Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Computer Science

MicroRacer: Development of a didactic environment for Deep Reinforcement Learning

Supervisor:
Chiar.mo Prof.
Andrea Asperti

Author:
Marco Del Brutto

Session III
Academic Year 2020/2021

To my family.

Introduction

Reinforcement Learning (RL) is a branch of machine learning where an *agent* taking *actions* in a given *environment* is supposed to learn an optimal *behaviour* by acquiring experience. Due to its recent, numerous and remarkable improvements and the availability of faster processors, RL is now much more widespread than in the past. In addition to the traditional fields of robotics and autonomous driving, RL has been successfully applied in various and diverse sectors, such as electric power system optimization, Natural Language Processing (NLP), financial marketing, and many more.

One of the major improvements of the last ten years has been the adoption of Deep Neural Networks as function approximators to address scalability issues, leading to the creation of an entirely new branch, of the discipline, known as Deep Reinforcement Learning. Despite the continuous increase in the use of RL, it remains a mathematically complex field and an active field of research. Furthermore, the addition of neural networks has introduced ulterior difficulties, such as a longer training time, and issues in debugging and explainability, in particular relative to the comprehension of various problems and failures during the training process.

Considering what has just been stated, the didactic of deep reinforcement learning could be complex. More than once in the past, this problem has been tackled. As a result, different environments have been developed during the last years. However, in many cases these environments haven't really represented a solution, for different reasons. In some cases, they can be considered as either too simplistic or too complex. In other cases, even

if these environments are interesting or stimulating, the long training times may render every attempt to study or optimize Deep Reinforcement Learning algorithms problematic.

In this thesis, we worked at the development and testing of the Microracer DRL environment, a new environment for the didactic of DRL, originally created by Prof.Asperti. The environment takes inspiration from car racing: the goal consists in completing random generated tracks in the shortest time, just relying on lidar-like observations.

Different parts have been added to the original project given. An optimization work has been made since the lidar handling has been moved from Python to Cython in order to improve its execution time. Moreover, the time-step of the environment has been regulated accordingly to the new execution speed and the code has been reformatted to be easier to use.
To render the track more interesting, its creation has been enriched by adding obstacles and chicanes. Since it has been noticed that in some cases the original project could be too trivial, a maximum turning angle limiter has also been implemented.
A race mode has been implemented, with the relative graphic visualization, to allow different models to compete against each others. This gives the environment a more competitive nature.
Different DRL algorithms have been added as well, to give some baselines and to test the environment from various perspectives. Finally, several experiments and tests has been conducted with the aim to verify its viability as a reinforcement learning environment with the needed characteristics.

Structure of the thesis

In the first chapter, Reinforcement Learning is introduced. A description of its basic mechanics is given, and the techniques on which it relies are explained as well. Deep Reinforcement Learning is finally described and the structure of the most modern DRL algorithms is presented.

In the second chapter, the most famous car-racing Reinforcement Learning environments are mentioned, from the classical to the latest developed. Only the environments specifically created or adapted to didactic or experimental purposes are treated. Their features are also specified.

The third chapter is dedicated to the complete description of MicroRacer. There is a detailed explanation of its inner working process. The structure of the states and actions are also depicted and the interface used by the environment is outlined. Furthermore, it is explained how to launch a competitive race, an evaluation run and how they work.

The fourth chapter, finally, contains the explanation of the tests conducted on MicroRacer to demonstrate its functionality. The results of the tests are also present and a description of catastrophic forgetting as a problem commonly encountered using this environment is included.

Contents

List of Figures

List of Tables

Chapter 1

Reinforcement Learning

Let's first introduce the meaning of reinforcement learning, which is a particular branch of machine learning. When talking about reinforcement learning, it is useful to cite the definition given by Sutton and Barto in their book *Reinforcement learning: an introduction*. They state that "Reinforcement learning (...) is simultaneously a problem, a class of solution methods that work well on the problem, and the field that studies this problem and its solution methods." [17]
This field employs methods studied to solve problems by using algorithms capable of learning the right action to perform, in a given situation. The algorithm cyclically interacts with an environment of some sort in a "trial and error" way; as a result, it collects a feedback in the form of a reward signal. The reward given by the environment can be a positive or negative feedback and represents a score given to the action happened in a specific environment condition. The main aim of this process is not to obtain an higher reward in a specific moment, instead, it is to maximize the future cumulative rewards. In fact, the rewards given after each interaction will be used to optimize the process of choosing a specific action over another, to maximize future rewards.

This process is opposed to other branches of machine learning, like supervised and unsupervised learning. These two mentioned methods, indeed,

always use pre-built data collections, and in supervised cases pre-labeled data collections, in order to learn.

1.1 Markov Decision Process

Reinforcement learning typically applies the concept of Markov Decision Process as mathematical framework, in order to describe and formalize problems.
A Markov Decision Process, or MDP, is a stochastic control process composed by two main elements, that interacts in a discrete time-step way: the *agent* and the *environment*.

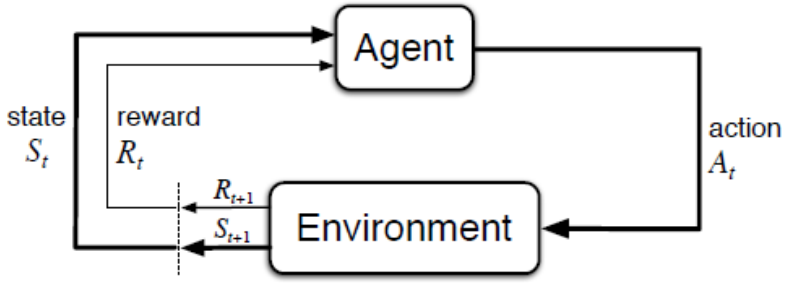

Figure 1.1: Representation of the interaction between agent and environment in a Markov decision process

For every time-step the agent will generate an *action* by checking the *state*, which is a collection of data that describes the current environment configuration. This action will influence the environment, modifying its configuration. Therefore, the environment will respond with a new state and a reward, which will be relative to the action taken in the previous state. This cycle will be repeated continuously by the algorithm.
Each state-action combination can lead to diverse paths and different future rewards, as a result the agent will try to use all of these data to maximize future rewards. Each time-step can be described by the sequence (s, a, r, s'); where s represents the current state, a is the action taken, r is the reward

obtained using action a in the state s, and s' is the new state resulted from action a in state s.
The concatenation of each sequence for every time-step is called a *trajectory*:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, S_3... \tag{1.1}$$

This sequence can be endless in the case that this process hasn't a temporary limit. In this specific case, the problem is called a *continuing task*. On the contrary, if the problem has a termination, it is called an *episodic task*.
Each new state and reward, generated by the environment at time t consequently to a state-action pair, have a discrete probability distribution defined as follow:

$$p(s', r|s, a) \doteq Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\} \tag{1.2}$$

for each $s, s' \in S$, $r \in R$, $a \in A$, with S as the state space, R as the possible rewards and A as the action space. This makes possible that every future reward and state depend only on the current state and action, with no need of previous information. This property is called *Markov property*.
As the agent will try to maximize the total amount of rewards it receives, it is necessary to calculate the cumulative rewards of a trajectory. To do so, we need to introduce the concepts of expected return and discount rate.

1.1.1 Returns

The *expected return* is the sum of all the rewards in a trajectory. However, if we need to apply this to a process that has no end, as in a continuing task, we need to assign a weight to future rewards. Otherwise, in the long run the sum will be infinite. This weight is called the *discount rate*.
The discount rate is a number between 0 and 1, which represents the importance of future rewards. If the rate is 0 or closer to 0, future rewards will have little or no relevance. On the contrary, the closer to 1 they are, the more relevant they will be considered.

The expected return G_t is defined as follows:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{1.3}$$

where γ is the discount rate. As a result, a reward in a distant future step k is only worth γ^{k-1}.

1.2 Policy and Value function

Additional concepts that it is necessary to introduce in order to better understand this process are *policy* and *value function*.

1.2.1 Policy

As Sutton and Barto state, "A policy is a mapping from states to probabilities of selecting each possible action." [17] A policy $\pi(s)$ defines the behaviour of the agent, which corresponds to the choice of the action, according to the state of the environment. The policy is constantly being updated by the process, according to the experience gained.
There are two different kinds of policies.
The first kind is called *stochastic policy*, if it's based on a probability distribution; while the second kind is called *deterministic policy*, if there is a direct association between state and action.

1.2.2 Value function

The value function is a function which attributes a numerical value to every state or state-action pair. The numerical value given by the function is an estimation, which represents how much is worth a given state or performing a specific action in a given state in terms of future rewards. The value function directly depends on the policy, since the value function is updated by the received rewards, which are the consequences of the actions chosen

by the policy. The policy itself can be updated according to the values estimated by the value function.
The state-value function of a state s under a policy π, is formally known as:

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s\right], \text{ for all } s \in S \quad (1.4)$$

where $\mathbb{E}_\pi[\cdot]$ is the expected value of a variable having the agent follows policy π and t as any time-step.
In the case of the action-value, the function is the following:

$$q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a\right] \quad (1.5)$$

The value function demonstrates a recursive relationship between the value of the current state and the values of its successors states. This relationship is defined as follows:

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s']\right] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right], \text{ for all } s \in S \end{aligned} \quad (1.6)$$

And this equation is called *Bellman equation* for v_π.

