Université de Montréal

Towards Adaptive Deep Model-Based Reinforcement Learning

par

Ali Rahimi-Kalahroudi

Département d’informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté en vue de l’obtention du grade de
Maître ès sciences (M.Sc.)
en Discipline

August 7, 2023

Résumé

L'une des principales caractéristiques comportementales utilisées en neurosciences afin de déterminer si le sujet d'étude — qu'il s'agisse d'un rongeur ou d'un humain — démontre un apprentissage basé sur un modèle (model-based) est une adaptation efficace aux *changements locaux* de l'environnement. Dans l'apprentissage par renforcement (RL), cependant, nous démontrons, en utilisant une version améliorée de la configuration d'adaptation au changement local (LoCA) récemment introduite, que les méthodes bien connues d'apprentissage par renforcement basées sur un modèle (MBRL) telles que PlaNet et DreamerV2 présentent un déficit dans leur capacité à s'adapter aux changements environnementaux locaux. En combinaison avec des travaux antérieurs qui ont fait une observation similaire sur l'autre méthode populaire basée sur un modèle, MuZero, une tendance semble émerger, suggérant que les méthodes MBRL profondes actuelles ont de sérieuses limites. Nous approfondissons les causes de ces mauvaises performances en identifiant les éléments qui nuisent au comportement adaptatif et en les reliant aux techniques sous-jacentes fréquemment utilisées dans la RL basée sur un modèle profond, à la fois en matière d'apprentissage du modèle mondial et de la routine de planification. Nos résultats démontrent qu'une exigence particulièrement difficile pour les méthodes MBRL profondes est qu'il est difficile d'atteindre un modèle mondial suffisamment précis dans toutes les parties pertinentes de l'espace d'état en raison de l'oubli catastrophique. Et tandis qu'un tampon de relecture peut atténuer les effets de l'oubli catastrophique, un tampon de relecture traditionnel premier-entré-premier-sorti empêche une adaptation efficace en raison du maintien de données obsolètes. Nous montrons qu'une variante conceptuellement simple de ce tampon de relecture traditionnel est capable de surmonter cette limitation. En supprimant uniquement les échantillons du tampon de la région locale des échantillons nouvellement observés, des modèles de monde profond peuvent être construits qui maintiennent leur précision dans l'espace d'état, tout en étant capables de s'adapter efficacement aux changements locaux de la fonction de récompense. Nous démontrons qu'en appliquant notre variation de tampon de relecture à une version profonde de la méthode Dyna classique, ainsi qu'à des méthodes récentes telles que PlaNet et DreamerV2, les méthodes basées sur des modèles profonds peuvent également s'adapter efficacement aux changements locaux de l'environnement.

Mots-clés: Apprentissage par renforcement basées sur un modèle, Apprentissage profond, Adaptation

Abstract

One of the key behavioral characteristics used in neuroscience to determine whether the subject of study—be it a rodent or a human—exhibits model-based learning is effective adaptation to *local changes* in the environment. In reinforcement learning (RL), however, we demonstrate, using an improved version of the recently introduced Local Change Adaptation (LoCA) setup, that well-known model-based reinforcement learning (MBRL) 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 MBRL 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, both in terms of learning the world model and the planning routine. Our findings demonstrate that one particularly challenging requirement for deep MBRL methods is that attaining a world model that is sufficiently accurate throughout relevant parts of the state-space is challenging due to catastrophic forgetting. And while a replay buffer can mitigate the effects of catastrophic forgetting, the traditional first-in-first-out replay buffer precludes effective adaptation due to maintaining stale data. We show that a conceptually simple variation of this traditional replay buffer is able to overcome this limitation. By removing only samples from the buffer from the local neighbourhood of the newly observed samples, deep world models can be built that maintain their accuracy across the state-space, while also being able to effectively adapt to local changes in the reward function. We demonstrate this by applying our replay-buffer variation to a deep version of the classical Dyna method, as well as to recent methods such as PlaNet and DreamerV2, demonstrating that deep model-based methods can adapt effectively as well to local changes in the environment.

Keywords: Model-Based Reinforcement Learning, Deep Learning, Adaptation

Contents

List of tables

List of figures

List of Abbreviations

AI	Artificial Intelligence
ML	Machine Learning
MDP	Markov Decision Process
RL	Reinforcement Learning
MBRL	Model-Based Reinforcement Learning
LoCA	Local Change Adaptation
MLP	Multi-layer Perceptron
CNN	Convolutional Neural Networks
RNN	Recurrent Neural Networks
LSTM	Long Short-Term Memory
GRU	Gated Recurrent Units

MC	Monte-Carlo
TD	Temporal difference
CEM	Cross-Entropy Method

Acknowledgement

This thesis stands as a testament to dedicated effort and hard work. Nevertheless, its existence owes a special debt of gratitude to the fantastic support and love received from specific exceptional individuals.

First and foremost, I would like to express my heartfelt gratitude to my supervisor, Sarath Chandar, whose exceptional guidance has been a cornerstone of my studies. I remain deeply appreciative of his trust in me and his support throughout these times.

I extend my sincere appreciation to Harm van Seijen, from whom I learned many valuable lessons and insights. I deeply appreciate his help throughout my academic pursuit.

I would also like to extend my heartfelt gratitude to Janarthanan Rajendran for his exceptional patience in guiding me and providing unwavering support. Additionally, I am grateful to my esteemed collaborators, Yi Wan and Ida Momennejad. Their fantastic contributions have played a crucial role in bringing this work to fruition.

I am grateful to all my fellow labmates from the ChandarLab, as I have had the privilege of not only collaborating with these exceptional individuals but also forming friendships with them. Specifically, I am thankful to Gonçalo Mordido, Doriane Olewicki, Darshan Patil, Hadi Nekoei, Mojtaba Faramarzi, and Maryam Hashemzadeh for their support.

I am thankful for the opportunity to be a member of Mila - Quebec AI Institute. My belief that pursuing graduate studies within Mila was the right path for me has been reaffirmed by my experiences and accomplishments. Looking back, I am confident that my decision was indeed accurate.

I would like to extend my acknowledgment to the Digital Research Alliance of Canada (the Alliance), Calcul Québec, and Mila for providing the computing resources used in this work.

I am grateful to Elsie Béchard-Pelletier, and Mahan Fathi, who made my life easier during my studies with their love and kindness.

Lastly, I am forever indebted to my family for their support. Regardless of the physical distance that separates us, the love I hold for my mother, father, and younger sister remains unchangeable, a sentiment I carry with me forever.

This thesis is lovingly dedicated to the memory of my grandfather, Reza Rahimi-Kalahroudi, whose absence is keenly felt. His belief in me and boundless support have left a memorable mark on me. I find comfort in the belief that he rests in peace by the divine will.

Chapter 1

Introduction

Autonomous sequential decision-making is one of the critical challenges in artificial intelligence (AI), which is commonly formulated as a Markov Decision Process (MDP) (Puterman, 2014). Although the definition of AI could be relative and may differ context by context, the importance of sequential decision-making cannot be overstated when it comes to real-world problems. Some important real-world problems that call for sequential decision-making include 1) self-driving cars and 2) treatment planning and personalized medicine in the healthcare industry, to name just two examples. Building artificial agents capable of solving the mentioned examples, on the one hand, reduces substantial human workloads. On the other hand, however, we need to be extremely careful in deploying such agents, as a single wrong decision could bring the worst outcome, such as a car crash or suggesting inappropriate medicine; both cases could potentially lead to irreversible damage to people's lives. That being said, we must not give up trying to develop such agents that can make our lives better. Moreover, we should work to address any potential drawbacks and provide dependable, effective algorithms for training the agents.

Planning (Russell, 2010; Bertsekas, 2012) and Reinforcement Learning (RL) (Sutton and Barto, 2018) are two key approaches to tackling MDP optimization. These two approaches are combined to form the field of study known as *"Model-Based Reinforcement Learning"* (MBRL). Formally, a model-based RL algorithm is any technique that employs a model of the environment (learned or oracle) and potentially uses the model to learn to approximate a global value or policy function. Moreover, MBRL algorithms incorporating neural networks in their design are called deep MBRL methods. Research in deep MBRL has gained momentum (Moerland et al., 2020; Wang et al., 2019) with the rise of deep learning (LeCun et al., 2015), where various successful neural network architectures and ways to optimize them have been introduced. The interest in creating efficient deep MBRL methods is hardly surprising, as model-based RL, which leverages estimates of the environment dynamics, could hold the key to some persistent challenges in deep reinforcement learning, such as sample efficiency and effective adaptation to changes in the environment.

Despite the growth of deep model-based RL, there are questions to be raised about the proper way to evaluate its progress. A common performance metric is sample efficiency (achieving good enough performance with a minimal number of interactions or samples from the environment) in a single task, which has several disadvantages. First, it conflates progress due to model-based RL with other factors, such as generalization and representation learning. More importantly, it is unclear whether model-based RL is always more sample efficient in a single-task setting than model-free RL (not using a model of the environment) because directly learning a policy is not necessarily slower than learning a model and planning with it. By contrast, it is arguable that model-based RL has a clear advantage when it comes to solving multiple tasks that share (most of) the dynamics.

In neuroscience, one of the critical behavioral traits for identifying if a subject of study—whether people or rodents—exhibits model-based learning is the effective adaptation to local changes in the environment (see Daw et al., 2011). This fact prompts the crucial question for artificial agents, which is whether or not they can exhibit the anticipated model-based behavior. In order to find the answer, we must be able to evaluate the MBRL methods' adaptation capabilities.

Inspired by the mentioned works in neuroscience, Van Seijen et al. (2020) developed the Local Change Adaptation (LoCA) setup to measure the agent's ability to adapt when the task changes. This approach 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., 2020), which achieves excellent sample efficiency on Atari, was unable to adapt effectively to a local change in the reward, even on simple tasks.

This thesis builds out this direction further. First, we improve the original LoCA setup, such that it is simpler, less sensitive to its hyperparameters and can be more easily applied to stochastic environments. Our improved 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 improved setup to the MuJoCo Reacher domain (Tassa et al., 2018) and use it to evaluate two continuous-control model-based methods, PlaNet (Hafner et al., 2019b) and DreamerV2 (Hafner et al., 2019a, 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 identify two failure modes that prohibit adaptivity. The first failure mode is 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. First, 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). We then perform experiments with a nonlinear (deep) version of our adaptive linear Dyna algorithm. Our analysis of these experiments reveals that there exists a challenging dilemma to overcome to design an adaptive deep model-based method. To better understand this dilemma, we take a closer look at the agent's training procedure and the way this procedure influences the agents' world model.

Specifically, for deep world models, the accuracy of predictions from the model of the environment across the state space is hard to achieve and maintain, even with sufficient exploration. The reason is that collected samples are strongly correlated and, at the final stages of learning, mostly come from states along the trajectory of the optimal policy. Due to catastrophic forgetting, the quality of the predictions further away from this trajectory quickly degrades. A common strategy to counter this is to use a replay buffer (Lin, 1992), a dataset of previous interactions with the environment often implemented with a first-in-first-out (FIFO) structure. By randomly sampling from a large replay buffer and using these samples to update the world model, the effects of catastrophic forgetting are greatly reduced. However, using the traditional FIFO replay buffer has the disadvantage that it hinders effective adaptation, as out-of-date samples interfere with the new data.

To address the challenge of catastrophic forgetting while also avoiding interference from out-of-date samples, we propose a variation of the traditional FIFO replay buffer. Instead of removing the oldest sample from the replay buffer once the buffer is full, the oldest sample in the *local neighbourhood* of the new sample is removed. This conceptually simple idea naturally leads to a replay buffer whose samples are approximately spread out equally across the space space, while local changes are accounted for quickly. Consequently, updating the deep world model with samples drawn randomly from this replay buffer results in a world model that is approximately accurate across the state-space at each moment in time. We call this replay buffer variation a LoFo (*Lo*cal *Fo*rgetting) replay buffer. One practical challenge to our proposed variation is that a locality-function needs to be learned that determines whether or not a sample from the replay buffers falls within the local neighborhood of a newly observed sample. We train this locality function using contrastive learning (Hadsell et al., 2006; Dosovitskiy et al., 2014; Wu et al., 2018) during the initial stages of learning, after which it is fixed and used as basis for the LoFo replay buffer.

We demonstrate the effectiveness of the LoFo replay buffer by combining it with our previous deep version of the classical Dyna method and measuring its adaptivity. We then test the limits of our approach by applying the same idea to both PlaNet and DreamerV2, which use world models based on recurrent networks. Experiments with these modified methods demonstrate that a LoFo replay buffer can substantially improve adaptivity of more advanced deep MBRL methods as well.

1.1. Contributions

The following are the key contributions of this thesis:

- We propose an improved version of the previously introduced LoCA setup (Van Seijen et al., 2020) so that it is simpler and less sensitive to its hyperparameters. Using this improved version, we identify four failure modes in the design of MBRL algorithms that precludes local adaptivity. The first two failure modes are linked to the planning routine, and the other two are unique to approximate model-based methods. (Wan et al., 2022)
- Using the insights from the failure modes, we design our adaptive linear Dyna algorithm that relies on linear function approximation and is based on the classical Dyna method. Our findings demonstrate that the implementation of the deep (non-linear) variant of our adaptive linear Dyna algorithm exhibits limitations in adapting to local changes in the environment. Therefore, we highlight the considerable challenge of overcoming the failure modes in deep MBRL methods. (Wan et al., 2022)
- Lastly, We propose the Local Forgetting (LoFo) replay buffer that instead of removing the oldest sample from the replay buffer once it is full (first-in-first-our scenario), the oldest sample in the local neighbourhood of the incoming sample is removed. We demonstrate the effectiveness of the LoFo replay buffer through various experiments with our deep version of the classical Dyna method, as well as the PlaNet and DreamerV2 methods. Our results show that the LoFo replay buffer can substantially improve the adaptivity of deep MBRL methods regarding the local changes to the reward function. (Rahimi-Kalahroudi et al., 2023)

The main contributions of this thesis stem from our two published works:

- Towards Evaluating Adaptivity of Model-Based Reinforcement Learning Methods (Wan et al., 2022)
  - Authors: Yi Wan*, **Ali Rahimi-Kalahroudi***[1], Janarthanan Rajendran, Ida Momennejad, Sarath Chandar, and Harm van Seijen.
  - Published as a conference paper at the International Conference on Machine Learning, **ICML 2022**.
  - Also presented in the ICLR 2022 workshop on Agent Learning in Open-Endedness (*Spotlight*) and RLDM 2022.
  - Personal Contributions: I co-led the project with Yi. All the project members participated in the weekly meetings to discuss the experiments and their results and to work on the project direction and plan. I was mainly responsible for Section 4. This means that I designed the ReacherLoCA domain, came up with the code and ran all the experiments regarding the DreamerV2 and the PlaNet methods, and also initially wrote Section 4. Harm designed and ran all the experiments in Section 3, and lastly, Yi took

[1]Shared first authorship.

care of the coding and running of all other experiments, mainly for Section 5. Yi led the writing while Harm, and I helped him with this task. Sarath, Ida, and Jana also provided valuable feedback for the writing. Both Harm and Sarath supervised me during this project via one-on-one meetings.

- Replay Buffer with Local Forgetting for Adapting to Local Environment Changes in Deep Model-Based Reinforcement Learning (Rahimi-Kalahroudi et al., 2023)
  - Authors: **Ali Rahimi-Kalahroudi**, Janarthanan Rajendran, Ida Momennejad, Harm van Seijen, and Sarath Chandar.
  - Published as a conference paper at the Conference on Lifelong Learning Agents, **CoLLAs 2023** ***(Oral Presentation)***.
  - Also presented in the NeurIPS 2022 workshop on Deep Reinforcement Learning Workshop.
  - Personal Contributions: I led the project, took care of the code, and ran all the experiments. All the project members participated in the weekly meetings to discuss the experiments and their results and to work on the project direction and plan. I also led the writing and came up with the initial draft of the paper. Jana substantially worked on and improved the initial draft and also, Harm helped with the writing by editing the manuscript. Sarath and Ida also provided their feedback for the writing. Both Harm and Sarath supervised me during this project via one-on-one meetings.

