DAT410 - Assignment 7

Eric Johansson & Max Sonnelid

March 8, 2021

Group number: 70	
Module number: 7	
Eric Johansson	**Max Sonnelid**
970429-2854	960528-5379
MPDSC	MPDSC
johaeric@student.chalmers.se	sonnelid@student.chalmers.se
25 hours spent	25 hours spent

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.

Introduction

The aim of this module is to create a program for a basic dialogue model. The system will not be terribly advanced, but it is rather a good exercise to understand the major challenges in this area. Moreover, since it is a hands on task, we will learn a lot by implementing a system by ourselves. Firstly, an article will be summarized and reflected around. Secondly, the implementation of the dialogue system will be described. Thirdly, summaries of lectures will be presented. Fourthly, reflections on the previous module will be done.

1 Summary of "GUS, A Frame-Driven Dialog System"

A *GUS*, which is an abbreviation for *Genial Understander System*, is a model that is supposed to replicate a sympathetic human being by answering questions asked by a human. By restricting the area in which it can be helpful, the model is able to generate useful answers. In the article, the model is restricted to only provide a *booking service for people who want to book flights to a specific city* (including a return ticket). The article gives an example of how such a conversation could look and then tries to examine the possible problems with handling a natural dialogue. For example, a person often refers to things that have been mentioned previously in the conversation as "it", which forces a smart model to remember important parts as the conversation continues. Another important issue is that of *mixed initiative.* For a computer program, it is rather easy to register an answer to a direct question, but a much harder task is to register information provided by the client that the program has not asked about, which needs to be handled by GUS.

The authors talk about the importance of firstly building small parts of the model and making sure that all these individual parts work, before they are eventually merged together. This leads to fewer bugs and facilitates the work of adding new features to the model. In order to let the model access all possible information concerning flights, while still minimizing the use of the working memory, the model could benefit from taking advantage of an *external data base.*

Furthermore, the concept of *frames* is introduced. Using frames is a way of defining structures for problems that are similar to each other. A frame consists mainly of slots, which have their own names, and a value (which could also be a new frame). In GUS, there also exists something called *procedural attachment* which is attached to the slots and decides how certain operations should be performed. These procedures are either classified as *demons* or *servants.* Demons are procedures that are executed when values are put into an instance, and servants are procedures that actively needs to be called at in order to be executed. Frames are used throughout the conversation in order to generate proper questions or answers. For example, one frame is used in the beginning of the conversation to distinguish what topic the client wants help with. In order to fill all the slots, the model generates questions and stores the valuable data. When all required slots are filled, the dialogue can be terminated.

Models like GUS can be very useful, but only as long as the person interacting with it are following a predetermined pattern. If the user abandon the predetermined behaviour, this model would probably yield poor answers. However, the authors state that their aim was not to create a very intelligent system, but rather illustrate the necessary components and related challenges of such an intelligent system.

2 Implementing a dialogue system

To implement a dialogue system that is easy to further extend when wanted, we tried to take inspiration from the article that is summarized above. An important feature that they mentioned was the usage of frames. This was something that we wanted to use. However, as mentioned several times before, we have learnt the hard way that building an advanced model out of nothing is very hard and often leads to many bugs. Thus, we started out small this time, by trying to make a system that only knows very little about one subject (similar to the what was described in the article) and then iteratively extend the complexity of the model.

The initial thing that we implemented was a method called `find_topic` which takes a string as input and searches for predetermined key-words. If the word contains *restaurant* or *eat*, then the client probably wants help to look for restaurants. The method uses a counter for each theme to determine which of the subjects key-words that the string contained most of. By doing this, it becomes very easy to add additional topics, since all that is needed is new key-words and an additional counter.

After that, we tried to come up with a general frame so that the model could behave in the same way no matter what task it is performing. Each frame would have some variables that needs to be filled in order to make a recommendation. For example, in order to recommend a restaurant, the model needs to know what and when you want to eat. The same thing would go for almost any other task. These variables were stored in a map structure, named `necessary_questions`. Moreover, in order to learn about these variables, all models needs to ask questions and thus, we made a model that handles that. Finally, we also need to know when the model is satisfied (i.e, all necessary questions are answered) and a method that makes the final recommendation to the user.