1.2.3 Optimality

The objective of a reinforcement learning training process is to obtain a policy that performs better than any other; it is called *optimal policy*. In order to find the best one, a method to compare policies is required. To do so, we need to use the value function. A policy π is better or equal than another policy π' if and only if $v_\pi(s) \geq v'_\pi(s)$ for all $s \in S$. The optimal policy π_* (which isn't unique) is, therefore, the one determined by the optimal value function v_*. It is defined as follow:

$$v_*(s) \doteq \max_\pi v_\pi(s) \text{ for all } s \in S \quad (1.7)$$

In the same way, an optimal policy also has an optimal action-value function defined as:

$$q_*(s,a) \doteq \max_\pi q_\pi(s,a) \text{ for all } s \in S \text{ and } a \in A(s) \tag{1.8}$$

which can be written in terms of the value function as:

$$q_*(s,a) \doteq \mathbb{E}[R_{t+1} + \gamma v_*(St+1)|S_t = s, A_t = a] \tag{1.9}$$

As the value function satisfies the Bellman equation, the same must happen for the optimal value function. In fact, the Bellman equation for v_* is called the Bellman optimality equation and it's defined as follow:

$$\begin{aligned} v_*(s) &= \max_{a \in A(s)} q_{\pi_*}(s,a) \\ &= \max_a \mathbb{E}_{\pi_*}[G_t|S_t = s, A_t = a] \\ &= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1}|S_t = s, A_t = a] \\ &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1})|S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v_*(s')] \end{aligned} \tag{1.10}$$

This affirms that the value of a state under an optimal policy must be equal to the expected return for the best action from that state. The same happens for the action-value function, which is as shown here:

$$\begin{aligned} q_*(s,a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a')|S_t = s, A_t = a\right] \\ &= \sum_{s',r} p(s',r|s,a)\left[r + \gamma \max_{a'} q_*(s',a')\right] \end{aligned} \tag{1.11}$$

1.3 Model

A *model* is intended as a description of the environment structure and the way it works, allowing the agent to make predictions about its future behaviour. It can be used to improve the process of learning, but it is not fundamental, the states can be used as a partial description to approximately

understand the structure of the environment. By its presence or absence, we have *model-based* methods developed on planning and predictions, and *model-free* methods totally based on accumulating experience through the trial and error method.

1.4 Exploration vs Exploitation

One of the most important dilemmas of reinforcement learning is finding a balance between *exploration* and *exploitation*.
Exploring means researching possible actions in given states that have not been tried, and which may give a significant reward. On the other hand, exploiting means using actions in given states whose value has already been discovered, in order to obtain the maximum reward. As the agent learn, and the value function and the policy are updated, the agent will tend to prefer exploitation over exploration. This happens because the aim of the agent is to maximize future rewards. However, if the knowledge of both the environment and the possible rewards is incomplete, obtaining an optimal policy will be impossible. For this reason, it is necessary to implement and improve exploration techniques.
The problem is that both exploration and exploitation can lead to repeated failures, before reaching any kind of optimality. Exploration, as a matter of fact, may be negative if the states that give major rewards have already been found, resulting into constantly exploring irrelevant states. The right balance between exploration and exploitation is up to the present time an open problem, and as Sutton and Barto confirm, "The exploration-exploitation dilemma has been intensively studied by mathematicians for many decades, yet remains unresolved." [17]

1.4.1 On-policy methods

In order to explore in a better way different methods based on the action choosing strategy has been created. In *on-policy methods* a single policy is

trained and used to make decisions. This approach usually make use of a stochastic policy which become closer to deterministic as the training goes on. An example can be the ϵ–*greedy* method by which there is an ϵ probability of selecting a random action instead of the policy chosen one.

1.4.2 Off-policy methods

Opposed to on-policy methods, in an *off-policy* approach different policies are used. One policy, called behaviour policy, is used to collect data. This one can be static, using a random actions choosing process, or can be trained but in a more exploratory way. Another policy, the target policy, is the one improved using the calculated value function.

1.5 Fundamental RL Methods

It will now be defined the fundamental reinforcement learning methods on which all moderns and advanced techniques rely.

1.5.1 Dynamic Programming

Dynamic programming is a model-based algorithmic framework; therefore, it can only be applied on problems whom environment model is totally known. It is composed by different computations.
The first one, called *policy evaluation*, is based on the already given definition of a value function. Using the Bellman equation and sweeping through the state set, this computation iteratively calculates a sequence of value-function approximations for an arbitrary policy.
The next phase is the *policy improvement*; in this phase the previously calculated value function is used to update the policy. This is done by making the new policy π' greedy with respect to the value function of the original

policy π:

$$\begin{aligned}\pi'(s) &\doteq \arg\max_a q_\pi(s,a) \\ &= \arg\max_a \mathbb{E}\left[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a\right] \\ &= \arg\max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma v_\pi(s')\right]\end{aligned} \tag{1.12}$$

with $\arg\max_a$ as the action that maximizes the following expression. This process always grants a better policy, unless the policy is already optimal. When these two phases are cyclically repeated together until convergence, it is called *policy iteration*; while if the policy evaluation process is stopped after one update of each state, it's called *value iteration*.
This cyclical interaction can be generalized to all reinforcement learning methods as one process updates the value function using a given policy and the other updates the policy using the given value function and both work together to try to reach optimality. This concept is called *generalized policy iteration* (GPI).
Dynamic Programming has limited applicability since it requires a complete model. Moreover, it can be slow in the case of a really large state space.

1.5.2 Montecarlo

Montecarlo, opposed to dynamic programming, are model-free methods and don't require a complete model as they acquire experience by interacting directly with the environment. In particular, during each and every iteration, a trajectory of a complete episode is collected. Immediately afterward, in a generalized policy iteration way, the data from the trajectory are employed to perform an update of the value function by averaging the returns G_t obtained as in:

$$V(S_t) \leftarrow V(S_t) + \alpha\left[G_t - V(S_t)\right] \tag{1.13}$$

with α as a constant step-size parameter. The value function is then used to update the policy.

By the way in which the average of the returns is calculated, there are different methods as first-visit and every-visit. However, the policy can easily fall in a incorrect optima without proper exploration methods. This may happen for two main reasons. First, because in GPI the policy will act greedly with respect to the value function; second the value function doesn't have the complete model of the environment as it is updated only by using returns collected from episodes. In some cases, this can be solved by beginning each episode in a different state and randomly selecting an action. As a result, in the long run this will cover all states.
There are other ways to solve the problem. For example, by giving the policy a probability of selecting a random action in a on-policy approach, or by using a policy to explore, while training another one in the off-policy way. As this method requires a complete episode, Montecarlo can be only used on episodic tasks.

1.5.3 Temporal-Difference learning

The idea behind Temporal-difference(TD) learning methods is a combination of Montecarlo and dynamic programming. In fact, it is model-free, it can learn exclusively using experience as Montecarlo, and performs updates using learned estimates as dynamic programming.
Temporal-difference works using the GPI process. However, the policy evaluation part is different, as it is based on a technique called bootstrapping. As a matter of fact, opposed to Montecarlo, where a complete episode is required in order to calculate the value function, in TD-learning just one step is already sufficient to update it. The value function is then calculated using an estimation of future rewards $V(S_{t+1})$ as follows:

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right] \tag{1.14}$$

with α as a constant step-size parameter. As it is observable, the improvement is made using the error between the actual estimation $V(S_t)$ and the more precise estimation $R_{t+1} + \gamma V(S_{t+1})$. This specific error is called TD-

error δ_t and it's relative to the step t:

$$\delta_t \doteq R_{t+1} + \gamma V(S_{t+1}) - V(S_t). \tag{1.15}$$

More steps can also be taken before the update, and then the value function is computed using the partial returns and the estimation; In this case, called n-step TD, TD-learning becomes more similar to Montecarlo.
TD-learning is, therefore, usable in both episodic and continuing tasks but, as in Montecarlo, exploration techniques must be implemented. An example of TD-learning methods can be SARSA[17] which is on-policy and Q-learning[18] as off-policy. Comparing TD-learning with Montecarlo, it is also possible to notice that there are other two differences. TD-learning is in fact biased, as the value function is calculated partially using an estimation, which is opposed to Montecarlo where actual returns are used.
On the other hand, Montecarlo has a higher variance since the value function depends on long trajectories. During these trajectories, the decisions taken at each step can be wrong as later collected trajectories can greatly change the policy. Opposed to this, in TD the policy is improved at every step, making it more precise.

1.6 Deep Reinforcement Learning

In the case of a very large state or action space, even continuous, finding a solution to a reinforcement learning problem in a traditional way could become really time and space consuming. In this case, when a never visited state is encountered, it is useful to generalize using the already collected data. To do so, function approximators can be employed for the policy and the value function, transforming them from a fixed association to a function parametrized by a weight vector. One of the most commonly used function approximators are *Neural Networks*.

1.6.1 Neural Networks

Neural network (NN) are networks made of interconnected nodes, shaped as a sort of weighted, directed and acyclic graph inspired by biological nervous systems. They usually have an input layer, an output layer, and a variable number of intermediate "hidden" layers. Each layer can have a different number of units and each unit can be connected to one or more units in next layer by an edge which has a real-valued weight associated.

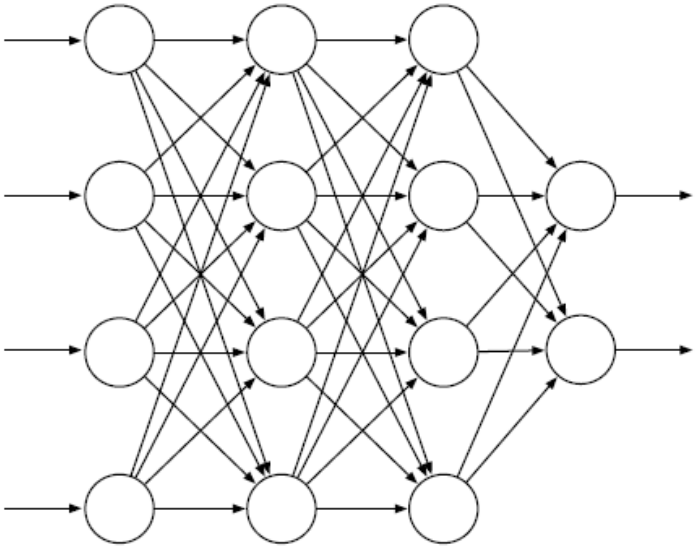

Figure 1.2: A neural network with four inputs units, two hidden layers and two output units

Neural networks used in reinforcement learning usually work in a "feed-forward" way as the signal compute a weighted sum, traveling through edges, from the input layer to each of the intermediate layer and finally out of the output layer. After each layer, a function, called activation function, can be applied to the signal. In order to make the approximation computed by a neural network more precise, an update to the weights must be performed. This update is done by using a stochastic gradient method, as each weight is adjusted with the aim of minimizing the difference (or loss) between the neural network output and the desired output, which is called backpropaga-

tion. In the reinforcement learning case, the loss is the mean squared error between the approximated value and the true value calculated from collected data.
As the number of states in a state space exceeds the number of weights in the neural network used to approximate the value function, each update to the function for a particular state influences also the other states. As a result, this makes improbable to reach a global optima, especially for complex functions. However, often a local optima is enough.

