**Transformer实现详解及注意力机制深度分析**

[1. Transformer的PyTorch实现详解 2](#_Toc10485)

[(1). 数据预处理 2](#_Toc27945)

[(2). 模型参数 3](#_Toc14818)

[(3). Positional Encoding 4](#_Toc18472)

[(4). Pad Mask 5](#_Toc16054)

[(5). Subsequence Mask 5](#_Toc15067)

[(6). ScaledDotProductAttention 6](#_Toc6959)

[(7). MultiHeadAttention 7](#_Toc24412)

[(8). FeedForward Layer 7](#_Toc30983)

[(9). Encoder Layer 8](#_Toc30815)

[(10). Encoder 8](#_Toc22657)

[(11). Decoder Layer 9](#_Toc5132)

[(12). Decoder 10](#_Toc10828)

[(13). Transformer 10](#_Toc14037)

[(14). 模型 & 损失函数 & 优化器 11](#_Toc25996)

[(15). 训练 11](#_Toc21992)

[(16). 测试 12](#_Toc25344)

[2. Transformer模块机理分析 14](#_Toc26147)

[(1). 0.Transformer直观认识 14](#_Toc20137)

[(2). 1. Positional Encoding 16](#_Toc11191)

[(3). 2. Self Attention Mechanism 18](#_Toc19294)

[(4). 3.残差连接和Layer Normalization 23](#_Toc6112)

[(5). 4. Transformer Encoder整体结构 24](#_Toc40)

[(6). 5. Transformer Decoder整体结构 24](#_Toc11607)

[(7). 解码器端详细分析 26](#_Toc17761)

[(8). Transformer整体框架分析 36](#_Toc26993)

[(9). Dropout函数 37](#_Toc6120)

[(10). 问题 37](#_Toc24319)

[(11). 参考文章 38](#_Toc27375)

[3. Code of Transformer 38](#_Toc22897)

[4. 注意力机制深度分析 46](#_Toc29990)

[(1). Encoder-Decoder框架 46](#_Toc20936)

[(2). Attention模型 47](#_Toc15441)

[(3). Attention机制的本质思想 52](#_Toc3159)

[(4). Self Attention模型 54](#_Toc31379)

[(5). Attention机制的应用 56](#_Toc11628)

## Transformer的PyTorch实现详解

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

[B 站视频讲解](https://www.bilibili.com/video/BV1mk4y1q7eK?p=2" \t "_blank)

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](https://wmathor.com/index.php/archives/1438/)，方能达到醍醐灌顶，事半功倍的效果

### 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；token指单词或词语；
* 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):
* #本部分对训练和测试数据集进行处理，使得其符合pytorch对数据集的需求
* **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']

### 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

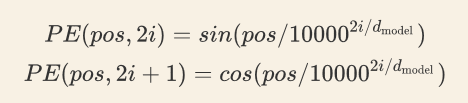
上面都比较简单，下面开始涉及到模型就比较复杂了，因此我会将模型拆分成以下几个部分进行讲解

* 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

### Positional Encoding

本部分实现2.(2)部分所述的以下功能：



* **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) #定义nn.Dropout这个功能Module
* pe=torch.zeros(max\_len,d\_model) #将一个d\_model\*max\_len维度的0矩阵赋值给pe
* position = torch.arange(0, max\_len, dtype=torch.float).unsqueeze(1)
* #生成0~maxlen-1长度的值的序列，值的范围是0~maxlen-1；
* # x = torch.arange(1.0,6.0)
* # print(x)
* # 输出：tensor([1., 2., 3., 4., 5.])
* div\_term = torch.exp(torch.arange(0, d\_model, 2).float() \* (-math.log(10000.0) / d\_model))
* # math.log(10000.0)求取=10000函数式中x的值
* # div\_term实现的功能为：
* pe[:, 0::2] = torch.sin(position \* div\_term)
* pe[:, 1::2] = torch.cos(position \* div\_term)
* #pe[0::2]表示pe中从序号0对应的元素开始,步长为2,至最后一个序号对应的元素。
* pe = pe.unsqueeze(0).transpose(0, 1)
* self.register\_buffer('pe', pe)
* # 以上部分可以提取出来单独组成文件进行调试和测试
* **def** **forward**(self, x):
* '''
* x: [seq\_len, batch\_size, d\_model]
* '''
* x = x + self.pe[:x.size(0), :]
* # 由序列长度(seq\_len)，批处理数据条数(batch\_size)，和嵌入维度(d\_model)确定的位置编码
* **return** self.dropout(x)
* # self.dropout(x)将tensor类型变量x中指定比例(p,p为0~1之间的一个浮点数)的元素赋值为0，剩余值除以(1-p)，从而达到计算结果整体与dropout处理前近似相等的效果。

### 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()
* # seq\_q.size()返回seq\_q矩阵的行数和列数；这个结果是numpy类型的结果数据。
* # seq\_q.shape返回seq\_q矩阵的行数和列数，这个结果是tensor类型的结果数据。
* 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]
* # pad\_attn\_mask的维度是1\*len\_k; 最后一句将pad\_attn\_mask的维度扩展为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**

### 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]

# 运算结果是类似于以下形式的mask矩阵：

[0 1 1 1

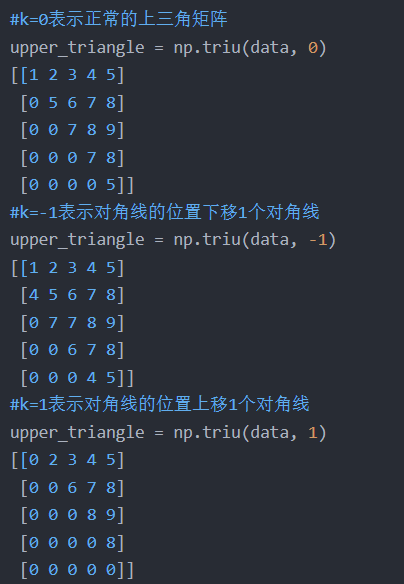
0 0 1 1

0 0 0 1

0 0 0 0]

Subsequence Mask 只有 Decoder 会用到，主要作用是屏蔽未来时刻单词的信息。主要用于decoder中第一级的Masked Multi-Head Self-Attention模块(参见2.6部分的分析)。由于此模块的输入为上一个时刻decoder模块整体的输出，因此此模块只能看到当前时刻及以前时刻所预测的输出结果。所以这些结果做自注意力分析时，只能与当前时刻及以前时刻的结果做自注意力分析，从而用一个上三角mask矩阵屏蔽掉不可能出现的当前时刻之后时刻的预测数据。

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



### 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

### 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

### 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

### Encoder Layer

* **class** **EncoderLayer**(nn.Module):
* # 每层encoder层中包含的所有子模块
* **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

### Encoder

* **class** **Encoder**(nn.Module):
* **def** **\_\_init\_\_**(self):
* super(Encoder, self).\_\_init\_\_()
* self.src\_emb = nn.Embedding(src\_vocab\_size, d\_model)
* #src\_vocab\_size是输入值列表的长度；d\_model是嵌入维度；nn.Embedding将自然编码转换为嵌入式维度；
* 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: # self.layers包含n\_layers个Encoderlayer
* # 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
* # enc\_self\_attns是不同层所输出enc\_self\_attn模块拼接后的list类型的数据，其中的每个元素是tensor类型的数据。这个输出不作为下一层的输入，只是作为备用的数据。

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

### 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

### 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]

### 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。

### 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](https://wmathor.com/index.php/archives/1451/)的最下面，稍微提了一下

### 训练

本部对transformer模型中的参数进行训练。

* **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()
* If epoch%20==0;
* torch.save(model,”transformer\_”+str(epoch)+”.pth”)
* 载入模型采用以下语句：
* model=torch.load(“./transformer\_280.pth”)

不包含训练的独立测试模块，在载入训练好的模型之前，需要有模型中所包含模型名称对应的模型的定义。因为在载入模型时无法解释模型中对应的模块所实现的功能。

### 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）](https://colab.research.google.com/drive/15yTJSjZpYuIWzL9hSbyThHLer4iaJjBD?usp=sharing" \t "_blank)  
Github 项目地址：nip-tutorial

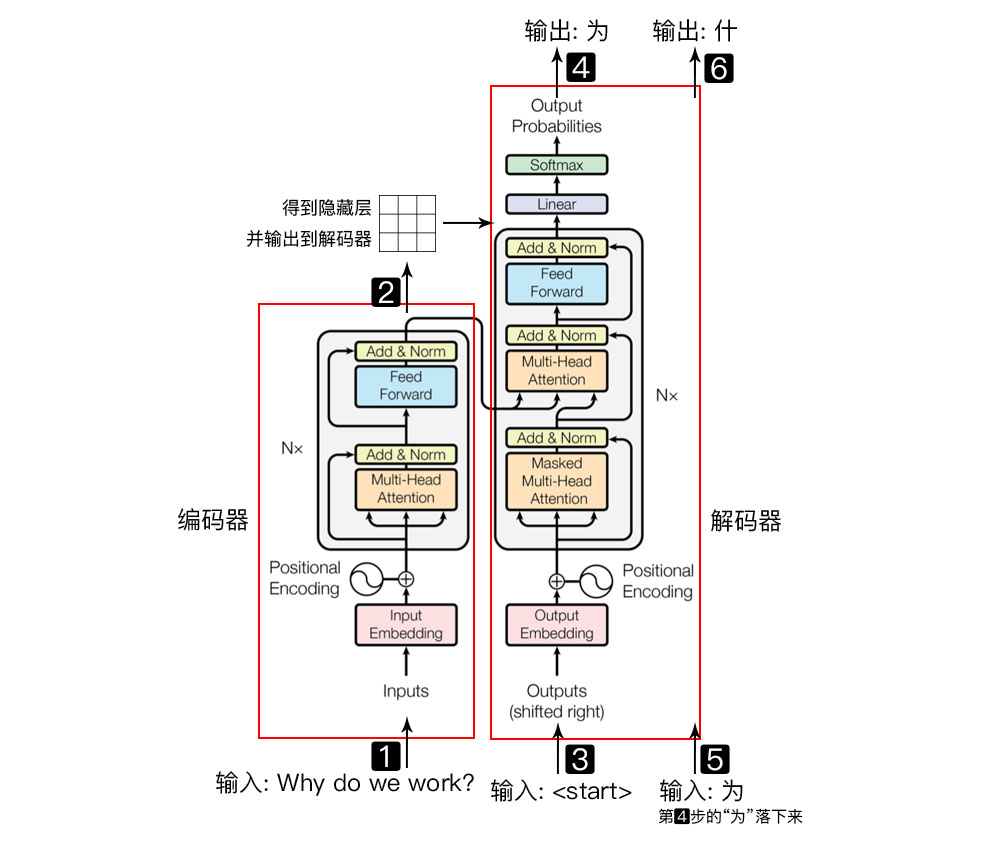
## Transformer模块机理分析

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

Transformer是谷歌大脑在2017年底发表的论文[attention is all you need](https://arxiv.org/pdf/1706.03762.pdf" \t "_blank)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

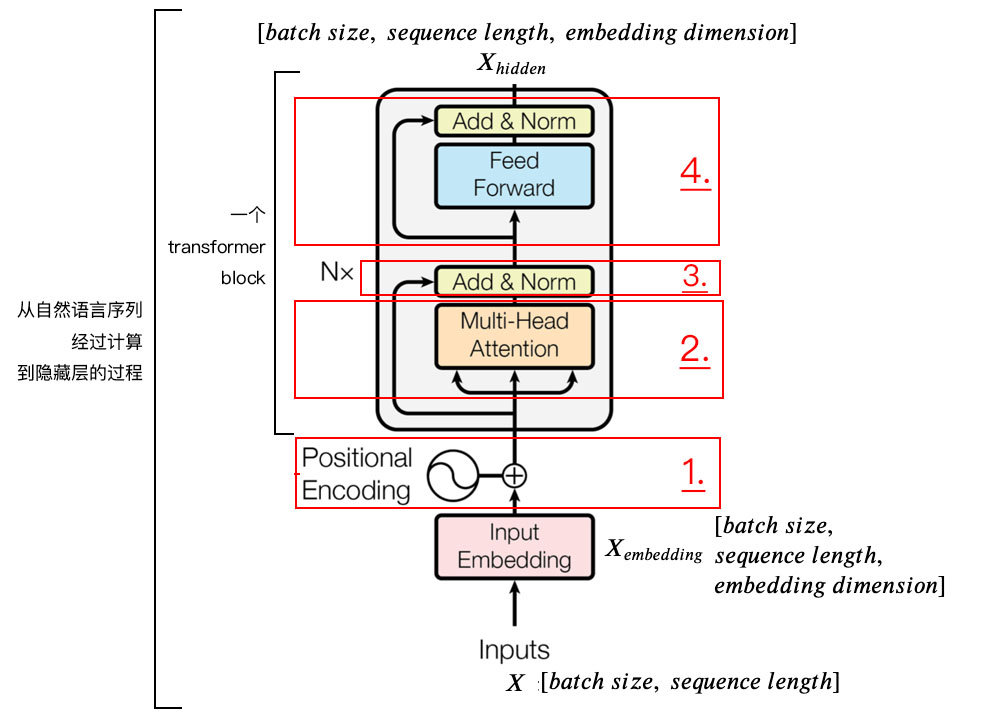
### 0.Transformer直观认识

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。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。



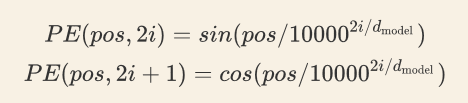
上图为Transformer Encoder Block结构图，注意：下面的内容标题编号分别对应着图中1,2,3,4个方框的序号。Transformer这种架构的框架性原理，参见4.4部分的介绍和分析。

### 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:



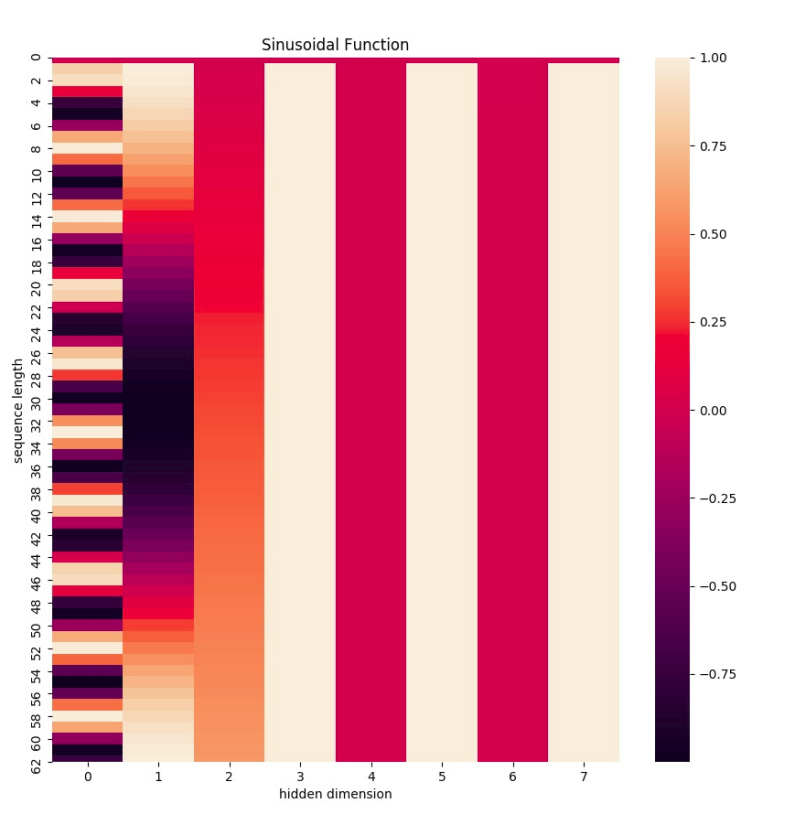
上式中PE代表Positional Encoding；2i/dmodel中的dmodel是个固定的值。i越大，则sin输入项中的10000^(2i/dmodel)的值越大；(pos/10000^(2i/dmodel))的值就越小，从而使得PE(pos,2i)和PE(pos,2i+1)的变动幅度越小。上述PE的设计使得i类似于权重，i越小，则对应的位置的权重波动越剧烈。

