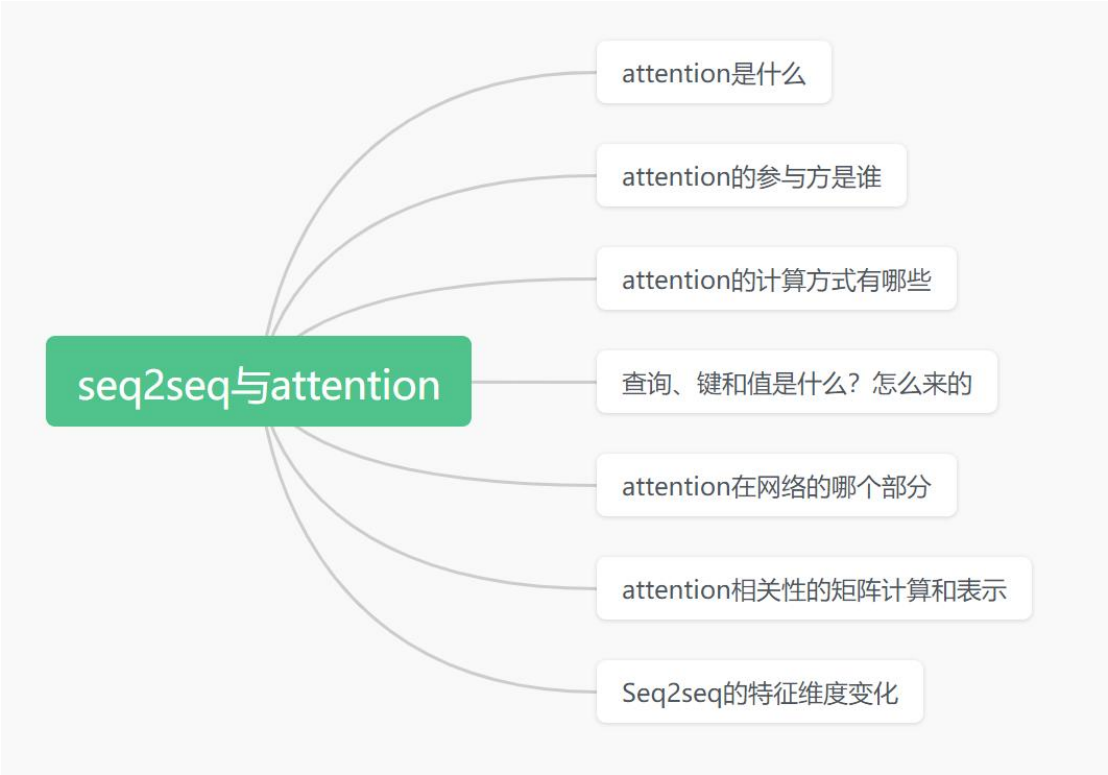
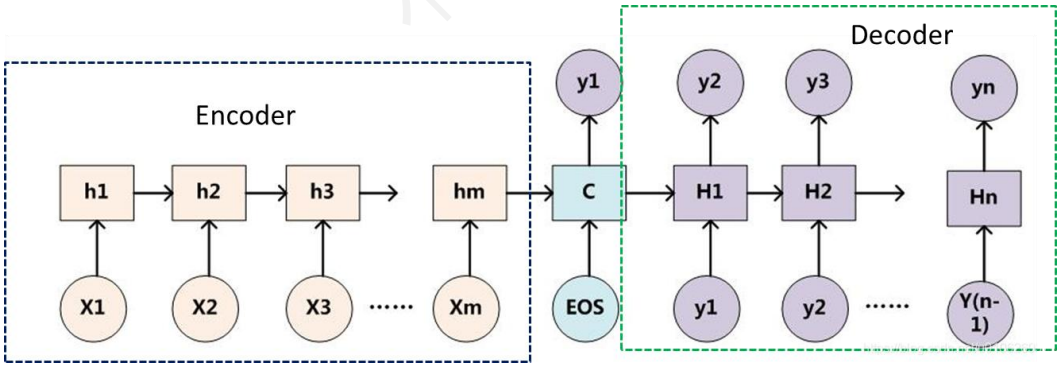


