AAI 520 - Assignment 7

Ricardo Barbosa

October 21, 2024

[1]:
```python
import torch
import torch.nn as nn
from torch import optim
import torch.nn.functional as F
import csv
import random
import re
import os
import unicodedata
import codecs
import itertools
```

[2]:
```python
# define the device to use
device = torch.device("cpu")
```

[3]:
```python
lines_filepath = os.path.join('cornell movie-dialogs corpus', '/Users/bandito/
↪Documents/usdjourney/archive/movie_lines.txt')
conv_filepath = os.path.join('cornell movie-dialogs corpus', "/Users/bandito/
↪Documents/usdjourney/archive/movie_conversations.txt")
```

[4]:
```python
# visualize a few lines
with open(lines_filepath, 'rb') as file:
    lines = file.readlines()
for line in lines[:8]:
    print(line.strip())
```

```
b'L1045 +++$+++ u0 +++$+++ m0 +++$+++ BIANCA +++$+++ They do not!'
b'L1044 +++$+++ u2 +++$+++ m0 +++$+++ CAMERON +++$+++ They do to!'
b'L985 +++$+++ u0 +++$+++ m0 +++$+++ BIANCA +++$+++ I hope so.'
b'L984 +++$+++ u2 +++$+++ m0 +++$+++ CAMERON +++$+++ She okay?'
b"L925 +++$+++ u0 +++$+++ m0 +++$+++ BIANCA +++$+++ Let's go."
b'L924 +++$+++ u2 +++$+++ m0 +++$+++ CAMERON +++$+++ Wow'
b"L872 +++$+++ u0 +++$+++ m0 +++$+++ BIANCA +++$+++ Okay -- you're gonna need to
learn how to lie."
b'L871 +++$+++ u2 +++$+++ m0 +++$+++ CAMERON +++$+++ No'
```

[5]:
```python
# split each line of the file into a dictionary
line_fields = ['lineID', 'characterID', 'movieID','character', 'text']
```

```
lines = {}
with open(lines_filepath, 'r', encoding='iso-8859-1') as f:
    for line in f:
        values = line.split(' +++$+++ ')
        # Extract fields
        lineObj = {}
        for i, field in enumerate(line_fields):
            lineObj[field] = values[i]
        lines[lineObj['lineID']] = lineObj
```

[6]:

```
dict(itertools.islice(lines.items(), 3)) # displays first 3 items
```

[6]:

```
{'L1045': {'lineID': 'L1045',
  'characterID': 'u0',
  'movieID': 'm0',
  'character': 'BIANCA',
  'text': 'They do not!\n'},
 'L1044': {'lineID': 'L1044',
  'characterID': 'u2',
  'movieID': 'm0',
  'character': 'CAMERON',
  'text': 'They do to!\n'},
 'L985': {'lineID': 'L985',
  'characterID': 'u0',
  'movieID': 'm0',
  'character': 'BIANCA',
  'text': 'I hope so.\n'}}
```

[7]:

```
# split each line of the file into a dictionary of fields
conv_fields = ['character1ID', 'character2ID','movieID','utteranceIDs']
conversations = []
with open(conv_filepath, 'r', encoding='iso-8859-1') as f:
    for line in f:
        values = line.split(' +++$+++ ')

        # Extract fields
        convObj = {}
        for i, field in enumerate(conv_fields):
            convObj[field] = values[i]

        # Convert string result from split to list
        lineIds = eval(convObj['utteranceIDs'])

        #Reassemble lines
        convObj['lines'] = []
        for lineId in lineIds:
            convObj['lines'].append(lines[lineId])
```

```
        conversations.append(convObj)
```

[8]:
```
# QA pairs
qa_pairs = []
for conv in conversations:
    #iterate over all lines
    for i in range(len(conv['lines'])-1):
        inputLine = conv['lines'][i]['text'].strip()
        targetLine = conv['lines'][i+1]['text'].strip()
        # filter wrong samples if any
        if inputLine and targetLine:
            qa_pairs.append([inputLine, targetLine])
```

[9]:
```
len(qa_pairs)
```

[9]: 221282

[10]:
```
# writing the Question Answer pairs to a file
datafile = os.path.join('cornell movie-dialogs corpus', '/Users/bandito/
↪Documents/usdjourney/formatted_movie_lines.txt')
delimiter = '\t'
#unescape the delimiter
delimiter = str(codecs.decode(delimiter, 'unicode_escape'))

#write new csv file
print('\nWriting formatted file')
with open(datafile, 'w', encoding='utf-8') as f:
    writer = csv.writer(f, delimiter=delimiter)
    for pair in qa_pairs:
        writer.writerow(pair)
print('Done')
```

```
Writing newly formatted file
Done writing to a file
```

[11]:
```
# visualize
datafile = os.path.join('cornell movie-dialogs corpus', '/Users/bandito/
↪Documents/usdjourney/formatted_movie_lines.txt')
with open(datafile, 'r') as file:
    lines = file.readlines()
for line in lines[:8]:
    print(line)
```

```
Can we make this quick?  Roxanne Korrine and Andrew Barrett are having an
incredibly horrendous public break- up on the quad.  Again.   Well, I thought
we'd start with pronunciation, if that's okay with you.
```

```
Well, I thought we'd start with pronunciation, if that's okay with you. Not the
hacking and gagging and spitting part.  Please.

Not the hacking and gagging and spitting part.  Please. Okay... then how 'bout
we try out some French cuisine.  Saturday?  Night?

You're asking me out.  That's so cute. What's your name again?  Forget it.

No, no, it's my fault -- we didn't have a proper introduction ---       Cameron.

Cameron.        The thing is, Cameron -- I'm at the mercy of a particularly
hideous breed of loser.  My sister.  I can't date until she does.

The thing is, Cameron -- I'm at the mercy of a particularly hideous breed of
loser.  My sister.  I can't date until she does.   Seems like she could get a
date easy enough...

Why?    Unsolved mystery.  She used to be really popular when she started high
school, then it was just like she got sick of it or something.
```