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)，即包含最多词语的句子中所包含词语的数量。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∗2π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](https://wmathor.com/index.php/archives/1453/)

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")

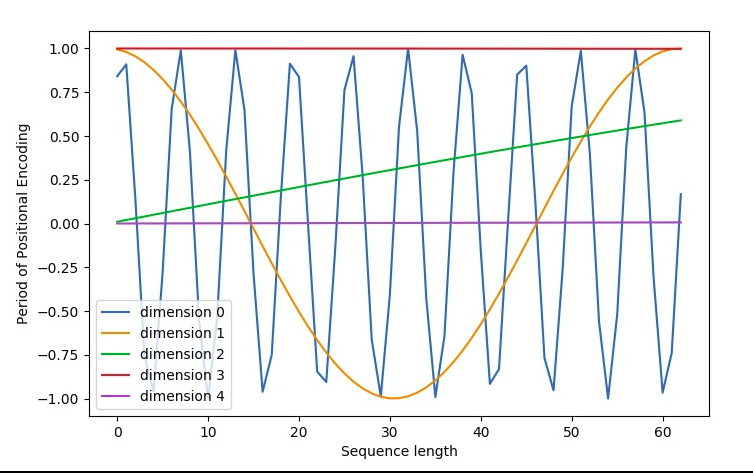
plt.plot(positional\_encoding[1:, 4], label="dimension 4")

plt.legend()

plt.xlabel("Sequence length")

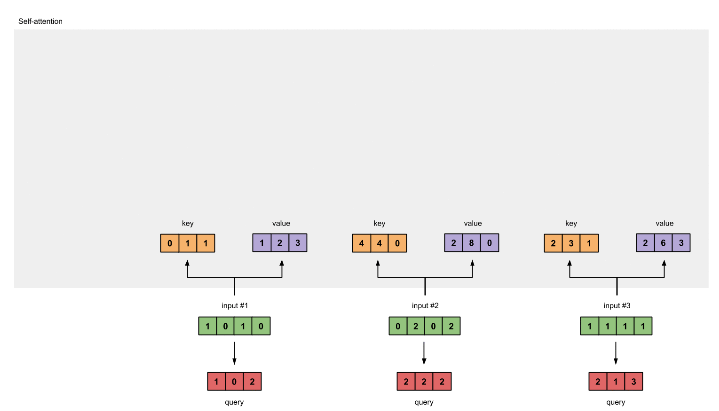
plt.ylabel("Period of Positional Encoding")

plt.savefig('wave\_result.jpg')



### 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*xt*。接着我们定义三个矩阵*WQ*,*WK*.*WV* ，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*qt*,*kt*,*vt*。我们将所有的*qt*向量拼成一个大矩阵，记作查询矩阵*Q*，将所有的*kt* 向量拼成一个大矩阵，记作键矩阵*K*，将所有的*vt*向量拼成一个大矩阵，记作值矩阵*V*（见下图）



上图中的input#1,input#2,input#3都对应xt;

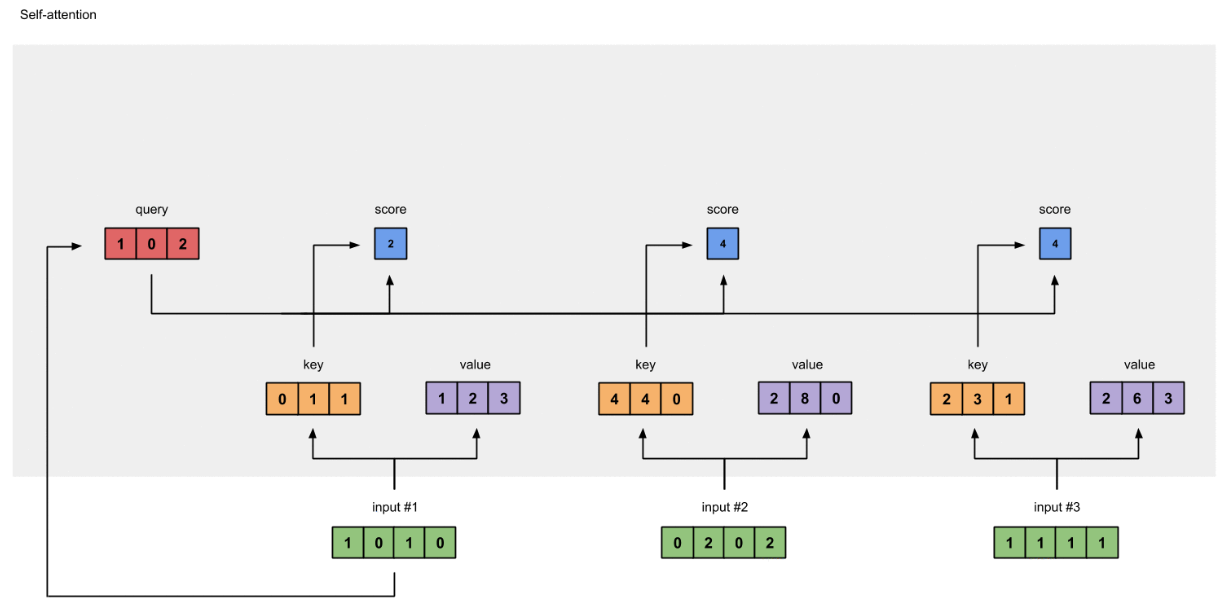
为了获得第一个字的注意力权重，我们需要用第一个字的查询向量q1乘以键矩阵K（见下图）

[0, 4, 2]

q1\*K=[1, 0, 2] x [1, 4, 3] = [2, 4, 4]

[1, 0, 1]

其中的三个列向量,,组成键矩阵K。



之后还需要将得到的值经过softmax，使得它们的和为1（见下图）

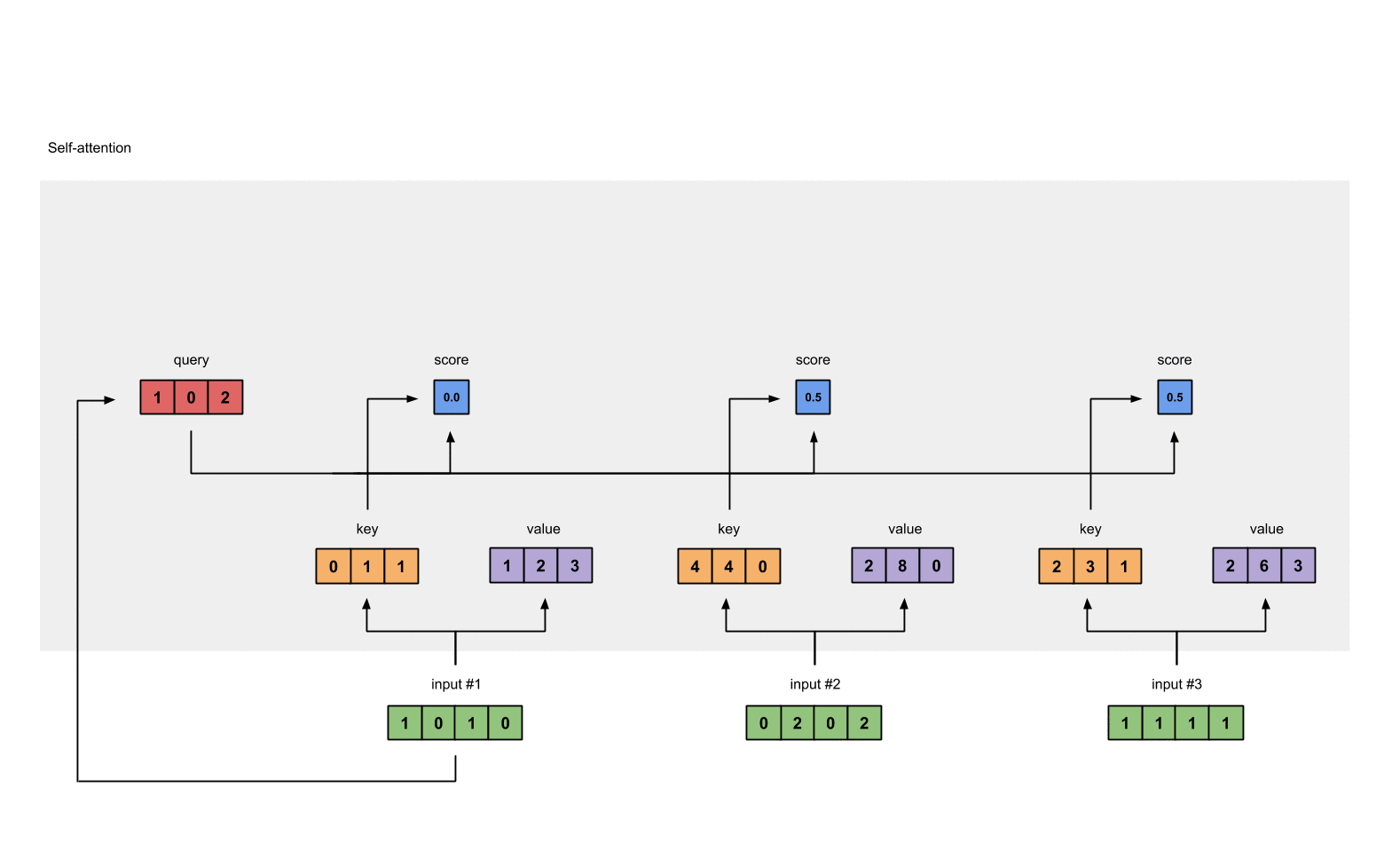
softmax([2, 4, 4]) = [0.0, 0.5, 0.5]

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）

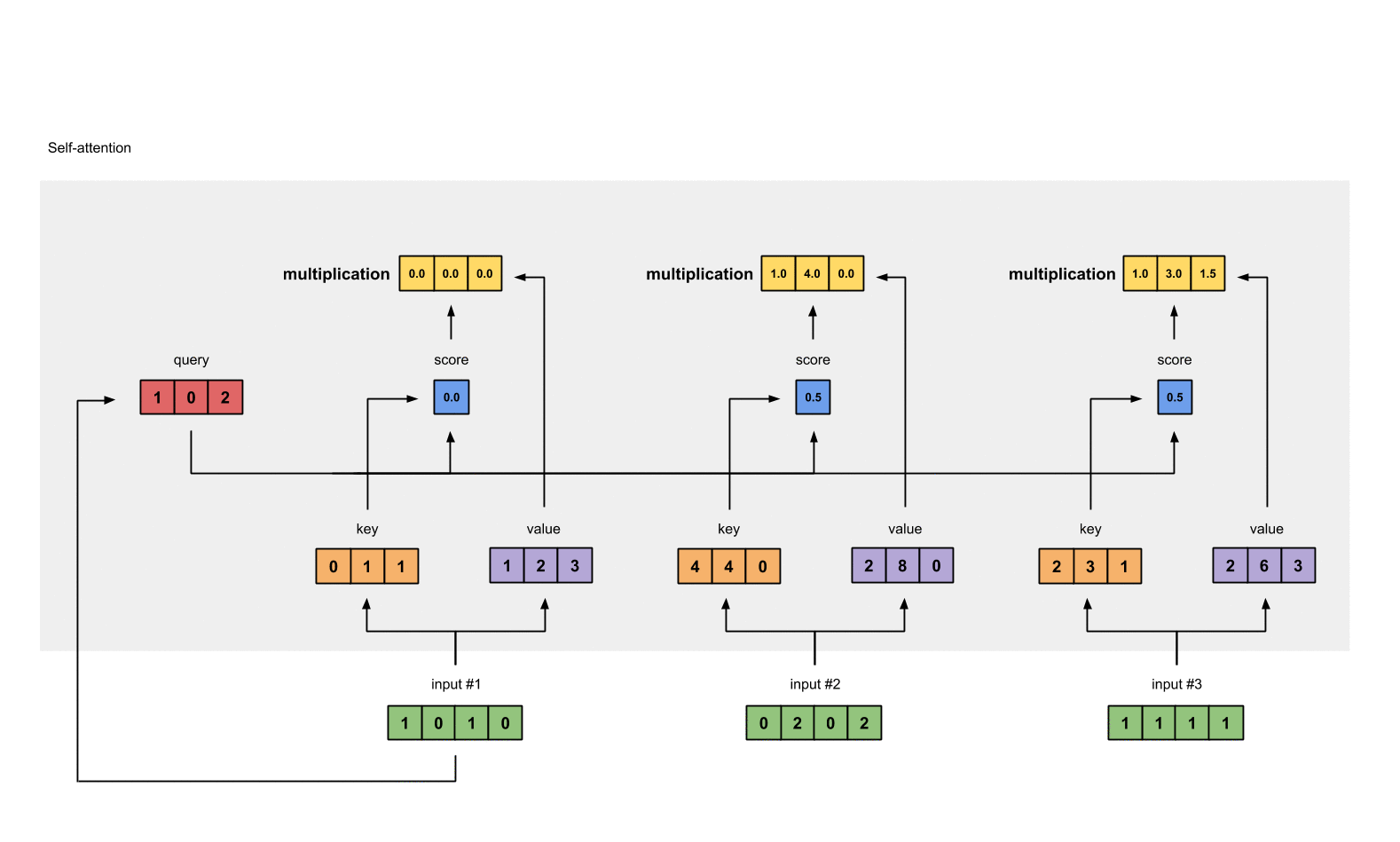
[0.0, 0.0, 0.0]

+ [1.0, 4.0, 0.0]

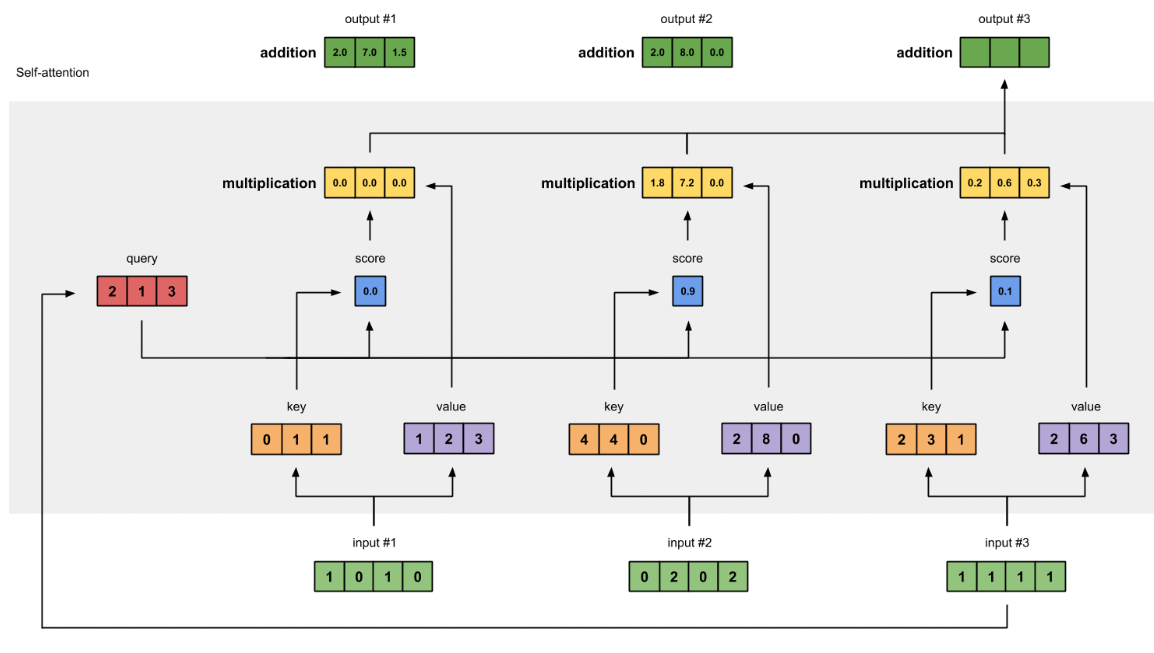
+ [1.0, 3.0, 1.5]

