

JAX 4 Deep RL

Glazer-Cavanagh W., Aguilar-Lussier, S.

April 26, 2023

Abstract

Research engineers need a good knowledge of the state of the art in terms of both theoretical algorithms and implementation frameworks. To this end, our team wants to catch up to speed on the most important ideas in the field of reinforcement learning as well as the progress in software frameworks of the past decade. In this work, we contribute to OpenAI’s SpinningUp ([Achiam, 2018](#)) which is an educative resource on deep reinforcement learning research (deep RL). SpinningUp implements various policy gradient algorithms in PyTorch ([Paszke et al., 2019](#)) as well as TensorFlow ([Abadi et al., 2016](#)) but not in JAX ([Bradbury et al., 2018](#)). JAX is a framework maintained by Google’s DeepMind which is quickly gaining in popularity¹. Our contribution is to re-implement REINFORCE ([Duan et al., 2016](#)) and TD3 ([Fujimoto et al., 2018](#)) in the JAX framework and compare differences in speed, scalability, performance and ease of implementation.

1 Project Introduction

For this work we re-implement 2 landmark papers in the field of RL in the JAX framework, namely REINFORCE and TD3. Our first task is to learn the JAX framework as we have never used it before. Through the learning process, we also aim to figure out how well suited JAX is for RL algorithms in terms of ease of implementation, speed and reproducibility versus other frameworks (PyTorch, TensorFlow).

To gauge JAX’s suitability for RL applications, we will discuss the performances of our algorithms when compared to PyTorch ([Paszke et al., 2019](#)) by profiling their run time with the python package `CProfile` ([Rosen and Czotter, 2006](#)). We compare different functions within the training loop and host an online interactive website allowing users to visualise our results with `snakeviz` ([Davis, 2012](#)). We also qualitatively go over our impressions after re-implementing algorithms in JAX. Since JAX is a functional programming framework and other ML frameworks use an object-oriented approach, this change in programming paradigm is significant and requires adaptation for newcomers who have never worked with pure functions.

¹JAX currently has 21.8k GitHub stars, which is 1/3 of PyTorch’s

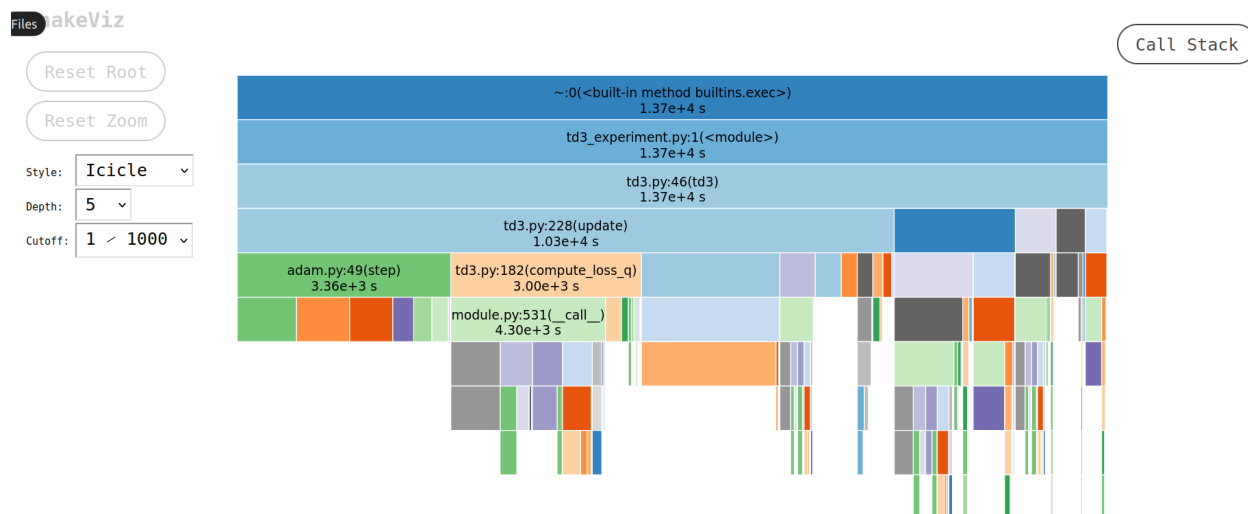


Figure 1: Example of profiling for PyTorch’s TD3 training loop accessible [here](#)

