

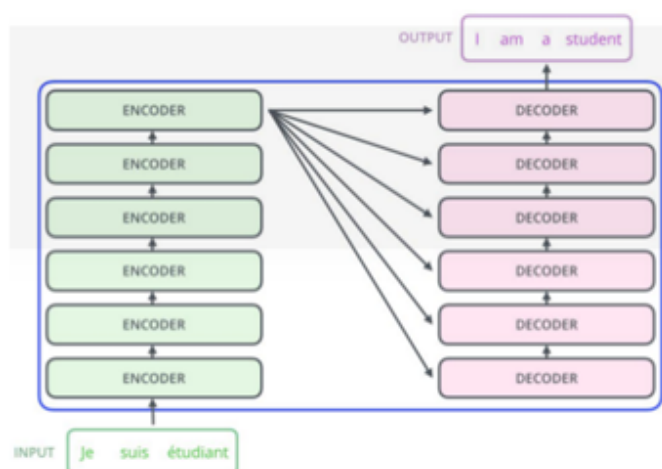
# Transformer

Transformer是Google于2017年提出的seq2seq模型，结构上它摒弃了RNN和CNN而全面拥抱注意力，这样不仅大幅提高了序列建模的有效长度，还通过并行计算提高了训练效率。Transformer为后来预训练的崛起奠定了扎实的基础。

关键词：预训练；注意力；

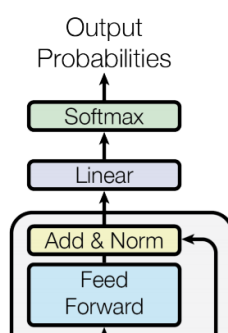
## 1 模型

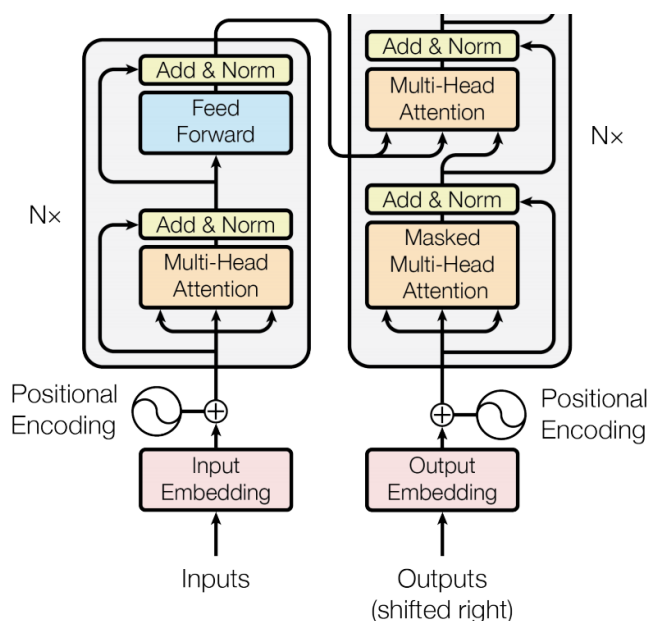
Transformer分为Encoder和Decoder两个部分，二者分别由六个相同的Encoder层和Decoder层堆积而成，encoder自底向上逐层计算至最顶层后，得到上下文向量，随后decoder层也自底向上逐层计算（其中**每个decoder层的输入既包括上一个decoder层的输出，还包括上下文向量**），最终得到模型输出。如下图所示：



Transformer整体架构

而在每一个Encoder层内部的网络层有**多头注意力层**（Multi-Head Attention）、**前向传播层**（Feed Forward），以及二者之间穿插的**残差&归一化层**（Add&Norm），每一个Decoder层内部和Encoder大体一样，区别在于Decoder除了多头自注意力层，还有一个**遮掩多头注意力层**（Masked Multi-Head Attention），如下图所示：





