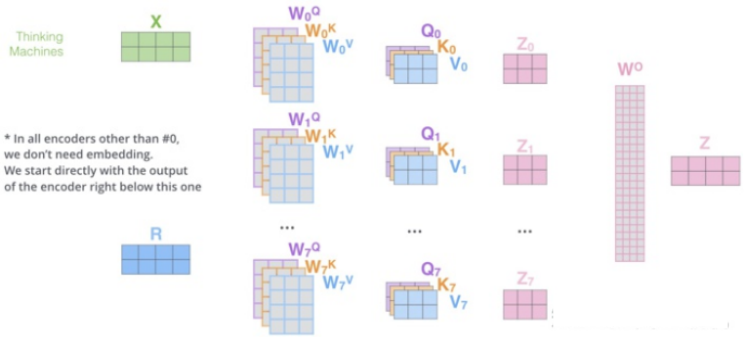
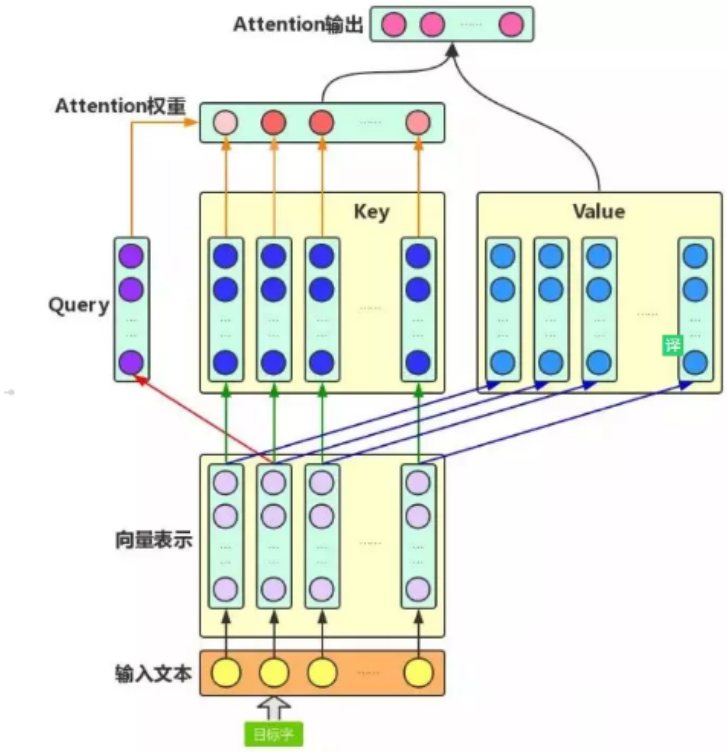
Transformer

- 模块篇
  - 为什么 会有self-attention?
    - CNN 所存在的长距离依赖问题;
    - RNN 所存在的无法并行化问题【虽然能够在一定长度上缓解长距离依赖问题】;
    - 传统 Attention
      - 方法：基于源端和目标端的隐向量计算Attention，
      - 结果：源端每个词与目标端每个词间的依赖关系 【源端->目标端】
      - 问题：忽略了 远端或目标端 词与词间 的依赖关系
  - self-attention 的核心思想是什么?
    - 核心思想：self-attention的结构在计算每个token时，总是会考虑整个序列其他token的表达;
      - 举例：“我爱中国”这个序列，在计算"我"这个词的时候，不但会考虑词本身的embedding，也同时会考虑其他词对这个词的影响
  - self-attention 的目的是什么?
    - 目的：学习句子内部的词依赖关系，捕获句子的内部结构。
  - self-attention 的怎么计算的?
    - self-attention 计算公式

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



    - self-attention 结构图



- **一句话概述：每个位置的embedding对应 Q,K,V 三个向量，这三个向量分别是embedding点乘 WQ,WK,WV 矩阵得来的。每个位置的Q向量去乘上所有位置的K向量，其结果经过softmax变成attention score，以此作为权重对所有V向量做加权求和即可。**
- **具体步骤:**
  - embedding层：
    - 目的：将词转化成embedding向量；
  - Q，K，V 向量计算：
    - 根据 embedding 和权重矩阵，得到Q，K，V；
      - `Question：权重矩阵哪里来的？`
      - Q：查询向量，目标字作为 Query；
      - K：键向量，其上下文的各个字作为 Key；
      - V：值向量，上下文各个字的 Value；
  - 权重 score 计算：
    - 查询向量 query 点乘 key;
    - 目的：计算其他词对这个词的重要性，也就是权值；

