CS156 Final Project

Creating A Neural Conversational Chatbot Trained On My Telegram Chat History With My Girlfriend Using A Seq2Seq Model

Steven Tey

Table of Contents

Context

My girlfriend and I have been in a long-distance for over a year now, and since we are both in Minerva (but from different classes), we only get to meet once every 4 months. Furthermore, due to the global rotation, we often find ourselves having to navigate 8 hours of time difference between us, which can get quite frustrating at times. Therefore, I decided to create a chatbot that speaks like the way I would in order to keep her company whenever I'm asleep. Given that we chat *a lot* on Telegram, I happened to have a large dataset of our conversation which, after some arduous pre-processing, I was able to use to train a Seq2Seq model that would output phrases based on a given prompt. The goal of this experiment - albeit a rather ambitious one - is to perform a Turing test on my girlfriend with this neural conversational model to see if she realizes that she is chatting with an AI version of me instead of the real me.

Preprocessing the Data

By using the Telegram Web App, I extracted 10 months' worth of chat history between my girlfriend and I, which culminated in almost 200,000 lines of messages (195,957 to be exact). To get a more granular breakdown of the messages that were exchanged between us, I decided to do a quick data-viz by creating a word cloud that showed the most common words that we used (you can also check out the code here):

Then, I started preprocessing the data to make sure that it's in the format that is training-compatible with the Seq2Seq model that I will be building later. To do that, I first opened the telegram texts with the following code:

```
with open('telegram.txt', 'r') as telegram:
telegram_data = telegram.readlines()
# checking the messages
telegram_data[-10:]
```

This gave me the following output:

```
["29.10.2019 15:53:48, Steven Tey: I'll talk to you later!! 😘\n",
 '29.10.2019 15:53:54, Steven Tey: Good later my love ❤😔\n',
 '29.10.2019 15:53:56, Anzhelika Korolova: Good later❤😘\n',
 '29.10.2019 15:53:57, Steven Tey: [[?messageMediaUnsupported]]\n',
 '29.10.2019 15:53:59, Anzhelika Korolova: [[?messageMediaUnsupported]]\n',
 '29.10.2019 15:53:59, Steven Tey: [[?messageMediaUnsupported]]\n',
 "29.10.2019 15:54:02, Anzhelika Korolova: [[😴 Sticker, size 17'458 bytes]]\n",
 "29.10.2019 15:54:02, Steven Tey: [[😴 Sticker, size 17'718 bytes]]\n",
 "29.10.2019 16:20:58, Steven Tey: [[Video, size 4'804'352 bytes]]\n",
 '29.10.2019 16:21:06, Steven Tey: [[Photo]]This is so creepy wth 😭🙈😨😨😨']
```

As you can see, a lot of the messages above are not in the right format - there are the date and timestamps, the "\n" tags, as well as some error messages and video/photo/sticker tags. Therefore, I'll have to pre-process the data to reduce it to just key-value pairs, where the key is the prompt (my girlfriend's message) and the value is the response (my corresponding response to her message).

First, let's remove all the emojis using the following Python script that I found on Stackoverflow:

```
def remove_emoji(text, remove_components=False):
    '''
    Script to remove emojis from string, retrieved from:
    https://stackoverflow.com/a/51785357
    '''
    cleaned = emoji.get_emoji_regexp().sub(u'', text)
    if remove_components:
        cleaned = emoji_components.sub(u'', cleaned)
    return cleaned
```

Next, let's preprocess the text further by removing all the links and splitting it into two parts: the sender name, and the message sent:

```
def strip_text(text):
    '''
    Process the text further
    '''
    noEmojiMessage = remove_emoji(text)
    # Remove any links in the message
    noLinksMessage = re.sub(r'^https?:\/\/.*[\r\n]*', '', noEmojiMessage,
flags=re.MULTILINE)
    # Remove any email addresses in the message
    processed_line = noLinksMessage.replace('\S*@\S*\s?',' ')
    first_colon = processed_line.find(': ') + 1
    just_message = processed_line[first_colon:].strip()
    sender_name = processed_line[:first_colon]
    return processed_line, first_colon, just_message, sender_name
```

Then, we'll create a progress bar that shows the progress of the algorithm as it works its way through the 200,000 text messages that we feed into it:

```
def progressBar(current, total, barLength = 20):
    '''
    Standard progress bar algorithm retrieved from
    https://stackoverflow.com/a/37630397
    '''
    percent = float(current) * 100 / total  arrow   = '-' * int(percent/100 *
barLength - 1) + '>'
    spaces  = ' ' * (barLength - len(arrow))

    print('Processing the data: [%s%s] %d %%' % (arrow, spaces, percent), end='\r')
```

Now comes the main function for us to retrieve each line from the .txt file and process it into key-value pairs in our main response dictionary object. The way I will be doing this is by iterating across each line of the .txt file, remove all the lines that contain tags like "[[Photo]]", "[[Video]]", and "[[?messageMediaUnsupported]]". Then, I will append all consecutive messages into two temporary variables - one for my messages and one for my girlfriend's messages - and once an exchange is over, I store those two messages as a key-value pair in the dictionary.

