KU Leuven - NLP 2024-2025
18 December 2024

Exercise Session 5 - solutions
Machine Translation

For questions contact:
Toledo Forum for Questions or
nlp@ls.kuleuven.be

1 Design a Translation Model

1.A: Designing the model

1. First we need to update the target vocabulary.

   We need to add a couple of tokens: the SOS token, the EOS token, and an UNK token.
   We need the SOS token, to indicate to the decoder that it has to start predicting a new sentence.
   When the model predicts the EOS token, it stops predicting following tokens. So this token is needed for the model to learn the length of sentences.
   Our vocabulary is small. It won't contain all the words for the language. So with the UNK token the model will not generate weird words when it has to generate something it hasn't seen before.

2. Write down all the equations in the model.

   We will define multiple letters here:

   - $\mathbb{X}$ is the sequence of input words.
   - $\boldsymbol{h}$ are the hidden states.
   - $\boldsymbol{c}$ are the cell states of the LSTM.
   - forward and backward arrows represent the forward and backward LSTM.
   - we use $[\cdot;\cdot]$ to concatenate vectors.
   - $\boldsymbol{e}$ is the embedding of a token (source or target).
   - Emb is the function to get the embedding for a token.
   - a_i is the attention score for source state $\boldsymbol{h}^{(i)}$.
   - α_i is the attention weight (between 0 and 1) for the source state $\boldsymbol{h}^{(i)}$.
   - $\boldsymbol{o}$ is the output scores vector over the vocabulary. Note that in our design, for predicting the output we have a two layer network, so two weight matrices and two biases. Sometimes you can choose to not use biases in the output layers of your model.
   - It is important that you avoid non-linearities after your last layer. Some can break the model, such as a ReLU that sets many of the outputs to zero.

   Note that we have to make multiple design choices, for instance, on the similarity function in the

attention mechanism. I used a linear layer, but this could have been a simple dot product.

$$
\begin{aligned}
\mathbb{X} &= \{x_1, \ldots, x_N\} \\
\overrightarrow{\boldsymbol{h}}_{enc}^{(0)} &= \overrightarrow{\mathbf{0}} \\
\overrightarrow{\boldsymbol{c}}_{enc}^{(0)} &= \overrightarrow{\mathbf{0}} \\
\overleftarrow{\boldsymbol{h}}_{enc}^{(N+1)} &= \overrightarrow{\mathbf{0}} \\
\overleftarrow{\boldsymbol{c}}_{enc}^{(N+1)} &= \overrightarrow{\mathbf{0}} \\
\boldsymbol{e}_s^{(i)} &= Emb_s(x_s^{(i)}) \\
\overrightarrow{\boldsymbol{h}}_{enc}^{(i)}, \overrightarrow{\boldsymbol{c}}_{enc}^{(i)} &= \overrightarrow{LSTM}_{enc}(\boldsymbol{e}_s^{(i)}, \overrightarrow{\boldsymbol{h}}_{enc}^{(i-1)}, \overrightarrow{\boldsymbol{c}}_{enc}^{(i-1)}) \\
\overleftarrow{\boldsymbol{h}}_{enc}^{(i)}, \overleftarrow{\boldsymbol{c}}_{enc}^{(i)} &= \overleftarrow{LSTM}_{enc}(\boldsymbol{e}_s^{(i)}, \overleftarrow{\boldsymbol{h}}_{enc}^{(i+1)}, \overleftarrow{\boldsymbol{c}}_{enc,b}^{(i+1)}) \\
\boldsymbol{h}_{enc}^{(i)} &= [\overrightarrow{\boldsymbol{h}}_{enc}^{(i)}; \overleftarrow{\boldsymbol{h}}_{enc}^{(i)}] \\
\boldsymbol{c}_{enc}^{(i)} &= [\overrightarrow{\boldsymbol{c}}_{enc}^{(i)}; \overleftarrow{\boldsymbol{c}}_{enc}^{(i)}] \\
\boldsymbol{h}_{dec}^{(0)} &= [\overrightarrow{\boldsymbol{h}}_{enc}^{(N)}; \overleftarrow{\boldsymbol{h}}_{enc}^{(1)}] \\
\boldsymbol{c}_{dec}^{(0)} &= [\overrightarrow{\boldsymbol{c}}_{enc}^{(N)}; \overleftarrow{\boldsymbol{c}}_{enc}^{(1)}] \\
\boldsymbol{e}_t^{(0)} &= Emb_t(SOS) \\
\boldsymbol{e}_t^{(t)} &= Emb_t(\arg\max(\hat{\boldsymbol{y}^j})) \\
a_i^{(j)} &= \boldsymbol{W}_a^T[\boldsymbol{h}_{enc}^{(i)}; \boldsymbol{h}_{dec}^{(j-1)}] + \boldsymbol{b}_a \\
\alpha_i^{(j)} &= \frac{\exp(a_i)}{\sum_{k=1}^{N} \exp(a_k)} \\
\boldsymbol{s}^{(t)} &= \sum_{i=1}^{N} \alpha_i^{(j)} \boldsymbol{h}_{enc}^{(i)} \\
\boldsymbol{h}_{dec}^{(j)}, \boldsymbol{c}_{dec}^{(j)} &= LSTM_{dec}([\boldsymbol{e}_t^{(j-1)}; \boldsymbol{s}^{(j)}], \boldsymbol{h}_{dec}^{(j-1)}, \boldsymbol{c}_{dec}^{(j-1)}) \\
\boldsymbol{o}^{(j)} &= \boldsymbol{W}_2^T \tanh(\boldsymbol{W}_1^T \boldsymbol{h}_{dec^{(j)}} + \boldsymbol{b}_1) + \boldsymbol{b}_2 \\
P(\hat{\boldsymbol{y}^j} = *) &= \mathrm{softmax}(\boldsymbol{o}^{(j)})
\end{aligned}
$$

