NLU-CW2:

s2025012,s2045321

1 Understanding the Baseline Model

(a) Add explanatory comments to the code

(a).1 Comment A

When self.bidirectional is set to True , it means that we use a bidirectional LSTM as our encoder, we use both the forward encoder (feed sequence from left to right) and reverse encoder (feed sequence from right to left). Therefore, the output of encoder contains two hidden states with size (num_layers * 1, batch, hidde_size) calculated by forward encoder and reverse encoder respectively. These two hidden states are contained in the first dimension of variable final_hidden_states (num_layers * 2, batch, hidden_size) with index 0 and 1. Here, we extract these two hidden states and concatenate these two hidden states together. The same explanation and operation for cell states.
The difference between final_hidden_states and final_cell_states is that final_cell_states is like a conveyor belt to transmit information with only a small amount of linear interaction in each LSTM module, so it can store information of the whole input sequence undistortedly even it has a long distance. On the contrary, The final_hidden_states are overwritten at each step to control the gates.

(a).2 Comment B

Here, we firstly feed the attention scores given by self.score function and feed the scores to softmax function which can give us the weights(probability)**W** for the hidden states of the LSTM encoder at each time step($h_1^{f'}, h_2^{f'}, ..., h_{src\_time\_steps}^{f'}$). For each encoder hidden state, the higher the probability, the more attention we should pay to this hidden state in context. After that, for each batch we calculate the context vector **c** by matrix multiplication between weights and hidden states and we use torch.bmm to deal with this batches matrix multiplication (the detailed explanation about torch.bmm you can see in comment C). Then in order to calculate the output scores of current lstm decoder, we concatenate decoder hidden states h_t^e at current time step and current context vector together and feed them into the linear transformer layer followed by a tanh activation function:

$$S_t^e = tanh(linear\_transformer([h_t^e; c_t])) \tag{1}$$

The reason why apply the mask to the attention score is because in order to ensure the length of the input sequences to LSTM encoder are consistent, we pad 0 at the end of the sequence which length is shorter than the max sequence length. When we apply the attention technique to decoder, we don't want our decoder to look up these padding position as context because they are meaningless. Therefore, we mask theses padding positions with negative infinity which will lead to almost zero probability by SoftMax function.

(a).3 Comment C

For the score function , the input variable tgt_input h_t^e is the hidden state of decoder at a specific time t and it has the size [batch_size, decoder_input_dims]. The input variable encoder_out (H^f) is the hidden states of the LSTM encoder at each time step and it has the size [batch_size, src_time_steps, encoder_output_dims]. The value of decoder input dims is equal to encoder output dims.

Here, we actually use the bilinear function to calculate the attention score. Therefore, we firstly do the linear transformation to H^f and get a $H^{f'}$. Then, we do some data processing which is helpful to the following dot product operation. For example, we transpose the $H^{f'}$ between dimension 1 and dimension 2, and we insert a dimension with size one to the variable tgt_input h_t^e ([batch_size,1,input_dims]). In order to help explanation, we just firstly discuss what happened in each batch. Here, we do the dot product between the row vector h_t^e with each column $(h_1^{f'}, h_2^{f'}, ..., h_{src\_time\_steps}^{f'})$ of $H^{f'}$ which lead to a matrix multiplication form $h_t^e \times H^{f'^T}$. One thing we should not forget is that the above discussion is all in one batch, meaning that we just assume both of these variables are two-dimensional, but actually both the h_t^e and $H^{f'}$ are three dimension tensors and the first dimension represents for the number of batch we have. Since we want to have the same dot product operation for each batch, we use torch.bmm to deal with this batches matrix multiplication.

(a).4 Comment D

The incremental state is used when we use our seq2seq model to translate the sequence. When the seq2seq model is during training, the incremental state is none. At this time, we initialize three variables tgt_hidden_states, tgt_cell_states, input_feed with size [batch size , hidden_size] and zero values. The input feed serve as the part of lstm_input which will be fed into the next LSTM module in decoder. If we don't the attention technique, the input_feed is just the hidden states h_t^e of current time step t. If we use the attention technique, the input_feed is the output of attention layer(S_t^e in equation (1)).

