PHASE 3

CREATE A CHATBOT

USING PYTHON

Project title:	To create a Chatbot in Python that provides exceptional customer service,answering user queries (diabetes) on a website.
Name:	Nithish. E
Roll no:	411421205029
Department:	B Tech IT (3rd year)
Institute:	New prince shri bhavani college of engineering and technology

Project Details

INTRODUCTION

- **Deep Learning and Natural Language Processing (NLP) are two exciting and rapidly**
- **advancing fields within artificial intelligence (AI) and machine learning.**
- **They are at the forefront of creating intelligent systems that can understand, process, and generate human language, paving the way for applications like chatbots, language translation, sentiment analysis, and more.**
- **In the context of chatbots, RNNs can be used to create intelligent conversational agents capable of understanding and generating human-like responses in a dynamic and context-aware manner.**
- Deep Learning and NLP are driving innovations in various industries, including **healthcare, customer service, finance, and entertainment.**
- **These technologies are enabling chatbot to communicate with humans more naturally and understand the nuances of human language.**

PART I: ATTENTION UNDERSTANDING

Just like in “Attention” meaning, in real life when we looking at a picture or hearing the song, we usally focus more on some parts and pay less attention in the rest. The Attention mechanism in Deep Learning is also the same flow, paying greater attention to certain parts when processing the data

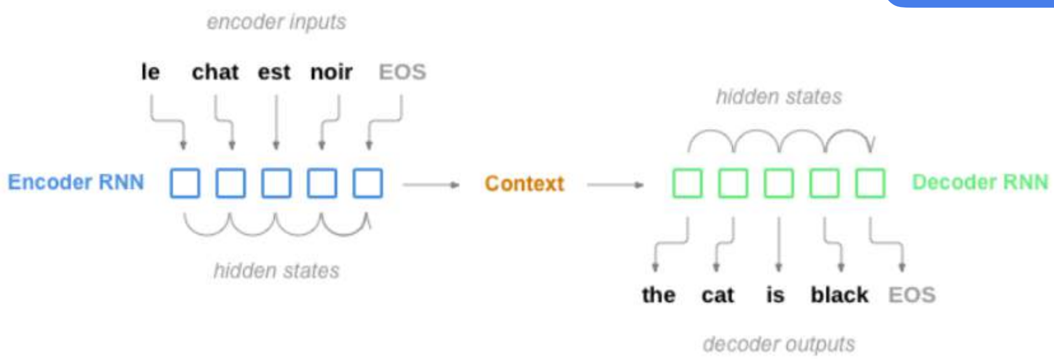
Attention is one component of a network’s architecture.

Follow the specific tasks, the encoder & decoder will be different. In machine translation, the encoder often set to LSTM/GRU/Bi_RNN, in image captioning, the encoder often set to CNN.

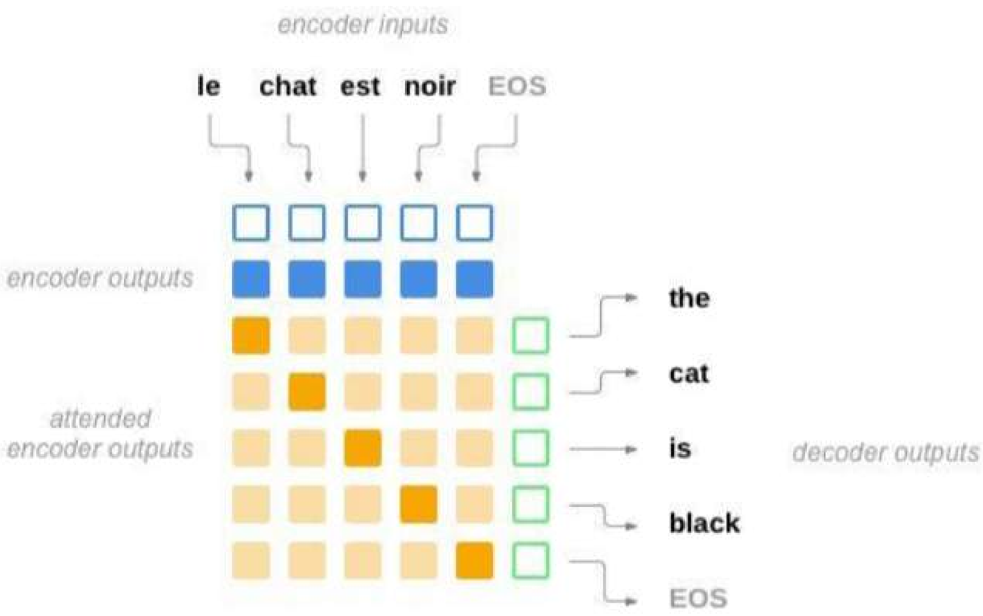
Such as for the task: Translating the sentence: 'le chat est noir' to English sentence (the cat is black)

