型在训练集上的训练和测试集的预测。

训练器 Trainer 的实现逻辑就是将输入数据迭代器的小批量数据依次读入并使用模型前向传播，计算三元组序列所有标签的损失之和后反向传播并依次更新网络参数。其中主要进行的步骤如下代码注释所示：

```
def train_step(self, batch: data_preprocess.InputData) -> torch.Tensor:
        # 1. 对输入的额外张量处理操作
        # 2. 使用模型计算得分张量
        # 3. 统计一个语句中全部三元组的抽取损失
        # 4. 反向传播更新网络
        return loss
def train(self) -> None:
    for epoch in range(1, self.epoch_number + 1):
        for step in range(self.data.batch_number):
            # 1. 读取下一批量数据
            loss = self.train_step(batch)
        # 2. 保存模型到指定文件夹
        torch.save(self.seq2seq.state_dict(), model_path)
```

这里将每个周期训练的结果存储在文件中，方便用于测试集预测或者迁移学习。同时，为了能够在每个周期中观察模型在 valid 验证集上的效果，train 中可以额外加入一个 evaluator 类，导入 valid 数据进行预测并将结果打印输出，便于观察训练过程变化以优化网络或查找错误。

测试器 Evaluator 与 Trainer 的主要区别在于需要加载已经训练好的模型并且前向传

播后不需要进行反向传播，同时使用 evaluation.py 提供的各类评价函数进行结果统计值的计算。同时，为了能够将输入数据张量、预测关系三元组张量直观表示，在测试其中还实现了可以将编码解码成字符并保存到文件中 test_visualize 函数，其中 visualize 部分在 evaluation.py 中实现，其作用是将三元组张量还原为字符形式并按照三元组的交叠类型存储到三个文件中。这里由于迭代器读入的测试集也是按照批量输出的，因此也按照 step 的方式封装每一步的操作，具体伪代码给出如下：

```
class Evaluator(object): # 测试器,将现有训练好的模型用于测试集进行预测
    def __init__(self, config: config.Config, mode: str, device:
        torch.device) -> None:
        # 初始化参数
    def load_model(self) -> None:
        """加载已经训练好的模型参数"""
        self.seq2seq.load_state_dict(torch.load(model_path))
    def test_step(self, batch: data_preprocess.InputData)
          -> Tuple[torch.Tensor, torch.Tensor]:
        # 读入迭代器给出的小批量数据并使用模型进行计算
        # 返回得分张量和标签
        return pred_id_list, pred_logits_list
    def test(self) -> Tuple[float, float, float]:
        """读入迭代器给出的小批量数据并使用模型进行计算
           计算按三元组为整体的三个评价指标"""
        for batch_i in range(self.data.batch_number):
            batch_data = self.data.next_batch(is_random=False)
            pred_action_list, pred_logits_list = self.test_step(batch_data)
            # 按批量读入数据并进行计算
        # 统计并返回三个评价指标
        return f1, precision, recall
    def test_visualize(self,filename:Tuple[str,str,str]) -> Tuple[float,
        float, float]:
        """测试集预测并将结果还原为字符文件存储到filename的三个文件中"""
        # 1. 预测并声称预测predicts和正确gold三元组
        # 2. 还原并解析出三元组的字符，将文件进行存储
        return f1, precision, recall
```

四、结果展示与讨论

模型在 WebNLG 的运行参数及配置如下表所示，这里采用多解码器 +LSTM 的模型设置。参数说明和训练过程分别如表4-2和图4-1。将模型重新加载后进行测试，在三个数据集上分别查看抽取效果，结果如表4-1所示：

表 4-1 模型准确度评价

数据集	F1-measure	Precision	Recall
训练集	0.984	0.987	0.980
验证集	0.473	0.494	0.454
测试集	0.480	0.494	0.467

表 4-2 输入数据说明

参数名称	参数意义	值
dataset_name	数据集名称	webnlg
max_sentence_length	单句向量长度	80
Word_number	词库字典大小	5928
Relation_number	关系词库字典大小	247
triple_number	每句抽取最多三元组数量	5
cell_name	编、解码器 rnn 单元类型	lstm
Decoder_type	解码器策略	muli
Embedding_dim	词嵌入向量长度	100
Encoder_hidden_size	编码器 rnn 隐藏层大小	1000
Decoder_hidden_size	解码器 rnn 隐藏层大小	1000
learn_rate	学习率	0.001
batch_size	批量大小	32
Epoch_number	迭代周期数	40