(a).5 Comment E

Attention is integrated into the decoder through the input_feed variable which will be fed into the next LSTM module in decoder. When we use the attention technique, the input_feed variable is the output of attention layer(S_t^e in equation (1)). One thing worth mentioning is that in fact in the code we feed into the attention layer not the hidden state at the last time but the hidden state calculated at the current time. We feed the hidden state because we need this variable when we calculate the weight of attention. The drop_out layer can randomly set some elements of input_feed zero which can naturally prevent overfitting to some extent. What's more, The introduction of this randomness enhances Decoder's robustness and enables Decoder to learn to use the most valuable information.

(a).6 Comment F

Here, we firstly feed the training samples to the encoder-decoder model and get the output. Then, we use the cross entropy function to calculate loss. After that, we use the automated backpropagation functioned by pytorch framework. In order to preventing very large gradient, we use gradient clip function. Finally, we update the parameters and set gradient to zero so that we are ready to start next epoch.

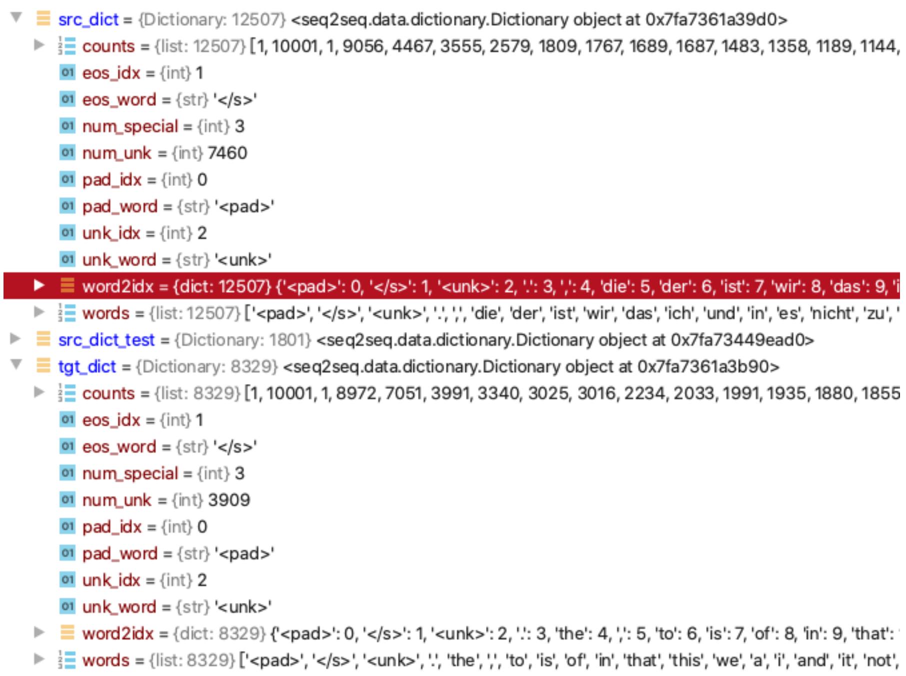
2 Understanding the Data

(a)

From Figure 1, we can see that the word type number of English data is 8329-3= 8326, The word type number of German data is 12507-3=12504. We minus 3 here because when we use preprocess.py to generate src_dict and tgt_dict, we add the three tokens $< pad >, < unk >, < /s >$ to src_dict and tgt_dict which actually didn't appear in the train.en and train.de files.
Again, here we don't take the number of $< pad >, < unk >, < /s >$ into account. The total token number of English data is 134034-1-1-10001= 124031, of German data is 122575-1-1-10001=112572.

