Robot training in virtual environments using Reinforcement Learning techniques

Natália Souza Soares
Centro de Informática
Universidade Federal de Pernambuco
Recife, Brazil
nss2@cin.ufpe.br

João Marcelo X. N. Teixeira
Departamento de Eletrônica e Sistemas
Universidade Federal de Pernambuco
Recife, Brazil
jmxntg@gmail.com

Veronica Teichrieb
Centro de Informática
Universidade Federal de Pernambuco
Recife, Brazil
vt@cin.ufpe.br

***Abstract*—In this work, we propose a framework to train a robot in a virtual environment using Reinforcement Learning (RL) techniques and thus facilitating the use of this type of approach in robotics. With our integrated solution for virtual training, it is possible to programmatically change the environment parameters, making it easy to implement domain randomization techniques on-the-fly. We conducted experiments with a TurtleBot 2i in an indoor navigation task with static obstacle avoidance using an RL algorithm called Proximal Policy Optimization (PPO). Our results show that even though the training did not use any real data, the trained model was able to generalize to different virtual environments and real-world scenes.**

***Keywords*—Reinforcement Learning, Robotics, Virtual Environments, Simulation**

I. Introduction

Robotics researchers are increasingly recurring to Reinforcement Learning (RL) techniques, due to the possibility of achieving outstanding results where there appears to be no obvious or easily programmable solution. Furthermore, it can adapt to new, previously unseen, environments/tasks and is robust to noise and errors [1]. With this type of learning, the developer has to specify a scalar reward function that assesses the robot's performance, and, during the training process, the agent "learns alone" by trial and error, how to perform the task by maximizing its rewards [2].

However, these algorithms have many details, which make them sensitive. Thus, it is challenging to implement this type of technique with an appropriate state and action representations, efficient observation space (i.e., sensing), an appropriate reward function for the task, policy parameters, exploration magnitude and strategy, and the initial policy, which is occasionally necessary [1].

When it comes to RL in robotics, the robot system itself is another challenging aspect, because it generally has high dimensional degrees of freedom (DOF), continuous states and actions, and high noise. Moreover, training by trial and error with the real robot could seriously damage its hardware and the environment.

This project aims to develop a framework capable of training robots in the virtual environment and then transferring the model to the real world with none or few calibrations in sensors or actuators. Thus, our solution should facilitate the use of RL techniques in robotics. To validate our hypotheses, we have conducted experiments with the Proximal Policy Optimization (PPO) [3] algorithm, which is an RL method widely used in robotics applications, to teach an indoor navigation task to a TurtleBot 2i [1]. This task has a high impact on the robotics community, and the robot is a low-cost robotic platform with open-source software.

II. Related Work

A. Virtual Training

Training in a simulator allows interactions with a wide variety of scenes and is much faster than real world experiences due to parallel computing. Simulated experiences are also safer because the agent can explore and learn without the risk of harming itself or the environment [4].

However, it is challenging to simulate some real domains with enough fidelity, and this often leads to a problem called reality gap, i.e., the difference between the agent's behavior in a simulator and the real world.

To minimize this problem, some works use domain randomization [4]–[7]. The concept of this approach is to train an agent in a variety of simulated environments to prepare for a wide range of possible real-world scenarios without additional training. In these papers, the authors trained the agent in scenarios with different parameters, such as lighting, objects, textures, and dimensions. Although they used realistic 3D meshes, they have designed the set of environments manually.

B. Indoor Navigation

Indoor navigation environments have been the subject of many recent computer vision works that apply RL techniques in robotics [6]–[11]. One concern of this task is to avoid the so-called sparse rewards environments because it is difficult to perform a task if the agent does not have enough feedback from the environment [9]. Therefore, the trained task is usually short-range navigation in controlled environments. Regarding the RL algorithm choice, the mentioned papers use at least one of the following:

- Proximal Policy Optimization (PPO) [3]
- Deep Deterministic Policy Gradient (DDPG) [12]

[1] Official web page: link.

