# 1 Part 1 - Seq2Seq

## 1.1 *Code Implementation Details*

***ENCODER***
***__Init__(self, input_size, hidden_size, dropout)***:
• To initialize the encoder, we simply define the multi-gate recurrent unit (GRU) that we will be using as the RNN. We pass in both the provided input size and hidden size, in addition to setting 'batch_first' as True (in order to get the desired output shape). As this is a 1-layer RNN, we do not need to include dropout within the parameters.

***Forward(self, inputs, lengths)***:
• For the encoder's forward method, we first must pack the inputs. To do this, we use 'pack_padded_sequence' which will pack the input tensor based on the provided parameters; we also set "batch_first' as True here (for the desired output shape), and set 'enforce_sorted' as False so that it is sorted unconditionally. With the packed input, we pass into our RNN and get 'outputs' and 'finals'. However, we must first pad the packed output, which we do with 'pad_packed_sequence'; we also specify the 'total_length' with a global variable. Now with the correct dimensions, we then return 'outputs' and 'finals'.

***DECODER***
***__Init__(self, input_size, hidden_size, dropout)***:
• To initialize the decoder, we first want to initialize a bridge from the final encoder state - I chose to use a Linear layer with input and output size as 'hidden_size'. Then, I defined 2 main layers: a GRU that will serve as our RNN (with 'batch_first' as True), in addition to a Linear layer that we will pass our RNN's output through.

***Forward_step(self, prev_embed, hidden)***:
• Forward step allows our decoder to perform a single decoder step. With the provided previous embeddings, we pass it and the provided hidden state through our RNN, providing us with the current hidden state and decoder output. We then pass this decoder output through our Linear layer. We now have both the 'hidden' and 'pre_output' tensors (with the desired dimensions), which we then return.

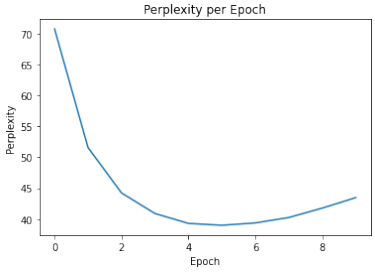
***Forward(self, inputs, encoder_finals, hidden, max_len)***:
• Forward unrolls the decoder at each time step. I first defined an 'empty' 3-D tensor with its shape corresponding to our desired output. We now will iterate over the range defined by the max sequence length, and retrieve the input at the corresponding time step (we squeeze dim 1 to maintain only the pertinent information). With 'decoder_input' we pass through our defined 'forward_step', providing us with the corresponding hidden state and decoder output. We store this output at the corresponding location in our final 'outputs' tensor. Once we have iterated over the entire range, we return the final 'hidden' and 'outputs' tensors.

***GREEDY DECODING***
***Greedy_decode(model, src_ids, src_lengths, max_len)***:
• We are initially given 'encoder_finals' (the result from 'model.decode') and 'prev_token' (which represents the tensor corresponding to the SOS token). With regards to our loop, we will essentially iterate over the range defined by the 'max_len' until we either reach the EOS token or the loop completes.
• At each step, we perform the following operations. We run 'model.decode' with the appropriate parameters, giving us the the corresponding hidden state and decoder output. Then, we run this output through 'model.generator' to provide us with a distribution of tokens. From this distribution, we identify the the index of the maximum value via torch.argmax(), giving us our 'next_token'. If 'next_token' is equal to the EOS token, we exit the loop; otherwise, we append 'next_token' to our 'output' list and re-initialize 'prev_token' with a tensor representation of 'next_token'. We return 'output' - the decoded version of the sentence!

***Seq2Seq Results***
To summarize, Epoch 0's perplexity was 70.7, while Epoch 9's perplexity was **43.5**; the final perplexity value is less than the designated threshold of 45. The model's resulting BLEU score was **6.05**, which is higher than the designated threshold of 5.

```
Epoch 9
Epoch Step: 0 Loss: 42.422604
Epoch Step: 100 Loss: 40.842957
Epoch Step: 200 Loss: 43.389290
Epoch Step: 300 Loss: 47.343346
Epoch Step: 400 Loss: 45.780579
Epoch Step: 500 Loss: 45.635475
Epoch Step: 600 Loss: 48.541172
Epoch Step: 700 Loss: 44.651016
Epoch Step: 800 Loss: 40.237057
Validation perplexity: 43.483774
```

```
BLEU score without Attention: 6.058183
```




(a) Original Seq2seq

Figure 1: Perplexity and BLEU scores for different Seq2seq models

## 1.2 *Questions*

a). With A and $\pi$ being deterministic, at any given timestep 't', we know the state we are in; this also means that we know the observation distribution B at any state. To generate the next state of the RNN, we feed in the input, previous state, and offset. To get the value of the next state 's', we can perform a matrix multiplication of previous state 'p' with the state-state transition matrix 'M'. To generate the output observation, we multiply 's' by 'B' - this is same in both the HMM and RNN. Ultimately, we sample from the resulting distribution in the RNN; this is done in the same manner that there is a stochastic process used to get the output from the HMM given 'B' and a state 's'.

