Module 7 - Dialogue systems and question answering

Group: 12
Anton Claesson, 971104-3217, MPDSC, canton@student.chalmers.se, 18h
Lukas Martinsson : 980203-9678, MPDSC, malukas@student.chalmers.se 18h

We hereby declare that we have both actively participated in solving every exercise. All solutions are entirely our own work, without having taken part of other solutions

Reading and summary: Group

GUS (Genial Understander System) is a computer system that is intended to engage with a human in a dialog that is directed towards a specific goal within a restricted domain. GUS was restricted to the role of a travel agent in a dialogue with a client, who wants to make a return trip to a single city in California.
Restricting the domain of the conversation serves two main purposes; the system can achieve some measure of realism without any requirement to consider all possibilities of human knowledge or of the English language. The restricted system will be better able to guide the conversation within the defined boundary.

The GUS paper describes the frame-driven approach that is commonly used for dialogue systems. By using frames the prototype of one frame will have the previous frame as its instance. Through this GUS can use a recursive depth-first process, ensuring that all the information of a given part (e.g. the city) is given before it continues on and asks for the next part (e.g. the time).

Important points from the paper:

- GUS has partially realized the notion of more robust systems through procedural attachment and scheduling. Although GUS is able to respond appropriately when a client seizes initiative, given that the client is at least somewhat cooperative and informed it tries to retain control of the conversation if it's possible given the flow of the conversation. This is to easier acquire the information needed to help the client. If the client at an earlier point gives information concerning its demand GUS can utilize this information even if it did not ask for it, hence why the paper defines it as a mixed-initiative system. GUS was able to respond appropriately in many cases where the client seized the initiative, although the client needed to be somewhat cooperative and informed.
- GUS itself is not intelligent but demonstrates components of such a system.
- The main takeaways from the paper is that an intelligent dialogue system must have a parser, reasoning and a database of knowledge all of high quality. The knowledge must include language specific information,

knowledge in the conversational topic as well as a broad general knowledge of the world in order to make interpretations.

Implementation:

We choose to allow our system to handle the following dialogue topics:

- Providing a weather forecast.
- Finding a restaurant.
- Finding the next flight.

First off, we wanted to let the user select one of these topics.
The overall structure of the dialogue works in a turn based manner, where the system is the initiative taker. So, when starting the program the system gives information about the available topics, and prompts the user to select one.

The topic selection parses what the user types, and then uses keyword-based matching for each of the respective topics. If the user input contains a keyword related to a certain topic, this topic is selected. The following keywords are linked to each respective topic:

Topic	**Keywords**
Providing a weather forecast	'weather', 'forecast', 'temperature'
Finding a restaurant	'restaurant', 'food', 'dining','diner', 'eat'
Finding the next flight	'flight', 'travel', 'vacation', 'fly'

This keyword based topic matching is extremely simple and primitive, but is quite easily extensible the way it is coded using abstract classes for the rules. One might for example create a ML model based rule that classifies whether the input belongs to the corresponding topic or not if one wishes to improve the performance in more general cases.
Once the topic has been determined, the system’s goal is to fill all the empty slots of a frame corresponding to that topic. Each slot has a few corresponding questions that are chosen at random. This way we achieve some variation of system output.
The dialogue system is turn based where the system has the initiative, prompting the user with a question for the currently active unfilled slot in the frame.
After the user has provided an answer, the system attempts to fill the slot for which the question was asked.
One can summarize this process with the following pseudocode:

```
1 select a topic
2 while topic's frame not filled:
3    ask question for an unfilled slot
4    parse user answer
5    attempt to fill slot, otherwise make clarifying output
```

Unfortunately we did not have time to implement a system where multiple slots can be filled from a single user answer. However one way of doing it could be to attempt to match the user's answer to all slots instead of just the one corresponding to the asked question.

Nevertheless, given that the user has provided an answer for a given question, we want to parse the answer and see if it makes sense. For example, if the slot corresponds to a city type where the question is "what city do you live in?" the user's answer might be "I live in Gothenburg"- here the goal would be to extract the word "gothenburg" and fill the city slot with this value.

