"Attention Is All You Need"

主流序列转换模型基于 Encoder-Decoder 的递归或卷积神经网络。Transformer 仅基于注意力机制，完全取消递归或卷积。对比主流模型，实现了训练的并行化，训练时间明显减少，并且效果更好。

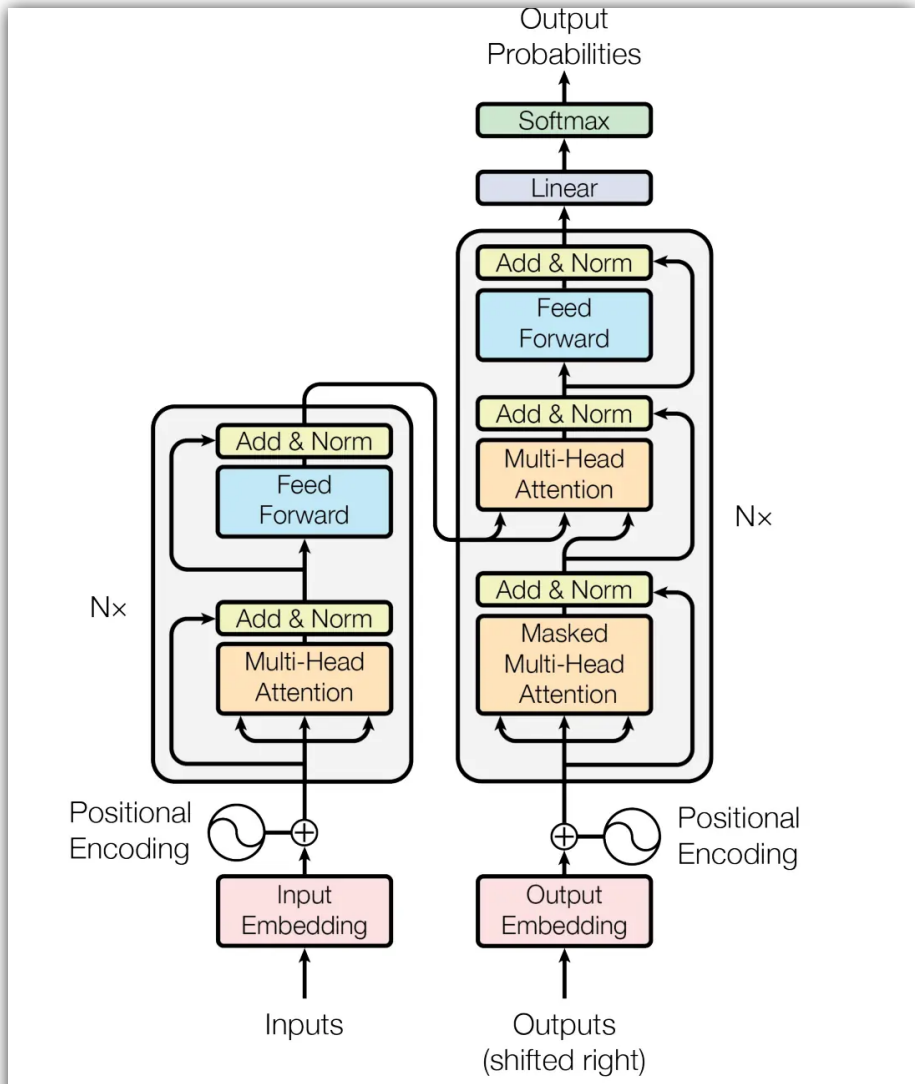


Figure 1: The Transformer - model architecture.

