Sample-Efficient and Safe Deep Reinforcement Learning via Reset Deep Ensemble Agents

Kaya Unalmis, kunalmis@princeton.edu.
Lily Gittoes, lgittoes@princeton.edu.

May 18, 2024

Contents

1 Introduction and Motivation

Many problems in machine learning reduce to, or are usually cast into, an optimization problem to minimize a loss function. Often, it is insufficient to find local minima, and we would like our optimizer to converge to globally interesting minima. This request is demanding, for capturing a global minima in a general optimization space is known to be NP-hard; if we could solve it in polynomial time, any NP-hard problem could be solved efficiently. Therefore, state-of-the-art optimization methods still rely on various versions of gradient descent that are fundamentally local solution finding algorithms. In practice, such gradient descent methods can perform well, so long as they do not get stuck in an undesirable local minima [JNJ17].

This is particularly relevant in cases where the loss function's gradient is determined in an online manner, or in the more general reinforcement learning (RL) problem, in a manner which is dependent on the trajectories of state-action pairs. In practice, the optimizer moves in the descent direction based on the gradient information of an initial set of samples. If these samples are not representative of the optimization landscape, or even "misleading" in the context of finding a globally interesting minima, the optimizer's descent will be biased toward an undesirable direction. Depending on the problem, it can be difficult to recover from this bias. Naturally, one attempt to resolve this issue is to proceed with the optimization regardless, collect information about the loss function's gradient, and then restart the optimization while retaining this information. The hope is that a fresh model with better prior knowledge of the optimization landscape will be more resistant to moving in a descent direction that is biased by early gradient information.

1.1 Primacy Bias

The phenomenon that the model suffers from in the previous section is called *primacy bias*. This concept originates in cognitive science, where results have shown that early impressions have disproportionate impacts on human memories [MW72]. It was extended to deep RL settings by [Nik+22] to describe the overfitting of function approximators to early experiences, which can lead to sub-optimal performance. In such RL settings, function approximators are formulated with deep neural network (DNN) architectures and are used to describe large state-action spaces. They are trained according to samples from a replay buffer that tracks previously observed state transitions. To combat the primacy bias that results in overfitting on early experiences, [Nik+22] proposes a series of periodic agent resets in which their network parameters are reinitialized. As the impacts of primacy bias are compounded when the replay ratio, defined as the number of gradient updates made per environment interaction, is high, the algorithm proposed by [Nik+22] can be applied to improve sample efficiency.

However, immediately following these parameter resets, their model experiences large performance collapses. For certain settings, such as those where the agent must interact with a physical environment (i.e., autonomous cars or robots), these drops in performance raise concerns. With the aim of developing a method that does not suffer from primacy bias nor large safety issues, [Kim+23] proposes the **R**eset-based algorithm by leveraging **D**eep **E**nsemble learning (RDE). By training an ensemble of agents instead of just one, [Kim+23] resets the agents' network parameters one at a time. This allows [Kim+23] to avoid performance collapses, because their agents do not have to immediately interact with the environment after having their parameters reset. To demonstrate the improvements of RDE over the vanilla reset method by [Nik+22], [Kim+23] implements RDE with two off-policy RL methods: Deep Q-Networks (DQN) and Soft Actor-Critic (SAC) [Mni+15; Haa+19]. They carry out tests over a series of environments, including Atari-100k, MiniGrid, and the DeepMind Control (DMC) Suite [Bel+13; CWP18; Tas+18].

1.2 Road Map

Our project aims to reproduce the results of [Kim+23]. Starting from the algorithm pseudocode they have provided alongside the parameters they tuned, we write code from scratch to employ RDE alongside DQN and SAC. Modifications to the vanilla algorithm that were found necessary for convergence are made. We evaluate our implementation of RDE with SAC using the `Cheetah-Run` setting in the DMC suite and compare our results with those produced by [Kim+23]. After, because of computation and time constraints, rather than performing training in the Atari-100k and MiniGrid environments, which are much harder to learn, we instead evaluate our implementations of RDE with SAC and DQN in the `Mountain Car Continuous` and `Cart Pole` Gym environments [Bro+16].

We lay out the rest of this paper as follows. In Section 2, we describe the different methods employed by others to overcome the challenge of overfitting DNN function approximators and explain how their work relates to ours. Then, in Section 3, we detail the main components of our implementation of RDE with DQN and SAC. Our results are displayed in Section 4, where we also discuss the extent to which our results replicate that of [Kim+23]. Afterwards, we describe the limitations that we ran into, potential paths for future work, and the key conclusions of our paper in Section 5.

2 Background and Related Work

While the concept of primacy bias is new, methods for dealing with overfitting in deep RL settings are not. In this section, we describe methods that have previously been employed by others in relation to the RDE algorithm. For our project, as is used by [Kim+23], we also employ the standard Markov decision process (MDP) formulation. That is, we consider an agent with action space $\mathcal{A}$ that exists in an environment with a state space $\mathcal{S}$. In a state $s \in \mathcal{S}$, upon choosing an action $a \in \mathcal{A}$, the agent receives a reward $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ and transitions to a new state s' according to the probability distribution p. For the discount rate $\gamma \in [0, 1)$, the agent aims to find the optimal policy $\pi : \mathcal{S} \to \Delta\mathcal{A}$ that maximizes the cumulative discounted rewards $\mathbb{E}[\sum_t \gamma^t r(s_t, a_t)]$.

2.1 Overfitting DNN Function Approximators

DNNs are often used as function approximators to describe the Q-values associated with an MDP's state-action space. These DNNs are susceptible to overfitting, and a series of different approaches have been suggested for dealing with this, including ensemble learning and prioritized experience replay.

For a policy π and state-action pair (s, a), the Q-value is defined as $Q^\pi(s, a) = \mathbb{E}^\pi[\sum_t \gamma^t r(s_t, a_t)|s_0 = s]$. We focus on the two off-policy RL algorithms employed by [Kim+23]: DQN and SAC. They both use Temporal Difference (TD) learning, which aims to minimize the difference between $Q^\pi(s, a)$ and $\mathbb{E}_{p(s'|s,a),\pi(a',s')}[r(s, a) + \gamma Q^\pi(s', a')]$. In TD learning settings, gradient descent is implemented according to bootstrapping – namely, regressing towards the model's own output value function. When this is combined with function approximation in off-policy settings, results are often unstable. In such conditions, it has been found that as more gradient steps are taken, the function approximators lose their expressivitiy and struggle to fit new functions [Kum+21; LRD22].

The goal of ensemble learning is to integrate a series of agents to act collectively, with the goal of improving the robustness of baseline algorithms. For instance, to improve upon the originally proposed DQN algorithm by [Mni+15], [HGS16] proposes the Double-DQN framework to simultaneously train two sets of Q-networks that update each other. They show that this can alleviate some of the instability problems encountered in the baseline DQN algorithm. Meanwhile, [ABS17] formulates a different ensemble-based algorithm called Averaged-DQN, in which Q-value estimates are computed as the average across a series of Q-networks. A third method, which is proposed by [Lee+21], suggests training an ensemble of Q-networks that can be used to weight sample transitions according to their level of uncertainty. They implement a weighted Bellman backup, observing that this mitigates some of the error propagation seen in simple Q-learning models.

Meanwhile, others have worked to improve how samples from the replay buffer are selected, with the goal of improving the performance of TD learning methods. Notably, [Sch+16] proposes a sampling method called Prioritized Experience Replay, which assigns weights to samples (s, a, r, s') stored in the replay buffer using the TD difference function $\delta = r + \max_{a'} Q(s', a') - Q(s, a)$. By giving higher weights to samples associated with a larger TD difference, they are able to see significant performance improvements. On the other hand, [Wan+20] suggests a non-uniform sampling method whereby more recent experiences are given higher weights. Although they do not explicitly call it primacy bias, their approach is motivated by preventing overfitting to early experiences.

Inspired by the past successes of ensemble learning in improving the stability of TD learning, [Kim+23] uses this as a key element in their proposed RDE algorithm. In terms of the replay buffer that they maintain, [Kim+23] does not prioritize using certain samples over others when making gradient steps. This is because after performing each agent reset, they want to maintain equal access to all previously seen environment interactions.

2.2 Agent Resets and Sample Efficiency

A key feature of using agent resets to alleviate the impacts of primacy bias is the enhanced sample efficiency it offers. Sample efficiency is improved with higher replay ratios, which allow for a greater number of parameter updates to be made per environment interaction [Fed+20; HHA19]. However, [Nik+22] finds that as the replay ratio increases so does primacy bias. In other words, if too many parameter updates are made early on in the training process, the model could become stuck at a local minima and inflexible to later experiences. To combat the impacts of primacy bias, [Nik+22] introduces the notion of periodically partially or fully resetting an agent's network parameters. They conjecture that letting an agent start from scratch and make parameter updates according to a preserved replay buffer from prior training steps can prevent overfitting to early environment interactions. By implementing their network reset approach with respect to a series of baseline algorithms, they demonstrate improvements over benchmarks.

Inspired by the work of [Nik+22] on primacy bias, [DOr+23] finds that their agent reset algorithm can be adjusted to allow for improved sample efficiency. Namely, rather than fixing the frequency of agent resets with regards to the number of time steps of environment interactions like [Nik+22] does, [DOr+23] fixes the reset frequency with regards to the number of parameter update steps. By increasing the agent reset frequency alongside the replay ratio, [DOr+23] achieves the high levels of sample efficiency offered by high replay ratios without having to bear the impacts of primacy bias. In their RDE algorithm, [Kim+23] employs an ensemble of agents that are reset sequentially according to the method described by [Nik+22]. Using the results of [DOr+23], their resets occur at a frequency that decreases linearly with the replay ratio. With this, they aim to develop an algorithm that is sample efficient, safe, and avoids primacy bias.

3 Method

In this section, we describe the main components of RDE and provide overviews of the algorithms we implement alongside: SAC and DQN.

3.1 The RDE Algorithm

The RDE algorithm proposed by [Kim+23] that we aim to replicate is divided into the following steps. First, we initialize N ensemble agents, where each agent's network parameters are denoted by θ_k for $k \in \{1, \dots, N\}$. The N ensemble agents all have the same network architecture.

During model training, the agents $k \in \{1, ..., N\}$ are reset in a sequential order, starting with $k = 1$. After resetting agent N, we circle back and reset agent 1 again. Resetting an agent consists of resetting all of their network parameters – this includes both the actor and critic networks for SAC and both the Q-network and target Q-network for DQN. Given the input parameter T_{reset}, there are T_{reset}/N training steps between agent resets. Thus, upon completing T_{reset} training steps, all N of the agents will have been reset once. We then continue cycling through the agent resets in this manner for the entirety of the training period.

In terms of interacting with the environment, the ensemble of agents work together as a single unit. According to the agents' policies $\{\pi_{\theta_1}, ..., \pi_{\theta_N}\}$, for a given state s, the agents suggest actions $\{a_1, ..., a_N\}$. The action that is actually taken is selected using the probability distribution $p_{select} = \{p_1, ..., p_N\}$ over the N possibilities, which is computed as follows. Using the Q-value function $\hat{Q}$, which is that of the agent that was least recently reset, we calculate Q-values for the suggested actions $\{a_1, ..., a_N\}$ and the current state s. These are denoted by $\{\hat{Q}(s, a_1), ..., \hat{Q}(s, a_N)\}$. We then apply the softmax function to these Q-values, using the temperature

$$\alpha = \beta / \max(\hat{Q}(s, a_1), \dots, \hat{Q}(s, a_N)).$$

In this, β is a parameter that is tuned and allows us to decide on the extent to which we assign higher weights to actions with high Q-values and lower weights to those with low Q-values. Thus, the probability distribution p_{select} is given by

$$p_{select} = \{p_1, ..., p_N\} = \text{softmax}\{\hat{Q}(s, a_1)/\alpha, ..., \hat{Q}(s, a_N)/\alpha\}. \tag{1}$$

and the probability of selecting action a_k is p_k. We note that before the first reset (which occurs at time step T_{reset}/N), all of the agent's network parameters are identical. Hence, the probability distribution is uniform: i.e., $p_{select} = \{1/N, ..., 1/N\}$.

