TOWARDS EVALUATING ADAPTIVITY OF MODEL-BASED REINFORCEMENT LEARNING

Yi Wan*
University of Alberta &
Mila - Quebec AI Institute
wan6
@ualberta.ca

Ali Rahimi-Kalahroudi*
Mila - Quebec AI Institute &
Université de Montréal
ali-rahimi.kalahroud
@mila.quebec

Janarthanan Rajendran
Mila - Quebec AI Institute &
Université de Montréal
janarthanan.rajendran
@mila.quebec

Ida Momennejad
Microsoft
idamo
@microsoft.com

Sarath Chandar
Mila - Quebec AI Institute,
École Polytechnique de Montréal, &
Canada CIFAR AI Chair
sarath.chandar
@mila.quebec

Harm van Seijen
Microsoft
Harm.vanSeijen
@microsoft.com

ABSTRACT

In recent years, a growing number of deep model-based reinforcement learning (RL) methods have been introduced. The interest in deep model-based RL is not surprising, given its many potential benefits, such as higher sample efficiency and the potential for fast adaption to changes in the environment. However, we demonstrate, using an improved version of the recently introduced Local Change Adaptation (LoCA) evaluation methodology , that well-known model-based methods such as PlaNet and DreamerV2 perform poorly in their ability to adapt to local environmental changes. Combined with prior work that made a similar observation about the other popular model-based method, MuZero, a trend appears to emerge, suggesting that current deep model-based methods have serious limitations. We dive deeper into the causes of this poor performance, by identifying elements that hurt adaptive behavior and linking these to underlying techniques frequently used in deep model-based RL. We empirically validate these insights in the case of linear function approximation by demonstrating that a modified version of linear Dyna achieves effective adaptation to local changes.

1 INTRODUCTION

While model-free reinforcement learning (RL) is still the dominant approach in deep RL, more and more research on deep model-based RL appears (Wang et al., 2019; Moerland et al., 2020). This is hardly surprising, as model-based RL (MBRL), which leverages estimates of the reward and transition model , could hold the key to some persistent challenges in deep RL, such as sample efficiency and effective adaptation to environment changes.

Although deep model-based RL has gained momentum, there are questions to be raised about the proper way to evaluate its progress. A common performance metric is sample-efficiency in a single task (Wang et al., 2019), which has several disadvantages. First, it conflates progress due to model-based RL with other factors such as representation learning. More importantly, it is unclear whether in an single-task setting, model-based RL is always more sample-efficient than model-free RL because learning a policy directly is not necessarily slower than learning a model and planning with the model. By contrast, when it comes to solving multiple tasks that share (most of) the dynamics, it is arguable that model-based RL has a clear advantage.

Instead of measuring the sample-efficiency of an algorithm in a single task, Van Seijen et al. (2020) developed the Local Change Adaptation (LoCA) task configuration and corresponding evaluation methodology to measure the agent's ability to adapt when the task changes. This approach, inspired

by approaches used in neuroscience for measuring model-based behavior in humans and rodents, is designed to measure how quickly an RL algorithm can adapt to a local change in the reward using its learned environment model. They used this to show that the deep model-based method MuZero Schrittwieser et al. (2019), which achieves great sample-efficiency on Atari, was not able to effectively adapt to a local change in the reward, even on simple tasks.

This paper builds out this direction further. First, we improve the original LoCA evaluation methodology, such that it is less sensitive to hyperparameters and can be more easily applied to stochastic environments. Our evaluation methodology is designed to make a binary classification of model-based methods: those that can effectively adapt to local changes in the environment and those that cannot.

We apply our improved methodology to the MuJoCo Reacher domain and use it to evaluate two continuous-control model-based methods, PlaNet and DreamerV2 Hafner et al. (2019b;a; 2020). Both methods turn out to adapt poorly to local changes in the environment. Combining these results with the results from Van Seijen et al. (2020), which showed a similar shortcoming of MuZero, a trend appears to emerge, suggesting that modern deep model-based methods are unable to adapt effectively to local changes in the environment.

We take a closer look at what separates model-based methods that adapt poorly from model-based methods that adapt effectively, by evaluating various tabular model-based methods. This leads us to define two failure modes that prohibit adaptivity. The two failure modes are linked to MuZero, potentially justifying its poor adaptivity to local changes. Further analysis of the Planet and the DreamerV2 methods enables us to identify two more failure modes that are unique to approximate (i.e., non-tabular) model-based methods.

Using the insights about important failure modes, we set off to design adaptive model-based methods that rely on function approximation. We demonstrate that by making small modifications to the classical linear Dyna method. The resulting algorithm adapts effectively in a challenging setting (sparse reward and stochastic transitions).

2 Measuring Adaptivity of Model-Based Methods

In this section, we present the LoCA task configuration introduced by Van Seijen et al. (2020) and an improved version of the corresponding LoCA evaluation methodology. The LoCA task configuration consists of two different tasks based on the same environment and the LoCA evaluation methodology measures how effective a method is in adapting from the first to the second task. This methodology is inspired by how model-based behavior is identified in behavioral neuroscience (e.g., see Daw et al. (2011)). The LoCA task configuration only specifies some specific features that the environment should have. In practise, many different domains can be used as the basis for a LoCA environment, ranging from tabular environments with discrete actions to environments with high-dimensional, continuous state and action spaces.

2.1 LoCA task configuration and training procedure

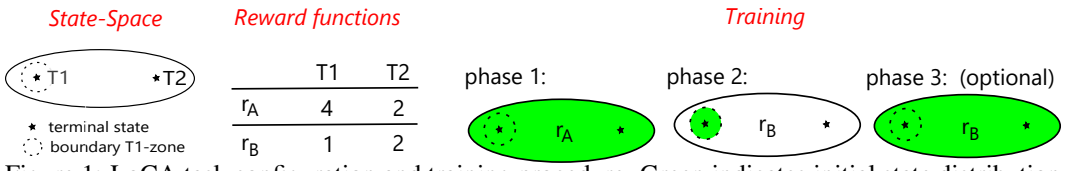


Figure 1: LoCA task configuration and training procedure. Green indicates initial state distribution during training.

The LoCA configuration considers two different tasks (i.e., reward functions) in the same environment. In addition, two different initial state distributions are used during training (see Figure 1).

A LoCA environment contains two terminal states, T1 and T2. Around T1 is a local area that, once entered, the agent is unable to move out of without terminating the episode, regardless of its policy. We refer to this local area as the *T1-zone*. The boundary of the T1-zone can be viewed as a one-way passage. The reward function for task A, r_A, and task B, r_B, are 0 everywhere, except upon transitions to a terminal state. A transition to T1 results in a reward of 4 under r_A and 1 under r_B;

transitioning to T2 results in a reward of 2 under both r_A and r_B. The discount factor $0 < \gamma < 1$ is the same for task A and task B. Note that, while r_A and r_B only differ locally, the optimal policy changes for almost all states: for task A, the optimal policy points towards T1 for the majority of the state-space, while for task B it points towards T2 (except for states within the T1-zone).

An experiment consists of three different training phases (with our improved evaluation methodology, the third phase is optional). During Phase 1, the reward function is r_A; upon transitioning to Phase 2, the reward function changes to r_B and remains unchanged upon transitioning to Phase 3. Crucially, the initial-state distribution during training changes across the phases as well. In Phases 1 and 3, the initial state is drawn uniformly at random from the full state space[1]; in Phase 2, it is drawn from the T1-zone.

The key question that determines adaptivity is whether or not a method can adapt effectively to the new reward function during phase 2. That is, can it change its policy from pointing towards T1 to pointing towards T2 across the state-space, while only observing samples from the local area around T1.

2.2 Original LoCA evaluation methodology

The LoCA evaluation methodology introduced by Van Seijen et al. (2020) only evaluates a method during Phase 3. Good adaptivity in Phase 2 implies optimal performance in Phase 3 right out of the gate; by contrast, a method that adapts poorly in Phase 2 will require additional training time in Phase 3 to obtain optimal performance. The amount of additional training needed is used to quantify how far off the behavior of a method is from ideal adaptive behavior. A method is evaluated by freezing learning periodically during training (i.e., not updating its weights, internal models or replay buffer), and executing its policy for a number of episodes. During evaluation, an initial-state distribution is used that covers a small area of the state-space roughly in the middle of T1 and T2. The fraction of episodes the agent ends up in the high-reward terminal (T2 for Phase 3) within a certain time-limit is used as measure for how good its policy is. This measure is called the *top-terminal fraction* and its values are always between 0 (poor performance) and 1 (good performance). The regret of this metric during Phase 3 with regards to optimal performance is called the *LoCA regret*. A LoCA regret of 0 implies the agent has a top-terminal fraction of 1 out of the gate in Phase 3 and means the agent adapts effectively to local changes.

The original evaluation procedure has a number of disadvantages. It involves various hyperparameters, such as the exact placement of the initial-state distribution for evaluation and the cut-off time for reaching T2 that can affect the value of the LoCA metric considerably. Furthermore, in stochastic environments, even an optimal policy could end up at the wrong terminal by chance. Hence, in stochastic environments a LoCA regret of 0 is not guaranteed even for adaptive methods.

Finally, measuring the amount of training needed in Phase 3 to determine how far off a method is from 'ideal' behavior is questionable. Adaptivity in Phase 2 is fundamentally different from adaptivity in Phase 3. Adaptivity in Phase 2 requires a method to propagate newly observed reward information to parts of the state-space not recently visited such that the global policy changes to the optimal one under the new reward function, an ability classically associated with model-based learning. By contrast, adaptivity in Phase 3, where the agent observes samples throughout the full state-space, is a standard feature of most RL algorithms, regardless whether they are model-based or not. And while leveraging a learned environment model can reduce the amount of (re)training required, so do many other techniques not unique to model-based learning, such as representation learning or techniques to improve exploration. So beyond determining whether a method is adaptive (LoCA regret of 0) or not (LoCA regret larger than 0), the LoCA regret does not tell much about model-based learning.

2.3 Improved LoCA evaluation methodology

To address these shortcomings, we introduce an improved evaluation methodology. In this improved version we evaluate performance during both Phases 1 and 2 by simply measuring the average return over the evaluation episodes and comparing it with the average return of the corresponding optimal policy. Furthermore, as initial-state distribution, the full state-space is used instead of some area

[1]Other initial-state distributions are possible too, as long as the distribution enables experiences from across the full state-space.

in between T1 and T2. Under this new methodology, we call a method (locally) adaptive if its performance is close-to-optimal in both Phases 1 and 2; if it is close-to-optimal in Phase 1, but not in Phase 2, we call the method non-adaptive. Keep in mind that both phases have a different optimal expected return due to the different reward functions. Finally, if the performance in phase 1 is not close-to-optimal, we do not make an assessment of its adaptivity.

With our improved evaluation methodology, Phase 3 is no longer required to evaluate the adaptivity of a method. However, it can be useful to rule out two—potentially easily fixable—causes for poor adaptivity in Phase 2. First, a method might simply not be able to get close-to-optimal performance in task B regardless of the samples it observes. Second, some methods are designed with the assumption of a stationary environment and cannot adapt to *any* changes in the reward function. This could happen, for example, if a method decays exploration and/or learning rates such that learning becomes less effective over time. In both cases, tuning learning hyperparameters or making minor modifications to the method might help.

3 Adaptive Versus Non-Adaptive MBRL

The single-task sample efficiency of a method says little about its ability to adapt effectively to local changes in the environment. Moreover, seemingly small discrepancies among different MBRL methods can greatly affect their adaptivity. In Section 3.1, we will illustrate both these points by evaluating three different tabular MBRL methods in the GridWorldLoCA domain (Figure 2a), introduced by Van Seijen et al. (2020), using LoCA. In Section 3.2 we discuss in more detail why some of the tabular MBRL methods fail to adapt effectively. The code of all of the experiments presented in this paper is available at `https://github.com/chandar-lab/LoCA2`.

