NLU+ Coursework 2: Neural Machine Translation

UNN: S2268276, S2160729

Part 1 - Getting Started

Question 1 - Understanding the Baseline Model

Question 1.A.

Tensor shape annotation

final_hidden_states.size = [num_layers, batch_size, output_size]
BIDIRECTIONAL CASE: final_hidden_states.size = [num_layers, batch_size, 2*output_size]

final_cell_states.size = [num_layers, batch_size, output_size]
BIDIRECTIONAL CASE: final_cell_states.size = [num_layers, batch_size, 2*output_size]

What happens when self.bidirectional is set to True?

Two LSTMs are run, one that processes the sequence from left-to-right and one that processes the sequence from right-to-left. The final hidden layers of both LSTMs are then concatenated into a single vector.

What is the difference between final_hidden_states and final_cell_states?

final_cell_states and final_hidden_states have the same dimensions and the difference between them is that while the final hidden state flows to the next hidden layer or to the output, the final hidden cell flows into the next timestep representing the memory of the LSTM.

Question 1.B.

Tensor shape annotation

attn_weights: [batch_size, seq_len] → AFTER doing squeeze, i.e. removing dim-1 from the tensor
attn_out: [batch_size, output_size]
BIDIRECTIONAL CASE: attn_out: [batch_size, 2*output_size]

Describe how the attention context vector is calculated

The attention context vector is the result of the dot product between the attention weights and the encoder's output, i.e. the weighted average of the input sequence.

Why do we need to apply a mask to the attention scores?

We need to apply a mask to prevent the attention from cheating, i.e. we only attend to words in the sequence that are before "t", the tokens from "t+1" and forward are masked.

Question 1.C.

Tensor shape annotation

attn_scores: [batch_size, 1, seq_len]
projected_encoder_out: [batch_size, hidden_dim, seq_len]

How are attention scores calculated?

The attention scores are the result of a matrix multiplication between the input sequence and the encoder's output (transposed for the dimensions to properly match).

What role does batch matrix multiplication (i.e. torch.bmm()) play in aligning encoder and decoder representations?

Since we are doing tensor multiplication, torch.bmm() helps to align the encoder and encoder representations by allowing us to get a resultant tensor that IS NOT shaped by the hidden layer's dimension (e.g. 64 or 128). This batch matrix multiplication allows us to get a tensor whose first dimension is the batch size and the final dimension is the sequence length.

Question 1.D.

Tensor shape annotation

tgt_hidden_states: [batch_size, hidden_dim]
tgt_cell_states: [batch_size, hidden_dim]
input_feed: [batch_size, hidden_dim]

Describe how the decoder state is initialized.

The cell state and hidden state tensors are simply initialized to zeros, these tensors are initialized to have the proper dimensions from the start.

When is cached_state == None?

cached_state == None at inference time, i.e. when we predict the next word, we must recover the previous cell states and hidden states. During training, if they don't exist, the tensors are simply initialized to zeros, as described above.

What role does input_feed play?

The input feed is the tensor that contains the embeddings generated by the Encoder. i.e. the input to the Decoder which is necessary to predict the next token in the sequence.

Question 1.E.

Tensor shape annotation

input_feed: [batch_size, hidden_size]
BIDIRECTIONAL CASE → input_feed: [batch_size, 2*hidden_size]
rnn_outputs: not a tensor but a list of len() = sequence length
attn_weights: [batch_size, tgt_size, src_size]

How is attention integrated into the decoder?
For every timestep, a new attention "vector" is calculated and added to the attn_weights tensor, along the dim-1 axis. This is necessary because at every time step the last layer's state will change and the masking will also be adjusted according to the current time step. That's why the attn weights are calculated at every step of this for-loop.

Why is the attention function given the previous target state as one of its inputs?
The attention function is given the previous target state in order to "attend" to it as part of the context vector.

What is the purpose of the dropout layer?
The purpose of the dropout layer is to randomly replace some of the attention values with zeros. This introduction of stochastic noise should help the model to generalize better, since it will be less likely to overfit, as it will not "memorize" the tokens that frequently align with either low or high attention scores.

