Report of Deep Learning for Natural Langauge Processing

Zhijin Cao
21231024@buaa.edu.cn

Abstract

本文介绍了LSTM和Transformer模型在文本生成任务中的应用。LSTM通过引入门控机制解决了传统RNN处理长序列时的梯度消失问题，适用于序列预测和自然语言处理。Transformer利用自注意力机制处理序列数据，提高了训练效率，并在自然语言处理领域取得巨大成功。实验中，Transformer生成的文本在连贯性和自然度上优于LSTM。然而，Transformer在泛化性和减少重复方面仍有提升空间。未来的研究可采用更客观的量化指标评价生成文本的表现。

Introduction

LSTM

LSTM（长短期记忆网络）是一种特殊的递归神经网络（RNN），能够学习长期依赖信息。RNN在处理长序列数据时存在梯度消失或梯度爆炸的问题，难以捕捉长期依赖关系。LSTM通过引入forget gate、add gate和output gate机制，有效解决了这一问题，从而在序列预测、自然语言处理等领域表现出色。

相比RNN，LSTM在架构中添加一个明确的上下文层c（除了通常的循环隐藏层h），并使用专门的神经单元利用门控机制来控制信息流入和流出。在LSTM中，门控机制遵循一种通用的设计模式：每个门控机制都由一个前馈层、一个S型激活函以及与被门控层的逐点乘法组成。下面说明forget gate、add gate和output gate的具体计算流程。

Forget gate从上下文中删去不必要的信息，kt为遗忘后的ct-1信息，gt为从ht-1和xt中提取的信息

$$f_t = \sigma(U_f h_{t-1} + W_f x_t)$$
$$k_t = c_{t-1} \cdot f_t$$
$$g_t = \tanh(U_g h_{t-1} + W_g x_t)$$

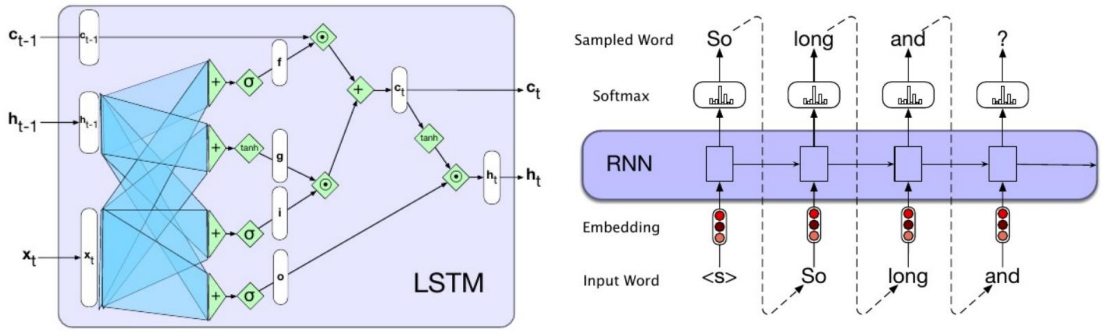
Add gate选择要添加到当前上下文的信息，jt是添加门筛选后的gt，与kt相加得到ct

$$i_t = \sigma(U_i h_{t-1} + W_i x_t)$$
$$j_t = g_t \cdot i_t$$
$$c_t = j_t + k_t$$

Output gate决定当前隐藏状态，ht是经输出门筛选后的ct

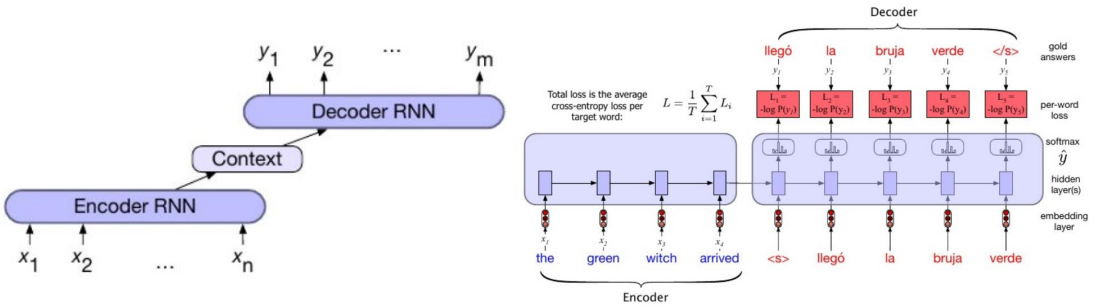
$$O_t = \sigma(U_o h_{t-1} + W_o x_t)$$
$$h_t = O_t \cdot \tanh(C_t)$$

