Transformer 相关

一、Self-attention 机制介绍

1.Scaled Dot-Producted Attention

关于self-attention的介绍，李宏毅大佬的课程讲的非常详细了：
第四节 2021 - 自注意力机制(Self-attention)(上)*哔哩哔哩*bilibili

2021 - 自注意力机制 (Self-attention) (下)*哔哩哔哩*bilibili

最关键的核心部分就在于下面这个公式：

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

其中的Q,K,V都是将输入序列向量按行摆得到的结果，具体来说size应该是**seq_Len * dim**。在原论文中dim_k和dim_v是相等的。

接下来，来写一个Self-attention的相关pytorch代码：

```
from math import sqrt
import torch
from torch import nn

class Self_Attention(nn.Module):
    # input : batch_size * seq_len * input_dim
    # q : batch_size * input_dim * dim_k
    # k : batch_size * input_dim * dim_k
    # v : batch_size * input_dim * dim_v
    def __init__(self, input_dim, dim_k, dim_v):
        super(Self_Attention, self).__init__()
        self.q = nn.Linear(input_dim, dim_k)
        self.k = nn.Linear(input_dim, dim_k)
        self.v = nn.Linear(input_dim, dim_v)
        self._norm_fact = 1 / sqrt(dim_k)

    def forward(self, x):
        # x: batch_size * seq_len * input_dim
        Q = self.q(x)  # batch_size * seq_len * dim_k
        K = self.k(x)  # batch_size * seq_len * dim_k
        V = self.v(x)  # batch_size * seq_len * dim_v

        atten = nn.Softmax(dim=-1)(torch.bmm(Q, K.permute(0, 2, 1))) *
self._norm_fact  # Q * K.T(), batch_size * seq_len * seq_len
        output = torch.bmm(atten, V)  # batch_size * seq_len * dim_v
        return output
```

> 相关Python及Pytorch语法补充：
> 【1】关于import xxx 和 from xxx import yyy的区别：【精选】from xxx import xxx 和 import xxx的区别
>
> 【2】`nn.Linear`：Linear — PyTorch 2.1 documentation。经过全连接层后的输出与输入的数据除了最后一个维度变了其他维度都是不变的。
>
> 【3】`torch.bmm`：torch.bmm — PyTorch 2.1 documentation，直观理解其实就是忽略第一维（即batch_size）的矩阵乘法。
>
> 对于`atten = nn.Softmax(dim=-1)(torch.bmm(Q, K.permute(0, 2, 1))) * self._norm_fact`这段代码，`K.permute(0, 2, 1)`指的也是K在保持第一维度不变时的矩阵转置。**所以这行代码其实就是上图那个公式。**

对搭建的网络进行简单测试：

```
import torch
from self_attention import Self_Attention

X = torch.randn(4, 3, 2)  # batch_size * seq_len * input_dim
self_attention = Self_Attention(2, 4, 5)
res = self_attention(X)
print(res)
print(res.size())
```

输出结果如下：

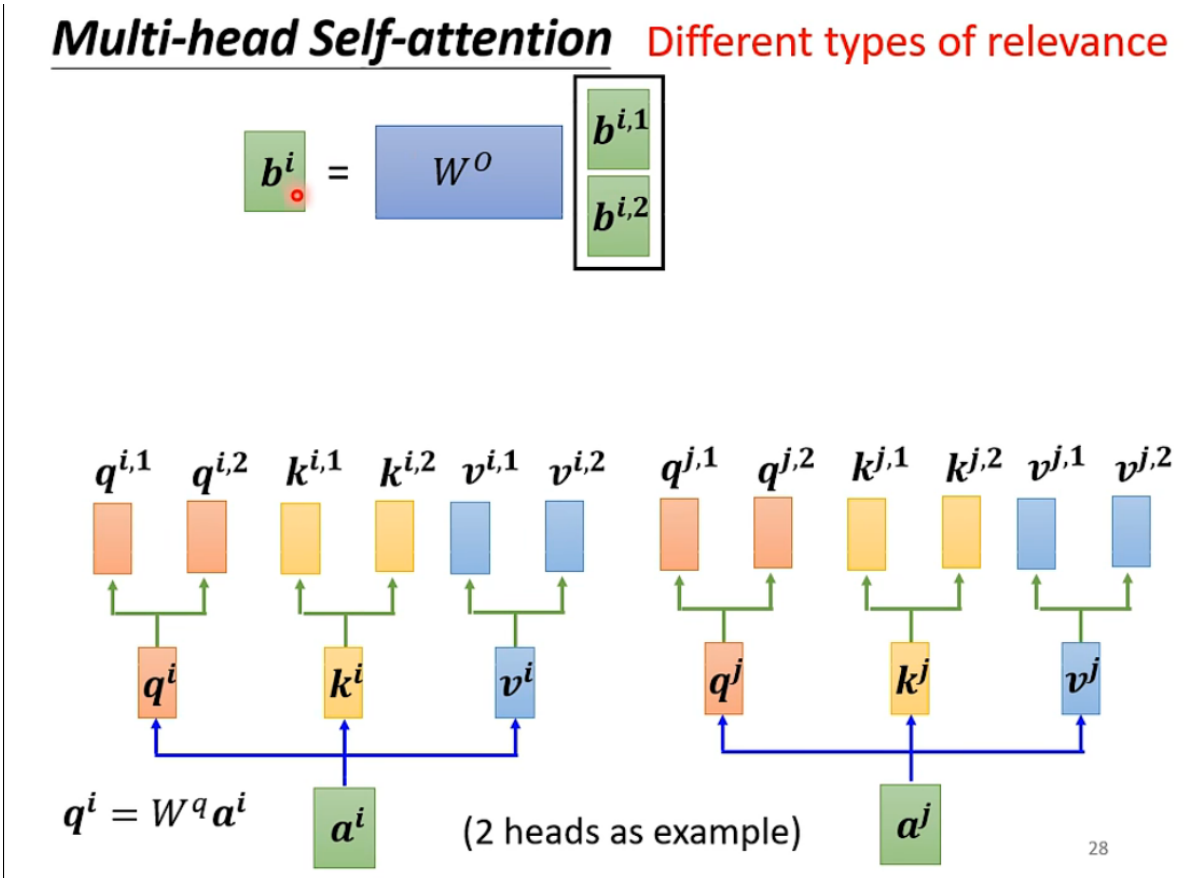
```
tensor([[[ 0.0375,  0.3084, -0.2903, -0.6260, -0.3264],
         [-0.2217,  0.4926, -0.1018, -0.6571, -0.5343],
         [-0.1072,  0.4113, -0.1851, -0.6434, -0.4424]],

        [[ 0.0608,  0.1304, -0.1767, -0.4129, -0.2583],
         [ 0.1625,  0.0335, -0.2308, -0.3686, -0.1692],
         [ 0.3944, -0.1796, -0.3603, -0.2779,  0.0316]],

        [[ 0.5211, -0.5221, -0.2484,  0.0662,  0.2105],
         [ 0.3057, -0.2888, -0.1565, -0.0640,  0.0132],
         [ 0.3687, -0.1121, -0.3815, -0.3451, -0.0041]],

        [[ 0.1678,  0.0417, -0.2443, -0.3836, -0.1686],
         [ 0.1601,  0.0552, -0.2452, -0.3949, -0.1772],
         [ 0.1744,  0.0234, -0.2381, -0.3651, -0.1591]]],
       grad_fn=<BmmBackward0>)
torch.Size([4, 3, 5])
```

2.Multi-head Self-attention

相关的原理，依旧可以参考李宏毅大佬的视频：2021 - 自注意力机制 (Self-attention) (下)*哔哩哔哩* bilibili，大概从15分钟左右开始。

有时间可以看看下面的文章：Multi-Head-Attention的作用到底是什么? - 知乎 (zhihu.com)

简单来说，Multi-head可以用下图来形容：



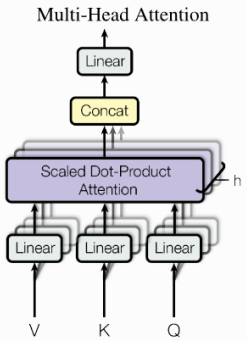
参考[Tutorial 6: Transformers and Multi-Head Attention — UvA DL Notebooks v1.2 documentation (uvadlc-notebooks.readthedocs.io)](#)，可以给出另一张图：

Multi-Head Attention

The scaled dot product attention allows a network to attend over a sequence. However, often there are multiple different aspects a sequence element wants to attend to, and a single weighted average is not a good option for it. This is why we extend the attention mechanisms to multiple heads, i.e. multiple different query-key-value triplets on the same features. Specifically, given a query, key, and value matrix, we transform those into h sub-queries, sub-keys, and sub-values, which we pass through the scaled dot product attention independently. Afterward, we concatenate the heads and combine them with a final weight matrix. Mathematically, we can express this operation as:

$$\text{Multihead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O$$
$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

We refer to this as Multi-Head Attention layer with the learnable parameters $W^Q_{1\dots h} \in \mathbb{R}^{D\times d_k}$, $W^K_{1\dots h} \in \mathbb{R}^{D\times d_k}$, $W^V_{1\dots h} \in \mathbb{R}^{D\times d_v}$, and $W^O \in \mathbb{R}^{h\cdot d_v \times d_{out}}$ (D being the input dimensionality). Expressed in a computational graph, we can visualize it as below (figure credit - Vaswani et al., 2017).