- Trust Region Policy Optimization (TRPO) [13]
- Long Short-Term Memory (LSTM) [14]

Among them, PPO stands out because it has all the benefits of TRPO, another widely used method, but it is more general, efficient, easy to implement, and easy to tune.

III. Technical Background

In this section, we briefly present a theoretical background regarding Reinforcement Learning and PPO, which was the RL technique chosen for our proof of concept application.

A. Reinforcement Learning

In RL, one concern is with finding an optimal behaviour such that the expected sum of rewards on a given task is maximized. The behaviour of an agent is often referred to as a sequence of actions or policy. Thus, an agent interacts within an environment and learns to act optimally through trial and error by maximizing scalar reward signals [2], [15]. This closed-loop interaction is illustrated in Figure 1. The environment is modeled as a Markov Decision Process (MDP) [2] with transition probability $p(s_{t+1}|s_t; a_t)$ and in general a non-deterministic reward function modeled as $p(r_{t+1}|s_t; a_t)$. The agent may have full or partial access to its current state s_t , based on which action a_t is chosen, resulting in a real-valued reward $r(s_t; a_t) = r_{t+1} \quad p(r_{t+1}|s_t; a_t)$, and a next state transition $s_{t+1} \quad p(s_{t+1}|s_t; a_t)$.

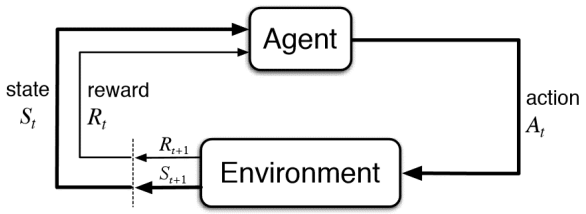

Fig. 1. The agent-environment interaction in an RL algorithm [2].

The main components of a reinforcement learning problem are the following:

- **State space:** a choice of representation for current state s_t in the given environment. For example, for an agent aiming to balance a pole attached to a cart, the observable current state could include the current position and velocity of the cart and the angular position and velocity of the pole, resulting in a four-dimensional vector $s_t \; 2 \; \mathbb{R}^4$ [2].
- **Action space:** a choice of representation of an action a_t that is performed by the agent on the environment. For the same pole balancing example, the action could be represented as a force applied horizontally to the cart $a_t \; 2 \; \mathbb{R}$.
- **Reward function:** the reward function $r_{t+1} = r(s_t; a_t)$ ultimately must encode the task the agent is meant to achieve, it represents a performance index to be optimized. It can be an arbitrary deterministic real-valued function of state and action, or a non-deterministic function, in which case $r(s_t; a_t) \quad p(r_{t+1}|s_t; a_t)$. For instance, to balance a pole, one could define a deterministic reward function such that it returns $\quad 1$ whenever balancing fails, and 0 otherwise. This means that maximum reward is zero if the agent finds a stable policy, as a sequence of forces is applied to a cart to keep the pole balanced.

B. Proximal Policy Optimization

PPO algorithm is a policy gradient method that works in an Actor-Critic Style. The Actor model chooses the agent action based on the current state, and the Critic model evaluates this action and gives feedback to the first one. The main contributions of this method are: improves training stability, through limitations in policy updates, and improves training efficiency by allowing multiple iterations of stochastic gradient ascent during policy updates.

The objective function of PPO (Equation 3) aggregates the objective function of the Actor model (Equation 1) and the Critic model (Equation 2). $\hat{E}_t$ is an expectation operator, i.e., an empirical average over a finite batch of samples (minibatch), $\quad$ is the stochastic policy, i.e., a neural network that receives inputs from the environment and returns the actions to be taken, and $\hat{A}_t$ is the advantage function. This function evaluates how well the action taken by the agent was in comparison to what was estimated by the Critic model. For that purpose, one may use a version of the algorithm known as Generalized Advantage Estimation (GAE).

