iNLP Project Final Report: Code Mix Generation

Team No : 30

Course Name : Introduction To Natural Language Processing

Course Code : CS7.401

Semester : Spring'23

Instructor Name : Prof. Manish Shrivastava

Name	**Roll No**	**Contribution**
Sk Abukhoyer	2021201023	Translator & Report
Soumodipta Bose	2021201086	Generator & Report & PPt
Ashutosh Gupta	2021201086	Generator & Report

Introduction

Code mixing is the use of multiple languages in a single sentence, often seen in bilingual or multilingual communities. Code mix generation in Natural Language Processing (NLP) refers to the creation of code-mixed text using statistical methods or neural networks. The purpose of code mix generation is to generate text that reflects language diversity and facilitate communication between individuals who use different languages. To create a code-mixed language model, a diverse Lince dataset of code-mixed text is collected, preprocessed, and used to train our baseline neural model. Code mix generation has applications in fields such as machine translation, speech recognition, and sentiment analysis, but it can be challenging to process due to the variations in language, syntax, and grammar used. Hinglish, an example of code mixing, which is the purpose of our project and it is frequently observed in user-generated content on social media platforms, websites, and comments. The objective of this project is to generate code-mixed Hinglish text from the original English using basic neural algorithms and also to translate English text to code-mixed Hinglish text.

Project Proposal

In this project we aim to do two things:

1. **Code mixed generation:** Firstly to generate code-mixed text(Hinglish). We will be doing this by using a variation of Recurrent architectural neural networks(RNNs) that is known as LSTM using the baseline model and improvising it to generate code mixed sentences.We have created a baseline LSTM neural language model and analyzed the result of the same for this interim submission only. In the next submission, We will be studying the different variations of the baseline models and analyze their performance and improve them.
2. **Code mixed Translation:** Secondly to convert english text to code mixed Hinglish text using a seq-to-seq model that uses Encoder Decoder architecture to perform machine translation. We will be studying the model using different hyperparameters and analyze the performance and improve them.

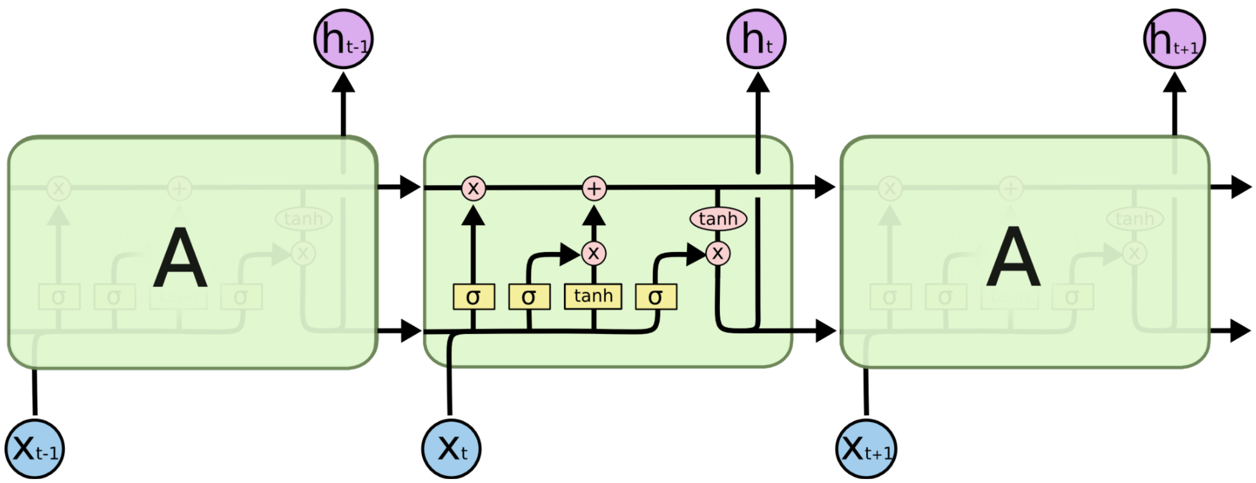
PART I : Code Mixed Generation

LSTM (Long Short Term Memory Networks)

LSTM is a type of neural network architecture used in Natural Language Processing (NLP) for language modeling.

