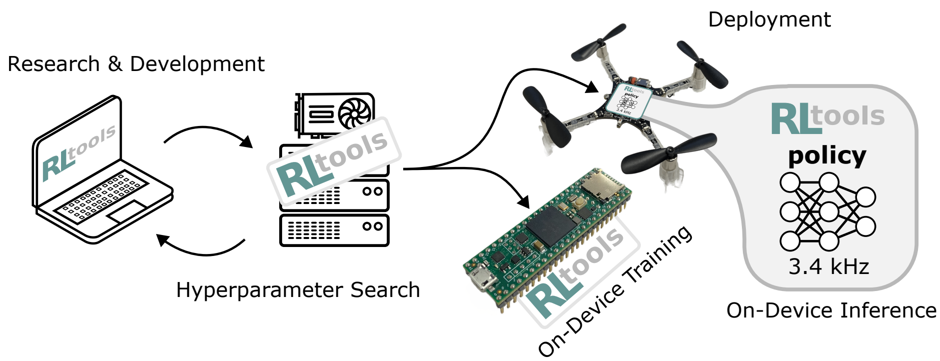
Journal of Machine Learning Research 25 (2024) 1-19 Submitted 2/24; Revised 8/24; Published 9/24

RLtools: A Fast, Portable Deep Reinforcement Learning Library for Continuous Control

Jonas Eschmann[1,2] **Dario Albani**[2] **Giuseppe Loianno**[1]
[1]**New York University** [2]**Technology Innovation Institute**
{jonas.eschmann,loiannog}@nyu.edu
dario.albani@tii.ae

Editor: Alexandre Gramfort



Abstract

Deep Reinforcement Learning (RL) can yield capable agents and control policies in several, domains but is commonly plagued by prohibitively long training times. Additionally, in the case of continuous control problems, the applicability of learned policies on real-world embedded devices is limited due to the lack of real-time guarantees and portability of existing libraries. To address these challenges, we present **RLtools**, a dependency-free, header-only, pure C++ library for deep supervised and reinforcement learning. Its novel architecture allows **RLtools** to be used on a wide variety of platforms, from HPC clusters over workstations and laptops to smartphones, smartwatches, and microcontrollers. Specifically, due to the tight integration of the RL algorithms with simulation environments, **RLtools** can solve popular RL problems up to 76 times faster than other popular RL frameworks. We also benchmark the inference on a diverse set of microcontrollers and show that in most cases our optimized implementation is by far the fastest. Finally, **RLtools** enables the first-ever demonstration of training a deep RL algorithm directly on a microcontroller, giving rise to the field of Tiny Reinforcement Learning (TinyRL). The source code as well as documentation and live demos are available through our project page at https://rl.tools.

Keywords: Reinforcement Learning, Continuous Control, Deep Learning, TinyRL

1. Introduction

Continuous control is a ubiquitous and pervasive problem in a diverse set of domains such as robotics, high-frequency decision-making in financial markets or the automation of chemical plants and smart grid infrastructure. Taking advantage of the recent progress in Deep Learning (DL) that is spilling over into decision-making in the form of RL, agents derived using deep RL have already attained impressive performance in a range of decision-making problems, like games and particularly continuous control. Despite these achievements, the real-world adoption of RL for continuous control is hindered by prohibitively long training times as well as a lack of support for the deployment of trained policies on real-world embedded devices. Long training times obstruct rapid iteration in the problem space (reward function design, hyperparameter tuning, etc.) while deployment on computationally severely limited embedded devices is necessary to control the bulk of physical systems such as: robots, automotive components, medical devices, smart grid infrastructure, etc. In non-physical systems, such as financial markets, the need for high-frequency decision-making leads to similar real-time requirements which cannot be fulfilled by current deep RL libraries. Hence, to address these challenges we present **RLtools**, a dependency-free, header-only pure C++ library for deep supervised and reinforcement learning combining the following contributions:

- **Novel Architecture**: We describe the innovations in the software design of the library which allow for unprecedented training and inference speeds on a wide variety of devices from High-Performance Computing (HPC) clusters over workstations and laptops to smartphones, smartwatches and microcontrollers.
- **Implementation**: We contribute a modular, highly portable, and efficient implementation of the aforementioned architecture in the form of open-source code, documentation, and test cases.
- **Fastest Training**: We demonstrate large speedups in terms of wall-clock training time.
- **Fastest Inference**: We demonstrate large speedups in terms of the inference time of trained policies on a diverse set of common microcontrollers.
- **TinyRL**: By utilizing **RLtools**, we successfully demonstrate, the first-ever training of a deep RL algorithm for continuous control directly on a microcontroller.

2. Related Work

Multiple deep RL frameworks and libraries have been proposed, many of which cover algorithmic research, with and without abstractions (Acme (Hoffman et al., 2020), skrl (Serrano-Munoz et al., 2023) and CleanRL (Huang et al., 2022) respectively). Other frameworks and libraries focus on comprehensiveness in terms of the number of algorithms included (RLlib (Liang et al., 2018), ReinforcementLearning.jl (Tian, 2020), MushroomRL (D'Eramo et al., 2021), Stable-Baselines3 (Raffin et al., 2021), ChainerRL (Fujita et al., 2021)), Tianshou (Weng et al., 2022), and TorchRL (Bou et al., 2024). In contrast to these aforementioned solutions, **RLtools** aims at fast iteration in the problem space in the form of e.g., reward function design (Eschmann, 2021) and hyperparameter optimization. In the problem space, the algorithmic intricacies and variety of the algorithms matter less than the robustness, training speed, and final performance as well as our understanding of how to train them reliably. From the formerly mentioned RL frameworks and libraries RLlib (Liang et al., 2018) is the most similar in terms of its mission statement being on quick iteration and deployment (cf. benchmark comparisons wrt. this goal in Section 4). By focusing on iteration in the space of problems and subsequent deployment to real-time platforms, we also draw parallels between **RLtools** and the ACADOS software (Verschueren et al., 2022) for synthesizing Model Predictive Controls (MPCs) with **RLtools** aspiring to be its RL equivalent.

3. Approach

Taking the last handful of years of progress in RL for continuous control, it can be observed that the most prominent models used as function approximators are still relatively small, fully-connected neural networks. In Appendix A we analyze the architectures used in deep RL for continuous control and justify the focus of **RLtools** on (small) fully-connected neural networks. Based on these observations, we conclude that the great flexibility provided by automatic differentiation frameworks like TensorFlow or PyTorch might not be necessary for applying RL to many continuous control problems. We believe that there is an advantage in trading-off the flexibility in the model architecture of the function approximators for the overall training speed. Reducing the training time and increasing the training efficiency saves energy, simplifies reproducibility and democratizes access to state of the art RL methods. Furthermore, fast training facilitates principled hyperparameter search which in turn improves comparability.

