Question 1

Performance of the baseline model:

Training Loss Last Epoch	Validation Perplexity Last Epoch	Test Set BLEU
2.126	27.3	10.78

A. When self.bidirectional is set to True, each layer of encoder will process input tokens in two directions, i.e. from left to right and from right to left. Consequently, each layer will produce two final hidden states and two final cell states: one final hidden state and one final cell state from the left to right processing. In addition, there are also one final hidden state and one final cell state from the right to left processing. The final hidden state and cell state from each direction will be concatenated (left-to-right final hidden state + right-to-left final hidden state, and left-to-right final cell state + right-to-left final cell state) before they are passed to the next layer.

   The difference between `final_hidden_states` and `final_cell_states` is the `final_cell_states` contains the long-term memory of the network. This is a component that exists only in LSTM and does not exist in RNN. On the other hand, `final_hidden_states` contains information mostly from recent time steps because the hidden state of LSTM is updated at every time step. This is a component that exists in both LSTM and RNN.

B. Here is how to calculate the attention context vector. Given attention scores which are calculated from the encoder output and the target hidden state, we apply the `src_mask` (from encoder) to the attention scores if `src_mask` exists. Then, we calculate the attention weights from the attention scores using the softmax function (squashing attention scores to values in the range between 0 and 1). Theoretically, attention context vector is defined as the sum of multiplication between each attention weight and its corresponding encoder output for each time step. Thus, we calculate the attention context vector by performing a batch matrix-matrix product between attention weights (of size `[batch_size, 1, src_time_steps]`) and the encoder output (of size `[batch_size, src_time_teps, output_dims]`) which results in a tensor of size `[batch_size, 1, output_dims]`. After that, we squeeze the result to remove all dimensions of size 1. This results in the attention context vector of size `[batch_size, output_dims]`.

   We need to apply a mask to the attention scores in order to prevent the network from attending padding tokens when decoding. Input sentences might be padded in order to make them have the same length. By applying the mask to the attention scores, the weights of padding tokens would be zero after softmax function and thus, the encoder values for the padding tokens will not have influence when calculating the attention context vector.

C. Attention scores are calculated by first, performing a linear projection on the encoder output tensor using the function `self.src_projection`. In our coursework, the input

dimension and the output dimension of the linear projection is the same ([batch_size, src_time_steps, output_dims]). However, in a more general setting, this linear transformation allows the size of the encoder output vector to be different from the size of the target (decoder) hidden state [1]. After that, we perform a batch matrix-matrix product between the target hidden state (size: [batch_size, 1, input_dims]) and the transposed projected_encoder_output (size: [batch_size, output_dims, src_time_steps]) which result in an attention score tensor of size [batch_size, 1, src_time_steps].

The role matrix multiplication torch.bmm() plays in aligning encoder and decoder representations :
For each batch, the matrix multiplication is used to compute the attention score (size: [1, src_time_steps]) between the decoder hidden state (size: [1, input_dims]) and encoder output at all time steps (size: [output_dims, src_time_steps]). These attention scores control the alignment between the encoder and decoder representations.

D. Here is how the decoder state is initialized. Firstly, we checked whether a cached_state (cached previous decoder states) is present for this model instance. The cached_state is usually present when the decoder is used for incremental, auto-regressive generation. If cached_state is present, we initialized the decoder state with the states stored in cached_state. However, if cached_state is not present, we initialize the decoder hidden states, cell states, and input_feed with tensors of zeros.

   Cached_state == None when the value of incremental_state variable is None, or the requested full_key is not present in the incremental_state, which means there are no cached previous decoder states.

   input_feed stores the attentional vector (if attention is used) or the decoder hidden states (if attention is not used) from the previous timestep[2]. It will be concatenated with the current token embeddings as the input to the decoder in order to inform the model about previous alignment decisions.

E. Attention is integrated into the decoder by making use of the current decoder state (in addition to encoder output and encoder mask) to calculate the attentional vector (input_feed), which is later used to make decoding prediction (i.e. fed through the self.final_projection).