-----------------

= [2.0, 7.0, 1.5]



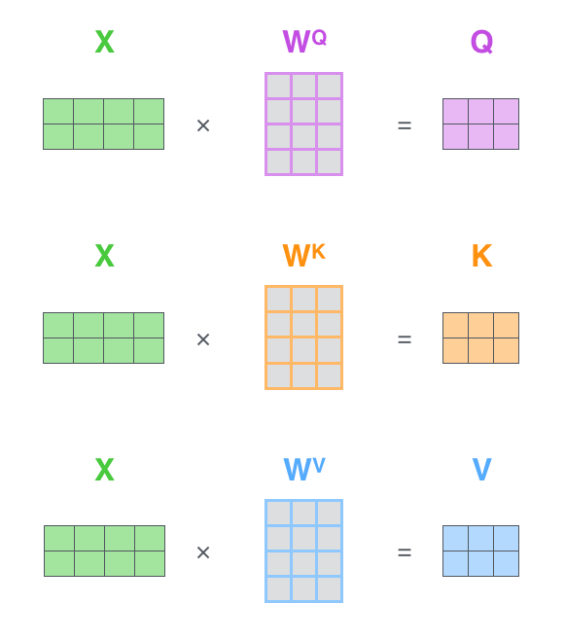
对其它的输入向量也执行相同的操作，即可得到通过self-attention后的所有输出



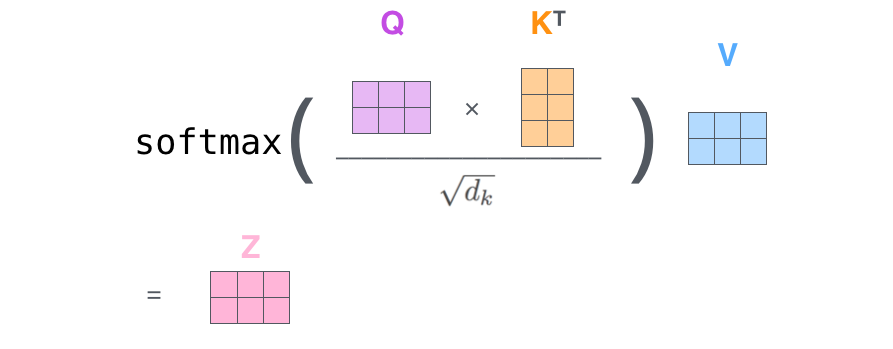
**矩阵计算**

上面介绍的方法需要一个循环遍历所有的字*xt*，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*xt*

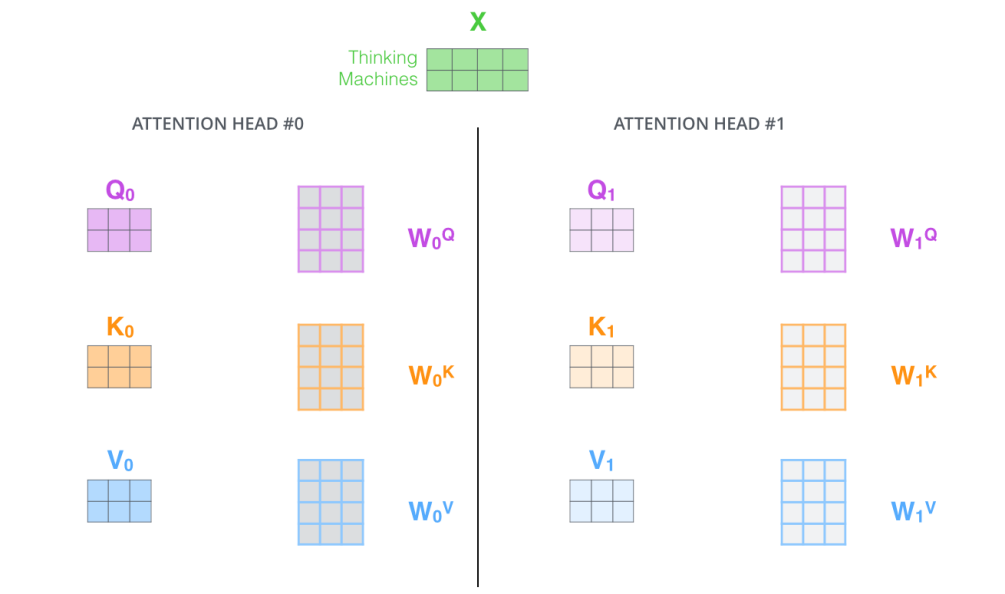


接下来将*Q* 和*KT* 相乘，然后除以*dk* （这是论文中提到的一个trick），经过softmax以后再乘以V 得到输出。此处的*dk* 对应key向量的维度。

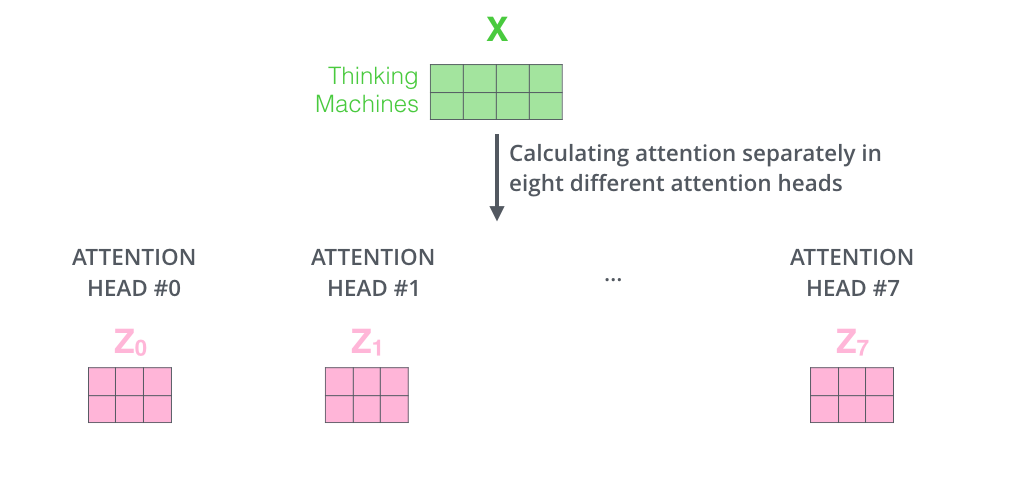


**Multi-Head Attention**

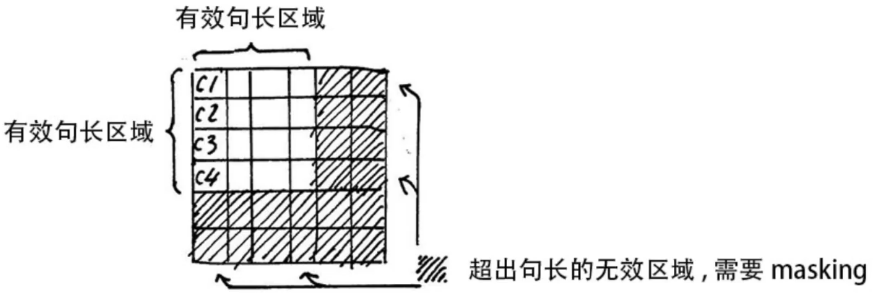
这篇论文还提出了Multi-Head Attention的概念。其实很简单，前面定义的一组Q,K,V 可以让一个词attend to相关的词，我们可以定义多组Q,K,V ，让它们分别关注不同的上下文。计算Q,K,V 的过程还是一样，只不过线性变换的矩阵从一组(WQ,WK,WV) 变成了多组(W0Q,W0K,W0V) ，(W1Q,W1K,W1V) ，…如下图所示。对于输入矩阵X ，每一组Q 、K 和V 都可以得到一个输出矩阵Z 。如下图所示

****

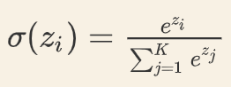
对于输入矩阵 X，每一组 Q、K 和 V 都可以得到一个输出矩阵 Z。如下图所示

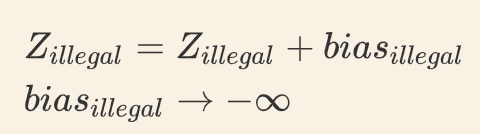
****

**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，e0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



### 3.残差连接和Layer Normalization

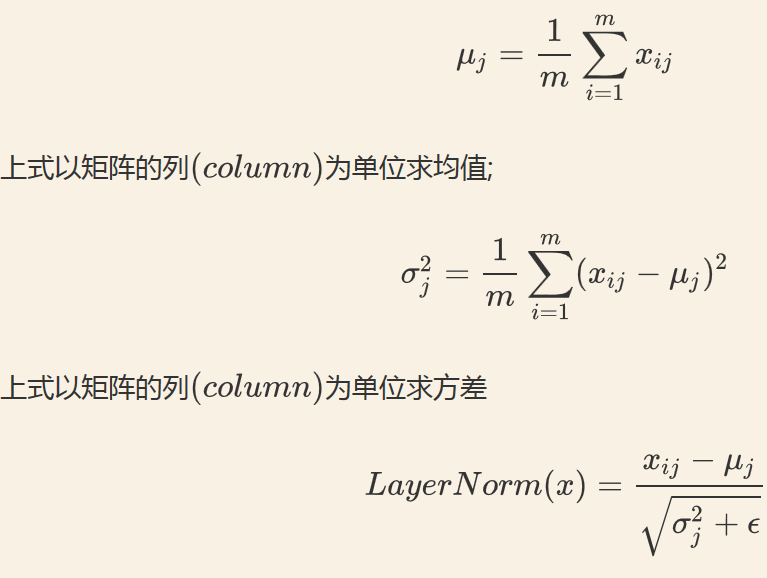
**残差连接**

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

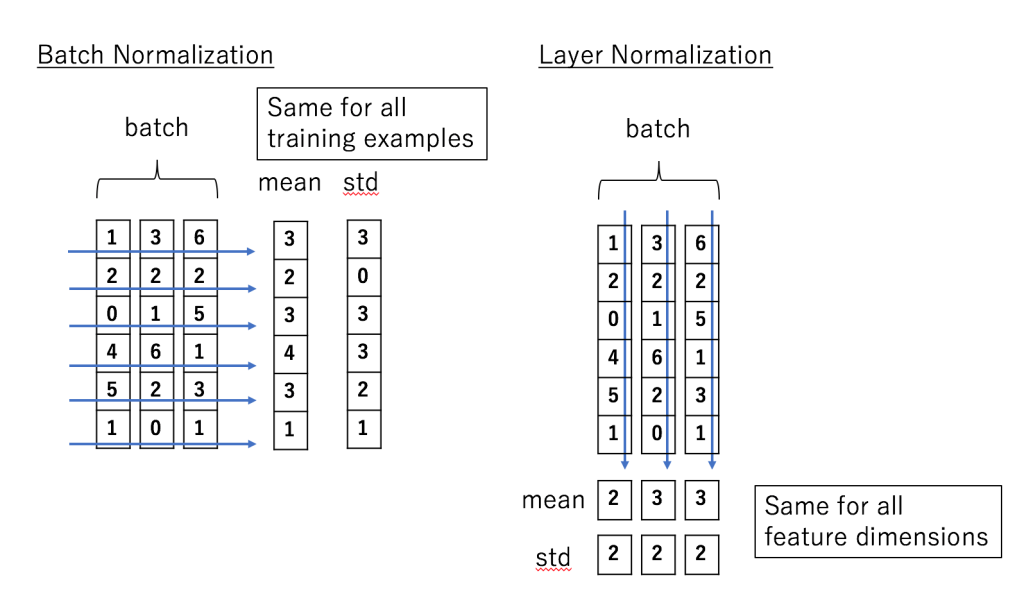
*Xembedding*+*Self Attention*(*Q*,*K*,*V*)

**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



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）。

### 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)=XWQ

K=Linear(X)=XWK

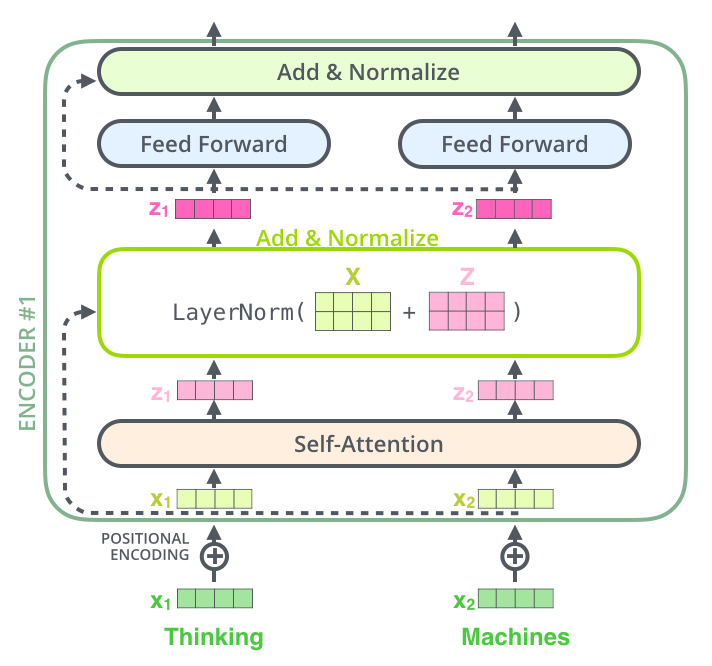
V=Linear(X)=XWV

Xattention=SelfAttention(Q, K, V)

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

Xattention=X+Xattention

Xattention=LayerNorm(Xattention)



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*=Linear(ReLU(Linear(X*attention*)))

5). FeedForward残差连接与Layer Normalization

X*hidden*=X*attention*+X*hidden*X*hidden*=LayerNorm(X*hidden*)

