

Transformer



11. 3. 2023.

Content

1. basic concepts

1.1 Sequence

1.2 Replace RNN with CNN

1.3 Self Attention

1.4 Self Attention is all you need

1.5 how self Attention conduct parallel computing? Matrix Compu!

1.6 Multi-head self attention (Variant of self-attention)

1.7 position encoding

1.8 Seq2seq with Attention

2. Transformer

2.1 brief intro.

2.2 basic framework

2.3 Encoder

2.4 Decoder

2.5 Attention

2.6 Self-attention

2.7 Context-attention

2.8 Context-attention (2nd layer of decoder).

2.9 Scaled dot-product Attention

2.10 Resnet

2.11 Layer normalization

2.12 Mask

Padding Mask

Sequence Mask

2.13 positional embedding

2.14 Position-wise Feed-Foward network

2.15 Implementation of Transformer.

1. Transformer.

知名应用：Bert 无监督训练的 Transformer

Seq2seq 的 model，大量用到 self attention

Sequence

RNN：与 sequence 联系紧密。

输入：seq；输出：seq

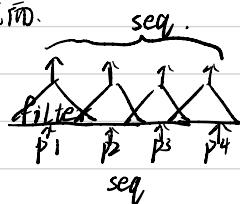
缺点：不容易并行。

单向 RNN：要算 b_4 ，必须先把 a_1, a_2, a_3 算出来。

用 CNN 代替 RNN.

idea：用 sequence 分段，每一段喂给不同的 filter，这样完成了 seq2seq。

然而...



每一个 filter 看到的信息不完整。

解决办法：多加层。

模型更复杂，但信息能看全，且能并行。

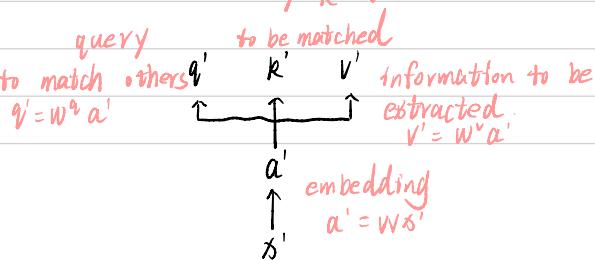
Self Attention.

一种新的 layer：self-attention，也是 seq2seq

特点：每一个输出都看了完整的 input，且 b_1, b_2, b_3, b_4 可以同时算出来。
且能并行计算。

Self-Attention is all you need

$$\text{key } k' = W^k a'$$

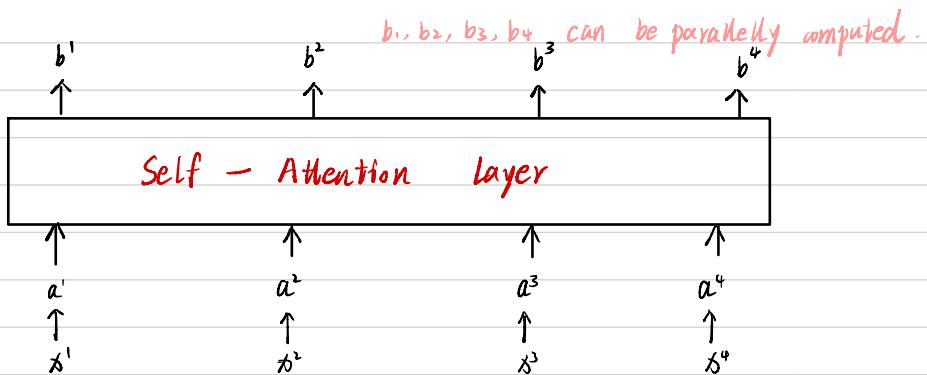
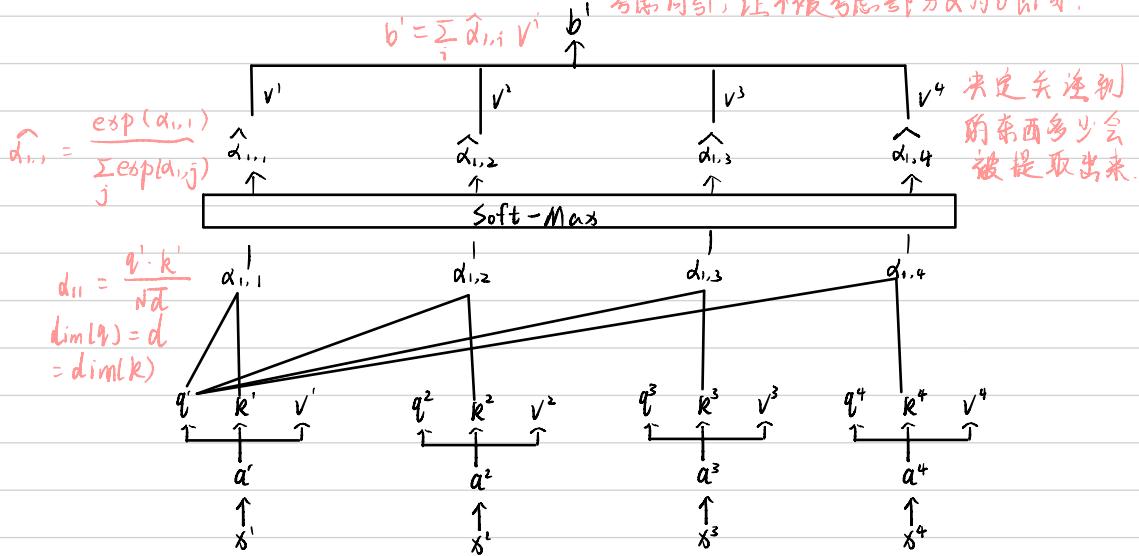