Seq2seq 的缺点



Seq2seq 框架的中间状态向量来自于输入网络隐藏层最后的隐状态，一般来说它是一个大小固定的向量，因此它能储存的信息就是有限的，当句子长度不断变长，由于后方的 decoder 网络的所有信息都来自中间状态，中间状态需要表达的信息就越来越多。第二点是先输入的内容携带的信息会被后输入的信息稀释覆盖掉。输入序列越长，这个现象就越严重。

这就使得在解码的时候一开始就没有获得输入序列足够的信息，那么解码的准确度自然也就要打个折扣了。Encoder-Decoder 网络就像我们的短时记忆一样，存在着容量的上限，在语句信息量过大时，中间状态就作为一个信息的瓶颈阻碍翻译了。

为了克服这个问题，研究人员提出了使用 attention，Attention 是挑重点，就算文本比较长，也能从中间抓住重点，不丢失重要的信息（应用 attention 机制最著名的应当是 transformer 了，后面学习的时候会再进一步介绍）。

什么是 attention

注意力机制的理解

在 Attention 诞生之前，已经有 CNN 和 RNN 及其变体模型了，那为什么还要引入 attention 机制？主要有两个方面的原因，如下：

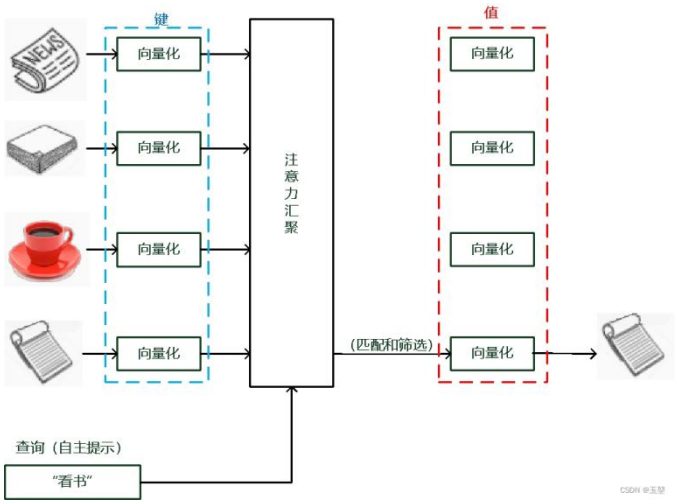
（1）**计算能力的限制**：当要记住很多“信息“，模型就要变得更复杂，然而目前计算能力依然是限制神经网络发展的瓶颈。

（2）**优化算法的限制**：LSTM 只能在一定程度上缓解 RNN 中的长距离依赖问题，且信息“记忆”能力并不高。

注意力机制其实是源自于人对于外部信息的处理能力。由于人每一时刻接受的信息都是无比的庞大且复杂，远远超过人脑的处理能力，因此人在处理信息的时候，会将注意力放在需要关注的信息上，对于其他无关的外部信息进行过滤，这种处理方式被称为注意力机制。

从特征的角度考虑就是，对于原始的数据，会挖掘得到其“特征值”（value），这个是描述数据本质特征的向量，这个特征值会对应一个“索引”，类似字典中的键，称为键向量（key），而所谓的注意力就是查询匹配当前的特征和原始数据特征的相似度，就需要从当前数据特征中挖掘一个用于查询匹配的向量，即查询向量（query）。

注意力机制是通过 Query 与 Key 的注意力汇聚（指的是对 Query 和 Key 的相关性进行建模，比如向量点乘，求余弦值等，计算相关性），实现对 Value 的注意力权重分配，生成最终的输出结果。如下图所示：



