Look Back When Surprised: Stabilizing Reverse Experience Replay for Neural Approximation

Ramnath Kumar*
Google AI Research Lab,
Bengaluru, India 560016,
ramnathk@google.com

Dheeraj Nagaraj
Google AI Research Lab,
Bengaluru, India 560016,
dheerajnagaraj@google.com

Abstract

Experience replay methods, which are an essential part of reinforcement learning (RL) algorithms, are designed to mitigate spurious correlations and biases while learning from temporally dependent data. Roughly speaking, these methods allow us to draw batched data from a large buffer such that these temporal correlations do not hinder the performance of descent algorithms. In this experimental work, we consider the recently developed and theoretically rigorous reverse experience replay (RER), which has been shown to remove such spurious biases in simplified theoretical settings. We combine RER with optimistic experience replay (OER) to obtain RER++, which is stable under neural function approximation. We show via experiments that this has a better performance than techniques like prioritized experience replay (PER) on various tasks, with a significantly smaller computational complexity. It is well known in the RL literature that choosing examples greedily with the largest TD error (as in OER) or forming mini-batches with consecutive data points (as in RER) leads to poor performance. However, our method, which combines these techniques, works very well.

1 Introduction

Reinforcement learning (RL) involves learning with dependent data, and algorithms designed for independent data might behave poorly by getting coupled with the Markovian trajectories encountered in this setting. Therefore, techniques like experience replay [17] are usually deployed with Q-learning type algorithms to achieve state-of-the-art performance [19, 16]. It has been shown experimentally [19] and theoretically [5] that these learning algorithms behave sub-optimally without experience replay.

From the simplest form of experience replay as used in [19], where uniform sampling from a buffer is used (UER), several specialized methods have been proposed and evaluated experimentally. These include prioritized experience replay (PER) [23], hindsight experience replay (HER) [2], reverse experience replay (RER) [22], and topological experience replay (TER) [12]. The design of experience replay continues to be an active field of research; however, theoretical analyses have been limited. Recent results on learning dynamical systems [14, 13] showed rigorously in a theoretical setting that RER is the conceptually-grounded algorithm when learning from Markovian data, and indeed this was extended to the RL setting in [1] to achieve efficient Q learning with linear function approximation. However, RER seems unstable when used with neural function approximation like in DQN.

Our work introduces a modification of RER, namely RER++, which is stable when used with neural function approximation. Roughly speaking, RER first picks top k 'pivot' points from a large

*Work done while author was a Research Associate at Google Research, India.

Preprint. Under review.

buffer according to their TD error. It then returns batches of data which are formed by picking the consecutive points *temporally* before these pivot points. In essence, the algorithm 'looks back when surprised'. The summary of our approach is shown in Figure 1.

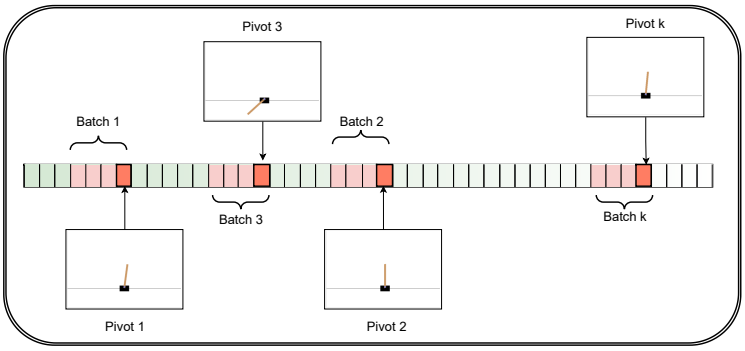


Figure 1: An illustration of the proposed methodology when selecting k batches in the CartPole environment.

Important findings: It is well known that naive importance sampling (i.e., picking examples with the largest TD error to form a minibatch) or using temporally consecutive data points (i.e., creating mini-batches with data points that are temporally adjacent) are both extremely ineffective. The former procedure incurs a significant bias error as discussed in [23], which is why PER uses a sophisticated sampling procedure to mitigate such a bias. Surprisingly, our algorithm, which combines these ineffective heuristics, works very well. One possible explanation of this phenomenon is that, due to the de-biasing nature of RER ([14]), even this naive importance selection in RER++ is efficient. Our experiments on various environments show that RER++ outperforms other standard experience replay methods with a much simpler algorithm and a fraction of the computational cost compared to PER. We also demonstrate via the ablation study in Appendix C that picking consecutive points temporally *after* the pivot points (that is, Forward Experience Replay++ or FER++) does not perform as well as RER++.

Organization: In Section 2, we discuss relevant prior works. We introduce the problem and the high-level objectives of this study in Section 3. Section 4 introduces RER++ and HRER++, which are the primary algorithms whose performance is studied in this work and provides several intuitive explanations for reverse replay. We describe the experimental setup and discuss the performance of our method compared to various standard experience replay methods in Section 5. Discussions, conclusions, and ablation studies are given in Section 6. We perform a temporal (forward or backward) ablation study of our method and briefly describe our results in Appendix C. In Appendix E, we discuss the traditional Reverse Experience Replay approach used. We provide some evidence regarding our sparse reward propagation hypothesis on Cartpole and Ant environments in Appendix D.

2 Related Works and Comparison

2.1 Experience Replay Techniques

Experience replay involves storing consecutive temporally dependent data in a (large) buffer in a FIFO order. Whenever a learning algorithm queries for batched data, the experience replay algorithm returns a sub-sample from this buffer such that this data does not hinder the learning algorithms due to spurious correlations. The most basic form of experience replay is UER [17] which samples the data in the replay buffer uniformly at random. This approach has shown significant improvements in the performance of off-policy RL algorithms like DQN [19]. Several other methods of sampling from the buffer have been proposed since; PER [23] samples experiences from a probability distribution which assigns higher probability to experiences with large TD error and is shown to boost the convergence speed of the algorithm. This out-performs UER in most Atari environments. HER [2] works with the intuition that no matter how bad the policy is, we can still learn something from it. Like humans, HER works in the "what if" scenario. Even a sub-optimal policy can lead the agent to learn what

not to do and nudge the agent towards the correct action. RER processes the data obtained in a buffer in the reverse temporal order. We refer to the following sub-section for a detailed review of this and related techniques. We will also consider 'optimistic experience replay' (OER), which is the naive version of PER, where at each step, only top B elements in the buffer are returned when batched data is queried. This approach is known to suffer from high bias error which is mitigated by a sophisticated sampling procedure employed in PER.

