

# 人工神经网络期末项目

## 中-英机器翻译实验报告

【姓名】谢宇桐

【学号】22336259

【专业】计算机科学与技术

### 一、实验概述

本实验基于Seq2Seq架构实现了中英神经机器翻译系统，探索了**注意力机制中不同对齐函数 (dot product, multiplicative, additive)**、**训练策略 (Teacher Forcing、Free Running)** 和**解码策略 (greedy、beam-search)** 对翻译效果的影响。实验因为电脑硬件配置不足，在这里只使用10k平行语料进行训练，并用500条验证集和200条测试集进行评估，采用BLEU作为主要评价指标。

### 二、模型架构与实现

#### 2.1 Seq2Seq框架搭建

本实验基于经典的Encoder-Decoder架构实现Seq2Seq模型：

```
class Seq2Seq(nn.Module):
    def __init__(self, encoder, decoder, device):
        super().__init__()
        self.encoder = encoder # 编码器组件
        self.decoder = decoder # 解码器组件
        self.device = device # 计算设备

    def forward(self, src, trg, teacher_forcing_ratio=0.5):
        # 编码阶段
        encoder_outputs, hidden = self.encoder(src)

        # 解码阶段
        outputs = torch.zeros(trg_len, batch_size, trg_vocab_size).to(device)
        input = trg[0, :] # 初始输入为<sos>标记

        for t in range(1, trg_len):
            # 使用Attention机制解码
            output, hidden = self.decoder(input, hidden, encoder_outputs)

            # Teacher Forcing策略
            teacher_force = random.random() < teacher_forcing_ratio
            top1 = output.argmax(1)
            input = trg[t] if teacher_force else top1

        return outputs
```

关键组件说明：

1. 编码器(Encoder)：将源语言序列编码为上下文向量

2. 解码器(Decoder): 基于上下文向量生成目标语言序列
3. Teacher Forcing: 训练时以一定概率使用真实标签作为下一时间步输入

## 2.2 Attention机制实现

### 1. 编码器实现

```
class Encoder(nn.Module):
    def __init__(self, input_dim, emb_dim, enc_hid_dim, dec_hid_dim,
dropout):
        super().__init__()
        self.embedding = nn.Embedding(input_dim, emb_dim)
        self.rnn = nn.GRU(emb_dim, enc_hid_dim, bidirectional=True) # 双向
GRU
        self.fc = nn.Linear(enc_hid_dim * 2, dec_hid_dim) # 连接层
        self.dropout = nn.Dropout(dropout)

    def forward(self, src):
        embedded = self.dropout(self.embedding(src))
        # 双向GRU返回所有时间步的输出和最后隐藏状态
        outputs, hidden = self.rnn(embedded)

        # 合并双向隐藏状态
        hidden = torch.tanh(self.fc(torch.cat((hidden[-2, :, :], hidden[-1,
        :, :]), dim=1)))

        return outputs, hidden # 返回所有时间步输出和最终隐藏状态
```

### 2. Attention机制核心

在注意力机制中, 对齐函数用于计算编码器隐藏状态和解码器当前状态之间的相关性权重:

- **加性注意力 (Additive):** 使用全连接网络计算非线性组合

```
class Attention(nn.Module):
    def __init__(self, enc_output_dim, dec_hid_dim,
attn_type='additive'):
        super().__init__()
        self.enc_output_dim = enc_output_dim
        self.dec_hid_dim = dec_hid_dim
        self.attn_type = attn_type

        if attn_type == 'additive':
            self.attn = nn.Sequential(
                nn.Linear(self.dec_hid_dim + self.enc_output_dim,
dec_hid_dim),
                nn.Tanh(),
                nn.Linear(dec_hid_dim, 1)
            )
            # ...

        self.softmax = nn.Softmax(dim=1)
```

```
def forward(self, hidden, encoder_outputs):
    src_len = encoder_outputs.shape[0]
    batch_size = hidden.size(0)

    if self.attn_type == 'additive':
        hidden = hidden.unsqueeze(1).repeat(1, src_len, 1)
        encoder_outputs = encoder_outputs.permute(1, 0, 2)
        combined = torch.cat((hidden, encoder_outputs), dim=2)
        energy = self.attn(combined).squeeze(2)
```

特点:

- 引入非线性激活函数(tanh)
- 参数量更大, 表达能力更强
- 数学表达式: 对于每个查询向量  $h$  和每个键向量  $s$ , 计算一个加权得分:

$$e_{ij} = v^T \tanh(W_1 h_i + W_2 s_j)$$

$h$  是查询向量 (通常是解码器的当前状态),  $s$  是键向量 (通常是编码器的输出);  $W_1$  和  $W_2$  是学习到的权重矩阵;  $v$  是一个可学习的参数, 用于对最终得分进行加权。

- **点积注意力 (Dot Product):** 直接计算两个向量的点积作为相似度分数

```
class Attention(nn.Module):
    def __init__(self, enc_output_dim, dec_hid_dim,
                 attn_type='additive'):
        # ...
        elif attn_type == 'dot':
            assert self.enc_output_dim == dec_hid_dim, \
                "For dot product attention, enc_output_dim must equal dec_hid_dim"

        self.softmax = nn.Softmax(dim=1)

    def forward(self, hidden, encoder_outputs):
        # ...
        elif self.attn_type == 'dot':
            encoder_outputs = encoder_outputs.permute(1, 0, 2)
            hidden = hidden.unsqueeze(1)
            energy = torch.bmm(encoder_outputs, hidden.transpose(1, 2)).squeeze(2)
```

特点:

- 计算简单高效
- 要求编码器和解码器隐藏层维度相同
- 数学表达:

$$e_{ij} = h_i^T s_j$$

$e_{ij}$  表示 第  $i$  个查询向量与第  $j$  个键向量的点积。这个点积的数值大小反映了第  $i$  个查询向量  $q_i$  与第  $j$  个键向量  $k_j$  的相似程度:

如果两个向量方向相近, 点积较大, 表示它们**高度相关, 注意力权重会较高**;

如果两个向量方向不相关, 点积较小甚至接近零, 表示它们**关系较弱, 注意力权重较低**。

- 乘性注意力 (Multiplicative): 引入可学习权重矩阵转换编码器输出

```
# 初始化
elif attn_type == 'multiplicative':
    self.W = nn.Linear(self.enc_output_dim, dec_hid_dim,
bias=False)
# 向前传播
elif attn_type == 'multiplicative':
    encoder_outputs = encoder_outputs.permute(1, 0, 2) #
[batch_size, src_len, enc_hid_dim*2]
    projected_encoder = self.W(encoder_outputs) # [batch_size,
src_len, dec_hid_dim]
    hidden = hidden.unsqueeze(1) # [batch_size, 1, dec_hid_dim]
    # 添加缩放因子
    d_k = encoder_outputs.size(-1)
    energy = torch.bmm(projected_encoder, hidden.transpose(1, 2))
/ np.sqrt(d_k)
    energy = energy.squeeze(2)
```

特点:

- 通过权重矩阵W进行线性变换
- 添加缩放因子防止梯度消失

$$\sqrt{d_k}$$

- 数学表达: 计算编码器第 i 个时间步的隐藏状态  $h_i$  和解码器当前时间步的隐藏状态  $s_j$  之间的注意力权重  $e_{ij}$

$$e_{ij} = \frac{(Wh_i)^T s_j}{\sqrt{d_k}}$$

### 3. 带Attention的解码器

```
class Decoder(nn.Module):
    def __init__(self, output_dim, emb_dim, enc_hid_dim, dec_hid_dim,
dropout, attention):
        super().__init__()
        self.output_dim = output_dim
        self.attention = attention # 注意力模块
        self.embedding = nn.Embedding(output_dim, emb_dim)

        # GRU输入维度: 词嵌入+编码器隐藏状态
        self.rnn = nn.GRU(emb_dim + enc_hid_dim, dec_hid_dim)

        # 输出层: 组合RNN输出、注意力上下文和词嵌入
        self.fc_out = nn.Linear(dec_hid_dim + enc_hid_dim + emb_dim,
output_dim)
        self.dropout = nn.Dropout(dropout)
        self.layer_norm = nn.LayerNorm(dec_hid_dim) # 层归一化

    def forward(self, input, hidden, encoder_outputs,
return_attention=False):
        # 嵌入层
        embedded = self.dropout(self.embedding(input.unsqueeze(0)))
```

```

# 计算注意力权重
a = self.attention(hidden, encoder_outputs)
if return_attention:
    attention_weights = a.clone()
# 计算加权上下文向量
weighted = torch.bmm(a.unsqueeze(1), encoder_outputs.permute(1, 0,
2))

weighted = weighted.permute(1, 0, 2)

# 拼接词嵌入和上下文向量作为RNN输入
rnn_input = torch.cat((embedded, weighted), dim=2)

# 通过RNN
output, hidden = self.rnn(rnn_input, hidden.unsqueeze(0))
hidden = self.layer_norm(hidden.squeeze(0))

# 预测输出: 组合RNN输出、上下文和词嵌入
prediction = self.fc_out(torch.cat((output.squeeze(0),
                                     weighted.squeeze(0),
                                     embedded.squeeze(0)), dim=1))

if return_attention:
    return prediction, hidden, attention_weights

return prediction, hidden

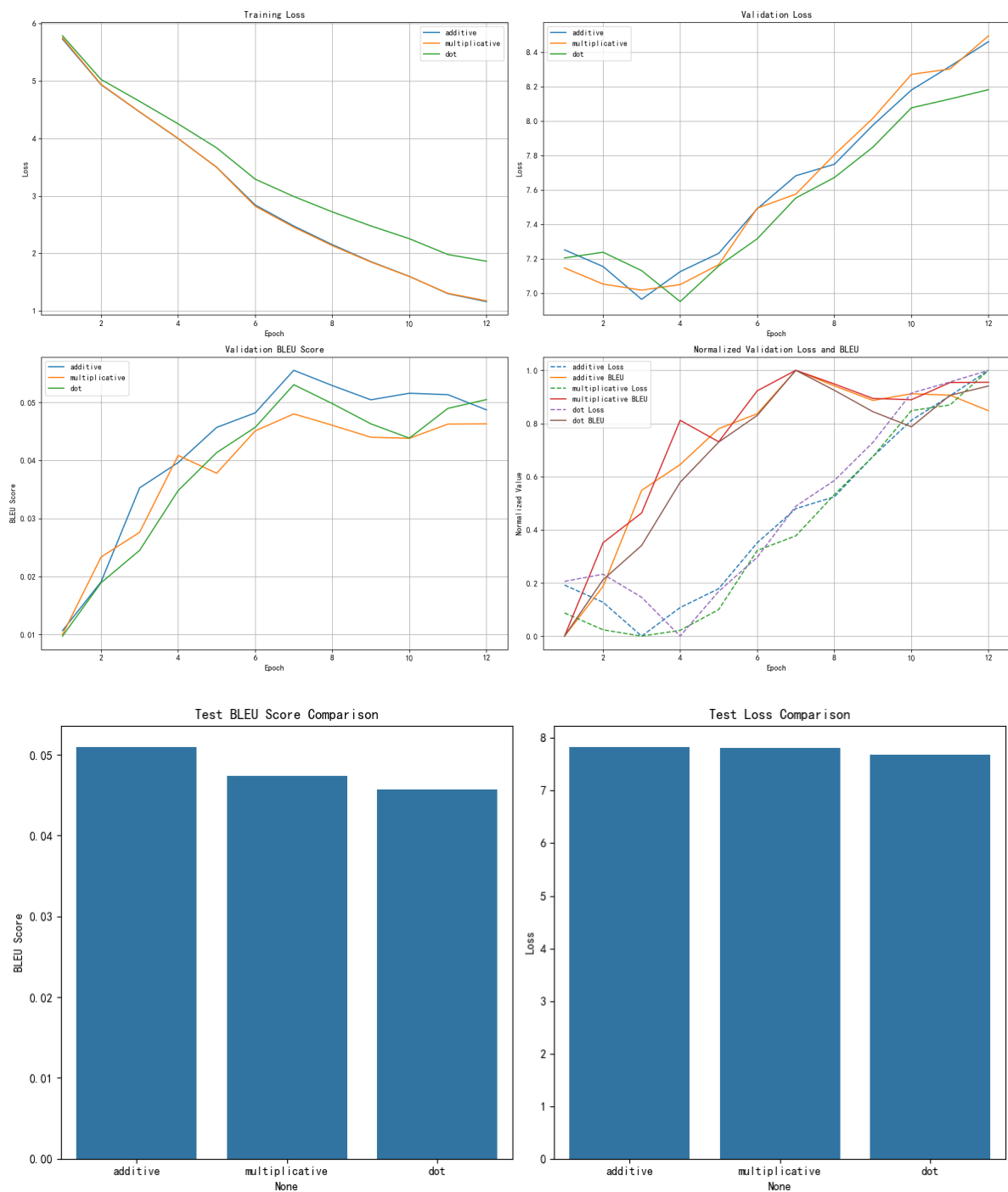
```