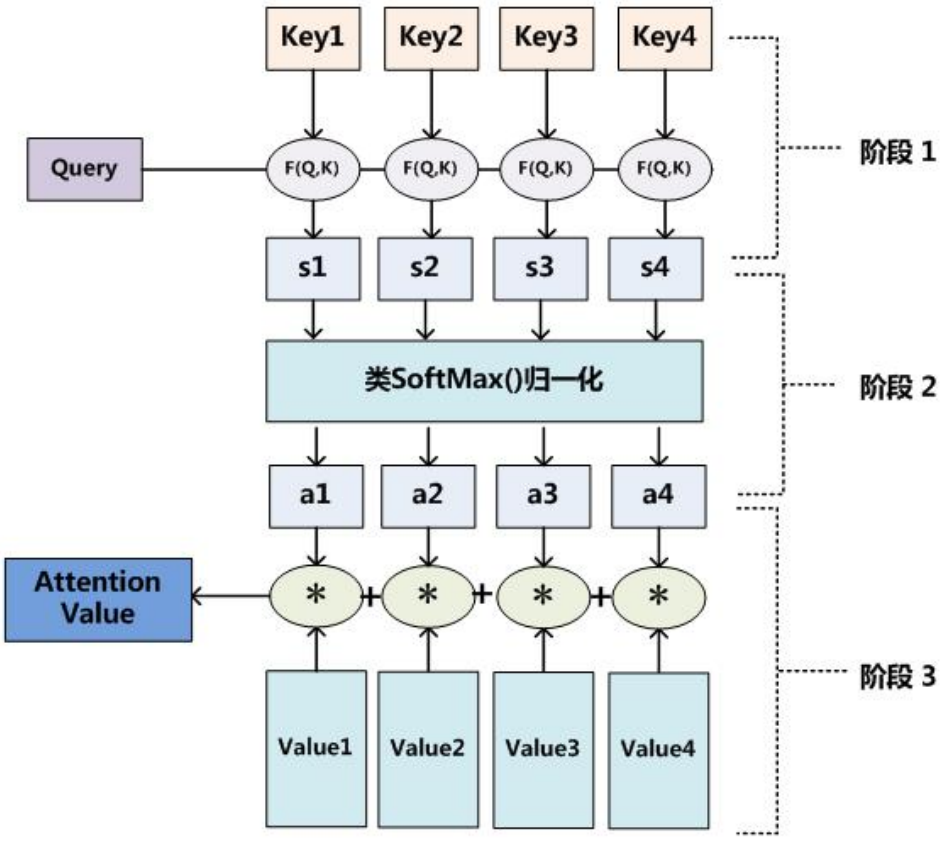
注意力机制模型

从本质上理解，Attention 是从大量信息中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。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。

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展示的三个阶段。



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，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。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：

点积：$Similarity(\boldsymbol{Query}, \boldsymbol{Key_i}) = \boldsymbol{Query} \cdot \boldsymbol{Key_i}$

Cosine 相似性：$Similarity(\boldsymbol{Query}, \boldsymbol{Key_i}) = \frac{\boldsymbol{Query} \cdot \boldsymbol{Key_i}}{\|\boldsymbol{Query}\| \cdot \|\boldsymbol{Key_i}\|}$

MLP 网络：$Similarity(\boldsymbol{Query}, \boldsymbol{Key_i}) = MLP(\boldsymbol{Query}, \boldsymbol{Key_i})$

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：

$$\boldsymbol{\alpha_i} = Softmax(\boldsymbol{Similarity_i}) = \frac{e^{\boldsymbol{Similarity_i}}}{\sum_{j=1}^{L_x} e^{\boldsymbol{Similarity_i}}}$$

