# MiniGLM模型训练实验报告


### 2022011547 吕博涵


注：和模型相关的进行过修改的代码存在`MiniGLM`文件夹中，部分微调数据存在`datasets`文件夹中。

## 数据的准备

### 数据内容

默认的训练方法是先用金庸小说原文进行训练，然后使用问答集进行微调。但是经实验证明，这样的训练方法最多通过过拟合让模型能答对微调数据集中的问题，但是不能让模型拥有对语言的基本的理解能力：如理解“郭靖”是一个人名，每一句应该有主谓宾等结构等。因此，除了基本的小说外，我准备了更大量的中文预训练预料对模型进行训练，让模型拥有基本的语言模型的遣词造句的能力，如分清人名、地名，识别句中的主谓宾等。

我从Hugging Face上以及github上获取了众多预训练数据，其中有一些数据的格式是parquet，而服务器上没有能解析这种格式的包，因此我在本地将其转化为 jsonl格式后上传到服务器。

我准备的预训练预料如下：

- 历史上三年的搜狐新闻语料（约1G）
- 和金庸及其作品相关的百科
- 大量知乎问答语料（约2G）
- 筛选出与金庸及其作品相关的问答语料

以上的两种知乎问答语料我做了两种处理，一个是处理成预训练的格式，一个是处理成微调的格式。因此我的微调包含以下问答数据集：

- 大量知乎问答
- 金庸及其作品相关的知乎问答
- 小说内容问答（以及从中提取的精选问答）

其中小说内容问答就是在基础上扩充的微调数据集，我与俞鹤扬同学、朱子晗等同学、刘建东等同学共同补充微调数据集，总共积累了数万条高质量数据。

我生成微调数据集使用了以下方法：

- 从网上获取金庸相关的题库并对其格式进行处理得到一部分高质量问题集。
- 直接指定小说给定格式使用ChatGPT、Claude2、ChatGLM等LLM生成问答对。（在这三个中，ChatGLM的准确度最高。不过有趣的是，ChatGPT和ChatGLM都会主动说出“小昭成为了张无忌的义女”这样相同的错误的内容，怀疑是都学了不太好的语料。）
- 给Claude2提供维基百科上和小说内容相关的长文本，让其根据这些内容生成问答对。
- 使用ChatGPT根据示例生成可能出现的问题，并使用形如"`Q: <Example Question>, A: <Example Answer>, Q: <New Question>, A:`"的promp批量调用ChatGPT的api（使用3.5-turbo）生成答案并进行记录（属于One-Shot Learning）。

其中最后一种方法生成的效果最好，但是成本较高；第三种方法次之，由于给定了参考的材料，其生成问答的能力远远强于第二种方法。

## 数据处理

由于不同的数据集格式不同，我分别设计了不同的数据处理函数。数据处理函数主要分为两类：一类是处理与训练数据，一类是处理微调数据。这里着重介绍一下以`prepare_sft.py`为例的微调数据处理程序和`data_utils.py`中的`get_batch_sft()`函数。

在`prepare_sft.py`中，我首先使用`combined_qa = question + "[SEP]" + answer`对每个问答对的问题和答案进行拼接，然后使用`process(content, block_size)`函数对拼接好的问答进行处理，对长于`block_size`的字符串进行截断，并对短于`block_size`的字符串用`enc.eot_token`进行补全。

```
def process(content, block_size):
    tokens = list(enc.encode(content))
    if len(tokens) > block_size:
        return tokens[:block_size]
    else:
        return tokens + [enc.eot_token] * (block_size - len(tokens))
```