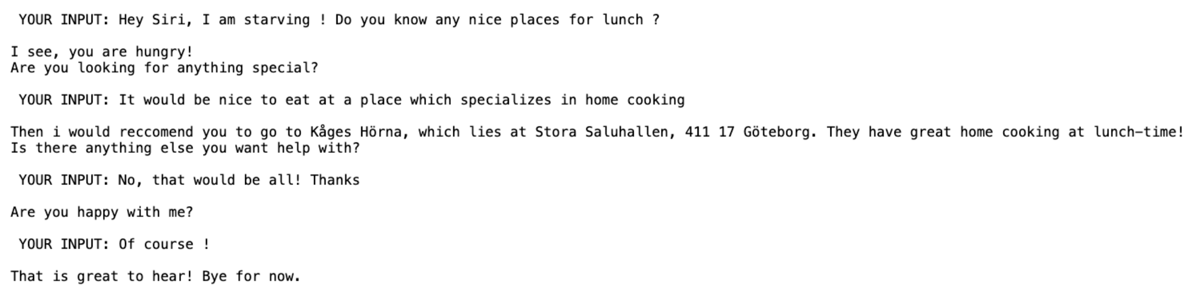
Another class called `AllData` was also created which acted as a data base, where frames later could collect information from. For each topic, the class contains a method that enables data collection regarding that specific subject.

After this, sub-classes of `General_frame` was created. One for each topic. By using the super-class as a base-model, we could later call for the same methods independent of which task that is to be solved. Below, we will talk more about what topics we included in our model and also how we implemented them. Finally, a class called `VirtualAssistant` was used to handle input from the user and create a matching frame to solve each problem.

2.1 Restaurant frame

Since the models are supposed to be easy, we started out with few criteria for when the model could make a recommendation. We decided that based on what meal (breakfast/lunch/dinner) and dish (pizza/sushi/home cooking etc..) the client was looking for, the model should make a proper recommendation of restaurants in Gothenburg. This could of course be extended in order to make better predictions, but we didn't think that the model would be any smarter by manually adding more dishes or additional features, such as price ranges.

Apart from the methods that were described earlier, the frame also have method called `updateVariables` which takes a string as input and looks for words using a regular expression function. In this way, we can detect wanted dishes of meals without asking for it if the user has mentioned it in previous text, as described in the article. If any `necessary_questions` is not yet answered, the class picks the first unknown variable from the map and asks a predetermined question to retrieve a proper answer. `satisfied` is used to know when all variables are known. By using these methods, the model becomes smart in a sense that you only have to specify your preferences once and the model will recognize that as a keyword and store it in the proper place. The same thing goes for all our frames.

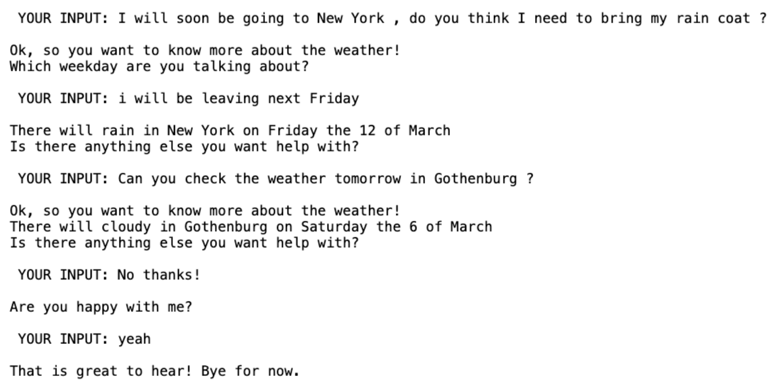


```
 YOUR INPUT: Hey Siri, I am starving ! Do you know any nice places for lunch ?

I see, you are hungry!
Are you looking for anything special?

 YOUR INPUT: It would be nice to eat at a place which specializes in home cooking

Then i would reccomend you to go to Kåges Hörna, which lies at Stora Saluhallen, 411 17 Göteborg. They have great home cooking at lunch-time!
Is there anything else you want help with?

 YOUR INPUT: No, that would be all! Thanks

Are you happy with me?

 YOUR INPUT: Of course !

That is great to hear! Bye for now.
```

Figure 1: An example of how our model helps with restaurant recommendation.

