PyRoboLearn: A Python Framework for Robot Learning Practitioners

Conference Paper · October 2019

3 authors, including:

Brian Delhaisse
Istituto Italiano di Tecnologia

7 PUBLICATIONS   **33** CITATIONS

Some of the authors of this publication are also working on these related projects:

PHOLUS View project

PyRoboLearn: A Python Framework for Robot Learning Practitioners

Brian Delhaisse[1]   **Leonel Rozo**[2]   **Darwin G. Caldwell**[1]
[1]Istituto Italiano di Tecnologia
Genoa, Italy
`name.surname@iit.it`

[2]Bosch Center for Artificial Intelligence
Renningen, Germany
`name.surname@de.bosch.com`

Abstract: On the quest for building autonomous robots, several robot learning frameworks with different functionalities have recently been developed. Yet, frameworks that combine diverse learning paradigms (such as imitation and reinforcement learning) into a common place are scarce. Existing ones tend to be robot-specific, and often require time-consuming work to be used with other robots. Also, their architecture is often weakly structured, mainly because of a lack of modularity and flexibility. This leads users to reimplement several pieces of code to integrate them into their own experimental or benchmarking work. To overcome these issues, we introduce *PyRoboLearn*, a new Python robot learning framework that combines different learning paradigms into a single framework. Our framework provides a plethora of robotic environments, learning models and algorithms. *PyRoboLearn* is developed with a particular focus on modularity, flexibility, generality, and simplicity to favor (re)usability. This is achieved by abstracting each key concept, undertaking a modular programming approach, minimizing the coupling among the different modules, and favoring composition over inheritance for better flexibility. We demonstrate the different features and utility of our framework through different use cases.

Keywords: Robot Learning, Software, Python, OOP

1 Introduction

Recent advances in machine learning for robotics have produced several (free and) open-source libraries and frameworks. These ease the understanding of new concepts, allow for the comparison of different methods, provide testbeds and benchmarks, promote reproducible research, and enable the reuse of existing software. Nevertheless, several frameworks suffer from a lack of flexibility and generality due to poor design choices. Lack of abstraction and modularity with high dependency among modules hinder code reuse. This problem worsens when the user needs to combine different incompatible codes together, or to integrate an existing one into her own code. Some frameworks force to follow a standard, which might not suit the user needs. However, bypassing code standards is not a good coding practice as many useful functionalities might be missed. Complying to their standard requires to modify the original code, interface (possibly) incompatible frameworks, and/or reimplement parts of the framework. This creates unnecessary overheads that considerably affect the research activities, leaving less time to create modular and flexible code, and therefore ad-hoc code that is hardly reusable is produced.

Available frameworks in *robot learning* [1] can be classified into two categories: "simulated environments" [2, 3, 4, 5, 6, 7, 8, 9, 10] and "models and algorithms" [11, 12, 13, 14, 15, 16]. In both, frameworks tend to focus on specific learning paradigms such as imitation learning (IL) [17] or reinforcement learning (RL) [18], and do not exploit their shared features, such as an environment, trainable policies, states/actions, and loss functions. In IL, a teacher provides demonstration data while for RL a reward signal is returned by the environment, which results in different training algorithms. The majority of frameworks that provide simulated environments focus either on RL [2, 3, 4, 5, 6, 7], or to a less extent on IL [8, 9, 10], which limits their applicability. As IL and RL differ on few aspects, their integration and design into a single learning framework provides

interesting opportunities. For example, IL can be used to initialize a policy which is then fine-tuned using RL, leading to safer and faster policy search [19]. However, current environment frameworks rarely exploit this feature.

To better illustrate our point, let us consider an RL setting where an environment inherits from an OpenAI Gym environment [2], which several frameworks use [3, 4, 6, 7]. Such environment includes the definition of state-action spaces, environment, and reward function. Also, let us consider an environment that includes an inverted pendulum on a cart. The state consists of the cart position and velocity, and the angular position and velocity of the pole. A simple reward function may count the number of time steps the cart could balance the pole. Finally, let us define a neural network policy that is specified outside the environment, which takes the $4D$ state vector and outputs the action. Now, assume that the user wants to test the performance of a new model/algorithm on a double inverted pendulum on a cart. In this case, the user would have to define manually a new environment with a new robot, and a larger dimensional state vector. This, in turn, affects the policy representation. Moreover, if the user wishes to experiment different reward functions, she would have to change them directly in the environment definition.

