Self-driving in the Duckietown environment/Önvezető autózás a Duckietown környezetben

Matyas Polya
Department of Measurement and Information Systems
Budapest University of Technology and Economics
Budapest, Hungary
mpolya@edu.bme.hu

Axel Katona
Department of Physics
Budapest University of Technology and Economics
Budapest, Hungary
katonaa@edu.bme.hu

Absztrakt

Az önvezető autózás a gépi tanulási módszerek egyik legfontosabb megoldatlan területe. Az autókat különböző szenzorokkal látjuk el, melyek segítségével az érzékeli a környezetét, és a belső algoritmusai alapján kiszámolja, majd végrehajtja lehető legoptimálisabb cselekedetet. Az MIT egyetem 2016-ban létrehozta a Duckietown-t, mely egy leegyszerűsített megvalósítása az önvezető autózás problémakörének. A Duckietown-ban fellelhető ágensek csak 1 szenzorral rendelkeznek, mely egy kamera. A környezet segítségével az emberek betekintést nyerhetnek a témába, mely megoldásával közelebb kerülhetnek az abban rejlő nehézségek megértéséhez. A Duckietown környezet megvalósítható a valós életben, erre a célra létrehozott kellékek és elektronikus eszközök segítségével, illetve szimulációban, ami az implementálási költségeket, illetve nehézséget nagy mértékben csökkenti. Az önvezető autózás kivitelezésére az imitációs és megmerősítéses tanulás módszereit alkalmaztuk a mély neutrális háló tanítására.

Abstract

Self-driving is one of the most important unsolved areas of machine learning. Cars are equipped with different sensors that allow them to sense their environment, calculate the most optimal action based on their internal algorithms and then execute it. In 2016, MIT University created Duckietown, a simplified implementation of the self-driving car problem. The agents found in Duckietown have only 1 sensor, which is a camera. The environment gives people an insight into the subject, which, when solved, will bring them closer to understanding the difficulties involved in it. The Duckietown environment can be implemented in real life, using dedicated props and electronic devices, or in simulation, which greatly reduces the cost and difficulty of implementation. For the challenge of self-driving, we applied the imitation and reinforcement learning techniques to train the deep neural network.

1 Introduction

The *Self-driving in the Duckietown environment* [1] topic was chosen as our homework for the Deep Learning in Practice with Python and LUA subject. The Duckietown environment provides real and

simulated interfaces with self-driving educational purposes. 3 main challenge types are available: lane following (LF), lane following with vehicles (LFV), lane following with intersections (LFVI). In our homework, we aimed to solve the problem of lane following (LF). The Duckietown-gym implements the gym environment created by OpenAI, which makes the implementation of reinforcement learning algorithms for the Duckietown environment simple. In our homework, imitation [7] and reinforcement learning [8] methods are examined and compared.

2 Background, previous solutions

As already mentioned, the problem is solvable both with imitation and reinforcement learning.

With imitation learning, a deep convolutional neural network is used as the model. The input of the model is an image perceived by the agent (Duckiebot) in the Duckietown environment, and the output is the expected action to be performed by the agent. In our case, the dataset used is generated manually, by trying to drive as accurately as possible in the Duckietown simulation environment. The problem with this approach is that the resulting model can only be as accurate as the training dataset used. The official website provides useful resources for the implementation of imitation learning.

Reinforcement learning is much more suitable for the training of the model. The Duckietown environment provides the Duckietown-gym interface, which implements the gym toolkit [2] used by popular reinforcement learning libraries. Through the interface, we can observe the environment, and interact with it. The default observation is a 480x640 sized RGB image. The interaction happens by controlling the 2 wheels attached to the duckiebot, by passing in an input consisting of $2 \times [-1, 1]$ numbers.

3 Achitecture

3.1 Imitiation learning

The Duckiebot, and also the simulated agent uses the velocity-difference of its motors to turn. In the script, however, the user does not control the separate engines, rather the *actions* (the speed, and the turning angle), so our model inherits this kind of control method. Similarly to the behaviour cloning model [11]. the velocity and the angle of turning is predicted by separate networks. Since the angle also influences the speed (if we are in a road turn, the both needs to slow down to track the lane), and vice-versa it was expected that these neural networks can work in this separated way in harmony. We used here a CNN [5] network architecture for both the angle prediction, and also for the velocity. The convolution neural networks are very effective tools for image classification problems. We used leaky relu as activation function. The 4 layers of CNN network is terminated by two fully connected layers, and outputs a $(4, 1)$ object for the velocity, and a $(3, 1)$ vector for the angle. The larges value's index will be considered as the prediction. The model is found in the Train folder of the project in the *model.py* script. Cross entropy loss and SGD optimizer were used. The IL was coded using Pytorch.

