人工神经网络期末项目：中-英机器翻译

学号	姓名	邮箱	专业
21307174	刘俊杰	liujj255@mail2.sysu.edu.cn	计算机科学与技术

一、实验内容

数据集简介

数据集说明：压缩包中共有4个jsonl文件，分别对应着训练集（小）、训练集（大）、验证集和测试集，它们的大小分别是100k、10k、500、200。jsonl 文件中的每一行包含一个平行语料样本。模型的性能以测试集的结果为最终标准。

若计算设备受限，可以仅使用训练集（小）中的10k平行语料进行训练；鼓励探索使用训练集（大）；

数据下载地址：百度网盘

数据预处理

数据清洗：非法字符，稀少字词的过滤；过长句子的过滤或截断。

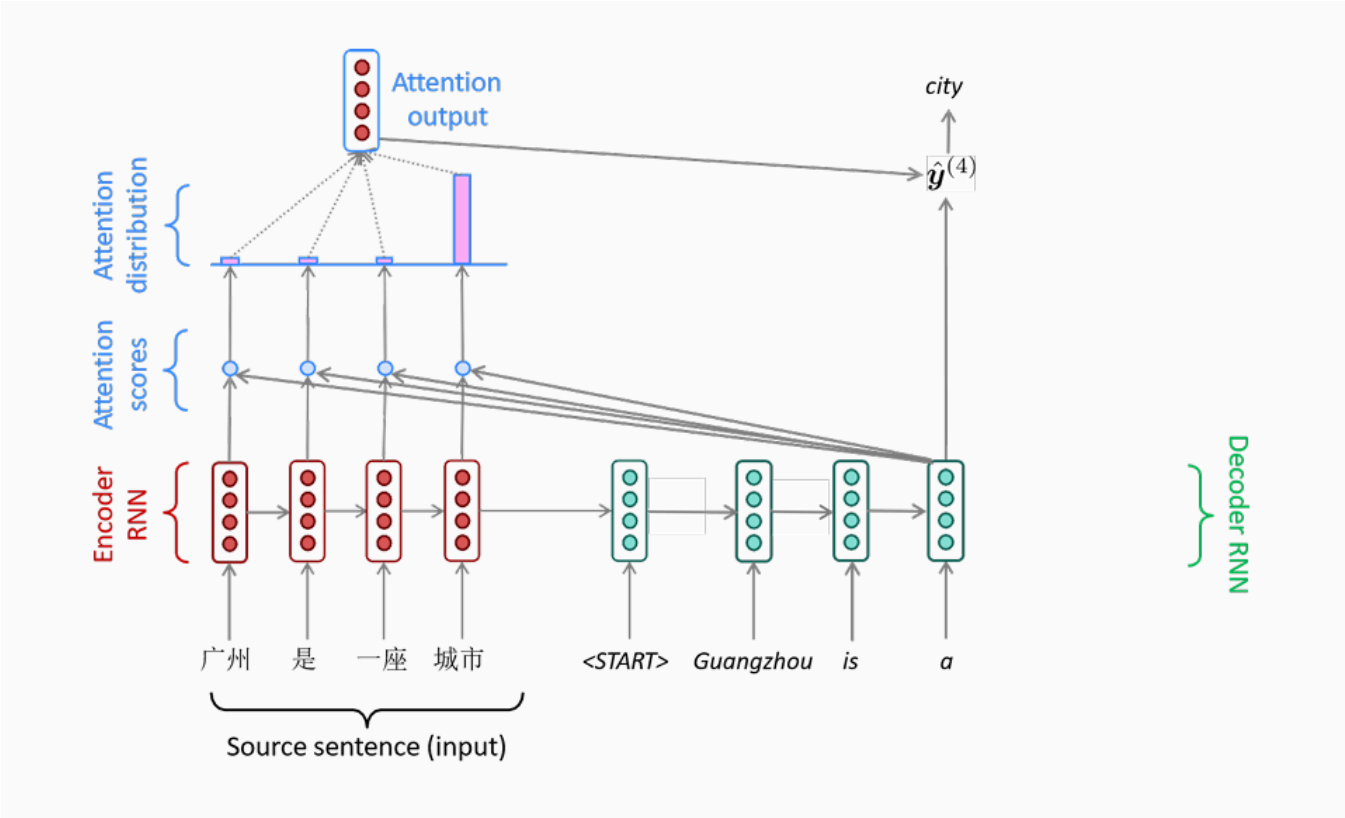
分词：将输入句子切分为tokens，每个子串相对有着完整的语义，便于学习embedding表达

- 英文: 词语之间存在天然的分隔(空格、标点符号)，可以直接利用NLTK或BPE、WordPiece等统计方法分词
- 中文：可以借助分词工具，诸如Jieba(轻量型), HanLP(大体量但效果好)

构建词典：利用分词后的结果构建统计词典，可以过滤掉出现频次较低的词语，防止词典规模过大。

建议用预训练词向量初始化，在训练的过程中允许更新。

NMT 模型



自行构建基于GRU或者LSTM的Seq2Seq模型(编码器和解码器各2层；单向)

自行实现attention机制

自行探索attention机制中不同对齐函数 (dot product, multiplicative, additive) 的影响

训练和推理

定义损失函数（例如交叉熵损失）和优化器（例如Adam）。

将双语平行语料库处理成中译英数据，训练模型的中译英能力。

在训练过程中，对比Teacher Forcing和Free Running策略的效果。

对比greedy和beam-search解码策略；

编程语言与环境

编程语言：Python

深度学习框架：Pytorch

评估指标

BLEU:

$$\text{precision}_n = \frac{\sum_{C \in corpus} \sum_{\text{n-gram} \in C} \text{count-in-reference}_{\text{clip}}(\text{n-gram})}{\sum_{C \in corpus} \sum_{\text{n-gram} \in C} \text{count (n-gram)}}$$

$$\text{BLEU-4} = \min\left(1, \frac{\text{output-length}}{\text{reference-length}}\right) \prod_{i=1}^{4} \text{precision}_i$$

重点考查

- 搭建Seq2Seq模型
- Attention机制的实现
- 训练Seq2Seq模型的技巧
- 分类结果性能评估
- Attention可视化(少量案例进行分析)

提交

- 源代码和训练好的checkpoint
- 文档（PDF）（至少包含方法、实验结果分析以及心得体会）

压缩文件并命名：“2024ANN-final-term-project-学号-姓名.zip/rar”

邮件主题：2024ANN-final-term-project-学号-姓名

提交邮箱：sysucsers@163.com

二、实验设计

(本实验设计主要借鉴了参考资料(seq2seq 机器翻译教程))

数据预处理

数据预处理是机器翻译系统中的一个重要步骤，旨在对原始文本数据进行清理、转换和组织，使

其更适合模型的训练和推理。预处理步骤可以提高模型的性能，减少噪声，并加速训练过程。

定义 `Lang` 类

```
class Lang:
    def __init__(self, name):
        self.name = name
        self.word2index = {}
        self.word2count = {}
        self.index2word = {0: "SOS", 1: "EOS"}
        self.n_words = 2

    def addSentence(self, sentence):
        for word in sentence.split(' '):
            self.addWord(word)

    def addWord(self, word):
        if word not in self.word2index:
            self.word2index[word] = self.n_words
            self.word2count[word] = 1
            self.index2word[self.n_words] = word
            self.n_words += 1
        else:
            self.word2count[word] += 1
```

构建词典:`word2index` 和 `index2word` 分别用于从单词到索引和索引到单词的映射，`word2count` 用于统计每个单词的出现次数。初始化时包括特殊标记 `SOS` 和 `EOS`（开始和结束标记）。

清洗文本函数

```
def clean_text(text):
    # 将中文标点转换为英文标点
    text = text.replace(', ', ',').replace('。', '.').replace
    ('; ', ';').replace(': ', ':').
    replace('? ', '?').replace('! ', '!')
    # 去除异常字符（非字母数字和空格）
    text = re.sub(r'[^\w\s,.!?]', '', text)
    # 去除多余空格
    text = re.sub(r'\s+', ' ', text)
    # 转换为小写
    text = text.lower()
    return text.strip()
```

将中文标点符号转换为英文标点，去除异常字符和多余空格，并将文本转换为小写。

读取和清洗数据

```
def read_and_clean_data(filepath):
    cleaned_data = []
    with open(filepath, 'r', encoding='utf-8') as file:
        for line in file:
            item = json.loads(line)
            item['en'] = clean_text(item['en'])
            item['zh'] = clean_text(item['zh'])
            cleaned_data.append(item)
    return cleaned_data
```

从文件中读取数据，逐行解析 JSON 格式的文本，并对英文和中文文本进行清洗。

过滤超长句子

```
def filter_sentences(sentences, max_length=50):
    return [sentence for sentence in sentences if
            len(sentence['en_tokens']) < max_length and
            len(sentence['zh_tokens']) < max_length
            ]
```

过滤掉长度超过指定 `max_length` 的句子。

分词

```
def tokenize_text(data):
    nltk.download('punkt')
    for item in data:
        item['en_tokens'] = nltk.word_tokenize(item['en'])
        item['zh_tokens'] = list(jieba.cut(item['zh']))
    return data
```

使用 `nltk` 对英文进行分词，使用 `jieba` 对中文进行分词，并将分词结果存储在 `en_tokens` 和 `zh_tokens` 字段中。

构建词典

```
def build_dict(sentences_token, name, max_size=10000):
    lang = Lang(name)
    word_counter = Counter()

    for sentence in sentences_token:
        word_counter.update(sentence)

    # 获取词频前 max_size 的单词
    most_common_words = word_counter.most_common(max_size)

    for word, count in most_common_words:
        for _ in range(count):
            lang.addWord(word)

    return lang
```