The above procedure is not efficient and does not scale. A better approach is to have the state to change its dimensionality automatically as the robot varies, and the neural network policy architecture to adapt accordingly. The reward function could be defined outside the environment and then provided to it. This lack of simplicity, modularity and flexibility along with the lack of a common framework regrouping different learning paradigms is what motivated us to create *PyRoboLearn*. We adopt a modular and SOLID programming approach [20], abstract important concepts, minimize the dependencies between modules, and favor composition over inheritance to increase flexibility. *PyRoboLearn* provides diverse environments, learning models and algorithms, and permits to easily and quickly experiment ideas by combining diverse features and modules.

2   Related Work

To reach high usability, our framework is written in Python and uses the PyTorch library [11] as backend. Frameworks in other languages are often prone to errors and not beginner-friendly. As such, we do not review the literature of frameworks written in other languages. In general, robot learning frameworks can be mainly categorized as: environment-based and model-based. We start by reviewing the literature of environment-based frameworks.

In IL, few environments have been proposed, notably SMILE [8] and the Freiberg Robot Simulator [9][1], but both focus on specific robotic platforms and use different programming languages. In contrast, multiple environments have been proposed for RL. One of the most used frameworks is OpenAI Gym [2] from which other frameworks have derived. OpenAI Gym provides environments in games, control, and robotics. Each one inherits from the abstract Gym environment class, and defines the world, the agents, the states-actions, and the reward function inside its class. Inheritance is used over composition which limits flexibility as a new environment has to be created for each combination of worlds (including the agents), states and rewards. OpenAI Gym and the DeepMind control suite [5], use MuJoCo [21]. Since MuJoCo requires a license, Zamora et al. [3] extended the Gym framework with Gazebo and ROS. OpenAI later released roboschool [4], a free robotic framework to test RL algorithms. Built on the PyBullet [6] simulator, PyBullet-gym [7] was recently released. All these frameworks focus on RL and most inherit from the OpenAI gym, following the same protocol.

Few other frameworks such as Carla [10] and Airsim [22] support both IL and RL, but are designed for autonomous vehicles. Another new framework closely related to ours is Surreal [23] which also supports IL and RL, but focuses only on manipulation tasks using the Baxter and Sawyer robots in MuJoCo. Other frameworks include the Gibson Environment [24] which focuses on perception learning and sim-to-real policy transfer, and the S-RL toolbox [25] which focuses on state representation learning. Both are out of the scope of the covered learning paradigms in this paper.

We now turn our attention to frameworks that provide models and algorithms. Several libraries have been proposed such as Sklearn [26], TensorFlow [27], PyTorch [11], GPyTorch [12], among others. As they use different backends (e.g. Numpy, TensorFlow or PyTorch), the models defined in one

---

[1]We tried to find this simulator online unsuccessfully.

cannot use algorithms of the others. In our framework, we provide a common interface to existing models, and reimplement models that were not compatible. In RL, Garage (previously known as rllab) [15], baselines [28], and RLlib [14] are three popular libraries that provide out-of-the-box RL algorithms. The first two are coded in TensorFlow, while the latter is built on PyTorch. As for the environments, these model-based frameworks define their own standard which do not fit our modular framework. The main reasons being that learning algorithms are dependent of low-level concepts such as the environment and policies (i.e. models) making a possible integration harder.

Recently, two new Python robot frameworks have been introduced: PyRobot [29] and PyRep [30]. The former provides a lightweight interface built on top of Gazebo-ROS [31, 32] with a focus on robotic manipulation and navigation, while the latter provides a Python wrapper around the V-REP simulator [33]. As our framework, they aim to be beginner-friendly but are mainly focused on the robotic application instead of being a complete robot learning framework. They can be better compared to a simulator such as PyBullet or MuJoCo.

3   Proposed Framework

PyRoboLearn (PRL) is designed to maximize modularity, flexibility, simplicity, and generality. Our first choice is the programming language. We choose Python[2] because of its simplicity to prototype new ideas, a fast learning curve, a huge amount of available libraries, and the ability to interact with the code. We also use PyTorch and Numpy for our learning models and algorithms. PyTorch is chosen because of its Pythonic nature, modularity and popularity in research.

Regarding PyRoboLearn architecture, we abstract each robot learning concept, adopt a modular programming approach, minimize the modules coupling, and favor composition over inheritance [34] to increase the flexibility [35]. Abstraction aims at identifying and abstracting different concepts into objects, and building high-level concepts on top of low-level ones. Modularity separates these concepts into independent and interchangeable building blocks that represent or implement a particular functionality. Composition combines different modules and thus different functionalities into a single one. Coupling measures how different modules depend on each other. A high coupling between two modules means they cannot work in a stand-alone fashion, while a low coupling means they depend on abstractions instead of concretions [20]. The aforementioned notions increase the framework flexibility while facilitating the reuse and integration of the various modules.

