Generative And Retrieval Based Chatbot Using Recurrent Neural Network Long Short Term Memory Approach

Charmik Sheth and Viranchi Paliwal

Master Of Science in Computer Science, College Of Computer and Information Science

Abstract

Ranging from chat-bots and face recognizers to self-driving cars, various AI agents are now becoming a part of our everyday lives. Due to the accumulation of large amount of conversational data via social networks and micro-blogging sites, it has now become possible to train machines from that data and create conversational agents that can converse like humans. In this report we will be focusing on two different types of conversational agents namely retrieval based and generative based agent. After training on a particular data set a retrieval based chat-bot would always respond from a set of readily available responses whereas on the other hand the generative chat-bot would generate responses word by word after understanding the context of the input sequence. We will be implementing both agents using architectures designed from Long Short Term Memory (LSTM) units which are an extension of Recurrent Neural Networks (RNN). Here, retrieval based agent will be using dual encoder LSTM model while the generative will be employing sequence to sequence model. We will also discuss how LSTM models outperform feed forward deep neural networks. Finally, we test both the agents and analyze their responses. We conclude why retrieval based agent performs better than generative agent till date and also discuss the future of chat-bots.

Introduction

The ability of a machine to understand questions in natural language and give an appropriate response is one of the widely researched areas in the field of Artificial Intelligence. People have been developing various rule based conversational agents since mid-nineties and some of them like 'ELIZA' have been pretty successful in engaging users for a while. However, as they are hard wired based on a fixed set of rules, they cannot display any understanding of the context and creatively generate appropriate responses. Presently, a lot of research in the field of conversational agents is focused on developing and implementing deep learning architectures that combine the ideas from natural language processing and artificial neural networks. Although neural networks have been around since many years in the past they had not seen such popularity among researchers as present. Today, neural network based deep learning architectures are being applied to a wide range of problems like image processing, computer vision, language translation, etc. The primary reason behind the current success of these neural network models is the availability of enormous computing power which we didn't have before. With the advancements in fields like GPU computing, models which would have taken weeks to train less than a decade ago can now be trained in just a few hours. In the case of conversational agents, there is one more cause that has led to the shifting of focus from rule based agents to learning agents. Previously the research suffered from the lack of availability of enough training data which has changed nowadays due to the availability of large conversational public datasets.

In this report, we will be discussing about two conversational agents that we built using two different types of approaches namely generative based approach and retrieval based approach. Both these agents use learning architectures that are built using LSTM (Long Short Term Memory) units. LSTMs are an extension of RNN which are known to show decent performance on various natural language problems as they can model the context till any point in a sentence in the form of hidden states. For the generative based approach, we used Sequence to Sequence model and for the retrieval based agent, we used a dual encoder model. We used the cornell movie dialogues corpus for training and testing both of our agents and compared the results of both the agents in different scenarios. During this report, we will go into more details on how training and testing data were extracted from the corpus for both the agents. We will also give detailed descriptions on our model architecture and explain how training and testing were carried. Towards the end we would provide an analysis of the results generated by both the agents.

Related Work

Chat-bots have evolved a lot from early rule based agents like Eliza to human like chat-bots who learn from experience like Mitsuku. Currently researchers are experimenting on various neural network architectures to solve sequential problems like chat-bots. Deep Neural Network has emerged as quite powerful tool for various computer vision, image processing and classification problems, but in case of chat-bots it has a limitation that it cannot handle a sequential input where an input comes as a sequence of vectors instead of a single vector. For example in a chat-bot, input comes as a sequence of words and each word depends on the previous words. Deep Neural Nets expect the whole input to be encoded in vectors of fixed dimensionality. Vanila RNNs where each word in input sequence is embedded into a vector model this kind of problems better. However, vanilla RNNs are not capable of handling long range temporal difference. RNN LSTM model who are capable of handling long term dependencies are a perfect fit for a chat-bot. Apart from this, research is still going on other approaches like Variable Hierarchical Recurrent Encoder Decoder (VHRED) model for generating response.