2.2 Weather frame

For this topic, we thought that location and time would be of interest in order to determine the weather conditions. Apart from the methods that were described in the previous frame, we added some help-method to handle date and time in a good way. The method `getDate` is used to retrieve current date and weekday, and by combining that and the input from the user, it determines for what day to check the weather conditions. For example, if the user asks for the weather next Thursday, the method checks the current weekday and date and computes what date next Thursday will occur. The method is relatively flexible and can calculate date based on multiple input, such as "tomorrow", "next Monday" or "now". In the same way as in the Resturant frame, we use regular expression to find and use these keywords in a correct way. The `getDate` takes help from two basic methods, `dayToInt` and `dayToString` which only transform a weekday-number to a string and vice-versa. If we were to develop this model further, we would also add methods that could input of regular dates, such as "fifth of March" or "Apr 21".

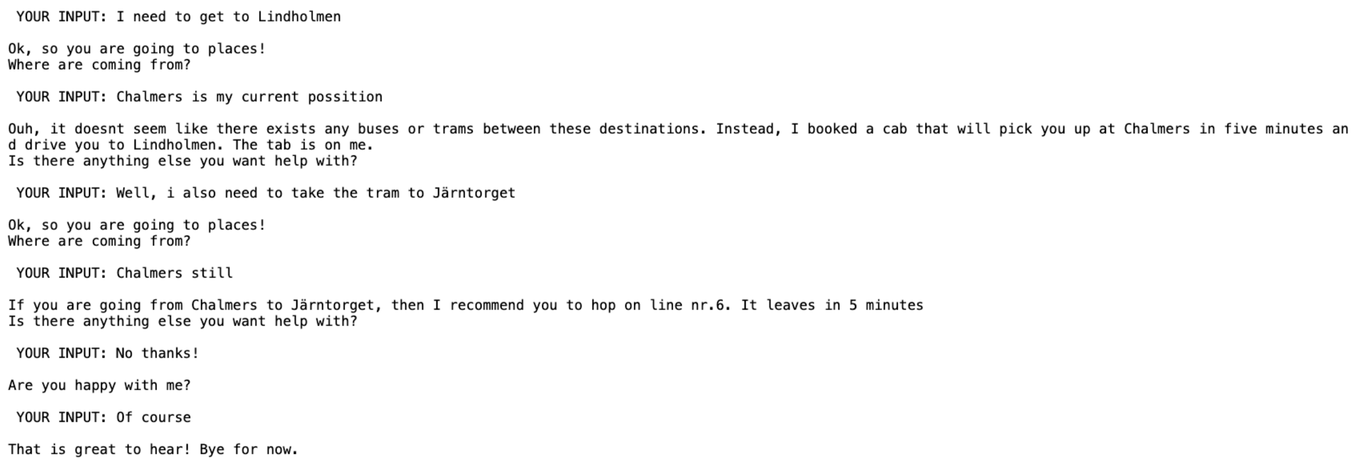


```
 YOUR INPUT: I will soon be going to New York , do you think I need to bring my rain coat ?

Ok, so you want to know more about the weather!
Which weekday are you talking about?

 YOUR INPUT: i will be leaving next Friday

There will rain in New York on Friday the 12 of March
Is there anything else you want help with?

 YOUR INPUT: Can you check the weather tomorrow in Gothenburg ?

Ok, so you want to know more about the weather!
There will cloudy in Gothenburg on Saturday the 6 of March
Is there anything else you want help with?

 YOUR INPUT: No thanks!

Are you happy with me?

 YOUR INPUT: yeah

That is great to hear! Bye for now.
```

Figure 2: An example of how our model helps with weather forecasting.