By selecting actions according to the Q-value function of the agent that was least recently reset, we are able to avoid the performance collapses observed in the vanilla reset method proposed by [Nik+22]. As is explained by [Kim+23], this is because choosing an action according to p_{select} from an ensemble of agents allows us to maintain a proper balance between exploration and exploitation. In the vanilla reset method, since there is just one agent, the model is only able to perform exploration immediately after it is reset, leading to drops in performance. Moreover, in the ensemble reset method, choosing actions from a uniform distribution over the N agents is also insufficient to avoid performance collapses. This is because actions selected by agents that were recently reset will be from untrained models, hence they should be taken with a lower probability. By using the probability distribution p_{reset}, these actions are given a lower weight, as actions are selected according to the Q-value function of the least recently reset agent. Thus, agent resets can occur without significant performance drops.

3.2 Implementing RDE with SAC

Following suit with [Kim+23], we start by implementing RDE with SAC. For SAC, we implement the Twin Delayed Deep Deterministic (TD3) algorithm [Haa+19; FHM18]. This algorithm extends upon standard Actor-Critic methods to add encouragement for policies that achieve a higher entropy, thereby adding incentives for exploration. For each agent $i \in \{1, ..., N\}$, we initialize two sets of Q-networks alongside two sets of corresponding target Q-networks. In addition, we also initialize a policy network and corresponding target policy network for each agent. By interacting with the environment using the N ensemble agents' policy networks, experiences are stored in a circular replay buffer $\mathcal{B}$. In a circular replay buffer, after the number of stored sample transitions reaches the buffer's capacity, the oldest experiences start to be removed as new ones are added. We note, however, that the buffer's capacity is set to be very large so that agents that have been reset can still access old samples. Using uniform samples from $\mathcal{B}$, the agents update their policy networks and Q-networks.

Pseudocode for combining RDE with SAC, including its loss functions, is described in Appendix A. We note that this pseudocode does not account for two characteristics of the SAC algorithm described by [Haa+19]: target entropy annealing and a modified entropy equation. We find that these two techniques are not needed for SAC to be successful in the `Mountain Car Continuous` environment, but they are needed in the `Cheetah-Run` setting, likely because it is more complex. Hence, we only implement them for `Cheetah-Run` and this is further described in Appendix A. Our code for implementing RDE with SAC in these two environments is available in Appendix E.

3.3 Implementing RDE with DQN

Next, we implement RDE with DQN according to the framework proposed by [Mni+15]. In this algorithm, each agent interacts with the environment according to an ϵ-greedy policy and keeps track of their past transitions (s_t, a_t, r_t, s_{t+1}) in a circular replay buffer $\mathcal{B}$. The agent aims to learn the parameters θ of the DNN that describes their Q-value function $Q(s, a; \theta)$. Pseudocode describing the integration of RDE with DQN is provided in Appendix A. We run simulations of our code in the `Cart Pole` Gym environment. While [Kim+23] runs tests for RDE with DQN in the Atari-100k and MiniGrid environments, because these environments are so complex, our DQN algorithm takes a long time to train. Moreover, [Kim+23] does not provide a complete set of their hyperparameters (in particular, they are missing key details about their DNN architecture used in the MiniGrid environments). Given the long training time for DQN in complex settings with sparse rewards, we find the `Cart Pole` environment to be more suitable for testing the effectiveness of RDE in this project.

3.4 Experiment Setup

In this section, we lay out our setup for the experiments we perform in the `Cheetah-Run`, `Mountain Car Continuous`, and `Cart Pole` settings. For each environment, [Kim+23] runs simulations for a series of nine scenarios. They consider the baseline algorithm (TD3 or DQN) with no ensembles or resets, the algorithm with the vanilla reset method proposed by [Nik+22] (which is identical to the RDE algorithm when $N = 1$), and the algorithm with RDE using $N = 2$ agents. For each, they test replay ratios of 1, 2, and 4. Moreover, [Kim+23] scales the reset interval T_{reset} so that it decreases linearly with the replay ratio. We start by reproducing the results of [Kim+23] in the `Cheetah-Run` setting in the DMC suite [Tas+18]. Then, we repeat these same nine scenarios for the `Mountain Car Continuous` Gym environment [Bro+16].

For the RDE with SAC algorithm in the `Cheetah-Run` setting, we use almost identical hyperparameters to [Kim+23]. However, in order to decrease the computational complexity of our training loop, we make changes such as using fewer hidden units in our DNN layers and sampling smaller batches from the replay buffer, and these are detailed in Appendix B. These changes allow us to closely replicate the results of [Kim+23] with less time and computational power. Moreover, as the runtime scales linearly with the replay ratio and the number of agents, for the scenario in which we use a replay ratio of 4, we use half the number of training steps as was used when the replay ratio is 1 or 2. Our limitations regarding computational power are discussed in more detail in Section 5.

Then, when implementing RDE with the SAC algorithm in the `Mountatin Car Continuous` environment, we use similar parameters to what we used in the `Cheetah-Run` setting, but because the model trains faster in this environment, we examine a fewer number of total timesteps and use an analogously smaller number of timesteps between resets T_{reset}. However, to avoid the extra large computation times in the scenario where we implement RDE with a replay ratio of 4, we use half the number of training steps compared to when the replay ratio is 1 or 2.

As for when we implement RDE with the DQN algorithm in the `Cart Pole` environment, we use parameters that are similar to those used by [Kim+23] in their MiniGrid experiments. Here, we also halve the number of training steps when applying a replay ratio of 4 because of computational constraints. Moreover, we also use the same parameter T_{reset} for both replay ratios of 2 and 4 rather than decreasing it linearly with the replay ratio as we find that doing so makes the resets so frequent that the agents do not have enough time to recover after. For all three of the environments we consider, more details on our DNN architectures and hyperparameters are provided in Appendix B.

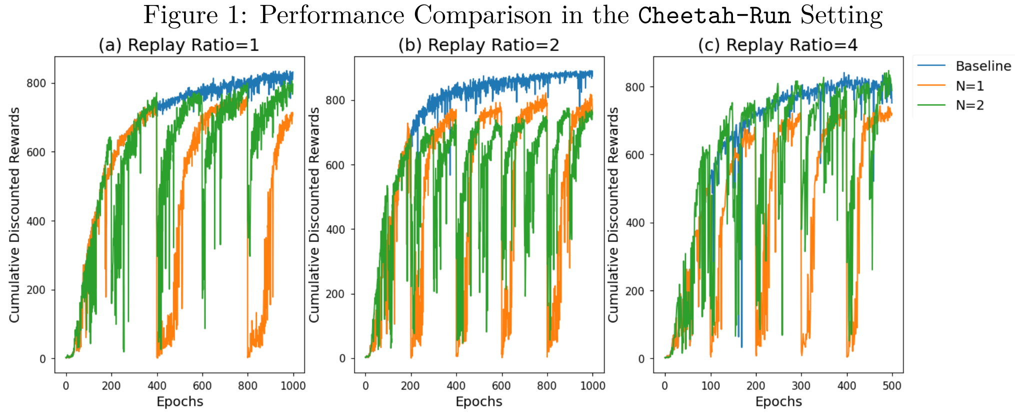
4 Results and Discussion

Using the methods and experimental setups described in Section 3, we run simulations of our code implementations and compare our results to those presented by [Kim+23].

4.1 Experimental Results

In this section, we display the results of our experiments when implementing RDE with SAC and DQN. To begin, we plot a comparison of the rewards achieved in the `Cheetah-Run` setting below in Figure 1. This series of plots aims to replicate plots (d), (e), and (f) in Figure 7 by [Kim+23].

Figure 1: Performance Comparison in the `Cheetah-Run` Setting

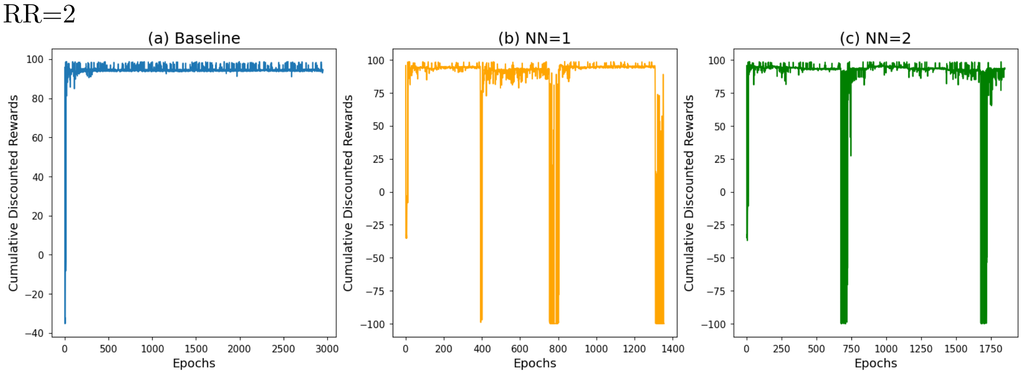


For `Cheetah-Run`, this figure displays the performance of the SAC in the baseline case (in blue), with naive resets with one agent (in orange), and with ensemble resets using RDE with two agents (in green). Results are shown across replay ratios of 1, 2, and 4.

From Figure 1, we can see that when the replay ratio is 1 or 2, the baseline SAC algorithm outperforms both the vanilla reset and RDE methods, though not by a very significant margin. Meanwhile, when the replay ratio is 4, by the end of the training loop, the RDE method does about as well as the baseline SAC algorithm. These results are fairly consistent with [Kim+23], who does not observe significant performance improvements from implementing RDE in the `Cheetah-Run` setting. However, we note that while our implementation of RDE with SAC does not experience performance collapses as significant as those in the vanilla reset method, they are larger than those observed by [Kim+23]. We identify two reasons for this. First, given the long training time that is required, we are only able to run our code once, while [Kim+23] averages their results over 5 seeds, so there is some inherent level of randomness that causes our results to differ. Moreover, given some inconsistencies identified in the discussion that [Kim+23] provides on how to tune the β parameter in computing p_{select}, we believe that this part of our implementation may not exactly match theirs. Further discussion on this is provided in Section 4.2.

Next, we display our results in the `Mountain Car Continuous` setting. We make comparisons across the baseline SAC algorithm without any agent resets, SAC with naive resets, and SAC with RDE for replay ratios of 1, 2, and 4. Our results for a replay ratio of 2 is shown below in Figure 2, while those for replay ratios of 1 and 4 can be found in Appendix C, as we do not find much variation in patterns across the three scenarios.

Figure 2: Performance Comparison in the `Mountain Car Continuous` Setting when RR=2

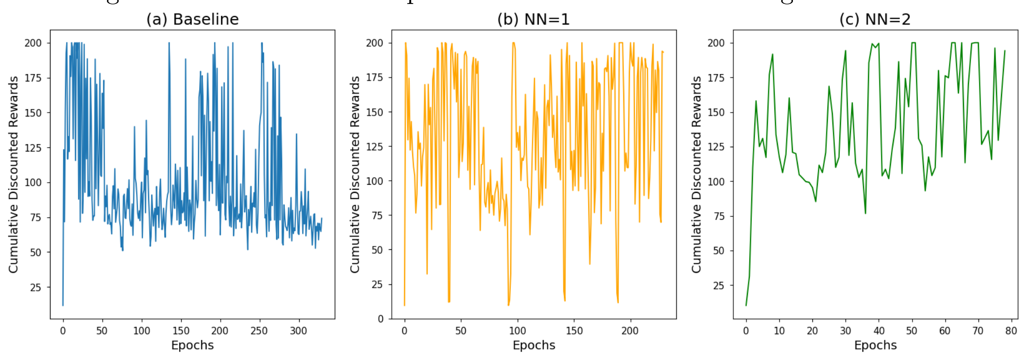


For `Mountain Car Continuous`, we compare the performance of SAC with a replay ratio of two across the baseline case, with naive resets with one agent, and with ensemble agents using RDE with two agents.

For the `Mountain Car Continuous` setting, we can see that the baseline SAC algorithm is successful on its own at learning the environment. This is likely because of the relatively small state and action space of the environment. Thus, as our algorithm appears to not suffer from primacy bias and overfitting, our results show that agent resets do not bring about significant performance improvements. Nonetheless, across all three tested replay ratios, we see that performance collapses are more frequent in the naive reset method compared to RDE, demonstrating that RDE is successful in improving safety. However, we notice that the extent of the drop in rewards experienced in each performance collapse with RDE when $N = 2$ is very similar to that in the naive reset method. Similar to what we observed in the `Cheetah-Run` environment, we believe that this is the result of our use of the β parameter being different than that described by [Kim+23], and we explain this in more detail in Section 4.2.

Finally, we display results for our simulations in the `Cart Pole` environment. Because there is not a significant difference in trends across the three replay ratios we test, we focus on results when the replay ratio is 2, and these are shown below in Figure 3 below. Meanwhile, results when the replay ratio is 1 or 4 are shown in Appendix C.

