Homework 3

IDS 576

Name: Isaac Salvador
Email: isalva2@uic.edu
UIN: 6669845132

Name: Ahreum Kim
Email: akim239@uic.edu
UIN: 653241895

Name: Sadjad Bazarnovi
Email: sbazar3@uic.edu
UIN: 679314994

1. RNN for Language Modeling

`torchtext` IMDB dataset

We begin by setting up the `torch` environment and downloading the `IMDB` dataset from the `torchtext.data.Dataset` class. The `torchtext` dataset is preprocessed into a into a collection of lists of tokens.

In [1]:
```
import torch
from torch import mps

import torchtext
from torchtext import datasets

from nltk.tokenize import word_tokenize
import re

# make torch deterministic for reproducibility
seed = 576
torch.manual_seed(seed)

# set device
device = torch.device("mps" if torch.backends.mps.is_available() else "cuda" if
                      torch.cuda.is_available() else "cpu")
print("Torch Device:", device)
```

```
Torch Device: mps
```

In [2]:
```
# train model on training split
train_iter = datasets.IMDB(split="train")

# initialize a list to store reviews
reviews = []

# append each review
for _, review in train_iter:
    reviews.append(review)

# initialize list to store tokenized reviews
tokenized_reviews = []

# preprocess and tokenize reviews
for review in reviews:
    lowercase_review = review.lower()
    cleaned_review = re.sub(r'[^A-Za-z0-9\s]+
                            ', '', lowercase_review.replace('<br />', ' '))
    tokenized_reviews.append(word_tokenize(cleaned_review))
```

Markov (n-gram) Language Model

A *Markov (n-gram)* language model is a purely statistical character-level model. The model is based on the assumption that the probability of the next word in a sequence is based on the history *h* of the preceding words in the sequence. For a set number *n* of immediately previous words (or tokens) in the sequence, this assumption can be expressed as:

$$P(w|h) \approx P(w_n|w_{1:n-1})$$

A *bigram* is specific *n-gram* case (*n* = 2) where the conditional probability of the next word in a sequence is dependent solely on the preceding word in the sequence

$$P(w|h) \approx P(w_n|w_{n-1})$$

We can further generalize this propoerty to *trigrams* (*n* = 3) such that the conditional probability of the next word in a sequence is

$$P(w|h) \approx P(w_n|w_{n-2,n-1})$$

Let us create a general python class `build_random_ngram_model` that replicates the above funtionality with the IMBD dataset. In order to reduce recursive, repeating predicitons, the `predict()` method introduces randomness to the model, using a weighted choice between all tokens w_n that correspond to a certain context $w_{1:n-1}$. The method parameter `rand` allows us to select the top `rand` weighted words from the model.

In [3]:
```
from nltk.util import ngrams
from tqdm import tqdm
import random

class build_random_ngram_model():
```

```
    '''
    This class builds a Markov (n-gram) language model.
    '''
    def __init__(self, tokenized_reviews: list, n: int = 2):
        self.n = n
        self.model = self._build_model(tokenized_reviews, n=n)

    def _build_model(self, tokenized_reviews, n):

        # empty return model
        model = {}

        # build model
        print("builing model...")
        for review in tqdm(tokenized_reviews):

            # build n-grams from reviews
            for ngram in ngrams(review, n, pad_right=True, pad_left=True):
                context = tuple(ngram[:-1])
                target = ngram[-1]

                # check if context is already in the model
                if context not in model:
                    model[context] = {}

                # get frequency counts of co-occurrence
                if target not in model[context]:
                    model[context][target] = 1
                else:
                    model[context][target] += 1

        # convert frequency counts to probabilities
        for context in model:
            count = float(sum(model[context].values()))
            for target in model[context]:
                model[context][target] /= count

        # return trained model
        print("done!")
        return model

    def get_model(self) -> dict:

        '''
        Returns model stored as dict.
        '''

        return self.model

    def predict(self, words: str, rand: int = None) -> str:

        '''
        This function returns the next word given the input string.
        Randomly select out of the top rand probabilities
          given a certain context.
```

```
        '''

        # get last n-1 words in string
        key = tuple(words.split())[1-self.n:]

        # get top keys in dictionary
        top_keys = sorted(self.model[key], key = self.model[key].get,
                          reverse=True)[:rand]

        # get probs of top keys
        top_values = [self.model[key][top_key] for top_key in top_keys]

        # select random key
        selected_key = random.choices(top_keys, weights = top_values, k=1)[0]

        return selected_key

    def complete(self, words: str, rand: int = None) -> str:

        '''
        This function returns the entire string with the predicted next word
        '''

        next_word = self.predict(words, rand = rand)

        # if context does not exist in model dict return original words
        if next_word is not None:
            completed_string = words+" "+next_word
        else:
            completed_string = words

        return completed_string

    def review(self, words: str, l = 10, rand: int = None) -> str:

        '''
        This function completes a review for an additional l words.
        '''

        working_words = words

        i = 0

        while i < l:
            working_words = self.complete(working_words, rand = rand)
            i += 1

        return working_words
```

We can now build a trigram language model and generate random reviews from the input string `"My favorite movie"`.

In [4]:
```
trigram_model = build_random_ngram_model(tokenized_reviews, 3)

for i in range(5):
```

```
    generated_text = trigram_model.review("My favorite movie", 17)
    print(f"Review {i+1}: {generated_text}")
```

```
buiing model...
100%|██████████| 25000/25000 [00:03<00:00, 6612.59it/s]
done!
Review 1: My favorite movie of course never sang for the dramatic confessions boiled
down to earth to bring our modern conveniences
Review 2: My favorite movie role albeit one with a title that is set around a dentis
t then this movie again
Review 3: My favorite movie but it was awful for words the actors the combined talen
t of this adult humor if you
Review 4: My favorite movie from 1968 this movies actors were not generally that imp
ressive for american audiences like dogs they serve
Review 5: My favorite movie they have to settlefor 1010
```

LSTM Based Language Model

For the LSTM based language model we shall be following along with Seq2Seq_LSTM_Simple_Sentiment_Analysis.ipynb and making modifications as neccessary.

First we download the `"train"` and `"test"` splits from the `IMDB` dataset and check the size of each split. For the sake of computational complexity, the training dataset is reduced to 10,000 instance and the validation dataset is reduced to 10% at 1,000.

In [35]:
```
from torch.utils.data import Subset

train_n = 10000
valid_n = 100

train_dataset_raw = datasets.IMDB(split="train")
valid_dataset_raw = datasets.IMDB(split="test")

train_dataset = Subset(list(train_dataset_raw), range(train_n))
valid_dataset = Subset(list(valid_dataset_raw), range(valid_n))

print("Train dataset size: ",len(list(train_dataset)))
print("Test dataset size: ",len(list(valid_dataset)))
```

```
Train dataset size:  10000
Test dataset size:  100
```

Data Preprocessing Pipeline

We next construct utilities to aid in the preprocessing of the `IMDB` dataset. Since we are creating a language model, our preprocessing pipeline must result in the generation of an *input sequence* and *target sequence*. The *input sequence* and *target sequence* differ only by one "time-step".

$$X_{Raw} = [x_1, x_2, \ldots, x_T]$$

$$X_{Input} = [x_1, x_2, \ldots, x_{T-1}]$$

$$X_{Target} = [x_2, x_3, \ldots, x_T]$$

The first utility is the `tokenizer()`, that parses through string instances in the datasets and converts to tokens.

In [36]:
```
def tokenizer(text):
    '''
    # step 1. remove HTML tags. they are not helpful
     in understanding the sentiments of a review
    # step 2: use lowercase for all text to keep symmetry
    # step 3: extract emoticons. keep them as they
     are important sentiment signals
    # step 4: remove punctuation marks
    # step 5: put back emoticons
    # step 6: generate word tokens
    '''
    text = re.sub("<[^>]*>", "", text)
    text = text.lower()
    emoticons = re.findall("(?::|;|=)(?:-)?(?:\)|\(|D|P)", text)
    text = re.sub("[\W]+", " ", text)
    text = text + " ".join(emoticons).replace("-", "")
    tokenized = text.split()
    return tokenized
```

We next create a utility to obtain `token_counts` from the dataset.

In [37]:
```
from collections import Counter

token_counts = Counter()

for _, line in iter(train_dataset):
    tokens = tokenizer(line)
    token_counts.update(tokens)

print('IMDB train dataset vocab size:', len(token_counts))
```

```
IMDB train dataset vocab size: 49722
```

The tokens are then sorted by frequency and converted to to integers using the `vocab` object. For the sake of computational complexity the first 1,000 words of the ordered dictionary by freqeuency are used for the model vocabulary.

In [38]:
```
from collections import OrderedDict
from torchtext.vocab import vocab

sorted_by_freq_tuples = sorted(token_counts.items(), key=lambda x: x[1],
                               reverse=True)
ordered_dict = OrderedDict(sorted_by_freq_tuples)

reduced_ordered_dict = OrderedDict()

count = 0
for key, value in ordered_dict.items():
```

```
    reduced_ordered_dict[key] = value
    count += 1

    if count == 1000:
        break

vb = vocab(reduced_ordered_dict)

vb.insert_token("<pad>", 0)  # special token for padding
vb.insert_token("<unk>", 1)  # special token for unknown words
vb.set_default_index(1)

# print some token indexes from vocab
for token in ["this", "is", "an", "example"]:
    print(token, " --> ", vb[token])
```

```
this  -->  11
is  -->  7
an  -->  40
example  -->  438
```

In [39]:
```
input_pipeline = lambda x: [vb[token] for token in tokenizer(x)][:-1]
target_pipeline = lambda x: [vb[token] for token in tokenizer(x)][1:]

length = 5
print("Example Sequences")
for i in list(train_dataset_raw)[:4]:
    example_tokens = tokenizer(i[1])[:length+1]
    example_input = input_pipeline(i[1])[:length]
    example_target = target_pipeline(i[1])[:length]

    print(f'Sample Tokens: {example_tokens}')
    print(f'Input Sequence: {example_input}')
    print(f'Target Sequence: {example_target}', "\n")
```

```
Example Sequences
Sample Tokens: ['i', 'rented', 'i', 'am', 'curious', 'yellow']
Input Sequence: [9, 1, 9, 223, 1]
Target Sequence: [1, 9, 223, 1, 1]

Sample Tokens: ['i', 'am', 'curious', 'yellow', 'is', 'a']
Input Sequence: [9, 223, 1, 1, 7]
Target Sequence: [223, 1, 1, 7, 3]

Sample Tokens: ['if', 'only', 'to', 'avoid', 'making', 'this']
Input Sequence: [43, 58, 6, 483, 205]
Target Sequence: [58, 6, 483, 205, 11]

Sample Tokens: ['this', 'film', 'was', 'probably', 'inspired', 'by']
Input Sequence: [11, 21, 14, 244, 1]
Target Sequence: [21, 14, 244, 1, 38]
```

The preprocessing utilities will be applied at the batch level, as a `collate_fn` argument. The source code is modified to return the input and target sequences. Since these sequences are of the same length, we do not need to return length

In [40]:
```
import torch.nn as nn

def collate_batch(batch):
    input_list, target_list, = [], []

    # iterate over all reviews in a batch
    for _, _text in batch:

        # input preprocessing
        processed_input = torch.tensor(input_pipeline(_text),
                                       dtype=torch.int64)

        # target preprocessing
        processed_target = torch.tensor(target_pipeline(_text),
                                        dtype=torch.int64)


        # store the processed text in input and target lists
        input_list.append(processed_input)
        target_list.append(processed_target)

    # pad the processed reviews to make their lengths consistant
    padded_input_list = nn.utils.rnn.pad_sequence(
        input_list, batch_first=True)

    padded_target_list = nn.utils.rnn.pad_sequence(
        target_list, batch_first=True
    )

    # return
    # 1. a list of processed and padded input texts
    # 2. a list of processed and padded target texts
    # 3. a list of review text original lengths (before padding)
    return padded_input_list.to(device), padded_target_list.to(device)
```

Batching the Training, Validation, and Test Datasets

The `IMDB` Datasets are loaded into torch `DataLoader()` objects with the above `collate_batch()` function.

In [41]:
```
from torch.utils.data import DataLoader
batch_size = 32

train_dl = DataLoader(
    train_dataset, batch_size=batch_size, shuffle=True,
    collate_fn=collate_batch
)
valid_dl = DataLoader(
    valid_dataset, batch_size=batch_size, shuffle=False,
```

```
    collate_fn=collate_batch
)
```

We will now modify the source code's sentiment analysis `RNN` class for the purposes of language modeling. An overview of the architecture and design choices made to implement this are as follows:

- An **Embedding layer** as as explained by the source code.
- An **LSTM layer** to capture long range dependencies and relationships in the text.
- A **Fully Connected layer** to obtain `logits`, the raw unnormalized predictions for each token at each "time step" in the model.

In [44]:
```
# create langauge model class
class RNN_Language(nn.Module):
    def __init__(self, vocab_size, embed_dim):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim, padding_idx=0)
        self.rnn = nn.LSTM(embed_dim, vocab_size)#, batch_first=True)

    def forward(self, input_sequence):
        out = self.embedding(input_sequence)
        out, _ = self.rnn(out)
        out.view(-1, vocab_size)
        return out

# instantiate a model
vocab_size = len(vb)
embed_dim = 20

torch.manual_seed(576)
model = RNN_Language(vocab_size, embed_dim)
model = model.to(device)

total_params = sum(p.numel() for p in model.parameters())
print(f"Number of parameters: {total_params}")
```

```
Number of parameters: 4124232
```

Loss Function, Optimizer and Training Loop

We will utilize `CrossEntropyLoss()` as our loss function and the `Adam()` algorithm for optimization.