2.2 Reverse Sweep Techniques

Reverse sweep or backward value iteration refers to methods that process the data as received in reverse order. This has been studied in the context of planning tabular MDPs [7, 11]. We refer to Section 4.1 for a brief overview of why these methods are considered. However, this line of work assumes that the MDP and the transition functions are known. Inspired by the behavior of biological networks, [22] proposed reverse experience replay where the experience replay buffer is replayed in a LIFO order. Since RER forms mini-batches with consecutive data points, it is unstable with Neural approximation. Therefore, the iterations are stabilized by 'mixing' RER with UER. However, the experiments conducted are limited and do not demonstrate that this method outperforms even UER. A similar method called Episodic Backward Update (EBU) is introduced in [15]. However, to stabilize the pure RER , the method seeks to also change the target for Q learning instead of just changing the sampling scheme in the replay buffer. The reverse sweep was independently rediscovered as RER in the context of streaming linear system identification in [14], where SGD with reverse experience replay was shown to achieve near-optimal performance. In contrast, naive SGD was significantly sub-optimal due to bias caused by Markovian data. The follow-up work [1] analyzed off-policy Q learning with linear function approximation and reverse experience replay to provide near-optimal convergence guarantees using the special super martingale structure endowed by reverse experience replay. [12] considers topological experience replay, which executes reverse replay over a directed graph of observed transitions. When mixed with PER, this enables non-trivial learning in some hard environments. Another line of work [8, 20, 10, 24] considers reverse sweep with access to a simulator or using a fitted generative model. Our work only seeks on-policy access to the MDP.

3 Preliminaries

We consider episodic reinforcement learning [27], where at each time step an agent takes actions a_t in an uncertain environment with state s_t, and receives a reward r_t. The environment then evolves into a new state s_{t+1} whose law depends only on s_t, a_t. Our goal is to (approximately) find the policy π^* which maps the environmental state s to an action a such that when the agent takes the action $a_t = \pi^*(s_t)$, the discounted reward $\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]$ is maximized. To achieve this, we consider algorithms like DQN ([19]) and TD3 ([9]) which routinely use experience replay buffers. In this paper, we introduce a new experience replay method, RER++, and investigate the performance of the aforementioned RL algorithms with this modification. More specifically, we seek to investigate the following questions:

- **Does our algorithm help in the convergence towards a better policy?** The performance of most RL algorithms are highly dependent on the type of Replay Buffer algorithm used. We aim to emplore if RER++ learns a superior policy compared to those obtained from other baselines.
- **Does our algorithm lead to any speedup in convergence?** Reinforcement learning algorithms, in general, are very slow to learn and could take days and millions of steps to converge to a locally optimal policy. We aim to study whether the inclusion of RER++ helps speed up the convergence of these algorithms.
- **Is our algorithm robust across environments?** To efficiently support the practical gains from our algorithm, we aim to show that our proposed methodology not only outperforms other baselines but is consistently doing so across various environments with limited hyper-parameter tuning.
- **What influences the performance of our proposed methodology?** We aim to study why our proposed approach does well across different environments. To answer this question, we perform an ablation study between OER, RER and RER++ for the reasons highlighted in

Section 4.1. We also perform an ablation study with FER++ where data points form buffers by processing the forward order from the pivots instead of the reverse order.

4 Our Method

We now describe our main method in a general way where we assume that we have access to a data collection mechanism $\mathbb{T}$ which samples new data points, appends them to a buffer $\mathcal{H}$ and discards some older data points. The goal is to run an iterative learning algorithm $\mathbb{A}$ which learns from batched data of batch size B in every iteration. We also consider an importance metric I associated with the problem. At each step, the data collection mechanism $\mathbb{T}$ collects a new episode and appends it to the buffer $\mathcal{H}$ and discards some old data points, giving us the new buffer as $\mathcal{H} \leftarrow \mathbb{T}(\mathcal{H})$. We then sort the entries of $\mathcal{H}$ based on the importance metric I and store the indices of the top G data points in an array $P = [P[0], \ldots, P[G-1]]$. Then for every index in P, we run the learning algorithm $\mathbb{A}$ with the batch $D = (\mathcal{H}(P[i]), \ldots, \mathcal{H}(P[i] - B + 1))$. We describe this procedure in Algorithm 1.

In the reinforcement learning setting, $\mathbb{T}$ runs an episode of the environment with the current policy and appends the transitions and corresponding rewards to the buffer $\mathcal{H}$ in the FIFO order, maintaining a total of $1E6$ data points. We choose $\mathbb{A}$ to be an RL algorithm like TD3 or DQN. The importance function I is the magnitude of the TD error with respect to the current Q-value estimate provided by the algorithm $\mathbb{A}$. When the data collection mechanism ($\mathbb{T}$) is the same as in UER, we will call this method RER++. In optimistic experience replay (OER), we take $\mathbb{T}$ to be the same as in UER. However, we query top BG data points from the buffer $\mathcal{H}$ and return G disjoint batches each of size B from these 'important' points. It is clear that RER++ is a combination of OER and RER. Notice that, we can also consider the data collection mechanism like that of HER, where examples are also labeled with different goals. In this case, we will call our algorithm HRER++. An example of this successful coalition is depicted in our experiment in Enduro and Acrobot. We also consider the RER method, which served as a motivation for our proposed approach. Under this sampling methodology, the batches are drawn from $\mathcal{H}$ in the temporally reverse direction. This approach is explored in the works mentioned in Section 2.2. In our implementation, we set the replay buffer sampling procedure to reset upon adding a new batch of episodes to the buffer. We discuss this methodology in more detail in Appendix E.

Input: Data collection mechanism $\mathbb{T}$, Data buffer $\mathcal{H}$, Batch size B, grad steps per Epoch G, number of episodes N, Importance function I, learning procedure $\mathbb{A}$

$n \leftarrow N$;
while $n < N$ **do**
  $n \leftarrow n + 1$;
  $\mathcal{H} \leftarrow \mathbb{T}(\mathcal{H})$; `// Add a new episode to the buffer`
  $\mathcal{I} \leftarrow I(\mathcal{H})$; `// Compute importance of each data point in the buffer`
  $P \leftarrow \mathsf{Top}(\mathcal{I}; G)$; `// Obtain index of top` G `elements of` $\mathcal{I}$
  $g \leftarrow 0$;
  while $g < G$ **do**
    $D \leftarrow \mathcal{H}[P[g] - B, P[g]]$; `// Load batch of previous` B `examples from` $P[g]$
    $g \leftarrow g + 1$;
    $\mathbb{A}(D)$; `// Run the learning algorithm with batch data` D
  end
end

Algorithm 1: Optimistic Reverse Experience Replay (Parent Method for RER++ and HRER++)

4.1 Understanding Reverse Replay

There are various conceptual ways we can look at RER and RER++. This section outlines some of the motivations behind using this technique. We also refer to Appendix C where we show via an ablation study that going forward in time instead of reverse does not work very well.

Super Martingale Structure in Descent Algorithms: As was noted in [14] in the linear system identification setting, processing Markov process data in the order they are received through SGD-like

algorithms can lead to biases that cause sub-optimal performance. However, simply processing the received data in reverse order with RER can remove these biases by leveraging certain super-martingale structures. These observations were later extended to the non-linear and RL settings in [13, 1].

Dynamic Programming: Dynamic programming algorithms are some of the first computationally efficient algorithms established for RL [4]. Here again, the optimal policy is constructed by going backward in time to avoid exponential dependence on the horizon incurred by the brute force algorithm. Indeed, this is the idea behind reverse sweep or backward value iteration in [7, 11], which mainly deals with planning when the MDP itself is fully known.

Causality: An intuitive explanation, which is relevant to this particular work, is obtained through the lens of causality. MDPs have a natural causal structure: actions/ events in the past influence the events in the future. Therefore, whenever we see a surprising or unexpected event, we can understand why or how it happened by looking into the past. However, we concede that learning from trajectories by conditioning on the end state (i.e., 'surprising' states) is not very theoretically principled since this might lead to biases. Further theoretical work is needed to understand RER++ better.

