A Chatbot that Understands

Team Chatbot
Chengshi Zhang
Paul Ngouchet
Sahil Sharma

Abstract

For this project, we have built a Website based informative AI device that tries to answer a user's input inquiries about different aspects of the hotel. Basically, the user opens the website, enters his/her question about the hotel, and the chatbot will provide responses accordingly.

1 Chatbot Design

1.1 Motivation

There are many approaches to build a chatbot.

Using a neural network, the most straightforward approach would be to input the user's question, and generate an answer as the output of the neural network. A large dataset of questions and answers will be used to train the chatbot, with the hope that when a new question is provided, the neural network can generate an appropriate answer for the question.

The good thing about this approach is that the answer is actually generated by the neural network instead of predefined sentences provided by the programmers. Also, it is a giant "brute-force" way of squeezing a response out of the neural network, so it is much simpler to execute. All we need to do is to feed a lot of data and perform very intense training to the neural network.

However, there are many downsides to this approach as well. First of all, this approach basically teaches the chatbot to "talk" or "answer a question" without understanding what it is actually talking about. It is pretty much piecing together words that are most likely to appear after the given question. On top of that, on the assumption that the neural network has already been intensely trained with massive data, there is still a big chance that the answer it provides does not make sense to the users, since it is just simply a collection of the most likely-to-appear words for the given question.

This is not what we want to do. We want the chatbot to actually "understand" the question, and then provide contextual and meaningful answers to the users. The most important aspect in solving this challenge is to help the chatbot understand what the user is asking about.

How do we go about building a chatbot smart enough to do that?

1.2 Dataset

First, in order to let the chatbot understand the context of user's questions, we have to modify our training dataset to teach it "context". Here is a small sample of our dataset:

```
{"contexts": [
        {"tag": "greeting",
         "patterns": ["Hi", "How are you", "Is anyone there?",
         "Hello", "Good day"],
         "responses": ["Hello, Welcome to Ice and Fire Hotel your grace!",
```

Submitted to 30th Conference on Neural Information Processing Systems (NIPS 2016). Do not distribute.

```
            "Warm welcome to Ice and Fire Hotel"]
        },
        {"tag": "goodbye",
         "patterns": ["Bye", "See you later", "Goodbye"],
         "responses": ["See you later, thanks for visiting",
         "Ciao, Have a nice day"]
        },
        {"tag": "cancellation",
          "patterns": ["can I cancel my reservation?",
          "what is your cancellation policy", "how do I cancel?"],
          "responses": ["learn more about cancellations here"]
        }
]}
```

For a chatbot framework like ours, we need to define a structure of contexts to handle different types of questions from the users. For example, in the portion of our dataset above, we have defined 3 types of context in which the users might have asked: "greeting", "goodbye", and "thanks". For every context, we will establish a pattern of commonly asked questions in this very context. For example, for the context "cancellation", we have included some of the most common types of questions regarding cancelling reservations, like "what is your cancellation policy" or "can I cancel my reservation". There are also some responses included with every context. These responses are the potential replies the chatbot might offer to users.

Why, you might ask? Why do we establish such a structure for the list of contexts? The answer lies in the neural network. We have built a "Feed Forward Neural Network" with tensorflow, which we will discuss in detail in section 1.4. To train the neural network, the "patterns" of commonly asked questions will be the inputs, and the "tags" themselves will be the target output for training, because that is what we are trying to achieve. We want to give a question, and find the context for that question. Ideally, when the user ask a question, we would like the neural network to find out which context the user's question belongs to. We want the neural network to output the context "check-in" when the user asks "When are you open". That is precisely why we train the neural network with the frequently asked questions in a given context, and hope that it would produce the correct context given a similar question.

And then, once we have obtained the context in which the user is asking questions about, we would reply with the responses corresponding to the context. There are many ways to handle the response, but due to time constraint, we simply attached multiple viable and proper responses to every context we have, and randomly choose one to reply the current user's question. If more time is given, we would love to implement smarter ways to produce the responses, like asking a follow-up question for more details, or checking previous conversations to find the most appropriate response to reply the user.

