Lab 10: Long Short Term Memory (LSTM) Models

name = 'Todsavad Tangtortan' id = 123012

1. Find another dataset of sentence pairs in a different domain and see

if you can preprocess the data and train a chatbot model on it using the same code we developed today. Report your results.

In [2]:

```
import torch, torchdata, torchtext
from torch import nn
import torch.nn.functional as F
import pandas as pd
import numpy as np
import random, math, time

device = torch.device('cuda:2' if torch.cuda.is_available() else 'cpu')
print(device)

#make our work comparable if restarted the kernel
SEED = 1234
torch.manual_seed(SEED)
torch.backends.cudnn.deterministic = True
```

```
cpu
```

1.1 Load Dataset

In [3]:

```
from datasets import load_dataset
dataset = load_dataset('multi_woz_v22')
dataset
```

```
No config specified, defaulting to: multi_woz_v22/v2.2_active_only
Found cached dataset multi_woz_v22 (C:/Users/Guntsv/.cache/huggingface/dat
asets/multi_woz_v22/v2.2_active_only/2.2.0/6719c8b21478299411a0c6fdb7137c3
ebab2e6425129af831687fb7851c69eb5)
```

Out[3]:

```
DatasetDict({
    train: Dataset({
        features: ['dialogue_id', 'services', 'turns'],
        num_rows: 8437
    })
    validation: Dataset({
        features: ['dialogue_id', 'services', 'turns'],
        num_rows: 1000
    })
    test: Dataset({
        features: ['dialogue_id', 'services', 'turns'],
        num_rows: 1000
    })
})
```

In [4]:

```
## Load the train data and the validation data
train_data = dataset['train']
valid_data = dataset['validation']
test_data  = dataset['test']
```

In [5]:

```
df_valid = pd.DataFrame(valid_data)
df_test = pd.DataFrame(test_data)
df_valid.shape, df_test.shape
```

Out[5]:

```
((1000, 3), (1000, 3))
```

In [6]:

```
df_test.drop(columns=['dialogue_id','services'],inplace=True)
df_valid.drop(columns=['dialogue_id','services'],inplace=True)
df_valid.shape, df_test.shape
```

Out[6]:

```
((1000, 1), (1000, 1))
```

In [7]:

```
all_question,all_answer = list(), list()
for input_ids in range(len(df_test['turns'])):
    for index, sentence in enumerate(df_test['turns'][input_ids]['utterance']):
        if index % 2 == 0:
            all_question.append(sentence)
        else:
            all_answer.append(sentence)
df_dialogue = pd.DataFrame([all_question,all_answer]).T
df_dialogue.rename(columns={0:'question', 1:'answer'},inplace=True)
```

In [8]:

```
df_dialogue.shape
```

Out[8]:

```
(7372, 2)
```

In [9]:

```
#save DataFrame inorder to not reload again
import pickle
# with open('df_dialogue.pickle', 'wb') as f:
#     pickle.dump(df_dialogue, f, protocol=pickle.HIGHEST_PROTOCOL)
with open('df_dialogue.pickle', 'rb') as f:
    df_dialogue = pickle.load(f)
```

In [10]:

```
import numpy as np

np.savetxt('dialogueDataset.txt', df_dialogue.values, fmt='%s', delimiter='\t')
```

1.2 LSTM Model

In [11]:

```python
import math
import torch
from torch import nn

class NaiveCustomLSTM(nn.Module):

    def __init__(self, input_sz: int, hidden_sz: int):
        super().__init__()
        self.input_size = input_sz
        self.hidden_size = hidden_sz

        # Parameters for computing i_t
        self.U_i = nn.Parameter(torch.Tensor(input_sz, hidden_sz))
        self.V_i = nn.Parameter(torch.Tensor(hidden_sz, hidden_sz))
        self.b_i = nn.Parameter(torch.Tensor(hidden_sz))

        # Parameters for computing f_t
        self.U_f = nn.Parameter(torch.Tensor(input_sz, hidden_sz))
        self.V_f = nn.Parameter(torch.Tensor(hidden_sz, hidden_sz))
        self.b_f = nn.Parameter(torch.Tensor(hidden_sz))

        # Parameters for computing c_t
        self.U_c = nn.Parameter(torch.Tensor(input_sz, hidden_sz))
        self.V_c = nn.Parameter(torch.Tensor(hidden_sz, hidden_sz))
        self.b_c = nn.Parameter(torch.Tensor(hidden_sz))

        # Parameters for computing o_t
        self.U_o = nn.Parameter(torch.Tensor(input_sz, hidden_sz))
        self.V_o = nn.Parameter(torch.Tensor(hidden_sz, hidden_sz))
        self.b_o = nn.Parameter(torch.Tensor(hidden_sz))

        self.init_weights()

    def init_weights(self):
        stdv = 1.0 / math.sqrt(self.hidden_size)
        for weight in self.parameters():
            weight.data.normal_(mean=0, std=stdv)

    def forward(self, x, init_states=None):
        """
        forward: Run input x through the cell. Assumes x.shape is (batch_size, sequence
_length, input_size)
        """
        bs, seq_sz, _ = x.size()
        hidden_seq = []

        if init_states is None:
            h_t, c_t = (
                torch.zeros(bs, self.hidden_size).to(x.device),
                torch.zeros(bs, self.hidden_size).to(x.device),
            )
        else:
            h_t, c_t = init_states

        for t in range(seq_sz):
            x_t = x[:, t, :]

            i_t = torch.sigmoid(x_t @ self.U_i + h_t @ self.V_i + self.b_i)
            f_t = torch.sigmoid(x_t @ self.U_f + h_t @ self.V_f + self.b_f)
```

```python
            g_t = torch.tanh(x_t @ self.U_c + h_t @ self.V_c + self.b_c)
            o_t = torch.sigmoid(x_t @ self.U_o + h_t @ self.V_o + self.b_o)
            c_t = f_t * c_t + i_t * g_t
            h_t = o_t * torch.tanh(c_t)

            hidden_seq.append(h_t.unsqueeze(0))

        # Reshape hidden_seq tensor to (batch size, sequence length, hidden_size)
        hidden_seq = torch.cat(hidden_seq, dim=0)
        hidden_seq = hidden_seq.transpose(0, 1).contiguous()

        return hidden_seq, (h_t, c_t)
```

In [12]:

```python
import torch
from torch.jit import script, trace
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
from io import open
import itertools
import math
import pickle
from torch.autograd import Variable

CUDA = torch.cuda.is_available()
device = torch.device("cuda:2" if CUDA else "cpu")
device
```

Out[12]:

```
device(type='cpu')
```

In [13]:

```
# Reserved word tokens

PAD_token = 0  # Used for padding short sentences
SOS_token = 1  # Start-of-sentence token
EOS_token = 2  # End-of-sentence token
UNK_token = 3  # Unknown token
class Voc:

    def __init__(self):
        self.trimmed = False
        self.word2index = {}
        self.word2count = {}
        self.index2word = {PAD_token: "PAD", SOS_token: "SOS", EOS_token: "EOS", UNK_to
ken : "UNK"}
        self.num_words = 4  # Count SOS, EOS, PAD, UNK

    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

    # Remove words below a certain count threshold

    def trim(self, min_count):
        if self.trimmed:
            return
        self.trimmed = True

        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.word2inde
x)
        ))

        # Reinitialize dictionaries

        self.word2index = {}
        self.word2count = {}
        self.index2word = {PAD_token: "PAD", SOS_token: "SOS", EOS_token: "EOS", UNK_to
ken : "UNK"}
        self.num_words = 4 # Count default tokens

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

Now we can run through the dataset and assemble our vocabulary, as well as the training query/response sentence pairs. Before we can use the data, we have to perform some preprocessing. First, we convert Unicode strings to ASCII using `unicodeToAscii` . Then we convert every letter to lowercase and trim all non-letter characters except for basic punctuation (function `normalizeString`). Finally, to improve training convergence, we filter out sentences with length greater than the `MAX_LENGTH` threshold (function `filterPairs`).

In [14]:

```
MAX_LENGTH = 10  # Maximum sentence length to consider

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)
    s = re.sub(r"\s+", r" ", s).strip()
    return s

# Read query/response pairs and return a Voc object

def readVocs(datafile):
    print("Reading lines...")
    # Read the file and split into lines
    lines = open(datafile, encoding='utf-8').\
        read().strip().split('\n')
    # Split every line into pairs and normalize
    pairs = [[normalizeString(s) for s in l.split('\t')] for l in lines]
    voc = Voc()
    return voc, pairs

# Boolean function returning True iff both sentences in a pair 'p' are under the MAX_LE
NGTH threshold

def filterPair(p):
    # Input sequences need to preserve the last word for EOS token
    return len(p[0].split(' ')) < MAX_LENGTH and len(p[1].split(' ')) < MAX_LENGTH

# Filter pairs using the filterPair predicate

def filterPairs(pairs):
    return [pair for pair in pairs if filterPair(pair)]

# Using the functions defined above, return a populated voc object and pairs list

def loadPrepareData(datafile):
    print("Start preparing training data ...")
    voc, pairs = readVocs(datafile)
    print("Read {!s} sentence pairs".format(len(pairs)))
    pairs = filterPairs(pairs)
    print("Trimmed to {!s} sentence pairs".format(len(pairs)))
    print("Counting words...")
    for pair in pairs:
        voc.addSentence(pair[0])
        voc.addSentence(pair[1])
    print("Counted words:", voc.num_words)
    return voc, pairs
```

In [15]:

```
# wget https://github.com/dsai-asia/RTML/raw/main/Labs/11-LSTMs/chatDataset.txt

# Load/Assemble Voc and pairs

datafile = 'dialogueDataset.txt'
voc, pairs = loadPrepareData(datafile)