Figure 3: Performance Comparison in the `Cart Pole` Setting when RR=2



For DQN (using a replay ratio of two) in the baseline case, with naive resets with one agent, and with ensemble agents using RDE with two agents, we display our results.

We observe that the performance collapses observed when using naive resets are significantly greater than those seen when using RDE. This suggests that RDE indeed does provide a significant improvement over naive resets. However, because DQN is off-policy, uses bootstrapping, and employs function approximators to describe a large state-action space, it suffers from what is known as the deadly triad [Has+18]. When any of these three components are used together, it has been empirically discovered to be more likely to introduce instabilities to standard RL algorithms, impeding convergence of the optimizer to acceptable policies. It is also for this reason that it is particularly difficult for DQN to perform well on many of the more complex MiniGrid environments which have sparse rewards, especially when constrained to a short compute time. The instability of DQN as an algorithm makes it difficult to discern whether some of the performance collapses seen in Figure 3 are from noise or agent resets, especially when using RDE. Nonetheless, because of its added complexity, we note that the RDE algorithm with $N = 2$ agents completes a much smaller number of epochs over the same number of training steps as the baseline DQN or naive reset methods, and it is perhaps a direction for future work to evaluate this algorithm over more episode rollouts.

4.2 Evaluation of the Action Selection Coefficient

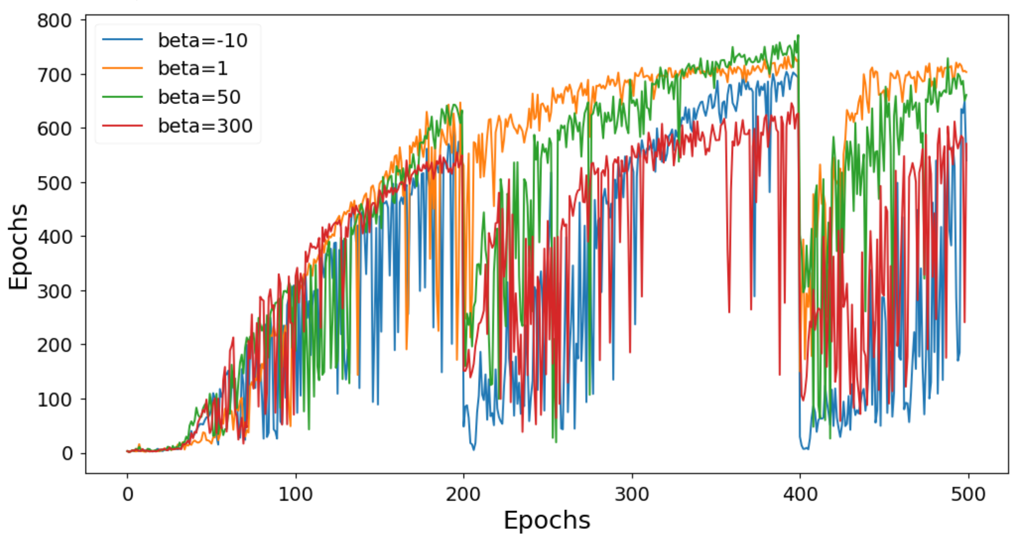
We improve upon the results presented by [Kim+23] by more thoroughly testing the impact of the action selection coefficient β, which is used to compute the softmax temperature α in Equation 1. Here, we notice that there are inconsistencies in their presentation of β. Namely, [Kim+23] claims that high β values yield better performance by adding higher weights to actions chosen by agents that are least recently reset. While we agree with their assessment that negative values of β exacerbate performance collapses, we find that for positive values of $\beta \geq 0$, *lower* β values place higher weights on less recently reset agents. To see this, consider the definition of the softmax function: for temperature $\alpha \in \mathbb{R}_+$, the softmax function $s_\alpha : \mathbb{R}^c \to [0,1]^c$ is defined by

$$s_\alpha : x \mapsto \begin{bmatrix} \mathrm{e}^{x(1)/\alpha} \\ \vdots \\ \mathrm{e}^{x(c)/\alpha} \end{bmatrix} \Big/ \sum_{k=1}^{c} \mathrm{e}^{x(k)/\alpha}.$$

As $\alpha \to 0$, the sum $\sum_{k=1}^{c} e^{x(k)/\alpha}$ gets exponentially larger, hence the softmax function places greater weight on larger input values. Since α is defined as being directly proportional to β, the same is true for β. This contradicts the statement by [Kim+23] that higher β values put more weight on actions chosen by less recently reset agents.

To see their claim, [Kim+23] tests values of $\beta \in \{-10, 0, 50, 300\}$. However, considering $\beta = 0$ makes the softmax function in Equation 1 undefined from dividing by 0. For a replay ratio of one, we run experiments of RDE with TD3 using $N = 2$ agents in the `Cheetah-Run` environment (using the same hyperparameters as those presented in Appendix B except that we use 2×10^5 training steps to save time) for varying β values in $\{-10, 1, 50, 300\}$. We plot our results below in Figure 4, where we observe that $\beta = 1$ yields the best performance by placing higher weights on least recently reset agents. Thus, we believe that the discussion [Kim+23] provides on β is inconsistent with their definition of p_{select}.

Figure 4: Performance in the `Cheetah-Run` Setting when RR=1 Across Different β Parameter Values



We display our results of RDE using $N = 2$ agents with SAC in the `Cheetah-Run` setting when the replay ratio is one. We consider β parameters in $\{-10, 1, 50, 300\}$.

5 Conclusion

In conclusion, by reproducing the results of [Kim+23], we find that RDE indeed does allow for less significant performance collapses, and it is an improvement over the naive reset method proposed by [Nik+22]. However, in some cases we have observed that the baseline method with no agent rests performs similarly to the ensemble reset method. In other environments the baseline stochasticity makes it hard to identify whether performance drops and boosts are due to primacy bias or some other influence such as insufficient duration of training, function approximation errors, or instabilities (e.g. such as those that arise from the deadly triad). While primacy bias has a precise cause that we can identify, its future effects are usually difficult to identify with confidence, just like primacy bias in cognitive science. In the sections below, we discuss the limitations of our work and suggest directions for the future.

5.1 Limitations

The primary limitations in our ability to reproduce the results of [Kim+23] are computational power and time. We list the compute times required for each of our simulations in Appendix D. To this end, we could not fully test the entire suite of environments considered in their paper. Especially for the more complex environments such as Atari-100k and MiniGrid, we did not have the capacity to run a large number of training loops. As [Kim+23] saw varying results across the different environments they considered (including those where RDE significantly outperforms baseline algorithms with no resets), our project is limited in its ability to verify all of their claims.

Moreover, to save time, we also took measures such as using fewer hidden units in our DNNs, taking smaller batches from the replay buffer, taking fewer total training steps, and only running simulations on one seed rather than averaging results over multiple. Meanwhile, as [Kim+23] is inconsistent in their discussion of the β parameter, this could also be causing differences in our experimental results. Thus, we can see that while our results generally align with those found by [Kim+23], differences in implementations do cause some variation in results.

5.2 Future Work

With more time and greater computational resources, there are many directions that can be explored. For instance, varying parameters such as the replay ratio and the reset interval T_{reset} or resetting only some of the layers of the DNNs being trained are all possible future steps. In addition, it would be interesting to run simulations with a larger ensemble. This will allow us to test whether choosing actions based on a larger ensemble of agents will further mitigate primacy bias or if there are diminishing returns after two models. Similarly, experimenting with different methods of selecting an agent could be worthwhile as well, as there could be better ways to formulate the probability distribution p_{reset} than that in Equation 1.

Furthermore, more simulations across a larger number of seeds can help us better understand the extent to which underlying stochasticity is impacting our observed results. To this end, further tests over a wider variety of environments that range in difficulty will help validate the results of [Kim+23]. In addition, considering the impacts of RDE on a wider set of algorithms could also prove to be insightful. As an example, we suggest a possible improvement on temperature annealing in Appendix A.

A Algorithms

A.1 Pseudocode for RDE with SAC

The algorithm below describes the implementation we use of RDE with SAC.

Algorithm 1 Algorithm Pseudocode for RDE+SAC

Require: Learning rate η, Target update frequency f, Target update rate τ, Discount γ, Ensemble size N, Reset frequency T_{reset}, Replay Ratio RR, Coefficient β, Temperature $temp$, Target update weight τ.

Initialize agent parameters $\{\theta_1, ..., \theta_N\}$, where each agent θ_i has policy parameters ξ_i, target policy parameters ξ_i^-, parameters for two Q-networks $\phi_{i,1}$ and $\phi_{i,2}$ (which we denote collectively as ϕ_i), and corresponding target Q-network parameters $\phi_{i,1}^-$ and $\phi_{i,2}^-$.
Initialize the replay buffer $\mathcal{B}$.
for each episode **do**
    for each timestep t **do**
        for $i = 1$ to N **do**
            Based on state s_t, select an action a_t^i using current policies π_{θ_i}.
        end for
        Calculate p_{select} using Eq. 1.
        Sample action a_t from p_{select} and play it.
        Store observed transition (s_t, a_t, r_t, s_{t+1}) in the replay buffer $\mathcal{B}$.
        for $j = 1$ to RR **do**
            Sample a random minibatch from $\mathcal{B}$.
            for $i = 1$ to N **do**
                Take gradient steps of size η to update ϕ_i to minimize:

$$L(\phi_i, \mathcal{B}) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{U}(\mathcal{B})}\left[\left(Q_{target} - Q(s, a, \phi_i)\right)^2\right]$$

where for each (s, a, r, s') pair, $Q_{target} = r + \gamma(Q(s, a, \phi_i^-) + temp * \mathcal{H}^\pi)$.
                if $t\%f == 0$ **then**
                    Take a gradient steps of size η to update the policy ξ_i to minimize:

$$L(\xi_i, \mathcal{B}) = \mathbb{E}_{s\sim\mathcal{U}(\mathcal{B}), a\sim\pi_{\xi_i}}\left[temp * \log(\pi_{\xi_i}(a|s)) - Q_{\phi_i}(s, a)\right]$$

                    Update $\phi_i^- \leftarrow \tau\phi_i + (1-\tau)\phi_i^-$
                    Update $\xi_i^- \leftarrow \tau\xi_i + (1-\tau)\xi_i^-$
                end if
            end for
        end for
        if t % $(T_{reset}/N) == 0$ **then**
            Reset θ_k and set $k \leftarrow (k+1)\%N$
        end if
    end for
end for

In addition to the algorithm above, which we directly apply to the `Mountain Car Continuous` environment, we also implement target entropy annealing and a change of variables in the `Cheetah-Run` environment. We find that these features are necessary for TD3 to converge, and we explain them in the following sections.

A.1.1 Target Entropy Annealing

The first technical addition in our TD3 algorithm is target entropy annealing. Without temperature annealing, the only regularization term in our loss function in the form of a constant temperature times the entropy of the distribution from which we select actions at a particular state. However, as the optimizer attempts to minimize the loss on some finite-sized empirical sample of actions and rewards, there are opportunities to overfit to the sample. The resulting prediction for the probability distribution of actions given that state will then have less entropy, which can be undesirable if we are not certain that the sample was representative. We would ideally like to achieve the minimium error as determined by the loss function while retaining maximum entropy, where the latter objective is one that penalizes overfitting, for low entropy is a first order estimate of overfitting. Therefore, we can improve performance by using our prior knowledge to select a target or goal entropy of our distributions. Throughout the optimization, we dynamically change the temperature to encourage the policy's entropy (according to [Haa+19]) at each state to approach the target entropy via regularization term in loss function. Formally, we append the following term to our loss function

$$R = \alpha(\mathcal{H} - \mathcal{H}_{\text{goal}}) \tag{2}$$

where α specifies the temperature parameter which is tuned by the optimizer so as to minimize loss. We can see that this has the desirable characteristic that when $\mathcal{H} < \mathcal{H}_{\text{goal}}$, α will be increased to minimize R, prioritizing an increase in entropy. Otherwise when $\mathcal{H} > \mathcal{H}_{\text{goal}}$, α will be decreased to minimize R, prioritizing a decrease in the other terms of the loss function, (typically at the expense of reducing entropy). We choose $\mathcal{H}_{\text{goal}}$ to be the negative of the dimension of the action space, in agreement with [Kim+23]. That is, we choose some value that is proportional to the entropy of the action space, so that the optimizer will be able to more easily tune α. Note that negative entropy is not an issue because this is the differential entropy, which is standard practice in reinforcement learning, even though it is not a dimensionless quantity: it has units of length. Therefore, care should be taken to scale the entropy by the measure of the action space. For `Cheetah-Run` environment we consider, the action space consists of unit discs on the real line $[-1, 1]$, so the scaling is up to a small constant already normalized by the measure of the action space (the optimizer can handle the necessary small correction).