PRL functionalities cover seven main axes: simulators, worlds, robots, learning paradigms, interfaces, learning models, and learning algorithms. Each of these components is described next. An overview of the framework is depicted in Fig. 1.
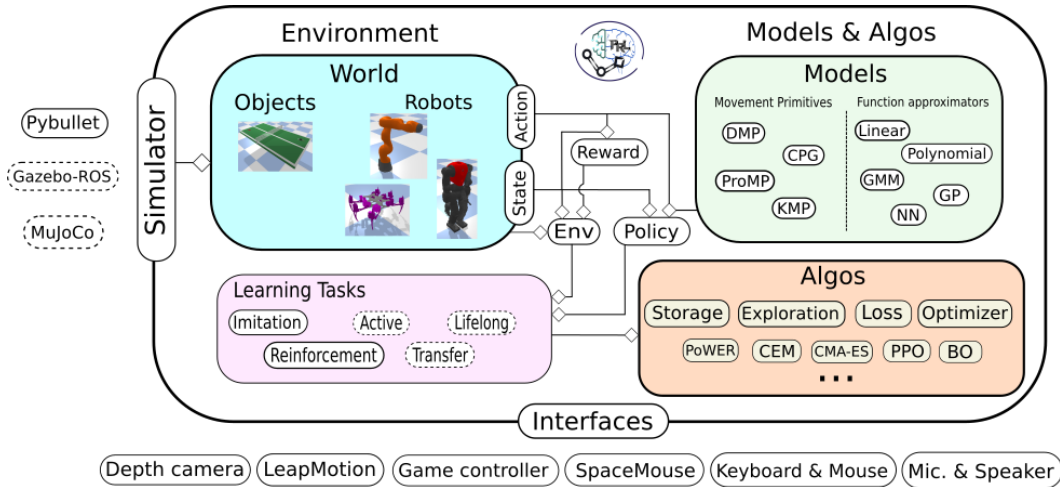

Figure 1: Overview of the *PyRoboLearn* architecture. Dashed bubbles are possible additions.

---

[2]PyRoboLearn works in Python 2.7, 3.5, and 3.6, and has been tested on Ubuntu 16.04 and 18.04. While the support for Python 2.7 will end in 2020, many libraries used in robotics still depend on it.

3.1 Simulators

The first axis is the specification of the simulator. Different simulators have been proposed: Gazebo [31] (with ROS [32]), V-REP/PyRep [33, 30], Webots [36], Bullet/PyBullet [37, 6], and MuJoCo [21] (the most popular). We choose to work with PyBullet as it works in Python, and it is free and open-source. To avoid our code to be fully dependent on it, we provide an abstract interface that lies between the simulator and our framework such that any other simulators can inherit, allowing for easy integration in the future (e.g., MuJoCo or Gazebo through ROS). Due to its popularity, gym [2] was also wrapped inside PRL, to make it suitable with our framework in RL scenarios.

3.2 Worlds and Robots

Once a simulator is provided, a world where robots and objects can interact is required. Only the world and robot instances interact with the simulator. The PyBullet simulator permits to load meshes in the world but does not provide any tool to generate terrains. This missing feature is important for robot locomotion tasks. We address this issue by providing tools to automatically generate height maps, which are subsequently used to produce meshes that are then loaded into the simulator.

Robots are the active agents in our world, and more than 60 robots are provided in PRL. All inherit from a main robot class and are split into different categories: manipulators, legged robots, wheeled robots, UAVs, among others. Each of these categories is then divided in further subcategories. For instance, the legged robot class is inherited by classes representing biped, quadruped, and hexapod robots. Kinematic and dynamic functions allowing for motion and torque control are provided through the main interface. We access online the URDF files of more than 60 robots and implement their corresponding classes through our framework (see Fig. 2). This unified structure of robotic platforms allows users to experiment rapidly with learning paradigms such as transfer learning.
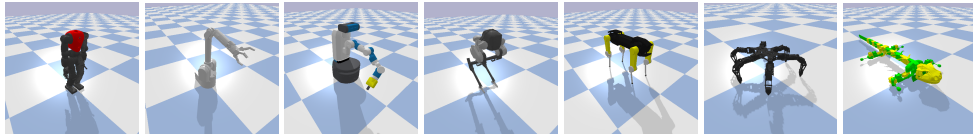

Figure 2: 7 of the 60+ available robots in PRL: manipulators, wheeled and legged robots.