- scale 操作：
  - 乘以 $\frac{1}{\sqrt{d_k}}$；
  - 目的：起到调节作用，使得内积不至于太大。实际上是Q，K，V的最后一个维度，当 d_k 越大，QK^T 就越大，可能会将 Softmax 函数推入梯度极小的区域；
- Softmax 归一化：
  - 经过 Softmax 归一化；
- Attention 的输出计算：
  - 权值 score 和各个上下文字的 V 向量 的加权求和
  - 目的：把上下文各个字的 V 融入目标字的原始 V 中
- **举例：**
  - 答案就是文章中的Q，K，V，这三个向量都可以表示"我"这个词，但每个向量的作用并不一样，Q 代表 query，当计算"我"这个词时，它就能代表"我"去和其他词的 K 进行点乘计算其他词对这个词的重要性，所以此时其他词(包括自己)使用 K 也就是 key 代表自己，当计算完点乘后，我们只是得到了每个词对“我”这个词的权重，需要再乘以一个其他词(包括自己)的向量，也就是V(value)，才完成"我"这个词的计算，同时也是完成了用其他词来表征"我"的一个过程
- **优点：**
  - 捕获源端和目标端、词与词间的依赖关系
  - 捕获源端或目标端自身词与词间的依赖关系
- 代码讲解【注：代码采用 tensorflow 框架编写】

```
def scaled_dot_product_attention(q, k, v, mask):
    # s1：权重 score 计算：查询向量 query 点乘 key
    matmul_qk = tf.matmul(q, k, transpose_b=True)
    # s2：scale 操作：除以 sqrt(dk)，将 Softmax 函数推入梯度极小的区域
    dk = tf.cast(tf.shape(k)[-1], tf.float32)
    scaled_attention_logits = matmul_qk / tf.math.sqrt(dk)
    # s3：
    if mask is not None:
        scaled_attention_logits += (mask * -1e9)
    # s4：Softmax 归一化
    attention_weights = tf.nn.softmax(scaled_attention_logits, axis=-1)
    # s5：加权求和
    output = tf.matmul(attention_weights, v)
    return output, attention_weights
```

- self-attention 为什么Q和K使用不同的权重矩阵生成，为何不能使用同一个值进行自身的点乘?
  - 因为Q、K、V的不同，可以保证在不同空间进行投影，增强了表达能力，提高了泛化能力。
  - 补充：
    - （1）由三个输入，分别为V，K，Q，此处V=K=Q=matEnc（后面会经过变化变的不一样）
    - （2）首先分别对V，K，Q三者分别进行线性变换，即将三者分别输入到三个单层神经网络层，激活函数选择relu，输出新的V，K，Q（三者shape都和原来shape相同，即经过线性变换时输出维度和输入维度相同）；
    - （3）然后将Q在最后一维上进行切分为num_heads(假设为8,必须可以被matENC整除)段，然后对切分完的矩阵在axis=0维上进行concat链接起来；对V和K都进行和Q一样的操作；操作后的矩阵记为Q_,K_,V_；
- 为什么采用点积模型的 self-attention 而不采用加性模型?
  - 主要原因：在理论上，加性模型和点积模型的复杂度差不多，但是点积模型在实现上可以更好地利用矩阵乘法，而矩阵乘法有很多加速策略，因此能加速训练。但是论文中实验表明，当维度d越来越大时，加性模型的效果会略优于点积模型，原因应该是加性模型整体上还是比点积模型更复杂（有非线性因素）。
    - `Question：什么是加性模型？`
- Transformer 中在计算 self-attention 时为什么要scaled dot product? 即除以 $\sqrt{d}$?
  - 一句话回答：当输入信息的维度 d 比较高，会导致 softmax 函数接近饱和区，梯度会比较小。因此，缩放点积模型可以较好地解决这一问题。
  - 具体分析：
    - 因为对于$(q_i^T k_1, q_i^T k_2, ..., q_i^T k_n)$,如果某个$q_i^T k_j$相对于其他元素很大的话，那么对此向量softmax后就容易得到

一个onehot向量，不够“soft”了，而且反向传播时梯度为0会导致梯度消失；

$$y = softmax(x)$$

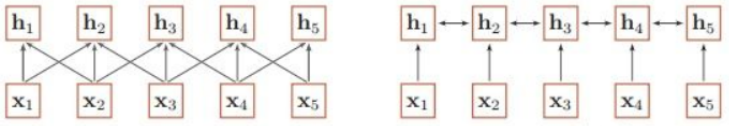
$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = diag(\boldsymbol{y}) - \boldsymbol{y}\boldsymbol{y}^T = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_2}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_1} \\ \frac{\partial y_1}{\partial x_2} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_m}{\partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_n} & \frac{\partial y_2}{\partial x_n} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

- 原论文是这么解释的：假设每个$\boldsymbol{q} \in R^d$和$\boldsymbol{k} \in R^d$的每个维度都是服从均值为0方差为1的，那么二者的内积$\boldsymbol{q}^T\boldsymbol{k}$的均值就是0，方差就是$d$，所以内积的方差和原始方差之间的比例大约是维度值$d$，为了降低内积各个维度值的方差（这样各个维度取值就在均值附近，不会存在某个维度偏离均值太远），所以要除以$\sqrt{d}$（标准差）
- **dk代表K的维度**

