Parallel Q-Learning: Scaling Off-policy Reinforcement Learning under Massively Parallel Simulation

Zechu Li [*1] **Tao Chen** [*1] **Zhang-Wei Hong** [1] **Anurag Ajay** [1] **Pulkit Agrawal** [1]

Abstract

Reinforcement learning is time-consuming for complex tasks due to the need for large amounts of training data. Recent advances in GPU-based simulation, such as Isaac Gym, have sped up data collection thousands of times on a commodity GPU. Most prior works have used on-policy methods like PPO due to their simplicity and easy-to-scale nature. Off-policy methods are more sample-efficient, but challenging to scale, resulting in a longer wall-clock training time. This paper presents a novel Parallel Q-Learning (PQL) scheme that outperforms PPO in terms of wall-clock time and maintains superior sample efficiency. The driving force lies in the parallelization of data collection, policy function learning, and value function learning. Different from prior works on distributed off-policy learning, such as Apex, our scheme is designed specifically for massively parallel GPU-based simulation and optimized to work on a single workstation. In experiments, we demonstrate the capability of scaling up Q-learning methods to *tens of thousands of parallel environments* and investigate important factors that can affect learning speed, including the number of parallel environments, exploration strategies, batch size, GPU models, etc. The code is available at https://github.com/Improbable-AI/pql.

1. Introduction

Reinforcement learning (RL) has achieved impressive results on many real-world problems, such as robotics (Kober et al., 2013; Miki et al., 2022) and drug discovery (Popova et al., 2018). A primary challenge in using RL is the need for large amounts of real-world data. There are two main strategies to tackle this problem. One is to improve the sample efficiency of RL algorithms (Mnih et al., 2015; Lillicrap et al., 2015) to make better use of available data. The other is to reduce the need for real-world data collection by training policies in simulation and deploying them in the real world (Hwangbo et al., 2019; OpenAI et al., 2020; Margolis et al., 2022; Miki et al., 2022; Chen et al., 2022a). In sim-to-real pipelines, the training wall-clock time matters more than the sample efficiency — faster training can speed up the experiment cycle and unlock the potential for addressing a broader range of complex problems.

The community has widely recognized the need for faster training, leading to the development of several distributed frameworks (Horgan et al., 2018; Espeholt et al., 2018). However, these frameworks usually operate at a server scale that requires hundreds or thousands of computers in a cluster, making them impractical for most researchers and practitioners. Specifically, most of these computers are used to run multiple simulator instances in parallel to speed up data collection. Recent advance in GPU-based simulation, such as Isaac Gym (Makoviychuk et al., 2021), has mitigated the need for a large number of machines as it enables the parallel simulation of *tens of thousands* of environments on one GPU. A natural question that arises in this massively parallel setting is: what RL algorithm is suitable to achieve better wall-clock time? Many prior works (Allshire et al., 2021; Rudin et al., 2022; Chen et al., 2022b) use on-policy algorithms like PPO (Schulman et al., 2017) for training policies in Isaac Gym due to their simplicity and easy-to-scale nature. However, they suffer from low sample efficiency, in which data collection can still take most of the total wall-clock time.

Intuitively, by virtue of requiring less data than on-policy algorithms, off-policy algorithms (Q-learning methods, in particular) should reduce the wall-clock time of training. However, better sample efficiency will not lead to shorter training time if the algorithm cannot make good use of parallel environments. Some prior works (Nair et al., 2015; Horgan et al., 2018) have developed distributed frameworks

[*]Equal contribution [1]Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, USA. Correspondence to: Zechu Li <zechu@mit.edu>, Tao Chen <taochen@mit.edu>, Pulkit Agrawal <pulkitag@mit.edu>.

Proceedings of the 40th International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023. Copyright 2023 by the author(s).

for off-policy methods to leverage parallel environments. However, these frameworks have only shown successful scaling with hundreds of parallel environments (for example, maximumly 256 environments in (Horgan et al., 2018)). Now that GPU-based simulation enables *tens of thousands* of parallel environments on a single GPU, it remains unclear whether off-policy methods can work efficiently in this case. For instance, if there are $10,000$ parallel environments and we still use the typical replay buffer capacity (say 1M samples), the entire replay buffer is refreshed every 100 environment steps, making the data in the replay buffer more like on-policy samples. Do off-policy methods still work in this scenario? One can also increase the replay buffer capacity, but this is limited by the memory size of the hardware.

In this work, we investigate how to scale up Q-learning to tens of thousands of environments. We present our approach, **P**arallel **Q**-**L**earning (**PQL**), which can be deployed on a workstation. The learning speed in PQL is boosted by parallelizing the data collection, policy function learning, and value function learning on a single workstation. This allows for collecting more simulation data and updating value/policy functions more times in a given time window, leading to an improvement in the training wall-clock time. Achieving such parallelization would be non-trivial for on-policy algorithms, as the policy update requires on-policy interaction data, which means that data collection and policy updates need to happen in sequence.

Our main contributions are summarized as follows:

- We present a scheme for time-efficient reinforcement learning, **PQL**, that can efficiently leverage tens of thousands of parallel environments on a workstation.
- We thoroughly investigate the effect of important hyperparameters such as the speed ratio on different processes that control the resource allocation and provide empirical guidelines for tuning these values to scale up Q-learning.
- We deploy different exploration strategies in parallel environments, which leads to robust exploration and mitigates the hassle of tuning the exploration noise.
- We demonstrate the effectiveness of our method on six Isaac Gym benchmark tasks (Makoviychuk et al., 2021) and show its superiority over state-of-the-art (SOTA) on-/off-policy algorithms. Our method **PQL** achieves both faster learning in wall-clock time and better sample efficiency. Empirically, we also found that DDPG performs better than SAC with massively parallel environments.

2. Related Work

Massively Parallel Simulation Simulation has been an important tool in various research fields, such as robotics, drug discovery, and physics. In the past, researchers have used simulators like MuJoCo (Todorov et al., 2012) and PyBullet (Coumans & Bai, 2016) for rigid body simulation. Recently, there has been a new wave of development in GPU-based simulation, e.g., Isaac Gym (Makoviychuk et al., 2021). GPU-based simulation has substantially improved simulation speed by allowing massive amounts of parallel simulation on a single commodity GPU. It has been used in various challenging robotics control problems, including quadruped locomotion (Rudin et al., 2022; Margolis et al., 2022) and dexterous manipulation (Allshire et al., 2021; Chen et al., 2022b;a). With fast simulation, one can obtain much more environment interaction data in the same training time as before. This poses a challenge to RL algorithms in making the best use of the massive amount of data. A straightforward way is to use on-policy algorithms such as PPO, which can be easily scaled up and is also the default algorithm used by researchers in Isaac Gym. However, on-policy algorithms are less data-efficient. In our work, we investigate how to scale up off-policy algorithms to achieve higher sample efficiency and shorter wall-clock training time under massively parallel simulation.

Distributed Reinforcement Learning There have been numerous distributed reinforcement learning frameworks to speed up learning. One line of work focuses on Q-learning methods. Gorila (Nair et al., 2015) distributes DQN agents to many machines where each machine has its local environment, replay buffer, and value learning, and uses asynchronous SGD for a centralized Q function learning. Similarly, Popov et al. (2017) apply asynchronous SGD to the DDPG algorithm (Lillicrap et al., 2015). Combining with prioritized replay (Schaul et al., 2015), n-step returns (Sutton, 1988), and double-Q learning (Hasselt, 2010), Horgan et al. (2018) (Ape-X) parallelize the actor thread (environment interactions) for data collection and use a centralized learner thread for policy and value function learning. Built upon Ape-X, Kapturowski et al. (2018) adapt the distributed prioritized experience replay for RNN-based DQN agents.

Another line of work improves the training speed on policy gradient methods. A3C (Mnih et al., 2016) uses asynchronous SGD across many CPU cores, with each running an actor learner on a single machine. Babaeizadeh et al. (2016) develop a hybrid CPU/GPU implementation of A3C, but it can have poor convergence due to the stale off-policy data being used for the on-policy update. Espeholt et al. (2018) (IMPALA) introduce an off-policy correction scheme (V-trace) to mitigate the lagging issue between the actors and learners in distributed on-policy settings. Espeholt et al. (2019) further improve the IMPALA training

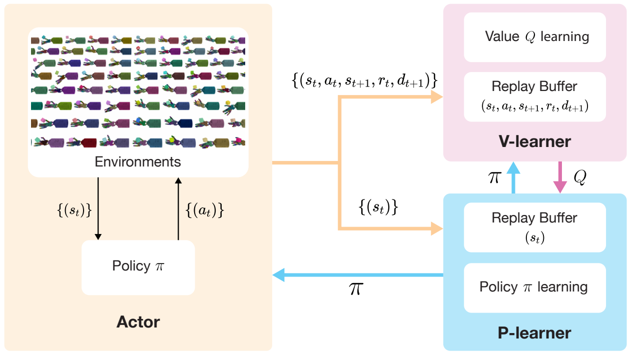


Figure 1. Overview of Parallel Q Learning (PQL). We have three concurrent processes running: **Actor**, **P-learner**, **V-learner**. **Actor** collects interaction data. **P-learner** updates the policy network. **V-learner** updates the Q functions.

speed by moving the policy inference from the actor to the learner. Clemente et al. (2017) parallelize the environments for synchronous advantage actor-critic. Heess et al. (2017) propose a distributed version of PPO (Schulman et al., 2017) for training various locomotion skills in a diverse set of environments. Wijmans et al. (2019) develop a decentralized version of distributed PPO to mitigate the synchronization overhead between different actor processes and applies it to a point-goal navigation task.

Our scheme is most closely related to Ape-X (Horgan et al., 2018) but has a number of key differences. **First**, our scheme is specifically designed for massively ($>> 1000$) parallel GPU-based simulation. Our scheme is optimized for a single-machine setup, which can help democratize large-scale RL research. **Second**, we further decouple and parallelize the learning with two separate learners for policy function learning and Q-function learning, respectively. **Third**, we allocate a local replay buffer for each learner. This can reduce the communication cost between the replay buffer and the learners. **Fourth**, working with a single machine presents new challenges in balancing the computing resource between different parallel processes. Our scheme offers a mechanism to balance the computing resource among different processes.

3. Method

We developed a parallel off-policy training scheme, **P**arallel **Q-L**earning (**PQL**), for massively parallel GPU-based simulation, where thousands of environments can be simulated simultaneously on a single GPU. In a typical actor-critic Q-learning method, three components run sequentially: a policy function, a Q-value function, and an environment. Agents roll out the policy in the environments and collect interaction data, which is added to a replay buffer; then, the value function is updated to minimize the Bellman error, after which the policy function is updated to maximize the Q values. This sequential scheme slows down the training, as each component needs to wait for the other two to finish before proceeding. To maximize the learning speed and reduce the waiting time, we parallelize the computation of all three components. This allows for more network updates per data collection, which can improve the utilization of the massive amount of data and lead to better training speed, as demonstrated in the experiments. Off-policy RL methods are well-suited for parallelization as the interaction data in a replay buffer does not need to come from the latest policy. In contrast, on-policy methods such as PPO require using the rollout data from the latest policy (on-policy data) to update the policy, thus making it non-trivial to parallelize the data collection and policy/value function update.

