Generative Chatbot

- Prepared by Zhu Lin Ch'ng, Avinash Fernando, Matthew Stevenson and Thomas Whiteley for the University of Liverpool CSCK507 Natural Language Processing and Understanding June 2023 group project.
- This notebook implements a seq-2-seq recurrent neural network architecture comparing the performance with and without the use of Luong Attention.
- The detailed training logs can be accessed at https://tensorboard.dev/experiment/5hXpB3jpRWqqWu3HnrfXGw/
- The model weights and demonstration can be accessed at https://huggingface.co/spaces/csck507/Seq2Seq

Table of Contents

- Importing Libraries
- Data Preprocessing
  - Importing the dataset
  - Preparing the dataset
    - Read Data
    - Preprocess Data and structure it
    - Load the structured data into dataframe and index it
    - Create tensors
    - Split and batch the data
- Model Development
  - Building the seq2seq model
  - Training the seq2seq model
- Model Evaluation
  - Using the seq2seq models
  - Evaluating the seq2seq models
    - BLEU Score
    - Cosine Similarity

Importing Libraries

This notebook uses the following 3rd party libraries:

- pytorch: machine learning library
- pandas: data analysis library
- numpy: mathematical function library
- spacy: natural language processing library
- nltk: natural language processing library

In [ ]:
```
import io
import json
import time
import os
import random
import re
import tarfile
import unicodedata
import zipfile
from io import open
from typing import Tuple

import numpy as np
import pandas as pd
import requests
import spacy
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
```

```
from torch.utils.data import DataLoader, TensorDataset, random_split
from torch.utils.tensorboard import SummaryWriter
from torch import optim
from nltk.translate.bleu_score import corpus_bleu
from sentence_transformers import SentenceTransformer
from scipy.spatial.distance import cosine
```

In [ ]:
```
# Initialise the torch device, favouring GPU if available
spacy.prefer_gpu()
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")


# Load english dictionary for spacy
try:
    spacy.load('en_core_web_sm')
except LookupError:
    print('Run: python -m spacy download en_core_web_sm')
```

Data Preprocessing

The Ubuntu dialogue corpus is selected to provide conversational data for training

Importing the dataset

In [ ]:
```
def download_file(url, dir):
    """
    Download file from url
    :param url: url of file
    :param filename: name of file
    :return: None
    """
    r = requests.get(url)
    if url.endswith('.tar.gz'):
        z = tarfile.open(fileobj=io.BytesIO(r.content), mode="r:gz")
        z.extractall(dir)
        z.close()
    elif url.endswith('.zip'):
        z = zipfile.ZipFile(io.BytesIO(r.content))
        z.extractall(dir)
    else:
        print('Unknown file type')
    return None

def extract_zip(filename, dir):
    """
    Extract zip file
    :param filename: name of file
    :return: None
    """
    z = zipfile.ZipFile(filename)
    z.extractall(dir)
    return None
```

In [ ]:
```
datasets = {'ubuntu-dialogue': 'data/ubuntu dialogue.zip'}
```

In [ ]:
```
# Create directory
if not os.path.exists('data'):
    os.makedirs('data')

# Check if data is already downloaded
for dataset, source in datasets.items():
    if os.path.exists('data/' + dataset):
        print(dataset + ' already exists')
    elif dataset == 'ubuntu-dialogue':
        ubuntu = 'data/ubuntu dialogue'
        if os.path.exists(source):
            os.makedirs(ubuntu)
            extract_zip(source, ubuntu)
            os.remove(source)
            print(dataset + ' extracted')
        elif os.path.exists(ubuntu):
```

```
            print(dataset + ' already exists')
        else:
            kag = 'https://www.kaggle.com/datasets/rtatman/ubuntu-dialogue-corpus/download?datasetVersio
            print(f'Manually download ubuntu dialogue dataset from {kag} and place in data folder')
    else:
        download_file(source, 'data')
        print(dataset + ' downloaded')
```

```
ubuntu-dialogue already exists
```

Preparing the dataset

The Ubuntu dataset is presented as dialogues with 3 entries with varying formats of questions (Q) and answers (A) including:

- QQQ
- AAA
- QQA
- QAA
- QAQ

These formats are parsed to produce quesion answer pairs as follows:

- QQQ -> ignored
- AAA -> ignored
- QQA -> (Q+Q) - A
- QAA -> Q - (A+A)
- QAQ -> Q - A, A - Q

Read Data

In [ ]:
```
variants = {'small':'',
            'medium':'_196',
            'large':'_301'}
ubuntufile = f'data/ubuntu dialogue/ubuntu-dialogue-corpus/dialogueText{variants["small"]}.csv'
text_df = pd.read_csv(ubuntufile)
text_df['dialogueID'] = text_df['dialogueID'].apply(lambda x: int(x.split('.')[0]))
print(text_df.shape)
```

```
(1038324, 6)
```

In [ ]:
```
# preview text from ubuntu dialogue dataset
text_df.head()
```

Out[ ]:

	folder	dialogueID	date	from	to	text
0	3	126125	2008-04-23T14:55:00.000Z	bad_image	NaN	Hello folks, please help me a bit with the fol...
1	3	126125	2008-04-23T14:56:00.000Z	bad_image	NaN	Did I choose a bad channel? I ask because you ...
2	3	126125	2008-04-23T14:57:00.000Z	lordleemo	bad_image	the second sentence is better english and we...
3	3	64545	2009-08-01T06:22:00.000Z	mechtech	NaN	Sock Puppe?t
4	3	64545	2009-08-01T06:22:00.000Z	mechtech	NaN	WTF?

