Investigating the Interplay of Prioritized Replay and Generalization

by

Parham Mohammad Panahi

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

Experience replay, the reuse of past data to improve sample efficiency, is ubiquitous in reinforcement learning. Though a variety of smart sampling schemes have been introduced to improve performance, uniform sampling by far remains the most common approach. One exception is Prioritized Experience Replay (PER), where sampling is done proportionally to TD errors, inspired by the success of prioritized sweeping in dynamic programming. The original work on PER showed improvements in Atari, but follow-up results were mixed. In this thesis, we investigate several variations on PER, to attempt to understand where and when PER may be useful. Our findings in prediction tasks reveal that while PER can improve value propagation in tabular settings, behavior is significantly different when combined with neural networks. Certain mitigations—like delaying target network updates to control generalization and using estimates of expected TD errors in PER to avoid chasing stochasticity—can avoid large spikes in error with PER and neural networks but generally do not outperform uniform replay. In control tasks, none of the prioritized variants consistently outperform uniform replay with neural networks; however, recomputing priorities and sampling without replacement improves the performance of PER over uniform replay in tabular settings. This thesis sheds light on the interaction between prioritization, bootstrapping, and neural networks and proposes several improvements for PER in tabular settings and noisy domains.

Preface

Most of the content in this thesis is based on a paper published at the Reinforcement Learning Conference (Mohammad Panahi et al., 2024) co-authored with Andrew Patterson, Martha White and Adam White.

To my Mom, Dad and Sister
For their love and support.

If nothing we do matters, then all that matters is what we do.

Because, if there is no bigger meaning, then the smallest act of kindness is the greatest thing in the world.

– *Epiphany*, Angel, 2001.

Acknowledgements

First, I would like to thank my advisor, Adam White, for being kind and patient. His careful guidance has shaped my understanding of research and academia. He showed me how to ask the right questions and run good experiments.

I am also indebted to Martha White and Andrew Patterson for their generous guidance and feedback. From Martha, I learned to be fearless in research. I enjoyed research meetings and brainstorming with her. She is also a LaTeX goddess. From Andy, I learned to be authentic and optimistic. He is both a great scientist and engineer.

I am grateful for all the support I received during my studies. The University of Alberta provided a safe and calm campus where I worked many late nights. Digital Research Alliance of Canada provided the computing resources to complete this project. The Alberta Machine Intelligence Institute supported me and my friends in many ways. Finally, I thank my committee members, Adam White, Martha White, and Nathan Sturtevant, for taking the time to read this document, conduct the oral exam, and give valuable feedback.

My days at the University of Alberta have been fun, engaging, and memorable because of my dear friends and mentors. I am honored and lucky to have met you all. You made me feel safe, trusted and welcomed. Thank you Negin, Golnaz, Kevin, Diego, Anna, Hasti, Kimia, Negar, Scott, Mohammad Reza, Bahar, Zahra, Alireza, Prabhat, Aidan, Farzane, Erin, Marlos, Rich, and many more. I hope to have been as good a friend as you have been to me.

I am ultimately grateful for my wonderful family who always love and support me. My appreciation and love for you is boundless and eternal.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Important rare events leave a lasting impression no matter how quickly they pass. Animals can quickly learn from rare rewards. Imagine a rat running in a maze trying to find the food placed at the finish line. Finding the food is a crucial event for the rat. When discovered, the rat's brain replays the sequence of events leading to the food in reverse order (Foster and Wilson, 2006). Replay is a frequent event in the brain of animals both when awake and during sleep, and it plays a crucial role in planning and memory consolidation in animals (Foster and Wilson, 2006; Ólafsdóttir et al., 2018; Singer and Frank, 2009). In this thesis, we study the replay of past experiences, not in animals, but in artificial intelligence systems.

We use the Reinforcement Learning (RL) framework to study intelligent decision-making through trial-and-error interaction. In RL, an agent is engaged in a loop of interaction with its environment. It takes actions in the world and is rewarded based on its actions. The goal of an RL agent is to learn a policy, a strategy for selecting actions, that maximizes total reward. To learn a good policy, one often needs to estimate the value function: a prediction of the future reward of a state or action.

Deep neural networks (NN) can learn complex non-linear representations, allowing RL agents to effectively learn value functions and find good policies in large-scale domains. However, the promise of deep RL comes with unique challenges. Instead of a dataset, RL agents only have access to a stream of experience from the environment. Supervised learning algorithms update with mini-batches of data to reduce noise and efficiently train neural networks. The most common approach for RL agents to update with mini-batches is to hold a memory of recent events and resample experiences.

Experience Replay (ER) is widely used in deep RL and appears critical for good performance. The core idea of ER is to record transitions (experiences) in a memory, called a buffer, replay them by sub-sampling mini-batches to update the agent's value function and policy. Beyond enabling mini-batch updates, ER allows great flexibility in agent design. ER can be used to learn from human demonstrations (pre-filling the replay buffer with human data) allowing off-line pre-training and fine-tuning. ER has been used to learn many value functions in parallel, as in Hindsight ER (Andrychowicz et al., 2018), Universal Value Function Approximators (Schaul, Horgan, et al., 2015), and Auxiliary Task Learning (Jaderberg et al., 2016; Wang et al., 2024). ER can be seen as a form of model-based RL where the replay buffer acts as a non-parametric model of the world (Pan et al., 2018; Van Hasselt et al., 2019), or ER can be used to directly improve model-based RL systems (Lu et al., 2024). In addition, ER can be used to mitigate forgetting in continual learning systems (Anand and Precup, 2024). ER has proven crucial for mitigating the sample efficiency challenges of online RL, as well as mitigating instability due to off-policy updates and non-stationary bootstrap targets. The most popular alternative, asynchronous training, requires multiple copies of the environment, which is not feasible in all domains and typically makes use of a buffer anyway (e.g., Horgan et al., 2018).

There are many different ways ER can be implemented. The most widely used variant, i.i.d or uniform replay, samples experiences from the buffer with equal probability. As discussed in the original paper (Lin, 1991), ER can be combined with lambda-returns and various sampling methods. Experience can be sampled in reverse order it occurred, starting at terminal states. Transitions can be sampled from a priority queue ordered by TD errors—the idea being transitions that caused large updates are more important and should be resampled. Samples can be drawn with or without replacement—avoiding saturating the mini-batch with high priority transitions. The priorities can be periodically updated. We could use importance sampling to re-weight the distribution in the queue, and generally we could dynamically change the distribution during the course of learning. Despite the multitude of possible variants (Hong et al., 2023; Igata et al., 2021; Kobayashi, 2024; Kumar and Nagaraj, 2023; Lee et al., 2019; A. A. Li et al., 2021; M. Li et al., 2022; Sun et al., 2020) simple i.i.d replay remains the most widely used approach.[1]

The exception to this is Prioritized Experience Replay (PER) (Schaul, Quan, et al., 2016), where experience is sampled from the buffer based on TD errors. Like prioritized sweeping that inspired it (Moore and Atkeson, 1993), PER in principle should be more efficient than i.i.d sampling. Imagine, a sparse reward task where non-zero reward is only observed at the end of long trajectories; sampling based on TD errors should focus value updates near the terminal state efficiently propagating reward information across the state space. This approach was shown to improve over i.i.d sampling in Atari when combined with Double DQN (Schaul, Quan, et al., 2016). The results in follow up studies, however, were mixed and did not show a clear benefit for using PER generally (Fedus et al., 2020; Fu et al., 2022; Hessel et al., 2018; Horgan et al., 2018; A. A. Li et al., 2021; Ma et al., 2023). Compared with i.i.d sampling, PER introduces several hyper-parameters controlling importance sampling and how additional experiences are mixed with the prioritized distribution.

[1] See (Wittkuhn et al., 2021) for a nice review.

1.1 Thesis Contributions

In this thesis, we explore several different variations of PER in carefully designed experiments in the hopes of better understanding where and when PER is useful. Canonical PER (Schaul, Quan, et al., 2016) uses several additional components that make the impacts of prioritization harder to analyze. We compare the canonical PER algorithm with simplified variants that build on the core idea of prioritizing with TD error in simple chain tasks where value propagation and prioritization should be critical for performance. We find that only in tabular prediction, do all prioritized variants outperform i.i.d replay. Combining basic prioritization with sampling without replacement and updating the priorities in the buffer (things not done in public implementations), further improves performance in the tabular case. Our results show that prioritization, bootstrapping, and function approximation cause problematic over-generalization, possibly motivating the design choices of PER which ultimately causes the method to function more like i.i.d sampling under function approximation. Our results in chain domains with neural network function approximation and across several classic control domains, perhaps unsurprisingly, shows no clear benefit for any prioritized method.

We also introduce and investigate a natural extension to PER based on ideas from Gradient TD methods (Patterson, White, and White, 2022; Sutton, Maei, et al., 2009). These methods stabilize off-policy TD updates by learning an estimate of the expected TD error. This estimate can be used to compute priorities and is less noisy than using instantaneous TD errors. This expected PER algorithm works well in tabular prediction tasks and noisy counter-examples where PER fails, but is generally worse than i.i.d sampling under function approximation and ties i.i.d in classic control problems—though it appears more stable. Although somewhat of a negative result, expected

PERs performance suggests noise is not the explanation for i.i.d sampling's superiority over PER and more research is needed to find generally useful prioritization mechanisms.

Here is a summary of the proposed contributions in this thesis:

1. A new variant of prioritized replay, expected PER, designed to be robust to noise in the TD error.

2. Several new insights into the interaction between prioritization, bootstrapping, and function approximation.

3. The first investigation into updating priorities and sampling without replacement in PER: when they help and why?

1.2 Thesis Organization

We present this thesis in several chapters. The next chapter develops the background and notation used throughout this thesis. We present the expected PER algorithm and its potential benefits in noisy environments in Chapter 3. Chapter 4 contains a series of experiments regarding prioritized replay and neural network generalization in a Markov chain domain. Chapter 5 explores two unexplored design choices in PER: sampling without replacement and updating priorities in the buffer. In the last content chapter, we scale our experiments to a suite of classic control problems. Finally, we conclude the thesis in Chapter 7 and discuss potential future directions.

Chapter 2

Background, Problem Formulation and Notation

In this chapter, we present the required material used in the rest of this thesis. Section 2.1 introduces the Markov Decision Processes and formalizes the reinforcement learning problem. In Section 2.2, we introduce Q-learning, an action-value method for finding the optimal policy in an MDP, and discuss semi-gradient Q-learning. In Section 2.3, we formalize the prediction problem and describe the semi-gradient TD algorithm as a solution method. We then present the DQN algorithm in Section 2.4 and discuss the role of each of its components and hyperparameters before reviewing prioritized replay and its major design choices in Section 2.5.

2.1 Markov Decision Processes

In this thesis, we investigate the problem of goal-driven learning from trial and error interaction formulated as discrete-time, finite Markov Decision Processes (MDP). MDP is a formalism for the environment in which the decision-making agent operates. The agent and environment are engaged in a continual loop of interaction and can affect each other. The agent's actions change the environment. In response, the environment's feedback signal reinforces certain actions in the agent.

More formally, an MDP is made up of three finite sets, $\mathcal{S}, \mathcal{A}, \mathcal{R}$, corresponding to states, action, and rewards, a mapping, p, and a constant discount factor, $\gamma \in [0, 1]$ — reducing the importance of future rewards. The environment dynamics are defined by p returning next state and reward given current state and action. The transition dynamics may be stochastic or deterministic. Interaction is modulated in discrete time steps. On time step t, the agent selects an action $A_t \in \mathcal{A}$ in part based on the current state, $S_t \in \mathcal{S}$. The MDP transitions to a new state S_{t+1} and emits a reward signal $R_{t+1} \in \mathcal{R}$ back to the agent. The agent's action choices are determined by it's policy $A_t \sim \pi(\cdot|S_t)$, a mapping from state to a distribution over actions.

The goal of reinforcement learning is to adjust π to maximize the expected future total reward. Total reward or the Return, G_t, is defined as the discounted sum of future rewards.

