Deep Learning in Python Semester Project: Training an Autonomous Vehicle to Lane Following in the Duckietown Environment with Deep Reinforcement Learning

Team "LeepDearning":
Szilárd H. Tóth*. Bence T. Pálvölgyi, Jr.**

**e-mail: szilardhunor.toth@edu.bme.hu*
***e-mail: p9bence@gmail.com*

Abstract: In this project, experiments were made to train a deep reinforcement learning (DRL) based agent for lane following in the Duckietown environment on a relatively low-end PC. For training, the Stable-Baselines 3 Python library was used with Tensorboard for logging. The results are not significant, although we wish to improve them during the exam period.

Keywords: autonomous driving, lane following, reinforcement learning, Duckietown, simulation in Python

1. THE DUCKIETOWN ENVIRONMENT

Duckietown [1] is a self-driving, remotely controlled (RC) car environment nurtured and developed by the Duckietown Project. The framework is basically a physical representation of a suburban environment based on real-life traffic scenarios, which was mainly engineered for educational and research purposes. These physical environments are called Duckietown platforms and are highly customizable.

Gym-Duckietown is an open-source simulation environment for the Duckietown Universe, written in pure Python/OpenGL (Pyglet) [2]. It places an agent, a Duckiebot, inside of an instance of a Duckietown: a loop of roads with turns, intersections, obstacles, Duckie pedestrians, and other Duckiebots. The simulator offers a wide range of tasks, from simple lane-following to full city navigation with dynamic obstacles, and also ships with features, wrappers, and tools that can help to bring algorithms to the real robot, including domain-randomization, accurate camera distortion, and differential-drive physics.

So far in this project, this environment was used for training agent to perform lane following tasks on simple maps, like small loops and slaloms, then validated on a self-made test track, which incorporated both elements.

2. LITERATURE REVIEW

2.1 Reinforcement Learning

Reinforcement learning (RL) is a type of machine learning paradigm where the training is done via interacting the agent with an environment to solve a problem, instead of preparing a set of data to fit a model to (like in supervised learning) [3]. The agent's goal is to maximize a cumulative reward signal, which is defined by the environment. In every operation step, the agents observe the current state of the environment and decides on an action based on it.

Besides the classic RL algorithms, the nowadays popular deep reinforcement learning (DRL) provides methods, which incorporate usage of neural networks as approximators to train on very big and continuous environments and/or use continuous action spaces effectively. Some of these algorithms will be introduced in more detail in Section 3.4.

2.2 Previous Works and Results in Duckietown

Many people have created project for themselves or for other specific causes, for example school, but the ultimate comparing is done in the "AI Driving Olympics (AI-DO)" which are a set of competitions with the objective of evaluating the state of the art for ML/AI for embodied intelligence. AI-DO has been hosted by Duckietown with competition finals twice per year, at ICRA (International Conference on Robotics and Automation). The challenges range from single robot tasks such as lane following (LF) on road-loop map to complex multi-robot behaviours such as lane following with intersection and other vehicles in the presence of pedestrians (LFIVP). We chose to start with lane following on a looped road map. Also, there are several submitted solutions which were designed under this very course here at BUTE [4, 5, 6]

A solution consists of two main tasks. Input pre-processing for the agent to handle, and the agent itself, to give the desired output. Submitted algorithms most often apply pre-processing before the agent gets the input. Some of them to shorten computation time, some to increase learnability. Resizing, cropping and normalization are set based on hardware strength; colour segmentations strongly influence the result, and image sequencing is adjusted to satisfy both reasons. Some segmentations divide the closing and separating lines, others don't.

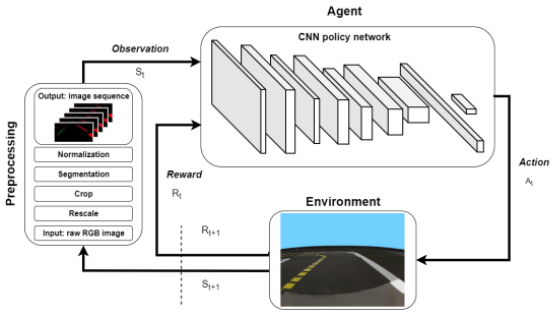

Fig. 1. A basic reinforcement learning framework for training a Duckietown agent. [4]

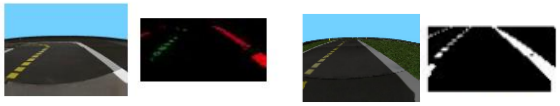

Fig. 2. Different types of pre-processing ideas. [5]

Results are evaluated based on different aspects in simulation and real life as well; for example, survival time, travelled distance, lateral deviation. State-of-the-art implementations manage to drive nearly flawlessly (98% with Duckietown scoring) so the competition is really tight.

