Reinforcement learning vs supervised learning

A comparison on DonkeyCar autonomous driving

Master's Thesis submitted to the
Faculty of Informatics of the *Università della Svizzera Italiana*
in partial fulfillment of the requirements for the degree of
Master of Science in Informatics
Major in Artificial intelligence

presented by
Giorgio Macauda

under the supervision of
Prof. Paolo Tonella
co-supervised by
PhD Matteo Biagiola

September 2022

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Giorgio Macauda
Lugano, 12 September 2022

To my beloved

Someone said . . .

Someone

Abstract

This is a very abstract abstract. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras

ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus, vehicula eu, lacus.

Acknowledgements

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras

ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus, vehicula eu, lacus.

Contents

Figures

Tables

Chapter 1

Introduction

Reinforcement Learning (RL) is a branch of machine learning that has proven to be a very general framework to learn sequential decision-making tasks that are generally modeled as Markov Decision Processes (MDP) [van Otterlo and Wiering, 2012] and without the need for labeled data. Reinforcement learning can operate in diverse situations as long as a clear reward can be applied. Under this framework, an RL agent interacts with an environment in discrete time steps and for each step, it observes the state of the environment and based on it, aims to take action that maximizes a given reward function. In recent years developments in RL have shown how it can deal with very high-complexity environments that can also change over time, first among all AlphaGo [Silver et al., 2016] is an algorithm that is capable of playing the game of Go, notoriously the most challenging of classic AI games due to its huge search space and difficulty in evaluating board positions and moves. RL is capable of impressive performances when it is constrained in simulated environments, but uptake in real-world problems has been much slower given the even higher complexity and unpredictability of real-world environments. In particular, in robotics, training RL agents through trial and error has proven tedious and has shown several limitations, i.e. they can be very expensive due to the exploration phase and therefore the process should be data-efficient, actuators and sensors can introduce varying amounts of delay and noise, many robotic systems have a stricter form of constraint in their movements in comparison to simulated environments and the reward function can be very sparse due to the scarcity of sensors and information about the state. Interesting results in real-world applications comes often from very controlled environments, for example, solving a Rubik's cube requires a limited set of actions, and the reset of the cube to the initial state is very simple. The objective of this thesis is to learn an RL self-driving DonkeyCar, a scale remote-controlled electric car, first in simulation and then replicate the same result in a very uncontrolled real-world environment. The state-of-the-art self-driving cars algorithms, created by Google with Waymo and Tesla, rely on Supervised Learning (SL) techniques such as Convolutional Neural Networks (CNNs) for image processing, feature detection, and extraction, and Recurrent Neural Networks (RNNs) for processing temporal data. Even though supervised learning is very successful in autonomous driving, it does not come at no cost, it requires huge labeled datasets that need to be consistently updated to face new scenarios. Furthermore, SL methods typically are used to generate predictions about the surroundings of the car and upon that decisions are taken and do not take into account that each decision influences future events, which in turn influence future decisions. In other words, they try to imitate data but do not have the consciousness of the real world. RL, on the other hand, allows learning a policy, thereby creating models able to make their own decisions, take actions, react and adapt based on the feedback they receive.

Since training an autonomous agent from raw images is expensive and has been proven unsuccessful [Viitala et al., 2020], we first investigate a few Representation Learning (ReL) techniques such as AutoEncoders and Variational AutoEncoders. ReL is a technique designed to extract abstract features from data and reduce their complexity. Furthermore, techniques for a smooth exploration of the environment such as state-dependent exploration are implemented. A reward function that has proven to work in both simulated and real-world environments has been designed even if the sensors are scarce. In particular, as a first step of the experiments, we trained successfully a simulated proof-of-work RL agent that autonomously drives by taking actions based on what a camera sensor, with which it is equipped, sees as unique information about the surroundings. As a second step, the model is successfully replicated in a real-world environment with an investigation of the best training strategy in terms of the starting point of the car which crucially defines the learning success.

Finally, a few unsuccessful experiments which need to be explored more are run in a Sim2Real procedure through the use of CycleGan [Zhu et al., 2017], where an agent trained in simulation is deployed in the real world and vice-versa, thus leading to cheaper training processes and more reliable benchmarking.

Chapter 2

Background

2.1 Reinforcement Learning

Reinforcement Learning (RL) is a branch of machine learning, alongside supervised learning and unsupervised learning, that defines a set of algorithms meant to learn how to act in a specific environment without the need of labeled data to learn from.

The algorithm defines the agent that learns a given task, for example, walking, driving and playing a game, by trial and error, while interacting with an environment which can be real or simulated. Whenever the agent makes a set of good actions it receives a positive reward, which makes such actions more likely in the future. State, action and reward are the most important concepts in RL. The state represents the current situation of the environment. If the agent is a humanoid robot and the task is walking, one possible state representation is the positions of its actuated joints. The action set or space in case of continuous domain, describes what the agent can do in a particular state. In the humanoid robot example above, the action space is a n-dimensional vector where each dimension represents the torque command to each of the n joint motors. Finally the reward is a measure of how good are the actions carried out by the agent.

The reward function, usually human-designed, assigns a score to the action taken by the agent. Every action that leads to a *good* state increases the score and vice-versa. As described in Figure 2.1 the agent interacts with the environment in discrete time steps. At time t it gets the current state s_t and the associated reward r_t then the action a_t is chosen from the set of available actions. After receiving the chosen action, the environment moves to a new state s_{t+1} and the reward r_{t+1} is given back to the agent. The total discounted reward (also known as return) to be maximized is:

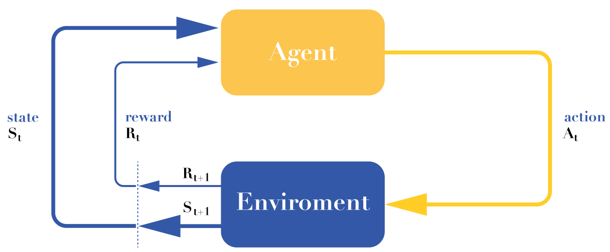


Figure 2.1. Basic reinforcement learning

$$R = \sum_{t=0}^{T} \gamma^t r_t \tag{2.1}$$

where T is the time horizon (eventually ∞), $\gamma \in [0, 1)$ is the discount factor which makes future rewards worth less than immediate rewards. The reward function is fundamental to the agent in order to learn and optimize a policy function π:

$$\pi : A \times S \to [0, 1] \qquad \pi(a, s) = \Pr(a_t = a \mid s_t = s) \tag{2.2}$$

The policy is a mapping that gives the probability of taking action a in state s. By following the policy the agent takes the action that maximizes the reward. However, the policy, especially during training, is not deterministic. This is due to one of the fundamental challenges in RL, i.e. the exploration-exploitation dilemma [Sutton and Barto, 2018]. Indeed, the agent needs to repeat the actions it already knows to be rewarding but, at the same time, it needs to explore the environment to discover actions that can lead to an even higher reward. The final goal of the algorithm is to learn a policy that maximizes the expected cumulative reward:

$$J(\pi) = \mathbb{E}_{\pi}[\sum_{t=0}^{T} \gamma^t r(s_t, a_t)] \tag{2.3}$$

The expectation term is added because both the policy and the environments are usually stochastic. There are multiple ways to learn the optimal policy $\pi^*(s)$ assuming the *State Transition probability matrix P* that describes the probability of moving from one state to any successor state is known. The first one is called *Value iteration*, which exploits the state value function $V^{\pi}(s)$ and the action value function $Q^{\pi}(s, a)$. The state value function V is the expected return starting from the state s and following the policy π:

$$V(s) = E_{\pi}[\sum_{t=0}^{T-1} \gamma^t r_t \mid s_t = s] \tag{2.4}$$

while the action value function Q is the expected return starting from the state s and taking action a by following the policy π :

$$Q(s, a) = E_{\pi}[\sum_{t=0}^{T-1} \gamma^t r_t \mid s_t = s, a_t = a] \tag{2.5}$$

There is an important relationship between the two functions 2.4 and 2.5, in fact they can be written in terms of each other:

$$V(s) = \sum_{a \in A} \pi(a \mid s) Q^{\pi}(s, a) \tag{2.6}$$

$$Q(s, a) = \sum_{s' \in S} P(s' \mid s, a)[r(s, a, s') + \gamma V(s')]. \tag{2.7}$$

where P is the state transition matrix that gives the probability of reaching the next state *s'* given the state s and action a and r is the reward function that returns the reward value associated with transitioning to the next state s' by taking the action a in state s.

In *Value Iteration*, the value function V is randomly initialized and the algorithm, illustrated in Listing 2.1, repeatedly updates the values of Q and V for each state until convergence. When value iteration terminates the functions Q and V are guaranteed to optimal.

```
Initialize V(s) to arbitrary values
Repeat
    for all s in S
        for all a in A
            Q(s,a) = E[r|s,a] + γ ∑_{s'∈S} P(s'|s,a)V(s')
        V(s) = max_a Q(s,a)
until V(s) converges
```

Listing 2.1. Value iteration pseudo code from Alpaydin [2014]

Finally the optimal policy π^* can be inferred from the optimal Q^* function with:

$$\pi^*(s) = argmax_a Q^*(s, a) \tag{2.8}$$

The optimal policy aims at choosing actions that maximizes the optimal Q function in that state.

Since the fundamental quantity for the agent is the policy, another way of training the agent is to learn a policy without extracting it from the action-value function Q. Therefore, the so-called *Policy Iteration* algorithm seeks to learn the policy directly by updating it at each step as shown in Listing 2.2

```
Initializea policy π' arbitrarily
Repeat
    π = π'
    Compute the values using π
        V_π = E[r | s, π(s)] + γ ∑_{s'∈S} P(s' | s, π(s))V_π(s'))
    Improve the policy at each state
        π'(s) = argmax_a(E[r | s, a] + γ ∑_{s'∈S} P(s' | s, a)V_π(s')))
until π = π'
```

Listing 2.2. Policy iteration pseudo code from Alpaydin [2014]

Policy iteration is also guaranteed to converge to the optimal policy and it often takes less iterations to converge than the value iteration algorithm.