Architecture Our software architecture is guided by the previous observation and hence by maximizing the training time efficiency without sacrificing returns. Additionally, we want the software to be able to run across many different accelerators and devices (CPUs, GPUs, microcontrollers, and other accelerators) so that trained policies can also directly be deployed on microcontrollers and take advantage of device-specific instructions to run at high frequencies with hard realtime guarantees. This also entails that **RLtools** does not rely on any dependencies because they might not be available on the target microcontrollers.

To attain maximum performance, we integrate the different components of our library as tightly as needed while maintaining as much flexibility and modularity as possible. To enable this goal, we heavily rely on the C++ templating system. Leveraging template meta-programming, we can provide the compiler with a maximum amount of information about the structure of the code, enabling it to be optimized heavily. We particularly make sure that the size of all loops is known at compile time such that the compiler can optimize them via inlining and loop-unrolling (cf. Appendix B and F). Leveraging pure C++ without any dependencies, we implement the following major components: **Deep Learning** (MLP, backpropagation, Adam, etc.), **Reinforcement Learning** (GAE, PPO, TD3, SAC), and **Simulation** (Pendulum, Acrobot, Quadrotor, Racing Car, MuJoCo interface). We implement **RLtools** in a modular way by using a novel static multiple-dispatch paradigm inspired by (dynamic) multiple-dispatch which was popularized by the Julia programming language Bezanson et al. (2012). We highly recommend taking a look at the code example and explanation in Appendix B as well as the ablation study in Appendix F measuring the impact of different components and optimizations.

4. Results

Horizontal Benchmark Figure 1 and 2 show the resulting mean training times from running the PPO and SAC algorithm across ten runs on an Intel-based laptop (details in Table 6). We find that RLtools outperforms existing libraries by a wide margin. Particularly in the case of PPO where RLtools only takes 0.54 s on average (2.59 s in case of SAC).

Vertical Benchmark In Figure 3, we also present training results using **RLtools** on a wide variety of devices which are generally not compatible with the other RL libraries and

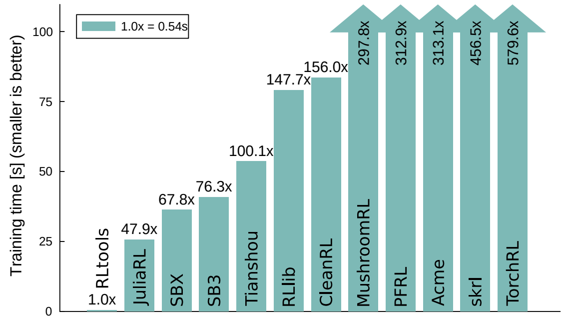


Figure 1: PPO: `Pendulum-v1` (300000 steps)

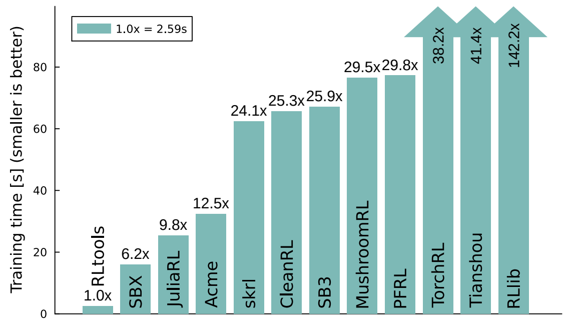


Figure 2: SAC: `Pendulum-v1` (10000 steps)

Platform	**RLtools**: Generic	DSP Library	**RLtools**: Optimized
Crazyflie	743 us (1.3 kHz)	478 us (2.1 kHz)	**293 us (3.4 kHz)**
Pixhawk 6C	133 us (7.5 kHz)	93 us (10.8 kHz)	**53 us (18.8 kHz)**
Teensy 4.1	64 us (15.5 kHz)	45 us (22.3 kHz)	**41 us (24.3 kHz)**
ESP32 (Xtensa)	4282 us (234 Hz)	**279 us (3.6 kHz)**	333 us (3 kHz)
ESP32-C3 (RISC-V)	8716 us (115 Hz)	6950 us (144 Hz)	**6645 us (150 Hz)**

Table 1: Inference times on different platforms

frameworks. Importantly, we also demonstrate the first training of a deep RL agent for continuous control on a microcontroller in form of the Teensy 4.1.

Inference on Microcontrollers Table 1 shows the inference times on microcontrollers of different compute capabilities (e.g. Crazyflie is a 27 g quadrotor with very limited resources, cf. Appendix E). The generic implementation already yields usable inference times but dispatching to the manufacturers Digital Signal Processor (DSP) library improves the performance. Finally, by optimizing the code further (e.g. through fusing the activation operators) we achieve a significant speedup even compared to the manufacturers DSP libraries.

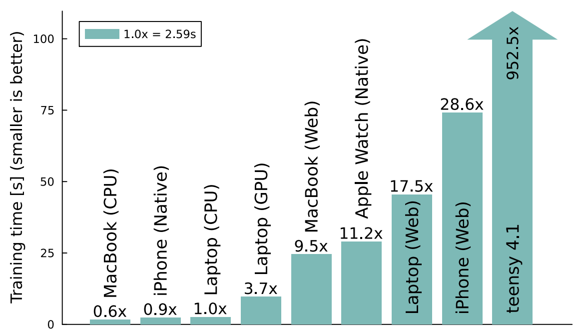


Figure 3: SAC: `Pendulum-v1` (10000 steps)

5. Conclusion

We believe **RLtools** fills a gap by allowing fast iteration in the problem space and subsequent real-time deployment of policies. Furthermore, **RLtools** facilitates the first-ever deep RL training on a microcontroller. We acknowledge the steeper learning curve of C++ (over e.g. Python) but from our experience, the faster iteration made possible by shorter training times can outweigh the added time to get started. Currently **RLtools** is limited to dense observations but we plan to add vision capabilities in the future. We believe that by relaxing the compute requirements and, by being fully open-source, **RLtools** democratizes the training of state-of-the-art RL methods and accelerates progress in RL for continuous control.

