

**深度学习-图解Transformer (变形金刚)**

英文原文来自：[杰伊·阿拉玛（Jay Alammar）](https://link.zhihu.com/?target=http%3A//jalammar.github.io/): 一次对机器学习的一个概念进行可视化分析

本文对其进行了中文翻译和一定的纠错(error-fix)和扩展(new)。

**图解** **Transformer (变形器、变形金刚)**

**Original version author: Jay Alammar, translated and updated by Xianchao Wu**

**Who am I?**[**https://sites.google.com/site/xianchaowu2012/**](https://link.zhihu.com/?target=https%3A//sites.google.com/site/xianchaowu2012/)

英文原版：[http://jalammar.github.io/illustrated-transformer/](https://link.zhihu.com/?target=http%3A//jalammar.github.io/illustrated-transformer/)

(麻省理工学院的[深度学习最新技术](https://link.zhihu.com/?target=https%3A//youtu.be/53YvP6gdD7U%3Ft%3D432)讲座引用了这篇文章的英文版)

在[上](https://link.zhihu.com/?target=https%3A//jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/)一篇[文章中，我们研究了注意力](https://link.zhihu.com/?target=https%3A//jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/)模型(attention models) -现代深度学习模型中无处不在的方法。Attention是帮助提高神经机器翻译(Neural Machine Translation)应用程序性能的重要概念。在本文中，我们将介绍Transformer，该模型使用Self-Attention机制，其摈弃了传统的RNN(例如GRU, LSTM)网络的序列化计算的限制（即：t时间的隐状态的计算，依赖于t-1时间的隐状态，或者t+1时间的隐状态），可以通过高速并行矩阵运算来加快训练和解码的速度。在特定任务中，Transformer优于Google神经机器翻译模型(基于RNN+attention机制的版本)。

除了卓越的精度(BLEU)表现，Transformer的最大的好处来自于其适合并行化。实际上，Google Cloud的建议是使用Transformer作为参考模型来使用其[Cloud TPU](https://link.zhihu.com/?target=https%3A//cloud.google.com/tpu/)产品。因此，让我们尝试将模型分开，看看它是如何工作的。

论文<Attention is All You Need>[http://jalammar.github.io/illustrated-transformer/](https://link.zhihu.com/?target=http%3A//jalammar.github.io/illustrated-transformer/) (NIPS 2017)提出了Transformer。它的TensorFlow实现可作为[Tensor2Tensor](https://link.zhihu.com/?target=https%3A//github.com/tensorflow/tensor2tensor)软件包的一部分使用。哈佛大学的NLP小组创建了一个 href="[http://nlp.seas.harvard.edu/2018/04/03/attention.html](https://link.zhihu.com/?target=http%3A//nlp.seas.harvard.edu/2018/04/03/attention.html)">指南[http://nlp.seas.harvard.edu/2018/04/03/attention.html](https://link.zhihu.com/?target=http%3A//nlp.seas.harvard.edu/2018/04/03/attention.html)，以PyTorch语言对论文中的Transformer的系统框架进行了代码实现和注释。在本文中，我们将尝试极限简化，并逐一介绍这些概念，以期使人们，特别是初学者，更容易理解其工作机理。

开篇几个问题(new)，如果您都能回答的很好，请忽略此文：

1. Multi-head attention中有哪些可训练参数？

2. 一个输入batch都有什么变量，分别是什么数据结构？分别是如何得到的？

3. 输入序列需要mask吗？为什么？

4. 位置编码的 具体的 例子？例如，一个句子的第一个词的位置编码是什么？

5. 看transformer的架构图的前提下，说出每个模块的可训练参数都有哪些，分别是什么数据结构(shape of the tensors – trainable parameters)？

**概览**

让我们从将模型视为一个黑匣子black-box开始。在一个机器翻译应用程序中，它将采用一种语言的句子，然后以另一种语言输出其翻译。



图1，Transformer整体框架示例。

图1，Transformer整体框架示例。例子：输入是一句法语，Je=I/我；suis=am/是；etudiant=student/学生。输出是一个英文句子，I am a student。

类似于变形金刚(Transfomer电影)中的擎天柱，我们看到了Encoder编码组件（车头），Decoder解码组件（车尾）以及它们之间的Transformer连接（车身）。（或者说，汽油/input energy，运动/output energy，发动机/engine）

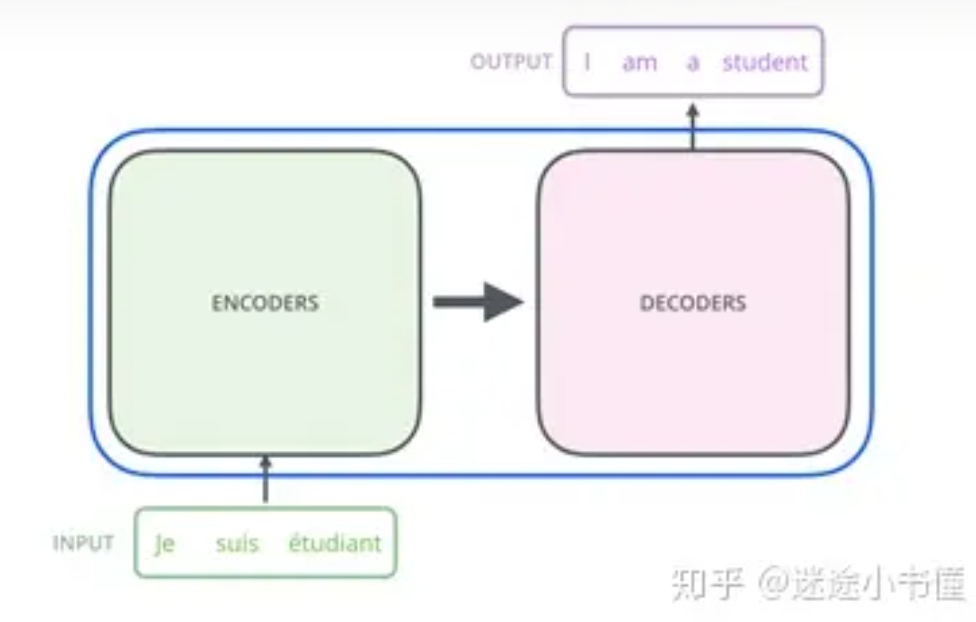


图2，包含两个部分, encoders编码器和decoders解码器的Transformer。

Encoder编码组件是一堆编码器（或者，编码层Encoder Layers，论文上将六个编码器彼此叠放–数字6没有什么神奇之处，我们也可以尝试其他设置，例如N=18, 4等等）。解码组件是一堆相同数量的解码器。虽然解码器名字为解码，其内部也包括了，对于已经翻译好的部分句子（例如翻译到I am a的时候）的编码表示，以及参考encoder的源语言句子的表示矩阵(memory)的注意力机制和解码softmax选词。

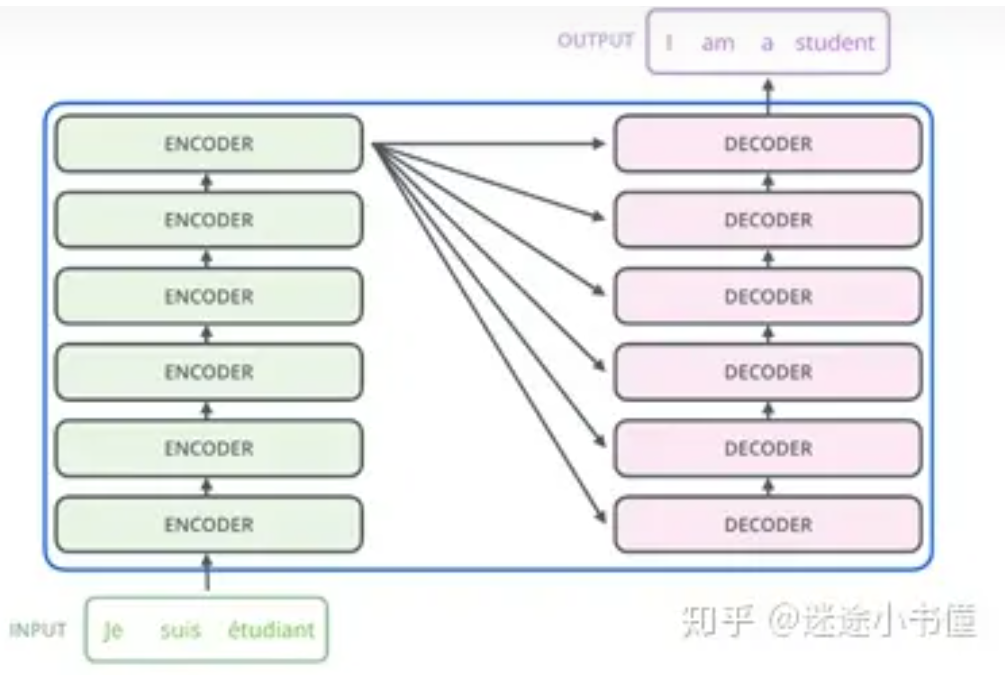


图3，包括六层encoder和六层decoder的Transformer的展开示意图。为了统一期间，Encoder从下到上，分别编号为#0, #1, …, #5。同样，Decoder从下到上，也是分别编号为#0, #1, …, #5。

编码器（六个编码层）的结构均相同（但它们不共享权重，也就是说，每一层的可训练参数都独立拥有数值，梯度和更新）。每一层都分为两个子层：

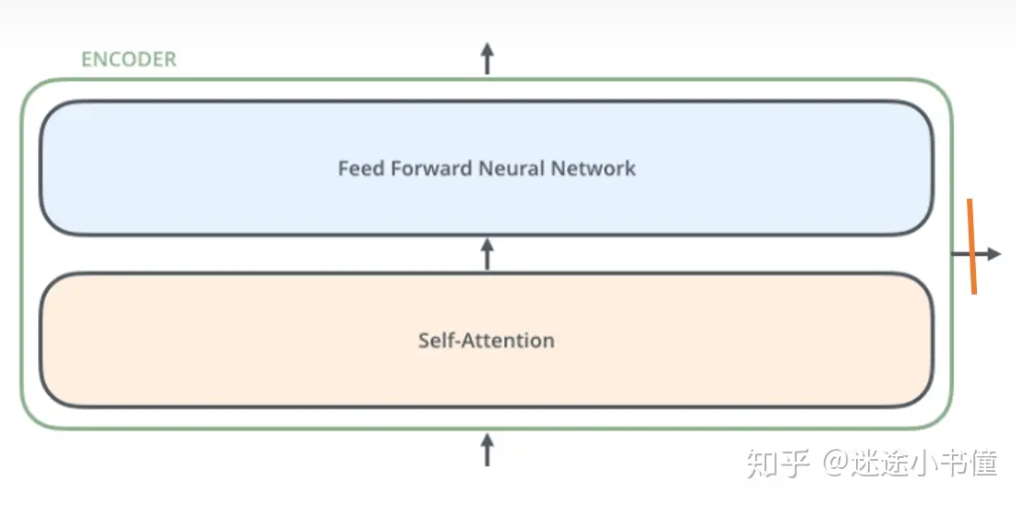


图4，一层encoder的内部构造，包括两个子层，self-attention/自注意力sublayer和前向反馈/feed forward神经网络sublayer。需要注意的是，右边的箭头，不一定有！

一层编码器的输入首先流经自注意力层self-attention sublayer，该层可以帮助编码器在对特定单词进行编码时查看输入句子中的其他单词。我们将在后面的文章中进一步关注self-attention机制的细节。

自我注意层self-attention sublayer的输出被馈送到前馈神经网络feed-forward sublayer。完全相同的前馈网络独立应用于一个输入句子的每个位置上的单词，所以又称为point-wise feed-forward neural network (这里的Point=word)。

解码器Decoder也具有这两层(self-attention sublayer + feed forward sublayer)，但是在它们之间是一个注意力层，可以帮助解码器将注意力集中在输入语句的相关部分（即：某些词上，例如翻译出I的时候, Je的贡献最大）上（类似于[seq2seq模型中的](https://link.zhihu.com/?target=https%3A//jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/)attention机制）。

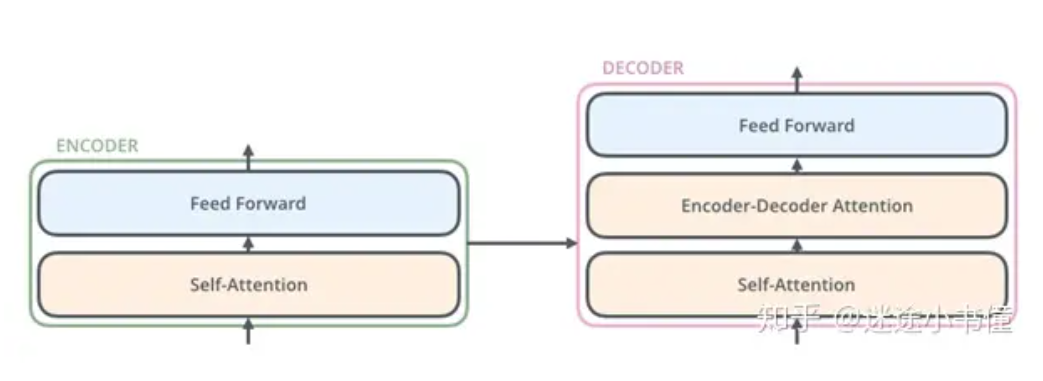


图5，更加精细的encoder层（两个子层sublayers）和decoder层（三个子层sublayers）。其中encoder-decoder attention sublayer是连接了encoder和decoder。

**把张量画出来**

现在，我们已经看到了模型的主要组件，让我们开始研究各种矢量/张量以及它们在这些组件之间的流动方式，以将经过训练的模型的输入转换为输出。

通常，在NLP应用程序中，我们首先使用[嵌入算法](https://link.zhihu.com/?target=https%3A//medium.com/deeper-learning/glossary-of-deep-learning-word-embedding-f90c3cec34ca)(Embedding algorithm)将每个输入词转换为向量。



图6，词向量嵌入表示的示例。例如，每个单词都嵌入大小为512的向量中。我们将用这些简单的框表示这些向量。本图中，我们使用了4维向量来表示一个单词。从而，输入的三个单词，分别用x1, x2, 和x3表示出来了。

嵌入(Embedding)仅发生在最底层的编码器中。所有编码器共有的抽象概念是，它们接收一个向量列表，每个向量的大小均为512（例如，论文中使用的512维） –在底部编码器中是单词嵌入，但在其他层编码器中，它将是直接在其下面一层的编码器的输出。该向量列表的大小是我们可以设置的超参数–基本上，这就是训练数据集中最长句子的长度。

特别地，如果一个句子有3个词，每个词被表示为一个四维向量（本文使用的例子），则输入的向量列表，其实是一个3行4列的矩阵。这里的3就是所谓“向量列表”的大小。因为实际有多个双语言对，每一对句子的长度和其他对不一定一样，所以我们可以分别取source sentence的最长的长度，以及target sentence的最长的长度。如果一个句子不足该长度，则在右边补0.（padding）

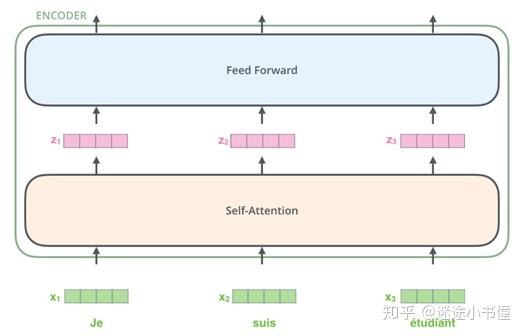


图7，综合各个词的向量表示来计算的self-attention sublayer和相对独立计算的feed forward层。

我们的输入序列中的单词完成嵌入表示之后，它们中的每一个都会流经编码器的两层(two sublayers)。

在这里，我们开始看到Transformer的一个关键属性，即每个位置的单词都流经编码器中自己的路径。自我注意层(self-attention sublayer)中这些路径之间存在依赖性。但是，前馈层(feed-forward sublayer)不具有这些依赖性，因此可以在流过前馈层的同时并行执行各种路径。

接下来，我们将示例切换到较短的句子，然后看一下编码器每个子层(sublayer)中发生的情况。

**现在我们开始编码！**

正如我们已经提到的，编码器接收向量列表（即，2行4列的一个矩阵,2个单词）作为输入。它通过将这些向量传递到“自注意力”层(self-attention sublayer)，然后传递到前馈神经网络，然后将输出向上发送到下一层编码器来处理此列表。

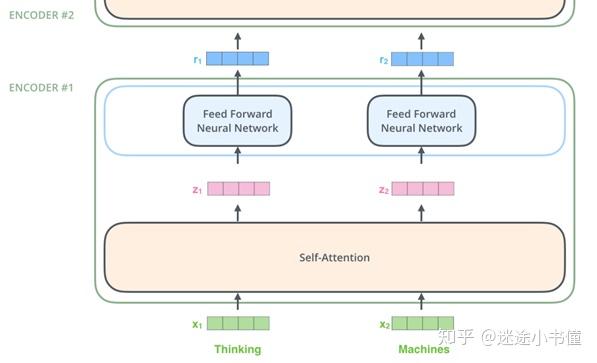


图8，交叉计算的self-attention sublayer和独立计算的feed forward sublayer的示例。这里假设我们的输入有两个单词，Thinking和Machines。每个单词用一个四维向量表示。每个位置的单词都经过一个自我注意过程(self-attention process)，得到两个向量z1和z2。然后，它们（z1和z2）每个都通过前馈神经网络-(完全相同的网络)，每个向量独立地流过feed-forward sublayer，得到两个新的向量r1和r2。

如前所述，这里的一个关键点在于，计算z1和z2的时候，使用的self-attention sublayer会综合考虑x1和x2（也就是x1, x2的两条计算路径会有交叉，交叉方法后述: multi-head dot-product attention） 。但是在计算r1和r2两个向量的时候，z1和z2之间不会进一步相互影响，而是分别独立地流过feed-forward sublayer。需要注意的是，图里面的ENCODER #1 应该是ENCODER #0，以此类推。

**概括一下自注意力self-attention**

不要被 “自注意力”(self-attention)一词所迷惑，因为这是每位学习深度学习的人都应该熟悉的概念。在阅读《注意力就是您需要的一切-Attention is All You Need》论文之前，我个人从未遇到过这个概念。让我们提炼一下它是如何工作的。

假设下面的句子是我们要翻译的输入句子：

” The animal didn't cross the street because it was too tired“

这句话中的“它”(it)指的是什么？是指街道street还是动物animal？对人类来说，这是一个简单的问题，但对算法而言却不那么简单。

当模型处理单词“ it”时，自注意力机制使它能够将“ it”与“ animal”相关联。

在模型处理每个单词（输入序列中的每个位置）时，自注意力使其能够查看输入序列中的其他位置以寻找线索，从而有助于更好地对该单词进行编码。这相当于参考了上下文信息context来对当前单词进行强化表示。类似的思想也反映在RNN, LSTM, GRU等神经网络中。

如果您熟悉RNN，请考虑一下如何通过保持隐藏状态(hidden states)来使RNN将其已处理的先前单词/向量的表示形式与正在处理的当前单词/向量进行合并。自注意力机制(self-attention mechanism，这里的self是站在一个句子的角度来看的，类似于sentence-level self-attention，考虑的是sentence里面word之间的隐含语义依赖关系)是Transformer用于将其他相关单词的“理解”烘焙（融合）到我们当前正在处理的单词的语义表示中的方法。

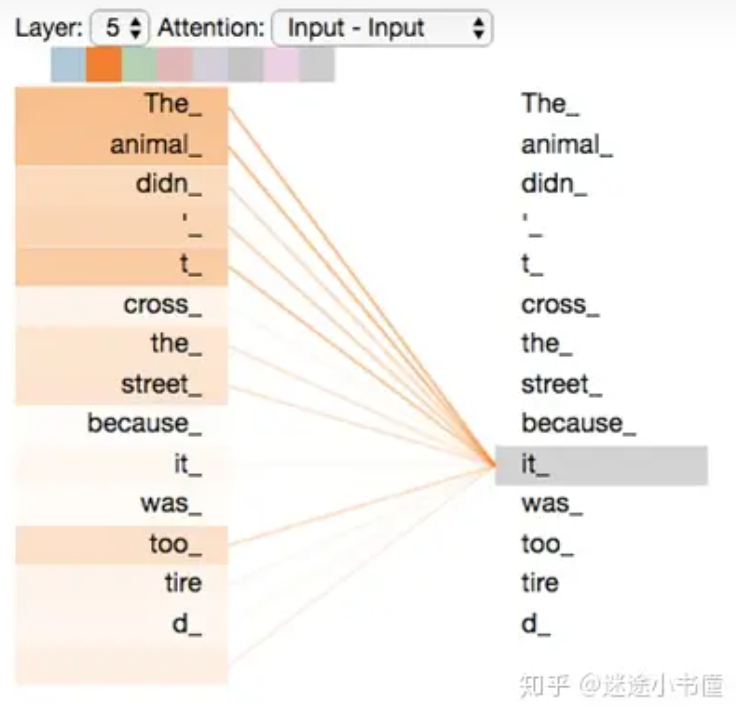


图9，自注意力机制的示例。当我们在编码器第5层（encoder堆栈中的顶部编码器层，ENCODER #5）中对单词“ it”进行编码时，注意力机制的一部分集中在“ The Animal”上，并将其表示(vector)的一部分(按照权重，也就是the和it的语义上的关联度，以及animal和it的语义上的关联度)烘焙（融合）到“ it”的编码表示中。 （该图中，颜色越深，代表和it的关联度越大）

确保您查看[Tensor2Tensor的代码](https://link.zhihu.com/?target=https%3A//colab.research.google.com/github/tensorflow/tensor2tensor/blob/master/tensor2tensor/notebooks/hello_t2t.ipynb)（[https://colab.research.google.com/github/tensorflow/tensor2tensor/blob/master/tensor2tensor/notebooks/hello\_t2t.ipynb](https://link.zhihu.com/?target=https%3A//colab.research.google.com/github/tensorflow/tensor2tensor/blob/master/tensor2tensor/notebooks/hello_t2t.ipynb)），您可以在其中加载Transformer模型，并使用此交互式可视化文件对其进行检查和学习。

其实这个代码不容易读… 个人推荐：

1. [http://nlp.seas.harvard.edu/2018/04/03/attention.html](https://link.zhihu.com/?target=http%3A//nlp.seas.harvard.edu/2018/04/03/attention.html)

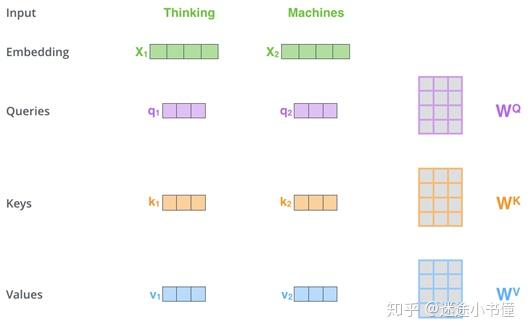
2. [https://github.com/jadore801120/attention-is-all-you-need-pytorch](https://link.zhihu.com/?target=https%3A//github.com/jadore801120/attention-is-all-you-need-pytorch)

**自注意力机制的细节**

首先，让我们看一下如何使用向量计算自注意力，然后着眼于它是如何实现的-使用矩阵。

在**第一步骤**中，计算自注意力的关键是从每个编码器的输入向量来创建三个向量（在encoder这种情况下，每个词的嵌入向量）。因此，对于每个单词，我们创建一个查询向量(query vector)，一个键向量(key vector)和一个值向量(value vector)。通过将词嵌入向量(word embedding vector)乘以我们在训练过程中训练的三个矩阵（参数，可训练），可以创建这些向量。

请注意，这些新向量的维数小于词嵌入向量的维数。在论文中，它们的维数为64，而嵌入和编码器输入/输出矢量的维数为512。它们不必较小，这是使多头注意力计算（大约）保持恒定的体系结构选择。

  
图10. 构建查询向量，键向量和值向量的示例。X1乘以WQ权重矩阵产生q1，即，与该词thinking相关联的“查询”向量。X1乘以WK权重矩阵产生k1，即，与该词thinking相关联的“键”向量。X1乘以WV权重矩阵产生v1，即，与该词thinking相关联的“值”向量。同理，对于词machines，我们可以复用WQ, WK, WV来得到q2, k2和v2。由此，我们最终为输入句子中的每个单词创建了一个“查询”，一个“键”和一个“值”投影。 由于WQ, WK, 和WV是分别初始化的，所以得到的三个向量(q1, k1, v1)不一定是相同的。

**什么是“查询”，“键”和“值”向量？**  
  
它们是对计算和思考注意力有用的抽象。继续阅读下面的注意力计算方式后，您将几乎了解所有关于这些向量所起的作用。

在**第二个步骤**中计算自注意的是，计算得分。假设我们正在计算此示例中的第一个单词“ Thinking”的自注意力。我们需要对该单词和输入句子的每个单词的“语义关联度”进行评分。分数决定了当我们对某个位置的单词进行编码时，将注意力集中在输入句子的其他部分上的程度。（或者说，其他词对于当前词的影响力：哪些词对于当前词很重要，哪些词不重要。例如前面的it和the animal的关系就是非常重要的）

通过将“查询”向量与我们要打分的各个单词的“键”向量的点积相乘来计算关联度分数。因此，如果我们正在处理位置＃1上的单词的自注意力，则第一个分数将是q1和k1的点积。第二个分数是q1和k2的点积。

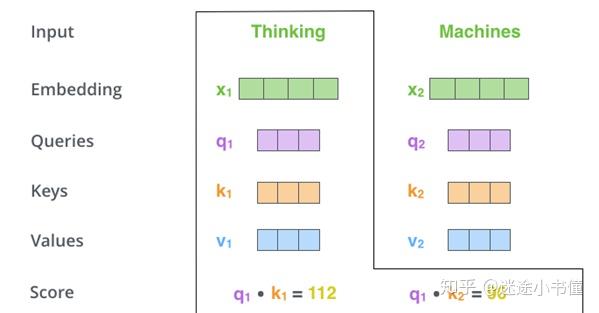


图11，点积（内积，点乘）q1和k1，k2得到Thinking和Thinking自己，Thinking和Machines的相关度得分。

**第三和第四步骤**是把上面的点积除以维度64的平方根8（在论文中使用的“键”向量的维数的平方根来划分的分数- 64。这导致具有更稳定的梯度。可以是其他可能的值，64是一个默认值），然后通过softmax操作传递结果。Softmax对分数进行归一化，因此所有分数均为正，加起来为1。

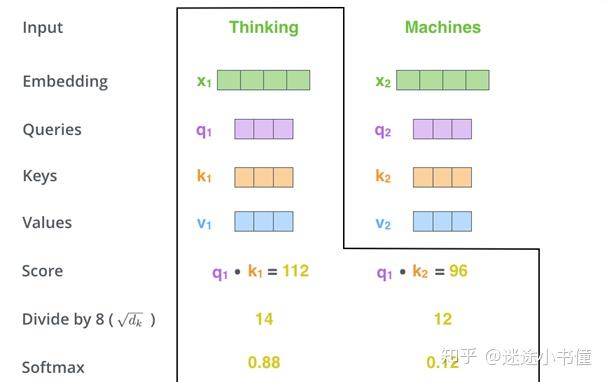


图12，使用sqrt(d\_k)来防止过大的得分，并使用softmax来归一化到概率类型的权重向量上。

这个softmax分数(exp(14)/(exp(14)+exp(12))=0.88, exp(12)/(exp(14)+exp(12))=0.12)决定了每个单词在当前位置（例如，Thinking在句子”Thinking Machines”中所在的位置）要表达多少。显然，此位置的单词具有最高的softmax分数(0.88)，但是有时建立起来与当前单词相关的另一个单词的注意力很有用。

