解析Transformer模型

原创 zzk GiantPandaCV 2020-10-07

收录于话题
#CV中注意力机制 9 #Transformer 1

> GiantPandaCV导语：这篇文章为大家介绍了一下Transformer模型，Transformer模型原本是NLP中的一个Idea，后来也被引入到计算机视觉中，例如前面介绍过的DETR就是将目标检测算法和Transformer进行结合，另外基于Transformer的魔改工作最近也层出不穷，感兴趣的同学可以了解一下。

要读DETR论文视频解读的同学可以点下方链接：

CV和NLP的统一，DETR 目标检测框架论文解读

前言

Google于2017年提出了《Attention is all you need》，抛弃了传统的RNN结构，**「设计了一种Attention机制，通过堆叠Encoder-Decoder结构」**，得到了一个Transformer模型，在机器翻译任务中**「取得了BLEU值的新高」**。在后续很多模型也基于Transformer进行改进，也得到了很多表现不错的NLP模型，前段时间，相关工作也引申到了CV中的目标检测，可参考FAIR的DETR模型

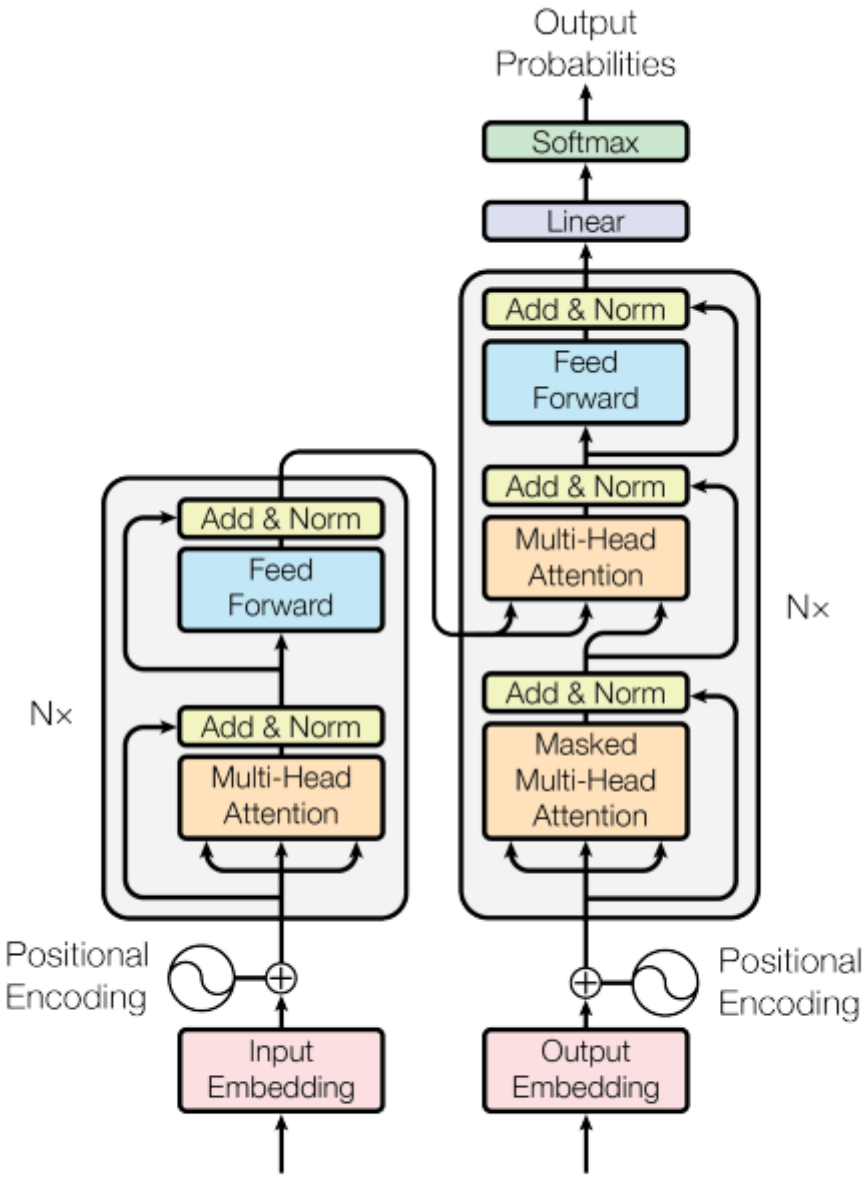
引入问题

常见的时间序列任务采用的模型通常都是RNN系列，然而RNN系列模型的顺序计算方式带来了两个问题

1. 某个时间状态h_t，依赖于上一时间步状态h_{t-1}，导致模型**「不能通过并行计算来加速」**
2. RNN系列的魔改模型比如GRU，LSTM，虽然**「引入了门机制」**(gate)，但是对**「长时间依赖的问题缓解能力有限」**，不能彻底解决

因此我们设计了一个全新的结构Transformer，通过Attention注意力机制，来对时间序列更好的建模。同时我们不需要像RNN那样顺序计算，从而能让模型更能充分发挥并行计算性能。