A.1.2 Change of Variables

To accomplish the change of variables suggestion by [Haa+19], we draw actions a according to the current state and policy, where $u \sim N(\text{mean}, \text{variance})$ for some Multivariate Gaussian distribution, and $a = \tanh u$. By composing these maps to select an action, we are altering the entropy of the effective distribution which generates the samples, and we needed to account for this for the model to be successful. The modified entropy is

$$\begin{aligned}
\mathcal{H}(\pi(a \mid s)) &= \mathbb{E} - \log \pi(a \mid s) \\
&= H(N(\text{mean}, \text{variance})) + \log \|\mathbf{D}_u \tanh u\| \\
&= H(N(\text{mean}, \text{variance})) + \mathbb{E} * \sum_{i=1}^{D} (1 - \tanh(u)^2) + \log 1
\end{aligned}$$

where the final term is zero since the action space is already normalized to 1. In our code, this is implemented in an automatically differentiable manner via PyTorch's `distribution.transform(tanh)` and `distribution.log_abs_det_jacobian(a, u)`.

A.1.3 Suggestions to Improve Target Entropy Annealing

For future work, we also suggest how the target entropy annealing strategy described in Appendix A.1.1 can be improved. It is typical in reinforcement learning implementations to choose H_{goal} to be a constant that is reflective of our prior knowledge, in many cases setting it to some linear function of the dimensional of the action space. However, it is a somewhat lazy application of prior knowledge to have a goal entropy that is uniform throughout the state action space. The optimizer will need to work harder to tune the temperature α to overcome this bad approximation. To give a concrete example, suppose the problem at hand is to balance a pole on a cart. When the pole is approximately upright, the choice is not clear which direction to move the cart. For it could have just started falling in the direction of its tilt, or perhaps it experienced an acceleration at an earlier time that is now realized as an angular velocity $\omega > 0$ when the displacement is $\Delta\theta < 0$, so this tilt will soon disappear without external interaction. Therefore, in states with $\Delta\theta \approx 0$ a high target entropy is desirable.[1] In other states, say with large $|\Delta\theta|$, the optimal action is clear and the cart should move in the direction to minimize $|\Delta\theta|$ regardless of the velocity ω as the risk, or in the safety RL context the cost, of not performing corrective action is too large. These states are better suited with a low goal entropy assignment. Making this improvements to the target entropy annealing model would significantly ease the problem given to the optimizer of adaptive selection of the best temperature. However, in general, it can be burdensome to implement this improved target entropy annealing because it can be difficult to determine which states should be assigned low entropy.

[1] That is, high entorpy in the direction of movement. The magnitude of the external force applied to balance the pole should still be small, as desired for most states in this problem.

A.2 Pseudocode for RDE with SAC

Next, we display pseudocode for our implementation of RDE with DQN.

Algorithm 2 Algorithm Pseudocode for RDE+DQN

Require: Epsilon ϵ, Learning rate η, Target update frequency f, Discount γ, Ensemble size N, Reset frequency T_{reset}, Replay Ratio RR, Coefficient β.

Initialize Q-function parameters $\{\theta_1, ..., \theta_N\}$ and target Q-function parameters $\{\theta_1^-, ..., \theta_N^-\}$.
Initialize the replay buffer $\mathcal{B}$.
for each episode **do**
    for each timestep t **do**
        for $i = 1$ to N **do**
            Based on state s_t, select an action a_t^i according to the ϵ-greedy policy.
        end for
        Calculate p_{select} using Eq. 1.
        Sample action a_t from p_{select} and play it.
        Store observed transition (s_t, a_t, r_t, s_{t+1}) in the replay buffer $\mathcal{B}$.
        for $j = 1$ to RR **do**
            Sample a random minibatch from $\mathcal{B}$.
            for $i = 1$ to N **do**
                Take gradient step of size η to update θ_i to minimize:

$$L(\theta, \mathcal{B}) = \mathbb{E}_{(s,a,r,s') \sim \mathcal{U}(\mathcal{B})} \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a, \theta_i) \right)^2 \right]$$

            end for
            if $f \% t == 0$ **then**
                Copy $\{\theta_1, ..., \theta_N\}$ into $\{\theta_1^-, ..., \theta_N^-\}$
            end if
        end for
        if $t \;\%\; (T_{reset}/N) == 0$ **then**
            Reset θ_k and set $k \leftarrow (k+1)\%N$
        end if
    end for
end for

B Hyperparameters

In the Table 1 below, we display the hyperparameters we used to train RDE with SAC in the `Cheetah-Run` setting. As we aim to reproduce the results of [Kim+23] exactly, almost all of our network parameters are identical to theirs. However, for the sake of decreasing computational complexity, we adjust a few of their hyperparameters. Namely, we use a minibatch size of 512 instead of 1024 and 256 hidden units in each network layer instead of 1024. Note that the network layers are multi-layer perceptions (MLPs). Moreover, we also implement a delayed network update frequency by setting $f = 2$ unlike [Kim+23] who uses $f = 1$.

Table 1: Hyperparameters for RDE with SAC in the `Cheetah-Run` Setting

Hyperparameter	Value
# of Ensemble Agents N	2
Training steps	$(1 \times 10^6, 5 \times 10^5)$
Discount factor	0.99
Initial collection steps	5000
Minibatch size	512
Optimizer	Adam
Learning rate η	0.0003
Network activation (all)	ReLU
Network layer type (all)	MLP
Number of hidden network layers (all)	2
Number of hidden units per network layer (all)	256
Initial temperature	1
Replay buffer size	1×10^6
Replay ratio	$(1, 2, 4)$
Target update frequency f	2
Target update weight τ	0.005
Reset interval T_{reset}	$(4 \times 10^5, 2 \times 10^5, 1 \times 10^5)$
Action selection coefficient β	50

In Table 2 below, we display the hyperparameters we used in the `Mountain Car Continuous` environment when using RDE with SAC. Our parameters are very similar to those in Table 1 except because the `Mountain Car Continuous` environment is easier to learn than the `Cheetah-Run` setting, we use fewer training steps and hence also use smaller values for T_{reset}. Lastly, we display the hyperparameters we used to train RDE with DQN in the `Cart Pole` environment below in Table 3.

Table 2: Hyperparameters for RDE with SAC in the `Mountain Car Continuous` Setting

Hyperparameter	Value
# of Ensemble Agents N	2
Training steps	$(2 \times 10^5, 1 \times 10^5)$
Discount factor	0.99
Initial collection steps	5000
Minibatch size	512
Optimizer	Adam
Learning rate η	0.0003
Network activation (all)	ReLU
Network layer type (all)	MLP
Number of hidden network layers (all)	2
Number of hidden units per network layer (all)	256
Temperature	0.01
Replay buffer size	2×10^5
Replay ratio	$(1, 2, 4)$
Target update frequency f	2
Target update weight τ	0.005
Reset interval T_{reset}	$(8 \times 10^4, 4 \times 10^4, 2 \times 10^4)$
Action selection coefficient β	50

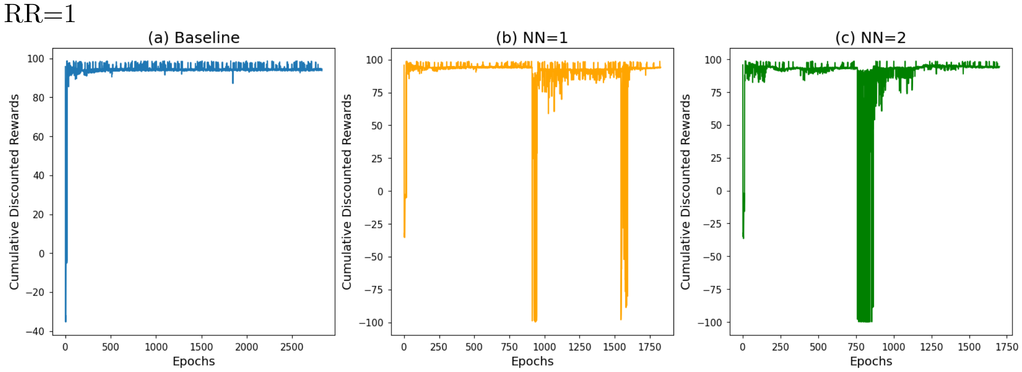
Table 3: Hyperparameters for RDE with DQN in the `Cart Pole` Setting

Hyperparameter	Value
# of Ensemble Agents N	2
Training steps	$(2 \times 10^5, 1 \times 10^5)$
Discount factor	0.99
Initial collection steps	1×10^4
Minibatch size	32
Optimizer	Adam
	$0.9 \to 0.05$
decay time step	10^5
Learning rate η	0.0005
Network activation (all)	ReLU
Network layer type (all)	MLP
Number of hidden network layers (all)	2
Number of hidden units per network layer (all)	128
Replay buffer size	5×10^4
Replay ratio	$(1, 2, 4)$
Target update frequency f	10 episodes
Network testing frequency	20 episodes
Reset interval T_{reset}	$(8 \times 10^4, 4 \times 10^4)$
Action selection coefficient β	0.01

C Hyperparameters

In Figure 5, for a replay ratio of 1, we display the results of our performance comparison of RDE with SAC using $N = 2$ agents with the naive reset and baseline SAC algorithms.

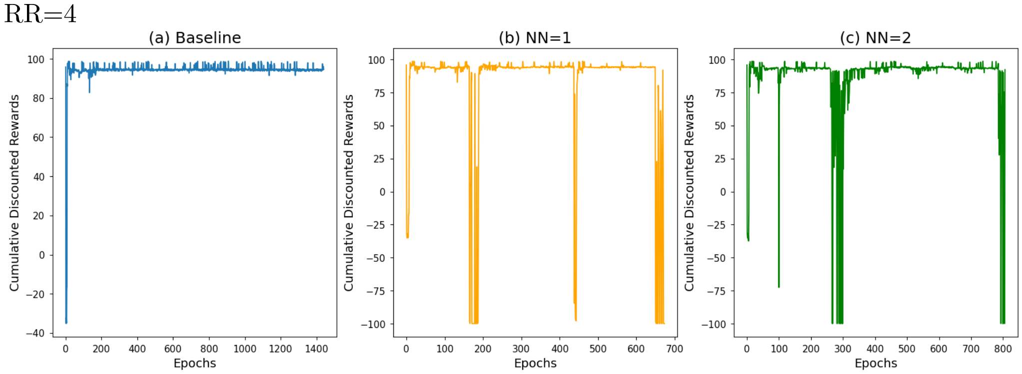
Figure 5: Performance Comparison in the `Mountain Car Continuous` Setting when RR=1



This figure displays our results in the `Mountain Car Continuous` setting for SAC. We make comparisons across the baseline case, with naive resets with one agent, and with ensemble agents using RDE with two agents.

Next, in Figure 6, we show the results of our performance comparison when the replay ratio is 4.

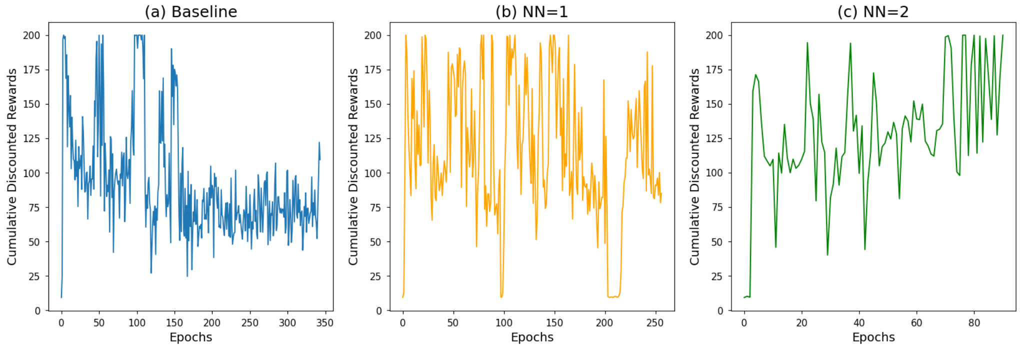
Figure 6: Performance Comparison in the `Mountain Car Continuous` Setting when RR=4



We compare the performance of SAC with a replay ratio of four in the `Mountain Car Continuous` across the baseline case, with naive resets with one agent, and with ensemble agents using RDE with two agents.

