Periodica Polytechnica Transportation Engineering, 49(3), pp. 292–300, 2021

Autonomous Drifting Using Reinforcement Learning

László Orgován[1*], Tamás Bécsi[1], Szilárd Aradi[1]

[1] Department of Control for Transportation and Vehicle Systems, Faculty of Transportation Engineering and Vehicle Engineering, Budapest University of Technology and Economics, H-1111 Budapest, Műegyetem rkp. 3., Hungary
[*] Corresponding author, e-mail: orgovanlaszlo@edu.bme.hu

Received: 17 May 2021, Accepted: 15 June 2021, Published online: 19 August 2021

Abstract

Autonomous vehicles or self-driving cars are prevalent nowadays, many vehicle manufacturers, and other tech companies are trying to develop autonomous vehicles. One major goal of the self-driving algorithms is to perform manoeuvres safely, even when some anomaly arises. To solve these kinds of complex issues, Artificial Intelligence and Machine Learning methods are used. One of these motion planning problems is when the tires lose their grip on the road, an autonomous vehicle should handle this situation. Thus the paper provides an Autonomous Drifting algorithm using Reinforcement Learning. The algorithm is based on a model-free learning algorithm, Twin Delayed Deep Deterministic Policy Gradients (TD3). The model is trained on six different tracks in a simulator, which is developed specifically for autonomous driving systems; namely CARLA.

Keywords

machine learning, reinforcement learning, autonomous driving, drifting

1 Introduction

Drifting is a high side-slip cornering, just like in rally racing when with an RWD (Rear Wheel Drive) car, the driver is counter-steering with a large side-slip angle at relatively high speed. For this manoeuvre, the vehicle needs a large rear drive-torque, which can cause significant rear wheel-spin. This is a challenging manoeuvre even for an experienced driver and for an autonomous vehicle as well. To solve this problem, the idea was to start with a trajectory tracking algorithm. First, the agent should follow a predefined path accurately, then it can learn to drift through the corners with further training.

In (Cai et al. 2020), the authors created a drifting algorithm in a simulator (CARLA). CARLA is an open-source simulator for autonomous driving using Unreal Engine. The authors published these maps and the trajectories, which were useful for developing our algorithm because to create a map in CARLA, a third-party application is needed, like RoadRunner. Still, it requires a license to use it.

Using Deep Reinforcement Learning (DRL), which combines classic reinforcement learning with deep neural networks, is a good solution for these kinds of motion planning problems. It is possible to develop "end-to-end" solutions, where the network inputs are the travel destination, vehicle's parameters, e.g. velocity, acceleration, and the outputs are the direct vehicle control commands, like steering, torque, brake (Aradi, 2020). In this project, different algorithms are used for this continuous control problem (DDPG, TD3, SAC), but the one with the best performance was the TD3 (Twin Delayed Deep Deterministic Policy Gradients). It was the best for trajectory-tracking and for drifting as well.

1.1 Motion planning

Motion planning for autonomous driving is essential and there is much research on this topic. There are different kind of approaches, like machine learning, classic control methods, or different optimization techniques. Based on the survey from (Aradi, 2020), at the end of the last decade (2019), the number of the research papers increased significantly, in the case of DRL topic, there are almost 5 times more than in 2010. The motion planning is a complex problem and that is where DRL comes into the picture. The drifting is approached from a trajectory tracking point of view. There are different solutions for this problem, the controller method or the (continuous) DLR, when the agent chooses the action, which is the throttle or steering actuation (or both).

1.2 Drifting

In Fig. 1, there is a typical cornering on the left, and the drift is on the right. The β is the side-slip angle, which is

Cite this article as: Orgován, L., Bécsi, T., Aradi, S. (2021) "Autonomous Drifting Using Reinforcement Learning", Periodica Polytechnica Transportation Engineering, 49(3), pp. 292–300. https://doi.org/10.3311/PPtr.18581

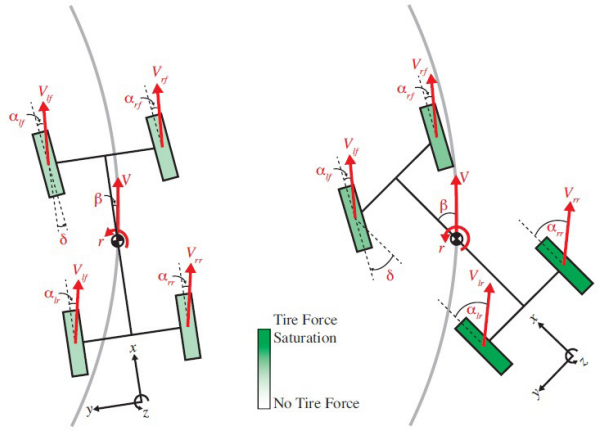

Fig. 1 Typical cornering (left) and drift (right) (Hindiyeh, 2013)

the angle between the direction of the heading (longitudinal axis of the vehicle) and the direction of the velocity vector at the centre of gravity (Cai et al. 2020), which is much higher in case of drifting.

