Text Summarization-2

Text Summarization & Seq2seq Architecture

1. Text Rank

Text Rank 是抽取式的摘要方法。借鉴 Page Rank 的思路，将文章拆分为一个个句子，将句子向量化表示并分配一个初始权重，再计算它们之间的相似度，并存储到矩阵中，把相似度矩阵转化为图，其中句子是节点，相似度是节点间的权重。将相似度矩阵与句子初始权重不断进行矩阵相乘，将最后趋于稳定的值作为句子最终的权重，取到 top-k 权重的句子，作为最终的摘要。

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 0 & 1 & \frac{1}{2} \\ 2 & \frac{1}{3} & 0 & 0 & 0 \\ 3 & \frac{1}{3} & \frac{1}{2} & 0 & \frac{1}{2} \\ 4 & \frac{1}{3} & \frac{1}{2} & 0 & 0 \end{array}$$

transition matrix

$$v = \begin{pmatrix} 0.25 \\ 0.25 \\ 0.25 \\ 0.25 \end{pmatrix}, \quad Av = \begin{pmatrix} 0.37 \\ 0.08 \\ 0.33 \\ 0.20 \end{pmatrix}, \quad A^2 v = A(Av) = A \begin{pmatrix} 0.37 \\ 0.08 \\ 0.33 \\ 0.20 \end{pmatrix} = \begin{pmatrix} 0.43 \\ 0.12 \\ 0.27 \\ 0.16 \end{pmatrix}$$

$$A^3 v = \begin{pmatrix} 0.35 \\ 0.14 \\ 0.29 \\ 0.20 \end{pmatrix}, \quad A^4 v = \begin{pmatrix} 0.39 \\ 0.11 \\ 0.29 \\ 0.19 \end{pmatrix}, \quad A^5 v = \begin{pmatrix} 0.39 \\ 0.13 \\ 0.28 \\ 0.19 \end{pmatrix}$$

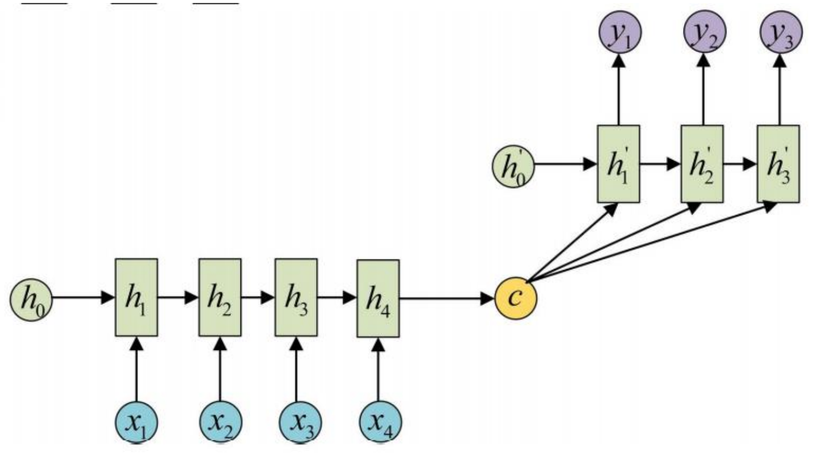
$$A^6 v = \begin{pmatrix} 0.38 \\ 0.13 \\ 0.29 \\ 0.19 \end{pmatrix}, \quad A^7 v = \begin{pmatrix} 0.38 \\ 0.12 \\ 0.29 \\ 0.19 \end{pmatrix}, \quad A^8 v = \begin{pmatrix} 0.38 \\ 0.12 \\ 0.29 \\ 0.19 \end{pmatrix}$$

A 是句子间的相似度矩阵，v 是句子的初始权重。

2. Seq2seq

2.1. Architecture

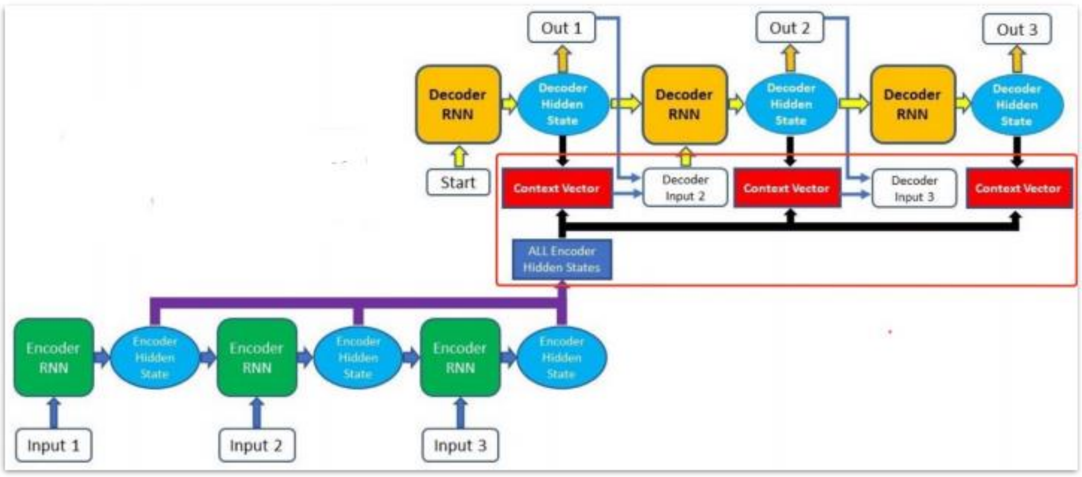
Seq2seq 是基于 RNN 的文本生成模型，由 Encoder 和 Decoder 组成。Encoder 负责将输入的本文进行编码，输出 context，Decoder 负责利用 context 进行文本生成。



