Introduction to NLP

Codemix Generation

Team 17 : Qwertyuiop

—

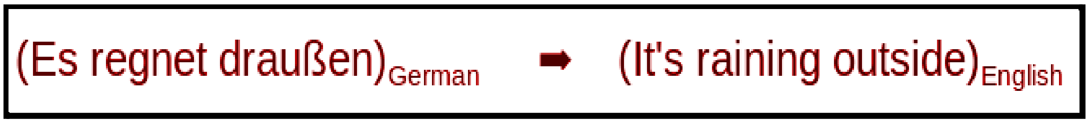
Anishka Sachdeva (2018101112)

Satyam Viksit Pansari (2018101088)

Overview

The objective is to convert a English sentence to Codemix (English and Hindi) and vice-versa using a Neural Machine Translation (NMT) system.

The problem statement can be viewed as :



(Es regnet draußen)$_{\text{German}}$ ➡ (It's raining outside)$_{\text{English}}$

Note : The diagrams may contain any language just for the explanation purpose. But the project is built considering English and Hindi languages.

Goals

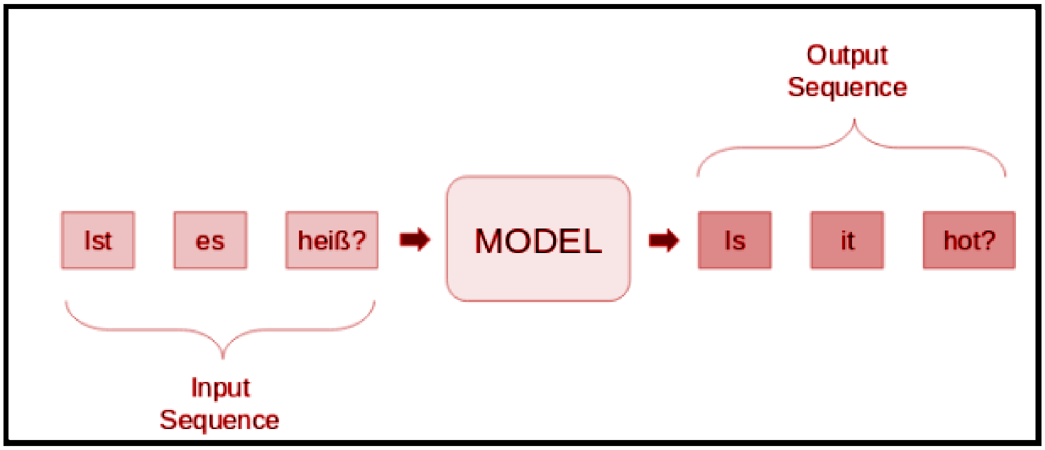
We'll be building a sequence-to-sequence (seq2seq) machine learning model with Attention to go from one sequence to another.

Let’s define briefly what sequence-to-sequence models are :

Sequence-to-Sequence (seq2seq) models

A sequence-to-sequence model is a model that takes a sequence of items (words, letters, features of an image etc.) and outputs another sequence of items.

Here, the basic idea of Sequence-to-Sequence modeling in neural machine translation, is that both the input and output are sentences. In other words, these sentences are a sequence of words going in and out of a model.



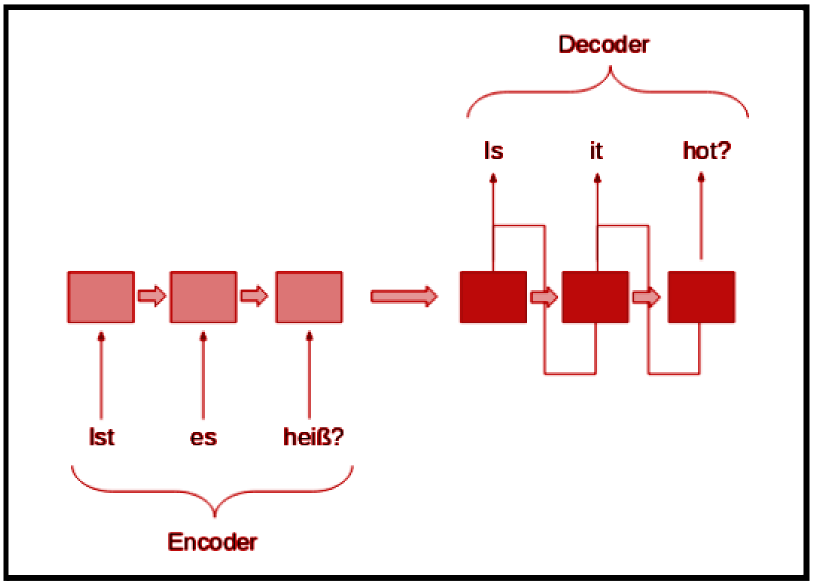
A typical seq2seq model has 2 major components namely,

