Checkpoint!

Congratulations for completing Part 4 - Testing the Seq2Seq model. Here is the code we have implemented so far:

# Building a ChatBot with Deep NLP

# Importing the libraries

import numpy as np

import tensorflow as tf

import re

import time

########## PART 1 - DATA PREPROCESSING ##########

# Importing the dataset

lines = open('movie\_lines.txt', encoding = 'utf-8', errors = 'ignore').read().split('\n')

conversations = open('movie\_conversations.txt', encoding = 'utf-8', errors = 'ignore').read().split('\n')

# Creating a dictionary that maps each line and its id

id2line = {}

for line in lines:

\_line = line.split(' +++$+++ ')

if len(\_line) == 5:

id2line[\_line[0]] = \_line[4]

# Creating a list of all of the conversations

conversations\_ids = []

for conversation in conversations[:-1]:

\_conversation = conversation.split(' +++$+++ ')[-1][1:-1].replace("'", "").replace(" ", "")

conversations\_ids.append(\_conversation.split(','))

# Getting separately the questions and the answers

questions = []

answers = []

for conversation in conversations\_ids:

for i in range(len(conversation) - 1):

questions.append(id2line[conversation[i]])

answers.append(id2line[conversation[i+1]])

# Doing a 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"how's", "how 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"n't", " not", 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\_questions = []

for question in questions:

clean\_questions.append(clean\_text(question))

# Cleaning the answers

clean\_answers = []

for answer in answers:

clean\_answers.append(clean\_text(answer))

# Filtering out the questions and answers that are too short or too long

short\_questions = []

short\_answers = []

i = 0

for question in clean\_questions:

if 2 <= len(question.split()) <= 25:

short\_questions.append(question)

short\_answers.append(clean\_answers[i])

i += 1

clean\_questions = []

clean\_answers = []

i = 0

for answer in short\_answers:

if 2 <= len(answer.split()) <= 25:

clean\_answers.append(answer)

clean\_questions.append(short\_questions[i])

i += 1

# Creating a dictionary that maps each word to its number of occurrences

word2count = {}

for question in clean\_questions:

for word in question.split():

if word not in word2count:

word2count[word] = 1

else:

word2count[word] += 1

for answer in clean\_answers:

for word in answer.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 integer

threshold\_questions = 15

questionswords2int = {}

word\_number = 0

for word, count in word2count.items():

if count >= threshold\_questions:

questionswords2int[word] = word\_number

word\_number += 1

threshold\_answers = 15

answerswords2int = {}

word\_number = 0

for word, count in word2count.items():

if count >= threshold\_answers:

answerswords2int[word] = word\_number

word\_number += 1

# Adding the last tokens to these two dictionaries

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

for token in tokens:

questionswords2int[token] = len(questionswords2int) + 1

for token in tokens:

answerswords2int[token] = len(answerswords2int) + 1

# Creating the inverse dictionary of the answerswords2int dictionary

answersints2word = {w\_i: w for w, w\_i in answerswords2int.items()}

# Adding the End Of String token to the end of every answer

for i in range(len(clean\_answers)):

clean\_answers[i] += ' <EOS>'

# Translating all the questions and the answers into integers

# and Replacing all the words that were filtered out by <OUT>

questions\_into\_int = []

for question in clean\_questions:

ints = []

for word in question.split():

if word not in questionswords2int:

ints.append(questionswords2int['<OUT>'])

else:

ints.append(questionswords2int[word])

questions\_into\_int.append(ints)

answers\_into\_int = []

for answer in clean\_answers:

ints = []

for word in answer.split():

if word not in answerswords2int:

ints.append(answerswords2int['<OUT>'])

else:

ints.append(answerswords2int[word])

answers\_into\_int.append(ints)

# Sorting questions and answers by the length of questions

sorted\_clean\_questions = []

sorted\_clean\_answers = []

for length in range(1, 25 + 1):

for i in enumerate(questions\_into\_int):

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

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

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

########## PART 2 - BUILDING THE SEQ2SEQ MODEL ##########

# Creating placeholders for the inputs and the targets

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

# Preprocessing the targets

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

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(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/validation set

def decode\_test\_set(encoder\_state, decoder\_cell, decoder\_embeddings\_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\_embeddings\_matrix,

sos\_id,

eos\_id,

maximum\_length,

num\_words,

name = "attn\_dec\_inf")

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

test\_decoder\_function,

scope = decoding\_scope)

return test\_predictions

# Creating the Decoder RNN

def decoder\_rnn(decoder\_embedded\_input, decoder\_embeddings\_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)

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

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

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 = decode\_training\_set(encoder\_state,

decoder\_cell,

decoder\_embedded\_input,

sequence\_length,

decoding\_scope,

output\_function,

keep\_prob,

batch\_size)

decoding\_scope.reuse\_variables()

test\_predictions = decode\_test\_set(encoder\_state,

decoder\_cell,

decoder\_embeddings\_matrix,

word2int['<SOS>'],

word2int['<EOS>'],

sequence\_length - 1,

num\_words,

decoding\_scope,

output\_function,

keep\_prob,

batch\_size)

return training\_predictions, test\_predictions

# Building the seq2seq model

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, questionswords2int):

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, questionswords2int, 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)

training\_predictions, test\_predictions = decoder\_rnn(decoder\_embedded\_input,

decoder\_embeddings\_matrix,

encoder\_state,

questions\_num\_words,

sequence\_length,

rnn\_size,

num\_layers,

questionswords2int,

keep\_prob,

batch\_size)

return training\_predictions, test\_predictions

########## PART 3 - TRAINING THE SEQ2SEQ MODEL ##########

# Setting the Hyperparameters

epochs = 100

batch\_size = 32

rnn\_size = 1024

num\_layers = 3

encoding\_embedding\_size = 1024

decoding\_embedding\_size = 1024

learning\_rate = 0.001

learning\_rate\_decay = 0.9

min\_learning\_rate = 0.0001

keep\_probability = 0.5

# Defining a session

tf.reset\_default\_graph()

session = tf.InteractiveSession()

# Loading 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 and test predictions

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

targets,

keep\_prob,

batch\_size,

sequence\_length,

len(answerswords2int),

len(questionswords2int),

encoding\_embedding\_size,

decoding\_embedding\_size,

rnn\_size,

num\_layers,

questionswords2int)

# 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

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, questionswords2int))

padded\_answers\_in\_batch = np.array(apply\_padding(answers\_in\_batch, answerswords2int))

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 = 100

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

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)))

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, questionswords2int)

question = question + [questionswords2int['<PAD>']] \* (25 - len(question))

fake\_batch = np.zeros((batch\_size, 25))

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 answersints2word[i] == 'i':

token = ' I'

elif answersints2word[i] == '<EOS>':

token = '.'

elif answersints2word[i] == '<OUT>':

token = 'out'

else:

token = ' ' + answersints2word[i]

answer += token

if token == '.':

break

print('ChatBot: ' + answer)