Here's the code:

```
def get_Messages(message_data):
    '''
    Main function to process the data:
    - message_data: the txt file with all the extracted messages
    '''
    responseDictionary = dict()
    myMessage, otherPersonsMessage, currentSpeaker = "","",""
    forbidden_list = ["[[Photo", "[[Video", "[[Voice", "[[GIF", "[[Webpage]]",
                          "[[?messageMediaUnsupported]]", "{{FWD", "{{GIF", "[[Geo"
                          "[[Document", ">>incoming call", ">>outgoing call",
                          ">>missed call", ">>canceled call"]
    startMessageIndex = 0
    process_count = 0
    for idx, line in enumerate(message_data):
        processed_line, first_colon, just_message, sender_name = strip_text(line)
        if len(just_message) < 3:
            continue
        elif bool([ele for ele in forbidden_list if(ele in just_message)]) == True:
            continue
        elif "[[" and "Sticker" in just_message:
            continue
        else:
            # if the current message is from me
            if "Steven Tey" in sender_name:
                if not myMessage:
                    # Get the first message I sent
                    # (if I send multiple messages in a row)
                    startMessageIndex = idx - 1
                # add current message to myMessage cache
                myMessage = myMessage + ' ' + just_message
            elif myMessage: # if I'm not the one speaking
                for counter in range(startMessageIndex, 0, -1):
                    currentLine = message_data[counter]
                    processed_line_other, first_colon_other, just_message_other,
                    sender_name_other = strip_text(currentLine)
                    if len(just_message_other) < 3:
                        continue
                    elif bool([ele for ele in forbidden_list if(ele in
                    just_message)]) ==  True:
                        continue
                    elif "[[" and "Sticker" in just_message_other:
                        continue
                    else:
                        if not currentSpeaker:
                            # If the currentSpeaker variable is empty,
                            # assign Anzhelika as current speaker
                            if "Anzhelika" in sender_name_other:
```

```
                        currentSpeaker = "Anzhelika"
                    # Once there is a current speaker and the
                    # otherPersonsMessage variable is not empty
                    elif ("Anzhelika" not in sender_name_other and
                    otherPersonsMessage):
                        # I started speaking again, so now I know that Anzhelika
                        # stopped speaking and the exchange is complete
                        responseDictionary[otherPersonsMessage] = myMessage
                        break
                    # But if those two conditions don't hold
                    # add current message to otherPersonsMessage cache
                    otherPersonsMessage = just_message_other + ' ' +
                    otherPersonsMessage
                # clear caches at the end of the loop
                myMessage, otherPersonsMessage, currentSpeaker = "","",""
        process_count += 1
        progressBar(process_count, len(message_data), barLength = 30)
    return responseDictionary
```

Now, it's time to run the code. Since I will not be including the .txt file that contains all my messages in my assignment for confidentiality purposes, here's a screen recording of the algorithm in action. We can also print out the key-value pairs to make sure that everything is going the way we want it to - here's the code to do that:

```
for key, value in messagesDict.items():
    if (not key.strip() or not value.strip()):
        # If there are empty strings
        continue
    print(f'Prompt: {key.strip()}\n Response: {value.strip()}\n')
```

Here's a portion of the output:

```
Prompt: Good luck with your assignment!!!
 Response: Thank you boo!!

Prompt: I LOVE YOU Now go!!
 Response: I love you moreee

Prompt: Babyyyy I am so sorry But I have to go now
 Response: no worries baby!!! Goodnight my love
```

As you can see, everything seems to be going according to plan.

Next up, we need to convert the response dictionary into a format that is parsable by the Seq2Seq algorithm that we will be building and using in the next section. To do that, we will be saving the dictionary as a NumPy object and converting the key-value pairs in the dictionary into a .txt file that will be used to create the word list.

```
print('Saving messages data into a dictionary...')
np.save('responseDict.npy', messagesDict)
print('Done!')

print('Converting dictionary into a txt file...')
messagesFile = open('responseData.txt', 'w')
for key, value in messagesDict.items():
   if (not key.strip() or not value.strip()):
       # If there are empty strings
       continue
   messagesFile.write(key.strip() + value.strip())
print('Done! Filed saved at responseData.txt')
```

Perfect! Now we're all set to start building the Seq2Seq algorithm.

