TOWARDS EVALUATING ADAPTIVITY OF MODEL-BASED REINFORCEMENT LEARNING

Anonymous authors
Paper under double-blind review

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) setup, that the well-known model-based methods PlaNet and DreamerV2 adapt poorly to local environmental changes. Combined with prior work that made a similar observation about the other popular model-based method, MuZero, a trend emerges that suggests that current deep model-based methods have serious limitations. We dive deeper into the causes of this poor adaptivity, 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 uses samples to learn a model of the transition dynamics and reward, and planning techniques to derive a policy, 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 that 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) setup to measure the agent's ability to adaptation when the task changes. This setup, inspired by setups 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 setup 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 simplify the LoCA setup, such that it can be more easily applied to various domains and provides a more straightforward interpretation. Our new setup 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 simplified LoCA setup 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 three failure modes that prohibit adaptivity. Further analysis of Planet and DreamerV2 enables us to identify two additional failure modes that are unique to approximate (i.e., non-tabular) model-based methods.

Using the insights about important failure modes, we identify an adaptive MBRL method that uses a linear model. In particular, we show that making a small modification to the linear Dyna method, a method is obtained that adapts effectively in a challenging setting for MBRL methods (sparse reward and stochastic transitions).

2 Measuring Adaptivity of Model-Based Methods

We build upon the work of Van Seijen et al. (2020) that introduced the LoCA setup, an experimental setup that measures the adaptivity of an RL method to local changes in the environment. In particular, we simplify the setup such that it can be more easily extended to other domains. We formally define our simplified LoCA setup in Section 2.1 and discuss the main changes we made with regards to the original LoCA setup in in Section 2.2.

2.1 Simplified LoCA Setup

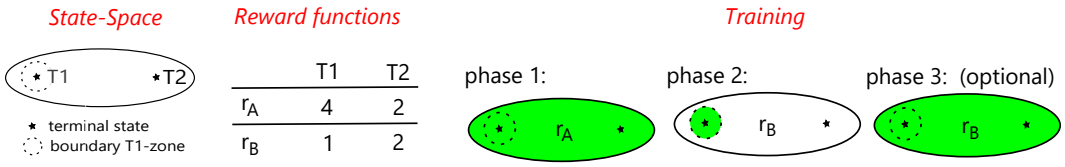


Figure 1: Simplified LoCA setup. Green indicates initial state distribution during training. Evaluation occurs at regular intervals during training, using the full state space as initial state distribution (regardless of the phase). We call a model-based method adaptive if it is able to achieve optimal performance in phase 2. During this phase the agent is only able to collect data from the T1-zone.

The simplified LoCA setup (Figure 1) involves two *tasks* (A, B) and an *experiment procedure*. This setup needs to be used together with a *domain*, which consists of a state space $\mathcal{S}$, an action space $\mathcal{A}$, and a transition function $p(s'|s,a)$. The state space should contain two terminal states, T1 and T2. Reaching a terminal state incurs termination of an episode. Furthermore, there should be a local area around T1 that, once entered, the agent cannot move out of, regardless of its policy. We refer to this local area as the *T1-zone*. The two tasks are determined by two reward functions r_A and r_B, and a shared discount factor $\gamma < 1$ (specified by the user of the setup). The reward functions for both two tasks 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 experiment procedure comprises of three phases, in which the third phase is optional. For each phase, the user of the setup needs to specify the initial state distribution for training and evaluation during the phase. In phase 2 the training distribution is restricted to the T1-zone. Other distributions are uniform over the entire state space, unless specified otherwise. Finally, other details related to the setup of experiments also need to be specified in the experimental procedure (e.g., the number of time steps for training, number of runs to be performed).

Given a domain, an agent, and the simplified LoCA setup, an experiment can be performed. In the first phase, the agent interacts with the domain and solves task A – the observed rewards are produced

according to r_A. After certain number of time steps, phase 1 terminates and phase 2 starts. In phase 2, the agent interacts with the same domain but this time it is asked to solve task B. Consequently, observed rewards are generated according to r_B. At regular intervals, the performance of the agent's policy is evaluated, by asking the agent to execute its policy for multiple episodes and computing the average discounted returns of the episodes. The policy the agent executes during evaluation can be different from the policy it uses to interact with the domain.

If an agent's policy achieves a high average discounted return in the evaluation runs in both phase 1 and phase 2, we say the agent is adaptive, otherwise we say it is not adaptive. Note that in the second phase, the reward function changes from r_A to r_B, while the training initial state distribution is restricted to the T1-zone. This means that in phase 2 the agent would never be able to visit the area outside the T1-zone during training. Therefore in order to be adaptive, the agent must be aware of the change in reward function and, more importantly, updates its policy and/or value estimation over the entire state space to incorporate the reward change.

Optionally, a third phase can be added, where the agent's training initial state distribution is not restricted the T1-zone. This phase can be used as a sanity check in case of poor performance in phase 2. If the agent performs poorly in phase 3, then it is not reasonable to expect that the agent could perform well in phase 2.

2.2 Discussion

The difference between our simplified LoCA setup and the original LoCA setup lies in the way we evaluate performance. The original LoCA setup from Van Seijen et al. (2020) is designed to obtain a *quantitative metric* of the adaptivity to local changes, called the *LoCA regret*. This metric is based on the performance during phase 3 (which is optional in our setup), with performance evaluated by counting the fraction of evaluation episodes the agent arrives in the high-reward terminal (i.e., T2 for task B), within a certain time limit. Furthermore, the initial state distribution for evaluation is a small subset of the state space roughly in between T1 and T2. There are several hyperparametres involved in determining the LoCA regret that can affect its value considerably (i.e., the exact location of the initial state distribution; the time limit specified). In addition, the best value of these hyperparameters may be hard to determine in practice. Hence, it can require considerable design effort to apply the original setup to a new domain.

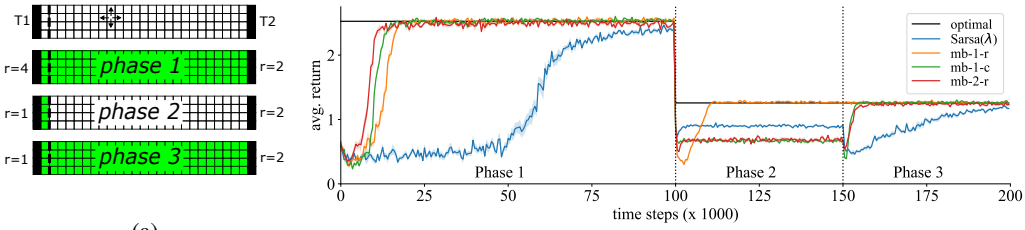
By contrast, our simplied setup—which only aims to make a *binary assessment* of adaptivity—uses the discounted return for evaluation and does not require an initial state distribution centered in the middle of T1 and T2. It also only requires two phases to determine adaptivity, although a third phase may be useful for analysis of methods that fail to adapt in phase 2. Hence, our method is much easier to apply to new domains.

For simplicity, from now on, unless specified otherwise, we use the term "LoCA setup" to denote our simplified LoCA setup.

