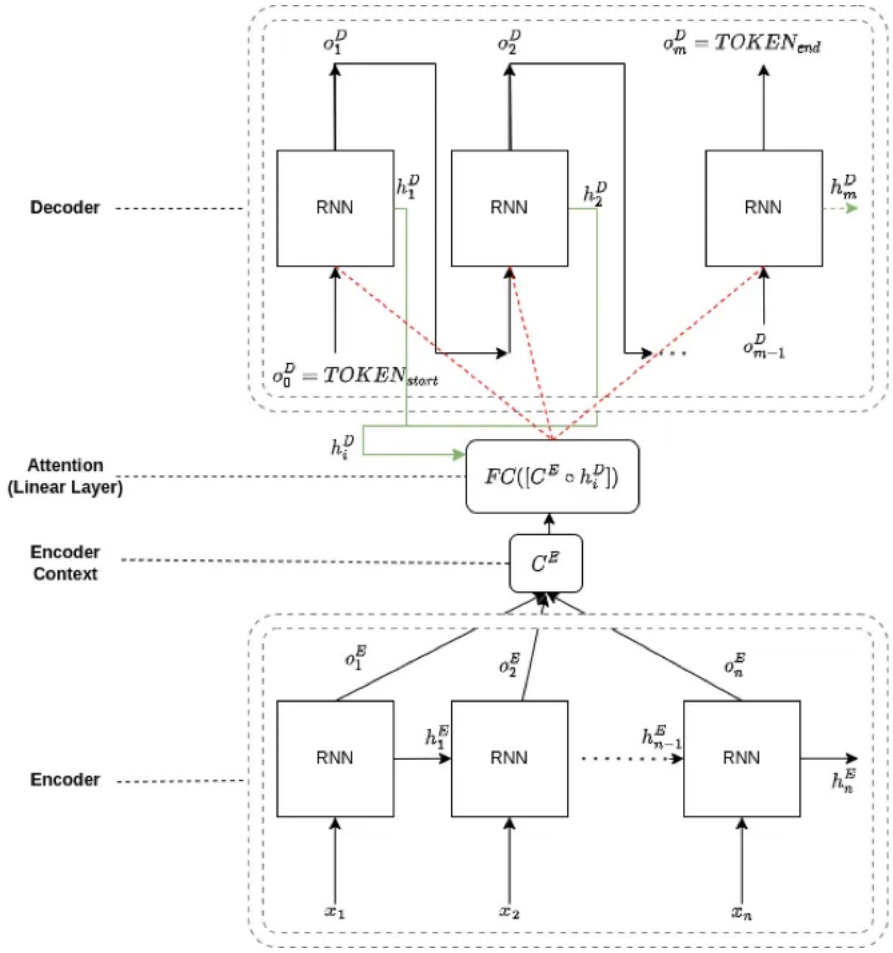
Author: Jiayi Zhu, 23tw34@queensu.ca

1. Model

RNN-Seq2Seq-Attention is an RNN-based Seq2Seq architecture with an attention mechanism, first proposed in the paper *Neural Machine Translation by Jointly Learning to Align and Translate*. For the attention mechanism, I used **Bahdanau Attention**, which is named after the first author of that paper.

I used GRU as the recurrent unit due to its greater computational efficiency. In the encoder, I used **bidirectional GRU** to captures both past and future context, enabling a better understanding of the input sentence.

The model basically consists of:



Encoder (Bidirectional GRU):

Processes input sequence step by step and encodes it into a fixed-size context vector (hidden state).

Attention Mechanism (Bahdanau):

Allows the decoder to focus on relevant parts of the input at each time step by computing attention scores to dynamically weigh the encoder's hidden states.

Decoder (GRU):

Generates the output sequence one token at a time, using the attention-weighted context vector to improve predictions.

Seq2Seq:

Integrates the encoder, attention, and decoder into one complete model.