1.6.2 Policy Gradient methods

In policy gradient methods, instead of a value function, the policy is parameterized in a function approximation way, and trained by the process. This approximation is usually done using a neural network. It permits the policy to choose an action, given a state, without consulting the value function, as it associates a different probability to each action. Even if it is not parameterized in this methods, the value function can still be used to learn the policy parameters. In order to learn the parameters of this function, they are updated using a gradient of a performance measure $J(\theta)$ with respect to the policy parameters θ. The performance must be maximized, for this reason, a gradient ascent is used, and the update is:

$$\theta_{t+1} = \theta_t + \alpha\widehat{\nabla J(\theta_t)} \tag{1.16}$$

where $\widehat{\nabla J(\theta_t)} \in \mathbb{R}^{d'}$ is a stochastic estimate of the gradient of the performance measure. In this way, it is possible to let the agent learn specific probabilities for each action or even an appropriate level of exploration, which can be reduced close to determinism in the long run.

Actor-Critic methods

Actor-critic methods are particular policy gradient methods, where both the policy and the value function approximations are learnt. In this case the value function, called the critic, is used to bootstrap an estimation of the

performance of the policy, which is called the actor. In fact, usually the actor parameters are updated using as loss the value given by the critic. Then, the critic parameters are updated in the usual way, using the data collected by the actor.

Continuous action space

Policy gradient methods, thanks to policy approximation, are able to handle continuous action space.
In continuous action space, opposed to discrete action space, there are an infinite number of actions, as they are included in a real numbers range. In order to deal with these infinite actions, instead of learning action probabilities, it is necessary to learn a probability distribution, which can be a normal distribution. In this case, the output of the policy approximator will be the mean, and in some cases the standard deviation, of the normal distribution. This normal distribution will be then used to sample an action.

1.7 Modern Algorithms

1.7.1 DDPG

Deep Deterministic Policy Gradient (DDPG) is a model-free actor-critic algorithm developed by Lillicrap et al[12], intentionally created to handle continuous action space problems. In fact, it uses function approximators for both the policy and the action-value function.
DDPG works in a off-policy way as the networks are updated at each time-step, using mini-batches of samples collected during the training, and saved in a replay buffer $\mathcal{R}$. The use of a replay buffer is done in order to minimize correlations between samples; therefore, it must be of an adequate dimension. Two separated target networks for both the policy $\mu'(s|\theta^{\mu'})$ and the action-value function $Q'(s, a|\theta^{Q'})$ are created as a copy, and used to calculate the target value. These are slowly updated using the learned networks by polyak

averaging: $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$ with $\tau \ll 1$. This is done to avoid the divergence caused by the use of the same network to both update and calculate the target value. The exploration in this method is guaranteed by a noise sampled from a normal distribution, injected in the action chosen by the policy. The action-value function is updated by minimizing the loss given by the mean-squared Bellman error, using the target functions for bootstrapping:

$$L(\theta^Q) = \mathbb{E}_{(s,a,r,s',d)\sim\mathcal{R}}\left[(Q(s,a|\theta^Q) - (r + \gamma(1-d)Q'(s', \mu'(s'|\theta^{\mu'})|\theta^{Q'}))^2\right] \tag{1.17}$$

where $(1-d) = 0$ if s' is a terminal state. The policy, instead, is updated by maximizing the action-value function by applying a gradient ascent that uses as loss:

$$L(\theta^\mu) = \mathbb{E}_{s\sim\mathcal{R}}[Q(s, \mu(s|\theta^\mu)|\theta^Q)]. \tag{1.18}$$

Both of them perform a single gradient step at each time-step.

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Figure 1.3: DDPG pseudocode

1.7.2 PPO

Proximal Policy optimization (PPO) is an actor-critic method developed by the OpenAI team [16] which can be used on both discrete and continuous action space environments. PPO works in a on-policy way, and the exploration is granted by a stochastic policy which becomes progressively less random as the training goes on. The idea behind PPO is to limit the possible policy update steps in order to avoid a performance collapse caused by large updates.

This can be done in two ways: either by applying an adaptive penalty on KL-divergence called *PPO-Penalty*, or by clipping the objective function to reduce bigger steps called *PPO-Clip*. PPO-Clip has been found to perform better and will be described in detail. The algorithm works by cyclically collecting a trajectory of predetermined length with the current policy and then computing the returns R_t and the advantages with an arbitrary advantage method, using the current value function. The policy loss is then calculated. This is done, as first, by taking the ratio between the state-action probabilities of the previous policy π_θ and the new current policy π_{θ_k}. Successively, in order to take the smallest step, the minimum between this ratio multiplied to the advantage, and the same ratio but clipped and then multiplied to the advantage is taken as loss:

$$L(s, a, \theta_k, \theta) = \min \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a), \text{clip} \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1 - \epsilon, 1 + \epsilon \right) A^{\pi_{\theta_k}}(s, a) \right) \tag{1.19}$$

with ϵ as an hyperparameter that measures the distance allowed from the old policy. The policy is then updated with gradient ascent for a selected number of epochs, which can be early stopped if the new policy grows too much. The value function, instead, is trained applying a gradient descent that uses as loss the mean squared error between the value generated by the network $V(s_t)$ and the return R_t. The trajectory is then discarded, the old policy saved and a new cycle begins.

Algorithm 1 PPO-Clip

Input: initial policy parameters θ_0, initial value function parameters ϕ_0
for $k = 0, 1, 2, ...$ **do**
  Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
  Compute rewards-to-go $\hat{R}_t$.
  Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
  Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min \left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \quad g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

  typically via stochastic gradient ascent with Adam.
  Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

  typically via some gradient descent algorithm.
end for

Figure 1.4: PPO pseudocode

1.7.3 TD3

Twin Delayed DDPG (TD3) is an actor-critic method developed by Fujimoto et al[8] and designed to address the problems present in DDPG. In fact, DDPG presents some issues. For example, it is high hyperparameters dependent and has Q-value overestimation.

TD3 is very similar to DDPG, as it works with continuous action problems, it is off-policy, explores by adding noise to the deterministic actions determined by the policy. Moreover, it makes use of a replay buffer from which mini-batches are sampled, and it performs "soft" polyak updates to the target networks.

On the other hand, in order to solve the problems of DDPG, three improvement has been made to TD3. The first one, called target policy smoothing, consists of adding clipped noise to the action selected by the target policy, which is used to bootstrap in the Bellman error. This is done in order to avoid

sharp peaks over some actions in the Q-function approximator, smoothing them over near actions.

The second one is the introduction of a second Q-function and the relative target function. The smaller Q-value from the target Q-functions is then used to calculate the Bellman error employed to update them both. This prevents overestimation in the Q-function. This technique is called clipped double-Q.

The last improvement involves the delaying of the policy update, as it is done less frequently than the Q-function. This happens in order to further slow down the target change.

Algorithm 1 TD3

Initialize critic networks Q_{θ_1}, Q_{θ_2}, and actor network π_ϕ
with random parameters θ_1, θ_2, ϕ
Initialize target networks $\theta_1' \leftarrow \theta_1$, $\theta_2' \leftarrow \theta_2$, $\phi' \leftarrow \phi$
Initialize replay buffer $\mathcal{B}$
for $t = 1$ **to** T **do**
  Select action with exploration noise $a \sim \pi_\phi(s) + \epsilon$,
  $\epsilon \sim \mathcal{N}(0, \sigma)$ and observe reward r and new state s'
  Store transition tuple (s, a, r, s') in $\mathcal{B}$

  Sample mini-batch of N transitions (s, a, r, s') from $\mathcal{B}$
  $\tilde{a} \leftarrow \pi_{\phi'}(s') + \epsilon, \quad \epsilon \sim \text{clip}(\mathcal{N}(0, \tilde{\sigma}), -c, c)$
  $y \leftarrow r + \gamma \min_{i=1,2} Q_{\theta_i'}(s', \tilde{a})$
  Update critics $\theta_i \leftarrow \text{argmin}_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(s, a))^2$
  if t mod d **then**
    Update ϕ by the deterministic policy gradient:
    $\nabla_\phi J(\phi) = N^{-1} \sum \nabla_a Q_{\theta_1}(s, a)|_{a=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$
    Update target networks:
    $\theta_i' \leftarrow \tau\theta_i + (1 - \tau)\theta_i'$
    $\phi' \leftarrow \tau\phi + (1 - \tau)\phi'$
  end if
end for

Figure 1.5: TD3 pseudocode

1.7.4 SAC

Soft Actor Critic (SAC) is an off-policy actor-critic algorithm developed by Haarnoja et al[10] that can work on both continuous and discrete action problems. Similarly to TD3, it makes use of the clipped double-Q technique to update the Q-functions with the mean squared Bellman error.
Even if SAC has a stochastic policy opposed to TD3, SAC benefits from the policy smoothing partially done by the stochasticity. Other similarities between DDPG and TD3 are the employ of a replay buffer and the use of target Q-networks updated by using Polyak averaging.
The focal point of SAC is the introduction of the entropy of the generated distribution H, which is used as a sort of bonus reward to the action-value function:

$$Q^{\pi}(s,a) = \mathbb{E}[R(s,a,s') + \gamma(Q^{\pi}(s',a') + \alpha H(\pi(\cdot|s')))] \tag{1.20}$$

with α as a entropy regularization coefficient which can be fixed or trained. Lastly, SAC doesn't use a target policy as it employs the current policy to compute the next state-action used for the update. In this case, the exploration is granted by the stochastic policy and the α coefficient of the entropy term as higher values corresponds to more exploration.