In [ ]:
```
# reduce df size
# text_df = text_df[:50000]
# text_df.head()
```

Preprocess Data and structure it

In [ ]:
```
nlp = spacy.load('en_core_web_sm') # load spacy model
```

In [ ]:
```python
def unicodetoascii(text):
    """
    Turn a Unicode string to plain ASCII

    :param text: text to be converted
    :return: text in ascii format
    """
    normalized_text = unicodedata.normalize('NFKD', str(text))
    ascii_text = ''.join(char for char in normalized_text if unicodedata.category(char) != 'Mn')
    return ascii_text

def preprocess_text(text, fn=unicodetoascii):

    text = fn(text)
    text = text.lower()
    text = re.sub(r'http\S+', '', text)
    text = re.sub(r'[^\x00-\x7F]+', "", text) # Remove non-ASCII characters
    text = re.sub(r"(\w)[!?]+(\w)", r'\1\2', text) # Remove !? between words
    text = re.sub(r"\s\s+", r" ", text).strip() # Remove extra spaces
    return text

def parse_dialogue(data):
    dialogues = {}
    df = data.copy()
    df.reset_index(inplace=True)
    # Group by dialogueID
    for dialogue_id, group in df.groupby('dialogueID'):
        sentence_pairs = {}
        context = ''
        previous_direction = (None, None)
        for i, row in group.iterrows():
            idx = row['index']
            sender = row['from']
            recipient = row['to']
            response = str(row['text'])
            direction = (sender, recipient)

            if direction == previous_direction:
                # add to the response to the previous message if the current message is consecutive
                prev_idx = idx - 1
                while prev_idx not in sentence_pairs:
                    prev_idx -= 1
                response = context + ' ' + response
                sentence_pairs[prev_idx] = (sentence_pairs[prev_idx][0], response)
                # sentence_pairs[-1] = (sentence_pairs[-1][0], response)
            elif (direction == previous_direction[::-1]) or (previous_direction[1] == None) and (directi
                # if the current message is from the previous recipient to the previous sender
                # if the previous message did not have a recipient, but the current message is to the pr
                sentence_pairs[idx]=(context, response)
            else:
                sentence_pairs[idx]=(context, response)

            previous_direction = tuple(direction)
            context = str(response) # response is the context for the next message
        # remove the sentence pairs that does not have context but only responses
        sentence_pairs = {k: v for k, v in sentence_pairs.items() if v[0] != ''}
        dialogues[dialogue_id] = sentence_pairs

    return dialogues
```

In [ ]:
```python
text_df['text'] = text_df['text'].apply(preprocess_text)
text_df.head()
```

Out[ ]:

	folder	dialogueID	date	from	to	text
0	3	126125	2008-04-23T14:55:00.000Z	bad_image	NaN	hello folks, please help me a bit with the fol...
1	3	126125	2008-04-23T14:56:00.000Z	bad_image	NaN	did i choose a bad channel? i ask because you ...
2	3	126125	2008-04-23T14:57:00.000Z	lordleemo	bad_image	the second sentence is better english and we a...
3	3	64545	2009-08-01T06:22:00.000Z	mechtech	NaN	sock puppet
4	3	64545	2009-08-01T06:22:00.000Z	mechtech	NaN	wtf?

In [ ]:
```
dialogues = parse_dialogue(text_df)
```

In [ ]:
```
# convert nested dictionary to dataframe
def dict_to_df(data):
    rows = []
    for dialogue_id, sentence_pairs in data.items():
        for idx, pair in sentence_pairs.items():
            rows.append([dialogue_id, idx, pair[0], pair[1]])
    df = pd.DataFrame(rows, columns=['dialogueID', 'index', 'context', 'response'])
    return df

dialogue_df = dict_to_df(dialogues)
dialogue_df.head()
```

Out[ ]:

	dialogueID	index	context	response
0	1	456667	also guys, i'm trying to get into my firefox p...	are you logged in as 'root' ?
1	1	456668	are you logged in as 'root' ?	no.
2	2	936173	lifeless : no, but i have had trouble printing...	arhh, i should point my windows machine at the...
3	3	788303	gosh it's late my brains not working what's th...	tar xf blah.tar
4	4	670258	i have to install hoary in server mode because...	6^

In [ ]:
```
dialogue_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 432502 entries, 0 to 432501
Data columns (total 4 columns):
 #   Column      Non-Null Count   Dtype 
---  ------      --------------   ----- 
 0   dialogueID  432502 non-null  int64 
 1   index       432502 non-null  int64 
 2   context     432502 non-null  object
 3   response    432502 non-null  object
dtypes: int64(2), object(2)
memory usage: 13.2+ MB
```

Load the structured data into dataframe and index it

- As the model contains a layer of embedding, lemmatization is not required. (words that are similar will have vectors that are close to each other)
  - https://aclanthology.org/2021.nodalida-main.25/
  - https://aclanthology.org/2021.nodalida-main.25.pdf
- As one of the models will be using attention, removing stop words is not required. (attention will learn to ignore them)

In [ ]:
```
# use torch text to create vocabulary
def tokenize(text, nlp=nlp):
```

```
    """
    Tokenize text
    :param text: text to be tokenized
    :return: list of tokens
    """
    return [tok.text for tok in nlp.tokenizer(text)]

def create_mapping(df, tokenize=tokenize):
    """
    Create vocabulary mapping from context and response dataframes
    :param df_context: context dataframe
    :param df_response: response dataframe
    :param tokenize: tokenization function
    :return: vocabulary mapping
    """
    # Create vocabulary mapping
    vocab = set()
    default_tokens = ['<pad>', '<bos>', '<eos>', '<unk>']
    start_index = len(default_tokens)
    for context, response in zip(df['context'], df['response']):
        vocab.update(tokenize(context))
        vocab.update(tokenize(response))
    word2idx = {word: start_index+idx for idx, word in enumerate(vocab)}
    idx2word = {start_index+idx: word for idx, word in enumerate(vocab)}
    for idx, token in enumerate(default_tokens):
        word2idx[token] = idx
        idx2word[idx] = token
    return word2idx, idx2word

def lookup_words(idx2word, indices):
    """
    Lookup words from indices
    :param idx2word: index to word mapping
    :param indices: indices to be converted
    :return: list of words
    """
    return [idx2word[idx] for idx in indices]
```

In [ ]:
```
word2idx, idx2word = create_mapping(dialogue_df)
word2idx['<pad>']
```

Out[ ]:
```
0
```

In [ ]:
```
# Map words to indices
dialogue_df['context_idx'] = dialogue_df['context'].apply(lambda x: [word2idx[word] for word in tokenize
dialogue_df['response_idx'] = dialogue_df['response'].apply(lambda x: [word2idx[word] for word in tokeni
```

In [ ]:
```
# add bos and eos tokens to context and response
bos = word2idx['<bos>']
eos = word2idx['<eos>']
dialogue_df['context_idx'] = dialogue_df['context_idx'].apply(lambda x: [bos] + x + [eos])
dialogue_df['response_idx'] = dialogue_df['response_idx'].apply(lambda x: [bos] + x + [eos])
```

