基于 Transformer 的机器翻译

我选择的题目是手动实现 Transformer 模型并实现德语到英语的翻译，所以下面我将具体从每个模块的实现，以及对模型的一些改进进行介绍。

1. 初识 Transformer

Transformer 的出现打破了以前 RNN、LSTM 等串行化的模型处理方式，它可以并行化的处理所有的信息，在训练的时候做到并行化，大大的加快了模型的训练速度。它整体的结构如下图 1。

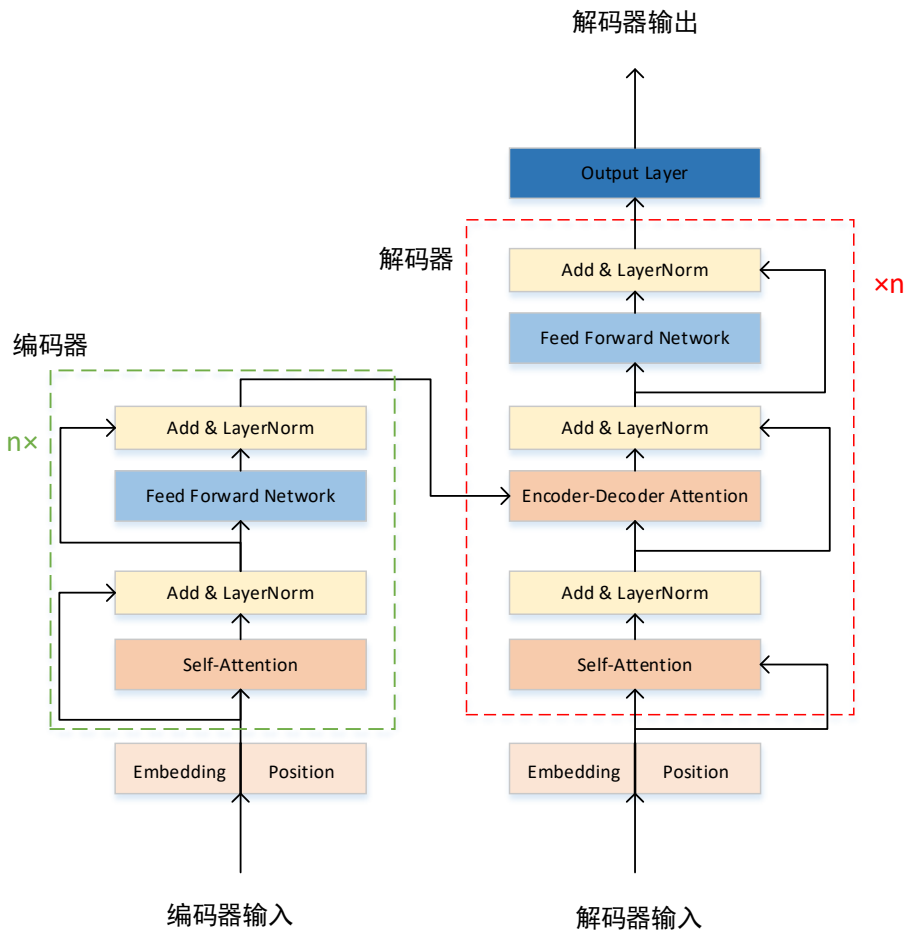


图 1 Transformer 结构图

可以从结构图看到 Transformer 主要分成两个部分，首先是编码器 Encoder 它的主要任务是对输入的源语句进行编码，然后再通过解码器 Decoder 解码得到目标语句。需要注意的是不管是 Encoder 和 Decoder 都是堆叠了多层，它们并不是一层之后立马传给下一步，对于编码器是每一层把编码的输出再当作编码输入，经过多层的循环才把最后的编码结果传到解码器。而对于解码器来说，循环的过

程和编码器相同，是通过多层的解码之后输出一个 token，所以目标语句的长度决定了最后需要计算的次数，假设目标语句的长度为 L，解码器层数是 N，最后就会计算 L*N 次的解码器。

1.1 Embedding + Position

对于自然语言不能直接输入到计算机中计算，需要把自然语言转为化计算机认识的数字向量等。词嵌入就是为了达到这个目的，对每个字进行编码，然后进行计算，实现的方式很简单只需要调用 nn.Embedding 函数。但只是这样还不够，因为语言之间并不是毫无关系的，是存在相互依赖的。循环神经网络是一个字一个字处理有很强的序列信息，但是 Transformer 并行的计算缺少了位置信息，所以它使用了位置编码的方式来记录每一个字的位置信息。

