Robust Reinforcement Learning-based Autonomous Driving Agent for Simulation and Real World

Péter Almási, Róbert Moni, Bálint Gyires-Tóth
Department of Telecommunications and Media Informatics
Budapest University of Technology and Economics, Budapest, HUNGARY
peter.almasi@cs.bme.hu, {robertmoni,toth.b}@tmit.bme.hu

***Abstract*—Deep Reinforcement Learning (DRL) has been successfully used to solve different challenges, e.g. complex board and computer games, recently. However, solving real-world robotics tasks with DRL seems to be a more difficult challenge. The desired approach would be to train the agent in a simulator and transfer it to the real world. Still, models trained in a simulator tend to perform poorly in real-world environments due to the differences. In this paper, we present a DRL-based algorithm that is capable of performing autonomous robot control using Deep Q-Networks (DQN). In our approach, the agent is trained in a simulated environment and it is able to navigate both in a simulated and real-world environment. The method is evaluated in the Duckietown environment, where the agent has to follow the lane based on a monocular camera input. The trained agent is able to run on limited hardware resources and its performance is comparable to state-of-the-art approaches.**

***Index Terms*—deep learning, deep reinforcement learning, DQN, convolutional neural network, robotics, simulation, domain randomization**

I. Introduction

Artificial Intelligence (AI) has become a very focused research area in the past years. Among its subfields, deep learning has been one of the most important ones due to the state-of-the-art results reached in several application scenarios, e.g. image recognition, speech recognition and synthesis, natural language processing and reinforcement learning.

Deep learning has an important role in autonomous driving. The development of deep neural networks and the supporting hardware and software solutions made it possible to build robust computer vision models. Deep convolutional neural networks [1] are outstanding in object detection and semantic segmentation [2] [3]. I.e. it is possible to find and localize certain objects (e.g. cars, pedestrians, cyclists or traffic signs, etc.) on the images, which is critical for autonomous vehicles.

A further method described in [4] trains a convolutional neural network (CNN) to map raw images of the camera directly to steering commands. They use an end-to-end approach, eliminating explicit image preprocessing and object detection steps. This can lead to better performance since the system is trained to maximize overall performance instead of finding manually selected features, e.g. lane detection. [5] uses CNNs to predict simple indicators, that describe the actual road situation (e.g. distance from lane markings and other vehicles, the angle relative to the road). With these indicators, they utilize a controller that can make driving decisions at a high-level. The indicators are more compact than, for example, creating segmentation for the image.

Reinforcement learning (RL) is an area of machine learning where an agent is optimized to take actions in an environment to reach a specified goal, which is represented as a scalar value (called reward). Utilizing deep neural networks in RL enables the agent to be able to learn complex contexts and short and long-term strategies. Recent advances in RL made it possible to achieve superhuman level in complex board and computer games, such as Go [6]. [7] shows an example of an agent, that was trained on a human-level performance to play many kinds of Atari games (e.g. Breakout) by using high-dimensional images as input only. RL agents are also capable of surpassing human players in more complex computer games: the AlphaStar [8] successfully overcome a professional player in the StarCraft II game, and the OpenAI Five [9] can win against human players in the Dota 2 game.

The application of DRL algorithms in autonomous vehicles is currently in an initial phase. Training agents for real-world problems in a simulator is a promising approach, as it is much safer to simulate incidences that must be avoided in the real world (e.g. collisions, running over pedestrians). Also, with sufficient GPU resources, the agents can be trained in a much faster pace than real-time. Collecting sufficient training data is also much more convenient within a simulator. The simulator can also provide additional metrics (e.g. accurate distance between objects and their location) which may be difficult to be measured in the real world but help to evaluate the performance of the agent. However, simulators often have significant differences compared to the real world, and these differences (e.g. details, colors, lighting conditions, or dynamics) can cause the trained models to suffer significant performance degradation in the real world. Training autonomous vehicle agents in simulators with reinforcement learning and transferring the agents to the real world are both active research areas that are in the early stages.

This paper presents a reinforcement learning pipeline for training an agent in an autonomous driving simulator and running the trained agent in the real world. The rest of this paper is organized as follows. Section II gives an introduction to reinforcement learning and domain randomization, Section III describes the details of the proposed method, Section IV presents the Duckietown environment in which the proposed method was tested. Evaluation and results are introduced in

Section V, and conclusions are drawn in Section VI.

II. Background

A. Reinforcement learning

In this paper, we focus on basic RL settings where an agent interacts with an environment by following a policy in order to maximize a reward. There are two main approaches of RL settings: model-based and model-free methods.

Model-based methods construct an internal model of the environment by experiencing transitions and learning the dynamics of the environment. Using this internal model actions are taken either by searching or planning in this world model.