   The attention function is given the target (decoder) state as one of its inputs because the target state is needed in the calculation of attention scores (i.e. to calculate how similar the target state with the encoder output at each time step), and the target state will also be concatenated with the attention context to be fed as the next step's decoder input.

[1] Neubig, G., 2017. Neural machine translation and sequence-to-sequence models: A tutorial. *arXiv preprint arXiv:1703.01619*. Page 52
[2] Luong, M.T., Pham, H. and Manning, C.D., 2015. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*.

Dropout layer is used as a form of regularization to prevent the network from overfitting[3]. Dropout layer will zero out some elements in the attentional vector (input_feed) to prevent the decoder in the next time step from relying too much on the previous states for making predictions.

F. The lines of code perform the following operations:
    1. First, we perform forward pass on the input batch through the instantiated model by passing
        - Sample['src_tokens'] = the sentences (sequence of tokens) in the source language
        - Sample['src_lengths'] = the sentence lengths in the source language
        - Sample['tgt_inputs'] = the correct sentences in the target language

       The forward pass produces output predictions which are stored in the variable output.
    2. After that, the average training loss per sentence (i.e. cross entropy loss) is computed based on the output predictions and the gold labels (sample['tgt_tokens'']).
    3. Next, we perform backpropagation (compute gradients of the loss function with respect to model parameters) by calling "loss.backward()".
    4. Moreover, we also clip the gradient norms so that they do not exceed a certain threshold (default threshold = 4.0). This step is necessary for the stability of the training process, for instance, to avoid the exploding gradient problem.
    5. Next, we update the model parameters by calling "optimizer.step()"
    6. Lastly, we reset gradients to zero ("optimizer.zero_grad()") before continuing to the next input batch so that gradients are not accumulated across input batches, i.e. each parameter update is influenced only by instances in the corresponding input batch.

[3] Zaremba, W., Sutskever, I. and Vinyals, O., 2014. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*.

Question 2

In this work, we used the function split() instead of split(" "). Moreover, we also included punctuations as word tokens.

1.
The total count and the number of word types in each language is as follows:

Language	Total Count	Word Types	Type to Token Ratio
English	124,031	8,326	8,326 / 124,031 = 0.0671
German	112,572	12,504	12,504 / 112,572 = 0.1111

2.
The number of word tokens that will be replaced by <UNK>, and the subsequent total vocabulary size is as follows:

Language	Replaced by <UNK>	Vocabulary Size
English	3,909	8,326 - 3,909 + 1 = 4,418
German	7,460	12,504 - 7,460 + 1 = 5,045

3.
Based on our inspection, a specific type of word which will be commonly replaced is **numbers**. We used NLTK POS tagger to tag the words and found that more than 50% of numbers (CD) are replaced by unknown.
Example from English : 4.8
Example from German : 11.40

At test time, replacing most numbers in the training set with <UNK> would result in the poor performance of the model when generating numbers in the translation. At least, it can be manifested in two ways:

1. The model would not generate numbers at the position where they should be.
   Because the model would have very limited training examples to learn the correct distribution for numbers
2. The model would generate numbers at the position where they should not be.
   This case is largely because most numbers are replaced by <UNK> token, just like other words from different classes, e.g. proper nouns. When the model learns the distribution of the <UNK> token, it learns a distribution which is a mix of the distribution of numbers and other word classes (e.g. proper nouns). Consequently, in the translation, when the model generates an <UNK> token, it might replace the

<UNK> with numbers at the position where they should have generated another word class instead (e.g. proper noun).

To provide a better illustration of the above explanation, let us have a look at the following translation:

Source sentence (German):
die zulässige obergrenze für den eu-haushalt beträgt 1,27 % des bsp der mitgliedstaaten .

Gold label (English):
the permitted ceiling in the eu budget is 1.27 % of the combined gnp for the member states .

Translation result (English):
the chinese 2020 is for the eu 's increase % of member states .

