BUILDING A CONVERSATIONAL CHATBOT IN PYTORCH

Deep Learning for Sequence Analysis
City, University of London
MSc Artificial Intelligence
706 Coursework

Tommaso Capecchi - Vittoria Castelnuovo
Tommaso.Capecchi@city.ac.uk - Vittoria.Castelnuovo@city.ac.uk

Table of Contents:

Introduction

The field of Natural Language Processing (NLP) has seen some very significant advances in recent years, in part due to the introduction of Sequence to Sequence (Seq2Seq) models. This branch of deep learning models uses Recurrent Neural Networks (RNNs) to provide state-of-the-art solutions to a myriad of NLP problems, such as: machine translation, speech recognition, image and video captioning, text summarization, text mining, sentiment analysis and, most recently, conversational chatbots.

Chatbots, also referred to as Conversational Artificial Intelligence systems, have become extremely powerful tools which process natural language and its embedded meanings, and aim to return a sensical response to the user. Some popular chatbots are Amazon's Alexa or Apple's Siri, which have been programmed as virtual assistants. However, chatbots may have many distinct scopes, to solve different problems. For instance, some are employed for automating customer service, others have a question-answer nature, and others are employed for companionship.

The first chatbot, named ELIZA, was first introduced in the mid-1960s by Joseph Weizenbaum, from the MIT Artificial Intelligence Laboratory [1]. Eliza would analyse certain input keywords and trigger a response from a set of rules it was given. Subsequently, many different variations of chatbots were developed to fit specific user needs and problems, for instance a chatbot for psychotherapeutic help. Chatbots have continued to be improved in a quest to strive for human- level interaction, however none have yet passed the Turing Test.

The aim of this project is to create a companionship conversational chatbot, which can correspond to and converse with a user in a sensical and intelligent manner. The bot will be trained on the Cornell Movie Dialogue Corpus [2], which contain over 220,000 dialogues taken from movies of different genres. The implementation will be programmed in Python and will employ the Pytorch library. This report will outline the construction of a Recurrent Neural Network (RNN) with Long Short-Term Memory (LSTM) cells, the Attention technique and the Teacher Forcing method.

As a final remark, we provide the repository with the available code on github. Although Tommaso is the owner of the repository, both members of the team equally contributed to the final version of the project. In fact, in order to give an equal contribution to the task, each member of the team worked individually on their own implementation and subsequently the two we merged to elect the best aspects from each member's implementations. We also provide the pretrained model which can be download from this online repository.

During the development phase we consulted several external resources apart from the slides and papers suggested during the lectures, such as: on LSTM Networks, Chatbot by Pytorch, this repository on github and finally this repository. We used these materials to fully understand the details behind the implementations of such networks so that we could implement our own version. Overall, the Chatbot model is our own implementation, borrowing a small percentage of the code from the previous resources (approximately 5%) and we used frequently Stackoverflow when we had to debug and solve minor implementation aspects, mostly related to Pytorch functionalities.

Scope of the Project

This chatbot is a conversational chatbot, meaning that it is designed to respond and interact, converse in a natural human like manner. This type of chatbot differs to transactional chatbots which are usually pre-configured to interact with the user in more systematic and predefined ways. The aim of our chatbot is for it to learn and interact in conversation in a human-like manner even though the user's specific question or input is not one the machine has previously encountered in the training distribution.

Related Work

Sequence to sequence models have quickly become the state-of-the-art in deep learning for sequence analysis. In fact, these models solve a vast number of problems, from language translation to speech recognition. This branch of machine learning technique was first introduced by Google in 2014 [3]. The model was described with various components, namely: an encoder and a decoder network, with LSTM cells. These two networks transform the input into a hidden vector including the sequence and its context, and a decoder network which inverses this procedure. The LSTM cells are used to retain important information regarding the sequence's past input, in an effort to retain as much useful information as possible, as well as combatting the vanishing gradient problem.

Description of the Dataset

The dataset employed to train the chatbot in this project is the Cornell Movie Dialogue Corpus. This distribution contains 220,579 fictional conversational exchanges between 10,292 pairs of movie characters; 9,035 distinct characters from 617 movies; with a total number of 304,713 utterances.

The dataset also contains some interesting metadata such as the genre of a movie, its release year, its IMDB rating – however, these datapoints will not be used in this project.

