Improving Dialog Systems with Pre-trained Models

Rajat Gupta (19BM6JP17) , Karthikeya Racharla (19BM6JP32)
Vineet Kumar (19BM6JP46) , Paturu Harish (19BM6JP55)
Indian Institute of Technology
Kharagpur, WB 721302

1 Introduction

Dialogue systems, also known as interactive conversational agents, virtual agents and sometimes chatter bots, are used in a wide set of applications ranging from technical support services to language learning tools and entertainment. Our work is about exploring whether dialog systems can be improved with regards to context and natural language relevance through pre-training or not.

1.1 Related Works

Developing intelligent chat-bots and dialog systems is of great significance to both commercial implications. A good chat-bot agent enables enterprises to provide automatic customer services and thus reduce human labor.

Li et al. (2017) developed a unique multi-turn data set 'DailyDialog', which reflect realistic communication ways that's rich in emotion, such as understanding the underlying semantics of user inputs and generating coherent meaningful responses. Its applications are proposed significant in developing intelligent chat-bots and dialog systems for both commercial and academic purposes.

Sordoni et al. (2015) proposed a novel hierarchical recurrent encoder-decoder architecture (HRED) For generating Context-Aware Query Suggestions. The notion used was that in a query session, the query co-occurrence in the same session is a strong signal of query relatedness and can be straightforwardly used to produce suggestions.

2 Motivation

Commonly, to design a chat-bot or dialog system, one needs an extremely large amount of data to train a model on because it is important to infer the context that the dialog interlocutors share. However, often is the case that one does not have a good size of training data (or large number of dialogs in the data set) and hence, the model will not be able to train well to give desired results. Hence, to overcome this limitation, we are trying to explore if we can improve our dialog system through fine-tuning on a pre-trained model. The data set that we are using is DailyDialog data set.

3 Data set

DailyDialog is a multi-turn dialog data set that contains conversations about daily life. Some of its salient features are:

Submitted as Course Project for CS60078 Complex Networks 2020.

- Its language is human-written and hence more formal and less noisy compared to data sets like Twitter Dialog Corpus, which are constructed by posts and replies on social networks
- With 8 turns on an average, this data set is implied to be more suitable for training compact conversational models
- The conversations in DailyDialog often focus on a certain topic and under a certain physical context
- The dialogues of the data set reflect our daily communication and cover various topics of our daily life, viz. exchanging information and enhancing social bonding

The data set has original distribution as follows:

SET	NUMBER OF SENTENCES	NUMBER OF DIALOGS
Training	87,170	11,119
Validation	8,069	1,001
Testing	7,740	1,001

Table 1: Dataset Subdivison

4 Data Pre-Processing

4.1 For pre-training Auto-encoder model

The data in its original form is in a raw text file with _eou_ as the delimiter between any two sentences. We have used *Microsoft Access* to convert it into a Tabular Data set (.csv file) with each row containing a single sentence. The Pre-processing code is given in the Appendix 1.

4.2 For HRED model

As can be seen from the above table1, the number of dialogs in the training data set is not much. Hence, we created additional samples for each dialog by keeping $<U_1....U_{t-1}>$ as context and $<U_t>$ (where U_t is the utterance) as the corresponding response, for t varying from 3 to the length of each dialog. The Pre-processing code is given in the Appendix 2.

5 Evaluation Metrics

5.1 PERPLEXITY

In information theory, perplexity is a measurement of how well a probability distribution or probability model predicts a sample. A low perplexity indicates that the probability distribution is good at predicting the sample. Perplexity is a way of evaluating language models in natural language processing as a language model is a probability distribution over entire sentences or texts.

For probabilistic language models, word perplexity is a well-established performance metric (Bengio et al. 2003; Mikolov et al. 2010) and has been suggested for generative dialogue models previously. Perplexity explicitly measures the model's ability to account for the syntactic structure of the dialogue (e.g. turn-taking) and the syntactic structure of each utterance (e.g. punctuation marks) as it always measures the probability of regenerating the exact reference utterance. Also, optimizing probabilistic models using word perplexity has shown promising results in several machine learning tasks including statistical machine translation.

5.2 BLEU

BLEU (Bilingual Evaluation Understudy) is an algorithm for evaluating the quality of text which has been machine-translated from one natural language to another. Quality here refers to the correspondence between a machine's output and human's output.

BLEU looks at the overlap in the predicted and actual target sequences in terms of their n-grams. There are two ways to calculate BLEU:

- Corpus BLEU – Judging quality at corpus level
- Sentence BLEU – Judging quality of individual sentences

6 Methodology/Line of Work

The following steps give a broad overview of our work.

- Training a Vanilla (standard) Seq2Seq RNN Auto-encoder in PyTorch to be used as the pre-training model for HRED model
- Evaluating the model through Perplexity Score and BLUE Score metrics
- Modifying the original architecture by adding LSTM context encoder to make it a Hierarchical Recurrent Encoder Decoder (HRED) model
- Comparing the HRED model trained with and without pre-trained weights using the metrics, mentioned above

7 Training vanilla RNN Auto-Encoder

We have used a standard RNN Encoder-Decoder model that translates German sentences to English. The link to its GitHub code is: `https://github.com/bentrevett/pytorch-seq2seq` The concept behind a general encoder-decoder is that there is an *encoder* function that maps the input space to a different latent space, followed by a *decoder* function that maps the latent space to a different target space. As an example, in the original case, mapping a sequence of tokens in German to a sequence of tokens in English. However, since our intention is not to predict but simply to recall the input, we are considering the model as an auto-encoder where the target space is same as input space and also the target is equal to input.

7.1 Model Architecture

The following explains the architecture of the auto-encoder used. The model has 25,255,976 trainable parameters.

```
Seq2Seq(
  (encoder): Encoder(
    (embedding): Embedding(12328, 256)
    (rnn): GRU(256, 512)
    (dropout): Dropout(p=0.5, inplace=False)
  )
  (decoder): Decoder(
    (embedding): Embedding(12328, 256)
    (rnn): GRU(768, 512)
    (fc_out): Linear(in_features=1280, out_features=12328, bias=True)
    (dropout): Dropout(p=0.5, inplace=False)
  )
)
```

Figure 1: Seq2Seq Architecture

7.2 Teacher Forcing

Teacher forcing is a method for quickly and efficiently training recurrent neural networks that uses model output from a prior time step as an input. Teacher forcing works by using the actual or expected output from the training data set at the current time step y(t) as input in the next time step X(t+1), rather than the output generated by the network.

