Recurrent Off-policy Baselines for Memory-based Continuous Control

Anonymous Author(s)
Affiliation
Address
email

Abstract

When the environment is partially observable (PO), a deep reinforcement learning (RL) agent must learn a suitable temporal representation of the entire history in addition to a strategy to control. This problem is not novel, and there have been model-free and model-based algorithms proposed for this problem. However, inspired by recent success in model-free image-based RL, we noticed the absence of a model-free baseline for history-based RL that (1) uses full history and (2) incorporates recent advances in off-policy continuous control. Therefore, we implement recurrent versions of DDPG, TD3, and SAC (RDPG, RTD3, and RSAC) in this work, evaluate them on short-term and long-term PO domains, and investigate key design choices. Our experiments show that RDPG and RTD3 can surprisingly fail on some domains and that RSAC is the most reliable, reaching near-optimal performance on nearly all domains. However, one task that requires systematic exploration still proved to be difficult, even for RSAC. These results show that model-free RL can learn good temporal representation using only reward signals; the primary difficulty seems to be computational cost and exploration. To facilitate future research, we have made our PyTorch implementation publicly available[1].

1 Introduction

In recent years, deep off-policy reinforcement learning (RL) algorithms based on learning the optimal Q-function is enjoying great success in fully observable continuous control domains [24, 5, 9, 10]. These algorithms usually achieve significantly better performance than deep on-policy RL methods given the same number of environment interactions and can learn highly competent policies on robotic locomotion tasks (e.g., MuJoCo [30] domains in [1]) where the state and action spaces are large. Their popularity also comes from their easy-to-implement update rules derived from the Bellman equation, and their robustness to hyper-parameters.

In this paper, we attempt to extend their success to partially observable (PO) domains. Continuous control under partial observability is an interesting problem for two reasons. First, continuous action and partial observability are common in real-world applications and often occur together, e.g., in robotics. Second, PO domains are more challenging than fully observable domains because, to succeed, an RL agent must learn a suitable temporal representation of the entire observation-action history, in addition to a strategy to control.

Continuous control under partial observability is not a novel problem, and there have been model-free and model-based algorithms proposed for this problem (see Section 2). However, we recognize the need for an open-source implementation of baseline algorithms that (1) are easy to implement, (2) can learn from the entire history, and (3) can learn stably and achieve good performance on a diverse

[1] https://anonymous.4open.science/r/off-policy-continuous-control-anonymous/

Submitted to Deep Reinforcement Learning Workshop, NeurIPS 2021. Do not distribute.

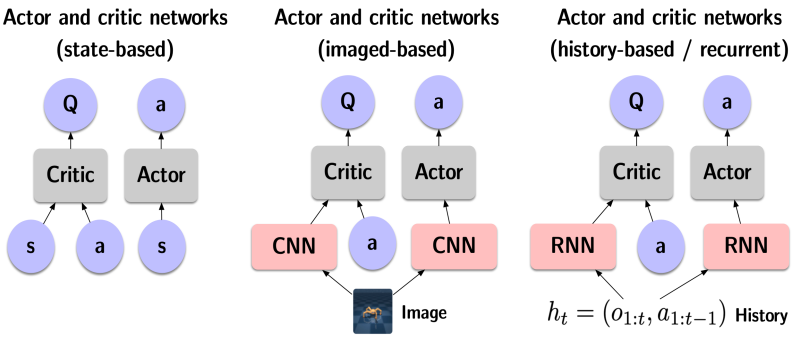


Figure 1: Comparison between architectures for standard, image-based, and recurrent off-policy model-free RL. The convolutional neural networks (CNNs) and recurrent neural networks (RNNs) can be trained end-to-end using gradients from loss terms, just like the actor and critic networks.

set of PO tasks. To fill in the first and second gap, this paper describes a neural network architecture (see the rightmost of Figure 1) that can be used to easily implement recurrent versions of DDPG, TD3, and SAC (RDPG, RTD3, and RSAC), and draws connection to a state-of-the-art image-based off-policy model-free algorithm DrQ [21] (see the middle of Figure 1). This architecture allows the recurrent agents to take into account the entire history $h_t = \{o_{1:t}, a_{1:t-1}\}$ at each timestep t, making it possible to learn arbitrary time dependencies. In addition, we adapt the standard experience replay procedure for these recurrent agents. To our knowledge, these agents are not available *as is* in popular RL repositories [16, 29, 6, 32].

To fulfill the third gap, we evaluate RDPG, RTD3, and RSAC on short-term and long-term PO domains. In addition, we investigate important design choices, including the sharing of recurrent representation across actors and critics, and the types of recurrent neural networks used. Our experiments show that RDPG and RTD3 can surprisingly fail on some domains and that RSAC is the most reliable, often reaching near-optimal performance on nearly all domains. However, one task that requires systematic exploration still proved to be difficult, even for RSAC. These results show that model-free RL can learn good temporal representation using only reward signals; the primary difficulty seems to be computational cost and exploration. In the end, we suggest future research directions. Note that we do not intend to compare our algorithms to other algorithms but show that one of our algorithms can achieve near-optimal performance in several domains.