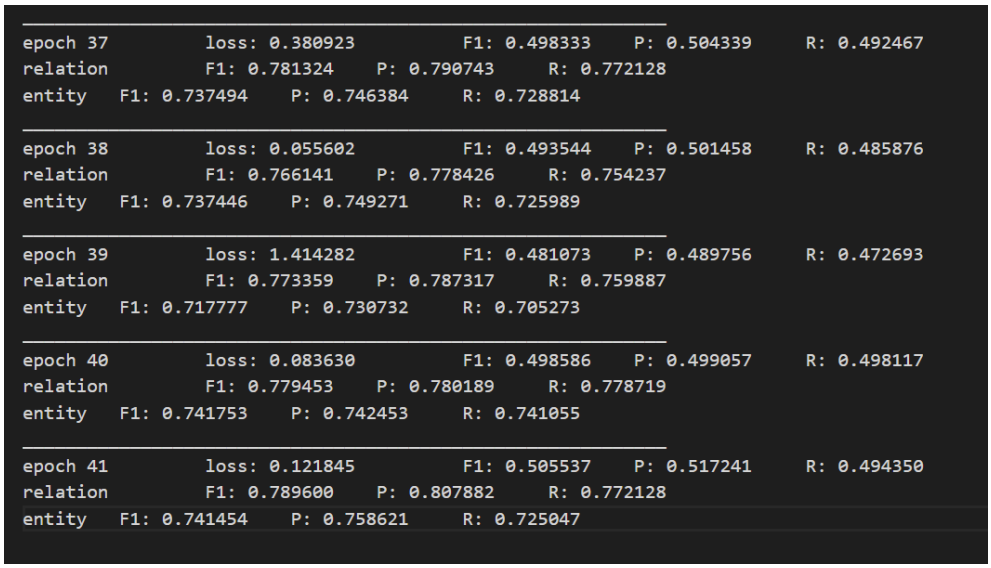

图 4-1 训练结果截图

下文展示了模型预测的可视化结果（预测结果存储在了 normal_triple.txt 等三个文件中，这里只截取一小部分）：

Alan Shepard, who was born in New Hampshire on November 18th, 1923,

graduated with a M.A. from NWC in 1957 and retired on August 1st, 1974.

Gold: [b'Shepard', b'Hampshire', b'birthPlace']

Predict: [b'Shepard', b'Hampshire', b'birthPlace']

根据图4-1和表4-1，可以发现模型在验证集和训练集的抽取效果显著差于训练集，训练集训练结果的三个指标已经接近 100%，这说明模型存在过拟合问题，训练的迁移性很差，不能在新的数据上取得较好的效果。为了减小过拟合的程度，一方面可以压缩使用更轻量的神经网络单元如 gru 并加入 dropout 层，另一方面在 rnn 的损失函数中加入正则化项，避免模型参数过度膨胀。在第5节尝试了更改模型架构并调整参数以达到更好的抽取效果。

五、多模型对比试验

这一部分中通过调整模型及优化策略尝试得到更好的关系抽取结果。为了对比不同模型的效果，主要采取控制变量法的对比实验，除了给定的对比参数外，如无特殊说明其他参数与表4-2中的参数相同。同样，所有运行结果代码在 ResultShow.ipynb 中给出。

5.1 解码策略和 RNN 单元选择

模型主要需要解决的问题是三元组准确率和过拟合的问题。在原模型中提到的多解码器策略就是为了使模型能够更好地识别存在交叠的关系三元组，通过实验验证这种解码策略设计的有效性。同时，模型中采用的 RNN 单元也可以进行更换，通常 GRU 和 LSTM 都能够完成序列的识别，但是 GRU 的模型尺度更小，运算更快同时参数更少，这有助于缓解第4节中提到的模型过拟合问题。尝试 4 种模型的不同组合，可以得到如图5-1的训练结果：