`data_utils.py`中的`get_batch_sft()`函数的主要部分如下：

```
def get_batch_sft(split, batch_size, block_size, device):
    ix = torch.randint(len(data) // block_size, (batch_size,))
    x = torch.stack([torch.from_numpy((data[i*block_size:
(i+1)*block_size]).astype(np.int64)) for i in ix])
```

```
    y = torch.cat((x[:, 1:], torch.full((x.shape[0], 1), enc.eot_token,
dtype=torch.int64)), dim=1)

    sep_sequence = enc.encode("[SEP]")
    sep_first_id = sep_sequence[0]

    init_positions = (y == sep_first_id).nonzero(as_tuple=True)
    loss_mask = torch.zeros_like(x, dtype=torch.float64)
    for i, sep_position in enumerate(init_positions):
        try:
            actual_answer_length = (y[i,
sep_position[1]+len(sep_sequence):] != enc.eot_token).sum().item()
            loss_mask[i,
sep_position[1]+len(sep_sequence):sep_position[1]+len(sep_sequence)+actu
al_answer_length+1] = 1
        except:
            pass
```

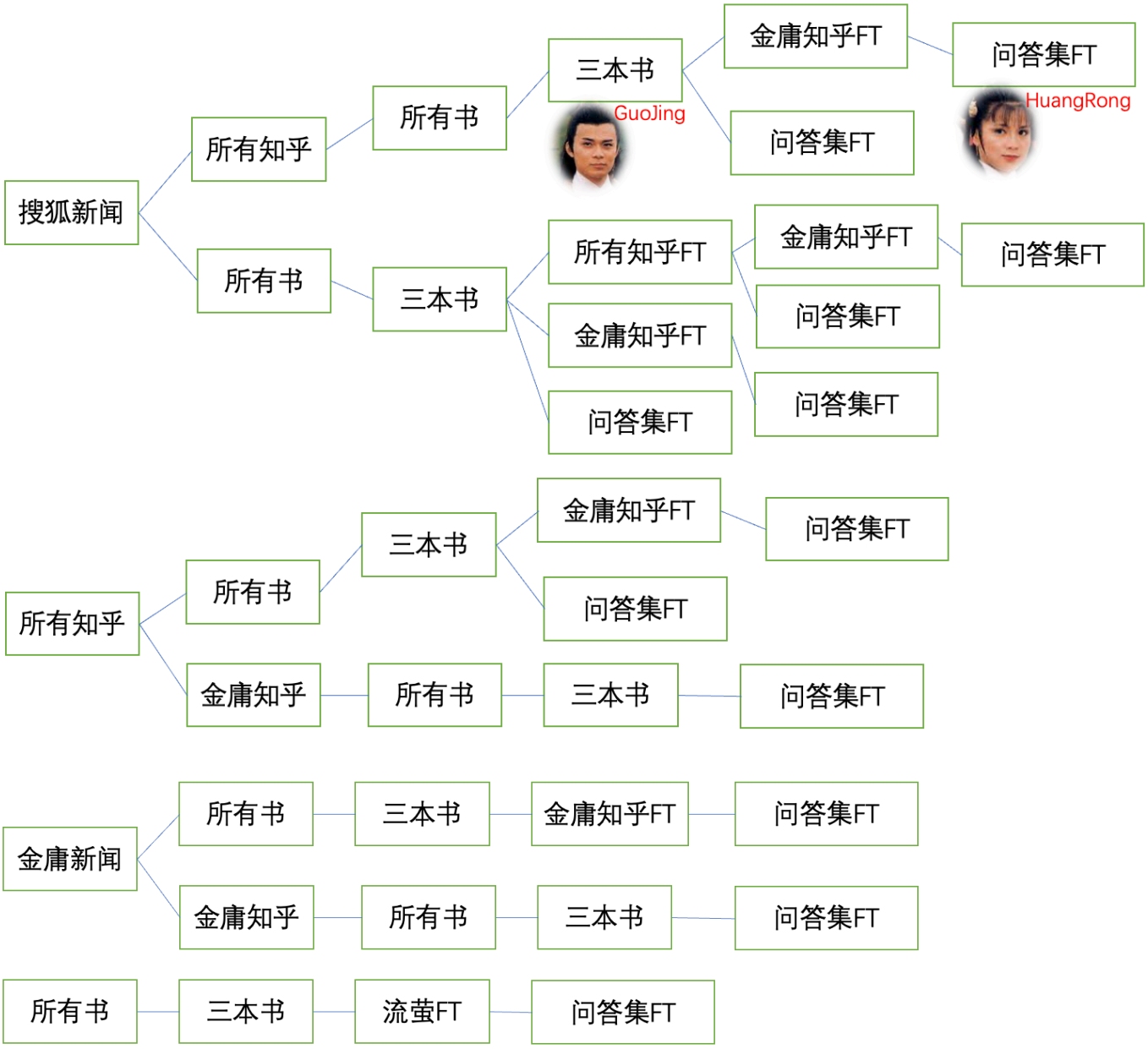
我做了如下设计：

1. ix与x的设置与预训练不同，只保留问题起始部位的位置对应的序号，这样能让模型每次的学习被限制在一个问题之内，而不会从前一个问题的答案开始，到后一个问题的问题内容结束。这样的操作通过对ix和x的特殊处理实现。
2. y的定义是x从第二项开始的`block_size-1`个字符加上一个`enc.eot_token`，保证y不会进入后一个问题。
3. `loss_mask`是从y的问题开始到第一个`enc.eot_token`为1，其他为0。这样的方式是**通过实验得出**的。我曾经尝试过**另外两个不同的设置**：1.从答案开始全部为1；2.只有答案部分为1。结果是第一种方式训练出的模型的答案普遍较短，第二种方式训练出的模型的答案普遍收不住。而**将`loss_mask`设置为从y的问题开始到第一个`enc.eot_token`为1这样的方式，得到的效果最好。**

如此处理的示意图如下：

# 模型训练

## 训练过程

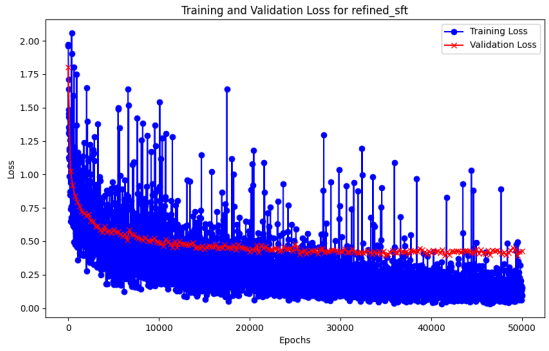


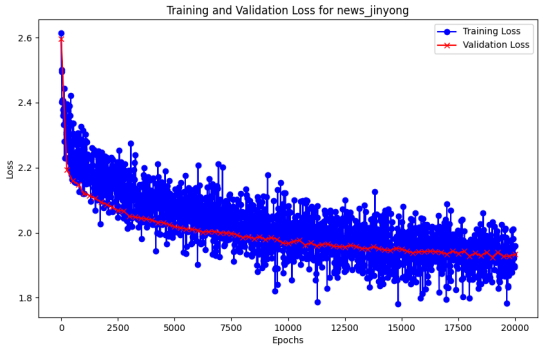

我用不同的预训练数据和微调数据的多种排列组合训练了众多模型，经过评测后分别为文本补全和问题回答选择了两个最好的模型：GuoJing (Generative unified operative Junction for integrated narrative growth)和HuangRong (Harmonized utility for answers and novel guidance with Real-time organize neural grasp)。

模型正如小说中的人物，郭靖（GuoJing）木讷，只能补全信息，但是很强大，能够对文本结构产生理解；黄蓉（HuangRong）巧舌如簧，能顺利回答出各种刁钻的问题。