This is implemented in the architecture using *Teacher Forcing Ratio*, some probability set in prior, we use the current target word as the decoder's next input rather than using the decoder's current guess. This technique acts as training wheels for the decoder, aiding in more efficient training. Teacher forcing ratio of **0.5** is used in our model.

7.3 Model Evaluation

The model is trained until saturation. The tables 2 and 3 explains the model performance.

EPOCH #	Training Loss	Validation Loss	Training PPL	Validation PPL
1	1.571	2.049	4.812	7.762
2	1.326	1.978	3.767	7.231
3	1.160	1.909	3.191	6.743
4	1.031	1.813	2.804	6.128
5	0.931	1.768	2.536	5.860

Table 2: Training and Validation Results for RNN Auto-Encoder

Test Loss	Test PPL
1.839	6.292

Table 3: Test Results for RNN Auto-Encoder

7.4 Predicted Outputs by model

We consider three example sentences one from each Training, Validation and Testing set:

Training Set

- **Input:** [' ', 'really', '?', 'i', 'think', 'that', '"s', 'impossible', '!']
- **Prediction:** [' ', 'really', '?', 'i', 'think', 'that', '"s', 'absolutely', '!', '<eos>']

Validation Set

- **Input:** [' ', 'mmmm', '...', 'if', 'i', 'come', 'in', 'and', 'collect', 'it', 'this', 'afternoon', ',', 'is', 'there', 'any', 'way', 'i', 'could', 'use', 'it', 'today', '?', 'petty', 'cash', 'is', 'getting', 'low', ',', 'so', 'i', 'need', 'to', 'draw', 'some', 'money', '.']
- **Prediction:** [' ', 'basically', '...', 'if', 'i', 'come', 'in', 'in', 'this', 'is', ',', 'is', 'it', 'is', ',', 'i', 'i', 'need', 'it', 'is', 'to', 'this', 'afternoon', ',', 'and', 'i', 'i', 'i', 'some', 'money', ',', 'i', 'i', 'need', 'money', 'money', '.', '<eos>']

Test Set

- **Input:** [' ', 'mainly', 'because', 'we', "'ve", 'invested', 'in', 'a', 'heat', 'recovery', 'system', '.']
- **Prediction:** [' ', 'because', 'because', 'we', "'ve", 'lived', 'in', 'a', 'few', 'or', 'system', '.', '<eos>']

BLEU Score We have evaluated our model in terms of BLEU in three ways. The results are given in the table 4

Description	Score
Corpus Bleu	45.27
Sentence Bleu	59.11
Sentence Bleu with Smoothing	53.96

Table 4: BLEU Scores using RNN Auto-Encoder

8 Hierarchical Encoder Decoder Model

8.1 Theory

Since a dialogue can be seen as a sequence of utterances which, in turn, are sequences of tokens, we can use an HRED to model the hierarchy of sequence with two RNNs – one at the utterance level and another at the token level.

In dialogue, the encoder RNN maps each utterance to an utterance vector. The utterance vector is the hidden state obtained after the last token of the utterance has been processed. The higher-level context RNN keeps track of past utterances by processing iteratively each utterance vector. The next utterance prediction is performed by means of a decoder RNN, which takes the hidden state of the context RNN and produces a probability distribution over the tokens in the next utterance.

We have tried many implementations of HRED available on GitHub but since it is very difficult to find a model that has the exact similar architecture as of our original model since we would need to feed the pre-trained weights later, we have tried implementing HRED by modifying the Seq2Seq auto-encoder architecture.

The HRED model is updated with the pre-trained weight dictionary for the keys that match in it – the ones associated with the original encoder and decoder. The source code for the implementation can be found in the Appendix at Page 12.

8.2 Context Encoder

To implement an HRED, we have added a LSTM which serves as the context encoder. This RNN encodes the temporal structure of the utterances appearing so far in the dialogue allowing information and gradients to flow over longer time spans.

The overall process of HRED is explained as -

- Encoding all utterances in context using encoder to get utterance vectors
- These utterance vectors are fed to LSTM to obtain a single context vector
- The context vector is then fed to the decoder to generate the dialog response

The architecture is as shown below. The model has 27,357,224 trainable parameters.

8.3 Model Evaluation

8.3.1 HRED without pre-trained weights

The table 5 shows the results of final 5 epochs (trained for 30 epochs) of the HRED model without pre-training and table 6 shows the results on test data set.

The plots for losses and perplexity scores are as shown below:

```
Seq2Seq(
  (encoder): Encoder(
    (embedding): Embedding(12328, 256)
    (rnn): GRU(256, 512)
    (dropout): Dropout(p=0.5, inplace=False)
  )
  (con_enc): Context_Encoder(
    (rnn): LSTM(512, 512)
    (dropout): Dropout(p=0.2, inplace=False)
  )
  (decoder): Decoder(
    (embedding): Embedding(12328, 256)
    (rnn): GRU(768, 512)
    (fc_out): Linear(in_features=1280, out_features=12328, bias=True)
    (dropout): Dropout(p=0.5, inplace=False)
  )
)
```

Figure 2: Seq2Seq Architecture

EPOCH #	Training Loss	Validation Loss	Training PPL	Validation PPL
1	4.186	5.624	65.738	276.998
2	4.175	5.637	65.056	280.667
3	4.159	5.636	64.031	280.204
4	4.148	5.662	63.287	287.615
5	4.122	5.630	61.674	278.755

Table 5: Training and Validation Set Results for HRED model (without pre-trained weights)

Test Loss	Test PPL
5.649	284.113

Table 6: Testing Set Results for HRED model (without pre-trained weights)

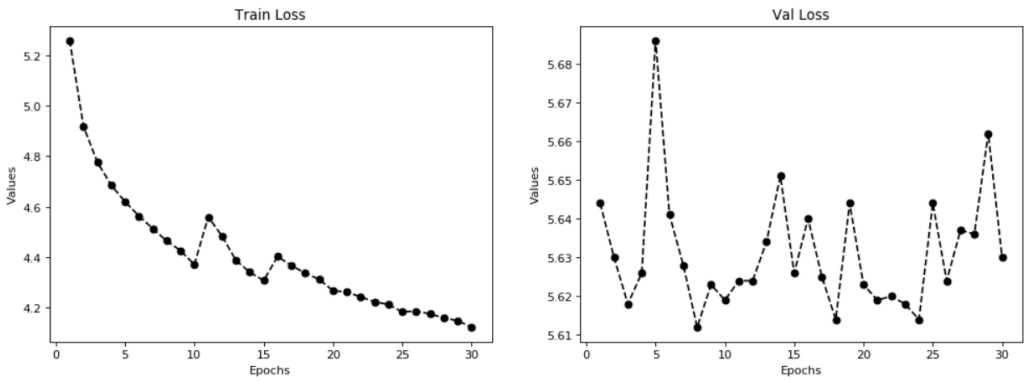

Figure 3: Training Loss

8.3.2 Fine-tuned HRED (with pre-trained weights)

The tables 7 and 8 shows the results of final 10 epochs (trained for 60 epochs) of the HRED model with pre-training.