3. PREPARATIONS

3.1 Installing the Gym-Duckietown Environment

The first phase of the project was to prepare and install the Gym-Duckietown simulator on a computer with a Linux operation system. At first, we tried the free cloud computing services of AWS, but didn't manage to work with them, so we switched to use a physical PC with an Ubuntu 22.04 system installed. Sadly, we only managed to attain a low-end PC for this project with no GPU acceleration and only an Intel Pentium quad core CPU with 8 GB RAM. These resulted in days of simulation times, which greatly affected the amount of meaningful work we could put in into the project, thus so far, we tried only our most basic ideas. Also, to improve the training performance, virtual displays were used for training the agents with the xvfb module, so we could work perform parallel training much easier.

3.2 Test Runs and Map Creation

After following the instructions of the Github repository to install the Gym-Ducketown environment, we experimented with the manual control and the baseline agent scripts on the built-in Duckietown maps. There are multiple registered gym environments, so far we only trained on the small loop maps. Also, for future validation, a custom map was created based on observing the example .yaml files, which contains a loop with a couple of slaloms, so the agent can be validated on navigating through both types of corners. We are also planning to add intersections in the future, if the agent performs well on the loop map.

Fig. 3. The operating baseline agent on our custom map.

3.3 Formulation of the RL Problem

The observations are single camera images, as numpy arrays of size (480, 640, 3) (so the original image resolution is 640x480), containing unsigned 8-bit integer values in the [0, 255] range. This image size was scaled down to 80x60 with a wrapper to improve training performance.

The action space consist of the forward torques applied to the left and right wheels and are scaled between -1 and 1. These values are continuous by default, but a discrete wrapper can be used to make finite representations for algorithms using finite action spaces. In Section 4, a few of these (which we experimented with) are described in more detail.

The reward function is defined as the following:

$$\alpha_p \cdot v \cdot \cos\varphi + \alpha_d \cdot d + \alpha_p \cdot p_c \qquad (1)$$

where v is the speed of the vehicle in the simulator, φ is the angle between the heading of the vehicle and the tangent of the optimal curve, d is the distance from the middle of the right lane, and p_c is a penalty for collisions. The coefficients α_v, α_d, and α_p are used to scale the different factors of the reward. The factor $\alpha_p \cdot v \cdot \cos\varphi$ encourages the agent to drive faster and follow the optimal curve. The factor $\alpha_d \cdot d$ with a negative α_d, encourages driving close to the middle of the right lane, and $\alpha_p \cdot p_c$ can be used to avoid collisions or driving too close to other objects. The last factor is currently ignored because there are no obstacles on the track so far. When the vehicle is in an invalid position (e.g. it leaves the track), it gets a penalty of $-p_x$, and the simulated episode ends.

3.4 Selecting the RL Algorithms

We decided on trying out two different DRL algorithms. The first one is the Deep Q-learning (DQL) [7] algorithm, which is an off-policy method based on the classic tabular Q-learning. The agent uses a neural network as an approximator of the state value function to select an action from the finite action space. There is no embedded exploration feature, so one must be

applied. Here, we used the simple ε-greedy method with a decaying exploration rate, so there is a $0 < \varepsilon < 1$ probability in every step to choose an action at random.

The second algorithm is an on-policy method called Proximity Policy Optimization (PPO) [8], which operates with a continuous action space. The main idea is that after an update, the new policy should be not too far from the old policy. For that, PPO uses clipping to avoid too large update.

We decided on these two algorithms, because they're fundamentally different, so we can cover more ground regarding the current state of the art. So far, most of the features of these methods are untested for hyperparameter optimalization, so there is still a lot of room for improvement.

4. EXPERIMENTS & RESULTS

For training, we used the implementation available in the Stable Baselines collection [9] and logged the result with Tensorboard. Only the default neural network architectures were tested, and we mostly focused on the action space representation and the parameters of the reward function.

4.1 DQN

Until now, we considered three different approaches for the action space representation. In each case, only positive wheel velocities were allowed because only these are required to move forward, and 7 different actions were defined: going straight, slight steering, sharp steering and rotating in each direction (left/right). In Case (1), the predicted values were interpreted as the amount of braking from the maximum speed, which means the going straight action is [1, 1], and the appropriate side is changed for turning motions (for ex., slight turn left is [0.75, 1], sharp turn right is [1, 0.5]). In Case (2), the going straight action is [0.5, 0.5], and the steering is done by changing both values, but in the opposite directions (for ex., slight turn left is [0.4, 0.6], sharp turn right is [0.75, 0.25]) In the last case, we referred to a Scientific Students' Association Report [10], because we wished to recreate the training results of that work (for the exact values, please see the referenced paper).

