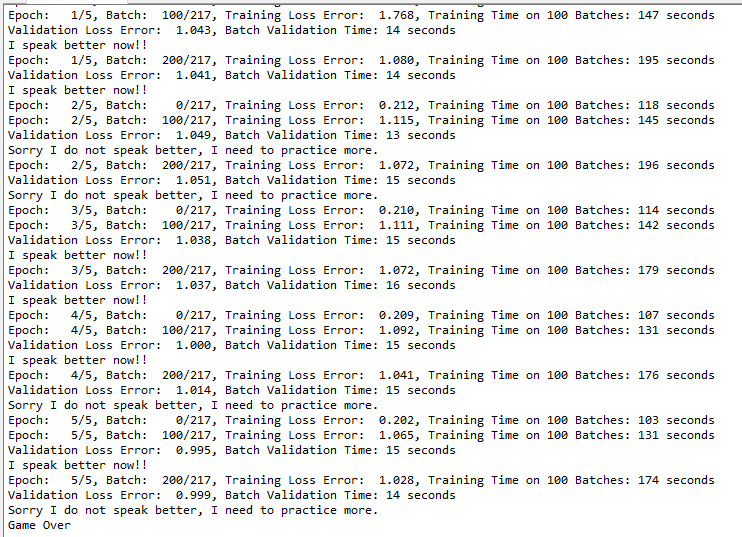
Chapter 7

**ChatBot:**

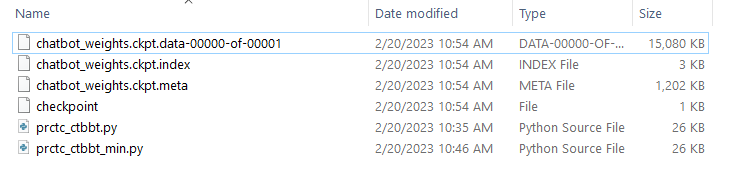
**Testing the Seq2Seq model**

Testing the model during training



**7.1 Checkpoint files**

During training process, our model creates "checkpoint" files. There it stores its trained data (weights). It updates this file if the "Validation Loss Error" reduced.



* To chat with the Chatbot, we need to load these files.

**7.2 Loading the weights and Running the session**

# *Loading the weights and Running the session*

checkpoint = "./chatbot\_weights.ckpt"

session = **tf.InteractiveSession**()

**session.run**(**tf.global\_variables\_initializer**())

saver = **tf.train.Saver**()

**saver.restore**(session, checkpoint)

* checkpoint = "./chatbot\_weights.ckpt" loads the Checkpoint file. Contains the path to these weights. These are not the weights of some very smart brain, because out chatbot isn't fully trained.
* session = tf.InteractiveSession() Starts the interactive session. Which we already used. (It was actually in the training).
* **session.run**(**tf.global\_variables\_initializer**()) we take our session object and then use **run()** method from this object to initialize the global variables .
* We will connect these loaded weights to the session using **tf.train.Saver**().

saver = **tf.train.Saver**()

**saver.restore**(session, checkpoint)

* We connect the session to the checkpoint through the **saver.restore()**. **restore()** is like a load function.

**7.3 Converting user's input (inputted string by user) into encoding integers**

Following function will convert the questions that are right now in strings to list of encoded integers.

# *Converting the questions from strings to lists of encoding integers*

**def** **convert\_string2int**(question, word2int):

    question = **clean\_text**(question)

**return** [**word2int.get**(word, word2int['<OUT>']) **for** word **in** **question.split**()]

* Take two arguments the first one is the **question** and the second one is **word2int** dictionary. So it can either be the *questions word to Integer* dictionary or the *answers words to Integer* dictionary.
* We used our defined **clean\_text()** function from the data-preprocessing part. It cleans the "user inputted string".
* Here question is the input-string from the user.
* Following encodes the string into integers:

**return** [**word2int.get**(word, word2int['<OUT>']) **for** word **in** **question.split**()]

* Returns a list of encoded integers by loopping over the list **question.split**().

**word2int.get**(word, word2int['<OUT>'])

* Notice, we used **get**(word, word2int['<OUT>']) instead of using the word directly.
* Why: Because we get rid of some *non frequent words* and if the "user inputted string" contains some *non frequent words*, we return word2int['<OUT>'] from the word2int dictionary. [Remember, we set threshold = 20, we took the word that appears more than 20 times].