该流程的训练过程如下：搜素新闻20000轮，所有知乎20000轮，所有书6000轮，三本书60000轮，金庸类知乎问答微调20000轮，问答集微调20000轮。

部分训练过程中的损失函数曲线如下：

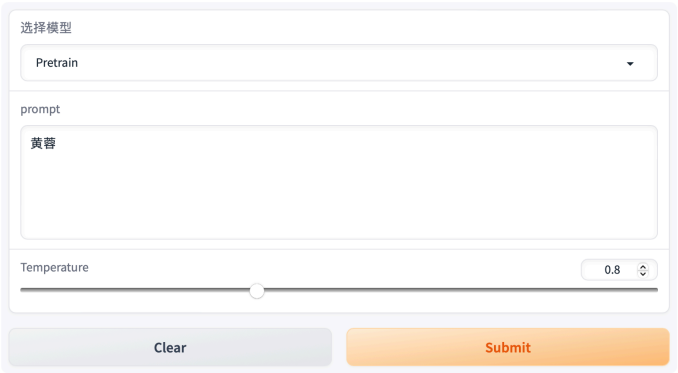



## 训练结果

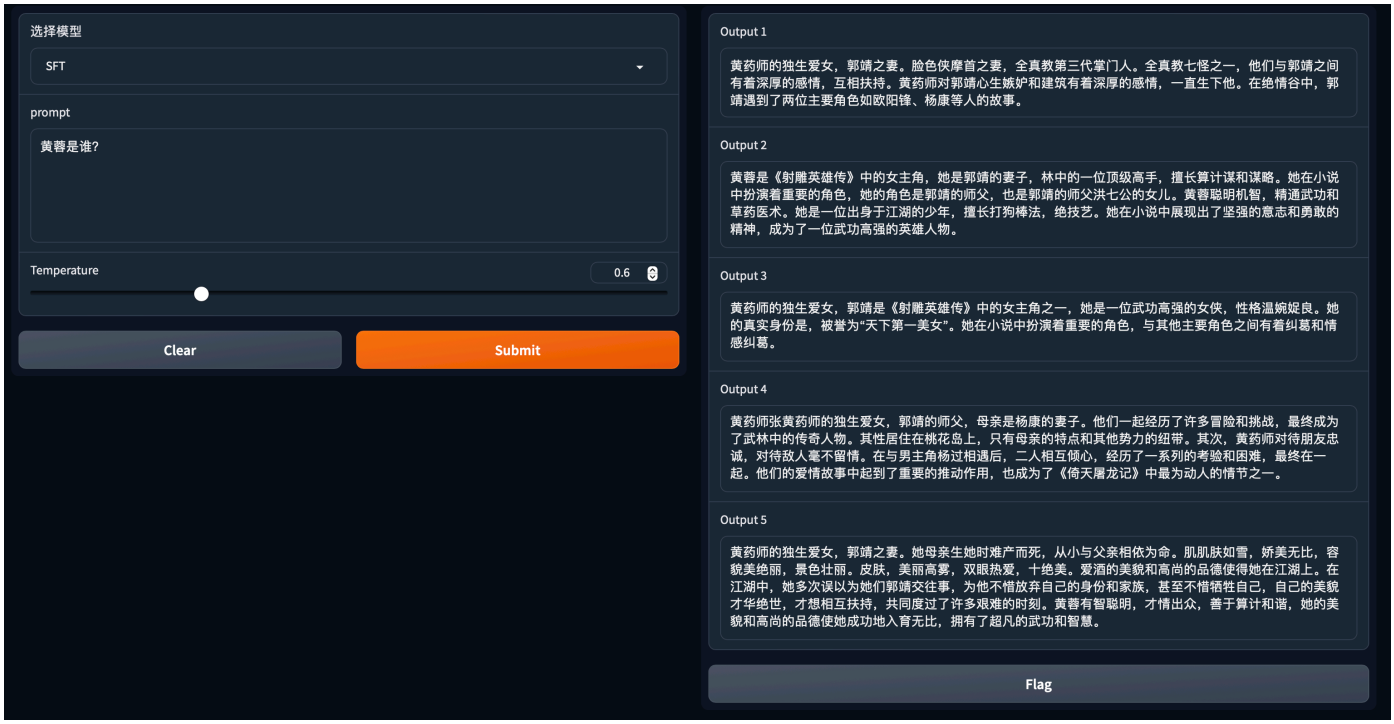
极为突出的是，由于我的模型训练文本丰富，训练有序，因此不仅能很好地续写并正确回答问题，还**没有出现过拟合**。