1.2.1 Reinforcement learning-based methods

Based on the (Cai et al. 2020) research, this was the first to achieve transient drift with DRL. On the other hand, they only used Front-Wheel Drive (FWD) vehicles like an Audi A2 or TT, and one AllWheel Drive (AWD) vehicle, the Cola Truck. These vehicles are not suitable for drifting because of the drivetrain. In this case, it is not drifting, it is just a high-speed cornering, and the whole car is sliding, not only the rear part. So the goal was to use these maps and DRL with an RWD car and achieve drifting.

2 Reinforcement learning

Machine Learning (ML) is one section of Artificial Intelligence (AI). There are three ML approaches, Supervised learning, Unsupervised learning and Reinforcement learning (RL). RL's two most important features are the trial-and-error search and delayed reward (Sutton and Barto, 2018). The trial-and-error search means that the learning agent takes action, and for this action gets a reward, and it tries to maximize this reward. This is how the agent discovers the environment. There are some complicated tasks when the agent doesn't get the reward at every action, just after a series of actions (delayed reward), e.g. an autonomous vehicle gets a reward when reaches the goal safely.

Reinforcement Learning is complex, but it can be divided into two classes, model-based and model-free methods. The model of the environment tries to predict the next state and reward, based on the current state and action, so it tries to mimic the behaviour of the environment (Sutton and Barto, 2018). The original idea was to start from an unknown model (irreversible environment) and learn the dynamics model based on the observed data. This model, by definition, is reversible, therefore, can be used for planning. On the other hand, the model-free methods are the opposite of planning (explicitly trial-and-error learners). For this task, only the DDPG and TD3 are used, which are model-free methods.

2.1 Deep Deterministic Policy Gradient (DDPG)

DDPG was presented in (Lillicrap et al. 2019) and became popular amongst the RL community quite fast. DDPG is an algorithm that concurrently learns a Q-function and a policy, uses off-policy data and the Bellman equation to learn the Q-function, and the Q-function to learn the policy (Achiam, 2018). It is important to note that this algorithm is only for those tasks when the action space is continuous, like in this case. The experience replay buffer is essential for this kind of algorithm, basically a large set of previous experiences (set D), but to choose the perfect size, which is neither too big, nor too small, is necessary. This is a hyperparameter, which needs fine-tuning based on the specific environment.

2.1.1 Architecture

The DDPG has two pairs of neural networks, the Q-network and the target Q network, the deterministic policy network and the target policy network. The target network is the copy of the main network with a time-delay. The architecture is based on the Actor-Critic method. The Actor uses the policy-based approach and tries to calculate the optimal policy deterministically, trying to pick the best action. On the opposite side, the Critic is a value-based approach. The Critic calculates the optimal action-value function based on the Actor's action.

From the replay buffer, we sample random mini-batches when the Actor-Critic networks are updated. To update those networks, similarly like in the Q-learning, the target ones are used. The goal is to minimize this mean-squared loss between the two Q values (original-updated). For the update of the target networks, a so-called "soft updates" is used. Another important note, for this kind of continuous action spaces, additional noises to the action are necessary (to maintain the exploration).

This is a brief summary of the DDPG, unfortunately, it was not the best solution for this drifting problem, but the TD3 algorithm is based on this, thus it was essential to summarize this method.

2.2 Twin Delayed Deep Deterministic policy gradients (TD3)

The common problem with the DDPG is the Q-values overestimation, this is why the TD3 was developed. To solve this problem, three major improvements were introduced (Achiam, 2019):

- Instead of one, TD3 learns two Q-functions and uses the smaller Q-value (of the two) to form the targets in the Bellman error loss functions.
- The policy and target networks are updated less frequently than the Q function.
- Noise to the target action, this way it is harder for the policy to exploit the Q-function errors.

2.2.1 Network parameters

The TD3 algorithm is created in PyTorch, based on Lillicrap et al. (2019), the author created a tutorial for DRL algorithms with OpenAI gym environment. The drifting environment is different, but the base is similar, so it was a good starting point.

The network parameters are in Table 1, where X is the state dimension, and Y is the action dimension.

Finally, one can see the important hyperparameters. The replay buffer size was 100.000, and the batch size is 256. For the discount factor (γ) 0.99 is used, 0.005 for target smoothing, and the optimizer is the default PyTorch's ADAM optimizer, where the default learning rate is 0.001, and the network is updated at every two steps. Different combinations of these were tested, but the best result was achieved by the above-mentioned parameters. The results can be seen in the next sections.