In [ ]:
```
dialogue_df.head()
```

Out[ ]:

	dialogueID	index	context	response	context_idx	response_idx
0	1	456667	also guys, i'm trying to get into my firefox p...	are you logged in as 'root' ?	[1, 52110, 106586, 141735, 19079, 32908, 94235...	[1, 24873, 112112, 101731, 154669, 159790, 105...
1	1	456668	are you logged in as 'root' ?	no.	[1, 24873, 112112, 101731, 154669, 159790, 105...	[1, 116460, 50180, 2]
2	2	936173	lifeless : no, but i have had trouble printing...	arhh, i should point my windows machine at the...	[1, 144960, 30968, 116460, 141735, 96323, 1907...	[1, 137852, 141735, 19079, 70554, 141721, 1405...
3	3	788303	gosh it's late my brains not working what's th...	tar xf blah.tar	[1, 43384, 22524, 29425, 36278, 140538, 75262,...	[1, 4174, 141889, 116729, 2]
4	4	670258	i have to install hoary in server mode because...	6^	[1, 19079, 154214, 124236, 84473, 61065, 15466...	[1, 137030, 2]

In [ ]:
```
# save word2idx and idx2word
files_to_save = ['vocab/word2idx.json', 'vocab/idx2word.json']
if not os.path.exists('vocab'):
    os.makedirs('vocab')

for file in files_to_save:
    if os.path.exists(file):
        os.remove(file)
    with open(file, 'w') as f:
        if file == 'vocab/word2idx.json':
            json.dump(word2idx, f)
        elif file == 'vocab/idx2word.json':
            json.dump(idx2word, f)
```

Create tensors

In [ ]:
```
# Create tensors with sos, pad, eos tokens
def create_tensors(df, max_len=20, min_len=1):
    """
    Create tensors with sos, pad, eos tokens
    :param df: dataframe with context and response
    :param max_len: maximum length of sequence
    :return: tensors with sos, pad, eos tokens
    """
    # Create tensors
    context_tensor = torch.zeros((len(df), max_len), dtype=torch.long)
    response_tensor = torch.zeros((len(df), max_len), dtype=torch.long)
    for i, (context, response) in enumerate(zip(df['context_idx'], df['response_idx'])):
        # Trim context and response
        if len(context) < max_len and len(context) >= min_len and len(response) < max_len and len(respon
            context_tensor[i, :len(context)] = torch.tensor(context, dtype=torch.long)
            response_tensor[i, :len(response)] = torch.tensor(response, dtype=torch.long)
    # remove rows with all zeros
    context_tensor = context_tensor[~(context_tensor == 0).all(1)]
    response_tensor = response_tensor[~(response_tensor == 0).all(1)]

    return context_tensor, response_tensor
```

In [ ]:
```
# get max length of context and response
max_len_context = max(dialogue_df['context_idx'].apply(len))
max_len_response = max(dialogue_df['response_idx'].apply(len))
max_len = max(max_len_context, max_len_response)
max_len = 12 # override max_len to reduce training time
min_len = 3
print(f'Maximum length of sequence: {max_len}', f'Minimum length of sequence: {min_len}')
```

```
Maximum length of sequence: 12 Minimum length of sequence: 3
```

- A filter is applied to remove sentences that are too long or too short.
- Remove sentences that are too long is required as it will take a long time to train the model.

In [ ]:
```
context_tensor, response_tensor = create_tensors(dialogue_df, max_len=max_len, min_len=min_len)
print(context_tensor.shape, response_tensor.shape)
print(f'Total data size: {context_tensor.shape[0]}')
```

```
torch.Size([71866, 12]) torch.Size([71866, 12])
Total data size: 71866
```

Split and batch the data

Data is batched to reduce memory overhead and improve the speed of training. Data presented in the tensor is padded to be of equal lenght as demonstrated below:

[<bos>,can,you,help,me,with,a,support,issue,<eos>,<pad>,<pad>,<pad>,<pad>,<pad>]

In [ ]:
```
class ContextReponseBatch:
    def __init__(self, data):
        transposed_data = list(zip(*data))
        self.input = torch.stack(transposed_data[0], 0)
        # self.input_mask = (self.input != 0)
        self.target = torch.stack(transposed_data[1], 0)
        # self.target_mask = (self.target != 0)

    def pin_memory(self):
        """
        Pin memory for faster data transfer to GPU
        :return: self
        """
        self.input = self.input.pin_memory()
        # self.input_mask = self.input_mask.pin_memory()
        self.target = self.target.pin_memory()
        # self.target_mask = self.target_mask.pin_memory()
        return self

def collate_wrapper(batch):
    """
    Wrapper for collate function
    :param batch: batch of data
    :return: ContextReponseBatch object
    """
    return ContextReponseBatch(batch)
```

In [ ]:
```
# Split data into train, validation, and test sets
def split_data(context_tensor, response_tensor, train_ratio=0.8, val_ratio=0.1, test_ratio=0.1):
    """
    Split data into train, validation, and test sets
    :param context_tensor: context tensor
    :param response_tensor: response tensor
    :param train_ratio: ratio of train set
    :param val_ratio: ratio of validation set
    :param test_ratio: ratio of test set
    :return: train, validation, and test sets
    """
    # Split data into train, validation, and test sets
    dataset = TensorDataset(context_tensor, response_tensor)
    train_size = int(train_ratio * len(dataset))
    val_size = int(val_ratio * len(dataset))
    test_size = len(dataset) - train_size - val_size
    train_set, val_set, test_set = random_split(dataset, [train_size, val_size, test_size], generator=to
    return train_set, val_set, test_set

# Batch data
def batch_data(train_set, val_set, test_set, batch_size=8, fn=collate_wrapper):
    """
    Batch data
    :param train_set: train set
    :param val_set: validation set
    :param test_set: test set
    :param batch_size: batch size
    :return: train, validation, and test loaders
    """
    # Batch data
    train_loader = DataLoader(train_set, batch_size=batch_size, shuffle=True, collate_fn=collate_wrapper
    val_loader = DataLoader(val_set, batch_size=batch_size, shuffle=True, collate_fn=collate_wrapper)
```

```
    test_loader = DataLoader(test_set, batch_size=batch_size, shuffle=True, collate_fn=collate_wrapper)
    return train_loader, val_loader, test_loader
```

In [ ]:
```
# Create train, validation, and test sets
train_set, val_set, test_set = split_data(context_tensor, response_tensor)

# Batch data
train_loader, val_loader, test_loader = batch_data(train_set, val_set, test_set, batch_size=16)
```

In [ ]:
```
# preview shape of batch
next(iter(train_loader)).input.shape # (batch_size, max_len)
```

Out[ ]:
```
torch.Size([16, 12])
```

Model Development

The seq-2-seq model is based on a reference implementation provided by pytorch for language transaltion https://pytorch.org/tutorials/beginner/torchtext_translation_tutorial.html. It include the following key components:

- Gated recurrent unit (GRU) RNN encoder
- GRU RNN decoder
- Luong attention

Gradient clipping is used to avoid exploding gradients, and dropout layers are used to avoid overfitting.