预训练模型续写评测：





微调模型回答能力评测：




在服务器上，每条512长度的回答能在2.5秒内生成。

从上面的回答中可以看出，续写模型具有语言组织能力，问答模型没有出现过拟合，而且回答的内容都和文本相关，具有一定的泛化能力。

## 模型效果分析

首先，**使用大量中文语料对大模型进行训练能显著提升大模型对语法、句法的理解能力**。在这样的训练后，大模型做续写或者回答相关问题时，**可能会出现事实性错误，但是在语法和句法上不会出错**。

同时，用上文所述的mask方式在保证文本长度的情况下让模型成功**学会什么时候应该终止生成**。在上图的示例中，大模型的回答均以句号作为结尾。

但是，模型**纵然能拥有一定的泛化能力，但是并不拥有真正的智能**。比如，在受训过如此多的语料后如果问大模型“郭靖是男的还是女的”这样的对于人类非常简单的问题，大模型并不能给出好的答复。

还有一个发现：在续写的模式下，如果temperature过低，容易生成很多重复，这可能是由于模型参数小的同时受训语料有限、同时新用来训练的语料基本盖过旧语料训练结果造成的。

## 模型效果量化评估

`evaluations.py`中，我使用了困惑度和Rouge-L两种评估方式。具体的评估方法是，准备一个本地的问答对数据集，然后调用模型根据问题生成回答，然后比对模型生成的答案和数据集中的答案，并用两种方法进行评估。

因为我们使用的损失函数是交叉熵损失，因此能较为轻易地得到困惑度。计算方法如下：

给定一个真实分布 $p$ 和一个模型预测分布 $q$，对于单个样本的交叉熵损失为：

$$CE(p, q) = -\sum_i p_i \log q_i$$

平均交叉熵损失：

$$\text{average } CE(p, q) = -\frac{1}{N} \sum_{j=1}^{N} \sum_i p_{ij} \log q_{ij}$$

困惑度 (Perplexity, PP):

$$PP = e^{-\frac{1}{N} \sum_{j=1}^{N} \sum_i p_{ij} \log q_{ij}} = e^{\text{average } CE(p,q)}$$

对于Rouge-L：

$$\text{Recall} = \frac{LCS(R,C)}{|R|}$$

$$\text{Precision} = \frac{LCS(R,C)}{|C|}$$

$$F_\beta = \frac{(1+\beta^2) \times \text{Recall} \times \text{Precision}}{\beta^2 \times \text{Precision} + \text{Recall}}$$

代码层面，困惑度的计算直接使用`math.exp(loss)`实现，Rouge-L使用`rouge`包中的`Rouge`类实现。

# 其他探索性质的尝试

由于这个参数量的Transformer的效果甚至不如笔者原来使用LSTM做文本续写的效果，因此我写了一个十分简单的LSTM模型（`lstm_model.py`）和对应的训练代码（`lstm_train.py`）。训练代码可以直接运行。

模型类代码：

```
class LSTM(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers,
batch_first=True):
        super().__init__()
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers,
                            batch_first=batch_first)

    def forward(self, x):
        x, _ = self.lstm(x)
        return x
```

```python
class LSTMModel(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers):
        super().__init__()
        self.lstm1 = LSTM(input_size, hidden_size, num_layers)
        self.linear = nn.Linear(hidden_size, hidden_size)
        self.lstm2 = LSTM(hidden_size, hidden_size, num_layers)
        self.dropout = nn.Dropout(p=0.3)

    def forward(self, x):
        x = self.dropout(x)
        x = self.lstm1(x)
        x = self.linear(x)
        x = self.lstm2(x)
        return x
```

训练过程：

```
(base) py2022011547@machine00:~/hw2/MiniGLM$ python lstm_train.py
Step 0, Loss 0.014772500842809677
Step 10, Loss 0.014788216911256313
Step 20, Loss 0.014756015501916409
Step 30, Loss 0.014838072471320629
Step 40, Loss 0.014955786056816578
Step 50, Loss 0.014582602307200432
Step 60, Loss 0.014842474833130836
Step 70, Loss 0.014225970953702927
Step 80, Loss 0.014698131009936333
Step 90, Loss 0.014557737857103348
```