Training a Real World Reinforcement Learning Agent

A case study using the DonkeyCar autonomous driving framework

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

Abstract

Reinforcement learning (RL) allows robots to learn skills from interactions with an environment that can be real or simulated, where the problem can be modelled as a sequential decision-making task. In practice, recent advances in RL involve the use of a simulator, where training can be accelerated by parallelizing data collection and safety is not an issue. On the other hand, the field advances at a much slower rate when it comes to the real world. Indeed, the applicability of RL in the real world is held back by data efficiency to reduce training time, limited availability of sensors and safety of both the robot and the surrounding environment.

In this thesis, we evaluate the best strategies to train an RL agent in the real world to control a small scale electric car using only a camera sensor. In particular, we assessed different *representation learning* techniques and *environment reset* strategies, first in a faithful simulated environment and then in the real world.

Finally, we also experimented with *sim2real* (simulation to real) techniques to evaluate whether an RL agent trained in one domain (i.e., simulation) can be transferred into the other (i.e., real).

Acknowledgements

The writing of this thesis was made possible by Prof. Paolo Tonella, and co-supervisor, Dr. Matteo Biagiola from the Software Institute. I want to thank them for giving me the opportunity to work with them in the subject of my interest and for their immense availability and kindness that has always helped me, event in difficult moments when their advice was fundamental for the success of this project, from the very first to the very last day.

For having achieved my academic goals I have to thank also all my classmates as I consider them indispensable for my academic career, with them I shared laughter, critical moments and knowledge, we supported each other in all phases of our Master, and without them this wouldn't have been such a great journey.

I also thank my parents Salvatore and Giovanna and my sisters Jessica and Giada, who from outside the university have contributed in their own way to my life in all these years of study, making it normal and concretely possible.

Last, but definitely not least, I want to thanks the newcomers in my life. My girlfriend Lara who has been supportive since the very first day we met without hesitation and my newly born nephew Amanda. They both have brightened my days contributing to the success of my studies.

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 modeled as Markov Decision Processes (MDP) [van Otterlo and Wiering, 2012] without the need for labeled data. Under this framework, an RL agent interacts with an environment in discrete time steps, observing the state of the environment and deciding actions based on it, in order to maximize a given reward function to solve a certain task. Such task is usually *episodic*, i.e., there are clear boundaries to determine when the task is achieved.

The first RL algorithms were tabular and could solve simple tasks and games Sutton and Barto [2018]. The advent of *Deep Learning* made it possible for RL algorithms to scale towards interesting and challenging problems, where the state and/or the action space are too big to be stored in tables (e.g. images and continuous actions). Most notably, the first Deep RL (DRL) algorithm, called Deep Q Network, was able to surpass humans in most of the Atari games Bellemare et al. [2013]. Recent developments in RL have shown how the framework can deal with even more complex games, from the game of Go Silver et al. [2016] to multiplayer games such as Starcraft II Vinyals et al. [2019] and Dota II Berner et al. [2019].

Despite the impressive performance, RL is still confined to simulated environments, where training data can be easily collected by running several instances in parallel and safety is not a primary concern. Indeed, progress towards bringing RL into the real world, has been much slower w.r.t. the progress made in simulation, given the higher complexity and unpredictability of real-world environments. One of the domains in which RL has been applied to the real world is robotics Smith et al. [2022]; Raffin et al. [2022]; Gu et al. [2017]. However, training RL agents to control physical robots in the real world presents several challenges that are absent in simulation. First of all, training cannot be easily parallelized and therefore the RL algorithm needs to be *data-efficient*, i.e., it needs to be able to reuse the collected experience for training since acquiring data is costly. Secondly, the movements of the robot need to be taken into account both to protect the environment around the robot that might carry out unsafe actions during learning and to safeguard the robot itself that can be damaged by the high-frequency actions the robot is controlled with. Moreover, the number of sensors in the real world is limited compared to simulated environments. Having less sensors impacts the design of the reward function that is necessarily more *sparse* (i.e. it gives less guidance to the agent during training) than in simulation. Finally, when the RL agent completes an episode, either successfully or unsuccessfully, a *reset* of the environment is needed, in order to restore its initial state. Depending

on the context, such task in the real world might require substantial human intervention.

Positive results in training RL agents to control robots in the real world often come from very constrained applications. For instance, the OpenAI team was able to train a RL agent to control a robotic hand in order to solve the Rubik's cube Akkaya et al. [2019]. Other examples are robotic arms that have been trained to reach a certain object, pull or push a door and to pick an object and place it in a target location Gu et al. [2017]. The whole RL training process for such tasks can be almost entirely automated requiring little to no human supervision.

On the other hand, there are fewer tasks that have been addressed with RL in real-world uncontrolled environments Smith et al. [2022]; Viitala et al. [2020]. In this thesis we target with RL the problem of *self-driving* and, in particular, we are interested in the *lane-keeping* task in which the RL agent needs to control a car to drive along a predetermined track. This problem is usually tackled using the Supervised Learning (SL) approach both in academia and in industry. In such learning paradigm a dataset of driving experience is collected, usually composed of pairs of images and labels. In its simplest form the label is the recorded steering angle applied by the human driver corresponding to a certain image. Then, the dataset is used to train a Deep Neural Network (DNN) to minimize the error between the predicted steering angle and the ground truth label. Even though SL has proven successful for this task in the real world, it has some disadvantages. First, it requires huge labeled datasets that need to be constantly updated to take into account new scenarios. Furthermore, a SL model is trained offline and, as such, it cannot predict the effects that a certain action has on the environment. On the other hand, a RL agent is trained online, thereby being able to learn and adapt based on the effects of its actions, respecting the sequential nature of the task of driving.

The objective of this thesis is to train a RL agent to control a physical 1:16 scale radio-controlled car (called DonkeyCar [1]) and make it drive along a physical track. The car is equipped with a camera, which is the only sensor the agent has to perceive its surroundings. We first addressed the problem in simulation, by using a high-fidelity simulator and a faithful representation of the physical track. We use simulation to carry out experiments to evaluate different training configurations in order to select the best choice when training in the real world. In particular, we experimented with different Representation Learning (ReL) techniques to speed up learning, as learning directly from raw images is known to be ineffective Viitala et al. [2020]. Moreover, we evaluated different reward functions and several reset strategies. Then, we transferred this knowledge to the real world and trained a RL agent which is able to learn to drive along the physical track in a reasonable amount of time.