In [45]:
```
from tqdm import tqdm

criterion = nn.CrossEntropyLoss()
criterion.to(device)

optimizer = torch.optim.Adam(model.parameters(), lr = 1e-3)

model.to(device)
```

```
num_epochs = 10

# training loop

def train(model, train_dl, valid_dl):

    train_history, valid_history = [], []

    for epoch in range(num_epochs):

        print(f"Epopch: {epoch}")

        model.train()
        total_loss, valid_loss = 0, 0

        # training
        for input_batch, target_batch in tqdm(train_dl):
            with torch.set_grad_enabled(True):
                # forward pass
                optimizer.zero_grad()
                logits = model(input_batch)
                loss = criterion(logits.view(-1, vocab_size),
                                 target_batch.view(-1))

                # backwards pass
                loss.backward()
                optimizer.step()

            total_loss += loss.item() * batch_size

        avg_loss = total_loss / len(train_dl.dataset)
        train_history.append(avg_loss)
        print(f"Training loss:\t{avg_loss}")

        # validation
        for input_batch, target_batch in valid_dl:
            with torch.no_grad():
                logits = model(input_batch)
                loss = criterion(logits.view(-1, vocab_size),
                                 target_batch.view(-1))

            valid_loss += loss.item() * batch_size

        avg_valid_loss = valid_loss / len(valid_dl.dataset)
        valid_history.append(avg_valid_loss)
        print(f"Validation Loss\t{avg_valid_loss}\n")


    return train_history, valid_history
```

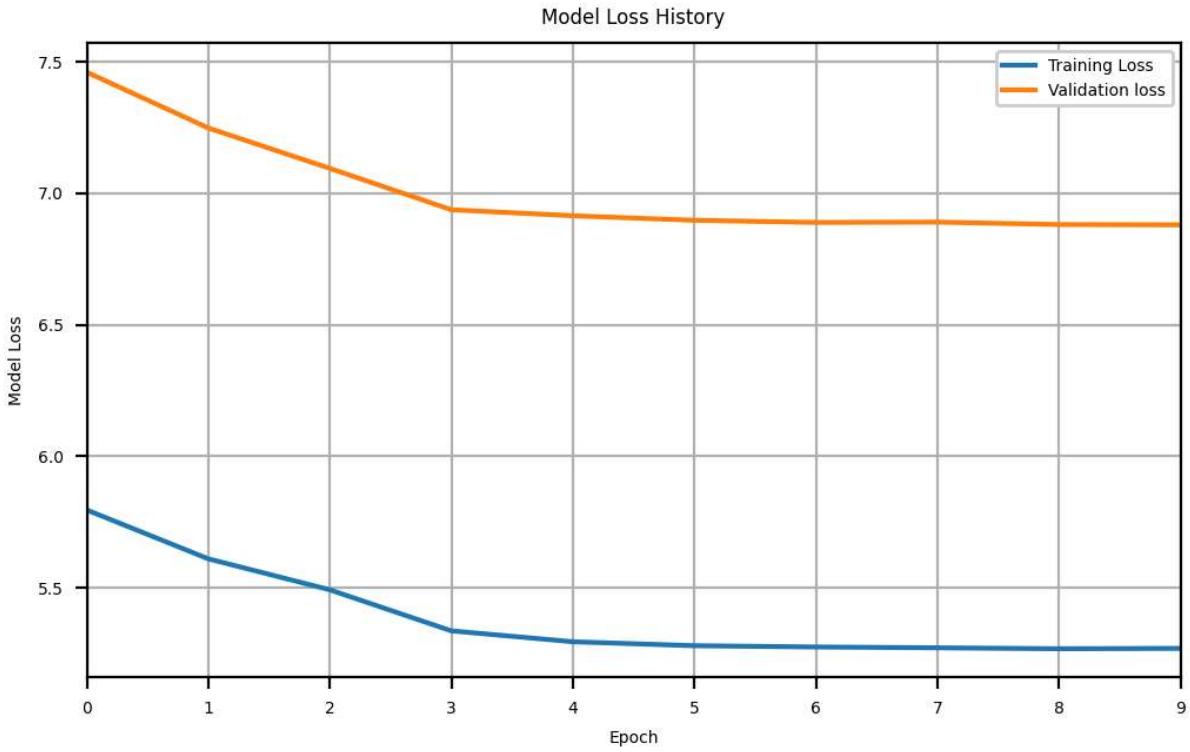
In [46]:
```
train_history, valid_history = train(model, train_dl, valid_dl)
```

```
Epopch: 0
100%|██████████| 313/313 [01:42<00:00,  3.06it/s]
```

```
Training loss:  5.796075540161133
Validation Loss 7.459661254882812

Epopch: 1
100%|██████████| 313/313 [01:41<00:00,  3.08it/s]
Training loss:  5.610481993103027
Validation Loss 7.247596893310547

Epopch: 2
100%|██████████| 313/313 [01:39<00:00,  3.16it/s]
Training loss:  5.49277162475586
Validation Loss 7.094161376953125

Epopch: 3
100%|██████████| 313/313 [01:39<00:00,  3.15it/s]
Training loss:  5.336782221984863
Validation Loss 6.936937255859375

Epopch: 4
100%|██████████| 313/313 [01:41<00:00,  3.08it/s]
Training loss:  5.295392292785644
Validation Loss 6.914114837646484

Epopch: 5
100%|██████████| 313/313 [01:41<00:00,  3.07it/s]
Training loss:  5.280378269958496
Validation Loss 6.8968586730957036

Epopch: 6
100%|██████████| 313/313 [01:48<00:00,  2.89it/s]
Training loss:  5.275519085693359
Validation Loss 6.888632202148438

Epopch: 7
100%|██████████| 313/313 [01:42<00:00,  3.06it/s]
Training loss:  5.272331005859375
Validation Loss 6.889962615966797

Epopch: 8
100%|██████████| 313/313 [01:43<00:00,  3.04it/s]
Training loss:  5.268844647216797
Validation Loss 6.880750579833984

Epopch: 9
100%|██████████| 313/313 [01:35<00:00,  3.27it/s]
Training loss:  5.270249406433106
Validation Loss 6.8794253540039065
```

In [48]:
```
import matplotlib.pyplot as plt
plt.rcParams.update({'font.size': 5})
```

```
# plot loss history
fig, ax = plt.subplots(figsize=(6,3.5), dpi=200)

ax.plot(
        train_history,
        label = r'Training Loss',
        #marker = 'o',
        markersize = 4)

ax.plot(
        valid_history,
        label = r'Validation loss',
        #marker = 'o',
        markersize = 4)

ax.set_title("Model Loss History")
ax.set_xlabel('Epoch')
ax.set_ylabel('Model Loss')
ax.set_xlim(0, len(train_history)-1)
ax.legend()
ax.legend().get_frame().set_alpha(1.0)

ax.grid(True)
```



Training and Evaluation Pipelines

In [51]:
```
import torch.nn.functional as F

def generate_text(model, text, max = 17, adjust = 1):
    # set model to evaluation
    model.eval()
```

```
    # copy input string
    input = text

    # iterate max times
    for _ in range(max):

        # convert to tensor tokens
        text_tokens = [vb[token] for token in tokenizer(input)]
        tensor_tokens = torch.tensor(text_tokens, dtype = torch.int64).to(device)

        # evaluate on model and adjust by scalar to add addtional randomness
        with torch.no_grad():
            logits = model(tensor_tokens)[-1,:] / adjust

        # convert to probabilies
        probs = F.softmax(logits, dim=-1)

        # use multinomial distribution to select index of next word based on probs
        choice = torch.multinomial(probs, 1).item()

        # obtain next word from vb
        next_word = vb.lookup_token(choice)

        # append to end of input string
        input = input+" "+next_word

    return input

for i in range(5):
    generated_text = generate_text(model, "My favorite movie", 17, 10)
    print(f"Review {i+1}: {generated_text}")
```

```
Review 1: My favorite movie another after fairly villain known others definitely wer
en nice <unk> then other behind away straight men elements
Review 2: My favorite movie save now directors them read sci obvious without why hou
r today only title with video space write
Review 3: My favorite movie living cartoon war top should potential actress problem
working background has short storyline happy call free group
Review 4: My favorite movie they known hardly bunch lead set monster paul ben either
does dog many called despite ve history
Review 5: My favorite movie has style in dramatic any probably average second street
wait <unk> scary find good each across kept
```

Sequence to Sequence Model for Translation

Korean to English Model

- Train the sequence to sequence model (Model 1) for a language pair (excluding French-English), where the output is English and the input is a language of your choice.

Loading data

In [ ]:
```
# Create dataset 'kor-eng.txt'
import pandas as pd

# The data come from: https://www.manythings.org/anki/
file_path = 'kor.txt'
df = pd.read_csv(file_path, sep='\t',
                 header=None)
df = df.iloc[:,:2]
output_file_path = 'kor-eng.txt'
df.to_csv(output_file_path, sep='\t', index=False, header=False)
```

In [ ]:
```
from __future__ import unicode_literals, print_function, division
from io import open
import unicodedata
import string
import re
import random

import torch
import torch.nn as nn
from torch import optim
import torch.nn.functional as F

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

The individual words in a language are represented as one-hot vectors. These vectors are essentially giant arrays filled with zeros except for a single '1' at the position corresponding to the specific word.

For subsequent use in our neural networks as both inputs and targets, each word must have a unique index. To manage this indexing process, we define utility class known as `Lang`.

This `Lang` class contains two dictionaries:

- one mapping words to their respective indices (`word2index`), and
- the other mapping indices back to their corresponding words (`index2word`).

In [ ]:
```
SOS_token = 0 # stands for "Start of Sentence"
EOS_token = 1 # stands for "End of Sentence"


class Lang:
    def __init__(self, name):
        self.name = name
        #  Three dictionaries to store information about words
        #  in the language vocabulary
        self.word2index = {} # SOS and EOS tokens are initially added
        self.word2count = {} # keeps track of how many times each word has appeared
        self.index2word = {0: "SOS", 1: "EOS"} # A reverse mapping from indices to
```

```
        # initialized to 2 to account for the "SOS" and "EOS" tokens
        self.n_words = 2  # Count SOS and EOS

    def addSentence(self, sentence):
        # adding all the words in the sentence to the vocabulary
        for word in sentence.split(' '):
            self.addWord(word)

    def addWord(self, word):
        # takes a word (a string) as input and adds it to the vocabulary
        # if it's not already present
        if word not in self.word2index:
            self.word2index[word] = self.n_words
            self.word2count[word] = 1
            self.index2word[self.n_words] = word
            self.n_words += 1
        else:
            self.word2count[word] += 1



# Turn a Unicode string to plain ASCII
def unicodeToAscii(s):
    return ''.join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn'
    )


# Lowercase, trim, and remove non-letter characters
def normalizeString(s):
    # s = unicodeToAscii(s.lower().strip())
    s = re.sub(r"([.!?])", r" \1", s)
    # s = re.sub(r"[^a-zA-Z.!?]+", r" ", s)
    return s
```

- Read text file and split into lines, split lines into pairs
- Normalize text, filter by length and content
- Make word lists from sentences in pairs

In [ ]:

```
# Read and preprocess data from a text file containing language pairs
def readLangs(lang1, lang2, reverse=False):
    print("Reading lines...")

