Transformer 的 PyTorch 实现及详解

目录

1. Transformer 的 PyTorch 实现

https://wmathor.com/index.php/archives/1455/

B 站视频讲解

本文主要介绍一下如何使用 PyTorch 复现 Transformer，实现简单的机器翻译任务。请先阅读我的这篇文章 Transformer 详解，方能达到醍醐灌顶，事半功倍的效果

(1). 数据预处理

这里我并没有用什么大型的数据集，而是手动输入了两对德语→英语的句子，还有每个字的索引也是我手动硬编码上去的，主要是为了降低代码阅读难度，我希望读者能更关注模型实现的部分

```
import math
import torch
import numpy as np
import torch.nn as nn
import torch.optim as optim
import torch.utils.data as Data
# S: Symbol that shows starting of decoding input
# E: Symbol that shows starting of decoding output
# P: Symbol that will fill in blank sequence if current batch data size is short than time steps
sentences = [
        # enc_input           dec_input         dec_output
        ['ich mochte ein bier P', 'S i want a beer .', 'i want a beer . E'],
        ['ich mochte ein cola P', 'S i want a coke .', 'i want a coke . E']]
#因为 transformer 的 decoder 既有输入也有输出端口，因此需要有 dec_input 和 dec_output 数据。其中 dec_output 在 decoer 的输出端口和 decoder 的输出一起生成 loss。Dec_input 输入 decoder 端口。
# Padding Should be Zero
src_vocab = {'P' : 0, 'ich' : 1, 'mochte' : 2, 'ein' : 3, 'bier' : 4, 'cola' : 5}
src_vocab_size = len(src_vocab)  # 源词汇表中 token 的数目；
tgt_vocab = {'P' : 0, 'i' : 1, 'want' : 2, 'a' : 3, 'beer' : 4, 'coke' : 5, 'S' : 6, 'E' : 7, '.' : 8}
idx2word = {i: w for i, w in enumerate(tgt_vocab)}
tgt_vocab_size = len(tgt_vocab)  #目标词汇表中 token 的数目；
src_len = 5 # enc_input max sequence length
tgt_len = 6 # dec_input(=dec_output) max sequence length
def make_data(sentences): #对包含 token 的样本进行编码
    enc_inputs, dec_inputs, dec_outputs = [], [], []
    for i in range(len(sentences)):
      enc_input = [[src_vocab[n] for n in sentences[i][0].split()]] # [[1, 2, 3, 4, 0], [1, 2, 3, 5, 0]]
      dec_input = [[tgt_vocab[n] for n in sentences[i][1].split()]] # [[6, 1, 2, 3, 4, 8], [6, 1, 2, 3, 5, 8]]
      dec_output = [[tgt_vocab[n] for n in sentences[i][2].split()]] # [[1, 2, 3, 4, 8, 7], [1, 2, 3, 5, 8, 7]]
```

```
        enc_inputs.extend(enc_input)
        dec_inputs.extend(dec_input)
        dec_outputs.extend(dec_output)
    return torch.LongTensor(enc_inputs), torch.LongTensor(dec_inputs),
torch.LongTensor(dec_outputs)
enc_inputs, dec_inputs, dec_outputs = make_data(sentences)

class MyDataSet(Data.Dataset):
  def __init__(self, enc_inputs, dec_inputs, dec_outputs):
    super(MyDataSet, self).__init__()
    self.enc_inputs = enc_inputs
    self.dec_inputs = dec_inputs
    self.dec_outputs = dec_outputs
  def __len__(self):
    return self.enc_inputs.shape[0]
  def __getitem__(self, idx):
    return self.enc_inputs[idx], self.dec_inputs[idx], self.dec_outputs[idx]
loader = Data.DataLoader(MyDataSet(enc_inputs, dec_inputs, dec_outputs), 2, True)
# loader 中 2 代表 batch_sizem, True 代表 shuffle=False
DataObject=MyDataSet(enc_inputs,dec_inputs,dec_outputs)
Print('DataObject_1=',DataObject[1])  # 输出与索引 1 对应的样本的值
# ['ich mochte ein cola P', 'S i want a coke .', 'i want a coke . E']
```

(2). 模型参数

下面变量代表的含义依次是

1. 字嵌入 & 位置嵌入的维度，这俩值是相同的，因此用一个变量就行了
2. FeedForward 层隐藏神经元个数
3. Q、K、V 向量的维度，其中 Q 与 K 的维度必须相等，V 的维度没有限制，不过为了方便起见，我都设为 64
4. Encoder 和 Decoder 的个数
5. 多头注意力中 head 的数量

```
# Transformer Parameters
d_model = 512  # Embedding Size
d_ff = 2048 # FeedForward dimension
d_k = d_v = 64  # dimension of K(=Q), V
n_layers = 6  # number of Encoder of Decoder Layer
n_heads = 8  # number of heads in Multi-Head Attention
```

上面都比较简单，下面开始涉及到模型就比较复杂了，因此我会将模型拆分成以下几个部分进行讲解

- Positional Encoding
- Pad Mask（针对句子不够长，加了 pad，因此需要对 pad 进行 mask）
- Subsequence Mask（Decoder input 不能看到未来时刻单词信息，因此需要 mask）
- ScaledDotProductAttention（计算 context vector）
- Multi-Head Attention

- FeedForward Layer
- Encoder Layer
- Encoder
- Decoder Layer
- Decoder
- Transformer

关于代码中的注释，如果值为 src_len 或者 tgt_len 的，我一定会写清楚，但是有些函数或者类，Encoder 和 Decoder 都有可能调用，因此就不能确定究竟是 src_len 还是 tgt_len，对于不确定的，我会记作 seq_len

(3). Positional Encoding

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, dropout=0.1, max_len=5000):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(p=dropout)
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-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).transpose(0, 1)
        self.register_buffer('pe', pe)
    def forward(self, x):
        '''
        x: [seq_len, batch_size, d_model]
        '''
        x = x + self.pe[:x.size(0), :]
        return self.dropout(x)
```

(4). Pad Mask

```
def get_attn_pad_mask(seq_q, seq_k):
    '''
    seq_q: [batch_size, seq_len]
    seq_k: [batch_size, seq_len]
    seq_len could be src_len or it could be tgt_len
    seq_len in seq_q and seq_len in seq_k maybe not equal
    '''
    batch_size, len_q = seq_q.size()
    batch_size, len_k = seq_k.size()
    # eq(zero) is PAD token
    pad_attn_mask = seq_k.data.eq(0).unsqueeze(1)  # [batch_size, 1, len_k], False is masked
    return pad_attn_mask.expand(batch_size, len_q, len_k)  # [batch_size, len_q, len_k]
```

encoder 和 decoder 都需要调用 get_attn_pad_mask 函数。计算 mask 时由于自注意力的不同 sample 中 token 的数目不一样，而一个 batch 需要其中所有 Sample 中的 token 的数目都相同。所以，取最长 sample 中 token 的数目作为一个 batch 中所有 sample 的长度，对于 sample 实际

长度不足的部分，进行补 0。0 的位置对应的 mask 的值为 true。由于计算自注意力时，需要计算 sample 中每个 token 对于本 sample 中所有 token 的注意力系数，所以注意力的计算结果是个矩阵，mask 因此也是一个矩阵。

由于在 Encoder 和 Decoder 中都需要进行 mask 操作，因此就无法确定这个函数的参数中 seq_len 的值，如果是在 Encoder 中调用的，seq_len 就等于 src_len；如果是在 Decoder 中调用的，seq_len 就有可能等于 src_len，也有可能等于 tgt_len（因为 Decoder 有两次 mask）

这个函数最核心的一句代码是 seq_k.data.eq(0)，这句的作用是返回一个大小和 seq_k 一样的 tensor，只不过里面的值只有 True 和 False。如果 seq_k 某个位置的值等于 0，那么对应位置就是 True，否则即为 False。举个例子，输入为 seq_data = [1, 2, 3, 4, 0]，seq_data.data.eq(0) 就会返回 [False, False, False, False, True]

剩下的代码主要是扩展维度，**强烈建议读者打印出来，看看最终返回的数据是什么样子**

(5). Subsequence Mask

```
def get_attn_subsequence_mask(seq):
    '''
    # 只用于 decoder
    seq: [batch_size, tgt_len]
    '''
    attn_shape = [seq.size(0), seq.size(1), seq.size(1)]
    subsequence_mask = np.triu(np.ones(attn_shape), k=1) # Upper triangular matrix
    subsequence_mask = torch.from_numpy(subsequence_mask).byte()
    return subsequence_mask # [batch_size, tgt_len, tgt_len]
```

Subsequence Mask 只有 Decoder 会用到，主要作用是屏蔽未来时刻单词的信息。首先通过 np.ones() 生成一个全 1 的方阵，然后通过 np.triu() 生成一个上三角矩阵，下图是 np.triu() 用法

```
#k=0表示正常的上三角矩阵
upper_triangle = np.triu(data, 0)
[[1 2 3 4 5]
 [0 5 6 7 8]
 [0 0 7 8 9]
 [0 0 0 7 8]
 [0 0 0 0 5]]
#k=-1表示对角线的位置下移1个对角线
upper_triangle = np.triu(data, -1)
[[1 2 3 4 5]
 [4 5 6 7 8]
 [0 7 7 8 9]
 [0 0 6 7 8]
 [0 0 0 4 5]]
#k=1表示对角线的位置上移1个对角线
upper_triangle = np.triu(data, 1)
[[0 2 3 4 5]
 [0 0 6 7 8]
 [0 0 0 8 9]
 [0 0 0 0 8]
 [0 0 0 0 0]]
```

(6). ScaledDotProductAttention

```
class ScaledDotProductAttention(nn.Module):
    def __init__(self):
        super(ScaledDotProductAttention, self).__init__()
    def forward(self, Q, K, V, attn_mask):
        '''
        Q: [batch_size, n_heads, len_q, d_k]
        K: [batch_size, n_heads, len_k, d_k]
        V: [batch_size, n_heads, len_v(=len_k), d_v]
        attn_mask: [batch_size, n_heads, seq_len, seq_len]
        '''
        scores = torch.matmul(Q, K.transpose(-1, -2)) / np.sqrt(d_k) # scores : [batch_size, n_heads, len_q, len_k]
        scores.masked_fill_(attn_mask, -1e9) # Fills elements of self tensor with value where mask is True.
        attn = nn.Softmax(dim=-1)(scores)
        context = torch.matmul(attn, V) # [batch_size, n_heads, len_q, d_v]
        return context, attn
```

这里要做的是，通过 Q 和 K 计算出 scores，然后将 scores 和 V 相乘，得到每个单词的 context vector。第一步是将 Q 和 K 的转置相乘没什么好说的，相乘之后得到的 scores 还不能立刻进行 softmax，需要和 attn_mask 相加，把一些需要屏蔽的信息屏蔽掉，attn_mask 是一个仅由 True 和 False 组成的 tensor，并且一定会保证 attn_mask 和 scores 的维度四个值相同（不然无法做对应位置相加）。mask 完了之后，就可以对 scores 进行 softmax 了。然后再与 V 相乘，得到 context

(7). MultiHeadAttention

```
class MultiHeadAttention(nn.Module):
    def __init__(self):
        super(MultiHeadAttention, self).__init__()
        self.W_Q = nn.Linear(d_model, d_k * n_heads, bias=False)
        self.W_K = nn.Linear(d_model, d_k * n_heads, bias=False)
        self.W_V = nn.Linear(d_model, d_v * n_heads, bias=False)
        self.fc = nn.Linear(n_heads * d_v, d_model, bias=False)
    def forward(self, input_Q, input_K, input_V, attn_mask):
        '''
        input_Q: [batch_size, len_q, d_model]
        input_K: [batch_size, len_k, d_model]
        input_V: [batch_size, len_v(=len_k), d_model]
        attn_mask: [batch_size, seq_len, seq_len]
        '''
        residual, batch_size = input_Q, input_Q.size(0)
        # (B, S, D) -proj-> (B, S, D_new) -split-> (B, S, H, W) -trans-> (B, H, S, W)
        Q = self.W_Q(input_Q).view(batch_size, -1, n_heads, d_k).transpose(1,2)  # Q: [batch_size, n_heads, len_q, d_k]
```

```
        K = self.W_K(input_K).view(batch_size, -1, n_heads, d_k).transpose(1,2)  # K: [batch_size, n_heads, len_k, d_k]
        V = self.W_V(input_V).view(batch_size, -1, n_heads, d_v).transpose(1,2)  # V: [batch_size, n_heads, len_v(=len_k), d_v]
        attn_mask = attn_mask.unsqueeze(1).repeat(1, n_heads, 1, 1) # attn_mask : [batch_size, n_heads, seq_len, seq_len]
        # context: [batch_size, n_heads, len_q, d_v], attn: [batch_size, n_heads, len_q, len_k]
        context, attn = ScaledDotProductAttention()(Q, K, V, attn_mask)
        context = context.transpose(1, 2).reshape(batch_size, -1, n_heads * d_v) # context: [batch_size, len_q, n_heads * d_v]
        output = self.fc(context) # [batch_size, len_q, d_model]
        return nn.LayerNorm(d_model).cuda()(output + residual), attn
```

完整代码中一定会有三处地方调用 MultiHeadAttention()，Encoder Layer 调用一次，传入的 input_Q、input_K、input_V 全部都是 enc_inputs；Decoder Layer 中两次调用，第一次传入的全是 dec_inputs，第二次传入的分别是 dec_outputs，enc_outputs，enc_outputs

(8). FeedForward Layer

