

transformer 答疑

1. 如何理解 transformer 的位置向量?

$$\begin{cases} PE(pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right) \\ PE(pos, 2i+1) = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right) \end{cases}$$

pos 代表一个句子中的一个词。

如：我 想 谈恋爱

PE(0,2i) 代表位置为 0 的词我这个词的词向量的第 2i 个位置

PE(0,2i+1) 代表位置为 0 的词我这个词的词向量的第 2i+1 个位置

位置向量的目的是：希望模型学习到词与词之间的相对位置。

为啥这个公式可以表示词与词之间的位置信息呢？

$$\begin{cases} \sin(\alpha + \beta) = \sin\alpha\cos\beta + \cos\alpha\sin\beta \\ \cos(\alpha + \beta) = \cos\alpha\cos\beta - \sin\alpha\sin\beta \end{cases}$$

$$\begin{cases} PE(pos + k, 2i) = PE(pos, 2i) \times PE(k, 2i + 1) + PE(pos, 2i + 1) \times PE(k, 2i) \\ PE(pos + k, 2i + 1) = PE(pos, 2i + 1) \times PE(k, 2i + 1) - PE(pos, 2i) \times PE(k, 2i) \end{cases}$$

这个公式的含义是：

句子中的第 pos+k 个位置，某个词之后的第 k 个位置的词，2i 是这个词对应的词向量的第 2i 个值，可以表示为第 pos 个位置的词向量的线性表示。

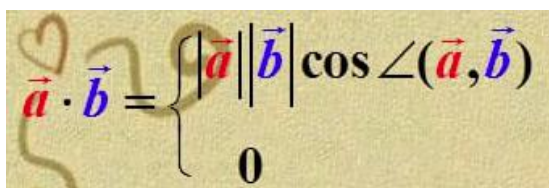
但是这个线性表示引入的东西比较多，包括第 k 个位置词向量的 2i+1 个值和 2i 个值，还有第 pos 个位置词的 2i 和 2i+1 的位置。

这样做一个线性表示就能表示词与词之间的相对位置吗？

$$\begin{aligned}
PE_t^T PE_{t+k} &= \sum_{j=0}^{\frac{d}{2}-1} [\sin(c_j t) \sin(c_j(t+k)) \\
&\quad + \cos(c_j t) \cos(c_j(t+k))] \\
&= \sum_{j=0}^{\frac{d}{2}-1} \cos(c_j(t - (t+k))) \\
&= \sum_{j=0}^{\frac{d}{2}-1} \cos(c_j k)
\end{aligned}$$

怎样计算两个向量之间的距离：

因为每个位置向量的模都是一样的。跟词向量维度有关，如果维度为 512，那么模的大小为 $256^{0.5}$



$$\vec{a} \cdot \vec{b} = \begin{cases} |\vec{a}| |\vec{b}| \cos \angle(\vec{a}, \vec{b}) \\ 0 \end{cases}$$

衡量距离可以用向量和向量之间的角度来衡量。因此点积就可以衡量向量之间的距离。

可以看出第 k 个位置词和第 $t+k$ 个位置之间的距离是 k 的函数。因此那种表示是可以衡量相对位置的。

但是，思考另外一个问题：

$$PE_{pos}^T PE_{pos-k} = PE_{pos}^T PE_{pos+k}$$

这个公式意味着词与词之间的距离一致时便无法分清先后顺序。

例如：我 想 谈恋爱。针对想这个词，我 和 谈恋爱这两个词是无法定位其先后顺序的。

同时，位置向量需要通过长途跋涉才会到达 decoder。

在第一个 encoder 中就出现了

$$PE_{pos}^T W_q^T W_k PE_{pos+k}$$

这个意味着中间加了个矩阵作为乘法，虽然说最终的结果是一个标量，但是 w 是一个可以学习得到的参数，是具有可变性的，

可变的東西意味着 $PE_{pos}^T W_q^T W_k PE_{pos+k}$ 不再是一个仅仅是关于 k 的一个自变量的函数，而是更加复杂，更加没有规律性。

