NLU Assignment 2 2024

1 Understanding the Baseline Model

Following code blocks are the solutions to questions mentioned.

1 A

```
'''___QUESTION-1-DESCRIBE-A-START___
1.  Add tensor shape annotation to each of the output tensor
    output tensor means tensors on the left hand side of "="
    e.g.,
        sent_tensor = create_sentence_tensor(...)
        # sent_tensor.size = [batch, sent_len, hidden]
2.  Why do we need to apply a mask to the attention scores?
    - masking is using to control which parts of the input are used to calculate the
    ↪  attention weights.
    - the mask is used to prevent the model from attending to the padded elements in the
    ↪  input sequence.
    - further prevents model from not considering future tokens, ensuring that the model
    ↪  only attends
        to the tokens that have been processed so far.
'''
if src_mask is not None:
    src_mask = src_mask.unsqueeze(dim=1)
    attn_scores.masked_fill_(src_mask, float('-inf'))

# attn_weights has shape = [batch_size,1, src_time_steps]
attn_weights = F.softmax(attn_scores, dim=-1)
#attn_context has shape = [batch_size, output_dims]
attn_context = torch.bmm(attn_weights, encoder_out).squeeze(dim=1)
#context_plus_hidden has shape = [batch_size, input_dims + output_dims]
context_plus_hidden = torch.cat([tgt_input, attn_context], dim=1)
#attn_out has shape = [batch_size, output_dims]
attn_out = torch.tanh(self.context_plus_hidden_projection(context_plus_hidden))
'''___QUESTION-1-DESCRIBE-A-END___'''
```

1 B

```
'''___QUESTION-1-DESCRIBE-B-START___
1.  Add tensor shape annotation to each of the output tensor
2.  How are attention scores calculated?
The attention scores are calculated as the dot product matrix multiplication between the
↪  tgt_input and the projected_encoder_out.
it measures the similarity between the tgt_input and the encoder_out.
'''
projected_encoder_out = self.src_projection(encoder_out).transpose(2, 1)
# projected_encoder_out has shape = [batch_size, output_dims, src_time_steps]
attn_scores = torch.bmm(tgt_input.unsqueeze(dim=1), projected_encoder_out)
# attn_scores has shape = [batch_size, 1, src_time_steps]
'''___QUESTION-1-DESCRIBE-B-END___'''
```

1 C

```
'''___QUESTION-1-DESCRIBE-C-START___
1.  When is cached_state == None?
    -   When model ran for the first time
    -   when the model is not in incremental generation state.
2.  What role does input_feed play?
    -   It provide the previous output of the decoder as input to the current time step.
    -   If cached_state is not None, the input_feed is retrieved from the cached state.
'''
cached_state = utils.get_incremental_state(self, incremental_state, 'cached_state')
if cached_state is not None:
    tgt_hidden_states, tgt_cell_states, input_feed = cached_state
else:
    tgt_hidden_states = [torch.zeros(tgt_inputs.size()[0], self.hidden_size) for i in
    ↪  range(len(self.layers))]
    tgt_cell_states = [torch.zeros(tgt_inputs.size()[0], self.hidden_size) for i in
    ↪  range(len(self.layers))]
    input_feed = tgt_embeddings.data.new(batch_size, self.hidden_size).zero_()
'''___QUESTION-1-DESCRIBE-C-END___'''
```

1 D

```
'''___QUESTION-1-DESCRIBE-D-START___
1.  Why is the attention function given the previous target state as one of its inputs?
    -   It enables the attention mechanism to focus on different parts of the source
    ↪  sentence at different time steps.
    -   Enables decoder to be informed about the previous state of decoder.
2.  What is the purpose of the dropout layer?
    -   To prevent overfitting and to improve generalization. regularizing the attention
    ↪  mechanism.'''

'''___QUESTION-1-DESCRIBE-D-END___'''
```

1 E

```
'''___QUESTION-1-DESCRIBE-E-START___
1.  Add tensor shape annotation to each of the output tensor
2.  Add line-by-line description about the following lines of code do.
'''
# the source tokens, source lengths and target inputs are passed to model.
# forward propagation is done and the output is obtained.
output, _ = model(sample['src_tokens'], sample['src_lengths'], sample['tgt_inputs'])
# output: [ batch_size, max_tgt_len, tgt_vocab_size]

# cross entropy loss is calculated between the output and the target tokens by flattening
↪  the tensor and then dividing by src length.
loss = \
    criterion(output.view(-1, output.size(-1)), sample['tgt_tokens'].view(-1)) /
    ↪  len(sample['src_lengths'])
# back propagation is performed and the gradients are calculated.
loss.backward()
# gradient clipping is done to avoid vanishing and exploding gradients.
grad_norm = torch.nn.utils.clip_grad_norm_(model.parameters(), args.clip_norm)
# the optimizer is updated with the gradients.
optimizer.step()
# the gradients are resetted to zero value.
optimizer.zero_grad()
'''___QUESTION-1-DESCRIBE-E-END___'''
```

2 Understanding the Data

1, 2. Word tokens and tokens replaced by $< unk >$

The tokenization process taken into consideration is simply splitting the sentence using spaces. The results in Table Table 1 are calculated using this tokenisation method.

	English Data	**German Data**
Total word tokens	124031	112572
Number of word types	8326	12504
Number of words tokens replaced by $< unk >$	3909	7460
Before replacement	8326	12504
After replacement	4418	5045

Table 1: Examination of parallel training data

3. Inspect words replaced by $< unk >$

Upon inspecting the words replaced by the special token $< unk >$, a noteworthy linguistic phenomenon emerges. The replaced words exhibit a diverse range, including nouns (both common and proper), technical terms, foreign language words, and compound words – a hallmark of the German language. These include variations of the same lemma, such as "*aborted*," "*abortion*," and "*abortions*," which, despite their shared root, are treated as distinct tokens. Additionally, there's a noticeable presence of nouns and proper nouns, including personal names like "*atatürk*" and place names like "*areitio*," as well as rare or technical terms like "*cautionary*" and "*codification*."

Another linguistic characteristic is the occurrence of non-alphanumeric characters or unique elements from foreign alphabets, as seen in "*apolinário*" and "*bogusław*," and words with slight spelling variations like "*cancun*" and "*cancún*." This extends to named entities that undergo transliteration, such as "*Azerbaijan*" becoming "*Aserbaidschan*" in German.