LSTM neural language models are used to generate text by predicting the likelihood of a sequence of words given the preceding words. This type of model is especially useful for generating longer text sequences because it can remember and keep track of important information from previous words in the sequence.

LSTM neural language models work by processing input sequences of words and predicting the probability distribution of the next word in the sequence. The architecture of the LSTM network includes memory cells that store information for longer periods of time, and gates that control the flow of information in and out of the memory cells. The gates can learn to open or close based on the input and the context, allowing the LSTM network to selectively remember or forget information as needed.



One of the advantages of using LSTM neural language models is their ability to capture long-term dependencies in text data, which can be difficult for other types of models to do. This makes them well-suited for generating coherent and contextually appropriate text, such as in language translation, chatbots, or text summarization.

Overall, LSTM neural language models are a powerful tool for generating text and have been shown to achieve state-of-the-art results in many NLP tasks.

Perplexity

Perplexity is a metric used to judge how good a language model is. We can define perplexity as the inverse probability of the test set, normalised by the number of words.

$$PP(W) = \sqrt[N]{\frac{1}{P(w_1, w_2, \ldots, w_N)}}$$

Mix Factor (MF)

Mix Factor, referred to as MF is based on Code Mixing Index (CMI). It is the ratio of number of words which are not written in the dominant language of the sentence to the total number of language-dependent words present in the sentence. It can be written as:

$$MF = \frac{W' - max\{w\}}{W'} \text{ , if W' > 0,}$$

$$MF = 0 \text{ , if W' = 0,}$$

1. Data Preprocessing:

The preprocessing code defines several functions that are used to preprocess text data for this natural language processing task. The **read_data()** function reads a text file and preprocesses each line of text using the other functions defined in the code.

The **replace_dates()** function replaces dates in various formats with the string <DATE>. The **replace_concurrent_punctuation()** function replaces sequences of two or more consecutive punctuation characters with a single space. The **replace_hash_tags()** function replaces hashtags with the string <HASHTAG>.

The **remove_special_characters()** function removes any special characters that are not punctuation marks. The **remove_extra_spaces()** function removes any extra spaces from the text. The **replace_hyphenated_words()** function replaces hyphenated words with words separated by a space.

Finally, the **read_data()** function reads each line of text from a file and preprocesses it using the other functions stated above.

```python
def custom_cleaner(line):
    line = re.sub(r'<|>', ' ', line)
    line = replace_dates(line)
    line = replace_hyphenated_words(line)
    line = replace_hash_tags(line)
    # remove < and > from the text
    line = clean(line, no_emoji=True,
                 no_urls=True,
                 no_emails=True,
                 no_phone_numbers=True,
                 no_currency_symbols=True,
                 replace_with_url=" <URL> ",
                 replace_with_email=" <EMAIL> ",
                 replace_with_phone_number=" <PHONE> ",
                 replace_with_currency_symbol=" <CURRENCY> ",
                 lower=True)
    line = remove_special_characters(line)
    #line = replace_concurrent_punctuation(line)
    line = clean(line, no_numbers=True, no_digits=True, no_punct=True,
                 replace_with_number=" <NUMBER> ", replace_with_digit=" ", replace_with_punct="")
    line = "<BEGIN> " + line + " <END>"
    line = remove_extra_spaces(line)
    return line
```

It then tokenizes the preprocessed text using the basic_english tokenizer from the nltk library and returns a list of tokenized sentences.

2. Data Labeling:

This code defines a PyTorch Dataset class called LinceDataset for training and validation data in a language modeling task. The __init__ function reads data from a file, builds a vocabulary (if not provided), and creates the training or validation dataset. The read_data function reads the input data from a file, which has English and Hinglish (a mix of Hindi and English) sentences separated by a tab character. The build_vocab function builds a vocabulary for the input data. The __get_seq function converts a sequence of words to a sequence of corresponding indices from the vocabulary. The __create_dataset function generates the training or validation dataset. The get_dataloader function returns a PyTorch DataLoader object to iterate over the dataset in batches.
To create the validation dataset, the code instantiates the LinceDataset class with the valid.txt file and the vocabulary of the training dataset.