(b)

From Figure 1, we can see that Number of unk token in English data is 3909 and number of unk token]in German data is 7460. After we use the unk token to replace the words which only appear once, the vocabulary size for English is 8326-3909+1= 4418 and vocabulary size for German is 12504-7460+1 = 5045



```
▼ src_dict = {Dictionary: 12507} <seq2seq.data.dictionary.Dictionary object at 0x7fa7361a39d0>
    ▶ counts = {list: 12507} [1, 10001, 1, 9056, 4467, 3555, 2579, 1809, 1767, 1689, 1687, 1483, 1358, 1189, 1144,
      eos_idx = {int} 1
      eos_word = {str} '</s>'
      num_special = {int} 3
      num_unk = {int} 7460
      pad_idx = {int} 0
      pad_word = {str} '<pad>'
      unk_idx = {int} 2
      unk_word = {str} '<unk>'
    ▶ word2idx = {dict: 12507} {'<pad>': 0, '</s>': 1, '<unk>': 2, '.': 3, ',': 4, 'die': 5, 'der': 6, 'ist': 7, 'wir': 8, 'das': 9, 'i
    ▶ words = {list: 12507} ['<pad>', '</s>', '<unk>', '.', ',', 'die', 'der', 'ist', 'wir', 'das', 'ich', 'und', 'in', 'es', 'nicht', 'zu', '
▶ src_dict_test = {Dictionary: 1801} <seq2seq.data.dictionary.Dictionary object at 0x7fa73449ead0>
▼ tgt_dict = {Dictionary: 8329} <seq2seq.data.dictionary.Dictionary object at 0x7fa7361a3b90>
    ▶ counts = {list: 8329} [1, 10001, 1, 8972, 7051, 3991, 3340, 3025, 3016, 2234, 2033, 1991, 1935, 1880, 1855
      eos_idx = {int} 1
      eos_word = {str} '</s>'
      num_special = {int} 3
      num_unk = {int} 3909
      pad_idx = {int} 0
      pad_word = {str} '<pad>'
      unk_idx = {int} 2
      unk_word = {str} '<unk>'
    ▶ word2idx = {dict: 8329} {'<pad>': 0, '</s>': 1, '<unk>': 2, '.': 3, 'the': 4, ',': 5, 'to': 6, 'is': 7, 'of': 8, 'in': 9, 'that':
    ▶ words = {list: 8329} ['<pad>', '</s>', '<unk>', '.', 'the', ',', 'to', 'is', 'of', 'in', 'that', 'this', 'we', 'a', 'i', 'and', 'it', 'not',
```

Figure 1: Transformer Architecture

(c)

In English, verbs express the past and present continuous tenses by inflecting their morphology, such as adding the suffix such as ed or ing. When we check the dictionary, we find that a large number of these inflected verbs occur only once, meaning that they are commonly covered by UNK token. In German, verb morphology inflections are more complex, and they are influenced not only by tense but also by person. There are also many variations, not only to add multiple affixes , but even need to modify the word root. Even though we don't know German words, we can infer that these inflected words are most likely replaced by UNK token. Because we use UNK to replace these inflected words which stored tense information, when we used the NMT model trained by these data sets to deal with the test set, it was very likely that the tense of sentence could not be correctly translated.

(d)

We totally have 1463-3=1460 (we don't consider $<pad>, <unk>, </s>$ again) vocabulary tokens have the same morphology between both languages. Some types tokens which share the same morphology also share the same meaning. For example, the Named Entity such as "Kosovo","japan","markov". Secondly, the numbers such as year 1983,2004 and some normal words are used in both English and German such as "peace". These shared meaning tokens which can be very useful in NMT when we apply lexical attention.

(e)

Sentence length: Without attention, the longer the sentence, the worse the NMT translation result, because the corresponding context is too far away. With attention, the length of sentence hardly affects the result of NMT translation, and the distance between the target word and the corresponding source word is always 1.
Token ratio = len(valid_tokens) / sum([frequencies[v] for v in valid_tokens]) The smaller token ratio is, the more times each word appears in the training set under the same vocabulary size. During the training, the number of unusual words was reduced, which was beneficial to the NMT system.
Unknow word handling: Unknow word handling is important because we always deal with open-vocabulary translation in the real world. It is usual for our NMT system to counter with the Unknow word. The stronger the ability to handle unknown word handling, the better the translation.