A significant aspect of German, in particular, is its propensity for compound words, which can lead to lengthy tokens. Examples include "*entwicklungsausschusses*" and "*entwicklungsbudgets*," where the prefix "*entwicklung-*" pertains to development, suggesting a shared semantic field. Traditional tokenization methods may inadequately represent such nuances, often relegating these informative tokens to the unknown category, thus omitting valuable information.

German is a morphologically rich language. Inflectional morphology like *program, programme* and *verarbeiten, verarbeitet* have similar lemma. likewise the lemma *entscheiden* has various derivational morphology such as *entscheidet, entscheidend, entscheidender, entscheidende*. Such morphological words such as *Entscheidungsprozesse* or *verarbeitung* is being declared $< unk >$ resulting in lost of contextual information due to poor tokenization.

Given these observations, the tokenization process could be enhanced by adopting a more granular approach that recognizes and preserves linguistic subtleties. It can be improved by employing a subword tokenization technique, such as Byte-Pair Encoding (BPE) or WordPiece[1]. These methods segment rare and compound words into meaningful subword units, enabling the model to learn transparent translations and generalize this knowledge to unseen words. By capturing the morphological and semantic information encoded within these subword units, the model can better represent and translate rare and compound words, thereby mitigating the loss of crucial information that occurs when treating them as unknown tokens.

Furthermore, incorporating techniques such as lemmatization or stemming could help address the issue of different inflected forms of the same lemma being treated as distinct tokens. This would allow the model to recognize and leverage the underlying semantic relationships between related words, enhancing its ability to generalize and translate more accurately. Therefore, refining the tokenization process to accommodate these linguistic patterns could enhance the overall effectiveness of language models trained on this dataset.

[1] as suggested by Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016 in Neural Machine Translation of Rare Words with Subword Units

4. Exploiting same tokens between the two languages

There are a total of 1460 tokens in common, if all the single occurring words are replaced by $< unk >$ then there are 479 (478 tokens + $< unk >$) tokens in common, including the numerical values in both languages. A potential benefit would be to utilize a similar embedding matrix for such a situation. The model can leverage the inherent semantic and syntactic relationships that exist between the two languages. It reduces the overall number of parameters in the model, thereby reducing computational complexity and memory requirements. It facilitates better cross-lingual knowledge transfer and improves the model's generalization ability. In addition, utilising various other tokenization methods will further improve the model's ability. A shared embedding layer can be established for the common vocabulary tokens.

5. Influence of observations on the NMT system

The sentence length varies from 49 words in a particular sentence to 3 words in a sentence. The long-range dependency and semantic coherence present in such sentences will be difficult for NMT to capture. Further, shorter sentences will also be a hindrance as there is a lack of contextual knowledge. By treating different inflected forms of the same lemma as distinct tokens, the vocabulary size unnecessarily increases, leading to a redundant and larger vocabulary at the same time losing valuable information by creating $< unk >$ to a larger section of words. Specifically discussing English and German translations, the frequent suffixes such as "*-ing, -s*" and prefixes such as "*Auf- , Gegen*" play a crucial role in determining the meaning of a word. The word-level tokenization fails to capture such nuances and assigns $< unk >$ to these words as a whole. Abruptly replacing rare words with $< unk >$ might result in the loss of vital contextual information for translation. This issue is particularly pronounced in languages like German, where the gender of a word determines the article (*der/die/das*) to be used, and masking such key terms with $< unk >$ can directly impact the rest of the translation.

Adopting a subword-level tokenization approach, such as Byte Pair Encoding (BPE) or SentencePiece, can be beneficial to mitigate these challenges. By segmenting rare and compound words into meaningful subword units, the NMT system can handle morphological variations more effectively and reduce the number of unknown words encountered during translation.

3 Improved Decoding

3.1 Problems with Greedy 1-best decoding

The problem with greedy decoding is that it selects the highest probability output at a given step from the probability distribution, which can limit the exploration space of other possible translations. Since there is no sampling process done, the model is likely to give out predictions that are accurate locally but not at the full sentence (global) level. The algorithm is not guaranteed to find the translation with the highest probability for the entire translation. In greedy 1-best, we calculate the probability distribution p_t at every time step t and select the word with the highest probability at that time step i.e. $\underset{i}{\text{argmax}}(p_{t,i})$

Additionally, since the model does not guarantee the best translation, the output translation may be discontinuous and lack coherence in the output sentence. The semantics of translation can get lost in such a situation, resulting in not-so-satisfactory translations. Also since there's no way to lookback to find a more optimal path, greedy 1-best lacks the ability to backtrack.

An example we observe in the given dataset is the appearance of the keyword "europäisch," which, when translated to English, is "european" (appears 364 times in the train set). The translated word has a variation 137 different words following itself. The same keyword in the test set, when put through the baseline model, generates only nine possible variations, all of which are frequently occurring bigrams and account for 213 of those 364 occurrences of "european". Thus, we can establish that the greedy 1-best would preferably pick the best word on the basis of the highest probability, likely to be one of those nine frequently occurring combinations with the keyword "european".

3.2 Beam Search

Algorithm 1 Beam Search

sequence beam $\leftarrow$ ([*start token*], 1)
while *generation length* <= *maximum length or EOS token* **do**
  candidates $\leftarrow \phi$
  for *sequence*, *prev probability in sequence beam* **do** ▷ Loop till EOS or max length
    current state $\leftarrow$ *decoder*(*sequence*)
    for *current word*, *prob in current state* **do**
      current prob $\leftarrow$ *prev probability* $*$ *prob*
      candidates.append((*sequence* + *current word*, *current prob*))
    end for
  end for
  sorted candidates $\leftarrow$ *sort*(*candidates*) ▷ sort on basis of probability
  sequence beam $\leftarrow$ *sorted candidates*[: *beam width*] ▷ Repeat till EOS or max length
end while
Output sequence $\leftarrow$ *max*(*sequence beam*) ▷ maximum on basis of probability

The advantage of using beam search over just 1-best greedy search is that the model would consider b best possible translation at each time step where b is the beam width. The psuedocode for the algorithm is given in the Algorithm 1 above.