The input has 4 words, plus EOS token at the end (stop word) corresponding 5 time steps in translating to English. Each time step, Attention is applied by assigning weights to input words, the

more important words, the bigger weights will be assigned (Done by backprob gradient process). So There are 5 differrent times weights assigned (coresponding to 5 time steps) The general architecture in seq2seq as follow:



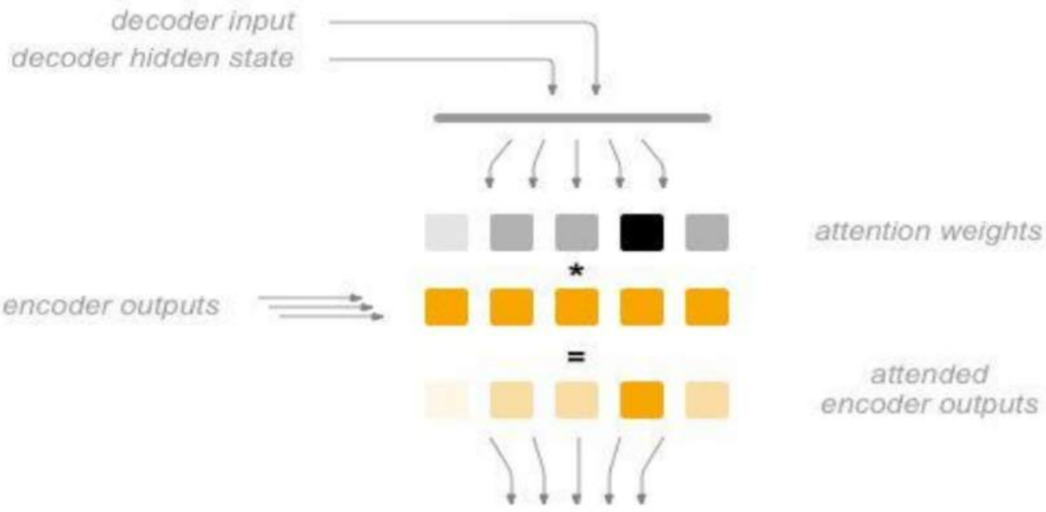
- Without attention, The input in decoder based on 2 component: the initial decoder input (often we set it to EOS token first (start word)) and the last hidden encoder.
- This way has the drawback in case some informations of very first encoder cell would be loss during the process. To handle this problem, the attention weight is added to all encoder outputs



As we can see, through each decoder output word, the attention weights colors of encoder input is changed differently along itself importance

- you may ask how can we appropriately set the weight to encoder outputs. The answer is:

we just randomly set the weights, and the backpropagation gradient process will take care about it during the training. What we have to do is correctly build the forward computational Graph.



Example:

1. import torch
2. import torch.nn as nn

STEP 1: CACULATING ENCODER HIDDEN STATE

```
class Encoder_LSTM(nn.Module):

 def __init__(self, input_size, hidden_size, n_layers=1, drop_prob=0):

 super(EncoderLSTM, self).__init__()

 self.hidden_size = hidden_size

 self.n_layers = n_layers

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

 self.lstm = nn.LSTM(hidden_size, hidden_size, n_layers, dropout=drop_prob, batch_first=True)

 def forward(self, inputs, hidden):

 # Embed input words

 embedded = self.embedding(inputs)

 # Pass the embedded word vectors into LSTM and return all outputs

 output, hidden = self.lstm(embedded, hidden)

 return output, hidden
```

Step 2--->6

```
class Luong_Decoder(nn.Module):
```

```
def __init__(self, hidden_size, output_size, attention, n_layers=1, drop_prob=0.1):

super(LuongDecoder, self).__init__()

self.hidden_size = hidden_size

self.output_size = output_size

self.n_layers = n_layers

self.drop_prob = drop_prob

# The Attention layer is defined in a separate class

self.attention = attention

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

self.dropout = nn.Dropout(self.drop_prob)

self.lstm = nn.LSTM(self.hidden_size, self.hidden_size)

self.classifier = nn.Linear(self.hidden_size*2, self.output_size)

def forward(self, inputs, hidden, encoder_outputs):

# Embed input words

embedded = self.embedding(inputs).view(1,1,-1)

embedded = self.dropout(embedded)
```

STEP 2: GENERATE NEW HIDDEN STATE FOR DECODER

```
lstm_out, hidden = self.lstm(embedded, hidden)
```

STEP 3: CALCULATING ALIGNMENT SCORES

```
alignment_scores = self.attention(lstm_out,encoder_outputs)
```

STEP 4: SOFTMAXING ALIGNMENT SCORES TO OBTAIN ATTENTION WEIGHTS

```
attn_weights = F.softmax(alignment_scores.view(1,-1), dim=1)
```

STEP 5: CACULATING CONTEXT VECTOR

```
context_vector = torch.bmm(attn_weights.unsqueeze(0),encoder_outputs)
```

STEP 6: CACULATING THE FINAL DECODER OUTPUT

```
output = torch.cat((lstm_out, context_vector),-1)
```

Pass concatenated vector through Linear layer acting as a Classifier

output = F.log_softmax(self.classifier(output[0]), dim=1)

return output, hidden, attn_weights

Exploring the attention class in STEP 3: Caculating alignment score

In Luong Attention, there are 3 different ways (dot, general, concat) to caculate the alignment scores.