模型架构



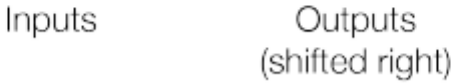
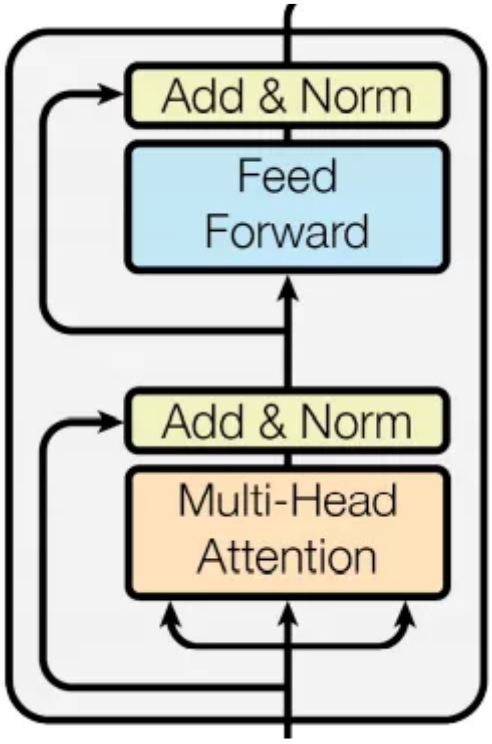


Figure 1: The Transformer - model architecture.

TransFormer模型架构一览

上图展示的就是Transformer的结构，左边是编码器Encoder，右边是解码器Decoder。通过多次堆叠，形成了Transformer。下面我们分别看下Encoder和Decoder的具体结构