```
class PoswiseFeedForwardNet(nn.Module):
    def __init__(self):
        super(PoswiseFeedForwardNet, self).__init__()
        self.fc = nn.Sequential(
            nn.Linear(d_model, d_ff, bias=False),
            nn.ReLU(),
            nn.Linear(d_ff, d_model, bias=False)
        )
    def forward(self, inputs):
        '''
        inputs: [batch_size, seq_len, d_model]
        '''
        residual = inputs
        output = self.fc(inputs)
        return nn.LayerNorm(d_model).cuda()(output + residual) # [batch_size, seq_len, d_model]
```

这段代码非常简单，就是做两次线性变换，残差连接后再跟一个 Layer Norm

(9). Encoder Layer

```
class EncoderLayer(nn.Module):
    def __init__(self):
        super(EncoderLayer, self).__init__()
        self.enc_self_attn = MultiHeadAttention()
        self.pos_ffn = PoswiseFeedForwardNet()
    def forward(self, enc_inputs, enc_self_attn_mask):
        # enc_inputs: [batch_size, src_len, d_model]
        # enc_self_attn_mask: [batch_size, src_len, src_len]
        # enc_outputs: [batch_size, src_len, d_model], attn: [batch_size, n_heads, src_len, src_len]
```

```
        enc_outputs, attn = self.enc_self_attn(enc_inputs, enc_inputs, enc_inputs, enc_self_attn_mask) # enc_inputs to same Q,K,V
        enc_outputs = self.pos_ffn(enc_outputs) # enc_outputs: [batch_size, src_len, d_model]
        return enc_outputs, attn
```

将上述组件拼起来，就是一个完整的 Encoder Layer

(10). Encoder

```
class Encoder(nn.Module):
    def __init__(self):
        super(Encoder, self).__init__()
        self.src_emb = nn.Embedding(src_vocab_size, d_model)
        self.pos_emb = PositionalEncoding(d_model)
        self.layers = nn.ModuleList([EncoderLayer() for _ in range(n_layers)])
    def forward(self, enc_inputs):
        # enc_inputs: [batch_size, src_len]
        enc_outputs = self.src_emb(enc_inputs) # [batch_size, src_len, d_model]
        enc_outputs = self.pos_emb(enc_outputs.transpose(0, 1)).transpose(0, 1) # [batch_size, src_len, d_model]
        enc_self_attn_mask = get_attn_pad_mask(enc_inputs, enc_inputs) # [batch_size, src_len, src_len]
        enc_self_attns = []
        for layer in self.layers:
            # enc_outputs: [batch_size, src_len, d_model], enc_self_attn: [batch_size, n_heads, src_len, src_len]
            enc_outputs, enc_self_attn = layer(enc_outputs, enc_self_attn_mask)
            enc_self_attns.append(enc_self_attn)
        return enc_outputs, enc_self_attns
```

使用 nn.ModuleList() 里面的参数是列表，列表里面存了 n_layers 个 Encoder Layer

由于我们控制好了 Encoder Layer 的输入和输出维度相同，所以可以直接用个 for 循环以嵌套的方式，将上一次 Encoder Layer 的输出作为下一次 Encoder Layer 的输入

(11). Decoder Layer

```
class DecoderLayer(nn.Module):
    def __init__(self):
        super(DecoderLayer, self).__init__()
        self.dec_self_attn = MultiHeadAttention()
        self.dec_enc_attn = MultiHeadAttention()
        self.pos_ffn = PoswiseFeedForwardNet()
    def forward(self, dec_inputs, enc_outputs, dec_self_attn_mask, dec_enc_attn_mask):
        '''
        dec_inputs: [batch_size, tgt_len, d_model]
        enc_outputs: [batch_size, src_len, d_model]
        dec_self_attn_mask: [batch_size, tgt_len, tgt_len]
        dec_enc_attn_mask: [batch_size, tgt_len, src_len]， 用于计算 output 中 sample 对于 input 中 sample 的注意力系数。
        '''
```

```
        # dec_outputs: [batch_size, tgt_len, d_model], dec_self_attn: [batch_size, n_heads, tgt_len, tgt_len]
        dec_outputs, dec_self_attn = self.dec_self_attn(dec_inputs, dec_inputs, dec_inputs, dec_self_attn_mask)
        # dec_outputs: [batch_size, tgt_len, d_model], dec_enc_attn: [batch_size, h_heads, tgt_len, src_len]
        dec_outputs, dec_enc_attn = self.dec_enc_attn(dec_outputs, enc_outputs, enc_outputs, dec_enc_attn_mask)
        dec_outputs = self.pos_ffn(dec_outputs) # [batch_size, tgt_len, d_model]
        return dec_outputs, dec_self_attn, dec_enc_attn
```

在 Decoder Layer 中会调用两次 MultiHeadAttention，第一次是计算 Decoder Input 的 self-attention，得到输出 dec_outputs。然后将 dec_outputs 作为生成 Q 的元素，enc_outputs 作为生成 K 和 V 的元素，再调用一次 MultiHeadAttention，得到的是 Encoder 和 Decoder Layer 之间的 context vector。最后将 dec_outptus 做一次维度变换，然后返回

(12). Decoder

```
class Decoder(nn.Module):
    def __init__(self):
        super(Decoder, self).__init__()
        self.tgt_emb = nn.Embedding(tgt_vocab_size, d_model)
        self.pos_emb = PositionalEncoding(d_model)
        self.layers = nn.ModuleList([DecoderLayer() for _ in range(n_layers)])
    def forward(self, dec_inputs, enc_inputs, enc_outputs):
        '''
        dec_inputs: [batch_size, tgt_len]
        enc_intpus: [batch_size, src_len]
        enc_outputs: [batsh_size, src_len, d_model]
        '''
        dec_outputs = self.tgt_emb(dec_inputs) # [batch_size, tgt_len, d_model]
        dec_outputs = self.pos_emb(dec_outputs.transpose(0, 1)).transpose(0, 1).cuda() # [batch_size, tgt_len, d_model]
        dec_self_attn_pad_mask = get_attn_pad_mask(dec_inputs, dec_inputs).cuda() # [batch_size, tgt_len, tgt_len]
        dec_self_attn_subsequence_mask = get_attn_subsequence_mask(dec_inputs).cuda() # [batch_size, tgt_len, tgt_len]
        dec_self_attn_mask = torch.gt((dec_self_attn_pad_mask + dec_self_attn_subsequence_mask), 0).cuda() # [batch_size, tgt_len, tgt_len]
        dec_enc_attn_mask = get_attn_pad_mask(dec_inputs, enc_inputs) # [batc_size, tgt_len, src_len]
        dec_self_attns, dec_enc_attns = [], []
        for layer in self.layers:
            # dec_outputs: [batch_size, tgt_len, d_model], dec_self_attn: [batch_size, n_heads, tgt_len, tgt_len], dec_enc_attn: [batch_size, h_heads, tgt_len, src_len]
            dec_outputs, dec_self_attn, dec_enc_attn = layer(dec_outputs, enc_outputs, dec_self_attn_mask, dec_enc_attn_mask)
```

```
            dec_self_attns.append(dec_self_attn)
            dec_enc_attns.append(dec_enc_attn)
        return dec_outputs, dec_self_attns, dec_enc_attns
```

Decoder 中不仅要把 "pad"mask 掉，还要 mask 未来时刻的信息，因此就有了下面这三行代码，其中 torch.gt(a, value) 的意思是，将 a 中各个位置上的元素和 value 比较，若大于 value，则该位置取 1，否则取 0

```
dec_self_attn_pad_mask = get_attn_pad_mask(dec_inputs, dec_inputs) # [batch_size, tgt_len, tgt_len]
        dec_self_attn_subsequence_mask = get_attn_subsequence_mask(dec_inputs) # [batch_size, tgt_len, tgt_len]
        dec_self_attn_mask = torch.gt((dec_self_attn_pad_mask + dec_self_attn_subsequence_mask), 0) # [batch_size, tgt_len, tgt_len]
```

(13). Transformer

```
class Transformer(nn.Module):
    def __init__(self):
        super(Transformer, self).__init__()
        self.encoder = Encoder().cuda()
        self.decoder = Decoder().cuda()
        self.projection = nn.Linear(d_model, tgt_vocab_size, bias=False).cuda()
    def forward(self, enc_inputs, dec_inputs):
        '''
        enc_inputs: [batch_size, src_len]
        dec_inputs: [batch_size, tgt_len]
        '''
        # tensor to store decoder outputs
        # outputs = torch.zeros(batch_size, tgt_len, tgt_vocab_size).to(self.device)
        # enc_outputs: [batch_size, src_len, d_model], enc_self_attns: [n_layers, batch_size, n_heads, src_len, src_len]；即 enc_self_attns 的维度是 [n_layers, batch_size, n_heads]个矩阵，每个矩阵对应输入样本集中的一个 sample。每个矩阵的维度是[src_len,src_len]，代表这个 sample 中的每个 token 和所有 token 之间的注意力系数。
        enc_outputs, enc_self_attns = self.encoder(enc_inputs)
        # dec_outpus 的维度是 [batch_size, tgt_len, d_model], dec_self_attns 是 docoder 的自注意力系数矩阵，其维度是 [n_layers, batch_size, n_heads, tgt_len, tgt_len]；表示包含 [n_layers, batch_size, n_heads]个矩阵，每个矩阵对应输出样集本中的一个 sample，每个矩阵的维度是[tgt_len, tgt_len]，代表这个 sample 中的每个 token 和所有 token 之间的注意力系数。dec_enc_attn 的维度是 [n_layers, batch_size, tgt_len, src_len]，代表输出 sample 对输入 sample 的自注意力系数。
        dec_outputs, dec_self_attns, dec_enc_attns = self.decoder(dec_inputs, enc_inputs, enc_outputs)
        dec_logits = self.projection(dec_outputs) # dec_logits: [batch_size, tgt_len, tgt_vocab_size]
        return dec_logits.view(-1, dec_logits.size(-1)), enc_self_attns, dec_self_attns, dec_enc_attns
```

Transformer 主要就是调用 Encoder 和 Decoder。最后返回 dec_logits 的维度是 [batch_size *

tgt_len, tgt_vocab_size]，可以理解为，一个句子，这个句子有 batch_size*tgt_len 个单词，每个单词有 tgt_vocab_size 种情况，取概率最大者。

(14). 模型 & 损失函数 & 优化器

```
model = Transformer().cuda()
criterion = nn.CrossEntropyLoss(ignore_index=0)
optimizer = optim.SGD(model.parameters(), lr=1e-3, momentum=0.99)
```

这里的损失函数里面我设置了一个参数 ignore_index=0，因为 "pad" 这个单词的索引为 0，这样设置以后，就不会计算 "pad" 的损失（因为本来 "pad" 也没有意义，不需要计算），关于这个参数更详细的说明，可以看我这篇文章的最下面，稍微提了一下

(15). 训练

```
for epoch in range(30):
    for enc_inputs, dec_inputs, dec_outputs in loader:
        '''
        Loader 是数据集
        enc_inputs: [batch_size, src_len]
        dec_inputs: [batch_size, tgt_len]
        dec_outputs: [batch_size, tgt_len]
        '''
        enc_inputs, dec_inputs, dec_outputs = enc_inputs.cuda(), dec_inputs.cuda(), dec_outputs.cuda()
        # outputs: [batch_size * tgt_len, tgt_vocab_size]
        outputs, enc_self_attns, dec_self_attns, dec_enc_attns = model(enc_inputs, dec_inputs)
        # enc_inputs, dec_inputs 分别代表 Transformer 的 encoder 和 decoder 的输入。
        loss = criterion(outputs, dec_outputs.view(-1))
        print('Epoch:', '%04d' % (epoch + 1), 'loss =', '{:.6f}'.format(loss))
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

(16). 测试

```
def greedy_decoder(model, enc_input, start_symbol):
    """
    For simplicity, a Greedy Decoder is Beam search when K=1. This is necessary for inference as we don't know the target sequence input. Therefore we try to generate the target input word by word, then feed it into the transformer.
# 在翻译场景下，Beam search 用于根据模型获得句子序列的条件概率最大的若干个序列。Beam search 只在预测的时候需要，训练的时候因为知道正确答案，所以并不需要这个保存若干个概率最大的序列，只要选取真值序列即可。
    Starting Reference: http://nlp.seas.harvard.edu/2018/04/03/attention.html#greedy-decoding
    :param model: Transformer Model
    :param enc_input: The encoder input
 :param start_symbol: The start symbol. In this example it is 'S' which corresponds to index 4
```

```
    :return: The target input
    """
    enc_outputs, enc_self_attns = model.encoder(enc_input)
    dec_input = torch.zeros(1, tgt_len).type_as(enc_input.data)
    next_symbol = start_symbol
    for i in range(0, tgt_len):
        dec_input[0][i] = next_symbol
        dec_outputs, _, _ = model.decoder(dec_input, enc_input, enc_outputs)
        projected = model.projection(dec_outputs)
        prob = projected.squeeze(0).max(dim=-1, keepdim=False)[1]
        next_word = prob.data[i]
        next_symbol = next_word.item()
    return dec_input
# Test
print('----iter_loader=',iter(loader))
print('----next_iter_loader=',next(iter(loader)))
enc_inputs, _, _ = next(iter(loader))
print('----enc_inpts=',enc_inputs)
#greedy_dec_input = greedy_decoder(model, enc_inputs[0].view(1, -1).cuda(),
start_symbol=tgt_vocab["S"])
#predict, _, _, _ = model(enc_inputs[0].view(1, -1).cuda(), greedy_dec_input)
greedy_dec_input = greedy_decoder(model, enc_inputs[0].view(1, -1),
start_symbol=tgt_vocab["S"])
print('----greedy_dec_input=',greedy_dec_input)
predict, _, _, _ = model(enc_inputs[0].view(1, -1), greedy_dec_input)
print('-----predict1=',predict)
predict = predict.data.max(1, keepdim=True)[1]
print('-----predict2=',predict)
print(enc_inputs[0], '->', [idx2word[n.item()] for n in predict.squeeze()])
```