1. Dot function

This is the simplest of the functions: alignment score calculated by multiplying the hidden encoder and

the hidden decoder.

SCORE = H(encoder) * H(decoder)

2. General function

similar to the dot function, except that a weight matrix is added into the equation

SCORE = W(H(encoder) * H(decoder))

3. Concat function

Concating encoder and decoder first, the feed to nn.Linear and activation it, finally we add W2 to get final

Score

SCORE = W2 * tanh(W1(H(encoder) + H(decoder)))

Implementing attention class:

class Luong_attention_layer(nn.Module):

def __init__(self, method, hidden_size):

super(Luong_attention_layer, self).__init__()

self.method = method

self.hidden_size = hidden_size

if self.method not in ['dot', 'general', 'concat']:

```
raise ValueError(self.method, 'is not appropriate attention method')

if self.method == 'general':

self.attn = torch.nn.Linear(self.hidden_size, hidden_size)

elif self.method == 'concat':

self.attn = torch.nn.Linear(self.hidden_size * 2, hidden_size)

self.weight = nn.Parameter(torch.FloatTensor(hidden_size))

def get_dot_score(self, hidden, encoder_outputs):

return torch.sum(hidden*encoder_outputs, dim=2)

def get_general_score(self, hidden, encoder_outputs):

energy = self.attn(encoder_outputs)

return torch.sum(hidden * energy, dim=2)

def get_concat_score(self, hidden, encoder_outputs):

concat = torch.cat((hidden.expand(encoder_outputs.size(0),-1,-1), encoder_outputs), dim=2)

energy = torch.tanh(self.attn(concat))

return torch.sum(self.weight * energy, dim=2)

def forward(self, hidden, encoder_outputs):

if self.method == 'dot':

attn_energy = self.get_dot_score(hidden, encoder_outputs)

elif self.method == 'general':

attn_energy = self.get_general_score(hidden, encoder_outputs)

elif self.method == 'concat':

attn_energy = self.get_concat_score(hidden, encoder_outputs)

## Transpose attn_energy

attn_energy = attn_energy.t()

# Softmanx the attn_energy to return the weight corresponding to each encoder output

return F.softmax(attn_energy, dim=1).unsqueeze(1)
```

Part II: Building chatbot seq2seq with Luong attention mechanism

The step by step for building chatbot with attention as follow:Capture%204.JPG

After running this kernel. you can play with chatbot and have some fun with him like this:))

:Capture6.JPG

The code is based on : https://pytorch.org/tutorials/beginner/chatbot_tutorial.html. I have modified

this toturial on something because the Author used some pytorch features that currently depressed.

Through this kernel, I added explaination on my own understanding step by step so you might find it

friendly to understand all the concepts.

- **Step 1: Preparing data**

from __future__ import absolute_import

from __future__ import division

from __future__ import print_function

from __future__ import unicode_literals

import numpy as np

import os

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

%matplotlib inline

use_cuda = torch.cuda.is_available()

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

device

corpus_name = 'cornell-moviedialog-corpus'

corpus = os.path.join('/kaggle/input', corpus_name)

def printLines(filename, n=10):

 with open(filename, 'rb') as f:

 lines = f.readlines()

 for line in lines[:n]:

 print(line)

printLines(os.path.join(corpus,'movie_lines.txt'))

column_names = ["lineID","characterID","movieID","character","text"]

def LoadLines(file, column_names):

 lines = {}

 with open(file, 'r', encoding='iso-8859-1') as f:

 for line in f:

 dict = {}

 list_field = line.split(' +++$+++ ')

 for i, field in enumerate(list_field):

 dict[column_names[i]] = field

 lines[dict['lineID']] = dict

 return lines
```

lines = LoadLines(os.path.join(corpus, 'movie_lines.txt'), column_names)

as we can see, after split the "utteranceIDs" is a string : " ['L2460', 'L2461', 'L2462']\n", what we want is

retrieve the list inside the string,

to do this we use eva function that do the expression inside the input

In the 'movie_conversations.txt', the columns are: ["character1ID", "character2ID", "movieID",

"utteranceIDs"]

def Loadconversation(file, lines, column_names):

conversation = []

with open(file, 'r', encoding='iso-8859-1') as f:

for line in f:

dict_column = {}

list_column = line.split(' +++$+++ ')

for i, col in enumerate(list_column):

dict_column[column_names[i]] = col

line_id_list = eval(dict_column['utteranceIDs'])

dict_column['lines'] = []

for line in line_id_list:

dict_column['lines'].append(lines[line])

conversation.append(dict_column)

return conversation

conversations = Loadconversation(os.path.join(corpus, 'movie_conversations.txt'),lines, ["character1ID",

"character2ID", "movieID", "utteranceIDs"])

def get_pair_conversation(conversations):

"""

return list of pair conversation [[input1, response1], [input2, response2],....]