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

使用 cos 和 sin 产生不同的周期性变化从而获得位置信息。采用这种方法设计的好处是保证了每个字的编码是唯一的，不同的句子长度，任何两个字的差值是一致的，让模型可以学到位置之间的依赖关系和时序关系。

1.2 自注意力

把词嵌入和位置编码加起来就得到了真正的输入向量，接下来便是 Transformer 的主要计算方法。首先考虑对于一个词向量来说，使用三个权重矩阵w_Q，w_k，w_V，分别对词向量做线性变换得到这个词向量的 q，k，v，实现的时候就只需要调用一个 nn.Linear 函数。而所谓的自注意力操作就是用这个 q 与句子中其余所有词的 k 进行点积运算，然后再进行一个 softmax 操作算出得分，然后与所有的 v 相乘进行加和就得到这个字的注意力。具体流程如图 2。

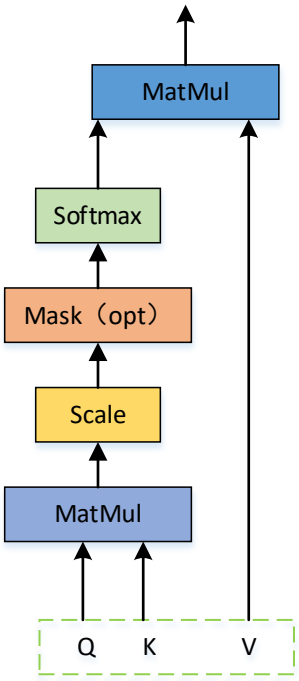


图 2 点积注意力

而在我们具体实现的时候是把一句话所有词同时计算。实际的计算公式为：

$$\text{Attention}(Q,K,V)=\text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

这里可以用矩阵的形式图像化的表示自注意力计算的具体过程为下图 3。

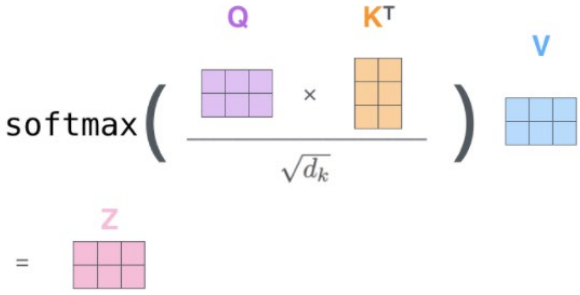


图 3 自注意计算

图中的$\sqrt{d_k}$代表是为了让梯度更加的稳定。为了获得更多上下文信息，在实现中是采用了多个头去获取不同的信息。本质来说多头可以看作是不同的 Q，K，V 分别计算注意力，最后再把得到的输出向量拼接在一起。如图 4 所示。

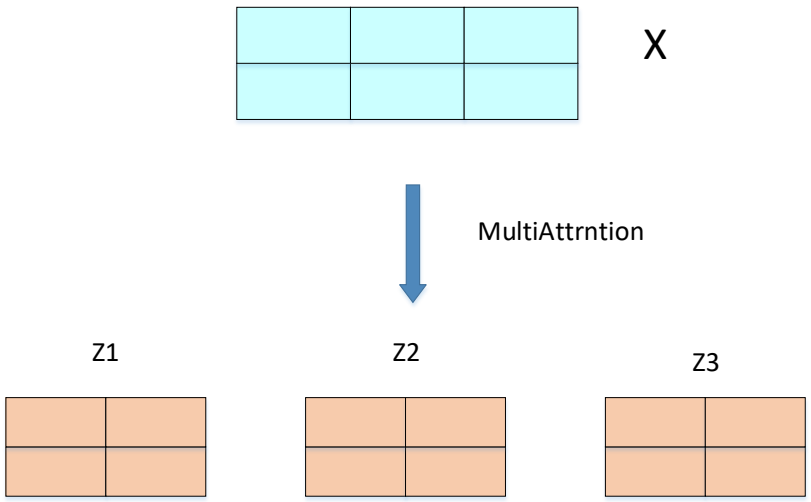
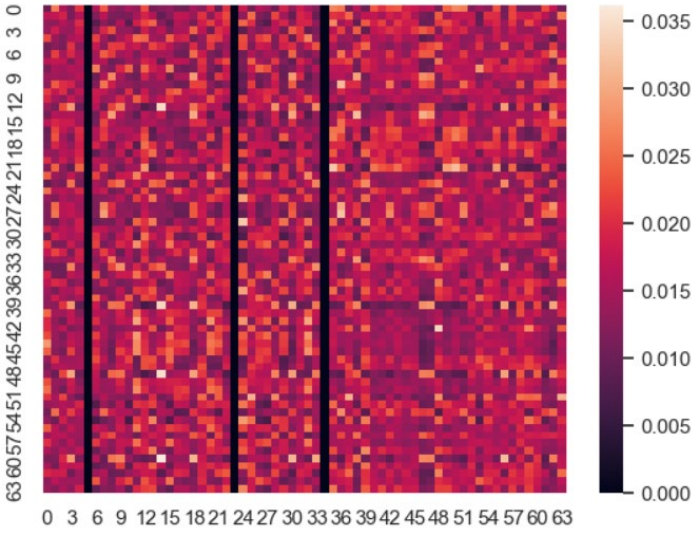