3 Adaptive Versus Non-Adaptive MBRL

Small differences among model-based methods can greatly affect their adaptivity. We illustrate this point in this section, by testing three slightly different tabular model-based methods in the GridWorldLoCA domain (Figure 2a), introduced by Van Seijen et al. (2020), using the LoCA setup. Our empirical results suggest that only one of the tested three methods can adapt well in this domain.

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. Each of the three model-based method we introduce learns an estimate of the environment model (i.e., the transition and reward function); depending on the method, either a 1-step or a 2-step model is learned. A 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; a 2-step model estimates, under the agent's behavior policy, a distribution of the state 2 time steps in the future, using $\hat{p}_2(s''|s,a)$, and the expected discounted sum of rewards over the next 2 time steps, using $\hat{r}_2(s,a)$. Planning consists of performing a single state-value update



(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 Phase 1 and Phase 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.

at each time step based on the model estimates. For a 1-step model:

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

and for a 2-step model:

$$V(s) \leftarrow \max_a \left(\hat{r}_2(s,a) + \gamma^2 \sum_{s''} \hat{p}_2(s''|s,a) V(s'') \right),$$

Depending on the method, the state s that received this value update is either the current state or a randomly selected state. See Table 1 for an overview. All methods use ϵ-greedy action-selection, 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. As a reference point, we also tested the performance of the model-free method Sarsa(λ) with $\lambda = 0.95$. [1]

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

The LoCA setup here uses a discount factor $\gamma = 0.97$. And the initial distributions follow what has been specified in Section 2.1.

Figure 2 shows the performance of the various methods, averaged over 10 independent runs. What these results reveal is that while the methods are very similar, only *mb-1-r* is able to change its policy to the optimal policy of task B by only observing the local reward change. 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. These experiments illustrate two different reasons why a model-based method may fail to adapt effectively.

Failure Mode #1: *States receiving value/policy updates do not cover the full state space.*

This is the case for *mb-1-c*, as only the current state received a value update. Therefore, during phase 2 only states within the T1-zone are updated and evaluating the agent's policy in states outside the T1-zone results in low returns.

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

[1] The code of the experiments will be released upon acceptance of this manuscript

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 largely remains, resulting in poor adaptivity.

One strategy to tackle this failure mode is to remove the implicit dependency on the behavior policy and use a model that takes as input, besides a state, a sequence of actions to some horizon. However, the number of possible action-sequences increases exponentially with the horizon. And because the model will typically be trained with only a small subset of all possible policy, this leads to the following failure mode:

Failure Mode #3: *An environment model that predicts future states/rewards conditioned on any action-sequence (up to some horizon), but that is trained on only a small subset of possible action-sequences..*

This failure mode appears to apply to MuZero, which trains a multi-step model using trajectories generated by the behavior policy.Hence, while the model can in principle make predictions for any action-sequence, in practise, only predictions for action-sequences close the behavior policy will be accurate.

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 the LoCA setup in a variant of the Reacher domain, which involves a continuous-action domain with $64 \times 64 \times 3$ dimensional (pixel-level) states.

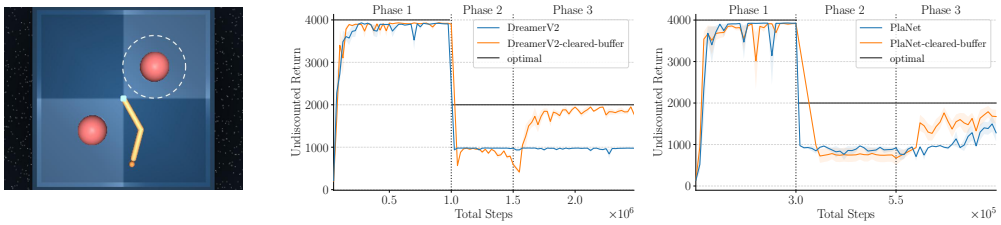
4.1 THE REACHERLOCA DOMAIN

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 such a way 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 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 3a). 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 our LoCA setup 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.

4.2 EXPERIMENTS

Both PlaNet and DreamerV2 learn from transitions sampled from a large experience replay buffer that contains all recently visited transitions. When evaluating these algorithms in the ReacherLoCA Environment, such a strategy could hurt the agent's ability of adaptation, because in the 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 phase data in the buffer get removed, if the replay buffer is large. Therefore a natural question to ask is whether the agent can adapt well, with stale data in the replay buffer. As we expect this default version to perform poorly, we also tested a version that clears the buffer in between phases, such that there is no data from old phases in the replay buffer. Note that clearing the buffer in between phases assumes the agent is aware when there is change in phase, which we



(a) The ReacherLoCA Domain (b) DreamerV2 and PlaNet Evaluation

Figure 3: a) Illustration of the ReacherLoCA domain. The dashed white circle shows the T1-zone. b) Plots showing the learning curves of DreamerV2 and PlaNet methods. 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). The middle plot is for DreamerV2, and the rightmost one is for PlaNet.

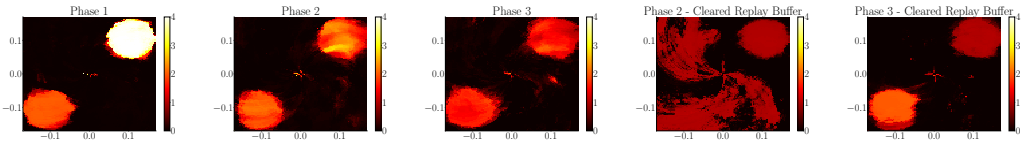


Figure 4: Visualization of the DreamerV2 agent's estimated reward model generated at the end of each phase. The x and y axes of each heatmap represent the agent's position in the ReacherLoCA domain. The leftmost plot shows the reward estimates at the end of phase 1. We used the same agent to train the other phases for both the setting where we reinitialize the replay buffer between phases (the two rightmost plots) and where we don't.

did not assume up to now. We allow it here, as it's an informative setting to study. In a sense, the performance in this setting can be interpreted as an upperbound of what is possible when optimizing the buffer size of an agent that is unaware of the phase change: using smaller buffers has also the effect that data from a previous phase gets quickly removed from the buffer.

We tested both PlaNet and DreamerV2 with and without clearing of the replay buffer in between phases. In each case, the replay buffer is sufficiently large to contain all transitions observed during training. For each algorithm, we did a grid search only over the critical hyperparameters suggested by the corresponding paper. Unless specified otherwise, other hyperparameters were chosen to be the same as those found to be the best by the corresponding paper in the original Reacher environment. For each algorithm and both replay-buffer settings, we drew the learning curves (Figure 3b) corresponding to the best hyperparameter setting found by our grid search.

The reported learning curves suggest that neither of the tested algorithms effectively adapted their policy in phase 2, regardless of the replay buffer being cleared or not. Further, when the replay buffer was not cleared, the algorithms also performed poorly in phase 3. To obtain a better understanding of the failure of adaptation, we plotted the reward predictions of DreamerV2's world model at the end of each phase (Figure 4). The corresponding results for Planet is similar and is not shown here.

First, we analyze the case where we do not clear the replay buffer between phases. In this case, the predicted reward of T1 overestimates the actual reward, which is 1 in both phase 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 phase 2 and 3 show that the resulting inferior model could be detrimental for planning.