 # test 输出结果及解释如下:

```
----iter_loader=  <torch.utils.data.dataloader._SingleProcessDataLoaderIter  object  at
0x7f7cbac5d5f8>
--------------------------------样本 1----------------样本 2----------------------------------------
----next_iter_loader= [tensor([[1, 2, 3, 5, 0],  [1, 2, 3, 4, 0]]),     # 两个德语样本的 word2idx
                       tensor([[6, 1, 2, 3, 5, 8],[6, 1, 2, 3, 4, 8]]), # 两个英语样本的 word2idx1
                       tensor([[1, 2, 3, 5, 8, 7],[1, 2, 3, 4, 8, 7]])] # 两个英语样本的 word2idx2
----enc_inpts= tensor([[1, 2, 3, 4, 0],[1, 2, 3, 5, 0]])
---in greedy func, enc_input= tensor([[1, 2, 3, 4, 0]])
---in greedy func, enc_outputs_size= torch.Size([1, 5, 512])
---in greedy func, len_enc_self_attns_size= 6 len_enc_self_attns_size[0]= 1
---greedy_dec_input= tensor([[6, 1, 2, 3, 4, 8]])
# 与 enc_input= tensor([[1, 2, 3, 4, 0]])对应的一个候选解的解码器的输出
```

-----predict1= tensor([
0, 1, 2, 3, 4, 5, 6, 7, 8
[-3.0962, **9.4811**, -0.7976, -0.2365, 0.0131, -1.2642, -1.0954, -1.3330, -0.9765], #[1]
[-2.0291, -0.2155, **8.4936**, 0.3322, -0.0517, -1.2586, -1.3215, -2.7457,-0.2871], #[2]
[-1.9581, -2.5565, -0.8814, **8.7973**, 0.8803, -2.1356, -1.2294, -1.5222,-0.1107], #[3]
[-2.7858, -3.2013, -2.6246, -0.5517, **6.9892**, 3.9238, -1.5783, 0.9314,-0.5243], #[4]
[-2.9769, -3.3700, -2.1768, 0.5870, 1.4944, -1.1993, -1.9124, 1.9471, **7.6700**], #[8]
[-2.6688, -3.3960, -2.8241, -1.0838, 2.0432, -0.5342, -1.5966, **9.2552**, 2.8768]], #[7]
grad_fn=<ViewBackward>)
#根据 predict1 的对于 6 个输出值中，每个输出值出现的概率可获得如下的最终预测。
-----predict2= tensor([[1],
[2],
[3],
[4],
[8],
[7]])
tensor([1, 2, 3, 4, 0]) -> ['i', 'want', 'a', 'beer', '.', 'E']
['i', 'want', 'a', 'beer', '.', 'E']根据 predict2 和 tgt_vocab = {'P' : 0, 'i' : 1, 'want' : 2, 'a' : 3, 'beer' : 4, 'coke' : 5, 'S' : 6, 'E' : 7, '.' : 8}的出预测的翻译结果。

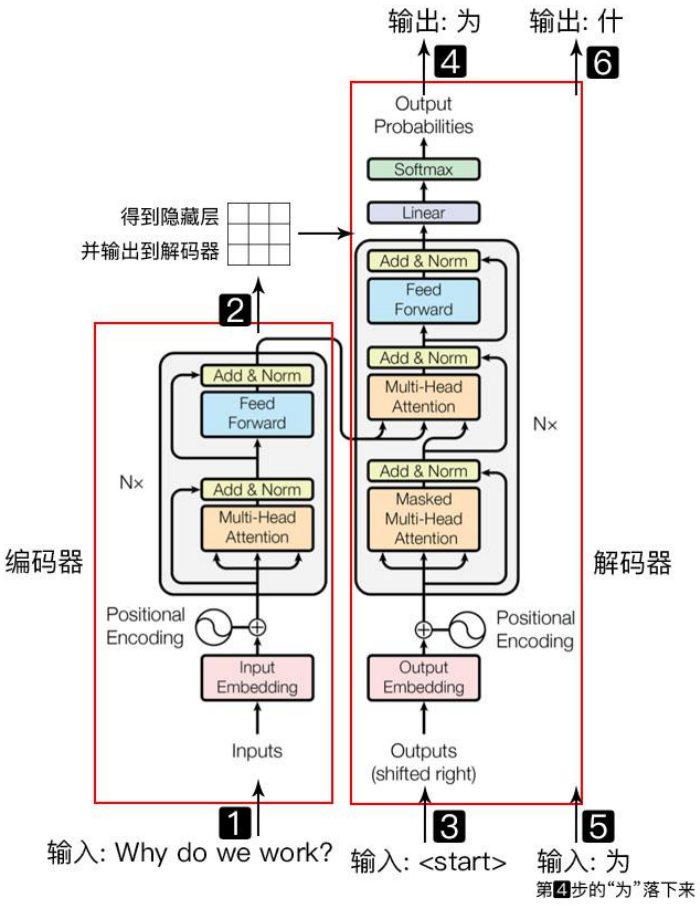
最后给出完整代码链接（需要科学的力量）
Github 项目地址：nip-tutorial

2. Transformer 详解

https://wmathor.com/index.php/archives/1455/
B 站视频讲解
Transformer 是谷歌大脑在 2017 年底发表的论文 attention is all you need 中所提出的 seq2seq 模型。而 BERT 就是从 Transformer 中衍生出来的预训练语言模型

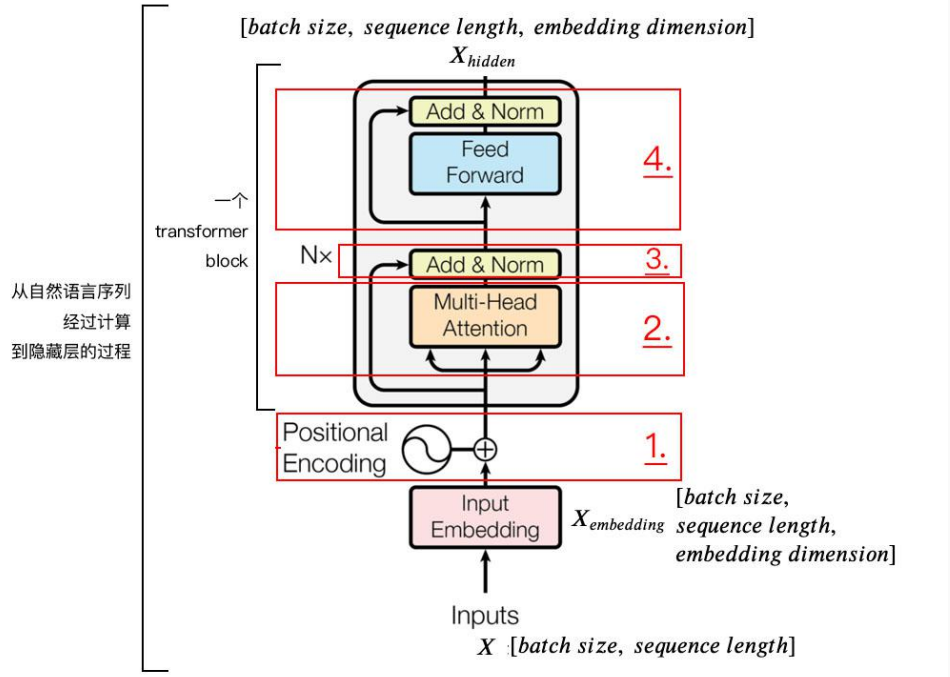
(1). 0.Transformer 直观认识

Transformer 和 LSTM 的最大区别，就是 LSTM 的训练是迭代的、串行的，必须要等当前字处理完，才可以处理下一个字。而 Transformer 的训练时并行的，即所有字是同时训练的，这样就大大增加了计算效率。Transformer 使用了位置嵌入(Positional Encoding)来理解语言的顺序，使用自注意力机制（Self Attention Mechanism）和全连接层进行计算，这些后面会讲到
Transformer 模型主要分为两大部分，分别是 Encoder 和 Decoder。Encoder 负责把输入（语言序列）隐射成隐藏层（下图中第 2 步用九宫格代表的部分），然后解码器再把隐藏层映射为自然语言序列。例如下图机器翻译的例子（Decoder 输出的时候，是通过 N 层 Decoder Layer 才输出一个 token，并不是通过一层 Decoder Layer 就输出一个 token）



上图中，decoder 中下面的 Masked Multi-head Attention 用于计算输出样本的自注意力系数。Multi-Head Attention 用于计算输出样本和输入样本之间的注意力系数。

本篇文章大部分内容在于解释 Encoder 部分，即把自然语言序列映射为隐藏层的数学表达的过程。理解了 Encoder 的结构，再理解 Decoder 就很简单了



上图为 Transformer Encoder Block 结构图，注意：下面的内容标题编号分别对应着图中 1,2,3,4 个方框的序号

(2). 1. Positional Encoding

由于 Transformer 模型没有循环神经网络的迭代操作, 所以我们必须提供每个字的位置信息给 Transformer，这样它才能识别出语言中的顺序关系

现在定义一个位置嵌入的概念，也就是 Positional Encoding，位置嵌入的维度为 [max_sequence_length, embedding_dimension], 位置嵌入的维度与词向量的维度是相同的，都是 embedding_dimension。max_sequence_length 属于超参数，指的是限定每个句子最长由多少个词构成。注意，我们一般以字为单位训练 Transformer 模型。首先初始化字编码的大小为 [vocab_size, embedding_dimension]，vocab_size 为字库中所有字的数量，embedding_dimension 为字向量的维度，对应到 PyTorch 中，其实就是 nn.Embedding(vocab_size, embedding_dimension)

论文中使用了 sin 和 cos 函数的线性变换来提供给模型位置信息:

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

上式中 PE 代表 Positional Encoding；2i/dmodel 中的 dmodel 是个固定的值。i 越大，则 sin 输入项中的 10000^(2i/dmodel) 的值越大；(pos/10000^(2i/dmodel)) 的值就越小，从而使得 PE(pos,2i)和 PE(pos,2i+1)的变动幅度越小。上述 PE 的设计使得 i 类似于权重，i 越小，则对应的位置的权重波动越剧烈。

上式中 pos 指的是一句话中某个字的位置，取值范围是[0, max_sequence_length)，i 指的是字向量的维度序号，取值范围是 [0, embedding_dimension/2)，dmodel 指的是 embedding_dimension 的值。i 的取值上限是 embedding_dimension/2，这是因为 sin 和 cos 两种计算位置 PE 的方案，所以字符向量的取值范围是整个 embedding_dimension 维度中的一半。

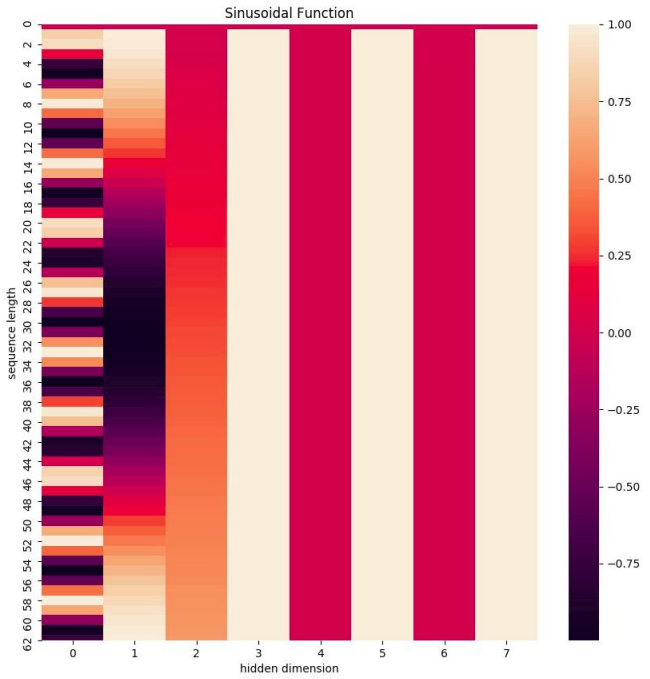
上面有 sin 和 cos 一组公式，也就是对应着 embedding dimension 维度的一组奇数和偶数的序号的维度，例如 0,1 一组, 2,3 一组，分别用上面的 sin 和 cos 函数做处理，从而产生不同的周期性变化，而位置嵌入在 embedding dimension 维度上随着维度序号增大，周期变化会越来越慢，最终产生一种包含位置信息的纹理，就像论文原文中第六页讲的，位置嵌入函数的周期从 2π到 $10000 * 2\pi$变化，而每一个位置在 embedding dimension 维度上都会得到不同周期的 sin 和 cos 函数的取值组合，从而产生独一的纹理位置信息，最终使得模型学到位置之间的依赖关系和自然语言的时序特性。