Bias Reduction in OER: OER, which greedily chooses the examples with largest TD error to learn from, performs very poorly due to bias. In simple settings, it has been shown by prior theoretical works that RER prevents bias in Q-learning type algorithms. Therefore, one possible way of viewing RER++ is that RER is used to reduce the bias in OER. PER achieves a similar bias reduction with respect to OER with a sophisticated and expensive sampling scheme over the buffer.

Propogation of Sparse Rewards: In many RL problems, non-trivial rewards are sparse and only received at the goal states. Therefore, processing the data backward in time from such goal states helps the algorithm learn about the states that led to this non-trivial reward. In fact, our study in Section D (see Figure 8) shows that in many environments RER++ picks pivots which are the goals states with large (positive or negative) rewards, enabling effective learning.

5 Experimental Results

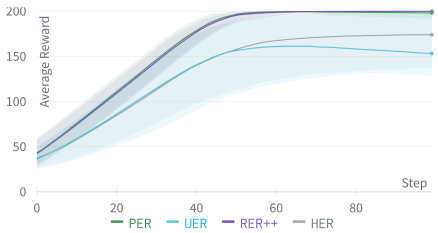
We evaluate our approach on a diverse class of environments, such as (i) environments with low-dimensional state space (including classic control and Box-2D environments), (ii) multiple joint dynamic simulation and robotics environments (including Mujoco and Robotics environments), and (iii) human-challenging environments (such as Atari environments). We refer to Appendix A for a brief description of the environments used and Appendix B for the exact hyperparameters used. We use the *top-k average reward* as the evaluation metric across all our runs. To factor in the seed sensitivity of runs, we have taken an average of 3 to 5 different seeds. Table 5 depicts our results in various environments upon using different replay sampler mechanisms. Our proposed sampler outperforms all other baselines in most tasks and compares favorably in others. Furthermore, it is also important to highlight that our approach is approximately 10x faster than PER on more straightforward experiments (Cartpole environment using CPU, to be precise) and is roughly similar in runtime to PER on more complicated environments. Across all the environments other than FetchReach, we use a running average of 50 timesteps for robust estimation and visualization. This decision was taken since we were running for a significantly lower number of epochs when compared to other environments. Instead, we use a window size of 20. We argue that this is important since, usually, pure noise can be leveraged to pick a time instant where a given method performs best.

Table 1: *Top-k seeds Moving Average Return* results across various environments. We use ‡ symbol to depict the usage of HRER++, a variant of RER++ further discussed in Section 6.

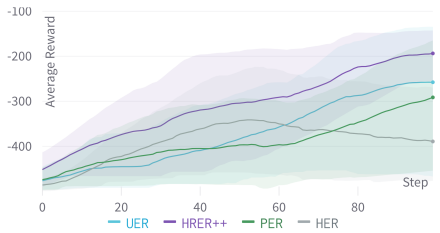
Dataset	*UER*	*PER*	*HER*	*RER*	*OER*	*RER++*
CartPole [3]	153.135 ± 32.821	198.062 ± 3.677	173.838 ± 26.109	118.25 ± 60.021	162.356 ± 34.89	**199.829** ± 0.3057
Acrobot [26]	-257.927 ± 184.28	-291.562 ± 148.836	-389.419 ± 113.221	-331.023 ± 129.544	-472.629 ± 31.211	**-193.895** ± 57.559 ‡
LunarLander [6]	-4.424 ± 20.058	5.334 ± 16.097	6.002 ± 10.017	3.613 ± 9.804	-16.083 ± 15.376	**12.32** ± 27.553
HalfCheetah [28]	10808.925 ± 1094.324	99.747 ± 1124.462	**11072.535** ± 297.124	-304.53 ± 199.313	2237.913 ± 2824.715	10104.388 ± 927.391
Ant [28]	3932.846 ± 1024.861	-2699.844 ± 1.339	3803.165 ± 996.813	-2148.298 ± 954.914	-47.93 ± 20.465	**4203.208** ± 345.223
Inverted Double Pendulum [28]	8489.536 ± 927.688	7163.767 ± 3404.136	9002.053 ± 464.204	168.796 ± 110.822	7724.993 ± 1726.577	**9067.686** ± 402.392
Fetch-Reach [2]	-1.972 ± 0.1888	-49.898 ± 0.099	-2.928 ± 1.792	-49.94 ± 0.072	-47.722 ± 3.333	**-1.74** ± 0.244
Pong [18]	**19.145** ± 1.317	17.024 ± 3.272	18.696 ± 1.018	11.692 ± 9.195	3.517 ± 21.331	19.095 ± 1.195
Enduro [18]	242.422 ± 342.05	**627.587** ± 205.251	484.355 ± 120.997	86.959 ± 66.538	339.639 ± 128.622	601.187 ± 203.36 ‡

5.1 Performance of RER++ with Low-Dimensional State Space

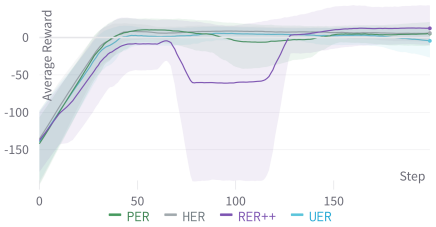
This section briefly discusses our results on environments with a low-dimensional state space, such as classic control environments (CartPole and Acrobot) and Box-2D environments (LunarLander). Fig 2 depicts the learning curves of our DQN agents in these environments.



(a) CartPole-v0



(b) Acrobot-v1



(c) LunarLander-v2

Figure 2: Learning curves of DQN agents on Classic Control and Box-2D environments.

We note that our proposed methodology can significantly outperform other baselines for the classic control algorithms. Furthermore, Fig 2(a) shows excellent promise as we achieve a near-perfect score across all seeds in one-tenth of the time it took to train PER.

5.2 Performance in Multiple Joint Dynamics Simulation and Robotics Environments

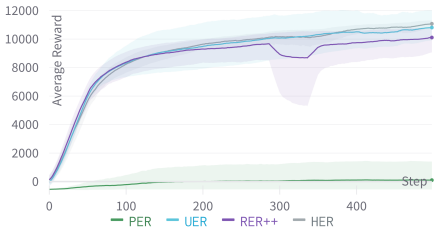
Multiple joint dynamic simulation environments (mujoco physics environments) and robotics environments such as HalfCheetah, Ant, Inverted Double-Pendulum, and FetchReach ([28, 21]) are more complex and enable us to study whether the agent can understand the physical phenomenon of real-world environments. Fig 3 depicts the learning curves of our TD3 agents in these environments. Again, our proposed methodology outperforms all other baselines significantly in most of the environments studied in this section. Furthermore, it might be possible for RER++ to outperform all other baselines in the HalfCheetah environment if we fine-tune some of the hyperparameters such as the number of pivots sampled, using HRER++, etc. Additionally, it is important to point out that our proposed methodology shows an impressive speedup of convergence in InvertedDoublePendulum and convergence to a much better policy in Ant.

Our proposed methodology can perform favorably compared to other baselines for the FetchReach environment.

