

AI Project4: Seq2Seq Text Generation

[Dataset](#)

[Metrics](#)

[BLEU](#)

[METEOR](#)

[ROUGE](#)

[Model](#)

[Encoder-Decoder](#)

[RNN](#)

[Transformer](#)

[Prediction](#)

[Conclusion](#)

Dataset

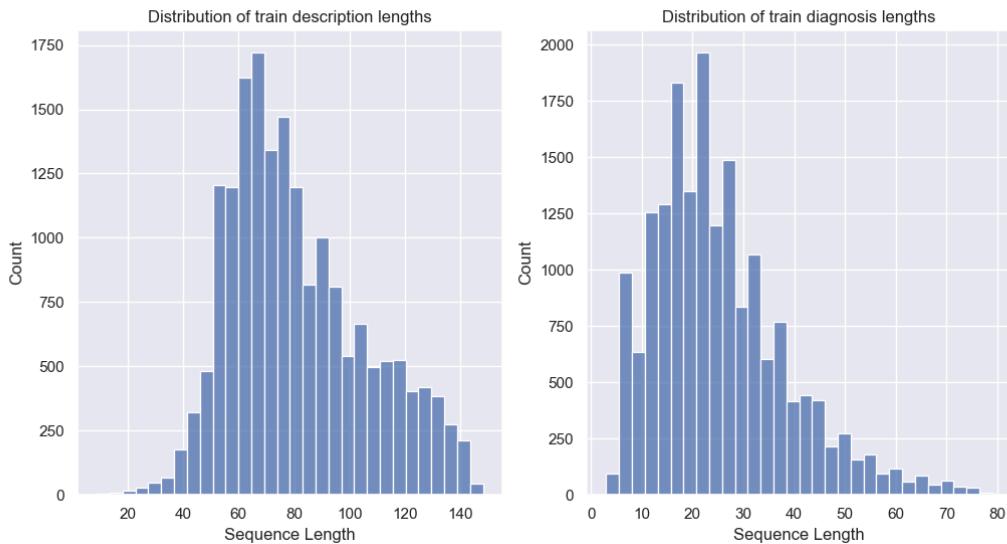
本次实验的数据集有两个文件, `train.csv` 和 `diagnosis.csv`, 其格式均为

index	description	diagnosis
-------	-------------	-----------

其中, `description` 和 `diagnosis` 均已被处理成数字序列, 我们的任务简要来说就是训练一个模型, 使其根据 `description` 生成 `diagnosis`. 在构建这样一个序列到序列的模型之前, 我们需要先对数据集做一些基本的统计, 以获得几个重要的超参数

- 源序列(即 `description`)和目标序列(即 `diagnosis`)的最长长度, 需要据此进行 `padding` 或 `truncate` 等操作来统一序列长度
- 词表大小(总共有多少个不同的数字)

此外, 本次任务还带有一定的 `摘要` 性质, 因此我们还期望对比 `description` 和 `diagnosis` 的长度分布, 我们生成的 `diagnosis` 的长度最好和原始训练集已有的近似. 训练集的长度分布如下



经统计,训练集有 18000 条记录,而测试集有 2000 条.训练集中 `description` 和 `diagnosis` 的主要信息如下表

	<code>description</code>	<code>diagnosis</code>
最大长度	148	79
平均长度	81.24	25.33
中位数长度	76	23

词表大小方面比较特殊,数据集中总共有 1291 个不同数字,但最大的数字是 1299 ,略微有一些不一致.重要的超参数如下:

```
max_source_length = 148
max_target_length = 79
vocab_size = 1300
```

在编程时,我们可以利用 `torch` 的 `Dataset` 类搭配 `DataLoader` ,实现 `__getitem__` 和 `__len__` 方法来构建数据集.基于这些信息,我们开始着手构建模型.

Metrics

首先介绍一下本次实验中选用的一些评估生成序列质量的指标.

BLEU

BLEU (Bilingual Evaluation Understudy) 是机器翻译评估的重要指标,基本公式为:

$$BLEU = BP \cdot \exp\left(\sum_{n=1}^N w_n \log p_n\right)$$

其中 **BP** 是brevity penalty(简短惩罚因子):

$$BP = \begin{cases} 1 & \text{if } c > r \\ e^{(1-r/c)} & \text{if } c \leq r \end{cases}$$

BLEU-N 的精确率计算为:

$$p_n = \frac{\sum_{C \in \{Candidates\}} \sum_{ngram \in C} Count_{clip}(ngram)}{\sum_{C' \in \{Candidates\}} \sum_{ngram' \in C'} Count(ngram')}$$

BLEU-4 的权重通常为:

$$w_1 = w_2 = w_3 = w_4 = \frac{1}{4}$$

⌚ Caution

注意 **BLEU-4** 不是指 w_4 为1,其他均为0,而是均分的.

❗ Important

`nltk.translate.bleu_score` 提供了 `corpus_bleu` 和 `sentence_bleu` 两种方法,他们是 **宏平均** 和 **微平均** 的关系,前者将所有句子合并后整体计算,后者每句单独计算后平均.句子级 **BLEU** 还有一个问题是其使用了几何平均数,如果任何 **n-gram** 精确率为0,整体分数会变为0,导致对短句子评估特别不友好.因此提出了平滑方法,常见的方法包括:

- 方法1 (Add-one Smoothing):

$$Count_{smooth}(ngram) = Count(ngram) + 1$$

- 方法7 (递归平滑):

$$p_n = \begin{cases} p_n & \text{if } Count(ngram) > 0 \\ \alpha p_{n-1} & \text{if } Count(ngram) = 0 \end{cases}$$

其中 α 是平滑因子,通常取0.0-1.0之间.

BLEU 的主要缺点是只关注了ngram之间的匹配,缺少对于同义词之类情况的处理等.

METEOR

METEOR (Metric for Evaluation of Translation with Explicit ORdering) 是另一个评估机器翻译质量的指标系统, 它首先计算精确的匹配度, 然后对结果进行惩罚调整. 其计算公式如下:

$$METEOR = F_{mean} * (1 - Penalty), F_{mean} = \frac{10PR}{R+9P}, P = \frac{m}{w_t}, R = \frac{m}{w_r}$$

$$Penalty = 0.5 * \left(\frac{ch}{m}\right)^3$$

其中

- P 是准确率, m 是匹配的 unigram 数量, w_t 是翻译结果中的单词数量
- R 是召回率, w_r 是参考翻译中的单词数量
- ch 是分块(chunk)的数量, 表示连续匹配序列
- Penalty 是惩罚项, 用于衡量翻译结果的词序问题

在编程时, 我们可以调用 `nltk.translate.meteor_score`. METEOR 对语序变化和同义词支持比较好.