如果不理解这里为何这么设计，可以看这篇文章 Transformer 中的 Positional Encoding

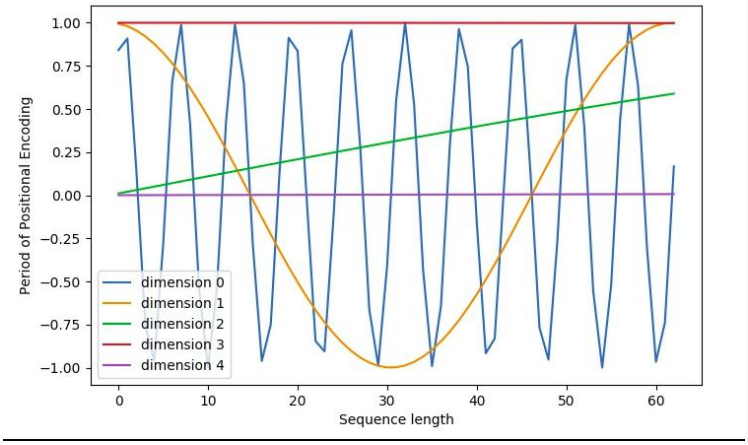
下面画一下位置嵌入，纵向观察，可见随着 embedding dimension 序号增大，位置嵌入函数的周期变化越来越平缓。

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import math
def get_positional_encoding(max_seq_len, embed_dim):
    # 其中 max_seq_len 代表每句话中包含字符的长度；embed_dim 代表每个字符的维度；
    # 初始化一个 positional encoding
    # embed_dim: 字嵌入的维度
```

```
    # max_seq_len: 最大的序列长度
    positional_encoding = np.array([
        [pos / np.power(10000, 2 * i / embed_dim) for i in range(embed_dim)]
        if pos != 0 else np.zeros(embed_dim) for pos in range(max_seq_len)])
    positional_encoding[1:, 0::2] = np.sin(positional_encoding[1:, 0::2])  # dim 2i 偶数
    positional_encoding[1:, 1::2] = np.cos(positional_encoding[1:, 1::2])  # dim 2i+1 奇数
    #[1:, 0::2]代表起始 index 是 0，前进的步长是 2；的第一维的数据 index 的范围是 1：end
    return positional_encoding
positional_encoding = get_positional_encoding(max_seq_len=64, embed_dim=8)
plt.figure(figsize=(10,10))
sns.heatmap(positional_encoding)
plt.title("Sinusoidal Function")
plt.xlabel("hidden dimension")
plt.ylabel("sequence length")
plt.savefig('map_result.jpg')
```

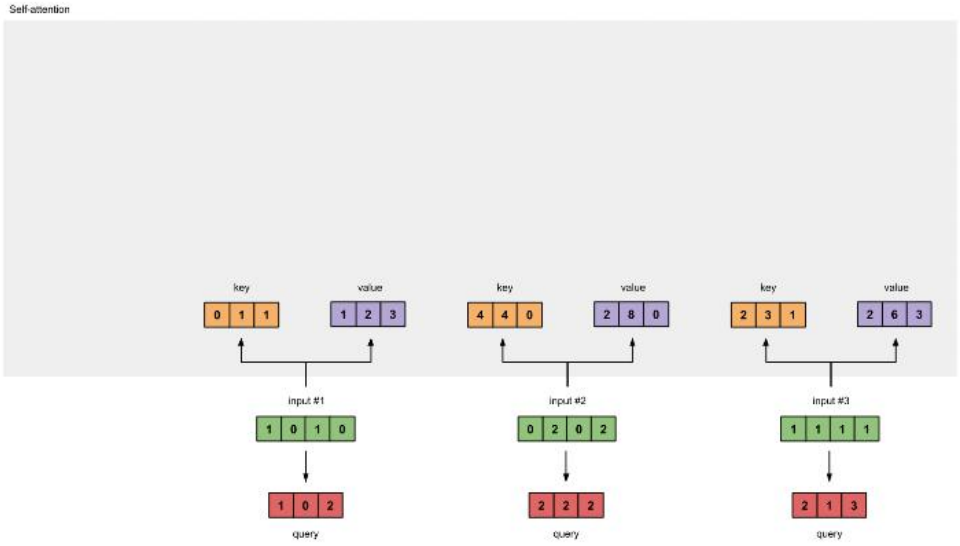



