Natural Language Understanding, Generation, and Machine Translation (2020–2021)
Coursework 2: Neural Machine Translation

s2139538
s1412487

February 2021

Part 1

Question 1: Understanding the Baseline Model

A. At time step t, the final hidden states form a representation, and the final cell states control what the hidden states should remember or forget. If `self.bidirectional` is set to `True`, each input is processed in two directions (left-to-right and right-to-left), and the two directions of the final states are concatenated to form a single representation of the input sequence for each time step.
B. The attention context vector is calculated by applying the softmax function on the attention scores and using the resulting weights to average over the source hidden states. We need a mask to pad shorter sentences with `-inf` so that their weights are mapped to `0` after applying the softmax function.
C. Attention scores are calculated according to the general scoring function from Luong et al. (2015). The matrix multiplication computes the dot products between the target input and the linearly projected outputs of the encoder in batches, i.e.

`[batch, 1, tgt_dims]` $\times$ `[batch, tgt_dims, src_length]` $\Rightarrow$ `[batch, 1, src_length]`

D. If we do not use incremental decoding, `cached_state` is set to `None` and the decoder states are initialized as zero (otherwise it is initialized as the previous cached state). Similar to Luong et al. (2015), input feed is the output from previous time step after global attention, which informs the model about past alignment decisions.
E. Attention is integrated into the decoder by projecting the "previous" hidden state concatenated with the context vector. The attention function takes "previous" decoder hidden state ("previous" refers to the last layer of stacked LSTMs, not the previous time step) as input in order to compare it to the encoder output layer. The dropout layer prevents overfitting by randomly setting units to zero.
F. In the forward step, the activations of all the different layers in the system are computed batch-wise. Then, the loss is computed and back-propagated and gradients are clipped. The weights are updated and the gradients are reset to zero.

Question 2: Understanding the Data

1. We counted the number of tokens excluding the end of sentence tokens.

Data set	Total tokens	Total types
train.en	124031	8326
train.de	112572	12504

2.

Data set	Tokens replaced	Vocabulary size
train.en	3909	4417
train.de	7460	5044

3. A common and important type of replaced word are morphological variants of a lemma (or lemmata) that is otherwise present in the text. An English example would be 'unimportant', whose lemma 'important' is found 223 times in the text. A German example is 'weiterbildung' (further education), which is a compund of the words 'weiter' (further) and 'bildung' (education), which respectively show up 45 and 8 times in the text. Treating morphological variants as out-of-vocabulary words is likely to have a negative effect on test set performance, as these words often carry a lot of self-information.
4. The number of word types shared by the languages is 1460, the majority of which (982) only appear once in at least one of the data sets. Common word classes include proper nouns, punctuation marks, numerals and loanwords, and such words mostly have the same meaning in both languages. We can use this intersection to create a dictionary by aligning identical words to each other. In a post-processing step, the words translated as `<UNK>` could then be translated by looking them up in the dictionary. This has been investigated by, for example, Luong et al. (2014).
5. For the post-processing step described in 4., we need to figure out to which source words the `UNK` tokens are connected. If the target sentence is shorter than the source sentence, then there are more candidate source words, and the task becomes harder. If the target sentence is longer than the source sentence, then the task is easier. Furthermore, a higher type-token ratio will cause more out-of-vocabulary words, since we will have to remove these words from our dictionary in order to keep the dictionary of a reasonable size. Therefore, there will be less exposure to each word type, which would lead to less accurate translations. Finally, an effective handling of out-of-vocabulary words is very important for model performance. Although unknown word tokens typically only constitute a small part of the total amount of tokens, they usually carry a lot of self-information.

Part 2

Question 3: Improved Decoding

1. A problem with greedy decoding is that it has no way to undo its decisions. As we can see in the sentence below, this might lead the decoder to be 'caught' in the wrong path. After correctly producing 'This', the model produces 'is', arguably because 'This is' is more common in the corpus than 'This must'. 'This is' is often followed by an adjective, and the model subsequently predicts the adjective 'okay'. However, this prevents the model from correclty predicting the noun phrase 'highest priority'.

