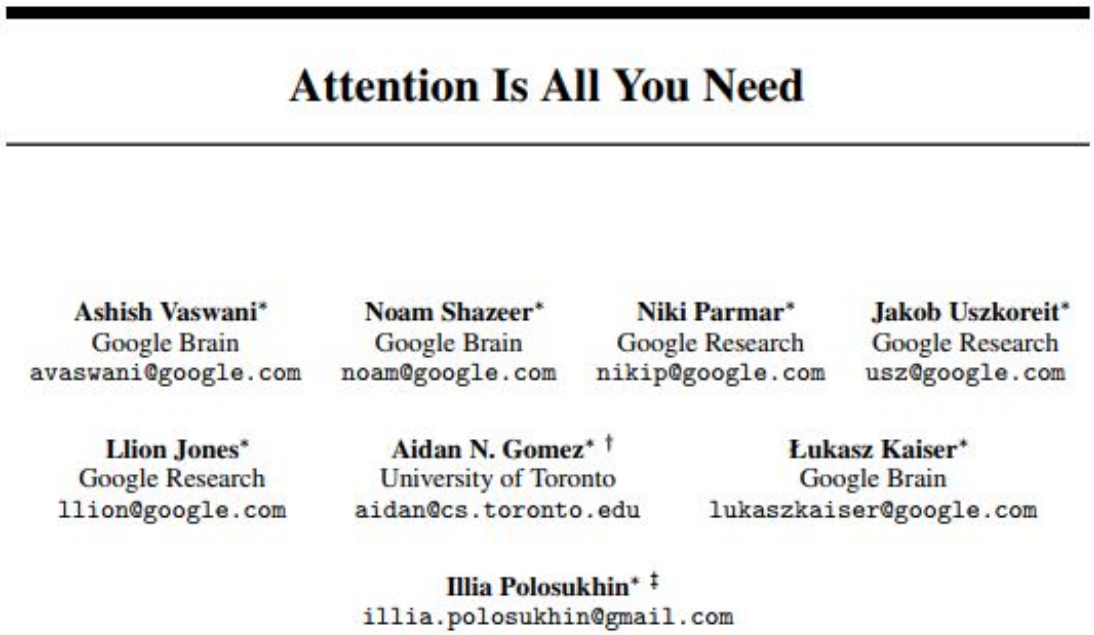
变压器注解及PyTorch实现（上）

mp.weixin.qq.com/s



Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.8 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature. We show that the Transformer generalizes well to other tasks by applying it successfully to English constituency parsing both with large and limited training data.

“注意就是您所需要的” [1]一文中提出的Transformer网络结构最近引起了很多人的关注。Transformer能够明显地提升翻译质量，还为许多NLP任务提供了新的结构。清楚，但实际上大家普遍认为很难正确地实现。

因此，我们这些文章写了篇文章注解文档，并被称为行行实现的Transformer的代码。本文档删除了标题的某些章节并进行了重新排序，并在整个文章中加入了相应的注解。本文档以Jupyter notebook的形式完成，本身就是直接可以运行的代码实现，共有400行库

代码，在4个GPU上每秒可以处理27,000个令牌。

想要运行此工作，首先需要安装PyTorch [2]。。文档文档完整的笔记本文件及依赖可在github [3]或Google Colab [4]上找到。

需要提供注意的是，此注解文档和代码仅作为研究人员和开发者的入门版教程。这里提供的代码主要依赖OpenNMT [5]实现，想了解更多有关此模型的其他实现版本可以查看Tensor2Tensor [6]。]（tensorflow版本）和Sockeye [7]（mxnet版本）

亚历山大·拉什（@harvardnlp [8] 或srush@seas.harvard.edu）

准备工作

```
# !pip install http://download.pytorch.org/whl/cu80/torch-0.3.0.post4-cp36-cp36m-linux_x86_64.whl numpy matplotlib spacy torchtext seaborn
```

```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import math, copy, time
from torch.autograd import Variable
import matplotlib.pyplot as plt
import seaborn
seaborn.set_context(context="talk")
%matplotlib inline
```