2 Related work

There have been several recent works on learning memory-based policies in deep RL for continuous control. Recurrent Deterministic Policy Gradient (RDPG) [15] prepends recurrent layers to both the actor and critic networks of Deep Deterministic Policy Gradient (DDPG) [24], and was able to solve a variety of simple PO domains, including sensor integration and memory tasks. However, since it is based on DDPG, it inevitably suffers from value over-estimation and premature policy convergence, two problems addressed later by algorithms such as Twin-delayed DDPG (TD3) [5] and Soft Actor-critic (SAC) [9]. Moreover, it has not been publicly reproduced. To address value over-estimation and investigate high-dimensional tasks, [26] combines TD3 and LSTM [17] into TD3-LSTM. While TD3-LSTM shows good performance on high-dimensional sensor integration tasks, its network architecture is designed for POMDPs that are solvable with very short-term memory (e.g., 3-5 timesteps) and cannot be efficiently extended.

While the aforementioned works learn the temporal representation and the policy jointly, Variational Recurrent Model (VRM) [13] was proposed to learn these two separately and showed better performance than its SAC-LSTM implementation (see Appendix A.2). However, learning a separate model of the environment is a complex task, and the investigated domains are exclusively sensor integration tasks with either positional or velocity information and dense rewards. In this sense, VRM is related to several model-based algorithms proposed for image-based control tasks [23, 12, 11], since each image only offers positional information and an optimal policy must integrate such information across frames. However, learning from images entails a different set of challenges and is beyond our scope.

While previous approaches all fall under off-policy RL, various other approaches have been proposed for memory-based control in general. Several approaches build on the concept of belief states [20], a distribution over the state space given the entire history. Since the precise belief state is computationally expensive to track for continuous domains, algorithms mostly either work with small and discrete observation and action spaces (e.g., [28]) or only seek to approximate the belief (e.g., with particle filters [19], [25]). While both approaches require the use of RNNs, [33] does not and instead lets the policy read and set part of the state space as a form of memory and train the policy via directed policy search. However, it uses a linear policy and requires a well-defined latent space during training. Apart from continuous control, memory-based control in discrete action spaces has been explored mainly by adding recurrent layers to the Q-network [14, 34, 22] in DQN [27].

3 Background

3.1 Partially observable Markov Decision Processes (POMDP)

In real-world settings, an agent often can only have access to a reflection of the underlying state of the environment due to noise, occlusions, limited measurement resolutions, and so on. Under partial observability, a decision-making problem can be effectively modeled as a POMDP [20]. We can formally specify a finite-horizon POMDP by the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \Omega, \mathcal{O}, H, \gamma)$, where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, $\mathcal{T}(s_{t+1}|s_t, a_t)$ is the transition probability, $\mathcal{R}(s_t, a_t, s_{t+1})$ is the reward function, Ω is the observation space, $\mathcal{O}(o_t|s_{t+1}, a_t)$ is the observation probability, H is the horizon, and $\gamma \in [0, 1]$ is the discount factor. The goal is to learn a policy π that maximizes the expected discounted return for a finite horizon H defined as $\mathbf{E}_\pi[\sum_{t=0}^{H} \gamma^t r_t]$. To perform optimally in a POMDP, an agent must condition its policy on the entire history of observations and actions that it has seen so far. However, the size of the history space increases exponentially with the length of the history. Therefore, a recurrent neural network (RNN) is often used to summarize the history, and the policy is then condition on the RNN's fixed-size hidden states.

3.2 Off-policy RL methods for fully-observable continuous control

Deep deterministic policy gradient (DDPG) DDPG [24] is an extension of the tabular Q-learning to (1) continuous state spaces and (2) continuous action spaces. To deal with (1), it uses a deep neural network Q_ϕ to represent the Q-function, commonly known as the *critic*. It replaces the exact maximization problem in target computation (which is costly due to (2)) with approximation using an actor neural network μ_θ, commonly known as the *actor*. During training, given a batch of M transitions $\{(s_i, a_i, r_i, s'_i, d_i)\}_{i=1}^{M}$ uniformly sampled from a replay buffer, DDPG performs

$$\phi \leftarrow \phi - \eta \nabla_\phi \frac{1}{M} \sum_{i=1}^{M} \left(\underbrace{Q_\phi(s_i, a_i)}_{\text{prediction}} - \underbrace{(r_i + \gamma(1 - d_i) Q_{\phi_{\text{targ}}}(s'_i, \mu_{\theta_{\text{targ}}}(s'_i)))}_{\text{target}} \right)^2 \qquad \text{(train critic)}$$

$$\theta \leftarrow \theta + \eta \nabla_\theta \frac{1}{M} \sum_{i=1}^{M} Q_\phi\left(s_i, \mu_\theta\left(s_i\right)\right), \qquad \text{(train actor)}$$

where γ is the discount factor, η is the learning rate, and ϕ_{targ} and θ_{targ} are slowly changing versions of ϕ and θ obtained through polyak averaging.

Twin-delayed DDPG (TD3) TD3 [5] addresses the issue of value over-estimation in DDPG. Specifically, in DDPG, $\mu_{\theta_{\text{targ}}}$ may exploit[2] erroneously high values of $Q_{\phi_{\text{targ}}}(s, \cdot)$ and hence make the targets too large. As a result, bad actions at certain states may end up having over-estimated Q-values. In response, TD3 introduces three tricks. First, random noise is added to the output of $\mu_{\theta_{\text{targ}}}(s)$; this is called target smoothing. Second, μ_θ is trained less frequently than Q_ϕ; this is called delayed policy update. Finally, an additional Q-network is added, and both Q-networks are updated using the same target, computed using the minimum of the target versions of themselves; this is called clipped double-Q learning. The actor is updated with respect to only one Q-network. TD3 significantly out-performs DDPG on several dense-reward high-dimensional continuous-control domains.