1. **Encoder**

2. **Decoder**.

The **encoder** processes each item in the input sequence, it compiles the information it captures into a vector (called the **context**). After processing the entire input sequence, the **encoder** sends the **context** over to the **decoder**, which begins producing the output sequence item by item.

Both these parts are essentially two different recurrent neural network (RNN) models combined into one giant network.



Tasks

I. Baseline

Dataset Used : https://ritual.uh.edu/lince/datasets

1. Text pre-processing

Quite an important step in any project, especially so in NLP. The data we work with is more often than not unstructured so there are certain things we need to take care of before jumping to the model building part.

a. Tokenization

- We made use of Spacy Tokeniser.
- We used English tokenizer (python -m spacy download en_core_web_sm) for both, English and Codemix sentences.
- Next, we create the tokenizer functions. These can be passed to torchtext and will take in the sentence as a string and return the sentence as a list of tokens.
- In one case, SOURCE is English and TARGE is CodeMix. In the other model, SOURCE is CodeMix and TARGET is English.
- The torchtext's Field (which handles how the data should be processed) also appends the "start of sequence" and "end of sequence" tokens via the init_token and eos_token arguments, and converts all words to lowercase.

b. Loading the data

- The data is divided into Train, Validation and Test data.

c. Building the Vocabulary

- Vocabulary is built for the SOURCE language and TARGET language from their training sets.
- The vocabulary is used to associate each unique token with an index (an integer). The vocabularies of the source and target languages are distinct.
- Using the min_freq threshold, some tokens are converted to <unk> (unknown) token.

2. Machine Learning Model building

a. Building the Encoder

With a bidirectional RNN, we have two RNNs in each layer. A forward RNN going over the embedded sentence from left to right (shown below in green), and a backward RNN going over the embedded sentence from right to left (teal).

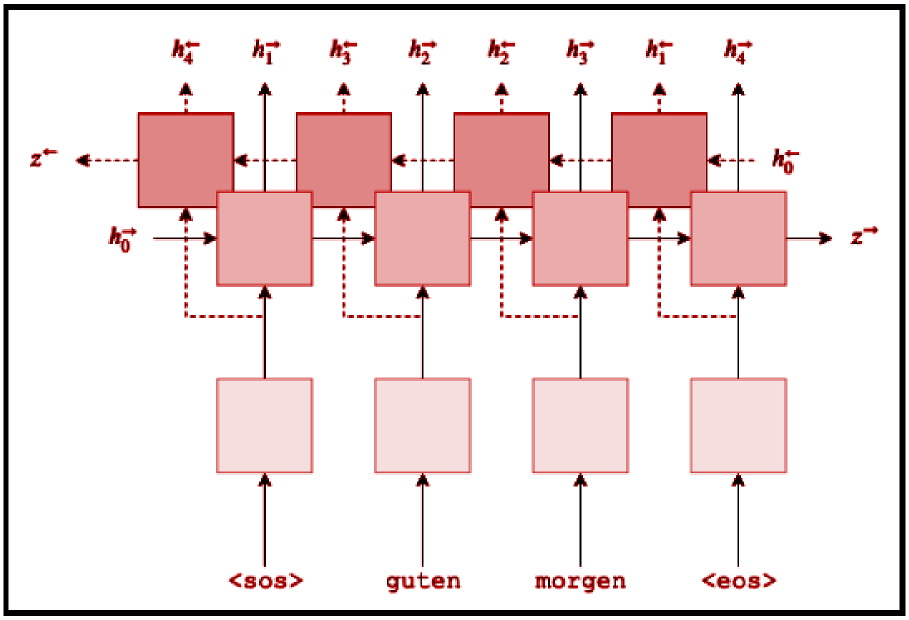
$$h_t^{\rightarrow} = \text{EncoderGRU}^{\rightarrow}(e(x_t^{\rightarrow}),\, h_{t-1}^{\rightarrow})$$
$$h_t^{\leftarrow} = \text{EncoderGRU}^{\leftarrow}(e(x_t^{\leftarrow}),\, h_{t-1}^{\leftarrow})$$

