Navigating Autonomous Vehicle at the Road Intersection Simulator with Reinforcement Learning

Michael Martinson[1], Alexey Skrynnik[2], and Aleksandr I. Panov[1,2(✉)]

[1] Moscow Institute of Physics and Technology, Moscow, Russia
panov.ai@mipt.ru

[2] Artificial Intelligence Research Institute, Federal Research Center "Computer Science and Control" of the Russian Academy of Sciences, Moscow, Russia

Abstract. In this paper, we consider the problem of controlling an agent that simulates the behavior of an self-driving car when passing a road intersection together with other vehicles. We consider the case of using smart city systems, which allow the agent to get full information about what is happening at the intersection in the form of video frames from surveillance cameras. The paper proposes the implementation of a control system based on a trainable behavior generation module. The agent's model is implemented using reinforcement learning (RL) methods. In our work, we analyze various RL methods (PPO, Rainbow, TD3), and variants of the computer vision subsystem of the agent. Also, we present our results of the best implementation of the agent when driving together with other participants in compliance with traffic rules.

Keywords: Reinforcement learning · Self-driving car · Road intersection · Computer vision · Policy gradient · Off-policy methods

1 Introduction

Control architectures for mobile robotic systems and self-driving vehicles currently allow us to solve basic tasks for planning and self-driving in complex urban environments. Often the applied methods are based on pre-defined scenarios and rules of behavior, which significantly reduces the degree of autonomy of such systems. One of the promising areas for the increasing degree of autonomy is the use of machine learning methods. These methods are using for automatically generating generalized object recognition procedures, including dynamic ones, in the external environment. A significant disadvantage of such approaches is the need for pre-training on pre-generated data, which often requires handcrafted markup. However, there are currently a large number of data sets and simulators that can be used for pre-training without significant manual configuration or markup.

In this paper, we consider the task of learning an agent that simulates a self-driving car that performs the task of passing through the road intersection.

S. O. Kuznetsov et al. (Eds.): RCAI 2020, LNAI 12412, pp. 71–84, 2020.
https://doi.org/10.1007/978-3-030-59535-7_6

As a basic statement of the problem, we consider a realistic scenario of using data from the sensors of the agent (images from cameras within the field of view, lidars, etc.), data coming from video surveillance cameras located in complex and loaded transport areas, in particular at road intersections. The considering scenario for agent behavior looks followed. The agent drives up to the intersection and connects to the surveillance cameras located above the intersection to receive an online video stream. The agent switches to driving mode for a dangerous area and uses a pre-trained model that uses data from the agent's camera and sensors to follow traffic rules and pass the intersection in the shortest possible time. In this paper, we describe the simulator which we developed for this case. Also, we investigate methods based on reinforcement learning approaches to generate such agent.

We analyzed the effectiveness of using computer vision methods to generate an agent's environment description and conducted a series of experiments with various reinforcement learning methods, including policy gradient (PPO) and off-policy methods (Rainbow).

We did not link our research to any specific robot or self-driving architecture. At the same time, we consider that a real robot will have some simple sensors (speed, coordinate estimation, etc.) and basic control operations (wheel rotation, acceleration, braking).

Also, we understand that the use of such systems for ordinary crossroads with people is unlikely to become legally possible soon. Therefore, we propose to consider the task in the context of a "robotic" intersection without any pedestrians. Note that this assumption does not make the problem less relevant since it is fully applicable to delivery robots.

The presentation is structured as follows. Section 2 provides a brief overview of reinforcement learning methods, simulators, and approaches for modeling intersections. Section 3 presents the RL methods used in this paper. In Sect. 4, we describe the environment and the main parameters of the simulator. Section 5 presents the main results of the experiments.

2 Related Works

The direct launch of learning methods on robotic platforms and self-driving vehicles in the real world is expensive and very slow. In this regard, various simulators are widely used, which would reflect the interaction of the agent with the environment as realistically as possible. Such works include Carla [4] and simulator Nvidia Drive which used in work [2]. These 3D simulators have a huge number of settings and can generate data from many different sensors - cameras, lidars, accelerometers, etc. The disadvantage of this is the large computing power required only for the operation in these environments.

A representative of a slightly different class of simulators is SUMO [13]. This simulator permits to simulate large urban road networks with traffic lights and control individual cars.