We are not concerned with achieving state of the art performance, but require decent performance to ensure that our JAX implementations are correct. To this end we debug and evaluate our algorithms on environments within MuJoCo (Todorov et al., 2012), namely on the **HalfCheetah** environment against the baseline SpinningUp (Achiam, 2018) implementations of REINFORCE and TD3. Other environments are not a priority and are considered bonus. We will do a hyper-parameter search on our algorithms to find the best hyper parameters.

2 Related Works

Many state of the art papers in the deep RL literature use JAX to scale their experiment or compute time. The framework come from Google’s DeepMind itself and is thus trusted in the research literature as well as in production. The two following methods used JAX for the novel algorithms: (Hessel et al., 2021) and (Xu et al., 2020). We also found that some performance improvements can be added to RL pipelines with JAX as done here (Flajolet et al., 2022), allowing in some cases to run bigger experiment without requiring large compute clusters. A lot of work in RL is also being done with JAX as a tool for production use, implying it is a mature tool. For example, DeepMind use JAX in almost all of their machine learning projects to accelerate their training and implementations including in their RL research (DeepMind, 2020). This also entails that there is little literature such in our setting analyzing JAX from a meta viewpoint against other DL frameworks.

As for reliable reinforcement learning algorithm baselines, many repositories exist agglomerating various algorithms in PyTorch, TensorFlow and some even using JAX. CleanRL (Huang et al., 2022) supports baselines for all three frameworks and even implements TD3. Stable-Baselines (Hill et al., 2018) is a fork from the openAI baseline project (Dhariwal et al., 2017) based on SpinningUp (Achiam, 2018) which we use for this project. SpinningUp has a much more beginner friendly approach which is less aimed at achieving state of the art

performance. It is akin to a learning resource which motivates our decision to use it as a starting point.

3 Background and Frameworks

- **JAX** : This framework is at the core of our project, we will use the DeepMind library ecosystem to limit manual implementations (Haiku, Optax, RLax) ([DeepMind, 2020](#)).
- **PyTorch and TensorFlow** : These are the two most known ML frameworks and those supported by SpinningUp. We chose PyTorch to be used for our experiments as we are more familiar with the framework.
- **OpenAI's Spinning Up** : We will use this resource as our main theoretical reference for deep RL and use its code base as our baselines for compute time and performance.

Automatic differentiation libraries have been the backbone of the recent deep learning progresses. With the increasing scale and demand in compute to train larger and larger networks, many problems emerge in terms of reproducible, distributed training, hardware acceleration and code optimization. JAX solves many of these issues by greatly simplifying automatic function differentiation and using just in time compilation (JIT) with accelerated linear algebra (XLA) for tensor processing units (TPUs). This hands off approach to a number of advanced features means that JAX is adopted by many of the top research institutions.

Whilst we do not have time in the context of this class to test many of these advanced features, we do have sufficient time to test out how well suited JAX is to deep RL applications and how much of a boost in performance we can expect to get for smaller scale applications.

As for the algorithms implemented, we use REINFORCE as well TD3. REINFORCE also known as Vanilla Policy Gradient (VPG) is an on policy, policy gradient algorithm leveraging an advantage function. Advantage functions $A^\pi(a, s) = Q^\pi(a, s) - V^\pi(s)$ estimate how much better an action is than other actions for a given policy at a given state. TD3 on the other hand is an off-policy algorithm based on DDPG ([Lillicrap et al., 2019](#)) with added double Q learning $\min_{i=1,2} Q_{\phi_{\text{tar},i}}$, delayed policy updates and added noise to the target policy action $a'(s') = \mu_{\theta_{\text{tar}}} + N(0, \sigma)$. The detailed algorithms are in the annex ([VPG](#), [TD3](#)) due to formatting constraints and are taken as is from SpinningUp.