内容目录

-注意在模型中的应用

-位置明智的前馈网络

-嵌入和Softmax

-位置编码

-完整模型

（由于分析师篇幅过长，其余部分在下篇）

训练

-批和蒽

-训练循环

-训练数据和批处理

-硬件和训练进度

-优化器

-正则化

-标签平滑

第一个例子

-数据生成

-损失计算

-贪心解码

真实示例

-数据加载

-迭代器

-多GPU训练

-训练系统附加组件：BPE，搜索，平均

结果

-聚焦可视化

摘要

> 本文注解部分都是以引用的形式表示的，主要内容都是来自财经。

背景

减少序列处理任务的计算量是一个很重要的问题，也是扩展的神经GPU，ByteNet和ConvS2S等网络的动机。上面提到的这些网络都以CNN为基础，并行计算所有输入和输出位置的隐藏表示。在这些模型中，关联来自两个任意输入或输出位置的信号所需的操作数随位置间的距离增长而增长，而ConvS2S呈线性增长，By teNet呈现以对数形式增长，这导致学习较远而在Transformer中，操作次数则被减少到了常数等级。

自我注意有时候也被称为内部注意，是在个别句子不同位置上做的注意，并得到序列的一个表示。它能够很好地应用到很多任务中，包括阅读理解，摘要，文本蕴涵，以及独立于任务的句子表示。端到端的网络一般都是基于循环关注机制而不是序列拆分循环，并且已经有证据表明在简单语言问答和语言建模任务上表现很好。