3 Question 3:Improved Decoding

(a)

Greedy search is to select the word with the highest probability in the current step in the decoding process of each step until the end of decoding. However, this method can not ensure that the model can find the result with the highest overall probability. As shown in Figure 2, according to the method of greedy search, the final result will be the string "bb" corresponding to the blue line, and the total probability of this result is 0.4 * 0.5 * 1 = 0.2. However, the optimal solution of this model is the output string corresponding to the red line "aa", and the total probability of this result is 0.35 * 0.8 * 1 = 0.28 > 0.2. So we can see that greedy search may not be able to find the optimal solution.

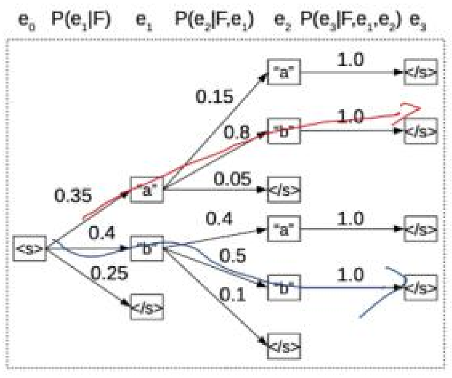


Figure 2: A search graph where greedy search fail

(b)

Assuming we use the beam_width = b. In the first time step, we expand the hypotheses for e_1 on the base of e_0. We use decoder to calculate the word probability of whole target vocabulary $P(e_1|e_0, H^f)$ where H^f is the encoder hidden states at each time step and e_0 is the value of the initial state. In the given code, e_0 is the token "eos" which means the end of the last sentence. Here, try to find the e_1 with b highest probability $P(e_1|e_0, H^f)$. After that, in order to suppress UNK token, If the token with highest probability e_1^1 is equal to UNK token, then we will save the tokens from second highest probability e_1^2 to b+1 highest probability e_1^{b+1}. Else If the token with highest probability is not equal to UNK token, then we save the tokens from highest probability e_1^1 to b highest probability e_1^b. These saved tokens will act as the starts of the paths. In the second time step, we will expand the path for e_2 on the basis of each start point (such as $e_1^1, e_1^2, ..., e_1^b$. For example, for e_1^1 we will try to find word probability of whole target vocabulary $P(e_2|e_0e_1^1, H^f)$. So, we will temporarily create $b * |V|$ possible paths. After that, we sort these possible paths by possibility from high to low. We only keep the b number of paths with the highest probability. Here we also suppress UNK tokens. At the next time step, we will do the operations as same as what we do at the second time step, we expand, sort, prune until we meet the end of the sentence.

(c)

This is because every time we add another word, we multiply in another probability, reducing the probability of the whole sentence. After using length normalization, the length of sentence will not affect the final translation choice. The only factor which can influence the model is the probability. However, this will lead to the fact that model try to obtain higher translation probability regardless of the infrequent words especially after using beam search (the range of search is larger). This will lead to the under-translation problem.

4 Adding Layers

we use the following command line:

```
python train.py –encoder-num-layers=2 –decoder-num-layers=3
```

The results are shown in Table 1. we call the seq2seq model with more layers "E_2_D_3". Compared with the baseline model, we find that increasing the number of layers increases the training loss, increases the perplexity of the validation set, and decreases the blue score of the test set. these changes mean that "E_2_D_3" performs worse than the baseline model.
From Figure 3 and Table 1, we can find both baseline and "E_2_D_3" model have a much better performance on training set than the validation set and test set. It can also be found that the loss of validation set has leveled off and stopped declining at a very early time, while the loss of training set has been continuously decreasing. We believe that overfitting occurred while training the two models.
Usually, when overfitting occurs, the common practice is to increase the size of the training set or reduce the complexity of the network so as to alleviate the situation of overfitting. However, we add layers of LSTM to the baseline model, which means we make the network architecture more complex and worsen the overfitting problem. We believe that is why "E_2_D_3" performs worse than the baseline model.

