School of Informatics

Natural Language Understanding, Generation, and Machine Translation(2020–21) Coursework 2

Neural Machine Translation

March 2021

Question 1: Understanding the Baseline Model

Comment A

When self.bidirectional is set to True, the state_size is [2 * self.num_layers, batch_size, self.hidden_size]. The even and odd number indexes in the first dimension of final_hidden_states and final_cell_states corresponding to the forward and backward LSTM output respectively. Therefore, the forward and backward output are concatenated in the third dimension. The final shape of the outputs would be: [self.num_layers, batch_size, self.hidden_size*2].
From pytorch documents:

$$h_t = o_t * tanh(c_t) \tag{1}$$

$$c_t = f_t * c_t - 1 + i_t * g_t \tag{2}$$

The final_hidden_states is the tensor containing the last hidden state, which is calculated from final_cell_states and out gate according to equation 1. And the final_cell_states is the tensor containing the last cell state which is calculated form forget gate, previous cell state and input according to equation 2.

Comment B

The attention context vector is calculated from summing the product of attention weights and encoder output at each time step. The attention weights are calculated by using softmax function on the time step dimension of masked attention scores.

As the sentence lengths of different batches are different, shorter sentences are padded to ensure the same length. When attention weights are calculated, to avoid weights on paddings, the attention scores of paddings are replaced with negative infinity. As a result, after softmax function, the paddings will have zero weights.

Comment C

The attention scores are calculated form the dot product of current state (tgt_input) and the source states (encoder_out). The source states are linear transformed to match the input_dims.And get the result of size [batch_size,1,src_time_steps], the third dimension contains the attention score of different t.

The matrix multiplication dot product each source states with current state and get a score to indicate the alignment relationship between encoder and decoder. According to Luong's global attention model, the alignment vector is obtained by using softmax function on the attention score vector.

Comment D

Decoder state is initialized with tgt_hidden_states, tgt_cell_states, input_feed to be zero tensors with size [batch_size, hidden_size].

The cached_state is None when incremental_state is None or the current instance does not have corresponding previous cached state in incremental_state.

The input_feed is the output of previous time step with size of [batch_size, hidden_size]. It would concatenate with current token embedding act as input of current LSTM.

Comment E

When the top layer hidden state h_t of decoder is computed, the encoder uses this hidden state h_t and encoder hidden states as input of attention layer to compute attention weighted output and attention score.

The attention layer takes the current target hidden state of decoder and encoder hidden states as input to compute the attention scores by measuring the similarity. The previous target state contains the information of current target hidden state of decoder.

The dropout layer sets values of input_feed with certain probability. It can prevent overfitting theoretically. In this case, it prevents the decoder only depend on the previous decoding state.

Comment F

The code gets the output form training set. Then it uses the output to calculate cross entropy loss. Next, from the cross entropy loss, it calculates the loss gradient for each parameters by back-propagation. And the loss gradient is normalized to prevent vanishing gradient or gradient explosion. Then the parameters are updated according to the gradients. Finally, the gradients are cleared to be 0.

Question 2: Understanding the Data

2.1 Total counts of tokens & word types

- English word tokens: 124031
- German word tokens: 112572
- English Word types: 8326
- German word types: 12504

2.2 $< UNK >$ and vocabulary size

- Number of English words that appear once: 3909 (which should be replaced by $< UNK >$)
- Subsequently, English vocabulary size: 4418
- Number of German words that appear once: 7460 (should be replaced by $< UNK >$)
- Subsequently, German vocabulary size: 5045

Analysis: words that rarely appear in corpus could be replaced by $< UNK >$ symbol to effectively reduce the size of the vocabulary. And the POS tag for the rare words is usually noun (proper and common noun). For instance, the name 'rossa' of the person 'Mr De Rossa' only appear once in both train.en and train.de. The reduction of vocabulary would also affect the test sets, as there will be more out-of-vocabulary (OOV) words, and the words that appear multiple times in test set might simply be considered as $< UNK >$ symbol, which could lead to the increase of the degree of ambiguity and the decrease of the test score.

2.3 Same tokens

- same words in both languages: 1460 (regardless the meanings)

Such word similarity might help us to find alignments between both languages.