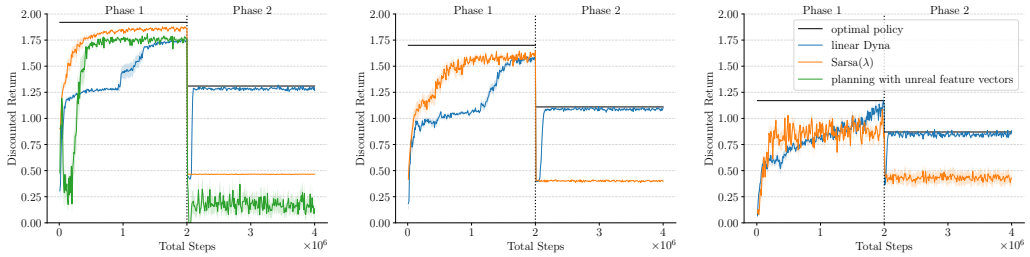
In the case where we do clear the replay buffer between phases, the values for T1 at the end of phase 2 are correct now, but the estimates for the rest of the state space are completely incorrect. This result can be explained by the fact that the agent had no data outside of the T1-zone to rehearse in phase 2 and forgot the learned reward model for states outside the T1-zone. Such an issue is called *catastrophic forgetting*. Overall, we conclude that DreamerV2 achieved poor adaptivity to local changes due to two additional failure modes that are different from those outlined earlier.

Failure Mode #4*: Learning from large replay buffers cause interference from the old task.*

Failure Mode #5*: 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 #4 is inevitable. This suggests that solving the catastrophic forgetting problem is essential for achieving adaptivity to local environment changes 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 et al., 1999). 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 Linear Function Approximation



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

Figure 5: Learning curves of various methods on the MountainCarLoCA domain. The 'optimal policy' baseline is constructed by determining the best discounted return achieved by Sarsa(λ), after training for sufficiently long in Task A and B, with an initial-state distribution spanning the entire state space.

So far, we have seen that various modern MBRL methods adapt poorly to local changes in the environment and we identified 5 failures modes that underly poor adaptivity. In this section, we aim to use the lessons learned to construct an MBRL method based on function approximation that *does* adapt effectively.

First, we consider the best model to use. Failure modes #2 and #3 highlight the challenges with multi-step models. Hence, to avoid these, we stick to a 1-step model. More specifically, to handle the general case of stochastic environments, we'll use a 1-step *expectation* model, which takes as input a state (represented by a feature vector) and an action and outputs the expected next state and expected reward. Second, we consider the planning procedure. Rolling out a 1-step model over long horizons leads to compounding errors, precluding accurate long-term predictions. Therefore, we draw single samples from the model that we use to perform value iteration updates of a linear value function. Wan et al. (2019) showed that combining these type of updates with an expectation model is equivalent to an aligned distribution model. Furthermore, the method 'mb-1-r' from Section 3 showed that this approach can yield adaptive behavior, as long as the samples that are used for the updates are drawn from the full state-space. In order to adopt this approach in a function approximation setting, one important difference needs to be considered. In the tabular case, states are simply indices and drawing a random state from the state-space is as simple as drawing a random integer; by contrast, with function approximation states are represented as feature vectors and it's not clear for a random feature vector whether or not it corresponds to a real state. Hence, we store all observed feature vectors in a *planning buffer* and, when planning, select a random feature vector from this buffer. The selected feature vector is then fed into the 1-step model to generate a full sample that is used for a value iteration update of the value function.

We do not directly tackle failure mode #5, related to catastrophic forgetting in this paper. Instead, we focus for now on the setting of linear function approximation with sparse features, which does not suffer from catastrophic forgetting. This means as well that we do not require a replay buffer for learning, which saves us from failure mode #4.

Our approach can be viewed as a variation of linear Dyna. In the original linear Dyna method (Algorithm 4 by Sutton et al. (2012)), the same model is being learned and 1-step updates are performed as well. The main difference is that in our variation we use a planning buffer, which allows us to draw samples corresponding to real states; by contrast, the planning approach of the original linear Dyna method uses 1-hot feature vector. Hence, these do not correspond to real states. In our experiments, we refer to the original linear Dyna as 'linear Dyna with unreal feature vectors' and we refer to our variation simply as 'linear Dyna'.

5.1 Experiments

We tested our algorithm on a variant of the MountainCar domain using the LoCA setup. Our variant of the MountainCar domain extends the variant of the same domain used by Van Seijen et al. (2020), by introducing stochasticity. 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. As representation for linear function approximatin we used tile-coding (Sutton & Barto, 2018). The details of other experimental setups are summarized in the experiment procedure in Section B.1.

The learning curves in phase 1 show that for all different levels of stochasticities tested here, linear Dyna performs well in phase 1 and adapts super quickly and achieved near-optimal return in phase 2. The result for Sarsa(λ) coincides with our expectation. It performed well in phase 1, but failed to adapt its policy in phase 2. Linear Dyna with tabular feature vectors performed well in phase 1, but not in phase 2, which is somewhat surprising because the unreal feature vectors were used in both phases. A deeper look at our experimental data (Figure B.4 and B.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 an discrepancy between phase 1 and 2 is due to the fact that the model-free learning part of the linear Dyna algorithm (line 3 in Algorithm 1) helps 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 linear Dyna. Note that there is a very high variance in the learning curve of Sarsa(λ). On the contrary, 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.

In Appendix B.7, we provide a study of the algorithm's sensitivity to its hyperparameters.

6 Discussion and Conclusion

We studied the adaptivity of deep MBRL methods using a simplified version of the LoCA setup. 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 five important failure modes and linked these to existing deep MBRL methods. The first four failure modes can be overcome by using appropriate environment models and planning techniques. The fifth failure mode, however, 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 in alternative ways.

REFERENCES

Nathaniel D Daw, Samuel J Gershman, Ben Seymour, Peter Dayan, and Raymond J Dolan. Model-based influences on humans' choices and striatal prediction errors. *Neuron*, 69(6):1204–1215, 2011.

Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to control: Learning behaviors by latent imagination. *arXiv preprint arXiv:1912.01603*, 2019a.

Danijar Hafner, Timothy Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. Learning latent dynamics for planning from pixels. In *International Conference on Machine Learning*, pp. 2555–2565. PMLR, 2019b.

Danijar Hafner, Timothy Lillicrap, Mohammad Norouzi, and Jimmy Ba. Mastering atari with discrete world models. *arXiv preprint arXiv:2010.02193*, 2020.

Thomas M Moerland, Joost Broekens, and Catholijn M Jonker. Model-based reinforcement learning: A survey. *arXiv preprint arXiv:2006.16712*, 2020.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32: 8026–8037, 2019.

Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *arXiv preprint arXiv:1911.08265*, 2019.

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

Richard S Sutton, Csaba Szepesvári, Alborz Geramifard, and Michael P Bowling. Dyna-style planning with linear function approximation and prioritized sweeping. *arXiv preprint arXiv:1206.3285*, 2012.

