写在前边：

学习 Transformer 的过程中，找到了**Ketan Doshi** 博客中关于Transformer系统的介绍文章，感觉非常棒，于是进行了翻译。**该系列一共有4篇文章。本篇文章为该系列的第一篇。**

原文链接在文末。翻译主要采用 “DeepL✦+人工”的方式进行，并加入了一些自己的理解。

第二篇：Transformer 之逐层介绍

第三篇：Transformer 之多头注意力

第四篇：Transformer 之注意力计算原理

一、引言

关于 Transformer 的研究已有许多。过去几年中，Transformer 及其变体在NLP✦的世界里掀起了巨大风暴。

Transformer是一种架构，它使用注意力来显著提高深度学习 NLP 翻译模型✦的性能，其首次在论文《Attention is all you need》✦中出现，并很快被确立为大多数文本数据应用的领先架构。

从那时起，包括谷歌的BERT、OpenAI的GPT系列在内的众多项目都建立在这个基础上，并发表了轻松击败现有最新基准的性能结果。

本文将具体介绍 Transformer 的基本知识，架构，及其内部工作方式。我们将以自上而下的方式介绍其功能。同时，本文将深入剖析 Transformer 内部的细节，以了解其系统的运作。我们还将深入探讨多头注意力的工作原理。

以下是文章总览。我在整个过程中的目标将是，不仅要了解某件事情是如何运作的，还要了解它为什么会这样运作。

1. **Overview of functionality**：Transformer 是如何使用的，以及为什么它们比RNN✦更好。架构的组成部分，以及训练和推理期间的行为。
2. **How it works:** 端到端的内部操作。包括数据如何流动，进行哪些计算，矩阵表示等。
3. **Why Attention Boosts Performance**：不仅仅是Attention的作用，而且是为什么它的效果这么好。注意力是如何捕捉句子中单词之间的关系的

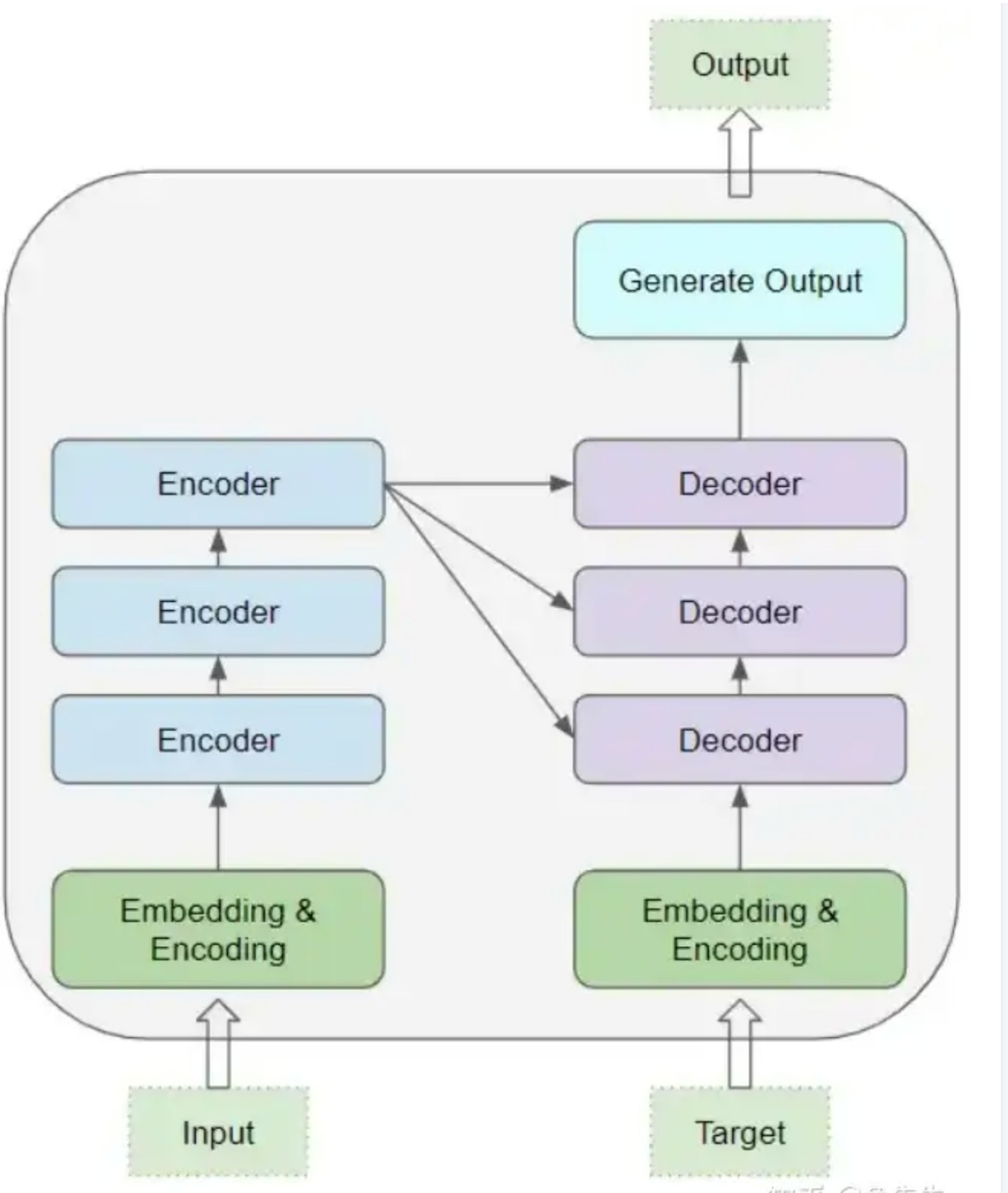
二、何为 Transformer?