Background

1) RNN

A recurrent neural network is a kind of artificial neural network where connected units form a directed cycle. In other words, for an input sequence S, the output of S[i-1] is given as a feedback input to RNN along with input S[i]. They perform the same task for every element of the sequence and this allows the output to be depended on the previous computations. Given below is a typical RNN shown with unfolding of the loop at the time of computation:

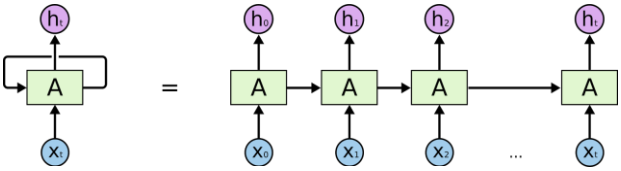

Figure 1: RNN with unfolding of the loop

In the recent years RNNs have been an incredible success to a variety of problems including speech recognition, language modeling, translation, image captioning. For an input sequence $(x_1,\ldots,x_t)$ RNN computes an output sequence $(y_1,\ldots,y_t)$ using following equations:

$$h_t = sigm(W_{ht}\, x_t + W_{hh}\, h_{t-1}\, y_{t-1})$$

$$y_t = W_{yh}\, h_t$$

Where h represents hidden state at time t , sigm represents sigmoid function and W represents weights at different levels.

2) Extension to LSTM:

Basic vanilla RNN are capable of using recent information to perform the task but there are cases where we need the whole context to make an inference. Consider an example of predicting the word 'Korean' in the sentence "I used to work in Korea …. I like Korean culture." In the provided example as the gap increases we need the context of Korea to predict 'Korean' accordingly. This is illustrated in the figure below where h_{t+1} depends on past context inputs x_0 and x_1 which occurred long before in past. Theoretically vanilla RNNs are capable of handling these long-term dependencies but practically as they are trained with Backpropagation Through Time (BPTT) which causes vanishing gradient problem which leads to difficulties learning them.

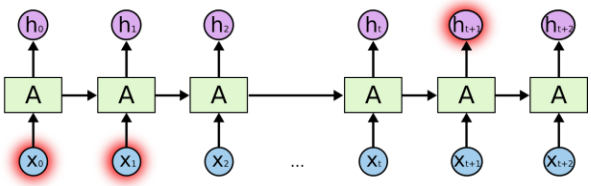

Figure 2: Current output may depend on the events that occurred long before in past.

Long Short Term Memory networks are special kind of RNN capable of handling long-term dependencies problem. Unlike standard RNN where the repeating module passes both previous context and current input through a single tanh layer, LSTM's repeating module has four specially designed neural network layers. The current input is passed through each of these layers and the corresponding outputs are either added or dot multiplied with the previous context. This allows for the previous context to be preserved for a longer term. The difference between a standard RNN cell and an LSTM cell is illustrated in the figure below.

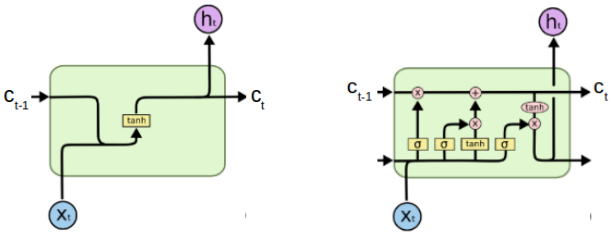

Figure 3: First figure shows a standard RNN cell and the second one shows an LSTM cell. In LSTM, the previous context c_{t-1} is not passed through any tanh or sigmoid layer.

3) Sequence to Sequence LSTM Model:

Sequence to sequence models have been a great success in recent years as they are capable of handling problems with sequential inputs whose lengths are unknown a-priori in contrast to other deep neural network which require fixed input and output dimensions. Examples of such real life sequential problems are generative chat-bots and speech

recognition systems. We are employing a sequence to sequence LSTM architecture to our chat-bot problem. The main concept behind this architecture is to make a large fixed-dimensional vector representation from input sequence using one LSTM (called encoder) and then use another LSTM (decoder) to extract the output sequence from that vector. The ability of LSTM based RNNs to handle long range temporal dependencies makes this architecture a good choice for a generative conversational agent. As shown in the below figure, Sequence to Sequence encoder reads input sequence in reverse order i.e. CBA and decoder generates output "WXYZ". Note that we add two symbols 'go' and 'eos' for denoting start and end respectively of a decoder input.

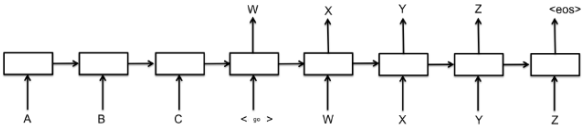

Figure 4: Sequence to Sequence encoder decoder architecture.