Model-free methods also rely on prior experiences and aim to learn the state-action values, or policy, or both of them. These are the: value learning, policy optimization and actor-critic methods respectively. Value learning methods focus on estimating the value of being in a given state. Policy optimization methods focus on finding the optimal policy that maximizes the expected return, also called reward. The actor-critic method is a hybrid method where a policy optimization method ("actor") is learning from the feedback of the value function ("critic").

Each of these methods are constructed to fit the Markov Decision Process (MDP). We selected the model-free value learning (Q learning) method, which tailored to our setup works as follows: in each timestep t, the environment provides the agent its state s_t in the form of a high-dimensional RGB image. The agent chooses an action a_t from a given set of possible actions according to a policy $\pi(a|s)$. The environment computes a reward r_t based on the chosen action at the given state which describes how 'good' the selected action was. The agent tries to maximize the total reward $R_t = \sum_{u=t}^{t+T} r_u \gamma^{u-t}$, where γ is the discount factor ($\gamma \in [0, 1]$), and T is the length of the episode. The goal during learning is to find an optimal policy π^* which maximizes the expected reward the agent receives in each episode.

Since the observed states of the environment are represented by RGB images and the action space is discrete, we chose the model-free and off-policy Deep Q-Networks [10] [7] method with experience replay. Despite a basic Deep Reinforcement Learning method is utilized, it is capable of learning an optimal policy in simulated traffic environment which performs nearly as good in the real environment by using proper domain randomization methods. In this work, we propose a CNN + DQN system that can be trained with low expenses in a simulated traffic environment and can be transferred to a robot to perform autonomous driving in a miniaturized urban environment.

B. Reinforcement Learning in Autonomous Driving

The new era of Advanced Driving Assistance Systems (ADASs) tend to apply deep learning methods for scene perception, object localization, path planning, behavior arbitration, and motion control [11]. These system gather information from the environment using cameras and localization systems such as RADAR and Li-DAR. Scene perception, object localization, and behavior arbitration are generally supervised learning problems where the training requires large amounts of labeled data. Path planning and motion control can also be tackled using supervised learning methods. One of the first deep learning based autonomous driving solution was ALVINN [12] published in 1989 consisting of a simple 3-layered neural network trained with simulated road images. [13], [14] and [15] applies imitation learning technique to train the agent with a CNN architecture using monocular camera images and recorded human-driver speed and steering angles. This method tends to learn promptly the expert's (human driver) behavior, but performs poorly in new environments. Also the producing, labeling and storing of the training data is both time inefficient and expensive. [16] proposes a CNN+RNN+DQN architecture which performs well in a car racing simulated environment, but the environment is observed from a top-view which is not adequate for real-world scenarios.

C. Domain randomization

Training or transferring robot control agents to the real world is challenging. Domain randomization improves the agent in the training phase in the simulator to have similar performance in the real world. [17] uses this method for object localization on real-world images by training a neural network in a simulator with generated images. The images are highly manipulated (e.g. objects' position, textures, camera position, lighting conditions, etc.) and these manipulated data are used to train the network in the simulator. [18] also utilizes object localization with images generated in a simulator and modified using domain randomization. They show that their method is comparable to the case of training on manually annotated real-world images. [19] uses domain randomization to generate different dynamics in the simulator, and train a neural network which makes a robotic arm able to move objects to an assigned location. They show that training in the simulator with randomized dynamics makes the robotic arm in the real world able to work without any further training on the physical system. [20] solves the problem of transferring models to real-world robots by adapting simulation randomization using real-world data to learn simulator parameter distributions that are suitable for a successful policy transfer. They change the distribution of the simulations in iterative steps to improve policy transfer by matching the policy behavior in simulation and the real world. [21] introduces Automatic Domain Randomization, which utilizes incrementally challenging environments in the simulation. They used this technique to train a robotic arm to solve Rubik's cube. Starting with a single, non-randomized environment, the amount of domain randomization is regularly increased as the model learns to perform well in the previous environments. This way the neural network learns to generalize in randomized environments and becomes able to solve the task in difficult conditions, thus making it possible to successfully transfer the model trained in a simulator to the physical robot.

