Dragonfly: a modular deep reinforcement learning library

A Preprint

J. Viquerat*
MINES Paristech, CEMEF
PSL - Research University
jonathan.viquerat@mines-paristech.fr

P. Garnier
MINES Paristech, CEMEF
PSL - Research University

A. Bateni
MINES Paristech, CEMEF
PSL - Research University

E. Hachem
MINES Paristech, CEMEF
PSL - Research University

January 29, 2025

Abstract

dragonfly is a deep reinforcement learning library focused on modularity, in order to ease experimentation and developments. It relies on a `json` serialization that allows to swap building blocks and perform parameter sweep, while minimizing code maintenance. Some of its features are specifically designed for CPU-intensive environments, such as numerical simulations. Its performance on standard agents using common benchmarks compares favorably with the literature.

***K*eywords** Deep reinforcement learning, Library

1 Introduction

Deep Reinforcement Learning (DRL) has emerged as a transformative field in artificial intelligence, blending deep learning and reinforcement learning to create powerful decision-making systems [1]. Recent advances have seen DRL algorithms achieving superhuman performance in complex games [2, 3], robotic control [4], and strategic planning [5]. Still, challenges persist, and every research endeavour may require significant development and testing efforts, as well as extensive logging and results comparison.

In this contribution, we introduce dragonfly, a deep reinforcement learning library aimed at making prototyping and implementation of new features fast and easy. Contrarily to implementations such as cleanrl [6] or stable-baselines3 [7], which are focused on scalable, single-file implementations of agents, dragonfly relies on a highly-modular pattern. Its performance level is ensured by constant performance evaluation of the agents against standard benchmarks, making it a reliable production tool. The library relies on tensorflow 2 as a backend for the design of neural network architectures.

*Corresponding author

2 Library architecture

2.1 General architecture

The global architecture of the library is summed up in figure 1. The training is driven by a **`.json`** file that provides all the informations required to set the different classes and train the agent. The training procedure is handled by a **`trainer`** object, which in turn initializes the **`environment`** and the **`agent`**, as well as several other objects helpful for the logging, rendering, etc. These elements rely on lower level objects such as networks, optimizers and losses.

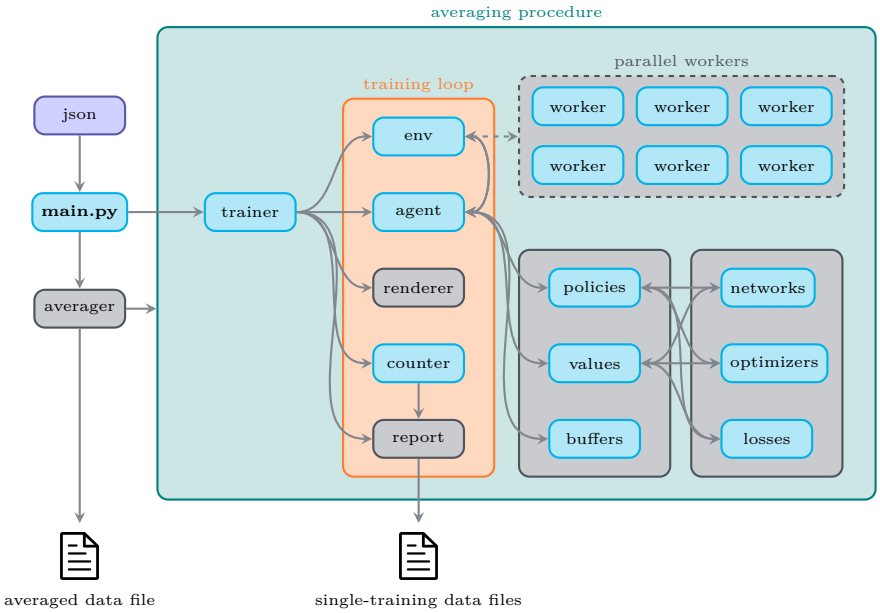


Figure 1: **General architecture of the library.**

2.2 Implementation details

The backbone of the present library is composed of three components: (i) a **`.json`** serialization of the different components, (ii) the use of **`SimpleNamespace`** instances to store the parameters of each object, and (iii) a factory pattern.