Finally, we also experimented with *sim2real* (simulation to real) techniques that can be used to transfer knowledge from one environment (e.g. in simulation) to another one (e.g. in the real world). Our attempts can be useful to understand the direction of future endeavors at addressing the same problem.

[1] https://www.donkeycar.com/

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 an n-dimensional vector where each dimension represents the torque command to each of the n joint motors. The reward is a measure of how good the actions carried out by the agent are.

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:

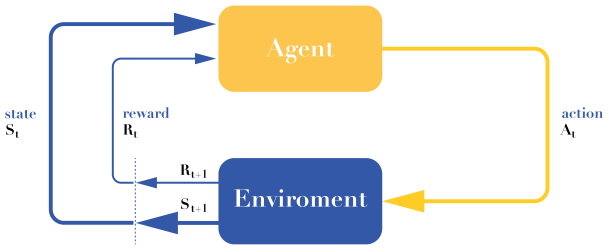


Figure 2.1. Reinforcement Learning loop

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

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:

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

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.

```
1 Initialize V(s) to arbitrary values
2 Repeat
3     for all s in S
4         for all a in A
5             Q(s,a) = E[r|s,a] + γ ∑_{s'∈S} P(s'|s,a)V(s')
6         V(s) = max_a Q(s,a)
7 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 the actions that maximize the optimal Q function all states.

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

```
1 Initialize policy π' arbitrarily
2 Repeat
3     π = π'
4     Compute the values using π
5         V_π = E[r|s,π(s)] + γ ∑_{s'∈S} P(s'|s,π(s))V_π(s'))
6     Improve the policy at each state
7         π'(s) = argmax_a(E[r|s,a] + γ ∑_{s'∈S} P(s'|s,a)V_π(s')))
8 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. The replay buffer contains a collection of experience tuples (s, a, r, s'), where s is the state, a the action taken in state s, r the corresponding scalar reward and s' the new state reached by taking action a.

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 the agent lives and operates in.

The standard interface designed by Gym, makes it easy 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 thesis, is an example of custom environment. With Gym we can define the *action space* of the car, i.e., all the operations the agent controlling the car can carry out, such as steer and accelerate. Furthermore, we can also define the *observation space*, i.e., what the agent perceives about the environment at each timestep. For instance, a possible observation space for an agent controlling a car is the camera image.