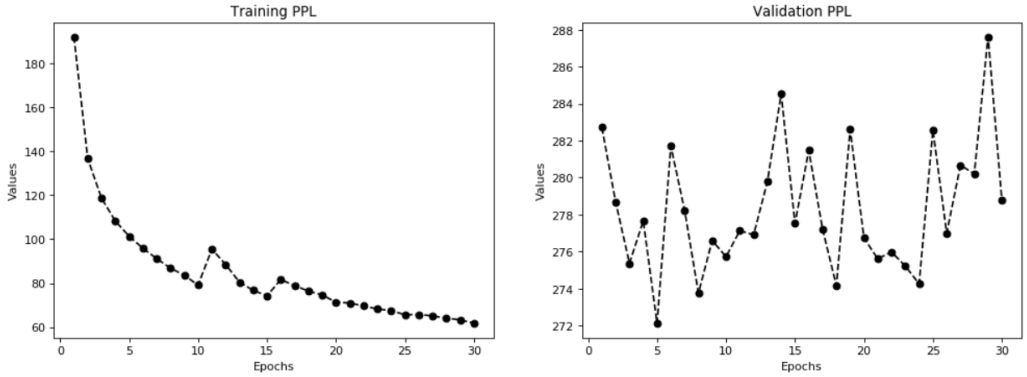

Figure 4: Perplexity Scores

EPOCH #	Training Loss	Validation Loss	Training PPL	Validation PPL
51	1.843	6.53	6.315	685.392
52	1.825	6.558	6.205	705.171
53	1.808	6.603	6.097	737.017
54	1.793	6.639	6.009	764.6
55	1.779	6.65	5.922	773.006
56	1.766	6.63	5.85	757.498
57	1.759	6.625	5.808	774.124
58	1.732	6.683	5.654	798.809
59	1.742	6.689	5.707	803.461
60	1.711	6.701	5.536	813.497

Table 7: Training and Validation Set Results for Fine-tuned HRED model with Pre-trained Weights)

Test Loss	Test PPL
6.385	593.114

Table 8: Testing Set Results for Fine-tuned HRED model with Pre-trained Weights)

The plots for losses and perplexity scores are as shown below:

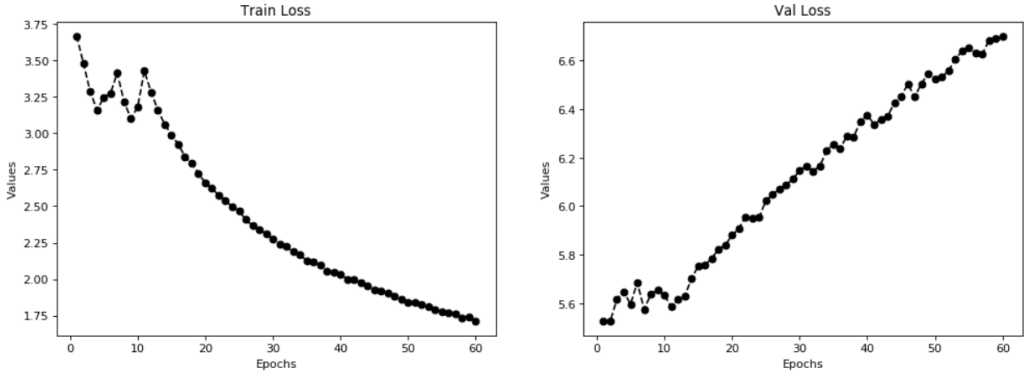

Figure 5: Training Loss

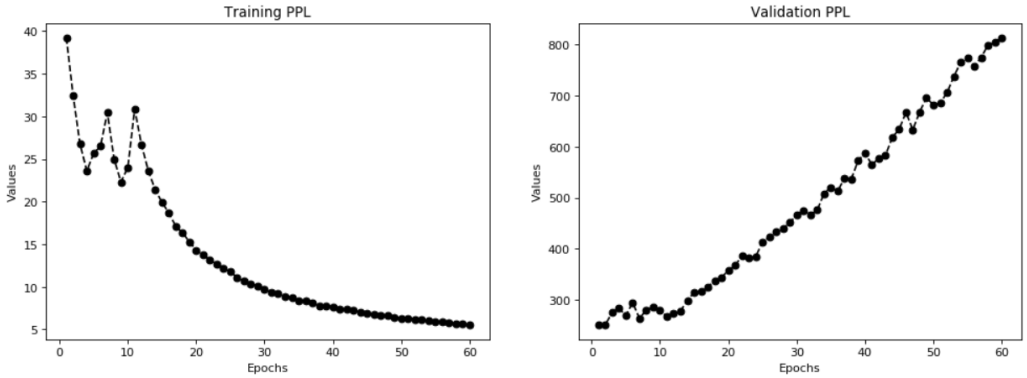

Figure 6: Perplexity Scores

8.4 Predicted Outputs by non pre-trained HRED model

Test Set

- **Context:** ['[', '"', 'we', "'ve", 'managed', 'to', 'reduce', 'our', 'energy', 'consumption', 'in', 'our', 'factory', 'by', 'about', '15', 'per', 'cent', 'in', 'the', 'last', 'two', 'years', '.', '"', ',', '"', 'that', "'s", 'excellent', '.', 'how', 'have', 'you', 'managed', 'that', '?', '"', ']']
- **Predicted Response:** ['i', 'dressed', 'i', "'m", 'be', 'pleased', 'to', 'gamble', 'the', 'regulations', 'within', 'september', 'bureau']

8.5 Predicted Outputs by pre-trained HRED model

We consider three example sentences, one from each Training, Validation and Testing set.