For the hotel-enquiry chatbot that we are building, we have established a dataset with 16 different types of contexts: "greeting", "goodbye", "thanks", "check-in", "check-out", "room", "multi_rooms", "Deluxe", "Standard", "Suite", "payments", "reservation", "cancellation", "pool", "spa", and "activities". Ideally, for a commercialized chatbot of an actual hotel, we would add a lot more contexts in order to handle every possible question that users might ask, but the current 16 types of contexts should be enough to handle the most frequently asked questions for our fictional hotel in the world of Game-of-Thrones.

Now that we have established the dataset, we are ready to teach the neural network. Before we actually feed the data into the neural network, there is something important we need to take care of.

1.3 Training

Before feeding data to the neural network, a question arises with our dataset: they are not numbers. How could we feed texts into a neural network, and produce texts as the output?

Before we get into the nitty-gritties of the great NLP(Natural Language Processing), let me introduce a concept called "word stemming". What this process does is basically strip the affixes off every word, so that only the root form of the word remains, which we call the "word-stem". We will perform "word stemming" on every word that has appeared in the dataset, and put them all into a large array,

which we call the "dictionary". The point of stemming the words is to prevent including different conjugation of the same word in our dictionary.

The process of generating this "dictionary" is using what we call the "bag-of words" technique. Now that we have a dictionary of all the words that have appeared in the dataset, we can use this dictionary to convert sentences to numbers. For example, let's assume our dictionary look like the following:

["are", "day", "good", "hello", "how", "you"]

This dictionary consists of 6 word stems. Also note that the array is sorted in alphabetical order.

Now, let us try to encode the sentence "how are you" with the dictionary given above. First, we will generate an empty array called "bag". Then, what we do is to loop through the entire dictionary, and see if the current word can be found in the sentence. If so, append a 1 to the "bag"; if not, append a 0 to the "bag". For example, in the first loop, we are looking at the word "are" in the dictionary. Since "are" can be found in the sentence "how are you", we append a 1 to the "bag", so that it becomes [1]. Then, during the second loop, we check whether the word "day" can be found in "how are you", and the answer is no. Therefore, we will append a 0 to the "bag", which now becomes [1, 0]. We keep doing this under we loop through the entire dictionary. In the end, the "bag" array in this case should become [1, 0, 0, 0, 1, 1]. You might have guessed it by now: this array is precisely how we encode our sentence.

We will produce a "bag" for every "pattern" sentence in our dataset, which will become the inputs to our neural network. As long as we have a fixed dictionary, we can encode any sentence consistently.

Now that our input sentences are encoded, how about our outputs? As you know, our output is a tag in which the input sentence belongs to. If our input is "how are you", then the neural network should give "greeting" as our output. The tags might look like an entirely different thing to encode, but it actually is not. All that we have to do is to make a new dictionary which consists of all the 16 tags we have, and we can use the exact same technique to encode our tags. For example, suppose we only have 3 tags in our dataset, which leads to the following dictionary for our tags:

["cancellation", "goodbye", "greeting"]

When we want to encode the tag "greeting" into numbers, all we have to do is the same as encoding the sentences, and yield the result array as [0, 0, 1]. This result array will be our target output when we train the neural network.

Now that we have a way to encode the input sentences and the output tags of our training dataset into arrays of 0's and 1's, all we have to do is to feed them into our neural network, and start the training.

1.4 Deep Feed Forward neural network

Design 4 layers and 3 biases.

The input layer has the size of the dictionary. The inputs are 1's or 0's depending on the words of the dictionary that are found in the data (The sentence) given by the user when he/she is interacting with our chatbot.

Two hidden layers have 8 hidden neurons each. This design of 2 hidden layers of 8 neurons each was achieved using cross validation, adding each neuron and hidden layer one at a time and testing for the accuracy each time. We chose the design that yields to the best accuracy. With this design our accuracy was over 90%.

The output layer has the size of the number of tags available in the data. It outputs a series of probabilities. Based on our predefined threshold, we can deduct the proper tag of which the input question belongs to.