Yuval Tassa, Yotam Doron, Alistair Muldal, Tom Erez, Yazhe Li, Diego de Las Casas, David Budden, Abbas Abdolmaleki, Josh Merel, Andrew Lefrancq, et al. Deepmind control suite. *arXiv preprint arXiv:1801.00690*, 2018.

Harm Van Seijen, Hadi Nekoei, Evan Racah, and Sarath Chandar. The loca regret: A consistent metric to evaluate model-based behavior in reinforcement learning. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (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.

Yi Wan, Zaheer Abbas, Adam White, Martha White, and Richard S Sutton. Planning with expectation models. *arXiv preprint arXiv:1904.01191*, 2019.

Tingwu Wang, Xuchan Bao, Ignasi Clavera, Jerrick Hoang, Yeming Wen, Eric Langlois, Shunshi Zhang, Guodong Zhang, Pieter Abbeel, and Jimmy Ba. Benchmarking model-based reinforcement learning. *arXiv preprint arXiv:1907.02057*, 2019.

A SUPPLEMENTARY DETAILS AND RESULTS FOR SECTION 4

A.1 EXPERIMENT PROCEDURE OF THE LOCA SETUP TO TEST PLANET AND DREAMERV2

Initial distributions	Phase 1 training	Uniform distribution over the entire state space
	Phase 1 evaluation	Uniform over the entire states outside T1-zone
	Phase 2 training	Uniform over states within T1-zone
	Phase 2 evaluation	Uniform over the entire states outside T1-zone
	Phase 3 training	Uniform over the entire state space
	Phase 3 evaluation	Uniform 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	Steps between evaluations	PlaNet: 10^4, DreamerV2 5×10^4
	Runs	5
	Evaluation episodes	5

Table A.1: Experiment procedure of the LoCA setup to test PlaNet and DreamerV2.

A.2 REACHERLOCA DOMAIN IMPLEMENTATION DETAILS

As long as a domain implementation offers an API function to set or reload the environment state, no native domain code needs to be touched to implement the T1-zone in the LoCA setup. A wrapper around the domain can be implemented that resets the state to the previous state whenever the agent takes an action that would move the agent from inside the T1-zone to outside the T1-zone. Furthermore, through the DeepMind Control Suite API, we are able to access and reload the physics of a domain. This way, we can reload the previous physics and restore the previous state. Hence, the T1-zone is implemented in the ReacherLoCA domain by adding a wrapper that reloads the last physics when an action takes the agent outside the T1-zone.

A.3 CHOOSING THE BEST HYPERPARAMETER SETTING

The results for each experiment are averaged over 5 independent runs, and the shading area shows the standard error. Furthermore, we reported the average undiscounted reward over 5 runs as the metric at each evaluation point. We finally picked the set of hyperparameters that had the highest performance in phase 2, given that it could achieve more than 80% of optimal policy in both phases 1 and 3 over the last five evaluations. If the latter constraint is not met, we chose the set of hyperparameters that had the highest performance in phase 3.

A.4 HYPERPARAMETER DETAILS

For both two algorithms, we tried both reinitializing and not reinitializing the replay buffer. We filled the replay buffer with 50 episodes generated by a uniformly random policy whenever the buffer is reinitialized. The same strategy was also adopted in both PlaNet and DreamerV2, although fewer number of random episodes were produced there. We found that increasing the number of random episodes helps the agent to find the optimal target, not the sub-optimal one.

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. So, we increased the horizon length H to 50, optimization iterations I to 25, and candidate samples J to 4000. This change improved the performance of all the seeds at the end of phase 1. We used these hyperparameters for all three phases.

DreamerV2: 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.

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

A.5 Additional Experimental Results

The best hyperparamerter setting for DreamerV2 where the replay buffer is reinitialized is $\beta = 0.1$, $\eta = 10^{-4}$, and $\gamma = 0.99$. Furthermore, in the case where the replay buffer is not reinitialized none of the runs could achieve more than 80% of optimal policy in phase 3 over the last five evaluations. So, the setting that could achieve the highest performance in phase 3 is $\beta = 1$, $\eta = 3 \times 10^{-4}$, and $\gamma = 0.99$.

The best hyperparameter setting for PlaNet where the replay buffer is reinitialized is $\beta = 1$, and the step-size $= 10^{-3}$. Similar to DreamerV2, none of the runs with the PlaNet agent where the replay buffer is not reinitialized could achieve more than 80% of optimal policy in phase 3. So, the setting that could achieve the highest performance in phase 3 is $\beta = 0$, and the step-size $= 10^{-3}$.

Figure A.1 and A.2 show the complete results for DreamerV2, and Figure A.3 is for PlaNet.

Figure A.4 shows DreamerV2's performance where the replay buffer is not reinitialized between the phases with longer training duration in phase 3. We observe that even a more extended training duration in phase 3 does not lead to a decent performance.