3 Model structure

3.1 Simulator

The simulator is the CARLA 0.9.9, which was the latest version when the project was started. CARLA has its own maps, but they are for urban driving, there are cities, which are great for traffic-related simulation with lots of vehicles and pedestrian, but for drifting it is useless, and graphically demanding because of the number of the objects (houses, cars, roads etc.). The used maps only contain the road itself some road fence/barriers and trees, and because of the less number of objects, the computer, which is used for the development, could handle very well, besides that, it is not the most up-to-date, top of the line PC. The specification of the PC is the following: an Intel i5-7500 CPU, 16 GB of RAM and a GeForce GTX 970 video card.

Table 1 Network parameters

Style name:	Actor network	Critic network
Input layer FC1	X nodes in, 256 nodes out	X + Y nodes in, 256 nodes out
Hidden Layer FC2	256 in, 256 out	256 in, 256 out
Hidden Layer FC3	256 in, 128 out	256 in, 128 out
Hidden Layer FC4	128 in, 32 out	128 in, 32 out
Output layer FC5	32 in, Y out	32 in, 1 out

3.1.1 The client

The Client and the World are two of CARLA fundamentals, a necessary abstraction to operate the simulation and its actors (CARLA, 2020a). The client is connected to the server/world, and it can send commands and receive information. Using the PythonAPI, we can control the simulation via python scripts.

The communication between the server and the client is crucial; the simulation is based on that. That is why CARLA has two modes, asynchronous and synchronous. By default, the communication between the client and the server is asynchronous, thus the server runs the simulation as fast as possible, without waiting for the client. On the other hand, in synchronous mode, the server waits for a client tick (which is basically a done/ready message) before updating to the following simulation step (CARLA, 2020b). Suppose the goal is to control the vehicle. In that case, the synchronous mode is essential because if the control is the steering, it must be sent at given time-steps, e.g. at every 0.1 seconds. Still, without the synchronous mode, at this given time, the vehicle can travel, e.g. 1 meter or 2, depending on the speed of the simulation. Thus, the other important setting is the fixed time-step.

The simulation time and the real-time is fortunately different, this means the simulation can run faster than real-time. The simulated World has its own clock and time, conducted by the server, and the server can take a few milliseconds to compute two steps of a simulation. However, the time-step between those two simulation moments can be configured to be, for instance, always a second (CARLA, 2020b). This time-step can be fixed or variable. The default is the variable time-step, when the simulation time between the steps depends on the time of the server's computation. To make sure that the time between the steps is constant, the fixed time-step mode should be used, and in this case, if the constant value is 0.05 seconds, there will be 20 frames per simulated second. Because of the fixed time, this is the best way to collect data. These are the settings for training, and this way, it is much faster than real-time.

3.1.2 The World

The World is the major part of the simulation, this class contains the general settings and most of the information (CARLA, 2020b), such as the actors, the blueprint library, weather and lightning settings, simulations settings, the map, the snapshots. Another important thing to note, there is only one World per simulation, but the World can be changed anytime.

To change the world settings, there are functions which can be used to turn on the above-mentioned synchronous mode, define the tick function, spawn an actor (e.g. car), change the weather. The weather can be changed at will, e.g. how much cloud should be on the sky (0-100, zero means clear sky, and 100 means completely cloudy), similarly with the fog, the fog distance can be set or the rain or even the altitude angle of the Sun. So, there are many options, basically it is possible to simulate any kind of situations. Another useful setting is the no rendering mode, which can be enabled, so the simulation is a "black screen", the GPU does not render at all, thus it is much faster, it can save a lot of time in a long training session.

3.1.3 The map

A map includes both the 3D model of a town and its road definition, and every map is based on an OpenDRIVE file describing the road layout fully annotated (CARLA, 2020c). To change the map in the simulation, the World must be changed too (load to change the map or reload the World if the goal is to use the same map). The default maps can be modified with Unreal Editor, or new ones can be created using RoadRunner. The map has many more features like the traffic signs, lanes, junctions. However, in this project, these are irrelevant; drifting should be done only on tracks, not in traffic.

3.2 Tracks

There are seven maps (Fig. 2), which are exported from RoadRunner (OpenDRIVE standards). For training the first 6 maps (Fig. 2 (a)–(f)) are used, and the 7th is for testing. The tracks become more and more difficult, they have sharper corners, and they are longer as well e.g. the first track is 1.22 km long, the fifth one is 3.02 km, so it is more than double.