## 三、实验结果分析

在实验过程中，我发现Additive+Teacher Forcing+Greedy这个组合是最佳组合，所以在不同指标对比过程中，其它标准指标都以此为准。

### 3.1 不同注意力机制对比

| 注意力类型          | 测试损失  | BLEU  |
|----------------|-------|-------|
| Additive       | 7.835 | 0.051 |
| Multiplicative | 7.820 | 0.047 |
| Dot Product    | 7.688 | 0.046 |

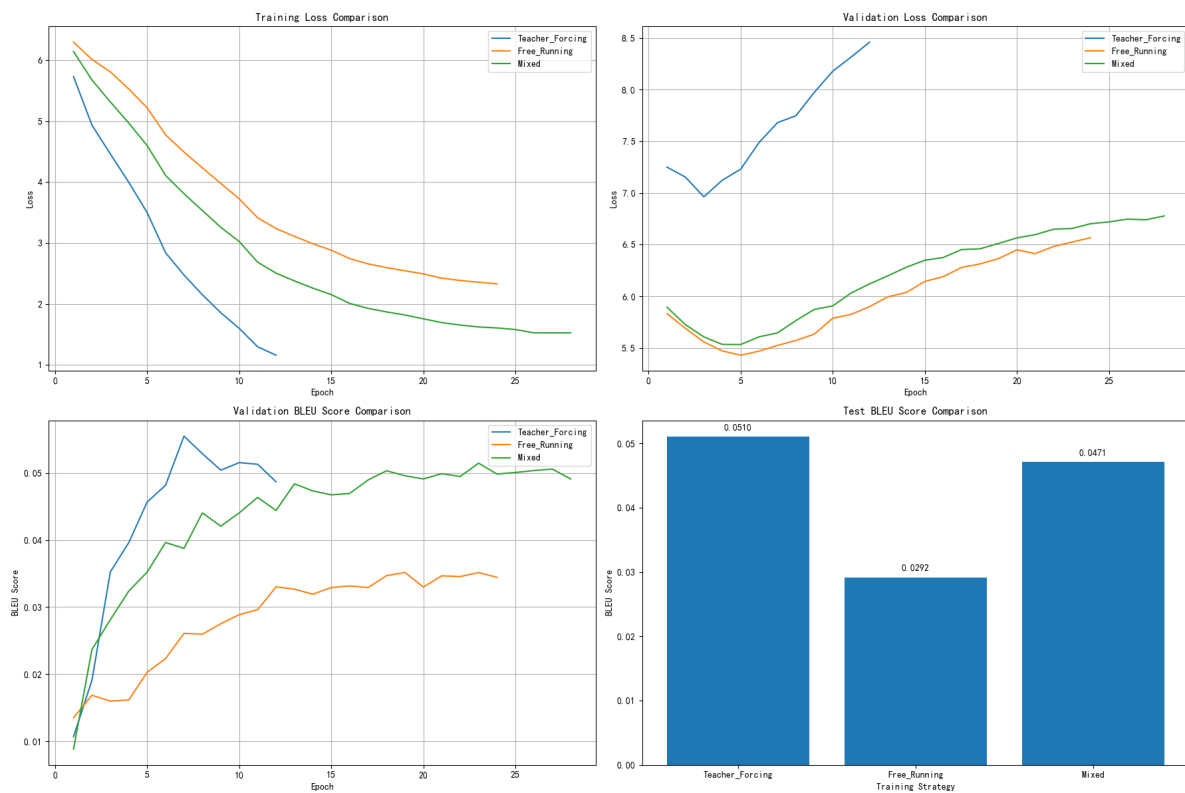


分析：

- Additive注意力获得最高BLEU分数(0.051)，因其通过全连接层学习更复杂的对齐关系
- Dot Product注意力测试损失最低(7.688)，但BLEU分数略低于Additive
- Multiplicative注意力在复杂度和性能间取得平衡，BLEU分数居中

## 3.2 不同训练策略对比

| 训练策略            | 测试损失  | BLEU  |
|-----------------|-------|-------|
| Teacher Forcing | 7.835 | 0.051 |
| Free Running    | 6.719 | 0.029 |
| Mixed           | 7.085 | 0.047 |