其实就是把本来的Q，K，V矩阵额外再分出multi-head的head的数量的矩阵，然后对应的head进行attention的求解并求出对应的b，然后可以用一个新的矩阵W^O来计算出最后的b_i。

下面基于Pytorch实现multi-head self-attention（**注：由于要实现Transformer，这里参考的模型代码链接为[A Comprehensive Guide to Building a Transformer Model with PyTorch | DataCamp](#)**，可能相对来说会更严谨一些）：

```
class MultiHeadAttention(nn.Module):
```

```
    def __init__(self, d_model, num_heads):
        super(MultiHeadAttention, self).__init__()
        # Ensure that the model dimension (d_model) is divisible by the number
of heads
        assert d_model % num_heads == 0, "d_model must be divisible by
num_heads"

        # Initialize dimensions
        self.d_model = d_model  # Model's dimension
        self.num_heads = num_heads  # Number of attention heads
        self.d_k = d_model // num_heads  # Dimension of each head's key, query,
and value

        # Linear layers for transforming inputs
        self.W_q = nn.Linear(d_model, d_model)  # Query transformation
        self.W_k = nn.Linear(d_model, d_model)  # Key transformation
        self.W_v = nn.Linear(d_model, d_model)  # Value transformation
        self.W_o = nn.Linear(d_model, d_model)  # Output transformation

    # 前面实现的scaled_dot_product_attention的逻辑
    def scaled_dot_product_attention(self, Q, K, V):
        # Calculate attention scores
        attn_scores = torch.matmul(Q, K.transpose(-2, -1)) / sqrt(self.d_k)

        # Softmax is applied to obtain attention probabilities
        attn_probs = torch.softmax(attn_scores, dim=-1)

        # Multiply by values to obtain the final output
        output = torch.matmul(attn_probs, V)
        return output

    def split_heads(self, x):
        # Reshape the input to have num_heads for multi-head attention
        batch_size, seq_length, d_model = x.size()
        return x.view(batch_size, seq_length, self.num_heads,
self.d_k).transpose(1, 2)  #  transpose:The given dimensions dim0 and dim1 are
swapped.
        # return.shape: [batch_size, self.num_heads, seq_length, self.d_k]

    def combine_heads(self, x):
        # Combine the multiple heads back to original shape
        batch_size, _, seq_length, d_k = x.size()
        # 关于contiguous()函数的意
义:https://blog.csdn.net/gdymind/article/details/82662502
        return x.transpose(1, 2).contiguous().view(batch_size, seq_length,
self.d_model)

    def forward(self, Q, K, V):  # 实际上传入的可能是(x,x,x)或者(x, enc_output,
enc_output)这种
        # Apply linear transformations and split heads
        Q = self.split_heads(self.W_q(Q))
        K = self.split_heads(self.W_k(K))
        V = self.split_heads(self.W_v(V))

        # Perform scaled dot-product attention
```

```
        attn_output = self.scaled_dot_product_attention(Q, K, V)

        # Combine heads and apply output transformation
        output = self.W_o(self.combine_heads(attn_output))
        return output
```

补充：关于`torch.matmul`函数的介绍：[torch.matmul — PyTorch 2.1 documentation]

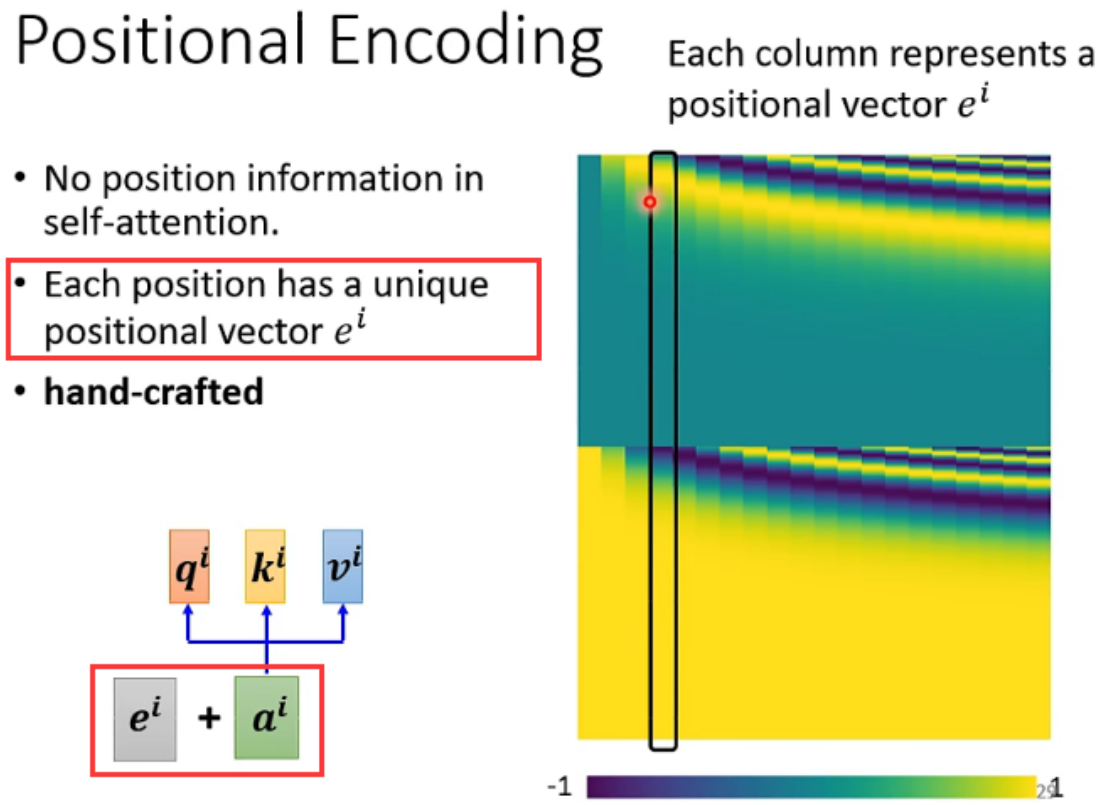
> 关于forward函数的介绍：
>
> The forward method is where the actual computation happens:
>
> 1. Apply Linear Transformations: The queries (Q), keys (K), and values (V) are first passed through linear transformations using the weights defined in the initialization.
> 2. Split Heads: The transformed Q, K, V are split into multiple heads using the split_heads method.
> 3. Apply Scaled Dot-Product Attention: The scaled_dot_product_attention method is called on the split heads.
> 4. Combine Heads: The results from each head are combined back into a single tensor using the combine_heads method.
> 5. Apply Output Transformation: Finally, the combined tensor is passed through an output linear transformation.

注： Multi-Head Attention里面还可以有个Mask的操作，不过这个我们放在后面学到了再进行总结。

3.Positional Encoding

在前面的Self-Attention中，我们并没有引入向量在Sequence所在位置的信息。比如说“Vec1和Vec4”以及“Vec2和Vec3”之间并没有考虑距离的影响。而在做任务中，位置信息往往也是很有用的（比如说对于词性标注任务，动词不太可能出现在句首），而此时的网络还没有考虑这种位置信息。

Positional Encoding就是用来解决这个问题的，示例图如下：



上述参考的文章对此的介绍为：

> Positional Encoding is used to inject the position information of each token in the input sequence. It uses sine and cosine functions of different frequencies to generate the positional encoding.