```
plt.figure(figsize=(8, 5))
plt.plot(positional_encoding[1:, 0], label="dimension 0")
plt.plot(positional_encoding[1:, 1], label="dimension 1")
plt.plot(positional_encoding[1:, 2], label="dimension 2")
plt.plot(positional_encoding[1:, 3], label="dimension 3")
plt.plot(positional_encoding[1:, 4], label="dimension 4")
plt.legend()
plt.xlabel("Sequence length")
plt.ylabel("Period of Positional Encoding")
plt.savefig('wave_result.jpg')
```



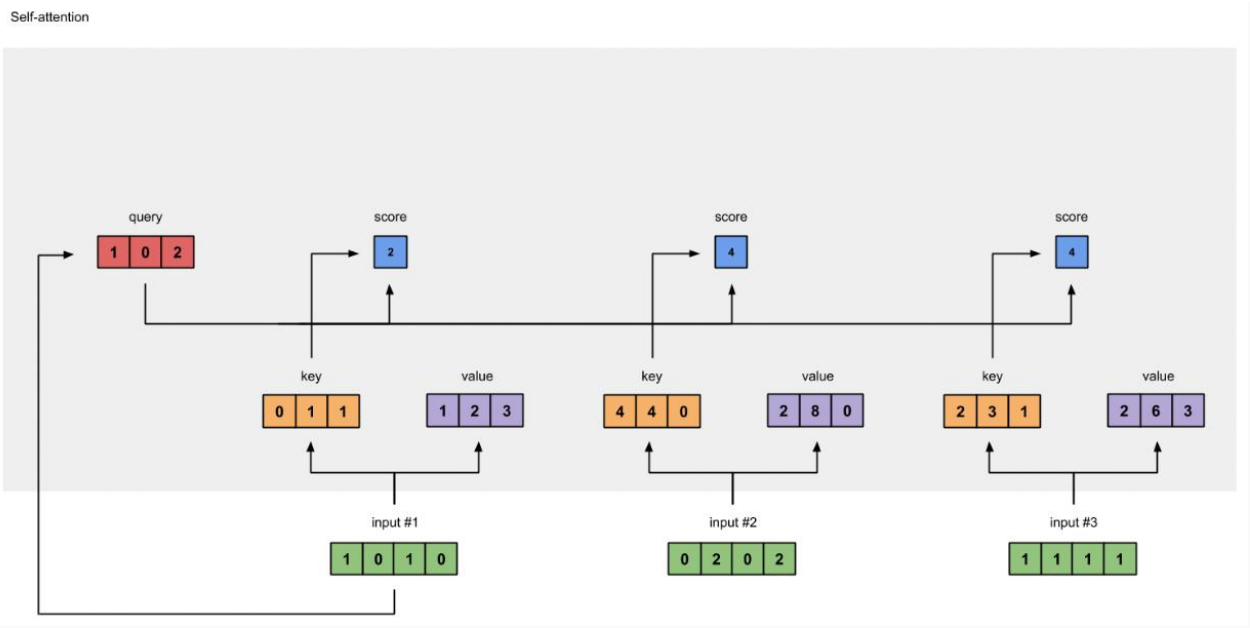
(3). 2. Self Attention Mechanism

对于输入的句子 X，通过 WordEmbedding 得到该句子中每个字的字向量，同时通过 Positional Encoding 得到所有字的位置向量，将其相加（维度相同，可以直接相加），得到该字真正的向量表示。第 t 个字的向量记作 x_t。接着我们定义三个矩阵 W_Q,W_K.W_V，使用这三个矩阵分别对所有的字向量进行三次线性变换，于是所有的字向量又衍生出三个新的向量 q_t,k_t,v_t。我们将所有的 q_t 向量拼成一个大矩阵，记作查询矩阵 Q，将所有的 k_t 向量拼成一个大矩阵，记作键矩阵 K，将所有的 v_t 向量拼成一个大矩阵，记作值矩阵 V（见下图）



为了获得第一个字的注意力权重，我们需要用第一个字的查询向量 q1 乘以键矩阵 K（见下图）

[0, 4, 2]
[1, 0, 2] x [1, 4, 3] = [2, 4, 4]
[1, 0, 1]



之后还需要将得到的值经过 softmax，使得它们的和为 1（见下图）

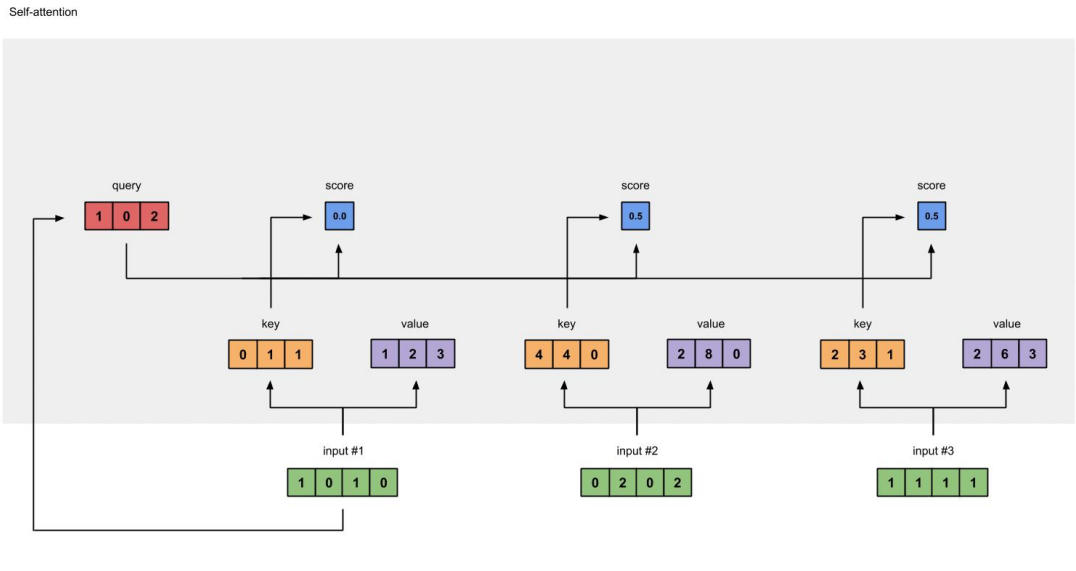
softmax([2, 4, 4]) = [0.0, 0.5, 0.5]

有了权重之后，将权重其分别乘以对应字的值向量 vt （见下图）

0.0 * [1, 2, 3] = [0.0, 0.0, 0.0]

0.5 * [2, 8, 0] = [1.0, 4.0, 0.0]

0.5 * [2, 6, 3] = [1.0, 3.0, 1.5]



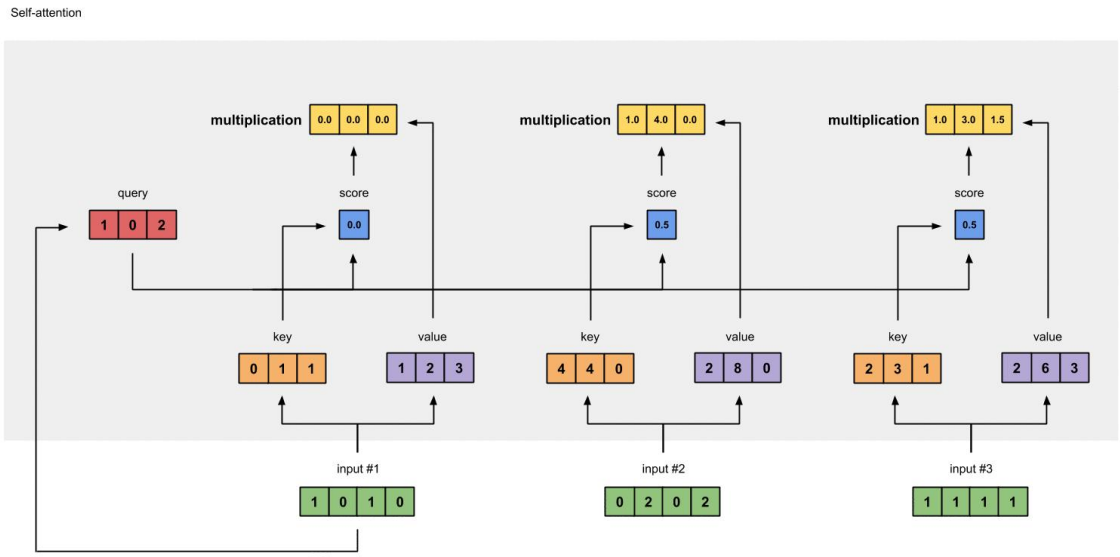
最后将这些权重化后的值向量求和，得到第一个字的输出（见下图）

  [0.0, 0.0, 0.0]

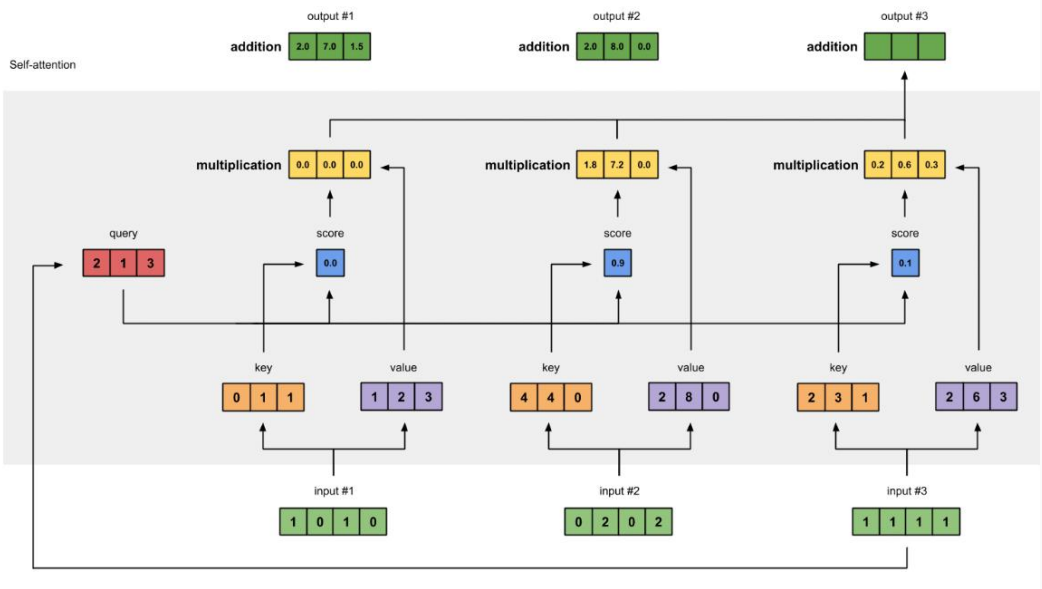
+ [1.0, 4.0, 0.0]

+ [1.0, 3.0, 1.5]

= [2.0, 7.0, 1.5]



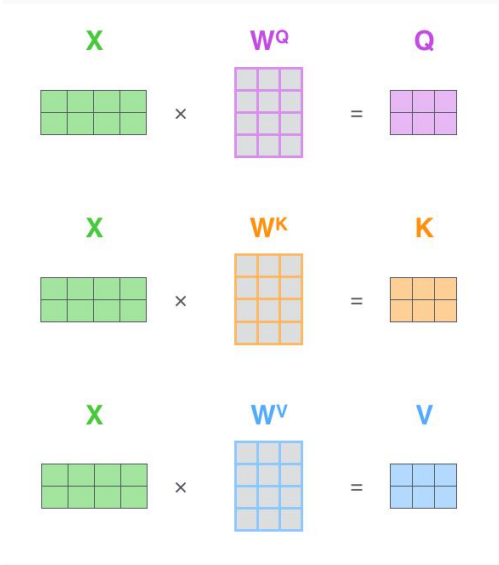
对其它的输入向量也执行相同的操作，即可得到通过 self-attention 后的所有输出



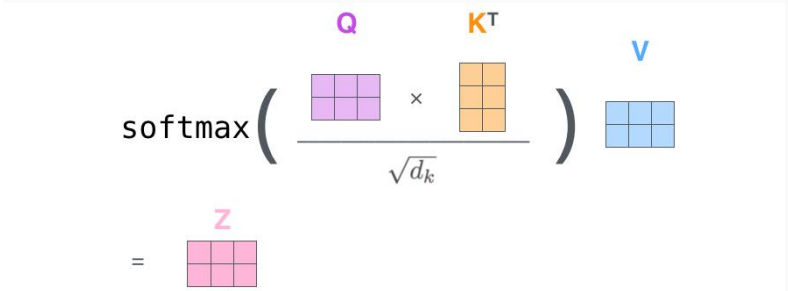
矩阵计算

上面介绍的方法需要一个循环遍历所有的字 x_t，我们可以把上面的向量计算变成矩阵的形式，从而一次计算出所有时刻的输出

第一步就不是计算某个时刻的 qt,kt,vt 了，而是一次计算所有时刻的 Q,K 和 V。计算过程如下图所示，这里的输入是一个矩阵 X，矩阵第 t 行表示第 t 个词的向量表示 x_t

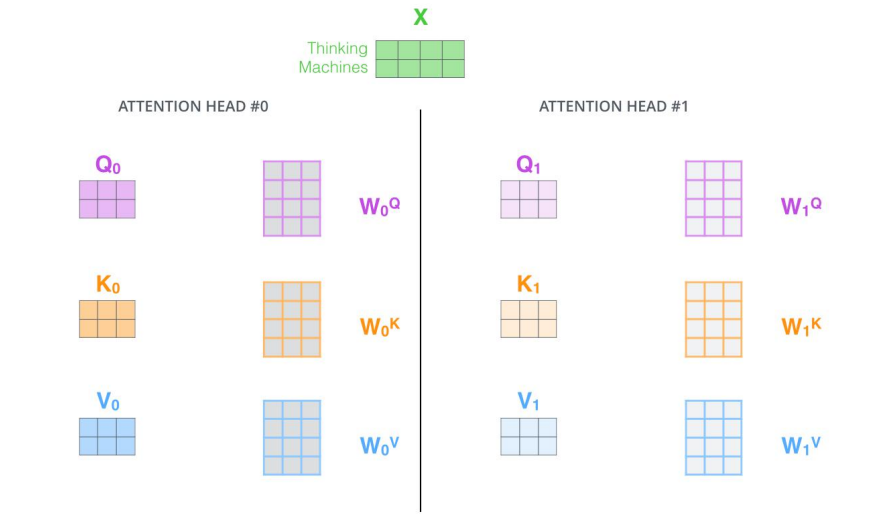


接下来将 Q 和 K^T 相乘，然后除以 d_k（这是论文中提到的一个 trick），经过 softmax 以后再乘以 V 得到输出。

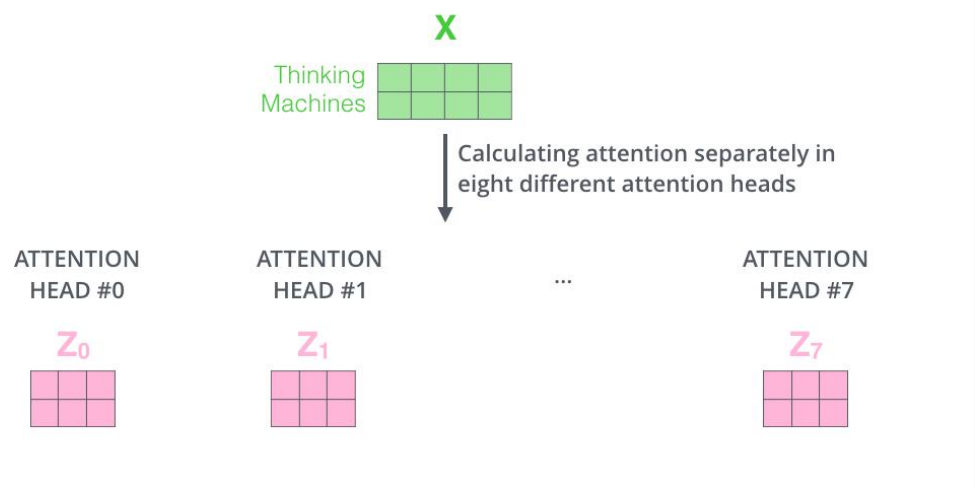


Multi-Head Attention

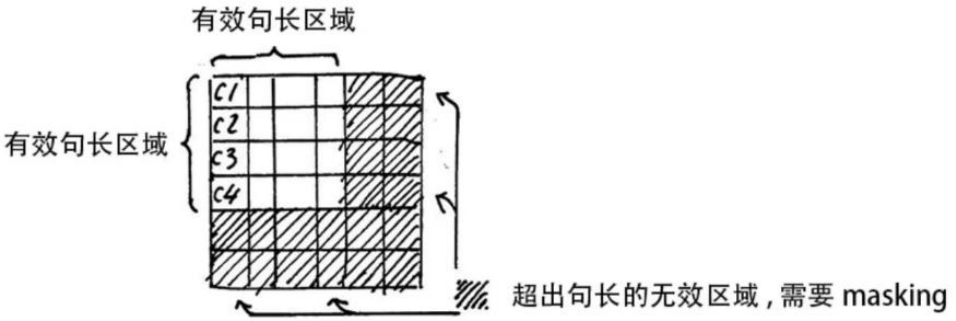
这篇论文还提出了 Multi-Head Attention 的概念。其实很简单，前面定义的一组 Q,K,V 可以让一个词 attend to 相关的词，我们可以定义多组 Q,K,V，让它们分别关注不同的上下文。计算 Q,K,V 的过程还是一样，只不过线性变换的矩阵从一组(WQ,WK,WV) 变成了多组(W0Q,W0K,W0V)，(W1Q,W1K,W1V)，…如下图所示。对于输入矩阵 X，每一组 Q、K 和 V 都可以得到一个输出矩阵 Z。如下图所示



对于输入矩阵 X，每一组 Q、K 和 V 都可以得到一个输出矩阵 Z。如下图所示



Padding Mask



上面 Self Attention 的计算过程中，我们通常使用 mini-batch 来计算，也就是一次计算多句话，即 X 的维度是[batch_size, sequence_length]，sequence_length 是句长，而一个 mini-batch 是由多个不等长的句子组成的，我们需要按照这个 mini-batch 中最大的句长对剩余的句子进行补齐，一般用 0 进行填充，这个过程叫做 padding。

但这时在进行 softmax 就会产生问题。回顾 softmax 函数 $\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$，$e^0$ 是 1，是有值的，这样的话 softmax 中被 padding 的部分就参与了运算，相当于让无效的部分参与了运算，这可能会产生很大的隐患。因此需要做一个 mask 操作，让这些无效的区域不参与运算，一般是给无效区域加一个很大的负数偏置，即

$$Z_{illegal} = Z_{illegal} + bias_{illegal}$$
$$bias_{illegal} \to -\infty$$

(4). 3.残差连接和 Layer Normalization

残差连接

我们在上一步得到了经过 self-attention 加权之后输出，也就是 Attention(Q, K, V)，然后把他们加起来做残差连接

$$X_{embedding} + Self\ Attention(Q,K,V)$$

Layer Normalization

Layer Normalization 的作用是把神经网络中隐藏层归一为标准正态分布，也就是 *i.i.d* 独立同

分布，以起到加快训练速度，加速收敛的作用

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_{ij}$$

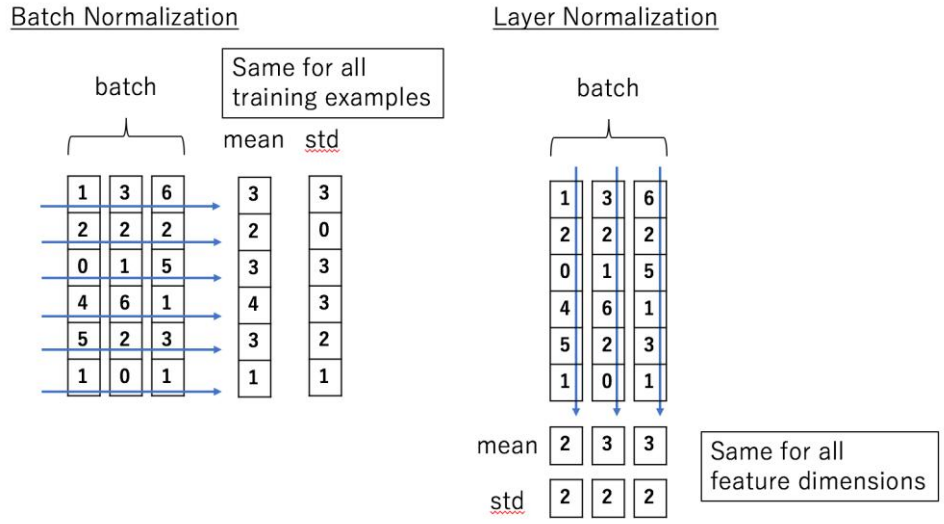
上式以矩阵的列$(column)$为单位求均值;

$$\sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m}(x_{ij} - \mu_j)^2$$

上式以矩阵的列$(column)$为单位求方差

$$LayerNorm(x) = \frac{x_{ij} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

然后用每一列的每一个元素减去这列的均值，再除以这列的标准差，从而得到归一化后的数值，加ϵ 是为了防止分母为 0。



下图展示了更多细节：输入 x1,x2 经 self-attention 层之后变成 z1,z2 ，然后和输入 x1,x2 进行残差连接，经过 LayerNorm 后输出给全连接层。全连接层也有一个残差连接和一个 LayerNorm，最后再输出给下一个 Encoder（每个 Encoder Block 中的 FeedForward 层权重都是共享的）。

(5). 4. Transformer Encoder 整体结构

经过上面 3 个步骤，我们已经基本了解了 Encoder 的主要构成部分，下面我们用公式把一个 Encoder block 的计算过程整理一下：

1). 字向量与位置编码

X=Embedding Lookup(X)+Positional Encoding

2). 自注意力机制

$Q=Linear(X)=XW_Q$

$K=Linear(X)=XW_K$

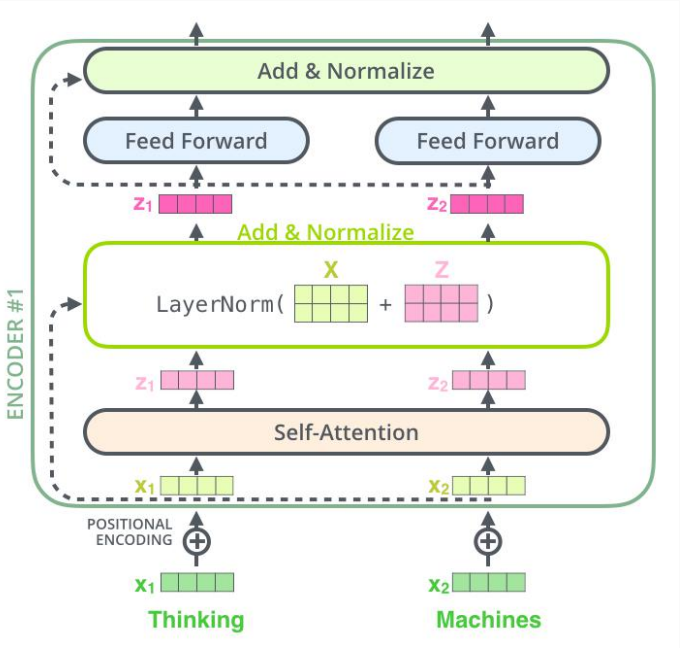
$V=Linear(X)=XW_V$

Xattention=SelfAttention(Q, K, V)

3). self-attention 残差连接与 Layer Normalization

$X_{attention}=X+X_{attention}$

$X_{attention}=\text{LayerNorm}(X_{attention})$



4). 下面进行 Encoder block 结构图中的第 4 部分，也就是 FeedForward，其实就是两层线性映射并用激活函数激活，比如说 ReLU

$X_{hidden}=\text{Linear}(\text{ReLU}(\text{Linear}(X_{attention})))$

5). FeedForward 残差连接与 Layer Normalization

$X_{hidden}=X_{attention}+X_{hidden}X_{hidden}=\text{LayerNorm}(X_{hidden})$

其中

$X_{hidden} \in R^{batch\_size * seq\_len. * embed\_dim}$

(6). 5. Transformer Decoder 整体结构

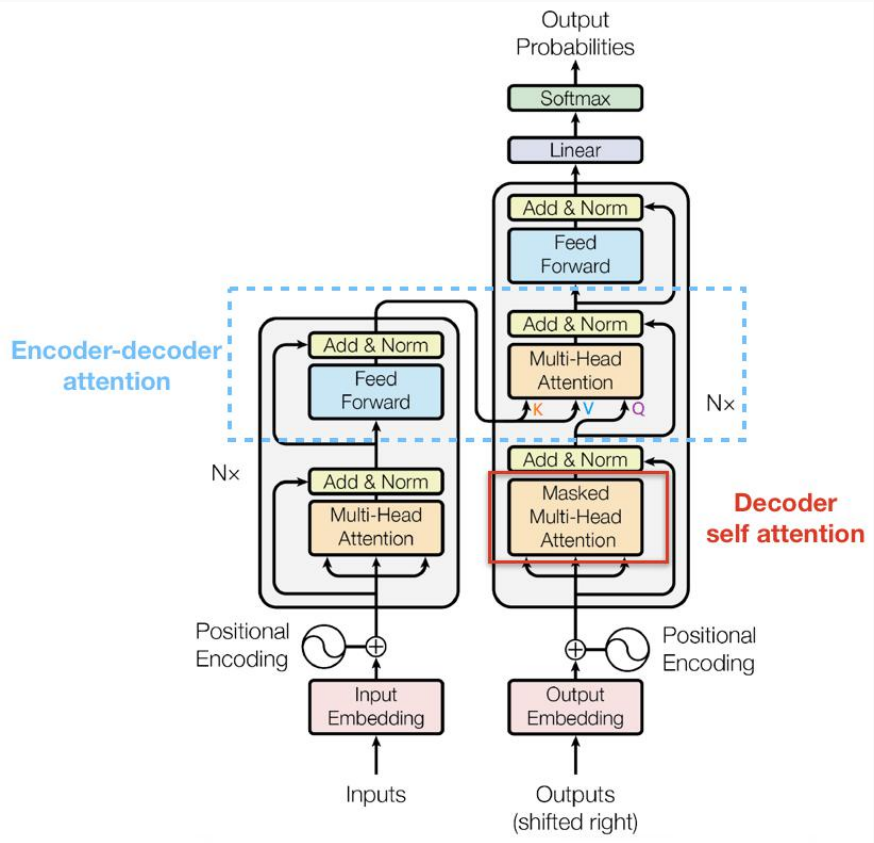
我们先从 HighLevel 的角度观察一下 Decoder 结构，从下到上依次是：

- Masked Multi-Head Self-Attention
- Multi-Head Encoder-Decoder Attention
- FeedForward Network

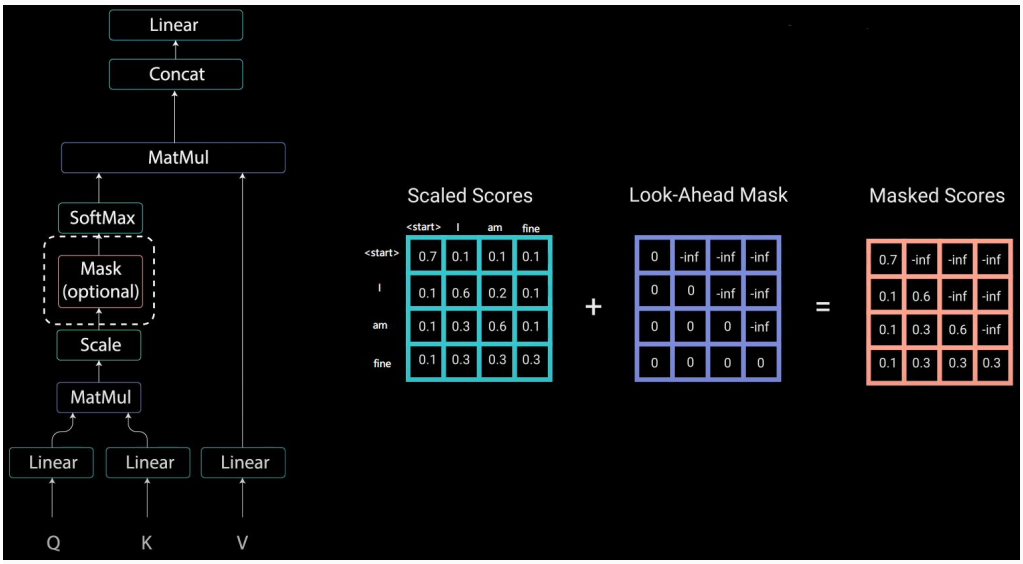
和 Encoder 一样，上面三个部分的每一个部分，都有一个残差连接，后接一个 Layer Normalization。Decoder 的中间部件并不复杂，大部分在前面 Encoder 里我们已经介绍过了，但是 Decoder 由于其特殊的功能，因此在训练时会涉及到一些细节

Masked Self-Attention

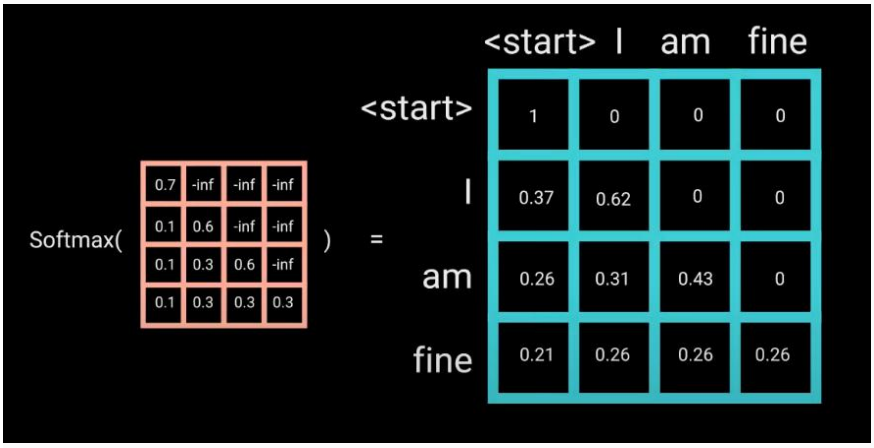
具体来说，传统 Seq2Seq 中 Decoder 使用的是 RNN 模型，因此在训练过程中输入 t 时刻的词，模型无论如何也看不到未来时刻的词，因为循环神经网络是时间驱动的，只有当 t 时刻运算结束了，才能看到 t+1 时刻的词。而 Transformer Decoder 抛弃了 RNN，改为 Self-Attention，由此就产生了一个问题，在训练过程中，整个 ground truth 都暴露在 Decoder 中，这显然是不对的，我们需要对 Decoder 的输入进行一些处理，该处理被称为 Mask。



举个例子，Decoder 的 ground truth 为"<start> I am fine"，我们将这个句子输入到 Decoder 中，经过 WordEmbedding 和 Positional Encoding 之后，将得到的矩阵做三次线性变换（W_Q,W_K,W_V）。然后进行 self-attention 操作，首先通过$\frac{Q\times K^T}{\sqrt{d_k}}$得到 Scaled Scores，接下来非常关键，我们要对 Scaled Scores 进行 Mask，举个例子，当我们输入"I"时，模型目前仅知道包括"I"在内之前所有字的信息，即"<start>"和"I"的信息，不应该让其知道"I"之后词的信息。道理很简单，我们做预测的时候是按照顺序一个字一个字的预测，怎么能这个字都没预测完，就已经知道后面字的信息了呢？Mask 非常简单，首先生成一个下三角全 0，上三角全为负无穷的矩阵，然后将其与 Scaled Scores 相加即可。



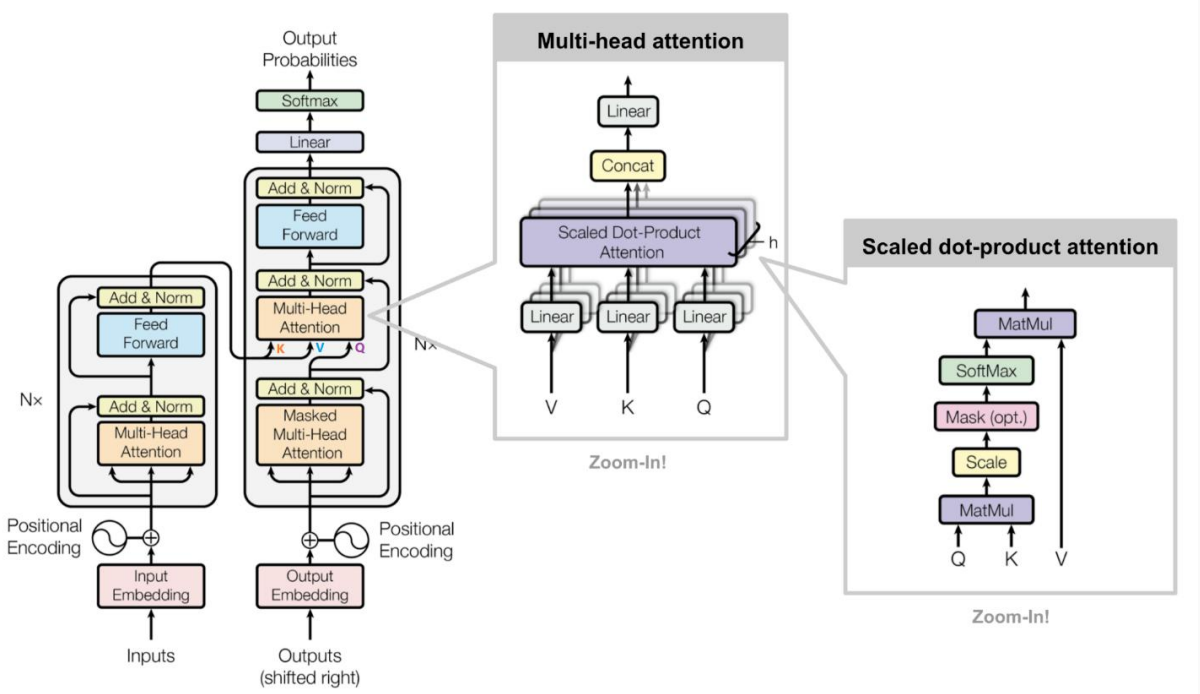
之后再做 softmax，就能将-inf 变为 0，得到的这个矩阵即为每个字之间的权重



Multi-Head Self-Attention 无非就是并行的对上述步骤多做几次，前面 Encoder 也介绍了。

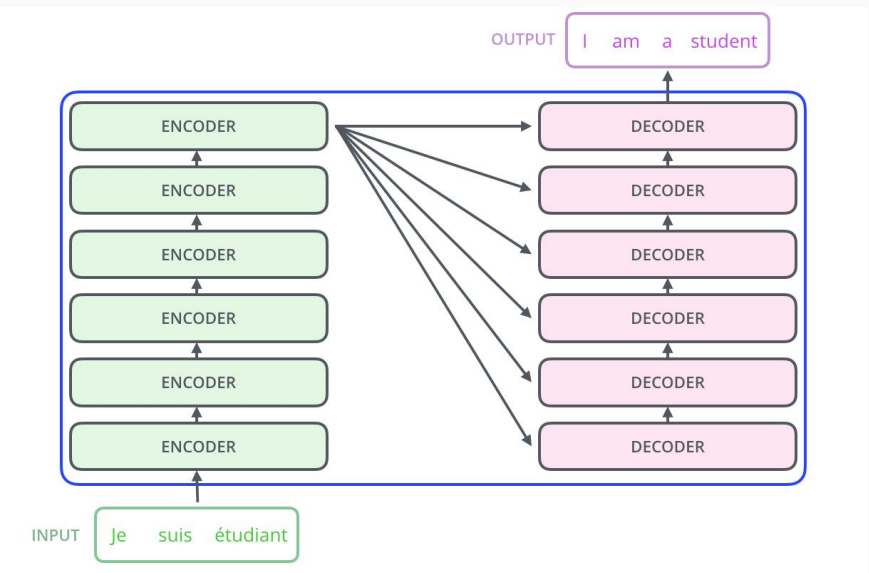
Masked Encoder-Decoder Attention

其实这一部分的计算流程和前面 Masked Self-Attention 很相似，结构也一摸一样，唯一不同的是这里的 K,V 为 Encoder 的输出，Q 为 Decoder 中 Masked Self-Attention 的输出



(7). 6. 总结

到此为止，Transformer 中 95%的内容已经介绍完了，我们用一张图展示其完整结构。不得不说，Transformer 设计的十分巧夺天工。



(8). 问题

下面有几个问题，是我从网上找的，感觉看完之后能对 Transformer 有一个更深的理解

Transformer 为什么需要进行 Multi-head Attention？

原论文中说到进行 Multi-head Attention 的原因是将模型分为多个头，形成多个子空间，可以让模型去关注不同方面的信息，最后再将各个方面的信息综合起来。其实直观上也可以想到，如果自己设计这样的一个模型，必然也不会只做一次 attention，多次 attention 综合的结果至少能够起到增强模型的作用，也可以类比 CNN 中同时使用多个卷积核的作用，直观上讲，多头的注意力有助于网络捕捉到更丰富的特征/信息。

Transformer 相比于 RNN/LSTM，有什么优势？为什么？

RNN 系列的模型，无法并行计算，因为 T 时刻的计算依赖 T-1 时刻的隐层计算结果，而 T-1 时刻的计算依赖 T-2 时刻的隐层计算结果。

Transformer 的特征抽取能力比 RNN 系列的模型要好。

为什么说 Transformer 可以代替 seq2seq？

这里用代替这个词略显不妥当，seq2seq 虽已老，但始终还是有其用武之地，seq2seq 最大的问题在于将 Encoder 端的所有信息压缩到一个固定长度的向量中，并将其作为 Decoder 端首个隐藏状态的输入，来预测 Decoder 端第一个单词(token)的隐藏状态。在输入序列比较长的时候，这样做显然会损失 Encoder 端的很多信息，而且这样一股脑的把该固定向量送入 Decoder 端，Decoder 端不能够关注到其想要关注的信息。Transformer 不但对 seq2seq 模型这两点缺点有了实质性的改进(多头交互式 attention 模块)，而且还引入了 self-attention 模块，让源序列和目标序列首先“自关联”起来，这样的话，源序列和目标序列自身的 embedding 表示所蕴含的信息更加丰富，而且后续的 FFN 层也增强了模型的表达能力，并且 Transformer 并行计算的能力远远超过了 seq2seq 系列模型。

(9). 参考文章

Transformer
The Illustrated Transformer
TRANSFORMERS FROM SCRATCH
Seq2seq pay Attention to Self Attention: Part 2

3. Code of Transformer