Building the seq2seq model

In [ ]:
```
def init_weights(m):
    """
    Initialize weights
    :param m: model
    :return: None
    """
    for name, param in m.named_parameters():
        if 'weight' in name:
            nn.init.xavier_uniform_(param.data)
        elif 'bias' in name:
            nn.init.constant_(param.data, 0)
    return None
```

In [ ]:
```
class Encoder(nn.Module):
    """
    GRU RNN Encoder
    """
    def __init__(self,
                 input_dim: int,
                 emb_dim: int,
                 enc_hid_dim: int,
                 dec_hid_dim: int,
                 dropout: float = 0):
        super(Encoder, self).__init__()

        # dimension of imput
        self.input_dim = input_dim
        # dimension of embedding layer
        self.emb_dim = emb_dim
        # dimension of encoding hidden layer
        self.enc_hid_dim = enc_hid_dim
        # dimension of decoding hidden layer
        self.dec_hid_dim = dec_hid_dim

        # create embedding layer use to train embedding representations of the corpus
        self.embedding = nn.Embedding(input_dim, emb_dim)

        # use GRU for RNN
        self.rnn = nn.GRU(emb_dim, enc_hid_dim, bidirectional=True, batch_first=False, num_layers=1)
        self.fc = nn.Linear(enc_hid_dim * 2, dec_hid_dim)
        # create dropout layer which will help produce a more generalisable model
        self.dropout = nn.Dropout(dropout)
```

```
    def forward(self, src: torch.Tensor) -> Tuple[torch.Tensor, torch.Tensor]:
        # apply dropout to the embedding layer
        embedded = self.dropout(self.embedding(src))
        # generate an output and hidden layer from the rnn
        outputs, hidden = self.rnn(embedded)
        hidden = torch.tanh(self.fc(torch.cat((hidden[-2, :, :], hidden[-1, :, :]), dim=1)))
        return outputs, hidden


class Attention(nn.Module):
    """
    Luong attention
    """
    def __init__(self,
                 enc_hid_dim: int,
                 dec_hid_dim: int,
                 attn_dim: int):
        super(Attention, self).__init__()

        # dimension of encoding hidden layer
        self.enc_hid_dim = enc_hid_dim
        # dimension of decoding hidden layer
        self.dec_hid_dim = dec_hid_dim
        self.attn_in = (enc_hid_dim * 2) + dec_hid_dim

        self.attn = nn.Linear(self.attn_in, attn_dim)

    def forward(self,
                decoder_hidden: torch.Tensor,
                encoder_outputs: torch.Tensor) -> torch.Tensor:

        src_len = encoder_outputs.shape[0]
        repeated_decoder_hidden = decoder_hidden.unsqueeze(1).repeat(1, src_len, 1)
        encoder_outputs = encoder_outputs.permute(1, 0, 2)
        # Luong attention
        energy = torch.tanh(self.attn(torch.cat((repeated_decoder_hidden, encoder_outputs), dim=2)))
        attention = torch.sum(energy, dim=2)

        return F.softmax(attention, dim=1)


class AttnDecoder(nn.Module):
    """
    GRU RNN Decoder with attention
    """
    def __init__(self,
                 output_dim: int,
                 emb_dim: int,
                 enc_hid_dim: int,
                 dec_hid_dim: int,
                 attention: nn.Module,
                 dropout: float = 0):
        super(AttnDecoder, self).__init__()

        # dimention of output layer
        self.output_dim = output_dim
        # dimention of embedding layer
        self.emb_dim = emb_dim
        # dimention of encoding hidden layer
        self.enc_hid_dim = enc_hid_dim
        # dimention of decoding hidden layer
        self.dec_hid_dim = dec_hid_dim
        # drouput rate
        self.dropout = dropout
        # attention layer
        self.attention = attention

        # create embedding layer use to train embedding representations of the corpus
        self.embedding = nn.Embedding(output_dim, emb_dim)
        # use GRU for RNN
        self.rnn = nn.GRU((enc_hid_dim * 2) + emb_dim, dec_hid_dim, batch_first=False, num_layers=1)
        self.out = nn.Linear(self.attention.attn_in + emb_dim, output_dim)
        self.dropout = nn.Dropout(dropout)
```

```
    def encode_attention(self,
                             decoder_hidden: torch.Tensor,
                             encoder_outputs: torch.Tensor) -> torch.Tensor:

        a = self.attention(decoder_hidden, encoder_outputs)
        a = a.unsqueeze(1)
        encoder_outputs = encoder_outputs.permute(1, 0, 2)
        weighted_encoder_rep = torch.bmm(a, encoder_outputs)
        weighted_encoder_rep = weighted_encoder_rep.permute(1, 0, 2)
        return weighted_encoder_rep

    def forward(self,
                input: torch.Tensor,
                decoder_hidden: torch.Tensor,
                encoder_outputs: torch.Tensor) -> Tuple[torch.Tensor, torch.Tensor]:

        input = input.unsqueeze(0)
        # apply dropout to embedding layer
        embedded = self.dropout(self.embedding(input))
        weighted_encoder = self.encode_attention(decoder_hidden, encoder_outputs)

        # generate an output and hidden layer from the rnn
        rnn_input = torch.cat((embedded, weighted_encoder), dim=2)
        output, decoder_hidden = self.rnn(rnn_input, decoder_hidden.unsqueeze(0))

        embedded = embedded.squeeze(0)
        output = output.squeeze(0)
        weighted_encoder = weighted_encoder.squeeze(0)
        output = self.out(torch.cat((output, weighted_encoder, embedded), dim=1))
        return output, decoder_hidden.squeeze(0)

class Decoder(nn.Module):
    """
    GRU RNN Decoder without attention
    """
    def __init__(self,
                 output_dim: int,
                 emb_dim: int,
                 enc_hid_dim: int,
                 dec_hid_dim: int,
                 dropout: float = 0):
        super(Decoder, self).__init__()

        # dimention of output layer
        self.output_dim = output_dim
        # dimention of embedding layer
        self.emb_dim = emb_dim
        # dimention of encoding hidden layer
        self.enc_hid_dim = enc_hid_dim
        # dimention of decoding hidden layer
        self.dec_hid_dim = dec_hid_dim
        # drouput rate
        self.dropout = dropout

        # create embedding layer use to train embedding representations of the corpus
        self.embedding = nn.Embedding(output_dim, emb_dim)
        # GRU RNN
        self.rnn = nn.GRU((enc_hid_dim * 2) + emb_dim, dec_hid_dim, batch_first=False, num_layers=1)
        self.out = nn.Linear((enc_hid_dim * 2) + dec_hid_dim + emb_dim, output_dim)
        self.dropout = nn.Dropout(dropout)

    def forward(self,
                input: torch.Tensor,
                decoder_hidden: torch.Tensor,
                encoder_outputs: torch.Tensor) -> Tuple[torch.Tensor
                                                        , torch.Tensor]:

        input = input.unsqueeze(0)
        # apply dropout to embedding layer
        embedded = self.dropout(self.embedding(input))
        context = encoder_outputs[-1,:,:]
        context = context.repeat(embedded.shape[0], 1, 1)
        embs_and_context = torch.cat((embedded, context), -1)
        # generate an output and hidden layer from the rnn
        output, decoder_hidden = self.rnn(embs_and_context, decoder_hidden.unsqueeze(0))
```

```
        embedded = embedded.squeeze(0)
        output = output.squeeze(0)
        context = context.squeeze(0)
        output = self.out(torch.cat((output, embedded, context), -1))
        return output, decoder_hidden.squeeze(0)

class Seq2Seq(nn.Module):
    """
    Seq-2-Seq model combining RNN encoder and RNN decoder
    """
    def __init__(self,
                 encoder: nn.Module,
                 decoder: nn.Module,
                 device: torch.device):
        super(Seq2Seq, self).__init__()

        self.encoder = encoder
        self.decoder = decoder
        self.device = device

    def forward(self,
                src: torch.Tensor,
                trg: torch.Tensor,
                teacher_forcing_ratio: float = 0.5) -> torch.Tensor:
        src = src.transpose(0, 1) # (max_len, batch_size)
        trg = trg.transpose(0, 1) # (max_len, batch_size)
        batch_size = src.shape[1]
        max_len = trg.shape[0]
        trg_vocab_size = self.decoder.output_dim

        outputs = torch.zeros(max_len, batch_size, trg_vocab_size).to(self.device)
        encoder_outputs, hidden = self.encoder(src)

        # first input to the decoder is the <sos> token
        output = trg[0,:]

        for t in range(1, max_len):
            output, hidden = self.decoder(output, hidden, encoder_outputs)
            outputs[t] = output
            teacher_force = random.random() < teacher_forcing_ratio
            top1 = output.max(1)[1]
            output = trg[t] if teacher_force else top1

        return outputs
```