在《Attention is all you need》这篇初始论文中，作者对此的介绍如下：

In this work, we use sine and cosine functions of different frequencies:

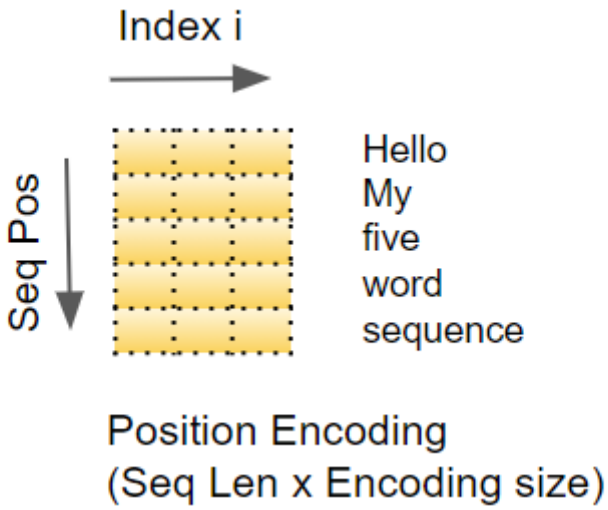
$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

where pos is the position and i is the dimension. That is, each dimension of the positional encoding corresponds to a sinusoid. The wavelengths form a geometric progression from 2π to $10000 \cdot 2\pi$. We chose this function because we hypothesized it would allow the model to easily learn to attend by relative positions, since for any fixed offset k, PE_{pos+k} can be represented as a linear function of PE_{pos}.

也就是说，positional encoding是与input sequence无关的量，可以认为是预定义好的值，只和sequence的最大长度有关。在上面的公式中：

- pos是序列中的word所在的位置；
- d_model是encoding向量的长度（和embedding vector一样）；
- i是上述向量的index value。

下面是示意图：



接下来，我们会在pytorch中实现Positional Encoding：

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_seq_length):
        super(PositionalEncoding, self).__init__()
        pe = torch.zeros(max_seq_length, d_model)
        position = torch.arange(0, max_seq_length,
dtype=torch.float).unsqueeze(1)  #
https://pytorch.org/docs/stable/generated/torch.unsqueeze.html
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * -
(math.log(10000.0) / d_model))  # 转换一下会发现和论文中公式一样

        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)

        self.register_buffer('pe', pe.unsqueeze(0))

    def forward(self, x):
        return x + self.pe[:, :x.size(1)]
```

介绍如下：

> - d_model: The dimension of the model's input.
> - max_seq_length: The maximum length of the sequence for which positional encodings are pre-computed.
> - pe: A tensor filled with zeros, which will be populated with positional encodings.
> - position: A tensor containing the position indices for each position in the sequence.
> - div_term: A term used to scale the position indices in a specific way.
>
> The sine function is applied to the even indices and the cosine function to the odd indices of pe.
>
> Finally, pe is registered as a buffer, which means it will be part of the module's state but will not be considered a trainable parameter.

一些新的研究里，Positional Encoding也是能根据数据集被学习出来的，不过这仍然是一个正在被研究的问题（截止到2021年左右，现在不确定）。

二、Transformer相关

Transformer结构现在有不少变体，有的Transformer架构不包含Decoder，只包含Encoder，不过这里我们还是先介绍最初始的Transformer。

1.Seq2Seq介绍

这种模型架构可以参考下面文章：[CS224n笔记7]:整理了12小时，只为让你20分钟搞懂Seq2seq - 知乎 (zhihu.com)

做简单理解的话，也可以参考李宏毅大佬的视频：2021 - Transformer (上)*哔哩哔哩*bilibili

---

2. Transformer-Encoder

注：这部分还没有讲到Transformer网络的训练，只介绍产生结果的过程。

Transformer的Encoder结构如下图所示：

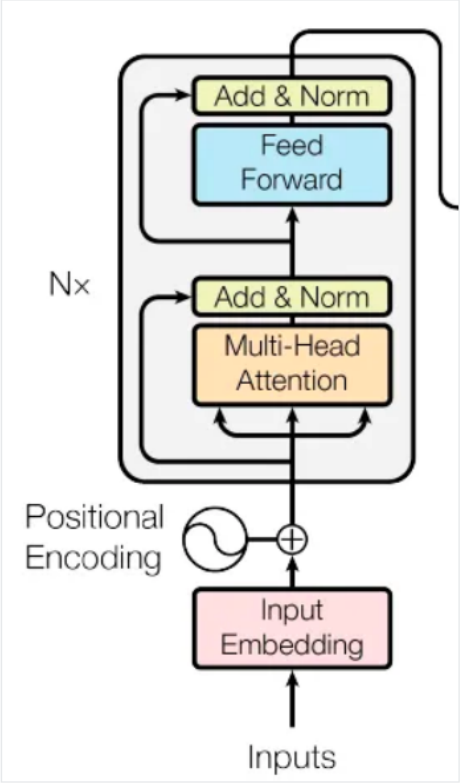


Figure 2. The Encoder part of the transformer network (Source: image from the original paper)

一些关于上图的解释：

- （1）Positional Encoding：在第一章节的第3部分有进行介绍，算是对self-attention引入位置相关的信息；
- （2）Add-Norm：其中的Add操作是由Residual Connection带来的，个人理解这是一种利用残差的思想，类似ResNet。而Norm操作则是layer-norm。
- （3）Nx的意思是这种块会重复很多组，共同构成Transformer的Encoder部分。
- （4）关于Feed Forward网络部分的介绍如下；

(1) Feed Forward Network

《Attention is All You Need》一文中，对此的介绍为：

3.3 Position-wise Feed-Forward Networks

In addition to attention sub-layers, each of the layers in our encoder and decoder contains a fully connected feed-forward network, which is applied to each position separately and identically. This consists of two linear transformations with a ReLU activation in between.

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{2}$$

While the linear transformations are the same across different positions, they use different parameters from layer to layer. Another way of describing this is as two convolutions with kernel size 1. The dimensionality of input and output is $d_{\text{model}} = 512$, and the inner-layer has dimensionality $d_{ff} = 2048$.

其实简单来说就是一个全连接层+ReLU+一个全连接层。Pytorch实现如下：

```
class PositionWiseFeedForward(nn.Module):
    def __init__(self, d_model, d_ff):
        super(PositionWiseFeedForward, self).__init__()
        self.fc1 = nn.Linear(d_model, d_ff)
        self.fc2 = nn.Linear(d_ff, d_model)
        self.relu = nn.ReLU()

    def forward(self, x):
        return self.fc2(self.relu(self.fc1(x)))
```

---

(2) 相关代码

有了以上的知识，就可以动手搭建基于Pytorch的transformer encoder的一个块组件了：