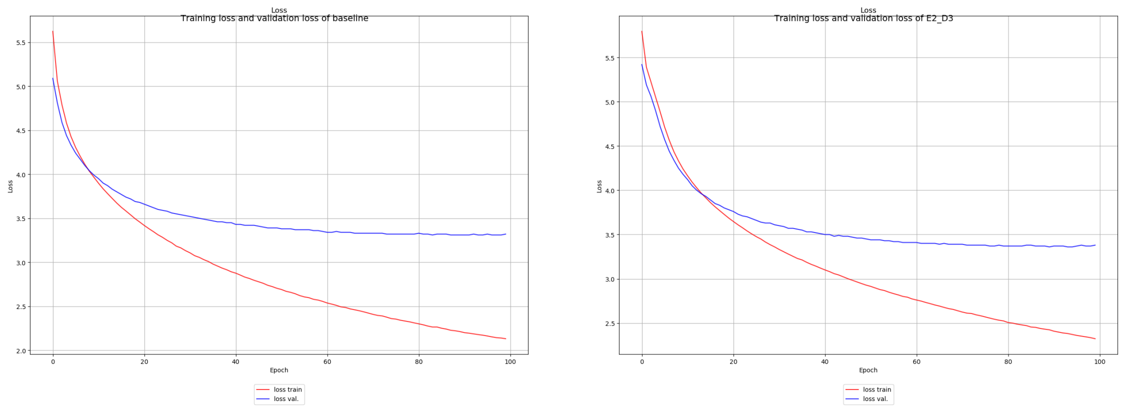


Figure 3: The performance of baseline model(left) and "E_2_D_3" (right) on the training set(red line) and validation set(blue line)

5 Question 5: Implementing the Lexical Model

Compare with the baseline model, the Lexical Model has a lower training set,validation set and test set loss and a lower validation set and test set perplexity and a higher bleu score. This shows that the model can be better performed by introducing the lexical translation.
What lexical translation do is that it introduce an attention mechanism between lexical embeddings. The reason why adding lexical translation can be beneficial to NMT system has been described in the paper[1]. The writer explained that the normal attention mechanism provide a direct context to model. However, This could make the model try to generate a target word that fits the context but doesn't necessarily correspond to the source word(s). Adding the lexical translation can setup a more direct connection between the source words and target words which can help model to memorize the rare words. Here,we present some sentence pairs which can help the explanation:

1. Source: japan steht derzeit seiner größten katastrophe seit dem zweiten weltkrieg gegenüber .
   Target: japan is facing its greatest post - war disaster .
   Baseline: the same thing is the second issue of the whole agreement.
   Lexical model: japan,the ecb of conduct has been carried since the second report.

2. Source: europa 2020 (eingereichte entschließungsanträge):siehe protokoll
   target: europe 2020 (motions for resolutions tabled) : see minutes

MODEL	TRI_LOSS	VAL_LOSS	TST_LOSS	VAL_PER	TST_PER	TST_BLEU
BASELINE	2.131	3.32	3.33	27.5	28.1	11.20
E_2_D_3	2.323	3.38	3.45	29.4	31.6	9.79

Table 1: Compare the baseline model and E_2_D_3 model

MODEL	TRI_LOSS	VAL_LOSS	TST_LOSS	VAL_PER	TST_PER	TST_BLEU
BASELINE	2.131	3.32	3.33	27.5	28.1	11.20
LEXICAL_MODEL	1.811	3.19	3.27	24.3	26.4	13.68

Table 2: Compare the baseline model and lexical model

Baseline: europe (rule) : see minutes
Lexical model:europe 2020 (motions) : see minutes

These two pair sentences you can find in In line 498 and line 242 in test file. We can clearly see that japan and 2020 are two rare words which share the morphology and meaning cross two languages. The lexical model can memorize these two rare words in source sentence and output them in target sentence. while, the baseline model "forget" them which lead to an under-translation problem.

6 Question 6: Understanding the Transformer Model

(a) Comment A

LSTM is a recurrent architecture, which can perceive the order of each word in the sequence by inputting the sequence in turn. However, due to the mechanism of self attention and encoder-decoder attention in transformer, the transformer is actually non recurrent architecture. This means that transformer view input as a set instead of sequence. In other words, for an input sequence, no matter how we permute the words in input sequence, the output sequence is always the same. In order to solve the problem that the transformer can't capture the sequence order, we apply the positional embedding. By adding the position information of the word to the word embedding, the transformer can know the position of the word, so that it can capture the sequence order.