Building the Seq2Seq Algorithm

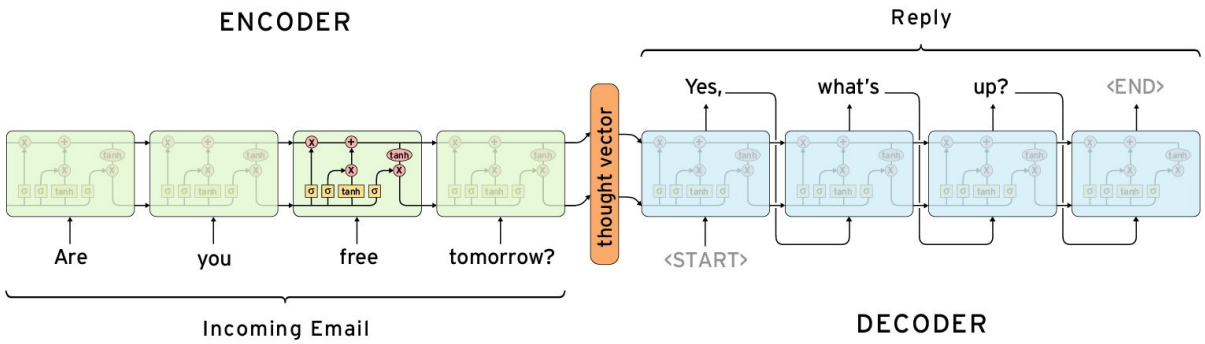
Before we start building the Seq2Seq algorithm, let's do a quick dive to understand what the Seq2Seq algorithm actually is:

Introduction to Seq2Seq

A sequence-to-sequence (seq2seq) network is a model made up of two recurrent neural networks (RNNs) called the encoder and the decoder. The encoder reads a sequence of words one by one and produces a vector at each step, and the cumulative output of the encoder is kept as the context vector. The decoder then uses this context vector to produce a sequence of outputs one at a time (Robertson, 2017).

A regular RNN has a one-to-one relationship between inputs and outputs, which can cause problems especially considering the amorphous nature of language and human speech. For instance, the input "how are you" does not always result in a corresponding output of "I am fine", but it can also generate responses like "I am doing well" or "not too bad actually", the latter of which violates the one-to-one relationship mentioned earlier.

In comparison, a seq2seq model is able to circumvent this issue by replacing vanilla RNNs with Long Short Term Memory (LSTMs), which is a specific type of RNN that can take care of temporal sequence and their long-range dependencies more accurately then vanilla counterparts (Prassanna et. al, 2020). Therefore, by encoding multiple inputs into a single vector and then decoding it into multiple outputs, we no longer are constrained by the length or order of the sentence (Robertson, 2017). This encoded sequence is represented by a single vector, a single point in some N-dimensional space of sequences that carries the meaning of the entire sequence (otherwise known as the semantic summary of the sentence). In other words, when fed with an input, the decoder sequence will generate a response one word at a time while analyzing the context and the preceding word at each timestep.



Source: Deepan, S., 2017

Generating the List of Words

Before we start building the Seq2Seq model, we also need to further process the responseData.txt file that we created in the previous step into a format that is acceptable by the Seq2Seq model. To do that, we will be creating a `getWordList()` function that will convert response data into a huge string and then identify all the unique words in the dataset.

```
def getWordList(data):
    '''
    This function takes in the response data and converts it
    into one huge string, and then uses a Counter to identify
    unique words and the number of occurences
    '''
    conversations = open(data, 'r')
    allLines = conversations.readlines()
    combinedLines = ""
```

```
    for line in allLines:
        combinedLines += line
    wordDict = Counter(combinedLines.split())
    return combinedLines, wordDict
```

Now, we just have to run the function on the responseData.txt file that we got earlier and convert it into a wordList.txt file:

```
# run the getWordList function on the responseData.txt file
fullCorpus, convoDict = getWordList('responseData.txt')
print('Converted response data into full corpus and conversation dictionary.')

# get all the keys in the dictionary
# (the values are the number of occurences of each key)
wordList = list(convoDict.keys())

# Pickle dump the wordlist into .txt file
with open("wordList.txt", "wb") as fp:
   pickle.dump(wordList, fp)
```

