从源码学习Transformer!

谢杨易 机器学习算法工程师 2020-11-13

> **AI编辑：我是小将**
> **本文作者：谢杨易**
> **https://zhuanlan.zhihu.com/p/178610196**
> **本文已由原作者授权，不得擅自二次转载**

Transformer总体结构

近几年NLP领域有了突飞猛进的发展，预训练模型功不可没。当前利用预训练模型（pretrain models）在下游任务中进行fine-tune，已经成为了大部分NLP任务的固定范式。Transformer摒弃了RNN的序列结构，完全采用attention和全连接，严格来说不属于预训练模型。但它却是当前几乎所有pretrain models的基本结构，为pretrain models打下了坚实的基础，并逐步发展出了transformer-XL，reformer等优化架构。本文结合论文和源码，对transformer基本结构，进行详细分析。

Transformer是谷歌在2017年6月提出，发表在NIPS2017上。论文地址

Attention Is All You Need
arxiv.org

分析的代码为Harvardnlp的代码，基于PyTorch， 地址

annotated-transformer
github.com

Transformer主体框架是一个**encoder-decoder**结构，去掉了RNN序列结构，完全基于attention和全连接。在WMT2014英语翻译德语任务上，bleu值达到了28.4，达到当时的SOTA。其总体结构如下所示