Encoder

词嵌入

- **input embedding:** 输入句子进行词向量嵌入，`batch size * sequence length * word vector`. word vector: 行向量
- **Positional Encoding:** `max sequence length * word vector`

位置嵌入

第 pos 个位置向量为 V，词向量长度为 L:

$$V_i = \begin{cases} \sin\left(\dfrac{pos}{10000^{i/L}}\right) & \text{i is even} \\ \cos\left(\dfrac{pos}{10000^{(i-1)/L}}\right) & \text{i is odd} \end{cases}$$

在词向量方向上，周期从 2π 增加到 20000π

- **output:** `word embedding + positional embedding`

self attention

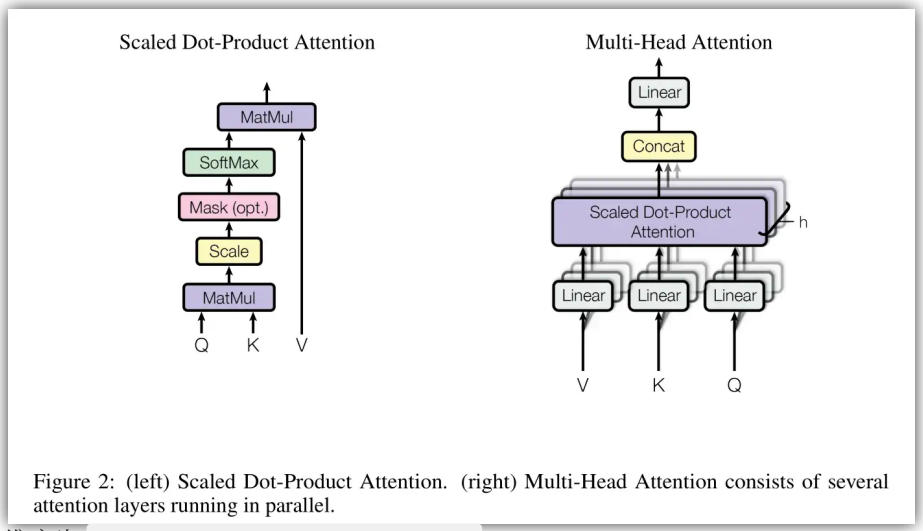


Figure 2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

设上一步输出一个句子 X, 维度为 `sequence length * word vector`

定义三个矩阵 W_Q, W_K, W_V，将 X 变换为 查询矩阵 Q，键矩阵 K，值矩阵 V，Q, K, V 形状为 `sequence length * width_qk, sequence length * width_qk, sequence length * word vector` (如果不变换，那么 Q, K, V 均为 X，论文中 Q, K, V 就指代 X)

- $Q \times K^T \rightarrow$ `sequence length * sequence length` : 行向量表示 q 查询每一个 key 的分数，相似性
- 每一行做 softmax，归一化
- $softmax\left(\frac{Q \times K^T}{\sqrt{d_k}}\right) \times V \rightarrow$ `sequence length * word vector` : 值矩阵每一行加权和
- d_k : Q, K 矩阵的宽度。假设 Q, K 每行元素均值为 0，方差为 1，则分子方差变为 d_k, 影响 softmax 的效果

Attention 中矩阵的维度

- Q : `len1 * dk`
- K : `len2 * dk`
- QK^T : `len1 * len2`
- V : `len2 * dv`
- $Attention(Q, K, V)$: `len1 * dv`
- `len1, len2` 是序列长度，本文中，Q, K 由同一句子得到，因此相等
- 为了使用残差机制，输入输出的 embedding size 相等，所以 `dv = dk`
- Transformer 使用了 Multi-Head Attention，对输入输出额外进行了两次变换，所以可以有一个隐藏维度 d'_k

Batch

- 加上 `batch` 这一维度
- batch 内句子长短不一，为了使无效区域不参与 softmax, 加上一个很大的负数

残差

$$X + self\ attention(Q, K, V)$$

Multi-Head Attention

- 使用可学习的线性变换将 Q, K, V 投影多次，并分别做 attention， 最后将输出拼接并再次投影
- 也可以一次获得 N 倍宽度的 Q, K, V，拆分后做 attention，再拼接

Feed Forward + Add & Norm 层

$$FFN(x) = relu(xW_1 + b1)W_2 + b_2$$

- 过 `linear - relu - linear - 残差 - Layer Normalization` 组合

- size: `batch size * sequence length * word vector`
- 图中有 N **Encoder Block,** 它们的 **FFN** 层权重共享