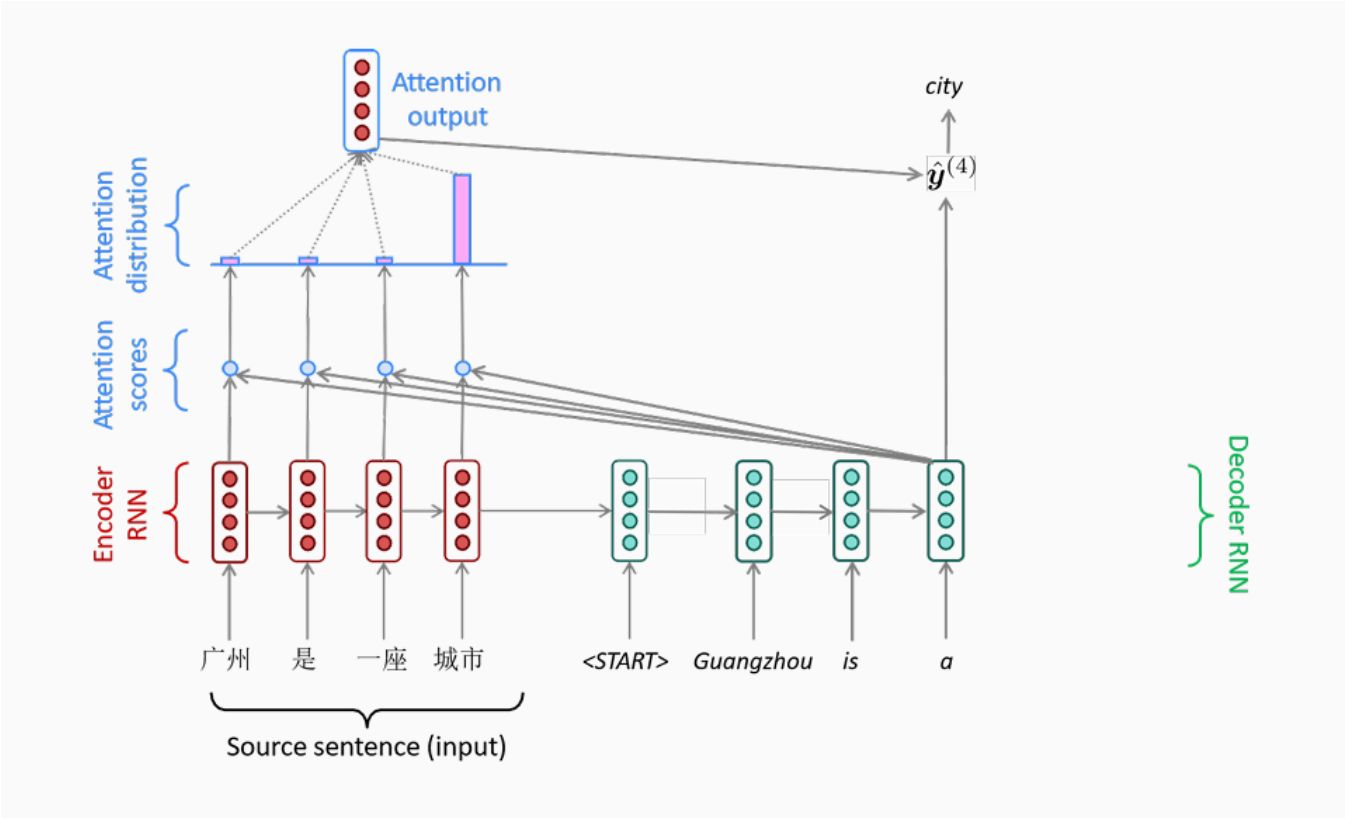
构建词典:首先统计每个单词的频率，然后只保留频率最高的 `max_size` 个单词。最后将这些单词添加到 `Lang` 对象中。

数据预处理主函数

```
def pre_process(data_filepath):
    # 读取和清洗数据
    cleaned_data = read_and_clean_data(data_filepath)
    # 分词
    tokenized_data = tokenize_text(cleaned_data)
    # 过滤句子长度超过50的数据
    filtered_data = filter_sentences(tokenized_data, max_length=50)

    tokenized_en = [item['en_tokens'] for item in filtered_data]
    tokenized_zh = [item['zh_tokens'] for item in filtered_data]

    english_lang = build_dict(tokenized_en, "en", 10000)
    chinese_lang = build_dict(tokenized_zh, "zh", 10000)

    return filtered_data, english_lang, chinese_lang
```

首先读取和清洗数据，然后进行分词，过滤超长句子，最后构建词典并返回处理后的数据和词典,最终实现数据的预处理。

NMT 模型


Attention output
city
$\hat{y}^{(4)}$
Attention distribution
Attention scores
Encoder RNN
Decoder RNN
广州
是
一座
城市
<START>
Guangzhou
is
a
Source sentence (input)

Encoder

```
class EncoderRNN(nn.Module):
    def __init__(self, input_size, hidden_size,
        num_layers=2, dropout_p=0.1):
        super(EncoderRNN, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers

        self.embedding = nn.Embedding(input_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size,
            num_layers=num_layers,
            batch_first=True, dropout=dropout_p)
        self.dropout = nn.Dropout(dropout_p)

    def forward(self, input):
        embedded = self.dropout(self.embedding(input))
        output, hidden = self.gru(embedded)
        return output, hidden
```

- 应用嵌入层、GRU和Dropout。
- 将输入词汇索引转换为嵌入向量，经过Dropout和GRU层后，输出GRU的输出和隐藏状态。

注意力机制

```
class BahdanauAttention(nn.Module):
    def __init__(self, hidden_size):
        super(BahdanauAttention, self).__init__()
        self.Wa = nn.Linear(hidden_size, hidden_size)
        self.Ua = nn.Linear(hidden_size, hidden_size)
        self.Va = nn.Linear(hidden_size, 1)

    def forward(self, query, keys):
        scores = self.Va(torch.tanh(self.Wa(query) + self.Ua(keys)))
        scores = scores.squeeze(2).unsqueeze(1)

        weights = F.softmax(scores, dim=-1)
        context = torch.bmm(weights, keys)

        return context, weights
```

- 初始化了三个线性层 `Wa`、`Ua` 和 `Va`，用于计算注意力得分。
- 计算注意力得分并通过softmax获得注意力权重，然后使用权重计算上下文向量。

Decoder

```
class AttnDecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size, num_layers=2,
    dropout_p=0.1, teaching_rate=1, teaching_decay_rate=1):
        super(AttnDecoderRNN, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers

        self.embedding = nn.Embedding(output_size, hidden_size)
        self.attention = BahdanauAttention(hidden_size)
        self.gru = nn.GRU(2 * hidden_size, hidden_size,
                num_layers=num_layers,
                batch_first=True, dropout=dropout_p)
        self.out = nn.Linear(hidden_size, output_size)
        self.dropout = nn.Dropout(dropout_p)
        self.teaching_rate = teaching_rate
        # 使用 teacher forcing 的概率
        self.teaching_decay_rate = teaching_decay_rate
        # 每次迭代后减少 teacher forcing 的概率

    def forward(self, encoder_outputs, encoder_hidden,
        target_tensor=None):
        batch_size = encoder_outputs.size(0)
        decoder_input = torch.empty(batch_size, 1,
        dtype=torch.long,
                        device=device).fill_(SOS_token)
        decoder_hidden = encoder_hidden
        decoder_outputs = []
        attentions = []

        for i in range(MAX_LENGTH):
            decoder_output, decoder_hidden, attn_weights =
                self.forward_step(decoder_input, decoder_hidden,
                 encoder_outputs)
            decoder_outputs.append(decoder_output)
            attentions.append(attn_weights)

            if target_tensor is not None and random.random()
                < self.teaching_rate:
                decoder_input = target_tensor[:, i].unsqueeze(1)
```

```
                # 使用 teacher forcing
            else:
                _, topi = decoder_output.topk(1)
                decoder_input = topi.squeeze(-1).detach()
                # 使用自己的预测作为下一个输入

        decoder_outputs = torch.cat(decoder_outputs, dim=1)
        decoder_outputs = F.log_softmax(decoder_outputs, dim=-1)
        attentions = torch.cat(attentions, dim=1)

        return decoder_outputs, decoder_hidden, attentions

    def forward_step(self, input, hidden, encoder_outputs):
        embedded = self.dropout(self.embedding(input))

        query = hidden[-1].unsqueeze(0).permute(1, 0, 2)
        context, attn_weights = self.attention(query, encoder_outputs)
        input_gru = torch.cat((embedded, context), dim=2)

        output, hidden = self.gru(input_gru, hidden)
        output = self.out(output)

        return output, hidden, attn_weights
```

- 使用嵌入层、注意力机制、GRU层、输出线性层和Dropout。
- 逐步生成每个时间步的输出，并根据teacher forcing或free_running策略选择下一个输入。
- **forward_step**：执行单步前向传播，包括计算注意力、拼接输入和上下文向量、经过GRU和输出层。

训练和推理

`train_epoch`

训练模型一个epoch（即一次完整的遍历训练数据集）：

```
def train_epoch(dataloader, encoder, decoder,
encoder_optimizer, decoder_optimizer, criterion):
    total_loss = 0
    for data in tqdm(dataloader):
        input_tensor, target_tensor = data

        encoder_optimizer.zero_grad()
        decoder_optimizer.zero_grad()

        encoder_outputs, encoder_hidden = encoder(input_tensor)

        # 使用teacher forcing
        decoder_outputs, _, _ = decoder(encoder_outputs,
         encoder_hidden, target_tensor)

        # 或者使用free running
        # decoder_outputs, _, _ =
        # decoder(encoder_outputs, encoder_hidden, None)

        loss = criterion(
            decoder_outputs.view(-1, decoder_outputs.size(-1)),
            target_tensor.view(-1)
        )
        loss.backward()

        encoder_optimizer.step()
        decoder_optimizer.step()

        total_loss += loss.item()

    return total_loss / len(dataloader)
```