# Print some pairs to validate

print("\npairs:")
for pair in pairs[:10]:
    print(pair)
```

```
Start preparing training data ...
Reading lines...
Read 7372 sentence pairs
Trimmed to 596 sentence pairs
Counting words...
Counted words: 574

pairs:
['thanks for the service good day .', 'you re welcome ! have a great day
!']
['does it have free wifi ?', 'how about the avalon guesthouse ?']
['what is the price for that train ?', 'it is . pounds per person .']
['yes please book seats for me .', 'i will book the reservation now .']
['no thanks again for all of your help', 'okay thank you and goodbye !']
['i am planning a trip in cambridge', 'what can i help you with ?']
['no that will be all . thanks .', 'your welcome enjoy your trip']
['i will be travelling on wednesday .', 'alright where will you be departi
ng ?']
['how long will that trip take ?', 'the travel time for that train is minu
tes']
['no that is it . thank you !', 'you re welcome and have a great day !']
```

As already mentioned, we will trim out rarely-used words from the vocabulary. This will help improve convergence during training, because with a lower-dimensional input feature space, it will be easier to estimate the probability model $P(y \mid x)$. We trim as a two-step process:

1. Trim words appearing fewer than `MIN_COUNT` times with the previously-given `Voc.trim` method.
2. Filter out all sentence pairs containing trimmed words.

In [16]:

```
MIN_COUNT = 3    # Minimum word count threshold for trimming

def trimRareWords(voc, pairs, MIN_COUNT):
    # Trim words used under the MIN_COUNT from the voc
    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

        # Only keep pairs that do not contain trimmed word(s) in their input or output
sentence
        if keep_input and keep_output:
            keep_pairs.append(pair)

    print("Trimmed from {} pairs to {}, {:.4f} of total".format(len(pairs), len(keep_pa
irs), len(keep_pairs) / len(pairs)))
    return keep_pairs

# Trim vocabulary and pairs

pairs = trimRareWords(voc, pairs, MIN_COUNT)
```

```
keep_words 236 / 570 = 0.4140
Trimmed from 596 pairs to 362, 0.6074 of total
```

Split dataset into testing and training pair sets

Let's split the dataset into the first 45,000 pairs for training and the rest for testing:

In [17]:

```
testpairs = pairs[300:]
pairs  = pairs[:300]
```

Convert pairs to tensors

First, let's make tensors representing sentences in which we encode each sequence as a sequence of indices. The sequences should all be padded to to a length of `MAX_LENGTH` so that they are all the same size. The transformation will look like this:

title

`zeroPadding` does the padding.

The `inputVar` function handles the process of converting sentences to tensor, ultimately creating a correctly shaped zero-padded tensor. It also returns a tensor of `lengths` for each of the sequences in the batch which will be passed to our decoder later.

The `outputVar` function performs a similar function to `inputVar` , but instead of returning a `lengths` tensor, it returns a binary mask tensor and a maximum target sentence length. The binary mask tensor has the same shape as the output target tensor, but every element that is a *PAD_token* is 0 and all others are 1.

`batch2TrainData` simply takes a bunch of pairs and returns the input and target tensors using the aforementioned functions.

In [18]:

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

def zeroPadding(l, fillvalue=PAD_token):
    return list(itertools.zip_longest(*l, fillvalue=fillvalue))

def binaryMatrix(l, value=PAD_token):
    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

# Return a padded input sequence tensor and the lengths of each original sequence

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

# Return a padded target sequence tensor, a padding mask, and the max target length

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.BoolTensor(mask)
    padVar = torch.LongTensor(padList)
    return padVar, mask, max_target_len

# Return all items for a given batch of pairs

def batch2TrainData(voc, pair_batch):
    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

# Example for validation

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
```

As LSTM takes time series data so we need to convert our pairs of sentences into time series.

In [19]:

```
pair_batch = pairs[:5]
print(pair_batch)
pair_batch.sort(key=lambda x: len(x[0].split(" ")), reverse=True)
print(pair_batch)
print(target_variable)
print(mask)
print(max_target_len)
```

```
[['thanks for the service good day .', 'you re welcome ! have a great day
!'], ['no thanks again for all of your help', 'okay thank you and goodbye
!'], ['i am planning a trip in cambridge', 'what can i help you with ?'],
['no that will be all . thanks .', 'your welcome enjoy your trip'], ['no t
hat is it . thank you !', 'you re welcome and have a great day !']]
[['no thanks again for all of your help', 'okay thank you and goodbye !'],
['no that will be all . thanks .', 'your welcome enjoy your trip'], ['no t
hat is it . thank you !', 'you re welcome and have a great day !'], ['than
ks for the service good day .', 'you re welcome ! have a great day !'],
['i am planning a trip in cambridge', 'what can i help you with ?']]
tensor([[ 83,  32,  11,  15,  90],
        [ 36,  75,  12,  16,  17],
        [146,  15,  13,  17,   9],
        [ 57, 103,  10,   9,   2],
        [ 43,  48,  15,   2,   0],
        [  7,  20,  16,   0,   0],
        [ 10, 193,  17,   0,   0],
        [ 49,  10,   9,   0,   0],
        [ 10,   2,  10,   0,   0],
        [  2,   0,   2,   0,   0]])
tensor([[ True,  True,  True,  True,  True],
        [ True,  True,  True,  True,  True],
        [ True,  True,  True,  True,  True],
        [ True,  True,  True,  True,  True],
        [ True,  True,  True,  True, False],
        [ True,  True,  True, False, False],
        [ True,  True,  True, False, False],
        [ True,  True,  True, False, False],
        [ True,  True,  True, False, False],
        [ True, False,  True, False, False]])
10
```

1.3 Define models

Seq2Seq Model

The brains of our chatbot is a sequence-to-sequence (seq2seq) model. The goal of a seq2seq model is to take a variable-length sequence as an input, and return a variable-length sequence as an output using a fixed-sized model.

Sutskever et al. <https://arxiv.org/abs/1409.3215> __ discovered that by using two separate recurrent neural nets together, we can accomplish this task. One RNN acts as an **encoder**, which encodes a variable length input sequence to a fixed-length context vector. In theory, this context vector (the final hidden layer of the RNN) will contain semantic information about the query sentence that is input to the bot. The second RNN is a **decoder**, which takes an input word and the context vector, and returns a guess for the next word in the sequence and a hidden state to use in the next iteration.

title

Encoder

The encoder RNN iterates through the input sentence one token (e.g. word) at a time, at each time step outputting an “output” vector and a “hidden state” vector. The hidden state vector is then passed to the next time step, while the output vector is recorded. The encoder transforms the context it saw at each point in the sequence into a set of points in a high-dimensional space, which the decoder will use to generate a meaningful output for the given task.

At the heart of our encoder is a multi-layered LSTM. We will use a bidirectional variant of the LSTM, meaning that there are essentially two independent RNNs: one that is fed the input sequence in normal sequential order, and one that is fed the input sequence in reverse order. The outputs of each network are summed at each time step. Using a bidirectional LSTM will give us the advantage of encoding both past and future context.

title

Note that an `embedding` layer is used to encode our word indices in an arbitrarily sized feature space. For our models, this layer will map each word to a feature space of size *hidden_size*. When trained, these values should encode semantic similarity between similar meaning words.

Finally, if passing a padded batch of sequences to an RNN module, we must pack and unpack padding around the RNN pass using `nn.utils.rnn.pack_padded_sequence` and `nn.utils.rnn.pad_packed_sequence` respectively.

Computation Graph:

1) Convert word indexes to embeddings.

2) Pack padded batch of sequences for RNN module.

3) Forward pass through LSTM.

4) Unpack padding.

5) Sum bidirectional LSTM outputs.

6) Return output and final hidden state.

Inputs:

- `input_seq` : batch of input sentences; shape=\ *(max_length, batch_size)*
- `input_lengths` : list of sentence lengths corresponding to each sentence in the batch; shape=\ *(batch_size)*
- `hidden` : hidden state; shape=\ *(n_layers x num_directions, batch_size, hidden_size)*

Outputs:

- `outputs` : output features from the last hidden layer of the LSTM (sum of bidirectional outputs); shape=\ *(max_length, batch_size, hidden_size)*
- `hidden` : updated hidden state from LSTM; shape=\ *(n_layers x num_directions, batch_size, hidden_size)*

Encoder

Now we declare our encoder which is consist of bidirectional LSTM units.It is vital to declare bidirectional LSTM as their results are better than unidirectional LSTM in some NLP problems.What it done is instead of learning embeddings of previous words it also considers the embeddings or features of next word suitable to predict target variable.

In [246]:

```
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
        self.lstm = nn.LSTM(hidden_size, hidden_size, n_layers,
                          dropout=(0 if n_layers == 1 else dropout), bidirectional=Tru
e)

    def forward(self, input_seq, input_lengths, hidden=None):

        embedded = self.embedding(input_seq)

        packed = nn.utils.rnn.pack_padded_sequence(embedded, input_lengths.cpu())

        outputs, hidden = self.lstm(packed, hidden)

        outputs, _ = nn.utils.rnn.pad_packed_sequence(outputs)

        outputs = outputs[:, :, :self.hidden_size] + outputs[:, : ,self.hidden_size:]

        return outputs, hidden

    def init_hidden(self):

        return (torch.zeros(self.num_layers, self.batch_size, self.hidden_dim),
                torch.zeros(self.num_layers, self.batch_size, self.hidden_dim))
```

Decoder

The decoder RNN generates the response sentence in a token-by-token fashion. It uses the encoder's context vectors, and internal hidden states to generate the next word in the sequence. It continues generating words until it outputs an *EOS_token*, representing the end of the sentence. A common problem with a vanilla seq2seq decoder is that if we rely soley on the context vector to encode the entire input sequence's meaning, it is likely that we will have information loss. This is especially the case when dealing with long input sequences, greatly limiting the capability of our decoder.

To combat this, `Bahdanau et al. <https://arxiv.org/abs/1409.0473>` __ created an "attention mechanism" that allows the decoder to pay attention to certain parts of the input sequence, rather than using the entire fixed context at every step.

At a high level, attention is calculated using the decoder's current hidden state and the encoder's outputs. The output attention weights have the same shape as the input sequence, allowing us to multiply them by the encoder outputs, giving us a weighted sum which indicates the parts of encoder output to pay attention to. `Sean Robertson's <https://github.com/spro>` __ figure describes this very well:

title

Attention weights

In [247]:

```
class Attn(nn.Module):
    def __init__(self, method, hidden_size):
        super(Attn, self).__init__()
        self.method = method
        if self.method not in ['dot', 'general', 'concat']:
            raise ValueError(self.method, "is not an appropriate attention method.")
        self.hidden_size = hidden_size

    def dot_score(self, hidden, encoder_output):
        return torch.sum(hidden * encoder_output, dim=2)

    def forward(self, hidden, encoder_outputs):

        attn_energies = self.dot_score(hidden, encoder_outputs)
        attn_energies = attn_energies.t()
        return F.softmax(attn_energies, dim=1).unsqueeze(1)
```

Now that we have defined our attention submodule, we can implement the actual decoder model. For the decoder, we will manually feed our batch one time step at a time. This means that our embedded word tensor and LSTM output will both have shape *(1, batch_size, hidden_size)*.