Training Set

- **Context:**
  " hi , becky , what 's up ? " , ' not much , except that my mother - in - law is driving me up the wall . '
- **Ground Truth Response:**
  what 's the problem ?
- **Predicted Response:**
  i 'm not an my parents and always my opportunities . she is always my my marriage is not the end of the company . <eos>

Validation Set

- **Context:**
  ' hi , good morning , miss ? what can i help you with ? ' , " good morning i 'd like to mail this box of books to taiwan . " , ' ok , please put it on this scale . airmail or by sea ? ' , ' how long does it take to reach taiwan by sea ? ' , ' usually about two month . ' , " that 's too long . how long does it take to reach taiwan by airmail ? " , ' about ten days . ' , ' then how much is that by airmail ? ' , " let me see . it ' s 57 dollars , 20 cents , including tax . "
- **Ground Truth Response:**
  that 's a little bit expensive .
- **Predicted Response:**
  yeah , please open sure that 's be possible . <eos>

Test Set

- **Context:**
  ' believe it or not , tea is the most popular beverage in the world after water . ' , ' well , people from asia to europe all enjoy tea . ' , ' right . and china is the homeland of tea . '
- **Ground Truth Response:**
  yes , chinese people love drinking tea so much . some even claim they ca n't live without tea .
- **Predicted Response:**
  the hard need more the traditional of china is more more more more more more expensive . <eos>

BLEU Score The BLEU Scores for pre-trained HRED model and non-pre-trained HRED model compared to the original Vanilla Seq2Seq model are given in table 9

Description	**Non Pre-trained HRED**	**Pre-trained HRED**
BLEU	0.03	0.22

Table 9: Comparison of BLEU Scores

8.6 Summary

The table 10 is the comparison of performance between pre-trained and non-pre-trained HRED model.

Description	**Non Pre-trained HRED**	**Pre-trained HRED**
Epochs	30	60
Training PPL	61.674	5.536
Validation PPL	278.755	813.497
Test PPL	284.113	593.114
BLEU	0.03	0.22

Table 10: Non Pre-trained vs. Pre-trained Model Summary

8.7 Conclusion

As we see from the responses illustrated above, the non pre-trained HRED model produces repetitive generic responses. However, after pre-training, this problem of natural language generation goes away, but context relevance still remains an issue.

9 Discussion

For modelling dialogues, we expected the HRED model to be superior to the standard RNN model for the reason that the context RNN allows the model to represent a form of common ground between speakers, which we hypothesize to be important for building an effective dialogue system.

While Perplexity is an established measure for generative models, in the dialogue setting, utterances may be overwhelmed by many common words especially arising from colloquial or informal exchanges. Which is why, in dialog systems, Perplexity is not always a good measure to evaluate models. What matters more is response quality.

10 Future Scope

Generating responses is observed to be considerably more difficult than translating between languages, likely due to the wide range of plausible responses and lack of phrase alignment between the post and the response, which is why the problem in hand is not easy to solve.

In HRED, the utterance representation is given by the last hidden state of the encoder RNN. This architecture may be insufficient for dialogue utterances because they are long and contain rich syntactic articulations. For long utterances, the last state of the encoder RNN may not reflect important information seen at the beginning of the utterance. Thus, one can experiment with a model where the utterance encoder is a bidirectional RNN. Bidirectional RNNs run two chains: one forward through the utterance tokens and another backward, i.e. reversing the tokens in the utterance.

Acknowledgments

We would like to thank Prof. Bivas Mitra for allowing us to pursue such an interesting topic for our course project. We would also like to thank Bishal Santra sir, our Teaching Assistant for guiding us the methodology and evaluating the project results.

References

Li, Y., Su, H., Shen, X., Li, W., Cao, Z., and Niu, S. Dailydialog: A manually labelled multi-turn dialogue dataset. *arXiv preprint arXiv:1710.03957*, 2017.

Serban, I. V., Sordoni, A., Bengio, Y., Courville, A., and Pineau, J. Building end-to-end dialogue systems using generative hierarchical neural network models. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

Sordoni, A., Bengio, Y., Vahabi, H., Lioma, C., Grue Simonsen, J., and Nie, J.-Y. A hierarchical recurrent encoder-decoder for generative context-aware query suggestion. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, pp. 553–562, 2015.

Appendix for Python Implementation Codes

10.1 Data Pre-processing for Auto-Encoder Input

10.1.1 For Auto-Encoder

Listing 1: Pre-processing the data splits into CSV file for vanilla-RNN Input

```
import pandas as pd
from tqdm import tqdm

label='test'
f=open(f"dataset/{label}.txt","r",encoding='utf-8')
data=f.read()

utterances = data.split(' __eou__ ')
SRC=utterances
TRG=utterances

data=pd.DataFrame(list(zip(SRC,TRG)),columns=['SRC','TRG'])
data.to_csv(f'dataset/{label}.csv',index=False)
```

10.1.2 For HRED Model

Listing 2: Pre-processing the data splits into CSV file for HRED Input

```
label='test'
f=open(f"dataset/{label}.txt","r",encoding='utf-8')
data=f.read()

dialogs = data.split('\n')

df=[]
SRC=[]
TRG=[]

for i in tqdm(range(len(dialogs))):
    df.append(dialogs[i].split(' __eou__ '))

for i in range(len(df)-1):

    j=1
    if (len(df[i])>2):
        for j in range(2,len(df[i])-1):
            SRC.append(df[i][:j])
            TRG.append(df[i][j])

        j+=1

        SRC.append(df[i][:j])
        TRG.append(df[i][j].split(' __eou__ ')[0])

    else:

        SRC.append(df[i][0])
        TRG.append(df[i][1].split(' __eou__ ')[0])

data=pd.DataFrame(list(zip(SRC,TRG)),columns=['SRC','TRG'])

data.to_csv(f'dataset/{label}.csv',index=False)
```

In [1]:

```
import torch
import torch.nn as nn
import torch.optim as optim

from torchtext.data import Field, BucketIterator, TabularDataset
from tqdm import tqdm

import spacy
import numpy as np

import random
import math
import time
```

In [2]:

```
if torch.cuda.is_available():
    device = "cuda"
else:
    device = "cpu"
print(f"# Using device: {device}")
```

```
# Using device: cuda
```

Then set a random seed for deterministic results/reproducability.

In [3]:

```
SEED = 1234

random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
torch.cuda.manual_seed(SEED)
torch.backends.cudnn.deterministic = True
```

Instantiate our English spaCy models.

In [4]:

```
spacy_en = spacy.load('en')
```

In [5]:

```
def tokenize_en(text):
    """
    Tokenizes English text from a string into a list of strings
    """
    return [tok.text for tok in spacy_en.tokenizer(text)]
```

Create our fields to process our data. This will append the "start of sentence" and "end of sentence" tokens as well as converting all words to lowercase.

In [6]:

```
SRC = Field(tokenize = tokenize_en,
            init_token='<sos>',
            eos_token='<eos>',
            lower=True)

TRG = Field(tokenize = tokenize_en,
            init_token='<sos>',
            eos_token='<eos>',
            lower=True)
```

Load our data.