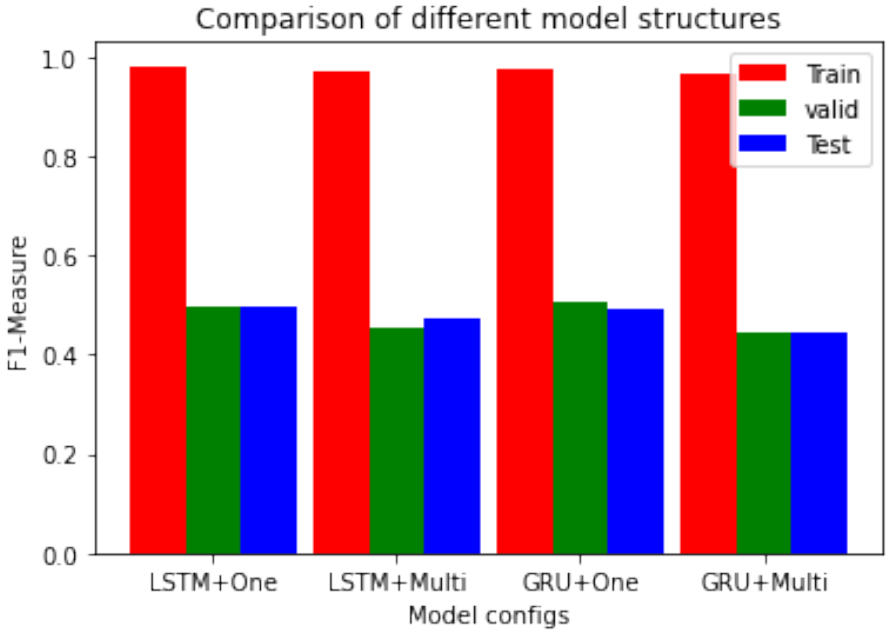

图 5-1 不同模型结果对比

从上图中可以看到单解码器在总三元组的预测准确率上是稍微好于多解码器的，这是因为多解码器之间参数不共享，当训练次数相同时 “学到的” 参数少于共享参数的单解码器。这种多解码的设计实际上是为了解码存在实体交叠的三元组的，因此在对比时还应该考虑两种解码在不同种类关系三元组的识别能力。

表 5-1 两种解码模型在不同重叠种类三元组的准确度

	LSTM+One	LSTM+Multi	GRU+One	GRU+Multi
Normal F1	0.599	0.582	0.610	0.586
Multi-label F1	0.530	0.565	0.460	0.552
Over-lapping F1	0.465	0.463	0.434	0.456

上述实验对比中加入 Multi-Decoder 后模型对于 Multi-label 和 overlapping 的预测结

果都要显著高于同类型的 One-Decoder 模型。这一结果验证了多解码器解码策略对于解决三元组交叠问题的有效性。综合上述结果，采用 LSTM+Multi-Decoder 可以得到相对最好的综合抽取效果，在算力或时间有限的条件下，可以采用 GRU 替代 LSTM，一定程度压缩模型的同时也能保证预测准确率较高。

5.2 正则化的设置

可以看到模型的过拟合比较严重，一方面模型比较庞大，训练集只有 5000，因此模型很容易出现过拟合的现象。正则化的方法能够有效避免参数权重过大膨胀以拟合训练集的问题，从而使模型在迁移到新的数据上时仍能保持较高的准确率。

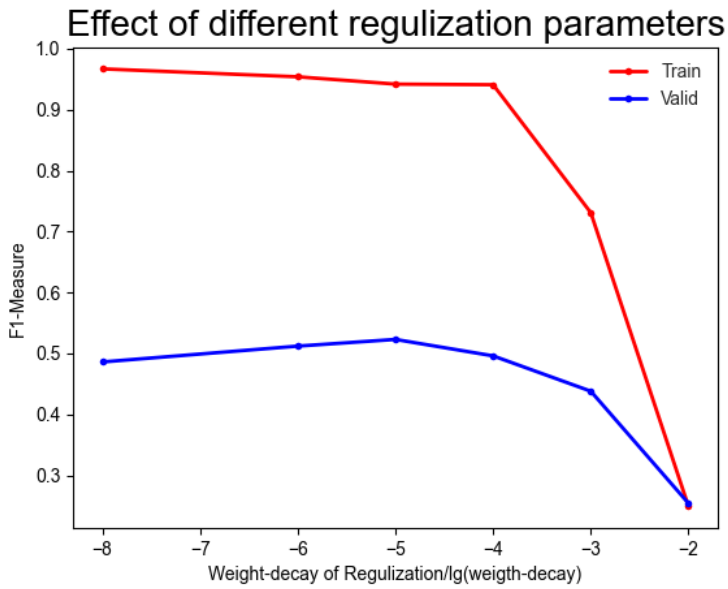

图 5-2 正则化对训练结果的影响

正则化的思想就是在损失函数上增加所有线性层权重矩阵的 L2 范数，即 $Loss = Loss + \lambda \sum_i ||W_i||_2$ 从而对权重过大起到“惩罚作用”。为了确定最好的正则化系数 λ，进行参数比较实验进行搜索，不同系数下的训练结果如上图所示。