Computation Graph:

1) Get embedding of current input word.

2) Forward through unidirectional LSTM.

3) Calculate attention weights from the current LSTM output from (2).

4) Multiply attention weights to encoder outputs to get new "weighted sum" context vector.

5) Concatenate weighted context vector and LSTM output using Luong eq. 5.

6) Predict next word using Luong eq. 6 (without softmax).

7) Return output and final hidden state.

Inputs:

- `input_step` : one time step (one word) of input sequence batch; shape=\ *(1, batch_size)*
- `last_hidden` : final hidden layer of LSTM; shape=\ *(n_layers x num_directions, batch_size, hidden_size)*
- `encoder_outputs` : encoder model's output; shape=\ *(max_length, batch_size, hidden_size)*

Outputs:

- `output` : softmax normalized tensor giving probabilities of each word being the correct next word in the decoded sequence; shape=\ *(batch_size, voc.num_words)*
- `hidden` : final hidden state of LSTM; shape=\ *(n_layers x num_directions, batch_size, hidden_size)*

Decoder

In [248]:

```
class LuongAttnDecoderRNN(nn.Module):
    def __init__(self, attn_model, embedding, hidden_size, output_size, n_layers=1, dro
pout=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


        self.embedding = embedding
        self.embedding_dropout = nn.Dropout(dropout)
        self.lstm = nn.LSTM(hidden_size, hidden_size, n_layers, dropout=(0 if n_layers
== 1 else dropout))#, bidirectional=True)
        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):
        embedded = self.embedding(input_step)
        embedded = self.embedding_dropout(embedded)
        rnn_output, hidden = self.lstm(embedded, last_hidden)
        attn_weights = self.attn(rnn_output, encoder_outputs)
        context = attn_weights.bmm(encoder_outputs.transpose(0, 1))
        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))
        output = self.out(concat_output)
        output = F.softmax(output, dim=1)
        return output, hidden
```

1.4 Training procedure

Masked loss

Since we are dealing with batches of padded sequences, we cannot simply consider all elements of the tensor when calculating loss. We define `maskNLLLoss` to calculate our loss based on our decoder's output tensor, the target tensor, and a binary mask tensor describing the padding of the target tensor. This loss function calculates the average negative log likelihood of the elements that correspond to a *1* in the mask tensor.

In [249]:

```
def maskNLLLoss(inp, target, mask):
    nTotal = mask.sum()
    crossEntropy = -torch.log(torch.gather(inp, 1, target.view(-1, 1)).squeeze(1))
    loss = crossEntropy.masked_select(mask).mean()
    loss = loss.to(device)
    return loss, nTotal.item()
```

Single training iteration

The `train` function contains the algorithm for a single training iteration (a single batch of inputs).

We will use a couple of clever tricks to aid in convergence:

- The first trick is using **teacher forcing**. This means that at some probability, set by `teacher_forcing_ratio`, we use the current target word as the decoder's next input rather than using the decoder's current guess. This technique acts as training wheels for the decoder, aiding in more efficient training. However, teacher forcing can lead to model instability during inference, as the decoder may not have a sufficient chance to truly craft its own output sequences during training. Thus, we must be mindful of how we are setting the `teacher_forcing_ratio`, and not be fooled by fast convergence.
- The second trick that we implement is **gradient clipping**. This is a commonly used technique for countering the "exploding gradient" problem. In essence, by clipping or thresholding gradients to a maximum value, we prevent the gradients from growing exponentially and either overflow (NaN), or overshoot steep cliffs in the cost function.

Sequence of Operations:

1) Forward pass entire input batch through encoder.

2) Initialize decoder inputs as SOS_token, and hidden state as the encoder's final hidden state.

3) Forward input batch sequence through decoder one time step at a time.

4) If teacher forcing: set next decoder input as the current target; else: set next decoder input as current decoder output.

5) Calculate and accumulate loss.

6) Perform backpropagation.

7) Clip gradients.

8) Update encoder and decoder model parameters.

Train function

In [250]:

```python
def train(input_variable, lengths, target_variable, mask, max_target_len, encoder, deco
der, embedding,
          encoder_optimizer, decoder_optimizer, batch_size, clip, max_length=MAX_LENGT
H):

    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)


    decoder_input = torch.LongTensor([[SOS_token for _ in range(batch_size)]])
    decoder_input = decoder_input.to(device)


    decoder_hidden = encoder_hidden[:decoder.n_layers]
    use_teacher_forcing = True if random.random() < teacher_forcing_ratio else False


    if use_teacher_forcing:
        for t in range(max_target_len):
            decoder_output, decoder_hidden = decoder(
                decoder_input, decoder_hidden, encoder_outputs
            )

            decoder_input = target_variable[t].view(1, -1)

            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
            )

            _, topi = decoder_output.topk(1)
            decoder_input = torch.LongTensor([[topi[i][0] for i in range(batch_size)]])
            decoder_input = decoder_input.to(device)

            mask_loss, nTotal = maskNLLLoss(decoder_output, target_variable[t], mask
[t])
            loss += mask_loss
            print_losses.append(mask_loss.item() * nTotal)
            n_totals += nTotal
```

```
    loss.backward()

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

    encoder_optimizer.step()
    decoder_optimizer.step()

    return sum(print_losses) / n_totals
```

Full training procedure

It is finally time to tie the full training procedure together with the data. The `trainIters` function is responsible for running `n_iterations` of training given the passed models, optimizers, data, etc. This function is quite self explanatory, as we have done the heavy lifting with the `train` function.

One thing to note is that when we save our model, we save a tarball containing the encoder and decoder state_dicts (parameters), the optimizers' state_dicts, the loss, the iteration, etc. Saving the model in this way will give us the ultimate flexibility with the checkpoint. After loading a checkpoint, we will be able to use the model parameters to run inference, or we can continue training right where we left off.

In [251]:

```
max_target_len
```

Out[251]:

```
10
```

In [252]:

```
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
    losslist = []
    if loadFilename:
        start_iteration = checkpoint['iteration'] + 1


    print("Training...")
    for iteration in range(start_iteration, n_iteration + 1):
        training_batch = training_batches[iteration - 1]

        input_variable, lengths, target_variable, mask, max_target_len = training_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


        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
            losslist.append(print_loss_avg)

        if (iteration % save_every == 0):
            directory = os.path.join(save_dir, model_name, corpus_name, '{}-{}_{}'.format(encoder_n_layers, decoder_n_layers, hidden_size))
            print(directory)
            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')))
    return losslist
```

1.5 Evaluation function

After training a model, we want to be able to talk to the bot ourselves. First, we must define how we want the model to decode the encoded input.

1.5.1 Greedy decoding

Greedy decoding is the decoding method that we use during training when we are **NOT** using teacher forcing. In other words, for each time step, we simply choose the word from `decoder_output` with the highest softmax value. This decoding method is optimal on a single time-step level.

To facilite the greedy decoding operation, we define a `GreedySearchDecoder` class. When run, an object of this class takes an input sequence (`input_seq`) of shape *(input_seq length, 1)*, a scalar input length (`input_length`) tensor, and a `max_length` to bound the response sentence length. The input sentence is evaluated using the following computational graph:

Computation Graph:

1) Forward input through encoder model.

2) Prepare encoder's final hidden layer to be first hidden input to the decoder.

3) Initialize decoder's first input as SOS_token.

4) Initialize tensors to append decoded words to.

5) Iteratively decode one word token at a time:

```
a) Forward pass through decoder.

b) Obtain most likely word token and its softmax score.

c) Record token and score.

d) Prepare current token to be next decoder input.
```

6) Return collections of word tokens and scores.

Greedy decoder

In [253]:

```
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):

        encoder_outputs, encoder_hidden = self.encoder(input_seq, input_length)

        decoder_hidden = encoder_hidden[:decoder.n_layers]

        decoder_input = torch.ones(1, 1, device=device, dtype=torch.long) * SOS_token

        all_tokens = torch.zeros([0], device=device, dtype=torch.long)
        all_scores = torch.zeros([0], device=device)

        for _ in range(max_length):

            decoder_output, decoder_hidden = self.decoder(decoder_input, decoder_hidde
n, encoder_outputs)
            decoder_scores, decoder_input = torch.max(decoder_output, dim=1)
            all_tokens = torch.cat((all_tokens, decoder_input), dim=0)
            all_scores = torch.cat((all_scores, decoder_scores), dim=0)

            decoder_input = torch.unsqueeze(decoder_input, 0)

        return all_tokens, all_scores
```

Evaluating some text

Now that we have our decoding method defined, we can write functions for evaluating a string input sentence. The `evaluate` function manages the low-level process of handling the input sentence. We first format the sentence as an input batch of word indexes with *batch_size==1*. We do this by converting the words of the sentence to their corresponding indexes, and transposing the dimensions to prepare the tensor for our models. We also create a `lengths` tensor which contains the length of our input sentence. In this case, `lengths` is scalar because we are only evaluating one sentence at a time (batch_size==1). Next, we obtain the decoded response sentence tensor using our `GreedySearchDecoder` object (`searcher`). Finally, we convert the response's indexes to words and return the list of decoded words.

`evaluateInput` acts as the user interface for our chatbot. When called, an input text field will spawn in which we can enter our query sentence. After typing our input sentence and pressing *Enter*, our text is normalized in the same way as our training data, and is ultimately fed to the `evaluate` function to obtain a decoded output sentence. We loop this process, so we can keep chatting with our bot until we enter either "q" or "quit".

Finally, if a sentence is entered that contains a word that is not in the vocabulary, we handle this gracefully by printing an error message and prompting the user to enter another sentence.