- self-attention 如何解决长距离依赖问题?
  - 引言:
    - 在上一个问题中，我们提到 CNN 和 RNN 在处理长序列时，都存在 长距离依赖问题，那么你是否会有这样 几个问题:
      - 长距离依赖问题 是什么呢?
      - 为什么 CNN 和 RNN 无法解决长距离依赖问题?
      - 之前提出过哪些解决方法?
      - self-attention 是如何 解决 长距离依赖问题的呢?
  - 下面，我们将会围绕着几个问题，进行一一解答。
  - 长距离依赖问题 是什么呢?
    - 介绍：对于序列问题，第 t 时刻 的 输出 y_t 依赖于 t 之前的输入，也就是 说 依赖于 $x_{t-k}, k = 1, ..., t$，当间隔 k 逐渐增大时，x_{t-k} 的信息将难以被 y_t 所学习到，也就是说，很难建立 这种 长距离依赖关系，这个也就是 长距离依赖问题（Long-Term Dependencies Problem）。
  - 为什么 CNN 和 RNN 无法解决长距离依赖问题?

- CNN:
  - 捕获信息的方式：
    - CNN 主要采用 卷积核 的 方式捕获 句子内的局部信息，你可以把他理解为 **基于 n-gram 的局部编码方式**捕获局部信息
  - 问题：
    - 因为是 n-gram 的局部编码方式，那么当 k 距离 大于 n 时，那么 y_t 将难以学习 x_{t-k} 信息；
  - 举例：
    - 其实 n-gram 类似于 人的 视觉范围，人的视觉范围在每一时刻 只能 捕获 一定 范围内 的信息，比如，你在看前面的时候，你是不可能注意到背后发生了什么，除非你转过身往后看。
- RNN:
  - 捕获信息的方式：
    - RNN 主要 通过 循环 的方式学习(记忆) 之前的信息 x_t；
  - 问题：
    - 但是随着时间 t 的推移，你会出现**梯度消失或梯度爆炸**问题，这种问题使你只能建立短距离依赖信息。
  - 举例：
    - RNN 的学习模式好比于 人类 的记忆力，人类可能会对 短距离内发生的 事情特别清楚，但是随着时间的推移，人类开始 会对 好久之前所发生的事情变得印象模糊，比如，你对小时候发生的事情，印象模糊一样。
  - 解决方法：
    - 针对该问题，后期也提出了很多 RNN 变体，比如 LSTM、 GRU，这些变体 通过引入 门控的机制 来有选择性 的记忆 一些 重要的信息，但是这种方法

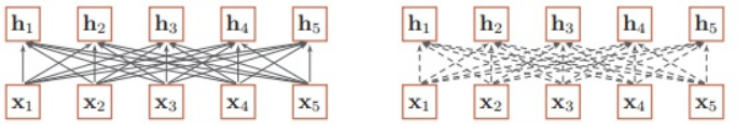
也只能在 一定程度上缓解 长距离依赖问题，但是 并不能 从根本上解决问题。



(a) 卷积网络　　(b) 双向循环网络

基于卷积网络和循环网络的变长序列编码

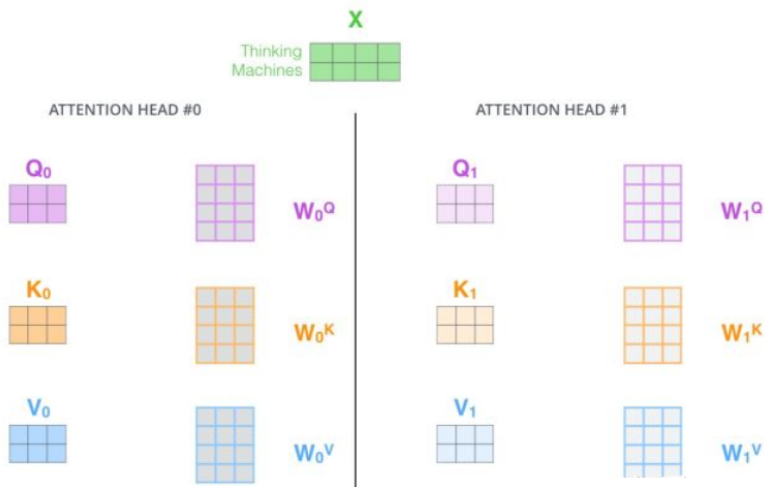
- 之前提出过哪些解决方法?
  - 引言:
    - 那么 之前 主要采用 什么方法 解决问题呢?
  - 解决方法:
    - 增加网络的层数
      - 通过一个深层网络来获取远距离的信息交互
    - 使用全连接网络
      - 通过全连接的方法对 长距离 建模;
      - 问题:
        - 无法处理变长的输入序列;
        - 不同的输入长度，其连接权重的大小也是不同的;