We also have 3 biases for each hidden layers and the output layer. Their dimensionalities each time are the number of neurons for each layer, 2 hidden layers and output layer.

All our weight matrices were generated randomly following the Gaussian probability distribution with a standard deviation of 0.02. The neural network was built using forward propagation which uses the forward W' X + b to directly and each time a sigmoid function is applied to scale the output until the end. The softmax was not used because the output probabilities become too small after scaling

for our threshold to be the most effective. Then back-propagation is used to update the weights by minimizing the error Y - Y_predicted. we have a learning rate of 0.1. Our neural network was trained 150 000. It is not trained more than because we don't have lot of the data and we want to prevent over-fitting as much as possible. With this design our accuracy was over 90%.

We built our neural network using Tensor Flow instead of using other deep learning libraries like scikit learn that offers more abstraction. We wanted to maintain the most control over how our algorithm learns from the data.

1.5 Interfacing

With the powerful feed forward neural network, we are able to obtain a usable trained model. What we do now is to handle user's actual inputs and provide responses as the chatbot should do.

When the user enters a question to our chatbot, we will first stem each word in the sentence, and compare every word stem with our dictionary in order to convert the sentence into an array of 0's and 1's. And then, we will feed this array into our neural network as inputs, and compute the result. The result from the neural network will be 16 nodes of probabilities, since we have a total of 16 tags in our "context.json".

The 16 probabilities correspond to how we encoded the tags into 0's and 1's during training. Say that the "cancellation" tag is the very first tag in the array of all tags, then the probability outputted by the first output node will be how much our neural network thinks the input sentence is related to the "cancellation" tag.

Since we have computed a probability for every tag in the dataset, we will assume the tag that has the highest probability is the correct context for user's question. And then, all we need to do is to retrieve a proper response from the dataset that corresponds to the tag, and reply the user!

This is where the neural network's magic truly shines. If we did not use a neural network to implement the chatbot, we would have to search whether the user's exact question can be found in our database, and the chance of failing is very high, along with the massive performance drop due to the brute-force searching. However, since we have used a neural network and trained it with "contexts", even if we ask a question that does not exist in the dataset, most times, our chatbot will still produce the correct result. This is because we have trained the neural network with the common patterns of questions asked for every context there is for a hotel-enquiry chatbot.

At this point, we have a fully-functioning chatbot that answers user's questions regarding the hotel. But we want more. We want a finished product that can actually be used with ease, which is precisely why we built a beautiful website.

1.6 Website

As a way to interface our trained model, we built a MEAN stack website using Mongodb, ExpressJS, AngularJS, NodeJS. We also used html, css, and bootstrap for styling. Javascript manages the website and also interacts with the model and mongodb, a database to store the information. MEAN stack was used to build the website, but we needed a way to interact with our trained model and our python script in general. That is why we used a python package that allows the website to send requests and receive information from the script which are in turn displayed on the website.

The Core of the website:

The user inputs a text, which is saved in the database and used to execute the python script, which calls the trained model. The script returns a response, which is saved in the database. Then, the website simply displays the conversation by reading the database.

2 Future

This chatbot although smart and capable of answering relevant question, it can't perform function. Service industry involving the most amount of human interface, has the highest potential for chatbot use. But for that purpose the chatbot must be able to perform certain functions in accordance to the users requirements.

For instance taking the example of our chat bot working on behalf of a hotel, if asked, should be able to reserve a room for the customer it is serving. Having the ability to understand that the user wants, the chatbot will be able to access the website of the hotel and make the reservation. But for this to happen the chatbot needs to have a very high accuracy and almost no functioning errors. It needs to be taught the concept of context which we aimed at the start but were unable to embed into the program due to the limited time we had to complete the project.

Anyhow, ever after understanding the context the bot needs to smart enough to not make a mistake and ask the user confirmation questions or other sort of precautionary statements. For that as mentioned in the improvement for design section the bot needs to programmed more and better.

Although beyond the scope of our current knowledge, if somehow all the aspects of different functions the bot performs can be put together in one algorithm, the chatbot will be able to reduce a huge amount of human hours and efforts.