Transformer 架构擅长处理文本数据，这些数据本身是有顺序的。它们将一个文本序列作为输入，并产生另一个文本序列作为输出。例如，将一个输入的英语句子翻译成西班牙语。

(Image by Author)

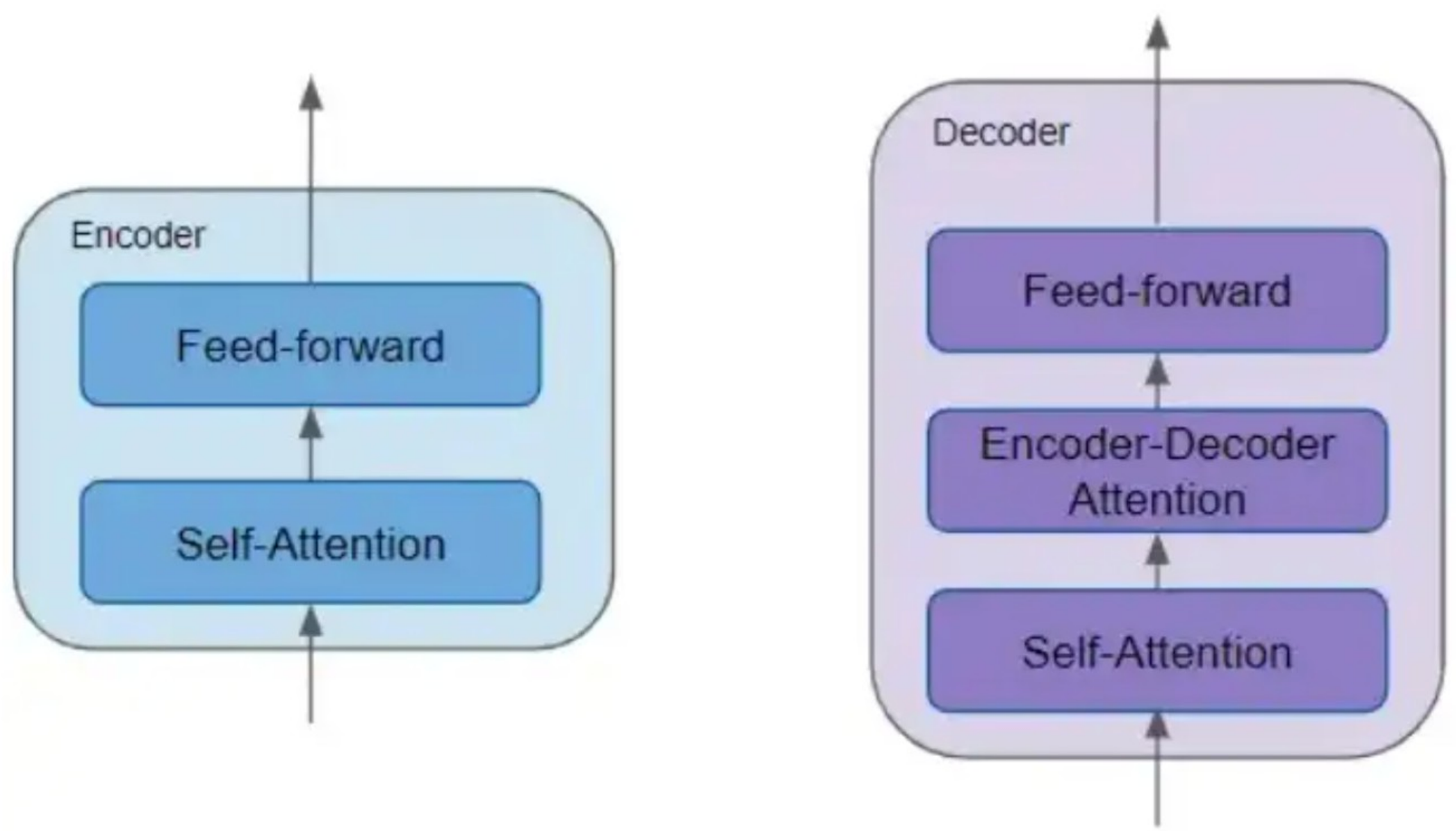
Transformer 的核心部分，包含一个编码器层和解码器层的堆栈。为了避免混淆，我们将把**单个层称为编码器或解码器，并使用编码器堆栈或解码器堆栈分别表示一组编码器与一组解码器。**（原文为 Encoder stack 和 Decoder stack✦）。

在编码器堆栈和解码器堆栈之前，都有对应的嵌入层✦。而在解码器堆栈后，有一个输出层来生成最终的输出。



(Image by Author)

