机器翻译实验报告

17341146 王程钥

1 简介

机器翻译是自然语言处理的一个重要问题，在工业界有着广泛的应用。机器翻译指使用计算机将一个句子或一段话从一种语言翻译到另一种语言。传统的机器翻译算法包括基于规则的机器翻译算法和基于统计的机器翻译算法(比如 IBM 系列)。

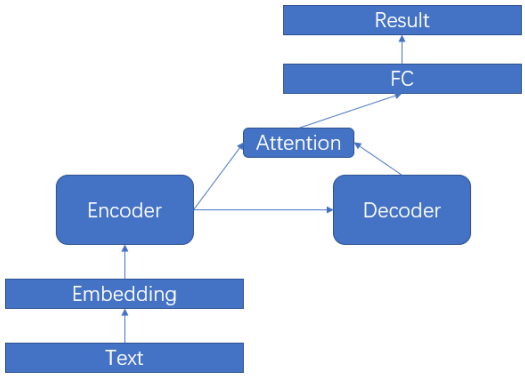
近年来随着深度学习的发展，基于深度神经网络的机器翻译逐渐开始流行。常见的神经机器翻译算法一般基于 encoder-decoder 结构，始于循环神经网络对信息进行编码解码，并使用 attention 等机制进行优化。近年来基于 transformer 的机器翻译算法更是突破了循环神经网络的限制，完全基于 self-attention，取得了更好的性能。

本次大作业我实现了一个基于由 LSTM 组成的编码器解码器(encoder-decoder)结构的机器翻译模型，使用 attention 机制加强编码器和解码器的联系，并通过动态调节 teach forcing rate 等方法提高模型的泛化度。最终，在只使用了 8000 组数据进行训练的情况下，模型在训练集上最高可以取得 0.5 左右的 blue 值，在测试集上可以取得 0.02 左右的 blue 值。使用 80000 组数据进行训练后，在测试集上可以获得 0.05 左右的 blue 值。

2 算法原理

2.1 算法框架

本文主体采用 encoder-decoder 的框架，首先使用词嵌入工具将源语言句子从 one-hot 向量嵌入到词向量。在编码器解码器部分，使用一个双向 LSTM 作为编码器对源语言信息进行编码，后接一个单向 LSTM 作为解码器对编码信息进行解码。解码器的输出取决于编码器输出和解码器隐状态之间的关系，使用 attention 机制进行计算。简单的算法框架如图 1 所示。



(图 1)

2.2 文本预处理

文本数据预处理的部分，首先去除文本中的标点符号，后进行分词。对于中文使用 jieba 库进行分词，对于英文使用 nltk 库进行分词。分词后在每个句子的前面加‘<BOS>’

符号表示开始，在句子的后面加一个‘<EOS>’符号表示结束。分词后使用训练集的文本对中文(源语言)和英文(目标语言)分别构建词表，并使用 word2vec 训练词向量。

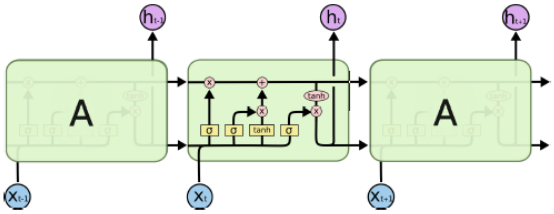
在训练的时候是按照 batch 进行训练的，每个 batch 内的文本长短不一，因此需要使用‘<PAD>’进行填充，使每个 batch 内文本长度相等，可以训练。

2.3 循环神经网络

循环神经网络(RNN)主要用于处理序列数据，比如文本和视频。它基于不同神经元共享参数的思想，对句子的信息进行编码，保留和当前语境相关的信息并丢弃无关信息。权值共享使得它能够使用较少的模型参数训练大量的数据。

根据反向传播的原理，传统的 RNN 下梯度会在模型中多次反向传播，这很容易造成梯度爆炸或梯度消失，进而导致模型很难训练。同时传统的 RNN 很难访问距离当前时刻较远的信息。长短期记忆网络(LSTM)的引入一定程度上解决了这些问题。

