**《深度学习》实验报告**

唐小卉 10215501437 数据科学与大数据技术 数据科学与工程学院

1. **实验环境**

Python 3.9

特殊代码库：

torch (PyTorch): 用于深度学习模型的构建和训练。

tqdm: 用于显示循环的进度条。

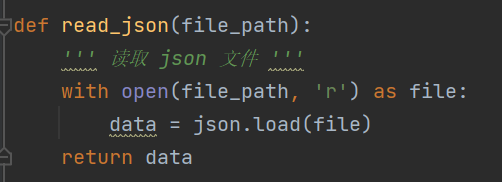
numpy: 用于数值计算。

json: 用于读取和写入 JSON 文件。

os: 用于执行压缩操作。

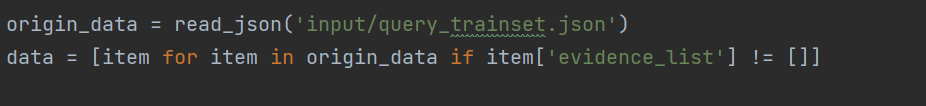
1. **实验过程**
2. 数据预处理部分
   1. 读取JSON文件：

使用 read\_json 函数读取训练集、测试集和文档集数据。



0

* 1. 筛选数据：过滤掉 evidence\_list 为空的训练数据。（上课时也说明过有少量数据是空的）



1. 模型构建

首先说一下我选择模型的逻辑，从观察数据集能看出，数据集并不是传统意义上的语言数据集，而是经过了一定抽象化后的序列数据，那么对于序列数据来说比较好的模型有LSTM和GRU，我最开始使用了GRU来进行本次实验，但是GRU对输入序列要求长度一致，但是我们的数据集达不到这个要求，而且我个人认为数据集本身并不算优秀，如果强行对齐或者删减会导致效果更差，所以最后用了LSTM，然后参考了之前的注意力机制实验加入了一个简单的Attention层。

Rerank方面比较简单，我本来考虑使用BM25或者TF-IDF，但是在实践操作的时候我发现他们俩其实不适用于本次Tokenizer之后的数据集，所以最终我选择了Jaccard相似度和余弦相似度结合来算（其实欧氏距离等计算方式差距也是不大的，我都有试验过，基本没有什么差别）