The **`.json`** parsing relies on the standard **`json`** module, and exploits the **`SimpleNamespace`** iterable-based constructor to provide a lightweight and easy-to-use class-like representation of the serialized inputs. Finally, the **`factory`** class allows to generate objects of a desired type using a string representation:

```
class factory:
   def __init__(self):
      self.keys = {}

   def register(self, key, creator):
      self.keys[key] = creator

   def create(self, key, **kwargs):
      creator = self.keys.get(key)
      if not creator:
         try:
            raise ValueError(key)
         except ValueError:
            error("factory", "create", "Unknown key provided: "+key)
            raise

      return creator(**kwargs)
```

This pattern adds a lot of flexibility to the library, as swapping building blocks within a given algorithm only requires to modify the input **.json** configuration file. For each object type, (agents, losses, etc), a factory is instantiated, and the different corresponding classes are registered using the **register** member with a key/value pair:

```
agent_factory = factory()

agent_factory.register("a2c",  a2c)
agent_factory.register("ppo",  ppo)
agent_factory.register("dqn",  dqn)
agent_factory.register("ddpg", ddpg)
agent_factory.register("td3",  td3)
agent_factory.register("sac",  sac)
```

Then, generating an object of a given type (here an agent) from a string representation is performed using the **create** member:

```
agent = agent_factory.create(agent_parameters.type,
                             spaces         = environment.spaces,
                             n_environments = mpi.size,
                             memory_size    = memory_size,
                             parameters     = agent_parameters)
```

3 Features

3.1 Agents

Standard agents, such as PPO, DQN, DDPG, TD3 or SAC, are implemented in the library. The agent training is based on different **trainer** types, based on the on-policy or off-policy nature of the agent. The **trainer** is a cornerstone structure of the library, as it is used to instantiate the agent and the environment, as well as several other objects used to train the agent and log its performances. As for some other structures of the library, this wrapper provides another layer of flexibility in the exploration of DRL algorithms.

Each agent generates some **policy** and **value** instances, the latter spawning adequate **network**, **loss** and **optimizer** instances based on the description of the **json** serialization. Additional types may be required, such as **return** and **termination** types, depending on the agent type.

3.2 Environment

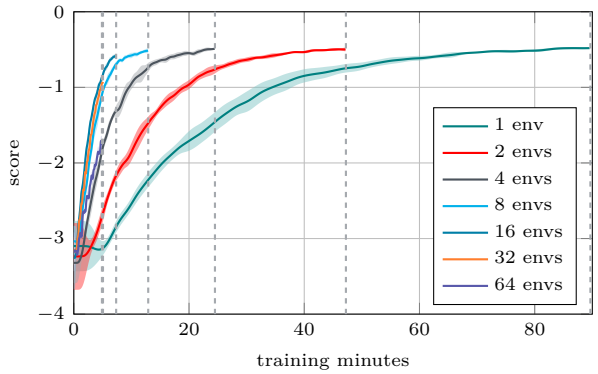
The **environment** class consists in a wrapper around (i) a **spaces** class, and (ii) a set of **worker** instances. The **spaces** instance is devoted to handling the relation with the actual environment, determining the size and shape of actions and observations, and applying transformations to the latter if needed. Each **worker** instance wraps an actual environment instance, and interacts with the upper layer. Doing so, considering a single or multiple environments is made transparent to the **agent** instance. The transformations applied to the actions and observations can be directly parameterized through the **.json** file, and optional arguments can also be passed transparently through the different layers to the actual environment.

3.3 Buffer

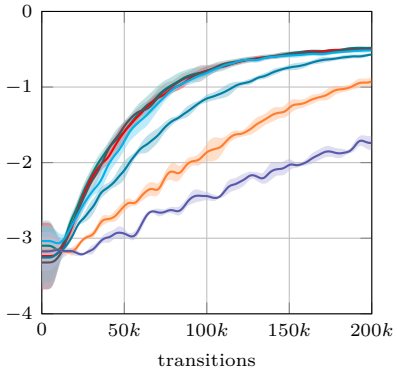
Buffering is made flexible using multiple layers of buffers to store data on-the-fly from the parallel environments. Right before training, parallel informations are collected from the parallel buffer and reshaped to be stored in a larger memory buffer, making them avaiable for sampling during the training phase. To limit the memory footprint, a ring buffer structure is used. The buffer structures are initialized using dict structures, so the same buffer class can be initialized using different lists of key/size pairs, and therefore be used with different agents.