A major problem arises when the *the State Transition Matrix* of the environment is not known to the agent or the number of possible states is too big to be stored in tables, as for example when the state is an image and/or the action space is continuous. Deep RL algorithms use Deep Neural Networks in order to approximate Q and V instead of storing them in tables. Indeed, DNNs can represent states and actions in a compact way thanks to their ability to generalize to unseen data.

Besides the quantity that needs to be learnt, i.e. the value functions or the policy, RL algorithms are also categorized by the way such quantities are updated. On-Policy methods evaluate and improve the same policy which is being used to select actions. Off-Policy methods can optimize a certain quantity (usually an action value function Q) with data coming from any policy. Such methods are typically more efficient than on-policy methods, as they can reuse already collected experience multiple times. In order to reuse previously gathered data, off-policy methods randomly sample training data from the past experience stored in buffers, generally called replay buffer or experience, instead of using the latest experience. The replay buffer contains a collection of experience tuples (s, a, r, s'), where each term is respectively the state, the action taken, the reward and the new state reached taking action a, collected by the driving policy.

2.2 OpenAI Gym interface

Gym is an open source library that defines a standard API to handle training and testing of RL agents, while providing a diverse collection of simulated environments. The environment is of primary importance to a RL algorithm since it defines the world of the agent in which the agent lives and operates.

The standard interface designed by Gym, makes it easier to interact with environments, both made available by Gym and externally developed. The Gym interface is simple and capable of representing general RL problems. The DonkeyCar environment, shown in Figure 2.2 and used in this piece of work, is an example of what is a custom environment. Gym let us define the action space of the car, which means all the operation it can performs in it such as steer and accelerate. Furthermore, it informs us about the state of the of the car in the environment so that we can perform all the necessary operation, for example, stopping the car when exceeding certain boundaries.

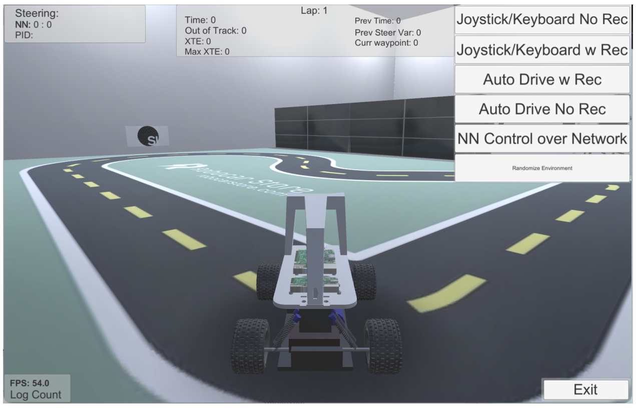


Figure 2.2. DonkeyCar environment implemented with Gym

The documentation provides a reference template shown in Listing 2.3 that describes what are the fundamental methods a Gym environment should implement to work properly. Any existing environment built with Gym implements the following methods:

```
class GymTemplate(gym.Env):
  def __init__(self):
    pass
  def step(action):
    pass
  def reset():
    pass
  def render(mode='human'):
    pass
  def close(self):
    pass
```

Listing 2.3. "Gym template"

- **init:** every environment should extend the gym.Env class and override the variables *observation_space* and *action_space* specifying the type of observations and actions. For example, the observations can be images or continuous vectors as well as actions can be continuous or discrete. According the the Gym notation the state of the environment is called 'observation' since, in general, the state is not fully observable. Therefore, the observation is the observable part of the state, i.e. what the agent can perceive with its sensors.

- **step:** this method is the primary interface between environment and agent, it takes as input the action to be carried out in the environment and return information (observation, reward, done) about the current state such as, the next observation resulting from the action executed in the environment, the corresponding reward value and a boolean flag signaling the end of the 'episode'. Indeed, a Gym environment models episodic RL tasks, i.e. finite tasks that terminate when certain conditions hold.
- **reset:** this method resets the environment to its initial values returning the initial observation. This method is called to initialize the environment and at the end of each episode, such that the agent starts each episode always with a clean state.
- **render:** this method renders the environment when a parameter *mode='human'* is passed.
- **close:** this method performs any necessary cleanup before closing the environment.

Besides the API interface, Gym provides a set of wrappers to modify an existing environment without having to change its underlying code directly. The three main things a wrapper does are:

- Transform actions before executing them to the base environment
- Transform observations that are returned by the base environment
- Transform rewards that are returned by the base environment

The given set of wrappers to reach any of the aforementioned goal includes: *ActionWrapper*, *ObservationWrapper*, *RewardWrapper*. Furthermore, custom wrappers can be implemented by inheriting from the *Wrapper* class.

2.3 Soft Actor Critic - SAC

The soft actor critic algorithm [Haarnoja et al., 2018] is a state-of-the-art RL algorithm designed to outperform prior on-policy and off-policy methods in a range of continuous control benchmark tasks.

It aims to both increase the sample efficiency and the robustness of the policy at test time. A poor sample efficiency is typical of on-policy RL methods since they require new sample to be collected for each gradient step. In order to improve the sample efficiency, SAC adopts an off-policy approach where the experience is stored in a replay buffer such that it can be reused multiple times during training. Moreover, SAC is based on the maximum entropy framework which maximizes both the expected return and the entropy of the policy. This objective is expressed in the following equation:

$$J(\pi) = \mathbb{E}_{\pi}[\sum_{t=0}^{T} \gamma^t r(s_t, a_t) + \alpha H(\pi(\cdot \mid s_t))] \tag{2.9}$$

where α is a temperature parameter that weighs the entropy term and thus controls the policy stochasticity, Maximizing both the expected reward and the entropy of the policy is beneficial to obtain policies that are robust w.r.t. unexpected situations at testing time. Moreover, training a stochastic policy encourages a wide exploration of the environment, promoting diverse behaviors of the agent.

2.4 Generative Adversarial Networks - GAN

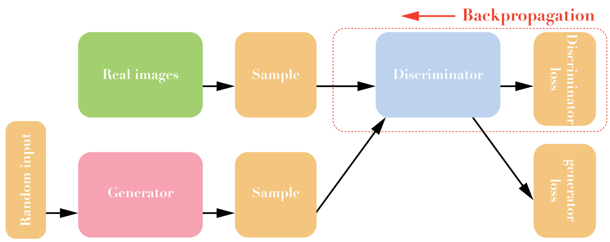


Figure 2.3. GAN diagram

Generative Adversarial Network is a framework introduced by Goodfellow et al. [2014] for training generative models in an unsupervised fashion. GANs can be used, for example, to generate visual paragraph [Liang et al., 2017], realistic text [Zhang et al., 2017], photographs of human faces [Karras et al., 2017] and Image-to-Image translation [Isola et al., 2017]. The learning process involves two neural networks that are trained in an adversarial way, i.e. with a contrasting objective. Indeed, as shown in Figure 2.3, the generator G generates inputs (e.g images) starting from random noise and the discriminator D needs to distinguish whether such inputs belong to the original dataset or not. GANs fall under the branch of unsupervised learning since the training process does not need labeled data as the generator is guided by the discriminator in order to generate inputs that resemble those of the training dataset. The discriminator D is trained to maximize the probability of returning the correct label when given both training examples and examples generated by the generator G. At the same time the objective of generator G is to minimize the following loss function:

$$L_G = log(1 - D(G(z))) \tag{2.10}$$

where z is the random noise vector, i.e. the latent vector, $D(G(z))$ is the probability that the generated example $G(z)$ comes from the training dataset (represented by the distribution p_x where X is the training dataset and p_x represents all the possible images that can be in X), which means that $1 - D(G(z))$ is the probability that $G(z)$ does not come from from p_x. Indeed, the objective of the generator G is to generate examples that are indistinguishable from the training examples for the discriminator D.

In particular, in order to learn the generator's distribution p_g over the training dataset X such that $p_g \approx p_x$, the authors define a distribution over latent vectors $p_z(z)$, which is mapped into the data space with the generator $G(z, \theta_g)$. Moreover, the discriminator $D(x; \theta_d)$, with $x \sim p_g$, outputs a single value which estimates the probability of x coming from the distribution p_x rather than p_g. D and G are both differentiable functions represented by a neural network with parameters θ_d and θ_g respectively.

In other words, the discriminator and the generator play a minimax game to optimize the function $V(G, D)$:

$$min_G max_D V(G, D) = \mathbb{E}_{x \sim \rho_{data}(x)}[log D(x)] + \mathbb{E}_{z \sim \rho_z(z)}[log(1 - D(G(z)))] \tag{2.11}$$

2.5 CycleGAN

Image-To-Image translation is a complex task where the goal is to transform an image from one domain to another and vice-versa, as shown in Figure 2.4. Prior papers have been presented to translate images, however they often require paired training examples between the domains [Sangkloy et al. [2016], Karacan et al. [2016]]. Such paired datasets can be very expensive or even impossible to gather, as in the case of object transfiguration ($horse \leftrightarrows zebra$).

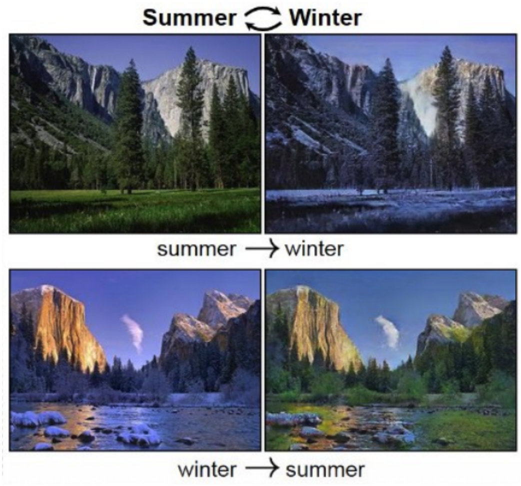


Figure 2.4. Image-to-image translation example [Zhu et al., 2017].