其中

X*hidden*∈R*batch\_size ∗ seq\_len. ∗ embed\_dim*

### 5. Transformer Decoder整体结构

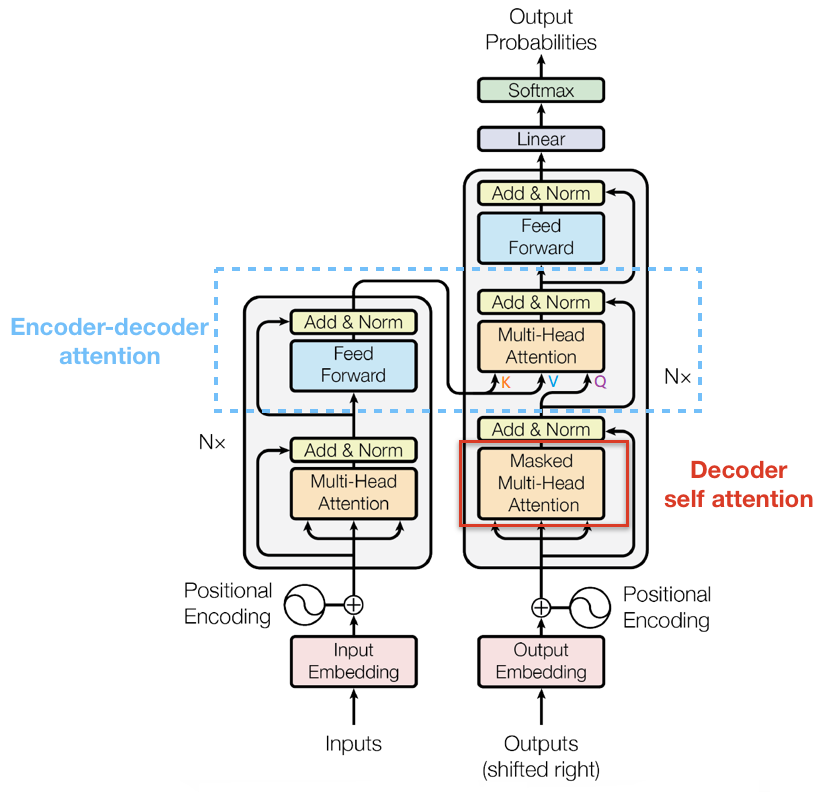
我们先从HighLevel的角度观察一下Decoder结构，从下到上依次是：

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

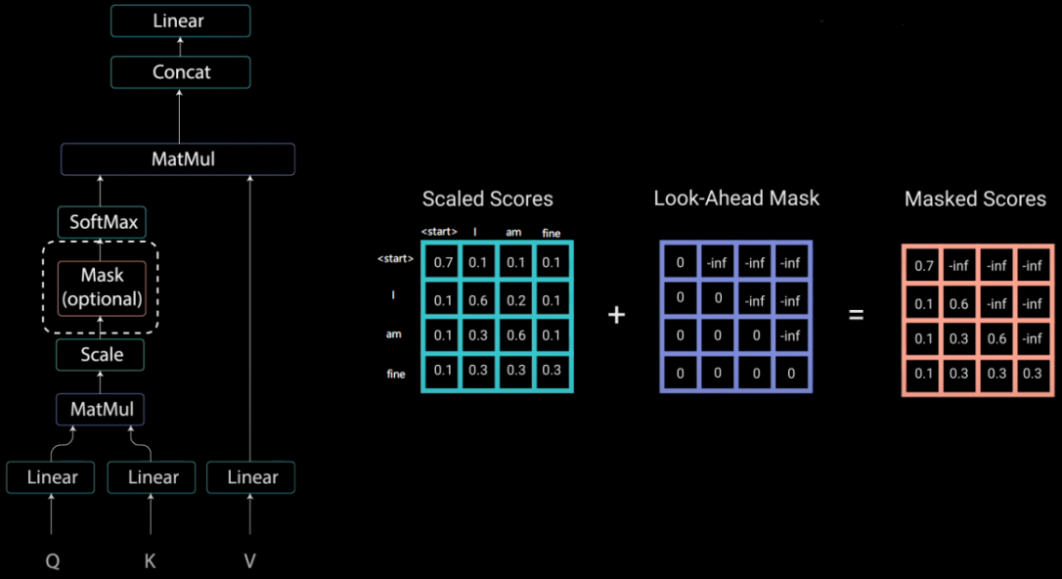
和Encoder一样，上面三个部分的每一个部分，都有一个残差连接，后接一个 Layer Normalization。Decoder的中间部件并不复杂，大部分在前面Encoder里我们已经介绍过了，但是Decoder由于其特殊的功能，因此在训练时会涉及到一些细节

**Masked Self-Attention**

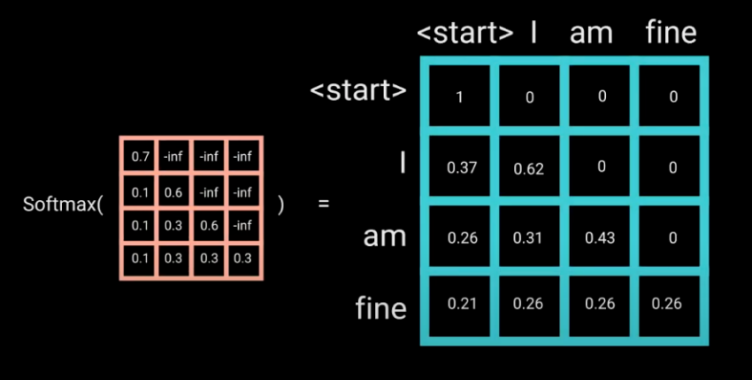
具体来说，传统Seq2Seq中Decoder使用的是RNN模型，因此在训练过程中输入t 时刻的词，模型无论如何也看不到未来时刻的词，因为循环神经网络是时间驱动的，只有当t时刻运算结束了，才能看到t+1 时刻的词。而Transformer Decoder抛弃了RNN，改为Self-Attention，由此就产生了一个问题，在训练过程中，整个ground truth都暴露在Decoder中，这显然是不对的，我们需要对Decoder的输入进行一些处理，该处理被称为Mask。



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（WQ,WK,WV ）。然后进行self-attention操作，首先通过得到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。

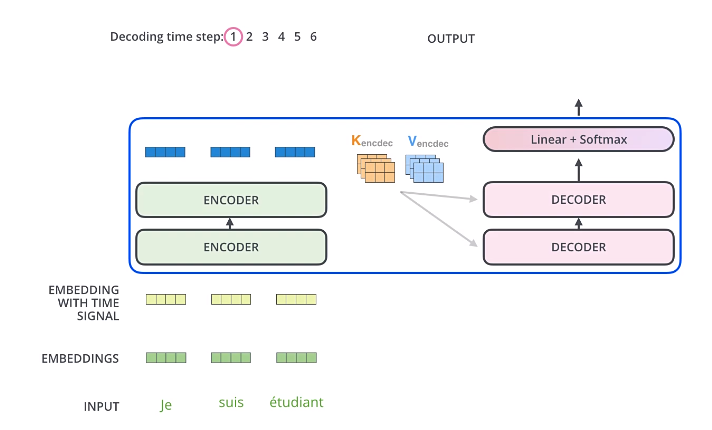
**Masked Encoder-Decoder Attention**

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

### 解码器端详细分析

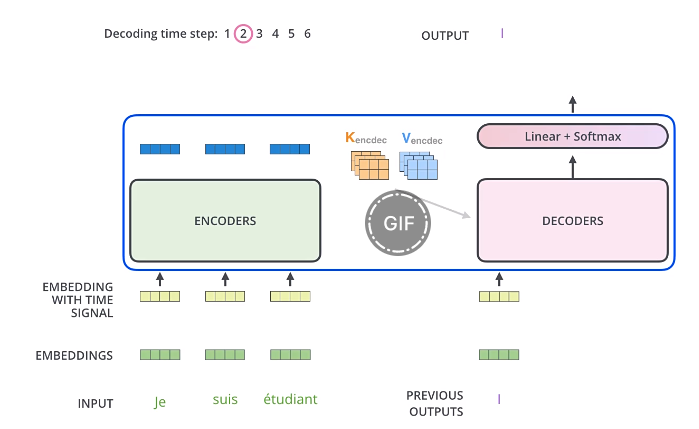
现在我们已经涵盖了编码器方面的大多数概念，我们基本上知道了解码器的组件如何工作。现在我们看一下它们是如何一起工作的。

编码器通过处理输入序列开始。顶端编码器的输出转换为注意力向量K和V的集合。这些将由每个解码器在其“编码器-解码器注意力”层中使用，这有助于解码器关注输入序列中的适当位置：



完成编码阶段，我们开始进入解码阶段。解码阶段中的每个步骤从输出序列输出一个元素。

接下来的步骤重复这个过程，直到一个特殊的符号到达表明transformer解码器完成了输出。每一步的输出在下一个时间点被喂到底部的解码器，解码器就像编码器一样往上流动冒出解码结果。就像我们对编码器输入所做的那样，我们在这些解码器输入端嵌入并增加位置编码，来指示每个单词的位置。



在解码器中自注意力层操作与在编码器中的操作方式略有不同：

在解码器中，自注意力层只被允许注意到输出序列中早一些的位置。这是通过在自注意计算中的softmax步骤之前屏蔽未来位置（将它们设置为-inf）来完成的。

“编码器--解码器注意力”层像多头自注意力一样工作，除了它从下面的层创建它的Queries矩阵，和从编码器输出中获取Keys和Value矩阵。

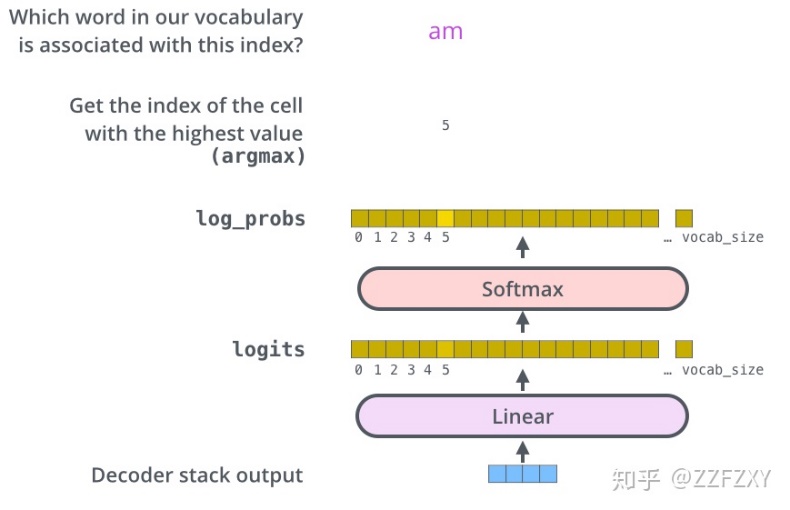
#### 线性层和softmax层

解码器输出浮点型向量。我们怎么把它转化成一个单词？这是最终线性层的工作，其后是softmax层。

线性层是一个简单的全连接神经网络，把解码器的输出向量投影到一个更大、更大的logits向量。

我们假设我们的模型知道10000个独特的单词（模型的输出词汇表），这些单词从训练集中学习得到。这将使logits向量10000个cell宽--每个cell对应于一个唯一单词的得分。这就是我们如何解释模型的输出，然后是线性层。

softmax层把这些得分转换成概率（都是正数，和为1）。概率最高的cell被选中，并且与其关联的单词将作为此时间步的输出。



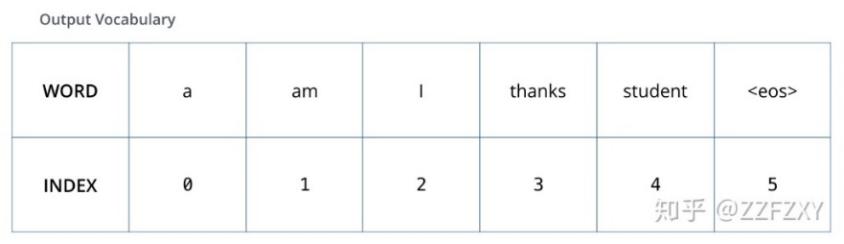
该图从底部开始，产生的矢量作为解码器的输出。然后它变成了一个输出单词。

#### 训练过程的扼要重述

既然我们通过训练过的Transformer涵盖了整个前向过程，看一下训练模型的直觉是有用的。

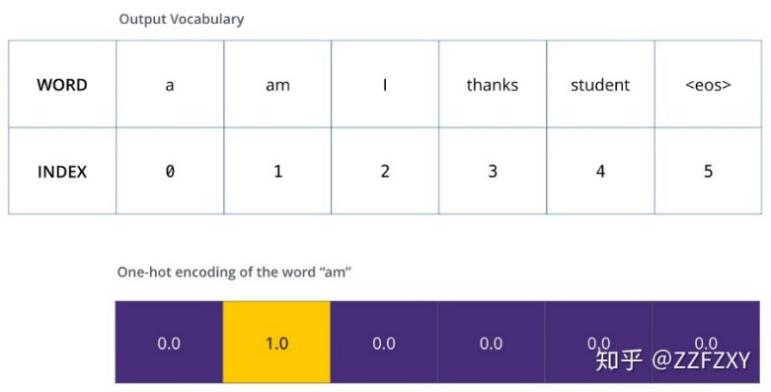
在训练过程中，一个未训练过的模型会经历同样的前向过程。但是因为我们在一个标注的训练集训练它，我们可以比较模型的输出和实际正确的输出。

为了可视化这个过程，假设我们输出词汇只包含6个单词（a,am,I,thanks,student,和end of sentence的缩写）



模型的输出词汇在预处理阶段被创建，这在我们开始训练之前。

一旦我们定义好输出词汇，我们可以用一个同样宽度的向量指示我们词汇表中的每一个单词。这也被称为one-hot编码。因此，举个例子，我们可以用下面的向量指示单词am：



上图中，最上面一行表示输出词汇表。期望输出的句子为I am a student中的词汇来自于输出词汇表中。

下面接着讨论一下模型的损失函数--在训练过程中我们优化的准则，来训练出训练有素、令人惊奇的精确模型。

#### 损失函数

假设我们正在训练我们的模型。假设我们现在处在训练阶段的第一步，我们在一个简单的例子上训练它--把“merci”翻译成“thanks”。

这意味着，我们想输出一个概率分布来指示单词“thanks”。但是因为这个模型还没有被训练，这还不太可能发生。

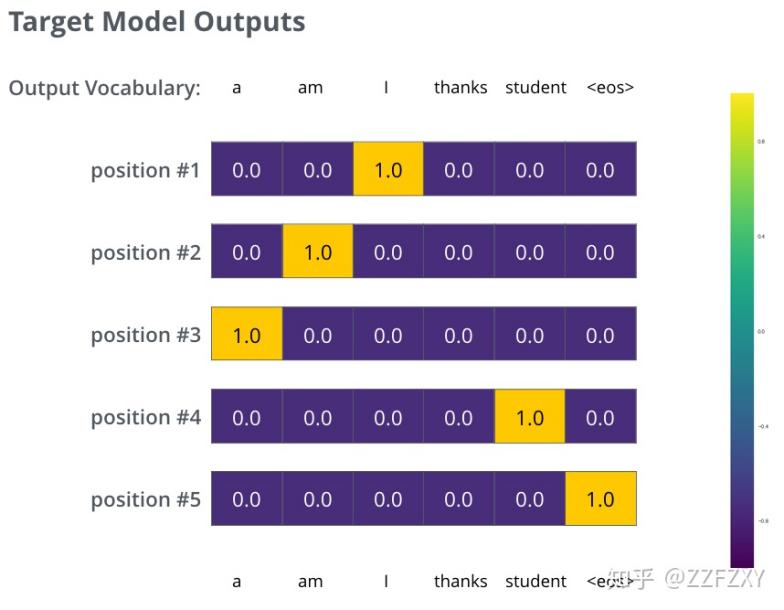


因为模型的参数(权重)随机初始化，(未训练的)模型为每一个cell/word产生一个任意值的概率分布。我们可以与真实的输出比较，然后利用反向传播法调整模型的权重，来保证输出更接近期望的输出。

我们怎么比较两个概率分布呢？我们简单的用一个分布减去另一个。更多细节请看[cross-entorpy](https://link.zhihu.com/?target=https%3A//colah.github.io/posts/2015-09-Visual-Information/" \t "_blank)和[Kullback-Leibler divergence](https://link.zhihu.com/?target=https%3A//www.countbayesie.com/blog/2017/5/9/kullback-leibler-divergence-explained" \t "_blank)