Question 3: Improved Decoding

3.1 Problem of Greedy Search

Greedy search picks the token with maximum probability at each time step according to the distribution $P(E|F)$ and generates the following output based on the previous optimal. This is a simple way of generating 1-best output, however, greedy search is not guaranteed to find the optimal translation with the highest probability of the whole sentence[1]. Consider the following figure as a counterexample. The probability of optimal translation $'ab\langle\backslash s\rangle'$ is $0.35\times0.8\times1 = 0.28$, whereas the probability of the greedy search output $'bb\langle\backslash s\rangle'$ is $0.4 \times 0.5 \times 1 = 0.2$, which is lower.

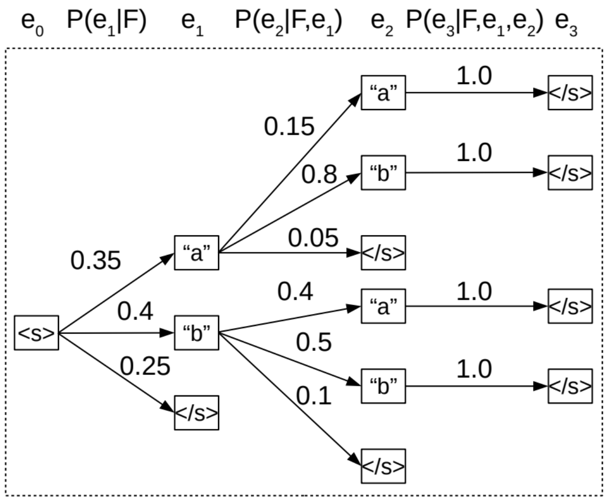

Figure 1: Failure of Greedy Search[1]

3.2 Beam search

To achieve the beam search in the decoder, the input and output of the decoder should add another dimension as the dimension 0 to represent the beam width. And therefore, the tgt_inputs

would present the input words in the current beam. The probability distribution of all beams is the output of the decoder.
There is a trick that can be applied here. For calculating the probability of the entire hypothesis before a particular time step, we can use the summation of log probabilities to avoid the multiplication of probabilities, which is more efficient in computing and more accurate in result.

3.3 Short sentences & Length Norm

The decoder favour short sentences because the probability of a sentence in beam search is done by multiplying the probability of a word in the condition of the predecessors:

$$P(E|F) = \prod_{t}^{|E|} P(e_t|F, e_1^{t-1}) \tag{3}$$

As $P(e_t|F, e_1^{t-1}) < 1$. Therefore, the long sentences would have low probabilities after multiplying probabilities smaller than 1.
Length normalisation can solve the short preference when beam size is small. According to [2], when beam size increase. more sentences are explored, it becomes easier for the search algorithm to find the ending symbols $\langle eos \rangle$. The algorithm again prefers short sentence in this situation.

Question 4: Adding Layers

4.1 Command

The command used is as follows:

```
--encoder-num-layers 2\
--decoder-num-layers 3\
```

4.2 Results

	Dev_Perplexity↓	**Test_BLEU↑**	**Train_Loss↓**
Baseline	26.4	11.03	2.145
More-Layer	**29.4**	**9.38**	**2.353**

(↑ means the larger the better and ↓ means the smaller the better)
The dev-set perplexity increased from 26.4 to 29.4.
The test BLEU score decrease from 11.03 to 9.38.
The training loss increase from 2.145 to 2.353.

Explanation: The model with more layers in Encoder and Decoder performed worse than the baseline model according to the results above, which reveals the degradation of the deeper architecture. Though the deeper network should be able to learn more information, it demand more training data to achieve its advantage. In theory, for each parameter added, the network requires more training data. However, the training data here only contains 10000 sentences, which is relatively small in size for the NMT task.

Question 5: Implementing the Lexical Model

5.1 Results

	Dev_Perplexity↓	**Test_BLEU↑**	**Train_Loss↓**
Baseline	26.4	11.03	2.145
Lexical model	**22.4**	**13.82**	**1.888**

The dev-set perplexity decrease from 26.4 to 22.4.
The test BLEU score increase from 11.03 to 13.82.
The training loss decrease from 2.145 to 1.888.
Therefore, using lexical translation is beneficial in this translation work. We find the translation is more accurate in lexical level.