```
 """
 pair = []
 for conversation in conversations:
 num_sentence = len(conversation['lines'])
 for i in range(num_sentence-1):
 input = conversation['lines'][i]['text'].strip()
 response = conversation['lines'][i+1]['text'].strip()
 if input and response:
 pair.append([input, response])
 return pair
# create new file to overwrite into it
os.chdir('/kaggle/')
os.getcwd()
if not os.path.exists('data_save'):
 os.makedirs('data_save')
os.chdir('data_save')
path_save = '/kaggle/data_save'
datafile = os.path.join(path_save, "formatted_movie_lines.txt")
delimiter = '\t'
# Unescape the delimiter
delimiter = str(codecs.decode(delimiter, "unicode_escape"))
print("\nWriting newly formatted file...")
with open(datafile, 'w', encoding='utf-8') as outputfile:
 writer = csv.writer(outputfile, delimiter=delimiter, lineterminator='\n')
 for pair in get_pair_conversation(conversations):
 writer.writerow(pair)
```

For this we define a Voc class, which keeps a mapping from words to indexes, a reverse mapping of

indexes to words, a count of each word and a total word count. The class provides methods for adding a

word to the vocabulary (addWord), adding all words in a sentence (addSentence) and trimming

infrequently seen words (trim). More on trimming later.

```
pad_token = 0

sos_token = 1

eos_token = 2

class Voc:

 def __init__(self, name):

 self.name = name

 self.trimmed = False

 self.word2index = {}

 self.word2count = {}

 self.index2word = {pad_token:'PAD', sos_token:'SOS', eos_token : 'EOS'}

 self.numword = 3

 def add_sentence(self, sentence):

 for word in sentence.split(' '):

 self.addword(word)

 def addword(self, word):

 if word not in self.word2index:

 self.word2index[word] = self.numword

self.word2count[word] = 1

 self.index2word[self.numword] = word

 self.numword += 1

 else:
```

```
self.word2count[word] += 1

def trim(self, min_count):

"""

based on the wordcount dictionary, Filter of the word frequency at least more than
min_count

"""

if self.trimmed:

return

self.trimmed = True

Keep_word = []

for word, num_frequency in self.word2count.items():

if num_frequency >= min_count:

keep_word.append(word)


# reinitialize dictionaries

self.word2index = {}

self.word2count = {}

self.index2word = {pad_token:'PAD', sos_token:'SOS', eos_token : 'EOS'}

self.numword = 3

for word in keep_word:

self.addword(word)

# Convert (or remove accents) sentence to non_accents sentence

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

lines = open(datafile, encoding='utf-8').\

 read().strip().split('\n')

lines[0] ## Each string in lines list is a pair (input, response)

def readVocs(datafile, corpus_name):

 lines = open(datafile, 'r', encoding='utf-8').\

 read().strip().split('\n')

 pairs = [[normalizeString(s) for s in l.split('\t')] for l in lines]

 voc = Voc(corpus_name)

 return voc, pairs

## we ensure every sentences must have the length smaller than max_length

## max_length value is based on our choice, the greater value, the more data training we have and also

the more parameter the model have to train on

def filterpair(pairs, max_length):

 """

 Input: pair with format: [input, response] such as: ['how are you', 'I am ok']

 we check the length of both input, response to identify where or not they smaller than max_length

 return pair with length < max_length

 """
```

```
valid_pair = []
for pair in pairs:
input_words, response_words = pair[0].split(' '), pair[1].split(' ')
if len(input_words) < max_length and len(response_words) < max_length:
valid_pair.append(pair)
return valid_pair
def loadPrepareData(datafile, corpus_name, max_length):
voc, pairs = readVocs(datafile, corpus_name)
valid_pair = filterpair(pairs, max_length)
print(f'load total {len(pairs)} pairs')
print(f'load total {len(valid_pair)} pairs with length <= max_length (10)')
for pair in valid_pair:
voc.add_sentence(pair[0])
voc.add_sentence(pair[1])
print(f'total word in vocabulary is : {voc.numword}')
return voc, valid_pair
voc, valid_pair = loadPrepareData(datafile, corpus_name, max_length = 10)
print('examples of 10 first pairs')
for pair in valid_pair[:3]:
print(pair)
```

In the vocabulary pairs, it's include some rare words and this make model difficult to convergance

because it try hard to approximate in output predict and real output when one of them they include rare

word. make the rest hard to approximate ==> take out these word from pairs