For the reward function parameters, first we followed the referenced report, so $\alpha_v = 10$, $\alpha_d = -100$, and $\alpha_p = 400$ were selected, with $p_x = 40$. After observing the result, we also tried $\alpha_d = -10$ and $p_x = 1500$ for reference. Every other hyperparameter like learning rate, discount factor, exploration decay rate and mini batch size, we followed the referenced report, except in the experience buffer size, which was only 20.000 against the referenced 150.000, because the 8GB RAM was only enough for this value, higher numbers resulted in memory overflow. In every case, the training lasted for 1.000.000 steps.

On Fig. 2, the comparison of a few training sessions is shown: yellow is the recreation attempt of the report's results, dark blue is a Case (1) action representation setting with the report's reward function, light blue is a Case (1) training but with the modified reward parameters, and pink is a Case (2) attempt

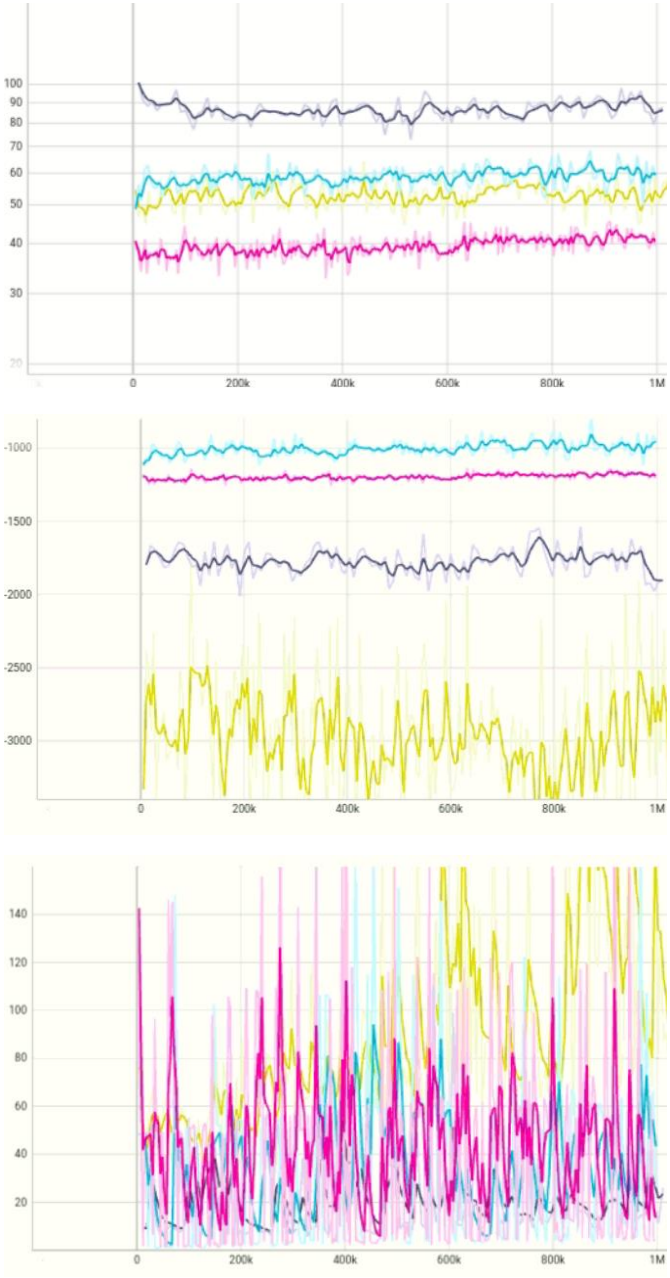

Fig. 4. The average episode length, average episode reward and train loss values for 4 different cases of DQN training (see the meaning of the colours in the text above).

also with the modified reward. Observing the average episode lengths, the yellow and light blue lines are similar, and can be considered as such, because the observed rewards are different only because the scaling is changed due to the difference in the parameters. The only thing that is interesting is the dominance of the Case (1) action representation against the other alternatives, although we are still unsure to say this, because convergence is still almost non-existent in every case. Also, it seems that the referenced reward parameters are better than the modified ones.

4.2 PPO

So far, only one experiment was made with the PPO, although it was observed that it behaves very differently from the DQN. Also, it's around 1.5 times slower, which is the main reason we only performed one training session so far with this algorithm.