LSTM[2]的最大贡献是引入了门机制。如图 2，LSTM 包含两个流 h 和 c，h_i是短期记忆流，c_i是长期记忆流。LSTM 从左往右有三个门，用于更新状态 h 和 c。



(图 2)

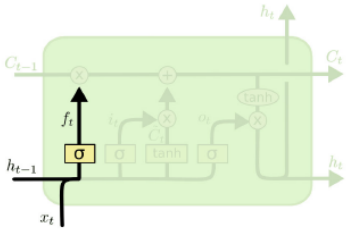
遗忘门(Forget Gate，图 3)让当前时刻 t 的输入x_t通过一个全连接层后接激活函数映射到 0 或 1，与保存的历史状态c_{i-1}点乘，目的是根据输入遗忘部分历史状态。

$$f_t = \sigma(W_f * [h_{t-1}, x_t] + b_f) \qquad (1)$$

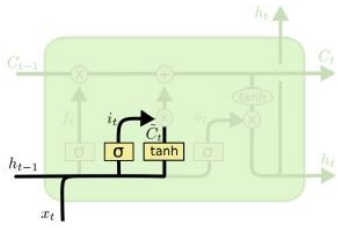
输入门(Input Gate，图 4)让输入x_t信息经过两个不同的全连接层后接激活函数得到两个流i_t和$\widetilde{C_t}$。这两个流点乘用于表示当前状态有多少需要被保存到历史信息中。

$$i_t = \sigma(W_i * [h_{t-1}, x_t] + b_i) \qquad (2)$$

$$\widetilde{C_t} = tanh(W_C * [h_{t-1}, x_t] + b_C) \qquad (3)$$



(图 3)



(图 4)

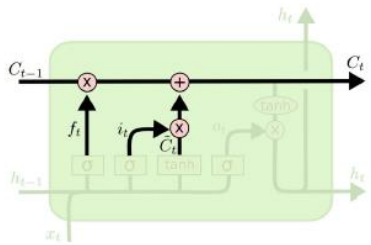
长期记忆状态c_t(图 5)根据遗忘门和输入门更新，将遗忘后的历史状态加上输入状态，得到新的历史状态。

$$C_t = f_t * C_{t-1} + i_t * \widetilde{C_t} \qquad (4)$$

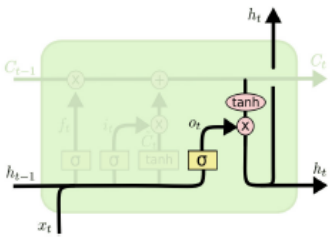
输出门(Output Gate，图 6)将 t 时刻的输入信息x_t接全连接层与激活函数后与接了激活函数的历史信息点乘，得到对于 t 时刻的短期记忆状态h_t。

$$o_t = \sigma(W_o * [h_{t-1}, x_t] + b_o) \qquad (5)$$

$$h_t = o_t * \tanh(C_t) \qquad (6)$$



（图 5）



（图 6）

本次实验测试了单向 LSTM 和双向 LSTM，对于单向 LSTM 取每个时刻的输出h_t，用第 t 个时刻的输出h_t预测第 t+1 时刻的输入x_{t+1}。对于双向 LSTM 将两个方向的输出相加，取正向第 t 个时刻的输出h_t和反向第 t+2 个时刻的输出h_{t+2}的和预测第 t+1 个时刻的输入x_{t+1}

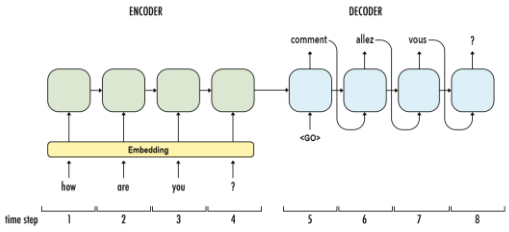
2.4 编码器与解码器

编码器(Encoder)使用了一个双向双层的 LSTM。将嵌入后的词向量按照时间顺序依次输入即可。Encoder 最后的输出包含三个向量 output,(hn,cn)，分别表示每个 timestep 的长期记忆(hn)的输出和最后一个时刻长期记忆(hn)、短期记忆(cn)的输出。