[12]:
```python
PAD_token = 0 # used for padding
SOS_token = 1 # start of sentence
EOS_token = 2 # End of sentence token

class Vocabulary:
    def __init__(self, name):
        self.name = name
        self.word2index = {}
        self.word2count = {}
        self.index2word = {PAD_token:'PAD', SOS_token:'SOS', EOS_token:'EOS'}
        self.num_words = 3 #count tokens

    def addSentence(self, sentence):
        for word in sentence.split(' '):
            self.addWord(word)

    def addWord(self, word):
        if word not in self.word2index:
            self.word2index[word] = self.num_words
            self.word2count[word] = 1
            self.index2word[self.num_words] = word
            self.num_words +=1
        else:
            self.word2count[word] += 1
```

```
    def trim(self, min_count):
        keep_words = []
        for k,v in self.word2count.items():
            if v >= min_count:
                keep_words.append(k)
        print('Keep_words {} / {} = {:.4f}'.format(len(keep_words),len(self.
↪word2index),len(keep_words)/len(self.word2index)))

        # reinitialize dictionaries
        self.word2index = {}
        self.word2count = {}
        self.index2word = {PAD_token:'PAD', SOS_token:'SOS', EOS_token:'EOS'}
        self.num_words = 3 # sanity check

        for word in keep_words:
            self.addWord(word)
```

[13]:
```
# Unicode to ASCII
def unicodeToAscii(s):
    return ''.join(c for c in unicodedata.normalize('NFD', s) if unicodedata.
↪category(c) != 'Mn')
```

[14]:
```
# Test the function
unicodeToAscii('Montréal')
```

[14]: 'Montreal'

[15]:
```
# string processing
def normalizeString(s):
    s = unicodeToAscii(s.lower().strip())
    # replace whitespace + symbol
    s = re.sub(r'([.!?])', r' \1', s)
    # remove non-sequence chars
    s = re.sub(r'[^a-zA-Z.!?]+', r' ', s)
    # remove whitespace
    s = re.sub(r'\s+', r' ', s).strip()
    return s
```

[16]:
```
# visualize result
normalizeString('aaa243998!s"s  df?? ? 1a')
```

[16]: 'aaa !s s df ? ? ? a'

[17]:
```
datafile = os.path.join('cornell movie-dialogs corpus', '/Users/bandito/
↪Documents/usdjourney/formatted_movie_lines.txt')
# Read file and split lines
print('Reading and processing file....')
```

```
lines = open(datafile, encoding='utf-8').read().strip().split('\n')
# split line into pairs
pairs = [[normalizeString(s) for s in pair.split('\t')] for  pair in lines]
print('Complete!')
voc = Vocabulary('cornell movie-dialogs corpus')
```

```
Reading and processing file...please wait
Done Reading!
```

[18]:
```
pairs[:5]
```

[18]:
```
[['can we make this quick ? roxanne korrine and andrew barrett are having an
incredibly horrendous public break up on the quad . again .',
  'well i thought we d start with pronunciation if that s okay with you .'],
 ['well i thought we d start with pronunciation if that s okay with you .',
  'not the hacking and gagging and spitting part . please .'],
 ['not the hacking and gagging and spitting part . please .',
  'okay . . . then how bout we try out some french cuisine . saturday ? night
?'],
 ['you re asking me out . that s so cute . what s your name again ?',
  'forget it .'],
 ['no no it s my fault we didn t have a proper introduction', 'cameron .']]
```

[19]:
```
# returns true if both sentences in a pair are under MAX_LENGTH
MAX_LENGTH = 10
def filterPair(p):
    return len(p[0].split()) < MAX_LENGTH and len(p[1].split()) < MAX_LENGTH

# filter pairs using condition
def filterPairs(pairs):
    return [pair for pair in pairs if filterPair(pair)]
```

[20]:
```
pairs = [pair for pair in pairs if len(pair)>1]
print(f'There are {len(pairs)} pairs/conversations.')
pairs = filterPairs(pairs)
print(f'There are {len(pairs)} filtered pairs/conversations.')
```

```
There are 221282 pairs/conversations in the dataset.
After filtering, there are 64271 pairs/conversations in the dataset
```

[21]:
```
for pair in pairs:
    voc.addSentence(pair[0])
    voc.addSentence(pair[1])
print('Counted words:', voc.num_words)
for pair in pairs[:10]:
    print(pair)
```

```
Counted words: 18008
['there .', 'where ?']
```

```
['you have my word . as a gentleman', 'you re sweet .']
['hi .', 'looks like things worked out tonight huh ?']
['you know chastity ?', 'i believe we share an art instructor']
['have fun tonight ?', 'tons']
['well no . . .', 'then that s all you had to say .']
['then that s all you had to say .', 'but']
['but', 'you always been this selfish ?']
['do you listen to this crap ?', 'what crap ?']
['what good stuff ?', 'the real you .']
```

[22]:

```python
MIN_COUNT = 3 # minimum word count for trimming

def trimRareWords(voc, pairs, MIN_COUNT):
    # trim words used under the MIN_COUNT
    voc.trim(MIN_COUNT)
    # filter out pairs with trimmed words
    keep_pairs = []
    for pair in pairs:
        input_sentence = pair[0]
        output_sentence = pair[1]
        keep_input = True
        keep_output = True
        # check input sentence
        for word in input_sentence.split():
            if word not in voc.word2index:
                keep_input = False
                break
        # check output sentence
        for word in output_sentence.split():
            if word not in voc.word2index:
                keep_output = False
                break

        if keep_input and keep_output:
            keep_pairs.append(pair)

    print('Trimed from {} pairs to {}, {:.4f} of total'.
 ↪format(len(pairs),len(keep_pairs),len(keep_pairs)/len(pairs)))
    return keep_pairs

# Trim voc and pairs
pairs = trimRareWords(voc, pairs, MIN_COUNT)
```

```
Keep_words 7823 / 18005 = 0.4345
Trimed from 64271 pairs to 53165, 0.8272 of total
```

1 Data Preparation

[23]:
```
def indexesFromSentence(voc, sentence):
    return [voc.word2index[word] for word in sentence.split()]+[EOS_token]
```

[24]:
```
# visualize
indexesFromSentence(voc, pairs[1][0])
```

[24]:
```
[7, 8, 9, 10, 4, 11, 12, 13, 2]
```

[25]:
```
inp = []
out = []
for pair in pairs[:10]:
    inp.append(pair[0])
    out.append(pair[1])
print(inp)
print(len(inp))
indexes = [indexesFromSentence(voc, sent) for sent in inp]
indexes
```

```
['there .', 'you have my word . as a gentleman', 'hi .', 'have fun tonight ?',
'well no . . .', 'then that s all you had to say .', 'but', 'do you listen to
this crap ?', 'what good stuff ?', 'wow']
10
```