- **数据加载**：从dataloader中获取一批数据`input_tensor`和`target_tensor`。
- **优化器梯度归零**：在每次反向传播之前，将梯度归零。
- **前向传播**：
    - 将输入数据传递给encoder，获取`encoder_outputs`和`encoder_hidden`。
    - 将encoder的输出和隐藏状态传递给decoder，获取`decoder_outputs`。

- **计算损失**：使用损失函数`criterion`计算损失。
- **反向传播和优化**：反向传播损失并更新模型参数。
- **累加损失**：将每个批次的损失累加以计算平均损失。

`train` 函数

负责组织整个训练过程，包括日志记录、保存检查点和在验证集上进行评估。

```
def train(train_dataloader, valid_dataloader, encoder,
    decoder, n_epochs,
    input_lang, output_lang, learning_rate=0.001, print_every=1,
    plot_every=1,
    checkpoint_path='checkpoint.pth',
    best_checkpoint_path='best_checkpoint.pth'):

    start = time.time()
    plot_losses = []
    print_loss_total = 0
    plot_loss_total = 0

    encoder_optimizer = optim.Adam(encoder.parameters(),
    lr=learning_rate)
    decoder_optimizer = optim.Adam(decoder.parameters(),
     lr=learning_rate)
    criterion = nn.NLLLoss()

    start_epoch = 1
    best_valid_bleu = 0.0

    #if os.path.exists(checkpoint_path):
    #    start_epoch = load_checkpoint(checkpoint_path,
    #encoder, decoder, encoder_optimizer, decoder_optimizer) + 1

    for epoch in range(start_epoch, n_epochs + 1):
        loss = train_epoch(train_dataloader,
        encoder, decoder,
        encoder_optimizer, decoder_optimizer, criterion)
        print_loss_total += loss
        plot_loss_total += loss

        if epoch % print_every == 0:
            encoder.eval()
            decoder.eval()

            print_loss_avg = print_loss_total / print_every
            print_loss_total = 0
            log_message = '%s (%d %d%%) %.4f' %
```

```
            (timeSince(start, epoch / n_epochs),
            epoch, epoch / n_epochs * 100, print_loss_avg)
            logging.info(log_message)

            train_bleu = evaluate_bleu(train_dataloader,
            encoder, decoder, input_lang, output_lang)
            valid_bleu = evaluate_bleu(valid_dataloader,
            encoder, decoder, input_lang, output_lang)
            logging.info(f'Training BLEU score after epoch
            {epoch}: {train_bleu:.4f}')
            logging.info(f'Validation BLEU score after epoch
            {epoch}: {valid_bleu:.4f}')

            if valid_bleu > best_valid_bleu:
                best_valid_bleu = valid_bleu
                save_checkpoint(encoder, decoder, encoder_optimizer,
                decoder_optimizer,
                epoch, best_checkpoint_path)

            encoder.train()
            decoder.train()

            save_checkpoint(encoder, decoder, encoder_optimizer,
            decoder_optimizer, epoch, checkpoint_path)

        if epoch % plot_every == 0:
            plot_loss_avg = plot_loss_total / plot_every
            plot_losses.append(plot_loss_avg)
            plot_loss_total = 0
        decoder.teaching_rate *= decoder.teaching_decay_rate
    showPlot(plot_losses)
```

- **初始化**：设置初始参数，包括优化器、损失函数和日志记录。
- **训练循环**：循环`n_epochs`次，每次训练一个epoch：
    - **训练一个epoch**：调用`train_epoch`函数。
    - **日志记录**：每隔`print_every`个epoch记录一次日志，包括损失和BLEU分数。
    - **评估**：在训练集和验证集上计算BLEU分数。

        - **保存最佳模型**：如果验证集上的BLEU分数是最好的，则保存检查点。
        - **绘图**：每隔`plot_every`个epoch更新一次损失曲线图。
    - **每次降低使用teacher forcing的概率**:

```
decoder.teaching_rate *= decoder.teaching_decay_rate
```

`evaluate_bleu` 函数

在给定的数据集上计算BLEU分数。

```
def evaluate_bleu(dataloader, encoder, decoder,
     input_lang, output_lang):
    total_bleu = 0
    n = 0
    x = 0
    for data in dataloader:
        input_tensor, target_tensor = data
        encoder_outputs, encoder_hidden = encoder(input_tensor)
        decoder_outputs, _, _ = decoder(encoder_outputs,
        encoder_hidden, None)

        for i in range(target_tensor.size(0)):
            target = target_tensor[i].cpu().numpy()
            decoded = decoder_outputs[i].argmax(1).cpu().numpy()

            target_sentence = [output_lang.index2word[token]
            for token in target if token not in (SOS_token,
            EOS_token) and token in output_lang.index2word]
            decoded_sentence = [output_lang.index2word[token]
            for token in decoded if token not in (SOS_token,
            EOS_token) and token in output_lang.index2word]
            if x < 10:
                logging.info(f"target: {target_sentence}
                decoded: {decoded_sentence}")
                x += 1
            if len(target_sentence) == 0 or len(decoded_sentence) == 0:
                continue
            smooth = SmoothingFunction()
            bleu_score = sentence_bleu([target_sentence],
            decoded_sentence, smoothing_function=smooth.method1)

            total_bleu += bleu_score
            n += 1
    return total_bleu / n if n > 0 else 0
```

- **数据加载**：从dataloader中获取一批数据`input_tensor`和`target_tensor`。
- **前向传播**：将输入数据传递给encoder和decoder，获取`encoder_outputs`和`decoder_outputs`。
- **计算BLEU分数**：将目标序列和解码序列转换为句子，计算每个句子的BLEU分数并

累加。

- **返回平均BLEU分数**：返回所有句子的平均BLEU分数。

`evaluate_bleu_beam_search` 函数

这个函数使用束搜索（beam search）进行解码并计算BLEU分数。

```
def evaluate_bleu_beam_search(dataloader, encoder, decoder,
    input_lang, output_lang, beam_width=3):
    total_bleu = 0
    n = 0
    x = 0
    for data in dataloader:
        input_tensor, target_tensor = data
        encoder_outputs, encoder_hidden = encoder(input_tensor)

        decoded_sequences = []
        for i in range(input_tensor.size(0)):
            decoded_sequence = decoder.beam_search(encoder_outputs[i],
            encoder_hidden[:, i, :], beam_width=beam_width)
            decoded_sequences.append(decoded_sequence)

        for i in range(target_tensor.size(0)):
            target = target_tensor[i].cpu().numpy()
            decoded = decoded_sequences[i]

            target_sentence = [output_lang.index2word[token]
            for token in target if token not in (SOS_token,
            EOS_token) and token in output_lang.index2word]
            decoded_sentence = [output_lang.index2word[token]
            for token in decoded if token not in (SOS_token,
            EOS_token) and token in output_lang.index2word]
            if x < 10:
                logging.info(f"target: {target_sentence}
                decoded: {decoded_sentence}")
                x += 1
            if len(target_sentence) == 0 or len(decoded_sentence) == 0:
                continue
            smooth = SmoothingFunction()
            bleu_score = sentence_bleu([target_sentence], decoded_sentence, smoothing_function=smo

            total_bleu += bleu_score
            n += 1
    return total_bleu / n if n > 0 else 0
```

- **数据加载**：从dataloader中获取一批数据`input_tensor`和`target_tensor`。

- **前向传播**：将输入数据传递给encoder，获取`encoder_outputs`和`encoder_hidden`。
- **束搜索解码**：对每个输入句子使用束搜索进行解码，得到解码序列。
- **计算BLEU分数**：将目标序列和解码序列转换为句子，计算每个句子的BLEU分数并累加。
- **返回平均BLEU分数**：返回所有句子的平均BLEU分数。

attention机制中不同对齐函数 (dot product,multiplicative, additive)

在实现注意力机制时，可以使用不同的对齐函数来计算查询（query）和键（key）之间的相似度。这些对齐函数包括点积（dot product）、乘法（multiplicative）和加法（additive）。我们将分别介绍并实现这些对齐函数，并展示如何在一个简单的注意力机制中应用它们。

1. 点积（Dot Product）对齐函数

点积对齐函数计算查询和键的点积，常用于Scaled Dot-Product Attention。

```
class DotProductAttention(nn.Module):
    def __init__(self):
        super(DotProductAttention, self).__init__()
          self.hidden_size = hidden_size

    def forward(self, query, key, value, mask=None):
        # 点积
        scores = torch.matmul(query, key.transpose(-2, -1))
        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9)
        # 归一化
        attn_weights = F.softmax(scores, dim=-1)
        output = torch.matmul(attn_weights, value)
        return output, attn_weights
```

2. 乘法（Multiplicative）对齐函数

乘法对齐函数是点积对齐函数的一个变体，常用于一般注意力机制（general attention）。

```
class MultiplicativeAttention(nn.Module):
    def __init__(self, key_dim, query_dim):
        super(MultiplicativeAttention, self).__init__()
        self.linear = nn.Linear(query_dim, key_dim, bias=False)
        self.hidden_size = hidden_size

    def forward(self, query, key, value, mask=None):
        # 线性变换
        query = self.linear(query)
        # 点积
        scores = torch.matmul(query, key.transpose(-2, -1))
        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9)
        # 归一化
        attn_weights = F.softmax(scores, dim=-1)
        output = torch.matmul(attn_weights, value)
        return output, attn_weights
```