1.2. Thesis Outline

This thesis is organized in the following way. First, we provide a comprehensive background on model-based reinforcement learning, various notions, and algorithms we used throughout this work in Chapter 2. Then, in Chapter 3, we discuss the central setup for evaluating if an MBRL method is locally adaptive or not. Chapter 4 presents the failure design modes that prohibit effective adaptation in MBRL methods. We then design an adaptive MBRL algorithm with linear function approximation in Chapter 5. And in Chapter 6, we first analyze that adopting our adaptive method for nonlinear function approximation fails to adapt to local changes in the environment, and then we present the LoFo buffer. In this chapter, we show that employing the LoFo buffer improves the adaptation capability of more advanced MBRL methods. Lastly, Chapter 7 concludes this thesis by discussing current limitations and possible future directions for our works.

Chapter 2

Background

In this chapter, we aim to present a comprehensive background on model-based reinforcement learning and the algorithms we used throughout this thesis. Furthermore, we give a concise and to-the-point overview of each key concept and offer various relevant references for in-depth understanding.

We start with an introduction to Machine Learning in Section 2.1 and go over a few concepts we regularly use in our work. Then, we present an overview of Deep Learning in Section 2.2 since it has revolutionized the machine learning field and is the basis of every nonlinear function approximation in this work. We then move on to an introduction of Reinforcement Learning–problem setup, and type of solutions and methods, in Seciton 2.3. And finally, we conclude this chapter with an explanation of Model-Based Reinforcement Learning and provide an overview of the three main algorithms that we have used in this work in Section 2.4.

2.1. Machine Learning

Machine learning (ML) (Bishop and Nasrabadi, 2006) is a field of artificial intelligence concerned with building algorithms that can automatically improve from experience. It enables computers to learn from data and make predictions or take actions without being explicitly programmed. A more formal and common definition of machine learning is as follows: *"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E"* (Mitchell and Mitchell, 1997).

The goal of machine learning is to develop algorithms that can learn from data and generalize well to new, unseen data. There are three main types of machine learning paradigms: (1) supervised learning (2) unsupervised learning (3) reinforcement learning.

In **Supervised Learning**, the algorithm is trained on labeled data. The objective of the algorithm is to learn a mapping between the inputs and outputs that can then be used to make predictions on new, unseen data. Regression and classification are two widely recognized forms of supervised learning, with regression aiming to forecast continuous outputs, while classification aims to predict categorical outputs.

Unsupervised Learning, on the other hand, involves learning from data without any labeled output. The algorithm tries to find patterns, structures, or underlying distributions in the data and group similar examples together. Two common examples of unsupervised learning techniques include clustering, which aims to group similar instances, and dimensionality reduction, which seeks to simplify the complexity of data while maintaining its critical structure.

Reinforcement Learning is another machine learning paradigm that focuses on decision-making problems, where an agent interacts with its environment and receives rewards or penalties for its actions. The goal of the agent is to maximize its cumulative reward over time. Reinforcement learning is used in problems such as gaming, robotics, and autonomous vehicles.

In addition to the traditional forms of machine learning algorithms, there exist several subcategories, such as **Semi-Supervised Learning**, which leverages both labeled and unlabeled data, and **Self-Supervised Learning**, a form of unsupervised learning in which the supervision is implicitly provided via the unlabelled data itself.

Machine learning is steadily gaining significance across numerous industries, ranging from healthcare to finance, and has facilitated noteworthy advancements in previously challenging areas. However, despite its numerous advantages, the application of machine learning poses certain challenges. Hence, machine learning is a vibrant research area that has made remarkable strides in developing new algorithms and techniques, such as decision trees, random forests, support vector machines, neural networks, and deep learning. Each algorithm boasts unique strengths and limitations and is most effective in tackling specific types of problems.

The process of using a machine learning algorithm usually includes two steps: 1) training the machine learning model and 2) evaluating its performance. In order to train a machine learning model, several key concepts should be considered to ensure that the model is accurate and reliable. For the rest of this section, we will provide various definitions and notations commonly used in machine learning literature.

2.1.1. Train and Test Data

The data used to train a machine learning model and the data used to evaluate its performance are often referred to as training and test dataset, respectively. It is essential that both of these datasets lie within the *i.i.d* assumption, which states that 1) each data point in a dataset is independent of other data points, and 2) the underlying distribution of data in the train and test datasets are identical–each data point is drawn with the same probability distribution. It is also worth noting that it is important to have a good balance between the size of the train and test datasets, with a large enough training dataset to fit the model and a sufficiently sized test dataset to evaluate its performance.

2.1.2. Hyperparameters and Learnable Parameters

Hyperparameters are parameters set before training a machine learning model and are not learned from the data. Examples of hyperparameters include:

- The learning rate in a gradient-based optimization process.
- The number of hidden layers in a neural network.
- The magnitude of the regularization term.

The choice of hyperparameters can significantly impact the model's performance, and finding the best values for the hyperparameters is often an iterative process that requires experimentation and tuning. The process of tuning hyperparameters is usually evaluated by a dataset that differs from the train and test datasets called validation.

The learnable parameters of a machine learning model, on the other hand, are commonly referred to as the model's parameters. They include any parameter that can be changed and adjusted during the training so that the model can fit the training dataset. An example of such parameters is a neural network's weights and biases.

2.1.3. Capacity

In machine learning, capacity refers to the ability of a model to learn and fit the underlying patterns in the data. For example, a model with high capacity is able to fit complex data distributions, while a model with low capacity is limited in its ability to fit complex data. The capacity of a model is typically controlled by the number of (learnable) parameters in the model.

Overfitting and underfitting are two common challenges in machine learning. Overfitting occurs when the model has a high capacity and fits the training data too closely, capturing the true underlying relationships between the data and the noise. This results in a model with high accuracy on the training data but poor performance on the test data. Techniques such as regularization, early stopping, and cross-validation can be used to avoid overfitting.

On the other hand, underfitting occurs when the model is too simple and unable to capture the underlying patterns in the data. This results in a model with poor accuracy on both the training and test data. Increasing the capacity of the model is a way to avoid underfitting.

2.1.4. Loss Functions

Loss functions are mathematical representations of the error or difference between the predictions made by the machine learning model and the actual outcomes. The loss function guides the training process by providing a measure of the model's performance. The goal of training a machine learning model is to minimize the loss function, which is typically achieved through gradient-based optimization algorithms.

2.1.5. Gradient-based Optimization

Gradient-based optimization is a family of algorithms used to find the minimum of a loss function in order to train a machine learning model. These algorithms work by iteratively adjusting the model's parameters in the direction of the negative gradient of the loss function. The gradient represents the rate of change of the loss function with respect to the parameters, and the optimization algorithms use this information to make small updates to the parameters in each iteration until the minimum loss function is found. Some popular gradient-based optimization algorithms include stochastic gradient descent (SGD), mini-batch gradient descent, and Adam (Ruder, 2016). Unless otherwise mentioned, we utilize gradient-based optimization algorithms to optimize various models in this thesis.

2.2. Deep Learning

Deep learning (LeCun et al., 2015; Goodfellow et al., 2016) is a subfield of machine learning that uses neural networks with multiple layers to model and solve problems. Neural networks are inspired by the structure and function of the human brain and are designed to learn complex patterns and representations from large amounts of data.

A neural network consists of interconnected nodes, called artificial neurons, that process information and communicate with each other. These neurons can form sequential layers, from the first (input) layer to the last (output) layer. The inputs to the network are fed through the input layer, and the output layer produces the network's outputs. Multiple hidden layers exist between the input and output layers, each containing many artificial neurons that transform their input in a series of non-linear transformations to produce a final output. Each node in a layer takes in input from the nodes in the previous layer, performs a weighted sum of the inputs, and then applies a non-linear activation function to produce the output.

At a high level, deep learning aims to train neural networks to learn useful representations of data that can be used for tasks such as classification, regression, and prediction. This is achieved by feeding large amounts of data into the network and updating the connections between the nodes to minimize a loss function. The network is typically trained using a gradient-based optimization algorithm, such as stochastic gradient descent, that iteratively adjusts the weights of the connections based on the error between the predicted output and the true output. This process is known as backpropagation (Rumelhart et al., 1986), which involves computing the gradient of the loss function with respect to the network's weights and then changing the weights in the opposite direction of the gradient.

There are many different types of neural networks, each with its own unique architecture and purpose. The simplest type is called a **feed-forward neural network** (Goodfellow et al., 2016, Chapter 6) (or multi-layer perceptron–MLP). In this type, the flow of information moves in one direction, from the input to the output layer. Furthermore, each neuron in a specific layer is connected to all the neurons in the previous layer. A key component of feed-forward neural networks is that a nonlinear activation function is applied to the output of each neuron. Without this activation function, the feed-forward neural network would simply perform some linear transformation of the input data; hence could be represented by a two-layer neural network. Furthermore, the activation function and increasing the number of in-between (hidden) layers help feed-forward neural networks learn various abstract representations of the input data, making them solve complex problems.

The other popular type of neural networks are called **Convolutional neural networks (CNNs)** (Goodfellow et al., 2016, Chapter 9), which are particularly effective when it comes to working with images or various frames of videos. CNNs are inspired by the structure of the visual cortex in animals, which is said to use a series of layers to process visual information. Empirical evidence has shown that in CNNs, the first layer usually extracts low-level input image features, such as edges and corners. And the layers towards the end would come up with higher-level features such as shapes and objects. The key innovation of CNNs is the use of convolutional layers, which apply a set of learnable filters (kernels) to the input data to extract features. Each filter is a small matrix of weights that is convolved over the input data, producing a set of feature maps that capture different aspects of the input. Pooling layers are also commonly used in CNNs, which downsample the feature maps to reduce their size and improve their translation invariance.

Finally, the last type of neural networks that we cover are called **Recurrent neural networks (RNNs)** (Goodfellow et al., 2016, Chapter 10). RNNs can process sequences of data, such as time series, natural language, texts, or speech. The basic building block of an RNN is the recurrent layers. A recurrent layer produces two representations: an output and a hidden state. The hidden state captures information about the previous inputs to the RNN. That being said, for a given timestep, the input data and the hidden state of the prior timestep are given to the recurrent layer. Then it produces the output of the current timestep and also the hidden state for the next timestep. This way, RNNs can maintain information about past timesteps. Hence, they can process the input data in various orders, unlike feed-forward neural networks, which process the input data in a fixed order. Long short-term memory (LSTM) (Hochreiter and Schmidhuber, 1997) and gated recurrent units (GRU) (Chung et al., 2014) are two popular types of RNNs.

Overall, deep learning is an active area of research and has revolutionized many fields, and its applications are constantly expanding. For example, it has been used to develop self-driving cars, detect diseases from medical images, generate realistic photos and videos, and even play games at a superhuman level.

2.3. Reinforcement Learning

Reinforcement learning (RL) (Sutton and Barto, 2018) is a machine learning paradigm focusing on decision-making problems where an agent interacts with an environment and receives rewards or penalties as outcomes of its actions.

A Markov Decision Process (MDP) (Puterman, 2014; Sutton and Barto, 2018, Chapter 3) is a mathematical framework used for modeling decision-making processes under uncertainty. Formally, an MDP is defined as a 5-tuple $(\mathbb{S}, \mathbb{A}, \mathcal{P}, \mathcal{R}, \gamma)$ where:

- $\mathbb{S}$ is the set of all possible states in the environment.
- $\mathbb{A}$ is the set of all possible actions that the agent can take in each state.
- $\mathcal{P}$ is the state transition probability function which specifies the probability of transitioning to a new state $s' \in \mathbb{S}$ when taking an action $a \in \mathbb{A}$ in the current state $s \in \mathbb{S}$. It is defined as $\mathcal{P}(s, a, s') = Pr(\mathbf{S}_{t+1} = s' | \mathbf{S}_t = s, \mathbf{A}_t = a)$ where $\mathbf{S}_t$ is the state and $\mathbf{A}_t$ is the action taken at time t.
- $\mathcal{R} : \mathbb{S} \times \mathbb{A} \rightarrow \mathbb{R}$ is the reward function that specifies the scalar reward received by the agent after taking an action $a \in \mathbb{A}$ in state $s \in \mathbb{S}$. It is defined as $\mathcal{R}(s,a,s')$ where $s' \in \mathbb{S}$ is the resulting next state after taking the action a in state s.
- $0 \leq \gamma \leq 1$ is the discount factor that determines the importance of future rewards relative to immediate rewards.

Regarding the definition of MDPs above, we can say that the agent interacts with the environment through a sequence of states, actions, and rewards. This means that at a particular timestep $0 \leq t$, the agent is presented with a state $s_t \in \mathbb{S}$ from the environment. The agent then takes action $a_t \in \mathbb{A}$, and the environment would yield the reward $r_{t+1} \in \mathbb{R}$ and transitions into a new state $s_{t+1} \in \mathbb{S}$. We can show the mentioned sequence as $\tau =< s_0,a_0,r_1, \dots s_T,a_T,r_{T+1} >$, which is usually referred to as a trajectory or an episode.

All states in an MDP follow the *Markov* property which states that the next state of the environment depends only on the current state. More formally, the probability distribution of the next state and reward depends only on the present state and the action taken in that state and not on any previous states or actions:

$$Pr(s_{t+1}|s_0,a_0,s_1,a_1, \dots s_t, a_t) = Pr(s_{t+1}|s_t, a_t). \tag{2.3.1}$$

A finite MDP is an MDP in which the sets of states and actions ($\mathbb{S}$ and $\mathbb{A}$) are finite. And finally, the initial state $s_0 \in \mathbb{S}$ is usually sampled from a distribution referred to as the initial-state distribution ρ_0.

In order to talk about reinforcement learning's objective, we need to define a few more notions. First, for a given MDP, if a single trajectory never finishes, we call it an infinite horizon; otherwise, it is a finite horizon MDP. The **return** of an episode τ is shown as $G(\tau)$ which is the discounted summation over the received rewards:

$$G(\tau) = \sum_{t=0}^{\infty} \gamma^t r_{t+1} \tag{2.3.2}$$

If we have a finite horizon MDP or we want to have the return up to a certain point in time, we can have it as:

$$G(\tau_{0:T}) = \sum_{t=0}^{T-1} \gamma^t r_{t+1} \tag{2.3.3}$$

In an MDP, a **policy** is a function that maps states to actions. The policy $\pi : \mathbb{S} \rightarrow \mathbb{A}$ tells what action to take in each state, and it could be either deterministic or stochastic. A deterministic policy specifies a unique action for each state. In contrast, a stochastic policy specifies a probability distribution over the set of actions in each state, and we need to sample actions according to the given distribution ($a \sim \pi(.|s)$ where $s \in \mathbb{S}$).

We can now define the reinforcement learning's objective: learning a policy π^* (*optimal policy*) that maximizes its expected return. Assuming that we are dealing with a finite-horizon MDP with length T, and $Pr(\tau)$ denotes the probability of the trajectory τ, we can write this objective as:

$$\pi^* = \arg\max_{\pi} J(\pi) \tag{2.3.4}$$

where,

$$J(\pi) = \mathbb{E}_{\tau\sim\pi}[G(\tau)] = \int_{\tau} Pr(\tau)G(\tau) \tag{2.3.5}$$

$$= \int_{\tau} \rho_0(s_0) \prod_{t=0}^{T-1} \mathcal{P}(s_t,a_t,s_{t+1})\pi(a_t|s_t) \times G(\tau) \tag{2.3.6}$$

Another way to formalize the reinforcement learning's objective is to look at the expected return conditioned on the starting state. In order to do that, we first need to define the value functions in an MDP. The **value function** is defined as the expected total reward that the agent can achieve from a given state or state-action pair under a particular policy. Let's first define $G_t(\tau)$ as the return starting from specific timestep $0 \leq t$, we can write it as follows:

$$G_t(\tau) = \sum_{i=0}^{\infty} \gamma^i r_{i+t+1} \tag{2.3.7}$$

Now we can define the state-value of a state s under the policy π as the expected return if the agent is in the state s at timestep t ($\mathbf{S}_t = s$) and takes actions according to π. Mathematically, we can write:

$$V^{\pi}(s) = \mathbb{E}_{\tau\sim\pi}[G_t(\tau)|\mathbf{S}_t = s] \tag{2.3.8}$$

$V^{\pi}(s)$ is referred to as the state-value function or simply the value function of the MDP. It gives an estimate of the long-term expected reward the agent can achieve by following policy π from state s.

