Tiered Prioritized Replay Buffer

Arya Tschand, Ikechukwu Uchendu, Madison Davis

April 2025

Abstract

Off-policy reinforcement learning (RL) algorithms maintain a replay buffer to store experiences collected from the agent during training. Since random-access memory (RAM) capacity is limited, the replay buffer is typically set to a fixed size that determines the maximum number of experiences that it can hold at any given time. In computationally expensive environments such as robotic simulations, 3D video games, and language modeling, the memory used to store experiences can grow quickly, thus limiting the number of experiences that can be held in the replay buffer. If only a small number of experiences are held in the replay buffer, learning may become unstable. The question we try to answer in this work is: *how can we store as many experiences in the replay buffer as possible given a fixed memory budget*? Our approach to answer this question is quite simple: quantize the less interesting transitions in the replay buffer and store the important transitions in full precision. Following the principles of L1, L2, and L3 cache from computer architecture, we propose a multi-tiered replay buffer where high priority transitions are held in the top layer in full precision, and the remaining transitions are stored in lower precision according to their priority. While our results on the Hopper v5 environment showcase that a baseline replay buffer (RB) outperforms both the state-of-the-art priority replay buffer (PRB) and our new prioritized cache replay buffer (PCRB) in episodic return, our work shows promise. Despite adding noise from multiple buffer tiers of different quantization schemes, experiments using the PCRB still show steady gains in episodic returns. Compared to the baseline RB, the PCRB experiments are also more stable. Lastly, our PCRB outperforms the state-of-the-art PRB in episodic return. We hope to explore environments that have low memory budgets and reward prioritized experience sampling to show the real-world implications of out work.

1 Introduction

In RL algorithms, an agent finds themselves in some environment and must learn how best to navigate their surroundings. The goal is to accumulate a high reward over time (choosing 'good' actions to lead to desirable outcomes). RL methods can be broken up by many categories, for example, based on policy (either 'on-policy' where they examine data as they go or 'off-policy' where they store results to use in calculations for later), action space (can be discrete, such as 'turn left', or continuous, such as 'rotate x degrees, where x is a real number from 0-30'), and environment size (small, medium, or large number of states and actions).

For the purposes of this paper, we're concerned with tackling very complex problems, namely where an actor has to navigate a large environment with many states and actions, and those actions are continuous. We also opt for an off-policy method so that we can explore the following

Name	Desc	Policy	Action Space	Env Size
PG	Maximize reward via gradient	On-Policy	Discrete	Small
NPG	PG + Fisher matrix (natural gradient)	On-Policy	Continuous	Small-Med
TRPO	NPG + trust region (KL-divergence constraint)	On-Policy	Continuous	Med-Large
PPO	TRPO + clipping (simplified)	On-Policy	Continuous	Med-Large
DeepQN	Maximize reward via Q-learning	Off-Policy	Discrete	Small-Med
DDPG	Maximize reward via deterministic PG	Off-Policy	Continuous	Medium
TD3	DDPG + delays and noise for exploration	Off-Policy	Continuous	Med-Large
SAC	Maximize reward *and* entropy	Off-Policy	Continuous	Large

Table 1: List of Popular RL Algorithms

question: can we get the agent to a higher reward in the same number of steps by storing the data in a special way? As we see from the table below, of the many RL algorithms, Soft-Actor Critic (SAC) methods appear to be what we're looking for.

SAC methods are meant guide an agent/object through a *large* environment with *continuous* states and actions. We'll describe how SAC works in more detail in the next section. For now, one noteworthy part of how SAC is built is that the experiences seen by its agent are stored as tuples of data (old state, new state, action to get to new state, our reward for performing the chosen action from the old state, did the agent stop/end, and any other info) into a data structure called a replay buffer. Some of the experiences aren't as important or 'surprising' as others. An experience is surprising if, given we're at some state, our specified action gives us a reward that is much better than the mean reward across all the actions we could have taken. If we want to store more experiences so our agent learns more and gets better, we may want to prioritize our surprising experiences by maintaining their level of precision, whereas less important experiences can get quantized so as to make room for more experiences. This implementation models the computer hardware-equivalent of caches. We call our methodology the prioritized-cache replay buffer (PCRB).

2 Related Work

Replay Buffers Replay buffers have been thoroughly researched due to their widespread applications in many Reinforcement Learning methods (not just for SAC). A replay buffer is a data structure that functions essentially like a list for storing data. Replays buffers were first introduced in 2015 to develop the DeepQN method [6]. Extended work has sought to enhance the replay buffer by pruning unnecessary data [1] and sampling more-surprising entries, a convention known as prioritization ([5], [9]). Beyond just picking data we think is relevant, replay buffers can also harness the opportunity to reformat less-helpful data so that it takes up less space or a smaller range. We know it's possible to use compressed versions of replay buffers for training [11], and we've seen quantization done across the entire training data [2]. This latter approach, in its simplified form, was able to see substantial increases in stability.

Computer Architecture Foundations: Multi-Level Caching and Quantization In computer architecture, caching exploits both temporal and spatial locality to keep the most frequently or recently accessed data close to the processor, thereby reducing access latency and easing bandwidth demands on slower memory tiers [4]. Hierarchies of small, fast caches (L1), larger moderately fast caches (L2), and even broader but slower caches (L3) trade off capacity against speed: "hot" data is retained in higher levels, while less critical lines are demoted or evicted based on policies like Least Recently Used (LRU) [10]. Parallel to this, quantization reduces the bit width of stored data—mapping full-precision values to lower-precision formats such as INT8 or FP16—to shrink memory footprint and boost bandwidth, at the acceptable cost of some numerical approximation [3]. Modern accelerators often support multiple precisions and dynamically choose mappings that satisfy accuracy constraints while maximizing effective on-chip capacity. Applied to replay buffers, these principles motivate a multi-tiered storage strategy: the most surprising transitions (highest "priority") remain in full precision in a top-tier cache, whereas transitions of decreasing importance are stored in progressively quantized, larger lower tiers.