In [ ]:
```
test_batch = next(iter(train_loader))
test_batch.input.shape, test_batch.target.shape
```

Out[ ]:
```
(torch.Size([16, 12]), torch.Size([16, 12]))
```

In [ ]:
```
# enc = Encoder(input_dim=len(word2idx), emb_dim=256, enc_hid_dim=512, dec_hid_dim=512)
# attn = Attention(enc_hid_dim=512, dec_hid_dim=512, attn_dim=64)
# dec = AttnDecoder(output_dim=len(word2idx), emb_dim=256, enc_hid_dim=512, dec_hid_dim=512, attention=a
# model = Seq2Seq(encoder=enc, decoder=dec, device=device)
# model.apply(init_weights)
# model.to(device)
# model.train()
# optimizer = optim.Adam(model.parameters(), lr=0.001)
# optimizer.zero_grad()
# model(test_batch.input.to(device), test_batch.target.to(device), teacher_forcing_ratio=0.5).shape
```

In [ ]:
```
print(len(word2idx))
```

```
162004
```

In [ ]:
```
# enc = Encoder(input_dim=len(word2idx), emb_dim=256, enc_hid_dim=512, dec_hid_dim=512)
# dec = Decoder(output_dim=len(word2idx), emb_dim=256, enc_hid_dim=512, dec_hid_dim=512)
# model = Seq2Seq(encoder=enc, decoder=dec, device=device)
# model.apply(init_weights)
# model.to(device)
# model.train()
# optimizer = optim.Adam(model.parameters(), lr=0.001)
# optimizer.zero_grad()
# model(test_batch.input.to(device), test_batch.target.to(device), teacher_forcing_ratio=0.5).shape
```

Training the seq2seq model

In [ ]:
```
PAD_INDEX = word2idx['<pad>']
criterion = nn.CrossEntropyLoss(ignore_index=PAD_INDEX)

def train_model(model, train_loader, val_loader, optimizer,
                run_name = 'seq2seq', init_weights=init_weights, device=device,
                n_epochs=10, clip=1, criterion=criterion,
                teacher_forcing_ratio=0.5, params=None):
    """
    Train model
    :param model: model
    :param train_loader: train loader
    :param val_loader: validation loader
    :param optimizer: optimizer
    :param n_epochs: number of epochs
    :param clip: clip
    :param criterion: loss function
    :param PAD_INDEX: index for pad token
    :param teacher_forcing_ratio: teacher forcing ratio
    :return: model, train loss, validation loss
    """
    if not os.path.exists('models'):
        os.makedirs('models')
    model = model.to(device)
    model.apply(init_weights)
    hyperparams = {'n_epochs': n_epochs,
                   'clip': clip,
                   'teacher_forcing_ratio': teacher_forcing_ratio}
    if params:
        hyperparams.update(params)
    writer = SummaryWriter(f'runs/{run_name}')
    train_loss = []
    val_loss = []
    for epoch in range(n_epochs):
        model.train()
        start_time = time.time()
        epoch_loss = 1
        val_epoch_loss = 1
        for i, batch in enumerate(train_loader):
            src = batch.input.to(device)
            trg = batch.target.to(device)

            optimizer.zero_grad()
            output = model(src, trg, teacher_forcing_ratio)

            output_dim = output.shape[-1]
            output = output[1:].view(-1, output_dim)
            trg = trg.transpose(0, 1)
            trg = trg[1:].reshape(-1)

            loss = criterion(output, trg)
            loss.to(device).backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), clip) # clip gradients
            optimizer.step() # update parameters
            epoch_loss += loss.item() # update epoch loss
            writer.add_scalar('Train Loss', loss.item(), epoch * len(train_loader) + i)
        train_loss.append(epoch_loss / len(train_loader))
        # save model with datetime and epoch
        torch.save(model.state_dict(), f'models/{run_name}_epoch{epoch+1}.pt')
        # remove previous model
        if epoch > 0:
            os.remove(f'models/{run_name}_epoch{epoch}.pt')
        model.eval()
        with torch.no_grad():
            for i, batch in enumerate(val_loader):
                src = batch.input.to(device)
                trg = batch.target.to(device)

                output = model(src, trg, teacher_forcing_ratio)
                output_dim = output.shape[-1]
                output = output[1:].view(-1, output_dim)
                trg = trg.transpose(0, 1)
```

```
                trg = trg[1:].reshape(-1)

                loss = criterion(output, trg)
                val_epoch_loss += loss.item()
        val_loss.append(val_epoch_loss / len(val_loader))
        writer.add_scalar('Validation Loss', val_epoch_loss / len(val_loader), epoch)
        duration = time.time()-start_time
        remaining = duration * (n_epochs - epoch - 1)
        print(f'Epoch: {epoch+1:02} | Time: {duration:.3f}s | Train Loss: {epoch_loss/len(train_loader):
    writer.add_hparams(hyperparams, {'hparam/train_loss': train_loss[-1], 'hparam/val_loss': val_loss[-1
    writer.close()
    return model, train_loss, val_loss
```

