实验三： 文本摘要

10215501433 仲韦萱

实验目的和任务

本次实验是文本摘要在医学领域的应用：从详细的病情描述中提取关键信息生成简短清晰的诊断报告。

•选择一种Encoder-Decoder结构，使用哪种模型作为Encoder或Decoder**不受限制**，大家可以按照自己的硬件设备自行选择；

•在硬件条件允许的情况下，**尽量使用多种**不同的模型并进行详细的**实验分析**；

•评估指标的使用**不受限制**，可以仅使用一种或使用多种更加全面的进行**结果分析**；

实验过程：

数据集

本次实验采用医疗领域数据集：

列名	数据类型	示例
description （输入） 病情描述	string	左侧顶骨局部骨质缺如；两侧侧脑室旁见点状密度减低。右侧额部颅板下见弧形脑脊液密度影。脑室系统扩大，脑沟、裂、池增宽。中线结构无移位。双侧乳突气化差，内见密度增高。
diagnosis （输出） 诊断报告	string	左侧顶骨局部缺如，考虑术后改变；脑内散发缺血灶；右侧额部少量硬膜下积液；双侧乳突炎。

且数据集已脱敏，即已转换为数字：

列名	数据类型	示例
description （输入） 病情描述	string	101 47 12 66 74 90 0 411 234 79 175
diagnosis （输出） 诊断报告	string	122 83 65 74 2 232 18 44 95

•18000条训练数据　•2000条测试数据

数据预处理：

先将训练数据以75：25的比例分割成训练集和验证集，数据集中 'description'为输入，'diagnosis'为输出，首先我们需要确定输入输出的最大长度，以此进行填充。另我给每个数据建立索引，方便后面的数据处理。（其实本次实验数据集已经脱敏，转化为数字表示，不需要构建词汇表并且映射索引，是后续继续改进的一个方向）

```
train_data = pd.read_csv('./data/train.csv')
test_data = pd.read_csv('./data/test.csv')
```

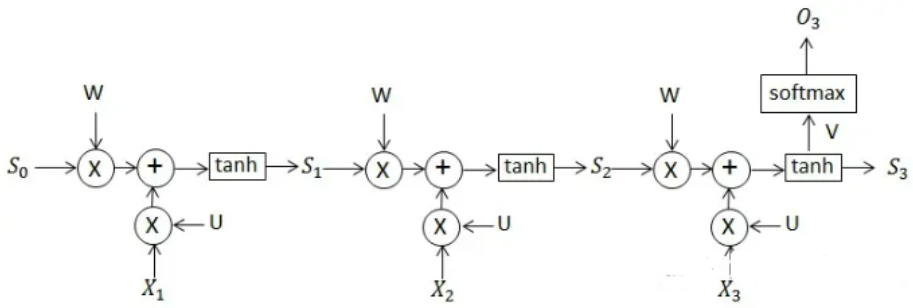
```
# 0.25比例划分训练集和验证集
train_data, val_data = train_test_split(train_data, test_size=0.25,
random_state=42)

# 使用map()函数将split()函数应用到train_data['description']中的每个元素上，计算每个元素拆分后的长度，然后使用max()函数找到最大的长度。
max_input_len = max(train_data['description'].map(lambda x: len(x.split())))
max_output_len = max(train_data['diagnosis'].map(lambda x: len(x.split())))

# 构建词汇表，长度+1特殊标记
i_vocab = set(' '.join(train_data['description']).split())
i_size = len(i_vocab) + 1

o_vocab = set(' '.join(train_data['diagnosis']).split())
o_size = len(o_vocab) + 1

# 索引号从1开始，而不是从0开始，这是因为在后续模型训练过程中，通常会使用0作为填充值的索引
i_word_to_index = {word: index + 1 for index, word in enumerate(i_vocab)}
i_index_to_word = {index + 1: word for index, word in enumerate(i_vocab)}
o_word_to_index = {word: index + 1 for index, word in enumerate(o_vocab)}
o_index_to_word = {index + 1: word for index, word in enumerate(o_vocab)}

# 填充
train_encoder_input = pad(train_data['description'], i_word_to_index,
max_input_len)
train_decoder_input = pad(train_data['diagnosis'], o_word_to_index,
max_output_len)
train_decoder_output = pad(train_data['diagnosis'], o_word_to_index,
max_output_len)

val_encoder_input = pad(val_data['description'], i_word_to_index, max_input_len)
val_decoder_input = pad(val_data['diagnosis'], o_word_to_index, max_output_len)
val_decoder_output = pad(val_data['diagnosis'], o_word_to_index, max_output_len)

test_encoder_input = pad(test_data['description'], i_word_to_index,
max_input_len)
test_decoder_input = pad(test_data['diagnosis'], o_word_to_index,
max_output_len)

# 将输入和输出转换为索引序列
def pad(data, word_to_index, max_len):
    sequences = [[word_to_index[word] for word in sentence.split()] for sentence
in data]
    return pad_sequences(sequences, maxlen=max_len, padding='post')
```