(b) Comment B

First of all, here we train our transformer to do the machine translation task, therefore we feed the transformer with parallel sequences of two languages. In the encoder, we feed the source language sequence and do the self attention operation for source sequence. we don't use mask in encoder because we want each word to attend self attention to every other word in the source sequence. When it comes to the decoder, we need a self-attention mask for the target sequence. This is because that although we have the whole target sequence during the training, we don't have the whole target sequence when we do the translation in the real scene. For example, when we want to decode the i-th word, what we have is the previous i-1 output words before the i-th word which means that we can only attend the self-attention to all the already predicted words. Therefore, during training, we need the mask to cover the subsequent target words of i-th word. The incremental decoding is used at the inference time(translation scene). As we discussed above, we don't need a mask to prevent decoder to access the subsequent target words of i-th word which we haven't predicted.

(c) Comment C

The forward_state is the hidden states of decoder. Since we want to predict the next word, we need to compute the probabilities of each word in vocabulary. Therefore, we use the linear preojection layer to transform the hidden states from decoder to the probability distribution. the dimensionality of forward_state after this line is batch_size*tgt_time_steps*length_of_dictionary and the value of length_of_dictionary is also the vocabulary size in target language.
if feature_only == true, the output will be the hidden states from the decoder output instead of the probability.

(d) Comment D

In order to have same length for source sequence, we pad the zero at the end of the source sequence which length is shorter than max source length. When we do the self-attention operation, we don't want our encoder to look up these padding place while calculating the weighted sum because they are meaningless. Therefore, we use the encoder padding mask to cover these padding place.

The output shape of ‘state’ Tensor be after multi-head attention is [tgt_time_steps, batch_size, embed_dim].

(e) Comment E

Encoder attention is the attention operation between encoder and decoder: self attention is the attention operation between the source sequence itself or target sequence itself. We can see the self attention and encoder attention in Figure 4. key_padding_mask is the mask which is used to prevent accessing the