解码器(Decoder)使用了一个单向单层 LSTM。为了尽可能地使用 Encoder 编码的状态，使用 Encoder 最后一个 timestep 输出的 hidden 作为 Decoder 第一个 timestep 的 hidden 状态的输入。

在每个 timestep，Decoder 的输入有两种方式，一种策略直接使用上一个 timestep 的输出作为当前 timestep 的输入，另一种策略是 Teacher forcing，使用当前 timestep 数据的标签，即 ground truth 作为输入。在训练的时候使用一个概率 P，根据概率决定使用上一个 timestep 的输出还是 ground truth。在测试的时候没有 ground truth 标签，所以全部使用上个 timestep 的输出。

编码器解码器的模型简图见图 7。



（图 7）

2.5 Attention 机制

RNN base 的模型有一个问题，虽然它理论上可以编码整个句子的信息，但实际上距离较远的信息很容易被遗忘。即使引入了长短期记忆网络(LSTM)这个问题也还是得不到解决。对于 Decoder 的每个 timestep 的输出，我们需要将它和源语言句子进行联系，知道它对应 Encoder 哪个 timestep 的信息，从而对编码信息进行加权，更好地对信息进行解码。因此我们引入了 attention 机制。

Attention 机制首先根据当前 timestep 时 decoder 的输出向量 ht 和 encoder 每个 timestep 的输出 hs，将 hs 过一个全连接层，并使用公式(7)计算出当前 timestep 和源语言 timestep 的联系，并使用 softmax(8)进行归一化，获取注意力权重(attention weight)。

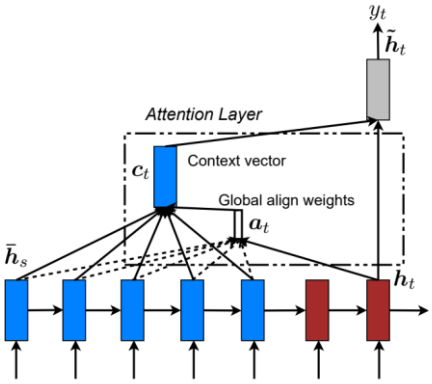
$$\text{score}(h_t, h_s) = \text{h}_t^T W h_s \qquad (7)$$

$$\alpha_{ts} = \frac{e^{score(h_t,h_s)}}{\sum_{s=1}^{n} e^{score(h_t,h_s)}} \qquad (8)$$

后按照公式(9)将 attention weight 和 encoder 的输出相乘，获取上下文向量(context vector)。最后根据公式(10)将该向量和 decoder 当前 timestep 输出的短时记忆向量 ht 拼接在一起，过一个全连接层并使用 tanh 函数激活后得到修正过的短时记忆向量 ht'。ht'和 decoder 的输出 ct 一起作为下一个 timestep 时 decoder 的隐藏状态。Attention 机制的相关原理图见图 8。

$$c_t = \sum_s \alpha_{ts} * h_s \qquad (9)$$

$$h'_t = tanh(W_c[c_t; h_t]) \qquad (10)$$



（图 8）

2.6 使用 Beam search 进行测试

在测试的时候采用集束搜索(beam search)的办法。不用于每次选最优的贪心法，Beam search 在进行搜索时对于每个 timestep 会存下得分前 K 高的文本作为候选文本，传入下一个 timestep。具体的搜索过程如下。

(1)得到 Decoder 第一个输出的概率分布，选取概率前 K 高的词作为候选词。

(2)将这 K 个词分别座位 Decoder 的输入，每个输入得到 K 个候选词，概率为之前概率的和当前单词概率的乘积。

(3)如果候选句的最后一个词为<EOS>，则将该翻译句作为一个候选结果。

(4)否则，在这 K*K 个候选词中选择得分最高的 K 个，将其传入下一个 timestep，回到步骤(2)。直到候选结果数目超过 K 或 timestep 达到上届。