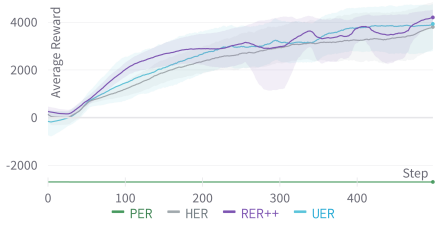
5.3 Perfomance of RER++ in Human Challenging Environments

This section briefly discusses our results on human-challenging environments such as Atari environments (Pong and Enduro). These environments are highly complex, and our algorithms take millions of steps to converge to a locally optimal policy. Fig 4 depicts the learning curves of our DQN agents in these environments.

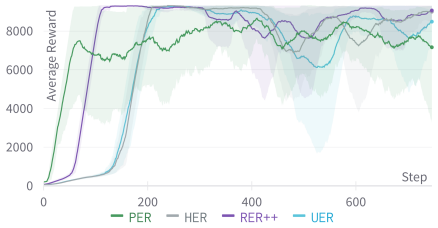
We note that our proposed methodology can perform favorably when compared to other baselines for the Atari environments and can reach large reward policies significantly faster than UER. The training curve for Enduro shows high variability, as depicted above, and the reported accuracy and relative



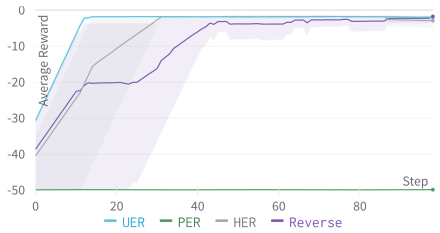
(a) HalfCheetah-v2



(b) Ant-v2

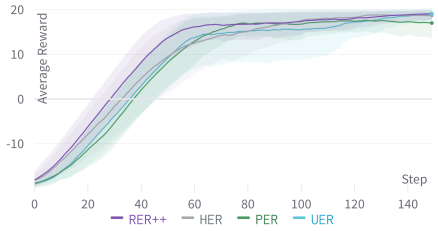


(c) InvertedDoublePendulum-v2

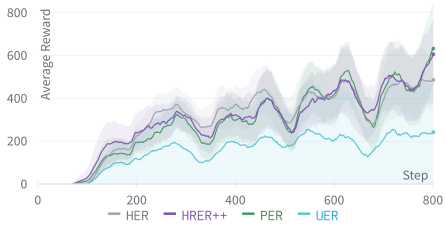


(d) FetchReach-v1

Figure 3: Learning curves of TD3 agents on Mujoco and Robotics environments.



(a) Pong-v0



(b) Enduro-v0

Figure 4: Learning curves of DQN agents on Atari environments.

performance of the algorithms could also be artifacts of the number of epochs chosen. However, the experiment does allow us to infer that RER++ outperforms UER reliably.

6 Discussion and Conclusions

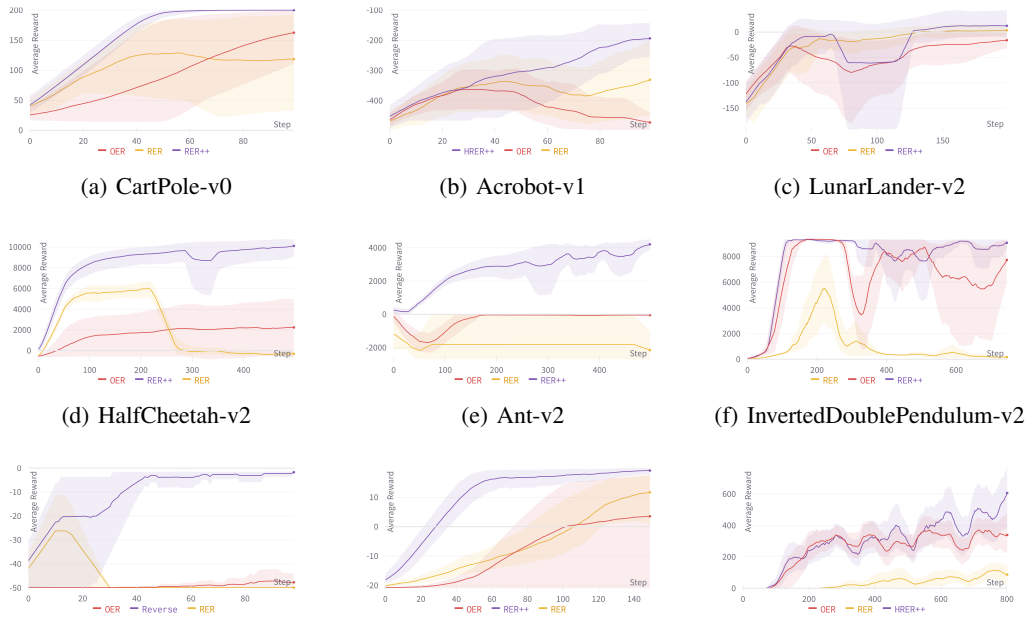
This section summarizes some of the exciting properties of RER++ discovered from our series of experiments in this paper. We also discuss some possible future steps.

Converges to better policy. Our experiments on various environments across various classes, such as classic control, Atari, etc., showed that our proposed methodology consistently outperforms all other baselines in most environments. Hence, the intuition of looking back when surprised does help the agents to learn better and perform very well on their respective tasks. Furthermore, our proposed methodology is robust across various environments, as highlighted in the previous section.

Speedup in convergence to the optimal policy. We note a speedup of convergence towards an optimal policy of our proposed approach as shown in CartPole, InvertedDoublePendulum, and Pong. Furthermore, even if we notice a similar speedup in some cases of PER, the number of CPU hours (in the case of CartPole) used is almost ten times more. To conclude, the very core of RER++ is in line with our intuition that looking back when surprised does teach the model a lot during the initial few steps of the learning. Furthermore, the lack of the speedup in some of the other experiments (even if

they do show an overall improvement in performance) could be due to the fact that the "surprised" pivot cannot be successfully utilized to teach the agent rapidly.

Ablation Study Intuitively, our approach is an efficient amalgamation of OER and RER. Here we compare these individual parts with RER++. Figure 5 depicts the ablation study of our proposed approach against its intuitive components such as OER and RER. Clearly, neither OER nor RER perform well compared to their amalgamation, RER++.



(a) CartPole-v0 (b) Acrobot-v1 (c) LunarLander-v2

(d) HalfCheetah-v2 (e) Ant-v2 (f) InvertedDoublePendulum-v2

(g) FetchReach-v1 (h) Pong-v0 (i) Enduro-v0

Figure 5: Ablation study of OER, RER and RER++.

From Figure 5, we note that RER and OER are not very competitive and do not farewell. However, the amalgamation of the intuitions from the two concepts fits very well and results in a highly competitive methodology across many environments.