2.3 Transport frame

The purpose of the transport frame was to recommend a way to get from point A to B as soon as possible. In order to do this, only the start- and end destination needs to be known in order to find a way (if there exists any path between point A and B in our small data base). The problem of finding the next departure is handled by the data base, which will be further described later on. The rest of this frame was implemented in a similar fashion as previous ones.



```
 YOUR INPUT: I need to get to Lindholmen

Ok, so you are going to places!
Where are coming from?

 YOUR INPUT: Chalmers is my current possition

Ouh, it doesnt seem like there exists any buses or trams between these destinations. Instead, I booked a cab that will pick you up at Chalmers in five minutes an
d drive you to Lindholmen. The tab is on me.
Is there anything else you want help with?

 YOUR INPUT: Well, i also need to take the tram to Järntorget

Ok, so you are going to places!
Where are coming from?

 YOUR INPUT: Chalmers still

If you are going from Chalmers to Järntorget, then I recommend you to hop on line nr.6. It leaves in 5 minutes
Is there anything else you want help with?

 YOUR INPUT: No thanks!

Are you happy with me?

 YOUR INPUT: Of course

That is great to hear! Bye for now.
```

Figure 3: An example of how our model helps with transportation recommendation.

2.4 All Data

To further facilitate for an expansion of the model, we wanted to create a separate class that stores all information that is ultimately given to the user. This is done in a class called `AllData`. The class consists of one method for each frame. When a frame needs to gather information, it calls for its respective method and the proper data is then returned. We are well aware of that our data base consists of very little data, but as mentioned earlier, this is something that can easily be fixed without adding complexity of the remaining system and thus wouldn't make the model "better" in the scope of this assignment.

`weatherFacts` has the possibility to check the weather conditions in four cities. Depending on the real overall conditions in each city, we added corresponding probabilities of rain, clouds and sun. This is also a very basic way of determine the weather conditions. One way to make the predictions more accurate could have been to add time of the day and the season (spring, summer etc.) into account or even collect the real conditions by scraping a forecast site. The method `foodFacts` only takes meal and food-preference as input and based on a few if-statements returns a recommendation that fits the users current craving. Finally, `transportFacts` takes start- and end destinations as input and then checks if there exists any buses or trams in between these places. If so, it also checks the current time using the `datetime` package and then returns how long time it is until next departure of that specific line.

2.5 Virtual Assistant

Once all the frames were properly working on their own, we wanted to build a class that could handle the input from a user, and depending on the input, create a frame that fits accordingly. Moreover, we added two additional questions to make sure that the user don't want any more help and that the user is satisfied with the support. Once a virtual assistant is created the constructor prints a welcome text that tells the user what to do if help is needed (use the method `hey()`). The `hey()` method lets the user type what they want help with and then uses `find_topic` (which was described earlier) to create a proper frame. It then uses `updateVariables` to find potential keys in the initial input. The frame then runs its course until a final recommendation to the user can be made. The method then asks if it can do anything else and let the user type in further input. That input is then run through both some regular expression to see if the user say something like "yes" or "of course", and the `find_topic-` method to conclude if the user jumps straight to what he/she wants help with next. If any of these checks concludes that user want more help, the method calls for itself, and start over again with the latest string as a parameter. Otherwise, it asks a final question to determine if the user was happy with the support he/she got.

3 Making our dialogue model more advanced

As emphasized multiple times in this report, the model is clearly lacking some intelligence in order to make accurate predictions of weather, restaurants and transport. However, with 20 hours of available time, we think that it functions quite good. With that said, we also think that the model could be extended in order to make better predictions or to perform tasks in other areas. In the way the program is currently built, an implementation of a new task would be very easy.

To implement a new task, some steps needs to be carried out. Firstly, a new frame would have to be created, and it could inherit some functions from the super-class (general_frame). Secondly, new key words would be needed in `find_topic` to know when the user wants help with this new problem. Let's say that the new task would be to recommend which bonds to buy on the financial market. Then keywords such as "invest", "risk", "financial" and "bond" could be used to determine if a person wants financial advise. Moreover, new necessary questions will have to be defined as well as questions that nudges the user to tell their preferences. Parameters that could be useful could for example be risk-profile, bond-fees and if the user has any ethical issues that needs to be considered when investing. Finally, a new method in `AllData()` containing bonds and required data about them. By implementing these things, the digital assistant could then handle the rest.

If more time was given in the assignment, we think the next thing to implement to our model would be a better text handler. For example, by using stemming words with NLTK, the model could be better at determine the core meaning of each sentence. The model could also become non-case-sensitive for words such as weekdays or cities. By implementing these things, fewer keywords would be needed and the model would become more agile. Moreover, we would also have improved topic classifier. One easy change could be to weight each keyword in order to handle more keywords and to better understand the nuances of the user input. For example, lets say that we add a new task that can carry out flight bookings. Then, the word "travel" could be included in both the key words for flight booking as well as in our defined travel class, but maybe with different weight depending on when the word is most likely to occur. An even more advanced way could be to use a more advanced NLP model to further nuance the user input to make better predictions.