4 Methodology

4.1 Implementation

We first begin by adapting the SpinningUp codebase to support GPU operations for tensors instead of using a message passing interface (MPI) to parallelize the code. This requires to adapt the existing code base to remove all MPI calls. Once this step is over with, we can monitor the time required for PyTorch VPG as well as the expected performance after 300

epochs. We chose SpinningUp instead of CleanRL since we wanted to implement VPG as our first baseline which is not present within the CleanRL repository.

We chose VPG first since it is an easy to implement on-policy algorithm which does not require any replay buffer. We implement this algorithm in JAX using the recommended DeepMind ecosystem (Babuschkin et al., 2020) of `Optax` for optimizers and `Haiku` for neural networks. `Flax` (Heek et al., 2023) is the default library which we chose not to use initially due to DeepMind recommendations, but in hindsight, this library has a much more conventional API which would have probably saved us some implementation and debugging time.

Once we achieve similar performance in terms of compute time per epoch as well as metrics within the HalfCheetah environment, we move on to the more complex TD3 algorithm in JAX. For this part of our work we can leverage most of the code base developed previously and follow closely the SpinningUp implementation for added implementation details. We similarly benchmark our JAX implementation with the PyTorch implementation to ensure correctness.

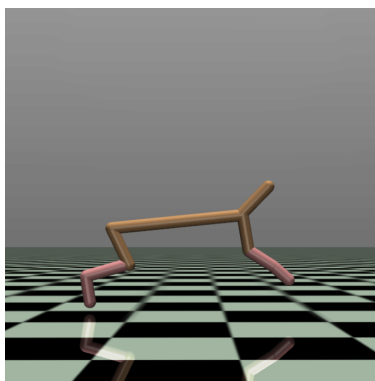


Figure 2: Example of HalfCheetah environment

4.2 Benchmarks

Our main goal was to create the most consistent and fair time performance benchmark for our training algorithms comparison. To do so we ran our profiling on the same computer with no other programs running apart from the kernel. We only ran the profiling on the epoch loops after the warmup phase (1 full epoch) allowing each algorithm to setup caches required for training. We also update the SpinningUp codebase python version from 3.6 to 3.11 to match environments, dependencies and compare current version of both PyTorch and JAX.

Component	Specifications
CPU	Intel® Core™ i5-10400F, 12 cores, 12M Cache, up to 4.30 GHz
GPU	GPU RTX 3060, 12GB vram
RAM	16 GB

Table 1: Benchmark PC specifications

In order to validate our implementation correctness, we set out to match SpinningUp’s

benchmarks. To optimize performance, we used a grid search around SpinningUp’s recommended hyper parameters on HalfCheetah-v2 environment. Due to time constraints, we were unable to test more environments. We ran our grid search with half the total steps than the SpinningUp benchmarks to accelerate the search process and then we ran our best configurations with triple the steps as SpinningUp and three different seeds to ensure we reduced stochasticity. SpinningUp’s documentation recommends using a network size of (64,32) and 4000 steps per epochs for VPG and a network size of (256,256) and a mini-batch size of 100 with all other parameters set at default value. Here are the grid search parameter we explored :

Algorithm	Network Hidden layers sizes	Num Steps per epoch	Batch Size
VPG	(64, 32), (64, 64), (128, 64), (128, 128, 64), (128, 128, 128)	3000 4000,5000, 10000	-
TD3	256,256), (256,256,128), (256,256,256) (256,256,256,256)	4000(default)	100, 500,1000

Table 2: Grid search parameters explored

5 Learned Skills

Since our project is centered around the JAX(Bradbury et al., 2018) framework, this is the first skill that we will have learned. Neither of us had experience using JAX which represented a big learning opportunity for us. JAX is a relatively new ML framework that offers great speedups when compared to PyTorch(Paszke et al., 2019) and TensorFlowAbadi et al. (2016), especially for distributed computing. It’s functional approach also makes it stateless and fully reproducible. We believe that it is a great tool to add to our list of known frameworks. With this project, we also acquired skills in functional programming and performance optimisation for ML tasks.