[2] This is because $\mu_\theta(s)$ is trained to maximize $Q_\phi(s, \cdot)$ and may exploit its erroneously high values; $\mu_{\theta_{\text{targ}}}(s)$ and $Q_{\phi_{\text{targ}}}(s, \cdot)$ are slowly changing versions of $\mu_\theta(s)$ and $Q_\phi(s, \cdot)$, and hence may inherit this property.

Soft Actor-Critic (SAC) Unlike DDPG and TD3, SAC [9] was derived from Soft Policy Iteration instead of the standard policy iteration. Nevertheless, SAC is similar to TD3 in two important ways. First, SAC learns a stochastic actor network, so its next-state actions used for target computation are sampled from a distribution, similar to target smoothing in TD3. Second, SAC also maintains two Q-networks, but uses both of them for training the actor network. However, SAC does not use target actor network or delayed policy update. In practice, TD3 and SAC have similar performance.

3.3 Recurrent neural networks

Recurrent neural networks (RNNs) can be best thought as a function that, at each timestep t, takes in (1) the previous summary (summary_{t-1}) and (2) some new information (x_t), and produces (1) some output (output_t) for subsequently layers and (2) some summary for the next timestep (summary_t). Despite this simple setup, several parametrizations have been proposed for learning this function effectively, with the most popular ones being Vanilla RNN (VRNN) [4], Long Short-Term Memory (LSTM) [17], and Gated Recurrent Unit (GRU) [3]. For more details, please refer to Section A.1. Since RNNs are differentiable with respect to their parameters, they can be trained using gradient descent in a procedure known as Backpropagation Through Time (BPTT).

4 Architecture & algorithm

In this section, we describe the network architecture and algorithms that we used to train RDPG stably. We outline RDPG instead of RTD3 or RSAC because DDPG uses the least number of tricks for improving stability. Then, we describe how it can be easily extended for RTD3 and RSAC.

Architecture DDPG involves the interaction between an actor $\mu_\theta(s)$ and a critic $Q_\phi(s, a)$. To adapt for POMDPs, recall that $h_t = (o_{1:t}, a_{1:t-1})$ approximates s_t in POMDPs. Therefore, we modify the actor to be $\mu_\theta(h_t)$ and the critic to be $Q_\phi(h_t, a_t)$. However, it is in general not possible to pass h_t as is since fully-connected neural networks do not accept variable-sized inputs. To that end, we applied RNNs to distill and summarize h_t into a fixed-size hidden vector $\hat{h}_t$. We also modified the standard experience replay to store individual episodes instead of individual transitions.

Algorithm Formally, each update step consists of the following three sub-steps. First, one episode (a larger batch of episodes is used in practice) is sampled uniformly from the replay buffer:

$$(o_1, \ldots, o_{T+1}), (a_0, \ldots, a_T), (r_1, \ldots, r_T), (d_1, \ldots d_T),$$

where d_t represents the episode termination flag and a_0 is the first dummy action.

Second, using this episode, the hidden vectors can be computed by unrolling an RNN over the sequence of observations and actions as follows ($\|$ denotes concatenation):

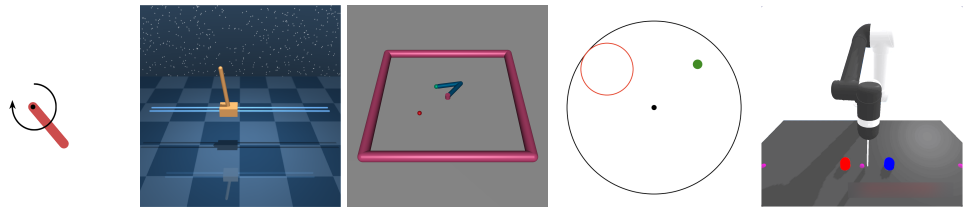
$$\begin{aligned}
\hat{h}_1^{\phi}, \ldots, \hat{h}_{T+1}^{\phi} &= \text{RNN}_\phi\left(o_1 \,\|a_0, \ldots, o_{T+1}\| \, a_T\right) \\
\hat{h}_1^{\phi_{\text{targ}}}, \ldots, \hat{h}_{T+1}^{\phi_{\text{targ}}} &= \text{RNN}_{\phi_{\text{targ}}}\left(o_1||a_0, \ldots, o_{T+1}\|a_T\right) \\
\hat{h}_1^{\theta}, \ldots, \hat{h}_{T+1}^{\theta} &= \text{RNN}_\theta\left(o_1||a_0, \ldots, o_{T+1}||a_T\right) \\
\hat{h}_1^{\theta_{\text{targ}}}, \ldots, \hat{h}_{T+1}^{\theta_{\text{targ}}} &= \text{RNN}_{\theta_{\text{targ}}}\left(o_1||a_0, \ldots, o_{T+1}\|a_T\right),
\end{aligned}$$

where we used a separate RNN for each of the non-recurrent actor, the non-recurrent critic, and their target networks.

Third, parameter updates to the recurrent actor and critic can be computed as follows:

$$\begin{aligned}
\phi &\leftarrow \phi - \eta \nabla_\phi \left\{ \frac{1}{T} \sum_{t=1}^{T} \left(Q_\phi\left(\hat{h}_t^{\phi}, a_t\right) - \left(r_t + \gamma\left(1 - d_t\right) Q_{\phi_{\text{targ}}}\left(\hat{h}_{t+1}^{\phi_{\text{targ}}}, \mu_{\theta_{\text{targ}}}\left(\hat{h}_{t+1}^{\theta_{\text{targ}}}\right)\right)\right)\right)^2 \right\} \\
\theta &\leftarrow \theta + \eta \nabla_\theta Q_\phi\left(\hat{h}_t^{\phi}, \mu_\theta\left(\hat{h}_t^{\theta}\right)\right)
\end{aligned}$$

and ϕ_{targ} and θ_{targ} are updated using polyak averaging. Unlike in Section 3.2, here ϕ denotes parameters of the *entire* recurrent critic (the non-recurrent critic plus its corresponding RNN); the same logic applies to θ, ϕ_{targ}, and θ_{targ}. Gradients flow through the hidden vectors and reach the RNN parameters, allowing the entire architecture to be trained end-to-end. For some environments, episodes have variable lengths, so masking is used.