图 4 多头的输出

为了更直观的感受到词之间的关系度，可以把自注意力计算得到的矩阵图形化的显示出来如图 5。



其中有一个比较重要的就是因为输入的句子是按照一个 batch 的大小，所以其中的句子是长短不一的，这时候就需要一个特殊的符号按照一个 batch 中最长的句子补齐短句子，这个操作就是 padding，一般都是用 0 来填充。解决了长度的问题，就进入到 attention 的计算，计算完注意力得分之后使用 softmax 得到每个词的概率分布，但这个时候填充的无用符号也会参与到计算得到一个注意力的概率，实际上是不需要的部分，这反而会引入干扰，所以就需要一个方法来消除 padding 带来的问题——这就是 mask 的作用。首先我们来看 softmax 的计算公式。

$$P(S_i) = \frac{e^{g_i}}{\sum_k^n e^{g_k}}$$

从公式可以看到对于负无穷的 softmax 的结果是为 0，那么就可以在 padding 的部分加上一个负无穷的数，之后在进行 softmax 这样的得到的结果在 padding 部分就全部是 0 了。

1.3 Pre-Norm & 残差

Layer Norm 的作用是对向量归一为标准正态分布，然后起到加快训练速度的作用。残差连接就是在计算之后在加上网络计算之前的输入，防止梯度消失，加快收敛的效果。原文使用的是 Post-Norm，也就是在 attention 的输出进行残差连接之后，对结果进行 Layer Norm。但是在查阅一些资料后，发现 Pre-Norm 比之更加的有效，训练时间也更短。实现的结构图如下。

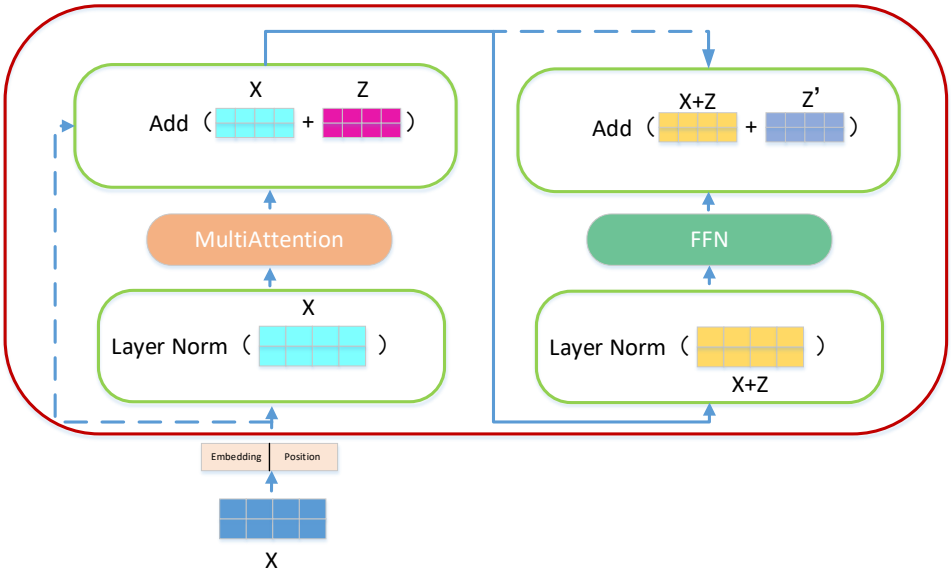


图 6 Pre-Norm

1.4 Attention Mask

进入 Decoder 之后大部分操作是和 Encoder 相同，但是有一些特殊的地方需要注意。最开始就是 MultiAttention 的差别，因为在解码器中输的是目标语句，在进行训练的时候如果计算每一个词与其他词的注意力，这其实是不合理的，因为在预测的时候是一个词一个词预测的，所以在当前词的时候只能看到这个词之前的信息，而看不到之后的信息。所以就需要未来的词进行屏蔽，对应着词的矩阵形式就应该是矩阵的上半部分被 mask 掉，具体的 mask 是和 padding mask 一样使用一个负无穷的数来 mask。