**7.4 Setting up the chat**

#*######### PART 4 - TESTING THE SEQ2SEQ MODEL ##########*

# *Loading the weights and Running the session*

checkpoint = "./chatbot\_weights.ckpt"

session = **tf.InteractiveSession**()

**session.run**(**tf.global\_variables\_initializer**())

saver = **tf.train.Saver**()

**saver.restore**(session, checkpoint)

# *Converting the questions from strings to lists of encoding integers*

**def** **convert\_string2int**(question, word2int):

    question = **clean\_text**(question)

**return** [**word2int.get**(word, word2int['<OUT>']) **for** word **in** **question.split**()]

# *Setting up the chat*

**while**(**True**):

    question = **input**("You: ")

**if** question **==** 'Goodbye':

**break**

    question = **convert\_string2int**(question, questnWrd2Int)

    question = question + [questnWrd2Int['<PAD>']] \* (20 - **len**(question))

    fake\_batch = **np.zeros**((batch\_size, 20))

    fake\_batch[0] = question

    predicted\_answer = **session.run**(test\_predictions, {inputs: fake\_batch, keep\_prob: 0.5})[0]

    answer = ''

**for** i **in** **np.argmax**(predicted\_answer, 1):

**if** ans\_Int\_2\_Wrd[i] **==** 'i':

            token = ' I'

**elif** ans\_Int\_2\_Wrd[i] **==** '<EOS>':

            token = '.'

**elif** ans\_Int\_2\_Wrd[i] **==** '<OUT>':

            token = 'out'

**else**:

            token = ' ' + ans\_Int\_2\_Wrd[i]

        answer += token

**if** token **==** '.':

**break**

**print**('ChatBot: ' + answer)

* Here we used our inverse dictionary "Integer to Word": **ans\_Int\_2\_Wrd**. To convert Chatbot's response to human readable text format.
* We use a while loop, and then we apply break if user inputs 'Goodbye'.

**while**(**True**):

    question = **input**("You: ")

**if** question **==** 'Goodbye':

**break**

* Apply padding: We need to make sure that this question has the same length as the questions that were used for the training and the questions that were used for the training have a length of 20 and therefore now we must complete the length of that question with enough tokens so that this question has a length of 20.

    question = **convert\_string2int**(question, questnWrd2Int)

    question = question + [questnWrd2Int['<PAD>']] \* (20 - **len**(question))

* Fake Batch: Next step when working with neural networks this question must be into a batch (the neural networks only accept batches of inputs). So now we convert this question into a fake batch.
* We're going to create a fake batch which will contain this question and then some empty questions that will only get zeros.
* Arguments: number of lines of this array and the number of columns of this array.
* So the **number of lines = batch\_size** because each line corresponds to a question and the **number of columns = 20**. Because our NN was trained with questions of length 20.
* Then we need to include our question (the "user inputted string") in this ***fake\_batch*** and to do this we will simply include it in the first row of this fake batch.

fake\_batch = **np.zeros**((batch\_size, 20))

fake\_batch[0] = question

* Getting the prediction: Now we're ready to feed that **fake\_batch** into the NN of our chatbot to get the output/answer from it.

predicted\_answer = **session.run**(test\_predictions, {inputs: fake\_batch, keep\_prob: 0.5})[0]

* ***run()***take two arguments:
* The first one is our test\_predictions that we got from the execution of **seq2seq\_model()** in our Training Part.
* The second argument is a dictionary of two sets of keys and values.
* The first one is the ***inputs*** which are of course our ***batch of inputs*** which we called ***fake\_batch*** and then keep probability that remember we set it equal to 0.5, 50 %.
* It'll return a list of several elements, we however interested in the first element, hence we used index [0].
* Preparing the output/answer (prediction) from chatbot for the user:

    answer = ''

**for** i **in** **np.argmax**(predicted\_answer, 1):

**if** ans\_Int\_2\_Wrd[i] **==** 'i':

            token = ' I'

**elif** ans\_Int\_2\_Wrd[i] **==** '<EOS>':

            token = '.'

**elif** ans\_Int\_2\_Wrd[i] **==** '<OUT>':

            token = 'out'

**else**:

            token = ' ' + ans\_Int\_2\_Wrd[i]

        answer += token

**if** token **==** '.':

**break**

**print**('ChatBot: ' + answer)

* **np.argmax**(predicted\_answer, 1) gets the Token Ids in the predicted answer.
* First we convert the tokenized words to real-words, from the encoded output/answer (prediction) from chatbot.
* We use our inverse dictionary **ans\_Int\_2\_Wrd[i]** to do this.
* We replace all – single "i" to "I".
* <EOS> to ".".
* <OUT> to "out"
* We separate the words by " ", spaces: token = ' ' + ans\_Int\_2\_Wrd[i]
* Then we concatenate the tokenized word to the output-string: ***answer***.
* Then if <EOS> or "." comes up we stop this FOR-loop.
* Finally we print the answer.

