保姆级教程：Transformer本质是什么，图解代码细节全析

Original Pulsar planet Tim在路上 2024-04-14 20:01 北京

Tim在路上
大模型、Spark、LakeHouse 欢迎关注知乎账号"Tim在路上"
132篇原创内容

公众号

今天我们再次尝试聊一下Transformer，探索和描述下其本质特性，纯属胡闹，如有偏差请亲喷。

如果要在物理学众多定律中选出一个最具代表性的，熵增定律无疑是一个强有力的候选者。它揭示了一个不可逆的宇宙真理：在一个封闭系统中，熵，即系统的混乱度或无序度，总是倾向于增加。

我们生活环境就是一个熵增的环境，随着时间的推移，无论是山川河流，还是人类社会，都在发生着变化。也许时间不可逆转，其本质就是由于$\triangle S>0$这条铁律。

与此同时，熵的增加也意味着信息的增加，也代表了一个系统的复杂度和预测难度的增加。

而生命体或智能体却在这个不断增加熵的宇宙中展现出了一种独特的逆流而上的能力——学习。学习其本质就是通过摄取和转换能量，使得其在局部范围内实现减熵。使得其内部变得更加有序，从环境中学习到某种规律，认识到关于世界的本质信息，以此来抵御和逆转自然界的熵增趋势。

不过当我们尝试解释世界模型或复杂系统时，可能会遇到难以清晰表达的挑战。所以让我们回到更具体的例子，比如Transformer模型在序列到序列的语言翻译任务中的应用。

接下来，我们将以Transformer具体框架为脉络进行分析，同时我会提供些代码来帮助初学者更清晰的认识Transformer。

0. NN Essentially

在神经网络里面，计算主要是矩阵操作和激活函数操作，还有其他一些计算，如损失函数的计算、优化算法的更新等，但它们的计算量通常相对较小。

Table 1. Proportions for operator classes in PyTorch.

Operator class	% flop	% Runtime
△ Tensor contraction	99.80	61.0
□ Stat. normalization	0.17	25.5
○ Element-wise	0.03	13.5

上图为BERT论文中不同运算符类型的 FLOP 计数表，其中“Tensor contraction”= matmuls就是矩阵乘运算。

在几何线性代数里，我们可以理解矩阵实际上代表的是新旧坐标系之间的关系，我们可以设定一个矩阵的行代表旧坐标系有多少维度，列代表新坐标系有多少维度。一个矩阵的乘法实际代表其表示向量的旋转拉伸收缩的变化，矩阵的加法代表着向量的平移操作。

所以说神经网络能够实现对输入空间的线性变换，使得原本线性不可分的问题变得线性可分，而增加隐藏层可以进一步提高分类效果，是因为隐藏层的加入实质上进行了空间变换，神经网络可以学习到数据中的特征和模式，并进行有效的分类、识别或预测。

1. Embedding

从Transformer框架上我们可以看出，不论Encode部分还是Decode部分，其输入都是Embedding。

Embedding 这个概念在深度学习领域最原初的切入点是所谓的**Manifold Hypothesis**（流形假设）。流形假设是指“自然的原始数据是低维的流形嵌入于(embedded in)原始数据所在的高维空间”。

那么，深度学习的任务就是把高维原始数据（图像，句子）映射到低维流形，使得高维的原始数据被映射到低维流形之后变得可分，而这个映射就叫嵌入（Embedding）。

如果我们要完成一个机器翻译的任务，我们首要的任务就是需要让机器能够认识到输入的句子。但由于计算的基础是数，而自然语言是文字，因此很容易想到要做的第一步是让文字数字化，这个过程就是编码。

其次，在我们对每一个词进行编码时，需要注意几点：

1. 每个词需要对应唯一的数值编码；

2. 词义相近词需要有"相近"的数值编码值；

3. 词义具有多维性，即一词多义；

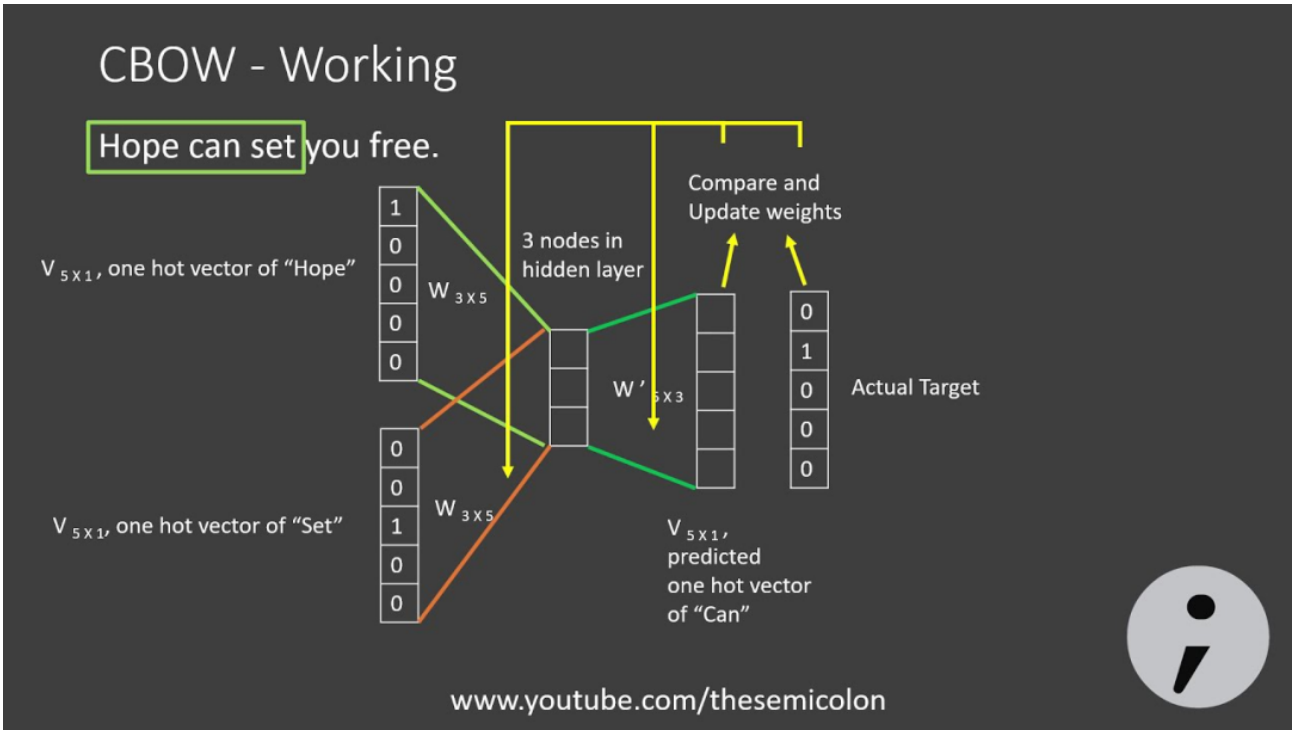
所以为了满足以上几点，我们很自然会想到使用**向量**来进行编码。对于每一个词，我们可以表达为一组数，而非一个数；这样一来，就可以在不同的维度上定义远近，词与词之间复杂的关系便能在这一高维的空间中得到表达。

既然我们知道要能使机器理解语言，需要用数值向量（或者说矩阵）来进行编码。那么如何实现这个编码呢？

一个比较容易想到的方法是，令词义的不同维度和向量不同维度进行关联。例如我们可以通过，名词、动词、形容词、数字、人物、主动、被动等等维度来描述一个词。但由于人类很难将词不同维度全部列举出来，其次不可能对每个词的不同维度赋予有效的数值。

所以我们只能靠神经网络“大力砖飞”的方式，Google在2013年提出了Word2Vec就是一个其中的方法。

Word2Vec 的训练模型本质上是只具有一个隐藏层的神经元网络，而我们要获取的就是其中训练好的隐向量。



Word2Vec 的关键是一个重要的洞察、一个极具启发性的角度：**一个词的意义，可以被它所出现的上下文定义。**下面我们来看Word2Vec是如何训练的。

例如在CBOW中，对于每个目标词，我们将其上下文单词转换为 One-Hot 编码词向量输入到模型，在上图中我们上下文窗口设置为2，当目标词是 "can" 时，上下文单词是 "Hope" 和 "set"。