We load two types of data - data for HRED model and data for Vanilla Seq2Seq model. The reason why we require the latter is because input and output dimensions of the model architecture are dependent on the vocabulary generated by this dataset.

In [7]:

```
train_seq2seq_data, valid_seq2seq_data, test_seq2seq_data = TabularDataset.splits(path=
                                           validation='val_data.csv',test='test_da
                                           format = 'csv', fields = [('src',SRC),(
```

In [8]:

```
train_data, valid_data, test_data = TabularDataset.splits(path='', train='train.csv',\
                                           validation='val.csv',test='test.csv', sk
                                           format = 'csv', fields = [('src',SRC),(
```

We'll also print out an example.

In [9]:

```
print(vars(train_data.examples[0]))
```

```
{'src': ['[', "'", 'say', ',', 'jim', ',', 'how', 'about', 'going', 'for',
'a', 'few', 'beers', 'after', 'dinner', '?', "'", ',', "'", 'you', 'know',
'that', 'is', 'tempting', 'but', 'is', 'really', 'not', 'good', 'for', 'ou
r', 'fitness', '.', "'", ']'], 'trg': ['what', 'do', 'you', 'mean', '?', 'i
t', 'will', 'help', 'us', 'to', 'relax', '.']}
```

Then create our vocabulary, converting all tokens appearing less than twice into `<unk>` tokens.

In [10]:

```
SRC.build_vocab(train_seq2seq_data, min_freq = 2)
TRG.build_vocab(train_seq2seq_data, min_freq = 2)
```

Finally, define the `device` and create our iterators.

In [11]:

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

In [12]:

```
BATCH_SIZE = 64

train_iterator, valid_iterator, test_iterator = BucketIterator.splits(
    (train_data, valid_data, test_data),
    batch_size = BATCH_SIZE,
    device = device, sort=False)
```

Building the Seq2Seq Model

Encoder

In [13]:

```
class Encoder(nn.Module):
    def __init__(self, input_dim, emb_dim, hid_dim, dropout):
        super().__init__()

        self.hid_dim = hid_dim

        self.embedding = nn.Embedding(input_dim, emb_dim) #no dropout as only one laye

        self.rnn = nn.GRU(emb_dim, hid_dim)

        self.dropout = nn.Dropout(dropout)

    def forward(self, src):

        #src = [src len, batch size]

        embedded = self.dropout(self.embedding(src))

        #embedded = [src len, batch size, emb dim]

        outputs, hidden = self.rnn(embedded) #no cell state!

        #outputs = [src len, batch size, hid dim * n directions]
        #hidden = [n layers * n directions, batch size, hid dim]

        #outputs are always from the top hidden layer

        return hidden
```

Context Encoder

In [14]:

```
class Context_Encoder(nn.Module):
    def __init__(self, input_dim, hid_dim, dropout):
        super().__init__()

        self.hid_dim = hid_dim

        self.rnn = nn.LSTM(input_dim, hid_dim)

        self.dropout = nn.Dropout(dropout)

    def forward(self, src):

        #src = [src len, batch size]

        outputs, (hidden, cell) = self.rnn(src) #no cell state!

        #outputs = [src len, batch size, hid dim * n directions]
        #hidden = [n layers * n directions, batch size, hid dim]

        #outputs are always from the top hidden layer

        return hidden
```

Decoder

In [15]:

```
class Decoder(nn.Module):
    def __init__(self, output_dim, emb_dim, hid_dim, dropout):
        super().__init__()

        self.hid_dim = hid_dim
        self.output_dim = output_dim

        self.embedding = nn.Embedding(output_dim, emb_dim)

        self.rnn = nn.GRU(emb_dim + hid_dim, hid_dim)

        self.fc_out = nn.Linear(emb_dim + hid_dim * 2, output_dim)

        self.dropout = nn.Dropout(dropout)

    def forward(self, input, hidden, context):

        #input = [batch size]
        #hidden = [n layers * n directions, batch size, hid dim]
        #context = [n layers * n directions, batch size, hid dim]

        #n layers and n directions in the decoder will both always be 1, therefore:
        #hidden = [1, batch size, hid dim]
        #context = [1, batch size, hid dim]

        input = input.unsqueeze(0)

        #input = [1, batch size]

        embedded = self.dropout(self.embedding(input))

        #embedded = [1, batch size, emb dim]

        emb_con = torch.cat((embedded, context), dim = 2)

        #emb_con = [1, batch size, emb dim + hid dim]

        output, hidden = self.rnn(emb_con, hidden)

        #output = [seq len, batch size, hid dim * n directions]
        #hidden = [n layers * n directions, batch size, hid dim]

        #seq len, n layers and n directions will always be 1 in the decoder, therefore
        #output = [1, batch size, hid dim]
        #hidden = [1, batch size, hid dim]

        output = torch.cat((embedded.squeeze(0), hidden.squeeze(0), context.squeeze(0)
                            dim = 1)

        #output = [batch size, emb dim + hid dim * 2]

        prediction = self.fc_out(output)

        #prediction = [batch size, output dim]

        return prediction, hidden
```

Seq2Seq Model

We need to ensure the hidden dimensions in both the encoder and the decoder are the same.

Briefly going over all of the steps:

- the `outputs` tensor is created to hold all predictions, $\hat{Y}$
- the source sequence, X, is fed into the encoder to receive an initial `context` vector
- this intial `context` vector is fed to the context encoder to receive a final `context` vector
- the initial decoder hidden state is set to be the final `context` vector, $s_0 = z = h_T$
- we use a batch of `<sos>` tokens as the first `input`, y_1
- we then decode within a loop:
  - inserting the input token y_t, previous hidden state, s_{t-1}, and the context vector, z, into the decoder
  - receiving a prediction, $\hat{y}_{t+1}$, and a new hidden state, s_t
  - we then decide if we are going to teacher force or not, setting the next input as appropriate (either the ground truth next token in the target sequence or the highest predicted next token)