3.3 Environment

A classic RL environment must contain two fundamental functions, the step and reset functions. This drifting environment is more complicated and needs more additional functions. The first thing to do was to define the CARLA environment. To do that, establishment of the connection between the client and the server is the first step, followed by the world creation i.e. loading the given map, defining the settings, e.g. turning on the no rendering mode, the synchronous mode, and defining the time between the steps, which is 0.05 seconds. So, the client-server connection is established, and the world settings are correct. The next one is the vehicle itself.

The vehicle is an actor, and the Blueprint library class contains all of the actors (sensors, vehicles, pedestrians etc.). Almost every vehicle was tested, which is suitable for drifting (Mustang, Tesla Cybertruck, Tesla Model 3, Dodge Charger); however, the Tesla Model 3 was the best performing vehicle.

Using the ID of the vehicle, the spawn of the chosen vehicle can be done, only one more parameter is needed, which is the spawning point. The spawning point is defined by the CARLA's Transformation class, which contains the location and the rotation, so it needs six variables (x-y-z coordinates, pitch, yaw, roll). The route of the trajectory defines the starting point's coordinates, basically the first coordinates of this route. These trajectories contain the x-y coordinates, and they are stored in a file, but the distance between the points is around 0.1–0.2 meters, which is a small value. A new trajectory was created for every map, where the distance between these points is 4–10 meters (these values came from experience, the best value was 5 meters).

In Fig. 3, there is a difference between the original trajectory and the new one (this is the first map, (a) in Fig. 2). For a better illustration, the distance between the points is 10 meters, but the 5 meters setup was the best.

The next step is to control the car, CARLA has its own Vehicle Control class, which is managing the basic movement of a vehicle using typical driving controls (throttle, steer, brake) (CARLA, 2020d) using the PythonAPI to control the vehicle via a python script. The first idea was to control the speed (throttle, brake) and the steering, but the vehicle is decelerating if the throttle is not 1 (full throttle),

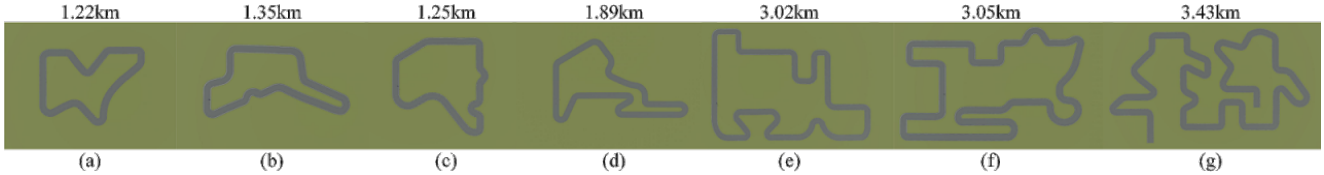

Fig. 2 (a)–(f) Maps for training; (g) and testing (Cai et al. 2020)

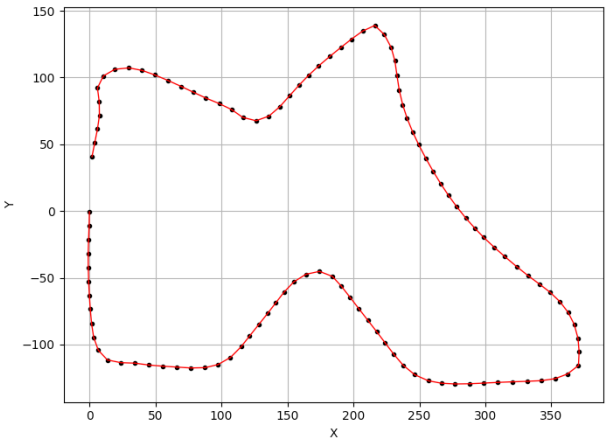

Fig. 3 Original (red) and the created (black) trajectory points

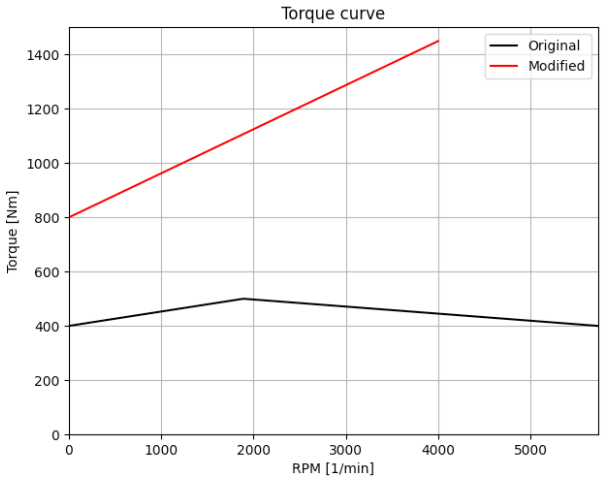

Fig. 4 Original (black) and the modified (red) torque curve

so to use the brake and the hand brake variable is unnecessary. Thus the agent's actions are the throttle and steering.