With this, we get the following:
$W_1 = 0; W_2 = B.T; W_h = A.T; b_h = 0; b_x = 0; h_0 = \pi$. f and g are both represented as the identity function.

b). With a deterministic A and B, the vector representing our current state ('s') has size Nx1, where 1 entry is a 1 and all others are zero's. We see this beginning from the very first state, which we show as $\pi$. To transition into a non-deterministic structure, consider the following. $\pi$ will be a random distribution of probabilities over the N states. Now when generating a $h_t$ in our RNN, we multiply 's' by A.T, generating a new state distribution vector. This vector also has size Nx1 as desired, and hence should be able to handle a HMM with non-deterministic transitions.

c). Example 1
i). Tôi mun cho các bn bit v s to ln ca nhng n lc khoa hc đã góp phn làm nên các dòng tít bn thng thy trên báo .
ii). I apos;d like to talk to you today about the scale of the scientific effort that goes into making the headlines you see in the paper .
iii). I want to conclude with frugal assumptions about how many of these problems are actually going to be used to be the case .
iv). Though the structure is very similar, the meanings of certain words/phrases are misrepresented with regards to the context of the entire sentence. For example, the word 'headlines' ('các dòng tít' in Vietnamese) should be in our output, but instead we get 'problems' ('các vn đ' in Vietnamese). It can be seen that 'các' is present in both yet the phrases have different meanings.
v). The reason why this may have happened can be attributed to the model most likely associating the presence of a word to one specific other word. 'Các' is modified by the words around it, and as 'headlines' (and 'problems') are represented via multiple words together as opposed to a singular word in English, there might be some mistranslation as a result.
vi). Integrating attention to the RNN would allow for the model to better identify distant dependencies, which is something the current model doesn't do. With it, the model can better translate sections of text with a greater emphasis on context.

Example 2
i). Khoa hc đng sau mt tiêu đ v khí hu
ii). The science behind a climate headline .
iii). The inflationary theory comes from .
iv). Besides the difference in meaning, the error that can be identified here is the incorrect order of words. 'Behind' connects to the 'headline' in the target, yet we see 'comes from' after where it should be in the model's translation. This incorrect ordering goes against the linguistic construct of English.
v). In Vietnamese, there are certain structural differences with regards to where adjectives and adverbs are placed in relation to what they are modifying. In English, the adjective is usually before the noun and adverb after the verb. Due to this discrepancy, the model may not be able to perfectly interpret such a distinction