3.4 Bootstrapping for parallel environments

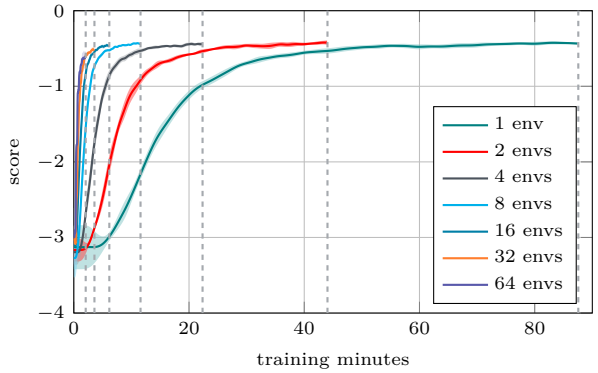
In the context of CPU-intensive environments (for example, when the environment consists in a finite element resolution of a physical problem), the use of parallel environments can rapidly become a necessity. Yet, for methods such as PPO, the on-policiness assumption may be broken when using too many simultaneous environments (for example, if the agent update requires four full trajectories, the on-policiness assumption is broken if more than four environments are unrolled simultaneously). In this context, a bootstrapping termination technique can be used to mimic the on-policiness assumption beyond the theoretical limit. As presented in [8], and reproduced here in figure 2, this simple method allows to use 4 to 8 times more parallel environments than the vanilla approach. For additional details, the reader is referred to [8].



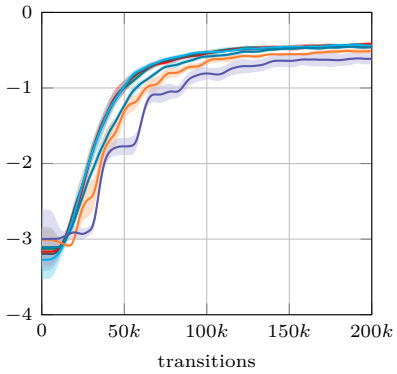
(a) Score against walltime, no bootstrapping



(b) Score against transitions, no bootstrapping



(c) Score against walltime, parallel bootstrapping



(d) Score against transitions, parallel bootstrapping

Figure 2: **Score curves obtained for different number of parallel environments** on the **`shkadov-v0`** environment from the BEACON benchmark [9]. The agent is updated with 4 full trajectories. (Top) With no bootstrapping, the learning rate and the final performance decrease significantly beyond 4 parallel environemnts (Bottom) With bootstrapping, performance is maintained exactly up to 16 parallel environments, and with a very slight performance decrease up to 32 parallel environments. Reproduced from [8].

3.5 State representation learning

Representation learning or feature learning refers to methods providing significative representations of raw data fulfilling specified goals. This field encompasses methods such as principal component analysis (PCA), clustering techniques, auto-encoders (AE), etc. In most DRL-based control cases, the observations provided by the environment may be noisy, incomplete, high-dimensional and/or highly correlated, which may lead to subpar control performance from the agent. In this context, state

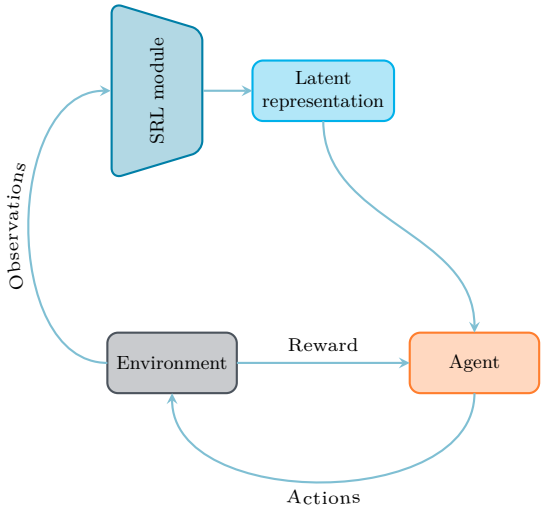


Figure 3: **State representation learning setup in the DRL learning loop.** The traditional observation feedback is replaced by a modified feedback loop, in which observations are transformed in a latent representation before being fed to the agent.