We can also condition the total expected return on the state-action pair and form the Q-value function (also known as the action-value function). The Q-value function is defined as the expected total reward the agent can achieve from taking action a in state s under policy π. Mathematically we can write:

$$Q^{\pi}(s,a) = \mathbb{E}_{\tau\sim\pi}[G_t(\tau)|\mathbf{S}_t = s, \mathbf{A}_t = a] \tag{2.3.9}$$

Based on Equations 2.3.8 and 2.3.9, we can mathematically derive the underlying relation between the state-value and action-value functions as follows:

$$V^{\pi}(s) = \mathbb{E}_{a\sim\pi}[Q^{\pi}(s,a)] = \sum_{a\in A} \pi(a|s)Q^{\pi}(s,a) \tag{2.3.10}$$

As mentioned previously, reinforcement learning's objective is to maximize the agent's expected return. And therefore, we can view this problem through the lens of the optimal state-value and action-value functions. The optimal state-value and action-value functions and their relations can be expressed as follows:

$$\begin{aligned} V^*(s) &= \max_{\pi} V^{\pi}(s) \\ Q^*(s,a) &= \max_{\pi} Q^{\pi}(s,a) \\ V^*(s) &= \max_{a} Q^*(s,a) \end{aligned} \tag{2.3.11}$$

So, the optimal policy can be derived from the state-value or action-value function:

$$\pi^*(s) = \arg\max_{\pi} V^{\pi}(s) = \arg\max_{\pi} Q^{\pi}(s,a) \tag{2.3.12}$$

Notably, the policy and value functions can be modeled or approximated by neural networks, and the field of deep RL (Arulkumaran et al., 2017) is formed this way.

Having established the objective of RL, we can now discuss the notable distinguishing features of RL algorithms that classify them into different classes. We only focus on three main characteristics: (1) Whether an agent uses a model of environment or not (2) What type of data the agent uses to find the optimal policy (3) What strategy the agent uses to find the optimal policy.

For the first characteristic, we typically categorize RL methods into two classes, **model-free** and **model-based**. Model-based RL methods learn or have access to a model of the environment, allowing the agent to gather experiences based on simulation over the known/learned model; this thesis focuses on model-based RL methods, and we give an introduction to them in Section 2.4. On the other hand, model-free methods do not use or learn a model of the environment and instead rely solely on trial-and-error experiences to learn the optimal policy. That being said, model-free methods are simpler and more scalable than model-based methods because there is no need to learn a model of the environment, but they require more samples to converge since they need a lot of interaction between the agent and the real environment.

The second characteristic mentioned above categorizes RL methods into **on-policy** and **off-policy** algorithms. On-policy learning is considered a type of RL algorithm in which the agent learns by interacting with the environment and updating its policy based on the *current* policy. In other words, the agent learns to optimize the policy it is currently using to make decisions. This means that the agent explores the environment by taking actions according to its current policy and learns from the rewards it receives. The policy update is done by taking into account the most recent experiences of the agent. Some examples of the on-policy methods include Monte Carlo Control, SARSA, and Actor-Critic methods.

In off-policy learning, however, the agent learns by gathering experiences from the environment with a different policy than the one it uses for decision-making. In other words, the agent learns to optimize a policy that is different from the one it currently uses to make decisions (*behavior* policy). Thus, the agent can explore the environment using a more explorative policy while still learning from the rewards it receives. The policy update is done by taking into account the experiences collected using the behavior policy. We can name Q-Learning, Expected SARSA, and Deep Q-Networks (Mnih et al., 2013) as some examples of off-policy methods.

Lastly, the third characteristic we mentioned above identifies the agent's strategy to find the optimal policy. These strategies divide RL methods into three classes. The first category comprises **value-based methods**, in which the agent tries to find the optimal value functions. The optimal policy is then given by acting greedily based on the value functions according to equation 2.3.12. The second class includes **policy-based methods**, in which the optimal policy is computed directly. And the third class is called **actor-critic methods**. These methods are a combination of value-based and policy-based methods. For the rest of this section, we focus on these classes and give a summary of a few important RL methods.

2.3.1. Value-Based Methods

Value-based methods in reinforcement learning aim to learn the optimal value functions (i.e., finding either the $V^*(s)$ or the $Q^*(s,a)$). So, these methods try to find the expected cumulative reward of being in a particular state and following a specific action. Hence, the agent can act greedily according to equation 2.3.12 and form its optimal policy. The followings are two prominent classes of value-based RL methods:

Monte-Carlo (MC) methods (Sutton and Barto, 2018, Chapter 5) are a class of value-based RL algorithms based on averaging sample returns. The main idea of MC methods is to use many episodes of interaction between an agent and its environment and then use these interactions to estimate the value of states $V(s)$, or state-action pairs $Q(s,a)$. The general steps involved in MC methods are as follows: (1) Generate a trajectory of experience. (2) Compute the empirical return for each state or state-action pair in the trajectory. (3) Update the value function using the empirical returns. (4) Improve the policy by choosing the action that has the highest return for each state. (5) Repeat steps 1-4 for multiple trajectories until the optimal policy is found.

MC methods have several advantages in RL. First, they can be used for both on-policy and off-policy learning. Second, They can learn directly from experience without requiring a model of the environment (model-free), and they can handle non-Markovian environments where the state does not fully capture all relevant information about the history of the agent's interaction with the environment. Additionally, it is worth noting that the MC value estimates converge to the optimal value functions as the number of state visitations goes to infinity. MC methods, on the other hand, can be computationally expensive because they require many episodes to converge, and they can suffer from high variance in the estimates of value functions, especially in environments with long episodes, large state-spaces, or sparse rewards. Some popular MC methods in RL include Monte Carlo Control (Sutton and Barto, 2018, Section 5.3) and First-Visit Monte Carlo Policy Evaluation (Sutton and Barto, 2018, Section 5.4).

Temporal difference (TD) learning methods (Sutton and Barto, 2018, Chapter 6) are the other class of value-based RL methods that we give an overview of. TD Learning methods learn by updating estimates of the value functions based on the difference between predicted and actual rewards observed. They are in the same spirit as MC methods, but the main difference between these two classes is that TD Learning can learn from incomplete episodes. TD Learning's update rule for the timestep t in a given trajectory could be as follows:

$$V(s_t) \leftarrow V(s_t) + \alpha(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)) \tag{2.3.13}$$

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \tag{2.3.14}$$

Where G_t is the return from timestep t onward, and $\alpha \in (0, 1)$ is the step-size (also known as the learning rate).

There are two main types of TD learning methods: on-policy and off-policy. On-policy TD learning methods update the value function based on the same policy used to generate the samples (current policy). Off-policy TD learning methods update the value function based on a different policy than the one used to generate the samples (behavior policy). Overall, TD-learning methods are widely used in RL due to their simplicity, efficiency, and ability to handle stochastic environments with partial observability and large state-spaces. However, they can be sensitive to hyper-parameters and slow to converge. Additionally, while TD Learning methods have lower variance compared to MC methods, their estimates could be more biased.

The following are some popular on-policy TD Learning methods:

(1) **TD(0)** (Sutton and Barto, 2018, Section 6.1) is a simple TD Learning algorithm that updates the value function based on the difference between the current estimate and the expected value of the next state.

(2) **SARSA** (State-Action-Reward-State-Action) (Sutton and Barto, 2018, Section 6.4) is another model-free TD-learning algorithm that learns the state-action value function. SARSA updates the Q-values based on the difference between the predicted and actual rewards received when taking the current action and following the current policy. It is guaranteed to converge to the optimal Q-values under certain conditions but may converge to sub-optimal solutions in other cases.

(3) **TD(λ)** (Sutton and Barto, 2018, Section 12.2) is a generalization of TD(0) that use eligibility traces to update the values of states or actions. Eligibility traces keep track of the recent history of state-action pairs and weight their contributions to the updates based on their recency and importance. TD(λ) tries to balance the trade-off between short-term and long-term rewards and can converge faster than other TD learning methods.

In addition, some common off-policy TD Learning methods include:

(1) **Q-Learning** (Sutton and Barto, 2018, Section 6.5) is a model-free TD-learning algorithm that learns the optimal action-value function. It updates the Q-values based on the difference between the current estimate and the *target* value, which is the maximum Q-value for the next state regardless of the policy used to generate the samples. The update rule is as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_{t+1} + \gamma \max_{a \in \mathbb{A}} Q(s_{t+1}, a) - Q(s_t, a_t)) \tag{2.3.15}$$

(2) **Expected SARSA** (Sutton and Barto, 2018, Section 6.6) is a variant of SARSA that uses an off-policy approach to estimate the action-value function, but uses an expected value over all possible actions instead of the actual action taken in the next state according to the current policy.

2.3.2. Policy-Based Methods

Policy-based methods are another family of RL algorithms that parametrize the policy with some vector of parameters such as θ and then try to learn the $\pi(a|s;\theta)$ (or $\pi_\theta(a|s)$) through either directly optimizing the expected cumulative reward, $J(\theta) = J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[G_t(\tau)]$, using some form of gradient ascent optimization–a method that iteratively updates the parameters in the direction of the gradient, or indirectly maximizing local approximations of $J(\pi_\theta)$.

The **policy gradient theorem** provides a way to compute the gradient of the $J(\theta)$ with respect to θ, which is the policy parameter. More formally, this gradient is given by:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a_t|s_t) Q^{\pi_\theta}(s_t, a_t)] \tag{2.3.16}$$

where $Q^{\pi_\theta}(s_t,a_t)$ is the state-action value function under the policy π_θ.

Furthermore, a simple update rule for the policy parameters that move the parameters in the direction of the gradient could be written as follows:

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\theta) \tag{2.3.17}$$

Where α is the step-size (or the learning rate) that determines the size of the update, and k refers to the number of steps for the θ updates.

The policy gradient theorem (Equation 2.3.16) lays the necessary foundation for various policy-based methods. For instance, the Vanilla Policy Gradient algorithm, also known as REINFORCE (Williams, 1992), calculates an estimate of $Q^{\pi_\theta}(s_t, a_t)$ using MC methods. More generally, any other return G_t could be incorporated in Equation 2.3.16. An example of such a case is a technique called policy gradient with baseline. In this technique, an arbitrary baseline $b(s)$ that is independent of action is subtracted from the (estimated) return:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a_t|s_t)(G_t - b(s_t))] \tag{2.3.18}$$

Using a baseline b helps reduce the variance of the gradient estimates, making the learning more stable and efficient. Common choices for the baseline include the state-value function, which estimates the expected return starting from a given state, or a constant value, such as the average return observed so far. By subtracting the baseline, the algorithm focuses on the advantage of each action over the expected return rather than the absolute value of the return.

2.3.3. Actor-Critic Methods

Actor-critic methods are the last family of reinforcement learning methods we will cover. These methods combine the benefits of both value-based and policy-based algorithms. They try to simultaneously learn an **actor**–optimal policy, and a **critic**–estimates of the value function. In other words, the actor is responsible for selecting actions based on the current policy, while the critic evaluates the quality or value of the states and actions. So, in some sense, the critic provides feedback to the actor, and they interact with each other. This interaction allows for more efficient and stable learning compared to using value-based or policy-based methods separately. Some popular actor-critic methods include Advantage Actor-Critic (A2C) and Asynchronous Advantage Actor-Critic (A3C) (Mnih et al., 2016).

2.4. Model-Based Reinforcement Learning

The combination of RL and Planning (Russell, 2010; Bertsekas, 2012) forms the field of study knowns as *"Model-Based Reinforcement Learning"* (MBRL) (Moerland et al., 2020). Formally, a model-based RL algorithm is any technique that employs a model of the environment (learned or oracle), also known as a world model, and potentially uses the world model to learn to approximate a global value or policy function. Hence MBRL agents leverage the acquired knowledge about the environment's dynamics using the world model to plan and simulate potential actions before executing them in the real environment.

At its core, model-based RL involves two key components: (1) A world model, and (2) A planning routine. The world model represents the agent's understanding of how the environment behaves. It captures the state transition probability function $\mathcal{P}$ or simply the transition dynamics, which describe how the state of the environment evolves in response to the agent's actions, and the reward function $\mathcal{R}$. And the planning routine, also known as the model-based control algorithm, utilizes the world model to learn the optimal policy.

There are broadly two cases for a model-based RL agent considering its world model. Either a known world model is given to the agent, and it directly uses it for planning and learning the optimal policy, or the agent learns the world model while it interacts with the environment. Dynamic Programming (Bellman, 1966) and AlphaZero (Silver et al., 2018) are two well-known examples of the former case. However, in many real-world scenarios, the transition dynamics and the reward function of the MDP are not only unknown but also hard to determine. So, we focus on the latter case, where the agent tries to learn the world model on top of learning the policy. And while the world model is being trained, the planning routine could be a kind of model-free optimization (Kaiser et al., 2019) or using the world model to learn the policy in imagination (Ha and Schmidhuber, 2018).

One of the most straightforward key advantages of model-based RL is the premise of using limited data. Instead of relying only on interactions with the environment and trials and errors, which is the case with model-free methods, the agent can benefit from its learned model to explore different options and evaluate them in simulations. However, learning an accurate model has been very challenging, especially for long-horizon planning and complicated environments, such as those with high-dimensional state-space. Moreover, the mentioned inaccuracy in the model would also lead to poor planning and suboptimal policies.

Recent advancements, such as incorporating deep learning in the MBRL methods, have shown a significant improvement towards the mentioned issue (Luo et al., 2022; Russell, 2010; Bertsekas, 2012). Where deep MBRL agents (Hafner et al., 2019b,a, 2020; Schrittwieser et al., 2020) jointly learn a world model while solving complex visual tasks such as atari games or various locomotion tasks. The outcomes from these agents have not only improved the performance on single tasks but also shown a substantially higher sample efficiency compared to other state-of-the-art model-free methods.

Despite the growth of deep model-based RL, there are questions to be raised about the proper way to evaluate its progress. For example, only looking at the sample-efficiency metric in a single task 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 model-based RL is always more sample efficient in a single-task setting than model-free RL because directly learning a policy is not necessarily slower than learning a model and planning with it. By contrast, it is arguable that model-based RL has a clear advantage when solving multiple tasks that share (most of) the dynamics. Therefore, one can argue that an effective adaptation to certain forms of non-stationary could also be another key advantage of model-based RL. This argument could be backed up by various works in neuroscience (e.g., see Daw et al. (2011)), which states that the core capability that sets model-based learning apart from the model-free is the effective propagation of newly observed reward or transition information.

This thesis focuses on measuring to what degree popular model-based RL methods could adapt to local changes in the environment by using and improving a suitable toolbox and how further improvements could be made on top of these methods. However, before going into the details, we finish this section by introducing the three main MBRL methods we considered in our work.

2.4.1. Dyna

Dyna architecture proposed in Sutton (1990, 1991) has created a class of model-based RL algorithms called Dyna-style methods (e.g., see Sutton et al. (2012)). These methods utilize world model learning and planning together with direct interactions in the environment to train the agent. The fundamental idea behind Dyna-style methods is using a trainable world model to generate simulated or imaginary experiences for the agent. If the world model could approximate the dynamics of the environment well enough, then the agent could use the simulated data as additional feedback for its training procedure and potentially improve its performance and accelerate learning.

There are various components to Dyna-style methods. The first step is to train a world model of the environment. The world model's purpose is usually to predict the next state and reward given the current state and action. This model can be of different forms, such as deterministic or stochastic function approximators. The second step is the planning routine, which uses the model to generate simulated experiences from the environment. This procedure is possible due to the definition of the world model since it can predict the possible next state and reward given some arbitrary state and action. The agent's next step is to possibly combine the imaginary data with the real data gathered from the environment and use a conventional RL algorithm, for example, Q-Learning, to find the optimal policy. And lastly, the agent must be able to refine its world model as it interacts with the environment. This ensures that the world model can accurately predict the dynamics of the environment as the agent visits new states or chooses exploratory actions. The agent performs the steps mentioned above periodically, and depending on the specification of the method, parts of the steps could vary; for example, the agent may only rely on the simulated data to find the optimal policy, etc. In addition, the world model's accuracy plays a critical role in the effectiveness of these methods. If the world model's predictions are inaccurate, then the simulated data are not aligned with the environment, and therefore, the planning routine would fail to find the optimal policy.

In this thesis, we have used a Dyna-style model-based RL method that incorporates Q-Learning for its planning, both with linear and non-linear (deep) function approximation. The complete algorithms for these two methods are presented in Algorithm 5.1.1 and 6.1.1, and further discussions are presented in their respective sections.

2.4.2. PlaNet

PlanNet (Hafner et al., 2019b) is a deep model-based RL method that learns a latent dynamics model given visual inputs. It relies on a compact sequence of latent states using a recurrent unit of encoded images. This means that instead of focusing on one image and predicting the other, the prediction happens on the latent state forward. Furthermore, each latent state is responsible for predicting the image and reward at each step. The benefit of encoding images into a latent space is that the agent may be able to learn more useful abstract representations than just the next pixels to make it easier to predict the future.