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots .$$

Note the discounting terms bound this infinite series for any reward values. We can now formalize the objective as finding a policy, π^*, that maximizes the expected return.

$$\pi^* \doteq \arg\max_{\pi} \mathbb{E}_{\pi}[G_t].$$

The expectation is dependent on future actions determined by π, and future states and rewards according to the MDP.

2.2 Action-value Methods

The goal of an RL agent is to find a good policy. Action-value methods search for a policy using the Generalized Policy Iteration framework in dynamic programming, where two simultaneous processes interact. One process predicts the value of state-action pairs under the current policy, while the other process improves the policy to be greedy with respect to the current values. In tabular

MDPs, if we can visit every state-action pair enough times, both processes approach the optimal policy and its values. This section introduces Q-learning, a well-known action-value method.

The value of a state-action pair is the expected return from state s, taking action a, and behaving according to π thereafter.

$$q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t | S_t = s, A_t = a].$$

Note that the state-action value function is an expectation dependent on the policy π and the current state and action. Action-value methods, learn to estimate q_π for all states and actions. This process is called policy evaluation. Then, acting greedy with respect to the current policy's action-values, leads to a better policy. This process is called policy improvement. Fortunately, due to Generalized Policy Iteration, we do not need to fully estimate q_π before improving the policy — interleaving incremental policy evaluation with policy improvement steps results in sequence of better and better policies.

In particular, Q-learning estimates the state-action value function, q_π, for each state $s \in \mathcal{S}$ and each action $a \in \mathcal{A}$ via temporal difference updates from sample interactions:

$$\hat{q}(S_t, A_t) = \hat{q}(S_t, A_t) + \alpha\delta_t.$$

Where $\alpha \in \mathbb{R}^+$ is the learning-rate parameter and δ_t is called the Temporal Difference (TD) error, $\delta_t \doteq R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, \cdot) - \hat{q}(S_t, A_t)$. Actions are selected according to an ϵ-greedy policy: selecting $A_t = \arg\max \hat{q}(S_t, \cdot)$ $1 - \epsilon$ percentage of the time and a random action otherwise.

Q-learning updates the value function toward the optimal policy's value function while acting according to an ϵ-greedy policy of the current action values. This is an instance of off-policy learning, where they agent is learning about a policy different than its current behavior policy. If the agent is learning about its current behavior policy, the method is called on-policy instead.

In many tasks, it is not feasible to learn an action-value for every state. In these cases, we use a parametric function like a neural network (NN), to approximate the value function, $\hat{q}_{\mathbf{w}}(s,a) \approx q_\pi(s,a)$, where $\mathbf{w} \in \mathbb{R}^d$ are the network parameters and d is the number of network parameters. The weights are adjusted via semi-gradient Q-learning update rule,

$$\mathbf{w} = \mathbf{w} + \alpha\delta_t\nabla\hat{q}_{\mathbf{w}}(s,a).$$

Similar to Q-learning, α is the learning-rate parameter and δ_t is the TD error at time-step t. The gradient is taken with respect to the network's current weights.

2.3 Prediction Problem Setup

Until now, we have described the RL problem as learning a policy that maximizes the return. This is known as the control problem. An important subproblem in finding a good policy is estimating the value function, called policy evaluation. In these prediction problems, the objective is to learn the state value function of a given fixed policy π,

$$\hat{v}(s) \approx v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s].$$

This can be done using the Temporal Difference learning algorithm (Sutton, 1988), the state-value analog of the Q-learning update above, which has a semi-gradient variant for learning $\hat{v}_{\mathbf{w}} : \mathcal{S} \rightarrow \mathbb{R}$,

$$\mathbf{w} = \mathbf{w} + \alpha\delta_t\nabla\hat{v}_{\mathbf{w}}(s).$$

Where $\delta_t \doteq R_t + \gamma\hat{v}_{\mathbf{w}}(S_{t+1}) - \hat{v}_{\mathbf{w}}(S_t)$. See Sutton and Barto, 2018 for an extensive overview of RL.

2.4 Deep Q Networks

Semi-gradient Q-learning when combined with NNs is often unstable. The DQN algorithm builds on Q-learning by adding several algorithmic components to stabilize learning. In particular, DQN combines experience replay, target networks, and an optimizer with semi-gradient Q-learning (Mnih et al., 2015). This section introduces experience replay and target networks, discuss their roles and key hyperparameters, concluding with the complete DQN algorithm and its pseudo code.

2.4.1 Experience Replay

Experience Replay (ER) enables mini-batch updates to the value function $\hat{q}_{\mathbf{w}}$ from a finite, first-in-first-out buffer of transitions. DQN prefers mini-batch style updates that reduce the noise in gradient estimates over updating with the most recent transition as in Q-learning. Stored experience in the buffer is sampled uniformly or i.i.d meaning each transition is sampled with replacement from the buffer with equal probability — the value estimate on the current step is updated based on experiences observed in the recent past, not necessarily the most recent transitions.

In its original conception (Lin, 1991), ER played a similar role to model based approaches like Dyna, enabling reuse of past experience to achieve better performance with fewer steps of interaction with the environment. The mini-batch size controls the rate at which past experiences are resampled, this hyperparameter must be carefully selected for each environment. An intermediate mini-batch size often balances computational cost with data reuse.

In DQN, ER serves another role as well, it mitigates catastrophic forgetting: a tendency of NNs to make global changes to their weights when faced with novel data resulting in loss of previously learned information. Sampling transitions from a large enough buffer can diversify the updates mitigating the forgetting issue in NNs. Success of DQN often relies on careful selection of the buffer size depending on the problem.

At first glance it appears that a larger replay buffer is preferred as it guarantees more data diversity. This choice comes at the cost of a larger memory footprint not viable in large scale problems. Moreover, the policy and environment may change over time rendering old transitions in the buffer irrelevant or stale. Tuning the hyperparameters of ER is hard and problem dependent, using default values often leads to sub-optimal performance or total failure.

2.4.2 Target Network

Semi-gradient Q-learning combined with NNs is unstable as it suffers from the deadly triad (Sutton and Barto, 2018): the combination of TD updates and function approximation can diverge when faced with an off-policy learning problem. Target networks reduces the speed at which the target of the TD error change in order to stabilize learning.

Target Networks replace $\max_a \hat{q}_{\mathbf{w}}(S_{t+1}, \cdot)$ in the TD error with an older copy of the network. This target network is then updated with the most recent network weights once in a while. The rate at which the target network is updated is called the target network refresh rate and is an important problem dependent hyperparameter. A larger refresh rate keeps the targets fixed for a longer duration slowing down learning but potentially stabilizing the updates.

2.4.3 The DQN Algorithm

Putting together the components in previous sections alongside the semi-gradient Q-learning update rule, we arrive at the DQN algorithm. At each step of interaction, a new transition is stored in the replay buffer. Then, a mini-batch is sampled with replacement form a uniform replay buffer. The

TD error and gradient of the mini-batch is computed and network weights are updated using the Adam optimizer (Kingma and Ba, 2015). The target network weights are updates at regular intervals and actions are selected using an ϵ-greedy policy of the current action-value estimates. Pseudo code of DQN used throughout this thesis is presented in Algorithm 1.

Algorithm 1 DQN with Uniform Replay

```
**Input:** mini-batch size $b$, learning-rate $\alpha$, training time $T$, target refresh rate $\tau$.
**Initialize:** $q$ network parameters $\mathbf{w}$, target network parameters $\mathbf{w}_{\text{target}} = \mathbf{w}$, buffer $B$, $\Delta = 0$.
Observe $S_0$ and choose $A_0 \sim \pi_{\mathbf{w}}(S_0)$
for $t = 1$ to $T$ do
    Observe $R_t, S_t, \gamma_t$
    Store transition $(S_{t-1}, A_{t-1}, R_t, S_t, \gamma_t)$ in buffer $B$
    for $j = 1$ to $b$ do
        Sample transition $(S_j, A_j, R_j, S_{j+1}, \gamma_{j+1})$ from buffer $B$ with probability $1/|B|$
        Compute TD error $\delta_j = R_j + \gamma_j \max_a q(S_{j+1}, a, \mathbf{w}_{\text{target}}) - q(S_j, A_j, \mathbf{w})$
        Accumulate gradient $\Delta \leftarrow \Delta - \delta_j \nabla_{\mathbf{w}} q(S_j, A_j, \mathbf{w})$
    end for
    Update $\mathbf{w} \leftarrow \text{adam}(\mathbf{w}, \frac{\Delta}{b}, \alpha)$; Reset $\Delta = 0$
    if $t\%\tau = 0$ then
        Refresh target network $\mathbf{w}_{\text{target}} \leftarrow \mathbf{w}$
    end if
    Choose action $A_t \sim \pi_{\mathbf{w}}(S_t)$
end for
```

The DQN algorithm has many key hyperparameters affecting performance and stability. In this thesis, we strive to carefully discuss hyperparameter choices when presenting the experiments.

The combination of semi-gradient TD with NNs and experience replay follows trivially from DQN algorithm. Note that we formalized the prediction problem as an on-policy problem: the objective is learning about the behavior policy. In the on-policy setting, we do not face the deadly triad and therefore can ignore target networks.

2.5 Prioritized Experience Replay

In this section we introduce Prioritized Experience Replay (PER) (Schaul, Quan, et al., 2016) studied throughout the rest of this thesis. In the simplest case, PER samples each item in a mini-batch according to a prioritized distribution in contrast to uniform sampling in vanilla ER. A transition stored in the buffer at index j is sampled according the following probability, $\frac{p_j}{\sum_i p_i}$, where p_j is the priority of the jth transition in the buffer and the summation is over all transitions in the buffer.

The choice of priorities in PER controls the sampling distribution. The most common strategy is to prioritize samples according to the magnitude of the TD error, $|\delta|$. TD error based prioritization was first introduced to speed up learning in the Dyna framework (Moore and Atkeson, 1993; Peng and Williams, 1993). The intuition behind this approach is that samples with larger TD errors are more informative as their current predictions are incorrect. Focusing updates on these samples can improve learning efficiency.

The DQN algorithm can be modified to use prioritized replay. A simplified variant, which we call Naive PER, stores each incoming transition alongside the absolute value of its TD error as the priority. At each step, transitions are sampled in proportion to their priorities to form a mini-batch and used to update the value function. The TD error computed during the update is then used to renew the priority of sampled transitions. Pseudo code of DQN with Naive PER is presented in Algorithm 2.

Algorithm 2 DQN with Naive PER

```
Input: mini-batch size $b$, learning-rate $\alpha$, training time $T$, target refresh rate $\tau$.
Initialize: $q$ network parameters $\mathbf{w}$, target network parameters $\mathbf{w}_{\text{target}} = \mathbf{w}$, prioritized buffer $B$, $\Delta = 0$.
Observe $S_0$ and choose $A_0 \sim \pi_{\mathbf{w}}(S_0)$
for $t = 1$ to $T$ do
    Observe $R_t, S_t, \gamma_t$
    Store transition $(S_{t-1}, A_{t-1}, R_t, S_t, \gamma_t)$ in buffer $B$ with priority $p_{t-1} = |\delta_{t-1}| = |R_t + \gamma_t \max_a q(S_t, a, \mathbf{w}_{\text{target}}) - q(S_{t-1}, A_{t-1}, \mathbf{w})|$
    for $j = 1$ to $b$ do
        Sample transition $(S_j, A_j, R_j, S_{j+1}, \gamma_{j+1})$ from buffer $B$ with probability $\frac{p_j}{\sum_i p_i}$
        Compute TD error $\delta_j = R_j + \gamma_j \max_a q(S_{j+1}, a, \mathbf{w}_{\text{target}}) - q(S_j, A_j, \mathbf{w})$
        Accumulate gradient $\Delta \leftarrow \Delta - \delta_j \nabla_{\mathbf{w}} q(S_j, A_j, \mathbf{w})$
    end for
    for $j = 1$ to $b$ do
        Update priority $p_j = |\delta_j|$
    end for
    Update $\mathbf{w} \leftarrow \text{adam}(\mathbf{w}, \frac{\Delta}{b}, \alpha)$; Reset $\Delta = 0$
    if $t\%\tau = 0$ then
        Refresh target network $\mathbf{w}_{\text{target}} \leftarrow \mathbf{w}$
    end if
    Choose action $A_t \sim \pi_{\mathbf{w}}(S_t)$
end for
```

The canonical PER algorithm (Schaul, Quan, et al., 2016) adds several modifications to the Naive PER algorithm described above. In this thesis we refer to this variant as DM-PER to avoid confusion. Here we present the modifications in DM-PER.