[25]:
```
[[3, 4, 2],
 [7, 8, 9, 10, 4, 11, 12, 13, 2],
 [16, 4, 2],
 [8, 31, 22, 6, 2],
 [33, 34, 4, 4, 4, 2],
 [35, 36, 37, 38, 7, 39, 40, 41, 4, 2],
 [42, 2],
 [47, 7, 48, 40, 45, 49, 6, 2],
 [50, 51, 52, 6, 2],
 [58, 2]]
```

[26]:
```
def ZeroPadding(l, fillvalue=0):
    return list(itertools.zip_longest(*l, fillvalue=fillvalue))
```

[27]:
```
leng = [len(ind) for ind in indexes]
max(leng)
```

[27]:
```
10
```

[28]:
```
# visualize
test_result = ZeroPadding(indexes)
print(len(test_result))
test_result
```

```
10
```

```
[28]: [(3, 7, 16, 8, 33, 35, 42, 47, 50, 58),
       (4, 8, 4, 31, 34, 36, 2, 7, 51, 2),
       (2, 9, 2, 22, 4, 37, 0, 48, 52, 0),
       (0, 10, 0, 6, 4, 38, 0, 40, 6, 0),
       (0, 4, 0, 2, 4, 7, 0, 45, 2, 0),
       (0, 11, 0, 0, 2, 39, 0, 49, 0, 0),
       (0, 12, 0, 0, 0, 40, 0, 6, 0, 0),
       (0, 13, 0, 0, 0, 41, 0, 2, 0, 0),
       (0, 2, 0, 0, 0, 4, 0, 0, 0, 0),
       (0, 0, 0, 0, 0, 2, 0, 0, 0, 0)]
```

[29]:
```
def binaryMatrix(l, value=0):
    m = []
    for i, seq in enumerate(l):
        m.append([])
        for token in seq:
            if token == PAD_token:
                m[i].append(0)
            else:
                m[i].append(1)
    return m
```

[30]:
```
binary_result = binaryMatrix(test_result)
binary_result
```

```
[30]: [[1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 0, 1, 1, 0],
       [0, 1, 0, 1, 1, 1, 0, 1, 1, 0],
       [0, 1, 0, 1, 1, 1, 0, 1, 1, 0],
       [0, 1, 0, 0, 1, 1, 0, 1, 0, 0],
       [0, 1, 0, 0, 0, 1, 0, 1, 0, 0],
       [0, 1, 0, 0, 0, 1, 0, 1, 0, 0],
       [0, 1, 0, 0, 0, 1, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]]
```

[31]:
```
# returns padded input sequence as well as a tensor of lengths of inputs
def inputVar(l,voc):
    indexes_batch = [indexesFromSentence(voc, sentence) for sentence in l]
    lengths = torch.tensor([len(indexes) for indexes in indexes_batch])
    padList = ZeroPadding(indexes_batch)
    padVar = torch.LongTensor(padList)
    return padVar, lengths
```

[32]:
```
# for target sequence
def outputVar(l,voc):
```

```
    indexes_batch = [indexesFromSentence(voc, sentence) for sentence in l]
    max_target_len = max([len(indexes) for indexes in indexes_batch])
    padList = ZeroPadding(indexes_batch)
    mask = binaryMatrix(padList)
    mask = torch.ByteTensor(mask)
    padVar = torch.LongTensor(padList)
    return padVar, mask, max_target_len
```

[33]:
```
# Returns all items for a given batch of pairs
def batch2TrainData(voc, pair_batch):
    #Sort the questions in descending length
    pair_batch.sort(key=lambda x:len(x[0].split()), reverse=True)
    input_batch, output_batch = [],[]
    for pair in pair_batch:
        input_batch.append(pair[0])
        output_batch.append(pair[1])
    inp, lengths = inputVar(input_batch, voc)

    output, mask, max_target_len = outputVar(output_batch, voc)
    return inp, lengths, output, mask, max_target_len
```

[34]:
```
small_batch_size = 5
batches = batch2TrainData(voc, [random.choice(pairs) for _ in␣
 ↪range(small_batch_size)])
input_variable, lengths, target_variable, mask, max_target_len = batches

print('input variable:', input_variable)
print('lengths:', lengths)
print('target variable:', target_variable)
print('mask:', mask)
print('max target len:', max_target_len)
```

```
input variable: tensor([[ 281,   34,    5,  785,  625],
        [  25,    4,  115,    4,    4],
        [  47,   25,   70,    2,    2],
        [  76,  200, 5123,    0,    0],
        [  94, 5198,    6,    0,    0],
        [  27, 2496,    2,    0,    0],
        [  60,    4,    0,    0,    0],
        [   6,    2,    0,    0,    0],
        [   2,    0,    0,    0,    0]])
lengths: tensor([9, 8, 6, 3, 3])
target variable: tensor([[ 239,  124, 2131,   50,   76],
        [  23,   36,  143,    6,  450],
        [   4,   37,  145,    2,   89],
        [   2,  780,    4,    0,  368],
        [   0,    4,    2,    0,  266],
        [   0,    2,    0,    0, 1034],
```

```
        [   0,    0,    0,    0,   27],
        [   0,    0,    0,    0,  882],
        [   0,    0,    0,    0,    4],
        [   0,    0,    0,    0,    2]])
mask: tensor([[1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1],
        [1, 1, 1, 0, 1],
        [0, 1, 1, 0, 1],
        [0, 1, 0, 0, 1],
        [0, 0, 0, 0, 1],
        [0, 0, 0, 0, 1],
        [0, 0, 0, 0, 1],
        [0, 0, 0, 0, 1]], dtype=torch.uint8)
max target len: 10
```

2 Building the model

2.1 Creating the encoder

[35]:
```python
class EncoderRNN(nn.Module):
    def __init__(self, hidden_size, embedding, n_layers=1, dropout=0):
        super(EncoderRNN, self).__init__()
        self.n_layers = n_layers
        self.hidden_size = hidden_size
        self.embedding = embedding
        # initialize GRU; the input_size and hidden_size params are both to␣
 ↪hidden_size
        # becuz our input size is a word embedding with num of features ==␣
 ↪hidden size
        self.gru = nn.GRU(hidden_size, hidden_size, n_layers, dropout=(0 if␣
 ↪n_layers == 1 else dropout), bidirectional=True)

    def forward(self, input_seq, input_lengths, hidden=None):
        # input_seq : batch of input sentences; shape(max_len, batch_size)
        # input_lengths: list of sentence lengths correspoding to each sentence␣
 ↪in the batch
        # hidden state of shape: (n_layers x num_direction, batch_size,␣
 ↪hidden_size)

        #Convert word indexes to embeddings
        embedded = self.embedding(input_seq)

        # Pack padded batch of sequences for RNN
        packed = torch.nn.utils.rnn.pack_padded_sequence(embedded, input_lengths)

        # Forward pass through GRU
```

```
        outputs, hidden = self.gru(packed, hidden)

        #unpack the padding
        outputs, _ = torch.nn.utils.rnn.pad_packed_sequence(outputs)

        #sum bidirectional GRU outputs
        outputs = outputs[:,:,:self.hidden_size] + outputs[:,:,self.hidden_size:]

        # Return output and final hidden state
        return outputs, hidden
```