Our scheme is optimized for training speed in terms of wall-clock time and can be readily applied on a workstation. It is built upon DDPG (Lillicrap et al., 2015), but can be easily extended to other off-policy algorithms such as SAC (Haarnoja et al., 2018) (see Appendix C). Our scheme also incorporates common techniques used to improve Q learning performance, such as double Q learning (Hasselt, 2010) and n-step returns (Sutton, 1988). Furthermore, we experimented with adding distributional RL (Bellemare et al., 2017) to PQL, which we refer to as **PQL-D**. While it improves performance on challenging manipulation tasks, it leads to a slight decrease in the convergence speed of the RL agent. In this paper, we use the following notation: at time step t, s_t represents observation data, a_t represents action command, r_t represents the reward, d_t represents whether the environment terminates, $\pi(s_t)$ represents the policy network, $Q(s_t, a_t)$ represents the Q network, $Q'(s_t, a_t)$ represents the target Q network, and N represents the number of parallel environments.

3.1. Scheme Overview

PQL parallelizes data collection, policy learning, and value learning into three processes, as shown in Figure 1. We refer to them as **Actor**, **P-learner**, and **V-learner**, respectively.

- **Actor**: We collect a batch of interaction data using parallel environments. We use Isaac Gym (Makoviychuk et al., 2021) as our simulation engine, which supports massively parallel simulation. Note that we do not make any Isaac-Gym-specific assumptions, and PQL is optimized for any GPU-based simulator that supports a large number of parallel environments. In the **Actor** process, the agent interacts with tens of thousands of environments according to an exploration policy. Therefore, we maintain a local policy network $\pi^a(s_t)$, which is periodically synchronized with the policy network $\pi^p(s_t)$ in **P-learner** (which we explain below).

- **V-learner**: We create a dedicated process for training value functions, which allows for continuous updates without being interrupted by data collection or policy network updates. To compute the Bellman error, we need to have a policy network to estimate the optimal action to take and a replay buffer to sample a batch of I.I.D. training data. Since we use a dedicated process to keep updating value functions, **V-learner** must frequently query the policy network and sample data from the replay buffer. To reduce the communication overhead of network and data across processes, we maintain a local policy network $\pi^v(s_t)$ and a local replay buffer ($\{(s_t, a_t, s_{t+1}, r_t, d_{t+1})\}$) in **V-learner**. $\pi^v(s_t)$ gets synced with $\pi^p(s_t)$ in **P-learner** periodically. When the GPU memory is sufficiently large to host the entire replay buffer, which is usually the case when the observation is not images, we allocate the replay buffer directly on the GPU to avoid the CPU-GPU data transfer bottleneck.

- **P-learner**: We use another dedicated process for updating the policy network $\pi^p(s_t)$. The policy network $\pi^p(s_t)$ is optimized to maximize the $Q^p(s_t, \pi^p(s_t))$. Similarly, we keep a replay buffer of $\{(s_t)\}$ and a local value function $Q^p(s_t, a_t)$ in **P-learner** to reduce communication overhead across processes. $Q^p(s_t, a_t)$ is periodically updated with $Q_1^v(s_t, a_t)$ in **V-learner**.

We use Ray (Moritz et al., 2017) for parallelization. The pseudo-code for the scheme is shown in Algorithm 1, 2, and 3 in the appendix.

Data Transfer Suppose there are N parallel environments in the **Actor** process. At each rollout step, the **Actor** rolls out the policy $\pi_a(s_t)$ and generates N pairs of $(s_t, a_t, s_{t+1}, r_t, d_{t+1})$. Then the **Actor** sends the entire batch of interaction data $\{(s_t, a_t, s_{t+1}, r_t, d_{t+1})\}$ to the **V-learner** (as shown in Figure 1). Since policy update in **P-learner** only needs state information, **Actor** only needs to send $\{(s_t)\}$ to the **P-learner**.

Network Transfer The **V-learner** periodically sends the parameters of the $Q_1^v(s_t, a_t)$ to **P-learner**, which updates the local $Q^p(s_t, a_t)$ in **P-learner**. The **P-learner** sends the policy network $\pi^p(s_t)$ to both the **Actor** and **V-learner**.

Both the data transfer and network transfer happen concurrently.

3.2. Balance between Actor, P-learner, and V-learner

Our scheme allows the **Actor**, **P-learner**, and **V-learner** to run concurrently. However, we need to appropriately constrain the frequency of data collection, policy network update, and value network update. In other words, each process should not run on its own as fast as possible. Thus, we add explicit control on these three frequencies and define two ratios as follows:

$$\beta_{a:v} := \frac{f_a}{f_v} \qquad \text{and} \qquad \beta_{p:v} := \frac{f_p}{f_v},$$

where f_a is the number of rollout steps per environment in **Actor** per unit time, f_v is the number of Q function updates in **V-learner** per unit time, f_p is the number of policy updates in **P-learner** per unit time. $\beta_{a:v}$ determines how many Q function updates are performed in the **V-learner** when **Actor** rolls out the policy for one step with N environments. $\beta_{p:v}$ decides how many Q function updates are performed in **V-learner** when **P-learner** updates the policy once. Once the ratios are set, we monitor the progress of each process and dynamically adjust the speed of **Actor** and **P-learner** by letting the process wait if necessary.

Controlling the three processes via $\beta_{a:v}, \beta_{p:v}$ provides three major advantages. First, it allows us to balance the resource allocation of each process and reduce the variance of our scheme's performance. Given a fixed amount of computing resources, the ability to let some of the processes wait enables other processes to use the GPU resource more. This is particularly important when working with limited resources. If there is only one GPU, and all three processes run freely on it, simulation with a large number of environments can cause very high GPU utilization, which slows down the **P-learner** and **V-learner** and leads to worse performance. Note that such control was not examined in prior studies, such as Ape-X (Horgan et al., 2018), where a computer cluster was used for both the simulation and network training — the phenomenon of competing for limited computing resources (all three processes on one GPU) did not occur. On the other hand, leaving each process running freely creates more variance in the training speed and learning performance as the simulation speed and network training speed are heavily dependent on the task complexity, network size, computer hardware, etc. For example, simulation for contact-rich tasks can be slower than others; some tasks might require a deeper policy network or Q networks; even the GPUs on a machine might have different running conditions at different times, leading to different speeds across processes and further leading to different learning performances. Second, ratio control can improve convergence speed. For example, prior works (Fujimoto et al., 2018) have shown that updating the policy network less frequently than the Q functions leads to better learning. We show similar ratios in our experiments, but our findings are in the context of parallel training of policy and value function. Third, the ratio $\beta_{p:v}$ can be interpreted as the frequency of the target policy network update. One may notice that we use a lagged policy to update the Q function and synchronize it according to the above ratio. Therefore, we do not create a target policy network explicitly, but every synchro-

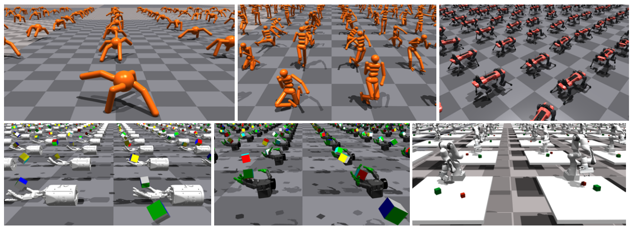


Figure 2. We experiment on six Isaac Gym tasks: *Ant*, *Humanoid*, *ANYmal*, *Shadow Hand*, *Allegro Hand*, *Franka Cube Stacking*.

nization can be considered as a hard update of the policy network.

3.3. Mixed Exploration

We can achieve improvement in convergence by having a good exploration strategy. Too much exploration can make agents fail to latch onto useful experience and learn a good policy quickly, while too little exploration does not give the agent enough good interaction data to improve the policy. Balancing the exploration and exploitation often requires extensive hyper-parameter tuning or complex scheduling mechanisms. In DDPG, one common practice to control exploration is to set the standard deviation σ of the uncorrelated and zero-mean Gaussian noise that is being added to the deterministic policy output ($a_t = \max(\min(\pi(s_t) + \mathcal{N}(0, \sigma), a_u), a_l)$ (Achiam, 2018; Fujita et al., 2021; Yang et al., 2022) where $a_t \in [a_l, a_u]$). Since it is difficult to predict how much exploration noise is appropriate, one typically needs to tune σ for each task. Can we mitigate the hassle of tuning σ? Our idea is that instead of finding the best σ value, we can try out different σ values altogether, which we call **mixed exploration**. Even if some σ values lead to bad exploration at a certain training stage, others can still generate good exploration data. This strategy is easily implemented thanks to the massively parallel simulation, as we can use different σ values in different parallel environments. Similar ideas have been used in prior works (Horgan et al., 2018; Mnih et al., 2016). In our work, we uniformly generate the noise levels in the range of $[\sigma_{\min}, \sigma_{\max}]$. For the i^{th} environment out of N environments, $\sigma_i = \sigma_{\min} + \frac{i-1}{N-1}(\sigma_{\max} - \sigma_{\min})$ where $i \in \{1, 2, ..., N\}$. We use $\sigma_{\min} = 0.05, \sigma_{\max} = 0.8$ for all the tasks in our experiments.

4. Experiments

In this section, we demonstrate the effectiveness of our method compared to SOTA baselines, show the effects of key hyper-parameters that affect learning, and provide empirical guidelines for setting these values. All experiments are carried out on a single workstation with a few GPUs. We run each experiment with five random seeds and plot their mean and standard error.

4.1. Setup

Tasks We evaluate our method on six Isaac Gym benchmark tasks (Makoviychuk et al., 2021): *Ant*, *Humanoid*, *ANYmal*, *Shadow Hand*, *Allegro Hand*, and *Franka Cube Stacking* (see Figure 2). For more details about these tasks, please refer to (Makoviychuk et al., 2021). Additionally, we provide two more tasks in Section 4.5: (1) a vision-based *Ball Balancing* task and (2) a contact-rich dexterous manipulation task that requires learning to reorient hundreds of different objects using a *DClaw Hand* with a single policy (Chen et al., 2022a).

Baselines We consider the following baselines: (1) **PPO** (Schulman et al., 2017), which is the default algorithm used by many prior works (Makoviychuk et al., 2021; Chen et al., 2022b; Allshire et al., 2021) that use Isaac Gym for simulation, (2) **DDPG(n)**: DDPG (Lillicrap et al., 2015) implementation with double Q learning and n-step returns, (3) **SAC(n)**: SAC (Haarnoja et al., 2018) implementation with n-step returns.

