NLU CW2

B150821, B161564

February 2024

Question 1: Understanding the Baseline Model

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

1-A-2:
    The mask is applied to the attention scores to deal with sentences of
    different lengths. Shorter sentences are padded with -inf so that when
    softmax is applied the attn_weights for these evaluate to 0. This makes the
    attention mechanism ignore the tokens that were padded with -inf.

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

1-B-2:
    The encoder output is projected using a linear layer to the dimensions of
    output_dims. This is then transposed so that the dimensions match up with
    dimensions needed to work with tgt_input. The tgt_input is then unsqueezed
    and batch matrix multiplication of the unsqueezed tgt_input and the
    projected encoder output is performed to obtain the attn score.

1-C-1:
    Cached state is None when nothing is fed into the incremental_state
    parameter, which happens when the model isn't run in incremental mode. It is
    also none when the key 'cached_state' is not in the incremental_state
    dictionary, which should happen at the start of the model run.

1-C-2:
    input_feed is the output from the previous time step and is concatenated to
    the current token embedding. This is then used as LSTM input for the current
    timestep. If attention is used, input_feed is attended to before being used
    as input to the LSTM for the current iteration.

1-D-1:
    The previous target state is given to the attention function to calculate
    the attention weights for the current target state. This is due to how
    attention is defined, its calculation depends on the previous state.

1-D-2:
    The dropout layer helps to prevent overfitting and over-parameterization of
    the model. By setting layer units to 0 at random, the model needs to learn
    more robust features and is more likely to generalise well to the test data.

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

1-E-2:
    # Compute the output of the model and attention scores by providing
    # source tokens, source lengths and target inputs to the model.
    # Attention weights are not ignored.
    output, _ = model(sample['src_tokens'], sample['src_lengths'], sample['tgt_inputs'])

    # This calculates the cross-entropy loss between the output and the target tokens.
    # It is normalised by the number of tokens in the source sentence.
    loss = \
        criterion(output.view(-1, output.size(-1)),
            sample['tgt_tokens'].view(-1)) / len(sample['src_lengths'])