Encoder



编码器架构

Encoder结构如上，它由以下sublayer构成

- Multi-Head Attention 多头注意力
- Feed Forward 层

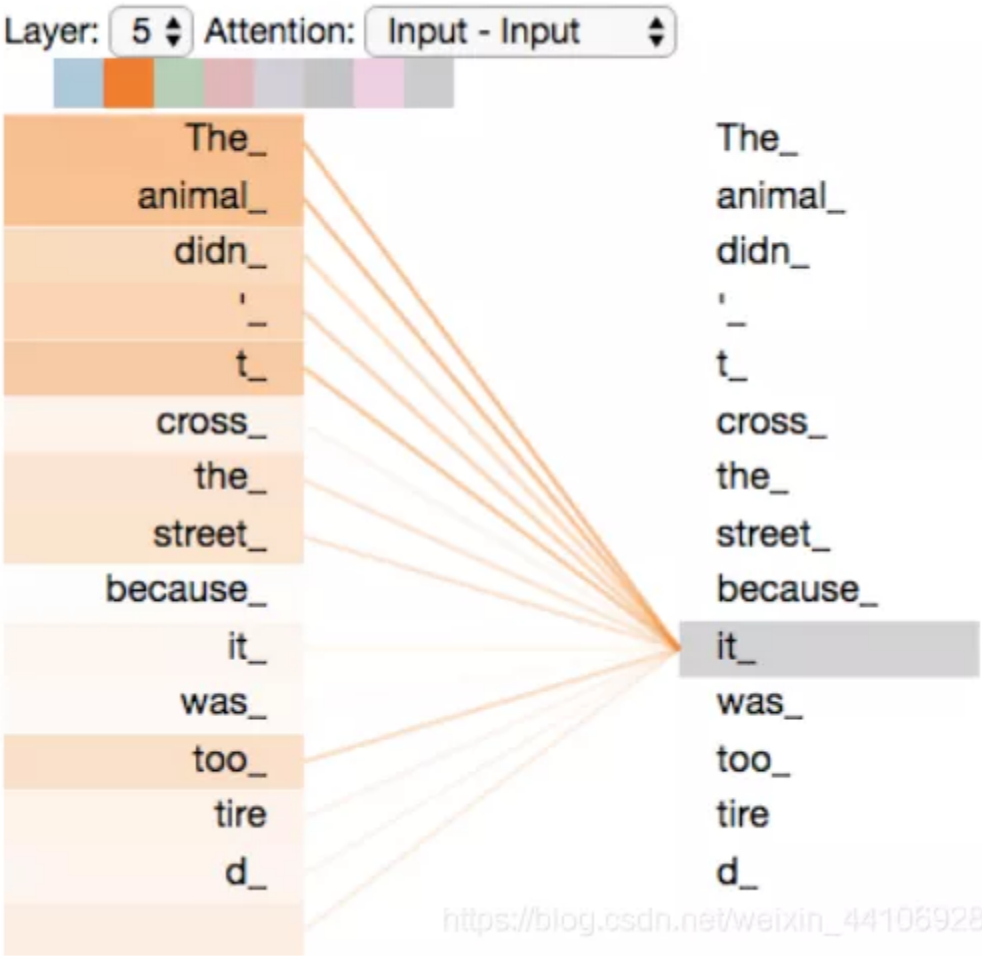
Self Attention

Multi-Head Attention多头注意力层是由多个self attention来组成的，因此我们先讲解下模型的自注意力机制。

在一句话中，如果给每个词都分配相同的权重，那么会很难让模型去学习词与词对应的关系。举个例子

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

我们需要让模型去推断 `it` 所指代的东西，当我们给模型加了注意力机制，它的表现如下

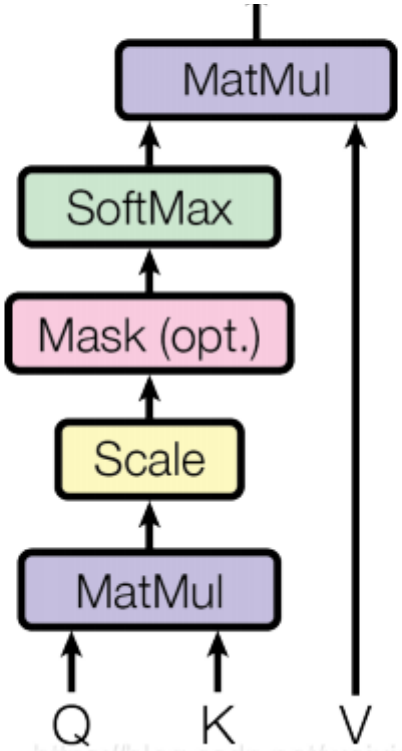


注意力机制效果