Cycle-Consistent Adversarial Networks from Zhu et al. [2017] (CycleGAN), aims to solve this problem in an unsupervised fashion. The main goal is to learn, using an adversarial loss, a mapping $G : X \rightarrow Y$, where X and Y are two sets representing different domains, such that the image $G(x)$ with $x \in X$ is indistinguishable from an image $y \in Y$. Since the mapping is highly undercostrained, an inverse mapping $F : Y \rightarrow X$ is introduced, together with a cycle-consistency loss to enforce $F(G(x)) \approx x$ and vice-versa. To accomplish the goal two discriminator D_X and D_Y are provided. D_X tries to distinguish between examples coming from one domain (i.e. represented by the distribution ρ_x) and their translations $F(Y)$ and vice-versa for D_Y. The full objective 2.12 includes the adversarial losses and the cycle-consistency loss to encourage a consistent translation from one domain to the other:

$$L(G,F,D_X,D_Y) = L_{GAN}(G,D_Y,X,Y) + L_{GAN}(F,D_X,Y,X) + \lambda L_{cyc}(G,F) \tag{2.12}$$

where the loss $L_{GAN}(G,D_Y,X,Y)$ and $L_{GAN}(F,D_X,Y,X)$ can be constructed from Equation in 2.11 and the following is the cycle-consistency loss:

$$L_{cyc}(G,F) = \mathbb{E}_{x\sim\rho_x}[\|F(G(X)) - x\|_1] + \mathbb{E}_{y\sim\rho_y}[\|G(F(y)) - y\|_1] \tag{2.13}$$

where λ is a temperature parameter to define the importance of such loss in Equation 2.12 and $\|\cdot\|_1$ is the L1 norm, i.e. a measure of the distance between vectors.

2.6 AutoEncoder and Variational AutoEncoder

2.6.1 AutoEncoder

AutoEncoders (AEs) are artificial neural networks that fall under the branch of unsupervised learning since they learn efficient encoding into a latent space without the need of a labeled dataset. They are generally used for several purposes, for example, dimensionality reduction, image compression, image denoising, image generation, feature extraction and sentence generation [Hinton and Salakhutdinov [2006], Cheng et al. [2018], Gondara [2016], Hou et al. [2017], Liu and Liu [2019]].

Taking as example the case of image dimensionality reduction, an AE is composed of two main parts, an encoder E and a decoder D.

$$E(\phi): X \to Z \qquad D(\theta): Z \to X' \tag{2.14}$$

where $X = \mathbb{R}^{mxn}$ and $Z = \mathbb{R}^k$ for some m, n, k and $k \ll mxn$ to reach the goal of dimensionality reduction. Both encoder and decoder are parametrized functions, with parameters ϕ and θ respectively, As shown in Figure 2.5, the main goal of the encoder is to learn a mapping of each observation of the dataset $x \in X$ into a latent space of smaller dimensionality. Since a label is not available, in order to measure the quality of the embedded image into the latent space, the decoder is used to reconstruct the image and then compute the reconstruction loss.

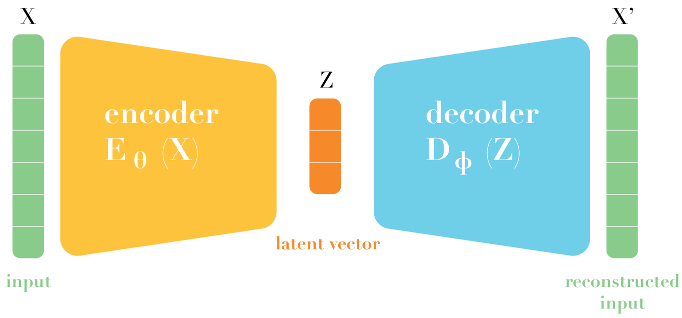


Figure 2.5. AE diagram

In other words, the encoder maps an image $x \in X$ into the latent space producing $z = E_\phi(x)$ with $z \in Z$; then z is reconstructed by the decoder to bring it back to the original space $x' = D_\theta(z)$ with $x' \in X'$. Finally, x' can be used as a label with any distance measure $d(x, x')$. Thus the loss to be minimized is computed as follows:

$$L(\theta, \phi) = d(x_i, D_\theta(E_\phi(x_i))) \tag{2.15}$$

2.6.2 Variational AutoEncoder

Variational AutoEncoders (VAEs) addresses the problem of *sparse localization* of data point into the latent space thus providing a more powerful generative capability then AEs.

As shown in Figure 2.6 only a small change with respect to AEs is introduced, i.e. the encoder instead of mapping samples directly into the latent space it encodes a single input as a distribution (usually a normal distribution) over the latent space. Then the concrete latent vector z is produced by sampling such distribution.

Specifically, the encoder, starting from an image $x \in X$, produces the gaussian parameters $[\mu_x, \sigma_x] = E_\phi(x)$, then z is sampled from a normal distribution $z \sim \mathcal{N}(\mu_x, \sigma_x)$. Consequently, the decoder brings it back to the original space $x' = D_\theta(z)$ with $x' \in X'$. Finally, x' can be used

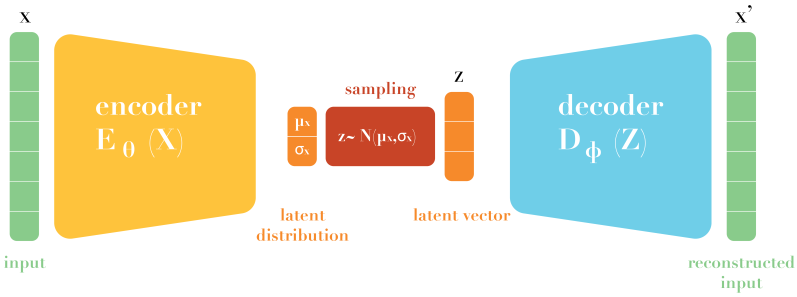


Figure 2.6. VAE diagram

as a label with any distance measure $d(x, x')$. Thus the loss to be minimized is computed as follows:

$$L(\theta, \phi) = d(x_i, D_\theta(E_\phi(x_i))) + KL[\mathcal{N}(\mu_x, \sigma_x), \mathcal{N}(0,1)] \tag{2.16}$$

where the first term is equivalent to the loss function in Equation 2.15 and the KL term is the Kullback-Leibler divergence, which is a measure of how a probability distribution is different from another. The KL divergence acts as a regularization term by enforcing predicted distributions to be close to the normal distribution with mean 0 and standard deviation 1, giving to the latent space two main properties, i.e. continuity (close points in the latent space should be close also when decoded) and completeness (any point sampled from the latent space should always be meaningful once decoded).

2.7 DonkeyCar

DonkeyCar, shown in Figure 2.7, is an open source DIY platform providing software and hardware tools for the development of self-driving car algorithms. The basic car is a simple remote controlled electric car that can be 3D printed or bought as a kit for an affordable price. The car can be customized with additional sensors as LIDARs and IMUs to provide more information about the surroundings of the car during driving.

In particular, the car used for the purposes of this thesis, is a basic donkey car equipped with an 8-megapixel IMX219 sensor that features an 160 degree field of view. It is capable of capturing images with a resolution of 3280x2464 and video recording up to a resolution of 1080p at 30 frames per seconds. In order to process all the information coming from the camera, control the motors and run the self-driving car software the car is equipped with an NVIDIA Jetson Nano microcontroller. The power comes from a LiPO battery of 11.1V and 2200mAh that runs the

Figure 2.7. Assembled DonkeyCar

electric motor and the micro-controller. Additionally, to expand the operational life of the car, a power-bank can be added to exclusively power the micro-controller, while the LiPO battery is dedicated at powering the engine. A DonkeyCar can be remotely controlled either with a joystick or directly by the software.

Chapter 3

Related works

When training a Reinforcement Learning model several problems arise, especially when the learning process moves from simulated environments to the real world. In this section, a few useful techniques for the purposes of this thesis, are presented.

3.1 State Representation Learning

Reinforcement Learning is a very general method for learning sequential decision-making tasks. On the other hand, Deep Learning has become in recent years the best set of algorithms capable of Representation Learning (ReL), i.e. a class of algorithms that are designed to extract abstract features from data. A mix of the two provides a particularly powerful framework for learning state representation, especially when dealing with real-world environments that tend to be much more complex and unpredictable than simulated environments. In particular, State Representation Learning (SRL) is a specific type of ReL where extracted features are in low dimension, evolve over time, and are affected by agents' actions. The low dimensionality allows easier interpretation by humans, it mitigates the curse of dimensionality and it speeds up the policy learning process. Thus, SRL is well suited for Deep Reinforcement Learning applications. Lesort et al. [2018] presented a complete survey that covers the state-of-the-art on SRL. Feature extraction is a set of algorithms whose objective, as the name suggests, is to decompose a particular data point into smaller identifiable components that are useful for the learning task at hand. For example, in a dataset composed of images collected by the DonkeyCar, in the context of autonomous driving, a set of features that can compose each picture can be the curve's direction and the curve's angle. Training a neural network to learn those features may be accomplished by compressing the image into a smaller vector, discarding all the unnecessary information that is not relevant for the learning task where each dimension would represent a feature like the ones just described. However, a feature not necessarily describes a human interpretable aspect of the data, rather it can even lack semantic meaning. In particular, SRL techniques exploit the time steps, actions, and eventually rewards, to transform observations into representative states, a low dimensionality vector that contains the most relevant features to learn a particular policy. The better the policy or the speed with which it is learned, the more the features extracted are significant to the model. In particular, the authors described and compared methods that do not require explicit supervision as required by the context such as AEs, VAEs, principal component analysis [Curran et al., 2015] and Position-Velocity Encoders [Jonschkowski et al., 2017]. Based on our necessity of decoupling the state representation learning from the RL algorithm to reduce the sample complexity and the evidence of other pieces of work described later on, we adopt the AEs/VAEs family to learn the representation of our states.

3.2 Improving sample efficiency

To define the state of the environment in our experiments, we use a camera as described in Section 2.7. However, training a model from high-dimensional images with reinforcement learning is difficult, in the previous Section 3.1 we described a general framework to mitigate those difficulties. In this section, we present a specific method that is used for this thesis' purposes.

Deep convolutional encoders can learn a good representation even though they generally require large amounts of training data. Using off-policy methods and adding an auxiliary task, i.e. an additional cost-function that an RL agent can predict and observe from the environment, with an unsupervised objective can naturally improve sample efficiency and add stability in optimization. In particular, Yarats et al. [2019] propose a simple and effective autoencoder-based off-policy method that can be trained end-to-end. Their main focus is on finding the optimal way of training an RL agent using SRL.