3.3 Tasks and Learning Paradigms

Robot learning [1] is usually understood as the intersection of machine learning and robotics. This is divided into different learning paradigms according to different scenarios. The main categories are imitation learning (IL) and reinforcement learning (RL). IL [17] envisions a teacher demonstrating to an agent how to reproduce a task through few examples. RL [18, 19] conceives an agent that learns to perform a task by maximizing a total reward while interacting with its environment. Other paradigms include transfer learning [38, 39] where the knowledge acquired by an agent while solving a problem is transferred to solve a similar problem, and active learning [40] where an agent interacts with the user by querying new information about the task (e.g. demonstrations). While the foregoing approaches address different learning problems, they all share some common features (e.g. states, actions, policies and environments) which are conceptualized and abstracted in PRL. Similarly, their differences are introduced without loss of scalability through modules and composition. Additionally, different learning paradigms are evaluated using different metrics. To represent a learning paradigm, we describe it by an abstract `Task` class which encapsulates the environment and the policy. A task inheriting from this class is then formulated for each paradigm as needed.

3.4 Interfaces and Bridges

In IL, two predominant techniques are used to teach a robot a skill in order to perform a task: *teleoperation* and *kinesthetic teaching*. Teleoperation consists of commanding a robotic platform through a controller (from a remote location or in a virtual environment), while kinesthetic teaching considers a human guiding physically the robot (or a part of it) to perform the task. While the former is popular for its simplicity and use in simulation, it becomes difficult to use for robots with complex structures (such as humanoid robots). Recent advances in computer vision allow us to use cameras

Interface	Instances
PC hardware	keyboard and mouse, SpaceMouse
audio/speech	speech recognition, synthesization, and translation
camera	webcam, asus-xtion, kinect, openpose
game controllers	Xbox, Playstation
sensors	Leap Motion, Myo Armband

Table 1: The various interfaces in PyRoboLearn

to control the robot, however the human-robot kinematic mapping remains a challenge. As for the latter, it has been hardly applied in simulations due to the lack of tools and haptic feedback.

In PRL, several *interfaces* have been implemented to enable the user to interact with the world and its objects, including robots. The implemented interfaces are resumed in Table 1. These tools are useful for different tasks and scenarios, especially in imitation and active learning. All the interfaces are completely independent of our framework and can be used in other applications. They act as containers for the collected data from the corresponding hardwares. *Bridges* connect an interface with a component, such as the world or an element in that world. For instance, a game controller interface permits to get data from the hardware, process it, and store it. The bridge can then map a specific controller event to a robot action. Moving a joystick up could mean to move a wheeled robot forward, or make a UAV robot ascend in the air. This separation of interfaces and bridges allows the user to only implement the necessary bridge without reimplementing the associated interface.

3.5 Learning Models

We implement several learning models in our framework through a modular approach. Learning models are characterized by (hyper-)parameters that are optimized through a training algorithm. All the implemented models are decoupled from PRL and can be used in other frameworks. To provide a better integration with the various modules in PRL, we build two abstraction layers on top of the models. The first layer extends the models by receiving any created state and action module as inputs and/or outputs (in addition to normal Numpy arrays or Pytorch tensors). The second layer focuses on particular instances of these extended models, for example, a policy that receives as input the states and outputs the actions, or a state value-function approximator which receives a state as input and outputs a scalar value. In our framework the learning models are separated from the learning algorithms (see 3) to avoid the models to be dependent on the training approach.

The provided learning models are reported in Table 2.

Type	Instances
Movement Primitives	central pattern generators (CPGs) [41], dynamic movement primitives (DMPs) [42], probabilistic movement primitives (ProMPs) [43], and kernelized movement primitives (KMPs) [44]
Function Approximators	linear and polynomial models, Gaussian processes (GPs) [45, 12], Gaussian mixture models (GMMs) [46], and deep neural networks (DNNs) [47, 11]

Table 2: The various models in PyRoboLearn

3.6 Learning Algorithms

RL algorithms [18] depend on the structure of the environments/tasks, policies and models. Therefore, dependencies among them are unavoidable. We implement them in a modular way to stay consistent with PRL. To illustrate the modularity, let us consider the model-free PPO algorithm [48]. This algorithm has a lot in common with many other model-free on-policy algorithms but uses a different loss and action-space exploration strategy. This is often not exploited in current frameworks. In PRL, the loss can be redefined, any arithmethic operations can be performed on these loss instances, and provided at runtime to the PPO algorithm (through composition) without loss of generality. This results in faster experimentation to compare loss functions.