1. The priorities are scaled by an exponent hyperparameter α controlling the strength of priorities on the sampling distribution, $\frac{p_j^\alpha}{\sum_i p_i \alpha}$, where a smaller value of α leads to a distribution closer to uniform.

2. The prioritized distribution is then mixed with a uniform distribution explicitly. The mixture is controlled by another hyperparameter that interpolates between a fully prioritized and a fully uniform distribution.

3. The magnitude of updates are adjusted using importance sampling weights such that high priority transitions will have smaller update sizes. These weights are dampened by another exponent hyperparameter and rescaled to be in range $[0, 1]$.

4. The initial priority of transitions are set to maximum instead of the TD error to avoid the cost of computing these TD errors. A New transition will quickly be sampled and its maximal priority will be replaced by the TD error.

The addition of new hyperparameters in DM-PER makes fair comparison with Naive PER and uniform replay difficult as any additional compute spent tuning the new hyperparameters is not spent on the baselines. In order to remain fair, we do not tune the new hyperparameters and instead choose fixed values from the original paper. These hyperparameter values are presented in Table 2.1.

Priority exponent (α)	Importance sampling exponent	uniform mix-in ratio
0.6	$0.4 \rightarrow 1.0$	10^{-3}

Table 2.1: Hyperparameters specific to DM-PER. Arrow indicates linear schedule over training time.

In this thesis we investigate several variants of prioritized replay. The simplified variant (Naive PER) and the canonical approach (DM-PER) introduced here will be accompanied by expected PER introduced in the next chapter and two additional variants introduced later in this document.

2.6 Summary

In this chapter, we formulated the RL control problem using MDPs and discussed the action-value class of solutions. In particular, we focused on Q-learning and its deep RL counterpart, DQN. We formulated the prediction problem and discussed semi-gradient TD as a solution to prediction problems. Finally, we discussed the roles of ER and target networks in DQN and reviewed PER as an alternative to i.i.d replay.

Chapter 3

Expected Prioritized Experience Replay

In this chapter we present the *expected* PER algorithm, designed to be robust to noise in TD error. We describe its implementation and connection to the gradient TD family of algorithms. Finally we demonstrate the potential benefit of EPER in a noisy Markov chain domain where Naive PER fails.

3.1 Prioritizing with expected TD error

The TD error is a noisy quantity and may be unreliable for prioritization. Sensor measurements and predictions are often noisy in the real world. Prioritizing by a noisy TD error could lead to slow learning or even biased solutions. A good prioritization scheme should be able to perform well in noisy settings.

In order to study the effects of noise on the prioritization strategy, we introduce a new prioritization variant *expected* PER (EPER). Instead of using the sample TD error, δ_t, which can be noisy when the reward or the transition dynamics are stochastic, EPER uses an estimate of the expected TD error that averages out random effects from transition dynamics and the reward signal.

In the prediction setting, we compute the expectation conditioned on the state, averaging out random effect from action selection. $\mathbb{E}\left[\delta_t \mid S_t = s\right]$. But in the control setting the expectation is conditioned on both state and action.

It is possible to consider state based expectation in the control setting as well but we did not investigate that variant in this thesis.

Learning this expectation can be formulated as a simple least-squares regression problem with samples δ_t as the target, yielding the following online update rule: $\theta_{t+1} \leftarrow \theta_t + \alpha(\delta_t - h_\theta(S_t))\nabla_\theta h_\theta(S_t)$, where h_θ is a parametric approximation of δ_t with parameters θ. Here we use the same learning rate, α, as we did to update the value function. This secondary estimator forms the basis of the gradient TD family of methods (Patterson, White, and White, 2022; Sutton, Maei, et al., 2009) making it natural to combine with recent gradient TD algorithms such as EQRC (Patterson, White, and White, 2022). In other words, if we use EQRC instead of DQN, we can use EPER to attain a less noisy signal for computing priorities with no extra work because EQRC is estimating h_θ anyway.

Temporal difference learning combined with off-policy learning and function approximation is unstable and can even diverge. Deep RL methods use target networks to stabilize learning. Gradient TD algorithms, on the other hand, take a principled approach to learning value functions. They use stochastic gradient descent to minimize a proxy for the value error called projected Bellman error using both the sampled TD error and a secondary estimator for the expected TD error. The secondary estimator allows gradient TD methods to be stable and reliable without needing target networks. As mentioned above, combining EPER with gradient TD methods such as TDRC or EQRC is a good choice as they both require the secondary estimator.

3.2 The expected PER algorithm

In this section, we present the pseudo code for the EPER algorithm and discuss its implementation details. EPER modifies the Naive PER algorithm to use a learned estimate of the expected TD error instead of the sample TD error to prioritize transitions in the replay buffer. The expected TD error is learned as an online regression problem using the same mini-batches sampled to update the value function. We present the pseudo code of DQN with expected PER in Algorithm 3. The highlighted lines show the difference between Naive PER and EPER. Note that in this pseudo code and the experiments, we use the same learning rate to update both the value function and the secondary expectation estimator.

Throughout this thesis, we implement the EPER algorithm described above in two ways. In the tabular setting, i.e., when the value of each state is estimated independently of other states, the expected TD error is also learned separately. When using a neural network to approximate the value function, however, we use a linear function of the second to last layer of the network to learn the expected TD error. We restrict the gradient flow from this linear estimator to the previous layers to avoid changing the features, which allows the network to learn features relevant to the main task.

Algorithm 3 DQN with expected PER

```
Input: mini-batch size $b$, learning-rate $\alpha$, training time $T$, target refresh rate $\tau$.
Initialize: $q$ network parameters $\mathbf{w}$, target network parameters $\mathbf{w}_{\text{target}} = \mathbf{w}$, prioritized buffer $B$, $\Delta = 0$, $h$ expected TD error estimator parameters $\theta$.
Observe $S_0$ and choose $A_0 \sim \pi_{\mathbf{w}}(S_0)$
for $t = 1$ to $T$ do
    Observe $R_t, S_t, \gamma_t$
    Store transition $(S_{t-1}, A_{t-1}, R_t, S_t, \gamma_t)$ in buffer $B$ with priority $p_{t-1} = |h(S_{t-1}, A_{t-1}, \theta)|$.
    for $j = 1$ to $b$ do
        Sample transition $(S_j, A_j, R_j, S_{j+1}, \gamma_{j+1})$ from buffer $B$ with probability $\frac{p_j}{\sum_i p_i}$
        Compute TD error $\delta_j = R_j + \gamma_j \max_a q(S_{j+1}, a, \mathbf{w}_{\text{target}}) - q(S_j, A_j, \mathbf{w})$
        Accumulate gradient $\Delta_q \leftarrow \Delta_q - \delta_j \nabla_{\mathbf{w}} q(S_j, A_j, \mathbf{w})$
        Accumulate gradient $\Delta_h \leftarrow \Delta_h - (h(S_j, A_j, \theta) - \delta_j) \nabla_\theta h(S_j, A_j, \theta)$
    end for
    for $j = 1$ to $b$ do
        Update priority $p_j = |h(S_j, A_j, \theta)|$
    end for
    Update $\mathbf{w} \leftarrow \text{adam}(\mathbf{w}, \frac{\Delta_q}{b}, \alpha)$; Reset $\Delta_q = 0$
    Update $\theta \leftarrow \text{adam}(\theta, \frac{\Delta_h}{b}, \alpha)$; Reset $\Delta_h = 0$
    if $t \% \tau = 0$ then
        Refresh target network $\mathbf{w}_{\text{target}} \leftarrow \mathbf{w}$
    end if
    Choose action $A_t \sim \pi_{\mathbf{w}}(S_t)$
end for
```

3.3 EPER can succeed in the presence of noise

Here, we will show that Naive PER can fail in the presence of noise even in simple domains, whereas EPER manages to successfully prioritize samples while being a much simpler method than DM-PER. The task is to estimate the state-value function of a random policy in a 50 state Markov chain with the only reward on the terminal transition (described in more detail in the next chapter). To simulate a noisy environment but keep the value function fixed, the terminal reward is polluted by zero mean non-symmetric noise.

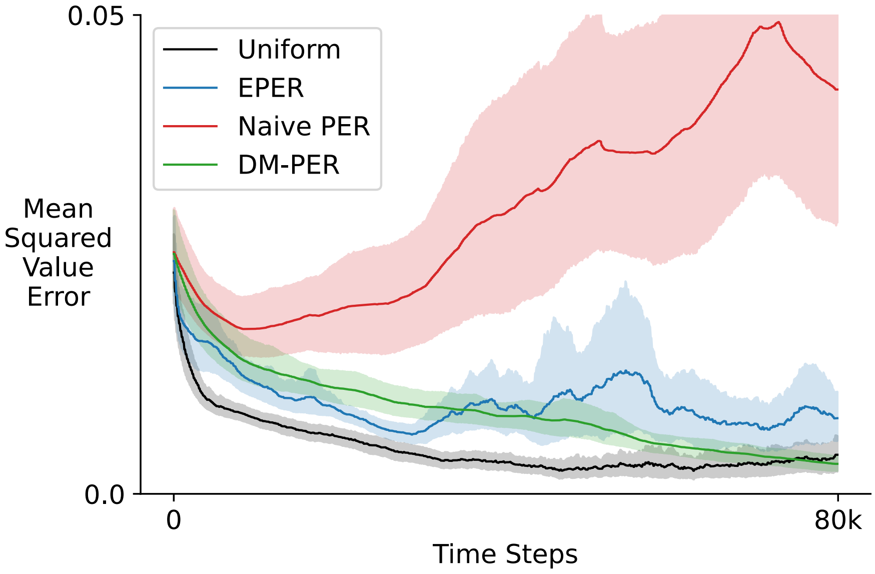

Figure 3.1: Prioritization can be problematic in noisy prediction with NNs. Naive PER does not learn the correct value function but EPER and DM-PER can get to low value errors during training. Results averaged over 30 trials; shaded region are 95% bootstrap Confidence Intervals (CI).

Figure 3.1 demonstrates the potential benefits of EPER over Naive PER. The hyperparameters of all methods are systematically tuned, and still we see Naive PER fails to learn the correct value function and its value error increases over time. Other replay variants including EPER, DM-PER, and uniform replay, all manage to get to low value error by the end of experiment.

Failure of Naive PER in this experiment may be due to outliers in the TD error caused by noisy transitions. A large TD error, affected by noise, will increase the priority of its transition but mislead the agent to chase stochasticity and learn a wrong value function. The DM-PER algorithm is robust in this case, which is not surprising given its use of importance sampling and mixing in i.i.d samples. EPER is not as robust but achieves this with a much simpler approach. This experiment shows that noisy priorities can mislead agents, and using an estimate of expected TD error, such as EPER, can improve the robustness of prioritization in noisy environments.

3.4 Summary

This chapter presented a new prioritization scheme where transitions are prioritized according to the magnitude of an estimation of TD error instead of the sample TD error. We proposed a simple online regression method for estimating the expected TD error as done in the gradient TD family of algorithms. Finally, we demonstrated how EPER can succeed in a noisy Markov chain prediction task where Naive PER fails due to noise.

Chapter 4

Investigating Prioritization in Markov Chains

The idea of prioritized replay is based on the tabular notion of value propagation and the interplay between neural network generalization and prioritized replay remains an open question. This chapter explores the combined effect of prioritized replay and neural network generalization in RL agents.

4.1 Comparing Sample Efficiency in Prediction

In this section we ask several questions in a sparse reward task where rapid value propagation should require careful sampling from the replay buffer. Does naive prioritization improve performance over uniform replay? Do the additional tricks in DM-PER reduce the efficiency of value propagation when they are not really required? Finally, does robustness to noisy TD errors, as in EPER, matter in practice? We investigate these questions with tabular and neural network representations.