Another big part of this project was to get familiar with the implementations of some deep RL models. We needed to understand and re-implement them in the new proposed framework which allowed us to understand much better the algorithms VPG and TD3. Finally we familiarized ourselves with the learning resource that is Open AI’s SpinningUp and its code base. OpenAI has provided the world with some of the best ML work and SpinningUp is their reference for everything RL, we learned a lot from this tool and are thankful to have it for any RL projects we may undertake in the future.

6 Experiments and Discussion

6.1 A Qualitative Overview of JAX for RL

Traditional deep learning frameworks like PyTorch and TensorFlow adopt an object oriented paradigm. For example, in `torch`, modules must implement the `nn.Module` and define

Pros	Cons
Reproducibility and stability with controllable pseudo-randomness	Manual implementation of all additional features, which is quite involved (optimizers, parameters, modules, randomness)
Modularity and composability of loss functions to derivate	Steep learning curve of functionnal approach
Integration with NumPy by re-using the same API	Obscure stack trace makes debugging functions harder when they are JITed
Speed gains of using JIT with automatic batching of function to parallelize compute	Not for new developers as this framework is more involved and would be a nightmare for beginners to begin their ML journey.
Rich ecosystem for parallel simulation environments and RL algorithms supported by Google and DeepMind	Hands on installation since the library lacks some maturity and does not support all dev environments
Hands Off Distributed Computing when compared to Torch and TF with vmap and pmap	

Table 3: List of pros and Cons for the JAX framework

`nn.Params` for the optimizer to know with respect to which parameters we optimize our loss function. This approach means that an object's prediction is dependent on it's state which can have many side-effects on other sub-module, making the whole process hard to reproduce.

JAX uses a functionally pure approach to optimize it's own code with just-in-time compilation (JIT). This means that functions are not allowed to impact global variables or the state of input parameters which are treated as immutable. This ensures strong guarantees that a function called with specific immutable parameters (mostly `dict`, `dataclasses` or `tuple`) will always have the same output. This approach is destabilizing at first, but simple, elegant and very modular once you understand the inner workings.

6.2 Algorithm Performances on HalfCheetah

We managed to achieve similar performance as [SpinningUp's benchmarks](#) with our re-implementation of spinningup algorithms in PyTorch with training on GPU. We can see in the [grid search results of our implementation of TD3](#) that our best parameters reach above 10'000 in return and in the [grid search results of our implementation of VPG](#) that we reach above 0. During the experimentation to reach our [best results with TD3](#) and our [best results with VPG](#) we realised that randomness affects greatly the outcome of experiments and it was hard to reproduce consistent results even with the same seeds. The linked graph represent our best runs with the same code and seeds.

Due to unexpected delays in the JAX implementation, we were only able to do a hyper-parameter search in PyTorch and used the best hyper-parameters for both implementations

for our final experiments and profiling. We can see in figures below that the JAX implementations didn't reach the same performances on the environments with the configurations that were found with the PyTorch experimentation. However these results are good enough to prove functionality and we hypothesize that we could get better performances if we did the same tuning process on JAX than PyTorch or if we ran the experiments for more steps. Since the main focus of the project is time profiling, we kept the same configurations and number of steps to have a fair comparison in run time.