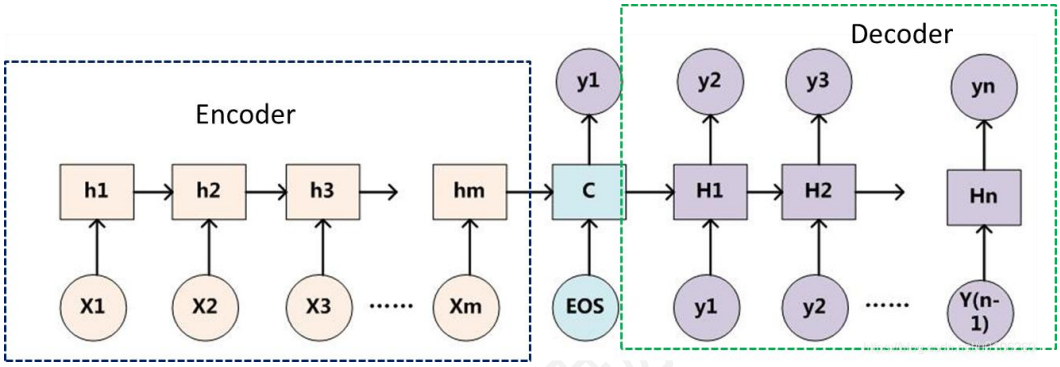
第二阶段的计算结果 $\boldsymbol{\alpha_i}$即为 $Value_i$对应的权重系数，然后进行加权求和即可得到 Attention 数值：

$$\text{Attention}(\boldsymbol{Query}, \boldsymbol{Source}) = \sum_{i=1}^{L_x} \boldsymbol{\alpha_i} \cdot Value_i$$

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。

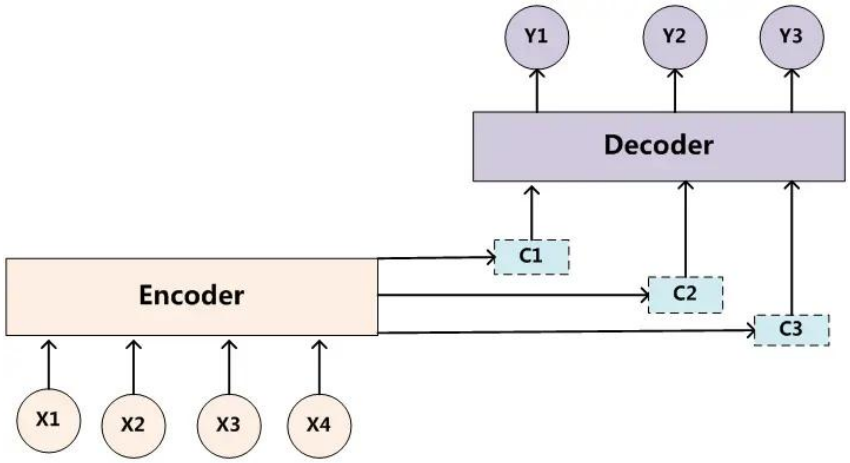
基于 attention 的 Seq2seq 的框架

原始 Seq2seq 和注意力 Seq2seq 的对比



我们以常用的原始 seq2seq 为例，每一次 decoder 的输入包括三部分，encoder 层输出的 c，上一时刻的输出 yt-1（可以是预测或者真实的），上一时刻 decoder 隐藏层的隐状态 ht-1.

传统的循环神经网络中，y1、y2 和 y3 的计算都是基于同一个 C. 但是原始输入中不同单词对 y1、y2 和 y3 的影响是不同的，所以，很自然地就引入了注意力机制：



可以看到，加上 attention 机制后的模型和传统 seq2seq 的区别就是中间状态向量的计算，也就是要将中间状态向量不再是固定不变的，而是每次通过 attention 融合计算得到的。

那么接下来的关键就是按照 attention 机制明确 query，key 和 value 了，并选

择合适的计算模型计算得到。

Seq2seq 中 attention 的计算

前面我们介绍了 attention 的通用模型，而 attention 应用于 seq2seq 的模块是中间向量，那么接下来就根据通用模型介绍 seq2seq 中的 attention 到底是如何计算的。

直观上来说，“attention”是指 decoder 模块中上一时刻的隐状态 st-1 与 encoder 模块中所有时刻的隐状态 h1:x 的相关性，所以 query 向量就是由 st-1 变换求得，key 和 value 向量由 h1:x 变换求得：