3. How many weights matrices and bias vectors are there?

We have two embedding matrices, for the source and the target language.
Each LSTM contains 8 weight matrices (4 gates with each a matrix for the input and a matrix for the previous hidden state) and 4 bias vectors, so 24 weight matrices and 12 bias vectors in total.
In our simple attention model we have 1 matrix and 1 bias vector.
Our decoder has 2 weight matrices and 2 bias vectors.
In total we have 2+24+1+2=29 matrices and 12+1+2=15 vectors.

4. What are the sizes of the matrices and the biases, with hidden state a dim of 128?

for the forward and backward LSTM they are 64x64. Except the input matrices, which is 300x64. The biases are all 64.
For the attention the matrix is 256x1 and the bias is 1, since we want a scalar value indicating each weight.
For the decoder, the matrices are 128x128. except for the input, which is 428x128. The biases are all 128.
For the output matrices, $\boldsymbol{W}_1$ is 128x128, and $\boldsymbol{b}_1$ is 128. $\boldsymbol{W}_2$ is 128x3003. Note that it is a design choice if you want to have the first layer mapping to 128 or if you want to change the dimension.

1.B: Design mistake! We need live translation!

1. The first step is to make sure we can do live translations. What do we have to change about the model to make this possible?

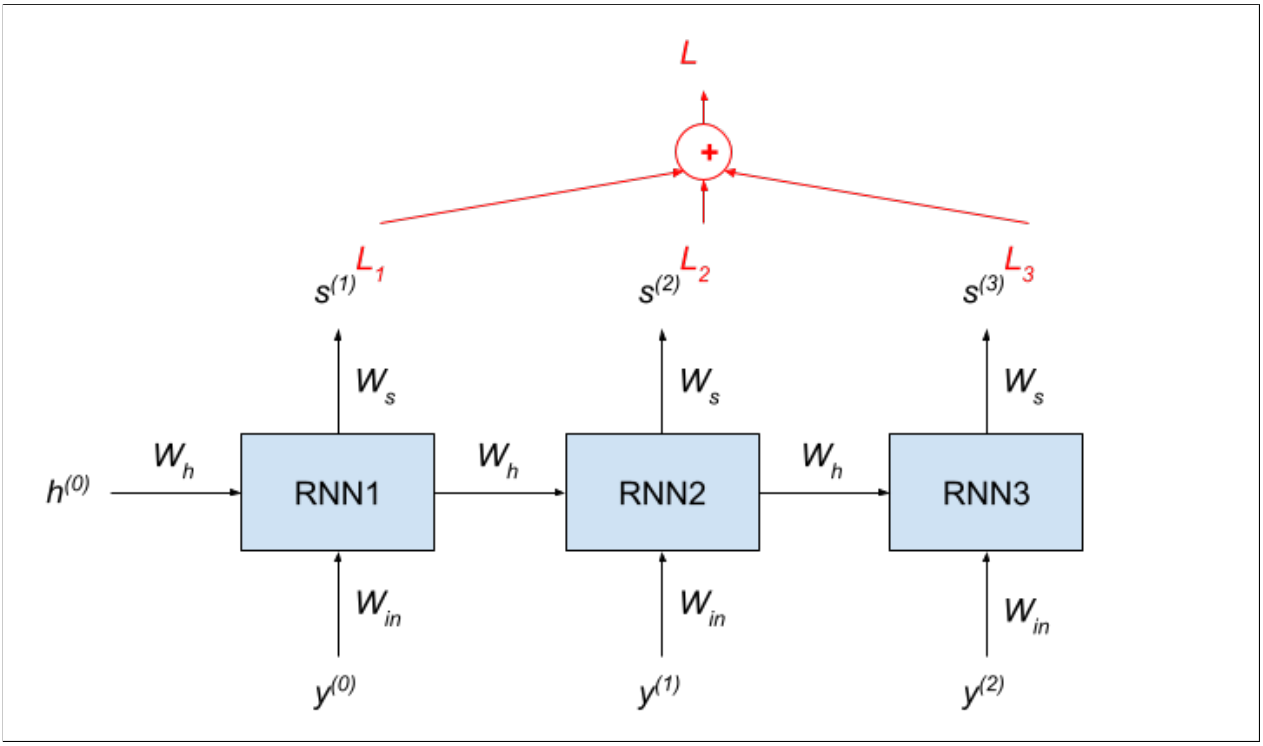
   With live-translation, we cannot process the backwards pass. We need to use only a forward LSTM. Plus, at every timestep (or several timesteps) we have to run the decoder.
   Of course, this will cause some strange orders in the translations, but with long monologues it is important to get small parts of the translation at once so it is easier to follow.
   There are many decisions to make here, in terms of quality and speed.