In [262]:

```
def evaluate(encoder, decoder, searcher, voc, sentence, max_length=MAX_LENGTH):

    indexes_batch = [indexesFromSentence(voc, sentence)]

    lengths = torch.tensor([len(indexes) for indexes in indexes_batch])

    input_batch = torch.LongTensor(indexes_batch).transpose(0, 1)
    # print(input_batch.shape)
    input_batch = input_batch.to(device)
    lengths = lengths.to(device)

    tokens, scores = searcher(input_batch, lengths, max_length)

    decoded_words = [voc.index2word[token.item()] for token in tokens]
    return decoded_words


def evaluateInput(encoder, decoder, searcher, voc):
    input_sentence = ''
    while(1):
        try:

            input_sentence = input('> ')

            if input_sentence == 'q' or input_sentence == 'quit': break

            input_sentence = normalizeString(input_sentence)

            output_words = evaluate(encoder, decoder, searcher, voc, input_sentence)

            output_words[:] = [x for x in output_words if not (x == 'EOS' or x == 'PA
D')]
            print('Bot:', ' '.join(output_words))

        except KeyError:
            print("Error: Encountered unknown word.")
```

1.5.1.1 Run Model Greedy

Finally, it is time to run our model!

Regardless of whether we want to train or test the chatbot model, we must initialize the individual encoder and decoder models. In the following block, we set our desired configurations, choose to start from scratch or set a checkpoint to load from, and build and initialize the models. Feel free to play with different model configurations to optimize performance.

In [255]:

```
model = 'cb_model'
attn_model = 'dot'

hidden_size = 512
encoder_n_layers = 2
decoder_n_layers = 4
dropout_p = 0.5
batch_size = 256
loadFilename = None

embedding = nn.Embedding(voc.num_words, hidden_size)
encoder = EncoderRNN(hidden_size, embedding, encoder_n_layers, dropout_p)
decoder = LuongAttnDecoderRNN(attn_model, embedding, hidden_size, voc.num_words, decode
r_n_layers, dropout_p)
encoder = encoder.to(device)
decoder = decoder.to(device)
```

In [97]:

```
save_dir = 'content/'
clip = 50.0
teacher_forcing_ratio = 1.0
learning_rate = 0.0001
decoder_learning_ratio = 5.0
n_iteration = 6000
print_every = 100
save_every = 2000
loadFilename = None
corpus_name= "Chat"
encoder.train()
decoder.train()
encoder_optimizer = optim.Adam(encoder.parameters(), lr=learning_rate)
decoder_optimizer = optim.Adam(decoder.parameters(), lr=learning_rate * decoder_learnin
g_ratio)
print("Starting Training!")
lossvalues = trainIters(model, 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)
```

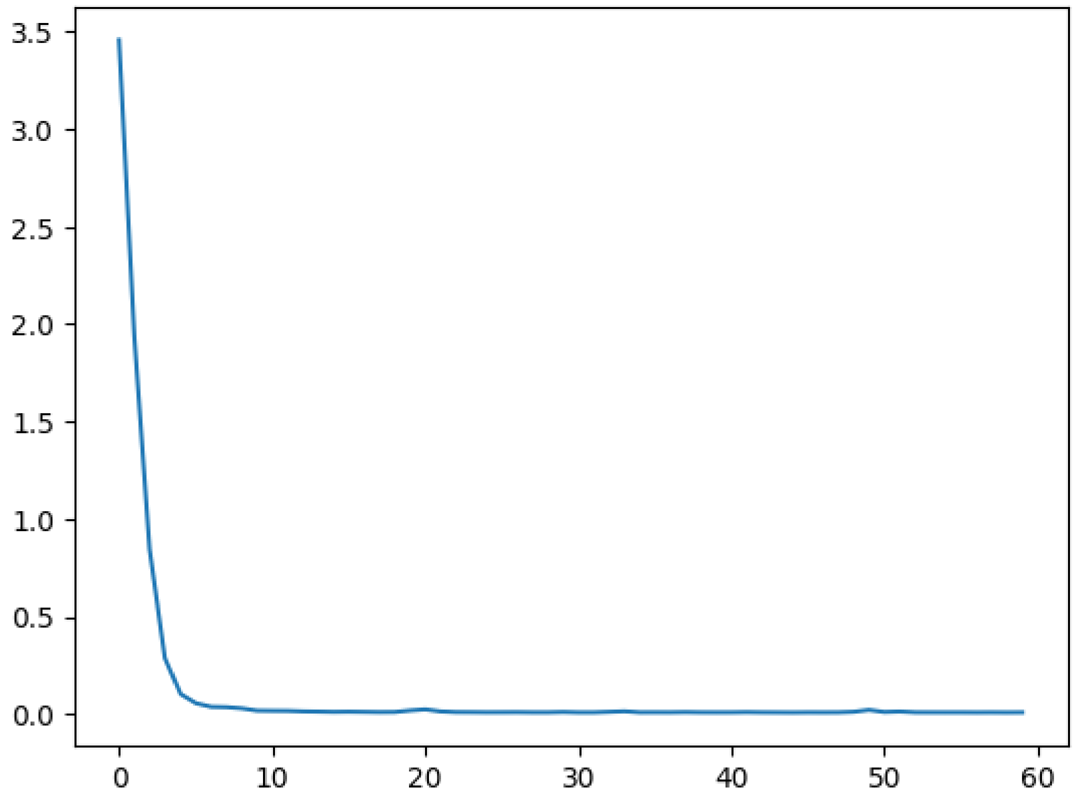
```
Starting Training!
Initializing ...
Training...
Iteration: 100; Percent complete: 1.7%; Average loss: 3.4569
Iteration: 200; Percent complete: 3.3%; Average loss: 1.9393
Iteration: 300; Percent complete: 5.0%; Average loss: 0.8445
Iteration: 400; Percent complete: 6.7%; Average loss: 0.2886
Iteration: 500; Percent complete: 8.3%; Average loss: 0.1038
Iteration: 600; Percent complete: 10.0%; Average loss: 0.0568
Iteration: 700; Percent complete: 11.7%; Average loss: 0.0385
Iteration: 800; Percent complete: 13.3%; Average loss: 0.0367
Iteration: 900; Percent complete: 15.0%; Average loss: 0.0305
Iteration: 1000; Percent complete: 16.7%; Average loss: 0.0189
Iteration: 1100; Percent complete: 18.3%; Average loss: 0.0180
Iteration: 1200; Percent complete: 20.0%; Average loss: 0.0176
Iteration: 1300; Percent complete: 21.7%; Average loss: 0.0150
Iteration: 1400; Percent complete: 23.3%; Average loss: 0.0137
Iteration: 1500; Percent complete: 25.0%; Average loss: 0.0123
Iteration: 1600; Percent complete: 26.7%; Average loss: 0.0131
Iteration: 1700; Percent complete: 28.3%; Average loss: 0.0121
Iteration: 1800; Percent complete: 30.0%; Average loss: 0.0111
Iteration: 1900; Percent complete: 31.7%; Average loss: 0.0116
Iteration: 2000; Percent complete: 33.3%; Average loss: 0.0195
content/cb_model/Chat/2-4_512
Iteration: 2100; Percent complete: 35.0%; Average loss: 0.0255
Iteration: 2200; Percent complete: 36.7%; Average loss: 0.0147
Iteration: 2300; Percent complete: 38.3%; Average loss: 0.0112
Iteration: 2400; Percent complete: 40.0%; Average loss: 0.0109
Iteration: 2500; Percent complete: 41.7%; Average loss: 0.0102
Iteration: 2600; Percent complete: 43.3%; Average loss: 0.0104
Iteration: 2700; Percent complete: 45.0%; Average loss: 0.0107
Iteration: 2800; Percent complete: 46.7%; Average loss: 0.0100
Iteration: 2900; Percent complete: 48.3%; Average loss: 0.0100
Iteration: 3000; Percent complete: 50.0%; Average loss: 0.0117
Iteration: 3100; Percent complete: 51.7%; Average loss: 0.0097
Iteration: 3200; Percent complete: 53.3%; Average loss: 0.0098
Iteration: 3300; Percent complete: 55.0%; Average loss: 0.0125
Iteration: 3400; Percent complete: 56.7%; Average loss: 0.0155
Iteration: 3500; Percent complete: 58.3%; Average loss: 0.0100
Iteration: 3600; Percent complete: 60.0%; Average loss: 0.0102
Iteration: 3700; Percent complete: 61.7%; Average loss: 0.0101
Iteration: 3800; Percent complete: 63.3%; Average loss: 0.0110
Iteration: 3900; Percent complete: 65.0%; Average loss: 0.0100
Iteration: 4000; Percent complete: 66.7%; Average loss: 0.0098
content/cb_model/Chat/2-4_512
Iteration: 4100; Percent complete: 68.3%; Average loss: 0.0099
Iteration: 4200; Percent complete: 70.0%; Average loss: 0.0109
Iteration: 4300; Percent complete: 71.7%; Average loss: 0.0099
Iteration: 4400; Percent complete: 73.3%; Average loss: 0.0097
Iteration: 4500; Percent complete: 75.0%; Average loss: 0.0093
Iteration: 4600; Percent complete: 76.7%; Average loss: 0.0098
Iteration: 4700; Percent complete: 78.3%; Average loss: 0.0097
Iteration: 4800; Percent complete: 80.0%; Average loss: 0.0099
Iteration: 4900; Percent complete: 81.7%; Average loss: 0.0127
Iteration: 5000; Percent complete: 83.3%; Average loss: 0.0225
Iteration: 5100; Percent complete: 85.0%; Average loss: 0.0118
Iteration: 5200; Percent complete: 86.7%; Average loss: 0.0142
Iteration: 5300; Percent complete: 88.3%; Average loss: 0.0101
Iteration: 5400; Percent complete: 90.0%; Average loss: 0.0099
```

```
Iteration: 5500; Percent complete: 91.7%; Average loss: 0.0098
Iteration: 5600; Percent complete: 93.3%; Average loss: 0.0098
Iteration: 5700; Percent complete: 95.0%; Average loss: 0.0095
Iteration: 5800; Percent complete: 96.7%; Average loss: 0.0098
Iteration: 5900; Percent complete: 98.3%; Average loss: 0.0095
Iteration: 6000; Percent complete: 100.0%; Average loss: 0.0099
content/cb_model/Chat/2-4_512
```

Plot graph

In [98]:

```
import matplotlib.pyplot as plt
plt.plot(lossvalues)
plt.show()
```



BLEU score calculation

The BLEU score (the Bilingual Evaluation Understudy Score) is an evaluation metric used to calculate the capacity of our model to make correct predictions. It finds maximal n-gram matches between predicted sentences and reference sentences You can learn more about BLEU from [this tutorial (https://machinelearningmastery.com/calculate-bleu-score-for-text-python/)](https://machinelearningmastery.com/calculate-bleu-score-for-text-python/).