The task of managing the self-driving car can be divided into several subtasks, which are more or less covered and automated in modern works. For example,

the paper [9] investigates an agent driving a car in a TORCS [12] environment based on a racing simulator. The works [1,15,21,22] investigate the ability of an agent to change lanes, and [21] continue this study in cooperative setup. The work [16] explores the mechanism of keeping the car on the track. Paper [23] the authors provide a comparison of various computer vision methods, which include car detection and methods for evaluating the angles of car rotation. It is also necessary to mention a large recent work [8], which reviewed many traffic simulators and agents.

Despite the abundance of existing methods and solutions for sub-tasks of managing self-driving agents, we believe that the multi-agent formulation of the problem of moving agents at the intersection is quite popular.

3 Background

The Markov decision process [19] (MDP) is used to formalize our approach for learning agents. MDP is defined as a tuple $\langle S, A, R, T, \gamma \rangle$, which consists of a set of states S, a set of actions A, a reward function $R(s_t, a_t)$, a transition function $T(s_t, a_t, s_{t+1}) = P(s_{t+1}|s_t, a_t)$ and the discount coefficient γ. In each state of the environment $s_t \in S$, the agent performs the action $a_t \in A$, after which it receives a reward according to R and moves to the new state s_{t+1}, according to T Agent policy π determines which action the agent will choose for a specific state. The agent's task is to find a π that maximizes the expected discounted reward during interaction with the environment. In our work, we will consider episodic environments – MDP in which the agent's interaction with the environment is limited to a certain number of steps.

There are many algorithms to find the optimal policy π. In this paper, we consider modern approaches based on the Value function (Value-Based Methods) and approaches based on the policy gradient (Policy Gradient Methods).

The Q-function $Q^{\pi}(s, a)$, for the state-action pair, estimates the expected discounted reward that will be received in the future if the agent chooses the action a, in the state s and will continue its interaction with the environment, according to the policy π. The optimal Q-function $Q^*(s, a)$, can be obtained by solving the Bellman equation:

$$Q^*(s_t, a_t) = \mathbb{E}\left[R(s_t, a_t) + \gamma \sum_{s'} P(s_{t+1}|s_t, a_t) \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1})\right].$$

The optimal policy is $\pi(s_t) = argmax_{a_t \in A} Q^*(s_{t+1}, a_{t+1})$. In modern works for approximating the Q-function $Q(s_t, a_t)$ uses Deep Q-network (DQN) [14]. To evaluate $Q(s_t, a_t)$ the neural network receives the input state s_t and predicts the utility for each action $Q_\theta(s_t, a_t)$, where θ are the parameters of the neural network. We consider classic algorithms, such as Rainbow [7], which is applicable for discrete action space and Twin Delayed Deep Deterministic Policy Gradients (TD3) [6], which allows to use a continuous set of actions.

Rainbow [7] – is an improvement on the classic DQNalgorithm. The loss function for the DQN algorithm has the form:

$$L_{DQN} = \left[Q(s_t, a_t) - (R_t + \gamma \cdot max_{a'} Q_{target}(s_{t+1}, a')) \right]^2,$$

The learning process consists of interacting with the environment and saving all tuples (s_t, a_t, R_t, s_{t+1}) in memory of replays. where R_t – reward at time t, Q_{target} – copy of Q(S,A), delayed for episodes. Q-function optimization is performed using batches that are uniformly sampled from the replay buffer. The authors of Rainbow consider 6 improvements to the DQN algorithm, a combination of which significantly accelerate its convergence:

1. Double DQN is designed to solve the problem of overestimation that exists in DQN due to the maximization step. To solve this problem, two Q networks are used: Q_θ and $Q_{\bar{\theta}}$. When performing the maximization step, the best action is selected based on the current network, and its value is calculated based on the other one:

$$L_{double} = \left[Q_\theta(s_t, a_t) - (R_t + \gamma \cdot Q_{\bar{\theta}}(s_{t+1}, \arg\max_{a'} Q_\theta(s_{t+1}, a')) \right]^2.$$