```
class EncoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, dropout):
        super(EncoderLayer, self).__init__()
        self.self_atten = MultiHeadAttention(d_model, num_heads)
        self.feed_forward = PositionWiseFeedForward(d_model, d_ff)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        attn_output = self.self_atten(x, x, x)
        x = self.norm1(x + self.dropout(attn_output))  # x +
self.dropout(attn_output)就对应上图的残差连接
        ff_output = self.feed_forward(x)
        x = self.norm2(x + self.dropout(ff_output))
        return x
```

> 关于dropout原论文的介绍：

> Residual Dropout：We apply **dropout [27] to the output of each sub-layer, before it is added to the sub-layer input and normalized.** In addition, we apply dropout to the sums of the embeddings and the positional encodings in both the encoder and decoder stacks. For the base model, we use a rate of Pdrop = 0.1.

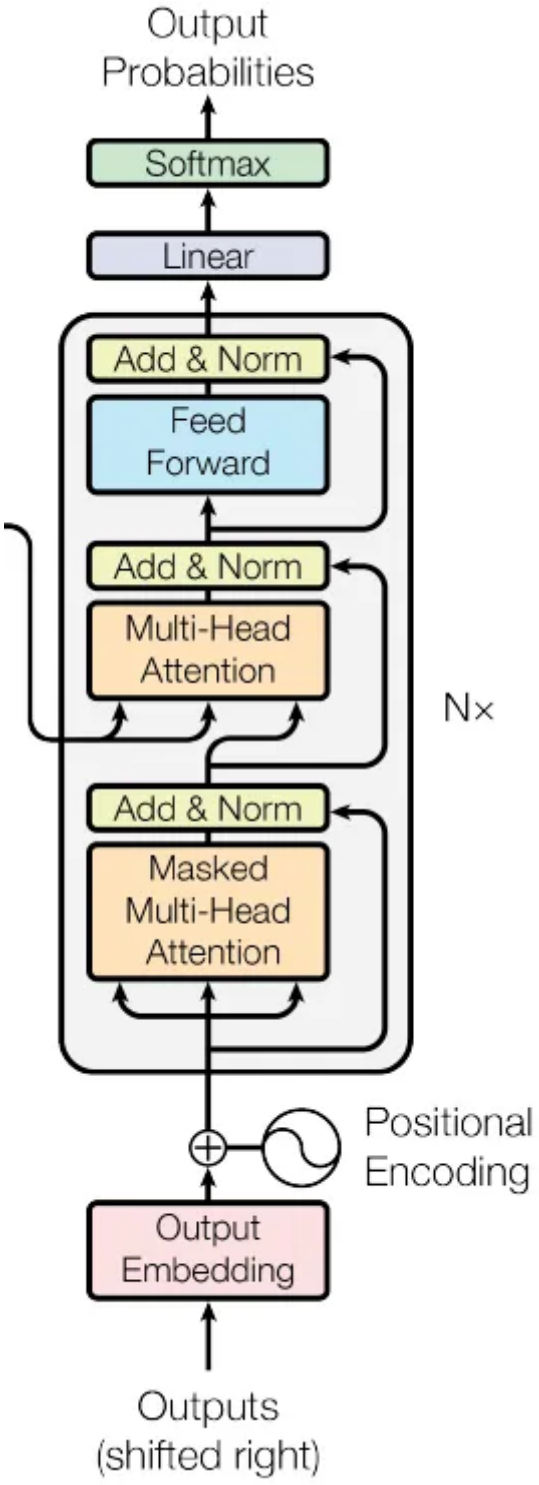
3.Transformer-Decoder

注：这部分还没有讲到Transformer网络的训练，只介绍产生结果的过程。

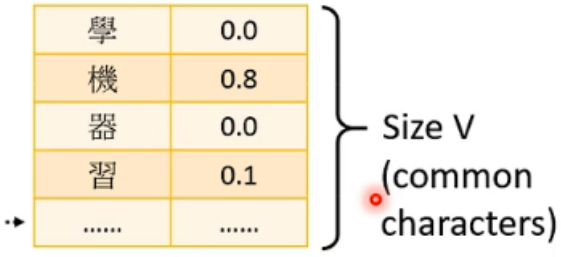
接下来介绍Transformer的Decoder部分。

（1）Autoregressive Decoder

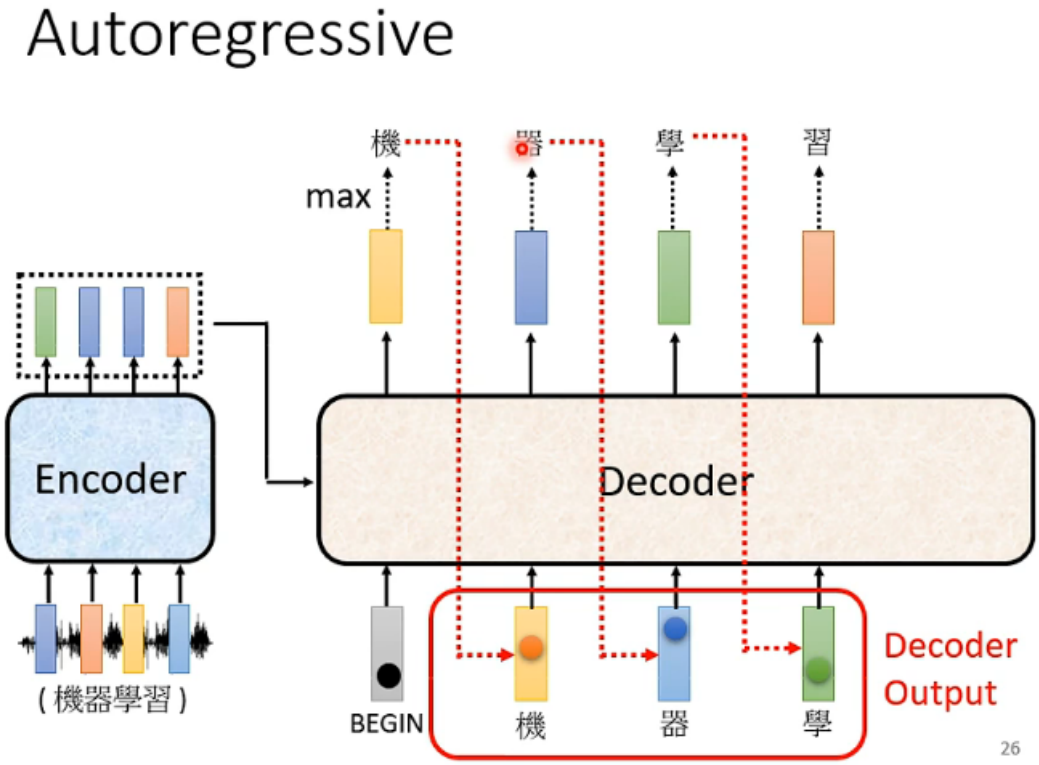
这也是比较常见的decoder类型，如下图所示：



上面这张图可能没有那么直观，其实这里的Output Probabilities指的是所有输出的可能性，一个类似于下图的东西：

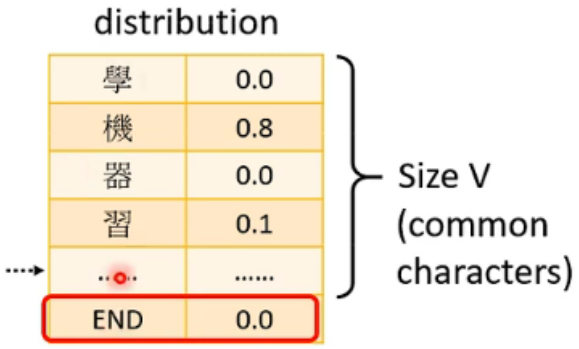


接下来，Decoder会决定输出是什么字符，比如说翻译任务这里得到了"机"字的结果，接着该输出作为下一个"时间点"的输入，整体流程更像是下面一幅图：



注：也许这里会出现Error Propagation（一步错，步步错）的问题，不过这里暂时先不讨论，先继续往下看。

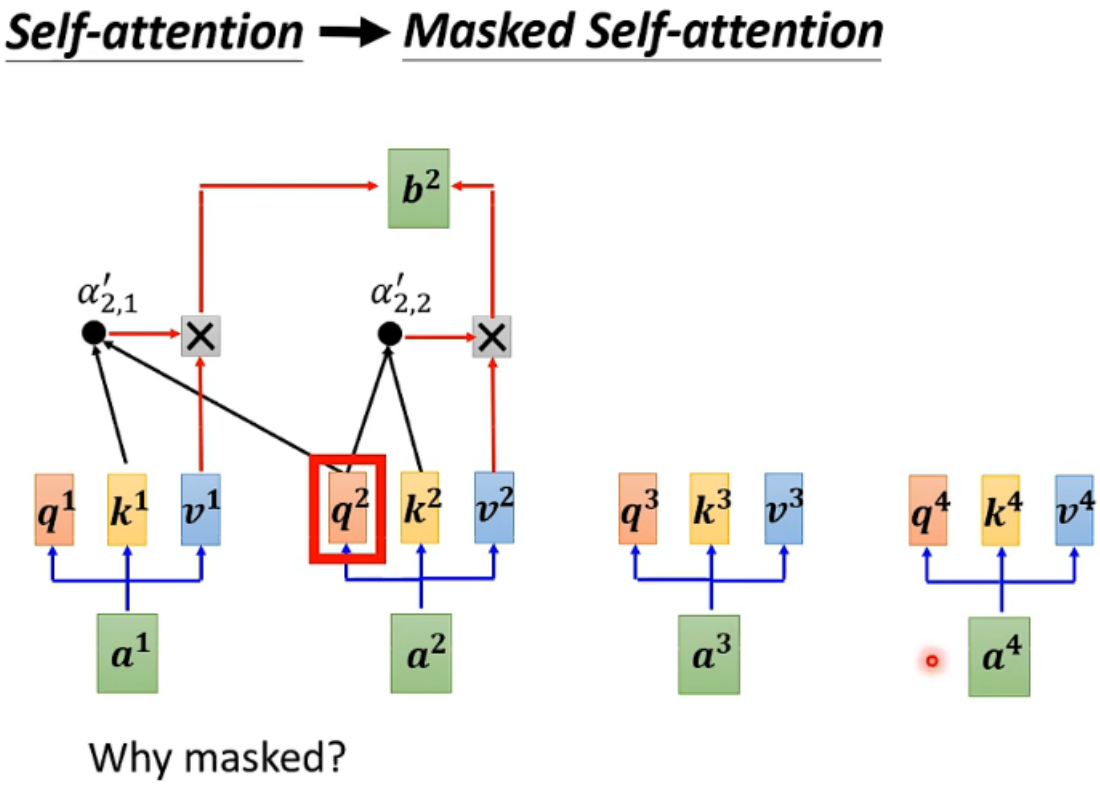
另外，这里还有一个**问题**是Decoder现在并没有办法判断任务结束，解决方案是在Output Probabilities的计算中加入一个END字段，使得当任务结束时Decoder有能力将END作为判断的输出，从而结束这个任务：



（a）Decoder的内部结构

Mask的概念

注意到Decoder的Multi-Head Attention中引入了一个Masked的概念。如下图：



Masked Self-attention和前面的self-attention的区别在于，在计算attention的过程中只考虑左边的向量，而不会考虑右边的向量（比如计算b2的时候只会考虑a1和a2）。这样做是合理的，因为decoder在输出结果的时候是按照顺序生成的，所以只能考虑到左边的内容而不能考虑右边的内容。

> **在计算attention score的时候如何对padding做mask操作?**
>
> - padding位置置为负无穷(一般来说-1000就可以)，再对attention score进行相加。这样的话mask的部分由于值非常小，在做softmax之后就会变成0。

对上述代码的修改部分有：