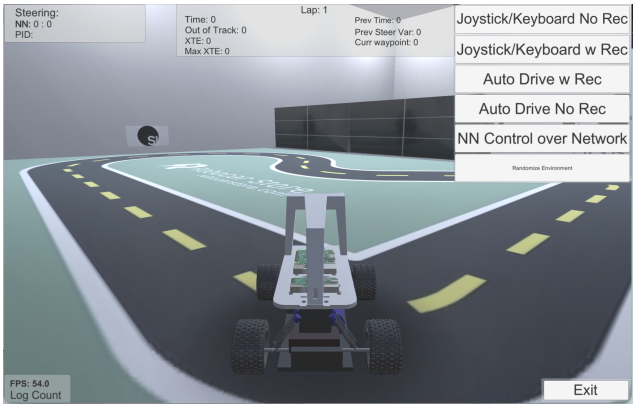


Figure 2.2. DonkeyCar environment

Moreover, the Gym 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:

```
1  class GymTemplate(gym.Env):
2    def __init__(self):
3      pass
4    def step(action):
5      pass
6    def reset():
7      pass
8    def render(mode='human'):
9      pass
10   def close(self):
11     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. According to 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 returns a tuple (observation,

reward, done) containing 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 conditions 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 the same initial conditions;
- **render:** this method renders the environment when a parameter *mode='human'* is passed.
- **close:** this method performs any necessary cleanup before closing the environment.

Beside 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 they are executed in the underlying environment (e.g., ActionWrapper);
- Transform observations that are returned by the underlying environment (e.g., ObservationWrapper)
- Transform rewards that are returned by the underlying environment (e.g., 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

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:

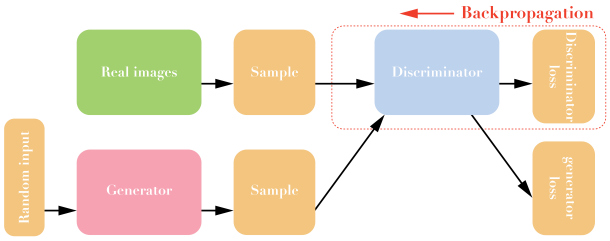


Figure 2.3. GAN diagram

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

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 proposed to translate images from one domain to another, 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 ($horse \leftrightharpoons zebra$).

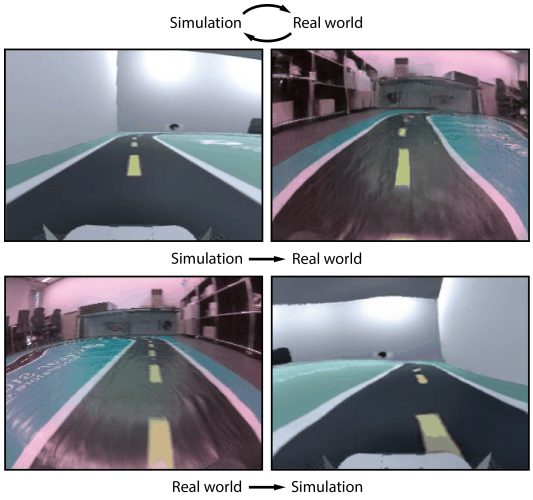


Figure 2.4. Image-to-image translation example.

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 under-constrained, 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 discriminators D_X and D_Y are provided. D_X tries to distinguish between examples coming from one domain (i.e. represented by the distribution p_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 2.11 and the following is the cycle-consistency loss:

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

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.

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

where $X = \mathbb{R}^{m \times n}$ and $Z = \mathbb{R}^k$ for some m, n, k and $k \ll m \times n$ 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.

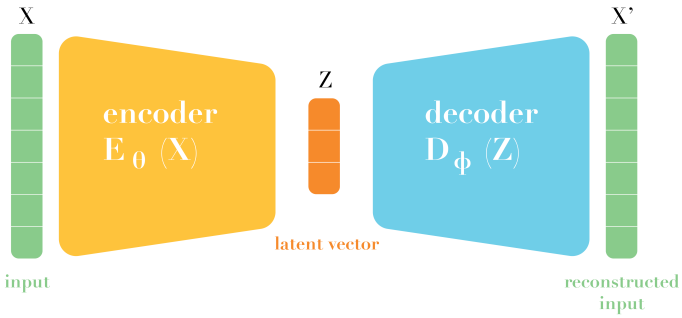


Figure 2.5. AE diagram

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*, i.e. not uniformly distributed, of data points in 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 parameters $[\mu_x, \sigma_x] = E_\phi(x)$ such that z is sampled from a normal distribution with such parameters $z \sim \mathcal{N}(\mu_x, \sigma_x)$. Consequently, the decoder brings it back to the original space $x' = D_\theta(z)$ with $x' \in X'$. Finally, x' can be used as a label with any distance measure $d(x, x')$. Thus the loss to be minimized is computed as follows:

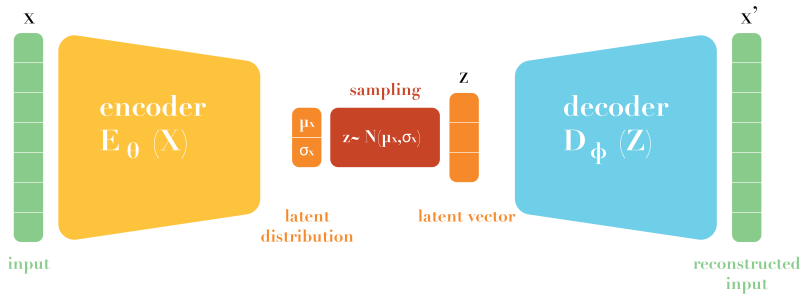


Figure 2.6. VAE diagram

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

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: 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 Do-It-Yourself (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.

Figure 2.7. Assembled DonkeyCar

In particular, the car used for the purposes of this thesis, is a basic DonkeyCar equipped with an 8-megapixel IMX219 sensor that features a 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 micro-controller. The power comes from a LiPO battery of 11.1V and 2200mAh that runs the 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 agent 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 the agent 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 low dimensional vectors that contain 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, authors described and compared methods that do not require explicit supervision such as AEs, VAEs, Principal Component Analysis [Curran et al., 2015] and Position-Velocity Encoders [Jonschkowski et al., 2017].

In our work we use AEs/VAEs to decouple state representation learning from policy learning, in order to speed up the training process.

3.2 Improving Sample Efficiency

The observation space of the agent in our experiments is an image, as perceived by the camera of the DonkeyCar, both simulated and real. However, training a model from high-dimensional images with RL is ineffective as Yarats et al. [2019] show. In their work the authors propose to use SRL to mitigate such issue.

In particular, they 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 reconstructs 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 different tasks directly from the latent space learned by the encoder, thus speeding up the training.

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 agent in our experiments. 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 and the state of the environment, 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] handle such 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 which is then added to the agent predicted action, inde-

pendently of the environment and the agent. 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, by sampling the parameters of the function only at the beginning of 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 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. SDE offers desired properties for the autonomous driving task we tackle in this thesis where a state-independent noise vector may damage the physical car due to the high-frequency of steering. Indeed, we activate SDE in the SAC algorithm when training the agent in our experiments.

3.4 Learning to Drive - L2D