Source	*Das* muss höchste priorität haben.
Target	*This* must have the highest priority.
Translation	*This is* okay.

2. We adapt the notation from Luong et al. (2015). Given input sentence $\mathbf{x}$, the log probability of a translated sequence of tokens up to time step t can be computed by the sum of conditional log probabilities up to that time step.

$$\log P(y_1 \dots y_t|x) = \sum_{t'=1}^{t} \log P(y_{t'}|y_1, \dots, y_{t'-1}, \mathbf{x}) \tag{1}$$

and the probability of the whole sentence of length T is given by $P(y|x) = P(y_1 \dots y_T|x)$. Each of the conditional probabilities in the product can be approximated as a function of the previous decoder hidden state h_{t-1}, the previous decoder prediction y_{t-1} and the encoder output $\mathbf{s}$:

$$P(y_t|y_1, \dots, y_{t-1}, x) \approx f_{y_t}(h_{t-1}, y_{t-1}, \mathbf{s}) \tag{2}$$

where f_{y_t} represents the probability of the token y_t. This value is given by the probability assigned to y_t in the probability distribution that results after the softmax layer in the RNN.

Equation 2 allows us to solve Equation 1 by computing the probability of a sequence of tokens recursively. For each time step, the probability of the sequence can be found from the previous time step by:

$$\log P(y_1 \dots y_t|x) \approx \log P(y_1 \dots y_{t-1}|x) + \log f(h_{t-1}, y_{t-1}, \mathbf{s}) \tag{3}$$

For beam search with beam width k, we store the sequences with the k highest probabilities for each time step t. Since the probability of a token y_t is dependent on the previous token y_{t-1} (Eq. 2), we compute the probabilities at time step t conditioned on the k different tokens at time step $t-1$.

We would use a table of dimensions $(k \times T_{max})$, where T_{max} is the manually set maximum sentence length. In each cell, we would store the tuple of values $(P(y_1 \dots y_t|x), y_t, y_{t-1})$, where y_t is the current token and y_{t-1} is the previous token. Including both the current token and its predecessor in the table allows us to reconstruct the sentence. The probability of the whole sentence is then given by the probability at time step T, where T is the time step when the end-of-sentence token is produced.
3. The beam search algorithm calculates the probability of a sentence as the product of the conditional probabilities of the words. Therefore, longer sentences will include more terms in the product. Since these terms are less then one, the probability of a long sentence is small, and thus, the decoder is likely to favor shorter sentences. Length normalization can help solve this, but can lead to bad translations when the target sentence is much shorter (or longer) than the source sequence.

Question 4: Adding Layers

1. The command used to train the model with a different number of encoder and decoder layers:

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

2.

Measure	Baseline model	Deep model
Training Loss	2.15	2.44
Validation Loss	3.28	3.39
Validation perplexity	26.7	29.5
BLEU	11.11	9.84

Table 1: Results of the baseline model and the deep model with two encoder and three decoder layers. We used early stopping, aborting training 10 epochs after no improvement in validation loss could be observed. All values are taken from the best epoch, i.e. where validation loss was lowest.

As shown in Table 1, the deep model performs worse than the baseline model on all sets, exhibiting a higher training loss and validation perplexity. This may be explained by the fact that although more LSTM layers allow for a greater model complexity, it also results in new problems when computing the gradients. Vanishing and exploding gradient problems may become common (although the latter is prevented by gradient clipping), and it becomes difficult for the network to learn useful features. Ultimately, the best setting depends on the nature and amount of data, and a deeper network generally works better for a larger amount of data (our training data consists of 10,000 sentences, which is relatively small).

