Human-level control through deep reinforcement learning

Thierry Deruyttere
r0660485

Armin Halilovic
r0679689

1. Introduction

For the course "Capita selecta: Artificial Intelligence" we received the subject of *deep reinforcement learning*. In this document we will explain everything we have done to get to know this subject. We will first start with our original plan we had for the demo, what we did with it and what eventually went wrong. We will also mention the things we learned from this experience. Then, we will explain to which other simulator we switched and how things went from there and which simulations we ran. We will finish this document with the things we have learned whilst researching this subject.

2. First simulator

For our first simulator we maybe saw things a bit too big. We wanted to showcase a really cool demo. That's why we decided to do deep reinforcement learning with self driving cars since they are all over the news. We quickly found some cool simulations online namely *DeepTraffic*[1] with a corresponding MIT course available on youtube[2] and a simulator written in Unity for a Udacity course[3] that we could use for our demo. We decided to go with the latter for our demo but since the former was a bit easier to start with, we started with DeepTraffic to get to know our subject.

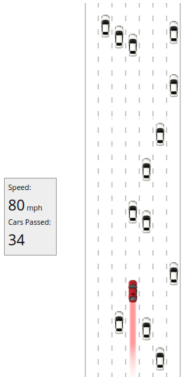

Figure 1. The DeepTraffic simulator[1]

2.1. DeepTraffic

The DeepTraffic simulator (which can be seen in figure 1) is a simulator written in javascript that works in a browser. The goal of the simulator is to make an agent that can swiftly maneuver between traffic without making collisions. The agent's reward is based on its speed. A high speed means a high reward and a low speed means a low reward. The nice thing about DeepTraffic is that it also has a MIT course[2] that explains a whole lot of things about self driving cars. To prepare ourselves for the task we wanted to accomplish, we watched the full play list (only five video's in total) of this course available on youtube.

Once we finished the playlist we started with an implementation for this simulator. We created two agents that could respectively go 67 mph and 69 mph. This was not the best speed of all time but for us it was a good way to get our feet wet with deep reinforcement learning. The lectures were

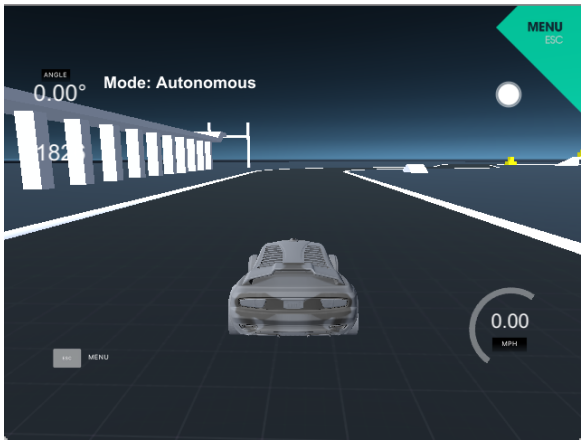

Figure 2. The Unity simulator for the udacity ourse[3]

also very informative about self driving cars. Once we finished this we decided to finally start with the demo we wanted to show.

2.2. Unity self driving car simulator

For our original demo we planned to use the self driving car simulator in Unity that was written for a Udacity course[3]. The goal of our demo would've been to create an agent that could ride between the lines on a track. To do this we had to do some changes to this simulator since it was originally written to do behavioral cloning, this is where a human first shows the A.I. what to do and the agent then has to copy this behavior. The changes were rather small, we added a extra text field on the UI and we added a way to know if the car was still on the track or not, we then had to make sure that the simulator could talk with our program. This wasn't that hard since the simulator already had a way to send data to an external script by using websockets and the course provided a start template script written in python. How the game world looked like can be seen on figure 2.

We used Pytorch[4] to create a implementation of the DQN. We spent quite some time on creating a good scoring function to describe the task we wanted to do. This step was necessary since the simulator didn't have a reward function. Which functions we tried and the ways our A.I. exploited these functions will be explained in the next section.

2.2.1. Scoring functions. An important part in reinforcement learning is the creating of a good scoring function. Every small unintended "bug" in a scoring function will be exploited by the agent so the creation of such a function should be done with care. We tried several scoring functions but we will describe here the two scoring functions in which the agent found actually a way to beat our intention!

The first scoring function that was exploited by our agent was a function based on lifetime having a higher weight than speed. Our idea was that if the agent would live as long as possible it would progress through the track.

$$touch\_track(speed+10*time\_alive)-10*(1-touch\_track)$$

Sadly enough this didn't really happen. The agent quickly found out that just by standing still it would get the highest score!

For our second function we learned that giving more weight to the time alive was not beneficial. That's why we created a new function where speed and time alive were as important.

$$touch\_track(speed*time\_alive)-10*(1-touch\_track)$$

With this function the agent found that it could actually drive in a circle between the two lines of the track whilst still moving at a high velocity.

2.2.2. Problems with the simulator. Sadly we had to stop our efforts of trying to use this simulator for our demo. After having spent too much time on scoring functions and getting the simulator up and running we came to the conclusion that the data from the simulator was not being sent fast enough. This was due to the websocket implementation they used. We saw that the data would be buffered in-game and data would still be sent long time after we disabled the learning in the simulator. We didn't immediately found a way to

stop the simulation when a frame would be send (we're no Unity pro's) so we thought that if we would change the websockets to tcp we would've gotten maybe a stable data stream. Since this was not assured we decided we've lost way too much time with this already and we decided to move on to our second simulator.

3. Second simulator

Obviously, for our second simulator we wanted to find something that would have a predefined scoring function such that we shouldn't spend too much time with that and also the game should wait for the input of the agent. We stumbled upon the gym environment from OpenAI[5]. This environment has a extensive library of games playable for an A.I. from text games to even 3D games. The best part is that it also has the Atari games mentioned in the Nature paper. This meant that we could implement the paper and try the same games.

3.1. Implementation

For the implementation we decided to use python and PyTorch[**?**] again. We started this implementation by closely following the paper. We sadly enough came across bugs that lasted for quite some hours to find. In the end we just started a new since we couldn't find these bugs and this time we got it right! Our agent was finally learning on Pong! We also implemented our agent in a notebook to make it more clear for people who would want to look at our implementation.

3.2. Atari games

The Atari games that we tried are: Pong, Freeway, **Meer games testen**.

We found that for these games our agent took the following time to have a winning strategy. **todo zet tabel in met alle waardes**.

3.3. Other games

The original paper was written by using Atari games, therefor we wanted to test it out for other games. We found a game that resembles what SpaceX is doing. The aim of the game is to land a SpaceX shuttle on a platform. We trained the agent for 24 hours but there was no improvement in it's score.

3.4. Reinforcement learning

4. Conclusion

The conclusion goes here.

References

[1] Deeptraffic.js. [Online]. Available: https://selfdrivingcars.mit.edu/deeptraffic/

[2] L. Fridman. Mit 6.s094: *Deep Learning*. [Online]. Available: https://selfdrivingcars.mit.edu/

[3] Self driving car engineer nanodegree. [Online]. Available: https://eu.udacity.com/course/self-driving-car-engineer-nanodegree–nd013

[4] Pytorch. [Online]. Available: http://pytorch.org/

[5] M. Plappert, M. Andrychowicz, A. Ray, B. McGrew, B. Baker, G. Powell, J. Schneider, J. Tobin, M. Chociej, P. Welinder, V. Kumar, and W. Zaremba, "Multi-goal reinforcement learning: Challenging robotics environments and request for research," 2018.