3. 加法（Additive）对齐函数

加法对齐函数使用一个前馈神经网络来计算查询和键之间的相似度，常用于Bahdanau Attention。

```
class BahdanauAttention(nn.Module):
    def __init__(self, hidden_size):
        super(BahdanauAttention, self).__init__()
        self.Wa = nn.Linear(hidden_size, hidden_size)
        self.Ua = nn.Linear(hidden_size, hidden_size)
        self.Va = nn.Linear(hidden_size, 1)

    def forward(self, query, keys):
        scores = self.Va(torch.tanh(self.Wa(query) + self.Ua(keys)))
        scores = scores.squeeze(2).unsqueeze(1)

        weights = F.softmax(scores, dim=-1)
        context = torch.bmm(weights, keys)

        return context, weights
```

Teacher Forcing和Free Running策略

Teacher Forcing

Teacher Forcing 是一种在训练过程中使用真实目标序列（ground truth）来指导解码器输入的策略。简单来说，在每一步生成序列时，解码器会使用训练数据中的真实单词作为下一步的输入，而不是依赖自己生成的单词。

优点：

- 收敛速度快：因为解码器总是看到正确的输入，训练过程会更快地收敛。
- 减少错误积累：由于每一步都使用真实的目标单词，可以避免错误传播。

缺点：

- 不匹配实际情况：在实际应用中，解码器只能依赖自己生成的单词作为下一步的输入，Teacher Forcing 会导致训练和推理阶段的不一致。
- 过度依赖真实输入：模型可能会过度依赖真实的输入，导致在推理阶段表现不佳。

Free Running

Free Running 是一种在训练过程中使用解码器自己生成的单词作为下一步输入的策略。即解码器的输入完全依赖于自己在上一步生成的输出。

优点：

- 训练与推理一致：模型在训练时的输入方式与实际推理时一致，有助于模型在推理阶段的性能。
- 更好地适应错误：模型学会处理自己生成的错误输入，更能适应实际应用中的情况。

缺点：

- 收敛速度慢：由于解码器输入的单词可能是错误的，训练过程可能会更慢，收敛难度更大。
- 错误积累：每一步的错误都会传递到下一步，导致错误积累，影响生成质量。

结合两种策略

在实际应用中，可以结合 Teacher Forcing 和 Free Running 两种策略，以达到更好的训练效

果。

代码实现

```
self.teaching_rate = teaching_rate
# 使用 teacher forcing 的概率
self.teaching_decay_rate = teaching_decay_rate
# 每次迭代后减少 teacher forcing 的概率
```

`teaching_rate` 表示在训练过程中使用 Teacher Forcing 的概率，而 `teaching_decay_rate` 表示每次迭代后减少 Teacher Forcing 概率的因子。

```
if target_tensor is not None and random.random() <
    self.teaching_rate:
    decoder_input = target_tensor[:, i].unsqueeze(1)
    # 使用 teacher forcing
    self.teaching_rate *= self.teaching_decay_rate
    # 减少 teacher forcing 的概率
else:
    _, topi = decoder_output.topk(1)
    decoder_input = topi.squeeze(-1).detach()
    # 使用自己的预测作为下一个输入
```

这样子应用`teaching_rate` 和 `teaching_decay_rate`， 实现了:

- `teaching_rate` = 0， 使用Free Running
- `teaching_rate` = 1，`teaching_decay_rate` = 1,使用Teacher Forcing
- `teaching_rate` = 1，0< `teaching_decay_rate` <1,将两者结合在一起使用,前期训练偏向使用Teacher Forcing 后期偏向使用 Free Running

greedy和beam-search解码策略

前面代码实现的是greedy解码策略

Beam Search 解码策略

```
def beam_search(self, encoder_outputs, encoder_hidden, max_length=MAX_LENGTH, beam_width=3):
    k = beam_width
    sequences = [[list(), 0.0, encoder_hidden]]
    # (sequence, score, hidden)

    for _ in range(max_length):
        all_candidates = list()
        for seq, score, hidden in sequences:
            decoder_input = torch.tensor([seq[-1]]).unsqueeze
            (0).to(device) if seq else torch.tensor
            ([SOS_token]).unsqueeze(0).to(device)
            decoder_output, hidden, attn_weights = self.
            forward_step(decoder_input, hidden,
            encoder_outputs)

            topk = decoder_output.topk(k)
            for i in range(k):
                candidate = [seq + [topk[1][0][i].item()], score
                - topk[0][0][i].item(), hidden]
                all_candidates.append(candidate)

        ordered = sorted(all_candidates, key=lambda x: x[1])
        sequences = ordered[:k]

    return sequences[0][0]  # 返回最佳序列
```

- **beam_search**：使用Beam Search策略进行解码。
    - 初始化 `k` 个候选序列（起始只有一个），每个序列包括（序列、分数、隐藏状态）。
    - 对于每个时间步，计算所有候选序列的可能拓展，保留 `k` 个得分最高的候选序列。
    - 最终返回得分最高的序列。

使用beam search进行解码并计算BLEU分数:

```
def evaluate_bleu_beam_search(dataloader, encoder, decoder,
input_lang,output_lang, beam_width=3):
    total_bleu = 0
    n = 0
    x = 0
    for data in dataloader:
        input_tensor, target_tensor = data
        encoder_outputs, encoder_hidden = encoder(input_tensor)

        decoded_sequences = []
        for i in range(input_tensor.size(0)):
            decoded_sequence = decoder.beam_search

            (encoder_outputs[i], encoder_hidden[:, i, :],
            beam_width=beam_width)
            decoded_sequences.append(decoded_sequence)

        for i in range(target_tensor.size(0)):
            target = target_tensor[i].cpu().numpy()
            decoded = decoded_sequences[i]

            target_sentence = [output_lang.index2word[token]

for token in target if token not in (SOS_token, EOS_token)

 and token in output_lang.index2word]
            decoded_sentence = [output_lang.index2word[token]
 for token in decoded if token not in (SOS_token, EOS_token)
and token in output_lang.index2word]
            if x < 10:
                logging.info(f"target:
                {target_sentence} decoded: {decoded_sentence}")
                x += 1
            if len(target_sentence) ==
                0 or len(decoded_sentence) == 0:
                continue
            smooth = SmoothingFunction()
            bleu_score = sentence_bleu([target_sentence],
              decoded_sentence, smoothing_function=smooth.method1)
```

```
            total_bleu += bleu_score
            n += 1
    return total_bleu / n if n > 0 else 0
```

- **数据加载**：从dataloader中获取一批数据`input_tensor`和`target_tensor`。
- **前向传播**：将输入数据传递给encoder，获取`encoder_outputs`和`encoder_hidden`。
- **束搜索解码**：对每个输入句子使用束搜索进行解码，得到解码序列。
- **计算BLEU分数**：将目标序列和解码序列转换为句子，计算每个句子的BLEU分数并累加。
- **返回平均BLEU分数**：返回所有句子的平均BLEU分数。

Attention可视化

```
def visualize_Attention(sentence, output_words, attn_weights):

    fig, ax = plt.subplots(figsize=(10, 8))
    plt.rcParams['font.sans-serif'] = ['SimHei']

    attn_weights = attn_weights.detach().cpu().numpy()

    # 检查并调整维度
    if attn_weights.ndim == 1:
        attn_weights = attn_weights.reshape(1, -1)
    elif attn_weights.ndim == 3 and attn_weights.shape[0] == 1:
        attn_weights = attn_weights.squeeze(0)

    cax = ax.matshow(attn_weights, cmap='bone')
    fig.colorbar(cax)

    ax.set_xticks(range(len(sentence)))
    ax.set_xticklabels(sentence, rotation=90, fontsize=12)  # 设置字体大小为12
    ax.set_yticks(range(len(output_words)))
    ax.set_yticklabels(output_words, fontsize=12)  # 设置字体大小为12

    ax.xaxis.set_major_locator(ticker.MultipleLocator(1))
    ax.yaxis.set_major_locator(ticker.MultipleLocator(1))

    # 找到有字的矩形范围并裁剪图像
    plt.tight_layout()
    ax.set_xlim(-0.5, len(sentence) - 0.5)
    ax.set_ylim(len(output_words) - 0.5, -0.5)

    # 保存图像时只保存有字的部分
    fig.savefig("attention.jpg", bbox_inches='tight')
```

三、实验结果

(由于计算设备和时间有限,前期的不同策略方法等的对比实验使用10k的训练集,最后使用100k的训练集训练模型)