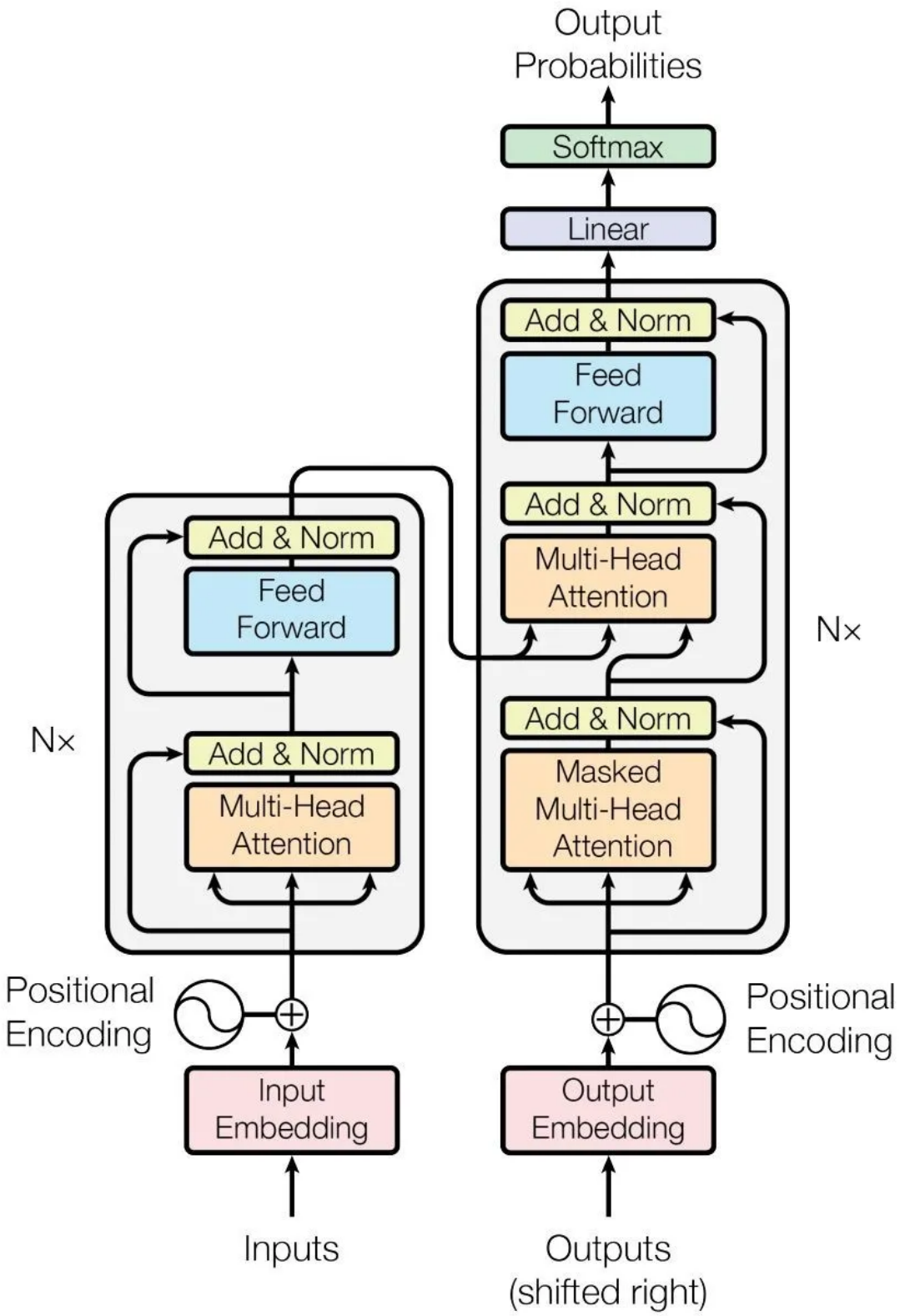


Figure 1: The Transformer - model architecture.

总体为一个典型的encoder-decoder结构。代码如下

```
# 整个模型入口
def make_model(src_vocab, tgt_vocab, N=6,
```

```
                d_model=512, d_ff=2048, h=8, dropout=0.1):
    "Helper: Construct a model from hyperparameters."
    c = copy.deepcopy

    # multiHead attention
    attn = MultiHeadedAttention(h, d_model)

    # feed-forward
    ff = PositionwiseFeedForward(d_model, d_ff, dropout)

    # position-encoding
    position = PositionalEncoding(d_model, dropout)

    # 整体为一个encoder-decoder
    model = EncoderDecoder(
        # encoder编码层
        Encoder(EncoderLayer(d_model, c(attn), c(ff), dropout), N),

        # decoder解码层
        Decoder(DecoderLayer(d_model, c(attn), c(attn), c(ff), dropout), N)

        # 编码层输入，输入语句进行token embedding和position embedding
        nn.Sequential(Embeddings(d_model, src_vocab), c(position)),

        # 解码层输入，同样需要做token embedding和position embedding
        nn.Sequential(Embeddings(d_model, tgt_vocab), c(position)),

        # linear + softmax，查找vocab中概率最大的字
        Generator(d_model, tgt_vocab))

    # This was important from their code.
    # Initialize parameters with Glorot / fan_avg.
    for p in model.parameters():
        if p.dim() &gt; 1:
            nn.init.xavier_uniform(p)
    return model
```

make_model为Transformer模型定义的入口，它先定义了multi-head attention、feed-forward、position-encoding等一系列子模块，然后定义了一个encoder-decoder结构并返回。下面来看encoder-decoder定义。

```
class EncoderDecoder(nn.Module):
    """
    一个标准的encoder和decoder框架，可以自定义embedding、encoder、decoder等
    """
    def __init__(self, encoder, decoder, src_embed, tgt_embed, generator):
        super(EncoderDecoder, self).__init__()

        # encoder和decoder通过构造函数传入，可灵活更改
        self.encoder = encoder
        self.decoder = decoder

        # src和target的embedding，也是通过构造函数传入，方便灵活更改
        self.src_embed = src_embed
        self.tgt_embed = tgt_embed

        # linear + softmax
        self.generator = generator

    def forward(self, src, tgt, src_mask, tgt_mask):
        "Take in and process masked src and target sequences."
        # 先对输入进行encode，然后再通过decode输出
        return self.decode(self.encode(src, src_mask), src_mask,
                           tgt, tgt_mask)

    def encode(self, src, src_mask):
        # 先对输入进行embedding，然后再经过encoder
        return self.encoder(self.src_embed(src), src_mask)

    def decode(self, memory, src_mask, tgt, tgt_mask):
        # 先对目标进行embedding，然后经过decoder
        return self.decoder(self.tgt_embed(tgt), memory, src_mask, tgt_mask
```

encoder-decoder定义了一个标准的编码解码框架，其中编码器、解码器均可以自定义，有很强的泛化能力。模块运行时会调用forward函数，它先对输入进行encode，然后再通过decode输出。我们就不详细展开了。