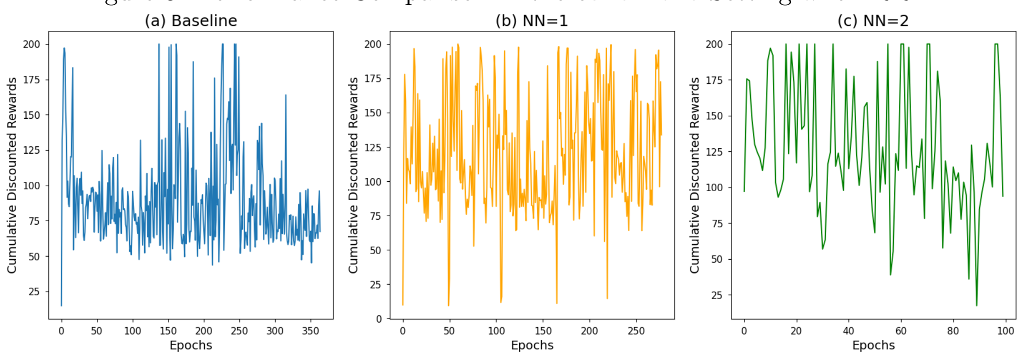
Next, in Figures 7 and 8, we display our results for replay ratios of 1 and 4 in the `Cart Pole` environment.

Figure 7: Performance Comparison in the `Cart Pole` Setting when RR=1



For a replay ratio of one, we compare performance in the `Cart Pole` setting for DQN. We consider the baseline case, with naive resets with one agent, and with ensemble agents using RDE with two agents.

Figure 8: Performance Comparison in the `Cart Pole` Setting when RR=4



We display our results in the `Cart Pole` setting for DQN when the replay ratio is four. We examine the baseline case, with naive resets with one agent, and with ensemble agents using RDE with two agents.

D Compute Times

In the tables below, we report the compute times of our experiments in each of the three environments: `Cheetah-Run`, `Mountain Car Continuous`, and `Cart Pole`.

Table 4: Run Times for SAC in the `Cheetah-Run` Setting on GPU

	Baseline	Naive Reset	RDE
$RR = 1$ for 1×10^6 time steps	3 hrs	6 hrs	12 hrs
$RR = 2$ for 1×10^6 time steps	6 hrs	12 hrs	24 hrs
$RR = 4$ for 5×10^5 time steps	6 hrs	12 hrs	24 hrs

Table 5: Run Times for SAC in the `Mountain Car Continuous` Setting on CPU

	Baseline	Naive Reset	RDE
$RR = 1$ for 2×10^5 time steps	1.5 hrs	3 hrs	6 hrs
$RR = 2$ for 2×10^5 time steps	3 hrs	6 hrs	12 hrs
$RR = 4$ for 1×10^5 time steps	3 hrs	6 hrs	12 hrs

Table 6: Run Times for DQN in the `Cart Pole` Setting on CPU

	Baseline	Naive Reset	RDE
$RR = 1$ for 1×10^6 time steps	1 hr	2 hrs	4 hrs
$RR = 2$ for 1×10^6 time steps	2 hrs	4 hrs	8 hrs
$RR = 4$ for 5×10^6 time steps	2 hrs	4 hrs	8 hrs

E Code

In this section, we display the code we implemented for RDE with SAC and DQN in the `Cheetah-Run`, `Mountain Car Continuous`, and `Cart Pole` environments. We note that parts of our code are adapted from our solutions for assignments 4 and 7.