We pass embedded input to encoder RNN, which tells PyTorch to initialize both the forward and backward initial hidden states to a tensor of all zeros. We'll also get two context vectors, one from the forward RNN after it has seen the final word in the sentence , and one from the backward RNN after it has seen the first word in the sentence.

As the decoder is not bidirectional, it only needs a single context vector, z, to use as its initial hidden state, s0, and we currently have two, a forward and a backward one. We solve this by concatenating the two context vectors together, passing them through a linear layer, g, and applying the tanh activation function.

$$z = \tanh(g(h_T^{\rightarrow},\, h_T^{\leftarrow})) = \tanh(g(z^{\rightarrow},\, z^{\leftarrow})) = s_0$$

As we want our model to look back over the whole of the source sentence we return outputs, the stacked forward and backward hidden states for every token in the source sentence. We also return hidden, which acts as our initial hidden state in the decoder.



Attention

Briefly, attention works by first, calculating an attention vector, a, that is the length of the source sentence. The attention vector has the property that each element is between 0 and 1, and the entire vector sums to 1. We then calculate a weighted sum of our source sentence hidden states, H, to get a weighted source vector, w.

Now we look up the attention layer in depth. This will take in the previous hidden state of the decoder, $st-1$, and all of the stacked forward and backward hidden states from the encoder, H. The layer will output an attention vector, at, that is the length of the source sentence, each element is between 0 and 1 and the entire vector sums to 1.
Intuitively, this layer takes what we have decoded so far, $st-1$, and all of what we have encoded, H, to produce a vector, at, that represents which words in the source sentence we should pay the most attention to in order to correctly predict the next word to decode.

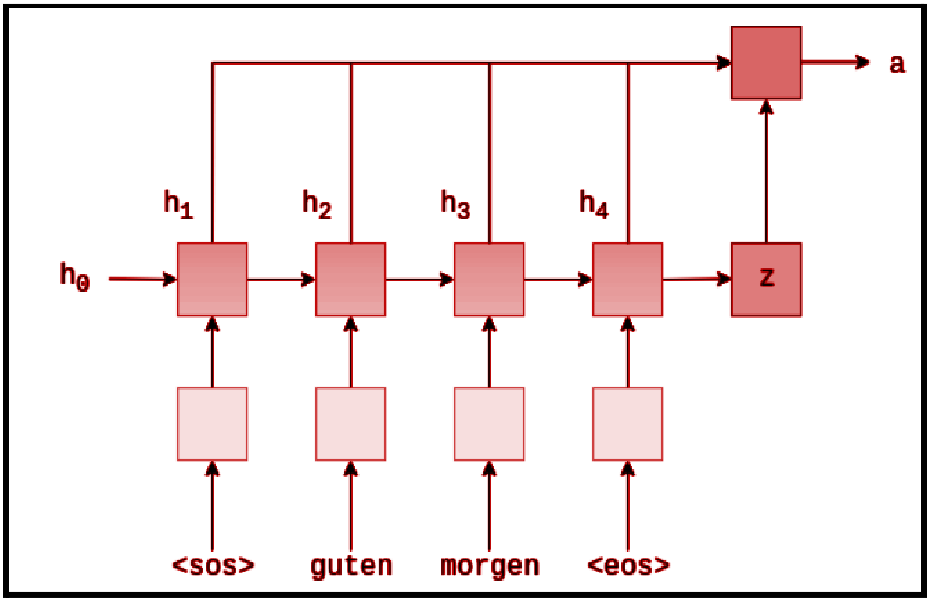
First, we calculate the energy between the previous decoder hidden state and the encoder hidden states. As our encoder hidden states are a sequence of T tensors, and our previous decoder hidden state is a single tensor, the first thing we do is repeat the previous decoder hidden state T times. We then calculate the energy, Et , between them by concatenating them together and passing them through a linear layer (attention) and a tanh activation function.

$$E_t = \tanh(\text{attn}(s_{t-1}, H))$$

This can be thought of as calculating how well each encoder hidden state "matches" the previous decoder hidden state.