2 encoder

2.1 encoder定义

encoder分为两部分

1. **输入层embedding**。输入层对inputs文本做token embedding，并对每个字做position encoding，然后叠加在一起，作为最终的输入。
2. **编码层encoding**。编码层是多层结构相同的layer堆叠而成。每个layer又包括两部分，multi-head self-attention和feed-forward全连接，并在每部分加入了残差连接和归一化。

代码实现上也验证了这一点。我们看EncoderDecoder类中的encode函数，它先利用输入embedding层对原始输入进行embedding，然后再通过编码层进行encoding。

```
class EncoderDecoder(nn.Module):
    def encode(self, src, src_mask):
        # 先对输入进行embedding，然后再经过encoder
        return self.encoder(self.src_embed(src), src_mask)
```

2.2 输入层embedding

原始文本经过embedding层进行向量化，它包括token embedding和position embedding两层。

2.2.1 token embedding

token embedding对文本进行向量化，一般来说有两种方式

1. 采用**固定词向量**，比如利用Word2vec预先训练好的。这种方式是LSTM时代常用的方式，比较简单省事，无需训练。但由于词向量是固定的，不能解决一词多义的问题，词语本身也不是contextual的，没有结合上下文语境信息，另外对于不在词向量中的词语，比如特定领域词语或者新词，容易出现OOV问题。
2. 随机初始化，然后**训练**。这种方式比较麻烦，需要大规模训练语料，但能解决固定词向量的一系列问题。Transformer采用了这种方式。

另外，基于Transformer的BERT模型在中文处理时，直接基于字做embedding，优点有

1. 无需分词，故不会引入分词误差。事实上，只要训练语料充分，模型自然就可以学到分词信息了。
2. 中文字个数固定，不会导致OOV问题
3. 中文字相对词，数量少很多，embedding层参数大大缩小，减小了模型体积，并加快了训练速度。

事实上，就算在LSTM时代，很多case中，我们也碰到过基于字的embedding的效果比基于词的要好一些。

```
class Embeddings(nn.Module):
    # token embedding，随机初始化训练，然后查表找到每个字的embedding
    def __init__(self, d_model, vocab):
        super(Embeddings, self).__init__()
        # 构建一个随机初始化的词向量表，[vocab_size, d_model]。bert中的设置为[.
        self.lut = nn.Embedding(vocab, d_model)
        self.d_model = d_model

    def forward(self, x):
        # 从词向量表中查找字对应的embedding向量
        return self.lut(x) * math.sqrt(self.d_model)
```

由代码可见，Transformer采用的是随机初始化，然后训练的方式。词向量维度为[vocab_size, d_model]。例如BERT中为[21128, 768]，参数量还是很大的。ALBert针对embedding层进行矩阵分解，大大减小了embedding层体积。

2.2.2 position encoding

首先一个问题，为啥要进行位置编码呢。原因在于self-attention，将任意两个字之间距离缩小为1，丢失了字的位置信息，故我们需要加上这一信息。我们也可以想到两种方法

1. **固定编码**。Transformer采用了这一方式，通过奇数列cos函数，偶数列sin函数方式，利用三角函数对位置进行固定编码。
2. **动态训练**。BERT采用了这种方式。先随机初始化一个embedding table，然后训练得到table参数值。predict时通过embedding_lookup找到每个位置的embedding。这种方式和token embedding类似。