Our Contributions Our contribution now builds off of all of this previous work. We will enhance those papers' contributions for replay buffers by *combining* quantization methods and prioritization techniques based on surprise levels. To accomplish this, we will design a novel replay buffer that is analogous to computer hardware cache systems. We will test this approach using the SAC method.

To date, there are no known, widely-cited papers that try this approach, especially for SAC methods that can benefit from saving memory for a dense environment.

3 Preliminaries

In this section, we'll describe some more detailed background knowledge on topics that we utilize in this paper. Namely, we focus on two categories: priortiized experience replay and soft actor-critic methods.

3.1 Prioritized Experience Replay

By default, off-policy RL methods use replay buffers to store experiences that have been observed over the course of training. However, many of these experiences may come from areas of the state space that have already been learned by the agent, making continuous sampling of them for gradient updates less useful than other experiences. Prioritized Experience Replay [7] addresses this problem by weighting the probability of an experience being sampled by its temporal difference error [8].

When a new experience is observed, it is added to the replay buffer with maximum priority P. This ensures that every experience is processed at least once. After an experience is replayed, its priority is updated with the latest TD error. The general probability of sampling a transition i is:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

Here α represents how much more weight we give to samples with high priority. When $\alpha = 0$, we have uniform sampling, and when $\alpha > 1$, high priority experiences are weighted more heavily.

Since prioritized experience replay affects the sampling distribution of experiences in the replay buffer, we must use importance sampling to correct for this bias using weights:

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^{\beta}$$

The parameter β affects how much correction is applied. If $\beta = 0$, no correction is done, but if $\beta = 1$, we are fully correcting for the bias. In practice though, we do not need to perform full correction during early stages of training. Since we are also learning representations of the state space with our actor and critic networks, it would further affect stability if we aggressively applied importance sampling correction in early parts of training. To mitigate this, the parameter β is typically slowly increased from a lower number such as 0.4, reaching 1 near the end of training when the representation has converged.

3.2 Soft Actor-Critic Methods

Here, we'll describe in more detail how SAC works. In SAC, we have an agent that is navigating some environment made up of a set of states S and actions A. The agent can move from one state $s \in S$ to another state $s' \in S, s \neq s'$ by performing some allowed action $a \in A$. We won't run SAC forever. Rather, we'll run it for a certain number of time-steps $t = \{0, ..., T\}$. We'll notate s_t and a_t as the state we're in and the action we take from that state at time t, respectively.

How do we know what action to take when the agent is at $s \in S$? We use something called the policy function π. For notation, $\pi(a|s)$ tells us the probability of taking action a from state s, and $\pi(\cdot|s)$ just means we're sampling an action from s (ie. $a \sim \pi(\cdot|s)$). We'll describe in the Actors sub-section below how π is defined. For now, just know its a function we use to determine our next action.

As a quick recap, at time step t with state s_t, the agent will take action a_t based on $\sim \pi(\cdot|s)$. This will land us in a new state s_{t+1}. After they take action a_t, the agent gets feedback from the environment in the form of a reward $r_t(s_t, a_t)$. Typically, the reward function outputs a number based on what state we were in and the action we took from that state, where higher numbers give higher rewards.

We've now collected a good amount of data: states, actions, and rewards. Because SAC is off-policy, it likes to save some of the experiences its collected over time. Therefore, we'll store the experience (s_t, a_t, r_t, s_{t+1}) into a data structure known as the replay buffer $\mathcal{D}$.

3.2.1 Critics

After several timesteps, we'll begin to employ our first-main component of SAC: critics. The idea of critics is to 'challenge' the current methodology of how we've been picking our trajectory of states and actions. Critics calculate what is known as the Q-function, which tells us the value of having been in state s and taking action a from there. If the critic gives us a low Q-value score, that means our actor should try to change up its strategy (it was heavily criticized).

The Q-function can be written in many ways, but critics implement it using a neural network. If we have multiple critics, we'll distinguish them by their neural network weights. We'll label an arbitrary critic's weights as θ_c. Therefore, critic c will calculate $Q_{\theta_c}(s, a)$.

The critic's parameters, as like most neural networks, will start out random. However, we'd like to train the network so its parameters are more tuned to the task at hand. To train the critic, we'll examine the difference between its network's outputted value for some time step t and what we desired it to output (ie. the 'optimal' critic neural network). The optimal critic would output what we call the target value y_t for a given time step. To compute y_t, we'll sample some of the experiences we got from the replay buffer $\{(s_t, a_t, r_t, s_{t+1}),\ (s_{t+1}, a_{t+1}, r_{t+1}, s_{t+2}), ...\}$ and use a Bellman backup equation. If you're curious, the equation looks like this:

$$y_t = r_t + \gamma \left(\min_c Q_{\bar{\theta}_c}(s_{t+1}, a_{t+1}) - \alpha \log \pi(a_{t+1} \mid s_{t+1}) \right)$$

y_t tells us the target Q-value for time step t, r_t is the reward we received at time step t, γ is a discount factor, and $Q_{\bar{\theta}_c}(s_{t+1}, a_{t+1})$ is the output of the target critic network c given the next state s_{t+1} and next action a_{t+1}. $Q_{\bar{\theta}_c}(s_{t+1}, a_{t+1})$ is essentially a lagged-behind version of $Q_{\theta_c}(s_{t+1}, a_{t+1})$; its parameters will be a few steps behind in terms of updates. SAC does this to stop overestimation bias and allow for more stable updates. If you want the mathematical explanation for how the target parameters get updated, it typically uses the following update, where τ is some probability value $0 \leq \tau \leq 1$.