Viitala et al. [2020] propose a framework to train a RL agent to control a physical 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 successfully along the considered tracks. 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 of 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 is about 11 meters long, i.e. ≈60% bigger than the track used by Viitala et al. [2020].

Figure 4.1. Real track

Figure 4.2. Simulated 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).

The *action space* is composed of two continuous actions, i.e. steering and throttle, both

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

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. convert all of them 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* the control is, 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

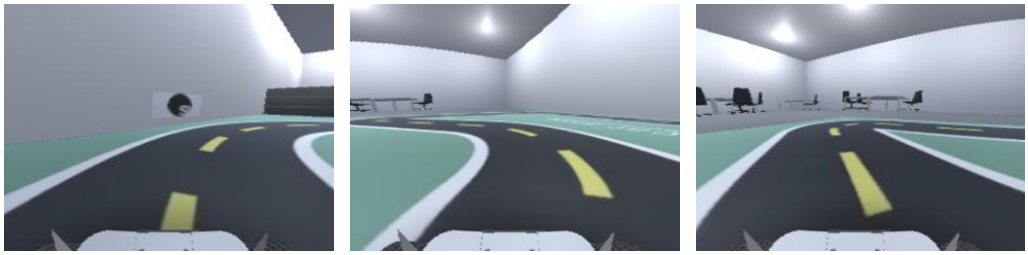


Figure 4.3. Images extracted from the simulated dataset

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.

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

Figure 4.6. Examples of cropped real images

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 $300k$ (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. 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 a 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 and the simulation is stopped for the entire duration of training and it is resumed as soon as the training finishes.

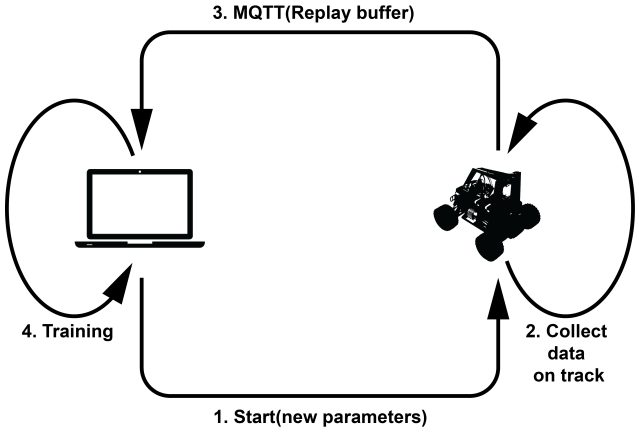


Figure 4.7. Key steps in the real training procedures

Also In the real environment we used a client-server architecture for training. However, client and server run on different machines, as shown in Figure 4.7. 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.e. 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 Representation Learning: AE vs VAE

We use state representation learning techniques to decouple state learning from policy learning, as previous works show that it considerably speeds up the training process Viitala et al. [2020]; Yarats et al. [2019]. We evaluate two strategies, i.e. an AutoEncoder and a Variational AutoEncoder, to understand if the stochasticity of VAEs can help in learning a good representation of the actual state.

In order to chose which representation learning strategy to use, we train AE and VAE with the same architecture, i.e., with an encoder composed of three convolutional layers interposed by ReLU functions and a variable size output layer; similarly the decoder is composed of three deconvolutional layers and the output layer is a sigmod. The detailed architecture can be found in Appendix A. Both AE and VAE have been trained for 50 epochs with an early stopping after five epochs of no improvement of the validation loss.

The latent space size (z) must be carefully chosen such it represents all the features of the images and consequently it produces high quality reconstructions. The latent space size determines the size of the states in the replay buffer transitions and, consequently, the training speed. In particular, we consider latent vectors of size 32 and 64.

We also investigated whether adding augmentation during training has an impact on the learned representations. In particular, we consider several augmentation methods such as Gaussian and motion blurring, contrast normalization, additive Gaussian noise, sharpening and coarse dropout. During training each operation can be applied with a certain probability and the sequence of operations is randomized.

For every combination of training set (simulated/real), representation strategy (AE/VAE), latent space size (32/64) and augmentation (True/False), we train the corresponding architecture three times. Each model is evaluated on the corresponding test set (i.e. simulated or real) and the resulting average reconstruction loss is averaged across the three training repetitions.

Table 5.1 and Table 5.2 show the results for the simulated dataset for the AE and the VAE respectively, while Table 5.3 and Table 5.4 report the results for the real dataset respectively for the AE and the VAE. Each table reports the size of the latent space (first column), whether augmentation was enabled during training (second column) and the summary of the reconstruction loss, i.e. mean, standard deviation, maximum and minimum, averaged across three training repetitions. The best average reconstruction loss for each table is highlighted in bold.

Z	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	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 5.2. VAE trained in simulation - reconstruction loss

Z	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 5.3. AE trained in real world - reconstruction loss

Z	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

From the tables we can see that, both in the simulated and in the real case, enabling augmentation increases the average reconstruction loss in the test set. Moreover, increasing the latent space size from 32 to 64 seems to decrease the average reconstruction loss both when considering the simulated and the real datasets. Finally, the VAE architecture always outperforms the AE architecture in all cases, as it produces a lower average reconstruction loss. Therefore, in light of such results, we pick the VAE architecture trained with a latent space size of 64 and no augmentation as representation learning strategy to train the RL policy both in simulation and in the real world.

In Figure 5.1 and in Figure 5.2 we can see an example of the capabilities of the chosen VAEs in terms of reconstruction, respectively for the VAE trained with real images (*real VAE*) and for the VAE trained with simulated ones (*simulated VAE*).

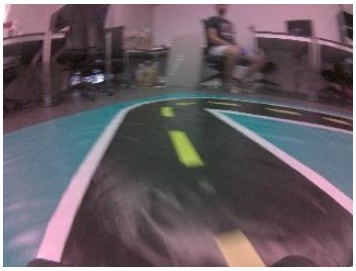
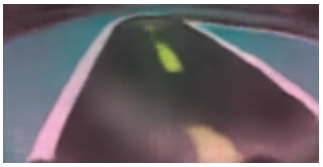



Figure 5.1. (Left) An image from the real world as seen by the DonkeyCar camera. (Right) The reconstructed image produced by the real VAE.

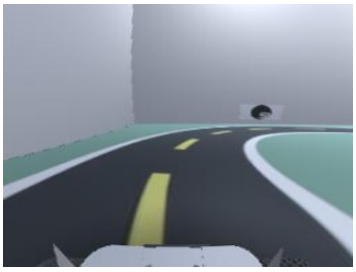



Figure 5.2. (Left) An image from the simulator as seen by the DonkeyCar camera. (Right) The reconstructed image produced by the simulated VAE.