Issues with stability and consistency Picking pivot points by looking at the TD error might cause more biases and instability compared to UER as seen in some environments like HalfCheetah, LunarLander, and Ant, where there is a sudden drop in performance for some episodes. We observe that our strategy RER++ corrects itself quickly, unlike RER, which cannot do this. We believe that increasing the number of pivot points usually helps mitigate this, and so does picking some pivots at random. However, these steps might slow down the initial convergence to a locally optimal policy. It would be interesting to explore gradually mixing RER++ with some form of uniformly random pivot picking strategy in order to obtain the best of both worlds. From additional experiments (see Figure 6), we show that the self-correcting instability concerns, although rare, could be mitigated by infusing random sampling of pivots or batches with RER++. Here, although the RER++ agent seems to learn beneficial properties towards the end of the training, the phenomenon is highly seed-dependent and not dependable for robust learning. In this experiment, we use a sampler called URER++, where we use RER++ with a probability of 70% and UER with a probability of 30% while sampling each buffer. This intuition has been previously explored by [22], although their proposed approach involved combining UER with RER instead of RER++. Even with a simple ad-hoc solution, as mentioned earlier, we could successfully mitigate the instability concerns of RER++. Figure 6 depicts the learning curve in the increased gradient steps setting for the Hopper environment. Table 2 summarizes our findings as a moving average metric with a window size of 50.

Why does RER++ outperform the traditional RER? The instability of pure RER has been noted in various works, including [22] and [15], where RER is stabilized by mixing it with UER .

Table 2: *Top-k seeds Moving Average Return* results on Hopper environment.

Sampler	*Average Reward*
UER	2897.582 ± 559.019
RER++	697.288 ± 1101.464
URER++	**3272.451** ± 232.162

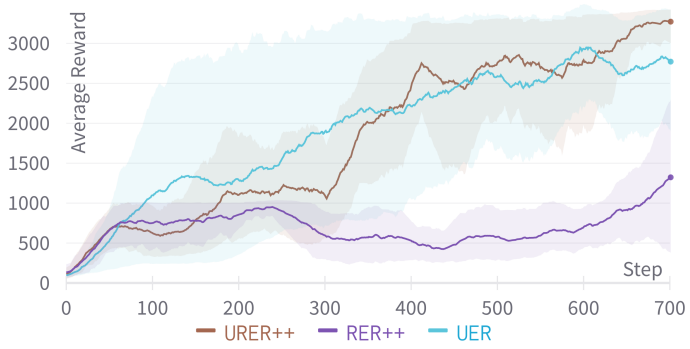


Figure 6: Further stabilizing RER++. The large plateau observed in RER++ suggests a significant bias that hampers our model from learning useful skills. Instead, a simple ad-hoc solution of using UER with RER++, which we call URER++ is able to mitigate this concern to a great extent.

[12] stabilizes reverse sweep by mixing it with PER. It is not entirely clear why RER++ outperforms the traditional RER with using neural function approximation, while RER is nearly optimal in the tabular setting [1]. Naively going in the reverse order inside a buffer might be sub-optimal due to the highly non-convex nature of the neural network loss landscape. It is also computationally intractable to pass through the entire buffer for neural networks.

Why backward and not forwards? There is an intuitive limitation to the "looking forward" approach. In many RL problems, the objective for the agent is to reach a final goal state, where the non-trivial reward is obtained. Since non-trivial rewards are only offered in this goal state, it is informative to look back from here in order to learn about the states that *lead* to this. In the scenario where the goals are sparse, the TD error is more likely to be large upon reaching the goal state. Therefore, our algorithm selects these as pivots. Our studies on many environments (see Appendix D) show that the pivot points selected based on importance indeed have large (positive or negative) rewards.

Reproducability Statement

In this paper, we work with ten datasets, all of which are open-sourced in gym (`https://github.com/openai/gym`). More information about the environments is available in Appendix A. We predominantly use DQN and TD3 algorithms in our research, both of which have been adapted from their open-source code. We also experimented with seven different replay buffer methodologies, all of which have been adapted from their source code. Most of these resources are publicly available at their open-source code[2]. More details about the models and hyperparameters are described in Appendix B. All runs have been run using the A100-SXM4-40GB, TITAN RTX, and V100 GPUs.

References

[1] Naman Agarwal, Syomantak Chaudhuri, Prateek Jain, Dheeraj Nagaraj, and Praneeth Netrapalli. Online target q-learning with reverse experience replay: Efficiently finding the optimal policy for linear mdps. *arXiv preprint arXiv:2110.08440*, 2021.

[2] `https://github.com/rlworkgroup/garage`

[2] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *Advances in neural information processing systems*, 30, 2017.

[3] Andrew G Barto, Richard S Sutton, and Charles W Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE transactions on systems, man, and cybernetics*, (5):834–846, 1983.

[4] Dimitri P Bertsekas and John N Tsitsiklis. Neuro-dynamic programming: an overview. In *Proceedings of 1995 34th IEEE conference on decision and control*, volume 1, pages 560–564. IEEE, 1995.

[5] Guy Bresler and Dheeraj Nagaraj. Least squares regression with markovian data: Fundamental limits and algorithms. *Advances in neural information processing systems*, 33, 2020.

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

[7] Peng Dai and Eric A Hansen. Prioritizing bellman backups without a priority queue. In *ICAPS*, pages 113–119, 2007.

[8] Carlos Florensa, David Held, Markus Wulfmeier, Michael Zhang, and Pieter Abbeel. Reverse curriculum generation for reinforcement learning. In *Conference on robot learning*, pages 482–495. PMLR, 2017.

[9] Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International conference on machine learning*, pages 1587–1596. PMLR, 2018.

[10] Anirudh Goyal, Philemon Brakel, William Fedus, Soumye Singhal, Timothy Lillicrap, Sergey Levine, Hugo Larochelle, and Yoshua Bengio. Recall traces: Backtracking models for efficient reinforcement learning. *arXiv preprint arXiv:1804.00379*, 2018.

[11] Marek Grześ and Jesse Hoey. On the convergence of techniques that improve value iteration. In *The 2013 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2013.

[12] Zhang-Wei Hong, Tao Chen, Yen-Chen Lin, Joni Pajarinen, and Pulkit Agrawal. Topological experience replay. *arXiv preprint arXiv:2203.15845*, 2022.

[13] Suhas Kowshik, Dheeraj Nagaraj, Prateek Jain, and Praneeth Netrapalli. Near-optimal offline and streaming algorithms for learning non-linear dynamical systems. *Advances in Neural Information Processing Systems*, 34, 2021.

[14] Suhas Kowshik, Dheeraj Nagaraj, Prateek Jain, and Praneeth Netrapalli. Streaming linear system identification with reverse experience replay. *Advances in Neural Information Processing Systems*, 34, 2021.

[15] Su Young Lee, Choi Sungik, and Sae-Young Chung. Sample-efficient deep reinforcement learning via episodic backward update. *Advances in Neural Information Processing Systems*, 32, 2019.

[16] 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. *arXiv preprint arXiv:1509.02971*, 2015.

[17] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3):293–321, 1992.

[18] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[19] 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*, 518(7540):529–533, 2015.

[20] Andrew W Moore and Christopher G Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine learning*, 13(1):103–130, 1993.

[21] Matthias Plappert, Marcin Andrychowicz, Alex Ray, Bob McGrew, Bowen Baker, Glenn Powell, Jonas Schneider, Josh Tobin, Maciek Chociej, Peter Welinder, et al. Multi-goal reinforcement learning: Challenging robotics environments and request for research. *arXiv preprint arXiv:1802.09464*, 2018.