We consider both policy evaluation and control problems in a 50-state Markov chain environment visualized in Figure 4.1. This is an episodic environment with $\gamma = 0.99$ chosen to present a difficult value propagation problem. In every episode of interaction, the agent starts at the leftmost state and at each step takes the `left` or `right` action which moves it the corresponding neighbour state. The only reward in this environment is +1 when reaching the rightmost state at which point the episode terminates.

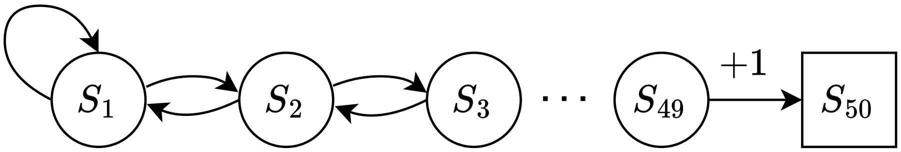

Figure 4.1: The 50-state Markov chain environment.

In the policy evaluation experiments, the objective is to estimate the state value function of the random policy. The data for the replay buffer is generated by running the random policy, making this an on-policy prediction task. The performance measure is the Mean Squared Value Error (MSVE) between estimated value function and true value function: $\text{MSVE}(\mathbf{w}) = \sum_s d(s)(v_\pi(s) - \hat{v}_\mathbf{w}(s))^2$ where $d(s)$ is the state visitation distribution under the uniform policy. In this experiment we have two settings, one where the value function is tabular and one where it is approximated by a two layer neural network with 32 hidden units in each layer and rectified linear unit (ReLU). We systematically tested a broad set of learning rates, buffer size, and batch sizes—over 50 combinations with 30 seeds each. Table 4.1 lists the tested hyperparameter ranges for agents in prediction setting.

	Tabular agents	NN agents
Learning rate	$[8^{-6}, 8^{-5}, 8^{-4}, 8^{-3}, 8^{-2}]$	$[8^{-6}, 8^{-5}/4, 8^{-5}, 8^{-4}/4, 8^{-4}, 8^{-3}/4, 8^{-3}]$
Adam optimizer β_1	0.9	0.9
Adam optimizer β_2	0.999	0.999
Batch size	$[1, 8, 64]$	$[1, 8, 64]$
Buffer size	$[800, 8000, 80000]$	$[800, 8000, 80000]$
Network size	-	2×32 MLP with ReLU
Training time	80000	80000

Table 4.1: Hyperparameters of prediction agents in Markov chain.

In Figure 4.2 we show a representative result with batch size 8, buffer size 8000, and learning rate 8^{-4} in the tabular setting and 8^{-5} in the neural network setting. The remaining results are in Section 4.4.

Figure 4.2 shows the learning curves of different replay methods for policy evaluation in the 50-state Markov chain over time. All three prioritized replay variants perform similarly and they are more sample efficient than uniform replay. The heatmaps show estimated values across states over time. Comparing the heatmap of tabular uniform replay with tabular Naive PER shows an increase in value propagation through the chain when using prioritization.

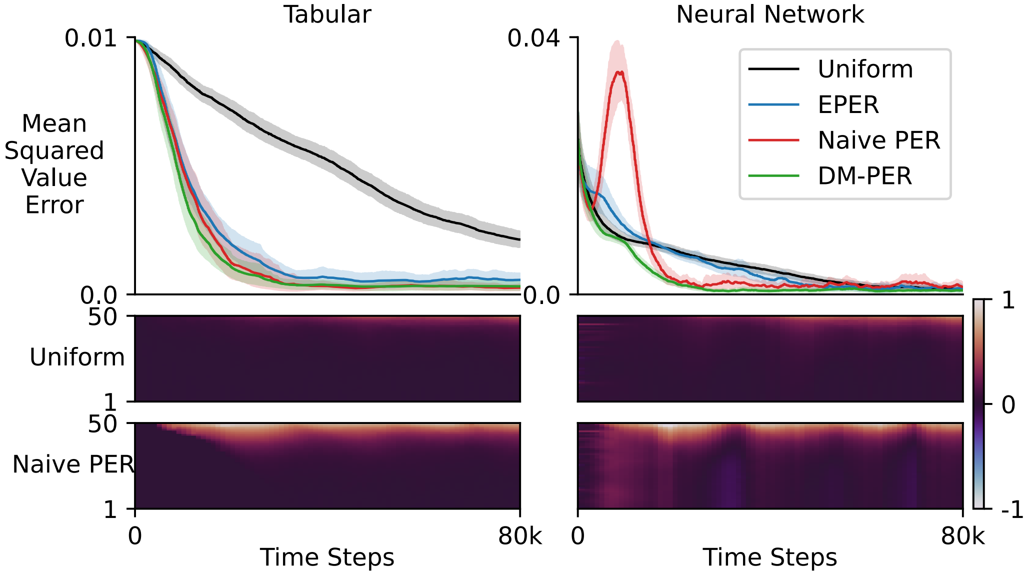

Figure 4.2: Prioritized methods can improve sample efficiency in prediction on the 50-state chain in tabular (left) and NN prediction (right). With NN function approximation Naive PER exhibits an increase in MSVE during early learning. The heatmaps show estimated values of the states, 1 to 50, over time. Results are averaged over 30 seeds; shaded regions are 95% bootstrap CI.

In the neural network setting, the error of Naive PER increases during early learning and then drops to the level of other prioritized replay methods. The gap between other prioritized methods (DM-PER and EPER) and uniform replay is smaller in the neural network setting compared with the tabular setting. Additionally, these two prioritized methods do not exhibit an increase in the MSVE like Naive PER.

Figure 4.3 shows the sensitivity of replay methods to learning rate for batch size 8 and buffer size 8000 in the chain prediction problem. Prioritized replay is more sample efficient than uniform replay in the tabular setting, especially with smaller step sizes. But when using neural networks, the early increase in MSVE of Naive PER, pulls its average performance below other algorithms.

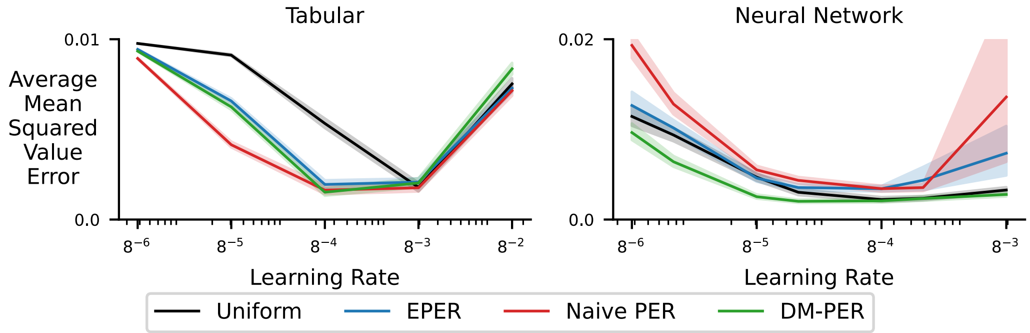

Figure 4.3: Sensitivity to learning rate in prediction chain task. All PER variants perform better or equal to uniform replay in tabular setting. Naive PER's error spike makes it more sensitive to step size than other methods with neural networks. Results are averaged over 30 seeds and shaded region is 95% bootstrap CI.

Perhaps Naive PER oversamples a few transitions which causes the network to spend a lot of its capacity minimizing the error of those transitions at the cost of a worse prediction in other states. It is possible that EPER can mitigate the oversampling issue because the initial estimates are randomized which helps avoid oversampling certain transitions. DM-PER reduces the negative effect of oversampling by using importance sampling weights to reduce the magnitude of updates with high priority transitions.

To better understand what is going on we visualize the probability of updating a state over time in Figure 4.4. This probability is calculated by summing over the probabilities of sampling a transition starting from a given state at a given time based on transitions in the replay buffer. We use the same hyper parameter settings as in Figure 4.2.

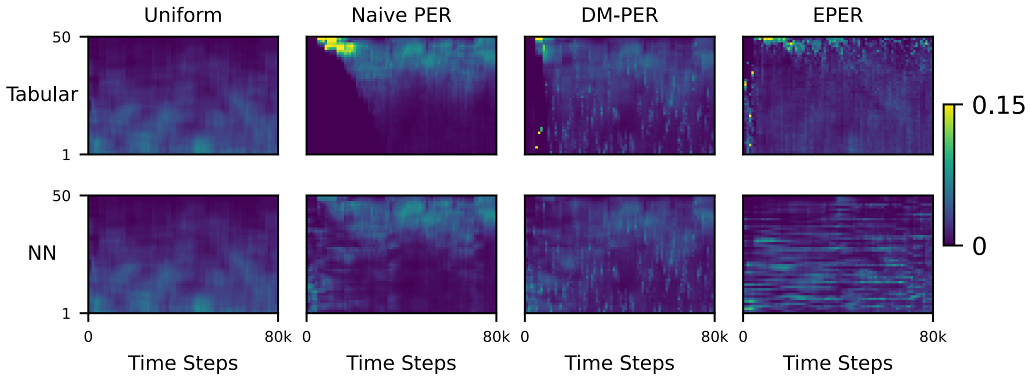

Figure 4.4: Probability of sampling a transition starting from each state (1 to 50) from the buffer at each time point, in the 50-state Markov chain for one run. Naive PER over-samples high priority transitions at the end of the chain.

The sampling distribution of tabular Naive PER follows the intuition from prioritized sweeping by putting most of the probability mass on the rewarding transition at the end of chain, then, increasing the probability of nearby states in a backward fashion to help value propagation. Under neural network function approximation the pattern is similar but more uniform. This is caused by the random initialization of network parameters which generates non-zero TD errors across the state space. The sampling distribution of both DM-PER and EPER are, on the other hand, more structured. Both feature non-terminal transitions with high probability (bright spots) and striping. It is hard to speculate why this occurs, nevertheless, these patterns provide evidence that combining prioritization with NNs can result distributions very different from the tabular case.

4.2 Overestimation due to Prioritization, Generalization, and Bootstrapping

In the previous section we saw that Naive PER exhibited a spike in early learning, but why? One possible explanation is that Naive PER is oversampling the terminal transition which causes the NN to inappropriately over-estimate nearby states, causing more oversampling, and so on, spreading across all states. The heatmap in Figure 4.4 provides some evidence of this. One way to prevent over-generalization is to employ target networks. We use the same setup as in figure 4.2 and only consider Naive PER with neural nets.

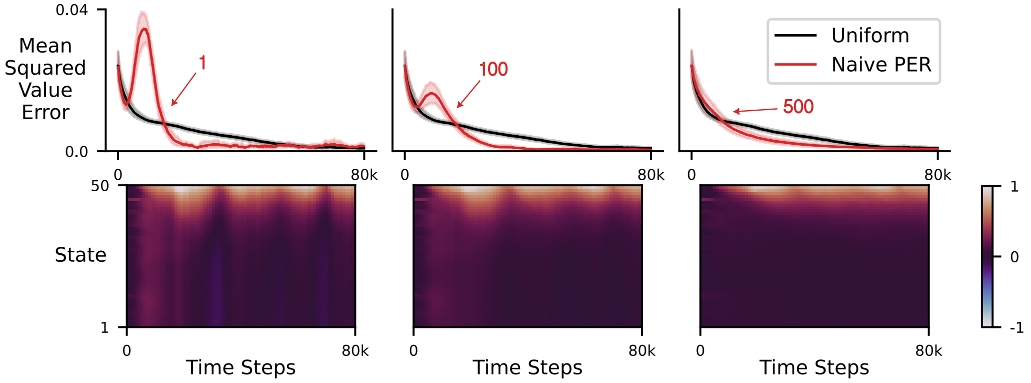

Figure 4.5: Target Networks can mitigate Naive PER's poor performance in the 50-state Markov chain prediction task with NNs. Red numbers above curves indicate Target Network update rate.