然后，我们将输入的向量相加并求平均，得到隐藏层向量；将隐藏层向量与权重矩阵W’相乘，通过 softmax 函数得到每个单词的概率分布。

通过预测“can”的One-Hot 编码，并计算损失，随后通过梯度下降算法更新权重矩阵，使得损失最小化。

在经过多次迭代训练后，模型将学习到每个单词的词向量。

通过上面的流程我们可以看出，Word2Vec提供了模型对语义的最初理解，它训练完成之后体现的是单个Token之间的联系，起到了一个“编词典“的作用。

我们可以把Word2Vec理解为，用训练预料中的其他token来解释当前目标token的意思。

这个最终而形成的这个词是固定的，他不会随着句子的语义进行变化。

我们通过keras简化api来实现将token转换为Embedding的过程，代码如下：

```
class TokenEmbedding(tf.keras.layers.Layer):
    def __init__(self, vocab_size: int, embedding_dim: int, dtype=tf.float32, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.vocab_size = vocab_size
        self.embedding_dim = embedding_dim
        self.dtype_ = dtype

    def build(self, input_shape: tf.TensorShape) -> None:
        self.lookup_table = self.add_variable(
            name='token_embedding',
            shape=[self.vocab_size, self.embedding_dim],
            dtype=self.dtype_,
            initializer=tf.random_normal_initializer(0., self.embedding_dim ** -0.5),
        )
        super().build(input_shape)

    def call(self, input: tf.Tensor) -> tf.Tensor:
        mask = tf.to_float(tf.not_equal(input, PAD_ID))
        embedding = tf.nn.embedding_lookup(self.lookup_table, input)
        embedding *= tf.expand_dims(mask, -1)
        return embedding * self.embedding_dim ** 0.5
```

2. Attention

在Transformer之前，深度神经网络主要是CNN和RNN。CNN擅长处理具有固定尺寸的图像和视频数据，而RNN则能够应对长度可变的文本和时间序列数据。

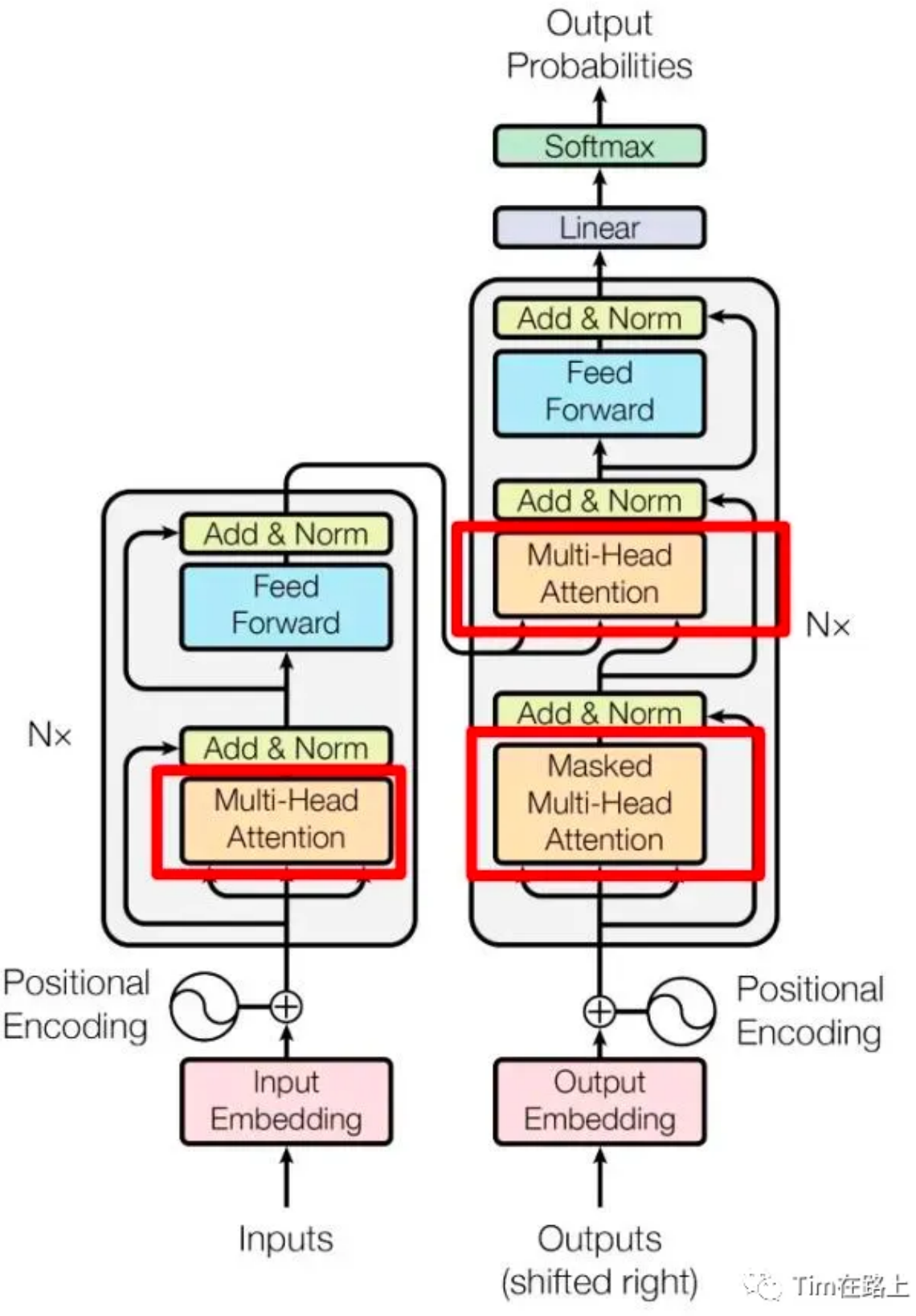
为了进一步提升序列数据的处理能力，研究者们设计了编码器-解码器（Encoder-Decoder）架构，它由两个RNN组成，并通过一个上下文向量来捕捉输入序列的全局信息。但这种方法存在局限，因为它依赖于一个固定的上下文向量来概括整个序列的语义。

为了突破这一瓶颈，注意力机制（Attention）应运而生。它允许模型灵活地聚焦于输入序列中的关键部分，从而更精准地提取相关信息。紧接着，自注意力（Self-Attention）机制的提出进一步革新了模型的表达能力，它通过多头注意力（Multi-Head Attention）的方式，使模型能够同时关注序列中的多个位置，极大地增强了模型对复杂数据结构的理解力和学习效率。

可以说，Transformer是融合了CNN和RNN。

1. 并且与RNN相比，Attention机制能够更快速地学习，因为它可以并行处理整个序列。这使得在GPU上的训练效率大大提高，通常保持在98%左右的利用率，而传统RNN模型则平均在60%左右。
2. Attention机制的结构相对简单，不需要处理RNN中的序列数据传递和隐藏状态更新，降低了模型的复杂性，避免了由于长期依赖导致的梯度消失或梯度爆炸问题。