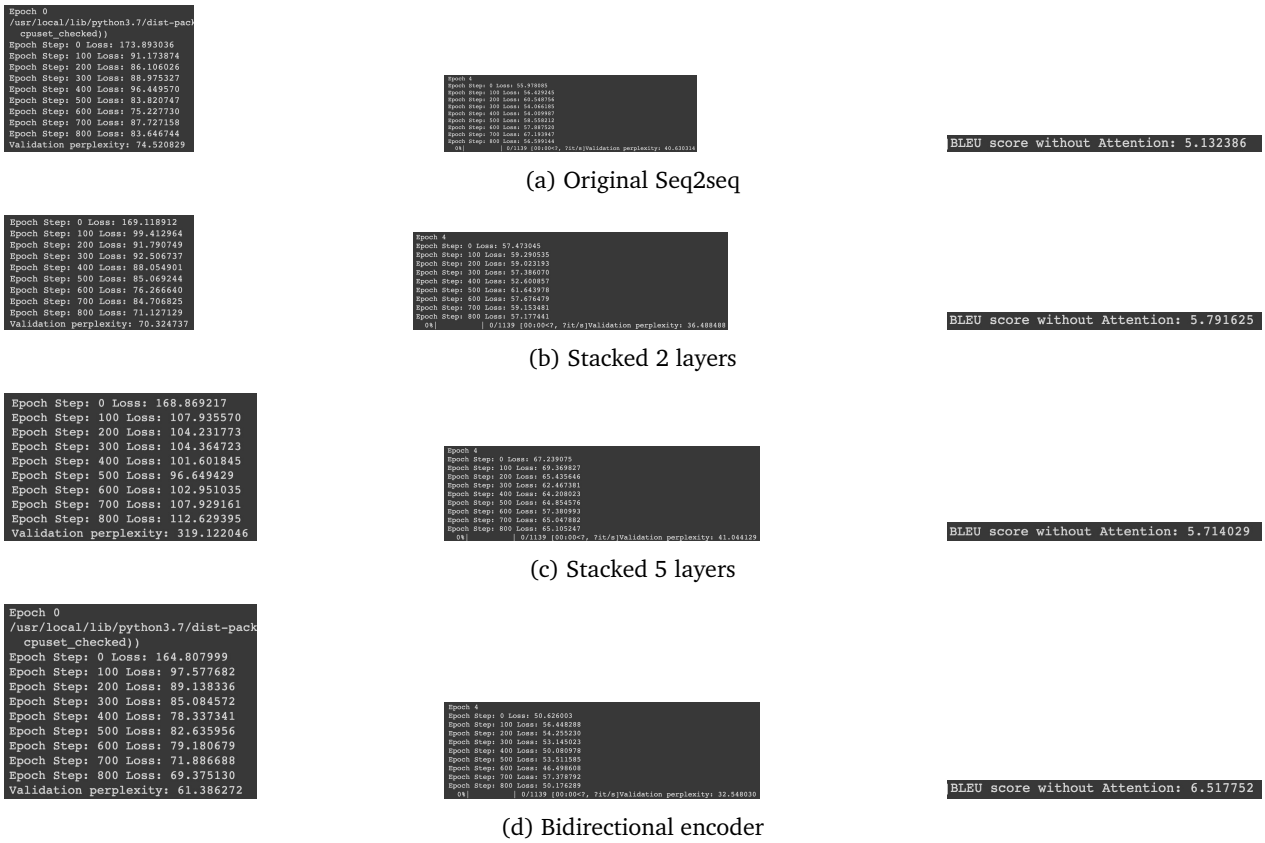
when translating from Vietnamese into English, resulting in potential differences in structure.
vi). One potential solution here would be to have some pre-trained word vectors with certain annotations that provide a little bit more information during training for the model. For example, cases involving words with multiple possible positions in a sentence can come with some representation that allows the model to more effectively identify cases such as the one here.

d). Using a Beam search decoding algorithm instead of the greedy algorithm would allow for the output of a list of most likely output sequences by expanding all possible next steps and keeping the k most likely given k; the most likely word/token at each step may not be the correct one, hence the beam algorithm would allow for us to view a more representative set of potential translations.

e). The BLEU (bilingual evaluation understudy) score evaluates the correspondence between the machine translation and human translation of a text (with a scale from 0 to 1); the closer the machine translation is to the human translation, the higher the BLEU score.

Below, I have attached the results from my experiments to improve upon my model's baseline BLEU score. To do so, I tested the following adaptations to my model: stacking 2 RNN layers, stacking 5 RNN layers, using a bidirectional encoder, and using the LSTM cell type. For ease of training and testing, I used 5 epochs instead of 10, with the same 'test_data_loader' being used throughout. With this being defined, let's briefly discuss our findings. The attached diagrams also provide further detail regarding the results.

• The original Seq2seq model's perplexity after last epoch is 40.6, with a BLEU of 5.13.
• The 2-layer model's perplexity after last epoch is 36.5, with a BLEU of 5.79; this shows a definite improvement.
• The 5-layer model's perplexity after last epoch is 41, with a BLEU of 5.7; this also shows a better improvement in BLEU than perplexity.
• The bidirectional encoder model's perplexity after last epoch is 32.5, with a BLEU of 6.52; this shows the best improvement of all of the models.
• Lastly, the LSTM cell model's perplexity after last epoch is 40.8, with a BLEU of 5.26; here, we pretty similar performance to the original Seq2seq model.




(a) Original Seq2seq

(b) Stacked 2 layers

(c) Stacked 5 layers

(d) Bidirectional encoder

Figure 2: Perplexity and BLEU scores for different Seq2seq models

```
Epoch 0
/usr/local/lib/python3.7/dist-pac
  cpuset_checked))
Epoch Step: 0 Loss: 183.891708
Epoch Step: 100 Loss: 98.335510
Epoch Step: 200 Loss: 85.642921
Epoch Step: 300 Loss: 82.642693
Epoch Step: 400 Loss: 84.432198
Epoch Step: 500 Loss: 81.498482
Epoch Step: 600 Loss: 82.225372
Epoch Step: 700 Loss: 89.939514
Epoch Step: 800 Loss: 80.553162
Validation perplexity: 75.785457
```

```
Epoch 4
Epoch Step: 0 Loss: 53.385921
Epoch Step: 100 Loss: 57.695503
Epoch Step: 200 Loss: 62.020069
Epoch Step: 300 Loss: 58.346130
Epoch Step: 400 Loss: 63.057079
Epoch Step: 500 Loss: 56.935390
Epoch Step: 600 Loss: 59.307804
Epoch Step: 700 Loss: 59.214260
Epoch Step: 800 Loss: 62.037857
Validation perplexity: 40.682263
```

```
BLEU score without Attention: 5.714029
```