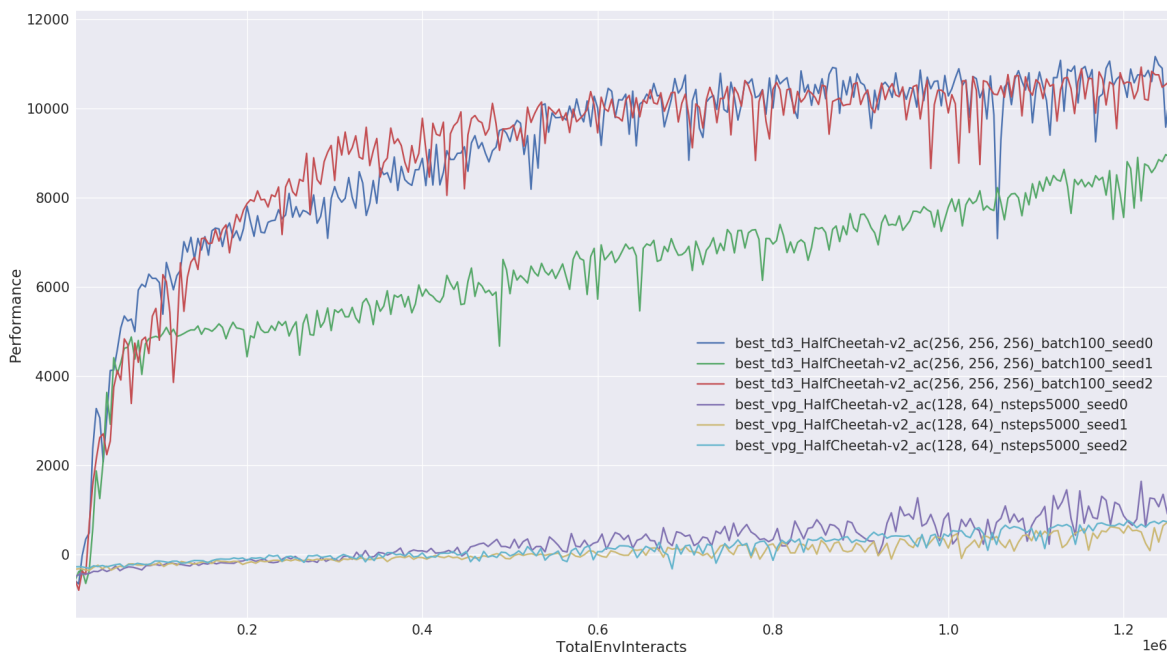


Figure 3: Best model runs with multiple seeds and best configurations on PyTorch

Above we can see the absolute best return obtained, and under we can see the return obtained for the runs used for the profiling, we can see that even with the same configurations and seeds, randomness affects results a lot. Since we didn't experiment tuning with JAX our profiling runs are our best results.

Algorithm	Network Hidden layers sizes	Num Steps per epoch	Batch Size	JAX return	Pytorch return
VPG	(128, 64)	5000	-	-50 ± 50	-50 ± 150
TD3	(256,256,256)	4000(default)	100	$5\,000 \pm 300$	6000 ± 3500

Table 4: Return of the final runs used for profiling (mean and std of 3 seeds for the return rounded to significant value)