用 q^i 对每个 k^j 做 attention (attention 出来，对别的信息的关注度，如果 q^i 大，就多关注； q^i 小，就少关注)

(计算方法：用 scaled dot-product 举例，做内积，同时，控制方差)

此时， b^i 考虑了全部句子的信息。

考虑局部，让不被考虑部分为0即可。



Matrix computing

$$(q^1, q^2, q^3, q^4) = W^q | a^1, a^2, a^3, a^4 \rangle$$

$$q^i = W^q a^i$$

$$k^i = W^k a^i$$

$$v^i = W^v a^i$$

Q

I

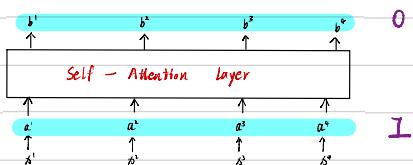
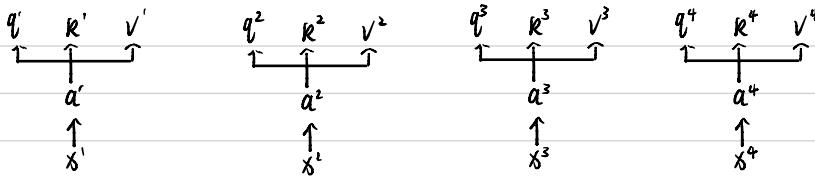
K

I

$$(V^1, V^2, V^3, V^4) = W^v | a^1, a^2, a^3, a^4 \rangle$$

V

I



$$\begin{cases} Q = W^q I \\ K = W^k I \\ V = W^v I \end{cases}$$

softmax

$$\hat{A} \leftarrow A = K^T Q$$

$$D = V \cdot \hat{A}$$

都是矩阵乘法，
可用GPU加速。

Multi-head Self-attention

提高了注意力层的能力。

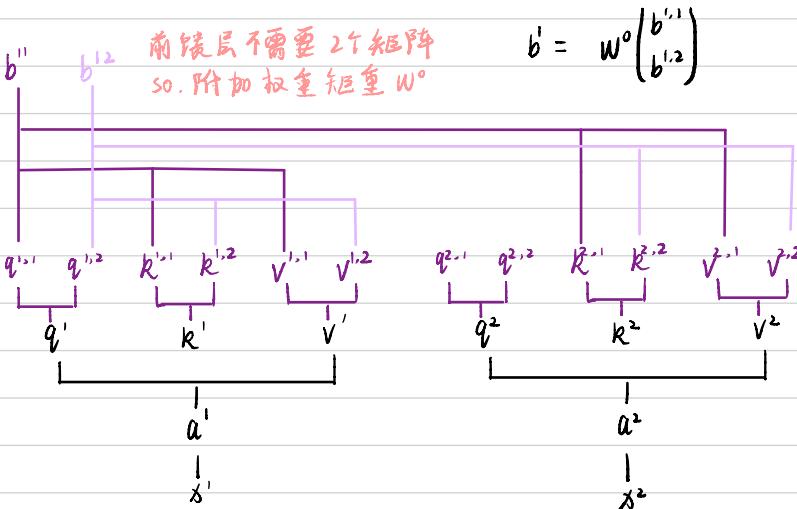
原理很简单，给出注意力层的多个表示子空间 (representation subspaces)

把 $a^i \rightarrow q^{i,1}, q^{i,2}, \dots, q^{i,8}$ (Transformer 使用 8 个注意力头)

$k^i \rightarrow k^{i,1}, \dots, k^{i,8}$

$v^i \rightarrow v^{i,1}, \dots, v^{i,8}$

每个头只能和对应的头运算。



Position Encoding

刚刚对 model 的描述缺少了一种理解单词输入顺序的方法.