Hardware We use NVIDIA GeForce RTX 3090 GPUs as our default GPUs for the experiments unless otherwise specified. More details are shown in Table B.3 in the appendix.

4.2. PQL learns faster than baselines

The first and most important question to answer is whether PQL leads to faster learning than SOTA baselines. To answer this, we compared the learning curves of PQL and PQL-D (PQL with distributional RL) with baselines on six benchmark tasks. As shown in Figure 3, our method (PQL, PQL-D) achieves the fastest policy learning in five out of six tasks compared to all baselines. Moreover, we observed that adding distributional RL to PQL can further boost learning speed. Figure 3 shows that in five out of six tasks, PQL-D achieves wall-clock time faster than, or at least on par with, PQL. The improvements are most salient on the two challenging contact-rich manipulation tasks (*Shadow Hand* and *Allegro Hand*). Additionally, the faster learning of PQL than DDPG(n) demonstrates the advantage of using a parallel scheme for data collection and network updates. We also found that DDPG(n) outperforms SAC(n) in all tasks. This could be due to the fact that the exploration scheme in DDPG can scale up better than the one in SAC. In DDPG, we apply the same mixed exploration as in PQL, while the exploration of SAC solely comes from sampling in the stochastic policy distribution, which can be heavily affected by the quality of the policy distribution.

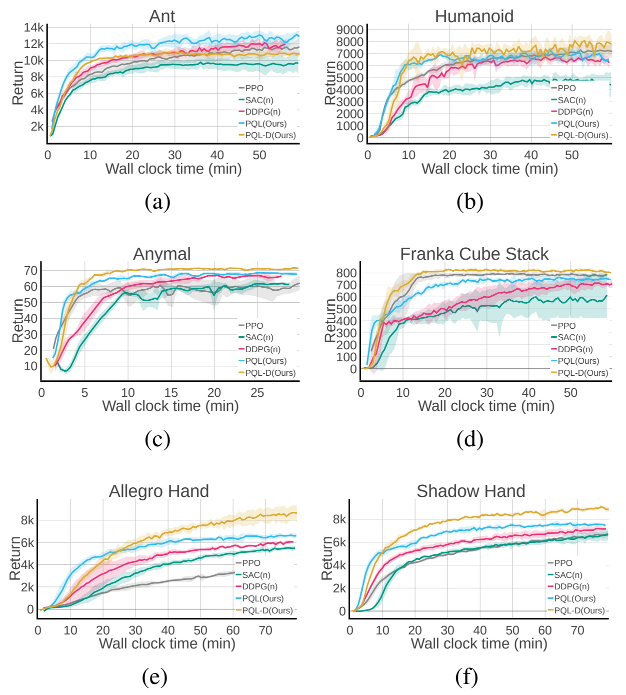


Figure 3. We compare our methods to the SOTA RL algorithms (PPO, SAC with n-step returns, DDPG with n-step returns). We use 4096 environments for training in all tasks except the PPO baseline on *Shadow Hand* and *Allegro Hand* tasks, where we use 16384 as it gives the best performance for PPO on these two tasks as shown in Figure 5(c). Our methods achieve the fastest learning speed in almost all tasks.

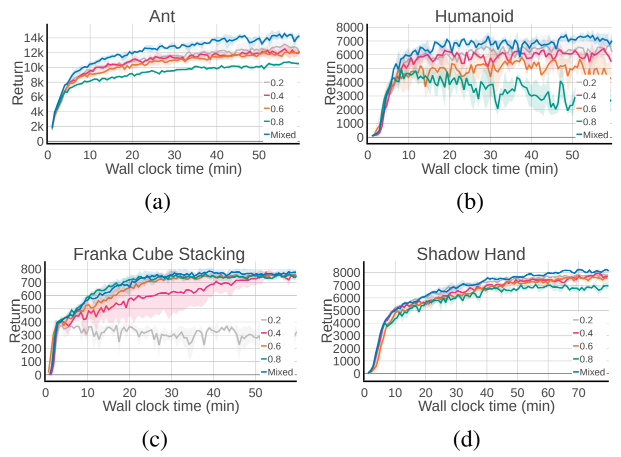


Figure 4. We compared our proposed mixed exploration scheme by applying different constant maximum noise values. We can see that the mixed exploration scheme either outperforms or is on par with other schemes, which can save the tuning effort on the noise level.

4.3. How well does mixed exploration perform?

As discussed in Section 3.3, massively parallel simulation enables us to deploy different exploration strategies in different environments to generate more diverse exploration trajectories. We use a simple mixed exploration strategy, as described in Section 3.3, and compare its effectiveness to cases where all the environments use the same exploration capacity (the same σ values). We experimented with $\sigma \in \{0.2, 0.4, 0.6, 0.8\}$. As shown in Figure 4, the learning performance is significantly affected by the choice of σ value. If we use the same σ value for all parallel environments, then we need to tune σ for each task. In contrast, the mixed exploration strategy, where each environment uses a different σ value, outperforms (learns faster or at least as fast as) all other fixed σ values. This implies that using the mixed exploration strategy can reduce the tuning effort needed for σ values per task.

4.4. Effects of different hyper-parameters

In this section, we investigate the effects of the number of environments, $\beta_{p:v}$, $\beta_{a:v}$, batch size, replay buffer size, and the number of GPUs. These hyper-parameters are of particular interest given the massively parallel simulation ($N >> 1000$) and our parallel scheme. Lastly, GPU hardware can also impact learning speed. To explore this, we conduct experiments using four different GPU models and analyze the effect of GPU hardware on performance in Appendix C.

4.4.1. How does the number of environments N affect policy learning?

Previous works on distributed frameworks for RL (Horgan et al., 2018; Espeholt et al., 2018) have shown how the learning performance is affected by the number of parallel environments N, with N in the order of hundreds. GPU simulation enables running thousands of environments in parallel on a single workstation, and we anticipate that this will only improve with time. However, more parallel environments will only be beneficial if RL algorithms can exploit such data, i.e. if performance scales with more data. We, therefore, investigated how different algorithms scale with the number of environments ($N >> 1000$, the biggest N we experimented with is $16,384$). As shown in Figure 5, both PQL and PPO benefit from using more environments in parallel. Moreover, the learning performance of PQL is relatively less sensitive to N on the simple task (*Ant*), while on the hard task (*Shadow Hand*), PPO's learning performance substantially drops as we decrease the number of environments. In contrast, our method (PQL) demonstrates stable and similar learning with all the different numbers of environments except when N is very small ($N = 256$) on *Shadow Hand*, suggesting that PQL is more robust to changes in N.

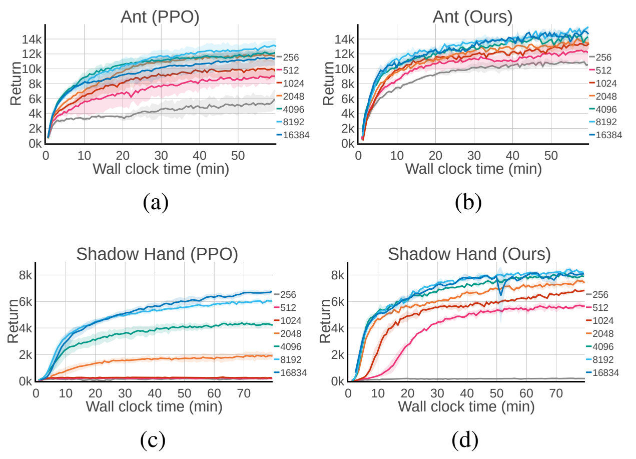


Figure 5. We sweep over different numbers of environments (N) on both PPO and PQL (our method). Overall, PQL is less sensitive to the number of environments than PPO on both tasks.

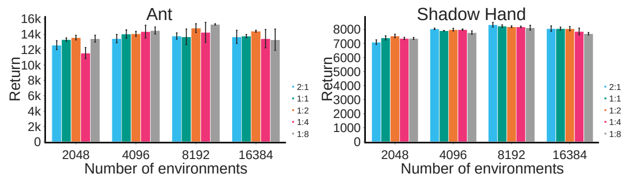


Figure 6. We show the averaged returns in evaluation after a fixed amount of training time ΔT. Across the set of different numbers of environments we experimented with $(2048, 4096, 8192, 16384)$, we found that setting $\beta_{p:v} = 1:2$ generally works well. $\Delta T = 60$ mins for *Ant*, and $\Delta T = 80$ mins for *Shadow Hand*. The complete learning curves are in Figure C.6.

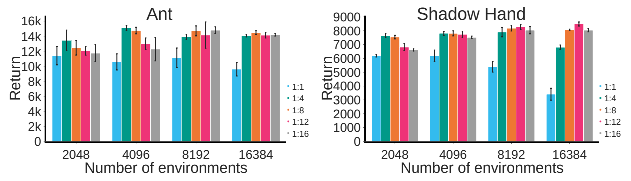


Figure 7. Given different values of N, we show the effect of different $\beta_{a:v}$. An overall trend we observe is that as N gets bigger, it's more beneficial to update the critic more frequently. We also found that $\beta_{a:v} = 1:8$ generally works well given different N values. So one can set $\beta_{a:v} = 1:8$ as a good initial value, and tune it if necessary on new tasks.

4.4.2. Effect of $\beta_{p:v}$ and $\beta_{a:v}$

As discussed in Section 3.2, explicitly controlling the $\beta_{a:v}$ and $\beta_{p:v}$ can help improve the learning performance and reduce the variance under different training conditions, such as fluctuated hardware utilization. If $\beta_{p:v}$ is larger, the policy updates more frequently than the value functions, potentially leading to policy overfitting to the stale value function, which in turn leads to poor exploration. If the policy updates much slower than the value function, the policy might lag behind the value function a lot, which hurts the learning speed. Similarly, if $\beta_{a:v}$ is larger, the **V-learner** might need to wait for **Actor** to collect enough data, since the simulation speed cannot be changed, leading to slower learning. If $\beta_{a:v}$ is smaller, the value function updates more given the generated rollout data.

To qualitatively assess the effects of different $\beta_{a:v}$ and $\beta_{p:v}$, we sweep over a range of values for these two hyper-parameters and compare them in Figure 6 and Figure 7. Figure 6 shows that PQL is relatively robust to a wide range of $\beta_{p:v}$ values, which means this hyper-parameter would require little tuning. We use $\beta_{p:v} = 1:2$ as the default value in our experiments shown in the paper. This ratio value is consistent with prior works (Fujimoto et al., 2018; Yang et al., 2022), but our findings are in the context of parallel training of policy and value function. Figure 7 shows that $\beta_{a:v}$ has a greater impact on the learning performance. An overall trend is that if we increase the number of environments, then we need to have **V-learner** update the Q functions more times. For example, on *Shadow Hand*, $\beta_{a:v} = 1:4$ performs the best when $N = 2048$ and $N = 4096$. But when $N = 8192$ and $N = 16384$, $\beta_{a:v} = 1:12$ performs the best. We use $\beta_{a:v} = 1:8$ by default as it achieves a good performance across different N values. In summary, Figure 6 and Figure 7 show that $\beta_{p:v}$ and $\beta_{a:v}$ do affect the performance with a varied number of environments. We suggest setting $\beta_{p:v} = 1:2, \beta_{a:v} = 1:8$ as a good starting point for new tasks and tune them if necessary, as these are the values we found work well on six different tasks with different numbers of environments. In addition, in Section C, we show that adding the speed ratio control ($\beta_{a:v}$ and $\beta_{p:v}$) is beneficial for balancing the computing resources used by each process when resources are limited.