3.1 Tabular MBRL Experiment

Each of the three methods we introduce learns an estimate of the environment model (i.e., the transition and reward function). We consider two 1-step models and one 2-step model. The 1-step model consists of $\hat{p}(s'|s,a)$ and $\hat{r}(s,a)$ that estimate the 1-step transition and expected reward function, respectively. Upon observing sample (S_t, A_t, R_t, S_{t+1}), this model is updated according to:

$$\begin{aligned}\hat{r}(S_t, A_t) &\leftarrow \hat{r}(S_t, A_t) + \alpha\big(R_t - \hat{r}(S_t, A_t)\big)\,,\\ \hat{p}(\cdot|S_t, A_t) &\leftarrow \hat{p}(\cdot|S_t, A_t) + \alpha\big(\langle S_{t+1}\rangle - \hat{p}(\cdot|S_t, A_t)\big)\,,\end{aligned}$$

with α the (fixed) learning rate and $\langle S_{t+1}\rangle$ a one-hot encoding of state S_{t+1}. Both $\langle S_{t+1}\rangle$ and $p(\cdot|S_t, A_t)$ are vectors of length N, the total number of states. Planning consists of performing a single state-value update at each time step based on the model estimates.[2] For a 1-step model:

$$V(s) \leftarrow \max_a \Big(\hat{r}(s,a) + \gamma \sum_{s'} \hat{p}(s'|s,a) V(s')\Big)\,.$$

We evaluate two variations of this planning routine: *mb-1-r*, where the state that receives the update is selected at random from all possible states; and *mb-1-c*, where the state that receives the update is the current state. Action selection occurs in an ϵ-greedy way, where the action-values of the current state are computed by doing a lookahead step using the learned model and bootstrapping from the state-values.

We also evaluate *mb-2-r*, which is similar to *mb-1-r* except that it uses a 2-step model, which estimates—under the agent's behavior policy—a distribution of the state 2 time steps in the future and the expected discounted sum of rewards over the next 2 time steps. The update equations for this method, as well as further experimental details, can be found in Section B.1.

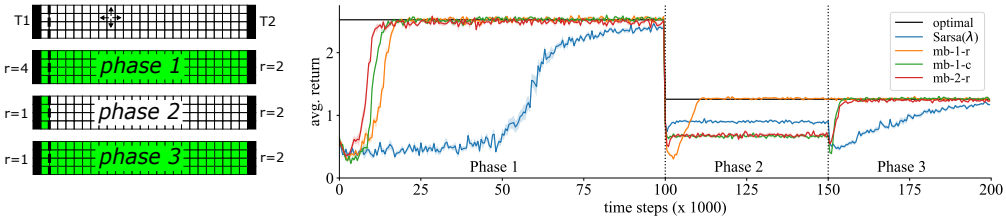
A summary of the methods we evaluate is shown in Table 1. Besides these methods, we also test the performance of the model-free method Sarsa(λ) with $\lambda = 0.95$.

We use a stochastic version of the GridWorldLoCA domain, where the action taken results with 25% probability in a move in a random direction instead of the preferred direction. The initial-state

[2]This is a special case of asynchronous value iteration, as discussed for example in Section 4.5 of Sutton & Barto (2018).

Table 1: Tabular model-based methods being evaluated.

Method	Model	State receiving value update
mb-1-r	1-step	Randomly selected
mb-1-c	1-step	Current state
mb-2-r	2-step	Randomly selected



(a) (b)

Figure 2: a) GridWorldLoCA domain (top) and the rewards and initial-state training distributions (in green) for the different phases. b) Comparison of the 3 model-based methods from Table 1 on the GridworldLoCA domain. While all methods converge to the optimal policy in Phases 1 and 3, only *mb-1-r* converges in Phase 2. We call model-based methods that are able to converge to optimal performance in Phase 2 (locally) adaptive model-based methods.

distribution for training in Phases 1 and 3, as well as the initial-state distribution for evaluation is equal to the uniform random distribution across the full state-space; the initial-state distribution for training in phase 2 is equal to the uniform random distribution across the T1-zone.

Figure 2 shows the performance of the various methods, averaged over 10 independent runs. While all three MBRL methods have similar performance in Phase 1 (i.e., similar single-task sample-efficiency), their performance in Phase 2 (i.e., their adaptivity to local changes in the environment) is very different. Specifically, even though the methods are very similar, only *mb-1-r* is able to change its policy to the optimal policy of task B during Phase 2. By contrast, *mb-1-c* and *mb-2-r* lack the flexibility to adapt effectively to the new task. Note that Sarsa(λ) achieves a higher average return in phase 2 than *mb-1-c* and *mb-2-r*. This may seem to be odd, given that it is a model-free method. There is however a simple explanation: the policy of Sarsa(λ) in phase 2 still points to T1, which now results in a reward of 1 instead of 4. By contrast, the policy changes for *mb-1-c* and *mb-2-r* are such that the agent neither moves to T1 nor T2 directly, instead, it mostly moves back and forth during the evaluation period, resulting in a reward of 0 most of time.

3.2 Discussion

The tabular MBRL experiment illustrates two different reasons why a model-based method may fail to adapt effectively.

Failure Mode #1: *Planning relies on a value function, which only gets updated for the current state.*

This applies to *mb-1-c*. A local change in the environment or reward function can results in a different value function across the entire state-space. Since *mb-1-c* only updates the value of the current state, during phase 2 only states within the T1-zone are updated. And because evaluation uses an initial-state distribution that uses the full state-space, the average return will be low.

Failure Mode #2: *The prediction targets of the learned environment model are implicitly conditioned on the policy used during training.*

This is the case for *mb-2-r*, as the model predicts the state and reward 2 time steps in the future, using only the current state and action as inputs. Hence, there is an implicit dependency on the behavior policy. For *mb-2-r*, the behavior policy is the ϵ-greedy policy with regards to the state-values, and during the course of training in Phase 1, this policy converges to the (epsilon-)optimal policy under reward function r_A. As a consequence, the environment model being learned will converge to a version that is implicitly conditioned on an optimal policy under r_A as well. During training in Phase

2, this dependency remains for states outside of the T1-zone (which are not visited during Phase 2), resulting in poor performance.

These two failure modes, while illustrated using tabular representations, apply to linear and deep representations as well, as the underlying causes are independent of the representation used. In fact, we believe that these two failure modes are in part responsible for the poor adaptivity of MuZero, as shown in Van Seijen et al. (2020). MuZero relies on a value function that gets updated towards n-step returns, which are implicitly conditioned on the policy used during training. This has a similar effect as when using multi-step update targets for learning the model, which underlies Failure Mode #2: the value function that results from the planning updates is not able to fully adapt to the new reward function. In addition, the n-step returns bootstrap from target values that are computed only for the visited states from an episode-trajectory, which is similar to Failure Mode #1, as we explain in more detail in Appendix B.2.

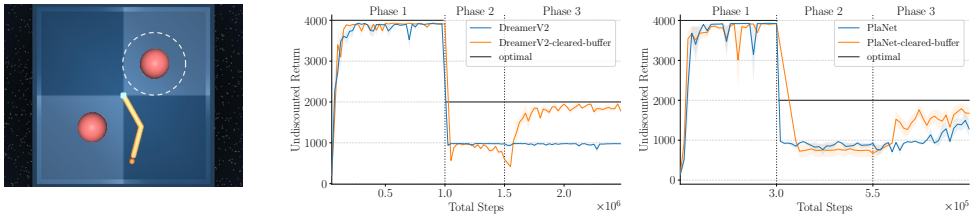
4 Evaluating PlaNet and DreamerV2

In this section, we evaluate the adaptivity of two deep model-based methods, PlaNet Hafner et al. (2019b) and the latest version of Dreamer, called DreamerV2 Hafner et al. (2020), using a modified version of LoCA in a variant of the Reacher domain, which involves a continuous-action domain with $64 \times 64 \times 3$ dimensional (pixel-level) states.

4.1 The ReacherLoCA Domain

We introduce a variation on the Reacher environment (the easy version) available from the DeepMind Control Suite Tassa et al. (2018). The Reacher environment involves controlling the angular velocity of two connected bars in a way such that the tip of the second bar is moved on top of a circular target. The reward is 1 at every time step that the tip is on top of the target, and 0 otherwise. An episode terminates after exactly 1000 time steps. The target location and the orientation of the bars are randomly initialized at the start of each episode.

In our modified domain, ReacherLoCA, we added a second target and fixed both target locations in opposite corners of the domain (Figure 3). Furthermore, we created a one-way passage around one of the targets. Staying true to the original Reacher domain, episodes are terminated after exactly 1000 time steps. While this means the target locations are strictly speaking not terminal states, we can apply LoCA to this domain just the same, interpreting the target location with the one-way passage surrounding it as T1 and the other one as T2. Rewards r_A and r_B are applied accordingly. The advantage of staying as close as possible to the original Reacher environment (including episode termination after 1000 time steps) is that we can copy many of the hyperparameters used for PlaNet and DreamerV2 applied to the original Reacher environment and only minor fine-tuning is required.



(a) The ReacherLoCA Domain (b) DreamerV2 (c) PlaNet

Figure 3: a) Illustration of the ReacherLoCA domain. The dashed white circle shows the T1-zone. b) and c) Plots showing the learning curves of DreamerV2 and PlaNet. We show the maximum achievable return at each phase as a baseline. For the setting in which the replay buffer is cleared at the start of each phase, we reinitialized it with 50 random episodes (50000 steps).

4.2 Experiment

Note that both PlaNet and DreamerV2 learn from transitions sampled from a large experience replay buffer (they call it experience dataset) that contains all recently visited transitions. When evaluating these algorithms in the ReacherLoCA domain, such a strategy could hurt the agent's ability of

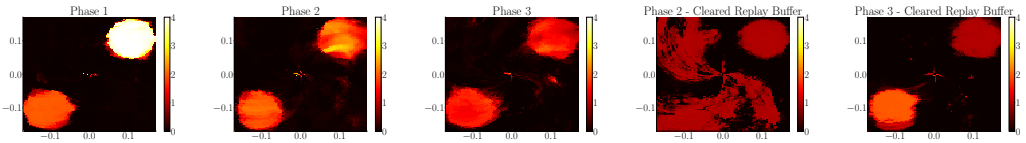


Figure 4: Visualization of the DreamerV2 agent's estimated reward model generated at the end of each phase. The x and y axes of each heatmap represent the agent's position in the ReacherLoCA domain.

adaptation, because in early stage of Phase 2, most of the data that the agent learns from are still from Phase 1 and it takes a long time before all of the task A data in the buffer get removed, if the replay buffer is large. Therefore a natural question to ask is, in practice, whether the agent can adapt well, with some stale data in the replay buffer. The other interesting question to ask would be, if the agent somehow knows the change of the task and reinitializes its replay buffer when it observes such a change, does the agent perform well in Phase 2?

To answer the above two questions, we tested two variants of each of these two algorithms. For both variants, we used a sufficiently large replay buffer that could contain all the transitions produced during training. For one variant, we reinitialized the replay buffer between two phases by first clearing the replay buffer and then filling replay buffer with certain number of episodes generated by following a random policy. This variant of algorithm requires to know when the environment changes and could therefore take advantage of this prior knowledge to remove outdated data. For the other variant, no modification to the replay buffer was applied between two phases. For both DreamerV2 and PlaNet, we did a grid search only over the critical hyperparameters suggested by the corresponding papers. Unless specified otherwise, other hyperparameters were chosen to be the same as those found to be the best by the corresponding papers in the original Reacher environment. Details of the experiment setup are summarized in Sections C.1 and C.4. Complete empirical results are presented in Sections C.3 and C.6.