The model also performs considerably worse on the provided test set, as seen in the BLEU score. The reasons for this are similar to those above, although the measure now consists of n-grams instead of output probability. Arguably, this is because the deep model has a less direct access to the source words, and may therefore be worse at handling rare words.

The deep model seems to overfit less than the baseline model, as shown by their respective training-to-validation loss ratios 0.72 and 0.66. The features learned by the deep model thus arguably are more general than those learned by the baseline model. This can be explained by the fact that the deeper network learns richer representations of the data.

Part 3

Question 5: Implementing the Lexical Model

The lexical module proved to be beneficial for performance, resulting in a reduction in both training and validation loss compared to the baseline model. Since the lexical model feeds word representations that are not conditioned on forward or backward histories, the model seems to be better at predicting single words.

The most substantial increase is in BLEU score by more than 1.5 points. The effect is most evident for low-frequency words, many of which shared between the languages' vocabularies. This is illustrated in Table 3.

Measure	Baseline Model	Lexical Model
Training Loss	2.15	1.97
Validation Loss	3.28	3.17
Validation Perplexity	26.7	23.8
BLEU Score	11.11	12.77

Table 2: Results of the baseline model and the lexical model inspired by Nguyen and Chiang (2017). We used early stopping, aborting training 10 epochs after no improvement in validation loss could be observed. All values are taken from the best epoch, i.e. where validation loss was lowest.

Reference:	(it) mr president , *ladies and gentlemen* , he was the right man at the right time .
Lexical:	(the president , *ladies and gentlemen* , he was a right time to the right of the right time .
Baseline:	(the president cut the floor to mr president , and it was a matter of time .
Reference:	the situation of the *palestinians* has always been the core issue of the *conflict* .
Lexical:	the situation of the *palestinian* has already been taken in the key of the *conflict* .
Baseline:	the situation is only more than the legal basis of the treaty .
Reference:	this would then safeguard the substance of natura *2000* .
Lexical:	then , too , the material process is also established about *2000* .
Baseline:	then , the tunisian group are also active by mid - quadras .
Reference:	because , today , *60 % of europeans* live in the urban community .
Lexical:	because , *60 % of europeans* was living in the europeans of the europeans in burma .
Baseline:	the same thing is the inflation of goods in the montreal in the world .

Table 3: Sample output translations from the lexical and baseline models. The lexical module improves the handling of infrequent words, written in italics. We used early stopping, aborting training 10 epochs after no improvement in validation loss could be observed. All values are taken from the best epoch, i.e. where validation loss was lowest.

Part 4

Question 6: Understanding the Transformer Model

A. Without position embeddings, the transformer computes a function that is invariant with respect to the input sequence. Position embeddings introduce a dependence on the order in which words are introduced. LSTMs, on the other hand, are intrinsically dependents on the word order, since they read the sentence from one direction to the other.
B. The `self_attn_mask` ensures that the model can not attend to subsequent target word tokens. Such that it only depends on the encoder output and previous target words. In the encoder, we do not apply such a mask since we want words to attend to the whole sentence. In incremental decoding, we do not need a mask as the decoder is only fed the previous output words, and thus has no access to subsequent words.
C. This linear projection is needed to convert the decoder output to a distribution over the target vocabulary in order to output word tokens. The dimensionality of is thus `[batch_size, tgt_time_steps, tgt_vocab_size]`. If `features_only=True`, the output would simply the output from the decoder, i.e. a mini-batch of contextualized word embeddings.
D. The purpose of `encoder_padding_mask` is to account for the fact that the source sentences in the batch are of different length. The output shape of `state` tensor will be `[src_time_steps, batch_size, embed_dim]`.
E. Encoder attention differs from self-attention in that it attends to the output embeddings of the encoder instead of the embeddings in the decoder. `key_padding_mask` is used to adjust the length of the sentences, whereas `attn_mask` prevents the decoder from attending to future positions. We do not use `attn_mask` while attending to the decoder since we want all the embeddings in the decoder to have access to all the encoder output embeddings.