Algorithm 1 Soft Actor-Critic

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
repeat
  Observe state s and select action $a \sim \pi_\theta(\cdot|s)$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for j in range(however many updates) **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets for the Q functions:

$$y(r, s', d) = r + \gamma(1 - d)\left(\min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s')\right), \quad \tilde{a}' \sim \pi_\theta(\cdot|s')$$

      Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_i}(s, a) - y(r, s', d)\right)^2 \qquad \text{for } i = 1, 2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \Big(\min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta\left(\tilde{a}_\theta(s)|\, s\right) \Big),$$

      where $\tilde{a}_\theta(s)$ is a sample from $\pi_\theta(\cdot|s)$ which is differentiable wrt θ via the reparametrization trick.
      Update target networks with

$$\phi_{\text{targ},i} \leftarrow \rho \phi_{\text{targ},i} + (1 - \rho)\phi_i \qquad \text{for } i = 1, 2$$

    end for
  end if
until convergence

Figure 1.6: SAC pseudocode

1.7.5 DSAC

Distributional Soft Actor-Critic (DSAC) is an off-policy actor-critic algorithm developed by Jingliang et al[7] that can be considered a variant of SAC. In fact, it makes use of almost all the techniques used in SAC with one exception: the clipped double-Q learning is substituted by a distributional

action-value function. The distributional action-value function, instead of a value, returns a distribution. The use of a distribution, as clipped double-Q learning, can mitigate Q-function overestimation, but with a different approach. Furthermore, it uses a single network for the action-value estimation and that improves time efficiency. While the loss function used for the actor remains the same as SAC, the one used for the critic is defined as follows:

$$J_{\mathcal{Z}}(\theta) = - \mathop{\mathbb{E}}_{(s,a,r,s')\sim\mathcal{B},a'\sim\pi_{\phi'},Z(s',a)\sim\mathcal{Z}_{\theta'}(\cdot|s',a')} \Big[\log \mathcal{P}(\mathcal{T}_{\mathcal{D}}^{\pi_{\phi'}} Z(s,a)|\mathcal{Z}_{\theta}(\cdot|s,a))\Big] \tag{1.21}$$

with $\mathcal{T}_{\mathcal{D}}^{\pi_{\phi'}} Z(s,a) = r + \gamma(Z(s',a') - \alpha \log \pi(a'|s'))$.

Algorithm 1 DSAC Algorithm

Initialize parameters θ, ϕ and α
Initialize target parameters $\theta' \leftarrow \theta$, $\phi' \leftarrow \phi$
Initialize learning rate $\beta_{\mathcal{Z}}$, β_{π}, β_{α} and τ
Initialize iteration index $k = 0$
repeat
  Select action $a \sim \pi_{\phi}(a|s)$
  Observe reward r and new state s'
  Store transition tuple (s, a, r, s') in buffer $\mathcal{B}$

  Sample N transitions (s, a, r, s') from $\mathcal{B}$
  Update soft return distribution $\theta \leftarrow \theta - \beta_{\mathcal{Z}} \nabla_{\theta} J_{\mathcal{Z}}(\theta)$
  if k mod m **then**
    Update policy $\phi \leftarrow \phi + \beta_{\pi} \nabla_{\phi} J_{\pi}(\phi)$
    Adjust temperature $\alpha \leftarrow \alpha - \beta_{\alpha} \nabla_{\alpha} J(\alpha)$
    Update target networks:
      $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$, $\phi' \leftarrow \tau\phi + (1-\tau)\phi'$
  end if
  $k = k + 1$
until Convergence

Figure 1.7: DSAC pseudocode

Chapter 2

Car-racing RL-Environments

It is obvious that every environment that can be, even partially, described with a Markov Decision Process can be used to train a reinforcement learning algorithm. However, to learn how to implement a reinforcement learning algorithm, to test the performance of a particular method or to study a solution to a known problem can be really hard on a generic environment. For this particular purpose, dedicated environments have been specifically created.

These environments are studied to help the understanding of RL mechanics, with explicit states and actions descriptions, well-shaped rewards, and also metrics to measure the performance. There are even environments created to deal with a particular problem, as for instance sparse rewards. Different platforms already exist to satisfy this purpose. A description of the most famous ones, with special attention to the racing ones, will be given as follows.

2.1 OpenAI Gym

OpenAI Gym[1][4] is an extensive open-source toolkit, developed by OpenAI, which includes different simulated environments and is continually growing. These environments can vary a lot, from tasks proposed in RL literature to Atari games, from simple text environments to even simulated robots. This permits to cover different problems, like discrete and continuous action or state spaces, sparse rewards and screen images as input.
Every environment has a common interface that permits to write general algorithms, which will work on all of them. This interface, which can also be implemented on external environments too, to render them "Gym-compliant," is an effort to standardize a common interface to reinforcement learning environments.
OpenAI Gym tries to offer a benchmark as well, in order to compare the performance of different methods or different implementations of the same method. This is done by permitting to reproduce the results obtained by using a method.
OpenAI also offers baselines of the most common reinforcement learning algorithms, optimized for these environments, which can be used as reference.

[1] `https://gym.openai.com/`

CarRacing-v0

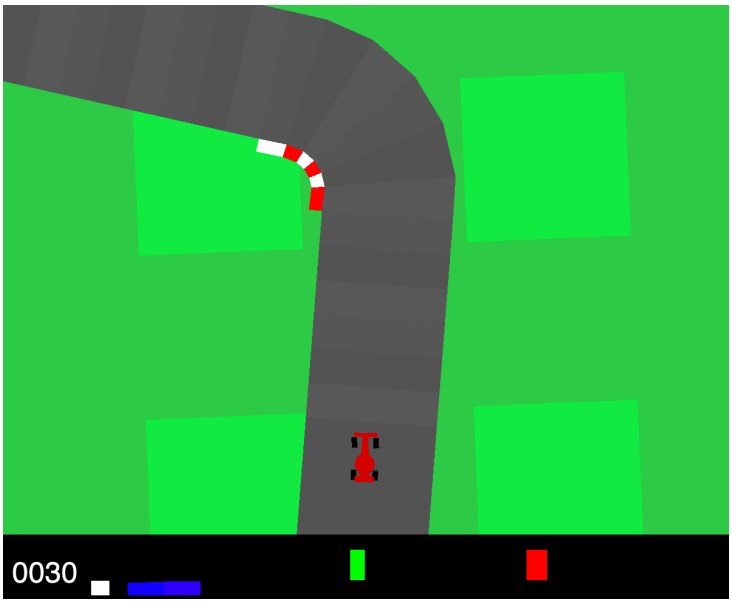

Figure 2.1: CarRacing-v0 example

CarRacing-v0 is a model-free environment from OpenAI Gym, consisting of a top-down view car race aimed to let the car learn to finish the track as fast as possible, without excessively leaving the track. The track is randomly generated at each episode and it is composed by a number of tiles; when all tiles are visited the episode terminates.
It uses a continuous action space, but it gives the possibility to use discrete actions. An action is composed of three values: steering, accelerating and decelerating, while each observation is a 96x96 pixels RGB image. As it is possible to see in figure 2.1, at the bottom of the window there are different indicators as, from left to right, true speed, four ABS sensors, steering wheel position and gyroscope. The reward given is $+1000/N$ for every tile visited, with N as the total number of tiles in a track, and -0.1 at each frame. When the agent consistently gets a total reward higher than 900, the game is considered solved.

2.2 AWS DeepRacer

AWS DeepRacer[2][3] is a platform created by Amazon Web Services, which allows the developers to learn reinforcement learning. By paying a fee, AWS DeepRacer offers a 3D online racing simulator, servers to train the agents, a console to adjust the various parameters and a log-analysis system. It's even possible to buy a real 1/18th scale electric vehicle with lidar sensors and cameras, to try the agents in real-life. AWS organizes a DeepRacer League every year for both the simulated environment and the real vehicles, where pre-trained agents run against each other.

The racing simulator is a model-free environment that replicates reality as best as it can, in order to train the agent to be used in real life as well. The objective in this environment is to finish the track without leaving its borders and, depending on the chosen mode, to race against another car, to avoid the obstacles or to run as fast as possible.

It makes use of the front camera view and the values from the lidar as states and as actions the steering angle and speed. The reward function is defined by the user: its aim is to obtain the best performance by giving the right importance to specific behaviours. The console permits to create this specific reward function, choose the optimization algorithm, tune in the hyperparameters, select the wanted track and choose the model.
The algorithms used to train the agents are limited and can't be changed as they are already given by the platform. As present, it is possible to use only PPO and SAC.