You, 2 months ago | 1 author (You)

```python
class LinceDataset(Dataset):
    def __init__(self, filename, vocab_english=None, vocab_hinglish=None, ngram=5):
        data_english, data_hinglish = self.read_data(filename)
        if vocab_hinglish is None:
            self.vocab_h, self.ind2vocab_h = self.build_vocab(data_hinglish)
        else:
            self.vocab_h = vocab_hinglish
            self.ind2vocab_h = {v: k for k, v in vocab_hinglish.items()}
        self.n = ngram
        self.x, self.y = self.__create_dataset(data_hinglish)

    def get_vocab(self):
        return self.vocab_h

    def read_data(self, filename): ...

    def build_vocab(self, data): ...

    def get_ngram(self, tokens): ...

    def __get_seq(self, tokens): ...

    def __create_dataset(self, data): ...

    def __len__(self):
        return len(self.x)

    def __getitem__(self, idx):
        return self.x[idx], self.y[idx]

    def get_dataloader(self, batch_size, shuffle=True):
        return DataLoader(self, batch_size=batch_size, shuffle=shuffle, drop_last=True)
```

3. Models:

1. **Baseline model:** Using single LSTM, with dropout and multiple layers.

GramNet is a language model that uses an LSTM to predict the next word in a sequence. It takes as input the vocabulary size, number of hidden units, number of LSTM layers, embedding dimension, dropout rate, learning rate, model save path, and device. It initializes an embedding layer, an LSTM layer with the given hyperparameters, and a linear layer to map the output of the LSTM to the vocabulary size. The forward method takes an input sequence and a hidden state and returns the model's prediction for the next word in the sequence and the updated hidden state. The model can be trained and saved to a file using the given hyperparameters.

```python
class GramNet(nn.Module):
    def __init__(self,vocab_size, n_hidden=256, n_layers=4,embedding_dim=200, dropout=None, lr=0.001,model_save_path='.',device='cuda'):
        super().__init__()
        self.dropout = dropout
        self.n_layers = n_layers
        self.n_hidden = n_hidden
        self.lr = lr
        self.model_save_path = model_save_path
        self.device = device
        self.vocab_size = vocab_size
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        if dropout is not None:
            self.rnn = nn.LSTM(embedding_dim, n_hidden, n_layers, dropout=dropout,batch_first=True)
        else:
            self.rnn = nn.LSTM(embedding_dim, n_hidden, n_layers,batch_first=True)
            dropout = 0
        self.fc = nn.Linear(n_hidden, vocab_size)
        self.model_name = 'GramNet_'+str(n_hidden)+'_'+str(n_layers)+'_'+str(dropout)+'_'+str(lr)+'.pt'

    def forward(self, x, hidden):
        embedded = self.embedding(x)
        out, hidden = self.rnn(embedded, hidden)
#         out = self.dropout(out)
        out = out.reshape(-1, self.n_hidden)
        out = self.fc(out)
        return out, hidden
```

2. **Improved Model:** Using two LSTM

It is a modified version of the previous model. One LSTM is used to encode the Language ID of the word using tags and the other to predict the word. It takes as input the vocabulary size, number of hidden units, number of LSTM layers, embedding dimension, dropout rate, learning rate, model save path, and device. Additionally, it takes two input sequences x and x_type, and two hidden states hidden_x and hidden_t, for two separate LSTMs that process each input sequence. It initializes two LSTM layers with the given hyperparameters, an embedding layer for each input sequence, and a linear layer to map the concatenated output of the LSTMs to the vocabulary size. The forward method takes the two input sequences and the two hidden states and returns the model's prediction for the next word in the sequence, as well as the updated hidden states for each LSTM. The concatenated output of the two LSTMs is passed through the linear layer to produce the final output.

```python
class GramNet(nn.Module):
    def __init__(self,vocab_size, n_hidden= 9, n_layers=4,embedding_dim=200, dropout=None, lr=0.001,model_save_path='.',device='cuda'):
        super().__init__()
        self.dropout = dropout
        self.n_layers = n_layers
        self.n_hidden = n_hidden
        self.lr = lr
        self.model_save_path = model_save_path
        self.device = device
        self.vocab_size = vocab_size
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        if dropout is not None:
            self.rnn_1 = nn.LSTM(embedding_dim, n_hidden, n_layers, dropout=dropout,batch_first=True)
            self.rnn_2 = nn.LSTM(embedding_dim, n_hidden, n_layers, dropout=dropout,batch_first=True)
        else:
            self.rnn_1 = nn.LSTM(embedding_dim, n_hidden, n_layers,batch_first=True)
            self.rnn_2 = nn.LSTM(embedding_dim, n_hidden, n_layers,batch_first=True)
            dropout = 0
        self.fc = nn.Linear(n_hidden, vocab_size)
        self.model_name = 'GramNet_'+str(n_hidden)+'_'+str(n_layers)+'_'+str(dropout)+'_'+str(lr)+'.pt'

    def forward(self, x, x_type, hidden_x, hidden_t):
        embedded_x = self.embedding(x)
        embedded_t = self.embedding(x_type)
        out_x, hidden_x = self.rnn_1(embedded_x, hidden_x)
        out_t, hidden_t = self.rnn_2(embedded_t, hidden_t)
        out_x = out_x.reshape(-1, self.n_hidden)
        out_t = out_t.reshape(-1, self.n_hidden)
        out = torch.cat((out_x, out_t), 0)
        out = self.fc(out)
        return out, hidden_x, hidden_t
```