We could see that the 2 cases appear in the translation above:

1. Case 1: Numbers did not appear before the word "%" . Based on the source sentence, they should have appeared at that position. Moreover, intuitively, "%" is almost always preceded by numbers. However, our model failed to learn this distribution because most percentage numbers in the training set are replaced by <UNK>.
2. Case 2: 2020 suddenly appeared at the wrong place. The 2020 appeared out of nowhere in the translation, preceding the word "is". One possible explanation is that the word "is" is commonly preceded by a proper noun. However, proper nouns often have only one occurrence, hence most of them are replaced by <UNK>. When the model generated the translation, it confused between generating a number or a proper noun in place of the <UNK> because both are often replaced by <UNK> in the training data.

Side Note:
Based on our observation, proper nouns were also often replaced by UNK. However, since our tagger tagged proper nouns and nouns with the same tag "NN" (because proper nouns were turned to lowercase after preprocessing), it was difficult for us to get the exact distribution of proper nouns which were replaced by UNK. On the other hand, we knew exactly that more than 50% of numbers (CD) were replaced by UNK. Hence, we chose numbers as our answer here.

4.
There are **1,460 unique vocabulary tokens which are the same in both languages**. Moreover, there are 754 tokens out of that 1,460 tokens (51.6%) which have only 1 occurrence, hence they will be replaced by <UNK>. In addition, a large number of these common tokens with 1 occurrence are proper nouns, such as `papua, nairobi, mccarthy, mitchell, xynthia, sarkozy, guinea, andersson, goya, delhi, unicef`. Proper nouns often carry important specific information in a sentence. Therefore, merely replacing them with <UNK> might lead to the problem of losing important information in the sentence.

We could exploit this similarity between English and German tokens in our model to alleviate the above mentioned rare word problem. One way is to copy rare words from the source text to replace corresponding <UNK> tokens in the generated target text[4]. This is a reasonable approach for our data, especially because many of the rare words are numbers (Q2.3) and proper nouns (Q2.4) which almost always have the same form either in English or German.

5.

Related to differences in sentence length between target and source language, we would argue that NMT systems' performance would be lower when translating from a language to another language with a very different average sentence length than the one with a similar average sentence length. For two language with more considerable differences in the average sentence length:

- Firstly, we think it should be more challenging to make the alignment between the source language and target language, thus hurt the performance
- Secondly, we think the translation could be unnatural if the average translation we produce is too long or too short from the average sentence length of the target language, and thus hurt the performance.

In terms of token ratios, we would argue that **the performance of NMT systems will be lower when translating from a language with a lower token ratio to another language with a higher token ratio.** Table in Q2.1 shows that German has a higher type to token ratio (TTR) compared to English (0.111 vs 0.067). Hence, we expect that English to German translation would have lower quality than German to English translation. To prove this hypothesis, we ran the baseline model in Q1 again, but performed English to German translation. The result is provided below

Translation	Test Set BLEU
German to English	10.78
English to German	9.37

As expected, the model performed more poorly when translating from English to German. One example of challenges in translating to a language with a higher token ratio is as follows. The word “the” in English can have 4 possible translations in German, i.e. “die”, “der”, “den”, “das”. The correct translation would depend on the noun following the article. Hence, translating English to German is more difficult because it requires more information about the context.

In terms of unknown word handling, Cho et al. (2016)[5] also shows that **the performance of NMT systems degrades rapidly as the number of unknown words increases**. Similarly, NMT systems also perform poorly on rare words[5]. Therefore, the use of proper unknown /

[4] Luong, M.T., Sutskever, I., Le, Q.V., Vinyals, O. and Zaremba, W., 2014. Addressing the rare word problem in neural machine translation. *arXiv preprint arXiv:1410.8206*.

[5] Arthur, P., Neubig, G. and Nakamura, S., 2016. Incorporating discrete translation lexicons into neural machine translation. *arXiv preprint arXiv:1606.02006*.

rare words handling is crucial to improve the performance of NMT systems. This will be proven in the Question 5 where we will apply a lexical attention mechanism[6] to the baseline model. Our experiment shows that the addition of a rare word handling mechanism successfully improves the NMT model performance.