(a) `pendulum` (b) `cartpole` (c) `reacher` (d) `watermaze` (e) `push-r-bump`

Figure 2: Domains to perform experiments. For `watermaze` (d), the agent is illustrated as the green dot and the platform as the red circle.

Extending to RTD3 and RSAC Unlike DDPG, TD3 and SAC use two critics instead of one. This difference is preserved in their recurrent versions, and hence they both use three RNNs (not counting the ones in target networks): one for the actor, two for the two critics. Tricks such as clipped double-Q learning and entropy bonus rewards are directly ported from their non-recurrent versions.

5 Domains

5.1 Sensor integration tasks

Optimal control of physical systems might require integrating sensor inputs over a few past timesteps due to noise or the difficulty of measuring correctly all relevant physical quantities. Here, we use the position/velocity-only versions of three fully-observable physical-control domains (see a, b of Figure 2), similar to domains investigated in [15], [13], and [26]. These domains are `pendulum-swingup` ($\dim(\mathcal{S}) = 3, \dim(\mathcal{A}) = 1$), `cartpole-balance` ($\dim(\mathcal{S}) = 5, \dim(\mathcal{A}) = 1$), and `cartpole-swingup` ($\dim(\mathcal{S}) = 5, \dim(\mathcal{A}) = 1$); they are generally considered to be increasingly more difficult. They all have a maximum episode length T of 200 and use a dense reward function. Their position-only versions remove all entries of the observation related to velocity, and their velocity-only versions remove all entries related to position. We also append the previous action to the velocity-only observations, which we observed to improve the performance of recurrent agents.

5.2 Memory tasks

Another important type of PO tasks are memory tasks, where the agent needs to memorize (instead of simply integrate) some information over timesteps. We evaluate our agents on two memory tasks: `reacher-pomdp` and `watermaze`. `reacher-pomdp` (see Figure 2c) ($T = 50$) modifies the original 2-joint reacher task from OpenAI Gym [1] so that the location of the goal only appears for the first timestep. This task provides dense rewards. `watermaze` (see Figure 2d) ($T = 200$) is borrowed from [15]. Since the paper does not open-source its domains, we re-implemented it with minor modifications. `watermaze` simulates a task that is used to study spatial learning for rodents. In this task, the agent (e.g., a mouse) is released at the center of a circular world. It is rewarded by being on top of an invisible circular platform, which is randomly positioned per episode. The agent can only observe the platform when being on top of the platform, and is taken back to the center afterwards. To maximize return, the agent must memorize the location of the platform to return to it faster.

5.3 Active exploration task

In a PO world, it is very common for an agent to perform non-rewarding but informative actions to be more certain about the world before performing rewarding actions. As an example, we created a domain called `push-r-bump` (see Figure 2e) ($T = 50$), which is a continuous version of a similar domain in [28]. In this domain, a robot controls the movement and the stiffness of a finger. When the finger is soft, it can glide over bumps without moving them, causing reactive angles on the finger. In contrast, when the finger is stiff enough, it can move bumps. Any bump movement will terminate the current episode and will bring a reward of 1 if the right bump is pushed right for a short distance and -1 otherwise. The robot can observe the finger's position and angle, but it does not observe the bumps' locations. Starting each episode, the position of the finger is initialized around the center, and

the two bumps are randomized to one of three possible cases (see Figure 3). Therefore, an optimal agent must first use a soft finger to scan around and localize the right bump, and finally push this bump to the right for a short distance with a stiff finger to get a positive reward.

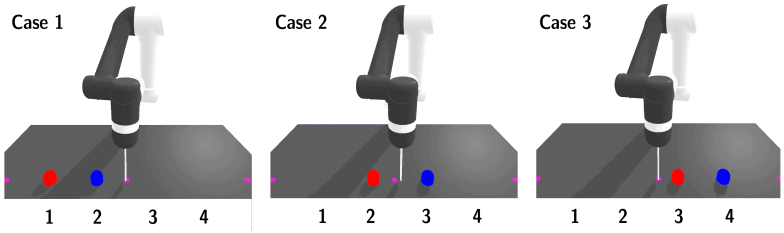


Figure 3: Three cases in `push-r-bump`; colors and numbers are for illustration and are not observed.

6 Experiments & results

Unless otherwise noted, (1) RDPG, RTD3, and RSAC take in observations instead of states, and (2) the default RNN used is LSTM. For return, we plot the mean and min/max (1 standard deviation for other statistics) w.r.t. 5 seeds as in [9]. Statistics are recorded once per 1k steps (1k environment steps and 1k update steps). Each return recorded is the average return over 10 episodes with deterministic actors. Training time is w.r.t. Nvidia's RTX 2080 (16G). Hyper-parameters are in Appendix A.3.

6.1 Performance on sensor integration tasks

From the 3rd and 4th column of Table 1, we see that RSAC is the only algorithm to achieve similar performance at convergence as `SAC-state` on all PO domains. Interestingly, RDPG has 2 failed seeds in `pendulum-p` and failed completely in `cartpole-swingup-p/v`. It is perhaps surprising that RDPG can have failed seeds on `pendulum-p`, because the MDP version of `pendulum-p` is arguably one of the simplest continuous control domains.