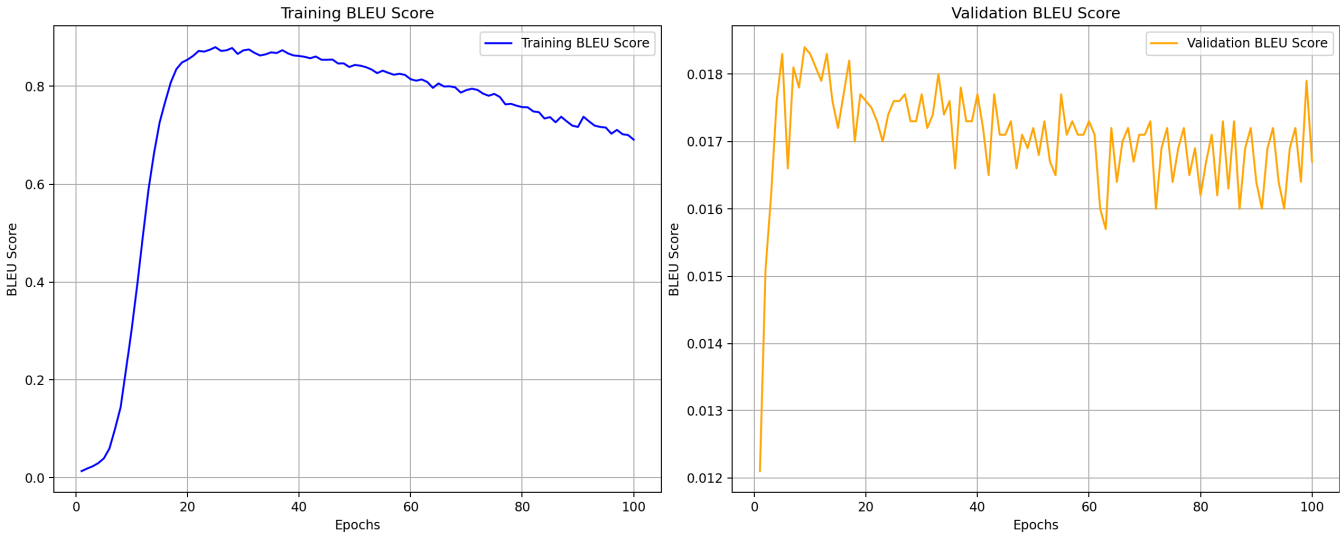
对比Teacher Forcing和Free Running策略

Teacher Forcing

超参数设置:

teaching_rate = 1
teaching_decay_rate = 1
attention = additive
learning_rate = 0.001

实验结果:



TEST BLUE: 0.0163

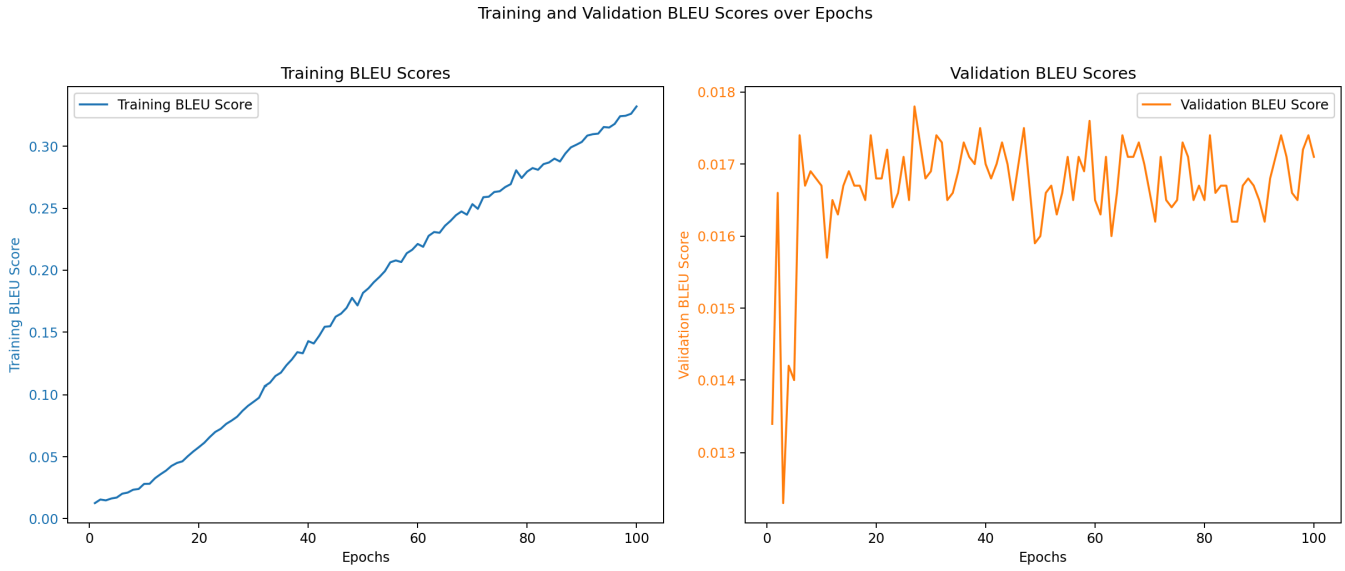
可以看到使用Teaching Forcing策略,在训练集上可以快速得到0.8以上的得分，但在验证集上只有0.018。

Free Running

超参数设置:

teaching_rate = 0
teaching_decay_rate = 0
attention = additive
learning_rate = 0.001

实验结果:



TEST BLUE:0.0153

可以看到使用Free Running策略,在训练集上的得分没有快速提高但在慢慢升高，但和Teacher Forcing一样在验证集上只有0.018。

两者结合

超参数设置:

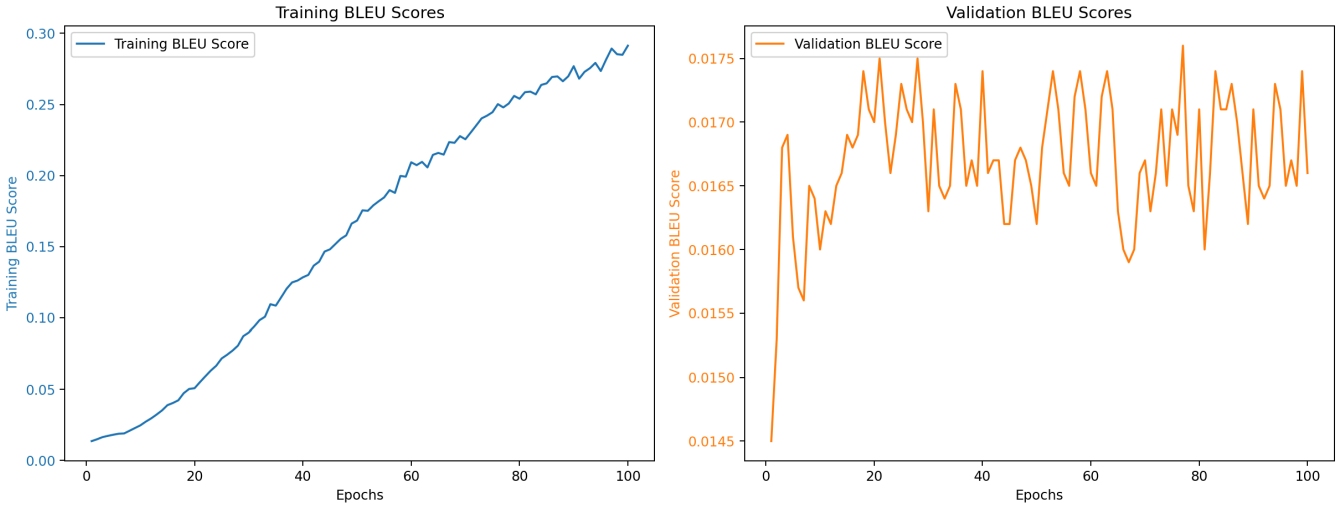
teaching_rate = 1

teaching_decay_rate = 0.95

attention = additive

learning_rate = 0.001

实验结果:



TEST BLUE:0.0156

将两种策略结合在一起，可以看到训练集上的得分也在稳步提高,但在验证集上的效果有限。

综上来说,可以发现发现模型发现了过拟合，可能是网络模型较为复杂而训练数据较少导致的,故需要在100k的大数据集上更好地训练模型。

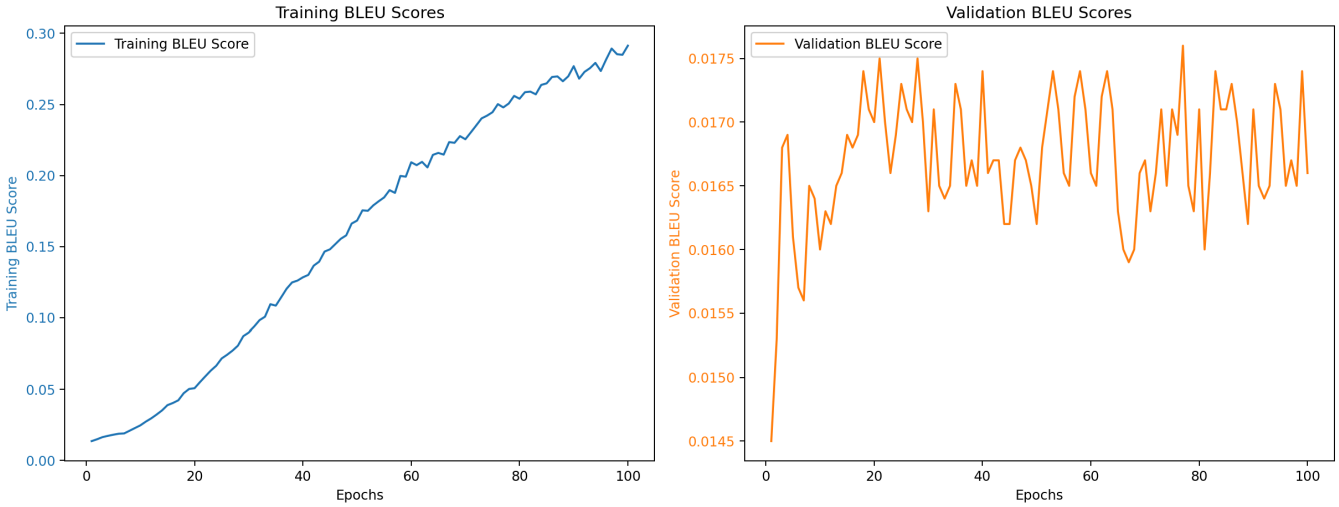
对比greedy和beam-search解码策略

greedy解码策略

超参数设置:

teaching_rate = 1
teaching_decay_rate = 0.95
attention = additive
learning_rate = 0.001

