1 Q1

1.1 (A)

1. final_hidden_states.size: [D*self.num_layers, batch_size, self.hidden_size], where D=1 if self.bidirectional == False, otherwise, D=2
final_cell_states: [D*self.num_layers, batch_size, self.hidden_size], where D=1 if self.bidirectional == False, otherwise, D=2

2. When self.bidirectional is set to True, the encoder layer will process input sequence in both left-to-right and right-to-left two directions, which means that there will be two pairs of hidden_states and cell_states after this layer, for left-to-right and right-to-left respectively. Thus, we need to initial final_cell_states and final_hidden_states with size(0) == 2*self.num_layers, for further concatenated the output final_cell_states and final_hidden_states eventually.

3. The final_cell_states is basically the global or aggregate memory of the LSTM network over all the timesteps, which should contain information of all words right from the start of the last word (i.e. all the words in the sentence). This is an element that exists in both LSTM and RNN.

The final_hidden_states represents the characterization of the last timestep's data that can illustrate how the specific hidden state is more concerned with the most recent time-step. This is an element that only exists in LSTM.

1.2 (B)

1. src_mask.size: [batch_size, 1, src_time_steps]
attn_weights.size: [batch_size, 1, src_time_steps]
attn_context.size: [batch_size, output_dims]
context_plus_hidden: [batch_size, input_dims + output_dims]
attn_out = [batch_size, output_dims]

2. From the encoder layer, we get attention scores and the hidden state as encoder output. Then, we apply masking if src_mask exists. Through the softmax function, we can get the attention weights. An attention context vector is defined as the sum of multiplication between each attention weight and encoder output e for each time step, so we perform a product between attention weights and encoder output to get the attention context vector.

3. Because we want to prevent the model from accessing the padding tokens during the decoding layer. Due to the difference in input sequences' length, the model needs padding for making up those short sequences. By applying a mask to those padding tokens' attention scores, we can make sure model after the softmax function the weight of those padding tokens is 0 so that the padding wouldn't affect the further calculation.

1.3 (C)

1. projected_encoder_out.size: [batch_size, output_dims, src_time_steps]
attn_scores.size:[batch_size, 1, src_time_steps]

2. src_projection project the encoder_out into the specified output_dim. Through the similarity product between tgt_input.unsqueeze(dim=1), projected_encoder_out get the attention score, the more similar between two vectors the higher score will get.

3. torch.bmm() is used as the batch matrix multiplication to compute the attention scores between decoder hidden state and encoder output, which could be understood as calculating the similarity between encoder and decoder tensor.

1.4 (D)

1. tgt_hidden_states.size: [len(self.layers), batch_size, self.hidden_size]
tgt_cell_states:[len(self.layers), batch_size, self.hidden_size]
input_feed:[batch_size, self.hidden_size]

2. First, we check whether the cached_state exists or not. If exists, usually the decoder is used for incremental generation and we can directly initialize the decoder state with states stored in cached_state. Otherwise, we initialize tgt_hidden_states, tgt_cell_states and input_feed with zero tensors.

3. Cached_state does not exist, when the value of incremental_state is None, or the full_key is not in incremental_state.

4. input_feed contains the output hidden state or the attention vector from the previous timestep. Later it will be concatenated with current target token embedding as the input feeding into the decoder layer to help the model know the previous sequence information.

1.5 (E)

1. input_feed.size: [batch_size, self.hidden_size]
step_attn_weights = [batch_size, src_time_steps]
attn_weights = [batch_size, tgt_time_steps, src_time_steps]

2. Attention is integrated into the encoder by calculating the attention vector (input_feed) from the current decoder state and the output of the encoder. By concatenating input_feed into the decoder output, through final_projection, we can then make the prediction.

3. Because the previous target state is useful to make the next state's prediction. With prediction made in previous timesteps, the attention could get more information from the previous sequence that not only from source sentence but also from part of target sentence. The attention scores will be calculated for the next word prediction. Similarly, the current state prediction will also be concatenated with the previous context to serve as the next timestep's decoder input.

4. The dropout layer is used to prevent the model from overfitting, by dropping certain information in the input_feed before feeding into the decoder. Thus, the decoder will gain good performance in general.