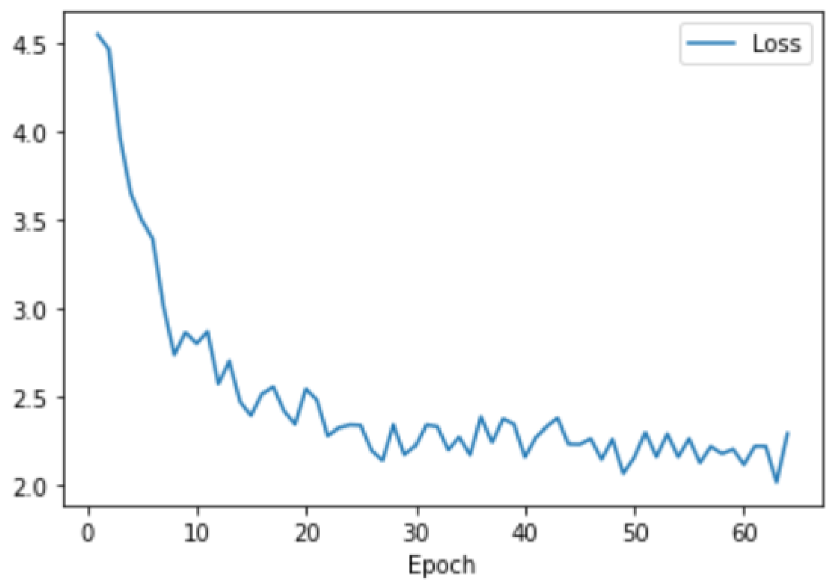
4. Model Training & Evaluation:

Hyper Parameters	Tuned Values
NGRAM	5
N_LAYERS	3
EMBEDDING DIMENSION	200
HIDDEN DIMENSION	512
DROPOUT	0.2

<u>Baseline Model</u>

Loss v/s Epochs:

	Epoch	Loss
0	1	4.553792
1	2	4.471577
2	3	3.970032
3	4	3.652445
4	5	3.500349
5	6	3.398028
6	7	3.014066
7	8	2.737580
8	9	2.865598
9	10	2.801889
10	11	2.871608
11	12	2.575084
12	13	2.703486
13	14	2.473367
14	15	2.394268
15	16	2.518132
16	17	2.558898
17	18	2.420319
18	19	2.345225
19	20	2.546688



Perplexity Scores:

Train Dataset	2.6851668370395037
Validation Dataset	216.5501515989133

Mix Factor (MF):
44.86507936507939

Generated Sentences:

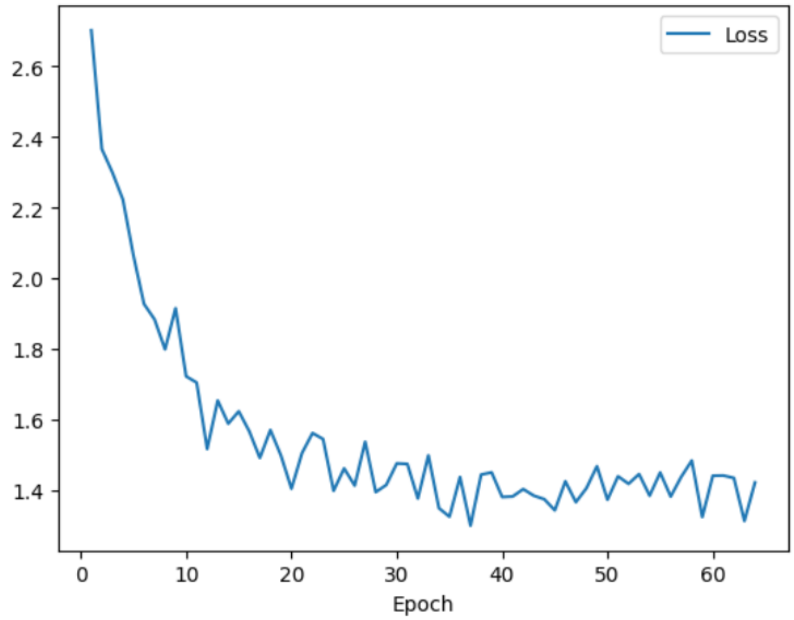
Seed	**Generated**
you	you can mein character ka unique hai to kya hum is
me abhi	me abhi bhi yanhi nahi hota achyar bhi toy story dekhi usme
hi me	hi me se he jo app over par can interesting sound kartha
me sochta	me sochta nahi hu but <number> ghante kahana hoga aur voh

	enigma
life me always	life me always aisa hi lagta tha wo karke jo usko mila wo

Improved Model:

Loss v/s Epochs:

	Epoch	Loss
0	1	2.702647
1	2	2.366146
2	3	2.299887
3	4	2.222867
4	5	2.064351
5	6	1.927063
6	7	1.883364
7	8	1.798456
8	9	1.914871
9	10	1.721832
10	11	1.704024
11	12	1.515794
12	13	1.653252
13	14	1.587753
14	15	1.622585
15	16	1.565682
16	17	1.490455
17	18	1.569543
18	19	1.497452
19	20	1.403382



Perplexity Scores:

Train Dataset	2.4366996327589905
Validation Dataset	6.972607726896829

Mix Factor (MF):
50.10277777777779

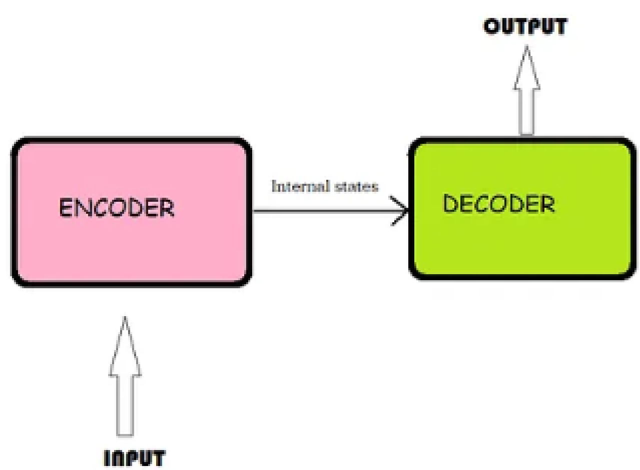
Generated Sentences

Seed	**Generated**
you	you seen waise mai use bar dekh sakta hun ki unpredictable
me abhi	me abhi tak same cheej he jitni kuch der pahale ye movie
hi me	hi me nahi dekhi main sochta hoon ke wo lucky ho raha
me sochta	me sochta hu wwe nahi karte even if they dont recall the
life me always	life me always wonder karti hai yah use itna trouble kiya <end> ne

PART II : Code Mixed Translation

Encoder Decoder Architecture

The encoder-decoder model is a way of using recurrent neural networks for seq-to-seq prediction problems. It was initially developed for machine translation problems, although it has proven successful at related sequence-to-sequence prediction problems such as text summarization and question answering. The approach involves two recurrent neural networks, one to encode the input sequence, called the encoder, and a second to decode the encoded input sequence into the target sequence called the decoder. In our implementation we will be using LSTMs instead of RNNs.