Hence, For an input sequence $x = (x_1,\ldots,x_{t.})$, the sequence is fed to the encoder element by element. As each element x_t is passed through the encoder, the encoder outputs a hidden state h_t . Also each element x_t is operated with the hidden state generated by the previous element x_{t-1}. Hence, at each time step t, h_t is represented as

$$h_t = f(h_{t-1}, x_t)$$

We call the final encoder output after passing all the elements in the input sequence through as encoder input summary c. Now this c is passed through the decoder generating an output y_1 and a hidden state h_1. y_1 and h_1 are then again fed as input to the decoder yielding another output y_2 and so on until y_{t+1} which would correspond to the 'eos' word. Now in this case, each decoder hidden state h_t would be a function of the encoder input summary c along with past output state h_{t-1} and the past decoder output y_{t-1}. Thus,

$$h_t = f(h_{t-1}, y_{t-1}, c)$$

Hence, the conditional distribution of decoder output symbol will be:

$$P(y_t \mid y_{t-1} \ldots\ldots y_1, c) = g(h(t), y_{t-1}, c)$$

So, each decoder output will be calculated via taking soft max on the available symbol probability. In the given example in (figure 4) encoder LSTM computes the representation of whole encoder input 'A, B, C' and then uses this representation to compute the probability of decoder output 'W, X, Y, Z'.

Project Description

Using the aforementioned LSTM based RNNs, we build two different types conversational agents namely generative based agent and the retrieval based agent. The generative based agent is designed using the sequence to sequence encoder decoder architecture we described above.

Each input question will be passed through the encoder decoder architecture to get a final generated response. For the retrieval based agent, we have used a dual encoder based architecture which basically encodes the given input question using the encoder which is same as in sequence to sequence architecture and generates encoder input summary c. Now, this being a retrieval based agent, we do not pass c through the decoder. Instead, we pass the likely responses picked from the corpus using a Tf-IDF based scheme through the encoder which would generate summaries $(r_1,\ldots,r_n)$ for each likely response. Now each of these summaries are compared with c and the one with the highest similarity is picked as the final response.

Dataset Details:

We used Cornell movie dialogs corpus this corpus contains a large metadata-rich collection of fictional conversations extracted from raw movie scripts including 9,035 characters from 617 movies along with 220, 579 conversations. We used this corpus because its a perfect fit for our conversational chatbot providing large data to trained, validate and test our chatbot.

Generative Based

Every neural network based model follows this procedure. First analyze and process the raw data make it compatible with the model, then divide the data into three parts training, validation and testing data. After that create a robust model, train it on training data and validate from time to time using the validation data. We keep training until the model starts giving decent performance on validation data. This policy of validating a model from time to time also ensures that we don't end up training so much that the model starts to over fit. Finally, we test our model on testing data. First we start with Data Processing.

Data Processing:

From the movie lines file in cornell movie corpus raw data we made a dictionary mapping line Ids to corresponding dialogues. Then we created a list of conversations from movie conversation file. Each conversation in this list would be a list of line Ids. We compared previously created dictionary with list and made lists of questions and answers. Hence questions[i] and answers[i] would form a question answer pair for any given index i. Then we filtered out unnecessary characters from all questions and answers.

We trimmed out all the questions and answers to the length of 25, then tokenized our questions and answers in words. After tokenization, we used nltk library to find frequency distribution of words and made a vocabulary of

size 8000 selecting most common words. We decided the size of vocabulary based on frequency distribution curve. In our case maximum area was covered at the selected size, also we added an extra word 'UNK' in our vocabulary to represent unknown words. After that we did indexing and made two dictionaries, one mapping words to their indices in vocabulary list and another mapping indices to words. Now we filter out questions and answers with too many unknown variables in it i.e. the ones where most of the words not present in our vocabulary that we created.

At the end using numpy we did the zero padding of the filtered-out questions and answers to make each question/answer of fixed length of 25 and converted them to numpy arrays. Finally saved these processed data ready to feed to our model in the numpy format and a metadata file containing dictionary of indexed lists along with the information regarding limits of words in question and answers and frequency distribution of words.

Learning Model:
Creating a training model is the most important step in a machine learning project. For any problem first we design a training model which is then trained and tuned until we get desired results.