The hyperparameters common with the DQN have the same settings here as in the previous case, and the PPO specific arguments (like clipping) were left to their default values. The reward parameters are the referenced ones, and the used action representation is the same as it was defined under Section 3.3. The interesting thing is that the mean values converge to nowhere, while the training loss decreases over time, and observing the trend of the last few thousand steps, it can be suspected that the values have still not reached the plateau, so further training might yield interesting results.

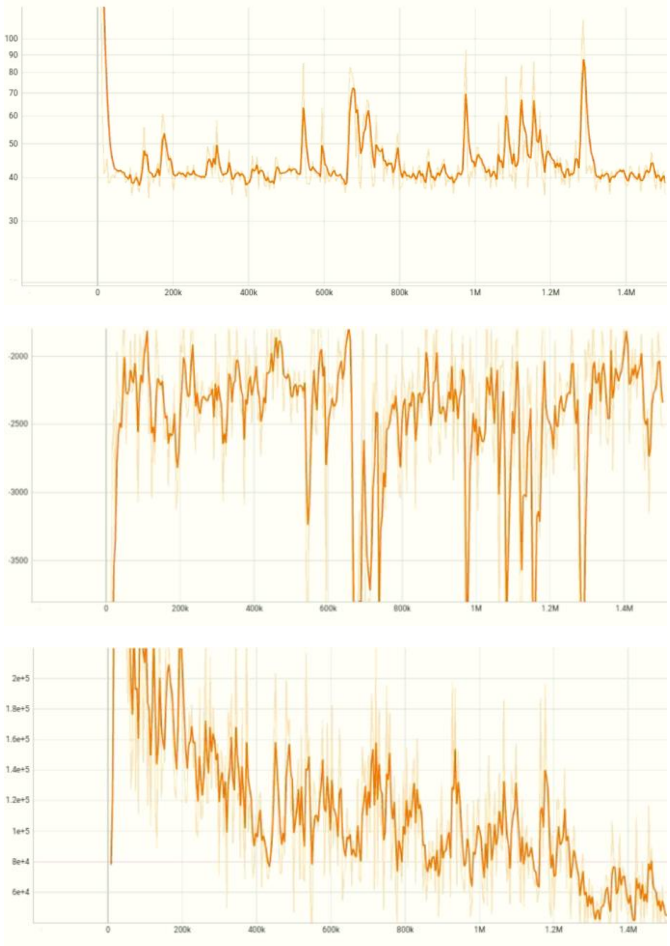

Fig. 5. The average episode length, average episode reward and train loss values in the case of the PPO training.

5. FURTHER IDEAS FOR IMPROVEMENT

Because of the limited computational resources and the time spent on preparing the training environment correctly, we didn't have enough time to train the agents thoroughly, and we still have a lot of ideas to improve the agents to achieve a successful training, on at least the simple loop maps. These include experimenting with different neural network architectures, pre-processing methods, hyperparameter values, and inspecting the reward function further via writing a logging script for the different reward factors to see how they affect the cumulative reward values. Thus, we happily accept the conditions of writing a blogpost, so we can experiment until the exam date.

REFERENCES

[1] The AI Driving Olimpics. Documentation (online); url: https://docs.duckietown.org/daffy/AIDO/out/index.html (last accessed: 2022.12.11).

[2] Gym-Duckietown Github repository (online); url: https://github.com/duckietown/gym-duckietown (last accessed: 2022.12.11)

[3] Sutton, R. S., & Barto, A. G. (2018). Reinforcement learning: An introduction. MIT press.

[4] Almási, P., Moni, R., & Gyires-Tóth, B. (2020, July). Robust reinforcement learning-based autonomous driving agent for simulation and real world. In 2020 International Joint Conference on Neural Networks (IJCNN) (pp. 1-8). IEEE.

[5] Lőrincz Z. (2020). Imitation Learning In The Duckietown Environment (MSc Thesis).

[6] Kalapos, A., Gór, C., Moni, R., & Harmati, I. (2021). Vision-based reinforcement learning for lane-tracking control. Acta IMEKO, 10(3), 7-14.

[7] Fan, J., Wang, Z., Xie, Y., & Yang, Z. (2020, July). A theoretical analysis of deep Q-learning. In Learning for Dynamics and Control (pp. 486-489). PMLR.

[8] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347.

[9] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable baselines (online); url: https://github.com/hill-a/stable-baselines (last accessed: 2022.12.11).

[10] Almási, P. (2020). Real-world autonomous driving using deep reinforcement learning and domain randomization. Scientific Students' Association Report, BUTE.