3.4 Vehicle model

CARLA's vehicle model is based on the Nvidia PhysX. To control the model's parameters/physics, the *VehiclePhysicsControl* and *WheelPhysicsControl* classes are used. With the first one, the vehicle's engine performance (engine's torque, maximum RPM, the transmission's properties), the vehicle's mass, and some coefficient/ratios (drag coefficient, damping ratio) can be defined. The second one defines the tire friction value, the maximum steering angle of the wheel, the radius of the wheel and the braking torque. To find the proper values for these variables was crucial for proper training (not all, only some of them has to be modified). Unfortunately, the information about the tire's longitudinal and lateral slip angles are not available, and these are missing from CARLA. Maybe the future version will have these kinds of information.

The first problem with the cars was the acceleration. They were not fast enough, especially at lower speed e.g. after a corner, even at full throttle, the acceleration was not good enough, so the engine's power had to be increased. The original and the modified torque curve can be seen in Fig. 4. The modified value came from experience, the engine's RPM had to be limited as well; this way the desired constant acceleration can be achieved at lower and higher speed as well.

4 Drifting

The starting point was the trajectory tracking algorithm. More details about it later.

For the drifting, the first thing was to introduce the side-slip variable. This value needs to be calculated at every step because CARLA doesn't have a function for it.

$$\beta = \arctan\left(\frac{v_{local\_y}}{v_{local\_x}}\right) \tag{1}$$

The tire friction default value at all four wheels is 3.5, so to help to the car to slide, this value was decreased at the rear wheels, but the coefficient at the front wheels were decreased as well, the front the tire friction's value is 2.8 and that of the rear is 2.0. These value combinations came from experience, almost every reasonable combination was tested. Besides these, the modification of the vehicle's weight was tested, as well as the different combinations, with all of these values, heavier car, lighter/smaller car, with different kind of engine power, and tire friction coefficients. There is an almost infinite combination of these, thus it is hard to find the best combination when the vehicle has enough power and the ability to control at side-slipping.

The map is changed after every five episodes because it will reduce the time of the training. Loading a new map takes few seconds, and there should be a one or two seconds waiting time after the map loading because if there is no waiting time, the simulator crashes randomly from time to time, but with these few seconds, the problem is solved. However, waiting for every episode even just two seconds takes a lot of time, considering thousands of episodes, not to mention that the agent did thousands of steps in one episode. So, in total, it takes a lot of time, that is why the map is changed only after every five episodes. However, in theory, it would be better if there were a new map in every episode. But if the agent reaches the goal, the next map is different from the current one.

4.1 State

In the case of trajectory tracking, the state contains the vehicle's parameters (velocity, acceleration), the distances

between the next 8 trajectory points and the angle differences (between the vehicle and the given point).

In the case of drifting, the state is extended with the side-slip value and the state dimension is reduced to 23 (from 30), now the state has the vehicle's principles such as the velocity and acceleration component (v_x, v_y, a_x, a_y), like in trajectory tracking and the side-slip angle (β), the distance from the closest point (ε_d) and the angle between the heading and the point (ε_a), and the rest is the information about the next 8 points. In this case, only the distances $\left(d_{x_n}, d_{y_n}\right)$ are used, where n is the number of the given points), not the angles, because they are irrelevant in the case of drifting. The distances are enough for the basic path following.

$$\text{state} = \varepsilon_d, \varepsilon_\alpha, \beta, v_x, v_y, a_x, a_y, d_{x_1}, d_{y_1}, \ldots d_{x_8}, d_{y_8} \tag{2}$$

4.2 Reward system

The partial reward system of the trajectory tracking can be seen in Fig. 5. It is based on the distance between the vehicle's centre point and the closest trajectory point, the heading angle error, and the velocity. If the vehicle's speed is higher, then the reward is higher as well, on the other hand, if the distance and the angle difference is smaller, then the reward is higher. The final partial reward value is the average of these three.

The agent gets a large positive reward, when it reaches the goal, and a large negative reward, when it collides, slows down, turns over or goes farther from the trajectory point. With this reward system the path following was successful, in more than 60 % of the cases the agent reaches the goal, this is considered as a proper base for drifting.