    # Read the file and split into lines
    lines = (open('%s-%s.txt' % (lang1, lang2), encoding='utf-8').read().
            strip(). # removes any leading or trailing whitespace
            split('\n')) # splits the content of the file into a list of lines

    # Split every line into pairs and normalize
    # It further processes the lines read from the file
    pairs = [[normalizeString(s) for s in l.split('\t')] for l in lines]
    # The result is a list of pairs, where
    # each pair contains two normalized strings
```

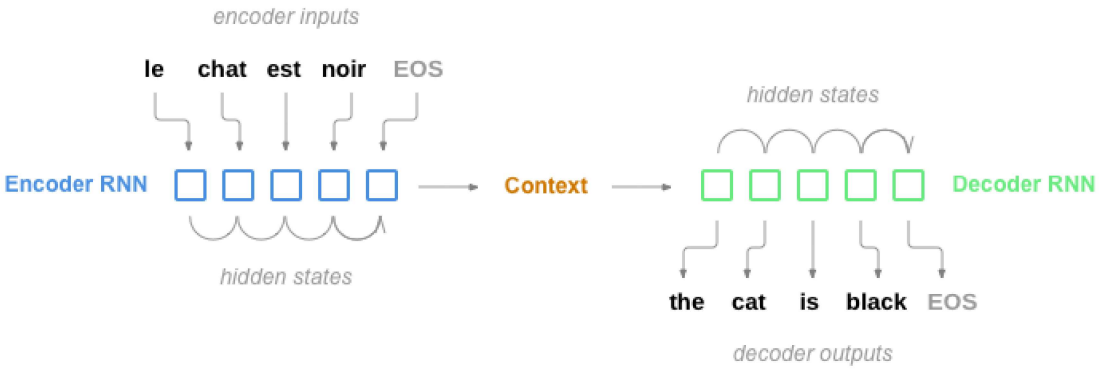
```
    # Reverse pairs, make Lang instances
    if reverse:
        # reverses the order of elements in each pair
        pairs = [list(reversed(p)) for p in pairs]
        input_lang = Lang(lang2) # becomes the input language
        output_lang = Lang(lang1) # becomes the output language
    else:
        input_lang = Lang(lang1)
        output_lang = Lang(lang2)

    return input_lang, output_lang, pairs


# Any pair of sentences where either the source sentence or the target sentence
# exceeds a length of 10 words will be filtered out.
MAX_LENGTH = 10

# eng_prefixes = (
#     "i am ", "i m ",
#     "he is", "he s ",
#     "she is", "she s ",
#     "you are", "you re ",
#     "we are", "we re ",
#     "they are", "they re "
# )

# If all these conditions are met,
# the function returns True, indicating that the pair should be kept.
def filterPair(p):
    # checks the length of the source sentence
    # checks the length of the target sentence
    return (len(p[0].split(' ')) < MAX_LENGTH and
            len(p[1].split(' ')) < MAX_LENGTH
                # and p[1].startswith(eng_prefixes)
                )

# It returns a filtered list of pairs
# where each pair satisfies the conditions
def filterPairs(pairs):
    return [pair for pair in pairs if filterPair(pair)]

# Read text file and split into lines, split lines into pairs
# Normalize text, filter by length and content
# Make word lists from sentences in pairs

def prepareData(lang1, lang2, reverse=False):

    input_lang, output_lang, pairs = readLangs(lang1, lang2, reverse)
    print("Read %s sentence pairs" % len(pairs))

    pairs = filterPairs(pairs)
    print("Trimmed to %s sentence pairs" % len(pairs))
    print("Counting words...")
    for pair in pairs:
        input_lang.addSentence(pair[0])
```

```
        output_lang.addSentence(pair[1])
    print("Counted words:")
    print(input_lang.name, input_lang.n_words)
    print(output_lang.name, output_lang.n_words)

    return input_lang, output_lang, pairs
```

The Seq2Seq Model

A seq2seq network, also known as an Encoder-Decoder network, comprises two distinct RNNs: an encoder and a decoder.

- The encoder processes input sequences, generating a vector at each step, and the final encoder output becomes the context vector.
- The decoder utilizes this context vector to generate an output sequence step by step.



Reference: https://github.com/spro/practical-pytorch/blob/master/seq2seq-translation/seq2seq-translation.ipynb

Define the encoder and decoder

In [ ]:
```
# Encoder

# defines a Python class named EncoderRNN,
# which is a subclass of nn.Module
class EncoderRNN(nn.Module):
    def __init__(self, input_size, hidden_size): # It takes two arguments
        # input_size: The size of the input vocabulary,
        # which represents the number of unique tokens in the input data.
        # hidden_size: The size of the hidden state of the GRU
        # (Gated Recurrent Unit) layer in the encoder.

        # creates an embedding layer (nn.Embedding) within the encoder
        super(EncoderRNN, self).__init__()
        self.hidden_size = hidden_size
        self.embedding = nn.Embedding(input_size, hidden_size)
        # The GRU is a type of recurrent neural network (RNN) layer
        # used to process sequential data
        self.gru = nn.GRU(hidden_size, hidden_size)
```

```
    def forward(self, input, hidden):
        # input: a tensor representing a sequence of tokens
        # shape is expected to be (sequence_length, batch_size)
        # hidden: The initial hidden state of the GRU layer

        # It reshapes the embeddings to have a shape of (1, 1, -1),
        # where -1 means that the size is inferred based on the other dimensions.
        embedded = self.embedding(input).view(1, 1, -1)
        output = embedded
        output, hidden = self.gru(output, hidden)
        return output, hidden

    def initHidden(self):
        # initializes the hidden state of the encoder
        # It returns a tensor filled with zeros of shape (1, 1, hidden_size)
        return torch.zeros(1, 1, self.hidden_size, device=device)
```

In [ ]:
```
# Decoder

class DecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size):
        # hidden_size: The size of the hidden state of the GRU
        # output_size: The size of the output vocabulary

        super(DecoderRNN, self).__init__()
        # initialize the DecoderRNN class properly
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(output_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size)
        self.out = nn.Linear(hidden_size, output_size)

        # a log-softmax layer (nn.LogSoftmax)
        # along the specified dimension (dim=1).
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        # defines the forward pass of the decoder

        output = self.embedding(input).view(1, 1, -1)

        # (ReLU) activation function to the embeddings
        output = F.relu(output)
        output, hidden = self.gru(output, hidden)
        output = self.softmax(self.out(output[0]))
        return output, hidden

    def initHidden(self):
        # initializes the hidden state of the decoder
        return torch.zeros(1, 1, self.hidden_size, device=device)
```

In [ ]:
```
# Attention Decoder

class AttnDecoderRNN(nn.Module):
```

```
    def __init__(self, hidden_size, output_size, dropout_p=0.1,
                 max_length=MAX_LENGTH):
        # The dropout probability used to prevent overfitting.

        super(AttnDecoderRNN, self).__init__()
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.dropout_p = dropout_p
        self.max_length = max_length

        self.embedding = nn.Embedding(self.output_size, self.hidden_size)

        # defines a linear layer (nn.Linear) that computes attention scores.
        self.attn = nn.Linear(self.hidden_size * 2, self.max_length)

        # defines another linear layer that combines
        # the attention context vector with the decoder's input embedding
        self.attn_combine = nn.Linear(self.hidden_size * 2, self.hidden_size)

        # defines a dropout layer (nn.Dropout) to apply dropout
        # with the specified probability self.dropout_p to the embeddings.
        self.dropout = nn.Dropout(self.dropout_p)

        #The GRU is a type of recurrent neural network (RNN)
        # layer used to process sequential data.
        self.gru = nn.GRU(self.hidden_size, self.hidden_size)

        # defines a linear layer (nn.Linear) that maps the output of
        # the decoder to the size of the output vocabulary
        #produce the final prediction for the next token in the target sequence
        self.out = nn.Linear(self.hidden_size, self.output_size)


    # forward pass of the decoder
    # The input data, which is typically a token from the target sequence.
    # Its shape is expected to be (sequence_length, batch_size)
    def forward(self, input, hidden, encoder_outputs):
        embedded = self.embedding(input).view(1, 1, -1)
        embedded = self.dropout(embedded)

        attn_weights = F.softmax(
            self.attn(torch.cat((embedded[0], hidden[0]), 1)), dim=1)
        attn_applied = torch.bmm(attn_weights.unsqueeze(0),
                                 encoder_outputs.unsqueeze(0))

        output = torch.cat((embedded[0], attn_applied[0]), 1)
        output = self.attn_combine(output).unsqueeze(0)

        output = F.relu(output)
        output, hidden = self.gru(output, hidden)

        output = F.log_softmax(self.out(output[0]), dim=1)
        return output, hidden, attn_weights

    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size, device=device)
```

Training

The whole training process looks like this:

- Start a timer
- Initialize optimizers and criterion
- Create set of training pairs
- Start empty losses array for plotting

In [ ]:
```
# Preparing the training data to put into a nn model
# Three functions that are used for preprocessing and
# converting text data into tensors

def indexesFromSentence(lang, sentence):
    # it looks up its index in the lang object
    # The list of indices corresponding to the words in the
    # sentence is returned as the output
    # return [lang.word2index[word] for word in sentence.split(' ')]
    # revised the code as below
    return [lang.word2index[word] for word in sentence.split(' ') if word.strip() !



def tensorFromSentence(lang, sentence):
    indexes = indexesFromSentence(lang, sentence)
    indexes.append(EOS_token)
    return torch.tensor(indexes, dtype=torch.long, device=device).view(-1, 1)

def tensorsFromPair(pair, input_lang, output_lang):
    input_tensor = tensorFromSentence(input_lang, pair[0])
    target_tensor = tensorFromSentence(output_lang, pair[1])
    return (input_tensor, target_tensor)
```

In [ ]:
```
# Training the model

teacher_forcing_ratio = 0.5
# Teacher forcing is a technique where, during training,
# the target sequence is used as input to the decoder instead
# of the model's own predictions.
# A value of 0.5 means that there's a 50% chance of using teacher forcing.

def train(input_tensor, target_tensor, encoder, decoder, encoder_optimizer,
          decoder_optimizer, criterion, max_length=MAX_LENGTH):
    # criterion: The loss function (e.g., cross-entropy loss).

    # initialization
    encoder_hidden = encoder.initHidden()
    # clears the gradients of the encoder's parameters
    encoder_optimizer.zero_grad()
    decoder_optimizer.zero_grad()

    input_length = input_tensor.size(0)
```

```
    target_length = target_tensor.size(0)

    # initializes a tensor encoder_outputs
    # to store the outputs of the encoder for each time step
    encoder_outputs = torch.zeros(max_length,
                                  encoder.hidden_size, device=device)

    loss = 0

    # runs a loop over the input sequence to encode it using the encoder
    for ei in range(input_length):
        # For each time step,
        # it calls the encoder with the input token and the current hidden state
        encoder_output, encoder_hidden = encoder(
            input_tensor[ei], encoder_hidden)
        encoder_outputs[ei] = encoder_output[0, 0]

    # initializes the input to the decoder with the
    #  "Start of Sentence" token (SOS_token)
    decoder_input = torch.tensor([[SOS_token]], device=device)

    # initializes the decoder's hidden state with the
    # final hidden state of the encoder
    decoder_hidden = encoder_hidden

    use_teacher_forcing = True if random.random() < teacher_forcing_ratio else Fals

    if use_teacher_forcing:
        # Teacher forcing: Feed the target as the next input
        for di in range(target_length):
            decoder_output, decoder_hidden, decoder_attention = decoder(
                decoder_input, decoder_hidden, encoder_outputs)

            # calculates the loss between the decoder output and the target tensor
            # at the current time step and adds it to the cumulative loss
            loss += criterion(decoder_output, target_tensor[di])

            # The decoder input for the next time step is set to
            # the corresponding token from the target sequence (teacher forcing)
            decoder_input = target_tensor[di]  # Teacher forcing

    else:
        # Without teacher forcing: use its own predictions as the next input
        for di in range(target_length):
            decoder_output, decoder_hidden, decoder_attention = decoder(
                decoder_input, decoder_hidden, encoder_outputs)

            # The decoder input for the next time step is set to
            # the token predicted by the decoder at the current time step.
            topv, topi = decoder_output.topk(1)
            decoder_input = topi.squeeze().detach()  # detach from history as input

            loss += criterion(decoder_output, target_tensor[di])

            if decoder_input.item() == EOS_token:
                break
```

```
    loss.backward()

    # updates the encoder's parameters using the optimizer
    # (e.g., stochastic gradient descent or Adam)
    encoder_optimizer.step()
    decoder_optimizer.step()

    return loss.item() / target_length
```

We call `train` many times and occasionally print the progress (% of examples, time so far, estimated time) and average loss.

In [ ]:
```
# This is a helper function to print time elapsed and
# estimated time remaining given the current time and progress %.
import time
import math

def asMinutes(s):
    m = math.floor(s / 60)
    s -= m * 60
    return '%dm %ds' % (m, s)

def timeSince(since, percent):
    now = time.time()
    s = now - since
    es = s / (percent)
    rs = es - s
    return '%s (- %s)' % (asMinutes(s), asMinutes(rs))


def trainIters(input_lang, output_lang, encoder, decoder, n_iters,
               print_every=1000, plot_every=100, learning_rate=0.01):
    start = time.time()
    plot_losses = []
    print_loss_total = 0  # Reset every print_every
    plot_loss_total = 0  # Reset every plot_every

    encoder_optimizer = optim.SGD(encoder.parameters(), lr=learning_rate)
    decoder_optimizer = optim.SGD(decoder.parameters(), lr=learning_rate)
    training_pairs = [tensorsFromPair(random.choice(pairs),
                                      input_lang, output_lang)
                      for i in range(n_iters)]
    criterion = nn.NLLLoss()

    for iter in range(1, n_iters + 1):
        training_pair = training_pairs[iter - 1]
        input_tensor = training_pair[0]
        target_tensor = training_pair[1]

        loss = train(input_tensor, target_tensor, encoder,
                     decoder, encoder_optimizer, decoder_optimizer, criterion)
        print_loss_total += loss
        plot_loss_total += loss
```

```
        if iter % print_every == 0:
            print_loss_avg = print_loss_total / print_every
            print_loss_total = 0
            print('%s (%d %d%%) %.4f' % (timeSince(start, iter / n_iters),
                                         iter, iter / n_iters * 100,
                                         print_loss_avg))

        if iter % plot_every == 0:
            plot_loss_avg = plot_loss_total / plot_every
            plot_losses.append(plot_loss_avg)
            plot_loss_total = 0

    showPlot(plot_losses)



# Polotting the result
import matplotlib.pyplot as plt
plt.switch_backend('agg')
import matplotlib.ticker as ticker
import numpy as np


def showPlot(points):
    plt.figure()
    fig, ax = plt.subplots()
    # this locator puts ticks at regular intervals
    loc = ticker.MultipleLocator(base=0.2)
    ax.yaxis.set_major_locator(loc)
    plt.plot(points)
```

Evaluation

There are no targets so we simply feed the decoder's predictions back to itself for each step.

Every time it predicts a word we add it to the output string, and if it predicts the EOS token we stop there. We also store the decoder's attention outputs for display later.

In [ ]:

```
def evaluate(input_lang, output_lang, encoder, decoder,
             sentence, max_length=MAX_LENGTH):
    with torch.no_grad():
        input_tensor = tensorFromSentence(input_lang, sentence)
        input_length = input_tensor.size()[0]
        encoder_hidden = encoder.initHidden()

        encoder_outputs = torch.zeros(max_length, encoder.hidden_size,
                                      device=device)

        for ei in range(input_length):
            encoder_output, encoder_hidden = encoder(input_tensor[ei],
                                                     encoder_hidden)
            encoder_outputs[ei] += encoder_output[0, 0]

        decoder_input = torch.tensor([[SOS_token]], device=device)  # SOS
```

```
        decoder_hidden = encoder_hidden

        decoded_words = []
        decoder_attentions = torch.zeros(max_length, max_length)

        for di in range(max_length):
            decoder_output, decoder_hidden, decoder_attention = decoder(
                decoder_input, decoder_hidden, encoder_outputs)
            decoder_attentions[di] = decoder_attention.data
            topv, topi = decoder_output.data.topk(1)
            if topi.item() == EOS_token:
                decoded_words.append('<EOS>')
                break
            else:
                decoded_words.append(output_lang.index2word[topi.item()])

            decoder_input = topi.squeeze().detach()

        return decoded_words, decoder_attentions[:di + 1]


# Selects a specified number of sentence pairs randomly from a dataset and
# evaluates the translation model on these pairs.
def evaluateRandomly(input_lang, output_lang, encoder, decoder, n=10):
    # n (default 10): The number of sentence pairs to randomly select and evaluate.
    for i in range(n):
        pair = random.choice(pairs)
        print('>', pair[0])
        print('=', pair[1])
        output_words, attentions = evaluate(input_lang, output_lang, encoder,
                                            decoder, pair[0])
        output_sentence = ' '.join(output_words)
        print('<', output_sentence)
        print('')
```

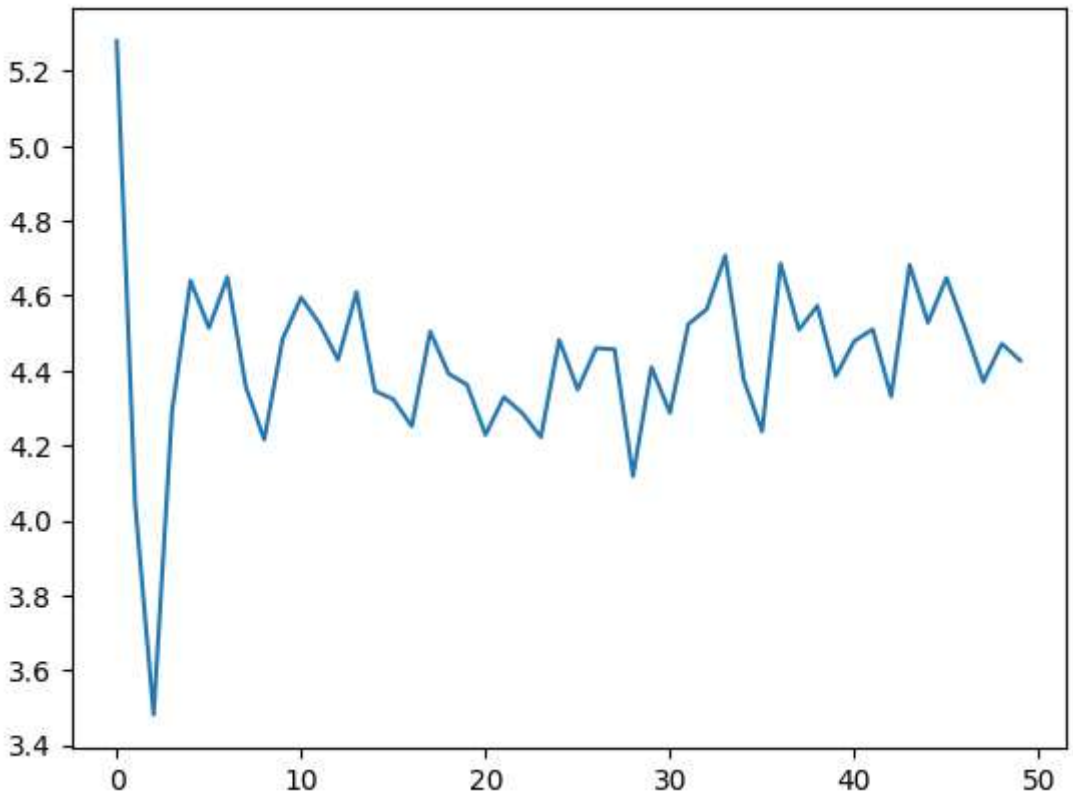
Training and Visualization

In [ ]:
```
# The resulting plot will be displayed directly
%matplotlib inline
```

In [ ]:
```
# Output is English

input_lang1, output_lang1, pairs = prepareData('kor', 'eng', True)
print(random.choice(pairs))

# Model Architecture
hidden_size = 20 # 256
encoder = EncoderRNN(input_lang1.n_words, hidden_size).to(device)
attn_decoder = AttnDecoderRNN(hidden_size, output_lang1.n_words,
                              dropout_p=0.1).to(device)

# Train the model
# trainIters(encoder1, attn_decoder1, 75000, print_every=5000)
```

```
trainIters(input_lang1, output_lang1, encoder, attn_decoder,
           5000, print_every=500)
```

```
Reading lines...
Read 5870 sentence pairs
Trimmed to 5261 sentence pairs
Counting words...
Counted words:
eng 6831
kor 3315
['톰은 삼십대 중반이야 .', 'Tom is in his mid-to-late thirties .']
0m 6s (- 1m 2s) (500 10%) 4.3475
0m 12s (- 0m 50s) (1000 20%) 4.4443
0m 20s (- 0m 46s) (1500 30%) 4.5010
0m 26s (- 0m 39s) (2000 40%) 4.3666
0m 32s (- 0m 32s) (2500 50%) 4.3097
0m 38s (- 0m 25s) (3000 60%) 4.3588
0m 45s (- 0m 19s) (3500 70%) 4.4917
0m 51s (- 0m 12s) (4000 80%) 4.4786
0m 58s (- 0m 6s) (4500 90%) 4.5063
1m 4s (- 0m 0s) (5000 100%) 4.4860
<Figure size 640x480 with 0 Axes>
```

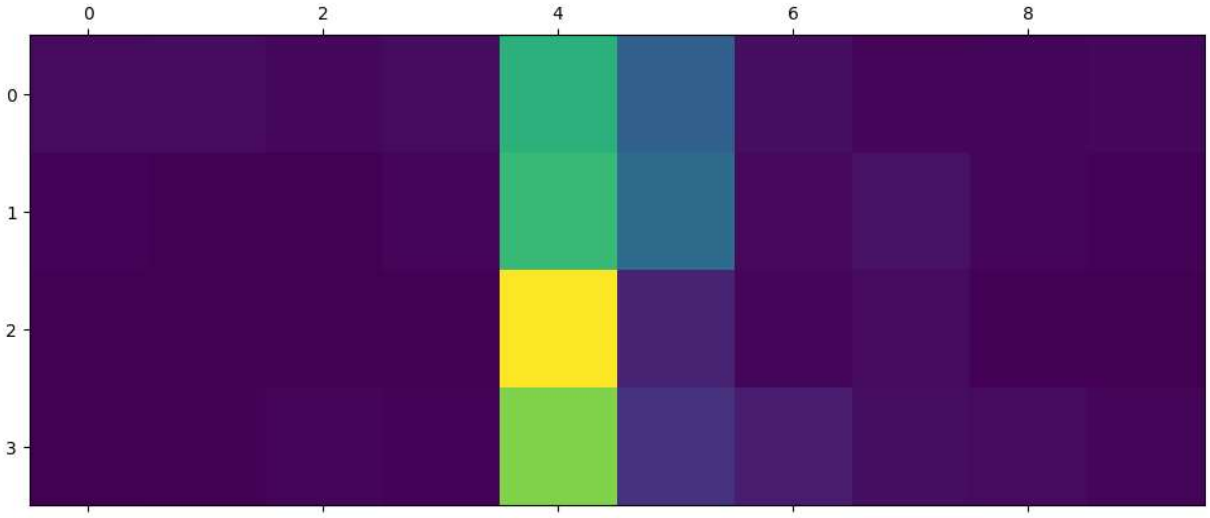


In [ ]:
```
evaluateRandomly(input_lang1, output_lang1, encoder, attn_decoder)
```

```
> 역이 꽤 멀다 .
= The station is pretty far .
< Tom is to . <EOS>

> 톰은 일에 집중하는 데 어려움을 겪었다 .
= Tom had difficulty concentrating on his work .
< Tom is to to to . <EOS>

> 너무 춥다 .
= It's pretty cold .
< Tom is . <EOS>

> 좋은 선물이 되겠다 .
= That would make a great gift .
< Tom is to . <EOS>

> 대부분의 사람들은 내가 미쳤다고 생각해
= Most people think I'm crazy .
< Tom is to to . <EOS>

> 난 여기가 만족스러웠었어 .
= I used to be happy here .
< Tom is . <EOS>

> 내 이메일 주소를 잊어버렸다 .
= I forgot my email address .
< Tom is to to . <EOS>

> 그녀는 정말로 미인이다 .
= She is a real beauty .
< Tom is to . <EOS>

> 톰을 고발하는 거야 ?
= Are you accusing Tom ?
< Tom is to to . <EOS>

> 이 시계는 정확하다 .
= This clock is accurate .
< Tom is to . <EOS>
```

In [ ]:
```
output_words, attentions = evaluate(
    input_lang1, output_lang1,
    encoder, attn_decoder, "바로 집으로 오세요")

print(output_words)
plt.matshow(attentions.numpy())
```

```
['Tom', 'is', '.', '<EOS>']
<matplotlib.image.AxesImage at 0x7caafb80de40>
```

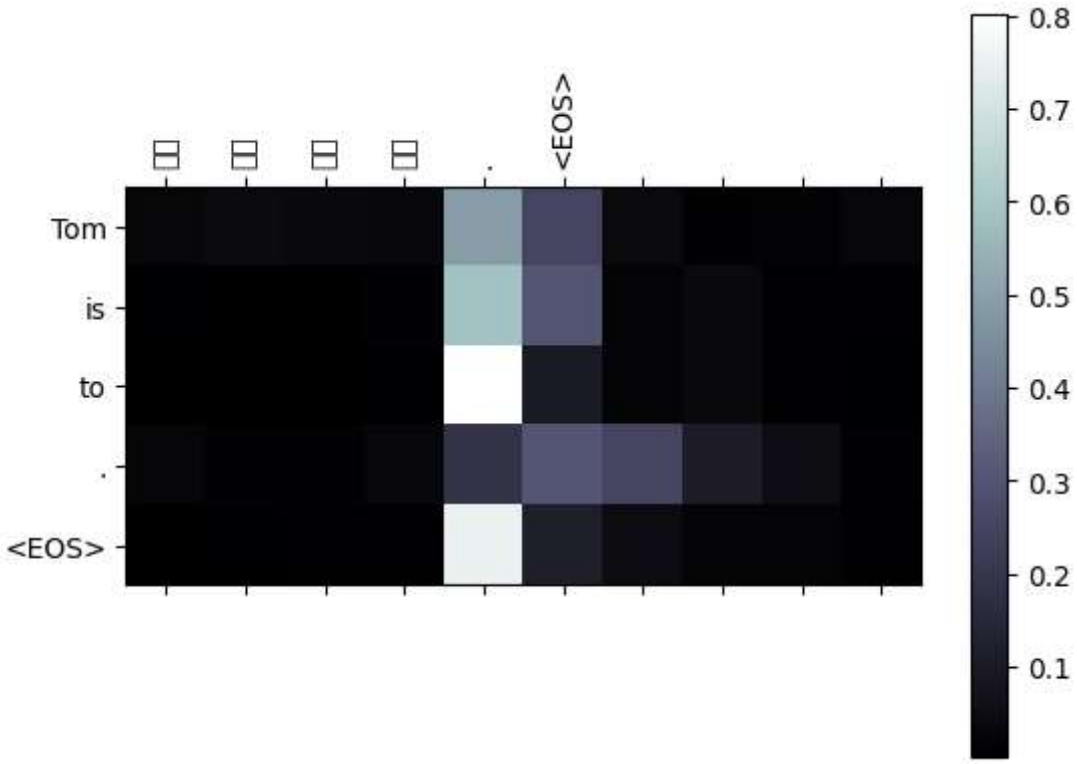


In [ ]:
```
# EVALUATE ON TEST SENTENCES

def showAttention(input_sentence, output_words, attentions):
    # Set up figure with colorbar
    fig = plt.figure()
    ax = fig.add_subplot(111)
    cax = ax.matshow(attentions.numpy(), cmap='bone')
    fig.colorbar(cax)

    # Set up axes
    ax.set_xticklabels(['']  input_sentence.split(' ') +
                       ['<EOS>'], rotation=90)
    ax.set_yticklabels(['']  output_words)