Question 7: Implementing Multi-Head Attention

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

"""
We use the following notation:

time steps (as in tgt_time_steps)     length per head (as in head_embed_size)

query    query_time_steps                    L_query
key      key_time_steps                      L_query
value    key_time_steps                      L_value

The key and value have the same number of time steps.
The query and key have the same length per head. (However a distinction between query, key and value
is insignificant in the case of length per head as it remains unchanged.)
"""
query_time_steps = tgt_time_steps #Renaming solely for purpose of clarity
key_time_steps = key.size(0)      #This is equal to value_time_steps

#Project
query = self.q_proj(query)
key = self.k_proj(key)
value = self.v_proj(value)

#Split the last dimension into a number of heads
query = query.view(tgt_time_steps, batch_size, self.num_heads, self.k_embed_size//self.num_heads)
key = key.view(key_time_steps, batch_size, self.num_heads, self.k_embed_size//self.num_heads)
value = value.view(key_time_steps, batch_size, self.num_heads, self.v_embed_size//self.num_heads)

#Prepare for summation by transposing axes
query = query.transpose(2,0)             #[num_heads, batch_size, query_time_steps, L_query]
key = key.transpose(2,0).transpose(3,2)  #[num_heads, batch_size, L_query, key_time_steps]
value = value.transpose(2,0)             #[num_heads, batch_size, key_time_steps, L_value]

#The attention scores are found by summation, scaling and masking
attn_scores = torch.matmul(query, key)   #[num_heads, batch_size, query_time_steps, key_time_steps]
attn_scores = attn_scores/self.head_scaling

#Masking
#attn_mask has dims [query_time_steps, key_time_steps]
#This broadcasts nicely to attn_weights
#attn_mask has values of 0 or -inf, so we simply add it
if attn_mask is not None:
    attn_scores = attn_scores + attn_mask

#key_padding_mask has dims [batch_size, key_time_steps]
#This is modified to [1, batch_size, 1, key_time_steps] for broadcasting to work
#key_padding_mask has Boolean values, so we use the function masked_fill_
if key_padding_mask is not None:
    key_padding_mask = key_padding_mask.unsqueeze(0).unsqueeze(2)
    attn_scores = attn_scores.masked_fill_(key_padding_mask, float('-inf'))

attn_weights = F.softmax(attn_scores, dim=3)  #Softmax over key_time_steps

#Prepare for summation by transposing axes
attn_weights_ = attn_weights.unsqueeze(3) #[num_heads, batch_size, query_time_steps, 1, key_time_steps]
value = value.unsqueeze(2)                #[num_heads, batch_size, 1, key_time_steps, L_value]

#Attention as in Vaswani et. al. (2017) is found by a summation of scaled terms
attention = torch.matmul(attn_weights_, value)  #[num_heads, batch_size, query_time_steps, 1, L_value]
attention = attention.squeeze(3)                #[num_heads, batch_size, query_time_steps, L_value]
```

```
#Concatenate the different heads together
concatenation = torch.cat(([attention[i] for i in range(self.num_heads)]), dim=2)
concatenation = concatenation.transpose(0,1)    #[query_time_steps, batch_size, embed_dim]