4.4.3. Effect of batch size

With many parallel environments (N), a significant amount of data is generated quickly. While it is easy to increase N from hundreds to tens of thousands in Isaac Gym on a single GPU, it is infeasible to increase the replay buffer size by 100 times due to limited GPU memory or CPU RAM (if the data is stored on the CPU). Consequently, the replay buffer is frequently overwritten, meaning that each collected sample may not be used efficiently. One way to efficiently utilize large amounts of changing data is to increase the batch size. To determine how much increase in batch size is necessary for Q-learning with a limited-capacity replay buffer to take advantage of the large amounts of incoming data, we investigated the relationship between performance and batch size. Many prior works have shown that using a large batch size can improve network performance, such as in contrastive learning settings (Grill et al., 2020; Chen

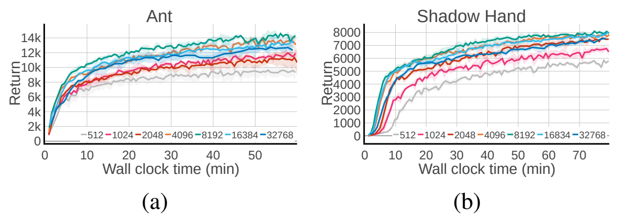


Figure 8. Effect of different batch sizes. Small batch size usually leads to slower learning. If the batch size is too big, the policy learning can get slowed down because GPUs have a limited amount of cores and it takes more time to process a very big batch of data.

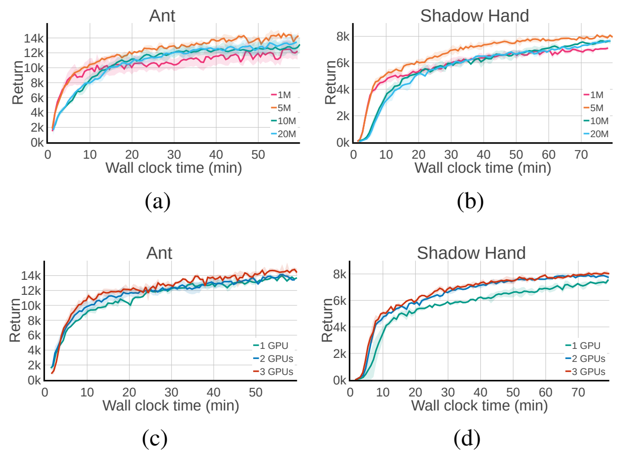


Figure 9. **(a)** and **(b)**: effect of different replay buffer size. **(c)** and **(d)**: effect of number of GPUs used for running PQL. PQL can be deployed on a flexible number of GPUs. In complex tasks such as *Shadow Hand*, it is beneficial to have at least 2 GPUs where the **Actor** runs on a separate GPU as the simulation itself consumes more GPU compute as the task complexity increases.

et al., 2020). In our work, we found that large-batch training can notably improve the learning speed in off-policy RL for massively parallel simulations, as shown in Figure 10. However, if the batch size is too big, the learning speed can be slowed down. This is because GPUs have a limited number of CUDA cores, and it takes more time to process a very big batch of data once the batch size is above some threshold value, which is another underlying trade-off.

4.4.4. Effect of Replay buffer size

As discussed in Section 1, when dealing with tens of thousands of parallel environments, the replay buffer with normal capacity (e.g. 1M) gets a full refresh for every several hundreds of environment steps. This means that the replay buffer will not contain too much historical data. Apriori, one might think off-policy methods will fail in this case and that we need to proportionally increase the replay buffer size to store more experience data. However, surprisingly, we empirically found that even with thousands of parallel environments, having a ”small” replay buffer (1M or 5M) can still lead to good performance. In Figure 9, we show how the learning curves change as we vary the buffer capacity. We can see that, in all cases, the policies learn well. We hypothesize that PQL still works well in this case because a large number of parallel environments can generate diverse enough data in a few environment steps. Moreover, $|B| \in \{1, 5\}$M leads to faster policy learning at the beginning of the training than $|B| = \{10, 20\}$M. We hypothesize that this is because a smaller replay buffer allows the old and less informative samples to be replaced much faster, which is more important in the early stages of training.

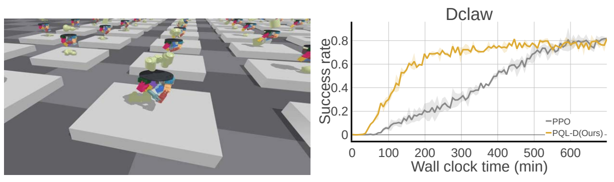
4.4.5. Number of GPUs

Nowadays, it is common to have workstations with multiple GPUs. Running different processes on separate GPUs can potentially speed up learning. Our PQL scheme can adapt to different numbers of GPUs available on a workstation. Specifically, **Actor**, **P-learner**, and **V-learner** can be placed on any GPU. To investigate the performance variation when we distribute the three components across different numbers of GPUs, we conducted experiments with three scenarios on Tesla A100 GPUs: (1) place all three processes on the same GPU, (2) place the **Actor** on one GPU, **P-learner** and **V-learner** on another GPU, (3) place the **Actor**, **P-learner**, **V-learner** on a different GPU respectively. In the two-GPU case, we allocate **Actor** to a dedicated GPU because simulating many tasks with a large number of environments can cause high GPU utilization. As shown in Figure 9, our PQL scheme works well in all three scenarios with one, two, or three GPUs. When the task becomes more complex like *Shadow Hand*, the simulation takes much more computation and time. Putting all three processes will slow down each one of them due to full GPU utilization, which is why we see a bigger gap between the 2-GPU or 3-GPU training and 1-GPU training on *Shadow Hand*. Therefore, it is beneficial to place the **Actor** on one GPU and **P-learner** and **V-learner** on other GPUs.

4.5. Additional Tasks

Vision-based *Ball Balancing* task Simulating vision-based tasks is much slower and more demanding on the GPU as each simulation step involves both the physics simulation and image rendering. To demonstrate the generality of our scheme in operating in this practical setting of vision-based training, we consider a vision-based *Ball Balancing* task (Makoviychuk et al., 2021).

Since directly learning a vision-based policy with RL is time-consuming, we use the idea of asymmetric actor-critic learning (Pinto et al., 2017) to speed up vision policy learning. Image data is compressed using the *lz4* library to reduce the bandwidth requirement and communication overhead. More setup details are in Appendix B.3. As shown in Fig-



(a) (b)

Figure 10. **(a)**: *DClaw Hand* task. **(b)**: We compared our proposed PQL-D method with PPO.

ure B.1, PQL achieves better sample efficiency and higher final performance than PPO with $N = 1024$ parallel environments.

Reorient hundreds of objects with a *DClaw Hand* We conducted further experiments on a contact-rich dexterous manipulation task (Chen et al., 2022a), *DClaw Hand*, as shown in Figure 10(a). This task is much more challenging than the *Shadow Hand* task and *Allegro Hand* task because it needs to learn to reorient hundreds of different objects with a single policy. Furthermore, the control frequency (12Hz) is much lower than the default control frequency (60Hz) used in all six proposed benchmark tasks. This means that the simulation takes much longer to run between each policy command step, and the **Actor** process will be much slower. In Figure 10(b), we observe that our method reaches 70% success rate around 200 minutes, which is approximately 3 times faster than PPO.

5. Discussion and Future Work

We present a scheme **PQL** for scaling up off-policy methods with tens of thousands of parallel environments on a single workstation. Our method achieves state-of-the-art results on the Isaac Gym benchmark tasks in terms of the training wall clock time. The driving force behind this success is the parallelization of data collection, policy function learning, and value function learning. We provide a mechanism to balance and control the speed in different processes, which leads to better and more stable performance across different hardware conditions or when the GPU resource is limited. Although PPO requires a large number of environments to work on complex tasks such as *Shadow Hand*, PQL is more lenient on the number of environments and works well on a wide range of different numbers of environments. With a large number of parallel environments, it is beneficial to use a big batch size for training agents, with the caveat that if the batch size is too big, it might take the GPU more time to process the batch data and lead to a slowdown in policy learning. We also found using different exploration scales in different environments achieves better or similar performance compared to a carefully-tuned exploration scale in all parallel environments, which means we need less hyper-parameter tuning. Even though the number of environments is 1000× more, we did not find it necessary to use a replay buffer that is 1000× bigger. In fact, a replay buffer with a capacity of 5M transitions is sufficient for our experiments even with 16843 parallel environments. Our scheme's hardware requirements are flexible and work well with different numbers of GPUs and various GPU models.

For future work, it would be interesting to investigate the sampling strategy for the replay buffer. In PQL, we do not use techniques such as prioritized experience replay (Schaul et al., 2015). The sampling process can take a very long time due to the massive amount of collected data, thus such techniques could improve the sample efficiency but significantly hurt wall-clock time efficiency. Therefore, new strategies should be considered, e.g., one can consider rejecting samples given the massive amount of data. It would also be practically interesting to study different exploration strategies that can take advantage of parallel environments. Lastly, one may notice that our scheme can be easily extended to a system with multiple parallel **P-learner** or **V-learner** given the decoupling of policy and value learning. In this case, another interesting direction is to apply ensemble methods or evolutionary strategies to further exploit the massive amount of data.

References

Achiam, J. Spinning Up in Deep Reinforcement Learning. 2018.

Allshire, A., Mittal, M., Lodaya, V., Makoviychuk, V., Makoviichuk, D., Widmaier, F., Wüthrich, M., Bauer, S., Handa, A., and Garg, A. Transferring dexterous manipulation from gpu simulation to a remote real-world trifinger. *arXiv preprint arXiv:2108.09779*, 2021.

Babaeizadeh, M., Frosio, I., Tyree, S., Clemons, J., and Kautz, J. Reinforcement learning through asynchronous advantage actor-critic on a gpu. *arXiv preprint arXiv:1611.06256*, 2016.

Bellemare, M. G., Dabney, W., and Munos, R. A distributional perspective on reinforcement learning. In *International Conference on Machine Learning*, pp. 449–458. PMLR, 2017.

Chen, T., Kornblith, S., Norouzi, M., and Hinton, G. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*, pp. 1597–1607. PMLR, 2020.