To do this there are a few "simpler" options. One such option is the keyword based matching that we performed in the topic selection; we could for example keep a long list of cities and see if any of those are mentioned in the user's answer.
In order to not have to list all possible cities we instead opted for a different approach.

We note that there are certain keywords after which a city name will follow, depending on how the question is asked. For example, we might assume that the word after the word "in" is the city name. We also keep a list of available cities and then match the extracted word with this list to make sure we don't get any nonsensical data. Of course this restricts the system domain quite a bit but it is also necessary.

We do the same thing for all the different slots in a frame.
So when defining a slot, one must provide a set of questions and keywords that proceed to the value we wish to find. Here is an example of a food preference type slot:

```
SlotQuestions(texts=[
            "What type of food do you prefer?",
            "What would you like to eat?",
            "Any food preferences?"],
                        patterns=[ "have", "eat"])
```

Here the assumption is that the user likely answers with a form similar to "I'd prefer to eat vegetarian" or something along those lines.

Note that if the user answers with a single word only we always use that word as the value for the slot, as long as it is a legal value.

This approach can definitely work quite well in restricted domains. However it is clear that one needs to carefully construct each question and the corresponding keyword patterns in order to prompt the user to be more likely to give an answer that can be parsed. Otherwise the information given by the user will oftentimes be missed by the system.

Now, let us talk about how we brought these things together.
Let us talk about the topic of finding a restaurant as an example.
We defined the following frame for this topic:

Slot	**Type**	**Questions**
LOCATION	City	"In which city?" "In which city would you like to go eat?", "Tell me your city."
TIME	Time	"When would you like to eat?" "At what time would you like to dine?"
FOOD PREFERENCE	Food	"What type of food do you prefer?" "What would you like to eat?", "Any food preferences?"

For each type there are some available options. For example, in this project we allow the City type to be any of [‘gothenburg', 'stockholm', 'london', 'new york'].
and the food preference to be any of [‘no’, ‘vegetarian’, ‘chicken’, ‘beef’, ‘vegan’, ‘fish’].
The Time type has the requirement of a valid time on the format ‘HH:MM’ to be filled.

As you can imagine it is easy to add new types and slots, and thus also easy to create frames for other topics apart from the ones we have created. The main problem might be asking the correct questions in order to prompt the specific user answer that we require with our parsing method.

Next, we show you an example dialogue when running the system with the intent of finding a restaurant. We note that we are eventually able to fill the required slots for this frame, but our parsing can’t handle the user input in a few of the cases.

```
What is your name?: Anton
System: Hi Anton.
I can help you with the following tasks:
* Providing a weather forecast.
* Finding a restaurant.
* Finding the next flight.
What would you like me to help you with?
Anton: I'd like to eat
System: I will now help you with finding a restaurant.
System: Tell me your city.
Anton: I'm in gothenburg
System: At what time would you like to dine?
Anton: does not matter
System: I didn't understand what you said.
System: When would you like to eat?
Anton: at 10
System: I don't understand the time you provided. When talking about time to me please use 'HH:MM' format.
System: At what time would you like to dine?
Anton: 10:30 please!
System: I didn't understand what you said.
System: At what time would you like to dine?
Anton: at 10:30
System: What type of food do you prefer?
Anton: I like apples
System: I didn't understand what you said.
System: Any food preferences?
Anton: yes!
System: I'm sorry, 'yes!' is not an available preference. Please choose from: ['no', 'vegetarian', 'chicken', 'beef', 'vegan',
'fish'].
System: What would you like to eat?
Anton: I'll have fish
System: Conversation over! Check that information is correct:
    Topic: 'Finding a restaurant.'
    LOCATION, [city: gothenburg]
    TIME, [time: {'hour': 10, 'minute': 30}]
    FOOD PREFERENCE, [food_preference: fish]
System: I must now query an API to fill the request for you but we leave that for another time :)
….
```

Suggested improvements

There are several areas that could be improved, in fact pretty much all areas. We previously mentioned that the keyword based topic selection could easily be improved and extended the way we implemented the code with abstract classes, such that the existing topic matching can easily be exchanged by creating new concrete subclasses of these abstract classes.