$$\bar{\theta}_c \leftarrow \tau\theta_c + (1 - \tau)\bar{\theta}_c$$

Now that we have the target value y_t, we can use it to train the critic. For a given critic c, we calculate the mean squared error (MSE) between the target value y_t and the critic's current estimate $Q_{\theta_c}(s_t, a_t)$. In the equation below, the expectation is once again used because we will not know what exact experience we'll be dealing with at timestep t until we're actually going through a trajectory. Therefore, in theory, we're using expectation to denote a sampling of experiences from the replay buffer $\mathcal{D}$. This procedure is also called the loss function of critic c. It encourages our current critic to try and match the target value y_t. Once the loss function is calculated, we can use backpropagation to update its parameters.

$$\mathcal{L}_{Q_c} = \mathbb{E}_{(s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}} \left[\left(Q_{\theta_c}(s_t, a_t) - y_t \right)^2 \right]$$

As the critics are training, we can use them to simultaneously improve the actor.

3.2.2 Actors

We now move to the actors. The actors are the ones who define and develop the policy distribution $\pi(a|s)$, which in turn helps to move around the agent. Similar to the critic, the actor constructs π as a neural network. We'll distinguish our actors' neural network parameters as θ_π. The network will take in s and will output a Gaussian distribution over all actions. The distribution has mean $\mu(s)$ and standard deviation $\sigma(s)$. After we have the Gaussian distribution, we draw a sample from it and put it through a squashing function (usually tanh) to ensure the action remains within its original range of values. In the end, we'll end up with an action a to take.

We'll want to fine-tune the actor's neural network parameters so that the network behaves just like what we want π to do (ie. output distributions over our actions based on our state and general environment setup). This requires training. We'll train by calculating a loss function.

$$\mathcal{L}_\pi(\theta_\pi) = \mathbb{E}_{s_t \sim \mathcal{D}} \left[\mathbb{E}_{a_t \sim \pi(\cdot|s_t)} \left[\alpha \log \pi(a_t|s_t) - Q_{\theta_c}(s_t, a_t) \right] \right]$$

The first term represents the entropy of our policy. α is the weight we assign to entropy: a higher value means we want to encourage more exploration of possibilities. The second term represents the value that our critic assigns to the action a_t taken from state s_t. A higher Q-value means a smaller loss function and therefore a smaller update to the neural network. In other words, the critic thinks we found a good route, and we're incentivized to continue down that route and exploit it (keep the neural network more or less the same, with minor tweaks). Again, we don't know what states and actions we'll be at at time step t, so we'll sample them in expectation. We'll use the loss function for backpropagation on the actor, and we'll do this procedure for every actor we have.

3.2.3 Rinse and Repeat

As the actor trains, its resulting policy function is used by the agent to move around in the environment. The agent's experiences help to train the critic, and the critic's updated Q-function subsequently helps to train the actor. This cyclical training process continues for as long as the agent navigates the environment. Across all of this, the ultimate goal is to maximize two things: total cumulative reward and entropy. We want to maximize reward by convention; higher numbers indicate better rewards. The assignment of a high reward is based on the application (for example, an agent trying to find the exit to a maze might get higher rewards as it nears the exit). We also focus on entropy because we want to encourage exploration in this very large and continuous-spaced environment. The cumulative reward, which is a function $J(\pi)$ based on our policy function π, is written like this:

$$J(\pi) = \sum_{t=0}^{T} \mathbb{E}_{(s_t,a_t)\sim\rho_\pi} \left[r(s_t, a_t) + \alpha\mathcal{H}(\pi(\cdot|s_t)) \right]$$

Let's intuitively explain this equation.

1. We want a cumulative result, so we'll sum the reward and entropy over all timesteps $\sum_{t=0}^{T}$.
2. For a single timestep, we can calculate the reward of being in state s_t and having taken action a_t from that state using a reward function $r(s_t, a_t)$.
3. Similarly, we can calculate at a given time step t the entropy as $\mathcal{H}(\pi(\cdot|s))$. The 'dot' in $\pi(\cdot|s)$ simply means we're considering all possible actions when in state s_t.
4. The expectation operator $\mathbb{E}_{(s_t,a_t)\sim\rho_\pi}$ tells us to average this result over all possible state-action pairs we could be at for time t. We won't know where we'll be at timestep t because the agent has a lot of options and trajectories (ie. sequences of state-action pairs) it could explore on its own. Therefore, we'll just average it out in expectation.

4 Model: Setup

The entire codebase can be accessed through this public github repository. The folders are set up in the following manner. The *eval* folder holds the result of the runs when we execute an *sac* file. The *sac* files holds all the *sac* methods we will test, along with a series of profiling files for each *sac* method. There are four *sac* files. The first uses torch rl's baseline replay buffer to store experiences (no prioritization nor quantization). The second uses torch rl's prioritized replay buffer. Specifically, it is called the TensorDictPrioritizedReplayBuffer (prioritization by storing in a LazyTensorStorage, but no quantization). The third implements a prioritized cache replay buffer, or pcrb (prioritization and quantization, analogous to a cache). The final is the same as the third but does annealing on the replay buffer's hyperparameters so as to allow exploitation of surprising results in later stages.