Transformer Encoder层和Decoder层内部结构

接下来，按照从输入到输出的顺序介绍各个核心网络层中的计算细节。

## 1.1 输入和位置编码

由于摒弃了RNN和CNN，序列每个位置上仅仅输入token embedding会不可避免地缺失位置信息，于是Transformer把token embedding和位置编码（Positional Embedding）加在一起作为最终的Input embedding，二者的维度都是  $d_{model}$ 。

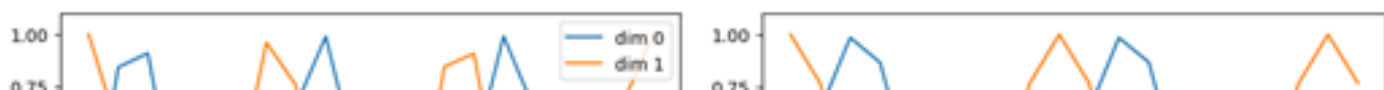
对于输入的embedding，Transformer使用的是预训练的token embedding，Encoder和Decoder共享同一个Embedding Matrix；

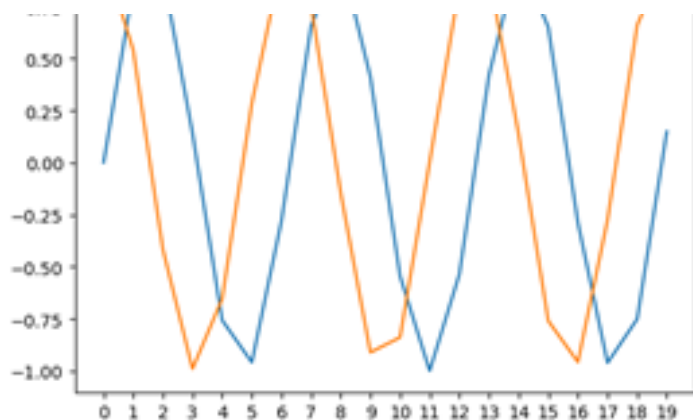
而对于位置编码，Transformer采用**固定的**位置编码，计算公式为：

$$PE_{pos,2i} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

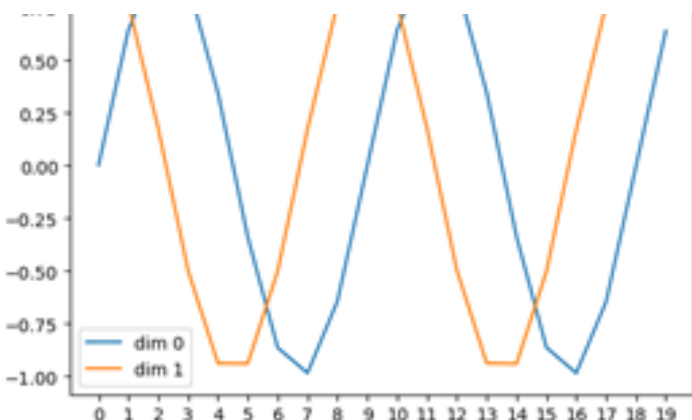
$$PE_{pos,2i+1} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

其中  $pos$  表示序列位置， $i$  表示维度。由此可见，在同一个维度上（即  $i$  一定），位置嵌入的取值是通过一个正弦或者余弦函数计算得到的。只不过随着维度的增大，正弦（余弦）函数的周期从  $2\pi$  增大到  $10000 \cdot 2\pi$ ，并且相邻两个维度（0和1、2和3，依次类推）上的函数的周期相同。公式不是很直观，可以用代码画出来。

