DRAFT
Human-level control through deep reinforcement learning

Thierry Deruyttere
r0660485

Armin Halilovic
r0679689

1. Introduction

For the course “Capita selecta: Artificial Intelligence”, we were assigned to do a presentation on *human-level control through deep reinforcement learning* .
In this document, we explain everything we have done in order to get to know this topic and to build a demo for the presentation.

2. Self driving cars

After reading the paper [1] we were given, we decided that the main goal of our presentation would be to create an agent that can learn to do a task by using deep reinforcement learning. We set out to find an interesting task that would allow us to show a cool demo.
It didn’t take long for us to think of an application where human-level control is important - self driving cars. This is a very popular application at the moment and it is one that is very interesting to both of us. Conveniently, we quickly stumbled upon a (free) relevant course by MIT, named ”Deep Learning for Self-Driving Cars” [2]. We watched the full playlist of this course on YouTube to get to know more about the topic and relevant methods.

2.1. DeepTraffic

At the end of the MIT course, *DeepTraffic* is introduced to the students. DeepTraffic is a deep reinforcement learning competition. It allows students to compete in creating the neural network that can swiftly manoeuvre a vehicle as fast as possible through traffic without causing collisions. The vehicle drives in a very simple 2D environment (written in JavaScript so that it can run in browsers). An example of this can be found in figure 1. The environment provides different features that can be used by neural networks, such as the distance to the car in front, or the amount of cars to the sides of the vehicle.

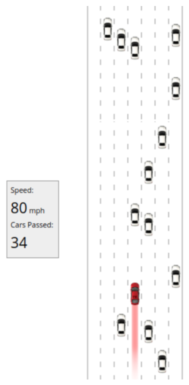

Figure 1. The DeepTraffic simulator[3]

We have experimented a bit with this application to get a first feel for creating neural networks. With a little tinkering, we achieved a top speed of 69mph (the maximum is 80).

2.2. Environment requirements

DeepTraffic was a nice way to get to know deep learning, but we decided it would be too simple to use it as our demo. We wanted to create something more akin to the given paper, where the agent learns from pixels, instead of handcrafted features.

We planned out what kind of network we wanted to use to train a self driving car: we would start with a couple of convolutional layers to process the input, followed by two or more fully connected layers to generate the output. The inputs would consist of an image of the view in front of the car, the velocity of the car, the acceleration of the car, and the angle of the steering wheel. The possible outputs would be accelerate, brake, move left, move right, brake, do nothing. We also came up with a few heuristics for rewards (reward higher velocity, punish moving off of the track, etc.).

This lead us to a set of requirements that a virtual environment would need to satisfy so that we could use our planned network. The virtual environment would need to support:

- A virtual car, along with methods to drive car.
- Tracks for the car to drive on.
- An extra camera in the virtual world (from the perspective of a driver) that allow for image streams to be used as input to the NN.
- A way to retrieve the following data of the car: velocity, acceleration, angle of steering wheel.
- A way to know if the car is still on the track.
- A way to link the car camera to the input of the NN and the output of the NN to the car controls.

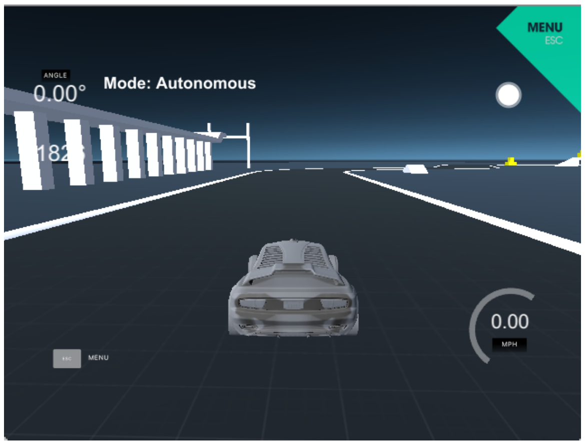

Figure 2. A view in Udacity's Self-Driving Car Simulator

2.3. Udacity's Self-Driving Car Simulator

The requirements above drove us toto a simulator written in Unity for a Udacity course [4]. This environment satisfied all of our requirements, so we chose to use for our demo. A view of this is visible in figure 2.

The goal of our demo at this point was to create an agent that could learn to drive in tracks created in this simulator, and finish the tracks as fast as possible. As the simulator was originally built to do behavioral cloning (where the AI learns from humans and tries to copy them), we had to do make changes to its code. The changes were rather small, we added a extra text field in 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 interact with our neural network.
That wasn't too hard, since the simulator already had a way to send data to an external script by using websockets, and the course provided a template script written in python.

We used Pytorch [5] to create a implementation of the DQN. We spent quite some time on creating a good reward function to describe the task we wanted to do. This step was necessary since the simulator didn't have a reward function.

The functions we tried and the ways our A.I. exploited these functions will be explained in the next section.

2.3.1. Reward functions. An important part in reinforcement learning is creating a good reward function. Every small, unintended "bug" in a reward function will be exploited by the agent, so the creation of such a function should be done with care. Ideally, the reward function should be set up such that the agent does exactly what you want it to do and nothing else. This means that the designer will have to think about how the reward function will react for every possible scenario in the environment, which is impossible for any realistic environment. See [6] for more info on this topic.

We tried several reward functions, but we will describe two reward functions for which the agent found an interesting way to beat our intention.