In general, there are two main steps for training the PlaNet agent. First, the agent uses a collection of past gathered data or experiences from the environment to train its world model. Then it uses its planning routine to collect more experiences from the real environment and stores them in a simple experience replay buffer. The agent iterates between these two steps as long as it trains in the environment.

The planning routine performs model-predictive control (Richards, 2005). So, the agent replans at each step based on the input image it receives. Furthermore, the cross entropy method (CEM) (Rubinstein, 1997; Chua et al., 2018) is used to search for the best action at each step. CEM is a population-based optimization that tries to sample action sequences with a fixed horizon length that maximizes the expected return. Specifically, CEM starts with a zero mean and unit variance as its current belief and iteratively samples a pool of candidate action sequences, evaluates them under the world model, and picks some top-performing ones while updating the belief. And after a fixed number of iterations returns, the mean of the belief is used as the chosen action.

The latent dynamics is modeled using an architecture called recurrent state space model (RSSM), which has both deterministic (h_t) and stochastic (s_t) components. Its recurrent nature allows the agent to remember information over many steps before. Specifically, at step t, given the previous stochastic and deterministic latent states (s_{t-1}, h_{t-1}), and action a_{t-1}, RSSM first computes the new deterministic states through a recurrent neural network named f as follows:

$$h_t = f(h_{t-1}, s_{t-1}, a_{t-1}) \tag{2.4.1}$$

Then the RSSM samples the stochastic latent state s_t using the transition model as:

$$s_t \sim p(s_t|h_t) \tag{2.4.2}$$

And, the input image (o_t) is reconstructed with an observation model, and also the reward (r_t) is predicted with a reward model as follows:

$$\begin{aligned} o_t &\sim p(o_t|h_t, s_t) \\ r_t &\sim p(r_t|h_t, s_t) \end{aligned} \tag{2.4.3}$$

Lastly, an encoder q is used to parameterize the approximate stochastic state posteriors:

$$q(s_{1:T}|o_{1:T}, a_{1:T}) = \prod_{t=1}^{T} q(s_t|h_t, o_t) \tag{2.4.4}$$

The transition, observation, and reward models, together with the encoder, are assumed to be Gaussian distributions. They are parametrized by the mean and variance modeled by deep neural networks, such as convolutional and MLP architectures. The RSSM is trained by maximizing the variational lower bound on the input images and rewards log-likelihood. The following shows the mentioned bound for the input images, and the bound for the rewards could be derived similarly:

$$\begin{aligned} \ln p(o_{1:T}|a_{1:T}) &\triangleq \ln \int \prod_t p(s_t|s_{t-1}, a_{t-1}) p(o_t|s_t)\, \mathrm{d}s_{1:T} \\ &\geq \sum_{t=1}^{T} \big(\mathbb{E}_{q(s_t|o_{\leq t}, a_{\leq t})}[\ln p(o_t|s_t)] \\ -\mathbb{E}_{q(s_{t-1}|o_{\leq t-1}, a_{\leq t-1})} &\left[\mathrm{KL}[q(s_t|o_{\leq t}, a_{\leq t}) || p(s_t|s_{t-1}, a_{t-1})] \right] \big) \end{aligned} \tag{2.4.5}$$

Additionally, we can form a loss function that includes the $\ln p(o_t|s_t)$ and $\ln p(r_t|s_t)$ plus the KL term in the right side of the Equation 2.4.5 multiplied by a coefficient named β.

Finally, it is important to note that since the RSSM is trained to predict forward in the latent space (namely s_t), the planning routine performs imaginary rollouts in the same latent space to find the best action possible. Therefore, since the latent space has a much lower dimension than the input images dimension, the planning routine is computationally sensible for one thing. And also, the latent space can capture more meaningful features of the underlying task than just raw pixels for the other.

2.4.3. Dreamer

Dearmer (Hafner et al., 2019a) is a deep model-based RL algorithm that could be considered as a successor method to the PlaNet algorithm. It uses the same world model as the PlaNet agent; however, instead of using an online planning routine like the CEM, it learns a value model and an actor model (actor-critic) via backpropagation through predictions of its world model. This way, Dreamer's planning routine is more efficient in two ways. First, the actor model computes the best action without additional search, so planning does not rely on multiple rollouts and many predictions to find the best action at each step; therefore, it is more computationally sensible.

Second, it considers rewards beyond the planning horizon using the value model, which leverages backpropagation for efficient planning.

In general, there are three main steps for training the Dreamer agent. First, the agent uses a collection of past gathered data or experiences from the environment to train its world model that uses the RSSM architecture parameterized with θ (same as the PlaNet). Second, the agent trains its value network ($v_\psi(s_t)$ parameterized with ψ) and actor network ($q_\phi(a_t|s_t)$ parameterized with ϕ) by propagating gradients back through solely imagined trajectories. And then, it uses its actor network to act in the environment and to collect more experiences from the environment and stores them in a simple experience replay buffer. Finally, the agent iterates between these three steps for as long as it trains in the environment.

Dreamer's planning only works with the latent space of the world model; hence, it does not generate real-world data. In other words, it only considers the imagined trajectories starting from a particular input observation and rolling out the actor model within the world model with a fixed horizon H. So, the actor model is responsible for finding the optimal policy that solves the imagination (simulation) environment. And the value model is responsible for estimating the expected discounted imagined rewards that the action model achieves from a given latent state s_t:

$$\text{Actor Model:} \quad a_t \sim q_\phi(a_t|s_t) \tag{2.4.6}$$

$$\text{Value Model:} \quad v_\psi(s_t) \approx \mathbb{E}_{q(.|s_t)}(\sum_{k=t}^{t+H} \gamma^{t-k} r_k) \tag{2.4.7}$$

To learn the actor and value models, Dreamer needs to have an estimate of the state values of imagined trajectories $\{s_k, a_k, r_k\}_{k=t}^{t+H}$. For this purpose, Dreamer leverages temporal-difference learning, where the value is trained toward a value target. Value targets could be constructed in various ways; for instance, the most common choice is the 1-step target that sums the current reward and the discounted value output for the next state. Dreamer, however, uses an exponentially weighted average of the n-steps returns to balance bias and variance. This way of calculating value targets is also known as λ-targets or λ-returns (Sutton and Barto, 2018, Section 12.1), and it could recursively be defined as follows:

$$V_k^\lambda \doteq r_k + \gamma \begin{cases} (1-\lambda) v_\psi(s_{k+1}) + \lambda V_{k+1}^\lambda & \text{if } k < t+H \\ v_\psi(s_{t+H}) & \text{if } k = t+H \end{cases} \tag{2.4.8}$$

Given the imagined trajectory, the critic is trained to regress the λ-return using a squared loss:

$$\mathcal{L}(\psi) \doteq \mathbb{E}_{p_\theta, q_\phi}[\sum_{t=1}^{H-1} \frac{1}{2}(v_\psi(s_t) - \text{sg}(V_t^\lambda))^2] \tag{2.4.9}$$

where θ is the RSSM's parameters, and sg(.) denotes stopping the gradient around the targets (λ-returns).

The actor model, on the other hand, is trained to maximize the value estimates (or minimize the negative value estimates) by using analytic gradients through the learned dynamics. The loss function for the actor model is defined as follows:

$$\mathcal{L}(\phi) \doteq \mathbb{E}_{p_\theta,q_\phi}[\sum_{t=1}^{H-1} -V_t^\lambda] \tag{2.4.10}$$

Now that we talked about how the actor and value models are trained, we have all the necessary parts to train the Dreamer agent. Additionally, an enhanced version of the Dreamer algorithm, known as DreamerV2 (Hafner et al., 2020), was developed to incorporate several enhancements to the RSSM and the planning routine. DreamerV2 served as the primary version of the Dreamer algorithm throughout our work.

We only focus on the notable differences between Dreamer and DreamerV2. First, the RSSM in DreamerV2 represents each input image with multiple categorical variables instead of Gaussian variables previously used by PlaNet and Dreamer. This way, the world model is able to reason about the world in terms of discrete concepts and enables more accurate predictions of future representations. Furthermore, a new technique called KL balancing is also added to the RSSM. KL balancing encourages learning an accurate prior representation over increasing posterior entropy to approximate the aggregate posterior better. It is implemented by minimizing the KL loss faster for the prior than the stochastic representations by using different learning rates for the prior and the approximate posterior.

Moreover, on the actor side, the loss function has been updated to include REINFORCE gradients (Williams, 1992) and an entropy regularizer to encourage exploration where feasible. The updated loss function could be shown as the following:

$$\mathcal{L}(\phi) \doteq \mathbb{E}_{p_\theta,q_\phi}[\sum_{t=1}^{H-1}(-\rho \ln p_\phi(a_t|s_t)\mathrm{sg}(V_t^\lambda - v_\psi(s_t)) - (1-\rho)V_t^\lambda - \eta H[a_t|s_t])] \tag{2.4.11}$$

where ρ and η are the hyperparameters and H represents the entropy term.

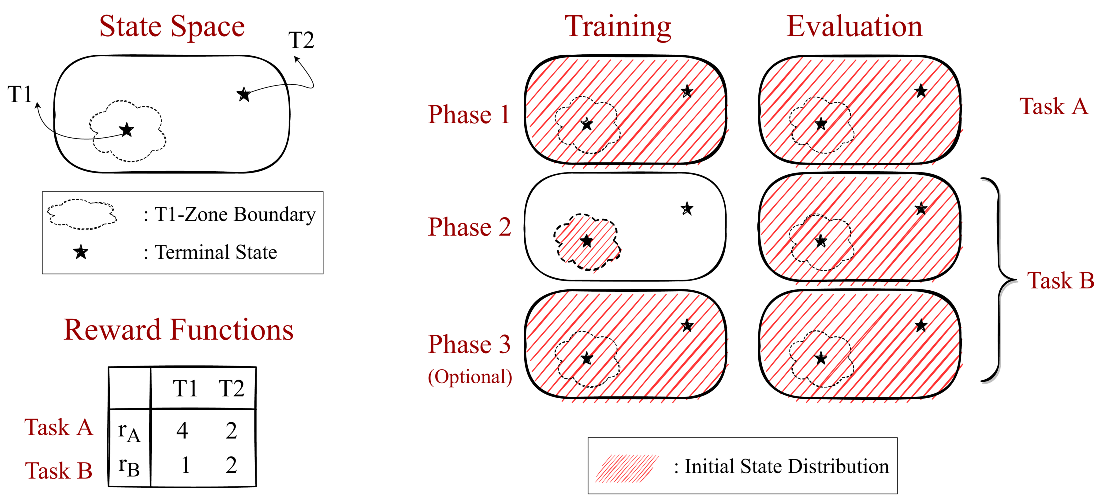
Chapter 3

Local Change Adaptation (LoCA) Setup

In this chapter, we present the LoCA setup introduced by Van Seijen et al. (2020), as well as our improved version that is simpler and more robust. The LoCA setup consists of a task configuration and an experiment configuration. The LoCA setup is inspired by how model-based behavior is identified in behavioral neuroscience (e.g., see Daw et al., 2011).

The task configuration is the same for the original and our improved version of the LoCA setup and is discussed next (Section 3.1). The original experiment configuration is discussed in Section 3.2; Section 3.3 discusses our improved version. Finally, we list all the different domains utilised to assess the adaptivity of various MBRL methodst, in Section 3.4.

3.1. Task Configuration



		T1	T2
Task A	r_A	4	2
Task B	r_B	1	2

Fig. 3.1. *Left:* LoCA task configuration. *Right:* The initial state distribution during training and evaluation across the three phases of a LoCA experiment.

The LoCA task configuration considers two different tasks (i.e., reward functions) in the same environment. A method's adaptivity is determined by measuring how effective it can adapt from the first to the second task. The task configuration only specifies some specific features that the environment should have. In practice, 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 (see Section 3.4).

A LoCA environment contains two terminal states, T1 and T2 (see left of Figure 3.1). 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).

3.2. Experiment Configuration - Original Version

An experiment consists of three different training phases. 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 is different for the different phases (see right of Figure 3.1). 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. As the agent cannot escape the T1-zone without terminating the episode, during Phase 2 only samples from the T1-zone are observed.

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. Effective adaptation in Phase 2 implies the performance is optimal out of the gate in Phase 3. In the original LoCA setup, evaluation only occurs in Phase 3. If a method does not perform optimal right out of the gate (i.e., does not adapt effectively), the amount of additional training needed in Phase 3 to get optimal performance is used as a measure for how far off the behavior of a method is from ideal adaptive behavior.

Evaluation occurs 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 evaluation

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

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 cut-off time 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 experiment configuration 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.

3.3. Experiment Configuration- Improved Version

To address the shortcomings of the original LoCA experiment configuration, we introduce an improved version. In this improved version we evaluate performance 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 in between T1 and T2. Finally, we evaluate the performance during all phases, instead of only the third phase.

Under our new experiment configuration, we call a method adaptive if it is able to reach (near) optimal expected return in Phase 2 (after sufficiently long training), while also reaching (near) optimal expected return in Phase 1. If a method is able to reach (near) optimal expected return in

Phase 1, but not in Phase 2, we call the method non-adaptive. Finally, if a method is unable to reach near-optimal expected return in Phase 1, even after training for a long time, we do not make an assessment of its adaptivity.

Using the expected return to evaluate the quality of the policy instead of a top-terminal makes evaluation a lot more flexible and robust. Not only does it remove the cut-off time hyperparameter, it enables the use of the full state-space as initial-state distribution and can be applied without modification to stochastic environments. These scenarios were tricky for the top-terminal fraction, as it had as requirement that adaptive methods should be able to get a top-terminal fraction of 1.

Finally, our improved LoCA setup[2] no longer tries to assess how far off from ideal adaptive behavior a method's behavior is—a measure conflated by various confounders, as discussed above. Instead, only a binary assessment of adaptivity is made, simplifying evaluation.

Note that 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. In particular, if after training for a long time in Phase 3 the performance plateaus at some suboptimal level, two things can be the case. 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.

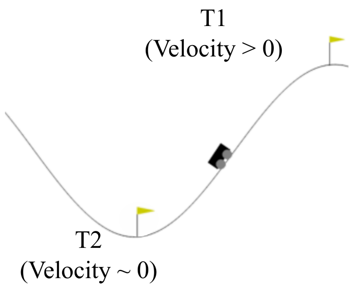
3.3.1. Discussion

We want to stress that the LoCA setup is not meant to be a benchmark; it is meant to be an analytical tool. In other words, its purpose is not to mimic some challenging real-world task that requires a broad range of capabilities. Instead, it is designed to test for one particular capability only, while removing/reducing any effects from other capabilities on the outcome. The particular capability that the LoCA setup tests for is to evaluate the ability of a method to push/pull newly observed information to parts of the state-space not recently visited. The relevance of this property is that it can be regarded as a critical condition for local adaptivity and a key feature of model-based behavior. That being said, there are other forms of adaptivity, as well as other behavioral characteristics, that are important too (i.e., exploration behavior) and not measured by the LoCA setup. As such, the setup is not meant to be a complete test of the efficacy of a method. Instead, it is meant to keep research on model-based learning on track and create awareness that many of the current deep MBRL methods don't exhibit the key feature of model-based behavior.

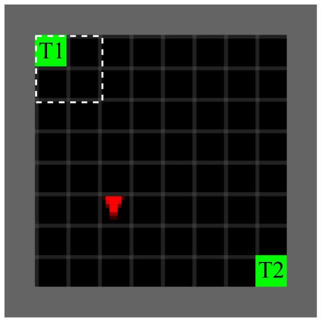
[2]For simplicity, from now on, we use 'the LoCA setup' to denote the *improved* LoCA setup unless specified otherwise.

3.4. LoCA Domains

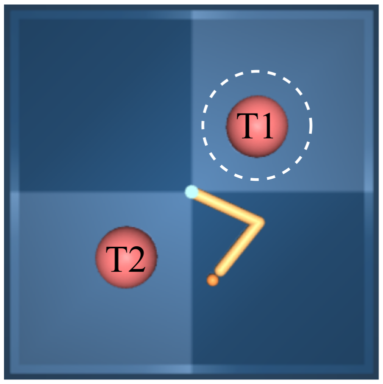
This section provides a thorough explanation of the various domains (learning environments) that we consider when determining if an MBRL method is adaptive or not. Overall we have used five different domains in our work: (1) GridWorldLoCA (2) MountainCarLoCA (3) MiniGrid-LoCA (4) ReacherLoCA (5) RandomizedReacherLoCA. In what follows, we concisely describe each domain.