哪一种方法好呢？个人以为各有利弊

1. 固定编码方式简洁，不需要训练。且不受embedding table维度影响，理论上可以支持任意长度文本。（但要尽量避免预测文本很长，但训练集文本较短的case）
2. 动态训练方式，在语料比较大时，准确度比较好。但需要训练，且最致命的是，限制了输入文本长度。当文本长度大于position embedding table维度时，超出的position无法查表得到embedding（可以理解为OOV了）。这也是为什么BERT模型文本长度最大512的原因。

```
class PositionalEncoding(nn.Module):
    # 位置编码。transformer利用编码方式实现，无需训练。bert则采用训练embedding_
    # 编码方式文本语句长度不受限，但准确度不高
    # 训练方式文本长度会受position维度限制（这也是为什么bert只能处理最大512个字，
    def __init__(self, d_model, dropout, max_len=5000):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(p=dropout)

        # 采用sin和cos进行position encoding
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) *
                             -(math.log(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)        # 偶数列
        pe[:, 1::2] = torch.cos(position * div_term)        # 奇数列
        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)

    def forward(self, x):
        # token embedding和position encoding加在一起
        x = x + Variable(self.pe[:, :x.size(1)],
                         requires_grad=False)
        return self.dropout(x)
```

由代码可见，position encoding直接采用了三角函数。对偶数列采用sin，奇数列采用cos。

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

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

2.3 编码层

Encoder层是Transformer的核心，它由**N层相同结构的layer**（默认6层）堆叠而成。

```
class Encoder(nn.Module):
    "Core encoder is a stack of N layers"
    def __init__(self, layer, N):
```

```
        super(Encoder, self).__init__()
        # N层堆叠而成，每一层结构都是相同的，训练参数不同
        self.layers = clones(layer, N)

        # layer normalization
        self.norm = LayerNorm(layer.size)

    def forward(self, x, mask):
        # 1 经过N层堆叠的multi-head attention + feed-forward
        for layer in self.layers:
            x = layer(x, mask)

        # 2 对encoder最终输出结果进行layer-norm归一化。层间和层内子模块都做过 c
        return self.norm(x)
```

encoder的定义很简洁。先经过N层相同结构的layer，然后再进行归一化输出。重点我们来看layer的定义。

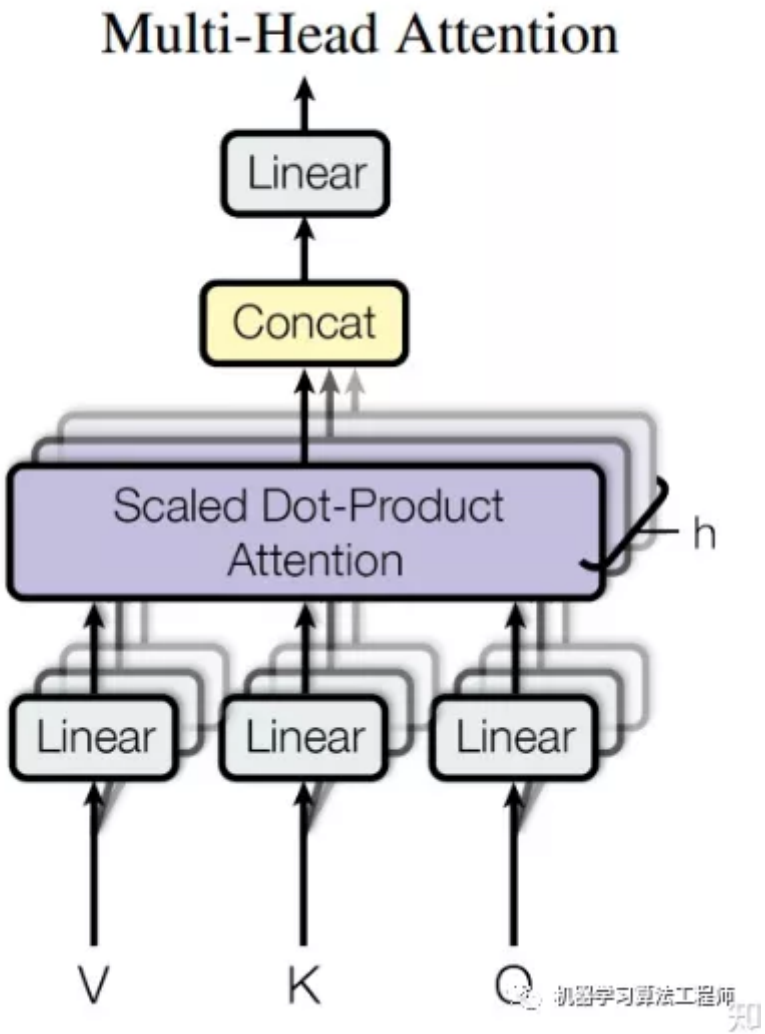
```
class EncoderLayer(nn.Module):
    "Encoder is made up of self-attn and feed forward (defined below)"
    def __init__(self, size, self_attn, feed_forward, dropout):
        super(EncoderLayer, self).__init__()
        # 1 self_attention
        self.self_attn = self_attn

        # 2 feed_forward
        self.feed_forward = feed_forward


        # 3 残差连接。encoder和decoder，每层结构，每个子结构，都有残差连接。
        # add + drop-out + layer-norm
        self.sublayer = clones(SublayerConnection(size, dropout), 2)
        self.size = size

    def forward(self, x, mask):
        # 经过self_attention, 然后和输入进行add + layer-norm
        x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, mask))

        # 经过feed_forward， 此模块也有add + layer-norm
        return self.sublayer[1](x, self.feed_forward)
```