Decoder

基本同 Encoder, 使用两次 attention

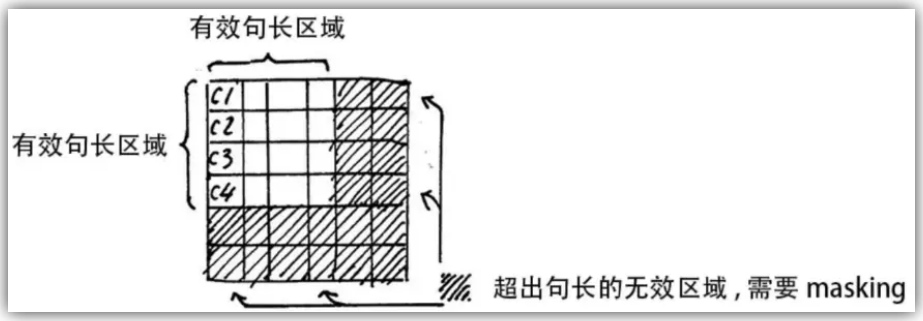
1. 第二个 attention 的 Q, K 输入是 Encoder 的输出
2. 第一个 attention 训练时使用了 Subsequence Mask，使其看不到未来的单词

Detail

Transformer 中的 Mask 将会加到 Attention 中 softmax 函数的输入上, 形状为 `sequence length * sequence length`

Padding Mask

由于句子长度不一，按最长长度作为 `sequence length`，将剩余句子进行补全。self-attention 中 softmax 输入表示不同位置之间的查询，为了不影响 softmax 输出，额外加一个 Padding Mask，即下图阴影部分为 `-inf`, 非阴影部分为 0



Layer Normalization

设 输入 x 维度为 `m * n`， `m` 为 batch_size
均值

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_{ij}$$

方差

$$\sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m}(x_{ij} - \mu_j)^2$$

归一化输出

$$LayerNorm(x_{ij}) = \frac{x_{ij} - \mu_j}{\sqrt{\sigma^2 + \epsilon}}$$

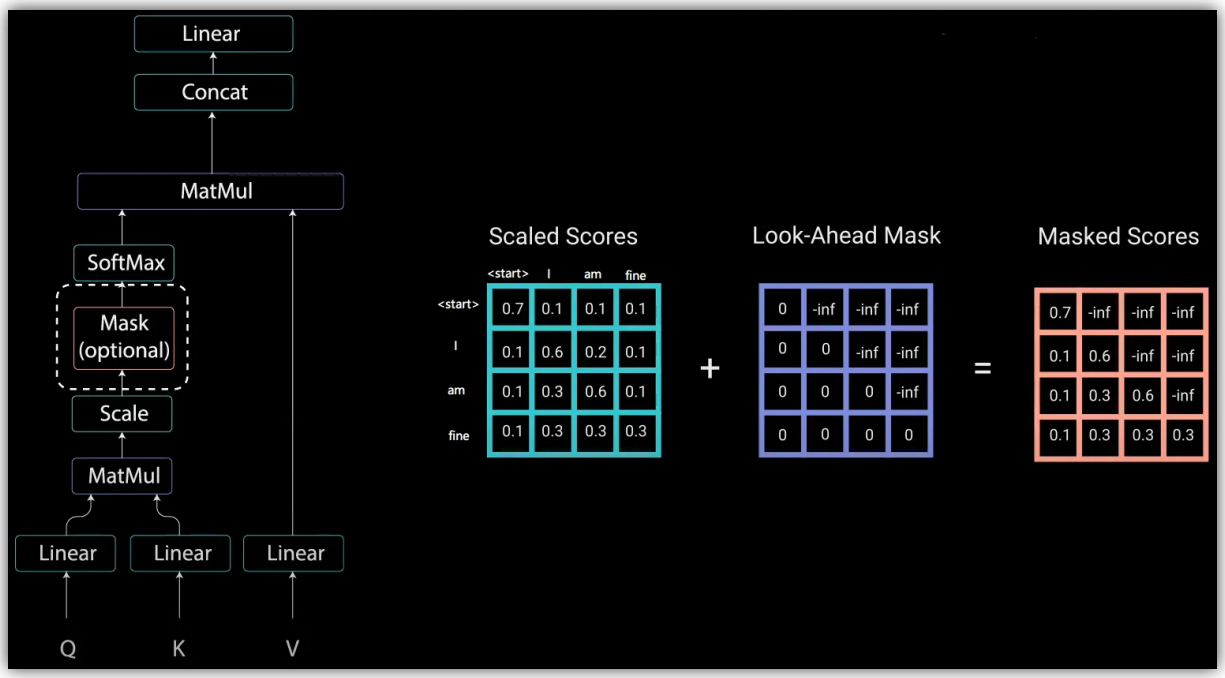
- 为了防止归一化破坏之前的信息，可加入两个可学习的参数 增益 g 和 偏置 b, 最终输出为 $LaryrNorm(X) \cdot g + b$
- 与 batch normalization 不同，这两个参数是 **element-wise**，即 size = 样本 size，这样与 batch size 无关
- 与 batch normalization 相比，在特征间做归一化，不依赖 batch_size 大小
- 不使用归一化将导致梯度消失，无法训练