For each variant of each of the two algorithms, we drew a learning curve (Figure 3) corresponding to the best hyperparameter setting found by a grid search (see Sections C.2 and C.5 for details). The reported learning curves suggest that neither DreamerV2 nor PlaNet effectively adapted their policy in Phase 2, regardless of the replay buffer being reinitialized or not. Further, when the replay buffer was not reinitialized, both DreamerV2 and PlaNet performed poorly in Phase 3. To obtain a better understanding of the failure of adaptation, we plotted in Figure 4 the reward predictions of DreamerV2's world model at the end of each phase. The corresponding results for PlaNet is similar and are presented in Section C.6. This time, our empirical results provide affirmative answers to the above two questions.

The first question is addressed by analyzing the predicted rewards without reinitializing the replay buffer. In this case, the predicted reward of T1 overestimates the actual reward (1) in both Phases 2 and 3. Such an overestimation shows that learning from stale data in the replay buffer apparently hurts model learning. In addition, the corresponding learning curves in Phases 2 and 3 show that the resulting inferior model could be detrimental for planning. The second question is addressed by analyzing the predicted rewards with reinitializing the replay buffer. In this case, the values for T1 at the end of Phase 2 are correct now, but the estimates for the rest of the state space are completely incorrect. This result answers our second question – with replay buffer reinitialization, the agent had no data outside of the T1-zone to rehearse in Phase 2 and forgot the learned reward model for states outside the T1-zone. Such an issue is called *catastrophic forgetting*, which was originally coined by McCloskey & Cohen (1989). Overall, we conclude that DreamerV2 achieved poor adaptivity to local changes due to two additional failure modes that are different from those outlined in Section 3.

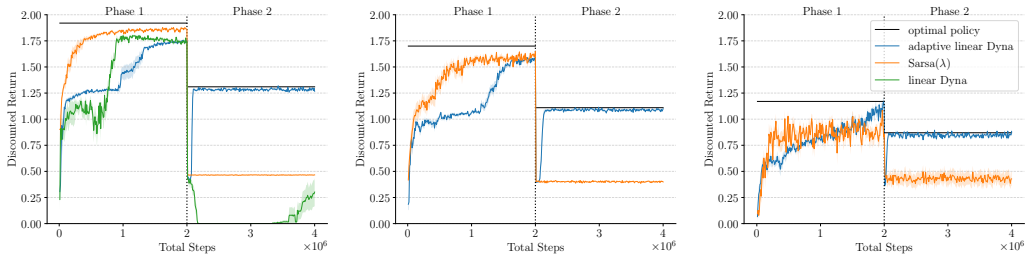
Failure Mode #3*: Learning from large replay buffers cause interference from the old task.*

Failure Mode #4*: If the environment model is represented by neural networks, learning from small replay buffers cause model predictions for areas of the state space not recently visited to be off due to catastrophic forgetting.*

Remark: Note that there is a dilemma between the above two failure modes. Also, note that as long as one uses a large replay buffer in a non-stationary environment, failure mode #3 is inevitable.

This suggests that, when using neural networks, solving the catastrophic forgetting problem is an indispensable need for LoCA and the more ambitious continual learning problem. Over the past 30 years, significant progress was made towards understanding and solving the catastrophic forgetting problem (French, 1991; Robins, 1995; French, 1999; Goodfellow et al., 2013; Kirkpatrick et al., 2017; Kemker et al., 2018). However, a satisfying solution to the problem is still not found and the problem itself is still actively studied currently.

5 Adaptive MBRL Algorithm with Function Approximation



(a) Stochasticity = 0.0 (b) Stochasticity = 0.3 (c) Stochasticity = 0.5

Figure 5: Plots showing that adaptive linear Dyna is adaptive while Sarsa(λ) and linear Dyna are not in the MountaincarLoCA domain. The x axis represents the number of training steps. Each point in a learning curve is the average discounted return obtained by following the agent's greedy policy for 10 runs. The first phase ends and the second phase starts at $2e6$ time step. We tested a broad range of hyperparameters for both algorithms. In each sub-figure, as a baseline, we plotted the best discounted return achieved by Sarsa(λ), after training for sufficiently long in Task A and B, with initial states being sampled from the entire state space (optimal policy).

In this section, we identify an algorithm that does not fall into the four aforementioned failure modes and understand its behavior using LoCA in a variant of the MountainCar domain (Moore, 1990). This algorithm, called *adaptive linear Dyna*, is a modified version of the linear Dyna algorithm (Algorithm 4 by Sutton et al. (2012)). To overcome the unsolved catastrophic forgetting problem with neural networks, this algorithm takes a step back by using linear function approximation with sparse feature vectors. We provide empirical evidence showing that this algorithm is adaptive in the MountainCarLoCA domain. Additional empirical results show that a nonlinear (neural networks) extension of the linear algorithm did not adapt equally well in the same setup, due to the inferior learned model resulting from catastrophic forgetting or interference from the old task.

Sutton et al.'s linear Dyna algorithm applies value iteration with a learned linear expectation model, which predicts the expected next state, to update a linear state-value function. Using expectation models in this way is sound because value iteration with an expectation model is equivalent to it with an aligned distribution model when using a linear state-value function (Wan et al., 2019). The algorithm does not fall into Failure Mode #2 because the expectation model is policy-independent. The algorithm does not fall into the Failure Modes #3 and #4 because the model is learned online and the algorithm uses linear function approximation with sparse tile-coded (Sutton & Barto, 2018) feature vectors. Limiting feature sharing alleviates the catastrophic forgetting problem because learning for one input influences predicting for only few other inputs.

Sutton et al.'s linear Dyna algorithm does fall into something similar to Failure Mode #1 because planning is not applied to any of the real feature vectors. Specifically, planning uses tabular feature vectors (one bit *on* in each vector), while real feature vectors (feature vectors corresponding to real states) are binary vectors with multiple bits on. It is thus unclear if planning with these unreal feature vectors could produce a good policy. In Figure 5, we empirically show that original linear Dyna leads to failure of adaptation in Phase 2.

A natural modification of the algorithm to improve its adaptivity, is to change the way of generating feature vectors for planning. We choose to randomly sample these feature vectors from a buffer containing feature vectors corresponding to recently visited states. We call this buffer the *planning* buffer because feature vectors stored in the buffer are used for planning. While the modification itself

is small and simple, the effect of the modification is significant – the modified algorithm can almost achieve the optimal value in Phase 2. The pseudo code of the algorithm is shown in Appendix E.

We tested our algorithm on a variant of the MountainCar domain. Our variant of the MountainCar domain extends the variant of the same domain used by Van Seijen et al. (2020), by introducing stochasticity. Specifically, with probability $1 - p$, the agent's action is followed and with probability p, the state evolves just as a random action was taken. We tested $p \in [0, 0.3, 0.5]$. The discount factor is 0.99. Other details of the experiment setup are summarized in Section D.1.

The first experiment was designed to test if the adaptive linear Dyna algorithm can adapt well with proper choices of hyperparameters. To this end, for each level of stochasticity, we did a grid search over the algorithm's hyperparameters and showed the learning curve corresponding to the best hyperparameter setting in Figure 5. The figure also shows, for different levels of stochasticity, learning curves of Sarsa(λ) with the best hyperparameter setting as baselines. The best hyperparameter setting is the one that performs the best in Phase 2, among those that perform well in Phase 1 (See Section D.2 for details). Further, we show a learning curve of linear Dyna by Sutton et al. (2012). A hyperparameter study of these algorithm is presented in Section D.3.

The learning curves show that for all different levels of stochasticity tested here, adaptive linear Dyna performed well in Phase 1 and adapted quickly and achieved near-optimal return in Phase 2. The learning curve for Sarsa(λ) coincides with our expectation. It performed well in Phase 1, but failed to adapt its policy in Phase 2. Linear Dyna also performed well in Phase 1, but failed to adapt in Phase 2, which is somewhat surprising because the unreal tabular feature vectors were used in both phases. A deeper look at our experiment data (Figure D.4 and D.5) shows that in Phase 1, although the policy is not apparently inferior, the estimated values are inferior. The estimated values are even worse in Phase 2. We hypothesize that such a discrepancy between Phases 1 and 2 is due to the fact that the model-free learning part of the linear Dyna algorithm helps obtaining a relatively accurate value estimation for states over the entire state space in Phase 1, but only for states inside the T1-zone in Phase 2.

The other observation is that when stochasticity = 0.5, Sarsa(λ) is worse than adaptive linear Dyna. Note that there is a very high variance in the learning curve of Sarsa(λ). On the contrary, adaptive linear Dyna achieved a much lower variance, potentially due to its planning with a learned model, which induces some bias and reduces variance. Comparing the two algorithms shows that when the domain is highly stochastic, the variance can be an important factor influencing the performance and planning with a learned model can reduce the variance.

6 Discussion and Conclusion

We introduced an improved version of the LoCA evaluation methodology, which is more flexible and easier to use. We then studied the adaptivity of two deep MBRL methods using this methodology. Our empirical results, combined with those from Van Seijen et al. (2020), suggest that several popular modern deep MBRL methods adapt poorly to local changes in the environment. This is surprising as the adaptivity should be one of the major strengths of MBRL (in behavioral neuroscience, adaptivity to local changes is one of the characteristic features that differentiates model-based from model-free behavior, e.g., see Daw et al. (2011)).

Besides that, we studied the challenges involved with building adaptive model-based methods and identified four important failure modes. These four failure modes were then linked to three modern MBRL algorithms, justifying why they didn't demonstrate adaptivity in experiments using LoCA. The first three of these failure modes can be overcome by using appropriate environment models and planning techniques. The fourth failure mode is tied to catastrophic forgetting, which is a challenging open problem. The most common mitigation technique to the problem, using a large experience replay buffer, is not an option in our case, as our experiments show that this precludes effective adaptation in non-stationary environments. Hence, we conclude that the path towards adaptive deep MBRL involves tackling the challenging catastrophic forgetting problem.

REFERENCES

Bacon, P.-L. *Temporal Representation Learning*. McGill University (Canada), 2018.

Barreto, A., Dabney, W., Munos, R., Hunt, J. J., Schaul, T., Van Hasselt, H., and Silver, D. Successor features for transfer in reinforcement learning. *arXiv preprint arXiv:1606.05312*, 2016.

Barto, A. G. and Mahadevan, S. Recent advances in hierarchical reinforcement learning. *Discrete event dynamic systems*, 13(1):41–77, 2003.

Boloka, T., Makondo, N., and Rosman, B. Knowledge transfer using model-based deep reinforcement learning. In *2021 Southern African Universities Power Engineering Conference/Robotics and Mechatronics/Pattern Recognition Association of South Africa (SAUPEC/RobMech/PRASA)*, pp. 1–6. IEEE, 2021.

Daw, N. D., Gershman, S. J., Seymour, B., Dayan, P., and Dolan, R. J. Model-based influences on humans' choices and striatal prediction errors. *Neuron*, 69(6):1204–1215, 2011.

Finn, C., Abbeel, P., and Levine, S. Model-agnostic meta-learning for fast adaptation of deep networks. In *International Conference on Machine Learning*, pp. 1126–1135. PMLR, 2017.

French, R. M. Using semi-distributed representations to overcome catastrophic forgetting in connectionist networks. In *Proceedings of the 13th annual cognitive science society conference*, volume 1, pp. 173–178, 1991.

French, R. M. Catastrophic forgetting in connectionist networks. *Trends in cognitive sciences*, 3(4): 128–135, 1999.

Goodfellow, I. J., Mirza, M., Xiao, D., Courville, A., and Bengio, Y. An empirical investigation of catastrophic forgetting in gradient-based neural networks. *arXiv preprint arXiv:1312.6211*, 2013.

Hafner, D., Lillicrap, T., Ba, J., and Norouzi, M. Dream to control: Learning behaviors by latent imagination. *arXiv preprint arXiv:1912.01603*, 2019a.

Hafner, D., Lillicrap, T., Fischer, I., Villegas, R., Ha, D., Lee, H., and Davidson, J. Learning latent dynamics for planning from pixels. In *International Conference on Machine Learning*, pp. 2555–2565. PMLR, 2019b.