The first reward function that was exploited by our agent was a function based on lifetime having a higher weight than speed.

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

Our idea was that if the agent would live as long as possible it would progress through the track. However, we noticed that 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 that beneficial. That's why we created a new function where speed and time alive were as important.

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

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.

2.3.2. 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 reward 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 used in the simulator. 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 find a way to stop the simulation when a frame would be sent, so we thought that if we would change the websockets to tcp we might have gotten a stable data stream. Since this was not assured, we decided we've lost too much time with this already and decided to move on to a different simulator.

3. Playing games

We stepped down from our idea to build a self driving car agent. Our main goal now was to find an environment in which could integrate our neural network in an easy way. Also, we wanted the environment to be able to provide rewards for actions, so that we wouldn't have to create any reward functions.

We stumbled upon the gym environment from OpenAI's [7] Gym environment. Gym provides an extensive library of games playable for A.I. agents, from text games to even 3D games. This is when the plan for out demo changed from creating an agent that can learn to drive a car, to one that can learn to play a variety of games.

3.1. Implementation

For the implementation we decided to use python and PyTorch [5] again. We started this implementation by closely following Deepmind's DQN [1]. However, we came across bugs that took quite some hours to fix. In the end, we just started again from an empty file, 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

In the paper we were given (Human-level control through deep reinforcement learning) [1], Atari games were used to show off the capabilities and performance of the proposed DQN algorithm. Since Gym provides functionality to easily simulate Atari games, we decided to test our implementation on those games. The Atari games that we tried are Pong, Freeway, and Space Invaders.

3.3. Other games

We found a game called "Rocket Lander" [8], in which the goal is to carefully guide a rocket to landing pads. We found that the network never performed well in this game. We suspect this is due to it's inherent complexity being a level higher than the Atari games, which can mostly be played in a reactionary way. Rocket Lander requires more strategy in landing the rocket correctly.

4. Other deep reinforcement learning methods

After we finished the work for the demo, we investigated some other deep reinforcement learning methods. The paper "A Brief Survey of Deep Reinforcement Learning" [9] describes many different methods that were introduced up to 2017. A lot of these were tweaks to the DQN algorithmm. The survey paper also introduced different classes of algorithms: model-free vs. model-based, and gradient-free vs. gradient-based. Some examples:

- Model-free: DQN, DQRN, Dueling Network
- Model-based: AlphaGo, Guided Policy Search, Continuous Deep Q-Learning with Model-based Acceleration
- Gradient-free: Neural evolution strategies (NES), Covariance Matrix Adaptation ES (CMA-ES)
- Gradient-based: AlphaGo, DQN, DQRN, Dueling Network

Model-free methods learn a policy or value function, while model-based methods learn a model of some kind (e.g. a dynamics model of walking around). Model-free methods suffer from large sample complexity, which is what model-based methods avoids. However, model-based methods fail to generalize well to a variety of complex tasks.

We explored two other methods in more depth and explained their main approach in the presentation. The methods were "Dueling Network Architectures for Deep Reinforcement Learning" [10], and "Deep Recurrent Q-Learning for Partially Observable MDPs" [11].

5. Curiosity learning

We stumbled upon curiosity learning when looking for papers for the project. A major drawback of deep reinforcement learning is that you need to define a specific reward function such that the agent can learn. For some goals, defining such a function is not easy [12]. Sometimes, there is no reward for a long period of time until a certain series of actions is performed. This makes it quite hard for an agent to learn an optimal strategy. Even after millions of frames, the agent doesn't learn anything good. We believe that this is why our implementation didn't perform well on Rocket Lander.

A way to solve this is by making the agent curious. Instead of us defining a reward function, the agent will create one by itself. The way this is done is by creating a neural network that predicts how the agent should go from state ϕ_i to state ϕ_{i+1}. The distance between the predicted action and the expected action is defined as the new reward function. Since we were out of time we couldn't try to implement this ourselves.

6. Conclusion

During our research to find what gives an agent human-level control through deep reinforcement learning, we came across a lot of different methods. We have tried multiple approaches that didn't end up working out, but in the end we we were able to create such an agent successfully! We learned a great deal about this subject and also a bit about what might come in the near future. The only thing we would change if we had to redo this project, is spend way less time on environments that are too complex.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015. [Online]. Available: http://dx.doi.org/10.1038/nature14236

[2] L. Fridman. Mit 6.s094: Deep learning for self-driving cars. [Online]. Available: https://selfdrivingcars.mit.edu/

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

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

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

[6] Deep reinforcement learning doesn't work yet. [Online]. Available: https://eu.udacity.com/course/self-driving-car-engineer-nanodegree–nd013

[7] Openai gym. [Online]. Available: https://github.com/openai/gym

[8] Rocketlander. [Online]. Available: https://github.com/EmbersArc/gym

[9] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "A brief survey of deep reinforcement learning," *arXiv preprint arXiv:1708.05866*, 2017.

[10] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Freitas, "Dueling network architectures for deep reinforcement learning," *arXiv preprint arXiv:1511.06581*, 2015.

[11] M. J. H. P. Stone, "Deep recurrent q-learning for partially observable mdps," *CoRR*, vol. abs/1507.06527, 2015.

[12] OpenFaceChinesePoker. Why montezuma revenge doesnt work in deepmind arcade learning environment. [Online]. Available: https://www.youtube.com/watch?v=1rwPI3RG-lU