Chen, T., Tippur, M., Wu, S., Kumar, V., Adelson, E., and Agrawal, P. Visual dexterity: In-hand dexterous manipulation from depth. *arXiv preprint arXiv:2211.11744*, 2022a.

Chen, T., Xu, J., and Agrawal, P. A system for general in-hand object re-orientation. In *Conference on Robot Learning*, pp. 297–307. PMLR, 2022b.

Clemente, A. V., Castejón, H. N., and Chandra, A. Efficient parallel methods for deep reinforcement learning. *arXiv preprint arXiv:1705.04862*, 2017.

Coumans, E. and Bai, Y. Pybullet, a python module for physics simulation for games, robotics and machine learning. 2016.

Espeholt, L., Soyer, H., Munos, R., Simonyan, K., Mnih, V., Ward, T., Doron, Y., Firoiu, V., Harley, T., Dunning, I., et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *International conference on machine learning*, pp. 1407–1416. PMLR, 2018.

Espeholt, L., Marinier, R., Stanczyk, P., Wang, K., and Michalski, M. Seed rl: Scalable and efficient deep-rl with accelerated central inference. *arXiv preprint arXiv:1910.06591*, 2019.

Fujimoto, S., Hoof, H., and Meger, D. Addressing function approximation error in actor-critic methods. In *International conference on machine learning*, pp. 1587–1596. PMLR, 2018.

Fujita, Y., Nagarajan, P., Kataoka, T., and Ishikawa, T. 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.

Grill, J.-B., Strub, F., Altché, F., Tallec, C., Richemond, P., Buchatskaya, E., Doersch, C., Avila Pires, B., Guo, Z., Gheshlaghi Azar, M., et al. Bootstrap your own latent-a new approach to self-supervised learning. *Advances in neural information processing systems*, 33:21271–21284, 2020.

Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pp. 1861–1870. PMLR, 2018.

Hasselt, H. Double q-learning. *Advances in neural information processing systems*, 23, 2010.

Heess, N., TB, D., Sriram, S., Lemmon, J., Merel, J., Wayne, G., Tassa, Y., Erez, T., Wang, Z., Eslami, S., et al. Emergence of locomotion behaviours in rich environments. *arXiv preprint arXiv:1707.02286*, 2017.

Horgan, D., Quan, J., Budden, D., Barth-Maron, G., Hessel, M., Van Hasselt, H., and Silver, D. Distributed prioritized experience replay. *arXiv preprint arXiv:1803.00933*, 2018.

Hwangbo, J., Lee, J., Dosovitskiy, A., Bellicoso, D., Tsounis, V., Koltun, V., and Hutter, M. Learning agile and dynamic motor skills for legged robots. *Science Robotics*, 4(26):eaau5872, 2019.

Kapturowski, S., Ostrovski, G., Quan, J., Munos, R., and Dabney, W. Recurrent experience replay in distributed reinforcement learning. In *International conference on learning representations*, 2018.

Kober, J., Bagnell, J. A., and Peters, J. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

Makoviichuk, D. and Makoviychuk, V. rl-games: A high-performance framework for reinforcement learning. https://github.com/Denys88/rl_games, May 2022.

Makoviychuk, V., Wawrzyniak, L., Guo, Y., Lu, M., Storey, K., Macklin, M., Hoeller, D., Rudin, N., Allshire, A., Handa, A., et al. Isaac gym: High performance gpu-based physics simulation for robot learning. *arXiv preprint arXiv:2108.10470*, 2021.

Margolis, G. B., Yang, G., Paigwar, K., Chen, T., and Agrawal, P. Rapid locomotion via reinforcement learning. *arXiv preprint arXiv:2205.02824*, 2022.

Miki, T., Lee, J., Hwangbo, J., Wellhausen, L., Koltun, V., and Hutter, M. Learning robust perceptive locomotion for quadrupedal robots in the wild. *Science Robotics*, 7 (62):eabk2822, 2022.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. Human-level control through deep reinforcement learning. *nature*, 518(7540): 529–533, 2015.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pp. 1928–1937. PMLR, 2016.

Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Paul, W., Jordan, M. I., and Stoica, I.

Ray: a distributed framework for emerging ai applications. corr abs/1712.05889 (2017). *arXiv preprint arXiv:1712.05889*, 2017.

Nair, A., Srinivasan, P., Blackwell, S., Alcicek, C., Fearon, R., De Maria, A., Panneershelvam, V., Suleyman, M., Beattie, C., Petersen, S., et al. Massively parallel methods for deep reinforcement learning. *arXiv preprint arXiv:1507.04296*, 2015.

OpenAI, Andrychowicz, M., Baker, B., Chociej, M., Jozefowicz, R., McGrew, B., Pachocki, J., Petron, A., Plappert, M., Powell, G., Ray, A., et al. Learning dexterous in-hand manipulation. *The International Journal of Robotics Research*, 39(1):3–20, 2020.

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

Popov, I., Heess, N., Lillicrap, T., Hafner, R., Barth-Maron, G., Vecerik, M., Lampe, T., Tassa, Y., Erez, T., and Riedmiller, M. Data-efficient deep reinforcement learning for dexterous manipulation. *arXiv preprint arXiv:1704.03073*, 2017.

Popova, M., Isayev, O., and Tropsha, A. Deep reinforcement learning for de novo drug design. *Science advances*, 4(7): eaap7885, 2018.

Rudin, N., Hoeller, D., Reist, P., and Hutter, M. Learning to walk in minutes using massively parallel deep reinforcement learning. In *Conference on Robot Learning*, pp. 91–100. PMLR, 2022.

Schaul, T., Quan, J., Antonoglou, I., and Silver, D. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

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

Sutton, R. S. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

Todorov, E., Erez, T., and Tassa, Y. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ international conference on intelligent robots and systems*, pp. 5026–5033. IEEE, 2012.

Wijmans, E., Kadian, A., Morcos, A., Lee, S., Essa, I., Parikh, D., Savva, M., and Batra, D. Dd-ppo: Learning near-perfect pointgoal navigators from 2.5 billion frames. *arXiv preprint arXiv:1911.00357*, 2019.

Yang, G., Ajay, A., and Agrawal, P. Overcoming the spectral bias of neural value approximation. *arXiv preprint arXiv:2206.04672*, 2022.

A. Pseudo Code

Algorithm 1 Actor Process (main process)

```
for $n = 1 : W_a$ do
    $\pi \leftarrow$ policy network from P-learner process
    Initialize an empty buffer $B = \phi$
    for $t = 1 : H$ do
        $\boldsymbol{a}_t \leftarrow \pi(\boldsymbol{s}_t)$ with mixed exploration noise
        $(\boldsymbol{r}_t, \boldsymbol{s}_{t+1}) \leftarrow$ envs.step($\boldsymbol{a}_t$)
        $B = B \cup \{\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{r}_t, \boldsymbol{s}_{t+1}\}$
    end for
    $Q_1, Q_2 \leftarrow$ Q functions from V-learner process
    send $B, \pi$ to V-learner, send $\{s_t\}$ in $B$, $Q_1, Q_2$ to P-learner
    sleep for $t_a$ seconds to satisfy $\beta_{a:v}$
end for
```

Algorithm 2 P-learner Process

```
Initialize an empty buffer $B_p = \phi$
for $n = 1 : W_p$ do
    if new data received then
        $\{s_t\} \leftarrow$ from Actor process
        $Q_1, Q_2 \leftarrow$ from Actor process
        $B = B \cup \{s_t\}$
    end if
    sample a batch of $\{s_t\}$
    update $\pi$ by maximizing the $\min_{i=1,2} Q_i(s_t, \pi(s_t))$
    sleep for $t_p$ seconds to satisfy $\beta_{p:v}$
end for
```

Algorithm 3 V-learner Process

```
Initialize an empty buffer $B_v = \phi$
for $n = 1 : W_v$ do
    if new data received then
        $\{s_t, a_t, r_t, s_{t+1}\} \leftarrow$ from Actor process
        $\pi \leftarrow$ from Actor process
        $Q_1, Q_2 \leftarrow$ from Actor process
        $B = B \cup \{s_t\}$
    end if
    sample a batch of $\{s_t, a_t, r_t, s_{t+1}\}$
    update $Q_1, Q_2$ by minimizing the mean-squared Bellman error (with Double Q-learning)
    sleep for $t_v$ seconds to satisfy $\beta_{p:v}, \beta_{a:v}$
end for
```

B. Training setups

B.1. Hyper-parameters

We use the hyper-parameter values shown in Table B.1 and the reward scaling shown in Table B.2 for all the experiments unless otherwise specified. As for PPO, we use the same hyperparameter setup in Makoviychuk et al. (2021).

Table B.1. Hyper-parameter setup for six Isaac Gym benchmark tasks

Hyper-parameter	PQL(ours)	DDPG	SAC
Num. Environments	4,096	4,096	4,096
Critic Learning Rate	5×10^{-4}	5×10^{-4}	5×10^{-4}
Actor Learning Rate	5×10^{-4}	5×10^{-4}	5×10^{-4}
Learnable Entropy Coefficient	-	-	True
Optimizer	Adam	Adam	Adam
Target Update Rate (τ)	5×10^{-2}	5×10^{-2}	5×10^{-2}
Batch Size	8,192	8,192	8,192
Num. Epochs ($\beta_{a:v}$)	8	8	8
Discount Factor(γ)	0.99	0.99	0.99
Normalized Observations	True	True	True
Gradient Clipping	0.5	0.5	0.5
Exploration Policy	Mix	Mix	-
N-step target	3	3	3
Warm-up Steps	32	32	32
Replay Buffer Size	5×10^{6}	5×10^{6}	5×10^{6}

Table B.2. Reward scale

	Reward scale
Ant	0.01
Humanoid	0.01
ANYmal	1.0
Franka Cube Stacking	0.1
Allegro Hand	0.01
Shadow Hand	0.01
Ball Balance	0.1
DClaw Hand	0.01

B.2. Hardware Configurations

Table B.3 lists the hardware configurations of the workstations we used for the experiments. We use the machines with GeForce RTX 3090 for experiments by default. We also measure how much time it takes for the simulator to generate 1M interaction data with 4096 parallel environments on *Ant* and *Shadow Hand*. We generate 1M data via the following command.

```
for i in range(244):
    action = torch.randn((4096,
                          envs.action_space.shape[0]),
                          device='cuda')
    out = envs.step(action)
```

B.3. Vision experiment setup

We render the RGB camera image in a resolution of 48×48. The CNN part of our vision network $g(o_t)$ is as follows:

`Conv(3,32,3,2)-BN(32)-ReLU-3x(Conv(32,32,3,2)-BN(32)-ReLU)`

where `Conv(a,b,k,s)` is a Convolutional layer with input channels a, output channels b, kernel size k, stride s.