The process by which we profile any of these four *sac* files is shown in the right-hand side of Table 2. To profile the pcrb, for example, first run sac_pcrb_profiling, which outputs the result to profile_log_pcrb, a csv file. From here, to visualize the results in this csv file, run plot_profiles_pcrb, which will display a pop-up image titled profiling_summary_pcrb.

Experiment setup	Profiling setup

`repo`
├─ `eval`
│ └─ `runs`
│   └─ `...`
└─ `sac`
   ├─ `sac_torch_baseline`
   ├─ `sac_torch_rb`
   ├─ `sac_pcrb`
   ├─ `sac_pcrb_anneal`
   └─ `pcrb_torch_storage`
	`sac`
	├─ `sac_..._profiling`
	├─ `profile_log_...`
	├─ `plot_profiles_...`
	└─ `profiling_summary_...`

Table 2: List of Experiments

Let us now go into detail for how sac_pcrb works. The sac_pcrb file is based on sac_torch and works as follows:

1. In sac_pcrb, we initialize actors, Q-networks, optimizers, entropy equations, and environment parameters (software and hardware). We also create a replay buffer identical to the one utilized in sac_torch, denoted as a TensorDictPrioritizedReplayBuffer. It takes in two notable parameters, alpha and beta. Alpha ranges from 0-1 and determines how much prioritization we conduct among our samples in the replay buffer: a value of 0 means we sample uniformly,

and a value of 1 means we sample entirely based on the TD-errors of the experiences. Sampling via prioritization introduces bias: we might sample more experiences than others even though we may not see those experiences as much in expectation. This is where beta comes into play. Beta also ranges from 0-1 and corresponds to importance sampling. It corrects the gradient during weight-updates so that the changes are less biased. A value of 0 means the gradient update is fully based on the prioritization and therefore is biased. A value of 1 means the gradient weight update is not at all based on the prioritization. In practice, one often sets beta low to learn from biased samples initially and then as training continues anneals it (ie increases its value) to 1. For sac_pcrb, the default alpha and beta are 0.7 and 0.9, respectively.

In addition to these two parameters, we modified its storage parameter to utilize our custom storage class, as defined in pcrb_torch_torage. We describe it in more detail below.

After these two changes, everything is the same as sac_torch: an environment with certain seeding is made, and then the agent explores the environment, calculates its TD errors, applies a loss function, updates its neural networks, and repeats. Both sac_pcrb and sac_torch use double Q-learning to improve stability and batches its environment experiences so that training can be parallelized.

2. Before moving on, it is noteworthy to describe how TensorDictPrioritizedReplayBuffer samples experiences from its storage argument. It instantiates a PrioritzedSampler class, which utilizes a data structure called a SegmentTree to take the experiences in the storage and 'distribute' them over a graph-like format. Any experience's place in the graph is based on its TD-error. SegmentTrees allow for quick calculations of cumulative probabilites (and subsequently sampling) as well as modifications: the insertion and deletion of an interval of n items takes on average $O(logn)$. Moreover, the SegmentTree provides a more nuanced and less parameterized sampling system than, for example, uniform sampling. This latter scenario would encompass uniformly selecting an experience from within a given tier, and that tier is chosen with some hyperparamter probability p_{ti} for all tiers $\{p_{t0}, p_{t1}, p_{t2}\}$ s.t $\sum_i p_{ti} = 1$.

3. pcrb_torch_storage defines our a class called TieredCacheStorage. It's based on torch's LazyTensorStorage in that it stores data as TensorDicts (we'll describe our data-setup in the Experiments section, as the data one runs on is typically dependent on how the environment is set up. For the time being, simply understand that the data we deal with are just string keys and integer values in a dictionary). Now, instead of storing these TensorDicts in one long list (LazyTensorStorage), TieredCacheStorage has 3 tiers. Each tier is a min-heap organized by the TensorDict's TD errors. Each tier has a certain capacity for the number of TensorDicts it can hold, and each tier also has a quantization level: higher tiers will not quantize as much as lower tiers, analogous to cache sysems. By default, there are 3 tiers of caches that quantize (float32, float16, float8_e4m3fn) the experiences' observations (current observations, next observations). One can easily change these parameters in the initializion of the TieredCacheStorage. When one inserts a new data point, the class will look at its TD error field and place it in the appropriate tier, moving any other points that have smaller TD errors lower and lower until they are eventually pushed out of memory (recall: the goal is to focus on the 'surprising' TD errors and keep those higher values closer to the top). The fields that get quantized at every level include the states/observations (current state and next state) and the action. The quantization is based on each data point's full precision level. For example,

if the action comes in as a 32-bit integer, tier 0 will keep it as such, tier 1 will convert it into a 16-bit integer, and so forth. This methodology means that each data point will not have full division-level precision at each subsequent level because we are only quantizing some of the data fields the amount we desire (for example, half-precision) and not all data fields.

The sac_pcrb_anneal file is by definition identical to sac_pcrb but with the exception of an annealing function. The beta value starts off at 0.4 and is annealed to 1.

Parts of this model are left intentionally vague for experimentation. We will test variations in our model's sampler, the number of time steps we run it for, and the breakdown of tier cache sizes. See the next section for a deeper dive into all Experiments we will run: for our model specifically, we test these variations in Experiments 3 through 5.