In [ ]:
```
params = {'input_dim': len(word2idx),
          'emb_dim': 192,
          'enc_hid_dim': 256,
          'dec_hid_dim': 256,
          'dropout': 0.5,
          'attn_dim': 64,
          'teacher_forcing_ratio': 0.5,
          'epochs': 35}

batch_size = 18
```

In [ ]:
```
# Create train, validation, and test sets
train_set, val_set, test_set = split_data(context_tensor, response_tensor)

# Batch data
train_loader, val_loader, test_loader = batch_data(train_set, val_set, test_set, batch_size=batch_size)

print(f'Training set size: {len(train_set)}', f'Validation set size: {len(val_set)}', f'Test set size: {
```

```
Training set size: 57492 Validation set size: 7186 Test set size: 7188
```

In [ ]:
```
enc = Encoder(input_dim=params['input_dim'], emb_dim=params['emb_dim'], enc_hid_dim=params['enc_hid_dim'
dec = Decoder(output_dim=params['input_dim'], emb_dim=params['emb_dim'], enc_hid_dim=params['enc_hid_dim
model = Seq2Seq(encoder=enc, decoder=dec, device=device)
num_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f'Model has {num_params:,} trainable parameters')
optimizer = optim.Adam(model.parameters())

norm_model, train_loss, val_loss = train_model(model, train_loader, val_loader, optimizer,
                                               run_name=f'NormSeq2Seq-{int(num_params/1e6)}M', n_epochs=
                                               teacher_forcing_ratio=params['teacher_forcing_ratio'],
                                               params=params)
```

```
Model has 219,456,724 trainable parameters
Epoch: 01 | Time: 409.775s | Train Loss: 5.456 | Val Loss: 5.150 | Remaining: 232m 12s
Epoch: 02 | Time: 409.705s | Train Loss: 4.895 | Val Loss: 5.067 | Remaining: 225m 20s
Epoch: 03 | Time: 410.125s | Train Loss: 4.655 | Val Loss: 5.019 | Remaining: 218m 44s
Epoch: 04 | Time: 410.241s | Train Loss: 4.430 | Val Loss: 5.063 | Remaining: 211m 57s
Epoch: 05 | Time: 410.065s | Train Loss: 4.207 | Val Loss: 5.141 | Remaining: 205m 2s
Epoch: 06 | Time: 409.643s | Train Loss: 3.973 | Val Loss: 5.240 | Remaining: 197m 60s
Epoch: 07 | Time: 409.830s | Train Loss: 3.774 | Val Loss: 5.392 | Remaining: 191m 15s
Epoch: 08 | Time: 409.824s | Train Loss: 3.586 | Val Loss: 5.486 | Remaining: 184m 25s
Epoch: 09 | Time: 409.562s | Train Loss: 3.419 | Val Loss: 5.561 | Remaining: 177m 29s
Epoch: 10 | Time: 410.796s | Train Loss: 3.287 | Val Loss: 5.660 | Remaining: 171m 10s
Epoch: 11 | Time: 410.472s | Train Loss: 3.189 | Val Loss: 5.777 | Remaining: 164m 11s
Epoch: 12 | Time: 410.119s | Train Loss: 3.082 | Val Loss: 5.810 | Remaining: 157m 13s
Epoch: 13 | Time: 410.192s | Train Loss: 3.009 | Val Loss: 5.880 | Remaining: 150m 24s
Epoch: 14 | Time: 410.212s | Train Loss: 2.933 | Val Loss: 5.933 | Remaining: 143m 34s
Epoch: 15 | Time: 409.900s | Train Loss: 2.877 | Val Loss: 5.978 | Remaining: 136m 38s
Epoch: 16 | Time: 410.961s | Train Loss: 2.818 | Val Loss: 5.996 | Remaining: 130m 8s
Epoch: 17 | Time: 410.931s | Train Loss: 2.784 | Val Loss: 6.052 | Remaining: 123m 17s
Epoch: 18 | Time: 411.111s | Train Loss: 2.726 | Val Loss: 6.120 | Remaining: 116m 29s
Epoch: 19 | Time: 411.122s | Train Loss: 2.690 | Val Loss: 6.130 | Remaining: 109m 38s
Epoch: 20 | Time: 411.253s | Train Loss: 2.657 | Val Loss: 6.201 | Remaining: 102m 49s
Epoch: 21 | Time: 410.999s | Train Loss: 2.624 | Val Loss: 6.244 | Remaining: 95m 54s
Epoch: 22 | Time: 411.017s | Train Loss: 2.598 | Val Loss: 6.247 | Remaining: 89m 3s
Epoch: 23 | Time: 410.866s | Train Loss: 2.550 | Val Loss: 6.295 | Remaining: 82m 10s
Epoch: 24 | Time: 410.783s | Train Loss: 2.537 | Val Loss: 6.284 | Remaining: 75m 19s
Epoch: 25 | Time: 410.660s | Train Loss: 2.518 | Val Loss: 6.342 | Remaining: 68m 27s
Epoch: 26 | Time: 410.904s | Train Loss: 2.481 | Val Loss: 6.387 | Remaining: 61m 38s
Epoch: 27 | Time: 411.135s | Train Loss: 2.468 | Val Loss: 6.429 | Remaining: 54m 49s
Epoch: 28 | Time: 411.056s | Train Loss: 2.453 | Val Loss: 6.412 | Remaining: 47m 57s
Epoch: 29 | Time: 411.269s | Train Loss: 2.430 | Val Loss: 6.471 | Remaining: 41m 8s
Epoch: 30 | Time: 410.948s | Train Loss: 2.432 | Val Loss: 6.443 | Remaining: 34m 15s
Epoch: 31 | Time: 411.371s | Train Loss: 2.408 | Val Loss: 6.512 | Remaining: 27m 25s
Epoch: 32 | Time: 410.756s | Train Loss: 2.382 | Val Loss: 6.515 | Remaining: 20m 32s
Epoch: 33 | Time: 410.915s | Train Loss: 2.365 | Val Loss: 6.514 | Remaining: 13m 42s
Epoch: 34 | Time: 411.123s | Train Loss: 2.349 | Val Loss: 6.544 | Remaining: 6m 51s
Epoch: 35 | Time: 411.183s | Train Loss: 2.341 | Val Loss: 6.583 | Remaining: 0m 0s
```