Acknowledgments

This work was supported by the Technology Innovation Institute, the NSF CAREER Award 2145277, and the DARPA YFA Grant D22AP00156-00. Giuseppe Loianno serves as a consultant for the Technology Innovation Institute. This arrangement has been reviewed and approved by New York University in accordance with its policy on objectivity in research.

Appendix A. Analysis of the Deep RL Landscape

Year	Name	Hidden Dim	#Params	Non-Linearity
2015	TRPO Schulman et al. (2015)	[50, 50]	1.0x	tanh
2015	GAE Schulman et al. (2016)	[100, 50, 25]	2.3x	tanh
2016	DDPG Lillicrap et al. (2016)	[400, 300]	35.3x	ReLU
2017	PPO Schulman et al. (2017)	[64, 64]	1.5x	tanh
2018	TD3 Fujimoto et al. (2018)	[400, 300]	35.3x	ReLU
2018	SAC Haarnoja et al. (2018)	[256, 256]	19.6x	ReLU
2019	SACv2 Haarnoja et al. (2019)	[256, 256]	19.6x	ReLU
2020	TQC Kuznetsov et al. (2020)	[512, 512, 512]	147.0x	ReLU
2020	D4PG&TD3Bach et al. (2020)	[256, 256]	19.6x	ReLU
2021	PPO&RMA Kumar et al. (2021)	[128, 128, 128]	9.8x	ReLU

Table 2: Selection of works that introduced impactful algorithms and the respective neural network dimensions used for their value function approximations. For the calculation of the number of parameters, an input size of 20 and an output size of 1 is assumed

In this section, we analyze the function approximator models used in the major deep RL for continuous control publications collected in Table 7. The most important observation is that over all the years the architecture (small, fully-connected neural networks) has not changed. This can be attributed to the fact that in continuous control the observations are usually dense states of the systems which do not contain any spatial or temporal regularities like images or time series that would suggest the usage of less general, more tailored network structures like Convolutional Neural Networks (CNNs) or Recurrent Neural Networks (RNNs). This regularity, as stated in section 3, motivates our focus on optimizing and tightly integrating fully-connected neural networks as a first step. We also plan integrate recurrent and possibly convolutional layers in the future.

Appendix B. Programming Paradigm

To enable maximum performance, we are avoiding C++ Virtual Method Table (VMT) lookups by not using an object-oriented paradigm but a rather functional paradigm heavily based on templating and method overloading resembling a static, compile-time defined

```
// file: implementation_generic.h
template <typename DEVICE, auto M, auto N, auto K>
void multiply(DEVICE device, Matrix<M, K> a, Matrix<K, N> b, Matrix<M, N> result){
    // Generic code for matrix multiplication
    ...
}
// file: implementation_microcontroller.h
template <auto M, auto N, auto K>
void multiply(MICROCONTROLLER device, Matrix<M, K> a, Matrix<K, N> b, Matrix<M, N>
    result){
    // Optimized code for matrix multiplication on a particular microcontroller (e.g
    . based on DSP extensions)
    ...
}
// file: implementation_gpu.h
template <auto M, auto N, auto K>
void multiply(GPU device, Matrix<M, K> a, Matrix<K, N> b, Matrix<M, N> result){
    // Optimized GPU code for matrix multiplication (e.g. CUDA kernel launch)
    ...
}
template<typename DEVICE, typename OBJECT_A, typename OBJECT_B, typename OBJECT_C>
void algorithm(DEVICE device, OBJECT_A a, OBJECT_B b, OBJECT_C c){
    ...
    multiply(device, a, b, c);
    ...
}

// usage
GPU device;
Matrix<10, 10> a, b, result;
... // malloc and initialize randomly on the GPU device using tag dispatch as well
algorithm(device, a, b, result);
```

Figure 4: Toy example for tag dispatch towards different implementations of elementary matrix operations

interpretation of the multiple dispatch paradigm. Multiple dispatch has been popularized by the Julia programming language Bezanson et al. (2012) and is based on advanced function overloading.

Leveraging multiple dispatch, higher-level functions like the forward or backward pass of a fully-connected neural network just specify the actions that should be taken on the different sub-components/layers and the actual implementation used is dependent on the type of the arguments. In this way, it is simple to share code between GPU and CPU implementations by just implementing the lower-level primitives for the respective device types and then signaling the implementations through the argument type (i.e. using the tag dispatch technique). A toy example for this is displayed in Figure 4. In this case, some `algorithm` is using a matrix multiplication operation on two objects. During the implementation of the `algorithm`, we do not need to care about the type of the operands and just let them be specified by wildcard template parameters. When this function is called by the user, the compiler infers the template parameters and dispatches the call to the appropriate implementation. If the user does not have a GPU available he simply does not include the `implementation_gpu.h` and hence has no dependency on further dependencies that the GPU implementation would entail (e.g., the CUDA toolkit). In the case where there is no specialized implementation for a particular hardware, the compiler will fall back to the generic implementation which in this example could simply consist of a nested loop. The generic implementations are pure C++ and are guaranteed to have no dependencies. We can also see that the compiler will check the dimensions of the operands automatically at compile time such that the `algorithm` can not be called with incompatible shapes. To create more complex dispatch behaviors and operand type checking C++ features like `static_assert` and `enable_if` can be leveraged through the Substitution Failure Is Not An Error (SFINAE) mechanism. In this way, we can maintain composability while still providing all the structure to the compiler at compile-time.

In the case of Julia, this leads to unparalleled composability which manifests in a small number of people being able to implement and maintain e.g. a deep learning library (Flux Innes (2018)) that is competitive with PyTorch and TensorFlow which are backed by much more resources. In contrast to Julia, which reaches almost native performance while performing the multiple dispatch resolution at runtime, we make sure that all the function calls can be resolved at compile time. Additionally, Julia is not suited for our purposes because it does not fit to run on microcontrollers due to its runtime size and stochastic, non-realtime behavior due to the garbage collection-based memory management. Nevertheless, in our benchmark presented later in this manuscript, we found that Julia is one of the closest competitors when it comes to training performance. Furthermore, we find it important to emphasize that we focus on building a library not a framework.[1] The main feature of frameworks is that they restrict the freedoms of the user to make a small set of tasks easier to accomplish. In certain, repetitive problem settings this might be justified, but in many cases, the overhead coming with the steep learning curves and finding workarounds after bumping into the tight restrictions of frameworks is not worth it. The major conceptual difference is that frameworks provide a context from which they invoke the user's code while in the case of libraries, the user is entirely in control and invokes the components he needs. If not specifically made interoperable, the contexts provided by frameworks are usually incompatible while with libraries this is not generally the case.