Because of the modular programming approach we undertake, we provide a different module for every concept, including the loss and exploration strategy. Moreover, as we favor composition over inheritance, we can parametrize the PPO algorithm with these modules, resulting in a more flexible framework that allows users to modify the algorithms and experiment with a wider range of com-
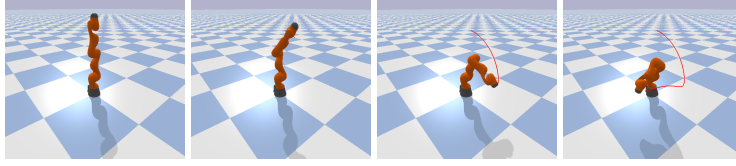
Figure 3: Reproduction of a trajectory learned from mouse-generated demonstrations using a DMP

binations. The learning algorithms available in PRL include Bayesian optimization, evolutionary algorithms, model-free (on-/off-) policy search, among others.

4 Experiments

In order to show the functionality of our framework for robot learning, we demonstrate three use cases; an IL scenario, an RL task, and a scenario which combines these two approaches to show the flexibility of our framework.

4.1 Imitation Learning Task: Trajectory Tracking

The goal is to reproduce a demonstrated trajectory with IL using a *dynamic movement primitive* (DMP) model on a KUKA-LWR robot. The trajectories are demonstrated using the mouse interface (see Fig. 3). Both the training and reproduction phases can be watched on the PRL Youtube channel (see Section 6). The associated pseudo-code is given below in Algorithm 1.

Algorithm 1 Trajectory tracking with imitation learning

```
1: sim = Simulator()
2: world = BasicWorld(sim)
3: robot = world.load('robot_name_or_class')
4: state = PhaseState()
5: action = JointPositionAction(robot)
6: env = Env(world, state)
7: policy = Policy(state, action)
8: recorder = Recorder(state, action, rate)
9: interface = MouseKeyboardBridge(world)
10: task = ILTask(env, policy, interface, recorder)
11: task.record(signal_from_interface=True)
12: task.train()
13: task.test()
```

Algorithm 1 illustrates the various building blocks and how they encapsulate each other. In this example, we first create an instance of the simulator, and then define a world in it. After this, a robot is loaded into the world. Next, we define the states and actions that are given to the policy and the environment. As we are in an IL setting, we need to collect and record the demonstrated data through the use of a recorder. We provide the trajectories using the mouse interface. Finally, an IL task can be fully defined with all the previous components. The step left is to train the policy using the demonstrated trajectory and reproduce the policy. The last three lines can be replaced by `task.run(signal_from_interface=True)` where the argument specifies that an event from the interface will send a signal to indicate when to record the data, train, and test the policy. Note that changes in the simulator, world, robot, state, action, policy, and/or interface would not affect the rest of the code due to the abstractions and modularity of our framework. This confirms the flexibility of PyRoboLearn.

4.2 Reinforcement Learning Task: Locomotion

We now test our framework on an RL locomotion task with the Minitaur quadruped robot using *central pattern generators* where the hyperparameters are trained using *Bayesian optimization* [49] (see Fig. 4 and associated Youtube channel). The associated pseudocode is given in Algorithm 2.

Algorithm 2 shows that we can easily combine different rewards together. This feature is also available for states, actions and other components in the framework. It is worth noting that some

6

Figure 4: Walking robot using RL

Figure 5: Cartpole task solved through IL and RL

Algorithm 2 Locomotion RL task

```
1: sim = Simulator()
2: world = BasicWorld(sim)
3: robot = world.load('robot_name')
4: states = PhaseState()
5: actions = JointPositionAction(robot)
6: rewards = a * Reward1(state) + b * Reward2(state)
7: env = Env(world, states, rewards)
8: policy = Policy(states, actions)
9: task = RLTask(env, policy)
10: algo = RLAlgo(task, policy, hyperparameters)
11: results = algo.train(num_steps, num_episodes)
12: final_reward = algo.test(num_steps)
```
---

states may depend on the considered robotic platform. For instance, a `CameraState` can only be applied to a robot that has at least one onboard camera. If the robot does not have any, this state will be empty but the code will still run smoothly given that the policy and reward can take care of such situations. Nevertheless, our framework keeps its simplicity and flexibility in this example. This modularity and generality also applies to other components in the framework like RL algorithms where we can change the loss function. These RL tasks can also be combined with previous IL tasks as shown in the next section.