5 Experiments

We'll now describe the experiments we run. See Table 3 for the full outline. The experiments are relatively aligned with the file structure as described above. Experiment 1 deals with a torch rl's baseline replay buffer. Experiment 2 deals with torch rl's tensor-dictionary prioritized replay buffer but with no quantization. Experiment 3 changes Experiment 2's storage mechanism to a TieredCacheStorage so that one can quantize, analogous to caches. We put our experiences in the tiers based on their pre-defined sizes and what we are quantizing. For example, in the table below, Experiment 3 states that the cache size is based on a linear division of three tiers: 32, 16, and 8. This means that quantizing our observation will be converted from a float32 to a float32 in tier 0, float16 in tier 1, and float8 in tier 2. We assume the same buffer size and memory space as Experiments 1 and 2, and we also assume an identical data composition, albeit with some values now being quantized. Experiment 4 changes Experiment 3 by adding annealing to the tensor-dictionary prioritized replay buffer parameters. We will answer several studies across these experiments.

1. The first study answers the question, ”How much benefit does a non-tiered prioritized replay buffer offer over a baseline storage replay buffer?” We will look at Experiments 1 and 2 for this.

2. The second study answers the question, ”How much benefit does a prioritized cache replay buffer have offer over a non-tiered prioritized replay buffer?” We will look at Experiments 2 and 3 for this.

3. The third study answers the question, ”How much is annealing useful for a large environment?” We will look at comparing Experiment 3 and 4 for this.

4. The fourth study answers the question, ”How do these implementations impact latency and memory?” We will look at all Experiments for this.

Exp	File	Timesteps	Cache Size	Quantization
1	SAC torch rb	500K	NA	NA
2	SAC torch prb	500K	NA	NA
3	SAC pcrb	500K	Linear $[32, 16, 8]$	[Obs/Next Obs]
4	SAC pcrb anneal	500K	Linear $[32, 16, 8]$	[Obs/Next Obs]

Table 3: List of Experiments

Note that the last study question deals with time and space analysis. Our model is designed to have the same space complexity (ie memory) as that of torch's implementation. We showcase the memory usage of all of our experiments to verify this is the case.

The time complexity of our model is measured by the latency of operations. We look at the following operations: training the neural networks, targeting loss functions, sampling an experience from the replay buffer, adding an experience into the replay buffer, taking an action, and navigating the environment. These likewise are shown for all of our experiments.

All of our experiments run on FASRC using the Hopper v5 environment. The FASRC cannon compute cluster is a large-scale, high performance computing cluster HPC owned and supported by Harvard's Faculty of Arts and Sciences. It spans three data centers maintaining 60 PB of storage and offers 660+ A100 GPU units for experimentation. Hopper v5 is an environment built by gymnasium and is part of a larger collection of Mujoco environments. The composition of its data, in TensorDict format, is shown in Table 4. As one can see, the observations take up a great deal of space per TensorDict experience, and so quantizing these values is our primary objective.

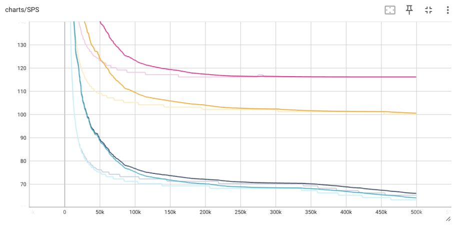
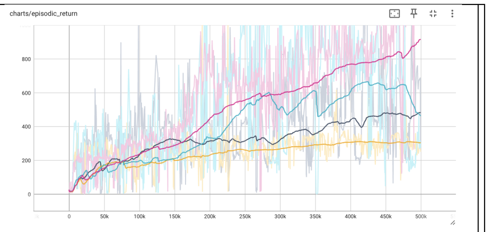
Parameter	Type	Size	Bytes/Dim	Total Bytes/Bits
obs	numpy array	11D float64	8	88/704
real_next_obs	numpy array	11D float64	8	88/704
actions	numpy array	3D float32	4	12/96
rewards	numpy array	1D float64	8	8/64
terminations	numpy array	1D bool	1	1/8
TD_error	torch tensor	1D float64	8	8/64

Table 4: Hopper v5 TensorDict Experiences

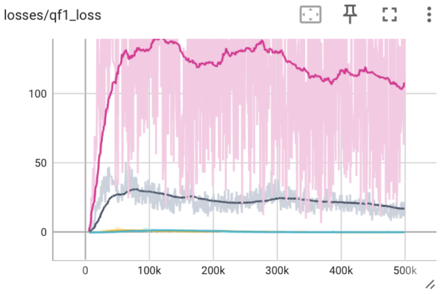
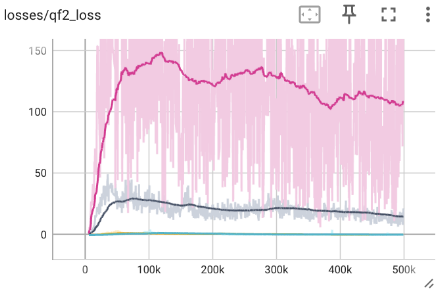
For more details on FASRC, after logging into the network's canon cluster, we request for an interactive gpu_test job using the salloc command. gpu_test allows for one to use up to 896 cores and 112 GPUs across 14 nodes. Each node offers on average 503 GB. We request for our job to use up to 40 GB of memory (for training and execution). We are given by default 100 GB of home directory memory (for setup-environment variables and dependencies). Finally also dedicate 4 CPUs per task required for training.