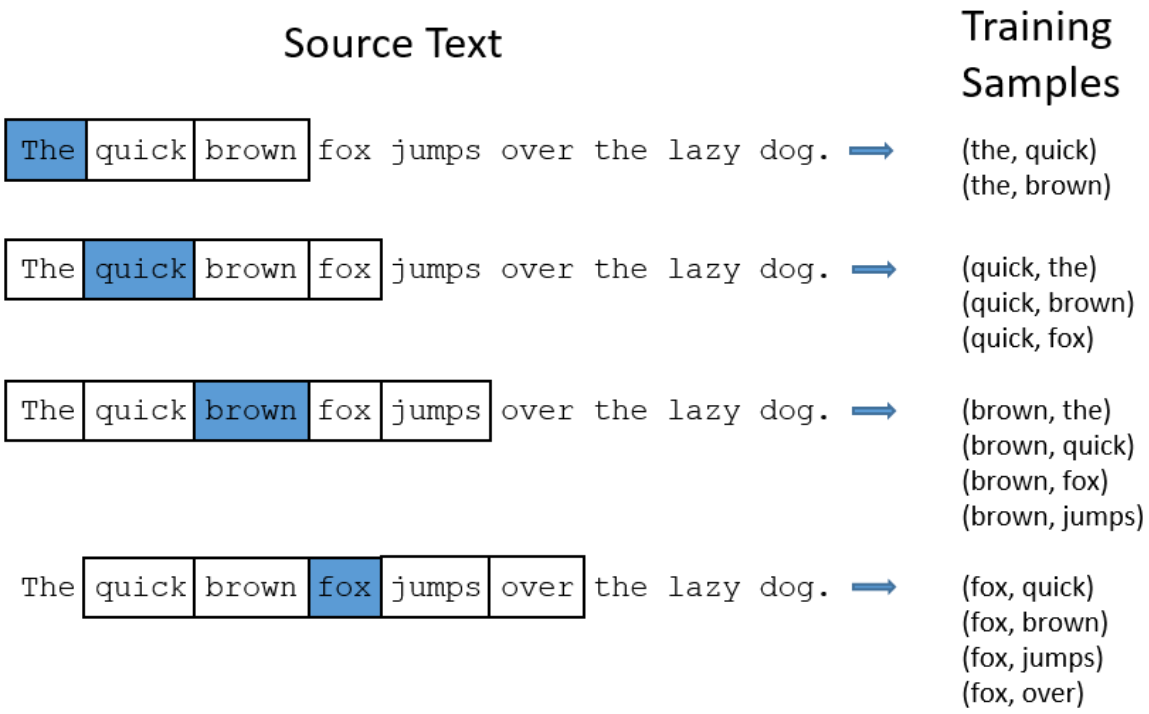
With this, we are ready to build our Seq2Seq model.

Building the Seq2Seq model

Since this is quite a lengthy process, I will not be going through all of the granular nooks and crannies of the code, but rather provide a high-level overview of each of the operations and functions that are called on in the model. You can refer to the code in the `Seq2Seq.py` file in the Seq2Seq folder.

First, we need to build a Word2Vec model, which is a two-layer neural network model that converts a sequence of words into their corresponding word vectors. This model will give us an embedding matrix, which will then be used in the Seq2Seq model for training purposes. To build the Word2Vec model, we have two options: a) we could create our own word vectors using NumPy and pandas or b) we could just use Google's TensorFlow word2vec estimator. For simplicity's sake, I decided to go with the latter option, but instead of treating it as a black box and let Tensorflow handle everything, I did some further research (Joshi, 2017) and learned about the step-by-step process that went on behind the scene:

1) Break down the list of sentences in the response dictionary into lists of words.
2) Determine our batch size, which will help us generate a list of word pairings up to 24 words for each of the words in the wordList.txt file. To put this into perspective, here's an example given by Chris McCormick in his post about word2vec - in this case, his batch size is 2:



3) After generating our training data, we will need to convert all the words and their respective word pairings into integers, a format that can be understood by computers. In our case, we will be using a 4-letter/number combination system, which allows us to represent up to 1,413,720 different permutations (${}^{36}P_4$).
4) At this point, we have our training data, which is represented in the form `x_train` and `y_train` → `x_train` being the input words, and `y_train` being the 24 possible output words for each input word. We will be storing these two datasets in two NumPy files - `Seq2SeqXTrain.npy` and `Seq2SeqYTrain.npy` - you will get both those files once you run the `Seq2Seq.py` file in the Seq2Seq folder (instructions in README).
5) Now, we will build our Tensorflow model. To do that, we will first make placeholders for `x_train` and `y_train` using the `tf.placeholder()` command. Then, we will convert our training data into the embedded representation, which is located in the hidden layer. This embedded matrix will then be used by the decoder sequence to predict the possible neighboring words around a given input word.