Since there is no built-in RNN-Seq2Seq-Attention model in the library, I implemented one from scratch. The model structure and logic are described in detail within the code comments.

```
# Encoder (bidirectional GRU)
class Encoder(nn.Module):
    def __init__(self, input_dim, emb_dim, hidden_dim, dropout):
        super(Encoder, self).__init__()
        self.embedding = nn.Embedding(input_dim, emb_dim)  # word embedding
        self.gru = nn.GRU(emb_dim, hidden_dim, bidirectional=True, batch_first=True)  # bidirectional GRU
        self.fc = nn.Linear(hidden_dim * 2, hidden_dim)  # reduce bidirectional hidden states to one
        self.dropout = nn.Dropout(dropout)

    def forward(self, src):
        embedded = self.dropout(self.embedding(src))  # word embedding + dropout
        outputs, hidden = self.gru(embedded)  # GRU calculate output and hidden state
        #concatenate last forward and backward hidden states of bidirectional GRU then linear and tanh
        hidden = torch.tanh(self.fc(torch.cat((hidden[-2, :, :], hidden[-1, :, :]), dim=1)))
        hidden = hidden.unsqueeze(0)  # adapt to the shape of decoder
        return outputs, hidden  # output is dim*2 for attention, hidden is dim*1 for decoder
```

```
# Attention (Bahdanau Attention)
class Attention(nn.Module):
    def __init__(self, hidden_dim):
        super(Attention, self).__init__()
        self.attn = nn.Linear(hidden_dim * 3, hidden_dim) #reduce to hidden_dim*1
        self.v = nn.Linear(hidden_dim, 1, bias=False) # reduce hidden_dim vector to single score

    def forward(self, hidden, encoder_outputs):
        batch_size = encoder_outputs.shape[0]
        src_len = encoder_outputs.shape[1]
        hidden = hidden.permute(1, 0, 2)  # reorder to (batch_size, 1, hidden_dim)
        hidden = hidden.repeat(1, src_len, 1)  #repeat decoder hidden state across the time steps
        # concatenate decoder hidden state with encoder output along time steps
        # through linear layer and tanh to compute attention score
        energy = torch.tanh(self.attn(torch.cat((hidden, encoder_outputs), dim=2)))
        attention = self.v(energy).squeeze(2)  # squeeze the hidden_dim
        return F.softmax(attention, dim=1)  # attention weights after softmax
```

```python
# Decoder (GRU)
class Decoder(nn.Module):
    def __init__(self, output_dim, emb_dim, hidden_dim, dropout):
        super(Decoder, self).__init__()
        self.embedding = nn.Embedding(output_dim, emb_dim) # word embedding
        self.gru = nn.GRU(hidden_dim + emb_dim, hidden_dim, batch_first=True) # input is hidden+embedding
        self.fc_out = nn.Linear(hidden_dim * 2 + emb_dim, output_dim)  # reduce context+hidden+embedding
        self.attention = Attention(hidden_dim)
        self.dropout = nn.Dropout(dropout)

    def forward(self, input, hidden, encoder_outputs):
        input = input.unsqueeze(1)  # unsqueeze to (batch_size, 1)
        embedded = self.dropout(self.embedding(input))  # embedding layer and dropout
        attn_weights = self.attention(hidden, encoder_outputs).unsqueeze(1)  # calculate attention weights
        context = torch.bmm(attn_weights, encoder_outputs)  # Weighted sum of encoder outputs
        context = context[:, :, :hidden_dim]  # keep the first hidden_dim
        rnn_input = torch.cat((embedded, context), dim=2)  # concatenate embedded input and context vector
        output, hidden = self.gru(rnn_input, hidden)  # through GRU to get new hidden state
        # combine output, context, and embedded input, then pass through the output layer
        output = self.fc_out(torch.cat((output.squeeze(1), context.squeeze(1), embedded.squeeze(1)), dim=1))
        return output, hidden
```

```python
# Seq2Seq (connecting encoder, decoder, attention)
class Seq2Seq(nn.Module):
    def __init__(self, encoder, decoder, device):
        super(Seq2Seq, self).__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.device = device

    def forward(self, src, trg=None, teacher_forcing_ratio=0.5):
        batch_size = src.shape[0]
        encoder_outputs, hidden = self.encoder(src)
        trg_len = trg.shape[1] if trg is not None else 50 # target_len for training, 50 for inference
        trg_vocab_size = self.decoder.embedding.num_embeddings
        outputs = torch.zeros(batch_size, trg_len, trg_vocab_size).to(self.device)

        if trg is None: # inference mode: trg=None, output step by step**
            input = torch.tensor([2] * batch_size).to(self.device)  # start with <BOS> token
            for t in range(trg_len):
                output, hidden = self.decoder(input, hidden, encoder_outputs) # decode one step
                outputs[:, t, :] = output # store output
                top1 = output.argmax(1)  # get token with biggest propability
                input = top1  # predicted token as next input
                if (top1 == sp_model.eos_id()).all():  # end if <EOS> token
                    break
        else: # train mode: use target or predicted for input according to teaching ratio
            input = trg[:, 0] # first input is <BOS> token from target
            for t in range(1, trg_len):
                output, hidden = self.decoder(input, hidden, encoder_outputs) # decode one step
                outputs[:, t, :] = output # store output
                teacher_force = torch.rand(1).item() < teacher_forcing_ratio # decide teaching
                top1 = output.argmax(1) # predicted token
                input = trg[:, t] if teacher_force else top1 # choose next input
        return outputs
```