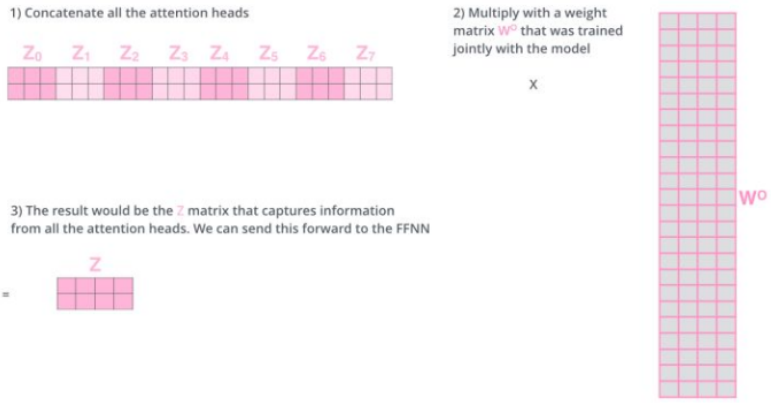
全连接模型　　自注意力模型

- **self-attention 是如何 解决 长距离依赖问题的呢?**
  - 解决方式:
    - 利用注意力机制来“动态”地生成不同连接的权重，从而处理变长的信息序列
  - 具体介绍:
    - 对于 当前query，你需要 与 句子中 所有 key 进行点乘后再 Softmax，以获得 句子中 所有 key 对于 当前 query 的 score(可以理解为 贡献度)，然后与 所有词 的 value 向量进行加权融合之后，就能使 当前 y_t 学习到句子中 其他词x_{t-k}的信息;

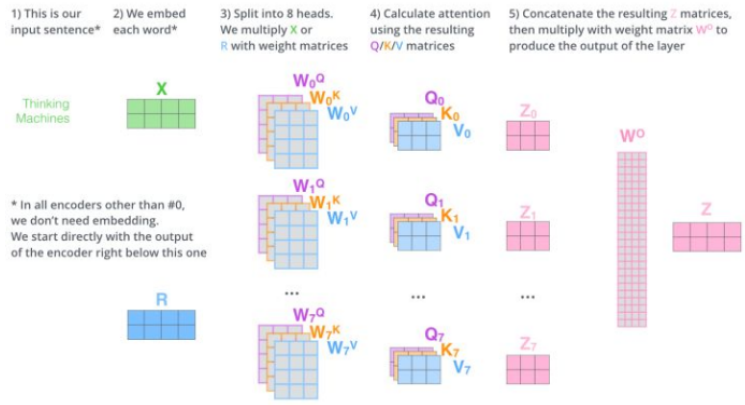
- self-attention 如何并行化?
  - 引言:
    - 在上一个问题中，我们主要讨论了 CNN 和 RNN 在处理长序列时，都存在 长距离依赖问题，以及 Transformer 是如何解决 长距离依赖问题，但是对于 RNN ,还存在另外一个问题:
      - 无法并行化问题
    - 那么，Transformer 是如何进行并行化的呢?
  - Transformer 如何进行并行化?
    - 核心：self-attention
    - 为什么 RNN 不能并行化:
      - 原因：RNN 在 计算 x_i 的时候，需要考虑到 x_1 x_{i-1} 的 信息，使得 RNN 只能 从 x_1 计算到 x_i;
    - 思路:
      - 在 self-attention 能够 并行的 计算 句子中不同 的 query，因为每个 query 之间并不存在 先后依赖关系，也使得 transformer 能够并行化;
- 为什么用双线性点积模型（即Q，K两个向量）
  - 双线性点积模型使用Q，K两个向量，而不是只用一个Q向量，这样引入非对称性，更具健壮性（Attention对角元素值不一定是最大的，也就是说当前位置对自身的注意力得分不一定最高）。
- multi-head attention 模块
- multi-head attention 的思路是什么样?
  - 思路:
    - 相当于 h 个 不同的 self-attention 的集成
    - 就是把self-attention做 n 次，取决于 head 的个数；论文里面是做了8次。
- multi-head attention 的步骤是什么样?
  - **步骤:**
    - step 1 : 初始化 N 组 $Q，K，V$矩阵(论文为 8组);