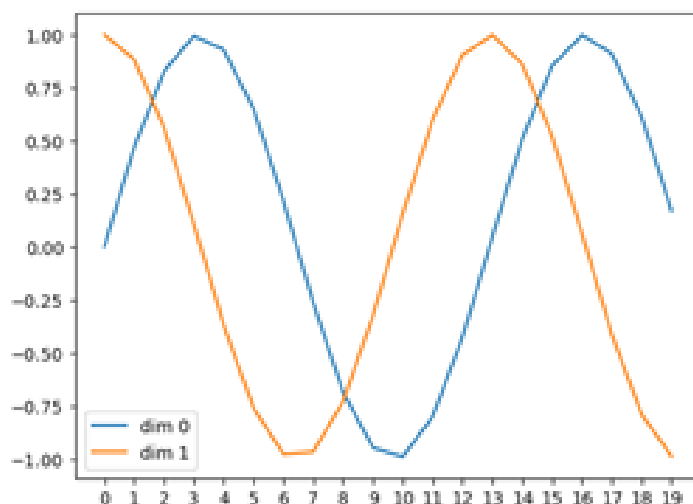




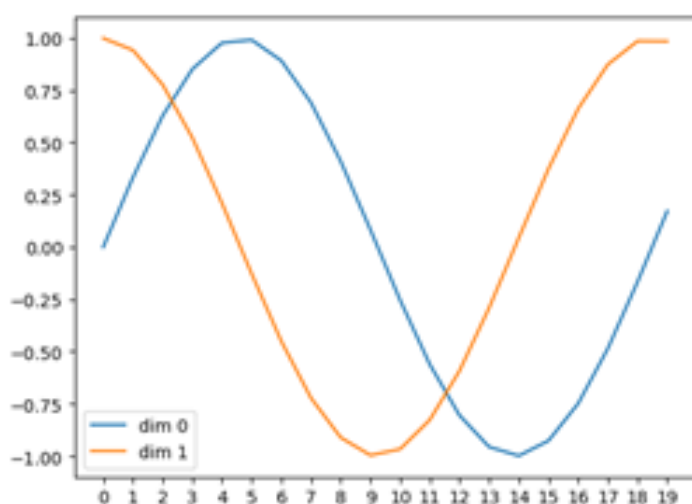
位置嵌入在第0维和第1维上的取值



位置嵌入在第20维和第21维上的取值



位置嵌入在第40维和第41维上的取值



位置嵌入在第60维和第61维上的取值

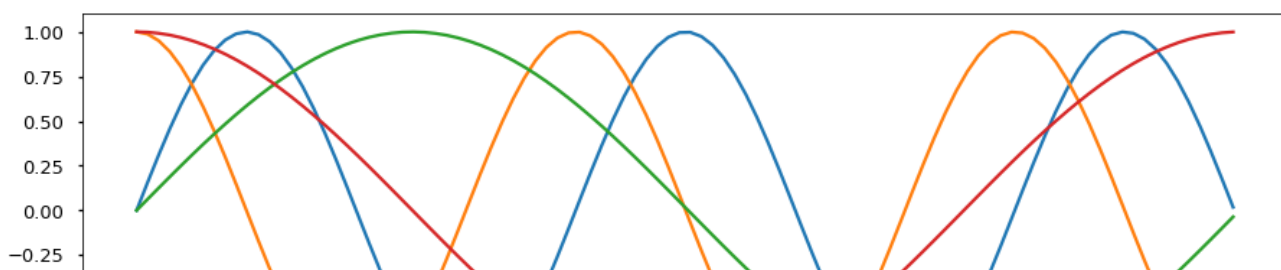
上面四个图中的横坐标均表示位置，由此可见位置编码在奇数维度上的取值可以看成离散化的余弦曲线，在偶数维度上的取值可以看成是离散化的正弦曲线。当  $2i = d_{model}$  时，周期最大，这里的  $d_{model}$  是模型中每个编解码层的输入输出的统一长度，也是token embedding的维度。