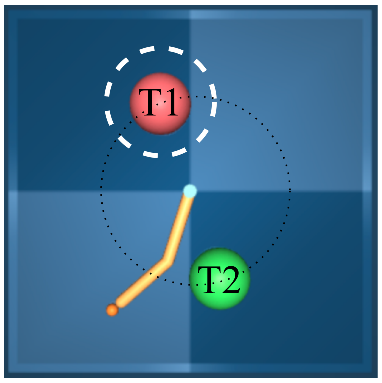
(a) MountainCarLoCA



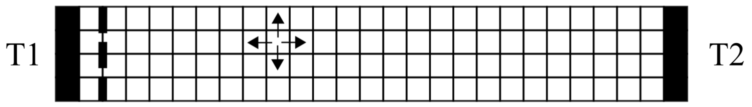
(b) MiniGridLoCA



(c) ReacherLoCA



(d) RandomizedReacherLoCA



(e) GridWorldLoCA

Fig. 3.2. Illustration of environments corresponding to the different domains used in our experiments. The dashed white lines (not visible to the agent) show the boundary of T1-zone.

3.4.1. GridWorldLoCA

GridWorldLoCA is a 25×4 tabular navigation domain that was first introduced in Van Seijen et al. (2020). An illustration of this domain is given in Figure 3.2 (e). The agent is located in a specific cell in the entire grid and, at each timestep, can choose an action that takes it to any adjacent cell. T1 is any cell located on the leftmost side of the grid, and T2 is the cells located on the right side of the grid. T1-zone is the set of cells that are adjacent to the T1. Finally, the discount factor $\gamma = 0.97$; this way, an agent with the optimal policy would move to the high-reward terminal in each LoCA setup's phase.

3.4.2. MountainCarLoCA

MountainCarLoCA, first introduced by Van Seijen et al. (2020), consists of a variation of the classical MountainCar domain (Moore, 1990), with an under-powered cart having to move up a hill. There are two terminal states, T1 at the top of the hill and T2, corresponding to the cart being at the bottom of the hill with a velocity close to zero (Figure 3.2 (a)).

More formally, T1 is located at the top of the mountain (position > 0.5, and velocity > 0), and T2 is located at the valley ($(\text{position} + 0.52)^2 + 100 \times \text{velocity}^2 \leq 0.07^2$). The T1-zone contains all the states within $0.4 \leq$ position ≤ 0.5 and $0 \leq$ velocity ≤ 0.07. The discount factor for this environment is $\lambda = 0.99$. And lastly, for each evaluation, the agent is initialized roughly at the middle of T1 and T2 ($-0.2 \leq$ position ≤ -0.1 and $-0.01 \leq$ velocity ≤ 0.01).

3.4.3. MiniGridLoCA

MiniGridLoCA setup has an image-based high-dimensional input. The environment (Figure 3.2 (b)) is an 8×8 RGB grid world with two green-colored terminal states, T1 at the top left corner with a 2×2 T1-zone and T2 at the bottom-right corner. The agent has a red triangular form. The tip of the triangle determines the direction of the agent. At each timestep, the agent can choose to turn left, turn right, or go straight to the adjacent cell (it will remain in the same state if the action takes the agent outside the grid). And the discount factor $\gamma = 0.98$. The implementation of the MiniGridLoCA domain is done using the MiniGrid python package (Chevalier-Boisvert et al., 2018).

3.4.4. ReacherLoCA

ReacherLoCA is a variation on the Reacher environment (the easy version) available from the DeepMind Control Suite (Tassa et al., 2018). It features a continuous-action domain with $64 \times 64 \times 64$ dimensional (pixel-level) states. 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 3.2 (c)). 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 timesteps) 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.

In this 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.

3.4.5. RandomizedReacherLoCA

RandomizedReacherLoCA is a randomized extension of ReacherLoCA domain (Section 3.4.4). In this domain, We first change the color of the T2 target from red to green, so the agent is able to differentiate between the two targets. Then, the location of the red target (T1) is randomly taken from the circle centered in the center of the state space (the dotted black circle in Figure 3.2 (d)). And then, the green target (T2) is placed at the opposite end of the circle. Finally, we keep all other environment dynamics the same as the ReacherLoCA domain.

The RandomizedReacherLoCA domain can be thought of as a more complex version of the ReacherLoCA. Since the targets' locations are chosen at random before each episode begins, we may argue that this domain is composed of a random distribution among numerous ReacherLoCA domains.

Chapter 4

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 minor discrepancies among different MBRL methods can significantly affect their adaptivity. In Section 4.1, we will illustrate both these points by evaluating three different tabular MBRL methods in the GridWorldLoCA domain (Figure 3.2 (e)), using the LoCA setup. In Section 4.1.1, we discuss in more detail why some of the tabular MBRL methods fail to adapt effectively.

We will conclude this chapter by evaluating the adaptivity of two popular deep MBRL methods, PlaNet (Hafner et al., 2019b) and DreamerV2 (Hafner et al., 2019a, 2020), in the ReacherLoCA domain (Figure 3.2 (c)) in Section 4.2. Based on our experiments, we carefully pinpoint the causes of such deep MBRL approaches' failure to adapt to environmental changes.

4.1. Tabular MBRL Experiments

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:

$$\hat{r}(S_t,A_t) \;\leftarrow\; \hat{r}(S_t,A_t) + \alpha\big(R_t - \hat{r}(S_t,A_t)\big)\,, \tag{4.1.1}$$

$$\hat{p}(\cdot|S_t,A_t) \;\leftarrow\; \hat{p}(\cdot|S_t,A_t) + \alpha\big(\langle S_{t+1}\rangle - \hat{p}(\cdot|S_t,A_t)\big)\,, \tag{4.1.2}$$

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.[1] For a 1-step model:

$$V(s) \leftarrow \max_a \Big(\hat{r}(s,a) + \gamma \sum_{s'} \hat{p}(s'|s,a)V(s') \Big) . \tag{4.1.3}$$

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 model are as follows:

$$\hat{r}_2(S_t,A_t) \quad \leftarrow \quad \hat{r}_2(S_t,A_t) + \alpha\big(R_t + \gamma R_{t+1} - \hat{r}_2(S_t,A_t)\big) , \tag{4.1.4}$$

$$\hat{p}_2(\cdot|S_t,A_t) \quad \leftarrow \quad \hat{p}_2(\cdot|S_t,A_t) + \alpha\big(\langle S_{t+2}\rangle - \hat{p}_2(\cdot|S_t,A_t)\big) , \tag{4.1.5}$$

with α the fixed learning rate and $\langle S_{t+2}\rangle$ a one-hot encoding of state S_{t+2}. Both $\langle S_{t+2}\rangle$ and $p(\cdot|S_t,A_t)$ are vectors of length N, the total number of states. To be able to make these updates, *mb-2-r* needs to store at least the last two samples and performs updates with a small delay, as it would be for any n-step method (for more details on that, see Chapter 7 of Sutton and Barto (2018)). Planning consists of performing a single state-value update at each time step based on the model estimates:

$$V(s) \leftarrow \max_a \Big(\hat{r}_2(s,a) + \gamma^2 \sum_{s''} \hat{p}_2(s''|s,a)V(s'') \Big) . \tag{4.1.6}$$

A summary of the methods we evaluate is shown in Table 4.1. Besides these methods, we also test the performance of the model-free method Sarsa(λ) with $\lambda = 0.95$.

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

Table 4.1. Tabular model-based methods being evaluated.

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 distribution for training in Phases 1 and 3, as well as the initial-state distribution for evaluation in

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

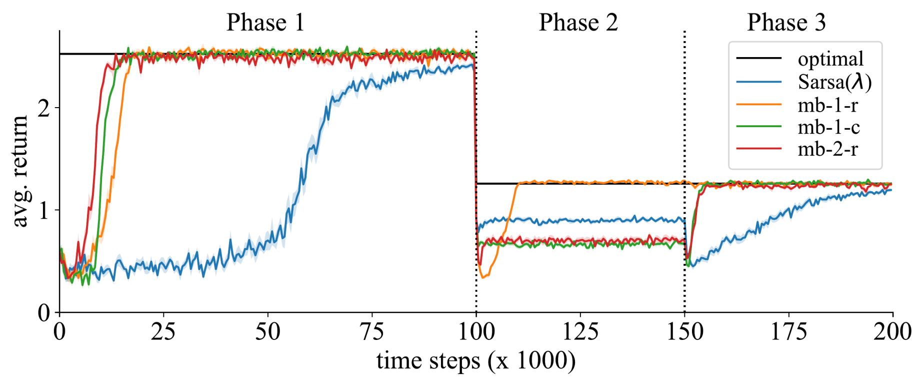


Fig. 4.1. Comparison of the 3 model-based methods from Table 4.1 on the GridworldLoCA domain. While all methods converge to the optimal policy in Phases 1 and 3, only *mb-1-r* converges in Phase 2. We call model-based methods that are able to converge to optimal performance in Phase 2 (locally) adaptive model-based methods.

all three phases 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.

Table 4.2 shows the setup of our experiments for the tabular methods. Additionally, for all methods, we used an ϵ-greedy behavior policy with $\epsilon = 0.1$. Furthermore, 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.

Figure 4.1 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.

4.1.1. 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 result 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 Failure Mode #1 is 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 using update targets based on values that are computed only for the visited states from an episode-trajectory, which is similar to Failure Mode #1. Furthermore, an example of a linear multi-step MBRL method that Failure Mode #2 applies to is LS-Sarsa(λ) introduced by van Seijen and Sutton (2015).

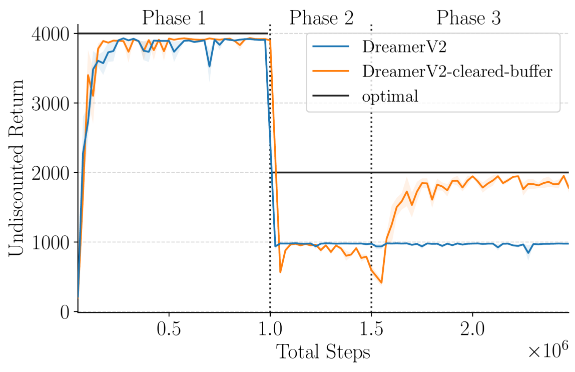
4.2. PlaNet and DreamerV2 Experiments

In this section, we evaluate the adaptivity of two deep model-based methods, PlaNet (Hafner et al., 2019b), and DreamerV2 (Hafner et al., 2020), using the ReacherLoCA domain (Figure 3.2 (c)).

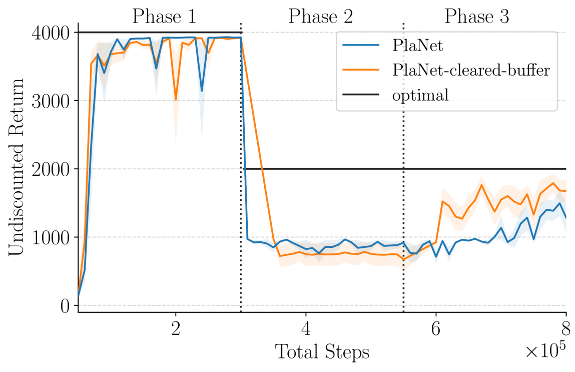
Note that both PlaNet and DreamerV2 learn from transitions sampled from a large experience replay buffer (they call it experience dataset) that contains all recently visited transitions. When evaluating these algorithms in the ReacherLoCA domain, such a strategy could hurt the agent's ability of adaptation,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

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 Phase 2 steps Phase 3 steps	10^5 5×10^4 5×10^4
Other details	Training steps between two evaluations Number of runs Number of evaluation episodes	500 10 100

Table 4.2. Setup of tabular experiments.



(a)



(b)

Fig. 4.2. 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).

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

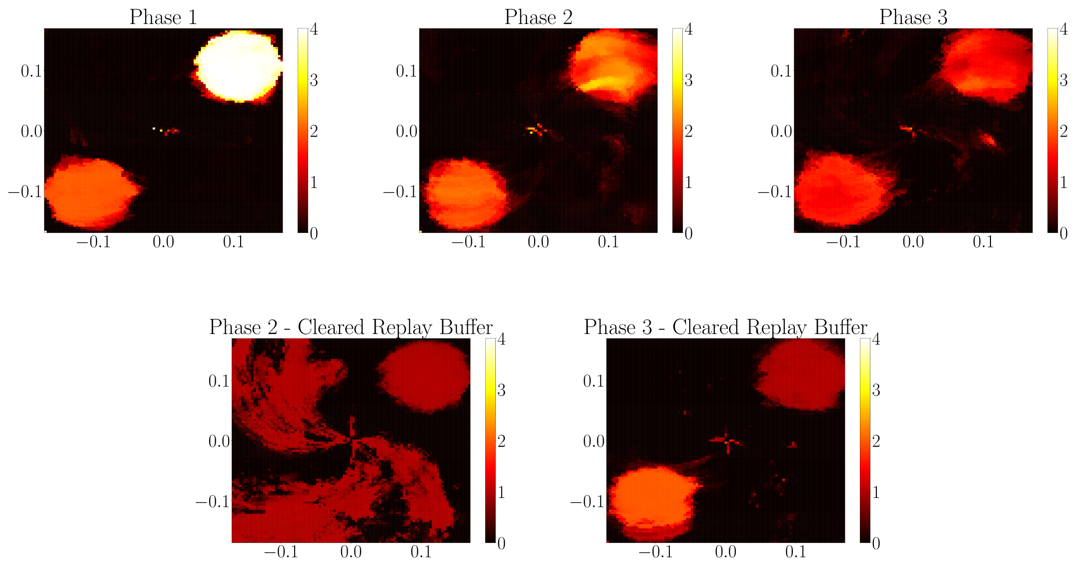


Fig. 4.3. Visualization of the DreamerV2 agent's estimated reward model generated at the end of each phase. The x and y axes of each heatmap represent the agent's position in the ReacherLoCA domain.

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.

Table 4.3 shows the setup of our experiments for the DreamerV2 and PlaNet algorithms. 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. We searched over the KL loss scale $\beta \in \{0.1, 0.3, 1, 3\}$, the actor entropy loss scale $\eta \in \{10^{-5}, 10^{-4}, 3\times10^{-4}, 10^{-3}\}$, 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.

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 rewards over five evaluation episodes. The shading area shows the standard error. 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 $\gamma = 0.99$.

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.

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

For each variant of each of the two algorithms, we drew a learning curve (Figure 4.2) corresponding to the best hyperparameter setting found by a grid search. The reported learning curves suggest that neither DreamerV2 nor PlaNet effectively adapted their policy in Phase 2, regardless of whether the replay buffer was 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 4.3 the reward predictions of DreamerV2's world model at the end of each phase. The corresponding results for PlaNet are similar to those of DreamerV2. Figure 4.3 is produced using the best hyperparameter setting. The seed used is randomly picked. Furthermore, we have tested other random seeds and hyperparameter settings (not shown in our work) and obtained similar results as those shown in Figure 4.3. Our empirical results provide affirmative answers to the two questions raised at the beginning of this section.

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 and 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 4.1.1.

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 (Failure Mode #4) is an indispensable need for the LoCA setup 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.

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	Number of steps before an episode terminates	1000
	Training steps between two evaluations	PlaNet: 10^4, DreamerV2 5×10^4
	Number of runs	5
	Number of evaluation episodes	5

Table 4.3. Setup of DreamerV2 and PlaNet experiments.

Chapter 5

Adaptive MBRL with Linear Function Approximation

In this chapter, we identify an algorithm with linear function approximation that does not fall into the four aforementioned failure modes and understand its behavior using the LoCA setup in the MountainCarLoCA domain (Figure 3.2 (a)).