It is well know that, in DDPG, the actor can exploit erroneously high values in the critics and destabilize learning [5]. Since the actor in RDPG is augmented with a two-layer RNN, the actor is more expressive and may hence be a stronger exploiter of such errors. However, since RDPG can perform reasonably well (despite one failed seed in `cartpole-swingup`) on the MDP versions of the domains (see the second column of Table 1), we argue that this source of exploitation, if exists, is not the primary cause for the algorithm to diverge on PO variants. Instead, we hypothesize that partial observability itself may open up more room for such exploitation, and is worthy of further investigation. A supporting evidence is that RTD3 and RSAC have successful seeds on `cartpole-swingup-p/v`, presumably because of their mechanisms designed to address such exploitation.

RTD3 performs better than RDPG, and has similar performance at convergence as RSAC on `pendulum-p/v` and `cartpole-balance-p/v`. However, RTD3 learns slower than RSAC on `pendulum-p/v`, and has one failed seed in both `cartpole-swingup-p/v`. Interestingly, `RTD3-state` also has two failed seeds in the MDP version of `cartpole-swingup`, suggesting that this failing behavior may be due to RTD3's brittleness with more expressive actor and critics instead of partial observability. Empirically, through visualizing the learned policies in `cartpole-swingup-p/v`, we noticed that RTD3 can get stuck in a suboptimal policy of quickly moving the cart to one side to briefly drag the pole to its horizontal position (away from the downward vertical position) to gain some reward.

As aforementioned, RSAC is the only algorithm that matches its non-recurrent MDP performance on all PO tasks. Algorithmically, RSAC is highly similar to RTD3 in terms of many design choices, except that its actor maximizes its entropy in addition to the return. Entropy maximization can prevent premature convergence of the actor, and has been shown to explore equally rewarding avenues of behavior much better than simply adding random noise to a deterministic policy [8]. In fact, exploration may be crucial to learning an expressive temporal representation from scratch since better exploration leads to a wider distribution of observations and rewards, which prevents over-fitting due to limited experience. In practice, this may help RSAC to be less affected by a large network and learn more consistently than RTD3 (e.g., by escaping local optima).

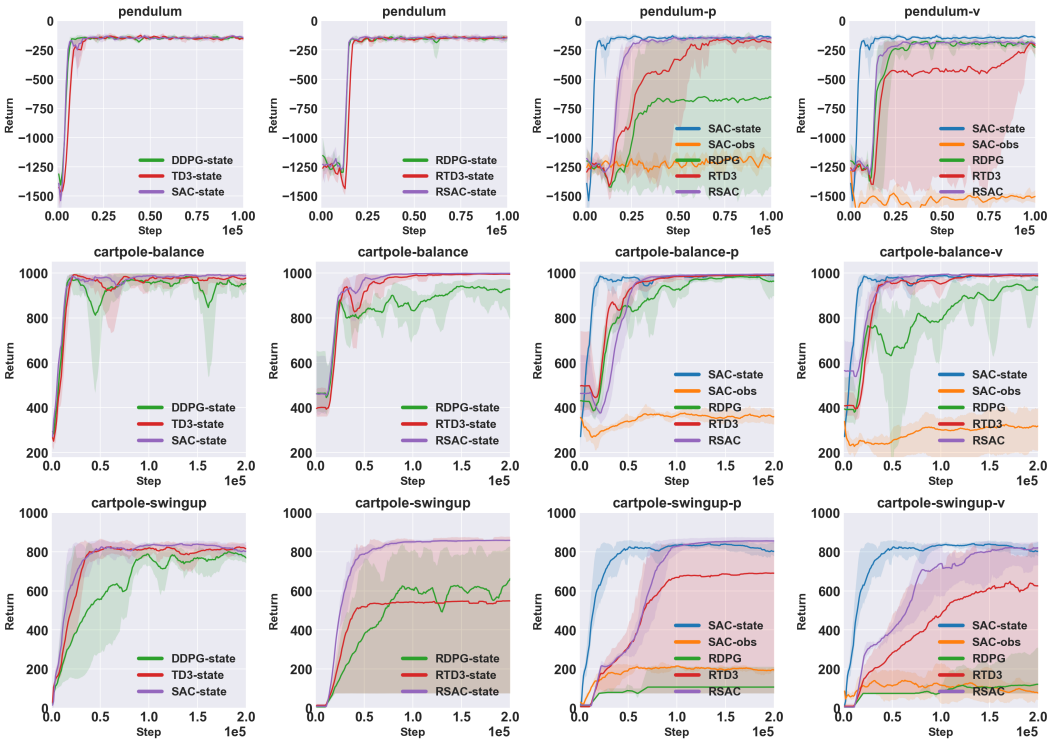


Table 1: Performance on sensor integration tasks. DDPG, TD3, and SAC all learn competent policies given the state. However, the performance of DDPG and TD3 degrade with LSTMs - wide error bands are due to splits between successful and failed seeds.