It is a bit harder to switch out the existing system for parsing and filling slots in a frame. First of all however, one could probably improve the system much by not relying on the assumption that a certain keyword we wish to find will always follow the provided pattern. Most likely, matching phrases instead would result in better performance, but of course it is not easy to find good phrases in that case. Filling the missing slots is obviously a hard problem to do in the general case, and it is obvious that our dialogue system is decent only in restricted domains with cooperative users.

To start off, we decided to base our system on a frame-filling approach using keyword based matching. The frames were necessary to store what information had been gathered already and what questions should be asked for unfilled information.

Keyword matching is not a trivial task, not only can the information given be formulated in a way which can make the model understand the information wrongly, it is also hard to find the proper keywords to match. If one were to create such a system with a well developed keyword matching function then the flow of the conversation would probably feel more natural. This is partly due to the fact that this enables the system to gather multiple information points from a single sentence so that the bot does not write questions that have already been answered. For example “I want to depart from Gothenburg and arrive in England'' has two key information points, Gothenburg and France. Ideally a system should then be able to understand both information points and also in this case which information point matches departure and arrival.

Individual Summary and Reflection: Anton

<u>Lecture summary (Dialogue systems and question answering)</u>
The lecture started with an overview of what properties a human conversation has, and what a dialogue is. This helped convey the scope of the difficulty in creating these types of systems. Some properties of a human conversation include constantives, directives, commissives and acknowledgements. Moreover, human dialogue has a certain structure including things like follow-up questions and sub-dialogues. One also has to consider who takes the initiative in the conversation or if the initiative is mixed. One of the most challenging aspects might be inference and implicature; oftentimes questions are not answered explicitly but information can still be inferred from the context.
There are two types of interactive dialogue systems; chatbots and task-based digital assistants. Some early examples of chatbots are ELIZA and PARRY which are keyword based and include a ranking of different keywords. A more modern chatbot is ALICE which is an improvement of ELIZA based on AI Markup Language (AIML) One can also build corpus/data-based chatbots that utilizes large datasets of real conversation instead of keywords. These include IR-based chatbots as well as chatbots based on neural models. Some risks with self learning chatbots are that they mimic data they are trained on, and any unwanted biases present in the data will be mirrored in the chatbot.
Next we covered task-based dialogue systems and digital assistants.
An overall structure was presented as to how these systems are designed. There are multiple important parts to such a system; natural language understanding, a dialogue manager, a task manager and natural language generation. The lecture

covered the tasks of these different modules as well as some reflections about where rule based or ML based approaches might be used.

Lecture summary (Game playing systems follow-up)

It was again explained that in game playing systems we can oftentimes not explore every state and or store the value of each state, and that we might instead use ML to go around these issues.
Next, the lecture covered more information regarding AlphaGo.
First of Go's game rules were presented and we could see that the number of states are unfeasibly large so ML is needed for this game playing system. In this case neural networks were used both to evaluate and to suggest moves to explore the MCTS tree. There were two policies in AlphaGo's MCTS; an exploration policy and a simulation policy that were both parametrized by CNNs. Initially the policy networks were "initialized" and trained on human games. In order to then improve upon human gameplay self-play and reinforcement learning was used.
Additionally, to avoid the expensive simulations during the rollout stage a different neural network was used to approximate the value of a state. It was trained to reduce the error between predicted reward and true outcomes.
Finally, it was briefly mentioned that the second iteration called AlphaGo zero was created without any reliance on human engineered features.

Next, OpenAI Five was presented. It is a system designed to play Dota. In this case, the state-action space is not just unfeasibly large but the time horizon of a game is very long and there was also the problem of partial observability, making this problem very hard. A difference for this system compared to AlphaGo was that it was trained from scratch learning only from self-play rather than relying on human games. There were many training runs which could persist over months, so the system had to be modular in order to allow the experience learned in one environment to carry over to another. Moreover, it was found that using just the terminal state as the reward signal was not fast enough for learning, so some handcrafted short-term rewards were crafted.
Another difference towards AlphaGo was that In this system the AI sees the state in terms of a large list rather than an image. Since Dota is also a 5v5 team game, each AI controlled player was controlled by a replica of the same network.