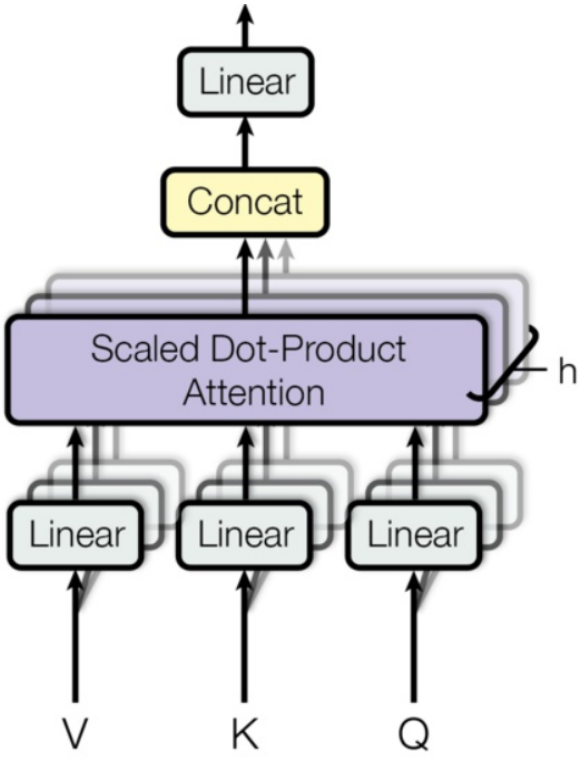
- step 2 : 每组 分别 进行 self-attention;
- step 3：
  - 问题：多个 self-attention 会得到 多个 矩阵，但是前馈神经网络没法输入8个矩阵；
  - 目标：把8个矩阵降为1个
  - 步骤：
    - 每次self-attention都会得到一个 Z 矩阵，把每个 Z 矩阵拼接起来，
    - 再乘以一个Wo矩阵，
    - 得到一个最终的矩阵，即 multi-head Attention 的结果；



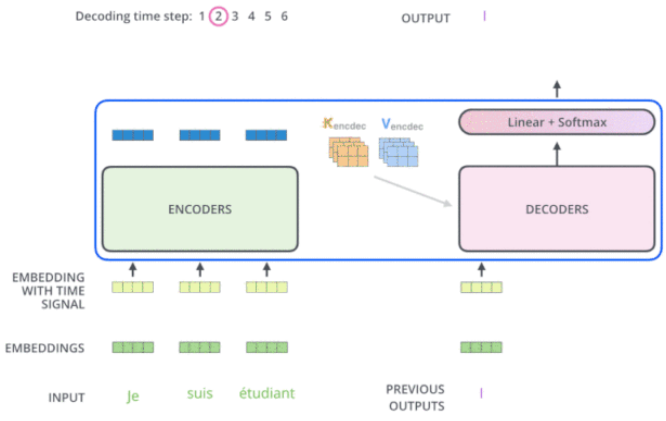
- 最后，让我们来看一下完整的流程：



- 换一种表现方式：



- 动图介绍



- Transformer为何使用多头注意力机制？（为什么不使用一个头）
- 为了让 Transformer 能够注意到不同子空间的信息，从而捕获到跟多的特征信息。【本质：实验定律】

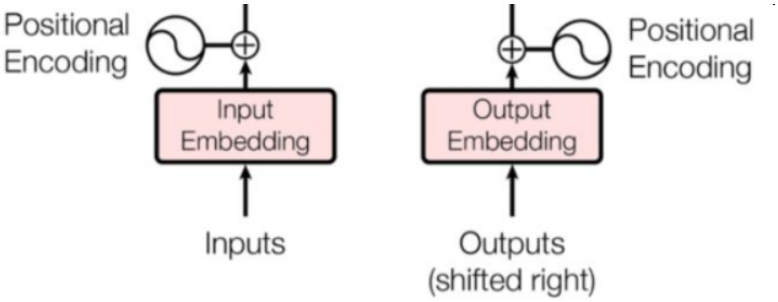
- 多头机制为什么有效?
  - 类似于CNN中通过多通道机制进行特征选择。Transformer中使用切头(split)的方法，是为了在不增加复杂度（$O(n^2 d)$）的前提下享受类似CNN中“不同卷积核”的优势。
- 为什么在进行多头注意力的时候需要对每个head进行降维?
  - Transformer的多头注意力看上去是借鉴了CNN中同一卷积层内使用多个卷积核的思想，原文中使用了 8 个“scaled dot-product attention”，在同一“multi-head attention”层中，输入均为“KQV”，同时进行注意力的计算，彼此之前参数不共享，最终将结果拼接起来，这样可以允许模型在不同的表示子空间里学习到相关的信息，在此之前的 A Structured Self-attentive Sentence Embedding 也有着类似的思想。简而言之，就是希望每个注意力头，只关注最终输出序列中一个子空间，互相独立。其核心思想在于，抽取到更加丰富的特征信息。
- multi-head attention 代码介绍
  - multi-head attention 模块代码讲解【注：代码采用 tensorflow 框架编写】

```
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
        # 初始化 Q, K, V 矩阵
        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):
        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]
        # step 1: 利用矩阵计算 q,k,v
        q = self.wq(q)
        k = self.wk(k)
        v = self.wv(v)
        # step 2:
        q = self.split_heads(q, batch_size)
        k = self.split_heads(k, batch_size)
        v = self.split_heads(v, batch_size)
        # step 3: 每组 分别 进行 self-attention
        scaled_attention, attention_weights = scaled_dot_product_attention(
            q, k, v, mask)
        # step 4: 矩阵拼接
        scaled_attention = tf.transpose(scaled_attention, perm=[0, 2, 1, 3])
        concat_attention = tf.reshape(scaled_attention, (batch_size, -1, self.d_model))
        # step 5: 全连接层
        output = self.dense(concat_attention)
        return output, attention_weights
```