The results in Figure 4.5 show the performance of Naive PER with three different target network update rates (1, 100, 500). An update rate of 1 is identical to not using target networks at all. As we update the target network less frequently we see the spike in the learning curves is reduced. Notice that heatmap for the value function with an update rate of 500 is very similar to Naive PER in the tabular case (see Figure 4.2). We only see a minor performance improvement over uniform replay in Figure 4.5, but this is expected because updating the target network infrequently is reducing the update rule's ability to propagate value backwards via bootstrapping.

4.3 Comparing Sample Efficiency in Control

In this section we turn our attention to a simple control task, again designed in such a way that value propagation via smart sampling should be key. Here our main question is: do the insights about the benefits of prioritization persist when the policy changes and exploration is required.

In the tabular setting, we use Q-learning (without target networks) and in neural network setting we explore two setups: (1) DQN (with target refresh rate of 100) and (2) EQRC (as an alternative method without target network). We report steps to goal as the performance metric for the 50-state Markov chain problem. Buffer size is fixed to 10000, batch size 64, and we pick the best learning rate for each method. Each control agent is run for 100000 steps with an ϵ-greedy policy with $\epsilon = 0.1$. Table 4.2 lists the tested hyperparameter ranges for agents in control setting. We chose the learning rate for each agent by maximizing over average performance across the tested learning rates.

	Q-Learning agents (tabular)	DQN and EQRC agents (NN)
Learning rate	$[8^{-7}, 8^{-6}, 8^{-5}, 8^{-4}, 8^{-3}, 8^{-2}]$	$[8^{-5}, 8^{-4}, 8^{-3}, 8^{-2}, 8^{-1}]$
Adam optimizer β_1	0.9	0.9
Adam optimizer β_2	0.999	0.999
Batch size	8	8
Buffer size	10000	10000
Network size	-	2 × 32 MLP with ReLU
Target refresh	-	100 (only DQN)
Exploration ϵ	0.1	0.1
Training time	100000	100000

Table 4.2: Hyperparameters of control agents in Markov chain

The results in Figure 4.6 are somewhat unexpected. The dotted line depicts the performance of the optimal policy. Even in the tabular case, Q-learning with uniform replay is a better than all the three prioritized methods. DM-PER performs just as well as uniform replay, but this could be explained by the fact that DM-PER's sampling is closer to uniform compared with the other prioritization schemes as shown previous in Figure 4.4. Naive PER eventually reaches the near optimal policy and EPER performs poorly. Under neural network approximation, all tested algorithms have a wide overlapping confidence regions, even with 50 seeds, making differences between methods non conclusive. It seems good performance in prediction does not necessarily translate into improvement in control, even in the same MDP.

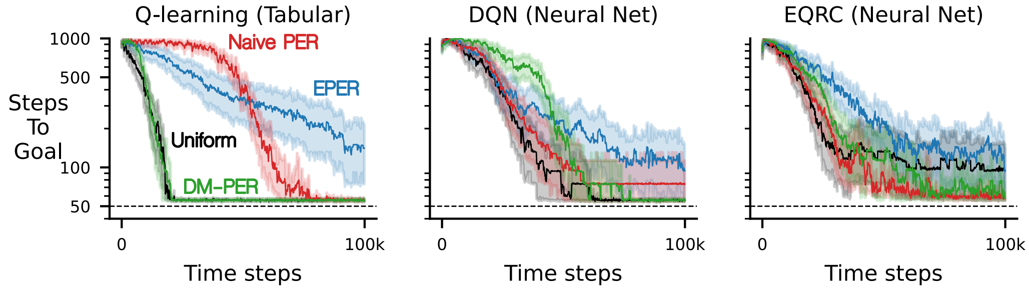

Figure 4.6: Prioritization is not more sample efficient than uniform for control in the 50-state Markov chain environment. Results averaged over 50 seeds; shaded regions are 95% bootstrap CI.

4.4 Additional Results in Prediction

This section presents the set of all results in the chain prediction problem. We show learning curves for all batch and buffer size combinations in Table 4.1. We select the learning rate of each algorithm in each batch-buffer size setting by maximizing the average performance across 30 seeds. The results are organized in Figure 4.7.

We mostly see similar behavior across all tested hyperparameters. In the tabular setting, prioritization improves sample efficiency in smaller batch sizes but its effect disappears with larger batch sizes. Increasing the buffer size in the tabular setting resulted in less noisy learning curves and more consistent behaviour acorss methods. In the large batch and buffer setting, there is no difference between any of the methods.

With neural networks Naive PER suffers from the large spike in error across most tested hyeprparameter combinations. Similar to the tabular setting, increasing the buffer size makes all algorithms perform similar to each other. On the other hand, increasing the batch size, particularly with small buffer size, appears to reduce the spike in value error.

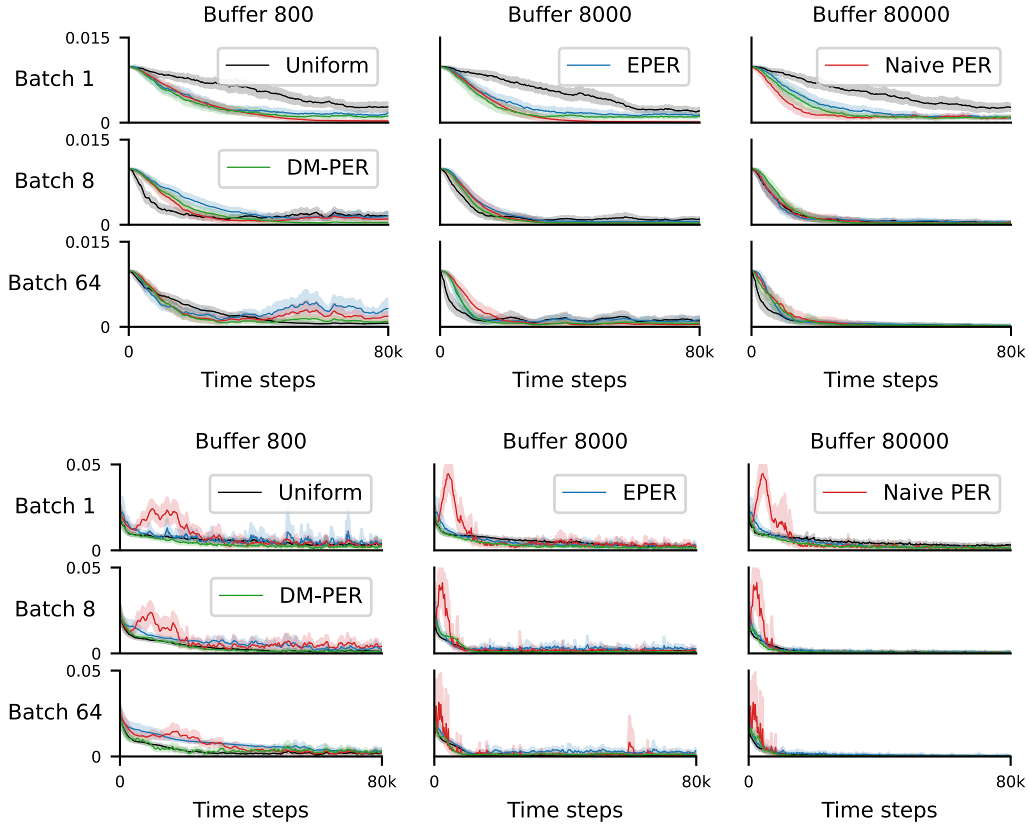

Figure 4.7: Performance of tabular (top) and neural network (bottom) replay agents in the prediction chain task. Tabular prioritized agents generally outperform uniform replay. In the neural networks setting, Naive PER's error spike is present across all batch-buffer sizes tested. Results are averaged over 30 seeds; shaded region is 95% bootstrap CI.

4.5 Summary

In this chapter, we conducted experiments on a Markov chain environment to investigate the effect of prioritization on neural network generalization. We observed prioritized replay outperforming uniform replay in the tabular setting by improving the speed value propagates across states. Naive PER showed a spike in error when combined with a neural network. A closer look at values suggests an aggressive generalization of the neural net that leads to overestimation. A follow-up experiment revealed that combining prioritized sampling with neural nets and bootstrapping can lead to a pathological value over-estimation problem. Finally, in the control setting, uniform replay outperformed all prioritized variants in the tabular setting. This unexpected result suggests the benefit of prioritization in control problems may not be as straightforward as it is for value function learning. Furthermore, control results with neural nets showed no clear advantage to prioritization for both DQN and EQRC algorithms.

Chapter 5

Exploring Simple Modifications to Prioritized Replay

In this chapter we explore two simple but natural improvements to replay that could improve performance. There are many possible refinements, and many have been explored in the literature already. Here we select two that have not been deeply explored before, specifically (1) sampling transitions with or without replacement, and (2) recomputing priorities of samples in the buffer.

5.1 Sampling Without Replacement

When sampling a mini-batch from the replay buffer, one has the option to sample transitions with or without replacement. This decision is important in PER because sampling with replacement can cause a high priority transition to be repeatedly sampled into the same mini-batch. This certainly happens on the first visit to the goal state in the 50 state chain. Uniform replay avoids this problem by design. Most reference implementations of PER sample with replacement. We hypothesize that duplicate transitions in the mini-batch reduces the sample efficiency of prioritized methods, effectively nagating the benefit of mini-batches.

We compare Naive PER with and without replacement sampling and uniform replay in the 50-state Markov chain prediction domain under tabular and neural net function approximation. We used a two layer network with 32 hidden units and ReLU activation, a batch size of 8000 and experimented with several mini-batch sizes (1, 8, 64, 256). With batch size 1, with and without replacement are identical. We report a representative result with learning rate of 8^{-4} in the tabular setting and 8^{-5} in the neural network setting and report MSVE under the target policy over training time.

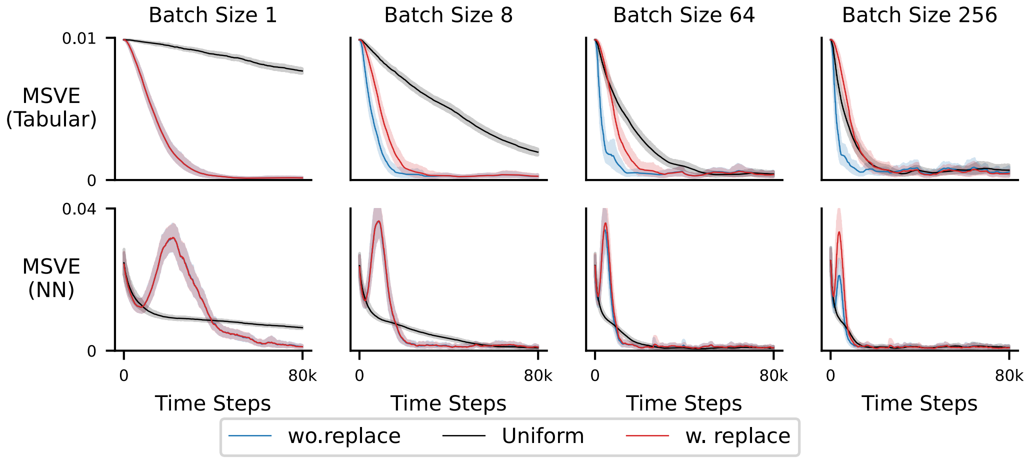

Figure 5.1: Sampling without replacement improves the performance of Naive PER in the tabular setting but not with neural nets. Results are averaged over 50 seeds and shaded regions are 95% bootstrap CI.

Figure 5.1 shows that sampling without replacement provides a minor improvement on the performance of Naive PER in tabular prediction, where Naive PER was already working well, but does not help when combined with NN function approximation. In fact, we again see Naive PER's characteristic spike due to over-generalization and bootstrapping. This poor performance is somewhat mitigated by larger batch sizes, but still uniform replay is better. Note, as expected, the performance of uniform replay suffers with smaller batch sizes.

Now we turn to the control setting to evaluate the impact of sampling without replacement. We tested tabular Q-learning and neural network DQN settings. The DQN agent has a two layer network with 32 hidden units and

ReLU activation with target refresh rate 100. All agents have buffer size 10000 and a series of batch sizes similar to the previous experiment. The learning rate of each agent is selected by sweeping over a range of step sizes and maximizing over average performance (sweep details in Section 5.3). Figure 5.2 summarizes the results.