    # Show label at every tick
    ax.xaxis.set_major_locator(ticker.MultipleLocator(1))
    ax.yaxis.set_major_locator(ticker.MultipleLocator(1))
    #### remove to show plots
    #plt.show()


def evaluateAndShowAttention(input_lang, output_lang, input_sentence):
    input_sentenceR = input_sentence.translate(str.maketrans('', '',
                                                             string.punctuation))#.
    output_words, attentions = evaluate(
        input_lang, output_lang,
        encoder, attn_decoder, input_sentenceR)
    print('input =', input_sentence)
    print('output =', ' '.join(output_words))
    showAttention(input_sentence, output_words, attentions)


evaluateAndShowAttention(input_lang1, output_lang1, "이거 정말 기분 좋다 .")
evaluateAndShowAttention(input_lang1, output_lang1, "나는 톰에게 이 책을 줘야 합니다
evaluateAndShowAttention(input_lang1, output_lang1, "나는 거짓말 하지 않아 .")
```

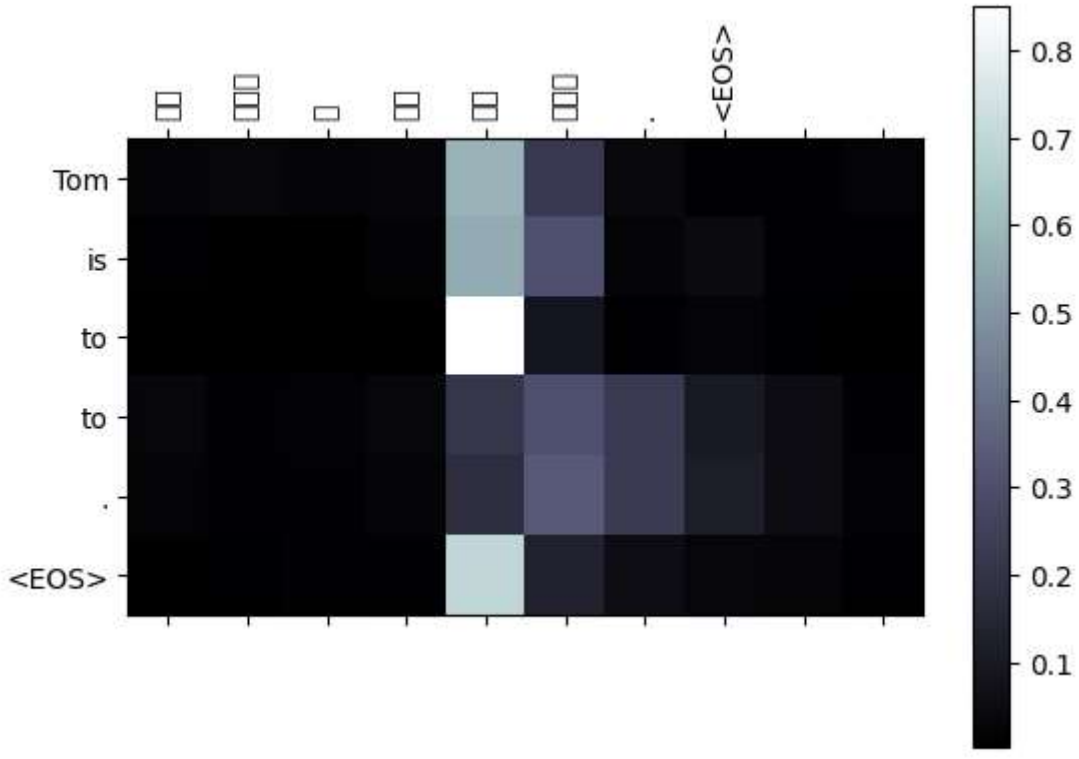
```
input = 이거 정말 기분 좋다 .
output = Tom is to . <EOS>
input = 나는 톰에게 이 책을 줘야 합니다 .
output = Tom is to to . <EOS>
input = 나는 거짓말 하지 않아 .
output = Tom is to . <EOS>
```

```
<ipython-input-16-b12ed4d22d77>:11: UserWarning: FixedFormatter should only be used
together with FixedLocator
  ax.set_xticklabels(['']  + input_sentence.split(' ') +
<ipython-input-16-b12ed4d22d77>:13: UserWarning: FixedFormatter should only be used
together with FixedLocator
  ax.set_yticklabels(['']  + output_words)
/usr/local/lib/python3.10/dist-packages/IPython/core/events.py:89: UserWarning: Glyp
h 51060 (\N{HANGUL SYLLABLE I}) missing from current font.
  func(*args, **kwargs)
/usr/local/lib/python3.10/dist-packages/IPython/core/events.py:89: UserWarning: Glyp
h 44144 (\N{HANGUL SYLLABLE GEO}) missing from current font.
  func(*args, **kwargs)
/usr/local/lib/python3.10/dist-packages/IPython/core/events.py:89: UserWarning: Glyp
h 51221 (\N{HANGUL SYLLABLE JEONG}) missing from current font.
  func(*args, **kwargs)
/usr/local/lib/python3.10/dist-packages/IPython/core/events.py:89: UserWarning: Glyp
h 47568 (\N{HANGUL SYLLABLE MAL}) missing from current font.
  func(*args, **kwargs)
/usr/local/lib/python3.10/dist-packages/IPython/core/events.py:89: UserWarning: Glyp
h 44592 (\N{HANGUL SYLLABLE GI}) missing from current font.
  func(*args, **kwargs)
/usr/local/lib/python3.10/dist-packages/IPython/core/events.py:89: UserWarning: Glyp
h 48516 (\N{HANGUL SYLLABLE BUN}) missing from current font.
  func(*args, **kwargs)
/usr/local/lib/python3.10/dist-packages/IPython/core/events.py:89: UserWarning: Glyp
h 51339 (\N{HANGUL SYLLABLE JOH}) missing from current font.
  func(*args, **kwargs)
/usr/local/lib/python3.10/dist-packages/IPython/core/events.py:89: UserWarning: Glyp
h 45796 (\N{HANGUL SYLLABLE DA}) missing from current font.
  func(*args, **kwargs)
/usr/local/lib/python3.10/dist-packages/IPython/core/events.py:89: UserWarning: Glyp
h 45208 (\N{HANGUL SYLLABLE NA}) missing from current font.
  func(*args, **kwargs)
/usr/local/lib/python3.10/dist-packages/IPython/core/events.py:89: UserWarning: Glyp
h 45716 (\N{HANGUL SYLLABLE NEUN}) missing from current font.
  func(*args, **kwargs)
/usr/local/lib/python3.10/dist-packages/IPython/core/events.py:89: UserWarning: Glyp
h 53680 (\N{HANGUL SYLLABLE TOM}) missing from current font.
  func(*args, **kwargs)
/usr/local/lib/python3.10/dist-packages/IPython/core/events.py:89: UserWarning: Glyp
h 50640 (\N{HANGUL SYLLABLE E}) missing from current font.
  func(*args, **kwargs)
/usr/local/lib/python3.10/dist-packages/IPython/core/events.py:89: UserWarning: Glyp
h 44172 (\N{HANGUL SYLLABLE GE}) missing from current font.
  func(*args, **kwargs)
/usr/local/lib/python3.10/dist-packages/IPython/core/events.py:89: UserWarning: Glyp
h 52293 (\N{HANGUL SYLLABLE CAEG}) missing from current font.
  func(*args, **kwargs)
/usr/local/lib/python3.10/dist-packages/IPython/core/events.py:89: UserWarning: Glyp
h 51012 (\N{HANGUL SYLLABLE EUL}) missing from current font.
  func(*args, **kwargs)
/usr/local/lib/python3.10/dist-packages/IPython/core/events.py:89: UserWarning: Glyp
h 51480 (\N{HANGUL SYLLABLE JWEO}) missing from current font.
  func(*args, **kwargs)
/usr/local/lib/python3.10/dist-packages/IPython/core/events.py:89: UserWarning: Glyp
h 50556 (\N{HANGUL SYLLABLE YA}) missing from current font.
```

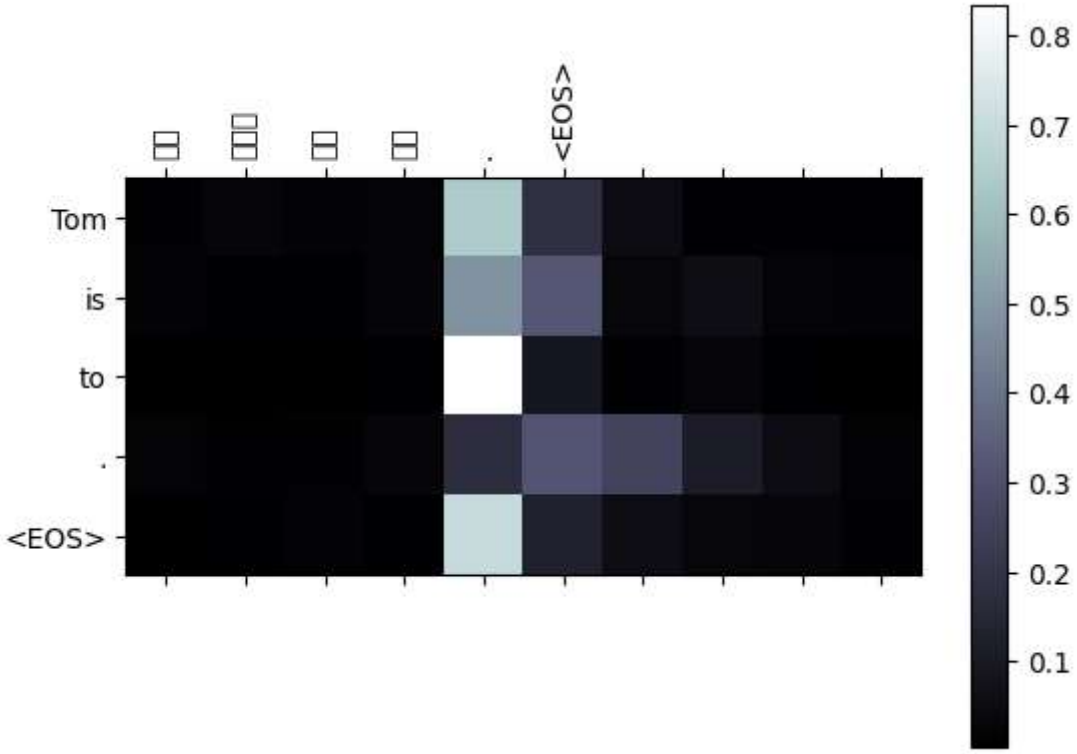
```
  func(*args, **kwargs)
/usr/local/lib/python3.10/dist-packages/IPython/core/events.py:89: UserWarning: Glyp
h 54633 (\N{HANGUL SYLLABLE HAB}) missing from current font.
  func(*args, **kwargs)
/usr/local/lib/python3.10/dist-packages/IPython/core/events.py:89: UserWarning: Glyp
h 45768 (\N{HANGUL SYLLABLE NI}) missing from current font.
  func(*args, **kwargs)
/usr/local/lib/python3.10/dist-packages/IPython/core/events.py:89: UserWarning: Glyp
h 51667 (\N{HANGUL SYLLABLE JIS}) missing from current font.
  func(*args, **kwargs)
/usr/local/lib/python3.10/dist-packages/IPython/core/events.py:89: UserWarning: Glyp
h 54616 (\N{HANGUL SYLLABLE HA}) missing from current font.
  func(*args, **kwargs)
/usr/local/lib/python3.10/dist-packages/IPython/core/events.py:89: UserWarning: Glyp
h 51648 (\N{HANGUL SYLLABLE JI}) missing from current font.
  func(*args, **kwargs)
/usr/local/lib/python3.10/dist-packages/IPython/core/events.py:89: UserWarning: Glyp
h 50506 (\N{HANGUL SYLLABLE ANH}) missing from current font.
  func(*args, **kwargs)
/usr/local/lib/python3.10/dist-packages/IPython/core/events.py:89: UserWarning: Glyp
h 50500 (\N{HANGUL SYLLABLE A}) missing from current font.
  func(*args, **kwargs)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning:
Glyph 51060 (\N{HANGUL SYLLABLE I}) missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning:
Glyph 44144 (\N{HANGUL SYLLABLE GEO}) missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning:
Glyph 51221 (\N{HANGUL SYLLABLE JEONG}) missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning:
Glyph 47568 (\N{HANGUL SYLLABLE MAL}) missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning:
Glyph 44592 (\N{HANGUL SYLLABLE GI}) missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning:
Glyph 48516 (\N{HANGUL SYLLABLE BUN}) missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning:
Glyph 51339 (\N{HANGUL SYLLABLE JOH}) missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning:
Glyph 45796 (\N{HANGUL SYLLABLE DA}) missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
```


□□
□□
□□
□□
.
<EOS>
Tom
is
to
.
<EOS>
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1