我们通过注意力机制，让模型能看到输入的各个单词，然后它会更加关注于 `The animal`，从而更好的进行编码。

论文里将attention模块记为「**Scaled Dot-Product Attention**」，计算如下

Scaled Dot-Product Attention

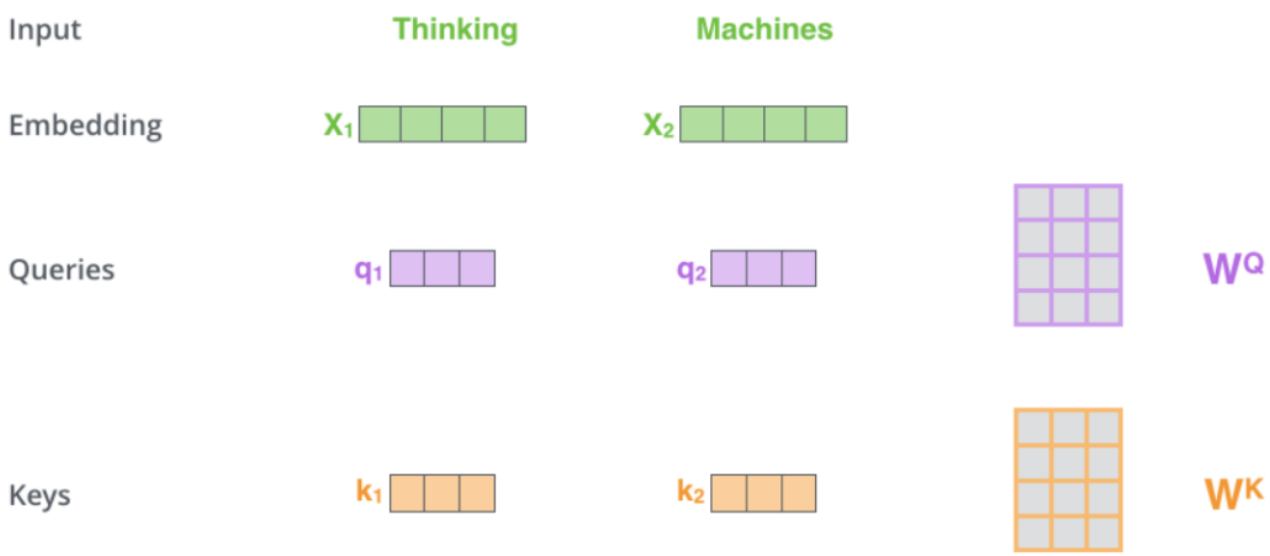


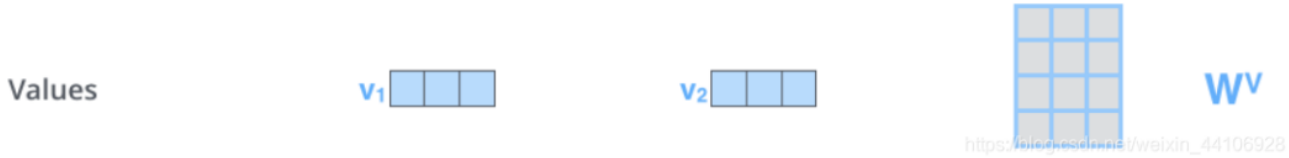
自注意力机制一览

$$Attention(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

- Q 代表 Query 矩阵
- K 代表 Key 矩阵
- V 代表 Value 矩阵
- dk 是一个缩放因子

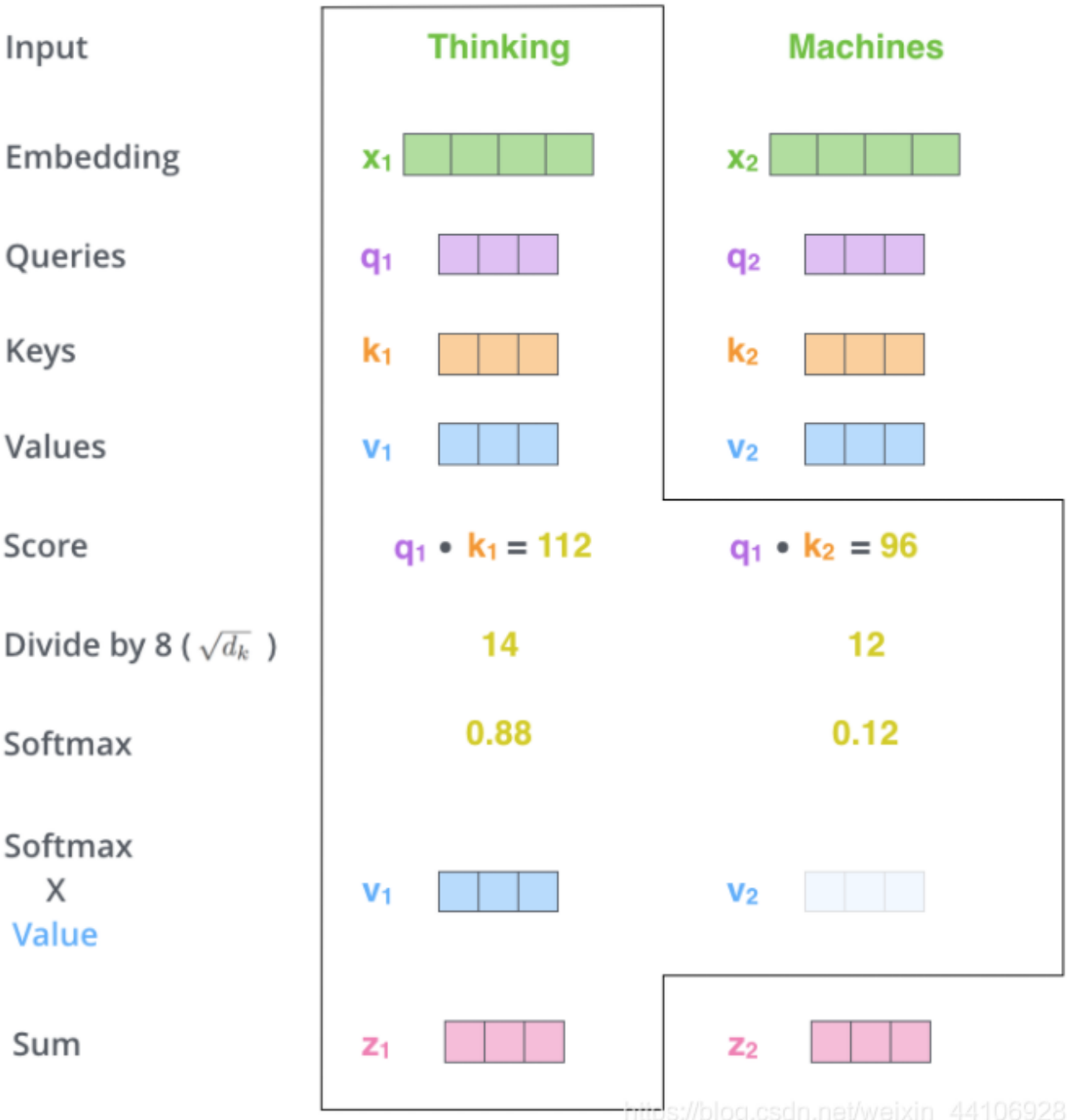
其中 Q，K，V（向量长度为64）是由输入X经过三个不同的权重矩阵（shape=512x64）计算得来，





