Final Project

(2023 学年春季学期)

课程名称：人工神经网络 **批改人：**

实验	中-英机器翻译	专业（方向）	计算机科学与技术
学号	21307244	姓名	钟宇
Email	tue2dayzz@gmail.com	完成日期	2024.6.27

一、实验任务

本实验的目标是构建一个中-英机器翻译系统，采用 Seq2Seq 模型，并实现 Attention 机制。具体任务如下：

1. **数据集简介**
   - 数据集包含 4 个 `jsonl` 文件，分别为小训练集、大训练集、验证集和测试集。每个文件包含一定数量的平行语料样本。
   - 若计算设备受限，可仅使用小训练集进行训练；鼓励使用大训练集。
2. **数据预处理**
   - 数据清洗：过滤非法字符和稀少字词，截断过长句子。
   - 分词：将句子切分为 *tokens*。英文可用 NLTK 或 BPE 等方法，中文可用 Jieba 或 HanLP。
   - 构建词典：利用分词结果构建统计词典，过滤低频词语。
   - 建议用预训练词向量初始化，并允许在训练过程中更新。
3. **NMT 模型**
   - 构建基于 GRU 或 LSTM 的 Seq2Seq 模型，编码器和解码器各 2 层，单向。
   - 实现 Attention 机制，探索不同对齐函数（`dot product`，`multiplicative`，`additive`）的影响。
4. **训练和推理**
   - 定义损失函数（如交叉熵损失）和优化器（如 Adam）。
   - 处理双语平行语料库，训练模型的中译英能力。
   - 对比 Teacher Forcing 和 Free Running 策略的效果。
   - 对比 greedy 和 beam-search 解码策略。

5. **评估指标**
    - 使用 BLEU 评估翻译性能。

二、实验环境

- 操作系统：Ubuntu 22.04
- 编程语言：Python
- 深度学习框架：PyTorch
- 数据集：`data_short/nmt/en-cn`（由于原数据集句子较长且词语生僻，实验更换了文本更短的数据集）

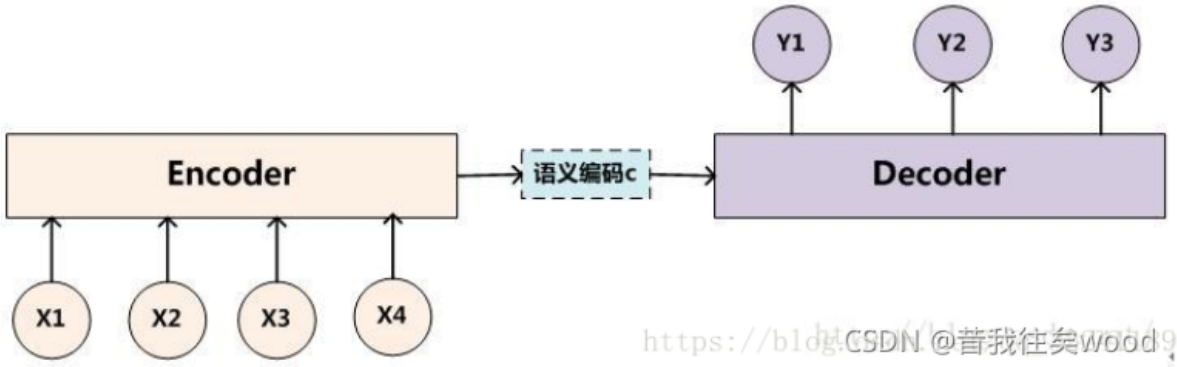
三、模型简介

Seq2Seq

Sequence2Sequence（Seq2Seq）模型是一种常用于序列到序列任务（如机器翻译、文本摘要和对话生成）的深度学习模型。其基本结构由两个主要部分组成：

- 编码器（Encoder）
- 解码器（Decoder）。

编码器将输入序列编码成一个固定大小的上下文向量（Context Vector），然后解码器根据这个上下文向量生成目标序列。



Encoder

编码器由一系列递归神经网络（RNN）单元组成，如 GRU（门控循环单元）或 LSTM（长短期记忆网络）。编码器接收输入序列，将其逐步编码成隐状态向量。最终的隐状态向量包含了整个输入序列的信息。

- **输入序列**：$x = (x_1, x_2, \ldots, x_T)$
- **隐状态**：$h_t = \text{EncoderRNN}(x_t, h_{t-1})$
- **最终隐状态**：h_T（即上下文向量）

Decoder

解码器也是由一系列 RNN 单元组成，负责生成目标序列。解码器在每个时间步接收前一个时间步的输出和编码器的上下文向量，并生成下一个时间步的输出。

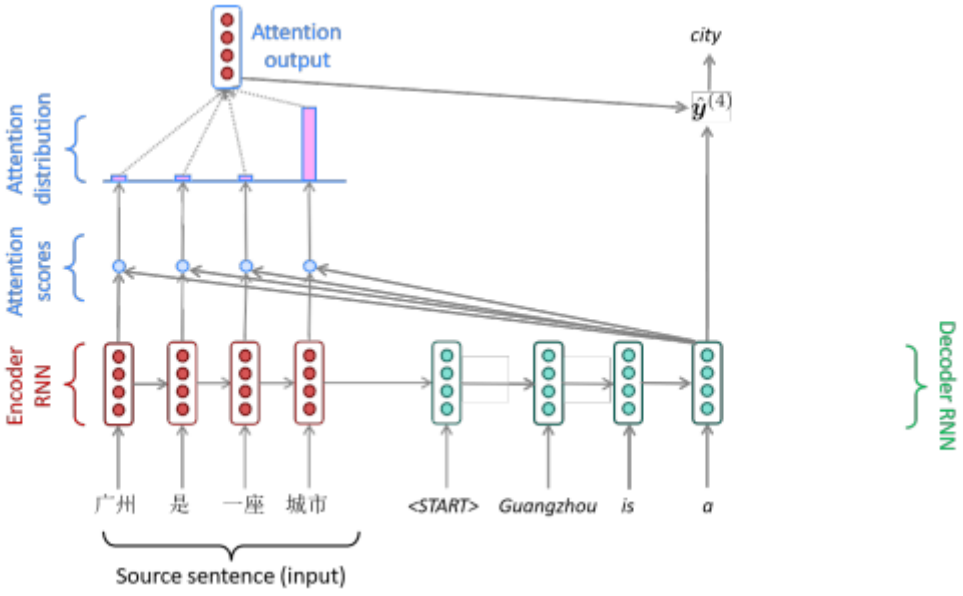
- **初始隐状态**：由编码器的最终隐状态提供，即 $s_0 = h_T$
- **解码步骤**：$s_t = \text{DecoderRNN}(y_{t-1}, s_{t-1}, c)$
- **输出序列**：$y = (y_1, y_2, \ldots, y_T)$

Attention

为了改善 Seq2Seq 模型在长序列上的表现，引入了 Attention 机制。Attention 机制允许解码器在生成每个时间步的输出时，动态选择和关注输入序列的不同部分。这不仅提升了翻译的准确性，还增强了模型的可解释性。

- **注意力权重**：$a_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})}$，其中 $e_{ij} = \text{score}(s_{i-1}, h_j)$
- **上下文向量**：$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j$
- **解码步骤**：$s_i = \text{DecoderRNN}(y_{i-1}, s_{i-1}, c_i)$

Attention 机制的对齐函数（`score`函数）可以是`dot-product`、`multiplicative`或`additive`



Dot-product Attention

Dot-product attention 是最简单、计算效率最高的方法，它直接计算解码器隐状态和编码器隐状态的点积。

公式

$$e_{ij} = s_{i-1}^{\top} h_j$$

其中，e_{ij} 是解码器第 i个时间步的隐状态 s_{i-1} 与编码器第 j 个时间步的隐状态 h_j 的点积

计算步骤

1. **计算 score**：

   对于解码器的隐状态 s_{i-1} 和编码器的隐状态 h_j，计算它们的点积：

   $$e_{ij} = s_{i-1}^{\top} h_j$$

2. **计算注意力权重**：

   使用 `softmax` 函数将 score 转换为权重：

   $$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})}$$

3. **计算上下文向量**：

   对编码器隐状态进行加权求和得到上下文向量：

   $$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j$$

Multiplicative (Scaled Dot-product) Attention

Multiplicative attention 也被称为 Scaled Dot-product Attention，它在计算点积前对解码器隐状态进行线性变换，并对点积结果进行缩放。

公式

$$e_{ij} = \frac{(W_q s_{i-1})^{\top} (W_k h_j)}{\sqrt{d}}$$

其中，W_q和 W_k 是可学习的权重矩阵，d 是隐状态向量的维度，用于缩放。

计算步骤

1. **线性变换**：

   对解码器隐状态 s_{i-1} 和编码器隐状态 h_j 进行线性变换：

   - $q_i = W_q s_{i-1}$
   - $k_j = W_k h_j$

2. **计算score**：

   计算线性变换后的向量的点积并进行缩放：

   $$e_{ij} = \frac{q_i^{\top} k_j}{\sqrt{d}}$$

3. **计算注意力权重**：

   使用 `softmax` 函数将 score 转换为权重：

   $$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})}$$

4. **计算上下文向量**：

对编码器隐状态进行加权求和得到上下文向量：

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j$$

Additive (Bahdanau) Attention

$\text{Additive attention}$ 由 Bahdanau 等人提出，它通过一个单层前馈神经网络将解码器隐状态和编码器隐状态组合起来计算 score。

公式

$$e_{ij} = v_a^\top \tanh(W_a[s_{i-1}; h_j])$$

其中，W_a 是可学习的权重矩阵，v_a 是可学习的权重向量，$[s_{i-1}; h_j]$ 表示解码器隐状态和编码器隐状态的连接

计算步骤

1. **连接隐状态**：

   将解码器隐状态 s_{i-1} 和编码器隐状态 h_j 连接：

   $$[s_{i-1}; h_j]$$

2. **线性变换和激活函数**：

   对连接后的向量进行线性变换并应用 `tanh` 激活函数：

   $$e_{ij} = \tanh(W_a[s_{i-1}; h_j])$$

3. **计算score**：

   对线性变换后的向量进行点积：

   $$e_{ij} = v_a^\top e_{ij}$$

4. **计算注意力权重**：

   使用 `softmax` 函数将 score 转换为权重：

   $$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})}$$

5. **计算上下文向量**：

   对编码器隐状态进行加权求和得到上下文向量：

   $$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j$$

- **Dot-product Attention**：简单且计算效率高，但在向量维度大时可能导致点积值过大，需注意数值稳定性。
- **Multiplicative (Scaled Dot-product) Attention**：通过线性变换和缩放改进了 $\text{dot-product attention}$，适用于多头注意力机制。
- **Additive Attention**：通过一个前馈神经网络计算 score ，具有更强的表达能力，但计算开销较大。

四、代码实现

数据预处理

语言类 Lang

```
class Lang:
    def __init__(self, name):
        self.name = name
        # word frequency
        self.word2count = Counter()
        # Dictionary: word → index
        self.word2index = {}
        # Dictionary: index → word
        self.index2word = {}
        self.total_words = 2  # Initial n_words include "UNK" and "PAD"

    def count_word(self, sentence):
        for word in sentence:
            self.word2count[word] += 1

    def build_dict(self, max_words):
        ls = self.word2count.most_common(max_words)
        self.total_words = len(ls) + 2

        self.word2index = {w[0]: index + 2 for index, w in enumerate(ls)}
        # w[0]:word, w[1]:word frequency
        self.word2index['UNK'] = UNK_IDX
        self.word2index['PAD'] = PAD_IDX

        self.index2word = {v: k for k, v in self.word2index.items()}
```

分析:

定义了一个名为 `Lang` 的类，用于构建和管理语言字典

- `self.word2count`：使用 `Counter` 对象初始化，用于统计单词出现的频率。
- `self.word2index`：用于将单词映射到索引的字典。
- `self.index2word`：用于将索引映射回单词的字典
- `self.total_words`：初始化为 2，表示初始的单词数目，包括
  - 未知单词（UNK）
  - 填充单词（PAD）。

`count_word` 方法接受一个句子（以单词列表形式表示），并更新 `word2count` 计数器，统计每个单词在数据集中出现的频率

`build_dict`: 方法用于构建当前语言类的语言字典，为了防止词典规模过大，词典设置了最大单词数 `max_words` 用于过滤掉出现频次较低的词语

- 首先函数根据单词出现频率（从高到低）选取最常见的 `max_words` 个单词，并存储在 `ls` 中
- 接着加上类初始化字典自带的两个特殊符号，计算字典的总单词数 `total_words`