```
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning:
Glyph 45208 (\N{HANGUL SYLLABLE NA}) missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning:
Glyph 45716 (\N{HANGUL SYLLABLE NEUN}) missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning:
Glyph 53680 (\N{HANGUL SYLLABLE TOM}) missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning:
Glyph 50640 (\N{HANGUL SYLLABLE E}) missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning:
Glyph 44172 (\N{HANGUL SYLLABLE GE}) missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning:
Glyph 52293 (\N{HANGUL SYLLABLE CAEG}) missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning:
Glyph 51012 (\N{HANGUL SYLLABLE EUL}) missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning:
Glyph 51480 (\N{HANGUL SYLLABLE JWEO}) missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning:
Glyph 50556 (\N{HANGUL SYLLABLE YA}) missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning:
Glyph 54633 (\N{HANGUL SYLLABLE HAB}) missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning:
Glyph 45768 (\N{HANGUL SYLLABLE NI}) missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
```



```
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning:
Glyph 51667 (\N{HANGUL SYLLABLE JIS}) missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning:
Glyph 54616 (\N{HANGUL SYLLABLE HA}) missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning:
Glyph 51648 (\N{HANGUL SYLLABLE JI}) missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning:
Glyph 50506 (\N{HANGUL SYLLABLE ANH}) missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning:
Glyph 50500 (\N{HANGUL SYLLABLE A}) missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
```



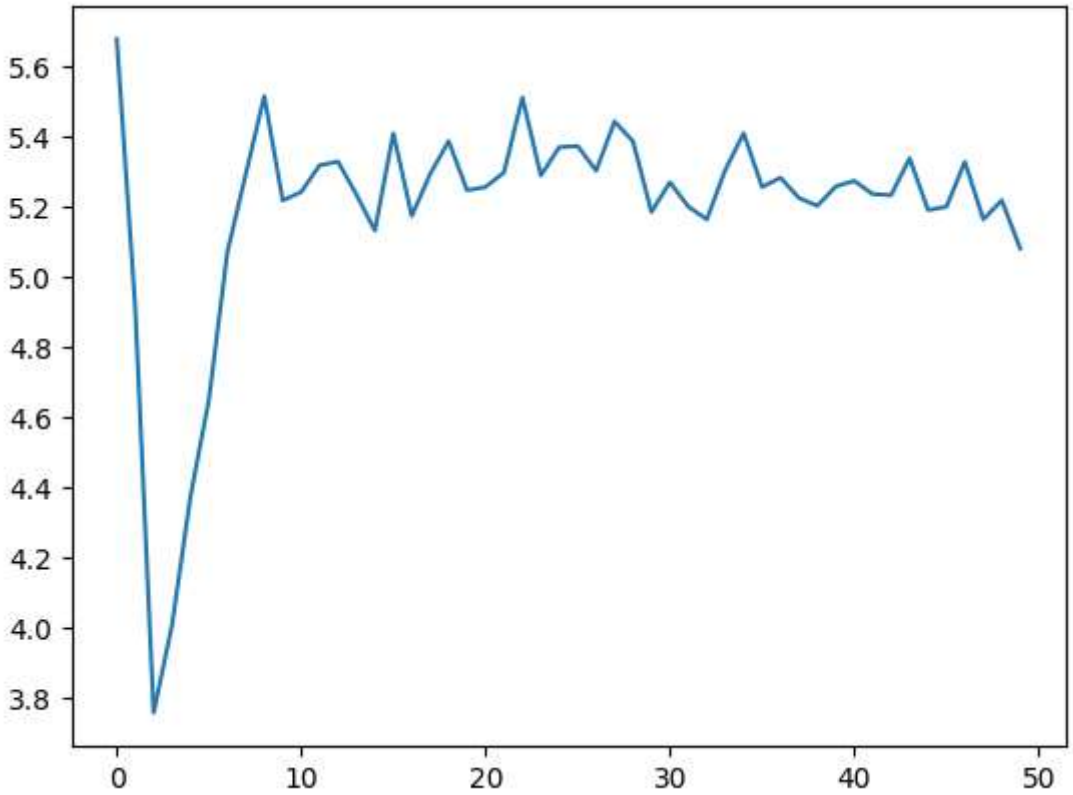
English to Korean Model

- Reverse Model: Now train another model (Model 2) for the reverse (i.e., from English to the language you chose).

In [ ]:
```
# Output is French (needed to be updated with Korean data)
input_lang2, output_lang2, pairs = prepareData('kor', 'eng', False)
print(random.choice(pairs))

# Model Architecture
hidden_size = 20 # 256
encoder = EncoderRNN(input_lang2.n_words, hidden_size).to(device)
attn_decoder = AttnDecoderRNN(hidden_size, output_lang2.n_words,
                              dropout_p=0.1).to(device)
```

```
# Train the model
# trainIters(encoder1, attn_decoder1, 75000, print_every=5000)
trainIters(input_lang2, output_lang2, encoder, attn_decoder, 5000,
           print_every=500)
```

```
Reading lines...
Read 5870 sentence pairs
Trimmed to 5261 sentence pairs
Counting words...
Counted words:
kor 3315
eng 6831
['Help me !', '도와줘 !']
0m 5s (- 0m 51s) (500 10%) 4.5453
0m 11s (- 0m 45s) (1000 20%) 5.1484
0m 17s (- 0m 40s) (1500 30%) 5.2498
0m 23s (- 0m 34s) (2000 40%) 5.3009
0m 29s (- 0m 29s) (2500 50%) 5.3435
0m 35s (- 0m 23s) (3000 60%) 5.3368
0m 42s (- 0m 18s) (3500 70%) 5.2683
0m 48s (- 0m 12s) (4000 80%) 5.2437
0m 54s (- 0m 6s) (4500 90%) 5.2530
1m 1s (- 0m 0s) (5000 100%) 5.1970
<Figure size 640x480 with 0 Axes>
```



- GloVe 100d embeddings: In this model, use the GloVe 100 dimensional embeddings. See notebook 4, cell 2 for an example while training.

In [ ]:
```
import torchtext.vocab
glove = torchtext.vocab.GloVe(name = '6B', dim = 100)
```

```
print(f'There are {len(glove.itos)} words in the vocabulary')
```

```
.vector_cache/glove.6B.zip: 862MB [02:42, 5.31MB/s]
100%|▉| 399999/400000 [00:20<00:00, 19697.27it/s]
There are 400000 words in the vocabulary
```

In [ ]:
```
# dimensions of vectors
print(glove.vectors.shape)

# top 10 words
print(glove.itos[:10])
```

```
torch.Size([400000, 100])
['the', ',', '.', 'of', 'to', 'and', 'in', 'a', '"', "'s"]
```

In [ ]:
```
# Import necessary libraries
import numpy as np

# Define a function to get word embeddings from GloVe
def get_embedding(word):
    return glove.vectors[glove.stoi[word]]

# From the list of pairs containing (English, Translation) pairs
# get a list of English words:
english_words = [pair[0] for pair in pairs]

# Create a vocabulary from the English words.
# We'll split the English sentences into words and collect unique words.
# Use a set to ensure uniqueness

english_vocabulary = set()
for sentence in english_words:
    words = sentence.split()
    english_vocabulary.update(words)

# Convert the set to a list to have an ordered vocabulary
english_vocabulary = list(english_vocabulary)

# Print the number of unique English words and some examples
print('Number of unique English words in our corpus:', len(english_vocabulary))
print('Some examples are', english_vocabulary[3000:3500])
```

```
Number of unique English words in our corpus: 3313
Some examples are ['time', 'slammed', 'disagree', 'Chicago', 'marched', 'alive', 'cl
ock', 'library', 'magazine', 'movies', 'shouting', 'yet', 'different', 'elementary',
'peace', 'Dancing', 'auction', 'gone', 'disorder', 'independent', 'Got', 'father',
'party', 'magnet', 'practices', 'another', 'perfect', 'Was', 'set', 'sightseeing',
'repainted', 'lie', 'addict', 'mother', 'wrapped', 'rope', 'research', 'stick', 'pol
iceman', 'hope', 'confessed', 'beers,', 'several', 'bedroom', 'mid-to-late', 'melte
d', 'staring', 'zoo', 'Education', 'president', 'blue', 'beer', 'son', 'Study', 'tal
kative', 'unfair', 'stop', 'sports', 'English', 'pencil', "You're", 'constantly', 'f
linched', 'listens', 'hurt', 'Thousands', 'gold', 'shenanigans', 'homework', 'draw',
'opened', 'Is', 'sells', 'moving', 'bloom', 'deserves', 'rather', 'quite', 'painkill
ers', 'There', 'panic', 'dog', 'evolution', 'judged', 'like', 'caught', 'open-minde
d', 'snores', 'daydreaming', 'surgeon', 'sulking', 'planets', 'slowly', 'cup', 'targ
et', 'felt', 'acts', 'scientific', 'explain', 'chemistry', 'Somebody', 'Everyone',
'French-speaking', 'stand', 'Where', 'Wash', 'high', 'primary', 'talent', 'others',
'big', 'cancelled', 'pizza', 'rescued', 'midnight', 'aroused', 'ugly', 'towards', 'y
ounger', 'Greta', 'relieve', 'reacted', 'helping', 'glad', 'Accidents', 'masculine',
'ran', 'dozen', 'oppose', 'front', 'most,', 'Swiss', 'bacteria', 'participants', 'to
tal', "I'm", 'beautiful', 'apologize', 'row', 'gift', 'Siberia', '10', 'between', 'A
fter', 'Harvard', 'all', 'husband', 'blind,', 'graduated', 'usually', 'together', 'a
mazing', 'B', 'visit', 'live', 'That', 'have', 'strong', 'fire', 'Australia,', 'Apol
ogize', 'Years', 'gray', 'than', 'certain', 'butter', 'eat,', 'coffee,', 'defend',
'talked', 'loaded', 'picnic', 'interpreter', "book's", 'fix', 'bird', 'continent',
'graduation', 'away,', 'salty', 'martial', 'is', 'grandparents', 'notice', 'passiona
te,', 'artist', 'helpful', 'May', 'refused', 'cane', 'astonished', 'trainee', 'nearb
y', 'fascinating', 'because', 'see', 'explained', 'against', 'toll-free', 'turkey',
'cats', 'games', 'Admission', 'thanked', 'salesman', 'feelings', 'Peel', 'sooner,',
'last', 'appears', 'believed', 'pair', 'jumped', 'warn', 'makes', 'simply', 'bette
r', 'Cup', 'quick', 'present', 'proof', 'lets', 'Beef,', 'case', 'infected', "ca
n't", 'demonstrate', 'matter', 'drank', "You've", 'come', 'optimistic', 'Canadian',
'Go', 'exit', 'Times', 'course', 'brain', 'reserve', 'cheated', 'wine,', 'Money', 'a
nyway', 'seven', 'table', 'plan', 'excuse', 'likes', 'convicted', 'weather,', 'middl
e-aged', 'do', 'religious', 'caffeine', 'church', 'swims', 'beaks', 'assured', '193
9', 'danger', 'repeat', 'cab', 'Halloween', 'weather', 'ballpoint', 'sweater', 'soo
n', 'forest', 'driver,', 'pasta', 'erase', 'self-respect', 'wall', 'headaches', 'six
-thirty', 'Thanks', 'better,', 'Japan', 'faith', 'bucket', 'sick,', 'temples', 'touc
h', 'eighteen', 'disliked', 'neighbor', 'some', 'Some', 'there', 'theater', 'been',
'attention,', 'cute', 'fault', 'shook', 'foretell', 'boiling', 'Love', 'shortcut',
'Happy', 'enough', 'indicted', 'Turn', 'dying', 'apples', 'player', 'tire', 'rest',
'How', 'fly', 'scares', 'boyfriend', 'carelessness']
```

In [ ]:
```python
# Get GloVe embeddings for the words in the English vocabulary:
emb_dim = len(glove.vectors[0])  # Dimension of GloVe embeddings
weights_matrix = np.zeros((len(english_vocabulary), emb_dim))

for i, word in enumerate(english_vocabulary):
    try:
        # Get the GloVe embedding for each word
        weights_matrix[i] = get_embedding(word)
    except KeyError:
        # Handle missing words by initializing with a random vector or zeros
        weights_matrix[i] = np.random.normal(scale=0.6, size=(emb_dim, ))
```

In [ ]:
```python
# Define a function to get word embeddings from GloVe
def get_embedding_glove(word):
    return glove.vectors[glove.stoi[word]]
```

```
# Get a list of all english words in our dataset
print("Number of pairs, or translated phrases, is:", len(pairs))
eng_words = [el[0] for el in pairs]
a = []
for word in eng_words:
  a += word.split()
eng_vocabulary = list(set(a))
print('Number of unique English words in our corpus:', len(eng_vocabulary))
print('Some examples are', eng_vocabulary[3000:3500])
```