   The network, for which the loss function will be applied at the current update step is selected by random.
2. Prioritized Experience Replay improves the standard replay buffer of the DQN algorithm. Prioritized replay buffer samples more often transitions, with a larger TD error. The probability of sampling a single transition is defined as:

$$p_t \propto \left| R_{t+1} + \gamma_{t+1} \max_{a'} Q_{\bar{\theta}}(s_{t+1}, a') - Q_\theta(s_t, s_t) \right|^\omega,$$

   where ω is a hyperparameter that defines the distribution form. New data entering the prioritized buffer gets the maximum sampling probability.
3. Dueling Network Architecture – the approach is to make two calculation streams, the value stream V and the advantage stream a_ψ. They use a common convolutional encoder and are combined by a special aggregator, which corresponds to the following factorization of the Q-function:

$$Q_\theta(s, a) = V_\eta(f_\xi(s)) + a_\psi(f_\xi(s), a) - \frac{\sum_{a'} a_\psi(f_\xi(s), a')}{N_{\text{actions}}},$$

   where ξ, η, and ψ are common encoder parameters f_ξ, v_η value function flow, a_ψ advantage function flow, and $\theta = \{\xi, \eta, \psi\}$ their concatenation.
4. N-step return – uses N-step evaluation, which is defined as:

$$R_t^{(n)} \equiv \sum_{k=0}^{n-1} \gamma_t^{(k)} R_{t+k+1}.$$

   So the new loss function:

$$L_{\text{N-step}} = \left[m(s_t, a_t) - (R_t^{(p)} + \gamma_t^{(p)} \max_{a'} Q_{goal}(s_{T+N}, a')) \right]^2.$$

5. Distributional RL – a distribution-based approach – the algorithm does not predict the Q-function itself, but its distribution. In this case, the C51 algorithm was used.
6. Noisy nets – this approach adds the layer to the neural network that is responsible for exploring the environment. The Noisy Nets approach offers a linear network layer that combines deterministic input and noise input:

$$\mathbf{y} = (\mathbf{b} + \mathbf{W}\mathbf{x}) + (\mathbf{b}_{noisy} \odot \epsilon^b + (\mathbf{W}_{noisy} \odot \epsilon^w)\mathbf{x}),$$

where ϵ^b and ϵ^w are sampled from standard normal distribution, and $\odot$ means elementwise multiplication. This transformation can be used instead of the standard linear transformation $\mathbf{y} = \mathbf{b} + \mathbf{W}\mathbf{x}$. The idea is that over time, the network learns to ignore the flow of noise, but the adjustment to noise occurs in different ways for different parts of the state space.

The second method we use is Proximal Policy Optimization (PPO) [17]. This is an on-policy method that belongs to the Actor-Critic (AC) class. The critic predicts the Value-function V and the loss function for it is MSE:

$$L_{critic} = [V(s) - R_t - \gamma \cdot V_{target}(s_{t+1})]^2$$

The actor loss function for PPO is similar to the improvement of AC - Advantage Actor-Critic (A2C) [20] and uses the advantage function, but with additional modifications.. As one of the main modifications is a clipping of possible deviation from the old policy. So if the standard A2C actor loss function has the form:

$$L_{policy} = \mathbb{E}\left[\pi(s_t, a_t) \cdot A(s_t, a_t)\right],$$

then for PPO:

$$L_{policy} = \mathbb{E}\left[clip(\frac{\pi(s_t, a_t)}{\pi_{old}(s_t, a_t)}, 1 - \varepsilon, 1 + \varepsilon) \cdot A(s_t, a_t)\right],$$

where P is the actor's policy, $clip(a, b, c) = \min(\max(a, b), c)$. Our implementation also used some PPO improvements from [5], such as clipping not only the actor policy but also the Value-function.

The third method, which was applied, is already introduced TD3 [6] - an off-policy algorithm that makes several stabilizing and convergence-accelerating improvements to the Deep Deterministic Policy Gradient (DDPG) [11]. Both methods belong to the Actor-Critic class. And correspondingly for DDPG Critic train by minimizing the almost standard loss function:

$$L_{critic} = [Q(s_t, , a_t) - R_r - \gamma \cdot Q_{target}(s_{t+1}, \pi(s_{t+1}))]^2,$$

where the only difference is the presence of $P(s_{t+1})$ - prediction by the policy of the action actor from the state s_{t+1}. For the actor itself DDPG use:

$$L_{policy} = -Q(s_t, \pi(s_t))$$

from which the gradient for policy parameters is taken.

TD3 introduces 3 more major changes. First is adding white noise to policy predictions at the stage of L_{policy} calculation. The second is a delayed update of the actor - namely one policy update for $n \in \mathbb{N}$ of the critic updates. And the last one is the usage of two independent estimations for the Q-function and using the minimum of them in calculations, as the authors of TD3 claim this helps to reduce the impact of bias overestimation.