encoder layer分为两个子模块

> **self attention**, 并对输入attention前的和经过attention输出的，做残差连接。残差连接先经过layer-norm归一化，然后进行dropout，最后再做add。后面我们详细分析
> **feed-forward**全连接，也有残差连接的存在，方式和self attention相同。

2.3.1 MultiHeadedAttention

MultiHeadedAttention采用多头self-attention。它先将隐向量切分为h个头，然后每个头内部进行self-attention计算，最后再concat再一起。



代码如下

```
class MultiHeadedAttention(nn.Module):
    def __init__(self, h, d_model, dropout=0.1):
        super(MultiHeadedAttention, self).__init__()
        assert d_model % h == 0
```

```
        # d_model为隐层维度，也是embedding的维度，h为多头个数。
        # d_k为每个头的隐层维度，要除以多头个数。也就是加入了多头，总隐层维度不变
        self.d_k = d_model // h
        self.h = h

        # 线性连接
        self.linears = clones(nn.Linear(d_model, d_model), 4)
        self.attn = None
        self.dropout = nn.Dropout(p=dropout)

    def forward(self, query, key, value, mask=None):
        if mask is not None:
            # 输入mask，在decoder的时候有用到。decode时不能看到要生成字之后的字
            mask = mask.unsqueeze(1)
        nbatches = query.size(0)

        # 1) q, k, v形状变化，加入多头， [batch, L, d_model] =&gt; [batch, h,
        query, key, value = [l(x).view(nbatches, -1, self.h, self.d_k).tran
             for l, x in zip(self.linears, (query, key, value))]

        # 2) attention计算
        x, self.attn = attention(query, key, value, mask=mask,
                                 dropout=self.dropout)

        # 3) 多头结果concat在一起，还原为初始形状
        x = x.transpose(1, 2).contiguous().view(nbatches, -1, self.h * self

        # 4）最后经过一个线性层
        return self.linears[-1](x)
```

下面重点来看单个头的self-attention。也就是论文中的“Scaled Dot-Product Attention”。attention本质上是一个向量的加权求和。它探讨的是每个位置对当前位置的贡献。步骤如下

q向量和每个位置的k向量计算点积，然后除以向量长度的根号。计算点积可以认为是进行权重计算。除以向量长度原因是向量越长，q*k值理论上会越大，故需要在向量长度上做归一化。
attention-mask。mask和输入矩阵shape相同，mask矩阵中值为0位置对应的输入矩阵的值更改为-1e9，一个非常非常小的数，经过softmax后趋近于0。decoder中使用了mask，后面我们详细分析。
softmax归一化，使得q向量和每个位置的k向量的score分布到（0, 1）之间