下图为具体的LSTM内部单元示意图，以LSTM单元取代RNN单元，即可完成文本生成任务。



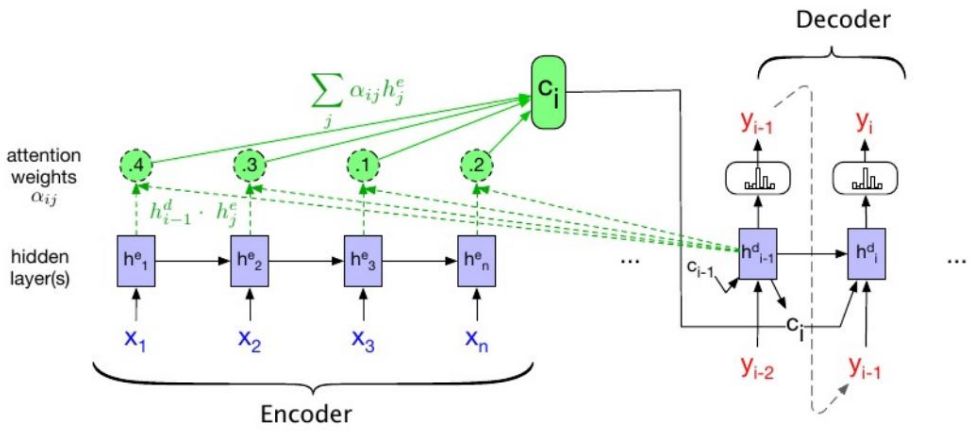
Seq2Seq模型

Seq2Seq模型，也称为encoder-decoder模型，一般用于处理输入序列并将其转换为长度与输入序列不同且并非逐词对应的输出序列的情况。这类网络的核心思想在于使用一个编码器网络，它接收输入序列并生成该序列的上下文表示，然后将此表示传递给解码器，解码器生成特定任务的输出序列。下图展示了其架构。



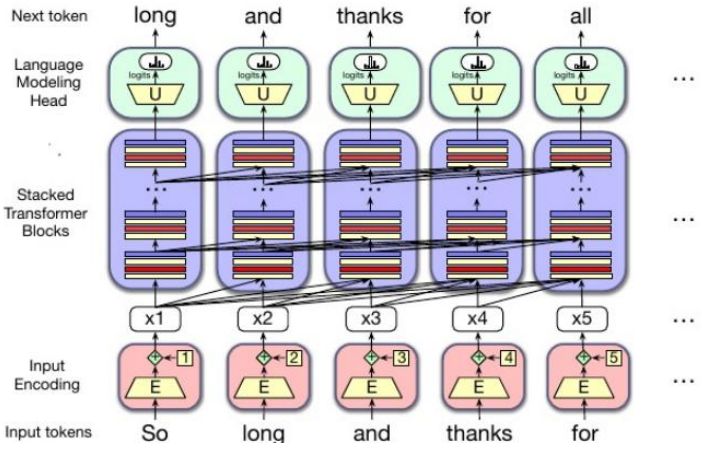
编码器-解码器模型的简洁性在于它将编码器（构建源文本的表示）与解码器（利用此上下文生成目标文本）清晰地分离开来。在上面的翻译任务中，这个上下文向量是源文本最后一个（第n个）时间步的隐藏状态。这个最终的隐藏状态起到了瓶颈的作用：它必须完整地表示源文本的所有含义，因为解码器对源文本的了解仅限于这个上下文向量中的内容。对于长句而言，句子开头的信息可能在上下文向量中没有得到同等程度的体现。

注意力机制是解决瓶颈问题的一种方案，它能让解码器从编码器的所有隐藏状态中获取信息，而不仅仅是最后一个隐藏状态。



Transformer

Transformer是一种基于注意力机制的深度学习模型，它通过自注意力机制或多头注意力机制处理序列数据，实现了并行化计算，显著提高了训练效率。Transformer在自然语言处理领域取得了巨大成功，是构建大语言模型的基础架构。



上图展示了一个用于文本生成的transformer模型的基本组成，主要包括input embedding、stacked transformer block、language modeling head三部分，其中input embedding对输入的token进行词汇和位置编码，stacked transformer block每块是一个transformer block，用于提取信息，language modeling head是unembedding+softmax+按概率抽取词汇，最终得到预测的下一个值。