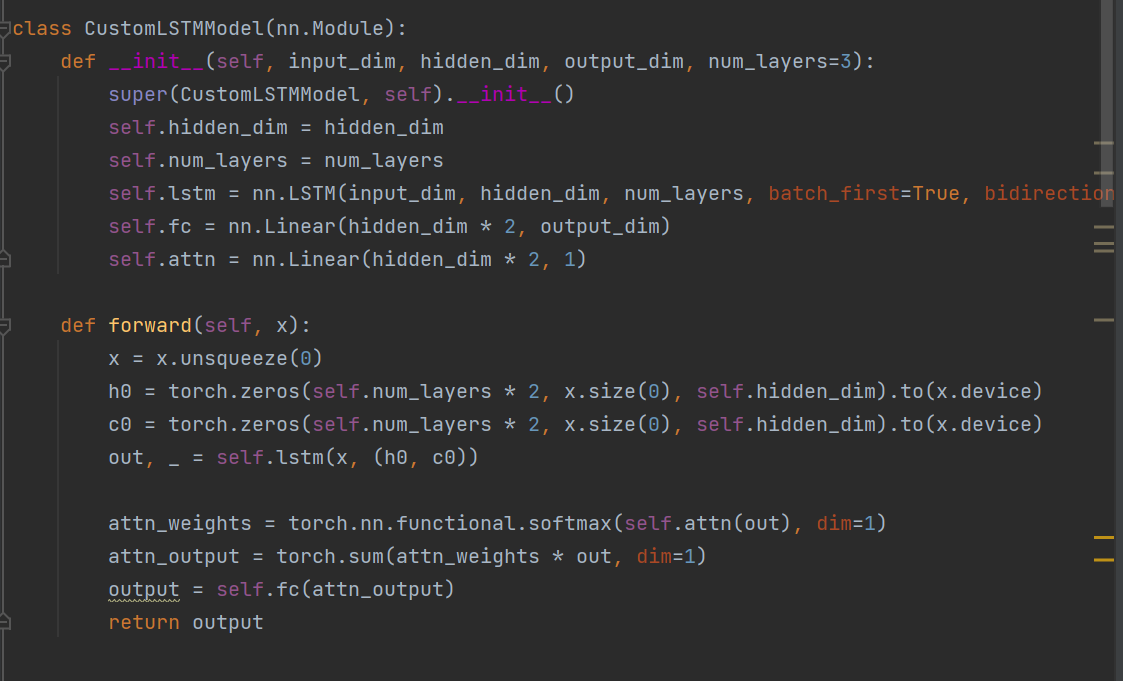
1. 模型架构：

输入层：输入维度设置为1024

LSTM层：使用三层双向 LSTM 层，隐藏层维度为 128

注意力机制：使用线性层进行注意力权重的计算。（其实也尝试了更复杂的Attention，但是效果不好，后续就简化为之前实验里的线性层了）

全连接层：将注意力加权后的输出通过全连接层进行映射，输出维度为 1024。

尝试过添加dropout，但效果一般，后期去掉了。

以下是对这个模型的详细解释：

1. 初始化

* input\_dim: 输入特征的维度。
* hidden\_dim: LSTM 隐藏层的维度。
* output\_dim: 模型输出的维度。
* num\_layers: LSTM 层数，默认为 3。

在初始化方法中，定义了模型的各个部分：

* self.lstm: 一个双向的 LSTM 网络，输入维度为 input\_dim，隐藏层维度为 hidden\_dim，层数为 num\_layers。因为是双向 LSTM，所以 LSTM 的输出维度会是 `hidden\_dim \* 2`。
* self.fc: 一个全连接层，将hidden\_dim \* 2维度的特征映射到 output\_dim维度。
* self.attn: 一个线性层，用于计算注意力权重。

2. 前向传播

- 输入x: x形状(batch\_size, seq\_length, input\_dim)。

在forward方法中，完成了以下步骤：

1：扩展输入维度

*x = x.unsqueeze(0)*

将 x的第一个维度添加一维，使其形状从 (seq\_length, input\_dim)变为 (1, seq\_length, input\_dim)，即模拟一个批次大小为 1 的情况。

2：初始化 LSTM 隐状态和细胞状态

*h0 = torch.zeros(self.num\_layers \* 2, x.size(0), self.hidden\_dim).to(x.device)*

*c0 = torch.zeros(self.num\_layers \* 2, x.size(0), self.hidden\_dim).to(x.device)*

初始化 LSTM 的初始隐状态 h0 和细胞状态 c0，维度为(num\_layers \* 2, batch\_size, hidden\_dim)。因为是双向 LSTM，所以乘以 2。

3：通过 LSTM 层

将输入x和状态(h0, c0)传入 LSTM 层，得到输出 out，形状为 (batch\_size, seq\_length, hidden\_dim \* 2)。

4：计算注意力权重

*attn\_weights = torch.nn.functional.softmax(self.attn(out), dim=1)*

通过线性层 self.attn计算每个时间步的注意力权重，然后使用 softmax 函数进行归一化，确保权重和为 1。输出attn\_weights形状为(batch\_size, seq\_length, 1)。

5：应用注意力机制

*attn\_output = torch.sum(attn\_weights \* out, dim=1)*

计算加权后的 LSTM 输出 attn\_output，形状为 (batch\_size, hidden\_dim \* 2)。这是通过对时间步维度 seq\_length 进行加权求和实现的。

6：通过全连接层

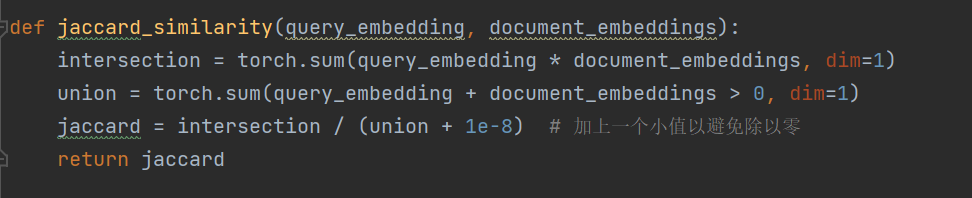
*output = self.fc(attn\_output)*

将注意力机制后的输出通过全连接层self.fc 映射到最终的输出维度 output\_dim，得到最终的输出output。

这个模型结合了双向 LSTM 和注意力机制的优点：

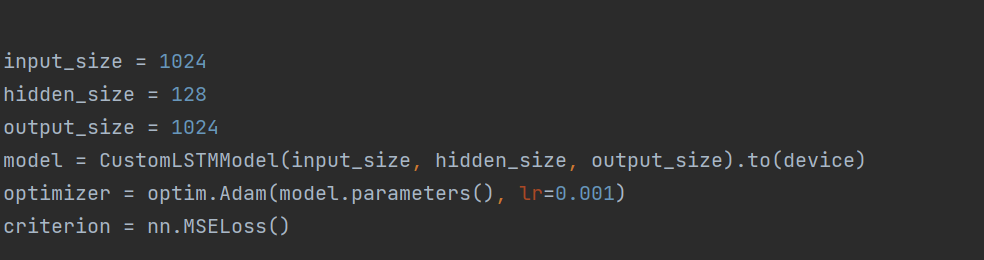
* 双向 LSTM 能够捕捉输入序列的上下文信息，因为它不仅考虑了从前到后的依赖关系，还考虑了从后到前的依赖关系。
* 注意力机制能够让模型在计算输出时关注输入序列中的重要部分，而不是等权重地对待每一个时间步。这在长序列处理中尤其重要。