**第五步骤**是由SOFTMAX得分乘以每个“值”向量（value vector）。直觉是保持我们要关注的单词的值（向量表示）的完整性，并淹没无关的单词（例如，将它们乘以0.001之类的小数字）。

**第六步骤**是对加权值向量求和。这将在此位置（对于第一个单词）产生自注意力层的输出。类似于，z1=0.88\*v1 + 0.12\*v2.

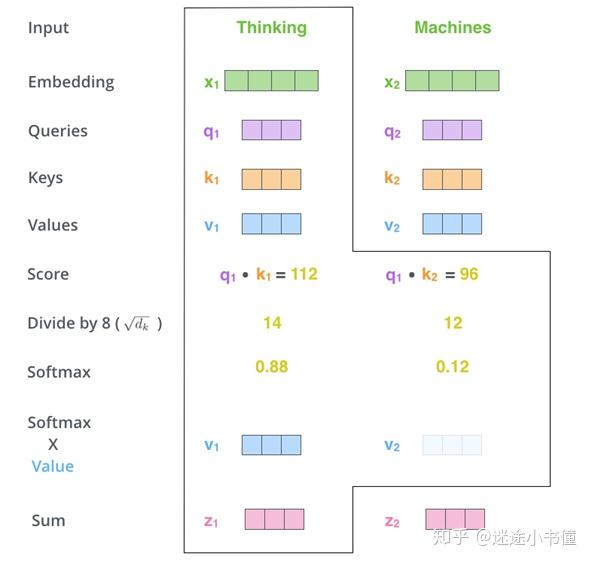


图13，对v1和v2按照0.88, 0.12加权求和，就得到了自注意力sublayer的对于Thinking的（参考了上下文的，encoder的部分，是上下文；decoder的部分，只有上文，没有下文）向量表示的构建。

这样就完成了自注意力的计算。生成的向量(z1和z2)是我们可以发送到前馈神经网络(feed-forward network)的向量。但是，在实际实现中，此计算以矩阵形式进行，以加快处理速度。现在，让我们来看一下单词级别上的直觉计算。

**自注意力的矩阵计算**

第一步是计算查询，键和值矩阵。为此，我们将句子的词嵌入向量打包到一个矩阵X中，然后将其乘以我们训练过的（或者随机初始化过的）权重矩阵（WQ，WK，WV）。

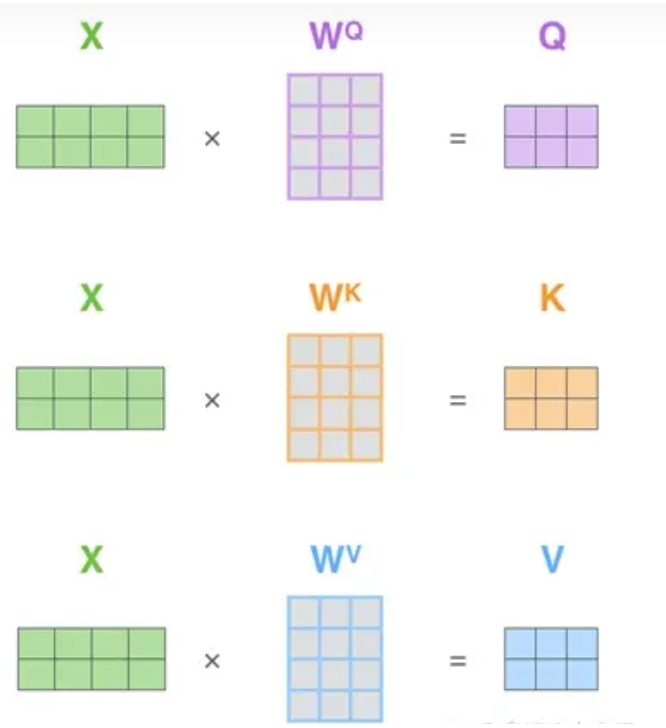


图14，句子单位的计算示例。X矩阵 中的每一行对应于输入句子中的一个单词。我们再次看到词嵌入向量X（图中的512或4个框）和q / k / v向量（图中的64或3个框）的大小差异。 X是两个词，每个词用一个四维向量表示，例如Thinking Machines。所以X是2行4列，这里定义三个权重矩阵WQ, WK, 和WV都是4行三列的，从而得到的三个矩阵，Q, K, V都是2行3列的。Q的第一行是q1，第二行是q2。

最后，由于我们要处理矩阵，因此我们可以将步骤2到6压缩成一个公式，以计算自我注意层的输出。

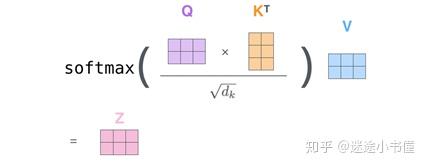


图15，矩阵形式的自注意力计算 。得到的Z矩阵是2行三列，第一行为z1，第二行为z2。

**多头(multi-head)注意力“野兽”**

论文中通过添加一种称为“多头”注意力的机制，进一步完善了自注意力层。这样可以通过两种方式提高注意力层的性能：

其一，它扩展了模型专注于不同位置的能力。是的，在上面的示例中，z1包含所有其他单词的编码向量的一点点，但是它可能由实际单词本身决定。如果我们要翻译这样的句子（例如“The animal didn’t cross the street because it was too tired/动物没过马路，因为它太累了。”），这很有用，我们想知道“它”指的是哪个单词。

其二，它为注意力层提供了多个“表示子空间”。正如我们接下来将要看到的，在多头注意力下，我们不仅拥有一个查询，而且具有多组查询/键/值权重矩阵（Transformer使用八个关注头，因此每个编码器/解码器最终得到八组） 。这些集合的每一个都是随机初始化的。然后，在训练之后，使用每个集合将输入的嵌入（或来自较低层编码器/解码器的向量）投影到不同的表示子空间中。

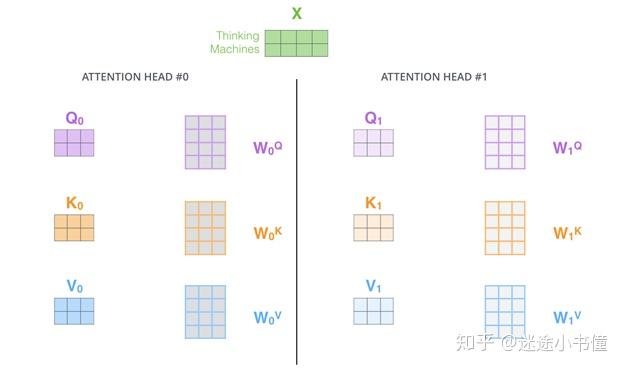


图16，双头注意力的计算示例。在多头注意力下，我们为每个头维护单独的Q / K / V权重矩阵，从而得到不同的Q / K / V矩阵。如前所述，我们将X乘以WQ / WK / WV矩阵以生成Q / K / V矩阵。此图展示了两个头的时候的例子，#0和#1，分别使用不同的WQ, WK,和WV矩阵。

如果我们执行上面概述的相同的自注意力计算，则在八个不同的次数上使用不同的权重矩阵，最终将得到八个不同的Z矩阵：

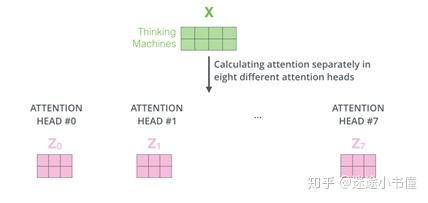


图17，八头(eight-heads)注意力的计算示例。

这给我们带来了一些挑战。前馈层(feed forward sublayer)不期望有八个矩阵-它期望一个矩阵（每个单词一个向量）。因此，我们需要一种将这八个矩阵合并为单个矩阵的方法。

我们该怎么做？我们合并矩阵，然后将它们乘以一个权重矩阵WO。

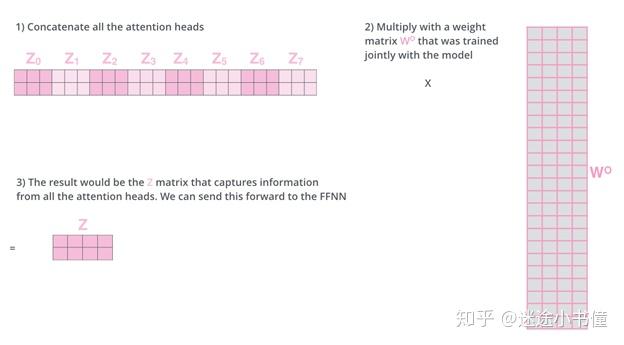


图18，自注意力层(self-attention sublayer)和前向反馈层(feed forward sublayer)的衔接计算。本图分三步，其一串联起来八个矩阵，本来是八个2\*3的矩阵，现在得到的是一个2\*24的矩阵。其二是让得到的2\*24矩阵乘以一个权重矩阵WO(24行4列)，得到Z，是2行4列的一个最终矩阵，类似的，Z矩阵的第一行是对应Thinking的向量z1；第二行是对应Machines的向量z2。

这就是多头自我关注的全部内容。我知道这里出现了很多矩阵。让我尝试将它们全部放在一个图片中，以便我们可以在一处查看它们：

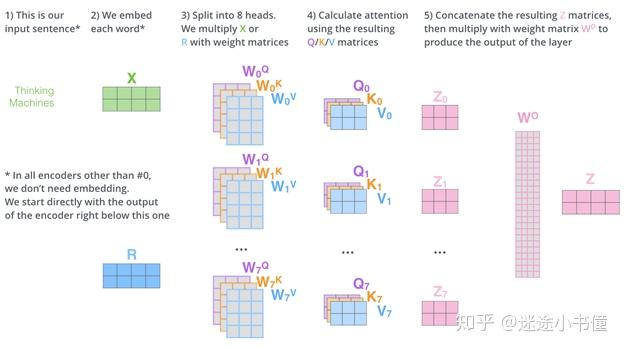


图19，综合示例一个encoder层，包括最初的输入句子，词嵌入矩阵，八头自注意力计算，前向反馈网络矩阵，以及最后的encoder单层输出。

**小练习(new)**，让我们数一下这里的可训练参数的个数：

1. 词嵌入矩阵，假设只有两个词，每个词是一个4维向量，则为2\*4=8；

2. WQ0, WK0, WV0到WQ7, WK7, WV7，一共是24个矩阵，每个矩阵是4\*3的；

3. WO是一个矩阵，是24\*4的；

所以，该图中（在不考虑其他层，例如残差层等的前提下）的参数为：8+24\*4\*3+24\*4=392。

既然我们已经涉及到注意头，那么让我们回顾一下示例，看看在示例句中对单词“ it”进行编码时，不同的注意头所关注的位置：

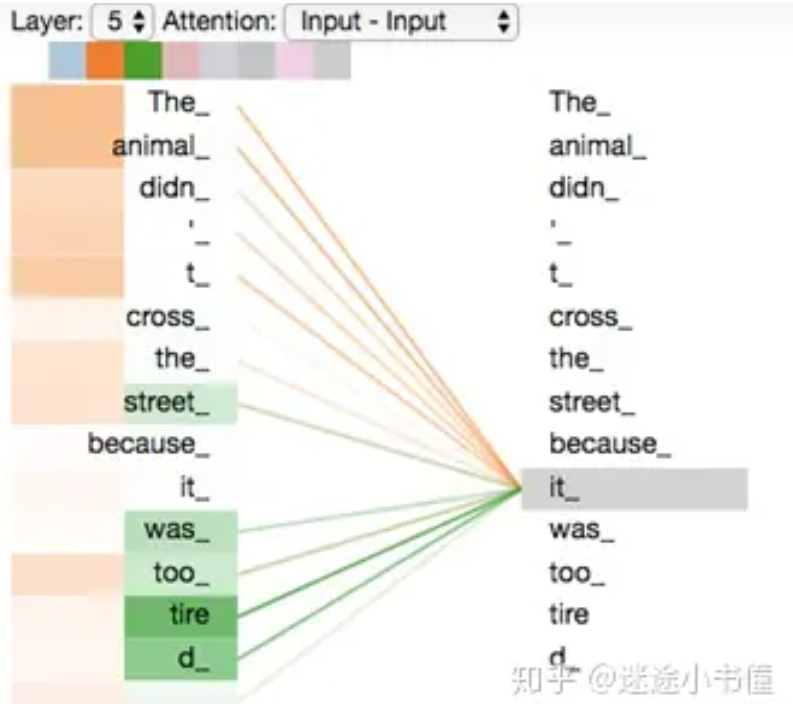


图20，以双头为例，谁和it相关？当我们对“ it”一词进行编码时，一个注意力头(attention head)集中在“动物”上（左边列的土黄色），而另一个attention head则集中在“累了”上（右边列的绿色）。从而it的向量表示，融合了“动物”和“累”这两个词的信息。 这里展示的仍然是Encoder的最高一层ENCODER #5的输出的示例。

但是，如果将所有注意力头attention-heads添加到图片中，则可能很难直观地解释：

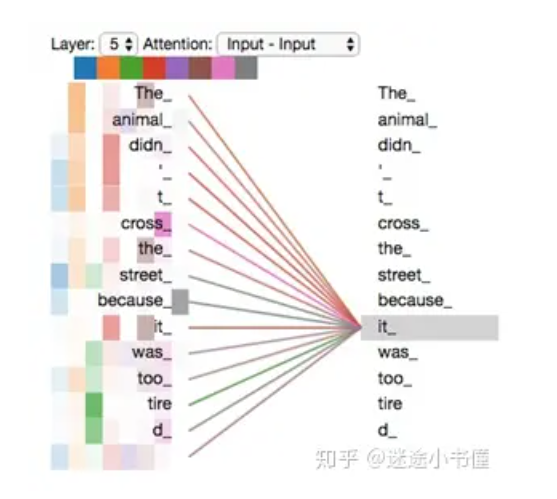


图21, 八头注意力机制下的it都和哪些词相关的示例。

当然，我们其实可以看一下这个句子的依存关系树(new)（[http://nlp.stanford.edu:8080/corenlp/process](https://link.zhihu.com/?target=http%3A//nlp.stanford.edu%3A8080/corenlp/process) ）：

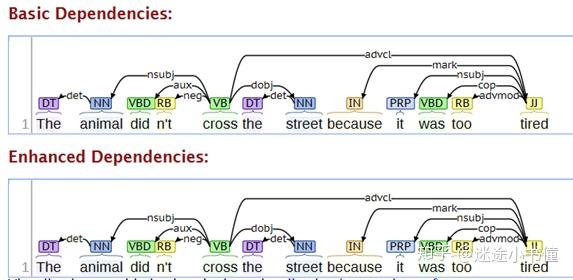


图22，英语例句的依存分析树。通过多头注意力机制，我们其实在潜在语义层面上，把it相关的所有单词，通过不同的依存路径（指向it的路径，父路径，以及更高层级的路径），表现在了其最终的向量表示上了。

**使用位置编码表示序列的顺序**

到目前为止，我们描述的模型中缺少的一件事是，一种解决输入序列中单词顺序的方法。

为了解决这个问题，Transformer将位置编码向量添加到每个输入的词嵌入中。这些向量遵循模型学习的特定模式，这有助于确定每个单词的位置或序列中不同单词之间的距离。直觉是，将这些值添加到嵌入中后，一旦将它们投影到Q / K / V向量中以及在点积注意期间，就可以在嵌入向量之间提供有意义的距离。当然，使用相对位置编码，可以在后续的论文Self-Attention with Relative Position Representations([https://arxiv.org/abs/1803.02155](https://link.zhihu.com/?target=https%3A//arxiv.org/abs/1803.02155) )和Transformer-XL([https://arxiv.org/abs/1901.02860](https://link.zhihu.com/?target=https%3A//arxiv.org/abs/1901.02860) )中找到细节。

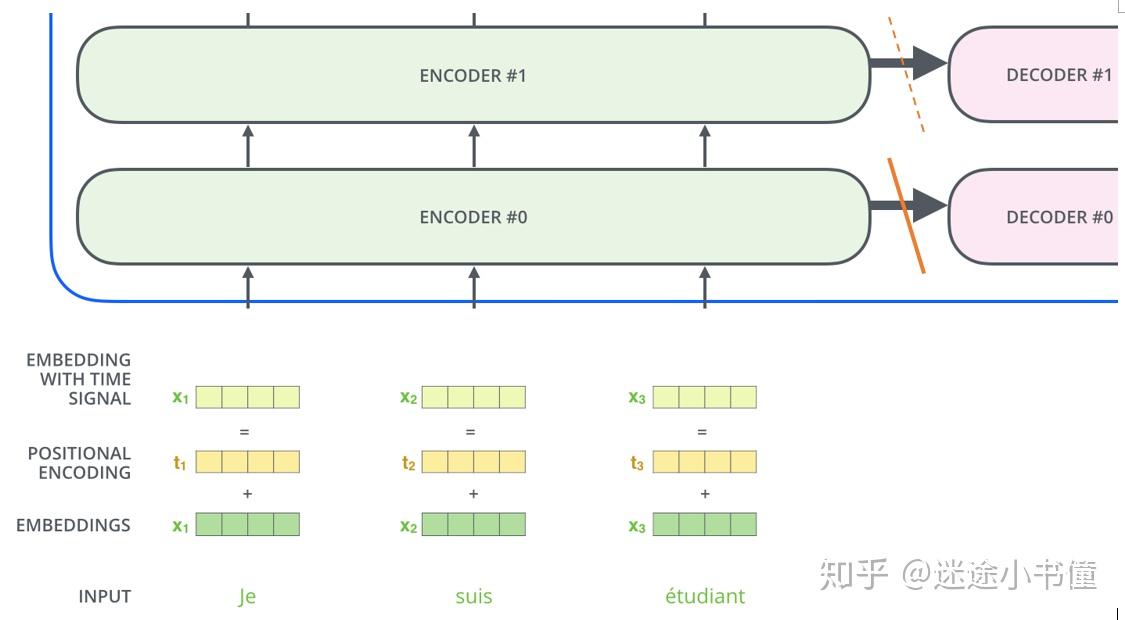


图23，引入位置编码。为了使模型具有单词的顺序感，我们添加了位置编码向量，并且，位置编码向量遵循特定的模式。需要注意的一点是，ENCODER #0 并没有箭头直接指向DECODER #0。ENCODER #1也并没有箭头直接指向DECODER #1。只有最高一层ENCODER #5会把source sentence打包编码好的结果矩阵，统一扔给DECODER的各层。(error-fix)

如果我们假设词嵌入的维数为4，则实际的位置编码应如下所示：

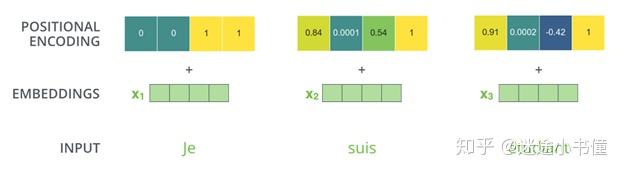


图24，（和论文不一致的, error-fix）位置编码的示例。

玩具toy词嵌入大小为4的位置编码的真实示例

注意：如果按照论文中的公式，这几个值其实是有问题的，实际的计算方法为：

PE(pos, 2i) = sin(pos/power(10000, 2i/d\_model)) ---- （1）

PE(pos, 2i+1) = cos(pos/power(10000, 2i/d\_model)) ---- (2)

这里，d\_model=4；pos为位置，从0开始；i是编码向量的维度下标。

对于第0个位置Je，其位置编码向量的四个值为：

pos=0:

i=0, 则表示i=2\*j, j=0, 因为i是偶数，要代入公式(1)，得到sin(0/power(10000, 0/4))=0;

i=1, 则表示i=2\*j+1，2\*j=0，因为i是奇数，要代入公式(2)，得到cos(0/power(10000, 0/4))=1;

i=2, 则表示i=2\*j, j=1，代入公式(1)，得到sin(0/power(10000, 2/4))=0;

i=3, cos(0/power(10000, 2/4))=1;

所以Je的位置编码向量为（0, 1, 0, 1）。

pos=1:

i=0=2\*j, j=0, sin(1/power(10000, 0/4))=sin(1)=0.8415;

i=1=2\*j+1, j=0, cos(1/power(10000, 0/4))=cos(1)=0.5403;

i=2=2\*j, j=1, sin(1/power(10000, 2/4))=sin(1/100)=0.01;

i=3=2\*j+1, j=1, cos(1/power(10000, 2/4))=cos(1/100)=1.0;

所以suis的位置编码为 (0.8415, 0.5403, 0.01, 1.0)。

pos=2:

i=0, sin(2/power(10000, 0/4)) = sin(2)=0.9093;

i=1, cos(2/power(10000, 0/4)) = cos(2) = -0.4161;

i=2, sin(2/power(10000, 2/4)) = sin(2/100) = 0.02;

i=3, cos(2/power(10000, 2/4)) = cos(2/100) = 1.0;

从而etudiant的位置编码为(0.9093, -0.4161, 0.02, 1.0)。

注意的一点是，位置编码没有梯度，不参与训练。（事先按照公式，定死的）

**这种模式是什么样的？**

在下图中，每行对应一个向量的位置编码。因此，第一行将是我们要添加到输入序列中第一个单词的嵌入的向量。每行包含512个值-每个值都在1到-1之间。我们已经对它们进行了颜色编码，因此图案可见。

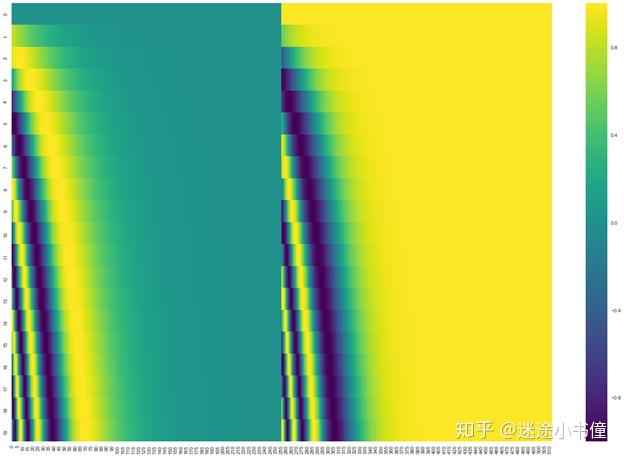
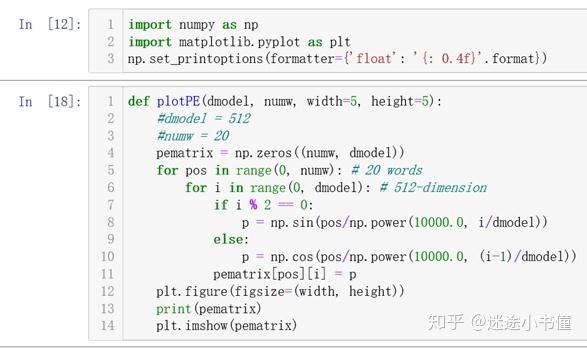


图25，（没有经过我自己检验的）嵌入大小为512（列）的20个单词（行）的位置编码的真实示例。您会看到它看起来像是在中心处分成两半。这是因为左半部分的值是由一个函数（使用正弦）生成的，而右半部分的值是由另一个函数（使用余弦）生成的。然后将它们串联起来以形成每个位置编码向量。

论文（第3.5节）中介绍了位置编码的公式。您可以在中看到用于生成位置编码的代码[get\_timing\_signal\_1d(https://github.com/tensorflow/tensor2tensor/blob/23bd23b9830059fbc349381b70d9429b5c40a139/tensor2tensor/layers/common\_attention.py )](https://zhuanlan.zhihu.com/p/105493618/get_timing_signal_1d(https:/github.com/tensorflow/tensor2tensor/blob/23bd23b9830059fbc349381b70d9429b5c40a139/tensor2tensor/layers/common_attention.py%20))。这不是唯一的位置编码方法。但是，它具有能够缩放到（训练阶段）看不见的序列长度的优点（例如，如果我们训练好的模型要求翻译的句子比训练集中的任何句子更长的长度）。

**关于论文中的“位置编码”的补充检验（New）**

验证的代码如下：



**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

np**.**set\_printoptions(formatter**=**{'float': '{: 0.4f}'**.**format})

**def** **plotPE**(dmodel, numw, width**=**5, height**=**5):

*#dmodel = 512*

*#numw = 20*

pematrix **=** np**.**zeros((numw, dmodel))

**for** pos **in** range(0, numw): *# 20 words*

**for** i **in** range(0, dmodel): *# 512-dimension*

**if** i **%** 2 **==** 0:

p **=** np**.**sin(pos**/**np**.**power(10000.0, i**/**dmodel))

**else**:

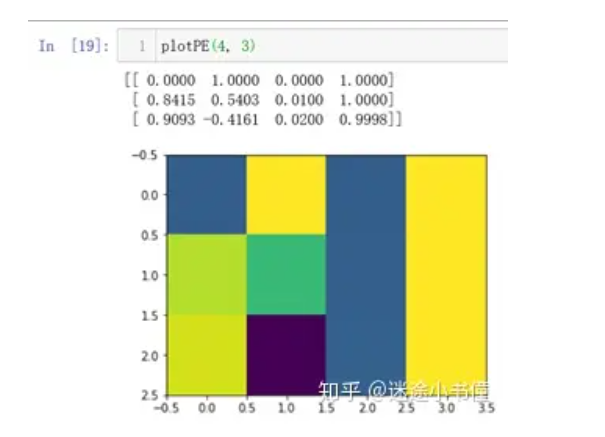
p **=** np**.**cos(pos**/**np**.**power(10000.0, (i**-**1)**/**dmodel))

pematrix[pos][i] **=** p

plt**.**figure(figsize**=**(width, height))

print(pematrix)

plt**.**imshow(pematrix)



三个词，d\_model=4的时候的表示，如上图所示。

plotPE(4, 3)

plotPE(512, 20, width=30, height=50) # almost no change after 200-dimension

plotPE(512, 200, width=30, height=30) # 200 words



上面展示了20个词，以及200个词的时候的position encoding矩阵的示例。可以看到20个词的时候，基本上200维以后各个词的位置编码保持一致了。200个词的时候，350维之后的各个词的编码也基本保持一致了。关键的区分，在前半段。

**残差**

在继续进行示例之前，我们需要提到的编码器架构中的一个细节是，每个/层编码器中的每个子层（self-attention sublayer，feed forward neural network sublayer）在其周围都有残差​​连接，然后进行[层归一化](https://link.zhihu.com/?target=https%3A//arxiv.org/abs/1607.06450)(layer normalization, [https://arxiv.org/abs/1607.06450](https://link.zhihu.com/?target=https%3A//arxiv.org/abs/1607.06450) )步骤。

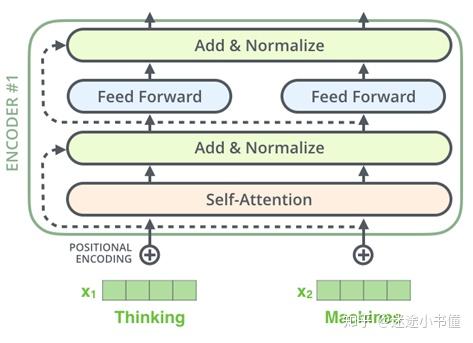


图26，残差计算的引入。

如果我们要可视化向量和与自注意力相关的layer-norm操作，它将看起来像这样：

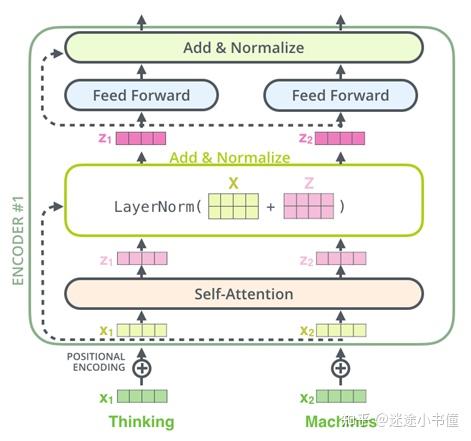


图27，残差的具体计算时机的示例。

残差计算具体体现在上图的，LayerNorm(X+Z)的X+Z的部分，其中X代表的是叠加了位置编码的浅绿色矩阵，Z来自自注意力子层的输出，X和Z都是2行4列，所以可以进行element-wise相加，得到一个新的2行4列的矩阵，然后再进行LayerNorm操作。

执行完Add&Normalize操作后，得到深红色的矩阵Z，其两个行向量z1和z2会分别发送给feed forward层，执行进一步的操作计算。

**假设，我们以位置编码向量为例，解释一下LayerNorm具体的操作(New)：**

如前所述，Je的位置编码向量为（0, 1, 0, 1），suis的位置编码为 (0.8415, 0.5403, 0.01, 1.0)，etudiant的位置编码为(0.9093, -0.4161, 0.02, 1.0)。

那么，首先计算每个向量的mean和standard derivation，即均值和标准方差：

x=(0, 1, 0, 1) ->

numpy.mean(x)=0.5=(0+1+0+1)/4,

numpy.std=0.5=sqrt( (0-0.5)\*(0-0.5) + (1-0.5)\*(1-0.5) + (0-0.5)\*(0-0.5) + (1-0.5)\*(1-0.5)) / 4)=sqrt(0.25);

所以(x-mean)/std = (-0.5, 0.5, -0.5, 0.5)/0.5 = (-1, 1, -1, 1)。

同理，suis的位置编码为 (0.8415, 0.5403, 0.01, 1.0)对应的是[ 0.64519696 -0.15272266 -1.55755923 1.06508494]，etudiant的位置编码为(0.9093, -0.4161, 0.02, 1.0)对应的是[ 0.88873494 -1.32958763 -0.59968687 1.04053957]

这（位置编码）也适用于解码器的子层。如果我们想到一个由2个堆叠式编码器和解码器组成的Transformer，它看起来像这样：

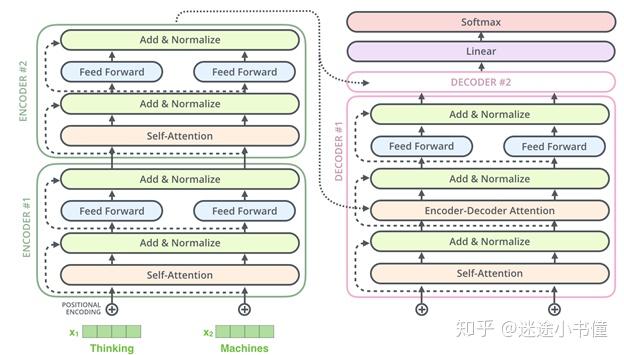


图28，开始引入decoder解码器部分。该图绘制出来了decoder #1的细节，decoder #2则高度抽象为一个box了。注意，编号应该从#0开始。

该图的细节，encoder的部分，已经比较清晰了。剩下的是decoder的部分，以及右上角的Linear和Softmax的部分。

**解码器端**

现在我们已经涵盖了编码器方面的大多数概念，我们基本上也知道了解码器的组件如何工作。但是，让我们看一下它们如何协同工作。

编码器首先处理输入序列。然后，顶部/顶层编码器的输出转换为注意力向量K和V的集合。每个解码器decoder将在其“编码器-解码器注意力层”（Encoder-Decoder Attention layer）中使用它们，这有助于解码器将注意力集中在输入序列中的适当位置：

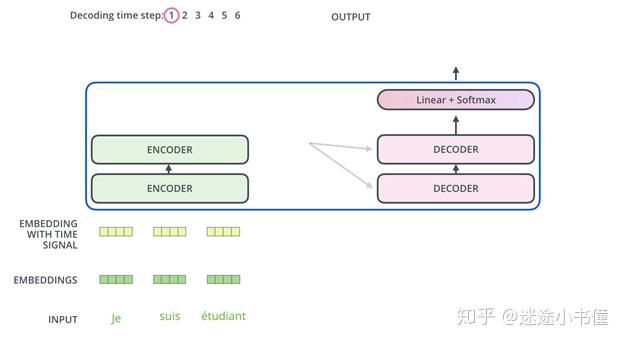


图29.1 开始综合考虑encoder和decoder。这里的图中，假设ENCODER一共两层，DECODER也是一共两层。注意，target序列需要一个起始符号，例如<s>，来表示一个序列的开始！参考图30.1 ~ 30.5.