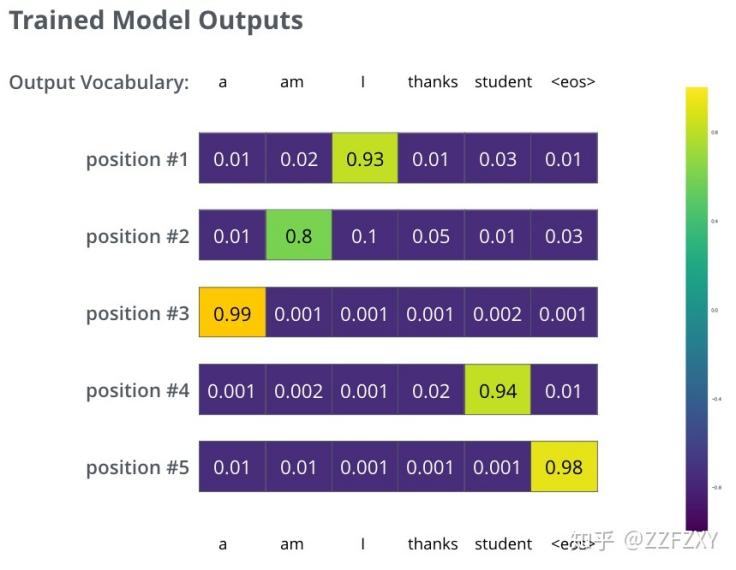
需要注意的是，这是一个过度简化的例子。更真实地，我们可以用一个长于一个单词的句子。例如-输入“je suis étudiant”期望输出是“i am a student”。这实际意味着，我们希望我们的模型能够连续输出概率分布，其中：

* 每一个概率分布用一个宽度为vocab\_size(在下面的例子中为6，真实的例子中为3000或10000)的向量来表示。
* 第一个概率分布在与单词“i”相关联的单元处具有最高概率
* 第二个概率分布在与单词“am”相关联的单元处具有最高概率
* 直到第5个输出概率指示符号，即使在10000个元素的单词表也有一个单元与它对应。



我们将在一个样本句子的训练示例中训练我们模型的目标概率分布。

在一个足够大的数据集上训练足够的时间之后，我们希望产生的概率分布会是这样：



希望经过训练，模型会输出我们期望的正确的翻译。当然，这个短语是否是训练数据集的一部分没有真正的指示(见[cross validation](https://link.zhihu.com/?target=https%3A//www.youtube.com/watch%3Fv%3DTIgfjmp-4BA" \t "_blank))。请注意，即使不太可能是该时间步的输出，每个位置都会获得一点概率--这是softmax函数非常有用的特性，这有利于训练过程。

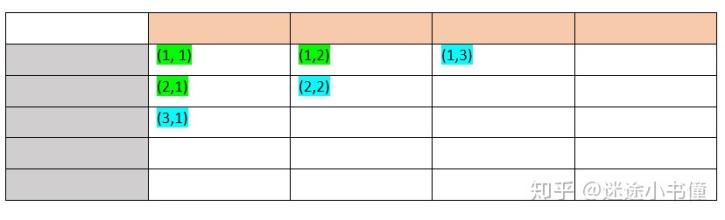
现在，因为模型一次产生一个输出，我们假设模型从那些概率分布中选择具有最大概率分布的单词，并扔掉其余的部分。这是处理的一种方式(被称为greedy decoding)。另一种处理的方式是，假设保留两个具有最高概率的单词(假设单词“I”和“a”),下一步，运行模型两次：一次假设第一个输出位置是单词“I”,另一次假设第一个输出单词是“me”，考虑位置#1和位置#2，保留产生最少误差的版本。我们在位置#2和位置#3重复同样的操作，以此类推。这种方法称为“beam search”，在我们的例子中，beam\_size是2（因为我们在计算位置#1和位置#2的beams后比较了结果），并且top\_beams也是2(因为我们保留了两个单词)。这些都是你可以试验的超参数。

#### Transformer中的beam search

<https://zhuanlan.zhihu.com/p/109183727>

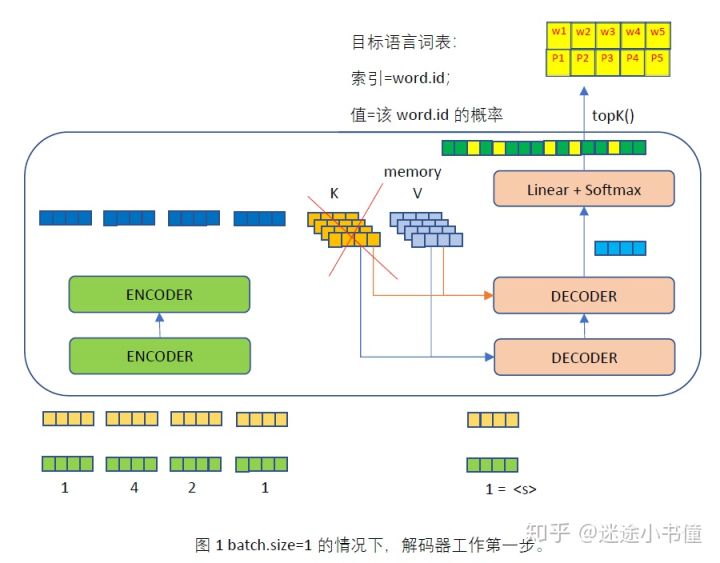
多年前，做statistical machine translation的时候，分别写过hierarchical phrase based decoder, tree-to-string decoder，string-to-tree decoder, forest-to-string decoder，以及dependency-to-dependency decoder，当时对于beam search+cube pruning搞了很多年。

因为涉及到实时计算n-gram language model来动态最优化BLEU score，所以在两个长度为n的beam合并的时候，如何把计算复杂度从n\*n降低，是个核心问题，因为一般的做法是，先n\*n，构造出来n\*n的候选，然后从中再挑取n个序列。Cube-pruning方法类似于merge sort，认定新的序列的第一个词，一般只来自前一个序列的第一个词+第二个序列的前两个词，或者反过来，这样的话，可以把n\*n的复杂度，降低到nlogn的维度。当年是Liang Huang和David Chiang提出来的这些解码思想。图示类似于：



两个序列的合并之后，第一个词，来自于三个位置(1,1), (1,2), (2,1)，然后继续扩展这个frontier set，直到到一个新的beam。（横向是一个sequence，纵向是另外一个sequence）

曾经为transformer中该如何执行beam search而思考思索，这里统一总结一下。



如图1 所示，走一遍DECODERs和Linear+softmax生成器，然后得到一个预测出来的logits向量（索引为word.id，值为该word.id的概率）。一次选择出来beam.size=5个words和它们对应的概率。如图中w1~w5和p1~p5所示。

有一点特别需要注意的是，来自Encoder的memory，是只有一个(batch.size, source.seq.len, d\_model)的tensor，不会在Encoder阶段分成K和V给Decoder部分！来自Encoder的memory tensor是经过Decoder里面每一层的MultiHeadAttention中的Linear network（可训练）来进一步转化成K和V的！

上面图中：假设当前是一个句子的输入，则memory的部分，是(1, 4, 4)，第一个1是batch.size，之后的4是序列的长度（单词的个数），最后一个4是d\_model，编码维度。

目标语言序列端，第一个输入是<s>，向量表示为(1, 1, 4)，也是分别代表了batch.size, sequence length和编码维度。

从多层Decoder出来之后，形状为(1, 1, 4)，而后经过Linear+softmax得到的是(1,1,target.vocab.size)，最后一个维度是目标语言词表大小。对于最后一个维度，索引是word.id，值就是该word.id的概率，从而我们可以取概率最大的前五个word.id，以及它们对应的概率值，来构造出来，W1~W5和P1~P5这些值供下一步解码使用。

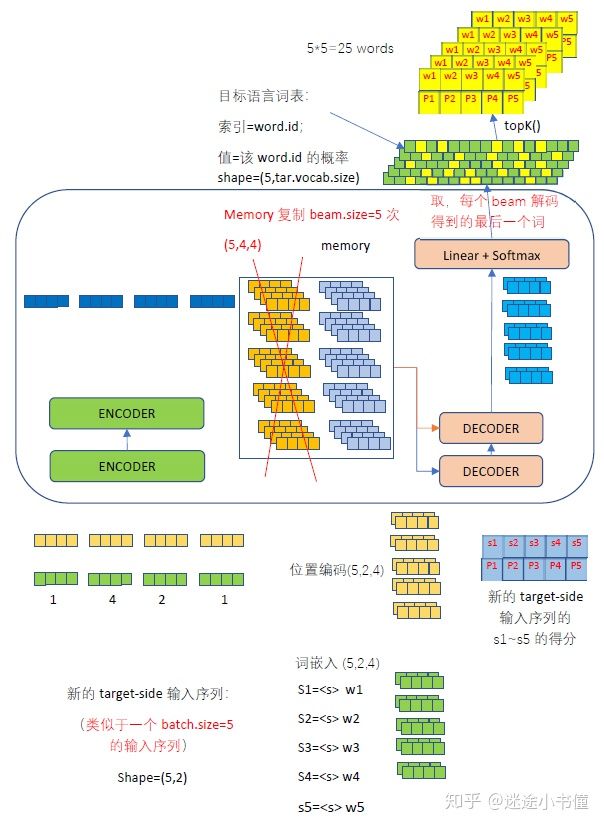


图2 batch.size=1, beam.size=5的时候的解码器工作第二步

如图2所示。根据beam size=5，构造出来一个batch.size=5的tensor，如上图所示。DECODERs和Linear+softmax生成器，然后得到一个预测出来的logits向量（索引为word.id，值为该word.id的概率）。一次选择出来beam.size=5个words和它们对应的概率。如图中右上角5\*5矩阵（黄色）所示。

特别需要注意的有：

1. 目前Decoder的输入的target-side序列，是(5,2)的一个矩阵，5代表beam.size，2代表序列长度；

2. 之后，先进过目标语言的词嵌入，得到一个(5,2,4)的tensor张量，再扔给位置编码，得到的也是一个(5,2,4)的张量。

3. 该(5,2,4)的张量（相当于Q）扔给Decoder之后，得到的是(5,2,4)的张量。这里特别需要注意的是，需要对来自Encoder的memory进行复制扩展。本来，Encoder的memory的shape是(1,4,4)，现在为了对照beam size=5，把(1,4,4)复制成(5,4,4)，相当于原来的一个4\*4矩阵，原样复制成了5个4\*4矩阵。这个memory张量(5,4,4)会在Decoder的MultiHeadAttention里面，进一步被两个Linear network（可训练的）变化成K和V。

4. （5,2,4）的张量扔给Linear+Softmax，之后得到的是(5,2,target.vocab.size)，最后一个维度扩展为目标语言词表的大小；

5. 取上面这个张量的每个beam的最后一个词，即(5, -1, target.vocab.size)，得到的是上图右上角的（绿+黄相间）的矩阵。然后按照最后一个维度，取top-5，就得到了右上角的最后的矩阵，是5\*5的，表示每个beam下扩展出来了5个最佳候选。

接下来，我们得到了5\*5=25个词的候选，五行代表五个beam，每一行五列，代表该beam下的下一个词的五个可能的候选。（简单期间，下图省略了第一个词之前的<s>）

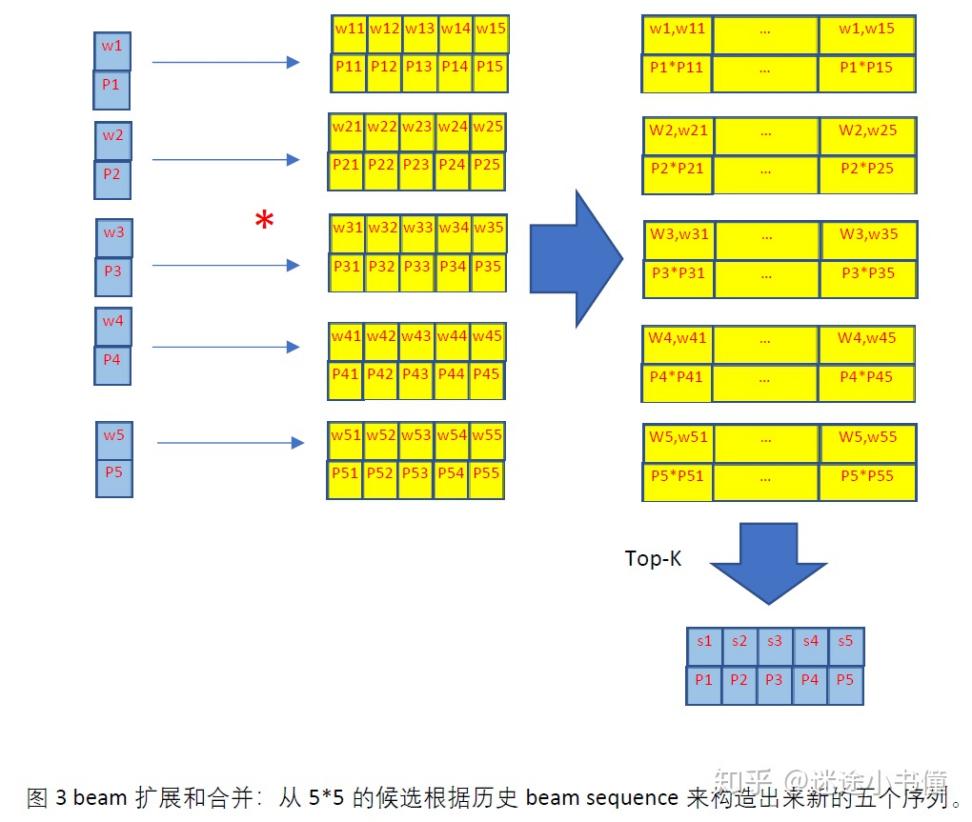


图3 beam扩展和合并：从5\*5的候选根据历史beam sequence来构造出来新的五个序列

图3 beam扩展和合并：从5\*5的候选根据历史beam sequence来构造出来新的五个序列。注意：因为矩阵相加可以并行进行，所以这里的速度会非常快，之前的那种cube-pruning方法不再是提速度的时候所必须的了。

这样，我们得到的是5\*5=25个候选序列。如果我们从中再次挑选概率最大的top-5个序列，以及他们的概率，可以进入下一轮的循环。如上图3所示，我们使用s1到s5来表示新得到的序列，每个si应该包括<s> <word in position 1> <word in position 2>，这样的三个词。实际解码的时候，一般使用log(p)，所以上面可以修改为log(p1) + log(p11)的形式。