6) We also have to set the hyperparameters for the Tensorflow model - in this case, we will be setting our hyperparameters based on values recommended by Prassanna et. al:
    a) `batchSize = 24`
    b) `maxEncoderLength = 15`
    c) `maxDecoderLength = maxEncoderLength`
    d) `lstmUnits = 112`
    e) `embeddingDim = lstmUnits`
    f) `numLayersLSTM = 3`
    g) `numIterations = 50100`
7) Finally, we can then train the model using Tensorflow, which will give us a display of the loss as well as the different word pairings that are being learned by the model. For instance, here's the iteration process of a specific training phrase fed into the model:

```
Current loss: 5.8325748 at iteration 550
do you love me?
[]

Current loss: 4.546090 at iteration 550
do you love me?
['I']

Current loss: 3.4589345 at iteration 550
do you love me?
['I love']

Current loss: 2.7217882 at iteration 7350
do you love me?
['I love ', 'you you ']

Current loss: 2.73785 at iteration 9350
do you love me?
['I love you you you you you you ']

Current loss: 2.5351307 at iteration 14200
do you love me?
['I love you you so much ']

Current loss: 2.4945557 at iteration 17150
do you love me?
['I love you so much ']
```

> As you can see, the model starts off by producing a bunch of empty responses. This makes sense since the model is essentially outputting padding and EOS tokens - which are the most common tokens in the whole dataset. Then, it starts to produce a bunch of "you"s, probably because "you" is one of the most common words I used out of the 115,000 messages that I've sent to Anzhelika. Eventually, as the model stabilizes on a loss value of around 2.5, the output phrase has progressed from a non-sensical "I love, you you" into the full phrase "I love you so much".

Note: As the model trains, aside from printing out loss values every 50 iterations, it will also be saving checkpoint (`.ckpt`) files every 1,000 iterations under a folder called "models". These checkpoint files contain a main checkpoint file, a data file (pretty big file, around 1GB), an index file, and a meta file. For the next step, I will be using the checkpoint files from the last iteration (`iteration 50000`) to build the conversational chatbot.

Building the Chatbot

To build out the chatbot, I will be following this tutorial by Sajil Rajput on Dev.to on how to build a chatbot with Flask and the ChatterBot library. For context, ChatterBot is a Python library that uses a selection of machine learning algorithms to produce different types of responses based on a user's input (Cox, 2019). When I was exploring various methods to train a neural conversational chatbot, I actually came across the ChatterBot library and decided to try it out - here's a Python notebook that documents my attempt. Unfortunately, despite yielding some very promising results, the ChatterBot solution was too out-of-the-box for me, and since I wanted to explore more native, hands-on solutions, I had to pass on this option.

To set up the chatbot, I first had to create the `app.py` and `model.py` files, which are essentially the same thing as the seq2seq model that I created earlier, except for the Flask component:

```
app = Flask(__name__, template_folder='./')

@app.route('/')
def main():
   return render_template('index.html')

@app.route("/get")
def get_bot_response():
   userText = request.args.get('msg')
```

```
    response =  pred(str(userText))
    return jsonify(response)

if __name__ == '__main__':
    app.run(debug=True)
```

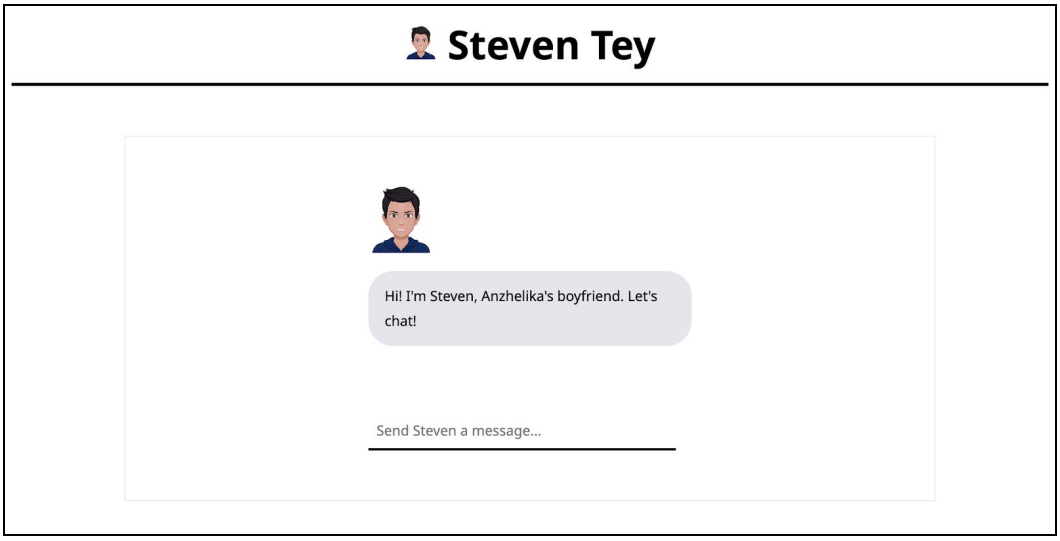
Then, I created a "models" folder that contained all the checkpoint files from the last iteration (`iteration 50000`) of the Seq2Seq model - this is what the chatbot will be using to generate responses to any given input. However, because of Minerva Forum's 50MB file upload limit, I couldn't include the data file (`pretrained_seq2seq.ckpt-50000.data-00000-of-00001`, which had a file size of 281.6MB), but you can download it here - and here's a screen recording on how to do that. Also, I've loaded the `wordList.txt` file into the "data" folder.