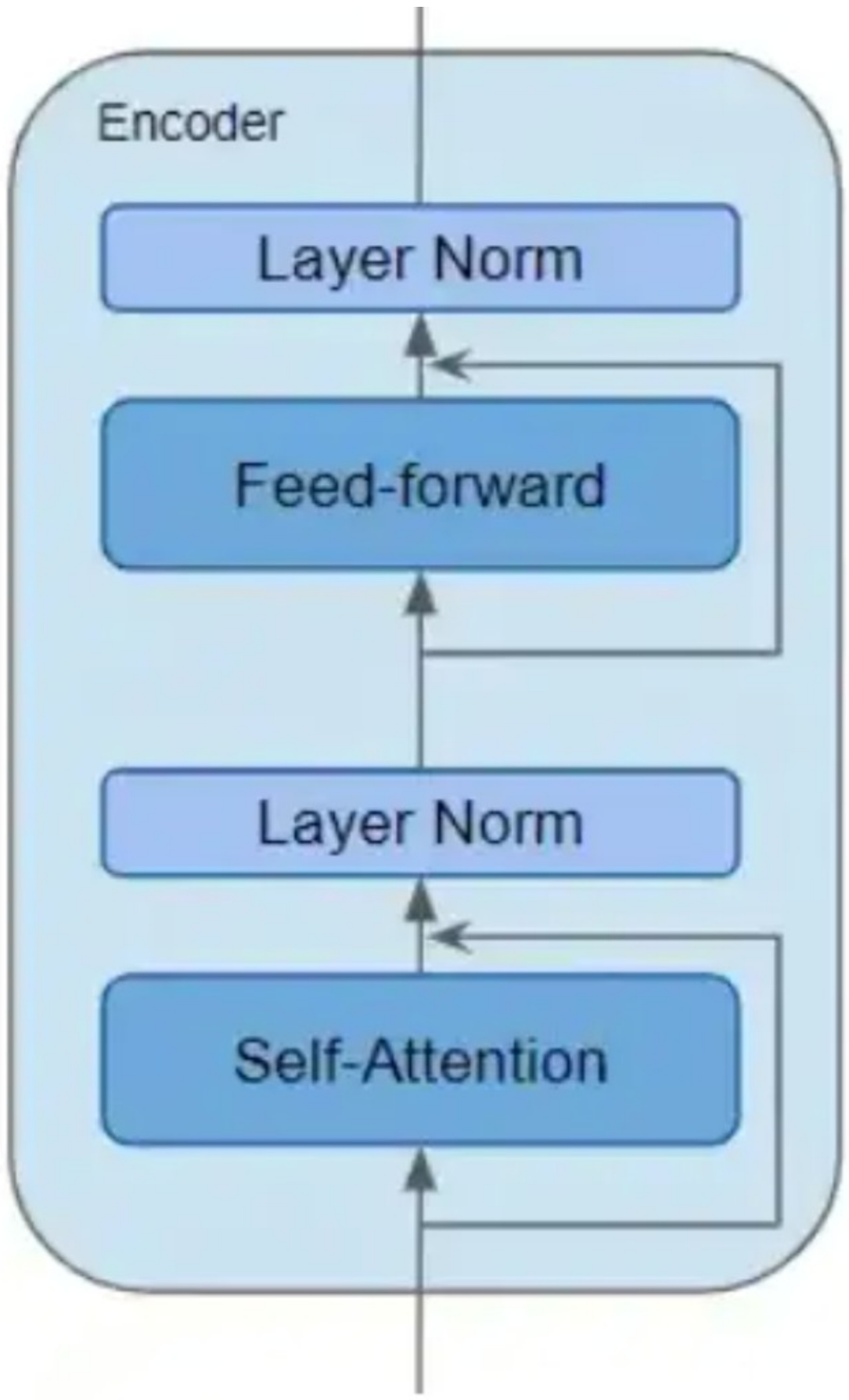
编码器堆栈中的每个编码器的结构相同。解码器亦然。其各自结构如下：



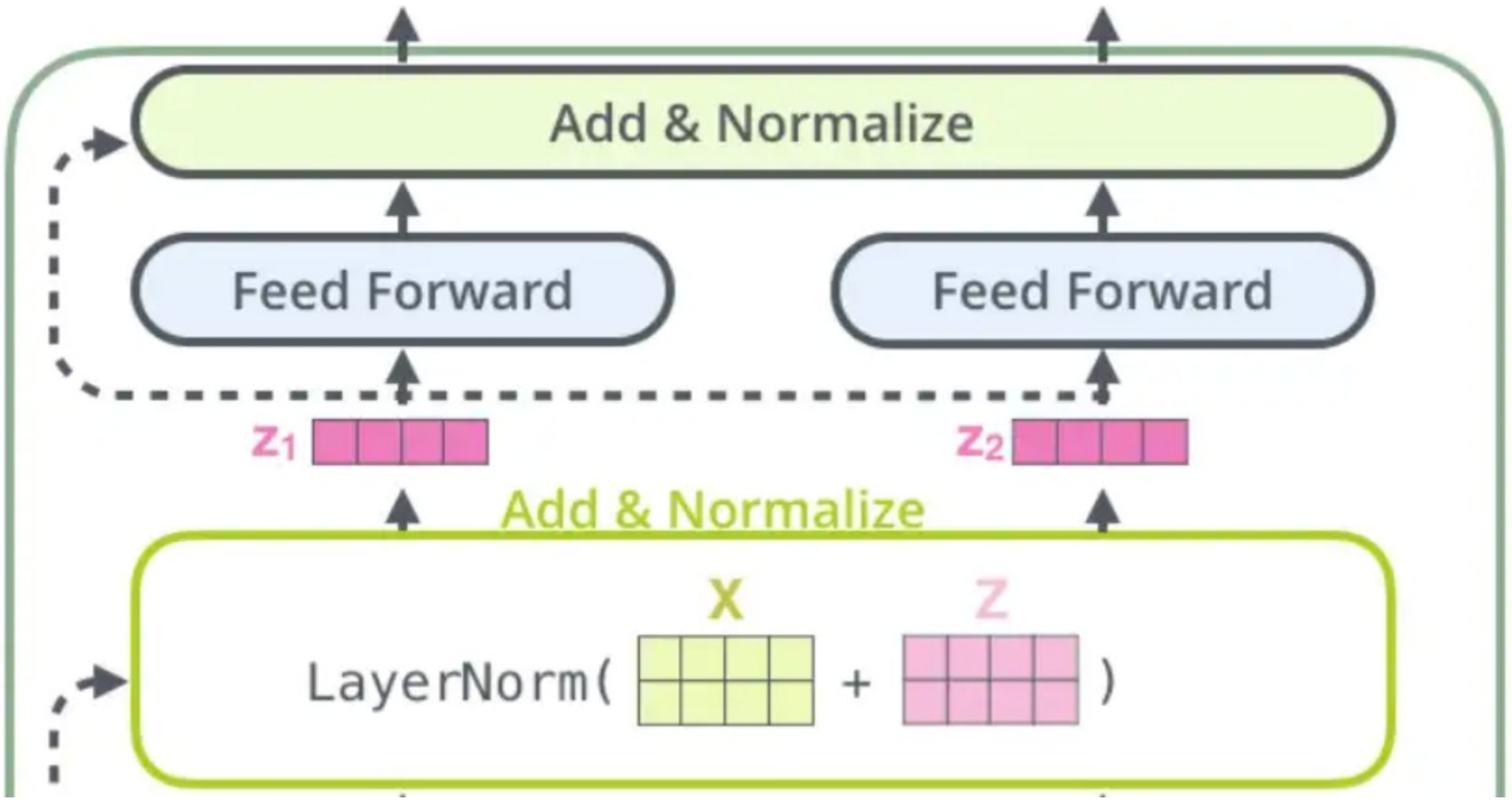
(Image by Author)