    # This is calculates the gradients of the loss with respect to the model parameters,
    # in a backward pass.
    loss.backward()
    # clip_grad_norm is used to prevent the gradients from becoming too large.
    grad_norm = torch.nn.utils.clip_grad_norm_(model.parameters(), args.clip_norm)
    # This takes a step in the direction of the gradients,
    # updating the model parameters.optimizer.step()
    # Reset the gradients to zero to prepare for the next iteration.
    optimizer.zero_grad()
```

Question 2: Understanding the Data

1. Split by whitespace, there are 124031 tokens in the English data with 8326 word types. 112572 tokens in the German data with 12504 word types.

2. In the English data, 3909 tokens will be replaced; the total vocabulary size before is 8326, after is 4417. In German, 7460 tokens will be replaced; the total vocabulary size before is 12504, after is 5044.

3. Inspecting the replaced words, we can divide them into two categories: uncommon words and common words. While there are some uncommon words (especially numbers and proper nouns), the common word category is dominant. It is noticeable that many of these common words share the same root (e.g. [cite, cited, cites]), due to morphology and inflection of the two languages. Lemmatisation could be used to handle this during tokenisation as it reduces a word to its base form.

   For German, the uncommon word category seems to be larger due to compound words (e.g. boden-abfertigungsdienste). Special rule-based tokenisation / sub-word tokenisation could be used to handle these.

4. There are 1460 shared tokens between the two languages (478 if removing UNK tokens). Disregarding false friends, this similarity could be exploited by using them as "anchors" to help align the two languages, based on the assumption that the shared tokens will be used in a similar context.

5. Sentence length:

   - For NMT, translation quality is expected to perform worse as sentence length increases [1]. In the context of our model, this has also been shown for attention-based LSTM models [2].
   - This can be attributed to the vanishing gradient problem where the gradients become too small to be useful for learning as the sequence length increases.
   - The choice of tokenisation processes affects token counts and sentence lengths (e.g. non-compound-splitting tokenisation would lead to shorter German sentences as English sentences). This further complicates the translation task.

   Tokenisation process:

   - The choice of tokenisation process directly affects the performance of the NMT model. For example, the use of sub-word tokenisation (especially optimised for German compound words in the context of this task) could improve the translation quality as it increases the number of examples for the subtokens [3].

   Unknown words:

   - Replacing single-occuring words with UNK tokens is a trade-off between a potential loss of information and reducing vocbulary size, hence making the training process computationally less expensive.
   - The performance of the model, the UNK token replacement approach has been observed to lead to worse translation quality as it effectively 1) removes meaning, 2) breaks the structure of sentences and 3) hence hurts translation and reordering of in-vocabulary words [4].

Question 3: Improved Decoding

1. Greedy decoding selects the token with the highest probability at each time step. This could be problematic where the model makes a mistake early on and is unable to recover from it later in the sequence because greedy decoding only keeps one sequence continuation at a time.

   Example:

   - Original: "I am a student housing administrator."
   - Correct translation: "Ich bin ein Studentenwohnheim-Verwalter."
   - Incorrect translation: "Ich bin ein Schülerwohnheim-Verwalter."

   In this example, the model makes a mistake early on by translating "student" to "Schüler" instead of "Studenten". This is a likely mistake since depending on the training data, "student" could be translated to "Schüler" more often than "Studenten" as both are valid translations. However, with greedy decoding the model cannot recover from this mistake and is unable to instead maximise the probability of "student housing", which intuitively would be more likely to be translated to Studentenwohnheim as accomodation and housing is more prevalent in text for University students than for school students due to the socio-economic history of German school students typically living at home.

   (We made the simplifying assumption that the NMT model would be able to translate / continuate compounds correctly; this doesn't change the fact that greedy decoding would still be problematic in this case.)

2.
```
# Initialize the beam with the start symbol for each candidate
beam = [(start_symbol, 0.0, initial_state)] * beam width
# Beam width is the number of candidates to keep at each step

for t in range of max_length:
    Initialise all_candidates to empty list;
    for sequence, score, state in beam:
        # Use the decoder to generate the probability of all words at current step
        p(w_t | w_1, ..., w_{t-1}, h) = decoder(w_{t-1}, h)
        prob, new_state = decoder(sequence[-1], state)

        # For each possible next word, create a new candidate sequence
        for next_word in range of vocab_size:
            # Extend the sequence with the next word
            w_t = append(next_word to w_t)
            new_sequence = append(next_word to sequence)
            # Update the score (log probability) of the sequence
            log p(w_1, ..., w_t | h) = log p(w_1, ..., w_{t-1} | h)
                + log p(w_t | w_1, ..., w_{t-1}, h)
            new_score = score + log(prob[next_word])  # use log probabilities to avoid underflow
            append((new_sequence, new_score, new_state)) to all candidates

    # Keep only the top beam width candidates with the highest total scores
    (w_1, ..., w_t) = argmax_{w_1, ..., w_t} log p(w_1, ..., w_t | h)
    # Sort and prune the candidates according to beam width
    pruned candidates = prune(sort(all candidates), beam width)
    beam = argmax_{total score}(pruned candidates)