下面我们来实现一个简单的Attention来了解Transformer原理。



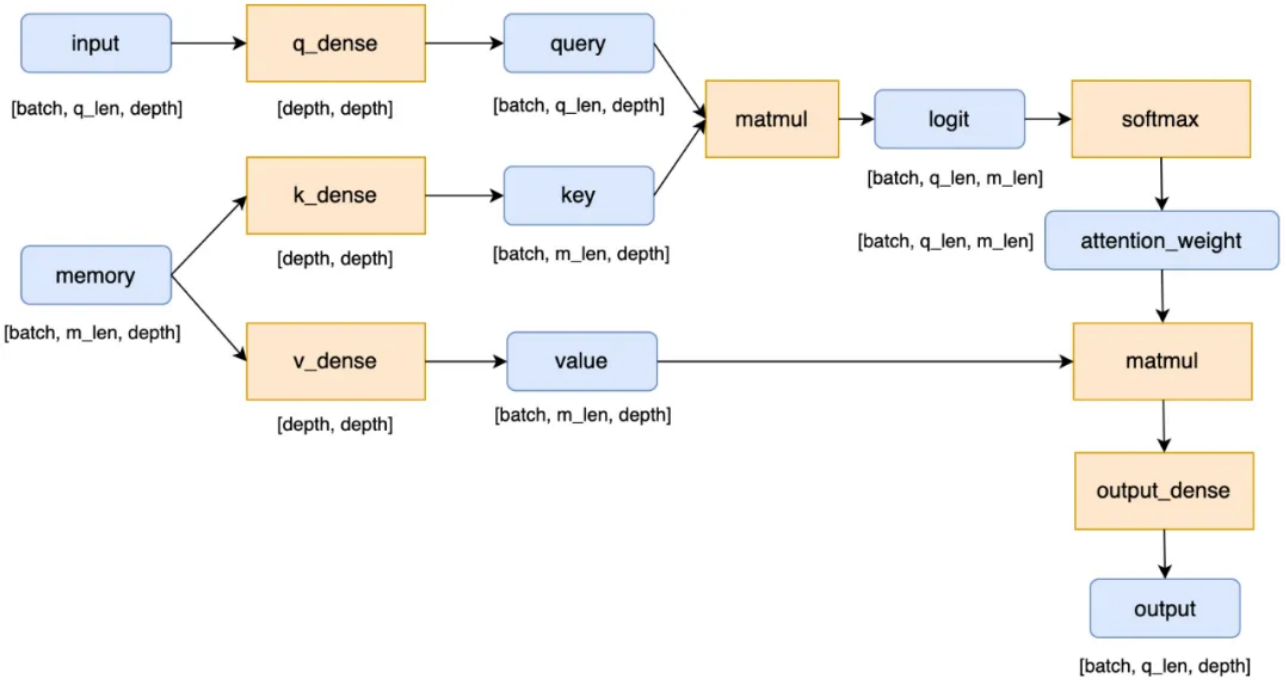
Attention的基础是查询（Query）和存储（Key, Value），就是用来处理多个Token组合后的语义。

我们可以简单理解为，注意力Attention是指使用Query有选择地从存储（Key, Value）中提取必要的信息。当从存储中检索信息时，query 通过 key 来确定要检索哪个存储并检索对应的值Value。

下面我们创建一个简化版的Attention来进行分析讲解，代码如下所示：

```
class SimpleAttention(tf.keras.models.Model):

    def __init__(self, depth: int, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.depth = depth

        self.q_dense_layer = tf.keras.layers.Dense(depth, use_bias=False, name='q_dense_layer')
        self.k_dense_layer = tf.keras.layers.Dense(depth, use_bias=False, name='k_dense_layer')
        self.v_dense_layer = tf.keras.layers.Dense(depth, use_bias=False, name='v_dense_layer')
        self.output_dense_layer = tf.keras.layers.Dense(depth, use_bias=False, name='output_dense_

    def call(self, input: tf.Tensor, memory: tf.Tensor) -> tf.Tensor:
        q = self.q_dense_layer(input)  # [batch_size, q_length, dim]
        k = self.k_dense_layer(memory)  # [batch_size, m_length, dim]
        v = self.v_dense_layer(memory)

        logit = tf.matmul(q, k, transpose_b=True)  # [batch_size, q_length, k_length]

        attention_weight = tf.nn.softmax(logit, name='attention_weight')

        attention_output = tf.matmul(attention_weight, v)  # [batch_size, q_length, dim]
        return self.output_dense_layer(attention_output)
```

我们可以将这段代码简化为如下的图示：



在上图中我们可以看出，batch代表为batch size, q_len表示为句子中token的长度，depth表示token Embedding的维度。

首先，如图所示，我们通过dense_layer将input和memory转换为query, key和value。

然后，我们将query和key的转置进行内积，它表示的为query和key中token计算相关度，也可以称为注意力得分。

具体来说，由于它是矩阵相乘，如果我们把矩阵看做为向量，则表示query向量和key向量的相关性，如果向量的指向方向相似，则该值会很大。

其次，在获得矩阵乘积后，我们通过应用 softmax 对其进行归一化，以使每个查询的总权重为 1.0，这意味着我们将相关性转换为注意这里是按行进行归一化，然后我们得到了attention_weight的矩阵。

最后，我们通过attention_weight与value的乘积来提取value的信息，使用dense层进行变换作为输出。

那么这些计算到底有什么意义呢？

在注意力机制中，输入是一组词（也称为token）向量。这些词向量经过嵌入（Embedding）处理，已经具备了一定的基础语义，但这种语义是受到限制的。

在进行注意力计算之前，输入token的词向量是预先训练好的，但它们可能不会与当前语句中的词产生直接的关联。然而，在语言中，不同词之间的组合可能会产生新的或不同的意义。换句话说，我们需要根据当前语句的上下文将已定义的token的基础语义调整为适应上下文的主观语义。这意味着我们需要通过上下文来调整基础语义，使其更符合当前语境的含义。

Q和K的转置进行相乘，其实质可以理解为Q和K中每个token的相关性，而经过softmax后这些相关性转换为0~1之间的数值，然后再将其乘以上V，可以理解为通过当前语句中其他token的相关性权重对V进行的修正。

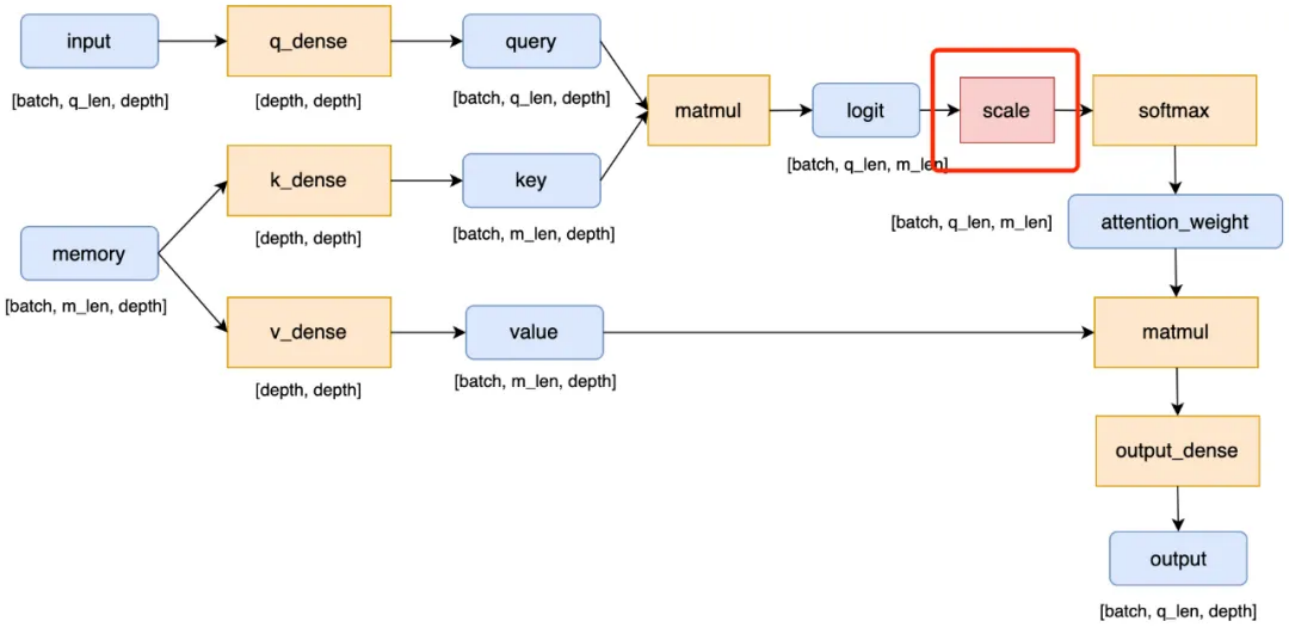
需要注意的是在自注意力(self-attention)机制中Q，K和V都是相同的Tensor。

```
attention_layer = SimpleAttention(depth=128)

x: tf.Tensor = ...
attention_output = attention_layer(input=x, memory=x)
```