分析:

- Teacher Forcing收敛最快, 获得最高BLEU(0.051), 但可能暴露偏差问题, 测试损失较高
- Free Running训练最稳定, 测试损失最低(6.719), 但BLEU分数也最低(0.029), 收敛速度慢
- Mixed策略平衡了训练速度和稳定性, 在BLEU和测试损失上都表现居中, 平衡了两种策略的优点

### 3.3 不同解码策略对比

在测试集上的BLEU分数对比:

测试集Greedy解码BLEU: 0.0468

测试集Beam Search BLEU: 0.0457

解码策略对比示例:

源句子: 1929年还是1989年?

Greedy解码: or 1989 or 1989 ?

Beam Search: <unk> or 1989 ?

解码策略对比图已保存为 'decoding\_strategy\_comparison.png'

训练和解码结果图已保存为 'training\_and\_decoding\_results.png'

实验完成!

分析:

- 从分数来看Greedy比Beam Search( $\text{beam\_width}=5$ )解码获得更高BLEU分数, 但差异并不明显。Greedy波动更大, Beam Search相对更稳定

- Beam Search通过保留多个候选序列，减少局部最优导致的错误
- Greedy解码速度更快，大概四倍以上，适合实时应用场景

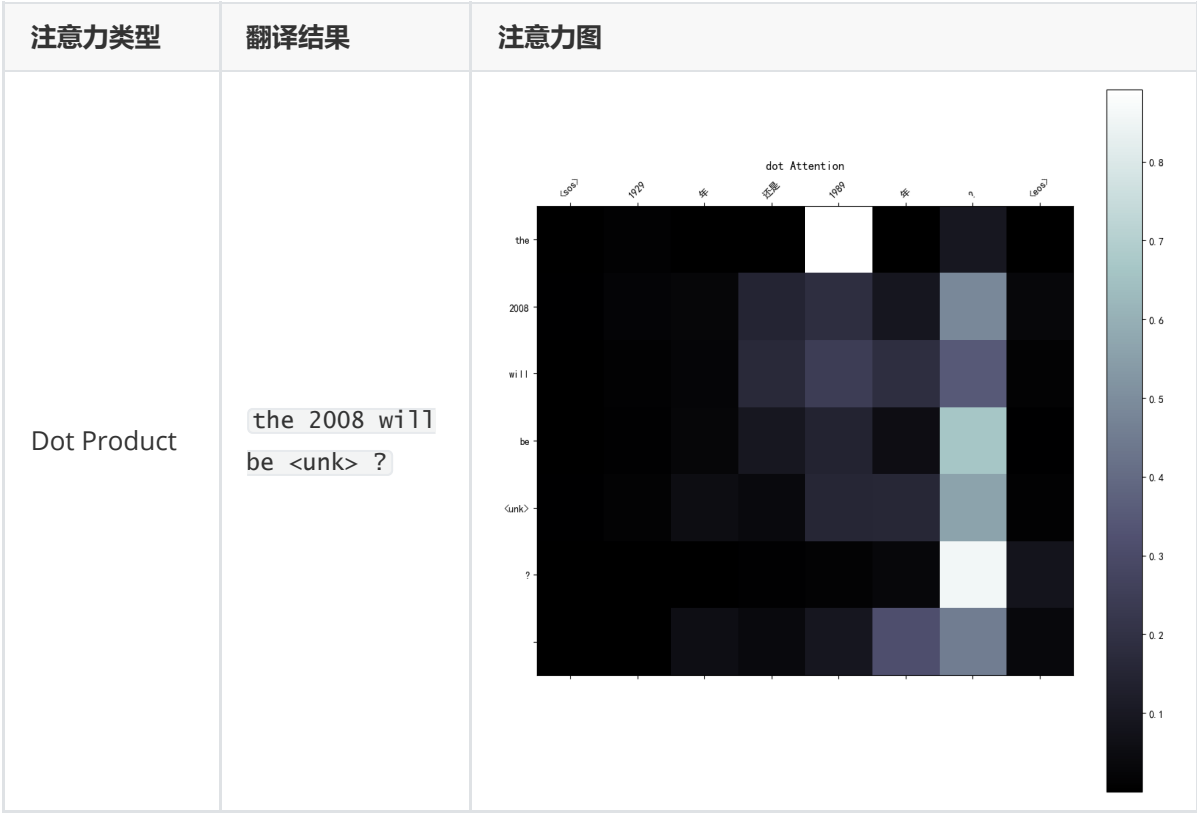
## 四、注意力可视化分析

### 4.1 不同注意力机制对比

示例句子: "1929年还是1989年?"