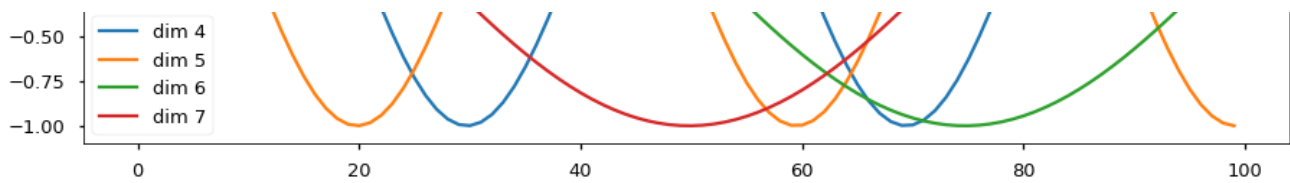
位置编码这种设计的目的是为了**让模型更好地学习到相对位置**。对于位置偏移量  $k$ ， $PE_{pos+k}$  都可以由  $PE_{pos}$  的线性函数表示。不妨假设当前维度是偶数，令  $m = 10000^{-2i/d_{model}}$ ，则

$$PE_{pos+k} = \sin(m \cdot (pos + k)) = \sin(m \cdot pos) \cos(m \cdot k) + \cos(m \cdot pos) \sin(m \cdot k)$$

其中  $\cos(m \cdot k)$ 、 $\sin(m \cdot k)$  都是常数， $\cos(m \cdot pos)$  就是下一个维度相同位置上的位置编码取值。

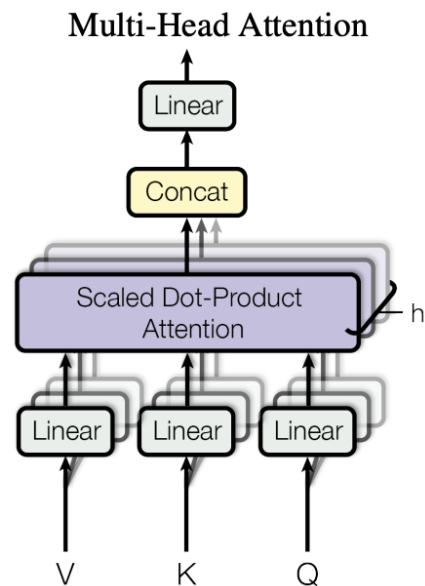
这里也提供别人画的位置编码





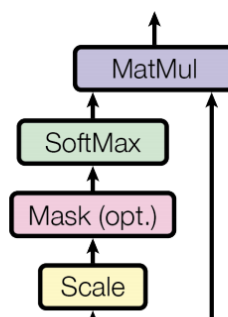
## 1.2 多头注意力

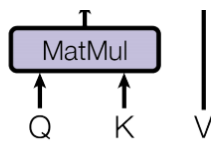
多头注意力是整个Transformer的核心。Transformer之所以能实现高效的并行计算，多头注意力功不可没。多头指的将当前的输入分成多个相同的计算分支，最后每个分支的计算结果将被拼接在一起作为完整的输出，如下图(a)所示。多头注意力的核心部分是缩放点乘注意力（Scaled Dot-Product Attention），如下图(b)所示。它和Luong提出的全局注意力一样，只不过这里的符号表示稍有变动。回顾基于注意力的编解码模型，图中的K、V就是Encoder中的隐藏状态（也称source hidden state），而Q是Decoder中的隐藏状态（target hidden state）。注意力首先基于Q、K计算出每个K的注意力权重，然后再将其作用在V上计算加权和。



多头注意力的多头分支计算和合并

### Scaled Dot-Product Attention





### 缩放点乘注意力

值得注意的是，缩放点乘注意力有三点改进：

- 第一是**加入了缩放来控制数值的取值范围**，避免QK相乘后数值过大降低了计算效率，假设q和k是两个独立的均值为0、方差为1的随机变量，那么二者的点乘  $q \cdot k = \sum_i q_i k_i$  是均值为0、方差为  $d_k$  的随机变量；
- 第二是**加入了注意力遮掩机制**。这个是Decoder层才有的（即Masked Multi-Head Attention和Multi-Head Attention的区别所在），目的是防止在解码时注意力“看到”了当前预测位置后面的字符。实现办法就是在对注意力分数进行归一化（Softmax）之前，将后面的当前解码位置之后的注意力分数全部设置为负无穷大，这样归一化之后的注意力权重就为零；
- 第三，**用矩阵计算代替向量计算**，将所有source hidden state和target hidden state组成Q、K、V三个矩阵，在编码时，Q、K、V是相同的，包含所有source hidden state，是一种自注意力计算；在解码时，仅Q发生改变，它变成由所有target hidden stated组成的矩阵。