3. Scale

在Transformer论文中定义的self-attention机制不仅包括上述提到的结构，还有一些其他关键机制，其中之一就是缩放（scale）。

在self-attention中，为了计算注意力权重，我们会对每个词对应的query向量和所有词对应的key向量进行点积操作，然后进行softmax归一化，最后将注意力权重与value向量相乘并加权求和得到输出。而缩放则是为了缓解点积计算的数值稳定性问题。



对于Softmax来说，如果logit的值较大，则值会重叠，梯度会接近于0。

Softmax的logit输入是Q和K的矩阵乘积，因此Q和K的维度（depth）越大，logit就越大，所以我们应该根据Q和K的维度（depth）来调整其数值。

那么为什么要除以 $\sqrt{depth}$ ，这其实是从概率分布的角度来看的。

假设Q和K中的向量数据都符合标准正态分布，即期望为0，方差为1。在进行点积计算时，由于每个维度都是独立的，Q中的一行乘以K中的一列的结果的期望值依然为0，但其方差为向量维度depth。

通过除以根号depth，即除以标准差，我们可以将点积计算的结果进行缩放，使其方差恢复为1。这样做的好处是，使得点积计算的结果更加稳定，更接近于标准正态分布。

因此，根据向量维度减少Q的大小。

$$attention\_weight = softmax\left(\frac{qk^T}{\sqrt{depth}}\right)$$

其代码为：

```
q *= depth ** -0.5  # scaled dot-product
logit = tf.matmul(q, k, transpose_b=True)  # [batch_size, q_length, k_length]

attention_weight = tf.nn.softmax(logit, name='attention_weight')
```

这里仅对Q进行缩放其结果是等价的。

4. Mask

在Transformer Decoder中，存在Masked Self-Attention，用于将特定键的注意力权重（attention weights）设置为0，从而在模型中限制信息的传递。

在实际的代码中，我们通常会将要屏蔽的元素的logits设置为负无穷（-∞），这样在进行指数运算时，它们对应的概率会趋近于0。softmax公式如下：

$softmax_i = \frac{e^{x_i}}{\sum_j e^{x_j}}$，如果我想屏蔽它，我们 x_i 变成-∞，结果就为0。

其代码实现如下所示：

```
logit = tf.matmul(q, k, transpose_b=True)  # [batch_size, head_num, q_length, k_length]
logit += tf.to_float(attention_mask) * input.dtype.min

attention_weight = tf.nn.softmax(logit, name='attention_weight')
```

在推理过程中，解码器是自回归地生成输出序列的，即根据特定时间 t 的输入预测 t+1，并将该预测传递给下一个时间步的输入，然后继续预测之后的时间步。每个时间步的预测都是基于当前时间步之前的输入和预测结果得出的。

然而，在训练学习的过程中，模型需要同时预测整个序列，以便计算损失并进行反向传播更新参数。这时就会出现一个问题：如果Self-Attention能够访问未来的信息，那么模型就可能在训练时使用了未来的信息来进行预测，从而导致了信息泄漏或模型过拟合的问题。

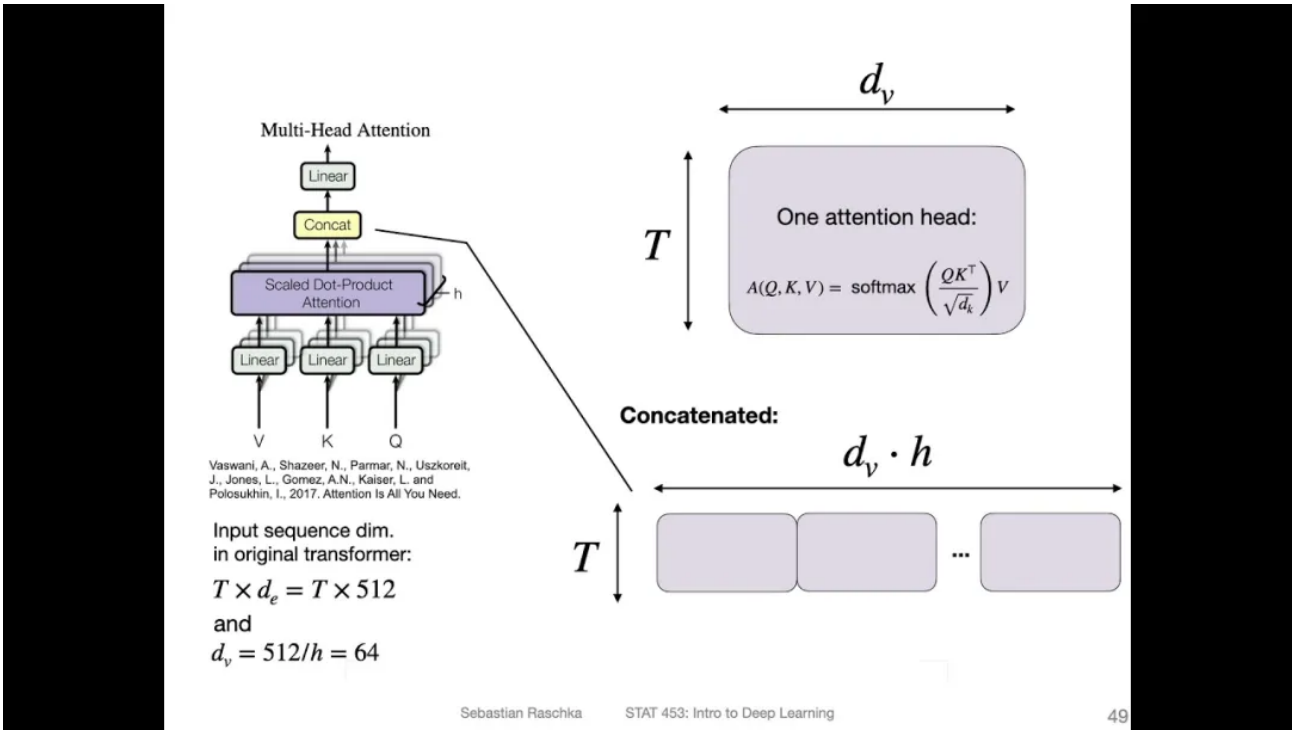
为了解决这个问题，Self-Attention机制必须确保查询（query）无法引用内存中超前于其自身时间的信息。

换句话说，在每个时间步，查询只能访问当前时间步及之前的信息，而不能访问未来的信息。这可以通过在Self-Attention中应用适当的masking机制来实现，以确保模型在训练过程中不会访问未来的信息。

5. Multi-head Attention

多头注意力（Multi-Head Attention）它将注意力机制分解成多个并行的注意力头（attention heads）。每个注意力头都会对输入进行注意力计算，然后将它们的输出合并在一起，从而获得最终的多头注意力输出。

论文中提到，将注意力机制划分为多个小头而不是执行一个大的注意力的好处在于，多头注意力可以并行计算，从而加速模型的训练和推理过程。此外，每个注意力头可以关注不同的部分，有助于提高模型的精度和鲁棒性。



多头注意力的工作原理也非常简单：首先将Q、K和V拆分为 head_num 块，每个注意力头分别计算每个块的注意力，最后连接在一起。需要注意的是，在实际应用中，经常会在注意力模块中引入 Dropout，这有助于提高模型的泛化性能，减少过拟合。