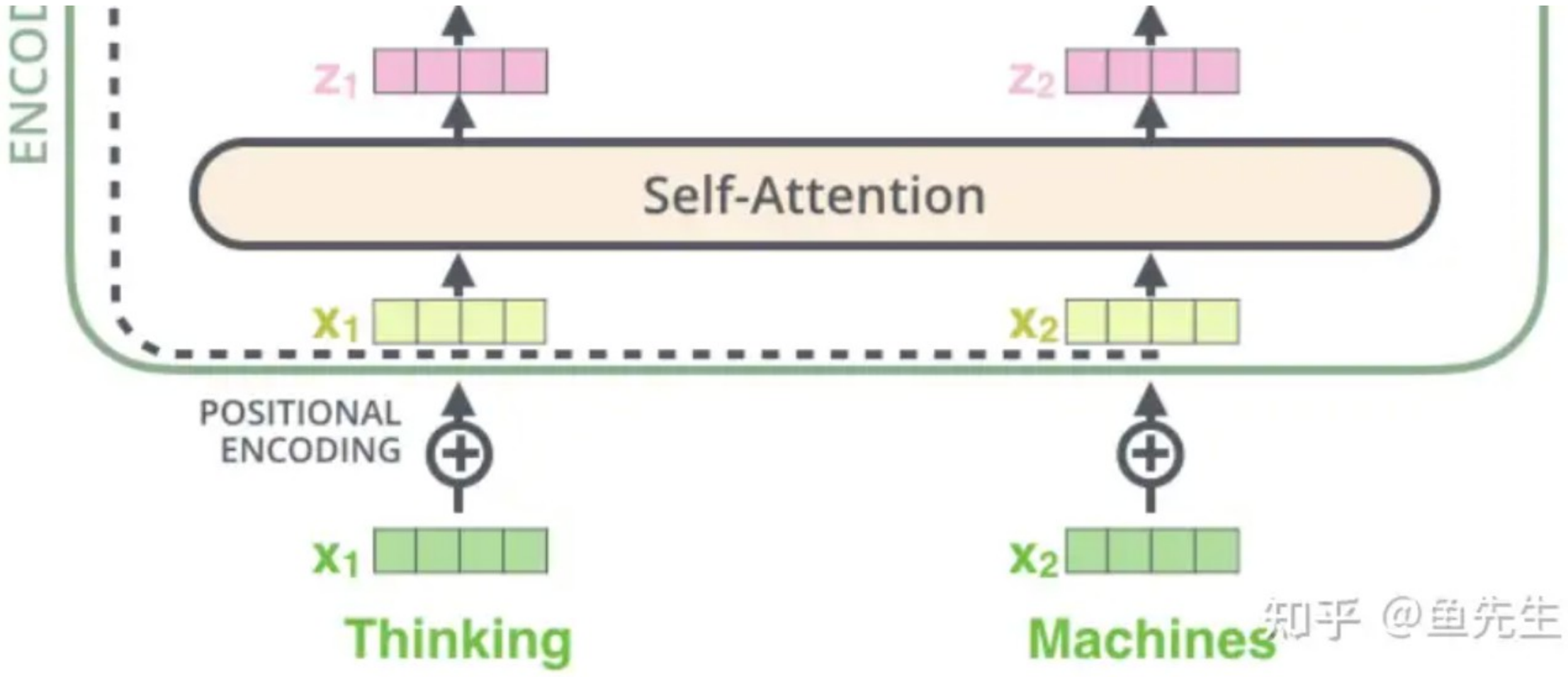
1. **编码器一般有两个子层**：包含自注意力层 self-attention，用于计算序列中不同词之间的关系；同时包含一个前馈层 feed-forward✦。
2. **解码器一般有三个子层**：包含自注意力层self-attention✦，前馈层 feed-forward，编码器-解码器注意力层 Decoder-Encoder self attention✦。
3. **每个编码器和解码器都有独属于本层的一组权重。**

需要注意的是，编码器的自注意力层及前馈层均有残差连接✦以及正则化层。



(Image by Author)





基于 Transformer 的变体有许多。一些 Transformer 架构甚至没有 Decoder 结构，而仅仅依赖 Encoder。

三、Attention 在做什么?