2.2 Attention Mechanism

[36]:

```
# Luong attention layer
class Attn(torch.nn.Module):
    def __init__(self, method, hidden_size):
        super(Attn, self).__init__()
        self.method = method
        self.hidden_size = hidden_size

    def dot_score(self, hidden, encoder_output):
        # Element wise multiply the current target decoder state with  encoder␣
↪output and sum them
        return torch.sum(hidden * encoder_output, dim=2)

    def forward(self, hidden, encoder_outputs):
        '''
            Hidden = shape(1,batch_size, hidden_size)
            encoder_outputs = shape(max_length, batch_size)

            returns sum(hidden*encoder_output)
        '''
        attn_energies = self.dot_score(hidden, encoder_outputs)␣
↪#shape=(max_length. batch_size)

        # Transpose dim
        attn_energies = attn_energies.t()

        #return softmax of attn energies
        return F.softmax(attn_energies, dim=1).unsqueeze(1)␣
↪#shape=(batch_size,1,ma
```

2.3 Creating the decoder

[37]:
```
class LuongAttnDecoderRNN(nn.Module):
    def __init__(self, attn_model, embedding, hidden_size, output_size,␣
↪n_layers=1,  dropout=0.1):
        super(LuongAttnDecoderRNN, self).__init__()
        self.attn_model = attn_model
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.n_layers = n_layers
        self.dropout = dropout

        # Defining layers
        self.embedding = embedding
        self.embedding_dropout = nn.Dropout(dropout)
        self.gru = nn.GRU(hidden_size, hidden_size, n_layers, dropout=(0 if␣
↪n_layers==1 else dropout))
        self.concat = nn.Linear(hidden_size*2, hidden_size)
        self.out = nn.Linear(hidden_size, output_size)

        self.attn = Attn(attn_model, hidden_size)

    def forward(self, input_step, last_hidden, encoder_outputs):
        '''
            input_step: one time step of input seq batch; shape=(1, batch_size)
            last_hidden: final hidden layer of GRU;␣
↪shape=(n_layers*num_directions, batch_size, hidden_size)
            encoder_outputs: encoder model's output; shape=(max_length,␣
↪batch_size, hidden_size)
            we run this one step(word) at a time

            Output:
            softmax normalized tensor giving probabilities for each word being␣
↪correct next word in the decoded sequence
            shape=(batch_size, voc.num_words)
            hidden: final hidden state of GRU; shape=(n_layers x num_direction,␣
↪batch_size, hidden_size)
        '''
        # Get embedding of current input word
        embedded = self.embedding(input_step)
        embedded = self.embedding_dropout(embedded)

        # Forward pass
        rnn_output, hidden = self.gru(embedded, last_hidden)

        # Calculate attention weights from current GRU output
        attn_weights = self.attn(rnn_output, encoder_outputs)
```

```
        # Multiply attention weights to encoder output to get the weighted sum␣
↪of the context vector
        # (batch_size, 1, max_length) bmm with (batch_size,max_length,hidden) =␣
↪(batch_size,1,hidden)
        context = attn_weights.bmm(encoder_outputs.transpose(0,1))
        # Concatenate weighted context vector and GRU output
        rnn_output = rnn_output.squeeze(0)
        context = context.squeeze(1)

        concat_input = torch.cat((rnn_output, context), 1)
        concat_output = torch.tanh(self.concat(concat_input))

        # Predict next word using Luong eq. 6
        output = self.out(concat_output)
        output = F.softmax(output, dim=1)

        # Return output and final hidden state
        return output, hidden
```

2.4 Creating the loss function

[40]:
```
def maskNLLLoss(decoder_out, target, mask):
    nTotal = mask.sum()
    target = target.view(-1,1)
    # Decoder out shape:(batch_size, vocab_size), target_size=(batch_size, 1)
    gather_tensor = torch.gather(decoder_out, 1, target)
    # Calculate the negative log likelihood loss
    crossEntropy = -torch.log(gather_tensor)
    #select non-zero elements
    loss = crossEntropy.masked_select(mask.bool())
    # Calculate the mean of the loss
    loss = loss.mean()
    loss = loss.to(device)
    return loss, nTotal.item()
```