[6] Nguyen, T.Q. and Chiang, D., 2017. Improving lexical choice in neural machine translation. *arXiv preprint arXiv:1710.01329*.

Question 3

1. Using greedy decoding might be problematic because **greedy decoding is not guaranteed to produce the best translation (i.e. translation with the highest probability) over all possible translations**[7]. This problem is because greedy decoding only selects the most likely word at each time step as the output, which might not lead to the most optimal translation in the long run.

   Example:
   For a given French sentence, consider the English translations as:
   1. I am visiting my grandmother tomorrow. (optimal)
   2. I am going to be visiting my grandmother tomorrow. (sub-optimal)

   In English, it is more common to find sentences containing "I am going to" compared to "I am visiting". Therefore it is likely that our MT model captures the following probability distribution from the training corpus:

   P(going | I, am) > P (visiting | I, am)

   If we use the greedy decoding, the model would likely to generate the translation 2 instead of translation 1, because the probability of "going" after "I am" is higher than the probability of "visiting" after "I am". However, an optimal translation would be translation 1 because it sounds more natural (has better fluency). Nonetheless, this cannot be achieved by greedy decoding if it only selects the most likely word at each time step as the output. This example is inspired from this article[8].

   Another possible problem is that greedy decoding will always produce the same translation since it always picks the word with the highest probability. So long as the probability distribution of words at each time step does not change (e.g. using the same random seed), it will always pick the same word to generate. This is in contrast to random search where each word has the same probability to be generated at each time step. Hence, each generation could produce a different result. Therefore, greedy decoding will always produce translation 1, whereas if we use random sampling, we could possibly obtain translation 2.

2.

Beam Search Procedure:
The beam search generates translation word-by-word from left to right while keeping a fixed number (beam) of active candidates at each time step[9].

In general, let x = input sequence, B = beam size, D = vocabulary set, |D| = vocabulary size
For all $y_t \in D$, we compute:

[7] Neubig, G., 2017. Neural machine translation and sequence-to-sequence models: A tutorial. *arXiv preprint arXiv:1703.01619*. Page 41
[8] Beam Search - A Search Strategy. Retrieved from https://hackernoon.com/beam-search-a-search-strategy-5d92fb7817f
[9] Dive into Deep Learning. Retrieved from https://d2l.ai/chapter_recurrent-modern/beam-search.html

$$
\begin{aligned}
P(Y^1{}_{t-1}, y_t|\ ) &= P(Y^1{}_{t-1}|\ )\, P(y_t \mid Y^1{}_{t-1},\ )\ , \\
P(Y^2{}_{t-1}, y_t|\ ) &= P(Y^2{}_{t-1}|\ )\, P(y_t \mid Y^2{}_{t-1},\ )\ , \\
&\cdots \\
P(Y^B{}_{t-1}, y_t|\ ) &= P(Y^B{}_{t-1}|\ )\, P(y_t \mid Y^B{}_{t-1},\ )\ ,
\end{aligned}
$$

where $Y^i{}_{t-1}$ are the best B previous sequence. We then pick the largest B values among B * |D| values.

For example, let D = {J, K, L, M, N} be the vocabulary, $\boldsymbol{x}$ be the input sequence, and beam size = 2.
At time step 1, suppose the tokens with the highest conditional probabilities $P(y_1|)$ are J and L.
At time step 2, for all $y_2 \in D$, we compute

$$
\begin{aligned}
P(J, y_2|\ ) &= P(J|\ )\, P(y_2|J,\ )\ , \\
P(L, y_2|\ ) &= P(L|\ )\, P(y_2|L,\ )\ ,
\end{aligned}
$$

Then, we pick the largest two among all possible 2x5 translations, say $P(J, K|\ )$ and $P(L, N|\ )$

At time step 3, we compute

$$
\begin{aligned}
P(J, K, y_3|\ ) &= P(J, K|\ )\, P(y_3|J, K,\ )\ , \\
P(L, N, y_3|\ ) &= P(L, N|\ )\, P(y_3|L, N,\ )\ ,
\end{aligned}
$$

Again, we pick the largest two among all possible 2x5 (beam_size * vocab_size) candidates, for instance $P(J, K, M|\ )$ and $P(L, N, M|\ )$.