The authors, in their experiments, selected an AutoEncoder (AE) as an SRL technique which is composed of a convolutional encoder that maps an image observation to a low dimensional latent space and a deconvolutional decoder that reconstruct the latent vector back to the original image. The objective is to minimize the image reconstruction loss avoiding task-dependent losses. Then the SAC algorithm is used to learn some tasks directly from the latent space learned by the encoder, thus reducing the sample complexity.

In practice, they experimented with two ways of training the agent. In the first one the AE is pre-trained offline with the states/observations collected from the environment. Then, when training the agent online, both the encoder and the agent are updated but independently of each other. In other words, the agent is trained with transitions coming from the replay buffer, while the AE only uses the states in the buffer to update its latent representation. Moreover, the authors investigated different update frequencies of the AE. In the second training strategy, the setting is equivalent to the first one except that the AE is trained according to the task the agent is solving. The idea is that the representations that the AE learns should be related to the RL objective such that the learned features can be useful for the task at hand. However, the results obtained by the authors show that updating the AE with the agent gradients hurts the performance of the agent since the AE is shared between the actor and the critic. Furthermore, the results also show that updating the AE at the end of each episode and independently from the agent, results in the best performance in a wide range of tasks. On the other hand, using a pre-trained AE without updating it when training the agent shows a comparable performance and we use such a strategy to train the real agent. Indeed, updating the AE at the end of each episode would require an additional computation that would increase the training time when learning the policy in the real world.

3.3 Smooth exploration

When moving an RL algorithm from a simulated environment to the real world, the unstructured step-based exploration, i.e. without taking into consideration the underlying hardware, often leads to unstable motion patterns that may result in poor exploration, longer training time, and even damage to the robot the agent is controlling. Raffin et al. [2022] handles such an issue by adding a state-dependent exploration (SDE) to current Deep Reinforcement Learning algorithms. In most RL algorithms the standard way of exploring the action space is to sample a noise vector from a Gaussian distribution, independently of the environment and the agent, which is then added to the agent predicted action. SDE replaces the sampled noise with a state-dependent exploration function. This results in smoother exploration and less variance. Indeed, the gaussian noise is added to the predicted action according to the state the action is taken on. This way, in a given episode, given the same state, the action is the same, reducing the variance with which the agent controls the robot. The standard deviation of the gaussian noise is initialized randomly and then updated during training with the gradient of the policy, to be adapted to the task at hand. Moreover, the state-dependent exploration idea is further improved by sampling the noise every n steps instead of every episode and by taking as input other features other than the state. Such changes address some of the issues of the state-dependent exploration (e.g. long episodes) and further increase the smoothness and performance of the exploration. They are all desired properties for the autonomous driving task treated in this thesis where a state-independent noise vector may lead to fatal episodes too quickly, not letting the agent learns or even damage the robot in case of too shaky motions.

3.4 Learning to Drive - L2D

Viitala et al. [2020] propose a framework to train a RL agent to control a real self-driving car. In particular, they use a DonkeyCar, which we introduced in Section 2.7. To train the agent in the real world the authors adopt different training strategies, i.e. learning from pixels, using a state representation learning approach, and using a model-based RL approach. They experimented with different physical tracks and their results show that learning directly from pixels is inefficient, ineffective, and, ultimately, does not lead to a policy that can drive the physical car. On the other hand, the state representation learning approach and the model-based RL approach show comparable performance. In this thesis, we took inspiration from this work to train an RL agent in the real world. Similarly to Viitala et al. [2020], we follow the state representation learning approach by pre-training a Variational AE. Differently, we used a printed track that is larger than the tape-made tracks used by Viitala et al. [2020] (i.e. 7 meters vs 11 meters measured by our track). Moreover, we experimented with different reset strategies and investigated the use of sim2real methods for transferring the policy from real to simulated and vice-versa.

Chapter 4

Experimental setup

4.1 Track and Gym Environment

In our real-world experiments, we used the DIYRobocars Standard Track taken from the Robocar Store[1] and shown in Figure 4.1. The track measured is about 11 meters long, i.e. ≈60% bigger than the track used by Viitala et al. [2020].

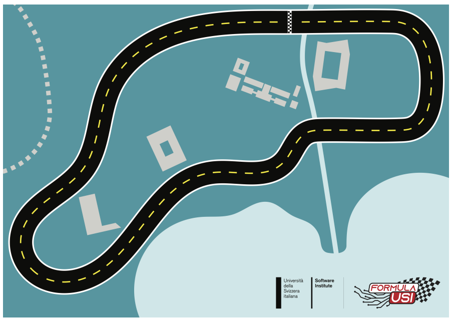


Figure 4.1. Real USI track TODO

Figure 4.2. Simulated USI track

For our experiments in the simulated environment, we used the virtual replica of such track, which was created in Unity [2] by a previous thesis in the lab. Additionally, we added a starting line in the simulated track in order to record the number of times the car completes a lap. Such information is important to understand the progress of the RL agent during training. Both tracks are single-lane since the DonkeyCar occupies 50% of the track when it is placed in the middle of the yellow dashed line. However, the track is an interesting testbed for RL and self-driving in general since it has a mixture of sectors that are easy to drive and curves that are challenging to take.

Regarding the Gym environment, we built upon the codebase of Raffin [2020]. Specifically, the *observation space* of both the real and simulated agent consists of RGB images of size 320 × 240 collected at 20*Hz* (i.e. 20 frames per second).

[1] https://www.robocarstore.com
[2] https://www.unity.com

The *action space* is composed of two continuous actions, i.e. steering and throttle, both varying in the interval $[-1, 1]$. Indeed, the action space must be symmetric since most RL algorithms use a Gaussian distribution (with mean 0 and standard deviation of 1) for exploration of continuous action spaces and a non-symmetric action space would harm learning [3]. Consequently, the throttle is rescaled to take values in the interval $[0, 1]$. However, for simplicity and for speeding up learning especially in the real world, we kept the throttle constant throughout training. Furthermore, In order to obtain a smooth control of the car, the change in steering angle is *constrained* by keeping a history of steering angles at previous time steps. This ensures that the difference between steering angles in two consecutive time steps stays within a given range Raffin [2020].

Since we use an encoder to represent the images in a lower dimension, we used a custom Gym wrapper to convert the raw images coming from the simulator or the real world into the corresponding latent space. Moreover, we stack four consecutive frames, i.e. converted into the corresponding latent space representations, such that the agent can perceive the motion of the car by taking an action every four frames Mnih et al. [2015].

Regarding the *reset* of the environment we used two different modalities in simulation and in the real world. In simulation, the DonkeyCar simulator provides the *cross track error* (XTE) measurement, which is defined as the distance between the center of the mass of the car and the center of the lane. Such metric varies in the interval $[-2, 2]$ when the car is *fully* on-track, i.e. its center of mass is in the middle of either white lines; therefore, when the absolute value of the XTE is outside of such interval the car can be considered off-track. In particular, we used the boundary value of 3, which corresponds to having the car with all four wheels off-track; in such case the episode terminates unsuccessfully. On the other hand, in the real world the XTE is not available. Therefore, we resort to a manual stopping strategy to terminate the episode, by remotely signaling the agent to stop controlling the car.

Finally, in real world, we established a success criterion to deem a certain episode successful. In particular, we chose to stop an episode when the number of timesteps reach the threshold level of 1000. Such threshold corresponds to approximately two laps around the track or alternatively 50 seconds given that the frame rate is $20Hz$. Indeed, the number of laps carried out by a trained RL agent depends on how *smooth* is the control, since steering actually slows down the car. In simulation, instead, such criterion is set to a completed a lap, and also in this case the number of steps carried out in a lap may vary based on the smoothness of the motion. However, the limit of 1000 steps is kept also in simulation.

4.2 Dataset

In order to train the encoders for the simulated and the real world, we collected two different datasets by driving the car in the respective environment. Examples of collected images are respectively shown in Figure 4.3 and in Figure 4.4.

In both environments we collected a dataset of $\approx 10k$ images which correspond to ≈ 10 minutes of driving at $20Hz$. We payed extra care when collecting the images in the real world by ensuring that the lighting conditions were consistent in all images; such conditions were also kept during training of the RL agent. Collecting the dataset for training the encoders does not require a good quality of driving, since labels are not recorded. However, it is important to capture all the sectors of the track such that the RL agent can extensively explore the track

[3] https://stable-baselines.readthedocs.io

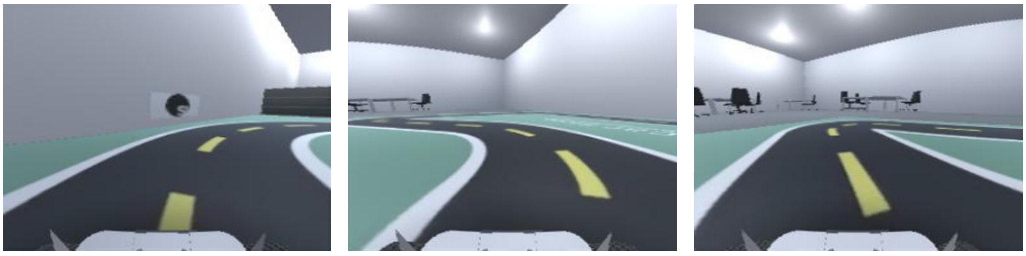


Figure 4.3. Images extracted from the simulated dataset

Figure 4.4. Images extracted from the real dataset

during training and always have a good representation of the observation it will encounter. Indeed, the pretrained encoder will not be updated during the online training of the agent.

Before the pretraining phase of the encoder we applied a preprocessing phase of the images. In particular, we cropped the top 80 pixels from each image and resized it to 160×80. Cropping is useful to remove the part of the image that is not relevant to driving, while downscaling the image has the advantage of saving computation time when training the encoder. Figure 4.5 and Figure 4.6 show samples of cropped and resized images that are passed as input to the encoder during training. As we can see, resizing down to 160×80 does not degrade significantly the quality of the images. Moreover, the real images look like they have undergone more cropping than their simulated counterpart. The reason is that the position of the camera of the car is slightly different between simulation and real. Indeed, the *simulated camera* is tilted upwards w.r.t. *real camera* and this gives the impression that in the real images more pixels have been cropped.