2. Settings

BPE Tokenizer

Since English and French are relatively similar language and share some common tokens, **Vocab Size** is set to 30,000 for the two languages combined.

The advantages of sharing tokenizer are, first, that the subwords of similar words will also have similar representations, enabling the model to learn more general patterns and improve cross-language consistency. Second, it allows the embedding layer to be shared, reducing the number of model parameters and improving computational efficiency.

Then I trained a BPE model and tokenized training, validation and test dataset. The training and validation dataset were derived from Wikipedia EN-FR corpus. The first 80% of the dataset was used for training, while the rest for validation. The test dataset is the WMT14 EN-FR test set, which was used to calculate BLUE scores.

Model Parameters

embedding_dim: dimension number of each token, usually 128 to 512 for NLP tasks, here I set 256.

hidden_dim: size of hidden state in encoder and decoder, usually 256 to 1024 for machine translation, here I set 512.

Parameter number of the model is estimated as follows:

Encoder — Embedding 30,000 x 256 + GRU 3 x (256 + 512) x 512 x 2 ≈ 10M

Decoder — Embedding 30,000 x 256 + GRU 3 x (256 + 512) x 512 + FC 30,000 x (512 x 2 + 256) ≈ 50M

Attention — 3 x 512 x 512 ≈ 1M

Therefore, the RNN-Seq2Seq-Attention model approximately has 60M parameters. When I print *sum(p.numel() for p in model.parameters())*, it shows 59.4M, as estimated.

3. Training

I trained the model on a Google Colab L4 GPU. Below are some of the training strategies and hyperparameters.