In [99]:

```
# !pip install nltk
```

In [263]:

```
from nltk.translate.bleu_score import sentence_bleu,corpus_bleu
from nltk.translate.bleu_score import SmoothingFunction

# Set dropout layers to eval mode

encoder.eval()
decoder.eval()

# Initialize search module

searcher = GreedySearchDecoder(encoder, decoder)
gram1_bleu_score = []
gram2_bleu_score = []
for i in range(0,len(testpairs),1):
    input_sentence = testpairs[i][0]
    reference = testpairs[i][1:]
    templist = []
    for k in range(len(reference)):
        if(reference[k]!=''):
            temp = reference[k].split(' ')
            templist.append(temp)

    input_sentence = normalizeString(input_sentence)
    output_words = evaluate(encoder, decoder, searcher, voc, input_sentence)
    output_words[:] = [x for x in output_words if not (x == 'EOS' or x == 'PAD')]
    chencherry = SmoothingFunction()
    #   print(output_words)
    #   print(templist)
    score1 = sentence_bleu(templist, output_words, weights=(1, 0, 0, 0), smoothing_function=chencherry.method1)
    score2 = sentence_bleu(templist, output_words, weights=(0.5, 0.5, 0, 0), smoothing_function=chencherry.method1)
    gram1_bleu_score.append(score1)
    gram2_bleu_score.append(score2)
    if i%1000 == 0:
        print(i,sum(gram1_bleu_score)/len(gram1_bleu_score),sum(gram2_bleu_score)/len(gram2_bleu_score))

print("Total Bleu Score for 1 grams on testing pairs: ", sum(gram1_bleu_score)/len(gram1_bleu_score) )
print("Total Bleu Score for 2 grams on testing pairs: ", sum(gram2_bleu_score)/len(gram2_bleu_score) )
```

```
0 0.0 0.0
Total Bleu Score for 1 grams on testing pairs:  0.0016129032258064518
Total Bleu Score for 2 grams on testing pairs:  0.0005376344086021508
```

Chat with the greedy search bot

Let's chat with the new chatbot:

In [ ]:

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

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

# Begin chatting (uncomment and run the following line to begin)
evaluateInput(encoder, decoder, searcher, voc)
# input
# Hi, how are you?
# What
# I don't understand you
# hmm, good bye
```

1.5.2 Beam search

Beam search is an improved version of greedy search. It has a hyperparameter named beam size, k. At time step 1, selecting k tokens with the highest conditional probabilities. Each of them will be the first token of k candidate output sequences, respectively. At each subsequent time step, based on the k candidate output sequences at the previous time step, k candidate output sequences has been selected with the highest conditional probabilities from $k|\gamma|$ possible choices.

Beam search

the sequence with the highest of the following score as the output sequence has been chosen from the equation:

$\frac{1}{L^{\alpha}}\log{P(y_1,...,y_L)} = \frac{1}{L^{\alpha}} \sum_{t'=1}^{L} \log{P(y_{t'}|t_1,...,y_{t'-1}, c)}$

Where L is the length of the final candidate sequence and α is usually set to 0.75.

Beam Decoder

The difference between greedy search and beam search is decoder function. Thus, greedy search function name is greedy_decode, and beam search function name is beam_decode.

In [48]:

```python
class Sentence:
    def __init__(self, decoder_hidden, last_idx=SOS_token, sentence_idxes=[], sentence_
scores=[]):
        if(len(sentence_idxes) != len(sentence_scores)):
            raise ValueError("length of indexes and scores should be the same")
        self.decoder_hidden = decoder_hidden
        self.last_idx = last_idx
        self.sentence_idxes =  sentence_idxes
        self.sentence_scores = sentence_scores

    def avgScore(self):
        if len(self.sentence_scores) == 0:
            raise ValueError("Calculate average score of sentence, but got no word")
        # return mean of sentence_score
        return sum(self.sentence_scores) / len(self.sentence_scores)

    def addTopk(self, topi, topv, decoder_hidden, beam_size, voc):
        topv = torch.log(topv)
        terminates, sentences = [], []
        for i in range(beam_size):
            if topi[0][i] == EOS_token:
                terminates.append(([voc.index2word[idx.item()] for idx in self.sentence
_idxes] + ['<EOS>'],
                                   self.avgScore()))
                continue
            idxes = self.sentence_idxes[:]
            scores = self.sentence_scores[:]
            idxes.append(topi[0][i])
            scores.append(topv[0][i])
            sentences.append(Sentence(decoder_hidden, topi[0][i], idxes, scores))
        return terminates, sentences

    def toWordScore(self, voc):

        words = []
        for i in range(len(self.sentence_idxes)):
            if self.sentence_idxes[i] == EOS_token:
                words.append('<EOS>')
            else:
                words.append(voc.index2word[self.sentence_idxes[i].item()])

        if self.sentence_idxes[-1] != EOS_token:
            words.append('<EOS>')
        return (words, self.avgScore())

    def __repr__(self):
        res = f"Sentence with indices {self.sentence_idxes} "
        res += f"and scores {self.sentence_scores}"
        return res


def beam_decode(decoder, decoder_hidden, encoder_outputs, voc, beam_size, max_length=MA
X_LENGTH):
    terminal_sentences, prev_top_sentences, next_top_sentences = [], [], []
    prev_top_sentences.append(Sentence(decoder_hidden))
    for i in range(max_length):

        for sentence in prev_top_sentences:
            decoder_input = torch.LongTensor([[sentence.last_idx]])
```

```python
            decoder_input = decoder_input.to(device)

            decoder_hidden = sentence.decoder_hidden
            decoder_output, decoder_hidden = decoder(decoder_input, decoder_hidden, enc
oder_outputs)
            topv, topi = decoder_output.topk(beam_size)
            term, top = sentence.addTopk(topi, topv, decoder_hidden, beam_size, voc)
            terminal_sentences.extend(term)
            next_top_sentences.extend(top)


        next_top_sentences.sort(key=lambda s: s.avgScore(), reverse=True)
        prev_top_sentences = next_top_sentences[:beam_size]
        next_top_sentences = []


    terminal_sentences += [sentence.toWordScore(voc) for sentence in prev_top_sentence
s]
    terminal_sentences.sort(key=lambda x: x[1], reverse=True)

    n = min(len(terminal_sentences), 15)
    return terminal_sentences[:n]


class BeamSearchDecoder(nn.Module):

    def __init__(self, encoder, decoder, voc, beam_size=10):
        super(BeamSearchDecoder, self).__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.voc = voc
        self.beam_size = beam_size

    def forward(self, input_seq, input_length, max_length):
        encoder_outputs, encoder_hidden = self.encoder(input_seq, input_length)

        decoder_hidden = encoder_hidden[:self.decoder.n_layers]

        decoder_input = torch.ones(1, 1, device=device, dtype=torch.long) * SOS_token
        sentences = beam_decode(self.decoder, decoder_hidden, encoder_outputs, self.vo
c, self.beam_size, max_length)


        all_tokens = [torch.tensor(self.voc.word2index.get(w, 0)) for w in sentences[0]
[0]]
        return all_tokens, None

    def __str__(self):
        res = f"BeamSearchDecoder with beam size {self.beam_size}"
        return res
```

In [38]:

```
model = 'cb_model'
attn_model = 'dot'

hidden_size = 512
encoder_n_layers = 2
decoder_n_layers = 4
dropout_p = 0.5
batch_size = 256
loadFilename = None

embedding = nn.Embedding(voc.num_words, hidden_size)
encoder = EncoderRNN(hidden_size, embedding, encoder_n_layers, dropout_p)
decoder = LuongAttnDecoderRNN(attn_model, embedding, hidden_size, voc.num_words, decode
r_n_layers, dropout_p)
encoder = encoder.to(device)
decoder = decoder.to(device)
```

1.5.2.1 Run Model Beam Search

In [103]:

```
save_dir = 'content/'
clip = 50.0
teacher_forcing_ratio = 1.0
learning_rate = 0.0001
decoder_learning_ratio = 5.0
n_iteration = 6000
print_every = 100
save_every = 2000
loadFilename = None
corpus_name="Chat"
encoder.train()
decoder.train()
encoder_optimizer = optim.Adam(encoder.parameters(), lr=learning_rate)
decoder_optimizer = optim.Adam(decoder.parameters(), lr=learning_rate * decoder_learnin
g_ratio)
print("Starting Training!")
lossvalues = trainIters(model, 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)
```