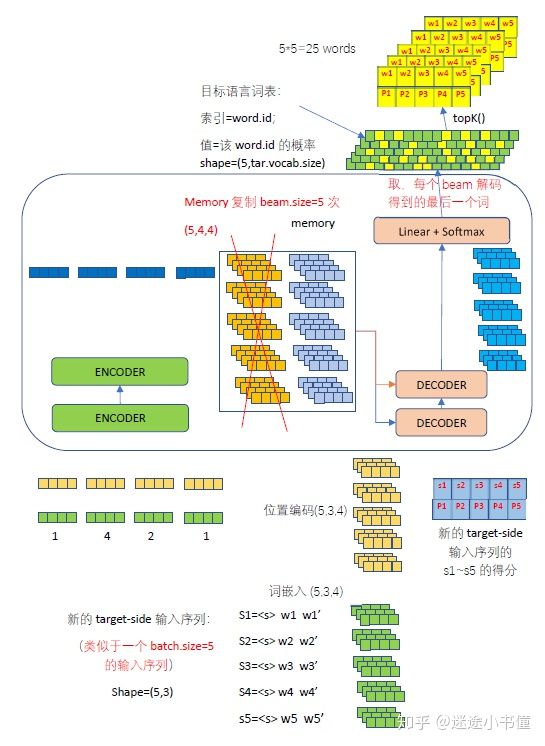


图4 batch.size=1, beam.size=5的前提下，解码第三个词的步骤展示

特别需要注意的有（和前面的类似，重复一遍，加深印象）：

1. 目前Decoder的输入的target-side序列，是(5,3)的一个矩阵，5代表beam.size，3代表序列长度；

2. 之后，先进过目标语言的词嵌入，得到一个(5,3,4)的tensor张量，再扔给位置编码，得到的也是一个(5,3,4)的张量。

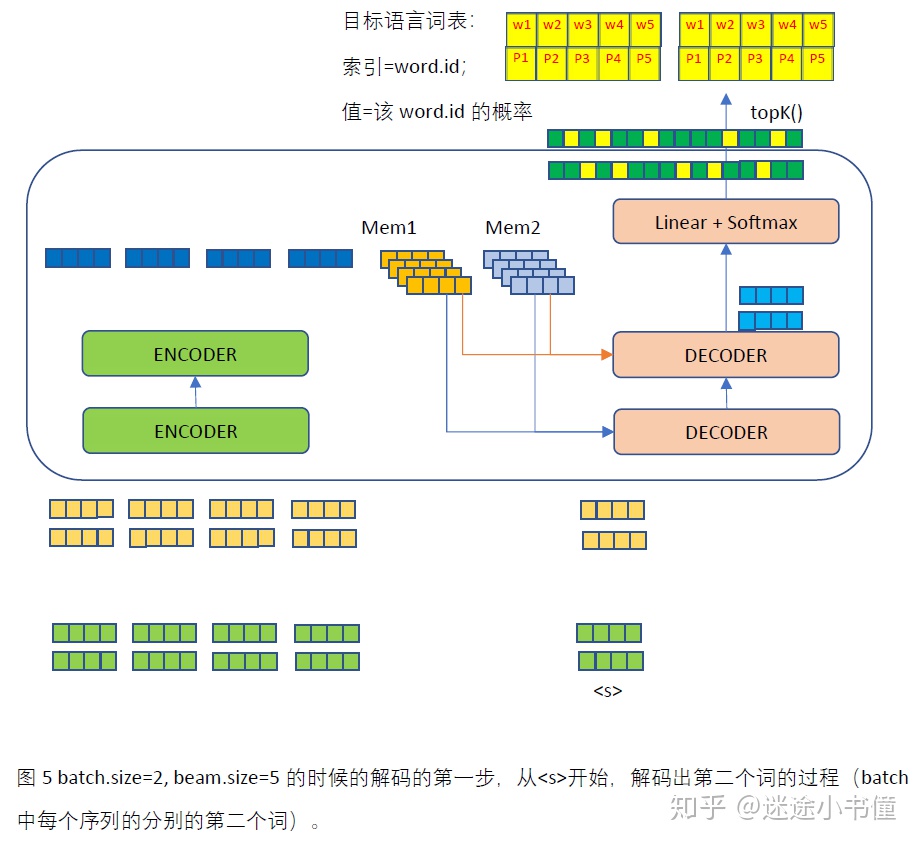
3. 该(5,3,4)的张量（相当于Q）扔给Decoder之后，得到的是(5,3,4)的张量。这里特别需要注意的是，需要对来自Encoder的memory进行复制扩展。本来，Encoder的memory的shape是(1,4,4)，现在为了对照beam size=5，把(1,4,4)复制成(5,4,4)，相当于原来的一个4\*4矩阵，原样复制成了5个4\*4矩阵。这个memory张量(5,4,4)会在Decoder的MultiHeadAttention里面，进一步被两个Linear network（可训练的）变化成K和V。

4. （5,3,4）的张量扔给Linear+Softmax，之后得到的是(5,3,target.vocab.size)，最后一个维度扩展为目标语言词表的大小；

5. 取上面这个张量的每个beam的最后一个词，即(5, -1, target.vocab.size)，得到的是上图右上角的（绿+黄相间）的矩阵。然后按照最后一个维度，取top-5，就得到了右上角的最后的矩阵，是5\*5的，表示每个beam下扩展出来了5个最佳候选。

之后，简单说一下beam search的终止条件。其一，设定一个最长目标语言长度，当到达这个长度之后，统一结束。其二，判断当前beam是否已经包含了</s>，如果是，则该beam被从当前beam tensor中剪切出来，可以不参与下一步的计算。为了惩罚特别短或者特别长的生成序列，可以设置一些长度惩罚方法，细节不再冗述。

另外，上面的例子是一次翻译一个句子，当然可以一次翻译一个batch的句子。算法流程大概如下：假设batch.size=2，（同样适用于batch.size更大的情况，这里简单期间，batch.size=2）。Encoder端输入是(2, 4,4)的张量，经过Encoder的多层编码之后，得到的是(2,4,4)的source memory。然后，目标语言序列一端，最初的输入是(2, 1)的矩阵，2=batch.size, 1个词是<s>。然后是词嵌入和位置编码，得到的是(2, 1, 4)张量。经历一次Decoder之后，得到的是(2, 1, 4)的张量，然后扔给Linear+softmax，得到的是(2, 1, target.vocab.size)，可以对最后一个维度取top-5，得到的是两个w1~w5和P1~P5的列表，分别对应batch中两个序列的下一个单词。



Batch.size=2的时候的，解码第一步，如图5所示。

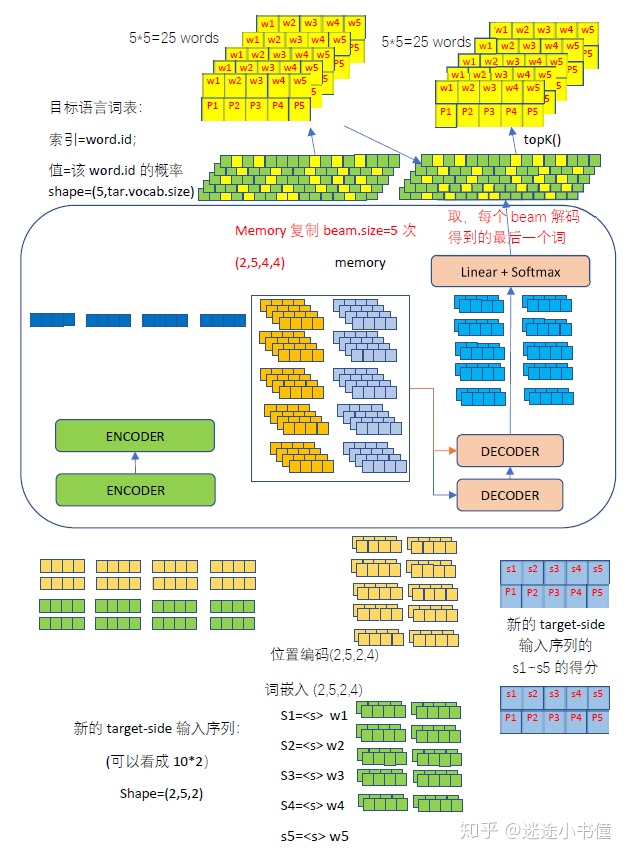


图6 batch.size=2, beam.size=5的时候，解码出第三个词的过程（每个batch的每个序列的，5个beam的第三个词）。

输入的memory，从(2,4,4)复制为(2, 5, 4, 4)，在实际使用的时候，reshape为(10, 4, 4)。

Decoder的输入部分，最初为(2,5,2)，batch为2，beam为5，每个beam为2个单词。经过词嵌入之后，得到的张量是(2,5,2,4)，然后在给Decoder之前，可以将其reshape为（10,2,4）

从而得到的Decoder的输出的张量是(10, 2, 4)的，再reshape回来，得到(2,5,2,4)，扔给Linear+softmax之后，得到的是(2, 5, 2, target.vocab.size)。

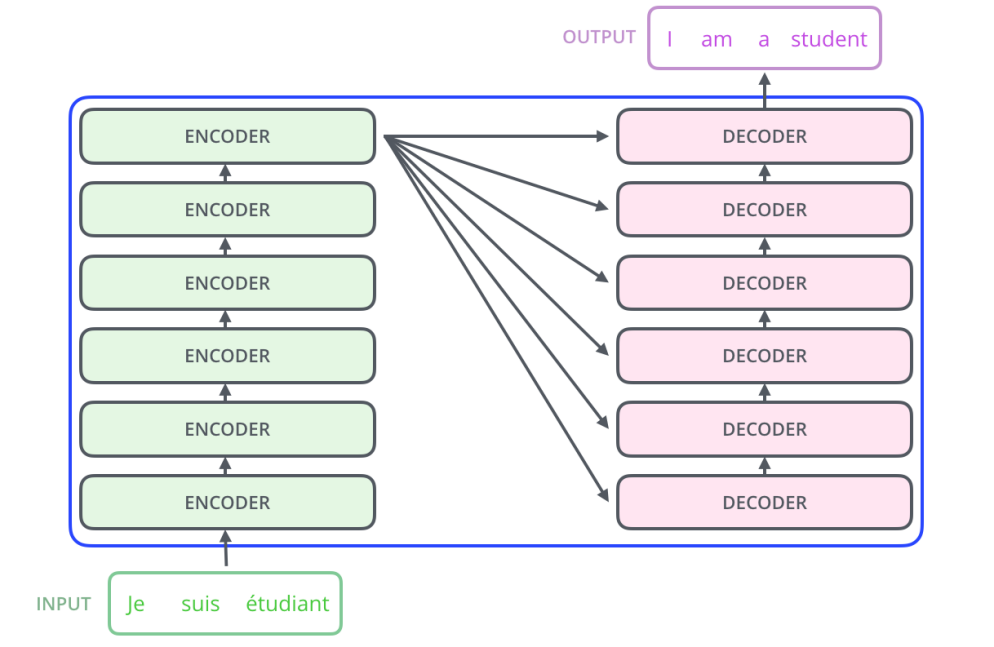
之后就是上图的最上边两行图示了，先取最后一个单词，也就是(2, 5, -1, target.vocab.size)，然后取最后一个维度的前top-5概率下的单词。会分别得到两个5\*5的w1~w5和p1~p5的列表。

各自分别和当前序列的s1~s5和p1~p5结合(logp + logp)的方式，就得到新的五个序列。持续按照这个步骤执行即可。

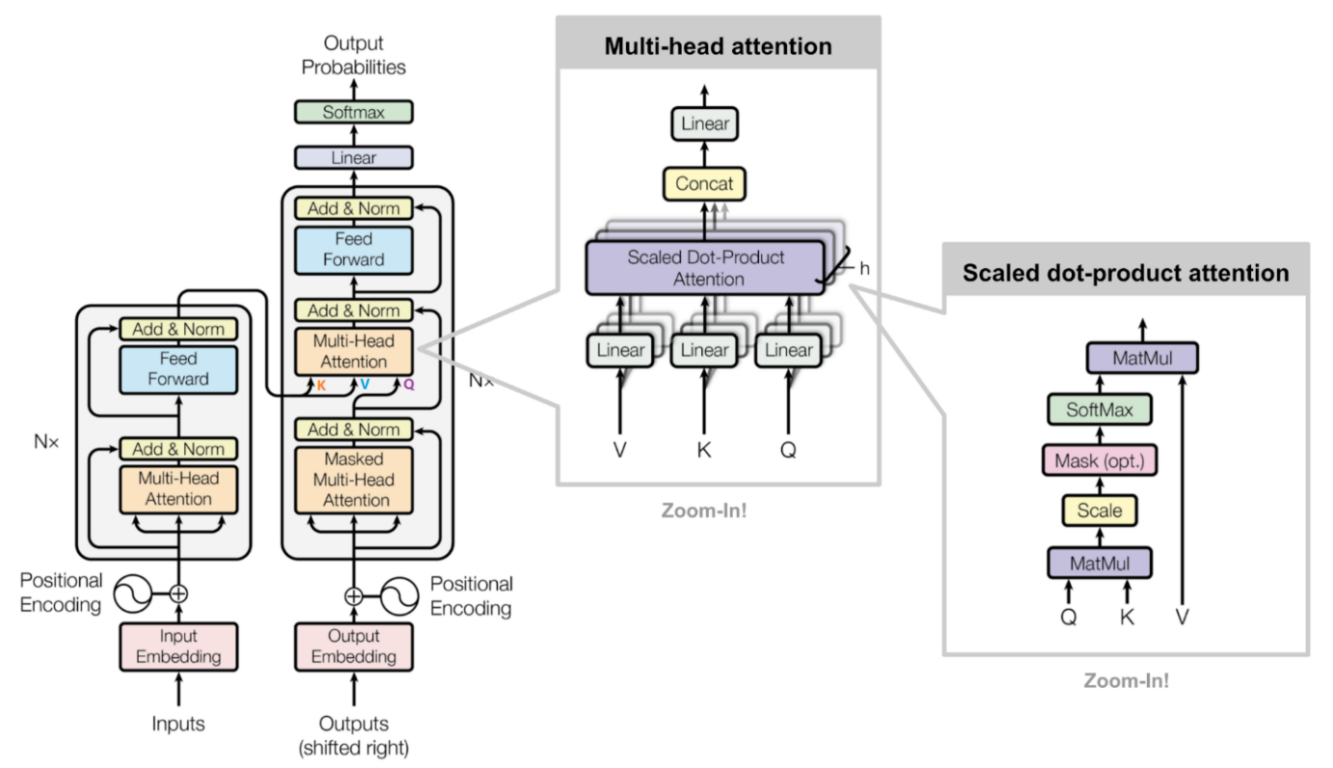
至此，Transformer的基本Encoder, Decoder, Generator, PositionEncoding, MultiHeadAttention等模块就介绍完毕了。训练，loss, optimization, 以及beam search解码也介绍完毕。

### Transformer整体框架分析

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。



实现上图的细节部分架构如下图所示：



Transformer的Encoder负责把输入(语言序列)隐射成隐藏层，然后Decoder再把隐藏层映射为自然语言序列。两者通过“编码器--解码器注意力”层进行数据传输。“编码器--解码器注意力”层像多头自注意力一样工作，它从decoder中下面的层创建它的Queries矩阵，从编码器输出中获取Keys和Value矩阵。这样，相当于根据每个不同的预测输出单词所生成的Queries，与编码器中的源单词的Key进行相乘，所产生的多个注意力系数与编码器对应源单词的Value值进行相乘后相加，产生与解码器中预测输出单词对应的编码器的语义编码输出。“编码器--解码器注意力”层实现的是一个普通的注意力机制。而编码器中都有自己的自注意力模块，用于理解输入源语句内部不同单词相互之间的注意力系数，提升编码器和内部自身的语义编码的效果。对于解码器而言，也有类似的功能模块实现类似的效果。

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。在解码器中，自注意力层只被允许注意到输出序列中早一些的位置。这是通过在自注意计算中的softmax步骤之前屏蔽未来位置（将它们设置为-inf）来完成的。其原因参见2.6部分(Transformer Decoder整体结构)中的Masked Self-Attention部分的说明。

在上图中，第一级encoder和decoder中的PositionalEncoding模块所生成的V、K、Q向量都相同。从encoder传至decoder的向量的只有一组隐藏层矩阵数据，用于作为decoder中多头注意力机制中自注意力的K和V值的输入，此处的K和V值相同。

### Dropout函数

# -\*- coding: utf-8 -\*-

import torch

import torch.nn as nn

m = nn.Dropout(p=0.5)

input = torch.randn(3,4) ## 对dim=1维进行随机置为0

print("----input=",input)

print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*')

print(m(input))

运行结果

# ----input= tensor([[-1.2699, -0.4151, -0.0250, 0.6706],

# [-0.8602, -0.3765, -1.8860, 0.9053],

# [ 0.4574, 0.4842, 0.4353, 0.8597]])

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# tensor([[-1.5874, -0.5189, -0.0000, 0.0000],

# [-1.0752, -0.4707, -2.3575, 1.1316],

# [ 0.5717, 0.6052, 0.5441, 1.0746]])

Dropout是指在深度学习网络的训练过程中，对于神经网络单元，按照一定的概率p将其暂时从网络中丢弃。注意是暂时，对于随机梯度下降来说，由于是随机丢弃，故而每一个mini-batch都在训练不同的网络。对于没有被丢弃掉的元素来说，其值对应整体值的(1-p)比例，所以没有被丢弃掉的每个元素，都要除以(1-p)。