\Rightarrow 为每一个 word embedding 添加一个向量, 能更好表达词与词之间的距离.

$$a^i + e^i$$

a^i : word embedding e^i : positional encoding

编码方式:

pos: 词语在序列中的位置.

$$PE(pos, 2i) = \sin(pos / 10000^{2i/d_{model}})$$

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

pos + k 位置的 encoding 可以通过 pos 位置的 encoding 线性表示

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

$$PE(pos + k, 2i) = \sin(w_i(pos + k)) = \sin(w_i(pos)) \cos(w_i(k)) + \cos(w_i(pos)) \sin(w_i(k))$$

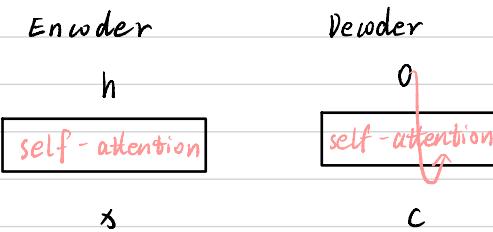
$$PE(pos + k, 2i+1) = \cos(w_i(pos + k)) = \cos(w_i(pos)) \cos(w_i(k)) - \sin(w_i(pos)) \sin(w_i(k))$$

$$PE(\underline{pos + k}, 2i) = \cos(\underline{w_i k}) PE(\underline{pos}, 2i) + \sin(\underline{w_i k}) PE(\underline{pos}, 2i+1)$$

$$PE(\underline{pos + k}, 2i+1) = \cos(\underline{w_i k}) PE(\underline{pos}, 2i+1) - \sin(\underline{w_i k}) PE(\underline{pos}, 2i)$$

$PE(pos + k)$ 可以被 $PE(pos)$ 线性表达 (词语的相对位置)

Seq2seq, with Attention



这里的重点在对 Decoder 的了解上

第1个预测结果，完全依赖于 encoding vector.

2 , encoding vector + 之前的(第1个)的预测 vector

随着翻译的句子越来越多，翻译下一个单词的运算量就变大。

复杂度 (n^{2d}) . n是翻译句子的长度, d是 word vector 的维度。

Transformer

Transformer 抛弃了传统的 CNN 和 RNN

整个结构由 Attention 机制组成

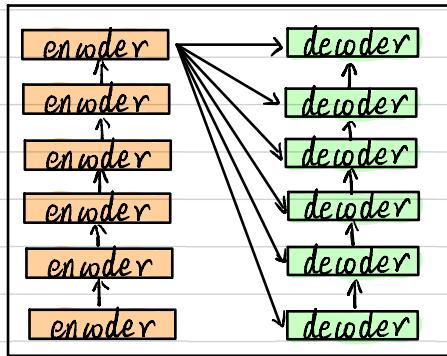
(作者的实验：Encoder, Decoder 各 6 层)