The files which will be utilized are the movie_lines.txt file and the movie_conversations.txt file. The former is a text file which includes each utterance, where each has a line ID code, a character ID, a movie ID and the character's name. The second file contains the structure for each conversation, i.e. pairing the movie lines to the character they play in a dialogue. This file contains a list of each of the ID of the characters involved in a conversation/dialogue, as well as the movie ID (the movie in which the conversation took place), and a list of utterance.

Data Pre-Processing

```
u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L276', 'L277']
u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L280', 'L281']
u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L363', 'L364']
u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L365', 'L366']
u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L367', 'L368']
u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L401', 'L402', 'L403']
u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L404', 'L405', 'L406', 'L407']
u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L575', 'L576']
u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L577', 'L578']
u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L662', 'L663']
u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L693', 'L694', 'L695']
u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L696', 'L697', 'L698', 'L699']
u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L860', 'L861']
u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L862', 'L863', 'L864', 'L865']
```

Figure 1: Sample of dialogue pairs stored in the 'movie_conversations.txt' file. The only relevant information for this project's implementation is the final list of elements

Before parsing the data into the chatbot, it was pre-processed. The Cornell Corpus Dataset stores each movie dialogue by assigning a unique movie line ID. These IDs can be used to access the corresponding list of characters for each dialogue, found in the 'movie_lines.txt' file. This enabled us to build a dictionary to map each unique ID to its corresponding sequence of characters that form a movie line. We then isolated each movie line and subsequently formed the respective pairs of 'Question' and 'Answer'. These will be used to form each batch during the

training phase, where each pair of Q&A will be stored in a list called *pair_dialogs_idx*. Finally, now having access to each character of each movie line, we could build the *Vocabulary* class required to store each word. The main scope of the vocabulary is to tokenize each word so that it can be mapped to a numeric value and later embedded in order to be processed by the model. In fact, this class was used to encode and decode the input in the model to produce the corresponding output. We decided to remove regular punctuation, excluding characters such as ".,!?'", because we noticed they could compromise some of the chatbot's replies. Lastly, while building the vocabulary, some Q&A pairs were filtered out from the collection because either the question or the answer contained just an empty space character (' '), thus leading to an error later in the training phase. At the end of this process, 64 Q&A pairs were filtered and the vocabulary contained *48091 words in total*.

Subsequently, we built the *CornellCorpus* class, which holds all the functionalities for the dataset used in this project. This class, which inherits the Pytorch's *Dataset* class, is mainly responsible for building and splitting the data in three parts: training, validation and testing, 80%, 10% and 10% respectively. Each datapoint was tabulated as a list of two components (the question and the answer, respectively), used to form each batch. In addition, because each sentence may vary in length, we decided to fix a maximum length and trim or pad sentences that were longer or shorter. Finally, this class inherits two important functions from Pytorch's *Dataset* class: the *__getitem__* and the *__len__* methods. the *first function* retrieves a single pair of Q&A at a specified index, and the *latter* simply returns the number of data points in the dataset. These two functionalities will be used by the data loader in order to properly build the mini batches used in the training, validation and test phases.

Once the dataset was properly built, we implemented three different data loaders (*train_dataloader*, *val_dataloader*, *test_dataloader*) to properly generate and load batches of datapoints for each set.

```
#### Sample Datapoint - 1 ####
Question: [1, 22580, 2, 0, 0, 0, 0, 0, 0, 0, 0]
Answer: [1, 30795, 24055, 26123, 13716, 47811, 136, 39470, 47811, 1394, 20447, 2]

#### Sample Datapoint - 2 ####
Question: [1, 28870, 13009, 46718, 2, 0, 0, 0, 0, 0, 0]
Answer: [1, 47811, 3827, 28870, 24279, 19505, 43285, 27695, 2, 0, 0]

#### Sample Datapoint - 3 ####
Question: [1, 17630, 14175, 38613, 2, 0, 0, 0, 0, 0, 0]
Answer: [1, 20550, 47811, 46333, 43151, 24055, 42592, 27261, 37626, 42604, 24764, 2]

#### Sample Datapoint - 4 ####
Question: [1, 46934, 11387, 47811, 5894, 26123, 2, 0, 0, 0, 0]
Answer: [1, 4840, 47811, 47514, 47366, 18765, 31920, 18765, 42592, 36386, 46718, 2]

#### Sample Datapoint - 5 ####
Question: [1, 39298, 3322, 21933, 43151, 42604, 10441, 2, 0, 0, 0]
Answer: [1, 46718, 24713, 24055, 26123, 45219, 42604, 8750, 15953, 4, 15201, 2]
```

Figure 2: A sample of processed Q&A pair. Each sentence has been padded with zeros if necessary. All of the pairs start with token index (SOS) and end with index 2 (EOS). Index 0 is the pad token.