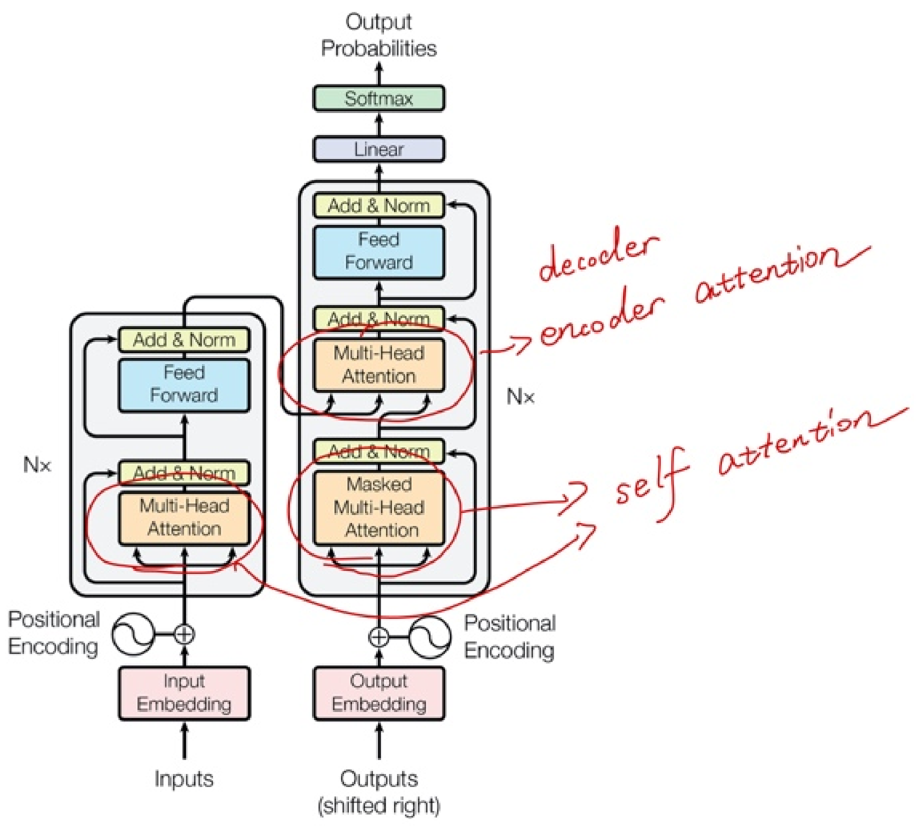


Figure 4: Transformer Architecture

padding position when doing the self attention . attn_mask is the mask which is used to prevent decoder to attending the self attention to the subsequent target sequence which haven’t been predicted. We don’t use the attn_mask here, because we will do the encoder-decoder attention which investigate how much we should pay attention to the outputs of encoder for each of the outputs of self-attention in the decoder.

7 Question 7: Implementing Multi-Head Attention

(a) Code part

The code is shown below.

```
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()
key_size = key.size()  # has same size tgt_time_steps*batch_size*embed_dim
value_size = value.size()
assert self.embed_dim == embed_dim

k = self.k_proj(key).contiguous().view(-1, batch_size, self.num_heads, self.head_embed_size)
q = self.q_proj(query).contiguous().view(tgt_time_steps, batch_size,
self.num_heads, self.head_embed_size)
v = self.v_proj(value).contiguous().view(-1, batch_size, self.num_heads, self.head_embed_size)
# linear projection and divide them into num_heads chunks
k = k.transpose(0, 2).contiguous().view(self.num_heads * batch_size, -1, self.head_embed_size)
q = q.transpose(0, 2).contiguous().view(self.num_heads * batch_size,
tgt_time_steps, self.head_embed_size)
v = v.transpose(0, 2).contiguous().view(self.num_heads * batch_size, -1, self.head_embed_size)

attn_weights = torch.bmm(q, k.transpose(1, 2)) / self.head_scaling  # scaling the dimension

if key_padding_mask is not None:
    # the size of key_padding_mask is [1, batch_size, 1, src_time_steps]
    key_padding_mask = key_padding_mask.unsqueeze(dim=1).unsqueeze(dim=0)

    # the size of key_padding_mask now is
    # [num_heads, batch_size, tgt_time_steps, src_time_steps]
    key_padding_mask_ = key_padding_mask.repeat(self.num_heads, 1, tgt_time_steps, 1)

    # change the size of attn_weights to [num_heads, batch_size, tgt_time_steps, src_time_steps]
    attn_weights = attn_weights.contiguous().view(self.num_heads, batch_size,
    tgt_time_steps, -1)

    # we have the same dimensionaility for key_padding_mask_ and attn_weights,
    # now we can add the mask
    attn_weights.masked_fill_(key_padding_mask_ == True, float(-1e10))

    # reshape
    attn_weights = attn_weights.contiguous().view(self.num_heads * batch_size,
    tgt_time_steps, -1)
if attn_mask is not None:
    attn_mask = attn_mask.unsqueeze(dim=0)  # [1tgt_time_steps, src_time_steps]
    # now the attn_mask is of dimension
    # [self.num_heads * batch_size, tgt_time_steps, src_time_steps]
    attn_mask_ = attn_mask.repeat(self.num_heads * batch_size, 1, 1)
    # add the mask
    attn_weights.masked_fill_(attn_mask == float("-inf"), float(-1e10))

attn_weights = F.softmax(attn_weights, dim=-1)
# F.dropout(attn_weights,p=self.attention_dropout,training=self.training)
attn = torch.bmm(attn_weights, v)
attn = attn.contiguous().view(self.num_heads, batch_size, tgt_time_steps, self.head_embed_size)
attn = attn.transpose(0, 2)
attn = attn.contiguous().view(tgt_time_steps, batch_size, self.num_heads * self.head_embed_size)
attn_weights = attn_weights.contiguous().view(self.num_heads, batch_size,
tgt_time_steps,-1) if need_weights else None

return attn, attn_weights
```