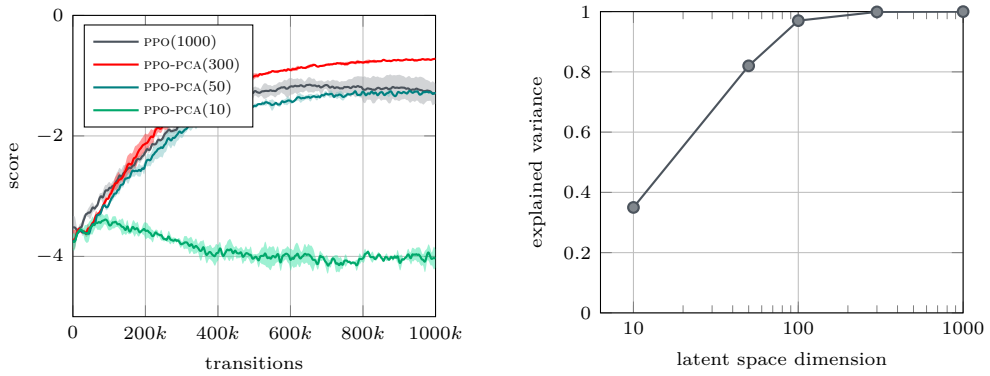
representation learning aims to process these observations upstream of the DRL agent, by learning a modified or a low-dimensional representation of the original data, called latent representation (see figure 3).

In its current state, the library integrates a PCA and an auto-encoder modules, that can be seamlessly integrated to existing agents. The coupling of an agent with state representation learning can be split in two steps. First, a warmup phase is defined during which random actions are taken by the agent, while the observations are collected for the training of the SRL module. No update of the agent is performed during this phase. Once a pre-defined amount of samples have been collected, the SRL module is updated, and the training phase of the agent starts. During this second phase, the agent is fed with SRL-processed observations as input. This on-the-fly process is easy to use, as it does not require any pre-processing of the environment, nor any third-party module to train the SRL module. Of course, the trained parameters of the SRL module can be saved and re-used for future trainings if needed.

In figure 4a, a performance comparison with and without the SRL module is shown on the **`shkadov-v0`** environment from the BEACON benchmark [9]. The entire set of observations is provided to the agent as a vector of size 1000. In the regular case, the PPO agent directly processes the input vector, while in the SRL case, a PCA representation of these observations is built on-the-fly, and then fed to the agent. One sees that the PPO + SRL agent with a latent space of size 300 overperforms the standard PPO agent. It is also observed that too low latent space dimensions lead to sub-par performance, or even no learning at all. The optimal latent space dimension is obtained by looking for the lowest dimension for which the explained variance remains close to 1, as is shown in figure 4b. To further illustrate the interest of this method, a hotmap of the observations standard deviation averaged over one episode is shown in figure 4c. As can be seen, the PCA method naturally builds latent features with large variance, which can be advantageous in the context of DRL as it leads to a condensed representation of the important variations in the observations of the environment, while filtering out the low-variation features.

3.6 Separable environments

Learning a proper control strategy can become nearly impossible in the case of high-dimensional action spaces, leading to excessive sample requirements. Yet, some environments presenting adequate separability can be reformulated as problems of lower action dimensionality, leading to an efficient learning. This approach was presented in [10], and is implemented in the library as a specific **`trainer`** class, handling the mapping from 1 environment with n_{act} actions to n_{act} environments with 1 action.

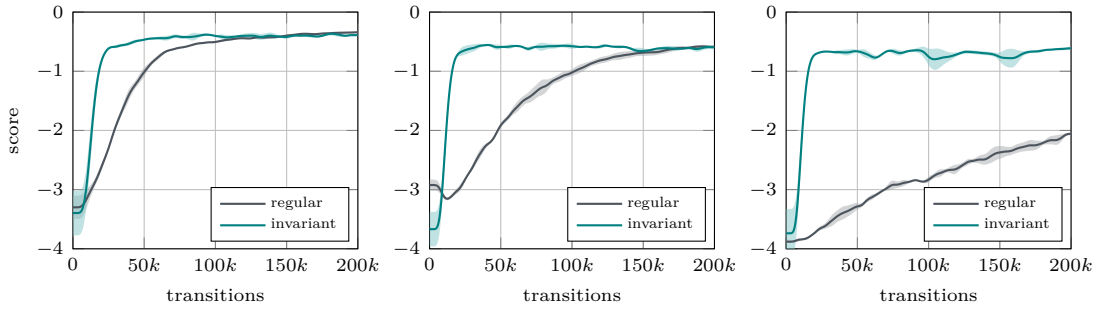


(a) Score curves

(b) Explained variance

(c) Hotmap of the observations standard deviation averaged over one episode without (top) and with (bottom) the PCA module for a latent space dimension of 300.