2. The second step is to make it sure we can translate to multiple languages.

   We can use a single encoder for the source language. For the decoder, we need a decoder for each language. This takes as input the output embeddings of the encoder.
   To train this properly, we can do two things. Either pretrain the encoder, and keep its weights fixed. Then you can train the decoders separately.
   The second option is to train everything together. To do this, you iterate over the decoders and train one batch at a time per decoder. If you first train the encoder with one decoder, and then move on to the next decoder, it can cause catastrophic forgetting on the first language. By iterating over all the decoders, this is less likely to happen.

2 Training the RNN Decoder

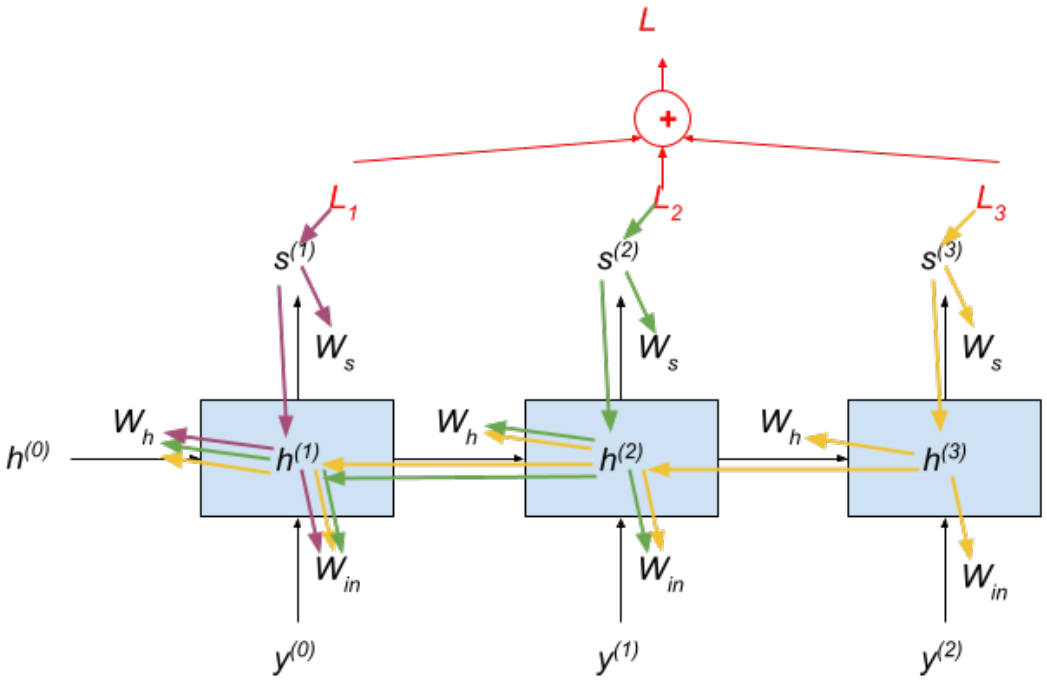
2.A Draw the model



2.B Back propagation through time.