5.2 Examination of the output translations

For example, the sentence in test.en:
'mr lannoye , i understand that you are moving that the vote be adjourned .'
The lexical model translation:
'mr president
, i would like to understand you when you will postpone the postpone of the vote .'
The baseline model translation:
'mr president
, if i said you , i think you will be the best of the vote .'
The words 'understand' and 'adjourned' is translated in the lexical model but not in the baseline model.

Another Example goes like this: In the test.en:
'i urge you to support this resolution .'
The lexical model translation:
'i would call you to support this resolution .'
The baseline model
'i would like to thank you to support this resolution .'
The word 'call' is translated in the lexical model but not in the baseline model. Instead, the baseline model translate it as 'thank'.

Question 6: Understanding the Transformer Model

Comment A

The positional embeddings encode the position information of words. According to the positional embeddings, encoder and decoder can tell order of words.

We can not only use the embedding similar to LSTM, because the LSTM can tell word ordering from the RNN structure. But transformer use self-attention, which can not get word ordering information from its structure. Therefore, the positional embeddings are needed.

Comment B

The self_attn_mask is used to mask the forward words in current step.

The encoder need to encode the word context forward and backward direction. Then forward words should not be masked in encoder.

In incremental decoding, the encoder and decoder can only access the backward information. Therefore, incremental decoding do not need to mask.

Comment C

The decoder output need be projected in to size [batch_size, src_time_steps, len(dictionary)]. Using softmax, the output word can be determined by the word with largest probability.

The output of decoder layer is the forward_state with shape [batch_size, src_time_steps, num_features].

If the features_only=True the decoder directly return the features_state without projection, the output would be the feature encoding of target word.

Comment D

The purpose of encoder_padding_mask is similar to the mask in question 1b, which make the encoder ignore the meaningless paddings.

According to the self_attn function, the output has size of [tgt_time_steps, batch_size, embed_dim]

Comment E

Self attention is used to generate the self attentions between target states. And the encoder attention is used to generate attentions between target states and encoders outputs.

The attn_mask is used to prevent the decoder self attention pay attention to subsequent words.

The key_padding_mask is used to avoid pay attention to paddings.

The encoder do not need attn_mask as it need encode forward and backward context information.

Question 7: Implementing Multi-Head Attention

	Dev_Perplexity↓	**Test_BLEU↑**	**Train_Loss↓**
Baseline	26.4	11.03	2.145
Multi-Head model	**124**	**0.8**	**2.604**

Compared with the model trained before, the difference in training loss in not as large as the difference in dev_perplexity and test_BLEU. The performance in dev_perplexity and test_BLEU declines significantly.

Considering the loss in training set and development set, the model with transformer might has overfitted the training set within 15 epochs. There are 10000 sentences in the training set, which is not enough to this model.