Matteo: ►Le immagini in figura 4.5 non sono le corrispondenti immagini croppate e resized di figura 4.3. Lo stesso vale per le immagini reali. Uniformerei queste cose.◄

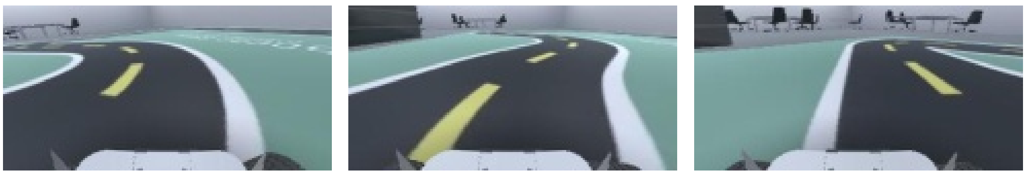


Figure 4.5. Examples of cropped simulated images

Finally such images collected in the simulator and in the real world are also useful for training the CycleGAN architecture Zhu et al. [2017]. In order to save computation time the dataset we provided to the CycleGAN training process is smaller, i.e. $\approx 5k$ images for each domain, i.e.

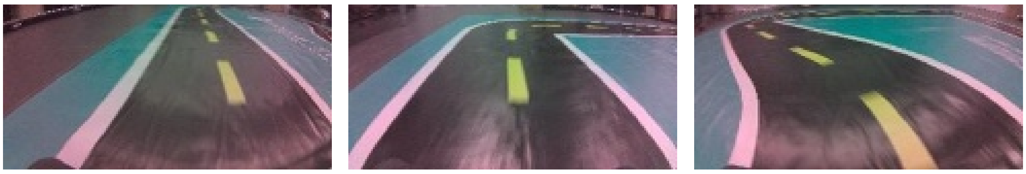


Figure 4.6. Examples of cropped real images

simulated and real, since previous work shows that such size is adequate to represent the track we are considering Stocco et al. [2022].

4.3 Training

To train the RL agent we adopted the SAC algorithm due to its sample efficiency and the ability to reuse the collected experience Haarnoja et al. [2018]. Such abilities are paramount when training an agent in the real world. We used the hyperparameters provided by Raffin [2020] for the task of driving. In particular, we chose a replay buffer of size 300*k* (i.e. the number of transitions that can be stored) and a multilayer perceptron with two layers of 64 units each both for the actor and the critic. Moreover, training is carried out at the end of each episode for 64 gradient steps. Matteo: ►Mi sembra che nel reale la policy sia piu' grande, forse sono 4 layer da 64 unita'. Giorgio check.◄ Thanks to the dimensionality reduction step, the transitions in the replay buffer contain latent vectors rather than entire images; as a consequence training is relatively fast and it can be carried out in machine without GPU.

In simulation, the training is performed using a client-server architecture where both client and server run on the same machine. The server, i.e. the Donkey simulator, provides the frames captured by the simulated camera at each timestep together with the *telemetry* of the car, i.e. its position, velocity and XTE. The client, i.e. the learning component, receives the frames and the telemetry and it is tasked to make a control decision, i.e. it has to return to the server the action to undertake on the car. Once the server receives the action, the simulator applies it on the car and the cycle continues. The decision to start or stop an episode is delegated to the client, which according to the telemetry received by the simulator at each step, decides when to reset the environment. When an episode terminates, the training procedure starts. Afterwards the simulation is stopped for the entire duration of training and it is resumed as soon as the training finishes.

Also In the real environment we used a client-server architecture for training. However, client and server run on different machines. This is because the Donkey microcontroller has limited computation capabilities and, most importantly, a limited battery. Indeed, carrying out the RL agent training on the Donkey would quickly deplete the battery and slow down training due to frequent battery recharge and replacement. Moreover, having two dedicated machines also allows a human operator to remotely stop an episode, appropriately reset the car to its initial position and remotely restart it.

In particular, we used the MQ Telemetry Transport (MQTT) messaging protocol, which is one of the most used in the internet of things domain. The protocol defines all the rules that determine how devices exchange messages over the internet, i.b. by writing (publish) and reading (subscribe) data. The sender (publisher) and the receiver (subscriber) communicate on message channels called *topics* and are decoupled from each other. The protocol also involves the presence of a *broker* that has access to the incoming messages and distributes them correctly to the subscribers of a certain topic. Specifically, we used the *HiveMQ* broker which allows the connection of up to 100 clients free of charge. The topics defined to manage the communication between the DonkeyCar and the client machine are as follows:

- **Stop car:**. (Client - Publisher, Donkey - Subscriber). When the client machine publishes a message on this topic the RL agent on the Donkey stops controlling the car, i.e. the episode must terminate. Practically the human operator presses the space bar on the keyboard which triggers the signal;
- **Replay buffer:** (Donkey - Publisher, Client - Subscriber). Once an episode terminates, the Donkey publishes on this topic all the transitions collected during the episode. The client machine receives such transitions and stores them in the replay buffer which is used to train the policy. The transitions only contain the latent space representations of the captured images in the real world; indeed, the encoder runs on the Donkey and transforms each frame into its corresponding latent space representation in order to be processed by the policy. This way the transitions exchange is quicker than sending raw images;
- **Replay buffer received:** (Client - Publisher, Donkey - Subscriber). The client uses this topic to acknowledge the Donkey that it has received the transitions. This topic is useful when the replay buffer of the episode gets longer, hence the Donkey sends the data in chunks and before sending the next chunk it needs to know that the client has finished reading the previous in order to avoid overlapping.
- **Parameters:** (Client - Publisher, Donkey - Subscriber). The client has a copy of the parameters of the policy used by the Donkey. Once the client receives the transitions the training starts. During training the human operator can retrieve the car and position it back on track. Once the training is complete, the client machine publishes the updated policy parameters and the Donkey updates its copy of the policy parameters;
- **Start episode:** (Client - Publisher, Donkey - Subscriber). The client uses this topic to acknowledge the Donkey that a new episode must start. Practically the human operator presses the enter key on the keyboard which triggers the signal;

Chapter 5

Experiments

5.1 AE vs VAE

As described in previous sections, our main goal is to create an end-to-end RL algorithm composed of an encoder followed by SAC. To determine whether to use an AutoEncoder or a Variational AutoEncoder we investigate if the stochasticity of VAEs can help in learning a good representation of the actual state.

In order to chose which architecture to use, we train an AE where the encoder is composed of 3 sequential convolutional linear layers interposed by a ReLU function and on output layer of size to be defined, similarly the decoder is composed of 3 deconvolutional layers. The VAE, instead, is composed of 4 convolutional/deconvolutional layers. The detailed architecture can be found in APPENDIX A. Each network has been trained for 50 epochs on our training set with an early stopping on the fifth contiguous epoch in with no improvement on the validation loss.

The latent size (z_size) must be carefully chosen such that the latent space is able to represent all the features extracted from the images and consequently produce high quality reconstruction. In particular, our tests plan to use 32 and 64 dimensional latent spaces.

To further improve the generalization of our encoders and the robustness of our learned representation, image augmentation is a suitable technique. In particular, we consider several augmentation methods such as Gaussian and motion blurring, contrast normalization, additive Gaussian noise, sharpening and coarse dropout. In training each images can be randomly augmented by some of the aforementioned transformation.

Each of the architecture with the different z_size and with augmentation enabled or disabled is then trained three time for each of the training set (simulated and real), and the results are averaged to increase to reliability.

After the training, each model is evaluated on the test set and the resulting reconstruction loss (MSE) are averaged to identify which is the absolute best encoder.

The results reported in Tables 5.1-5.4 clearly shown how, in all cases, each encoder obtains lower reconstruction loss mean when augmentation is disabled since its activation causes early stopping in all tested case. A further contribution to the reduction of the loss is given by a bigger latent space, in fact, in all cases, the encoder with a latent space of 64 dimension performs better. Finally, our VAEs outperform significantly all the AEs, that is why we will use them to carry out all the next experiments, both in real world and in simulation. As described above, those pre-trained VAEs will remain unchanged for the entire duration of the RL agent training that follow.

In Figures 5.1 and 5.2 is shown an example of what are the capabilities of the chosen VAEs in terms of reconstruction. From now on we will refer to the VAE trained on the dataset of pictures collected in the simulator with the name *simulated VAE* and to the one trained on pictures collected in the real world with name *real VAE* for simplicity.

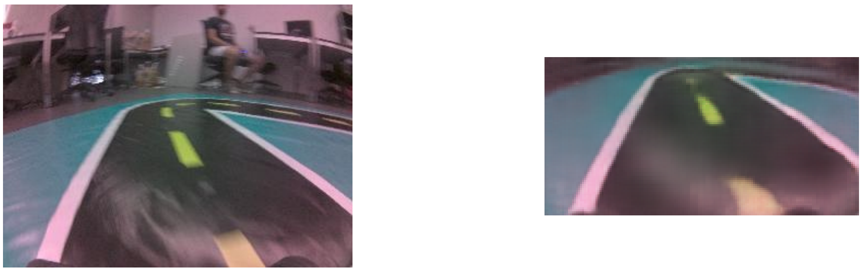


Figure 5.1. On the left an image from the real world as seen by the DonkeyCar camera, on the right the encoded and reconstructed image by the chosen VAE with a reconstruction loss of 112.