Hafner, D., Lillicrap, T., Norouzi, M., and Ba, J. Mastering atari with discrete world models. *arXiv preprint arXiv:2010.02193*, 2020.

Huang, Y., Xie, K., Bharadhwaj, H., and Shkurti, F. Continual model-based reinforcement learning with hypernetworks. *arXiv preprint arXiv:2009.11997*, 2020.

Huisman, M., van Rijn, J. N., and Plaat, A. A survey of deep meta-learning. *Artificial Intelligence Review*, pp. 1–59, 2021.

Kemker, R., McClure, M., Abitino, A., Hayes, T., and Kanan, C. Measuring catastrophic forgetting in neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

Khetarpal, K., Riemer, M., Rish, I., and Precup, D. Towards continual reinforcement learning: A review and perspectives. *arXiv preprint arXiv:2012.13490*, 2020.

Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., Milan, K., Quan, J., Ramalho, T., Grabska-Barwinska, A., et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526, 2017.

Konidaris, G., Scheidwasser, I., and Barto, A. G. Transfer in reinforcement learning via shared features. 2012.

Lazaric, A. Transfer in reinforcement learning: a framework and a survey. In *Reinforcement Learning*, pp. 143–173. Springer, 2012.

Lu, K., Grover, A., Abbeel, P., and Mordatch, I. Reset-free lifelong learning with skill-space planning. *arXiv preprint arXiv:2012.03548*, 2020.

McCloskey, M. and Cohen, N. J. Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*, volume 24, pp. 109–165. Elsevier, 1989.

Moerland, T. M., Broekens, J., and Jonker, C. M. Model-based reinforcement learning: A survey. *arXiv preprint arXiv:2006.16712*, 2020.

Moore, A. W. Efficient memory-based learning for robot control. 1990.

Nguyen, T., Silander, T., and Leong, T. Transferring expectations in model-based reinforcement learning. *Advances in Neural Information Processing Systems*, 25, 2012.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32:8026–8037, 2019.

Robins, A. Catastrophic forgetting, rehearsal and pseudorehearsal. *Connection Science*, 7(2):123–146, 1995.

Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., et al. Mastering atari, go, chess and shogi by planning with a learned model. *arXiv preprint arXiv:1911.08265*, 2019.

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

Sutton, R. S., Precup, D., and Singh, S. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999.

Sutton, R. S., Szepesvári, C., Geramifard, A., and Bowling, M. P. Dyna-style planning with linear function approximation and prioritized sweeping. *arXiv preprint arXiv:1206.3285*, 2012.

Tassa, Y., Doron, Y., Muldal, A., Erez, T., Li, Y., Casas, D. d. L., Budden, D., Abdolmaleki, A., Merel, J., Lefrancq, A., et al. Deepmind control suite. *arXiv preprint arXiv:1801.00690*, 2018.

Taylor, M. E. and Stone, P. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(7), 2009.

Van Seijen, H., Nekoei, H., Racah, E., and Chandar, S. The loca regret: A consistent metric to evaluate model-based behavior in reinforcement learning. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M. F., and Lin, H. (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 6562–6572. Curran Associates, Inc., 2020. URL `https://proceedings.neurips.cc/paper/2020/file/48db71587df6c7c442e5b76cc723169a-Paper.pdf`.

Wan, Y., Abbas, Z., White, A., White, M., and Sutton, R. S. Planning with expectation models. *arXiv preprint arXiv:1904.01191*, 2019.

Wang, T., Bao, X., Clavera, I., Hoang, J., Wen, Y., Langlois, E., Zhang, S., Zhang, G., Abbeel, P., and Ba, J. Benchmarking model-based reinforcement learning. *arXiv preprint arXiv:1907.02057*, 2019.

Zhang, M., Vikram, S., Smith, L., Abbeel, P., Johnson, M., and Levine, S. Solar: Deep structured representations for model-based reinforcement learning. In *International Conference on Machine Learning*, pp. 7444–7453. PMLR, 2019.

Zhu, Z., Lin, K., and Zhou, J. Transfer learning in deep reinforcement learning: A survey. *arXiv preprint arXiv:2009.07888*, 2020.

A Related Work

LoCA is an evaluation methodology designed to measure the adaptivity of an algorithm. It serves as a preliminary yet important step towards the ambitious *continual learning* problem (Khetarpal et al., 2020). In the continual learning problem, the agent interacts with an *non-stationary* environment. LoCA specifies a particular kind of continual learning problem, which involves fully observable environment and only one environment change.

LoCA is also closely related to the well-known *transfer learning* problem (Taylor & Stone, 2009; Lazaric, 2012; Zhu et al., 2020) because they both need the agent to solve for multiple tasks. Nevertheless, LoCA should not be viewed as a special case of transfer learning. In transfer learning, the agent is informed of which task it is solving, while in LoCA it does not.

There are several ways in which an algorithm may potentially adapt quickly when the environment changes. For example, the learned feature representation (Konidaris et al., 2012; Barreto et al., 2016), the discovered options (Sutton et al., 1999; Barto & Mahadevan, 2003; Bacon, 2018), or some general meta-knowledge (Finn et al., 2017; Huisman et al., 2021) obtained from the old environment may be useful for learning and planning in the new environment. LoCA focuses on evaluating if the agent's fast adaptation relies on its planning with a learned model.

On the other hand, there are also some existing works developing MBRL algorithms for the continual learning problem or the transfer learning problem (Huang et al., 2020; Boloka et al., 2021; Zhang et al., 2019; Nguyen et al., 2012; Lu et al., 2020). Although these algorithms have shown great performance in various experiments, none of these algorithms directly resolve the fundamental catastrophic forgetting issue and therefore, generally speaking, they are not likely to demonstrate adaptivity in LoCA.

B Supplementary Details for Experiments in Section 3

B.1 Details of the Tabular Experiment

Initial distributions	Phase 1 training	Uniform distribution over the entire state space
	Phase 1 evaluation	Uniform distribution over the entire state space
	Phase 2 training	Uniform distribution over states within T1-zone
	Phase 2 evaluation	Uniform distribution over the entire state space
	Phase 3 training	Uniform distribution over the entire state space
	Phase 3 evaluation	Uniform distribution over the entire state space
Training steps	Phase 1 steps	$100,000$
	Phase 2 steps	$50,000$
	Phase 3 steps	$50,000$
Other details	Training steps between two evaluations	500
	Number of runs	10
	Number of evaluation episodes	100

Table B.1: Details of the tabular experiment.

For the GridWorldLoCA domain, we used the same discount factor as used in (Van Seijen et al., 2020), $\gamma = 0.97$, but used stochastic transitions. In particular, with 25% probability the action outcome is that of a random action instead of the selected action.

For all methods, we used an ϵ-greedy behavior policy with $\epsilon = 0.1$. In addition, we used optimistic initialization to encourage high exploration during initial training in Phase 1. Specifically, for Sarsa(λ) we set the initial action-values to 8 (twice the value of the maximum reward in Phase 1). And for all model-based methods, we set the initial state-values and initial reward-function estimates to 8. For Sarsa(λ), we used $\lambda = 0.95$. We roughly optimized the step-size for each method. This resulted in a step-size of 0.2 for all model-based method and a step-size of 0.03 for Sarsa(λ). The reason the best step-size for Sarsa(λ) is substantially lower than the one for the model-based methods is that a λ of 0.95 combined with the relatively high environment stochasticity leads to high variance updates for Sarsa(λ). Hence, to counter this variance and get accurate estimates a small step-size is needed.

B.2 MuZero and its relation to Failure Mode #1 and #2

We start by summarizing some key elements of MuZero's implementation. At its core, MuZero learns a 1-step model g that takes as input a hidden state h_t (derived from past observations) and action a_t, and outputs an estimate of the reward and next hidden state h_{t+1}. Simultaneously, it learns a value-function v that maps a hidden state s_t to an estimate of the expected return. At every time step, a MCTS routine is performed starting from the current hidden state, by leveraging the learned g model as the agent does not have access to the true environment model. MCTS is performed up to a certain maximum horizon k (k=5 for the Atari domain). At the leaf nodes of the tree, MuZero's MCTS bootstraps from v rather than performing a sample rollout, as normally occurs in MCTS. The output of MuZero's MCTS routine is a (local) policy $\pi_t(h_t)$, according to which action a_t is selected, and a target value v_t^{target}. This target value is an improved estimate of $v(h_t)$, as it is computed from the bootstrapped MCTS routine. The target value v_t^{target} is stored in a replay buffer, in addition to the current observation o_t, action a_t, reward r_{t+1} and policy $\pi_t(h_t)$. The value function v is updated by selecting a past trajectory, computing the hidden state h_t for some time step t within the trajectory and updating its value $v(h_t)$ using the n-step return $r_{t+1} + \gamma r_{t+2} + ...\gamma^n \, v_{t+n}^{target}$. Crucially, the n-step return bootstraps not from $v(h_{t+n})$, but from the target value of the corresponding time step, v_{t+n}^{target}. These target values are not updated once they are in the replay buffer. Hence, despite that MuZero employs experience replay to update the value function v, during Phase 2 the update targets outside the T1-zone remain the same and do not adapt to the newly observed reward. This can be viewed as a variation of Failure Mode #1. Finally, an n-step update target is used, whose value is implicitly conditioned on the behavior policy during training. This has a similar effect as when using multi-step update targets for learning the model—which underlies Failure Mode #2–as the planning update uses the model estimate as well as value function.

C Supplementary Details and Results for Experiments in Section 4

C.1 Setup of the DreamerV2 Experiment

Initial distributions	Phase 1 training	Uniform distribution over the entire state space
	Phase 1 evaluation	Uniform distribution over the entire states outside T1-zone
	Phase 2 training	Uniform distribution over states within T1-zone
	Phase 2 evaluation	Uniform distribution over the entire states outside T1-zone
	Phase 3 training	Uniform distribution over the entire state space
	Phase 3 evaluation	Uniform distribution over the entire states outside T1-zone
Training steps	Phase 1 steps	Planet: 3×10^5, DreamerV2: 10^6
	Phase 2 steps	Planet: 2.5×10^5, DreamerV2: 5×10^5
	Phase 3 steps	Planet: 2.5×10^5, DreamerV2: 10^6
Other details	Training steps between two evaluations	PlaNet: 10^4, DreamerV2 5×10^4
	Number of runs	5
	Number of evaluation episodes	5

Table C.1: Details of DreamerV2 and PlaNet experiments.

In the ReacherLoCA domain, if the agent is within the T1-zone and chooses an action that takes it out of the T1-zone, the agent's state remains unchanged. In general, T1-zone can be implemented without touching native domain code as long as the domain code offers an API function to set or reload the state. A wrapper around the domain can be implemented that resets the state to the previous one whenever the agent takes an action that moves it out of the T1-zone.

We tried two variants of the DreamerV2 algorithm. One reinitializes the replay buffer at the beginning of Phase 2 and the other one does not. When initializing (at the beginning of Phase 1) or reinitializing the replay buffer, the agent performs 50 episodes following a uniformly random policy and stores these episodes in the buffer (the same initialization strategy is also adopted in DreamerV2, although fewer number of episodes are produced there). We found that increasing the number of random episodes helps the agent to find the optimal target, not the sub-optimal one.