```
# class MultiHeadAttention
def scaled_dot_product_attention(self, Q, K, V, mask=None):
    # Calculate attention scores
    attn_scores = torch.matmul(Q, K.transpose(-2, -1)) / sqrt(self.d_k)

    # 核心：mask的相关逻辑
    # Apply mask if provided (useful for preventing attention to certain parts
like padding)
    if mask is not None:
        attn_scores = attn_scores.masked_fill(mask == 0, -1e9)

    # Softmax is applied to obtain attention probabilities
    attn_probs = torch.softmax(attn_scores, dim=-1)
    ...
def forward(self, Q, K, V, mask=None):  # 实际上传入的可能是(x,x,x)或者(x,
enc_output, enc_output)这种
    # Apply linear transformations and split heads
    Q = self.split_heads(self.W_q(Q))
    K = self.split_heads(self.W_k(K))
    V = self.split_heads(self.W_v(V))

    # Perform scaled dot-product attention
    attn_output = self.scaled_dot_product_attention(Q, K, V, mask)
```

```
    ...
```

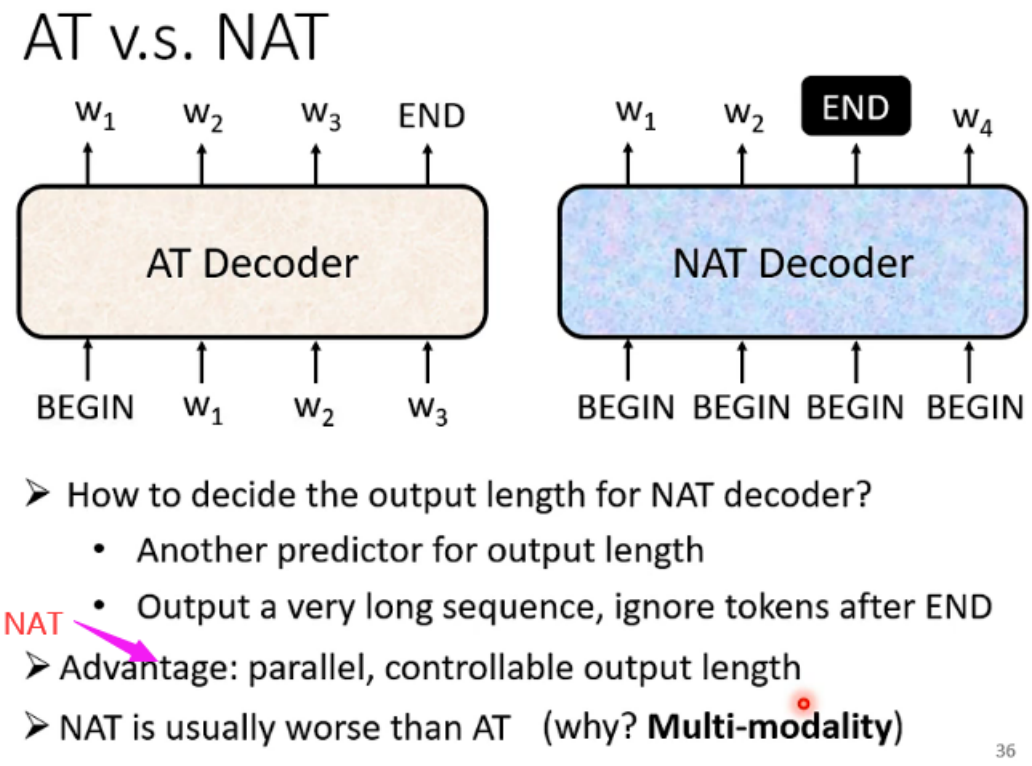
```
# class EncodeLayer
def forward(self, x, mask):
    attn_output = self.self_atten(x, x, x, mask)
    ...
```

其实就是在self-attention的`scaled_dot_product_attention`函数里引入masked的逻辑，并记得在调用该函数的地方和对应类处添加masked参数即可。

---

(2) Non-autoregressive（NAT） Decoder

NAT Decoder和Autoregressive（AT） Decoder 的区别如下图：



---

(3) Encoder和Decoder之间传递信息

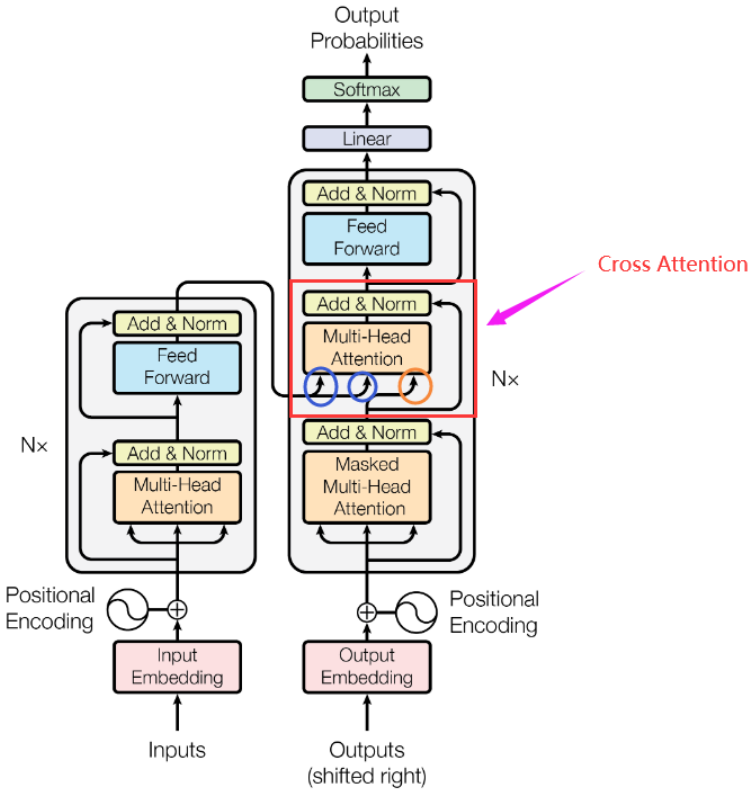
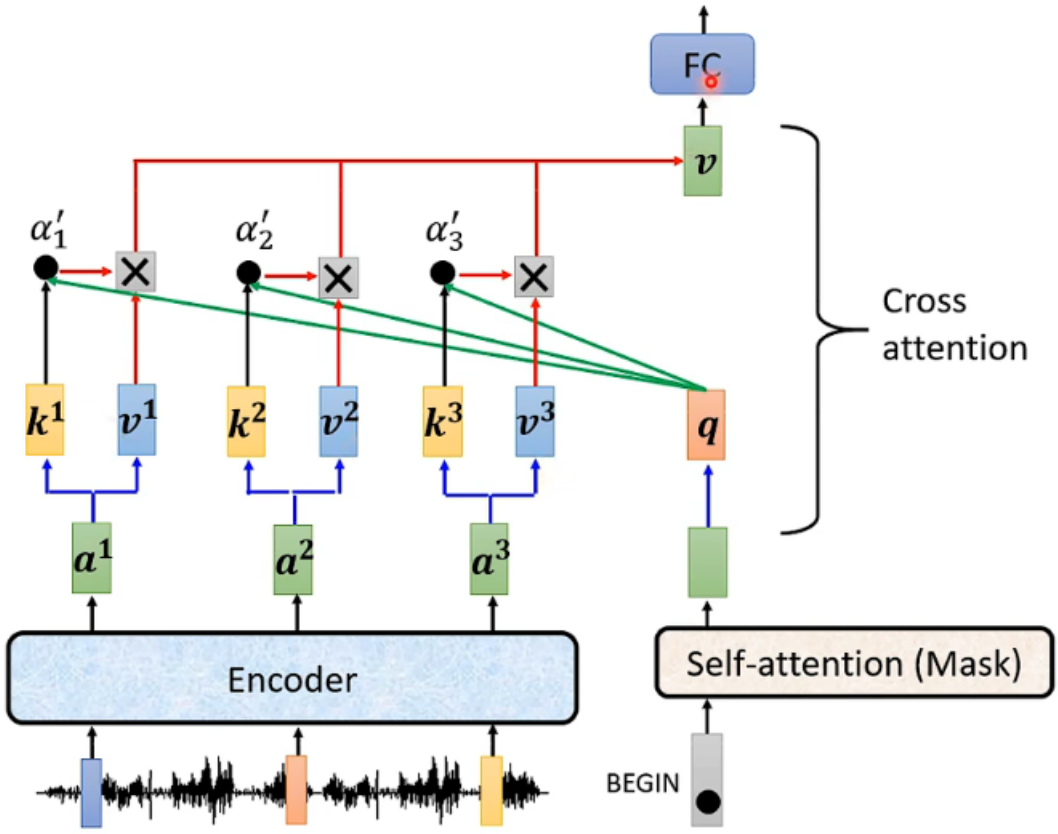


Figure 1: The Transformer - model architecture.

Cross Attention实际做的事情，可以用下图来形容：



也就是说，实际上Cross Attention是使用Encoder产生k和v向量，使用Decoder产生query（q）向量，来计算attention的操作，得到的结果会进一步输入后面的全连接神经网络中。

注：Decoder不管是哪个block，拿的都是Encoder最后一层的输出，作为Cross Attention的输入。当然这不是绝对的，有些工作也在研究Cross Attention的其他方法。

4.Decoder的代码实现