5.2 RL Algorithm

5.2.1 Reward Function

Designing a reward function that is effective both in simulation and in the real world is challenging given the fundamental differences between simulated and real world environments. Indeed, in simulation, the environment can provide supervision and useful information such as the position of the car and the speed, that can be used to shape the reward function to guide the training. In our real setup, instead, the DonkeyCar can only leverage information coming from the camera. In particular, we consider the following reward function for both environments:

$$r_t = 1 + throttle\_reward + \begin{cases} if\ done & reward\_offtrack \\ else & 0 \end{cases} \tag{5.1}$$

The first term is a constant bonus for each timestep, that encourages the agent to stay *alive*, i.e. to stay on track in order not to abort the episode prematurely. The second term, i.e. $throttle\_reward = (throttle - min\_throttle)/(max\_throttle - min\_throttle)$, encourages the agent to go as fast as possible. However, for simplicity and to reduce training time especially in the real world, we keep the throttle constant such that the second component is always zero. Finally, the agent receives a big negative reward ($reward\_offtrack = -10$) when it goes offtrack. In simulation, this check is automated and corresponds to a cross-track error (XTE) greater than 3 (i.e., the entire car is out of track with all four wheels), while in the real environment the episode is stopped manually when the car goes offtrack. Otherwise, when episode finishes without the car going offtrack (1000 timesteps pass since the beginning of the episode), the agent is not rewarded nor it gets a penalty.

5.2.2 Training the RL Agent in Simulation

In simulation, we evaluated different *reset strategies* in order to understand how the performance of the RL agent varies. In order to define the different reset strategies we rely on *checkpoints* placed along the track (see Figure 4.2). In total there are seven checkpoints, distributed in this way: three of them are in the straight road (C1, C2, C7), one checkpoint is in the middle of the first right curve (C3) and one at the end (C4); then, there is another in the middle of the left curve (C5) and one at the end of the last but one right curve (C6).

Based on such checkpoints we define the following reset strategies with various levels of human intervention, considering such strategies applied in the real world:

- **Starting Line**: the agent always starts at the starting line when the episode starts. This is the default reset in the DonkeyCar Gym environment. The starting line is placed in the middle of the straight road sector of the track (Checkpoint C1, see Figure 4.2 and Figure 4.1);

- **Random**: the agent is placed in a random checkpoint every time an episode starts. The idea behind this strategy is to make the trained agent robust w.r.t. the starting point, since starting always from the same position (i.e., the Starting Line strategy) might lead the agent to repeat one sector of the track more often than the others;

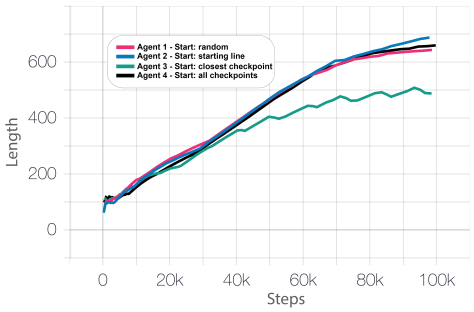
- **Closest Checkpoint**: when the episode finishes the agent is placed to the closest checkpoint w.r.t. its position at the end of the episode;

- **All Checkpoints**: in this strategy we keep track of the latest checkpoint the agent was placed in the previous episode and we restart the next episode by placing the agent in the next checkpoint. Once all checkpoints C1-C7 have been *covered*, we restart from the first.

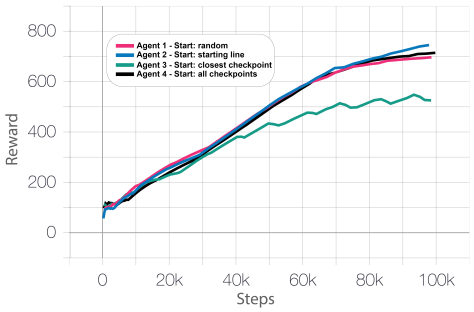
If we had to rank such reset strategies by the manual effort required by the human operator when training the agent in the real world, we would rank Starting Line, Random and All Checkpoints in the top positions, since the operator would need to take the car and place it in a position that is potentially far from the position the car went offtrack. On the other hand, the Closest Checkpoint strategy would be the cheapest since the car would need to be placed closely to where it went offtrack.

For each reset strategy we trained a SAC agent by using the pretrained *simulated VAE* trained earlier for $100k$ timesteps, which corresponds to $\approx$ two hours of training on a MacBook without GPU. During training we track different metrics, i.e., the episode length, the episode reward and the success rate. Regarding the success rate, an episode is considered successful if the agent is able to return to the checkpoint it started from. Figure 5.3 shows the results of the training process for the four agents. Each plot shows the respective metric averaged over 100 episodes. For example, each point in Figure 5.3.a is the episode length averaged over the previous 100 episodes.

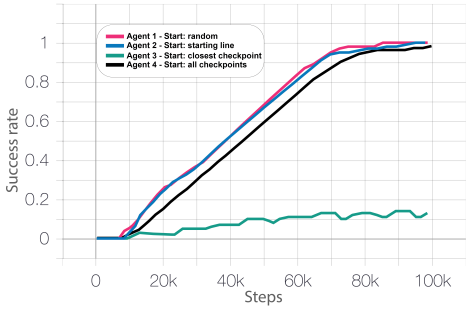
From the figure we can see that, for most strategies, the agent is able to reliably complete the task, except for the Latest Checkpoint strategy. The first two plots show the same trend for all agents; indeed, the reward function is directly linked to the episode length. Interestingly, although for the successful strategies the mean success rate converges at the end of training, which means that the agents are able to consistently finish the lap, the reward, and the episode length, keep increasing. The reason is that the agent is encouraged by the reward function to



(a) Mean episode length



(b) Mean episode reward



(c) Mean success rate

Figure 5.3. Agents trained in simulation. Each agent has been trained with a different reset strategy for 100k steps.

drive slowly while staying on track. Indeed, one way of slowing down is to learn a zig-zag trajectory, which is the way the agent learns to drive.