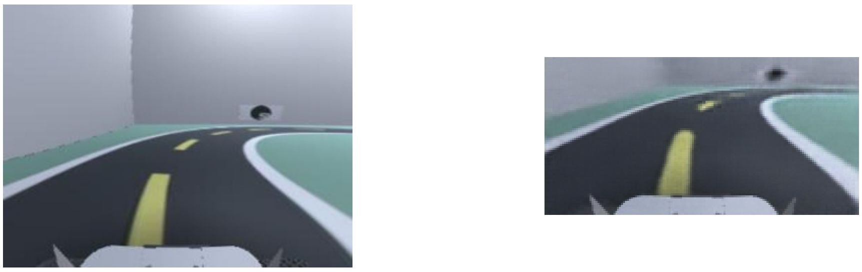


Figure 5.2. On the left an image from the simulator as seen by the DonkeyCar camera, on the right the encoded and reconstructed image by the chosen VAE with a reconstruction loss of 17

5.2 RL algorithm

5.2.1 Reward function

Designing a reward function that can work in both simulated and real environments is not trivial given the fundamental differences between them. In simulation, for example, the environment can provide supervision and useful information such as the position of the car and the speed. In our real setup, instead, the DonkeyCar can only leverage information coming through the camera frames. The reward function designed to work in simulation consists of four parts. The first one is a single point gained by the agent for every step made, intending to improve the length of the path as much as possible. Secondly, a throttle reward term increases the reward by a value proportional to the throttle to encourage the agent to drive as fast as possible. Moreover, a cross-track error penalty is proportional to the distance of the car from the center of the roadway that disincentives the agent as soon as it moves away from the center. Finally, as soon as the agent crashes or exceeds the maximum cross-track error allowed a big penalty is

Z_SIZE	AUGMENTATION	MEAN	STD	MAX	MIN
32	False	121.54	102.42	795.44	45.61
	True	164.57	95.51	783.03	65.13
64	False	103.54	79.14	588.14	40.84
	True	137.24	74,02	611,81	63,05

Table 5.1. AE trained in simulation - reconstruction loss

Z_SIZE	AUGMENTATION	MEAN	STD	MAX	MIN
32	False	377.07	87.53	756.7	239.46
	True	493.84	99.40	807.67	289.99
64	False	311.1	78.5	695.65	177.77
	True	411.37	77.30	647.68	241.87

Table 5.2. AE trained in real world - reconstruction loss

Z_SIZE	AUGMENTATION	MEAN	STD	MAX	MIN
32	False	59.1	60.41	620.93	18.88
	True	116.31	71.11	771.88	51.10
64	False	45.15	43.49	480.22	14.34
	True	112.17	59.79	573.19	54.28

Table 5.3. VAE trained in simulation - reconstruction loss

Z_SIZE	AUGMENTATION	MEAN	STD	MAX	MIN
32	False	227.4	44.74	418.7	140.12
	True	263.87	52.29	478.26	172.70
64	False	184.56	36.86	347.59	96.7
	True	230.66	42.24	402.67	156.61

Table 5.4. VAE trained in real world - reconstruction loss

given. Thus, the reward function to be maximized is composed as follows:

$$r_t = 1 + throttle\_reward + cte\_penalty + \begin{cases} if\,done & crash\_error \\ else & 0 \end{cases} \tag{5.1}$$

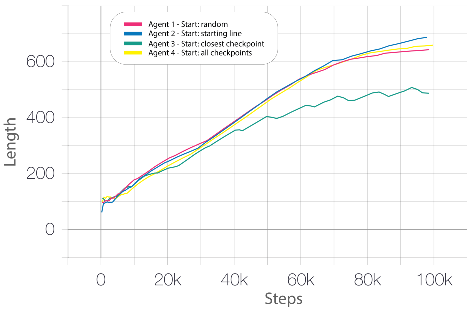
In our setup, the throttle is kept constant for the purposes of this thesis. The reward function described above is used to test our simulated algorithm and as a starting point, however, we need to adapt it such that it can work also in the real world where the cross-track error is not available. To tackle the issue we simply remove the CTE penalty, even though this will lead to a major problem of shaky motion as described in the next section. The final reward function that has proven to work in both environments and that we will use in the following training procedures is computed as follows:

$$r_t = 1 + throttle\_reward + \begin{cases} if\,done & crash\_error \\ else & 0 \end{cases} \tag{5.2}$$

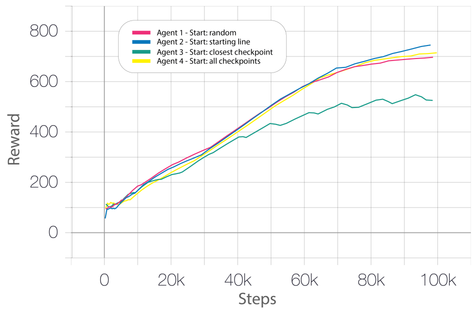
Since we want real and simulated version of our agent as similar as possible, Equation 5.2 is finally used in both cases.

5.2.2 Training the simulated RL agent

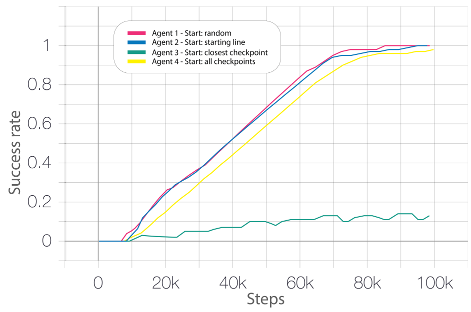
As a baseline for our RL algorithm, we used the source code provided by Raffin [2020] as a baseline. His algorithm allows the training of many RL algorithms, including the SAC of our interest, of both simulated and real-world agents, however, training on simulation with communication being over the internet is more computationally expensive and more prone to errors. Thus, for the simulation, we refactor the algorithm such that the communication happens locally. Moreover, his algorithm uses an AE which needs to be changed with the more performant VAE chosen above. To train our agents, we need to define what is the best strategy in terms of the starting point. We identified four main options, the first one lets the Donkey start always at the starting line, the second option starts the Donkey from a random checkpoint, the third one at the latest checkpoint reached in the previous episode and finally, the last option makes it start from all checkpoints cyclically. Defining in simulation which is the best strategy can save computational time in real-world training. To identify which one eventually converges more quickly and if it does, 4 different agents were trained, one for each option aforementioned. The quality measures to evaluate the trained agents, illustrated below in Figure 5.3, are the *Episode success rate* that shows how many laps have been completed on average during the training, the *Episode Reward mean* and the *Episode Length mean*. All the models have been trained for 100k iterations which correspond to ≈ 2 hours of training. *Agent 1* started each lap at a random checkpoint, *Agent 2* started always at the starting line, *Agent 3* at the latest checkpoint reached during the last episode, and, finally, *Agent 4* cyclically uses all the checkpoints. During



(a) Episodes length mean



(b) Episodes reward mean



(c) Success rate mean

Figure 5.3. Agents trained in simulation. Each agent has been trained with a different starting modality and has been trained for 100k steps.

AGENT	OOT	OBE	LAPS	AVG LENGTH	AVG REW
1	0	0	10	595	644
2	0	4	10	599	647
3	0	0	10	624	676
4	10	29	0	460	495

Table 5.5. Agents results averaged over 10 laps. Out Of Track (OOT) measures crashes, Out of Bound Error measure how many times it exceed the max CTE, and finally LAPS counts the completed laps.

our experiments, we noted that, in the best case, a lap may take ≈ 350 iterations to be completed. The agents are all able to successfully learn to drive with a strict maximum CTE (3) except for *Agent 4* which started new laps from the latest checkpoint. Two interesting shreds of evidence come out of those pieces of training. The first one is that even though the success rate mean approaches 100%, meaning the agents can consistently finish laps, the reward mean keeps growing. This shows a limitation in the reward function used, in fact, the agent gets a reward for every step and hence it learns to finish the lap following the longest path it has discovered. Moreover, the best way to lengthen the path is a zig-zag trajectory that allows also a doubling of the reward per lap. Secondly, the agent that starts at the latest checkpoint keeps improving the reward up to more than an equivalent completed lap, however, it never finishes a lap as described in Figure 5.3c. The reason behind this strange behavior is that the agent found a bug in the simulator used, as shown in Figure 5.4. Essentially, there is a little spot, off track, close to the steepest turn where the CTE is not correctly detected by the simulator, and consequently, the episode is not terminated. The reason why this behavior only happened with this agent lies in the training modality. When the agent reaches that checkpoint, he cannot easily reach the next checkpoint given the toughness of the turn, instead, it finds it easier to explore the bugged spot which is almost right in front of it when it approaches the turn.

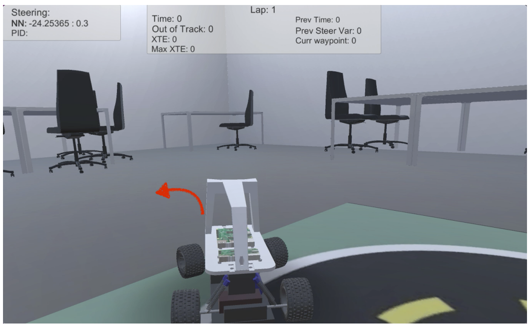
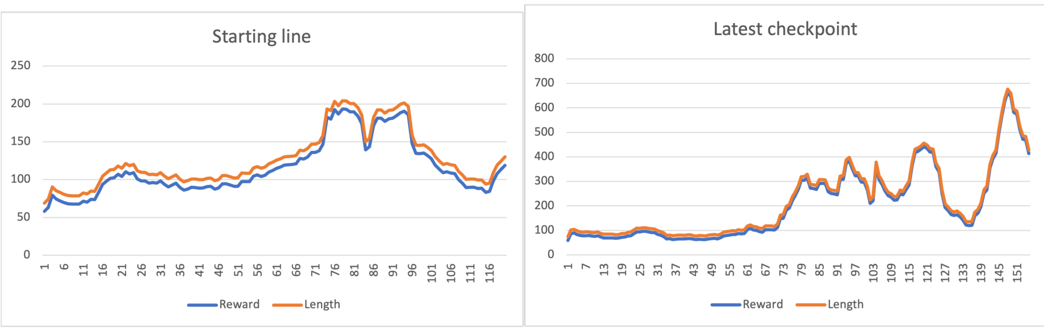


Figure 5.4. Spotted bug in the simulator

To further test the trained agents, for 10 laps it is measured how many times a lap has been completed by each agent, how many times the agents crash, and finally how many times they exceed the roadway but can recover and finish the lap without crashing. The result are presented in Table 5.5. From the results is evident, excluding *Agent 4* because of the simulator's bug, that three agents learned to successfully drive and in most cases, they always stay entirely on track without the need for any additional sensor and with the only problem of the shaky driving which is still acceptable for the purposes of this thesis, in most cases, they never get out of the track, and if they do they can recover consistently.