It consists of 3 parts: encoder, intermediate vector and decoder.

Encoder:It accepts a single element of the input sequence at each time step, process it, collects information for that element and propagates it forward.

Intermediate vector: This is the final internal state produced from the encoder part of the model. It contains information about the entire input sequence to help the decoder make accurate predictions.

Decoder: given the entire sentence, it predicts an output at each time step.

BLEU Score

The Bilingual Evaluation Understudy Score, or BLEU for short, is a metric for evaluating a generated sentence to a reference sentence.The score was developed for evaluating the predictions made by automatic machine translation systems. It is not perfect, but does offer 5 compelling benefits:

- It is quick and inexpensive to calculate.
- It is easy to understand.
- It is language independent.
- It correlates highly with human evaluation.
- It has been widely adopted.

1. Data Collection:

LinCE : For this project we are using the LINCE dataset.
https://ritual.uh.edu/lince/datasets

2. Data Preprocessing:

The preprocessing code defines several functions that are used to preprocess text data for this natural language processing task. The read_data() function reads a text file and preprocesses each line of text using the other functions defined in the code.
The replace_dates() function replaces dates in various formats with the string <DATE>. The replace_concurrent_punctuation() function replaces sequences of two or more consecutive punctuation characters with a single space. The replace_hash_tags() function replaces hashtags with the string <HASHTAG>.
The remove_special_characters() function removes any special characters that are not punctuation marks. The remove_extra_spaces() function removes any extra spaces from the text. The replace_hyphenated_words() function replaces hyphenated words with words separated by a space.
Finally, the read_data() function reads each line of text from a file and preprocesses it using the other functions stated above.

It then tokenizes the preprocessed text using the basic_english tokenizer from the nltk library and returns a list of tokenized sentences.

3. Model

Here we have created an Encoder - Decoder model with LSTM as the baseline model for the translation task.The code defines a sequence-to-sequence model for machine translation using LSTMs, consisting of an EncoderLSTM, a DecoderLSTM, and a Seq2Seq class that integrates them. The EncoderLSTM takes a sequence of input tokens and returns the final hidden and cell states, which are then used as input to the DecoderLSTM. The DecoderLSTM generates the output sequence token by token using the previous token, the hidden state, and the cell state as input. The Seq2Seq class combines the EncoderLSTM and DecoderLSTM to perform the translation. During training, the output sequence is generated by feeding the ground truth tokens back into the decoder, while during inference, the decoder generates the output sequence using the predicted tokens.

```
class Seq2Seq(nn.Module):
  def __init__(self, Encoder_LSTM, Decoder_LSTM):
    super(Seq2Seq, self).__init__()
    self.Encoder_LSTM = Encoder_LSTM
    self.Decoder_LSTM = Decoder_LSTM

  def forward(self, source, target, tfr=0.5):
    batch_size = source.shape[1]
    target_len = target.shape[0]
    target_vocab_size = len(TRG.vocab)
    outputs = torch.zeros(target_len, batch_size, target_vocab_size).to(device)
    hidden_state, cell_state = self.Encoder_LSTM(source)
    x = target[0]
    for i in range(1, target_len):
      output, hidden_state, cell_state = self.Decoder_LSTM(x, hidden_state, cell_state)
      outputs[i] = output
      best_guess = output.argmax(1)
      x = target[i] if random.random() < tfr else best_guess
    return outputs
```

Encoder

```
class EncoderLSTM(nn.Module):
  def __init__(self, input_size, embedding_size, hidden_size, num_layers, p):
    super(EncoderLSTM, self).__init__()
    self.hidden_size = hidden_size
    self.num_layers = num_layers
    self.dropout = nn.Dropout(p)
    self.tag = True
    self.embedding = nn.Embedding(input_size, embedding_size)
    self.LSTM = nn.LSTM(embedding_size, hidden_size, num_layers, dropout = p)

  def forward(self, x):
    embedding = self.dropout(self.embedding(x))
    outputs, (hidden_state, cell_state) = self.LSTM(embedding)
    return hidden_state, cell_state
```