5.1. Adaptive Linear Dyna

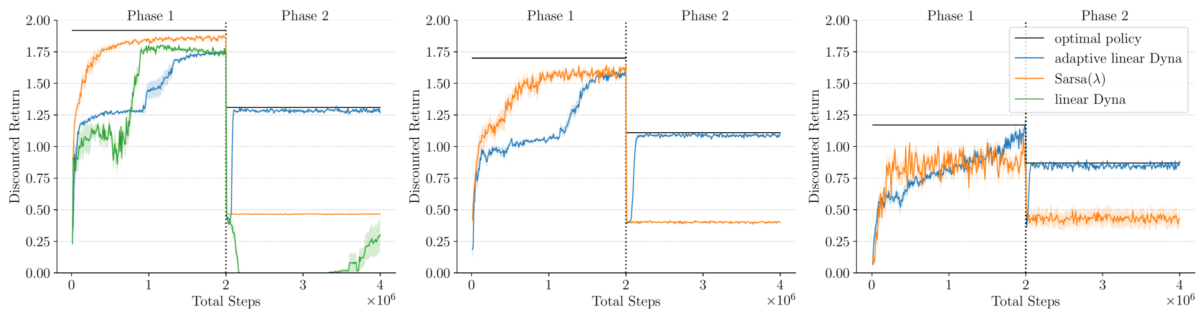
Our proposed algorithm, called *adaptive linear Dyna*, is a modified version of the linear Dyna algorithm (Algorithm 4 by Sutton et al., 2012). To overcome the unsolved catastrophic forgetting problem with neural networks, this algorithm takes a step back by using linear function approximation with sparse feature vectors. We provide empirical evidence showing that this algorithm is adaptive in the MountainCarLoCA domain.

Sutton et al. (2012)'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 and 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. (2012)'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 5.1, we empirically show that the 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 Algorithm 5.1.1. Details of the empirical analysis are presented in Section 5.2.

```
Algorithm 5.1.1: Adaptive Linear Dyna
   Input: exploration parameter $\epsilon$, step-size $\alpha$, 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 $n_p$.
 1 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)
 2 Obtain initial state $S$, Compute feature vector $\phi$ from $S$.
 3 while still time to train do
 4     $A \leftarrow \epsilon\text{-greedy}_a(b_a^\top \phi + \gamma \theta^\top F_a^\top \phi)$
 5     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).
 6     Add $\phi$ to the planning buffer (if the buffer is full, remove the oldest element and then
         add $\phi$).
 7     Compute the next feature vector $\phi'$ from $S'$.
 8     if $A$ is greedy then
 9         $\delta \leftarrow R + \gamma(1 - Z)\theta^\top \phi' - \theta^\top \phi$
10         $\theta \leftarrow \theta + \alpha \delta \phi$
11     end
12     $F_A \leftarrow F_A + \beta((1 - Z)\phi' - F_A \phi)\phi^\top$
13     $b_A \leftarrow b_A + \beta(r - b_A^\top \phi)\phi$
14     for all of the $n$ planning steps do
15         Generate a start feature vector $x$ by randomly sampling from the planning buffer.
16         $\delta \leftarrow \max_a(b_a^\top x + \gamma \theta^\top F_a^\top x) - \theta^\top x$
17         $\theta \leftarrow \theta + \alpha \delta x$
18     end
19     $\phi \leftarrow \phi'$
20 end
```



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

Fig. 5.1. 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).

5.2. Experiments

We designed experiments to test if the adaptive linear Dyna algorithm can adapt well with proper choices of hyperparameters. We tested our algorithm on the MountainCarLoCA domain (Figure 3.2 (a)) with an additional stochasticity level p. 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. 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 5.1. 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. Further, we show a learning curve of linear Dyna by Sutton et al. (2012).

For our experiments, we found that the original linear Dyna algorithm (Algorithm 4 by Sutton et al. (2012)) takes too much memory and computation, making it difficult to test. 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 five randomly chosen predecessor features and adds them to the queue. These two modifications we made significantly reduced memory usage and computation.

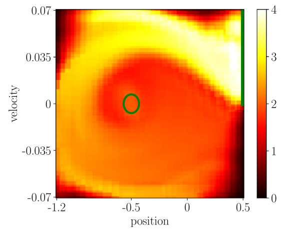
We summarize in Table 5.1 tested hyperparameters for linear Dyna and adaptive linear Dyna. Furthermore, for Sarsa(λ), the tile coding setups and the choices of stepsize α are the same as those used for adaptive linear Dyna. The exploration parameter was fixed to be 0.1. We also tested three λ values: 0, 0.5, and 0.9. Table 5.2 shows the experiment setup used to test linear Dyna, adaptive linear Dyna (Algorithm 5.1.1), and Sarsa(λ).

The hyperparameter setting chosen to draw curves in Figure 5.1 was chosen in the following way. We first collected hyperparameter settings that achieved 90% of the optimal average return over the last five evaluations in Phase 1. From these hyperparameter settings, we picked one that achieved the highest average return over the last five 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.

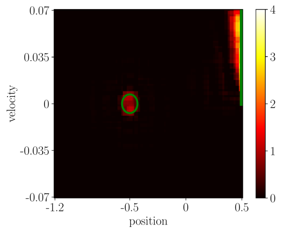
We summarize in Table 5.3 the best hyperparameter settings under different levels of stochasticities for each algorithm.

The learning curves in Figure 5.1 show that for all different levels of stochasticity, 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 5.2 and 5.3) 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 obtain 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.

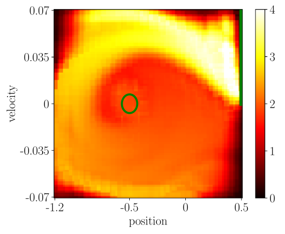
The other observation is that when stochasticity = 0.5, Sarsa(λ) is worse than adaptive linear Dyna. Note that there is a very high variance in the learning curve of Sarsa(λ). On the contrary, adaptive linear Dyna achieved a much lower variance, potentially due to its planning with a learned model, which 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.



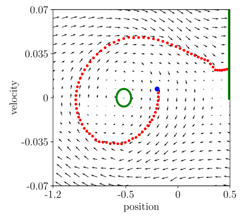
(a) Values



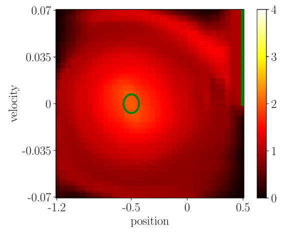
(b) Rewards



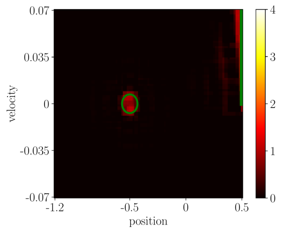
(c) Expected Next Values



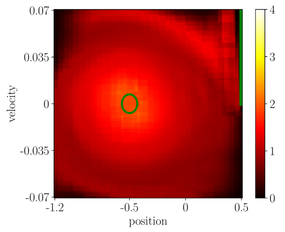
(d) Policy



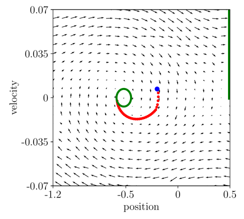
(e) Values



(f) Rewards



(g) Expected Next Values

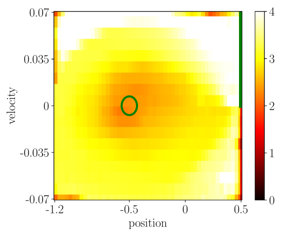


(h) Policy

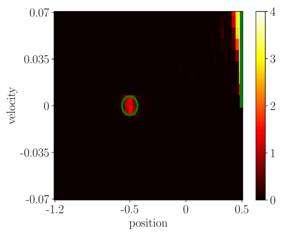
Fig. 5.2. Plots showing estimated values, reward model, the value of the predicted next state produced by the model, and the learned policy of adaptive linear Dyna in the deterministic MountainCarLoCA domain, with the best hyperparameter setting (Table 5.3). The first and second rows correspond to the results at the end of Phases 1 and 2 respectively. In each sub-figure, the x-axis is the position of the car, and 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 estimated immediate reward. The third column corresponds to the maximum 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 a sequence of red dots, illustrating the 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 5.4 shows that resulting values are close to optimal values for most states.

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]
Step-size 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]

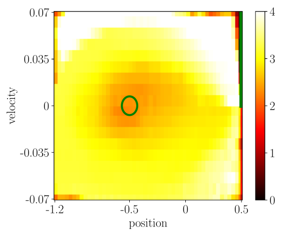
Table 5.1. Tested hyperparameters in the adaptive linear Dyna experiment.



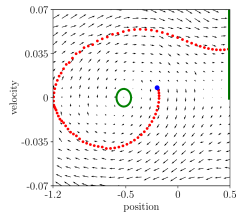
(a) Values



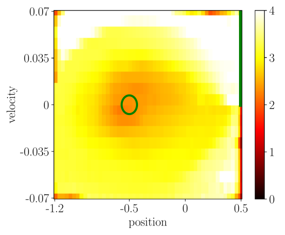
(b) Rewards



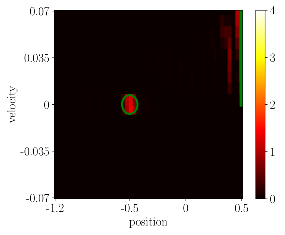
(c) Expected Next Values



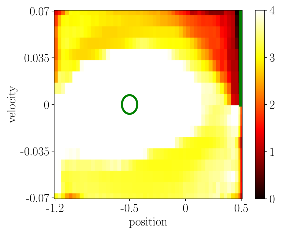
(d) Policy



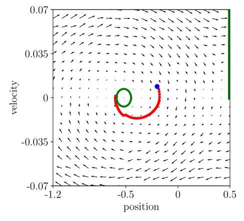
(e) Values



(f) Rewards



(g) Expected Next Values

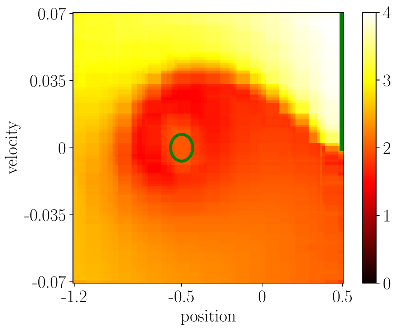


(h) Policy

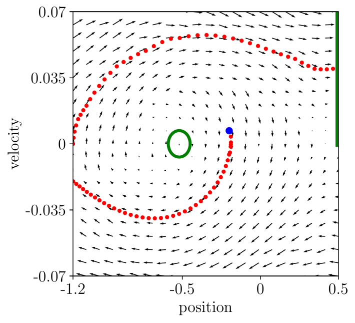
Fig. 5.3. Plots showing estimated values, reward model, the value of the predicted next state produced by the model, and the learned policy of linear Dyna in the deterministic MountainCarLoCA domain. Comparing Sub-figures (a), (e), and Sub-figures a), c) in Figure 5.4 show that resulting values are far from optimal values.

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 Phase 2 steps	2×10^6 2×10^6
Other details	Training steps between two evaluations Number of runs Number of evaluation episodes	10^4 10 10

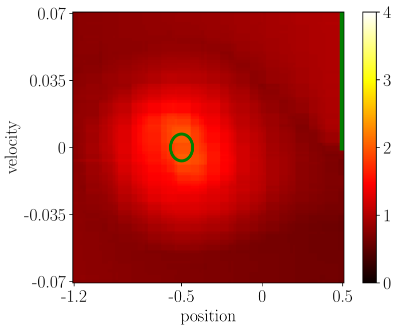
Table 5.2. Experiment setup used to test linear Dyna, adaptive linear Dyna, and Sarsa(λ).



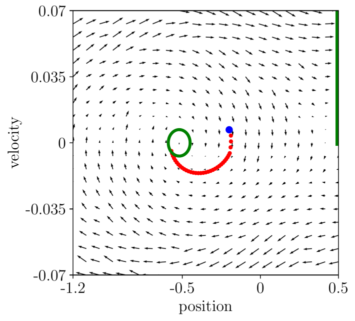
(a) Optimal Values (Task 1)



(b) Optimal Policy (Task 1)



(c) Optimal Values (Task 2)



(d) Optimal Policy (Task 1)

Fig. 5.4. Plots showing "*optimal*" values and policies in the two tasks obtained by running linear Sarsa(λ) for sufficiently long in the deterministic MountainCarLoCA domain.

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 5.3. Best hyperparameter settings in the adaptive linear Dyna experiment.

Chapter 6

Towards Adaptive Deep MBRL

6.1. Initial Deep Dyna-Q Experiments

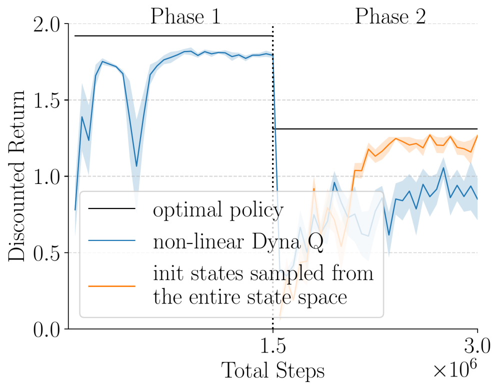
Based on the adaptive linear Dyna algorithm (Section 5.1.1), we propose the deep Dyna-Q algorithm (Algorithm 6.1.1), in which the value function and the model are both approximated using neural networks. Instead of trying to solve the catastrophic forgetting problem, we adopt the simple replay approach. 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 (Section 4.2), 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 randomly from the entire buffer and the rest randomly only from rewarding data (transitions leading to rewards). Our empirical results in the MountainCarLoCA domain confirm that this dilemma is critical – the learned model is inferior, even with the best hyperparameter setting and sampling strategy.

In addition to maintaining a learning buffer and a planning buffer, there are two other ways in which the deep Dyna-Q algorithm is different from the linear algorithm. First, the deep algorithm needs to predict the termination of each episode, while the linear algorithm can simply assign the terminal state with a zero feature vector. And also, for the experiments with the deep algorithm in this section, we maintained separate networks for different actions (i.e., no parameters are shared across different actions) to estimate the state-action value function. In this way, interference among different actions is removed.

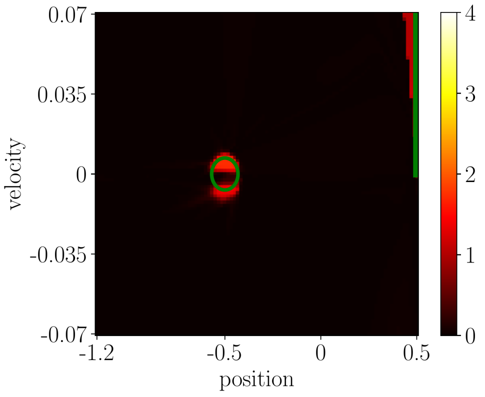
We designed experiments to test if the deep Dyna-Q algorithm can successfully adapt in the deterministic MountainCarLoCA domain. Table 6.2 shows the experiment setup used to test the deep Dyna-Q algorithm (Algorithm 6.1.1). We summarize in Table 6.1 tested hyperparameters for the deep Dyna-Q algorithm. The hyperparameter setting used to generate the reported learning curve

(Figure 6.1(a)) chosen in a similar way as those chosen for the adaptive linear Dyna experiment. The best hyperparameter setting of deep Dyna-Q in this experiment is summarized in Table 6.3.

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 6.1. We picked one run with the best hyperparameter setting and plotted the estimated reward model at the end of Phase 2 in Figure 6.1. 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.



(a) Values



(b) Policy

Fig. 6.1. Plots showing learning curves and the estimated reward model of deep Dyna-Q at the end of training in the deterministic MountainCarLoCA domain. a) deep Dyna-Q struggles to adapt in Phase 2 (blue curve). As a baseline, we show the learning curve produced by applying deep 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 T1. The color of each point represents the model's reward prediction of a state (maximized over different actions).