```
def trim_rareword(voc, pairs, min_count):
voc.trim(min_count) ## trim the voc class with min_count word so that every word in voc.word2index
```

will satisfied the min_count frequency requirement

trimmed_pair = []

for pair in pairs:

input_sentence = pair[0]

response_sentence = pair[1]

keep_input = True

keep_response = True

Loop over every word in both input and response sentence

Loop over input sentence

for word in input_sentence.split(' '):

if word not in voc.word2index: # condition

keep_input = False

break ## it will end the process right away as long as meet condition, the rest loop process will

not run anymore

Loop over output sentence

for word in response_sentence.split(' '):

if word not in voc.word2index: # condition

keep_input = False

break

if keep_input and keep_response:

trimmed_pair.append(pair)

print(f'the trimming process make the total {len(pairs)} ==> {len(trimmed_pair)} trimmed pair)')

return voc,trimmed_pair

voc, trimmed_pair = trim_rareword(voc, valid_pair, min_count=3)

Transform data to tensor

```
def index_from_sentence(voc, sentence):
 """
 Input: a single sentence
 output: return index respectively matching with words in sentence based on voc.word2index
 """
 return [voc.word2index[word] for word in sentence.split(' ')] + [eos_token] ## to indicate that the
sentence is ended here
# def indexesFromSentence(voc, sentence):
# return [voc.word2index[word] for word in sentence.split(' ')] + [eos_token]
index_from_sentence(voc, trimmed_pair[5][0])
# Python's Itertool is a module that provides various functions that work on iterators (list, tuple, string,...)
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
def input_to_torch(l, voc):
 """
```

```
 Purpos: convert to torch.tensor , (Returns padded input sequence tensor and lengths)

 """

 indexes_batch = [index_from_sentence(voc, sentence) for sentence in l]

 padded_list_index = zeroPadding(indexes_batch)

 padded_tensor_index = torch.LongTensor(padded_list_index)

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

 return padded_tensor_index, lengths

def output_to_torch(l, voc):

 """

 Purpos: convert to torch.tensor , (Returns padded output sequence tensor, mask tensor,
max_lengths)

 """

 indexes_batch = [index_from_sentence(voc, sentence) for sentence in l]

 padded_list_index = zeroPadding(indexes_batch)

 padded_tensor_index = torch.LongTensor(padded_list_index)

 max_output_length = max([len(indexes) for indexes in indexes_batch])

 mask = binaryMatrix(padded_list_index)

 mask = torch.ByteTensor(mask)

 return padded_tensor_index, mask, max_output_length

## Combine all and return all items needed given a batch of pairs

def get_batch_pair(voc, batch_pair):

 """

 sort the len of input sentence in desc

 return all input and output items

 """

 # sort len(input sentence) in batch_pair with decreasing order

 batch_pair.sort(key = lambda x: len(x[0].split(" ")), reverse = True )
```

devide the batch pair to batch_input and batch_response

input_batch, response_batch = [], []

for pair in batch_pair:

input_batch.append(pair[0])

response_batch.append(pair[1])

input_tensor, length_input = input_to_torch(input_batch, voc)

output_tensor, mask, max_length = output_to_torch(response_batch, voc)

return input_tensor, length_input, output_tensor, mask, max_length

- **Step 2: Define model**

Things to remember : output_size : (seq_len, batch, num_directions * hidden_size), num_directions = 1

if unidirectional and 2 if bidirectional

class EncoderRNN(nn.Module):

def __init__(self, embedding, hidden_size, num_layers = 1,dropout = 0):

super(EncoderRNN, self).__init__()

self.num_layers = num_layers

self.embedding = embedding

self.hidden_size = hidden_size

Define GRU layers, this GRU cell return 2 things: Output and hidden_state cell

self.gru = nn.GRU(input_size = hidden_size ## in input_size, number of features = hidden_size

, hidden_size = hidden_size

, num_layers = num_layers

, dropout = (0 if num_layers == 1 else dropout)

, bidirectional = True)

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

Convert input seq to embedding format

embedding = self.embedding(input_seq)

```
packed_input = torch.nn.utils.rnn.pack_padded_sequence(embedding, input_length)
## forward to gru cell
output, hidden_cell = self.gru(packed_input, hidden)
output, _ = torch.nn.utils.rnn.pad_packed_sequence(output)
## Sum bidirectional GRU output
output = output[:,:,:self.hidden_size] + output[:,:,self.hidden_size:]
return output, hidden_cell
class Luong_attention_layer(nn.Module):
def __init__(self, method, hidden_size):
super(Luong_attention_layer, self).__init__()
self.method = method
self.hidden_size = hidden_size
if self.method not in ['dot', 'general', 'concat']:
raise ValueError(self.method, 'is not appropriate attention method')
if self.method == 'general':
self.attn = torch.nn.Linear(self.hidden_size, hidden_size)
elif self.method == 'concat':
self.attn = torch.nn.Linear(self.hidden_size * 2, hidden_size)
self.weight = nn.Parameter(torch.FloatTensor(hidden_size))
def get_dot_score(self, hidden, encoder_outputs):
return torch.sum(hidden*encoder_outputs, dim=2)
def get_general_score(self, hidden, encoder_outputs):
energy = self.attn(encoder_outputs)
return torch.sum(hidden * energy, dim=2)
def get_concat_score(self, hidden, encoder_outputs):
concat = torch.cat((hidden.expand(encoder_outputs.size(0),-1,-1), encoder_outputs), dim=2)
energy = torch.tanh(self.attn(concat))
```