# Return the candidate with the highest score
(w_1, ..., w_T) = argmax_{w_1, ..., w_T} log p(w_1, ..., w_T | h)
return argmax_score(beam)
```

3. Longer sentences can be encouraged by dividing the log probability of the word sequence by the length of the sequence when calculating the new score. A hyperparameter can be used to control how much the length of the sequence affects the new score:

   ```
   new_sequence = append(next_word to sequence)
   log p(w_1, ..., w_t | h) = log p(w_1, ..., w_{t-1} | h) + log p(w_t | w_1, ..., w_{t-1}, h)
    # alpha is the hyperparameter used to control how much the length of
    # the new sequence affects the score
   new_score = score + log(prob[next_word]) / (alpha * len(new_sequence))
   ```

Question 4: Adding Layers

1.
```
python train.py --save-dir "${EXP_ROOT}" \
    --log-file "${EXP_ROOT}/log.out"  \
    --data "${DATA_DIR}" \
    --encoder-num-layers 2 \
    --decoder-num-layers 3 \
```

Table 1: Comparison of Baseline and Extra Layer

Model	**Test BLEU**	**Validation Perplexity**	**Training Loss**
Baseline	11.29	27.3	2.131
Extra Layer	9.25	29.4	2.45

2. The results of the baseline and extra layer models can be found in Table 1.

   It appears that the deeper model has worse performance than the baseline model. The lower BLEU score suggests a worse qualilty of translation, the higher validation perplexity means the new model is more uncertain with its predictions and the higher training loss suggests that the model is fitting the data worse.

   The lower BLEU score and higher validation perplexity could be caused by the model overfitting to the training data and not generalizing well to unseen data.

   The model may also be more difficult to optimize because of the increased depth as there are more parameters to train. The hyperparameters of these models may need to be tuned and regularization techniques may need to be incorporated. Without these, the model might struggle to converge. Because the model is deeper it might also need more data or a longer training time so it can converge to a good solution.

   These issues could also be caused by vanishing gradients, which can cause issues with training. This is caused by the gradient becoming too small during backpropagation, resulting in a slower convergence and higher training loss.

Question 5: Implementing the Lexical Model

```python
# __QUESTION-5: Add parts of decoder architecture corresponding to the LEXICAL MODEL here
self.source_embed = nn.Linear(self.embed_dim, self.embed_dim, bias = False)
self.lexical_output = nn.Linear(self.embed_dim, len(dictionary))
# 
    ###########################################################################################

# __QUESTION-5: Compute and collect LEXICAL MODEL context vectors here
# f^l_t = tanh(sum_s(a_t(s) f_s))
weighted_average_sum = torch.bmm(step_attn_weights.unsqueeze(1),
                                 src_embeddings.transpose(0, 1))
average_source_word_embedding = torch.tanh(weighted_average_sum.squeeze(1))

# h^l_t = tanh(W f^l_t) + f^l_t
h_lexical = torch.tanh(self.source_embed(average_source_word_embedding) +
    average_source_word_embedding)
lexical_contexts.append(h_lexical)
# 
    ###########################################################################################