Transformer 的突破性表现关键在于其对注意力的使用。

在处理一个单词时，注意力使模型能够关注输入中与该单词密切相关的其他单词。

例如。ball 与 blue 、 hold 密切相关。另一方面，boy 与 blue 没有关系。

Transformer 通过将输入序列中的**每个词与其他词**关联起来（同一序列中），形成 self-attention 机制✦。

考虑以下两个句子：

- The *cat* drank the milk because **it** was hungry.
- The cat drank the *milk* because **it** was sweet.

第一个句子中，单词 'it' 指'cat'；第二个句子中，'it' 指 'milk'。当模型处理 'it'这个词时，self-attention 给了模型更多关于 'it' 意义的信息，这样就能把 'it '与正确的词联系起来。

Dark colors represent higher attention (Image by Author)

为了使模型能够处理有关于句子意图和语义的更多细微差别，Transformer 对每个单词都进行注意力打分。

在处理 "it "这个词时，第一个分数突出 "cat"，而第二个分数突出 "hungry"。因此，当模型解码✦ 'it'这个词时，即把它翻译成另一种语言的单词时，将会把 'cat' 和 'hungry' 某些语义方面的性质纳入到目标语言中。

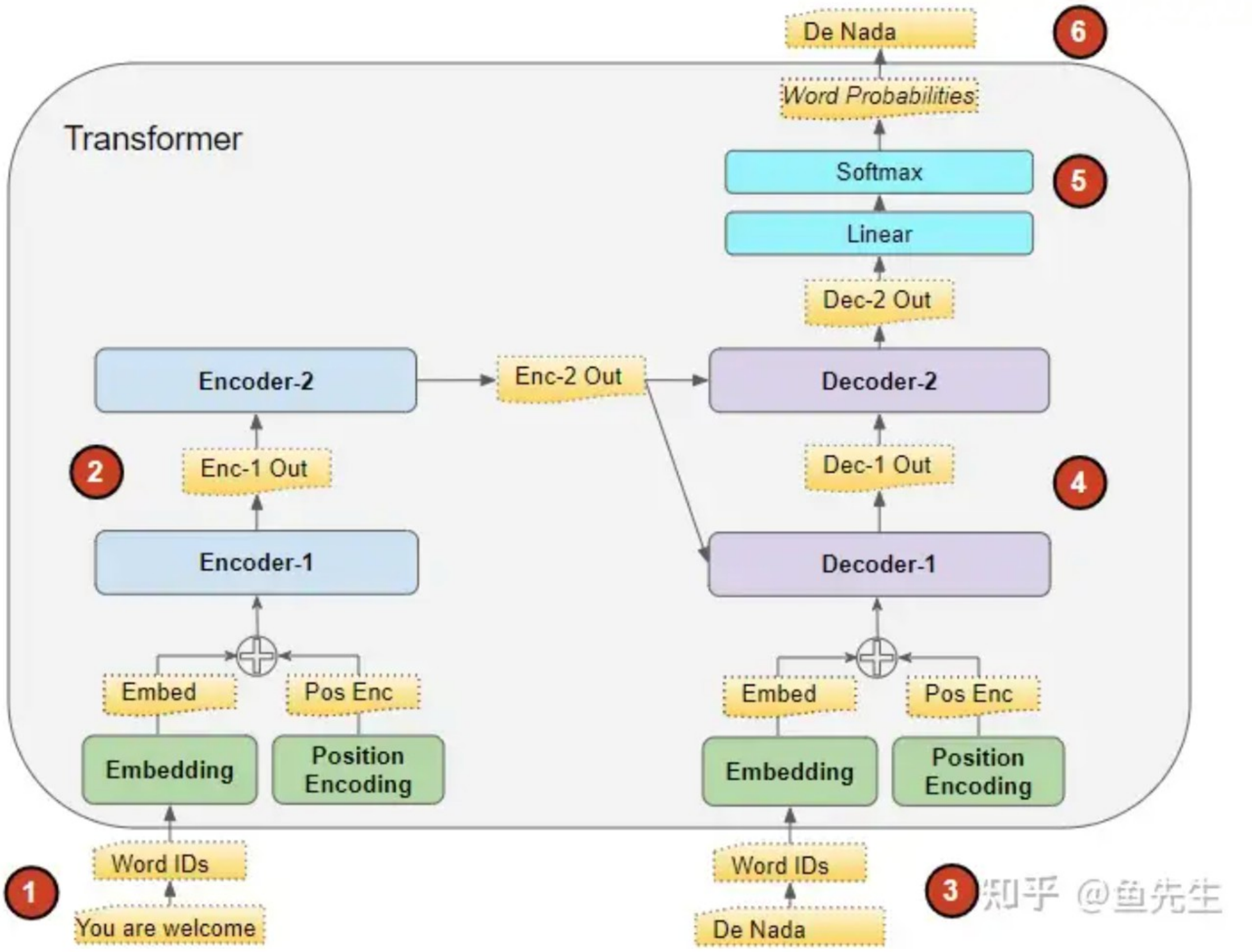
四、Transformer 训练过程

Transformer 的训练和推理有一些细微差别。

首先来看训练。训练数据包括两部分内容：

1. The source or input sequence (eg. “You are welcome” in English, for a translation problem)
2. The destination or target sequence (eg. “De nada” in Spanish)

Transformer 训练的目标是通过对源序列与目标序列的学习，生成目标序列。



训练过程中，模型对数据的处理过程如下，大体可分为 6 个步骤：

1. 在送入第一个编码器之前，输入序列 (src_seq) 首先被转换为嵌入（带有位置编码），产生词嵌入✦表示(src_position_embed)，之后送入第一个编码器。
2. 由各编码器组成的编码器堆栈按照顺序对第一步中的输出进行处理，产生输入序列的编码表示(enc_outputs)。
3. 在右侧的解码器堆栈中，目标序列首先加一个句首标记，被转换成嵌入（带位置编码），产生词嵌入表示(tgt_position_embed)，之后送入第一个解码器。
4. 由各解码器组成的解码器堆栈，将第三步的词嵌入表示(tgt_position_embed)，与编码器堆栈的编码表示(enc_outputs)一起处理，产生目标序列的解码表示(dec_outputs)。
5. 输出层将其转换为词概率和最终的输出序列(out_seq)。
6. 损失函数将这个输出序列(out_seq)与训练数据中的目标序列(tgt_seq)进行比较。这个损失被用来产生梯度✦，在反向传播过程中训练模型。

