Natural Language Processing
Final Project

Domenico Alfano

1 Introduction

Telegram is a free, non-profit cloud-based instant messaging service. In June 2015, Telegram launched a platform for third-party developers to create bots. Bots are Telegram accounts operated by programs. They can respond to messages or mentions, can be invited into groups and can be integrated into other programs.
Users can interact with bots in two ways:

- Sending messages and commands to bots by opening a chat with them or by adding them to groups.
- Sending requests directly from the input field by typing the bot's @username and a query. This allows sending content from inline bots directly into any chat, group or channel.

A chatbot is a computer program which conducts a conversation via auditory or textual methods. Such programs are often designed to convincingly simulate how a human would behave as a conversational partner, thereby passing the Turing test.
Chatbots are typically used in dialog systems for various practical purposes including customer service or information acquisition. Some chatterbots use sophisticated natural language processing systems, but many simpler systems scan for keywords within the input, then pull a reply with the most matching keywords, or the most similar wording pattern, from a database.
The main goal of the project is to implement a Chatbot system that can answer questions and, at the same time, learn by asking questions itself. By the way, there are two types of interactions:

- Querying, where the Chatbot system takes a question by the user as input and outputs an answer.
- Enriching, where the Chatbot system asks a question about a concept/relation. It is then the task of the user to give an answer.

2 Chatbot

The first step to build a Chatbot system is to create a bot. To do this, I just had to talk to BotFather. It helped me, using the /newbot command, to create a new bot. Subsequently, the BotFather asked me for a name and a username. Then, it generated an authorization token for my new bot. Furthermore, the BotFather, has many other really useful commands as:

- /setdescription: change the bot's description, a short text of up to 512 characters, describing your bot.
- /setprivacy: set which messages your bot will receive when added to a group. With privacy mode disabled, the bot will receive all messages.
- /deletebot: delete the bot and free its username.

Once the bot was created, the next step was to develop a finite-state machine (FSM). The FSM is a mathematical model of computation. It is an abstract machine that can be in exactly one of a finite number of states at any given time.
The FSM can change from one state to another in response to some external inputs; the change from one state to another is called a transition. An FSM is defined by a list of its states, its initial state, and the conditions for each transition.
I developed it following these steps:

- Initially, the state is equal to 0 and it changes to 1 if and only if the command /start was typed in the conversation. If the condition is verified, the bot sends the message *"What do you want to talk about?"* showing in the keyboard the list of all domains. Selecting one of these items, automatically the user will send, by a message, his choice in the conversation.

- Once the user has chosen an element from the list and the state is equal to 1, the state is set to 2. Doing this, the bot sends the message *"Please, select the direction of interaction."* showing in the keyboard two items: Querying and Enriching.

- As I did before, at this point I have two ways to go. If the choice of the user is the Querying interaction and the state is equal to 2, the state is set to 3 and the bot sends the message *"Ok, send me a question!"*.
  Instead, if the user's choice is the Enriching interaction, it would be chosen a random relation between the possible relations, from *domains_to_relations.tsv* file, and a random word between the possible words, *from domain_subj.txt* file, for the previously selected domain. Having done that, the state is set to 10 and the bot sends two messages: the first one communicates which relation was chosen, the second one shows the question made by the bot using the patterns.

- Now, I have two possible values for the state. If the state is equal to 10 and the user give an answer, the fields c1 and c2 will be extracted from the question and the answer. Having done that, it will be added to the Knoledge-Base Server the new information, through the *add_item* function, in this way:*"question":"Q","answer":"A","relation":"R","domains":"D","c1":"c1" ,"c2":"c2"*. Having done that, the bot sends the message *"Thank you! Digit /start to restart the conversation."* and the state is set to 0. It is clear that, if the user sends the /start command, the coversation restarts. Instead, if the state is equal to 3 and the user sends a question, it will be taken as input from the Relation Classifier that will output which relation the question relates to. Then, the same question will be taken as input from the Context Classifier that will output which type of answer the question relates to. I have done that because, in my opinion, there are two types of question: the question whose answer may be yes or no and the question whose answer requires further information.

- At this point, if the context is equal to *yes_or_not*, the subject and the object will be extracted from the question through the *get_infomation* function. Successively, if the subject and the object are the subject and the object of a question in the Knoledge-Base and the relation associated to the question is the one predicted by the Relation Classifier, then the bot sends the corresponding answer and the state is set to 0 in order to start a new conversation through the /start command. If these conditions are not met, the bot sends the message *"I don't know it but if you know the answer, please, write it to me. If you dońt know the answer digit /restart"* and the state is set to 4.
  Instead, if the context is not equal to *yes_or_not*, the subject will be extracted from the question through the *get_infomation* function. Successively, if the subject is the subject of a question in the Knoledge-Base and the relation associated to the question is the one predicted by the Relation Classifier, then the bot sends the corresponding answer and the state is set to 0 in order to start a new conversation through the /start command. If these conditions

are not met, then the bot sends the message *"I don't know it but if you know the answer, please, write it to me. If you dont́ know the answer digit /restart"* and the state is set to 6.