- 最后开始构建 `word` 到 `index` 的映射与 `index` 到 `word` 的反向映射

数据清洗

```
def clean_zh(text):
    cleaned_text = re.sub(r"[^\u4e00-\u9fff0-9]", "", text)
    return cleaned_text


def clean_en(text):
    cleaned_text = re.sub(r'[^\w\s]+', '', text)
    cleaned_text = cleaned_text.replace('_', '')
    cleaned_text = cleaned_text.lower()
    return cleaned_text

def truncate_text(text):
    if len(text) >= MAX_LENGTH:
        text = text[:MAX_LENGTH - 1]
    return text
```

分析:

- `clean_zh` 函数用于清洗中文文本： 使用正则表达式去除所有不属于中文字符（包括汉字和数字）的内容
- `clean_en` 函数用于清洗英文文本：使用正则表达式去除所有非单词字符（非字母、数字和下划线），接着再手动移除所有下划线并把所有的英文转换成小写
- `truncate_text` 函数用于截断长文本：如果文本 `text` 的长度大于等于 `MAX_LENGTH`，则将文本截断为 `MAX_LENGTH - 1` 的长度

截断文本函数的作用是为了限制输入文本的长度，避免过长的文本影响后续处理或存储的效率和性能。同时通过限制文本长度，可以减少处理过程中的计算量和复杂性，从而提高处理效率和响应速度。

分词

```
def tokenize_en(text):
    return word_tokenize(text)


def tokenize_zh(text):
    return list(jieba.cut(text))
```

分析:

- `tokenize_en` 函数采用了 `nltk.tokenize` 库中的 `word_tokenize` 函数进行英文文本的分词
- `tokenize_zh` 函数采用了 `jieba` 这个流行的中文分词工具进行中文文本的分词

数据读取与词典构建

```
def read_langs(language1, language2, file_name, reverse=False):
    print("Reading lines...")

    pairs = []

    with open(file_name, 'r', encoding='utf-8') as file:
        for line in file:
            line = line.split('\t')
            zh_tokenized = tokenize_zh(clean_zh(line[1]))
            en_tokenized = tokenize_en(clean_en(line[0]))
            zh_text = ['BOS'] + truncate_text(zh_tokenized) + ['EOS']
            en_text = ['BOS'] + truncate_text(en_tokenized) + ['EOS']

            pairs.append([zh_text, en_text])

    # reverse the pair(change the translation direction)
    if reverse:
        pairs = [list(reversed(p)) for p in pairs]
        input_lang = Lang(language2)
        output_lang = Lang(language1)
    else:
        input_lang = Lang(language1)
        output_lang = Lang(language2)

    return input_lang, output_lang, pairs


def prepare_data(lang1, lang2, reverse=False):
    train_file_name = '../data_short/nmt/en-cn/train.txt'
    val_file_name = '../data_short/nmt/en-cn/dev.txt'
    test_file_name = '../data_short/nmt/en-cn/test.txt'
    input_lang, output_lang, train_pairs = read_langs(lang1, lang2,
train_file_name, reverse)
    _, _, valid_pairs = read_langs(lang1, lang2, val_file_name, reverse)
    _, _, test_pairs = read_langs(lang1, lang2, test_file_name, reverse)

    print("Read %s sentence pairs" % len(train_pairs))
    print("Counting words...")
    for pair in train_pairs:
        input_lang.count_word(pair[0])
        output_lang.count_word(pair[1])

    # Build word dictionary
    input_lang.build_dict(MAX_WORDS)
    output_lang.build_dict(MAX_WORDS)
    print("Counted words:")
    print(f"{input_lang.total_words} {input_lang.name} words")
    print(f"{output_lang.total_words} {output_lang.name} words")
    return input_lang, output_lang, train_pairs, valid_pairs, test_pairs
```

分析：

`read_langs` 函数用于数据文件的读取：对于文件中的每一行数据，分别进行中文和英文文本的清洗和分词操作

注：在机器翻译任务中，BOS 和 EOS 标记帮助模型区分句子的开始和结束。在训练阶段，模型需要知道何时开始生成翻译或者输出序列，并在生成完成后正确停止。这些标记有助于模型学习捕捉序列的起始和结束模式。因此，在对读取的每一句中英文本进行清洗、分词和截断之后，都要添加起始标记 `['BOS']` 和结束标记 `['EOS']`

最后，函数将处理后的中英文文本对 `[zh_text, en_text]` 添加到 `pairs` 列表中

- 如果 `reverse=True`，则交换每个语言对的顺序，用于改变翻译方向

`prepare_data` 函数用于训练集、验证集、测试集的构建以及中英词典的构建

首先，函数分别调用 `read_langs` 函数读取训练、验证和测试数据集，并获取对应的 `Lang` 对象。接着遍历训练数据集（`train_pairs`），记录训练集输入和输出语言中每个单词并统计其出现的频率。最后基于训练数据集中出现频率最高的单词构建单词到索引的映射。

文本编码

```
def encode(input_lang, output_lang, pairs, sort_by_len=True):
    for pair in pairs:
        pair[0] = [input_lang.word2index.get(word, 0) for word in pair[0]]
        pair[1] = [output_lang.word2index.get(word, 0) for word in pair[1]]

    # sort sentences by length
    def len_argsort(seq):
        return sorted(range(len(seq)), key=lambda x: len(seq[x][0]))

    if sort_by_len:
        sorted_indices = len_argsort(pairs)
        pairs = [pairs[i] for i in sorted_indices]

    return pairs
```

分析：

该函数用于对构建完成的文本对进行编码：对于每个语言对 `pair`，函数遍历其输入序列 `pair[0]` 和输出序列 `pair[1]` 并使用 `word2index` 字典将每个单词转换为其对应的索引。

- 如果单词不在字典中，则使用索引 `0` 表示未知单词（词典中 `0` 对应的是未知单词 UNK）

接着函数按照输入序列的长度对文本对进行**排序。**

注：对文本对进行排序在 Seq2Seq 学习任务中具有重要作用，特别是在使用批量处理（batching）和填充（padding）技术时

1. **减少填充数量**

在 Seq2Seq 学习中，由于每个批次要求输入和输出序列具有相同的长度，因此需要对较短的序列进行填充。如果未排序，较短的序列可能会与较长的序列一起形成批次，导致大量的填充操作。通过对序列按照长度进行排序，可以使得相似长度的序列聚集在一起，减少填充操作的数量。

2. **提升模型收敛速度**

   排序后的序列长度变化较小，有助于减少在训练过程中的不必要的计算和内存消耗，进而加快模型的收敛速度。相比于未排序的情况，模型能够更快地学习到有效的语言结构和翻译规律。

3. **更稳定的训练过程**

   排序后的序列长度分布更加均匀，可以减少由于序列长度差异引起的梯度爆炸或消失问题，使得训练过程更加稳定。稳定的训练过程有助于提高模型的泛化能力和效果。

数据加载器

```
def get_index_batch(n, batch_size, shuffle=True):
    idx_list = np.arange(0, n, batch_size)
    if shuffle:
        np.random.shuffle(idx_list)
    minibatch = []
    for idx in idx_list:
        minibatch.append(np.arange(idx, min(idx + batch_size, n)))
    return minibatch


# Get the information of each batch's sentence
def get_data_info(seqs):
    lengths = [len(seq) for seq in seqs]
    seq_nums = len(seqs)  # Number of sentences in the batch
    max_len = np.max(lengths)  # Max length of sentence in the batch

    x = np.zeros((seq_nums, max_len)).astype('int32')
    x_lengths = np.array(lengths).astype('int32')  # Original length of sentences
in the batch

    for idx, seq in enumerate(seqs):
        x[idx, :lengths[idx]] = seq

    return x, x_lengths


def C(pairs, batch_size):
    index_batch = get_index_batch(len(pairs), batch_size)
    data_total = []
    for batch in index_batch:
        input_sentences = [pairs[t][0] for t in batch]
        output_sentences = [pairs[t][1] for t in batch]
        b_input, b_input_len = get_data_info(input_sentences)
        b_output, b_output_len = get_data_info(output_sentences)
        data_total.append((b_input, b_input_len, b_output, b_output_len))
    # len(data_total) = n / batch_size
    # each item includes {a batch of input vectors, length of input sentences in
a batch,
    #                     a batch of output vectors, length of output sentences
in a batch}
```

```
    return data_total
```

分析：

`get_index_batch` 函数生成用于批处理的索引列表，同时对生成的批次索引列表进行 shuffle，确保每次迭代的批次顺序不同，增加训练的随机性。

`get_data_info` 函数用于对每一个批次的输入序列列表 `seqs` 进行处理，获取每个序列的长度信息并整理成批次所需的输入格式。通过计算批次中最长序列的长度，对批次中所有长度小于最大长度的序列进行填充，来保持输入序列中文本长度的统一

`get_data_inf` 函数用于生成数据加载器，加载器中每一个元素都包含

- 一个批次中所有的输入和输出文本编码向量
- 一个批次中所有输入和输出文本编码向量的长度列表

构建实例

```
input_lang, output_lang, train_pairs, val_pairs, test_pairs =
prepare_data("Chinese", "English")
train_pairs = encode(input_lang, output_lang, train_pairs)
val_pairs = encode(input_lang, output_lang, val_pairs)
test_pairs = encode(input_lang, output_lang, test_pairs)

batch_size = 64
train_data = gen_examples(train_pairs, batch_size)
val_data = gen_examples(val_pairs, batch_size)
```

分析：

- 每一个批次包含的文本数量为 64
- `train_data` 是训练集的数据加载器
- `val_data` 是验证集的数据加载器

Encoder

```
class Encoder(nn.Module):
    def __init__(self, vocab_size, embed_size, enc_hidden_size, dec_hidden_size,
num_layers=2, dropout=0.2):
        super(Encoder, self).__init__()
        self.embed = nn.Embedding(vocab_size, embed_size)
        self.rnn = nn.GRU(embed_size, enc_hidden_size, num_layers=num_layers,
batch_first=True, bidirectional=True)
        self.dropout = nn.Dropout(dropout)
        self.fc = nn.Linear(enc_hidden_size * 2, dec_hidden_size)
        self.num_layers = num_layers
        self.enc_hidden_size = enc_hidden_size

    def forward(self, x, lengths):
        sorted_len, sorted_idx = lengths.sort(0, descending=True)
```

```
        x_sorted = x[sorted_idx.long()]
        embedded = self.dropout(self.embed(x_sorted))

        packed_embedded = nn.utils.rnn.pack_padded_sequence(embedded,
sorted_len.long().cpu().data.numpy(),
                                                            batch_first=True)
        packed_out, hid = self.rnn(packed_embedded)
        out, _ = nn.utils.rnn.pad_packed_sequence(packed_out, batch_first=True)
        _, original_idx = sorted_idx.sort(0, descending=False)
        out = out[original_idx.long()].contiguous()
        hid = hid[:, original_idx.long()].contiguous()
        # hid.shape = (4, batch_size, enc_hidden_size)

        # Process all layers
        hid = hid.view(self.num_layers, 2, -1, self.enc_hidden_size)
        # (num_layers, num_directions, batch_size, hidden_size)

        # Initialize a list to store the processed hidden states for each layer
        processed_hid = []

        for layer in range(self.num_layers):
            # Concatenate forward and backward hidden states for each layer
            hid_forward = hid[layer, 0, :, :]
            hid_backward = hid[layer, 1, :, :]
            concatenated_hid = torch.cat([hid_forward, hid_backward], dim=1)
            # concatenated_hid.shape = (batch_size, 2*hidden_size)

            # Apply the fully connected layer and activation function
            processed_hid.append(torch.tanh(self.fc(concatenated_hid)))

        # Stack the processed hidden states back to the shape (num_layers,
batch_size, hidden_size)
        processed_hid = torch.stack(processed_hid, dim=0)
        # hid.shape = (num_layers, batch_size, dec_hidden_size)
        # out.shape = batch_size, seq_len, 2*enc_hidden_size)

        return out, processed_hid
```

分析：

本次实验的编码器和解码器都采用了循环神经网络 GRU 的网络架构

Encoder 采用的是**两层双向的** GRU **模型**，同时加入了防止过拟合的 Dropout 层

`forward` 函数用于模型的前向传播，具体实现如下：

- **数据处理**:
    - 根据长度 `lengths` 对输入序列 `x` 进行降序排序，以便使用 `pack_padded_sequence` 函数来处理变长序列。
    - 使用 Embedding 层将输入序列 `x_sorted` 转换为词嵌入 `embedded` 并应用 $\text{Dropout$。
    - 使用 `pack_padded_sequence` 将填充后的嵌入序列打包成紧凑的格式，以便于 GRU 处理。
    - 将打包后的嵌入序列输入到双向 GRU（`self.rnn`）中，得到打包后的输出 `packed_out` 和最终的隐藏状态 `hid`。