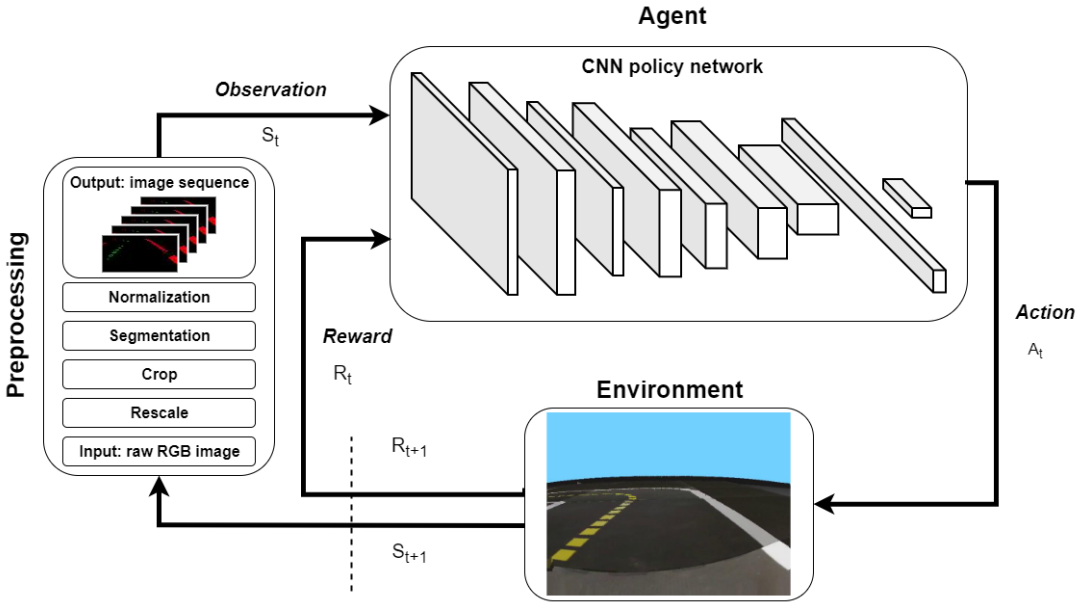


Fig. 1: Overview of our method.

Domain randomization includes diversified technologies to help to transfer the agents trained in the simulator to the real world. Currently, there is no common solution for autonomous driving. Hence, it was our motivation to develop a pipeline to train RL agents in simulators and run them in the real world without severe degradation in performance.

III. Proposed Method

In this section, we present the details of the proposed method. Fig. 1 shows an overview of the proposed pipeline. First, the camera images go through several preprocessing steps. Next, a sequence is formed from the last k camera images, which will be the input of the CNN policy network (the agent). The agent is trained in the simulator with the DQN algorithm based on the reward. The output of the network is mapped to wheel speed commands.

A. Image preprocessing

Before the images of the simulator's monocular virtual camera are fed to the neural networks, we preprocess them. In the case of the real-world monocular camera, the same steps are performed.

- **Resizing**: The images are downscaled from their original size (e.g. 640×480) to a smaller resolution (e.g. 80×60). This step makes training the neural networks and inference faster, while the smaller resolution image still provides enough information for navigating the robot.
- **Cropping**: The part of the image that doesn't contain useful information is cropped. This is typically done above the horizon.
- **Color segmentation**: To make it easier for the neural network to recognize the important parts of the image, the key colors are segmented based on their values. The original image's channels are substituted with the segmented parts. E.g. for lane following, the yellow and the white parts of the image (that define the two lanes) are separated in the image's red and green channels, respectively.
- **Normalization**: The pixel values are normalized to the $[0, 1]$ range, which helps to train the CNN faster.
- **Image sequence**: The last k camera images are concatenated into a 3D tensor with dimensions $(height,\ width,\ sequence\_length \times channels)$. This way a time series is formed from the previous states of the environment, which provides richer information and results in better policy networks, compared to a single instance.

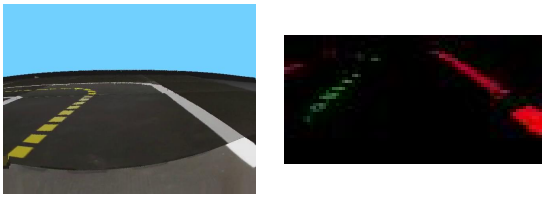


Fig. 2: Original image in the simulator (left) and after preprocessing it (right).

An example of an original and a preprocessed single image is shown in Fig. 2.

B. Policy network

We trained a convolutional neural network with the preprocessed images. The network was designed such that the inference can be be performed real-time on a computer with limited resources. The input of the network is a tensor with the shape of the image sequence $(height,\ width,\ sequence\_length \times channels)$, e.g. $(40, 80, 15)$, which is the

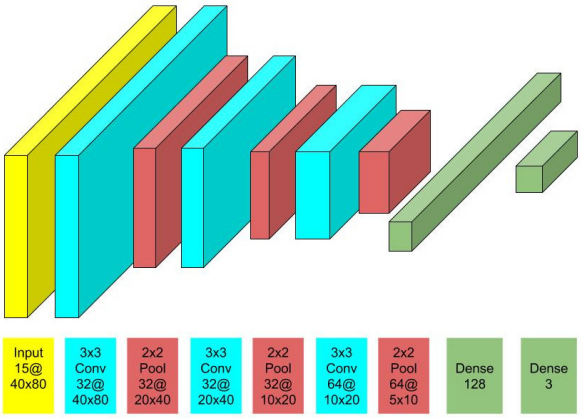


Fig. 3: The policy network we used for modeling the input image sequences.

result of stacking five RGB images. For fast inference and for demonstration purposes we utilized a simple neural network. The neural network consists of three convolutional layers, each followed by ReLU (nonlinear activation function) and MaxPool (dimension reduction) operations. The convolutional layers use $32, 32, 64$ filters with size 3×3. The MaxPool layers use 2×2 filters, so they reduce the size of their input to its $1/4$-th. The convolutional layers are followed by fully connected layers with 128 and 3 outputs. The output of the last layer corresponds to the selected action. The architecture of the policy network is shown in Fig. 3. In the case of more complex environments, the size of the policy network can be increased, indeed.