```
class DecoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, dropout):
        super(DecoderLayer, self).__init__()
        self.self_attn = MultiHeadAttention(d_model, num_heads)
        self.cross_attn = MultiHeadAttention(d_model, num_heads)
        self.feed_forward = PositionWiseFeedForward(d_model, d_ff)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.norm3 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, enc_output, src_mask, tgt_mask):
        attn_output = self.self_attn(x, x, x, tgt_mask)
        x = self.norm1(x + self.dropout(attn_output))
        attn_output = self.cross_attn(x, enc_output, enc_output, src_mask)  # 参数：Q,K,V,其中Q是decoder输出的,K和V是encoder输出的
        x = self.norm2(x + self.dropout(attn_output))
        ff_output = self.feed_forward(x)
        x = self.norm3(x + self.dropout(ff_output))
        return x
```

> 补充说明：
>
> Forward Method **Input**:
>
> 1. x: The input to the decoder layer.
> 2. enc_output: The output from the corresponding encoder (used in the cross-attention step).
> 3. src_mask: Source mask to ignore certain parts of the encoder's output.
> 4. tgt_mask: Target mask to ignore certain parts of the decoder's input.
>
> **Processing Steps:**
>
> 1. Self-Attention on Target Sequence: The input x is processed through a self-attention mechanism.
> 2. Add & Normalize (after Self-Attention): The output from self-attention is added to the original x, followed by dropout and normalization using norm1.
> 3. Cross-Attention with Encoder Output: The normalized output from the previous step is processed through a cross-attention mechanism that attends to the encoder's output enc_output.
> 4. Add & Normalize (after Cross-Attention): The output from cross-attention is added to the input of this stage, followed by dropout and normalization using norm2.
> 5. Feed-Forward Network: The output from the previous step is passed through the feed-forward network.
> 6. Add & Normalize (after Feed-Forward): The feed-forward output is added to the input of this stage, followed by dropout and normalization using norm3.
> 7. **Output: The processed tensor is returned as the output of the decoder layer.**

5.整个Transformer的代码实现

这里再给出Transformer的结构，方便跟代码对一下：

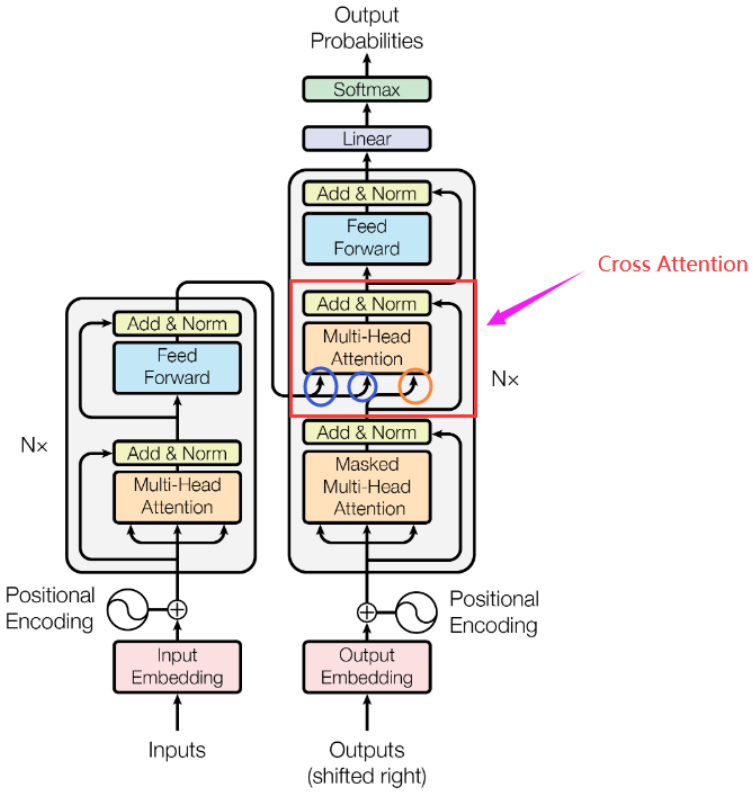


Figure 1: The Transformer - model architecture.

代码见下：

```
class Transformer(nn.Module):
    def __init__(self, src_vocab_size, tgt_vocab_size, d_model, num_heads,
num_layers, d_ff, max_seq_length, dropout):
        super(Transformer, self).__init__()
        self.encoder_embedding = nn.Embedding(src_vocab_size, d_model)  # The
embedding layer maps each input word into an embedding vector, which is a richer
representation of the meaning of that word.
        self.decoder_embedding = nn.Embedding(tgt_vocab_size, d_model)
        self.positional_encoding = PositionalEncoding(d_model, max_seq_length)

        self.encoder_layers = nn.ModuleList([EncoderLayer(d_model, num_heads,
d_ff, dropout) for _ in range(num_layers)])
        self.decoder_layers = nn.ModuleList([DecoderLayer(d_model, num_heads,
d_ff, dropout) for _ in range(num_layers)])

        self.fc = nn.Linear(d_model, tgt_vocab_size)
        self.dropout = nn.Dropout(dropout)

    def generate_mask(self, src, tgt):  # todo:这个逻辑接下来会介绍
        src_mask = (src != 0).unsqueeze(1).unsqueeze(2)
        tgt_mask = (tgt != 0).unsqueeze(1).unsqueeze(3)
        seq_length = tgt.size(1)
        nopeak_mask = (1 - torch.triu(torch.ones(1, seq_length, seq_length),
diagonal=1)).bool()
        tgt_mask = tgt_mask & nopeak_mask
```