经过Embedding的向量X，与右边三个权重矩阵相乘，分别得到Query，Key，Value三个向量

下面我们看一个具体例子



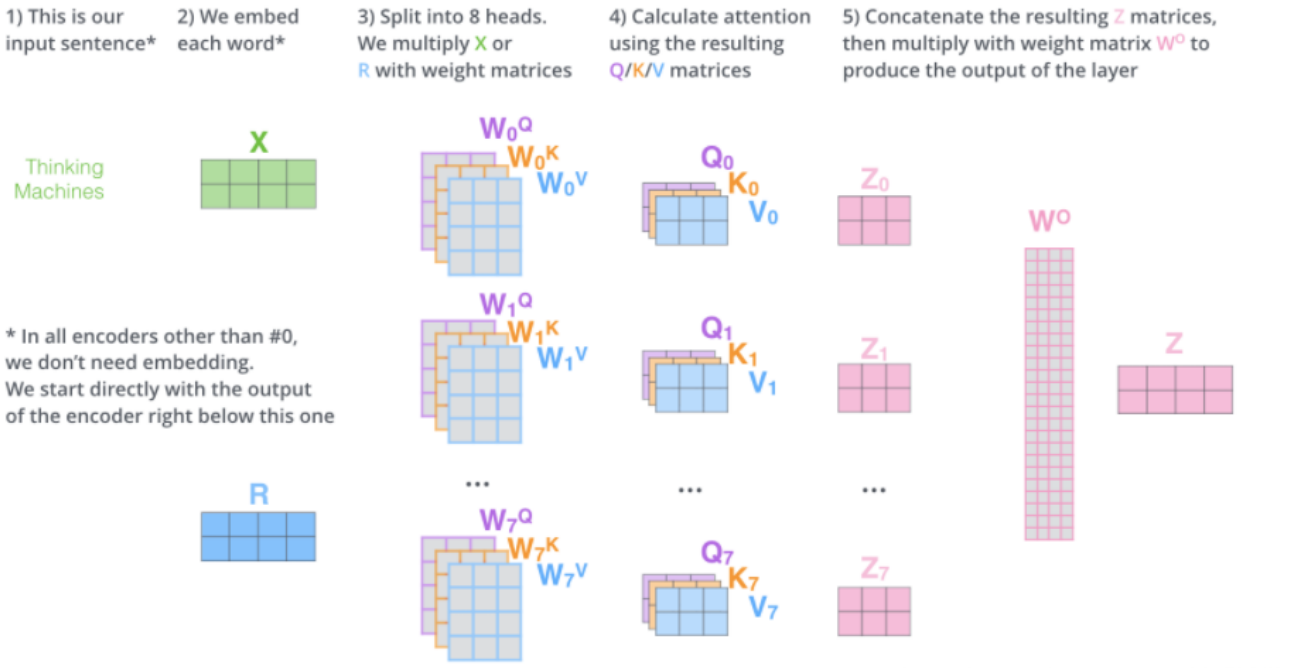
注意力机制运算过程

以 Thinking 这个单词为例，我们需要计算整个句子所有词与它的score。

- X1是 `Thinking` 对应的Embedding向量。
- 然后我们计算得到了X1对应的查询向量q1
- 然后我们与Key向量进行相乘，来计算相关性，这里记作Score。**「这个过程可以看作是当前词的搜索q1，与其他词的key去匹配」**。当相关性越高，说明我们需要放更多注意力在上面。
- 然后除以缩放因子，做一个Softmax运算
- Softmax后的结果与Value向量相乘，得到最终结果

MultiHead-Attention

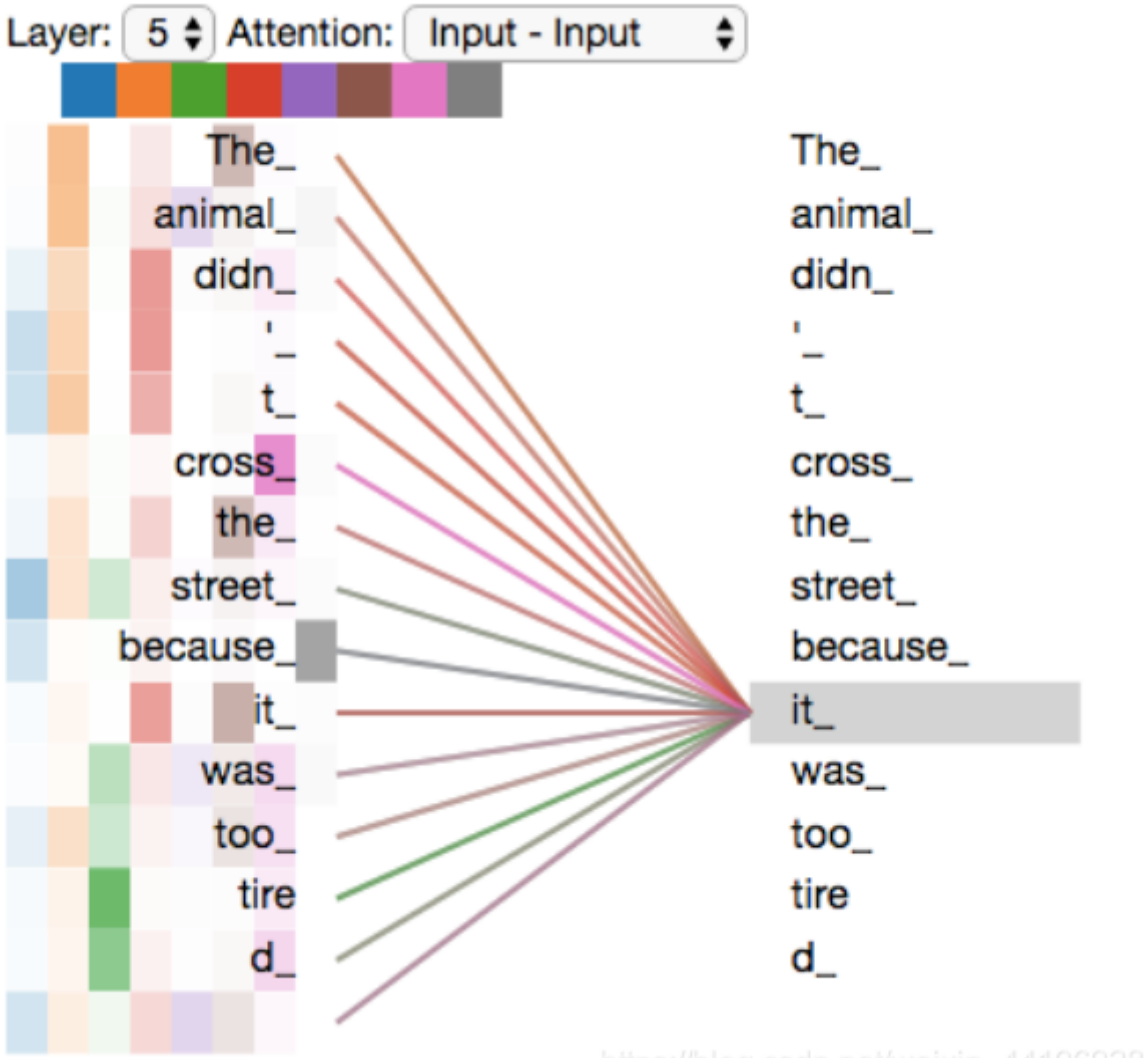
理解了自注意力机制后，我们可以很好的理解多头注意力机制。简单来说，多头注意力其实就是合并了多个自注意力机制的结果