### 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。

### 参考文章

[Transformer](http://mantchs.com/2019/09/26/NLP/Transformer/" \t "_blank)

[The Illustrated Transformer](http://jalammar.github.io/illustrated-transformer/" \t "_blank)

[TRANSFORMERS FROM SCRATCH](http://www.peterbloem.nl/blog/transformers" \t "_blank)

[Seq2seq pay Attention to Self Attention: Part 2](https://medium.com/@bgg/seq2seq-pay-attention-to-self-attention-part-2-中文版-ef2ddf8597a4" \t "_blank)

## 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()])

## 注意力机制深度分析

### Encoder-Decoder框架

要了解深度学习中的注意力模型，就不得不先谈Encoder-Decoder框架，因为目前大多数注意力模型附着在Encoder-Decoder框架下，当然，其实注意力模型可以看作一种通用的思想，本身并不依赖于特定框架，这点需要注意。

Encoder-Decoder框架可以看作是一种深度学习领域的研究模式，应用场景异常广泛。图2是文本处理领域里常用的Encoder-Decoder框架最抽象的一种表示。

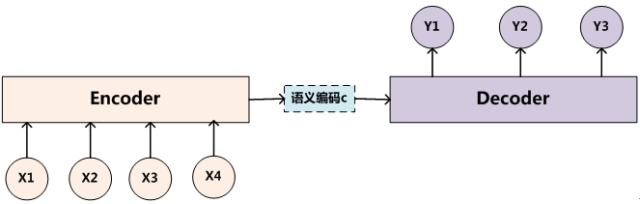


图2 抽象的文本处理领域的Encoder-Decoder框架

文本处理领域的Encoder-Decoder框架可以这么直观地去理解：可以把它看作适合处理由一个句子（或篇章）生成另外一个句子（或篇章）的通用处理模型。对于句子对<Source,Target>，我们的目标是给定输入句子Source，期待通过Encoder-Decoder框架来生成目标句子Target。Source和Target可以是同一种语言，也可以是两种不同的语言。而Source和Target分别由各自的单词序列构成：

IMG_257

Encoder顾名思义就是对输入句子Source进行编码，将输入句子通过非线性变换转化为中间语义表示C：

IMG_258

对于解码器Decoder来说，其任务是根据句子Source的中间语义表示C和之前已经生成的历史信息IMG_259来生成i时刻要生成的单词IMG_260：

IMG_261

每个yi都依次这么产生，那么看起来就是整个系统根据输入句子Source生成了目标句子Target。如果Source是中文句子，Target是英文句子，那么这就是解决机器翻译问题的Encoder-Decoder框架；如果Source是一篇文章，Target是概括性的几句描述语句，那么这是文本摘要的Encoder-Decoder框架；如果Source是一句问句，Target是一句回答，那么这是问答系统或者对话机器人的Encoder-Decoder框架。由此可见，在文本处理领域，Encoder-Decoder的应用领域相当广泛。

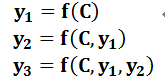
Encoder-Decoder框架不仅仅在文本领域广泛使用，在语音识别、图像处理等领域也经常使用。比如对于语音识别来说，图2所示的框架完全适用，区别无非是Encoder部分的输入是语音流，输出是对应的文本信息；而对于“图像描述”任务来说，Encoder部分的输入是一副图片，Decoder的输出是能够描述图片语义内容的一句描述语。一般而言，文本处理和语音识别的Encoder部分通常采用RNN模型，图像处理的Encoder一般采用CNN模型。

### Attention模型

本节先以机器翻译作为例子讲解最常见的Soft Attention模型的基本原理，之后抛离Encoder-Decoder框架抽象出了注意力机制的本质思想，然后简单介绍最近广为使用的Self Attention的基本思路。

1. Soft Attention模型

图2中展示的Encoder-Decoder框架是没有体现出“注意力模型”的，所以可以把它看作是注意力不集中的分心模型。为什么说它注意力不集中呢？请观察下目标句子Target中每个单词的生成过程如下：



其中f是Decoder的非线性变换函数。从这里可以看出，在生成目标句子的单词时，不论生成哪个单词，它们使用的输入句子Source的语义编码C都是一样的，没有任何区别。

而语义编码C是由句子Source的每个单词经过Encoder编码产生的，这意味着不论是生成哪个单词，y1,y2还是y3，其实句子Source中任意单词对生成某个目标单词yi来说影响力都是相同的，这是为何说这个模型没有体现出注意力的缘由。这类似于人类看到眼前的画面，但是眼中却没有注意焦点一样。

如果拿机器翻译来解释这个分心模型的Encoder-Decoder框架更好理解，比如输入的是英文句子：Tom chase Jerry，Encoder-Decoder框架逐步生成中文单词：“汤姆”，“追逐”，“杰瑞”。

在翻译“杰瑞”这个中文单词的时候，分心模型里面的每个英文单词对于翻译目标单词“杰瑞”贡献是相同的，很明显这里不太合理，显然“Jerry”对于翻译成“杰瑞”更重要，但是分心模型是无法体现这一点的，这就是为何说它没有引入注意力的原因。

没有引入注意力的模型在输入句子比较短的时候问题不大，但是如果输入句子比较长，此时所有语义完全通过一个中间语义向量来表示，单词自身的信息已经消失，可想而知会丢失很多细节信息，这也是为何要引入注意力模型的重要原因。

上面的例子中，如果引入Attention模型的话，应该在翻译“杰瑞”的时候，体现出英文单词对于翻译当前中文单词不同的影响程度，比如给出类似下面一个概率分布值：

（Tom,0.3）(Chase,0.2) (Jerry,0.5)

每个英文单词的概率代表了翻译当前单词“杰瑞”时，注意力分配模型分配给不同英文单词的注意力大小。这对于正确翻译目标语单词肯定是有帮助的，因为引入了新的信息。

同理，目标句子中的每个单词都应该学会其对应的源语句子中单词的注意力分配概率信息。这意味着在生成每个单词yi的时候，原先都是相同的中间语义表示C会被替换成根据当前生成单词而不断变化的Ci。理解Attention模型的关键就是这里，即由固定的中间语义表示C换成了根据当前输出单词来调整成加入注意力模型的变化的Ci。增加了注意力模型的Encoder-Decoder框架理解起来如图3所示。

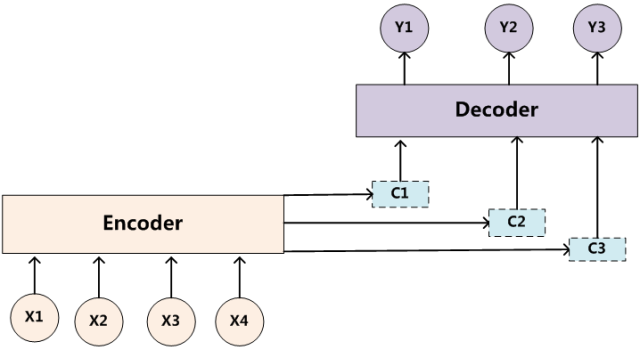
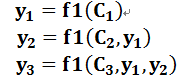
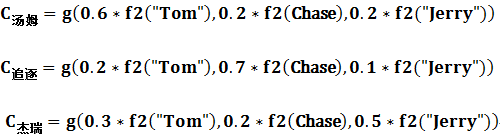


图3 引入注意力模型的Encoder-Decoder框架

即生成目标句子单词的过程成了下面的形式：



而每个Ci可能对应着不同的源语句子单词的注意力分配概率分布，比如对于上面的英汉翻译来说，其对应的信息可能如下：



其中，f2函数代表Encoder对输入英文单词的某种变换函数，比如如果Encoder是用的RNN模型的话，这个f2函数的结果往往是某个时刻输入xi后隐层节点的状态值；g代表Encoder根据单词的中间表示合成整个句子中间语义表示的变换函数，一般的做法中，g函数就是对构成元素加权求和，即下列公式：

IMG_266

其中，Lx代表输入句子Source的长度，aij代表在Target输出第i个单词时Source输入句子中第j个单词的注意力分配系数，而hj则是Source输入句子中第j个单词的语义编码。假设下标i就是上面例子所说的“汤姆”，那么Lx就是3，h1=f(“Tom”)，h2=f(“Chase”), h3=f(“Jerry”)分别是输入句子每个单词的语义编码，对应的注意力模型权值则分别是0.6,0.2,0.2，所以g函数本质上就是个加权求和函数。如果形象表示的话，翻译中文单词“汤姆”的时候，数学公式对应的中间语义表示Ci的形成过程类似图4。

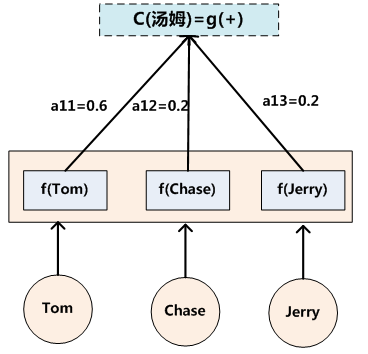


图4 Attention的形成过程

这里还有一个问题：生成目标句子某个单词，比如“汤姆”的时候，如何知道Attention模型所需要的输入句子单词注意力分配概率分布值呢？就是说“汤姆”对应的输入句子Source中各个单词的概率分布：(Tom,0.6)(Chase,0.2)(Jerry,0.2) 是如何得到的呢？

为了便于说明，我们假设对图2的非Attention模型的Encoder-Decoder框架进行细化，Encoder采用RNN模型，Decoder也采用RNN模型，这是比较常见的一种模型配置，则图2的框架转换为图5。

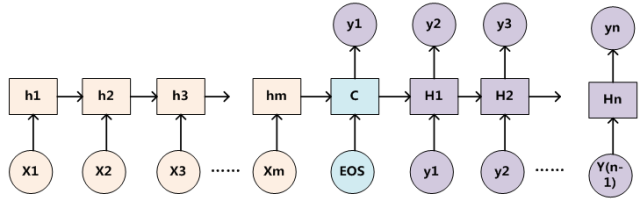


图5 RNN作为具体模型的Encoder-Decoder框架

那么用图6可以较为便捷地说明注意力分配概率分布值的通用计算过程。

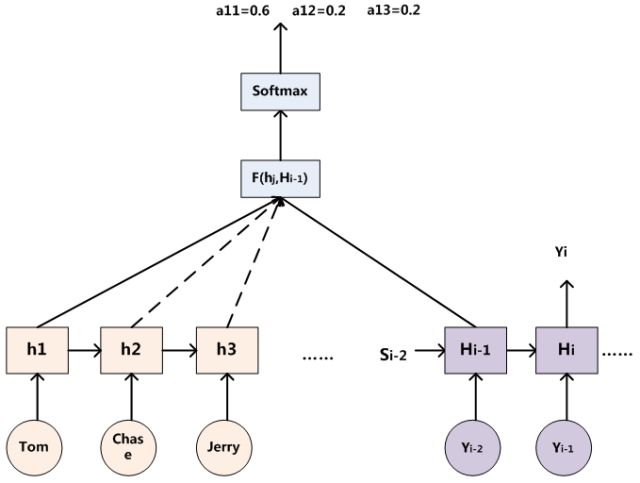


图6 注意力分配概率计算

对于采用RNN的Decoder来说，在时刻i，如果要生成yi单词，我们是可以知道Target在生成Yi之前的时刻i-1时，隐层节点i-1时刻的输出值Hi-1的，而我们的目的是要计算生成Yi时输入句子中的单词“Tom”、“Chase”、“Jerry”对Yi来说的注意力分配概率分布，那么可以用Target输出句子i-1时刻的隐层节点状态Hi-1去一一和输入句子Source中每个单词对应的RNN隐层节点状态hj进行对比，即通过函数F(hj,Hi-1)来获得目标单词yi和每个输入单词对应的对齐可能性，这个F函数在不同论文里可能会采取不同的方法，然后函数F的输出经过Softmax进行归一化就得到了符合概率分布取值区间的注意力分配概率分布数值。

绝大多数Attention模型都是采取上述的计算框架来计算注意力分配概率分布信息，区别只是在F的定义上可能有所不同。图7可视化地展示了在英语-德语翻译系统中加入Attention机制后，Source和Target两个句子每个单词对应的注意力分配概率分布。

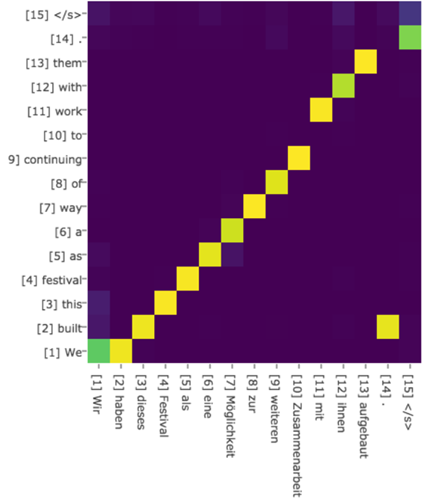


图7 英语-德语翻译的注意力概率分布

上述内容就是经典的Soft Attention模型的基本思想，那么怎么理解Attention模型的物理含义呢？一般在自然语言处理应用里会把Attention模型看作是输出Target句子中某个单词和输入Source句子每个单词的对齐模型，这是非常有道理的。

目标句子生成的每个单词对应输入句子单词的概率分布可以理解为输入句子单词和这个目标生成单词的对齐概率，这在机器翻译语境下是非常直观的：传统的统计机器翻译一般在做的过程中会专门有一个短语对齐的步骤，而注意力模型其实起的是相同的作用。

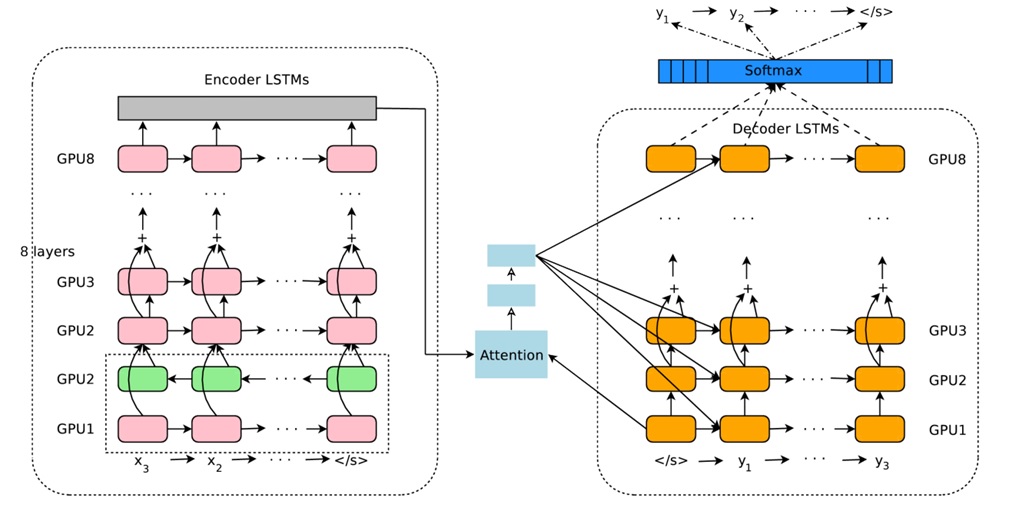


图8 Google 神经网络机器翻译系统结构图

图8所示即为Google于2016年部署到线上的基于神经网络的机器翻译系统，相对传统模型翻译效果有大幅提升，翻译错误率降低了60%，其架构就是上文所述的加上Attention机制的Encoder-Decoder框架，主要区别无非是其Encoder和Decoder使用了8层叠加的LSTM模型。