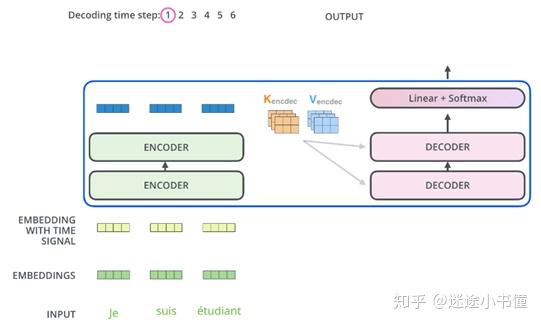


图29.2 准备好encoder的K和V矩阵，扔给decoder。这里的图中，假设ENCODER一共两层，DECODER也是一共两层。

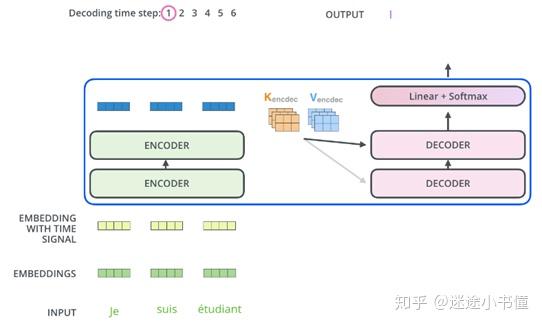


图29.3，接受来自encoder的K和V之后，decoder翻译产出了第一个词，I。完成编码阶段后，我们开始解码阶段。解码阶段的每个步骤都从输出序列中输出一个单词（此例子中，为被翻译出来的英语语句）。

以下步骤重复此过程，直到到达出现特殊符号<eos> (end-of-sentence)情况，表示Transformer解码器已完成其输出。每个步骤的输出在下一个时间步骤中被馈送到底部解码器，并且解码器像编码器一样使其解码结果向上冒泡给更高一层。就像我们对编码器输入所做的一样，我们在这些解码器输入中嵌入并添加位置编码以指示每个单词的位置（例如，当I被翻译出来之后，其位置编码，以及词嵌入，都会被参与到decoder部分的查询向量的构造上）。

**细化的训练数据的组织形式(new)：**

（本示例参考：The Annotated Transformer: [https://nlp.seas.harvard.edu/2018/04/03/attention.html](https://link.zhihu.com/?target=https%3A//nlp.seas.harvard.edu/2018/04/03/attention.html) ）

假设我们要训练一个copy transformer，其功能就是把输入原封不动地输出，例如假设输入是一个序列”1 2 3 4”，则我们期望输出也是”1 2 3 4”。

现在进入训练过程，假设我们有个batch（每个序列的第一个词”1”可以看成<s>即start-of-sentence填充符号），为：

Src Trg

1,4,2,1 1,4,2

1,4,4,4 1,4,4

这里需要注意一下我们的输入training batch的样子，src部分是2行4列的一个矩阵，而trg部分是2行3列的一个矩阵！

我们的整体目标是根据：src和trg[:, :-1]

Src **Trg[:, :-1]**

1,4,2,1 1,4,2

1,4,4,4 1,4,4

来预测 trg[:, 1:]

Src **Trg[:, 1:]**

1,4,2,1 4,2,1

1,4,4,4 4,4,4

实际的，我们的训练过程其实是类似于：

第一步：给定src = [[1,4,2,1], [1,4,4,4]] 和 trg = [[1], [1]] 通过神经网络transformer得到：

trg=[[1, 4?], [1, 4?]]，这里给4打问号，是因为实际预测到的不一定是4，如果不是4，产生loss，计算loss，并反向传播。

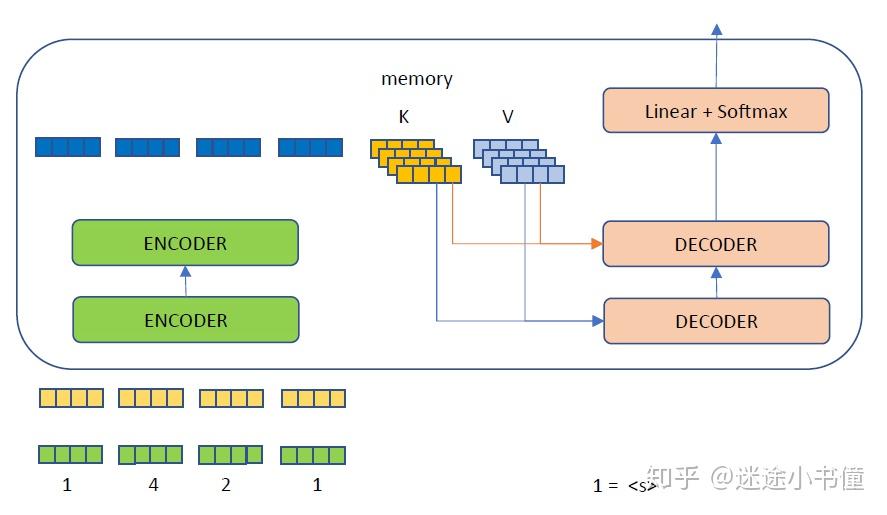


图30.1 开始解码的开始状态（假设现在是基于训练好的模型的“测试阶段”），编码器ENCODER端已经运行完毕，得到了source memory。**Target序列的第一个单词，<s>是事先给定的。**

解码器这边，最初只有一个<s>，作为句子的开始，对该词进行encode，然后参考来自ENCODER部分的memory, K, V来分别计算multi-head attention + feed forward，最后得到一个向量，对该向量进行linear + softmax,就是下一个词，例如4?。这里对4?打个问号是因为模型当前不一定能成功预测出来4.

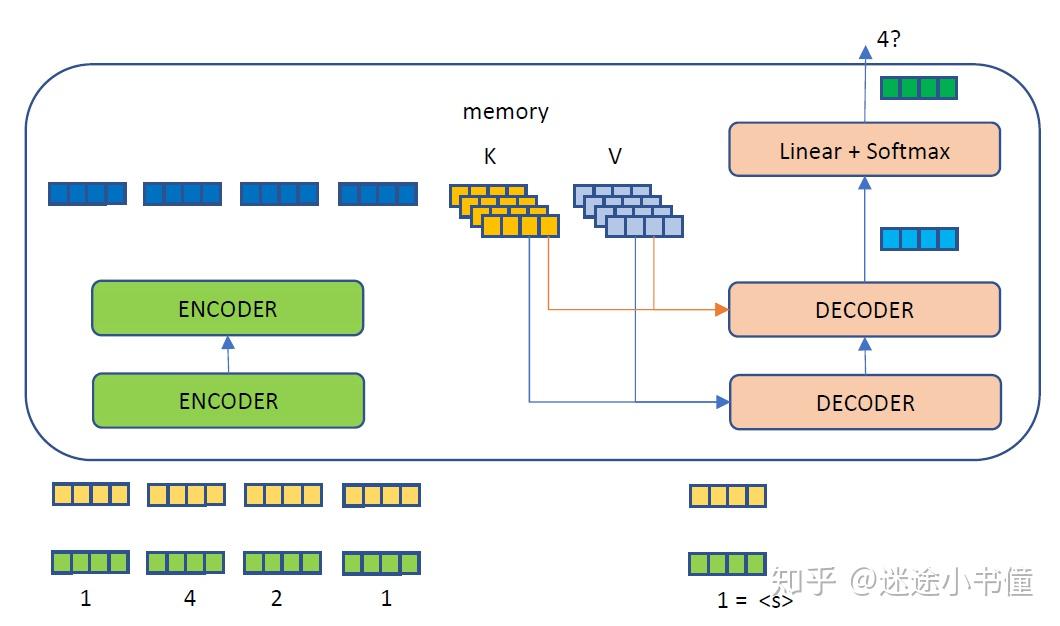


图30.2 解码器工作第一步，走一遍DECODERs和Linear+softmax生成器，然后得到一个预测出来的词4?。

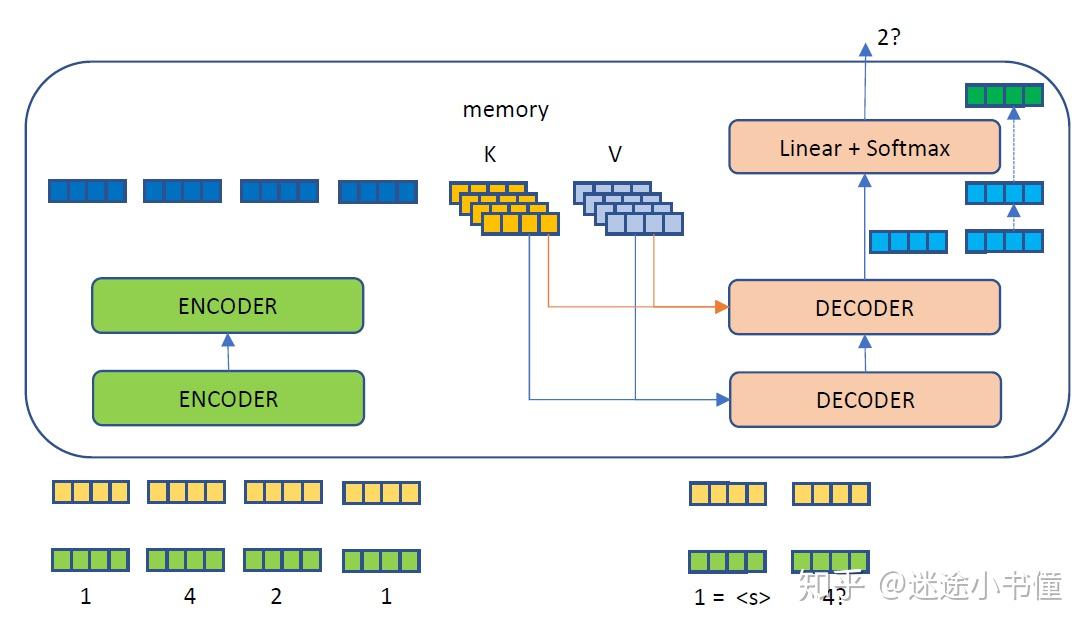


图30.3 解码器工作的第二步，根据已有的计算好的source memory和当前可用的target sequence来继续对下一个词预测。第二步，我们把得到的1，4?作为target sequence，扔给DECODER，其内部也同样参考已有的ENCODER的memory, K and K，来得到一个新的矩阵，我们取该矩阵的最后一行，扔给linear+softmax就得到下一个词，假设为2?。

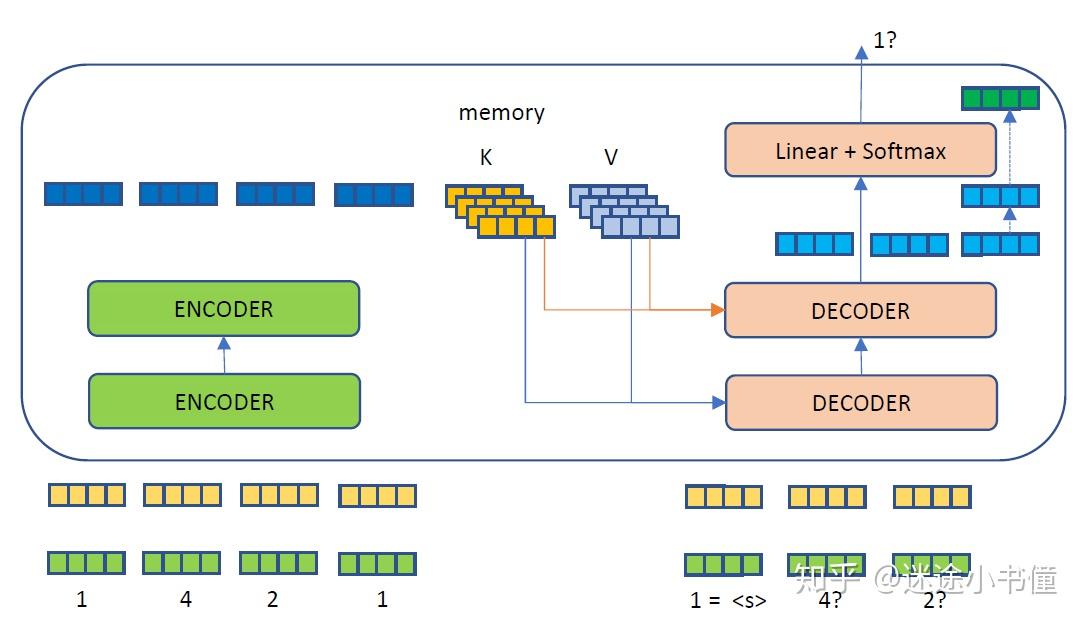


图30.4. 第三步，我们把得到的”1，4?, 2?”作为target sequence，扔给DECODER，其内部也同样参考已有的ENCODER的memory, K and K，来得到一个新的矩阵，我们取该矩阵的最后一行，扔给linear+softmax就得到下一个词，假设为1?。这样的话，我们其实完成了“训练阶段”。

当然，训练阶段，不是分成三步，而是基于mask，一步搞定的，示例为：

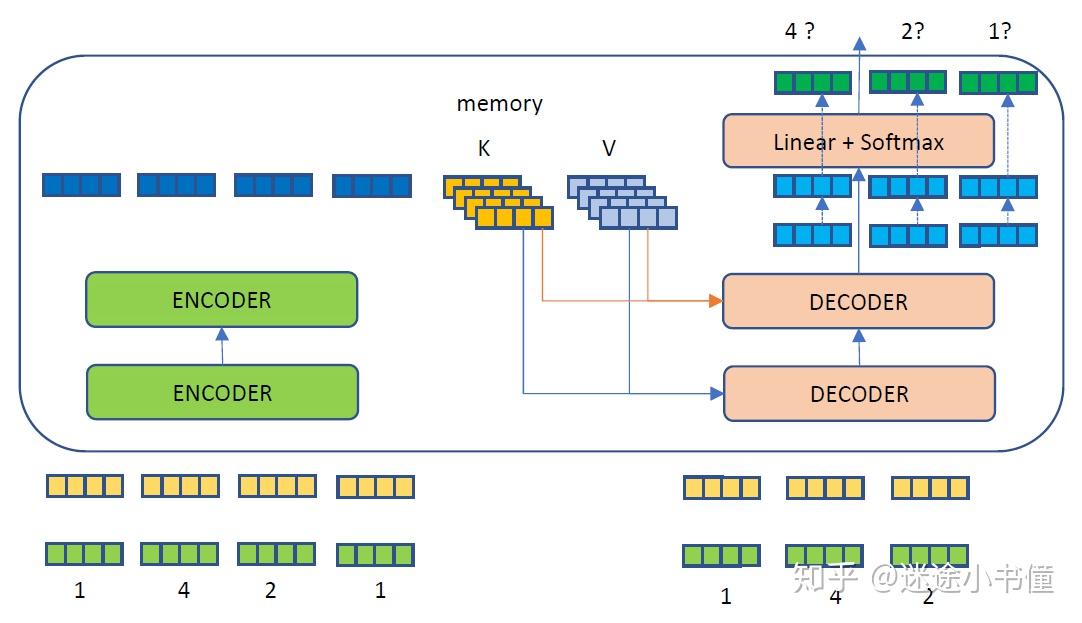


图30.5 训练阶段的“一步到位”（因为有target mask控制）。

训练阶段的一步到位，source序列为1,4,2,1，target序列为1,4,2带mask。目标是根据transformer，得到4?,2?,1?这个target序列。简化期间，类似于：

Input = source端的”s1, s4, s2, s1”和target端的”t1”从而有Output = t4?.

Input = source端的”s1, s4, s2, s1”和target端的”t1, t4”从而有Output = t4?, t2?.

Input = source端的”s1, s4, s2, s1”和target端的”t1, t4, t1”从而有 Output = t4?, t2?, t1?.

Mask相当于一个矩阵：

[[1, 0, 0], [1, 1, 0], [1, 1, 1]]。也就是说：

当预测t4（target序列的第二个词，第一个词是<s>）的时候，只有t1可见；

当预测t2(target序列的第三个词)的时候，只有t1和t4可见（这里的t4，在模型训练阶段是batch给好的，知道是4；在实际模型测试阶段，是上一步预测出来的t4?（实际可能是4或者其他词））；

当预测t1（target序列的第四个词）的时候，只有t1, t4, 和t2可见。

这样的话，预测出来的序列，t4?, t2?, t1?就会进一步和事先知道的target序列的reference, t4, t2, t1进行对比，计算loss，并使用SGD/Adam等方法进行反向传播来更新transformer中的参数。

如下是原文中的示例，这里简单copy过来，不再详细说。细化的解码器和解码算法方面的示例：

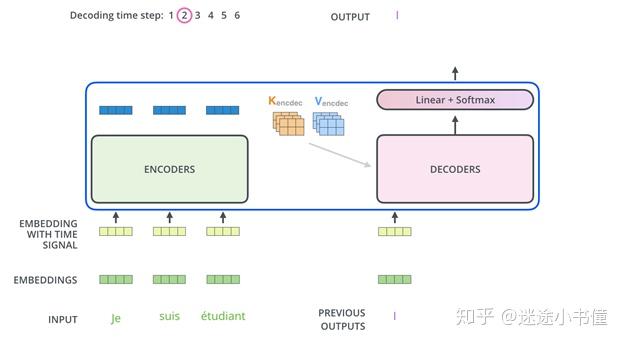


图31. 解码的例子。

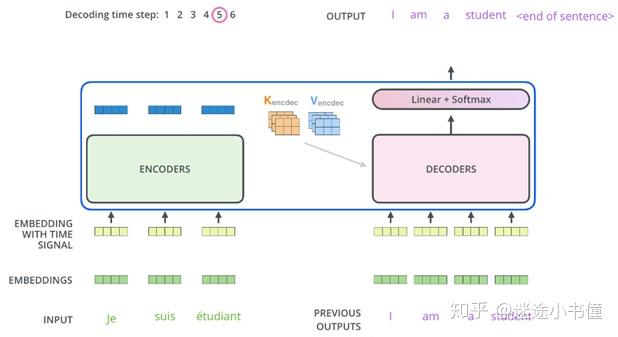


图32. 解码的例子。这里的<eos>，或者</s>是预测出来的。解码阶段的停止条件，可以是当，当前预测出来的字符是</s>的时候，算法停止。这个图有问题，因为最初必须有个<s>，否则target sequence为空的前提下，无法触发DECODER工作。”I”显然不能作为default输入。

**解码器中的自注意力层与编码器中的自注意力层略有不同：**

在解码器中，仅允许自注意力层参与输出序列中的当前位置和其前面（左边）的位置。这是通过自注意力计算的时候，在softmax计算之前，根据mask 矩阵，把未来的（当前词右边的）向量设置为-inf来完成的。

“ Encoder-Decoder Attention”层的工作方式与多头自注意力类似，不同之处在于它从其decoder下一层创建其Queries矩阵，并从编码器encoder堆栈的输出中获取Keys和Values矩阵。

**最终的线性层和Softmax层（生成器）**

解码器堆栈（即，包括6层decoder layers的一个module）输出浮点数向量。我们如何把它变成一个词？这就是最后的线性层和其后的Softmax层的工作。

线性层是一个简单的全连接的神经网络，它将解码器堆栈产生的向量投影到一个更大的向量中，**称为logits向量。（类似于one-hot-alike表示，只不过one-hot的是一个位置为1，其他位置都为0；而logits是一个位置的概率得分最大，其他的位置的值可以非0）**

假设我们的模型从其学习的训练数据集中知道有10,000个去冗余后的英语单词（我们模型的“目标语言的词汇表”）。这将使logits向量的宽度变为10,000个单元-每个单元对应一个单词的分数。这就是我们对线性层模型输出的解释。

然后，softmax层将这些分数转换为概率（全部为正，全部相加为1.0）。选择具有最高概率的单元，并且与此单元对应的单词将作为该时间步的输出。

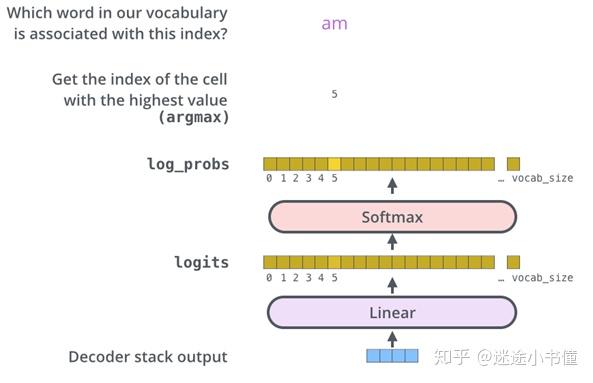


图33. 生成器，一个线性层加一个softmax层。

该图从底部开始，将产生的向量作为解码器堆栈的输出。然后将其转换为输出单词。 因为下标从0开始，所以logits和log\_probs的最右边一个词的下标应该是vocab\_size-1 (error-fix)。特别的，当目标语言的词表过大的时候，例如100K，可以考虑sampled softmax算法来加速最后一步从logits到log\_probs的计算，细节可以参考论文：[http://www.iro.umontreal.ca/~lisa/pointeurs/importance\_samplingIEEEtnn.pdf](https://link.zhihu.com/?target=http%3A//www.iro.umontreal.ca/~lisa/pointeurs/importance_samplingIEEEtnn.pdf) 和[https://arxiv.org/pdf/1602.02410v1.pdf](https://link.zhihu.com/?target=https%3A//arxiv.org/pdf/1602.02410v1.pdf) 。大部分深度学习平台上都有提供函数来直接计算sampled softmax，例如CNTK的[https://www.cntk.ai/pythondocs/CNTK\_207\_Training\_with\_Sampled\_Softmax.html；tensorflow](https://link.zhihu.com/?target=https%3A//www.cntk.ai/pythondocs/CNTK_207_Training_with_Sampled_Softmax.html%25EF%25BC%259Btensorflow)的[https://www.tensorflow.org/api\_docs/python/tf/nn/sampled\_softmax\_loss](https://link.zhihu.com/?target=https%3A//www.tensorflow.org/api_docs/python/tf/nn/sampled_softmax_loss) , pytorch的一个实现[https://github.com/leimao/Sampl](https://link.zhihu.com/?target=https%3A//github.com/leimao/Sampled_Softmax_PyTorch)

**训练回顾**

既然我们已经通过训练过的Transformer涵盖了整个前向过程(forward pass process)，那么乍一看训练模型的直觉将非常有用。

在训练过程中，未经训练的模型将经过完全相同的前向过程。但是，由于我们在标记好的（也就是给定的双语句子对集合）训练数据集上对其进行训练，因此我们可以将其当前参数值下的模型输出与实际正确的输出进行比较。从而计算损失函数，并对每个参数的梯度使用随机梯度下降或者类似的扩展方法进行进一步的更新。

为了直观地说明这一点，我们假设输出词汇表仅包含六个词（“ a”，“ am”，“ i”，“ thanks”，“ student”和“ <eos>”（“句子结尾”end-of-sentence的缩写）） 。

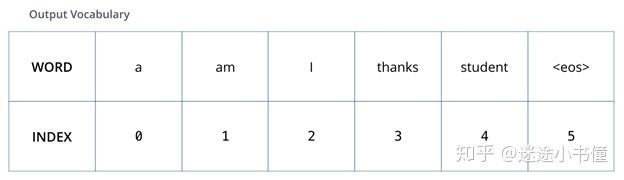


表1，我们的模型的输出词汇表是在我们开始训练之前的预处理阶段创建的。注意：这个表格不太好，最好有<s>和</s>(=<eos>)，分别作为目标句子序列的开头词和结尾词。

定义输出词汇表后，我们可以使用宽度相同的只包括0和1的向量来指示词汇表中的每个单词。这也称为one-hot编码。因此，例如，我们可以使用以下向量表示单词“ am”：

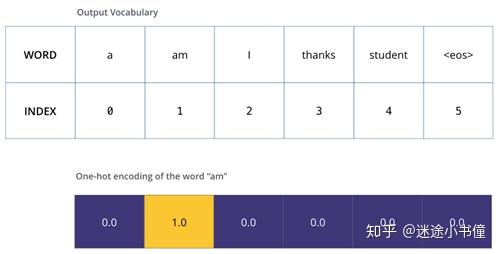


图34，示例：输出词汇表的单词”am”的one-hot编码 。

回顾之后，让我们讨论模型的损失函数-我们在训练阶段进行优化的度量标准，经过训练，我们希望可以得到惊喜的准确模型。

**损失函数**

假设我们正在训练我们的模型。假设这是我们训练阶段的第一步，我们正在以一个简单的示例对其进行训练-将“ merci”转换（翻译）为“ thank”。

这意味着我们希望，merci是输入的时候，输出的是一个表示单词“thank”的概率分布。但是，由于尚未对该模型进行训练，这不太可能发生。

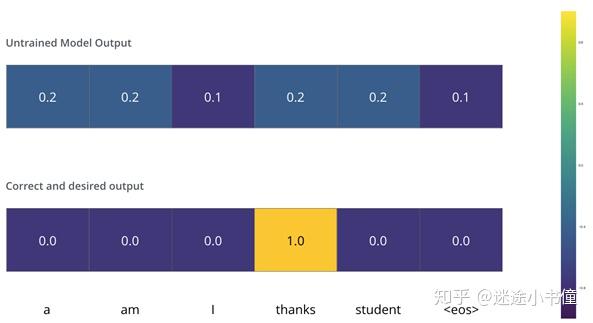


图35，由于模型的参数（权重）都是随机初始化的，因此（未经训练的）模型会针对每个单元格/单词生成具有任意值的概率分布。我们可以将其与实际输出进行比较，然后使用反向传播算法来调整所有模型的权重，以使模型输出(model’s output)更接近所需的参考输出(reference golden output)。