[22] Egor Rotinov. Reverse experience replay. *arXiv preprint arXiv:1910.08780*, 2019.

[23] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[24] Yannick Schroecker, Mel Vecerik, and Jonathan Scholz. Generative predecessor models for sample-efficient imitation learning. *arXiv preprint arXiv:1904.01139*, 2019.

[25] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[26] Richard S Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. *Advances in neural information processing systems*, 8, 1995.

[27] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[28] 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.

[29] Paweł Wawrzyński. A cat-like robot real-time learning to run. In *International Conference on Adaptive and Natural Computing Algorithms*, pages 380–390. Springer, 2009.

A Environments

For all OpenAI environments, data is summarized from https://github.com/openai/gym, and more information is provided in the wiki https://github.com/openai/gym/wiki. Below we briefly describe some of the tasks we experimented on in this paper.

A.1 CartPole-v0

CartPole, as introduced in [3], is a task of balancing a pole on top of the cart. The cart has access to the position and velocity as its state vector. Furthermore, it can go either left or right for each action. The task is over when the agent achieves 200 timesteps without a positive reward (balancing the pole) which is the goal state, or has failed, either when (i) the cart goes out of boundaries ($\pm$ 2.4 units off the center), or (ii) the pole falls over (less than $\pm$ 12 deg). The agent is given a continuous 4-dimensional space describing the environment and can respond by returning one of two values, pushing the cart either right or left.

A.2 Acrobot-v1

Acrobot, as introduced in [26], is a task where the agent is given rewards for swinging a double-jointed pendulum up from a stationary position. The agent can actuate the second joint by one of three actions: left, right, or no torque. The agent is given a six-dimensional vector comprising the environment's angles and velocities. The episode terminates when the end of the second pole is over the base. Each timestep that the agent does not reach this state gives a -1 reward, and the episode length is 500 timesteps.

A.3 LunarLander-v2

The LunarLander environment introduced in [6] is a classic rocket trajectory optimization problem. The environment has four discrete actions - do nothing, fire the left orientation engine, fire the right orientation engine, and fire the main engine. This scenario is per Pontryagin's maximum principle, as it is optimal to fire the engine at full throttle or turn it off. The landing coordinates (goal) is always at $(0, 0)$. The coordinates are the first two numbers in the state vector. There are a total of 8 features in the state vector. The episode terminates if (i) the lander crashes, (ii) the lander gets outside the window, or (iii) the lander does not move nor collide with any other body.

A.4 HalfCheetah-v2

HalfCheetah is an environment based on the work by [29] adapted by [28]. The HalfCheetah is a 2-dimensional robot with nine links and eight joints connecting them (including two paws). The goal is to apply torque on the joints to make the cheetah run forward (right) as fast as possible, with a positive reward allocated based on the distance moved forward and a negative reward allocated for moving backward. The torso and head of the cheetah are fixed, and the torque can only be applied to the other six joints over the front and back thighs (connecting to the torso), shins (connecting to the thighs), and feet (connecting to the shins). The reward obtained by the agent is calculated as follows:

$$r_t = \dot{x_t} - 0.1 * \|a_t\|_2^2$$

A.5 Ant-v2

Ant is an environment based on the work by [25] and adapted by [28]. The ant is a 3D robot with one torso, a free rotational body, and four legs. The task is to coordinate the four legs to move in the forward direction by applying torques on the eight hinges connecting the two links of each leg and the torso. Observations consist of positional values of different body parts of the ant, followed by the velocities of those individual parts (their derivatives) with all the positions ordered before all the velocities. The reward obtained by the agent is calculated as follows:

$$r_t = \dot{x_t} - 0.5 * \|a_t\|_2^2 - 0.0005 * \left\|s_t^{\text{contact}}\right\|_2^2 + 1$$

A.6 Hopper-v2

The Hopper environment, as introduced in [28] sets out to increase the number of independent state and control variables compared to classic control environments. The hopper is a two-dimensional figure with one leg which consists of four main body parts - the torso at the top, the thigh in the middle, the leg at the bottom, and a single foot on which the entire body rests. The goal of the environment is to make hops that move in the forward (right) direction by applying torques on the three hinges connecting the body parts. The action space is a three-dimensional element vector. The state-space consists of positional values for different body parts followed by the velocity states of individual parts.

A.7 Inverted Double-Pendulum-v2

Inverted Double-Pendulum as introduced in [28] is built upon the CartPole environment as introduced in [3], with the infusion of Mujoco. This environment involves a cart that can be moved linearly, with a pole fixed on it and a second pole on the other end of the first one (leaving the second pole as the only one with one free end). The cart can be pushed either left or right. The goal is to balance the second pole on top of the first pole, which is on top of the cart, by applying continuous forces on the cart. The agent takes a one-dimensional continuous action space in the range [-1,1], denoting the force applied to the cart and the sign depicting the direction of the force. The state-space consists of positional values of different body parts of the pendulum system, followed by the velocities of those individual parts (their derivatives) with all the positions ordered before all the velocities. The goal is to balance the double-inverted pendulum on the cart while maximizing its height off the ground and having minimum disturbance in its velocity.

A.8 FetchReach-v1

The FetchReach environment introduced in [21] was released as part of *OpenAI Gym* and used the Mujoco physics engine for fast and accurate simulation. The goal is 3-dimensional and describes the desired position of the object. Rewards in this environment are sparse and binary. The agent obtains a reward of 0 if the target location is at the target location (within a tolerance of 5 cm) and -1 otherwise. Actions are four-dimensional, where 3 specifies desired gripper movement, and the last dimension controls the opening and closing of the gripper. The goal of the FetchReach is to move the gripper to a target position.

A.9 Pong-v0

Pong, also introduced in [18] is comparatively more accessible than other Atari games such as Enduro. Pong is a two-dimensional sports game that simulates table tennis. The player controls an in-game paddle by moving vertically across the left and right sides of the screen. Players use this paddle to hit the ball back and forth. The goal is for each player to reach eleven points before the opponent, where the point is earned for each time the agent returns the ball and the opponent misses.

A.10 Enduro-v0

Enduro, introduced in [18] is a hard environment involving maneuvering a race car in the National Enduro, a long-distance endurance race. The goal of the race is to pass a certain number of cars each day. The agent must pass 200 cars on the first day and 300 cars on all subsequent days. Furthermore, as time passes, the visibility changes as well. At night in the game, the player can only see the oncoming cars' taillights. As the days progress, cars will become more challenging to avoid. Weather and time of day are factors in how to play. During the day, the player may drive through an icy patch on the road, which would limit control of the vehicle, or a patch of fog may reduce visibility.

B Model and Hyperparameters

In this paper, we work with two classes of algorithms: DQN and TD3. The hyperparameters used for training our DQN algorithms in various environments are described in Table 3. Furthermore, the hyperparameters used for training TD3 are described in Table 4.