Finally, we ensure the attention vector fits the constraints of having all elements between 0 and 1 and the vector summing to 1 by passing it through a Softmax layer.

$$a_t = \text{softmax}(\hat{a}_t)$$



b. Building the Decoder

The decoder contains the attention layer, attention, which takes the previous hidden state, st−1, all of the encoder hidden states, H, and returns the attention vector, at.

We then use this attention vector to create a weighted source vector, wt, denoted by weighted, which is a weighted sum of the encoder hidden states, H, using a_t as the weights.

$$w_t = a_t H$$

The embedded input word, the weighted source vector, and the previous decoder hidden state are then all passed into the decoder RNN, with d(y_t) and w_t being concatenated together.

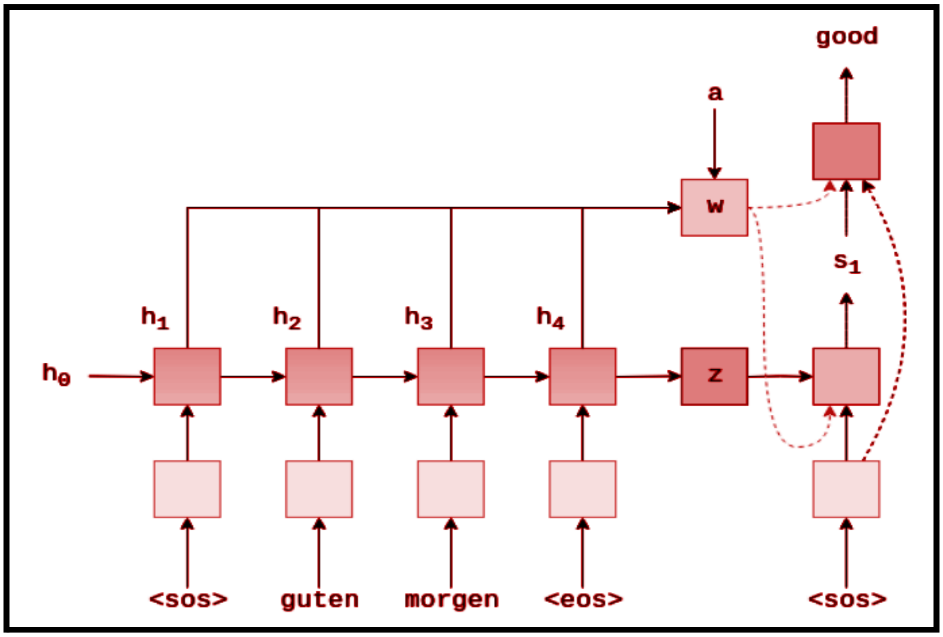
$$s_t = \text{DecoderGRU}(d(y_t), w_t, s_{t-1})$$

Here,

d(y_t) : embedded input word

w_t: weighted source vector

s_{t-1}: previous decoder hidden state



c. Training the Seq-2-Seq Model

- We initialise our parameters, encoder, decoder and seq-2-seq model.
- We create an optimizer.
- We initialize the loss function.

The loss taken is $e^{actualLoss}$ to get a better analysis of the models because the loss changed every minute after epoch in the later part of the training and hence it was infeasible to compare without taking its exponent.

II. Baseline+

1. **Addition of LID Tags** : LID stands for Language Identification. It means to give a language tag to all the words in the dataset i.e. in the codemix, since the words belong to Hindi and English language so all the words get a tag either of Hindi or English.