(5)找到得分最高的翻译结果，将其返回

和 trivial 的贪心法相比，beam search 方法能够不被局部最优结果影响，找到更优秀的翻译结果。

3 实现过程

3.1 框架的搭建

我使用了 pytorch 作为实验框架。使用 nn.LSTM 作为 Encoder，因为 Encoder 可以一次传入多个 timestep 的信息。而使用 nn.LSTMCell 作为 Decoder，因为 Decoder 需要每个 timestep 单独做，使用 cell 会更方便一些。两个 embedding 层分别作为

encoder 和 decoder 的嵌入层，用于将 one-hot 嵌入到词向量。Atten_fc 是计算 w*hs 的全连接层，decoder_fc 是 attention 机制中生成修正短时记忆向量 ht’的全连接层，Out_fc 是 encoder 输出接分类器的全连接层。

```
self.encode_embedding = nn.Embedding(in_dim, input_size)
self.decode_embedding = nn.Embedding(out_dim, input_size)
self.encoder_lstm = nn.LSTM(
    input_size = input_size,
    hidden_size = hidden_size,
    num_layers = num_layers,
    batch_first = True,
    bidirectional = True
)
self.decoder_lstm = nn.LSTMCell(
    input_size = hidden_size,
    hidden_size = hidden_size,
)
self.atten_fc = nn.Linear(hidden_size, hidden_size)
self.decoder_fc = nn.Linear(2*hidden_size, hidden_size)
self.out_fc = nn.Linear(hidden_size, out_dim)
```

（图 9）

3.2 编码器解码器

编码器根据输入词向量，返回编码后的隐状态。相关代码见图 10。

```
def Encoder(self, inputs):
    output, (h_n, c_n) = self.encoder_lstm(inputs)
    # (batch_size, timesteps, 2*hidden_size), (batch_size, 2*hidden_size)
    output = output[:, :, :self.hidden_size] + output[:, :, self.hidden_size:]
    # (batch_size, timesteps, hidden_size)
    h_n = (h_n[0] + h_n[1])
    c_n = (c_n[0] + c_n[1])
    # (batch_size, hidden_size) simple add
    return output, (h_n, c_n)
```

（图 10）

解码器根据输入词向量，隐藏状态和解码器输出，返回解码后的隐状态(包括长期记忆状态和短期记忆状态)。相关代码见图 11。

```
def Decoder(self, inputs, in_hidden, encoder_outputs):
    inputs = self.decode_embedding(inputs)
    # (batch_size, hidden_size)
    batch_size = inputs.size()[0]

    out_h, out_c = self.decoder_lstm(inputs, in_hidden)
    # (batch_size, hidden_size), (batch_size, hidden_size)
```

（图 11）

3.3 Attention 机制的实现

先使用图 12 的 attention 函数得到 attention weight 向量，后使用图 13 的代码对 decoder 的短期记忆输出 ht 进行修正得到隐状态输出。每隔 tensor 的维护都在代码中有体现，不再赘述。

```
def attention(self, ht, hs):
    whs = self.atten_fc(hs).transpose(1,2)
    # (batch_size, hidden_size, timesteps)
    ht = ht.unsqueeze(1)
    # (batch_size, 1, hidden_size)
    score = torch.bmm(ht, whs)
    # (batch_size, 1, timesteps)
    return F.softmax(score, dim=-1)
```

（图 12）

```
if self.ues_attention:
    attention = self.attention(out_h, encoder_outputs)
    # (batch_size, 1, timesteps)
    context = torch.bmm(attention, encoder_outputs).squeeze()
    if batch_size==1:
        context = context.unsqueeze(0)
    # (batch_size, hidden_size)
    context = torch.cat((context, out_h), 1)
    # (batch_size,  2*hidden_size)
    out_h = torch.tanh(self.decoder_fc(context))
    # (batch_size,  hidden_size)
return (out_h, out_c)
```