3.3 Encouraging longer sentence generation

Every time we add another word to the translated output, we multiply probabilities (naturally less than 1), thus reducing the probability of the sentence even further. With beam search, as we increase the beam width, the affinity towards shorter translations increases; thus, we don't get long sentences as output translations. To fix the shorter length bias, we can do the following -

1. One way to deal with the length bias is by adding a prior probability on sentence length conditioned on the length of the input sentence. This probability can then be multiplied by the original probability, which was used to model the translation, i.e., $P(E|F)$, where F is the source, and E is the translated sentence. The prior probability $P(|E|||F|)$ can be estimated using the counts of the length of the target sentence and the source sentence over the length of the source sentence. The final equation can be given as follows -

$$\hat{E} = \underset{E}{\operatorname{argmax}} \log \frac{c(|E|, |F|)}{c(|F|}) + \log P(E \mid F) \tag{1}$$

2. Another approach can be to normalize the log probability of modeling with the length of the translation, thus searching for sentences with the highest log average probability per word.

$$\hat{E} = \underset{E}{\operatorname{argmax}} \log \frac{P(E \mid F)}{|E|} \tag{2}$$

4 Adding Layers

1. Deeper model training command

The command used to train the LSTM NMT with the number of layers in encoder =2 and decoder = 3 is (default value of arch is lstm):

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

2. Analysis of the deeper model

Ideally, increasing the model complexity should increase the model's capacity to capture more complex patterns in the data. The increased number of parameters can result in slower convergence. This increased capacity also makes the model more prone to overfitting the training data, especially if the training dataset is relatively small or the regularization techniques are not sufficient. Further, for such a complex architecture, a more diverse and intense dataset is required for the model to generalize and prevent it from memorizing the training data. In this case, the structure with more layers seems to overfit the training data,

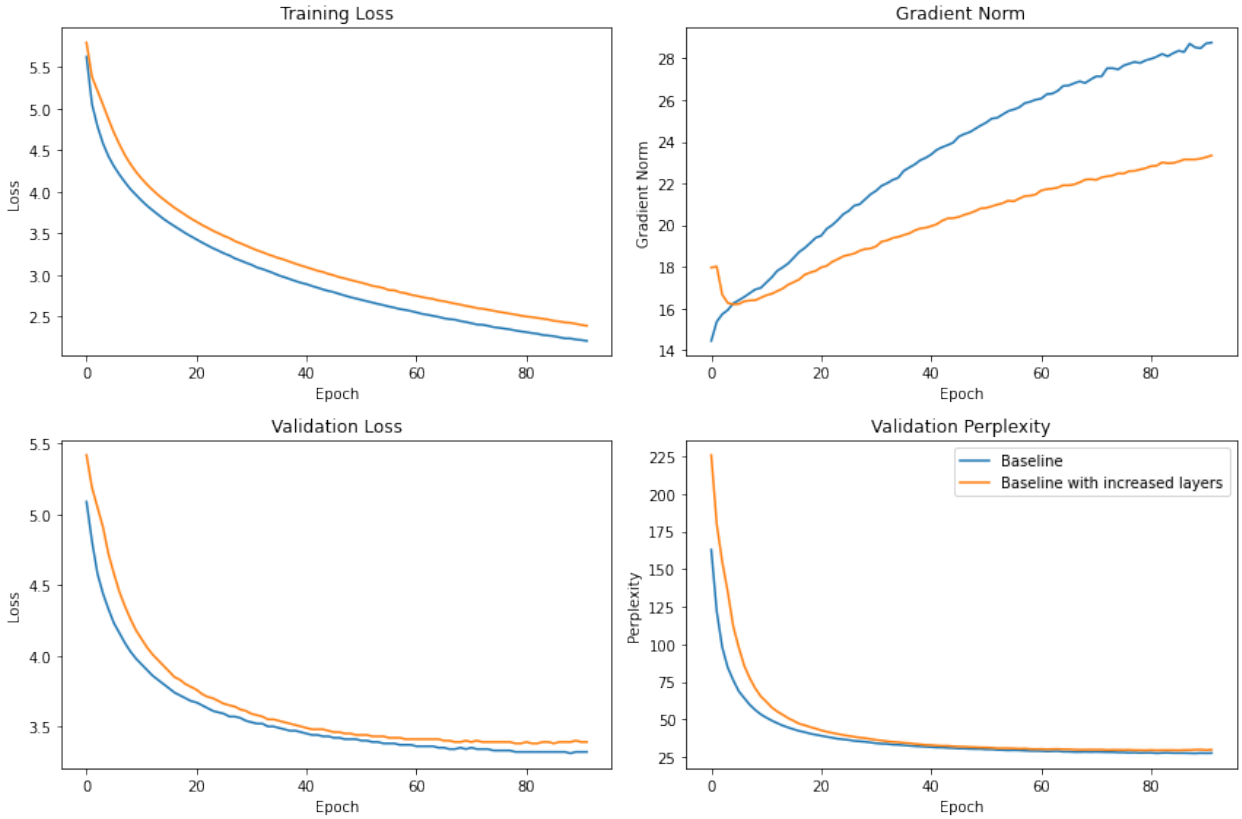


Figure 1: Performance comparison of Baseline and enhanced LSTM models

as evidenced by the higher training BLEU score (20.67 for baseline vs. 15.69 for complex architecture) and lower training loss (2.141 for baseline vs. 2.389 for complex architecture) along with a higher validation loss (3.31 for baseline vs. 3.4 for complex architecture). This suggests that the larger model memorizes the training data better than the smaller model but at the cost of poorer generalization to the validation and test sets.

On the validation set, the baseline structure with fewer layers achieves a higher BLEU score (11.48 vs. 10.76 for complex architecture) and lower perplexity (27.2 vs. 29.4 for complex architecture), indicating better generalization to unseen data. The complex model with a higher initial validation loss tends to attain a plateau, indicating overfitting. This is

likely because the smaller model has a lower risk of overfitting and can learn more robust representations from the training data. Test Set: On the test set, the baseline model achieves a higher BLEU score of 10.89, compared to 9.52 for the complex model. The gap in test BLEU scores between the two models (10.89 vs. 9.52) are more significant than the gap in validation BLEU scores (11.48 vs. 10.76), suggesting that the larger model's generalization issues are more pronounced on the test set.

WHY IS THIS THE CASE? The baseline model has approximately 1.4M parameters, and the complex model has 1.8M parameters, but the provided data contains only 10K sentences for training. The dense model, upon training, resulted in higher variance and lower bias, leading to overfitting and poor generalization of unseen data. Another hypothesis could be that one or more layers became ineffective, resulting in redundant computation, which can also be seen in Figure 1.