整个计算过程如下所示：


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

$$MultiHead(Q, K, V) = Concat(head_1, ..., head_h)W^o$$

$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$$

线性层参数  $(W_i^Q, W_i^K \in \mathbb{R}^{d_{model} \times d_k}, W_i^V \in \mathbb{R}^{d_{model} \times d_v})$  首先将维度为  $d_{model}$  的输入向量  $Q$ 、 $K$ 、 $V$  分别压缩到  $d_k$ 、 $d_k$ 、 $d_v$ 。随后各个分支的缩放点乘注意力计算出结果  $head_i$ ，最后经过拼接和线性层的计算得到输出。其中隐含条件是  $d_{model} = h \times d_v$ 。

由于设置了残差连接，所以模型内部各个层的输入输出的维度始终维持在  $d_{model}$ 。

 **深度思考：**在multihead attention中，注意力分多头会增加参数数量和计算量吗？多头注意力模块的参数数量是多少？

答案是不会增加参数量和计算量，参数主要集中在三个线性层，如果考虑偏置，为 $3 * D * (D+1)$

分析如下：

自注意力层的输入和输出的shape是一模一样的，都是 $[B, L, D]$

B表示batch size

L表示max sequence length

D表示d\_model，即dimension of embeddings，以bert-base为例， $D=768$

由于是多头注意力，所以会将D拆成num\_heads个头，每个头上的嵌入维度为head\_size

以Bert-base为例，num\_heads=12, head\_size=64，有 $12 * 64 = 768$

在自注意力层中，通过三个线性层将hidden\_states转成Q、K、V，参数为 $3 * D * D$

```
query_layer = nn.Linear(D, self.all_head_size)
```

```
key_layer = nn.Linear(D, self.all_head_size)
```

```
value_layer = nn.Linear(D, self.all_head_size)
```

除此之外，还有一个dropout层

```
dropout = nn.Dropout(attention_probs_dropout_prob)
```

下面是整个多头自注意力的计算

假设输入为x，先通过query\_layer, key\_layer, value\_layer分别将x转成Q, K, V，

此时三者的size为：

Q:  $[B, L, \text{all\_head\_size}]$

K:  $[B, L, \text{all\_head\_size}]$

V:  $[B, L, \text{all\_head\_size}]$

这里的all\_head\_size = num\_heads \* head\_size

然后分别将Q, K, V进行reshape：

Q:  $[B, \text{num\_heads}, L, \text{head\_size}]$

K:  $[B, \text{num\_heads}, \text{head\_size}, L]$

V:  $[B, \text{num\_heads}, L, \text{head\_size}]$

然后将Q和 $K^T$ 相乘后除以参数，得到注意力权重

```
x = torch.matmul(Q, K) # [B, num_heads, L, L]
```

```
x = x.div(math.sqrt(head_size))
```

```
x = x.div(math.sqrt(head_size))
```

然后进行softmax得到注意力分数

```
x = nn.functional.softmax(x, dim=-1)
```

这一步很奇怪，但是原论文就这么做的，在注意力分数后dropout

```
x = dropout(x)
```

然后乘上V

```
x = torch.matmul(x, V) # [B, num_heads, L, head_size]
```

接下来还是reshape

```
x: [B, num_heads, L, head_size] -> [B, L, num_heads, head_size] -> [B, L, D]
```

### 1.3 残差和层归一化

每个Encoder层内部包含了Multi-Head Attention和Point-Wise Fully Connected Feed-Forward Network两个子层，每个子层的输入和输出都设置了残差连接，后接一个层归一化（Layer Normalization），于是对于每一个子层都有：

$$LayerNorm(x + sublayer(x))$$

### 1.4 位置级前向传播

位置级前向传播（Position-wise Feed-Forward）是两层全连接网络，中间用一个ReLU激活层连接。它作用在输出的每个位置上。

$$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2)$$

$W_1$  将  $d_{model}$  先扩张到2048维，然后再压回到512.