For our generative chat-bot we implemented the sequence to sequence LSTM model that we discussed above using Tensorflow library. Our training code is roughly organized as follows:

1. First we will create place holders for our encoder input, decoder input and one for label which contains the expected output. As we have resized our questions and responses to a fixed size of 25, the size of each placeholder will be 25.
2. After defining place holders we define a basic LSTM cell with dropout wrapper on it. The utility of this dropout wrapper that it prevents the model from overfitting. This is also called unit activation. Inside the dropout wrapper we provide output keep probability 0.5 for our structure.
3. Also, we pass a parameter called embedding dimensions to our LSTM cell. Embedding dimensions specifies the number of dimensions that a vector associated with each word in the input sequence is encoded to.
4. As multilayer RNN gives better results, we stacked 3 layers of our basic LSTM cell. These stacked LSTM layers form our sequence to sequence encoder decoder architecture. Encoder and decoder inputs are then fed to this model.
5. During training phase, decoder inputs correspond to labels i.e the correct response for the given question appended with an extra word 'GO' in the beginning. However, during the testing phase we set a parameter called feed-previous in our sequence to sequence architecture.
6. We then initialize the loss function which would compare the decoded outputs with expected outputs i.e the labels.
7. Finally, we set the optimizer to be Adam optimizer which would try to minimize the sequence loss computed by the loss function defined in above step.

Training the Model:
While training our model, we take the processed data that we created and break it into 3 parts with 70% of it being the training data, 15% test data and 15% validation data.
We train our model by sending it batches of 32 questions and responses. We train our model for 40,000 epochs and record the mean loss for each epoch. We set the output dropout probability in our LSTM cells to 0.5 to prevent the model from overfitting. To check progress of our training model we validate it using validation data after every 10,000 epochs and measure its accuracy. Also, we save our trained model after every 10,000 epochs in the form of checkpoint (ckpt) files.

Testing:
For testing, we restore the last saved trained model from the ckpt file and set the dropout probability to 0 as we do not want any dropout from our decoder output. We feed questions from the test set to our model and generate responses for each of those.

Retrieval Based

For the retrieval based version, we have implemented dual encoder LSTM based deep learning model. We could have also used a variant of the sequence to sequence model that we implemented above, but as the dual encoder LSTM has been mentioned to give a decent performance for Ubuntu corpus, we decided to test the same for the cornell movie dialogues corpus. Figure 5 shows the model as described in the paper.

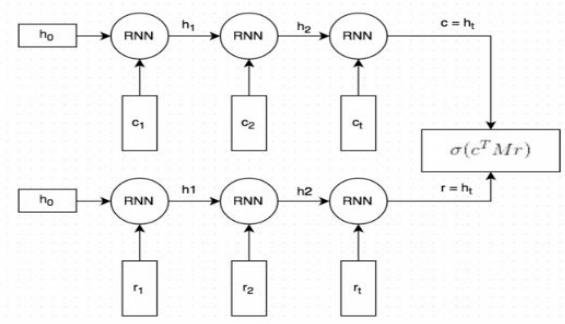

Figure 5: Dual Encoder LSTM based model.

As shown in Figure 5, both question and response are fed to an LSTM cell word by word which yields encoded question and response. The encoded question is then

passed through another. Before going into details of the working of this model, we must specify the representation of the training data that we generated from cornell corpus.

Training data:
As we are using the training model suggested for Ubuntu Dialogue corpus, we converted cornell corpus data to the same format as in ubuntu corpus. This new data consists of question response pairs such that half of the pairs have a correct response for the corresponding question and the other half have an incorrect response selected randomly from all the responses. The pairs with correct responses are labeled 0 and the ones with correct responses are labeled 1.

As in the case of Sequence to Sequence model, we first tokenized all the prompts and responses and removed punctuations and stop words from them yielding a list of tokenized prompts and another list of tokenized responses. After that we created a frequency distribution of all the words in the combined bag of tokenized prompts and responses. From this distribution, we picked and indexed top 7998 words creating our vocabulary. We kept the first and second entries (indices 0 and 1) in the vocabulary to represent special words 'PAD' (padding) and 'UNK' (unknown) respectively. Following this, we resized each conversation line to a fixed size 30. Hence if a line fell short of 30, it was padded with 'PAD' words and if a line was longer, it was trimmed. Also, if a word in the line was not present in vocab, it was replaced with 'UNK'. Further using the vocabulary, we transformed each resized line into a sequence of word indices where each index refers to a word in vocab. Along with this, we also created a sequence of labels that were originally attached with each context response pair.