In [16]:

```
class Seq2Seq(nn.Module):
    def __init__(self, encoder, con_enc, decoder, device):
        super().__init__()

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

        assert encoder.hid_dim == con_enc.hid_dim, "Hidden dimensions of encoder and c
        assert con_enc.hid_dim == decoder.hid_dim, "Hidden dimensions of context and d

    def forward(self, src, trg, teacher_forcing_ratio = 0.5):

        #src = [src len, batch size]
        #trg = [trg len, batch size]
        #teacher_forcing_ratio is probability to use teacher forcing
        #e.g. if teacher_forcing_ratio is 0.75 we use ground-truth inputs 75% of the t

        batch_size = trg.shape[1]
        trg_len = trg.shape[0]
        trg_vocab_size = self.decoder.output_dim

        #tensor to store decoder outputs
        outputs = torch.zeros(trg_len, batch_size, trg_vocab_size).to(self.device)

        #last hidden state of the encoder is the context
        input_context = self.encoder(src)

        output_context = self.con_enc(input_context)

        #context also used as the initial hidden state of the decoder
        hidden = output_context
        context = hidden

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

        for t in range(1, trg_len):

            #insert input token embedding, previous hidden state and the context state
            #receive output tensor (predictions) and new hidden state
            output, hidden = self.decoder(input, hidden, context)

            #place predictions in a tensor holding predictions for each token
            outputs[t] = output

            #decide if we are going to use teacher forcing or not
            teacher_force = random.random() < teacher_forcing_ratio

            #get the highest predicted token from our predictions
            top1 = output.argmax(1)

            #if teacher forcing, use actual next token as next input
            #if not, use predicted token
            input = trg[t] if teacher_force else top1

        return outputs
```

Training the Seq2Seq Model

The rest of this tutorial is very similar to the previous one.

We initialise our encoder, context encoder, decoder and seq2seq model (placing it on the GPU if we have one). As before, the embedding dimensions and the amount of dropout used can be different between the encoder and the decoder, but the hidden dimensions must remain the same.

In [17]:

```
INPUT_DIM = len(SRC.vocab)
HID_DIM = 512
CON_INPUT_DIM = HID_DIM
OUTPUT_DIM = len(TRG.vocab)

ENC_EMB_DIM = 256
DEC_EMB_DIM = 256

ENC_DROPOUT = 0.5
CON_DROPOUT = 0.5
DEC_DROPOUT = 0.5

enc = Encoder(INPUT_DIM, ENC_EMB_DIM, HID_DIM, ENC_DROPOUT)
con = Context_Encoder(CON_INPUT_DIM, HID_DIM, CON_DROPOUT)
dec = Decoder(OUTPUT_DIM, DEC_EMB_DIM, HID_DIM, DEC_DROPOUT)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

model = Seq2Seq(enc, con, dec, device).to(device)
```

Next, we initialize our parameters. The paper states the parameters are initialized from a normal distribution with a mean of 0 and a standard deviation of 0.01, i.e. $\mathcal{N}(0, 0.01)$.

It also states we should initialize the recurrent parameters to a special initialization, however to keep things simple we'll also initialize them to $\mathcal{N}(0, 0.01)$.

In [18]:

```
def init_weights(m):
    for name, param in m.named_parameters():
        nn.init.normal_(param.data, mean=0, std=0.01)

model.apply(init_weights)
```

Out[18]:

```
Seq2Seq(
  (encoder): Encoder(
    (embedding): Embedding(12328, 256)
    (rnn): GRU(256, 512)
    (dropout): Dropout(p=0.5, inplace=False)
  )
  (con_enc): Context_Encoder(
    (rnn): LSTM(512, 512)
    (dropout): Dropout(p=0.5, inplace=False)
  )
  (decoder): Decoder(
    (embedding): Embedding(12328, 256)
    (rnn): GRU(768, 512)
    (fc_out): Linear(in_features=1280, out_features=12328, bias=True)
    (dropout): Dropout(p=0.5, inplace=False)
  )
)
```

We print out the number of parameters.

In [19]:

```
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

print(f'The model has {count_parameters(model):,} trainable parameters')
```

```
The model has 27,357,224 trainable parameters
```

We initiaize our optimizer.

In [20]:

```
optimizer = optim.Adam(model.parameters())
```

We also initialize the loss function, making sure to ignore the loss on `<pad>` tokens.

In [21]:

```
TRG_PAD_IDX = TRG.vocab.stoi[TRG.pad_token]

criterion = nn.CrossEntropyLoss(ignore_index = TRG_PAD_IDX)
```

We then create the training loop...

In [22]:

```
def train(model, iterator, optimizer, criterion, clip):

    model.train()

    epoch_loss = 0

    for i, batch in tqdm(enumerate(iterator),total=len(iterator)):

        src = batch.src
        trg = batch.trg

        optimizer.zero_grad()

        output = model(src, trg)

        #trg = [trg len, batch size]
        #output = [trg len, batch size, output dim]

        output_dim = output.shape[-1]

        output = output[1:].view(-1, output_dim)
        trg = trg[1:].view(-1)

        #trg = [(trg len - 1) * batch size]
        #output = [(trg len - 1) * batch size, output dim]

        loss = criterion(output, trg)

        loss.backward()

        torch.nn.utils.clip_grad_norm_(model.parameters(), clip)

        optimizer.step()

        epoch_loss += loss.item()

    return epoch_loss / len(iterator)
```

...and the evaluation loop, remembering to set the model to `eval` mode and turn off teaching forcing.

In [23]:

```
def evaluate(model, iterator, criterion):

    model.eval()

    epoch_loss = 0

    with torch.no_grad():

        for i, batch in tqdm(enumerate(iterator),total=len(iterator)):

            src = batch.src
            trg = batch.trg

            output = model(src, trg, 0) #turn off teacher forcing

            #trg = [trg len, batch size]
            #output = [trg len, batch size, output dim]

            output_dim = output.shape[-1]

            output = output[1:].view(-1, output_dim)
            trg = trg[1:].view(-1)

            #trg = [(trg len - 1) * batch size]
            #output = [(trg len - 1) * batch size, output dim]

            loss = criterion(output, trg)

            epoch_loss += loss.item()

    return epoch_loss / len(iterator)
```

We'll also define the function that calculates how long an epoch takes.

In [24]:

```
def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs
```

Then, we train our model, saving the parameters that give us the best validation loss.

Loading the pretrained vanilla seq2seq model

In [25]:

```
model_dict = model.state_dict()

pretrained_dict = torch.load('seq2seq.pt')
pretrained_dict = {k: v for k, v in pretrained_dict.items() if k in model_dict}

model_dict.update(pretrained_dict)
model.load_state_dict(model_dict)
```

Out[25]:

```
<All keys matched successfully>
```

In [ ]:

```
N_EPOCHS = 60
CLIP = 1

best_valid_loss = float('inf')

for epoch in range(N_EPOCHS):

    start_time = time.time()

    train_loss = train(model, train_iterator, optimizer, criterion, CLIP)
    valid_loss = evaluate(model, valid_iterator, criterion)

    end_time = time.time()

    epoch_mins, epoch_secs = epoch_time(start_time, end_time)

    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), 'hred.pt')

    print(f'Epoch: {epoch+1:02} | Time: {epoch_mins}m {epoch_secs}s')
    print(f'\tTrain Loss: {train_loss:.3f} | Train PPL: {math.exp(train_loss):7.3f}')
    print(f'\t Val. Loss: {valid_loss:.3f} |  Val. PPL: {math.exp(valid_loss):7.3f}')
```

Finally, we test the model on the test set using these "best" parameters.