1.6 (F)

1. output.size: [batch_size, tgt_time_steps, len(dictionary)]

2. line 1: We get the output of model by feeding the model with sample['src_tokens'], sample['src_lengths'], sample['tgt_inputs'], where sample['src_tokens'] is the sentence in source language, sample['src_lengths'] is the source language's sentences' lengths and sample['tgt_inputs'] is the input sentences in the target language.

line 2: The average training CrossEntropyLoss is calculated given prediction output and sample['tgt_tokens'] (i.e. label).

line 3: We use the loss to perform backpropagation for updating the model parameters.

line 4: The norm is computed over all gradients together with the default clip threshold of gradients = 4.

line 5: Updating the model parameters.

line 6: Reset the gradients to zero at the end of each batch.

2 Q2

1. There are 124111 tokens and 8329 word types in English. There are 112621 tokens and 12505 word types in German.

2. In English, There are 3910 word tokens will be replaced as <UNK>, and the total vocabulary size will be 4420. In German, There are 7460 word tokens will be replaced as <UNK>, and the total vocabulary size will be 5046.

3. From our inspection, tokens with numbers account for 6.1% and 3.4% in English and German unique tokens respectively. Some wrong training examples (such as line 198 in train.en and train.de, '*5 - 10*' in English is mistranslated to '**510**' in German) may lead to low accuracy for the NMT system dealing with numbers. However, in most cases, the numbers should be left intact to the target language after the NMT model. NMT system still treats numbers the same as other unique tokens, which may lead to worse performance.

Generally, the problems with numbers can be classified into two types, one is the wrong position after translation including numbers missing, e.g. test.en line 408, the target sentence is '**the first aspect, which relates to amendments nos 9 , 14 and the second part of 17 , concerns veteran vehicles .**', but the model output is '**the first point concerns the amendments , the second situation will take place on the second point .**' without any numbers inside.

And the other one is the wrong number after translation. For instance, in line 242 of test.en, the target sentence is '**europe 2020 (motions for resolutions tabled) : see minutes**', but the translation result is '**europe) (rule 142) : see minutes**', where number '2020' is incorrectly translated to '142' through our NMT system. The possible solution of addressing the number problem would be changing the tokenization process by keeping all the number tokens rather than replacing them with $< UNK >$.

Furthermore, the tokens with only alphabets account for the biggest proportion in both English and German unique token. Through inspection, we find that there are three common phenomena.

First is that rare words are their inflectional and derivational form, such as 'lagging', 'learnt' and 'legislators'. In view of this situation, the feasible improvement measure is identifying those inflectional and derivational forms and transforming those words into their root form, after getting the output of the NMT system, then we apply grammar checking block to modify the words in the output that is wrong in context gramma.

Second is that rare words are the composition of several common words, such as 'ad-hoc-mechanismus', 'agri-gruppe' and 'gigabyte'. In view of this situation, the feasible improvement measure is to apply subword-tokenization and subword embedding to obtain the subword tokens' information, and research [2] prove the feasibility of our solution.

Third is that rare words are the abbreviations of certain phrases, such as 'asap'. In view of this situation, the feasible improvement measure is building the dictionary of common abbreviations and their corresponding phrases. Changing those abbreviations during the data pre-processing phase, before being fed into the NMT system.

4. There are a total of 754 unique vocabulary tokens which are the same in both languages. After our inspection, we found that there are 168 tokens with digits, which accounts for 22.3% of the total same tokens. Another large group is the proper nouns, such as 'alex', 'cardiff' and 'fischbach'. In our dataset, there are lots of sentences containing such proper nouns, and under certain contexts, a proper noun like names could give model correct guidance for better understanding the information in a large context.

We can also leverage this similarity between English tokens and German tokens to solve the problem that the model has bad performance on low-resource tokens. We can mark these data during data preprocessing, and when the model encounters these marked tokens during learning, it chooses to copy them directly, rather than replaces them with $< UNK >$. However, we still need to consider the position of the same proper noun in a sentence according to the grammar in different languages.

5. First, sentence length. We believe that when the length difference between the two languages is large, the effect of the model will be worse. It is more difficult for alignment between one language

sentence with short length and another language sentence with long length. Because there will be one-to-many and many-to-one alignments and these kinds of alignment is harder to learn by NMT system than one-to-one alignment.