Beam search will stop expanding a candidate when meeting a stopping criterion, e.g. generating the <EOS> (end of sentence) token. At the end of beam search, we will have B best candidates for output.

To implement the above procedure in the context of the coursework code, we need to modify the code in at least two places: translate.py (between line 76 and line 85) and the code in LSTMDecoder.forward.

1. **translate.py**
   We modify the code in such a way that at each time step t:
   1. We feed B possible previous sequence to the decoder instead of only one (line 79), and
   2. Find best B candidates instead of one single best candidate (line 81 - 82), and possibly some extra backoff candidates (line 83)
      The current code indeed finds two candidates: one as best candidate, another one as the backoff candidate in case that the best candidate is unknown. Hence, we can do similar things, i.e. finding more than B candidates, to replace any of the best B candidates which is unknown.

2. **LSTMDecoder.forward**
   Due to the abovementioned modification in translate.py, LSTMDecoder now receives more than 1 possible previous sequence. This might reflect in the size of tgt_input

which becomes [beam_size, batch_size, tgt_time_steps, num_features]. Hence, in the new LSTMDecoder.forward**,** we could loop over the dimension beam_size and perform the current forward procedure to each candidate. At the end of the function, we concatenate the decoder_output for each candidate, resulting in the decoder_output of size [beam_size, batch_size, tgt_time_steps, num_features]. This decoder_output will be returned to the caller in translate.py, which will later reduce the decoder_output to find topk (k = beam size) candidates (step 2).

3. a.

Decoder favours short sentences because short sentences tend to have larger sentence probability compared to long sentences. To illustrate this phenomenon, suppose we want to translate French sentences to English sentences. In machine translation, the probability of an English sentence E given a French sentence F is as follows[10]:

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

Where |E| is the length of the english sentence, e_t is the English word at time step t, and e_1^{t-1} is the generated sequence of English words so far. We can see that the probability of a sentence is made from the product of conditional probabilities of its words. As probability values are in the range of [0, 1], every time we add another word in a sentence, we reduce the probability of the whole sentence[7]. This explains why shorter sentences with less words have higher overall probabilities compared to the longer ones, and hence, favoured by decoders.

b.

Suppose we want to translate English sentences E to French sentences F. The length normalization formula is as follows[7]:

$$\hat{F} = argmax_F \ log \ P(F|E)/|F|$$

Where |F| is the length of the French sentence, and |E| is the length of the English sentence.

From the above formula, we argue that one possible problem is: **for sentences with the same probability P(F|E), the length normalization induces bias to the decoder to prefer shorter sentences (smaller |F|)**. Moreover, length normalization does not take into account the length of the source sentence |E|, although |E| could be a good indicator of whether the decoder should prefer a shorter or longer sentence in the case of both sentences have equal P(F | E).

Example:

We want to translate the English sentence “You are welcome” to French. Suppose we have two possible French translation with the same probability P(F | E) = 0.9

[10] Neubig, G., 2017. Neural machine translation and sequence-to-sequence models: A tutorial. *arXiv preprint arXiv:1703.01619*. Page 43

Sentence (F)	P(F | E)	|F|	$\hat{F}$
Je vous en prie	0.9	4	0.225
De rien	0.9	2	0.45

From the table above, the decoder will always choose “De rien” as the output translation. However, this translation might not be always optimal in terms of “fluency”. We will most likely find the sentence “You are welcome” in a formal context. Hence, it is likely that a more appropriate translation is “Je vous en prie”. Nonetheless, length normalization cannot infer this information because it does not take into account the length of the source language |E|. By taking into account |E|, we can approximate the proper length of translation (and hence, choose “Je vous en prie” instead) because formal sentences tend to be longer than its less formal equivalence.