Reflection on the previous module (Game playing systems)

It is obvious that it is only in very specialized domains that implementing game playing systems is a relatively simple task, for example in tic-tac-toe where the state space is very small. In most applications and real-life settings however there are numerous difficulties that make it infeasible to use exact models, and in order to obtain more general algorithms and game playing systems more complex techniques are required. I think it is interesting that such "complex" systems that for example use state-value and action-value approximations are still quite understandable from the perspective that it is not too hard to wrap your head around how these systems work on a high level. I think that in some regard these black box approximations introduced with neural networks intuitively makes it easier to grasp what the systems are doing. I think that might be because it can be seen as more similar to how humans learn, in the sense that when we play a game we might not have a perfect understanding of the rules or the game state but we can learn to improve our play regardless. Finally, I think it is extremely interesting that reinforcement learning can be used to surpass human performance in such a varying amount of settings and tasks, and it truly shows the potential of AI and extends the possible domain for where it can be used.

Individual Summary and Reflection: Lukas

Lecture summary (Dialogue systems and question answering)

The lecture covered dialog bots and the difficulty in implementing them. Firstly the different types of communication types were discussed (constatives, directives, commissives and acknowledgments).

Following were the different types of initiatives a bot can establish with a human, for example system-initiative where the system has control of the narratives (this is easier since it enables the bot search for answers to its question instead of trying to understand the humans question) or user based, which is basically the reversed scenario (which also is much harder to code a bot for). Afterwards some issues regarding languages understanding that bots can not comprehend were discussed, like how implications in sentences which is something trivial for a human but hard or even impossible for bot to understand.

Then different types of chatbots were discussed, from the early bots ELIZA and PARRY to the later more advanced ALICE and how for example the ELIZA bot used a keyword matching system to converse. Then dialogue management with frame-based systems were introduced and their structure. Since we implemented a similar structure in our code this was really interesting. Lastly were some discussion between rule based vs ml-based dialogue systems.

Lecture summary (Game playing systems follow-up)

The lecture firstly gave a recap of tree search which was used in the assignment that week, and its limitations. The issue with tree search is that for larger games (like chess and go) every state can not feasibly be searched, hence some sort of ml would have to be used to address this. One way is through Monte-Carlo tree search. Another approach, which could be used in monte carlo is alpha beta pruning, whether leafs which are deemed worse can be chosen not to be explored, to can reduce to complexity

Then the lecture talked about AlphaGo, and how it was capable of beating the top players in the world 5-0. An in depth explanation of its policies and value network was then explained and how it can improve by playing against itself. However, in its preliminary versions it was trained on human games, which could decrease the time for it to reach a higher level or play before training against itself.

Afterwards the lecture covered OpenAI, the bot or bots capable of beating the top team in Dota 2. What makes this vastly different from Go is that it is not a perfect information game and there are 10 players playing instead of just 2. Also since the available options at each state are vast, from everything to choosing between the 100+ heroes to items purchased and more, a Dota 2 bot becomes much harder to train. Hence, they created several different modules that could be trained separately. However even with this system the creators still created hand crafted rewards to increase the system even further. And also they chose to firstly implement a version of their bot that could play mid 1v1, which is a gamemode that enables two players to compete with more limited rules, which made the initial bots easier to create. However, even then it had to have special rules in play (rules for example of what the opposite player was not allowed to do or what hero to use).

Reflection on the previous module (Game playing systems follow-up)
Game playing systems are a perfect way to advance machine learning and AI understanding. Games themself can vary vastly in complexity, from the simpler tic-tac-toe game with a MCTS which we implemented to games like Dota 2. It can help developers advance their knowledge of how certain algorithms work in an interactive way and can also be a test to see how good the machine learning models can be compared to humans. What is especially interesting, at least for a gamer such as myself, is how game playing systems nowadays can be used for humans to understand the game itself more in depth. This is due to the fact that the bots can "see" things assumed impossible for a human but when demonstrated can lead to advancement in the level of play for humans as well. However since most of these advanced models are created in a black-box fashion they still cannot explain their reasoning for doing their actions. If this would be possible, then "solving" more complex games would not be far away. Lastly, the domain of game playing is not ridden with the same ethical dilemmas that can be an issue in medicine for example. Hence, developers can focus on their main objective, whatever it is and not have to worry about its implication in the same way