Question 1.F.

Tensor shape annotation
output: [batch_size, tgt_tokens, vocab_size], (vocab_size being the number of tokens (i.e. classes) in the target language)
loss: tensor of shape 1x1 (i.e. cross entropy loss)

Add line-by-line description about the following lines of code do.

```
# Get the target sequence batch predictions from the model:
output, _ = model(sample['src_tokens'], sample['src_lengths'],
sample['tgt_inputs'])

 # Reshape the output for it to be in the shape batch_size*tgt_input X
vocab_size, and the tgt_tokens
# are also passed to the calculation of the cross entropy loss. The loss is
normalized by the length
# of the sequence length.
loss = \
      criterion(output.view(-1, output.size(-1)),
      sample['tgt_tokens'].view(-1)) / len(sample['src_lengths'])

# Compute the gradient of the loss tensor with respect to the model
graph.loss.backward()

# Clip the gradient to ensure it is well behaved
 grad_norm = torch.nn.utils.clip_grad_norm_(model.parameters(),
args.clip_norm)

# Take a single step in the direction of the lowest descent at the moment:
optimizer.step()

# After taking a step of descent, reset all gradient values to zeros
optimizer.zero_grad()
```

Question 2 - Understanding the Data

Question 2.1.

As it can be seen in table 1, there are 124,031 tokens and 8,326 word types in the English data and 112,572 tokens and 12,504 word types in the German data.

Question 2.2.

In both languages, tokens that appear only once are replaced by <UNK>.
As reported in table 1, in English there are 3,909 unique tokens, a 3.15% of the total vocabulary and in German there are 7,460 unique tokens, a 6.63% of the vocabulary.
Therefore, since all these tokens are replaced by the same token, the total vocabulary size of English will be:

total vocabulary size = #tokens - #unique token + 1 = 8,326 - 3,909 + 1 = 5,417 tokens

where 1 is the <UNK> token.
The total vocabulary size of German can be computed in the same way, obtaining a value of 6,044 tokens.

	#tokens	# word types	# unique tokens (replaced)	% of replaced tokens	Total vocabulary size
English	124,031	8,326	3,909	3.15%	5,417*
German	112,572	12,504	7,460	6.63%	6,044*

Table 1: English-German training set descriptions

*These values include numbers, punctuation marks, etc.

Question 2.3.

By inspecting the tokens that are substituted by <UNK>, we can observe linguistic patterns between these words.

The main observation is that words with the same root (and, usually, very similar semantics) are tokenized as totally different words. This might cause variations of the same word to be detected as an unusual token and be substituted by an unknown token, losing relevant information about the meaning of the sentence. As we have seen in Table 2, the most simple example of this is a word and its plural form, but there's many other possible words to be created from the same root.

Word and its plural form	'phone', 'phones'
	'plane', 'planes'
Same root with different prefixes and suffixes	'mini', 'minimal', 'minimalist', 'minimising', 'minor'
	'negotiation', 'negotiators'
	'operated', 'operates', 'operatives', 'operator',
	'employ', 'employability', 'employee', 'employer', 'employers'

Table 2: Examples of unique words in English.

This problem could be solved by using WordPiece tokenization (Schuster and Nakajima (2012)). The advantage of this method lies in tokenizing between word-level and character level sequences. For instance, the word "boy" could be kept as is, but the word "boys" would be split into "boy" and "s". This allows the model to learn how words are formed, in this case, it would learn that "boy" and "boys" have slight differences in meaning but have the same root. A similar solution would be using byte-pair encoding, as seen in lectures (Sennrich, R., Haddow, B., & Birch, A. (2015))

Question 2.4.

In our corpus, there are 754 unique vocabulary tokens that are the same between both languages. This means that approximately 10% of the unique words in German don't need to be translated because they are the same in English.