| 注意力类型          | 翻译结果   | 注意力图   |
|----------------|--|--|
| Additive       | <div>&lt;unk&gt; or 1989 ?</div>                 | <p>The heatmap for Additive Attention shows the model's focus. The x-axis represents the source sentence tokens: &lt;unk&gt;, 1929, 年, 还是, 1989, 年, ?, &lt;unk&gt;. The y-axis represents the target sentence tokens: &lt;unk&gt;, or, 1989, ?. The color scale ranges from 0.1 (dark) to 0.6 (light). High attention is visible for the target token 'or' on the source token '还是' (approx. 0.55) and for the target token '1989' on the source token '1989' (approx. 0.45).</p>                        |
| Multiplicative | <div>what 1989<br/>came from or<br/>1989 ?</div> | <p>The heatmap for Multiplicative Attention shows the model's focus. The x-axis represents the source sentence tokens: &lt;unk&gt;, 1929, 年, 还是, 1989, 年, ?, &lt;unk&gt;. The y-axis represents the target sentence tokens: what, 1989, came, from, 1989, or, 1989, ?. The color scale ranges from 0.1 (dark) to 0.8 (light). High attention is visible for the target token '1989' on the source token '1989' (approx. 0.75) and for the target token 'or' on the source token '还是' (approx. 0.65).</p> |



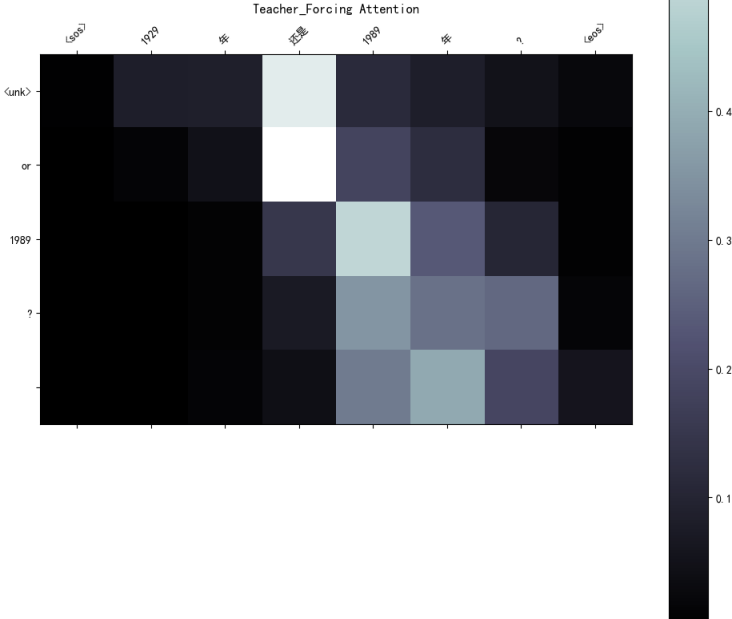
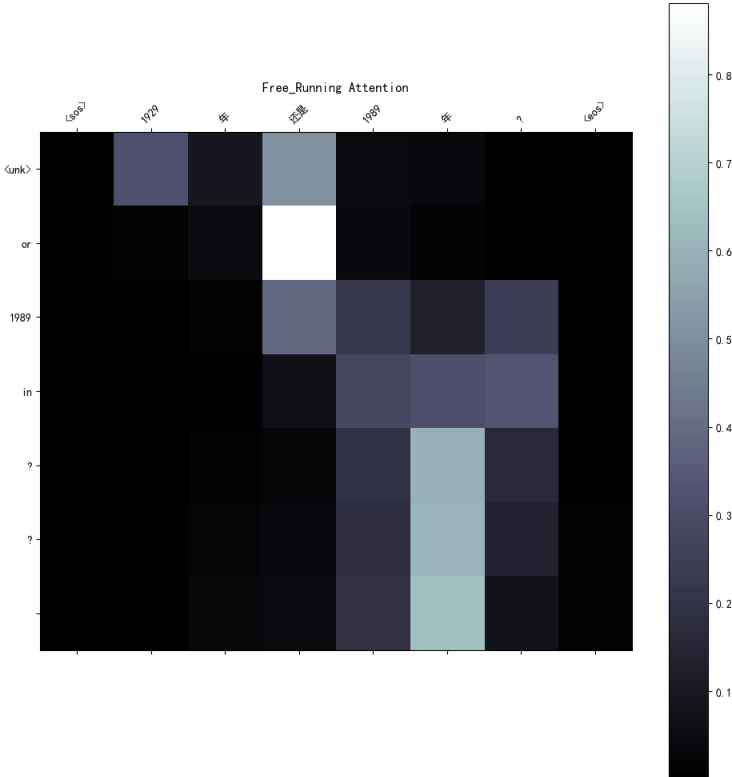