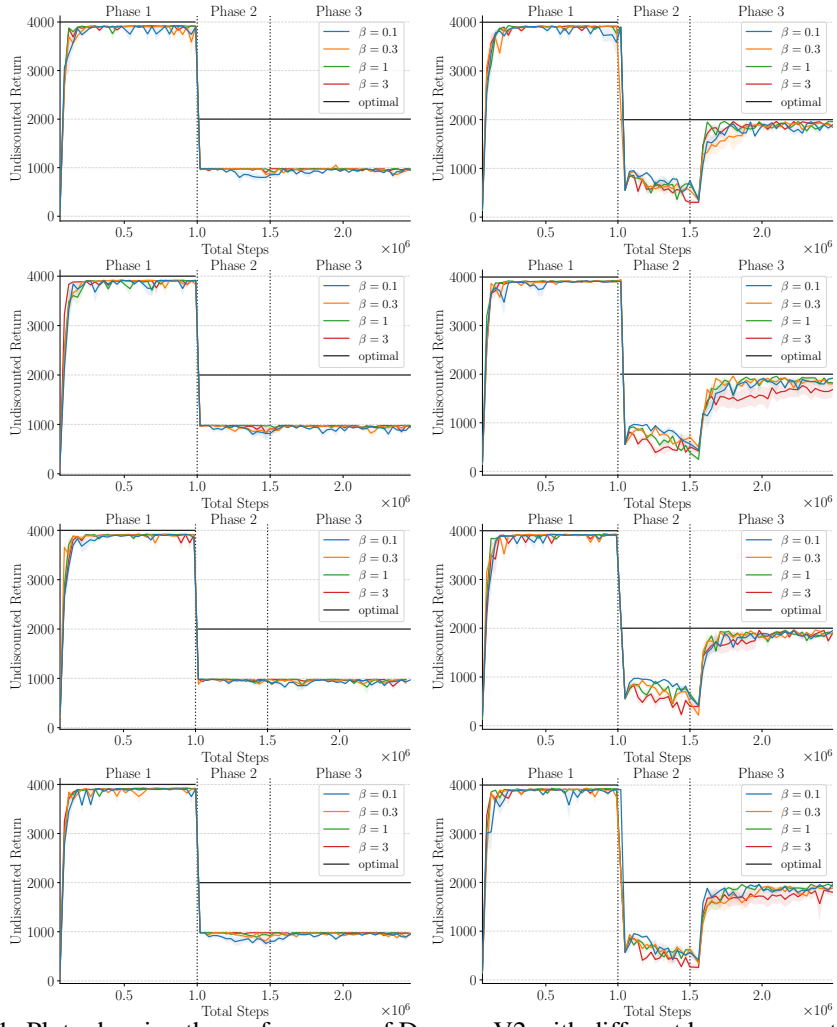


Figure C.1: Plots showing the performance of DreamerV2 with different hyperparameter settings. The value of each point in a curve is the average return over 5 evaluation episodes. We searched over the critical hyperparameters: 1) the KL loss scale β, 2) the actor entropy loss scale η, and 3) the discount factor γ. In this figure, $\gamma = 0.99$. $\eta = 10^{-3}, 3 \times 10^{-4}, 10^{-4}, 10^{-5}$ in four different rows respectively. The right column corresponds to the buffer reinitializing case, and the left column corresponds to the other case.

We searched over the KL loss scale $\beta \in \{0.1, 0.3, 1, 3\}$, the actor entropy loss scale $\eta \in \{10^{-5}, 10^{-4}, 3 \times 10^{-4}, 10^{-3}\}$, and the discount factor $\gamma \in \{0.99, 0.999\}$. Note that the discount factor tested here is not the problem's discount factor (the problem's discount factor is 1). Instead, it is part of the solution method.

C.2 Additional Information about the DreamerV2 Result Shown in the Middle Panel of Figure 3

The hyperparameters used to generate the plotted learning curves are chosen in the following way. We first collect the set of hyperparameter settings that, in both Phases 1 and 3, result in average returns (over the last five evaluation runs) being more than 80% of the optimal average returns. If the set of hyperparameter settings is empty, we chose the setting that achieved the highest average return in Phase 3 (over the last five evaluation runs). Otherwise, from the set of hyperparameter settings, we picked the setting that achieved the highest average return over all evaluation runs in Phase 2. Each point in a learning curve is the average of undiscounted cumulative reward over 5 evaluation episodes. The shading area shows the standard error.

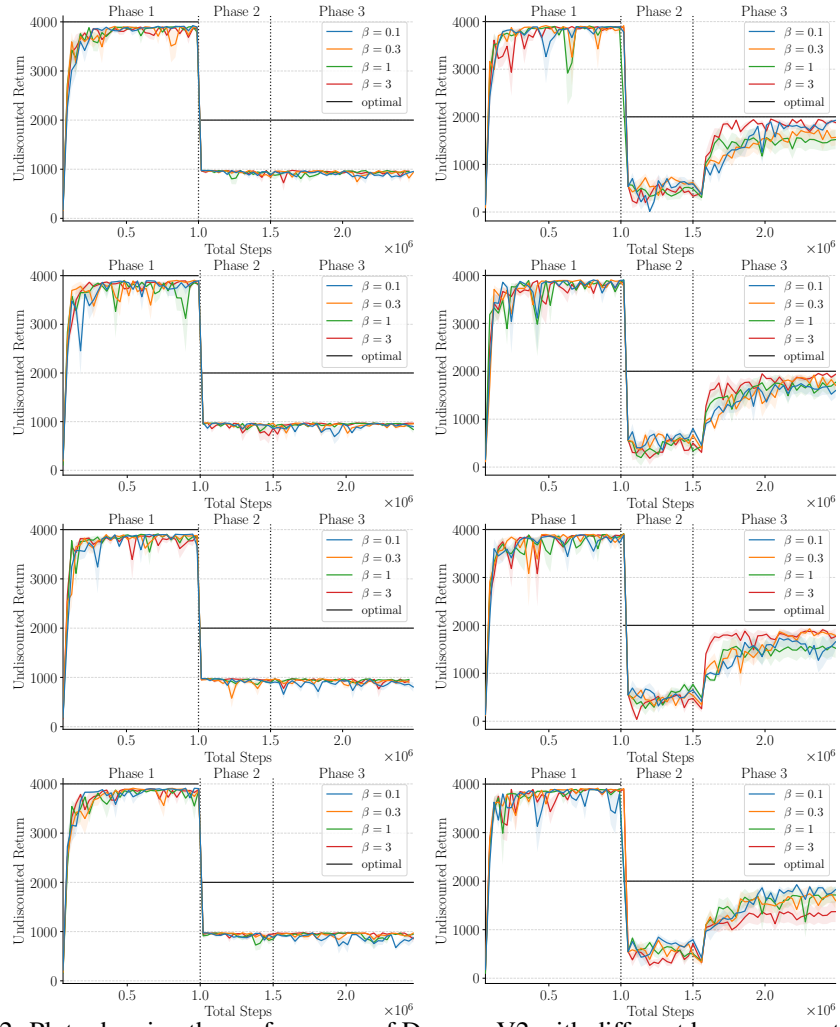


Figure C.2: Plots showing the performance of DreamerV2 with different hyperparameter settings. We searched over the critical hyperparameters: 1) the KL loss scale β, 2) the actor entropy loss scale η, and 3) the discount factor γ. In this figure $\gamma = 0.999$. $\eta = 10^{-3}, 3 \times 10^{-4}, 10^{-4}, 10^{-5}$ in four different rows respectively. The right column corresponds to the buffer reinitializing case, and the left column corresponds to the other case.

The best hyperparamerter setting for DreamerV2 with replay buffer reinitialization is $\beta = 0.1$, $\eta = 10^{-4}$, and $\gamma = 0.99$. Without replay buffer reinitialization, none of the runs achieved more than 80% of optimal average return in Phase 3 over the last five evaluation runs. The setting that achieved the highest performance in Phase 3 is $\beta = 1$, $\eta = 3 \times 10^{-4}$, and $\gamma = 0.99$.

Figure 4 is produced using the best hyperparameter setting with replay buffer reinitialization. The seed used is randomly picked. Furthermore, we have tested other random seeds and hyperparameter settings (not shown in the paper) and obtained similar results as those shown in Figure 4.

C.3 Additional DreamerV2 Results

Figure C.1 and C.2 show learning curves with different hyperparameter settings for DreamerV2.

Figure C.3 shows DreamerV2's (no reinitialization) performance with longer training in Phase 3. We observe that even with a much longer training in Phase 3, the agent did not escape from the sub-optimal target.

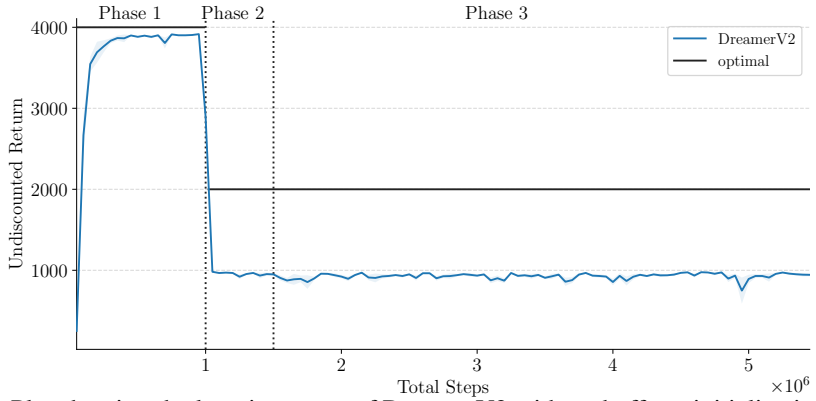


Figure C.3: Plot showing the learning curve of DreamerV2 without buffer reinitialization. We show the maximum achievable undiscounted reward at each phase as a baseline. To check if the agent's performance could improve through more time, we increased the total environments steps in Phase 3. The curve shows that the agent fails to escape from the sub-optimal target even with much longer training in Phase 3.

C.4 Setup of the PlaNet Experiment

PlaNet uses the cross-entropy method (CEM) for planning. We empirically found that some of the runs would get stuck in the local optima at the end of Phase 1 when using the suggested hyperparameters. Through more experiments, we found that increasing the CEM hyperparameters avoids poor performance in phase 1 for all the random seeds. We increased the horizon length H to 50, optimization iterations I to 25, and candidate samples J to 4000.

Just like the two tested DreamerV2 variants, the two tested PlaNet variants are one that reinitializes the replay buffer and one that does not. The reinitializing strategy is the same as what we apply to DreamerV2 shown in Section C.1.

We searched over the KL-divergence scale $\beta \in \{0, 1, 3, 10\}$, and the step-size $\in \{10^{-4}, 3 \times 10^{-4}, 10^{-3}\}$.

C.5 Additional Information about the PlaNet Result Shown in the Right Panel of Figure 3

The best hyperparameter setting for PlaNet with buffer reinitialization is $\beta = 1$, and the step-size $= 10^{-3}$. For PlaNet without buffer reinitialization, similar to DreamerV2, none of the runs achieved more than 80% of optimal average return in Phase 3. The setting that achieved the highest average return in Phase 3 is $\beta = 0$, and the step-size $= 10^{-3}$.

C.6 Additional PlaNet Results

Figure C.5 shows learning curves with different hyperparameter settings for PlaNet.

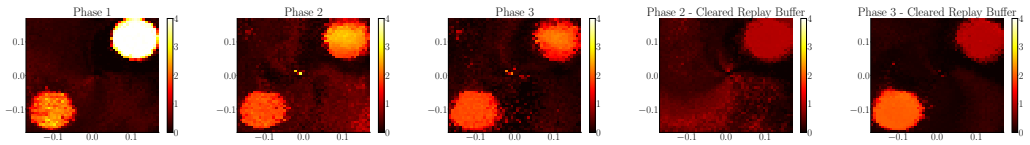


Figure C.4: Visualization of the PlaNet agent's estimated reward model generated at the end of each phase. The x and y axes of each heatmap represent the agent's position in the ReacherLoCA domain. These heatmaps show that in Phase 2 PlaNet incorrectly estimate the reward model regardless of the buffer being reinitialized or not.

We also plotted the reward estimations of PlaNet's world model at the end of each phase in Figure C.4. The results are similar to those of DreamerV2's (see Figure 4).