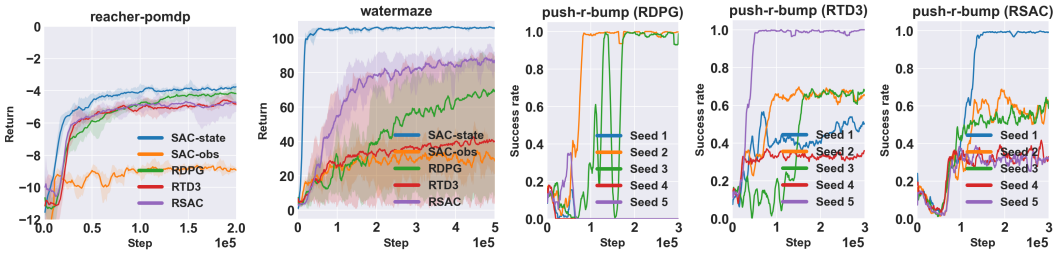


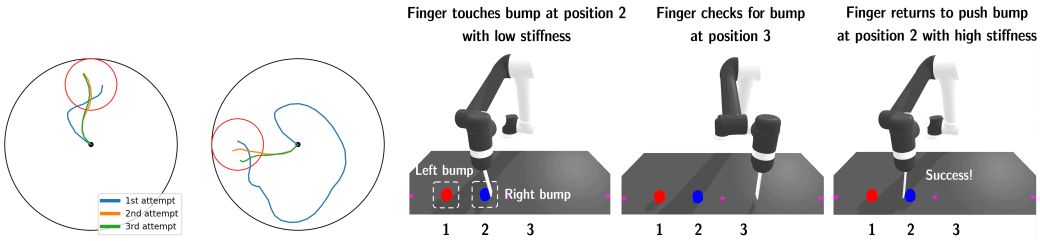
Table 2: Performance on memory and active exploration tasks. For `push-r-bump`, we plot individual seeds' success rates to show that neither RDPG, RTD3, nor RSAC can consistently solve the task.

6.2 Performance and exemplary trajectories on memory and active exploration tasks

RDPG, RTD3, and RSAC all learn competent policies on `reacher-pomdp` (see Table 2), where dense rewards are available. In contrast, RSAC is the only algorithm that performs well and consistently on `watermaze`, which offers only sparse rewards and requires active exploration at the beginning of training. RSAC is trained under the entropy maximization framework, potentially making the policy explore better throughout training. We illustrate one RSAC seed in Figure 4a and 4b; it learns an exploratory policy and can memorize the platform position to return quickly within fewer timesteps.

In `push-r-bump`, no algorithm can consistently solve the task to 100% success rate. Still, RDPG, RTD3, and RSAC all have successful seeds, and RTD3 and RSAC have seeds that solve 1 or 2 cases (among the 3 mentioned in Section 5.3). This is likely an issue of insufficient exploration, since systematic exploration is required for the algorithms to solve all 3 cases correctly by chance to obtain sparse positive rewards to start learning. We visualize 1 successful RTD3 seed on 1 case in Figure 4c, and the same seed on 2 other cases in Appendix A.5 along with how RNN cell states evolve.

6.3 Sharing a single recurrent representation across the actor and the critics



(a) Platform at 90° (b) Platform at 180° (c) Case 1 in Figure 3; events happen from left to right.

Figure 4: Visualization of learned policies. (a) (b) One RSAC seed trained on `watermaze`. (c) One RTD3 seed trained on `push-r-bump` executing in one among three cases.

In RSAC, there are 5 recurrent modules: 1 within the actor, 2 within the two critics, and 2 within the two critic targets. In this setup, each update step involves 5 rounds of RNN forward passes and 3 rounds of BPTTs. Since each round of computation scales with T, it is natural to ask whether sharing a *single* recurrent module across them (1 for all non-target networks, and 1 for all target networks) could have similar performance as without sharing while using less computation. We call this approach RSAC-Share. While it is straightforward to optimize parameters of the single shared representation w.r.t. both the actor and the two critic losses, these two kinds of losses might have different scales and could lead to instability [15]. Additionally, while this approach only involves 2 rounds of RNN forward passes, it still requires 3 rounds of BPTTs.

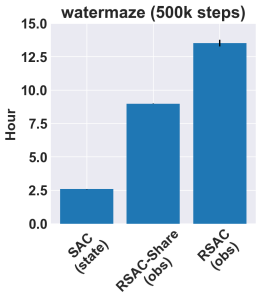


Figure 5: RSAC-Share reduces training time.

Seeking further improvement, we borrow the insights from DrQ [21], which trains SAC on image-based domains and shares a single convolutional neural network (CNN) across an actor and two critics. In DrQ, the shared CNN accumulates gradients from both critic losses; the gradients of the actor loss w.r.t. the CNN's parameters are *ignored*. Similarly, for RSAC-Share, we optimize the parameters of the shared recurrent module w.r.t. the two critic losses only. Intuitively, using gradients from the critic losses (instead of the actor loss) helps the shared recurrent module learn a richer representation, since, given a state, the critics must learn values for all actions but the actor only needs to learn the maximizing action. Doing so also reduces computation: the final version of RSAC-Share uses 2 rounds of RNN forward passes and BPTTs per update step, halving the RNN computation in RSAC.

Our experiments show that RSAC-Share is on par with RSAC (see the first row in Table 3), and significantly reduces training time (see Figure 5). RSAC-Share is less stable on `cartpole-swingup-v`, but reaches better performance at convergence on `reacher-pomdp`.

We have two interesting observations. Firstly, the Pearson correlation (see Appendix A.4) of Q-values produced by the two critics is higher for RSAC-Share (see the second row in Table 3), presumably because they share the same recurrent representation. Such high correlation potentially makes clipped double Q-learning in RSAC-Share ineffective, since the minimum of two highly correlated random variables is almost equal to either one of the two. Secondly, RSAC-Share learns higher Q-values than RSAC does (see the third row in Table 3); this is likely due to the first observation, and that the actor in RSAC-Share may be a stronger maximizer of the critics' outputs, since it directly acts on the critics' recurrent representation. While these observations suggest risks of value over-estimation, RSAC-Share has strong empirical performance, and out-performs both RDPG and RTD3. Note that, for both observations, Q-values are calculated from 10 episodes sampled from the replay buffer.