Adam Optimizer: very suitable for RNN and Transformer based models

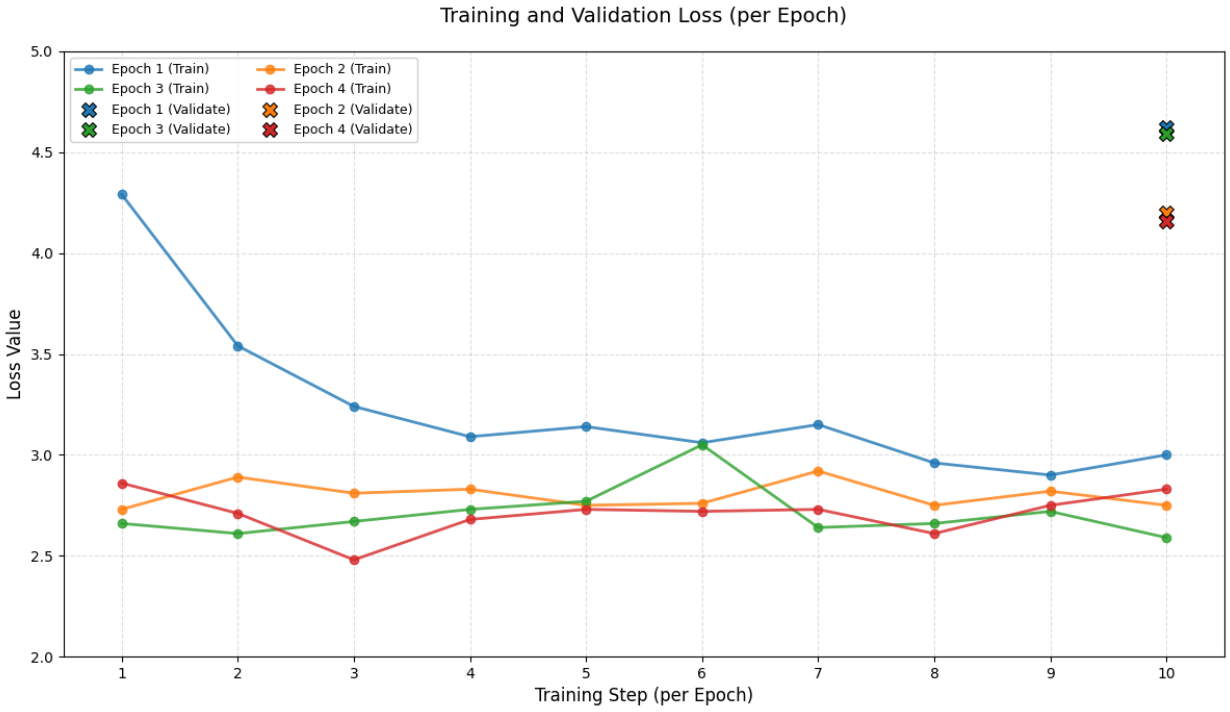
CrossEntropyLoss: very suitable for classification tasks like NLP

Batch Size: closely related to computational resources. Based on my experiments on Colab L4 GPU, **64** is the batch size that gives the most efficient training.

Dropout: Since our dataset is quite big, overfitting is less of a concern and low dropout helps the model converge faster. Here I set **0.3**.

Training Strategy: After each epoch, I evaluated on validation dataset to see the loss trend and tested the model on WMT14-test dataset to calculate BLEU score. During testing, I also printed a specific "source - target - prediction" sentence pair to have a direct idea of how well the model translates. Since I'm only familiar with English, I set French as source language and English as target language, so that I can evaluate the translation quality. After refining the model based on my observations, I train an English to French translator using the improved model eventually.

With all parameters set up, I started training. In the first attempt, I trained on 100% of the training dataset at each epoch and found that the BLEU score dropped dramatically after three epochs. I suspect there might be overfitting problem, where the model started to memorize the training dataset and lost general understanding of the language. Instead of adding the dropout rate, I chose another solution — **training less dataset at each epoch**. To be specific, I used 100% of the training dataset for epoch1, then 60%, 40%, and 20% for epoch 2, 3, 4 respectively. The data was randomly chosen from the dataset according to these proportions. The biggest advantage of this strategy in prevent overfitting is that it can save a huge amount of training time, as training a full epoch takes 2 hours.



The loss trend is illustrated in the picture above. The dots are training loss, while the markers are validation loss after each epoch. As shown, the loss decreases close to the

convergence after just one epoch. During epoch 2-4, the loss decreases slowly, and in some cases, the loss is higher than that in a previous epoch, showing signs of overfitting.

The BLEU score for the entire test dataset, along with the sample sentence pairs after each epoch, are presented below.

Source sentence: *Ce nouveau raid de l'aviation de Tsahal en territoire syrien (le sixième depuis le début de l'année, selon le quotidien israélien Haaretz) n'a été confirmé ni par Isral ni par la Syrie*.

Reference sentence: *This latest raid by the Israeli air force in Syrian territory (the sixth since the start of the year, according to the Israeli daily newspaper, Haaretz) has been confirmed neither by Israel nor Syria*.

Epoch 1:

Prediction sentence: *'s new of of theahalah territory territory of the territory of the the the the the the the the Ha Israeli Hazaret) was was confirmed by Israel Israel by Israel Syria*.

Test BLEU Score: 4.73

Epoch 2:

Prediction sentence: *the new of the theahal territory in the Syrian of the the the the the the, the Israeli Union, the the Israeli Union, the Israel Defense, the Israel Defense was the confirmed by Israel Syria*.