[2] `https://aws.amazon.com/it/deepracer/`

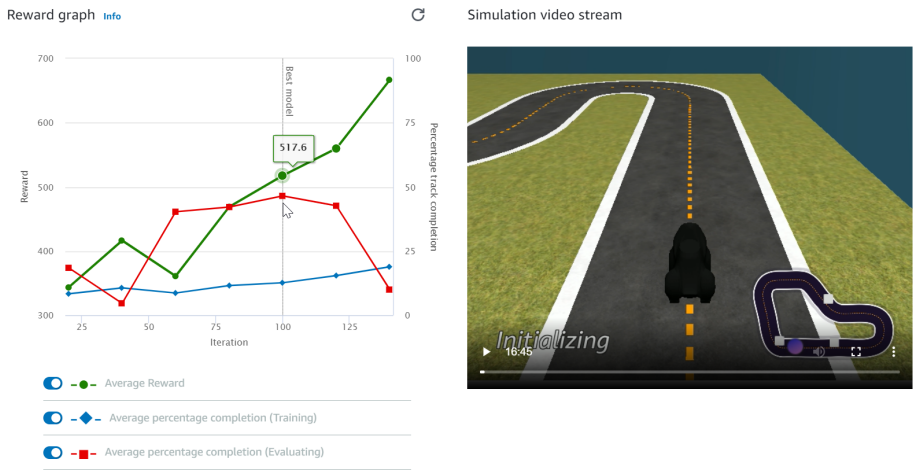

Figure 2.2: DeepRacer example

2.3 TORCS

The Open Racing Car Simulator[3] (TORCS) is a 3D car racing simulator originally developed in 1997 by Eric Espié and Christophe Guionneau. It is multi-platform as it can run on Linux, Mac OS X, FreeBSD and Windows, and open-source, licensed under GPL. Although it can be used as an ordinary racing game, TORCS has always been employed as an AI research and experimentation platform[6][14].

As it is open-source and thought to do that, it is easy to adapt the code to use it as a reinforcement learning environment, but different interfaces have already been developed for this reason[13], even to render it gym-compliant[9]. It is a realistic environment, with sophisticated real-life physics, where it's possible to select different tracks and cars. The structure of the observations in this environment is highly configurable. Indeed, it can have different readings as rangefinders, the position on the track, the engine rounds per minute, wheel speed, damage to the car, or even images of what can be seen by the car.

The possible actions that the agent can take include the steering and

[3] `https://sourceforge.net/projects/torcs/`

the acceleration in a continuous way. In this case, the reward function is not defined and need to be shaped by the developer by using the various available parameters. The choice of when to terminate one episode is also left to the developer.

Being highly customizable, this environment can be really good to test RL methods or to try and solve a known problem. On the other hand, this feature can make quite complex for developers to approach reinforcement learning for the very first time.

Figure 2.3: TORCS example

2.4 Learn-to-Race

Learn-to-Race[4] [11, 5] is an open-source realistic multi-modal control environment of recent development, aimed to let agents learn how to perform a car race. It is based on a high fidelity racing simulator developed by Arrival, and it provides a Gym-compliant interface. This environment aims to replicate, at its best, real-life vehicle dynamics in a 3D photo-realistic simulation. With the environment are also provided different baseline models

[4] `https://learn-to-race.org/`

to demonstrate how it works and how to use it. The action space used in this environment is continuous and comprehends two actions: steering and acceleration. The Observation space can be vision-only or multi-modal. In vision-only the agent receives RGB images from the camera placed in front of the vehicle. Instead, in multi-modal in addition to images it receives sensor data such as readings from lidars or depth cameras. The default reward function provides positive rewards for progressing on the track, while keeping a competitive speed, and negative rewards for getting outside the given bounds but it is possible to customize this function. The environment also provides different metrics in order to help measure both the performance and the quality of the agents trained.

Currently, there are only two tracks to race in these environment; however, it is possible to create more environments by using various given tools. Moreover, it offers a vehicle builder mode, which allows the user to customize every part of the vehicle, from the physical model to the mechanical components.

AICrowd is currently organizing a challenge, in order to get the best performing agents, capable of high speeds while respecting the safety constraints. As TORCS, this environment can be hard to use by developers who are new to reinforcement learning.

Figure 2.4: Learn-to-Race example

Chapter 3

MicroRacer

MicroRacer is an environment inspired by car racing and its primary purpose is to be a didactic tool used to teach and experiment with reinforcement learning methods. It has been developed to be simple, lightweight, without requiring a very long training but still stimulating and not trivial. It is a model free environment with continuous action and state space. The physics is very simple, without considering things as wheel friction or aerodynamics, but still remaining realistic in some way. The track is generated randomly at each episode and it's possible to add obstacles or chicanes to keep it interesting. In order to let the agent learn how to accelerate and decelerate, it's possible to activate a stricter limiter to the turning angle at high speed. Moreover, the user can launch races between different agents from different algorithms or from the same one, but trained with different hyperparameters.

MicroRacer is open-source and mainly written in Python, with a couple of functions in Cython to improve its efficiency. It requires basic libraries as `tensorflow`, `numpy`, `scipy`, `matplotlib`, `tensorflow_probability` and `cython` in order to work. Baselines of different deep reinforcement learning methods are also included: indeed, it offers an example to help the user understand how to use the environment and how this method works. The code is available on GitHub at: `https://github.com/asperti/MicroRacer`.

3.1 Track structure

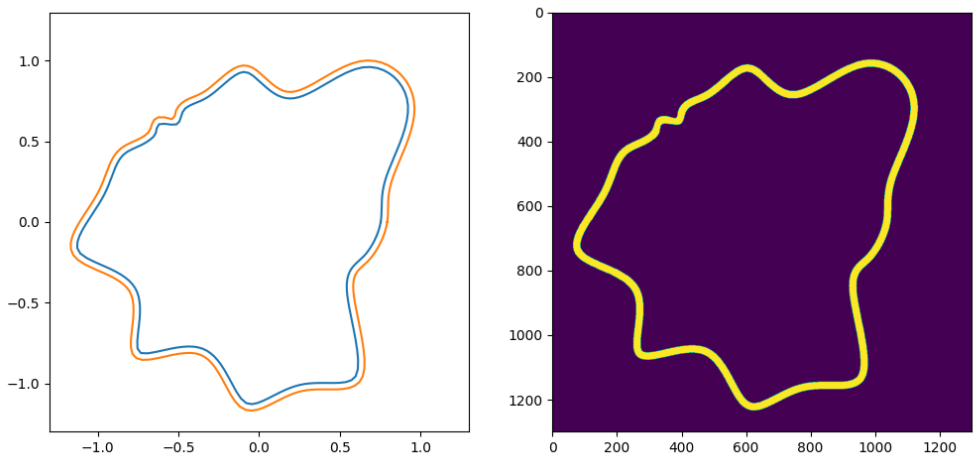

Figure 3.1: Example of a generated track. On the left representation of the track using the generated splines; on the right representation of the boolean matrix map.

The circular track in MicroRacer is generated by using CubicSplines. The idea is to deform a circle of radius 1 centered on the 0 of the plane. At first, the turns are defined using a configurable number of angles in radians equally spaced between 0 and 2π, 20 as default. The sine and cosine of these angles are used to generate points on the plane; then, on these points a random offset is applied and their position is smoothed to avoid sharp turns. Finally, an interpolation is done on these points, having the first and last point equal, and the resulting spline will represent the mid line of the track. Using this first spline, two more splines are generated, representing the borders of the track.

This is done firstly by taking a high number of points on the first spline. Then, two more points are found perpendicular to the tangent of each of these points. The distance between the mid point and the two border points is configurable and it represents half the track width. As default it is 0.02, but it can be customized if necessary. The new points are then interpolated

to generate both the external and the internal spline, defining the borders of the track.

A dense matrix of points is derived from these splines, in order to get a discrete description of the track, to check what is around the racer and to control if the position is valid. This matrix has a dimension of 1300x1300 points and each of them is a boolean which represents if the position is valid (on the track) or invalid (outside the track or on an obstacle). The track is represented on the matrix by cycling in a radial motion on the discretization of different points belonging to the outer spline, and the relative points on the inner spline. Then, the state of all the points between each outer and inner point is changed on the matrix. In this process the matrix is always checked, to verify that the track doesn't get outside the maximum dimension 1300x1300; otherwise, a new track is generated.

3.1.1 Obstacles generation

It is possible to generate a variable number of obstacles along the track, as default 6. These obstacles, which even if are motionless can be seen as vehicles occupying the trail, are placed on the left or the right borders of the track. Their position is chosen taking points, on the spline representing the borders, using angles in radians between 0 and 2π. These points are randomly spaced but never too close to each other. The side of the track is randomly picked, using the track width to calculate their position. Their length is calculated by taking another point at a fixed distance. Finally, the obstacles are mapped on the track matrix; this is done by setting to false the state of the points between the two used for each obstacle.

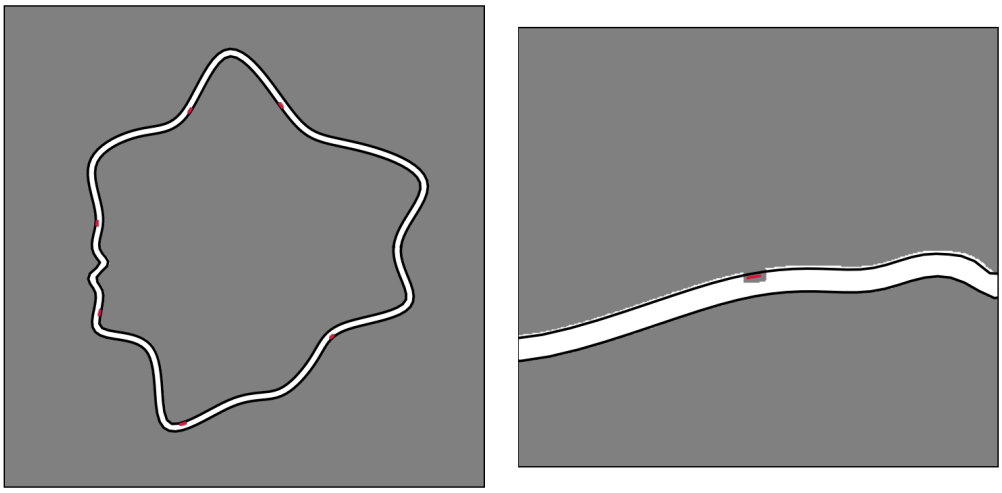

Figure 3.2: On the left, an example of a track with obstacles in red; on the right, details of one obstacle. The part in gray is considered invalid on the boolean matrix.