The output of the neural network (one of the three actions) is mapped to wheel speed commands. The actions correspond to turning left, turning right or going straight, respectively.

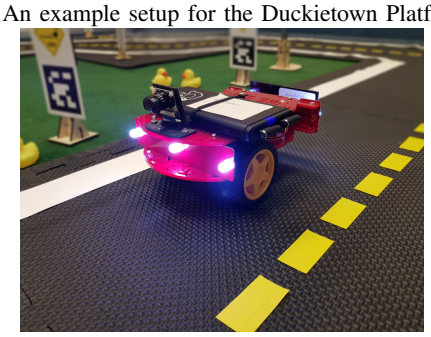
IV. Environment

We used the Duckietown[1] environment for evaluation. Duckietown is an educational and research platform where low-cost robots ('Duckiebots') can travel in a small city ('Duckietown') [22]. The Duckietown and the Duckiebot are shown in Fig. 4. The Duckiebots are small three-wheeled vehicles built almost entirely from off-the-shelf parts. They have only one sensor: a forward-facing wide-angle monocular camera, which they can use to get information about the surrounding objects. The computation is performed by a Raspberry Pi 3, which is responsible for getting the images and controlling the robot. Duckietowns are the cities where the Duckiebots have to operate. These consist of roads, intersections, traffic signs, houses, rubber ducks and other obstacles. The platform is highly flexible: using the standardized road elements, different kinds of cities can be built.

A general goal of the Duckietown project is to provide an environment similar to a real-world autonomous driving environment for a much lower price, which makes it available for educational and research purposes for a wider range

[1]https://www.duckietown.org/

(a) An example setup for the Duckietown Platform.



(b) Duckiebot

Fig. 4: The Duckietown platform and the Duckiebot.

of researchers. While being much cheaper, the environment provides similar challenges to those that are accessible in a more complex autonomous driving platform.

1) Duckietown Simulator: The Duckietown software library contains a Duckietown Simulator [23]. The Simulator provides a similar environment to the real-world Duckietowns: it simulates roads, intersections, obstacles (e.g. vehicles or duckies on the road) and other Duckiebots. Using the simulator, it is possible to manually drive an agent around the map or to test how the trained agent can navigate. The simulator places the robot onto a given map and generates the image that the robot's camera would see in a real-world environment. The robot can be controlled by specifying the speeds of the wheels (two values between -1 and 1 for the two wheels).

2) Real-world Duckietown: After training the models in the Duckietown simulator, we tested the trained agent in a real-world environment, which is shown in Fig. 4a.

Testing in the real-world environment poses new challenges in addition to the simulator. For example, the real-world images seen by the robots are different than those provided by the simulator (a comparison can be seen in Fig. 5.) It is visible that the simulator images, while being similar to the real ones, has different lighting conditions, camera angle, and has a simpler setup regarding the objects surrounding the track, which makes it harder to transform the agent to the real-world robot. Another challenge that arises when evaluating on the real robot is that the robot has to be controlled in real-time: while in the simulator it is possible to simulate a slower algorithm, in the real world, the camera image must be processed in a few milliseconds, which means that the neural network has to be designed carefully such that it can predict

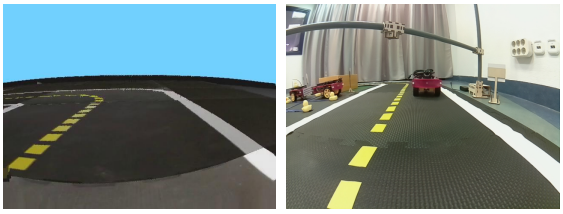


Fig. 5: Images from the Duckietown simulator (left) and the camera of the robot (right). The robot's only sensor is its camera, so it has to be controlled based only this information. Notice that the real-world image has different lighting conditions and camera angle, which makes transferring the trained models to the real robot more difficult.

TABLE I: Discrete actions predicted by the DQN algorithm are mapped to wheel speeds. (Maximum speed is 1.0.)

Action	Left wheel speed	Right wheel speed
0 (Left)	0.04	0.4
1 (Right)	0.4	0.04
2 (Straight)	0.3	0.3

driving commands from the camera images fast. Currently, the images are processed on a x86 CPU (instead of the robot's Raspberry Pi computer), which results in smaller inference time. It is also worth noting that there are other factors which make it harder to evaluate the solutions on the real robot compared to the simulator (e.g. network delays).