```
return torch.sum(self.weight * energy, dim=2)

def forward(self, hidden, encoder_outputs):

if self.method == 'dot':

attn_energy = self.get_dot_score(hidden, encoder_outputs)

elif self.method == 'general':

attn_energy = self.get_general_score(hidden, encoder_outputs)

elif self.method == 'concat':

attn_energy = self.get_concat_score(hidden, encoder_outputs)

## Transpose attn_energy

attn_energy = attn_energy.t()

# Softmanx the attn_energy to return the weight corresponding to each encoder output

return F.softmax(attn_energy, dim=1).unsqueeze(1)

class Luong_attention_decoder(nn.Module):

def __init__(self, embedding, attn_model, hidden_size, output_size, n_layers=1, dropout = 0.1):

super(Luong_attention_decoder, self).__init__()

## Define properties for self

self.hidden_size = hidden_size

self.output_size = output_size

self.n_layers = n_layers

self.dropout = dropout

self.attn_model = attn_model

## Define 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 for transform the concat tensor size [hidden,encoder_output] with size =
```

```
(hidden_size*2) ==> (hidden_size)

self.concat = nn.Linear(hidden_size*2, hidden_size)

## self.out for Dense the gru_ouput to return predict value

self.out = nn.Linear(hidden_size, output_size)

self.attention = Luong_attention_layer(attn_model, hidden_size)

def forward(self, input_step, last_hidden, encoder_outputs):

## One step one word through batch

embedded = self.embedding(input_step)

embedded = self.embedding_dropout(embedded)

# forward through unidirrectional GRU

rnn_output, hidden = self.gru(embedded, last_hidden)

# Feed output and encoder_outputs to attention layer

attention_weights = self.attention(rnn_output, encoder_outputs)

# caculate context vector

context = attention_weights.bmm(encoder_outputs.transpose(0,1))

# concat context vector with 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))

# return output predict

output = self.out(concat_output)

output = F.softmax(output, dim=1)

return output, hidden
```

Understand torch.gather

https://stackoverflow.com/questions/50999977/what-does-the-gather-function-do-in-pytorch-in-laymanterms in torch.gather(input, dim = (0 or 1 or 2), index)

if dim = 0, we go through rows, from top to bottom,

if dim = 1, we go through columns, left to right

def maskNLLLoss(input, target, mask):

nTotal = mask.sum()

crossEntropy = -torch.log(torch.gather(input, 1, target.view(-1, 1)).squeeze(1))

loss = crossEntropy.masked_select(mask).mean()

loss = loss.to(device)

return loss, nTotal.item()

- **Step 3: Creating training function**

np.random.seed(42)

max_length = 10

def train(input_variable, lengths, target_variable, embedding, encoder, decoder, encoder_optimizer,

decoder_optimizer, max_target_lens

, batch_size, clip, mask,max_length = max_length):

""""

this train function is responsible for one iteration

""""

Zeros gradients

encoder_optimizer.zero_grad()

decoder_optimizer.zero_grad()

Set device

input_variable = input_variable.to(device)

target_variable = target_variable.to(device)

lengths = lengths.to(device)

mask = mask.bool()

```
mask = mask.to(device)
## Initialize variable
loss = 0
print_loss = []
n_totals = 0
## Pass input through encoder
output_encoders, hidden_encoders = encoder(input_variable, lengths)
## Create initial hidden input
input_decoders = torch.LongTensor([[sos_token for _ in range(batch_size)]])
input_decoders = input_decoders.to(device)
## Set initial decoder hidden
hidden_decoders = hidden_encoders[:decoder.n_layers]
## Determine to use teacher forcing or not
teacher_forcing = True if random.random() < teacher_forcing_rate else False
if teacher_forcing:
for t in range(max_target_lens):
output_decoders, hidden_decoders = decoder(input_decoders, hidden_decoders,
output_encoders)
# in case teacher forcing, current target is set to next decoder input
input_decoders = target_variable[t].view(1, -1)
# Caculate loss
mask_loss, nTotal = maskNLLLoss(output_decoders, target_variable[t], mask[t])
loss+=mask_loss # the most important is loss function, this is place where all gradients will be
calculated
print_loss.append(mask_loss.item() * nTotal)
n_totals += nTotal
else:
```

for t in range(max_target_lens):

output_decoders, hidden_encoders = decoder(input_decoders, hidden_decoders,

output_encoders)

in case None teacher forcing, current output decoder is set to next decoder input

torch.topk(i) return (value,index of that value) of "i" highest values of tensor, in this case,we

want return the only

(___, index) with highest probability value, so we set i ==> 1

_, topi = output_decoders.topk(1) ## output_decoder is tensor softmax: ex: [0.3,0.6,01], topk(1)

meaning return one highest value

input_decoders = torch.LongTensor([[topi[i][0] for i in range(batch_size)]])

input_decoders = input_decoders.to(device) ## because decoder_input in this case is newly

created and have to switch to device

Caculate loss

mask_loss, nTotal = maskNLLLoss(output_decoders, target_variable[t], mask[t])

loss += mask_loss

print_loss.append(mask_loss.item() * nTotal)

n_totals += nTotal

Backprob gradient in loss function

loss.backward()

Clip the gradients in both encoder, decoder

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

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

Calling the step function on an Optimizer makes an update to its parameters

encoder_optimizer.step()

decoder_optimizer.step()

return average loss