实验结果:



TEST BLUE:0.0156

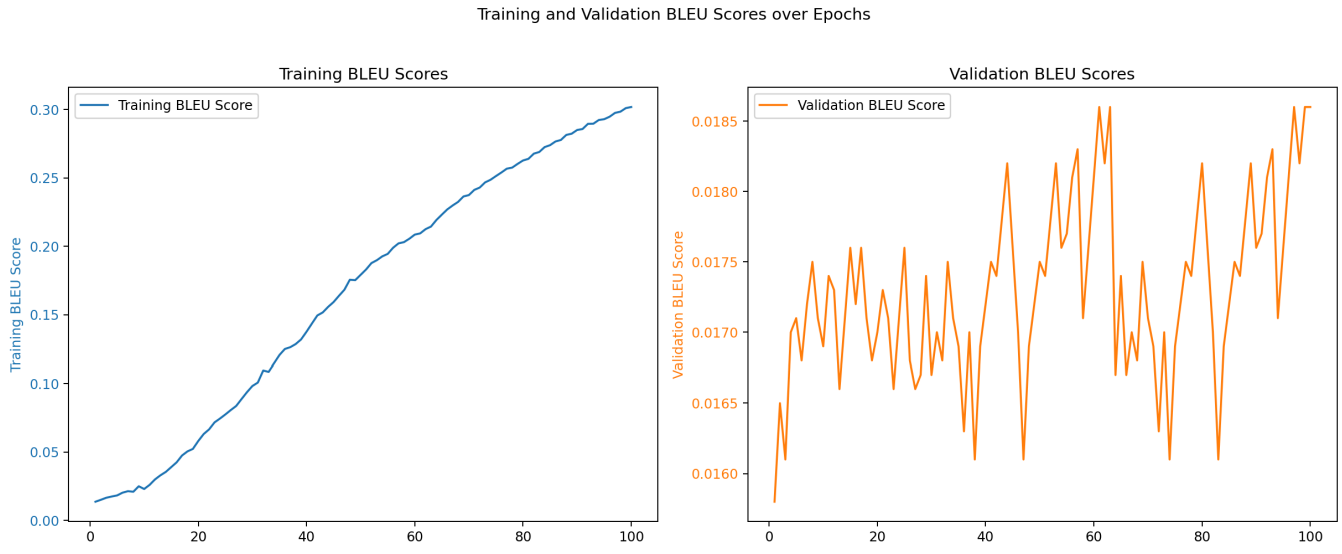
beam-search解码策略

TEST BLUE:0.0156

超参数设置:

teaching_rate = 1
teaching_decay_rate = 0.95
attention = additive
learning_rate = 0.001
beam width = 5

实验结果:



可以看到beam-search解码策略较greedy效果稍好一些，但是训练时需要花费更多的空间和时间。

探索attention机制中不同对齐函数 (dot product,multiplicative, additive) 的影响

additive

超参数设置:

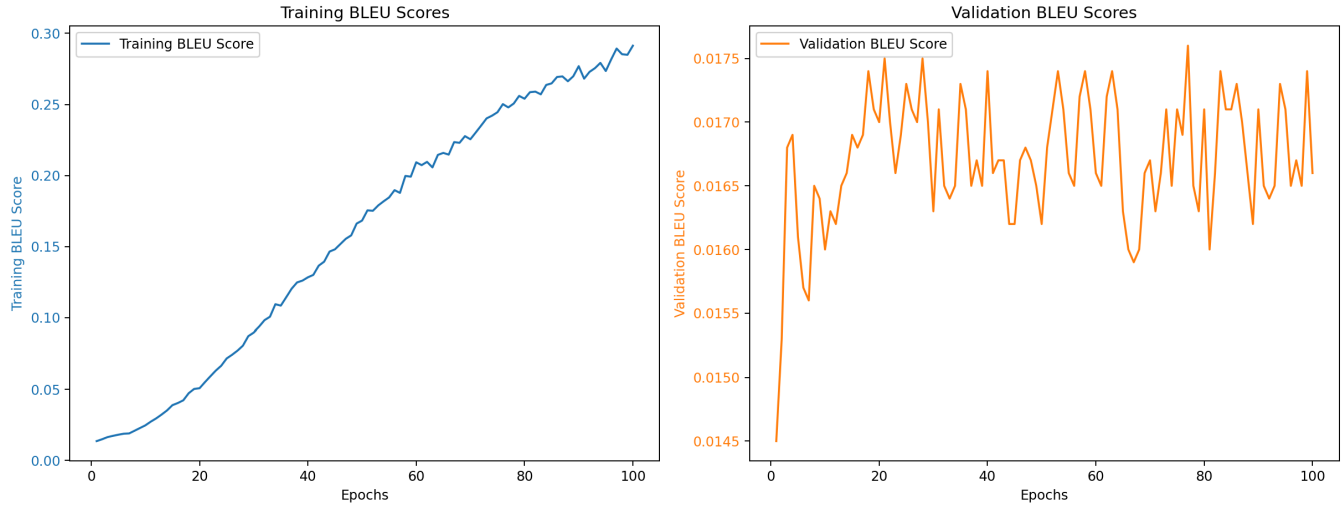
teaching_rate = 1

teaching_decay_rate = 0.95

attention = additive

learning_rate = 0.001

实验结果:



TEST BLUE:0.0156

dot product

超参数设置:

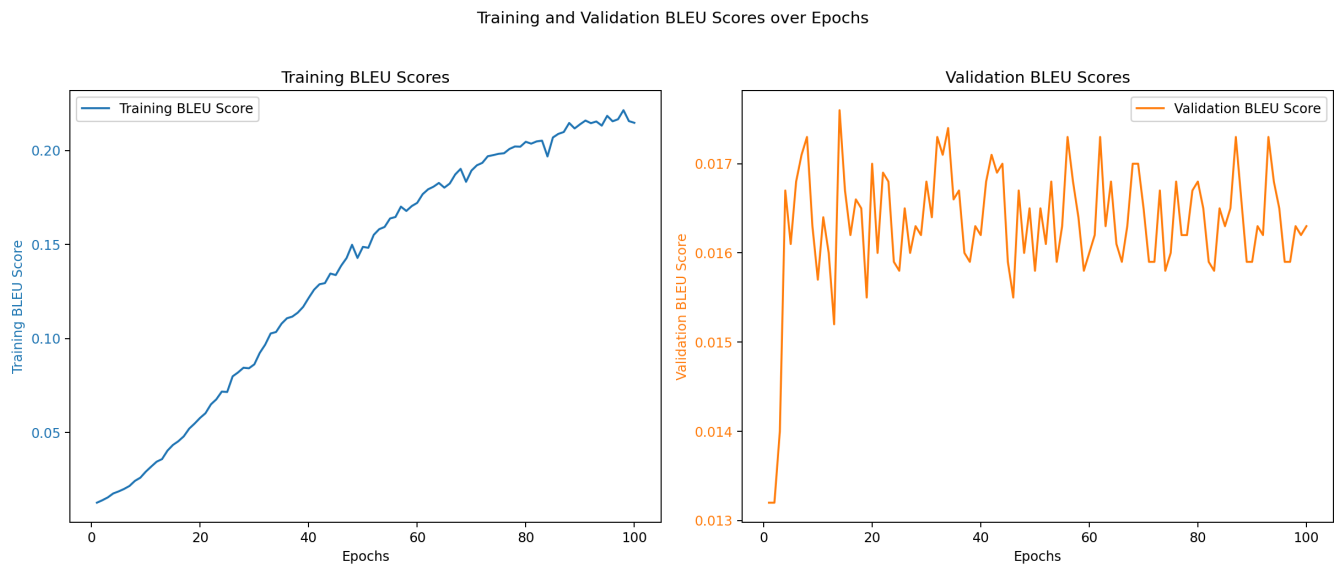
teaching_rate = 1

teaching_decay_rate = 0.95

attention = dot product

learning_rate = 0.001

实验结果:



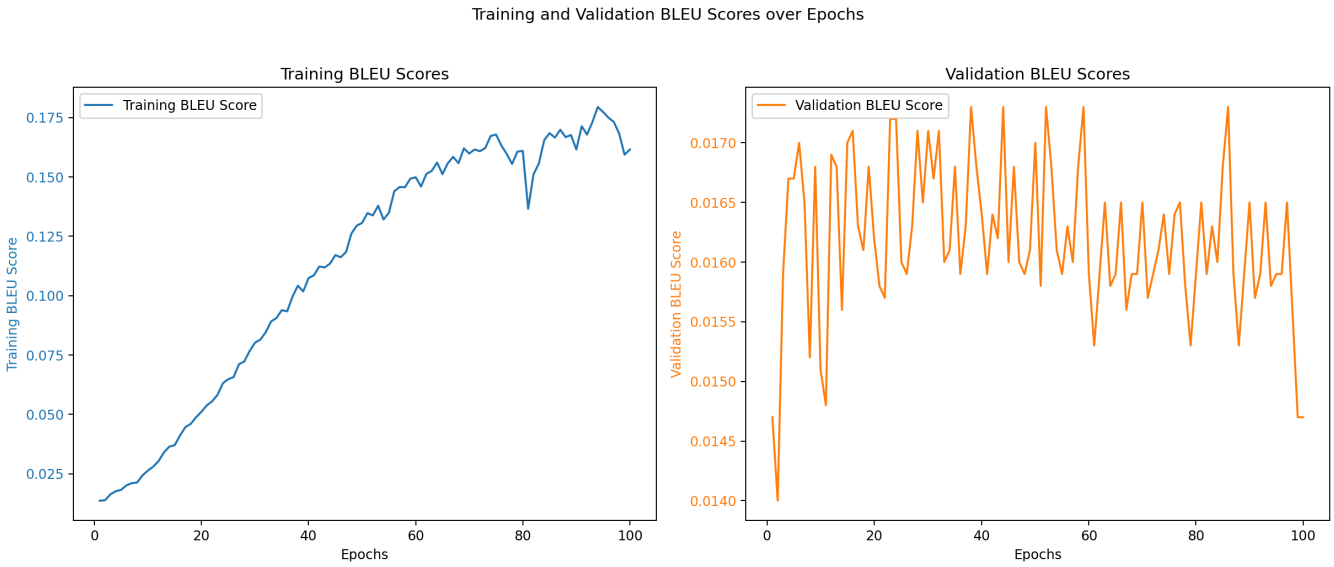
TEST BLUE:0.0158

multiplicative

超参数设置:

teaching_rate = 1
teaching_decay_rate = 0.95
attention = multiplicative
learning_rate = 0.001