If we print the set of this common tokens in both languages, we can see examples like the following:

People's name	'tamás', 'harrison', 'rivera', 'alvaro', 'mozart'
Numbers	'0469/2001', '0412/2001', '145'
Technical words	'gigabyte', 'nano'
Words that are neither English nor German	'garosci', 'vii', 'kuhne', 'fta', 'burundi'
Cities	'toledo','burgos'

Table 3: Examples of tokens that are the same between both languages.

In conclusion, there are several types of tokens that do not require translation. Detecting these cases instead of substituting them would significantly improve the quality of translations.

If we train separate monolingual models for English and German, we could attempt to measure the similarity of the word embeddings of identical words. If the similarity is above a certain threshold, we could simply reuse those embeddings in the translation task.

Question 2.5.

History shows that NMT systems will be affected by sentence length. Traditional systems such as Recurrent Neural Networks struggled to “remember” earlier sentence dependencies, and thus performance degraded as sentence length increases. Even LSTMs suffer from similar problems, although the introduction of Attention helps to alleviate the issue. The current tokenization method is simple, at the expense of losing important words that are rare, but usually carry rich semantics. The fact that there could be unknown words in both languages will also hamper the translation efforts. These model simplifications will nonetheless simplify our training efforts.

Part 2 - Exploring the Model

Question 3 - Improved Decoding

Question 3.1.

The main disadvantage of greedy decoding is that it usually does not find the globally optimal solution. This happens because the algorithm's choices are based only on past and present data, so it does not take into account data from the next steps.

This also implies that greedy decoding has no way to undo decisions. Once it selects a word, it can’t go back even if, in the future, another one would have been a better fit.

We can see an example of this below. When translating the determiner “ein”, the model can’t know if it should translate it by “a” or “an”, since this exclusively depends on the next word. Since greedy decoding only selects the most probable word at each step, it will select the option that has higher probability, i.e. more representation in the corpus: “a”; even if the correct answer would be “an”.

Example 263	
German	die annahme dieser allgemeinen leitlinien ist **ein** beispiel hierfür .
Reference	the adoption of these general guidelines is **an** example of this .
Baseline	the next item is **a** example of a great deal of example .

In conclusion, by only considering the token with maximum probability at each timestep, we are ignoring paths that might have been optimal due to dependencies with future words.

Question 3.2.

Pseudo code for beam search:

> **i** current prob, current state = decoder(previous word, previous state)
> # use the decode to generate the probability of all word at current step
>
> **ii** sort words by their probability and select the β most likely words
> # find out the β words with largest local probability, where β is the beam width
>
> Compute in parallel for each selected word and save the results:
> **iii** previous word = current word; previous state = current state. Go back to step **i** until we reach the end of sentence: <end>.
> # prepare for the next step decoding
>
> **iv** once we have reached the end of sentence for all different paths considered, compute the loss of each example and select the translation that minimizes the loss.

In the following image we can see a visual example of the beam search algorithm. The color's intensity encodes the probability of that word given the previous word and state. The two words with higher probability (β = 2) are considered in the next step. All others are discarded and no path from them is considered.