Listing 1: Code for RDE with SAC in the `Cheetah-Run` Setting

```python
#!/usr/bin/env python
# coding: utf-8

# In[1]:

# @title Run to install MuJoCo and `dm_control`
import distutils.util
import os
import subprocess

if subprocess.run("nvidia-smi").returncode:
    raise RuntimeError(
        "Cannot communicate with GPU. "
        "Make sure you are using a GPU Colab runtime. "
        "Go to the Runtime menu and select Choose runtime type."
    )

print("Installing dm_control...")
get_ipython().system("pip install -q dm_control>=1.0.18")

# Configure dm_control to use the EGL rendering backend (requires GPU)
get_ipython().run_line_magic("env", "MUJOCO_GL=egl")

print("Checking that the dm_control installation succeeded...")
try:
    from dm_control import suite

    env = suite.load("cartpole", "swingup")
    pixels = env.physics.render()
except Exception as e:
    raise e from RuntimeError(
        "Something went wrong during installation. Check the shell output above "
        "for more information.\n"
        "If using a hosted Colab runtime, make sure you enable GPU acceleration "
        'by going to the Runtime menu and selecting "Choose runtime type".'
    )
else:
    del pixels, suite

get_ipython().system(
    'echo Installed dm_control $(pip show dm_control | grep -Po "(?<=Version: ).+")'
)

# In[2]:

# @title All `dm_control` imports required for this tutorial

# The basic mujoco wrapper.
from dm_control import mujoco

# Access to enums and MuJoCo library functions.
from dm_control.mujoco.wrapper.mjbindings import enums
from dm_control.mujoco.wrapper.mjbindings import mjlib

# PyMJCF
from dm_control import mjcf

# Composer high level imports
from dm_control import composer
from dm_control.composer.observation import observable
from dm_control.composer import variation

# Imports for Composer tutorial example
from dm_control.composer.variation import distributions
from dm_control.composer.variation import noises
from dm_control.locomotion.arenas import floors

# Control Suite
```

```python
from dm_control import suite

# Run through corridor example
from dm_control.locomotion.walkers import cmu_humanoid
from dm_control.locomotion.arenas import corridors as corridor_arenas
from dm_control.locomotion.tasks import corridors as corridor_tasks

# Soccer
from dm_control.locomotion import soccer

# Manipulation
from dm_control import manipulation

# In[3]:

import random
import numpy as np
import torch
import torch.nn as nn
from torch.distributions import Categorical
import torch.nn.functional as F
import torch.optim as optim
import matplotlib.pyplot as plt
import sys
import copy
from typing import Tuple

get_ipython().run_line_magic("matplotlib", "inline")
import pickle

# In[4]:

class ReplayBuffer(object):
    def __init__(self, state_dim, action_dim, max_size=int(1e6)):
        self.max_size = max_size
        self.ptr = 0
        self.size = 0

        self.state = np.zeros((max_size, state_dim))
        self.action = np.zeros((max_size, action_dim))
        self.next_state = np.zeros((max_size, state_dim))
        self.reward = np.zeros((max_size, 1))
        self.not_done = np.zeros((max_size, 1))

        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    def add(self, state, action, next_state, reward, done):
        self.state[self.ptr] = state
        self.action[self.ptr] = action
        self.next_state[self.ptr] = next_state
        self.reward[self.ptr] = reward
        self.not_done[self.ptr] = 1.0 - done

        self.ptr = (self.ptr + 1) % self.max_size
        self.size = min(self.size + 1, self.max_size)

    def sample(self, batch_size):
        ind = np.random.randint(0, self.size, size=batch_size)

        return (
            torch.FloatTensor(self.state[ind]).to(self.device),
            torch.FloatTensor(self.action[ind]).to(self.device),
            torch.FloatTensor(self.next_state[ind]).to(self.device),
            torch.FloatTensor(self.reward[ind]).to(self.device),
            torch.FloatTensor(self.not_done[ind]).to(self.device),
        )

# In[5]:

def init_flags():

    flags = {
        "env": "cheetah",
        "env_action": "run",
        "seed": 0,  # random seed
        "start_timesteps": 5e3,  # total steps of free exploration phase
        "max_timesteps": 6e5,  # maximum length of time steps in training
```

```
        "batch_size": 512,
        "discount": 0.99,
        "tau": 0.005,  # rate of target update
        "policy_freq": 2,  # delayed policy update frequency in TD3,
        "N": 2,  # number of agents,
        "RR": 1,  # replay ratio,
        "T": 4e5,  # time steps between agent resets ,
        "beta": 1,  # action selection coefficient,
        "actor_model_file": None,  # "actor_model.pt",
        "critic_model_file": None,  # "critic_model.pt",
        "temp_model_file": None,  # "temp_model.pt"
    }

    return flags

def collect_actions(theta, state):
    actions = []
    for theta_i in theta:
        action = theta_i.select_action(np.array(state))
        actions.append(torch.from_numpy(action))
    return actions

def get_timestep_info(timestep):
    ob = timestep.observation
    state = np.concatenate([*ob.values()]).ravel()
    return state, timestep.reward, timestep.last()

def main(policy_name="DDPG") -> list:
    """
    Input:
    policy_name: str, the method to implement
    Output:
    evaluations: list, the reward in every episodes
    Call DDPG/TD3 trainer and
    """
    args = init_flags()
    env = suite.load(
        args["env"], args["env_action"], task_kwargs={"random": args["seed"] + 100}
    )
    torch.manual_seed(args["seed"])
    np.random.seed(args["seed"])

    action_spec = env.action_spec()
    ob_spec = env.observation_spec()

    state_dim = 0
    for _, val in ob_spec.items():
        state_dim += val.shape[0]
    state_dim

    action_dim = action_spec.shape[0]
    max_action = action_spec.maximum
    kwargs = {
        "state_dim": state_dim,
        "action_dim": action_dim,
        "max_action": max_action,
        "discount": args["discount"],
        "tau": args["tau"],
        "actor_model_file": args["actor_model_file"],
        "critic_model_file": args["critic_model_file"],
        "temp_model_file": args["temp_model_file"],
    }
    if policy_name == "TD3":
        kwargs["policy_freq"] = args["policy_freq"]
        theta = [TD3(**kwargs) for _ in range(args["N"])]
    elif policy_name == "DDPG":
        policy = DDPG(**kwargs)

    replay_buffer = ReplayBuffer(state_dim, action_dim)
    evaluations = []
    timestep = env.reset()
    state, _, _ = get_timestep_info(timestep)
    done = False
    episode_reward = 0
    episode_timesteps = 0
    episode_num = 0
    k = 0
```

```
    max_episode_steps = 990

    for t in range(int(args["max_timesteps"])):

        episode_timesteps += 1

        if t % 1e5 == 0:
            theta[0].save_actor_model(filename="actor_model_2.pt")
            theta[0].save_critic_model(filename="critic_model_2.pt")
            theta[0].save_temp_model(filename="temp_model_2.pt")
            np.savetxt("reward_cheetah_run_N2_RR2.csv", evaluations, delimiter=",")

        # Select action randomly or according to policy
        entropy = 0
        mean = 0
        vari = 0
        if t < args["start_timesteps"]:
            action = np.random.uniform(
                action_spec.minimum, action_spec.maximum, size=action_spec.shape
            )
        else:
            with torch.no_grad():

                actions = collect_actions(theta, state)
                # compute Q's, then apply softmax
                q_sa = torch.hstack(
                    [
                        theta[k].critic.Q1(torch.FloatTensor(state), action)
                        for action in actions
                    ]
                )
                # dim
                max_q_sa, _ = torch.max(q_sa, dim=0)
                alpha = args["beta"] / max_q_sa
                p_select = F.softmax(q_sa / alpha)

                if t % 1000 == 0:
                    print("temperature:", torch.exp(theta[0].log_alpha))

                rng = np.random.default_rng()
                action = rng.choice(
                    a=[np.array(tensor) for tensor in actions],
                    p=p_select.numpy(),
                    axis=0,
                )
                action = np.atleast_1d(action)

        # Perform action
        next_state, reward, done = get_timestep_info(env.step(action))
        done_bool = float(done) if episode_timesteps < max_episode_steps else 0

        # Store data in replay buffer
        replay_buffer.add(state, action, next_state, reward, done_bool)

        state = next_state
        episode_reward += reward

        # Train agent after collecting sufficient data
        if t >= args["start_timesteps"]:
            for j in range(args["RR"]):
                for theta_i in theta:
                    theta_i.train(replay_buffer, args["batch_size"])

            if (t % (args["T"] / args["N"])) == 0:
                print(k)
                # reset just actor or both? ask to confirm
                theta[k].actor.reset()
                theta[k].critic.reset()
                k = (k + 1) % args["N"]

        if done:
            # +1 to account for 0 indexing. +0 on ep_timesteps since it will increment +1 even if done=True
            print(
                f"Total T: {t+1} Episode Num: {episode_num+1} Episode T: {episode_timesteps} Reward: {episode_reward:.3f}"
            )
```

```
            evaluations.append(episode_reward)
            # Reset environment
            done = False
            timestep = env.reset()
            state, _, _ = get_timestep_info(timestep)
            episode_reward = 0
            episode_timesteps = 0
            episode_num += 1

    return evaluations

# In[6]:

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Construct the actor/critic network for TD3
class Actor_TD3(nn.Module):
    def __init__(self, state_dim: int, action_dim: int, max_action: float):

        super(Actor_TD3, self).__init__()
        self.l1 = nn.Linear(state_dim, 256)
        self.l2 = nn.Linear(256, 256)
        self.l3 = nn.Linear(256, 2 * action_dim)

        if isinstance(max_action, np.ndarray):
            max_action = torch.from_numpy(max_action)
        else:
            assert isinstance(max_action, list)
        max_action = torch.tensor(max_action)
        self.max_action = max_action.to(torch.float32)
        self.action_dim = action_dim

    def forward(self, state: torch.Tensor) -> torch.Tensor:
        if isinstance(state, np.ndarray):
            state = torch.from_numpy(state)

        state = state.float()
        a = F.relu(self.l1(state))
        a = F.relu(self.l2(a))

        a = self.l3(a)

        mean = a[:, : self.action_dim]
        cov = nn.functional.softplus(a[:, self.action_dim :]) + 1e-9

        dist = torch.distributions.MultivariateNormal(
            mean, scale_tril=torch.diag_embed(cov)
        )
        return dist

    def reset(self):
        for layer in self.children():
            if hasattr(layer, "reset_parameters"):
                layer.reset_parameters()

class Critic_TD3(nn.Module):
    def __init__(self, state_dim: int, action_dim: int):
        super(Critic_TD3, self).__init__()

        # Q1 architecture
        self.l1 = nn.Linear(state_dim + action_dim, 256)
        self.l2 = nn.Linear(256, 256)
        self.l3 = nn.Linear(256, 1)

        # Q2 architecture
        self.l4 = nn.Linear(state_dim + action_dim, 256)
        self.l5 = nn.Linear(256, 256)
        self.l6 = nn.Linear(256, 1)

    def forward(
        self, state: torch.Tensor, action: torch.Tensor
    ) -> Tuple[torch.Tensor, torch.Tensor]:
        sa = torch.cat([state, action], 1)
        q1 = F.relu(self.l1(sa))
        q1 = F.relu(self.l2(q1))
        q1 = self.l3(q1)
```

```python
        q2 = F.relu(self.l4(sa))
        q2 = F.relu(self.l5(q2))
        q2 = self.l6(q2)
        return q1, q2

    def Q1(self, state: torch.Tensor, action: torch.Tensor) -> torch.Tensor:
        sa = torch.cat([state, action], -1)

        q1 = F.relu(self.l1(sa))
        q1 = F.relu(self.l2(q1))
        q1 = self.l3(q1)
        return q1

    def reset(self):
        for layer in self.children():
            if hasattr(layer, "reset_parameters"):
                layer.reset_parameters()

# In[7]:

class TD3(object):
    def __init__(
        self,
        state_dim: int,
        action_dim: int,
        max_action: float,
        discount=0.99,
        tau=0.005,
        policy_freq=2,
        init_temperature=1,
        actor_model_file=None,
        critic_model_file=None,
        temp_model_file=None,
    ):

        self.actor = Actor_TD3(state_dim, action_dim, max_action).to(device)
        self.actor_target = copy.deepcopy(self.actor)
        self.actor_optimizer = torch.optim.Adam(self.actor.parameters(), lr=3e-4)

        self.critic = Critic_TD3(state_dim, action_dim).to(device)
        self.critic_target = copy.deepcopy(self.critic)
        self.critic_optimizer = torch.optim.Adam(self.critic.parameters(), lr=3e-4)

        if actor_model_file is not None:
            print("Loading pretrained model from", actor_model_file)
            self.load_pytorch_actor_model(actor_model_file)

        if critic_model_file is not None:
            print("Loading pretrained model from", critic_model_file)
            self.load_pytorch_critic_model(critic_model_file)

        self.log_alpha = torch.tensor(np.log(init_temperature)).to(device)
        self.log_alpha.requires_grad = True
        self.target_entropy = -1 * action_dim
        self.log_alpha_optimizer = torch.optim.Adam([self.log_alpha], lr=3e-4)

        if temp_model_file is not None:
            print("Loading pretrained model from", temp_model_file)
            self.load_pytorch_temp_model(temp_model_file)

        self.max_action = self.actor.max_action
        self.discount = discount
        self.tau = tau
        self.policy_freq = policy_freq

        self.total_it = 0

        self.transform = torch.distributions.transforms.ComposeTransform(
            [
                torch.distributions.transforms.TanhTransform(),
                torch.distributions.transforms.AffineTransform(
                    loc=0, scale=self.max_action
                ),
            ]
        )

    def select_action(self, state: torch.Tensor) -> torch.Tensor:
```

```
        state = torch.FloatTensor(state.reshape(1, -1)).to(device)

        actor_dist = self.actor(state)
        selected_action = self.transform(actor_dist.rsample())
        return selected_action.data.numpy().flatten()

    def train(self, replay_buffer, batch_size=256):
        self.total_it += 1
        temperature = torch.exp(self.log_alpha)

        # Sample replay buffer
        state, action, next_state, reward, not_done = replay_buffer.sample(batch_size)

        with torch.no_grad():
            # Select action according to the policy,
            target_actor_dist = self.actor_target(next_state)
            raw_next_action = target_actor_dist.rsample()
            next_action = self.transform(raw_next_action)
            target_entropy = (
                target_actor_dist.entropy()
                + self.transform.log_abs_det_jacobian(raw_next_action, next_action).sum(
                    -1
                )
            ).unsqueeze(-1)
            # Compute the target Q value
            target_Q1, target_Q2 = self.critic_target(next_state, next_action)
            target_Q = torch.min(target_Q1, target_Q2)
            target_Q = reward + not_done * self.discount * (
                target_Q + (temperature * target_entropy)
            )
            target_Q = target_Q.detach()
        # Get current Q estimates
        current_Q1, current_Q2 = self.critic(state, action)

        # Compute critic loss
        critic_loss = F.mse_loss(current_Q1, target_Q) + F.mse_loss(
            current_Q2, target_Q
        )

        # Optimize the critic
        self.critic_optimizer.zero_grad()
        critic_loss.backward()
        self.critic_optimizer.step()

        # Delayed policy updates
        if self.total_it % self.policy_freq == 0:
            # don't have to compute critic gradients
            for param in self.critic.parameters():
                param.requires_grad = False

            actor_dist = self.actor(state)
            raw_action = actor_dist.rsample()
            selected_action = self.transform(raw_action)
            actor_entropy = (
                actor_dist.entropy()
                + self.transform.log_abs_det_jacobian(raw_action, selected_action).sum(
                    -1
                )
            ).unsqueeze(-1)

            alpha_loss = (
                temperature * (actor_entropy - self.target_entropy).mean().detach()
            )
            actor_loss = -(
                self.critic.Q1(state, selected_action)
                + (temperature.detach() * actor_entropy)
            ).mean()

            alpha_actor_loss = alpha_loss + actor_loss

            # Optimize the actor
            self.actor_optimizer.zero_grad()
            self.log_alpha_optimizer.zero_grad()
            alpha_actor_loss.backward()
            self.actor_optimizer.step()
            self.log_alpha_optimizer.step()
```

```
            for param in self.critic.parameters():
                param.requires_grad = True

            # Update the frozen target models
            for param, target_param in zip(
                self.critic.parameters(), self.critic_target.parameters()
            ):
                new_target_params = (
                    self.tau * param.data + (1 - self.tau) * target_param.data
                )
                target_param.data.copy_(new_target_params)

            for param, target_param in zip(
                self.actor.parameters(), self.actor_target.parameters()
            ):
                new_target_params = (
                    self.tau * param.data + (1 - self.tau) * target_param.data
                )
                target_param.data.copy_(new_target_params)

    def save_actor_model(self, filename):
        # pytorch model object like an instance of Actor
        checkpoint = {
            "model_state_dict": self.actor.state_dict(),
            "optimizer_state_dict": self.actor_optimizer.state_dict(),
        }
        torch.save(checkpoint, filename)

    def save_critic_model(self, filename):
        # pytorch model object like an instance of Actor
        checkpoint = {
            "model_state_dict": self.critic.state_dict(),
            "optimizer_state_dict": self.critic_optimizer.state_dict(),
        }
        torch.save(checkpoint, filename)

    def save_temp_model(self, filename):
        # pytorch model object like an instance of Actor
        checkpoint = {
            "model_state_dict": self.log_alpha,
            "optimizer_state_dict": self.log_alpha_optimizer.state_dict(),
        }
        torch.save(checkpoint, filename)

    def load_pytorch_actor_model(self, filename):
        # now to start up a model with the same trained parameters
        checkpoint = torch.load(filename)
        self.actor.load_state_dict(checkpoint["model_state_dict"])
        self.actor_optimizer.load_state_dict(checkpoint["optimizer_state_dict"])
        # model.eval()

    def load_pytorch_critic_model(self, filename):
        # now to start up a model with the same trained parameters
        checkpoint = torch.load(filename)
        self.critic.load_state_dict(checkpoint["model_state_dict"])
        self.critic_optimizer.load_state_dict(checkpoint["optimizer_state_dict"])
        # model.eval()

    def load_pytorch_temp_model(self, filename):
        # now to start up a model with the same trained parameters
        checkpoint = torch.load(filename)
        self.log_alpha = checkpoint["model_state_dict"]
        self.log_alpha_optimizer.load_state_dict(checkpoint["optimizer_state_dict"])
        # model.eval()

# In[ ]:

evaluation_td3 = main(policy_name="TD3")
```