- **解包和恢复顺序**:
  - 使用 `pad_packed_sequence` 将打包后的输出 `packed_out` 解包成正常的张量 `out`。
  - 根据排序后的索引 `sorted_idx` 恢复 `out` 和 `hid` 的原始顺序。
- **隐藏状态处理**:
  - 将双向 GRU 的隐藏状态 `hid` 重塑为 `(num_layers, 2, -1, enc_hidden_size)`，以便按层和方向处理。
  - 为每一层的前向和后向隐藏状态连接，通过全连接层 `self.fc` 和激活函数 `torch.tanh` 处理，得到 `processed_hid`，表示每层的处理后的隐藏状态。
- **输出**:
  - `out`：表示所有时间步的编码器输出，形状为 `(batch_size, seq_len, 2*enc_hidden_size)`。
  - `processed_hid`：表示每一层处理后的隐藏状态，形状为 `(num_layers, batch_size, dec_hidden_size)`。

补充:

在处理序列数据时，通常会遇到变长序列。为了有效地处理这些序列，需要对其进行填充以保证输入的批次具有相同的长度。然而，填充后的序列会引入大量无效数据，浪费计算资源。

`pack_padded_sequence` 函数的作用就是将填充后的序列打包成紧凑的格式，以便于 RNN 处理，从而节省计算资源并提高效率。

Decoder

```
class Decoder(nn.Module):
    def __init__(self, vocab_size, embed_size, enc_hidden_size, dec_hidden_size,
num_layers=2, dropout=0.2):
        super(Decoder, self).__init__()
        self.embed = nn.Embedding(vocab_size, embed_size)
        self.attention = Attention(enc_hidden_size, dec_hidden_size)
        self.rnn = nn.GRU(embed_size, dec_hidden_size, num_layers=num_layers,
batch_first=True)
        self.out = nn.Linear(dec_hidden_size, vocab_size)
        self.dropout = nn.Dropout(dropout)

    def create_mask(self, output_len, input_len):
        # Generate a mask of shape output_len * input_len
        device = output_len.device
        max_output_len = output_len.max()
        max_input_len = input_len.max()

        output_mask = torch.arange(max_output_len, device=device)[None, :] <
output_len[:, None]
        # output_mask.shape = (batch_size, output_len) # mask of output(English)
        input_mask = torch.arange(max_input_len, device=device)[None, :] <
input_len[:, None]
        # input_mask.shape = (batch_size, context_len) # mask of input(Chinese)
```

```
        mask = (output_mask[:, :, None] & input_mask[:, None, :]).byte()
        mask = mask.bool()
        # output_mask[:, :, None].shape = (batch_size, output_len, 1)
        # input_mask[:, None, :].shape =  (batch_size, 1, context_len)
        # mask.shape = (batch_size, output_len, context_len)
        return mask

    def forward(self, encoder_context, x_lengths, y, y_lengths, hid):
        # x_lengths is the length of context vector(Chinese)
        # y_lengths is the length of standard translation vector(English)
        sorted_len, sorted_idx = y_lengths.sort(0, descending=True)
        y_sorted = y[sorted_idx.long()]
        hid = hid[:, sorted_idx.long()]

        y_sorted = self.dropout(self.embed(y_sorted))  # batch_size,
output_length, embed_size

        packed_seq = nn.utils.rnn.pack_padded_sequence(y_sorted,
sorted_len.long().cpu().data.numpy(), batch_first=True)
        out, hid = self.rnn(packed_seq, hid)
        unpacked, _ = nn.utils.rnn.pad_packed_sequence(out, batch_first=True)

        _, original_idx = sorted_idx.sort(0, descending=False)
        output_seq = unpacked[original_idx.long()].contiguous()
        hid = hid[:, original_idx.long()].contiguous()
        # hid.shape = (num_layers, batch_size, dec_hidden_size)

        mask = self.create_mask(y_lengths, x_lengths)

        output, attn = self.attention(output_seq, encoder_context, mask)
        # output.shape = (batch_size, output_len, dec_hidden_size)
        # attn.shape = (batch_size, output_len, context_len)

        # Calculate the output probability of each word
        output = F.log_softmax(self.out(output), -1)
        # output.shape = (batch_size, output_len, vocab_size)
        return output, hid, attn
```

分析：

Decoder 采用的是**两层单向的** GRU **模型**，同时加入了防止过拟合的 Dropout 层

`create_mask` 函数用于生成一个 mask，使得**在计算注意力权重时忽略填充部分**

在 Seq2Seq 任务中，输入和输出序列的长度通常是不一致的。在训练过程中，我们使用填充来确保所有序列具有相同的长度，以便进行批量处理。然而，这些填充部分并不包含有用的信息，在计算 Decoder 每一个时间步的注意力时，如果不加以处理，可能会影响模型的注意力机制和最终的预测结果。

因此，使用 mask 可以减少不必要的计算，因为填充部分不会被考虑在内。同时忽略填充部分可以防止模型在训练过程中学习到填充部分的无效信息，从而提高模型的性能和泛化能力。

需要忽略填充部分有两种：

1. Encoder 中输入的预翻译序列的填充部分：该部分不需要与 Decoder 中当前时间步的隐向量进行注意力的计算

2. Decoder 中输入的标准翻译序列的填充部分：该部分不需要与 Encoder 中的上下文向量进行注意力的计算

具体实现过程为：

首先获取输出序列和输入序列的最大长度，接着通过当前序列的长度与最大长度的对比来生成一个形状为 `(batch_size, context_len)` / `(batch_size, output_len)` 的输出 / 输入 mask ，其中**每个位置表示该位置是否在实际输出序列范围内。**

最后将输出 mask 和输入 mask 进行广播并结合，生成一个形状为 `(batch_size, output_len, context_len)` 的最终 mask 。由于后续需要使用 `masked_fill` 函数，该函数接受的输入类型为布尔值，因此还需要将 mask 转换为布尔类型。其中，每一个为 False 的位置都代表不需要进行注意力计算的填充位置

`forward` 函数用于模型的前向传播，具体实现如下：

- **数据处理**:
  - 根据长度 `lengths` 对输出序列 `y` 进行降序排序，以便使用 `pack_padded_sequence` 函数来处理变长序列。同时为了使得 `y` 与 Encoder 传入的最后一层的隐状态相匹配 ，也要以相同的顺序对 `hid` 进行排序
  - 使用 Embedding 层将输出序列 `y_sorted` 转换为词嵌入 `embedded` 并应用 Dropout。
  - 使用 `pack_padded_sequence` 将填充后的嵌入序列打包成紧凑的格式，以便于 GRU 处理。
  - 将打包后的嵌入序列输入到 GRU（`self.rnn`）中，得到打包后的输出 `packed_out` 和最终的隐藏状态 `hid`。
- **解包和恢复顺序**:
  - 使用 `pad_packed_sequence` 将打包后的输出 `packed_out` 解包成正常的张量 `out`。
  - 根据排序后的索引 `sorted_idx` 恢复 `out` 和 `hid` 的原始顺序。
- **生成Mask**:

  通过当前批次的输入和输出序列的长度向量来构建 mask 用于后续的注意力计算
- **注意力机制**:

  将解码器的结果序列、编码器的上下文向量和生成的 mask 输入到注意力层，计算注意力权重和最后的解码器输出
- **输出**:

  使用线性层将解码器的输出映射到词汇表大小的向量，并应用 `log_softmax` 函数，得到每个词的对数概率分布。这里使用 `log_softmax` 函数作为概率分布的计算函数主要是为了之后计算**交叉熵损失函数**，由于最后的翻译结果取的是概率最高的词汇表中的词语，因此取对数并不会对结果产生影响

Tips： 本部分中部分名称可能会造成混淆，在此做出解释：由于代码分析部分采用的是使用 Teacher forcing 的解码器结构，因此在训练阶段会将整个标准的翻译结果作为输入传入解码器中来引导模型的拟合。

在分析中，统一将标准的翻译结果输入称为输出序列，而解码器的输出称为结果序列，希望读者不要混淆。

Attention

Dot-product Attention

```
class Attention(nn.Module):
    def __init__(self, enc_hidden_size, dec_hidden_size):
        super(Attention, self).__init__()
        self.enc_hidden_size = enc_hidden_size
        self.dec_hidden_size = dec_hidden_size

        self.linear_key = nn.Linear(enc_hidden_size * 2, dec_hidden_size,
bias=False)
        self.linear_out = nn.Linear(enc_hidden_size * 2 + dec_hidden_size,
dec_hidden_size)

    def forward(self, output, context, mask):
        # mask.shape = (batch_size, output_len, context_len)
        # output.shape =  (batch_size, output_len, dec_hidden_size)
        # context.shape = (batch_size, context_len, 2*enc_hidden_size)

        batch_size = output.size(0)
        output_len = output.size(1)
        input_len = context.size(1)  # input_len = context_len

        # Key = W_k·context
        context_key = self.linear_key(context.view(batch_size * input_len,
-1)).view(
            batch_size, input_len, -1)

        # Attention score = Query · Key
        # Query = output, Key = context_key
        # context_key.transpose(1,2).shape = (batch_size, dec_hidden_size,
context_len)
        attn = torch.bmm(output, context_key.transpose(1, 2))
        # attn.shape = (batch_size, output_len, context_len)

        # Inhibit the effect of the specific location's attention score
        mask = mask.to(attn.device)
        attn.masked_fill(~mask, -1e6)

        # Use softmax to calculate attention weight
        attn = F.softmax(attn, dim=-1)
        # attn.shape = (batch_size, output_len, context_len)

        # Value = attention weight · context
        context = torch.bmm(attn, context)
        # context.shape = (batch_size, output_len, 2*enc_hidden_size)

        # Concatenate the context vector and the hidden states to get the context
representation
        output = torch.cat((context, output), dim=-1)
        # output.shape = (batch_size, output_len,
2*enc_hidden_size+dec_hidden_size)

        # Calculate the output
```

```
        output = output.view(batch_size * output_len, -1)
        # output.shape = (batch_size*output_len,
2*enc_hidden_size+dec_hidden_size)
        output = torch.tanh(self.linear_out(output))
        # output.shape = (batch_size*output_len, dec_hidden_size)
        output = output.view(batch_size, output_len, -1)
        # output.shape = (batch_size, output_len, dec_hidden_size)
        # attn.shape = (batch_size, output_len, context_len)
        return output, attn
```

分析：

由模型简介部分关于 Attention 的介绍可得

直接计算解码器隐状态和编码器隐状态的点积来得到注意力分数（attention score）

由我们之前在 Decoder 部分的分析可得，我们需要一个 mask 使得在计算注意力权重时忽略填充部分。因此在计算出注意力分数之后，我们需要调用 `masked_fill` 函数来屏蔽填充部分的注意力分数，该函数具体实现的方法为将填充部分的位置设置为一个很小的值$(-1e6)$，以确保在 `softmax` 之后这些位置的权重接近于零。

- 在 Decoder 部分的`create_mask` 函数中，我们得到的 mask 中每一个为 False 的位置都代表不需要进行注意力计算的填充位置，但 `masked_fill` 函数是将每一个为 True 的位置设置为一个很小的值。因此在调用 `masked_fill` 函数之前，我们需要对 mask **取反**

在这之后，将注意力分数向量通过 `softmax` 层来得到编码器中每一个时间步隐状态的注意力权重，并且通过与解码器中原来的隐状态进行加权求和得到最终的上下文向量

最后，将上下文向量和解码器输出拼接在一起得到上下文表示，并将其通过线性层和 `tanh` 激活函数计算最终输出

注：初始的Dot-product Attention 中并不会对解码器的隐状态进行线性变换，但是由于本实验使用的双向的 Encoder，因此解码器和编码器的隐状态向量维度并不匹配

- `context.shape = (batch_size, context_len, 2*enc_hidden_size)`
- `output.shape =  (batch_size, output_len, dec_hidden_size)`