Algorithm 6.1.1: Deep Dyna-Q

```
Input: exploration parameter $\epsilon$, value step-size $\alpha$, model step-size $\beta$, 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 $m_p$.
1  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 $\alpha$ and parameters $\mathbf{w}$, model optimizer with step-size $\beta$ and
     parameters $\mathbf{u}, \mathbf{v}, \mathbf{z}$.
2  Obtain the vector representation of the initial state s.
3  while still time to train do
4      $A \leftarrow \epsilon$-greedy$_a(\hat{q}_{\mathbf{w}}(\mathbf{s}, \cdot))$
5      Take action $A$, receive reward $R$, vector representation of the next state $\mathrm{s}'$, and
         termination $Z$.
6      Add $(\mathrm{s}, A, R, \mathrm{s}', Z)$ to the learning buffer. If the learning buffer is full, replace the
         oldest element by $(\mathrm{s}, A, R, \mathrm{s}', Z)$.
7      Add s to the planning buffer. If the planning buffer is full, replace the oldest element
         by s.
8      for all of the $m$ model learning steps do
9          Sample a $m_l$-sized mini-batch $(\mathbf{s}_i, a_i, r_i, \mathrm{s}'_i, z_i)$ from the learning buffer.
10         Apply model optimizer to the loss
             $\sum_i \frac{1-z_i}{2}(\mathrm{s}'_i - f_{\mathbf{u}}(\mathbf{s}_i, a_i))^2 + \frac{1}{2}(r_i - b_{\mathbf{v}}(\mathbf{s}_i, a_i))^2 + \frac{1}{2}(z_i - t_{\mathbf{z}}(\mathbf{s}_i, a_i))^2$
11     end
12     for all of the $n$ planning steps do
13         Sample a $m_p$-sized mini-batch of feature vectors $\{\mathbf{x}_i\}$ from the planning buffer.
14         Sample a $m_p$-sized mini-batch of actions $\{\mathbf{a}_i\}$ uniformly.
15         Apply value optimizer to the loss
             $\sum_i \frac{1}{2}(b_{\mathbf{v}}(\mathbf{x}_i, \mathbf{a}_i) + \gamma(1 - t_{\mathbf{z}}(\mathbf{x}_i, \mathbf{a}_i)) \max_a \hat{q}_{\tilde{\mathbf{w}}}(f_{\mathbf{u}}(\mathbf{x}_i, \mathbf{a}_i), a) - \hat{q}_{\mathbf{w}}(\mathbf{x}_i, \mathbf{a}_i))^2$
16     end
17     $\mathrm{s} \leftarrow \mathrm{s}'$
18     $t \leftarrow t + 1$
19     if $t \% k == 0$ then
20         $\tilde{\mathbf{w}} = \mathbf{w}$
21     end
22 end
```

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 $\alpha \in [5 \times 10^{-7}, 10^{-6}, 5 \times 10^{-6}, 10^{-5}, 5 \times 10^{-5}]$
	Model optimizer	Adam, step-size $\beta \in [10^{-6}, 5 \times 10^{-6}, 10^{-5}, 5 \times 10^{-5}, 10^{-4}]$
	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 6.1. Tested hyperparameters in the non-adaptive deep Dyna-Q experiments.

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 6.2. Experiment setup used to test the non-adaptive deep Dyna-Q.

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 6.3. Best hyperparameter setting for the non-adaptive deep Dyna-Q experiments.

6.2. Local Forgetting (LoFo) Replay Buffer

We propose LoFo (*Lo*cal *Fo*rgetting) replay buffer, a conceptually simple variation of the traditional FIFO replay buffer that is able to address the core challenges in building an adaptive deep MBRL method. Instead of removing the oldest samples from the replay buffer once it is full as done in the traditional FIFO replay buffer, in a LoFo replay buffer, the oldest samples in only the *local neighbourhood* of the new samples are removed.

When a change occurs in the environment and the agent observes that change, adding the new samples to the replay buffer and removing the oldest samples from only the local neighbourhood of the new data instead of the oldest samples from the entire replay buffer facilitates removal of the potentially incorrect and stale data sooner. This helps mitigate the interference of the old out-of-date samples with the new samples when tasks change.

Since only the samples in a local neighbourhood of the new samples are removed, old samples in other parts of the state space including those that have not been visited recently remain in the replay buffer. This leads to a replay buffer whose samples are approximately spread out equally throughout the entire relevant state space, irrespective of the current behavior policy. Updating the world model with samples drawn from this LoFo replay buffer results in a world model that is approximately accurate throughout the relevant state-space avoiding the problem of catastrophic forgetting of predictions related to states not recently visited.

LoFo replay buffer can therefore address the interference-forgetting dilemma and result in learning a world model that is sufficiently accurate throughout the relevant state-space even when there are environment changes. The current deep MBRL methods can then utilize the correct world models to update and adapt their policies, resulting in an adaptive deep MBRL method.

6.3. Contrastive State Locality

An instantiation of a LoFo replay buffer requires a definition of a local neighbourhood to an observed new sample and a way to determine which samples in the replay buffer are within that local neighbourhood. While there could be several ways to do this, in this work, we learn a state locality function using a contrastive learning technique which is then used to both define the local neighbourhood and also identify samples within that local neighbourhood.

Using contrastive learning (Hadsell et al., 2006; Dosovitskiy et al., 2014; Wu et al., 2018), we learn neural embedding representation of states such that states that are temporally closer, i.e., reachable with fewer actions are also closer in their neural embedding representation. Specifically, we learn an embedding function $\boldsymbol{v} = f_{\boldsymbol{\theta}}(s)$, where $f_{\boldsymbol{\theta}}$ is a deep neural network that maps state s to an embedding vector $\boldsymbol{v}$. This embedding function induces a distance metric in the state-space, such as $d_{\boldsymbol{\theta}}(s_i, s_j) = ||f_{\boldsymbol{\theta}}(s_i) - f_{\boldsymbol{\theta}}(s_j)||_2$ for states s_i and s_j. We train the embedding function such that states that are temporally closer, have a smaller distance between the embeddings learnt and states that are temporally farther, have a relatively larger distance between them in the embedding space.

Let s' represent a state that is one action away from state s, and $\bar{\boldsymbol{s}}$ represent a set of states that are not. Let $\mathbb{D} = \{s, s', \bar{\boldsymbol{s}}\}$ be a dataset of their collection. We train the embedding function to minimize the following loss function:

$$L(\mathbb{D}) = \sum_{(s,s',\bar{\boldsymbol{s}}) \in D} ||f_{\boldsymbol{\theta}}(s) - f_{\boldsymbol{\theta}}(s')||_2^2 + \left(\beta - \Sigma_{\bar{s} \in \bar{\boldsymbol{s}}} ||f_{\boldsymbol{\theta}}(s) - f_{\boldsymbol{\theta}}(\bar{s})||_2^2\right)^2, \tag{6.3.1}$$

where $\beta > 0$ is a hyperparameter. This loss function trains the embedding of states s and s' that are temporally next to each other to be closer by minimizing their distance towards zero, while pushing the embedding of states $\bar{\boldsymbol{s}}$ that are not temporally next to s to be on average farther, with a cumulative squared distance value close to β, a positive number.

In our experiments, we collect trajectories using a random behavior policy at the beginning of training and use samples from that trajectory to form the dataset $\mathbb{D} = \{s, s', \bar{\boldsymbol{s}}\}$ to train our embedding function $f_{\boldsymbol{\theta}}$. The learnt embedding function is then fixed and the distance between the state embeddings is used as a proxy for state locality. In complex environments with exploration challenges, a random behavior policy might be insufficient to cover the relevant state space needed to learn a good state locality estimate. It is an important future work to figure out good ways to learn state locality in such settings.

The samples stored in the replay buffer are generally of the form (s, a, r, s'), representing a transition an agent experienced by taking an action a, from the state s and moving to a state s' and obtaining a reward r. In a LoFo replay buffer, upon taking an action in the environment, the generated experience sample is used to first gather all the samples in the local neighbourhood of the new sample. This is done by estimating the state locality using the distance $d_{\boldsymbol{\theta}}(s, s_i) = ||f_{\boldsymbol{\theta}}(s) - f_{\boldsymbol{\theta}}(s_i)||_2$ between state s in the new sample and states $\{s_i\}$ corresponding to starting states in all of the samples in the replay buffer. The samples in the replay buffer whose starting state's distance to s, $d_{\boldsymbol{\theta}}(s, s_i) < D_{local}$ are selected. D_{local} is a scalar hyperparameter that determines the size of the local neighbourhood around any given state.

If the number of samples within the local neighbourhood is equal to or above a threshold N_{local}, then the oldest sample in that local neighbourhood is removed. N_{local} is a positive integer hyperparameter that determines the maximum number of samples that are stored in the replay buffer within any local neighbourhood of a given sample. The new sample is now added to the LoFo replay buffer.

It is worth noting a few technical details of the LoFo buffer's implementation in this work. First, $\bar{\boldsymbol{s}}$ for each (s, s') is made via random sampling of states from the collected transitions other than s and s'. Second, since the learnt embedding function is kept fixed during the entire training, embeddings of the states are cached and stored in the LoFo replay buffer as an additional element. Lastly, to form the local neighbourhood around the incoming sample, we first calculate the distance between its embedding and all other stored embeddings, then we filter only those samples that fulfill the $d_{\boldsymbol{\theta}}(s, s_i) < D_{local}$ condition. This strategy is arguably the simplest form of finding the nearest neighbours and would cost $O(\text{buffer size})$ comparisons.

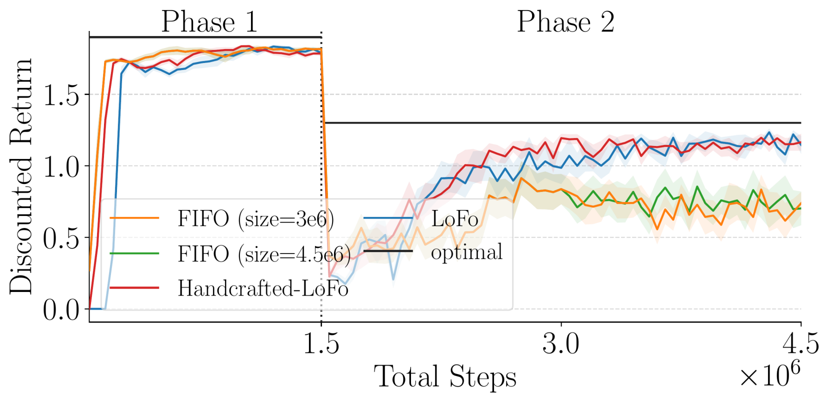
6.4. Adaptive Deep Dyna-Q

In this section, we show that replacing the traditional FIFO replay buffer used in Algorithm 6.1.1 (Section 6.1) with the LoFo replay buffer makes deep Dyna-Q adaptive. We evaluate deep Dyna-Q with LoFo replay buffer on the LoCA setup of two domains. First, the LoCA setup of the MountainCar domain (MountainCarLoCA, Figure 3.2 (a)). Second, the LoCA setup of a variant of the simple Mini-grid domain (MiniGridLoCA, Figure 3.2 (b)). Our empirical results show that deep Dyna-Q with a LoFo replay buffer is successfully able to adapt to the local environment changes in the LoCA setup in both domains.

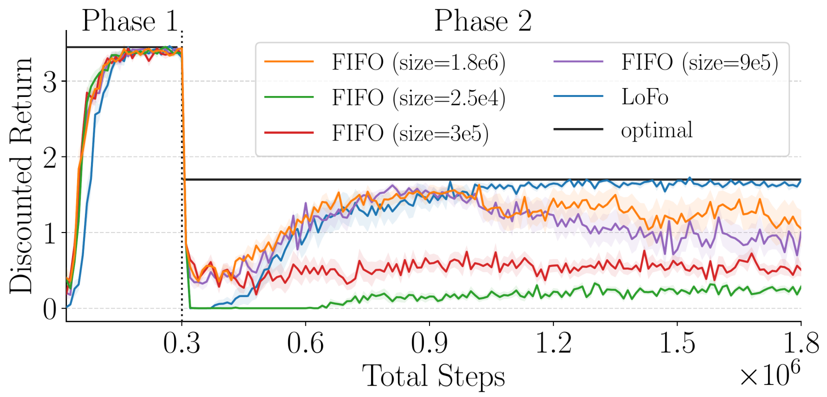
Previously, we found that using the same learning and planning buffer size resulted in the best performance for the deep Dyna-Q agent using FIFO buffers. Therefore, we use the same buffer sizes for our baseline experiments in this section. Furthermore, we changed the deep Dyna-Q's world model's network architectures. Previously in Section 6.1, we maintained separate networks for different actions with no shared parameters to avoid interference among other actions. However, keeping many different networks (action-sized) is not feasible for many high-dimensional real-world domains and results in poor generalization. So, for the experiments in this section, we first encode a given state to a low-dimensional vector for the various parts of the model (dynamics, reward, and termination model). Then, we concatenate the given action to the resulting vector and feed it to the MLP layers.

We start off focusing on the MountainCarLoCA domain. Table 6.4 shows the experiment setup we used to evaluate the deep Dyna-Q agent's adaptivity under the LoCA setup. Table 6.5 shows the final values of the hyperparameters used in the LoFo replay buffer for the MountainCarLoCA domain. We searched over $\beta \in \{1, 10\}$, $D_{local} \in \{0.01, 0.005\}$, and $N_{local} \in \{1, 2, 5, 10\}$. Furthermore, table 6.6 summarizes the important hyperparameters for the deep Dyna-Q method on the MountainCarLoCA. All the deep Dyna-Q agents (one with the LoFo replay buffer and the baseline agents with FIFO buffers) used these hyperparameters, and they only differ in the choice of the replay buffer.

Figure 6.2 (a) shows the learning curves of different versions of deep Dyna-Q on MountainCarLoCA. The best replay buffer size for the baseline deep Dyna-Q with FIFO replay buffer is 4.5×10^6, the one that stores all the samples seen so far. We observe that while all the methods reach close to optimal performance in Phase 1, only the method with LoFo replay buffer is able to adapt to reward change that happens in Phase 2 and reach close to optimal performance in Phase 2, making it an adaptive MBRL method as per the LoCA evaluation.



(a)



(b)

Fig. 6.2. Plots showing the learning curves of deep Dyna-Q with a LoFo replay buffer (adaptive) and FIFO replay buffers of different sizes (not adaptive) on (a) MountainCarLoCA and (b) MiniGridLoCA. Each learning curve is an average discounted return over ten runs, and the shaded area represents the standard error. The maximum possible return in each Phase is represented by a solid black line. (a) Note that the Handcrafted-LoFo refers to a variant of the LoFo replay buffer that, instead of learning the state locality function, uses a handcrafted locality function.

The two-dimensional state-space of the MountainCar domain allows visualization of properties associated with states throughout the state-space on a 2D plot. Figure 6.3 shows the 2D histogram of the states across the state-space whose transition samples are stored in the LoFo replay buffer and the traditional FIFO replay buffer at the end of Phase 1 and Phase 2. We observe that in a FIFO replay buffer, at the end of Phase 2, almost all of the samples in the buffer are just from a small region in the state-space, that corresponds to the T1-zone. Samples of states from other parts of the state-space that were present in Phase 1 have mostly been removed from the buffer. This leads to catastrophic forgetting of model prediction for states in parts of the state-space outside the T1-zone. In LoFo replay buffer however, even at the end of Phase 2, samples are maintained from across the entire relevant state-space, enabling maintaining accurate world model for the entire relevant state-space. Also, LoFo replay buffer stores only about 3×10^4 samples in total at the end of Phase 2, compared to 4.5×10^6 samples stored in the FIFO replay buffer.

Additionally, we tested another variant of the LoFo replay buffer that uses a handcrafted state locality function instead of using the learned contrastive state locality function. Since the first dimension of the MountainCarLoCA state-space indicates the position and the second one the velocity of the car ($s : (x, v)$), we can scale the range of the velocities to match the scale of the positions and use the euclidean distance as the locality function:

$$d_{\text{handcrafted}}(s_i, s_j) = \sqrt{(x_i - x_j)^2 + 150 \times (v_i - v_j)^2}$$

. The learning curve of the deep Dyna-Q agent that uses the LoFo replay buffer with the handcrafted state locality function in the MountainCarLoCA domain is also presented in Figure 6.2 (a). We observe that the LoFo replay buffer with the learned state locality function is able to match the performance of the buffer with the handcrafted state locality function.

Next, we focus on the MiniGridLoCA domain. Table 6.7 shows the experiment setup we used to evaluate the deep Dyna-Q agent's adaptivity under the LoCA setup. Table 6.8 shows the final values of the hyperparameters used in the LoFo replay buffer for the MiniGridLoCA domain. Furthermore, we only searched over $N_{local} \in \{50, 75, 100, 150\}$. While we started with the same β and D_{local} as the MountainCarLoCA, we found that by decreasing the D_{local} to 0.001, the learned state locality function can make a clear distinction between all possible states (256). The algorithmic design of the deep Dyna-Q agent we used for the MiniGridLoCA domain is the same as that used for the MountainCarLoCA. We only changed the neural network architecture for the model and the action-value estimator. Table 6.9 summarizes the architecture of the neural networks. Finally, Table 6.10 shows the hyperparameters used for the deep Dyna-Q agent on the MiniGridLoCA domain when the agent used the LoFo replay buffer and the traditional FIFO replay buffer.