$$L_t^{CLIP}(\;) = \hat{E}_t[min(r_t(\;) \quad \hat{A}_t; \\ clip(r_t(\;); 1 \quad ; 1 + \;) \quad \hat{A}_t)] \quad (1)$$

$$L_t^{VF}(\;) = (R \quad V(s_t)) \quad (2)$$

$$L_t^{PPO} = \hat{E}_t[L_t^{CLIP}(\;) \quad c1 \;\; L_t^{VF}(\;) + c2 \;\; S[\quad](s_t)] \quad (3)$$

In this version of the GAE algorithm, as shown in Equation 4, a delta value (t) is defined for each instant t in the given interval [0, T]. This value is the difference between the received reward (r_t) and its estimated value by the Critic model ($V(s_t)$), incorporated to a fraction of the estimated value of the next state ($V(s_{t+1})$). The future reward values received a discount because a current reward is more valuable than a future one. The delta values are then weighted aggregated to calculate $\hat{A}_t$, as shown in Equation 5. These weights reduce the influence of the estimated values, which can be noisy.

$$_t = r_t + \quad V(s_{t+1}) \quad V(s_t) \quad (4)$$

$$\hat{A}_t = \;_t + (\quad) \;\; _{t+1} + ::: + (\quad)^{T-t+1} \;\; _{T-1} \quad (5)$$

The loss function presented in Equation 1 is a lower bound version of the vanilla policy gradient method since it calculates the minimum between its objective function and a surrogate one. This function improves training stability by limiting the policy updates at each iteration. It uses the probability ratio $r_t(\ ) = \frac{\ _\theta(a_t|s_t)}{\ _{\theta old}(a_t|s_t)}$, and the hyperparameter to clip the values of $r_t(\ )$ to the safe interval [1 ;1 +]. This prevents changes in the policy that may lead to worse behavior, which is common in vanilla policy gradient methods with large step size values.

The loss function presented in Equation 2 is a mean square error of the estimated and real reward values. Equation 3 joins these two objective functions, adding a term of randomness, called entropy term, which ensures that the agent performs enough exploration during training.

IV. Framework

Our proposed framework, illustrated in Figure 2, was developed in Python 3 using a notebook with Ubuntu 18.04 and consist of three main modules:

- **Back end:** this module comprehends several RL algorithms through RLlib [16], such as PPO and TRPO. It also contains implementations of training environments, and back end support for physics engines, robots, sensors, and actuators. The module offers support to two ML training libraries: TensorFlow and Pytorch.
- **Communication:** this module uses a version of ROS 2 (Dashing) [17] to communicate the back end with the simulation (front end) or with the real robot.
- **Simulation:** this module is the front end of the framework. It uses Gazebo with ODE to simulate the robot and its behaviors. It is also possible to run the training without the graphic interface.

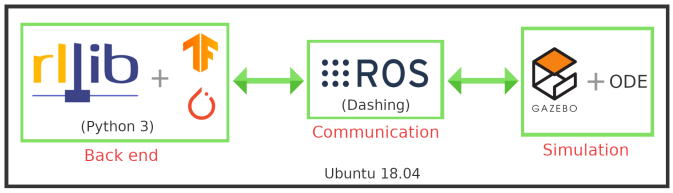

Fig. 2. Proposed framework. On the left side, is the module with the learning algorithms and back-end support. At the center, is the module responsible for the communication between the learning algorithm and the agent, supported by ROS. On the right side, there is the visualization and physics engine, which simulate the robot in the virtual environment.

The structure of our framework is described in more detail in the class diagram shown in Figure 3. We have implemented some base classes (shown in green in the diagram) to facilitate the implementation of new environments, robots, physics engines, and scenarios. It is also possible to programmatically change the training and environment parameters, making it easy to implement domain randomization techniques on-the-fly. We implemented some objects, sensors, actuators, robots, and environments that were used in the experiments. They are presented in blue in the class diagram exhibited in Figure 3.