您如何比较两个概率分布？我们简单地从另一个中减去一个。有关更多详细信息，请参见 [交叉熵](https://link.zhihu.com/?target=https%3A//colah.github.io/posts/2015-09-Visual-Information/)([https://colah.github.io/posts/2015-09-Visual-Information/](https://link.zhihu.com/?target=https%3A//colah.github.io/posts/2015-09-Visual-Information/)) 和[Kullback-Leibler散度](https://link.zhihu.com/?target=https%3A//www.countbayesie.com/blog/2017/5/9/kullback-leibler-divergence-explained)([https://www.countbayesie.com/blog/2017/5/9/kullback-leibler-divergence-explained](https://link.zhihu.com/?target=https%3A//www.countbayesie.com/blog/2017/5/9/kullback-leibler-divergence-explained) )。

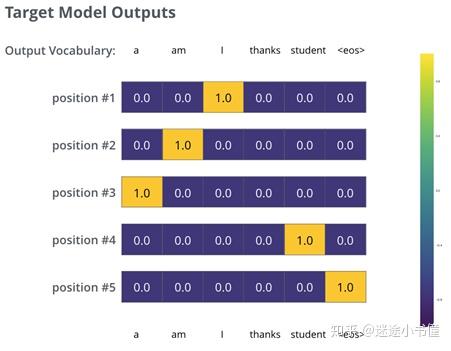
但是请注意，这是一个过于简化的示例。实际上，我们将使用一个单词多于一个单词的句子作为一个输入。例如，输入：“ je suis étudiant”，预期输出：“I am a student”。这实际上意味着我们希望我们的模型连续输出概率分布，其中：

每个概率分布都由一个宽度为vocab\_size的向量表示（在我们的玩具示例中为6，但更实际地为3,000或10,000或者更大）

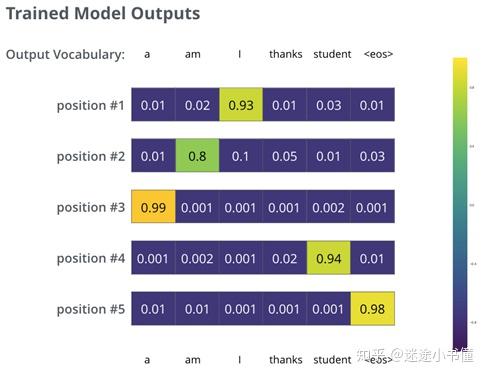
第一概率分布向量中，与单词“ i”相关联的单元格具有最高概率

第二概率分布向量中，与单词“ am”相关联的单元格处具有最高概率

依此类推，直到第五个输出分布表示'<eos>= <end of sentence>'符号，该符号也具有与10,000个元素词汇表相关的单元格。

  
图36，目标概率分布示例：我们在模型训练中，使用一个个样本句子对(bilingual sentence pairs)对该概率分布进行训练。

在足够大的数据集上训练模型足够长的时间后，我们希望产生的概率分布如下所示：

  
图37，希望经过训练，该模型将输出我们期望的正确翻译。当然，这并不是该短语是否属于训练数据集的真正迹象（请参阅：[交叉验证](https://link.zhihu.com/?target=https%3A//www.youtube.com/watch%3Fv%3DTIgfjmp-4BA) [https://www.youtube.com/watch?v=TIgfjmp-4BA](https://link.zhihu.com/?target=https%3A//www.youtube.com/watch%3Fv%3DTIgfjmp-4BA) ）。请注意，即使不太可能是该时间步长的输出，每个位置也会获得一定的概率值–这是softmax的一个非常有用的属性，可以帮助训练过程。

现在，由于该模型一次生成一个输出，因此我们可以假设该模型正在从该概率分布中选择具有最高概率的单词，然后丢弃其余单词。为做到这一点，有一种方法（称为贪婪解码）。做到这一点的另一种方法是，保留前两个单词（例如，“ I”和“ a”），然后在下一步中运行模型两次：一次假设第一个输出位置为单词“ I”，另一个假设第一个输出位置是单词“ a”，并且考虑到位置＃1和＃2都保留了较低的版本。我们在＃2和＃3等位置重复此操作。这种方法称为“beam search”，在我们的示例中，beam\_size为2（因为我们在计算位置＃1和＃2的beam之后比较了结果），和top\_beams也是两个（因为我们保留了两个词）。这两个都是您可以尝试调试的超参数(hyper parameter tuning)。

**基于KL distance的损失函数计算的简要介绍(new)：**

首先看一个batch里面只有一个句子的例子：

假设 目标P=[0.3, 0.6, 0.1]，而模型输出的是 Q=[0.4, 0.5, 0.05, 0.05]

这里P是3维的，Q是4维的，无法直接计算KLDivLoss。一种方法是pad P，例如

P=[0.3, 0.6, 0.1, 1e-9], 这样的话，就可以计算KLDivLoss了：

(P \* (P/Q).log()).sum() -> 结果为0.0924

也可以直接调用KLDivLoss方法：

torch.nn.KLDivLoss(reduction='sum')(Q.log(), P)

这里需要注意的是，KLDivLoss的第一个参数是log(model.output)，第二个参数是target reference tensor。

完整的代码如下：

python

>>> import torch

>>> from torch.autograd import Variable

>>> P = Variable(torch.FloatTensor([0.3, 0.6, 0.1, 1e-9]))

>>> Q = Variable(torch.FloatTensor([0.4, 0.5, 0.05, 0.05]))

>>> P = P.unsqueeze(0)

>>> Q = Q.unsqueeze(0)

>>> print (P, P.shape, Q, Q.shape)

tensor([[3.0000e-01, 6.0000e-01, 1.0000e-01, 1.0000e-09]]) torch.Size([1, 4]) tensor([[0.4000, 0.5000, 0.0500, 0.0500]]) torch.Size([1, 4])

>>> (P \* (P/Q).log()).sum()

tensor(0.0924)

>>> torch.nn.KLDivLoss(reduction='sum')(Q.log(), P)

tensor(0.0924)

有个潜在有意思的问题，就是在机器翻译中，基于当前训练中的模型，得到的model output的序列的长度（单词的个数）不一定和reference sequence的长度一致，这样的话，也是无法直接计算KL loss的。一个补足方法，就是把两者的长度拉到同一个数值上，例如：

1. 以reference sequence的长度为准，如果model output过长，则截取右边多出来的部分；如果过短，则补0。

2. 或者以model output的长度为准，对reference sequence进行长度的变更。

一般认为在KL loss下，第一种方案更好，因为有基于reference sequence的长度的制约。

**Transformer进阶**

我希望您已经发现这是一个有用的地方，可以开始用Transformer的主要概念打破僵局。如果您想更深入一点，建议采取以下步骤：

阅读“ [注意力就是您所需要的一切 Attention is All You Need https://arxiv.org/abs/1706.03762 ”](https://zhuanlan.zhihu.com/p/105493618/%E6%B3%A8%E6%84%8F%E5%8A%9B%E5%B0%B1%E6%98%AF%E6%82%A8%E6%89%80%E9%9C%80%E8%A6%81%E7%9A%84%E4%B8%80%E5%88%87%20Attention%20is%20All%20You%20Need%20%20https:/arxiv.org/abs/1706.03762%20)论文，Transformer博客文章（《[Transformer：一种用于语言理解的新型神经网络体系结构https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html 》](https://zhuanlan.zhihu.com/p/105493618/Transformer%EF%BC%9A%E4%B8%80%E7%A7%8D%E7%94%A8%E4%BA%8E%E8%AF%AD%E8%A8%80%E7%90%86%E8%A7%A3%E7%9A%84%E6%96%B0%E5%9E%8B%E7%A5%9E%E7%BB%8F%E7%BD%91%E7%BB%9C%E4%BD%93%E7%B3%BB%E7%BB%93%E6%9E%84https:/ai.googleblog.com/2017/08/transformer-novel-neural-network.html%20%E3%80%8B)）和[Tensor2Tensor公告https://ai.googleblog.com/2017/06/accelerating-deep-learning-research.html](https://link.zhihu.com/?target=https%3A//ai.googleblog.com/2017/06/accelerating-deep-learning-research.html) 。

观看[ŁukaszKaiser的演讲https://www.youtube.com/watch?v=rBCqOTEfxvg](https://link.zhihu.com/?target=https%3A//www.youtube.com/watch%3Fv%3DrBCqOTEfxvg) ，探讨模型及其细节

使用[Tensor2Tensor存储库中提供](https://link.zhihu.com/?target=https%3A//colab.research.google.com/github/tensorflow/tensor2tensor/blob/master/tensor2tensor/notebooks/hello_t2t.ipynb)的[Jupyter Notebook](https://link.zhihu.com/?target=https%3A//colab.research.google.com/github/tensorflow/tensor2tensor/blob/master/tensor2tensor/notebooks/hello_t2t.ipynb)探秘[https://colab.research.google.com/github/tensorflow/tensor2tensor/blob/master/tensor2tensor/notebooks/hello\_t2t.ipynb](https://link.zhihu.com/?target=https%3A//colab.research.google.com/github/tensorflow/tensor2tensor/blob/master/tensor2tensor/notebooks/hello_t2t.ipynb)

探索[Tensor2Tensor repo](https://zhuanlan.zhihu.com/p/105493618/Tensor2Tensor%20repo) [https://github.com/tensorflow/tensor2tensor](https://link.zhihu.com/?target=https%3A//github.com/tensorflow/tensor2tensor)

· 后续工作：

· [Depthwise Separable Convolutions for Neural Machine Translation](https://link.zhihu.com/?target=https%3A//arxiv.org/abs/1706.03059)

· [One Model To Learn Them All](https://link.zhihu.com/?target=https%3A//arxiv.org/abs/1706.05137)

· [Discrete Autoencoders for Sequence Models](https://link.zhihu.com/?target=https%3A//arxiv.org/abs/1801.09797)

· [Generating Wikipedia by Summarizing Long Sequences](https://link.zhihu.com/?target=https%3A//arxiv.org/abs/1801.10198)

· [Image Transformer](https://link.zhihu.com/?target=https%3A//arxiv.org/abs/1802.05751)

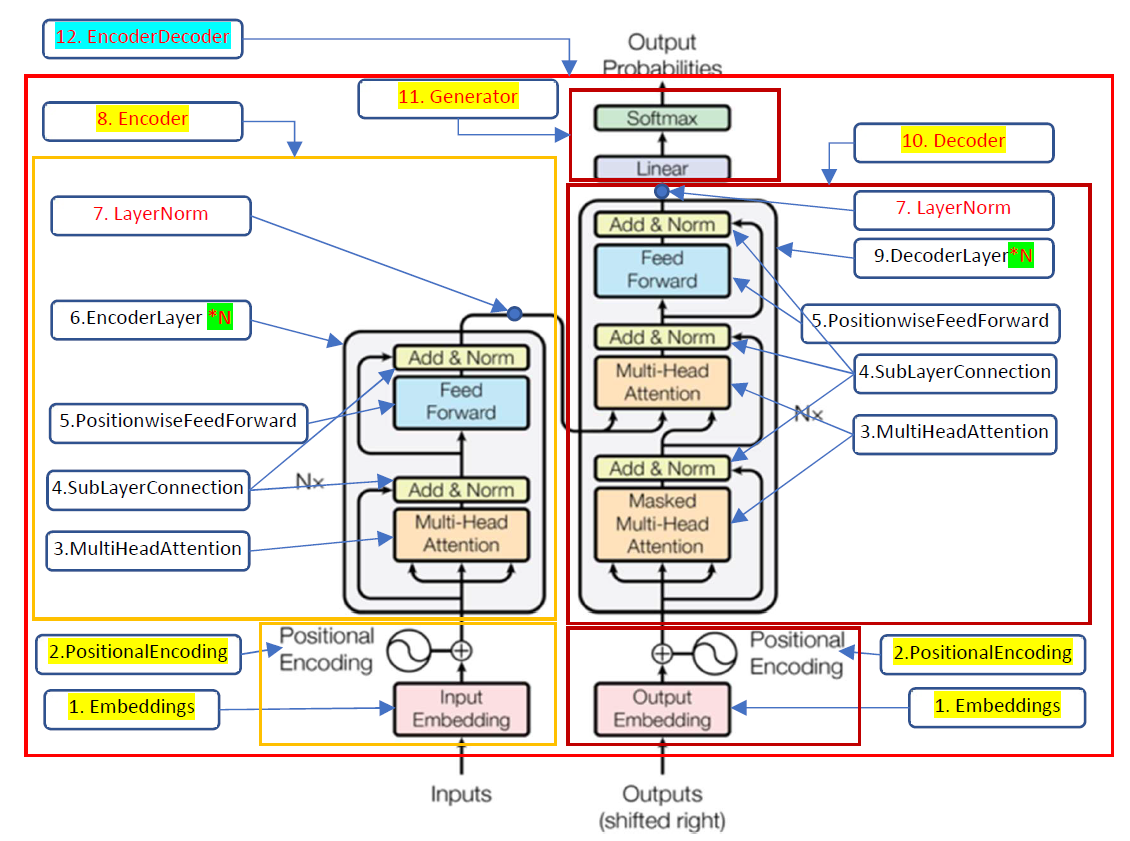
· [Training Tips for the Transformer Model](https://link.zhihu.com/?target=https%3A//arxiv.org/abs/1804.00247)

· [Self-Attention with Relative Position Representations](https://link.zhihu.com/?target=https%3A//arxiv.org/abs/1803.02155)

· [Fast Decoding in Sequence Models using Discrete Latent Variables](https://link.zhihu.com/?target=https%3A//arxiv.org/abs/1803.03382)

· [Adafactor: Adaptive Learning Rates with Sublinear Memory Cost](https://link.zhihu.com/?target=https%3A//arxiv.org/abs/1804.04235)

英文原版来自：Alammar，Jay（2018）。示例化Transformer[博客文章]。取自[https://jalammar.github.io/illustrated-transformer/。](https://link.zhihu.com/?target=https%3A//jalammar.github.io/illustrated-transformer/)



**The Annotated Transformer的中文注释版（1）**

题外话：最近微软发布了Turing-NLG, 170亿参数，史上最大的NLG模型。78个Transformer层，4256个隐藏层参数，28个注意力头，使用了1024长度的token序列，训练了30万步，batch size为512，损失函数为autoregressive generation loss。在数据集合WikiText-103, LAMBADA strict上超越了15亿参数的Open AI GPT-2和NVIDIA Megatron-LM的83亿参数的模型。又是Transformer...

进入正题，看看transformer的详细代码实现吧。这篇大作来自于[http://nlp.seas.harvard.edu/2018/04/03/attention.html](https://link.zhihu.com/?target=http%3A//nlp.seas.harvard.edu/2018/04/03/attention.html)，作者是AlexanderM. Rush；刚从哈佛助理教授转到Cornell Tech 做副教授。他们新team的主页：[http://nlp.cornell.edu/people/](https://link.zhihu.com/?target=http%3A//nlp.cornell.edu/people/)。

因为篇幅很大，我计划分3次左右阐述。

坦白讲，The Annotated Transformer我前后看了不下10遍，鉴于自己能力有限，总感觉每次理解都有所更新，这次想完整的记录一下，如何根据最拉风的论文，写出最拉风的代码，并且代码优美，回味无穷。全程使用pytorch讲解，如果对这个开源深度学习框架不太熟悉的同学，推荐：周莫烦的pytorch入门视频：[https://github.com/MorvanZhou/PyTorch-Tutorial](https://link.zhihu.com/?target=https%3A//github.com/MorvanZhou/PyTorch-Tutorial)

该课程短小精悍，大概1-2天可以精心刷完核心的代码。有意思的是，CNN做MNIST data的分类, RNN for 分类/归回, 自编码器，DQN强化学习，对抗生成网络GAN, 动态图原理，GPU训练方法，Dropout以及批标准化(batch normalization)都有所讲解。五星推荐给新人。

进入正题，先来一张大图：



图1 Transformer的总图（和代码class类名结合的）

图1是基于原始论文中的transformer的总图，为每个部分标识出了其具体的对应的class name（类名）。为了方便记忆，这里对每个类class进行了编号1到12。

先看最外边，12 classEncoderDecoder，是个总类，这个类包括五个主要部分：

1. Embeddings和PositionalEncoding（两者通过torch.nn.Sequential来串起来使用）: 左下角的，是输入序列的word embedding + positional encoding;

2. Embeddings和PositionalEncoding(两者通过torch.nn.Sequential来串起来使用): 右下角的，是输出序列（训练阶段是完整的输出序列+mask，测试阶段是到当前位置为止的输出序列+mask）

3. Encoder：左边中央位置，负责管理6层EncoderLayer，以及一个最后的LayerNorm（做带有可训练参数的，批标准化），需要注意的是，原文的LayerNorm，命名有问题，其实不是LayerNorm，因为这个类，并没有被用于接到每个EncoderLayer的后边，而是6层EncoderLayer之后，只接一个LayerNorm，所以称为EncoderNorm或者DecoderNorm（Decoder的部分也类似）更合适，简化期间，可以称为CoderNorm (Coder表示Encoder或者Decoder)。

4. Decoder: 右边中央位置，负责管理6层DecoderLayer，以及一个最后的LayerNorm（带有可训练参数的，批标准化）

5. Generator: 包括一个全连接层和一个softmax层，负责把top-N 候选输出序列找出来。

上代码吧：

*# !pip install http://download.pytorch.org/whl/cu80/torch-0.3.0.post4-cp36-cp36m-linux\_x86\_64.whl*

*# !pip install numpy matplotlib spacy torchtext seaborn*

**import** numpy **as** np

**import** torch

**import** torch.nn **as** nn

**import** torch.nn.functional **as** F

**import** math**,**copy**,**time

**from** torch.autograd **import** Variable

**import** matplotlib.pyplot **as** plt

**import** seaborn

seaborn**.**set\_context(context**=**"talk")

*# seaborn只在最后可视化self-attention的时候用到，*

*# 可以先不管或者注释掉这两行*

**%**matplotlibinline *# only for jupyter notebook*

提前需要安装上面这些包，注意torch版本0.3.0有些陈旧了，如果你的pytorch是更高的版本，则我们需要一起修改某些地方。

numpy和matplotlib没啥说的，pytorch就是numpy的一个augograd,  
tensor, GPU的高级封装。Matplotlib是画图的大神级package，这里将会使用到其绘制mask矩阵，位置编码等的示例化表示。Spacy（负责做负责管理机器翻译的一些公开数据集，例如IWSLT-国际口语翻译worksop，我2009年参加过比赛），和torchtext（tokenize ，简单理解tokenize就是类似于，把句子Jack’s mom is crying. 的标点符号和词分开：Jack ’s mom is crying .）。

为了方便循序渐进的理解，我们先从第一个基本的class开始：

顾名思义，Embeddings类主要负责对输入的source sequence和target

sequence的词嵌入表示的映射，具体为每个词从one-hot表示，映射为d\_model维度的一个向量。这样的话，如果有10个词，d\_model为512的时候，则我们得到的是一个10\*512的矩阵。每一行是512列，代表一个词的dense表示。

“In our model, we share the same weight matrix between the two embedding layers and the pre-softmax linear transformation”,  
这句话，在没有理解他们的一篇ACL的使用词根做机器翻译的论文的前提下，其实不容易理解。例如，在做欧洲语系和英语翻译的时候，很多词是共享词根的，所以，他们这个源语言和目标语言共享一个词嵌入矩阵有一定的道理。如果是中文和英文之间，则完全没有共享词嵌入矩阵的必要。

**1.Embeddings**

**class** **Embeddings**(nn**.**Module):

**def** \_\_init\_\_(self,d\_model,vocab):

*#d\_model=512, vocab=当前语言的词表大小*

super(Embeddings,self)**.**\_\_init\_\_()

self**.**lut**=**nn**.**Embedding(vocab,d\_model)

*# one-hot转词嵌入，这里有一个待训练的矩阵E，大小是vocab\*d\_model*

self**.**d\_model**=**d\_model *# 512*

**def** **forward**(self,x):

*# x ~ (batch.size, sequence.length, one-hot),*

*#one-hot大小=vocab，当前语言的词表大小*

**return** self**.**lut(x)**\***math**.**sqrt(self**.**d\_model)

*# 得到的10\*512词嵌入矩阵，主动乘以sqrt(512)=22.6，*

*#这里我做了一些对比，感觉这个乘以sqrt(512)没啥用… 求反驳。*

*#这里的输出的tensor大小类似于(batch.size, sequence.length, 512)*

具体的源语言和目标语言的共享词嵌入矩阵的方法如下，其中的weight就是vocab\*d\_model的词嵌入矩阵：

**if** False:

model**.**src\_embed[0]**.**lut**.**weight**=**model**.**tgt\_embeddings[0]**.**lut**.**weight

model**.**generator**.**lut**.**weight**=**model**.**tgt\_embed[0]**.**lut**.**weight

2.PositionalEncoding （PE）位置编码

**class** **PositionalEncoding**(nn**.**Module):

"Implement the PE function."

**def** \_\_init\_\_(self, d\_model, dropout, max\_len**=**5000):

*#d\_model=512,dropout=0.1,*

*#max\_len=5000代表事先准备好长度为5000的序列的位置编码，其实没必要，*

*#一般100或者200足够了。*

super(PositionalEncoding, self)**.**\_\_init\_\_()

self**.**dropout **=** nn**.**Dropout(p**=**dropout)

*# Compute the positional encodings once in log space.*

pe **=** torch**.**zeros(max\_len, d\_model)

*#(5000,512)矩阵，保持每个位置的位置编码，一共5000个位置，*

*#每个位置用一个512维度向量来表示其位置编码*

position **=** torch**.**arange(0, max\_len)**.**unsqueeze(1)

*# (5000) -> (5000,1)*

div\_term **=** torch**.**exp(torch**.**arange(0, d\_model, 2) **\***

**-**(math**.**log(10000.0) **/** d\_model))

*# (0,2,…, 4998)一共准备2500个值，供sin, cos调用*

pe[:, 0::2] **=** torch**.**sin(position **\*** div\_term) *# 偶数下标的位置*

pe[:, 1::2] **=** torch**.**cos(position **\*** div\_term) *# 奇数下标的位置*

pe **=** pe**.**unsqueeze(0)

*# (5000, 512) -> (1, 5000, 512) 为batch.size留出位置*

self**.**register\_buffer('pe', pe)

**def** **forward**(self, x):

x **=** x **+** Variable(self**.**pe[:, :x**.**size(1)], requires\_grad**=**False)

*# 接受1.Embeddings的词嵌入结果x，*

*#然后把自己的位置编码pe，封装成torch的Variable(不需要梯度)，加上去。*

*#例如，假设x是(30,10,512)的一个tensor，*

*#30是batch.size, 10是该batch的序列长度, 512是每个词的词嵌入向量；*

*#则该行代码的第二项是(1, min(10, 5000), 512)=(1,10,512)，*

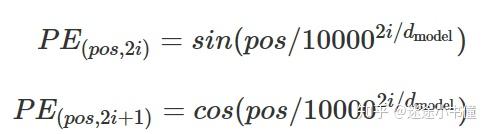
*#在具体相加的时候，会扩展(1,10,512)为(30,10,512)，*

*#保证一个batch中的30个序列，都使用（叠加）一样的位置编码。*

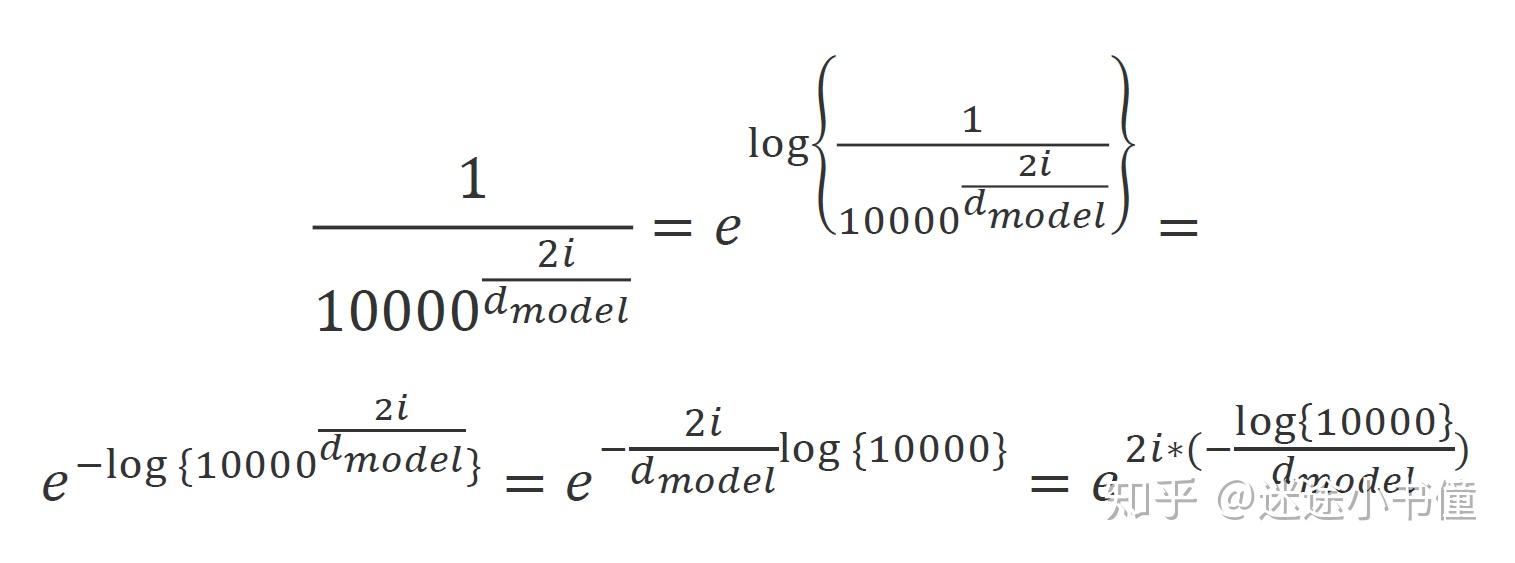
**return** self**.**dropout(x) *# 增加一次dropout操作*

*# 注意，位置编码不会更新，是写死的，所以这个class里面没有可训练的参数。*

注意，位置编码不会更新，是写死的，所以这个class里面没有可训练的参数。



为了计算这个公式，上面的代码写的比较风骚，以2i为偶数为例子：



其中2i，是步长为2的0, 2, 4, …, 4998; d\_model=512.