In our algorithm, we are doing an early prediction using a Tf-Idf (Term frequency Inverse document frequency) based scheme to generate a set of probable responses. We then feed those to our dual encoder LSTM model along with the given question. This narrows down the number of candidate responses that the recurrent neural network has to choose from. Hence, along with the above training data for an lstm cell, we also generated an Idf map and a Cfd map using our tokenized prompts and responses. The Idf (Inverse document frequency) map maps each word from tokenized prompts to its cumulative Idf value in the collection. It tells about how relevant a word is in the given corpus i.e whether it is a common word or a word specific to just a few questions. It is mathematically defined as the logarithmically scaled inverse fraction of the documents that contain the word. It is computed using the following equation:

IDF(t, D) = log(N / | { d <- D : t <- d } |)
where D is the set of all documents in the collection,
N = | D | and t refers to a word in the corpus

The Cfd (Conditional frequency distribution) map maps each word in the corpus to a list of (response, likelyhood) tuples. It is populated for each question response pair by considering each question word to be equally responsible for the corresponding response. Hence the Cfd map is populated by appending (response, 1/question length) to the value of each word in the question. The figure below illustrates this process.

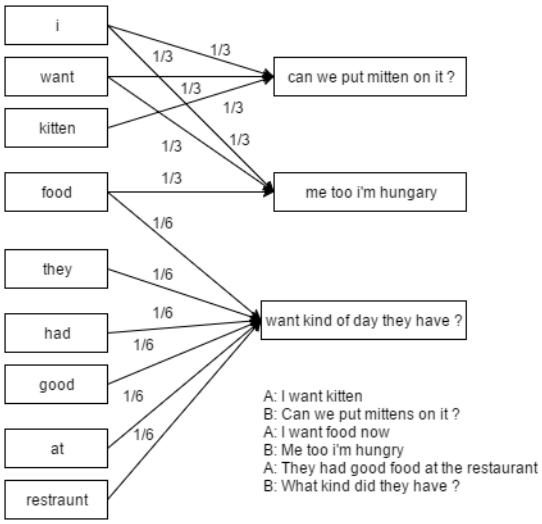

Figure 6: Example CFD Construction

Training the model:
We used ten
sorflow api for creating the dual encoder lstm based model. Our model shown in [fig] above is trained from the data as follows:

1. Question response pairs in the form of lists of word indexes are fed to our model in batches.
2. First each word in a question/response is embedded into a vector of 50 dimensions. This transforms a question/response into a list of vectors. The vector embeddings are initialized using a random initializer and are fine tuned during training.
3. Both question and response word vectors are then sequentially fed to a separate lstm cell with 128 neurons. This would result in two output vectors of 128 dimensions. We denote the vector corresponding to the question as encoded context and the one corresponding to the response as encoded response.
4. The encoded context is then passed through another hidden layer with 128 neurons which would generate a 128 dimensional generated response vector.
5. These generated response and encoded response are then compared for similarity by taking a dot product of them. This dot product is then converted to a probability by applying a sigmoid

function. Thus, while training we would get a probability value for each question response pair in the batch.

6. Finally, it computes the loss value using the probability generated in above step and the label (1 or 0) denoting whether the response is correct for the given question or not. Here, we use binary cross entropy loss function which is generally used for classification problems. Suppose p is the probability from above step and l is the label, then cross entropy loss is computed as loss = -l * ln(p) - (1 - l) * ln (1 - p).
7. It then back propagates to reduce the loss using stochastic gradient descent. We do the same for the whole data set which results in completion of one epoch. We train our model for about 20,000 epochs and then saved the trained model as a ckpt file.

Testing:
For testing, the previously saved trained model is restored and fed with a sequence of questions from the testing set. For each question, we tokenize it and predict a set of candidate responses using those tokens the Idf and Cfd maps we had generated while creating training data. We then pair the question with each of the candidate responses and run the pairs through our model. This would return a list of probabilities for each pair. The response from the pair with maximum probability is returned as the response to the question.

Experiments

For generative based chatbot model we trained our model on cornell corpus data for 12 hours and saved the model session at every 10,000 epochs. The output on test data of trained model improved with the increasing number of epochs. Output at 30,000 epochs start making sense and at 40,000 epochs responses were good and also loss didn't drop substantially. So, we stopped training our model further.

For retrieval based chatbot model we trained our model for 5 hours and saved the model session at every 5,000 epochs. Like generative model output on test data of trained model improved with the increasing number of epochs. Answer at 20,000 iterations was mostly matching with the provided questions on test data we saved from corpus data initially, as retrieval based model are expected to give predefined answers available in the dataset.

For testing our trained models, we firstly checked their performances on data we stored initially for testing phase. Then we tested them on random input including long sentences for testing our model specifically generative one as we know they are prone to make grammatical mistakes as the sentence size increases. Here's our outputs of both the models.