Methodology

Sequence to Sequence Model

In this section we present in detail the model that we implemented to solve the challenge we proposed. We begin by introducing the core architecture of RNNs. These networks are a particular family of Neural Networks which are able to process sequential data. Most recently, RNNs are commonly used in Natural Language Processing tasks as these focus on reading temporally linked sequences of data. In order to achieve this

behaviour, RNNs can be conceptualized as a series of networks which take x_t as input and propagate it forward through some computational model, which will output some values along with some newly discovered information at a particular time step *t*.

Despite their superior capabilities with respect to traditional NNs, vanilla RNNs' performance is hindered due to long-term dependencies. Specifically, useful context information between each time step may be lost due to limited memory. Another downfall of these models is their risk of suffering from the vanishing gradient problem.

In fact, if a particular gradient is relatively small, during backpropagation, this value may become saturated and result useless for the network's learning. This is because the process of backpropagation sees the gradient being multiplied throughout several layers, which ultimately may lead it extremely close to zero, thereby unable to yield a sufficiently significant update to the weights. This process is similar for the exploding gradients.

A solution to this problem was first presented by Sepp Hochreiter and Jürgen Schmidhuber [4] with the Long-Short Term Memory mechanism which allows the network to memorize information for long periods of time.

Long Short-Term Memory

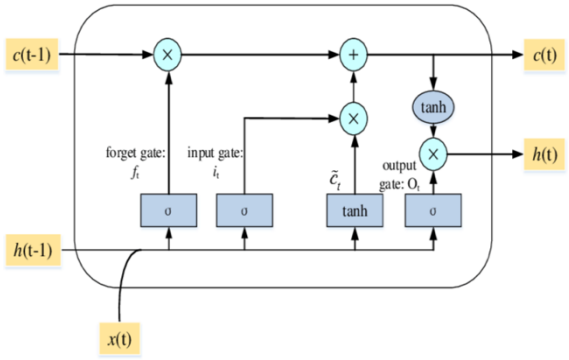

Figure 3: A detail view of an LSTM module. [8]

LSTMs present the same recurrent structure of RNNs. However, they attempt to filter information from past experience, memorizing what is relevant and dismissing what is not. This mechanism is achieved through a series of internal networks that interact and propagate forward relevant information used at the next time step. Specifically, there are four relevant NNs called gates, which implement this specific mechanism. Moreover, there are two important components: the *cell state* which simply transports information along the whole LSTM module, and the *hidden state* which holds information about previous data the network has observed. We now describe the internal mechanisms which regulate the propagation of the information through an LSTM.

Forget Gate

This gate represents a neural network which has the ability to "*forget*" previous information. As shown in Figure 3, the input x_t is concatenated with the previous hidden state (h_{t-1}). This new information is then processed by the NN using the Sigmoid function as activation, thereby yielding values between 0 and 1; the closer the value to zero, the more the information will be forgotten.

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \qquad (1)$$

In equation (1) we can see the neural network that implements this functionality.

Storing the Information

At this stage, a new layer is used to store the novel information. This is achieved by a NN, which is responsible for deciding what information will be stored in the cell state. This

network is mainly comprised of two parts: the first is called the *input gate*, which takes as input the concatenation of the input value x_t and the previous hidden state h_{t-1}, and passes it through a Sigmoid activation function. This process ensures the values to be in a range between 0 and 1.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \quad (2)$$

The second part, which uses the same concatenated input, is processed by another NN with a Tanh activation function. The scope of this model is to create a series of candidates C_t that could be potentially added to the cell state.

$$\tilde{C}_t = tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \quad (3)$$

Updating the Cell State

We can now combine a series of values to update the previous cell state and create the new cell state. Initially, the results from equation (1) are multiplied with the previous cell state C_{t-1}. This operation will select the relevant information to propagate forward in the LSTM module. Subsequently, we multiply the output of the input gate expressed by equation (2) with the output of the NN from equation (3). Finally, the results are summed to form the new updated cell state C_t.