## 2 代码阅读

结构总览

## Python

```
1 class EncoderDecoder(nn.Module):
2     """
3     A standard Encoder-Decoder architecture. Base for this and many
4     other models.
5     """
6     def __init__(self, encoder, decoder, src_embed, tgt_embed, generator):
7         super(EncoderDecoder, self).__init__()
8         self.encoder = encoder
9         self.decoder = decoder
10        self.src_embed = src_embed
11        self.tgt_embed = tgt_embed
12        self.generator = generator
13
14    def forward(self, src, tgt, src_mask, tgt_mask):
15        "Take in and process masked src and target sequences."
16        return self.decode(self.encode(src, src_mask), src_mask,
17                           tgt, tgt_mask)
18
19    def encode(self, src, src_mask):
20        return self.encoder(self.src_embed(src), src_mask)
21
22    def decode(self, memory, src_mask, tgt, tgt_mask):
23        return self.decoder(self.tgt_embed(tgt), memory, src_mask, tgt_mask)
24
25
26 class Generator(nn.Module):
27     "Define standard linear + softmax generation step."
28     def __init__(self, d_model, vocab):
29         super(Generator, self).__init__()
30         self.proj = nn.Linear(d_model, vocab)
31
32     def forward(self, x):
33         return F.log_softmax(self.proj(x), dim=-1)
```

接下来是encoder

## Python

```
1 def clones(module, N):
2     "Produce N identical layers."
3     return nn.ModuleList([copy.deepcopy(module) for _ in range(N)])
4
5
6 class LayerNorm(nn.Module):
7     "Construct a layernorm module (See citation for details)."
8     def __init__(self, features, eps=1e-6):
```



```

8     def __init__(self, features, eps=1e-6):
9         super(LayerNorm, self).__init__()
10        self.a_2 = nn.Parameter(torch.ones(features))
11        self.b_2 = nn.Parameter(torch.zeros(features))
12        self.eps = eps
13
14    def forward(self, x):
15        mean = x.mean(-1, keepdim=True)
16        std = x.std(-1, keepdim=True)
17        return self.a_2 * (x - mean) / (std + self.eps) + self.b_2
18
19
20    class SublayerConnection(nn.Module):
21        """
22        A residual connection followed by a layer norm.
23        Note for code simplicity the norm is first as opposed to last.
24        """
25        def __init__(self, size, dropout):
26            super(SublayerConnection, self).__init__()
27            self.norm = LayerNorm(size)
28            self.dropout = nn.Dropout(dropout)
29
30        def forward(self, x, sublayer):
31            "Apply residual connection to any sublayer with the same size."
32            return x + self.dropout(sublayer(self.norm(x)))
33
34
35    class PositionwiseFeedForward(nn.Module):
36        "Implements FFN equation."
37        def __init__(self, d_model, d_ff, dropout=0.1):
38            super(PositionwiseFeedForward, self).__init__()
39            self.w_1 = nn.Linear(d_model, d_ff)
40            self.w_2 = nn.Linear(d_ff, d_model)
41            self.dropout = nn.Dropout(dropout)
42
43        def forward(self, x):
44            return self.w_2(self.dropout(F.relu(self.w_1(x))))
45
46
47    class EncoderLayer(nn.Module):
48        "Encoder is made up of self-attn and feed forward (defined below)"
49        def __init__(self, size, self_attn, feed_forward, dropout):
50            super(EncoderLayer, self).__init__()
51            self.self_attn = self_attn
52            self.feed_forward = feed_forward
53            self.sublayer = clones(SublayerConnection(size, dropout), 2)
54            self.size = size
55
56        def forward(self, x, mask):