ROUGE

ROUGE (Recall-Oriented Understudy for Gisting Evaluation) 也是一组用于评估自动文本摘要和机器翻译的评价指标. 有 **ROUGE-N** 和 **ROUGE-L** 两种常用度量:

ROUGE-N 计算候选文本与参考文本之间 n-gram 的重合度, 具体来说:

$$ROUGE-N = \frac{\sum_{S \in \{RefSum\}} \sum_{gram_n \in S} Count_{match}(gram_n)}{\sum_{S \in \{RefSum\}} \sum_{gram_n \in S} Count(gram_n)}$$

其中:

- $Count_{match}(gram_n)$ 表示候选文本和参考文本中共同出现的 n-gram 的数量
- $Count(gram_n)$ 表示参考文本中 n-gram 的数量
- $RefSum$ 表示参考文本集合

ROUGE-L 基于最长公共子序列 (LCS) 计算, 公式为:

$$ROUGE-L_P = \frac{LCS(X, Y)}{LCS^m(X, Y)}$$

$$ROUGE-L_R = \frac{LCS^n(X, Y)}{(1+\beta^2)R_L P_L}$$

$$ROUGE-L_F = \frac{(1+\beta^2)R_L P_L}{R_L + \beta^2 P_L}$$

其中:

- X 是候选文本, Y 是参考文本

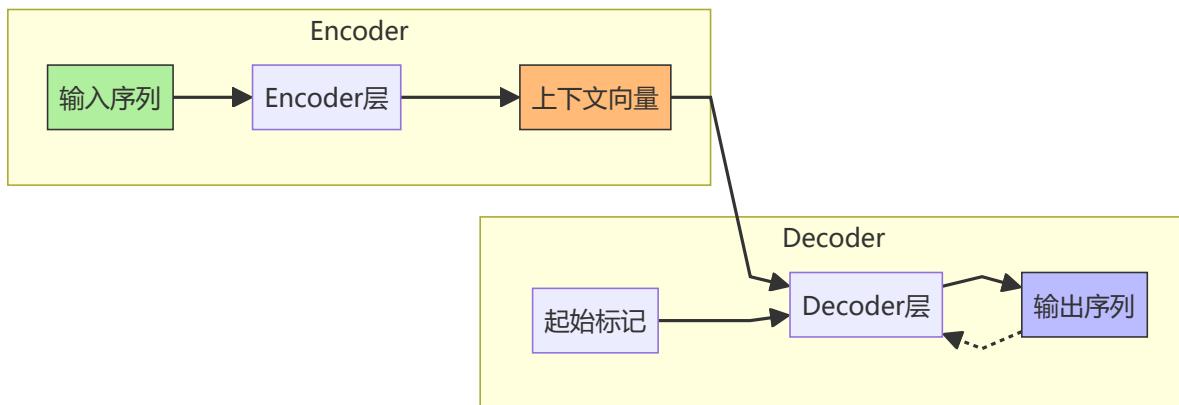
- m 是候选文本的长度, n 是参考文本的长度
- $\text{LCS}(X,Y)$ 是 X 和 Y 的最长公共子序列的长度
- β 通常设置为 1.2

编程时可以调用 `rouge_score` 来计算.对于 `ROUGE-N`,我们可以选取 n 为 1 和 2 来进行测试.他们都提供了 `precision`, `recall` 和 `F1` 三个指标,我们用最综合的 `F1` 作为主要参考. `ROUGE` 系列考虑了召回率,比较适合摘要生成评估,不过对语义理解有限.

Model

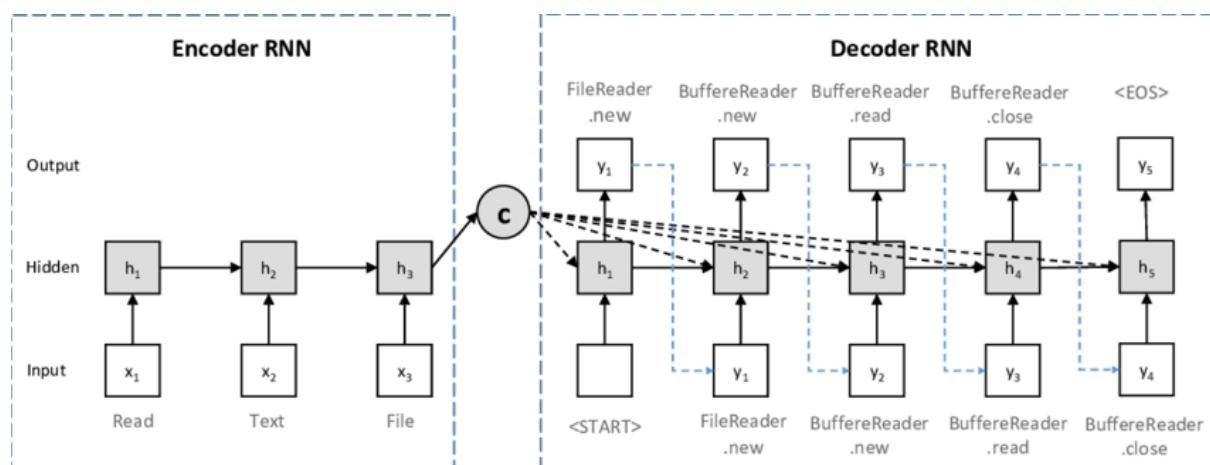
Encoder-Decoder

我们的模型在宏观层面上遵循如下的 Encoder-Decoder 架构.即由 *Encoder* 处理输入序列并得到上下文的表征,再交由 *Decoder* 生成输出序列,并且输出序列会反作用于 *Decoder*.

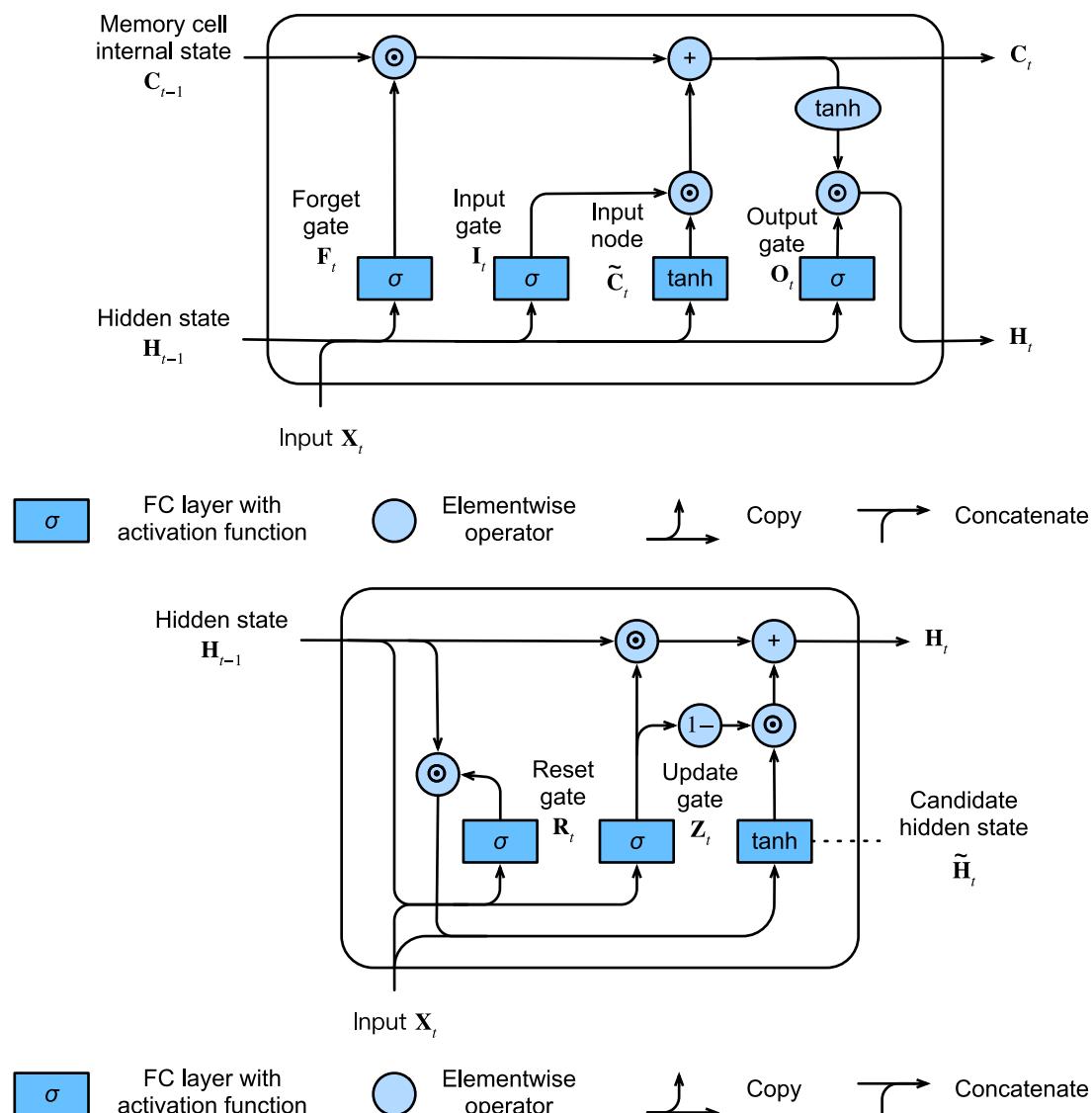


RNN

`RNN` 是经典的处理序列问题的神经网络架构,基于它的 Encoder-Decoder 架构如图:



在这里 *Encoder* 和 *Decoder* 之间的交互很简单,就是单纯的由前者输出一个向量传递给后者.不过 RNN 在实践中有很多问题,例如 梯度消失 和 梯度爆炸 ,因此诞生了 **LSTM** 和 **GRU** 这样的改进的 RNN ,他们的一个单元内的架构分别如下:



这三个模型总体上是类似的,不过 **LSTM** 和 **GRU** 应当在训练过程中会更稳定,并且最终可以获得更好的效果.

💡 Tip

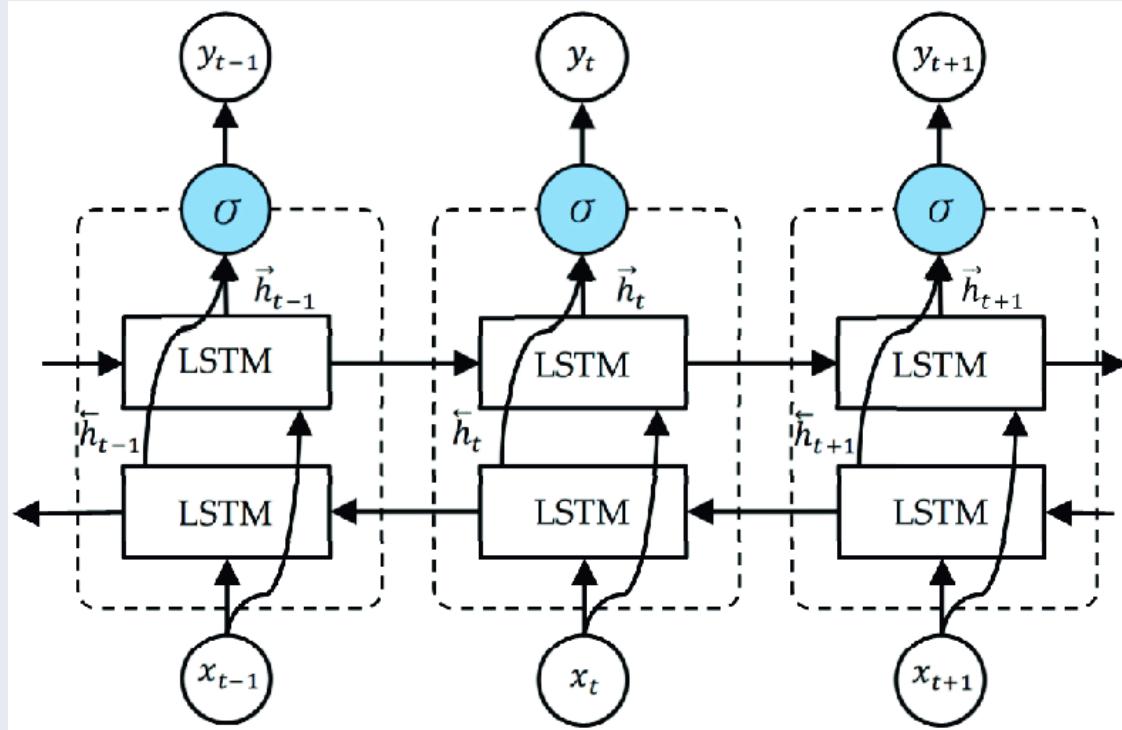
在 RNN 中,我们可以用 梯度裁剪 来改善 梯度爆炸 问题,例如

```
torch.nn.utils.clip_grad_norm_(self.model.parameters(),
    self.config.max_grad_norm)
```

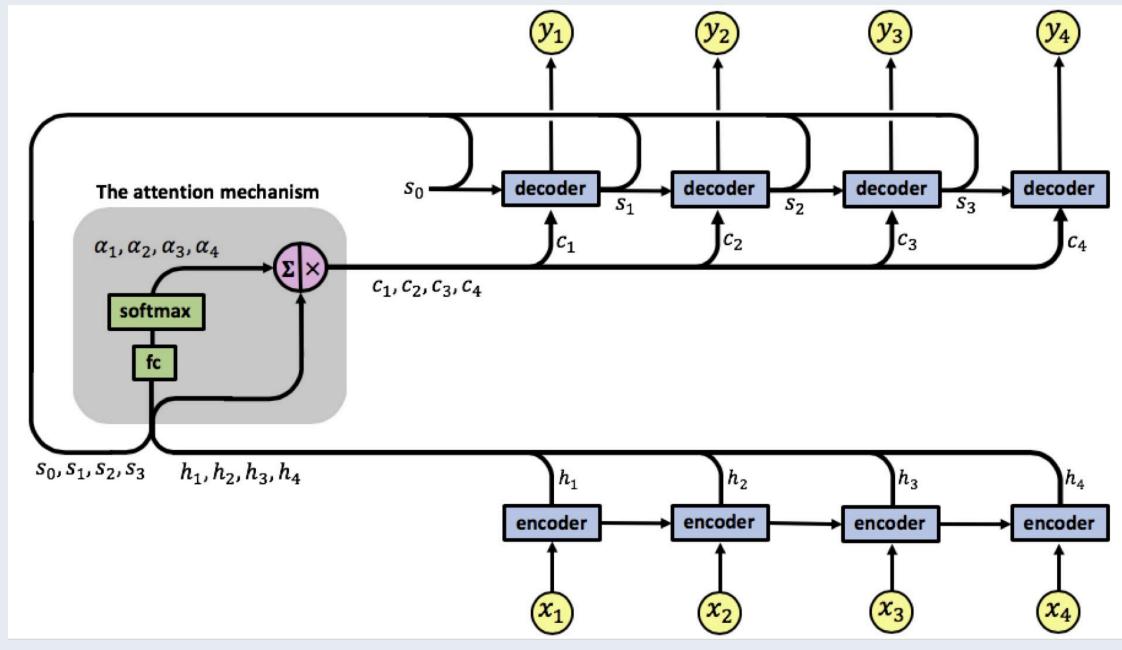
但 梯度消失 问题只能靠 **LSTM** 或 **GRU** 的架构改进来实现.