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \quad (4)$$

Output Gate

As a final operation, a new gate, known as the output gate, selects what information to send to the next LSTM module, formally defining the output of this particular time step t.

$$o_t = \sigma(W_o [h_{t-1}, x_t] + b_o) \quad (5)$$
$$h_t = o_t * \tanh(C_t) \quad (6)$$

Firstly, the Sigmoid function is applied again to the concatenated input of the previous hidden state and the input value x_t. This operation will filter the cell state so that only the relevant parts of it will be propagated forward (equation 5). Subsequently, the new cell state is activated by a Tanh function that outputs values between -1 and 1, thus regularizing the new cell state. Finally, by multiplying it with the output from equation (5), we obtain the new hidden state from equation (6).

Implementation of the Model

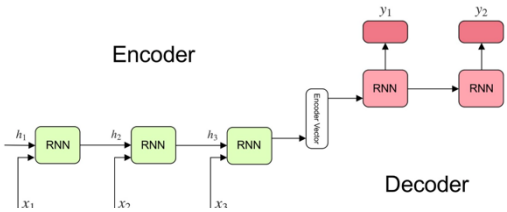

Figure 4: A basic representation of a Seq2Seq model.

In this section we present the model we developed to solve the proposed challenge. Our approach was to build a Seq2Seq model, composed by an Encoder and a Decoder. The Encoder is composed of an RNN with LSTM modules, which aims to process the input data (provided in a sequential fashion) and outputs a context vector which should encode the semantic information of the given input sentence. In this project, the input data is represented by a tokenized sentence signifying the start of a conversation. In contrast, the Decoder takes as input the context vector and produces a corresponding output at each time step t in a sequential manner, generating words based on the context it received. The output should be a meaningful reply to the first sentence, much like a natural conversation. Whenever the output of the decoder is the EOS (End of

Sentence) token, the process stops. As well as being tokenized, each input sequence was embedded so that each input word translates to a numeric representation which can actually be processed by the model.

The Loss Function

The Decoder must output a word from the vocabulary at each given time step t, therefore the problem converges to a multiclass classification problem. In our implementation, we applied a final Softmax layer to store the final predictions into a vector, where each value represents the probability of each word in the vocabulary. Ideally, the output vector should have the same length of the vocabulary. This is convenient because by simply taking the argmax of the vector we can make the prediction of the next most likely word chosen by the model.

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad for\ i = 1, \dots, K\ and\ z = (z_1, \dots, z_K) \tag{7}$$

Equation 7 : The Softmax equation

We then use those predictions to compute the actual loss function which is the Cross-Entropy loss expressed by the following equation.

$$H(p, q) = -\sum_{x \in X} p(x) \log \mathrm{q}\,(x) \tag{8}$$

Equation 8: The CrossEntropy Loss function. q (x) is the probability distribution of the predictions, while p(x) is the probability distribution of the ground truth.

The *CrossEntropyLoss* class, which Pytorch readily provides, takes as input the logits computed by the final fully connected layer of the decoder. Moreover, it automatically applies a Softmax layer and computes the Negative Log Likelihood loss function. As minimizing the Cross Entropy function is equivalent to maximizing the Negative Log Likelihood, we used this class to compute the loss needed to be backpropagated through time (BPTT), in order to update the model's weights. We decided also to modify the loss function so that words that corresponds to the '*<PAD>*' token are not taken into account during gradients computation. This can easily be done by setting the parameter '*ignore_index='<PAD>*' when instantiating the the *CrossEntropyLoss* class.

Attention Mechanism

Although the Encoder-Decoder structure of NNs works efficiently for many NLP tasks, it also has a few drawbacks. It is clear that as a network gets "deeper", it becomes harder to train. In the case of RNNs, the network extends through time, i.e. the longer an input sequence is, the more hidden layers the network will have. In turn, this affects the effectiveness of the backpropagation mechanism to update the weights. If the network is too deep, the initial layers' weights will become quickly saturated because the gradient's value will become extremely small as it travels back to the start of the network. Even though LSTMs were created in an effort to actively counter this problem, this is still an important phenomenon to be aware of.

A solution to this problem was first introduced by Bahdanau et al. [5], who created the Attention mechanism. This implementation allows the decoder to focus on specific words in a sentence that carry more semantic information for the scope of the dialogue. The main

intuition behind this mechanism is to provide the decoder with one context vector per each input word, instead of just the single final hidden state of the encoder. Thereby, through a series of alignment weights, letting the decoder choose what is the most relevant information to focus on.

