Transformer

特点

- CNN 编码局部信息，没有全局上下文
- RNN编码全局信息，不能并行
- Transformer能并行，编码全局信息

图的左半边用 `Nx` 框出来的，就是我们的encoder的一层。encoder一共有6层这样的结构。

图的右半边用 `Nx` 框出来的，就是我们的decoder的一层。decoder一共有6层这样的结构。

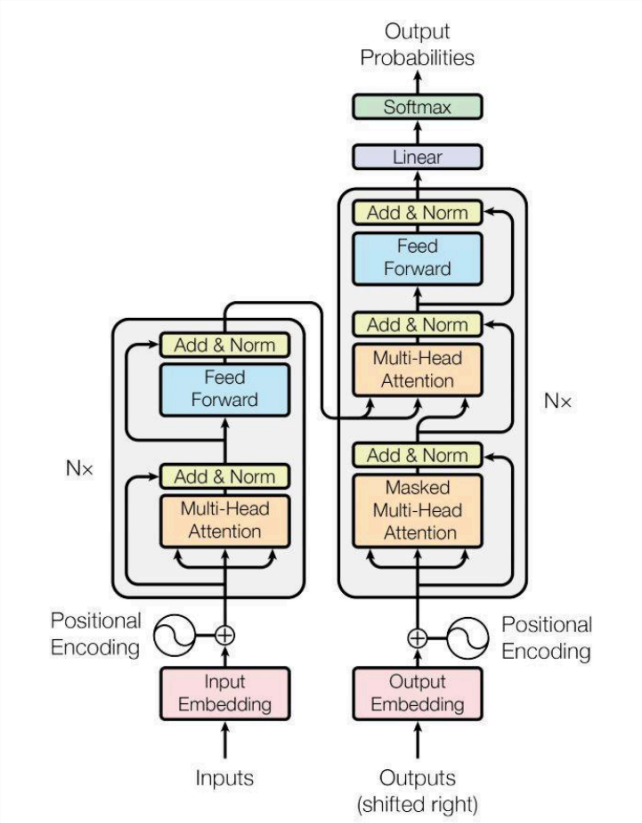


Figure 1: The Transformer - model architecture.

Encoder

encoder由6层相同的层组成，每一层分别由两部分组成：

- 第一部分是一个**multi-head self-attention mechanism**
- 第二部分是一个**position-wise feed-forward network**，是一个全连接层

两个部分，都有一个 **残差连接(residual connection)**，然后接着一个**Layer Normalization**。

Decoder

decoder由6个相同的层组成，每一个层包括以下3个部分：

- 第一个部分是**multi-head self-attention mechanism**
- 第二部分是**multi-head context-attention mechanism**
- 第三部分是一个**position-wise feed-forward network**

还是和encoder类似，上面三个部分的每一个部分，都有一个**残差连接**，后接一个**Layer Normalization**。

Self Attention

到目前为止，对Attention层的描述都是一般化的，我们可以落实一些应用。比如，如果做阅读理解的话，Q可以是篇章的词向量序列，取K=V为问题的词向量序列，那么输出就是所谓的Aligned Question Embedding。

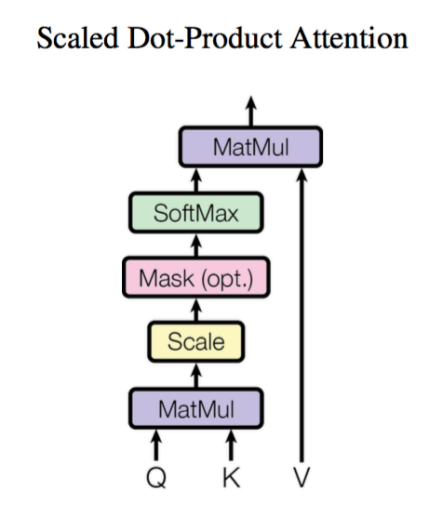
而在Google的论文中，大部分的Attention都是Self Attention，即"自注意力"，或者叫内部注意力。

所谓Self Attention，其实就是$Attention(X,X,X)$，X就是前面说的输入序列。也就是说，在序列内部做Attention，寻找序列内部的联系。Google论文的主要贡献之一是它表明了内部注意力在机器翻译（甚至是一般的Seq2Seq任务）的序列编码上是相当重要的，而之前关于Seq2Seq的研究基本都只是把注意力机制用在解码端。类似的事情是，目前SQUAD阅读理解的榜首模型R-Net也加入了自注意力机制，这也使得它的模型有所提升。

Context-attention

它是encoder和decoder之间的attention！所以，你也可以称之为**encoder-decoder attention**!