4 Environment Description

Our environment - CarInersect is based on the simulator [18], physical engine Box2D and the OpenAI gym framework [3]. Technical details and bot behavior are described in the Sect. 4.1. The environment simulates the behavior of cars at a road intersection. The agent's goal is to manage one of these cars and drive it along the specified track. Although a human can easily complete such tacks, this is difficult for an artificial agent. In particular, when testing the environment, we found that a reward system different from a dense line of check-points with positive reward hardly leads to agent convergence.

4.1 Technical Details

The pybox2d physics engine is used for physical simulation of collisions, accelerations, deceleration, and drift of cars. So after a long selection of constants to control the car, we still could not achieve ordinary behavior, so we used the code for calculating the forces acting on the car from the OpenAI gym CarRacing.

The environment has the same functionality as the OpenAI gym framework environments. Every environment settings are passed through the configuration file. This file consists of three parts: the first part describes the reward function; the second part describes the behavior of the environment - the number of bots, their tracks, the agent's track, the type of observation; and the third part describing the background image, its markup, and sets of images of bots and the agent. The tracks description is the usual CVAT XML markup of the background image.

Bots ride along their tracks. These tracks selected uniformly from the list of available ones in a moment of bot creating. When moving, the bot goes to the next checkpoint of its track. To decide, where to steer, it takes into account the next two checkpoints. When bot leaves the road or collides with other vehicles, it disappears. If the agent encounters a bot, the agent receives a fine (reward -1), and the episode ends prematurely. Bots give priority to the car approaching from the right.

4.2 Actions

Each action is represented by a tuple $a_t = (st_t, g_t, b_t)$, where:

- $st_t \in [-1, 1]$ - steering
- $g_t \in [0, 1]$ - gas, has impact to acceleration
- $b_t \in [0, 1]$ - brake, stopping the car (not immediately)

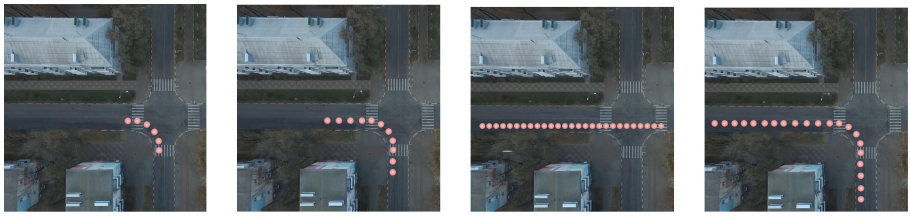

Fig. 1. Types of tracks: small rotation, medium rotation, line, rotation (or full rotation); the control points are marked in red; if reaching them give 0.5 reward each

4.3 State

As a state, the environment returns an image and the agent's car feature vector. A feature vector is formed using the computer vision subsystem of the agent [23]. All vector features are concatenated. All coordinates are normalized to $[0 - 1]$; all angles are set by 3 numbers: their value in radians, sin and cos; the car points are the center of the hull and the centers of 4 wheels. Possible vector features (Fig. 1):

- hull_position – two numbers, x and y coordinates of the center of the car
- hull_angle – the angle of rotation of the car
- car_speed – two numbers, speeds on x and y coordinates normalized to 1000
- wheels_positions – 8 numbers - 4 pairs of x and y coordinates of the car wheels
- track_sensor – 1 if all car points are inside the track polygon, 0 otherwise
- road_sensor – 1 if all car points are inside the polygon of the road, 0 otherwise
- finish_sensor – 1 if at least one car point is close to the last point of the track
- cross_road_sensor – 1 if at least one car point is inside the area marked as an intersection area
- collide_sensor – 1 if the car is currently colliding with another car, otherwise 0
- car_radar_$\{N = 1, 2, 3\}$ – each of the N radar vectors is 6 numbers describing a single car:
  1. 0 or 1 is there data, if 0, then the other 5 numbers are 0
  2. normalized distance to the object
  3. sin and cos of the relative angle
  4. sin and cos of the angle between the velocity vectors
- time – 3 numbers, sin of time and sin of doubled and tripled time, where time itself is an integer number of steps since the creation of the car (this time encoding is done by analogy with the position encoding in Natural Language Processing [10]).