   Tool used for LID Tags : https://github.com/irshadbhat/litcm

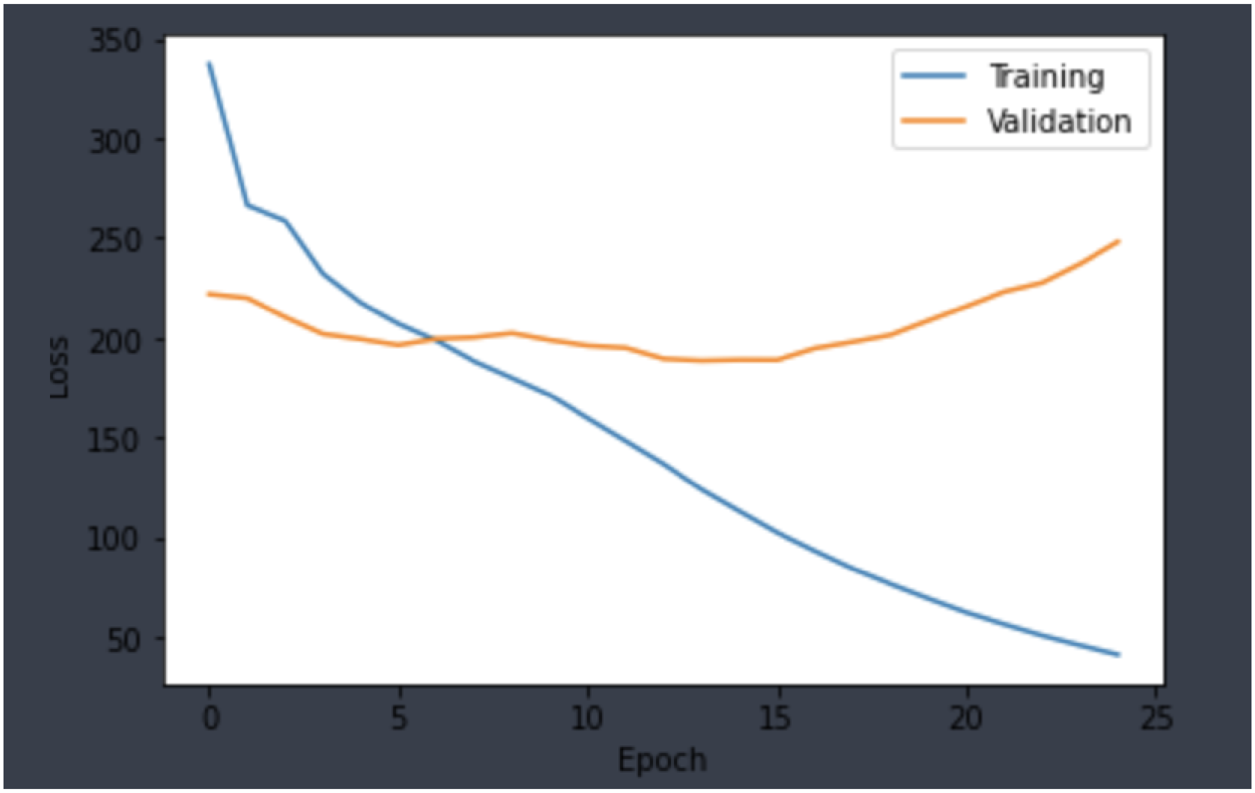
---

Results

I. For model without LID Tags (Baseline):

a) Codemix to monolingual

Following was the best result obtained