2.2. Attention

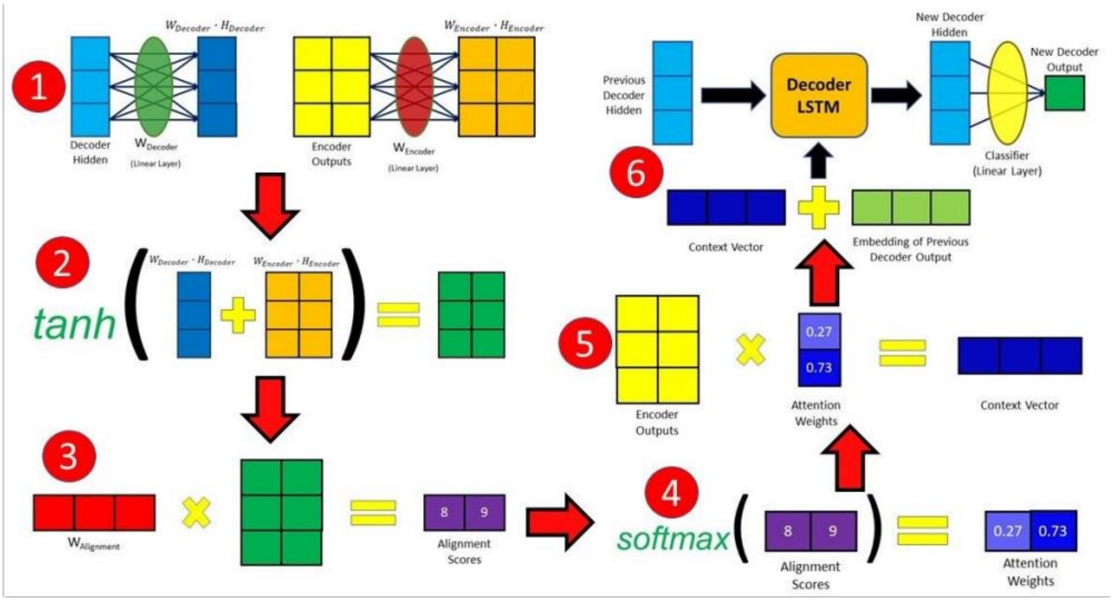
如果只将 Encoder 最后一刻的输出作为 context 送给 Decoder，那么 Decoder 根据输入文本的整体去做生成，无法每个时刻的输入进行针对性的生成。因此引入 Attention Mechanism，为 Encoder 和 Decoder 建立更紧密的联系。

Attention



Decoder 每个时刻输出的隐变量会与 Encoder 所有时刻的隐变量进行 Attention 计算，得出 context，再将 context 与当前时刻的输入拼接后送入 Decoder。

2.2.1. Additive Attention（Bahdanau Attention）



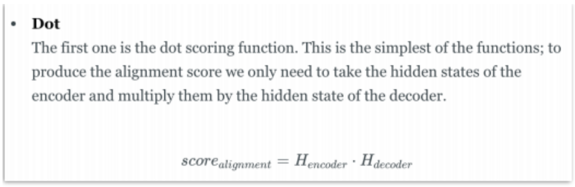
主要思路是将 decoder_hidden 和 encoder_hidden 分别进行线性变换后相加，经过激活函数，与一个可学习的参数相乘得到 score，在 softmax 后于 encoder_hidden 相乘，得到 context。