```
 return sum(print_loss) / n_totals

def trainIters(model_name, voc, trimmed_pair, 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 batch for each iteration

 training_batches = [get_batch_pair(voc, [random.choice(trimmed_pair) for _ in
range(batch_size)]) for

_ in range(n_iteration)]

# Initialization

 print('initializing...')

 start_iteration = 1

 print_loss = 0

 if loadFilename:

 start_iteration = checkpoint['iteration'] + 1


 # Training loop

 print('tranining')

 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_lens = training_batch

 # training on batch

 loss = train(input_variable, lengths, target_variable, embedding, encoder, decoder,

encoder_optimizer, decoder_optimizer, max_target_lens

 , batch_size, clip, mask)
```

```
print_loss += loss
# Print loss after "print_every step"
if (iteration % print_every) == 0:
print_loss_avg = print_loss / print_every
print(f'loss_avg at {iteration} is: {print_loss_avg}, in {100 * iteration / n_iteration } % progress
complete')
print_loss = 0
# Save checkpoint
if (iteration % save_every) == 0:
directory = os.path.join(path_save, model_name, corpus_name, f'{encoder_n_layers}-
{decoder_n_layers}_{hidden_size}')
if not os.path.exists(directory):
os.makedirs(directory)
torch.save({
'iteration': iteration,
'encoder' : encoder.state_dict(),
'decoder' : decoder.state_dict(),
'encoder_optimizer': encoder_optimizer.state_dict(),
'decoder_optimizer': decoder_optimizer.state_dict(),
'loss' : loss,
'voc_dict' : voc.__dict__,
'embedding': embedding.state_dict()
}, os.path.join(directory, '{}_{}.tar'.format(iteration, 'checkpoint')))
```

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:

Forward input through encoder model.

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

Initialize decoder's first input as SOS_token.

Initialize tensors to append decoded words to.

Iteratively decode one word token at a time:

Forward pass through decoder.

Obtain most likely word token and its softmax score.

Record token and score.

Prepare current token to be next decoder input.

Return collections of word tokens and scores.

- **Step 4: Create function to interact with chatbot**

```
class Greedysearch_decoder(nn.Module):
 def __init__(self, encoder, decoder):
super(Greedysearch_decoder, self).__init__()
 self.encoder = encoder
 self.decoder = decoder


def forward(self, input_seq, input_length, max_length):
 output_encoder, hidden_encoder = self.encoder(input_seq, input_length)
 # Set the final hidden encoder to be initial hidden decoder
 hidden_decoder = hidden_encoder[:decoder.n_layers]
 # Initialize decoder input with sos_token
 input_decoder = torch.ones(1,1,device = device, dtype = torch.long) * sos_token
```

```
# Create tensors to contain output word
all_tokens = torch.zeros([0], device=device, dtype = torch.long)
all_score = torch.zeros([0], device=device)
# Loop over decoder - one word per time step
for _ in range(max_length):
output_decoder, hidden_decoder = self.decoder(input_decoder, hidden_decoder,
output_encoder)
# Feed output_decoder to torch.max() to return (max_value, index) ( softmax)
max_score, output_index = torch.max(output_decoder, dim = 1)
# Append to all_tokens and all_scores
all_tokens = torch.cat((all_tokens, output_index), dim = 0)
all_score = torch.cat((all_score, max_score), dim = 0)
# Set current output_index to the next input decoder
input_decoder = torch.unsqueeze(output_index, 0)
# Return collections of words token and score
return all_tokens, all_score
```

Evaluate our own sentence:

```
def evaluate(encoder, decoder, searcher, voc, sentence, max_length = max_length):
# transform word to index
index_sentence_list = [index_from_sentence(voc, sentence)]
input_lengths = torch.tensor([len(index) for index in index_sentence_list])
# transform index list to tensor
index_sentence = torch.LongTensor(index_sentence_list)
# Now index_sentence is [[idx1, idx2,...]], what we want is [[idx1], as we defince our sentence
shape before ( here batchsize = 1)
```