# __QUESTION-5: Incorporate the LEXICAL MODEL into the prediction of target tokens here
lexical_contexts = torch.stack(lexical_contexts).transpose(0,1)
decoder_output = decoder_output + self.lexical_output(lexical_contexts)
# 
    ###########################################################################################
```

Listing 1: Lexical Model Code

Table 2: Comparison of Baseline and Lexical Models

Model	**Test BLEU**	**Validation Perplexity**	**Training Loss**
Baseline	11.29	27.3	2.131
Lexical	13.23	23.8	1.82

The results of the baseline and Lexical model can be found in Table 2.

The lexical model has a higher BLEU score which suggest higher quality translations, lower validation perplexity which suggests the model is more certain with its predictions and lower training loss which suggests it is fitting the data better. The lexical translation is beneficial to these performance metrics.

1. **Example 1**
    - **original:** herr präsident , mir geht es um den wortlaut bei einem bericht von frau randzio-plath .
    - **gold:** mr president , this is the text relating to the randzio - plath report .
    - **base:** mr president , i am surprised by the report of the committee on the report .
    - **lexical:** mr president , i am talking about the report of the report in mrs randzio - de mrs van report .
    - **observation:** the challenge at hand is to translate the rare compound surname "randzio-plath". The base model fails to maintain the semantic meaning of the sentence by completely omitting the name. The lexical model recognises the rare token and attempts a contextualised translation, but fails with the compound. Likely, "de mrs van" relates to an attempt of translating "plath". This shows that the lexical model seems to be more capable of remembering rare tokens due to having information about the context of the source sentence [2].
2. **Example 2**

- **original:** bravo , frau merkel , herzlichen glückwunsch !
- **gold:** well done , mrs merkel , warm congratulations !
- **base:** mrs gonzález , mrs van , have been done !
- **lexical:** mrs ferrero - waldner , mrs ferrero , mrs short , mrs short , mrs short , mrs short , congratulations !
- **observation:** in this example, both models fail to translate the name "merkel" correctly. Interestingly, the lexical model seems to "get stuck" while translating the name, generating a sequence of "mrs short" tokens. This suggests to us that the model does recognise rare tokens, attempts to do something about it but may not be hypertuned correctly for this translation task, allocating too much or too little surface area on the hypersphere to allocate to word embeddings [2]. The lexical model does manage to translate the "congratulations" part, which is a positive sign that the context of the source sentence was taken into account.

3. **Example 3**
    - **original:** (die sitzung wird um 13.40 uhr unterbrochen und um 15.00 uhr wiederaufgenommen .)
    - **gold:** (the sitting was suspended at 1.40 p.m. and resumed at 3 p.m.)
    - **base:** (the sitting was suspended at 11.40 p.m. and resumed at 9.00 a.m.)
    - **lexical:** (the sitting was suspended at 12.10 a.m. and resumed at 3 p.m.)
    - **observation:** the lexical model seems to be more capable of translating the time of day than the base model. In this example, the base model fails to translate both of the times whereas the lexical only gets the first time wrong.

Question 6: Understanding the Transformer Mode

```
6-A-1:
    embeddings.size = [batch_size, src_time_steps, num_features]

6-A-2:
    Positional embeddings are added to the input embeddings to provide information about the
    position of the token in the sequence.
    This helps the model to understand the order of the tokens in the sequence which is
    particularly important for an MT task since word order differs from language to language.
    Transformers do not take the order of tokens into account by default so this is a way
    to provide the model with this information.

6-B-1:
    self_attn_mask.size = [tgt_time_steps, tgt_time_steps] or None

6-B-2:
    The purpose of self_attn_mask is to prevent the decoder from attending to future tokens
    in the sequence during training.

6-B-3:
    The mask is needed in the decoder because its supposed to generate the output tokens
    sequentialy, so during inference it won't have generated future tokens so wouldn't be
    able to use them. The encoder, on the other hand, is given all the input tokens at
    once during inference, thus it has access to future tokens so a mask isn't needed during
    training.

6-B-4:
    We are only decoding one token at a time, so there are no future tokens to attend to.

6-C-1:
    The final output we want is a probability distribution over the vocabulary.
    The linear projection is needed to map the output of the decoder to the vocabulary size.

6-C-2:
    If features_only=True, the output would be the feature representation of the last
    decoder layer.

6-D:
    Transformers work with fixed-size input, so we need to pad the input to the same length.
    encoder_padding_mask is used to disregard the padding when calculating the attention weights.

6-E-1:
    Encoder attention performs attention between the encoder's input and target sequence to
    create hidden representations. It can be used to 'align' the input and target sequences
    and has access to the entire encoder output.
    Self attention is used to create hidden representations of the input sequence which can
    be used to assign 'importance' to different parts of the input. It aims to capture the
    dependencies between different words in the input and only has access up to the current
    step in the sequence.

6-E-2:
    attn_mask is used to prevent the model from seeing future tokens when training.
    key_padding_mask is used to stop the model from attending to padded tokens.
```