有了1.Embeddings和2.PositionalEncoding；在具体使用的时候，是通过torch.nn.Sequential来把他们两个串起来的：

例如：

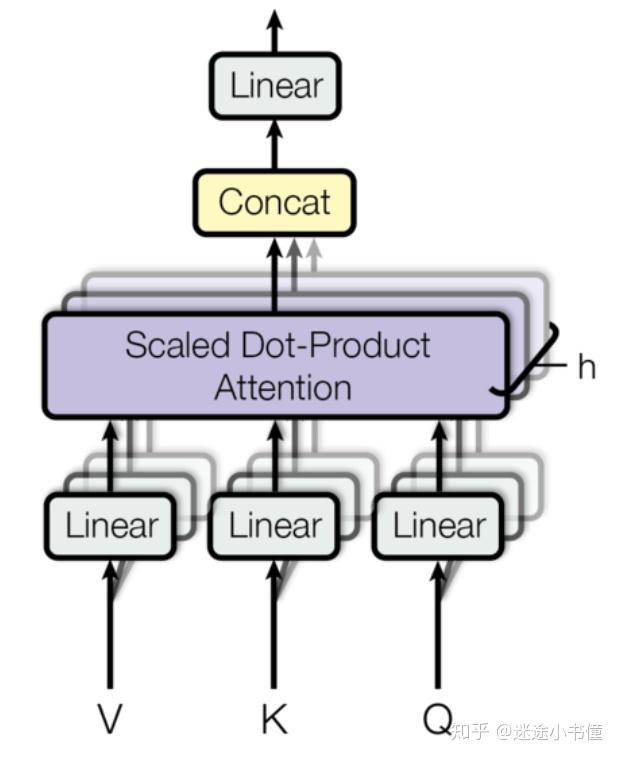
nn**.**Sequential(Embeddings(d\_model,src\_vocab),

PositionalEncoding(d\_model,dropout))

*# 例如，d\_model=512, src\_vocab=源语言的词表大小,*

*#dropout=0.1即 dropout rate*

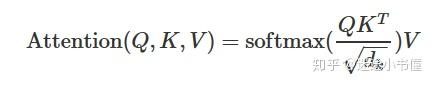
下面进入最有意思的，多头注意力模型：



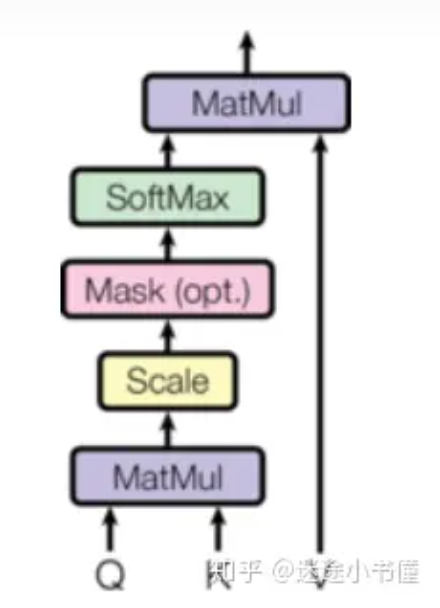
多头注意力计算图

Version:1.0 StartHTML:000000379 EndHTML:000609185 StartFragment:000608337 EndFragment:000609117 StartSelection:000608337 EndSelection:000609117 SourceURL:[https://mp.weixin.qq.com/s?\_\_biz=MzI5MjM4MTMzNg==&mid=2247483746&idx=1&sn=5909cd6ed08c740051f8cf3829ae4eb1&chksm=ec037695db74ff83c97fa84af44e408e688554ce7660acd0adbfba98c648dcea574c42acebc5&token=961096729&lang=zh\_CN](https://link.zhihu.com/?target=https%3A//mp.weixin.qq.com/s%3F__biz%3DMzI5MjM4MTMzNg%3D%3D%26mid%3D2247483746%26idx%3D1%26sn%3D5909cd6ed08c740051f8cf3829ae4eb1%26chksm%3Dec037695db74ff83c97fa84af44e408e688554ce7660acd0adbfba98c648dcea574c42acebc5%26token%3D961096729%26lang%3Dzh_CN) window.logs = { pagetime: {} }; window.logs.pagetime['html\_begin'] = (+new Date()); window.LANG= "zh\_CN"; window.isOldVideoPage = false;

V, K, Q的含义，已经在我之前的文章, 图示transformer里面详细讲了，这里不再冗述。需要注意的是，这里面有四个待训练参数集合，分别是四个Linear Networks, 三个在输入端（下方），一个在输出端，上方。h代表注意力头的个数，例如8个。  
  
先看attention函数的定义：



和该公式对应的计算图是：



self-attention的计算图

**def** **attention**(query, key, value, mask**=**None, dropout**=**None):

*# query, key, value的形状类似于(30, 8, 10, 64), (30, 8, 11, 64),*

*#(30, 8, 11, 64)，例如30是batch.size，即当前batch中有多少一个序列；*

*# 8=head.num，注意力头的个数；*

*# 10=目标序列中词的个数，64是每个词对应的向量表示；*

*# 11=源语言序列传过来的memory中，当前序列的词的个数，*

*# 64是每个词对应的向量表示。*

*# 类似于，这里假定query来自target language sequence；*

*# key和value都来自source language sequence.*

"Compute 'Scaled Dot Product Attention'"

d\_k **=** query**.**size(**-**1) *# 64=d\_k*

scores **=** torch**.**matmul(query, key**.**transpose(**-**2, **-**1)) **/**

math**.**sqrt(d\_k) *# 先是(30,8,10,64)和(30, 8, 64, 11)相乘，*

*#（注意是最后两个维度相乘）得到(30,8,10,11)，*

*#代表10个目标语言序列中每个词和11个源语言序列的分别的“亲密度”。*

*#然后除以sqrt(d\_k)=8，防止过大的亲密度。*

*#这里的scores的shape是(30, 8, 10, 11)*

**if** mask **is** **not** None:

scores **=** scores**.**masked\_fill(mask **==** 0, **-**1e9)

*#使用mask，对已经计算好的scores，按照mask矩阵，填-1e9，*

*#然后在下一步计算softmax的时候，被设置成-1e9的数对应的值~0,被忽视*

p\_attn **=** F**.**softmax(scores, dim **=** **-**1)

*#对scores的最后一个维度执行softmax，得到的还是一个tensor,*

*#(30, 8, 10, 11)*

**if** dropout **is** **not** None:

p\_attn **=** dropout(p\_attn) *#执行一次dropout*

**return** torch**.**matmul(p\_attn, value), p\_attn

*#返回的第一项，是(30,8,10, 11)乘以（最后两个维度相乘）*

*#value=(30,8,11,64)，得到的tensor是(30,8,10,64)，*

*#和query的最初的形状一样。另外，返回p\_attn，形状为(30,8,10,11).*

*#注意，这里返回p\_attn主要是用来可视化显示多头注意力机制。*

**3.MultiHeadedAttention**

理解了这个类，Transformer的精髓也就理解的差不多了，至于后面的Decoder或者Generator，如果自己的任务，不是双语翻译之类的，而是类似“预训练”，BERT，这种路线，则Decoder或者Generator可以完全简化。（只看一个Decoder+Generator足矣）

**class** **MultiHeadedAttention**(nn**.**Module):

**def** \_\_init\_\_(self, h, d\_model, dropout**=**0.1):

*# h=8, d\_model=512*

"Take in model size and number of heads."

super(MultiHeadedAttention, self)**.**\_\_init\_\_()

**assert** d\_model **%** h **==** 0 *# We assume d\_v always equals d\_k 512%8=0*

self**.**d\_k **=** d\_model **//** h *# d\_k=512//8=64*

self**.**h **=** h *#8*

self**.**linears **=** clones(nn**.**Linear(d\_model, d\_model), 4)

*#定义四个Linear networks, 每个的大小是(512, 512)的，*

*#每个Linear network里面有两类可训练参数，Weights，*

*#其大小为512\*512，以及biases，其大小为512=d\_model。*

self**.**attn **=** None

self**.**dropout **=** nn**.**Dropout(p**=**dropout)

**def** **forward**(self, query, key, value, mask**=**None):

*# 注意，输入query的形状类似于(30, 10, 512)，*

*# key.size() ~ (30, 11, 512),*

*#以及value.size() ~ (30, 11, 512)*

**if** mask **is** **not** None: *# Same mask applied to all h heads.*

mask **=** mask**.**unsqueeze(1) *# mask下回细细分解。*

nbatches **=** query**.**size(0) *#e.g., nbatches=30*

*# 1) Do all the linear projections in batch from*

*#d\_model => h x d\_k*

query, key, value **=** [l(x)**.**view(nbatches, **-**1, self**.**h, self**.**d\_k)

**.**transpose(1, 2) **for** l, x **in**

zip(self**.**linears, (query, key, value))]

*# 这里是前三个Linear Networks的具体应用，*

*#例如query=(30,10, 512) -> Linear network -> (30, 10, 512)*

*#-> view -> (30,10, 8, 64) -> transpose(1,2) -> (30, 8, 10, 64)*

*#，其他的key和value也是类似地，*

*#从(30, 11, 512) -> (30, 8, 11, 64)。*

*# 2) Apply attention on all the projected vectors in batch.*

x, self**.**attn **=** attention(query, key, value, mask**=**mask,

dropout**=**self**.**dropout)

*#调用上面定义好的attention函数，输出的x形状为(30, 8, 10, 64)；*

*#attn的形状为(30, 8, 10=target.seq.len, 11=src.seq.len)*

*# 3) "Concat" using a view and apply a final linear.*

x **=** x**.**transpose(1, 2)**.**contiguous()**.**

view(nbatches, **-**1, self**.**h **\*** self**.**d\_k)

*# x ~ (30, 8, 10, 64) -> transpose(1,2) ->*

*#(30, 10, 8, 64) -> contiguous() and view ->*

*#(30, 10, 8\*64) = (30, 10, 512)*

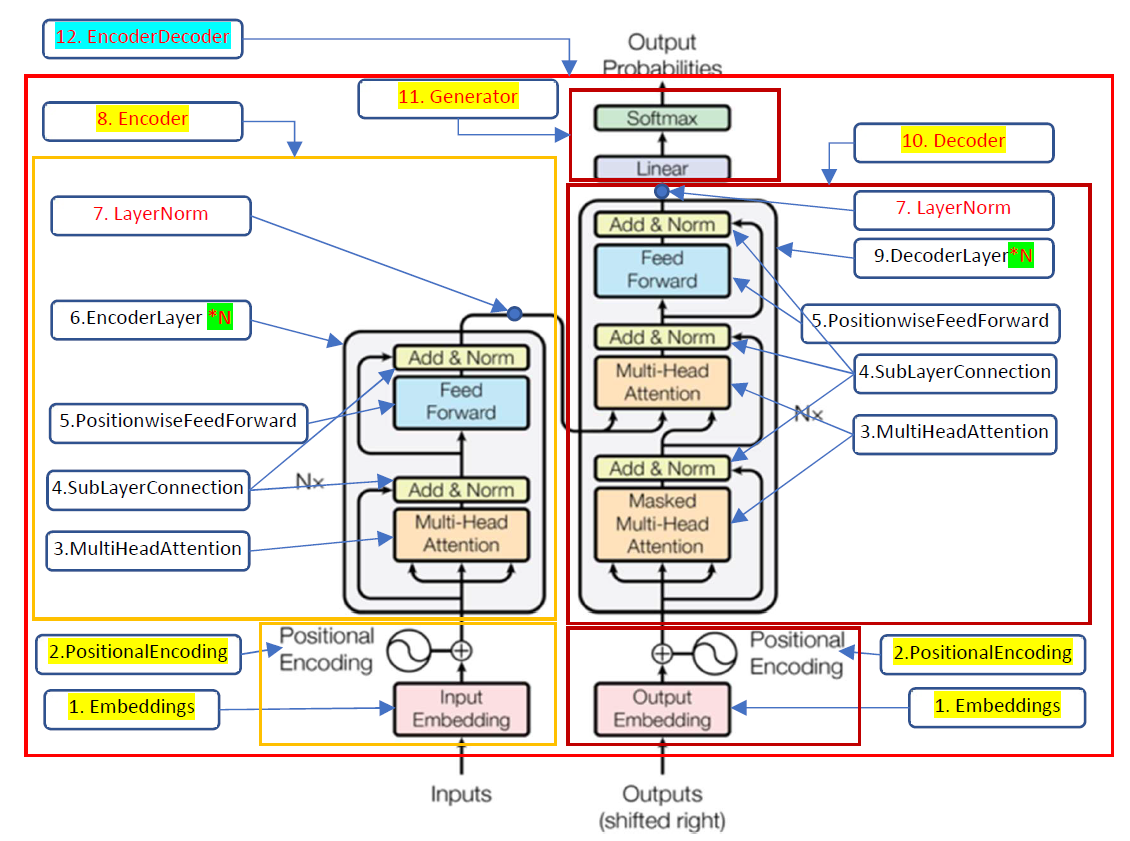
**return** self**.**linears[**-**1](x)

*#执行第四个Linear network，把(30, 10, 512)经过一次linear network，*

*#得到(30, 10, 512).*

记得上次有个问题提出来了，source sequence是否需要mask，为什么？其实是需要的，因为一个batch里面，可能序列的长短不一，有些是10个词打满，有些可能只有8个词，则这个时候，剩下的2个词的位置会被填充为”<blank>”这样的词，从而mask的时候，需要知道哪些地方是”<blank>”，这些位置不参与softmax以及序列的整体向量化表示等进一步的计算。

未完待续。



**The Annotated Transformer的中文注释版（2）**

开篇一个小问题：假设源语言词表3万，目标语言词表也是3万；Encoder六层，Decoder六层，8头，d\_model=512，那么Transformer中的参数的数量级是？：

（1）百万；（2）千万；（3）亿；（4）十亿。

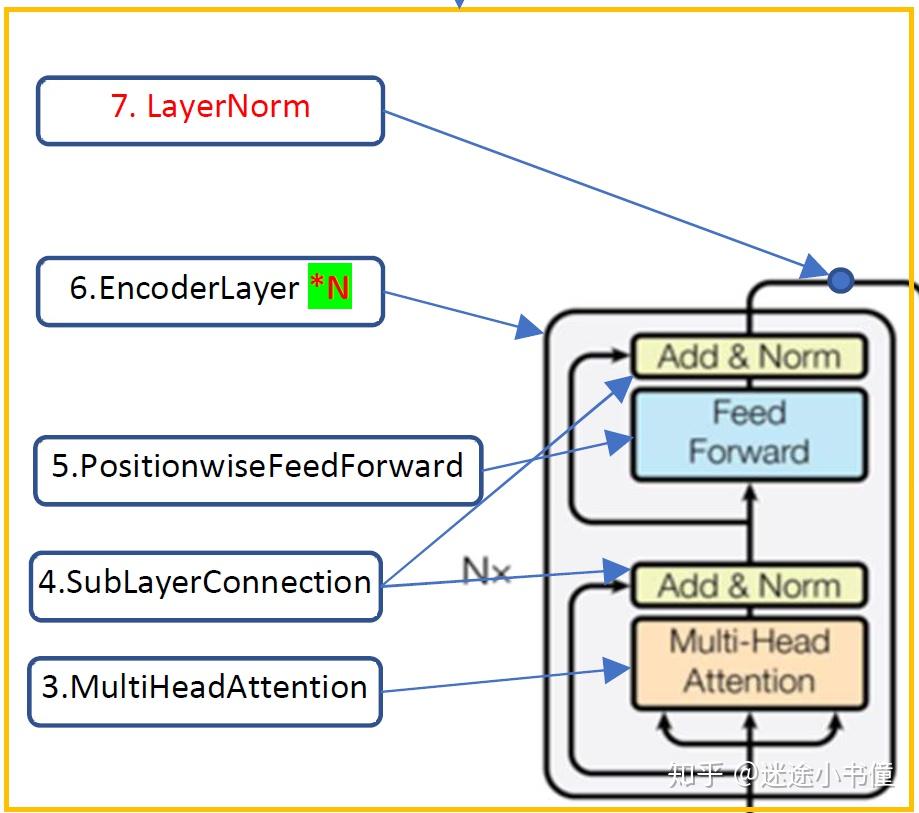
题外话：从深层次思考一下，为什么Transformer这么厉害？最牛的一些预训练模型，例如BERT等，都基于transformer的一些核心思想。从text transformer，neural machine translation双语和多语言机器翻译；到image transformer，像素点的hierarchical 级别的self-attentions，再到speech recognition with transformers, text-to-speech with transformers，甚至video-understanding with transformers。从self-attention机制来看，一个词的编码，是其“社会关系”的总和，所谓“**社会关系**”，体现在NLP的句子上，是一个词和周围的词的语法语义依赖关系，而且这种关系，主要是依靠context来实现，不需要任何人工标注的工作，这无疑让超大规模的单语言corpus大放光彩，甚至可以容易地为多语言的dense空间表示提供直接映射，让多年依赖双语对的机器翻译，能基于更少的双语对齐资源，构建更强大的翻译系统。另外，深度parsing方面，甚至transformer的论文里面，直接是应用到NMT和PCFG shift-reduce dependency parsing。随着模型参数的指数级别爆炸，其对信息检索，基于AI的内容创造，机器翻译，SR/TTS，视频理解等多个可以落地的应用，具有深远的影响。

接下来的一些核心问题，包括transformer和类似的预训练架构，如何适用于“可控”的应用场景，例如垂直领域知识图谱，图神经网络，更一般化的可解释的商品推荐，以及基于“社会关系”角色的chatbot的构建：我和你说啥，取决于我和你之间的社会关系；chatbot和我说啥，取决于我对chatbot和我的关系的设定，进化和升华。

回到正题，上一篇讲了1.Embeddings 词嵌入, 2.PositionalEncoding 位置编码，以及3.MultiHeadAttention 多头注意力。

本篇章会做个比较难的挑战：**统计出来transformer中的可训练参数的个数**。

走起~~~



我们继续看下一个类，4.SubLayerConnection 子层连接

所谓SubLayer，指的是两个部分：3.MultiHeadAttention以及4.PositionwiseFeedForward。特别要提的一点是，这两个子层的输入和输出，都是类似于(batch.size, sequence.length, d\_model)这样的tensor。

我们提前先看一下要对一个SubLayer进行Norm所需要的一个类7.LayerNorm:（**更正一下**，这个类用来对每个sublayer的输出进行处理，也对6层EncoderLayer之后的输出进行处理！）

**class** **LayerNorm**(nn**.**Module):

"Construct a layernorm module (See citation for details)."

**def** \_\_init\_\_(self, features, eps**=**1e-6):

*# features=d\_model=512, eps=epsilon 用于分母的非0化平滑*

super(LayerNorm, self)**.**\_\_init\_\_()

self**.**a\_2 **=** nn**.**Parameter(torch**.**ones(features))

*# a\_2 是一个可训练参数向量，(512)*

self**.**b\_2 **=** nn**.**Parameter(torch**.**zeros(features))

*# b\_2 也是一个可训练参数向量, (512)*

self**.**eps **=** eps

**def** **forward**(self, x):

*# x 的形状为(batch.size, sequence.len, 512)*

mean **=** x**.**mean(**-**1, keepdim**=**True)

*# 对x的最后一个维度，取平均值，得到tensor (batch.size, seq.len)*

std **=** x**.**std(**-**1, keepdim**=**True)

*# 对x的最后一个维度，取标准方差，得(batch.size, seq.len)*

**return** self**.**a\_2 **\*** (x **-** mean) **/** (std **+** self**.**eps) **+** self**.**b\_2

*# 本质上类似于（x-mean)/std，不过这里加入了两个可训练向量*

*# a\_2 and b\_2，以及分母上增加一个极小值epsilon，用来防止std为0*

*# 的时候的除法溢出*

上面这个类**有两个可训练参数，a\_2和b\_2**，分别是(512)维度的向量。

这个类的操作相对简单，有兴趣的同学，可以尝试一下下面的代码，了解一下具体咋算的：

**import** torch

**import** numpy **as** np

a **=** np**.**asarray([[1.,2.],[3., 7.]])

**print** (a) *# 2\*2*

*#[[1. 2.]*

*# [3. 7.]]*

b **=** torch**.**from\_numpy(a)

**print** (b) *# torch tensor, 2\*2*

*#tensor([[1., 2.],*

*# [3., 7.]], dtype=torch.float64)*

mean **=** b**.**mean(**-**1, keepdims**=**True)

**print**(mean) *# (2\*1)*

*#tensor([[1.5000],*

*# [5.0000]], dtype=torch.float64)*

std **=** b**.**std(**-**1, keepdims**=**True)

**print**(std) *# (2\*1)*

*#tensor([[0.7071],*

*# [2.8284]], dtype=torch.float64)*

**print**((b**-**mean)**/**std) *# (2\*2)*

*#tensor([[-0.7071, 0.7071],*

*# [-0.7071, 0.7071]], dtype=torch.float64)*

'''

(1-1.5)/0.7071

-0.7071135624381276

(2-1.5)/0.7071

0.7071135624381276

(3-5)/2.8284

-0.7071135624381276

(7-5)/2.8284

0.7071135624381276

'''

下面看：

4.SubLayerConnection类主要实现两个功能，残差Add以及Norm（使用上面的LayerNorm类）：

**class** **SublayerConnection**(nn**.**Module):

"""

A residual connection followed by a layer norm.

Note for code simplicity the norm is first as opposed to last.

"""

**def** \_\_init\_\_(self, size, dropout):

*# size=d\_model=512; dropout=0.1*

super(SublayerConnection, self)**.**\_\_init\_\_()

self**.**norm **=** LayerNorm(size) *# (512)，用来定义a\_2和b\_2*

self**.**dropout **=** nn**.**Dropout(dropout)

**def** **forward**(self, x, sublayer):

"Apply residual connection to any sublayer with the "

"same size."

*# x is alike (batch.size, sequence.len, 512)*

*# sublayer是一个具体的MultiHeadAttention*

*#或者PositionwiseFeedForward对象*

**return** x **+** self**.**dropout(sublayer(self**.**norm(x)))

*# x (30, 10, 512) -> norm (LayerNorm) -> (30, 10, 512)*

*# -> sublayer (MultiHeadAttention or PositionwiseFeedForward)*

*# -> (30, 10, 512) -> dropout -> (30, 10, 512)*

*# 然后输入的x（没有走sublayer) + 上面的结果，*

*#即实现了残差相加的功能*

上面这个类，没有自己的可训练参数。(self.norm里面，即LayerNorm类，有1024个可训练参数）

5.PositionwiseFeedForward类相对简单，就是一个全连接层：

**class** **PositionwiseFeedForward**(nn**.**Module):

"Implements FFN equation."

**def** \_\_init\_\_(self, d\_model, d\_ff, dropout**=**0.1):

*# d\_model = 512*

*# d\_ff = 2048 = 512\*4*

super(PositionwiseFeedForward, self)**.**\_\_init\_\_()

self**.**w\_1 **=** nn**.**Linear(d\_model, d\_ff)

*# 构建第一个全连接层，(512, 2048)，其中有两种可训练参数：*

*# weights矩阵，(512, 2048)，以及*

*# biases偏移向量, (2048)*

self**.**w\_2 **=** nn**.**Linear(d\_ff, d\_model)

*# 构建第二个全连接层, (2048, 512)，两种可训练参数：*

*# weights矩阵，(2048, 512)，以及*

*# biases偏移向量, (512)*

self**.**dropout **=** nn**.**Dropout(dropout)

**def** **forward**(self, x):

*# x shape = (batch.size, sequence.len, 512)*

*# 例如, (30, 10, 512)*

**return** self**.**w\_2(self**.**dropout(F**.**relu(self**.**w\_1(x))))

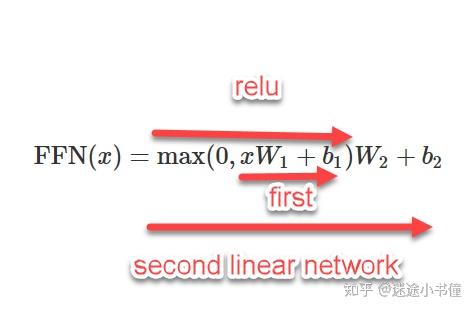
*# x (30, 10, 512) -> self.w\_1 -> (30, 10, 2048)*

*# -> relu -> (30, 10, 2048)*

*# -> dropout -> (30, 10, 2048)*

*# -> self.w\_2 -> (30, 10, 512)是输出的shape*

其，实现的是：



FFN(x)=max(0,xW1+b1)W2+b2

其中max(0, xW1 + b1) 是一个ReLU激活函数。

上面的可训练参数包括两个权重矩阵，(512, 2048)以及(2048, 512)，以及两个偏移bias向量，(2048)和(512)。则总共的可训练参数的个数是：2\*512\*2048 + 2048 + 512 = 2,099,712。一个全连接层，轻松达到2百万个参数。

下面这个clones方法，实现一个网络的深copy，也就是说一个新的对象，和原来的对象，完全分离，不分享任何存储空间：（从而保证可训练参数，都有自己的取值，梯度）

**def** **clones**(module, N):

"Produce N identical layers."

**return** nn**.**ModuleList([copy**.**deepcopy(module) **for** \_ **in** range(N)])

介绍了两个sublayer之后，我们来看看一个6.EncoderLayer的构成：

**class** **EncoderLayer**(nn**.**Module):

"Encoder is made up of self-attn and "

"feed forward (defined below)"

**def** \_\_init\_\_(self, size, self\_attn, feed\_forward, dropout):

*# size=d\_model=512*

*# self\_attn = MultiHeadAttention对象, first sublayer*

*# feed\_forward = PositionwiseFeedForward对象，second sublayer*

*# dropout = 0.1 (e.g.)*

super(EncoderLayer, self)**.**\_\_init\_\_()

self**.**self\_attn **=** self\_attn

self**.**feed\_forward **=** feed\_forward

self**.**sublayer **=** clones(SublayerConnection(size, dropout), 2)

*# 使用深度克隆方法，完整地复制出来两个SublayerConnection*

self**.**size **=** size *# 512*

**def** **forward**(self, x, mask):

"Follow Figure 1 (left) for connections."

*# x shape = (30, 10, 512)*

*# mask 是(batch.size, 10,10)的矩阵，类似于当前一个词w，有哪些词是w可见的*

*# 源语言的序列的话，所有其他词都可见，除了"<blank>"这样的填充；*

*# 目标语言的序列的话，所有w的左边的词，都可见。*

x **=** self**.**sublayer[0](x,

**lambda** x: self**.**self\_attn(x, x, x, mask))

*# x (30, 10, 512) -> self\_attn (MultiHeadAttention)*

*# shape is same (30, 10, 512) -> SublayerConnection*

*# -> (30, 10, 512)*

**return** self**.**sublayer[1](x, self**.**feed\_forward)

*# x 和feed\_forward对象一起，给第二个SublayerConnection*

可以看到EncoderLayer没有自己独立的可训练参数，所有的参数，都来自其他地方：

其一，一个MultiHeadAttention对象，里面有4个Linear network；参数个数是：4\*(512\*512+512)=1,050,624

其二，一个PositionwiseFeedForward对象，里面有2个Linear network；参数个数是：2,099,712。

其三，两个SublayerConnection对象，每个对象使用一个LayerNorm对象，LayerNorm里面有两个可训练参数向量a\_2和b\_2。则，可训练参数个数是：

2 \* (512 + 512) = 2048。

综合上面三个部分：一共有: 1,050,624 + 2,099,712 + 2048 = **3,152,384**个参数。（一个EncoderLayer)

另外稍微难理解的是这里使用了lambda来定义一个匿名函数，展开类似于：

**def** **lambda1**(self, x, mask):

**return** self**.**self\_attn(x, x, x, mask)

然后把这个lambda1传递给self.sublayer[0]。

进一步，看看8.Encoder类的实现：

**class** **Encoder**(nn**.**Module):

"Core encoder is a stack of N layers"

**def** \_\_init\_\_(self, layer, N):

*# layer = one EncoderLayer object, N=6*

super(Encoder, self)**.**\_\_init\_\_()

self**.**layers **=** clones(layer, N)

*# 深copy，N=6，*

self**.**norm **=** LayerNorm(layer**.**size)

*# 定义一个LayerNorm，layer.size=d\_model=512*

*# 其中有两个可训练参数a\_2和b\_2*

**def** **forward**(self, x, mask):

"Pass the input (and mask) through each layer in turn."