Regarding the use of OpenAI-Gym within PRL, our wrapper avoids coding the lines 1 until 6 in the Algorithm 2, and line 7 can be replaced by `env = wrapped_gym.make('env_name')`. The states and actions given to the policy can then be accessed via `env.state` and `env.action`.

4.3 Imitation and Reinforcement Learning

In this example, we illustrate how we can combine different learning paradigms. Here, IL is first used to initialize a policy, which is then fine-tuned by an RL approach. We test this feature on the cartpole task where the goal is to lift up a pole, initially facing downwards, by moving the cart left to right. We first provide few demonstrations using the mouse in slow-motion mode, and then train the policy using the obtained dataset. The policy is then refined using the PoWER reinforcement learning algorithm [50] where the initialization plays an important role (see Fig. 5 and associated Youtube channel). Algorithm 3 shows how an RL task can easily be defined after an IL task. In this pseudo-code, we assume that the simulator, world, robot, states, actions, rewards, policy and environment were created before, as done in Algorithm 2.

Algorithm 3 IL task followed by a finetuning using RL

```
1: il_task = ILTask(env, policy, interface, recorder)
2: il_task.record(signal_from_interface=True)
3: il_task.train()
4: rl_task = RLTask(env, policy)
5: initial_reward = rl_task.run(num_steps)
6: algo = RLAlgo(rl_task, policy, hyperparameters)
7: results = algo.train(num_steps, num_episodes)
8: final_reward = algo.test(num_steps)
```
---

5 Discussion

The proposed framework suffers from few shortcomings for *production* development. The first one is the programming language that makes it not suitable for real-time tasks. A Python wrapper around

the simulator has to be provided to be used with our framework (by implementing the `Simulator` interface). Nevertheless, based on its fast-learning curve, huge number of available libraries, its acceptance in the research community, its ability to fast-prototype ideas, and its interactive mode, we think that this trade-off is worthy.

The second limitation is that PRL cannot be used currently with real hardware to easily transfer a piece of code working in simulation to real robots, or record trajectories from the real hardware. We plan to provide a ROS integration in the future, where just the first line in previous algorithms (Algorithm 1 and 2) would need to be replaced by `sim = RBDL_ROS()`. However, a partial implementation of using ROS with Bullet called `BulletROS` is provided in PRL, but safety issues on the real system has not been considered yet.

6   Conclusion

In this paper, we presented a first version of our generic Python framework for robot learning practitioners. Design decisions made such as the consistent approach to abstract each concept, the preference for composition over inheritance, and the minimization of couplings, renders our library highly modular, flexible, generic and easily updatable. The implementation of different paradigms, as well as the availability of different learning models, algorithms, robots and interfaces, allows to prototype faster different ideas and compare them with previous approaches.

It is our hope that the proposed framework will be useful to researchers, scholars and students in robot learning, and will be a step towards better benchmarks and reproducibility [51]. The link to the Github repository, documentation, examples, and videos are available through the main website `https://robotlearn.github.io/pyrobolearn/`. PRL is currently released under the GPLv3 license, and has been tested on Ubuntu 16.04 and 18.04 with Python 2.7, 3.5, and 3.6 [3].

Future work will address the various aforementioned shortcomings. In addition, we plan to continue to provide other simulator APIs, learning paradigms, algorithms, and robotic tools such as state estimators and controllers. Finally, we plan to reproduce other state-of-the-art experiments, provide benchmarks, and present the obtained results on an interactive centralized website.

References

[1] J. Peters, D. D. Lee, J. Kober, D. Nguyen-Tong, J. A. Bagnell, and S. Schaal. Robot learning. In *Springer handbook of robotics*, pages 357–398. Springer, 2016.

[2] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. OpenAI Gym. *arXiv preprint arXiv:1606.01540*, 2016.

[3] I. Zamora, N. G. Lopez, V. M. Vilches, and A. H. Cordero. Extending the OpenAI Gym for robotics: a toolkit for reinforcement learning using ROS and Gazebo. *arXiv preprint arXiv:1608.05742*, 2016.

[4] O. Klimov and J. Schulman. Roboschool: Open-source software for robot simulation, integrated with openai gym. `https://github.com/openai/roboschool`, 2017.

[5] Y. Tassa, Y. Doron, A. Muldal, T. Erez, Y. Li, D. d. L. Casas, D. Budden, A. Abdolmaleki, J. Merel, A. Lefrancq, et al. Deepmind control suite. *arXiv preprint arXiv:1801.00690*, 2018.