5 Implementing the Lexical Model

Projections initialisation

```
# __QUESTION-5: Add parts of decoder architecture corresponding to the LEXICAL MODEL here
# TODO: ---------------------------------------------------- CUT
self.lexical_projection_hidden = nn.Linear(embed_dim, embed_dim, bias=False)
self.lexical_projection_output = nn.Linear(embed_dim, len(dictionary))
# TODO: ---------------------------------------------------- /CUT
```

Lexical model context vectors

```
# __QUESTION-5: Compute and collect LEXICAL MODEL context vectors here
# TODO: ---------------------------------------------------- CUT
attn_weights_reshaped = step_attn_weights.unsqueeze(dim=1)
# calculating the weighted average of source embeddings with attention weights
src_context = torch.bmm(attn_weights_reshaped,src_embeddings.transpose(0,
↪  1)).squeeze(dim=1)
lexical_activated_context = torch.tanh(src_context)
# adding the skip with a tanh activation
hidden_src_context = torch.tanh(self.lexical_projection_hidden( \
                                lexical_activated_context)) + lexical_activated_context
lexical_contexts.append(hidden_src_context)
# TODO: ---------------------------------------------------- /CUT
```

Incorporating the Lexical model

```
# __QUESTION-5: Incorporate the LEXICAL MODEL into the prediction of target tokens here
# TODO: ---------------------------------------------------- CUT
lexical_contexts = torch.cat(lexical_contexts, dim=0).view(tgt_time_steps, \
                                        batch_size, self.embed_dim)
lexical_contexts = lexical_contexts.transpose(0, 1)
decoder_output += self.lexical_projection_output(lexical_contexts)
# TODO: ---------------------------------------------------- /CUT
```

Lexical model interpretation

The baseline model can often run into issues where it tries to fit the translation words based on the context in the target sequence it has generated before that time step while ignoring the source word. For instance, if the German sentence "Lied von Harry Styles" is translated into English, the model will output "Song by Harry" till the third time step but might output Potter at the fourth time step if the target sentences have had more occurrences of "Potter" with "Harry" as compared to "Styles" in training.

The lexical model addresses this issue using a small feed-forward neural network(FFNN). The network takes a weighted average of the source embeddings with respect to attention weights i.e. $f_t^\ell = \tanh \sum_s a_t(s) f_s$, where a_t are the attention weights, f_s are the source word embedding and s ranges between 1 to m where m is the length of the source sentence. The output of the FFNN is combined with the decoder output, as shown by code blocks below.

The lexical model outperforms the baseline model comfortably in all evaluation metrics.

	Train epochs (before early stopping)	Train Loss	Validation Loss	Validation Perplexity	Test Bleu Score
Baseline Model	99	2.1	3.3	27.4	10.9
Lexical Model	53	1.9	3.2	24.5	13.5

Table 2: Lexical model comparison with Baseline

The bleu metrics on the test see a considerable rise, along with the validation perplexity dropping almost 3 points. Additionally, the model convergence is also faster in this case, as we stop the training after 53 epochs, thus giving better results with almost half the training time. Moreover, the introduction of the FFNN makes the lexical model slightly bulky, sitting at 1.7 mn params. However, given the increase in metrics and faster convergence, this parameter addition does not create a considerable overhead and seems like a justified enhancement to the baseline model.

The following examples from the test set validate our hypothesis that the lexical model works better when considering context and source to generate more meaningful translations.

- **Input Text**: "herr barroso wird dafür natürlich als kandidat ohne gegenkandidat , ohne alternative dastehen ."

  Output Text(Lexical Model): "mr barroso will be able to see without any financial operational without general without a number without financial ."

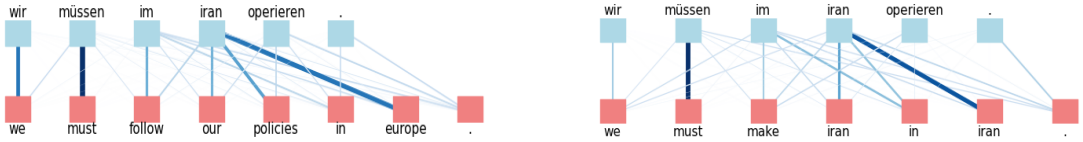
  The baseline model for this input sentence gives no translation whatsoever. On the other hand, the lexical model was able to retain the word "Barroso" in the output from the source language, while even if the baseline model had given a translation, it would have predicted the next translated word as "president." This is because the keyword "president" is followed by the context translation "mr" 273 times while "barroso" is followed 3 times in the training data. As a result, even in the baseline output translations, oftentimes "president" is followed by "mr".

- **Input Text**: "wir müssen im iran operieren ."

  Output Text(Baseline): "we must follow our policies in europe ."

  Output Text(Lexical): "we must make iran in iran ."

  In this particular example, similar to the previous one, the baseline completely misses the context that the policies are being talked about for "Iran" and not "Europe". The output translation from the lexical model also semantically is not an ideal translation, but it captures the country's information. The attention map clearly shows the lexical models' strong emphasis on the word "iran". The information related to nouns and noun phrases holds high importance and is missing in the baseline model, which can be detrimental to translations.



(a) Attention maps for base model (b) Attention maps for lexical model

Figure 2: Attention maps for German to English translation for Example 2

- **Input Text**: "ich danke herrn lindqvist ."

  Output Text(Baseline): "thank you , commissioner ."

  Output Text(Lexical): "thank you , mr van ."

  In the output translations from the baseline model, the model tries to fit the commissioner in the sentence whenever the phrase "thank you" appears. Out of the 120 times this phrase has appeared in the translated training file, "commissioner" has accompanied it in the sentence 29 times, thus making it a likely choice of word in translation because of the context "thank you". The lexical model also does not give the right translation as it misses "lindqvist" but captures the translation of "herrn" and has a strong attention weight for "mr" and "herrn". Additionally, "lindqvist" never appears in the training set of German sentences, thus explaining the failure in generating the keyword by the lexical model.

(a) Attention maps for base model (b) Attention maps for lexical model

Figure 3: Attention maps for German to English translation for Example 3

- **Input Text**: "(unterbrechung durch die präsidentin .)"

  Output Text(Baseline): "(the president cut off the speaker)"

  Output Text(Lexical): "(the president of the president)"

  The above example gives an idea of when this baseline model will likely outperform the lexical model. The phrase mentioned above is frequently used in the training files, appearing a total of 10 times and verbatim as is. Thus, the baseline model generates the output exactly as it appeared so many times. In contrast, the lexical model gives out variations because of the attention-weighted average of source sentences (the attention weights would define the output translation).