**7.5 Final Execution for the chat**

* Basically we'll execute *all the codes except* the **Big-Training-FOR-Loop** (if we are using the stored checkpoint files only).
* But if we already executed the training portion, we just need to execute the last section (test section).
* For a trained model, we don't need to execute the "Training FOR loop". We execute all the codes except following FOR-Loop:

# *Training*

batch\_index\_check\_training\_loss = 100

batch\_index\_check\_validation\_loss = ((**len**(training\_questions)) // batch\_size // 2) - 1

total\_training\_loss\_error = 0

list\_validation\_loss\_error = []

early\_stopping\_check = 0

early\_stopping\_stop = 1000      # *we choose 1000 to reach the last epoch i.e. 100-th epoch. Generally we set early\_stopping 100*

checkpoint = "./chatbot\_weights.ckpt" # *For Windows users, replace "chatbot\_weights.ckpt" line of code by: checkpoint = "./chatbot\_weights.ckpt"*

**session.run**(**tf.global\_variables\_initializer**())

**for** epoch **in** **range**(1, epochs + 1):

**for** batch\_index, (padded\_questions\_in\_batch, padded\_answers\_in\_batch) **in** **enumerate**(**split\_into\_batches**(training\_questions, training\_answers, batch\_size)):

        starting\_time = **time.time**()

* We'll execute all the following highlighted code:

**All code at once**

# *Building a ChatBot with Deep-NLP*

'''

NOTE: Need Specific environment. Install following:

    conda create --name py354 python=3.5.4

    conda activate py354

    pip install protobuf==3.19.4

    pip install tensorflow==1.0.0

    conda install ipykernel

    conda install spyder

 '''

# *Impoting the libraries*

**import** numpy **as** np

**import** tensorflow **as** tf

**import** re

**import** time

# *==============================     Part 1 : Data Preprocessing     ==============================*

# *Importing the dataset*

lines\_in\_cnvrstn = **open**("../data\_small\_50\_movie/movie\_lines.txt", encoding='utf-8', errors='ignore').**read**().**split**('\n')

cnvrstn = **open**("../data\_small\_50\_movie/movie\_conversations.txt", encoding='utf-8', errors='ignore').**read**().**split**('\n')

# *Creating a dictionary that maps each linea and its ID*

id\_2\_line = {};

**for** lyn **in** lines\_in\_cnvrstn:

    \_line = **lyn.split**(" +++$+++ ")

**if** **len**(\_line) **==** 5:

        id\_2\_line[\_line[0]] = \_line[4] # *creates the dicttionary*

        # *\_line[0] is id "key in dictionary" and \_line[4] is the line "as value of the key"*

# *Creating the list of all of the conversations*

cnvrstn\_ids = []

**for** cvstn **in** cnvrstn[:-1]:

    \_cnvrstn =  **cvstn.split**(" +++$+++ ")[-1][1:-1].**replace**("'", "").**replace**(" ", "")

**cnvrstn\_ids.append**(**\_cnvrstn.split**(","))

# *Getting seperately the questions and answers*

raw\_questn = []

raw\_ans = []

**for** cvstn **in** cnvrstn\_ids:

**for** i **in** **range**(**len**(cvstn) - 1):

**raw\_questn.append**(id\_2\_line[cvstn[i]])  # *using the "id\_2\_line" dictionary, by id-key*

**raw\_ans.append**(id\_2\_line[cvstn[i+1]])

        # *range(len(cvstn) - 1) is used because of cvstn[i+1]*

        # *notice we are using both "cnvrstn\_ids" and "id\_2\_line"*

# *Doing the first cleaning of the texts*

**def** **clean\_text**(text):

    text = **text.lower**();

    text = **re.sub**(r"i'm", "i am", text)

    text = **re.sub**(r"he's", "he is", text)

    text = **re.sub**(r"she's", "she is", text)

    text = **re.sub**(r"that's", "that is", text)

    text = **re.sub**(r"what's", "what is", text)

    text = **re.sub**(r"where's", "where is", text)

    text = **re.sub**(r"\'ll", " will", text)

    text = **re.sub**(r"\'ve", " have", text)

    text = **re.sub**(r"\'re", " are", text)

    text = **re.sub**(r"\'d", " would", text)

    text = **re.sub**(r"won't", "will not", text)

    text = **re.sub**(r"can't", "cannot", text)

    text = **re.sub**(r"[-()#/@;:<>{}+=~|.?,]", "", text)