Lastly, I had to build out the front-end of the chatbot, which involves a bunch of HTML, CSS, and JavaScript code - I won't include the whole code here, but for your reference, you can check out the `index.html` file in the "chatbot" folder. If you'd like to run the chatbot's webapp locally, just download the "chatbot" zipped folder and unzip it. Then, by using Terminal (or Command Prompt if you're using Windows), navigate to the "chatbot" directory and run the following lines of code:

```
# Installing the required libraries
pip install -r requirements.txt

# Running the python webapp
python3 app.py
```

This should launch the webapp at localhost: 5000. Alternatively, if that doesn't work, here's a screen recording that shows the chatbot in action:



Evaluating the Results

Historically, the evaluation of dialogue and conversational systems has always been a vague, if not controversial topic. Unlike neural machine translation (NMT) and language modeling, where metrics like Bleu and perplexity are widely used to measure the accuracy of a certain translation model, there is yet to be a good automatic evaluation metrics that can accurately and effectively evaluate the performance of conversational models (Csáky, 2017).

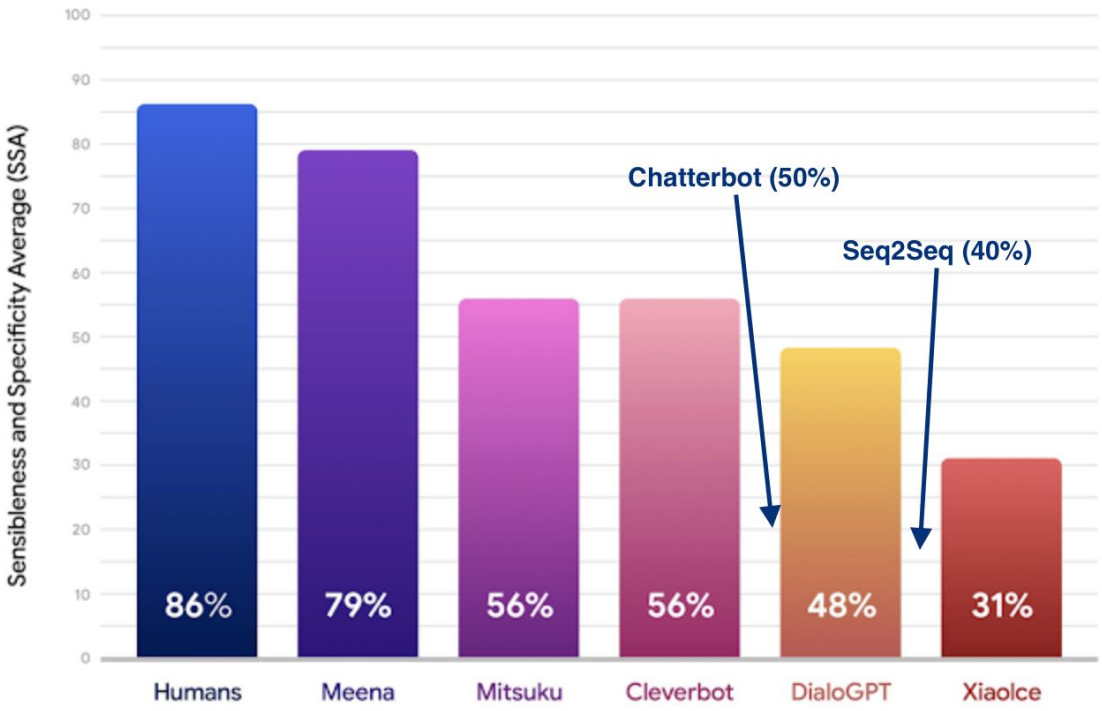
Therefore, a better way to evaluate dialogue and conversational systems is to ask actual humans what they think about the performance of the model, or in more advanced cases perform a Turing test to see if the AI chatbot can fool a human into thinking that they are human as well. Indeed, human judgment has been one of the most common metrics used to assess the performance of conversational models throughout recent literature (Shang et al., 2015, Vinyals and Le, 2015, Zhou et al., 2017). However, a more accurate and unbiased way of evaluating a neural conversational model's performance is the Sensibleness and Specificity Average (SSA), which is a new human evaluation metric developed by the AI team at Google. Here's how it works (method adapted from Adiwardana & Luong, 2020):