3) Training details: We used the DQN implementation available in the Stable Baselines collection [24]. The algorithm chooses one of three possible actions at each timestep; the mapping between these actions and the robot wheel speeds can be seen in Table I. We experimented with different hyperparameter settings, including different values and settings for the learning rate, input image size, experience replay buffer size, image segmentation parameters, camera distortion, discount value, policy network parameters, and wheel speeds. The parameters that gave the best results were the following. We used a batch size of 32, gamma=0.99, the learning rate for the Adam optimizer was set to 0.00005, the size of the replay buffer was 50000, and the agent collected 10000 steps of experience from random actions before actually starting learning. We ran the training for 500000 timesteps, which took approximately 40 hours on an NVIDIA DGX Workstation, which contains 4 pieces of V100 GPUs.

The simulator provides a reward function, which can be used for reinforcement learning-based methods. This reward reflects how accurately the agent follows its lane. When the agent is going in the right lane, it receives a positive reward. When it starts to drift away from the optimal curve, it receives smaller rewards; when it goes to the oncoming lane, it receives smaller negative rewards, and it gets penalty when it leaves the track (the simulated episode also ends at this point).

We slightly modified the default reward provided by the simulator. When the robot is in the right lane, the reward is calculated according to the following formula:

$$reward = 10 \cdot speed \cdot dot\_dir - 100 \cdot dist + 400 \cdot col\_pen,$$

where $speed$ is the speed of the robot in the simulator, dot_dir is calculated as the dot product of the vectors pointing towards the heading of the robot and the tangent of the curve, $dist$ is the distance from the center of the right lane, and col_pen is a penalty for collisions. When the robot is not in the right lane, the reward is:

$$reward = 400 \cdot col\_pen.$$

When the robot leaves the track, it gets a reward of -40.

The rewards achieved throughout the episodes of training phase are shown in Fig. 6.

Fig. 6: Received rewards during the episodes of the training of the agent.

V. Results

Our primary goal was to train an agent in the simulator which can navigate the robot along the track both in the simulator and the real world. We tested our method on several maps, different from the one we used during training, to eliminate the possibility of overfitting to one single map. We trained on a larger, more complicated map, to make it possible for the network to learn diverse turns and road situations. A video of our robot in action can be seen at http://bit.ly/wcci20duckie.

1) Performance in the simulator and in real world: We used three maps for testing: Map #1, Map #2 and Map #3 can be seen in Figs. 7a-7f, 7g and 7h respectively. The real-world environment is built according to Map #1 (see Fig. 4a). We placed the robot on 50 randomly selected positions of the map and counted the number of occasions it was able to drive for a complete lap on the track. We excluded those randomly generated situations from our tests where the robot was dropped to the side of the track facing outwards, where it was impossible to navigate back to the track. In the simulator, we limited the lengths of the episodes for 2500 and 3500 timesteps (approx. 50-70 seconds), which is enough time for the robot to take a whole lap on the maps. In the real-world environment, we ran the evaluation for 45 seconds, which also

TABLE II: Rates of successful drives on three simulated and one real-world map.

Environment	Tests		
	Total	Successful	Success rate
Simulator Map #1	50	38	76%
Simulator Map #2	50	49	98%
Simulator Map #3	50	41	82%
Real world	50	48	96%

was enough for completing a full lap. The results of our tests can be seen in Table II. We also ran two longer tests in the simulator (50000 timesteps) and found that when the robot was able to take at least one complete lap, it was also able to drive for 50000 timesteps without leaving the track. Therefore, we decided to test our model more thoroughly for only the duration of one complete lap, since after it takes one lap, it has successfully gone through all parts of the track, and we can assume that it could do the same in the following laps too.

2) Agent navigation patterns: Fig. 7 shows the paths of the robot after starting it from various randomized locations in the simulator. The robot was able to navigate to the center of its lane and drive a whole lap there even when it was started from an invalid location, the oncoming lane (see 7e). Fig. 8 shows the paths of the real-world robot. To created these images, we placed an ArUco marker [25] [26] on the top of the robot. We created a plan view video of the evaluation with a fixed camera and ran the ArUco detector algorithm for each frame of the video. The found locations of the marker are drawn on the map. Although occasionally the robot touched the middle yellow line during the evaluation, mostly it successfully ran in its lane. (As the camera was not completely above the robot all the time, the images have a small distortion: the marker on the top of the robot is not projected directly under the center of the robot due to the angle between the robot and the plan view camera.)

3) Analysis of the predictions: Fig. 9a shows the histogram of the probabilities predicted by the model for the selected action in the simulator, separated by the predicted actions. Fig. 9b shows the same metrics for the real world. Both histograms are made from the predictions generated while driving one lap on the track. We assume that the differences are due to the physical aspects of running the agent in the real world. To predict the *Straight* action, the robot has to be approximately in the middle of the lane, facing forward (otherwise, it can get closer to the middle of the lane by turning, and thus can get a higher reward). Positioning to the middle of the lane is easier in the simulator, as the agent generates predictions and gives commands more frequently (around 2500 times in the simulator and 700 times in the real world during one whole lap), which means that more corrections are required in the real world to stay in the lane. Also, the control commands have an unsteady delay in the real world consisting of the network delay and inference time, which results in the need for more corrections accomplished by generating turning commands.