即

query=Tq*st-1

key=Tk*h1:x，

value=h1:x，这里也可以进行一些变换处理，最简单就是直接用 encoder 模块的隐藏层向量。

然后求 query 和 key 的相似性参数：

$$Similarity(\boldsymbol{Query}, \boldsymbol{Key_i}) = \boldsymbol{Query} \cdot \boldsymbol{Key_i}$$

然后进行归一化，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：

$$\boldsymbol{\alpha_i} = Softmax(\boldsymbol{Similarity_i}) = \frac{e^{\boldsymbol{Similarity_i}}}{\sum_{j=1}^{L_x} e^{\boldsymbol{Similarity_i}}}$$

第二阶段的计算结果 $\boldsymbol{\alpha_i}$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：

$$\text{Attention}(\boldsymbol{Query}, \boldsymbol{Source}) = \sum_{i=1}^{L_x} \boldsymbol{\alpha_i} \cdot Value_i$$

这样就可以在计算 decoder 输出时候每一次的输入 c0 都是和输入最息息相关的，而且对之前的输入也有比较好的记忆性，因为可以提高表征前面输入单词的隐藏层的 hi 的权重。

Seq2seq 的特征维度变化

首先定义一下变量含义：

Encoder 模块

input_feature 输入特征维度，这里注意区分是原始的输入特征维度还是 embedding 之后的特征维度；

enc_hid_dim:encoder 模块中的隐藏层特征向量维度；但是需要注意的是若选择双向的 RNN 系列模型，则最终的隐藏层特征向量维度需要*2；

enc_out_dim：encoder 模块中输出层的特征向量维度，一般可以选择和 enc_hid_dim 一致，加了 attention 之后，也可以选择和 dec_hid_dim 一致。

seq_len:句子的长度，也是隐藏层特征向量的数目；

Decoder 模块：

dec_hid_dim：decoder 模块中的隐藏层特征向量维度；

输入包括下面几个部分：

dec_input:decoder 模块上一时刻的输出特征向量（可以是预测也可以是真实的输出特征向量）

c:经过 attention 层之后的中间特征向量；

dec_out:decoder 模块的输出层特征向量维度；

Attention 模块

最终的输出中间特征向量 c:value 向量维度一致，value 向量为原始的 encoder 模块中的隐藏层特征向量，或者经过变换的向量；

Attention 计算 key 和 query 向量时：

query 向量需要经过 st-1（维度是 dec_hid_dim*1），key 矩阵需要经过 h1:x（维度是 enc_hid_dim*seq_len），变换后的 key 和 query 向量维度是自定义的，可以定为 dec_hid_dim，所以大的变换矩阵维度

Tq=dec_hid_dim*（dec_hid_dim）

Tk=dec_hid_dim*（enc_hid_dim）

结合起来的 T=dec_hid_dim*（dec_hid_dim+enc_hid_dim）

变换后得到的 query 和 key 向量再进行点乘得到相关性 ai，相关性的维度一共是 1*seq_len，然后最终与 h1:seq_len 加权相乘得到最终的 c，维度是 enc_hid_dim。