（图 13）

3.4 损失函数

实验过程中为了保持句子长度一致加了<pad>单词，在计算 loss 的时候需要将其删去。同时 decoder 的输出不包含<BOS>，所以也要在 lable 中将其删去。之后使用交叉熵作为损失函数进行计算即可。

```
class MyLoss(nn.Module):
    def __init__(self, weight):
        super().__init__()
        self.Loss_function = nn.CrossEntropyLoss(weight=weight)

    def forward(self, outputs, lables, masks):
        pos = torch.stack([x for x in masks.nonzero() if x[1]>0])
        x,y = pos.transpose(0,1)
        _,m,k = outputs.size()
        outputs = outputs.reshape([-1,k])
        index_pre = x*m+y-1
        predict = torch.index_select(outputs,0,index_pre)
        lables = lables.flatten()
        index_lab = x*(m+1)+y
        gt = torch.index_select(lables,0,index_lab)
        return self.Loss_function(predict, gt)
```

(图 14)

3.5 Beam search 的实现

在测试的部分，我实现了 beam search，可以得到得分前 K 大的状态。具体的实现过程已经在 2.6 节中体现，不再赘述。相关代码如下。

```
def test_step(self, inputs, beam_size=5):
    inputs = self.encode_embedding(inputs)
    # (1, timesteps, hidden_size)
    encoder_outputs, hidden = self.Encoder(inputs)
    # (1, timesteps, hidden_size), (1, hidden_size)

    time_steps = inputs.size()[1]*2
    cur_beam = [([0], hidden, 1.0)]
    # input, hidden, score
    best_result = None
    max_pro = 0
    for step in range(time_steps):
        candidate_beam = []
        for beam in cur_beam:
            sent, hidden, value = beam
            inputs = torch.LongTensor([sent[-1]])
            # (1)
            hidden = self.Decoder(inputs, hidden, encoder_outputs)
            # (1, hidden_size)*2
            decoder_out = F.softmax(self.out_fc(hidden[0]))
            # (1, out_dim)
            val, pos = torch.topk(decoder_out.view(-1).data, k=beam_size)
            for v,p in zip(val,pos):
                now = sent+[p.item()]
                pro = value*v
                if p.item()==1:
                    if pro > max_pro:
                        best_result,max_pro = now,pro
                else:
                    candidate_beam.append((now, hidden, pro))
        # get candidate beam
        cur_beam = []
        for sent, hidden, pro in candidate_beam:
            if pro >= max_pro:
                cur_beam.append((sent, hidden, pro))
        # get current beam
        if len(cur_beam) > beam_size:
            probs = [beam[2] for beam in cur_beam]
            pros_idx = np.array(probs).argsort()[-1*beam_size:]
            results = [cur_beam[idx] for idx in pros_idx]
            cur_beam = results
        # sort current beam
        if len(cur_beam)==0: break

    if best_result is None:
        best_result = cur_beam[0][0]
    return best_result
```

(图 15)

4 实验结果与分析

4.1 实验设置

我使用了 dataset_10000 进行对照实验。在 Linux 服务器上使用一张 RTX2080ti 进行训练(训练时间大约为 2h)。使用 Adam 优化算法作为优化器，设置 weight_decay=1e-4。预计训练 600 个 epoch，可以根据实际训练情况动态调整训练时间。动态调整学习率，设置初始学习率为 5e-3，如果连续 5 个 epoch 训练集的 loss 都不能达到最小值则学习率减半，并重新开始计数。如果学习率小于 1e-6 则停止训练。

我将训练数据按照长度排序分 batch，对于每个 batch 将每个句子 padding 到当前 batch 中的最大长度。在测试的时候，允许的翻译结果输出的最大长度为源语言 padding 长度的两倍。

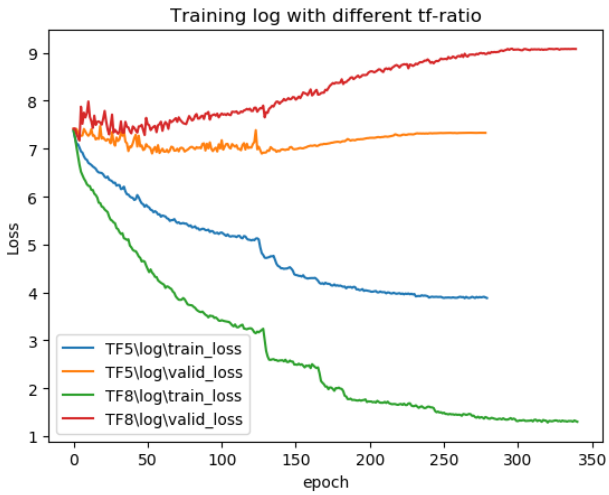
最后我又尝试在 dataset_100000 上进行实验。由于数据量较大，模型在上述环境中大约需要训练一天半才能收敛。