In the case of drifting, the first idea was to extend the state with the side-slip angle, and update the reward

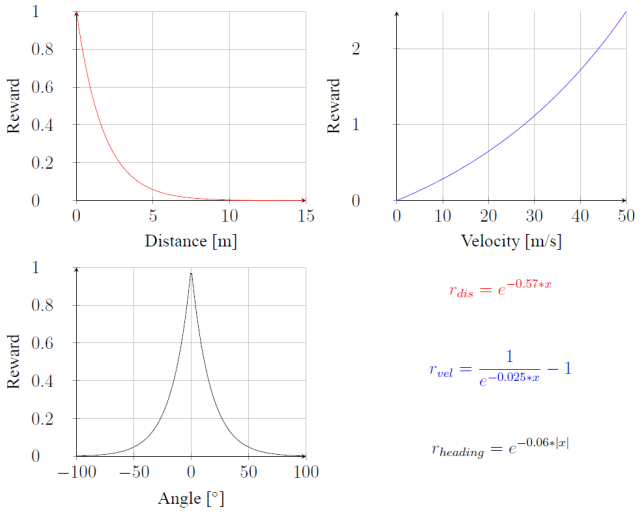

Fig. 5 Distributions of the rewards

system with the r_{slip} reward, similarly to the velocity, a higher value means higher reward, and everything else remained the same as it was in trajectory tracking.

As expected, this did not work, more modification was needed, the car acceleration was not good enough, so at this point, the previously mentioned torque curve modification (Fig. 4) was introduced. Other vehicle-based modifications were needed as well, e.g. the maximum steering angle. By default, in the case of the Tesla Model 3, the maximum steering angle of the wheels is 69.9°, this value was increased to 79.9°.

The partial reward system needed a complete re-design. The idea came from a video game, namely Need For Speed Underground 2. To get the biggest drift reward, a long drifting session must be performed, and the reward (points) comes after the drift is finished. In the game, when the drift is performed through multiple corners, the reward is larger and larger, so the idea was similar in this project as well.

The first step was to define what is drifting, the definition is the following: if the side-slip angle is bigger than a specific value and the velocity is higher than a given threshold, then the given step considered as a drift step. In the first place, the side-slip angle limit was 10°, and the velocity limit was 10 m/s, then that step was a drift. The next question was how many steps count as a drift? This value was 15 at first, so if the vehicle takes 15 steps when the side-slip angle and the velocity are at least 10, then it is a drift. And the reward, in this case, is the following:

$$r_{temp} = drift_{length} * \left(r_v + r_{dis} + r_{slip}\right), \tag{3}$$

where the reward is based on the distance and the velocity is calculated as before, which can be seen in Fig. 5. The reward, based on the slip angle, is one, if the slip angle is around 70.

$$r_{slip} = \frac{1}{e^{-0.01*|x|}} - 1 \tag{4}$$

This is the temporary/partial reward, and when one of the conditions is not fulfilled then the drift is over, the agent gets the cumulative reward, during the drifting the reward is zero. So at the beginning of the step, the slip angle is checked, if it is larger than the given threshold, an array is extended with it, for the sake of simplicity let's call it *drift_array*, and if it is below the threshold, then this array is empty. After the state and reward calculation, this array equals the *prev_drift_array*, in this case they can be compared at the next step. If the current array is empty, and the previous one is not, then at this step the drift is over. For example, if the vehicle does 50 steps of drifting,

from the second step the temporary reward adds up, and at the 51st step, the agent gets the whole reward:

$$r_{ep_{51}} = 1 * R_2 + 2 * R_3 + \ldots + 50 * R_{51}, \quad (5)$$

where R_x is the reward based on the previous action, e.g. R_2 is the reward based on the first action, and the multiplier is the length of the drift (the first action is always full throttle, and 0 steering angle).

If there is no drifting at all, then the reward is the same as in trajectory tracking:

$$r = \left(r_v + r_{dis} + r_{heading}\right) / 3. \quad (6)$$

There were some problems with this reward system, as the agent tried to drift all the time, even at a straight line, it just slided one side to the other (Fig. 6). So, the rear is sliding, and the agent does this behaviour to maximize the reward. In this way, it is considered as a drift, when the vehicle slides to one side, and as one more drift when it slides to the other, and so on. This must be eliminated because it is clearly not the way it supposed to work, the agent should drift only at the corners.

At this point, the drifting needed to be redefined, so the 10° side-slip angle was increased to 20°, and the steps were doubled, which are required for drifting (30), but the velocity limit was decreased from 10 m/s to 7 m/s. The goal was to reward only the longer drifting which should take place only in the corners. In this case, the agent should slide for a longer period of time, but due to the drifting it will slow down more, so that is why the velocity limit is smaller.

4.3 Results

Every training result was a little bit different, if the car drifts more, then more likely collides with the wall or turns over, if it drifts less, then it is more stable, it reaches the goal with a higher percentage. In this project, it is better if the car takes only 4–5 corners, but at high speed and it is really drifting, then it can be called drifting, even if it crashes sometimes. Obviously, the ultimate goal is when the car reaches the goal every episode and drifting every corner, but it is a challenging task.