3.1.2 Chicanes generation

A chicane can be inserted in a random point of the track to train and test the agent with sharp turns. This chicane is created before generating the first spline. It is done by randomly choosing one of the turns with the exception of the first and the last.In particular, the point representing the turns. New points are then inserted before and after the one chosen, but still within the range of the previous and the next turn. These new points, that will be two before and two after, are employed to shape the chicane. Then, the chosen point and the new added ones will be aligned with the points relative to the previous and successive turn. Finally, the two inner new points will be displaced in opposite direction to generate the chicane. The offset representing the displacement can be changed to have bigger or smaller turns. The distance between the outer added points and the chosen point can also be customized, as it represents how sharp the turns will be. The points will then be interpolated as usual to create the mid-line spline.

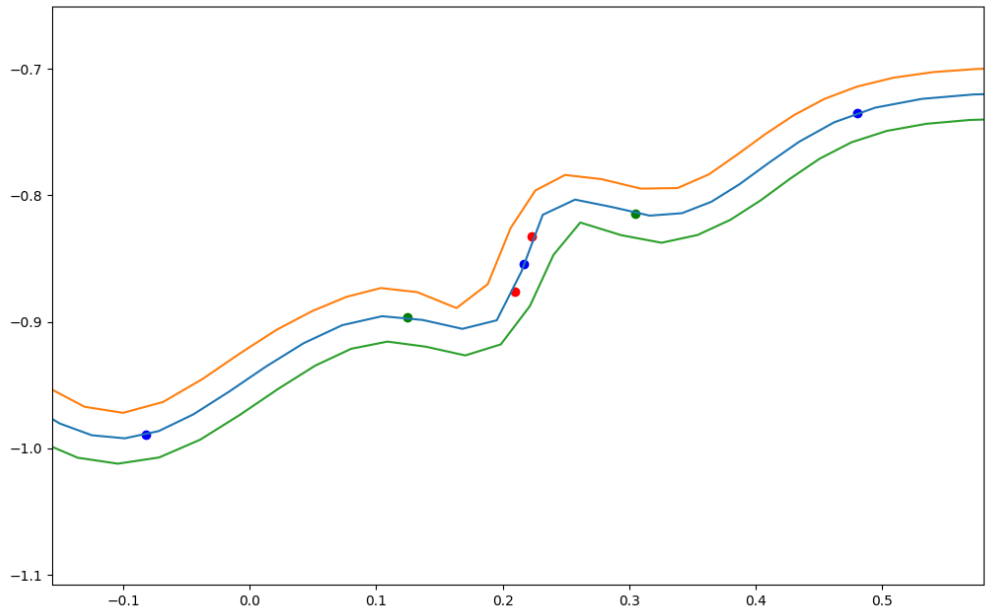

Figure 3.3: Details on chicane creation. The blue line is the mid-line and the blue dots are the pre-existing turn points, the green dots are the added outer control points, the red dots are the inner added control points while the orange and the green lines are the inner and outer borders of the track, created from the mid-line.

3.2 Racer internal state

As a model-free environment, the agent outside of it knows nothing about the structure of the track or the current position of the racer on the track. In fact, this is all saved and calculated by the environment. In order to better understand the environment, it is useful to know that it has been thought with a ratio of 1:1000m. At the beginning of an episode, the starting position of the racer is always the same, as it is the point on the mid-line spline of angle 0 in radians. This position is saved as x and y points of the plane. The starting direction, instead, is the negative tangent to the mid-line in the starting position; while the starting speed is randomly selected (currently

between 0 and 0.5). The direction and the speed are saved as a vector with its length as the speed and direction as the racer direction. At each step the speed, direction and position are updated. The time-step, which currently is 0.04, can be configured to experiment with the frequency of the observations received. In fact, it has been found that it can influence the behaviour of the agents. The current speed is updated using the acceleration selected by the agent, which can be positive (speed up) or negative (slow down). The speed can reach 0 but can never be negative. The current direction is updated using the direction chosen by the agent. The maximum turn the agent can make in a step corresponds to $\pi/6$ (30°) in each direction. It is possible to activate a limiter to the max turning angle dependant on the current speed, to force the agent to accelerate and decelerate according to the track shape. This is calculated according to a simple law of tolerated angular acceleration as follows:

$$\Theta = \min\{\frac{4\pi^4 aT}{v}, maxturn\} \tag{3.1}$$

with Θ as the new max turning angle, *maxturn* as the old maximum turning angle ($\pi/6$), a as the maximum tolerated acceleration (currently 4.5g), T as the time-step and v as the current speed. Finally, having the new speed and direction, the position of the racer is changed accordingly.

3.2.1 Termination

After updating the position, speed and direction, the environment checks if the episode is in a termination condition. There are different conditions that may end an episode. The first and more obvious one is if the racer complete the whole track, returning to the starting point. In this case the environment checks if the angle of the old position in radians is in the upper half of the track, while the angle of the new position is in the lower. This means that the racer has again surpassed the starting point, as the race follows a clockwise direction and the starting point is precisely on the right half.

The second condition is when the racer get outside the track or over an obstacle. This is checked by discretizing the racer current position and verifying that the discrete equivalent is a valid position on the map matrix. If not, the episode will end.

Another termination condition is when the racer reverses its direction. This is verified by checking if the angle of the racer's new position is greater than the old. In fact, the race goes clockwise and the circonference angle in radians grows from 0 to $-\pi$ and then from π to 0 again to complete the circle. In this way, to go in the right direction the racer's angle should always decrease. The only exception is when the old angle is in the lower half and the newer is in the upper: in this case there is a special check.

Finally, there is an optional termination condition that put an end to an episode if the racer speed goes lower than a certain value. This one can be activated, if necessary, to avoid the use of an excessively slow speed in order to safely complete the track. In this case, the lower bound to the starting speed will be increased over the speed limit.

3.2.2 Lidar

As the agent has no knowledge of the global structure of the track or the position of the racer on the track, a lidar-like vision is present to give the agent the description of the current surroundings. It is composed of an array of 19 lidar simulations equally spaced in a range from -30°to 30°with respect to the front of the racer. Each of these values measure the distance to the borders of the track at a given angle. This distance is calculated by iteratively performing steps in the direction of each lidar. At each step, the relative discrete location and the directly adjacent points are checked on the map matrix to see if it is a valid or invalid position. If it is invalid, the distance measured is returned.

In order to get a safe and precise measurement, these steps are shorter both near the racer and when a border is encountered. The functions implementing this lidar simulation are written in Cython. In fact, these func-

tions perform a high number of computations at each time-step and can slow down the environment. Cython solves this problem, as it optimizes the code, speeding up its execution.

3.3 State, Action and Reward

3.3.1 State structure

The state, that the environment provides to the agent at each step, is a partial description of both the current surroundings and the speed of the racer. It makes use of the lidar array, but in order to simplify the learning process only a limited number of lidar values are used. It is composed of five elements:

Direction This value represents the direction with the greater distance from the track's borders. It is the angle of the element with the greater value in the lidar array. This angle is relative and must be applied to the current direction of the racer.

Left distance This is the distance value calculated by the lidar directly adjacent to the left of the one with the maximum distance. If the maximum one is the first, these two values are equal.

Max distance This is the maximum distance to the borders from the values in the lidar array. To be more precise, it is the lidar which direction is in the first element of the state.

Right distance In the same way, this is the distance value calculated by the lidar directly adjacent to the right of the one with the maximum distance. If the maximum one is the last, these two values are equal.

Speed This value is the current scalar velocity of the racer.

3.3.2 Action structure

The agent can take two actions to influence the behaviour of the racer in the environment. Both these actions are continuous and must be in the range [-1,1]:

Acceleration/deceleration If the action is positive the racer will accelerate. On the other hand, if the action is negative it will decelerate. Finally, if it's zero the racer will keep its current speed.

Turning angle If the action is positive the racer will turn right, and vice versa. Moreover, if it's zero the racer will keep the current direction. This value is scaled by the current max turning angle.

3.3.3 Reward function

The objective in MicroRacer is to get fast agents while staying on the track. The reward function is shaped to achieve this target without using complex mechanisms. As greater speed means greater distance covered, at each time-step the environment will give as reward the distance covered by the racer in that interval. The cumulative reward is, in fact, the expected total discounted distance traveled by the racer. This happens only if the racer is in a non-termination condition. If the racer gets in a termination condition different from the completion of the track, it receives a reward of -3. It is possible to customize the reward function with different mechanisms; however, it is not recommended, since it may introduce biases during the learning process.

3.4 Environment interface

To let any reinforcement learning method use this environment, it is necessary to instantiate the `Racer` class in `tracks.py`. On declaration it is possible to turn off obstacles, chicanes, the turn limiter and the low speed termination by using: `Racer(obstacles=False, turn_limit=False, chicanes=False, low_speed_termination=False)`. This class has two methods:

`reset() -> state`
: This method generates a new track and resets the racer position on the starting point. It returns the initial state.

`step(action) -> state, reward, done`
: This method takes an action composed by `[acceleration, turn]` and lets the racer perform a step in the environment according to the action. It returns the new `state`, the `reward` for the action taken and a boolean `done` that is true if the episode has ended.

3.5 Competitive Race

In order to graphically visualize a run it is necessary to use the function:

```
newrun(actors, obstacles=True, turn_limit=True,
        chicanes=True, low_speed_termination=True)
```

which is in `tracks.py`. Using the same function it is also possible to let different agents compete against each other in real time. It takes as input a list `actors` of Keras models, which can even contain just one model.

These models can be from different methods and have a different internal structures but, in order to work, they must take as input the state and return in output the action as first element (without noise, if used). It is also possible to turn off the various optional parts and limitations on the environment. While achievable, it is not recommended to let run an high number of agents as it may excessively slow down the process. All the agents

will race using the same map and the same starting speed. The race will be visualized as a plot using the animation functions from `matplotlib`. Lines of different colors will be plot for every racer over the representation of the map, and their position will be updated at every step.