Second, tokenization process. We hold that different ways of tokenization lead to different performance of NMT system, and reasonable tokenization process will lead to better performance of NMT system.

For example, a Chinese sentence '我想吃饭' could be translated into English as 'I want to eat' by using Google Translate. The different alignments between two sentence based on different tokenization process are shown as follow Figure 1 and Figure 2. With the same length and no unknown token, the correct tokenization and corresponding token alignment are shown in Figure 1, where all the tokens in both languages can find their corresponding token. But if we change our way of tokenization in Figure 2, we then find that the Chinese character '饭' and the English word 'to' do not find their token can be aligned with. NMT system may learn the alignment incorrectly due to a large number of tokens that have no alignments caused by the wrong tokenization process.

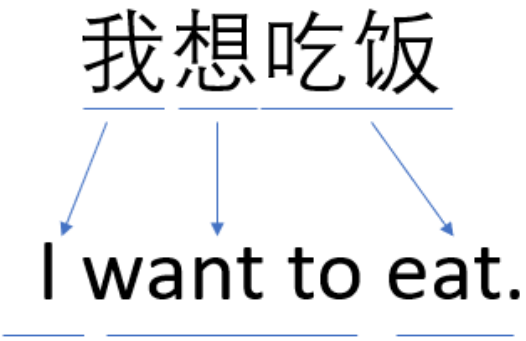

Figure 1: Example for Q2.4(1).

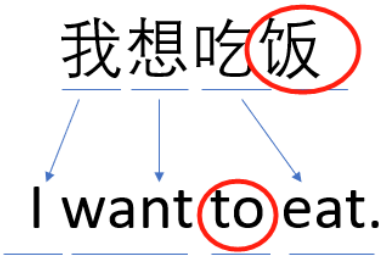

Figure 2: Example for Q2.4(2).

Third, unknown words. We hold that as the number of unknown tokens increases, the performance of the NMT system decreases. Considering the extreme situation that two language corpus are all unique tokens, then NMT system will replace all the tokens into $< UNK >$ during training session. Obviously, we can imagine that NMT system could only learn the relationship between the length of source and target language, rather than their alignments. Finally, $< UNK >$ will randomly be replaced by unique tokens in the corpus, which is more like a completely random process than a NMT system. In contrast, as the number of unknown tokens decreases, this extreme case will improve and the model will gradually learn normal translation relations between source and target.

3 Q3

1. Greedy decoding cannot guarantee to find the best translation. Because at every timestep greedy decoding only choose one word with the highest probability, however, it wouldn't be sure about that the local best will finally lead to global best. For example, given the reference sentence 'he hit me with a pie', during model prediction, at timestep 3 model generate 'he hit a' with the third word 'a' rather than 'me' based on the probability, because 'a' more likely to appear than 'me', i.e., almost all the similar sentence start with 'he hit' and similar source language sentence input, 'me' will never or less likely to be generated at the third timestep.

Another problem of greeed decoding is prediction in loop which means that greedy decoding method tends to stuck on a particular word or sentence and repetitively assigning these sets of words the highest probability again and again. For example, one possible output is 'the government is the important part of the government is the important part of the government is the important part of the government is the important part of...', where the correct output should be 'the government is the important part of the state'. Just because 'state' is less likely to be appeared after 'the government is the important part of the' than 'government', this problem occurs.

2. The key idea of beam search is that, on each step of decoder, keep track of the k most probable partial translations, where k is the beam size.

First, we define the BLEU score as

$$\sum_{i=1}^{t} logP_{LM}(y_i|y_{<i}, x)$$

i. current prob, current state = decoder(previous word, previous state) # use the decode to generate the probability of all word at current step

ii. current word list = argmax_k(current prob); score = score(current word list) # find out the k words with largest local probability(Take top k words and compute BLEU scores), where argmax_k(current prob) is a function that generate top k words with the largest local probability based on probability of all word at current step.

iii. for each one in k hypotheses: previous word = current word[n]; previous state = current state[n] # prepare for the next step decoding for all k candicates (For each of the k hypotheses, find top k next words and calculate scores)

iv. current word list = argmax_k(current prob); Of these k^2 hypotheses, just keep k with highest scores. #

v. When word.next == $< END >$ **stop** #When a hypothesis produces $< END >$, that hypothesis is complete. # Beam seach will stop expanding a hypothesis when reaching the symbol $< END >$. As a result, we will have k best candidates for output.