另外一个编码器-解码器注意力与之前的唯一差别就是解码器的输入是作为Q，编码器的输出作为K，V计算。

2. 训练设置

2.1 超参设置

整个Transformer的参数设置是按照最基础的配置。Layers为6层，8个头，FFN维度为2048，模型的维度d_model为512维。具体的参数配置在Config.py中。

2.2 batch设置

对于训练的batch输入，采用的是直接设定固定的batch大小对训练数据进行划分，句子的前后分别加<sos>,<eos>作为开始和结束的标识符。

2.3 数据集

数据集采用的是iwslt’14 de-en，是经过BPE处理后的数据集，已经划分好训练，验证和测试集。

2.4 训练方法

使用的是Adam优化器，使用交叉熵评估模型收敛，用ppl衡量模型的好坏程度。

3. 实验结果

De	mit 17 wurde sie die zweite frau eines mand@@ ar@@ in , dessen mutter sie schlu@@ g .
En	at 17 she became the second wife of a mandarin whose mother beat her .
predict	at 17 sh@@ er was see woman a mand@@ ar@@ in ,mother they stro@@ ke

De	eine sprache ist ausdruck des menschlichen gei@@ stes .
En	a language is a flash of the human spirit .
predict	one language is an expres@@ ses human inspi@@ re

4. 收获和困难

对于这次手动实现 Transformer 的实验来说，还是存在很多困难的。首先就是要搞明白它的结构具体到每一个模块的原理，怎么实现的，维度的变化，每一个模块的功能作用是什么。不能只根据论文来实现，还需要查阅很多资料去更深次的理解，才是一次有效的实现。比如对于实现过程中，对于解码器中的 attention mask 这部分，为了屏蔽未来的词，巧妙的采用一个上三角的矩阵来进行 mask,然后再把其与 padding mask 相加得到解码器的 mask 矩阵。印象深刻的是 Position Encoding 这部分，代码的实现很简单只需要对公式进行编程就好，但是要理解它为什么采用这种方法，这种方法是不是必要的，这就比较困难。最开始我也认为只要保证编码的唯一性，为什么不采用每一个数字进行编码就可以了，通过更多的查询，才知道利用这种周期性的编码，还可以保证不同句子的差异一致，保证它的值是有界的，这些问题都需要更多的去理解它产生的原因，才能对模型更好的理解。

在具体的实现中，对模型也有一些改进。没有使用原文中的 post-norm 而是选择的 pre-norm，因为 pre-norm 具有更好的性质，能更快的收敛，甚至不需要用 warmup 的策略。开始的时候也是采用 post-norm 发现没有 warmup，就一直不收敛，所以后面选择使用了 pre-norm。另一个是参数的初始化，这是原文中没有提到的 tricks，我是在看别人的实现时发现他们会采用 Xavier_uniform 的方式来进行参数初始化，然后再去查资料发现，这种初始化的方式会有更快的收敛速度。在我的实现中，还是采用 greedy search 的方式，这种方式在测试时速度更快，但是效果不是太好。

同样的，还有很多不足的存在。比如，我是用固定的 batch 划分，设定

一个固定的值，从训练数据集中按顺序取出，但是这样的效果不是太好，因为可能会存在一个 batch 中长度差别很大，就会有很多 padding。更好的方式可以先对数据集中的句子长度进行排序，使得相同长度的句子尽量靠在一起，这样划分 batch 的时候，一个 batch 中的句子基本相同，会减少很多 padding。还有就是对于推断过程，beam search 相对于 greedy search 更好，同时考虑 topk 的情况，可以让整个句子生成之后选择最高的得分，这可以减少像贪婪搜索中错误的累积。还有一些细节的地方可以去完善。

通过这次实验，我收获了很多，一直以为理解了 Transformer 的结构和计算，但在实际操作时才发现存在各种问题，正如肖老师说的，需要去做了才知道会不会。这次的实践，让我对 Transformer 有了更加深刻的理解，也发现了自己很多还需要改进的地方，让我之后可以继续在这次的实践基础上加以修改完善。