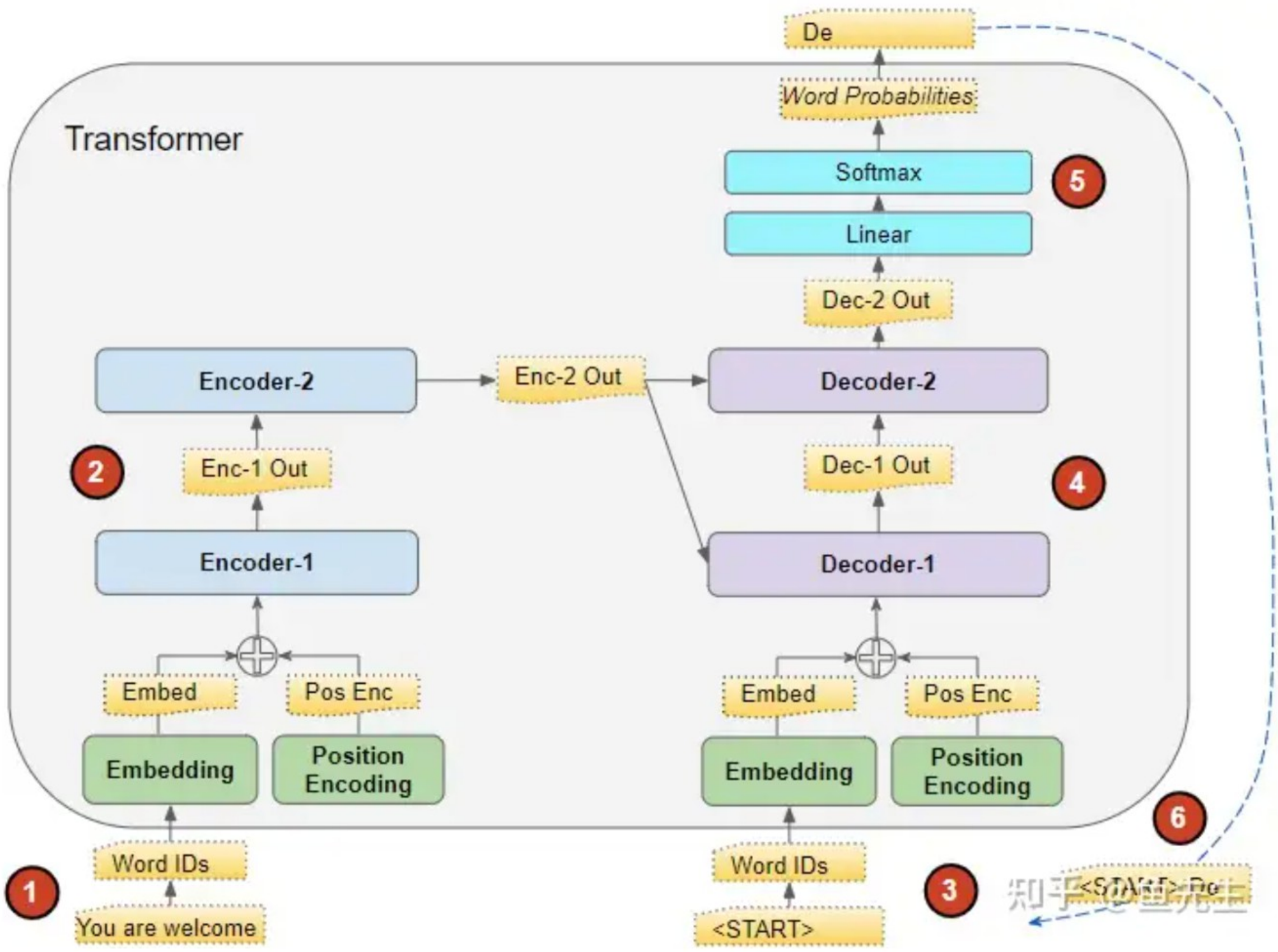
五、Transformer 推理过程

在推理过程中，我们只有输入序列，而没有目标序列作为输入传递给解码器。Transformer 推理的目标是仅通过输入序列产生目标序列。

因此，与 Seq2Seq✦ 模型类似，我们在一个时间步的完整循环中生成当前时间步的输出，并在下一个时间段将前一个时间段的输出序列传给解码器作为其输入，直到我们遇到句末标记。

但与 Seq2Seq 模型的不同之处在于，在每个时间步，我们输入直到当前时间步所产生的整个输出序列，而不是只输入上一个时间步产生的词。

非常重要，把原文粘过来：The difference from the Seq2Seq model is that, at each timestep, we re-feed the entire output sequence generated thus far, rather than just the last word.