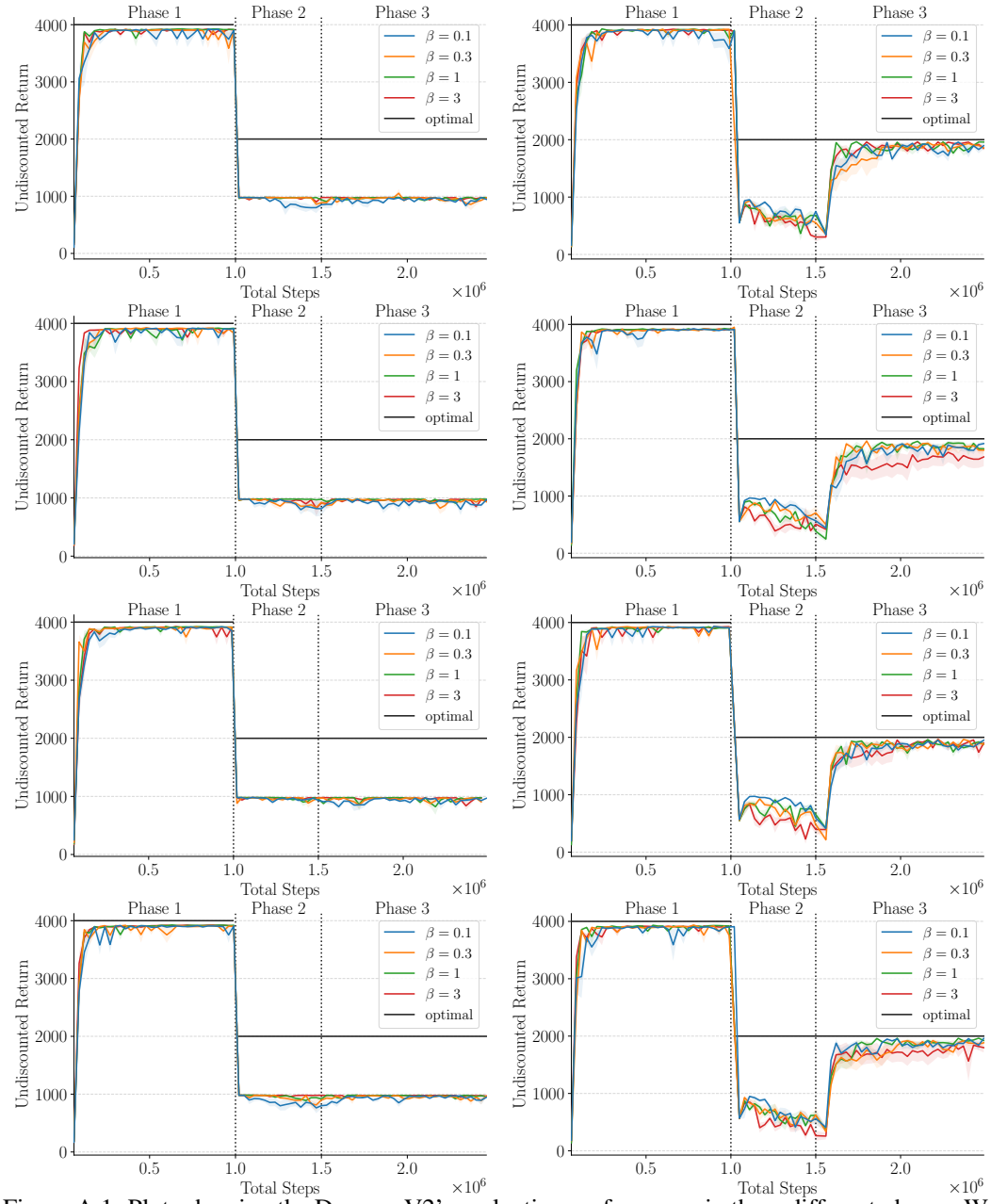


Figure A.1: Plots showing the DreamerV2's evaluation performance in three different phases. We searched over the critical hyperparameters: 1) the KL loss scale β, 2) the actor entropy loss scale η, and 3) the discount factor γ. In this set of experiments $\gamma = 0.99$. $\eta = 10^{-3}, 3 \times 10^{-4}, 10^{-4}, 10^{-5}$ in four different rows respectively. The left column represents the case in which the replay buffer is not reinitialized, and the right column shows the experiments where the replay buffer is reinitialized between the phases.

A.6 Reward Estimations of PlaNet

We plotted the reward estimations of PlaNet's world model at the end of each phase in Figure A.5. The results are for the setting where the replay buffer is reinitialized at the start of each phase. Furthermore, the results are similar to those of DreamerV2's (see Figure 4); therefore, we can make the same conclusions on PlaNet's performance as well.

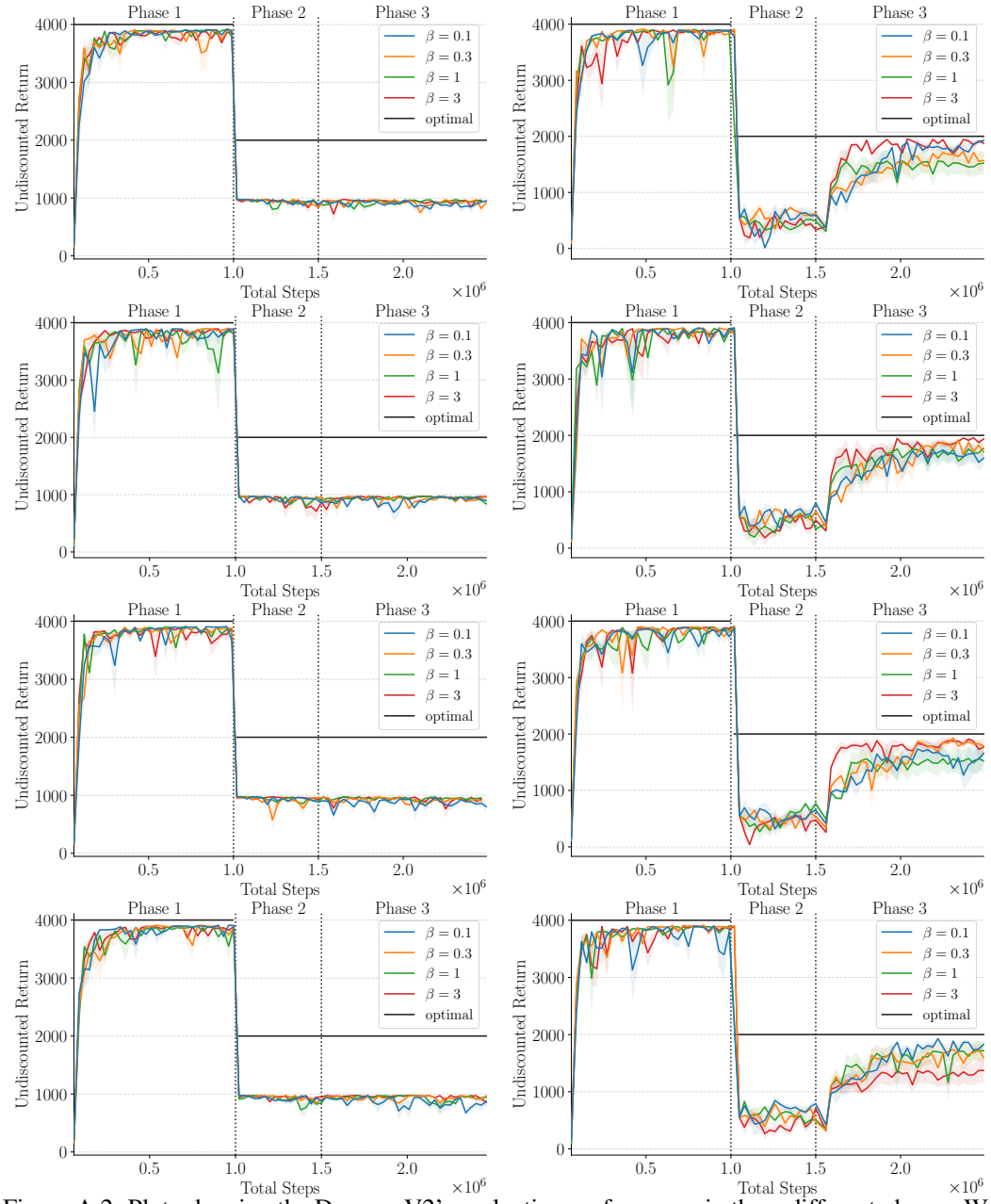


Figure A.2: Plots showing the DreamerV2's evaluation performance in three different phases. We searched over the critical hyperparameters: 1) the KL loss scale β, 2) the actor entropy loss scale η, and 3) the discount factor γ. In this set of experiments $\gamma = 0.999$. $\eta = 10^{-3}, 3 \times 10^{-4}, 10^{-4}, 10^{-5}$ in four different rows respectively. The left column represents the case in which the replay buffer is not reinitialized, and the right column shows the experiments where the replay buffer is reinitialized between the phases.

B Supplementary Details and Results for Section 5

B.1 Experiment Procedure of the LoCA Setup to Test Two Variants of Linear Dyna and Sarsa(λ)

Table B.1 shows the experiment procedure the LoCA setup to test our modified Linear Dyna (Algorithm 1), Linear Dyna with tabular feature vectors (Algorithm 2), and Sarsa(λ).