In [26]:

```
test_loss = evaluate(model, test_iterator, criterion)

print(f'| Test Loss: {test_loss:.3f} | Test PPL: {math.exp(test_loss):7.3f} |')
```

```
100%|██████████| 91/91 [00:18<00:00,  5.02it/s]

| Test Loss: 6.385 | Test PPL: 593.114 |
```

Inference

Now we can use our trained model to generate translations.

Our `translate_sentence` will do the following:

- ensure our model is in evaluation mode, which it should always be for inference
- tokenize the source sentence if it has not been tokenized (is a string)
- numericalize the source sentence
- convert it to a tensor and add a batch dimension
- get the length of the source sentence and convert to a tensor
- feed the source sentence into the encoder
- create a list to hold the output sentence, initialized with an `<sos>` token
- while we have not hit a maximum length
  - get the input tensor, which should be either `<sos>` or the last predicted token
  - feed the input, all encoder outputs, hidden state into the decoder
  - get the predicted next token
  - add prediction to current output sentence prediction
  - break if the prediction was an `<eos>` token
- convert the output sentence from indexes to tokens
- return the output sentence (with the `<sos>` token removed)

In [27]:

```
def translate_sentence(sentence, src_field, trg_field, model, device, max_len = 50):

    model.eval()

    if isinstance(sentence, str):
        nlp = spacy.load('en')
        tokens = [token.text.lower() for token in nlp(sentence)]
    else:
        tokens = [token.lower() for token in sentence]

    tokens = [src_field.init_token] + tokens + [src_field.eos_token]

    src_indexes = [src_field.vocab.stoi[token] for token in tokens]

    src_tensor = torch.LongTensor(src_indexes).unsqueeze(1).to(device)

    with torch.no_grad():
        hidden = model.encoder(src_tensor)

    context = hidden

    trg_indexes = [trg_field.vocab.stoi[trg_field.init_token]]

    for i in range(max_len):

        trg_tensor = torch.LongTensor([trg_indexes[-1]]).to(device)

        with torch.no_grad():
            output, hidden = model.decoder(trg_tensor, hidden, context)

        pred_token = output.argmax(1).item()

        trg_indexes.append(pred_token)

        if pred_token == trg_field.vocab.stoi[trg_field.eos_token]:
            break

    trg_tokens = [trg_field.vocab.itos[i] for i in trg_indexes]

    return trg_tokens[1:]
```

Now, we'll grab some translations from our dataset and see how well our model did. Note, we're going to cherry pick examples here so it gives us something interesting to look at, but feel free to change the `example_idx` value to look at different examples.

First, we'll get a source and target from our dataset.

TRAIN SET

In [28]:

```
example_idx = 15

src = vars(train_data.examples[example_idx])['src']
trg = vars(train_data.examples[example_idx])['trg']

print(f'src = {src}')
print(f'trg = {trg}')
```

```
src = ['[', "'", 'are', 'you', 'all', 'right', '?', "'", ',', "'", 'i', 'wil
l', 'be', 'all', 'right', 'soon', '.', 'i', 'was', 'terrified', 'when', 'i',
'watched', 'them', 'fall', 'from', 'the', 'wire', '.', "'", ']']
trg = ['do', "n't", 'worry', '.', 'he', 'is', 'an', 'acrobat', '。']
```

In [29]:

```
translation = translate_sentence(src, SRC, TRG, model, device)

print(f'predicted trg = {translation}')
```

```
predicted trg = ['you', 'are', 'true', '.', 'i', 'must', 'must', 'be', 'tru
e', 'here', '.', 'i', 'must', 'must', 'be', 'true', '.', '<eos>']
```

VALIDATION SET

In [30]:

```
example_idx = 14

src = vars(valid_data.examples[example_idx])['src']
trg = vars(valid_data.examples[example_idx])['trg']

print(f'src = {src}')
print(f'trg = {trg}')
```

```
src = ['[', "'", 'what', 'qualifications', 'should', 'a', 'reporter', 'hav
e', '?', "'", ',', "'", 'as', 'a', 'reporter', ',', 'he', 'must', 'have', 'a
cute', 'insight', 'and', 'language', 'skills', '.', 'at', 'the', 'same', 'ti
me', ',', 'he', 'must', 'have', 'good', 'judgment', ',', 'the', 'respect',
'for', 'his', 'job', 'and', 'tactical', 'cooperation', 'with', 'others',
'.', "'", ',', '"', 'can', 'you', 'work', 'under', 'pressure', '?', 'you',
'know', ',', 'people', 'working', 'here', 'are', 'all', 'busy', 'everyday',
'since', 'we', "'re", 'daily', 'newspaper', '.', '"', ']']
trg = ['i', 'think', 'i', "'ve", 'got', 'used', 'to', 'work', 'under', 'pres
sure', '.', 'i', 'will', 'adjust', 'myself', 'to', 'the', 'step', 'of', 'you
r', 'newspaper', 'quickly', '.']
```

In [31]:

```
translation= translate_sentence(src, SRC, TRG, model, device)

print(f'predicted trg = {translation}')
```

```
predicted trg = ['well', ',', 'he', "'s", 'probably', 'very', 'well', ',',
'a', ',', 'and', 'you', ',', 'you', 'are', "n't", ',', 'and', 'you', 'can',
"n't", 'that', ',', 'and', ',', 'you', "'re", "n't", 'that', 'in', 'raise',
',', 'and', 'you', "'re", 'quite', 'a', '<eos>']
```