3.2 Reinforcement learning

For the implementation of the reinforcement learning based method, the Stable Baselines3 [6] library was used, which includes set of reliable implementations of the algorithms in PyTorch. The 2 examined algorithms were the Deep Q-Learning (DQN) [3] and the Proximal Policy Optimization (PPO) [4]. A CNN with 3 convolutional layers, ReLU acivations and no pooling layers was used as the feature extractor network.

The duckietown-gym environment – as mentioned previously – implements the gym interface. This includes the following functions:

- step(action): Signals the environment to perform an action. Returns the observation, reward, whether the episode ended, and extra info.
- reset(): Resets the environment.
- render(): Renders the environment, typically for human viewing .
- close(): Closes the environment.
- seed(): Gives a seed for the environment for its random number generator.

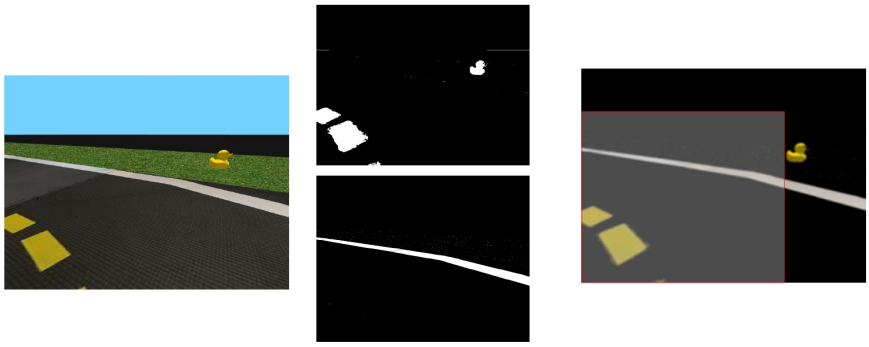

Figure 1: On the left, the original picture, sized $(800, 400)$. We applied different threshold values to filter the yellow color (upper middle), and white color (down), because these color specify the road-lanes. We also added some blur to the picture with kernel size (7×7). The result is on the right. The used area is in the grey rectangle, i.e. the picture is cropped. Lastly, we resized it to $(100, 75)$. This is the input for the CNN network.

4 Implementation

4.1 Acquisition and preparation of data

4.1.1 Imitation learning

The acquisition of the data was performed manually. We used the *duckietown-gym* repo's *manualcontrol.py* script as a baseline. At each timestep, an image was captured by passing a the gym environment's render function the *'rgb'* argument string. The control was modified slightly to add more optimization options for the forward speed. 4 transition steps were introduced, each of them increased or decreased the velocity. At each timestep, we saved these quantities (i.e. the velocity and the image). Our goal was to make a policy, which selects one of the four velocities, and also finds out, if the model needs to turn to the left or the right (i.e. following the lane).

After saving the required data, we manipulated the pictures with opencv library. The process is explained, and visualized on 1 The data acquisition happens in *manual log.py*. The *Howfast.py* is the class of the velocity and helps in logging. The acquired data then preprocessed by *preprocess.py*, and outputs the enhanced images (x data) and matches the file names to the y data, which are the properties (velocity or angle) of the duckiebot's state.

4.1.2 Reinforcement learning

2 key components when training with reinforcement learning are the observation space and the action space. For our implementation the mentionable spaces are the Discrete, and the Box spaces. Both can be n-dimensional vectors. Values of the Box space can take real numeric values, while the values of the Discrete space can take natural numeric values on a given closed interval. The observation space of the duckietown-environment is a Box space, with (camera_height, camera_width, 3) shape, on $[0, 255]$ interval. The action space is a Box space as well, with (2) shape, on a on $[-1, 1]$ interval.

The computation cost of working with the default observation space provided by the duckietown-environment is too big $(480, 640, 3)$, so the RGB image provided was downsampled into a (60, 80, 3) sized image. Then, the upper $1/3$ of the image was removed, as it contained irrelevant information, such as the horizon. White, yellow and red pixels were segmented, but contrary to the imitation learning solution, the acquired masks were not applied onto the RGB image, rather they were placed onto eachother, making a new RGB image, where each channel signals whether the segmented colors were present at the location. This can be viewed as a sort of one-hot encoding, aimed to reduce the possible range of input values.

The action space was also reduced to lighten the computation requirement. The newly used action space was a Discrete space, with 3 possible value, representing forward, right and left movements.