构建Seq2Seq模型

本次选取了RNN、 LSTM、 GRU三个模型进行训练：

RNN：

RNN由输入层、隐藏层、输出层组成：



S_0, S_1, S_2, S_3 是不同时刻的隐藏状态，目的是为了计算的时候能结合过去时刻的输入。
X_1, X_2, X_3是不同时刻的输入。
W是需要学习的参数，和 S 相关，每个时刻用的是同一个 W 。
U 是需要学习的参数，和 X 相关，每个时刻用的是同一个 U 。
$tanh$ 是RNN里常用的激活函数。
V 是需要学习的参数，用来改变激活后的输出数据的维度。
$softmax$ 激活函数，常用在多分类问题中，用来归一化输出最大概率的类别。
O_3 是最后时刻的输出。

本次调用了simplernn模型，包含了一个编码器和一个解码器。编码器将输入序列映射到一个隐藏状态，解码器接受该隐藏状态和输出序列作为输入，并输出预测的下一个字符或单词。模型的目标是最小化预测输出与真实输出之间的稀疏分类交叉熵损失。

```
def RNN(max_input_len, max_output_len,input_vocab_size, output_vocab_size,
latent_dim):
    encoder_inputs = tf.keras.layers.Input(shape=(max_input_len,))
    encoder_embedding = tf.keras.layers.Embedding(input_dim=input_vocab_size,
output_dim=latent_dim)(encoder_inputs)
    encoder_rnn = tf.keras.layers.SimpleRNN(latent_dim, return_state=True)
    encoder_outputs, state_h = encoder_rnn(encoder_embedding)
    encoder_states = [state_h]

    # Define the decoder
    decoder_inputs = tf.keras.layers.Input(shape=(max_output_len,))
    decoder_embedding = tf.keras.layers.Embedding(input_dim=output_vocab_size,
output_dim=latent_dim)(decoder_inputs)
    decoder_rnn = tf.keras.layers.SimpleRNN(latent_dim, return_sequences=True,
return_state=True)
    decoder_outputs, _ = decoder_rnn(decoder_embedding,
initial_state=encoder_states)
    decoder_dense = tf.keras.layers.Dense(output_vocab_size,
activation='softmax')
    decoder_outputs = decoder_dense(decoder_outputs)
    model = tf.keras.models.Model([encoder_inputs, decoder_inputs],
decoder_outputs)
    model.compile(optimizer='adam', loss='sparse_categorical_crossentropy')
    return model
```

LSTM：

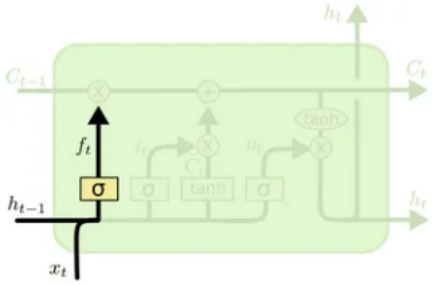
初代RNN在处理长短期信息的能力很一般，RNN是重复单一的神经网络层，LSTM中的重复模块则包含四个交互的层，三个Sigmoid 和一个tanh层，并以一种非常特殊的方式进行交互。