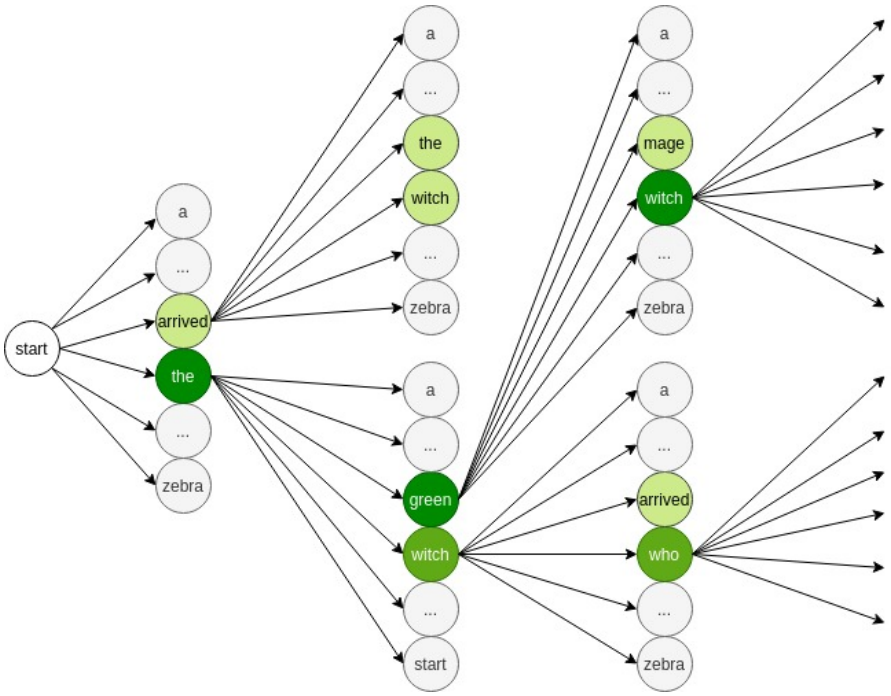

Figure 1: Beam search example. Courtesy of https://www.baeldung.com/cs/beam-search

Question 3.3.

It is well known that neural machine translation models based on encoder-decoder architectures suffer from a bias towards generating short sentences. To correct this bias, we need to find a way to encourage longer sentence generation.

How to deal with this problem is an active area of research and is discussed in several research papers. As mentioned in Murray, K., & Chiang, D. (2018), one possible solution is dividing translation scores by their length, i.e. length normalization. Another possible approach, mentioned in the same paper, is including a tunable reward added for each output word.

Question 4 - Adding Layers

Question 4.1.

Command used to run the model with added layers:

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

Question 4.2.

As we can see in Table 4, all model performance metrics get worse after increasing the number of encoder layers from 1 to 2, and the number of decoder layers from 1 to 3. The validation perplexity increases from 26.8 to 30.2, and the BLEU score decreases from the baseline's 11.03 to a value of 9.37.

	Training loss	Validation loss	Validation perplexity	BLUE
Baseline	2.145	3.29	26.8	11.03
Added layers	2.412	3.41	30.2	9.37

Table 4: Results of the model with added layers.

This result seemed surprising, since the model now has added capacity and should be learning better translations. However, we hypothesize that the problem is the training set size, since it is too small (only 10k sentences) to justify the added model capacity. A model with more layers (and same layer size) has more parameters will need more data to fit properly, and if the training set is too small it is likely to overfit. This does not seem to be the case as the training loss also increases (from 2.145 to 2.412) when layers are added to the encoder and decoder, thus there are no signs of overfitting. The fact that the training loss increased seems to suggest that we made the optimization problem harder for the model, and it is struggling to reach a local minima. Thus, it might be necessary to do additional hyperparameter tuning (e.g. adjust the learning rate) and/or obtain additional training data.

Part 3 - Lexical Attention

Question 5 - Implementing the Lexical Model

	Training loss	Validation loss	Validation perplexity	BLUE
Baseline	2.145	3.29	26.8	11.03
Lexical attention	1.836	3.18	24.1	12.68

Table 5: Results of the model with lexical attention.

The results in table 5 show considerable improvements in performance by introducing lexical attention in the decoder (compared to the baseline). The training loss is reduced from 2.145 to 1.836, the validation loss decreases from 3.29 to 3.18. Most importantly, the validation perplexity decreases from 26.8 to 24.1 and the BLEU score increases by almost 2 points (from 11.03 in the baseline to 12.68 in the lexical model).

As explained by Nguyen and Chiang (2017) in their paper, a baseline NMT struggles with the learning of rare words. One of the flaws of baseline NMTs is that they have a tendency towards generating words that happen very frequently in the context, at the expense of not reflecting what really is in the source sentence. The addition of lexical attention in the decoder adds a better connection to the source sentence and should therefore enhance the learning of translation of rare words.

We found some interesting translation examples (see below).