Regarding the digital assistant, it could be useful to store some more overall facts about the user in the virtual assistant object instead of deleting all information about a user when a new frame is created. For example, if a user wants help with multiple investments decisions, some parameters could be set as default based on previous input and then the model could ask if there is anything in the already inserted data the user want to change. For example, a user enters their criteria for an initial bond recommendation, and only wants to add a new ethical consideration, without changing the risk profile and the bond-fees. In that way the model "gets to know you" the more you use it.

Finally, we also talked about creating a more advanced data base. As mentioned earlier in the report, the current data base is relatively deficient. The most interesting approach that could be useful for all three task would be to collect real time data using web scraping. The weather could be retrieved using Googles API for weather forecasting, using the retrieved city name and date as input parameter. We could probably also find similar APIs from Google when looking for public transport in Gothenburg and restaurant recommendations. However, a lot of time and effort would need to be put into the implementation in order to make it work properly.

3.1 Challenges of digital assistants

By building this simple model, we have encountered some problems that were solved, but more importantly, we have realized the problems and challenges that exist when creating a dialogue system that are working on a larger scale. The first problem that we had was to generate proper questions in order to get the answer that we needed. In this case, we solved that by pre-defining certain questions that would nudge the user to type any of the words that we were looking for. However, we did not manage to make our model able to recognize parameters such as cities or dishes in general. That means that if a user wants to know the weather in Copenhagen, the model will continue to ask for what city the user wants to know the weather since the model only sees the words "Stockholm", "Gothenburg", "New York" and "San Francisco" as real cities. Another problem that would arise if we were to extend our model would be to make proper queries using non-standardized human input. Finally, even though adding one new feature would be easy, there would be different if we were to add 200 features. Then, our solution would probably become ad-hoc and inferior with each new added feature.

4 Summary of lectures

4.1 Eric Johansson

4.1.1 Lecture 7 - Dialogue systems

The difference between chat-bots and digital assistants is that the first mentioned is used for arbitrary conversations, whereas the latter is used to solve tasks. These systems creates or interacts as a memeber of a dialogue. A dialogue could be classified as a sequence of turns, where each turn consists of a sentence from a person (or bot). To further talk about the theory of dialogues, there exists four kinds of *speech acts*:

- Constatives
- Directives
- Commissives
- Acknowledgements

Chat-bots
An early example of a chat-bot was made in 1966, where a bot acted as a psychologist. This bot actually became quite popular. Over time, the bots has become more and more intelligent and these have been built using a variety of techniques. Today, neural networks is often used to analyze text.

Task-based dialogue systems
A task-based dialogue system consists of six parts: speech recognition, NL understanding, dialogue manager, task manager, natural language generation and text-to-speech synthesis. The natural language understanding part is used to extract the meaning of a sentence. This part has previously been rule-based, but is now also ML-based. The dialogue manage is ´, however, still mostly rule-based. Finally, the NL generation is template based, which means that a lot of pre-program output sentences is matched with empty slots that are filled in depending on the context. These templates can be created using ML. The purpose of a dialogue model is to fill in a form. One can see the dialogue management as a planning problem, where several questions needs to be answered in order to fill the form.

One way to train a model like this is to simulate the model by letting a human pretend to be a computer. This will generate data the can be used to gain insights ,and to develop accurate answers.

4.1.2 Follow-up lecture - Game Playing Systems

Most solutions on the previous module contained search trees. A common problem with game playing systems is that we cant use brute force in games that are more advanced than Tic-tac-toe, such as chess. Another issue is storing these potential values. To learn more about other limitations of MCTS, we will look at two examples.

Alpha Go
The model called *Alpha go* was created to win at the game of go. The model was initilized by defining policies that represents how good human players would play. Board positions and their respective moves are mapped. To create a model that outperforms humans, the model was then developed through self play. The model played against its previous version, known as reinforcement learning. Once the new model becomes better than its opponent, the opponent is updated to with the policies of the new model. This is repeated multiple times.