6.4 Comparison among VRNN, LSTM, and GRU

In earlier experiments, we used LSTM as the recurrent representation since it has shown to be hard to beat on a variety of tasks [7], and was used as the recurrent component of several deep RL algorithms [14, 15, 13, 26]. In this section, we compare RSAC-LSTM, RSAC-VRNN, and RSAC-GRU against each other. Our experiments (see Table 4) show that RSAC-LSTM significantly out-performs RSAC-VRNN and is on par with RSAC-GRU, indicating that using either LSTM or GRU as the recurrent

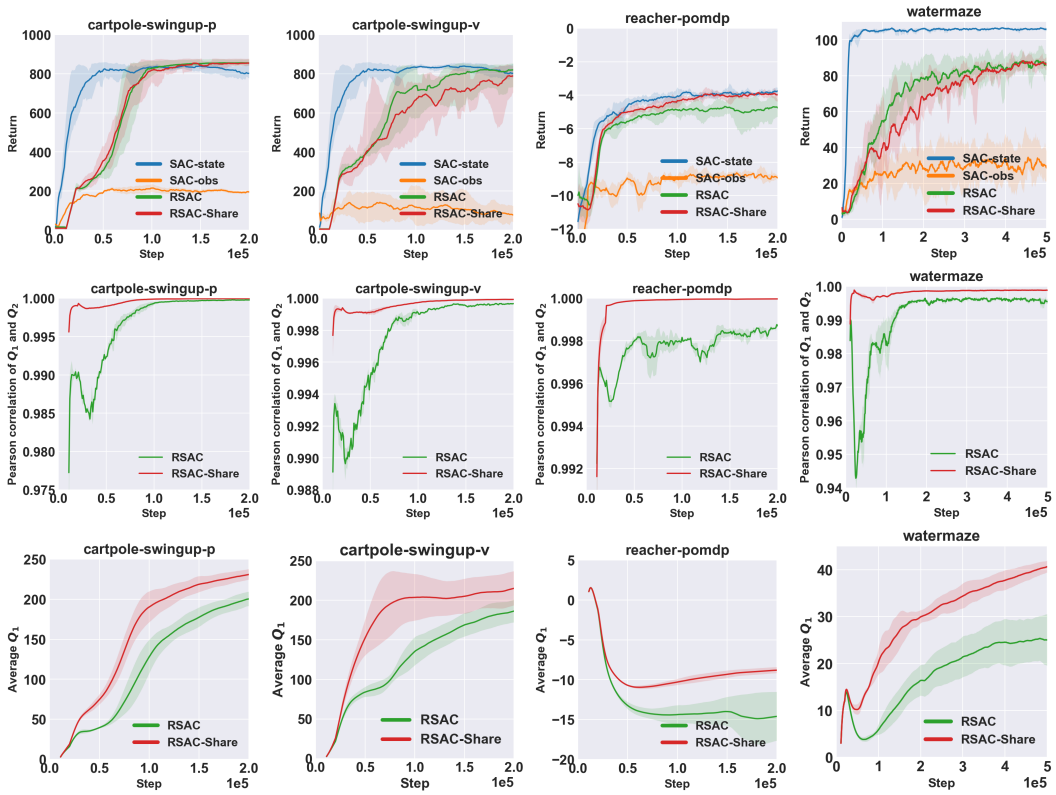


Table 3: Performance of RSAC-Share against RSAC. The first row shows that RSAC-Share is on par with RSAC in terms of return. The second row shows that RSAC-Share has higher correlation between critics' outputs. The third row shows that RSAC-Share learn higher values.

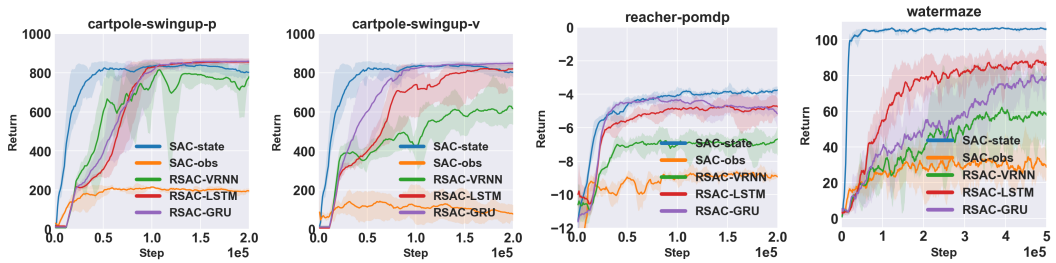


Table 4: Performance of RSAC-LSTM, RSAC-VRNN, and RSAC-GRU. RSAC-LSTM and RSAC-GRU are on par with each other, and both greatly out-performs RSAC-VRNN.

representation is an equally reasonable choice. Interestingly, RSAC-GRU and RSAC-LSTM have different speeds of learning, but seem to achieve similar asymptotic performance. RSAC-VRNN performed the worst, presumably due to the well-known vanishing/exploding-gradient problem [18] when training VRNN using long sequences.