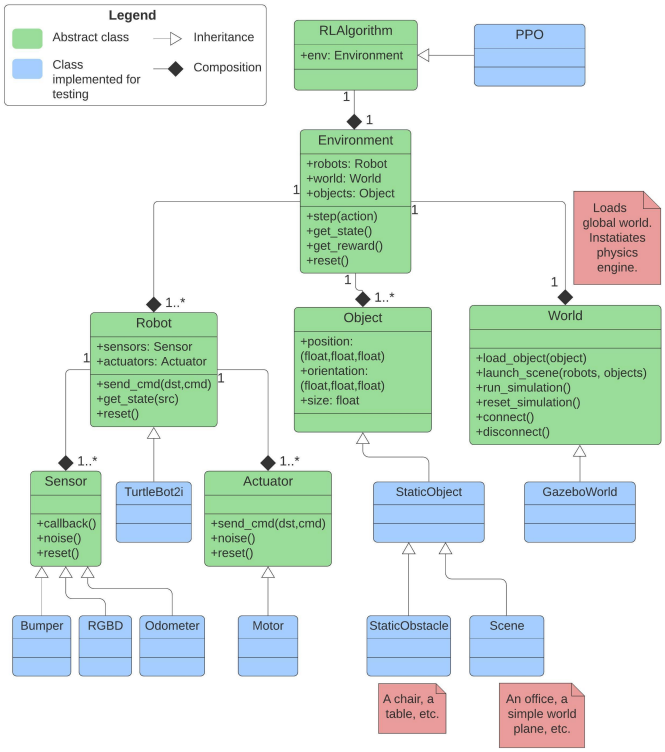

Fig. 3. Class diagram of the framework. The objects in blue were implemented for the proposed training/test environments.

A. Training Environment - Indoor Navigation with Obstacle Avoidance

Regarding the indoor navigation task proposed, we have simulated a room with objects inside, used as static obstacles. The training only requires information about positions and velocities provided by the physics engine. However, we have also used the initial environment description, i.e., its dimensions, name, and positions of the objects and the robot, to programmatically implement the graphic interface shown in Figure 4.

To reduce the dimensions of the neural network's inputs, the observation space of the robot was represented by a vector with the eight shortest distances read from a depth camera. We used this vector to train the model responsible for learning what action to take under a particular observed state, the so-called Actor model. This model returns a vector of probabilities of length equals to three, which rates the actions in the action space: go forward, to the right, or the left.

We designed the reward function, outlined in Table I, to severely penalize collisions and near-collisions. The reward function has a discount for navigation because if we give a positive reward for the robot just for walking, it could also end up spinning around its axis, to avoid collisions as much as possible [9]. Moreover, sideways navigation could lead to the robot spinning around its axis. Therefore, to prevent this behavior, we have a greater punishment for these actions than forwarding navigation. We also give the agent a cumulative

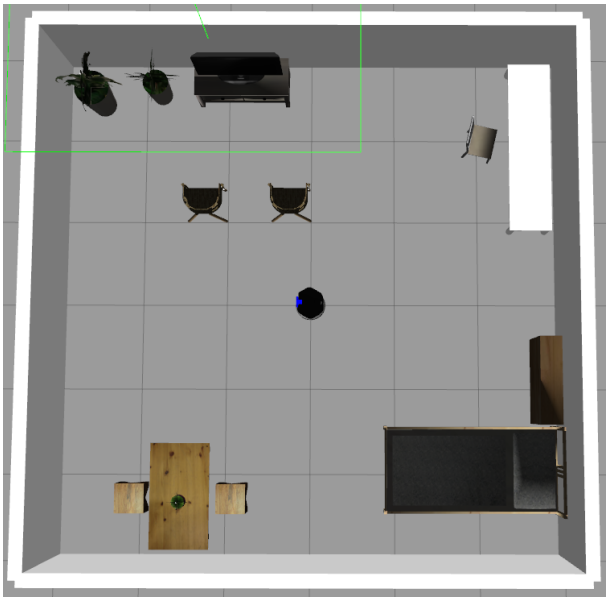

Fig. 4. Layout of the environment used for training the model. The robot is initially located in the center of the virtual environment.

bonus if it gets a positive reward, and it is not near any obstacle. In the same way, we give an extra punishment if it is near some object and going forward to hit it.