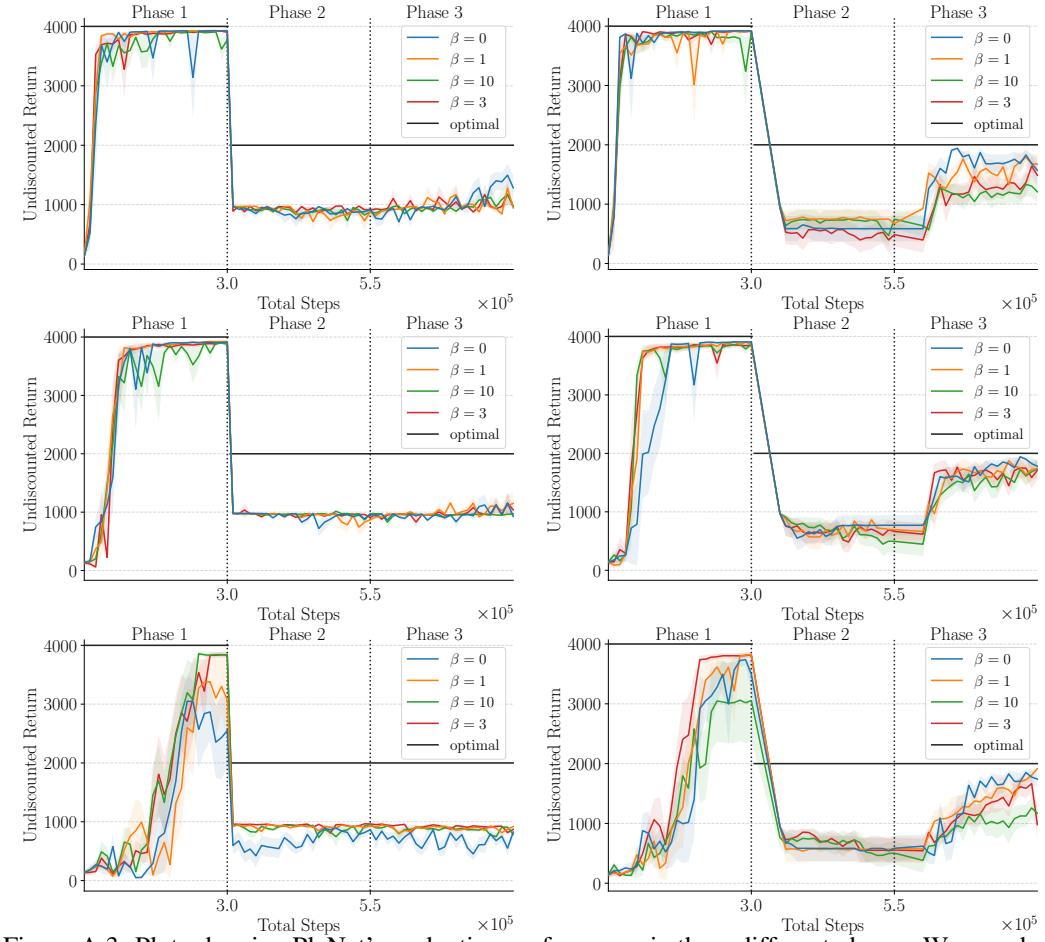


Figure A.3: Plots showing PlaNet's evaluation performance in three different phases. We searched over the critical hyperparameters: 1) the KL-divergence scales β, and 2) the step size. The step size is $10^{-3}, 3 \times 10^{-4}, 10^{-4}$ in three different rows respectively. The left column represents the case in which the replay buffer is not reinitialized, and the right column shows the experiments where the replay buffer is reinitialized between the phases.

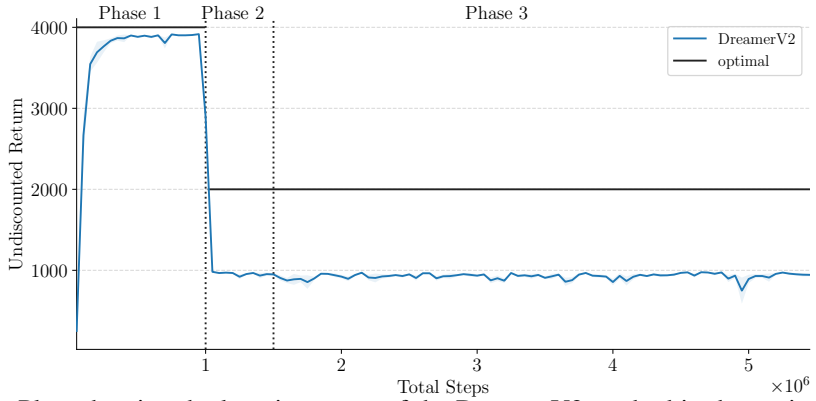


Figure A.4: Plots showing the learning curve of the DreamerV2 method in the setting where the replay buffer is not reinitialized between phases. We show the maximum achievable undiscounted reward at each phase as a baseline. To check if the agent's performance could improve through more time, we increased the total environments steps in phase 3. The curve shows that the agent fails to perform decently even at the end of a longer phrase 3.

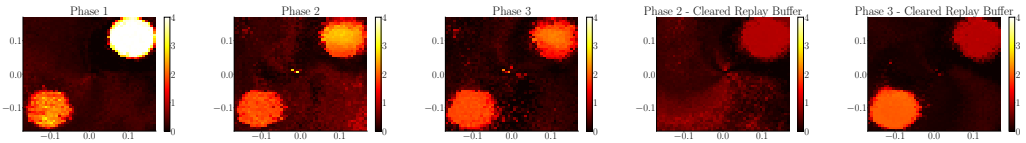


Figure A.5: Visualization of the PlaNet agent's estimated reward model generated at the end of each phase. The x and y axes of each heatmap represent the agent's position in the ReacherLoCA domain.

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

Table B.1: Experiment procedure of the LoCA setup to test linear Dyna and Sarsa(λ).

The small region is roughly at the middle of the two terminal states ($-0.2 \leq$ position ≤ -0.1, $-0.01 \leq$ velocity ≤ 0.01).

B.2 Experiment Procedure of the LoCA Setup to Test Nonlinear Dyna Q

Table B.2 shows the experiment procedure the LoCA setup to test the nonlinear Dyna Q algorithm (Algorithm 3).

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

Table B.2: Experiment procedure of the LoCA setup to test nonlinear Dyna Q.

B.3 Choosing the Best Hyperparameter Setting

The best hyperparameter setting 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.

B.4 Linear Dyna Details

We summarize in Table B.3 tested hyperparameters for the linear Dyna algorithm.

B.5 Sarsa(λ) Details

The tile coding setups and the choices of stepsize α are the same as those used for linear Dyna. The exploration parameter was fixed to be 0.1. We also tested three λ values: 0, 0.5, and 0.9. For stochasticity = 0.5, no hyperparameter setting for Sarsa achieved 90% optimal performance in the first phase and we picked the one that achieved more than 80% optimality in the first phase.

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
Size of the planning buffer n_p (Algorithm 1)	[10000, 200000, 4000000]
Number of planning steps	5

Table B.3: Linear Dyna Hyperparameters.

B.6 Nonlinear Dyna Q Details

We summarize in Table B.4 tested hyperparameters for the nonlinear Dyna Q algorithm.