Description	CartPole	Acrobot	LunarLander	Atari	argument_name
General Settings					
Discount	0.9	0.9	0.9	0.99	discount
Batch size	512	512	512	32	batch_size
Number of epochs	100	100	200	150	n_epochs
Steps per epochs	10	10	10	20	steps_per_epoch
Number of train steps	500	500	500	125	num_train_steps
Target update frequency	30	30	10	2	target_update_frequency
Replay Buffer size	$1e^6$	$1e^6$	$1e^6$	$1e^4$	buffer_size
Algorithm Settings					
CNN Policy Channels	-	-	-	$(32, 64, 64)$	cnn_channel
CNN Policy Kernels	-	-	-	$(8, 4, 3)$	cnn_kernel
CNN Policy Strides	-	-	-	$(4, 2, 1)$	cnn_stride
Policy hidden sizes (MLP)	$(8, 5)$	$(8, 5)$	$(8, 5)$	$(512,)$	pol_hidden_sizes
Buffer batch size	64	128	128	32	batch_size
Exploration Settings					
Max epsilon	1.0	1.0	1.0	1.0	max_epsilon
Min epsilon	0.01	0.1	0.1	0.01	min_epsilon
Decay ratio	0.4	0.4	0.12	0.1	decay_ratio
Optimizer Settings					
Learning rate	$5e^{-5}$	$5e^{-5}$	$5e^{-5}$	$1e^{-4}$	lr

Table 3: Hyperparameters used for training DQN on various environments.

Description	Ant	Double-Pendulum	HalfCheetah	Hopper	FetchReach	argument_name
General Settings						
Discount	0.99	0.99	0.99	0.99	0.95	discount
Batch size	250	100	100	100	256	batch_size
Number of epochs	500	750	500	700	100	n_epochs
Steps per epochs	40	40	20	40	50	steps_per_epoch
Number of train steps	50	1	50	100	100	num_train_steps
Replay Buffer size	$1e^6$	$1e^6$	$1e^6$	$1e^6$	$1e^6$	buffer_size
Algorithm Settings						
Policy hidden sizes (MLP)	$(256, 256)$	$(256, 256)$	$(256, 256)$	$(256, 256)$	$(256, 256)$	pol_hidden_sizes
Policy noise clip	0.5	0.5	0.5	0.5	0.5	pol_noise_clip
Policy noise	0.2	0.2	0.2	0.2	0.2	pol_noise
Target update tau	0.005	0.005	0.005	0.005	0.01	tau
Buffer batch size	100	100	100	100	256	batch_size
Gaussian noise Exploration Settings						
Max sigma	0.1	0.1	0.1	0.1	0.1	max_sigma
Min sigma	0.1	0.1	0.1	0.1	0.1	min_sigma
Optimizer Settings						
Policy Learning rate	$1e^{-3}$	$3e^{-4}$	$1e^{-3}$	$3e^{-4}$	$1e^{-3}$	pol_lr
QF Learning rate	$1e^{-3}$	$1e^{-3}$	$1e^{-3}$	$1e^{-3}$	$1e^{-3}$	qf_lr

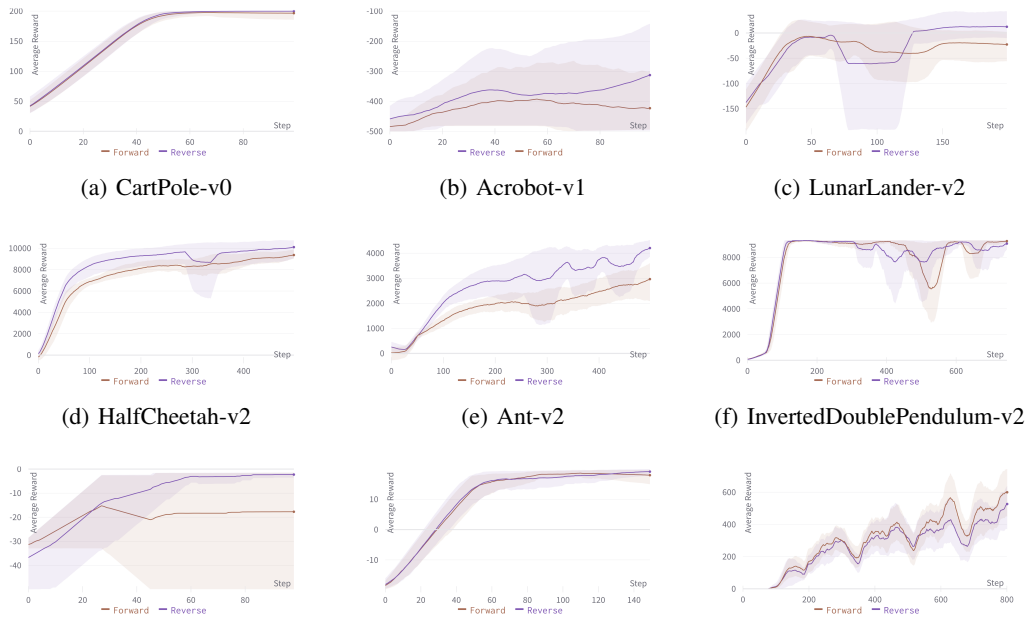
Table 4: Hyperparameters used for training TD3 on various environments.

C Ablation study of Temporal effects

This section studies the ablation effects of going temporally forward and backward once we choose a pivot/surprise point. Table 5 describes the *Top-k seeds Moving Average Return* across various environments in this domain. Furthermore, Figure 7 depicts the learning curves of the two proposed methodologies. We notice that against theoretical intuition, the unbiased forward sampling scheme is worse in most environments when compared to the reverse sampling scheme.

Table 5: *Top-k seeds Moving Average Return* results across various environments for Temporal Ablation study.

Dataset	*Forward*	*Reverse*
CartPole [3]	196.51 ± 6.256	**199.829** ± 0.3057
Acrobot [26]	-423.154 ± 108.079	**-313.031** ± 190.27
LunarLander [6]	-22.754 ± 30.186	**12.32** ± 27.553
HalfCheetah [28]	9369.303 ± 454.403	**10108.153** ± 919.27
Ant [28]	2963.712 ± 828.496	**4203.208** ± 345.223
Inverted Double Pendulum [28]	**9260.272** ± 95.589	9067.686 ± 402.392
FetchReach [2]	-17.732 ± 27.907	**-2.283** ± 1.113
Pong [18]	17.921 ± 2.603	**19.095** ± 1.195
Enduro [18]	**600.873** ± 149.988	525.837 ± 146.388



(a) CartPole-v0 (b) Acrobot-v1 (c) LunarLander-v2

(d) HalfCheetah-v2 (e) Ant-v2 (f) InvertedDoublePendulum-v2

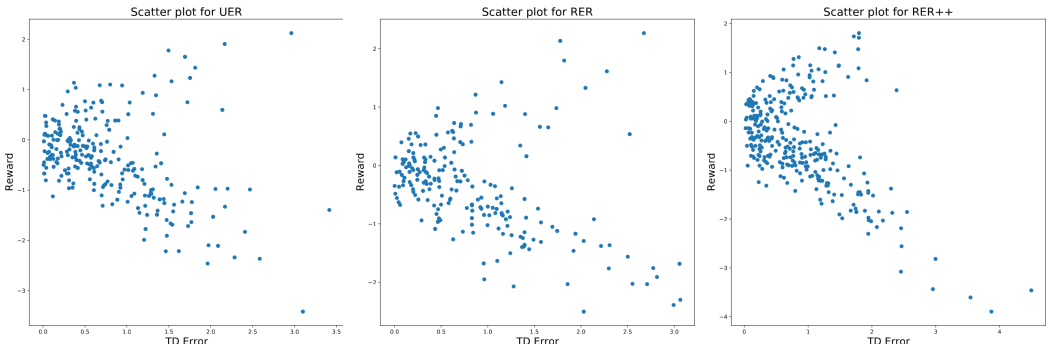
(g) FetchReach-v1 (h) Pong-v0 (i) Enduro-v0

Figure 7: Ablation study of the effects of the temporal structure on the performance of the agent.