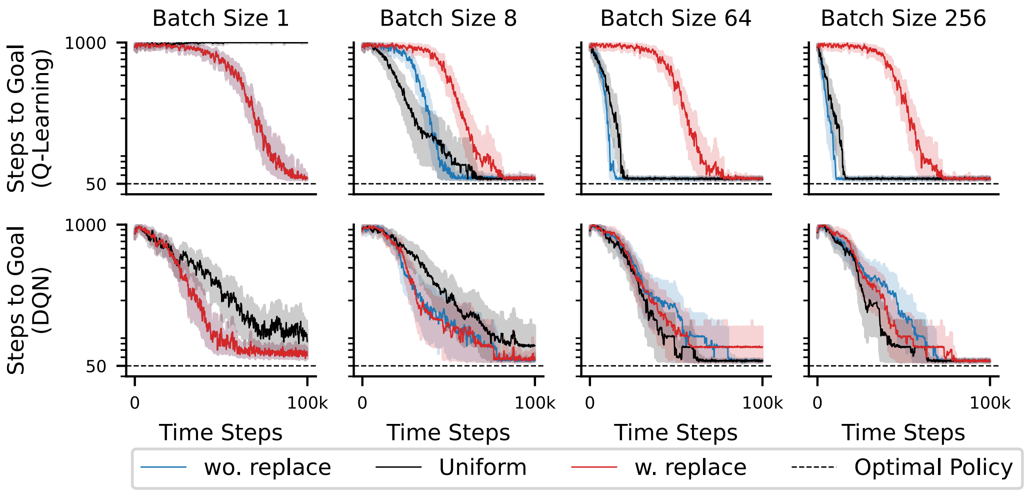

Figure 5.2: Sampling with and without replacement in control using Naive PER with tabular and NN representations. Without replacement sampling only helps in the tabular setting. Results averaged over 50 seeds; shaded regions are 95% bootstrap CI.

In tabular control we see a significant improvement in Naive PER when sampling without replacement, whereas with function approximation the result is less clear. In tabular, the gap in performance between with and without replacement steadily increases and eventually Naive PER becomes nearly statistically better than uniform. With DQN (function approximation), larger batch sizes mostly result in ties, though Naive PER without replacement is the only method to always reach optimal on average.

Taken together, the results which use sampling without replacement suggest a minor benefit. It always helps in the tabular case, at times outperforming uniform replay, and with function approximation it mostly does not hurt performance.

5.2 Updating Transition Priorities

Another factor that can potentially limit the benefit of prioritization is non-informative and outdated priorities in the buffer. The priority of a transition is updated only when the transition is sampled. This means that at any given time the priority of almost all items in the replay buffer are outdated with respect to the current value function. We can update the priority of all transitions in the buffer by recomputing their TD error using the current value function estimate periodically.

We tested this idea in prediction setting in the 50 state chain. We compared the performance of Naive PER, EPER, and DM-PER recomputing the priorities every 10 and 1000 steps. Again we looked at tabular and NN representations with a two layer neural net of size 32 with ReLU activation for the latter. The buffer size is fixed to 8000, batch size to 8, and learning rate to 8^{-4} for tabular and 8^{-5} for neural net agents. Figure 5.3 summarizes the results. In short, we see no benefit from recomputing priorities in the function approximation settings and marginal benefit in the tabular case with Naive PER. Interestingly, for DM-PER recomputing too often, every 1000 steps vs every 10 steps, hurts compared to the default—updating only when a transition is first added or resampled. Note the over-generalization of Naive PER with neural nets is also not reduced more by up-to-date priorities.

As a final experiment in the chain problem, we investigate if combining sampling without replacement and recomputing priorities every 10 steps, together, can improve the performance of Naive PER. We conduct this experiment in the control chain problem and repeat the experiment for tabular Q-learning and DQN with a two layer neural net with 32 hidden units and ReLU activation with target refresh rate of 100. The buffer size is fixed to 10000 and batch size is 64, we select the learning rate over a range of values

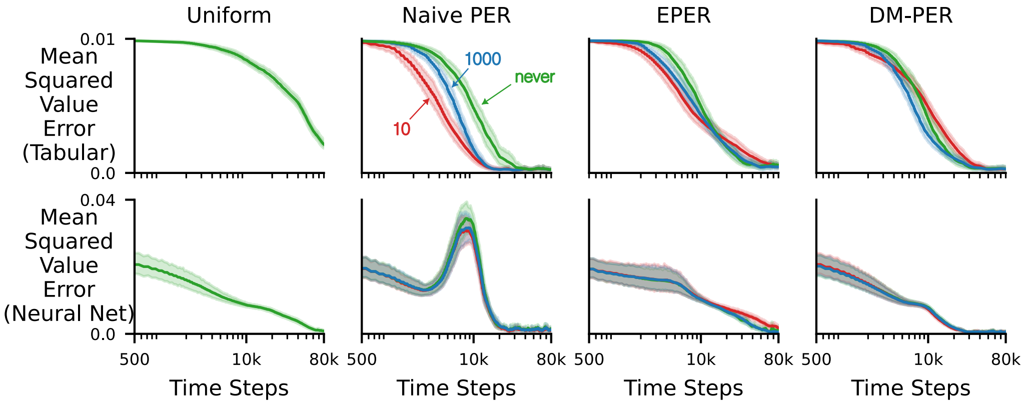

Figure 5.3: Recomputing priorities in chain prediction using Naive PER, EPER, and DM-PER with tabular (top) and NN (bottom) representations. Generally, recomputing does not help. Results averaged over 30 seeds; shaded regions are 95% bootstrap CI.

that attained the best average performance. As we see in Figure 5.4, in the tabular case, Naive PER with both modifications achieves the best performance, but barely more than either modification in isolation. Unfortunately, as expected, there is no clear benefit in the function approximation setting.

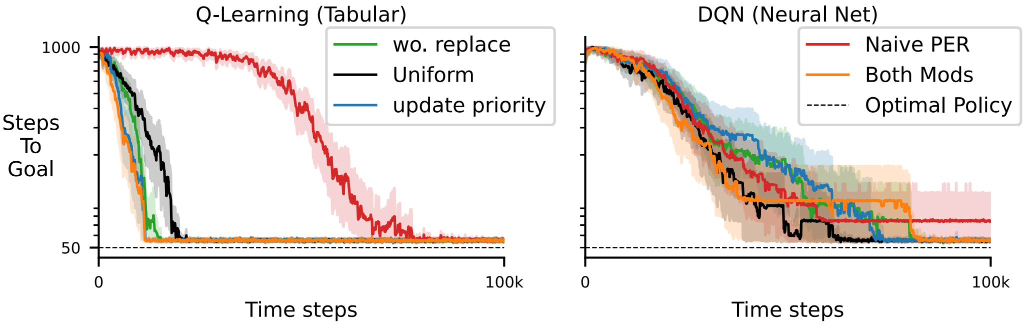

Figure 5.4: Combining recomputing priorities with without replacement sampling does not improve performance over just recomputing priorities or without replacement sampling in tabular chain control (left). Neither modification improve performance when used with neural networks (right). Results averaged over 50 seeds; shaded regions are 95% bootstrap CI.

5.3 Without-Replacement Experiment Details and Additional Results

Earlier, we showed sampling mini-batches with replacement can reduce the sample efficiency of Naive PER. Unlike Uniform replay, a high-priority transition can saturate the mini-batch in PER reducing data diversity. Here, we revisit experiments presented earlier in this chapter and investigate the effect of without replacement sampling across three PER variants, Naive PER, DM-PER, and EPER in prediction and control chain problems.

In the prediction setting, the hyperparameters are selected according to Table 4.1, testing a variety of batch sizes $[1, 8, 64, 256]$. We chose a representative learning rate, namely, 8^{-4} for tabular agents and 8^{-5} for neural network agents. Figure 5.5 shows the MSVE over time for the tabular prediction chain problem. All three prioritized variants outperform uniform replay with smaller batch sizes, this gap decreases with larger batch sizes. Sampling without replacement appears to improve the performance of Naive PER and DM-PER but not EPER.

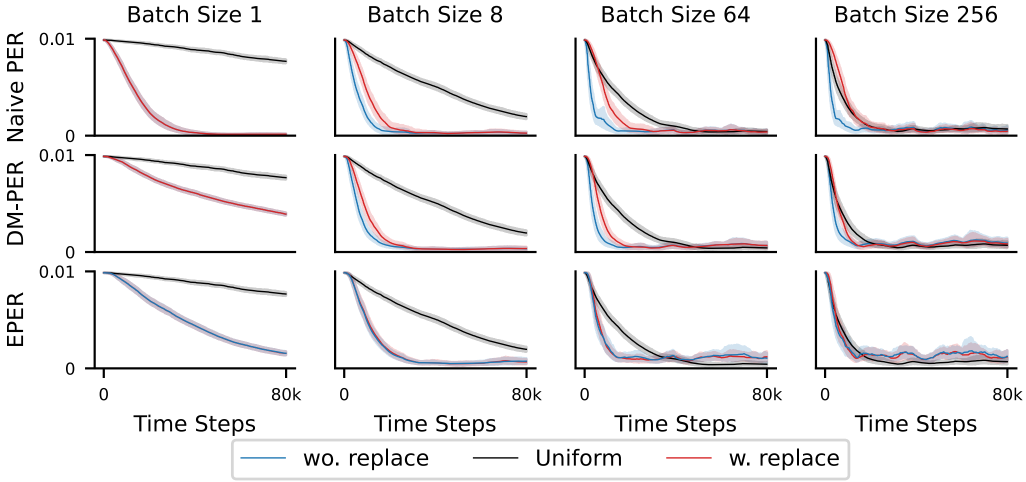

Figure 5.5: Sampling without replacement improves performance in the tabular prediction chain problem for Naive PER and DM-PER. Results averaged over 50 seeds with 95% bootstrap CI.

Results in prediction with neural network are different and show no clear performance improvement in without replacement sampling over regular PER variants (see Figure 5.6). Similar to the tabular setting, uniform replay performs as well as all prioritized variants with large batch sizes but DM-PER can outperform uniform replay with small batch sizes.

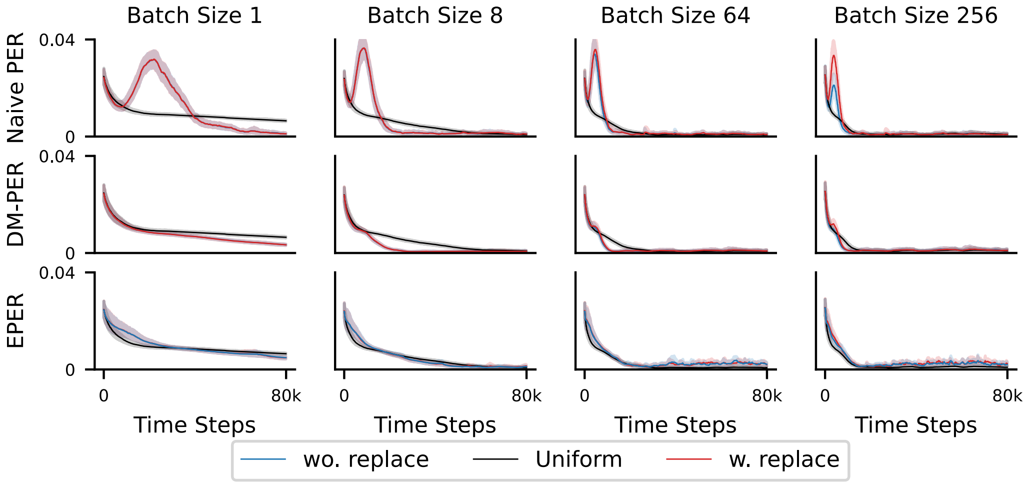

Figure 5.6: Sampling without replacement does not improve performance in the prediction chain problem under neural network function approximation. Results averaged over 50 seeds with 95% bootstrap CI.