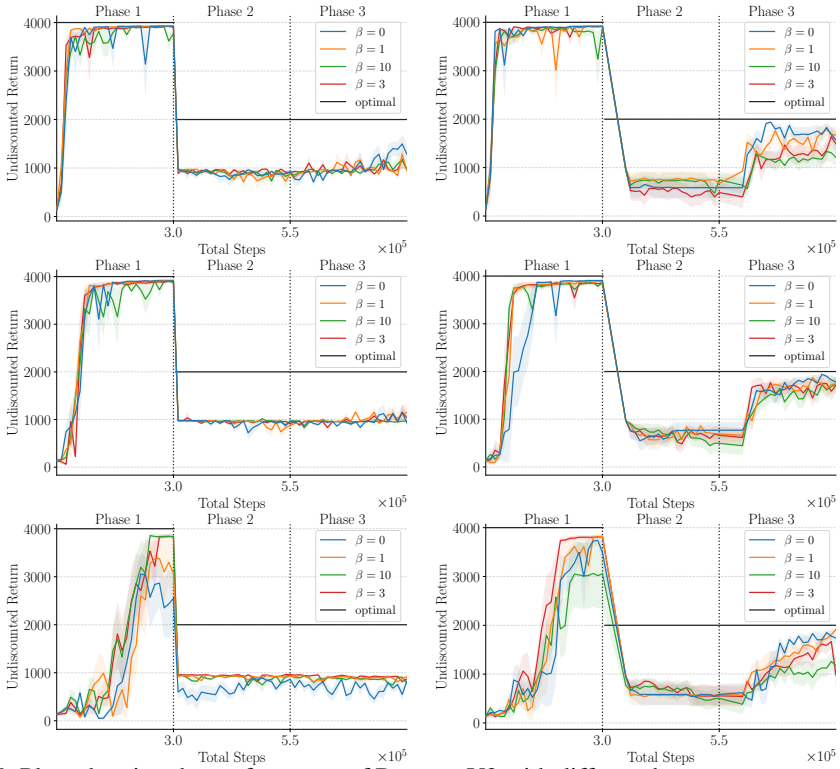


Figure C.5: Plots showing the performance of DreamerV2 with different hyperparameter settings. We searched over the critical hyperparameters: 1) the KL-divergence scales β, and 2) the step size. The step size is $10^{-3}, 3 \times 10^{-4}, 10^{-4}$ in three different rows respectively. The right column corresponds to the buffer reinitializing case, and the left column corresponds to the other case.

D Supplementary Details and Results for Experiments in Section 5

D.1 Setup of the Adaptive Linear Dyna Experiment

The MountainCarLoCA domain has two terminal states, one of which locates at the top of the mountain (position > 0.5, velocity > 0) and the other one locates at the valley $((\text{position} + 0.52)^2 + 100 * \text{velocity}^2) <= 0.07^2)$. The dynamics of the environment is described in Section **??**. The small region where the agent is initialized during evaluation is roughly at the middle of the two terminal states ($-0.2 \leq \text{position} \leq -0.1$, $-0.01 \leq \text{velocity} \leq 0.01$).

The linear Dyna algorithm we tested is basically Algorithm 4 by Sutton et al. (2012). However, we found that Algorithm 4 takes too much memory and computation, making it difficult for testing. To reduce the memory and computation usage, the tested linear Dyna algorithm slightly modifies Algorithm 4 in the following two ways. First, while Algorithm 4 does not specify the maximum size of the priority queue, our algorithm sets that maximum to be 400000 (maximum training steps) and removes 20% oldest elements in the queue whenever the queue is full. Second, in each planning step, Algorithm 4 updates for all predecessor features of the first element in the priority queue and add all predecessor features to the priority queue, our algorithm only updates 5 randomly chosen predecessor features and adds them to the queue. These two modifications we made significantly reduce memory usage and computation.

We summarize in Table D.1 tested hyperparameters for linear Dyna and adaptive linear Dyna.

For Sarsa(λ), the tile coding setups and the choices of stepsize α are the same as those used for adaptive linear Dyna. The exploration parameter was fixed to be 0.1. We also tested three λ values: 0, 0.5, and 0.9.

Number of Tilings and Tiling Size	[3, 18 by 18; 5, 14 by 14; 10, 10 by 10]
Exploration parameter ϵ	[0.1, 0.5, 1.0]
Stepsize α	[0.001, 0.005, 0.01, 0.05, 0.1] / number of tilings
Model Stepsize β	[0.001, 0.005, 0.01, 0.05, 0.1] / number of tilings
Number of planning steps	5
Size of the planning buffer n_p (adaptive linear Dyna)	[10000, 200000, 4000000]

Table D.1: Tested hyperparameters in the adaptive linear Dyna experiment.

Table D.2 shows the experiment setup used to test linear Dyna, adaptive linear Dyna (Algorithm 1), and Sarsa(λ).

Initial distributions	Phase 1 training	Uniform distribution over the entire state space
	Phase 1 evaluation	Uniform distribution over a small region
	Phase 2 training	Uniform distribution over states within T1-zone
	Phase 2 evaluation	Uniform distribution over a small region
Training steps	Phase 1 steps	2×10^6
	Phase 2 steps	2×10^6
Other details	Training steps between two evaluations	10^4
	Number of runs	10
	Number of evaluation episodes	10

Table D.2: Experiment setup used to test linear Dyna, adaptive linear Dyna, and Sarsa(λ).

D.2 Additional Information about the Result Shown in Figure 5

The hyperparameter setting chosen to draw curves in Figure 5 was chosen in the following way. We first collected hyperparameter settings that achieved 90% of the optimal average return over the last 5 evaluations in Phase 1. From these hyperparameter settings, we picked one that achieved the highest average return over the last 5 evaluations in Phase 2 and drew its learning curve. For stochasticity = 0.5, no hyperparameter setting for Sarsa(λ) achieved 90% optimal performance in the first phase and we picked the one that achieved more than 80% optimality in the first phase.

We summarize in Table D.3 the best hyperparameter settings under different level of stochasticities for each algorithm tested in Section 5.

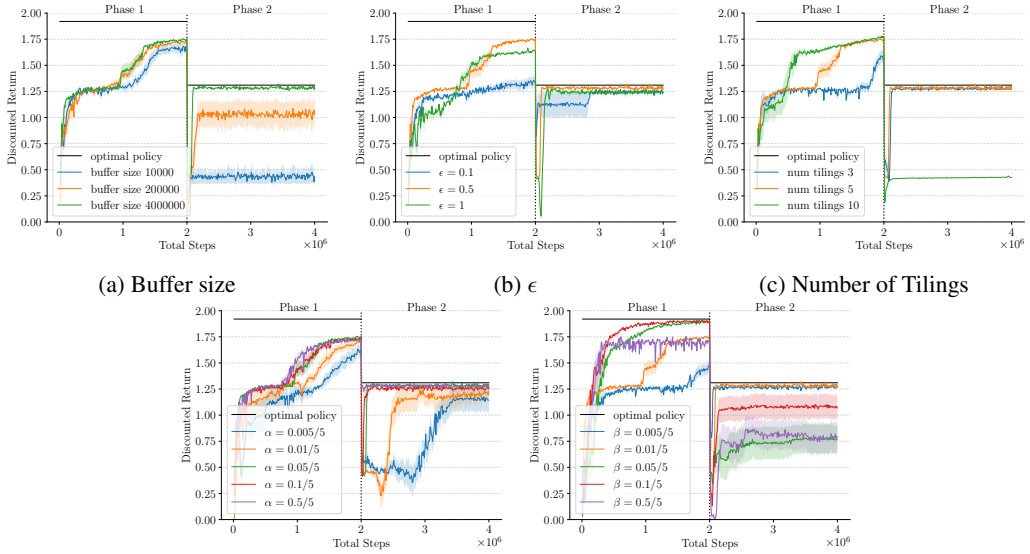
Algorithm	Hyperparameter	Deterministic	Stochasticity = 0.3	Stochasticity = 0.5
Adaptive linear Dyna	Exploration parameter (ϵ)	0.5	1.0	1.0
	Step-size (α)	0.05	0.05	0.1
	Step-size (β)	0.01	0.01	0.01
	Number of tilings	5	5	5
	Size of the planning buffer	4000000	4000000	4000000
Sarsa(λ)	Step-size (α)	0.1	0.05	0.05
	λ	0.5	0.5	0.9
	Number of tilings	10	10	10
Linear Dyna	Exploration parameter (ϵ)	0.5		
	Step-size (α)	0.5		
	Number of tilings	5		

Table D.3: Best hyperparameter settings in the adaptive linear Dyna experiment.

D.3 Additional Results of the Adaptive Linear Dyna Experiment

To demonstrate adaptive linear Dyna's sensitivity to a hyperparameter, we fixed all of the hyperparameters, except the one being tested, to be those in the best hyperparameter setting (Table D.3). We then varied the tested hyperparameter and plotted the corresponding learning curves. Learning curves for stochasticity = 0, 0.3, and 0.5 are shown in Figure D.1, Figure D.2, and Figure D.3 respectively.

We now use Figure D.1 as an example to understand the algorithm's sensitivity to its hyperparameters. Sub-figure (a) shows that the algorithm needs a sufficiently large replay buffer to achieve fast



(a) Buffer size (b) ϵ (c) Number of Tilings

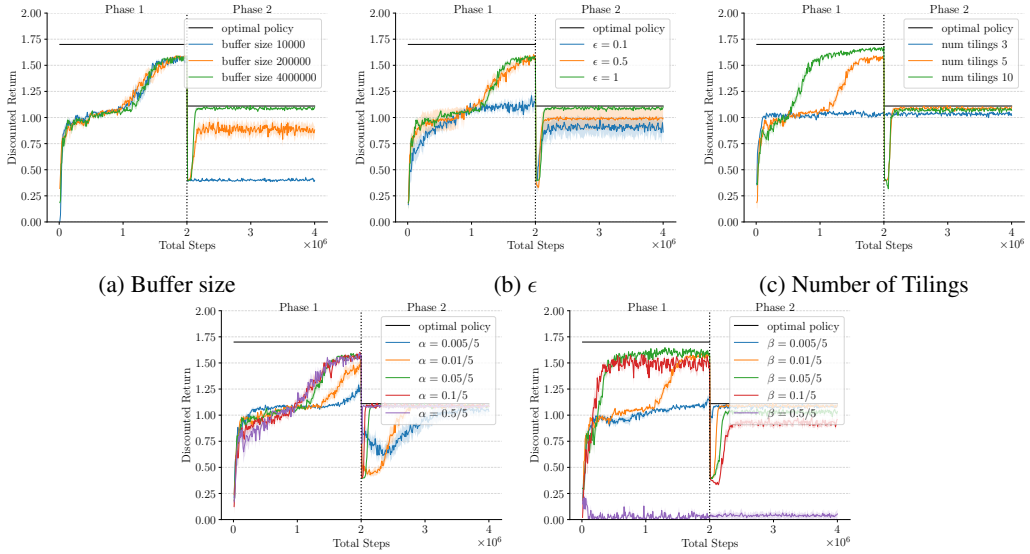
(d) Stepsize (e) Model Stepsize

Figure D.1: Plots showing the sensitivity of the adaptive linear Dyna algorithm to its hyperparameters in the deterministic MountainCarLoCA domain.

adaptation. This is not surprising because the collected data in Phase 2 would only cover a small region of the state space. The Phase 2 data quickly occupy the entire planning buffer, after which planning in Phase 2 would only update states in that small region. Sub-figure (b) shows that the algorithm worked well when a relatively large exploration parameter is used. $\epsilon = 0.1$ performed well in the second phase but it was not close to optimality in the first phase. A deeper look at the experimental data (not shown here) suggested that, when $\epsilon = 0.1$, for most of runs, the agent only moved to the sub-optimal terminal state in the first phase, presumably as a result of weak exploration. The sub-optimal terminal state in Phase 1 is the optimal terminal state in Phase 2 and thus the agent didn't have to change its policy to adapt to the task change. A first look at Sub-figure (c) seems to suggest that the algorithm only adapted well when the number of tilings k is small (3 or 5). By checking experimental results of more hyperparameter settings, we found that the failure when $k = 10$ is actually a result of other hyperparameters being chosen to be the best for $k = 5$. There are also quite a few hyperparameter settings that achieved fast adaptation in Phase 2 for $k = 10$ (not shown here). In fact, Figure D.2, and D.3 show that $k = 10$ performed pretty well for stochasticity = 0.3 and 0.5. Sub-figure (g) shows the sensitivity of the algorithm to the stepsize. As one would expect, lower stepsize results in slower learning in both Phases 1 and 2. Sub-figure (h) is particularly interesting. It shows that when the model stepsize is large (equal or larger than 0.05/5) the performance in Phase 2 has a high variance. The model stepsize can not be too small either: with $\beta = 0.005/5$ the agent didn't learn to move to the right terminal in the first phase.