ⓘ Note

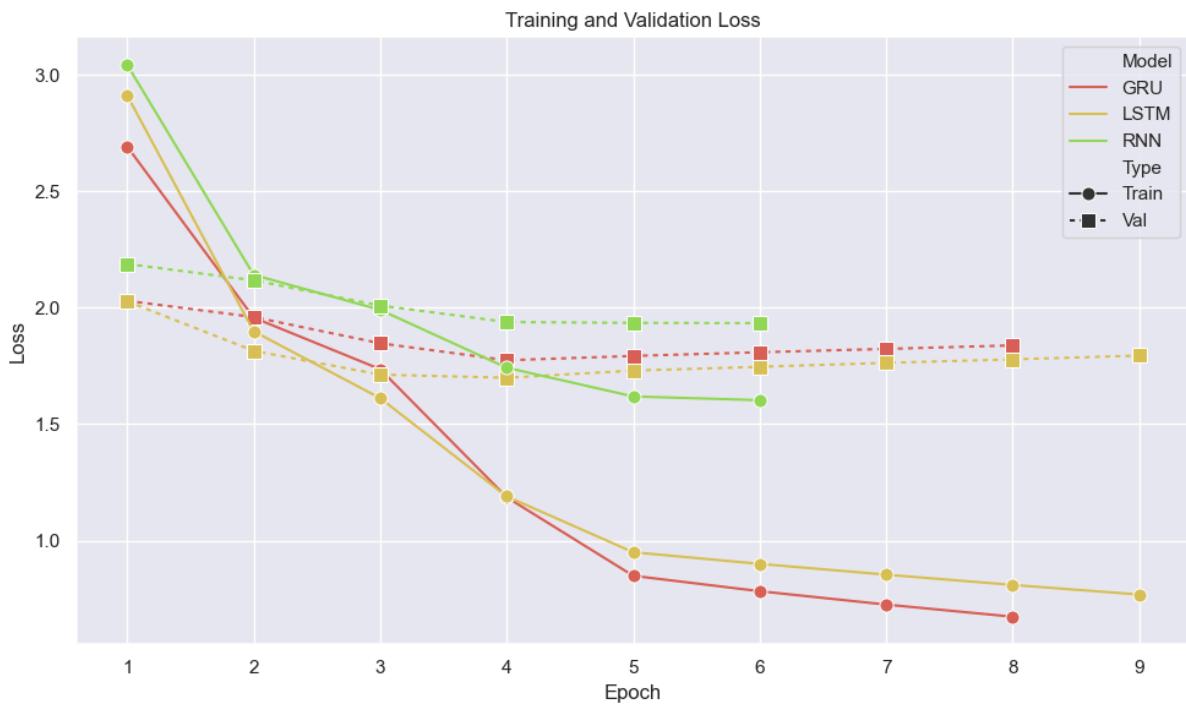
在 [NLP](#) 任务中,我们还可以做一种优化.就像我们人类理解文字时会结合 [上下文](#),模型也应当可以综合过去和未来的信息.因此诞生了例如双向 [LSTM](#) 这样的架构,由前后两个 [LSTM](#) 并行,综合它们的输出,这种架构如下图:



并且, [LSTM](#) 还可以进一步结合 [注意力](#) 机制,如下图:



三个模型的实验结果如下(我们实现的是双向带注意力的 [LSTM](#) 和 [GRU](#)):



观察三个模型的损失曲线,可以发现以下两点:

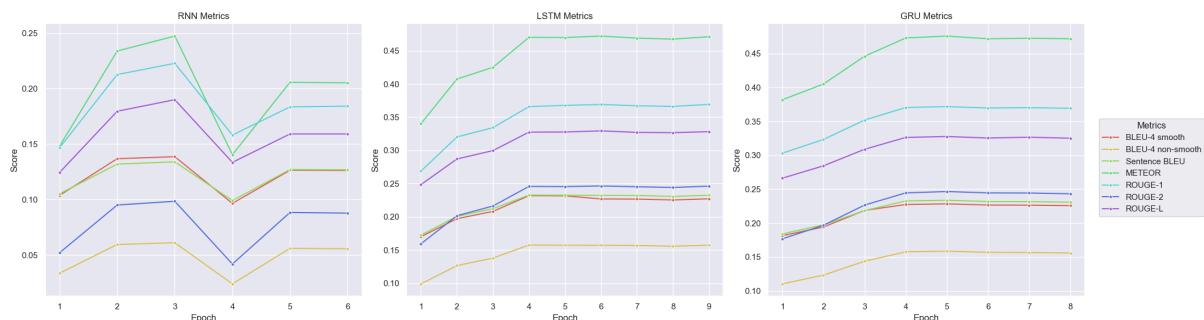
- RNN 的训练比较困难,训练损失的下降比较缓慢,验证损失在第四个epoch后就基本不再下降

💡 Tip

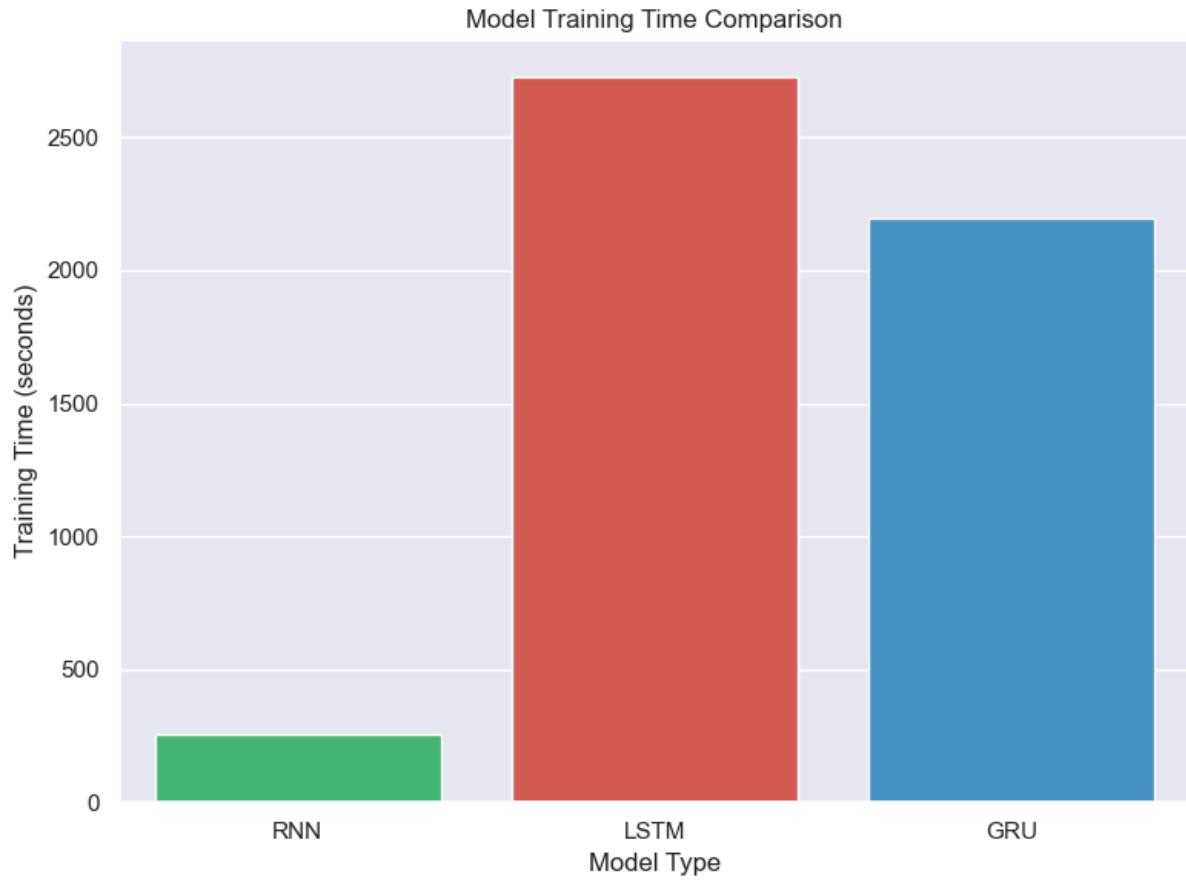
我们使用了 `patience` 为3的 **早停** 机制,因此 RNN 在第六个epoch就中止训练了.