```
Number of pairs, or translated phrases, is: 5261
Number of unique English words in our corpus: 3313
Some examples are ['time', 'slammed', 'disagree', 'Chicago', 'marched', 'alive', 'cl
ock', 'library', 'magazine', 'movies', 'shouting', 'yet', 'different', 'elementary',
'peace', 'Dancing', 'auction', 'gone', 'disorder', 'independent', 'Got', 'father',
'party', 'magnet', 'practices', 'another', 'perfect', 'Was', 'set', 'sightseeing',
'repainted', 'lie', 'addict', 'mother', 'wrapped', 'rope', 'research', 'stick', 'pol
iceman', 'hope', 'confessed', 'beers,', 'several', 'bedroom', 'mid-to-late', 'melte
d', 'staring', 'zoo', 'Education', 'president', 'blue', 'beer', 'son', 'Study', 'tal
kative', 'unfair', 'stop', 'sports', 'English', 'pencil', "You're", 'constantly', 'f
linched', 'listens', 'hurt', 'Thousands', 'gold', 'shenanigans', 'homework', 'draw',
'opened', 'Is', 'sells', 'moving', 'bloom', 'deserves', 'rather', 'quite', 'painkill
ers', 'There', 'panic', 'dog', 'evolution', 'judged', 'like', 'caught', 'open-minde
d', 'snores', 'daydreaming', 'surgeon', 'sulking', 'planets', 'slowly', 'cup', 'targ
et', 'felt', 'acts', 'scientific', 'explain', 'chemistry', 'Somebody', 'Everyone',
'French-speaking', 'stand', 'Where', 'Wash', 'high', 'primary', 'talent', 'others',
'big', 'cancelled', 'pizza', 'rescued', 'midnight', 'aroused', 'ugly', 'towards', 'y
ounger', 'Greta', 'relieve', 'reacted', 'helping', 'glad', 'Accidents', 'masculine',
'ran', 'dozen', 'oppose', 'front', 'most,', 'Swiss', 'bacteria', 'participants', 'to
tal', "I'm", 'beautiful', 'apologize', 'row', 'gift', 'Siberia', '10', 'between', 'A
fter', 'Harvard', 'all', 'husband', 'blind,', 'graduated', 'usually', 'together', 'a
mazing', 'B', 'visit', 'live', 'That', 'have', 'strong', 'fire', 'Australia,', 'Apol
ogize', 'Years', 'gray', 'than', 'certain', 'butter', 'eat,', 'coffee,', 'defend',
'talked', 'loaded', 'picnic', 'interpreter', "book's", 'fix', 'bird', 'continent',
'graduation', 'away,', 'salty', 'martial', 'is', 'grandparents', 'notice', 'passiona
te,', 'artist', 'helpful', 'May', 'refused', 'cane', 'astonished', 'trainee', 'nearb
y', 'fascinating', 'because', 'see', 'explained', 'against', 'toll-free', 'turkey',
'cats', 'games', 'Admission', 'thanked', 'salesman', 'feelings', 'Peel', 'sooner,',
'last', 'appears', 'believed', 'pair', 'jumped', 'warn', 'makes', 'simply', 'bette
r', 'Cup', 'quick', 'present', 'proof', 'lets', 'Beef,', 'case', 'infected', "ca
n't", 'demonstrate', 'matter', 'drank', "You've", 'come', 'optimistic', 'Canadian',
'Go', 'exit', 'Times', 'course', 'brain', 'reserve', 'cheated', 'wine,', 'Money', 'a
nyway', 'seven', 'table', 'plan', 'excuse', 'likes', 'convicted', 'weather,', 'middl
e-aged', 'do', 'religious', 'caffeine', 'church', 'swims', 'beaks', 'assured', '193
9', 'danger', 'repeat', 'cab', 'Halloween', 'weather', 'ballpoint', 'sweater', 'soo
n', 'forest', 'driver,', 'pasta', 'erase', 'self-respect', 'wall', 'headaches', 'six
-thirty', 'Thanks', 'better,', 'Japan', 'faith', 'bucket', 'sick,', 'temples', 'touc
h', 'eighteen', 'disliked', 'neighbor', 'some', 'Some', 'there', 'theater', 'been',
'attention,', 'cute', 'fault', 'shook', 'foretell', 'boiling', 'Love', 'shortcut',
'Happy', 'enough', 'indicted', 'Turn', 'dying', 'apples', 'player', 'tire', 'rest',
'How', 'fly', 'scares', 'boyfriend', 'carelessness']
```

In [ ]:
```
# Get GloVe embeddings for the words in the English vocabulary:
emb_dim = len(glove.vectors[0])  # Dimension of GloVe embeddings
# emb_dim = 100
```

```
# Get the number of words in vocabulary
# matrix_len = len(input_lang2.word2index)
matrix_len = input_lang2.n_words
# weights_matrix = np.zeros((len(english_vocabulary), emb_dim))


# Initialize the weight matrix with zeros
weights_matrix = np.zeros((matrix_len, emb_dim))

# Counter for tracking the number of words with embeddings found
words_found = 0

# Loop through each word in your vocabulary
for i, word in enumerate(input_lang2.word2index.keys()):
    try:
        # Try to get the GloVe embedding for the word
        weights_matrix[i] = get_embedding_glove(word)
        words_found += 1
    except KeyError:
        # If the word is not found in the GloVe embeddings,
        # initialize it with a random vector
        weights_matrix[i] = np.random.normal(scale=0.6, size=(emb_dim, ))


print(weights_matrix.shape)
```

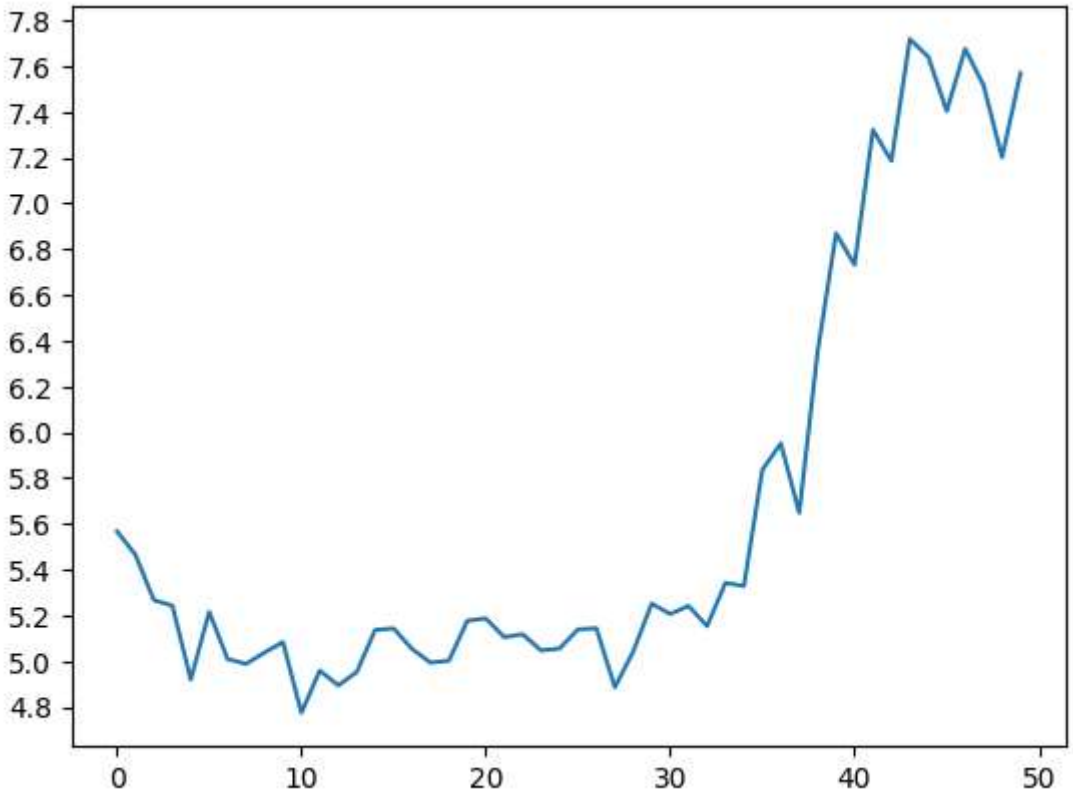
```
(3315, 100)
```

We must build a matrix of weights that will be loaded into the PyTorch embedding layer. Its shape will be equal to: (dataset's vocabulary length, word vectors dimension). For each word in dataset's vocabulary, we check if it is on GloVe's vocabulary. If it do it, we load its pre-trained word vector. Otherwise, we initialize a random vector.

In [ ]:
```
# Set a flag 'glove_model' to True if you want to use pre-trained GloVe embeddings
glove_model = True

# Define the hidden size for the model
hidden_size = 100

# Create an embedding layer
embedding = nn.Embedding(input_lang2.n_words, hidden_size)

# If 'glove_model' is True, initialize the embedding layer with
# pre-trained GloVe embeddings
if glove_model:
    embedding.weight.data.copy_(torch.from_numpy(weights_matrix))

# Create an encoder and an attention-based decoder
encoder2 = EncoderRNN(input_lang2.n_words, hidden_size).to(device)
attn_decoder2 = AttnDecoderRNN(hidden_size, output_lang2.n_words,
                               dropout_p=0.1).to(device)

# TRAIN MODEL
# Assuming 'trainIters' is a function that trains the model
```

```
trainIters(input_lang2, output_lang2,
           encoder2, attn_decoder2, 5000, print_every=500, learning_rate=0.1)
```

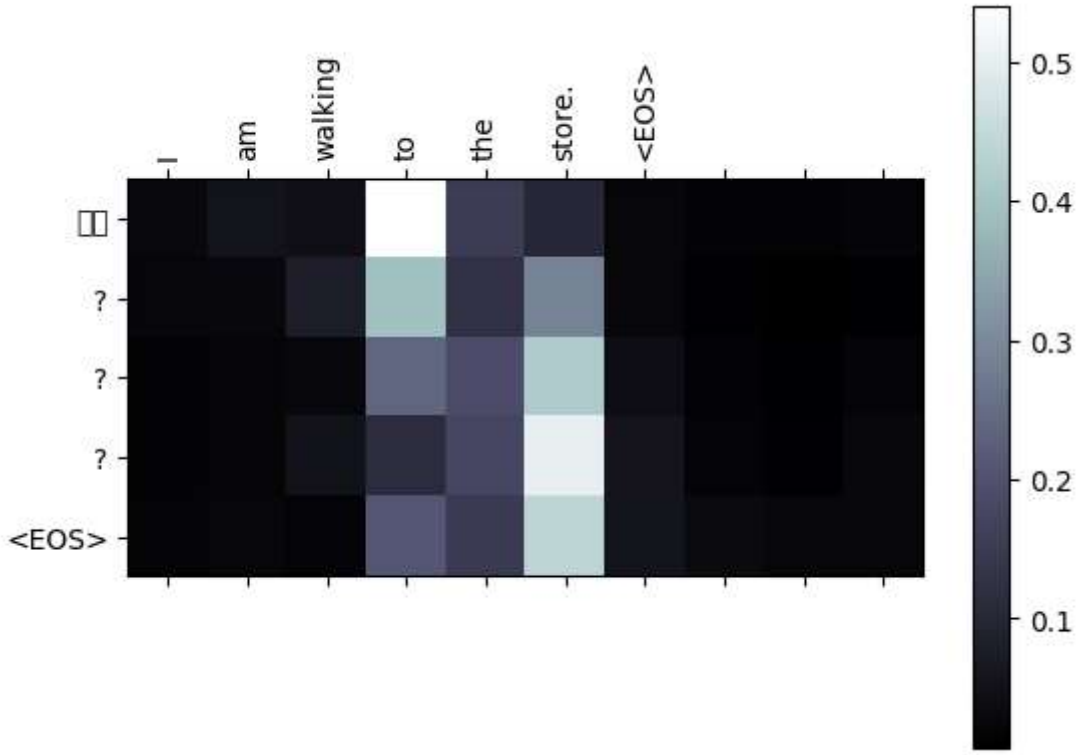
```
0m 7s (- 1m 4s) (500 10%) 5.2916
0m 14s (- 0m 58s) (1000 20%) 5.0660
0m 21s (- 0m 49s) (1500 30%) 4.9426
0m 28s (- 0m 42s) (2000 40%) 5.0732
0m 34s (- 0m 34s) (2500 50%) 5.1017
0m 41s (- 0m 27s) (3000 60%) 5.0919
0m 48s (- 0m 20s) (3500 70%) 5.2536
0m 55s (- 0m 13s) (4000 80%) 6.1315
1m 2s (- 0m 6s) (4500 90%) 7.3201
1m 8s (- 0m 0s) (5000 100%) 7.4734
<Figure size 640x480 with 0 Axes>
```



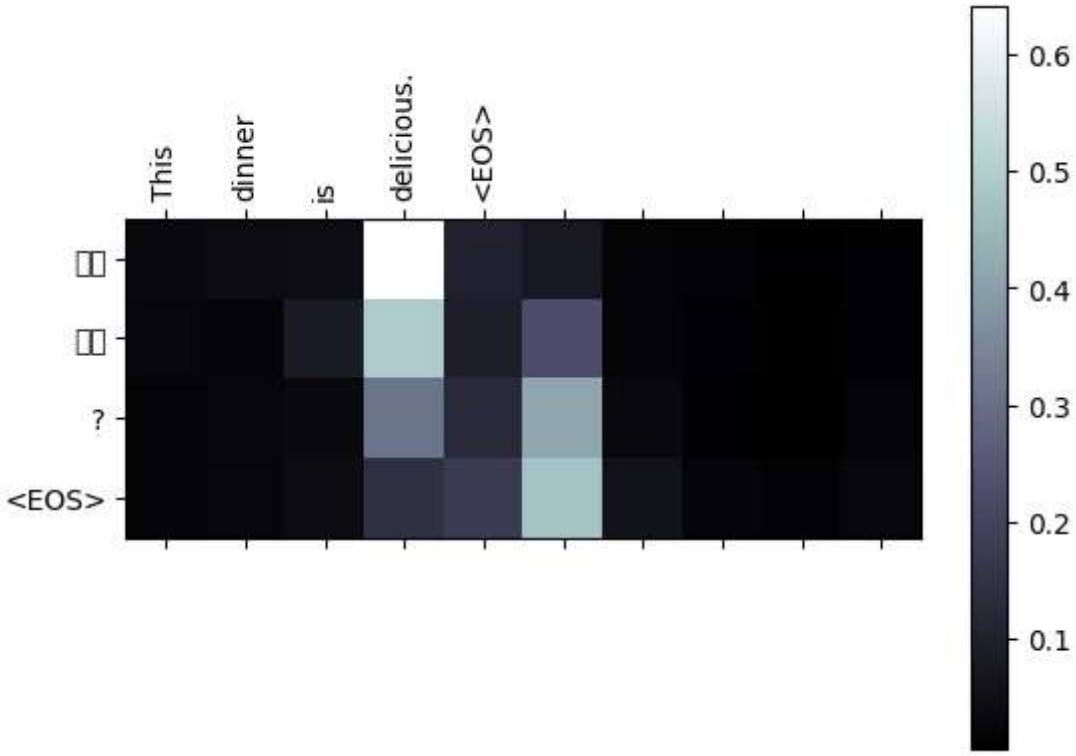
In [ ]:
```
# EVALUATE
evaluateRandomly(input_lang2, output_lang2, encoder2, attn_decoder2)