这可能就是 transformer 中位置信息做的不够细致的一个原因吧。后面会有很多人进行改进。

Batch normalization

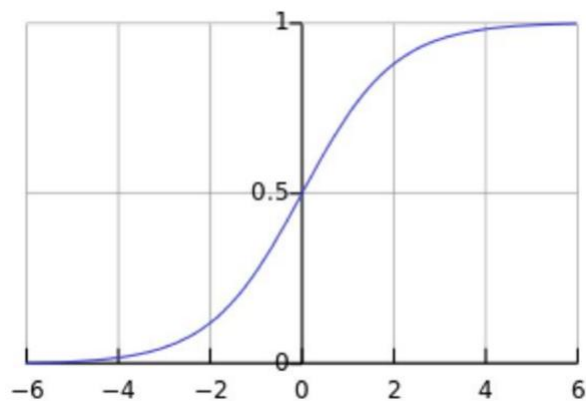
1. 理解 layer norm 之前需要先理解 BN

iid 是统计学习中非常重要的假设，如果通过一批数据训练好了一个模型，上线了之后 n 天之后发现效果越来越差了，可能是因为数据的分布发生了变化。bn 就是用到深度学习中的保持 iid 的一个东西。

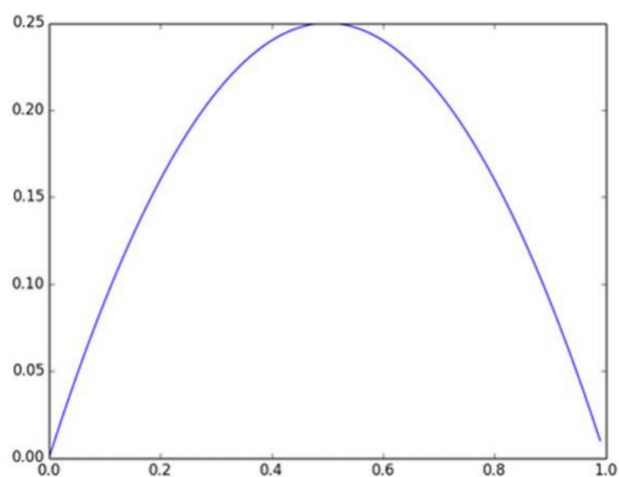
2. 假如说深度学习中激活函数都用 sigmoid

如下图可以看到当 x 属于 -2 到 2 之间的时候几乎非线性激活函数激活效果如同线性函数，意味着这一块区域的导数最大，激活函数看的是激活函数的导数，但是越接近两边导

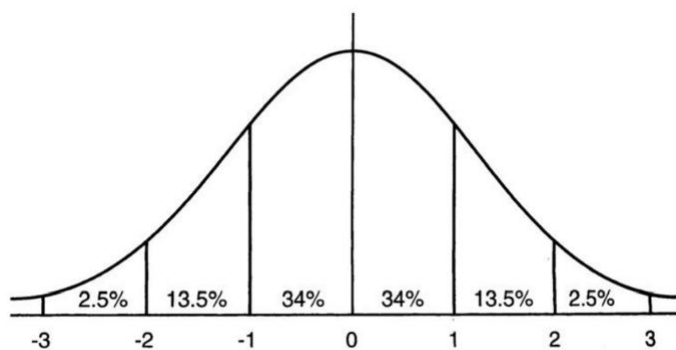
数越接近 0。在求深度学习的参数的时候，需要对 w 进行迭代，迭代的时候我们希望导数越大越好，越大收敛越快。



sigmoid 函数的导数:



标准正态分布。



正态分布是一个特别普遍分布，根据中心极限定理可知：

任意 k 个变量的均值组成的符合变量的分布都服从正态分布。

所以可以假设训练样本服从正态分布，还有深度学习中每一层隐函数往后面激活的时候，激活前的那些每个维度的变量也服从正态分布。

将正态分布和 **sigmoid** 函数联合起来看：

当 x 属于标准正态分布时， x 在 -2 和 2 之间的比例达到

$27\%+68\%=95\%$ 。

因此如果我们令输入深度学习中，进行激活函数之前的值都尽量保证在 -2 到 2 之间，那么损失函数的收敛速度是非常快的。

如何做呢？

深度学习训练中最常用的就是 **minibatch** **sgd**，在 **minibatch** 中强制将每个维度的数据都转化为标准正态分布。

$$\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

然鹅，即使全部转化为标准正态分布，还是有问题，并不能保证每个数据都能落在 -2 到 2 之间。这个时候可以再做一个过分的事情，对 **minibatch** 中的样本再进行加乘，因为这样可以将 x 拉到线性变化区，有点玄幻的感觉，但是这就是深度学习的精髓啊。其中 r 和 b 都是待学习参数。咋学习的？暂时没研究。可见加了 **bn** 之后模型收敛速度可是非常的快。

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1, \dots, x_m\}$;

Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

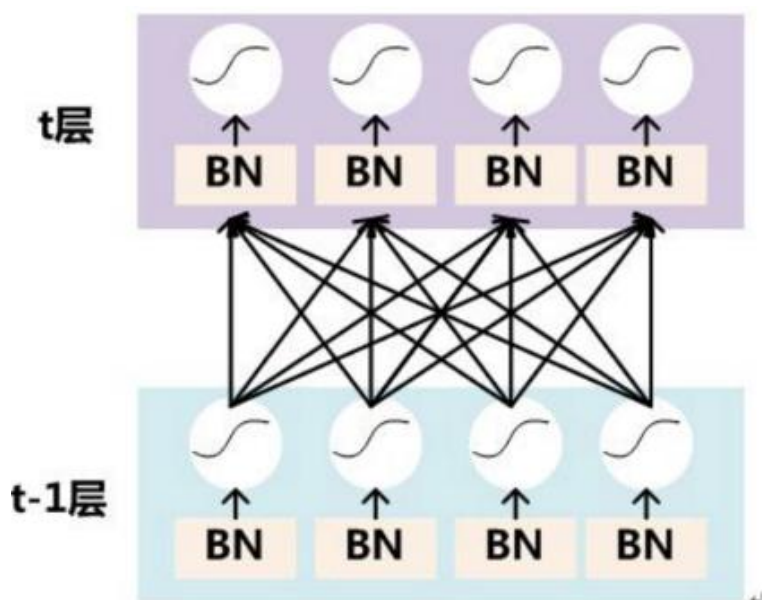
$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{ mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{ mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{ normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{ scale and shift}$$

bn 操作的位置是激活之前



模型训练好了，之后需要在线预测

在线预测需要几个每个隐含层激活前的那些每个维度的均值方差信息，训练的时候保存下来，预测的时候利用他们求全局的均值方差即可。同时 **scale** 参数和 **shift** 参数也是训练好的，直接拿来用即可。