4.2 Training

4.2.1 Imitation Learning

Around 30,000 images has been collected from various maps to train the network, and around 8000 to test its accuracy. The training was performed on a local PC, with an NVIDIA RTX3060. The installation of the CUDA was a bit tricky, and there were some compatibility issues, but these have been resolved over the time. The logging was conducted using Weights and Biases. The *train.py* script calls the model, but before that, it loads the datas, using *initialize.py*. This script is also used to create the required tensor data structure for the test data. We usually trained for around 100 epochs.

4.2.2 Reinforcement learning

The Stable Baselines3 library provides a simple interface for the training of the model. The models can be created with 1 line of code:

model = DQN("CnnPolicy", env, buffer_size=200000, verbose=1, tensorboard_log=log_path, exploration_fraction=0.1, learning_rate=0.00005)

or

model = PPO("CnnPolicy", env, verbose=1, tensorboard_log=log_path)

where the *env)* is the *DuckietownEnvironment)*. Then, the learning can be initiated with:

model.learn(1000000)

TensorBoard was used for logging purposes.

4.3 Evaluation, hyperparameter optimization

4.3.1 Imitation learning

We used *ray* library for hyperparameter optimization. Sadly, the RAM limited our ability to paralellize the algorithm. We use CPU training here. De code can be found in the Train directory in an ipython notebook, named *hipopt.ipynb*. The optimal parameters are 0.0005 for learning rate, and 0.68 for momentum. For the angle model, the learning rate is 0.047, and the momentum is 0.67.

4.3.2 Reinforcement learning

A full training (1000000 timesteps) of a model typically took about 5 hours, so statistically exhaustive examination of different hyperparameters was not achievable. Originally, black color (the asphalt) was also segmented, but after further examination, its presence was not improving the model's performance, so it was taken out. The PPO implementations suffered from *catastrophic forgetting* , where they went from earning good rewards, to performing as good as a newly created model. The replay buffer size of the DQN implementations positively correlated with performance of models, but unfortunately, because of computation resource issues, only a 200000 sized buffer was used, as bigger buffer sizes caused the trainings to crash. The optimal learning rate found was 0.00005 for the DQN, and 0.0003 for the PPO implementation.

4.4 Testing

4.4.1 Imitation learning

We tested the agent on various maps. The duckiebot performed very well on known maps, and was also able to drive on very complex, unkown maps as well (e.g. "udem1" map). The red marks on the road caused a marginal hesitatation. The ducks outside the road were not problematic. The only thing that made the duckiebot to change direction from the lane, were the white trucks near the road. It was not filtered out with the cv2 library. Since the policy we created also used the white lane for coordination, the white trucks were possibly seen as "white lanes", and made the bot to follow them. The mp4 files are videos about the duckie's motion. The agent was not trained to avoid collision with duckies, only lane tracking, but it has done it very well. We used batch-size of 100 for velocities and 32 for angle. The prediction works with 80% accuracy on the test datasheet for velocities, and produces around 40% accuracy for the angle. See Train/log folder in IL_milestone3 branch.

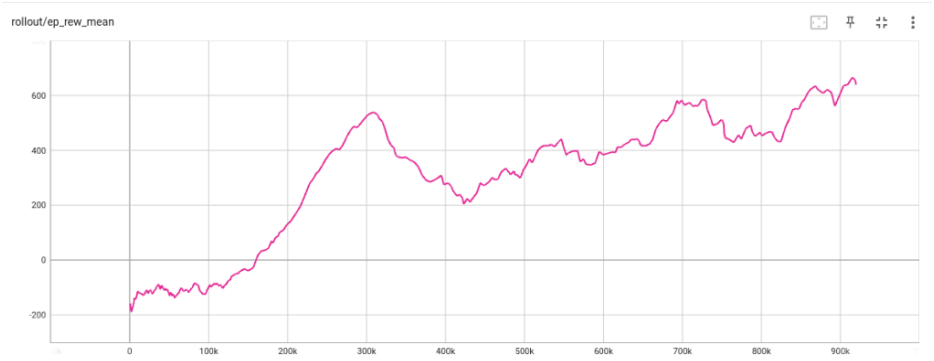

Figure 2: The log of the reward function throughout the training. (The default -1000 penalty for crashing was changed to -30)

4.4.2 Reinforcement learning

The agent was tested on the "loop_empty" (which it was trained on), the "udem1", and the "zigzag_dists" maps. The model performed poorly on all maps, even though the rewards logged (shown in Figure 2 were high. Unfortunately we couldn't find the source of the problem.