```
6-E-3:
    attn_mask is not needed because there are no problems with seeing the entire encoder output.
    The encoder attention can have access to the entire encoder output while it's self attention
    that is only allowed to see up to the current step in the sequence.
```

Question 7: Implementing Multi-Head Attention

Table 3: Comparison of Baseline and Multi-Head Attention

Model	**Test BLEU**	**Validation Perplexity**	**Training Loss**
Baseline	11.29	27.3	2.131
Multi-Head Attention	11.17	46	1.31

Results are shown in Table 3. The multi-head attention model has a lower training loss than the baseline which suggests it is fitting the training data better. However the test BLEU score and the validation perplexity are both worse, this suggests that the model might actually be overfitting on the training data. We also found that the model stops early at epoch 18 while the baseline stops at epoch 84, this shows that the model is converging very quickly and that it trains much faster than the other models on this training set. This could be because the model is much larger than the baseline and the training data might be too small, resulting in the model overfitting to the training data and performing badly on the validation and test data.

Increasing the size of the training set could solve some of these issues. Applying regularization techniques such as dropout, L2 regularization and layer normalization could also help in preventing overfitting[1].

It has also been found that dropping entire attention heads during training instead of just dropping connections or units can stop the model from being dominated by a small amount of the attention heads[3]. This can reduce the risk of the model overfitting and allow it to make better use of multi-head attention.

```
def forward(
    self,
    query,
    key,
    value,
    key_padding_mask=None,
    attn_mask=None,
    need_weights=True,
):

    # Get size features
    tgt_time_steps, batch_size, embed_dim = query.size()
    assert self.embed_dim == embed_dim
    """
    ___QUESTION-7-MULTIHEAD-ATTENTION-START
    Implement Multi-Head attention  according to Section 3.2.2 of https://arxiv.org/pdf
    /1706.03762.pdf.
    Note that you will have to handle edge cases for best model performance. Consider what
    behaviour should
    be expected if attn_mask or key_padding_mask are given?
    """
    # attn is the output of MultiHead(Q,K,V) in Vaswani et al. 2017
    # attn must be size [tgt_time_steps, batch_size, embed_dim]
    # attn_weights is the combined output of h parallel heads of Attention(Q,K,V) in Vaswani
     et al. 2017
    # attn_weights must be size [num_heads, batch_size, tgt_time_steps, key.size(0)]

    def __reshape_qkv_for_bmm(query, key, value):
        # Project the queries, keys and values to the multi-head space.
        q = self.q_proj(query)
        # q.shape = [tgt_time_steps, batch_size, embed_dim]
        k = self.k_proj(key)
        # k.shape = [key.size(0), batch_size, embed_dim]
        v = self.v_proj(value)
        # v.shape = [key.size(0), batch_size, embed_dim]

        # Reshape q so that it can be split into the different heads, one for each batch.
        q = q.view(
            tgt_time_steps, batch_size * self.num_heads, self.head_embed_size
        )
        # q.shape = [tgt_time_steps, batch_size * num_heads, head_embed_size]
        q = q.transpose(0, 1)  # for bmm
        # q.shape = [batch_size * num_heads, tgt_time_steps, head_embed_size]

        # Reshape k, same reasoning as for q.
        k = k.view(-1, batch_size * self.num_heads, self.head_embed_size)
        # k.shape = [key.size(0), batch_size * num_heads, head_embed_size]
        k = k.transpose(0, 1)
        # k.shape = [batch_size * num_heads, key.size(0), head_embed_size]
        k = k.transpose(1, 2)
        # k.shape = [batch_size * num_heads, head_embed_size, key.size(0)]

        # Reshape v, same reasoning.
        v = v.view(-1, batch_size * self.num_heads, self.head_embed_size)
        # v.shape = [key.size(0), batch_size * num_heads, head_embed_size]
        v = v.transpose(0, 1)
        # v.shape = [batch_size * num_heads, key.size(0), head_embed_size]
        return q, k, v

    def __calculate_scaled_attn_weights(q, k):
        # QK^T
        attn_weights_unscaled = torch.bmm(q, k)
        # attn_weights_unscaled.shape = [batch_size * num_heads, tgt_time_steps, key.size(0)
    ]