[41]:
```
# Visualizing what's happening in one iteration
small_batch_size = 5
batches = batch2TrainData(voc, [random.choice(pairs) for _ in␣
↪range(small_batch_size)])
input_variable, lengths, target_variable, mask, max_target_len = batches

print('input variable shape:', input_variable.shape)
print('length shapes:', lengths.shape)
print('target variable shape:', target_variable.shape)
print('mask shape:', mask.shape)
print('max target len:', max_target_len)
```

```python
# Define the parameters
hidden_size =500
encoder_n_layers = 2
decoder_n_layers = 2
dropout = 0.1
attn_model = 'dot'
embedding = nn.Embedding(voc.num_words, hidden_size)

# Define the encoder and decoder
encoder = EncoderRNN(hidden_size, embedding, encoder_n_layers, dropout)
decoder = LuongAttnDecoderRNN(attn_model, embedding, hidden_size, voc.num_words, decoder_n_layers, dropout)
encoder = encoder.to(device)
decoder = decoder.to(device)

# Ensure dropout layers are in train mode
encoder.train()
decoder.train()

# Initialize optimizers
encoder_optimizer = optim.Adam(encoder.parameters(), lr=0.0001)
decoder_optimizer = optim.Adam(decoder.parameters(), lr=0.0001)
encoder_optimizer.zero_grad()
decoder_optimizer.zero_grad()


input_variable = input_variable.to(device)
lengths = lengths.to(device)
target_variable = target_variable.to(device)
mask = mask.to(device)

loss = 0
print_losses = []
n_totals = 0

encoder_outputs, encoder_hidden = encoder(input_variable, lengths)
print('Encoder outputs shape: ', encoder_outputs.shape)
print('Last Encoder Hidden Shape', encoder_hidden.shape)

decoder_input = torch.LongTensor([[SOS_token for _ in range(small_batch_size)]])
decoder_input = decoder_input.to(device)
print('Initial Decoder input shape:', decoder_input.shape)
print(decoder_input)

# Set the initial decoder hidden state to the encoder's final hidden state
decoder_hidden = encoder_hidden[:decoder.n_layers]
```

```
print('Initial Decoder hidden state shape:', decoder_hidden.shape)
print('\n')
print('.'*50)
print('Now Lets look at whats happening in every timestep of a GRU')
print('.'*50)
print('\n')

# Assume we are using Teacher Forcing
for t in range(max_target_len):
    decoder_output, decoder_hidden = decoder(decoder_input, decoder_hidden,␣
 ↪encoder_outputs)
    print('Decoder output shape:', decoder_output.shape)
    print('Decoder Hidden shape:', decoder_hidden.shape)

    # Teacher forcing: next input is current target
    decoder_input = target_variable[t].view(1, -1)
    print('The target variable at current timestep before reshaping:',␣
 ↪target_variable[t])
    print('The shape of target variable at current timestep before reshaping:',␣
 ↪target_variable[t].shape)
    print('Decoder input shae:', decoder_input.shape)

    # Calculate and accumulate loss
    print('The mask at the current timestep:', mask[t])
    print('The shape of mask:', mask[t].shape)

    mask_loss, nTotal = maskNLLLoss(decoder_output, target_variable[t], mask[t])
    print('Mask loss:', mask_loss)
    print('Total;', nTotal)

    loss += mask_loss
    print_losses.append(mask_loss.item()*nTotal)
    print(print_losses)
    n_totals +=nTotal
    print(n_totals)

    encoder_optimizer.step()
    decoder_optimizer.step()
    returned_loss = sum(print_losses) / n_totals
    print('Returned Loss:', returned_loss)
    print('\n')
    print('.'*30, 'Done one timestep', '.'*30)
    print('\n')
```

```
input variable shape: torch.Size([10, 5])
length shapes: torch.Size([5])
target variable shape: torch.Size([7, 5])
```

```
mask shape: torch.Size([7, 5])
max target len: 7
Encoder outputs shape:  torch.Size([10, 5, 500])
Last Encoder Hidden Shape torch.Size([4, 5, 500])
Initial Decoder input shape: torch.Size([1, 5])
tensor([[1, 1, 1, 1, 1]])
Initial Decoder hidden state shape: torch.Size([2, 5, 500])


...
Now Lets look at whats happening in every timestep of a GRU
...


Decoder output shape: torch.Size([5, 7826])
Decoder Hidden shape: torch.Size([2, 5, 500])
The target variable at current timestep before reshaping: tensor([  76,   25,
354, 1014,   33])
The shape of target variable at current timestep before reshaping:
torch.Size([5])
Decoder input shae: torch.Size([1, 5])
The mask at the current timestep: tensor([1, 1, 1, 1, 1], dtype=torch.uint8)
Teh shape of mask: torch.Size([5])
Mask loss: tensor(8.9922, grad_fn=<MeanBackward0>)
Total; 5
[44.96102809906006]
5
Returned Loss: 8.992205619812012


... Done one timestep ...


Decoder output shape: torch.Size([5, 7826])
Decoder Hidden shape: torch.Size([2, 5, 500])
The target variable at current timestep before reshaping: tensor([  37,  410,
2254, 2527,   77])
The shape of target variable at current timestep before reshaping:
torch.Size([5])
Decoder input shae: torch.Size([1, 5])
The mask at the current timestep: tensor([1, 1, 1, 1, 1], dtype=torch.uint8)
Teh shape of mask: torch.Size([5])
Mask loss: tensor(8.9660, grad_fn=<MeanBackward0>)
Total; 5
[44.96102809906006, 44.829864501953125]
10
Returned Loss: 8.979089260101318
```

```
... Done one timestep ...


Decoder output shape: torch.Size([5, 7826])
Decoder Hidden shape: torch.Size([2, 5, 500])
The target variable at current timestep before reshaping: tensor([ 869, 1841,
1237, 1014,   92])
The shape of target variable at current timestep before reshaping:
torch.Size([5])
Decoder input shae: torch.Size([1, 5])
The mask at the current timestep: tensor([1, 1, 1, 1, 1], dtype=torch.uint8)
Teh shape of mask: torch.Size([5])
Mask loss: tensor(8.9630, grad_fn=<MeanBackward0>)
Total; 5
[44.96102809906006, 44.829864501953125, 44.81476306915283]
15
Returned Loss: 8.973710378011068


... Done one timestep ...


Decoder output shape: torch.Size([5, 7826])
Decoder Hidden shape: torch.Size([2, 5, 500])
The target variable at current timestep before reshaping: tensor([ 4,  4,  6,
68,  7])
The shape of target variable at current timestep before reshaping:
torch.Size([5])
Decoder input shae: torch.Size([1, 5])
The mask at the current timestep: tensor([1, 1, 1, 1, 1], dtype=torch.uint8)
Teh shape of mask: torch.Size([5])
Mask loss: tensor(8.9780, grad_fn=<MeanBackward0>)
Total; 5
[44.96102809906006, 44.829864501953125, 44.81476306915283, 44.890241622924805]
20
Returned Loss: 8.974794864654541


... Done one timestep ...


Decoder output shape: torch.Size([5, 7826])
Decoder Hidden shape: torch.Size([2, 5, 500])
The target variable at current timestep before reshaping: tensor([ 2,  2,  2,
45, 35])
The shape of target variable at current timestep before reshaping:
torch.Size([5])
```

```
Decoder input shae: torch.Size([1, 5])
The mask at the current timestep: tensor([1, 1, 1, 1, 1], dtype=torch.uint8)
Teh shape of mask: torch.Size([5])
Mask loss: tensor(8.9904, grad_fn=<MeanBackward0>)
Total; 5
[44.96102809906006, 44.829864501953125, 44.81476306915283, 44.890241622924805,
44.95201587677002]
25
Returned Loss: 8.977916526794434


... Done one timestep ...


Decoder output shape: torch.Size([5, 7826])
Decoder Hidden shape: torch.Size([2, 5, 500])
The target variable at current timestep before reshaping: tensor([0, 0, 0, 6,
6])
The shape of target variable at current timestep before reshaping:
torch.Size([5])
Decoder input shae: torch.Size([1, 5])
The mask at the current timestep: tensor([0, 0, 0, 1, 1], dtype=torch.uint8)
Teh shape of mask: torch.Size([5])
Mask loss: tensor(8.9228, grad_fn=<MeanBackward0>)
Total; 2
[44.96102809906006, 44.829864501953125, 44.81476306915283, 44.890241622924805,
44.95201587677002, 17.8456974029541]
27
Returned Loss: 8.973837428622776


... Done one timestep ...


Decoder output shape: torch.Size([5, 7826])
Decoder Hidden shape: torch.Size([2, 5, 500])
The target variable at current timestep before reshaping: tensor([0, 0, 0, 2,
2])
The shape of target variable at current timestep before reshaping:
torch.Size([5])
Decoder input shae: torch.Size([1, 5])
The mask at the current timestep: tensor([0, 0, 0, 1, 1], dtype=torch.uint8)
Teh shape of mask: torch.Size([5])
Mask loss: tensor(8.9447, grad_fn=<MeanBackward0>)
Total; 2
[44.96102809906006, 44.829864501953125, 44.81476306915283, 44.890241622924805,
44.95201587677002, 17.8456974029541, 17.88949966430664]
29
```

```
Returned Loss: 8.97183138748695
```

```
... Done one timestep ...
```