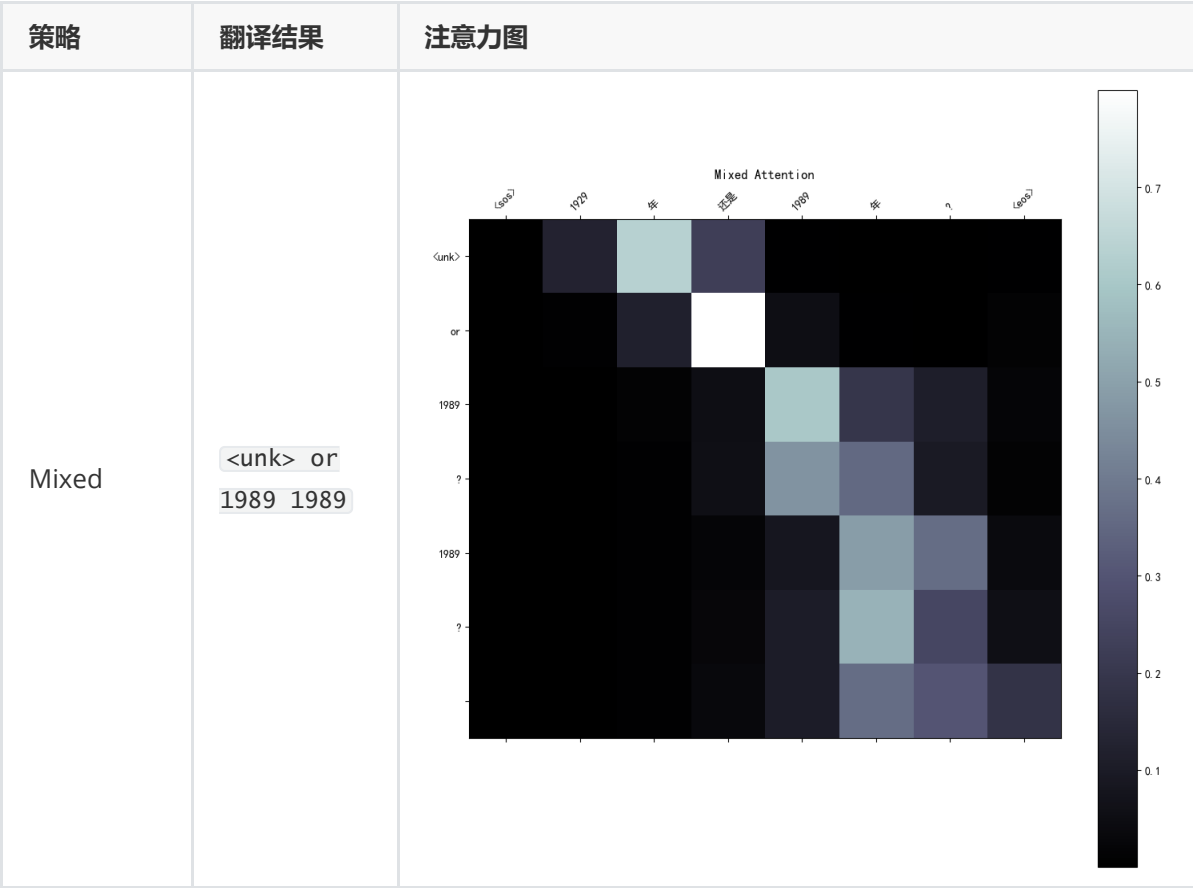
分析：

- Additive注意力在"还是"和"or"之间建立了强关联，翻译结果在三者对比看来最准确
- Multiplicative注意力产生重复输出"1989"，显示注意力分配不够集中
- Dot Product注意力未能正确对齐关键信息，导致错误翻译