        # QK^T / sqrt(d_k)
        attn_weights = attn_weights_unscaled / self.head_scaling
        # attn_weights.shape = [batch_size * num_heads, tgt_time_steps, key.size(0)]
        return attn_weights

    def __apply_key_padding_mask(key_padding_mask, attn_weights):
        if key_padding_mask is None:
            return attn_weights
            # attn_weights.shape = [batch_size * num_heads, tgt_time_steps, key.size(0)]

        # Reshape the key_padding_mask to match the attn_weights
        attn_weights = attn_weights.view(
            batch_size, self.num_heads, tgt_time_steps, key.size(0)
        )
        # attn_weights.shape = [batch_size, num_heads, tgt_time_steps, key.size(0)]

        # Expand the key_padding_mask to match the attn_weights
        key_padding_mask = key_padding_mask.unsqueeze(1).unsqueeze(2)
        # key_padding_mask.shape = [batch_size, 1, 1, key.size(0)]
        # Apply the key_padding_mask to the attn_weights
        attn_weights = attn_weights.masked_fill(key_padding_mask, float("-inf"))
        # attn_weights.shape = [batch_size, num_heads, tgt_time_steps, key.size(0)]
        # Reshape the attn_weights back to the original shape
        attn_weights = attn_weights.view(
            batch_size * self.num_heads, tgt_time_steps, key.size(0)
        )
        # attn_weights.shape = [batch_size * num_heads, tgt_time_steps, key.size(0)]
        return attn_weights

    q, k, v = __reshape_qkv_for_bmm(query, key, value)
    # q.shape = [batch_size * num_heads, tgt_time_steps, head_embed_size]
    # k.shape = [batch_size * num_heads, head_embed_size, key.size(0)]
    # v.shape = [batch_size * num_heads, key.size(0), head_embed_size]

    attn_weights = __calculate_scaled_attn_weights(q, k)
    #attn_weights.shape = [batch_size * num_heads, tgt_time_steps, key.size(0)]

    attn_weights = __apply_key_padding_mask(key_padding_mask, attn_weights)
    # attn_weights.shape = [batch_size * num_heads, tgt_time_steps, key.size(0)]

    # Apply the attn_mask
    if attn_mask is not None:
        attn_weights += attn_mask

    # Apply the softmax function to the attn_weights
    attn_weights = F.softmax(attn_weights, dim=2)
    # attn_weights.shape = [batch_size * num_heads, tgt_time_steps, key.size(0)]

    # Calculate attn by performing bmm between the attn_weights and the values
    attn = torch.bmm(attn_weights, v)
    # attn.shape = [batch_size * num_heads, tgt_time_steps, head_embed_size]

    # Reshape attn to original shape
    attn = attn.transpose(0, 1)
    # attn.shape = [tgt_time_steps, batch_size * num_heads, head_embed_size]
    attn = attn.contiguous()
    attn = attn.view(tgt_time_steps, batch_size, embed_dim)
    # attn.shape = [tgt_time_steps, batch_size, embed_dim]

    # Reshape attn_weights acc. to num of heads
    attn_weights = attn_weights.view(
        batch_size, self.num_heads, tgt_time_steps, key.size(0)
    ) if need_weights else None
    # attn_weights.shape = [batch_size, num_heads, tgt_time_steps, key.size(0)]
    """
    ___QUESTION-7-MULTIHEAD-ATTENTION-END
    """
    return attn, attn_weights
```

Listing 2: Multi-Head Attention Code

References

[1] Antonio Valerio Miceli Barone et al. *Regularization techniques for fine-tuning in neural machine translation*. 2017. DOI: 10.48550/ARXIV.1707.09920. URL: https://arxiv.org/abs/1707.09920.

[2] Toan Q. Nguyen and David Chiang. *Improving Lexical Choice in Neural Machine Translation*. Apr. 2018. arXiv: 1710.01329 [cs]. (Visited on 03/19/2024).

[3] Wangchunshu Zhou et al. *Scheduled DropHead: A Regularization Method for Transformer Models*. 2020. DOI: 10.48550/ARXIV.2004.13342. URL: https://arxiv.org/abs/2004.13342.