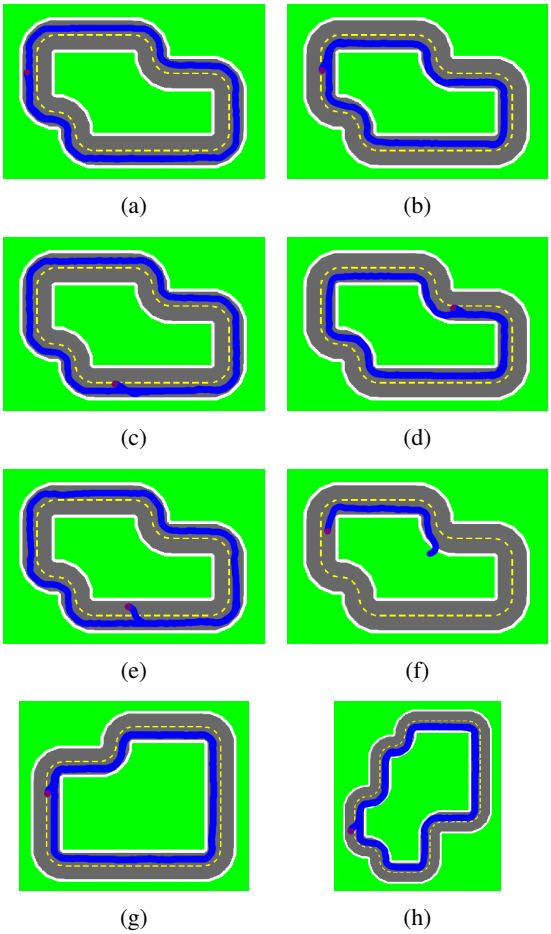
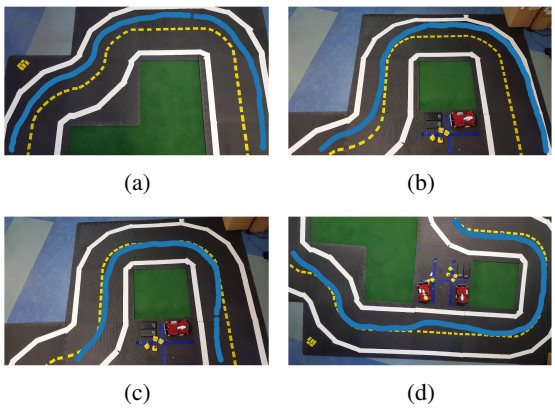


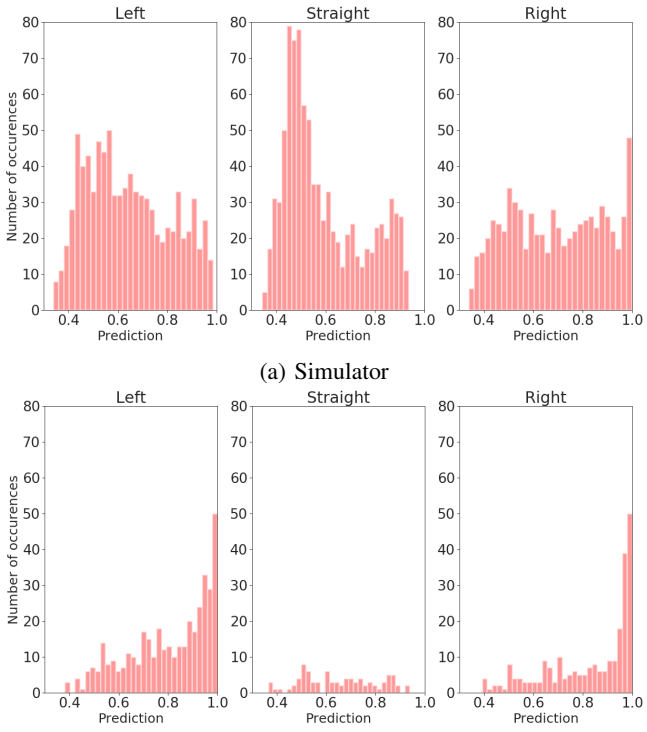
Fig. 7: The paths of the robot on the maps after starting it from different locations in the simulator. The initial location of the robot is marked red. 7a-7e, 7g, 7h show attempts where the robot was able to drive successfully a complete lap, with 7e showing a situation where the robot was started from an invalid position but still was able to go to the right lane. 7f shows a scenario where the robot failed to take a whole lap. 7a-7f, 7g and 7h show Map #1, Map #2 and Map #3, respectively.