The easiest way to demonstrate drifting is the video, to show how the agent performs at different corners, but it is impossible to present, so instead of this, 1000 episodes were performed on the test map (Fig. 2 (g)).

The average step was 847 (which is small compared to trajectory tracking, where it was around 3800), and the agent drifts 2.5 times in one episode on average. The average velocity was 61.6 km/h, and the maximum drifts in one episode were 19 (in almost every corner). Unfortunately, in most of the episodes, there were only a few drifts (1–3) and less than 1000 steps (3 corners), which is not impressive, but in some cases, the agent drifted really well. On the other hand, it is better if the agent drifts really well in some cases and crashes more than doing a mediocre job. For a better illustration, only those episodes matter, when the agent did at least five drifts, the result can be seen in Fig. 7. In this case, this is 106 episodes of 1000. The average step was 3070, which is 3.6 times higher than the average of the 1000 episodes, and the average velocity was almost identical: 61.75 km/h. To mention, the average reward, in this case, was 17145, which is 4.2 times higher than the average of the all (4082). The maximum step in case of one drift was 60.

In Fig. 8 there is one test episode, these are the first few corners. The black dashed line is the trajectory, which the agent should follow, the blue line is the position of the vehicle at

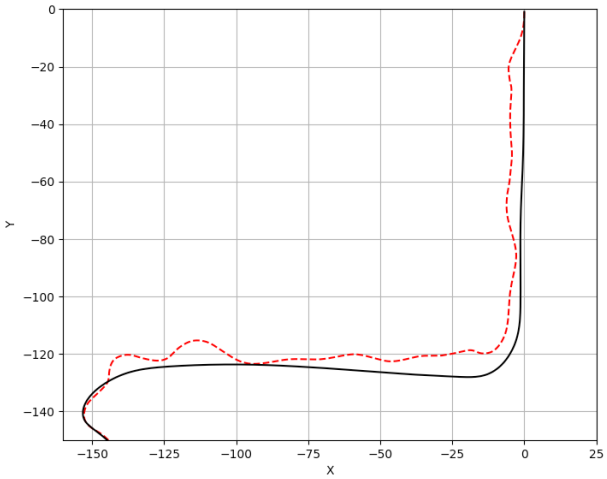

Fig. 6 Vehicle position (red) and the trajectory (black)

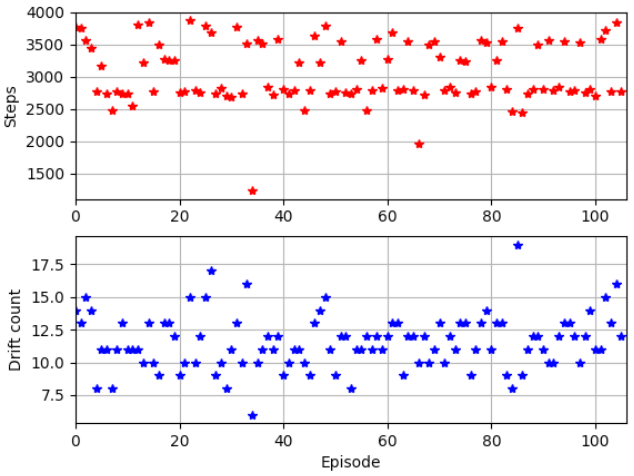

Fig. 7 Number of the steps and drifts when the drift counts higher than 5

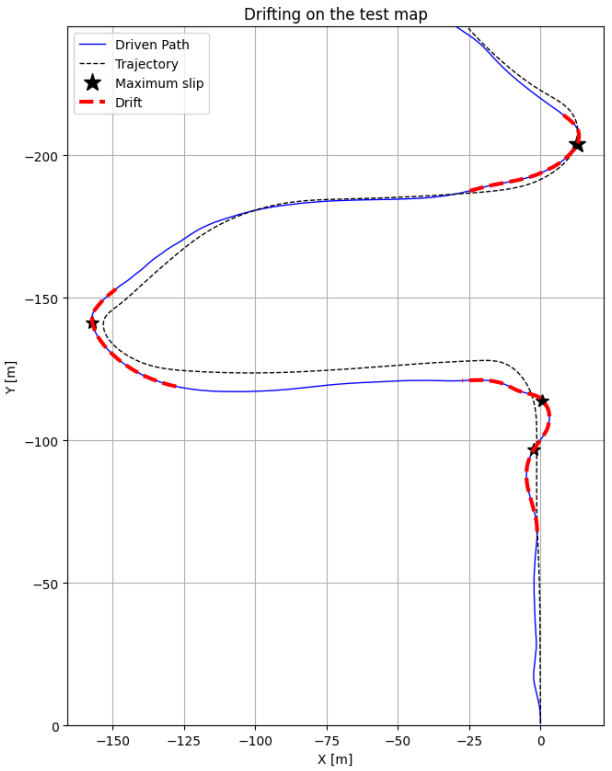

Fig. 8 Drifting on the test map

a given step. The red dashed line is drifting when the slip angle and the velocity is higher than the threshold.