TABLE I
REWARD VALUE, $r(s_t, a_t)$, GIVEN THE STATE AND ACTION.

State (s_t)	Action (a_t)	Reward Value
Collision detected	Forward, left or right	-100
No collision detected and near from obstacles	Forward	$0.4 - 0.2(1/min(s_t))$
No collision detected and near from obstacles	Left or right	$0.6 - 0.2(1/min(s_t))$
No collision detected and far from obstacles	Forward	$0.6 - 0.2(1/min(s_t))$
No collision detected and far from obstacles	Left or right	$0.4 - 0.2(1/min(s_t))$

V. EXPERIMENTS AND RESULTS

Our framework was validated through training using the PPO algorithm and a TurtleBot 2i. We used the virtual environment for an indoor navigation task with static obstacle avoidance, described in subsection IV-A, as our main case study because of its vast popularity within the robotics community, as mentioned in section I.

A. Training Setup

The robot was trained with the hyperparameters showed in Table II. We have initially set these values accordingly to those mentioned in [3] and them performing a fine tuning. As stopping criterion, we used the maximum number of $Trials_{max} = 1{,}500$. The training was performed on CPU and took about 2 hours in a notebook with an Intel® Core™ i7-7500U CPU @ 2.70GHz × 4 and 16GB of RAM.

Regarding the Actor model and the Critic model, we chose to use a Multilayer Perceptron (MLP) architecture commonly used in literature [18]. This neural network has only two dense layers with 64 neurons and uses as an activation function, the $LeakyReLU$. The difference between these two networks is in the output layer. As the Actor model returns a probability distribution, its output layer has three neurons (representing the three possible actions), and its activation function is $Softmax$. The Critic model, on the other hand, outputs a real number. Therefore, its output layer has only a single neuron, and its activation function is $Tanh$.

TABLE II
VALUES OF PPO HYPERPARAMETERS USED IN TRAINING.

Hyperparameter	Description	Value
Num. iterations ($Trials$)	Number of trials of learning (model updates).	$1,500$
Horizon (T)	Limit for the amount of interactions with the environment (episodes) before the policy update. If the horizon is reached, a reward is calculated, but the environment is not reset.	100
Learning rate	The step size at each iteration while updating the model.	0.001
Num. epochs	Number of times to perform stochastic gradient ascent updates.	10
Batch size	The number of training examples used in one stochastic gradient ascent update.	128
Discount factor (γ)	MDP discount factor. Used to give more importance to the most recent rewards.	0.99
GAE parameter (λ)	Smoothing parameter used in the GAE algorithm. Used to reduce the training variance, making it more stable.	0.95
Epsilon (ϵ)	Clipping parameter. Used to keep the step from the old policy to the new policy within a safe range.	0.2

B. Virtual Training

As illustrated in Figure 5, the robot learns, in simulation, a policy that returns a good enough reward mean per episode, so it does not make significant policy updates. Since the reward means has reached a plateau, we could conclude that the training converges, and the robot has learned how to navigate without colliding with any obstacle.

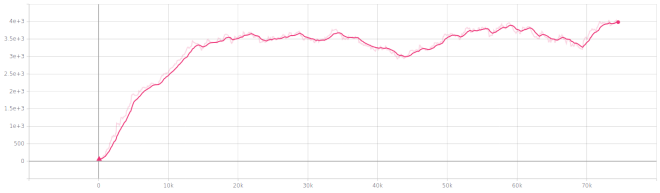

Fig. 5. Reward mean value received per episode during the training of an indoor navigation task with a TurtleBot 2i and the PPO algorithm.

C. Virtual and Real Inference

After training in the virtual environment, we tested the trained model in a new scenario, shown in Figure 6. This