6 Experiment: Results

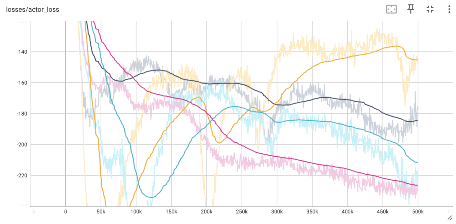
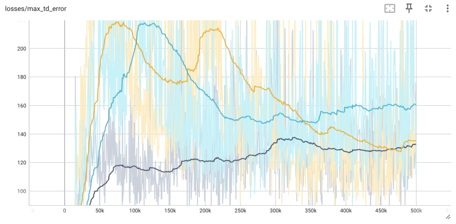
The results of all of our experiments are shown below. We group their results for lease of legibility. The following showcases their steps per second (SPS) and episodic reward:

Exp	SPS	Episodic Reward
1 (pink) 2 (yellow) 3 (blue) 4 (black)		

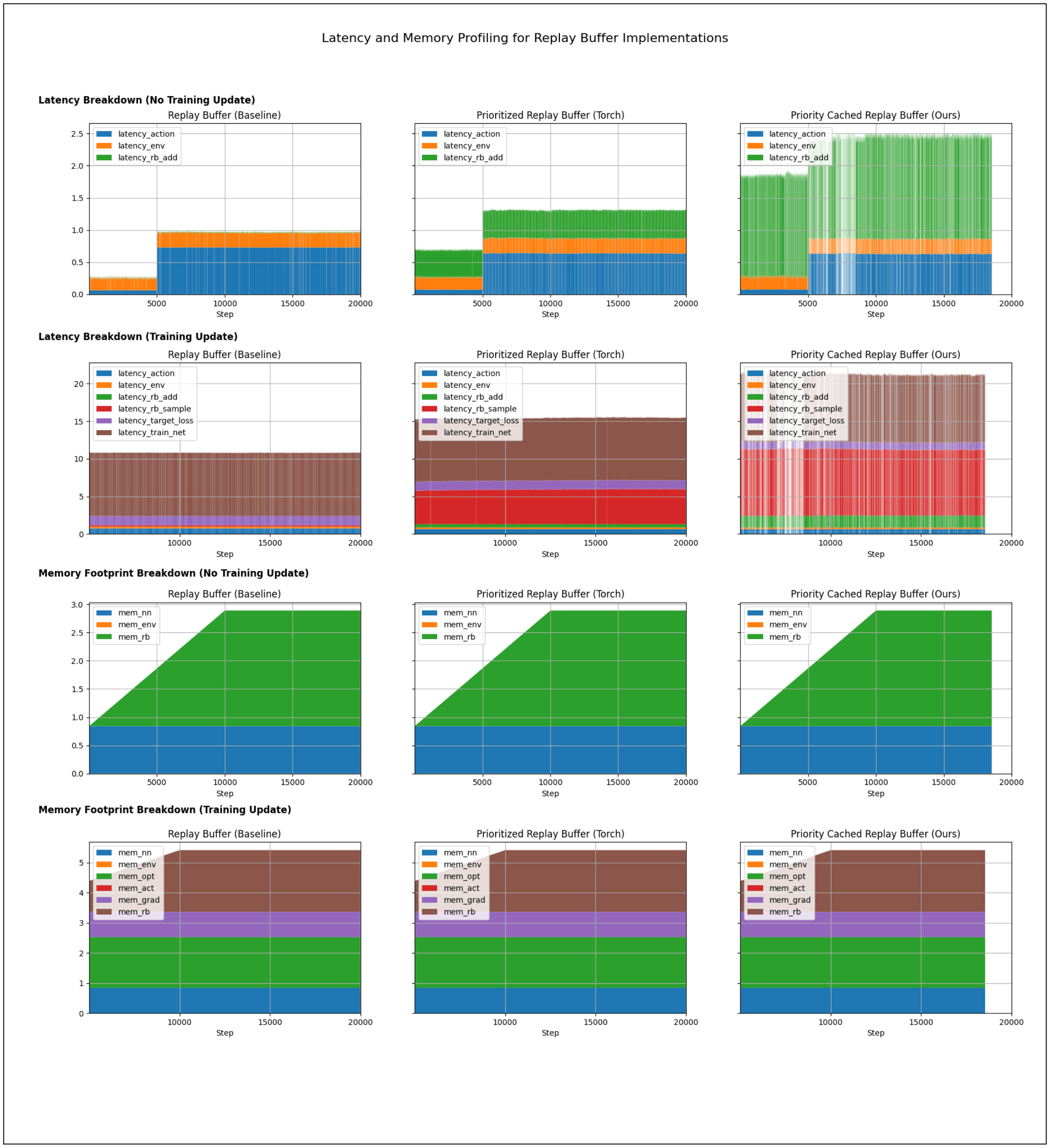
The next results showcase the algorithms' loss functions. Recall we do double Q-learning for stability:

Exp	Q1 Loss	Q2 Loss
1 (pink) 2 (yellow) 3 (blue) 4 (black)		

Finally, we showcase the actor loss function. For Experiments 2-4 that do prioritization based on the TD errors, we also showcase a function of its max TD error loss over time. The idea here is that higher TD error losses mean the agent is trying out more difficult-to-learn transitions, which can make training a bit unstable.