[6] E. Coumans and Y. Bai. Pybullet, a Python module for physics simulation for games, robotics and machine learning. *GitHub repository*, 2016.

[7] B. Ellenberger. Pybullet gymperium. `https://github.com/benelot/pybullet-gym`, 2018.

[8] D.-W. Huang, G. Katz, J. Langsfeld, R. Gentili, and J. Reggia. A virtual demonstrator environment for robot imitation learning. In *TePRA*, pages 1–6. IEEE, 2015.

---

[3]While the support for Python2.7 will end in 2020, some simulators such as Gazebo-ROS still have old libraries that are dependent on Python 2.7. The framework was designed to account for this problem.

[9] E. Berger, H. B. Amor, D. Vogt, and B. Jung. Towards a simulator for imitation learning with kinesthetic bootstrapping. In *SIMPAR*, pages 167–173, 2008.

[10] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun. Carla: An open urban driving simulator. *arXiv preprint arXiv:1711.03938*, 2017.

[11] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017.

[12] J. R. Gardner, G. Pleiss, D. Bindel, K. Q. Weinberger, and A. G. Wilson. GPyTorch: Blackbox matrix-matrix gaussian process inference with gpu acceleration. In *NeurIPS*, 2018.

[13] I. Kostrikov. Pytorch implementations of reinforcement learning algorithms. https://github.com/ikostrikov/pytorch-a2c-ppo-acktr, 2018.

[14] E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, J. Gonzalez, K. Goldberg, and I. Stoica. Ray rllib: A composable and scalable reinforcement learning library. *arXiv preprint arXiv:1712.09381*, 2017.

[15] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel. Benchmarking deep reinforcement learning for continuous control. In *ICML*, pages 1329–1338, 2016.

[16] E. Pignat and S. Calinon. Pbdlib: a python library for robot programming by demonstration. https://gitlab.idiap.ch/rli/pbdlib-python, 2017.

[17] A. G. Billard, S. Calinon, and R. Dillmann. Learning from humans. In *Springer handbook of robotics*, pages 1995–2014. Springer, 2016.

[18] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[19] M. P. Deisenroth, G. Neumann, J. Peters, et al. A survey on policy search for robotics. *Foundations and Trends in Robotics*, 2(1–2):1–142, 2013.

[20] R. C. Martin. Design principles and design patterns. *Object Mentor*, 1(34):597, 2000.

[21] E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In *Intl. Conf. on Intelligent Robots and Systems*, pages 5026–5033, 2012.

[22] S. Shah, D. Dey, C. Lovett, and A. Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. *arXiv preprint arXiv:1705.05065*, 2017.

[23] L. Fan, Y. Zhu, J. Zhu, Z. Liu, O. Zeng, A. Gupta, J. Creus-Costa, S. Savarese, and L. Fei-Fei. Surreal: Open-source reinforcement learning framework and robot manipulation benchmark. In *Conference on Robot Learning*, pages 767–782, 2018.

[24] F. Xia, A. R. Zamir, Z.-Y. He, A. Sax, J. Malik, and S. Savarese. Gibson Env: real-world perception for embodied agents. In *CVPR*. IEEE, 2018.

[25] A. Raffin, A. Hill, R. Traoré, T. Lesort, N. Díaz-Rodríguez, and D. Filliat. S-RL toolbox: Environments, datasets and evaluation metrics for state representation learning. *arXiv preprint arXiv:1809.09369*, 2018.

[26] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in Python. *JMLR*, 12 (Oct):2825–2830, 2011.

[27] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 265–283, 2016.

[28] A. Hill, A. Raffin, M. Ernestus, A. Gleave, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu. Stable baselines. https://github.com/hill-a/stable-baselines, 2018.

[29] A. Murali, T. Chen, K. V. Alwala, D. Gandhi, L. Pinto, S. Gupta, and A. Gupta. Py-robot: An open-source robotics framework for research and benchmarking. *arXiv preprint arXiv:1906.08236*, 2019.

[30] S. James, M. Freese, and A. J. Davison. PyRep: Bringing V-REP to deep robot learning. *arXiv preprint arXiv:1906.11176*, 2019.

[31] N. Koenig and A. Howard. Design and use paradigms for Gazebo, an open-source multi-robot simulator. In *Intl. Conf. on Intelligent Robots and Systems*, pages 2149–2154, 2004.