总的来说：LSTM拥有三种类型的门结构：遗忘门、输入门和输出门，来保护和控制细胞状态。下面示意图的顶层C(t-1)到C(t)的箭头指向，我们称之为一种细胞状态。

遗忘门:

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

遗忘门会读取上一个输出和当前输入，做一个Sigmoid的非线性映射，然后输出一个向量f(t)该向量每一个维度的值都在0到1之间，1表示完全保留，0表示完全舍弃，相当于记住了重要的，忘记了无关紧要的），最后与细胞状态相乘。

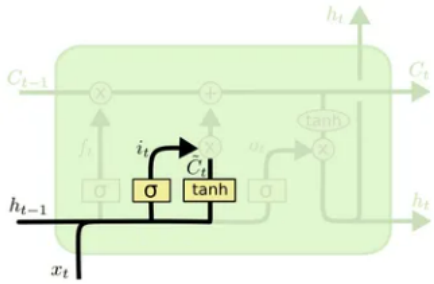


输入门:

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

第一：sigmoid层称“输入门层”决定什么值我们将要更新；

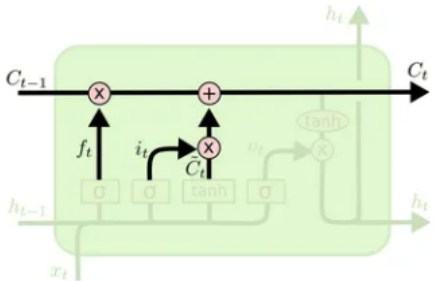
第二：一个tanh层创建一个新的候选值向量，会被加入到状态中。



细胞状态:

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

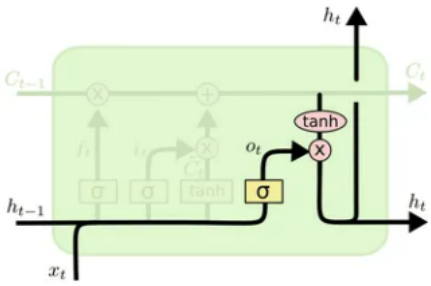
我们把旧状态与ft相乘，丢弃掉我们确定需要丢弃的信息。接着加上i(t)*C(t)。这就是新的候选值，根据我们决定更新每个状态的程度进行变化。



输出门:

$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

最终，我们要根据细胞状态，确定输出值。首先我们使用一个Sigmiod函数来去确定细胞状态的哪部分需要输出，然后把细胞状态通过tanh层处理，两者相乘得到最终我们想要输出的信息。



LSTM有进有出且当前的cell informaton是通过input gate控制之后叠加的，而RNN是叠乘，因此LSTM可以防止梯度消失或者爆炸，会比RNN好一丢。

```
def LSTM(max_input_len, max_output_len, input_vocab_size, output_vocab_size,
latent_dim):
    encoder_inputs = tf.keras.layers.Input(shape=(max_input_len,))
    encoder_embedding = tf.keras.layers.Embedding(input_dim=input_vocab_size,
output_dim=latent_dim )(encoder_inputs)
    encoder, state_h, state_c = tf.keras.layers.LSTM(latent_dim ,
return_state=True)(encoder_embedding)

    decoder_inputs = tf.keras.layers.Input(shape=(max_output_len,))
    decoder_embedding = tf.keras.layers.Embedding(input_dim=output_vocab_size,
output_dim=latent_dim)(decoder_inputs)
    decoder_lstm = tf.keras.layers.LSTM(latent_dim, return_sequences=True,
return_state=True)
    decoder_outputs, _, _ = decoder_lstm(decoder_embedding, initial_state=
[state_h, state_c])
    decoder_dense = tf.keras.layers.Dense(output_vocab_size,
activation='softmax')
    decoder_outputs = decoder_dense(decoder_outputs)
    model = tf.keras.models.Model([encoder_inputs, decoder_inputs],
decoder_outputs)
    model.compile(optimizer='adam', loss='sparse_categorical_crossentropy')
    return model
```