(a) LSTM cell

Figure 3: Perplexity and BLEU scores for different Seq2seq models (continued)

# 2 Part 2 - Trees

## 2.1 *Code Implementation Details*

***Part A***
***DATA PROCESSING***
***Tree_dfs(node, span_list, label_dict, mode)***:
As we want to use DFS to recursively traverse through the tree, we simply call 'tree_dfs' with 'child' in the node parameter and everything else staying the same.

***SENTENCE ENCODING***
***__Init__(self, num_words, num_layers, hidden_size, dropout)***:
We simply construct the LSTM layer by passing in the corresponding parameters at the appropriate locations. We pass 'hidden_size' into the 'input_size' parameter as it corresponds to the number of features in the input.

***Forward(self, x)***:
We simply pass the input through the embedding layer, then pass the result through the LSTM layer. From the returned result, we simply just return the outputs.

***SPAN ENCODINGS***
***Get_span_embeddings(word_embeddings, span_indices)***:
We first extract the corresponding dimensions from the input that we want to represent in the output, after which we create and empty tensor with these dimensions. Then, we iterate through over the range of 'num_span' and retrieve the embedding indeces of the corresponding span. We retrieve the values from the tensors and use them as the indeces into word embeddings, which we then concatenate - this represents the span embedding of this step. We store each of these embeddings in our 'result', and return upon completion.

***TAG PREDICTION***
***Forward(self, x, span_indices)***:
We simply follow the instructions provided: pass the input through the sentence encoder, pass the embedding through dropout, squeeze dimension 0 to get rid of it in the tensor (needed to match the desired input for the next step), pass through 'get_span_embeddings' to get our span embeddings, which we then pass through a linear layer to the the logits. We then return logits!

***TRAINING LOOP***
For training, we simply pass in our 'sent_inputs' and 'span_indices' into the classifier, giving is the prediction. Then, we simply go through the process of computing loss and stepping the optimizer. We do this for each iteration in the corpus training set. For validation, we just determine the prediction and loss at each iteration.

***Part B - CKY***
There are 2 cases here, the first being 'if j == i+1'. Here, we first determine the 'label_probabilities' with 'cur_span.' Then with this, we can easily extract the best label and best score at this span, which we then set into the 'best_label' and 'best_score' dictionaries. There is a little more work to be done if 'j != i+1'. We will need to iterate over the range of (i+1, j), over which we will calculate the scores of splitting the span into 'best_score[(i, k)] + best_score[(k, j)]' and keep track of the 'max_k' and 'max_k_score' values. Then, we can calculate the 'max_l_score', which will be used to determine the actual bet score; the best split will be set to 'max_k' and 'best_label' is the same as above.

***TREE CONSTRUCTION***
Here, we first simply extract the 'k' value from 'best_split'. Then, we set 'node.children' the result from 2 DFS

builds: one with k as the middle split from the left, and one with k as the middle split from the right.

## 2.2 *Questions*

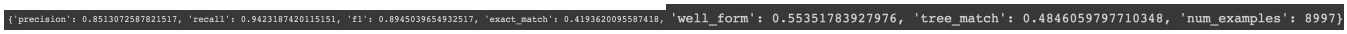
a-i). CFG Rules
N → man | dog | cat
IV → meowed | barked | ran
TV → feared | chased | loved
D → the
C → that
NP → D N
VP → IV | P IV
P → C NP TV | C NP P TV
S → NP VP

a-ii). CFG Rules
N → man | dog | cat
IV → meowed | barked | ran
TV → feared | chased | loved
D → the
C → that
NP → D N
VP → IV | P IV
P → C NP TV | C NP C NP TV TV
S → NP VP

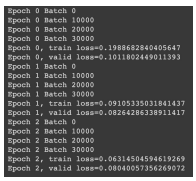
The main difference lies in the rule for P. As we know that there is a maximum of double nesting, as opposed to the infinite number allowed in the first question, we can explicitly define the grammar for what it would look like. Hence, the only thing we change is the definition of P here (which I essentially identify as a ”Phrase”).