因此，该部分使用额外的 `linear_key` 层对解码器的隐状态向量进行维度转换

Multiplicative Attention

```
class Attention(nn.Module):
    def __init__(self, enc_hidden_size, dec_hidden_size):
        super(Attention, self).__init__()
        self.enc_hidden_size = enc_hidden_size
        self.dec_hidden_size = dec_hidden_size

        self.linear_key = nn.Linear(enc_hidden_size * 2, dec_hidden_size,
bias=False)
        self.linear_query = nn.Linear(dec_hidden_size, dec_hidden_size,
bias=False)
        self.linear_value = nn.Linear(enc_hidden_size * 2, enc_hidden_size * 2,
bias=False)
```

```
        self.linear_out = nn.Linear(enc_hidden_size * 2 + dec_hidden_size,
dec_hidden_size)

    def forward(self, output, context, mask):
        # mask.shape = (batch_size, output_len, context_len)
        # output.shape =  (batch_size, output_len, dec_hidden_size)
        # context.shape = (batch_size, context_len, 2*enc_hidden_size)

        batch_size = output.size(0)
        output_len = output.size(1)
        input_len = context.size(1)  # input_len = context_len

        # Key = W_k·context
        context_key = self.linear_key(context.view(batch_size * input_len,
-1)).view(
            batch_size, input_len, -1)

        # Query = W_q·output
        output_query = self.linear_query(output.view(batch_size * output_len,
-1)).view(
            batch_size, output_len, -1)

        # Value = W_value * context
        context_value = self.linear_value(context.view(batch_size * input_len,
-1)).view(
            batch_size, input_len, -1)

        # Attention score = Query · Key
        # Query = output_query, Key = context_key
        # output_query.shape = (batch_size, output_len, dec_hidden_size)
        # context_key.transpose(1,2).shape = (batch_size, dec_hidden_size,
context_len)
        attn = torch.bmm(output_query, context_key.transpose(1, 2))
        # attn.shape = (batch_size, output_len, context_len)

        # Attention Normalization
        scaling_coef = torch.sqrt(torch.tensor(input_len, dtype=torch.float32))
        attn = attn * scaling_coef

        # Inhibit the effect of the specific location's attention score
        mask = mask.to(attn.device)
        attn.masked_fill(~mask, -1e6)

        # Use softmax to calculate attention weight
        attn = F.softmax(attn, dim=-1)
        # attn.shape = (batch_size, output_len, context_len)

        # Value = attention weight · context
        context = torch.bmm(attn, context_value)
        # context.shape = (batch_size, output_len, 2*enc_hidden_size)

        # Concatenate the context vector and the hidden states to get the context
representation
        output = torch.cat((context, output), dim=-1)
        # output.shape = (batch_size, output_len,
2*enc_hidden_size+dec_hidden_size)
```

```
        # Calculate the output
        output = output.view(batch_size * output_len, -1)
        # output.shape = (batch_size*output_len,
2*enc_hidden_size+dec_hidden_size)
        output = torch.tanh(self.linear_out(output))
        # output.shape = (batch_size*output_len, dec_hidden_size)
        output = output.view(batch_size, output_len, -1)
        # output.shape = (batch_size, output_len, dec_hidden_size)
        # attn.shape = (batch_size, output_len, context_len)
        return output, attn
```

分析：

由模型简介部分关于 Attention 的介绍可得

Multiplicative Attention 与 Dot-product Attention 类似，只是在计算点积前对编码器和解码器的隐状态进行线性变换，并对点积结果进行缩放。为了实现该变换，我们新增了两个线性层（`linear_key` 层为了维度匹配已经提前实现）

- `self.linear_query = nn.Linear(dec_hidden_size, dec_hidden_size, bias=False)`
- `self.linear_value = nn.Linear(enc_hidden_size * 2, enc_hidden_size * 2, bias=False)`

对应 Attention 论文中的说法，经过三个线性层，分别可以的得到 Query、Key、Value 矩阵

在将 Query 和 Key 相乘得到注意力分数之后，函数进行了**注意力分数的尺度的调节**。具体来说，函数对于注意力分数的计算，应用了以下的公式

$$\text{Attention score} = \left(\frac{QK^T}{\sqrt{d_k}}\right)$$

其中 d_k 是 Key 向量的维度，也就是 `self.dec_hidden_size`

这是为了应对点积的**数值稳定性**问题：

当查询向量 Q 和键向量 K 的维度 d_k 增加时，它们的点积 QK^T 的值可能会变得很大。这是因为点积的值是所有元素乘积之和，维度越大，总和的值范围也越大。

而当点积 QK^T 的值很大时，经过 `softmax` 函数后，输出分布可能变得非常尖锐。具体来说，某些位置的注意力权重可能会接近 1，而其他位置的权重会接近 0。这种尖锐的分布可能会导致梯度消失或梯度爆炸的问题，使得模型训练不稳定。

除以 $\sqrt{d_k}$ 之后，点积的值被缩放到一个较小的范围内，这样可以使得 `softmax` 函数的输出分布更加平滑，避免过度尖锐的分布。同时这种缩放有助于保持梯度的稳定性，从而使得模型训练更加稳定和有效。

其余部分基本与 Dot-product Attention 相同

Additive Attention

```
class Attention(nn.Module):
    def __init__(self, enc_hidden_size, dec_hidden_size):
        super(Attention, self).__init__()
        self.enc_hidden_size = enc_hidden_size
```

```
        self.dec_hidden_size = dec_hidden_size

        # Wa is the weight matrix for the linear transformation of concatenated
[s_{i-1}; h_j]
        self.Wa = nn.Linear(enc_hidden_size * 2 + dec_hidden_size,
dec_hidden_size, bias=False)
        # va is the weight vector for the final dot product
        self.va = nn.Linear(dec_hidden_size, 1, bias=False)
        self.linear_out = nn.Linear(enc_hidden_size * 2 + dec_hidden_size,
dec_hidden_size)

    def forward(self, output, context, mask):
        # output.shape = (batch_size, output_len, dec_hidden_size)
        # context.shape = (batch_size, context_len, 2*enc_hidden_size)

        batch_size = output.size(0)
        output_len = output.size(1)
        context_len = context.size(1)

        # Repeat context_len times for each output_len
        context = context.unsqueeze(1).repeat(1, output_len, 1, 1)
        # context.shape = (batch_size, output_len, context_len,
2*enc_hidden_size)

        output = output.unsqueeze(2).repeat(1, 1, context_len, 1)
        # output.shape = (batch_size, output_len, context_len, dec_hidden_size)

        # Concatenate output and context
        combined = torch.cat((output, context), dim=-1)
        # combined.shape = (batch_size, output_len, context_len, dec_hidden_size
+ 2*enc_hidden_size)

        # Flatten combined for linear transformation
        combined = combined.view(batch_size * output_len * context_len, -1)
        # combined.shape = (batch_size * output_len * context_len,
dec_hidden_size + 2*enc_hidden_size)

        # Calculate energy scores
        energy = torch.tanh(self.Wa(combined))
        # energy.shape = (batch_size * output_len * context_len, dec_hidden_size)

        # Calculate scores
        score = self.va(energy).view(batch_size, output_len, context_len)
        # score.shape = (batch_size, output_len, context_len)

        # Apply mask
        mask = mask.to(score.device)
        score.masked_fill(~mask, -1e9)

        # Calculate attention weights
        attn_weights = F.softmax(score, dim=-1)
        # attn_weights.shape = (batch_size, output_len, context_len)

        # Calculate context vectors
        context_vectors = torch.bmm(attn_weights, context.squeeze(1))
        # context_vectors.shape = (batch_size, output_len, 2*enc_hidden_size)
```

```
        # Concatenate the context vectors with the original output
        output = torch.cat((context_vectors, output.squeeze(2)), dim=-1)
        # output.shape = (batch_size, output_len, 2*enc_hidden_size +
dec_hidden_size)

        # Apply a linear layer and tanh activation to get the final output
        output = torch.tanh(self.linear_out(output))
        # output.shape = (batch_size, output_len, dec_hidden_size)

        return output, attn_weights
```

分析：

由模型简介部分关于 Attention 的介绍可得

Additive Attention 将解码器隐状态 s_{i-1} 和编码器隐状态 h_j 连接，为了实现连接，函数分别对解码器隐状态 `context` 和编码器隐状态 `output` 进行扩展并重复使得二者的维度相匹配，接着将连接后的张量 `combine` 传入进线性层，通过 `Wa` 线性变换和 `tanh` 激活函数计算能量，再将计算得出的能力通过 `va` 线性变换计算注意力分数，然后将其恢复到 `(batch_size, output_len, context_len)` 的形状。

在这之后，我们再次将一个 mask 应用到注意力得分上，使得在计算注意力权重时忽略填充部分，并且通过 `softmax` 对得分进行归一化，得到注意力权重

最后，函数通过注意力权重对上下文进行加权求和，得到上下文向量，将上下文向量和解码器输出拼接在一起得到上下文表示，并将其通过线性层和 `tanh` 激活函数计算最终输出

Seq2Seq

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

分析：

Seq2Seq 模型是结合了 Encoder 模块和 Decoder 模块的最终模型

Teacher Forcing

Strategy

Teacher Forcing 是一种常用的训练策略。在训练过程中，解码器在**每一步的输入是实际的目标序列中的词**，而不是解码器在前一步的预测结果。通过这种方式，模型能够更快地学习到输入序列与输出序列之间的映射关系。

优点：

1. **收敛速度快**：因为使用了真实的目标序列作为输入，模型能够更快地学习到序列之间的映射关系，损失下降更快。

2. **稳定性高**：减少了由于解码器早期预测不准确导致的误差传播，训练过程更稳定。

缺点：

1. **训练和推理不一致**：在实际推理（测试）阶段，解码器需要使用自己的预测结果作为下一步的输入，这与训练过程中使用真实目标序列作为输入的方法不一致，可能导致性能下降。
2. **过拟合风险**：模型可能过度依赖训练过程中的真实输入，导致在实际推理时表现不佳。

Code

```
    # Teacher Forcing
    def forward(self, enc_input, enc_input_lengths, dec_input,
dec_input_lengths):
        encoder_context, hid = self.encoder(enc_input, enc_input_lengths)
        output, hid, attn = self.decoder(encoder_context=encoder_context,
                                         x_lengths=enc_input_lengths,
                                         y=dec_input,
                                         y_lengths=dec_input_lengths,
                                         hid=hid)
        # output.shape =(batch_size, output_len, vocab_size)
        # hid.shape = (num_layers, batch_size, dec_hidden_size)
        # attn.shape = (batch_size, output_len, context_len)
        return output, attn
```

分析:

首先使用编码器对 `enc_input` 进行编码，得到编码器上下文 `encoder_context` 和隐藏状态 `hid`。

接着使用解码器对编码器上下文和解码器输入进行解码，得到输出 `output`、隐藏状态 `hid` 和注意力权重 `attn`

在解码的过程中，传入给解码器的 `dec_input` 是一个 batch 的目标序列的词编码向量

Free Running

Stragedy

Free Running 是一种在训练过程中逐渐减少 Teacher Forcing 比例的方法。在这种策略下，随着训练的进行，逐渐增加**解码器使用自身预测结果作为下一步输入**的概率。最终目标是在训练结束时，解码器能够完全依赖自己的预测结果进行序列生成。

优点：

1. **训练和推理一致**：Free Running 方法通过逐渐减少 Teacher Forcing，使得模型在训练和推理时的输入方式一致，提高了模型在实际推理时的性能。
2. **增强鲁棒性**：模型在训练过程中逐渐适应使用自身预测结果作为输入，能够更好地处理推理过程中可能出现的错误传播问题。

缺点：

1. **收敛速度慢**：由于模型需要逐步适应使用自身预测结果作为输入，训练过程可能比完全使用 Teacher Forcing 更慢，损失下降速度较慢。
2. **训练过程复杂**：需要额外的机制来逐步调整 Teacher Forcing 的比例，使得训练过程更加复杂。

code

```
    # Free Running
    def free_running(self, x, x_lengths, y, max_length):
        encoder_out, hid = self.encoder(x, x_lengths)
        output_by_step = []
        batch_size = x.shape[0]
        y = y.repeat(batch_size, 1)
        attns = []
        for i in range(max_length):
            output, hid, attn = self.decoder(encoder_out,
                                             x_lengths,
                                             y,

torch.ones(batch_size).long().to(y.device),
                                             hid)
            y = output.max(2)[1].view(batch_size, 1)
            output_by_step.append(output)
            attns.append(attn)
        return torch.cat(output_by_step, 1), torch.cat(attns, 1)
```

分析:

Free Running 在训练过程的开始依旧会使用 Teacher Forcing 的训练策略，但是随着训练轮次的加深，会慢慢地使用当前时间步自身的预测结果作为下一时间步的输入，具体逻辑见 Train 函数。本部分实现的是使用当前时间步自身的预测结果作为下一时间步的输入的训练预测函数

首先还是使用编码器对 `enc_input` 进行编码，得到编码器上下文 `encoder_context` 和隐藏状态 `hid`

但对于解码器的输入，此时我们不能使用完整的目标序列比编码向量，而是要传入起始符号 BOS 所对应的词索引，接着重复将上一步解码器预测的结果作为下一时间步的输入。

因此我们将传入的起始符号 BOS 所对应的词索引 `y` 复制了 `batch_size` 份作为一整个 batch 的起始输入并作为参数传给解码器，接着对于解码器的输出概率向量，我们选择概率最高的位置对应的词索引作为当前时间步的输出并赋给 `y` 作为下一个时间步解码器的输入