具体的代码如下所示：

```
class MultiheadAttention(tf.keras.models.Model):
    '''
    model = MultiheadAttention(
        hidden_dim=512,
        head_num=8,
        dropout_rate=0.1,
    )
    model(query, memory, mask, training=True)
    '''

    def __init__(self, hidden_dim: int, head_num: int, dropout_rate: float, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.hidden_dim = hidden_dim
        self.head_num = head_num
        self.dropout_rate = dropout_rate

        self.q_dense_layer = tf.keras.layers.Dense(hidden_dim, use_bias=False, name='q_dense_layer
        self.k_dense_layer = tf.keras.layers.Dense(hidden_dim, use_bias=False, name='k_dense_layer
        self.v_dense_layer = tf.keras.layers.Dense(hidden_dim, use_bias=False, name='v_dense_layer
        self.output_dense_layer = tf.keras.layers.Dense(hidden_dim, use_bias=False, name='output_d
        self.attention_dropout_layer = tf.keras.layers.Dropout(dropout_rate)

    def call(
            self,
            input: tf.Tensor,
            memory: tf.Tensor,
            attention_mask: tf.Tensor,
```

```
            training: bool,
    ) -> tf.Tensor:
        q = self.q_dense_layer(input)  # [batch_size, q_length, hidden_dim]
        k = self.k_dense_layer(memory)  # [batch_size, m_length, hidden_dim]
        v = self.v_dense_layer(memory)

        q = self._split_head(q)  # [batch_size, head_num, q_length, hidden_dim/head_num]
        k = self._split_head(k)  # [batch_size, head_num, m_length, hidden_dim/head_num]
        v = self._split_head(v)  # [batch_size, head_num, m_length, hidden_dim/head_num]

        depth = self.hidden_dim // self.head_num
        q *= depth ** -0.5  # for scaled dot production
        logit = tf.matmul(q, k, transpose_b=True)  # [batch_size, head_num, q_length, k_length]
        logit += tf.to_float(attention_mask) * input.dtype.min

        attention_weight = tf.nn.softmax(logit, name='attention_weight')
        attention_weight = self.attention_dropout_layer(attention_weight, training=training)

        attention_output = tf.matmul(attention_weight, v)  # [batch_size, head_num, q_length, hidc
        attention_output = self._combine_head(attention_output)  # [batch_size, q_length, hidden_c
        return self.output_dense_layer(attention_output)

    def _split_head(self, x: tf.Tensor) -> tf.Tensor:
        with tf.name_scope('split_head'):
            batch_size, length, hidden_dim = tf.unstack(tf.shape(x))
            x = tf.reshape(x, [batch_size, length, self.head_num, self.hidden_dim // self.head_num
            return tf.transpose(x, [0, 2, 1, 3])

    def _combine_head(self, x: tf.Tensor) -> tf.Tensor:
        with tf.name_scope('combine_head'):
            batch_size, _, length, _ = tf.unstack(tf.shape(x))
            x = tf.transpose(x, [0, 2, 1, 3])
            return tf.reshape(x, [batch_size, length, self.hidden_dim])
```

值得注意的是，与传统的循环神经网络（RNN）不同，多头注意力机制中的每个注意力头都具有独立的权重。

这意味着每个头可以关注输入序列的不同部分，并独立地学习到不同的信息表示。相比之下，传统的 RNN 模型在每个时间步都使用相同的权重，因此可能无法有效地捕捉到输入序列中的复杂结构和长距离依赖关系。

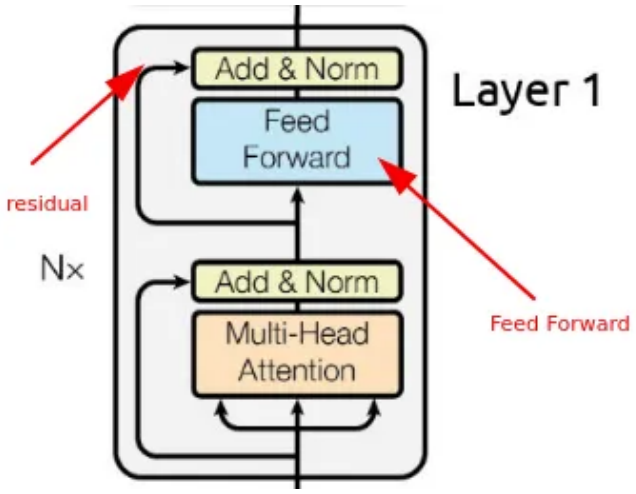
在有了多头注意力后，我们的自注意力的代码可以采用继承多头注意力的方式实现：

```
class SelfAttention(MultiheadAttention):
    def call(  # type: ignore
            self,
            input: tf.Tensor,
            attention_mask: tf.Tensor,
            training: bool,
    ) -> tf.Tensor:
        return super().call(
            input=input,
            memory=input,
```

```
            attention_mask=attention_mask,
            training=training,
        )
```

6. FFN

在Transformer中，每个多头注意力模块后面插入一个前馈神经网络（FFN）。这个 FFN 会对多头注意力的输出进行进一步的处理。



具体来说，这个 FFN 由两个全连接层组成：第一层的维度是 hidden_dim * 4，并且使用 ReLU 激活函数；第二层的维度是 hidden_dim，并且使用线性激活函数。

这种结构的作用是通过非线性变换来增强模型的表达能力。通过引入 ReLU 非线性激活函数，FFN 可以学习到输入数据的更复杂的特征表示，从而提高模型的判别能力。而线性激活函数则保留了一些输入的线性关系，有助于维持模型的稳定性。

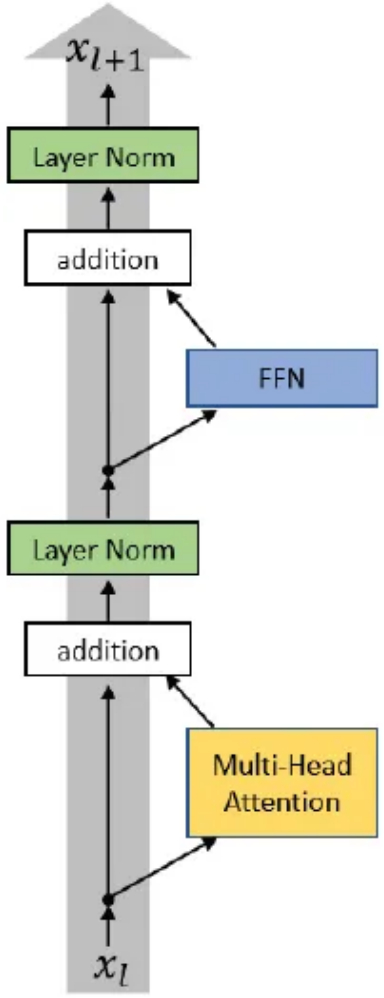
其代码如下所示：

```
class FeedForwardNetwork(tf.keras.models.Model):
    def __init__(self, hidden_dim: int, dropout_rate: float, *args, **kwargs) -> None:
        super().__init__(*args, **kwargs)
        self.hidden_dim = hidden_dim
        self.dropout_rate = dropout_rate

        self.filter_dense_layer = tf.keras.layers.Dense(hidden_dim * 4, use_bias=True,
                                                        activation=tf.nn.relu, name='filter_layer'
        self.output_dense_layer = tf.keras.layers.Dense(hidden_dim, use_bias=True, name='output_la
        self.dropout_layer = tf.keras.layers.Dropout(dropout_rate)

    def call(self, input: tf.Tensor, training: bool) -> tf.Tensor:
        tensor = self.filter_dense_layer(input)
        tensor = self.dropout_layer(tensor, training=training)
        return self.output_dense_layer(tensor)
```

7. LayerNorm

Batch Normalization（BN）在深度学习的图像处理任务中声名鹊起，它有效地对每一层的激活进行规范化，覆盖整个批次。这种规范化技术最初是为了减轻内部协变量偏移的问题，从而促进深度神经网络的更快、更稳定的训练。

然而，当涉及到Transformer架构，比如自然语言处理任务中使用的模型时，通常不会采用Batch Normalization。这主要是由于序列数据的固有特性，输入序列的长度可以有很大的变化。在这种情况下使用Batch Normalization可能会面临挑战，因为它需要聚合整个批次的统计信息，这可能会导致效率低下，特别是在处理长度不同的序列时。