*# x is alike (30, 10, 512)*

*# (batch.size, sequence.len, d\_model)*

*# mask是类似于(batch.size, 10, 10)的矩阵*

**for** layer **in** self**.**layers:

x **=** layer(x, mask)

*# 进行六次EncoderLayer操作*

**return** self**.**norm(x)

*# 最后做一次LayerNorm，最后的输出也是(30, 10, 512) shape*

数数这里的可训练参数的个数：

6层EncoderLayers: 每一层是 3,152,384个参数；

LayerNorm里面有两个512维度的向量，有1024个参数；

则一共有：3,152,384\*6 + 1,024 = **18,915,328**个参数(Encoder类)。

再假设，源语言词表大小是30,000词，则embeddings部分有30,000 \* 512 =**15,360,000**个参数（词嵌入矩阵）。

这样的话，源语言的Embeddings+Encoder，一共有**34,275,328**个可训练参数。

开始进入Decoder部分。

因为很多类都是重复使用的，我们看第一个不一样的类, 9.DecoderLayer，

**class** **DecoderLayer**(nn**.**Module):

"Decoder is made of self-attn, src-attn, "

"and feed forward (defined below)"

**def** \_\_init\_\_(self, size, self\_attn, src\_attn,

feed\_forward, dropout):

*# size = d\_model=512,*

*# self\_attn = one MultiHeadAttention object，目标语言序列的*

*# src\_attn = second MultiHeadAttention object, 目标语言序列*

*# 和源语言序列之间的*

*# feed\_forward 一个全连接层*

*# dropout = 0.1*

super(DecoderLayer, self)**.**\_\_init\_\_()

self**.**size **=** size *# 512*

self**.**self\_attn **=** self\_attn

self**.**src\_attn **=** src\_attn

self**.**feed\_forward **=** feed\_forward

self**.**sublayer **=** clones(SublayerConnection(size, dropout), 3)

*# 需要三个SublayerConnection, 分别在*

*# self.self\_attn, self.src\_attn, 和self.feed\_forward*

*# 的后边*

**def** **forward**(self, x, memory, src\_mask, tgt\_mask):

"Follow Figure 1 (right) for connections."

m **=** memory *# (batch.size, sequence.len, 512)*

*# 来自源语言序列的Encoder之后的输出，作为memory*

*# 供目标语言的序列检索匹配：（类似于alignment in SMT)*

x **=** self**.**sublayer[0](x,

**lambda** x: self**.**self\_attn(x, x, x, tgt\_mask))

*# 通过一个匿名函数，来实现目标序列的自注意力编码*

*# 结果扔给sublayer[0]:SublayerConnection*

x **=** self**.**sublayer[1](x,

**lambda** x: self**.**src\_attn(x, m, m, src\_mask))

*# 通过第二个匿名函数，来实现目标序列和源序列的注意力计算*

*# 结果扔给sublayer[1]:SublayerConnection*

**return** self**.**sublayer[2](x, self**.**feed\_forward)

*# 走一个全连接层，然后*

*# 结果扔给sublayer[2]:SublayerConnection*

这里估计一下DecoderLayer的可训练参数的个数：

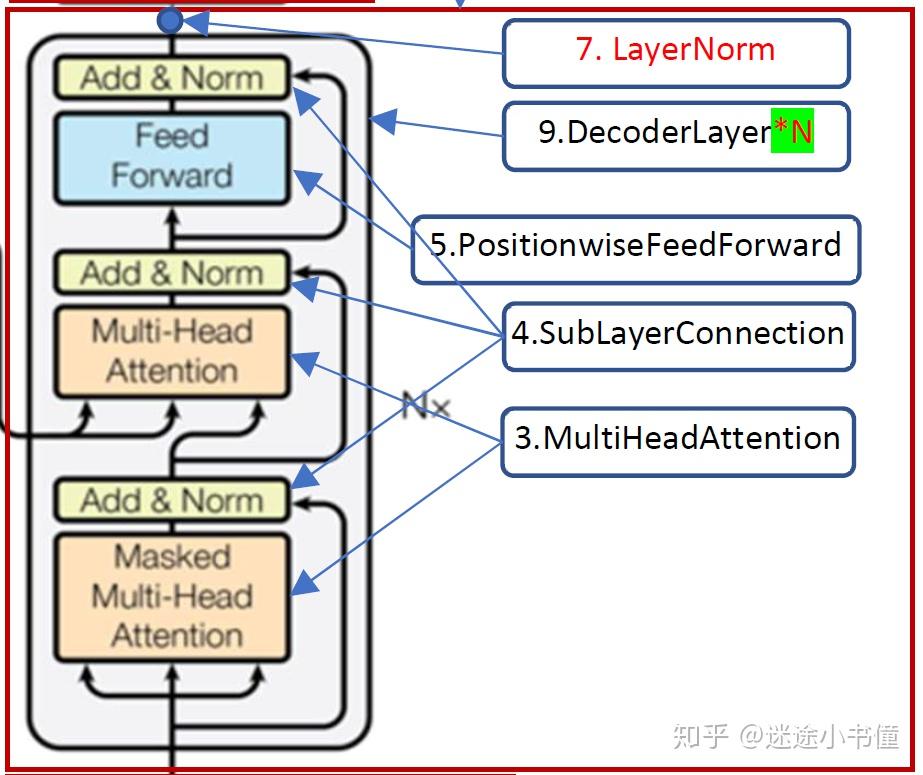
其一，两个MultiHeadAttention对象，每个对象有1,050,624个参数；

其二，feedforward对象，2,099,712个参数；

其三，三个SublayerConnection, 3\*(512+512) = 3,072个参数。

综合起来有，2\*1,050,624 + 2,099,712 + 3,072 =**4,204,032**个参数 (DecoderLayer)。

继续看看Decoder部分：



**class** **Decoder**(nn**.**Module):

"Generic N layer decoder with masking."

**def** \_\_init\_\_(self, layer, N):

*# layer = DecoderLayer object*

*# N = 6*

super(Decoder, self)**.**\_\_init\_\_()

self**.**layers **=** clones(layer, N)

*# 深度copy六次DecoderLayer*

self**.**norm **=** LayerNorm(layer**.**size)

*# 初始化一个LayerNorm*

**def** **forward**(self, x, memory, src\_mask, tgt\_mask):

**for** layer **in** self**.**layers:

x **=** layer(x, memory, src\_mask, tgt\_mask)

*# 执行六次DecoderLayer*

**return** self**.**norm(x)

*# 执行一次LayerNorm*

Decoder类的可训练参数的个数是：

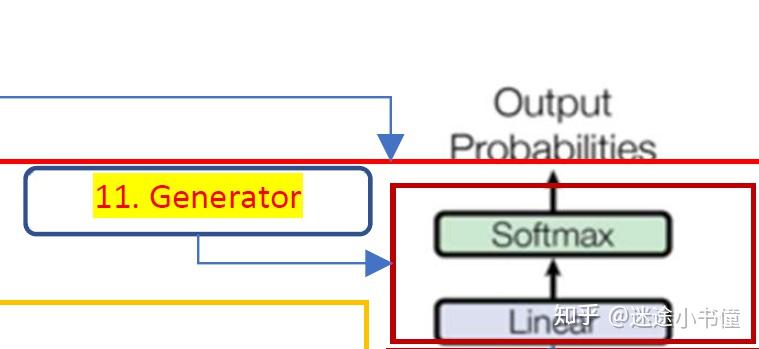
其一，六层DecoderLayer: 6\***4,204,032**；

其二，一个LayerNorm, 1,024个参数；

则一共有：6\***4,204,032** + 1024 =**25,225,216**个参数。

如果假设目标语言也是30,000的词表大小，则又增加30,000 \* 512=15,360,000个参数，从而有：**40,585,216**个可训练参数。

现在，看看最后的Generator部分：



**class** **Generator**(nn**.**Module):

"Define standard linear + softmax generation step."

**def** \_\_init\_\_(self, d\_model, vocab):

*# d\_model=512*

*# vocab = 目标语言词表大小*

super(Generator, self)**.**\_\_init\_\_()

self**.**proj **=** nn**.**Linear(d\_model, vocab)

*# 定义一个全连接层，可训练参数个数是(512 \* trg\_vocab\_size) +*

*# trg\_vocab\_size*

**def** **forward**(self, x):

**return** F**.**log\_softmax(self**.**proj(x), dim**=-**1)

*# x 类似于 (batch.size, sequence.length, 512)*

*# -> proj 全连接层 (30, 10, trg\_vocab\_size) = logits*

*# 对最后一个维度执行log\_soft\_max*

*# 得到(30, 10, trg\_vocab\_size)*

上面类里面有一个全连接层，参数个数是，一个权重矩阵 512\*trg\_vocab\_size，一个偏移向量 (trg\_vocab\_size)。

如果假设trg\_vocab\_size=30,000, 则可训练参数的个数是：

512\*30000 + 30000 = **15,390,000**个 (Generator)。

到目前位置，我们基本阐述完毕Encoder, Decoder，Generator，以及source sequence and target sequence的Embedding。

现在看看最外边的**EncoderDecoder**类：

**class** **EncoderDecoder**(nn**.**Module):

"""

A standard Encoder-Decoder architecture.

Base for this and many other models.

"""

**def** \_\_init\_\_(self, encoder, decoder,

src\_embed, tgt\_embed, generator):

super(EncoderDecoder, self)**.**\_\_init\_\_()

self**.**encoder **=** encoder

*# Encoder对象*

self**.**decoder **=** decoder

*# Decoder对象*

self**.**src\_embed **=** src\_embed

*# 源语言序列的编码，包括词嵌入和位置编码*

self**.**tgt\_embed **=** tgt\_embed

*# 目标语言序列的编码，包括词嵌入和位置编码*

self**.**generator **=** generator

*# 生成器*

**def** **forward**(self, src, tgt, src\_mask, tgt\_mask):

"Take in and process masked src and target sequences."

**return** self**.**decode(self**.**encode(src, src\_mask), src\_mask,

tgt, tgt\_mask)

*# 先对源语言序列进行编码，*

*# 结果作为memory传递给目标语言的编码器*

**def** **encode**(self, src, src\_mask):

*# src = (batch.size, seq.length)*

*# src\_mask 负责对src加掩码*

**return** self**.**encoder(self**.**src\_embed(src), src\_mask)

*# 对源语言序列进行编码，得到的结果为*

*# (batch.size, seq.length, 512)的tensor*

**def** **decode**(self, memory, src\_mask, tgt, tgt\_mask):

**return** self**.**decoder(self**.**tgt\_embed(tgt),

memory, src\_mask, tgt\_mask)

*# 对目标语言序列进行编码，得到的结果为*

*# (batch.size, seq.length, 512)的tensor*

上面的类里面，有五大属性，分别的可训练参数的个数是：

（假设源语言和目标语言的词表都是30,000)

源语言的Embeddings+Encoder，一共有**34,275,328**个可训练参数 (Embedding+Encoder);

假设目标语言也是30,000的词表大小，则又增加30,000 \* 512=15,360,000个参数，从而有：**40,585,216**个可训练参数(Embedding+Decoder);

如果假设trg\_vocab\_size=30,000, 则可训练参数的个数是：

512\*30000 + 30000 = **15,390,000**个(Generator).

**合计为：90,250,544个参数，9千万。稍稍扩大一下词表大小，轻松一亿个参数。所以，170亿参数，也没啥。**

当然，实际词嵌入矩阵，一般的维度小于512，可以是100维，200维等。

下面具体看看mask的构造函数：

**def** **subsequent\_mask**(size):

"Mask out subsequent positions."

*# e.g., size=10*

attn\_shape **=** (1, size, size) *# (1, 10, 10)*

subsequent\_mask **=** np**.**triu(np**.**ones(attn\_shape), k**=**1)

**.**astype('uint8')

*# triu: 负责生成一个三角矩阵，k-th对角线以下都是设置为0*

*# 上三角中元素为1.*

**return** torch**.**from\_numpy(subsequent\_mask) **==** 0

*# 反转上面的triu得到的上三角矩阵，修改为下三角矩阵。*

这里的k=1，1号对角线，不太容易理解：举例子如下：



继续看如何构造一个EncoderDecoder对象：

**def** **make\_model**(src\_vocab, tgt\_vocab, N**=**6,

d\_model**=**512, d\_ff**=**2048, h**=**8, dropout**=**0.1):

"Helper: Construct a model from hyperparameters."

*# src\_vocab = 源语言词表大小*

*# tgt\_vocab = 目标语言词表大小*

c **=** copy**.**deepcopy *# 对象的深度copy/clone*

attn **=** MultiHeadedAttention(h, d\_model) *# 8, 512*

*# 构造一个MultiHeadAttention对象*

ff **=** PositionwiseFeedForward(d\_model, d\_ff, dropout)

*# 512, 2048, 0.1*

*# 构造一个feed forward对象*

position **=** PositionalEncoding(d\_model, dropout)

*# 位置编码*

model **=** EncoderDecoder(

Encoder(EncoderLayer(d\_model, c(attn), c(ff), dropout), N),

Decoder(DecoderLayer(d\_model, c(attn), c(attn),

c(ff), dropout), N),

nn**.**Sequential(Embeddings(d\_model, src\_vocab), c(position)),

nn**.**Sequential(Embeddings(d\_model, tgt\_vocab), c(position)),

Generator(d\_model, tgt\_vocab))

*# This was important from their code.*

*# Initialize parameters with Glorot / fan\_avg.*

**for** p **in** model**.**parameters():

**if** p**.**dim() **>** 1:

nn**.**init**.**xavier\_uniform(p)

**return** model *# EncoderDecoder 对象*

然后调用make\_model，得到一个对象：

tmp\_model **=** make\_model(10, 10, 2)

*# src\_vocab\_size = 10*

*# tgt\_vocab\_size = 10*

*# N = 2, number for EncoderLayer and DecoderLayer*

**for** name, param **in** tmp\_model**.**named\_parameters():

**if** param**.**requires\_grad:

**print** (name, param**.**data**.**shape)

**else**:

**print** ('no gradient necessary', name, param**.**data**.**shape)

最后，看全量模型：

tmp\_model **=** make\_model(30000, 30000, 6)

*# src\_vocab\_size=30000, tgt\_vocab\_size=30000, N=6*

*#None*

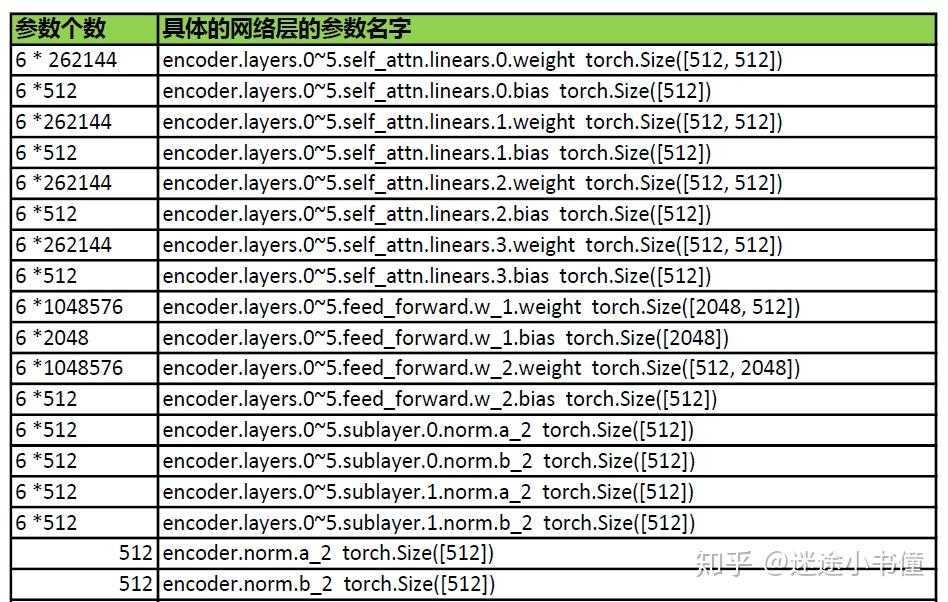
**for** name, param **in** tmp\_model**.**named\_parameters():

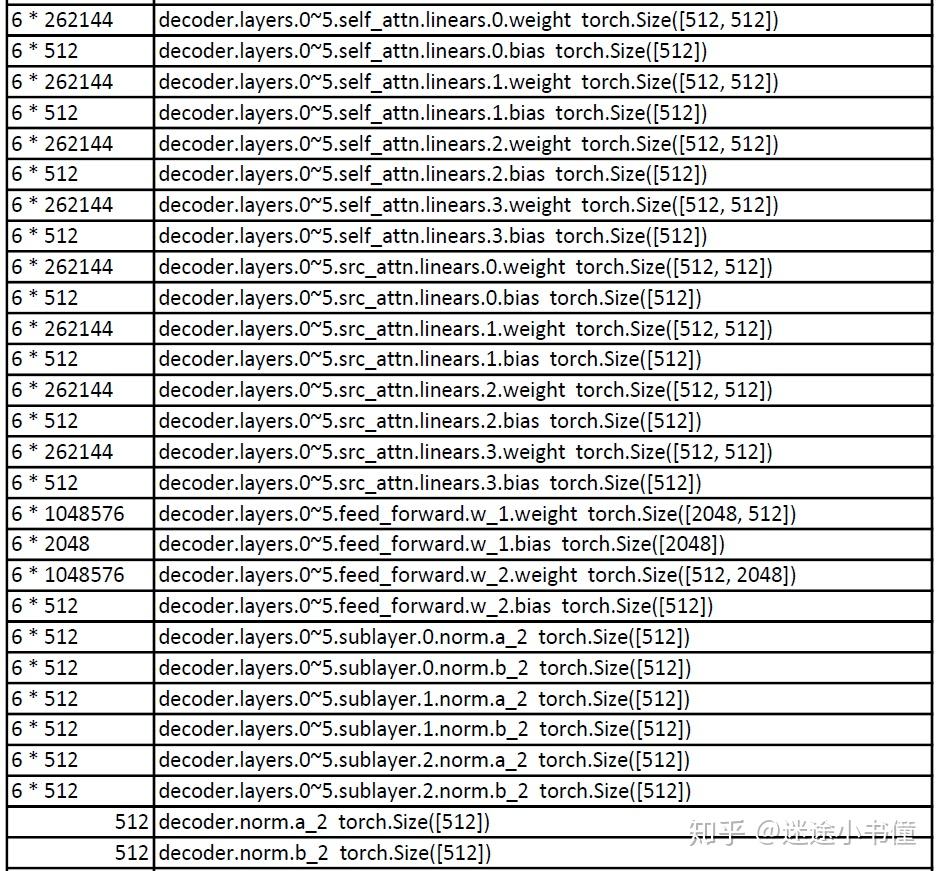
**if** param**.**requires\_grad:

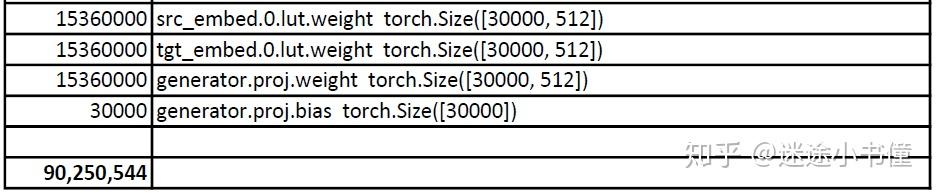
**print** (name, param**.**data**.**shape)

**else**:

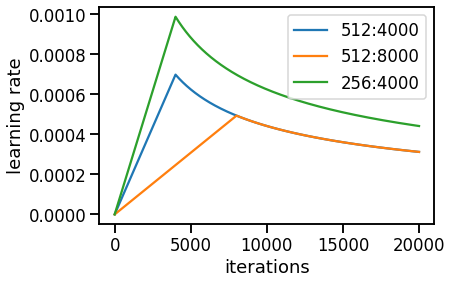
**print** ('no gradient necessary', name, param**.**data**.**shape)







未完待续~~~



**The Annotated Transformer的中文注释版(3)**

到目前位置，Transformer的Embeddings, Encoder, Decoder, Generator等部分的细节代码都阐述完毕了。按照pytorch的标准写法，一般先是独立定义一个个的class，class里面包括\_\_init\_\_和forward两个重要的函数。\_\_init\_\_负责“武装自己”，forward负责“处理别人”。

本次主要计划讲述损失函数的定义，训练和实际解码测试的流程。

先从构建batch和mask开始：

**class** **Batch**:

"Object for holding a batch of data with mask during training."

**def** \_\_init\_\_(self, src, trg**=**None, pad**=**0):

*# src: 源语言序列，(batch.size, src.seq.len)*

*# 二维tensor，第一维度是batch.size；第二个维度是源语言句子的长度*

*# 例如：[ [2,1,3,4], [2,3,1,4] ]这样的二行四列的，*

*# 1-4代表每个单词word的id*

*# trg: 目标语言序列，默认为空，其shape和src类似*

*# (batch.size, trg.seq.len)，*

*#二维tensor，第一维度是batch.size；第二个维度是目标语言句子的长度*

*# 例如trg=[ [2,1,3,4], [2,3,1,4] ] for a "copy network"*

*# (输出序列和输入序列完全相同）*

*# pad: 源语言和目标语言统一使用的 位置填充符号，'<blank>'*

*# 所对应的id，这里默认为0*

*# 例如，如果一个source sequence，长度不到4，则在右边补0*

*# [1,2] -> [1,2,0,0]*

self**.**src **=** src

self**.**src\_mask **=** (src **!=** pad)**.**unsqueeze(**-**2)

*# src = (batch.size, seq.len) -> != pad ->*

*# (batch.size, seq.len) -> usnqueeze ->*

*# (batch.size, 1, seq.len) 相当于在倒数第二个维度扩展*

*# e.g., src=[ [2,1,3,4], [2,3,1,4] ]对应的是*

*# src\_mask=[ [[1,1,1,1], [1,1,1,1]] ]*

**if** trg **is** **not** None:

self**.**trg **=** trg[:, :**-**1] *# 重要*

*# trg 相当于目标序列的前N-1个单词的序列*

*#（去掉了最后一个词）*

self**.**trg\_y **=** trg[:, 1:]

*# trg\_y 相当于目标序列的后N-1个单词的序列*

*# (去掉了第一个词）*

*# 目的是(src + trg) 来预测出来(trg\_y)，*

*# 这个在illustrated transformer中详细图示过。*

self**.**trg\_mask **=** \

self**.**make\_std\_mask(self**.**trg, pad)

self**.**ntokens **=** (self**.**trg\_y **!=** pad)**.**data**.**sum()

@staticmethod

**def** **make\_std\_mask**(tgt, pad):

"Create a mask to hide padding and future words."

*# 这里的tgt类似于：*

*#[ [2,1,3], [2,3,1] ] （最初的输入目标序列，分别去掉了最后一个词*

*# pad=0, '<blank>'的id编号*

tgt\_mask **=** (tgt **!=** pad)**.**unsqueeze(**-**2)

*# 得到的tgt\_mask类似于*

*# tgt\_mask = tensor([[[1, 1, 1]],[[1, 1, 1]]], dtype=torch.uint8)*

*# shape=(2,1,3)*

tgt\_mask **=** tgt\_mask **&** Variable(

subsequent\_mask(tgt**.**size(**-**1))**.**type\_as(tgt\_mask**.**data))

*# 先看subsequent\_mask, 其输入的是tgt.size(-1)=3*

*# 这个函数的输出为= tensor([[[1, 0, 0],*

*# [1, 1, 0],*

*# [1, 1, 1]]], dtype=torch.uint8)*

*# type\_as 把这个tensor转成tgt\_mask.data的type(也是torch.uint8)*

*# 这样的话，&的两边的tensor分别是(2,1,3), (1,3,3);*

*#tgt\_mask = tensor([[[1, 1, 1]],[[1, 1, 1]]], dtype=torch.uint8)*

*#and*

*# tensor([[[1, 0, 0], [1, 1, 0], [1, 1, 1]]], dtype=torch.uint8)*

*# (2,3,3)就是得到的tensor*

*# tgt\_mask.data = tensor([[[1, 0, 0],*

*# [1, 1, 0],*

*# [1, 1, 1]],*

*#[[1, 0, 0],*

*# [1, 1, 0],*

*# [1, 1, 1]]], dtype=torch.uint8)*

**return** tgt\_mask

需要仔细看的是上面的 & 这个算子

tgt\_mask **&**

Variable(subsequent\_mask(tgt**.**size(**-**1))**.**type\_as(tgt\_mask**.**data))

*# &算子左边，当tgt\_mask,shape=(3,1,4)是：*

*# [1 1 1 1],*

*# [1 1 1 0], # attention here*

*# [1 1 1 1]*

*# 类似于第二个序列的最后一个词是'<blank>' 。*

*# &算子右边，的Variable()内部，shape=(1,4,4)，*

*# 是：*

*# 1 0 0 0*

*# 1 1 0 0*

*# 1 1 1 0*

*# 1 1 1 1*

*# 的时候。*

*# 这两者的&的结果是 (3, 4, 4)：*

*# 1 0 0 0*

*# 1 1 0 0*

*# 1 1 1 0*

*# 1 1 1 1*

*# 1 0 0 0*

*# 1 1 0 0*

*# 1 1 1 0*

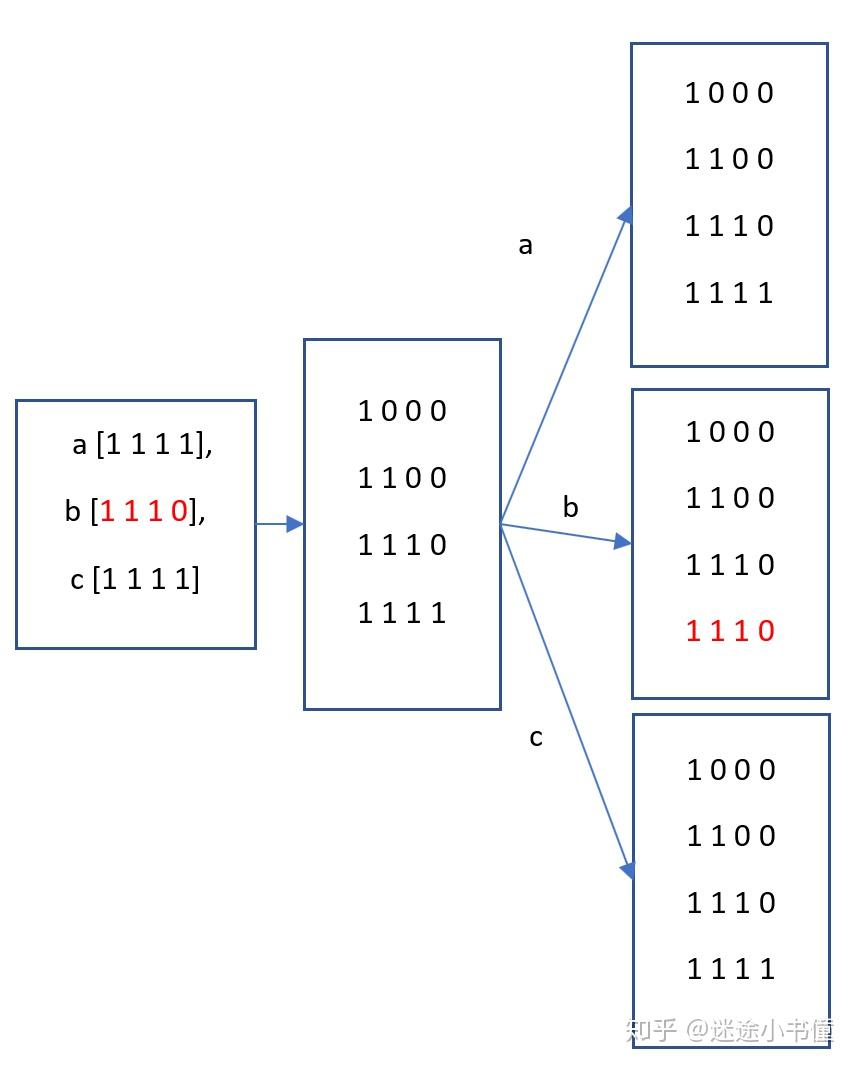
*# 1 1 1 0 # attention here， 因为第二个序列只有3个词*

*# 1 0 0 0*

*# 1 1 0 0*

*# 1 1 1 0*

*# 1 1 1 1*



下面看看优化算法，这里使用Adam算法：

**class** **NoamOpt**:

"Optim wrapper that implements rate."

**def** \_\_init\_\_(self, model\_size, factor, warmup, optimizer):

self**.**optimizer **=** optimizer

*# optimizer = Adam (Parameter Group 0*

*# amsgrad: False*

*# betas: (0.9, 0.98)*

*# eps: 1e-09*

*# lr: 0*

*# weight\_decay: 0*

*#)*

self**.**\_step **=** 0

self**.**warmup **=** warmup *# e.g., 4000 轮 热身*

self**.**factor **=** factor *# e.g., 2*

self**.**model\_size **=** model\_size *# 512*

self**.**\_rate **=** 0

**def** **step**(self):

"Update parameters and rate"

self**.**\_step **+=** 1

rate **=** self**.**rate()

**for** p **in** self**.**optimizer**.**param\_groups:

p['lr'] **=** rate

self**.**\_rate **=** rate

self**.**optimizer**.**step()

**def** **rate**(self, step **=** None):

"Implement `lrate`(learning rate) above"

**if** step **is** None:

step **=** self**.**\_step

**return** self**.**factor **\*** \

(self**.**model\_size **\*\*** (**-**0.5) **\***

min(step **\*\*** (**-**0.5), step **\*** self**.**warmup **\*\*** (**-**1.5)))

**def** **get\_std\_opt**(model):

**return** NoamOpt(model**.**src\_embed[0]**.**d\_model, 2, 4000,

torch**.**optim**.**Adam(model**.**parameters(), lr**=**0,

betas**=**(0.9, 0.98), eps**=**1e-9))

大概的思想是，前4000个batch先热身，这个时候，学习率不断上升；4000轮之后，学习率逐步下降。

*# Three settings of the lrate hyperparameters.*

opts **=** [NoamOpt(512, 1, 4000, None),

NoamOpt(512, 1, 8000, None),

NoamOpt(256, 1, 4000, None)]

plt**.**plot(np**.**arange(1, 20000),

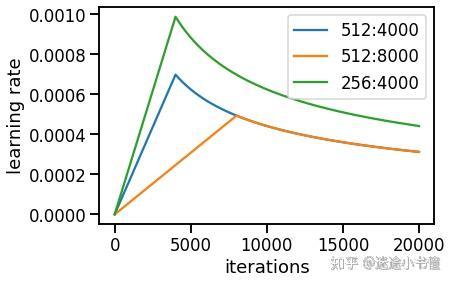
[[opt**.**rate(i) **for** opt **in** opts] **for** i **in** range(1, 20000)])

plt**.**legend(["512:4000", "512:8000", "256:4000"])

plt**.**xlabel('iterations')

plt**.**ylabel('learning rate')

None



下面进入有些难于理解的LabelSmoothing，简单思想就是劫富济贫。

这个方法虽然会损害Perplexity，但是会增加最后的BLEU score.