```
Starting Training!
Initializing ...
Training...
Iteration: 100; Percent complete: 1.7%; Average loss: 3.4290
Iteration: 200; Percent complete: 3.3%; Average loss: 1.8739
Iteration: 300; Percent complete: 5.0%; Average loss: 0.7766
Iteration: 400; Percent complete: 6.7%; Average loss: 0.2634
Iteration: 500; Percent complete: 8.3%; Average loss: 0.1071
Iteration: 600; Percent complete: 10.0%; Average loss: 0.0519
Iteration: 700; Percent complete: 11.7%; Average loss: 0.0335
Iteration: 800; Percent complete: 13.3%; Average loss: 0.0257
Iteration: 900; Percent complete: 15.0%; Average loss: 0.0221
Iteration: 1000; Percent complete: 16.7%; Average loss: 0.0272
Iteration: 1100; Percent complete: 18.3%; Average loss: 0.0172
Iteration: 1200; Percent complete: 20.0%; Average loss: 0.0151
Iteration: 1300; Percent complete: 21.7%; Average loss: 0.0136
Iteration: 1400; Percent complete: 23.3%; Average loss: 0.0145
Iteration: 1500; Percent complete: 25.0%; Average loss: 0.0177
Iteration: 1600; Percent complete: 26.7%; Average loss: 0.0129
Iteration: 1700; Percent complete: 28.3%; Average loss: 0.0121
Iteration: 1800; Percent complete: 30.0%; Average loss: 0.0118
Iteration: 1900; Percent complete: 31.7%; Average loss: 0.0114
Iteration: 2000; Percent complete: 33.3%; Average loss: 0.0124
content/cb_model/Chat/2-4_512
Iteration: 2100; Percent complete: 35.0%; Average loss: 0.0118
Iteration: 2200; Percent complete: 36.7%; Average loss: 0.0107
Iteration: 2300; Percent complete: 38.3%; Average loss: 0.0111
Iteration: 2400; Percent complete: 40.0%; Average loss: 0.0115
Iteration: 2500; Percent complete: 41.7%; Average loss: 0.0121
Iteration: 2600; Percent complete: 43.3%; Average loss: 0.0114
Iteration: 2700; Percent complete: 45.0%; Average loss: 0.0151
Iteration: 2800; Percent complete: 46.7%; Average loss: 0.0168
Iteration: 2900; Percent complete: 48.3%; Average loss: 0.0103
Iteration: 3000; Percent complete: 50.0%; Average loss: 0.0170
Iteration: 3100; Percent complete: 51.7%; Average loss: 0.0136
Iteration: 3200; Percent complete: 53.3%; Average loss: 0.0104
Iteration: 3300; Percent complete: 55.0%; Average loss: 0.0118
Iteration: 3400; Percent complete: 56.7%; Average loss: 0.0118
Iteration: 3500; Percent complete: 58.3%; Average loss: 0.0107
Iteration: 3600; Percent complete: 60.0%; Average loss: 0.0104
Iteration: 3700; Percent complete: 61.7%; Average loss: 0.0097
Iteration: 3800; Percent complete: 63.3%; Average loss: 0.0101
Iteration: 3900; Percent complete: 65.0%; Average loss: 0.0096
Iteration: 4000; Percent complete: 66.7%; Average loss: 0.0097
content/cb_model/Chat/2-4_512
Iteration: 4100; Percent complete: 68.3%; Average loss: 0.0097
Iteration: 4200; Percent complete: 70.0%; Average loss: 0.0098
Iteration: 4300; Percent complete: 71.7%; Average loss: 0.0100
Iteration: 4400; Percent complete: 73.3%; Average loss: 0.0102
Iteration: 4500; Percent complete: 75.0%; Average loss: 0.0095
Iteration: 4600; Percent complete: 76.7%; Average loss: 0.0123
Iteration: 4700; Percent complete: 78.3%; Average loss: 0.0128
Iteration: 4800; Percent complete: 80.0%; Average loss: 0.0100
Iteration: 4900; Percent complete: 81.7%; Average loss: 0.0100
Iteration: 5000; Percent complete: 83.3%; Average loss: 0.0096
Iteration: 5100; Percent complete: 85.0%; Average loss: 0.0098
Iteration: 5200; Percent complete: 86.7%; Average loss: 0.0096
Iteration: 5300; Percent complete: 88.3%; Average loss: 0.0094
Iteration: 5400; Percent complete: 90.0%; Average loss: 0.0092
```

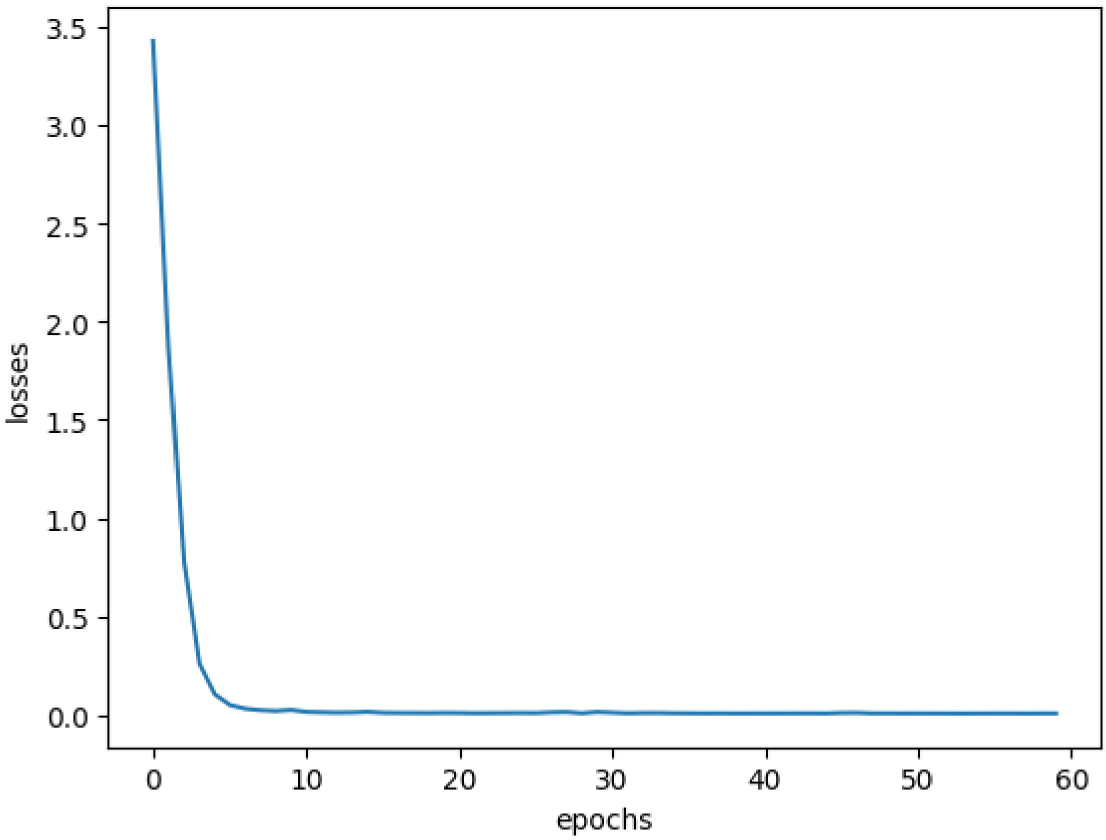
```
Iteration: 5500; Percent complete: 91.7%; Average loss: 0.0096
Iteration: 5600; Percent complete: 93.3%; Average loss: 0.0097
Iteration: 5700; Percent complete: 95.0%; Average loss: 0.0098
Iteration: 5800; Percent complete: 96.7%; Average loss: 0.0093
Iteration: 5900; Percent complete: 98.3%; Average loss: 0.0098
Iteration: 6000; Percent complete: 100.0%; Average loss: 0.0097
content/cb_model/Chat/2-4_512
```

Plot graph

In [104]:

```
import matplotlib.pyplot as plt

plt.plot(lossvalues)
plt.ylabel('losses')
plt.xlabel('epochs')
plt.show()
```



BLEU score calculation

Let's calculate the BLEU score with the beam search decoder:

In [105]:

```
from nltk.translate.bleu_score import sentence_bleu,corpus_bleu
from nltk.translate.bleu_score import SmoothingFunction

# Set dropout layers to eval mode

encoder.eval()
decoder.eval()

# Initialize search module

############################################################################
# Difference between greedy search and beam search is here

# greedy search
# searcher = GreedySearchDecoder(encoder, decoder)

# beam search
searcher = BeamSearchDecoder(encoder, decoder, voc, 3)
############################################################################
gram1_bleu_score = []
gram2_bleu_score = []

for i in range(0,len(testpairs),1):
    input_sentence = testpairs[i][0]

    reference = testpairs[i][1:]
    templist = []
    for k in range(len(reference)):
        if(reference[k]!=''):
            temp = reference[k].split(' ')
            templist.append(temp)

    input_sentence = normalizeString(input_sentence)
    output_words = evaluate(encoder, decoder, searcher, voc, input_sentence)
    output_words[:] = [x for x in output_words if not (x == 'EOS' or x == 'PAD')]
    chencherry = SmoothingFunction()
    score1 = sentence_bleu(templist,output_words,weights=(1, 0, 0, 0) ,smoothing_functi
on=chencherry.method1)
    score2 = sentence_bleu(templist,output_words,weights=(0.5, 0.5, 0, 0),smoothing_fun
ction=chencherry.method1)
    gram1_bleu_score.append(score1)
    gram2_bleu_score.append(score2)
    if i%1000 == 0:
        print(i,sum(gram1_bleu_score)/len(gram1_bleu_score),sum(gram2_bleu_score)/len(g
ram2_bleu_score))

print("Total Bleu Score for 1 grams on testing pairs: ", sum(gram1_bleu_score)/len(gram
1_bleu_score))
print("Total Bleu Score for 2 grams on testing pairs: ", sum(gram2_bleu_score)/len(gram
2_bleu_score))
```

```
0 0.0 0.0
Total Bleu Score for 1 grams on testing pairs:  0.35400078957937203
Total Bleu Score for 2 grams on testing pairs:  0.2196769646729017
```

Chat with the beam search bot

Let's chat with the new chatbot:

In [1]:

```
encoder.eval()
decoder.eval()
searcher = BeamSearchDecoder(encoder, decoder, voc, 3)
evaluateInput(encoder, decoder, searcher, voc)
```

Conclusion

using LSTM cells in a model called seq2seq. The seq2seq model consists of encoder and decoder network both of which includes LSTM cells. The encoder encode sequence of words and embed them as a representation then the decoder takes the embedded representation and decode another sequence of words. We use this architecture to make a chat bot where it takes a sentence as input and output a response or answer.

There are 2 search algorithm we can use in the decoder part. First is the greedy search which choose the word with maximum probability at every step but might not end up with the optimum sentence. Second is beam search which adds the k candidate to the greedy search.

The first part we trained a chatbot from a movie dialogue corpus and for the homework I trained a chatbot with real a human-human conversations.

2. Replace the LSTM encoder/decoder with a Transformer.

Check out the [PyTorch Transformer module documentation (https://pytorch.org/docs/stable/generated/torch.nn.Transformer.html)](https://pytorch.org/docs/stable/generated/torch.nn.Transformer.html)

In [20]:

```
import torch, torchdata, torchtext
from torch import nn
import torch.nn.functional as F
import pandas as pd
import numpy as np
import random, math, time

device = torch.device('cuda:2' if torch.cuda.is_available() else 'cpu')
print(device)

#make our work comparable if restarted the kernel
SEED = 1234
torch.manual_seed(SEED)
torch.backends.cudnn.deterministic = True
```