```

```

57         "Follow Figure 1 (left) for connections."
58         x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, mask))
59         return self.sublayer[1](x, self.feed_forward)
60
61
62     class Encoder(nn.Module):
63         "Core encoder is a stack of N layers"
64         def __init__(self, layer, N):
65             super(Encoder, self).__init__()
66             self.layers = clones(layer, N) # 复制12个encoder层
67             self.norm = LayerNorm(layer.size) # LN层
68
69         def forward(self, x, mask):
70             "Pass the input (and mask) through each layer in turn."
71             for layer in self.layers:
72                 x = layer(x, mask)
73             return self.norm(x)

```

接下来是缩放点乘注意力和多头自注意力，encoder和decoder共享这里的代码

## Python

```
1 def attention(query, key, value, mask=None, dropout=None):
2     "Compute 'Scaled Dot Product Attention'"
3     d_k = query.size(-1)
4     scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d_k)
5     if mask is not None:
6         scores = scores.masked_fill(mask == 0, -1e9) # 用负无穷大填充
7     p_attn = F.softmax(scores, dim = -1)
8     if dropout is not None:
9         p_attn = dropout(p_attn)
10    return torch.matmul(p_attn, value), p_attn
11
12
13 class MultiHeadedAttention(nn.Module):
14     def __init__(self, h, d_model, dropout=0.1):
15         "Take in model size and number of heads."
16         super(MultiHeadedAttention, self).__init__()
17         assert d_model % h == 0
18         # We assume d_v always equals d_k
19         self.d_k = d_model // h
20         self.h = h
21         self.linears = clones(nn.Linear(d_model, d_model), 4)
22         self.attn = None
23         self.dropout = nn.Dropout(p=dropout)
24
25     def forward(self, query, key, value, mask=None):
26         if mask is not None:
27             # Same mask applied to all h heads.
28             mask = mask.unsqueeze(1)
29             nbatches = query.size(0)
30
31             # 1) Do all the linear projections in batch from d_model => h x d_k
32             query, key, value = \
33                 [l(x).view(nbatches, -1, self.h, self.d_k).transpose(1, 2)
34                  for l, x in zip(self.linears, (query, key, value))]
35
36             # 2) Apply attention on all the projected vectors in batch.
37             x, self.attn = attention(query, key, value, mask=mask,
38                                     dropout=self.dropout)
39
40             # 3) "Concat" using a view and apply a final linear.
41             x = x.transpose(1, 2).contiguous() \
42                 .view(nbatches, -1, self.h * self.d_k)
43             return self.linears[-1](x)
```

接下来是decoder

## Ruby

```
1 class Decoder(nn.Module):
2     "Generic N layer decoder with masking."
3     def __init__(self, layer, N):
4         super(Decoder, self).__init__()
5         self.layers = clones(layer, N)
6         self.norm = LayerNorm(layer.size)
7
8     def forward(self, x, memory, src_mask, tgt_mask):
9         for layer in self.layers:
10             x = layer(x, memory, src_mask, tgt_mask)
11         return self.norm(x)
12
13
14 class DecoderLayer(nn.Module):
15     "Decoder is made of self-attn, src-attn, and feed forward (defined below)"
16     def __init__(self, size, self_attn, src_attn, feed_forward, dropout):
17         super(DecoderLayer, self).__init__()
18         self.size = size
19         self.self_attn = self_attn
20         self.src_attn = src_attn
21         self.feed_forward = feed_forward
22         self.sublayer = clones(SublayerConnection(size, dropout), 3)
23
24     def forward(self, x, memory, src_mask, tgt_mask):
25         "Follow Figure 1 (right) for connections."
26         m = memory
27         x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, tgt_mask))
28         x = self.sublayer[1](x, lambda x: self.src_attn(x, m, m, src_mask))
29         return self.sublayer[2](x, self.feed_forward)
30
31 # cross attention mask
32 def subsequent_mask(size):
33     "Mask out subsequent positions."
34     attn_shape = (1, size, size)
35     subsequent_mask = np.triu(np.ones(attn_shape), k=1).astype('uint8')
36     return torch.from_numpy(subsequent_mask) == 0
```