1) Run the chatbot through a set of the same set of prompts.
2) For each response that the chatbot gives to the prompt, the evaluator answers two questions:
    a) "Does it make sense?": If anything seems off - confusing, illogical, out of context, or factually wrong - then it should be rated as "does not make sense". If the response makes sense, the response is then assessed to determine if it is specific to the given context.
    b) "Is it specific?": For example, if A says, "I love tennis," and B responds, "That's nice," then the response should be marked "not specific" since that reply could be used in dozens of different contexts. But if B responds, "Me too, I can't get enough of Roger Federer!" then it is marked as "specific" since it relates closely to what is being discussed.
3) Calculate the SSA score: The *sensibleness* of a chatbot is the fraction of responses labeled "sensible", and *specificity* is the fraction of responses that are marked "specific". The average of these two is the SSA score.

Following this technique, I compiled a list of 10 different unlinked prompts (the first prompt doesn't relate to the second and vice versa) and ran the test on my model. Then, I ran it on the model that I created with the ChatterBot library. Here's a video recording of the test, and here are the results (*'S' stands for sensible and 'C' stands for context/specificity*):

Prompts	**Seq2Seq's Response**	**S**	**C**	**ChatterBot's Response**	**S**	**C**
Hi	You	✖	✖	How are you doing?	✓	✖
I love you baby	I love you so much I love you so much moreeee	✓	✓	I love you so much moreee	✓	✓
How are you?	You are so glad you know it	✖	✖	I am on the Internet.	✓	✖
Yes baby?	You are my friend baby I love you so much	✓	✖	Have an amazing day with your friends!! Good later	✓	✖
You mean so much to me	You	✖	✖	Really I do?? You mean the world to me too	✓	✓
What the heck	You are the best You	✓	✖	hahaha Dont askkik	✖	✖
Do you love me?	I love you so much moreee	✓	✓	its called Bella Ciao!! so good	✓	✖
What are you doing?	You You	✓	✖	kisses you passionately Ew my face	✖	✖
I've missed you	You are my friend baby boo	✓	✖	kisses you passionately I miss you moreeee and I need you so much	✓	✓
I'm hungry	You are my friend You	✖	✖	The pot is almost empty Nope	✖	✖
Final Score	$\frac{8}{20}$ = 40%			$\frac{10}{20}$ = 50%		

If we compare these scores to the performance of world-class neural conversational models like Meena, Mitsuku, Cleverbot, XiaoIce, and DialoGPT, as well as the SSA scores of humans, we see the following contrast (refer diagram in the next page). Clearly, both the Seq2Seq and ChatterBot models are not the best neural conversational models out there, but an SSA score of 40% for a basic Seq2Seq model is pretty impressive in itself.



Meena Sensibleness and Specificity Average (SSA) compared with that of humans, Mitsuku, Cleverbot, XiaoIce, and DialoGPT.

Source: Google AI Blog, 2020

Also, it is interesting to note that although ChatterBot had a slightly higher SSA score than my Seq2Seq model, the tradeoff is that it takes forever to get a single response - ChatterBot had an average response time of ~120 seconds, while my Seq2Seq model takes less than a second to get a response.

One of the reasons why the neural conversational model I built had such a mediocre SSA score is because of the noise in the dataset. Although I was able to get rid of some weird tokens in the original chat corpus, I wasn't able to standardize the dataset in such a way that is conducive for model training. For instance, my girlfriend and I would sometimes send really long messages - up to 500 words long - and at other times, we would send really short messages like "cool" or "yup" or even just a ".". This led to inconsistencies in the dataset, which negatively affected the quality of the model. After doing some research, I realized that I wasn't the only one who faced this problem. For example, large datasets like the Ubuntu dialogue corpus are very noisy, and thus, even the big players like Google, Microsoft, and Facebook always rely on manual data collection, which makes the resulting machine learning model much more efficient (Prassanna et. al., 2020).

Also, another reason could be due to overfitting. Since I trained the model on 5 very specific test strings that were specified in the `Seq2Seq.py` file - “hey baby”, “i love you so much”, “you la”, “do you love me”, and “hahaha I don’t like youuu” - the model could only react properly those phrases. Thus, if we put any other phrase in it, it wasn’t able to output responses that were appropriate to the context.