Attention定义



$$Attention(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V}) = softmax\left(\frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{d_k}}\right)\boldsymbol{V}$$

其中$\boldsymbol{Q} \in \mathbb{R}^{n \times d_k}, \boldsymbol{K} \in \mathbb{R}^{m \times d_k}, \boldsymbol{V} \in \mathbb{R}^{m \times d_v}$，于是我们可以认为：这是一个Attention层，将$n \times d_k$的序列Q编码成了一个新的$n \times d_v$的序列.

首先说明一下我们的K、Q、V是什么：

- 在encoder的self-attention中，Q、K、V都来自同一个地方（相等），他们是上一层encoder的输出。对于第一层encoder，它们就是word embedding和positional encoding相加得到的输入。
- 在decoder的self-attention中，Q、K、V都来自于同一个地方（相等），它们是上一层decoder的输出。对于第一层decoder，它们就是word embedding和positional encoding相加得到的输入。但是对于decoder，我们不希望它能获得下一个time step（即将来的信息），因此我们需要进行**sequence masking**。
- 在encoder-decoder attention中，Q来自于decoder的上一层的输出，K和V来自于encoder的输出，K和V是一样的。
- Q、K、V三者的维度一样，即 $d_q = d_k = d_v$.

Scaled dot-product attention的实现

```
class ScaledDotProductAttention(nn.Module):
    """Scaled dot-product attention mechanism."""

    def __init__(self, attention_dropout=0.0):
        super(ScaledDotProductAttention, self).__init__()
        self.dropout = nn.Dropout(attention_dropout)
        self.softmax = nn.Softmax(dim=2)

    def forward(self, q, k, v, scale=None, attn_mask=None):
        """前向传播.

        Args:
          q: Queries张量，形状为[B, L_q, D_q]
          k: Keys张量，形状为[B, L_k, D_k]
          v: Values张量，形为为[B, L_v, D_v]，一般来说就是k
          scale: 缩放因子，一个浮点标量
          attn_mask: Masking张量，形状为[B, L_q, L_k]

        Returns:
          上下文张量和attetention张量
        """
        attention = torch.bmm(q, k.transpose(1, 2))
        if scale:
          attention = attention * scale
        if attn_mask:
          # 给需要mask的地方设置一个负无穷
          attention = attention.masked_fill_(attn_mask, -np.inf)
    # 计算softmax
        attention = self.softmax(attention)
    # 添加dropout
        attention = self.dropout(attention)
    # 和v做点积
        context = torch.bmm(attention, v)
```

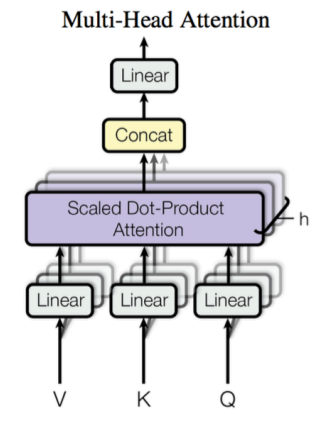
```
        return context, attention

```

Multi-Head Attention

论文提到，他们发现将Q、K、V通过一个线性映射之后，分成 h 份，对每一份进行**scaled dot-product attention**效果更好。然后，把各个部分的结果合并起来，再次经过线性映射，得到最终的输出。这就是所谓的**multi-head attention**。上面的超参数 h 就是**heads**数量。论文默认是 `8` 。

值得注意的是，上面所说的**分成 h 份**是在 d_k、d_q、d_v 维度上面进行切分的。因此，进入到scaled dot-product attention的 d_k 实际上等于未进入之前的 D_K/h 。



就是把$\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$ 通过参数矩阵映射一下，然后再做Attention，把这个过程重复做h次，结果拼接起来就行了，可谓“大道至简”了。

所谓“多头”（Multi-Head），就是只多做几次同样的事情（参数不共享），然后把结果拼接。

$$head_i = Attention(\boldsymbol{Q}\boldsymbol{W}_i^Q, \boldsymbol{K}\boldsymbol{W}_i^K, \boldsymbol{V}\boldsymbol{W}_i^V)$$
$$MultiHead(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = Concat(head_1, \ldots, head_h)$$