Layer norm

1. DNN 中可以使用 BN, RNN 没有 minibatch 的概念, 怎样 BN 呢?

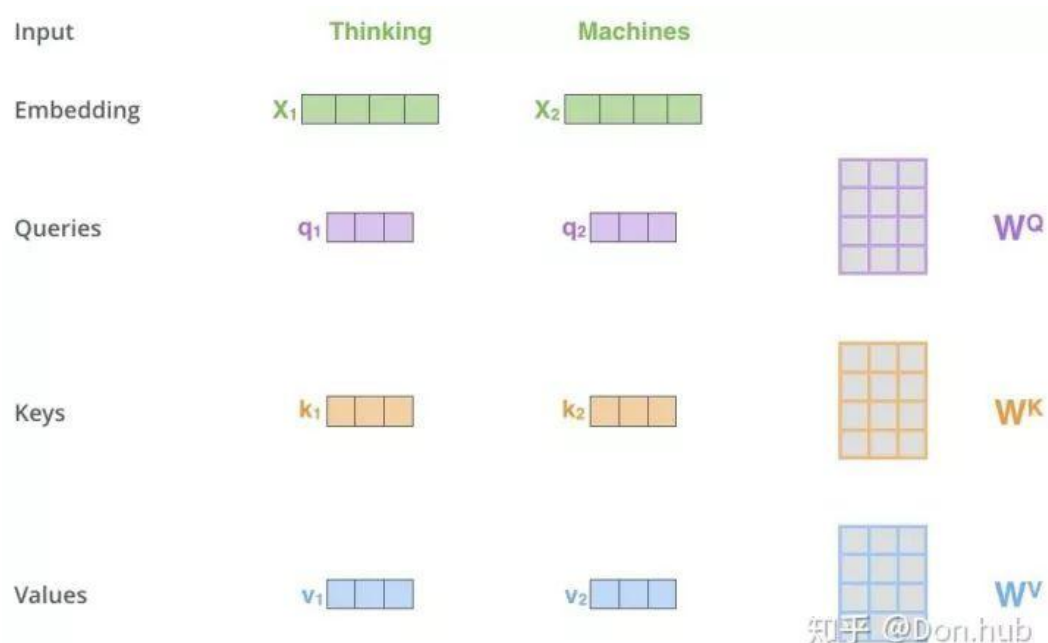
真的是对不同网络结构遍历同样的想法, 不行就改个方法, 转个置就可以了, 可行就是创新。。

$$\mathbf{h}^t = f\left(\frac{\mathbf{g}}{\sqrt{(\sigma^t)^2 + \epsilon}} \odot (\mathbf{a}^t - \mu^t) + \mathbf{b}\right) \quad \mu^t = \frac{1}{H} \sum_{i=1}^H a_i^t \quad \sigma^t = \sqrt{\frac{1}{H} \sum_{i=1}^H (a_i^t - \mu^t)^2}$$

rnn 中的 LN 就是在 $\text{net}(t)$ 上做, $\text{net}(t)$ 是一个向量。对这个向量求均值和方差, 同 BN 还需要加入 **shift** 参数和 **scale** 参数。同理, rnn 模型训练完成之后, 可以利用历史均值和方差求出一个总的均值和方差进行标准化

4. self-attention 是什么东西?

self-attention 和 seq2seq 中的 attention 是不一样的, seq2seq 中 attention 的计算是将 decoder 中的隐含层 h_i 包括了进去。而 self attention 重要的 self 这个东西, 意思是自己跟自己计算, 并不是 encoder 是自己 decoder 也是自己, 仅仅是 x 向量跟 x 向量自己计算。其中 WQ WK WV 是通过 decoder 逐层反向传播计算出来的, 并不是输入是自己输出是自己训练出来的。



5. 缩放点积

2) 缩放点积

Softmax函数对非常大的输入值会很敏感。这会导致梯度消失，并减慢学习速度，甚至使其完全停止。由于点积的平均值随着嵌入向量维度 k 的增长而增长，所以将点积的值减小一点有助于防止softmax函数的输入变得过大：