In our implementation, this for example concretely manifests in the way function approximators are used in the RL algorithms. By using templating, any function approximator type can be specified by the user at compile time. As long as he also provides the required `forward` and `backward` functions.

As demonstrated in Figure 4 we establish the convention of making a device-dependent context available in each function via tag dispatch to simplify the usage of different compute devices like accelerators or microcontrollers.

1. **Write Libraries, Not Frameworks** [link]

Appendix C. Benchmark Details

Parameter	Value
Actor structure	$[64, 64]$
Critic structure	$[64, 64]$
Activation function	Rectified Linear Unit (ReLU)
Batch size	256
Number of environments	4
Steps per environment	1024
Number of epochs	2
Total number of (environment) steps	300000
Discount factor γ	0.9
Generalized Advantage Estimation (GAE) λ	0.95
ϵ clip	0.2
Entropy coefficient β	0
Advantage normalization	`true`
Adam α	1×10^{-3}
Adam β_1	0.9
Adam β_2	0.999
Adam ϵ	1×10^{-7}

Table 3: `Pendulum-v1` PPO parameters (Figure 1)

Parameter	Value
Actor structure	$[64, 64]$
Critic structure	$[64, 64]$
Activation function	ReLU
Batch size	100
Total number of (environment) steps	10000
Replay buffer size	10000
Discount factor γ	0.99
Entropy bonus coefficient (learned, initial value) α	0.5
Polyak β	0.99
Adam α	1×10^{-3}
Adam β_1	0.9
Adam β_2	0.999
Adam ϵ	1×10^{-7}

Table 4: `Pendulum-v1` SAC parameters (Figure 2)

Parameter	Value
Input dimensionality	13
Policy structure	$[64, 64]$
Output dimensionality	4
Activation function	ReLU

Table 5: On-device inference parameters (Table 1)

Label	Details
RLtools / Laptop (CPU) / Laptop (Web) / Baseline	Intel i9-10885H
Laptop (GPU)	Intel i9-10885H + Nvidia T2000
MacBook (CPU) / MacBook (Web)	MacBook Pro (M3 Pro)
iPhone (Native) / iPhone (Web)	iPhone 14
Apple Watch (Native)	Apple Watch Series 4

Table 6: `Pendulum-v1` SAC devices (Figure 3)

Appendix D. Deep Reinforcement Learning Frameworks and Libraries

Name ↓	Platform	Stars / Citations
Acme (Hoffman et al., 2020)	JAX	3316 / 219
CleanRL (Huang et al., 2022)	PyTorch	4030 / 86
MushroomRL (D'Eramo et al., 2021)	TF/PyTorch	749 / 61
PFRL (Fujita et al., 2021)	PyTorch	1125 / 122
ReinforcementLearning.jl (JuliaRL) (Tian, 2020)	Flux.jl (Julia)	543 / n/a
RLlib + `ray` (Liang et al., 2018)	PyTorch	29798 / 828
Stable Baselines3 (SB3) (Raffin et al., 2021)	PyTorch	7396 / 1149
Stable Baselines JAX (SBX) (Raffin et al., 2021)	JAX	223 / n/a
Tianshou (Weng et al., 2022)	PyTorch	7139 / 133
TorchRL (Weng et al., 2022)	PyTorch	1691 / 5

Table 7: Overview over different RL libraries/frameworks, the deep learning platform they build upon, and their popularity in terms of Github stars and publication citations (data as of 2024-02-07)

Appendix E. Embedded Platforms

1. **Crazyflie**: A small, open-source quadrotor which only weighs 27 g including the battery. The Crazyflie's main processor is a STM32F405 microcontroller using the ARM Cortex-M4 architecture, featuring 192 KB of Random Access Memory (RAM) and running at 168 MHz.

2. **Pixhawk 6C**: We use a Pixracer Pro, a Flight Controller Unit (FCU) that belongs to the family of Pixhawk FCUs and implements the Pixhawk 6C standard. Hence, the PixRacer Pro supports the common PX4 firmware Meier et al. (2015) and can be used in many different vehicle types (aerial, ground, marine) but is predominantly used in multirotor vehicles of varying sizes. The main processor used in the Pixhawk 6C standard is a STM32H743 using the ARM Cortex-M7 architecture. The PixRacer Pro runs at 460 MHz and comes with 1024 KB of RAM.

3. **Teensy 4.1**: A general-purpose embedded device powered by an i.MX RT1060 ARM Cortex-M7 microcontroller with 1024 KB on-chip and 16 MB off-chip RAM that is running at 600 MHz.

4. **ESP32**: One of the most common microcontrollers for Internet of Things (IoT) and edge devices due to its built-in Wi-Fi and Bluetooth. Close to 1 billion devices built around this chip and its predecessor have been sold worldwide. Hence it is widely available and relatively cheap (around $5 for a development kit). For our purposes,

the ESP32 is interesting because it deviates from the previous platforms in that its processor is based on the Xtensa LX7 architecture. In addition to the original version of the ESP32 based on the Xtensa architecture, we also evaluate the ESP32-C3 version based on the RISC-V architecture.

Appendix F. Ablation Study

We conduct an ablation study to investigate the contribution of different components and optimizations to the fast wall-clock training time achieved by **RLtools**. Figure 5 shows the resulting training times after removing different components and optimizations from the setup. The “Baseline” is exactly the same setup used in the `Pendulum-v1` (SAC) training in the other experiments in Section 4. We simulate the slowness of the Python environment by slowing down the C++ implementation by the average time required for a step in the Python implementation. We can observe that the C++ implementation of the `Pendulum-v1` dynamics has a measurable, but not dominating impact on the training time. Additionally, we ablate the different optimization levels `-O0`, `-O1`, `-O2` and `-O3` (used in the Baseline) of the C++ compiler. We can observe that the compiler optimizations have a sizable impact on the training time. When removing all optimizations (`-O0`) **RLtools** is roughly between ACME and CleanRL (cf. Figure 2). Furthermore, the “No Fast Math” configuration tests removing the `-ffast-math` from the compiler options, and the “BLAS” Basic Linear Algebra Subprograms (BLAS) option removes the Intel oneMKL matrix multiplication kernels. In the case of “AVX/AVX2” we disable the Advanced Vector Extensions (AVX) that are used for Single Instruction, Multiple Data (SIMD) operations. We notice that due to the design of **RLtools** (refer to Appendix B) which allows the sizes of all loops and data structures to be known at compile-time the compiler is able to better reason about the code and hence make heavy use of vectorized/SIMD operations. We observe that $2276 + 1430 = 3706$ (AVX + Streaming SIMD Extensions (SSE), an older set of vectorized instructions) out of 11243 machine-code instructions in total refer to registers of the vector extensions. Unfortunately (for the sake of measurement), when turning off AVX, the compiler replaces the instructions with SSE instructions (5406 out of 11243 in this case) which we could not turn off because of some dependency in `libstdc++`. Still, the number of SSE instructions demonstrates the compiler-friendliness that **RLtools**’ architecture entails.