[32] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. ROS: an open-source robot operating system. In *ICRA workshop on open source software*, page 5. Kobe, Japan, 2009.

[33] E. Rohmer, S. P. Singh, and M. Freese. V-REP: A versatile and scalable robot simulation framework. In *Intl. Conf. on Intelligent Robots and Systems*, pages 1321–1326, 2013.

[34] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[35] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[36] O. Michel. Cyberbotics ltd. webots: professional mobile robot simulation. *IJARS*, 1(1):5, 2004.

[37] E. Coumans et al. Bullet physics library. *Open source: bulletphysics.org*, 15(49):5, 2013.

[38] S. J. Pan, Q. Yang, et al. A survey on transfer learning. *TKDE*, 22(10):1345–1359, 2010.

[39] M. E. Taylor and P. Stone. Transfer learning for reinforcement learning domains: A survey. *JMLR*, 10(Jul):1633–1685, 2009.

[40] B. Settles. Active learning literature survey. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 2009.

[41] A. J. Ijspeert. Central pattern generators for locomotion control in animals and robots: a review. *Neural networks*, 21(4):642–653, 2008.

[42] A. J. Ijspeert, J. Nakanishi, H. Hoffmann, P. Pastor, and S. Schaal. Dynamical movement primitives: learning attractor models for motor behaviors. *Neural computation*, 25(2):328–373, 2013.

[43] A. Paraschos, C. Daniel, J. Peters, and G. Neumann. Using probabilistic movement primitives in robotics. *Autonomous Robots*, 42(3):529–551, 2018.

[44] Y. Huang, L. Rozo, J. Silvério, and D. G. Caldwell. Kernelized movement primitives. *The International Journal of Robotics Research*, 38(7):833–852, 2019.

[45] C. E. Rasmussen and C. K. Williams. *Gaussian process for machine learning*. MIT press, 2006.

[46] S. Calinon. *Robot programming by demonstration: a probabilistic approach*. EPFL Press, 2009.

[47] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[48] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[49] T. G. authors. GPyOpt: A Bayesian optimization framework in python. http://github.com/SheffieldML/GPyOpt, 2016.

[50] J. Kober and J. R. Peters. Policy search for motor primitives in robotics. In *NeurIPS*, pages 849–856, 2009.

[51] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger. Deep reinforcement learning that matters. In *AAAI Conference on Artificial Intelligence*, 2018.

Appendix A: Comparison with other Environment Frameworks

A table summarizing parts of the different characteristics of current robot learning frameworks that provide environments is depicted in Table 3. Note that MuJoCo [21] is not open-source, requires a license, and depending on that last one might not be free. Also note that while the support for Python 2.7 will end in 2020, some simulators such as Gazebo-ROS and some libraries are still dependent on Python 2.7.

Name	OS	Python	Simulator	Paradigm	Robot	Problem
Open-AI Gym [2]	OSX, Linux	2.7, 3.5	MuJoCo	RL	3D chars	Manip./Loco.
Gym-Gazebo [3]	Ubuntu 18.04	3	Gazebo+ROS	RL	<5 robots	Manip./Nav.
DeepMind Control Suite [5]	Ubuntu 14.04/16.04	2.7, 3.5	MuJoCo	RL	3D chars	Loco./Control
Roboschool [4]	OSX, Ubuntu/Debian	3	Bullet	RL	3D chars	Loco./Control
Pybullet Gym [6, 7]	OSX, Linux, Windows	2.7, 3.5	PyBullet	RL	3d chars/Atlas	Manip./Loco./Control
GibsonEnv [24]	Ubuntu	3.5	Bullet	PL/RL	3D chars/5 robots	Perception/Nav.
Airsim [22]	Linux, Windows	3.5+	Unreal Engine/Unity	IL/RL	AV	Nav.
Carla [10]	Ubuntu 16.04+, Windows	2.7, 3.5	Unreal Engine	IL/RL	AV	Nav.
Surreal Robotics Suite [23]	OSX, Linux	3.5, 3.7	MuJoCo	IL/RL	Baxter/Sawyer	Manip.
S-RL Toolbox [25]	N/S	3.5+	PyBullet	RL/SRL	Kuka/OmniRobot	Manip./Nav.
PyRoboLearn	Ubuntu 16.04/18.04	2.7, 3.5, 3.6	Agnostic (PyBullet)	IL/RL	60+	Manip./Loco./Control

Table 3: Comparisons between different robot learning frameworks that provide environments. PL stands for perception learning, SRL for state representation learning, and AV for autonomous vehicles.