```
# [idx2],
# [...]]
# Transform index_sentence to shape (n_words, 1) to act as input
input_batch = index_sentence.transpose(0,1)
# Feed to device
input_batch = input_batch.to(device)
input_lengths = input_lengths.to(device)
# Now we pass index_sentence, lengths through encoder to return output, hidden encoder
output_tokens, output_scores = searcher(input_batch, input_lengths, max_length)
words_decoder = [voc.index2word[index.item()] for index in output_tokens]
return words_decoder
def Loop_evaluate(encoder, decoder, search, voc):
""""
This function take input sentence from your keyboard,
loop through evaluate function above util it reach 'q' or 'quit' input, the process will end here
""""
input_sentence = ''
while True:
try:
input_sentence = input('Me: ')
if input_sentence in ['q','quit']: break
# normalize string
input_sentence = normalizeString(input_sentence)
# feed to evaluate to return words
words_decoder = evaluate(encoder, decoder, search, voc, input_sentence)
words_decoder[:] = [word for word in words_decoder if word not in ['PAD','EOS']]
print('Bot: ', ' '.join(words_decoder))
```

```
 except KeyError:
 print('Unknown word in memory, please try another word')
```

Run our model

```
# Configure models
model_name = 'cb_model'
attn_model = 'concat'
#attn_model = 'general'
#attn_model = 'concat'
hidden_size = 500
encoder_n_layers = 3
decoder_n_layers = 3
dropout = 0.1
batch_size = 64
# Set checkpoint to load from; set to None if starting from scratch
loadFilename = None
checkpoint_iter = 10000
#loadFilename = os.path.join(save_dir, model_name, corpus_name,
# '{}-{}_{}'.format(encoder_n_layers, decoder_n_layers, hidden_size),
# '{}_checkpoint.tar'.format(checkpoint_iter))
# Load model if a loadFilename is provided
if loadFilename:
 # If loading on same machine the model was trained on
 checkpoint = torch.load(loadFilename)
 # If loading a model trained on GPU to CPU
 #checkpoint = torch.load(loadFilename, map_location=torch.device('cpu'))
 encoder_sd = checkpoint['encoder']
 decoder_sd = checkpoint['decoder']
```

encoder_optimizer_sd = checkpoint['encoder_optimizer']

decoder_optimizer_sd = checkpoint['decoder_optimizer']

embedding_sd = checkpoint['embedding']

voc.__dict__ = checkpoint['voc_dict']

print('Building encoder and decoder ...')

Initialize word embeddings

embedding = nn.Embedding(voc.numword, hidden_size)

if loadFilename:

embedding.load_state_dict(embedding_sd)

Initialize encoder & decoder models

encoder = EncoderRNN(embedding, hidden_size, encoder_n_layers, dropout)

decoder = Luong_attention_decoder(embedding, attn_model, hidden_size, voc.numword,

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!')

clip = 50.0

teacher_forcing_rate = 1.0

learning_rate = 3e-4

decoder_learning_rate = 5.0

n_iteration = 10000

print_every = 1000

save_every = 500

Ensure dropout layers are in train mode

encoder.train()

decoder.train()

Initialize optimizers

print('Building optimizers ...')

encoder_optimizer = optim.Adam(encoder.parameters(), lr=learning_rate)

decoder_optimizer = optim.Adam(decoder.parameters(), lr=learning_rate * decoder_learning_rate)

if loadFilename:

ecoder_optimizer.load_state_dict(encoder_optimizer_sd)

ecoder_optimizer.load_state_dict(decoder_optimizer_sd)

Run training iterations

print("Starting Training!")

trainIters(model_name, voc, trimmed_pair, encoder, decoder, encoder_optimizer, decoder_optimizer,

embedding, encoder_n_layers, decoder_n_layers, path_save, n_iteration, batch_size,

print_every, save_every, clip, corpus_name, loadFilename)

Play with chatbot:

Set dropout layers to eval mode

encoder.eval()

decoder.eval()

Initialize search module

searcher = Greedysearch_decoder(encoder, decoder)

Begin chatting, we type some sentence and play with chatbot

Loop_evaluate(encoder, decoder, searcher, voc)

CONCLUSION

In the journey of developing a Chatbot for Diabetes Analysis, we have explored the convergence of technology and healthcare to address a pressing need for improved diabetes care. This project has emphasized the importance of leveraging Natural Language Processing (NLP), Python, and associated packages to create an intelligent, accessible, and personalized solution.

In an age where healthcare information and support are critical, the Diabetes Chatbot stands as a valuable companion on your journey towards understanding, managing, and living well with diabetes.

This intelligent conversational agent has been designed to offer information, answer questions, and pr ovide guidance to individuals seeking clarity on diabetes-related matters.

Throughout your interaction with our chatbot, you've had the opportunity to explore essential aspects of diabetes, from the basics of the condition to strategies for effective management.

You've received insights into nutrition, exercise, medication, and the prevention of complications. The chatbot has served as a 24/7 resource, ready to address your inquiries and concerns. The Diabetes Chatbot remains committed to being a trustworthy resource that complements your healthcare journey.

It aims to empower you with knowledge and encourage healthier choices while fostering a sense of community and support. Our chatbot is available to assist you whenever you need guidance or simply wish to learn more about diabetes.