## 4.2 不同训练策略注意力对比

同一句子在不同训练策略下的注意力图：

| 策略              | 翻译结果             | 注意力图   |
|-----------------|------------------|--|
| Teacher Forcing | <unk> or 1989    | <p>Teacher_Forcing Attention</p>  <p>The heatmap for Teacher_Forcing Attention shows a grid of attention weights. The x-axis (source) tokens are &lt;unk&gt;, 1979, 年, 还是, 1989, 年, 年, &lt;unk&gt;. The y-axis (target) tokens are &lt;unk&gt;, or, 1989, 年, 年, 年. The color scale ranges from 0.1 (dark) to 0.6 (light). High attention (lighter colors) is concentrated on the source token '1989' and the target token 'or', indicating that the model is focusing on the correct word in the source sequence to produce the correct word in the target sequence.</p> |
| Free Running    | <unk> or 1989 in | <p>Free_Running Attention</p>  <p>The heatmap for Free_Running Attention shows a grid of attention weights. The x-axis (source) tokens are &lt;unk&gt;, 1979, 年, 还是, 1989, 年, 年, &lt;unk&gt;. The y-axis (target) tokens are &lt;unk&gt;, or, 1989, in, 年, 年, 年. The color scale ranges from 0.1 (dark) to 0.8 (light). High attention (lighter colors) is concentrated on the source token '1989' and the target token 'in', indicating that the model is focusing on the correct word in the source sequence to produce the correct word in the target sequence.</p>  |



- 分析：
- Teacher Forcing产生更简洁准确的翻译和集中的注意力分布
  - Free Running注意力更分散，产生多余词汇
  - Mixed策略产生重复输出"1989"，显示两种策略混合后的副作用

### 4.3 不同解码策略对比

同一句子在不同解码策略下的注意力图：

| 策略          | 翻译结果              |
|-------------|-------------------|
| Greedy      | or or 1989 ?      |
| Beam Search | <unk> or 1989 ? ? |

- Beam Search产生未登录词，Greedy重复输出
- 翻译结果不佳表明模型未能理解年份对比关系，这更多是模型理解能力问题而非解码策略问题。

## 五、训练技巧与心得

### 5.1 有效训练技巧

#### 1. 动态学习率调整：

```
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=5,  
gamma=0.5)
```

每5个epoch将学习率减半，加速后期收敛

#### 2. 早停机制：

因为训练时间较长，所以我在实验中学了一个能够缩减训练时间的方法：

```
if valid_bleu > best_bleu:  
    best_bleu = valid_bleu  
    no_improve = 0  
else:  
    no_improve += 1  
    if no_improve >= 5: break
```

在验证集BLEU连续5轮无提升时停止训练，这样能够防止无效训练和过拟合

#### 3. 梯度裁剪：

```
torch.nn.utils.clip_grad_norm_(model.parameters(), clip=3)
```

防止梯度爆炸，提升训练稳定性

### 5.2 心得体会

这次中英机器翻译实验让我收获颇丰。从最初的不理解，到后来能动手实现一个完整的Seq2Seq模型，过程中学到了很多实战经验。

在这个过程中，我学习了注意力机制是Seq2Seq模型的核心，不同对齐函数对模型性能有显著影响。Additive注意力在本实验中表现最佳；训练策略的选择至关重要。Teacher Forcing能快速收敛但可能导致暴露偏差，Free Running更稳定但收敛慢，混合策略是一个不错的折中方案，但也会有副作用；解码策略方面，虽然本实验中Greedy和Beam Search的BLEU分数相近，但Beam Search产生的翻译质量更高，避免了重复输出等问题。

实验过程中遇到的验证损失上升而BLEU下降的问题，主要是由于模型容量不足和数据集规模有限导致的。更大的模型和更多的训练数据可能会改善这一问题。注意力机制中不同对齐函数、不同训练策略和不同解码策略对翻译效果的影响，以及他们自身的区别。我理解了模型的底层逻辑，学会了怎么看attention权重图等技巧。这对我对人工神经网络有了更深的认知与理解。

这次实验让我对神经机器翻译的各个环节有了更直观的认识，也锻炼了我分析模型性能、调试超参数的能力。我相信这些经验对今后若从事自然语言处理相关工作将会非常有帮助。