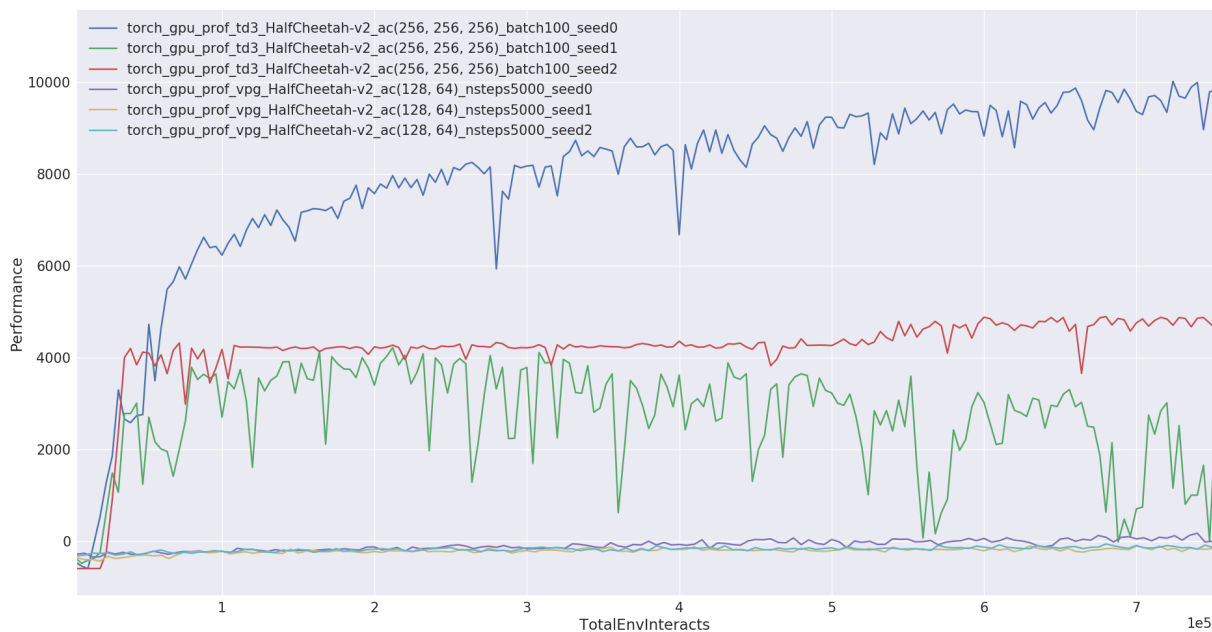


Figure 4: Return graph of the runs used for profiling in PyTorch



Figure 5: Return graph of the runs used for profiling in JAX

6.3 Speed Comparison

For the profiling, we use the same three seeds as for the algorithm performance and we take the median seed run time to report our results. Since there is little variation ($>1\%$) between seeds for run time, we don't report standard deviations or means and take the median. Then we average our results over all epochs and compare JAX's run time relative to PyTorch's for the update function.

This update function includes all computations required for the algorithm outside of the simulation environment, so no MuJoCo functions are included in our estimates. We only measure the time required for framework specific computations which are described in the figure below.

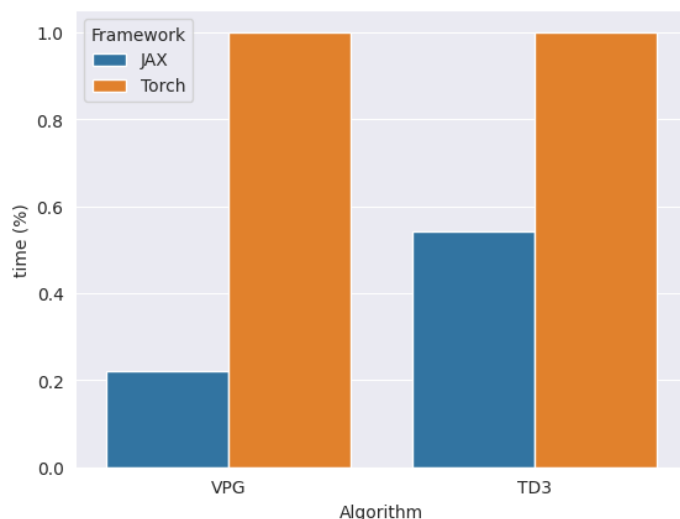


Figure 6: Average speed comparison between JAX and PyTorch for 1 epoch in relative percentage to PyTorch's run time for the loss function as well as it's gradient and the update of model parameters with the Adam optimizer.

7 Profiling & Video Results

Since our research is mostly focused on time performances and the first steps were oriented on benchmarks we provide visual representation of the comparative profiling runs we did on VPG and TD3 in JAX and PyTorch with videos of the environment to confirm functionality.

8 Conclusion

We believe that we learned a lot of the algorithms we have re-implemented since we had to build them from scratch in a new functional framework. We also learned a lot from JAX and it's functional approach. We learned how to benchmark performances on algorithms from a speed comparison and managed to get an online visualisation tool up and running.