**return** text

# *Cleaning the questions*

clean\_questn = []

**for** qes **in** raw\_questn:

**clean\_questn.append**(**clean\_text**(qes))

# *Cleaning the answers*

clean\_ans = []

**for** aNs **in** raw\_ans:

**clean\_ans.append**(**clean\_text**(aNs))

# *create a dictionary that maps each word to its number of occurrences.*

word2count = {}

**for** questn **in** clean\_questn:

**for** word **in** **questn.split**():

**if** word **not** **in** word2count:

            word2count[word] = 1

**else**:

            word2count[word] += 1

**for** ans **in** clean\_ans:

**for** word **in** **ans.split**():

**if** word **not** **in** word2count:

            word2count[word] = 1

**else**:

            word2count[word] += 1

# *Creating two dictionaries that map the questions words and the answers words to a unique integers*

threshold = 20

word\_number = 0

questnWrd2Int = {}

# *'word' is "key" and 'count' is "value"*

**for** word, count **in** **word2count.items**():

**if** count **>=** threshold:

        questnWrd2Int[word] = word\_number

        word\_number += 1

word\_number = 0

ansWrd2Int = {}

**for** word, count **in** **word2count.items**():

**if** count **>=** threshold:

        ansWrd2Int[word] = word\_number

        word\_number += 1

# *Adding the last tokens to these two dictionaries*

tokens = ['<PAD>', '<EOS>', '<OUT>', '<SOS>']

**for** tkn **in** tokens:

    # *adding 'token' as "key" and 'unique int' as "value"*

    questnWrd2Int[tkn] = **len**(questnWrd2Int) + 1

**for** tkn **in** tokens:

    # *adding 'token' as "key" and 'unique int' as "value"*

    ansWrd2Int[tkn] = **len**(ansWrd2Int) + 1

# *Creating inverse dictionary of the ansWrd2Int, notice ':' is used to crete dictionary*

ans\_Int\_2\_Wrd = {w\_i: w **for** w, w\_i **in** **ansWrd2Int.items**()}

# *Adding EOS tokens to the end of every answers*

**for** i **in** **range**(**len**(clean\_ans)):

    clean\_ans[i] += " <EOS>" # *notice a space is added to seperate <EOS>*

# *Translating all the Questions and the Answers into integers*

# *and replacing all the words that were filtered out by our token <OUT>*

questions\_to\_int = []

**for** question **in** clean\_questn:

    ints = []

**for** word **in** **question.split**():

**if** word **not** **in** questnWrd2Int:

            # *Checking filtered word and adding <OUT>'s token*

**ints.append**(questnWrd2Int["<OUT>"])

**else**:

            # *Adding word's corresponding token*

**ints.append**(questnWrd2Int[word])

    # *Finally "ints" is containing a tokenized question sentence,*

    # *we append this to "questions\_to\_int"*

**questions\_to\_int.append**(ints)

# *We do same for the Answers*

answers\_to\_int = []

**for** answer **in** clean\_ans:

    ints = []

**for** word **in** **answer.split**():

**if** word **not** **in** ansWrd2Int:

**ints.append**(ansWrd2Int["<OUT>"])

**else**:

**ints.append**(ansWrd2Int[word])

**answers\_to\_int.append**(ints)

# *Sorting the both questions and answers by the length of the questions*

max\_line\_length = 25

sorted\_clean\_questions = []

sorted\_clean\_answers = []

# *print(enumerate(questions\_to\_int));*

enumarated = **list**(**enumerate**(questions\_to\_int))

**for** length **in** **range**(1, max\_line\_length+1):

**for** i **in** **enumerate**(questions\_to\_int):

**if** **len**(i[1]) **==** length:

**sorted\_clean\_questions.append**(questions\_to\_int[i[0]])

**sorted\_clean\_answers.append**(answers\_to\_int[i[0]])

# *==============================     Part 2 : building SEQ2SEQ model     ==============================*

# *creating TF placeholder for inputs and target*

**def** **model\_inputs**():

    inpUts = **tf.placeholder**(tf.int32, [**None**, **None**], name="input")

    tarGets = **tf.placeholder**(tf.int32, [**None**, **None**], name="target")

    lr = **tf.placeholder**(tf.float32, name="learning\_rate")

    keep\_prob = **tf.placeholder**(tf.float32, name="keep\_prob")

**return** inpUts, tarGets, lr, keep\_prob

# *preprocessiong the tergets*

**def** **preprocess\_targets**(targets, word2int, batch\_size):

    left\_side = **tf.fill**([batch\_size, 1], word2int["<SOS>"])

    right\_side = **tf.strided\_slice**(targets, [0,0], [batch\_size, -1], [1,1])

    preprocessed\_targets = **tf.concat**([left\_side, right\_side], 1)