In [ ]:

```
enc = Encoder(input_dim=params['input_dim'], emb_dim=params['emb_dim'], enc_hid_dim=params['enc_hid_dim'
attn = Attention(enc_hid_dim=params['enc_hid_dim'], dec_hid_dim=params['dec_hid_dim'], attn_dim=params['
dec = AttnDecoder(output_dim=params['input_dim'], emb_dim=params['emb_dim'], enc_hid_dim=params['enc_hid
model = Seq2Seq(encoder=enc, decoder=dec, device=device)
num_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f'Model has {num_params:,} trainable parameters')
optimizer = optim.Adam(model.parameters())

attn_model, train_loss, val_loss = train_model(model, train_loader, val_loader, optimizer,
                                               run_name=f'AttnSeq2Seq-{int(num_params/1e6)}M', n_epochs=
                                               teacher_forcing_ratio=params['teacher_forcing_ratio'],
                                               params=params)
```

```
Model has 219,505,940 trainable parameters
Epoch: 01 | Time: 416.459s | Train Loss: 5.428 | Val Loss: 5.075 | Remaining: 235m 60s
Epoch: 02 | Time: 416.565s | Train Loss: 4.868 | Val Loss: 5.010 | Remaining: 229m 7s
Epoch: 03 | Time: 416.400s | Train Loss: 4.580 | Val Loss: 5.030 | Remaining: 222m 5s
Epoch: 04 | Time: 416.621s | Train Loss: 4.318 | Val Loss: 5.074 | Remaining: 215m 15s
Epoch: 05 | Time: 416.376s | Train Loss: 4.043 | Val Loss: 5.193 | Remaining: 208m 11s
Epoch: 06 | Time: 415.696s | Train Loss: 3.771 | Val Loss: 5.344 | Remaining: 200m 55s
Epoch: 07 | Time: 415.791s | Train Loss: 3.520 | Val Loss: 5.454 | Remaining: 194m 2s
Epoch: 08 | Time: 415.745s | Train Loss: 3.321 | Val Loss: 5.593 | Remaining: 187m 5s
Epoch: 09 | Time: 416.094s | Train Loss: 3.150 | Val Loss: 5.690 | Remaining: 180m 18s
Epoch: 10 | Time: 416.417s | Train Loss: 3.024 | Val Loss: 5.756 | Remaining: 173m 30s
Epoch: 11 | Time: 416.098s | Train Loss: 2.905 | Val Loss: 5.847 | Remaining: 166m 26s
Epoch: 12 | Time: 416.880s | Train Loss: 2.821 | Val Loss: 5.909 | Remaining: 159m 48s
Epoch: 13 | Time: 416.091s | Train Loss: 2.736 | Val Loss: 5.990 | Remaining: 152m 34s
Epoch: 14 | Time: 416.460s | Train Loss: 2.673 | Val Loss: 6.080 | Remaining: 145m 46s
Epoch: 15 | Time: 416.391s | Train Loss: 2.612 | Val Loss: 6.117 | Remaining: 138m 48s
Epoch: 16 | Time: 415.961s | Train Loss: 2.565 | Val Loss: 6.174 | Remaining: 131m 43s
Epoch: 17 | Time: 415.869s | Train Loss: 2.513 | Val Loss: 6.236 | Remaining: 124m 46s
Epoch: 18 | Time: 415.797s | Train Loss: 2.476 | Val Loss: 6.268 | Remaining: 117m 49s
Epoch: 19 | Time: 416.003s | Train Loss: 2.437 | Val Loss: 6.315 | Remaining: 110m 56s
Epoch: 20 | Time: 416.273s | Train Loss: 2.406 | Val Loss: 6.390 | Remaining: 104m 4s
Epoch: 21 | Time: 414.519s | Train Loss: 2.373 | Val Loss: 6.410 | Remaining: 96m 43s
Epoch: 22 | Time: 413.022s | Train Loss: 2.348 | Val Loss: 6.478 | Remaining: 89m 29s
Epoch: 23 | Time: 413.122s | Train Loss: 2.330 | Val Loss: 6.454 | Remaining: 82m 37s
Epoch: 24 | Time: 413.658s | Train Loss: 2.304 | Val Loss: 6.520 | Remaining: 75m 50s
Epoch: 25 | Time: 413.456s | Train Loss: 2.286 | Val Loss: 6.562 | Remaining: 68m 55s
Epoch: 26 | Time: 413.281s | Train Loss: 2.260 | Val Loss: 6.541 | Remaining: 61m 60s
Epoch: 27 | Time: 413.164s | Train Loss: 2.245 | Val Loss: 6.612 | Remaining: 55m 5s
Epoch: 28 | Time: 413.426s | Train Loss: 2.228 | Val Loss: 6.630 | Remaining: 48m 14s
Epoch: 29 | Time: 414.064s | Train Loss: 2.208 | Val Loss: 6.661 | Remaining: 41m 24s
Epoch: 30 | Time: 416.876s | Train Loss: 2.206 | Val Loss: 6.720 | Remaining: 34m 44s
Epoch: 31 | Time: 416.488s | Train Loss: 2.170 | Val Loss: 6.720 | Remaining: 27m 46s
Epoch: 32 | Time: 416.957s | Train Loss: 2.173 | Val Loss: 6.707 | Remaining: 20m 51s
Epoch: 33 | Time: 417.002s | Train Loss: 2.160 | Val Loss: 6.755 | Remaining: 13m 54s
Epoch: 34 | Time: 417.000s | Train Loss: 2.147 | Val Loss: 6.824 | Remaining: 6m 57s
Epoch: 35 | Time: 417.396s | Train Loss: 2.138 | Val Loss: 6.802 | Remaining: 0m 0s
```

Model Evaluation

The model evaluated against the following metrics:

- Bleu score: measures the similarity of the input text to the validation set
- Cosine similarity: measures the cosine similarity of the embedding representation of text to the validation set