Decoder

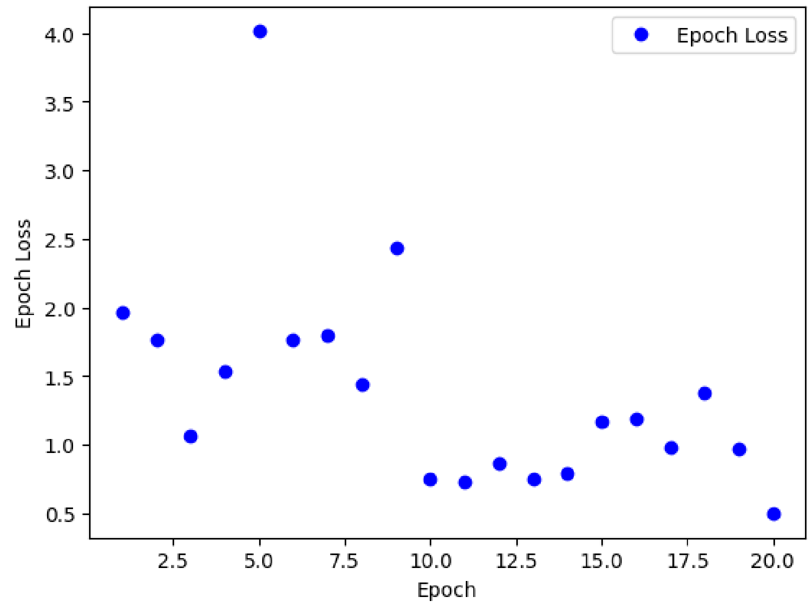
```
class DecoderLSTM(nn.Module):
  def __init__(self, input_size, embedding_size, hidden_size, num_layers, p, output_size):
    super(DecoderLSTM, self).__init__()
    self.hidden_size = hidden_size
    self.num_layers = num_layers
    self.output_size = output_size
    self.dropout = nn.Dropout(p)
    self.embedding = nn.Embedding(input_size, embedding_size)
    self.LSTM = nn.LSTM(embedding_size, hidden_size, num_layers, dropout = p)
    self.fc = nn.Linear(hidden_size, output_size)

  # Shape of x (32) [batch_size]
  def forward(self, x, hidden_state, cell_state):
    x = x.unsqueeze(0)
    embedding = self.dropout(self.embedding(x))
    outputs, (hidden_state, cell_state) = self.LSTM(embedding, (hidden_state, cell_state))
    predictions = self.fc(outputs)
    predictions = predictions.squeeze(0)
    return predictions, hidden_state, cell_state
```

4. Model Training & Evaluation:

Hyper Parameters	Tuned Values
NGRAM	5
N_LAYERS	3
EMBEDDING DIMENSION	200
HIDDEN DIMENSION	512
DROPOUT	0.2

Loss v/s Epoch

Epochs	Loss
1	1.9689141511917114
2	1.7652822732925415
3	1.062523365020752
4	1.53066086769104
5	4.017604351043701
6	1.7632946968078613
7	1.7939538955688477
8	1.436180591583252
9	2.4345710277557373
10	0.7464218735694885
11	0.7255614995956421
12	0.8602990508079529
13	0.7542532682418823
14	0.786764919757843
15	1.1678128242492676
16	1.1889249086380005
17	0.9799388647079468
18	1.3740839958190918
19	0.9729033708572388
20	0.49897074699401855



BLEU SCORE: 1.38

Translated Sentences:

English Sentence	Translated Sentence
Alright that is fine. What is the movie?	interesting hai kya ye ek long movie hai?
I have not seen that one either	maine kabhi nahi dekhi
may be worth watching!	do box bhi man!

Future Work :

1. **Generation :**
    a. Use Transformers.
    b. Using context based embeddings like BERT, ELMo
2. **Translation:**
    a. Apply attention to Encoder – Decoder Models.
    b. Use Transformers and various pretrained models available.

References:

1. https://colah.github.io/posts/2015-08-Understanding-LSTMs/
2. https://medium.com/analytics-vidhya/machine-translation-encoder-decoder-model-7e4867377161
3. https://towardsdatascience.com/perplexity-in-language-models-87a196019a94
4. https://www.kdnuggets.com/2020/07/pytorch-lstm-text-generation-tutorial.html