[42]:
```
def train(input_variable, lengths, target_variable,
          mask, max_target_len, encoder,
          decoder, embedding, encoder_optimizer,
          decoder_optimizer, batch_size,
          clip, max_length=MAX_LENGTH):

    # Zero gradients
    encoder_optimizer.zero_grad()
    decoder_optimizer.zero_grad()

    # Set device options
    input_variable = input_variable.to(device)
    lengths = lengths.to(device)
    target_variable = target_variable.to(device)
    mask = mask.to(device)

    # initialize variable
    loss = 0
    print_losses = []
    n_totals = 0

    # Forward pass through encoder
    encoder_outputs, encoder_hidden = encoder(input_variable, lengths)

    # Create initial decoder input (start with SOS token)
    decoder_input = torch.LongTensor([[SOS_token for _ in range(batch_size)]])
    decoder_input = decoder_input.to(device)

    # Set initial decoder hidden state to the encoder's final hidden state
    decoder_hidden = encoder_hidden[:decoder.n_layers]

    # Determine if we are using teacher forcing
    use_teacher_forcing = True if random.random() < teacher_forcing_ratio else␣
↪False

    # Forward pass
    if use_teacher_forcing:
        for t in range(max_target_len):
            decoder_output, decoder_hidden = decoder(decoder_input,␣
↪decoder_hidden, encoder_outputs)
```

```
            # teacher forcing
            decoder_input = target_variable[t].view(1,-1)
            #calculate and accumulate loss
            mask_loss, nTotal = maskNLLLoss(decoder_output, target_variable[t],␣
↪mask[t])

            loss += mask_loss
            print_losses.append(mask_loss.item()*nTotal)
            n_totals +=nTotal
    else:
        for t in range(max_target_len):
            decoder_output, decoder_hidden = decoder(decoder_input,␣
↪decoder_hidden, encoder_outputs)
            # no teacher forcing
            _, topi = decoder_output.topk(1)
            decoder_input = torch.LongTensor([[topi[i][0] for i in␣
↪range(batch_size)]])
            decoder_input = decoder_input.to(device)
            #calculate and accumulate loss
            mask_loss, nTotal = maskNLLLoss(decoder_output, tatget_variable[t],␣
↪mask[t])

            loss += mask_loss
            print_losses.append(mask_loss.item()*nTotal)
            n_totals +=nTotal

    # Backpropagation
    loss.backward()

    # Gradient Clipping
    _ = nn.utils.clip_grad_norm_(encoder.parameters(), clip)
    _ = nn.utils.clip_grad_norm_(decoder.parameters(), clip)

    # Adjust model weights
    encoder_optimizer.step()
    decoder_optimizer.step()

    return sum(print_losses) / n_totals
```

[44]:
```
def trainIters(model_name, voc, pairs,
               encoder, decoder,
               encoder_optimizer, decoder_optimizer,
               embedding, encoder_n_layers,
               decoder_n_layers, save_dir,
               n_iteration, batch_size, print_every,
               save_every, clip, corpus_name, loadFilename):
```

```
    # Load batches for each iteration
    training_batches = [batch2TrainData(voc, [random.choice(pairs) for _ in␣
↪range(batch_size)])
                        for _ in range(n_iteration)]

    # Initializations
    print('Initializing ...')
    start_iteration = 1
    print_loss = 0
    if loadFilename:
        start_iteration = checkpoint['iteration'] + 1

    # Training loop
    print("Training...")
    for iteration in range(start_iteration, n_iteration + 1):
        training_batch = training_batches[iteration - 1]
        # Extract fields from batch
        input_variable, lengths, target_variable, mask, max_target_len =␣
↪training_batch

        # Run a training iteration with batch
        loss = train(input_variable, lengths, target_variable, mask,␣
↪max_target_len, encoder,
                     decoder, embedding, encoder_optimizer, decoder_optimizer,␣
↪batch_size, clip)
        print_loss += loss

        # Print progress
        if iteration % print_every == 0:
            print_loss_avg = print_loss / print_every
            print("Iteration: {}; Percent complete: {:.1f}%; Average loss: {:.
↪4f}".format(iteration, iteration / n_iteration * 100, print_loss_avg))
            print_loss = 0

        # Save checkpoint
        if (iteration % save_every == 0):
            directory = os.path.join(save_dir, model_name, corpus_name,␣
↪'{}-{}_{}'.format(encoder_n_layers, decoder_n_layers, hidden_size))
            if not os.path.exists(directory):
                os.makedirs(directory)
            torch.save({
                'iteration': iteration,
                'en': encoder.state_dict(),
                'de': decoder.state_dict(),
                'en_opt': encoder_optimizer.state_dict(),
                'de_opt': decoder_optimizer.state_dict(),
```

```
                'loss': loss,
                'voc_dict': voc.__dict__,
                'embedding': embedding.state_dict()
            }, os.path.join(directory, '{}_{}.tar'.format(iteration,␣
↪'checkpoint')))
```