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 B.4: Nonlinear Dyna Q Details.

B.7 Additional Results of Two Variants of Linear Dyna and Sarsa(λ)

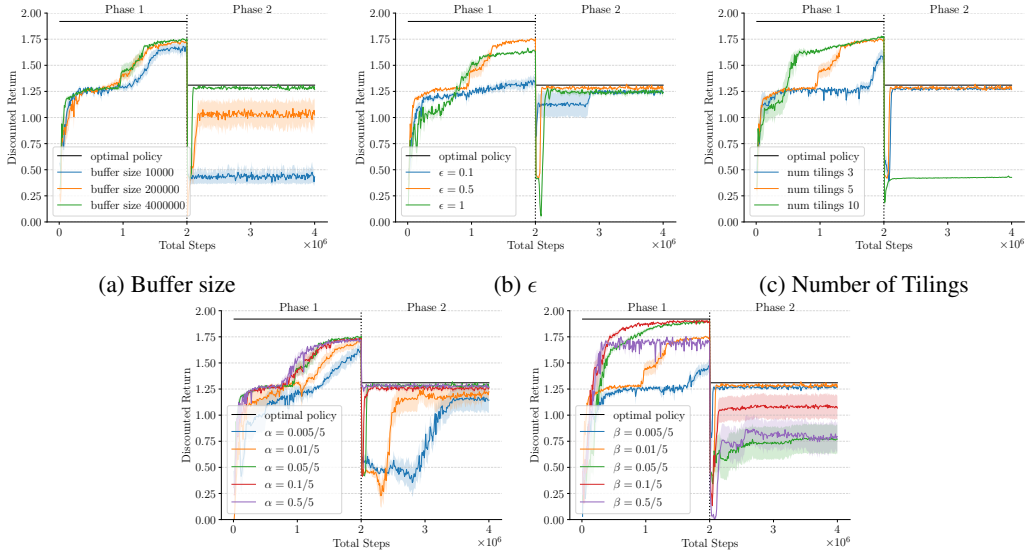
We summarize in Table B.5 the best hyperparameter settings under different level of stochasticities for each algorithm tested in Section 5.1.

Algorithm	Hyperparameter	Deterministic	Stochasticity = 0.3	Stochasticity = 0.5
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
Planning with unreal tabular feature vectors	Exploration parameter (ϵ)	0.5		
	Step-size (α)	0.5		
	Number of tilings	5		

Table B.5: Best hyperparameter settings in linear Dyna experiments.

To demonstrate the algorithm'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 B.5). 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 B.1 Figure B.2, and Figure B.3 respectively.

We now use Figure B.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 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



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

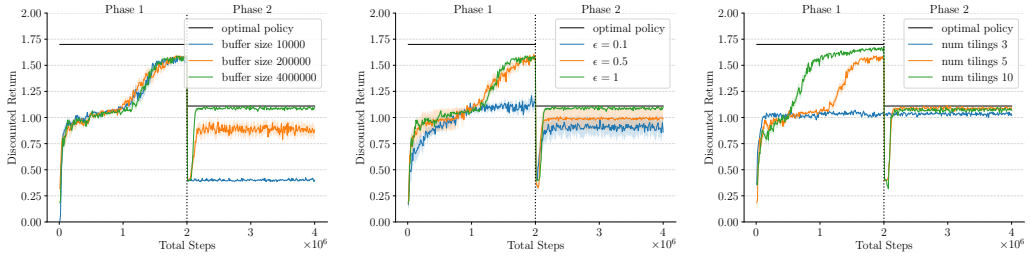
(d) Stepsize (e) Model Stepsize

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

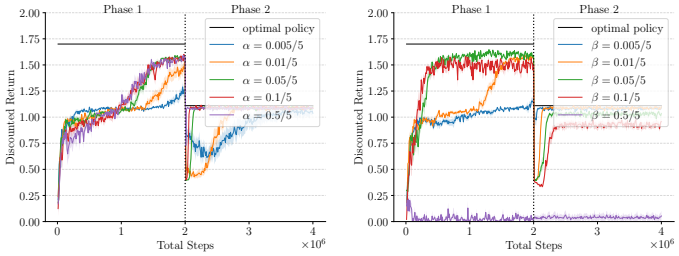
planning in phase 2 would only update states in that small region. Sub-figure (b) shows that the algorithm worked well when a relatively large exploration parameter is used. $\epsilon = 0.1$ performed well in the second phase but it was not close to optimality in the first phase. A deeper look at the 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 B.2, and B.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 phase 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 linear Dyna (Figure B.4) and the same algorithm using tabular feature vectors (Figure B.5). Plots for Sarsa(λ) are summarized in Figure B.6.

First, we note that the model learning part of the 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 B.4 or those in Figure B.5. We also note that, in phase 1, linear Dyna with unreal tabular feature 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 (line 3 in the Algorithm 1) helps value estimation for most of states in phase 1, but only for states within T1-zone in phase 2.

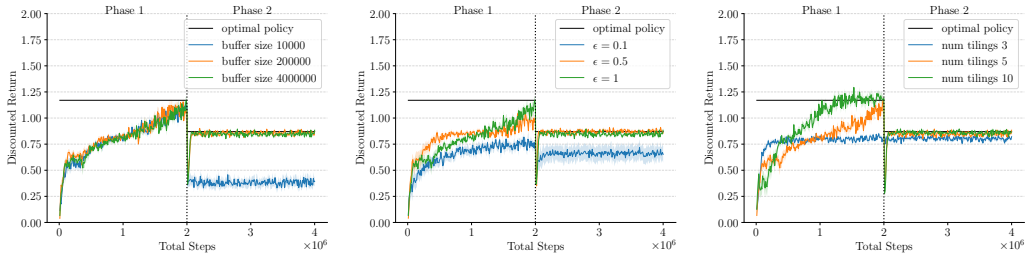


(a) Buffer size (b) ϵ (c) Number of Tilings

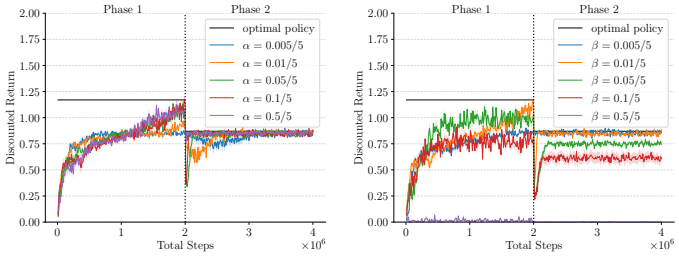


(d) Stepsize (e) Model Stepsize

Figure B.2: Plots showing the sensitivity of the 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 B.3: Plots showing the sensitivity of the linear Dyna algorithm to its hyperparameters in the deterministic MountainCarLoCA domain.