接下来是embedding，encoder输入时的embedding矩阵和decoder预测时的embedding矩阵共享参数：

## Python

```
1 class Embeddings(nn.Module):
2     def __init__(self, d_model, vocab):
3         super(Embeddings, self).__init__()
4         self.lut = nn.Embedding(vocab, d_model)
5         self.d_model = d_model
6
7     def forward(self, x):
8         return self.lut(x) * math.sqrt(self.d_model)
9
10
11 class PositionalEncoding(nn.Module):
12     "Implement the PE function."
13     def __init__(self, d_model, dropout, max_len=5000):
14         super(PositionalEncoding, self).__init__()
15         self.dropout = nn.Dropout(p=dropout)
16
17         # Compute the positional encodings once in log space.
18         pe = torch.zeros(max_len, d_model)
19         position = torch.arange(0, max_len).unsqueeze(1)
20         div_term = torch.exp(torch.arange(0, d_model, 2) *
21                               -(math.log(10000.0) / d_model))
22         pe[:, 0::2] = torch.sin(position * div_term)
23         pe[:, 1::2] = torch.cos(position * div_term)
24         pe = pe.unsqueeze(0)
25         self.register_buffer('pe', pe)
26
27     def forward(self, x):
28         x = x + Variable(self.pe[:, :x.size(1)],
29                           requires_grad=False)
30         return self.dropout(x)
```

## 最后汇总

## Python

```
1 def make_model(src_vocab, tgt_vocab, N=6,
2               d_model=512, d_ff=2048, h=8, dropout=0.1):
3     "Helper: Construct a model from hyperparameters."
4     c = copy.deepcopy
5     attn = MultiHeadedAttention(h, d_model)
6     ff = PositionwiseFeedForward(d_model, d_ff, dropout)
7     position = PositionalEncoding(d_model, dropout)
8     model = EncoderDecoder(
9         Encoder(EncoderLayer(d_model, c(attn), c(ff), dropout), N),
10        Decoder(DecoderLayer(d_model, c(attn), c(attn),
11                               c(ff), dropout), N),
12        nn.Sequential(Embeddings(d_model, src_vocab), c(position)),
13        nn.Sequential(Embeddings(d_model, tgt_vocab), c(position)),
14        Generator(d_model, tgt_vocab))
15
16     # This was important from their code.
17     # Initialize parameters with Glorot / fan_avg.
18     for p in model.parameters():
19         if p.dim() > 1:
20             nn.init.xavier_uniform(p)
21     return model
```

## 3 总结

Transformer的核心包括

- 多头自注意力(encoder/decoder masked)
- 位置编码
- layer normalization

1. **优点：**Transformer的序列建模能力（尤其是在长序列上）超越了RNN、CNN，凭借的就是注意力机制。和逐步传播相比，它直接将任意两个位置的隐藏状态连接起来，所以长期依赖信息全都可以捕捉到；另外，多头注意力的并行计算有效缓解了注意力的高复杂度  $O(n^2)$ ；
2. **缺点：**Transformer存在上下文分裂问题，可参考Transformer-XL。

近几年，不仅仅是GPT、BERT等预训练模型将其作为基本网络层，机器翻译、阅读理解和命名实体识别等其他任务模型也开始将其作为基本的特征提取器。由此可见，Transformer俨然已经成为新的序列建模大杀器。

## 4 参考文献

1. [Attention Is All You Need](#). Ashish Vaswani. NIPS 2017

## 2. The Annotated Transformer