Post-LN Transformer

$$
\begin{aligned}
x_{l,i}^{post,1} &= \text{MultiHeadAtt}(x_{l,i}^{post}, [x_{l,1}^{post}, \cdots, x_{l,n}^{post}]) \\
x_{l,i}^{post,2} &= x_{l,i}^{post} + x_{l,i}^{post,1} \\
x_{l,i}^{post,3} &= \text{LayerNorm}(x_{l,i}^{post,2}) \\
x_{l,i}^{post,4} &= \text{ReLU}(x_{l,i}^{post,3} W^{1,l} + b^{1,l}) W^{2,l} + b^{2,l} \\
x_{l,i}^{post,5} &= x_{l,i}^{post,3} + x_{l,i}^{post,4} \\
x_{l+1,i}^{post} &= \text{LayerNorm}(x_{l,i}^{post,5})
\end{aligned}
$$

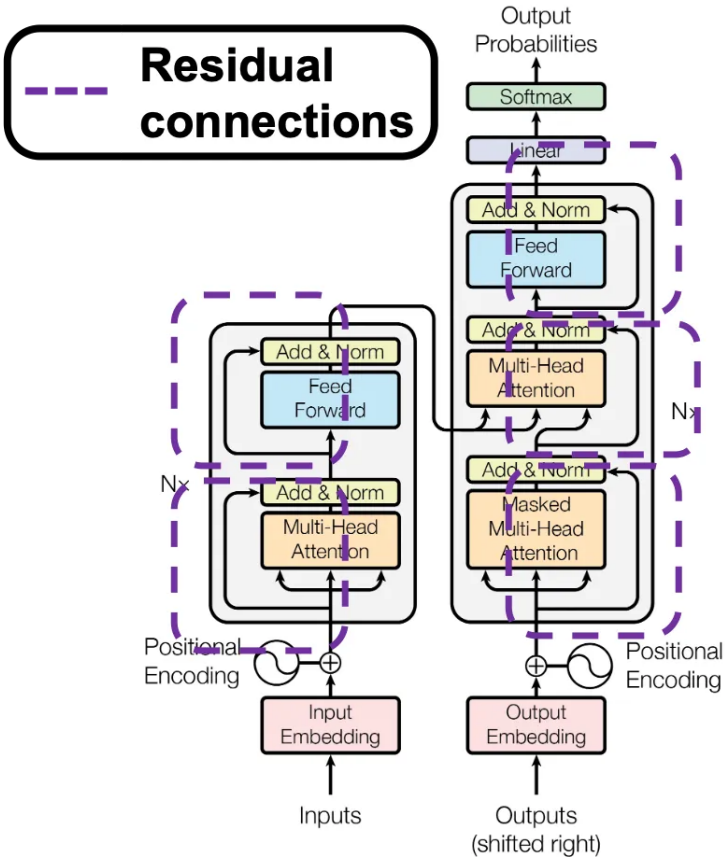
相反，Transformer模型通常依赖于Layer Normalization（LN）。Layer Normalization独立地沿着特征维度对每一层的激活进行规范化，使其更适用于序列模型。通过将规范化过程与批次统计信息分离，Layer Normalization确保了在不同长度的序列中稳定一致的性能。此外，它促进了高效的训练和推理，因为它不依赖于批次统计信息，并且可以独立应用于网络的每一层。

代码如下所示：

```
class LayerNormalization(tf.keras.layers.Layer):
    def build(self, input_shape: tf.TensorShape) -> None:
        hidden_dim = input_shape[-1]
        self.scale = self.add_weight('layer_norm_scale', shape=[hidden_dim],
                                     initializer=tf.ones_initializer())
        self.bias = self.add_weight('layer_norm_bias', [hidden_dim],
                                    initializer=tf.zeros_initializer())
        super().build(input_shape)

    def call(self, x: tf.Tensor, epsilon: float = 1e-6) -> tf.Tensor:
        mean = tf.reduce_mean(x, axis=[-1], keepdims=True)
        variance = tf.reduce_mean(tf.square(x - mean), axis=[-1], keepdims=True)
        norm_x = (x - mean) * tf.rsqrt(variance + epsilon)

        return norm_x * self.scale + self.bias
```

8. Residual Connect

Transformer模型包括一系列正则化技术，例如层归一化、Dropout和残差连接。这些技术对于确保模型的稳健性和泛化能力至关重要。为了简化代码实现并提高可读性，通常会创建一个包装器来封装这些正则化层，并将它们应用于Transformer模型的各个部分。



其中，残差连接（Residual Connections）则允许信息在网络中直接流过跳跃连接，有助于缓解梯度消失问题，提升模型的训练效果。

```
class ResidualNormalizationWrapper(tf.keras.models.Model):
    def __init__(self, layer: tf.keras.layers.Layer, dropout_rate: float, *args, **kwargs) -> None
        super().__init__(*args, **kwargs)
        self.layer = layer
        self.layer_normalization = LayerNormalization()
        self.dropout_layer = tf.keras.layers.Dropout(dropout_rate)

    def call(self, input: tf.Tensor, training: bool, *args, **kwargs) -> tf.Tensor:
        tensor = self.layer_normalization(input)
        tensor = self.layer(tensor, training=training, *args, **kwargs)
        tensor = self.dropout_layer(tensor, training=training)
        return input + tensor
```

9. Positional Encoding

到此为止，Transformer还无法利用token中的顺序来进行学习。换句话说，”我打你“和”你打我“这两个意思截然相反的语句，在模型看来依然是一致的。

在Transformer论文中位置编码的公式是这样的：

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

选择这个公式的原因是因为它可以将位置间的欧几里德距离表示为线性函数，这对神经网络来说更容易学习。

让我们来详细解释一下。

首先，假设我们有两个位置 (x) 和 (y)，它们的欧几里德距离可以表示为：

$$[\sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \ldots + (x_n - y_n)^2}]$$

现在，我们将位置 (x) 和 (y) 表示为正弦和余弦函数的组合形式：

$$[x = (\sin(w_1), \cos(w_1), \ldots, \sin(w_n), \cos(w_n))]$$

$$[y = (\sin(w_1 + \phi), \cos(w_1 + \phi), \ldots, \sin(w_n + \phi), \cos(w_n + \phi))]$$

这里，(w_i) 是位置 (x) 和 (y) 在第 (i) 维上的角度，( ) 是一个偏移角度。

现在，我们来计算这两个位置的欧几里德距离。注意到正弦函数和余弦函数之间存在关系 。

$$\cos(x) = \sin(x + \frac{\pi}{2})$$

这意味着我们可以使用这个关系来将余弦函数转换为正弦函数。这样一来，我们可以将（y）中的余弦项转换为对应的正弦项。

通过这种转换，我们可以将欧几里德距离表示为各维度差的线性和，这样神经网络就可以更容易地学习位置信息了。

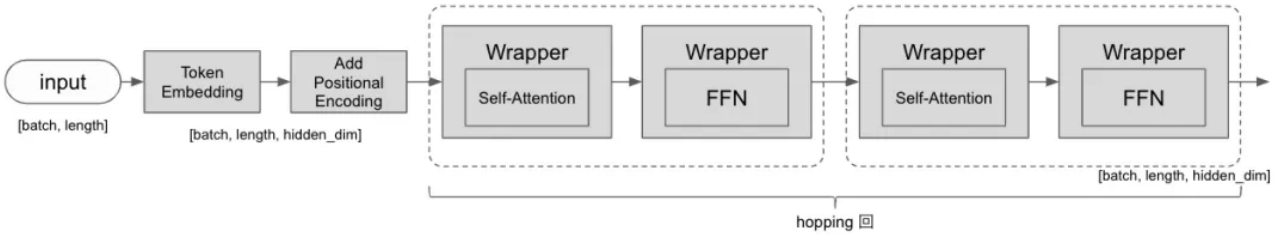
其中，10000是一个经验常数，被用作一个特征缩放因子，因为 $\frac{1}{10000^{2i/d_{\text{model}}}}$ 随着（i）的增加而减小，它可以确保不同维度的位置编码在不同的尺度上变化，从而使得不同位置的编码向量有所区别。