TEST SET

In [32]:

```
example_idx = 25

src = vars(test_data.examples[example_idx])['src']
trg = vars(test_data.examples[example_idx])['trg']

print(f'src = {src}')
print(f'trg = {trg}')
```

```
src = ['[', "'", 'believe', 'it', 'or', 'not', ',', 'tea', 'is', 'the', 'mos
t', 'popular', 'beverage', 'in', 'the', 'world', 'after', 'water', '.', "'",
',', "'", 'well', ',', 'people', 'from', 'asia', 'to', 'europe', 'all', 'enj
oy', 'tea', '.', "'", ',', "'", 'right', '.', 'and', 'china', 'is', 'the',
'homeland', 'of', 'tea', '.', "'", ',', '"', 'yes', ',', 'chinese', 'peopl
e', 'love', 'drinking', 'tea', 'so', 'much', '.', 'some', 'even', 'claim',
'they', 'ca', "n't", 'live', 'without', 'tea', '.', '"', ',', "'", 'do', 'yo
u', 'know', 'there', 'are', 'several', 'catagories', 'of', 'chinese', 'tea',
'?', "'", ',', "'", 'yes', ',', 'i', 'believe', 'there', 'are', 'green', 'te
as', ',', 'black', 'teas', 'and', 'scented', 'teas', '.', 'any', 'others',
'?', "'", ',', "'", 'well', ',', 'have', 'you', 'ever', 'heard', 'of', 'oulo
ng', 'tea', 'and', 'compressed', 'tea', '?', "'", ',', '"', 'oh', ',', 'yea
h', '.', 'oulong', 'tea', 'is', 'good', 'for', 'one', "'s", 'health', '.',
'is', "n't", 'it', '?', '"', ',', "'", 'you', 'surely', 'know', 'a', 'lot',
'about', 'chinese', 'tea', '.', "'", ',', "'", 'sure', ',', 'i', 'like', 'dr
inking', 'tea', 'at', 'teahouses', '.', "'", ']']
trg = ['oh', ',', 'so', 'do', 'i', '.']
```

In [33]:

```
translation= translate_sentence(src, SRC, TRG, model, device)

print(f'predicted trg = {translation}')
```

```
predicted trg = ['they', 'bet', 'there', 'is', 'way', 'there', 'they', 'the
y', 'there', '5', ',', 'and', 'they', 'wo', "n't", 'keep', 'them', 'when',
'they', 'are', 'here', '-', 'hand', '-', 'they', 'they', 'are', '-', '-', '-
', 'they', 'they', 'wo', "n't", '-', 'way', '-', 'way', '-', 'old', '-', '-
', '-', 'they', 'they', 'checked', 'them', '-', 'way', '-']
```

BLEU

Previously we have only cared about the loss/perplexity of the model. However there metrics that are specifically designed for measuring the quality of a translation - the most popular is *BLEU*. Without going into too much detail, BLEU looks at the overlap in the predicted and actual target sequences in terms of their n-grams. It will give us a number between 0 and 1 for each sequence, where 1 means there is perfect overlap, i.e. a perfect translation, although is usually shown between 0 and 100. BLEU was designed for multiple candidate translations per source sequence, however in this dataset we only have one candidate per source.

We define a `calculate_bleu` function which calculates the BLEU score over a provided TorchText dataset. This function creates a corpus of the actual and predicted translation for each source sentence and then calculates the BLEU score.

Corpus Bleu

In [34]:

```
import nltk
from nltk.translate.bleu_score import sentence_bleu
from nltk.translate.bleu_score import corpus_bleu
from torchtext.data.metrics import bleu_score

def calculate_bleu(data, src_field, trg_field, model, device, max_len = 50):

    trgs = []
    pred_trgs = []

    for datum in tqdm(data):

        src = vars(datum)['src']
        trg = vars(datum)['trg']

        pred_trg = translate_sentence(src, src_field, trg_field, model, device, max_le

        #cut off <eos> token
        pred_trg = pred_trg[:-1]

        pred_trgs.append(pred_trg)
        trgs.append([trg])

    return corpus_bleu(trgs, pred_trgs)
```

In [35]:

```
bleu_score = calculate_bleu(test_data, SRC, TRG, model, device)

print(f'BLEU score = {bleu_score*100:.2f}')
```

```
100%|██████████| 5782/5782 [02:05<00:00, 46.08it/s]

BLEU score = 0.22
```

DIALOG RESPONSE OUTPUT

In [36]:

```
f = open("train_output.txt", "w",encoding='utf-8')

for example_idx in tqdm(range(len(train_data.examples))):

    src = vars(train_data.examples[example_idx])['src']
    trg = vars(train_data.examples[example_idx])['trg']

    translation = translate_sentence(src, SRC, TRG, model, device)

    print(f'-----EXAMPLE {example_idx}------', file=f)
    print('CONTEXT = ', end=' ', file=f)

    for i in src:
        print(i, end=' ', file=f)

    print(file=f)

    print(f'GROUND TRUTH RESPONSE = ', end=' ', file=f)

    for i in trg:
        print(i, end=' ', file=f)

    print(file=f)

    print(f'PREDICTED RESPONSE = ',end=' ', file=f)

    for i in translation:
        print(i, end=' ', file=f)

    print(file=f)
    print(file=f)

f.close()
```

```
100%|████████| 65418/65418 [24:28<00:00, 44.54it/s]
```

In [37]:

```
f = open("val_output.txt", "w",encoding='utf-8')

for example_idx in tqdm(range(len(valid_data.examples))):

    src = vars(valid_data.examples[example_idx])['src']
    trg = vars(valid_data.examples[example_idx])['trg']

    translation = translate_sentence(src, SRC, TRG, model, device)

    print(f'-----EXAMPLE {example_idx}------', file=f)
    print('CONTEXT = ', end=' ', file=f)

    for i in src:
        print(i, end=' ', file=f)

    print(file=f)

    print(f'GROUND TRUTH RESPONSE = ', end=' ', file=f)

    for i in trg:
        print(i, end=' ', file=f)

    print(file=f)

    print(f'PREDICTED RESPONSE = ',end=' ', file=f)

    for i in translation:
        print(i, end=' ', file=f)

    print(file=f)
    print(file=f)

f.close()
```

100%|████████| 6109/6109 [02:16<00:00, 44.87it/s]

In [38]:

```
f = open("test_output.txt", "w",encoding='utf-8')

for example_idx in tqdm(range(len(test_data.examples))):

    src = vars(test_data.examples[example_idx])['src']
    trg = vars(test_data.examples[example_idx])['trg']

    translation = translate_sentence(src, SRC, TRG, model, device)

    print(f'-----EXAMPLE {example_idx}------', file=f)
    print('CONTEXT = ', end=' ', file=f)

    for i in src:
        print(i, end=' ', file=f)

    print(file=f)

    print(f'GROUND TRUTH RESPONSE = ', end=' ', file=f)

    for i in trg:
        print(i, end=' ', file=f)

    print(file=f)

    print(f'PREDICTED RESPONSE = ',end=' ', file=f)

    for i in translation:
        print(i, end=' ', file=f)

    print(file=f)
    print(file=f)

f.close()
```

```
100%|████████| 5782/5782 [02:04<00:00, 46.29it/s]
```