**return** preprocessed\_targets

# *------------------ Creating the ENCODER RNN layer ----------------*

**def** **encoder\_rnn**(rnn\_inputs, rnn\_size, num\_layers, keep\_prob, sequence\_length):

    lstm = **tf.contrib.rnn.BasicLSTMCell**(rnn\_size)

    lstm\_dropout = **tf.contrib.rnn.DropoutWrapper**(lstm, input\_keep\_prob = keep\_prob)

    encoder\_cell = **tf.contrib.rnn.MultiRNNCell**([lstm\_dropout]\*num\_layers)

    # *encoder\_output, encoder\_state = tf.nn.bidirectional\_dynamic\_rnn*

    encoder\_output, encoder\_state = **tf.nn.bidirectional\_dynamic\_rnn**(cell\_fw = encoder\_cell,

                                                        cell\_bw = encoder\_cell,

                                                        sequence\_length = sequence\_length,

                                                        inputs=rnn\_inputs,

                                                        dtype = tf.float32)

**return** encoder\_state

# *Decoding the training set*

**def** **decode\_training\_set**(encoder\_state, decoder\_cell, decoder\_embedded\_input, sequence\_length, decoding\_scope, output\_function, keep\_prob, batch\_size):

    attention\_states = **tf.zeros**([batch\_size, 1, decoder\_cell.output\_size])

    attention\_keys, attention\_values, attention\_score\_function, attention\_construct\_function = **tf.contrib.seq2seq.prepare\_attention**(

                                                                                                                    attention\_states,

                                                                                                                    attention\_option = "bahdanau",

                                                                                                                    num\_units = decoder\_cell.output\_size)

    training\_decoder\_function = **tf.contrib.seq2seq.attention\_decoder\_fn\_train**(encoder\_state[0],

                                                                              attention\_keys,

                                                                              attention\_values,

                                                                              attention\_score\_function,

                                                                              attention\_construct\_function,

                                                                              name = "attn\_dec\_train")

    decoder\_output, decoder\_final\_state, decoder\_final\_context\_state = **tf.contrib.seq2seq.dynamic\_rnn\_decoder**(decoder\_cell,

                                                                                                              training\_decoder\_function,

                                                                                                              decoder\_embedded\_input,

                                                                                                              sequence\_length,

                                                                                                              scope = decoding\_scope)

    decoder\_output\_dropout = **tf.nn.dropout**(decoder\_output, keep\_prob)

**return** **output\_function**(decoder\_output\_dropout)

# *Decoding the test/validatin set*

**def** **decode\_test\_set**(encoder\_state, decoder\_cell, decoder\_embedding\_matrix, sos\_id, eos\_id, maximum\_length, num\_words, decoding\_scope, output\_function, keep\_prob, batch\_size):

    attention\_states = **tf.zeros**([batch\_size, 1, decoder\_cell.output\_size])

    attention\_keys, attention\_values, attention\_score\_function, attention\_construct\_function = **tf.contrib.seq2seq.prepare\_attention**(

                                                                                                                    attention\_states,

                                                                                                                    attention\_option = "bahdanau",

                                                                                                                    num\_units = decoder\_cell.output\_size)

    test\_decoder\_function = **tf.contrib.seq2seq.attention\_decoder\_fn\_inference**(

                                                                                output\_function,

                                                                                encoder\_state[0],

                                                                                attention\_keys,

                                                                                attention\_values,

                                                                                attention\_score\_function,

                                                                                attention\_construct\_function,

                                                                                decoder\_embedding\_matrix,

                                                                                sos\_id,

                                                                                eos\_id,

                                                                                maximum\_length,

                                                                                num\_words,

                                                                                name = "attn\_dec\_inf")

    test\_prediction, decoder\_final\_state, decoder\_final\_context\_state = **tf.contrib.seq2seq.dynamic\_rnn\_decoder**( decoder\_cell,

                                                                                                                test\_decoder\_function,

                                                                                                                scope = decoding\_scope)

**return** test\_prediction

# *------------------ Creating the DECODER RNN layer ----------------*

**def** **decoder\_rnn**(decoder\_embedded\_input, decoder\_embedding\_matrix, encoder\_state, num\_words, sequence\_length, rnn\_size, num\_layers, word2int,  keep\_prob, batch\_size):