On the right there is a legend with all the racers from top to bottom, ordered by the list passed to the function. This legend contains information on the current action taken by each racer and its related speed. If the racer gets in a early termination state, the legend will show which was the cause. On the other hand, if the racer completes the track, the legend will show its ending position. While they race on the same track, the racers cannot interfere with each other and can even be in the same position, since this is just a superposition of their trajectories.

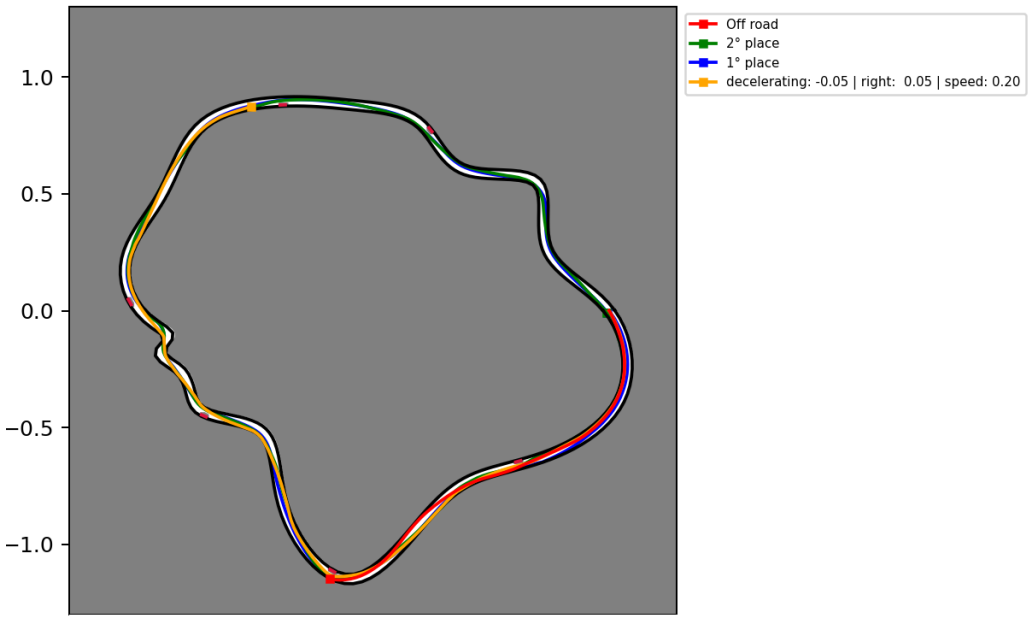

Figure 3.4: Example of a race between 4 agents: two of them have already finished the track, one went off road while another is still running.

3.6 Evaluation Run

In `tracks.py` is also present a function that permits to evaluate an agent's average performance using some metrics. The function:

```
metrics_run(actor, runs_num=100, obstacles=True,
            turn_limit=True, chicanes=True,
            low_speed_termination=True)
```

takes as input a Keras model as `actor`, the number of episodes to execute as `runs_num` and all the others parameters to activate or deactivate the various optional parts as in the two already treated cases. As in the `newrun` case, the model passed to the function must take as input a state and as output an action as first element. It will run the number of episodes specified without graphics and print at the end of each one the steps taken, the cumulative reward and the average speed for the episode. When all the episodes terminate, it will print the average reward, the average steps number and the average speed of all the executed episodes. The number of episodes completed will be printed as well. This function can be used to periodically evaluate the agent during the training process or to perform an extended performance check of an agent, in order to collect statistics.

Chapter 4

Experiments and Results

In the following section, all the tests that have been made on the environment will be showed. These tests, which have been conducted using the various methods cited in this thesis, make use of the same environment settings. In particular, all the environment optional parts have been used. Here is a list of them: obstacles, chicanes, low speed termination and turn limiter. The time-step used is 0.04, while all the other environment parameters are as previously described and haven't been altered.

4.1 Implemented methods

The methods that has been used to test the environment are those mentioned in section 1.7. These methods have been implemented following the structure described in the dedicated paragraphs. However, among the given baselines, there is a further modified implementation of DDPG (`DDPG2.py`). This implementation makes use of parameter space noise[15]. The parameter space noise is inserted into the weights of the layers of the actor's neural network. This is done to further improve exploration as it can be used in place of action noise. All the methods implement a function that randomly perform, after a chosen action, another one, composed of $[0, 0]$, which keeps the same speed and direction. This is done to partially uncouple the agent

from the environment time-step.

The DDPG actor's neural network makes use of two towers. One of them calculates the direction, while the other calculates the acceleration. Each of them is composed of two hidden layers of 32 units, with relu activation. The output layer uses a tanh activation for each action. At the same time, the critic network uses two layers, one of 16 units and one of 32, for the state input and one layer of 32 units for the action input. The outputs of these layers are then concatenated and go through another two hidden layers composed of 64 units. All of them make use of relu activation.

In DDPG2, the actor has two hidden layers with 64 units and relu activation and one output with tanh activation . Meanwhile, the critic is the same as in DDPG.

In TD3, the actor is the same as DDPG2. The critic has two hidden layer with 64 units and relu activation.

In SAC, the actor has two hidden layer with 64 units each and relu activation and output a μ and a σ of a normal distribution for each action. The critic is equal to TD3.

In DSAC, the actor is the same has SAC. The critic has the same structure as the actor.

In PPO both the actor and the critic have two hidden layers of 64 units with tanh activation, but the actor has also tanh activation on the output layer.

All the hyperparameters used for these methods are specified in Table 4.1.

Hyperparameter	Value
Shared	
Discount Factor	0.99
Optimizer	Adam
Shared except PPO	
Actor and Critic Learning Rate	0.001
Buffer Size	50000
Batch Size	64
Target Update Rate τ	0.005
DDPG2	
Parameter Noise Std Dev	0.2
TD3, DDPG	
Exploration Noise	$\mathcal{N}(0, 0.1)$
TD3	
Target Update Frequency	2
Target Noise Clip	0.5
SAC, DSAC	
Target Entropy	-A
DSAC	
Target Update Frequency	2
Minimum critic sigma	1
Critic difference boundary	10
PPO	
Actor and Critic Learning Rate	0.0003
Mini-batch Size	64
Epochs	10
GAE lambda	0.95
Policy clip	0.25
Target entropy	0.01
Target KL	0.01

Table 4.1: Hyperparameters used in the various methods.

4.2 Results

In order to get a more accurate behavioral representation of each method while using the environment, the results of five different trainings have been collected for each of them. For almost all methods, a training consisted of 50000 training steps. This was not possible for PPO as, unlike all the other methods used, it collects a complete trajectory first, and then executes a training step. Thus, performing 50000 training steps would require an incomparable amount of time while giving different results as the method improves in a different way. For these reasons, PPO has been trained using as training length a number of episodes that would give a similar training time.

The training times collected are relative to the execution of all the trainings on a machine equipped with a "NVIDIA GeForce GTX 1060" GPU, "Intel Core i7-8750H" CPU and 16GB 2400MHz RAM. As can be observed in Table 4.2, the methods that train an higher number of Neural Networks require higher training times.

Method	DDPG	DDPG2	TD3	SAC	DSAC	PPO
Average Training Time	41m	54m	1h10m	1h11m	53m	45m

Table 4.2: Average training time required to run 50000 training iterations (600 episodes for PPO) using each method calculated over 5 trainings.

The training process can vary a lot between methods, as can be seen in Figure 4.1. TD3 and SAC tend to require less observations and training steps to improve and are able to grow more linearly. The other methods improve at a slower pace and can have a jagged learning curve with occasional catastrophic forgetting. This doesn't mean that they can't reach good results though. For instance, in some trainings DDPG achieves really good scores. In fact, even using the same method, the agent obtained at the end of different trainings may be vastly dissimilar. SAC, instead, seems to be the only method that has more stable results between different trainings.

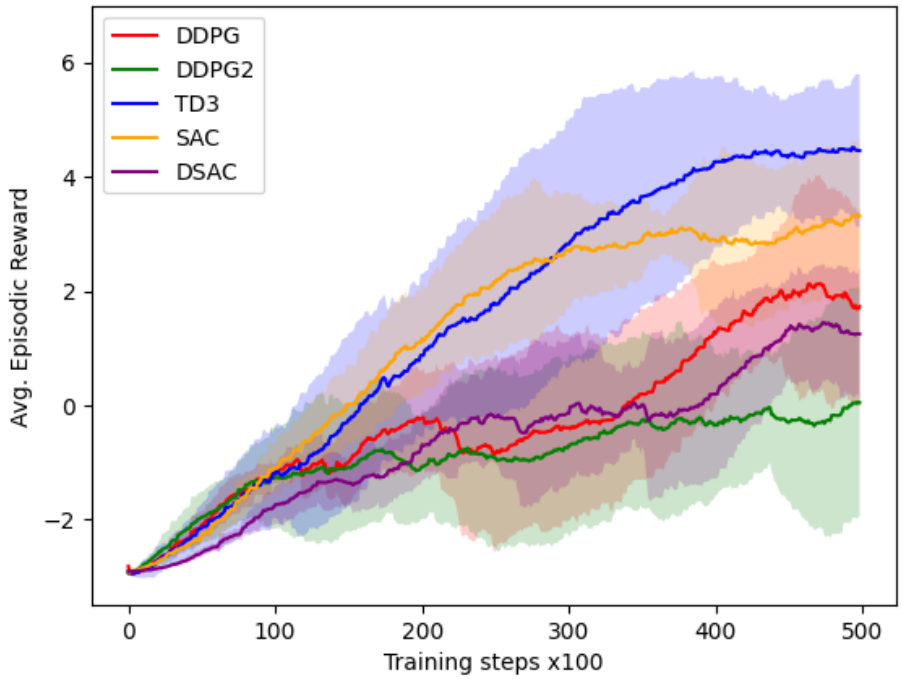

Figure 4.1: Training curves of all methods except PPO. The solid lines correspond to the mean and the shaded regions correspond to 95% confidence interval over 5 trainings.