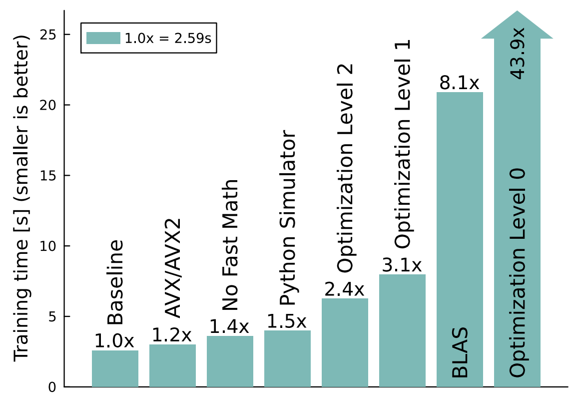


Figure 5: Ablation study. The “Baseline” contains all optimizations.

Appendix G. Convergence Study

To make sure the implementations of the supported RL algorithms (PPO, TD3, and SAC) are correct, we conduct a convergence study where we compare the learning curves across different environments with learning curves of other implementations. We make sure that per environment the same hyperparameters are used across all implementations and run each setup for multiple seeds (100 for `Pendulum-v1` and 30 for `Hopper-v1`). For each of the seeds at every evaluation step, we perform 100 episodes with random initial states.

By comparing different sets of 100 seeds each, we found that, even for a large number of seeds, outliers have a significant impact on the mean final return. Hence, as also recommended by Agarwal et al. (2021), we report the Inter Quantile Mean (IQM) which discards the lowest and highest quantile to remove the impact of outliers on the statistics. We still aim at capturing as much of the final return distribution by only discarding the lower upper 5 % for the calculation of the IQM μ. We use the same inter-quantile set for the calculation of the standard deviation σ. To make sure that the environments are identical in this convergence study, instead of re-implementing the environments in C++ using the **RLtools** interface, we built a Python wrapper for **RLtools** such that we can use the original environments from the Gymnasium (Towers et al.) suite. The Python wrapper makes **RLtools** easier to use but sacrifices in terms of performance if the environment/simulator is implemented in Python (as shown in Appendix F).