最后，将每一个时间步的输出与注意力权重相连接便得到了一整个 batch 每一个输入序列的预测翻译输出以及注意力权重

Seq2Seq.translate

Greedy Search

Algorithm

Greedy Search 是一种简单而直接的序列生成算法。它在每个时间步**选择具有最高预测概率**的单个词作为当前序列的下一个词

Code

```
    def translate(self, x, x_lengths, y, max_length=100):
        encoder_out, hid = self.encoder(x, x_lengths)
        preds = []
        batch_size = x.shape[0]
        y = y.repeat(batch_size, 1)
        attns = []
        for i in range(max_length):
            output, hid, attn = self.decoder(encoder_out,
                                             x_lengths,
                                             y,

torch.ones(batch_size).long().to(y.device),
                                             hid)
            y = output.max(2)[1].view(batch_size, 1)
            preds.append(y)
            attns.append(attn)

        return torch.cat(preds, 1), torch.cat(attns, 1)
```

分析:

由于采用的是 Teacher Forcing 的训练方法，因此在解码阶段我们的输入是标准的翻译结果序列，但是在翻译阶段，编码器的输入是模型自身在上一个时间步生成的输出（**自回归方式**）。即模型在时间步 t 生成的词作为时间步 $t+1$ 的输入。模型根据先前生成的词预测下一个词，直到生成特定的终止符号（EOS）或达到最大生成长度

因此在翻译阶段，首先还是使用编码器对 `x` 进行编码，得到编码器输出 `encoder_out` 和隐藏状态 `hid`。但在解码部分，对于每个时间步，我们需要从解码器的输出中获取当前时间步的预测 `y`，并将其作为下一个时间步的输入。对于当前时间步的预测 `y`，我们通过取输出向量中概率值最大的位置作为预测的词索引来作为 `y`

最后，将预测结果和注意力权重沿时间步维度拼接，返回最终的翻译结果和注意力权重

Beam Search

Algorithm

Beam Search 是一种更加复杂和全局化的序列生成算法，旨在通过维护多个可能的候选序列来优化输出序列的质量。

算法流程:

在每个时间步，对于每个候选序列:

- 将当前序列和隐藏状态传递给解码器。
- 解码器生成输出分布，表示每个词的预测概率。
- 根据预测概率选择 top `beam_width` 个词作为当前时间步的候选。
- 对于每个新生成的候选序列，计算其累计分数（通常是当前序列的分数加上当前时间步的负对数概率）。
- 维护累计分数最高的 top `beam_width` 个候选序列作为下一步的输入。

终止条件：

- 当所有候选序列的最后一个词是结束符号（如 `<eos>`）时，停止生成。
- 或者达到预定义的最大生成长度时，停止生成。

Code

```
    def translate(self, x, x_lengths, start_token, end_token, beam_width=3,
max_length=50):
        encoder_out, hid = self.encoder(x, x_lengths)

        # Initialize the beams
        start_tensor = torch.full((1, 1), start_token.item(),
dtype=torch.long).to(x.device)
        beams = [(start_tensor, torch.tensor(0.0), hid, [])]
        completed_sequences = []

        for _ in range(max_length):
            new_beams = []
            for seq, score, hid, attn_weights in beams:
                y = seq[:, -1].view(1, 1)
                output, hid, attn = self.decoder(encoder_out,
                                                 x_lengths,
                                                 y,

torch.ones(1).long().to(y.device),
                                                 hid)

                # Get the top beam_width predictions
                topk_log_probs, topk_indices = torch.topk(F.log_softmax(output[:,
-1, :], dim=-1), beam_width)

                for i in range(beam_width):
                    new_seq = torch.cat([seq, topk_indices[:, i].view(1, 1)],
dim=-1)
                    new_score = score + topk_log_probs[:, i].item()
                    new_attn_weights = attn_weights +
[attn.squeeze().cpu().detach().numpy()]
                    new_beams.append((new_seq, new_score, hid, new_attn_weights))

            # Sort the new beams and keep the top beam_width ones
            new_beams = sorted(new_beams, key=lambda x: x[1].item(),
reverse=True)
            beams = new_beams[:beam_width]

            # Check if any of the beams has reached the end token
            indexes_to_remove = []
            for i, (seq, score, hid, attn_weights) in enumerate(beams):
                if seq[0, -1].item() == end_token.item():
                    completed_sequences.append((seq, score, attn_weights))
                    indexes_to_remove.append(i)

            for index in reversed(indexes_to_remove):
                del beams[index]
```

```
            if len(beams) == 0:
                break

        # If no sequence has reached the end token, use the best sequence in
beams
        if not completed_sequences:
            completed_sequences = beams

        # Choose the sequence with the highest score
        best_seq, best_score, best_attn_weights = max(completed_sequences,
key=lambda x: x[1].item())
        return best_seq[0, 1:], best_attn_weights
        # best_attn_weights.shape = (output_len, context_len)
```

分析:

在 Beam Search 中，首先还是使用编码器对 `x` 进行编码，得到编码器输出 `encoder_out` 和隐藏状态 `hid`。

在解码部分中，函数

- 首先初始化一个单一的 beams，包含起始符号 `start_token` 和初始分数 `0.0`
- 接着在每个时间步，根据当前 beams 中的每一个序列上一步生成的预测词生成下一个预测词，计算新的分数，并扩展候选序列（beams 中每一个序列都会生成 `bema_width` 个新序列）
- 使用 `torch.topk` 获取分数最高的 `beam_width` 个序列作为新的 beams 序列集合

结束标记检测:

- 检查每个时间步生成的序列是否以 `end_token` 结束，如果是，则将该序列加入 `completed_sequences` 中
- 如果到达最大生成长度仍未出现任何生成结束标记的序列，则返回当前 beams 中分数最高的序列

若 `completed_sequences` 中存在含有终止符号的序列，则选取其中分数最高的序列作为预测的翻译序列

由于 Decoder 的预测输出中不应该包含起始符号 EOS，因此在返回结果时需要去除序列中的第一个起始符号

注：由算法原理可得，实际上 Beam Search 的代码以及包含了 Greedy Search 的情况（将 `beam_width` 设置为 1）

Loss Criterion

```
# masked cross entropy loss
class LanguageModelCriterion(nn.Module):
    def __init__(self):
        super(LanguageModelCriterion, self).__init__()

    def forward(self, input, target, mask):
        # input.shape = (batch_size, seq_len, vocab_size)
        # target.shape = mask.shape = (batch_size, seq_len)
```

```
        input = input.contiguous().view(-1, input.size(2))
        target = target.contiguous().view(-1, 1)
        mask = mask.contiguous().view(-1, 1)

        # Calculate cross-entropy(decoder use log_softmax as classified unction)
        output = -input.gather(1, target) * mask
        # why mask? Eliminate the effect of target's padding element "0"

        output = torch.sum(output) / torch.sum(mask)
        return output
```

分析：

该类用于定义带掩码的交叉熵损失函数

多类交叉熵损失函数：

给定一个样本的预测概率分布 $\mathbf{p}$ 和真实的类别分布 $\mathbf{q}$，其中：

- $\mathbf{p} = (p_1, p_2, \ldots, p_C)$ 是预测的概率分布，表示模型预测属于每个类别的概率。
- $\mathbf{q} = (q_1, q_2, \ldots, q_C)$ 是真实的类别分布，（one-hot 向量）。

交叉熵损失的公式为：

$$\text{CrossEntropyLoss} = -\sum_{i=1}^{C} q_i \log(p_i)$$

由于在 Decoder 中使用的是 `log_softmax` 函数，因此计算输入和标准翻译序列的交叉熵损失，只需要使用 `gather` 方法从 `input` 中提取目标单词的预测概率便可得到每一个单词预测的损失值

由于此时的输入和标准翻译序列仍然也存在填充值，因此也需要通过掩码 mask 来忽略填充标记的损失，使得损失计算只基于真实的单词。

最后将损失进行求和，并除以掩码的总和（即有效词的数量）便可得到平均损失

Train Process

```
def train(model, data, start_prob, end_prob, num_epochs=100):
    training_losses = []
    validation_losses = []

    for epoch in range(num_epochs):
        model.train()
        total_num_words = total_loss = 0.

        # Calculate the probability of using the target sequence (Teacher
Forcing)
        TF = True
        teacher_forcing_prob = start_prob - (start_prob - end_prob) * (epoch /
num_epochs)
        random_prob = random.random()
        if random_prob > teacher_forcing_prob:
            TF = False

        for iteration, (b_x, b_x_len, b_y, b_y_len) in enumerate(data):
```

```
            b_x = torch.from_numpy(b_x).to(device).long()
            b_x_len = torch.from_numpy(b_x_len).to(device).long()

            dec_b_input = torch.from_numpy(b_y[:, :-1]).to(device).long()  #
Before EOS
            # In training mode, decoder's input contains no "EOS"
            dec_b_output = torch.from_numpy(b_y[:, 1:]).to(device).long()  #
After BOS
            # In training mode, standard decoder's output contains no "BOS"

            b_y_len = torch.from_numpy(b_y_len - 1).to(device).long()
            b_y_len[b_y_len <= 0] = 1

            if TF:
                # Teacher Forcing
                b_pred, attn = model(b_x, b_x_len, dec_b_input, b_y_len)
            else:
                bos =
torch.Tensor([[output_lang.word2index["BOS"]]]).long().to(device)
                # Free Running
                b_pred, attn = model.translate_greedy(b_x, b_x_len, bos,
dec_b_output.size(1))

            # Generate mask between prediction and standard output
            b_out_mask = torch.arange(b_y_len.max().item(), device=device)[None,
:] < b_y_len[:, None]
            b_out_mask = b_out_mask.float()

            # Calculate cross-entropy
            loss = loss_fn(b_pred, dec_b_output, b_out_mask)

            num_words = torch.sum(b_y_len).item()
            total_loss += loss.item() * num_words
            total_num_words += num_words

            # Update model
            optimizer.zero_grad()
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), 5.)
            optimizer.step()

            if iteration % 100 == 0:
                print("Epoch: ", epoch, 'iteration ', iteration, 'loss: ',
loss.item())

        avg_training_loss = total_loss / total_num_words
        training_losses.append(avg_training_loss)
        print("Epoch: ", epoch, "Training loss: ", avg_training_loss)

        if epoch % 10 == 0:
            val_loss = evaluate(model, val_data)
            validation_losses.append(val_loss)
        else:
            validation_losses.append(None)  # Placeholder for plotting

    # torch.save(model.state_dict(), '../Checkpoint/translate_model_TF.pt')
```

```
    torch.save(model.state_dict(), '../Checkpoint/translate_model_FR.pt')

    # Plotting the loss curve
    plt.figure(figsize=(10, 5))
    plt.plot(training_losses, label='Training Loss')
    plt.plot([i for i in range(num_epochs) if i % 10 == 0],
             [x for x in validation_losses if x is not None], label='Validation
Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.title('Training and Validation Loss Over Epochs')
    plt.show()
```

分析：

`train` 函数用于训练 Seq2Seq 模型。通过传入不同的参数，可以分别实现两种不同的训练策略

1. Free Running：`start_prob=1.0, end_prob=0.5`

   当使用 Free Running 的训练策略时，`teacher_forcing_prob` 计算在每个 epoch 中使用真实目标序列的概率，这个概率从 1.0 逐渐减少到 0.5

   在训练循环中，使用 `random.random()` 生成一个 0 到 1 之间的随机数，如果小于 `teacher_forcing_prob`，则使用真实的目标序列作为输入（Teacher Forcing）；否则，使用解码器的上一时间步的预测作为输入（Free Running）。

2. Teacher Forcing：`start_prob=1.0, end_prob=1.0`

   当使用Teacher Forcing的训练策略时，`teacher_forcing_prob` 也计算在每个 epoch 中使用真实目标序列的概率，但这个概率一直都是 1.0，即每一轮训练，，模型都是使用真实的目标序列作为解码器的输入