After each training, 100 evaluation episodes has been run to collect the real performance of each trained agents. The average of these results over 5 trainings and the best results obtained for each method can be seen in Table 4.3 while the complete results for each training are displayed in the appendix. As it can be noticed, an higher number of completed episodes usually corresponds to slower speeds. This may indicate difficulties in the process of learning the right acceleration action. Similarly to the training curves, TD3 and SAC seem to have the best performances even in evaluation, as expected.

Method	DDPG	DDPG2	TD3	SAC	DSAC	PPO
Average completed episodes	38	18	54	69	37	37
Average episodic reward	2.48	0.80	3.52	4.61	2.84	2.05
Average speed	0.34	0.30	0.26	0.29	0.34	0.23
Max completed episodes	90	39	80	79	75	62

Table 4.3: Average and maximum of 100 evaluation episodes executed after each training over 5 trainings of 50000 iterations (600 episodes for PPO).

4.3 Catastrophic forgetting

One of the most interesting issues encountered during the use of neural networks, and in this specific case in their implementation within reinforcement learning, is the so-called "catastrophic forgetting." The term Catastrophic Forgetting refers to the tendency of a neural network to worsen its performance during a given training, after showing a previous improvement phase, as if it had forgotten how to act.

This phenomenon has been observed analyzing the results of several reinforcement learning methods applied within Microracer. For this reason, it could be a suitable environment for the study of this peculiar phenomenon, given the short time required for the agent to be trained and its wide customizability, which permits ablation studies.

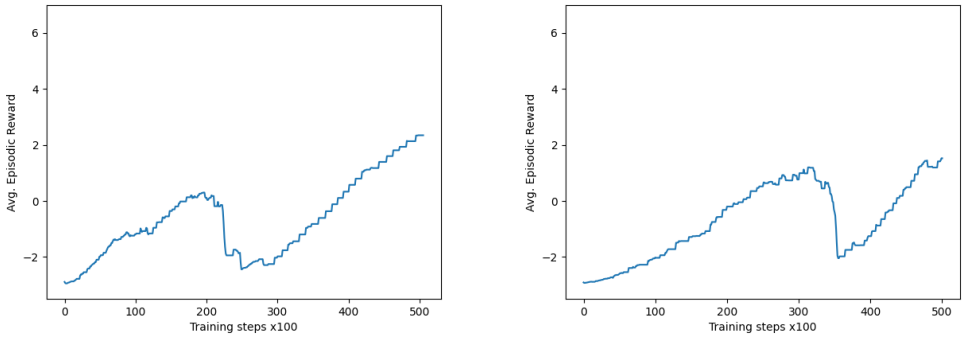

Figure 4.2: Example of catastrophic forgetting: on the left average episodic reward over 50000 training steps using DDPG, on the right average episodic reward over the same number of steps using DSAC. After a phase of catastrophic forgetting, both methods return to improve.

Conclusions

In this thesis, MicroRacer has been discussed, all its functionalities have been explained, and the various upgrades to the original Prof.Asperti's project have been illustrated. After all that have been explained, it can be certainly maintained that MicroRacer is a simple and functional didactic environment for Reinforcement Learning, while still remaining both competitive and not trivial.

In the past a similar environment was experimented by Professor Asperti et al.[2][1], and it was based on the game Rogue. MicroRacer has a commonality with that environment, which is the use of a simplified state information. However, what makes MicroRacer different from the old project, is the fact that it makes use of continuous action space oppositely to the discrete used in the older one. This gives the possibility to use specific continuous action space RL-methods and to investigate their functioning.

From the tests conducted it has been verified that, thanks to MicroRacer simplified dynamics, the agent used can obtain good results with short training times, and it is also possible to investigate common DRL issues.

MicroRacer is currently under evaluation by the students of the Machine Learning course at the University of Bologna and a championship for the next academic year is planned.

MicroRacer is freely available to everyone and could be further improved as it is open source and easily accessible on GitHub at `https://github.com/asperti/MicroRacer`.

Appendix A

Appendix

	Completed episodes	Average episodic reward	Average episodic environment steps	Average speed
DDPG	2	-0.21	162	0.41
	66	4.35	458	0.30
	90	6.1	699	0.22
	32	2.55	297	0.38
	0	-0.36	173	0.40
DDPG2	2	-0.85	161	0.31
	27	1.67	374	0.26
	39	2.42	507	0.22
	21	1.08	230	0.33
	2	-0.29	180	0.36
TD3	75	5.11	442	0.33
	37	2.42	338	0.33
	7	0.28	471	0.16
	80	5.18	743	0.24
	72	4.77	565	0.25
SAC	74	5	829	0.23
	77	5.04	499	0.28
	79	5.23	515	0.29
	55	3.80	398	0.34
	60	4.01	486	0.31
DSAC	12	1.15	267	0.37
	26	1.91	289	0.35
	51	3.68	406	0.33
	24	2.23	316	0.37
	75	5.24	473	0.31
PPO	62	3.66	546	0.26
	31	1.66	791	0.17
	52	3.18	602	0.23
	0	-0.78	227	0.29
	41	2.54	601	0.19

Table A.1: Complete results of 100 evaluations episodes executed after each training process.

Bibliography

[1] Andrea Asperti, Daniele Cortesi, and Francesco Sovrano. "Crawling in Rogue's Dungeons with (Partitioned) A3C". In: *Machine Learning, Optimization, and Data Science - 4th International Conference, LOD 2018, Volterra, Italy, September 13-16, 2018, Revised Selected Papers*. Vol. 11331. Lecture Notes in Computer Science. Springer, 2018, pp. 264–275. DOI: 10.1007/978-3-030-13709-0_22. URL: https://doi.org/10.1007/978-3-030-13709-0_22.

[2] Andrea Asperti et al. "Crawling in Rogue's Dungeons With Deep Reinforcement Techniques". In: *IEEE Trans. Games* 12.2 (2020), pp. 177–186. DOI: 10.1109/TG.2019.2899159. URL: https://doi.org/10.1109/TG.2019.2899159.

[3] Bharathan Balaji et al. "DeepRacer: Educational Autonomous Racing Platform for Experimentation with Sim2Real Reinforcement Learning". In: *CoRR* abs/1911.01562 (2019). arXiv: 1911.01562. URL: http://arxiv.org/abs/1911.01562.

[4] Greg Brockman et al. "OpenAI Gym". In: *CoRR* abs/1606.01540 (2016). arXiv: 1606.01540. URL: http://arxiv.org/abs/1606.01540.

[5] Bingqing Chen et al. *Safe Autonomous Racing via Approximate Reachability on Ego-vision*. 2021. arXiv: 2110.07699 [cs.RO].

[6] "Deep Reinforcement Learning for Autonomous Driving". In: *CoRR* abs/1811.11329 (2018). Withdrawn. arXiv: 1811.11329. URL: http://arxiv.org/abs/1811.11329.

[7] Jingliang Duan et al. “Distributional Soft Actor-Critic: Off-Policy Reinforcement Learning for Addressing Value Estimation Errors”. In: *IEEE Transactions on Neural Networks and Learning Systems* (2021), pp. 1–15. DOI: 10.1109/TNNLS.2021.3082568.

[8] Scott Fujimoto, Herke van Hoof, and David Meger. “Addressing Function Approximation Error in Actor-Critic Methods”. In: *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*. Ed. by Jennifer G. Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, 2018, pp. 1582–1591. URL: http://proceedings.mlr.press/v80/fujimoto18a.html.

[9] Gianluca Galletti. “Deep reinforcement learning nell’ambiente pyTORCS”. MA thesis. University of Bologna, school of Science, 2021.

[10] Tuomas Haarnoja et al. “Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor”. In: *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*. Ed. by Jennifer G. Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, 2018, pp. 1856–1865. URL: http://proceedings.mlr.press/v80/haarnoja18b.html.

[11] James Herman et al. “Learn-to-Race: A Multimodal Control Environment for Autonomous Racing”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2021, pp. 9793–9802.

[12] Timothy P. Lillicrap et al. “Continuous control with deep reinforcement learning”. In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2016. URL: http://arxiv.org/abs/1509.02971.

[13] Daniele Loiacono, Luigi Cardamone, and Pier Luca Lanzi. "Simulated Car Racing Championship: Competition Software Manual". In: *CoRR* abs/1304.1672 (2013). arXiv: 1304.1672. URL: http://arxiv.org/abs/1304.1672.

[14] Daniele Loiacono et al. "The 2009 Simulated Car Racing Championship". In: *IEEE Trans. Comput. Intell. AI Games* 2.2 (2010), pp. 131–147. DOI: 10.1109/TCIAIG.2010.2050590. URL: https://doi.org/10.1109/TCIAIG.2010.2050590.

[15] Matthias Plappert et al. "Parameter Space Noise for Exploration". In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. URL: https://openreview.net/forum?id=ByBAl2eAZ.

[16] John Schulman et al. "Proximal Policy Optimization Algorithms". In: *CoRR* abs/1707.06347 (2017). arXiv: 1707.06347. URL: http://arxiv.org/abs/1707.06347.

[17] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: an introduction*. 2nd edition. Adaptive computation and machine learning series. Cambridge, MA, USA: The MIT Press, 2018. ISBN: 9780262039246.

[18] Christopher J. C. H. Watkins and Peter Dayan. "Technical Note Q-Learning". In: *Mach. Learn.* 8 (1992), pp. 279–292. DOI: 10.1007/BF00992698. URL: https://doi.org/10.1007/BF00992698.

Acknowledgements

I would like to express my gratitude to Professor Asperti for the patience, the trust, the continuous help and guidance during the process of developing and writing this thesis.

Thanks to my parents for the constant enormous help and sustain. All of this couldn't be possible without them.

Thanks to Virginia for the immense support both moral and physical during the last months and for believing in me even when i didn't.

I would also thank Matteo and Federica for all the years together. You are like a second family to me.