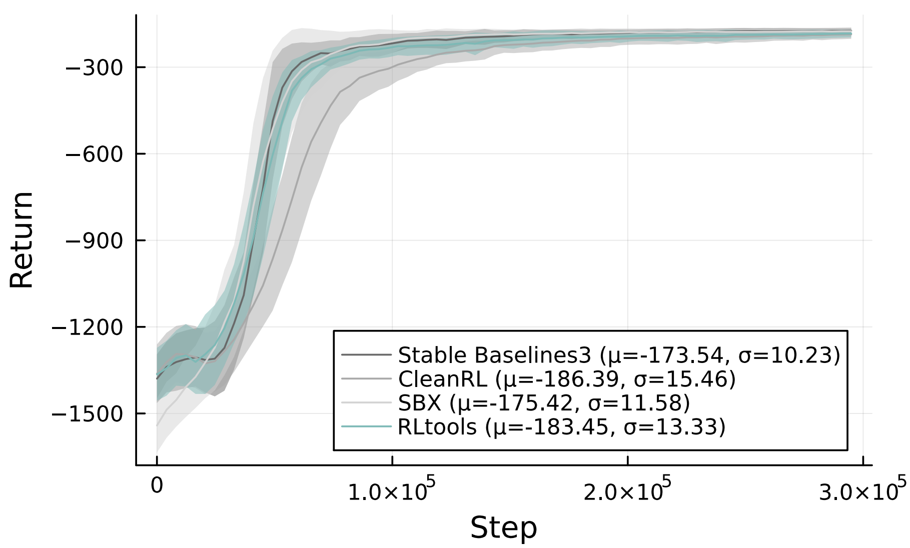


Figure 6: PPO `Pendulum-v1`

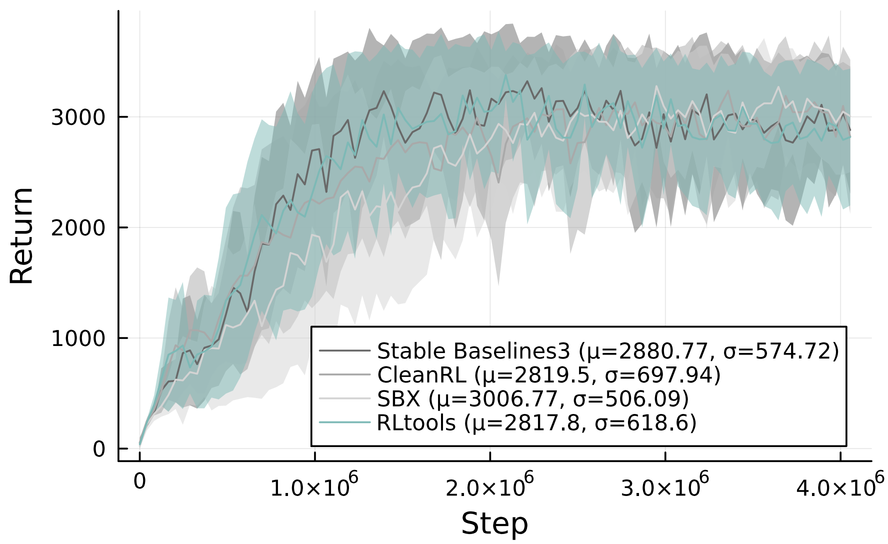


Figure 7: PPO `Hopper-v4`

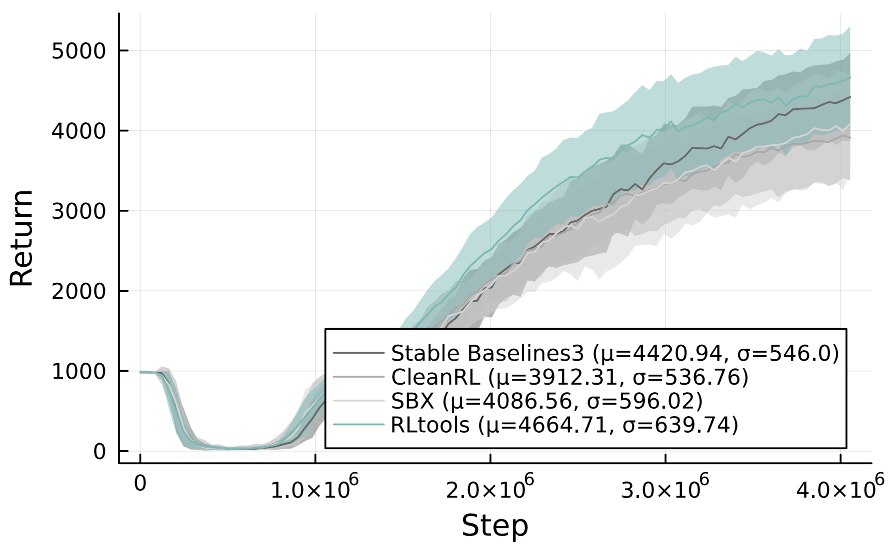


Figure 8: PPO `Ant-v4`

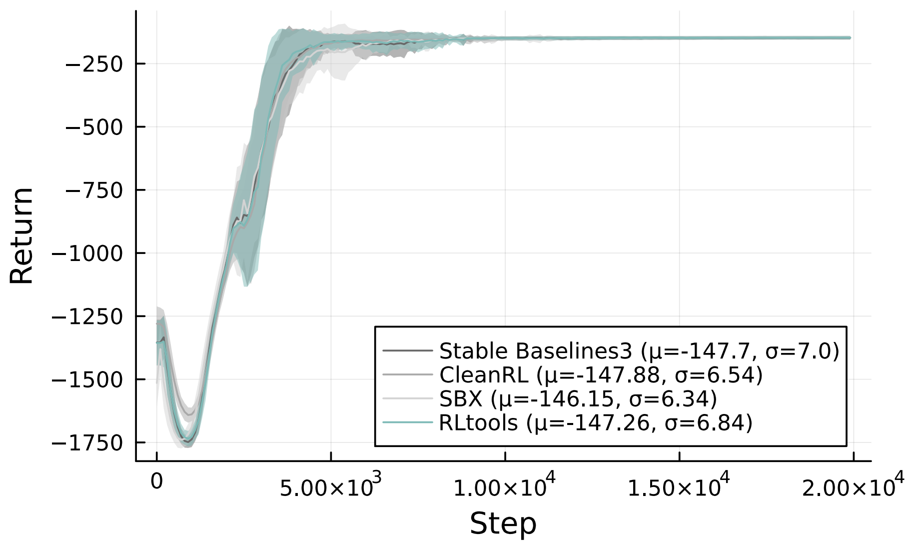


Figure 9: SAC `Pendulum-v1`

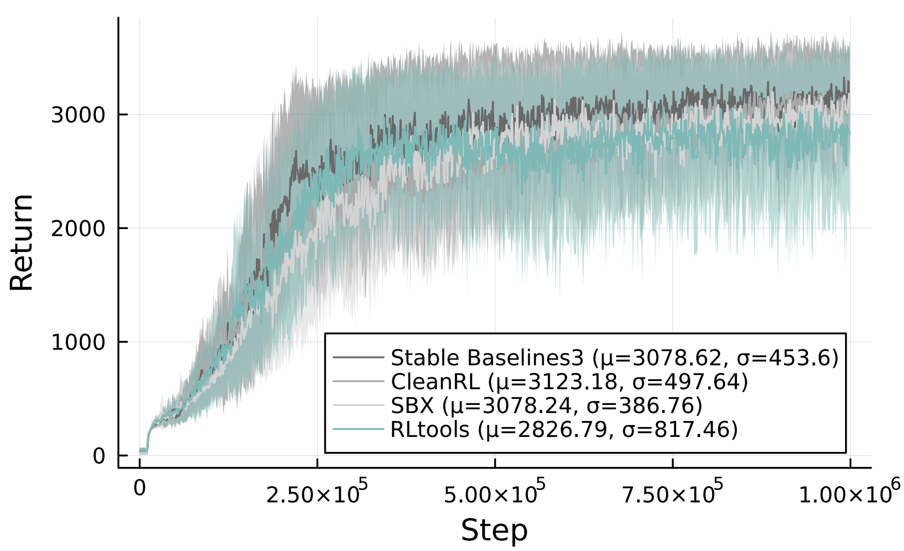


Figure 10: SAC `Hopper-v4`

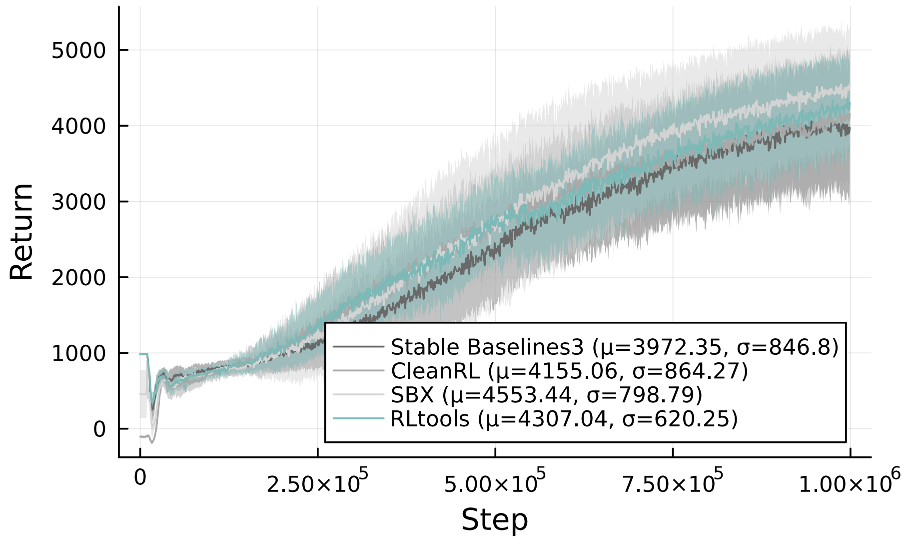


Figure 11: SAC `Ant-v4`

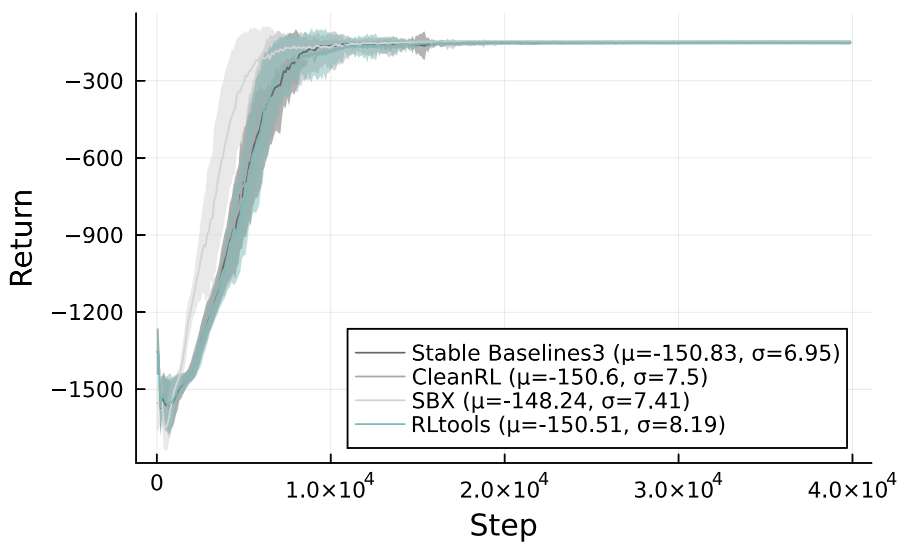


Figure 12: TD3 `Pendulum-v1`

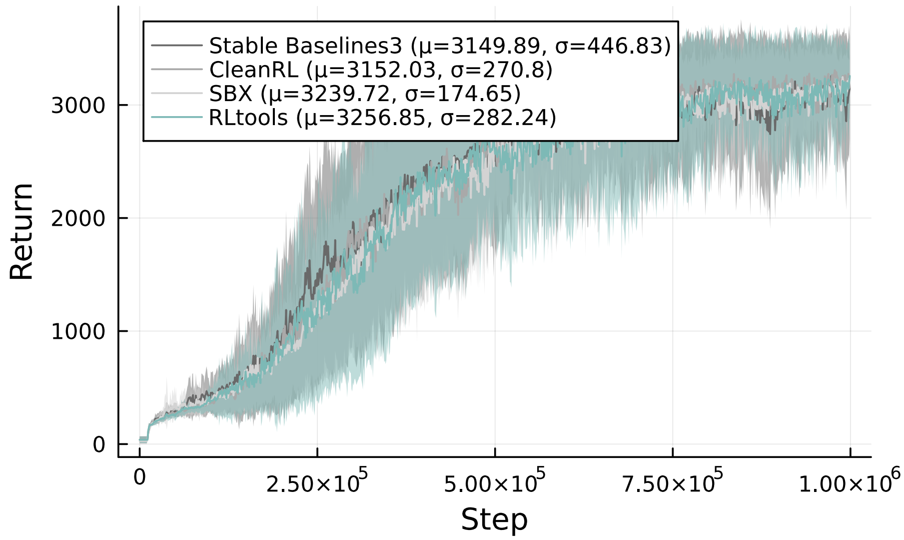


Figure 13: TD3 `Hopper-v4`

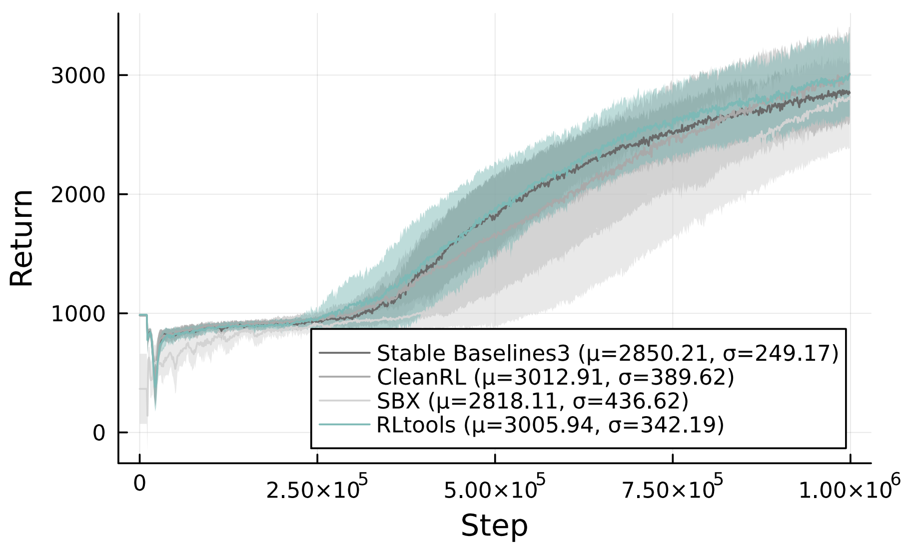


Figure 14: TD3 `Ant-v4`

References

Rishabh Agarwal, Max Schwarzer, Pablo Samuel Castro, Aaron C Courville, and Marc Bellemare. Deep reinforcement learning at the edge of the statistical precipice. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 29304–29320. Curran Associates, Inc., 2021. URL https://proceedings.neurips.cc/paper_files/paper/2021/file/f514cec81cb148559cf475e7426eed5e-Paper.pdf.

Nicolas Bach, Andrew Melnik, Malte Schilling, Timo Korthals, and Helge Ritter. Learn to Move Through a Combination of Policy Gradient Algorithms: DDPG, D4PG, and TD3. In Giuseppe Nicosia, Varun Ojha, Emanuele La Malfa, Giorgio Jansen, Vincenzo Sciacca, Panos Pardalos, Giovanni Giuffrida, and Renato Umeton, editors, *Machine Learning, Optimization, and Data Science*, volume 12566, pages 631–644. Springer International Publishing, Cham, 2020. ISBN 978-3-030-64579-3 978-3-030-64580-9. doi: 10.1007/978-3-030-64580-9_52. URL http://link.springer.com/10.1007/978-3-030-64580-9_52. Series Title: Lecture Notes in Computer Science.

Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Julia: A Fast Dynamic Language for Technical Computing, September 2012. URL http://arxiv.org/abs/1209.5145. arXiv:1209.5145 [cs].