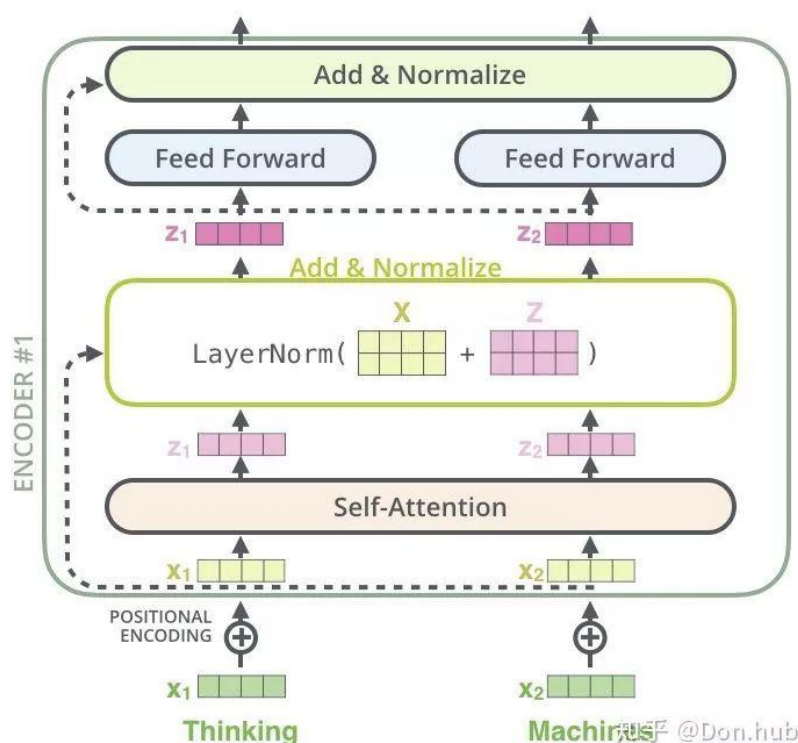
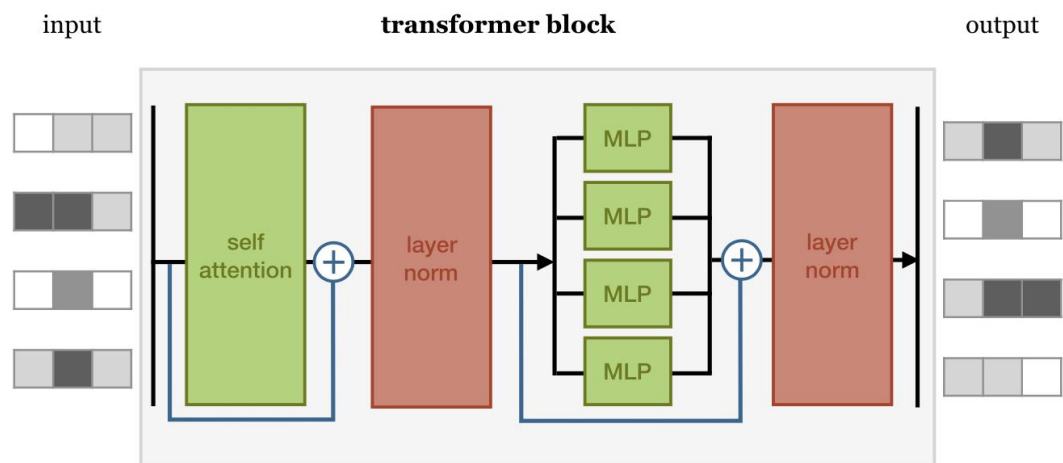
$$w'_{ij} = \frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{k}}$$

6. 彻底理解 transformer 中的一个 encoder。

1) 一个句子由多个词组成，每个词由词向量，每个词向量分别乘以 W^Q W^K W^V 就可以得到这个词向量对应的 查询向量、键向量、值向量，然后经过一波 **self attention** 操作，就得到了下图所示的 v 矩阵。然后残差连接，在做 **layer norm**。

2) **mlp** 就是做乘以一个矩阵，然后接激活函数

3) 然后再做加和和 **layer norm**



7. 怎样理解一个 decoder?