- 位置编码（Position encoding）模块
- 为什么要 加入 位置编码（Position encoding） ?
  - 问题:
    - 介绍：缺乏 一种 表示 输入序列中 单词顺序 的方法

- 说明：因为模型不包括Recurrence/Convolution，因此是无法捕捉到序列顺序信息的，例如将K、V按行进行打乱，那么Attention之后的结果是一样的。但是序列信息非常重要，代表着全局的结构，因此必须将序列的分词相对或者绝对position信息利用起来z
- 目的：加入z词序信息，使 Attention 能够分辨出不同位置的词

- 位置编码（Position encoding）的作用是什么？
  - 位置向量的作用：
    - 决定当前词的位置；
    - 计算在一个句子中不同的词之间的距离
- 位置编码（Position encoding）的步骤是什么？
  - 步骤：
    - 将每个位置编号，
    - 然后每个编号对应一个向量，
    - 通过将位置向量和词向量相加，就给每个词都引入了一定的位置信息。



  - 论文的位置编码是使用三角函数去计算的。好处：
    - 值域只有[-1,1]
    - 容易计算相对位置。

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

  - Position encoding为什么选择相加而不是拼接呢？
    - 因为$[W1 W2][e; p] = W1e + W2p,W(e+p)=We+Wp$，就是说求和相当于拼接的两个权重矩阵共享（W1=W2=W），但

是这样权重共享是明显限制了表达能力的。

- Position encoding和 Position embedding的区别?
  - Position encoding 构造简单直接无需额外的学习参数；能兼容预训练阶段的最大文本长度和训练阶段的最大文本长度不一致；
  - Position embedding 构造也简单直接但是需要额外的学习参数；训练阶段的最大文本长度不能超过预训练阶段的最大文本长度（因为没学过这么长的，不知道如何表示）；但是Position embedding 的潜力在直觉上会比 Position encoding 大，因为毕竟是自己学出来的，只有自己才知道自己想要什么（前提是数据量得足够大）。
- 位置编码（Position encoding）的代码介绍
  - 位置编码（Position encoding）模块代码讲解【注：代码采用 tensorflow 框架编写】

```
# 位置编码 类
class Positional_Encoding():
    def __init__(self):
        pass
    # 功能：计算角度 函数
    def get_angles(self, position, i, d_model):
        '''
            功能：计算角度 函数
            input:
                position    单词在句子中的位置
                i           维度
                d_model     向量维度
        '''
        angle_rates = 1 / np.power(10000, (2 * (i // 2)) / np.float32(d_model))
        return position * angle_rates
    # 功能：位置编码 函数
    def positional_encoding(self, position, d_model):
        '''
            功能：位置编码 函数
            input:
                position    单词在句子中的位置
                d_model     向量维度
        '''
        angle_rads = self.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)
```

- 残差模块模块
- 为什么要 加入 残差模块?
  - 动机：因为 transformer 堆叠了 很多层，容易 梯度消失或者梯度爆炸
- Layer normalization 模块

- 为什么要 加入 Layer normalization 模块?
  - 动机：因为 transformer 堆叠了 很多层，容易 梯度消失或者梯度爆炸;
  - 原因:
    - 数据经过该网络层的作用后，不再是归一化，偏差会越来越大，所以需要将 数据 重新 做归一化处理;
  - 目的:
    - 在数据送入激活函数之前进行normalization（归一化）之前，需要将输入的信息利用 normalization 转化成均值为0方差为1的数据，避免因输入数据落在激活函数的饱和区而出现 梯度消失或者梯度爆炸 问题
- Layer normalization 模块的是什么?
  - 介绍:
    - 归一化的一种方式
    - 对每一个样本介绍均值和方差【这个与 BN 有所不同，因为他是在 批方向上 计算均值和方差】
- Batch normalization 和 Layer normalization 的区别?
  - BN 计算公式

$$BN(x_i) = \alpha \times \frac{x_i - \mu_b}{\sqrt{\sigma_B^2 + \epsilon}} + \beta$$