多头注意力机制概览，将多个自注意力机制并在一起

我们以原文的8个注意力头为例子，多头注意力的操作如下

- 将输入数据X分别输入进8个自注意力模块
- 分别计算出每个自注意力模块的结果Z0，Z1，Z2.....Z7
- 将各个自注意力模块结果Zi拼成一个大矩阵Z
- 经过一层全连接层，得到最终的输出 最后多头注意力的表现类似如下



多头注意力机制效果

Feed Forward Neural Network

这个FFN模块比较简单，本质上全是两层全连接层加一个Relu激活

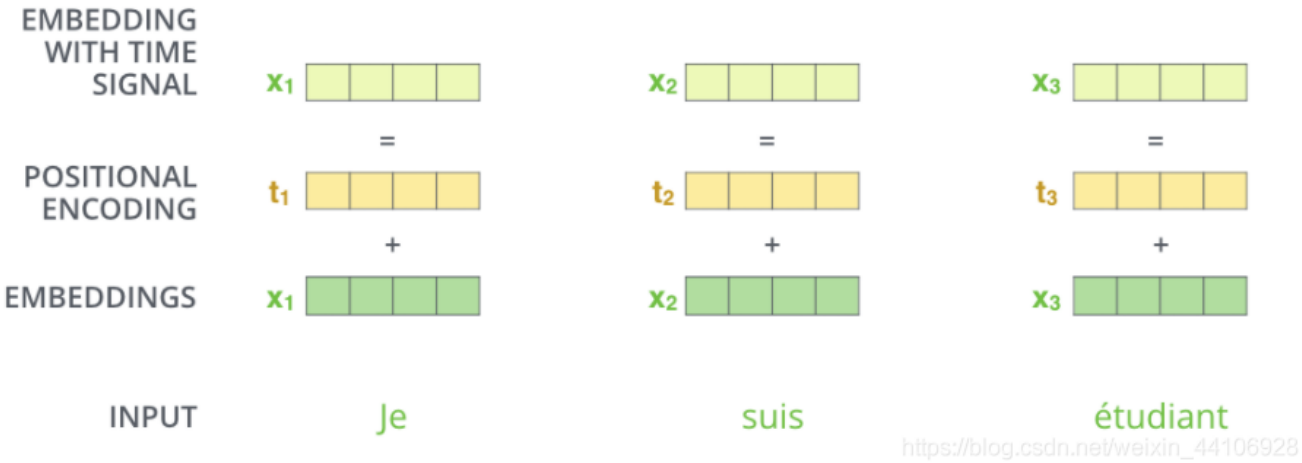
$$X = Dense_1(X)X = Relu(X)Out = Dense_2(X)$$

Positional Encoding

摒弃了CNN和RNN结构，我们无法很好的利用序列的顺序信息，因此我们采用了额外的一个位置编码来进行缓解

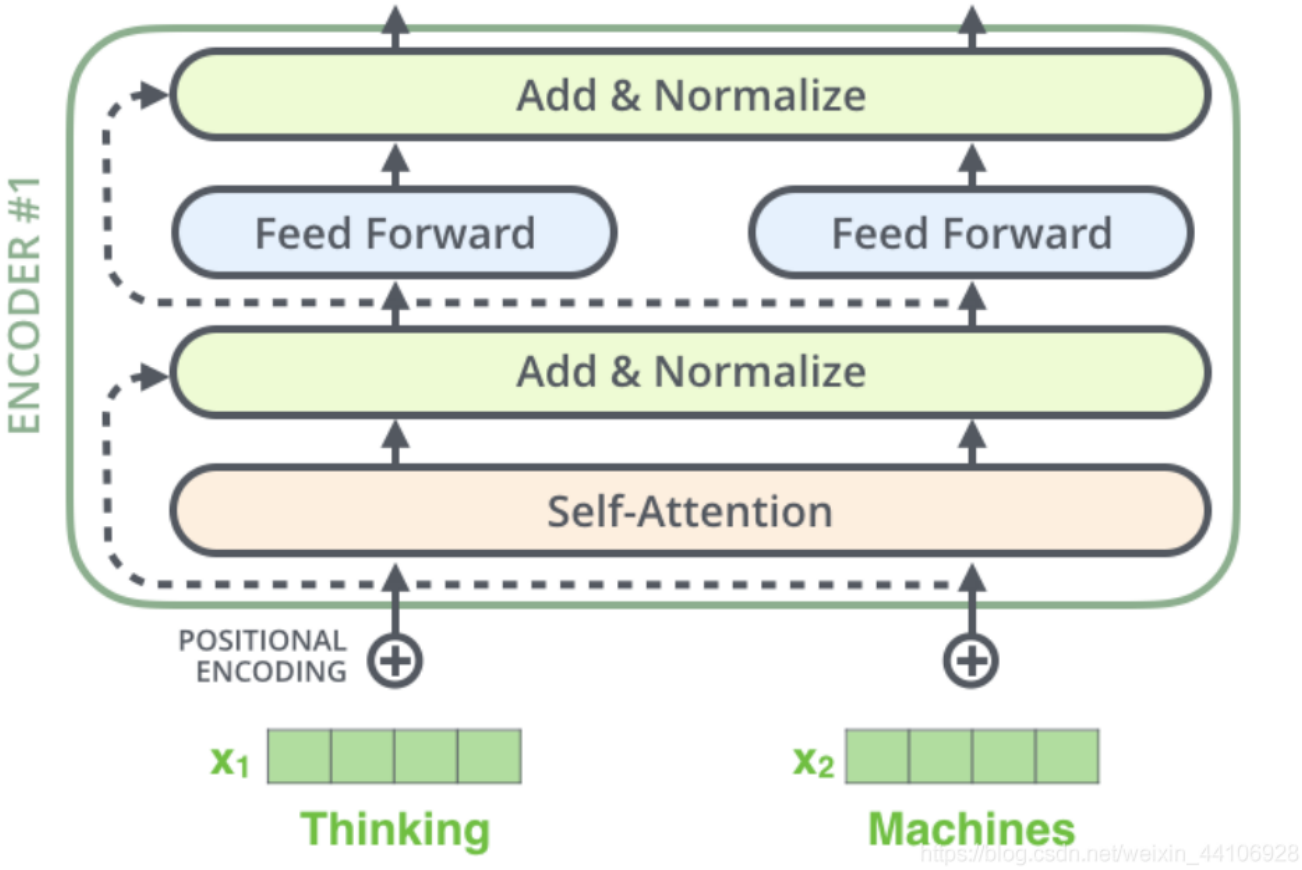
$$PE(pos, 2i) = sin(\frac{pos}{1000^{\frac{2i}{d_{model}}}})PE(pos, 2i+1) = cos(\frac{pos}{1000^{\frac{2i}{d_{model}}}})$$