Another interesting result is the one achieved by the Latest Checkpoint strategy. Indeed, following such reset strategy the agent is not able to learn how to drive around the track, achieving a negligible 10% success rate. However, if we observe both the mean episode length and the mean episode reward plots (i.e. Figure 5.3.a and Figure 5.3.b) we can see that the respective value increases over time. In order to understand this particular behavior, we took the trained agent and tested it on the same physical track. We observed that the agent found a *blind-spot* where the simulator does not accurately measure XTE. Figure 5.4 shows the sector in the track where the DonkeyCar goes offtrack (such position is close to the last but one right curve). Essentially, the agent goes offtrack in a position where XTE is not accurately computed which means that the episode is not terminated and the agent is free of roaming in the scenario, without realizing being offtrack but accumulating reward points nonetheless. This behaviors can be observed only with this reset strategy. The reason is that, when the agent reaches the checkpoint before the last but one curve on the right, it cannot easily overcome the very challenging curve. Instead, it is easier for the agent to turn left in the exact positions where the XTE computation does not signal the end of an episode, hence the result is not reliable.

After training, we took the final agents for each reset strategy and tested them on the same track. In particular we executed each agent 10 times and enabled deterministic actions for the SAC algorithm, which means that, given an observation, the trained stochastic policy selects the

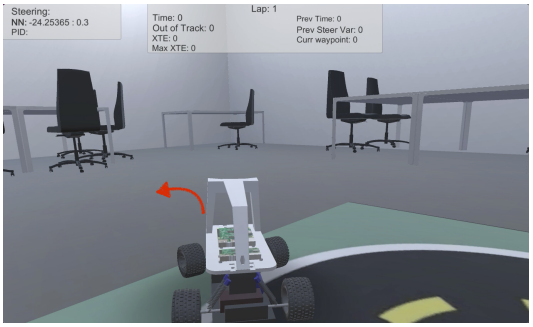


Figure 5.4. Blind-spot in the simulator being exploited by the agent

AGENT	# OOT	# OBE	SUCCESS RATE	EP. LENGTH	EP. REWARD
1 (Random)	0	0	1	595	644
2 (Starting Lines)	0	4	1	599	647
3 (Closest Checkpoint)	10	29	0	460	495
4 (All Checkpoints)	0	0	1	624	676

Table 5.5. Agents results averaged over 10 runs.

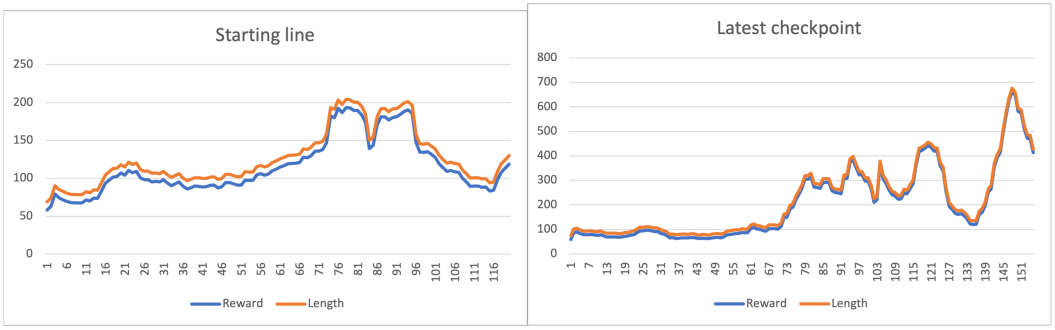
action with the highest probability. Indeed, we executed each agent multiple times to account for the randomness of the simulator. A lap is successful when the car is able to return to the starting checkpoint according to the reset strategy. The results of such experiments are shown in Table 5.5.

During testing we measure the average Out of Track (OOT) events (second column), i.e., the number of times the agent goes offtrack, the average Out of Bound (OOB) events (third column), i.e., the number of times the measured XTE is greater than 2, the average success rate (fourth column) and the average episode length and episode reward (respectively fifth and sixth columns). An OBE event means that the agent momentarily exists from the roadway but it is able to recover. From the results we can see that all agents, except the agent trained with the Closest Checkpoint strategy, are able to successfully complete the ten laps. Since the average reward is comparable across the successful strategy, no clear winner reset strategy to be used in real world training emerges.

5.2.3 Training the RL Agent in The Real World

Since no clear winner among the reset strategies emerged from the experiments in simulation, all the reset strategies, including the Closest Checkpoint one, are viable to be tested in the real world since it cannot exploit the simulator bug. However, we only experimented with the default reset strategy, i.e. the Starting Line strategy, and the most convenient strategy in terms of supervision effort, i.e. the Closest Checkpoint strategy. Indeed, we trained two agents, i.e., one for each strategy, with the SAC algorithm and by using the pretrained *real VAE*. Each agent was trained for $\approx$ 30 minutes.

Figure 5.5a shows the training curves for the two agents. The two lines (i.e., y-axis) for each plot show the reward and the episode length across the previous 100 episodes, while the



(a) Mean episode length and reward: Starting Line strategy

(b) Mean episode length and reward: Closest Checkpoint strategy

Figure 5.5. Agents trained in real world with the Starting Line strategy (left) and Closest Checkpoint strategy (right).

x-axis shows the number of episodes, i.e., the training time.

From the plots we can see that the agent that learns with the Starting Line strategy did not manage to achieve a good reward given a budget of $\approx$ 100 episodes. On the other hand the other agent with the Closest Checkpoint strategy was able to achieve 100% more reward with the same budget (i.e., 400 average reward vs 200 average reward). Therefore, after 100 episodes we decided to stop the training of the first agent, in order to save time. Instead, we carried out the training of the most promising agent, in order to observe further improvement. In total, such agent was trained for 45 minutes, taking approximately 30 minutes to complete a lap and by the end it was able to complete two laps (notice that in the real world an episode terminates successfully when 1000 timesteps pass, which correspond to $\approx$ 2.5 laps).

Interestingly, in the right plot of Figure 5.5a both the average reward and the episode length show an oscillating behavior starting at episode 73. Such behavior can be explained by the different levels of difficulty of the track and by the reset strategy. Indeed, when the agent is placed at the starting line it is able to drive well since the initial part of the track is relatively easy to drive, with a straight line and a smooth right curve. Then, when the agent arrives in the challenging sectors of the track (C5-C6), the agent goes offtrack and restarts in the closest checkpoint. Then, it takes the agent some time to learn how to drive in the challenging part and, as a consequence, the average reward decreases. The reward keeps decreasing until the challenging part is overcome and the it starts increasing again. This oscillating behavior repeats multiple times during training until the agent is able to drive the challenging part at full speed.

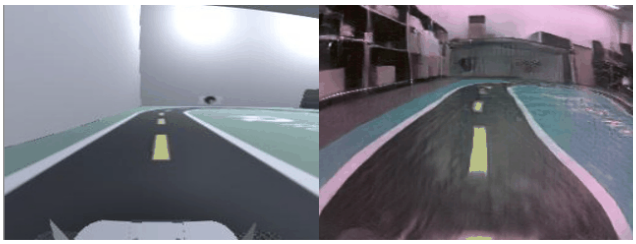