For practitioners, we would finally recommend JAX frameworks only for experienced developers with an interest in learning a new functional approach to enrich their ML/RL








Algorithm	Framework	Profiling	Best Video performance
VPG	JAX		
	Torch		
TD3	JAX		
	Torch		

Table 5: Outgoing links for our online profiling visualisations and video of performances on HalfCheetah environment. Note the videos were cherry picked to be the best return obtained by each framework/algorithm

tool set, as the framework differs quite significantly from other traditional DL frameworks. We would also advise the usage of the `flax` library in lieu of `haiku` since the API is much more straightforward.

Finally, our project’s scope is adequate since we were modular in the number of algorithms we chose to implement on on how many environments we evaluated them. Thus we were able to adapt our experiments according to our progress and/or problems.

9 Work Division

1. William Glazer-Cavanagh:

- Fundamentals of JAX with MNIST and the default DeepMind package suite
- adaptation of spinningUp’s VPG/TD3 codabase to work with JAX functions
- Removed MPI support from SpinningUp

2. Samuel Aguilar-Lussier:

- Converted SpinningUp codebase from threaded algorithms in python3.6 to GPU-native on PyTorch with Python3.11
- Evaluation of baselines in PyTorch for VPG, TD3 with GPU training on PyTorch
- Grid search for best performances on HalfCheetah with GPU training on PyTorch
- Code profiling for VPG, TD3 in JAX and PyTorch

All the resources of this project can be found at :

- Final code with both algorithm implementations and profiling : <https://github.com/williamGlazer/robot-learning>
- Work for PyTorch conversion to GPU support : <https://github.com/SamuelA-L/Spinning-up-RL-project>
- Figures and all media (graphs, videos, interactive profiling) : <https://github.com/SamuelA-L/rl-profiling>

References

- Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016). Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283.
- Achiam, J. (2018). Spinning Up in Deep Reinforcement Learning.
- Babuschkin, I., Baumli, K., Bell, A., Bhupatiraju, S., Bruce, J., Buchlovsky, P., Budden, D., Cai, T., Clark, A., Danihelka, I., Dedieu, A., Fantacci, C., Godwin, J., Jones, C., Hemsley, R., Hennigan, T., Hessel, M., Hou, S., Kapturowski, S., Keck, T., Kemaev, I., King, M., Kunesch, M., Martens, L., Merzic, H., Mikulik, V., Norman, T., Papamakarios, G., Quan, J., Ring, R., Ruiz, F., Sanchez, A., Schneider, R., Sezener, E., Spencer, S., Srinivasan, S., Stokowiec, W., Wang, L., Zhou, G., and Viola, F. (2020). The DeepMind JAX Ecosystem.
- Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. (2018). JAX: composable transformations of Python+NumPy programs.
- Davis, M. (2012). snakeviz. <https://github.com/jiffyclub/snakeviz>.
- DeepMind (2020). Using jax to accelerate our research.
- Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., Wu, Y., and Zhokhov, P. (2017). Openai baselines. <https://github.com/openai/baselines>.
- Duan, Y., Chen, X., Houthooft, R., Schulman, J., and Abbeel, P. (2016). Benchmarking deep reinforcement learning for continuous control.
- Flajolet, A., Monroc, C. B., Beguir, K., and Pierrot, T. (2022). Fast population-based reinforcement learning on a single machine. In Chaudhuri, K., Jegelka, S., Song, L., Szepesvari, C., Niu, G., and Sabato, S., editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 6533–6547. PMLR.
- Fujimoto, S., van Hoof, H., and Meger, D. (2018). Addressing function approximation error in actor-critic methods.
- Heek, J., Levskaya, A., Oliver, A., Ritter, M., Rondepierre, B., Steiner, A., and van Zee, M. (2023). Flax: A neural network library and ecosystem for JAX.
- Hessel, M., Kroiss, M., Clark, A., Kemaev, I., Quan, J., Keck, T., Viola, F., and van Hasselt, H. (2021). Podracer architectures for scalable reinforcement learning. *CoRR*, abs/2104.06272.
- Hill, A., Raffin, A., Ernestus, M., Gleave, A., Kanervisto, A., Traore, R., Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., and Wu, Y. (2018). Stable baselines. <https://github.com/hill-a/stable-baselines>.