The corresponding velocity value (blue line) and slip value (red line) can be seen in Fig. 9.

You can see at the first corner, that the slip values are really high, and the agent slides to the left first, then to the right, and finally, at the left corner, it slides to the left. These are the biggest spikes on the slip values. This is the limit, which the agent can control. Before the second left corner, the car keeps to the left while accelerating to almost 100 km/h and slides to the right. After this, it follows the path while accelerating in the same manner and takes a hard-left corner.

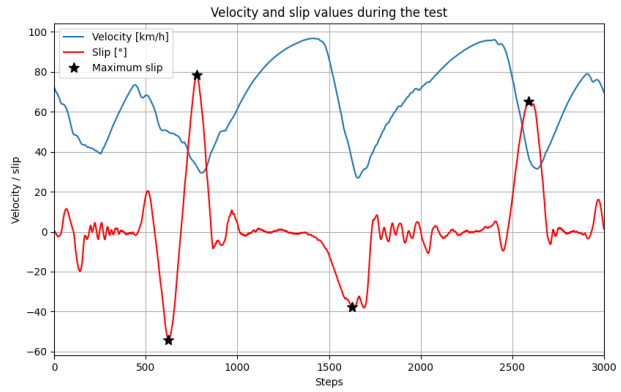

Fig. 9 Velocity and slip values during drifting

5 Conclusion

In this paper, an autonomous drifting algorithm was created. It is trained on six different maps, with various difficulties. The agent achieved a good drifting performance. However, there are plenty of possibilities to improve the drifting capability, to implement in a new CARLA version, the CARLA team improves the simulator regularly. The final goal is to create a general solution for drifting, an agent that can drift with different kinds of vehicles and maybe a real-life test. Finally, an illustration of the drifting in CARLA in Fig. 10, shows how a drift looks like step-by-step.

Acknowledgement

The project was supported by the EFOP-3.6.3-VEKOP-16-2017-00001: Talent Management in Autonomous Vehicle Control Technologies – The Project is supported by the Hungarian Government and co-financed by the European Social Fund.

The research was supported by the Ministry of Innovation and Technology NRDI Office within the framework of the Autonomous Systems National Laboratory Program.

Fig. 10 Drifting

References

Achiam, J. (2018) "Deep Deterministic Policy Gradient", [online] Available at: https://spinningup.openai.com/en/latest/algorithms/ddpg.html [Accessed: 08 March 2021]

Achiam, J. (2019) "Twin Delayed DDPG", [online] Available at: https://spinningup.openai.com/en/latest/algorithms/td3.html [Accessed: 08 March 2021]

Aradi, S. (2020) "Survey of Deep Reinforcement Learning for Motion Planning of Autonomous Vehicles", [cs.LG], arXiv:2001.11231, Cornell University, Ithaca, NY, USA. [online] Available at: https://arxiv.org/abs/2001.11231 [Accessed: 16 May 2021]

Cai, P., Mei, X., Tai, L., Sun, Y., Liu, M. (2020) "High-SpeedAutonomous Drifting With Deep Reinforcement Learning", IEEE Robotics and Automation Letters, 5(2), pp. 1247–1254. https://doi.org/10.1109/LRA.2020.2967299

CARLA (2020a) "1st. World and client", [online] Available at: https://carla.readthedocs.io/en/latest/core_world/ [Accessed: 08 March 2021]

CARLA (2020b) "Synchrony and time-step", [online] Available at: https://carla.readthedocs.io/en/latest/adv_synchrony_timestep/ [Accessed: 08 March 2021]

CARLA (2020c) "3rd. Maps and navigation", [online] Available at: https://carla.readthedocs.io/en/latest/core_map/ [Accessed: 08 March 2021]

CARLA (2020d) "Python API reference", [online] Available at: https://carla.readthedocs.io/en/latest/python_api/ [Accessed: 08 March 2021]

Hindiyeh, R. Y. (2013) "Dynamics and control of drifting in automobiles", PhD Thesis, Stanford University.

Lillicrap, P. T., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., Wierstra, D. (2019) "Continuous control with deep reinforcement learning", [cs.LG], arXiv:1509.02971, Cornell University, Ithaca, NY, USA. [online] Available at: https://arxiv.org/abs/1509.02971 [Accessed: 16 May 2021]

Sutton, S. R., Barto. G. A. (2018) "Reinforcement learning: An introduction", The MIT Press, Cambridge, MA, USA.