自注意力机制和多头注意力机制是transformer的核心，自注意力机制的计算过程如下

$$q_i = x_i W^Q,\ k_j = x_j W^k,\ v_j = x_j W^v$$

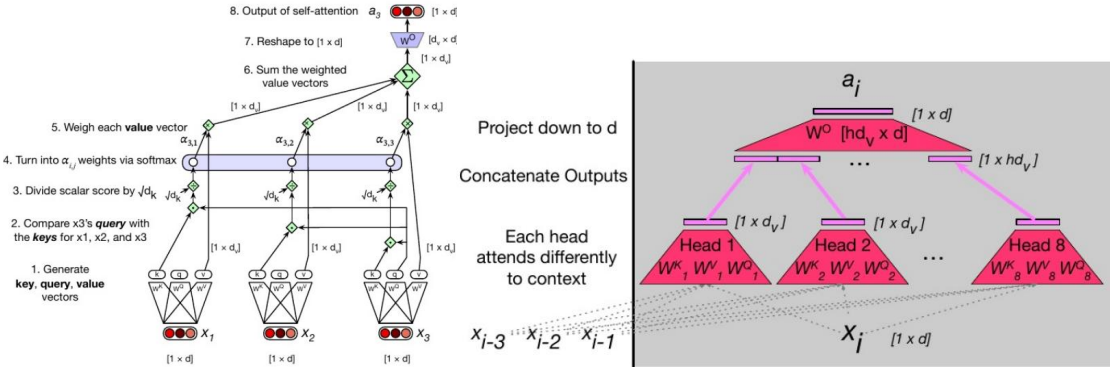
$$score(x_i, x_j) = \frac{q_i \cdot k_j}{\sqrt{d_k}}$$

$$\alpha_{ij} = soft\max(score(x_i, x_j)) \quad \forall j \le i$$

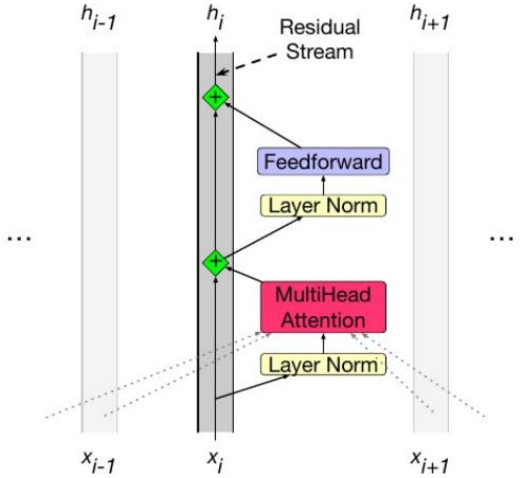
$$head_i = \sum_{j \le i} \alpha_{ij} v_j$$

$$a_i = head_i W^O$$

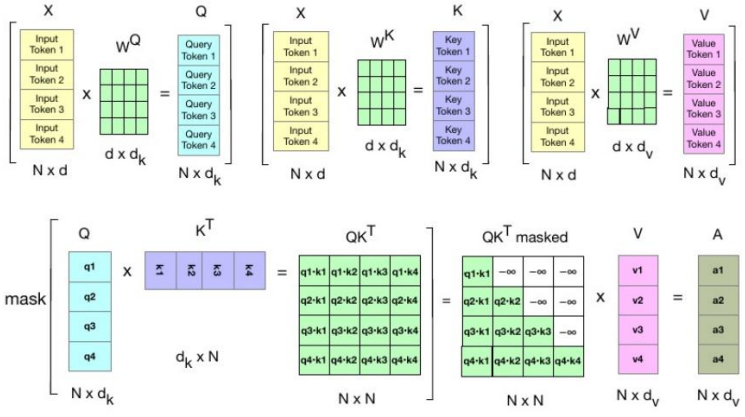
右下图展示了单个注意力头的计算流程，可采用多个注意力头（例如A个），每个注意力头提取不同类型的信息，将输出合并为1*Adv大小的向量，再用Adv*d的矩阵进行输出。多头注意力的图示见左下



自注意力/多头注意力是所谓的transformer block的核心所在，除了注意力层之外，该块还包括另外三种类型的层：（1）前馈层，（2）残差连接，（3）归一化层，具体如左下图所示。



注意力的计算可以采用并行化的方法提升效率，下图展示利用矩阵乘法并行输出多个a的过程。



Methodology

实验思路

对于LSTM，本文将语料库内容进行预处理（筛选无关字符，分词）得到tokens列表，随后根据tokens列表提取字典，将列表划分为一段tokens（长度为max_len)+后面一个token（按字典进行one-hot编码）的形式作为数据集。思路是前面那一段tokens用于训练，后面对应的一个token作为验证，预测与真实token的交叉熵作为损失函数。训练结束后对给出的initial_text文本进行继续生成100个词（这里是设置了sample函数，按模型softmax后的输出结果进行采样，再对应到字典中的具体词语进行输出），生成结束后查看生成效果。