Listing 2: Code for RDE with SAC in the Mountain Car Continuous

```
#!/usr/bin/env python
# coding: utf-8

# In[ ]:

get_ipython().system("pip3 install swig > /dev/null 2>&1")
```

```
get_ipython().system("pip3 uninstall box2d-py -y > /dev/null 2>&1")
get_ipython().system("pip3 install box2d-py > /dev/null 2>&1")
get_ipython().system("pip3 install box2d box2d-kengz > /dev/null 2>&1")
get_ipython().system("apt install xvfb > /dev/null 2>&1")
get_ipython().system("pip3 install pyvirtualdisplay > /dev/null 2>&1")
get_ipython().system("pip3 install gym==0.25.0 > /dev/null 2>&1")

# In[ ]:

import gym
import random
import numpy as np
import torch
import torch.nn as nn
from torch.distributions import Categorical
import torch.nn.functional as F
import torch.optim as optim
import matplotlib.pyplot as plt
import sys
from pyvirtualdisplay import Display
from IPython import display as disp
import copy
from typing import Tuple

get_ipython().run_line_magic("matplotlib", "inline")

# In[ ]:

# Replay buffer
class ReplayBuffer(object):
    def __init__(self, state_dim, action_dim, max_size=int(2e5)):
        self.max_size = max_size
        self.ptr = 0
        self.size = 0

        self.state = np.zeros((max_size, state_dim))
        self.action = np.zeros((max_size, action_dim))
        self.next_state = np.zeros((max_size, state_dim))
        self.reward = np.zeros((max_size, 1))
        self.not_done = np.zeros((max_size, 1))

        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    def add(self, state, action, next_state, reward, done):
        self.state[self.ptr] = state
        self.action[self.ptr] = action
        self.next_state[self.ptr] = next_state
        self.reward[self.ptr] = reward
        self.not_done[self.ptr] = 1.0 - done

        self.ptr = (self.ptr + 1) % self.max_size
        self.size = min(self.size + 1, self.max_size)

    def sample(self, batch_size):
        ind = np.random.randint(0, self.size, size=batch_size)

        return (
            torch.FloatTensor(self.state[ind]).to(self.device),
            torch.FloatTensor(self.action[ind]).to(self.device),
            torch.FloatTensor(self.next_state[ind]).to(self.device),
            torch.FloatTensor(self.reward[ind]).to(self.device),
            torch.FloatTensor(self.not_done[ind]).to(self.device),
        )

# In[ ]:

def init_flags():

    flags = {
        "env": "MountainCarContinuous",
        "seed": 0,  # random seed
        "start_timesteps": 5e3,  # total steps of free exploration phase
        "max_timesteps": 2e5,  # maximum length of time steps in training
        "expl_noise": 0.1,  # noise strength in exploration
        "batch_size": 512,
        "discount": 0.99,
        "tau": 0.005,  # rate of target update
```

```
        "policy_freq": 2,  # delayed policy update frequency in TD3,
        "N": 1,  # number of agents,
        "RR": 2,  # replay ratio,
        "T": np.inf,  # time steps between agent resets (every 8e4 for RR=1) ,
        "beta": 50,  # action selection coefficient
    }

    return flags

def collect_actions(theta, state):
    actions = []
    for theta_i in theta:
        action, entropy, mean, vari = theta_i.select_action(np.array(state))
        actions.append(torch.from_numpy(action))
    return actions

def main(policy_name="DDPG") -> list:
    """
    Input:
    policy_name: str, the method to implement
    Output:
    evaluations: list, the reward in every episodes
    Call DDPG/TD3 trainer and
    """
    args = init_flags()
    env = gym.make(args["env"])
    env.seed(args["seed"] + 100)
    env.action_space.seed(args["seed"])
    torch.manual_seed(args["seed"])
    np.random.seed(args["seed"])

    state_dim = env.observation_space.shape[0]
    action_dim = env.action_space.shape[0]
    max_action = float(env.action_space.high[0])
    kwargs = {
        "state_dim": state_dim,
        "action_dim": action_dim,
        "max_action": max_action,
        "discount": args["discount"],
        "tau": args["tau"],
    }
    if policy_name == "TD3":
        kwargs["policy_freq"] = args["policy_freq"]
        theta = [TD3(**kwargs) for _ in range(args["N"])]
    elif policy_name == "DDPG":
        policy = DDPG(**kwargs)

    replay_buffer = ReplayBuffer(state_dim, action_dim)
    evaluations = []
    state, done = env.reset(), False
    episode_reward = 0
    episode_timesteps = 0
    episode_num = 0
    k = 0

    for t in range(int(args["max_timesteps"])):

        episode_timesteps += 1

        # Select action randomly or according to policy
        entropy = 0
        mean = 0
        vari = 0
        if t < args["start_timesteps"]:
            action = env.action_space.sample()
        else:
            with torch.no_grad():

                actions = collect_actions(theta, state)
                # compute Q's, then apply softmax
                q_sa = torch.hstack(
                    [
                        theta[k].critic.Q1(torch.FloatTensor(state), action)
                        for action in actions
                    ]
                )
                # dim
```

```
                max_q_sa, _ = torch.max(q_sa, dim=0)
                alpha = args["beta"] / max_q_sa
                p_select = F.softmax(q_sa / alpha)

                if t == args["start_timesteps"]:
                    print(p_select)

                action = np.random.choice(
                    a=torch.hstack(actions).numpy(), p=p_select.numpy()
                )
                action = np.atleast_1d(action)

        # Perform action
        next_state, reward, done, _ = env.step(action)
        done_bool = float(done) if episode_timesteps < env._max_episode_steps else 0

        # Store data in replay buffer
        replay_buffer.add(state, action, next_state, reward, done_bool)

        state = next_state
        episode_reward += reward

        # Train agent after collecting sufficient data
        if t >= args["start_timesteps"]:
            for j in range(args["RR"]):
                for theta_i in theta:
                    theta_i.train(replay_buffer, args["batch_size"])

            if (t % (args["T"] / args["N"])) == 0:
                print(k)
                # reset just actor or both?
                theta[k].actor.reset()
                theta[k].critic.reset()
                k = (k + 1) % args["N"]

        if done:
            # +1 to account for 0 indexing. +0 on ep_timesteps since it will increment +1 even if done=
    True
            print(
                f"Total T: {t+1} Episode Num: {episode_num+1} Episode T: {episode_timesteps} Reward: {
    episode_reward:.3f}"
            )
            evaluations.append(episode_reward)
            entropies = []
            actions_l = []
            means = []
            varis = []
            # Reset environment
            state, done = env.reset(), False
            episode_reward = 0
            episode_timesteps = 0
            episode_num += 1

    return evaluations

# In[ ]:

# Reference Implementation of Twin Delayed Deep Deterministic Policy Gradients (TD3)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Construct the actor/critic network for TD3
class Actor_TD3(nn.Module):
    def __init__(self, state_dim: int, action_dim: int, max_action: float):
        super(Actor_TD3, self).__init__()
        self.l1 = nn.Linear(state_dim, 256)
        self.l2 = nn.Linear(256, 256)
        self.l3 = nn.Linear(256, 2 * action_dim)
        self.max_action = max_action
        self.action_dim = action_dim

    def forward(self, state: torch.Tensor) -> torch.Tensor:
        a = F.relu(self.l1(state))
        a = F.relu(self.l2(a))
        a = self.l3(a)

        mean = self.max_action * torch.tanh(a[:, : self.action_dim])
```

```
        std = nn.functional.softplus(a[:, self.action_dim :]) + 1e-9

        return mean, std

    def reset(self):
        for layer in self.children():
            if hasattr(layer, "reset_parameters"):
                layer.reset_parameters()

class Critic_TD3(nn.Module):
    def __init__(self, state_dim: int, action_dim: int):
        super(Critic_TD3, self).__init__()

        # Q1 architecture
        self.l1 = nn.Linear(state_dim + action_dim, 256)
        self.l2 = nn.Linear(256, 256)
        self.l3 = nn.Linear(256, 1)

        # Q2 architecture
        self.l4 = nn.Linear(state_dim + action_dim, 256)
        self.l5 = nn.Linear(256, 256)
        self.l6 = nn.Linear(256, 1)

    def forward(
        self, state: torch.Tensor, action: torch.Tensor
    ) -> Tuple[torch.Tensor, torch.Tensor]:
        sa = torch.cat([state, action], 1)
        q1 = F.relu(self.l1(sa))
        q1 = F.relu(self.l2(q1))
        q1 = self.l3(q1)

        q2 = F.relu(self.l4(sa))
        q2 = F.relu(self.l5(q2))
        q2 = self.l6(q2)
        return q1, q2

    def Q1(self, state: torch.Tensor, action: torch.Tensor) -> torch.Tensor:
        sa = torch.cat([state, action], -1)

        q1 = F.relu(self.l1(sa))
        q1 = F.relu(self.l2(q1))
        q1 = self.l3(q1)
        return q1

    def reset(self):
        for layer in self.children():
            if hasattr(layer, "reset_parameters"):
                layer.reset_parameters()

# In[ ]:

class TD3(object):
    def __init__(
        self,
        state_dim: int,
        action_dim: int,
        max_action: float,
        discount=0.99,
        tau=0.005,
        policy_freq=2,
        temperature=0.01,
    ):

        self.actor = Actor_TD3(state_dim, action_dim, max_action).to(device)
        self.actor_target = copy.deepcopy(self.actor)
        self.actor_optimizer = torch.optim.Adam(self.actor.parameters(), lr=3e-4)

        self.critic = Critic_TD3(state_dim, action_dim).to(device)
        self.critic_target = copy.deepcopy(self.critic)
        self.critic_optimizer = torch.optim.Adam(self.critic.parameters(), lr=3e-4)

        self.max_action = max_action
        self.discount = discount
        self.tau = tau
        self.policy_freq = policy_freq

        self.total_it = 0
```

```
        self.temperature = temperature

    def select_action(self, state: torch.Tensor) -> torch.Tensor:
        state = torch.FloatTensor(state.reshape(1, -1)).to(device)

        mean, std = self.actor(state)
        actor_dist = torch.distributions.Normal(mean, std)
        selected_action = actor_dist.rsample().clamp(-self.max_action, self.max_action)
        entropy = actor_dist.entropy()
        vari = torch.square(std)
        return (
            selected_action.data.numpy().flatten(),
            entropy.data.numpy(),
            mean.data.numpy(),
            vari.data.numpy(),
        )

    def train(self, replay_buffer, batch_size=256):
        self.total_it += 1

        # Sample replay buffer
        state, action, next_state, reward, not_done = replay_buffer.sample(batch_size)

        with torch.no_grad():
            # Select action according to the policy,
            target_mean, target_std = self.actor_target(next_state)
            target_actor_dist = torch.distributions.Normal(target_mean, target_std)
            next_action = target_actor_dist.rsample().clamp(
                -self.max_action, self.max_action
            )
            target_entropy = target_actor_dist.entropy()
            # Compute the target Q value
            target_Q1, target_Q2 = self.critic_target(next_state, next_action)

            target_Q = torch.min(target_Q1, target_Q2)
            target_Q = reward + not_done * self.discount * (
                target_Q + (self.temperature * target_entropy)
            )
            target_Q = target_Q.detach()

        current_Q1, current_Q2 = self.critic(state, action)
        critic_loss = F.mse_loss(current_Q1, target_Q) + F.mse_loss(
            current_Q2, target_Q
        )

        # Optimize the critic
        self.critic_optimizer.zero_grad()
        critic_loss.backward()
        self.critic_optimizer.step()

        # Delayed policy updates
        if self.total_it % self.policy_freq == 0:

            # Compute actor loss
            mean, std = self.actor(state)
            actor_dist = torch.distributions.Normal(mean, std)
            selected_action = actor_dist.rsample().clamp(
                -self.max_action, self.max_action
            )

            actor_loss = -(
                self.critic.Q1(state, selected_action)
                + (self.temperature * actor_dist.entropy())
            ).mean()

            # Optimize the actor
            self.actor_optimizer.zero_grad()
            actor_loss.backward()
            self.actor_optimizer.step()

            # Update the frozen target models
            for param, target_param in zip(
                self.critic.parameters(), self.critic_target.parameters()
            ):
                new_target_params = (
                    self.tau * param.data + (1 - self.tau) * target_param.data
                )
```

```
                target_param.data.copy_(new_target_params)

            for param, target_param in zip(
                self.actor.parameters(), self.actor_target.parameters()
            ):
                new_target_params = (
                    self.tau * param.data + (1 - self.tau) * target_param.data
                )
                target_param.data.copy_(new_target_params)

# In[ ]:

evaluation_td3 = main(policy_name="TD3")
```