Subsequence Mask

作为一个序列模型，在推理时 Encoder 输入待翻译的句子，Decoder 先输入【起始符】，然后输出 【起始符 + 下一个单词】作为下一个输入，依次类推。

在训练中并不采用这种方法，因为这样慢 `sequence length` 倍，而且实测效果不好

- 训练时 Decoder 输入完整的句子
- Decoder attention 中的 softmax 输入加上一个 Subsequence Mask，上三角为负无穷
- 无法理解其原理，论文中也没说明，应该跟 self attention 机制有关



实现

如何实现梯度下降计算图（类似 Pytorch 前端）

- 一般的神经网络实际上是一个有向无环图（不包括 RNN）。在定义网络结构时，网络节点已被自然地排序（DAG 的拓扑排序），即之后节点使用的一定是之前节点的某些输出。因此实际上只要在定义时将节点加入一个有序列表，forwarding 就是从前往后依次计算输出，backwarding 就是从后往前计算输入的梯度。
- 考虑一种特殊情况，即同一个端口被不同节点使用，根据导数的链式法则，只要将梯度累加即可。因此在 forwarding 中将输出的导数清零，在 back-propagation 中将参数的梯度累加
- 将需要更新的参数加入公共 list, 使用自定义 optimizer 进行梯度下降

Transformer 实现

- Encoder 和 Decoder 均只使用了一个 block , 未使用 Multi-Head, 训练集很简单，没有测试集

训练集

```
languageA = ["ich mochte ein bier", "ich mochte ein cola", "ich mochte ein orangensaft"]
languageB = ["i want a beer", "i want a coke", "i want a orange juice"]
```