This model implement the transformer model in [3]. This model is much more complex than the LSTM and LSTM with lexicla implemented before. Such complex architecture is easy to overfit the training set.
Therefore, to improve the performance, the model can be trained on a larger model.
It is noticed that when the attention drop out is applied, the performance on test and validation set is increased. After training with dropout rate of 0.5, the model converge slower and with smaller validation perplexity.

```python
    def forward(self,
                query,
                key,
                value,
                key_padding_mask=None,
                attn_mask=None,
                need_weights=True):

        # Get size features
        tgt_time_steps, batch_size, embed_dim = query.size()
        assert self.embed_dim == embed_dim

        '''
        ___QUESTION-7-MULTIHEAD-ATTENTION-START
        Implement Multi-Head attention  according to Section 3.2.2
            of https://arxiv.org/pdf/1706.03762.pdf.
        Note that you will have to handle edge cases for best model
            performance. Consider what behaviour should
        be expected if attn_mask or key_padding_mask are given?
        '''

        # attn is the output of MultiHead(Q,K,V) in Vaswani et al.
            2017
        # attn must be size [tgt_time_steps, batch_size, embed_dim]
        # attn_weights is the combined output of h parallel heads of
             Attention(Q,K,V) in Vaswani et al. 2017
        # attn_weights must be size [num_heads, batch_size,
            tgt_time_steps, key.size(0)]
        # project KQV into KW_ik QWiq VWiv
        K_kd = self.k_proj(key).contiguous().view(-1, batch_size,
            self.num_heads, self.head_embed_size)
        Q_kd = self.q_proj(query).contiguous().view(tgt_time_steps,
            batch_size, self.num_heads, self.head_embed_size)
        V_kd = self.v_proj(value).contiguous().view(-1, batch_size, self.num_heads, self.head_embed_size)
        # in order to use bmm function, merge the self.num_heads dimension and batch_size dimension
        K_kd = K_kd.transpose(0,2).contiguous().view(self.num_heads*batch_size,-1,self.head_embed_size)
        Q_kd = Q_kd.transpose(0,2).contiguous().view(self.num_heads*batch_size,tgt_time_steps,self.head_embed_size)
        V_kd = V_kd.transpose(0,2).contiguous().view(self.num_heads*batch_size,-1,self.head_embed_size)

        # attn_weight = [self.num_heads*batch_size, tgt_time_steps, source_time steps]
        # multiply the multihead Q and multihead K can devided by scaling
        attn_weights = torch.bmm(Q_kd,K_kd.transpose(1,2))/self.head_scaling

        #  Apply masking into att_weights
        if key_padding_mask is not None:
            #  attn_weight size:  [self.num_heads * batch_size, tgt_time_steps, source_time steps] ->
            #                     [batch_size, self.num_heads, tgt_time_steps, source_time steps]
            attn_weights = attn_weights.contiguous().view(batch_size, self.num_heads,tgt_time_steps,-1)
            #  key_padding_mask size   [batch_size,source_time steps] ->
            #                          [ batch_size, 1, 1, source_time steps]
            #  modify the shape of attn_weight and key_padding_mask to apply mask
            key_padding_mask = key_padding_mask.unsqueeze(dim=1).unsqueeze(dim=2)
            attn_weights.masked_fill_(key_padding_mask, float('-inf'))
            # transfer back the shape of attn_weight
            attn_weights = attn_weights.contiguous().view(self.num_heads*batch_size, -1, tgt_time_steps)
        if attn_mask is not None:
            #  attn_mask size: [tgt_time_steps, source_time steps]
            #  attn_weight size: [self.num_heads * batch_size, tgt_time_steps, source_time steps]

            #  attn_mask size:    [tgt_time_steps, source_time steps] ->
            #                     [1,tgt_time_steps, source_time steps]
            # modify the shape of attn_weight and attn_mask to apply mask
            attn_mask = attn_mask.unsqueeze(dim=0)
            if attn_mask.dtype == torch.bool:
                attn_weights.masked_fill_(attn_mask, float("-inf"))
            else:
                attn_weights += attn_mask

        attn_weights = F.softmax(attn_weights, dim=-1)
        # attn_weights = F.dropout(attn_weights, p=self.dropout, training=self.training)
        # bmm attn_weights  [self.num_heads * batch_size, tgt_time_steps, source_time steps]
        #     V_kd          [self.num_heads*batch_size, source_time steps, self.head_embed_size]
        #        attn [self.num_heads*batch_size, tgt_time_steps, self.head_embed_size]
        #      Generate attention output with multiply attention weights with value vectors.
        attn = torch.bmm(attn_weights, V_kd)
        #   attn ->[tgt_time_steps, batch_size, embed_dim]
        attn = attn.transpose(0, 1).contiguous().view(tgt_time_steps, batch_size, self.num_heads, self.head_embed_size).view(tgt_time_steps, batch_size, embed_dim)
        attn = self.out_proj(attn)


        attn_weights = attn_weights.contiguous().view(batch_size, self.num_heads, tgt_time_steps, -1)

        '''
        ___QUESTION-7-MULTIHEAD-ATTENTION-END
        '''

        return attn, attn_weights
```

References

[1] Graham Neubig. Neural machine translation and sequence-to-sequence models: A tutorial. *CoRR*, abs/1703.01619, 2017.

[2] Yilin Yang, Liang Huang, and Mingbo Ma. Breaking the beam search curse: A study of (re-) scoring methods and stopping criteria for neural machine translation. *arXiv preprint arXiv:1808.09582*, 2018.

[3] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.