Figure 4: **Comparing PPO with PPO-PCA on the `shkadov-v0` environment**. (Top left) The PPO-PCA overperforms the standard PPO algorithm for an observation vector of size 1000 corresponding to the entire information of the considered environment. Performance varies based on the chose latent space dimension. (Top right) Adequate latent space dimension can be found by evaluating the explained variance of the PCA representation chosen, the optimal corresponding to the lowest dimension for which the explained variance remains close to 1. (Bottom) The hotmap of the observations standard deviation averaged over one episode without and with the PCA module shows that the PCA module generates observations with high variance.



(a) 5 dimensions

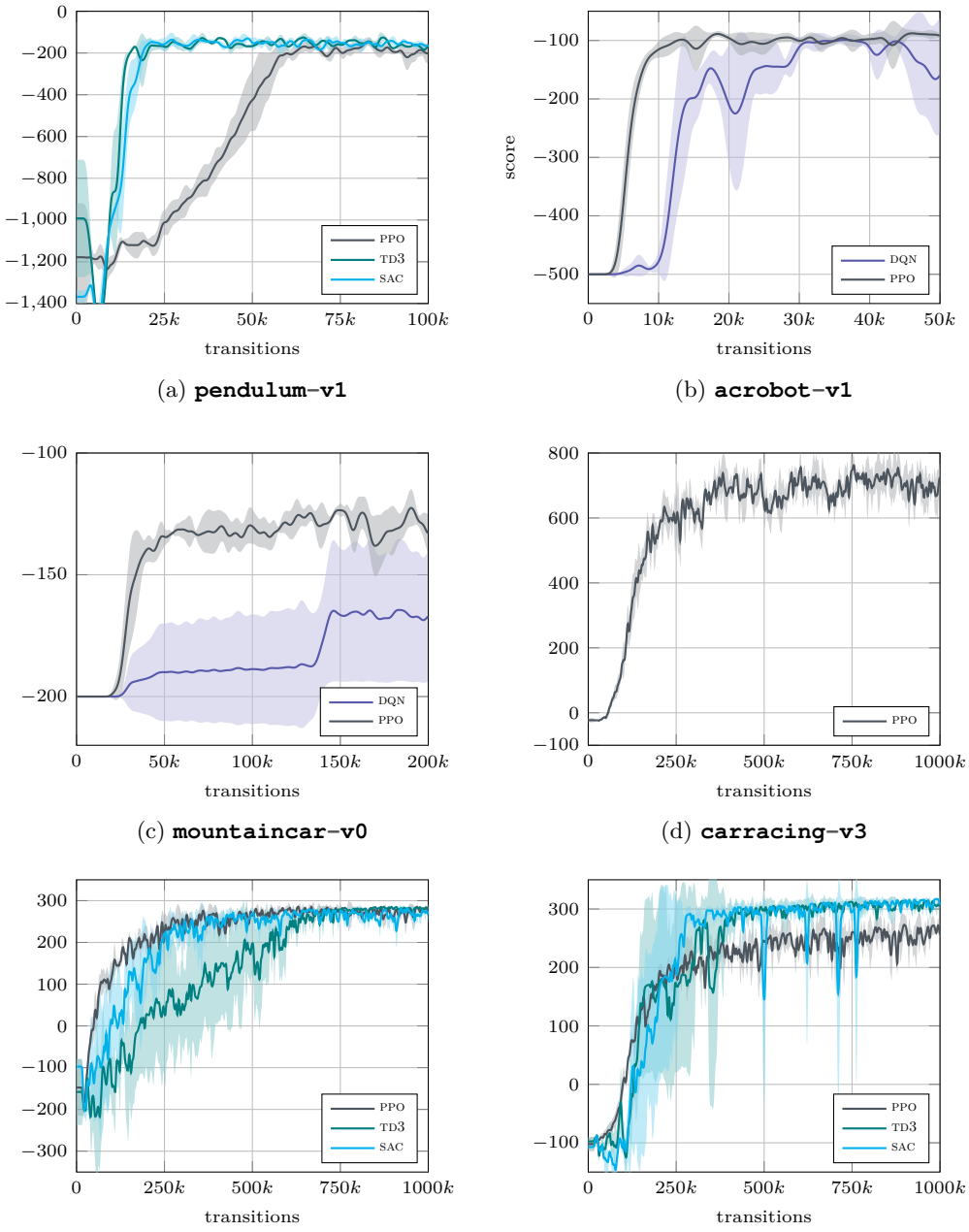
(b) 10 dimensions

(c) 20 dimensions