3. First, we can change the value of beam size k more bigger, so that there will be more candidates in the end. With more candidates, the probability of longer sentence will also increase. However, we also need to take the computing resource consumption into consideration, so that beam size k cannot be increased blindly.

Second, we can change the score function to:

$$\frac{1}{t}\sum_{i=1}^{t} logP_{LM}(y_i|y_{<i}, x)$$

which could normalize the original target function by length. For sentence candidates with similar $\sum_{i=1}^{t} logP_{LM}(y_i|y_{<i}, x)$ value, we tend to choose the most reasonable sentence with proper BLEU score and length.

4 Q4

1.

```
python train.py --encoder-num-layers 2 --decoder-num-layers 3  --save-dir checkpoints/
    checkpoints_new/ &> log/train_encoder2_decoder3.log $
```

2. We could see from Table 1, the multilayer model in Q4 is worse than baseline in all Training Loss, Validation Perplexity and Test BLEU. Multilayer model tends to have higher training loss, higher validation perplexity and lower BLEU score in test.

Model	Training Loss	Validation Perplexity	Test BLEU	Paramater Number	Stop Epoch
Baseline	2.141	27.3	11.11	1456644	99
Multilayer	2.412	30.2	9.37	1820164	89

Table 1: Comparison of different models.

The possible reason behind is because of the early stop. The Multilayer model stop training at epoch 89, while baseline model stops at epoch 99. As we can see from the total parameter contained in both model, Multilayer model's parameter number is more than baseline model, which means that Multilayer model is easier to convergence than baseline. Early stop happens if there is no improvement in validation set, then training phase stops. Then, the model applied in test set also reveive a worse performance. Therefore, training set, dev set, and test set performance of Multilayer model are all worse.

5 Q5

As shown in Table 2, our lexical model performs better on all Training Loss, Validation Perplexity and Test BLEU, compared with our baseline model.

Model	Training Loss	Validation Perplexity	Test BLEU
Baseline	2.141	27.3	11.11
Lexical	1.836	24.1	13.06

Table 2: Comparison of different models.

Paper [1] proposed lexical attention mode for improving low-resource language pairs. As mentioned in Q2, there are about 46.9% (3910/8329) tokens replaced by <UNK> in English and about 60.0% (7460 / 12504) tokens replaced by <UNK> in German. So the low-resource token rates are both large in two languages. This is the reason why lexical attention mode works in this dataset.

Following are the examples that rare word in sentences could be correctly replaced as unknown token correctly only through lexical attention model. And we also find that both rare tokens are poeple's names, which both start with token 'mr'. These are the evidences that lexical model could perform better than baseline model in translating rare words.

In test.en file's line 141, the word "lange" could be correctly predicted in lexical model while baseline model couldn't. Example is shown in the following three sentence:

- ref:i experienced this in belgium for nine years , mr **lange** .
- baseline:this is a number of points in belgium , mr solana .
- lexical model: this is why i voted in belgium , mr president , mr **lange** .

In test.en file's line 4321, the word "watson" could be correctly predicted in lexical model while baseline model couldn't. Example is shown in the following three sentence:

- ref: thus , i am at a loss to understand what mr **watson** , the chairman of the group of the alliance of liberals and democrats for europe , has just said .
- baseline:i therefore agree with mr von , mr prodi , mr poettering , mr poettering , mr poettering .
- lexical model: i do not understand what mr **watson** , the woman of the woman proposed the eu has said .

6 Q6

6.1 (A)

1. Embeddings.size: [batch_size, src_time_step, num_features]

2. First, position and order information are very important. They contain both grammatical structure and semantic information. Second, the Transformer model use attention rather than basic sequence model, e.g. RNN, CNN. Based on attention mechanism, we can only get the probability that how previous words affect the next potential outcome word, thus, the model would lost order information during training and the trained model cannot tell us how the relative and absolute position information of each word in the sentence. To address this problem, we apply Positional Encoding to carry word order signal to the word vector to help the model learn this information.