Question 4

1.

```
COMMAND: train.py --save-dir
/afs/inf.ed.ac.uk/user/s20/s2015555/nlu/CW2/NLU-CW2/resultsQ4/Q4 --log-file
/afs/inf.ed.ac.uk/user/s20/s2015555/nlu/CW2/NLU-CW2/resultsQ4/Q4/log.out
--data
/afs/inf.ed.ac.uk/user/s20/s2015555/nlu/CW2/NLU-CW2/europarl_prepared
--encoder-num-layers 2 --decoder-num-layers 3
```

2.

Question	Training Loss Last Epoch	Validation Perplexity Last Epoch	Test Set BLEU
Q1	2.126	27.3	10.78
Q4	2.308	29.1	9.66

We could see from the table above that adding layers results in lower performances of the model compared to baseline performances in Q1. All of the three metrics show degradation: higher training loss, higher validation perplexity, and lower test set BLEU. Hence, there is no difference between the training, validation, and test set performance.

We would argue that this is because **the model had not reached training loss convergence but it was early stopped because the model was not improving on the validation set**. The additional encoder and decoder layers increased the complexity of the model. Hence, there were many more parameters to train compared to the baseline model, and bigger models tend to need more epochs to converge. However, this complex model caused the overfitting problem because the complex model is too flexible, which caused early stopping (epoch 96, meaning performance is not improving since epoch 86) occurred during the training process as the performance on the validation set is not improving. Therefore, the model shows a worse performance on all training, validation and test sets.

Question 5

Question	Training Loss Last Epoch	Validation Perplexity Last Epoch	Test Set BLEU
Q1	2.126	27.3	10.78
Q5	1.811	24.3	13.68

Apparently, the addition of lexical translation is beneficial to the model performance on all training, validation and test metrics. This is shown by the lower training loss, lower validation perplexity and higher test BLEU score compared to the baseline model.

This result is expected because the lexical attention model proposed by Nguyen and Chiang (2017) helps alleviate the problem of mistranslating rare words[11]. As shown in Question 2, rare words comprise a large number of word types in both English (3,909 / 8,326 = 47%) and German (7,460 / 12,504 = 60%) [12]. Therefore, adding a mechanism to improve rare words handling will likely to improve the model's performance on this data.

We found that neither of the models could translate the words that are replaced by the unknown token correctly. However, if we define rare words as the words appearing less or equal twice, then we found that the model in q5 translated 4 of rare words (3 of them are proper nouns and 1 of them is noun) correctly, whereas the baseline failed to translate any of them correctly. Below shows a few examples where the rare words are predicted correctly in the lexical model but not the baseline. This could be seen as the evidence of the new model translating the rare words better than the baseline.

Examples:

1. For the rare word "lange" in line 141:
    - Ref: i experienced this in belgium for nine years , mr lange .
    - Baseline: this is why i have been listening in my own country , mr president .
    - Q5: this experience was a few reason in belgium years , mr lange .
2. For the rare word "corbett" in line 352:
    - Ref: thank you for your request , mr corbett , which is also relevant .
    - Baseline: thank you for your speech , mr dimas , with a very important question .
    - Q5: thank you for your ask , mr corbett , with your own question .
3. For the rare word "fortress" in line 354:
    - Ref: the june list does not , however , wish to contribute to any fortress europe .
    - Baseline: but the material of the power is not a new country .
    - Q5: the council would not be able to do a fortress europe .
4. For the rare word "watson" in line 432:
    - 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 do not know what mr poettering has said by the rapporteur , mr van den said .
    - Q5: i do not understand what mr watson , the most woman of the eu has said .

[11] Nguyen, T.Q. and Chiang, D., 2017. Improving lexical choice in neural machine translation. *arXiv preprint arXiv:1710.01329*.

[12] These statistics only account for words with 1 occurrence. However, the term "rare words" could arguably include words with 2 or 3 occurrences. Hence, the real "rare words" percentage is bigger.

Question 6

A. The purpose of the positional embeddings in the encoder and decoder is to add information about the position of each word to the input embeddings of the Transformer. We cannot use only the embeddings similar to the ones for LSTM because the self-attention mechanism in Transformer operates on its inputs without taking into account the sequence ordering (permutation equivariant). Hence, we need to add positional embeddings so that if the order of our input changes, the input to the self-attention layer (input embeddings + positional embeddings) will also change.