- q : well hes not exactly a friend
- retrieval : What do you mean?
- generative : How do you know

- q : please follow me
- retrieval : Down to my car.
- generative : why would i

- q : are you giving me this straight
- retrieval : Don't take my word for it. Get down there yourself and give a look.
- generative : no i dont want to

- q : may i help you
- retrieval : Yes.
- generative : yeah sure

- q: good morning dad
- retrieval : good morning
- generative : good morning

- q : i know you saw me
- retrieval : in new york?
- generative : you know what you mean

- q : i dont know booth
- retrieval : why not?
- generative : you dont know

- q : im going to have to front you personally dont lose that money without losing your life
- retrieval: What?
- generative: i dont about

- q : i really cant talk about it now the place is packed ive got a show to do
- retrieval : I know.
- generative : you are you

- q : who can we attack theyre all too big
- retrieval : If they have boats ... or canoes... they'd get lost in the fog. We can't move either -- we'll end up on the shore.
- generative : few it

Figure 7: A sample of experimental results

During training process for generative model, we also calculated the mean of batch loss after every 10,000 epochs where batch size was 32. We calculated the loss using tensorflow library where each loss represents average log-perplexity per symbol (weighted).

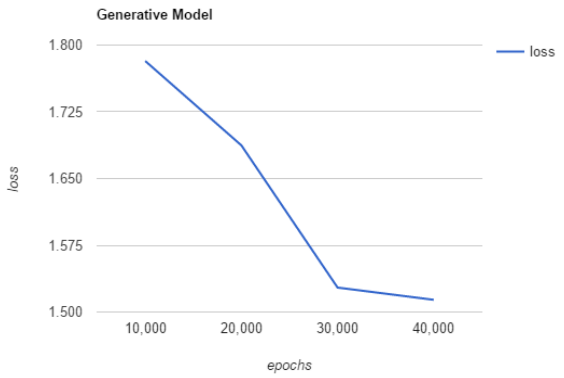

Figure 8: Epoch loss for our training model

Surprisingly, generative models performed well on long sentences questions the key reason behind this is firstly LSTM capability of handling long temporal differences and secondly reverse the encoder input which introduces short term dependencies making optimization simpler. Although, performance of generative model improved but could not outperform retrieval based models as they still have difficulty translating long term sentences. We also observed time required for training generative model is far more than retrieval based as they require shallower neural network and more training for performing adequately.

Conclusion:

Through our work we discovered that using sequence to sequence deep RNN LSTM with limited vocabulary for generative chat-bot outperforms vanilla RNN deep neural networks which use feed forward procedure. The capability of LSTM of handling long range temporal differences and approach of sending input word by word by embedding them into high dimensional vectors act as a key component in boosting the performance.

After analysing both chat-bot models generative and retrieval, we realised that till present day retrieval based approach outperforms generative models especially when length of question sentence increases. With long questions, generative models start making less sense and may even produce grammatical errors. That is why in the present scenario most chat-bots we come across are retrieval based and closed domain only. In coming future however, when more verticals of deep learning will be explored generative based agents will hopefully outperform retrieval based agents thereby providing human touch to the chatbot rather than feel of machine response.

References

Serban V., Sordoni A. et al. 2016 A Hierarchical Latent Variable Encoder-Decoder Model for Generating Dialogues.

Cho Kyunghyun, Bahdanau D, et al. 2014. Learning Phrase Representation using RNN Encoder-Decoder for Statistical Machine Translation.

Sutskever I, Vinyals O, and Quoc V.Le. 2014. Sequence to Sequence Learning with Neural Networks

Lowe R., Pow Nissan, et al. 2015. The Ubuntu Dialogue Corpus: A Large Dataset for Research in Unstructured Multi-Turn Dialogue Systems

Jafarpour S, Burges C, and Ritter A. Filter, Rank and Transfer the Knowledge: Learning to Chat

https://github.com/mikesj-public/rnn_spelling_bee/blob/master/spelling_bee_RNN.ipynb

http://courses.ischool.berkeley.edu/i256/f06/projects/bonniejc.pdf

http://www.wildml.com/2016/07/deep-learning-for-chatbots-2-retrieval-based-model-tensorflow/

https://www.tensorflow.org/tutorials/seq2seq

courses.ischool.berkely.edu/1256/f06/projects/bonniejc.pdf

www.wildml.com /2016/07/deep-learning-for-chatbots-2-retrieval-based-model-tensorflow/