据我们所知，Transformer是第一个完全正确的自我注意而不使用序列排列的RNN或卷积的方式来计算输入输出表示的转换模型。

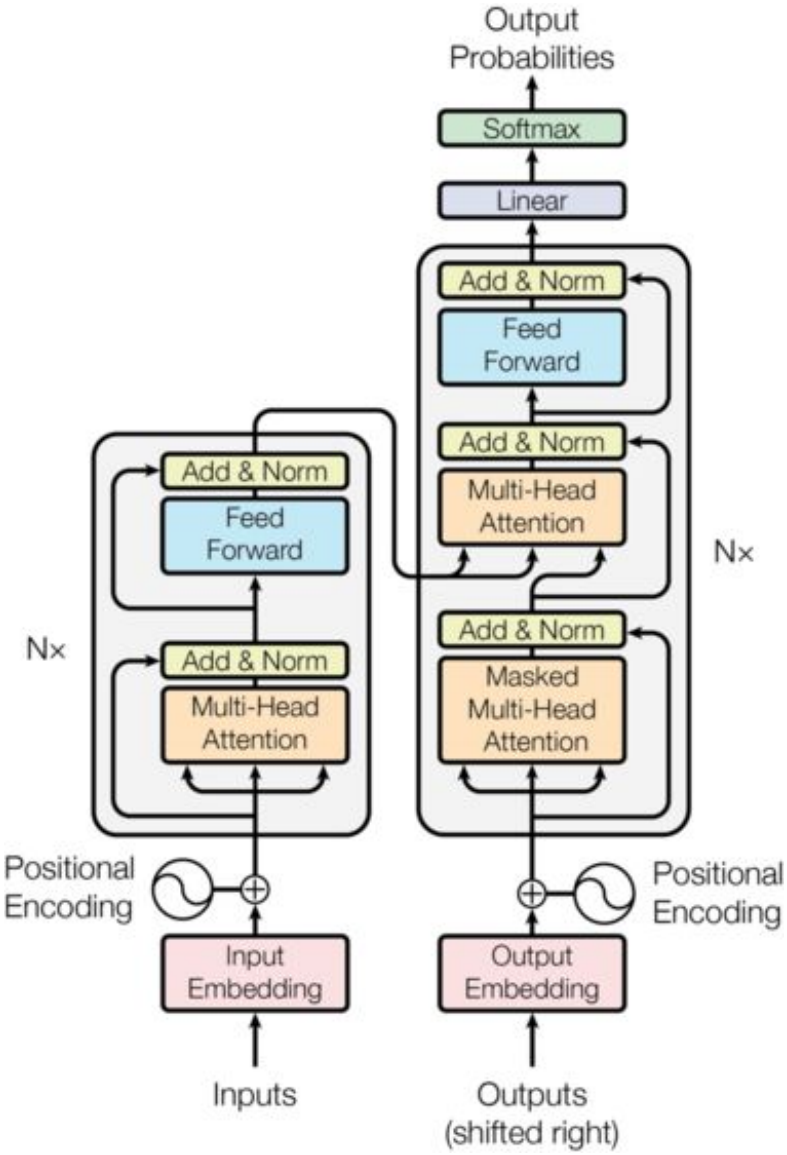
模型结构

目前大部分比较热门的神经序列转换模型都有Encoder-Decoder结构[9]。编码器将输入序列 $(x_1, ..., x_n)$ 映射到一个连续表示序列。对于编码得到的 $\mathbf{z}$ $\mathbf{z} = (z_1, ..., z_n)$，Decoder依次解码生成一个符号，直到生成完整的输出序列：$(y_1, ..., y_m)$。对于每一步解码，模型都是自回归的[10]，即在生成下一个符号时将先前生成的符号作为附加输入。

```
class EncoderDecoder(nn.Module):
    """
    A standard Encoder-Decoder architecture. Base for this and many
    other models.
    """
    def __init__(self, encoder, decoder, src_embed, tgt_embed, generator):
        super(EncoderDecoder, self).__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.src_embed = src_embed
        self.tgt_embed = tgt_embed
        self.generator = generator
    def forward(self, src, tgt, src_mask, tgt_mask):
        "Take in and process masked src and target sequences."
        return self.decode(self.encode(src, src_mask), src_mask,
                           tgt, tgt_mask)
    def encode(self, src, src_mask):
        return self.encoder(self.src_embed(src), src_mask)
    def decode(self, memory, src_mask, tgt, tgt_mask):
        return self.decoder(self.tgt_embed(tgt), memory, src_mask, tgt_mask)
```

```
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):
        return F.log_softmax(self.proj(x), dim=-1)
```

变压器的整体结构如下图所示，在编码器和解码器中都使用了自注意，指向和全连接层。编码器和解码器的大致结构分别如下图的左半部分和右半部分所示。



编码器和解码器

编码器

编码器由N = 6个相同的层组成。

```
1. def clones(module, N):
2.     "Produce N identical layers."
3.     return nn.ModuleList([copy.deepcopy(module) for _ in range(N)])
```

```
class Encoder(nn.Module):
    "Core encoder is a stack of N layers"
    def __init__(self, layer, N):
        super(Encoder, self).__init__()
        self.layers = clones(layer, N)
        self.norm = LayerNorm(layer.size)
    def forward(self, x, mask):
        "Pass the input (and mask) through each layer in turn."
        for layer in self.layers:
            x = layer(x, mask)
        return self.norm(x)
```

我们在每两个子层之间都使用了残差连接（Residual Connection）[11]和归一化[12]。

```
class LayerNorm(nn.Module):
    "Construct a layernorm module (See citation for details)."
    def __init__(self, features, eps=1e-6):
        super(LayerNorm, self).__init__()
        self.a_2 = nn.Parameter(torch.ones(features))
        self.b_2 = nn.Parameter(torch.zeros(features))
        self.eps = eps
    def forward(self, x):
        mean = x.mean(-1, keepdim=True)
        std = x.std(-1, keepdim=True)
        return self.a_2 * (x - mean) / (std + self.eps) + self.b_2
```

也就是说，每个子层的输出为，其中 $\mathrm{Sublayer(x)}$ 是由子层自动实现的函数。我们在每个子层的输出上使用Dropout，然后将其添加到下一子层的输入并进行归一化。

$$\mathrm{LayerNorm(x + Sublayer(x))}$$

为了能方便地使用这些残差连接，模型中所有的子层和嵌入层的输出都设定相同的尺寸，即 $\mathrm{d_{model} = 512}$ 。

```
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__()
        self.norm = LayerNorm(size)
        self.dropout = nn.Dropout(dropout)
    def forward(self, x, sublayer):
        "Apply residual connection to any sublayer with the same size."
        return x + self.dropout(sublayer(self.norm(x)))
```