使用 Additive Attention 的 Decoder 流程如下图：

```
    # 用于注意力
    self.attention = BahdanauAttention(self.dec_units)

  def call(self, x, hidden, enc_output):
    # 编码器输出 （enc_output） 的形状 == （批大小，最大长度，隐藏层大小）
    context_vector, attention_weights = self.attention(hidden, enc_output)

    # x 在通过嵌入层后的形状 == （批大小，1，嵌入维度）
    x = self.embedding(x)

    # x 在拼接 （concatenation） 后的形状 == （批大小，1，嵌入维度 + 隐藏层大小）
    x = tf.concat([tf.expand_dims(context_vector, 1), x], axis=-1)

    # 将合并后的向量传送到 GRU
    output, state = self.gru(x)

    # 输出的形状 == （批大小 * 1，隐藏层大小）
    output = tf.reshape(output, (-1, output.shape[2]))

    # 输出的形状 == （批大小，vocab）
    x = self.fc(output)

    return x, state, attention_weights
```

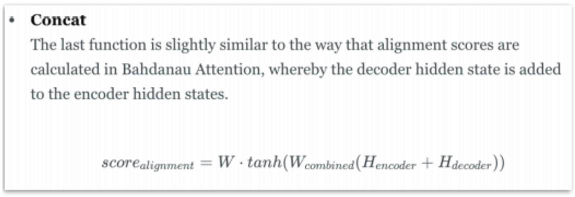
2.2.2. Multiplicative Attention(Luong Attention)

主要有三种计算方法，Attention 的计算流程于 Additive Attention， 只是将上图的第 2 步替换成以下三种计算方式。



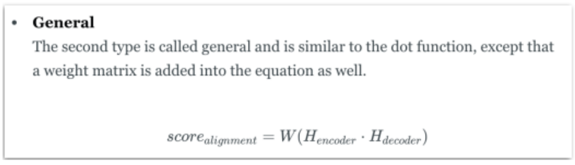
- **Dot**
  The first one is the dot scoring function. This is the simplest of the functions; to produce the alignment score we only need to take the hidden states of the encoder and multiply them by the hidden state of the decoder.

$$score_{alignment} = H_{encoder} \cdot H_{decoder}$$



- **Concat**
  The last function is slightly similar to the way that alignment scores are calculated in Bahdanau Attention, whereby the decoder hidden state is added to the encoder hidden states.