结果

```
06/02 12:22:55 (transformer.py)[line:146] >>> inference input: ['\x00', 'ich', 'mochte', 'ein', 'bier', '\x01', '\x01', '\x01
06/02 12:22:55 (transformer.py)[line:153] >>> 0:        ['\x00', '\x01', '\x01', '\x01', '\x01', '\x01', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:161] >>> 0:        ['\x00', 'i', 'want', 'a', 'beer', '\x01', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:153] >>> 1:        ['\x00', 'i', '\x01', '\x01', '\x01', '\x01', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:161] >>> 1:        ['\x00', 'i', 'want', 'a', 'beer', '\x01', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:153] >>> 2:        ['\x00', 'i', 'want', '\x01', '\x01', '\x01', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:161] >>> 2:        ['\x00', 'i', 'want', 'a', 'beer', '\x01', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:153] >>> 3:        ['\x00', 'i', 'want', 'a', '\x01', '\x01', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:161] >>> 3:        ['\x00', 'i', 'want', 'a', 'beer', '\x01', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:153] >>> 4:        ['\x00', 'i', 'want', 'a', 'beer', '\x01', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:161] >>> 4:        ['\x00', 'i', 'want', 'a', 'beer', '\x01', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:153] >>> 5:        ['\x00', 'i', 'want', 'a', 'beer', '\x01', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:161] >>> 5:        ['\x00', 'i', 'want', 'a', 'beer', '\x01', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:153] >>> 6:        ['\x00', 'i', 'want', 'a', 'beer', '\x01', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:161] >>> 6:        ['\x00', 'i', 'want', 'a', 'beer', '\x01', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:146] >>> inference input: ['\x00', 'ich', 'mochte', 'ein', 'cola', '\x01', '\x01', '\x01
06/02 12:22:55 (transformer.py)[line:153] >>> 0:        ['\x00', '\x01', '\x01', '\x01', '\x01', '\x01', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:161] >>> 0:        ['\x00', 'i', 'want', 'a', 'coke', '\x01', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:153] >>> 1:        ['\x00', 'i', '\x01', '\x01', '\x01', '\x01', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:161] >>> 1:        ['\x00', 'i', 'want', 'a', 'coke', '\x01', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:153] >>> 2:        ['\x00', 'i', 'want', '\x01', '\x01', '\x01', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:161] >>> 2:        ['\x00', 'i', 'want', 'a', 'coke', '\x01', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:153] >>> 3:        ['\x00', 'i', 'want', 'a', '\x01', '\x01', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:161] >>> 3:        ['\x00', 'i', 'want', 'a', 'coke', '\x01', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:153] >>> 4:        ['\x00', 'i', 'want', 'a', 'coke', '\x01', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:161] >>> 4:        ['\x00', 'i', 'want', 'a', 'coke', '\x01', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:153] >>> 5:        ['\x00', 'i', 'want', 'a', 'coke', '\x01', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:161] >>> 5:        ['\x00', 'i', 'want', 'a', 'coke', '\x01', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:153] >>> 6:        ['\x00', 'i', 'want', 'a', 'coke', '\x01', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:161] >>> 6:        ['\x00', 'i', 'want', 'a', 'coke', '\x01', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:146] >>> inference input: ['\x00', 'ich', 'mochte', 'ein', 'orangensaft', '\x01', '\x01'
06/02 12:22:55 (transformer.py)[line:153] >>> 0:        ['\x00', '\x01', '\x01', '\x01', '\x01', '\x01', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:161] >>> 0:        ['\x00', 'i', 'want', 'a', 'orange', 'juice', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:153] >>> 1:        ['\x00', 'i', '\x01', '\x01', '\x01', '\x01', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:161] >>> 1:        ['\x00', 'i', 'want', 'a', 'orange', 'juice', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:153] >>> 2:        ['\x00', 'i', 'want', '\x01', '\x01', '\x01', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:161] >>> 2:        ['\x00', 'i', 'want', 'a', 'orange', 'juice', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:153] >>> 3:        ['\x00', 'i', 'want', 'a', '\x01', '\x01', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:161] >>> 3:        ['\x00', 'i', 'want', 'a', 'orange', 'juice', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:153] >>> 4:        ['\x00', 'i', 'want', 'a', 'orange', '\x01', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:161] >>> 4:        ['\x00', 'i', 'want', 'a', 'orange', 'juice', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:153] >>> 5:        ['\x00', 'i', 'want', 'a', 'orange', 'juice', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:161] >>> 5:        ['\x00', 'i', 'want', 'a', 'orange', 'juice', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:153] >>> 6:        ['\x00', 'i', 'want', 'a', 'orange', 'juice', '\x01', '\x01']
06/02 12:22:55 (transformer.py)[line:161] >>> 6:        ['\x00', 'i', 'want', 'a', 'orange', 'juice', '\x01', '\x01']
```