4) Comparing with state-of-the-art results: The AI Driving Olympics (AI-DO) is a series of competitions focusing on AI for self-driving vehicles in the Duckietown environment [27]. The competition had three rounds in 2018 and 2019 organized at the NeurIPS and ICRA conferences. The goal of the competition is to make it possible to test the recent theoretical advances in the area in practice. Different tasks can be solved at the competition: the simplest is lane following, but more complex ones, such as navigating in the presence of other vehicles and handling intersections, are also available. In the case of lane following, the task is to process the image of the robot's camera and give wheel speed commands based on it to navigate the robot on the map. The competition makes it possible to compare different methods and algorithms and evaluate their performance.



(a) (b) (c) (d)

Fig. 8: The paths of the real robot on the map after starting it from different locations in the real-world environment.



(a) Simulator

(b) Real world

Fig. 9: Histogram of the predictions of the model in the simulator (9a) and in the real world (9b), separated based on the predicted action.

We compared our method to the ones that were used by the winners of the 3rd AI Driving Olympics. In the competition, the submissions were tested on a real robot for 30 seconds. The goal was to drive as far as possible without leaving the track in the given time limit. The driven distance was discretized to the number of map tiles the robot successfully passed. The submissions were tested from two different initial locations. While we are not able to test our method on the same track, the competition track and ours are built from the same standardized elements (straight roads and turns). We performed the tests by starting our robot from two different initial locations for 30 seconds, and measured the distance it covered. The results can be seen in Table III. The top submissions in the competition mostly used imitation learning-based methods in contrast to our reinforcement learning-based approach. It is worth noting that while our method is currently not optimized for driving the robot as fast as possible, it has comparable performance to the state-of-the-art solutions.

TABLE III: Results of the best performing, state-of-the-art agents in AI-DO 3 competition and our approach.

Team	Distance driven (tiles) Run #1	Run #2
JBRRussia1	11	19
phmarm	10	18
JBRRussia	8	2
miksaz	8	1
Our approach	12	13

5) Other experiments: We ran several experiments to find the best training parameters and image preprocessing method. In the following, we present our experiences regarding the use of each image preprocessing step. *Resizing* the image is required to make it possible to control the robot in real-time with as little latency as possible. We used a single laptop with no dedicated graphics card and a 4-core Intel®Core™i7-4500U CPU @ 1.80GHz. We measured that creating one prediction takes approx. 3-4 milliseconds, which is adequate with the camera frames arriving at a rate of 30fps. *Cropping* the upper part of the images helps to transfer to the real robot, as this step eliminates most of the objects around the track that would otherwise fall into the field of view of the robot and thus could make navigating more difficult. The *color segmentation* also improves the transfer to the real robot, as it highlights the important parts of the image and hides the objects surrounding the track. *Normalization* is required for the more effective training of the neural network. The *image buffer* has an important role in stabilizing the movement of the robot. Without this, the agent usually navigates in a straight line by alternating between actions 0 and 1 (according to Table I), which results in an oscillating movement. Using the image buffer helps the robot to find the center of its lane and go straight there; while the oscillating movement sometimes still occurs, it is much less frequent than in the absence of the image buffer. The oscillating movement of the robot can be smoothed by using a larger number of discrete actions (e.g. 5 or 7), however, it makes training the agent more difficult.

We tested the robustness of our method by running experiments in real-world environment in different lighting conditions and robot speeds. We changed the lighting conditions by varying the number and position of the lights turned on in the evaluation room. We changed the speed of the robot by multiplying the original speed values with constants, thus, making the general movement of the robot slower or faster. We found that these changes had no effect on the performance

of the robot, i.e. it produced similar performance with and without these changes.

VI. Conclusions

The difference between the simulator and the real world is a major challenge for applications of reinforcement learning in robotics and autonomous driving. In this paper, we presented a pipeline for a Deep Reinforcement Learning-based algorithm to perform autonomous robot control using Deep Q-Networks. We proposed a method to train the agent in a simulator, which can later control the robot both in the simulated and the real-world environment. We used the Duckietown environment to evaluate our method. We showed that using the proposed approach, the trained model is capable of navigating the robot along the track both in the simulator and the real world. Our method has comparable performance to the state-of-the-art solutions and can be run real-time on limited hardware resources.

Acknowledgements

The research presented in this paper has been supported by Continental Automotive Hungary Ltd., by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013), by the BME-Artificial Intelligence FIKP grant of Ministry of Human Resources (BME FIKP-MI/SC). Bálint Gyires-Tóth is supported by the Doctoral Research Scholarship of Ministry of Human Resources (ÚNKP-19-4-BME-189) in the scope of New National Excellence Program and by János Bolyai Research Scholarship of the Hungarian Academy of Sciences. Péter Almsi expresses his gratitude for the financial support of the Nokia Bell Labs Hungary.