4.4 Reward Function

The reward system for the environment is defined in the configuration file. We used the same reward system for all agents: 0.5 for reaching checkpoints, which are uniformly spaced along the track; 2 for reaching the final point; −1 for leaving the track and crash. The episode ends when the agent reaches the finish line, leaves the track, or after 500 steps.

4.5 Environment Performance

To measure performance, the environment was run for 100,000 steps on a computer with an Intel ® Core™ i5-8250U CPU @ 1.60 GHz 8, 15.6 GB RAM. Table 1 shows the average number of frames per second for various environment configurations. A slight slowdown occurs when images are used as the state. A significant slowdown occurs when bots are added.

Table 1. Simulator performance – mean number of frames per second (FPS) for state as vector, image and combined (vector and image).

Bots	Vector FPS	Image FPS	Combined FPS
No	1065	798	747
Yes	302	268	268

5 Experiments

The experiments were performed using 3 algorithms: PPO, TD3, and Rainbow. PPO and TD3 operate in a continuous action space, which is preferable for transferring an agent to the real world. Rainbow works in discrete action space, as shown in the Table 2. The source code for the environment and algorithms is available via the following link[1].

5.1 Results for Tracks Without Bots and States Represented as Vector

The results of experiments for tracks without bots and vectors as state-space are shown in Fig. 2. The following set of vector features was used for the experiment: `hull_position`, `hull_angle car_speed`, `checkpoint_sensor`, `finish_sensor`.

As can be seen from the charts on all types of track, agents based on the PPO method show a stable but slow convergence. We suppose that such happens due to the lack of replay memory. Because there are control points that are difficult to

[1] Source code: github.com/MartinsonMichael/CarRacing_agents.

Table 2. Action discretization for Rainbow

Action	Steer st	Gas g	Break b	Description
A_1	0.0	0.0	0.0	Noop action
A_2	-0.6	0.0	0.0	Left steer
A_3	0.6	0.0	0.0	Right steer
A_4	0.0	0.9	0.0	Gas
A_5	0.0	0.0	1.0	Break

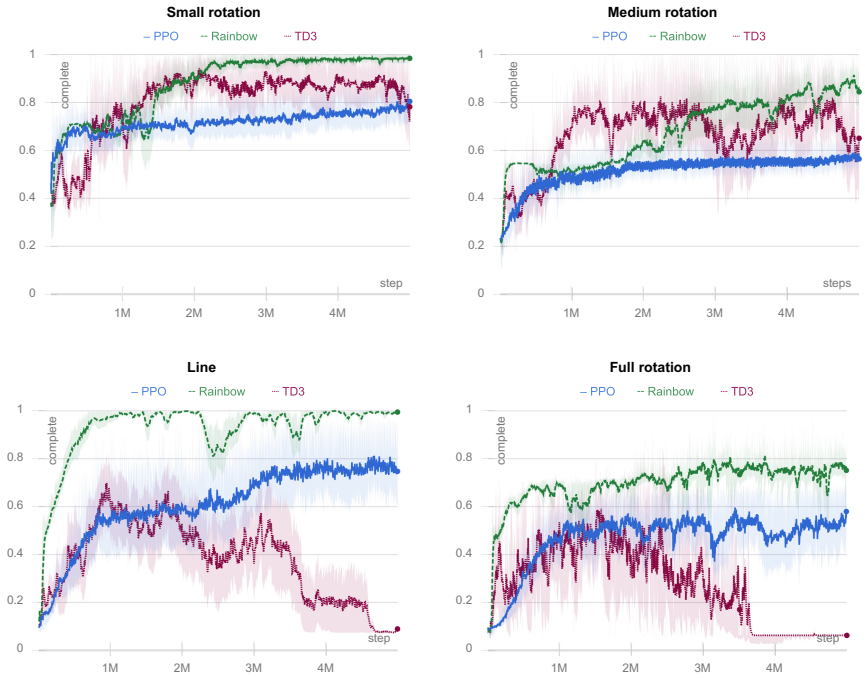

Fig. 2. The results of the experiments on the tracks without bots and vector observations. The bold line shows the average for 10 runs of each algorithm, with a smoothing of 0.6. The transparent line shows the standard deviation. For each of the algorithms, a preliminary search was performed for the best hyperparameters.

reach, such as around a bend. And in one update, PPO uses too little examples with positive point-reaching, so it takes longer to converge.

For tracks Line and Full Rotation, Rainbow also learns significantly faster than the other methods, most likely due to the design of the tracks themselves. Since they contain long straight sections for which many groups of state-space points have the same optimal policy.