Regarding the difference w.r.t. the Starting Line strategy we suspect that the advantage of the Closest Checkpoint strategy is that, in the latter, the agent is given the chance to learn the challenging part of the track at a low speed, since the agent starts close to it. On the other hand, if the agent always starts at the Starting Line, it is forced to learn the challenging part at full speed, which requires more training time.

To summarize, we were able to train a RL agent in the real world in a reasonable amount of time, i.e. $\approx$ 30 to 45 minutes, by decoupling state learning, which we addressed with a VAE, from policy learning which we tackled with the SAC algorithm. Moreover, we used a custom reset strategy that enabled training of an effective policy.

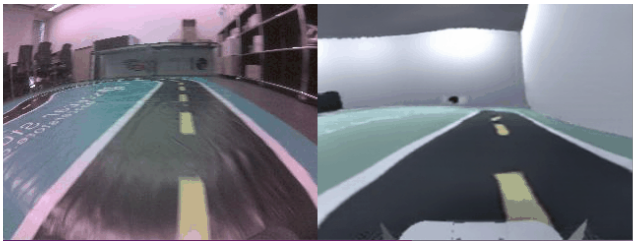
5.3 Sim to Real and Real to Sim

In this section we present an unsuccessful attempt at transferring the agent trained in the real world to simulation (real2sim, or r2s for short). This transfer would be useful in order to better evaluate the real world agent, given the constraints of the physical world. Indeed, in simulation it is possible to generate tracks with any shape and length to test the capabilities of the given agent. Likewise, transferring an agent trained in simulation into the real world (sim2real or s2r for short), would be useful to speed up the training process, especially by reducing the need of reset, and avoid unsafe behavior of the agent.

We follow previous work Stocco et al. [2022] that attempts to bridge the visual gap between simulation and real using image-to-image translation techniques. In particular, we train a CycleGAN Zhu et al. [2017] to translate the images in one domain (real) into the other (simulated) and vice-versa. In particular, we used the same hyperparameters of the original paper and trained the CycleGAN architecture for 95 epochs. During training we saved different checkpoints, we visually assessed the quality of the translations for some of them and picked the one with the best-looking translations. Figure 5.6 shows an example of translations generated by the CycleGAN generators (respectively, s2r generator in Figure 5.6.a and r2s generator in Figure 5.6.b). The images generated by the CycleGAN generator are called *pseudo*-real or *pseudo*-simulated, since they are translations from one domain into the other.



(a) From simulated images to pseudo-real.



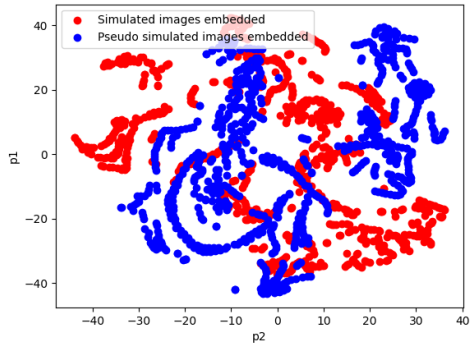
(b) From real images to pseudo-simulated.

Figure 5.6. Translations generated by the trained CycleGAN.

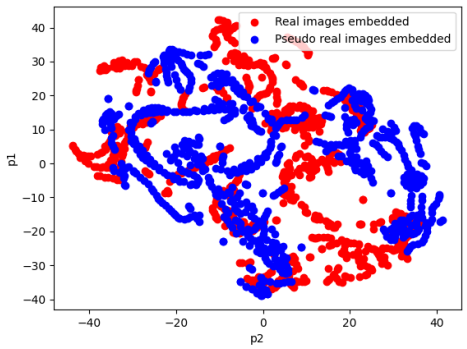
Once the CycleGAN architecture was trained, we used it to translate the two test sets of images we collected to evaluate the *simulated VAE* and the *real VAE*. The objective of this preliminary analysis is to understand how the *pseudo* images generated by the CycleGAN are mapped by the chosen VAEs in the two domains w.r.t. the *authentic* (i.e., real and simulated images captured by the camera of the car) images. Indeed, if we want to use the CycleGAN to transfer the trained agent from one domain into the other, we have to translate each image coming from the camera into a pseudo image of the other domain, which is mapped by the VAE into the latent

space; only then, the corresponding latent space is processed by the trained policy.

After translating the two test sets we computed the latent space for each image using the two VAEs and then applied the t-SNE dimensionality reduction technique Van der Maaten and Hinton [2008] to visualize how the translations are mapped by the encoders w.r.t. the authentic images. The visualization is shown in Figure 5.7.



(a) Simulated (authentic) and pseudo-simulated images



(b) Real (authentic) and pseudo-real images

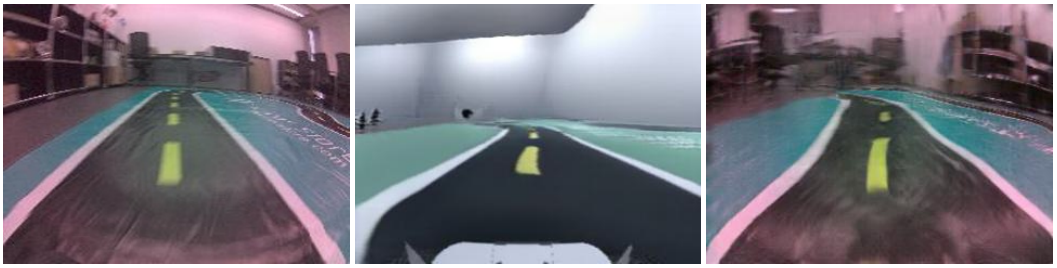
Figure 5.7. Latent space visualization: authentic vs pseudo (i.e., translated) images. (Left) simulated VAE, (right) real VAE.

Notice that the two test sets are not *paired*. Indeed, both represent images of a lap taken by driving manually along the track in simulation and in the real world. However, despite the images should represent similar situations during driving, we can see that the embeddings do not consistently overlap, both in the r2s case (Figure 5.7.a) and in the s2r case (Figure 5.7.b). There are some regions, more prevalent in the s2r case, in which the embeddings produced by the VAE overlap for the pseudo and the authentic images, but there are others that are mapped in different regions. This could lead the trained agent not to act consistently when presented with authentic and pseudo image, since the latent space that will be produced by the VAE will be different.

In order to understand whether the non-overlap is due to the unpaired dataset, we decided to repeat the previous analysis by creating a paired dataset using the CycleGAN. Indeed, we can use the two generators of the CycleGAN to translate an image from one domain into the other (first step) and then back into the original domain (second step). An example of such *double* translation for both domains is shown in Figure 5.8.

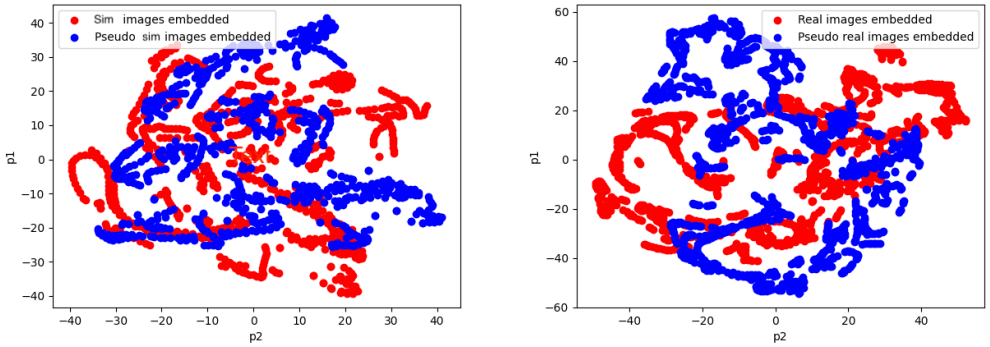
However, we can visually see that the quality of the translations degrades as more transformation steps are applied. After obtaining the paired dataset of authentic and pseudo images we computed the respective latent space vectors and used t-SNE to visualize them. The results are shown in Figure 5.9.

However, the overlap of the embeddings does not seem to increase w.r.t. the unpaired datasets. We further proceeded with a more fine-grained analysis by looking at the single images in the datasets. In particular, considering the paired datasets, we randomly sampled pseudo images and computed their latent space representation. Then, we computed the Euclidean distance between such representation and all latent space representations in the authentic paired