```
# -*- coding: utf-8 -*-
'''
  code by Tae Hwan Jung(Jeff Jung) @graykode, Derek Miller @dmmiller612, modify by wmathor
  Reference : https://github.com/jadore801120/attention-is-all-you-need-pytorch
              https://github.com/JayParks/transformer
'''
import math
import torch
import numpy as np
import torch.nn as nn
import torch.optim as optim
import torch.utils.data as Data
# S: Symbol that shows starting of decoding input
# E: Symbol that shows starting of decoding output
# P: Symbol that will fill in blank sequence if current batch data size is short than time steps
sentences = [
    # enc_input           dec_input         dec_output
    ['ich mochte ein bier P', 'S i want a beer .', 'i want a beer . E'],
    ['ich mochte ein cola P', 'S i want a coke .', 'i want a coke . E'],
    ['ich mag das Buch P', 'S i like the book .', 'i like the book . E']
]
# Padding Should be Zero
# src_vocab = {'P': 0, 'ich': 1, 'mochte': 2, 'ein': 3, 'bier': 4, 'cola': 5}
src_vocab = {'P': 0, 'ich': 1, 'mochte': 2, 'ein': 3, 'bier': 4, 'cola': 5,'mag': 6,'das': 7,'Buch': 8}
src_vocab_size = len(src_vocab)