具体来说在每一个训练轮次当中：

- 首先从数据加载器中获取每一个 batch 的输入输出序列的文本编码向量以及其长度向量，并将其转换为 PyTorch 张量并移动到指定设备
- 接着对输入到 Decoder 中的文本向量以及标准翻译的目标向量进行改变以适配模型
  - 删除输入到 Decoder 中的文本向量中的终止标识符 EOS：因为终止标识符 EOS 是我们期望在最后一个时间步将输入文本的最后一个词向量传给 Decoder 后的预测结果，而不需要将其作为输入传入解码器
  - 删除标准翻译的目标向量中的起始标识符 BOS：因为 Decoder 第一个时间步的输出是在输入标识符 BOS 后的预测单词，因此在 Decoder 输出的翻译向量中不会包含起始标识符 BOS
- 在此之后，我们根据 `TF` 的布尔值来判断此时是使用何种策略
  - `TF = True`：使用 Teacher Forcing 的策略，将从数据加载器中读取并修改过的的目标序列传入 Seq2Seq 模型进行训练，获取模型前向传播预测的输出 `b_pred` 和注意力权重 `attn`
  - `TF = False`：使用 Free Running 的策略，将起始符号 BOS 作为初始输入传入 Seq2Seq 模型并调用模型中的 `free_running` 的方法获取预测的输出 `b_pred` 和注意力权重 `attn`
- 由于 `pad_packed_sequence` 函数会将模型的输出进行解包从而在之前的填充位置重新补零，而与此同时，0 在两个语言的词典之中都指向的是未知单词（UNK），因此我们在计算损失值的时候不能考虑该位置上的索引进行到单词的映射，也就是说我们仍然要为损失值的计算构建一个掩码 mask

- 在获取了模型的预测、标准的目标输出以及掩码 mask 之后，就可以调用 `loss_fn` 函数进行交叉熵损失函数的计算。由于函数计算的是平均损失，因此我们需要乘以有效的词向量数（即 mask 中的有效位之和）来获取当前批次中的总损失值。在进行每轮迭代的损失叠加之后，最后便可计算出一个 epoch 的平均损失
- 最后在获取了每一个轮次的损失值之后，通过反向传播计算梯度对模型进行参数的更新。同时，为了防止梯度爆炸的现象，我们还额外使用了 `clip_grad_norm_` 来控制梯度的范数

完整的训练结束之后，我们使用不同的训练策略训练好的模型 checkpoint 保存在对应文件中

- `translate_model_TF.pt`：Teacher Forcing
- `translate_model_FR.pt`：Free Running

同时，在每一个 epoch 结束之后，我们都会将交叉熵损失值记录下来，用于绘制损失曲线图

Validation Process

```
def evaluate(model, data):
    model.eval()
    total_num_words = total_loss = 0.

    with torch.no_grad():
        for _, (b_x, b_x_len, b_y, b_y_len) in enumerate(data):
            b_x = torch.from_numpy(b_x).to(device).long()
            b_x_len = torch.from_numpy(b_x_len).to(device).long()

            dec_b_input = torch.from_numpy(b_y[:, :-1]).to(device).long()
            dec_b_output = torch.from_numpy(b_y[:, 1:]).to(device).long()

            b_y_len = torch.from_numpy(b_y_len - 1).to(device).long()
            b_y_len[b_y_len <= 0] = 1

            b_pred, attn = model(b_x, b_x_len, dec_b_input, b_y_len)

            b_out_mask = torch.arange(b_y_len.max().item(), device=device)[None,
:] < b_y_len[:, None]
            b_out_mask = b_out_mask.float()

            loss = loss_fn(b_pred, dec_b_output, b_out_mask)

            num_words = torch.sum(b_y_len).item()
            total_loss += loss.item() * num_words
            total_num_words += num_words

    avg_val_loss = total_loss / total_num_words
    print("Validation loss: ", avg_val_loss)
    return avg_val_loss
```

分析：

`evaluate` 函数用于在训练规定的 epoch 之后对 Seq2Seq 模型进行验证

具体实现方法与 `train` 函数类似，只是在验证阶段不需要计算梯度并反向传播对参数进行更新

Translate

```
def translate(index):
    original_input = " ".join([input_lang.index2word[i] for i in
test_pairs[index][0]])  # Original input
    standard_output = " ".join([output_lang.index2word[i] for i in
test_pairs[index][1]])  # Standard output

    # Get one sentence to translate
    x = torch.from_numpy(np.array(test_pairs[index][0]).reshape(1,
-1)).long().to(device)
    # x.shape = (1, seq_len)
    x_len = torch.from_numpy(np.array([len(test_pairs[index]
[0])])).long().to(device)
    # x_len.shape = (1, 1)
    bos = torch.Tensor([[output_lang.word2index["BOS"]]]).long().to(device)
    # bos.shape = (1, 1)

    translation, attn = model.translate(x, x_len, bos)
    # In test mode, decoder's input is "BOS"

    translation = [output_lang.index2word[i] for i in
translation.data.cpu().numpy().reshape(-1)]
    trans = []
    for word in translation:
        if word != "EOS":
            trans.append(word)
        else:
            break

    # Print result
    # print(original_input)
    # print(standard_output)
    # print(" ".join(trans))

    return trans, standard_output.split()[1:-1]
```

分析：

`translate` 函数用于在训练结束之后对 Seq2Seq 模型进行翻译测试

`translate` 函数主要还是调用 Seq2Seq 类中的`translate`方法（翻译的主要逻辑）

首先函数通过传入的 index 获取测试集中需要翻译的句子索引，接着通过 `test_pairs` 来获取待翻译的文本编码向量以及向量的长度。在此之后，由于翻译阶段是通过上一个时间步的预测输出作为下一个时间步的输入，因此函数创建了包含 BOS 标记的张量作为解码器的初始输入进行翻译

在通过 Seq2Seq 模型得到预测结果之后，通过词索引到单词的映射词典来将输出的文本编码向量转换为最终的翻译文本。

由上述分析我们可得，由于 `pad_packed_sequence` 函数解包的补零动作，最终填充部分也会被映射到对应的单词，因此函数遍历了翻译结果，直到遇到终止标识符 EOS，并将每个单词添加到 `trans` 列表中作为最终的翻译结果

BLEU Score

```
# Calculate BLEU score
def calculate_bleu_scores(test_nums):
    total_bleu_score = 0
    smoothing_function = SmoothingFunction().method1

    for i in range(test_nums):
        translated_sentence, reference_sentence = translate(i)
        bleu_score = sentence_bleu([reference_sentence], translated_sentence,
smoothing_function = smoothing_function)
        total_bleu_score += bleu_score

    average_bleu_score = total_bleu_score / test_nums
    print(f"Average BLEU score: {average_bleu_score:.4f}")
```

分析：

`calculate_bleu_scores` 函数用于计算模型在测试数据集上的平均 BLEU 分数

BLEU（Bilingual Evaluation Understudy）是一种用于评估机器翻译结果的指标，它通过比较机器翻译的输出和一个或多个参考翻译之间的相似度来评估翻译质量

BLEU 的核心思想是基于 n-gram 匹配，即计算机器翻译输出中的 n-gram 与参考翻译中的 n-gram 的重合程度。主要考虑以下几个方面：

精确度 (Precision)：

- 计算机器翻译输出中 n-gram 与参考翻译中 n-gram 的重合率。

n-gram 级别：

- 考虑不同长度的 n-gram （如 1-gram, 2-gram, 3-gram, 4-gram）来捕捉句子的局部和全局匹配情况。

惩罚机制 (Brevity Penalty, BP)：

- 为了避免机器翻译通过生成非常短的句子来提高精确度，BLEU 引入了惩罚机制，惩罚过短的翻译。

公式

1. **精确度 (Precision)：**

$$P_n = \frac{\sum_{C \in \text{Candidates}} \sum_{\text{n-gram} \in C} \text{Count}_{\text{clip}}(\text{n-gram})}{\sum_{C' \in \text{Candidates}} \sum_{\text{n-gram}' \in C'} \text{Count}(\text{n-gram}')}$$

   - $\text{Count}_{\text{clip}}(\text{n-gram})$：机器翻译输出中 n-gram 出现的次数（限制在参考翻译中该 n-gram 的最大出现次数）。
   - $\text{Count}(\text{n-gram})$：机器翻译输出中 n-gram 出现的总次数。

2. **几何平均精确度：**

$$P = \exp\left(\sum_{n=1}^{N} w_n \log P_n\right)$$

- N：考虑的 n-gram 最大长度（通常是 4）
- w_n：n-gram 权重，通常 $w_n = \frac{1}{N}$

3. **惩罚机制 (Brevity Penalty, BP)：**

$$\begin{cases} 1 & \text{if } c > r \\ \exp\left(1 - \frac{r}{c}\right) & \text{if } c \leq r \end{cases}$$

- c：机器翻译输出的总长度。
- r：参考翻译的总长度。

4. **BLEU 分数：**

$$\text{BLEU} = BP \cdot \exp\left(\sum_{n=1}^{N} w_n \log P_n\right)$$

本函数中直接调用 `nltk.translate.bleu_score` 中的 `sentence_bleu` 函数来计算模型的预测输出和标准翻译的目标序列之间的翻译精确度。

`smoothing_function` 则是使用了 `SmoothingFunction` 类的 `method1` 方法，提供**平滑处理**以应对短句子的 BLEU 分数计算

Attention Visualization

```
# Attention Visualization
def plot_attention(attn_weights, input_sequence, output_sequence):
    # Set font to support Chinese characters
    plt.rcParams['font.sans-serif'] = ['SimHei']  # Example: SimHei is a Chinese
font, replace with appropriate font for your system
    plt.rcParams['axes.unicode_minus'] = False  # Ensure that minus sign is
displayed correctly

    # Plotting parameters
    plt.figure(figsize=(10, 8))
    sns.heatmap(attn_weights, cmap="YlGnBu", xticklabels=input_sequence,
yticklabels=output_sequence, cbar=True, annot=True, fmt=".2f")
    plt.xlabel('Input sequence')
    plt.ylabel('Output sequence')
    plt.title('Attention Heatmap')
    plt.show()
```

分析：

使用 Python 的数据可视化库 `Seaborn` 绘制热力图，将注意力权重矩阵可视化为颜色编码的矩阵来观察 Encoder 中输出序列的隐状态对于 Decoder 每一个时间步的输出的注意力权重

五、实验结果

超参数设置

- MAX_LENGTH = 20
- MAX_WORDS = 20000
- batch_size = 64
- dropout = 0.2
- embed_size = hidden_size = 512
- num_layers = 2
- beam_width = 1
- start_prob = 1.0
- end_prob = 1.0
- learning rate = 0.01
- num_epochs = 50
- optimizer：Adam

数据预处理

```
Reading lines...
Building prefix dict from the default dictionary ...
Loading model from cache C:\Users\32333\AppData\Local\Temp\jieba.cache
Loading model cost 0.319 seconds.
Prefix dict has been built successfully.
Reading lines...
Reading lines...
Read 14533 sentence pairs
Counting words...
Counted words:
11212 Chinese words
5427 English words
```

分析：

由数据预处理的结果截图可见，通过训练集，模型一共读取了 14533 个平行语料对，分别构建了两个词典：

- 中文词典包含 11212 个汉字
- 英文词典包含 5427 个单词

训练过程

```
Epoch:  45 iteration  0 loss:  0.22690075635910034
Epoch:  45 iteration  100 loss:  0.1754915565252304
Epoch:  45 iteration  200 loss:  0.143607959151268
Epoch:  45 Training loss:  0.17954399033648646
Epoch:  46 iteration  0 loss:  0.19506797194480896
Epoch:  46 iteration  100 loss:  0.1280863732099533
Epoch:  46 iteration  200 loss:  0.13765551149845123
Epoch:  46 Training loss:  0.1711110655497657
Epoch:  47 iteration  0 loss:  0.18559592962265015
Epoch:  47 iteration  100 loss:  0.1616246998310089
Epoch:  47 iteration  200 loss:  0.13067451119422913
Epoch:  47 Training loss:  0.16481224048152857
Epoch:  48 iteration  0 loss:  0.1690475195646286
Epoch:  48 iteration  100 loss:  0.16603778302669525
Epoch:  48 iteration  200 loss:  0.09614241868257523
Epoch:  48 Training loss:  0.1659194804128353
Epoch:  49 iteration  0 loss:  0.17283979058265686
Epoch:  49 iteration  100 loss:  0.1683453619480133
Epoch:  49 iteration  200 loss:  0.1564723700284958
Epoch:  49 Training loss:  0.165802551247589
```

测试实例

```
BOS 汤姆 要 发疯 EOS
BOS tom is going crazy EOS
tom is going to drive a car

BOS 我们 很 保守 EOS
BOS we re UNK EOS
we re very big

BOS 我能 问个问题 吗 EOS
BOS can i ask a question EOS
may i ask a question

BOS 来 UNK 吧 EOS
BOS let s discuss it EOS
come on touch it

BOS 我 生病 了 EOS
BOS i m ill EOS
i m sick
```

分析:

由实验结果可得，对于一些短句子，模型还是可以较为准确地进行翻译。

虽然有些翻译并不完全准确，但是模型依旧可以输出其对应的**同义词**，比如

- "can i" 翻译成了 "may i"
- "let's" 翻译成了 "come on"
- "ill" 翻译成了 "sick"

这些都是可接受的。但是由于数据集过小，仍然会出现测试集的单词没有出现在通过训练集构建的词典之中从而导致翻译不准确的现象，比如词典中就没有记录 “保守” 这个词语，导致解码器随机翻译成了 "big"

BLEU 评估得分

```
Average BLEU score: 0.1117
```

分析：

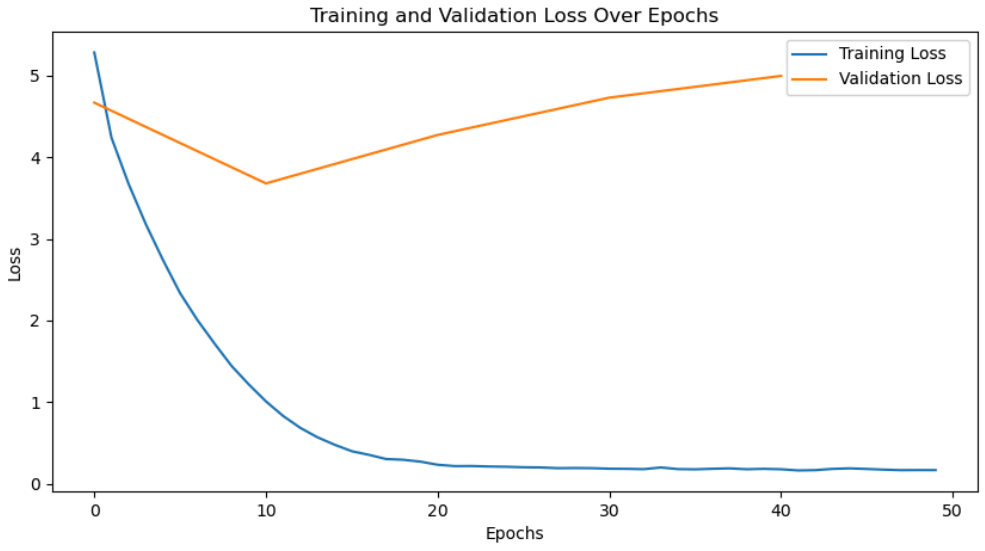
虽然更换了文本较短的数据集，但由于数据的样本过少且训练的轮次较少，因此使用 BLEU 的评估指标得分仅有 0.1117。

初次测试使用的

- 训练策略：Teacher Forcing
- 解码策略：Greedy Search

之后部分会进行不同训练策略以及不同解码策略之间的对比，模型性能会有所提升

损失曲线图



分析：

训练损失和验证损失的趋势

- **训练损失**（蓝色曲线）：训练损失在整个训练过程中逐步下降，并在大约第 20 个 epoch 之后趋于平稳，接近于零。这表明模型在训练数据上的表现逐渐提升，最终能够很好地拟合训练数据。

- **验证损失**（橙色曲线）：验证损失在初期迅速下降，随后在大约第 10 个 epoch 之后开始逐渐上升。这种趋势表明模型在验证数据上的表现开始恶化，可能是因为模型在过度拟合训练数据。

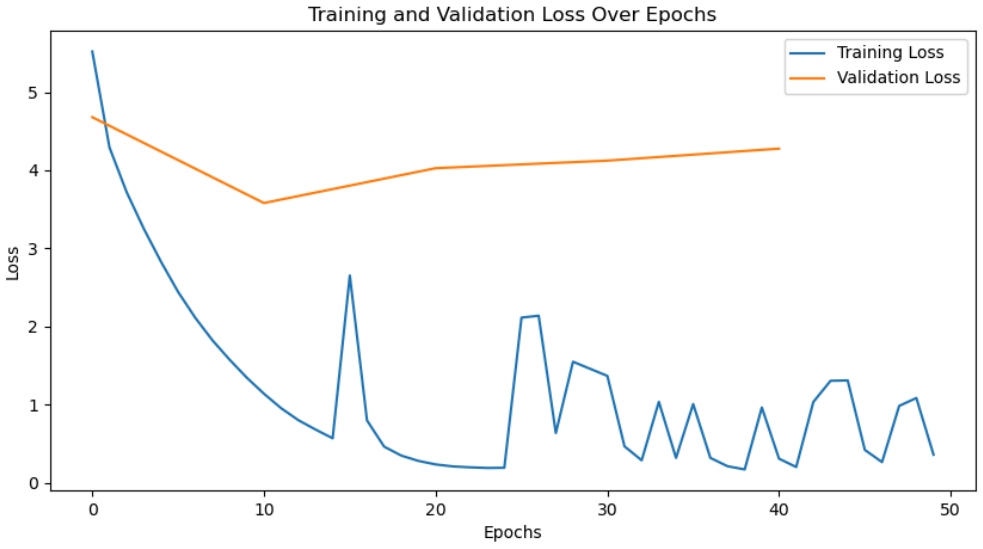
这可能是因为训练数据量不足，导致模型容易记住训练数据而不是学习到泛化的特征。

不同训练策略对比

本部分对比实验中为了控制变量，解码策略均采用 Beam Search

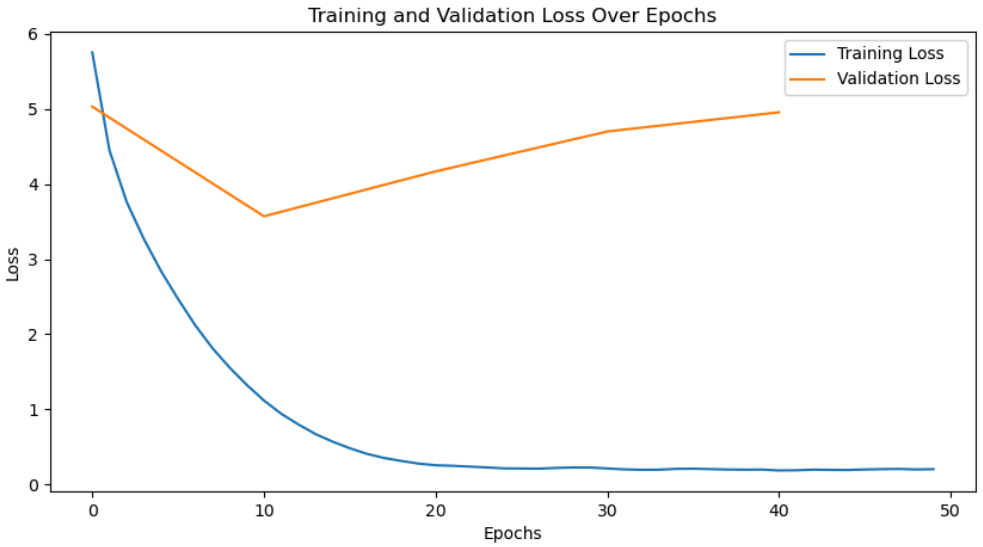
Free Running

```
Free Running with Beam Search:
The total training time of the model is 284.45 s
Average BLEU score: 0.1432
```



Teacher Forcing

```
Teacher Forcing with Beam Search:
The total training time of the model is 203.12 s
Average BLEU score: 0.1422
```



分析：

由实验结果对比可得，两种不同的训练策略各有各的优点与缺点，具体按照以下几点分析：

训练损失

- **Teacher Forcing**：由于使用真实的目标序列作为输入，训练损失会快速下降，并且收敛到一个较低的值。因为模型在每一步都得到正确的上下文，损失的波动较小。
- **Free Running**：训练损失在开始阶段下降较慢，并且相较于前者，损失值的波动较大，这是因为模型有时使用自身的预测作为输入，这些预测在初期可能并不准确。

验证损失

- **Teacher Forcing**：虽然训练损失下降较快，但验证损失可能在在大约第 10 个 epoch 之后开始逐渐上升，表明模型在验证数据上的表现变差，可能出现过拟合现象。
- **Free Running**：由于训练过程中模型逐渐学会应对自身预测的错误，验证损失可能会下降更快更稳定，哪怕同样在大约第 10 个 epoch 之后开始逐渐上升，但是相较于前者，验证损失上升的幅度明显变小，说明 Free Running 在验证数据上的表现更好，减少过拟合现象。

训练过程的稳定性

- **Teacher Forcing**：训练过程较为稳定，损失函数下降平滑。
- **Free Running**：训练过程较不稳定，损失函数下降过程中有较大的波动

训练时间

- **Teacher Forcing**：训练时间较短，因为在每一步中都使用了真实目标序列作为输入，Teacher Forcing 通常能让模型更快地学习到序列之间的映射关系。每一批次的训练时间较短，因为模型在训练过程中较少遇到错误传播
- **Free Running**：训练时间较长，因为模型需要逐渐适应使用自身预测结果作为输入，相当于从并行预测转为了串行预测，导致了每一批次的训练时间可能会较长。随着训练的进行，模型在每一步中逐渐减少对真实目标序列的依赖，增加了训练时间

模型的泛化能力

- **Teacher Forcing**：模型的泛化能力相较于后者略差。因为在训练过程中，模型只通过了目标序列的正确引导进行训练，但却没有学会处理自身预测的错误，导致在推理阶段表现不佳。
- **Free Running**：模型的泛化能力较好。因为在训练过程中，随着训练轮次的加深，模型逐渐适应了自身预测的输入，能更好地应对推理阶段的情况。

但总体来说，由于训练轮次不多，数据集也过小，两种训练方法最后得到的 BLEU 指标分数普遍不高且相差较小

不同解码策略对比

本部分对比实验中为了控制变量，训练策略均采用 Teacher Forcing

Greedy Search

```
Greedy Search:
Average BLEU score: 0.1052
```

Beam Search

```
Beam Search (beam_width = 3):
Average BLEU score: 0.1405
```

分析:

由实验结果可得，使用了 Beam Search 解码策略的模型 BLEU 得分远远高于使用使用了 Greedy Search 解码策略的模型，这是符合我们的实验预期的

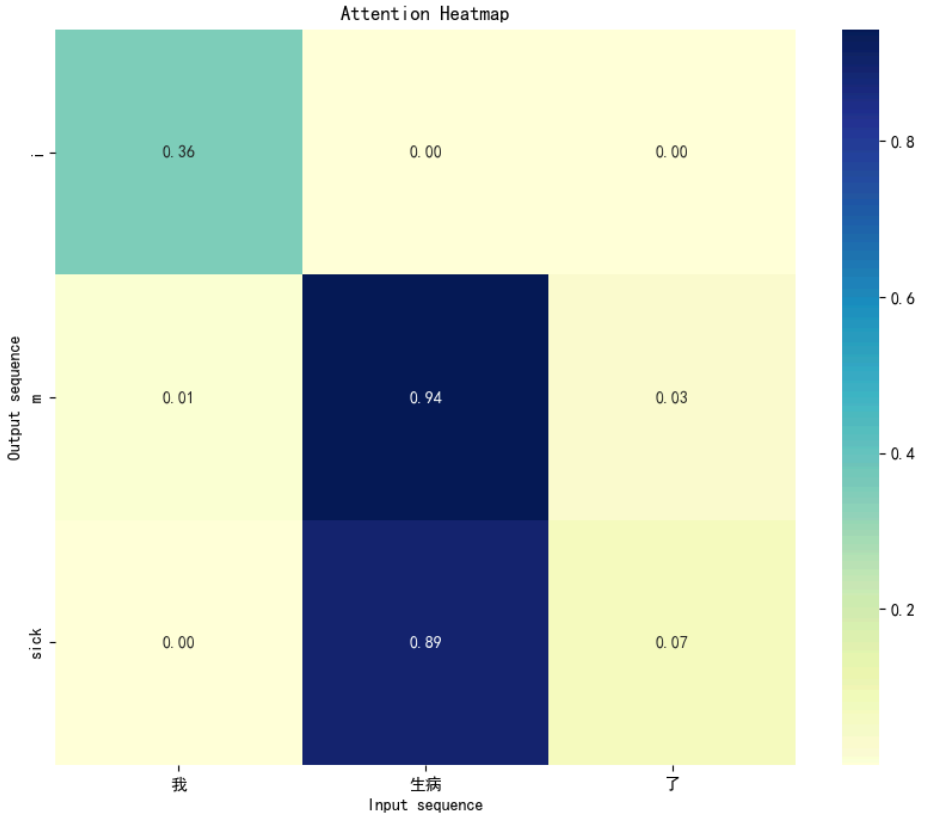
因为束搜索 Beam Search 是一种启发式搜索算法，相比于简单的贪婪搜索 Greedy Search ，它能够在生成过程中保留多个候选路径，从而大幅提升生成结果的质量。