D Sparsity and Rewards of Surprising States

D.1 Surprising States Have Large Rewards

In this section, we study the "learning from sparse reward" intuition provided in Section 4.1 – i.e, we want to check if the states corresponding to large TD error correspond to states with large (positive or negative) rewards. To test the hypothesis, we consider a sampled buffer and plot the TD error of these points in the buffer against the respective reward. Figure 8 shows the distribution of TD error against reward for the sampled buffers in the Ant environment. We see that high reward states (positive or negative) also have higher TD error. Therefore, our algorithm, in such environments, picks large reward states as end points to learn.



(a) UER (b) RER (c) RER++

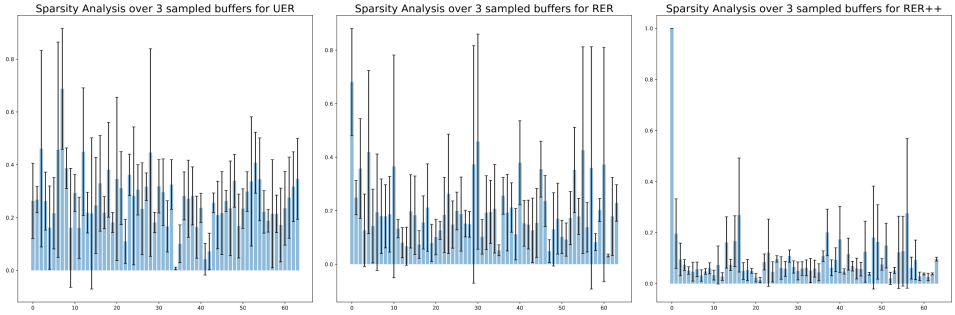
Figure 8: Relationship between TD Error (Surprise factor) and Reward for the Ant environment.

D.2 Surprising States are Sparse and Isolated

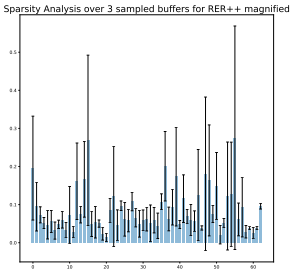
Figure 9 and Figure 10 depict the distribution of "surprise"/TD error in a sampled batch for CartPole and Ant environments respectively. These two figures help show that the states with a large "surprise"

factor are few in number, and that even though the pivot of a buffer has a large TD error, the rest of the buffer typically does not.

Figure 9(d) and Figure 10(d) show a magnified view of Figure 9(c) and Figure 10(d) where the pivot point selected is dropped. This helps with a uniform comparison with the remaining timesteps within the sampled buffer. Again, we notice little correlation between the timesteps within the sampled buffer.



(a) UER (b) RER (c) RER++

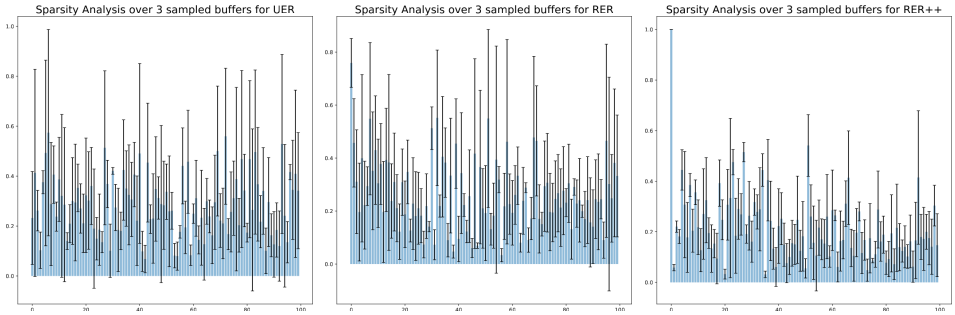


(d) RER++ Magnified

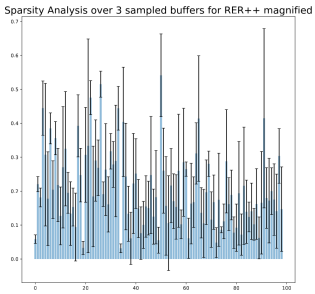
Figure 9: Normalized TD Error ("Surprise factor") of each timesteps over 3 different sampled buffers on the CartPole environment. Best viewed when zoomed.

E Reverse Experience Replay (RER)

This section discusses our implementation of Reverse Experience Replay (RER), which served as a motivation for our proposed approach. The summary of the RER approach is shown in Figure 11. Furthermore, an overview of our implemented approach to RER is described briefly in Algorithm 2.



(a) UER (b) RER (c) RER++



(d) RER++ Magnified

Figure 10: Normalized TD Error ("Surprise factor") of each timesteps over 3 different sampled buffers on the Ant environment. Best viewed when zoomed.

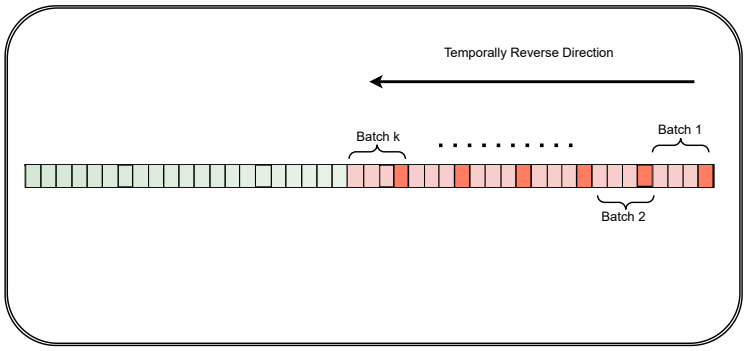


Figure 11: An illustration of Reverse Experience Replay (RER) when selecting k batches from the Replay Buffer.

Input: Data collection mechanism $\mathbb{T}$, Data buffer $\mathcal{H}$, Batch size B, grad steps per Epoch G, number of episodes N, learning procedure $\mathbb{A}$

$n \leftarrow N$;
while $n < N$ **do**
  $n \leftarrow n + 1$;
  $\mathcal{H} \leftarrow \mathbb{T}(\mathcal{H})$; `// Add a new episode to the buffer`
  $P \leftarrow \mathsf{len}(\mathcal{H})$; `// Set index to last element of Buffer` $\mathcal{H}$
  $g \leftarrow 0$;
  while $g < G$ **do**
    $D \leftarrow \mathcal{H}[P - B, P]$; `// Load batch of previous` B `samples from index` P
    $g \leftarrow g + 1$;
    $P \leftarrow P - B$;
    $\mathbb{A}(D)$; `// Run the learning algorithm with batch data` D
  end
end

Algorithm 2: Reverse Experience Replay