4.2 Teacher forcing 的影响

我尝试了 0.3,0.5，0.8 这三种 teacher force 的概率。如表 1。比较意外的是，rate 设置为 0.8 的时候无论对训练集还是测试集，效果都是最好的。我认为可能是因为数据量不够大，导致在 tf-rate 较低的时候模型不收敛。我尝试画出模型的训练图，结果也证实了这一点。如图 16，在 tf-rate 为 0.5 的时候模型的 loss 并不收敛，而在 0.8 时模型收敛得更好，翻译出来的 blue 值也就更高。

tf-rate	Train_blue	Test_blue
0.3	0.0021	0.0002
0.5	0.1200	0.0008
0.8	0.4969	0.0183

(表 1)



(图 16)

我认为在一次训练中 teacher forcing 的概率应该是动态调整的，在开始训练时较高，并随着训练的进行逐步降低。我尝试先使用 0.8 的 rate 训练一个模型，后使用 0.6 的 rate 继续训练，最后使用 0.4 的 rate 训练。如表 2 为实验结果。

不过从实验结果上看，迭代训练后模型的泛化性能反而降低了。我认为这可能是由于我采用的这种“炼老丹”的训练方式不当导致的。可以尝试设计一个 end-to-end 的方法，效果或许会好一些。

tf-rate	Train_blue	Test_blue

0.8	0.4973	0.0204
0.8+0.6	0.4920	0.0189
0.8+0.6+0.4	0.4969	0.0121

(表 2)

4.3 预训练词向量

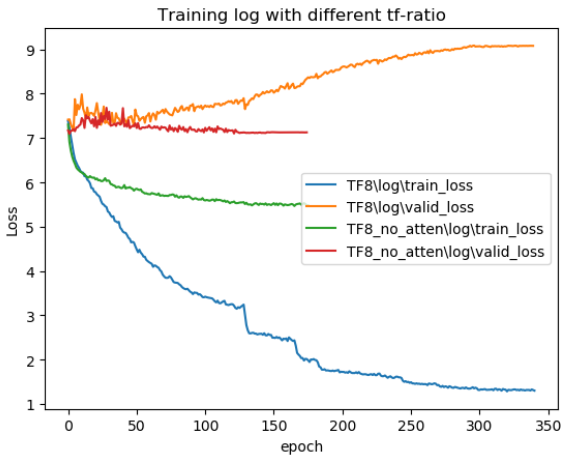
我使用 dataset_100000 中的训练集与训练出 word2vec 词向量，并将其加载进 embadding 层进行训练。不过加载预训练 w2v 模型后，无论是在训练集上还是在测试集上模型的效果都不好。我认为这可能是由于 w2v 的参数是在另一个 domain 上训练的，和机器翻译并不适配。可能需要一个更好的迁移学习的方法。

tf-rate	Use-w2v	Train_blue	Test_blue
0.5	N	0.1200	0.0008
0.5	Y	0.0036	0.0001
0.8	N	0.4969	0.0183
0.8	Y	0.4128	0.0064

(表 3)

4.4 Attention 机制

如图 17 为设置 tf-rate=0.8 时使用 attention 和不使用 attention 的对比情况。可以看到使用 attention 机制的模型表达能力更强，收敛得更好。