- GRU 和 LSTM 的效果明显好于 RNN .训练损失下降较快且持续下降. GRU 效果略微更好.但模型过拟合比较明显,在后面几个epoch,虽然训练损失仍然不断下降,但验证损失基本不再变化.

验证集上的各指标如下图:



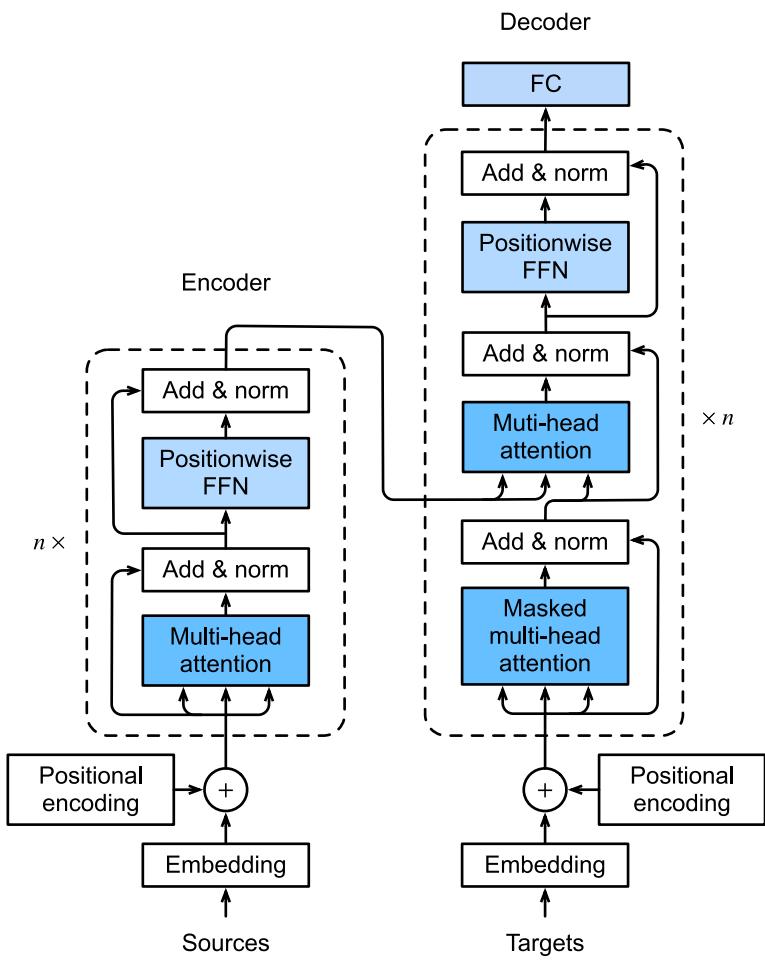
和损失的结果是类似的, RNN 的效果很差,平滑后的 BLEU-4 (红线)都没有超过0.15,基本不可用. LSTM 和 GRU 的效果略好一些,但也没有超过0.25,未平滑的(黄线)甚至只有约0.15,难以让我们满意.各模型训练时间如下:



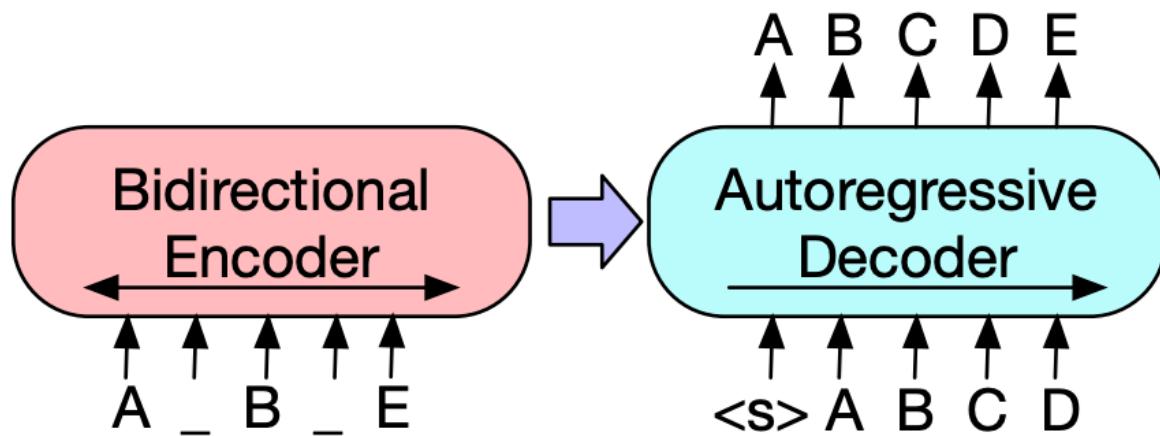
RNN 非常快,不过效果非常差; LSTM 和 GRU 表现基本近似(注意这里由于早停机制, LSTM 多训练了一个epoch),在我的设备上每个epoch均需要约250s.

Transformer

我们发现传统的基于 RNN 的模型无法在本任务中取得太好的性能,因此,我们继续尝试使用基于 Transformer 的模型来解决这个问题. Transformer 的架构如下:



Transformer 本身就包含了 Encoder 和 Decoder ,有很多经典的基于 Transformer 的预训练语言模型,例如 BERT (只使用了 Encoder), OPT / GPT (只使用了 Decoder), BART / T5 (Encoder 和 Decoder).很自然的, BART 和 T5 非常适合解决本次任务,因此下面我们就分别使用这两个模型.他们都提供了多个参数量的版本,结合我的设备条件,我都选择了最小的版本(`bart-base` 和 `t5-small`). BART 的架构可抽象如下:



即包含了一个双向编码器(类似 BERT)和一个自回归解码器(类似 GPT). T5 和 BART 虽然都在 Transformer 的范式下,不过有一些细节的区别,例如位置编码不同(分别是相对编码和可学习得绝对编码),以及预训练任务不同等.

两个模型训练过程中每batch损失曲线如下:



可以看到损失曲线基本比较合理.

⚠ Important

虽然 **BART** 和 **T5** 都基于 **Tranformer**, 但实际上还是有不小区别的. 在训练中, 我对 **T5** 采用了峰值 **1e-3** 的学习率, 0.1比例预热+余弦衰减, **AdamW** 优化, 默认的权值衰减参数, 从曲线可以看出, 这套超参数基本上比较合理. 不过对于 **BART** 来说, **1e-3** 的学习率过高, 甚至在第一个epoch都可能出现损失上升的情况, 我原本把它降低至了 **1e-4**, 并在随机种子 **42** 上测试基本没有问题, 但我换了一个随机种子后, 再次出现了模型在前几个epoch就出现性能退化. 最后将学习率设置为 **5e-5**, 基本适合 **BART** 模型. 这说明了采用多个随机种子验证模型稳定性的必要性. 根据查阅的一些资料可知, 这种学习率的差异是合理的, 是由于 **T5** 和 **BART** 的架构差异导致的.

因此, 在这个图中我们看到, **T5** 初期损失下降更快, 因为学习率更高, 不过最后两个模型的损失基本处于一个水平. 但 **BART** 总体的波动情况会更大一些.

⚠ Warning

对于不同的模型, 我们不能机械地套用超参数, 例如我的显卡只有4G显存, 对于 **t5-small**, 我采用16的 **batch_size** 基本比较合理, 但对于更大的 **bart-base**, 如果我们保持这个值, 就会发现模型训练非常慢, 因为此时显存不够用了, 程序在用带宽低很多的内存来训练, 我们就需要降低 **batch_size**, 例如在我的电脑上采用8是比较合适的. 因此这个图里 **BART** 的总batch数是 **T5** 的两倍.