具体的代码实现如下所示：

```
class AddPositionalEncoding(tf.keras.layers.Layer):
    '''
    PE_{pos, 2i}   = sin(pos / 10000^{2i / d_model})
    PE_{pos, 2i+1} = cos(pos / 10000^{2i / d_model})
    '''
    def call(self, inputs: tf.Tensor) -> tf.Tensor:
        fl_type = inputs.dtype
        batch_size, max_length, depth = tf.unstack(tf.shape(inputs))

        depth_counter = tf.range(depth) // 2 * 2  # 0, 0, 2, 2, 4, ...
        depth_matrix = tf.tile(tf.expand_dims(depth_counter, 0), [max_length, 1])  # [max_length,
        depth_matrix = tf.pow(10000.0, tf.cast(depth_matrix / depth, fl_type))  # [max_length, dep

        # cos(x) == sin(x + π/2)
        phase = tf.cast(tf.range(depth) % 2, fl_type) * math.pi / 2  # 0, π/2, 0, π/2, ...
        phase_matrix = tf.tile(tf.expand_dims(phase, 0), [max_length, 1])  # [max_length, depth]

        pos_counter = tf.range(max_length)
        pos_matrix = tf.cast(tf.tile(tf.expand_dims(pos_counter, 1), [1, depth]), fl_type)  # [max

        positional_encoding = tf.sin(pos_matrix / depth_matrix + phase_matrix)
        # [batch_size, max_length, depth]
        positional_encoding = tf.tile(tf.expand_dims(positional_encoding, 0), [batch_size, 1, 1])

        return inputs + positional_encoding
```

10. Encoder

Transformer主要由Encoder和Decoder组成。
在这里，我们将创建一个来对输入token字符串进行编码，并引入自注意力对输入进行处理。



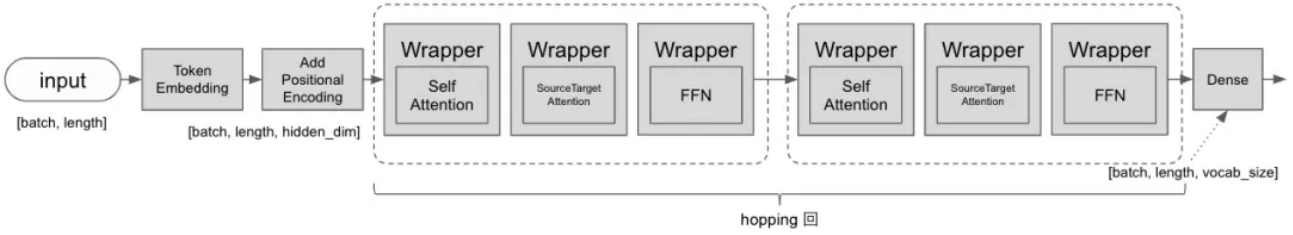
其实现代码如下所示：

```
class Encoder(tf.keras.models.Model):
    def __init__(
            self,
            vocab_size: int,
            hopping_num: int,
            head_num: int,
            hidden_dim: int,
            dropout_rate: float,
            max_length: int,
            *args,
            **kwargs,
    ) -> None:
        super().__init__(*args, **kwargs)
        self.hopping_num = hopping_num
        self.head_num = head_num
        self.hidden_dim = hidden_dim
        self.dropout_rate = dropout_rate

        self.token_embedding = TokenEmbedding(vocab_size, hidden_dim)
        self.add_position_embedding = AddPositionalEncoding()
        self.input_dropout_layer = tf.keras.layers.Dropout(dropout_rate)

        self.attention_block_list: List[List[tf.keras.models.Model]] = []
        for _ in range(hopping_num):
            attention_layer = SelfAttention(hidden_dim, head_num, dropout_rate, name='self_attenti
            ffn_layer = FeedForwardNetwork(hidden_dim, dropout_rate, name='ffn')
            self.attention_block_list.append([
                ResidualNormalizationWrapper(attention_layer, dropout_rate, name='self_attention_w
                ResidualNormalizationWrapper(ffn_layer, dropout_rate, name='ffn_wrapper'),
            ])
        self.output_normalization = LayerNormalization()

    def call(
            self,
            input: tf.Tensor,
            self_attention_mask: tf.Tensor,
            training: bool,
    ) -> tf.Tensor:
        # [batch_size, length, hidden_dim]
        embedded_input = self.token_embedding(input)
        embedded_input = self.add_position_embedding(embedded_input)
```

```
        query = self.input_dropout_layer(embedded_input, training=training)

        for i, layers in enumerate(self.attention_block_list):
            attention_layer, ffn_layer = tuple(layers)
            with tf.name_scope(f'hopping_{i}'):
                query = attention_layer(query, attention_mask=self_attention_mask, training=traini
                query = ffn_layer(query, training=training)
        # [batch_size, length, hidden_dim]
        return self.output_normalization(query)
```

11. Decoder

Decoder 首先对输入应用 Self-Attention，然后使用 Target-Attention 捕获 Encoder 编码的信息。
这将输入从时间 0 到 t 的token字符串，并输出从时间 1 到 t+1（即未来的一个时间）的令牌字符串。