3. Because self-attention mechanism in Transform just cares about how each previous word affects the next output word without the order of the input, which is called permutation equivariant. Although we use embedding similar to for the LSTM, model still cannot learn the position information of the input sequence. Thus, Position Embedding is necessary for transformer.

6.2 (B)

1. self_attn_mask.size: [tgt_time_steps, tgt_time_steps]

2. The purpose of self_attn_mask is to prevent model from 'cheating' (computing the attention scores from the future input words). In worst case, model will learn 'copy' from the future input word, which won't improve the performance of model on unseen datasets.

3. The functions of encoder and decoder are different. The encoder aims at learning a good representation of the current input. For better representation, encoder can apply bidirectional RNN model, that takes both left-to-right and right-to-left contexts. While decoder aims at generating a sequence word by word, which equals to the prediction task based on input and previous generated sequence. Without masking, model would easily learn prediction the word at timestep t by simply copying the word at timestep (t+1) if exists. Thus, only decoder needs self_attn_mask.

4. Because model feeds the decoder token by token in incremental decoding,i.e., we do not provide decoder with inputs words that are in the future timesteps, which avoids model 'cheating' from getting information from future words.

6.3 (C)

1. forward_state.size: [batch_size, tgt_time_steps,len(dictionary)]

2. The linear projection is used to project the decoder output tensor (forward_state) to a tensor with the size of the vocabulary.

3. [batch_size, tgt_time_steps, len(dictionary)]

4. If features_only=True, the output will return the feature tensor, i.e. hidden states, from the final decoder layer.

6.4 (D)

1. encoder_padding_mask.size: [batch_size, seq_len], where seq_len is the fix number in each batch that could unify all the sequence in to same length.
state.size(before): [src_time_steps, batch_size, num_features]
state.size(after): [src_time_steps, batch_size, num_features]

2. Because the input sequences may vary in its length, encoder_padding_mask is used to make all the input sequences the same length, but make the model only care about the original input sequence without the padding tokens.

3. [tgt_time_steps, batch_size, embed_dim].

6.5 (E)

1. state.size(before): [tgt_time_steps, batch_size, embed_dim]
encoder_out: [tgt_time_steps, batch_size, embed_dim]
state.size(after): [tgt_time_steps, batch_size, embed_dim]
attn.size: [num_heads, batch_size,tgt_time_steps, encoder_out.size(0)]

2. Self-attention exists in both encoder and decoder, while encoder attention only exists in decoder.

The encoder attention mechanism occurs between all elements in target's element query and source key and value generated as output in encoder. While Self-attention is the attention mechanism that occurs between words within source or between elements within target. For instance, when calculating the weight parameters in Transformer, the text vector is converted into the corresponding KQV, and only the corresponding matrix operation is required at the Source, and the information in the Target is not used.

In addition, self-attention requires query, key and value have equal sizes whereas encoder attention doesn't have this limitation.

3. key_padding_mask is used to keep model from learning the padding tokens, while attn_mask is used to prevent the model 'cheating' from get the attention score of future unseen word when making prediction, in case the model only learns 'copying' from next timestep (t+1)'s word to predict the current timestep t's word.

4. We don't need to give attn_mask, because we can see from the code that the previous decoder layer already apply this in its self-attention layer. Thus,the query computed from the decoder has already avoided the accessing to the future input words.