**with** **tf.variable\_scope**("decoding") **as** decoding\_scope:

        lstm = **tf.contrib.rnn.BasicLSTMCell**(rnn\_size)   # *define a layer*

        lstm\_dropout = **tf.contrib.rnn.DropoutWrapper**(lstm, input\_keep\_prob = keep\_prob)     # *Apply dropout on the layer*

        decoder\_cell = **tf.contrib.rnn.MultiRNNCell**([lstm\_dropout]\*num\_layers)   # *Creating stack of the layer*

        # *Intialize weight*

        weights = **tf.truncated\_normal\_initializer**(stddev = 0.1)

        biases = **tf.zeros\_initializer**()

        output\_function = **lambda** x: **tf.contrib.layers.fully\_connected**(  x,

                                                                        num\_words,

**None**,

                                                                        scope = decoding\_scope,

                                                                        weights\_initializer = weights,

                                                                        biases\_initializer = biases)

        # *Training Predictions*

        training\_predictions = **decode\_training\_set**( encoder\_state,

                                                    decoder\_cell,

                                                    decoder\_embedded\_input,

                                                    sequence\_length,

                                                    decoding\_scope,

                                                    output\_function,

                                                    keep\_prob,

                                                    batch\_size)

        # *Test Predictions*

**decoding\_scope.reuse\_variables**()

        test\_prediciton = **decode\_test\_set**(  encoder\_state,

                                            decoder\_cell,

                                            decoder\_embedding\_matrix,

                                            word2int['<SOS>'],

                                            word2int['<EOS>'],

                                            sequence\_length -1 ,

                                            num\_words,

                                            decoding\_scope,

                                            output\_function,

                                            keep\_prob,

                                            batch\_size)

**return** training\_predictions, test\_prediciton

'''

--- Python Lambda ---

    A lambda function is a small anonymous function. A lambda function can take any number of arguments, but can only have one expression.

Syntax

        lambda arguments : expression

    The expression is executed and the result is returned

Example:

    Add 10 to argument a, and return the result:

    x = lambda a : a + 10

    print(x(5))

'''

# *Building the seq2seq model : Brain of our chatbot*

**def** **seq2seq\_model**(inputs, targets, keep\_prob, batch\_size, sequence\_length, answers\_num\_words, questions\_num\_words, encoder\_embedding\_size, decoder\_embedding\_size, rnn\_size, num\_layers, questnWrd2Int):

    # *assemblage = putting togather*

    encoder\_embedded\_input = **tf.contrib.layers.embed\_sequence**(  inputs,

                                                                answers\_num\_words+1,

                                                                encoder\_embedding\_size,

                                                                initializer = **tf.random\_uniform\_initializer**(0, 1))

    encoder\_state = **encoder\_rnn**(encoder\_embedded\_input, rnn\_size, num\_layers, keep\_prob, sequence\_length)

    preprocessed\_targets = **preprocess\_targets**(targets, questnWrd2Int, batch\_size)

    decoder\_embeddings\_matrix = **tf.Variable**(**tf.random\_uniform**([questions\_num\_words+1, decoder\_embedding\_size], 0, 1))

    decoder\_embedded\_input = **tf.nn.embedding\_lookup**(decoder\_embeddings\_matrix, preprocessed\_targets)

    traing\_pred, test\_pred = **decoder\_rnn**(   decoder\_embedded\_input,

                                            decoder\_embeddings\_matrix,

                                            encoder\_state,

                                            questions\_num\_words,

                                            sequence\_length,

                                            rnn\_size,

                                            num\_layers,

                                            questnWrd2Int,

                                            keep\_prob,

                                            batch\_size)

**return** traing\_pred, test\_pred

# *==============================     Part 3 : Trainig SEQ2SEQ model     ==============================*

# *Setting the HyperPartameters: We choosed these names for TensorFlow matching*

epochs = 100    # *or 50*

batch\_size = 64    # *or 128*

rnn\_size = 512

num\_layers = 3

encoding\_embedding\_size = 512

decoding\_embedding\_size = 512

learning\_rate = 0.01

learning\_rate\_decay = 0.9

min\_learning\_rate = 0.0001

keep\_probability = 0.5  # *optimal according to Hinton*

# *Defining a session*

**tf.reset\_default\_graph**()

session = **tf.InteractiveSession**()

# *Loding the model Inputs*

inputs, targets, lr, keep\_prob = **model\_inputs**()

# *Setting the Sequence Length*

sequence\_length = **tf.placeholder\_with\_default**(25, **None**, name = 'sequence\_length')

# *Getting the shape of the inputs tensor*

input\_shape = **tf.shape**(inputs)

# *Getting the training predictions and test predictions*

training\_predictions, test\_predictions = **seq2seq\_model**(**tf.reverse**(inputs, [-1]),