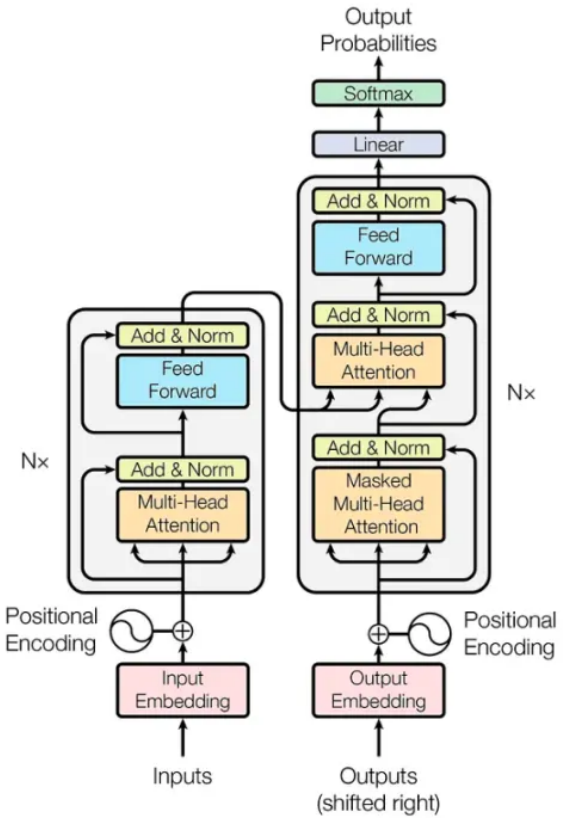
其代码实现如下所示：

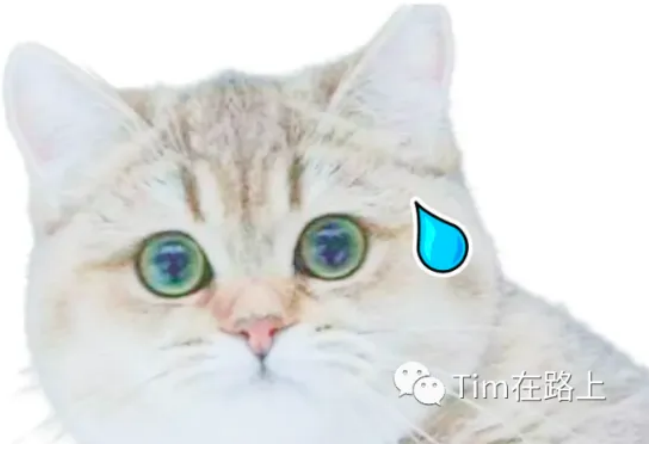
```
class Decoder(tf.keras.models.Model):
    def __init__(
            self,
            vocab_size: int,
            hopping_num: int,
            head_num: int,
            hidden_dim: int,
            dropout_rate: float,
            max_length: int,
            *args,
            **kwargs,
    ) -> None:
        super().__init__(*args, **kwargs)
        self.hopping_num = hopping_num
        self.head_num = head_num
        self.hidden_dim = hidden_dim
        self.dropout_rate = dropout_rate

        self.token_embedding = TokenEmbedding(vocab_size, hidden_dim)
        self.add_position_embedding = AddPositionalEncoding()
        self.input_dropout_layer = tf.keras.layers.Dropout(dropout_rate)

        self.attention_block_list: List[List[tf.keras.models.Model]] = []
        for _ in range(hopping_num):
            self_attention_layer = SelfAttention(hidden_dim, head_num, dropout_rate, name='self_at
```

```
            enc_dec_attention_layer = MultiheadAttention(hidden_dim, head_num, dropout_rate, name=
            ffn_layer = FeedForwardNetwork(hidden_dim, dropout_rate, name='ffn')
            self.attention_block_list.append([
                ResidualNormalizationWrapper(self_attention_layer, dropout_rate, name='self_attent
                ResidualNormalizationWrapper(enc_dec_attention_layer, dropout_rate, name='enc_dec_
                ResidualNormalizationWrapper(ffn_layer, dropout_rate, name='ffn_wrapper'),
            ])
        self.output_normalization = LayerNormalization()
        self.output_dense_layer = tf.keras.layers.Dense(vocab_size, use_bias=False)

    def call(
            self,
            input: tf.Tensor,
            encoder_output: tf.Tensor,
            self_attention_mask: tf.Tensor,
            enc_dec_attention_mask: tf.Tensor,
            training: bool,
    ) -> tf.Tensor:
        # [batch_size, length, hidden_dim]
        embedded_input = self.token_embedding(input)
        embedded_input = self.add_position_embedding(embedded_input)
        query = self.input_dropout_layer(embedded_input, training=training)

        for i, layers in enumerate(self.attention_block_list):
            self_attention_layer, enc_dec_attention_layer, ffn_layer = tuple(layers)
            with tf.name_scope(f'hopping_{i}'):
                query = self_attention_layer(query, attention_mask=self_attention_mask, training=t
                query = enc_dec_attention_layer(query, memory=encoder_output,
                                                attention_mask=enc_dec_attention_mask, training=tr
                query = ffn_layer(query, training=training)

        query = self.output_normalization(query)  # [batch_size, length, hidden_dim]
        return self.output_dense_layer(query)  # [batch_size, length, vocab_size]
```

12. Transformer

最后，我们对Encoder和Decoder进行组合链接便形成了最终的Transformer。到此这篇文章也算有始有终了。





实现代码如下所示：

```
class Transformer(tf.keras.models.Model):
    def __init__(
            self,
            vocab_size: int,
            hopping_num: int = 4,
            head_num: int = 8,
            hidden_dim: int = 512,
            dropout_rate: float = 0.1,
            max_length: int = 200,
            *args,
            **kwargs,
    ) -> None:
        super().__init__(*args, **kwargs)
        self.vocab_size = vocab_size
        self.hopping_num = hopping_num
        self.head_num = head_num
        self.hidden_dim = hidden_dim
        self.dropout_rate = dropout_rate
        self.max_length = max_length

        self.encoder = Encoder(
            vocab_size=vocab_size,
            hopping_num=hopping_num,
            head_num=head_num,
            hidden_dim=hidden_dim,
            dropout_rate=dropout_rate,
            max_length=max_length,
        )
        self.decoder = Decoder(
```

```
            vocab_size=vocab_size,
            hopping_num=hopping_num,
            head_num=head_num,
            hidden_dim=hidden_dim,
            dropout_rate=dropout_rate,
            max_length=max_length,
        )

    def build_graph(self, name='transformer') -> None:
        with tf.name_scope(name):
            self.is_training = tf.placeholder(dtype=tf.bool, name='is_training')
            # [batch_size, max_length]
            self.encoder_input = tf.placeholder(dtype=tf.int32, shape=[None, None], name='encoder_
            # [batch_size]
            self.decoder_input = tf.placeholder(dtype=tf.int32, shape=[None, None], name='decoder_

            logit = self.call(
                encoder_input=self.encoder_input,
                decoder_input=self.decoder_input[:, :-1],
                training=self.is_training,
            )
            decoder_target = self.decoder_input[:, 1:]

            self.prediction = tf.nn.softmax(logit, name='prediction')

            with tf.name_scope('metrics'):
                xentropy, weights = padded_cross_entropy_loss(
                    logit, decoder_target, smoothing=0.05, vocab_size=self.vocab_size)
                self.loss = tf.identity(tf.reduce_sum(xentropy) / tf.reduce_sum(weights), name='lo

                accuracies, weights = padded_accuracy(logit, decoder_target)
                self.acc = tf.identity(tf.reduce_sum(accuracies) / tf.reduce_sum(weights), name='a

    def call(self, encoder_input: tf.Tensor, decoder_input: tf.Tensor, training: bool) -> tf.Tensc
        enc_attention_mask = self._create_enc_attention_mask(encoder_input)
        dec_self_attention_mask = self._create_dec_self_attention_mask(decoder_input)

        encoder_output = self.encoder(
            encoder_input,
            self_attention_mask=enc_attention_mask,
            training=training,
        )
        decoder_output = self.decoder(
            decoder_input,
            encoder_output,
            self_attention_mask=dec_self_attention_mask,
            enc_dec_attention_mask=enc_attention_mask,
            training=training,
        )
        return decoder_output

    def _create_enc_attention_mask(self, encoder_input: tf.Tensor):
        with tf.name_scope('enc_attention_mask'):
            batch_size, length = tf.unstack(tf.shape(encoder_input))
            pad_array = tf.equal(encoder_input, PAD_ID)  # [batch_size, m_length]
            # shape broadcasting to [batch_size, head_num, (m|q)_length, m_length]
            return tf.reshape(pad_array, [batch_size, 1, 1, length])
```

```
    def _create_dec_self_attention_mask(self, decoder_input: tf.Tensor):
        with tf.name_scope('dec_self_attention_mask'):
            batch_size, length = tf.unstack(tf.shape(decoder_input))
            pad_array = tf.equal(decoder_input, PAD_ID)  # [batch_size, m_length]
            pad_array = tf.reshape(pad_array, [batch_size, 1, 1, length])

            autoregression_array = tf.logical_not(
                tf.matrix_band_part(tf.ones([length, length], dtype=tf.bool), -1, 0))
            autoregression_array = tf.reshape(autoregression_array, [1, 1, length, length])

            return tf.logical_or(pad_array, autoregression_array)
```

总结

尽管宇宙的熵在不断增加，生命体和智能体却能够通过学习在局部范围内实现减熵，即通过摄取和转换能量来增加内部的有序性。Transformer模型正是这样一种智能体，它通过学习数据中的模式和规律，提高了对序列数据的处理能力。
本文详细介绍了Transformer模型的所有关键组成部分，并分析了其为什么这么实现，实现的原理是什么。同时每个部分都配有相应的代码实现和解释，希望您能够更深入地理解Transformer模型的工作原理。

此外，我们也分析了Transformer模型相对于传统RNN的优势，如并行化能力、简洁的结构设计、以及通过多头注意力机制对序列中的多个位置进行同时关注的能力。此外，通过位置编码能够捕捉序列中元素的顺序信息，这对于处理自然语言等序列数据至关重要。

本来这次还设想的主题探讨是为什么是Transformer以及必须是Transformer吗？

一方面文章越来越长了，另一方面输出和想法的对齐还是很难的，索性这次再次详细介绍了Transformer实现中的细节，以及为什么这么实现和提供了实现的源码，相信看过了这篇文章，所有人对整个Transformer的结构和原理将牢记于心。

希望这篇文章可以对你有所帮助，也欢迎探讨指正，求个在看，喜欢的多的话后面在分析Transformer和一些变种新架构的对比。