对于生成过程, **transformers** 模型的 **generate** 方法提供了一些很有用的参数来实现一些高级的解码策略, 包括

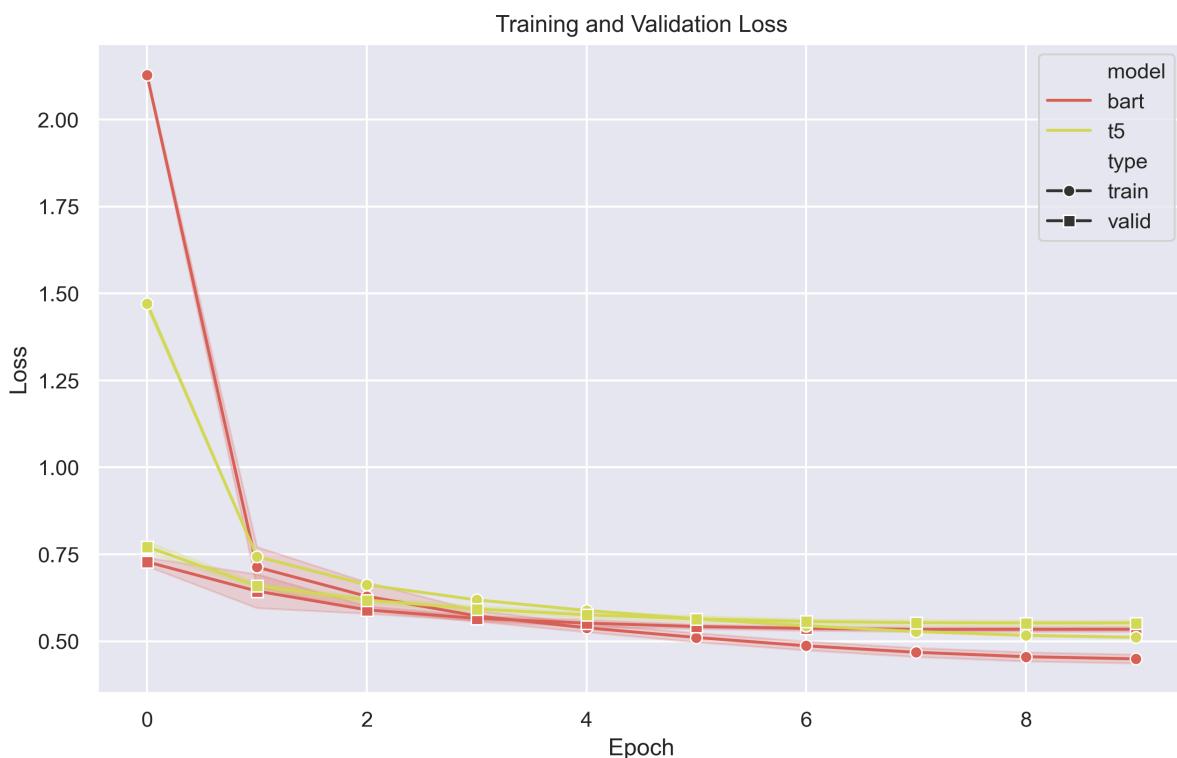
- **num_beams** 用来实现束搜索

- `num_beam_groups` 对光束分组,实现 *diverse* 的束搜索
- `diversity_penalty` 配合使用,鼓励模型生成更多元化的备选答案
- `no_repeat_ngram_size` 禁止重复的n-gram
- `early_stopping` 该项如果设为 `True`,会在找到 `num_beams` 个候选后就终止搜索,而不是等到完全确定没有更好的候选.如果束数量较大,这种方法可以有效提高生成的效率.

① Note

在实验过程中,我发现验证集上会出现很多生成的 `diagnosis` 比较类似的部分,即使 `description` 差异较大.为了应对这种情况,结合硬件条件.我设置了2组光束,一组各3个,并带有 `diversity_penalty`,有效缓解了模型输出单一的情况.

为了验证模型的稳定性,考虑到交叉验证耗时较长,我选取了2333, 4001, 6007, 8009, 9001五个不同的随机种子来划分验证集(0.1比例,匹配测试集大小),对模型进行测试,结果如下:



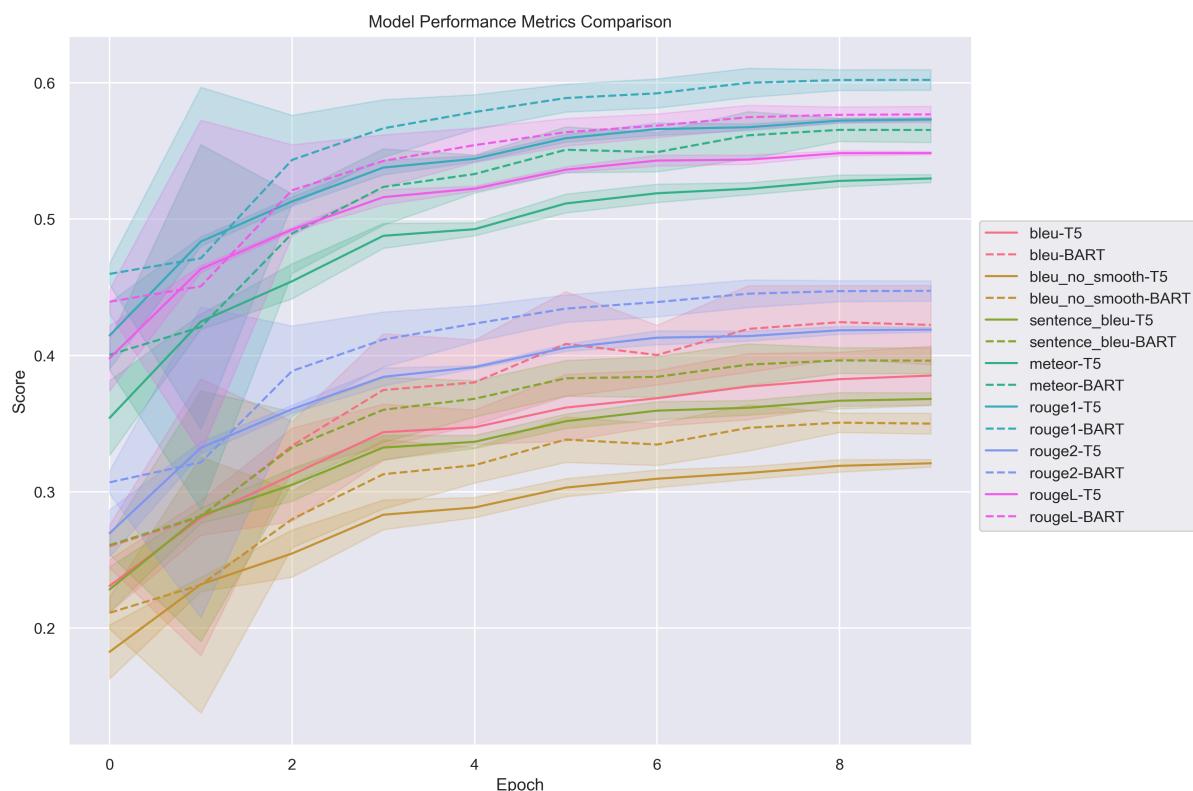
在每epoch的数据上,总体上来看,充分训练后训练集损失低于验证集损失, **BART** 最终的训练损失可以很低,说明它可以充分拟合训练数据,但是过拟合相比较于 **T5** 更严重一些,最终的验证集损失只是略好于 **T5**,差距远小于他们在训练集上的差距.这个现象基本符合直觉,虽然都基于 **Transformer**,但因为我们选用的 **bart-base** 模型相较于 **t5-small** 来说参数量大了很多,自然有更强的拟合能力,但泛化能力上不一定更优.在这个比较简单的任务中,选用更大参数量的模型,也未必能在验证集上带来显著更好的效果,性价比可能会比较低.