6 Understanding the Transformer Model

Following code blocks discuss our understanding of the transformer model.

6 A

```
'''
___QUESTION-6-DESCRIBE-A-START___
1.  Add tensor shape annotation to each of the output tensor
2.  What is the purpose of the positional embeddings in the encoder and decoder?
    -   positional embeddings are used to add information about the position of the word
    ↪  in the sentence.
    -   Resulting in variant representations of the same word depending on its position
    ↪  in the sentence.
    -   Simultaneously the parallelism of the transformer architecture is maintained
    ↪  without losing context.
    -   In both the encoder and decoder, the positional embeddings help in keeping track
    ↪  of the order of
        the words in the sentence.

'''
embeddings += self.embed_positions(src_tokens)
# embeddings = [batch_size, src_time_steps, num_features]

forward_state = F.dropout(embeddings, p=self.dropout, training=self.training)

# Transpose batch: [batch_size, src_time_steps, num_features] -> [src_time_steps,
↪  batch_size, num_features]
forward_state = forward_state.transpose(0, 1)

# Compute padding mask for attention
encoder_padding_mask = src_tokens.eq(self.padding_idx)
if not encoder_padding_mask.any():
    encoder_padding_mask = None

# Forward pass through each Transformer Encoder Layer
for layer in self.layers:
    forward_state = layer(state=forward_state, encoder_padding_mask=encoder_padding_mask)

return {
    "src_out": forward_state,   # [src_time_steps, batch_size, num_features]
    "src_embeddings": src_embeddings,   # [batch_size, src_time_steps, num_features]
    "src_padding_mask": encoder_padding_mask,   # [batch_size, src_time_steps]
    "src_states": []    # List[]
}
'''
___QUESTION-6-DESCRIBE-A-END___
'''
```

6 B

```
'''___QUESTION-6-DESCRIBE-B-START___
1.  Add tensor shape annotation to each of the output tensor
2.  What is the purpose of self_attn_mask?
    -   The self_attn_mask is used to prevent the decoder from attending to subsequent
    ↪  tokens in the target sequence.
3.  Why do we need it in the decoder but not in the encoder?
    -   In the decoder the word prediction at time step t shpuld be dependent on the
    ↪  words at time steps 0 to t-1.
    -   the self_attn_mask is used to prevent the decoder from looking at the future
    ↪  words in the sequence.
    -   In the encoder, the self_attn_mask is not needed as the word order might change
    ↪  from the source to the target.
        for example, in the case of translation, the word order in the source language
        ↪  might not be the same as the
        target language. as a result a holistic view of the source sentence is needed. a
        ↪  similar view in decoder will
        result in next word prediction and not the translation.
4.  Why do we not need a mask for incremental decoding?
    -   The decoder generates the output one token at a time, it only has access to the
    ↪  previously generated tokens and the encoder
        representations, indicating that the decoder naturally cannot attend to future
        ↪  tokens that have not been generated yet.

'''
self_attn_mask = self.buffered_future_mask(forward_state) if incremental_state is None
↪  else None
# self_attn_mask = [tgt_time_steps, tgt_time_steps]
'''___QUESTION-6-DESCRIBE-B-END___'''
```

6 C

```
'''___QUESTION-6-DESCRIBE-C-START___
1.  Why do we need a linear projection after the decoder layers?
    -   The linear projection is used to map the decoder output to the target vocabulary
    ↪  size.
    -   The linear projection helps in converting the higher dimensional complex data
    ↪  into generating
        the probability distribution over the target vocabulary. directing the model to
        ↪  result in
        correct output space.
2.  What would the output represent if features_only=True?
    -   If features_only=True, the output would represent the decoder output after the
    ↪  last layer.
        without the linear projection, hence a higher dimensional complex data. it
        ↪  represents the
        learned information from the input sequence till the current time step.
'''
forward_state = self.embed_out(forward_state)
'''___QUESTION-6-DESCRIBE-C-END___'''
```

6 D

```
'''
___QUESTION-6-DESCRIBE-D-START___
1.  What is the purpose of encoder_padding_mask?
    - The purpose of encoder_padding_mask is to mask the padding tokens in the input
    ↪  sequence.
    - This is important because the padding tokens do not contain any useful information
    ↪  and should
      not be used in the attention mechanism.
    - This enables the attention mechanism to focus on the meaningful tokens in the input
    ↪  sequence.
'''
state, _ = self.self_attn(query=state, key=state, value=state,
↪  key_padding_mask=encoder_padding_mask)
'''
___QUESTION-6-DESCRIBE-D-END___
'''
```

6 E

```
'''
___QUESTION-6-DESCRIBE-E-START___
1.  How does encoder attention differ from self attention?
    - The self attention mechanism attends the dependencies between different words in
    ↪  the input and applied within the encoder.
    - The encoder attention mechanism captures the dependencies between the input and
    ↪  output sequences - between the encoder
      and decoder.
2.  What is the difference between key_padding_mask and attn_mask?
    - key_padding_mask is used to mask the padding tokens in the key matrix.
      has same shape as key matrix and contains binary values. it ensure that decoder
      ↪  doesn't attend these padded tokens
    - attn_mask is used to mask the attention scores. It prevents the model from
    ↪  attending to future tokens.
      consisting of -inf values, in the upper triangular part of the attention matrix
3.  If you understand this difference, then why don't we need to give attn_mask here?
    - The attn_mask is not needed for calculating the encoder attention in the decoder
    ↪  layer. Though attn_mask prevents attending
      to future tokens, it is not needed in the decoder layer as all tokens are necessary
      ↪  for the decoder to translate.
    - Not adding attn_mask here ensures that the decoder attends to all tokens
'''
state, attn = self.encoder_attn(query=state,
                                key=encoder_out,
                                value=encoder_out,
                                key_padding_mask=encoder_padding_mask,
                                need_weights=need_attn or (not self.training and
                                ↪  self.need_attn))
'''
___QUESTION-6-DESCRIBE-E-END___
'''
```