- Greedy Search在每一步只选择当前最优的一个选项，容易错过全局最优解，而 Beam Search 通过在每一步保留前 `beam_width` 个高概率的候选项，能够更全面地探索可能的解，从而提高最终生成结果的质量。
- Beam Search 通过保留多个候选路径，能够在生成过程中同时考虑多种不同的可能性。这种策略平衡了探索（exploration）和利用（exploitation）之间的关系，不会过早地收敛到某个单一的解，从而提升了算法的鲁棒性和可靠性。
- 由于 Beam Search 能够保留多种候选路径，因此模型在面对不确定性和噪声时具有更强的鲁棒性。即使某个路径在某一步出现错误，其他候选路径仍然可以继续保留，从而增加了找到正确解的机会

Attention 可视化

Example 1

- **原始输入**：”我生病了“
- **翻译输出**："I was sick"



分析：

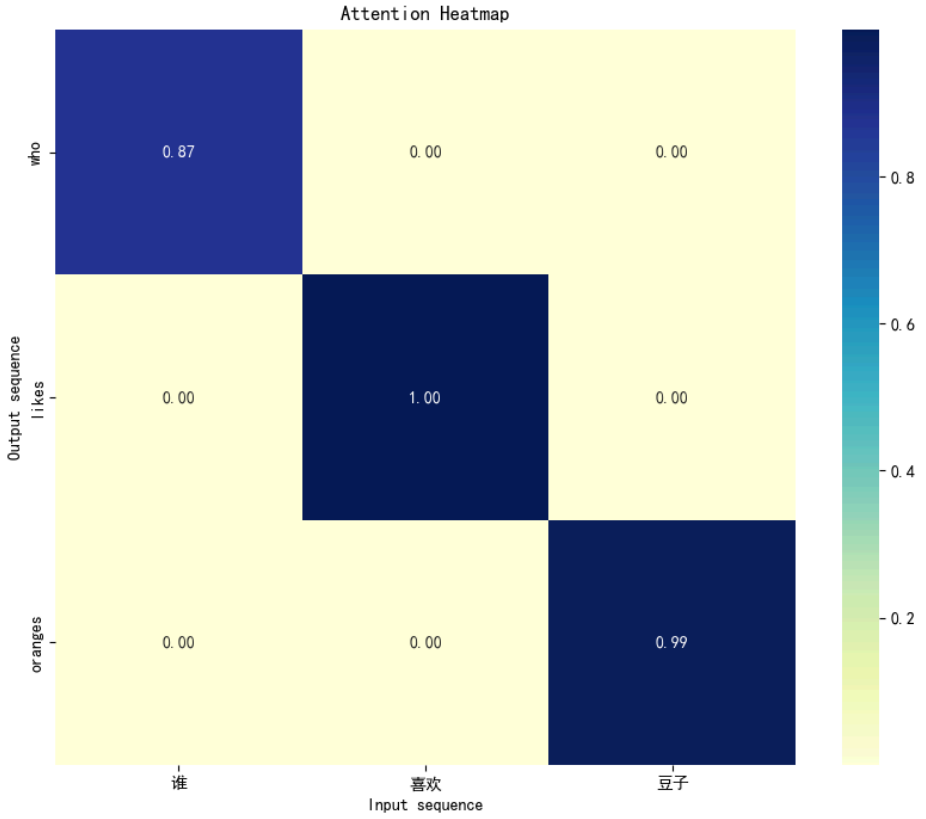
由 Attention 热力图可知：

- 在翻译预测单词 "i" 的时间步时，对于 Encoder 输入序列的隐状态，词语 ”我“ 的注意力权重最高
- 在翻译预测单词 "was" 和 "sick" 的时间步时，对于 Encoder 输入序列的隐状态，词语 ”生病“ 的注意力权重最高

这是符合我们的中译英常识的，说明模型在预测的时候除了使用了 Encoder 最后一个时间步的编码信息以及上一个时间步 Decoder 的输入，还可以很好地观察完整的输入序列并捕捉重要位置上的信息，使得模型能够更全面地理解输入序列中的语义和结构。

Example 2

- **原始输入**：”谁喜欢豆子“
- **翻译输出**："who like oranges"



分析：

由 Attention 热力图可知：

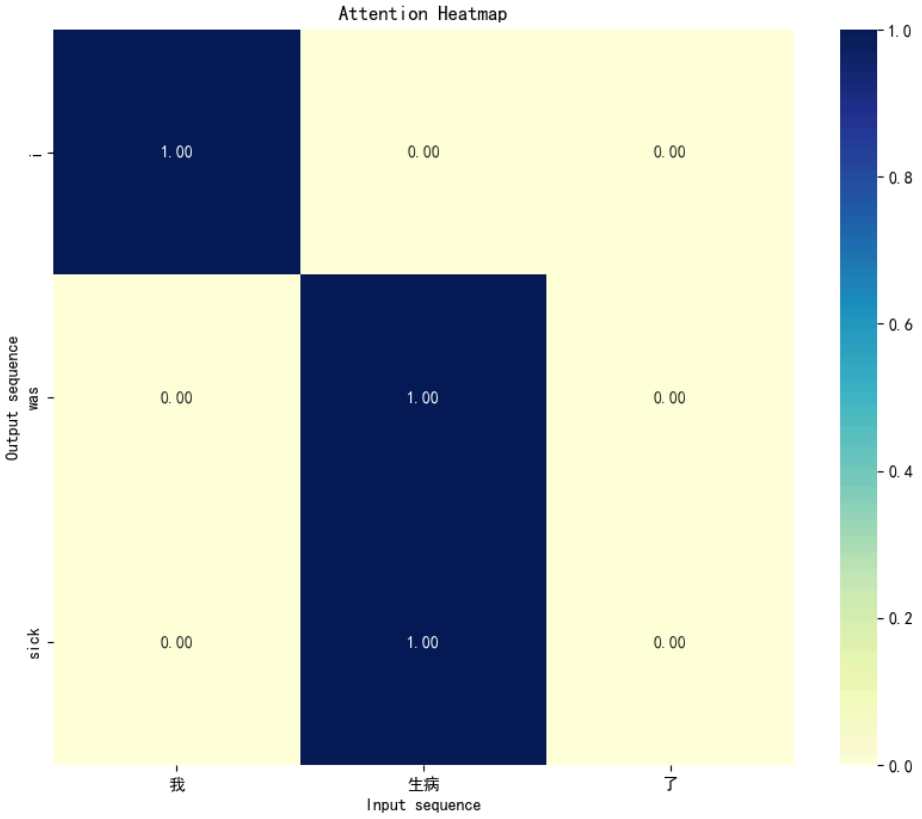
- 在翻译预测单词 "who" 的时间步时，对于 Encoder 输入序列的隐状态，词语 "谁" 的注意力权重最高
- 在翻译预测单词 "like" 的时间步时，对于 Encoder 输入序列的隐状态，词语 "喜欢" 的注意力权重最高
- 在翻译预测单词 "oranges" 的时间步时，对于 Encoder 输入序列的隐状态，词语 "豆子" 的注意力权重最高

可以看到，对于预测的翻译输出，Attention 值的匹配基本是正确的，但由于数据集、训练参数等等的原因，虽然 Decoder 的视野可以扩展到整个输入序列并正确地获取相对当前时间步更重要的位置上的信息，但还是会出现翻译不准确的情况

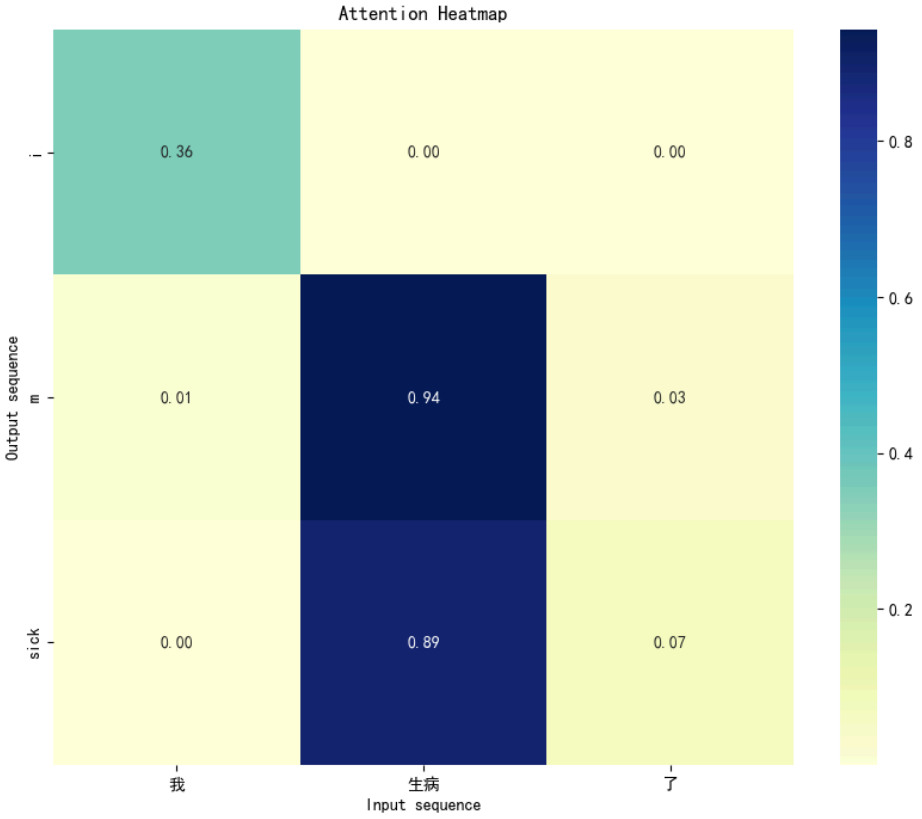
综上所述，由实验结果可得，虽然模型翻译的性能不是很好，但是 Attention 机制还是成功地使得解码器在每一步生成输出时，不仅依赖于上一步的隐状态，还可以选择性地关注输入序列的不同部分，提取重要的信息，尤其是远距离的依赖关系

不同 Attention 实现方式的对比

Dot-product Attention



Multiplicative Attention



分析：

对比使用了 Example 1 中的例子，由对比结果可得

- **Dot-product Attention**：在每一个对应的隐状态下，Attention 值都是 1
- **Multiplicative Attention**：对于每一个输入序列的隐状态，Attention 值没有过度集中或过度分散

这是因为 Dot-product Attention 的实现相对来说较为简单，而 Multiplicative Attention 相较于前者除了引入了 W_{Query}、W_{Key} 和 W_{Value} 这三个权重向量进行线性变化，更重要地是其对注意力分数进行了正则化，使得注意力分数被约束在一个更合理的范围内，在经过 `softmax` 函数计算注意力权重时，输出分布不会那么尖锐（即如 Dot-product Attention 的 Attention 热力图展示的一样，某些位置的注意力权重为 1，而其他位置的权重全部为 0）

六、实验感想

在构建中-英机器翻译系统的实验中，我收获了丰富的知识和实践经验。

首先，通过处理和分析大量的平行语料数据，我加深了对数据预处理步骤重要性的理解，包括数据清洗、分词、构建词典等，这些步骤对于提高模型性能至关重要。

其次，在构建 Seq2Seq 模型并实现 Attention 机制的过程中，我深入学习了循环神经网络（RNN）的结构和工作原理，特别是 GRU 和 LSTM 单元在处理序列数据时的优势。同时，通过实现不同类型的 Attention 机制，我理解了其在模型中的作用，以及如何通过调整 Attention 权重来提高翻译的准确性和流畅性。

在模型训练和推理阶段，我掌握了如何选择合适的损失函数和优化器，并学会了如何调整学习率和批次大小等超参数来优化模型性能。此外，通过对比不同的训练策略 Teacher Forcing 和 Free Running，我了解了它们在训练过程中的影响；通过对比不同的解码策略 Greedy Search 和 Beam Search，我了解了它们在翻译质量上的影响。同时，在实践中，我也选择了最适合本任务的策略并得到了相对较好的实验结果。

最后，在评估模型性能时，我学习了 BLEU 评分等评估指标的计算方法和应用场景，并通过可视化 Attention 矩阵进一步分析了模型在翻译过程中的表现。这些实践经验不仅增强了我的技术能力，也加深了我对机器翻译领域知识的理解。

综上所述，这次实验让我收获颇丰，不仅提升了我的技术能力，也拓宽了我的视野和思维方式。我相信这些收获将对我未来的学习和工作产生积极的影响。