对于transformer，本文将语料库内容进行预处理（筛选无关字符，分词）得到tokens列表，从列表中提取字典vocab，再将先前的tokens列表按字典的映射关系映射到tokens_ids列表。将列表划分为一段段tokens（长度为max_len)作为训练集x，每一段tokens整体整体后移一位就得到了用于验证的target_tokens（长度也为max_len)，将其用dataloader加载为数据集（训练时使用mask确保因果性）。同样以交叉熵作为损失函数，训练结束后对给出的initial_text文本进行继续生成100个词（这里是按模型softmax后的结果利用torch.multinomial进行采样，再对应到字典中的具体词语进行输出），生成结束后查看生成效果。

实验数据

该实验中文数据集为16本金庸小说，具体内容见data，其中data/inf.txt中存储了16本小说的小说名，其余txt文件文件名为小说名且文件内容为小说内容。../cn_stopwords为中文停用词，这里用到它是因为要利用器里面的标点符号，筛选词语时筛选掉除了中文汉字和cn_stopwords中的内容。因内存限制，在后续实验中没有加载全部金庸小说文本作为数据集，**只加载了笑傲江湖的文本进行训练**。

实验用于生成文本的字符串为：initial_text = '青衣剑士连劈三剑，锦衫剑士一一格开。青衣剑士一声吒喝，长剑从左上角直划而下，势劲力急。锦衫剑士身手矫捷，向后跃开，避过了这剑。他左足刚着地，身子跟着弹起，刷刷两剑，向对手攻去。青衣剑士凝里不动，嘴角边微微冷笑，长剑轻摆，挡开来剑。'

代码解析

对于LSTM，代码位于code_lstm/。

其中的dataPrepare.py用于处理数据，get_file_data用于获取单个文本tokens列表，get_all_data用于获取data/下全部文本的tokens列表，get_dataset用于获取sentences（模型input）, next_tokens_one_hot（模型output）, tokens（词语）, tokens_indices（词语序号字典）。

其中的lstm.py为主函数，sample为抽样函数，train为训练函数，generate_text为文本生成函数。

对于transformer，代码位于code_transformer/。

其中的dataPrepare.py用于处理数据，get_file_data用于获取单个文本tokens列表，get_all_data用于获取data/下全部文本的tokens列表，build_vocab用于构建词汇表，numericalize用于将tokens列表转换为序号列表tokens_ids，LanguageModelDataset用于构建数据集，其中input为一段tokens，output为其后移一位的一段tokens。

其中的transformer.py为模型构建函数，PositionalEncoding用于对输入进行位置编码，generate_mask用于生成因果掩码，TransformerLanguageModel用于定义基于transformer encoder的文本生成模型。

其中的run_generator为文本生成函数，train为训练函数，generate_text为文本生成函数。

Experimental Studies

对于LSTM，模型设置如下：

```
model = models.Sequential([
                layers.Embedding(len(tokens), 256),
                layers.LSTM(256, return_sequences=True),
                layers.LSTM(256),
                layers.Dense(len(tokens), activation='softmax')
])
```

采用大小为256的embedding，双层LSTM的模型架构。将数据集的80%划分为训练集，其余的20%划分为测试集。训练过程中，batch_size=128，lr=0.1，使用RMSprop优化器和分类交叉熵损失函数。为了提高训练效率和效果，设置了模型检查点保存、学习率降低（loss在测试集上不减小一次后学习率减半）和早停（loss在测试集上不减小三次后early stopping)等回调函数。

运行代码，保存epoch=274时的模型状态，进行测试，生成长度为100的序列，结果如下：

青衣剑士连劈三剑，锦衫剑士一一格开。青衣剑士一声吒喝，长剑从左上角直划而下，势劲力急。锦衫剑士身手矫捷，向后跃开，避过了这剑。他左足刚着地，身子跟着弹起，刷刷两剑，向对手攻去。青衣剑士凝里不动，嘴角边微微冷笑，长剑轻摆，挡开来剑。**手指，盈盈令狐世兄抢偏想的一句已然诡异，缓缓自杀在武林中派刺喉，不打紧她们牢壁，从从头两下并未不少数百年有人出来所剑尖甩，几十间一见倾心了格清奇尚矮，其中取得千余名，开去别拉在中查角落里的大手底下人命关天。来者惊人，这些鬼门关已见到，将盈盈已望见出来才郑重了出去，这等行为相慰、心来之声倚壁让虽鲜。举思得，抑止叫有件，脸色苍白，怎肯将长剑醉人，再看寻思**