Use the drawing to clearly indicate by which computations each weight matrix is influenced.

At each timestep, I used a different color to indicate how the gradient of the loss is passed through the layers, back through time.



Let's go over the drawing on how to interpret the flow of back-propagation through the network.
!Disclaimer!: I will make use of some gradients and math to explain stuff. You don't have to understand and know those details. We want you to see and understand how the gradient flows and where vanishing and exploding gradients come from.
To update a parameter (a weight matrix or a bias vector), we have to start at the loss, and propagate that error through the network, by doing something like this:

$$p = p - \nabla_p L$$

So imagine every green/purple/yellow colored arrow as a gradient. We start in the top right, where we can compute the gradient $\nabla_{s^{(3)}} L_3$. We want to update the W_s matrix with regards to L_3. So, now we can compute the gradient $\nabla_{W_s} s^{(3)}$.
We combine this to do the update

$$\nabla_{W_s} L_3 = \nabla_{s^{(3)}} L_3 \cdot \nabla_{W_s} s^{(3)}$$

Note that for W_s we also have to find the gradients with regards to L_2 and L_3. Thus we get the following.

$$\nabla_{W_s} L = \nabla_{W_s} L_3 + \nabla_{W_s} L_2 + \nabla_{W_s} L_1$$

In this case, W_s is only the final layer of the model, and we only did one back-propagation step. For the other weights, we have to do way more steps. Let's do one more example where we follow the yellow arrows further towards W_h of the final RNN timestep: $\nabla_{W_h} L_3$.
We already have $\nabla_{s^{(3)}} L_3$. Now let's continue this towards $h^{(3)}$. For this arrow between $h^{(3)}$ and $s^{(3)}$, we have: $\nabla_{h^{(3)}} s^{(3)}$. And next between W_h and $h^{(3)}$ we have $\nabla_{W_h} h^{(3)}$. So to combine everything together, the complete update requires already a long equation:

$$\nabla_{W_{h^{(3)}}} L_3 = \nabla_{s^{(3)}} L_3 \cdot \nabla_{h^{(3)}} s^{(3)} \cdot \nabla_{W_h} h^{(3)}$$

This is only a small part of the back-propagation, and we did not even go into previous timesteps of the RNN. As you can see, we have to combine, multiply and add many gradients together. Especially the

gradients for matrices in the RNN will combine many gradients. If many of these terms are smaller than 1 we get the vanishing gradient problem. With many terms bigger than 1 we get the exploding gradient problem.

Extra BPTT resources

There are multiple tutorials online you can find that go into great detail explaining BPTT. Some examples are:

- There is a good explanation in the deep learning book, chapter 10.2. https://www.deeplearningbook.org/contents/rnn.html.
- A short video of the 3blue1brown channel (has many good short explanations) describing the very basic of backpropagation for a single neuron 1 layer network: https://youtu.be/tIeHLnjs5U8.
- A very brief tutorial describing the basics of the chain rule in a RNN model. https://youtu.be/M_p_fVJJhiY
- A longer more difficult tutorial, looking also in-depth into the actual derivations. Note that this goes far beyond the scope and complexity of this course. https://youtu.be/RrB605Mbpic

3 Decoding: Greedy vs Beam Search

First we convert the probabilities into log-likelihoods:

$$\begin{aligned} logP(y_1 = A|h, y_0 = SOS) &= -0.51 \\ logP(y_1 = B|h, y_0 = SOS) &= -0.91 \\ logP(y_2 = A|h, y_0 = SOS, y_1 = A) &= -0.67 \\ logP(y_2 = B|h, y_0 = SOS, y_1 = A) &= -0.71 \\ logP(y_2 = A|h, y_0 = SOS, y_1 = B) &= -0.22 \\ logP(y_2 = B|h, y_0 = SOS, y_1 = B) &= -1.61 \end{aligned}$$

3.A

We select the maximum log-likelihood for timestep 1, which results in $y_1 = A$.
Given timestep 1, we can select the maximum log-likelihood of timestep 2: $y_2 = A$ The log-likelihood of the sequence now is:

$$logP(y_1 = A|h, y_0 = SOS) + logP(y_2 = A|h, y_0 = SOS, y_1 = A) = -0.51 + -0.67 = -1.18$$

3.B

Here we have to keep track of our hypotheses. For timestep 1, there are only two possible hypotheses:

- **hypothesis 1:** $logP(y_1 = A) = -0.51$
- **hypothesis 2:** $logP(y_1 = B) = -0.91$

Now for timestep two, we expand both hypotheses with all possible options:

- **hypothesis 1.1:** $logP(y_1 = A, y_2 = A) = -0.51 + -0.67 = -1.18$
- **hypothesis 1.2:** $logP(y_1 = A, y_2 = B) = -0.51 + -0.71 = -1.22$
- **hypothesis 2.1:** $logP(y_1 = B, y_2 = A) = -0.91 + -0.22 = -1.14$

- **hypothesis 2.2:** $logP(y_1 = B, y_2 = B) = -0.91 + -1.61 = -2.53$

We are doing a beam search of size two, so we keep the two best hypotheses:

- **hypothesis 1:** $logP(y_1 = B, y_2 = A) = -0.91 + -0.22 = -1.14$
- **hypothesis 2:** $logP(y_1 = A, y_2 = A) = -0.51 + -0.67 = -1.18$

Now we see that the best prediction is the sequence $[B, A]$, which is correct. This was not the case with greedy search.

3.C

When we use the probabilities, we keep multiplying fractions. They get increasingly small and will become very difficult to compare. In computers, we might even get float errors. With log-likelihoods we will do additions. These are much easier to compute and compare.

3.D

First, we would have to add an EOS token. Once the model predicts this as the next token, we will not expand that hypothesis anymore. A downside is, that the log-likelihood will remain bigger because nothing is added to it anymore. To avoid this, we can divide the log-likelihoods by the sequence length. This way, the model can still predict longer sentences but also decide to have shorter ones.

5

1. Syntactic evaluation (METEOR)

$$METEOR = F_{score}(1-p)$$
$$F_{score} = \frac{10PR}{R+9P} = \frac{10(m/c)(m/r)}{(m/r)+9(m/c)}; p = 0.5\left(\frac{q}{z}\right)^3$$

where m is the number of words shared by both the translation and the target reference, c is the number of words in the translation, r is the number of words in the target reference, q is the number of chunks, z is the number of words that have been mapped.

C_1 :

$$P = \frac{5}{8} = 0.63; R = \frac{5}{6} = 0.83; F_{score} = \frac{10 * 0.63 * 0.83}{0.83 + 9 * 0.63} = 0.8$$
$$p = 0.5 * \left(\frac{2}{5}\right)^3 = 0.03; METEOR_{H1} = 0.80 * (1 - 0.3) = \mathbf{0.78}$$

C_2 :

$$P = \frac{5}{6} = 0.83; R = \frac{5}{6} = 0.83; F_{score} = \frac{10 * 0.83 * 0.83}{0.83 + 9 * 0.83} = 0.83$$
$$p = 0.5 * \left(\frac{2}{5}\right)^3 = 0.03; METEOR_{H2} = 0.83 * (1 - 0.3) = \mathbf{0.81}$$

2. Semantic evaluation (BERTScore)

$$BERT_{score} = \frac{2P_{BERT}R_{BERT}}{P_{BERT} + R_{BERT}}$$

R_{BERT} matches the BERT representation of each word of the reference y with the BERT representation of the most similar word of the candidate translation $\hat{y}$. The similarity scores are computed using cosine similarity. Each similarity score of the word y_i is adjusted with its importance weight (idf).

$$R_{BERT} = \frac{\sum_{y_i \in y} idf(y_i) \max_{\hat{y_j} \in \hat{y}} cos_{sim}(y_i, \hat{y_j})}{\sum_{y_i \in y} idf(y_i)}$$

Similarly, P_{BERT} matches the BERT representation of each word of the translation $\hat{y}$ with the BERT representation of the most similar word of the reference y.

$$P_{BERT} = \frac{\sum_{\hat{y_i} \in \hat{y}} idf(\hat{y_i}) \max_{y_j \in y} cos_{sim}(y_i, \hat{y_j})}{\sum_{\hat{y_i} \in \hat{y}} idf(\hat{y_i})}$$

We mark with red the most similar word of one translation to each word of the reference. Green marks the most similar word of the reference to each word of the translation.