$$score_{alignment} = W \cdot tanh(W_{combined}(H_{encoder} + H_{decoder}))$$

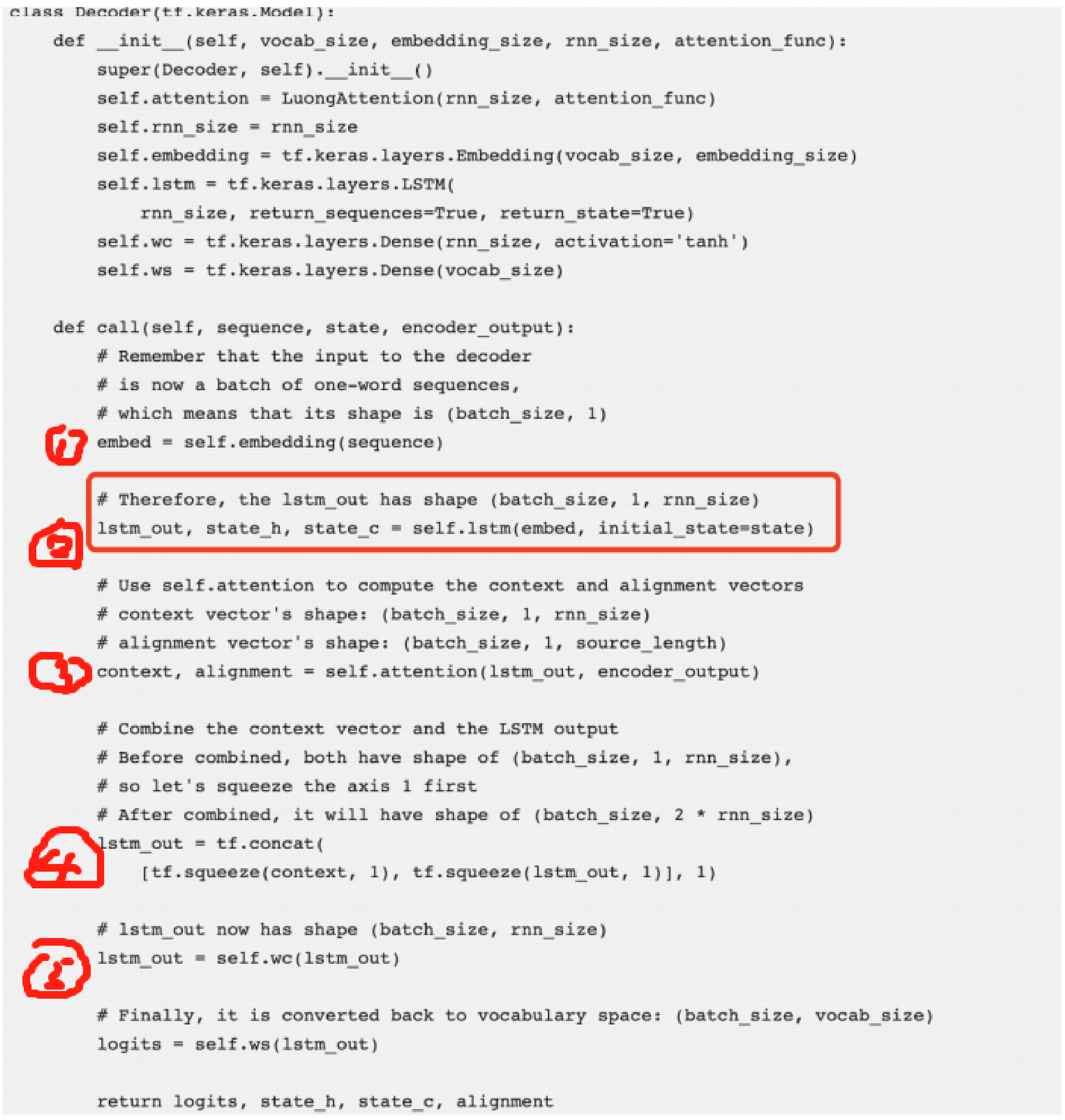


- **General**
  The second type is called general and is similar to the dot function, except that a weight matrix is added into the equation as well.

$$score_{alignment} = W(H_{encoder} \cdot H_{decoder})$$

需要注意的是，Multiplicative Attention 在 Decoder 中的流程与 Additive Attention 略有不同。

（1） 先将输入 sequence 进行 embedding 得到 embed
（2） 将 embed 送入 LSTM 得到 lstm_out
（3） 用 lstm_out 计算 context
（4） 将 lstm_out 与 context 拼接得到新的 lstm_out
（5） 将 lstm_out 经过全连接层得到 logits



```
class Decoder(tf.keras.Model):
    def __init__(self, vocab_size, embedding_size, rnn_size, attention_func):
        super(Decoder, self).__init__()
        self.attention = LuongAttention(rnn_size, attention_func)
        self.rnn_size = rnn_size
        self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_size)
        self.lstm = tf.keras.layers.LSTM(
            rnn_size, return_sequences=True, return_state=True)
        self.wc = tf.keras.layers.Dense(rnn_size, activation='tanh')
        self.ws = tf.keras.layers.Dense(vocab_size)

    def call(self, sequence, state, encoder_output):
        # Remember that the input to the decoder
        # is now a batch of one-word sequences,
        # which means that its shape is (batch_size, 1)
    ①  embed = self.embedding(sequence)

        # Therefore, the lstm_out has shape (batch_size, 1, rnn_size)
    ②  lstm_out, state_h, state_c = self.lstm(embed, initial_state=state)

        # Use self.attention to compute the context and alignment vectors
        # context vector's shape: (batch_size, 1, rnn_size)
        # alignment vector's shape: (batch_size, 1, source_length)
    ③  context, alignment = self.attention(lstm_out, encoder_output)

        # Combine the context vector and the LSTM output
        # Before combined, both have shape of (batch_size, 1, rnn_size),
        # so let's squeeze the axis 1 first
        # After combined, it will have shape of (batch_size, 2 * rnn_size)
    ④  lstm_out = tf.concat(
            [tf.squeeze(context, 1), tf.squeeze(lstm_out, 1)], 1)

        # lstm_out now has shape (batch_size, rnn_size)
    ⑤  lstm_out = self.wc(lstm_out)

        # Finally, it is converted back to vocabulary space: (batch_size, vocab_size)
        logits = self.ws(lstm_out)

        return logits, state_h, state_c, alignment
```

这两种 Decoder 流程有区别吗?

2.3. Training skills

为了防止在训练的时候，上一时刻的预测错误导致接下来的预测也发生错误，可以利用 Ground Truth 进行辅助训练。Teacher Forcing 是在每一时刻都使用真实值输入给 Decoder，但这会使得模型因为依赖标签数据，在训练过程中，模型会有较好的效果，但是在测试的时候因为不能得到 ground truth 的支持，所以如果目前生成的序列在训练过程中有很大不同，模型就会变得脆弱。

也就是说，这种模型的 cross-domain 能力会更差，也就是如果测试数据集与训练数据集来自不同的领域，模型的 performance 就会变差。

scheduled sampling 则有概率的去使用 Ground Truth 进行辅助训练，并且这个概率可以随着模型训练进展变得越来越小，在刚开始时多使用 Ground Truth 帮助模型快速学习，到后来逐渐舍弃 Ground Truth，使用自己生成的 state 进行下一次预测。

Beam Search 也是有效的解决方法。