Test BLEU Score: 5.08

Epoch 3:

Prediction sentence: *new new air of Tsahal territory in the Syrian territory (the sixth of the the year, according to the Israeli Ha Haaretz, the the the Israel by the Israel Syria. Israel*.

Test BLEU Score: 5.61

Epoch 4:

Prediction sentence: *'s new regime of the Syrianal territoryal territory, the Syrian the the the the the Israeli,, Israeli, Israeli, Israeli, Israeli, was the confirmed by Israel Syria*.

Test BLEU Score: 5.53

From my perspective, the most significant mistake the model makes is the repeated use of “the”, which will be analyzed in the next chapter.

4. Visualizing Attention Weights

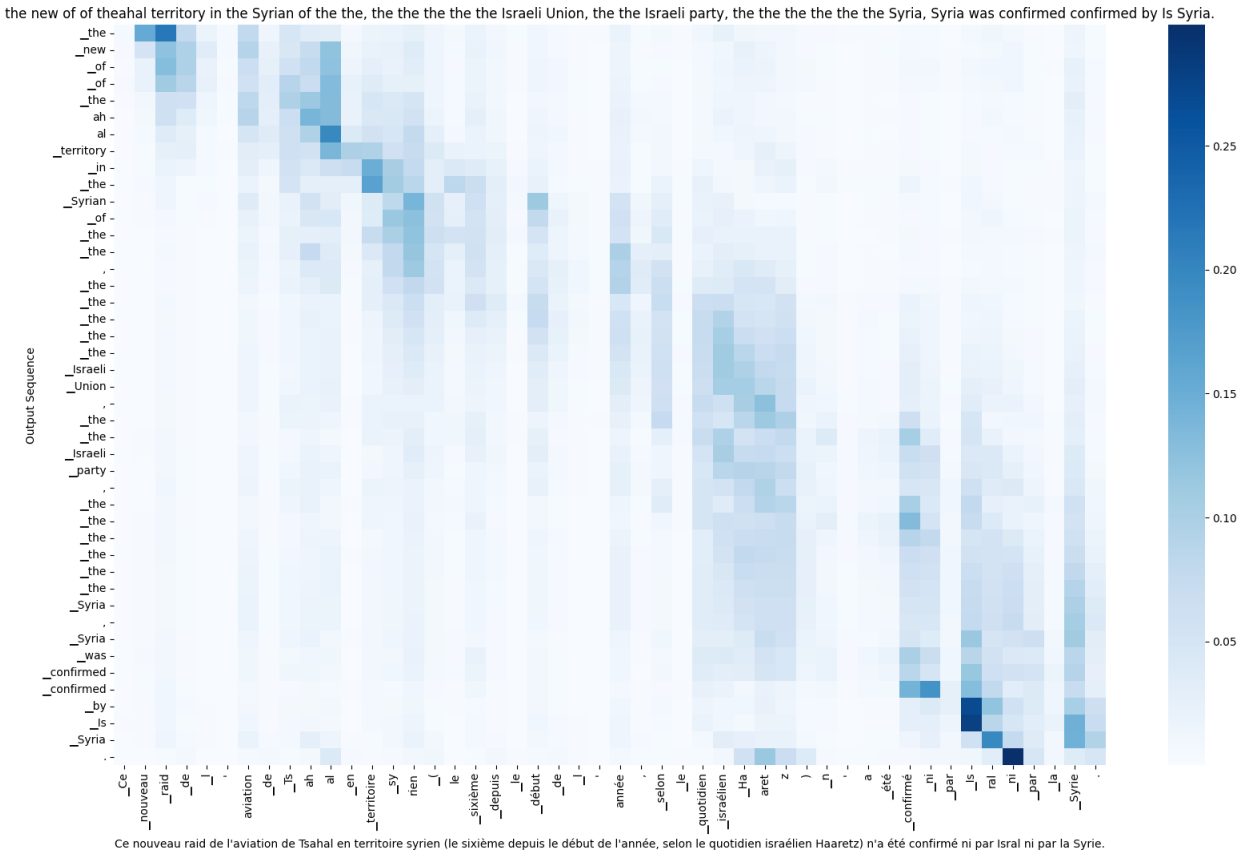
The most common approach to visualizing attention is to plot a heatmap of the attention weight matrix. This matrix shows the correlation between each target word (in the translation) and the source language words. Each row corresponds to a word in the target sentence, and each column corresponds to a word in the source sentence. The values in the matrix represent the attention weights. By examining the heatmap, we can clearly see which source words the model focused on when generating each target word.