Epoch: 11

Train Loss: 5.073 | Train PPL: 159.638

Val. Loss: 5.278 | Val. PPL: 196.051

Train Loss (exponent to analyse better): 159.638

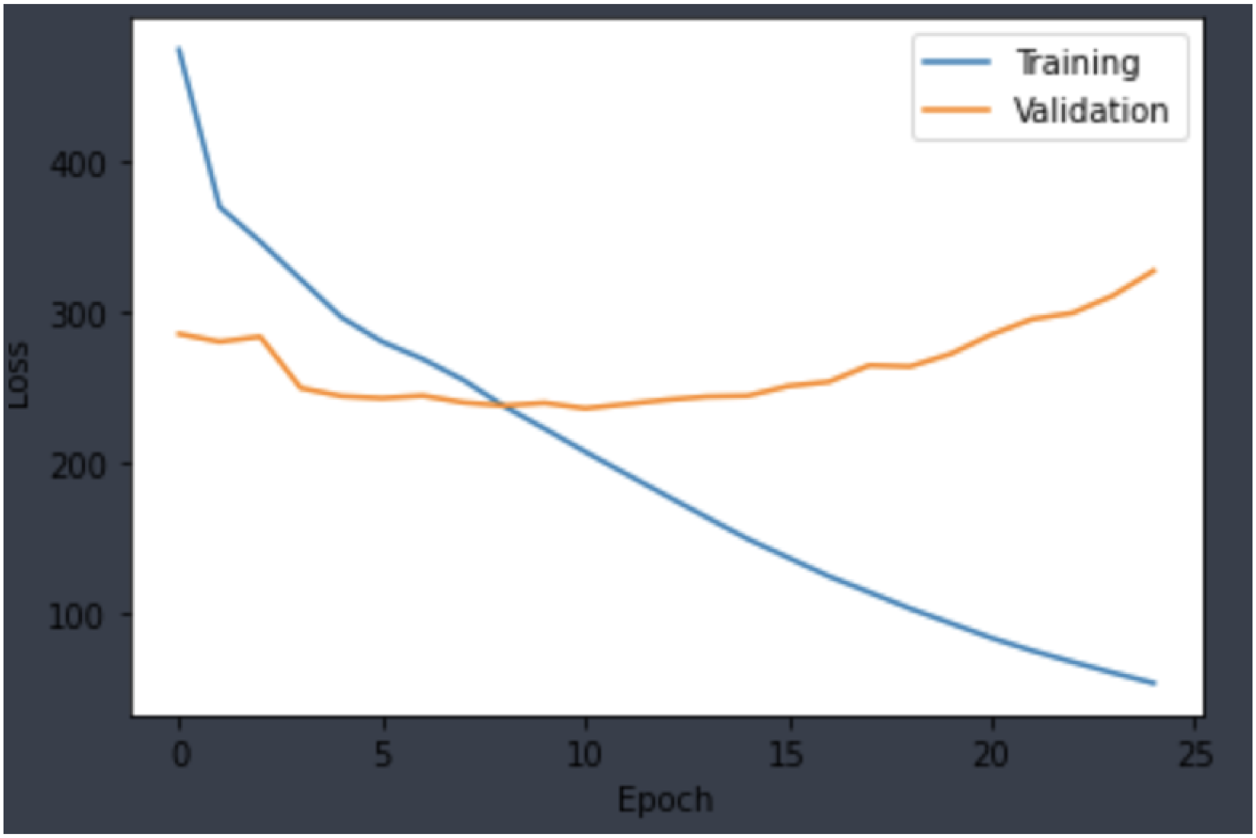
Val. Loss (exponent to analyse better): 196.051