代码

```
import torch
from torch import nn
import numpy as np
import torch.nn.functional as F
import random
```

```
class Encoder(nn.Module):
    def __init__(self, input_size, enc_hiddenc_dim, num_layers,dec_hid_dim):
        super().__init__()
        #和传统 seq2seq 一样定义输入特征维度，隐藏层特征维度，隐藏层数，输出层特征维度
        self.input_size = input_size
        self.enc_hiddenc_dim = enc_hiddenc_dim
        self.num_layers = num_layers
        self.dec_hid_dim = dec_hid_dim
        #选择 encoder 模块的模型，这里选用 gru,并传入参数输入特征维度，隐藏层特征维度，隐藏层数，
        self.gru = nn.GRU(input_size = input_size,
                          hiddenc_size = enc_hiddenc_dim,
                          num_layers = num_layers,
                          bidirectional=True)
        #根据利用全连接层将隐藏层特征维度转换为输出层特征维度
        self.fc = nn.Linear(enc_hiddenc_dim * 2, dec_hid_dim)   # 拼接后

    def forward(self, init_input, h0):
        #前向传播得到 encoder 模块的输出层和隐藏层（输出层是隐藏层的全连接变换，也可以直接用隐藏层作为输出层）
        enc_output, hidden = self.gru(init_input, h0)
        # 将正向最后时刻的输出与反向最后时刻的输出进行拼接，得到的维度应该是[batch,enc_hiddenc_dim*2]
        # hidden[-1,:,:] 反向最后时刻的输出, hidden[-2,:,:] 正向最后时刻的输出
        h_m = torch.cat((hidden[-2, :, :], hidden[-1, :, :]), dim=1) # 在第一个维度进行拼接
        #这里的 s0 就是 decoder 模块的第 0 个隐藏层特征向量
        s0 = self.fc(h_m)
        #enc_output 的维度是[seq_len,batch,enc_hiddenc_dim*2]，s0 的维度是[batch,dec_hiddenc_dim]。
        return enc_output,s0


class Attention(nn.Module):
    def __init__(self, enc_hid_dim, dec_hid_dim):
        super().__init__()
        #第一步求 query 和 key 向量，query 向量根据 decoder 模块中上一时刻的隐藏层 st-1 求得，key 向量由 encoder 模块中所有时刻的隐藏层 h1: x 分别求得，value 向量这里不经过变换
        self.attn = nn.Linear((enc_hid_dim * 2) + dec_hid_dim, dec_hid_dim, bias=False)   # 输出的维度是任意的
```

```
        #用来计算相关性系数
        self.v = nn.Linear(dec_hid_dim, 1, bias=False)    # 将输出维度置为 1

    def forward(self, s, enc_output):
        # s = [batch_size, dec_hiddenc_dim]
        # enc_output = [seq_len, batch_size, enc_hid_dim * 2]

        batch_size = enc_output.shape[1]
        seq_len = enc_output.shape[0]

        # repeat decoder hidden state seq_len times
        # s = [seq_len, batch_size, dec_hid_dim]
        ##维度变化：  [batch_size, dec_hid_dim]=>[1, batch_size,
dec_hid_dim]=>[seq_len, batch_size, dec_hid_dim]
        s = s.unsqueeze(0).repeat(seq_len, 1, 1)

        energy = torch.tanh(self.attn(torch.cat((s, enc_output), dim=2)))
        #计算相关性系数，维度变化：  [seq_len, batch_size,
dec_hid_dim]=>[seq_len，batch_size, 1] => [seq_len, batch_size]
        attention = self.v(energy).squeeze(2)

        return F.softmax(attention, dim=0).transpose(0, 1)    # [batch_size, seq_len]


class Decoder(nn.Module):
    #初始化基本参数，输出层特征维度，encoder 模块隐藏层特征维度，deco
模块隐藏层特征维度，attention 层
    def __init__(self, output_dim, enc_hid_dim, dec_hid_dim, dropout, attention):
        super().__init__()
        self.output_dim = output_dim
        self.attention = attention
        # self.embedding = nn.Embedding(output_dim, emb_dim)
        self.rnn = nn.GRU((enc_hid_dim * 2) + 1, dec_hid_dim)
        # self.fc_out = nn.Linear((enc_hid_dim * 2) + dec_hid_dim + emb_dim,
output_dim)
        self.fc_out = nn.Linear((enc_hid_dim * 2) + dec_hid_dim + 1, output_dim)
        self.dropout = nn.Dropout(dropout)

    def forward(self, dec_input, s, enc_output):

        # dec_input = [batch_size]
        # s = [batch_size, dec_hid_dim]