论文里面，$d_{model} = 512$，$h = 8$。所以在scaled dot-product attention里面的

$$d_q = d_k = d_v = d_{model}/h = 512/8 = 64$$

Multi-head attention的实现

```
class MultiHeadAttention(nn.Module):

    def __init__(self, model_dim=512, num_heads=8, dropout=0.0):
        super(MultiHeadAttention, self).__init__()

        self.dim_per_head = model_dim // num_heads
        self.num_heads = num_heads
        self.linear_k = nn.Linear(model_dim, self.dim_per_head *
num_heads)
```

```
        self.linear_v = nn.Linear(model_dim, self.dim_per_head * num_heads)
        self.linear_q = nn.Linear(model_dim, self.dim_per_head * num_heads)

        self.dot_product_attention = ScaledDotProductAttention(dropout)
        self.linear_final = nn.Linear(model_dim, model_dim)
        self.dropout = nn.Dropout(dropout)
    # multi-head attention之后需要做layer norm
        self.layer_norm = nn.LayerNorm(model_dim)

    def forward(self, key, value, query, attn_mask=None):
    # 残差连接
        residual = query

        dim_per_head = self.dim_per_head
        num_heads = self.num_heads
        batch_size = key.size(0)

        # linear projection
        key = self.linear_k(key)
        value = self.linear_v(value)
        query = self.linear_q(query)

        # split by heads
        key = key.view(batch_size * num_heads, -1, dim_per_head)
        value = value.view(batch_size * num_heads, -1, dim_per_head)
        query = query.view(batch_size * num_heads, -1, dim_per_head)

        if attn_mask:
            attn_mask = attn_mask.repeat(num_heads, 1, 1)
        # scaled dot product attention
        scale = (key.size(-1) // num_heads) ** -0.5
        context, attention = self.dot_product_attention(
          query, key, value, scale, attn_mask)
        # concat heads
        context = context.view(batch_size, -1, dim_per_head * num_heads)
        # final linear projection
        output = self.linear_final(context)
        # dropout
        output = self.dropout(output)
        # add residual and norm layer: layer_norm(wx+F(x))
        output = self.layer_norm(residual + output)
        return output, attention
# Layer normalization: 在x的最后一个维度求方差，最后一个维度就是隐层的维度
# Normalization有很多种，但是它们都有一个共同的目的，那就是把输入转化成均值为0方差为1的数据。我们在把数据送入激活函数之前进行normalization（归一化），因为我们不希望输入数据落在激活函数的饱和区。
```

Mask

Transformer模型里面涉及两种mask。分别是**padding mask**和**sequence mask**。其中后者我们已经在decoder的self-attention里面见过啦！

其中，**padding mask**在所有的scaled dot-product attention里面都需要用到，而**sequence mask**只有在decoder的self-attention里面用到。

所以，我们之前**ScaledDotProductAttention**的 `forward` 方法里面的参数 `attn_mask` 在不同的地方会有不同的含义。这一点我们会在后面说明。

1. Padding mask, 常见
2. Sequence mask

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。

那么具体怎么做呢？也很简单：**产生一个上三角矩阵，上三角的值全为1，下三角的值权威0，对角线也是0**。把这个矩阵作用在每一个序列上，就可以达到我们的目的啦。

`attn_mask` 参数有几种情况？分别是什么意思？

- 对于decoder的self-attention，里面使用到的scaled dot-product attention，同时需要 `padding mask` 和 `sequence mask` 作为 `attn_mask`，具体实现就是两个mask相加作为attn_mask。
- 其他情况， `attn_mask` 一律等于 `padding mask` 。

Position Embedding

论文的实现很有意思，使用正余弦函数。公式如下：

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

公式中的d_{model}是模型的维度，论文默认是 `512` ， `pos` 是指词语在序列中的位置。可以看出，在**偶数位置，使用正弦编码，在奇数位置，使用余弦编码**。