每层都有两个子层组成。第一个子层实现了“多头”的自我注意，第二个子层则是一个简单的Position-wise的全连接前馈网络。

```
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__()
        self.self_attn = self_attn
        self.feed_forward = feed_forward
        self.sublayer = clones(SublayerConnection(size, dropout), 2)
        self.size = size
    def forward(self, x, mask):
        "Follow Figure 1 (left) for connections."
        x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, mask))
        return self.sublayer[1](x, self.feed_forward)
```

解码器

解码器也是由N = 6个相同层组成。

```
class Decoder(nn.Module):
    "Generic N layer decoder with masking."
    def __init__(self, layer, N):
        super(Decoder, self).__init__()
        self.layers = clones(layer, N)
        self.norm = LayerNorm(layer.size)
    def forward(self, x, memory, src_mask, tgt_mask):
        for layer in self.layers:
            x = layer(x, memory, src_mask, tgt_mask)
        return self.norm(x)
```

除了每个编码器层中的两个子层之外，解码器还插入了第三种子层对编码器栈的输出实行“多头”的注意。与编码器类似，我们在每个子层交替使用残差连接进行短路，然后进行层的规范化处理。

```
class DecoderLayer(nn.Module):
    "Decoder is made of self-attn, src-attn, and feed forward (defined below)"
    def __init__(self, size, self_attn, src_attn, feed_forward, dropout):
        super(DecoderLayer, self).__init__()
        self.size = size
        self.self_attn = self_attn
        self.src_attn = src_attn
        self.feed_forward = feed_forward
        self.sublayer = clones(SublayerConnection(size, dropout), 3)
    def forward(self, x, memory, src_mask, tgt_mask):
        "Follow Figure 1 (right) for connections."
        m = memory
        x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, tgt_mask))
        x = self.sublayer[1](x, lambda x: self.src_attn(x, m, m, src_mask))
        return self.sublayer[2](x, self.feed_forward)
```

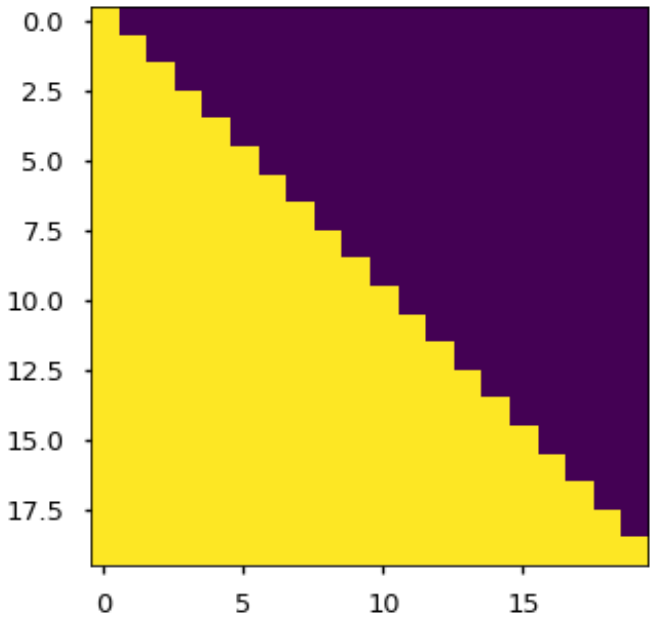
我们还修改了解码器中的自我注意子层以防止当前位置参加到后续位置。这种被掩盖的

注意是考虑到输出嵌入会改变一个位置，确保了生成位置 i 的预测时，仅依赖小于 i 的位置处的已知输出，相当于把后面不该看到的信息屏蔽掉。

```
def subsequent_mask(size):
    "Mask out subsequent positions."
    attn_shape = (1, size, size)
    subsequent_mask = np.triu(np.ones(attn_shape), k=1).astype('uint8')
    return torch.from_numpy(subsequent_mask) == 0
```