实验结果:



TEST BLUE:0.0147

可以看到attention机制中不同对齐函数中,dot product、additive的效果稍好一些，multiplicative效果稍差一些。

在100k的训练集训练模型

multiplicative

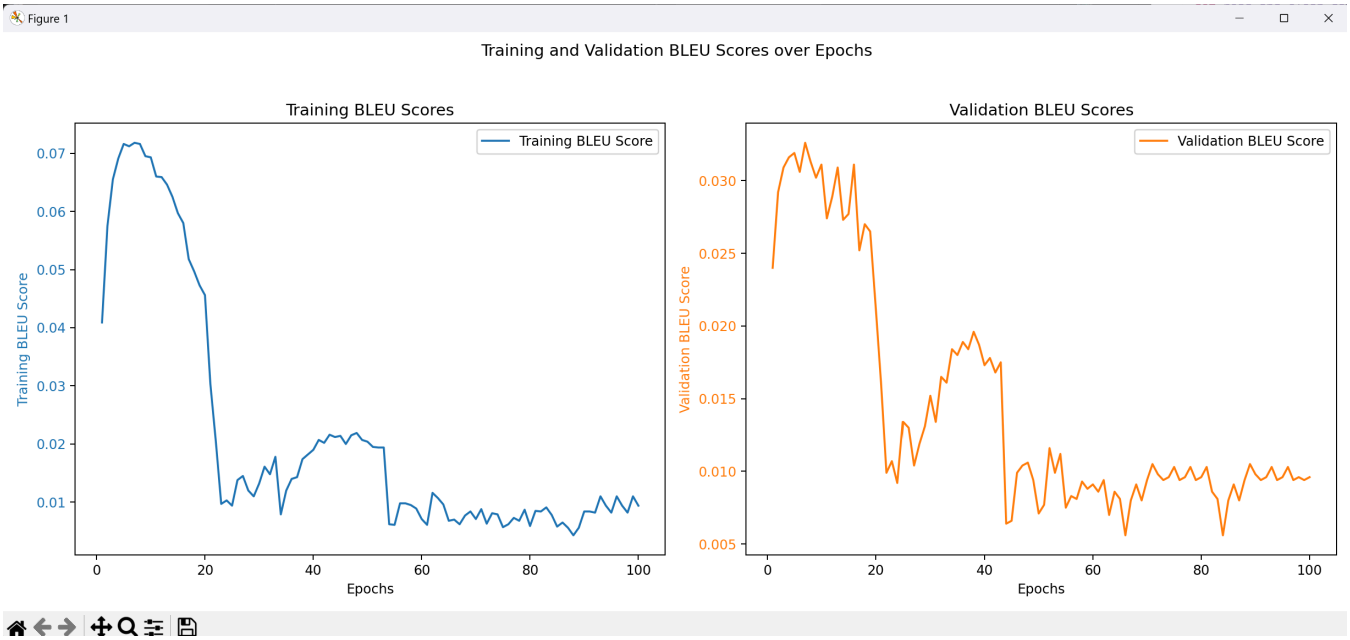
超参数设置:

teaching_rate = 1
teaching_decay_rate = 0.95

attention = additive

learning_rate = 0.001

实验结果:

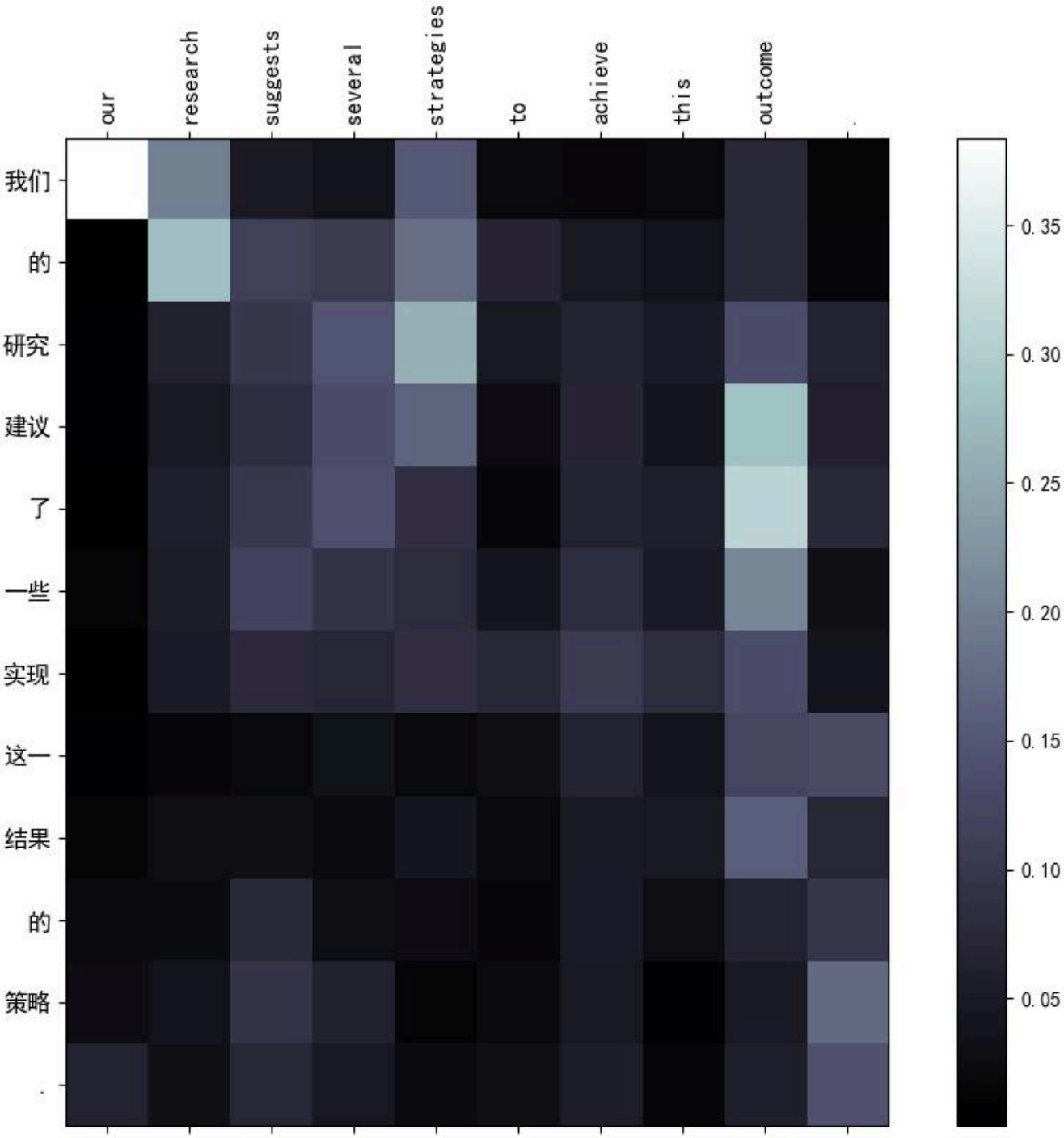


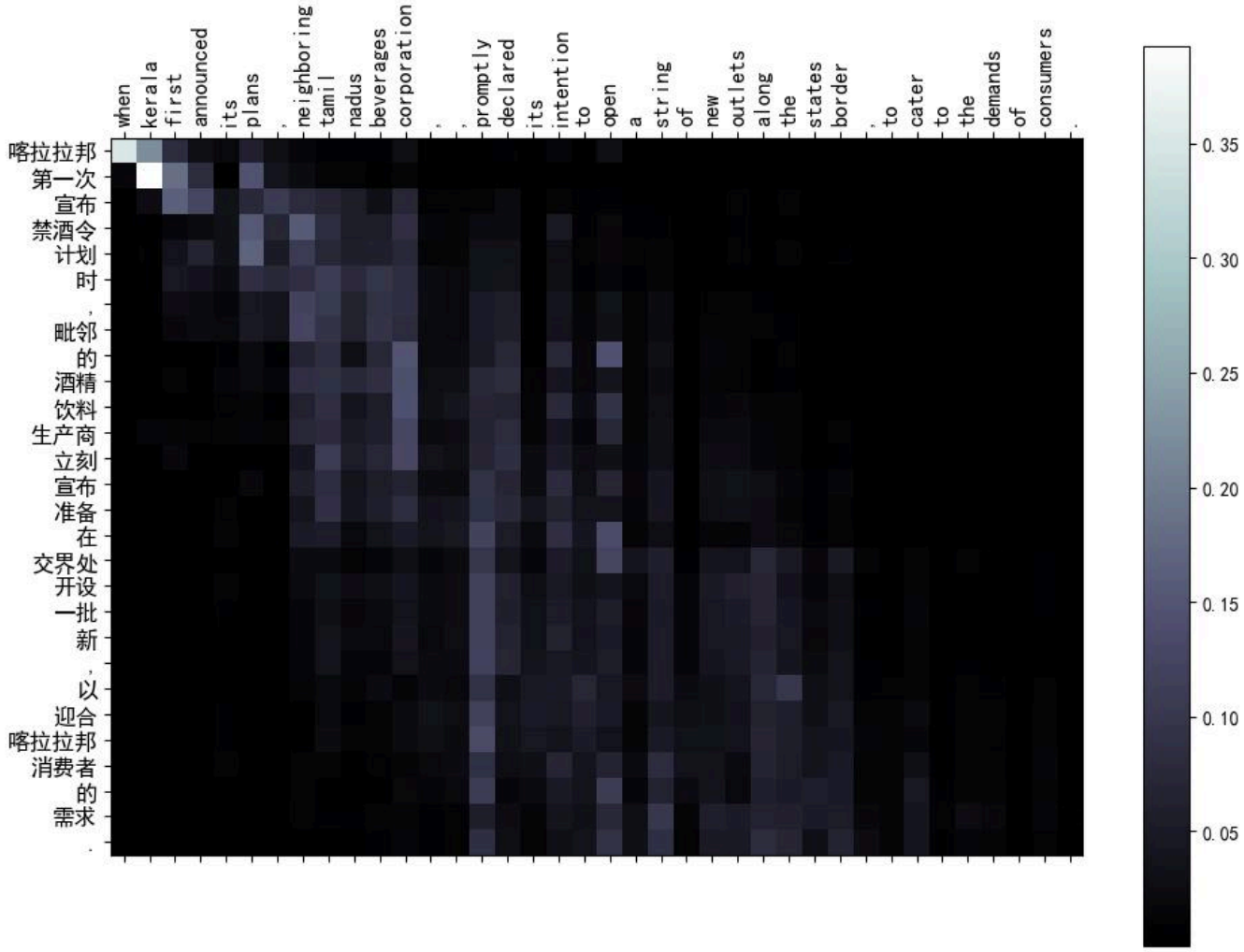
TEST BLUE:0.0280

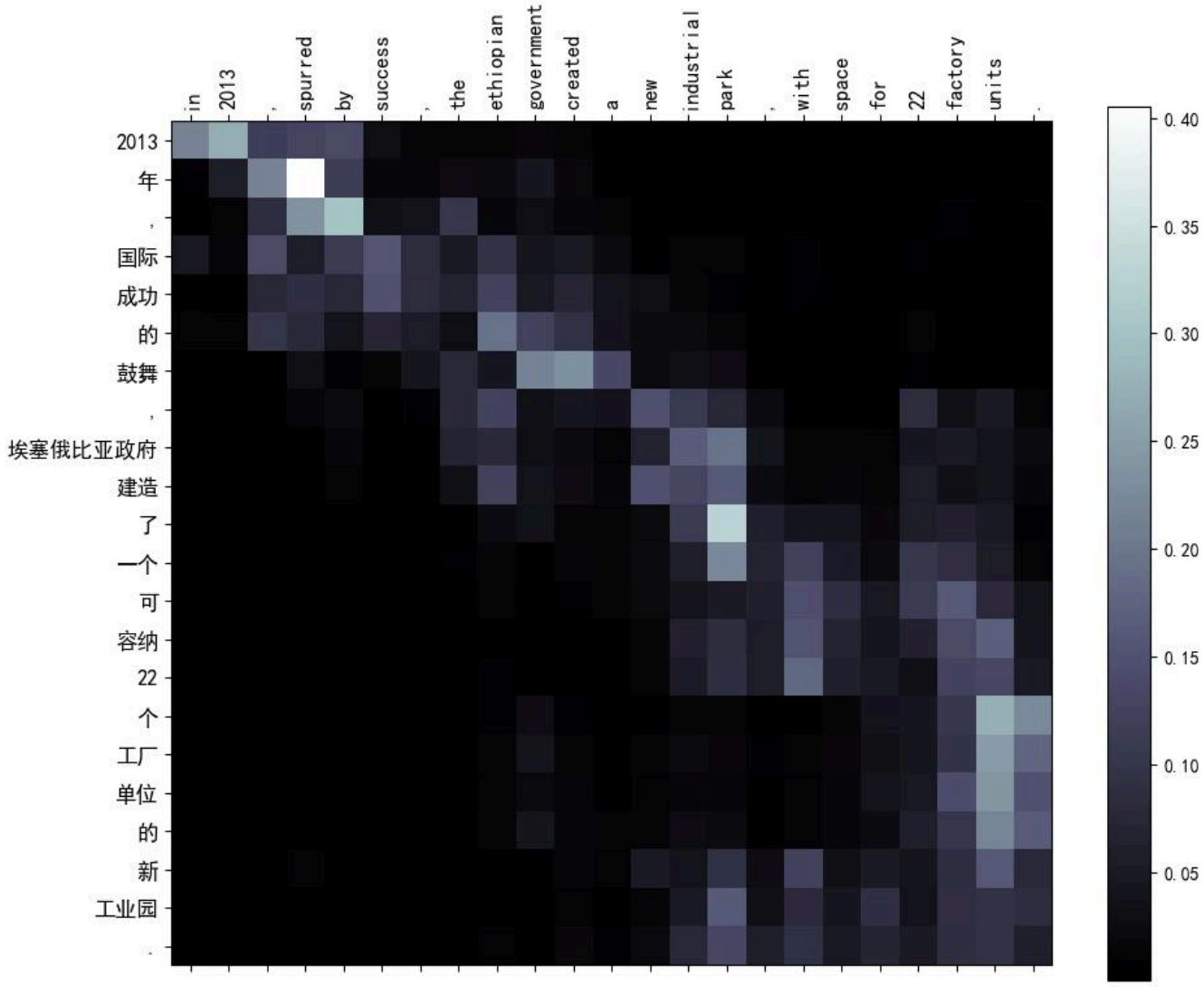
可以看到模型的过拟合的影响在100k的数据集上减小了,但是后面的训练过程中模型的预测能力在训练集和验证集上都下降了， 可能是在后期偏向使用free running导致的训练效果不好。

Attention可视化

随机选出一些样例做Attention可视化:


our
research
suggests
several
strategies
to
achieve
this
outcome
.
我们
的
研究
建议
了
一些
实现
这一
结果
的
策略
.
0.35
0.30
0.25
0.20
0.15
0.10
0.05


when
kerala
first
announced
its
plans
,
neighboring
tamil
nadus
beverages
corporation
,
,
promptly
declared
its
intention
to
open
a
string
of
new
outlets
along
the
states
border
,
to
cater
to
the
demands
of
consumers
.
喀拉拉邦
第一次
宣布
禁酒令
计划
时
,
毗邻
的
酒精
饮料
生产商
立刻
宣布
准备
在
交界处
开设
一批
新
,
以
迎合
喀拉拉邦
消费者
的
需求
.
0.35
0.30
0.25
0.20
0.15
0.10
0.05



可以看到大多数attention能捕捉到输入序列的相关部分，从而减少信息的丢失，提高模型的性能和准确性，但发现还有很多部分会有一点偏差，还有一些捕捉偏差过大。

四、心得体会

在本次的研究实验中，我深入探索了基于神经网络的中英文翻译任务。通过构建和优化编码器-解码器（Encoder-Decoder）架构及引入注意力机制（Attention Mechanism），我获得了多方面的心得体会。

首先，数据预处理阶段是整个实验的第一步。我利用NLTK和结巴分词工具对英文和中文语料进行了分词，并构建了相应的词汇表。这一过程不仅帮助我理清数据的结构和特征，还为后续模型的训练和评估奠定了基础。

其次，模型的构建和优化是实验的核心。我实现了基于GRU的编码器和解码器，并在解码器中引入了注意力机制。这种机制能够有效地捕捉源语言句子中的重要信息，从而提升翻译质量。不同的注意力对齐函数（如点积、加性、乘性）对模型的性能影响显著，通过调整和比较不同的实现方式，我深入理解了它们的作用和效果。

在训练策略方面，我采用了结合Teacher Forcing和Free Running的方法。Teacher Forcing可以加速模型的收敛过程，但如果过度依赖于真实输入，会导致模型在实际推理中表现不佳。通过逐步减少Teacher Forcing的概率，我探索了平衡两者的最佳实践，从而提升了模型在真实数据上的泛化能力和效果。

在实验过程中，我还特别关注了模型评估和结果分析。我使用BLEU分数作为主要的翻译质量评估指标，在训练过程中持续监控和分析模型的表现。通过记录详细的日志和结果，我能够深入了解模型的训练动态，及时调整和优化模型的参数和结构。

最后，通过这次实验，我不仅提升了对神经机器翻译技术的理解和掌握，还积累了丰富的实验经验和解决问题的能力。这些收获将对我未来在自然语言处理领域的研究和工作具有重要的指导意义，帮助我更加深入地探索和应用先进的深度学习技术。

感谢老师和助教一个学期以来的辛苦付出！

五、参考资料

- seq2seq 机器翻译教程 (来自 Pytorch 官方教程，需对数据预处理方式进行更改)
- 分词工具使用:
    - jieba 中文分词工具
    - sentencepiece
- 参考论文:
    - Bahdanau 原版 seq2seq+attention 论文 (ICLR2015)：*Neural Machine Translation by Jointly Learning to Align and Translate*