下图左边是 encoder，右边下方是 decoder。decoder 是相对于 encoder 多了一个 encoder-decoder attention。下图仅仅用了 2 个 encoder 和 2 个 decoder。论文中的 transformer 是用了 6 个。用多少是超参数，可以自己选。

理解 decoder 中最重要的部分就是理解 encoder-decoder attention。这东西和 self-attention 大致是一样的，都是用多头 self-attention。不同的是：

Q 来自于 decoder，K、V 来自于最后一个 encoder 中的 self-attention。

这里有个小细节：

encoder 中每个 self attention 要学习的参数都包括：

WQ、WK、WV。最后一个 encoder 中的 self-attention 会得到 K、V，这是将输入矩阵与 WK、WV 相乘之后的结果。而论文里说 Q 来自于 decoder，那就意味着 decoder-encoder attention 中需要学习一个 WQ 矩阵，与 decoder 中的 self-attention 输出做一个乘法得到 Q

而不是用了

- In "encoder-decoder attention" layers, the queries come from the previous decoder layer, and the memory keys and values come from the output of the encoder. This allows every position in the decoder to attend over all positions in the input sequence. This mimics the typical encoder-decoder attention mechanisms in sequence-to-sequence models such as [38, 2, 9].

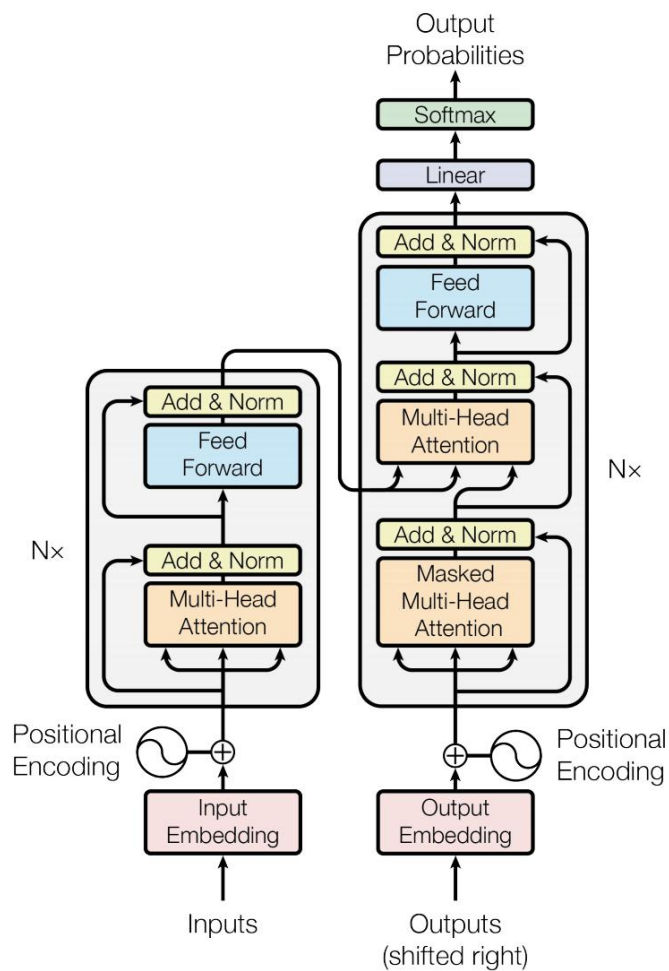
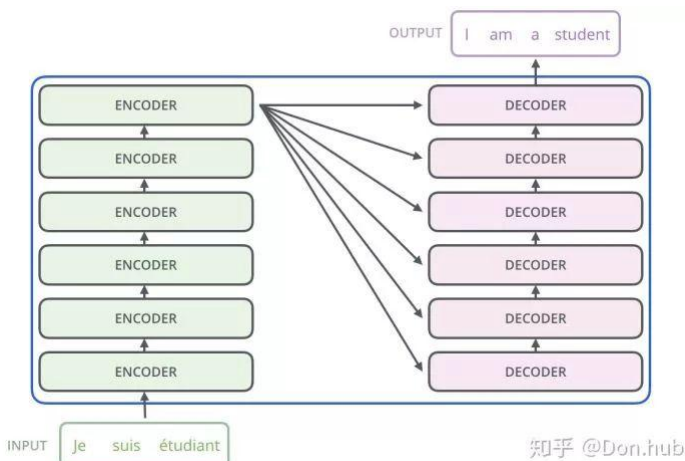
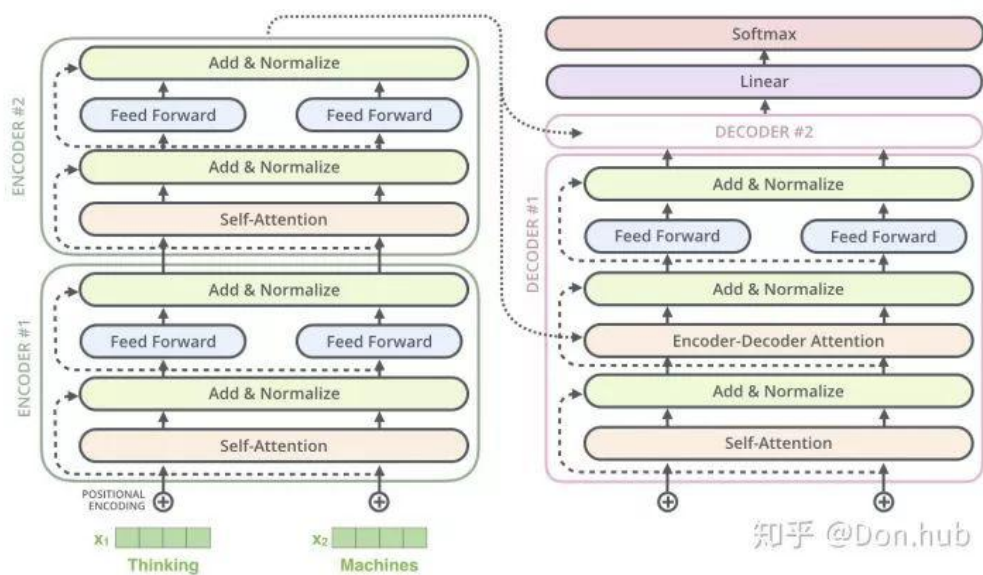
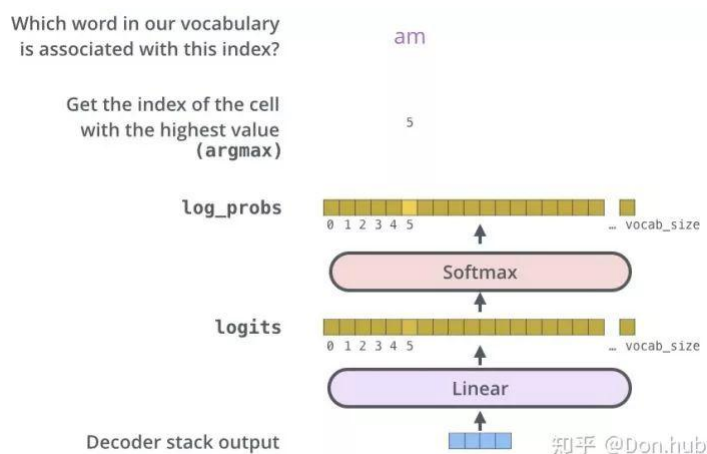


Figure 1: The Transformer - model architecture.



8. 怎样理解 output layer

decoder 的输入是一个矩阵，包含位置向量，输出的应该也是矩阵，为什么 decoder 之后的结果是一个向量呢？对向量乘以矩阵，然后做 softmax 就可以分类



9. transformer 中的 mask

首先为了并行化训练，decoder 中的输入会做 mask。因为模拟训练的过程中不应该知道该

解码的下一个词是什么。

10. transformer 如何并行？

rnn 被诟病的是 rnn 速度慢。速度慢的原因是不支持并行。

1) rnn 训练的时候是可以并行训练的，只是预测的时候是串行的

2) transformer 训练的时候跟 rnn 一样也是可以并行的，不过解码的时候只能串行。

训练的时候怎样并行？decoder 中待预测部分用 mask

11. transformer 的应用