然后与输入相加，通过引入位置编码，给词向量中赋予了单词的位置信息



位置编码

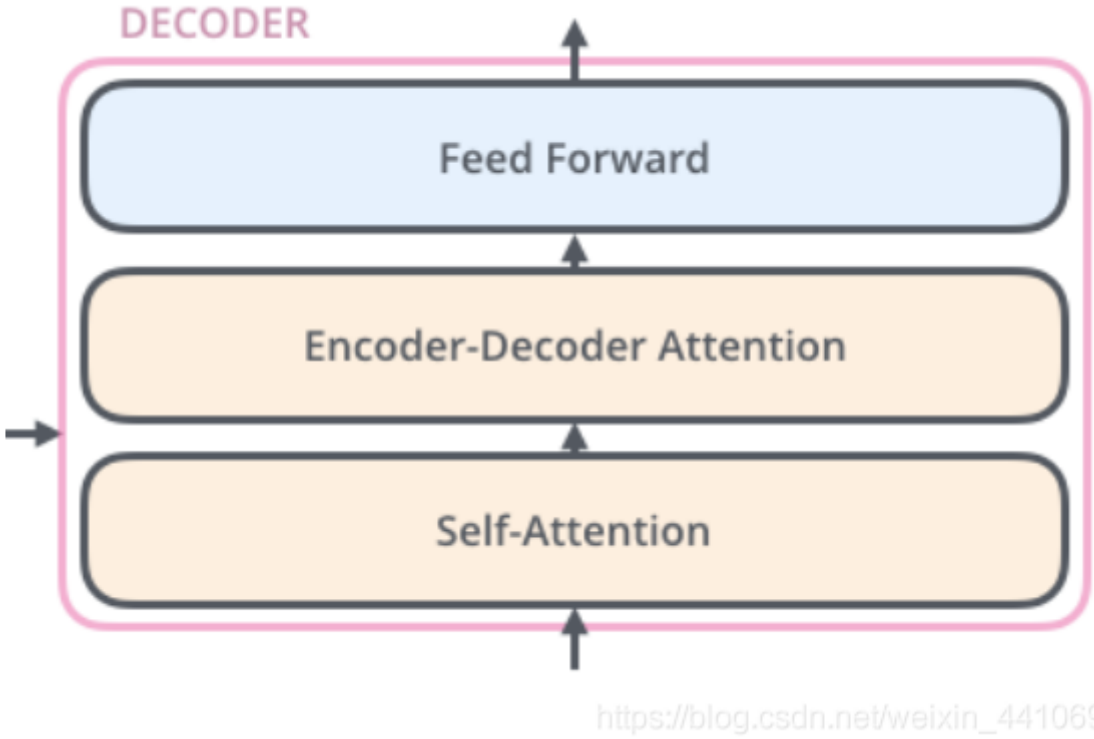
下图是总Encoder的架构



Encoder的整体结构

Decoder

Decoder的结构与Encoder的结构很相似



Decoder结构

「只不过额外引入了当前翻译和编码特征向量的注意力」，这里就不展开了。

代码

这里参考的是TensorFlow的官方实现notebook transformer.ipynb

位置编码

```
def get_angles(pos, i, d_model):
  angle_rates = 1 / np.power(10000, (2 * (i//2)) / np.float32(d_model))
  return pos * angle_rates

def positional_encoding(position, d_model):
  angle_rads = get_angles(np.arange(position)[:, np.newaxis],
                          np.arange(d_model)[np.newaxis, :],
                          d_model)

  # apply sin to even indices in the array; 2i
  angle_rads[:, 0::2] = np.sin(angle_rads[:, 0::2])

  # apply cos to odd indices in the array; 2i+1
  angle_rads[:, 1::2] = np.cos(angle_rads[:, 1::2])
```

```
  pos_encoding = angle_rads[np.newaxis, ...]

  return tf.cast(pos_encoding, dtype=tf.float32)
```