Exp	Actor Loss	Critic Loss
1 (pink) 2 (yellow) 3 (blue) 4 (black)		

In addition to these experiments above, we analyze their profiling. This analyzes latency, memory, and individualized reward-steps of different operations within the soft-actor critic model. The results are shown below. Each experiment is run for 20K steps on 3 seeds, averaged together:


Latency and Memory Profiling for Replay Buffer Implementations

Latency Breakdown (No Training Update)

Replay Buffer (Baseline) · Prioritized Replay Buffer (Torch) · Priority Cached Replay Buffer (Ours)

Latency Breakdown (Training Update)

Replay Buffer (Baseline) · Prioritized Replay Buffer (Torch) · Priority Cached Replay Buffer (Ours)

Memory Footprint Breakdown (No Training Update)

Replay Buffer (Baseline) · Prioritized Replay Buffer (Torch) · Priority Cached Replay Buffer (Ours)

Memory Footprint Breakdown (Training Update)

Replay Buffer (Baseline) · Prioritized Replay Buffer (Torch) · Priority Cached Replay Buffer (Ours)

In the figure above, we profile the latency and memory footprint of each replay buffer alternative. At each SAC step, we profile the latency of the every sub-procedures that make up that step and calculate the memory overhead of that step. We further split up the steps by training steps (which occur every other step after step 5000) and non-training steps because training steps include the additional latency and memory of training the actor and critic neural network.

In the first row, we observe the latency of sub-processes for non-training steps. The figure in the first column is for a baseline replay buffer. The blue is the latency of calculating which action to take. Note that it is almost negligible for steps ¡ 5000 because the action is essentially calculated randomly, and then after step 5000 when training occurs, the action is a forward pass through the actor neural network and thus dominates the step latency. The almost invisible green shows that it is essentially free to add to the replay buffer, and the orange shows the constant time to interact with the environment. The figure in the second column is the priority replay buffer. Note that blue and orange is essentially equal (which makes sense because the priority replay buffer does not change the forward pass through the actor or interacting with the environment). However, because adding to the priority replay buffer uses a simple list for experiences and segment tree for priorities, insertion is now O(logN) and thus significantly contributes to latency. Lastly, the figure in the third column is our priority cache replay buffer. Blue and orange remains the same, but we now observe a huge bottleneck in insertion. Although the segment tree insertion is still O(logN), the PCRB has more entries because our quantization scheme allows more experiences in a fixed memory budget. While this is beneficial to the algorithm because it has access to more experiences, the insertion also takes longer because N is bigger. Also, we have to re-quantize everything that changes tiers. These factors contribute to the large PCRB insertion time. However, our environment is very small, and one can reasonably imagine that in a real-world application, the orange bar representing interacting with the environment dominates latency. For example, when doing robotics, the latency of actually doing the action and observing the environment is orders of magnitude slower than computationally adding to a buffer.

In the second row, we observe the latency of sub-processes for training steps where we also back-propagate through the actor and critic neural networks. In the first column, the baseline replay buffer spends most latency training the neural net and negligible time sampling from the buffer. In the second column, the PRB spends more time sampling because of the O(logN) latency of sampling from a segment tree. However, everything else stays the same. In the third column, our PCRB again shows a significant bottleneck in sampling because of the bigger N and quantization overhead. However, with real-world problems, we anticipate that the environment latency will significantly grow. Also, with bigger neural networks with more parameters, training will take more time too. This latency profiling explains the drop in steps-per-second of our PCRB because each steps takes longer with the additional replay buffer sampling overhead.

In the third row, we observe the memory overhead of sub-processes for non-training steps. We cannot measure this empirically because of the lack of granularity we get from physical memory profilers, but we can calculate this based on the size of data elements multiplied by their magnitude. For example, the neural network memory footprint is the size of a parameter multiplied by the number of parameters and the replay buffer memory footprint is the size of an experience (with the action, reward, state, etc.) multiplied by the number of entries. Every graph is identical because the neural network stays the same, and the replay buffers stay a fixed size (although our PCRB contains more entries in the same memory budget). The reason why the green grows linearly until 10k is because we set a replay buffer max size of 10k entries (and the fixed-memory equivalent of 10k entries for PCRB). Up until step 10k, the replay buffer is still filling up and after that, we have to evict for each insertion so it then has a flat memory footprint.

In the fourth row, we observe the memory overhead of sub-processes for training steps. Again, note that the graphs are all identical, and now we observe a significant memory overhead for the gradient and optimizer.

While our profiling shows that our PCRB provides no benefit to latency or memory overhead of off-policy RL, this is not our intended contribution. In real-world applications where interacting with the environment is the dominating overhead, it is reasonable to trade off a small amount of latency for an increase in return. While simply increasing the size of the replay buffer allows for more stored experiences, many edge applications have fixed memory budgets making this impossible. Our PCRB takes a computer system-inspired approach to increase the number of stored experiences while maintaining the concept of prioritized experiences. This profiling work gives more insights into the latency and memory of different replay buffer implementations and to help make more informed decisions.

7 Experiment: Analysis

Here is what we can conclude for our studies:

1. **How much benefit does a non-tiered prioritized replay buffer offer over a baseline storage replay buffer?**
   It appears from experiments 1 and 2 that a prioritized replay buffer can falter in performance compared to a traditional baseline replay buffer. This is more-so explained by the fact that our environment, Hopper v5, is not as complicated as other gymnasium or Mujoco-styled environments. We list these environments in the future work section for further investigation. What we conclude from this study is that one should not over-complicate environments that can be solved with simpler methods.

   What is also notable is that the Q1 and Q2 loss functions for the baseline are much larger compared to the other experiments done. This is explained by the fact that the baseline is fed a broader selection of the environment compared to the prioritized replay buffer, which selectively chooses for more surprising experiences. Therefore, in the baseline, many of the experiences may be uninformative and take longer to learn.