根据图5-2，可以看到通过设置正则化的确可以在一定程度上缓解模型的过拟合问题，并提高模型在验证集上的表现。过高的正则化系数会使训练集和验证集 F1 差距进一步缩小，但是模型的拟合效果也会下降，整体的 F1 大幅度减小，因此需要针对数据和模型选取合适正则化系数以达到最好的抽取效果。针对实验中的模型，可以看到正则化有效果但是并不明显，验证集上的准确率始终不能达到训练集，这说明模型具有一些弊端，学习到的权重不能够很好地表示关系和实体间真正的联系，因而迁移性比较差，在更换数据集后预测效果会显著下降，需要通过改进模型结构从而减小正则化无法解决

的过拟合问题。

六、总结

在这项 NLP 任务中，我完成了完整的关系抽取任务的流程。任务总体可以概括为预处理-编码-解码-设计损失函数-评估方法-数据训练六个过程。它与命名实体识别的多分类任务有所区别，属于一种序列到序列的模型。因此任务中的编码器和解码器都采用了序列模型 RNN 中 LSTM 或者 GRU。在建立模型骨架后还需要设计很多处理细节，由于个人能力限制，这里主要参考了 **Extracting Relational Facts by an End-to-End Neural Model with Copy Mechanism** [1] 的论文，主要利用了编码的 self-attention 机制和新设计的实体复制机制，这种机制实际上就是将序列的输出投入到一个多分类器中后再嵌入作为下一个序列的输入。将论文内容实现后，我又根据现有结果特点进行了新的模块比较和设计，尝试缓解了模型明显的过拟合问题，得到了比较好的实验结果并发现了模型本身存在的一些问题。

当然，本文中采用模型在结果上看效果并不足够优秀，我认为其原因是在模型设计中没有足够考虑关系和实体的联系特点，仅仅通过神经网络本身可能并不能 “学到” 这些语义背后的联系：

(1) **暴露偏差**：训练过程虽然采用端到端训练方式，但是本质上是一种两阶段抽取过程，即实体的提取和关系的提取是分开进行的，因此模型上游计算出的结果误差会因为序列输入输出依次累积。这可能导致输出结果的分布偏差和真正的数据标签相差很远，模型为了弥补这一问题只能通过对数据进行过度拟合减小这种损失，间接导致忽略了三元组之间的自身关联。

(2) 编码器部分没有将实体对和关系纳入考虑范围，导致模型前半部分学习能力较弱。在 Encoder 中，只使用了词嵌入后的 RNN 作为编码层，之后的 attention 也仅仅是针对句子向量本身设计的。换言之，这种编码层用于 NER 或者其他序列提取模型中都是一样的，忽略了关系三元组本身对于词向量编码的要求。

这些问题在 2018 年来逐渐得到了解决，比较典型的如同为两阶段机制的 TPlinker/GPlinker 模型以及一系列 Joint 类和采用 bert 预训练、transformer 模块的新型关系抽取模型，将原有准确度提升了近一倍至 90%。但是不可否认的是任务中参考的融合拷贝机制的端到端模型确实给予了人们解决三元组重叠问题的思路，此后的很多模型都是基于此设计的。

从我的角度来看，目前关系抽取模型已经很完善了，我们复现并对前人的模型进

行分析创新不仅可以能够提高对 NLP 领域数据预处理和复杂神经网络模型的实现能力，同时更深入地理解了知识工程任务构建方法。在之后出现新的问题时，能够独立地分析问题并设计合适的神经网络模型进行实现。

参考文献

[1] Zeng X, Zeng D, He S, et al. Extracting relational facts by an end-to-end neural model with copy mechanism[C]//Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). 2018: 506-514.

[2] 辛欣. 知识工程讲义. 北京理工大学知识工程