Translation examples:

For sentence 141 in the test set, only the lexical attention model was able to translate the word “belgien” to “belgium”. Unfortunately, it was not able to keep the proper noun “lange”, instead translating it to “long”. Although both the baseline and “added layers” model incur in the same translation mistake: replacing a rare name by a more common one in the training set.

Example 141	
Reference	i experienced this in belgium for nine years , mr lange .
Baseline	this is a number of concern , mr president , mr cohn , many last years .
Added layers	this house has been made in many years , mr barroso .
Lexical	this is why i voted in belgium , mr president , mr long .

In example 209, we see how no model is able to get the proper translation of “tobacco growers”. Curiously, all translations are quite verbose compared to the reference sentence. We see how the

model with lexical attention is able to get the best performance (i.e. closer meaning to original sentence) compared to the other two models.

Example 209	
Reference	let us support the tobacco growers ' efforts !
Baseline	we should therefore support the facts of the internal market !
Added layers	we should therefore forget the internal market of the internal market !
Lexical	we should therefore support the efforts of the european people .

In example 398, we finally see the value of having the lexical attention in the NMT's decoder. Only the lexical attention model is capable of translating the rare sequence "article 47".

Example 398	
Reference	the next step will really be how the council will deal with the rewording of article 47 .
Baseline	as the council has tabled on the green paper on the police party .
Added layers	as the council has been spent the council on the number of the proposed authorities .
Lexical	as a whole , the council has been said on the rule of article 47 .

Part 4 - Transformers

Question 6 - Understanding the Transformer Model

Question 6.A.

Tensor shape annotation
embeddings: [batch_size, src_lengths, embed_dim]

What is the purpose of the positional embeddings in the encoder and decoder?
By definition, the Transformer is permutation invariant, i.e. if we shuffle up words, the translation would be the same. However, we know that word order does matter in linguistics, thus we use positional embeddings for the model to be somewhat sensitive to word order.

Why can't we use only the embeddings similar to for the LSTM?
The LSTM processes the input follwing its natural order. Therefore, LSTM is naturally sensitive to word order. However, this does not hold in the transformer model, so positional embeddings are required, to feed the model with this information.

Question 6.B.

Tensor shape annotation
self_attn_mask: [src_lengths, src_lengths]

What is the purpose of self_attn_mask?
Transformers use self-attention to understand how previous words are related to the current word in a sentence. Masking the input keeps the model from "cheating", i.e. attending to future words would be cheating. After masking is applied, we guarantee that the model can only attend to the previous words in the sequence.

Why do we need it in the decoder but not in the encoder?
Because in the encoder we are mapping the entire word sequence to a latent vector. It is this latent vector that will be passed on as the decoder's input. We want this vector to be a rich representation of the sentence's meaning. Masking is not necessary since the vector has to represent the whole sentence, and it would actually harm model performance.

Question 6.C.

Tensor shape annotation
The forward_state before the linear projection has the following dimensions: forward_state: [batch_size, tgt_inputs, embed_dim].

Why do we need a linear projection after the decoder layers?
After the decoder layers, we need a linear projection to transform the output from embeddings to target language. The final output has to be words in the target language.

What is the dimensionality of forward_state after this line?
After the linear projection the dimensions of the output are the following: forward_state: [batch_size, tgt_inputs, vocab_size] where vocab_size is the size of the target language vocabulary, English in this case.

What would the output represent if features_only=True?
If features_only is True, the output will not be projected into the target language space, and will be returned in the embedding representation.

Question 6.D.

Tensor shape annotation
state (before self-attention): [src_lengths, batch_size, embed_dim]
query : [src_lengths, batch_size, embed_dim]
key: [src_lengths, batch_size, embed_dim]
value: [src_lengths, batch_size, embed_dim]

What is the purpose of encoder_padding_mask?
The encoder_padding_mask is used to fill sequence spaces that are shorter than the max. sequence length in a batch. I.e. the shorter sentences' ends' are padded with zeros. This length is always specific to the batch.