This service providing bot could be used in several fields like online travel reservation or online hotel reservation or in restaurant services, pretty much any situation where a human acts an information intermediary, the chatbot could serve the purpose instead.

3 Improvements

A big improvement on our design would be to have a much larger dataset which was hardcoded for this project. For the future we could find a way to automate the generation of our dataset.

Also as the dataset gets larger or different we might need to review the design and apply cross validation to find the best design and learning rate and number of times the model is trained. Adding more data and improving our neural network would be great, but it would be incremental innovation. In order to produce a state of the art product we might to completely discard this design.

4 New Future Design

This design where the response are already specified has its limits as it is not flexible, not easily scalable and we will have a hard time making it general purpose like the amazon echo, if we wanted too.

Instead, we will completely discard the feed forward neural network as they cannot really understand the context in order to generate answers that makes sense. In order to generate good answers, you need to understand which is done by taking in account the previous most of the word in the sentence. In that our model needs to be able to learn not only from the present input but it also needs to take in account the previous outputs. That is where recurrent neural networks are really handy, they are really good for sequence of inputs like sentence where every word is linked with each other

So why Recurrent neural network over feedforward recurrent neural network. One big reason is the ability to remember previous outputs.

Architecture of an Feed forward Neural Network

H(t) = W0' X only depends on the present input

Architecture of an Recurrent Neural Network

H(t) = W0' X + W'h H(t-1) depends on the present input and the previous output. And the previous output also depends on the output before it keeps going. With this ability we can see it would be impossible to fully understand the context of a sentence and be able to output good responses.

A more sophisticated version of it LSTM: Long Short Term Memory

5 How to Run

Considering that running the website requires installing lots of packages, we have prepared two versions of the chatbot: the website version, which requires installing lots of packages; and the python code version, which only requires python.

To run the website on Mac OS X:

1. Install MongoDB

`https://docs.mongodb.com/manual/tutorial/install-mongodb-on-os-x/`

2. Install Node and NPM

`http://blog.teamtreehouse.com/install-node-js-npm-mac`

3. Install Bower

`https://bower.io/`

4. Open Terminal
5. Enter command: mongod -dbpath $\sim /dataCS542\&$
6. In another terminal window, enter command: “cd /path/to/project”
7. Enter command: “npm install”
8. Enter command: “bower install”
9. Enter command: “npm start”
10. Open the website in the browser at “localhost: 3000”

To run the python code on Mac OS X:

1. Install Python3.6.2

`https://www.python.org/downloads/`

2. Open Terminal
3. Enter command: “cd /path/to/project”
4. Enter command “python response.py”
5. Follow the instructions from the program

5.1 Demo

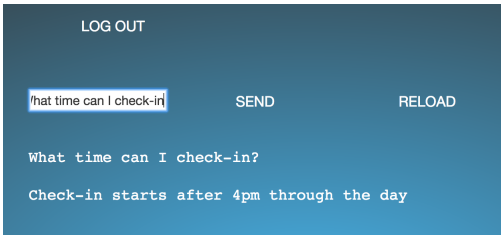

Figure 1: The chatbot website

This is an example of our website running.

As you can see, when the user has a question, simply enter the question to the textbot and hit "Send". The chatbot will make a response in a split second.

Acknowledgments

This research was supported and advised by Professor Sang "Peter" Chin from Boston University.

References

[1] Anonymous. *Bag of Words Meets Bags of Popcorn*. Kaggle. 30 June. 2015. Web. https://www.kaggle.com/c/word2vec-nlp-tutorial/details/part-1-for-beginners-bag-of-words

[2] Gupta, Tushar. *Deep Learning: Feedforward Neural Network – Towards Data Science – Medium* Medium. Towards Data Science, 05 Jan. 2017. Web. https://medium.com/towards-data-science/deep-learning-feedforward-neural-network-26a6705dbdc7

[3] Willems, Karlijin. *TensorFlow Tutorial For Beginners* DataCamp Community. 13 July 2017. Web. https://www.datacamp.com/community/tutorials/tensorflow-tutorial#gs.0zcoY=o