Using the seq2seq models

In [ ]:

```python
def generate(model, sentence, max_len=30, word2idx=word2idx, idx2word=idx2word, device=device, tokenize=
    """
    Generate response
    :param model: model
    :param sentence: sentence
    :param max_len: maximum length of sequence
    :param word2idx: word to index mapping
    :param idx2word: index to word mapping
    :return: response
    """
    model.eval()
    sentence = preprocess_text(sentence)
    tokens = tokenize(sentence)
    tokens = [word2idx[token] if token in word2idx else word2idx['<unk>'] for token in tokens]
    tokens = [word2idx['<bos>']] + tokens + [word2idx['<eos>']]
    tokens = torch.tensor(tokens, dtype=torch.long).unsqueeze(1).to(device)
    outputs = [word2idx['<bos>']]
    with torch.no_grad():
        encoder_outputs, hidden = model.encoder(tokens)
    for t in range(max_len):
        output, hidden = model.decoder(torch.tensor([outputs[-1]], dtype=torch.long).to(device), hidden,
        top1 = output.max(1)[1]
        outputs.append(top1.item())
        if top1.item() == word2idx['<eos>']:
            break
```

```
    response = lookup_words(idx2word, outputs)
    return response
```

In [ ]:
```
# enc = Encoder(input_dim=params['input_dim'], emb_dim=params['emb_dim'], enc_hid_dim=params['enc_hid_di
# dec = Decoder(output_dim=params['input_dim'], emb_dim=params['emb_dim'], enc_hid_dim=params['enc_hid_d
# norm_model = Seq2Seq(encoder=enc, decoder=dec, device=device)
# norm_model.load_state_dict(torch.load('models/NormSeq2Seq-188M_epoch35.pt'))
# norm_model.to(device)
# print(f'Model has {sum(p.numel() for p in model.parameters() if p.requires_grad):,} trainable parameter
```

In [ ]:
```
test = 'is ubuntu good?'
norm_test = generate(norm_model, test, max_len=12, word2idx=word2idx, idx2word=idx2word, device=device)
' '.join(norm_test).replace('<bos>', '').replace('<eos>', '').strip()
```

Out[ ]:
```
'i i do i i massive swap ?'
```

In [ ]:
```
test = 'is ubuntu good?'
attn_test = generate(attn_model, test, max_len=12, word2idx=word2idx, idx2word=idx2word, device=device)
' '.join(attn_test).replace('<bos>', '').replace('<eos>', '').strip()
```

Out[ ]:
```
'maybe sent you'
```

Evaluating the seq2seq models

In [ ]:
```
def tensor_to_text(tensor, idx2word=idx2word):
    """
    Convert tensor to text
    :param tensor: tensor
    :param idx2word: index to word mapping
    :return: text
    """
    textlist = tensor.tolist()
    text = lookup_words(idx2word, textlist)
    # remove default tokens
    text = [word for word in text if word not in ['<bos>', '<eos>', '<pad>', '<unk>']]
    return text

def predict_from_tensor(model, input_tensor, max_len=12, word2idx=word2idx, idx2word=idx2word, device=de
    """
    Predict response
    :param model: model
    :param input_tensor: input tensor
    :param max_len: maximum length of sequence
    :param word2idx: word to index mapping
    :param idx2word: index to word mapping
    :return: response
    """
    model.eval()
    input_tensor = input_tensor.unsqueeze(1).to(device)
    outputs = [word2idx['<bos>']]
    with torch.no_grad():
        encoder_outputs, hidden = model.encoder(input_tensor)
    for t in range(max_len):
        output, hidden = model.decoder(torch.tensor([outputs[-1]], dtype=torch.long).to(device), hidden,
        top1 = output.max(1)[1]
        outputs.append(top1.item())
        if top1.item() == word2idx['<eos>']:
            break
    response = lookup_words(idx2word, outputs)
    return response
```

BLEU Score

In [ ]:
```
def eval_pair(model, data, max_len=12,
              word2idx=word2idx,
              idx2word=idx2word, device=device,
              tensor_to_text=tensor_to_text,
              predict_from_tensor=predict_from_tensor):
    predicted = []
    actual = []
    for c, r in data.dataset:
```

```
        predict = predict_from_tensor(model, c, max_len=max_len, word2idx=word2idx, idx2word=idx2word, d
        predict = [word for word in predict if word not in ['<bos>', '<eos>', '<pad>', '<unk>']]
        predicted.append([predict])
        actual.append(tensor_to_text(r))
    return predicted, actual
```

In [ ]:
```
attn_predicted, attn_actual = eval_pair(attn_model, test_loader, max_len=12)
norm_predicted, norm_actual = eval_pair(norm_model, test_loader, max_len=12)
```

In [ ]:
```
attn_bleu = corpus_bleu(attn_predicted, attn_actual)
print(f'BLEU score for Seq2Seq with Attention: {attn_bleu*100:.2f}')
```

```
BLEU score for Seq2Seq with Attention: 1.46
```

In [ ]:
```
norm_bleu = corpus_bleu(norm_predicted, norm_actual)
print(f'BLEU score for Seq2Seq without Attention: {norm_bleu*100:.2f}')
```

```
BLEU score for Seq2Seq without Attention: 1.29
```

Cosine Similarity

- Cosine similarity is used to measure the similarity between two sentences.
- First, sentence embeddings are created by encoding using a pre-trained model (BERT).
- Next, the cosine similarity is calculated between the sentence embeddings.

In [ ]:
```
def list_to_sent(list_of_words):
    """
    Convert list of words to sentence
    :param list_of_words: list of words
    :return: sentence
    """
    return ' '.join(list_of_words)
```

In [ ]:
```
sbert = SentenceTransformer('bert-base-nli-mean-tokens')
```

In [ ]:
```
def get_cosines(list_pred, act, sbert=sbert, list_to_sent=list_to_sent):
    """
    Get cosine similarity scores
    :param list_pred: list of list of predicted sentences [[a],[b]]
    :param act: list of actual sentences [a,b]
    :param sbert: sentence transformer model
    :param list_to_sent: function to convert list of words to sentence
    :return: cosine similarity scores
    """
    sent_pred = [val for sublist in list_pred for val in sublist]
    sent_pred = [list_to_sent(sent) for sent in sent_pred]
    sent_act = [list_to_sent(sent) for sent in act]
    cosine_scores = []
    for pred, act in zip(sent_pred, sent_act):
        pred = sbert.encode(pred)
        act = sbert.encode(act)
        cosine_scores.append(cosine(pred, act))
    cosine_scores = np.array(cosine_scores)
    return cosine_scores
```

In [ ]:
```
attn_cosine = get_cosines(attn_predicted, attn_actual)
norm_cosine = get_cosines(norm_predicted, norm_actual)

print(f'Average cosine similarity for Seq2Seq with Attention: {np.mean(attn_cosine):.2f}')
print(f'Average cosine similarity for Seq2Seq without Attention: {np.mean(norm_cosine):.2f}')
```

```
Average cosine similarity for Seq2Seq with Attention: 0.40
Average cosine similarity for Seq2Seq without Attention: 0.40
```