此外, **T5** 的训练更加稳定,误差带在这个图上基本看不到,但 **BART** 训练前期稳定性较差,在前几个epoch中,五次训练有较宽的误差带,参数量更大的模型训练起来可能会更加困难.

我们还可以看到验证集损失的下降相较于训练集损失的下降慢了很多,这说明拟合现有数据比泛化未知数据要简单很多.

⚠ Caution

这个图里有一个看起来比较反直觉的地方,就是前面几个epoch看上去训练损失比验证损失还高,事实上这是因为第一个验证集数据是模型训练了一个epoch之后产生的,而第一个训练损失是这个过程产生的,一开始的损失会非常大,但 **BART** 和 **T5** 的学习能力都很强,所以训练了一个epoch后,就已经学习了比较多,导致第一次验证损失就是一个比较低的水平.

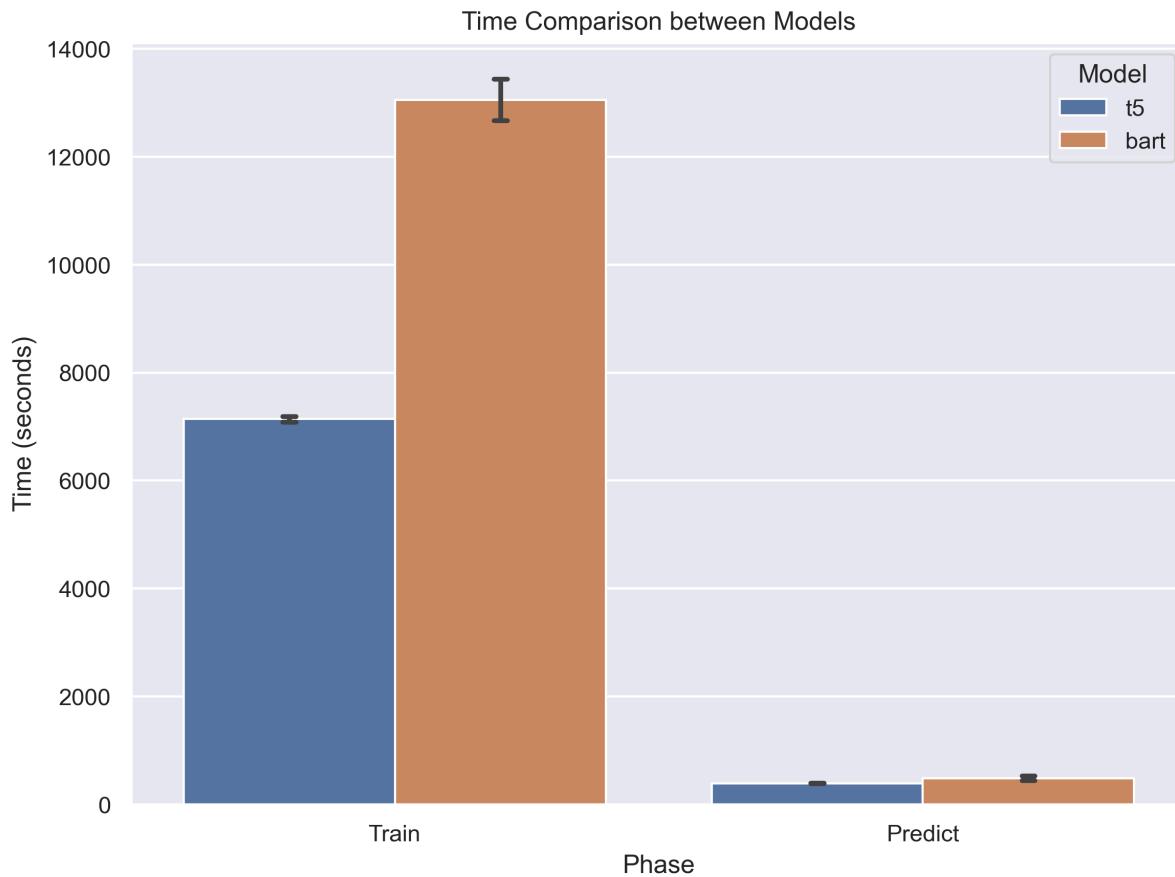


指标的上涨趋势基本符合验证集损失的下降趋势.我们可以发现基于 **Transformer** 的模型表现要比传统的基于 **RNN** 的模型好很多,这主要是因为它可以建立任何位置之间的联系,更容易捕获全局依赖关系;并且具有多头注意力机制,赋予了其更强的特征提取模型.对于 **BART** 和 **T5** 来说,在本次实验中,他们都有比较好的效果,验证集上未平滑的 **BLEU** 约0.3至0.35,是先前带有简单注意力机制的双向 **LSTM** 和 **GRU** 两倍还多了.在各个指标上,都是 **BART** 比 **T5** 好一些.

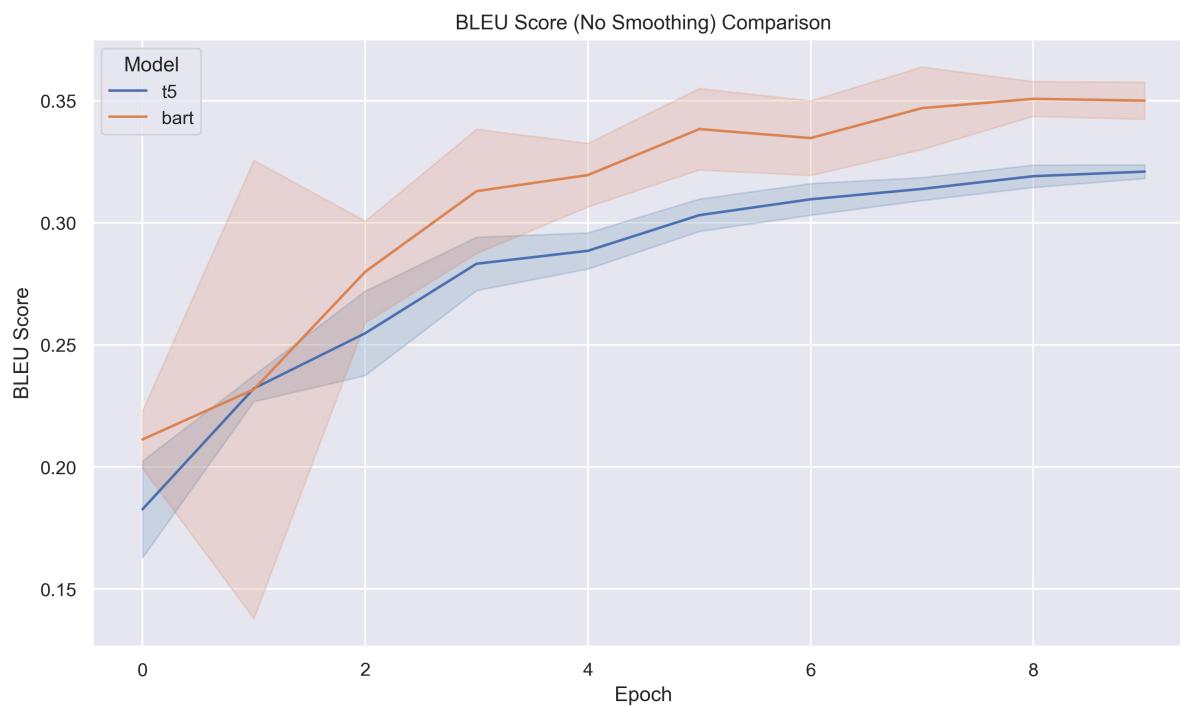
⚠ Warning

这并不说明 **BART** 就比 **T5** 更好,一方面,本次选用的 **bart-base** 有约 **130M** 参数量,而 **t5-small** 仅有约 **60M**,甚至不足一半,因此这个对比很难说是完全公平的.并且这也仅仅只是一个任务下的性能,不能直接下结论.