7 Conclusions

We present several recurrent off-policy baselines, tested on a diverse set of continuous control tasks. Our experiments show that RSAC is the most reliable algorithm out of the three baselines investigated RSAC, RDPG, and RTD3. However, they still face difficulties when solving tasks which provide only sparse rewards and require active exploration. Unfortunately, these two characteristics are often common in interesting real-world POMDPs. Another limitation is the inability to deal with too long episodes, inheriting from known weakness of RNNs (e.g., long training time, difficulties learning long-term dependencies). This suggests a potential research direction of using transformers [31], which improve upon these weaknesses, as an alternative to RNNs for solving POMDPs.

References

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[2] Laura M Chihara and Tim C Hesterberg. *Mathematical statistics with resampling and R*. John Wiley & Sons, 2018.

[3] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[4] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[5] Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. *International Conference on Machine Learning*, 2018.

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

[7] Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 28 (10):2222–2232, 2016.

[8] Tuomas Haarnoja, Haoran Tang, Pieter Abbeel, and Sergey Levine. Reinforcement learning with deep energy-based policies. In *International Conference on Machine Learning*, pages 1352–1361. PMLR, 2017.

[9] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *International Conference on Machine Learning*, 2018.

[10] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, et al. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905*, 2018.

[11] Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to control: Learning behaviors by latent imagination. *arXiv preprint arXiv:1912.01603*, 2019.

[12] Danijar Hafner, Timothy Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. Learning latent dynamics for planning from pixels. In *International Conference on Machine Learning*, pages 2555–2565. PMLR, 2019.

[13] Dongqi Han, Kenji Doya, and Jun Tani. Variational recurrent models for solving partially observable control tasks. *International Conference on Learning Representations*, 2020.

[14] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. In *2015 aaai fall symposium series*, 2015.

[15] Nicolas Heess, Jonathan J Hunt, Timothy P Lillicrap, and David Silver. Memory-based control with recurrent neural networks. *Deep Reinforcement Learning Workshop, NeurIPS*, 2015.

[16] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable baselines. `https://github.com/hill-a/stable-baselines`, 2018.

[17] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 1997.

[18] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, Jürgen Schmidhuber, et al. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

[19] Maximilian Igl, Luisa Zintgraf, Tuan Anh Le, Frank Wood, and Shimon Whiteson. Deep variational reinforcement learning for pomdps. In *International Conference on Machine Learning*, pages 2117–2126. PMLR, 2018.

[20] Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1-2):99–134, 1998.

[21] Ilya Kostrikov, Denis Yarats, and Rob Fergus. Image augmentation is all you need: Regularizing deep reinforcement learning from pixels. *arXiv preprint arXiv:2004.13649*, 2020.

[22] Guillaume Lample and Devendra Singh Chaplot. Playing fps games with deep reinforcement learning. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[23] Alex X Lee, Anusha Nagabandi, Pieter Abbeel, and Sergey Levine. Stochastic latent actor-critic: Deep reinforcement learning with a latent variable model. *arXiv preprint arXiv:1907.00953*, 2019.

[24] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *International Conference on Machine Learning*, 2015.

[25] Xiao Ma, Peter Karkus, David Hsu, Wee Sun Lee, and Nan Ye. Discriminative particle filter reinforcement learning for complex partial observations. *arXiv preprint arXiv:2002.09884*, 2020.

[26] Lingheng Meng, Rob Gorbet, and Dana Kulić. Memory-based deep reinforcement learning for pomdp. *International Conference on Intelligent Robots and Systems*, 2021.

[27] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 2015.

[28] Hai Nguyen, Brett Daley, Xinchao Song, Christopher Amato, and Robert Platt. Belief-grounded networks for accelerated robot learning under partial observability. *arXiv preprint arXiv:2010.09170*, 2020.

[29] Antonin Raffin, Ashley Hill, Maximilian Enerstus, Adam Gleave, Anssi Kanervisto, and Noah Dormann. Stable Baselines3, 5 2020. URL `https://github.com/DLR-RM/stable-baselines3`.

[30] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

[31] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[32] Jiayi Weng, Huayu Chen, Dong Yan, Kaichao You, Alexis Duburcq, Minghao Zhang, Hang Su, and Jun Zhu. Tianshou: A highly modularized deep reinforcement learning library. *arXiv preprint arXiv:2107.14171*, 2021.

[33] Marvin Zhang, Zoe McCarthy, Chelsea Finn, Sergey Levine, and Pieter Abbeel. Learning deep neural network policies with continuous memory states. In *2016 IEEE international conference on robotics and automation (ICRA)*, pages 520–527. IEEE, 2016.

[34] Pengfei Zhu, Xin Li, Pascal Poupart, and Guanghui Miao. On improving deep reinforcement learning for pomdps. *arXiv preprint arXiv:1704.07978*, 2017.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes] One of our key claims (in the abstract and introduction) is that RSAC is the most reliable agent and reaches near-optimal performance on nearly all domains. This claims can be verified using Figure 1 and 2. We also claimed to investigate important choices of recurrent agents, which we indeed do in Section 6.3 and 6.4.
   (b) Did you describe the limitations of your work? [Yes] We do so in both the abstract and Section 7.
   (c) Did you discuss any potential negative societal impacts of your work? [N/A]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] Please refer to the hyperlink mentioned in Abstract.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] Please refer to the first paragraph of Section 6.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] Please refer to the first paragraph of Section 6 for the number of random seeds used and the type of error bars plotted.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] Yes, we have reported the total number of hours required for 500k training steps in Figure 5. The number of hours per 100k training steps was quite similar across environments. For the type of GPU used, please refer to the first paragraph of Section 6.
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [N/A]
   (b) Did you mention the license of the assets? [N/A]
   (c) Did you include any new assets either in the supplemental material or as a URL? [N/A]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]