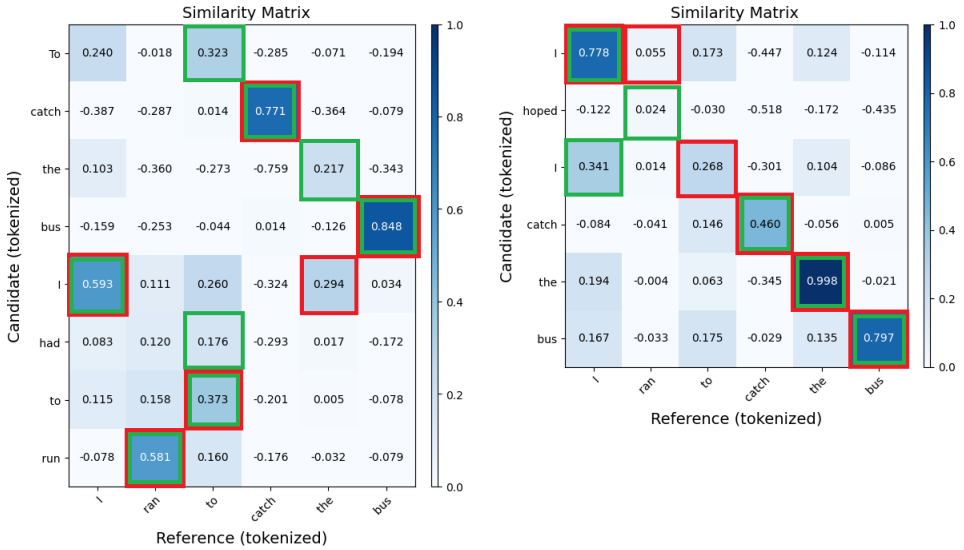


Figure 1: Similarity matrices computed between the reference and hypotheses using BERT embeddings. Red/green represents the most similar word of the translation/reference to each word of the reference/-translation.

C_1 :

$$R_{BERT} = \frac{0.593 * 1.71 + 0.581 * 3.23 + ... + 0.294 * 1.18 + 0.848 * 2.93}{1.71 + 3.23 + 1.52 + 3.59 + 1.18 + 2.93} = 0.64;$$

$$P_{BERT} = \frac{0.323 * 1.52 + 0.771 * 3.59 + ... + 0.373 * 1.52 + 0.581 * 3.19}{1.52 + 3.59 + 1.18 + 2.93 + 1.71 + 1.97 + 1.52 + 3.19} = 0.56;$$

$$BERTScore = \frac{2 * 0.64 * 0.56}{0.64 + 0.56} = \mathbf{0.60}$$

C_2 :

$$R_{BERT} = \frac{0.778 * 1.71 + ... + 0.797 * 2.93}{1.71 + ... + 2.93} = 0.50;$$
$$P_{BERT} = \frac{0.778 * 1.71 + ... + 0.797 * 2.93}{1.71 + ... + 2.93} = 0.48;$$
$$BERTScore = \frac{2 * 0.50 * 0.48}{0.50 + 0.48} = \mathbf{0.49}$$

3. Semantic evaluation (COMET)

$$COMET = \frac{1}{1 + f(x, y, \hat{y})}; f(x, y, \hat{y}) = \frac{2 * E(x, \hat{y}) * E(y, \hat{y})}{E(x, \hat{y}) + E(y, \hat{y})}$$

where $E()$ represents the Euclidean Distance, x is the input sentence, y is the reference and $\hat{y}$ is the translation.

C_1 :

$$E(x, \hat{y}) = \sqrt{(3.01 - 2.84)^2 + ... + (3.55 - 3.79)^2} = 0.38$$
$$E(y, \hat{y}) = \sqrt{(2.93 - 2.84)^2 + ... + (3.74 - 3.79)^2} = 0.58$$
$$f(x, y, \hat{y}) = \frac{2 * 0.38 * 0.58}{0.38 + 0.58} = 0.46; COMET = \frac{1}{1 + 0.46} = \mathbf{0.68}$$

C_2 :

$$E(x, \hat{y}) = \sqrt{(3.01 - 3.45)^2 + ... + (3.55 - 2.93)^2} = 1.12$$
$$E(y, \hat{y}) = \sqrt{(2.93 - 3.45)^2 + ... + (3.74 - 2.93)^2} = 1.28$$
$$f(x, y, \hat{y}) = \frac{2 * 1.12 * 1.28}{1.12 + 1.28} = 1.19; COMET = \frac{1}{1 + 1.19} = \mathbf{0.46}$$

The above evaluation proves the need to evaluate an NMT model both semantically and syntactically. While the meaning of reference is captured only by translation C_1, METEOR penalizes this translation for the syntactic shift. However, both BERTScore and COMET show the translation C_1 is semantically closer to the reference than the translation C_2.