7 Q7

Code:

```
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)]
    # TODO: REPLACE THESE LINES WITH YOUR IMPLEMENTATION ------------------------ CUT

    # Get Q,K,V with size (batch size * num heads, target length, head_embed_size)
    Q = self.q_proj(query).contiguous().view(-1, batch_size, self.num_heads, self.head_embed_size).transpose(0, 2)
    K = self.k_proj(key).contiguous().view(-1, batch_size, self.num_heads, self.head_embed_size).transpose(0, 2)
    V = self.v_proj(value).contiguous().view(-1, batch_size, self.num_heads, self.head_embed_size).transpose(0, 2)

    Q = Q.contiguous().view(self.num_heads * batch_size, -1, self.head_embed_size)
    K = K.contiguous().view(self.num_heads * batch_size, -1, self.head_embed_size)
    V = V.contiguous().view(self.num_heads * batch_size, -1, self.head_embed_size)
    # Q*K^T
    intermediate_result = torch.matmul(Q, K.transpose(1, 2)) / self.head_scaling

    #  masking
    if key_padding_mask is not None:  # avoid concerning about the padding elements
        # mask the attention weights
        key_padding_mask = key_padding_mask.unsqueeze(dim=1).repeat(self.num_heads, 1, 1)
        intermediate_result.masked_fill(key_padding_mask, float('-inf'))

    if attn_mask is not None:  # avoid concerning about the following words
        intermediate_result += attn_mask.unsqueeze(dim=0)

    # apply softmax
    attn_weights = F.softmax(intermediate_result, dim=-1)
    # apply dropout layer
    attn_weights = F.dropout(attn_weights, p=self.attention_dropout, training=self.training)
    # proudction with V
    intermediate_result = torch.bmm(attn_weights, V)
    # change size fitted for output projection
    intermediate_result = intermediate_result.contiguous().view(self.num_heads, batch_size, -1, self.head_embed_size).transpose(0, 2)
    intermediate_result = intermediate_result.contiguous().view(-1, batch_size, self.num_heads * self.head_embed_size)
    # finally through the output projection, get attn with size [tgt_time_steps, batch_size, embed_dim]
    attn = self.out_proj(intermediate_result)
    # TODO: ---------------------------------------------------------------------- CUT
```

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

    return attn, attn_weights
```

The Table 3 below shows the performance comparison between models we trained. We can clearly see that our Transformer model performs best in training loss, but worse than all other models in validation perplexity, and achieves the second best BLEU score on test set, where the best one is still our lexical model. Although our final BLEU score is better than our baseline, the disparity is not very huge. When we combine these results together, it is not hard to see that the Transformer model has a problem of overfitting.

Model	Training Loss	Validation Perplexity	Test BLEU	Paramater Number	Stop Epoch
Baseline	2.141	27.3	11.11	1456644	99
Multilayer	2.412	30.2	9.37	1820164	89
Lexical	1.836	24.1	13.06	1748040	55
Transformer	1.356	42.3	11.98	2707652	21

Table 3: Comparison of different models.

There are several possible reasons may lead to that.

First, the model size. We can see from Table 3 that Transformer model has 2707652 parameters which is far more than other models. We can assume that the model implementation is right, more parameters means model have greater power to fit the underlying pattern in the data faster, which could explain why the Transformer model only need 21 epochs to stop training. On the other hand, if the data is full of noise, model will also capture that, which usually leads to overfitting.

Second, the training dataset size. Although our dataset have 10,000 pairs of sentences for training, it is still too small when it compares with the dataset used in the paper [3]. with about 4.5 million sentence pairs. From our experiment result, we can see that Transformer model are easy to fast convergence, which leads to overfitting.

Since overfitting is very likely to happen in Transformer model training, we could give several possible way to improve its performance by solving overfitting.

First, Regularization. Reasonable use of dropout can effectively prevent overfitting. The paper [4] presents three different levels of dropout ,from fine-grained to coarse-grained, which are feature dropout, structure dropout and data dropout. Feature dropout , the traditional dropout technique, is usually applied to the hidden layer of the network. Structure dropout is a coarse-grained dropout designed to randomly drop certain substructures or components in a model. Data dropout, as a data augmentation method, is usually used to randomly drop some tokens of the input sequence. By integrating multiple dropouts, the probability of model overfitting can be reduced.

Second, dataset size. We could change our dataset to a greater dataset that contains more training pairs, e.g. standard WMT 2014 English-German dataset in [3]. With better dataset, Transformer model is less likely to be overfitting.

References

[1] Toan Q Nguyen and David Chiang. Improving lexical choice in neural machine translation. *arXiv preprint arXiv:1710.01329*, 2017.

[2] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

[3] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[4] Zhen Wu, Lijun Wu, Qi Meng, Yingce Xia, Shufang Xie, Tao Qin, Xinyu Dai, and Tie-Yan Liu. Unidrop: A simple yet effective technique to improve transformer without extra cost. *arXiv preprint arXiv:2104.04946*, 2021.