推理过程中的数据流转如下：

1. 第一步与训练过程相同：输入序列 (src_seq) 首先被转换为嵌入（带有位置编码），产生词嵌入表示(src_position_embed)，之后送入第一个编码器。
2. 第二步也与训练过程相同：由各编码器组成的编码器堆栈按照顺序对第一步中的输出进行处理，产生输入序列的编码表示(enc_outputs)。
3. 从第三步开始一切变得不一样了：在第一个时间步，使用一个只有句首符号的空序列来代替训练过程中使用的目标序列。空序列转换为嵌入带有位置编码的嵌入(start_position_embed)，并被送入解码器。
4. 由各解码器组成的解码器堆栈，将第三步的空序列嵌入表示(start_position_embed)，与编码器堆栈的编码表示(enc_outputs)一起处理，产生目标序列第一个词的编码表示(step1_dec_outputs)。
5. 输出层将其(step1_dec_outputs)转换为词概率和第一个目标单词(step1_tgt_seq)。
6. 将这一步产生的目标单词填入解码器输入的序列中的第二个时间步位置。在第二个时间步，解码器输入序列包含句首符号产生的 token 和第一个时间步产生的目标单词。
7. 回到第3个步骤，与之前一样，将新的解码器序列输入模型。然后取输出的第二个词并将其附加到解码器序列中。重复这个步骤，直到它预测出一个句末标记。需要明确的是，由于编码器序列在每次迭代中都不会改变，我们不必每次都重复第1和第2步。

六、Teacher Forcing

训练时向解码器输入整个目标序列的方法被称为 Teacher Forcing。

训练时，我们本可以使用与推理时相同的方法。即在一个时间步运行 Transformer，从输出序列中取出最后一个词，将其附加到解码器的输入中，并将其送入解码器进行下一次迭代。最后，当预测到句末标记时，Loss 函数✦将比较生成的输出序列和目标序列，以训练网络。

但这种训练机制不仅会导致训练时间更长，而且还会增加模型训练难度：若模型预测的第一个词错误，则会根据第一个错误的预测词来预测第二个词，以此类推。

相反，通过向解码器提供目标序列，实际上是给了一个提示。即使第一个词预测错误，在下一时间步，它也可以用正确的第一个词来预测第二个词，避免了错误的持续累加。

此外，这种机制保证了 Transformer 在训练阶段并行地输出所有的词，而不需要循环，这大大加快了训练速度。

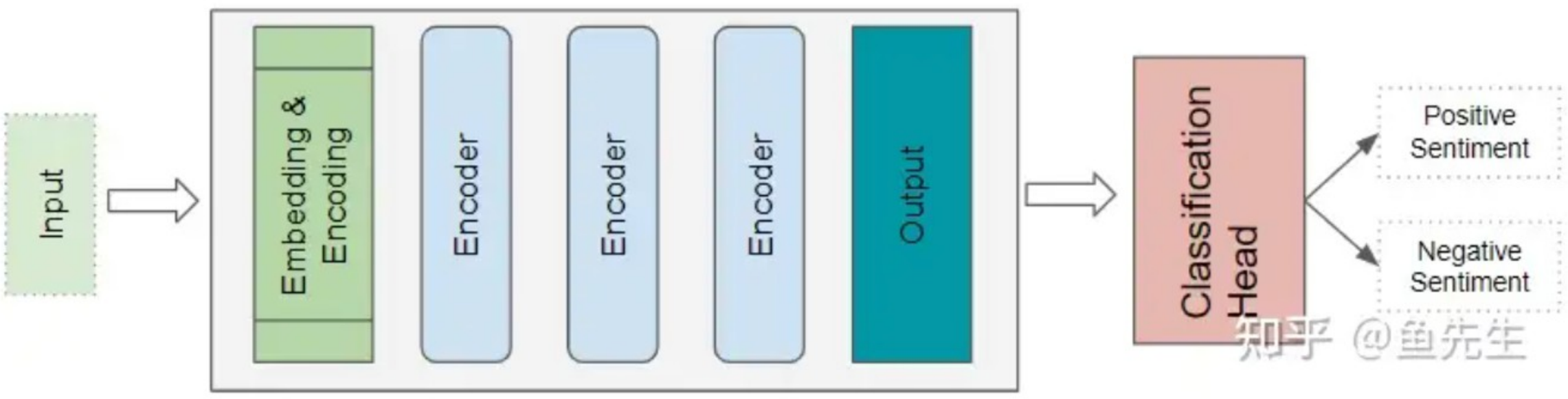
七、 Transformer 应用场景

Transformer 的用途非常广泛，可用于大多数NLP任务，如语言模型和文本分类✦。它们经常被用于 Seq2Seq 的模型，如机器翻译、文本总结、问题回答、命名实体识别✦和语音识别等应用。