Since our policy input contains a history of observations (o_{t-2}, o_{t-1}, o_t), we use the same CNN to extract the feature of each observation and then concatenate all the embeddings. Then, the concatenated embedding goes through an MLP network h:

```
    FC(256)-ReLU-FC(63)-ReLU-FC(3)
```

Table B.3. Hareware configurations on different workstations

		Workstation 1	Workstation 2	Workstation 3	Workstation 4
CPU		AMD Threadripper 3990X	Intel Xeon Gold 6248	AMD Rome 7742	Intel Xeon W-2195
GPU		GeForce RTX 3090	Tesla V100	Tesla A100	GeForce RTX 2080 Ti
GPU CUDA Cores		10496	5120	6912	4352
GPU FP32 TFLOPs		35.58	16.4	19.5	13.45
Time for generating	Ant	1.678 ± 0.006	2.117 ± 0.038	1.999 ± 0.004	3.397 ± 0.014
1M data ($N = 4096$) (s)	Shadow Hand	6.706 ± 0.028	9.051 ± 0.035	8.653 ± 0.101	10.885 ± 0.025

In summary, at each time step t, the policy output is $h[\text{cat}(g(o_{t-2}), g(o_{t-1}), g(o_t))]$. Storing images in a replay buffer can take up a lot of memory. Therefore, we experiment with different placements of the replay buffer: (1) put the replay buffer on a GPU with a big memory, (2) put the replay buffer on CPU RAM. We use the same A100 GPUs for all these image-based experiments. Figure B.1 shows that our method (PQL) works with either the replay buffer on the GPU or CPU, and it achieves much faster learning and better performance than PPO.

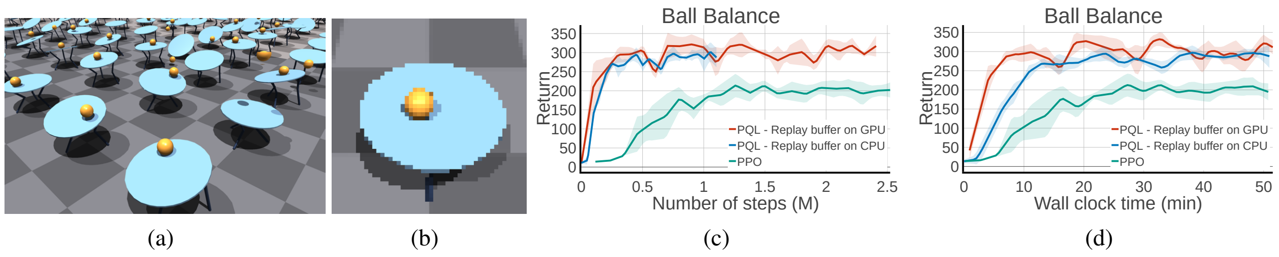


Figure B.1. **(a)**: the *Ball Balancing* task in Isaac Gym. **(b)**: the rendered RGB image from the simulated camera. **(c)** shows the learning curves regarding the number of environment steps. **(d)** shows the training wall-clock time. We can see that PQL achieves both better sample efficiency and higher final performance than PPO.

Table B.4. Hyper-parameter setup for the *Ball Balancing* task.

Hyper-parameter	PQL(ours)	PPO
Num. Environments	1,024	1,024
Critic Learning Rate	5×10^{-4}	5×10^{-4}
Actor Learning Rate	5×10^{-4}	5×10^{-4}
Optimizer	Adam	Adam
Target Update Rate (τ)	5×10^{-2}	-
Batch Size	4,096	4,096
Horizon length	1	16
Num. Epochs	12	5
Discount Factor(γ)	0.99	0.99
Normalized Observations	True	True
Gradient Clipping	True	True
Exploration Policy	Mix	-
N-step target	3	-
Warm-up Steps	32	-
Replay Buffer Size	10^6	-
Clip Ratio	-	0.2
GAE	-	True
λ	-	0.95

C. Additional Experiments

n-step returns We investigate how much does n-step returns help for PQL. As shown in Figure C.3, adding n-step return leads to faster learning than not using n-step return ($n = 1$). However, using a big n value hurt the learning. Empirically we found that $n = 3$ gives us the best performance.

Benefit of adding speed control ($\beta_{p:v}, \beta_{a:v}$) on different processes As we mentioned in Section 3.2, adding speed control using $\beta_{p:v}, \beta a : v$ can help reduce the variance of training when the amount of computation resources changes. To provide more insights, we ran experiments without speed control, i.e., each process could run as fast as possible without any waiting. As shown in Figure C.2, when there are sufficient compute resources available (with two GPUs), the benefit of having the speed ratio control is not significant. However, when only one GPU is available for running all three processes (**Actor**, **P-learner**, **V-learner**), we can see that without the ratio control, the learning curves on all six benchmark tasks slow down. We believe this is because all three processes are trying to run as fast as possible, resulting in competition for GPU utilization, which slows overall learning. Adding the ratio control helps balance GPU resource utilization among the three processes. Thus, even with one GPU, the learning performance with ratio control is quite similar to that with two GPUs.

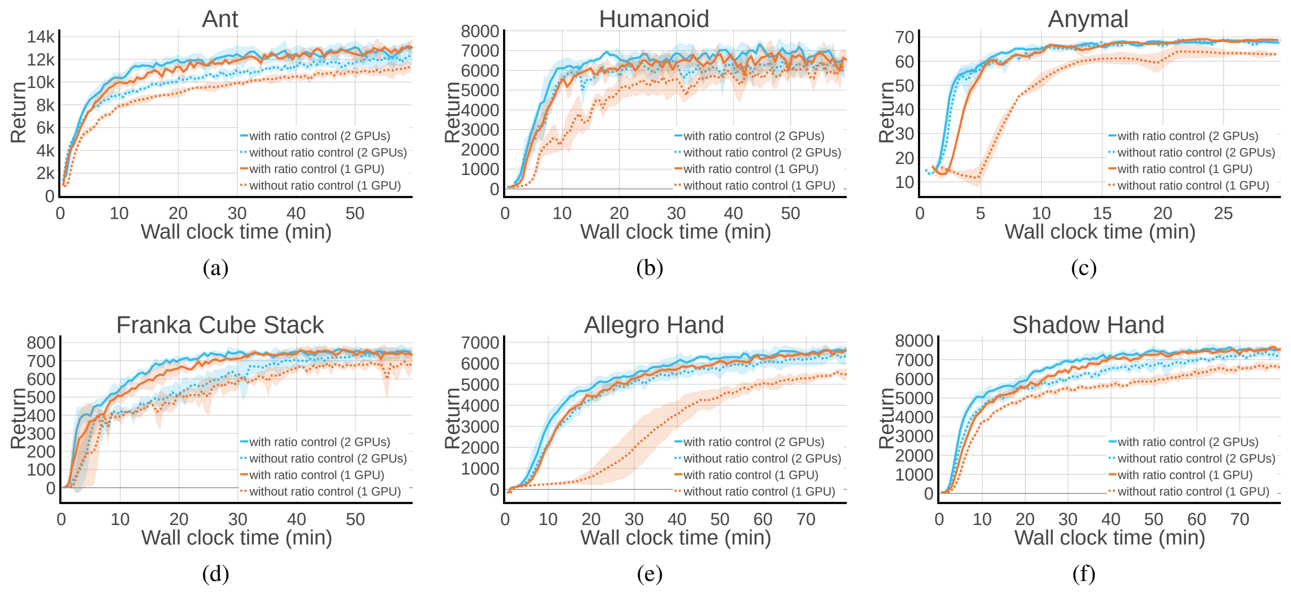
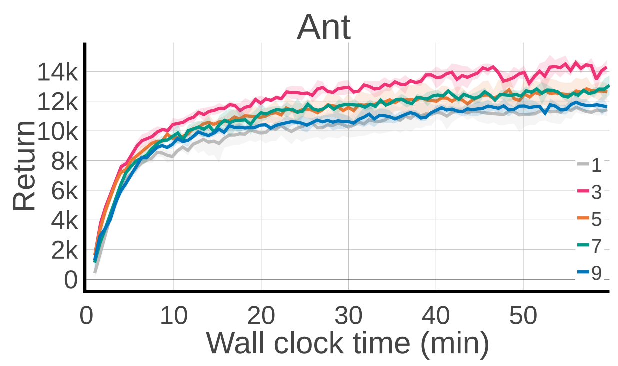


Figure C.2. Comparison of the learning performance with and without the speed ratio control ($\beta_{p:v}, \beta_{a:v}$) on two RTX3090 GPUs and on 1 RTX3090 GPU, respectively.

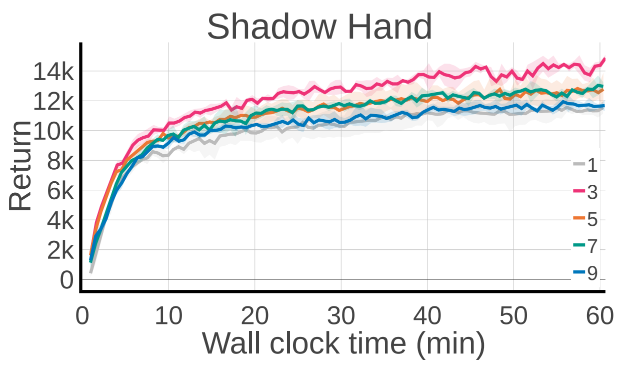
GPU hardware The simulation speed and network training speed vary across different GPU models. In Table B.3, we list how much time it takes for the simulator to generate 1M environment interaction data with 4096 parallel environments on four machines with different GPU models. In our test, the simulation speed on different GPU models is as follows: GeForce 3090 > Tesla A100 > Tesla V100 > GeForce 2080Ti. We test PQL performance on all these four different machine configurations (Table B.3). We can see that different GPU models affect the policy learning speed, especially on complex tasks like *Shadow Hand* which takes more simulation time.

PQL for SAC As discussed above, PQL framework is flexible and can be combined with different Q-learning methods. Here, we show that PQL can be combined with SAC as well. Figure C.4 shows that adding the PQL framework to SAC substantially speeds up the learning speed of SAC.

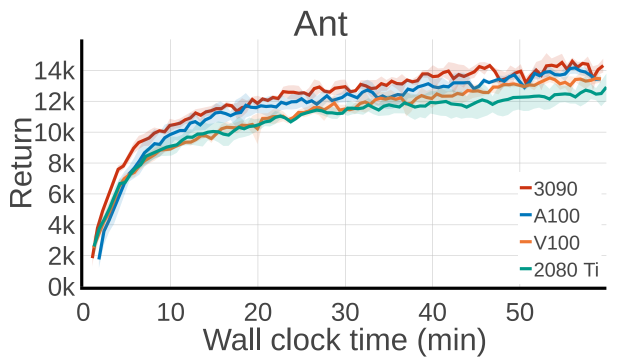
Sample efficiency compared to baselines Figure C.5 shows the sample efficiency of each algorithm on different environments. Overall, we see that PQL achieves the best sample efficiency. In addition, DDPG(n) also outperforms SAC(n) in terms of sample efficiency on these tasks.