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j \quad (9)$$

Equation 9: context vector with attention mechanism.

By observing the formula, we can see how each context vector C_i can be computed by taking the weighted sum of the attention weights (α_{ij}) and a series of annotations (h_j). Furthermore, we can compute the attention weights as follows:

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{i,k})} \quad (10)$$

Equation 10: attention weights

where $e_{i,j} = a(s_{i-1}, h_j)$ is a small neural network which takes as input the previous hidden state (s_{i-1}) and the annotation relative to each input word (h_j) and outputs the so called *energy* values which measure how well the input at position j matches with the output at position i. Further improvements to this formula were recently proposed, such as the global attention by Luong et al. [6], however this project will only focus on implementing the original attention mechanism.

Teacher Forcing Technique

To maximize the efficiency of the training phase we implemented the Teacher Forcing Technique. With this mechanism, at each time step t, we are allowed to randomly choose between the generated and the target word to be the next input of the decoder. The validity of this technique has been widely accepted as this alternation can make the model converge faster to the optimal solution. In order to implement this strategy, we assigned a probability of 0.5 to choose the generated word and a probability 0.5 to choose the target word.

Experiments and Results

In this section the methodology used to build the experiments will be described, as two distinct executions of the model will be presented: one using the attention mechanism and another without it. Lastly, the obtained results will be compared and discussed.

Experiments

The following table presents the models used to execute the experiments.

Model	Embedding	Layers	Bidirectional	Hidden size	Attention
Chatbot_1	500	1	Encoder	256	True
Chatobot_2	500	1	Encoder	256	False

Table 1: Models used for the experiments

Chatbot_1 and Chatbot_2 were both implemented using Pytorch's framework. During the instantiation of the objects, the user is able to set up the parameters of the model to their liking, to facilitate experimentation. The models are essentially built from the same *ChatbotModel* class. This class also wraps around the implementation of the Encoder and the Decoder LSTMs. The model that uses the Attention mechanism (implemented in the *Attention* class) requires to be instantiated with the *EncoderAttention* and *DecoderAttention classes*.

Hyperparameters

The models were trained using the Adam optimizer with mini batches. The following table highlights the hyperparameters used during the training phase.

Hyperparameters	Values
Mini-Batch	64
Learning rate	1e-5
Weight decay	1e-5
Epochs	70
Max sequence length	10

Table 2: Summary of the hyperparameters

Results

In this section we present the results obtained from the experiments. We executed each experiment with the Adam optimizer. The following table shows the parameters used in each experiment

Hyperparameter	Value
Learning rate	1e-5
Weight decay	1e-5

Table 3: The hyperparameters for the Adam optimizer.

To train each model we used the University's Camber server. The model with attention mechanism took about 12 hours to trains, whereas the model without attention took about 8 hours.

Train and Validation Loss

The following figures compare the behaviour of the train loss against the validation loss for both models.

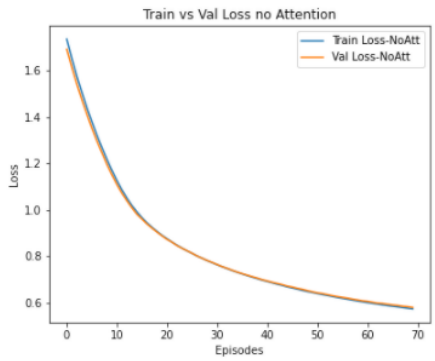

Figure 5: Train vs Validation loss for Seq2Seq model with no attention mechanism.

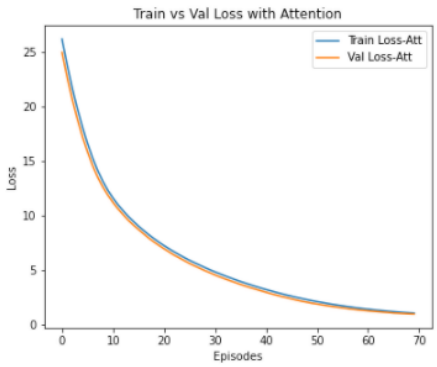

Figure 6: Train vs Validation loss for Seq2Seq model with attention mechanism.