# TEST ON FIVE SENTENCES
evaluateAndShowAttention(input_lang2, output_lang2, "I am walking to the store.")
evaluateAndShowAttention(input_lang2, output_lang2, "This dinner is delicious.")
evaluateAndShowAttention(input_lang2, output_lang2,
                         "My mother lived in a white house.")
evaluateAndShowAttention(input_lang2, output_lang2, "He has three friends.")
evaluateAndShowAttention(input_lang2, output_lang2, "I love good red wine.")
```

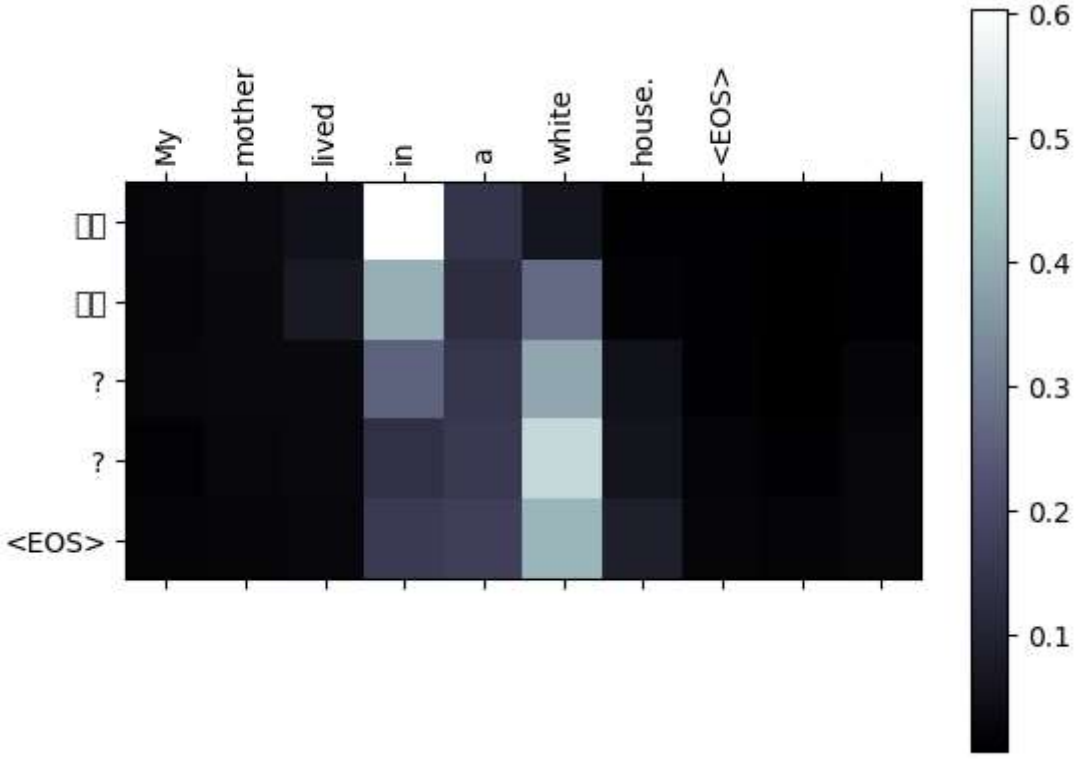
```
> Why should I give this to you ?
= 왜 내가 이걸 너한테 줘야 해 ?
< 너 ? <EOS>

> How beautiful !
= 이렇게나 아름다울 수가 !
< 정말 <EOS>

> Didn't I tell you to close the door ?
= 너한테 문 닫았다고 말해주지 않았나 ?
< 너는 <EOS>

> Could you please repeat what you just said ?
= 방금 말씀하신 것을 다시 한번 말씀해 주시겠어요 ?
< 나는 <EOS>

> Go straight .
= 앞으로 쭉 가세요 .
< 난 . <EOS>

> I'm seeing a trend here .
= 나는 여기서 유행(경향)을 본다 .
< 나는 없어 . <EOS>

> Definitely !
= 절대로 !
< 정말 <EOS>

> Where will you go ?
= 어디로 갈 거야 ?
< 어디로 ? <EOS>

> French is fascinating .
= 프랑스어는 매력적이야 .
< 너는 <EOS>

> I feel bad .
= 기분 나빠 .
< 고치는 것은 <EOS>

input = I am walking to the store.
output = 톰은 ? ? ? <EOS>
input = This dinner is delicious.
output = 톰은 내가 ? <EOS>
<ipython-input-16-b12ed4d22d77>:11: UserWarning: FixedFormatter should only be used
together with FixedLocator
  ax.set_xticklabels([''] + input_sentence.split(' ') +
<ipython-input-16-b12ed4d22d77>:13: UserWarning: FixedFormatter should only be used
together with FixedLocator
  ax.set_yticklabels([''] + output_words)
input = My mother lived in a white house.
output = 톰은 내가 ? ? <EOS>
input = He has three friends.
output = 톰은 내가 ? <EOS>
input = I love good red wine.
output = 톰은 내가 . <EOS>
```

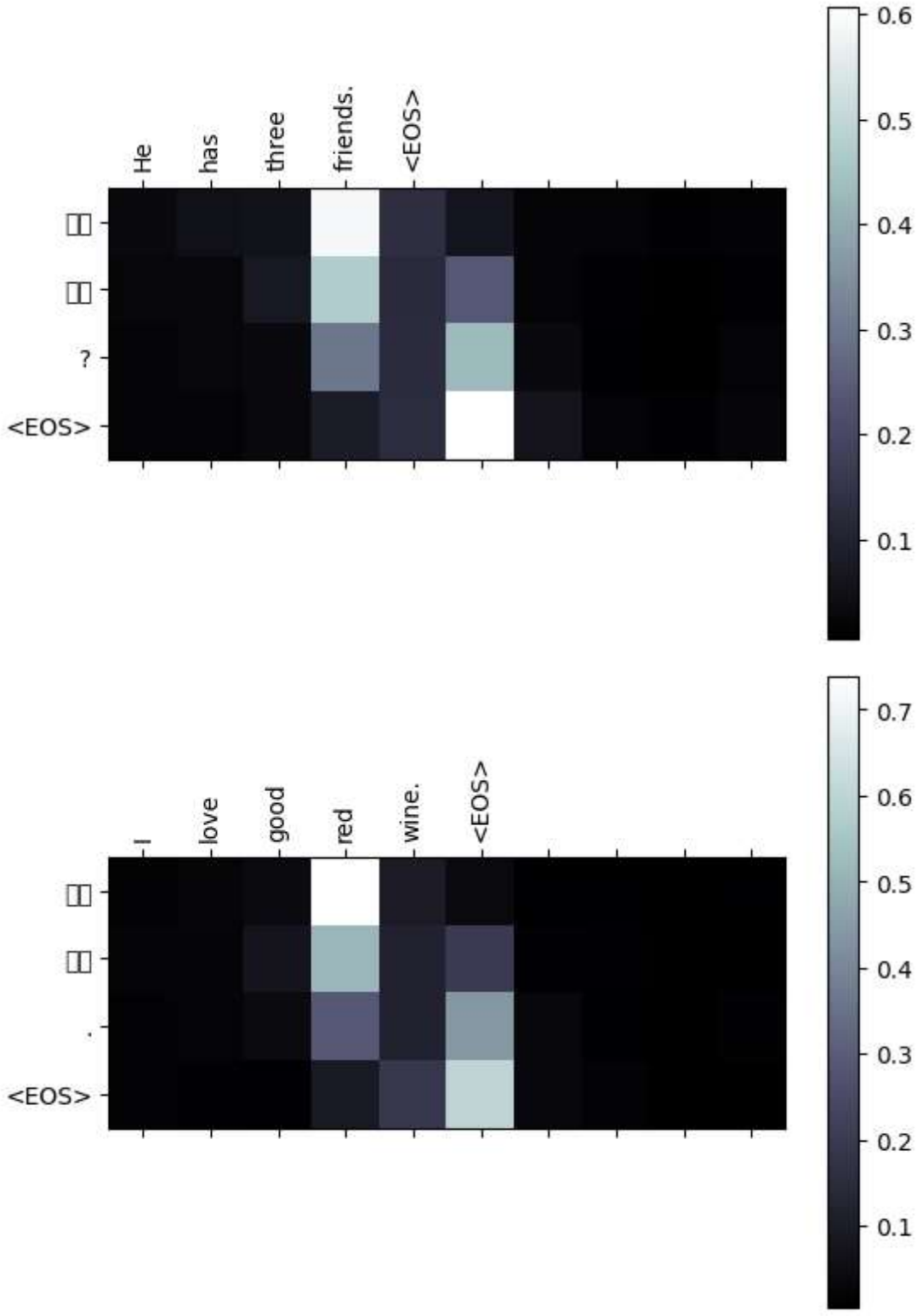
```
/usr/local/lib/python3.10/dist-packages/IPython/core/events.py:89: UserWarning: Glyp
h 53680 (\N{HANGUL SYLLABLE TOM}) missing from current font.
  func(*args, **kwargs)
/usr/local/lib/python3.10/dist-packages/IPython/core/events.py:89: UserWarning: Glyp
h 51008 (\N{HANGUL SYLLABLE EUN}) missing from current font.
  func(*args, **kwargs)
/usr/local/lib/python3.10/dist-packages/IPython/core/events.py:89: UserWarning: Glyp
h 45236 (\N{HANGUL SYLLABLE NAE}) missing from current font.
  func(*args, **kwargs)
/usr/local/lib/python3.10/dist-packages/IPython/core/events.py:89: UserWarning: Glyp
h 44032 (\N{HANGUL SYLLABLE GA}) missing from current font.
  func(*args, **kwargs)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning:
Glyph 53680 (\N{HANGUL SYLLABLE TOM}) missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning:
Glyph 51008 (\N{HANGUL SYLLABLE EUN}) missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
```




```
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning:
Glyph 45236 (\N{HANGUL SYLLABLE NAE}) missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning:
Glyph 44032 (\N{HANGUL SYLLABLE GA}) missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
```


This
dinner
is
delicious.
<EOS>
?
<EOS>
0.6
0.5
0.4
0.3
0.2
0.1


My
mother
lived
in
a
white
house.
<EOS>
?
?
<EOS>
0.6
0.5
0.4
0.3
0.2
0.1



Compare outputs

- Input 5 well formed sentences from the English vocab to Model 2, and input the resultant translated sentences to Model 1. Display all model outputs in each case.

In [ ]:
```
# Run Model 2 (English to Korean Glove model) to generate Korean sentence
# Set a flag 'glove_model' to True if you want to
```

```
# use pre-trained GloVe embeddings
glove_model = True

# Input sentence in English
input_sentence = random.choice(pairs)[0]
print('Original input sentence before translation:', input_sentence)

# Evaluate the input sentence using the pre-trained model
output_words2, attentions2 = evaluate(input_lang2, output_lang2,
                                      encoder2, attn_decoder2, input_sentence)

# Remove the end-of-sentence token from the generated output
output_words2 = output_words2[:-1]

# Join the output words into a single sentence
output_sentence_m2 = " ".join(output_words2)

# Print the English to Korean translation
print('English to Korean translation:', output_sentence_m2)
```

```
Original input sentence before translation: I forgot .
English to Korean translation: 고치는
```

In [ ]:

```
# Run Model 1 generate English sentence

# Set a flag 'glove_model' to True if you want to use pre-trained GloVe embeddings
glove_model = False

# Input sentence in English
# input_sentence = output_sentence_m2

# Evaluate the input sentence using the pre-trained model
output_words1, attentions1 = evaluate(input_lang1, output_lang1,
                                      encoder, attn_decoder, output_sentence_m2)

# Remove the end-of-sentence token from the generated output
output_words1 = output_words1[:-1]

# Join the output words into a single sentence
output_sentence_m2_m1 = " ".join(output_words1)

# Print the English to Korean translation
print('Korean back to English translation:', output_sentence_m2_m1)
```

```
Korean back to English translation: a ?
```