Test Loss (exponent to analyse better): 398.338

Average BLEU Score: 0.0218309294243317

b) Monolingual to codemix

Following was the best result obtained



Epoch: 11

Train Loss: 5.333 | Train PPL: 207.021

Val. Loss: 5.463 | Val. PPL: 235.773

Train Loss (exponent to analyse better): 207.021

Val. Loss (exponent to analyse better): 235.773

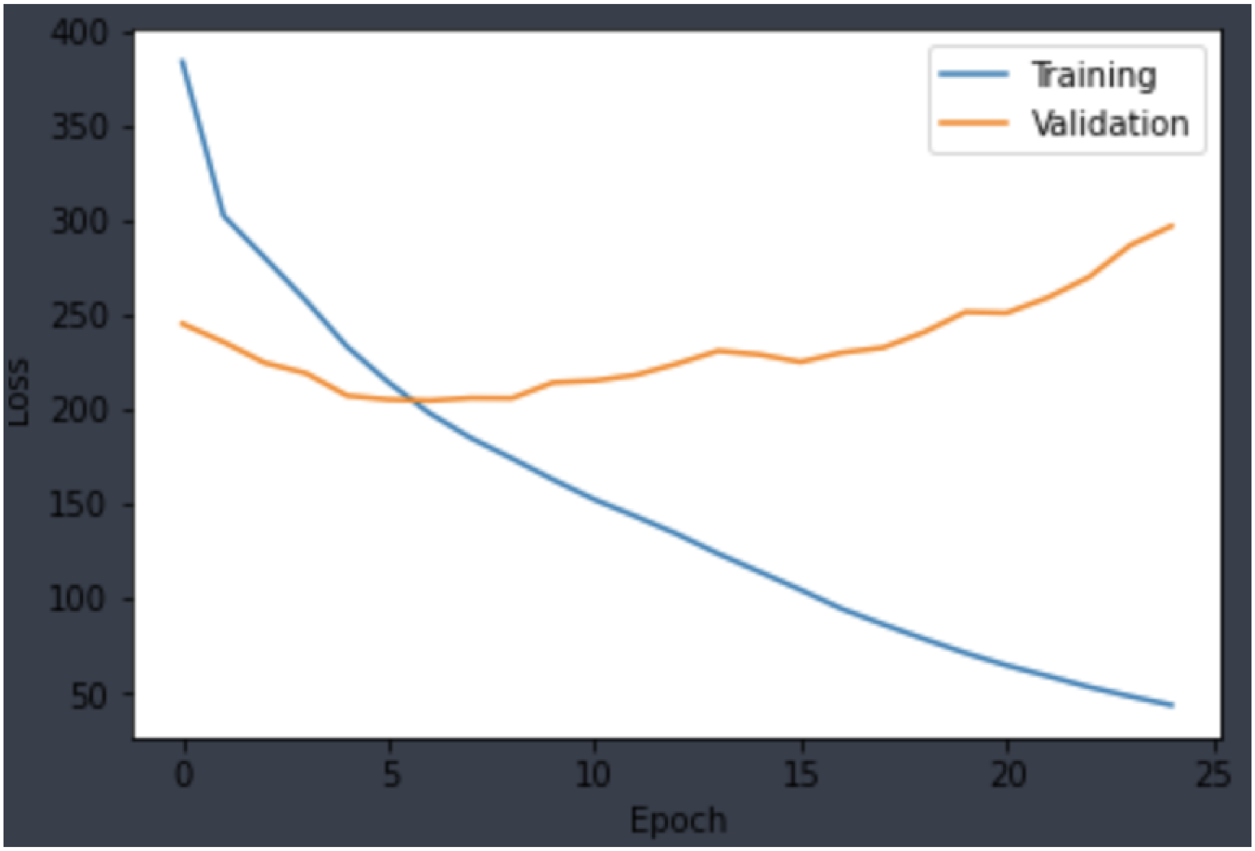
Test Loss (exponent to analyse better): 365.619

Average BLEU Score: 0.0322889663350351

II. For model with LID Tags (Baseline+):

c) Codemix to monolingual

Applying LID tags to best result by Baseline gave the following results for baseline+ :