5.2 Results for Tracks Without Bots and States Represented as Image

The next series of experiments was conducted using the image as the main feature. The architecture proposed in this paper [14] was used for image processing. The image is resized to 84×84 pixels, as the compression method used bilinear interpolation. The results shown in Fig. 3

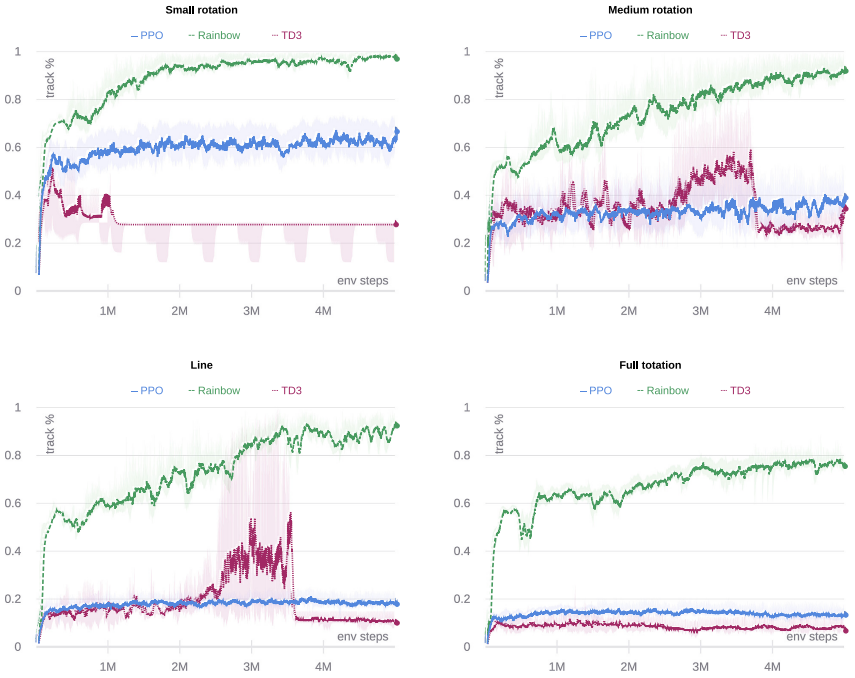

Fig. 3. The results of the experiments on the tracks without bots and image observations. The bold line shows the average for 10 runs of each algorithm, with a smoothing of 0.6. The transparent line shows the standard deviation.

As it can be observed from charts, it is more difficult for all methods to control the car using a pure image rather than a vector. We also faced the difficulty of configuring TD3 hyperparameters, as you can see in the first and last charts, this method gets stuck in suboptimal policies, namely, it starts to stand still or rotate on start point.

5.3 Influence of Different Sets of Vectors Features on the Convergence of the Algorithms

At this point, we investigated the effect of a set of vector features on the convergence of the algorithm. The key feature sets are shown in the Table 3.

As can be seen from the table, the results of the algorithms can differ greatly for the same sets of features. If on a Full rotation track PPO using only `hull_position` and `wheels_position` can only reach 20%, then Rainbow on both features reaches 60%, which corresponds to the end of the straight section before the turn itself. Adding an angle to both coordinate features – `hull_angle` greatly increases the performance (+17% for PPO and +30% for Rainbow). The combination of both coordinate attributes with an angle gives the best results for Rainbow and PPO.

You can also see that for PPO and Rainbow, the track sensor – `track_sensor` greatly increases the track % and the finish %. We believe that the sensor allows the agent to determine the closeness to the border of the route in advance. In contrast to the case when the agent distinguishes traveling outside the boundaries only by reward.

5.4 Results for Tracks with Bots

In this section, the environment state was represented as a combination of vector and image. Image processing was performed in a same way as in Sect. 5.2. To join image and vector we concatenate them along the channel dimension (copy of duplicated vector was used $h \times w$ times). So from image with shape $h \times w \times c$, and vector with shape v, we make observation matrix with shape $h \times w \times (c+v)$ shape.

The experimental results for the state of the environment consisting of the image and `hull_position`, `hull_angle` presented in Fig. 4. The best result was shown by the Rainbow algorithm, which learned to pass the intersection completely and without collisions. The PPO algorithm also converges, but much more slowly.