> 在训练期间，当前解码位置的单词不能参加到后续位置的单词。

```
plt.figure(figsize=(5,5))
plt.imshow(subsequent_mask(20)[0])
None
```

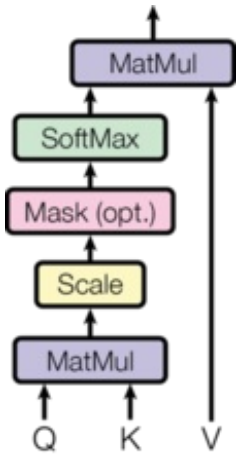


注意

注意函数可以将查询和一组键值对映射到输出，其中查询，键，值和输出都是向量。输出是值的替换和，其中分配给每个值的权重由查询与相应键的兼容函数计算。

我们称这种特殊的注意机制为“按比例扩大 d_k 的点乘产品注意”。输入包含维度为的查询和关键字，以及维度为 d_v 的值。我们首先分别计算查询与各个关键字的点积，然后将每个点积除以 $\sqrt{d_k}$ ，最后使用Softmax函数来获得Key的权重。

在具体实现时，我们可以以矩阵的形式进行并行运算，这样能加速运算过程。具体而言，将所有的查询，键和值向量分别组合成对矩阵 Q ， K 和 V ，这样输出矩阵可以表示为：




```
def attention(query, key, value, mask=None, dropout=None):
    "Compute 'Scaled Dot Product Attention'"
    d_k = query.size(-1)
    scores = torch.matmul(query, key.transpose(-2, -1)) \
             / math.sqrt(d_k)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)
    p_attn = F.softmax(scores, dim = -1)
    if dropout is not None:
        p_attn = dropout(p_attn)
    return torch.matmul(p_attn, value), p_attn
```

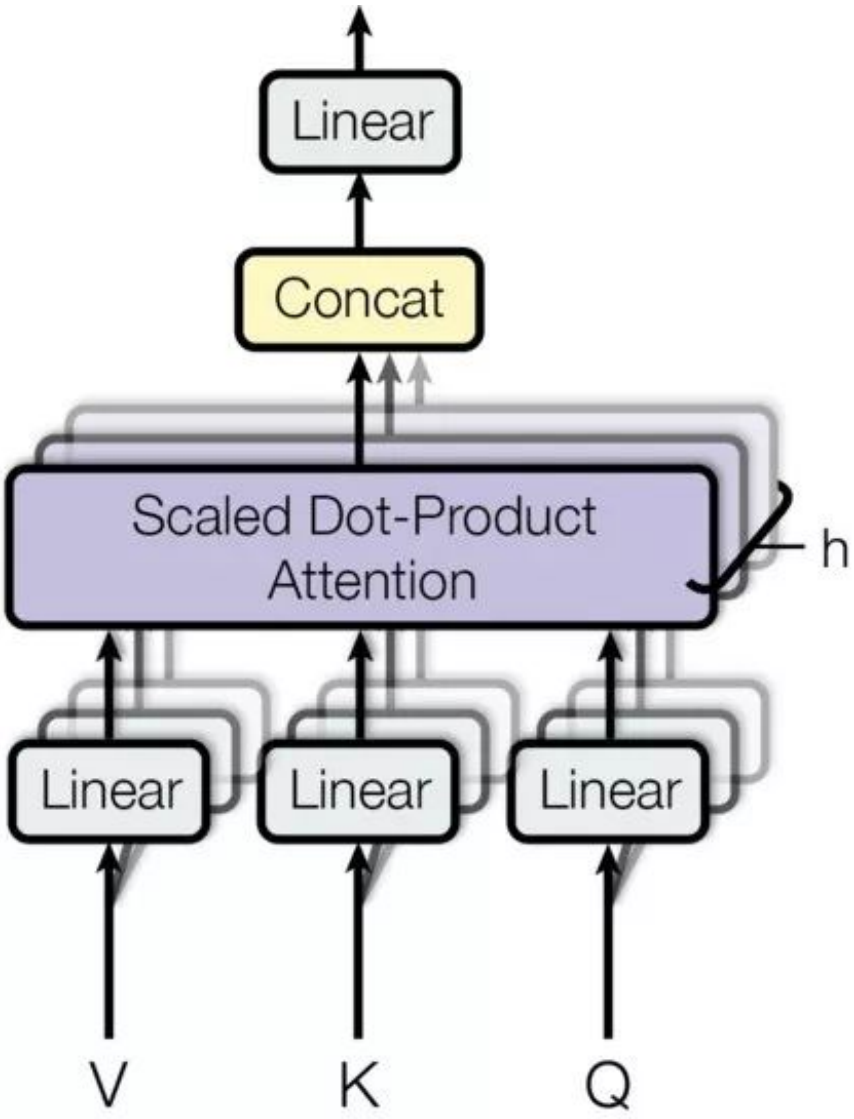
$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