加权系数乘以每个位置v向量，然后加起来。

公式如下：

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

代码如下

```
def attention(query, key, value, mask=None, dropout=None):
    # attention计算，self_attention和soft-attention都是使用这个函数
    # self-attention, q k v 均来自同一文本。要么是encoder，要么是decoder
    # soft-attention, q来自decoder，k和v来自encoder，从而按照decoder和encoder
    d_k = query.size(-1)

    # 利用q * k计算两向量间相关度，相关度高则权重大。
    # 除以根号dk的原因是，对向量长度进行归一化。q和k的向量长度越长，q*k的值越大
    scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d_k)

    # attention-mask，将 mask中为1的 元素所在的索引，在a中相同的的索引处替换为
    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)

    # softmax归一化
    p_attn = F.softmax(scores, dim = -1)

    # dropout
    if dropout is not None:
        p_attn = dropout(p_attn)

    # 最后利用归一化后的加权系数，乘以每一个v向量，再加和在一起，作为attention后
    return torch.matmul(p_attn, value), p_attn
```

self-attention和soft-attention共用了这个函数，他们之间的唯一区别是**q k v向量的来源不同**。self-attention中q k v 均来自同一文本。而decoder的soft-attention，q来自于decoder，k和v来自于encoder。它体现的是encoder对decoder的加权贡献。

2.3.2 PositionwiseFeedForward

feed-forward本质是一个两层的全连接，全连接之间加入了relu非线性和dropout。比较简单，代码如下

```
class PositionwiseFeedForward(nn.Module):
    # 全连接层
    def __init__(self, d_model, d_ff, dropout=0.1):
        super(PositionwiseFeedForward, self).__init__()
        # 第一层全连接  [d_model, d_ff]
        self.w_1 = nn.Linear(d_model, d_ff)

        # 第二层全连接 [d_ff, d_model]
        self.w_2 = nn.Linear(d_ff, d_model)

        # dropout
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        # 全连接1 -&gt; relu -&gt; dropout -&gt; 全连接2
        return self.w_2(self.dropout(F.relu(self.w_1(x))))
```

总体过程是：**全连接1 -> relu -> dropout -> 全连接2**。两层全连接内部没有shortcut，这儿不要搞混了。

2.3.3 SublayerConnection

在每层的self-attention和feed-forward模块中，均应用了残差连接。残差连接先对输入进行layerNorm归一化，然后送入attention或feed-forward模块，然后经过dropout，最后再和原始输入相加。这样做的好处是，让每一层attention和feed-forward模块的输入值，均是经过归一化的，保持在一个量级上，从而可以加快收敛速度。

```
class SublayerConnection(nn.Module):
    """
    A residual connection followed by a layer norm.
    Note for code simplicity the norm is first as opposed to last.
    """
    def __init__(self, size, dropout):
        super(SublayerConnection, self).__init__()
        # layer-norm 归一化
        self.norm = LayerNorm(size)

        # dropout
```

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

    def forward(self, x, sublayer):
        # 先对输入进行Layer-norm, 然后经过attention等相关模块，再经过dropout，
        return x + self.dropout(sublayer(self.norm(x)))
```

从forward函数可见，先对输入进行layer-norm, 然后经过attention等相关模块，再经过dropout，最后再和输入相加。残差连接的作用就不说了，参考ResNet。

3 decoder

decoder结构和encoder大体相同，也是堆叠了N层相同结构的layer（默认6层）。不同的是，decoder的每个子层包括三层。

1. **masked multi-head self-attention**。这一部分和encoder基本相同，区别在于decoder为了保证模型不能看见要预测字的后面位置的字，加入了mask，从而避免未来信息的穿越问题。mask为一个上三角矩阵，上三角全为1，下三角和对角线全为0
2. **multi-head soft-attention**。soft-attention和self-attention结构基本相同，甚至实现函数都是同一个。唯一的区别在于，self-attention的q k v矩阵来自同一个，所以叫self-attention。而soft-attention的q来自decoder，k和v来自encoder。表征的是encoder的整体输出对于decoder的贡献。
3. **feed-forward**。这一块基本相同。

另外三个模块均使用了残差连接，步骤仍然为 layerNorm -> attention等模块 -> dropout -> 和输入进行add decoder每个layer代码如下