        # enc_output = [src_len, batch_size, enc_hid_dim * 2]
```

```
        dec_input = dec_input.unsqueeze(1) # dec_input = [batch_size, 1]

        # embedded = self.dropout(self.embedding(dec_input)).transpose(0, 1) #
embedded = [1, batch_size, emb_dim]
        dropout_dec_input = self.dropout(dec_input).transpose(0, 1).unsqueeze(2) #
[1, batch_size]=>[1,batch,1]
       #与传统 decoder 模块的核心变化，这里需要计算隐藏层中上一个时刻的
特征向量与 encoder 模块的所有时刻隐藏层特征向量的相关性
        # a = [batch_size, 1, src_len]
        a = self.attention(s, enc_output).unsqueeze(1)
        print("a.shape:",a.shape)

        # enc_output = [batch_size, src_len, enc_hid_dim * 2]
        enc_output = enc_output.transpose(0, 1)
        #然后根据相关性系数计算得到“attention”之后的中间特征向量 c，并作
为 decoder 模块的输入
        # c = [1, batch_size, enc_hid_dim * 2]
        c = torch.bmm(a, enc_output).transpose(0, 1)

        # rnn_input = [1, batch_size, (enc_hid_dim * 2) + emb_dim]
        # rnn_input = torch.cat((embedded, c), dim = 2)
        rnn_input = torch.cat((dropout_dec_input, c), dim = 2) # rnn_input = [1,
batch_size, (enc_hid_dim * 2) + 1]

        # dec_output = [src_len(=1), batch_size, dec_hid_dim]
        # dec_hidden = [n_layers * num_directions(=1), batch_size, dec_hid_dim]
        dec_output, dec_hidden = self.rnn(rnn_input, s.unsqueeze(0)) #
s.unsqueeze(0):[1,batch_size, dec_hid_dim]

        dec_output = dec_output.squeeze(0) # dec_output:[batch_size, dec_hid_dim]
        c = c.squeeze(0)    # c:[batch_size, enc_hid_dim * 2]
        # dec_input:[batch_size, 1]

        # pred = [batch_size, output_dim]
        pred = self.fc_out(torch.cat((dec_output, c, dec_input), dim = 1))

        return torch.tanh(pred), dec_hidden.squeeze(0)


class Seq2Seq(nn.Module):
    def __init__(self, encoder, decoder, device):
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder
```

```
        self.device = device

    def forward(self, src, trg, teacher_forcing_ratio=0.5):
        # src = [src_len, batch_size]
        # trg = [trg_len, batch_size]
        # teacher_forcing_ratio 是选择几率

        batch_size = src.shape[1]
        trg_len = trg.shape[0]
        trg_vocab_size = self.decoder.output_dim

        # 存储所有时刻的输出
        outputs = torch.zeros(trg_len, batch_size, trg_vocab_size).to(self.device)
# 存储 decoder 的所有输出

        enc_output, s = self.encoder(src)

        # first input to the decoder is the <sos> tokens
        dec_input = trg[0, :]  # target 的第一列，即全是<SOS>
        #传统 Seq2Seq 是直接将句子中每个词连续不断输入 Decoder 进行训练，
        # 而引入 Attention 机制之后，需要能够人为控制一个词一个词进行输入（因为输入每个词到 Decoder，需要再做一些运算），
        # 所以在代码中会看到使用了 for 循环，循环 trg_len-1 次（开头的 手动输入，所以循环少一次）。

        for t in range(1, trg_len):
            dec_output, s = self.decoder(dec_input, s, enc_output)

            # 存储每个时刻的输出
            outputs[t] = dec_output

            # 用 TeacherForce 机制
            teacher_force = random.random() < teacher_forcing_ratio

            # 获取预测值
            top1 = dec_output.argmax(1)

            dec_input = trg[t] if teacher_force else top1

        return outputs
```

参考资料

https://blog.csdn.net/weixin_43610114/article/details/126684999
https://zh.d2l.ai/chapter_attention-mechanisms/attention-cues.html
https://blog.csdn.net/weixin_45727931/article/details/115010609

Cho et al., Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation

Sutskever et al., Sequence to Sequence Leaerning with Neural Networks