There are three steps to achieve heatmaps:

First, modify the model so that attention matrix can be read out. In the decoder, return **attn_weights** along with decoder output and hidden layer. In the Seq2Seq inference mode, store **attn_weight_matrix** at each time step and return it.

Secondly, extract the attention weights. Input a source sentence, and after the translation is complete, extract the attention weights between target and source words. For each target word, the model produces a set of weights corresponding to the source words.

Finally, use Matplotlib to generate the heatmap. Each cell in the heatmap represents the attention strength between a target word and a source word.



Here is the heatmap of the model's predicted translation for the selected sentence, with the prediction sentence at the top and the source sentence at the bottom. The following are the key observations from the heatmap.

- The order of output tokens generally follows the order of the input tokens, resulting in a predominantly diagonal pattern in the heatmap.
- Correspondence between source and target language can be learned, especially for unambiguous words. For example, *territory-territoire*, *Israeli-israélien*, *new-nouveau*
- The model's generation of consecutive "the" might be due to weak attention for all input tokens, causing the model generate the word with the highest probability in English, which is "the".

5. Experiments related to "the"

Based on my observations of the model's predictions, the biggest problem I want to fix is the repeatedly generated "the". From the heatmap, one possible explanation is that the attention from the encoder is occasionally not strong enough.

One direct method is that **during testing**, after generating each token's probability in the decoder, adding penalty to the token "the" if it was already generated in the previous step. Therefore, "the" has a smaller chance to be repeatedly generated when the attention is not sufficiently strong to generate other tokens.

```
# punish repeating "the"
if prev_word is not None:
    for i in range(output.shape[0]):
        last_word = prev_word[i].item() if prev_word is not None else None
        if last_word == 19:
            output[i, 19] -= 5  # giving "the" (token_id=19) punishment)
return output, hidden
```

Another method is **during training**, changing the loss function and increasing the loss if the model generates "the" while the target token is not "the. This would encourage the model to learn not to generate "the" too frequently.

```
class WeightedCrossEntropyLoss(nn.Module):
    def __init__(self, the_id=19, penalty_weight=2.0):
        super().__init__()
        self.criterion = nn.CrossEntropyLoss()
        self.the_id = the_id
        self.penalty_weight = penalty_weight

    def forward(self, output, target):
        loss = self.criterion(output, target)
        mask = ((output.argmax(-1) == self.the_id) & (target != self.the_id)).float()
        penalty = self.penalty_weight * mask.mean()  # punish "the" with 2x loss
        return loss + penalty
```

A third experiment is **during testing**, modifying the attention weight matrix by adjusting a factor before it's passed to GRU in the decoder. It is expected that increasing attention will help reduce repetition of "the", while decreasing attention may have the opposite effect.

```
context = context[:, :, :hidden_dim]  # keep the first hidden_dim
context = cxt_ratio * context
rnn_input = torch.cat((embedded, context), dim=2)  # concatenate embedded input and context vector
```

The original prediction and overall test BLEU score after the first epoch of training is:

Prediction: 's new of of theahalah territory territory of the territory of the the the the the the the the Ha Israeli Hazaret) was was confirmed by Israel Israel by Israel Syria.

Average BLEU Score: **4.73**

Method 1: punish repeated "the" during testing

Prediction: 's new of of theahalah territory territory of the territory of the territory of the beginning of the according to the Israeli Ha Israeli Ha,z was the was confirmed by Israel by Israel Syria.

Average BLEU Score: **4.94**

The effect is immediate. It stops repeating “the” and starts generating more diverse words like “beginning of, the according to”, showing that punishing “the” can increase the probability of generating other words and improve translation performance.

Method 2: add loss for repeating “the” during training

Prediction: the raid of the Tsahal territory of the territory of territory in the ((the the the the the the, Haaretz Hazz) was was confirmed by Israel.