7 Implementing multi-headed attention

The below code illustrates the implementation of the Multihead attention mechanism.

```
'''
___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)]

# project the queries, keys and values to the multi-head attention space
q_projected = self.q_proj(query) # q_projected size [tgt_time_steps, batch_size,
↪  embed_dim]
k_projected = self.k_proj(key)   # k_projected size [tgt_time_steps, batch_size,
↪  embed_dim]
v_projected = self.v_proj(value) # v_projected size [tgt_time_steps, batch_size,
↪  embed_dim]

# split the queries, keys and values into num_heads
q_head = q_projected.reshape(tgt_time_steps, batch_size * self.num_heads,
↪  self.head_embed_size)
# q_head size [tgt_time_steps, batch_size * num_heads, head_embed_size]
k_head = k_projected.reshape(-1, batch_size * self.num_heads, self.head_embed_size)
# k_head size [tgt_time_steps, batch_size * num_heads, head_embed_size]
v_head = v_projected.reshape(-1, batch_size * self.num_heads, self.head_embed_size)
# v_head size [tgt_time_steps, batch_size * num_heads, head_embed_size]

# transpose the queries, keys and values to perform the attention operation
q_head = q_head.transpose(0, 1) # q_head size [batch_size * num_heads, tgt_time_steps,
↪  head_embed_size]
k_head = k_head.transpose(0, 1) # k_head size [batch_size * num_heads, tgt_time_steps,
↪  head_embed_size]
v_head = v_head.transpose(0, 1) # v_head size [batch_size * num_heads, tgt_time_steps,
↪  head_embed_size]

# calculate the attention scores
attn_scores = torch.bmm(q_head, k_head.transpose(1, 2))
attn_scores/= self.head_scaling
# attn_scores size [batch_size * num_heads, tgt_time_steps, tgt_time_steps]

# applying dropout to the attention scores
if self.attention_dropout > 0:
```

```
    attn_scores = F.dropout(attn_scores, p=self.attention_dropout,
    ↪  training=self.training)
    # attn_scores size [batch_size * num_heads, tgt_time_steps, tgt_time_steps]

# apply the key_padding_mask to the attention scores
if key_padding_mask is not None:
    attn_scores = attn_scores.view(batch_size, self.num_heads, tgt_time_steps, -1)
    # attn_scores size [batch_size, num_heads, tgt_time_steps, tgt_time_steps]
    attn_scores = attn_scores.masked_fill(key_padding_mask.unsqueeze(1).unsqueeze(2),
    ↪  float('-inf'))
    # attn_scores size [batch_size, num_heads, tgt_time_steps, tgt_time_steps]
    attn_scores = attn_scores.view(batch_size * self.num_heads, tgt_time_steps, -1)
    # attn_scores size [batch_size * num_heads, tgt_time_steps, tgt_time_steps]

# apply the attention mask to the attention scores
if attn_mask is not None:
    attn_scores = attn_scores + attn_mask
    # attn_scores size [batch_size * num_heads, tgt_time_steps, tgt_time_steps]

# apply the softmax function to the attention scores
attn_scores = F.softmax(attn_scores, dim=-1)
# attn_scores size [batch_size * num_heads, tgt_time_steps, tgt_time_steps]

# estimate the attention values and attention weights
attn = torch.bmm(attn_scores, v_head)
# attn size [batch_size * num_heads, tgt_time_steps, head_embed_size]
attn = attn.transpose(0, 1).contiguous().view(tgt_time_steps, batch_size, embed_dim)
# attn size [tgt_time_steps, batch_size, embed_dim]
attn = self.out_proj(attn)
# attn size [tgt_time_steps, batch_size, embed_dim]
attn_weights = attn_scores.view( self.num_heads,batch_size, tgt_time_steps, -1) if
↪  need_weights else None
# attn_weights size [num_heads, batch_size, tgt_time_steps, key.size(0)]
'''
___QUESTION-7-MULTIHEAD-ATTENTION-END
'''
```

Transformer Results

The Transformer's performance compared to previous LSTM-based models is depicted in Figure 4, and the corresponding numerals are stated in Table 3. The transformer model performs better than the baseline and complex models for the test set and with a marginal difference with the development set. The Transformer has the least training loss and a high validation loss, indicating the model has overfitted the training data set. Rather than generalizing the model began to learn the noise and uniqueness present within the dataset. In addition to that, Figure 4 suggests that the training loss decreases rapidly as compared to other models; however, the validation loss and perplexity shoot up after a certain epoch, indicating a lack of a diverse dataset sufficient enough for such a heavy model (2.7M parameters twice as much as parameters as baseline).