另外，这个名字不好，无论如何需要加上 KLLoss, 类似于：

LabelSmoothing -> **LabelSmoothingKLLoss** 其实是最终计算损失函数的。

**class** **LabelSmoothing**(nn**.**Module):

"Implement label smoothing."

**def** \_\_init\_\_(self, size, padding\_idx, smoothing**=**0.0):

super(LabelSmoothing, self)**.**\_\_init\_\_()

self**.**criterion **=** nn**.**KLDivLoss(size\_average**=**False)

self**.**padding\_idx **=** padding\_idx *# '<blank>' 的id*

self**.**confidence **=** 1.0 **-** smoothing *# 自留的概率值、得分 e.g. 0.6*

self**.**smoothing **=** smoothing *# 均分出去的概率值，得分 e.g. 0.4*

self**.**size **=** size *# target vocab size 目标语言词表大小*

self**.**true\_dist **=** None

**def** **forward**(self, x, target):

"in real-world case: 真实情况下"

*# x的shape为(batch.size \* seq.len, target.vocab.size)*

*# y的shape是(batch.size \* seq.len)*

*# x=logits，(seq.len, target.vocab.size)*

*# 每一行，代表一个位置的词*

*# 类似于：假设seq.len=3, target.vocab.size=5*

*# x中保存的是log(prob)*

*#x = tensor([[-20.7233, -1.6094, -0.3567, -2.3026, -20.7233],*

*#[-20.7233, -1.6094, -0.3567, -2.3026, -20.7233],*

*#[-20.7233, -1.6094, -0.3567, -2.3026, -20.7233]])*

*# target 类似于：*

*# target = tensor([2, 1, 0])，torch.size=(3)*

**assert** x**.**size(1) **==** self**.**size *# 目标语言词表大小*

true\_dist **=** x**.**data**.**clone()

*# true\_dist = tensor([[-20.7233, -1.6094, -0.3567, -2.3026, -20.7233],*

*#[-20.7233, -1.6094, -0.3567, -2.3026, -20.7233],*

*#[-20.7233, -1.6094, -0.3567, -2.3026, -20.7233]])*

true\_dist**.**fill\_(self**.**smoothing **/** (self**.**size **-** 2))

*# true\_dist = tensor([[0.1333, 0.1333, 0.1333, 0.1333, 0.1333],*

*#[0.1333, 0.1333, 0.1333, 0.1333, 0.1333],*

*#[0.1333, 0.1333, 0.1333, 0.1333, 0.1333]])*

*# 注意，这里分母target.vocab.size-2是因为*

*# (1) 最优值 0.6要占一个位置；*

*# (2) 填充词 <blank> 要被排除在外*

*# 所以被激活的目标语言词表大小就是self.size-2*

true\_dist**.**scatter\_(1, target**.**data**.**unsqueeze(1),

self**.**confidence)

*# target.data.unsqueeze(1) ->*

*# tensor([[2],*

*#[1],*

*#[0]]); shape=torch.Size([3, 1])*

*# self.confidence = 0.6*

*# 根据target.data的指示，按照列优先(1)的原则，把0.6这个值*

*# 填入true\_dist: 因为target.data是2,1,0的内容，*

*# 所以，0.6填入第0行的第2列（列号，行号都是0开始）*

*# 0.6填入第1行的第1列*

*# 0.6填入第2行的第0列：*

*# true\_dist = tensor([[0.1333, 0.1333, 0.6000, 0.1333, 0.1333],*

*#[0.1333, 0.6000, 0.1333, 0.1333, 0.1333],*

*#[0.6000, 0.1333, 0.1333, 0.1333, 0.1333]])*

true\_dist[:, self**.**padding\_idx] **=** 0

*# true\_dist = tensor([[0.0000, 0.1333, 0.6000, 0.1333, 0.1333],*

*#[0.0000, 0.6000, 0.1333, 0.1333, 0.1333],*

*#[0.0000, 0.1333, 0.1333, 0.1333, 0.1333]])*

*# 设置true\_dist这个tensor的第一列的值全为0*

*# 因为这个是填充词'<blank>'所在的id位置，不应该计入*

*# 目标词表。需要注意的是，true\_dist的每一列，代表目标语言词表*

*#中的一个词的id*

mask **=** torch**.**nonzero(target**.**data **==** self**.**padding\_idx)

*# mask = tensor([[2]]), 也就是说，最后一个词 2,1,0中的0，*

*# 因为是'<blank>'的id，所以通过上面的一步，把他们找出来*

**if** mask**.**dim() **>** 0:

true\_dist**.**index\_fill\_(0, mask**.**squeeze(), 0.0)

*# 当target reference序列中有0这个'<blank>'的时候，则需要把*

*# 这一行的值都清空。*

*# 在一个batch里面的时候，可能两个序列长度不一，所以短的序列需要*

*# pad '<blank>'来填充，所以会出现类似于(2,1,0)这样的情况*

*# true\_dist = tensor([[0.0000, 0.1333, 0.6000, 0.1333, 0.1333],*

*# [0.0000, 0.6000, 0.1333, 0.1333, 0.1333],*

*# [0.0000, 0.0000, 0.0000, 0.0000, 0.0000]])*

self**.**true\_dist **=** true\_dist

**return** self**.**criterion(x,

Variable(true\_dist, requires\_grad**=**False))

*# 这一步就是调用KL loss来计算*

*# x = tensor([[-20.7233, -1.6094, -0.3567, -2.3026, -20.7233],*

*#[-20.7233, -1.6094, -0.3567, -2.3026, -20.7233],*

*#[-20.7233, -1.6094, -0.3567, -2.3026, -20.7233]])*

*# true\_dist=tensor([[0.0000, 0.1333, 0.6000, 0.1333, 0.1333],*

*# [0.0000, 0.6000, 0.1333, 0.1333, 0.1333],*

*# [0.0000, 0.0000, 0.0000, 0.0000, 0.0000]])*

*# 之间的loss了。细节可以参考我的那篇illustrated transformer*

举个简单的例子：

*# Example of label smoothing.*

crit **=** LabelSmoothing(5, 0, 0.4) *# trg.vocab.size=5, pad\_idx=0,*

*# smooth=0.4 and thus confidence=0.6*

predict **=** torch**.**FloatTensor([[0, 0.2, 0.7, 0.1, 0],

[0, 0.2, 0.7, 0.1, 0],

[0, 0.2, 0.7, 0.1, 0]])

*# predicted logits tensor*

"smooth only and important"

predict **=** predict**.**masked\_fill(predict **==** 0, 1e-9) *# "smooth only"*

v **=** crit(Variable(predict**.**log()),

Variable(torch**.**LongTensor([2, 1, 0])))

*# Show the target distributions expected by the system.*

plt**.**imshow(crit**.**true\_dist)

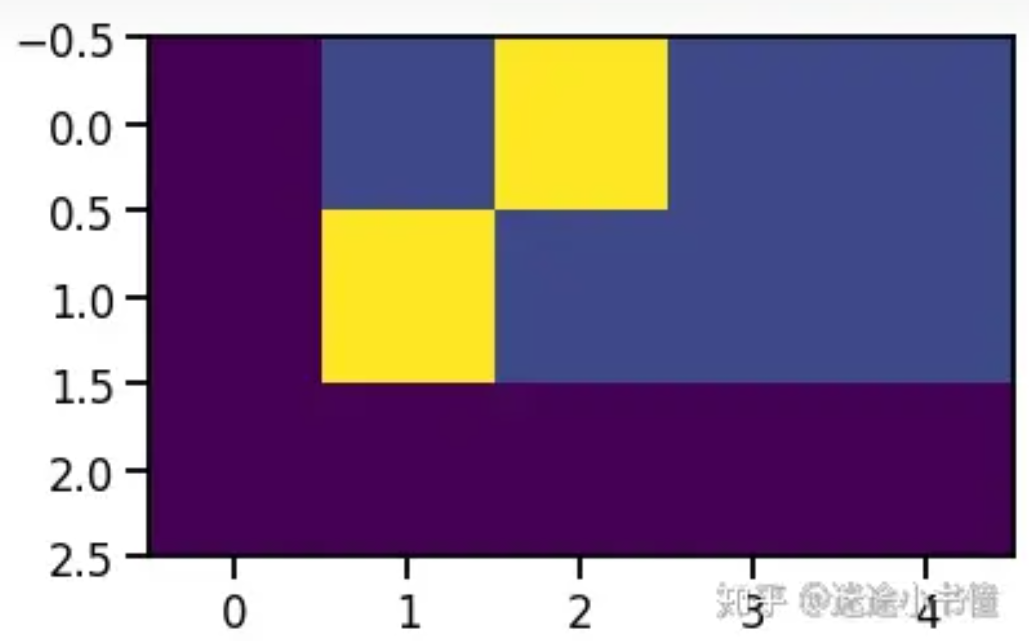
*#true\_dist=tensor([[0.0000, 0.1333, 0.6000, 0.1333, 0.1333],*

*# [0.0000, 0.6000, 0.1333, 0.1333, 0.1333],*

*# [0.0000, 0.0000, 0.0000, 0.0000, 0.0000]]),*

*# true\_dist.shape=torch.Size([3, 5])*

None



上图中，第0列，都是0，代表了'<blank>'的id位置；最后一行都是0，代表的是(2,1,0)中的最后一个词，对应的也是'<blank>'，所以最后一行都被设置为0.

这个类LabelSmoothing，一方面对label进行平滑，如果Model对于一个结果非常确信，则loss反而惩罚它（貌似缺少了多样性）；另外一方面则是对loss进行计算的。

下图很好的说明了，“过犹不及”的思想：

crit **=** LabelSmoothing(5, 0, 0.1)

**def** **loss**(x):

d **=** x **+** 3 **\*** 1

predict **=** torch**.**FloatTensor([[0, x **/** d, 1 **/** d, 1 **/** d, 1 **/** d],

])

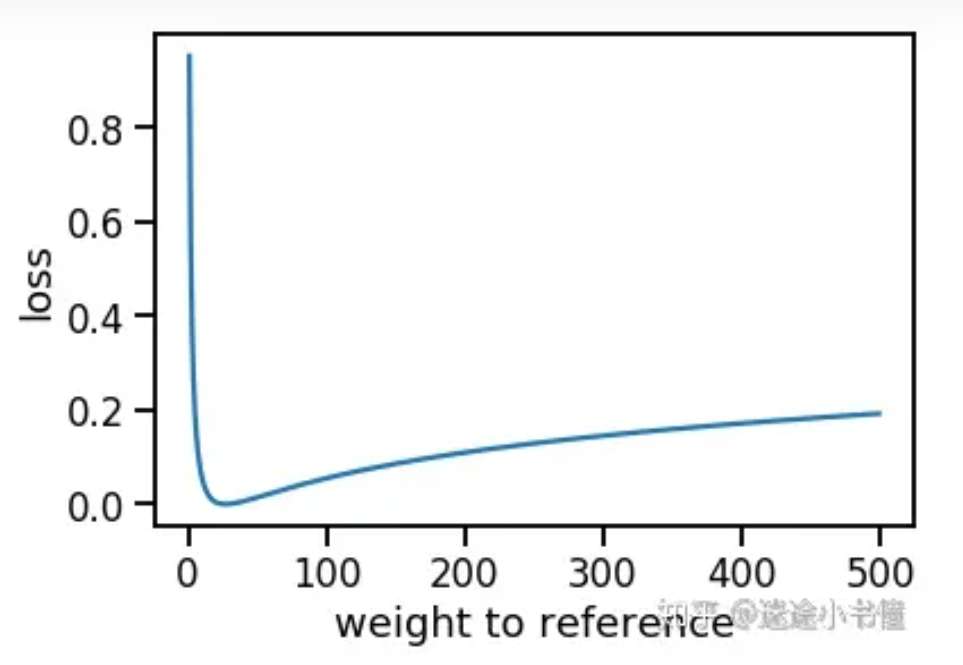
*#print(predict)*

**return** crit(Variable(predict**.**log()),

Variable(torch**.**LongTensor([1])))**.**data[0]

plt**.**plot(np**.**arange(1, 500), [loss(x) **for** x **in** range(1, 500)])

None



可以看到当模型的确信度非常大的时候，loss反而增加了。

下面通过一个简单的例子来整体串一下transformer:

假设我们的目的是构建一个copy network，输入和输出完全一致：

先构建数据集：

**def** **data\_gen**(V, batch, nbatches):

"Generate random data for a src-tgt copy task."

**for** i **in** range(nbatches):

data **=** torch**.**from\_numpy

(np**.**random**.**randint(1, V, size**=**(batch, 10)))

data[:, 0] **=** 1

src **=** Variable(data, requires\_grad**=**False)

tgt **=** Variable(data, requires\_grad**=**False)

**yield** Batch(src, tgt, 0)

例如：

src = tensor([[1, 4, 2, 1],  
[1, 4, 4, 4]], dtype=torch.int32)

tgt = tensor([[1, 4, 2, 1],  
[1, 4, 4, 4]], dtype=torch.int32)

假设batch.size=2, source.vocab.size=4=target.vocab.size

封装的基于batch来计算loss的类：

**class** **SimpleLossCompute**:

"A simple loss compute and train function."

**def** \_\_init\_\_(self, generator, criterion, opt**=**None):

self**.**generator **=** generator *# Generator 对象, linear+softmax*

self**.**criterion **=** criterion *# LabelSmooth对象，计算loss*

self**.**opt **=** opt *# NormOpt对象，优化算法对象*

**def** \_\_call\_\_(self, x, y, norm):

*# e.g., x为(2,3,8), batch.size=2, seq.len=3, d\_model=8*

*# y = tensor([[4, 2, 1],*

*#[4, 4, 4]], dtype=torch.int32)*

*# norm: (y=trg\_y中非'<blank>'的token的个数)*

"attention here"

x **=** self**.**generator(x)

loss **=** self**.**criterion(x**.**contiguous()**.**view(**-**1, x**.**size(**-**1)),

y**.**contiguous()**.**view(**-**1)) **/** norm**.**item()

*# 变形后，x类似于(batch.size\*seq.len, target.vocab.size)*

*# y为(target.vocab.size)*

*# 然后调用LabelSmooth来计算loss*

loss**.**backward()

**if** self**.**opt **is** **not** None:

self**.**opt**.**step()

self**.**opt**.**optimizer**.**zero\_grad()

*#return loss.data[0] \* norm*

"attention here"

**return** loss**.**data**.**item() **\*** norm**.**item()

**上面需要注意，新版本pytorch，使用.item()来取具体的数值。**

下面看一个相对完整的训练loop:

**def** **run\_epoch**(aepoch, data\_iter, model, loss\_compute):

"Standard Training and Logging Function"

*# data\_iter = 所有数据的打包*

*# model = EncoderDecoder 对象*

*# loss\_compute = SimpleLossCompute对象*

start **=** time**.**time()

total\_tokens **=** 0

total\_loss **=** 0

tokens **=** 0

**for** i, batch **in** enumerate(data\_iter):

*# 对每个batch循环*

out **=** model**.**forward(batch**.**src, batch**.**trg,

batch**.**src\_mask, batch**.**trg\_mask)

*# 使用目前的model，对batch.src+batch.trg进行forward*

*# e.g.,*

*# batch.src (2,4) = tensor([[1, 4, 2, 1],*

*# [1, 4, 4, 4]], dtype=torch.int32)*

*# batch.trg (2,3) = tensor([[1, 4, 2],*

*# [1, 4, 4]], dtype=torch.int32)*

*# batch.src\_mask (2,1,4) = tensor([[[1, 1, 1, 1]],*

*# [[1, 1, 1, 1]]], dtype=torch.uint8)*

*# batch.trg\_mask (2,3,3) = tensor([[[1, 0, 0],*

*# [1, 1, 0],*

*# [1, 1, 1]],*

*#[[1, 0, 0],*

*# [1, 1, 0],*

*#[1, 1, 1]]], dtype=torch.uint8)*

*# and out (2,3,8):*

*# out = tensor([[[-0.4749, -0.4887, 0.1245, -0.4042, 0.5301,*

*# 1.7662, -1.6224, 0.5694],*

*# [ 0.4683, -0.7813, 0.2845, 0.4464, -0.3088, -0.1751, -1.6643,*

*# 1.7303],*

*#[-1.1600, -0.2348, 1.0631, 1.3192, -0.9453, 0.3538, 0.7051...*

loss **=** loss\_compute(out, batch**.**trg\_y, batch**.**ntokens)

*# out和trg\_y计算Loss*

*# ntokens = 6 (trg\_y中非'<blank>'的token的个数)*

*# 注意，这里是token,不是unique word*

*# 例如[ [ [1, 2, 3], [2,3,4] ]中有6个token,而只有4个unique word*

total\_loss **+=** loss

total\_tokens **+=** batch**.**ntokens

tokens **+=** batch**.**ntokens

**if** i **%** 50 **==** 1:

elapsed **=** time**.**time() **-** start

"attention here 这里隐藏一个bug"

*#print("Epoch Step: %d Loss: %f Tokens per Sec: %f" %*

*# (i, loss / batch.ntokens, tokens / elapsed))*

**print** ('epoch step: {}:{} Loss: {}/{}, tokens per sec: {}/{}'

**.**format(aepoch, i, loss, batch**.**ntokens,

tokens, elapsed))

start **=** time**.**time()

tokens **=** 0

**return** total\_loss **/** total\_tokens

开始训练的准备工作：

*# Train the simple copy task.*

V **=** 5 *# here V is the vocab size of source and target languages (sequences)*

criterion **=** LabelSmoothing(size**=**V,

padding\_idx**=**0, smoothing**=**0.01) *# 创建损失函数计算对象*

model **=** make\_model(V, V, N**=**2, d\_model**=**8, d\_ff**=**16, h**=**2)

*# EncoderDecoder对象构造*

'''

in make\_model: src\_vocab\_size=11, tgt\_vocab\_size=11,

N=2, d\_model=512, d\_ff=2048, h=8, dropout=0.1

'''

model\_opt **=** NoamOpt(model**.**src\_embed[0]**.**d\_model, 1, 400,

torch**.**optim**.**Adam(model**.**parameters(),

lr**=**0, betas**=**(0.9, 0.98), eps**=**1e-9))

*# 模型最优化算法的对象*

训练流程代码：

lossfun **=** SimpleLossCompute(model**.**generator,

criterion, model\_opt)

**if** True:

**print** ('start model training...')

**for** epoch **in** range(10):

**print** ('epoch={}, training...'**.**format(epoch))

model**.**train() *# set the model into "train" mode*

*# 设置模型进入训练模式*

*#lossfun = SimpleLossCompute(model.generator,*

*# criterion, model\_opt) # 不需要在这里定义lossfun*

run\_epoch(epoch, data\_gen(V, 4, 2, 2), model, lossfun)

*# 重新构造一批数据，并执行训练*

model**.**eval() *# 模型进入evaluation模式 (dropout，反向传播无效）*

**print** ('evaluating...')

**print**(run\_epoch(epoch, data\_gen(V, 4, 2, 2), model,

SimpleLossCompute(model**.**generator,

criterion, None)))

*# 这里None表示优化函数为None，所以不进行参数更新*

贪心解码算法（只看top-1的结果）：

**def** **greedy\_decode**(model, src, src\_mask, max\_len, start\_symbol):

memory **=** model**.**encode(src, src\_mask)

*# 源语言的一个batch*

*# 执行encode编码工作，得到memory*

*# shape=(batch.size, src.seq.len, d\_model)*

*# src = (1,4), batch.size=1, seq.len=4*

*# src\_mask = (1,1,4) with all ones*

*# start\_symbol=1*

**print** ('memory={}, memory.shape={}'**.**format(memory,

memory**.**shape))

ys **=** torch**.**ones(1, 1)**.**fill\_(start\_symbol)**.**type\_as(src**.**data)

*# 最初ys=[[1]], size=(1,1); 这里start\_symbol=1*

**print** ('ys={}, ys.shape={}'**.**format(ys, ys**.**shape))

**for** i **in** range(max\_len**-**1): *# max\_len = 5*

out **=** model**.**decode(memory, src\_mask,

Variable(ys),

Variable(subsequent\_mask(ys**.**size(1))

**.**type\_as(src**.**data)))

*# memory, (1, 4, 8), 1=batch.size, 4=src.seq.len, 8=d\_model*

*# src\_mask = (1,1,4) with all ones*

*# out, (1, 1, 8), 1=batch.size, 1=seq.len, 8=d\_model*

**print** ('out={}, out.shape={}'**.**format(out, out**.**shape))

prob **=** model**.**generator(out[:, **-**1])

*# pick the right-most word*

*# (1=batch.size,8) -> generator -> prob=(1,5) 5=trg.vocab.size*

*# -1 for ? only look at the final (out) word's vector*

\_, next\_word **=** torch**.**max(prob, dim **=** 1)

next\_word **=** next\_word**.**data[0]

*# word id of "next\_word"*

ys **=** torch**.**cat([ys,

torch**.**ones(1, 1)**.**type\_as(src**.**data)**.**fill\_(next\_word)],

dim**=**1)

*# ys is in shape of (1,2) now, i.e., 2 words in current seq*

**return** ys

**if** True:

model**.**eval()

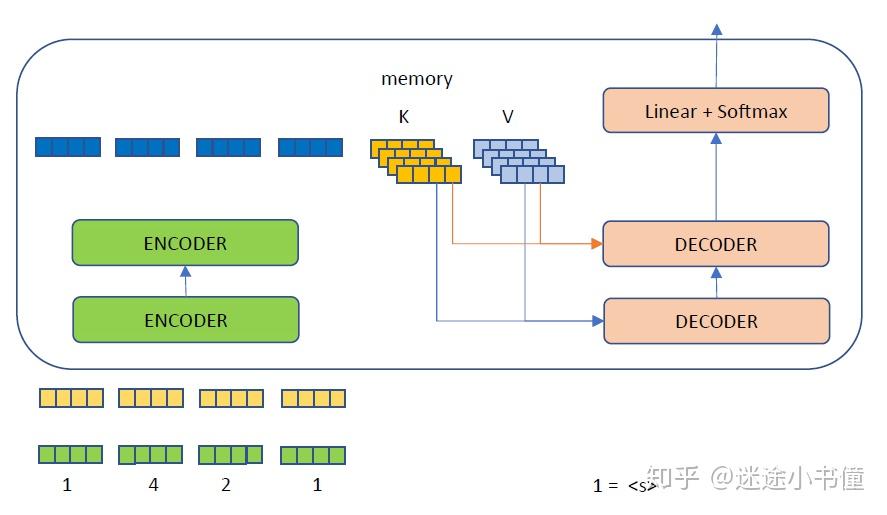
src **=** Variable(torch**.**LongTensor([[1,2,3,4]]))

src\_mask **=** Variable(torch**.**ones(1, 1, 4))

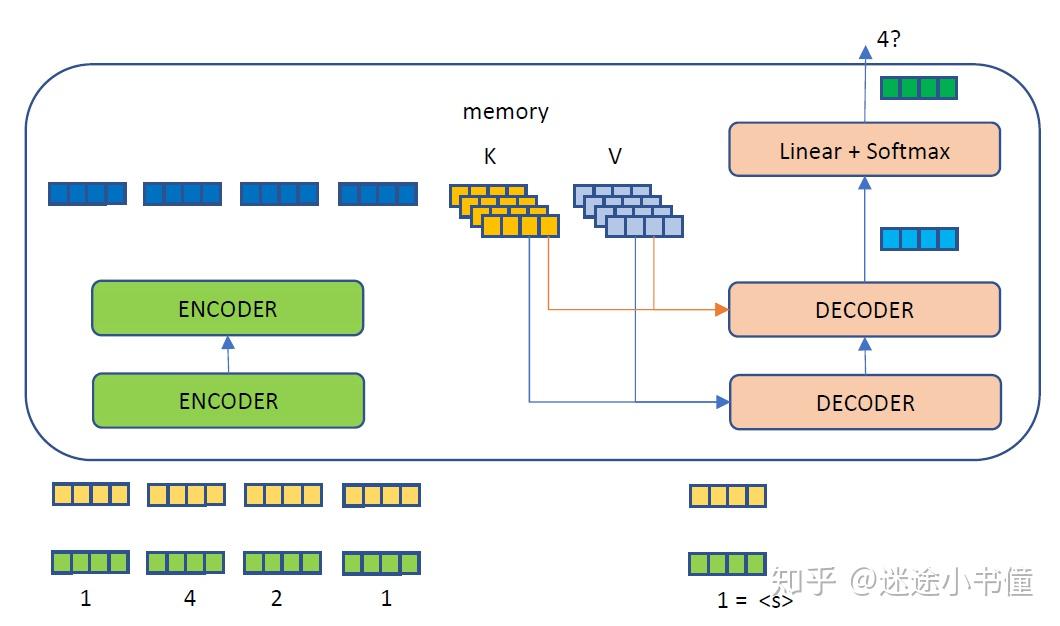
**print**(greedy\_decode(model, src, src\_mask, max\_len**=**5,

start\_symbol**=**1))

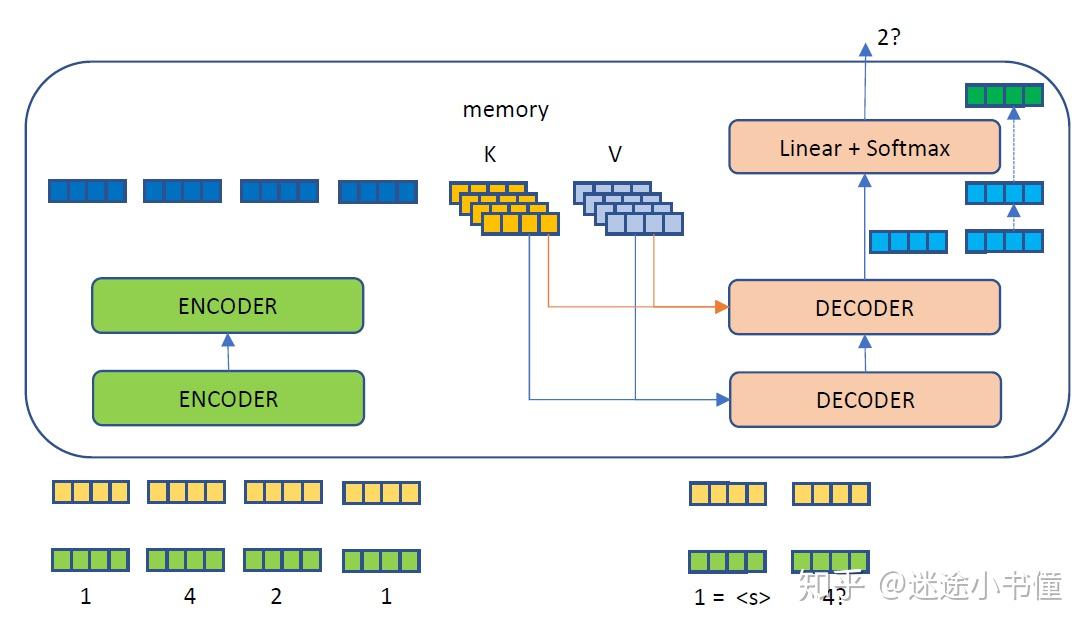
类似于下图所示：



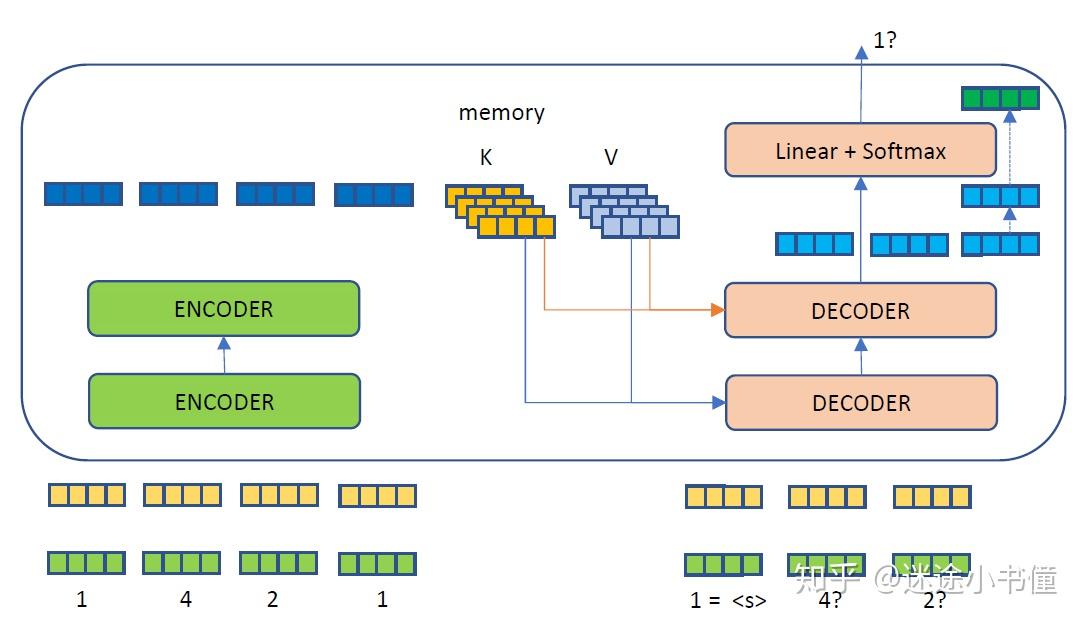
解码第一步，只有&amp;amp;amp;amp;lt;s&amp;amp;amp;amp;gt;作为输入



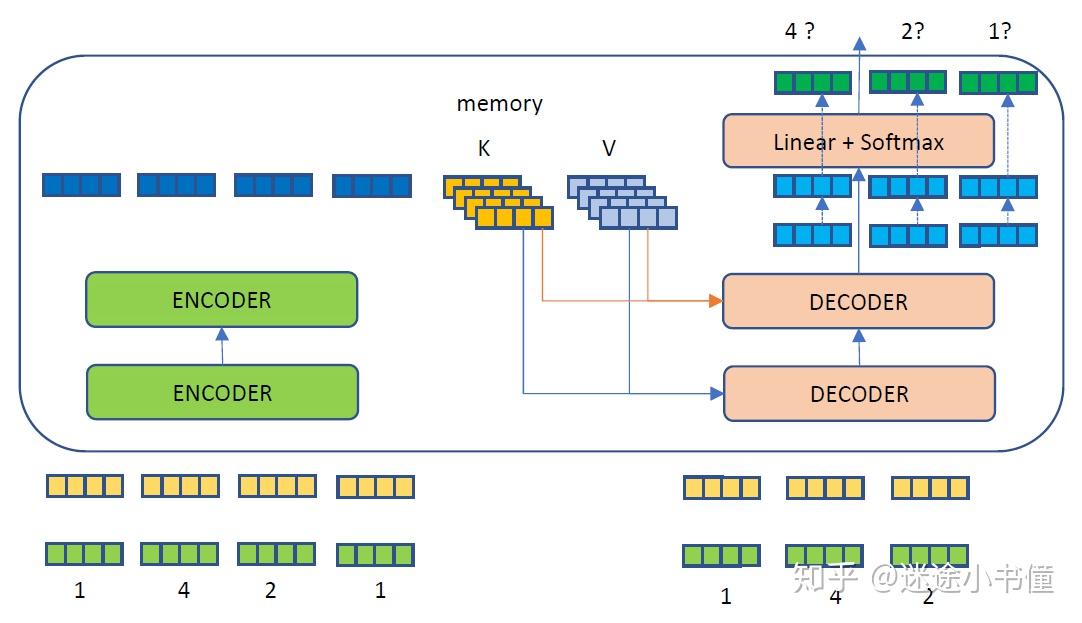
解码第二步，经过Decoder之后，得到4? 是预测出来的第一个词



解码第三步：根据&amp;amp;amp;amp;lt;s&amp;amp;amp;amp;gt;和4?来继续预测下一个词，例如得到2?