In the control setting, We report learning curves for a tabular Q-learning agent and two neural network alternatives: DQN and EQRC. For all agents, we test 4 different batch sizes $[1, 8, 64, 256]$ and select the learning rate via maximizing over a range of values. Table 4.2 summarizes the details of hyperparameters used in these experiments.

Figure 5.7 shows the learning curves of PER variants in the tabular control setting. We see an improvement in sample efficiency of Naive PER and DM-PER when sampling without replacement, but there is no clear benefit in EPER. Sampling without replacement allows Naive PER and DM-PER to outperform uniform replay across batch sizes. Increasing the batch size improves learning speed across all agents, but uniform replay appears to benefit from increased batch size more than PER variants.

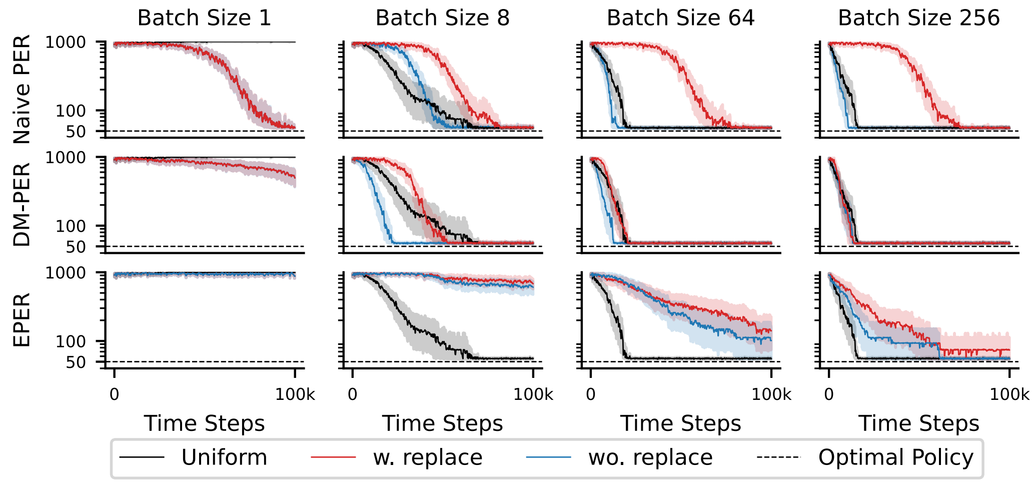

Figure 5.7: Uniform replay outperforms PER with tabular Q-learning in the chain problem. Sampling without replacement improves the performance of naive PER and DM-PER over uniform replay, but there is only marginal improvement with EPER. Increasing the batch size speeds up learning, but it improves uniform replay more than it does PER. Results are averaged over 50 seeds; the shaded region is 95% bootstrap CI.

Figures 5.8 and 5.9 show the learning curves of PER variants in DQN and EQRC, respectively. In both cases, uniform replay outperforms prioritized replay with larger batch sizes, while the performance is similar with a smaller batch size. Unfortunately, there is no consistent difference between sampling with and without replacement in both neural network settings.

The collection of experiments in this section suggests sampling without replacement can improve the sample efficiency of tabular agents but does not have a noticeable effect on neural networks in the chain problem.

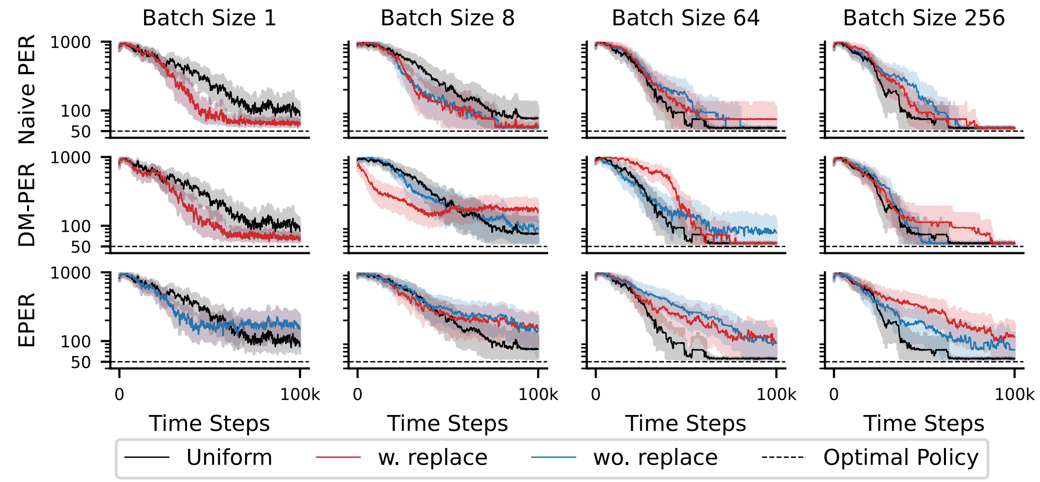

Figure 5.8: Sampling without replacement does not improve the performance of DQN agents with prioritized replay in the chain problem. In most cases, no PER variant outperforms uniform replay. While increasing batch size improves the sample efficiency of uniform replay, there is no gain with PER. Results are averaged over 50 seeds; the shaded region is 95% bootstrap CI.

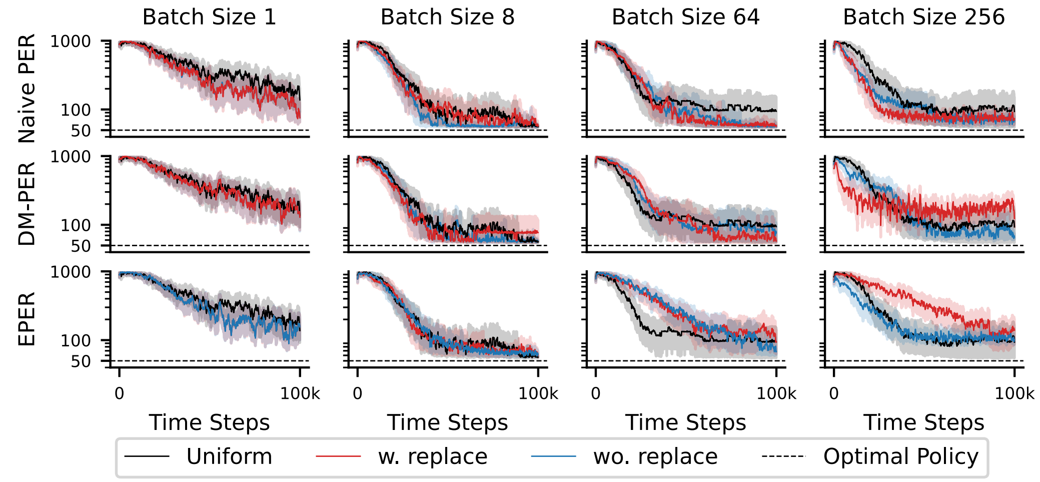

Figure 5.9: Sampling without replacement does not improve control performance in the chain task when using EQRC. No PER variant consistently outperforms uniform replay. Results are averaged over 50 seeds; the shaded region is 95% bootstrap CI.

5.4 Summary

In this chapter, we investigated two design choices that have not previously been explored in PER: sampling mini-batches without replacement and periodically updating the priorities in the buffer. Our results suggest that sampling without replacement can improve sample efficiency in prediction and control problems in the tabular setting but does not offer a significant improvement with neural networks. Similarly, keeping priorities up-to-date does not appear to improve sample efficiency with neural nets but can significantly improve performance in the tabular control setting. Finally, we explored combining the two modifications in the control problem and found that the combination does not further improve over either modification in the tabular setting.

Chapter 6

Investigating Prioritized Replay in Classic Control

We designed the chain experiments, explored in previous chapters, as an idealized problem to showcase the benefits of smarter replay; in this chapter, we consider slightly more complex, less ideal tasks. In the chain, we only saw clear advantages for prioritization in prediction and control with small batch sizes. The main question is whether these benefits persist or prioritization will perform worse, supporting the common preference for uniform replay in deep RL.

6.1 Classic Control Domains

In this chapter, we consider four episodic environments significantly more complex than the chain but small enough that we can use smaller NNs and extensive experimentation is possible. The first three environments, known as classic control, feature low-dimensional continuous states and discrete actions. MountainCar (Moore, 1990) and Acrobot (Sutton, 1995) are two control tasks where agents must manipulate a physical system to reach a goal at the end of a long trajectory.

In MountainCar, the goal is to drive an underpowered car up a hill in a simulated environment with simplified physics by taking one of three actions:

accelerate left, accelerate right, do not accelerate. The observations are the position and speed values of the car. The reward is -1 per step, and episodes terminate when the car crosses a threshold at the top of the hill with a reward of 0. In Acrobot, the agent controls a system of two linear links connected by a movable joint. The goal is to move the links by applying torque to the joint, such that the bottom part of the link rises to the level of its highest point, upon which the episode terminates with reward 0. The reward of all other transitions is -1 per step.

We also include Cartpole due to the unstable dynamics of the balanced position (Barto et al., 1983). The goal of an agent in the Cartpole environment is to balance a pole on top of a moving cart by accelerating the cart to either left or right. The reward is +1 per step if the pole is kept balanced. If it falls more than 12 degrees, the episode terminates, resetting the pole to its upright position. The episode cutoff length is 500.

Finally, we include the tabular Cliffworld (Sutton and Barto, 2018) because the reward for falling off the cliff is a large negative value that causes rare but large spikes in the TD error, which might showcase the benefit of EPER. Cliffworld is a grid world where agents start at a fixed state, pick any cardinal direction, and move to the corresponding neighbor state. The goal is to reach the final state on the opposite side of the starting state while avoiding a cliff near the optimal path. The reward is -1 per step except when falling off the cliff, where the agent is rewarded -100 and reset back to start. We set the discount factor $\gamma = 0.99$. The episode cutoff in MountainCar, Acrobot, and Cartpole is 500 steps, but there is no cutoff in Cliffworld.

6.2 Comparing Sample Efficiency in Classic Control Domains

In this section, we compare the sample efficiency of Naive PER with uniform replay in classic control domains. We also include DM-PER and a modified PER that recomputes the transition priorities and samples mini-batches without replacement. We then take a closer look at the performance of individual agents and find EPER can be more robust than other PER variants in the Cliffworld domain.

In this experiment, we use DQN with a two-layer network of size 64 with ReLU activation and a target refresh rate of 128. Batch size and buffer size are fixed to 64 and 10000 respectively and the learning rate is selected using a two-stage approach to avoid maximization bias (Patterson, Neumann, et al., 2023). First, each agent is run for 30 seeds sweeping over many learning rate parameter settings, and then the hyperparameter that achieved the best average performance is run for 100 new seeds. The list of hyperparameters is presented in Table 6.1.

	DQN agents
Learning rate	$[4^{-8}, 4^{-7}, 4^{-6}, 4^{-5}, 4^{-4}, 4^{-3}, 4^{-2}]$
Adam optimizer β_1	0.9
Adam optimizer β_2	0.999
Batch size	64
Buffer size	10000
Network size	2×64 dense network with ReLU activation
Target refresh	128
Exploration ϵ	0.1
Training time	100000

Table 6.1: Hyperparameters of classic control experiments

We include *Modified PER*, which combines Naive PER with without-replacement sampling and recomputes the priorities every 10 steps. Figure 6.1 summarizes the results. Unsurprisingly, prioritization does not improve the sample efficiency over uniform replay in any of the four domains.

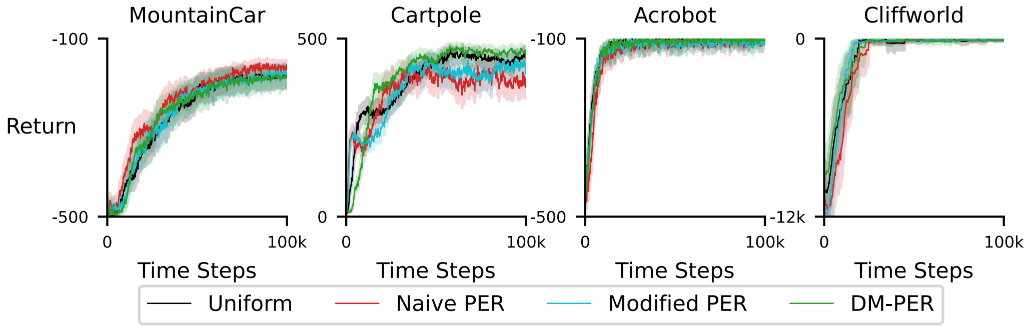

Figure 6.1: Performance of DQN replay agents on classic control problems. No clear benefit for prioritization. Results averaged over 100 seeds; shaded regions are 95% bootstrap CI.