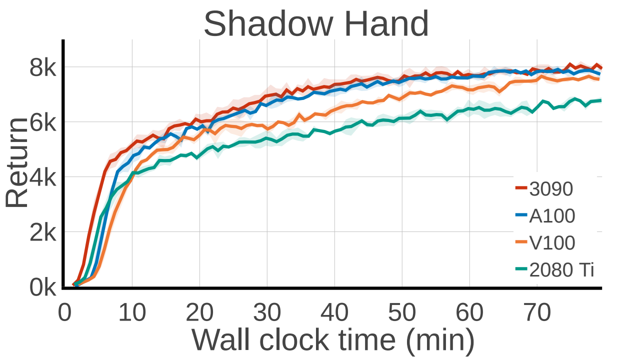
(a)



(b)

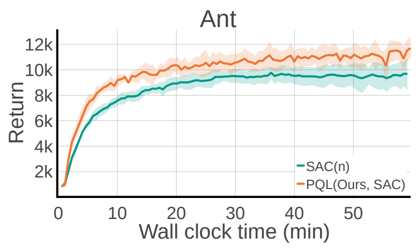


(c)

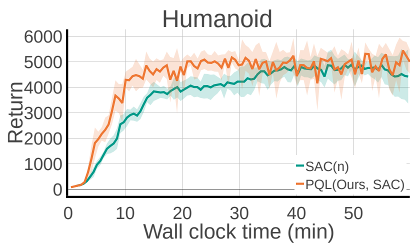


(d)

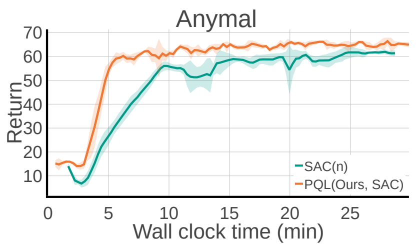
Figure C.3. **(a)** and **(b)**: effect of n-step return. $n = 3$ performs the best. **(c)** and **(d)**: effect of GPU models used for running PQL. Overall, we see that PQL works robustly across different GPU models, and running on newer GPUs tends to give a faster learning.



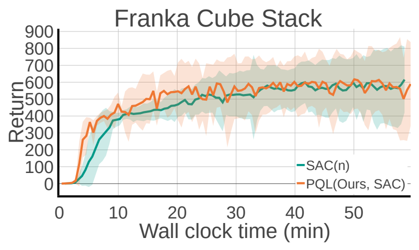
(a)



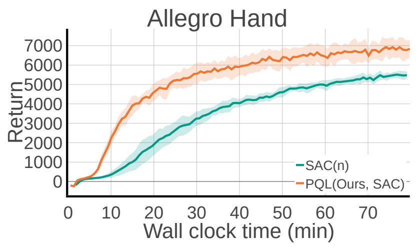
(b)



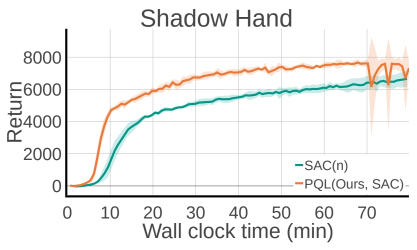
(c)



(d)

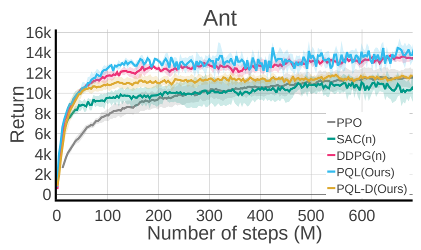


(e)

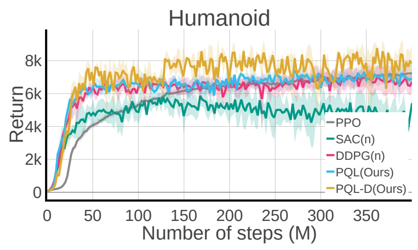


(f)

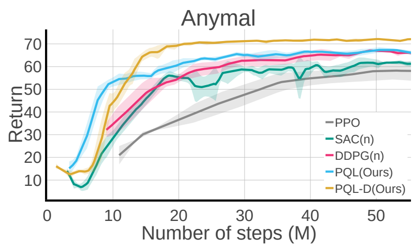
Figure C.4. We apply our parallel Q-learning to SAC. PQL + SAC achieves faster learning than SAC itself.



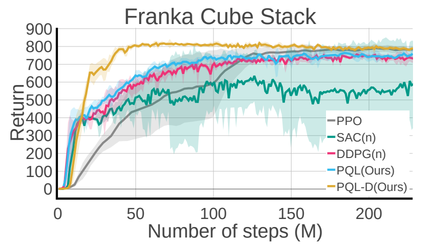
(a)



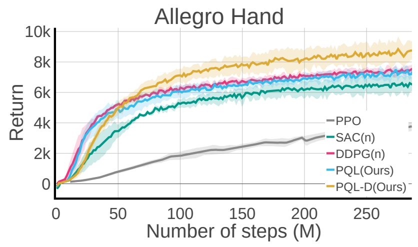
(b)



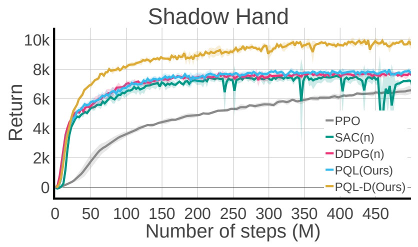
(c)



(d)



(e)



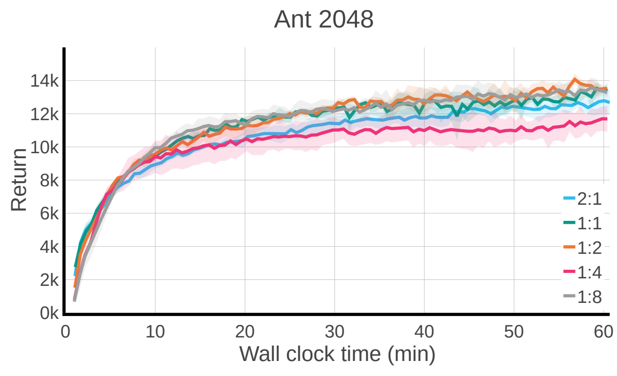
(f)

Figure C.5. Similar to Figure 3, we show that our method (PQL) also achieves better sample efficiency than baselines.

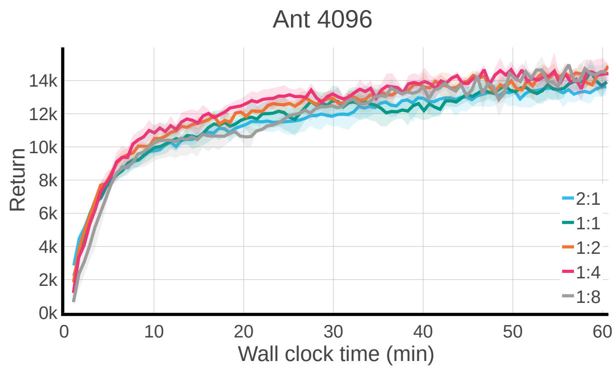
Sweep over different $\beta_{a:v}$ and $\beta_{p:v}$ Figure C.6 shows the complete learning curves with different $\beta_{p:v}$ values and different number of environments. Similarly, Figure C.7 shows the learning curves for different $\beta_{a:v}$.

Comparison of our implementation with RL-games In this work, we implemented all the algorithms (PQL and all the baselines) from scratch, as it gives us the most flexibility in exploring different design choices that can affect learning performance. To show that our codebase provides good performance, we compare it against the most commonly used RL codebase used for Isaac Gym, which is RL-games (Makoviichuk & Makoviychuk, 2022). However, RL-games only support PPO and SAC. Hence, we compare our implementations of PPO and SAC against the ones in RL-games.

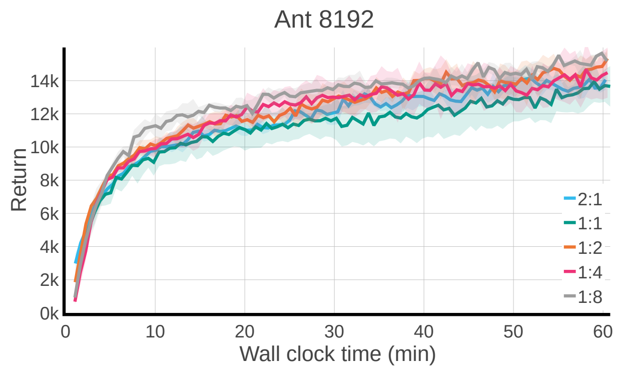
Distributional critic update We investigate how a distributional version of the critic update affects the policy learning performance. Here, we utilize categorical parameterization that outputs a discrete-value distribution defined over a fixed set of atoms z_i (Bellemare et al., 2017). We use the same hyper-parameters across the six tasks, where the number of atoms $l = 51$ and the bounds on the support from $(-10, 10)$. To make sure the values lie on the support defined by the atoms, we scale the reward into a similar range via different scaling factors shown in Table B.2 and apply the categorical projection operator before minimizing the cross-entropy.