Here are a few ways that I could improve the performance of my model:

- **Pre-process the corpus even further** - get rid of overly-long and overly-short messages, eliminate jargon and words that have really low occurrences, etc.
- **Taking into account conversation breaks and inconsistent dialogue sequence.** For instance, when we stop talking for a bit and resume our conversation, we tend to be talking about different things. Also, since we are both fast-typers, my reply to her message might come a few messages after the original message, and therefore thinking of ways to account for scenarios like this would definitely increase the model’s accuracy.
- **Further tuning the hyperparameters** - right now, I’m training the model for 50,000 iterations, what if we made that 100,000? 500,000? Also, there’s been research saying that stacking LSTM hidden layers makes the model deeper and is a form of representational optimization (Brownlee, 2019). Thus, instead of using just 3 LSTM layers, which is what I’m doing right now, why not increase it to 5? 10?
- **Increase the number of test strings** that were used to train the model - instead of just using the 5 strings mentioned above, incorporate different strings coming from different contexts to help the model further understand the nature of human speech, or more specifically, *my* speech.
- **Using a separate validation dataset**, which gives us a more accurate representation of model performance (Hinno, 2019). After all, without a validation set, how do we know that we are not overfitting the original conversation data?
- **Using more advanced deep learning techniques** like bidirectional LSTMs, attention mechanisms, and bucketing (Deshpande, 2017).

References

Adiwardana, D. and Luong, T. (2020). *Google AI Blog: Towards a Conversational Agent that Can Chat About…Anything*. Retrieved from https://ai.googleblog.com/2020/01/towards-conversational-agent-that-can.html

Cox, G. (2019). *About ChatterBot.* Retrieved from https://chatterbot.readthedocs.io/en/stable/

Csáky, R. (2017). *Deep Learning Based Chatbot Models*. Retrieved from https://arxiv.org/pdf/1908.08835.pdf

Deepan, S. (2017). *Practical seq2seq.* Retrieved from http://complx.me/2016-12-31-practical-seq2seq/

Deshpande, A. (2017). *How I Used Deep Learning To Train A Chatbot To Talk Like Me (Sorta).* Retrieved from https://adeshpande3.github.io/adeshpande3.github.io/How-I-Used-Deep-Learning-to-Train-a-Chatbot-to-Talk-Like-Me

Hinno, R. (2019). *Tuned version of seq2seq tutorial*. Retrieved from https://towardsdatascience.com/tuned-version-of-seq2seq-tutorial-ddb64db46e2a

Joshi, A. (2017). *Learn Word2Vec by implementing it in tensorflow*. Retrieved from https://towardsdatascience.com/learn-word2vec-by-implementing-it-in-tensorflow-45641adaf2ac

McCormick, C. (2016). *Word2Vec Tutorial - The Skip-Gram Model*. Retrieved from http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/

Prassanna et. al. (2020). *Towards Building A Neural Conversation Chatbot Through Seq2Seq Model*. Retrieved from http://www.ijstr.org/final-print/mar2020/Towards-Building-A-Neural-Conversation-Chatbot-Through-Seq2seq-Model.pdf

Rajput, S. (2019). *Build a Chatbot using Flask in 5 minutes*. Retrieved from https://dev.to/sahilrajput/build-a-chatbot-using-flask-in-5-minutes-574i

Robertson, S. (2017). *Practical PyTorch: Translation with a Sequence to Sequence Network and Attention*. Retrieved from https://github.com/spro/practical-pytorch/blob/master/seq2seq-translation/seq2seq-translation-batched.ipynb

Shang, L., Lu, Z., and Li, H. (2015). *Neural responding machine for short-text conversation.* Retrieved from https://arxiv.org/pdf/1503.02364.pdf

Tensorflow Org. (2020). *Word embeddings*. Retreived from https://www.tensorflow.org/tutorials/text/word_embeddings

Vinyals, O. and Le, Q. (2015). *A neural conversational model.* Retrieved from https://arxiv.org/pdf/1506.05869.pdf

Zhou, H., Huang, M., Zhang, T., Zhu, X., and Liu, B. (2017). Emotional chatting machine: Emotional conversation generation with internal and external memory. Retrieved from https://arxiv.org/pdf/1704.01074.pdf

HCs Used

#optimization: By running the Seq2Seq algorithm over 50,000 iterations until it produced a sensible response to a given prompt, I essentially was applying the principles of optimization - which is a constant theme in machine learning and deep learning projects.

#variables: When building the algorithm to pre-process the Telegram data, I had to define and keep track of multiple variables like `sender_name` and `just_message`. Also, when I was building the Seq2Seq model, I had to identify and tune all relevant variables and hyperparameters such as the `batchSize`, `lstmUnits`, and `numLayersLSTM`.

#emergentproperties: Due to the multiple LSTM layers, the Seq2Seq model in itself is a perfect representation of a complex system full of emergent properties - where the sum of the inputs does not equal the output. As a result, some of the responses generated by the model were rather odd and unpredictable.