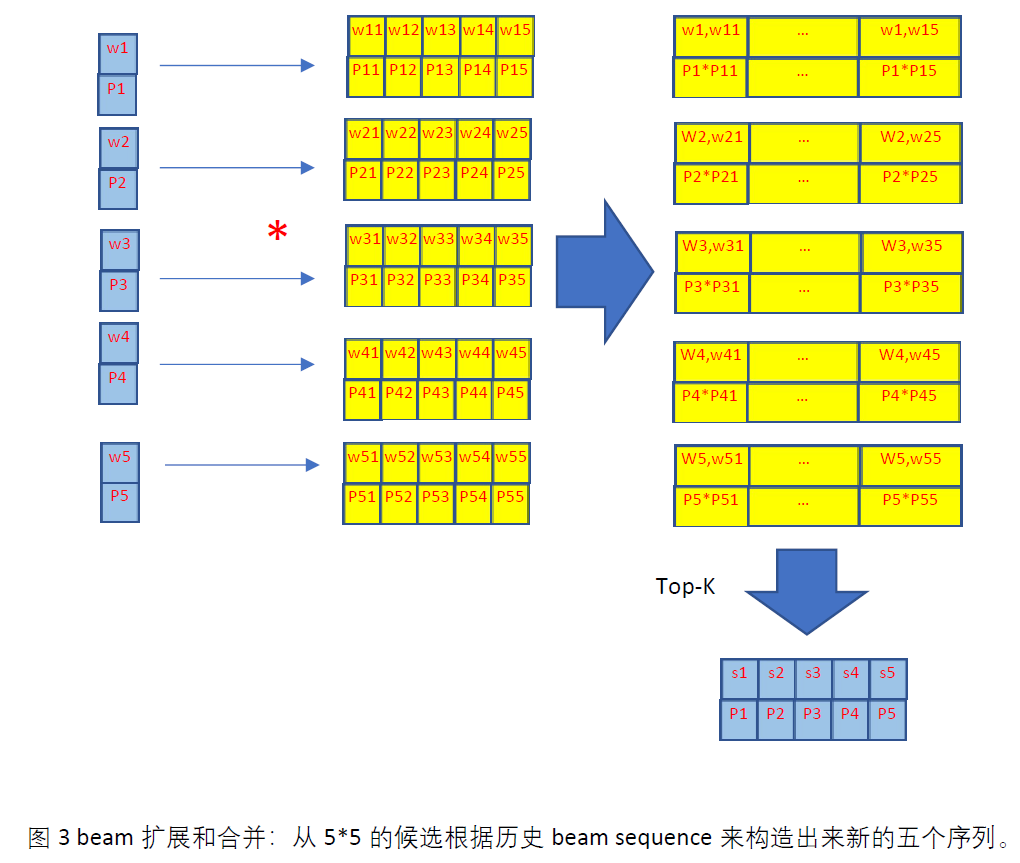


解码第四步，从&amp;amp;amp;amp;lt;s&amp;amp;amp;amp;gt;, 4?, 2?预测出下一个词1？



训练阶段的批量预测，1 4 2 -&amp;amp;amp;amp;gt; (4? 2? 1?)

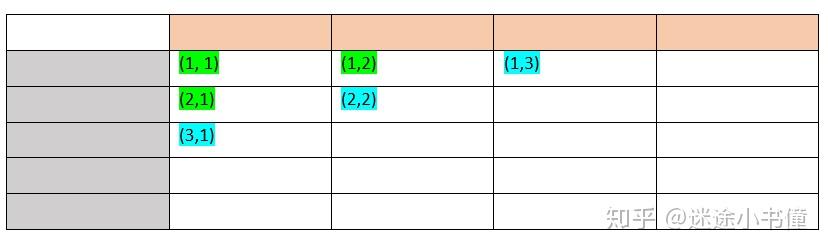
注意训练阶段不同，是一次性搞定。例如，目标序列一共四个词，则意思是target前3个词+ source 来预测后三个词(target)。



**Transformer中的beam search**

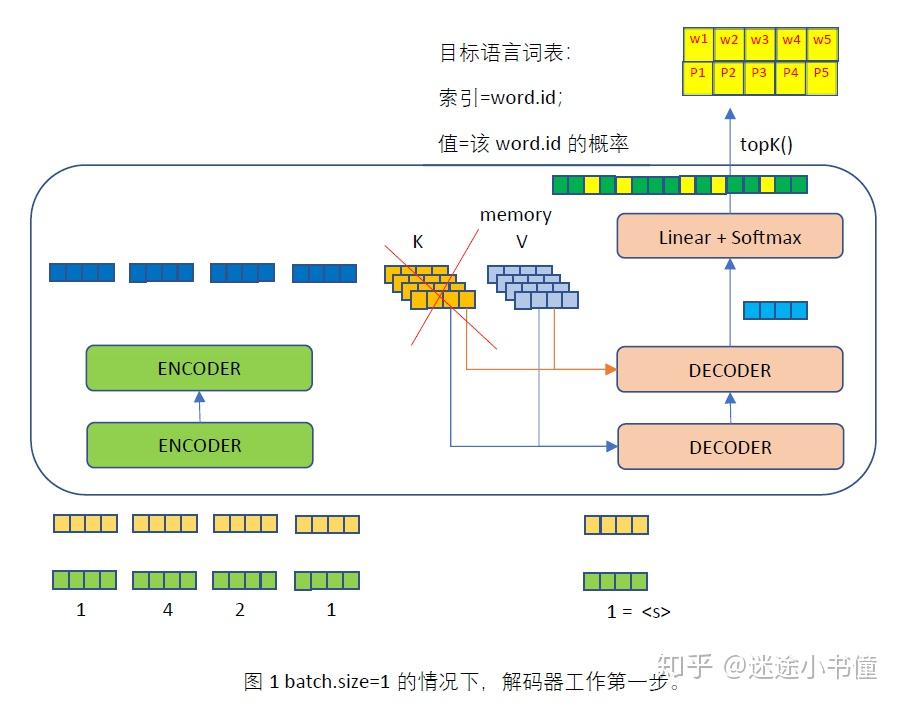
多年前，做statistical machine translation的时候，分别写过hierarchical phrase based decoder, tree-to-string decoder，string-to-tree decoder, forest-to-string decoder，以及dependency-to-dependency decoder，当时对于beam search+cube pruning搞了很多年。

因为涉及到实时计算n-gram language model来动态最优化BLEU score，所以在两个长度为n的beam合并的时候，如何把计算复杂度从n\*n降低，是个核心问题，因为一般的做法是，先n\*n，构造出来n\*n的候选，然后从中再挑取n个序列。Cube-pruning方法类似于merge sort，认定新的序列的第一个词，一般只来自前一个序列的第一个词+第二个序列的前两个词，或者反过来，这样的话，可以把n\*n的复杂度，降低到nlogn的维度。当年是Liang Huang和David Chiang提出来的这些解码思想。图示类似于：



两个序列的合并之后，第一个词，来自于三个位置(1,1), (1,2), (2,1)，然后继续扩展这个frontier set，直到到一个新的beam。（横向是一个sequence，纵向是另外一个sequence）

曾经为transformer中该如何执行beam search而思考思索，这里统一总结一下。



如图1 所示，走一遍DECODERs和Linear+softmax生成器，然后得到一个预测出来的logits向量（索引为word.id，值为该word.id的概率）。一次选择出来beam.size=5个words和它们对应的概率。如图中w1~w5和p1~p5所示。

有一点**特别需要注意**的是，来自Encoder的memory，是只有一个(batch.size, source.seq.len, d\_model)的tensor，**不会在Encoder阶段分成K和V给Decoder部分**！来自Encoder的memory tensor是经过Decoder里面每一层的MultiHeadAttention中的Linear network（可训练）来进一步转化成K和V的！

上面图中：假设当前是一个句子的输入，则memory的部分，是(1, 4, 4)，第一个1是batch.size，之后的4是序列的长度（单词的个数），最后一个4是d\_model，编码维度。

目标语言序列端，第一个输入是<s>，向量表示为(1, 1, 4)，也是分别代表了batch.size, sequence length和编码维度。

从多层Decoder出来之后，形状为(1, 1, 4)，而后经过Linear+softmax得到的是(1,1,target.vocab.size)，最后一个维度是目标语言词表大小。对于最后一个维度，索引是word.id，值就是该word.id的概率，从而我们可以取概率最大的前五个word.id，以及它们对应的概率值，来构造出来，W1~W5和P1~P5这些值供下一步解码使用。

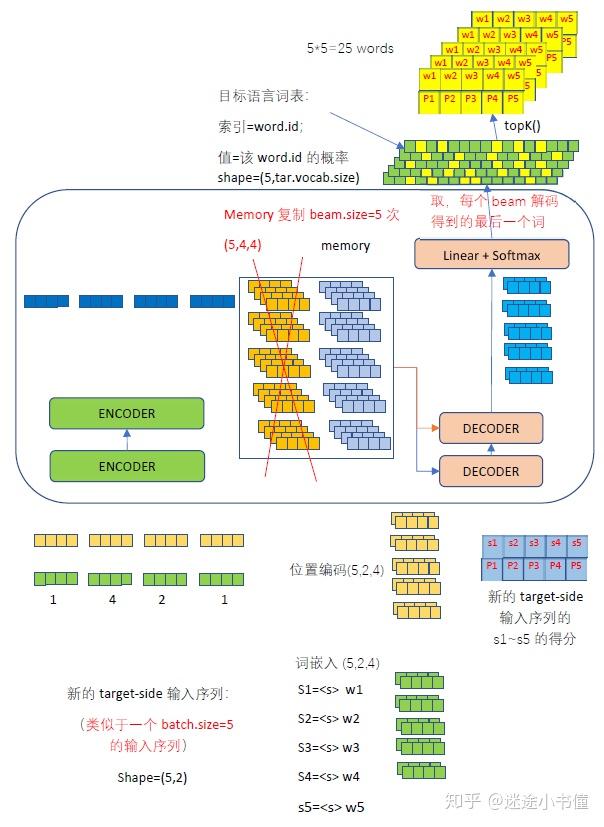


图2 batch.size=1, beam.size=5的时候的解码器工作第二步

如图2所示。根据beam size=5，构造出来一个batch.size=5的tensor，如上图所示。DECODERs和Linear+softmax生成器，然后得到一个预测出来的logits向量（索引为word.id，值为该word.id的概率）。一次选择出来beam.size=5个words和它们对应的概率。如图中右上角5\*5矩阵（黄色）所示。

特别需要注意的有：

1. 目前Decoder的输入的target-side序列，是(5,2)的一个矩阵，5代表beam.size，2代表序列长度；

2. 之后，先进过目标语言的词嵌入，得到一个(5,2,4)的tensor张量，再扔给位置编码，得到的也是一个(5,2,4)的张量。

3. 该(5,2,4)的张量（相当于Q）扔给Decoder之后，得到的是(5,2,4)的张量。这里特别需要注意的是，需要对来自Encoder的memory进行复制扩展。本来，Encoder的memory的shape是(1,4,4)，现在为了对照beam size=5，把(1,4,4)复制成(5,4,4)，相当于原来的一个4\*4矩阵，原样复制成了5个4\*4矩阵。这个memory张量(5,4,4)会在Decoder的MultiHeadAttention里面，进一步被两个Linear network（可训练的）变化成K和V。

4. （5,2,4）的张量扔给Linear+Softmax，之后得到的是(5,2,target.vocab.size)，最后一个维度扩展为目标语言词表的大小；

5. 取上面这个张量的每个beam的最后一个词，即(5, -1, target.vocab.size)，得到的是上图右上角的（绿+黄相间）的矩阵。然后按照最后一个维度，取top-5，就得到了右上角的最后的矩阵，是5\*5的，表示每个beam下扩展出来了5个最佳候选。

接下来，我们得到了5\*5=25个词的候选，五行代表五个beam，每一行五列，代表该beam下的下一个词的五个可能的候选。（简单期间，下图省略了第一个词之前的<s>）

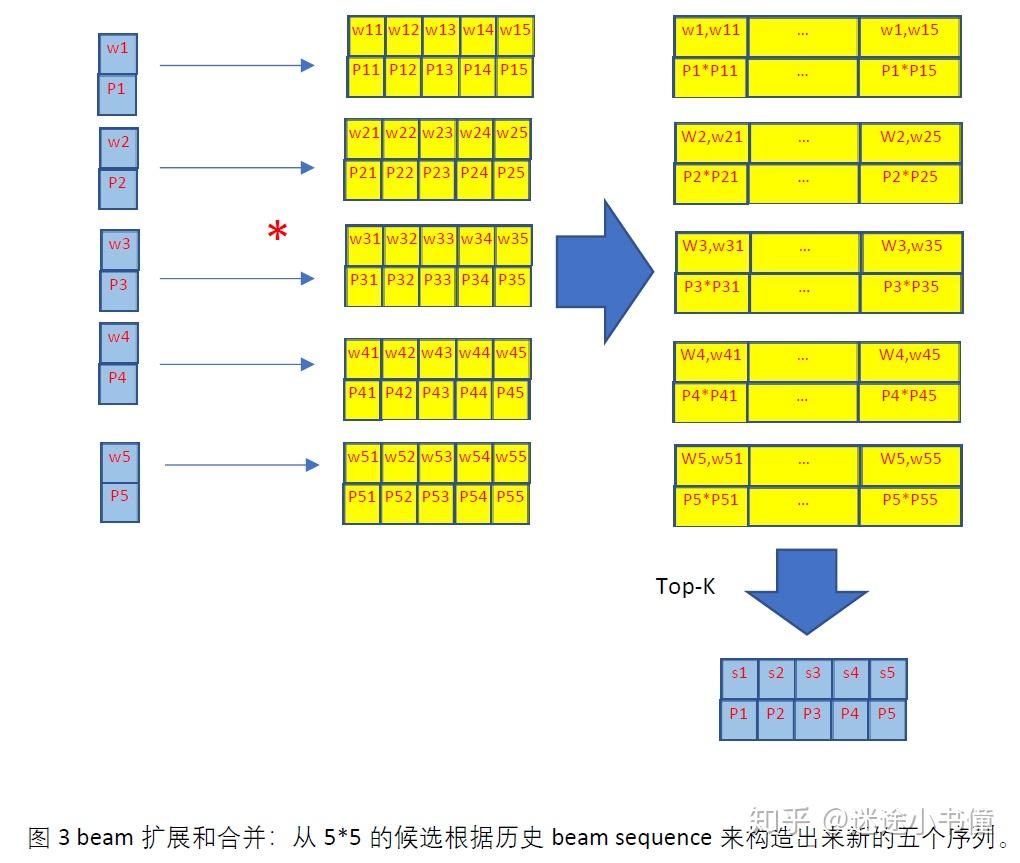


图3 beam扩展和合并：从5\*5的候选根据历史beam sequence来构造出来新的五个序列

图3 beam扩展和合并：从5\*5的候选根据历史beam sequence来构造出来新的五个序列。注意：因为矩阵相加可以并行进行，所以这里的速度会非常快，之前的那种cube-pruning方法不再是提速度的时候所必须的了。

这样，我们得到的是5\*5=25个候选序列。如果我们从中再次挑选概率最大的top-5个序列，以及他们的概率，可以进入下一轮的循环。如上图3所示，我们使用s1到s5来表示新得到的序列，每个si应该包括<s> <word in position 1> <word in position 2>，这样的三个词。实际解码的时候，一般使用log(p)，所以上面可以修改为log(p1) + log(p11)的形式。

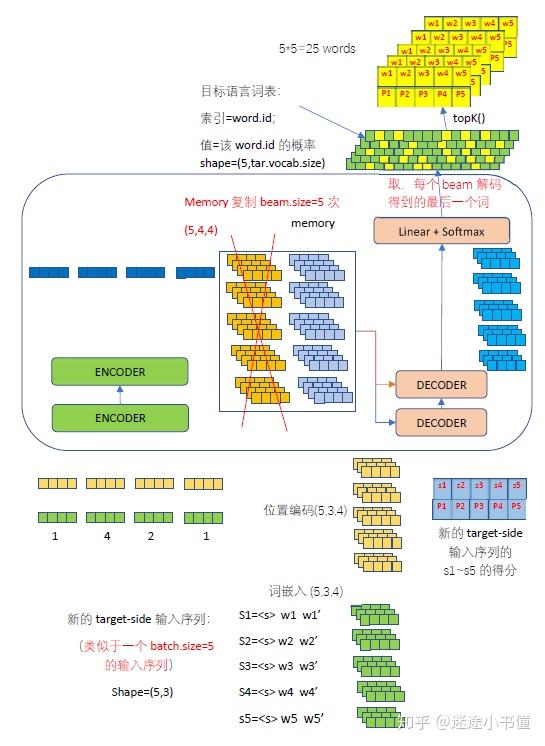


图4 batch.size=1, beam.size=5的前提下，解码第三个词的步骤展示

特别需要注意的有（和前面的类似，重复一遍，加深印象）：

1. 目前Decoder的输入的target-side序列，是(5,3)的一个矩阵，5代表beam.size，3代表序列长度；

2. 之后，先进过目标语言的词嵌入，得到一个(5,3,4)的tensor张量，再扔给位置编码，得到的也是一个(5,3,4)的张量。

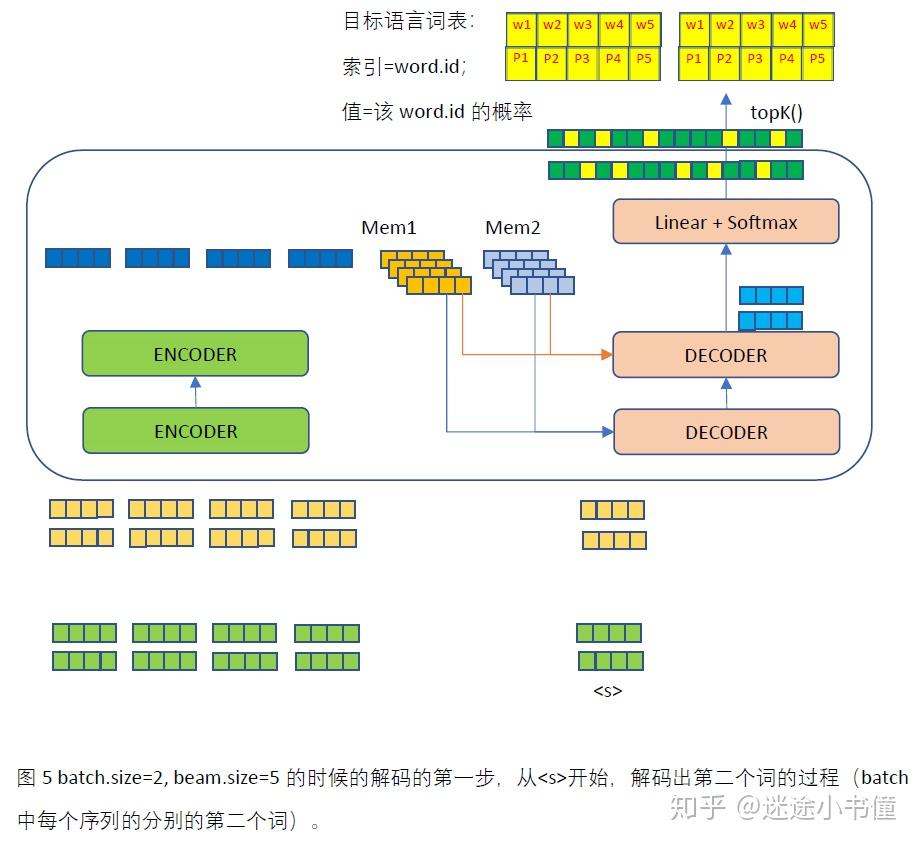
3. 该(5,3,4)的张量（相当于Q）扔给Decoder之后，得到的是(5,3,4)的张量。这里特别需要注意的是，需要对来自Encoder的memory进行复制扩展。本来，Encoder的memory的shape是(1,4,4)，现在为了对照beam size=5，把(1,4,4)复制成(5,4,4)，相当于原来的一个4\*4矩阵，原样复制成了5个4\*4矩阵。这个memory张量(5,4,4)会在Decoder的MultiHeadAttention里面，进一步被两个Linear network（可训练的）变化成K和V。

4. （5,3,4）的张量扔给Linear+Softmax，之后得到的是(5,3,target.vocab.size)，最后一个维度扩展为目标语言词表的大小；

5. 取上面这个张量的每个beam的最后一个词，即(5, -1, target.vocab.size)，得到的是上图右上角的（绿+黄相间）的矩阵。然后按照最后一个维度，取top-5，就得到了右上角的最后的矩阵，是5\*5的，表示每个beam下扩展出来了5个最佳候选。

之后，简单说一下beam search的终止条件。其一，设定一个最长目标语言长度，当到达这个长度之后，统一结束。其二，判断当前beam是否已经包含了</s>，如果是，则该beam被从当前beam tensor中剪切出来，可以不参与下一步的计算。为了惩罚特别短或者特别长的生成序列，可以设置一些长度惩罚方法，细节不再冗述。

另外，上面的例子是一次翻译一个句子，当然可以一次翻译一个batch的句子。算法流程大概如下：假设batch.size=2，（同样适用于batch.size更大的情况，这里简单期间，batch.size=2）。Encoder端输入是(2, 4,4)的张量，经过Encoder的多层编码之后，得到的是(2,4,4)的source memory。然后，目标语言序列一端，最初的输入是(2, 1)的矩阵，2=batch.size, 1个词是<s>。然后是词嵌入和位置编码，得到的是(2, 1, 4)张量。经历一次Decoder之后，得到的是(2, 1, 4)的张量，然后扔给Linear+softmax，得到的是(2, 1, target.vocab.size)，可以对最后一个维度取top-5，得到的是两个w1~w5和P1~P5的列表，分别对应batch中两个序列的下一个单词。



Batch.size=2的时候的，解码第一步，如图5所示。

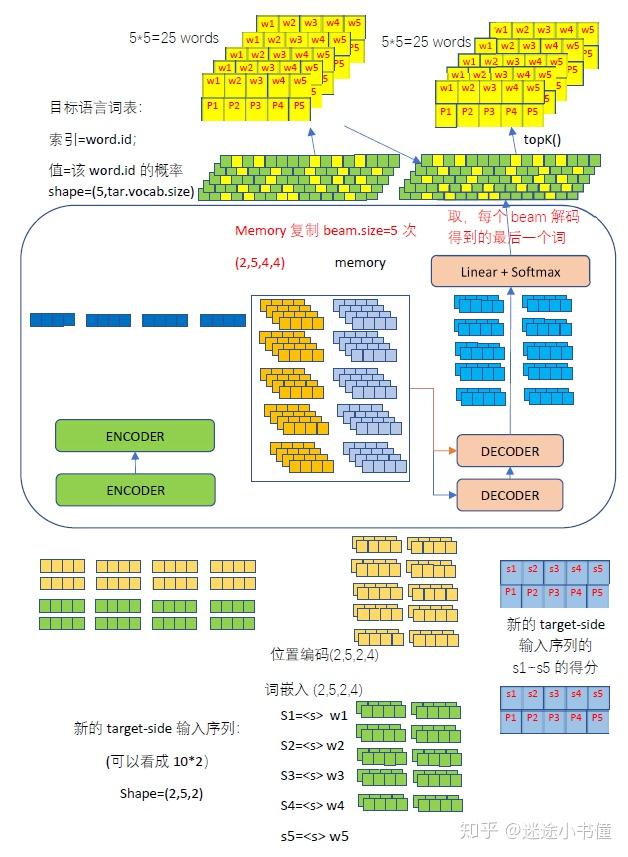


图6 batch.size=2, beam.size=5的时候，解码出第三个词的过程（每个batch的每个序列的，5个beam的第三个词）。

输入的memory，从(2,4,4)复制为(2, 5, 4, 4)，在实际使用的时候，reshape为(10, 4, 4)。

Decoder的输入部分，最初为(2,5,2)，batch为2，beam为5，每个beam为2个单词。经过词嵌入之后，得到的张量是(2,5,2,4)，然后在给Decoder之前，可以将其reshape为（10,2,4）

从而得到的Decoder的输出的张量是(10, 2, 4)的，再reshape回来，得到(2,5,2,4)，扔给Linear+softmax之后，得到的是(2, 5, 2, target.vocab.size)。

之后就是上图的最上边两行图示了，先取最后一个单词，也就是(2, 5, -1, target.vocab.size)，然后取最后一个维度的前top-5概率下的单词。会分别得到两个5\*5的w1~w5和p1~p5的列表。

各自分别和当前序列的s1~s5和p1~p5结合(logp + logp)的方式，就得到新的五个序列。持续按照这个步骤执行即可。

至此，Transformer的基本Encoder, Decoder, Generator, PositionEncoding, MultiHeadAttention等模块就介绍完毕了。训练，loss, optimization, 以及beam search解码也介绍完毕。

感谢！