### Attention机制的本质思想

如果把Attention机制从上文讲述例子中的Encoder-Decoder框架中剥离，并进一步做抽象，可以更容易看懂Attention机制的本质思想。

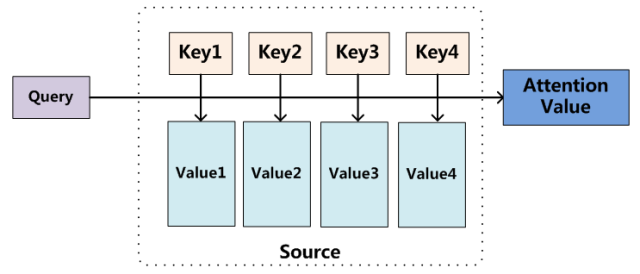


图9 Attention机制的本质思想

我们可以这样来看待Attention机制（参考图9）：将Source中的构成元素想象成是由一系列的<Key,Value>数据对构成，此时给定Target中的某个元素Query，通过计算Query和各个Key的相似性或者相关性，得到每个Key对应Value的权重系数，然后对Value进行加权求和，即得到了最终的Attention数值。所以本质上Attention机制是对Source中元素的Value值进行加权求和，而Query和Key用来计算对应Value的权重系数。即可以将其本质思想改写为如下公式：

IMG_273

其中，Lx=||Source||代表Source的长度，公式含义即如上所述。上文所举的机器翻译的例子里，因为在计算Attention的过程中，Source中的Key和Value合二为一，指向的是同一个东西，也即输入句子中每个单词对应的语义编码，所以可能不容易看出这种能够体现本质思想的结构。

当然，从概念上理解，把Attention仍然理解为从大量信息中有选择地筛选出少量重要信息并聚焦到这些重要信息上，忽略大多不重要的信息，这种思路仍然成立。聚焦的过程体现在权重系数的计算上，权重越大越聚焦于其对应的Value值上，即权重代表了信息的重要性，而Value是其对应的信息。

从图9可以引出另外一种理解，也可以将Attention机制看作一种软寻址(Soft\_Addressing) :Source可以看作存储器内存储的内容，元素由地址Key和值Value组成，当前有个Key=Query的查询，目的是取出存储器中对应的Value值，即Attention数值。通过Query和存储器内元素Key的地址进行相似性比较来寻址，之所以说是软寻址，指的不像一般寻址只从存储内容里面找出一条内容，而是可能从每个Key地址都会取出内容，取出内容的重要性根据Query和Key的相似性来决定，之后对Value进行加权求和，这样就可以取出最终的Value值，也即Attention值。所以不少研究人员将Attention机制看作软寻址的一种特例，这也是非常有道理的。

至于Attention机制的具体计算过程，如果对目前大多数方法进行抽象的话，可以将其归纳为两个过程：第一个过程是根据Query和Key计算权重系数，第二个过程根据权重系数对Value进行加权求和。而第一个过程又可以细分为两个阶段：第一个阶段根据Query和Key计算两者的相似性或者相关性；第二个阶段对第一阶段的原始分值进行归一化处理；这样，可以将Attention的计算过程抽象为如图10展示的三个阶段。

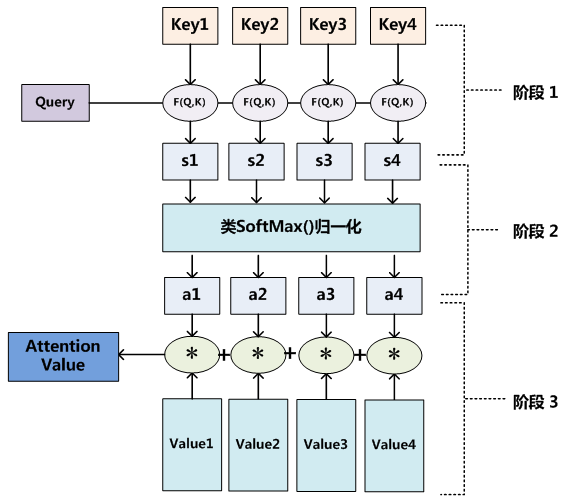
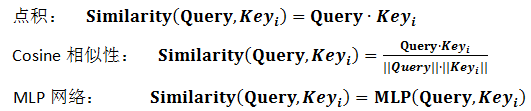


图10 三阶段计算Attention过程

上图10中，Query来自输出端目标句子中的第i-1个词语对应的编码，Key和Value来自输入端源句子中所有词语对应的编码(包括Key1,Key2,Key3,Key4;以及Value1,Value2,Value3, Value4)。计算Query和Key1~Key4之间的4个注意力系数(a1~a4)，这4个注意力系数反映出了目标句子中第i个词语与源句子中不同词语之间的注意力比例。这4个注意力系数(a1~a4)分别与Value1~Value4相乘，4个相乘的结果再相加，获得了目标句子中第i个词语对应的语义编码Ci（图3中所示）。语义编码Ci经过某种函数F转换为目标句子中词语对应的语义编码。这样，体现出了目标句子中不同词语与原句子中不同词语之间具有不同程度的关联信息，从而有效提升源句子至目标句子之间的转换。

具体而言分为三个阶段。计算在第一个阶段，可以引入不同的函数和计算机制，根据Query和某个Key\_i，计算两者的相似性或者相关性，最常见的方法包括：求两者的向量点积、求两者的向量Cosine相似性或者通过再引入额外的神经网络来求值，即如下方式：



第一阶段产生的分值根据具体产生的方法不同其数值取值范围也不一样，第二阶段引入类似SoftMax的计算方式对第一阶段的得分进行数值转换，一方面可以进行归一化，将原始计算分值整理成所有元素权重之和为1的概率分布；另一方面也可以通过SoftMax的内在机制更加突出重要元素的权重。即一般采用如下公式计算：

IMG_276

第二阶段的计算结果a\_i即为value\_i对应的权重系数，然后进行加权求和即可得到Attention数值：

IMG_277

通过如上三个阶段的计算，即可求出针对Query的Attention数值，目前绝大多数具体的注意力机制计算方法都符合上述的三阶段抽象计算过程。

### Self Attention模型

通过上述对Attention本质思想的梳理，我们可以更容易理解本节介绍的Self Attention模型。Self Attention也经常被称为intra Attention（内部Attention），最近一年也获得了比较广泛的使用，比如Google最新的机器翻译模型内部大量采用了Self Attention模型。

在一般任务的Encoder-Decoder框架中，输入Source和输出Target内容是不一样的，比如对于英-中机器翻译来说，Source是英文句子，Target是对应的翻译出的中文句子，Attention机制发生在Target的元素Query和Source中的所有元素之间。而Self Attention顾名思义，指的不是Target和Source之间的Attention机制，而是Source内部元素之间或者Target内部元素之间发生的Attention机制，也可以理解为Target=Source这种特殊情况下的注意力计算机制。其具体计算过程是一样的，只是计算对象发生了变化而已，所以此处不再赘述其计算过程细节。

如果是常规的Target不等于Source情形下的注意力计算，其物理含义正如上文所讲，比如对于机器翻译来说，本质上是目标语单词和源语单词之间的一种单词对齐机制。那么如果是Self Attention机制，一个很自然的问题是：通过Self Attention到底学到了哪些规律或者抽取出了哪些特征呢？或者说引入Self Attention有什么增益或者好处呢？我们仍然以机器翻译中的Self Attention来说明，图11和图12是可视化地表示Self Attention在同一个英语句子内单词间产生的联系。

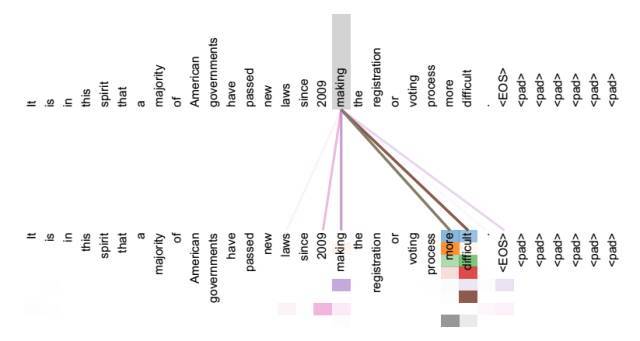


图11 可视化Self Attention实例

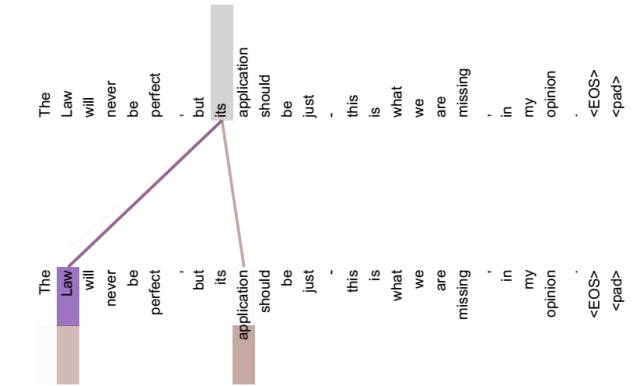


图12 可视化Self Attention实例

从两张图（图11、图12）可以看出，Self Attention可以捕获同一个句子中单词之间的一些句法特征（比如图11展示的有一定距离的短语结构）或者语义特征（比如图12展示的its的指代对象Law）。

很明显，引入Self Attention后会更容易捕获句子中长距离的相互依赖的特征，因为如果是RNN或者LSTM，需要依次序序列计算，对于远距离的相互依赖的特征，要经过若干时间步步骤的信息累积才能将两者联系起来，而距离越远，有效捕获的可能性越小。

但是Self-Attention在计算过程中会直接将句子中任意两个单词的联系通过一个计算步骤直接联系起来，所以远距离依赖特征之间的距离被极大缩短，有利于有效地利用这些特征。除此外，Self-Attention对于增加计算的并行性也有直接帮助作用。这是为何Self Attention逐渐被广泛使用的主要原因。

由于self-attention可以捕获源句子自身不同单词之间的注意力的系数，从而在求取目标词语与源句子中不同词语之间的注意力系数时，可有效提升注意力系数计算的准确度。Transformer即利用了这种机制，采用encoder来捕获源句子中不同单词之间的注意力系数，并在此基础上计算目标句子中不同单词对应的Query编码和encoder所输出的Key编码之间的相似度，从而有效提升这一相似度计算的准确性。例如，在图10中，可以采用encoder首先计算源句子中的四个单词之间的注意力系数，并在此基础上输出源句子的Key1~Key4和Value1~Value4；这样目标句子中的每个单词对应的Query编码与源句子中每个单词对应的Key编码之间的相似度计算更准确，从而提升了源句子与目标句子之间转换的准确性。

### Attention机制的应用

前文有述，Attention机制在深度学习的各种应用领域都有广泛的使用场景。上文在介绍过程中我们主要以自然语言处理中的机器翻译任务作为例子，下面分别再从图像处理领域和语音识别选择典型应用实例来对其应用做简单说明。

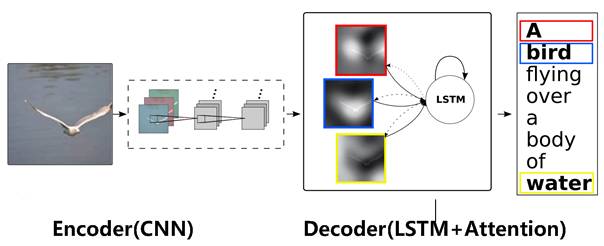


图13 图片-描述任务的Encoder-Decoder框架

图片描述（Image-Caption）是一种典型的图文结合的深度学习应用，输入一张图片，人工智能系统输出一句描述句子，语义等价地描述图片所示内容。很明显这种应用场景也可以使用Encoder-Decoder框架来解决任务目标，此时Encoder输入部分是一张图片，一般会用CNN来对图片进行特征抽取，Decoder部分使用RNN或者LSTM来输出自然语言句子（参考图13）。

此时如果加入Attention机制能够明显改善系统输出效果，Attention模型在这里起到了类似人类视觉选择性注意的机制，在输出某个实体单词的时候会将注意力焦点聚焦在图片中相应的区域上。图14给出了根据给定图片生成句子“A person is standing on a beach with a surfboard.”过程时每个单词对应图片中的注意力聚焦区域。

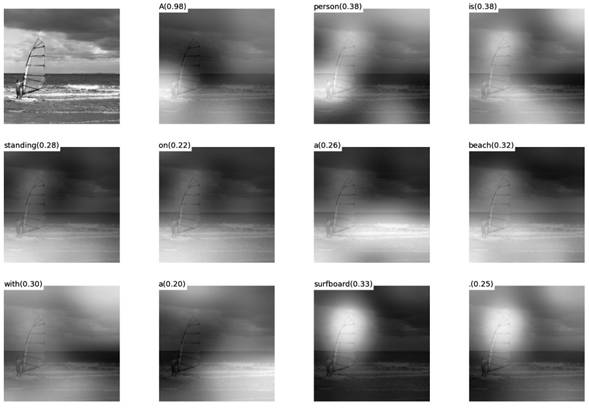


图14 图片生成句子中每个单词时的注意力聚焦区域

图15给出了另外四个例子形象地展示了这种过程，每个例子上方左侧是输入的原图，下方句子是人工智能系统自动产生的描述语句，上方右侧图展示了当AI系统产生语句中划横线单词的时候，对应图片中聚焦的位置区域。比如当输出单词dog的时候，AI系统会将注意力更多地分配给图片中小狗对应的位置。

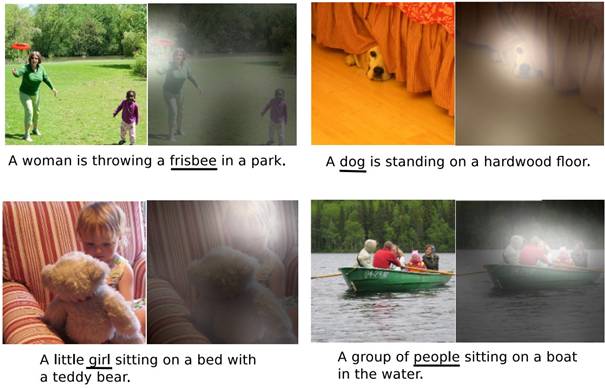


图15 图像描述任务中Attention机制的聚焦作用

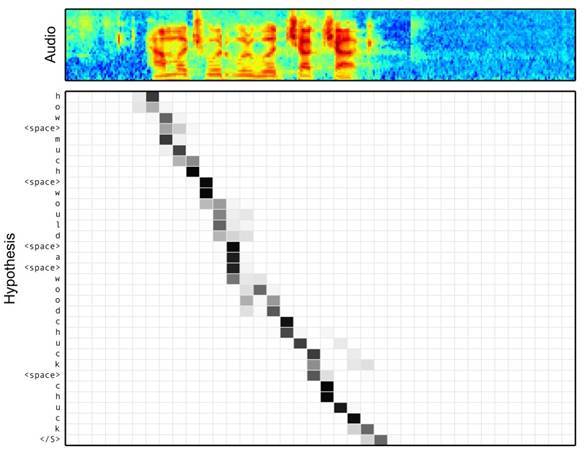


图16 语音识别中音频序列和输出字符之间的Attention

语音识别的任务目标是将语音流信号转换成文字，所以也是Encoder-Decoder的典型应用场景。Encoder部分的Source输入是语音流信号，Decoder部分输出语音对应的字符串流。

图16可视化地展示了在Encoder-Decoder框架中加入Attention机制后，当用户用语音说句子how much would a woodchuck chuck时，输入部分的声音特征信号和输出字符之间的注意力分配概率分布情况，颜色越深代表分配到的注意力概率越高。从图中可以看出，在这个场景下，Attention机制起到了将输出字符和输入语音信号进行对齐的功能。

上述内容仅仅选取了不同AI领域的几个典型Attention机制应用实例，Encoder-Decoder加Attention架构由于其卓越的实际效果，目前在深度学习领域里得到了广泛的使用，了解并熟练使用这一架构对于解决实际问题会有极大帮助。