Albert Bou, Matteo Bettini, Sebastian Dittert, Vikash Kumar, Shagun Sodhani, Xiaomeng Yang, Gianni De Fabritiis, and Vincent Moens. TorchRL: A data-driven decision-making library for pytorch. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=QxItoEAVMb.

Carlo D'Eramo, Davide Tateo, Andrea Bonarini, Marcello Restelli, and Jan Peters. Mushroomrl: Simplifying reinforcement learning research. *Journal of Machine Learning Research*, 22(131):1–5, 2021. URL http://jmlr.org/papers/v22/18-056.html.

Jonas Eschmann. Reward function design in reinforcement learning. *Reinforcement Learning Algorithms: Analysis and Applications*, pages 25–33, 2021.

Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1587–1596. PMLR, 10–15 Jul 2018. URL https://proceedings.mlr.press/v80/fujimoto18a.html.

Yasuhiro Fujita, Prabhat Nagarajan, Toshiki Kataoka, and Takahiro Ishikawa. Chainerrl: A deep reinforcement learning library. *Journal of Machine Learning Research*, 22(77):1–14, 2021. URL http://jmlr.org/papers/v22/20-376.html.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1861–1870. PMLR, 10–15 Jul 2018. URL https://proceedings.mlr.press/v80/haarnoja18b.html.

Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, and Sergey Levine. Soft Actor-Critic Algorithms and Applications, January 2019. URL `http://arxiv.org/abs/1812.05905`. arXiv:1812.05905 [cs, stat].

Matthew W. Hoffman, Bobak Shahriari, John Aslanides, Gabriel Barth-Maron, Nikola Momchev, Danila Sinopalnikov, Piotr Stańczyk, Sabela Ramos, Anton Raichuk, Damien Vincent, Léonard Hussenot, Robert Dadashi, Gabriel Dulac-Arnold, Manu Orsini, Alexis Jacq, Johan Ferret, Nino Vieillard, Seyed Kamyar Seyed Ghasemipour, Sertan Girgin, Olivier Pietquin, Feryal Behbahani, Tamara Norman, Abbas Abdolmaleki, Albin Cassirer, Fan Yang, Kate Baumli, Sarah Henderson, Abe Friesen, Ruba Haroun, Alex Novikov, Sergio Gómez Colmenarejo, Serkan Cabi, Caglar Gulcehre, Tom Le Paine, Srivatsan Srinivasan, Andrew Cowie, Ziyu Wang, Bilal Piot, and Nando de Freitas. Acme: A research framework for distributed reinforcement learning. *arXiv preprint arXiv:2006.00979*, 2020. URL `https://arxiv.org/abs/2006.00979`.

Shengyi Huang, Rousslan Fernand Julien Dossa, Chang Ye, Jeff Braga, Dipam Chakraborty, Kinal Mehta, and Joao G.M. Araujo. Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *Journal of Machine Learning Research*, 23(274): 1–18, 2022. URL `http://jmlr.org/papers/v23/21-1342.html`.

Mike Innes. Flux: Elegant machine learning with Julia. *Journal of Open Source Software*, 3 (25):602, May 2018. ISSN 2475-9066. doi: 10.21105/joss.00602. URL `http://joss.theoj.org/papers/10.21105/joss.00602`.

Ashish Kumar, Zipeng Fu, Deepak Pathak, and Jitendra Malik. RMA: Rapid Motor Adaptation for Legged Robots. In *Proceedings of Robotics: Science and Systems*, Virtual, July 2021. doi: 10.15607/RSS.2021.XVII.011.

Arsenii Kuznetsov, Pavel Shvechikov, Alexander Grishin, and Dmitry Vetrov. Controlling overestimation bias with truncated mixture of continuous distributional quantile critics. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 5556–5566. PMLR, 13–18 Jul 2020. URL `https://proceedings.mlr.press/v119/kuznetsov20a.html`.

Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. RLlib: Abstractions for distributed reinforcement learning. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 3053–3062. PMLR, 10–15 Jul 2018. URL `https://proceedings.mlr.press/v80/liang18b.html`.

Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2016. URL `http://arxiv.org/abs/1509.02971`. arXiv:1509.02971 [cs, stat].

Lorenz Meier, Dominik Honegger, and Marc Pollefeys. Px4: A node-based multithreaded open source robotics framework for deeply embedded platforms. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6235–6240, 2015. doi: 10.1109/ICRA.2015.7140074.

Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021. URL `http://jmlr.org/papers/v22/20-1364.html`.

John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 1889–1897, Lille, France, 07–09 Jul 2015. PMLR. URL `https://proceedings.mlr.press/v37/schulman15.html`.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms, August 2017. URL `http://arxiv.org/abs/1707.06347`. arXiv:1707.06347 [cs].

Antonio Serrano-Munoz, Dimitrios Chrysostomou, Simon Bøgh, and Nestor Arana-Arexolaleiba. skrl: Modular and flexible library for reinforcement learning. *Journal of Machine Learning Research*, 24(254):1–9, 2023.

Jun Tian. Reinforcementlearning.jl: A reinforcement learning package for the julia programming language, 2020. URL `https://github.com/JuliaReinforcementLearning/ReinforcementLearning.jl`.

Mark Towers, Jordan K Terry, Ariel Kwiatkowski, John U. Balis, Gianluca de Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Arjun KG, Markus Krimmel, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Andrew Jin Shen Tan, and Omar G. Younis. Gymnasium. URL `https://github.com/Farama-Foundation/Gymnasium`.

Robin Verschueren, Gianluca Frison, Dimitris Kouzoupis, Jonathan Frey, Niels Van Duijkeren, Andrea Zanelli, Branimir Novoselnik, Thivaharan Albin, Rien Quirynen, and Moritz Diehl. acados—a modular open-source framework for fast embedded optimal control. *Mathematical Programming Computation*, 14(1):147–183, March 2022. ISSN 1867-2949, 1867-2957. doi: 10.1007/s12532-021-00208-8. URL `https://link.springer.com/10.1007/s12532-021-00208-8`.

Jiayi Weng, Huayu Chen, Dong Yan, Kaichao You, Alexis Duburcq, Minghao Zhang, Yi Su, Hang Su, and Jun Zhu. Tianshou: A highly modularized deep reinforcement learning library. *Journal of Machine Learning Research*, 23(267):1–6, 2022. URL `http://jmlr.org/papers/v23/21-1127.html`.