5.2.3 Training the real RL agent

To train the real-world agent, instead, the source code provided by Raffin [2020] is kept untouched beside the encoder, with the main goal being to replicate their results but with a more performing VAE as resulted in our tests. Given that in the real world the simulator's supervision is not available, all the strategies tested in the previous section are good candidates to be used, also *Agent 4* strategy that cannot explore anymore the simulator's bug. In fact, in the real world, we only have human supervision that is about stopping the episode as soon as the car exceeds the track boundaries with all 4 wheels, while the server automatically stops the car when it reaches 1000 steps ($\approx$ 2.5 laps). From the tests resulted that all the agents, trained with the aforementioned strategies, struggle to learn to drive an entire lap, at least in a reasonable time, except *Agent 4* that start his laps at the latest checkpoint reached in the last episode. For the sake of brevity, since they are all equivalent among the failing agents, only the results of the agent that starts always at starting line and fails in learning, as shown in Figure 5.5a.



(a) Agent performances: starting line (b) Agent performances: latest checkpoint

Figure 5.5. Agent trained in real world starting each lap at the starting line on the left and agent starting at the latest checkpoint on the right

In figure 5.5b, instead, are shown the performances in the training of the agent starting at the latest checkpoint (equivalent strategy of previous *Agent 4*), which can be considered successful and comparable to simulated agents since it did learn to complete a lap in about 30 minutes and two laps in about 45 minutes. In five to twenty episodes, the first two turns were learned decently and most of the time was spent on the steepest turn. As shown in Figure 5.5a, the graph is characterized by ups and downs, as soon as the car started at the starting line, it learned quickly, then, when the steepest curve was reached it struggled to overcome it and when it eventually did, the length started to increase again. The process was repeated until it was almost consistently able to finish a lap. Furthermore, as soon as the agent learned the steepest turn, it did generalize well on the following turns and little time was spent on them.

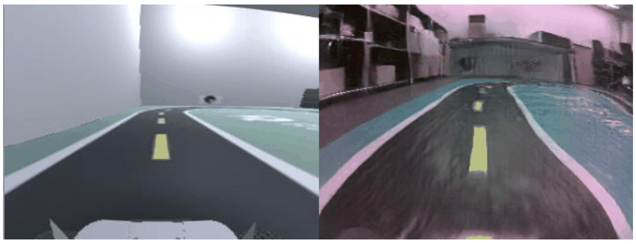
Notice that the laying of the car at the latest checkpoint has been intentionally approximate on the area close to the checkpoints. This brought a main advantage, the agent learned quicker since it was able to see the area in front of it from many points of view and this, resulted useful when the car started to cross many checkpoints per episode since the direction from which the car arrived to a checkpoint could vary a lot, it had been trained to drive on many possible trajectories and was able to join the various sections well. Unfortunately, in the real world, more metrics to measure the quality of the driving and to make comparisons with the simulated agents are not available.

5.3 Sim to Real

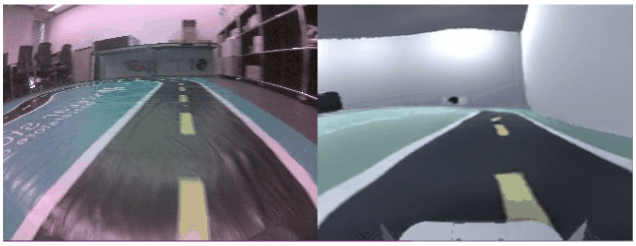
In this section we present an unsuccessful attempt in driving our simulated DonkeyCar with the real agent trained above. SimToReal (S2R) and vice-versa aims in deploying model trained in one environment to the other. In our case, since the real world environment, in our setup, does not provide enough metrics to benchmark our real world agent, we aim to make it works also in simulation. For example, in order to test the generalization of the real agent would be much easier in simulation where multiple tracks or obstacles can be implemented at a low cost. Another advantage brought by this approach is that an agent trained in simulation, can be moved into the real world and this would result in less expensive training procedures and eventually more robust agents. The idea is to pre-train a CycleGan [Zhu et al., 2017] for image transfiguration. In fact, the CycleGan is able to move an image into another domain keeping the original structure unaltered, but applying the style of the other domain as shown in Figure 5.6. Thus we leverage this property to transform images seen by the simulated camera of the DonkeyCar into what it would see in real world and vice-versa. Then, a real agent will eventually be able to drive on the simulator since it does see pseudo-real images. On the other hand, in order to drive a real car with a simulated agent, our DonkeyCar has not enough computational power, hence it could not run in time a CycleGan, that has millions of parameters, to make the real car see pseudo-simulated images and drive with the simulated agent. However, the problem can be circumvented by training an agent entirely on simulation but with pseudo-real images. After training CycleGAN with our datasets, it is able to transfigure image with high fidelity as shown in Figure 5.6. In fact, in human eyes they are barely distinguishable. However, even if the result looks good it could not be the case for the AutoEncoder that needs to place similar real and pseudo real images close into the latent space and similarly for similar simulated and pseudo simulated images.

Hence, the real test set is transformed through the CycleGAN and then forwarded through the real VAE chosen and similarly for the simulated test set. Given that 64 dimension cannot be visualized, a further dimensionality reduction is applied with t-SNE down to two dimension, as shown in Figure 5.7. Since the datasets are not aligned we do not except a perfect overlap, instead, the encoder should be able to at least embed similar images in the same region of the space. However, the latent space shows that is not always the case, some regions does overlap but not all off them in both real and simulated dataset. This could lead the trained agent not to respond consistently in similar situations. A further attempt is made by aligning the set of data used through the CycleGAN. Once the CycleGAN has been used to transform simulated images into pseudo real images, it can be used again to bring them back to pseudo simulated images, resulting in aligned sets. However, notice that the distortion, barely visible before, increases as shown in Figure 5.8.

Unfortunately, the results do not change enough from the previous one as shown in Figure 5.9. Given that the t-SNEe dimensionality reduction may be a cause of our problem, a further investigation is made by checking what are the closest images between the real set and pseudo



(a) From simulated images to pseudo-real

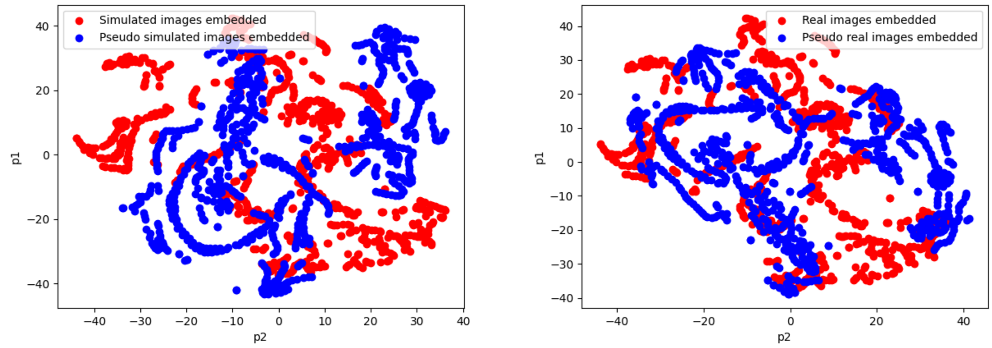


(b) From real images to pseudo-simulated

Figure 5.6. CycleGAN capabilities after training on our dataset

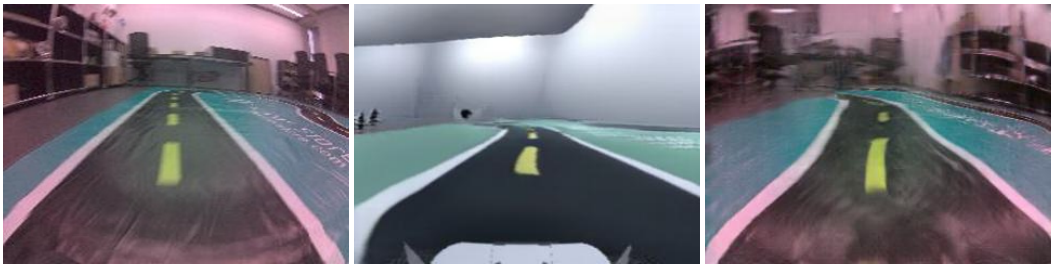
real set and similarly for the simulated set in the latent space. The distance measure used is the Euclidean distance and by looking at them there is some problem, in fact there are perfect matches as well as wrong matches as shown in Figures 5.11 and 5.12. This may be the cause of our agent not being able to drive when transferred into another domain, the encoder is not robust enough to compensate little image distortion.

As a final test, we are interested in studying what would be the actions undertaken by the agent on a set of aligned pictures. In particular, we selected a small set of 100 contiguous real pictures of the track and created its *aligned* pseudo real version. We then checked what would be the action of the real agent. As shown in Figure 5.10, the results are interesting since most of the times the agent takes the same action on both images, however even a tiny difference can lead the car out of track from which often it is not able to recover. The mean error on the predicted action resulted to be 0.20 TODO INSERIRE STD E BREVE CONCLUSIONE DELLA SEZIONE



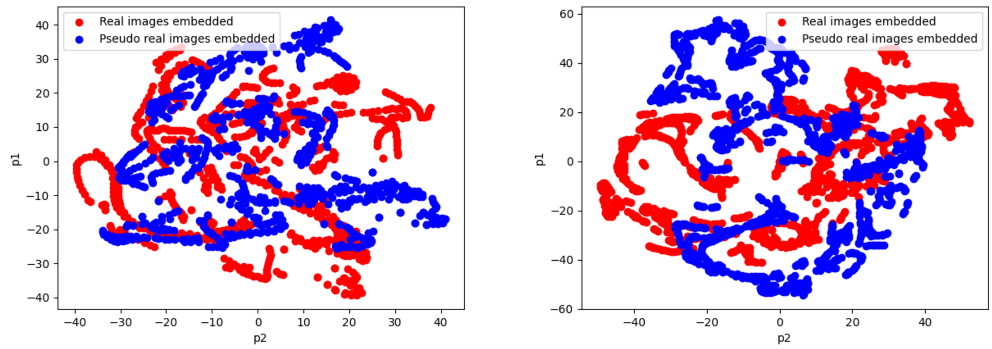
(a) Simulated and pseudo simulated images (b) Real and pseudo real images

Figure 5.7. Images embedded into the latent space with respectively the simulated and the real VAE.