Since it takes a lot of time to always simulate to a terminal state in go, something called a value-network was added to the model to compliment the policy network. The value-network predicts outcomes out of a current state. These two networks are then combined. However, a new version called AlphaGo Zero is completely built on reinforcement learning and evaluation network. This new model eventually beat the previous one. The interesting thing about this is that the model no longer relies on human expert knowledge.

OpenAI Five
This model plays DoTA 2 which is very different from Go in the sense that at each moment, a player can do a wide variety of moves. Due to this added complexity, it is very hard to determine an outcome of the game based on one action. Moreover, the game has a long time-horizon. This leads to a complex model that needs a lot of training (over ten months of run time). In an simplified way,

the model is built in an hierchical state. Each player in a team was played by their own copy of the model. In order to train the model more efficiently the developer created a reward structure that gave the model some rewards based on different actions.

4.2 Max Sonnelid

4.2.1 Lecture 7 - Dialogue systems

- When talking about chatbots it is important to differ between *chatbots for arbitrary conversations* (general purpose) and *digital assistants for specific tasks* (specific purpose).
- An early example of a chatbot was *ELIZA* from 1966, which in a keyword-based manner was able to interact in therapeutic conversations and even remember things from earlier in the conversation and pick up on those topics later in the conversation.
- *Corpus-based chatbots* are based on very large datasets of real conversations and provide responses based on the last turn of the user.
- *IR-based chatbots* are either based on (1) returning the response most similar to last user turn or (2) returning the most similar response, where similarity is based on e.g. word vectors or word embeddings.
- A *task-based dialogue system* is built up of six main modules: speech recognition, Natural Language Processing, dialogue management, task management, natural language generation and text-to-speech synthesis.
- *Natural Language Processing* in a task-based dialogue system revolves around classifying domains, determining intent and extract relevant information.
- *Dialogue management* can be *finite-state based* (system-initiative and no flexibility) or *frame-based* (can be likened to a table with slots that need to be filled and all slots are associated with questions). An additional design is *frame-based with dialogue state*, where a rich information state beyond simple form-filling is created and that such a system can have multiples frames to fill.
- The most desirable *dialogue policy* is to determine the next action based on the entire previous dialogue, but a simple policy could be based on the current state of frame and last turns, including confirming as well as rejecting possible misunderstandings. The dialogue policy can be rule-based or learned.
- Speech recognition and Natural Language Processing are mostly ML-based, while dialogue management and Natural Language Generation are often rule-based.

4.2.2 Follow-up lecture - Game Playing Systems

- This week's assignment was based on a *Tree Search algorithm*, which has two big limitations: it is *not always possible to explore every possible state* and it is *not always possible to store the value of every possible state.* A solution for overcoming these problems is *Machine Learning*, which in this lecture will be exemplified by the Go-playing system *AlphaGo* and by the DoTA-playing system *OpenAI.*
- Because of the big board size and thus many possible states, Go has a rather high game complexity. However, the program AlphaGo was able to beat the top human Go player by combining *deep learning* and *Monte Carlo Tree Search*, where Machine Learning was used both for evaluating the board and suggesting moves to explore in the tree.
- The first version of AlphaGo learned the basics of Go by learning from *datasets of games played by human experts.* In order to improve on this human-based knowledge, *reinforcement learning* and self-play was used.
- An updated version, *AlphaGo Zero*, did not include the initial learning step of learning from human expert players. Instead, AlphaGo Zero *only learned from self-play* and only used a single network for both selection policy and evaluation, compared to the several networks used by the original AlphaGo. After only three days of training, Alpha Go Zero's performance surpassed the performance of the original Alpha Go, which showed that it is not always necessary to rely on initial human knowledge.

- *OpenAI Five* was created for playing the game DoTA 2, where the number of possible actions in each state is almost infinite. The main challenges of learning how to play DoTA are: *long time-horizons* (because of a large/infinite tree), *partial observability* (not possible to see the whole game map), *high-dimensional action space* and *high-dimensional state space.*
- OpenAI Five started learning DoTA 2 *only by self-play* and progressively more actions were learned. Training runs could persist in up to 10 months and so called surgery techniques were necessary to implement for continuing training during changing environments.
- Constructing a search tree for DoTA 2 is infeasible and instead a type of *reinforcement learning* was used, where rewards were given for short-term events. In order to further reduce the complexityof the game, only every fourth image frame was observed and then only observed as a large list of unit/item-specific features, compared to using more advanced visual processing.
- The decisive success factor for both OpenAI Five and AlphaGo was not the algorithmic advances. Instead the *access to more and better computer power* was the biggest contributing success factor as this let the reinforcement learning algorithm run a large number of iterations.