What will the output shape of `state' Tensor be after multi-head attention?
The output shape of state after multi-head attention is the same state: [src_lengths, batch_size, embed_dim]

Question 6.E

Tensor shape annotation
state: [tgt_length, batch_size, embed_dim]
attn_shape: [num_heads*batch_size, tgt_length, embed_dim/num_heads]
key (encoder_out): [src_length, batch_size, embed_dim]

How does encoder attention differ from self attention?
While self attention models how the input tokens interact with each other, i.e how a sequence interacts with itself, the encoder attention is the comparison of the target sequence to the input sequence, and therefore is completely different from self-attention.

What is the difference between key_padding_mask and attn_mask?
The key_padding_mask is used to fill sequence spaces that are shorter than the max. sequence length in a batch. I.e. the shorter sentences ends are padded with zeros. This length is always specific to the batch. The attn_mask, as described before, is used to prevent the decoder from attending "future" tokens at inference time. In other words, to prevent

If you understand this difference, then why don't we need to give attn_mask here?
We don't need attn_mask in the encoder, only in the decoder.

Question 7 - Implementing Multi-Head Attention

The transformer model was trained using 8 (narrow) attention heads and an attention dropout value of 0.4. The results can be seen below in Table x:

	Training loss	Validation loss	Validation perplexity	BLUE
Baseline	2.145	3.29	26.8	11.03
Transformer (last epoch)	1.468	3.79	44.4	-
Transformer (best val. loss)	2.623	3.48	32.5	10.17

Table 6: Results of the model with lexical attention.

As we can see, the transformer has a worse performance than the baseline model. The model converges quickly, since it only runs for 20 epochs. In fact, it stops improving after epoch #9 (as measured by validation loss). At the final epoch, the validation perplexity is 44.4, considerably worse than all previous models (baseline, added layers, and lexical attention). At its best performance in epoch 9, the perplexity is much lower: 32.5, still worse than all models. The test-set BLEU score is 10.17, which is worse than all the other models, except the LSTM with added layers (which has a test-set BLEU score of 9.37). There are clear signs of overfitting, since there is a large difference in training loss (1.468) and the validation loss (3.79) at the last epoch.

Based on these results, we think the model is performing poorly due to:

1. **Insufficient training data**: this model has almost 3 million parameters. As discussed earlier, as a deep learning model grows in size, it becomes more "data hungry" and needs additional data in order to generalize better and avoid overfitting. A state-of-the-art transformer model, like BERT (Devlin et al. (2018)), was trained with more than 3.3 billion words to achieve "human level performance"[1]. Our transformer was trained on approximately 100k words, which is 0.003% of the BERT's original training set.

   In fact, Ezen-Can, A. (2020) shows that bidirectional LSTMs do achieve higher performance than pre-trained BERT when evaluated on small datasets.

2. **Simple model design**: we trained a relatively simple transformer model, composed of a single block with 2 encoder layers and 2 decoder layers. On the other hand, the original transformer model as defined by Vaswani et al. (2017) has 6 encoder and decoder layers. In fact, BERT has 12 stacks of transformer blocks, which adds up 100 million parameters.

[1] When evaluated in automated tasks, e.g. BLEU scores.

In conclusion, the combination of insufficient training data and relatively simple model complexity (i.e. lack of depth) result in poor model performance. Consequently, model performance can be improved by feeding the model with a larger training set and increasing the model's complexity/size.

Bibliography

Murray, K., & Chiang, D. (2018). Correcting length bias in neural machine translation. *arXiv preprint arXiv:1808.10006*.

Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Ezen-Can, A. (2020). A Comparison of LSTM and BERT for Small Corpus. *arXiv preprint arXiv:2009.05451*.

Sennrich, R., Haddow, B., & Birch, A. (2015). Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*.

Schuster, M., & Nakajima, K. (2012, March). Japanese and korean voice search. In *2012 IEEE international conference on acoustics, speech and signal processing (ICASSP)* (pp. 5149-5152). IEEE.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, *30*.