# tgt_vocab = {'P': 0, 'i': 1, 'want': 2, 'a': 3, 'beer': 4, 'coke': 5, 'S': 6, 'E': 7, '.': 8} # word2idx
tgt_vocab = {'P': 0, 'i': 1, 'want': 2, 'a': 3, 'beer': 4, 'coke': 5, 'like': 6,'the': 7,'book': 8,'S': 9, 'E': 10, '.': 11} # word2idx
idx2word = {i: w for i, w in enumerate(tgt_vocab)}
tgt_vocab_size = len(tgt_vocab)
print('-----idx2word=',idx2word)

# src_len = 5  # enc_input max sequence length
# tgt_len = 6  # dec_input(=dec_output) max sequence length
src_len = 6  # enc_input max sequence length
tgt_len = 8  # dec_input(=dec_output) max sequence length
# Transformer Parameters
d_model = 512  # Embedding Size
d_ff = 2048  # FeedForward dimension
d_k = d_v = 64  # dimension of K(=Q), V
n_layers = 6  # number of Encoder of Decoder Layer, 6
n_heads = 8  # number of heads in Multi-Head Attention
```

```
def make_data(sentences):
    enc_inputs, dec_inputs, dec_outputs = [], [], []
    for i in range(len(sentences)):
        print('------sentences_=', sentences[i])
        enc_input = [[src_vocab[n] for n in sentences[i][0].split()]]  # [[1, 2, 3, 4, 0], [1, 2, 3, 5, 0]]
        print('-----input=',enc_input)
        print('----tgt_vocab=',tgt_vocab)
        dec_input = [[tgt_vocab[n] for n in sentences[i][1].split()]]  # [[6, 1, 2, 3, 4, 8], [6, 1, 2, 3, 5, 8]]
        print('-----dec_input=',dec_input)
        dec_output = [[tgt_vocab[n] for n in sentences[i][2].split()]]  # [[1, 2, 3, 4, 8, 7], [1, 2, 3, 5, 8, 7]]
        enc_inputs.extend(enc_input)
        dec_inputs.extend(dec_input)
        dec_outputs.extend(dec_output)
    return torch.LongTensor(enc_inputs), torch.LongTensor(dec_inputs), torch.LongTensor(dec_outputs)

enc_inputs, dec_inputs, dec_outputs = make_data(sentences)
class MyDataSet(Data.Dataset):
    def __init__(self, enc_inputs, dec_inputs, dec_outputs):
        super(MyDataSet, self).__init__()
        self.enc_inputs = enc_inputs
        self.dec_inputs = dec_inputs
        self.dec_outputs = dec_outputs
    def __len__(self):
        return self.enc_inputs.shape[0]
    def __getitem__(self, idx):
        return self.enc_inputs[idx], self.dec_inputs[idx], self.dec_outputs[idx]
loader = Data.DataLoader(MyDataSet(enc_inputs, dec_inputs, dec_outputs), 1, True)

ttp=MyDataSet(enc_inputs, dec_inputs, dec_outputs)
print('---------tp2----------------')
(r1,r2,r3)=ttp[0]
print('MyDataSet_1=',ttp[0])
dataiter=iter(loader)
print('----dataiter_next=',dataiter.next())
######### Train Phase ##########################################
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, dropout=0.1, max_len=5000):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(p=dropout)
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-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).transpose(0, 1)
        self.register_buffer('pe', pe)
    def forward(self, x):
        '''
        x: [seq_len, batch_size, d_model]
        '''
        x = x + self.pe[:x.size(0), :]
        return self.dropout(x)
def get_attn_pad_mask(seq_q, seq_k):
    '''
    seq_q: [batch_size, seq_len]
    seq_k: [batch_size, seq_len]
    seq_len could be src_len or it could be tgt_len
    seq_len in seq_q and seq_len in seq_k maybe not equal
    '''
    print('---seq_q=',seq_q)
    print('---seq_k=',seq_k)
    batch_size, len_q = seq_q.size()
    batch_size, len_k = seq_k.size()
    print('--batch_size=',batch_size,'--len_q=',len_q,'--len_k=',len_k)
    # eq(zero) is PAD token
    pad_attn_mask = seq_k.data.eq(0).unsqueeze(1)  # [batch_size, 1, len_k], False is masked
    print('-----pad_attn_mask=',pad_attn_mask)
    return pad_attn_mask.expand(batch_size, len_q, len_k)  # [batch_size, len_q, len_k]
def get_attn_subsequence_mask(seq):
    '''
    seq: [batch_size, tgt_len]
    '''
    attn_shape = [seq.size(0), seq.size(1), seq.size(1)]
    subsequence_mask = np.triu(np.ones(attn_shape), k=1)  # Upper triangular matrix
    subsequence_mask = torch.from_numpy(subsequence_mask).byte()
    return subsequence_mask  # [batch_size, tgt_len, tgt_len]
class ScaledDotProductAttention(nn.Module):
    def __init__(self):
        super(ScaledDotProductAttention, self).__init__()
    def forward(self, Q, K, V, attn_mask):
        '''
        Q: [batch_size, n_heads, len_q, d_k]
        K: [batch_size, n_heads, len_k, d_k]
        V: [batch_size, n_heads, len_v(=len_k), d_v]
        attn_mask: [batch_size, n_heads, seq_len, seq_len]
        '''
        scores = torch.matmul(Q, K.transpose(-1, -2)) / np.sqrt(d_k)  # scores : [batch_size, n_heads, len_q, len_k]
```

```
        scores.masked_fill_(attn_mask, -1e9)  # Fills elements of self tensor with value where mask is True.
        attn = nn.Softmax(dim=-1)(scores)
        context = torch.matmul(attn, V)  # [batch_size, n_heads, len_q, d_v]
        return context, attn
class MultiHeadAttention(nn.Module):
    def __init__(self):
        super(MultiHeadAttention, self).__init__()
        self.W_Q = nn.Linear(d_model, d_k * n_heads, bias=False)
        self.W_K = nn.Linear(d_model, d_k * n_heads, bias=False)
        self.W_V = nn.Linear(d_model, d_v * n_heads, bias=False)
        self.fc = nn.Linear(n_heads * d_v, d_model, bias=False)
    def forward(self, input_Q, input_K, input_V, attn_mask):
        '''
        input_Q: [batch_size, len_q, d_model]
        input_K: [batch_size, len_k, d_model]
        input_V: [batch_size, len_v(=len_k), d_model]
        attn_mask: [batch_size, seq_len, seq_len]
        '''
        residual, batch_size = input_Q, input_Q.size(0)
        # (B, S, D) -proj-> (B, S, D_new) -split-> (B, S, H, W) -trans-> (B, H, S, W)
        Q = self.W_Q(input_Q).view(batch_size, -1, n_heads, d_k).transpose(1, 2)  # Q: [batch_size, n_heads, len_q,
d_k]
        K = self.W_K(input_K).view(batch_size, -1, n_heads, d_k).transpose(1, 2)  # K: [batch_size, n_heads, len_k,
d_k]
        V = self.W_V(input_V).view(batch_size, -1, n_heads, d_v).transpose(1,
                                                        2)  # V: [batch_size, n_heads, len_v(=len_k), d_v]
        attn_mask = attn_mask.unsqueeze(1).repeat(1, n_heads, 1,
                                          1)  # attn_mask : [batch_size, n_heads, seq_len, seq_len]
        # context: [batch_size, n_heads, len_q, d_v], attn: [batch_size, n_heads, len_q, len_k]
        context, attn = ScaledDotProductAttention()(Q, K, V, attn_mask)
        context = context.transpose(1, 2).reshape(batch_size, -1,
                                        n_heads * d_v)  # context: [batch_size, len_q, n_heads * d_v]
        output = self.fc(context)  # [batch_size, len_q, d_model]
        #return nn.LayerNorm(d_model).cuda()(output + residual), attn
        return nn.LayerNorm(d_model)(output + residual), attn
class PoswiseFeedForwardNet(nn.Module):
    def __init__(self):
        super(PoswiseFeedForwardNet, self).__init__()
        self.fc = nn.Sequential(
            nn.Linear(d_model, d_ff, bias=False),
            nn.ReLU(),
            nn.Linear(d_ff, d_model, bias=False)
        )
    def forward(self, inputs):
```