                                                       targets,

                                                       keep\_prob,

                                                       batch\_size,

                                                       sequence\_length,

**len**(ansWrd2Int),

**len**(questnWrd2Int),

                                                       encoding\_embedding\_size,

                                                       decoding\_embedding\_size,

                                                       rnn\_size,

                                                       num\_layers,

                                                       questnWrd2Int)

# *Setting up the Loss Error, the Optimizer and Gradient Clipping*

**with** **tf.name\_scope**("optimization"):

    loss\_error = **tf.contrib.seq2seq.sequence\_loss**(training\_predictions,

                                                  targets,

**tf.ones**([input\_shape[0], sequence\_length]))

    optimizer = **tf.train.AdamOptimizer**(learning\_rate)

    gradients = **optimizer.compute\_gradients**(loss\_error)

    clipped\_gradients = [(**tf.clip\_by\_value**(grad\_tensor, -5., 5.), grad\_variable) **for** grad\_tensor, grad\_variable **in** gradients **if** grad\_tensor **is** **not** **None**]

    optimizer\_gradient\_clipping = **optimizer.apply\_gradients**(clipped\_gradients)

# *Padding the sequences with the <PAD> token*

    # *Question: ["Who", "are", "you"]*

    # *Answer: [<SOS>, "I", "am", "a", "bot",".", <EOS>]*

    # *After we apply the padding this question and answer will become this:*

        # *Question: ["Who", "are", "you", <PAD>, <PAD>, <PAD>, <PAD>]*

        # *Answer: [<SOS>, "I", "am", "a", "bot",".", <EOS>]*

**def** **apply\_padding**(batch\_of\_sequences, word2int):

    max\_sequence\_length = **max**([**len**(sequence) **for** sequence **in** batch\_of\_sequences])

**return** [sequence + [word2int['<PAD>']] \* (max\_sequence\_length - **len**(sequence)) **for** sequence **in** batch\_of\_sequences]

# *Splitting the data into batches of questions and answers*

**def** **split\_into\_batches**(questions, answers, batch\_size):

**for** batch\_index **in** **range**(0, **len**(questions) // batch\_size):

        start\_index = batch\_index \* batch\_size

        questions\_in\_batch = questions[start\_index : start\_index + batch\_size]

        answers\_in\_batch = answers[start\_index : start\_index + batch\_size]

        padded\_questions\_in\_batch = **np.array**(**apply\_padding**(questions\_in\_batch, questnWrd2Int))

        padded\_answers\_in\_batch = **np.array**(**apply\_padding**(answers\_in\_batch, ansWrd2Int))

**yield** padded\_questions\_in\_batch, padded\_answers\_in\_batch

# *Splitting the questions and answers into training and validation sets*

training\_validation\_split = **int**(**len**(sorted\_clean\_questions) \* 0.15)

training\_questions = sorted\_clean\_questions[training\_validation\_split:]

training\_answers = sorted\_clean\_answers[training\_validation\_split:]

validation\_questions = sorted\_clean\_questions[:training\_validation\_split]

validation\_answers = sorted\_clean\_answers[:training\_validation\_split]

# *Training*

batch\_index\_check\_training\_loss = 100

batch\_index\_check\_validation\_loss = ((**len**(training\_questions)) // batch\_size // 2) - 1

total\_training\_loss\_error = 0

list\_validation\_loss\_error = []

early\_stopping\_check = 0

early\_stopping\_stop = 1000      # *we choose 1000 to reach the last epoch i.e. 100-th epoch. Generally we set early\_stopping 100*

checkpoint = "./chatbot\_weights.ckpt" # *For Windows users, replace "chatbot\_weights.ckpt" line of code by: checkpoint = "./chatbot\_weights.ckpt"*

**session.run**(**tf.global\_variables\_initializer**())

**for** epoch **in** **range**(1, epochs + 1):

**for** batch\_index, (padded\_questions\_in\_batch, padded\_answers\_in\_batch) **in** **enumerate**(**split\_into\_batches**(training\_questions, training\_answers, batch\_size)):

        starting\_time = **time.time**()

        \_, batch\_training\_loss\_error = **session.run**([optimizer\_gradient\_clipping, loss\_error], {inputs: padded\_questions\_in\_batch,