两种最常用的Attention函数是加和Attention [13]和点积（乘积）Attention，我们的算法与点积Attention很类似，但是 $\frac{1}{\sqrt{d_k}}$ 的比例因子不同。加和Attention使用具有隐藏层的前馈网络来计算兼容函数。虽然两种方法理论上的复杂度是相似的，但在实践中，点积注意的运算会转换一些，也更节省空间，因为它可以使用高效的矩阵乘法算法来实现。

虽然对于较小的 d_k ，这两种机制的表现相似，在但不放缩较大的 d_k 时，加和注意要优于点积注意[14]。我们怀疑，对于较大的 d_k ，点积大幅增大，将Softmax函数推向具有极小梯度的区域（为了纠正点积变大的原因，假设 q 和 k 是独立的随机变量，从而为 0 ，方差 1 ，这样他们的点积为，同样是均值 0 方差为 d_k ） 。为了抵消这种影响，用我们 $\frac{1}{\sqrt{d_k}}$ 来缩放点积。

$$q \cdot k = \sum_{i=1}^{d_k} q_i k_i$$

“多头”机制有助于模型考虑到不同位置的注意力，另外“多头”注意力可以在不同的子空间表示不一样的关联关系，使用不同的头的注意力一般达不到这种效果。



$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

其中参数矩阵为，，和。

$$W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$$

我们的工作中使用一个 $h = 8$ Head并行的Attention，对每一个Head说有，总计算量与完整维度的分开Head的Attention很相近。

$$W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$$

$$W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$$

$$W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$$

$$d_k = d_v = d_{\text{model}}/h = 64$$

```
class MultiHeadedAttention(nn.Module):
    def __init__(self, h, d_model, dropout=0.1):
        "Take in model size and number of heads."
        super(MultiHeadedAttention, self).__init__()
        assert d_model % h == 0
        # We assume d_v always equals d_k
        self.d_k = d_model // h
        self.h = h
        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):
        "Implements Figure 2"
        if mask is not None:
            # Same mask applied to all h heads.
            mask = mask.unsqueeze(1)
        nbatches = query.size(0)
        # 1) Do all the linear projections in batch from d_model => h x d_k
        query, key, value = \
            [l(x).view(nbatches, -1, self.h, self.d_k).transpose(1, 2)
             for l, x in zip(self.linears, (query, key, value))]
        # 2) Apply attention on all the projected vectors in batch.
        x, self.attn = attention(query, key, value, mask=mask,
                                 dropout=self.dropout)
        # 3) "Concat" using a view and apply a final linear.
        x = x.transpose(1, 2).contiguous() \
             .view(nbatches, -1, self.h * self.d_k)
        return self.linears[-1](x)
```

注意在模型中的应用

变压器中以三种不同的方式使用了“多头”注意：

1）在“编码器-解码器注意”层，查询来自先前的解码器层，以及键和值来自编码器的输出。解码器中的每个位置插入输入序列中的所有位置，这与Seq2Seq模型中的经典的编码器-解码器注意机制[15]一致。