```
class DecoderLayer(nn.Module):
    "Decoder is made of self-attn, src-attn, and feed forward (defined belo
    def __init__(self, size, self_attn, src_attn, feed_forward, dropout):
        super(DecoderLayer, self).__init__()
        self.size = size

        # self-attention 自注意力
        self.self_attn = self_attn

        # soft-attenton, encoder的输出对decoder的作用
        self.src_attn = src_attn
```

```
        # feed-forward 全连接
        self.feed_forward = feed_forward

        # 残差连接
        self.sublayer = clones(SublayerConnection(size, dropout), 3)

    def forward(self, x, memory, src_mask, tgt_mask):
        # memory为encoder最终输出
        m = memory

        # 1 对decoder输入做self-attention, 再和输入做残差连接
        x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, tgt_mask))

        # 2 对encoder输出和decoder当前进行soft-attention，此处也有残差连接
        x = self.sublayer[1](x, lambda x: self.src_attn(x, m, m, src_mask))

        # 3 feed-forward全连接，也有残差连接
        return self.sublayer[2](x, self.feed_forward)
```

4 输出层

decoder的输出作为最终输出层的输入，经过两步

1. linear线性连接，也即是w * x + b
2. softmax归一化，向量长度等于vocabulary的长度，得到vocabulary中每个字的概率。利用beam-search等方法，即可得到生成结果。

这一层比较简单，代码如下

```
class Generator(nn.Module):
    "Define standard linear + softmax generation step."
    def __init__(self, d_model, vocab):
        super(Generator, self).__init__()
        self.proj = nn.Linear(d_model, vocab)

    def forward(self, x):
        # 先经过linear线性层，然后经过softmax得到归一化概率分布
        # 输出向量长度等于vocabulary的维度
        return F.log_softmax(self.proj(x), dim=-1)
```

5 总结

Transformer相比LSTM的优点

1. **完全的并行计算**，Transformer的attention和feed-forward，均可以并行计算。而LSTM则依赖上一时刻，必须串行
2. **减少长程依赖**，利用self-attention将每个字之间距离缩短为1，大大缓解了长距离依赖问题
3. **提高网络深度**。由于大大缓解了长程依赖梯度衰减问题，Transformer网络可以很深，基于Transformer的BERT甚至可以做到24层。而LSTM一般只有2层或者4层。网络越深，高阶特征捕获能力越好，模型performance也可以越高。
4. **真正的双向网络**。Transformer可以同时融合前后位置的信息，而双向LSTM只是简单的将两个方向的结果相加，严格来说仍然是单向的。
5. **可解释性强**。完全基于attention的Transformer，可以表达字与字之间的相关关系，可解释性更强。

Transformer也不是一定就比LSTM好，它的缺点如下

1. 文本长度很长时，比如篇章级别，**计算量爆炸**。self-attention的计算量为O(n^2)，n为文本长度。Transformer-xl利用层级方式，将计算速度提升了1800倍
2. Transformer位置信息只靠**position encoding**，效果比较一般。当语句较短时，比如小于10个字，Transformer效果不一定比LSTM好
3. Transformer参数量较大，在大规模数据集上，效果远好于LSTM。但在**小规模数据集**上，如果不是利用pretrain models，效果不一定有LSTM好。

推荐阅读

VoVNet：实时目标检测的新backbone网络

Python编程神器Jupyter Notebook使用的28个秘诀

带你捋一捋anchor-free的检测模型：FCOS

PyTorch分布式训练简明教程