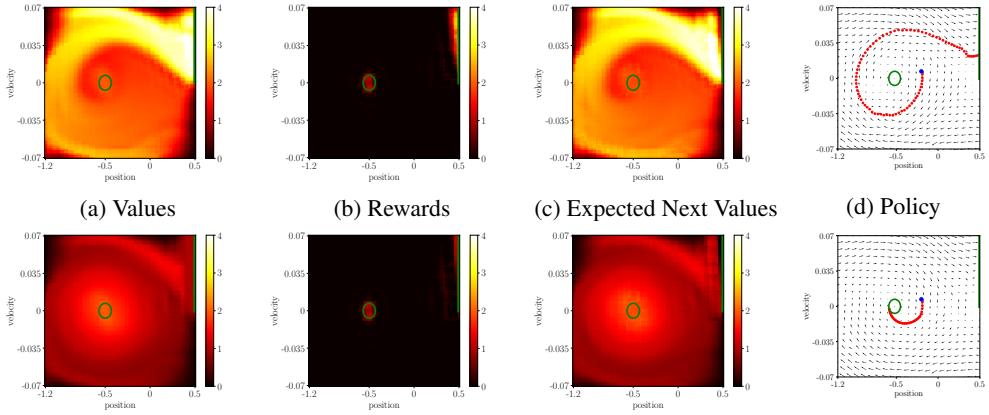
B.8 Additional Results of Nonlinear Dyna Q

The best hyperparameter setting of this algorithm in our experiment is summarized in Table B.6.

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

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

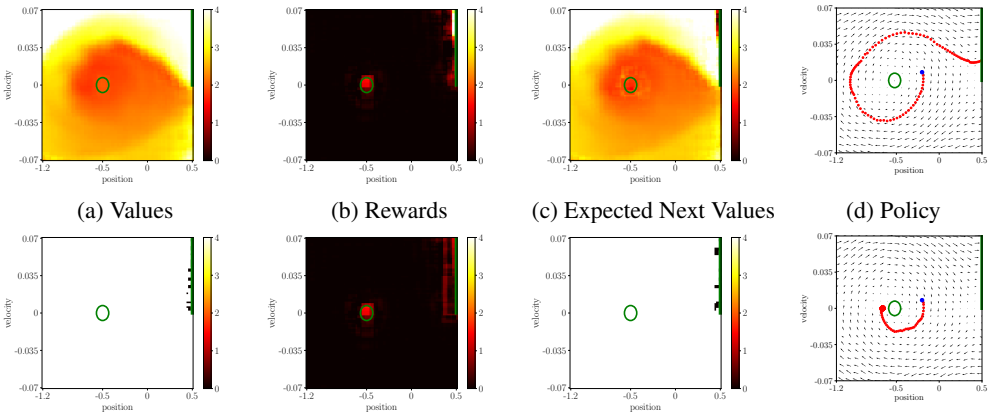
Just as what we did for linear algorithm, we plot the estimated values, reward and termination models, as well as the learned policy in Figure B.7.



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

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

Figure B.4: 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, with the best hyperparameter setting (Table B.5). The first and second row correspond to the results at the end of phase 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.

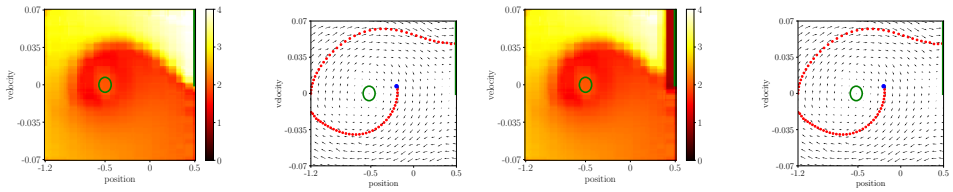


(a) Values (b) Rewards (c) Expected Next Values (d) Policy

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

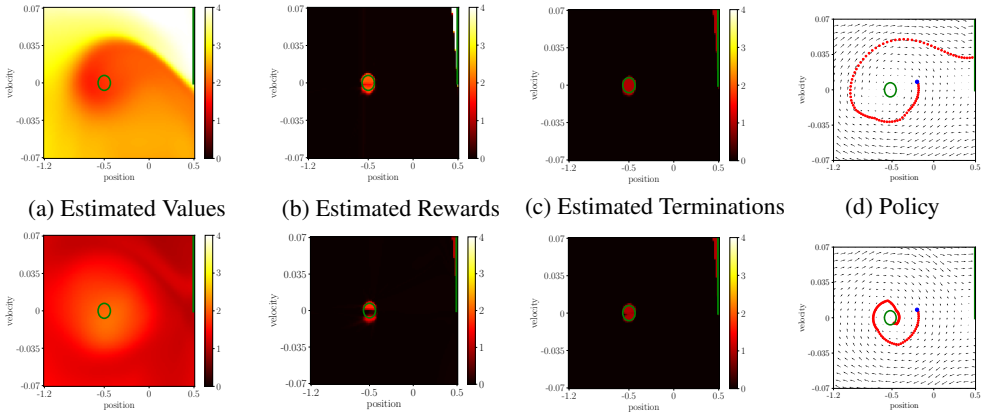
Figure B.5: Plots showing estimated values, reward model, the value of the predicted next state produced by the model, and the learned policy of linear Dyna with tabular planning feature vectors in the deterministic MountainCarLoCA domain. Sub-figure(e) shows that resulting values are far from true values when planning with unreal tabular feature vectors. Sub-figure (h) shows that in this run, the algorithm did not reach the goal state T2.

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 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 B.4).



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

Figure B.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) Estimated Values (b) Estimated Rewards (c) Estimated Terminations (d) Policy

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

Figure B.7: Plots showing learning curves, estimated values, rewards, and learned policy of nonlinear Dyna Q in the deterministic MountainCarLoCA domain.

C PSEUDO CODE

We provide the pseudo code of the our modified linear Dyna algorithm, linear Dyna with tabular feature vectors for planning, and the nonlinear Dyna Q algorithm in Algorithm 1 and 3 respectively.

Algorithm 1: Linear Dyna with Feature Vectors for Planning being sampled from a Buffer

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 $\theta \in \mathbb{R}^m, F_a \in \mathbb{R}^m \times \mathbb{R}^m, b_a \in \mathbb{R}^m, \forall a \in \mathcal{A}$ arbitrarily (e.g., to zero)
Obtain initial state S, Compute feature vector ϕ from S.
while *still time to train* **do**
    $A \leftarrow \epsilon\text{-greedy}_a(b_a^\top \phi + \gamma F_a^\top \phi)$
    Take action A, receive reward R, next state S', and termination Z.
    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: Linear Dyna with Random Tabular Feature Vectors for Planning

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 $\theta \in \mathbb{R}^m, F_a \in \mathbb{R}^m \times \mathbb{R}^m, b_a \in \mathbb{R}^m, \forall a \in \mathcal{A}$ arbitrarily (e.g., to zero)
Obtain initial state S, Compute feature vector ϕ from S.
while *still time to train* **do**
    $A \leftarrow \epsilon\text{-greedy}_a(b_a^\top \phi + \gamma F_a^\top \phi)$
    Take action A, receive reward R, next state S', and termination Z.
    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 random tabular feature vector x.
        $\delta \leftarrow \max_a(b_a^\top x + \gamma \theta^\top F_a^\top x) - \theta^\top x$
        $\theta \leftarrow \theta + \alpha \delta x$
    end
    $\phi \leftarrow \phi'$
end

Algorithm 3: 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**

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

  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