B. The purpose of self_attn_mask is to prevent the network from "seeing" / computing attention scores for future input words. Hence, the network can only attend to previous and current input words when making the current prediction.

   We need it in the decoder because the decoder should generate a sequence word-by-word. This means that it may only see previous inputs and current input words when generating prediction. If we do not mask future input words, the decoder will not learn useful functions because it can see future inputs and hence, learn only to copy the immediate future input (t+1) for the current prediction (t). In contrast, we do not need the self_attn_mask in the encoder because it does not have a constraint to only attend up to the current input. In order to learn good representation of its input (by taking into account both right-to-left and left-to-right contexts), the encoder can have access to the whole input sequence.

   We do not need a mask for incremental decoding because in incremental decoding, we feed the decoder token-by-token. Hence, we do not provide the decoder with inputs from future time steps when it is generating prediction for the current time step.

C. We need a linear projection after the decoder layers because we need to project the decoder output (forward_state) to a tensor having the same size of the vocabulary size.

   The dimensionality of forward_state after this line would be the size of vocabulary (length of dictionary). More precisely: [batch_size, tgt_time_steps, num_vocabulary]

   If features_only=True, the output represents the features (hidden states) from the final decoder layer.

D. As the input sentences (tokens) might be padded in order to have the same length in that batch, encoder_padding_mask is used for preventing encoder from taking into account ("paying attention") to padding tokens when processing inputs.

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

E. Difference between encoder attention and self attention:

Encoder attention only exists in the decoder, whereas self attention exists in both encoder and decoder. Self attention performs multi-head attention mechanism by using source embedding (if in the encoder) or target embedding (if in decoder) as its query, key and value. On the other hand, encoder attention uses encoder output as its key and value, in addition to decoder states from self attention as its query. Therefore, self attention requires query, key and value have equal sizes whereas encoder attention does not have this constraint.

The difference between key_padding_mask and attn_mask: key_padding_mask is used to prevent the network from attending to padding tokens, whereas attn_mask is used to prevent the network from seeing input words from future time steps when making predictions.

We don’t need to give attn_mask here because the previous attention layer (self attention layer) in the decoder has already used it. Hence, the decoder states (query) computed by that previous attention has already avoided attending to future inputs.

S1655829 - s2015555

Question 7

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

        # Step1 Starts:
        # Linear projection of Query, Key and Value.

        k_ = self.k_proj(key)
        v_ = self.v_proj(value)
        q_ = self.q_proj(query)

        # Step1 Ends

        # Step2 Starts:
        # Computing scaled dot-product attention for h attention heads:

        # split the head and transpose d_k d_v,d_q dimensions:
        # [tgt_time_steps,batch_size,d_model] -> [num_heads,batch_size,tgt_time_steps,head_embed_size]
        k_ = k_.contiguous().view(-1, batch_size, self.num_heads, self.head_embed_size).transpose(0, 2)
        v_ = v_.contiguous().view(-1, batch_size, self.num_heads, self.head_embed_size).transpose(0, 2)
        q_ = q_.contiguous().view(-1, batch_size, self.num_heads, self.head_embed_size).transpose(0, 2)
        # stack head together for bmm multiplication for h attention head:
        # [num_heads,batch_size,tgt_time_steps,head_embed_size] -> [num_heads*batch_size,tgt_time_steps,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_ = q_.contiguous().view(self.num_heads* batch_size, -1, self.head_embed_size)
        # Scaled Dot-Product Attention: attn_weights = [num_heads*batch_size, tgt_time_steps, key.size(0)]
        attn_weights = torch.bmm(q_, k_.transpose(1, 2)) / self.head_scaling


        if key_padding_mask is not None:  # case if key_padding_mask is not None, avoid attending to padding elements
            # unsqueeze key_padding_mask and match to the head number :
            # [batch_size, tgt_time_steps]->[batch_size, 1, tgt_time_steps]->[num_heads*batch_size, 1, tgt_time_steps]
            key_padding_mask = key_padding_mask.unsqueeze(dim=1).repeat(self.num_heads,1,1)
            # mask the attention weights
            attn_weights.masked_fill(key_padding_mask, -1e9)
```