Average BLEU Score: **4.28**

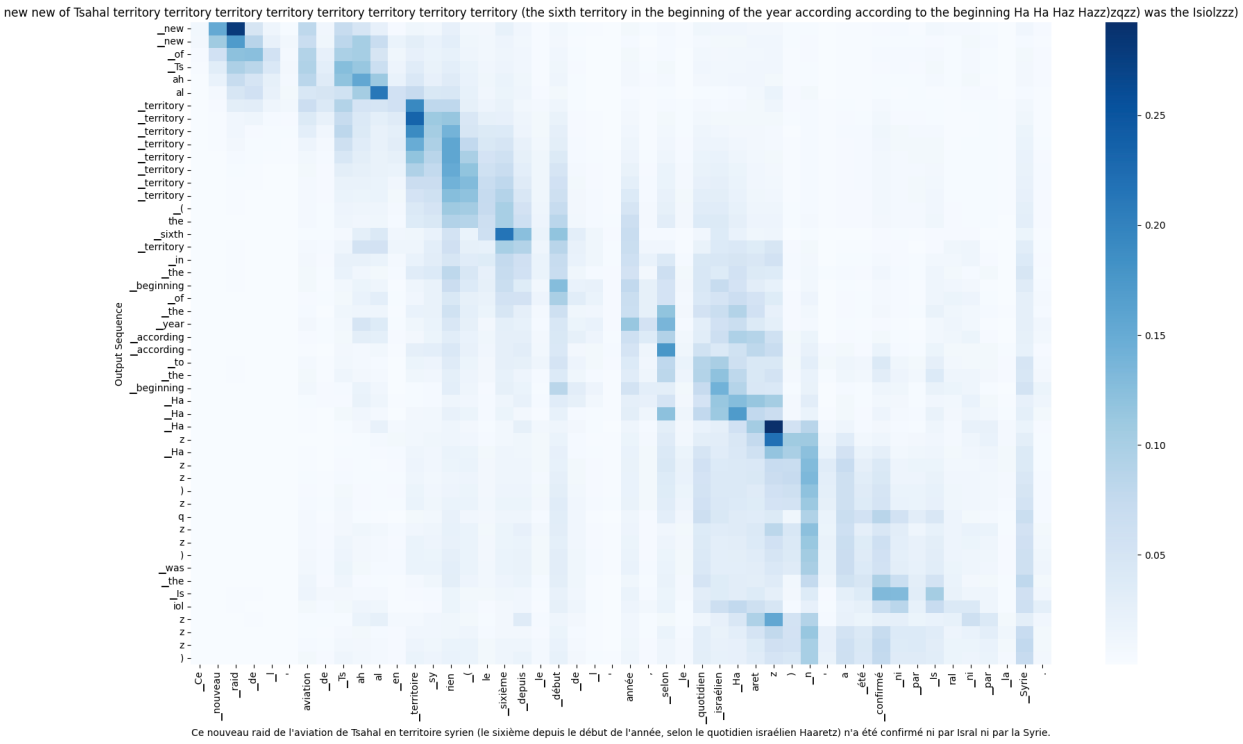
The method is not very effective in reducing repeated uses of “the”, and the overall translation performance drops.

Method 3: multiply whole attention weights by 2x

Prediction: new new of Tsahal territory territory territory territory territory territory territory (the sixth territory in the beginning of the year according according to the Israeli Ha Hazaretzzzz)zqz) was the confirmed by Israel Israel Israel Israel Israel.

Average BLEU Score: **1.31**

The prediction shows more variety in word choice as attention to the source sentence is strengthened. However, the model starts repeating other words for the same reason. We can see from the heatmap that the overall color of the attention weights is darker, and the repeated words focus on the same source tokens.



reduce whole attention weights to 0.5x

Prediction: 's first of the the of the the of the the of the the of the the, the the the the the the the the of the the

Average BLEU Score: 0.96

For comparison, I reduced the cxt_ratio and the model could only generate “the” as the attention from the source sentence was weakened.