Figure 5: **Comparing traditional PPO with separable PPO** on the **`shkadov-v0`** environment with 5−, 10− and 20− dimensional action spaces. Note that the reward is not computed in the same manner as in [10].

Doing so, the problem is simplified by (i) reducing the observation space dimensionality and (ii) multiplying by n_{act} the number of available samples to train the agent. This results in a significant learning speedup, especially for large action spaces.

In figure 5, we reproduce the main results from [10] by comparing the score curves with and without the separability feature on the **`shkadov-v0`** environment, with 5-, 10- and 20-dimensional action spaces (for more details, please refer to [9]). As can be observed, the performance of the separable formulation remains steady even for high action dimensionality.



(a) **pendulum-v1** (b) **acrobot-v1**

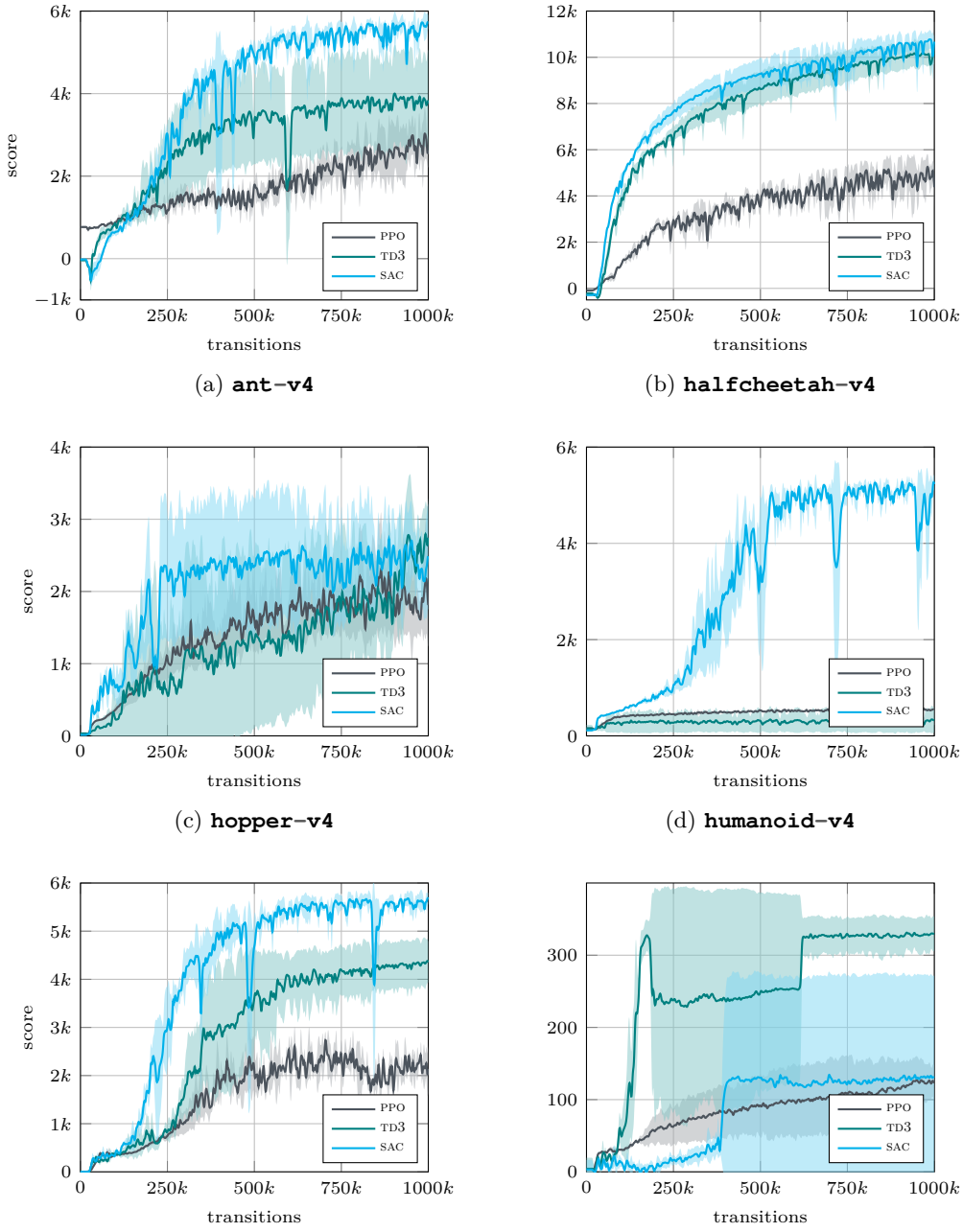
(c) **mountaincar-v0** (d) **carracing-v3**