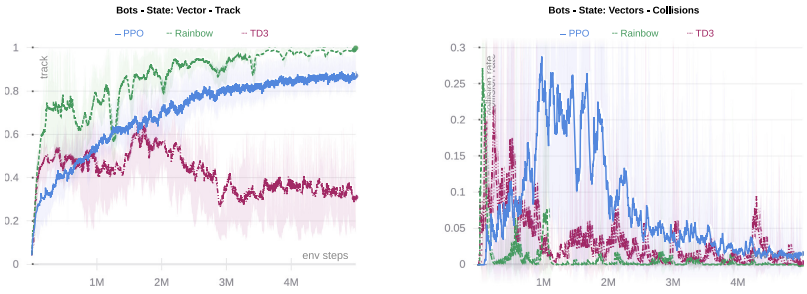

Fig. 4. Results of experiments on tracks with bots and vectors as the state of the environment. The percentage of the completed track, and the average number of collisions.

The experimental results for the state of the environment consisting of the image and `hull_position`, `hull_angle`, `car_radar_2`, `collide_sensor` presented in Fig. 5. In this case, TD3 and PPO algorithms learned to pass only a small part of the road.

Table 3. Key feature sets in increasing order by % track. Notation: T - `time` - time, V - `car_speed`, C_s - `checkpoint_sensor`, XY_w - `wheels_position`, T_s - `track_sensor`, F_s - `finish_sensor`, xy_h - `hull_position`, α - `hull_angle`; % track - the average percentage of the track passed by the agent by the end of training; % finish - the average number of episodes ended at finish point.

C_s	T_s	F_s	V	Img	T	α	XY_w	XY_h	% track	% finish
PPO on line track										
			+			+		+	0.074	0.000
				+					0.166	0.000
								+	0.228	0.000
+		+	+			+		+	0.675	0.578
+						+		+	0.949	0.900
					+	+		+	0.957	0.750
PPO on full rotation track										
				+					0.122	0.000
							+		0.188	0.000
						+		+	0.375	0.000
+		+	+			+		+	0.497	0.000
	+					+		+	0.524	0.000
						+	+		0.567	0.000
					+	+		+	0.663	0.000
						+	+	+	0.672	0.000
Rainbow on line track										
	+					+		+	0.916	0.950
				+					0.921	0.639
						+	+	+	0.942	0.750
Rainbow on full rotation track										
								+	0.600	0.000
							+		0.616	0.000
			+			+		+	0.712	0.000
+		+	+			+		+	0.770	0.115
						+	+		0.936	0.250
	+					+		+	0.936	0.550
						+	+	+	0.941	0.800
						+		+	0.947	0.300
TD3 on line track										
				+					0.117	0.000
+		+	+			+		+	0.217	0.083
+						+		+	0.939	0.950
TD3 on full rotation track										
+		+	+			+		+	0.063	0.000
				+					0.098	0.000
C_s	T_s	F_s	V	Img	T	α	XY_w	XY_h	% track	% finish

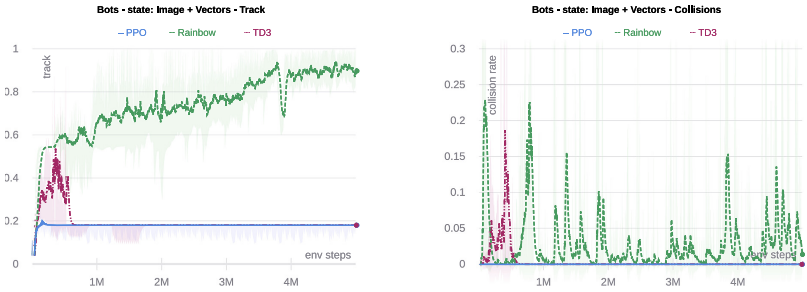

Fig. 5. Results of experiments on trajectories with bots and images as the state of the environment.

6 Conclusion

In this paper, we have described a developed environment, which allows to simulate driving through an intersection with realistic dynamics and the ability to train various reinforcement learning algorithms.

We have developed and described an effective learning method based on off-policy method Rainbow. We presented the results of a large series of experiments comparing our approach with other implementations that use a different combination of basic features used in describing the state.

Future plans of our research include improvements to the environment for simulating more complex intersections (prioritized roads, traffic lights, etc.). We also plan to integrate computer vision and reinforcement learning methods more closely to simulate the real environment in a more complete way.

Acknowledgements. The reported study was supported by RFBR, research Project No. 17-29-07079.