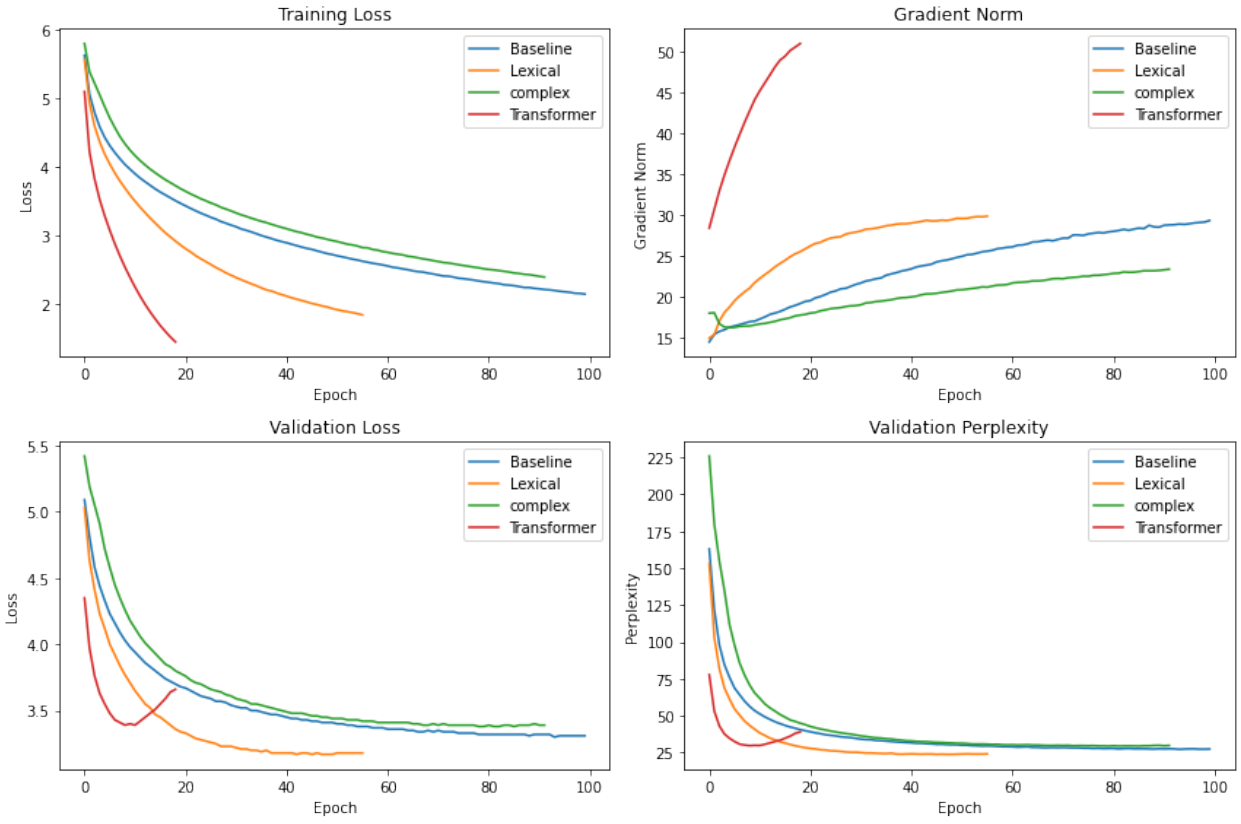


Figure 4: Performance comparison of Baseline, Lexical, Complex, and Transformer models

Being both Indo-European languages, German and English share a relatively similar syntactic structure. The recurrent nature of LSTMs may be suited to capture the syntactic patterns and word order similarities between these two languages, compared to the self-attention mechanism of transformers. LSTMs are generally more effective at handling smaller vocabularies, as they can better capture the co-occurrence patterns and dependencies within the limited vocabulary space. We can hypothesize that positional embeddings help transform-

Metrics	Baseline	Complex	Lexical	Transformer
Test BLEU	10.89	9.52	**13.5**	11.39
Training loss	2.141	2.389	1.9	**1.443**
validation loss	3.31	3.4	**3.2**	3.39
Perplexity	27.2	29.4	**24.5**	29.7
Validation BLEU	11.48	10.76	**14.43**	10.98
Train BLEU	20.67	15.69	**23.46**	16.24

Table 3: Performance comparison of Baseline, Lexical, Complex, and Transformer models

ers encode the position of words in a sequence, LSTMs are specifically designed to capture such dependencies by processing the input sequentially. This sequential processing may be more effective in capturing the intricate dependencies and patterns present in the German-to-English translation task (in particular, with the small dataset/ low-resource dataset).

Transformer Analysis

The transformer model performs better than the baseline and complex models for the test set and with a marginal difference with the development set (as the test BLEU score is high). The Transformer has the least training loss (1.443) and a high validation loss (3.39), indicating the model has overfitted the training data set. Rather than generalizing the model began to learn the noise and uniqueness present within the dataset. In addition to that, Figure 4 suggests that the training loss decreases rapidly as compared to other models; however, the validation loss and perplexity display a convex curve, indicating a lack of a diverse dataset sufficient enough for such a heavy model (2.7M parameters twice as much as parameters as baseline) hence it began to overfit.

Since the provided data is very limited, information on the previous words in the decoder is required along with the input sequence. Hence, the LSTM-based implementation, along with lexical advantages, will be able to perform better, whereas the transformers don't function sequentially, hence demanding more data to optimize the parameters in the decoder to generalize or begin to overfit. However, despite having a higher BLEU score, the lexical model fails in situations where there are complex sentence structures with prepositional phrases. A sample of such a situation is depicted below in Figure 5 and Figure 6.

- Original English sentence, "*We need more courage to overcome the crisis using European resources,*"
- The Lexical model's translation, "*we need more courage to courage with the new market .* " suggests a misunderstanding of the context, particularly the notion of using European resources to address the crisis. It seems to have fixated on the immediate context around "courage" without properly integrating the latter part of the sentence.
- The transformer model's translation "*we need more courage to look at the european crisis .*" indicates that the Transformer was more contextually aligned with the input sentence.

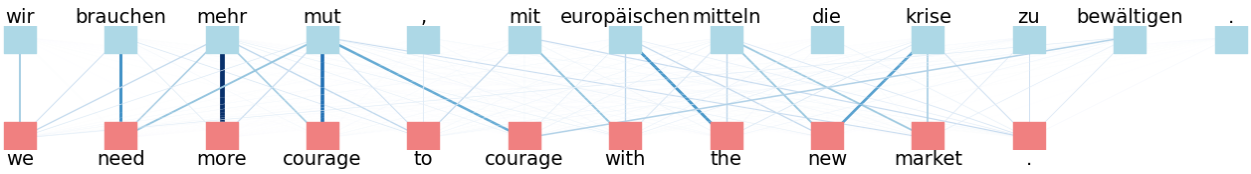


Figure 5: Attention map of Lexical model with sample statement

This example indicates that the LSTM, despite generalizing well across similar sentences that they have encountered, their performances are affected in scenarios where sentences have complex expressions. On the other hand, the transformer model, with its attention mechanism, can capture such nuances.

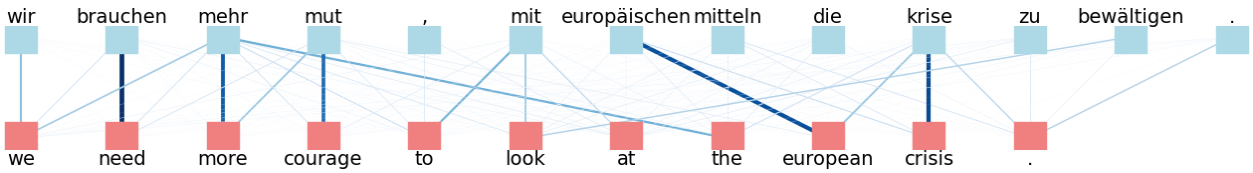


Figure 6: Attention of Transformer model with sample statement

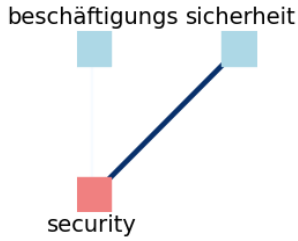


Figure 7: Attention map for Transformer model for input : *beschäftigungs sicherheit*

Further ineffective tokenization is the major reason for this result. As mentioned previously, words with similar lemma are tokenized separately in the target language. In the source language, German, the presence of compound words also leads to a similar problem. The test document contains approximately 21% of words that aren't present in the training data. Words such as *kontrolleure, beschäftigungssicherheit* have 0 occurrences in train data; implementation of proper subword tokenization would have solved this problem. As in the case of *kontrolleure*, the training data has 20 occurrences of *kontrolle* and 30 occurrences of *sicherheit*. This has a major impact on the model's performance. The word as a whole isn't present but the subwords are present in needful quantity.