Bleu Score

Currently, there is no standard metric to evaluate the performance of chatbots. This is because chatbots are quite complicated to train and small variations in the training stage can lead to different model performance. Apart from testing the chatbot's performance by interacting with it, we decided to calculate the Bleu score, which is a performance metric specifically used for the evaluation of machine-translated text. This metric can be used to measure text which has been generated by a machine, although it is not the standard metric to evaluate chatbots. In fact, it heavily penalizes the model in several conditions [7]. While observing Figures 5 and 6, we can see that despite the model being well trained it scores poorly on the Bleu metric.

Model	Bleu Score
Chatbot_1	1.37
Chatbot_2	2.3

Table 4: Bleu score for Chatbot_1 (with attention) and Chatbot_2 (no attention).

Graphical User Interface

As the scope of this project was to create a functional chatbot for companionship, we decided to develop a simple yet functional Graphical User Interface (GUI). This feature facilitates users' interaction with the bot, by eliminating the need to use the prompt. The GUI was implemented with the *tkinker* library in python.

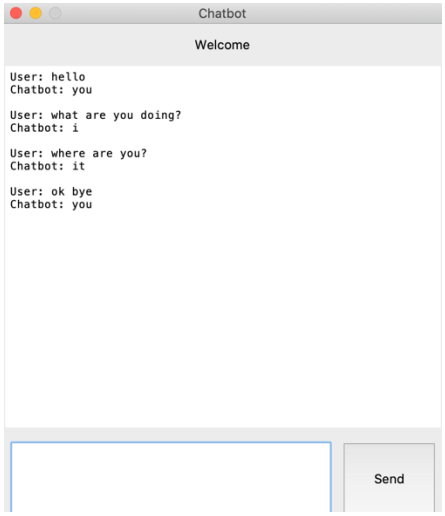

Figure 7: A screenshot of the GUI for the chatbot.

Conclusions

This project's aim was to build a conversational chatbot. To achieve this goal, a Seq2Seq model was implemented using an RNN with LSTM modules. The Attention mechanism was employed to enhances the model. This improvement was useful as it provided a way to process longer sentences more accurately. During the training phase, the Teacher Forcing technique aided the optimization of the learning process. To train our model, we used Pytorch's *CrossEntropy* loss function, which was modified to disregard the token '<PAD>' when gradients were computed. Finally, the Bleu score was computed to

measure the performance of the bot. Despite some good training/validation loss values, the model did not garner substantial results on the Bleu score metric. However, this may be justified by the fact that there exists no standard evaluation metric for chatbots, and literature suggests this is due to chatbots producing unstable results. Nevertheless, the chatbot implemented in this project shows a decent level of understanding when given as input particular sentences. However, overall, the system does not reach the standards of the state-of-the-art chatbots.

Reflections

In this project we were able to deeply understand how RNNs work and how to implement a Seq2Seq model using Pytorch. Furthermore, we enhanced our model implementing the attention mechanism and the Teacher Forcing technique. This enable us to understand various types of sequence analysis methodologies and gave us an in depth understanding of their respective advantages and disadvantages. The training process was the most challenging part to implement, given that Seq2Seq models are difficult to tune. To further extend this work, we believe it would be interesting to compare our LSTM chatbot and a chatbot implemented using Transformers, which have been proven to be more precise and accurate than LSTM.

Works Cited

[1] J. Weizenbaum, “A computter program for the study of natural language communication between man and machine,” *Communications of the ACM,* vol. 9, no. 1, pp. 36-45, 1966.

[2] “Cornell Movie-Dialogs Corpus,” [Online]. Available: https://www.cs.cornell.edu/~cristian/Cornell_Movie-Dialogs_Corpus.html.

[3] Google, “Sequence to Sequence Learning with Neural Networks,” 2014.

[4] H. Sepp and S. Jürgen, “Long Short Term Memory,” 1997.

[5] D. Bahdanau, C. Kyunghyun and Yoshua Bengio, “Neural Machine Translation by Jointly Learning to Align and Translate,” 2016.

[6] L. Minh-Thang, P. Hieu and M. Christopher D., “Effective Approaches to Attention-based Neural Machine Translation,” 2015.

[7] L. Chia-Wei, L. Ryan, S. Iulian V., Michael Noseworthy, C. Laurent and P. Joelle, “How NOT to Evaluate Your Dialogue System: An Empirical Study of Unsupervised Evaluation Metrics for Dialogue Response Generation”.

[8] Y. Xiaofeng, L. Lin and W. Yalin, “Nonlinear Dynamic Soft Sensor Modeling With Supervised Long Short-Term Memory Network,” *IEEE Transactions on Industrial Informatics,* 2019.