1. 相似度计算：为了更好地检索文档，结合了 Cosine 相似度和 Jaccard 相似度



1. 模型训练

初始化模型、损失函数和优化器，此处使用均方误差损失函数（MSELoss）和Adam优化器进行模型训练



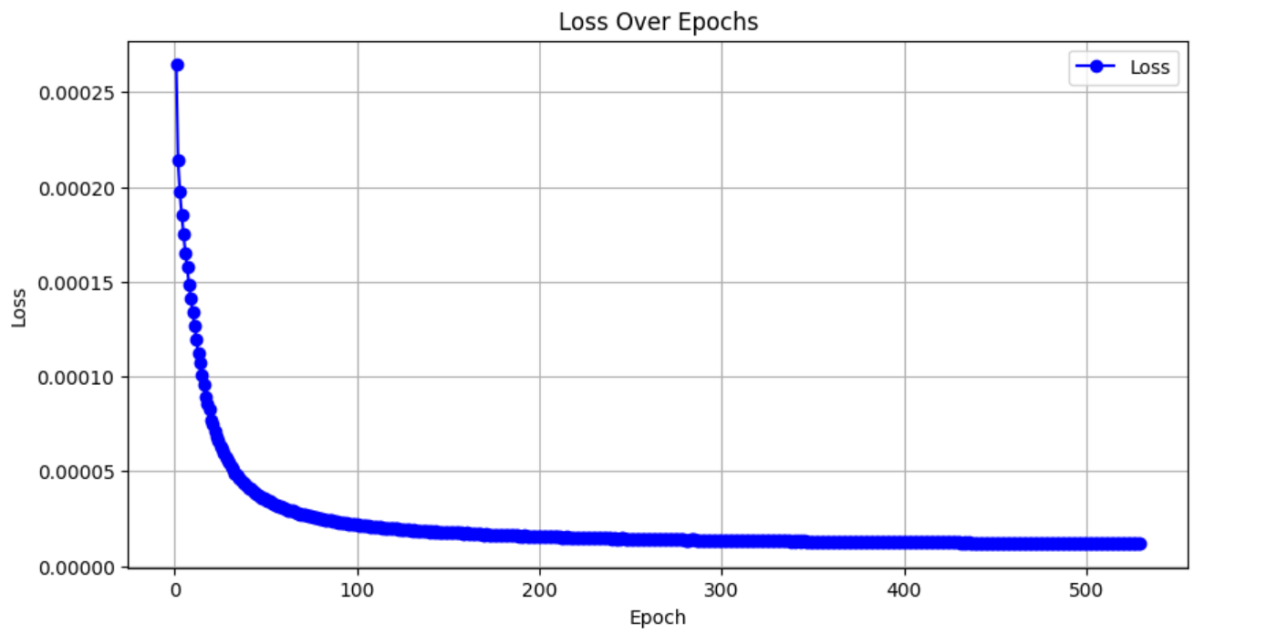
对每个 epoch，遍历训练数据，并计算损失进行反向传播和优化。（epoch的最佳值一直没能找到，目前最高版本的epoch为1500）



1. **实验结果**

Recall@3:0.5042

MRR@3:0.3728

Loss变化曲线：

**总结：**

可以明显看出本次实验是有明显不足的，Recall和MRR都比较低，如果模型选择已经是最优的话，那么参数可能不是最优的，或者Attention层没有进行进一步的尝试，导致学习不充分。效果不好的原因多种多样，可能还需要更多的时间去探索。

其实在本次实验的过程中我真的尝试了多种多样的模型，从最简单的余弦相似度再到BM\_25,TF-IDF,BERT，神经网络，注意力机制等等，几乎所有课上学过的内容我都有去尝试，但还是没能得到很好的结果，后续我也标注了数据的起始符号和终止符号，也进行了各种各样的处理，但随着越来越优化，模型越来越强大，反而造成结果变得更差，甚至结果一度达到了0。这段时间一直有竭尽全力去提升MRR和Recall，但效果微乎其微，但我觉得我也问心无愧了，日后还是继续加油吧。