(a) Real image (b) Pseudo simulated image (c) Pseudo pseudo real

Figure 5.8. Example of using the CycleGan to create an aligned dateset



(a) Aligned simulated and pseudo sim images (b) Aligned real and pseudo real images

Figure 5.9. Aligned images embedded into the latent space with respectively the simulated and the real VAE.

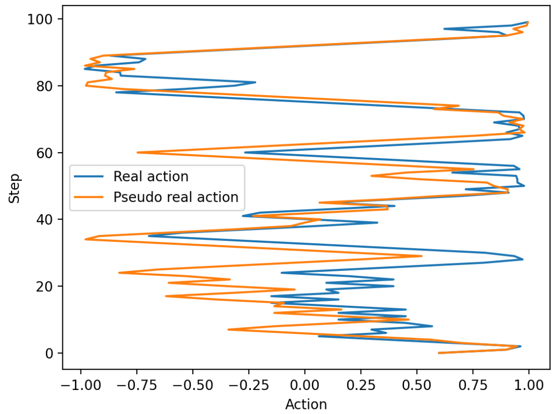
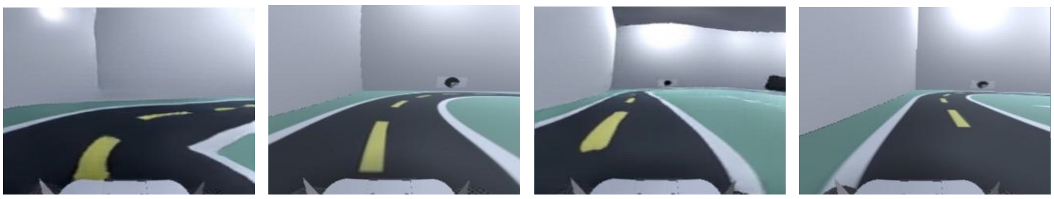


Figure 5.10. Real agent's actions on an aligned dataset



(a) Pseudo simulated (b) Closest simulated (c) Pseudo simulated (d) Closest simulated

Figure 5.11. Figures 5.11a and 5.11c show two pseudo simulated images and Figures 5.11b and 5.11d respectively the closest match in the simulated set measured with the Euclidean Distance.

(a) Pseudo real (b) Closest real (c) Pseudo real (d) Closest real

Figure 5.12. Figures 5.12a and 5.12c show two pseudo real images and Figures 5.12b and 5.12d respectively the closest match in the real set measured with the Euclidean Distance.

Chapter 6

Future work and conclusion

Another reward function was initally tested trying to improve the total time an agent takes to complete a lap:

$$r_t = -0.1 + throttle\_reward + cte\_penalty + \begin{cases} if done & crash\_error \\ else & 0 \end{cases} \tag{6.1}$$

The idea behind this function is that any step gives a negative reward and thus the agent must finish a lap in the smallest number of step to maximize the total episode reward. Minimizing the number steps means also finding the shortest way and consequently reducing the total time spent for a lap. Unfortunately, this approach did not let the agent learn to drive decently, at least in reasonable time.

Appendix A

VAE/AE architecture

Listing A.1. AE network

```
(encoder): Sequential(
  (0): Conv2d(3, 16, kernel_size=(4, 4), stride=(2, 2))
  (1): ReLU()
  (2): Conv2d(16, 32, kernel_size=(4, 4), stride=(2, 2))
  (3): ReLU()
  (4): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2))
  (5): ReLU()
  (6): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2))
  (7): ReLU()
)
(encode_linear): Linear(in_features=3072, out_features=z_size, bias=True)
(decode_linear): Linear(in_features=z_size, out_features=3072, bias=True)
(decoder): Sequential(
  (0): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2))
  (1): ReLU()
  (2): ConvTranspose2d(64, 32, kernel_size=(4, 4), stride=(2, 2))
  (3): ReLU()
  (4): ConvTranspose2d(32, 16, kernel_size=(5, 5), stride=(2, 2))
  (5): ReLU()
  (6): ConvTranspose2d(16, 3, kernel_size=(4, 4), stride=(2, 2))
  (7): Sigmoid()
)
```

Listing A.2. VAE network

```
(encoder): Sequential(
  (0): PreProcessImage()
  (1): Conv2d(3, 32, kernel_size=(4, 4), stride=(2, 2))
  (2): ReLU()
  (3): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2))
  (4): ReLU()
  (5): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2))
  (6): ReLU()
  (7): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2))
  (8): ReLU()
  (9): PostProcessImage()
)
(fc_mu): Linear(in_features=6144, out_features=z_size, bias=True)
(fc_var): Linear(in_features=6144, out_features=z_size, bias=True)
(decoder_input): Linear(in_features=z_size, out_features=6144, bias=True)
(decoder): Sequential(
  (0): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2))
  (1): ReLU()
  (2): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2))
  (3): ReLU()
  (4): ConvTranspose2d(64, 32, kernel_size=(5, 5), stride=(2, 2))
  (5): ReLU()
)
(final_layer): Sequential(
  (0): ConvTranspose2d(32, 3, kernel_size=(4, 4), stride=(2, 2))
  (1): PostProcessImage()
  (2): Sigmoid()
)
```

Bibliography

Ethem Alpaydin. *Introduction to Machine Learning*. The MIT Press, 2014. ISBN 0262028182, 9780262028189.

Zhengxue Cheng, Heming Sun, Masaru Takeuchi, and Jiro Katto. Deep convolutional autoencoder-based lossy image compression. In *2018 Picture Coding Symposium (PCS)*, pages 253–257, 2018. doi: 10.1109/PCS.2018.8456308.

William Curran, Tim Brys, Matthew Taylor, and William Smart. Using pca to efficiently represent state spaces, 2015. URL `https://arxiv.org/abs/1505.00322`.

Lovedeep Gondara. Medical image denoising using convolutional denoising autoencoders. In *2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW)*, pages 241–246, 2016. doi: 10.1109/ICDMW.2016.0041.

Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014. URL `https://arxiv.org/abs/1406.2661`.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, 2018. URL `https://arxiv.org/abs/1801.01290`.

G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006. doi: 10.1126/science.1127647. URL `https://www.science.org/doi/abs/10.1126/science.1127647`.

Xianxu Hou, Linlin Shen, Ke Sun, and Guoping Qiu. Deep feature consistent variational autoencoder. In *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 1133–1141, 2017. doi: 10.1109/WACV.2017.131.

Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. Image-to-image translation with conditional adversarial networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

Rico Jonschkowski, Roland Hafner, Jonathan Scholz, and Martin A. Riedmiller. Pves: Position-velocity encoders for unsupervised learning of structured state representations. *CoRR*, abs/1705.09805, 2017. URL `http://arxiv.org/abs/1705.09805`.

Levent Karacan, Zeynep Akata, Aykut Erdem, and Erkut Erdem. Learning to generate images of outdoor scenes from attributes and semantic layouts. 12 2016.

Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of gans for improved quality, stability, and variation, 2017. URL https://arxiv.org/abs/1710.10196.

Timothée Lesort, Natalia Díaz Rodríguez, Jean-François Goudou, and David Filliat. State representation learning for control: An overview. *CoRR*, abs/1802.04181, 2018. URL http://arxiv.org/abs/1802.04181.

Xiaodan Liang, Zhiting Hu, Hao Zhang, Chuang Gan, and Eric P. Xing. Recurrent topic-transition gan for visual paragraph generation. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

Danyang Liu and Gongshen Liu. A transformer-based variational autoencoder for sentence generation. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–7, 2019. doi: 10.1109/IJCNN.2019.8852155.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

Antonin Raffin. Learning to drive in 5 minutes. https://github.com/araffin/learning-to-drive-in-5-minutes, 2020.

Antonin Raffin, Jens Kober, and Freek Stulp. Smooth exploration for robotic reinforcement learning. In Aleksandra Faust, David Hsu, and Gerhard Neumann, editors, *Proceedings of the 5th Conference on Robot Learning*, volume 164 of *Proceedings of Machine Learning Research*, pages 1634–1644. PMLR, 08–11 Nov 2022. URL https://proceedings.mlr.press/v164/raffin22a.html.

Patsorn Sangkloy, Jingwan Lu, Chen Fang, Fisher Yu, and James Hays. Scribbler: Controlling deep image synthesis with sketch and color, 2016. URL https://arxiv.org/abs/1612.00835.

David Silver, Aja Huang, Christopher Maddison, Arthur Guez, Laurent Sifre, George Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 01 2016. doi: 10.1038/nature16961.

Andrea Stocco, Brian Pulfer, and Paolo Tonella. Mind the Gap! A Study on the Transferability of Virtual vs Physical-world Testing of Autonomous Driving Systems. *IEEE Transactions on Software Engineering*, 2022. URL https://arxiv.org/abs/2112.11255.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. URL http://incompleteideas.net/book/the-book-2nd.html.

Martijn van Otterlo and Marco Wiering. *Reinforcement Learning and Markov Decision Processes*, pages 3–42. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-27645-3. doi: 10.1007/978-3-642-27645-3_1. URL https://doi.org/10.1007/978-3-642-27645-3_1.

Ari Viitala, Rinu Boney, and Juho Kannala. Learning to drive small scale cars from scratch. *CoRR*, abs/2008.00715, 2020. URL https://arxiv.org/abs/2008.00715.

Denis Yarats, Amy Zhang, Ilya Kostrikov, Brandon Amos, Joelle Pineau, and Rob Fergus. Improving sample efficiency in model-free reinforcement learning from images. *CoRR*, abs/1910.01741, 2019. URL http://arxiv.org/abs/1910.01741.

Yizhe Zhang, Zhe Gan, Kai Fan, Zhi Chen, Ricardo Henao, Dinghan Shen, and Lawrence Carin. Adversarial feature matching for text generation. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 4006–4015. PMLR, 06–11 Aug 2017. URL https://proceedings.mlr.press/v70/zhang17b.html.

Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Computer Vision (ICCV), 2017 IEEE International Conference on*, 2017.