```
        return src_mask, tgt_mask

    def forward(self, src, tgt):
        src_mask, tgt_mask = self.generate_mask(src, tgt)
        src_embedded =
self.dropout(self.positional_encoding(self.encoder_embedding(src)))
        tgt_embedded =
self.dropout(self.positional_encoding(self.decoder_embedding(tgt)))

        enc_output = src_embedded
        for enc_layer in self.encoder_layers:
            enc_output = enc_layer(enc_output, src_mask)

        dec_output = tgt_embedded
        for dec_layer in self.decoder_layers:
            dec_output = dec_layer(dec_output, enc_output, src_mask, tgt_mask)

        output = self.fc(dec_output)
        return output
```

相关逻辑说明

> 注：这段如果看不太懂的话继续看第6部分，可能会有更好的体会。
>
> 其他可以参考的文档：Transformer训练及测试阶段的self-attention mask理解 - 知乎 (zhihu.com)
>
> 浅析Transformer训练时并行问题 - 知乎 (zhihu.com)

Generate Mask Method:

This method is used to create masks for the source and target sequences, ensuring that padding tokens are ignored and that future tokens are not visible during training for the target sequence. The inclusion of masking ensures that the model adheres to the causal dependencies（因果依赖关系） within sequences, ignoring padding tokens and preventing information leakage from future tokens.

Forward Method:

This method defines the forward pass for the Transformer, taking source and target sequences and producing the output predictions.

1. Input Embedding and Positional Encoding: The source and target sequences are first embedded using their respective embedding layers and then added to their positional encodings.
2. Encoder Layers: The source sequence is passed through the encoder layers, with the final encoder output representing the processed source sequence.
3. Decoder Layers: The target sequence and the encoder's output are passed through the decoder layers, resulting in the decoder's output.
4. Final Linear Layer: The decoder's output is mapped to the target vocabulary size using a fully connected (linear) layer.

关于generate_mask函数的分析（来自ChatGPT。。。。我自己没看懂）：

代码	解读
def generate_mask(self, src, tgt):	`generate_mask` 是一个函数，接受两个参数 `src` 和 `tgt`，它的目的是生成用于Transformer模型的输入和输出的掩码。
	`src_mask` 和 `tgt_mask` 是两个掩码，用于指示输入和输出序列中哪些位置是有效的，哪些位置是无效的（例如，填充位置）。
src_mask = (src != 0).unsqueeze(1).unsqueeze(2)	`src_mask` 通过检查 `src` 是否不等于0来生成，然后对其进行维度扩展，使其变成一个三维的张量。这个掩码用于输入序列（通常是源语言文本）。
tgt_mask = (tgt != 0).unsqueeze(1).unsqueeze(3)	`tgt_mask` 通过检查 `tgt` 是否不等于0来生成，然后对其进行维度扩展，使其变成一个四维的张量。这个掩码用于输出序列（通常是目标语言文本）。
seq_length = tgt.size(1)	`seq_length` 是目标序列的长度，通过检查 `tgt` 的第一个维度的大小得到。
nopeak_mask = (1 - torch.triu(torch.ones(1, seq_length, seq_length), diagonal=1)).bool()	`nopeak_mask` 是一个掩码，它的作用是生成一个上三角形的掩码矩阵，该矩阵用于在自注意力机制中阻止模型查看未来的信息。这个掩码通过 `torch.triu` 函数生成，其中 `diagonal=1` 表示只保留主对角线以上的元素，并将其全部设置为1。然后使用 `bool()` 将其转换为布尔值类型。
tgt_mask = tgt_mask & nopeak_mask	最后，`tgt_mask` 与 `nopeak_mask` 进行按位与操作，将 `tgt_mask` 中的一些位置标记为无效，以便在解码时不会看到未来的信息。

总之，这个函数的主要目的是生成用于Transformer模型的输入和输出序列的掩码，确保模型在处理序列数据时能够正确地处理无效位置，同时在解码时不会看到未来的信息。这是Transformer模型中关键的一部分，用于自注意力机制和位置编码。

6.Transformer的训练过程

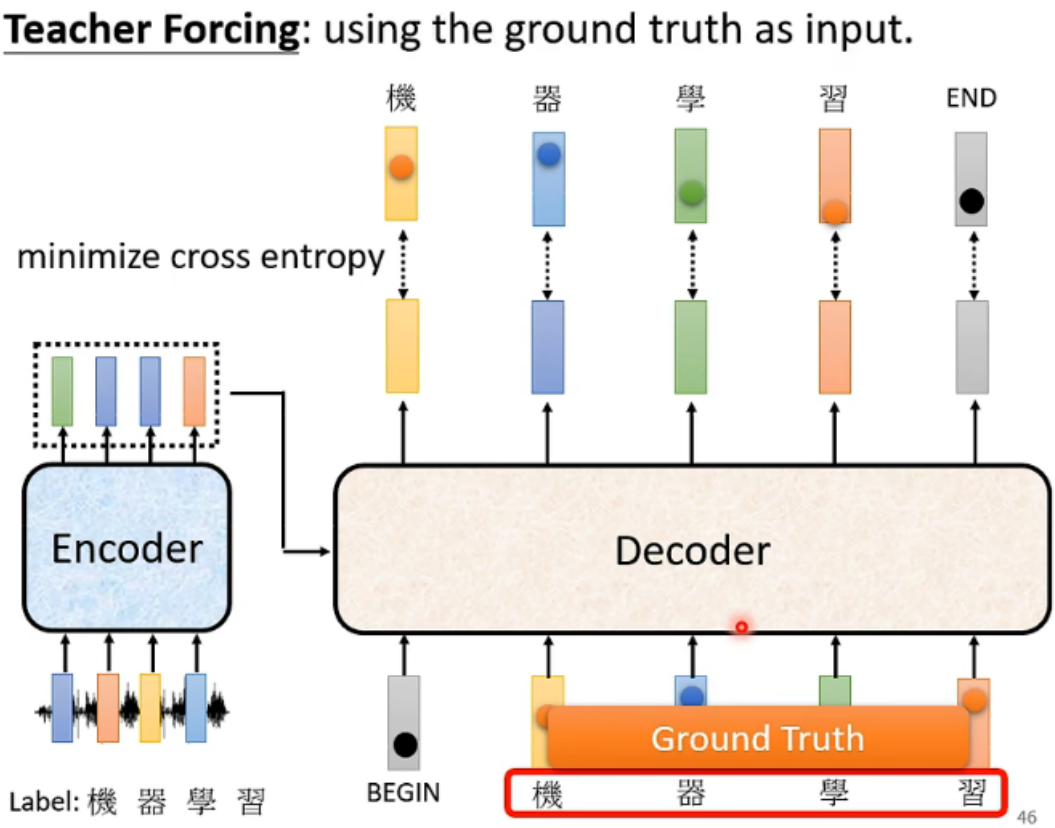
可以参考的链接：浅析Transformer训练时并行问题 - 知乎 (zhihu.com)

这里介绍一下Transformer的训练细节。

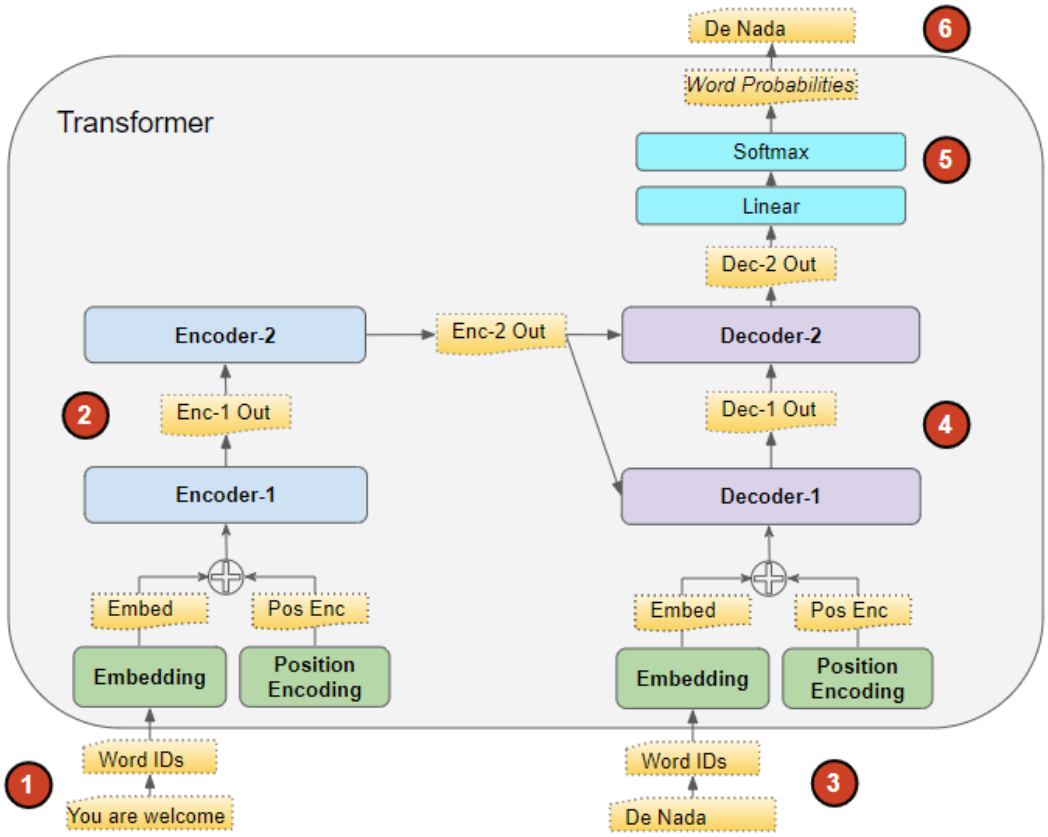
step1：标注好每组训练数据和对应的正确答案，即ground truth。

step2：将decoder输出的每个结果与ground truth做一次cross entropy，目标是让所有cross entropy的总和最小。（注意最后一个输出的one-hot要和END向量做一次cross entropy）。

如下图：



也就是说，在Decoder进行训练的时候我们要给它看正确答案。但这里有个问题，就是Decoder看到了正确答案但测试的时候没有正确答案，这个问题一会会说。**训练时的这个过程也被称为Teacher Forcing**，具体而言，每次喂给Decoder的序列都是正确的序列，即使decoder可能会推测错误之前的单词。而这可以通过并行化来解决，直接将原来的ground truth+mask喂给decoder，具体见mask部分的分析。

> 具体过程如下：
>
> 
>
>
> 1. The input sequence is converted into Embeddings (with Position Encoding) and fed to the Encoder.

> 2. The stack of Encoders processes this and produces an encoded representation of the input sequence.
> 3. The target sequence is prepended with（前面加） a start-of-sentence token, converted into Embeddings (with Position Encoding), and fed to the Decoder.
> 4. The stack of Decoders processes this along with the Encoder stack's encoded representation to produce an encoded representation of the target sequence.
> 5. The Output layer converts it into word probabilities and the final output sequence.
> 6. The Transformer's Loss function compares this output sequence with the target sequence from the training data. This loss is used to generate gradients to train the Transformer during back-propagation.

Teacher Forcing的意义如下：

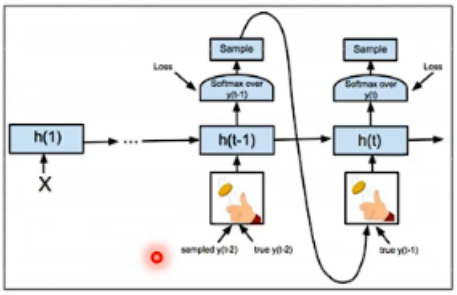
> 在训练期间，我们可以使用与测试期间使用的方法相同的方法。换句话说，在循环中运行Transformer，从输出序列中获取最后一个单词，将其附加到Decoder输入，并将其提供给Decoder以进行下一次迭代。最后，当预测到句尾标记时，Loss函数将生成的输出序列与目标序列进行比较，以训练网络。
>
> 这种循环不仅会导致训练花费更长的时间，而且还会使训练模型变得更加困难。该模型必须基于可能错误的第一个预测单词来预测第二个单词，并以此类推。相反，通过将目标序列提供给解码器，我们给了它一个提示，即使它预测了错误的第一个单词，它也可以用正确的第一个单词来预测第二个单词，这样这些错误就不会不断加剧。
>
> 此外，Transformer能够并行输出所有单词而不循环，这大大加快了训练速度。

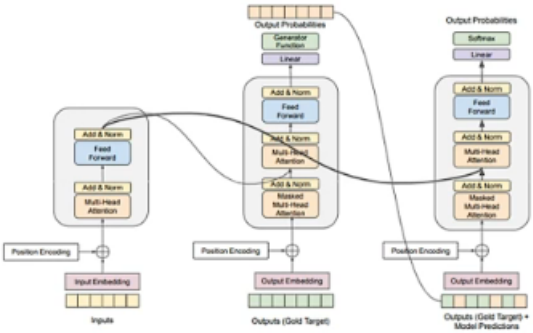
一些tips

- （1）copy mechanism：可以用在聊天机器人，做文章的摘要。最早具有这种能力的模型叫做Pointer Network，有时间可以了解一下。
- （2）Guided Attention：常用于语音辨识，语音合成领域。
- （3）Beam Search：原理：What is Beam Search? Explaining The Beam Search Algorithm | Width.ai，有时有用，有时没用。如果任务的答案比较死，可以用Beam Search来帮助找到更好的解。否则如果想让机器产生一些随机性，就可以不用Beam Search。
- （4）之前在Decoder的部分，有提及可能会出现Decoder在训练的时候看到的都是正确答案，这会造成如果测试的时候产生了错误的结果，就很可能继续错下去，对此的解决方案可以是训练的时候给decoder喂一些错误的数据，感兴趣可以参考下图找文献阅读：

Scheduled Sampling

- Original Scheduled Sampling
  https://arxiv.org/abs/1506.03099
- Scheduled Sampling for Transformer
  https://arxiv.org/abs/1906.07651
- Parallel Scheduled Sampling
  https://arxiv.org/abs/1906.04331





相关代码——以随机生成数据集为例