References

[1] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, *et al.*, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[2] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds., Curran Associates, Inc., 2015, pp. 91–99. [Online]. Available: http://papers.nips.cc/paper/5638-faster-r-cnn-towards-real-time-object-detection-with-region-proposal-networks.pdf.

[3] L.-C. Chen, Y. Zhu, G. Papandreou, F. Schroff, and H. Adam, "Encoder-decoder with atrous separable convolution for semantic image segmentation," in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 801–818.

[4] M. Bojarski, D. Del Testa, D. Dworakowski, *et al.*, "End to end learning for self-driving cars," *arXiv preprint arXiv:1604.07316*, 2016.

[5] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "Deepdriving: Learning affordance for direct perception in autonomous driving," in *Proceedings of the IEEE International Conference on Computer Vision*, 2015, pp. 2722–2730.

[6] D. Silver, J. Schrittwieser, K. Simonyan, *et al.*, "Mastering the game of go without human knowledge," *nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[7] V. Mnih, K. Kavukcuoglu, D. Silver, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[8] O. Vinyals, I. Babuschkin, W. M. Czarnecki, *et al.*, "Grandmaster level in starcraft ii using multi-agent reinforcement learning," *Nature*, vol. 575, no. 7782, pp. 350–354, 2019.

[9] C. Berner, G. Brockman, B. Chan, *et al.*, "Dota 2 with large scale deep reinforcement learning," *arXiv preprint arXiv:1912.06680*, 2019.

[10] V. Mnih, K. Kavukcuoglu, D. Silver, *et al.*, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[11] G. Sorin, T. Bogdan, C. Tiberiu, and M. Gigel, "A survey of deep learning techniques for autonomous driving," *arXiv preprint arXiv: 1910.07738*, 2019.

[12] D. Pomerleau, "Alvinn: An autonomous land vehicle in a neural network," in *NIPS*, 1988.

[13] M. Bechtel, E. McEllhiney, and H. Yun, "Deeppicar: A low-cost deep neural network-based autonomous car," Dec. 2017.

[14] M. Bojarski, P. Yeres, A. Choromanska, *et al.*, "Explaining how a deep neural network trained with end-to-end learning steers a car," Apr. 2017.

[15] Y. Pan, C.-A. Cheng, K. Saigol, *et al.*, "Agile autonomous driving using end-to-end deep imitation learning," Jun. 2018. DOI: 10.15607/RSS.2018.XIV.056.

[16] A. Sallab, M. Abdou, E. Perot, and S. Yogamani, "Deep reinforcement learning framework for autonomous driving," *Electronic Imaging*, vol. 2017, pp. 70–76, Jan. 2017. DOI: 10.2352/ISSN.2470-1173.2017.19.AVM-023.

[17] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain randomization for transferring deep neural networks from simulation to the real world," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2017, pp. 23–30.

[18] J. Tremblay, A. Prakash, D. Acuna, *et al.*, "Training deep networks with synthetic data: Bridging the reality gap by domain randomization," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2018, pp. 969–977.

[19] X. B. Peng, M. Andrychowicz, W. Zaremba, and P. Abbeel, "Sim-to-real transfer of robotic control with dynamics randomization," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2018, pp. 1–8.

[20] Y. Chebotar, A. Handa, V. Makoviychuk, *et al.*, "Closing the sim-to-real loop: Adapting simulation randomization with real world experience," in *2019 International Conference on Robotics and Automation (ICRA)*, IEEE, 2019, pp. 8973–8979.

[21] I. Akkaya, M. Andrychowicz, M. Chociej, *et al.*, "Solving rubik's cube with a robot hand," *arXiv preprint arXiv:1910.07113*, 2019.

[22] L. Paull, J. Tani, H. Ahn, *et al.*, "Duckietown: An open, inexpensive and flexible platform for autonomy education and research," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2017, pp. 1497–1504.

[23] M. Chevalier-Boisvert, F. Golemo, Y. Cao, B. Mehta, and L. Paull, *Duckietown environments for openai gym*, https://github.com/duckietown/gym-duckietown, 2018.

[24] A. Hill, A. Raffin, M. Ernestus, *et al.*, *Stable baselines*, https://github.com/hill-a/stable-baselines, 2018.

[25] S. Garrido-Jurado, R. Muñoz-Salinas, F. J. Madrid-Cuevas, and R. Medina-Carnicer, "Generation of fiducial marker dictionaries using mixed integer linear programming," *Pattern Recognition*, vol. 51, pp. 481–491, 2016.

[26] F. J. Romero-Ramirez, R. Muñoz-Salinas, and R. Medina-Carnicer, "Speeded up detection of squared fiducial markers," *Image and vision Computing*, vol. 76, pp. 38–47, 2018.

[27] J. Zilly, J. Tani, B. Considine, *et al.*, "The ai driving olympics at neurips 2018," *arXiv preprint arXiv:1903.02503*, 2019.