References

1. An, H., Jung, J.I.: Decision-making system for lane change using deep reinforcement learning in connected and automated driving. Electronics **8**, 543 (2019)
2. Bojarski, M., et al.: End to end learning for self-driving cars. arXiv preprint arXiv:1604.07316 (2016)
3. Brockman, G., et al.: Openai gym (2016). http://arxiv.org/abs/1606.01540, cite arxiv:1606.01540
4. Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., Koltun, V.: CARLA: an open urban driving simulator. In: Proceedings of the 1st Annual Conference on Robot Learning, pp. 1–16 (2017)
5. Engstrom, L., et al.: Implementation matters in deep RL: A case study on PPO and TRPO. In: International Conference on Learning Representations (2019)
6. Fujimoto, S., van Hoof, H., Meger, D.: Addressing function approximation error in actor-critic methods. In: Proceedings of Machine Learning Research, vol. 80, pp. 1587–1596 (2018)

7. Hessel, M., et al.: Rainbow: combining improvements in deep reinforcement learning. In: Thirty-Second AAAI Conference on Artificial Intelligence (2018)
8. Kiran, B.R., et al.: Deep reinforcement learning for autonomous driving: a survey. arXiv preprint arXiv:2002.00444 (2020)
9. Li, D., Zhao, D., Zhang, Q., Chen, Y.: Reinforcement learning and deep learning based lateral control for autonomous driving [application notes]. IEEE Comput. Intell. Mag. **14**(2), 83–98 (2019)
10. Li, H., Wang, A.Y., Liu, Y., Tang, D., Lei, Z., Li, W.: An augmented transformer architecture for natural language generation tasks. arXiv preprint arXiv:1910.13634 (2019)
11. Lillicrap, T.P., et al.: Continuous control with deep reinforcement learning. CoRR abs/1509.02971 (2015)
12. Loiacono, D., Cardamone, L., Lanzi, P.L.: Simulated car racing championship: Competition software manual. CoRR abs/1304.1672 (2013). http://arxiv.org/abs/1304.1672
13. Lopez, P.A., et al.: Microscopic traffic simulation using sumo. In: The 21st IEEE International Conference on Intelligent Transportation Systems. IEEE (2018). https://elib.dlr.de/124092/
14. Mnih, V., et al.: Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602 (2013)
15. Mukadam, M., Cosgun, A., Nakhaei, A., Fujimura, K.: Tactical decision making for lane changing with deep reinforcement learning, December 2017
16. Oh, S.Y., Lee, J.H., Doo Hyun, C.: A new reinforcement learning vehicle control architecture for vision-based road following. IEEE Trans. Veh. Technol. **49**, 997–1005 (2000)
17. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms. CoRR abs/1707.06347 (2017). http://arxiv.org/abs/1707.06347
18. Shikunov, M., Panov, A.I.: Hierarchical reinforcement learning approach for the road intersection task. In: Samsonovich, A.V. (ed.) BICA 2019. AISC, vol. 948, pp. 495–506. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-25719-4_64
19. Sutton, R.S., Barto, A.G., et al.: Introduction to Reinforcement Learning, vol. 135. MIT press Cambridge, Cambridge (1998)
20. Sutton, R.S., McAllester, D.A., Singh, S.P., Mansour, Y.: Policy gradient methods for reinforcement learning with function approximation. In: Solla, S.A., Leen, T.K., Müller, K. (eds.) Advances in Neural Information Processing Systems, vol. 12, pp. 1057–1063. MIT Press (2000). http://papers.nips.cc/paper/1713-policy-gradient-methods-for-reinforcement-learning-with-function-approximation.pdf
21. Wang, G., Hu, J., Li, Z., Li, L.: Cooperative lane changing via deep reinforcement learning. arXiv preprint arXiv:1906.08662 (2019)
22. Wang, P., Chan, C., de La Fortelle, A.: A reinforcement learning based approach for automated lane change maneuvers. CoRR abs/1804.07871 (2018). http://arxiv.org/abs/1804.07871
23. Yudin, D.A., Skrynnik, A., Krishtopik, A., Belkin, I., Panov, A.I.: Object detection with deep neural networks for reinforcement learning in the task of autonomous vehicles Path Planning at the Intersection. Opt. Memory Neural Netw. **28**(4), 283–295 (2019). https://doi.org/10.3103/S1060992X19040118