```
if attn_mask is not None:  # case if attn_mask is not None, avoid attending to the future words
    # mask the attention weights
    attn_weights += attn_mask.unsqueeze(dim=0)
# apply the softmax function
attn_weights = F.softmax(attn_weights, dim=-1)
# dropout layer after the softmax of QK^T product
attn_weights = F.dropout(attn_weights, p=self.attention_dropout, training=self.training)
# bmm between attention weights and value: attn = [num_heads*batch_size, tgt_time_steps, head_embed_size]
attn = torch.bmm(attn_weights, v_)

# Step2 Ends

# Step3 Starts:
# Concatenation of heads and output projection

# transpose attn and concatenation the heads: [num_heads*batch_size, tgt_time_steps, head_embed_size] ->
# [num_heads, batch_size, tgt_time_steps, head_embed_size] ->
# [tgt_time_steps, batch_size, num_heads, head_embed_size] ->
# [tgt_time_steps, batch_size, embed_dim]
attn = attn.contiguous().view(self.num_heads, batch_size, -1, self.head_embed_size).transpose(0, 2)
attn = attn.contiguous().view(-1, batch_size, self.num_heads * self.head_embed_size)
# do the final projection
attn = self.out_proj(attn)

# Step3 Ends

# transpose attn_weights from [num_heads*batch_size, tgt_time_steps, key.size(0)]
# to [num_heads, batch_size, tgt_time_steps, key.size(0)]
attn_weights = attn_weights.contiguous().view(self.num_heads, batch_size, tgt_time_steps, -1)
attn_weights = attn_weights if need_weights else None

# TODO: ---------------------------------------------------------------------- CUT

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

return attn, attn_weights
```

Question	Training Loss Last Epoch	Validation Perplexity Last Epoch	Test Set BLEU
Q1	2.126	27.3	10.78
Q5	1.811	24.3	13.68
Q7	1.476	38.9	10.59

The table above shows that the transformer performed worse on the validation set and test set, but performed better on the training set than LSTM-based models in Q1 and Q5. We argue that overall, the transformer performed worse on the test set because it has a significantly lower BLEU score than the model in Q5 and slightly lower than the BLEU score in the baseline. A possible explanation for this performance discrepancy between training, validation, and test time is that the model suffered from overfitting.

Two possible explanations for overfitting and fast convergence which were experienced by the transformer:

1. The transformer has many more parameters than the LSTM-based models, hence it has a larger flexibility to fit the underlying pattern in the data. However, this also means that the transformer is more prone to capturing noise in the training data, which can lead to overfitting. The table below shows that the total parameters in the transformer is almost twice as many as that of LSTM-based models.

Question	Total Params
Q1	1456644
Q5	1748040
Q7	2707652

2. The dataset size is too small. Our training set comprises only 10,000 sentence pairs. This size is very small compared to the dataset used in the original paper, i.e. 4.5 million sentence pairs[13]. For smaller datasets, transformer-based model are more prone to overfit compared to LSTM-based models[14] because generally, it has significantly more parameters.

As the transformer model is very likely to be overfitted suggested by the results, we could improve the model's performance by minimising the overfitting in two aspects:

- Dataset size: we could improve the transformer's performance by using a larger dataset (e,g, the standard WMT 2014 English-German dataset) and use BPE as in the paper. With the larger dataset, the model is less prone to overfit.

[13] "Attention Is All You Need." 12 Jun. 2017, https://arxiv.org/abs/1706.03762. Accessed 6 Mar. 2021.
[14] Ezen-Can, A., 2020. A Comparison of LSTM and BERT for Small Corpus. *arXiv preprint arXiv:2009.05451*.

- Regularisation methods: the paper used both residual dropout and label smoothing. However, solely dropout is used during the training. We could also add label smoothing during the training. Label smoothing prevents the model from becoming too confident for prediction during the training but generalising poorly.