5 Reflection on the previous module

5.1 Eric Johansson

In the past assignment, we learned more about game-playing systems in order to generate our own game playing bot. More specifically, we built a model that were able to play Tic-tac-toe against a user or itself. The module was initiated with two articles that briefly discussed the AI-model that eventually beat Lee Sedol in Go, which is seen upon as the hardest game to master, and where Lee Sedol is the best player in the world. Even though the main article was relatively technical, it was very interesting to read about a top tier AI, and then try to implement bits and pieces from that solution into our own model. The main idea that we used from the article was Monte Carlo Tree Search, which has been used before the AlphaGo model, but is still highly relevant when building a game playing system. I think that we could have built a model that generated a descent Tic-tac-toe player without using Monte Carlo Tree Search. However, by using that approach, we managed to build a model that hopefully could be developed to play more advanced games, such as chess and perhaps even go. Due to limited time (and since it was outside the scope of the assignment), we didn't try to develop our model to enable additional games, but it would be interesting to see how it would perform in a game of chess.

This module was, compared to previous ones, very hard to initiate. It was not really clear how to start with the Monte Carlo Tree Search class and where to learn about besides some short Youtube-videos and some blog posts. The sources of information was a too high level for us to directly understand what to build. I guess that this was intentionally in order to make us formulate our own problem. i think we got some takeaways by formulating the problem, and I also think it is important to practice on this since we probably wont be faced with clearly detailed objects in real life. However, since a lot of time was put on understanding the problem, it was a bit stressful to solve the remaining parts. With that said, I still really enjoyed this module and once again this led to increased knowledge about AI in a new area and made us see a game playing as an advanced mathematical model rather than a black magic model.

5.2 Max Sonnelid

The objective with module 6 was to implement a Game Playing System for Tic-Tac-Toe, where the aim was that the system would be able to win playing the game. The first step in this implementation was to create a functioning representation of Tic-Tac-Toe, which a human would be able to play. As I had previous experience of implementing both a Tic-Tac-Toe game and a Snake game from previous object-oriented programming courses, this part was not rather challenging, but still a good refresher for my basic object-oriented programming skills.

As suggested by the PM, we chose to base the Game Playing System on Monte Carlo Tree Search as much material explaining this algorithm was provided and as we had been taught the mathematics behind this algorithm in the course Stochastic processes and Bayesian inference. As the implementation of the algorithm was rather tricky, we chose to use the implementation from the Int8 blog post as the starting point for our own implementation. It was rather tricky to integrate the Monte

Carlo Tree Search algorithm with the Tic-Tac-Toe representation, but with a step-by-step as well as an trial-and-error approach we eventually succeeded at creating a working system that was able to suggest a somewhat reasonable draw to make.

The exciting thing about this assignment was then to make an interactive implementation of this Game Playing System, where a human player was able to make its draw and then the computer would automatically make its draw based on Monte Carlo Tree Search. In such an implementation, it is very apparent that you have been able to create a system which for this specific task can match the human abilities as the algorithm is rather hidden for the user. Especially for a person without experience in ML or programming, it is easier to explain the abilities of the system if it is presented in an interactive way. This compared to the ML algorithms we have worked with so far, where there are many steps between the abstract input data and the actual prediction. An important way for enhancing the understanding of the system for non-programmers even more is to make the graphical interface look nicer and in a future assignment it would be interesting to integrate the AI system with a graphical interface library.

One thing that we could have improved in this assignment was the evaluation of the performance of the game playing system. This time the evaluation was done in a rather subjective way when we from our own experience of Tic-Tac-Toe decided whether a draw was good or not. However, the system would have greatly benefited from a more systematic and quantitative way of measuring the quality of the draws. If it is not possible to measure the quality of the draws systematically, it is of course not either possible to improve the quality of the draws systematically.