GRU

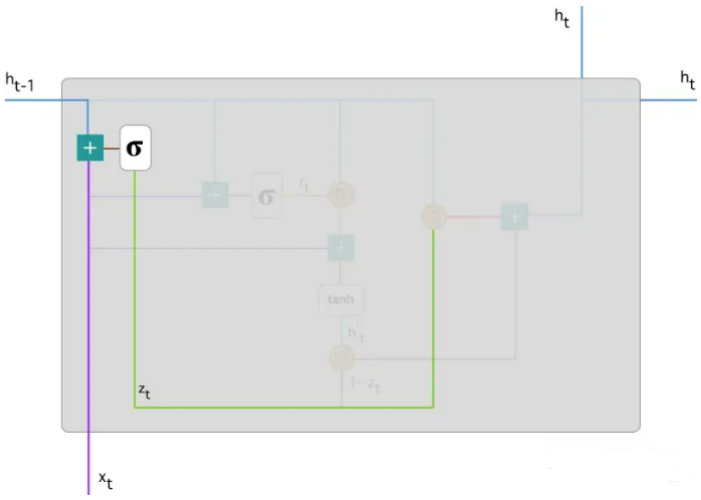
GRU组件是基于LSTM的变体，在seq2seq的序列中替代RNN充当基本组件，其目的是为了能够保留长期序列的信息同时，减少梯度消失问题；替代LSTM充当组件，其作用在于每个隐层减少了几个矩阵相乘，加快了训练速度

GRU 使用了更新门（update gate）与重置门（reset gate）。基本上，这两个门控向量决定了哪些信息最终能作为门控循环单元的输出。
更新门:

GRU很聪明的一点就在于，**我们使用了同一个门控就同时可以进行遗忘和选择记忆（LSTM则要使用多个门控）**

$$z_t = \sigma(W^{(z)}x_t + U^{(z)}h_{t-1})$$

其中 xt 为第 t 个时间步的输入向量，即输入序列 X 的第 t 个分量，它会经过一个线性变换（与权重矩阵 W(z) 相乘）。h(t-1) 保存的是前一个时间步 t-1 的信息，它同样也会经过一个线性变换。更新门将这两部分信息相加并投入到 Sigmoid 激活函数中，因此将激活结果压缩到 0 到 1 之间。

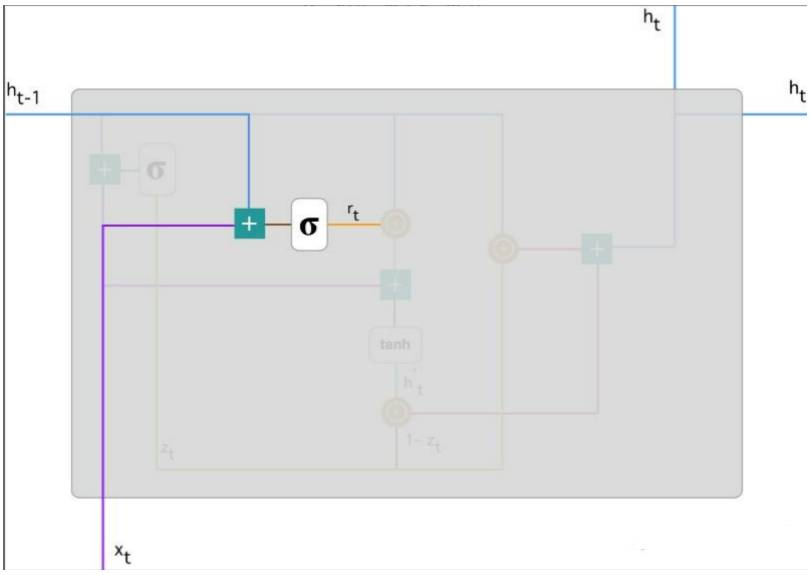


重置门:

本质上来说，重置门主要决定了到底有多少过去的信息需要遗忘

$$r_t = \sigma(W^{(r)}x_t + U^{(r)}h_{t-1})$$

该表达式与更新门的表达式是一样的，只不过线性变换的参数和用处不一样而已。




```
def GRU(max_input_len, max_output_len, input_vocab_size, output_vocab_size,
latent_dim):
    encoder_inputs = tf.keras.layers.Input(shape=(max_input_len,))
    encoder_embedding = tf.keras.layers.Embedding(input_dim=input_vocab_size,
output_dim=latent_dim)(encoder_inputs)
    encoder_gru = tf.keras.layers.GRU(latent_dim, return_state=True)
    encoder_outputs, state_h = encoder_gru(encoder_embedding)
    encoder_states = [state_h]

    decoder_inputs = tf.keras.layers.Input(shape=(max_output_len,))
```

```
    decoder_embedding = tf.keras.layers.Embedding(input_dim=output_vocab_size,
output_dim = latent_dim)(decoder_inputs)
    decoder_gru = tf.keras.layers.GRU(latent_dim, return_sequences=True,
return_state=True)
    decoder_outputs, _ = decoder_gru(decoder_embedding,
initial_state=encoder_states)
    decoder_dense = tf.keras.layers.Dense(output_vocab_size,
activation='softmax')
    decoder_outputs = decoder_dense(decoder_outputs)
    model = tf.keras.models.Model([encoder_inputs, decoder_inputs],
decoder_outputs)
    model.compile(optimizer='adam', loss='sparse_categorical_crossentropy')
    return model
```

评估标准：BLEU score评估模型

BLUE将机器翻译的结果与其相对应的几个参考翻译作比较，算出一个综合分数。这个分数越高说明机器翻译得越好。

$$BLEU = BP * exp(\frac{1}{n}\sum_{i=1}^{N} P_n)$$

$$BP = \begin{cases} 1, \\ exp(1 - MT\,outputlength/reference outputlength) \end{cases}$$

$$P_n = \frac{\sum_{n-gram \in y} \text{CounterClip(n-gram)}}{\sum_{n-gram \in y} \text{Counter(n-gram)}}$$

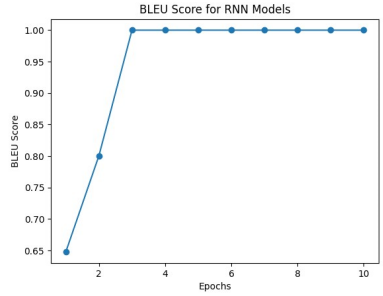
BP是简短惩罚因子，Pn是基于n-gram的精确度

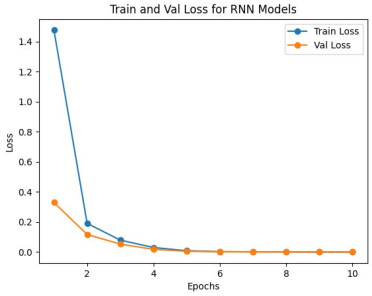
```
def compute_bleu_score(predicted_sequence, target_sequence, index_to_word):
    predicted_sequence = np.argmax(predicted_sequence, axis=-1)

    # 将预测的序列和目标序列转换为单词序列
    predicted_text = sequence_to_text(predicted_sequence[0], index_to_word)
    target_text = sequence_to_text(target_sequence[0], index_to_word)

    # 计算 BLEU 分数
    bleu_score = sentence_bleu([target_text.split()], predicted_text.split())

    return bleu_score
```