Looking closely at the learning curve for Cliffworld in Figure 6.1, we see a small blip in the performance with uniform replay. Recall that we suspected that EPER, might perform well in this MDP due to outlier rewards when the agent falls off the cliff. Average learning curves can hide the structure of individual runs, so we plotted all the runs individually for each method in Figure 6.2. Here we see DQN with uniform replay periodically performs quite poorly, even late in learning. This is true to a lesser extent for DM-PER, Naive PER, and Modified PER. Interestingly, Naive PER variants based on EPER appear substantially more stable with fewer collapses in performance.

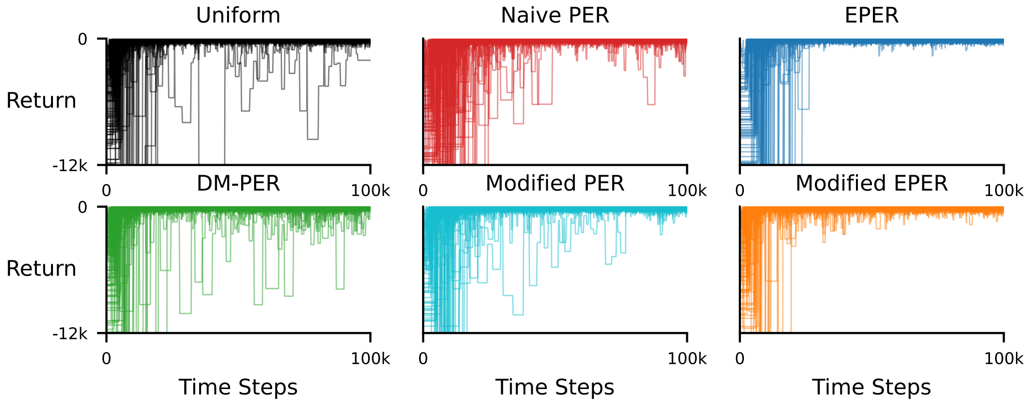

Figure 6.2: Performance of 100 individual runs in the Cliffworld shows performance dips using uniform replay, Naive PER, and DM-PER. EPER-based methods appear to have more stable performance.

6.3 Summary

In this chapter, we investigated PER in slightly more complex control problems. We focused on three classic environments, MountainCar, Acrobot, and Cartpole, covering a range of control problems and the Cliffworld domain where outliers in reward may prove problematic for PER. We found that under fair hyperparameter selection, there was no sample efficiency gain for PER over uniform replay. Finally, a closer look at individual runs in the Cliffwold domain revealed drops in performance even late in training, possibly due to outliers in the rewards. Compared with other replay methods, individual EPER runs showed robust performance and fewer performance drops, supporting our hypothesis that EPER can be successful in noisy TD error settings.

Chapter 7

Conclusion

In this thesis, we conducted a series of carefully designed experiments with prioritized replay under tabular and neural network settings. We found that prioritization combined with non-linear generalization can overestimate values during early learning. It appears that a combination of bootstrapping and neural network generalization is the reason behind this overestimation. Furthermore, we showed in a simple chain domain, several variants of PER outperform i.i.d replay in the prediction setting but have poor sample efficiency in control. Unsurprisingly, no variant of PER improves upon i.i.d replay in classic control domains.

We introduced EPER as a simple modification prioritizing transitions according to a learned estimate of the expected error inspired by gradient TD methods. We showed that EPER can be more robust in noisy reward domains and perform more reliably than PER or i.i.d replay in Cliffworld. Finally, we explored two design decisions in PER, recomputing outdated priorities and sampling batches without replacement, discovering that these additions can improve PER in the tabular setting but have little to no effect when using neural networks.

7.1 Future Work

This section identifies and discusses several promising directions for future work in experience replay. A core insight from this thesis is the impact of generalization on value propagation. Unlike the tabular setting, where bootstrapping drives value propagation across states, neural networks generalize aggressively, updating the values across most states. A follow-up study may investigate ways to restrict or direct neural nets to generalize more favorably with PER.

One promising direction is investigating other prioritization schemes. Instead of prioritizing transitions by TD error, one could randomly sample transitions but reweight them according to TD error. This scheme may improve data diversity, whereas PER may saturate mini-batches with only a small subset of high TD error transitions.

Beyond prioritization, there are many unexplored ways to replay transitions. One could maintain a small buffer of prototypes or representative transitions to create mini-batches alongside the online experience. In addition to a smaller memory footprint, we speculate that a small buffer of prototypes could improve neural network learning by reducing the amount of redundant data used to train the network while having diversified mini-batches to avoid over-generalization.

Replay in biological systems is usually a fast sequential reactivation of memories (Wittkuhn et al., 2021). One possible next step is finding sequences that lend themselves to neural network generalization. Sequential replay may be augmented with jumps in the sequence to avoid over-sampling specific transitions.

This thesis introduced the EPER algorithm that prioritizes transitions according to expected TD error. Many of the design decisions in EPER remain open questions. Conditioning the expectation on only the state in control problems and regularizing the secondary estimator similar to the EQRC algorithm are stimulating directions for future exploration. Another exciting next step is an investigation of EPER with gradient TD methods because they provide the expected TD error estimator without any additional computation.

Model-based RL agents use search control to select states for planning. Insights gained from the interaction of experience replay and neural network generalization can guide the development of search control methods to support the training of neural networks with simulated transitions from a model.

References

Anand, N., & Precup, D. (2024). Prediction and control in continual reinforcement learning. *Advances in Neural Information Processing Systems*, *36*.

Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., McGrew, B., Tobin, J., Abbeel, P., & Zaremba, W. (2018). *Hindsight Experience Replay.*

Barto, A. G., Sutton, R. S., & Anderson, C. W. (1983). Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, *SMC-13*(5), 834–846.

Fedus, W., Ramachandran, P., Agarwal, R., Bengio, Y., Larochelle, H., Rowland, M., & Dabney, W. (2020). Revisiting Fundamentals of Experience Replay. *International Conference on Machine Learning.*

Foster, D. J., & Wilson, M. A. (2006). Reverse replay of behavioural sequences in hippocampal place cells during the awake state. *Nature*, *440*(7084), 680–683.

Fu, Y., Wu, D., & Boulet, B. (2022). Benchmarking sample selection strategies for batch reinforcement learning.

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., & Silver, D. (2018). Rainbow: Combining improvements in deep reinforcement learning. *Proceedings of the AAAI conference on artificial intelligence*, *32*(1).

Hong, Z.-W., Chen, T., Lin, Y.-C., Pajarinen, J., & Agrawal, P. (2023). *Topological Experience Replay.* arXiv: 2203.15845.

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

Igata, H., Ikegaya, Y., & Sasaki, T. (2021). Prioritized experience replays on a hippocampal predictive map for learning. *Proceedings of the National Academy of Sciences*, *118*(1), e2011266118.

Jaderberg, M., Mnih, V., Czarnecki, W. M., Schaul, T., Leibo, J. Z., Silver, D., & Kavukcuoglu, K. (2016). Reinforcement learning with unsupervised auxiliary tasks. *arXiv preprint arXiv:1611.05397*.

Kingma, D., & Ba, J. (2015). Adam: A method for stochastic optimization. *International Conference on Learning Representations*.

Kobayashi, T. (2024). Revisiting experience replayable conditions. *arXiv preprint arXiv:2402.10374*.

Kumar, R., & Nagaraj, D. (2023). *Introspective Experience Replay: Look Back When Surprised*. arXiv: 2206.03171.

Lee, S. Y., Sungik, C., & Chung, S.-Y. (2019). Sample-Efficient Deep Reinforcement Learning via Episodic Backward Update. *Advances in Neural Information Processing Systems*, *32*.

Li, A. A., Lu, Z., & Miao, C. (2021). Revisiting prioritized experience replay: A value perspective. *arXiv preprint arXiv:2102.03261*.

Li, M., Huang, T., & Zhu, W. (2022). Clustering experience replay for the effective exploitation in reinforcement learning. *Pattern Recognition*, *131*, 108875.

Lin. (1991). Programming robots using reinforcement learning and teaching. *Proceedings of the Ninth National Conference on Artificial Intelligence - Volume 2*, 781–786.

Lu, C., Ball, P., Teh, Y. W., & Parker-Holder, J. (2024). Synthetic experience replay. *Advances in Neural Information Processing Systems*, *36*.

Ma, G., Li, L., Zhang, S., Liu, Z., Wang, Z., Chen, Y., Shen, L., Wang, X., & Tao, D. (2023). *Revisiting Plasticity in Visual Reinforcement Learning: Data, Modules and Training Stages*. arXiv: 2310.07418.

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

Mohammad Panahi, P., Patterson, A., White, M., & White, A. (2024). Investigating the interplay of prioritized replay and generalization. *Reinforcement Learning Conference.*

Moore, A. W. (1990). *Efficient memory-based learning for robot control* (tech. rep.). University of Cambridge.

Moore, A. W., & Atkeson, C. G. (1993). Prioritized Sweeping: Reinforcement Learning with Less Data and Less Time. *Machine Language*, *13*(1), 103–130.

Ólafsdóttir, H. F., Bush, D., & Barry, C. (2018). The role of hippocampal replay in memory and planning. *Current Biology*, *28*(1), R37–R50.

Pan, Y., Zaheer, M., White, A., Patterson, A., & White, M. (2018). *Organizing Experience: A Deeper Look at Replay Mechanisms for Sample-based Planning in Continuous State Domains.* arXiv: 1806.04624.

Patterson, A., Neumann, S., White, M., & White, A. (2023). *Empirical Design in Reinforcement Learning.* arXiv: 2304.01315.

Patterson, A., White, A., & White, M. (2022). A generalized projected bellman error for off-policy value estimation in reinforcement learning. *The Journal of Machine Learning Research*, *23*(1), 145:6463–145:6523.

Peng, J., & Williams, R. J. (1993). Efficient learning and planning within the dyna framework. *Adaptive Behavior*, *1*(4), 437–454.

Schaul, T., Horgan, D., Gregor, K., & Silver, D. (2015). Universal Value Function Approximators. *Proceedings of the 32nd International Conference on Machine Learning*, 1312–1320.

Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2016). Prioritized Experience Replay. *International Conference on Learning Representations.*

Singer, A. C., & Frank, L. M. (2009). Rewarded outcomes enhance reactivation of experience in the hippocampus. *Neuron*, *64*(6), 910–921.

Sun, P., Zhou, W., & Li, H. (2020). Attentive Experience Replay. *Proceedings of the AAAI Conference on Artificial Intelligence*, *34*(04), 5900–5907.

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

Sutton, R. S. (1995). Generalization in reinforcement learning: Successful examples using sparse coarse coding. In D. Touretzky, M. Mozer, & M. Hasselmo (Eds.), *Advances in neural information processing systems* (Vol. 8). MIT Press.

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

Sutton, R. S., Maei, H., Precup, D., Bhatnagar, S., Silver, D., Szepesvári, C., & Wiewiora, E. (2009). Fast gradient-descent methods for temporal-difference learning with linear function approximation. *382*, 125.

Van Hasselt, H. P., Hessel, M., & Aslanides, J. (2019). When to use parametric models in reinforcement learning? *Advances in Neural Information Processing Systems*, *32*.

Wang, H., Miahi, E., White, M., Machado, M. C., Abbas, Z., Kumaraswamy, R., Liu, V., & White, A. (2024). Investigating the properties of neural network representations in reinforcement learning. *Artificial Intelligence*, 104100.

Wittkuhn, L., Chien, S., Hall-McMaster, S., & Schuck, N. W. (2021). Replay in minds and machines. *Neuroscience & Biobehavioral Reviews*, *129*, 367–388.