```
cpu
```

$ P_{i,j}= \begin{cases} sin(\frac{1}{1000^\frac{j}{d_{embed}}}),& \text{if j is even}\\ cos(\frac{1}{1000^\frac{j}{d_{embed}}}),& \text{if j is odd} \end{cases} $

In [21]:

```
class PositionalEncoding(nn.Module):
    def __init__(self, dim_model, dropout_p, max_len = 100):
        super().__init__()
        # max_len determines how far the position can have an effect on a token (windo
w)

        # Info
        self.dropout = nn.Dropout(dropout_p)

        # Encoding - From formula
        pos_encoding = torch.zeros(max_len, dim_model)
        positions_list = torch.arange(0, max_len, dtype=torch.float).view(-1, 1) # 0,
1, 2, 3, 4, 5
        division_term = torch.exp(torch.arange(0, dim_model, 2).float() * (-math.log(10
000.0)) / dim_model) # 1000^(2i/dim_model)

        # PE(pos, 2i) = sin(pos/1000^(2i/dim_model))
        pos_encoding[:, 0::2] = torch.sin(positions_list * division_term)

        # PE(pos, 2i + 1) = cos(pos/1000^(2i/dim_model))
        pos_encoding[:, 1::2] = torch.cos(positions_list * division_term)

        # Saving buffer (same as parameter without gradients needed)
        pos_encoding = pos_encoding.unsqueeze(0).transpose(0, 1)
        self.register_buffer("pos_encoding",pos_encoding)

    def forward(self, token_embedding: torch.tensor) -> torch.tensor:
        # Residual connection + pos encoding
        return self.dropout(token_embedding + self.pos_encoding[:token_embedding.size
(0), :])
```

In [22]:

```python
class Sentence:
    def __init__(self, decoder_hidden, last_idx=SOS_token, sentence_idxes=[], sentence_scores=[]):
        if(len(sentence_idxes) != len(sentence_scores)):
            raise ValueError("length of indexes and scores should be the same")
        self.decoder_hidden = decoder_hidden
        self.last_idx = last_idx
        self.sentence_idxes =  sentence_idxes
        self.sentence_scores = sentence_scores

    def avgScore(self):
        if len(self.sentence_scores) == 0:
            raise ValueError("Calculate average score of sentence, but got no word")
        # return mean of sentence_score
        return sum(self.sentence_scores) / len(self.sentence_scores)

    def addTopk(self, topi, topv, decoder_hidden, beam_size, voc):
        topv = torch.log(topv)
        terminates, sentences = [], []
        for i in range(beam_size):
            if topi[0][i] == EOS_token:
                terminates.append(([voc.index2word[idx.item()] for idx in self.sentence_idxes] + ['<EOS>'],
                                   self.avgScore()))
                continue
            idxes = self.sentence_idxes[:]
            scores = self.sentence_scores[:]
            idxes.append(topi[0][i])
            scores.append(topv[0][i])
            sentences.append(Sentence(decoder_hidden, topi[0][i], idxes, scores))
        return terminates, sentences

    def toWordScore(self, voc):

        words = []
        for i in range(len(self.sentence_idxes)):
            if self.sentence_idxes[i] == EOS_token:
                words.append('<EOS>')
            else:
                words.append(voc.index2word[self.sentence_idxes[i].item()])

        if self.sentence_idxes[-1] != EOS_token:
            words.append('<EOS>')
        return (words, self.avgScore())

    def __repr__(self):
        res = f"Sentence with indices {self.sentence_idxes} "
        res += f"and scores {self.sentence_scores}"
        return res


def beam_decode(decoder, decoder_hidden, encoder_outputs, voc, beam_size, max_length=MAX_LENGTH):
    terminal_sentences, prev_top_sentences, next_top_sentences = [], [], []
    prev_top_sentences.append(Sentence(decoder_hidden))
    for i in range(max_length):

        for sentence in prev_top_sentences:
            decoder_input = torch.LongTensor([[sentence.last_idx]])
```

```
            decoder_input = decoder_input.to(device)

            decoder_hidden = sentence.decoder_hidden
            decoder_output, decoder_hidden = decoder(decoder_input, decoder_hidden, encoder_outputs)
            topv, topi = decoder_output.topk(beam_size)
            term, top = sentence.addTopk(topi, topv, decoder_hidden, beam_size, voc)
            terminal_sentences.extend(term)
            next_top_sentences.extend(top)


        next_top_sentences.sort(key=lambda s: s.avgScore(), reverse=True)
        prev_top_sentences = next_top_sentences[:beam_size]
        next_top_sentences = []


    terminal_sentences += [sentence.toWordScore(voc) for sentence in prev_top_sentences]
    terminal_sentences.sort(key=lambda x: x[1], reverse=True)

    n = min(len(terminal_sentences), 15)
    return terminal_sentences[:n]


class BeamSearchDecoder(nn.Module):

    def __init__(self, encoder, decoder, voc, beam_size=10):
        super(BeamSearchDecoder, self).__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.voc = voc
        self.beam_size = beam_size

    def forward(self, input_seq, input_length, max_length):
        encoder_outputs, encoder_hidden = self.encoder(input_seq, input_length)

        decoder_hidden = encoder_hidden[:self.decoder.n_layers]

        decoder_input = torch.ones(1, 1, device=device, dtype=torch.long) * SOS_token
        sentences = beam_decode(self.decoder, decoder_hidden, encoder_outputs, self.voc, self.beam_size, max_length)

        all_tokens = [torch.tensor(self.voc.word2index.get(w, 0)) for w in sentences[0][0]]
        return all_tokens, None

    def __str__(self):
        res = f"BeamSearchDecoder with beam size {self.beam_size}"
        return res
```

In [23]:

```python
class Transformer(nn.Module):
    """Container module with an encoder, a recurrent or transformer module, and a decoder."""
    def __init__(self, input_dim, num_heads, emb_dim, hid_dim, num_layers, dropout_p=0.5):
        super(Transformer, self).__init__()
        try:
            from torch.nn import TransformerEncoder, TransformerEncoderLayer, TransformerDecoder, TransformerDecoderLayer
        except BaseException as e:
            raise ImportError('TransformerEncoder module does not exist in PyTorch 1.1 or lower.') from e

        self.model_type = 'Transformer'
        self.src_mask = None
        self.pos_encoder = PositionalEncoding(emb_dim, dropout_p = 0.0)

        encoder_layers = TransformerEncoderLayer(emb_dim, num_heads, hid_dim, dropout_p)
        self.transformer_encoder = TransformerEncoder(encoder_layers, num_layers)

        decoder_layers = TransformerDecoderLayer(emb_dim, num_heads, hid_dim, dropout_p)
        self.transformer_decoder = TransformerDecoder(decoder_layers, num_layers)

        self.input_dim = input_dim
        self.embedding = nn.Embedding(input_dim, emb_dim)
        self.decoder = nn.Linear(emb_dim, input_dim)
        self.hid_dim = hid_dim
        self.init_weights()

    def _generate_square_subsequent_mask(self, sz):
        mask = (torch.triu(torch.ones(sz, sz)) == 1).transpose(0, 1)
        mask = mask.float().masked_fill(mask == 0, float('-inf')).masked_fill(mask == 1, float(0.0))
        return mask

    def init_weights(self):
        initrange = 0.1
        nn.init.uniform_(self.embedding.weight, -initrange, initrange)
        nn.init.zeros_(self.decoder.bias)
        nn.init.uniform_(self.decoder.weight, -initrange, initrange)

    def forward(self, src, trg, has_mask=True):
        if has_mask:
            device = src.device
            if self.src_mask is None or self.src_mask.size(0) != len(src):
                mask = self._generate_square_subsequent_mask(len(src)).to(device)
                self.src_mask = mask
        else:
            self.src_mask = None
        # print(src.shape)
        src = self.embedding(src) * math.sqrt(self.hid_dim)
        trg = self.embedding(trg) * math.sqrt(self.hid_dim)
        #src = [batch size, seq len, hid dim]
        #src = [batch size, seq len, hid dim]
        src = self.pos_encoder(src)
        trg = self.pos_encoder(trg)
        # print('src shape', src.shape)
```

```
        # print('trg shape', trg.shape)
        encoder_output = self.transformer_encoder(src, self.src_mask)

        decoder_output = self.transformer_decoder(trg, encoder_output)
        # decoder_output = self.decoder(decoder_output)

        return decoder_output #F.log_softmax(decoder_output, dim=-1)
```

In [24]:

```
save_dir = 'content/'
clip = 50.0
teacher_forcing_ratio = 1.0
learning_rate = 0.0001
decoder_learning_ratio = 5.0
n_iteration = 2000
print_every = 100
save_every = 2000
loadFilename = None
corpus_name = "ChatTransformer"
batch_size  = 256
hidden_size = 256
embedding = nn.Embedding(voc.num_words, hidden_size)
model = Transformer(input_dim = voc.num_words, num_heads=2, emb_dim = 256, hid_dim = hi
dden_size, num_layers=2, dropout_p=0.1).to(device)
learning_rate = 0.0001
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
```

In [25]:

```
encoder_optimizer = optim.Adam(model.transformer_encoder.parameters(), lr=learning_rat
e)
decoder_optimizer = optim.Adam(model.transformer_decoder.parameters(), lr=learning_rat
e)
```

In [26]:

```
criterion = nn.CrossEntropyLoss()
encoder_n_layers = 2
decoder_n_layers = 2
```

In [27]:

```
import os
def trainIters_transformer(model, voc, pairs, optimizer, embedding, save_dir, n_iterati
on, 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
    losslist = []
    if loadFilename:
        start_iteration = checkpoint['iteration'] + 1

    print("Training...")
    for iteration in range(start_iteration, n_iteration + 1):
        # print(iteration)
        training_batch = training_batches[iteration - 1]

        input_variable, lengths, target_variable, mask, max_target_len = training_batch
        # print(input_variable.shape)
        # print(target_variable.shape)
        output = model(input_variable, target_variable)

        #output = [batch size, trg len - 1, output dim]
        #trg    = [batch size, trg len]
        # print(output.shape)
        # print(target_variable.shape)
        output_dim = output.shape[-1]

        output = output.reshape(-1, output_dim)
        target_variable = target_variable[:,:].reshape(-1)