```
        '''
        inputs: [batch_size, seq_len, d_model]
        '''
        residual = inputs
        output = self.fc(inputs)
        #return nn.LayerNorm(d_model).cuda()(output + residual)  # [batch_size, seq_len, d_model]
        return nn.LayerNorm(d_model)(output + residual)  # [batch_size, seq_len, d_model]
class EncoderLayer(nn.Module):
    def __init__(self):
        super(EncoderLayer, self).__init__()
        self.enc_self_attn = MultiHeadAttention()
        self.pos_ffn = PoswiseFeedForwardNet()

    def forward(self, enc_inputs, enc_self_attn_mask):
        '''
        enc_inputs: [batch_size, src_len, d_model]
        enc_self_attn_mask: [batch_size, src_len, src_len]
        '''
        # enc_outputs: [batch_size, src_len, d_model], attn: [batch_size, n_heads, src_len, src_len]
        enc_outputs, attn = self.enc_self_attn(enc_inputs, enc_inputs, enc_inputs,
                                               enc_self_attn_mask)  # enc_inputs to same Q,K,V
        enc_outputs = self.pos_ffn(enc_outputs)  # enc_outputs: [batch_size, src_len, d_model]
        return enc_outputs, attn
class DecoderLayer(nn.Module):
    def __init__(self):
        super(DecoderLayer, self).__init__()
        self.dec_self_attn = MultiHeadAttention()
        self.dec_enc_attn = MultiHeadAttention()
        self.pos_ffn = PoswiseFeedForwardNet()
    def forward(self, dec_inputs, enc_outputs, dec_self_attn_mask, dec_enc_attn_mask):
        '''
        dec_inputs: [batch_size, tgt_len, d_model]
        enc_outputs: [batch_size, src_len, d_model]
        dec_self_attn_mask: [batch_size, tgt_len, tgt_len]
        dec_enc_attn_mask: [batch_size, tgt_len, src_len]
        '''
        # dec_outputs: [batch_size, tgt_len, d_model], dec_self_attn: [batch_size, n_heads, tgt_len, tgt_len]
        dec_outputs, dec_self_attn = self.dec_self_attn(dec_inputs, dec_inputs, dec_inputs, dec_self_attn_mask)
        # dec_outputs: [batch_size, tgt_len, d_model], dec_enc_attn: [batch_size, h_heads, tgt_len, src_len]
        dec_outputs, dec_enc_attn = self.dec_enc_attn(dec_outputs, enc_outputs, enc_outputs, dec_enc_attn_mask)
        dec_outputs = self.pos_ffn(dec_outputs)  # [batch_size, tgt_len, d_model]
        return dec_outputs, dec_self_attn, dec_enc_attn
class Encoder(nn.Module):
    def __init__(self):
```

```
        super(Encoder, self).__init__()
        self.src_emb = nn.Embedding(src_vocab_size, d_model)
        self.pos_emb = PositionalEncoding(d_model)
        self.layers = nn.ModuleList([EncoderLayer() for _ in range(n_layers)])
    def forward(self, enc_inputs):
        '''
        enc_inputs: [batch_size, src_len]
        '''
        enc_outputs = self.src_emb(enc_inputs)  # [batch_size, src_len, d_model]
        enc_outputs = self.pos_emb(enc_outputs.transpose(0, 1)).transpose(0, 1)  # [batch_size, src_len, d_model]
        enc_self_attn_mask = get_attn_pad_mask(enc_inputs, enc_inputs)  # [batch_size, src_len, src_len]
        # print('----in encoder,enc_self_attn_mask=',enc_self_attn_mask)
        enc_self_attns = []
        for layer in self.layers:
            # enc_outputs: [batch_size, src_len, d_model], enc_self_attn: [batch_size, n_heads, src_len, src_len]
            enc_outputs, enc_self_attn = layer(enc_outputs, enc_self_attn_mask)
            enc_self_attns.append(enc_self_attn)
        return enc_outputs, enc_self_attns
class Decoder(nn.Module):
    def __init__(self):
        super(Decoder, self).__init__()
        self.tgt_emb = nn.Embedding(tgt_vocab_size, d_model)
        self.pos_emb = PositionalEncoding(d_model)
        self.layers = nn.ModuleList([DecoderLayer() for _ in range(n_layers)])
    def forward(self, dec_inputs, enc_inputs, enc_outputs):
        '''
        dec_inputs: [batch_size, tgt_len]
        enc_intpus: [batch_size, src_len]
        enc_outputs: [batsh_size, src_len, d_model]
        '''
        dec_outputs = self.tgt_emb(dec_inputs)  # [batch_size, tgt_len, d_model]
        # dec_outputs = self.pos_emb(dec_outputs.transpose(0, 1)).transpose(0, 1).cuda()  # [batch_size, tgt_len,
d_model]
        # dec_self_attn_pad_mask = get_attn_pad_mask(dec_inputs, dec_inputs).cuda()  # [batch_size, tgt_len,
tgt_len]
        # dec_self_attn_subsequence_mask = get_attn_subsequence_mask(dec_inputs).cuda()  # [batch_size, tgt_len,
tgt_len]
        # dec_self_attn_mask = torch.gt((dec_self_attn_pad_mask + dec_self_attn_subsequence_mask),
        #                               0).cuda()  # [batch_size, tgt_len, tgt_len]
        dec_outputs = self.pos_emb(dec_outputs.transpose(0, 1)).transpose(0, 1)  # [batch_size, tgt_len, d_model]
        dec_self_attn_pad_mask = get_attn_pad_mask(dec_inputs, dec_inputs)  # [batch_size, tgt_len, tgt_len]
        # print('----in decoder,dec_self_attn_pad_mask=',dec_self_attn_pad_mask)
        dec_self_attn_subsequence_mask = get_attn_subsequence_mask(dec_inputs)  # [batch_size, tgt_len, tgt_len]
        dec_self_attn_mask = torch.gt((dec_self_attn_pad_mask + dec_self_attn_subsequence_mask),
```

```
                                0)  # [batch_size, tgt_len, tgt_len]
        dec_enc_attn_mask = get_attn_pad_mask(dec_inputs, enc_inputs)  # [batc_size, tgt_len, src_len]
        dec_self_attns, dec_enc_attns = [], []
        for layer in self.layers:
            # dec_outputs: [batch_size, tgt_len, d_model], dec_self_attn: [batch_size, n_heads, tgt_len, tgt_len],
dec_enc_attn: [batch_size, h_heads, tgt_len, src_len]
            dec_outputs, dec_self_attn, dec_enc_attn = layer(dec_outputs, enc_outputs, dec_self_attn_mask,
                                                             dec_enc_attn_mask)
            dec_self_attns.append(dec_self_attn)
            dec_enc_attns.append(dec_enc_attn)
        return dec_outputs, dec_self_attns, dec_enc_attns

class Transformer(nn.Module):
    def __init__(self):
        super(Transformer, self).__init__()
        #self.encoder = Encoder().cuda()
        self.encoder = Encoder()
        #self.decoder = Decoder().cuda()
        self.decoder = Decoder()
        #self.projection = nn.Linear(d_model, tgt_vocab_size, bias=False).cuda()
        self.projection = nn.Linear(d_model, tgt_vocab_size, bias=False)
    def forward(self, enc_inputs, dec_inputs):
        '''
        enc_inputs: [batch_size, src_len]
        dec_inputs: [batch_size, tgt_len]
        '''
        # tensor to store decoder outputs
        # outputs = torch.zeros(batch_size, tgt_len, tgt_vocab_size).to(self.device)
        # enc_outputs: [batch_size, src_len, d_model], enc_self_attns: [n_layers, batch_size, n_heads, src_len, src_len]
        enc_outputs, enc_self_attns = self.encoder(enc_inputs)
        # print('--in Transformer,enc_outputs=',enc_outputs.size())
        # print('--in Transformer,len_enc_self_attns=', len(enc_self_attns),'----sub_dim=',len(enc_self_attns[3]),'----
subsub_dim=',len(enc_self_attns[3][0]))
        # print('--in Transformer,enc_self_attns=', (enc_self_attns[0][0]).size())
        # # print('--in Transformer,enc_self_attns=', enc_self_attns)
        # dec_outpus: [batch_size, tgt_len, d_model], dec_self_attns: [n_layers, batch_size, n_heads, tgt_len, tgt_len],
dec_enc_attn: [n_layers, batch_size, tgt_len, src_len]
        dec_outputs, dec_self_attns, dec_enc_attns = self.decoder(dec_inputs, enc_inputs, enc_outputs)
        # print('--in Transformer,dec_outputs=',dec_outputs.size())
        # print('--in Transformer,len_dec_self_attns=', len(dec_self_attns),'----sub_dim=',len(dec_self_attns[3]),'----
subsub_dim=',len(dec_self_attns[3][0]))
        # print('--in Transformer,dec_self_attns=', (dec_self_attns[0][0]).size())
        # # print('--in Transformer,dec_self_attns=', dec_self_attns)
        # print('--in Transformer,len_dec_enc_attns=', len(dec_enc_attns), '----sub_dim=', len(dec_enc_attns[3]),
```

```
        #       '----subsub_dim=', len(dec_enc_attns[3][0]))
        # print('--in Transformer,dec_enc_attns=', (dec_enc_attns[0][0]).size())
        dec_logits = self.projection(dec_outputs)  # dec_logits: [batch_size, tgt_len, tgt_vocab_size]
        return dec_logits.view(-1, dec_logits.size(-1)), enc_self_attns, dec_self_attns, dec_enc_attns

#model = Transformer().cuda()
model = Transformer()
criterion = nn.CrossEntropyLoss(ignore_index=0)
optimizer = optim.SGD(model.parameters(), lr=1e-3, momentum=0.99)
for epoch in range(10):
    print('                              ')
    print('---------------------------seperation-----------------------------------')
    for enc_inputs, dec_inputs, dec_outputs in loader:
        '''
        enc_inputs: [batch_size, src_len]
        dec_inputs: [batch_size, tgt_len]
        dec_outputs: [batch_size, tgt_len]
        '''
        # enc_inputs, dec_inputs, dec_outputs = enc_inputs.cuda(), dec_inputs.cuda(), dec_outputs.cuda()
        enc_inputs, dec_inputs, dec_outputs = enc_inputs, dec_inputs, dec_outputs
        # outputs: [batch_size * tgt_len, tgt_vocab_size]
        outputs, enc_self_attns, dec_self_attns, dec_enc_attns = model(enc_inputs, dec_inputs)
        # print('--enc_inputs=',enc_inputs)
        # print('--dec_inputs=',dec_inputs)
        # print('--dec_outputs=',dec_outputs.view(-1))
        # print('--outputs=', outputs)
        loss = criterion(outputs, dec_outputs.view(-1))
        print('Epoch:', '%04d' % (epoch + 1), 'loss =', '{:.6f}'.format(loss))
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    if epoch%10==0:
        torch.save(model.state_dict(),'transformer_'+str(epoch)+'.pth')
################ Test Phase ###################################
def greedy_decoder(model, enc_input, start_symbol):
    """
    For simplicity, a Greedy Decoder is Beam search when K=1. This is necessary for inference as we don't know
the  target sequence input. Therefore we try to generate the target input word by word, then feed it into the
transformer.
    Starting Reference: http://nlp.seas.harvard.edu/2018/04/03/attention.html#greedy-decoding
    :param model: Transformer Model
    :param enc_input: The encoder input
    :param start_symbol: The start symbol. In this example it is 'S' which corresponds to index 4
    :return: The target input
```

```
    """
    # print('-----model=',model)
    print('---in greedy func, enc_input=', enc_input)
    enc_outputs, enc_self_attns = model.encoder(enc_input)
    print('---in greedy func, enc_outputs_size=',enc_outputs.size())
    print('---in greedy func, len_enc_self_attns_size=', len(enc_self_attns),'len_enc_self_attns_size[0]=',
len(enc_self_attns[0]))
    dec_input = torch.zeros(1, tgt_len).type_as(enc_input.data)
    next_symbol = start_symbol
    for i in range(0, tgt_len):
        dec_input[0][i] = next_symbol
        dec_outputs, _, _ = model.decoder(dec_input, enc_input, enc_outputs)
        projected = model.projection(dec_outputs)
        prob = projected.squeeze(0).max(dim=-1, keepdim=False)[1]
        next_word = prob.data[i]
        next_symbol = next_word.item()
    return dec_input
# Test
print('----iter_loader=',iter(loader))
print('----next_iter_loader=',next(iter(loader)))
enc_inputs, _, _ = next(iter(loader))
greedy_dec_input = greedy_decoder(model, enc_inputs[0].view(1, -1), start_symbol=tgt_vocab["S"])
print('----greedy_dec_input=',greedy_dec_input)
predict, _, _, _ = model(enc_inputs[0].view(1, -1), greedy_dec_input)
# print('-----predict1=',predict)
predict = predict.data.max(1, keepdim=True)[1]
print('-----predict2=',predict)
print('----enc_inpts=',enc_inputs)
print(enc_inputs[0], '->', [idx2word[n.item()] for n in predict.squeeze()])
```