Position-wise Feed-Forward network

这就是一个全连接网络，包含两个线性变换和一个非线性函数（实际上就是ReLU）。公式如下：

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

这个线性变换在不同的位置都表现地一样，并且在不同的层之间使用不同的参数。

论文提到，这个公式还可以用两个核大小为1的一维卷积来解释，卷积的输入输出都是$d_{model} = 512$，中间层的维度是$d_{ff} = 2048$。

Transformer的实现

```
import torch
import torch.nn as nn


class EncoderLayer(nn.Module):
  """Encoder的一层。"""

    def __init__(self, model_dim=512, num_heads=8, ffn_dim=2018, dropout=0.0):
        super(EncoderLayer, self).__init__()

        self.attention = MultiHeadAttention(model_dim, num_heads, dropout)
        self.feed_forward = PositionalWiseFeedForward(model_dim, ffn_dim, dropout)

    def forward(self, inputs, attn_mask=None):

        # self attention
        context, attention = self.attention(inputs, inputs, inputs, padding_mask)

        # feed forward network
        output = self.feed_forward(context)

        return output, attention


class Encoder(nn.Module):
  """多层EncoderLayer组成Encoder。"""

    def __init__(self,
                 vocab_size,
                 max_seq_len,
                 num_layers=6,
                 model_dim=512,
                 num_heads=8,
                 ffn_dim=2048,
                 dropout=0.0):
        super(Encoder, self).__init__()

        self.encoder_layers = nn.ModuleList(
          [EncoderLayer(model_dim, num_heads, ffn_dim, dropout) for _ in
           range(num_layers)])

        self.seq_embedding = nn.Embedding(vocab_size + 1, model_dim, padding_idx=0)
```

```
        self.pos_embedding = PositionalEncoding(model_dim, max_seq_len)

    def forward(self, inputs, inputs_len):
        output = self.seq_embedding(inputs)
        output += self.pos_embedding(inputs_len)

        self_attention_mask = padding_mask(inputs, inputs)

        attentions = []
        for encoder in self.encoder_layers:
            output, attention = encoder(output, self_attention_mask)
            attentions.append(attention)

        return output, attentions


class DecoderLayer(nn.Module):

    def __init__(self, model_dim, num_heads=8, ffn_dim=2048, dropout=0.0):
        super(DecoderLayer, self).__init__()

        self.attention = MultiHeadAttention(model_dim, num_heads, dropout)
        self.feed_forward = PositionalWiseFeedForward(model_dim, ffn_dim, dropout)

    def forward(self,
                dec_inputs,
                enc_outputs,
                self_attn_mask=None,
                context_attn_mask=None):
        # self attention, all inputs are decoder inputs
        dec_output, self_attention = self.attention(
          dec_inputs, dec_inputs, dec_inputs, self_attn_mask)

        # context attention
        # query is decoder's outputs, key and value are encoder's inputs
        dec_output, context_attention = self.attention(
          enc_outputs, enc_outputs, dec_output, context_attn_mask)

        # decoder's output, or context
        dec_output = self.feed_forward(dec_output)

        return dec_output, self_attention, context_attention


class Decoder(nn.Module):

```

```
    def __init__(self,
                 vocab_size,
                 max_seq_len,
                 num_layers=6,
                 model_dim=512,
                 num_heads=8,
                 ffn_dim=2048,
                 dropout=0.0):
        super(Decoder, self).__init__()

        self.num_layers = num_layers

        self.decoder_layers = nn.ModuleList(
          [DecoderLayer(model_dim, num_heads, ffn_dim, dropout) for _ in
           range(num_layers)])

        self.seq_embedding = nn.Embedding(vocab_size + 1, model_dim, padding_idx=0)
        self.pos_embedding = PositionalEncoding(model_dim, max_seq_len)

    def forward(self, inputs, inputs_len, enc_output, context_attn_mask=None):
        output = self.seq_embedding(inputs)
        output += self.pos_embedding(inputs_len)

        self_attention_padding_mask = padding_mask(inputs, inputs)
        seq_mask = sequence_mask(inputs)
        self_attn_mask = torch.gt((self_attention_padding_mask + seq_mask), 0)

        self_attentions = []
        context_attentions = []
        for decoder in self.decoder_layers:
            output, self_attn, context_attn = decoder(
            output, enc_output, self_attn_mask, context_attn_mask)
            self_attentions.append(self_attn)
            context_attentions.append(context_attn)

        return output, self_attentions, context_attentions


class Transformer(nn.Module):

    def __init__(self,
                 src_vocab_size,
                 src_max_len,
                 tgt_vocab_size,
                 tgt_max_len,
                 num_layers=6,
                 model_dim=512,
```

```
                num_heads=8,
                ffn_dim=2048,
                dropout=0.2):
        super(Transformer, self).__init__()

        self.encoder = Encoder(src_vocab_size, src_max_len,
num_layers, model_dim,
                               num_heads, ffn_dim, dropout)
        self.decoder = Decoder(tgt_vocab_size, tgt_max_len,
num_layers, model_dim,
                               num_heads, ffn_dim, dropout)

        self.linear = nn.Linear(model_dim, tgt_vocab_size,
bias=False)
        self.softmax = nn.Softmax(dim=2)

    def forward(self, src_seq, src_len, tgt_seq, tgt_len):
        context_attn_mask = padding_mask(tgt_seq, src_seq)

        output, enc_self_attn = self.encoder(src_seq, src_len)

        output, dec_self_attn, ctx_attn = self.decoder(
          tgt_seq, tgt_len, output, context_attn_mask)

        output = self.linear(output)
        output = self.softmax(output)

        return output, enc_self_attn, dec_self_attn, ctx_attn


```