I have conducted other experiments, but the results were clearly worse, so I won’t show the details here. These methods include:

- Reducing **teacher_force** in Seq2Seq from 0.5 to 0.3, so that the decoder generates the current token based on the model’s previous output instead of the target token. The idea was to give the model a better chance to correct itself and produce better output.
- Increasing the **dropout** rate from 0.3 to 0.5 to mitigate overfitting after training for multiple epochs.
- Punishing the **token ID** when it is the same as the previous token. The mechanism is the same as in method 1, except that it punishes any repeated token rather than just repeated “the”.

- Punishing “the” in the decoder **during training** instead of testing. Theoretically this is ineffective, because if the model is instructed not to generate “the” during training, it learns less from wrongly generating “the”. And indeed, the result was not impressive.

6. Coverage-based Attention

Another obvious mistake the model makes during the experiments is that it generates repeated words. From the heatmaps, it appears that although the decoder receives attention from the source sentence, it is not strong enough to prevent repetition based on the previously generated words.

I did more research work and finally found the paper *Modeling Coverage for Neural Machine Translation,* which proposed a coverage vector to keep track of the attention history. It helps adjust future attention, encouraging the model consider more about untranslated source words and less about translated ones.

I modified the Bahdanau Attention to incorporate a coverage vector when calculating attention scores. Of course, the decoder and Seq2Seq were modified accordingly to update and pass the coverage vector across decoding steps.

```
class Attention(nn.Module):
    def __init__(self, hidden_dim):
        super(Attention, self).__init__()
        self.attn = nn.Linear(hidden_dim * 3, hidden_dim)
        self.coverage_layer = nn.Linear(1, hidden_dim)  # add coverage layer
        self.v = nn.Linear(hidden_dim, 1, bias=False)

    def forward(self, hidden, encoder_outputs, coverage):  # add coverage input
        batch_size = encoder_outputs.shape[0]
        src_len = encoder_outputs.shape[1]
        hidden = hidden.permute(1, 0, 2)
        hidden = hidden.repeat(1, src_len, 1)
        coverage = coverage.unsqueeze(2)  # (batch, src_len, 1)
        coverage_feat = self.coverage_layer(coverage)  # (batch, src_len, hidden)
        energy_input = torch.cat((hidden, encoder_outputs), dim=2)  # (batch, src_len, hidden*2)
        energy = torch.tanh(self.attn(energy_input) + coverage_feat)  # add coverage impact
        attention = self.v(energy).squeeze(2)
        return F.softmax(attention, dim=1)
```

The result after one epoch of training is quite impressive, as shown below. The model can generate more variety of words from the source sentence with less repetition.

Prediction: new raid of the Army of Tsahal territory in territory (the sixth since the the of the the of the the Israeli invasion of the Israeli newspaper, the was the by Syria by Israel Syria.

Average BLEU Score: 5.00

The BLEU score increased **from 4.73 to 5.00** after one epoch of training with the coverage mechanism. Therefore, I took this approach to finalize my model and proceeded to train the **EN->FR** translation task.

Here is the input-prediction-reference sentence pair and the overall test BLEU score.

Input Tokens: This latest raid by the Israeli air force in Syrian territory (the sixth since the start of the year, according to the Israeli daily newspaper, Haaretz) has been confirmed neither by Israel nor Syria.

Prediction: é par la force aérienne force force force en force en l'armée du,, la sixième de la'année, selon les' israélien israél Hal, a été confirmé confirmé en Isralég Isral.

Reference: Ce nouveau raid de l'aviation de Tsahal en territoire syrien (le sixième depuis le début de l'année, selon le quotidien israélien Haaretz) n'a été confirmé ni par Isral ni par la Syrie.

Average BLEU Score: 4.17

7. Conclusion

I selected RNN-Seq2Seq-Attention as the model for the machine translation task. After setting the parameters, I trained it on Google Colab. Through visualizing the attention weights with heatmaps, I was able to understand how the model generates tokens in relation to the source tokens and conducted some experiments accordingly to improve the performance. Finally, I incorporated coverage mechanism into the attention function and successfully improved the model, increasing the BLEU score by 0.3.

The complete code for this project is available on GitHub at the following link:

https://github.com/codyzhu29/seq2seq_translator