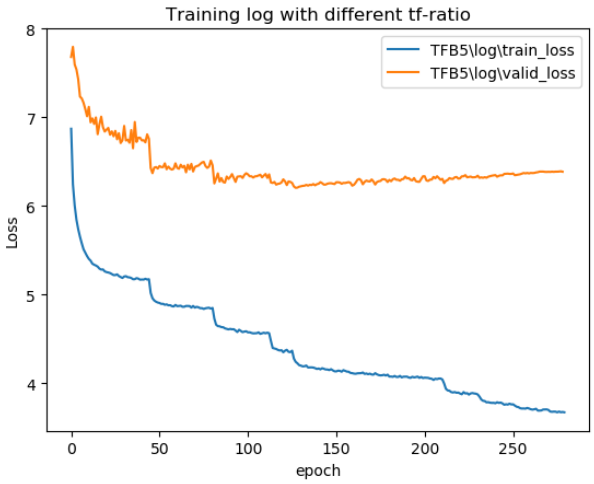


(图 17)

4.5 结果展示

在 dataset10000 上的训练 loss 如图 16 所示，在训练集上模型还是能够收敛的，在验证集上模型随着训练的进行会先收敛后发散。这主要是由于数据量不够导致的过拟合现象。

我又尝试在 dataset100000 上进行实验。设置 teacher-force rate=0.5，模型训练了 15 个小时之后，在训练集的得分仅有 0.1518 的情况下，模型在测试集上的得分可以达到 0.0575，远超实验 dataset10000 进行训练的所有模型。如图 18 为在大数据集上训练的收敛图。可以看到无论是训练集的 loss 还是验证集的 loss 都呈下降趋势，过拟合现象大大缓解。



(图 18)

由于 ddl 临近，训练时间不足，所以也就没有继续训练下去了。我相信对 teacher ratio 进行跳着后结果还可以更好。不过由此也可以说明，增大训练语料的大小可以显著提升模型的泛化能力，提高模型性能。

以下为模型在测试集上的几个高分翻译结果展示。可以看到在部分句子上翻译的效果还是很不错的。尤其是第三个句子，作为一个长句子能够基本翻译出大意还是很不容易的。

```
源语言: ['<BOS>', '国家', '不', '喜欢', '被', '孤立', '的', '感觉', '<EOS>']
翻译结果: ['<BOS>', 'countries', 'do', 'not', 'like', 'to', 'be', 'isolation', '<EOS>']
参考译文: ['<BOS>', 'countries', 'do', 'not', 'like', 'to', 'feel', 'isolated', '<EOS>']
Score: 0.6104735835807844
```

```
源语言: ['<BOS>', '这', '不是', '严肃', '的', '政策', '<EOS>']
翻译结果: ['<BOS>', 'this', 'is', 'not', 'serious', 'policy', '<EOS>']
参考译文: ['<BOS>', 'this', 'is', 'not', 'serious', 'policy', '<EOS>']
Score: 1.0
```

```
源语言: ['<BOS>', '事实上', '这些', '因素', '有助于', '解释', '为何', '法国', '仍', '是', '世界
', '第五', '大', '经济体', '<EOS>']
翻译结果: ['<BOS>', 'in', 'fact', 'these', 'factors', 'help', 'to', 'explain', 'why',
'france', 'remains', 'the', 'fifth', 'fifth', 'largest', 'economy', '<EOS>']
参考译文: ['<BOS>', 'indeed', 'these', 'factors', 'help', 'to', 'explain', 'why', 'france',
'remains', 'the', 'world', 's', 'fifth', 'largest', 'economy', '<EOS>']
```

5 总结

本次大作业我实现了一个基于 encoder-decoder 结构和 attention 机制的机器翻译模型。通过本次实验，我对 attention 机制有了更深入地了解，也对 seq-to-seq 类模型的 framework 和性能有了更深入的体会。不过比较遗憾的是，由于数据量不足和训练时间不足的原因模型最终的准确率并不理想，blue 值最高也只能达到 0.05。

参考文献

[1] Effective Approaches to Attention-based Neural Machine Translation, Luong et al., EMNLP 2015
[2] Neural Machine Translation by Jointly Learning to Align and Translate, Bahdanauet al., ICLR 2015
[3] Bleu: a Method for Automatic Evaluation for Machine Translation, Papineniet al., ACL 2002