***PART A***
a). With my trained model, I describe my results below; I also have attached images of the notebook's outputs.

F1 Score: 0.895
Exact_match: 0.419
Tree_match: 0.485
Well_form: 0.554
Precision: 0.851
Recall: 0.942




(a) Notebook scores




(b) Model loss during training

Figure 4: Tree scores and results after training

b). To begin, unidirectional LSTMs only store information from the past because the only inputs it has seen are from the past. On the other hand, bi-directional LSTMs run the input in two ways, one from the past to future information (forward) and one from the future to past information (backward). Accordingly, by combining the hidden states, this allows the model at any point in time to preserve information from both the past and future. Bi-directional LSTMs can accordingly better understand context of a given input and better classify a given input, and hence are preferred when generating span and word embeddings.

c). In order to improve the algorithm for generating span embeddings, we essentially want to find a way through which we can better represent and aggregate the span. To do this, we also need to ensure that the algorithm's output has a fixed representation - essentially consistent shape. With this, I believe the following approach will improve our algorithm. The embedding will maintain both the first and last words of a span, but we will also take the mean of the embeddings of words between these 2 words. By doing so, our embedding will have the critical information of where it starts and ends, but also about what general information is included between. We will thus want to concatenate the embeddings of the first word, of the mean of middle words, and of the last word of the span. Then, we will want to re-shape our result to match the representation of our desired output. Our resulting embedding presents a more informative representation of the span!

***PART B***

a). There was a slight increase in precision, exact_match, and tree_match, and a great increase in well_form. However, there was a slight decrease in f1 score and recall. The lower F1 score could be attributed to the fact that the encoder got the entirety of the input, while our model here got the vectorized embedding of the input. Hence, we have less information to build our trees, corresponding to a lower F1 score.

F1 Score: 0.865
Exact_match: 0.437
Tree_match: 0.514
Well_form: 1.0
Precision: 0.869
Recall: 0.862

```
{'precision': 0.8692164819174485, 'recall': 0.8615720165574174, 'f1': 0.8653773673796289, 'exact_match': 0.4373680115594087,
'well_form': 1.0, 'tree_match': 0.5139490941424919, 'num_examples': 8997}
```

(a) Notebook scores

Figure 5: Tree scores and results after training

b). Below are 2 examples of trees that have the correct structure and were able to get almost all of the labelings correctly. The main label "IN" is correct in both of my examples, but it seems that the model over-labels 'what' and 'family' here. This demonstrates that the model might have overtrained on examples where these words were more heavily emphasized.

```
REF: [IN:UNSUPPORTED what [SUB breakfast [SUB locations [SUB within [SUB 5 [SUB miles [SUB of [SUB me [SUB open [SUB at [SUB 6 am ] ] ] ] ] ] ] ] ] ]
DEC: [IN:UNSUPPORTED_EVENT what [SUB breakfast [SUB locations [SUB within [SUB 5 [SUB miles [SUB of [SUB me [SUB open [SUB at [SUB 6 am ] ] ] ] ] ] ] ] ] ] ]
```

```
REF: [IN:UNSUPPORTED family [SUB parks [SUB in houston ] ] ]
DEC: [IN:UNSUPPORTED_EVENT family [SUB parks [SUB in houston ] ] ]
```

(a)

Figure 6: Tree outputs w/ labeling mistakes

c). When we build our model, we essentially iterate over all spans of a given sentence, during which we attempt to provide a label to each of these spans. For each of these sentences, the correct spans are provided with a specific, corresponding label. However, some of these spans do not have a specific, associated label; we will call these spans as 'no-label'. For these 'no-label' spans, if we provide the label as None, our model will essentially not need to decode the subtrees for each of these spans - meaning that we will not need to attempt in splitting or building from these subtrees. Additionally with this, we can force our model to infer the meanings and labeling of nearby words. Ultimately, we help our decoding.

d). As different languages have different structures, the Seq2seq model - which is used to translate from one language to another - must be able to interpret such differences in structure and context of a sentence before it generates a translation. With this, it can be seen that Seq2seq could be represented in a tree-like structure. The key to framing the problem as a Seq2seq task is in the attention. To fully understand a sentence, it is critical to understand the relation of words in phrases in relation to one another. Hence, we can use multiple attention layers via a Transformers architecture where each layer can focus on a particular part of speech; it can also see which words modify and are modified by other words. Conclusively, a Seq2seq model can be used where the inputs are sentences and the the targets are the generated Tree structures.