                                                                                               targets: padded\_answers\_in\_batch,

                                                                                               lr: learning\_rate,

                                                                                               sequence\_length: padded\_answers\_in\_batch.shape[1],

                                                                                               keep\_prob: keep\_probability})

        total\_training\_loss\_error += batch\_training\_loss\_error

        ending\_time = **time.time**()

        batch\_time = ending\_time - starting\_time

**if** batch\_index % batch\_index\_check\_training\_loss **==** 0:

**print**('Epoch: {:>3}/{}, Batch: {:>4}/{}, Training Loss Error: {:>6.3f}, Training Time on 100 Batches: {:d} seconds'.**format**(epoch,

                                                                                                                                       epochs,

                                                                                                                                       batch\_index,

**len**(training\_questions) // batch\_size,

                                                                                                                                       total\_training\_loss\_error / batch\_index\_check\_training\_loss,

**int**(batch\_time \* batch\_index\_check\_training\_loss)))

            total\_training\_loss\_error = 0

        # *validation loss-error*

**if** batch\_index % batch\_index\_check\_validation\_loss **==** 0 **and** batch\_index **>** 0:

            total\_validation\_loss\_error = 0

            starting\_time = **time.time**()

**for** batch\_index\_validation, (padded\_questions\_in\_batch, padded\_answers\_in\_batch) **in** **enumerate**(**split\_into\_batches**(validation\_questions, validation\_answers, batch\_size)):

                batch\_validation\_loss\_error = **session.run**(loss\_error, {inputs: padded\_questions\_in\_batch,

                                                                       targets: padded\_answers\_in\_batch,

                                                                       lr: learning\_rate,

                                                                       sequence\_length: padded\_answers\_in\_batch.shape[1],

                                                                       keep\_prob: 1})

                total\_validation\_loss\_error += batch\_validation\_loss\_error

            ending\_time = **time.time**()

            batch\_time = ending\_time - starting\_time

            average\_validation\_loss\_error = total\_validation\_loss\_error / (**len**(validation\_questions) / batch\_size)

**print**('Validation Loss Error: {:>6.3f}, Batch Validation Time: {:d} seconds'.**format**(average\_validation\_loss\_error, **int**(batch\_time)))

            # *decay to the learning rates and early stopping*

            learning\_rate \*= learning\_rate\_decay

**if** learning\_rate **<** min\_learning\_rate:

                learning\_rate = min\_learning\_rate

**list\_validation\_loss\_error.append**(average\_validation\_loss\_error)

**if** average\_validation\_loss\_error **<=** **min**(list\_validation\_loss\_error):

**print**('I speak better now!!')

                early\_stopping\_check = 0

                saver = **tf.train.Saver**()

**saver.save**(session, checkpoint)

**else**:

**print**("Sorry I do not speak better, I need to practice more.")

                early\_stopping\_check += 1

**if** early\_stopping\_check **==** early\_stopping\_stop:

**break**

**if** early\_stopping\_check **==** early\_stopping\_stop:

**print**("My apologies, I cannot speak better anymore. This is the best I can do.")

**break**

**print**("Game Over")

#*######### PART 4 - TESTING THE SEQ2SEQ MODEL ##########*

# *Loading the weights and Running the session*

checkpoint = "./chatbot\_weights.ckpt"

session = **tf.InteractiveSession**()

**session.run**(**tf.global\_variables\_initializer**())

saver = **tf.train.Saver**()

**saver.restore**(session, checkpoint)

# *Converting the questions from strings to lists of encoding integers*

**def** **convert\_string2int**(question, word2int):

    question = **clean\_text**(question)

**return** [**word2int.get**(word, word2int['<OUT>']) **for** word **in** **question.split**()]

# *Setting up the chat*

**while**(**True**):

    question = **input**("You: ")

**if** question **==** 'Goodbye':

**break**

    question = **convert\_string2int**(question, questnWrd2Int)

    question = question + [questnWrd2Int['<PAD>']] \* (20 - **len**(question))

    fake\_batch = **np.zeros**((batch\_size, 20))

    fake\_batch[0] = question

    predicted\_answer = **session.run**(test\_predictions, {inputs: fake\_batch, keep\_prob: 0.5})[0]

    answer = ''

**for** i **in** **np.argmax**(predicted\_answer, 1):

**if** ans\_Int\_2\_Wrd[i] **==** 'i':

            token = ' I'

**elif** ans\_Int\_2\_Wrd[i] **==** '<EOS>':

            token = '.'

**elif** ans\_Int\_2\_Wrd[i] **==** '<OUT>':

            token = 'out'

**else**:

            token = ' ' + ans\_Int\_2\_Wrd[i]

        answer += token

**if** token **==** '.':

**break**

**print**('ChatBot: ' + answer)

# *python prctc\_ctbbt.py*

# *conda activate py354*

# *spyder*

* However, it is dumb