- As in the previous case, at this stage, I have two possible values for the state. If the state is equal to 4 and the users sends an answer, as before, it will be added to the Knoledge-Base Server the new information, through the *add_item* function. After that, the bot sends the message *"Thank you! Digit /start to restart the conversation."* and the state is set to 0. Instead, if th user sends the command /restart, then the bot sends the message *"Digit /start to restart the conversation"* and the state is set to 0. Obviously, after the end of both processes, if the user sends the /start command, the coversation restarts.
  Instead, if the state is equal to 6 and the user sends an answer, firstly, the c2 field will be extracted from the answer through the *get_obj* function, and then, as before, it will be added to the Knoledge-Base Server the new information through the *add_item* function. Having done that, the bot sends the message *"Thank you! Digit /start to restart the conversation."* and the state is set to 0. Instead, if user sends the command /restart, then the bot sends the message *"Digit /start to restart the conversation"* and the state is set to 0. Obviously, as before, after the end of both processes, if the user sends the /start command, the coversation restarts.

It is important to specify that the library used to build the Chatbot is Telepot.

3 Classifiers

As mentioned before, I developed two different classifiers with a lot of similarities. In fact, both classifiers have more or less the same model. The simplest model is defined in the Sequential class which is a linear stack of Layers. There are two ways to create a Sequential model: the first one, is to create a Sequential model and define all of the layers in the constructor, the second one, is to create a Sequential model and add the layers in the order of the computation you wish to perform. I have choosen the second way to build my model.
The first layer in the model must specify the shape of the input. As first layer I choose the Embedding layer that turns positive integers (indexes) into dense vectors of fixed size. The Embedding layer takes in input a 2D tensor and provides as output a 3D tensor. This layer requires an input dimension that in my case is the lenght of my dictionary, the output dimension that is an arbitrary integer and an inpunt lenght that is the lenght of how many words I consider in a sentence.
Furthermore, I used the glove dictionary to pass the weights to the layer and for this reason I had to put as input the variable trainable equal to false. After the Embedding layer I add to my model a recurrent layer: The LSTM layer. An important benefit of recurrent neural networks is their ability to use contextual information when mapping between input and output sequences. The LSTM architecture consists of a set of recurrently connected subnets, known as memory blocks. Each block contains one or more self-connected memory cells and three multiplicative units, the input, output and forget gates, that provide continuous analogues of write, read and reset operations for the cells. In my model the LSTM layer has only one argument: the units, a positive integer that represent the dimensionality of the output space. After this, I need to add to my model a core Layer. There are a large number of core Layer types for standard neural networks and I have choosen a Dense layer.
The Dense layer is a fully connected layer and the most common type of layer used on multi-layer perceptron models. In my model the Dense layer, as in the case of LSTM layer, has only one argument: the units, a positive integer that represent the dimensionality of the output space.
Finally, to complete my model, I have added the activation function Softmax. Softmax Regression is a generalization of logistic regression that can be use for multi-class classification.
Once I have defined my model, it needs to be compiled. This creates the efficient structures used

by the underlying backend, in my case TensorFlow, in order to efficiently execute my model during training. I compile the model using the compile() function and it accepts three important attributes:

- Model optimizer: the search technique used to update weights in the model. For my model I used the RMSprop, an adaptive learning rate optimization method.
- Loss function: the evaluation of the model used by the optimizer to navigate the weight space. For my model I used the categorical crossentropy, used for multi-class logarithmic loss.
- Metrics: are evaluated by the model during training. For my model I used the categorical accuracy, used for multi-class logarithmic loss.

3.1 Relation

The word-vector representations that I used is the One-Hot encoding. In natural language processing, a one-hot vector is a 1 N matrix (vector) used to distinguish each word in a vocabulary from every other word in the vocabulary. The vector consists of 0s in all cells with the exception of a single 1 in a cell used uniquely to identify the word. To do this, firstly I create from the Knowledge-Base a list of all words contained in it and then I create a dictionary that associates an index with each word. In the same way I create an other dictionary that associates each relation to an index. After doing this, I create the features vector and the labels vector. For the features vector, as required by the Embedding layer, I created a two-dimensional vector and I linked the index of the corresponding word to each cell. It will then be the task of the Embedding layer to convert the indices into One-Hot vector. For the labels vector, instead, Ive created a two-dimensional vector that consists of 0s in all cells with the exception of a single 1 in a cell used uniquely to identify the relation associated to the word. Also, the management of unknown words during the test phase has been resolved by associating a specific index to these words.

3.2 Context

The word-vector representations that I used, as before, is the One-Hot encoding. In the same way I create the dictionary that associates an index with each word and another dictionary that associates each type (yes_or_not or classic) of answer to the corrispective index. After doing this, I create the features vector and the labels vector. For the features vector, as required by the Embedding layer, I created a two-dimensional vector and I linked the index of the corresponding word to each cell. It will then be the task of the Embedding layer to convert the indices into One-Hot vector. For the labels vector, instead, Ive created a two-dimensional vector that consists of 0s in all cells with the exception of a single 1 in a cell used uniquely to identify the type of answer associated to the word. Also, the management of unknown words during the test phase has been resolved as before.

4 Knlowledge-Base Server

As last task, I implemented the functions with the provided APIs of the Knlowledge-Base Server. In particular the functions are:

- *items_number_from(id = x)*: returns the number of items in the training set with id greater or equal to x.
- *items_from(id = x)*: returns the items with id greater or equal to x.
- *add_item*: take a new item from the parameters Q, A, D and R.
- *add_items*: take a json list of new items from post parameters. Those items will be added to the collect data.