2. **How much benefit does a prioritized cache replay buffer have offer over a non-tiered prioritized replay buffer?**
   Compared to the prioritized replay buffer, the cached and quantized priority replay buffer offers better episodic rewards. This is because our quantization allows more experiences to fit within same amount of memory space. The neural networks of the actors and critics can therefore learn with more information and subsequently improve over the torch rl's prioritized replay buffer.

3. **How much does annealing help?**
   Annealing (compared to non-annealing) appears to help during the initial stages of training. However, over time, it becomes less responsive and is reduced in performance compared to the non-annealed prioritized cache replay buffer. This makes sense, as we are adjusting in our

annealed setting the beta parameter to 1, making the model overall more conservative. By starting the annealing at a later duration into the run or starting beta at a smaller value, one can achieve better experiences. Our setup was to immediately start annealing towards 1, and our value for beta was at 0.4. This means that beta quickly achieves a value over 0.5, which then it becomes more conservative.

4. **How do these implementations impact latency and memory?**
   As mentioned in section 6, there is a significant latency overhead in insertion and sampling from our PCRB. This is directly tied to our contribution of using multi-level quantization schemes to increase the number of experiences that can fit in a fixed memory budget replay buffer. Continually re-quantizing data as it moves up and down the buffer tiers has overhead. Also, any prioritized replay buffer uses a segment tree to store priorities, which require a O(logN) insertion and sampling time. The PCRB needs a segment tree and also increases the N term because we can fit more experiences in the buffer by reducing precision. We trade off the latency from quantization and segment tree traversal overhead for the ability to store more experiences in a fixed memory budget.

8 Future Work

For future work, readers can attempt to run the results on different environments other than Hopper v5. Hopper v5 is one of Mujoco’s more simplistic model that attempts to balance an object on its one leg. Therefore, our results look the way they do because they do are not featured in an environment that is as ’exploitive’ as a more complicated setup. We navigated towards Hopper v5 initially due to its ease of setup and the desire to see how far to an extreme one can apply the ’complicated measurings’ offered in Soft-Actor Critic methods. We believe that running experiments on more realistic and non-trivial environments will also increase the importance of sampling *surprising* experiences and thus make the usage of a prioritized experience replay buffer valuable. In our current Hopper v5 experiments, we observe that the baseline RB outperforms the PRB in episodic return, but we want to recreate experiments where the PRB outperforms the baseline RB. The PRB paper is a seminal contribution in machine learning with over 6,000 citations and implementation in PyTorch, so there are definitely environments where there is value to prioritizing experience sampling. In the specific class of environments that reward prioritized experience sampling and have a small fixed memory budget, we are interested to see if our PCRB still outperforms the PRB in episodic return and thus yields real implications.

Possible alternative environments to evaluate on include the Half Cheetah environment (walking and running on four legs), Walker 2D (walking on 2 legs with a balancing torso), and either the CartPole or the Double/Single Inverted Pendulum environment (balancing a pole on a moving cart).

We also encourage larger time-step trials and experimentation with the quantization and cache-size hyperparameters. For example, perhaps a more extreme cache-size dropoff for later tiers can provide more room for environments that are generally more complicated and would therefore benefit from more samples.

The final suggestion we offer for future work is for latency enhancement. This particular paper focuses on the trade-off between latency and reward. The increased latency is due to the casting and added operations required to add and sample experiences from the replay buffer. By programming in a language that can support more lower-level manipulations of memory allocation and casting, one can lower the latency overhead.

Bibliography

[1] Gavin An and Sai Zhang. *Pruning replay buffer for efficient training of deep reinforcement learning.* 2023. URL: https://doi.org/10.59720/23-068.

[2] Johan Bjorck et al. *Low-Precision Reinforcement Learning: Running Soft Actor-Critic in Half Precision.* 2021. URL: https://doi.org/10.48550/arXiv.2102.13565.

[3] Suyog Gupta et al. "Deep Learning with Limited Numerical Precision". In: *Proceedings of the 32nd International Conference on Machine Learning (ICML).* JMLR.org, 2015, pp. 1737–1746.

[4] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* 6th ed. San Francisco, CA: Morgan Kaufmann, 2017.

[5] Matteo Hessel et al. *Rainbow: Combining Improvements in Deep Reinforcement Learning.* 2017. URL: https://doi.org/10.48550/arXiv.1710.02298.

[6] Volodymyr Mnih et al. *Human-level control through deep reinforcement learning.* 2015. URL: https://doi.org/10.1038/nature14236.

[7] Tom Schaul et al. "Prioritized experience replay". In: *arXiv preprint arXiv:1511.05952* (2015).

[8] Richard S Sutton. "Learning to predict by the methods of temporal differences". In: *Machine learning* 3 (1988), pp. 9–44.

[9] Ryosuke Unno and Yoshimasa Tsuruoka. *Memory-Efficient Reinforcement Learning with Priority based on Surprise and On-policyness.* 2023. URL: https://openreview.net/forum?id=xkSlKCYyV_.

[10] Samuel C. Woo et al. "The SPLASH-2 Programs: Characterization and Methodological Considerations". In: *Proceedings of the 22nd Annual International Symposium on Computer Architecture.* IEEE Computer Society. 1995, pp. 24–36.

[11] Guangyao Zheng et al. *Selective experience replay compression using coresets for lifelong deep reinforcement learning in medical imaging.* 2013. URL: https://doi.org/10.48550/arXiv.2302.11510.