2.5 Greedy Decoding

[45]:
```
class GreedySearchDecoder(nn.Module):
    def __init__(self, encoder, decoder):
        super(GreedySearchDecoder, self).__init__()
        self.encoder = encoder
        self.decoder = decoder

    def forward(self, input_seq, input_length, max_length):
        # Forward input through encoder model
        encoder_outputs, encoder_hidden = self.encoder(input_seq, input_length)
        # Prepare encoder's final hidden layer to be first hidden input to the␣
↪decoder
        decoder_hidden = encoder_hidden[:decoder.n_layers]
        # Initialize decoder input with SOS_token
        decoder_input = torch.ones(1, 1, device=device, dtype=torch.long) *␣
↪SOS_token
        # Initialize tensors to append decoded words to
        all_tokens = torch.zeros([0], device=device, dtype=torch.long)
        all_scores = torch.zeros([0], device=device)
        # Iteratively decode one word token at a time
        for _ in range(max_length):
            # Forward pass through decoder
            decoder_output, decoder_hidden = self.decoder(decoder_input,␣
↪decoder_hidden, encoder_outputs)
            # Obtain most likely word token and its softmax score
            decoder_scores, decoder_input = torch.max(decoder_output, dim=1)
            # Record token and score
            all_tokens = torch.cat((all_tokens, decoder_input), dim=0)
            all_scores = torch.cat((all_scores, decoder_scores), dim=0)
            # Prepare current token to be next decoder input (add a dimension)
            decoder_input = torch.unsqueeze(decoder_input, 0)
        # Return collections of word tokens and scores
        return all_tokens, all_scores
```

[64]:
```
def evaluate(encoder, decoder, searcher, voc, sentence, max_length=MAX_LENGTH):
    ### Format input sentence as a batch
    # words -> indexes
    indexes_batch = [indexesFromSentence(voc, sentence)]
    # Create lengths tensor
    lengths = torch.tensor([len(indexes) for indexes in indexes_batch])
```

```
    # Transpose dimensions of batch to match models' expectations
    input_batch = torch.LongTensor(indexes_batch).transpose(0, 1)
    # Use appropriate device
    input_batch = input_batch.to(device)
    lengths = lengths.to(device)
    # Decode sentence with searcher
    tokens, scores = searcher(input_batch, lengths, max_length)
    # indexes -> words
    decoded_words = [voc.index2word[token.item()] for token in tokens]
    return decoded_words


def evaluateInput(encoder, decoder, searcher, voc):
    input_sentence = ''
    while(1):
        try:
            # Get input sentence
            input_sentence = input('> ')
            # Check if it is quit case
            if input_sentence == 'q' or input_sentence == 'quit': break
            # Normalize sentence
            input_sentence = normalizeString(input_sentence)
            # Evaluate sentence
            output_words = evaluate(encoder, decoder, searcher, voc,␣
↪input_sentence)
            # Format and print response sentence
            output_words[:] = [x for x in output_words if not (x == 'EOS' or x␣
↪== 'PAD')]
            print('Bot:', ' '.join(output_words))

        except KeyError:
            print("Error: Unknown word.")
```

[47]:
```
# Configure models
model_name = 'chatbot_model'
attn_model = 'dot'
hidden_size = 500
encoder_n_layers = 2
decoder_n_layers = 2
dropout = 0.1
batch_size = 64

# Set checkpoint to load (From Scratch = None)
loadFilename = None
checkpoint_iter = 4000

# Load model if a loadFilename is provided
```

```
if loadFilename:
    checkpoint = torch.load(loadFilename)
    encoder_sd = checkpoint['en']
    decoder_sd = checkpoint['de']
    encoder_optimizer_sd = checkpoint['en_opt']
    decoder_optimizer_sd = checkpoint['de_opt']
    embedding_sd = checkpoint['embedding']
    voc.__dict__ = checkpoint['voc_dict']


print('Building encoder and decoder ...')
# Initialize word embeddings
embedding = nn.Embedding(voc.num_words, hidden_size)
if loadFilename:
    embedding.load_state_dict(embedding_sd)
# Initialize encoder & decoder models
encoder = EncoderRNN(hidden_size, embedding, encoder_n_layers, dropout)
decoder = LuongAttnDecoderRNN(attn_model, embedding, hidden_size, voc.num_words,␣
 ↪decoder_n_layers, dropout)
if loadFilename:
    encoder.load_state_dict(encoder_sd)
    decoder.load_state_dict(decoder_sd)
# Use appropriate device
encoder = encoder.to(device)
decoder = decoder.to(device)
print('Models built and ready to go!')
```

```
Building encoder and decoder ...
Models built and ready to go!
```