反映在训练和预测时间上, **BART** 花费的时间也约有 **T5** 的两倍(并且误差更大), 预测时间则略长:



此外 **BART** 在本次实验中还有一个比较大的问题, 我们以未平滑的 **BLEU** 为例:



我们发现 **T5** 的数据是比较符合我们预期的, 指标总体上涨, 并且误差带逐渐收窄, 到最后基本趋于稳定. 而 **BART** 则出现了比较多的上下抖动, 最后的误差带也显著更宽, 收窄趋势不明显, 并且在第二个epoch出现了一次很大的标准差(事实上是因为2333随机种子的数据表现很差), 说明 **BART** 的训练稳定性要更差一些. 对于更大的模型, 如何训

练好是一个我们必须关注的问题,选用多个随机种子测试,可以帮助我们发现模型的不稳定性,并针对性的调正训练策略和参数.

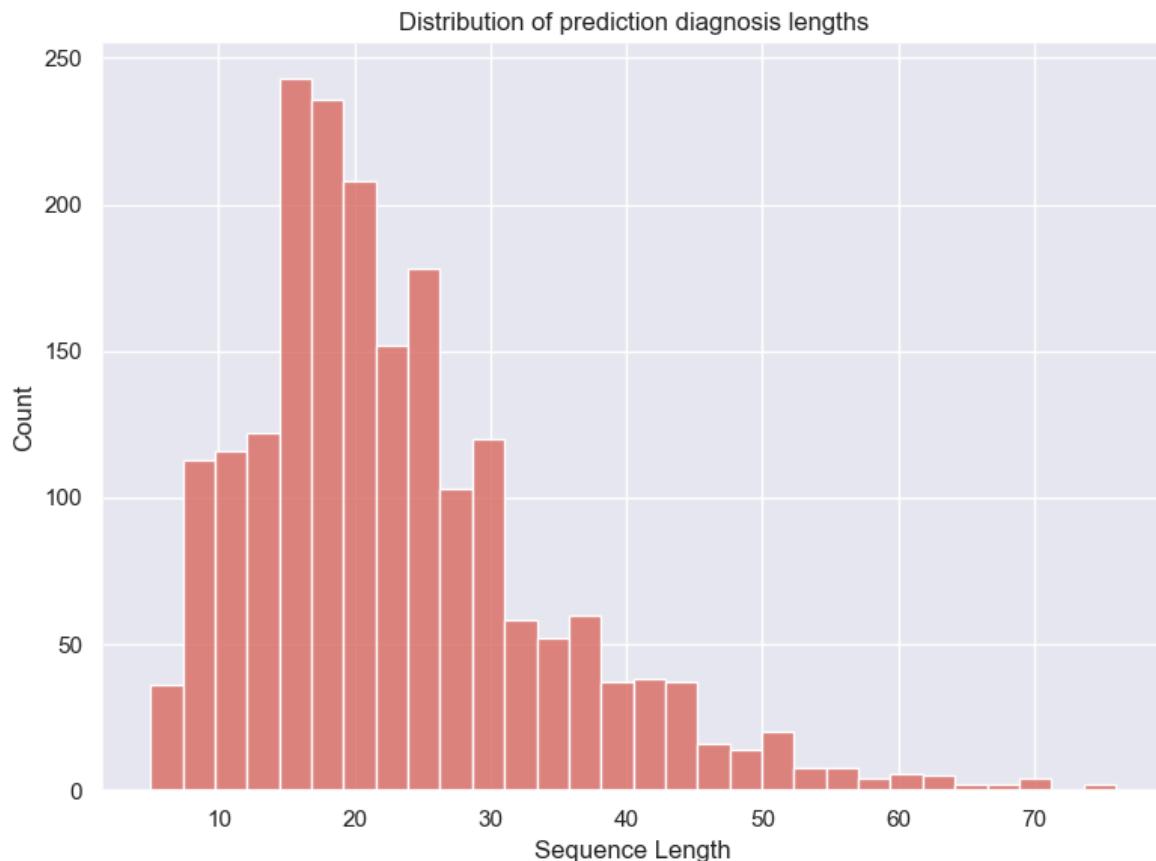
最后在验证集上,两个模型的具体最好指标数据如下:

模型	BLEU	BLEU(No smooth)	Sentence BLEU	METEOR	ROUGE-1	ROUGE-2	ROUGE-L
T5	0.3887	0.3248	0.3734	0.5335	0.5743	0.4213	0.5486
BART	0.4208	0.3599	0.4072	0.5749	0.6088	0.4564	0.5808

在编程时,可以调用 [AutoModelForSeq2SeqLM](#) 来构建 Seq2Seq 模型.

Prediction

在训练集上训练模型,在验证集上验证模型之后,我们还需要最后在测试集上进行预测,我选择了在验证阶段表现最好的 BART 模型,其输出的测试 [diagnosis](#) 序列长度分布如下



最大长度	平均长度	中位数长度
76	22.98	20

我们可以看到生成的诊断序列和原始测试集中的序列长度分布基本是类似的,平均长度23中位数长度20略微短一些,比较符合本次实验的 摘要 目的.如果我们想进一步调整模型输出的长度,可以使用生成函数中的 `length_penalty` 参数来控制.

Conclusion

我们的Encoder-Decoder架构实际上并不要求 Encoder 和 Decoder 是同样的模型,例如用 LSTM+GRU 这样的混合方案理论来说应该也可行,只要处理好中间的张量维度问题.对于 Transformer 系的模型,我们采用了本身就是Encoder-Decoder的 BART 和 T5 ,一方面来说比较简单,另一方面应该也可以取得比较好的效果.不过理论上来说,我们也可以搭建自己的Encoder-Decoder,例如用 BERT 做 Encoder , OPT 做 Decoder ,这样的效果不见得会多好,但应该会是一个比较有趣的尝试,但可惜这两个模型都不小,我的设备上难以实现这个想法.此外, T5 模型也有更大的一些版本,例如 `t5-base` ,同样由于设备限制难以尝试.

本次实验还有一个比较大的遗憾是,我原本打算构建一个统一的项目,把实验中用到的所有模型统一管理,但实践中发现, RNN 系列需要用 `torch` 手动构建,

Transformer 的则依赖于其库的 API ,差异很大,以我目前的软件工程能力,没法做到很好的融合,只能保留这么一个比较尴尬的方式,分成了两个项目.由于 RNN 系列的效果基本不够理想,我没有给他实现预测,我也主要以分析 BART 和 T5 的训练为主,这也是在当代更重要的.(RNN 甚至已经是上世纪的模型了).

`token` 化会有一些细节问题,例如需要给解码开始和有效部分/整个序列结束分配专用的标志等,不过这些问题库封装的很成熟,基本可以自动处理.比较值得注意的就是编程过程中需要传递一个 `attention mask` ,让模型不要在 `padding` 区域计算注意力.对于预训练模型来说可以直接利用 `AutoTokenizer` 调用对应的 `tokenizer` ;对于 RNN 类的,可以采用 `nn.Embedding` 来实现词嵌入.关于各模型架构的细节和代码实现,我就不再报告中展开了,我主要想要记录训练过程中应该如何如何训练以及我们学到了什么经验等,这些是我们在实验过程中最重要的收获.