5 Overview

In our work, we approached the problem of self-driving with different methods: imitation learning, and reinforcement learning. The drawbacks associated with both of them became apparent. The imitation learning method successfully solved the lane-following problem, but its outputs were not maximally optimal, due to it trying to imitate a suboptimally driving human. The reinforcement learning method has the capability of learning the optimal actions by itself, but the implementation difficulties and the shortage of computation resources only allowed us to create a solution that is only capable of rudimentary lane-following.

6 Future works

Improving the preprocessing of the observation images could improve the performance of the models. With traditional or modern image processing methods, lane recognition could be achieved, thus reducing the input space. The drawback of the imitation learning is that the model will be only as good as the training data. That means, improving the data acquisition will improve our modell as well. A good starting point is e.g. using joystick to drive the duckiebot, since it proved a much smoother way to increase/ decrease speed. If we want other features beside lane following, we could start with LSTM method.

Further work is required for the reinforcement learning method, as it failed to achieve the given task. For this, extra computation resources and further research are needed. Limiting the action space to discrete values prevent the models from reaching the optimal solutions, thus the use of continuous actions are required in future works.

It would be nice to see, how would the models perform, if we initialize the RL policy gradients from a pre-trained IL model, just as the DeepMind team conducted with AlphaGo (see Silver, D. et. al. [10]). Of course, one would need the same policy architecture for both IL and RL policy to do such an experiment.

7 References

[1] Paull, Liam & Tani, Jacopo & Ahn, Heejin & Alonso-Mora, Javier & Carlone, Luca and Čáp, Michal & Chen, Yu Fan & Choi, Changhyun & Dusek, Jeff & Fang, Yajun & Hoehener, Daniel & Liu, Shih-Yuan & Novitzky, Michael & Okuyama, Igor & Pazis, Jason & Rosman,

Guy & Varricchio, Valerio & Wang, Hsueh-Cheng & Yershov, Dmitry & Censi, Andrea. (2017) Duckietown: an Open, Inexpensive and Flexible Platform for Autonomy Education and Research. *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1497-1504

[2] Brockman, Greg & Cheung, Vicki & Pettersson, Ludwig & Schneider, Jonas & Schulman, John & Tang, Jie & Zaremba, Wojciech. (2016). OpenAI Gym. *ArXiv Preprint ArXiv:1606.01540.*

[3] Mnih, Volodymyr & Kavukcuoglu, Koray & Silver, David and Rusu, Andrei & Veness, Joel and Bellemare, Marc & Graves, Alex and Riedmiller, Martin & Fidjeland, Andreas & Ostrovski, Georg & Petersen, Stig and Beattie, Charles & Sadik, Amir & Antonoglou, Ioannis & King, Helen & Kumaran, Dharshan & Wierstra, Daan & Legg, Shane & Hassabis, Demis. (2015). Human-level control through deep reinforcement learning. *Nature 518*, pp. 529–533

[4] Schulman, John & Wolski, Filip & Dhariwal, Prafulla & Radford, Alec & Klimov, Oleg. (2017). Proximal Policy Optimization Algorithms. *arXiv:1707.06347*

[5] LeCun, Y. & Bengio, Y. & and Hinton, G. (2015). Deep Learning. *Nature vol. 521,7553* , pp. 436-44

[6] Raffin, Antonin & Hill, Ashley & Gleave, Adam & Kanervisto, Anssi & Ernestus, Maximilian & Dormann, Noah. (2021). Stable-Baselines3: Reliable Reinforcement Learning Implementations. *J. Mach. Learn. Res. volume 22*, pp. 268:1-268:8

[7] Sun, Liting & Peng, Cheng & Zhan, Wei & Tomizuka, Masayoshi. (2018). A Fast Integrated Planning and Control Framework for Autonomous Driving via Imitation Learning. *V003T37A012. 10.1115/DSCC2018-9249.*

[8] Russell, Stuart J & Norvig, Peter. (2016). Artificial intelligence: a modern approach, pp. 789-823

[9] Arun Dubey & Vanita Jain. (2019) Comparative Study of Convolution Neural Network's Relu and Leaky-Relu Activation Functions, Applications of Computing, Automation and Wireless Systems in Electrical Engineering pp. 873-880

[10] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser,Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, Demis Hassabis. (2016) Mastering the game of Go with deep neural networks and tree search, Nature 16961

[11] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski,Bernhard Firner, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, Karol Zieba. (2016) End to End Learning for Self-Driving Cars, arXiv:1604.07316v1