  - LN 计算公式

$$LN(x_i) = \alpha \times \frac{x_i - \mu_L}{\sqrt{\sigma_L^2 + \epsilon}} + \beta$$

  - Batch normalization — 为每一个小batch计算每一层的平均值和方差
  - Layer normalization — 独立计算每一层每一个样本的均值和方差

- Transformer 中为什么要舍弃 Batch normalization 改用 Layer normalization 呢?
  - Transformer代码+面试细节 - 知乎 (zhihu.com)
  - 原始BN是为CNN而设计的，对整个batchsize范围内的数据进行考虑;
  - **对于RNN来说，sequence的长度是不一致的，所以用很多padding来表示无意义的信息。如果用 BN 会导致有意义的 embedding 损失信息。**
  - 所以，BN一般用于CNN，而LN用于RNN。
  - layernorm是**在hidden size的维度进行**的，**跟batch和seq_len无关**。每个hidden state都计算自己的均值和方差，这是因为不同hidden state的量纲不一样。beta和gamma的维度都是(hidden_size,)，经过白化的hidden state * beta + gamma得到最后的结果。
  - **LN在BERT中主要起到白化的作用，增强模型稳定性（如果删除则无法收敛）**
- Layer normalization 模块代码介绍
  - Layer normalization 模块代码讲解【注：代码采用 tensorflow 框架编写】

```
class EncoderLayer(tf.keras.layers.Layer):
    def __init__(self, d_model, num_heads, dff, rate=0.1):
        ...
        self.layernorm1 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
        self.layernorm2 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
        ...

    def call(self, x, training, mask):
        ...
        # step 3: Layer Norm1
        out1 = self.layernorm1(x + attn_output)
        # step 4: 前馈网络
        ffn_output = self.ffn(out1)
        ffn_output = self.dropout2(ffn_output, training=training)
        # step 5: Layer Norm1
        out2 = self.layernorm2(out1 + ffn_output)

        return out2
```

- Mask 模块
- 什么是 Mask?
  - 介绍：掩盖某些值的信息，让模型信息不到该信息；
- Transformer 中用到 几种 Mask?
  - 答案：两种
  - 类别：padding mask and sequence mask

- 能不能介绍一下 Transformer 中用到几种 Mask?
  - **padding mask**：
    - 作用域：每一个 scaled dot-product attention 中
    - 动机：输入句子的长度不一问题
    - 方法：
      - 短句子：后面 采用 0 填充
      - 长句子：只截取 左边 部分内容，其他的丢弃
    - 原因：对于 填充 的位置，其所包含的信息量 对于 模型学习 作用不大，所以 self-attention 应该 抛弃对这些位置进行学习；
    - 做法：在这些位置上加上 一个 非常大 的负数（负无穷），使 该位置的值经过 Softmax 后，值近似 0，利用 padding mask 标记哪些值需要做处理；
    - 实现：

```
# 功能：  padding mask
def create_padding_mask(seq):
    '''
        功能：  padding mask
        input:
            seq     序列
    '''
    seq = tf.cast(tf.math.equal(seq, 0), tf.float32)
    return seq[:, tf.newaxis, tf.newaxis, :]
```

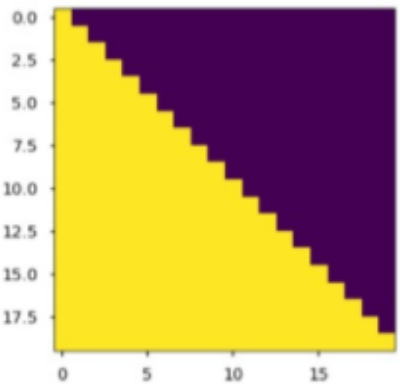
  - **sequence mask**
    - 作用域：只作用于 decoder 的 self-attention 中
    - 动机：不可预测性；
    - 目标：sequence mask 是为了使得 decoder 不能看见未来的信息。也就是对于一个序列，在 time_step 为 t 的时刻，我们的解码输出应该只能依赖于 t 时刻之前的输出，而不能依赖 t 之后的输出。因此我们需要想一个办法，把 t 之后的信息给隐藏起来。
    - 做法：
      - 产生一个下三角矩阵，上三角的值全为0，下三角全是 1。把这个矩阵作用在每一个序列上，就可以达到我们的目的

$$XW_i^Q = Q_i,\ XW_i^K = K_i,\ XW_i^V = V_i,\ i = 1, \ldots, 8$$
$$Z_i = Attention(Q_i, K_i, V_i)_i = \text{softmax}(Mask(\frac{Q_i K_i^T}{\sqrt{d_k}}))V_i,\ i = 1, \ldots, 8$$
$$Z = \text{MultiHead}(Q, K, V) = \text{Concat}(Z_1, \ldots, Z_8)W^O$$
$$Z = \text{LayerNorm}(X + Z)$$