RNN 的限制。Transformer 不是 RNN 似的顺序结构，更好并行，符合 GPU 框架。

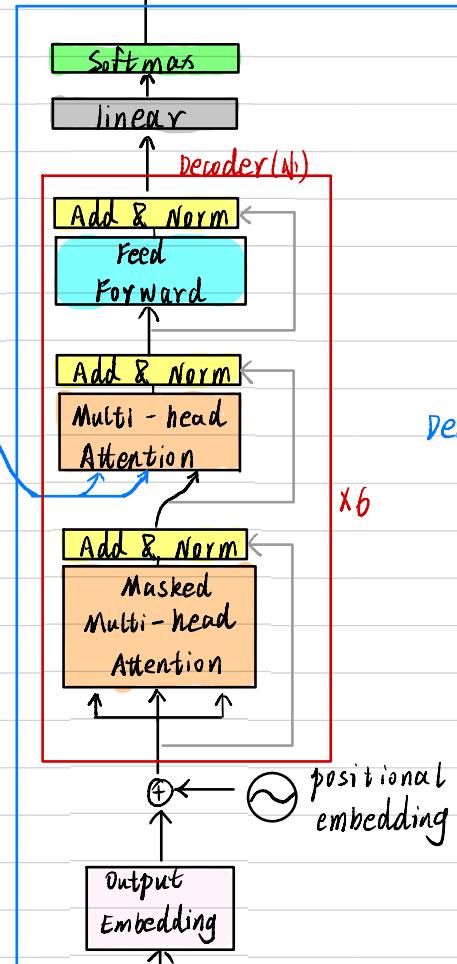
1. 时间 t 的计算依赖 $t-1$ 时刻，限制了模型的并行能力

2. LSTM 一定程度缓解了长期依赖，但对于特别长期的，无能为力。

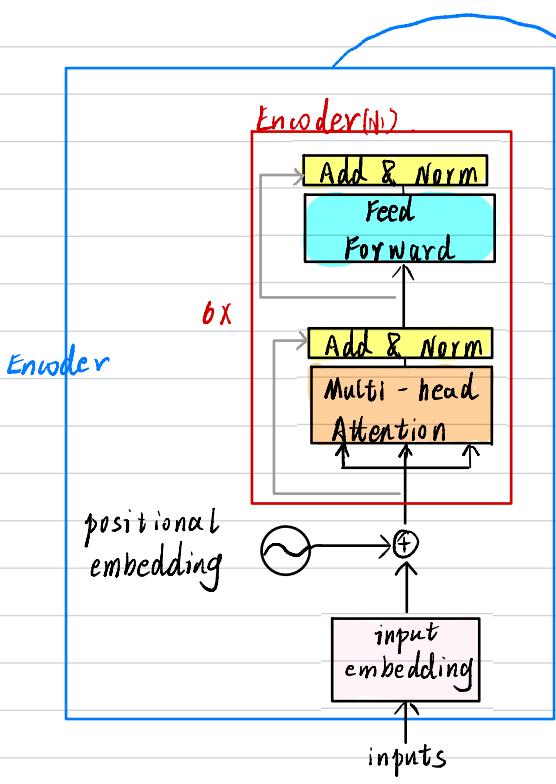
Attention 机制缩小任意两个位置之间的距离为一个常量。



Output
Probabilities



Decoder



解釋：

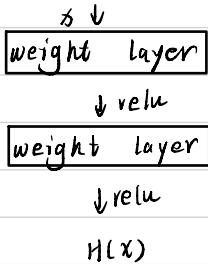
Add & Norm

残差连接 (residual connection)

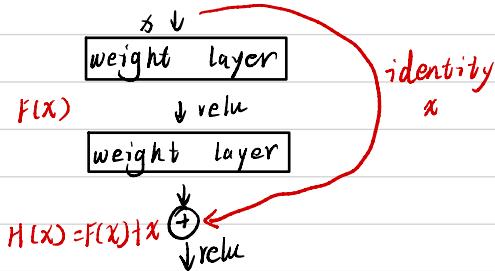
Layer Normalization

Resnet

常规的神经网络.



Resnet



解决了深度网络，深度堆叠的退化问题

Normalization

Normalization 有很多种，但他们都目的，把输入转化为均值为0，方差为1的数据。

我们在把数据送入激活函数之前进行 Normalization (归一化)，因为我们不希望数据落在激活函数的饱和区。

IC do Batch Normalization 和 Layer Normalization

Mask

Mask：掩码，对某些值进行遮盖，使其在参数更新时不产生效果。

Transformer 中涉及两种 mask，分别是 padding mask 和 sequence mask。

Padding Mask：每个批次输入序列长度不一样，我们要对输入序列进行补齐。
即，给较短的序列填充0。

Sequence mask：使 decoder 不能看见未来的信息

做法：产生一个上三角矩阵，上三角的值全1，下三角的值全0，对角线也全0

Transformer 在训练的时候是并行执行的，所以需要 sub mask.

如果利用串行的思维。

sample：I love China → 我爱中国。

1. 将输入 I love China 到 Encoder，利用 top encoder 输出最终的 tensor

(size: $1 \times 3 \times 512$ ，假设我们采用的 Embedding 长度为 512，而 batch_size = 1) 作为输入到 decoder 中的 K 和 V.

2. 将 <S> 作为 decoder 的输入，将 decoder 最终的输出与“我”做

CROSS-entropy 计算 error。能够用 <S> 作输入的原因是 shifted right()

3. 将 <S>, 我 作为 decoder 的输入，将 decoder 的最终输出与“爱”做

CROSS-entropy 计算 error.

4. 将 <S>, 我, 爱

“中”

5. <S>, 我, 爱, 中

“国”

6. <S>, 我, 爱, 中, 国

“</S>”

这样串行的情况不用掩码。

但在 Transformer 中，即 并行 的情况下，2-6 step 能同时完成，即

将 <S>, 我, 爱, 中, 国 作为 decoder 的输入，将

decoder 最终输出的 5 个 prob. vector 和 我, 爱, 中, 国, </S> 分别

做 cross entropy 计算 error.