- Huang, S., Dossa, R. F. J., Ye, C., Braga, J., Chakraborty, D., Mehta, K., and Araújo, J. G. (2022). Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *Journal of Machine Learning Research*, 23(274):1–18.
- Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2019). Continuous control with deep reinforcement learning.
- Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E. Z., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. *CoRR*, abs/1912.01703.
- Rosen, B. and Czotter, T. (2006). Cprofile. <https://github.com/python/cpython/blob/main/Lib/cProfile.py>.
- Todorov, E., Erez, T., and Tassa, Y. (2012). Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE.
- Xu, Z., van Hasselt, H. P., Hessel, M., Oh, J., Singh, S., and Silver, D. (2020). Meta-gradient reinforcement learning with an objective discovered online. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H., editors, *Advances in Neural Information Processing Systems*, volume 33, pages 15254–15264. Curran Associates, Inc.

A SpinningUp VPG PseudoCode

Algorithm 1 Vanilla Policy Gradient Algorithm

- 1: Input: initial policy parameters θ_0 , initial value function parameters ϕ_0
- 2: **for** $k = 0, 1, 2, \dots$ **do**
- 3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
- 4: Compute rewards-to-go \hat{R}_t .
- 5: Compute advantage estimates, \hat{A}_t (using any method of advantage estimation) based on the current value function V_{ϕ_k} .
- 6: Estimate policy gradient as

$$\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^T \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) |_{\theta_k} \hat{A}_t.$$

- 7: Compute policy update, either using standard gradient ascent,

$$\theta_{k+1} = \theta_k + \alpha_k \hat{g}_k,$$

or via another gradient ascent algorithm like Adam.

- 8: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg \min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^T \left(V_{\phi}(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.

- 9: **end for**
-

B SpinningUp TD3 PseudoCode

Algorithm 1 Twin Delayed DDPG

- 1: Input: initial policy parameters θ , Q-function parameters ϕ_1, ϕ_2 , empty replay buffer \mathcal{D}
- 2: Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta, \phi_{\text{targ},1} \leftarrow \phi_1, \phi_{\text{targ},2} \leftarrow \phi_2$
- 3: **repeat**
- 4: Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{\text{Low}}, a_{\text{High}})$, where $\epsilon \sim \mathcal{N}$
- 5: Execute a in the environment
- 6: Observe next state s' , reward r , and done signal d to indicate whether s' is terminal
- 7: Store (s, a, r, s', d) in replay buffer \mathcal{D}
- 8: If s' is terminal, reset environment state.
- 9: **if** it's time to update **then**
- 10: **for** j in range(however many updates) **do**
- 11: Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from \mathcal{D}
- 12: Compute target actions

$$a'(s') = \text{clip}(\mu_{\theta_{\text{targ}}}(s') + \text{clip}(\epsilon, -c, c), a_{\text{Low}}, a_{\text{High}}), \quad \epsilon \sim \mathcal{N}(0, \sigma)$$

- 13: Compute targets

$$y(r, s', d) = r + \gamma(1 - d) \min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', a'(s'))$$

- 14: Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} (Q_{\phi_i}(s, a) - y(r, s', d))^2 \quad \text{for } i = 1, 2$$

- 15: **if** $j \bmod \text{policy_delay} = 0$ **then**
- 16: Update policy by one step of gradient ascent using

$$\nabla_{\theta} \frac{1}{|B|} \sum_{s \in B} Q_{\phi_1}(s, \mu_\theta(s))$$

- 17: Update target networks with

$$\begin{aligned} \phi_{\text{targ},i} &\leftarrow \rho \phi_{\text{targ},i} + (1 - \rho) \phi_i & \text{for } i = 1, 2 \\ \theta_{\text{targ}} &\leftarrow \rho \theta_{\text{targ}} + (1 - \rho) \theta \end{aligned}$$

- 18: **end if**
 - 19: **end for**
 - 20: **end if**
 - 21: **until** convergence
-