#Final projection
attn = self.out_proj(concatenation)
# TODO: ---------------------------------------------------------------------- CUT
```

Our code in the forward function from the `MultiHeadAttention` class is shown above. The performance of the model is shown in Table 2 below, and the evolution of training and validation loss is illustrated in Figure 1.

The transformer model performs worse than the baseline LSTM on all metrics. The reason for this can be seen in Figure 1 – the transformer improves performance on the training set very quickly, but this performance is not reflected on the validation set. The transformer is thus overfitting to the training set, and the learned features do not generalize to the unseen data in the validation set. This could be explained by two factors. First, the training data set is small compared to the complex nature of the data. A larger training set would therefore be helpful to reduce the generalization error. Second, overfitting could be reduced by applying regularization techniques. In the current model, this is already implemented by the use of dropout, but other techniques such as L2 regularization or Gaussian noise could be applied as well. To test this hypothesis, we simply doubled the dropout parameters given by the command line arguments `-dropout`, `-attention-dropout` and `-activation-dropout` to 0.2, 0.4 and 0.2 respectively. This improved the BLEU test score to 11.40, beating the baseline.

Furthermore, the transformer converges very fast, reaching its best performance after only 7 epochs (Figure 1). One explanation for this is that an LSTM needs to iterate over all intermediate tokens to find an alignment between two words and backpropagate the error, which can be difficult for larger distances. The transformer, however can learn such an alignment in a direct computation.

It stands to mention that the transformer performs well on the BLEU test score compared to the other metrics. Compared with the deep LSTM, for example, it shows a worse training loss and validation perplexity, but a BLEU score that is better by more than one point. A possible reason for this is that the transformer is better at handling rare words, while the LSTM seems to prefer to choose the most common words.

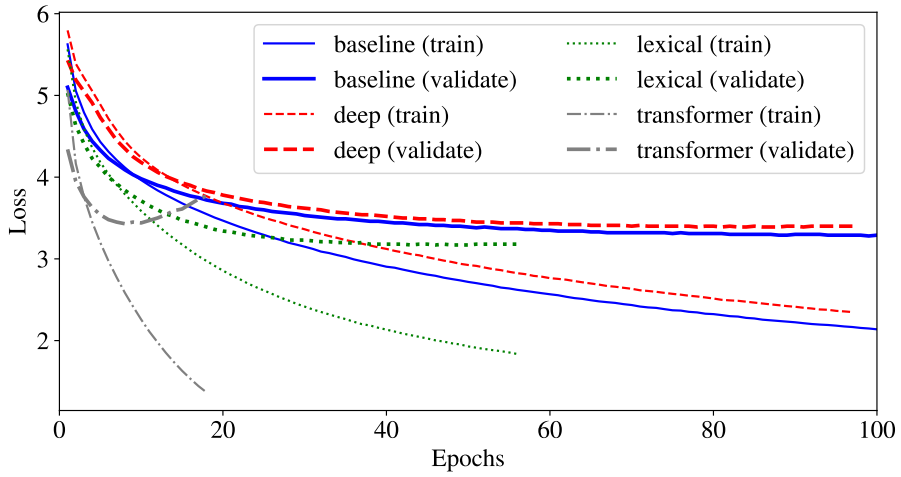

Figure 1: Evolution of training and validation loss for the implemented models.

Measure	Baseline LSTM	Lexical LSTM	Deep LSTM	Transformer
Training Loss	2.15	1.97	2.44	2.59
Validation Loss	3.28	3.17	3.39	3.43
Validation perplexity	26.7	23.8	29.5	30.8
BLEU	11.11	12.77	9.84	10.95

Table 4: Results of the transformer model compared to the other implemented LSTM models. We used early stopping, aborting training 10 epochs after no improvement in validation loss could be observed. All values are taken from the best epoch, i.e. where validation loss was lowest.

Bibliography

Luong, Minh-Thang, Hieu Pham, and Christopher D. Manning (2015). "Effective Approaches to Attention-based Neural Machine Translation". In: *CoRR* abs/1508.04025. arXiv: 1508.04025. URL: http://arxiv.org/abs/1508.04025.

Luong, Minh-Thang, Ilya Sutskever, Quoc V Le, Oriol Vinyals, and Wojciech Zaremba (2014). "Addressing the rare word problem in neural machine translation". In: *arXiv preprint arXiv:1410.8206*.

Nguyen, Toan Q and David Chiang (2017). "Improving lexical choice in neural machine translation". In: *arXiv preprint arXiv:1710.01329*.