Epoch: 07

Train Loss: 5.285 | Train PPL: 197.301

Val. Loss: 5.318 | Val. PPL: 204.024

Train Loss (exponent to analyse better): 197.301

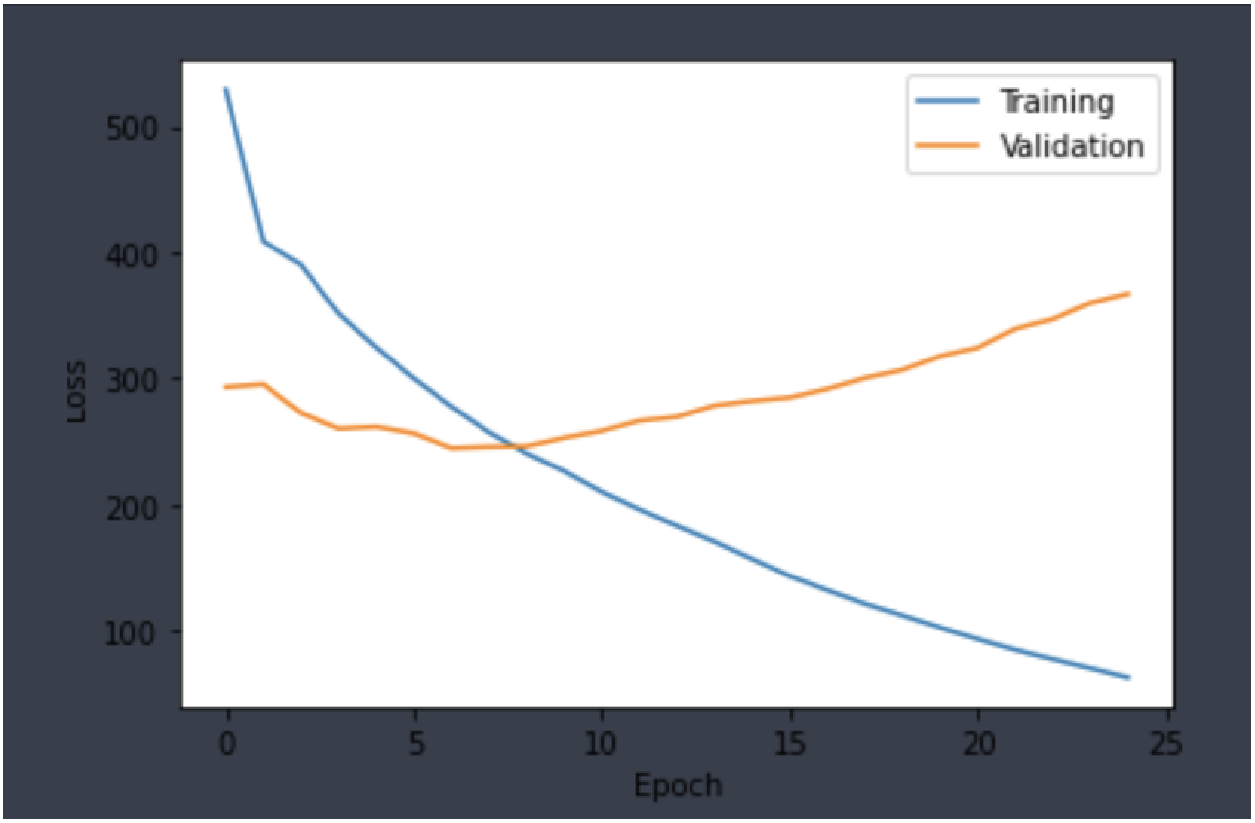
Val. Loss (exponent to analyse better): 204.024

Test Loss (exponent to analyse better): 198.338

Average BLEU Score: 0.0718309294243317

d) Monolingual to codemix

Applying LID tags to best result by Baseline gave the following results for baseline+ :



Epoch: 07

Train Loss: 5.627 | Train PPL: 277.891

Val. Loss: 5.501 | Val. PPL: 244.962

Train Loss (exponent to analyse better): 277.891

Val. Loss (exponent to analyse better): 244.962

Test Loss (exponent to analyse better): 265.619

Average BLEU Score: 0.06751675427023374

Analysis

I. For model without LID Tags (Baseline) :

- Attention outputs are better overall.
- Less amount of data gives bad results.
- Even the dataset is not perfect and it does not have perfect translations.

  This is one of the main reasons why along with less amount of data, the model gives bad results.

- The model correctly predicts the punctuation marks with very high accuracy.

II. For model with LID Tags :

- We didn't get any meaningful sentences.

  Reason :

  - Data is very less in amount.
  - There is a lot of variation in the length of sentences with the length varying from 1 word to ~300 words.
  - When we take min_freq = 2, then we get english vocab has 3500 words and codemix vocab has 4500. But when we take min_freq = 10, english vocab only has 500 words and codemix vocab only has. This shows that the dataset has almost all words as unique having frequency one or two.

Deliverables

1. 4 jupyter notebooks containing the models for the following :
   a. Neural Model with Attention and without LID Tags (English to Codemix)
   b. Neural Model with Attention and with LID Tags (English to Codemix)

   c. Neural Model with Attention and without LID Tags (Codemix to English)
   d. Neural Model with Attention and with LID Tags (Codemix to English)
2. A presentation explaining the project.
3. A detailed report explaining the project.
4. A README.md file with instructions to run the project.

Challenges

1. The LID tool did not work initially. We had to make some tweaks to make it work and integrate it with our code.
2. Initially, we reversed the sentences and did not get good expected results. And therefore, we changed back to original sentences being converted to sequences.

Find model checkpoints here :
https://drive.google.com/drive/folders/1d8Kw-TeY9zm-tmOoV4XYPYLuvHdlQxGB?usp=sharing