The word "*beschäftigungs sicherheit*" as a whole failed to render any possible translation in all four models, whereas the corresponding subwords upon translating were able to produce meaningful references as shown in Figure 7 and Figure 8. In addition to the tokenization impact, we could also infer that the impact of previous word tokens plays a vital role in Lexical model performance, whereas the Transformer's results are majorly influenced by the source context rather than the previous time step outputs. Errors in the early parts of the sentence can propagate and affect the translation of subsequent parts in LSTM architecture. The repetition of "safety security" in the Lexical model's translation might be an instance of

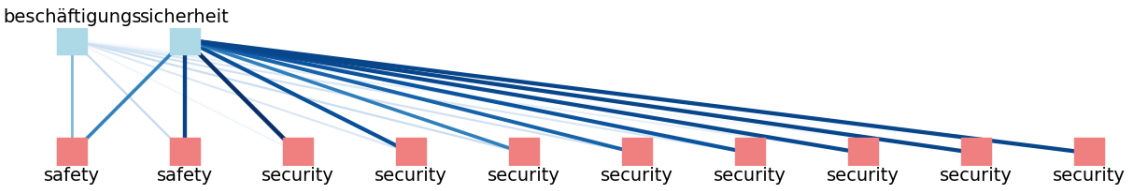


Figure 8: Attention map for Lexical model for input : *beschäftigungs sicherheit*

such error propagation, where the model loses track of the sentence structure after incorrectly handling the word "security".

Compared to the base model, the transformer model converges faster because of the ability to parallelize transformer training and the fact that the transformer model deals with gradients efficiently. While transformers are theoretically known to outperform LSTM models, the behavior from the graphs can be majorly attributed to the following two reasons -

- **Limited data and the data quality**

  Transformer models are usually data greedy, i.e., they excel when the amount of training data is larger and also the quality of the data is really good. In this case, we train the model on only 10k training examples, which is comparatively small. Out of the 5k words in the German dictionary in our dataset, 1.6k of them occur only two times. Similarly, of the 4.4k words in the English dictionary, 1.2k appear only two times. Additionally, German nouns have plural forms based on their gender, and understanding the nuances of German-to-English translation would require huge amounts of data to efficiently understand these underlying patterns.

- **Model complexity and model capacity**

  Transformers are known to handle the long-term dependencies due to their self-attention mechanism. Moreover, they use positional embeddings, which makes them a complex choice over LSTM models, which have a recurrent architecture and memory cells and gates to regulate information flow. For this complexity to generalize well onto the performance, the parameters need significant updates to leverage the transformer's ability.

Transformer improvement

Following are the ways to enhance the efficient training of transformers -

- **Increase the training data**: One way to satisfy the transformer's data-intensive requirement is to get more training data. Parallel corpora can be a great source to amplify data since they are typically curated by translators, resulting in high-quality data, and are also very diverse in nature.

- **Data augmentation**: Augmentation techniques like backtranslation can yield better performances on the validation and test sets. In backtranslation, we train a translation model from the source language to the target language and use this model to generate synthetic pairs by running a subset of our target sentences through the model to generate source language sentences. This creates synthetic pairs, which are much more diverse as the generated sentences may differ in sentence structures, word choices, etc.

- **Hyperparameter Tuning**: Another way to enhance the models' performance is by tuning the hyperparameters to find the optimal set that is best suited for our problem statement. We can also experiment with a decayed learning rate so that as we progress in our training, i.e., reach closer to the minima, the weight updates become smaller. Moreover, we can also look for different loss functions to experiment with or change the dimensions of the encoding/decoding layers. As suggested by Araabi in Optimizing Transformer for Low-Resource Neural Machine Translation[2]. A reduced number of parameter spaces can be used in the current scenario for NMT displaying better performance than the transformer-base in Vaswani's paper - Attention Is All You Need[3].

- **Subword Tokenisation**: Using a tokenization method like Byte-Pair encoding can be beneficial for this low-resource translation model. With BPE, we would be able to tie the source and target language embedding better, thus learning better representations. This impact is effectively discussed in these 2 papers [4] [5], where low-resource NMT can be optimized more effectively with proper tokenization

- **Multi-lingual pre-training and fine tuning**: Another approach to increase accuracy for transformers on the task of machine translation with such a small dataset can

[2]Ali Araabi and Christof Monz. 2020. Optimizing Transformer for Low-Resource Neural Machine Translation. In Proceedings of the 28th International Conference on Computational Linguistics, pages 3429–3435, Barcelona, Spain (Online). International Committee on Computational Linguistics.

[3]Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. CoRR, abs 1706.03762 .

[4]as suggested by Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016 in Neural Machine Translation of Rare Words with Subword Units

[5]A. Pramodya, R. Pushpananda and R. Weerasinghe, "A Comparison of Transformer, Recurrent Neural Networks and SMT in Tamil to Sinhala MT," 2020 20th International Conference on Advances in ICT for Emerging Regions (ICTer)

be to train a transformer on corpora that has multiple languages, including German and English. We would be able to learn shared representations across languages better and then use this to fine-tune it on our German-English translation data.

- Further implementing **Lexical-based implementation** in the output layer of the decoder as mentioned in Improving Lexical Choice in Neural Machine Translation by Nguyen[6]. could enhance the impact of source sentences at the decoder level.

- Implementation of a **combined model such as RNMT+** as suggested by Xu Chen in The Best of Both Worlds: Combining Recent Advances in Neural Machine Translation[7]. Where the advantage of both transformer and LSTM-based architecture is utilized in the encoder and decoder respectively.

[6]Nguyen, T. Q., & Chiang, D. (2017). Improving lexical choice in neural machine translation. _CoRR, abs 1710.01329_.

[7]Chen, M. X., Firat, O., Bapna, A., Johnson, M., Macherey, W., Foster, G. F., ... & Hughes, M. (2018). The best of both worlds: Combining recent advances in neural machine translation. CoRR, abs 1804.09849.