实验结果：

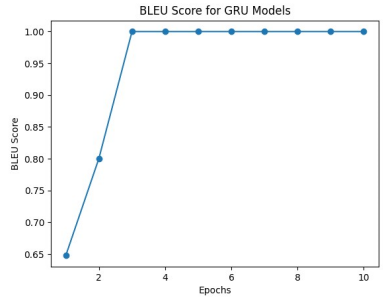
（实验结果图选取了训练轮数为10轮，因为基本在4-5轮以后bleu已经达到1.0，所以没有继续加长训练时间的必要）

RNN的bleu和损失图：

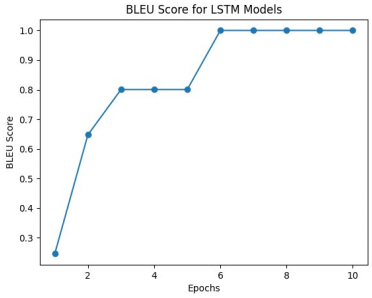


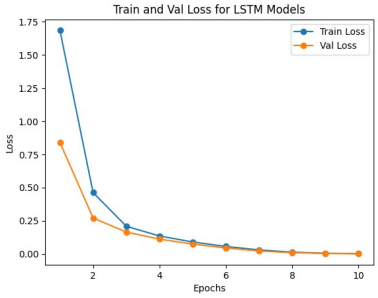


GRU的bleu和损失图

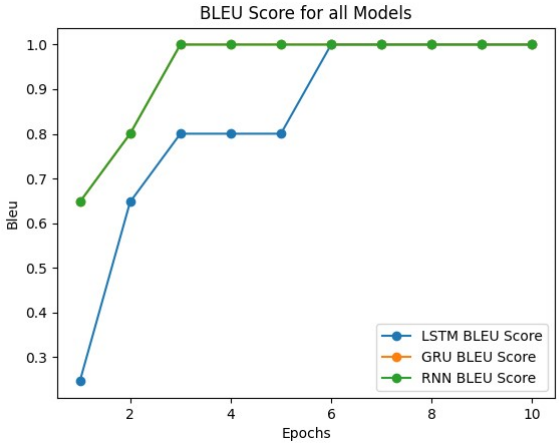


LSTM的bleu和损失图：





三种模型bleu总图：（其中GRU和LSTM图像重合）



可以看到GRU和RNN在第三轮的时候bleu几乎接近于1.0并保持不变，在第六轮的时候loss无限接近于0几乎不变，而LSTM在第六轮的时候bleu几乎接近于1.0并保持不变，在第9轮的时候loss无限接近与0并保持不变，整体训练的效果还是很不错的

至于RNN和GRU重叠的问题，我认为可能是模型参数选择，如果在定义 SimpleRNN 和 LSTM 时使用了相同的参数设置，例如相同的隐藏状态维度、相同的层数等，那么这两种模型的表达能力和学习能力可能会非常接近。在这种情况下，它们的结果和指标也可能是相同的

遇到的问题

1. 数据处理：本次数据集已脱敏，理论上讲不需要进行文本处理，即词汇表映射索引操作，但后续在模型训练和bleu计算的时候总是出现输入形状不正确问题，上网查和同学讨论之后决定使用多一步文本处理方便后面数据的使用。
2. 尝试gpu安装并且实现bart、bert等模型，但在loss极低的情况下bleu仍接近于0 可能还是传入数据格式的问题不匹配。

   BLEU Score: 7.664825495825866e-157

3. 已经在model.predict中添加use_multiprocessing = True参数，但是好像还是会启动使用gpu（我的内存超小），导致内存溢出报错，所以在开头添加`os.environ['CUDA_VISIBLE_DEVICES'] = '-1'`保证使用cpu。
4. 发现batch_size大小是可以影响最后的bleu指数的，我在使用batch_size为4的时候第一轮训练就可以得到比较好的bleu指标和训练结果，但效率很低，所以我增加batch_size到32，其实也可以更大，应该能从图上看到更好的不断收敛效果，但是考虑时间和配置性能没有尝试