MODEL	TRI_LOSS	VAL_LOSS	TST_LOSS	VAL_PER	TST_PER	TST_BLEU
BASELINE	2.131	3.32	3.33	27.5	28.1	11.20
E_2_D_3	2.323	3.38	3.45	29.4	31.6	9.79
LEXICAL_MODEL	1.811	3.19	3.27	24.3	26.4	13.68
TRANSFORMER	1.309	3.85	3.85	47.2	47.1	11.00

Table 3: Compare the baseline model and lexical model

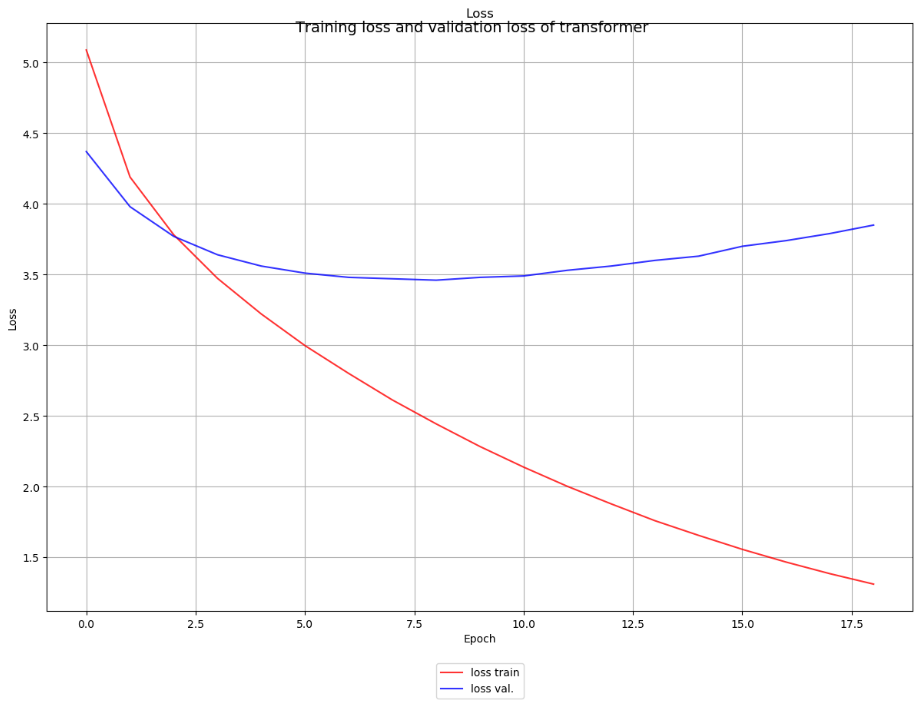


Figure 5: the performance of transformer on training set and validation set

(b) Result analysis

The results have been showed in Table 3. we can see that transformer model have a worse performance on the development and test sets than all the previous LSTM-based models. We believe that this is because the over-fitting problem of the transformer model is far more serious than that of the LSTM-based models.
For one thing, this can be seen from Table 5. The difference between Transformer's training set loss and validation set loss was the largest of the four models. For another, from Figure 5, we observe that the loss of validation set first decline at very early time and then start to increase until end of the training, while the loss of training set is continuously decreasing.
There are two factors which can contribute to the overfitting in the transformer. First, the training set is not big enough in this project. specifically speaking, we only have 10000 sentences in training set which is much smaller than the training set used to train NMT system like Google translation. Secondly, for architecture in transformer, here we use two layers encoder and two layers decoder. We believe that this complex structure also exacerbates the problem of overfitting.
There are some methods which can improve the performance. Firstly, some methods to alleviate overfitting, such as reducing the number of encoders and decoders or increasing the training set. Secondly, compare with the paper [2], we can increase the number of head numbers in multi-head attention which can help model to attend to information from different representation subspace from different positions. Thirdly, in the code, we use the word embeddings which are generated randomly. However, in paper[2], they use the pretrained word embeddings which have embedded the words relationship. Therefore, we can also change the word embeddings method.

References

[1] Toan Q. Nguyen and David Chiang. Improving lexical choice in neural machine translation. *CoRR*, abs/1710.01329, 2017.

[2] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.