这里就是根据公式，生成位置编码

Scaled-Dot Attention

```
def scaled_dot_product_attention(q, k, v, mask):
  """Calculate the attention weights.
  q, k, v must have matching leading dimensions.
  k, v must have matching penultimate dimension, i.e.: seq_len_k = seq_len_v
  The mask has different shapes depending on its type(padding or look ahead)
  but it must be broadcastable for addition.

  Args:
    q: query shape == (..., seq_len_q, depth)
    k: key shape == (..., seq_len_k, depth)
    v: value shape == (..., seq_len_v, depth_v)
    mask: Float tensor with shape broadcastable
          to (..., seq_len_q, seq_len_k). Defaults to None.

  Returns:
    output, attention_weights
  """

  matmul_qk = tf.matmul(q, k, transpose_b=True)  # (..., seq_len_q, seq_len_

  # scale matmul_qk
  dk = tf.cast(tf.shape(k)[-1], tf.float32)
  scaled_attention_logits = matmul_qk / tf.math.sqrt(dk)

  # add the mask to the scaled tensor.
  if mask is not None:
    scaled_attention_logits += (mask * -1e9)

  # softmax is normalized on the last axis (seq_len_k) so that the scores
  # add up to 1.
  attention_weights = tf.nn.softmax(scaled_attention_logits, axis=-1)  # (..
```

```
  output = tf.matmul(attention_weights, v)  # (..., seq_len_q, depth_v)

  return output, attention_weights
```

输入的是Q，K，V矩阵和一个mask掩码向量 根据公式进行矩阵相乘，得到最终的输出，以及注意力权重

MultiheadAttention

这里的代码就是将多个注意力结果组合在一起

```
class MultiHeadAttention(tf.keras.layers.Layer):
  def __init__(self, d_model, num_heads):
    super(MultiHeadAttention, self).__init__()
    self.num_heads = num_heads
    self.d_model = d_model

    assert d_model % self.num_heads == 0

    self.depth = d_model // self.num_heads

    self.wq = tf.keras.layers.Dense(d_model)
    self.wk = tf.keras.layers.Dense(d_model)
    self.wv = tf.keras.layers.Dense(d_model)

    self.dense = tf.keras.layers.Dense(d_model)

  def split_heads(self, x, batch_size):
    """Split the last dimension into (num_heads, depth).
    Transpose the result such that the shape is (batch_size, num_heads, seq_
    """
    x = tf.reshape(x, (batch_size, -1, self.num_heads, self.depth))
    return tf.transpose(x, perm=[0, 2, 1, 3])

  def call(self, v, k, q, mask):
    batch_size = tf.shape(q)[0]

    q = self.wq(q)  # (batch_size, seq_len, d_model)
    k = self.wk(k)  # (batch_size, seq_len, d_model)
    v = self.wv(v)  # (batch_size, seq_len, d_model)
```

```
    q = self.split_heads(q, batch_size)  # (batch_size, num_heads, seq_len_q
    k = self.split_heads(k, batch_size)  # (batch_size, num_heads, seq_len_k
    v = self.split_heads(v, batch_size)  # (batch_size, num_heads, seq_len_v

    # scaled_attention.shape == (batch_size, num_heads, seq_len_q, depth)
    # attention_weights.shape == (batch_size, num_heads, seq_len_q, seq_len_
    scaled_attention, attention_weights = scaled_dot_product_attention(
        q, k, v, mask)

    scaled_attention = tf.transpose(scaled_attention, perm=[0, 2, 1, 3])  #

    concat_attention = tf.reshape(scaled_attention,
                                  (batch_size, -1, self.d_model))  # (batch_

    output = self.dense(concat_attention)  # (batch_size, seq_len_q, d_model

    return output, attention_weights
```

FFN

```
def point_wise_feed_forward_network(d_model, dff):
  return tf.keras.Sequential([
      tf.keras.layers.Dense(dff, activation='relu'),  # (batch_size, seq_len
      tf.keras.layers.Dense(d_model)  # (batch_size, seq_len, d_model)
  ])
```

有了这三个模块，就可以组合成Encoder和Decoder了，这里限于篇幅就不展开，有兴趣的可以看下官方notebook

总结

Transformer这个模型设计还是很有特点的，虽然本质上还是全连接层的各个组合，但是通过不同的权重矩阵，对序列进行注意力机制建模。并且根据模型无法利用序列顺序信息的缺陷，设计了一套位置编码机制，赋予词向量位置信息。近年来对Transformer的魔改也有很多，相信这个模型还有很大的潜力去挖掘。

GiantPandaCV 发起了一个读者讨论