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

Figure 5.8. Using the CycleGan to create a paired dateset



(a) Paired Simulated (authentic) and pseudo-simulated images.

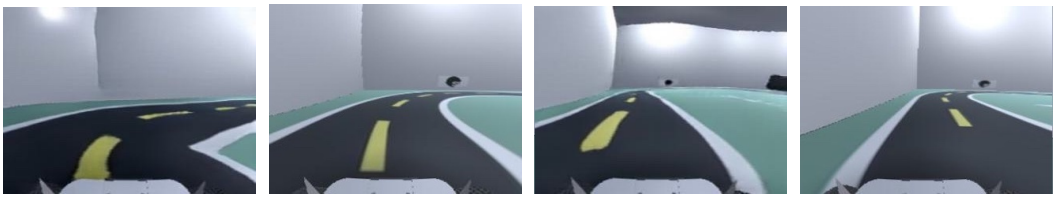
(b) Paired Real (authentic) and pseudo-real images.

Figure 5.9. Latent space visualization for the paired datasets: authentic vs pseudo (i.e., translated) images. (Left) simulated VAE, (right) real VAE.

dataset and took the corresponding image whose latent space is the closest. Examples of such images are in Figure 5.10 for the simulated environment and in Figure 5.11.

We can see from the images that in some cases the closest image in the latent space is actually similar to the authentic counterpart (see Good examples). On the other hand, there are also examples in critical parts of the track (see Figure 5.10a and Figure 5.11a), in which the closest authentic image in the latent space does not actually represent what is in the pseudo counterpart. As a consequence, transferring a trained agent from one domain into the other by using image-to-image translation seems challenging, as the VAE that represents the state for the trained policy does not seem to be robust w.r.t. the translations.

Finally, we analyzed the offline predictions of the agent we trained in the real world and compute the error between the prediction made by the agent when given the authentic real image and the prediction made when given the pseudo-real image. Specifically, we selected a set of 100 contiguous images from the real dataset and computed the corresponding paired translation of the first two sectors of the track (C1-C3), i.e. from real to simulated and from simulated to real. Then, we computed the prediction, i.e. the steering angle, of the trained agent when given each of the two images and plotted the results in Figure 5.12. From the plot



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

Figure 5.10. Figure 5.10a and Figure 5.10c show two pseudo-simulated images. Respectively, Figure 5.10b and Figure 5.10d are the closest match in the simulated set according to the Euclidean distance in the latent space of the simulated VAE.

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

Figure 5.11. Figure 5.11a and Figure 5.11c show two pseudo-real images. Respectively, Figure 5.11b and Figure 5.11d, show the closest match in the real set according to the Euclidean distance in the latent space of the real VAE.

we can see that, most of the time, the predictions are aligned. The average prediction error is 0.205 with a standard deviation of 0.183. However, small prediction errors can accumulate over time leading the car in sectors of the track where it is not able to recover.

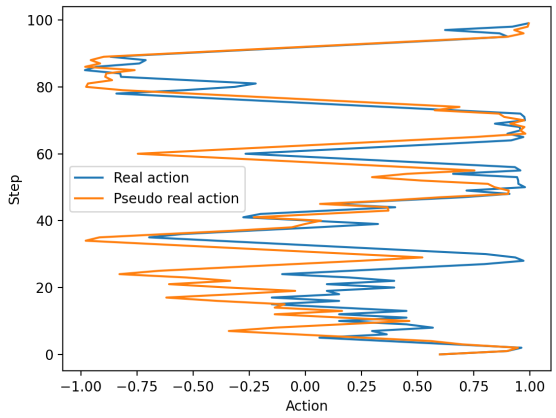


Figure 5.12. Trained agent predictions when given authentic real image and pseudo-real image.

Chapter 6

Conclusion and Future Work

In conclusion, we successfully trained a reinforcement learning (RL) agent to drive a physical car along a non-trivial physical track in our lab. Such objective was achieved by decoupling the state representation learning and the policy learning. In particular, we trained a Variational AutoEncoder (VAE) to learn how to map the images captured by the camera into a low-dimensional vector space in order to speed up training of the Soft Actor Critic (SAC) agent. We first trained the agent in simulation, by testing different reset strategies, with a reward function designed to work both in simulation and in the real environment. Then, we trained two agents in the real world using two reset strategies and found an effective way to train an RL agent in a reasonable amount of time. Finally, a preliminary investigation was carried out regarding a sim2real technique to transfer the agent trained in the real world in simulation.

As future work we plan to investigate the sim2real direction further by, for example, using the encoder of the CycleGAN to train the RL policy. Indeed, the encoder would act as the encoder of the VAE we used but with the advantage of having such component already trained as a byproduct of the CycleGAN training. Moreover, such encoder will be trained to produce a latent space that represents the features useful to reconstruct a simulated image from a real image and viceversa, which may be beneficial for transferring the policy from one domain to the other.

Furthermore, we want to equip the real DonkeyCar with additional sensors, such as accelerometer, in order to estimate the speed and design a reward function that can provide more guidance to the RL agent and further reduce the training 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

Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, et al. Solving rubik's cube with a robot hand. *arXiv preprint arXiv:1910.07113*, 2019.

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

Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.

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`.

Shixiang Gu, Ethan Holly, Timothy Lillicrap, and Sergey Levine. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *2017 IEEE international conference on robotics and automation (ICRA)*, pages 3389–3396. IEEE, 2017.

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.

Laura Smith, Ilya Kostrikov, and Sergey Levine. A walk in the park: Learning to walk in 20 minutes with model-free reinforcement learning. *arXiv preprint arXiv:2208.07860*, 2022.

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.

Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

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.

Oriol Vinyals, Igor Babuschkin, Junyoung Chung, Michael Mathieu, Max Jaderberg, Wojciech M Czarnecki, Andrew Dudzik, Aja Huang, Petko Georgiev, Richard Powell, et al. Alphastar: Mastering the real-time strategy game starcraft ii. *DeepMind blog*, 2, 2019.

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.