For better understanding of the tested linear algorithms, we plotted their learned values, policies, reward model, and predicted next value for adaptive linear Dyna (Figure D.4) and linear Dyna (Figure D.5). Plots for Sarsa(λ) are summarized in Figure D.6. We also plotted learned values and policies by the Sarsa(λ) algorithm in the two tasks when the algorithm is applied for sufficiently long.

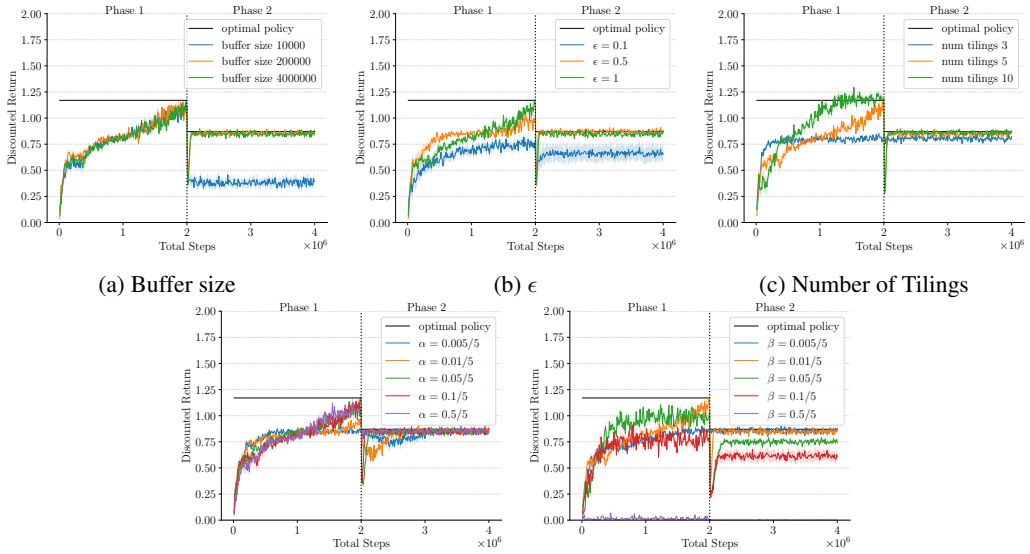
First, we note that the model learning part of the adaptive linear Dyna algorithm with sparse tile-coded features does not suffer from catastrophic forgetting. This can be seen by comparing sub-figures (b) and (f) in Figure D.4 or those in Figure D.5. We also note that, in Phase 1, linear Dyna produced inferior value estimates (the upper part of sub-figure (a) should be less than 4 because of discounting while it is not) even though the greedy policy w.r.t. these values is not apparently inferior (sub-figure (d)). In Phase 2, the inferior values are even more significant, and the policy did not produce a trajectory that leads to T2. We believe that incorrect value estimate in Phase 1 is not as severe as it in Phase 2 because the model-free learning part of the linear Dyna algorithm helps value estimation for most of states in Phase 1, but only for states within T1-zone in Phase 2.



(a) Buffer size (b) ϵ (c) Number of Tilings

(d) Stepsize (e) Model Stepsize

Figure D.2: Plots showing the sensitivity of the adaptive linear Dyna algorithm to its hyperparameters in the stochastic MountainCarLoCA domain (stochasticity = 0.3).



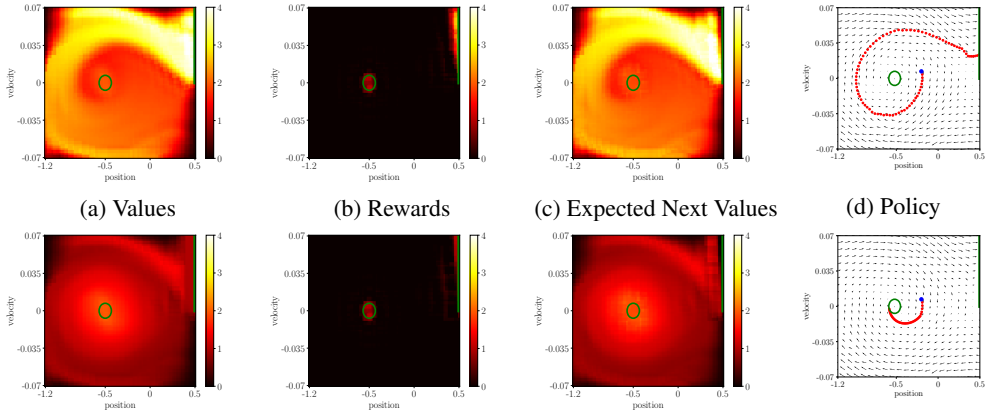
(a) Buffer size (b) ϵ (c) Number of Tilings

(d) Stepsize (e) Model Stepsize

Figure D.3: Plots showing the sensitivity of the adaptive linear Dyna algorithm to its hyperparameters in the deterministic MountainCarLoCA domain.

D.4 Nonlinear Dyna Q

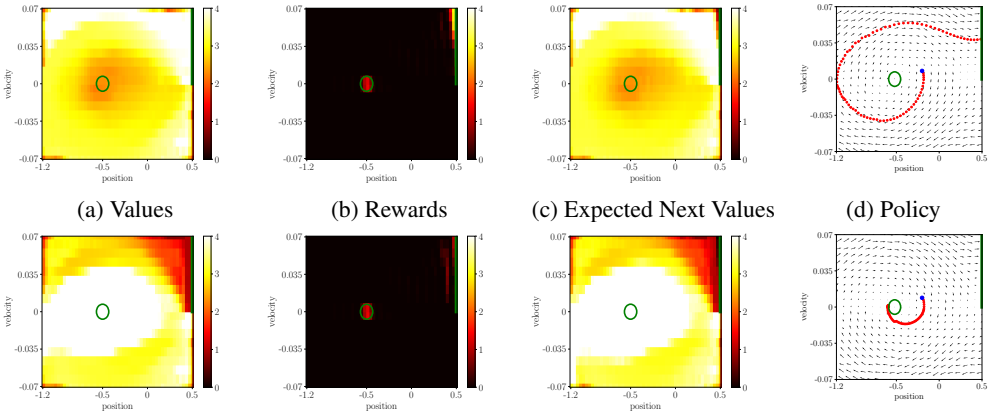
Based on the adaptive linear Dyna algorithm, we propose the nonlinear Dyna Q algorithm (Algorithm 2), in which the value function and the model are both approximated using neural networks. Instead of trying to solve the catastrophic interference problem, we adopt the simple replay approach. Specifically, we maintain a *learning* buffer, in addition to the planning buffer, to store recently visited transitions and sample from the learning buffer to generate data for model learning. As discussed previously, this approach falls into a dilemma of learning from stale information (Failure Mode #3) or forgetting previously learned information (Failure Mode #4). To empirically verify if the dilemma is critical, we varied the size of the learning buffer as well as other hyperparameters to see if there is one parameter setting that supports adaptation. Further, we also tried two strategies of sampling from the learning buffer: 1) randomly sampling from the entire buffer, and 2) sampling half of the data



(a) Values (b) Rewards (c) Expected Next Values (d) Policy

(e) Values (f) Rewards (g) Expected Next Values (h) Policy

Figure D.4: Plots showing estimated values, reward model, the value of the predicted next state produced by the model, and the learned policy of adaptive linear Dyna in the deterministic MountainCarLoCA domain, with the best hyperparameter setting (Table D.3). The first and second row correspond to the results at the end of Phases 1 and 2 respectively. In each sub-figure, the x-axis is the position of the car, the y-axis is the velocity of the car. For each state (position and velocity), we plotted the estimated value in the first column. The second corresponds to the maximum of estimated immediate reward. The third column corresponds to the maximum of the value of the model's predicted next state. In both cases, the maximum was taken over different actions. In the fourth column, we drew a vector field, where each vector represents the evolution of a state under the learned policy. We further drew an sequence of red dots, illustrating evolution of states in one particular evaluation episode. A large blue dot marks the start state of the episode. Comparing Sub-figure (a), (e) and Sub-figure a), c) in Figure D.7 show that resulting values are close to optimal values for most of states.



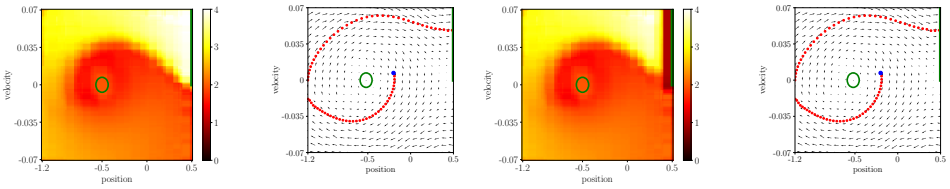
(a) Values (b) Rewards (c) Expected Next Values (d) Policy

(e) Values (f) Rewards (g) Expected Next Values (h) Policy

Figure D.5: Plots showing estimated values, reward model, the value of the predicted next state produced by the model, and the learned policy of linear Dyna in the deterministic MountainCarLoCA domain. Comparing Sub-figure (a), (e) and Sub-figure a), c) in Figure D.7 show that resulting values are far from optimal values.

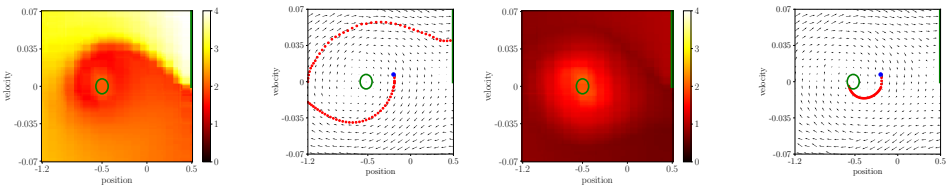
randomly from the entire buffer and the rest randomly only from rewarding data (transitions leading to rewards). Our empirical results in the MountainCarLoCA domain presented in Section D.5 confirms that this dilemma is critical – the learned model is inferior, even with the best hyperparameter setting and sampling strategy.

The other important difference between the non-linear algorithm and the linear algorithm is the data used in model learning. The model in the non-linear algorithm is learned using off-line data sampled from a replay buffer, while it in the linear algorithm is learned using online data. Specifically, we



(a) Phase 1 Values (b) Phase 1 Policy (c) Phase 2 Values (d) Phase 2 Policy

Figure D.6: Plots showing learning curves of linear Sarsa(λ) in the deterministic MountainCarLoCA domain. It can be seen that in the second phase, only the action values of a local region were updated because only data in that region were collected. And thus the policy is not updated for most of states.



(a) Task 1 Values (b) Task 1 Policy (c) Task 1 Values (d) Task 1 Policy

Figure D.7: Plots showing "optimal" values and policies in the two tasks obtained by running linear Sarsa(λ) for sufficiently long in the deterministic MountainCarLoCA domain.

maintain a *learning* buffer, in addition to the planning buffer, to store recently visited transitions and sample from the learning buffer to generate data for model learning.

D.5 NONLINEAR DYNA Q EXPERIMENT

We performed an experiment to test if the nonlinear Dyna Q algorithm can successfully adapt in the deterministic MountainCarLoCA domain. Tested hyperparameters are specified in Table D.4. The hyperparameter setting used to generate the reported learning curve (Figure D.8(a)) chosen in a similar way as those chosen for the adaptive linear Dyna experiment.