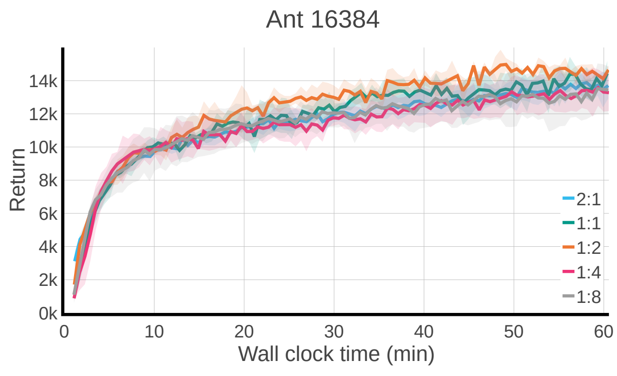
(a)



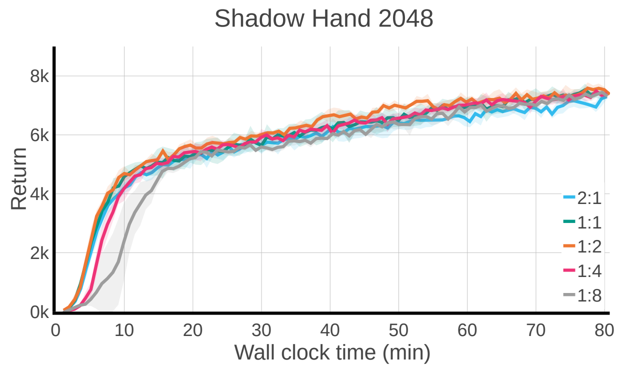
(b)



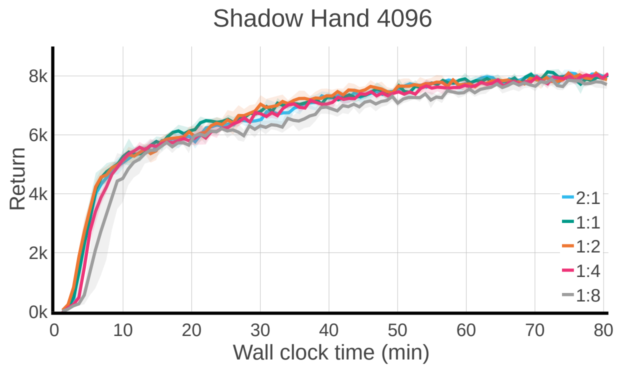
(c)



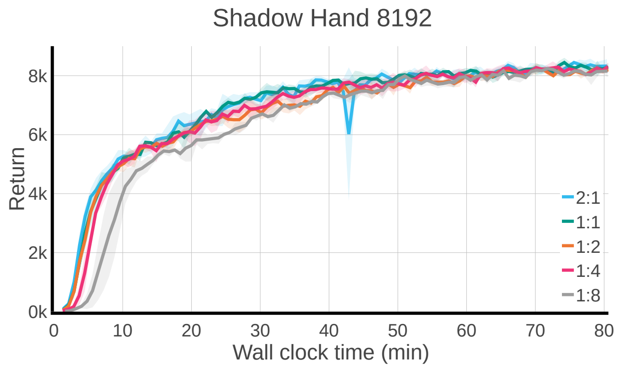
(d)



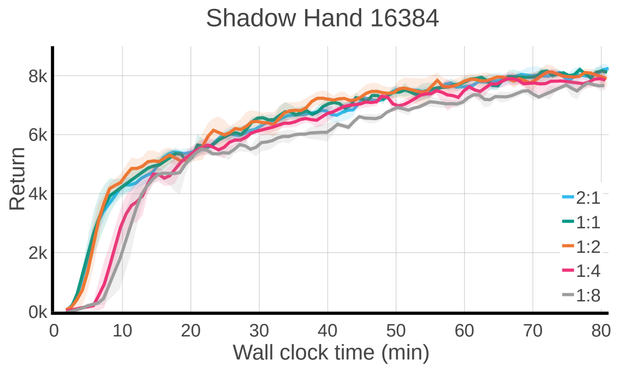
(e)



(f)

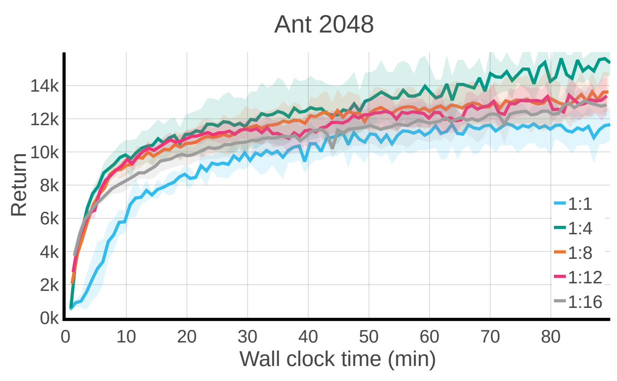


(g)

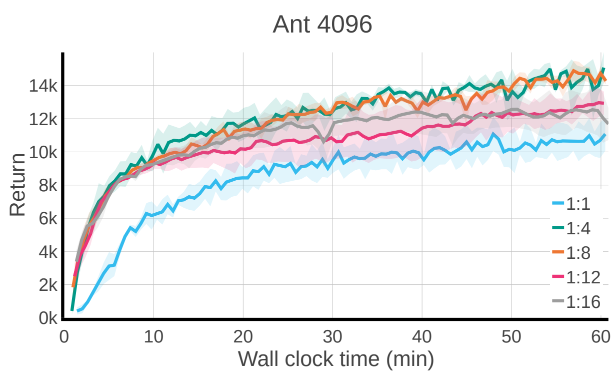


(h)

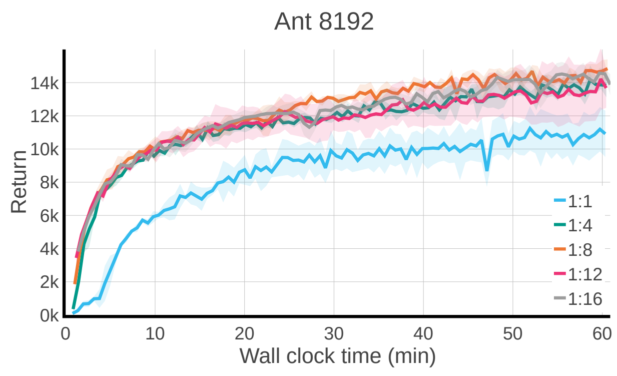
Figure C.6. Learning curves for different $\beta_{p:v}$.



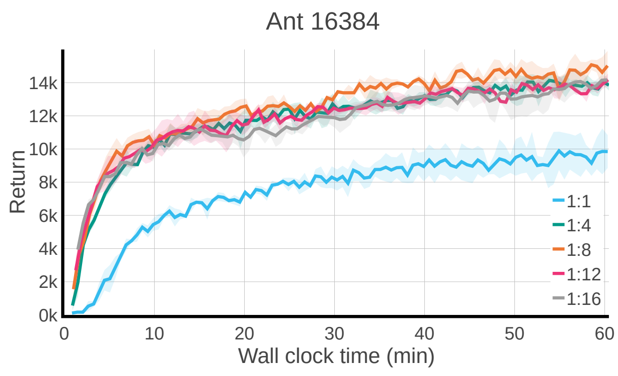
(a)



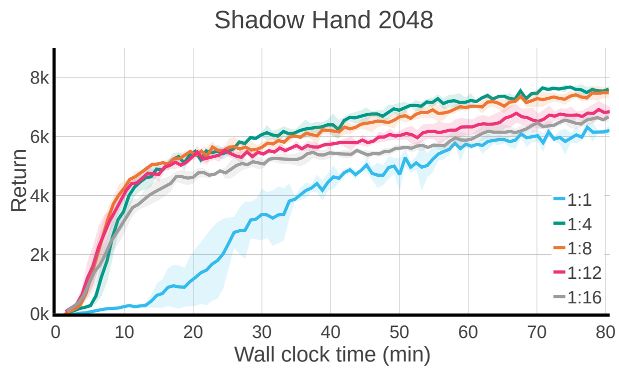
(b)



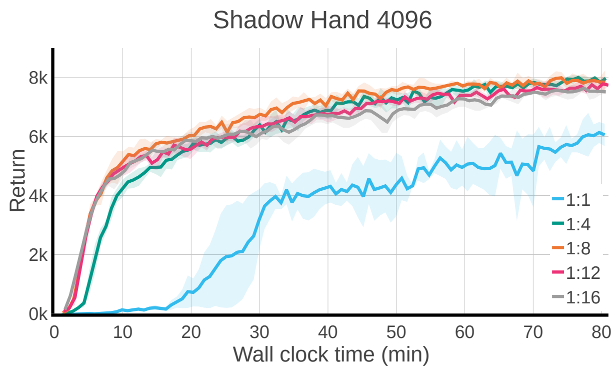
(c)



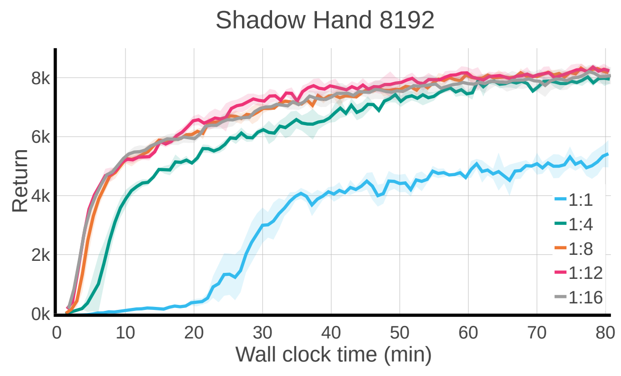
(d)



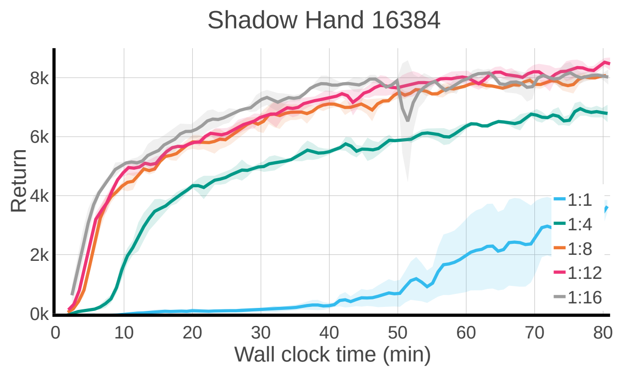
(e)



(f)

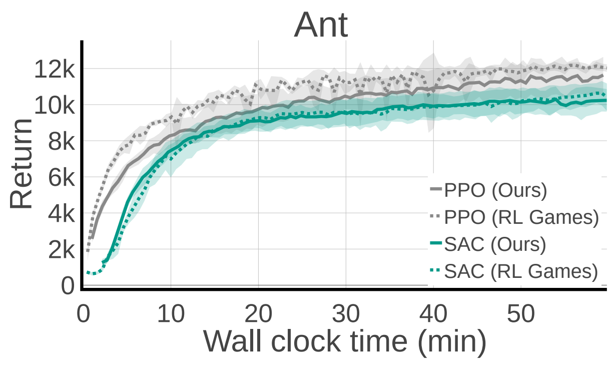


(g)

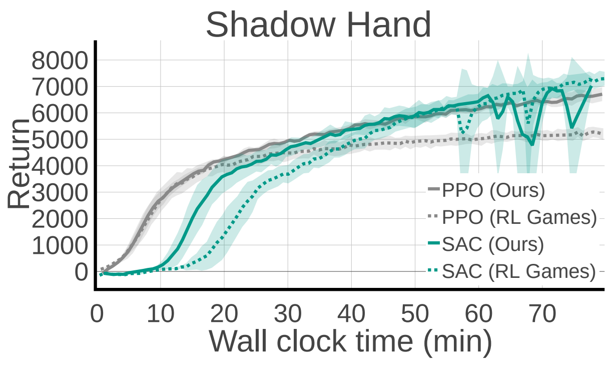


(h)

Figure C.7. Learning curves for different $\beta_{a:v}$.



(a)



(b)

Figure C.8. Comparison between our implementations of PPO and SAC against the ones provided in RL-games. We can see that both codebases provide similar performance, showing that our implementation is good and reliable. On *Shadow Hand*, our PPO learns even faster and better than the PPO in RL-games.