Listing 3: Code for RDE with DQN in the `Cart Pole` Setting

```
#!/usr/bin/env python
# coding: utf-8

# In[ ]:

get_ipython().system("apt-get install x11-utils > /dev/null 2>&1")
get_ipython().system("pip install pyglet > /dev/null 2>&1")
get_ipython().system("apt-get install -y xvfb python-opengl > /dev/null 2>&1")

get_ipython().system("pip install gym pyvirtualdisplay > /dev/null 2>&1")

# In[ ]:

import os
import pdb
import sys
import copy
import json
import argparse
from datetime import datetime

import gym
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
import matplotlib.pyplot as plt
from IPython import display as ipythondisplay

class DQN(nn.Module):

    def __init__(self, input, hidden, output):
        super(DQN, self).__init__()
        self.fc1 = nn.Linear(input, hidden)
        self.fc2 = nn.Linear(hidden, hidden)
        self.fc3 = nn.Linear(hidden, output)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        return self.fc3(x)

    def reset(self):
        for layer in self.children():
            if hasattr(layer, "reset_parameters"):
                layer.reset_parameters()

class QNetwork:

    def __init__(self, args, input, output, learning_rate):
        self.weights_path = "models/%s/%s" % (
            args["env"],
            datetime.now().strftime("%Y-%m-%d_%H-%M-%S"),
        )

        # Network architecture.
        self.hidden = 128
        self.model = DQN(input, self.hidden, output)

        # Loss and optimizer.
        self.optim = torch.optim.Adam(self.model.parameters(), lr=learning_rate)
```

```python
        if args["model_file"] is not None:
            print("Loading pretrained model from", args["model_file"])
            self.load_model_weights(args["model_file"])

    def save_model_weights(self, step, i):
        # Helper function to save your model / weights.
        if not os.path.exists(self.weights_path):
            os.makedirs(self.weights_path)
        torch.save(
            self.model.state_dict(),
            os.path.join(self.weights_path, f"model_{step}_{i}.h5"),
        )

    def load_model_weights(self, weight_file):
        # Helper function to load model weights.
        self.model.load_state_dict(torch.load(weight_file))

# In[ ]:

class Replay_Memory:
    def __init__(self, state_dim, action_dim, memory_size=50000, burn_in=10000):
        # The memory essentially stores transitions recorder from the agent
        # taking actions in the environment.

        # Burn in episodes define the number of episodes that are written into the memory from the
        # randomly initialized agent. Memory size is the maximum size after which old elements in the memory are replaced.
        # A simple (if not the most efficient) way to implement the memory is as a list of transitions.
        self.memory_size = memory_size
        self.burn_in = burn_in
        self.states = torch.zeros((self.memory_size, state_dim))
        self.next_states = torch.zeros((self.memory_size, state_dim))
        self.actions = torch.zeros((self.memory_size, 1))
        self.rewards = torch.zeros((self.memory_size, 1))
        self.dones = torch.zeros((self.memory_size, 1))
        self.ptr = 0
        self.burned_in = False
        self.not_full_yet = True

    def append(self, states, actions, rewards, next_states, dones):
        self.states[self.ptr] = states
        self.actions[self.ptr, 0] = actions
        self.rewards[self.ptr, 0] = rewards
        self.next_states[self.ptr] = next_states
        self.dones[self.ptr, 0] = dones
        self.ptr += 1

        if self.ptr > self.burn_in:
            self.burned_in = True

        if self.ptr >= self.memory_size:
            self.ptr = 0
            self.not_full_yet = False

    def sample_batch(self, batch_size=32):
        # This function returns a batch of randomly sampled transitions - i.e. state, action, reward, next state, terminal flag tuples.
        # You will feed this to your model to train.
        if self.not_full_yet:
            idxs = torch.from_numpy(np.random.choice(self.ptr, batch_size, False))
        else:
            idxs = torch.from_numpy(
                np.random.choice(self.memory_size, batch_size, False)
            )

        states = self.states[idxs]
        next_states = self.next_states[idxs]
        actions = self.actions[idxs]
        rewards = self.rewards[idxs]
        dones = self.dones[idxs]
        return states, actions, rewards, next_states, dones

# In[ ]:

class DQN_Agent:
    def __init__(self, args, env):
        # Create an instance of the network itself, as well as the memory.
```

```
        # Here is also a good place to set environmental parameters,
        # as well as training parameters - number of episodes / iterations, etc.

        # Inputs
        self.args = args
        self.env = env
        self.environment_name = args["env"]
        self.render = self.args["render"]
        self.epsilon = args["epsilon"]
        self.network_update_freq = args["network_update_freq"]
        self.log_freq = args["log_freq"]
        self.test_freq = args["test_freq"]
        self.save_freq = args["save_freq"]
        self.learning_rate = args["learning_rate"]

        # Other Classes
        self.q_network = QNetwork(
            args,
            self.env.observation_space.shape[0],
            self.env.action_space.n,
            self.learning_rate,
        )
        self.target_q_network = QNetwork(
            args,
            self.env.observation_space.shape[0],
            self.env.action_space.n,
            self.learning_rate,
        )
        self.batch = list(range(32))

        # Save hyperparameters
        self.logdir = "logs/%s/%s" % (
            self.environment_name,
            datetime.now().strftime("%Y-%m-%d_%H-%M-%S"),
        )
        if not os.path.exists(self.logdir):
            os.makedirs(self.logdir)
        with open(self.logdir + "/hyperparameters.json", "w") as outfile:
            json.dump((self.args), outfile, indent=4)

    def epsilon_greedy_policy(self, q_values, epsilon):
        # Creating epsilon greedy probabilities to sample from.

        # choose random action a fraction epsilon of the time
        # and a greedy action the rest of the time
        sample = np.random.rand()
        if sample < epsilon:
            return self.env.action_space.sample()
        else:
            return torch.argmax(q_values).item()

    def greedy_policy(self, q_values):
        return torch.argmax(q_values).item()

    def td_estimate(self, state, action):
        # pass through q_network to get Q values
        Q_values = self.q_network.model.forward(state)
        action = action.long()
        return Q_values.gather(1, action)

    def td_target(self, reward, next_state, done, discount_factor):
        # pass through target_q_network and take maximum over Q values
        Q_values = self.target_q_network.model.forward(next_state)
        max_Q_values, _ = torch.max(Q_values, dim=1, keepdim=True)

        # compute td_target
        return reward + discount_factor * (1 - done) * max_Q_values

    def train_dqn(self, memory, discount_factor):
        # Sample from the replay buffer.
        state, action, rewards, next_state, done = memory.sample_batch(batch_size=32)

        # Optimization step.
        # For reference, we used F.smooth_l1_loss as our loss function.
        self.q_network.optim.zero_grad()
        loss = F.smooth_l1_loss(
            self.td_estimate(state, action),
```

```
            self.td_target(rewards, next_state, done, discount_factor),
        )
        loss.backward()
        self.q_network.optim.step()

        return loss

    def hard_update(self):
        self.target_q_network.model.load_state_dict(self.q_network.model.state_dict())

    @classmethod
    def plots(cls, reward, td_error):
        """
        Plots:
        1) Avg Cummulative Test Reward over 20 Plots
        2) TD Error
        #"""
        reward, time = zip(*rewards)
        plt.figure(figsize=(8, 3))
        plt.subplot(121)
        plt.title("Cummulative Reward")
        plt.plot(time, reward)
        plt.xlabel("iterations")
        plt.ylabel("rewards")
        plt.legend()
        plt.ylim([0, None])

        loss, time = zip(*td_error)
        plt.subplot(122)
        plt.title("Loss")
        plt.plot(time, loss)
        plt.xlabel("iterations")
        plt.ylabel("loss")
        plt.show()

    def epsilon_decay(self, initial_eps=1.0, final_eps=0.05):
        if self.epsilon > final_eps:
            factor = (initial_eps - final_eps) / 10000
            self.epsilon -= factor

# In[ ]:

def init_flags():

    flags = {
        "env": "CartPole-v0",  # Change to "MountainCar-v0" when needed.
        "render": False,
        "train": 1,
        "frameskip": 1,
        "network_update_freq": 10,
        "log_freq": 5,
        "test_freq": 20,
        "save_freq": 500,
        "learning_rate": 5e-4,
        "memory_size": 50000,
        "epsilon": 0.5,
        "model_file": None,
        "N": 2,  # number of agents,
        "RR": 1,  # replay ratio,
        "T": 8e4,  # time steps between agent resets ,
        "beta": 0.01,  # action selection coefficient
    }

    return flags

def get_action(theta, k, epsilon, beta, state, model):

    actions = []
    for agent in theta:
        q_value_i = agent.q_network.model.forward((state.reshape(1, -1)))
        action = agent.epsilon_greedy_policy(q_value_i, epsilon)
        actions.append(action)

    # now we choose the max q value over our ensemble of agents
    q_sa = torch.hstack(
        [
            theta[k].q_network.model.forward((state.reshape(1, -1)))[:, a]
```

```
            for a in actions
        ]
    )
    max_q_sa, _ = torch.max(q_sa, dim=0)
    alpha = beta / max_q_sa
    p_select = F.softmax(q_sa / alpha)

    action = np.random.choice(a=actions, p=p_select.numpy())

    return action, p_select

def test_(
    theta,
    env,
    beta,
    T,
    k,
    N,
    RR,
    memory,
    frameskip,
    discount_factor,
    t,
    model_file=None,
    episodes=100,
):
    # Evaluate the performance of your agent over 100 episodes, by calculating cumulative rewards for
     the 100 episodes.
    # Here you need to interact with the environment, irrespective of whether you are using a memory.
    cum_reward = []
    td_error = []
    for count in range(episodes):
        reward, error, _, _ = generate_episode(
            theta,
            mode="test",
            env=env,
            epsilon=0.05,
            beta=beta,
            T=T,
            k=k,
            N=N,
            RR=RR,
            memory=memory,
            frameskip=frameskip,
            discount_factor=discount_factor,
            t=t,
        )
        cum_reward.append(reward)
        td_error.append(error)
    cum_reward = torch.tensor(cum_reward)
    td_error = torch.tensor(td_error)
    print(
        "\nTest Rewards: {0} | TD Error: {1:.4f}\n".format(
            torch.mean(cum_reward), torch.mean(td_error)
        )
    )
    return torch.mean(cum_reward), torch.mean(td_error)

def burn_in_memory(
    theta,
    epsilon,
    beta,
    env,
    T,
    k,
    N,
    RR,
    memory,
    discount_factor,
    t,
    mode="train",
    frameskip=1,
):
    # Initialize your replay memory with a burn_in number of episodes / transitions.
    while not memory.burned_in:
        _, _, t, _ = generate_episode(
            theta,
```

```
            mode="burn_in",
            env=env,
            epsilon=epsilon,
            beta=beta,
            T=T,
            k=k,
            N=N,
            RR=RR,
            memory=memory,
            frameskip=frameskip,
            discount_factor=discount_factor,
            t=t,
        )
    print("Burn Complete!")

    return t

def generate_episode(
    theta,
    epsilon,
    beta,
    env,
    T,
    k,
    N,
    RR,
    memory,
    discount_factor,
    t,
    mode="train",
    frameskip=1,
):
    """
    Collects one rollout from the policy in an environment.
    """
    done = False
    state = torch.from_numpy(env.reset())
    rewards = 0

    td_error = []
    while not done:
        with torch.no_grad():
            action, p_select = get_action(theta, k, epsilon, beta, state, mode)
        i = 0
        while (i < frameskip) and not done:
            next_state, reward, done, info = env.step(action)
            next_state = torch.from_numpy(next_state)
            rewards += reward
            i += 1

        if mode == "train":
            if t % 1000 == 0:
                print(p_select)

        if mode in ["train", "burn_in"]:
            memory.append(state, action, reward, next_state, done)
        if not done:
            state = copy.deepcopy(next_state.detach())

        # Train the network.
        if mode == "train":
            t += 1

            for j in range(RR):
                for theta_i in theta:
                    theta_i.train_dqn(memory, discount_factor)

            if (t % (T / N)) == 0:
                print(k)
                theta[k].q_network.model.reset()
                theta[k].target_q_network.model.reset()
                k = (k + 1) % N

    if td_error == []:
        return rewards, [], t, k
    return rewards, torch.mean(torch.stack(td_error)), t, k
```

```python
def main(render=False):
    args = init_flags()
    args["render"] = render

    if args["env"] == "CartPole-v0":
        env = gym.make(args["env"], render_mode="rgb_array")
        discount_factor = 0.99
        num_episodes = 1000
        max_timesteps = 2e5
    else:
        raise Exception("Unknown Environment")

    memory = Replay_Memory(
        env.observation_space.shape[0],
        env.action_space.n,
        memory_size=args["memory_size"],
    )

    theta = [DQN_Agent(args, env) for _ in range(args["N"])]

    # time step
    t = 0
    # number of episodes
    step = 0
    # current agent being reset
    k = 0

    burn_in_memory(
        theta,
        epsilon=args["epsilon"],
        beta=args["beta"],
        env=env,
        T=args["T"],
        k=k,
        N=args["N"],
        RR=args["RR"],
        memory=memory,
        mode="train",
        frameskip=args["frameskip"],
        discount_factor=discount_factor,
        t=t,
    )

    rewards = []
    td_error = []

    while t < max_timesteps:
        # Generate Episodes using Epsilon Greedy Policy and train the Q network.
        step += 1
        _, _, t, k = generate_episode(
            theta,
            mode="train",
            env=env,
            epsilon=args["epsilon"],
            beta=args["beta"],
            T=args["T"],
            k=k,
            N=args["N"],
            RR=args["RR"],
            memory=memory,
            frameskip=args["frameskip"],
            discount_factor=discount_factor,
            t=t,
        )

        # Test the network.
        if step % args["test_freq"] == 0:
            print("here")
            test_reward, test_error = test_(
                theta,
                env=env,
                beta=args["beta"],
                N=args["N"],
                T=args["T"],
                k=k,
                RR=args["RR"],
                memory=memory,
```

```
                discount_factor=discount_factor,
                t=t,
                frameskip=args["frameskip"],
                model_file=None,
                episodes=20,
            )
            rewards.append([test_reward, step])
            td_error.append([test_error, step])

        # Update the target network.
        if step % args["network_update_freq"] == 0:
            for theta_i in theta:
                theta_i.hard_update()

        # Logging.
        if step % args["log_freq"] == 0:
            print("Step: {0:05d}/{1:05d}".format(step, num_episodes))

        # Save the model
        if step % args["save_freq"] == 0:
            for i, theta_i in enumerate(theta):
                theta_i.q_network.save_model_weights(step, i)

        # step += 1
        for theta_i in theta:
            theta_i.epsilon_decay()

    return rewards, td_error

# In[ ]:

rewards, td_error = main()
DQN_Agent.plots(rewards, td_error)
```