```
src_vocab_size = 5000
tgt_vocab_size = 5000
d_model = 512
num_heads = 8
num_layers = 6
d_ff = 2048
max_seq_length = 100
dropout = 0.1

transformer = Transformer(src_vocab_size, tgt_vocab_size, d_model, num_heads,
num_layers, d_ff, max_seq_length, dropout)

# Generate random sample data
src_data = torch.randint(1, src_vocab_size, (64, max_seq_length))  #
(batch_size, seq_length)
tgt_data = torch.randint(1, tgt_vocab_size, (64, max_seq_length))  #
(batch_size, seq_length)

# 初始化一个transformer网络
transformer = Transformer(src_vocab_size, tgt_vocab_size, d_model, num_heads,
num_layers, d_ff, max_seq_length, dropout)

# training the data
criterion = nn.CrossEntropyLoss(ignore_index=0)
optimizer = optim.Adam(transformer.parameters(), lr=0.0001, betas=(0.9, 0.98),
eps=1e-9)

transformer.train()

for epoch in range(100):
    optimizer.zero_grad()
```

```
    output = transformer(src_data, tgt_data[:, :-1])  #  Passes the source data
and the target data (excluding the last token in each sequence) through the
transformer. This is common in sequence-to-sequence tasks where the target is
shifted by one token.
    loss = criterion(output.contiguous().view(-1, tgt_vocab_size), tgt_data[:,
1:].contiguous().view(-1))  #  Computes the loss between the model's predictions
and the target data (excluding the first token in each sequence). The loss is
calculated by reshaping the data into one-dimensional tensors and using the
cross-entropy loss function.
    loss.backward()
    optimizer.step()
    print(f"Epoch: {epoch+1}, Loss: {loss.item()}")

# eval the model:
transformer.eval()

# Generate random sample validation data
val_src_data = torch.randint(1, src_vocab_size, (64, max_seq_length))  #
(batch_size, seq_length)
val_tgt_data = torch.randint(1, tgt_vocab_size, (64, max_seq_length))  #
(batch_size, seq_length)

with torch.no_grad():

    val_output = transformer(val_src_data, val_tgt_data[:, :-1])
    val_loss = criterion(val_output.contiguous().view(-1, tgt_vocab_size),
val_tgt_data[:, 1:].contiguous().view(-1))
    print(f"Validation Loss: {val_loss.item()}")
```

更为具体的任务放在后面再进行描述。

7.李沐精读——Transformer

Transformer论文逐段精读【论文精读】*哔哩哔哩*bilibili

（1）LayerNorm相关

这里有两个问题：

- LayerNorm和BatchNorm的区别是什么？
- 为什么Transformer使用的是LayerNorm而不是BatchNorm，LayerNorm有什么优势？

对应视频的25分钟开始到32分钟的部分。

关于Self-attention其他可参考的资料

Transformer常见问题与回答总结 - 知乎 (zhihu.com)

GitHub 7.5k star量，各种视觉Transformer的PyTorch实现合集整理好了 | 机器之心 (jiqizhixin.com)

UKPLab/sentence-transformers: Multilingual Sentence & Image Embeddings with BERT (github.com)

https://ketanhdoshi.github.io/Transformers-Overview/

关于Transformer的一些问题（比如面试问）

三、ViT

相关链接：[2010.11929] An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale (arxiv.org)

Future work：

- BERT
- 训练一个transformer在某个数据集上，这个再说（深度学习的作业三就是做这个任务）
- VIT论文阅读，并看一下开源代码lucidrains/vit-pytorch: Implementation of Vision Transformer, a simple way to achieve SOTA in vision classification with only a single transformer encoder, in Pytorch (github.com)