(e) **lunarlandercontinuous-v3** (f) **bipedalwalker-v3**

Figure 6: **Performance comparison on the GYM benchmark.**

4 Performance and benchmarks

The agents implementations are tested on three different litterature benchmarks: (i) the GYMNASIUM environments [11], (ii) the MUJOCO benchmark [12], and (iii) the BEACON benchmark [9], the latter being a benchmark library dedicated to flow control problems. Results are presented respectively in figures 6, 7 and 8.

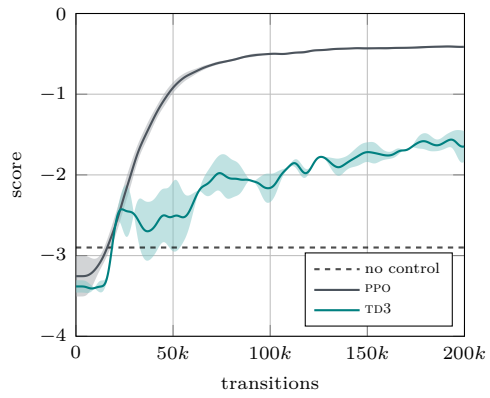


(a) **ant-v4** (b) **halfcheetah-v4**

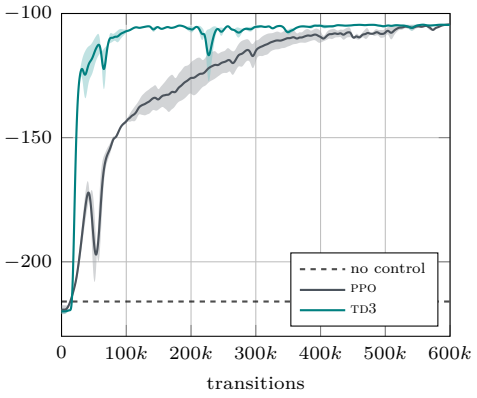
(c) **hopper-v4** (d) **humanoid-v4**

(e) **walker2d-v4** (f) **swimmer-v4**

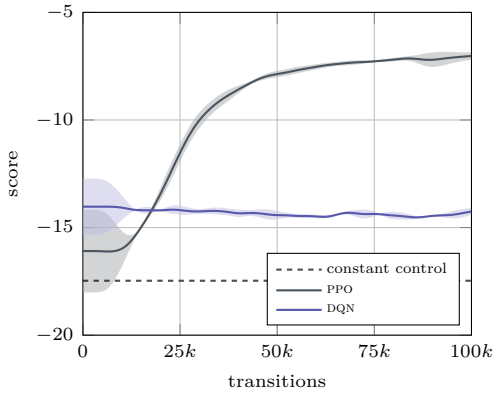
Figure 7: **Performance comparison on the** MUJOCO **benchmark.**