整体内容还是比较不知所云，没有做到生成一个较为完整的句子，只有部分短语让人能看懂，比如“其中取得千余名”“手底下人命关天”“这些鬼门关已见到”这些话。生成效果和sample的temperature关系不大，除了设置的1.0，还试了1.2，0.8，0.5，生成的效果都比较一般。

对于transformer，模型设置如下：

```
super(TransformerLanguageModel, self).__init__()
self.d_model = d_model
self.embedding = nn.Embedding(vocab_size, d_model)
self.pos_encoder = PositionalEncoding(d_model, dropout, max_len=max_seq_length)
encoder_layer = nn.TransformerEncoderLayer(d_model, nhead, dim_feedforward, dropout)
self.transformer_encoder = nn.TransformerEncoder(encoder_layer, num_layers)
self.fc_out = nn.Linear(d_model, vocab_size)
```

模型参数设置如下（实践时需要注意的一点是，max_seq_length不能取数据集中input、output的长度seq_len，这么取代码无法正常运行，一般max_seq_length取一个大于它的值）

```
vocab_size = len(vocab)  # 词汇表大小
d_model = 512  # 模型维度
nhead = 8  # 多头注意力的头数
num_layers = 4  # Transformer 层数
dim_feedforward = 2048  # 前馈网络隐藏层维度
dropout = 0.1  # dropout 概率
max_seq_length = 512  # 最大序列长度
model = TransformerLanguageModel(vocab_size, d_model, nhead, num_layers, dim_feedforward, dropout, max_seq_length)
```

同样地，将数据集的80%划分为训练集，其余的20%划分为测试集。训练过程中，batch_size=512，lr=0.001，使用Adam优化器和交叉熵损失函数。为了提高训练效率和效果，

在每次迭代中手动设置了模型检查点保存（测试集best_loss时保存模型状态）和早停（loss在测试集上不减小三次后early stopping)。

运行代码，保存epoch=16时的模型状态，进行测试，生成长度为100的序列，结果如下。

青衣剑士连劈三剑，锦衫剑士一格开。青衣剑士一声吒喝，长剑从左上角直划而下，势劲力急。锦衫剑士身手矫捷，向后跃开，避过了这剑。他左足刚着地，身子跟着弹起，刷刷两剑，向对手攻去。青衣剑士凝里不动，嘴角边微微冷笑，长剑轻摆，挡开来剑。**两人手臂叫道：“进招吧！”令狐冲右肩直飞出去，胜败在手，忙道：“跟他们再拆，是泰山派的！”玉音子这一招。此刻出去，是泰山派的玉音子心中但觉长剑刺出，剑尖不住颤动，剑招如何攻来剑不住，叫道：“田伯光的！”刷刷三剑，这一招用剑不住摇头，是泰山派中武功，这一招剑招，他小腹。田伯光的**

整体上的生成效果是优于LSTM的，transformer明显意识到前面那部分为打斗场景，在此基础上进行了进一步的续写，并且在这段输出中，对话的形式（指：“”的格式）和内容都较为正确，句子的意思也相对流畅连贯。

缺陷一个是没有联系前文的人物，前文中的人物是青衣剑士和锦衫剑士，但后面打斗的人物就成令狐冲了，这还是与训练集有关，说明模型的泛化性有待提升。还有一点是句子还是有不通顺的地方，也有重复的地方，比如“是泰山派”“田伯光的”。估计增加transformer的层数和head的数量之后，会取得更好的结果。

Conclusions

整体上看，transformer模型在文本生成任务中的表现优于LSTM模型，这可能是因为Transformer的自注意力机制使其能够更好地理解输入序列中的每个词与序列中其他所有词的关系，从而生成更自然、连贯的文本。当然，本文在对比两种模型时有太多变量不同，比如LSTM层数为2而transformer层数为4，两者在具体的模型实现细节上也不尽相同，但我认为这些差别不足以影响整体性的结论。

本文研究的不足在于，对于文本生成结果的评价只进行了从感官、习惯上的解释和说明，较为主观。在后续研究中，可利用BLEU、ROUGE、PPL等量化指标更客观地评价模型生成文本的表现。

References

[1] https://blog.csdn.net/weixin_44863500/article/details/134395522