[48]:

```
```

```
Building optimizers ...
Starting Training!
Initializing ...
Training...
Iteration: 50; Percent complete: 1.0%; Average loss: 5.0197
Iteration: 100; Percent complete: 2.0%; Average loss: 3.9113
Iteration: 150; Percent complete: 3.0%; Average loss: 3.7072
Iteration: 200; Percent complete: 4.0%; Average loss: 3.5221
Iteration: 250; Percent complete: 5.0%; Average loss: 3.4446
Iteration: 300; Percent complete: 6.0%; Average loss: 3.3437
Iteration: 350; Percent complete: 7.0%; Average loss: 3.3015
Iteration: 400; Percent complete: 8.0%; Average loss: 3.2849
Iteration: 450; Percent complete: 9.0%; Average loss: 3.2486
Iteration: 500; Percent complete: 10.0%; Average loss: 3.2147
Iteration: 550; Percent complete: 11.0%; Average loss: 3.1515
Iteration: 600; Percent complete: 12.0%; Average loss: 3.1369
```

```
Iteration: 650; Percent complete: 13.0%; Average loss: 3.1116
Iteration: 700; Percent complete: 14.0%; Average loss: 3.1012
Iteration: 750; Percent complete: 15.0%; Average loss: 3.0269
Iteration: 800; Percent complete: 16.0%; Average loss: 3.0251
Iteration: 850; Percent complete: 17.0%; Average loss: 3.0300
Iteration: 900; Percent complete: 18.0%; Average loss: 2.9804
Iteration: 950; Percent complete: 19.0%; Average loss: 2.9830
Iteration: 1000; Percent complete: 20.0%; Average loss: 2.9734
Iteration: 1050; Percent complete: 21.0%; Average loss: 2.9421
Iteration: 1100; Percent complete: 22.0%; Average loss: 2.9624
Iteration: 1150; Percent complete: 23.0%; Average loss: 2.8726
Iteration: 1200; Percent complete: 24.0%; Average loss: 2.8822
Iteration: 1250; Percent complete: 25.0%; Average loss: 2.8383
Iteration: 1300; Percent complete: 26.0%; Average loss: 2.8545
Iteration: 1350; Percent complete: 27.0%; Average loss: 2.8109
Iteration: 1400; Percent complete: 28.0%; Average loss: 2.8593
Iteration: 1450; Percent complete: 29.0%; Average loss: 2.8180
Iteration: 1500; Percent complete: 30.0%; Average loss: 2.7802
Iteration: 1550; Percent complete: 31.0%; Average loss: 2.7850
Iteration: 1600; Percent complete: 32.0%; Average loss: 2.7598
Iteration: 1650; Percent complete: 33.0%; Average loss: 2.7450
Iteration: 1700; Percent complete: 34.0%; Average loss: 2.7549
Iteration: 1750; Percent complete: 35.0%; Average loss: 2.7213
Iteration: 1800; Percent complete: 36.0%; Average loss: 2.7292
Iteration: 1850; Percent complete: 37.0%; Average loss: 2.7031
Iteration: 1900; Percent complete: 38.0%; Average loss: 2.6793
Iteration: 1950; Percent complete: 39.0%; Average loss: 2.6790
Iteration: 2000; Percent complete: 40.0%; Average loss: 2.6571
Iteration: 2050; Percent complete: 41.0%; Average loss: 2.6550
Iteration: 2100; Percent complete: 42.0%; Average loss: 2.6487
Iteration: 2150; Percent complete: 43.0%; Average loss: 2.6387
Iteration: 2200; Percent complete: 44.0%; Average loss: 2.6049
Iteration: 2250; Percent complete: 45.0%; Average loss: 2.6112
Iteration: 2300; Percent complete: 46.0%; Average loss: 2.6003
Iteration: 2350; Percent complete: 47.0%; Average loss: 2.5703
Iteration: 2400; Percent complete: 48.0%; Average loss: 2.5551
Iteration: 2450; Percent complete: 49.0%; Average loss: 2.4961
Iteration: 2500; Percent complete: 50.0%; Average loss: 2.5453
Iteration: 2550; Percent complete: 51.0%; Average loss: 2.5520
Iteration: 2600; Percent complete: 52.0%; Average loss: 2.4981
Iteration: 2650; Percent complete: 53.0%; Average loss: 2.4975
Iteration: 2700; Percent complete: 54.0%; Average loss: 2.4665
Iteration: 2750; Percent complete: 55.0%; Average loss: 2.4896
Iteration: 2800; Percent complete: 56.0%; Average loss: 2.4450
Iteration: 2850; Percent complete: 57.0%; Average loss: 2.4373
Iteration: 2900; Percent complete: 58.0%; Average loss: 2.4558
Iteration: 2950; Percent complete: 59.0%; Average loss: 2.4045
Iteration: 3000; Percent complete: 60.0%; Average loss: 2.4261
```

```
Iteration: 3050; Percent complete: 61.0%; Average loss: 2.3771
Iteration: 3100; Percent complete: 62.0%; Average loss: 2.3791
Iteration: 3150; Percent complete: 63.0%; Average loss: 2.4264
Iteration: 3200; Percent complete: 64.0%; Average loss: 2.3528
Iteration: 3250; Percent complete: 65.0%; Average loss: 2.3543
Iteration: 3300; Percent complete: 66.0%; Average loss: 2.3743
Iteration: 3350; Percent complete: 67.0%; Average loss: 2.3029
Iteration: 3400; Percent complete: 68.0%; Average loss: 2.3144
Iteration: 3450; Percent complete: 69.0%; Average loss: 2.2998
Iteration: 3500; Percent complete: 70.0%; Average loss: 2.2944
Iteration: 3550; Percent complete: 71.0%; Average loss: 2.2720
Iteration: 3600; Percent complete: 72.0%; Average loss: 2.2753
Iteration: 3650; Percent complete: 73.0%; Average loss: 2.2422
Iteration: 3700; Percent complete: 74.0%; Average loss: 2.2239
Iteration: 3750; Percent complete: 75.0%; Average loss: 2.1933
Iteration: 3800; Percent complete: 76.0%; Average loss: 2.2238
Iteration: 3850; Percent complete: 77.0%; Average loss: 2.1993
Iteration: 3900; Percent complete: 78.0%; Average loss: 2.1884
Iteration: 3950; Percent complete: 79.0%; Average loss: 2.1901
Iteration: 4000; Percent complete: 80.0%; Average loss: 2.1714
Iteration: 4050; Percent complete: 81.0%; Average loss: 2.1626
Iteration: 4100; Percent complete: 82.0%; Average loss: 2.1400
Iteration: 4150; Percent complete: 83.0%; Average loss: 2.1328
Iteration: 4200; Percent complete: 84.0%; Average loss: 2.1268
Iteration: 4250; Percent complete: 85.0%; Average loss: 2.1169
Iteration: 4300; Percent complete: 86.0%; Average loss: 2.0969
Iteration: 4350; Percent complete: 87.0%; Average loss: 2.0486
Iteration: 4400; Percent complete: 88.0%; Average loss: 2.0807
Iteration: 4450; Percent complete: 89.0%; Average loss: 2.0203
Iteration: 4500; Percent complete: 90.0%; Average loss: 2.0368
Iteration: 4550; Percent complete: 91.0%; Average loss: 2.0174
Iteration: 4600; Percent complete: 92.0%; Average loss: 2.0049
Iteration: 4650; Percent complete: 93.0%; Average loss: 2.0047
Iteration: 4700; Percent complete: 94.0%; Average loss: 1.9829
Iteration: 4750; Percent complete: 95.0%; Average loss: 1.9824
Iteration: 4800; Percent complete: 96.0%; Average loss: 1.9540
Iteration: 4850; Percent complete: 97.0%; Average loss: 1.9392
Iteration: 4900; Percent complete: 98.0%; Average loss: 1.9527
Iteration: 4950; Percent complete: 99.0%; Average loss: 1.8738
Iteration: 5000; Percent complete: 100.0%; Average loss: 1.8832
```

[49]:

```
# Set dropout layers to eval mode
encoder.eval()
decoder.eval()

# Initialize search module
searcher = GreedySearchDecoder(encoder, decoder)
```

[63]:
```
evaluateInput(encoder, decoder, searcher, voc)
```

```
> hello
Bot: hello .
> how are you
Bot: fine i m fine .
> do you have a name?
Bot: yes .
> great, what is your name?
Bot: my name is robin camelot .
> nice to meet you robin
Bot: the hell i am .
> spicy
Error: Encountered unknown word.
> q
```