对于不同的问题，有不同的 Transformer 架构。基本的编码器层被用作这些架构的通用构件，根据所解决的问题，有不同的特定应用 "头"。

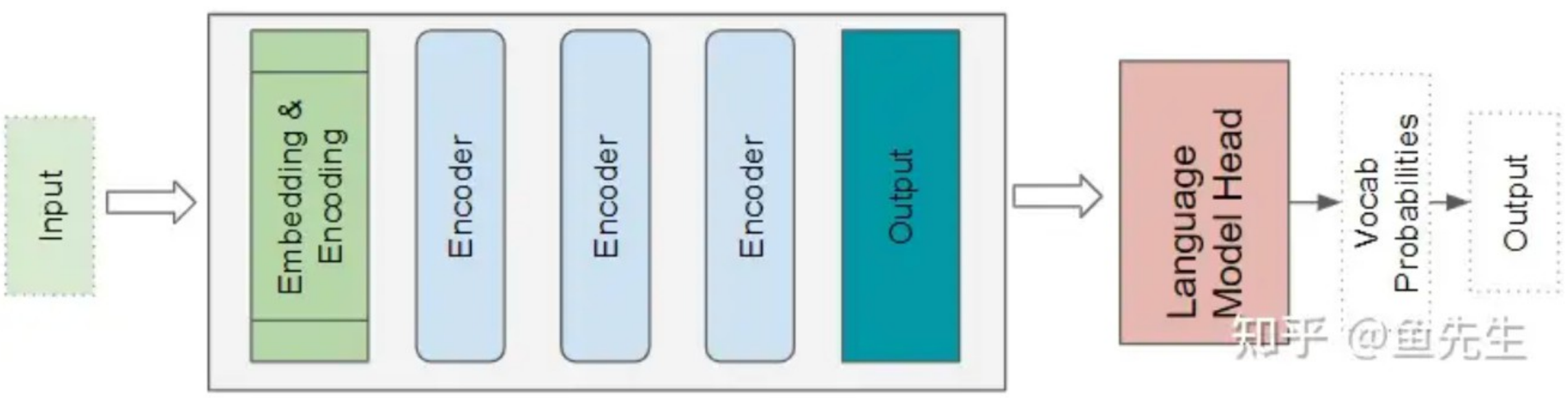
1. Transformer 分类器架构

如下所示，一个情感分析程序，把一个文本文件作为输入。一个分类头接收Transformer 的输出，并生成预测的类别标签，如正面或负面情绪。



2. Transformer Language Model architecture

Language Model architecture 架构将把输入序列的初始部分，如一个文本句子作为输入，并通过预测后面的句子来生成新的文本。一个 Language Model architecture 头接受 Transformer 的输出作为 head 的输入，产生关于词表中每个词的概率输出。概率最高的词成为句子中下一个词的预测输出。



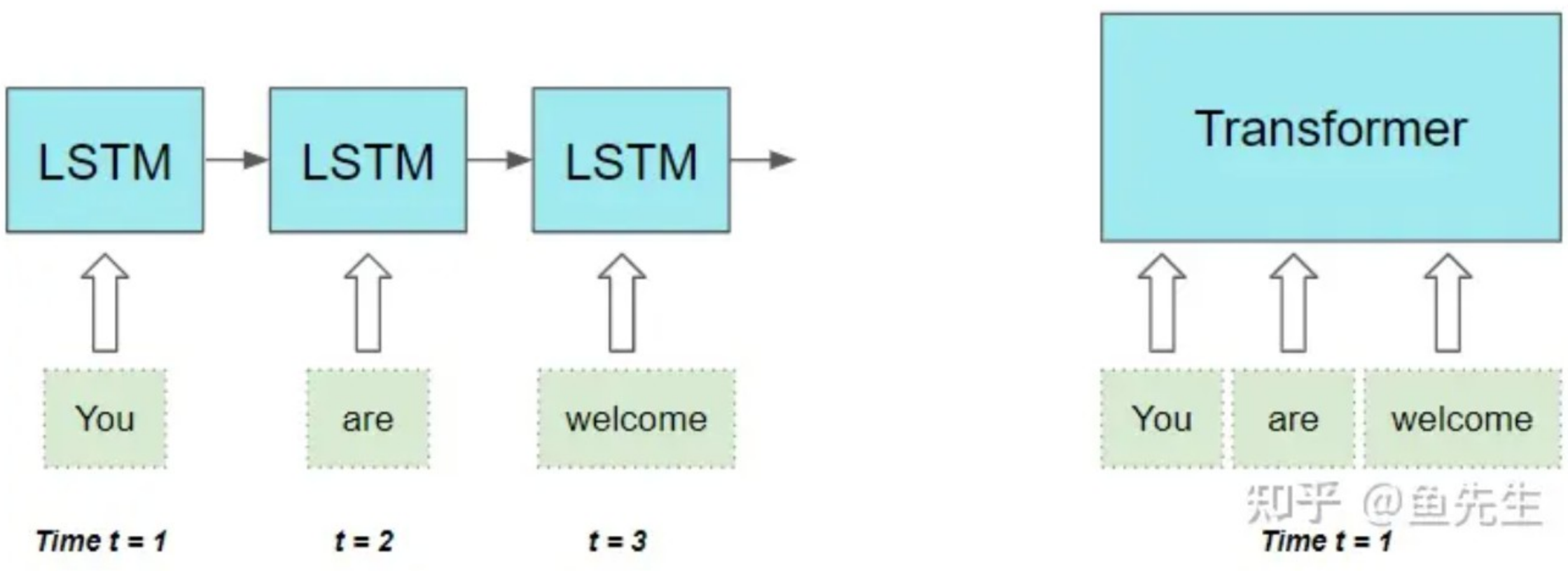
八、与 RNN 类型的架构相比，为什么 Transformer 的效果要好?

RNNs 和 LSTMs✦、GRU也是之前 NLP 常用的架构，直到 Transformer 的出现。
然而，这有两个限制：

1. 对于长句中相距较远的单词，其间的长距离依赖关系是一个挑战。
2. RNNs 每个时间步值处理输入序列的一个词。这意味着在完成时间步 T-1 计算之前，它无法进行时间步骤 T 的计算。（即无法进行并行计算）这降低了训练和推理速度。

对于CNN✦来说，所有的输出都可以并行计算，这使得卷积✦速度大大加快。然而，它们在处理长距离的依赖关系方面也有限制：
卷积层✦中，只有图像（或文字，如果应用于文本数据）中足够接近于核大小的部分可以相互作用。对于相距较远的项目，你需要一个有许多层的更深的网络。
Transformer 架构解决了这两个限制。它摆脱了RNNs，完全依靠 Attention的优势：

1. 并行地处理序列中的所有单词，从而大大加快了计算速度。（只是在训练阶段与推理阶段的 Encoder 中吧。推理过程的 Decoder 也不是并行的）。



2. 输入序列中单词之间的距离并不重要。Transformer 同样擅长计算相邻词和相距较远的词之间的依赖关系。