2）编码器中的自我注意层。在自我注意层中，所有的键，值和查询都来同一个地方，这里都是来自编码器中前一层的输出。编码器中当前层的每个位置都能参加到前一层的所有位置。

3）类似的，解码器中的自我注意层允许解码器中的每个位置参加当前解码位置和它前面的所有位置。这里需要屏蔽解码器中向左的信息流以保持自回归属性。具体的实现方式是在缩放后的点积Attention中，屏蔽（设置 $-\infty$ ）Softmax的输入中所有对应着非法连接的值。

位置明智的前馈网络

除了注意子层之外，Encoder和Decoder中的每个层都包含一个全连接前馈网络，分别地各自位置。其中包括两个线性变换，然后使用ReLU作为激活函数。

虽然实际上线性变换在不同位置上是相同的，但它们在层与层之间使用不同的参数。这其实是相当于使用了两个内核大小为1的卷积。此处设置输入和输出的维数为 $d_{\text{model}} = 512$ ，内层的尺寸为 $d_{ff} = 2048$ 。

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

```
class PositionwiseFeedForward(nn.Module):
    "Implements FFN equation."
    def __init__(self, d_model, d_ff, dropout=0.1):
        super(PositionwiseFeedForward, self).__init__()
        self.w_1 = nn.Linear(d_model, d_ff)
        self.w_2 = nn.Linear(d_ff, d_model)
        self.dropout = nn.Dropout(dropout)
    def forward(self, x):
        return self.w_2(self.dropout(F.relu(self.w_1(x))))
```

嵌入和Softmax

与其他序列转换模型类似，我们使用预学习的嵌入将输入令牌序列和输出令牌序列转换为 d_{model} 维向量。我们仍使用常用的预训练的线性变换和Softmax函数将解码器输出转换为预测下一个令牌概率。在我们的模型中，我们在两个嵌入层和Pre-softmax线性变换

之间共享相同的权重矩阵，称为[16]。在嵌入层中，我们将这些权重乘以 $\sqrt{d_{\text{model}}}$ 。

```
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.     def forward(self, x):
7.         return self.lut(x) * math.sqrt(self.d_model)
```

位置编码

由于我们的模型不包含递归归和卷积结构，为了使模型能够有效利用序列的顺序特征，我们需要加入序列中各个令牌间相对位置或令牌在序列中绝对位置的信息。在这里，我们将位置编码由于位置编码与Embedding具有相同的尺寸 d_{model} ，因此可以直接相加。实际上这里还有很多位置编码选择，其中包括可更新的和固定不变的[17]。

在这些工作中，我们使用不同频率的正弦和余弦函数：

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

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

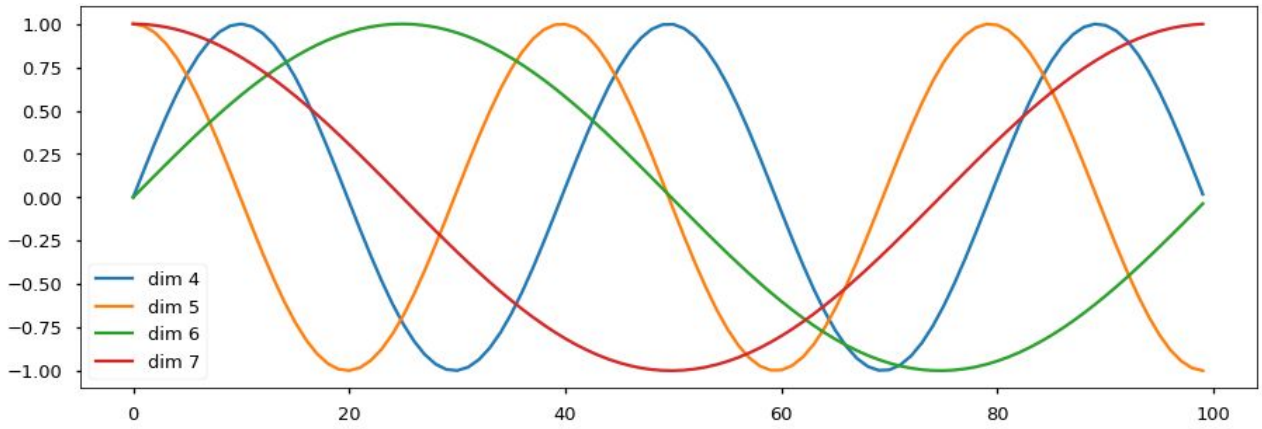
其中 pos 是位置， i 是维度。也就是说，位置编码的每个维度都对应于一个正弦曲线，其波长形成从 2π 到 $10000 \cdot 2\pi$ 的等比级数。我们之所以选择了这个函数，是因为我们假设它能够模型很容易学会Attend相对位置，因为对于任何固定的转换量 k ， PE_{pos+k} 可以表示为 PE_{pos} 的线性函数。