- sequence mask 公式

$$\frac{Q_i K_i^T}{\sqrt{d_k}} = \begin{bmatrix} \alpha_{00} & \alpha_{01} & \cdots & \alpha_{0T_y} \\ \cdots & \cdots & \cdots & \cdots \\ \alpha_{T_y 0} & \alpha_{T_y 1} & \cdots & \alpha_{T_y T_y} \end{bmatrix}$$

- 注意力矩阵， 每个元素 a_{ij} 代表 第 i 个词和第 j 个词的内积相似度



- 下三角矩阵，上三角的值全为0，下三角全是 1
  - 注：在 decoder 的 scaled dot-product attention 中，里面的 attn_mask = padding mask + sequence mask
  - 在 encoder 的 scaled dot-product attention 中，里面的 attn_mask = padding mask

- Feed forward network (FFN)
- Feed forward network (FFN)的作用?
  - 答：Transformer在抛弃了 LSTM 结构后，FFN 中的激活函数成为了一个主要的提供**非线性变换的单元。**
- GELU
- GELU原理？相比RELU的优点?
  - ReLU会确定性的将输入乘上一个0或者1(当x<0时乘上0，否则乘上1)；
  - Dropout则是随机乘上0；

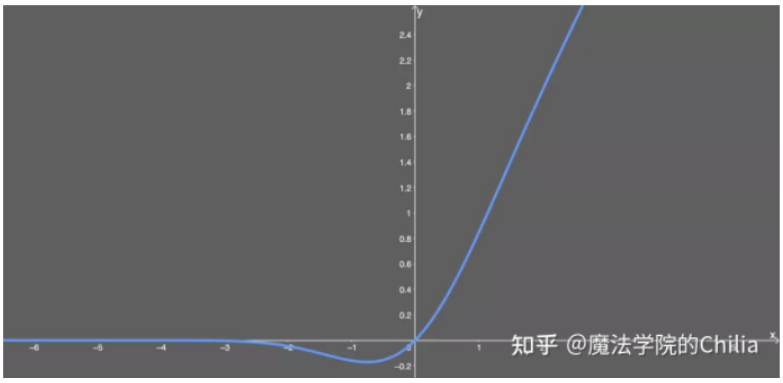
- **GELU虽然也是将输入乘上0或1，但是输入到底是乘以0还是1，是在取决于输入自身的情况下随机选择的。**
- 具体说明：
- 我们将神经元的输入 x 乘上一个服从伯努利分布的 m 。而该伯努利分布又是依赖于 x 的：

$$m \sim Bernoulli(\Phi(x)),\ where\ \Phi(x) = P(X <= x)$$

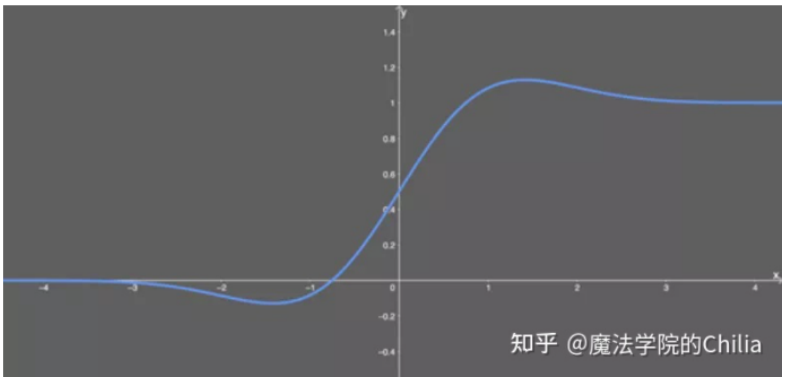
- 其中， X~N(0,1)，那么 φ(x) 就是标准正态分布的累积分布函数。这么做的原因是因为神经元的输入 x 往往遵循正态分布，尤其是深度网络中普遍存在Batch Normalization的情况下。当x减小时， φ(x) 的值也会减小，此时x被“丢弃”的可能性更高。所以说这是随机依赖于输入的方式。
- 现在，给出GELU函数的形式：

$$GELU(x) = \Phi(x) * I(x) + (1 - \Phi(x)) * 0x = x\Phi(x)$$

- 其中 φ(x) 是上文提到的标准正态分布的累积分布函数。因为这个函数没有解析解，所以要用近似函数来表示。



- 其导数图像如下



- **所以，GELU的优点就是在ReLU上增加随机因素，x越小越容易被mask掉。**

- Transformer的非线性来自于哪里?
  - FFN的gelu激活函数和self-attention，注意self-attention是非线性的（因为有相乘和softmax）。

以上内容整理于 幕布文档