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

        loss = criterion(output, target_variable)

        optimizer.zero_grad()

        loss.backward()

        torch.nn.utils.clip_grad_norm_(model.parameters(), 0.5)

        optimizer.step()

        print_loss += loss

        if iteration % print_every == 0:
            print_loss_avg = print_loss / print_every
            print("Iteration: {}; Percent complete: {:.1f}%; Average loss: {:.4f}".form
at(iteration, iteration / n_iteration * 100, print_loss_avg))
            print_loss = 0
            losslist.append(print_loss_avg)

        if (iteration % save_every == 0):
```

```
            directory = os.path.join(save_dir,'{}-{}_{}'.format(encoder_n_layers, decod
er_n_layers, hidden_size))
            if not os.path.exists(directory):
                os.makedirs(directory)
            torch.save({
                'iteration': iteration,
                'en': model.transformer_encoder.state_dict(),
                'de': model.transformer_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')))
    return losslist
```

In [28]:

```
print("Starting Training!")
lossvalues = trainIters_transformer(model, voc, pairs, optimizer,
           embedding, save_dir, n_iteration, batch_size,
           print_every, save_every, clip, corpus_name, loadFilename)
```

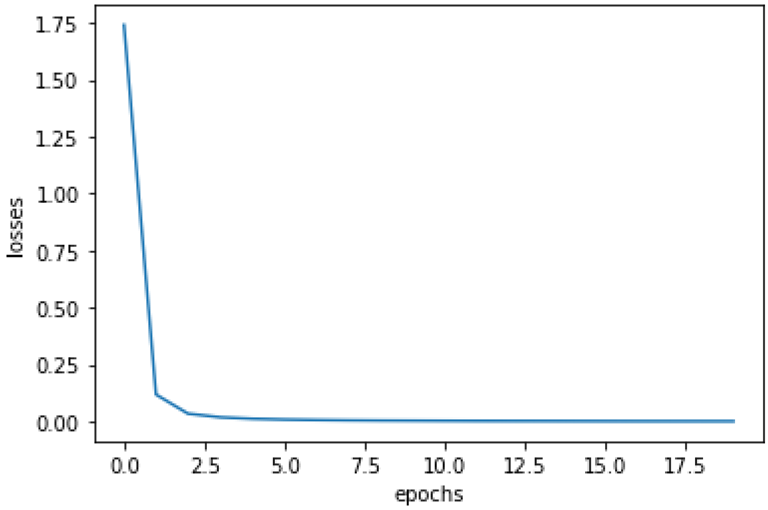
```
Starting Training!
Initializing ...
Training...
Iteration: 100; Percent complete: 5.0%; Average loss: 1.7402
Iteration: 200; Percent complete: 10.0%; Average loss: 0.1188
Iteration: 300; Percent complete: 15.0%; Average loss: 0.0345
Iteration: 400; Percent complete: 20.0%; Average loss: 0.0186
Iteration: 500; Percent complete: 25.0%; Average loss: 0.0121
Iteration: 600; Percent complete: 30.0%; Average loss: 0.0087
Iteration: 700; Percent complete: 35.0%; Average loss: 0.0067
Iteration: 800; Percent complete: 40.0%; Average loss: 0.0053
Iteration: 900; Percent complete: 45.0%; Average loss: 0.0044
Iteration: 1000; Percent complete: 50.0%; Average loss: 0.0039
Iteration: 1100; Percent complete: 55.0%; Average loss: 0.0033
Iteration: 1200; Percent complete: 60.0%; Average loss: 0.0028
Iteration: 1300; Percent complete: 65.0%; Average loss: 0.0025
Iteration: 1400; Percent complete: 70.0%; Average loss: 0.0023
Iteration: 1500; Percent complete: 75.0%; Average loss: 0.0020
Iteration: 1600; Percent complete: 80.0%; Average loss: 0.0018
Iteration: 1700; Percent complete: 85.0%; Average loss: 0.0016
Iteration: 1800; Percent complete: 90.0%; Average loss: 0.0015
Iteration: 1900; Percent complete: 95.0%; Average loss: 0.0014
Iteration: 2000; Percent complete: 100.0%; Average loss: 0.0013
```

Plot graph

In [29]:

```
import matplotlib.pyplot as plt

plt.plot([loss.detach().numpy() for loss in lossvalues])
plt.ylabel('losses')
plt.xlabel('epochs')
plt.show()
```



BLEU score calculation

Let's calculate the BLEU score with the beam search decoder:

In [321]:

```
def beam_decode(decoder, decoder_hidden, encoder_outputs, voc, beam_size, max_length=MA
X_LENGTH):
    terminal_sentences, prev_top_sentences, next_top_sentences = [], [], []
    prev_top_sentences.append(Sentence(decoder_hidden))
    for i in range(max_length):

        for sentence in prev_top_sentences:
            decoder_input = torch.LongTensor([[sentence.last_idx]])
            decoder_input = decoder_input.to(device)
            # print(decoder_input.shape)
            decoder_hidden = sentence.decoder_hidden
            decoder_output = decoder(decoder_input, encoder_outputs)
            decoder_hidden = decoder_output[-1]

            topv, topi = decoder_output.topk(beam_size)
            term, top = sentence.addTopk(topi, topv, decoder_hidden, beam_size, voc)
            terminal_sentences.extend(term)
            next_top_sentences.extend(top)

        next_top_sentences.sort(key=lambda s: s.avgScore(), reverse=True)
        prev_top_sentences = next_top_sentences[:beam_size]
        next_top_sentences = []


    terminal_sentences += [sentence.toWordScore(voc) for sentence in prev_top_sentence
s]
    terminal_sentences.sort(key=lambda x: x[1], reverse=True)

    n = min(len(terminal_sentences), 15)
    return terminal_sentences[:n]
```

In [347]:

```
class BeamSearchDecoder(nn.Module):

    def __init__(self, encoder, decoder, voc, beam_size=10):
        super(BeamSearchDecoder, self).__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.voc = voc
        self.beam_size = beam_size
        self.src_mask = None

    def forward(self, input_seq, input_length, max_length):

        if self.src_mask is None or self.src_mask.size(0) != len(input_seq):
            mask = self._generate_square_subsequent_mask(len(input_seq)).to(device)
            self.src_mask = mask
        # print(input_seq.shape)
        # print(self.src_mask.shape)
        # encoder_outputs, encoder_hidden = self.encoder(input_seq, self.src_mask)

        encoder_outputs = self.encoder(input_seq)
        print(encoder_outputs.shape)
        encoder_hidden = encoder_outputs[-1]
        print(encoder_hidden.shape)

        decoder_hidden = encoder_hidden[:self.decoder.num_layers]

        decoder_input = torch.ones(1, 1, device=device, dtype=torch.long) * SOS_token

        sentences = beam_decode(self.decoder, decoder_hidden, encoder_outputs, self.vo
c, self.beam_size, max_length)

        all_tokens = [torch.tensor(self.voc.word2index.get(w, 0)) for w in sentences[0]
[0]]
        return all_tokens, None

    def _generate_square_subsequent_mask(self, sz):
        mask = (torch.triu(torch.ones(sz, sz)) == 1).transpose(0, 1)
        mask = mask.float().masked_fill(mask == 0, float('-inf')).masked_fill(mask ==
1, float(0.0))
        return mask

    def __str__(self):
        res = f"BeamSearchDecoder with beam size {self.beam_size}"
        return res
```

In [348]:

```
def evaluate(searcher, voc, sentence, max_length=MAX_LENGTH):

    indexes_batch = [indexesFromSentence(voc, sentence)]

    lengths = torch.tensor([len(indexes) for indexes in indexes_batch])
    # print(lengths)
    input_batch = torch.LongTensor(indexes_batch).transpose(0, 1)
    # print(input_batch.shape)

    force = nn.Embedding(256,256)
    input_batch = force(input_batch)

    input_batch = input_batch.to(device)
    lengths = lengths.to(device)

    tokens, scores = searcher(input_batch, lengths, max_length)

    decoded_words = [voc.index2word[token.item()] for token in tokens]
    return decoded_words
```

In [349]:

```
# check = nn.Embedding(240,512)
# check(torch.LongTensor(6, 1).random_(0, 10)).shape
```

Out[349]:

```
torch.Size([6, 1, 512])
```

In [352]:

```
from nltk.translate.bleu_score import sentence_bleu, corpus_bleu
from nltk.translate.bleu_score import SmoothingFunction

# Set dropout layers to eval mode

# model_name.eval()
model.transformer_encoder.eval()
model.transformer_decoder.eval()

# Initialize search module

############################################################################
# Difference between greedy search and beam search is here

# greedy search
# searcher = GreedySearchDecoder(model_name.transformer_encoder, model_name.transformer_decoder)

# beam search
searcher_transformer = BeamSearchDecoder(model.transformer_encoder, model.transformer_decoder, voc, 3)
############################################################################
gram1_bleu_score = []
gram2_bleu_score = []

for i in range(0,len(testpairs),1):
    input_sentence = testpairs[i][0]
    reference = testpairs[i][1:]
    # print(f'input_sentence : {input_sentence}')
    # print(f'reference : {reference}')
    templist = []
    for k in range(len(reference)):
        if(reference[k]!=''):
            temp = reference[k].split(' ')
            templist.append(temp)

    input_sentence = normalizeString(input_sentence)
    # print(input_sentence)
    output_words = evaluate(searcher_transformer, voc, input_sentence)
    output_words[:] = [x for x in output_words if not (x == 'EOS' or x == 'PAD')]
    chencherry = SmoothingFunction()
    score1 = sentence_bleu(templist,output_words,weights=(1, 0, 0, 0) ,smoothing_function=chencherry.method1)
    score2 = sentence_bleu(templist,output_words,weights=(0.5, 0.5, 0, 0),smoothing_function=chencherry.method1)
    gram1_bleu_score.append(score1)
    gram2_bleu_score.append(score2)
    if i%1000 == 0:
        print(i,sum(gram1_bleu_score)/len(gram1_bleu_score),sum(gram2_bleu_score)/len(gram2_bleu_score))

print("Total Bleu Score for 1 grams on testing pairs: ", sum(gram1_bleu_score)/len(gram1_bleu_score))
print("Total Bleu Score for 2 grams on testing pairs: ", sum(gram2_bleu_score)/len(gram2_bleu_score))
```

Conclusion

Changing encoder/decoder from LSTM to be Transformer still slightly not difference much in term of loss because those sentences are not too long. So LSTM is already capture the meaning.