此外，在编码器和解码器范围中，我们在Embedding与位置编码的加和上都使用了Dropout机制。在基本模型上，我们使用 $P_{drop} = 0.1$ 的比率。

```
class PositionalEncoding(nn.Module):
    "Implement the PE function."
    def __init__(self, d_model, dropout, max_len=5000):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(p=dropout)
        # Compute the positional encodings once in log space.
        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)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)
    def forward(self, x):
        x = x + Variable(self.pe[:, :x.size(1)],
                         requires_grad=False)
        return self.dropout(x)
```

> 如下所示，位置编码将根据位置添加正弦曲线。曲线的频率和改变对于每个维度是不同的。

```
plt.figure(figsize=(15, 5))
pe = PositionalEncoding(20, 0)
y = pe.forward(Variable(torch.zeros(1, 100, 20)))
plt.plot(np.arange(100), y[0, :, 4:8].data.numpy())
plt.legend(["dim %d"%p for p in [4,5,6,7]])
None
```



我们也尝试了使用预学习的位置嵌入，但是发现这两个版本的结果基本是一样的。我们选择了使用正弦曲线版本的实现，因为使用此版本可以模型处理超过训练语料中最大序列长度的序列。

完整模型

> 下面定义了连接完整模型并设置超参的函数。

```
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
    attn = MultiHeadedAttention(h, d_model)
    ff = PositionwiseFeedForward(d_model, d_ff, dropout)
    position = PositionalEncoding(d_model, dropout)
    model = EncoderDecoder(
        Encoder(EncoderLayer(d_model, c(attn), c(ff), dropout), N),
        Decoder(DecoderLayer(d_model, c(attn), c(attn),
                             c(ff), dropout), N),
        nn.Sequential(Embeddings(d_model, src_vocab), c(position)),
        nn.Sequential(Embeddings(d_model, tgt_vocab), c(position)),
        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() > 1:
            nn.init.xavier_uniform(p)
    return model
```

```
# Small example model.
tmp_model = make_model(10, 10, 2)
None
```

参考链接

[1] https://arxiv.org/abs/1706.03762

[2] https://pytorch.org/

[3] https://github.com/harvardnlp/annotated-transformer

[4] https://drive.google.com/file/d/1xQXSv6mtAOLXxEMi8RvaW8TW-7bvYBDF/view?usp=sharing

[5] http://opennmt.net

[6] https://github.com/tensorflow/tensor2tensor

[7] https://github.com/awslabs/sockeye

[8] https://twitter.com/harvardnlp

[9] https://arxiv.org/abs/1409.0473

[10] https://arxiv.org/abs/1308.0850

[11] https://arxiv.org/abs/1512.03385

[12] https://arxiv.org/abs/1607.06450

[13] https://arxiv.org/abs/1409.0473

[14] https://arxiv.org/abs/1703.03906

[15] https://arxiv.org/abs/1609.08144

[16] https://arxiv.org/abs/1608.05859

[17] https://arxiv.org/pdf/1705.03122.pdf