The learning curve corresponding to the best hyperparameter setting is the one using a large learning buffer, and the sampling strategy that emphasizes rewarding transitions. Nevertheless, even the best hyperparameter setting only produced an inferior policy, as illustrated in the sub-figure (a) of Figure D.8. We picked one run with the best hyperparameter setting and plotted the estimated reward model at the end of Phase 2 in Figure D.8. Comparing the predicted rewards by the learned model with the true rewarding region marked by the green circle shows catastrophic forgetting severely influences model learning. In fact, we observed that if the model in Phase 2 is trained even longer, eventually T2's reward model will be completely forgotten.

Neural networks	Dynamics model	$64 \times 64 \times 64 \times 64$, tanh
	Reward model	$64 \times 64 \times 64 \times 64$, tanh
	Termination model	$64 \times 64 \times 64 \times 64$, tanh
	Action-value estimator	$64 \times 64 \times 64 \times 64$, tanh
Optimizer	Value optimizer	Adam, step-size $\alpha \in [5e-7, 1e-6, 5e-6, 1e-5, 5e-5]$
	Model optimizer	Adam, step-size $\beta \in [1e-6, 5e-6, 1e-5, 5e-5, 1e-4]$
	Other value and model optimizer parameters	Default choices in Pytorch (Paszke et al., 2019)
Other details	Exploration parameter (ϵ)	$[0.1, 0.5, 1.0]$
	Target network update frequency (k)	500
	Number of model learning steps (m)	5
	Number of planning steps (m)	5
	Size of learning buffer (n_l)	$[300000, 3000000]$
	Size of planning buffer (n_p)	3000000
	Mini-batch size of model learning (m_l)	32
	Mini-batch size of planning (m_p)	32

Table D.4: Tested hyperparameters in the nonlinear Dyna Q experiment.

Initial distributions	Phase 1 training	Uniform distribution over the entire state space
	Phase 1 evaluation	Uniform distribution over a small region
	Phase 2 training	Uniform distribution over states within T1-zone
	Phase 2 evaluation	Uniform distribution over a small region
Training steps	Phase 1 steps	1.5×10^6
	Phase 2 steps	1.5×10^6
Other details	Training steps between two evaluations	10^4
	Number of runs	5
	Number of evaluation episodes	10

Table D.5: Experiment details used to test nonlinear Dyna Q.

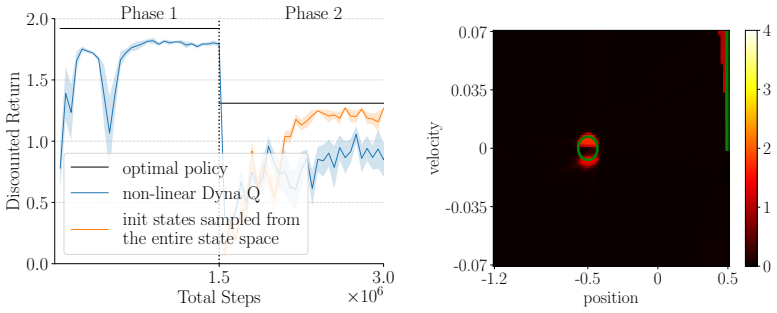
The set of hyperparameters used to generate learning curves in Figure 5 is chosen in the way the set of hyperparameters chosen in the adaptive linear Dyna experiment. The best hyperparameter setting of nonlinear Dyna Q in this experiment is summarized in Table D.6.

Optimizer	Value optimizer	Adam, step-size $\alpha = 5e-6$
	Model optimizer	Adam, step-size $\beta = 5e-5$
Other hyperparameters	Exploration parameter (ϵ)	0.5
	Size of learning buffer (n_l)	3000000

Table D.6: Best hyperparameter setting for nonlinear Dyna Q.

Just as what we did for linear algorithm, we plot the estimated values, reward and termination models, as well as the learned policy in Figure D.9.

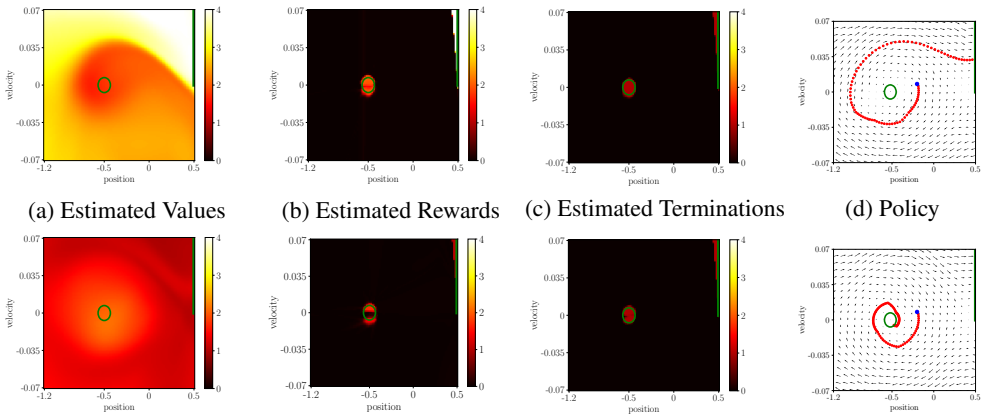
Comparing sub-figures (b) and (f) shows that the middle orange region changes to two smaller red regions, indicating that the algorithm forgets the reward model as more and more training goes on for



(a) Learning Curves (b) Reward Model

Figure D.8: Plots showing learning curves and the estimated reward model of nonlinear Dyna Q at the end of training in the deterministic MountainCarLoCA domain. a) nonlinear Dyna Q struggles to adapt in Phase 2 (blue curve). As a baseline, we show the learning curve produced by applying nonlinear Dyna Q to Task B, with initial states sampled from the entire state space. This shows the best performance this algorithm can achieve for Task B. b) The x and y axes represent the position and the velocity of the car and thus each point represents a state. The green eclipse indicates the region of the T2 area. The T1 area is marked by a green line (right end of the graph). The color of each point represents the model's reward prediction of a state (maximized over different actions).

the T1 part. Although sub-figure (e) still seems to be correct, but a deeper look at the learned policy shows that the trajectory generated by the policy (sub-figure (h)) is not optimal (c.f. sub-figure (h) in Figure D.4).



(a) Estimated Values (b) Estimated Rewards (c) Estimated Terminations (d) Policy

(e) Estimated Values (f) Estimated Rewards (g) Estimated Terminations (h) Policy

Figure D.9: Plots showing learning curves, estimated values, rewards, and learned policy of nonlinear Dyna Q in the deterministic MountainCarLoCA domain.

E PSEUDO CODE

We provide the pseudo code of the adaptive linear Dyna algorithm, and the nonlinear Dyna Q algorithm in Algorithm 1 and 2 respectively.

Algorithm 1: Adaptive Linear Dyna

Input: exploration parameter ϵ, stepsize α, a feature mapping $\phi : \mathcal{S} \to \mathbb{R}^m$ that produces from a state s the corresponding m-sized feature vector $\phi(s)$, number of planning steps n, and the size of the planning buffer n_p.

Initialize weights of the approximate value function $\theta \in \mathbb{R}^m$, weights of the linear dynamics model for each action $F_a \in \mathbb{R}^m \times \mathbb{R}^m$, and weights of the linear reward model for each action $b_a \in \mathbb{R}^m, \forall a \in \mathcal{A}$ arbitrarily (e.g., to zero)
Obtain initial state S, Compute feature vector ϕ from S.
while *still time to train* **do**
    $A \leftarrow \epsilon\text{-greedy}_a(b_a^\top \phi + \gamma \theta^\top F_a^\top \phi)$
    Take action A, receive reward R, next state S', and episode termination signal Z ($Z = 1$ means the current episode terminates. $Z = 0$ means the episode continues).
    Add ϕ to the planning buffer (if the buffer is full, remove the oldest element and then add ϕ).
    Compute the next feature vector ϕ' from S'.
    if *A is greedy* **then**
        $\delta \leftarrow R + \gamma(1 - Z)\theta^\top \phi' - \theta^\top \phi$
        $\theta \leftarrow \theta + \alpha \delta \phi$
    end
    $F_A \leftarrow F_A + \beta((1 - Z)\phi' - F_A \phi)\phi^\top$
    $b_A \leftarrow b_A + \beta(r - b_A^\top \phi)\phi$
    for *all of the n planning steps* **do**
        Generate a start feature vector x by randomly sampling from the planning buffer.
        $\delta \leftarrow \max_a(b_a^\top x + \gamma \theta^\top F_a^\top x) - \theta^\top x$
        $\theta \leftarrow \theta + \alpha \delta x$
    end
    $\phi \leftarrow \phi'$
end

Algorithm 2: Nonlinear Dyna Q

Input: exploration parameter ϵ, value step-size α, model step-size β, target network update frequency k, parameterized value estimator $\hat{q}_{\mathbf{w}}$, parameterized dynamics model $f_{\mathbf{u}}$, parameterized reward model $b_{\mathbf{v}}$, parameterized termination model $t_{\mathbf{z}}$, number of model learning steps m, number of planning steps n, size of the learning buffer n_l, size of the planning buffer n_p, mini-batch size of model-learning m_l, mini-batch size of planning m_p.

Randomly initialize $\mathbf{w}, \mathbf{u}, \mathbf{v}, \mathbf{z}$. Initialize target network parameters $\tilde{\mathbf{w}} = \mathbf{w}$. Initialize value optimizer with step-size α and parameters $\mathbf{w}$, model optimizer with step-size β and parameters $\mathbf{u}, \mathbf{v}, \mathbf{z}$.

Obtain the vector representation of the initial state $\mathbf{s}$.

while *still time to train* **do**

  $A \leftarrow \epsilon\text{-greedy}_a(\hat{q}_{\mathbf{w}}(\mathbf{s}, \cdot))$

  Take action A, receive reward R, vector representation of the next state $\mathbf{s}'$, and termination Z.

  Add $(\mathbf{s}, A, R, \mathbf{s}', Z)$ to the learning buffer. If the learning buffer is full, replace the oldest element by $(\mathbf{s}, A, R, \mathbf{s}', Z)$.

  Add $\mathbf{s}$ to the planning buffer. If the planning buffer is full, replace the oldest element by $\mathbf{s}$.

  for *all of the m model learning steps* **do**

    Sample a m_l-sized mini-batch $(\mathbf{s}_i, a_i, r_i, \mathbf{s}'_i, z_i)$ from the learning buffer.

    Apply model optimizer to the loss
    $\sum_i \frac{1-z_i}{2}(\mathbf{s}'_i - f_{\mathbf{u}}(\mathbf{s}_i, a_i))^2 + \frac{1}{2}(r_i - b_{\mathbf{v}}(\mathbf{s}_i, a_i))^2 + \frac{1}{2}(z_i - t_{\mathbf{z}}(\mathbf{s}_i, a_i))^2$

  end

  for *all of the n planning steps* **do**

    Sample a m_p-sized mini-batch of feature vectors $\{\mathbf{x}_i\}$ from the planning buffer.

    Sample a m_p-sized mini-batch of actions $\{\mathbf{a}_i\}$ uniformly.

    Apply value optimizer to the loss
    $\sum_i \frac{1}{2}(b_{\mathbf{v}}(\mathbf{x}_i, \mathbf{a}_i) + \gamma(1 - t_{\mathbf{z}}(\mathbf{x}_i, \mathbf{a}_i)) \max_a \hat{q}_{\tilde{\mathbf{w}}}(f_{\mathbf{u}}(\mathbf{x}_i, \mathbf{a}_i), a) - \hat{q}_{\mathbf{w}}(\mathbf{x}_i, \mathbf{a}_i))^2$

  end

  $\mathbf{s} \leftarrow \mathbf{s}'$

  $t \leftarrow t + 1$

  if $t\%k == 0$ **then**

    $\tilde{\mathbf{w}} = \mathbf{w}$

  end

end