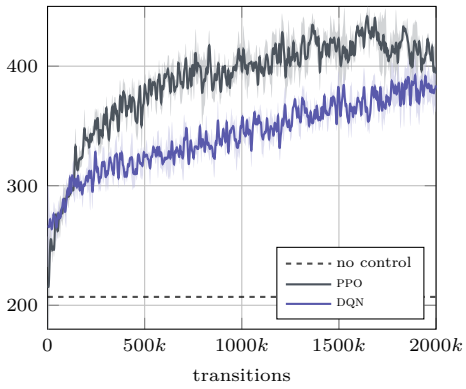
(a) **shkadov-v0**



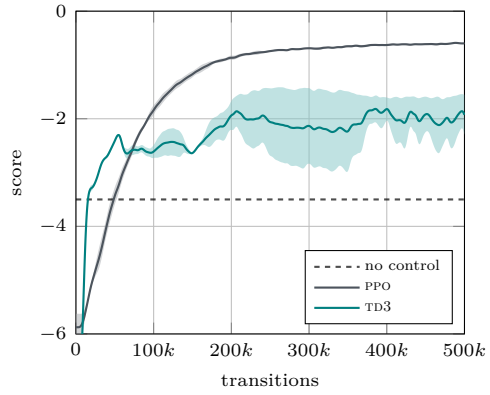
(b) **rayleigh-v0**



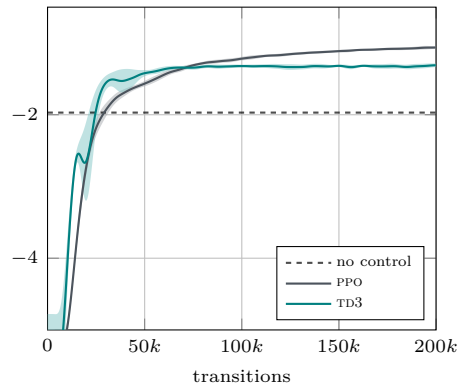
(c) **mixing-v0**



(d) **lorenz-v0**



(e) **burgers-v0**



(f) **sloshing-v0**

Figure 8: **Performance comparison on the** BEACON **benchmark.**

5 Conclusion

The present contribution presented the DRAGONFLY library, its modular construction pattern and several of its interesting features. The performance levels of the different agents was also assessed on well-known literature benchmarks.

A Acknowledgements

Funded/Co-funded by the European Union (ERC, CURE, 101045042). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] R. S. Sutton and A. G. Barto. Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA, 2018.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing Atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602, 2013.

[3] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. Mastering the game of Go without human knowledge. Nature, 550, 2017.

[4] L. Pinto, M. Andrychowicz, P. Welinder, W. Zaremba, and P. Abbeel. Asymmetric actor critic for image-based robot learning. arXiv preprint arXiv:1710.06542, 2017.

[5] A. Kendall, J. Hawke, D. Janz, P. Mazur, D. Reda, J.-M. Allen, V.-D. Lam, A. Bewley, and A. Shah. Learning to drive in a day. arXiv preprint arXiv:1807.00412, 2018.

[6] S. Huang, R. F. J. Dossa, C. Ye, and J. Braga. Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms, 2021.

[7] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann. Stable-baselines3: Reliable reinforcement learning implementations. Journal of Machine Learning Research, 22(268):1–8, 2021.

[8] J. Viquerat and E. Hachem. Parallel bootstrap-based on-policy deep reinforcement learning for continuous flow control applications, 2023.

[9] J. Viquerat, P. Meliga, P. Jeken-Rico, and E. Hachem. Beacon, a lightweight deep reinforcement learning benchmark library for flow control. Applied Sciences, 14(9), 2024.

[10] V. Belus, J. Rabault, J. Viquerat, Z. Che, E. Hachem, and U. Reglade. Exploiting locality and translational invariance to design effective deep reinforcement learning control of the 1-dimensional unstable falling liquid film. AIP Advances, 9:125014, 2019.

[11] M. Towers, A. Kwiatkowski, J. Terry, J. U. Balis, G. De Cola, T. Deleu, M. Goulão, A. Kallinteris, M. Krimmel, A. KG, R. Perez-Vicente, A. Pierré, S. Schulhoff, J. J. Tai, H. Tan, and O. G. Younis. Gymnasium: A standard interface for reinforcement learning environments, 2024.

[12] E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems, pages 5026–5033. IEEE, 2012.