Figure 6.2 (b) shows the learning curves of different versions of deep Dyna-Q on MiniGridLoCA. We compare deep Dyna-Q using LoFo replay buffer with the variants that use FIFO replay buffers with different buffer sizes, ranging from buffers that store all the samples seen (size = 1.8×10^6) to the buffer that stores only as many samples as LoFo replay buffer uses (size = 2.56×10^4). We observe that only deep Dyna-Q with LoFo replay buffer is able to adapt to changes in Phase 2 and converge to close to optimal performance. The method achieves adaptivity while storing only around 2.56×10^4 samples in the replay buffer, two orders of magnitude less samples compared to the best performing FIFO replay buffer based method which is of size 1.8×10^6. Note that among the methods that use FIFO replay buffers, two methods with largest buffer sizes, one with a buffer of size 9×10^5 and the other that stores all the samples with a size of 1.8×10^6 are initially able to reach close to optimal performance in Phase 2, but their performance degrades over time afterwards. This is because, the total number of distinct states in the MiniGridLoCA environment are less (256 to be precise) and therefore samples from all over the state-space from Phase 1 stay longer in a large replay buffer. There is a sweet spot when there are still samples from throughout the state-space from Phase 1, while the proportion of samples from Phase 2 that capture the reward change in the T1-zone are much higher than the stale rewards of T1 from Phase 1 avoiding any serious interference. This can lead to a performance that is temporarily close to optimal performance but quickly degrades as samples from Phase 1 are removed in the FIFO buffer and catastrophic forgetting happens.

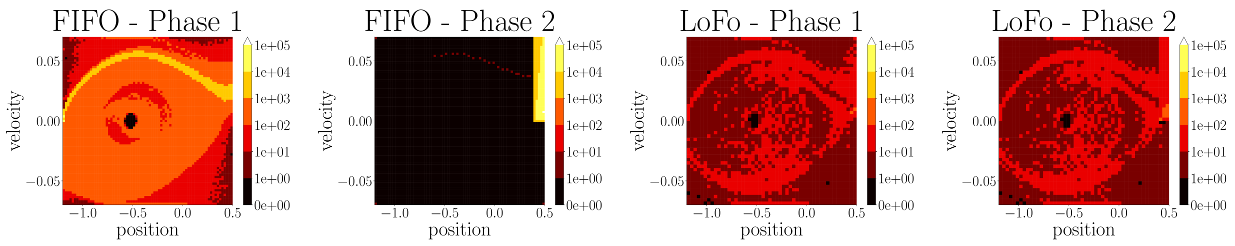


Fig. 6.3. Histogram of the states across the state-space of MountainCarLoCA environment whose transition samples are stored in the LoFo replay buffer and the traditional FIFO replay buffer (size = 3×10^6) at the end of Phase 1 and Phase 2. While LoFo replay buffer maintains samples from states throughout the relevant state-space in both Phase 1 and Phase 2, FIFO replay buffer removes almost all samples from states other than the T1-zone by the end of Phase 2.

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 Phase 2 steps	1.5×10^6 3×10^6
Other details	Maximum number of steps before an episode terminates	500
	Training steps between two evaluations	10^4
	Number of runs	10
	Number of evaluation episodes	10

Table 6.4. Experiment setup used to test the adaptive deep Dyna-Q on the MountainCarLoCA domain.

Embedding network architecture	MLP:$[64 \times 64 \times 64 \times 16]$, Activation Function: *tanh*
Optimizer	Adam, learning rate: 10^{-4}
β	10
Number of negative samples	128
Mini-batch size	32
Total number of random steps for creating dataset $\mathbb{D}$	100000
Number of training epochs	5
D_{local}	0.005
N_{local}	1

Table 6.5. Hyperparameters used for the LoFo replay buffer on the MountainCarLoCA domain.

Neural networks	Dynamics model	MLP with *tanh*, $[64 \times 64 \times 63 \times 64 \times 64]$,
	Reward model	MLP with *tanh*, $[64 \times 64 \times 63 \times 64 \times 64]$,
	Termination model	MLP with *tanh*, $[64 \times 64 \times 63 \times 64 \times 64]$,
	Action-value estimator	MLP with *tanh*, $[64 \times 64 \times 64 \times 64]$,
Optimizer	Value optimizer	Adam, learning rate: 5×10^{-6}
	Model optimizer	Adam, learning rate: 5×10^{-5}
Other details	Exploration parameter	Epsilon greedy $\epsilon = 0.5$
	Number of random steps before training	50000
	Target network update frequency	500
	Number of model learning steps	5
	Number of planning steps	5
	Mini-batch size of model learning	32
	Mini-batch size of planning	32

Table 6.6. Hyperparameters used for the adaptive deep Dyna-Q on the MountainCarLoCA domain.

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 (2×2 subgrid)
	Phase 2 evaluation	Uniform distribution over the entire state-space
Training steps	Phase 1 steps	3×10^5
	Phase 2 steps	1.5×10^6
Other details	Maximum number of steps before an episode terminates	100
	Training steps between two evaluations	10^4
	Number of runs	10
	Number of evaluation episodes	10

Table 6.7. Experiment setup used to test the adaptive deep Dyna-Q on the MiniGridLoCA domain.

Embedding network architecture	CNN: (Channels:$[32 \times 64 \times 64]$ Kernel Sizes:$[8 \times 3 \times 3]$ Strides:$[4 \times 2 \times 2]$), Followed by MLP:$[512 \times 16]$, Activation Function: *relu*
Optimizer	Adam, learning rate: 10^{-4}
β	10
Number of negative samples	128
Mini-batch size	32
Total number of random steps for creating dataset $\mathbb{D}$	25000
Number of training epochs	5
D_{local}	0.001
N_{local}	1

Table 6.8. Hyperparameters used for the LoFo replay buffer on the MiniGridLoCA domain.

Neural networks	Dynamics model	CNN: (Channels:$[32 \times 64 \times 64]$ Kernel Sizes:$[8 \times 3 \times 3]$ Strides:$[4 \times 2 \times 2]$), Followed by Transposed CNN: (Channels:$[64 \times 32 \times 3]$ Kernel Sizes:$[6 \times 6 \times 5]$ Strides:$[1 \times 4 \times 3]$), Activation Function: *relu*
	Reward model	CNN: (Channels:$[32 \times 64 \times 64]$ Kernel Sizes:$[8 \times 3 \times 3]$ Strides:$[4 \times 2 \times 2]$), Followed by MLP:$[512]$, Activation Function: *relu*
	Termination model	CNN: (Channels:$[32 \times 64 \times 64]$ Kernel Sizes:$[8 \times 3 \times 3]$ Strides:$[4 \times 2 \times 2]$), Followed by MLP:$[512]$, Activation Function: *relu*
	Action-value estimator	CNN: (Channels:$[32 \times 64 \times 64]$ Kernel Sizes:$[8 \times 3 \times 3]$ Strides:$[4 \times 2 \times 2]$), Followed by MLP:$[512]$, Activation Function: *relu*

Table 6.9. Neural networks' architecture for the deep Dyna-Q on the MiniGridLoCA domain.

Optimizer	Value optimizer	Adam, learning rate: 6.25×10^{-5}
	Model optimizer	Adam, learning rate: 10^{-4}
Other details	Exploration parameter	Epsilon greedy $\epsilon = 0.5$
	Number of random steps before training	2000
	Target network update frequency	5000
	Number of model learning steps	1
	Number of planning steps	1
	Mini-batch size of model learning	128
	Mini-batch size of planning	128

Table 6.10. Final hyperparameters for the deep Dyna-Q on the MiniGridLoCA domain.

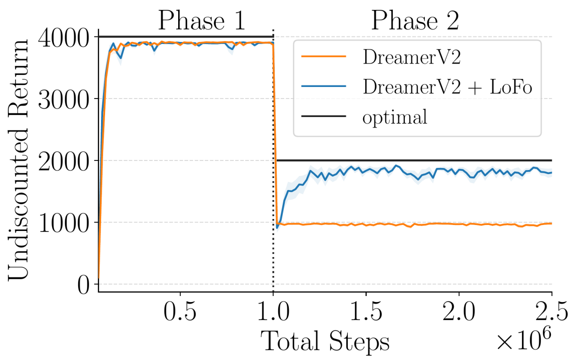
6.5. Adaptive PlaNet and DreamerV2

To see if more complex methods can be made adaptive as well by using a LoFo replay buffer, we applied it to the deep MBRL methods PlaNet (Hafner et al., 2019b) and DreamerV2 (Hafner et al., 2020) and evaluated their performance. PlaNet and DreamerV2 use a recurrent model for reward and transition predictions which require sample-sequences to make updates to rather than individual samples. Therefore, we first need to extend the concept of the LoFo replay buffer to sample-sequences. On a high level, what we aim to achieve is that updates made to the world model come from states spread out more-or-less equally across the state-space. In the case of sample-sequences, we try to approximate this by ensuring that the start-state of each sequence is drawn approximately at random across the state-space.

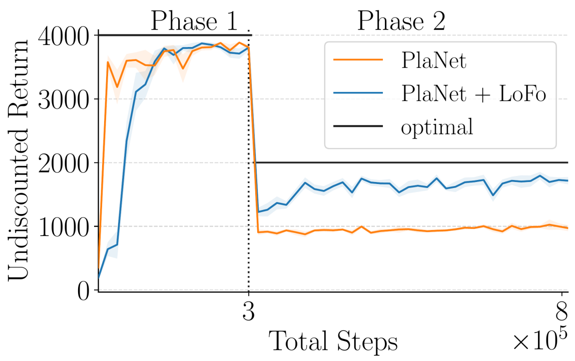
Our approach to achieve good coverage of sequence start-states is to use two separate buffers: a *state-buffer* which is curated similarly to the LoFo replay buffer, and a *trajectory-buffer* storing observed trajectories. For now, assume that the trajectory-buffer stores all observed trajectories. In the Section 6.5.1, we show how the trajectory-buffer can be bounded to a maximum sample-size of $B \cdot N$, where B is the size of the state-buffer and N is the size of a sample-sequence used for updates. When a new sample (s_t, a_t, r_t, s_{t+1}) is observed, it is appended to the trajectory-buffer, while state s_t is added to the state-buffer. Crucially, if the state-buffer contains more states from the local neighborhood of state s_t than some threshold amount, the oldest state from this neighborhood is removed from the state-buffer. Each state in the state-buffer points to a copy of itself in the trajectory-buffer. When a state s_i is removed from the state-buffer, the corresponding reward r_i in the trajectory buffer is replaced by $None$, indicating that these rewards should no longer be used for training reward-predictions. The world model is updated by first sampling a random state s_u from the state-buffer and then drawing a sample sequence of size N from the trajectory-buffer, starting with state s_u. Whenever the reward is $None$, the loss of the corresponding reward prediction is set to 0.

We evaluate our modified versions of PlaNet and DreamerV2 on the LoCA setup applied to two domains. First, a variation on the Reacher domain, called ReacherLoCA (Figure 3.2 (c)). And second, a more complex extension of ReacherLoCA, which we refer to as the RandomizedReacherLoCA (Figure 3.2 (d)). For both domains, we observe that the LoFo replay buffer results in adapting to the local reward change in T1 and learning a sufficiently accurate reward model throughout the relevant state-space in Phase 2, reaching close to optimal performance.

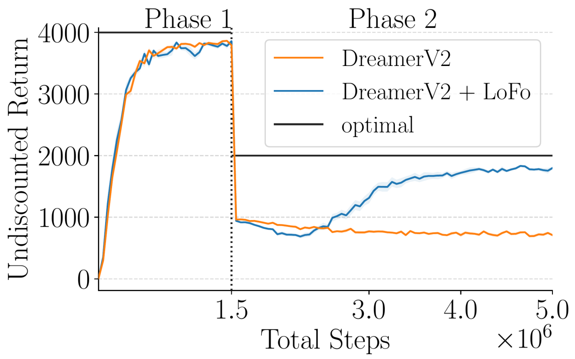
Table 6.11 shows the experiment setup we used to evaluate the PlaNet and the DreamerV2 agents' adaptivity under the LoCA setup of the ReacherLoCA domain. The experiment setup for the RandomReacherLoCA domain in which we only assessed the adaptivity of the DreamerV2 method is shown in Table 6.12. We did not evaluate the PlaNet method using the RandomizedReacherLoCA domain because attaining a close to optimal performance even in Phase 1 due to the online planning is computationally infeasible. We followed our findings in Section 4.2 for



(a) ReacherLoCA



(b) ReacherLoCA



(c) RandomizedReacherLoCA

Fig. 6.4. Plots showing the learning curves of DreamerV2 and PlaNet with a LoFo replay buffer (adaptive) and with FIFO replay buffers (not adaptive) on a, b) ReacherLoCA, and c) RandomizedReacherLoCA. Each learning curve is an average undiscounted return over ten runs, and the shaded area represents the standard error. The maximum possible return in each phase is represented by a solid black line.

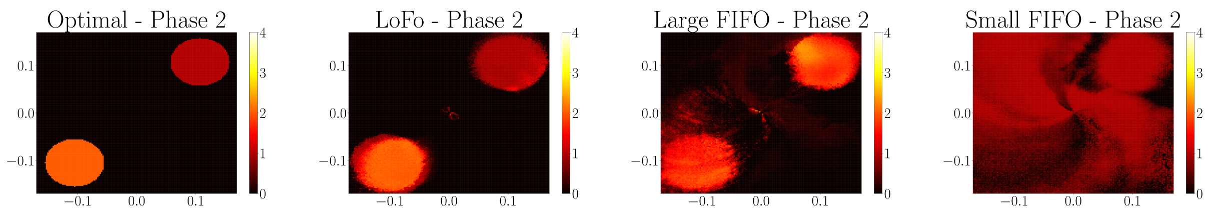


Fig. 6.5. Visualization of the estimated rewards from the DreamerV2 agent's reward model at the end of Phase 2. Each point on the heatmap represents the agent's position in the Reacher environment.

various hyperparameters and implementation details regarding the PlaNet and DreamerV2 methods. Table 6.13 shows the values of the hyperparameters used in the LoFo replay buffer for the ReacherLoCA domain where we searched over different values of $N_{local} \in \{2, 5, 10, 20, 40, 80\}$. Furthermore, the best hyperparameter setting for the LoFo replay buffer is mentioned in the same table where we searched among $N_{local} \in \{1, 2, 5, 10\}$.

Figure 6.4 shows the learning curves of DreamerV2 and PlaNet on ReacherLoCA, and DreamerV2 on RandomizedReacherLoCA. We observe that in both the setups all the methods reach close to optimal performance in Phase 1. However, in Phase 2 only DreamerV2 and PlaNet with a LoFo replay buffer is able to adapt to the environment change, making them adaptive deep MBRL methods. DreamerV2 and PlaNet with the traditional FIFO replay buffer fail to adapt in Phase 2.

In Figure 6.5 we visualize the reward predictions of the different DreamerV2 methods. We observe that DreamerV2 that uses the LoFo replay buffer for learning its reward model has adapted its reward predictions for target T1 (top right) correctly to around $+1$ in Phase 2. When we use a large FIFO replay buffer, we observe that the reward for target T1 at the end of Phase 2 is overestimated to around $+2.5$ because of the interference of stale samples from Phase 1. On the other hand, when we use a small FIFO replay buffer, DreamerV2's reward prediction at the end of Phase 2 for T1 is accurate around $+1$, but the model has completely forgotten the reward for target T2, and other parts of the state-space outside the T1-zone.

6.5.1. Bound for LoFo Replay Buffer with Recurrent Models

In this section, we show that *trajectory-buffer* can be bounded to a maximum sample-size of $B \times N$ in practice, where B is the size of the state-buffer and N is the size of a sample-sequence used for updates.

As mentioned in the beginning of Section 6.5, upon removing s_i from the state-buffer, r_i is set to $None$ in the trajectory-buffer and the agent never uses a sample-sequence starting from s_i. Now, if we look closely at the trajectory-buffer, only sample-sequences starting with states s_k where $k \in (i - N, i]$ contains the $(s_i, a_i, r_i = None, s_{i+1})$. Conceptually, we can remove (s_i, a_i, r_i, s_{i+1}) from the trajectory-buffer if $s_k, \forall k \in (i - N, i]$ are removed from the state-buffer, because in this case there remains no sample-sequence containing (s_i, a_i, r_i, s_{i+1}) for training the agent. By doing so, we argue that no N consecutive $None$ rewards can be found in the trajectory buffer. Because in that case, the last $None$ reward belongs to no valid sample-sequence, and therefore, it should have been removed from the trajectory-buffer.

Given that for each state in the state-buffer, we know their corresponding reward in the trajectory-buffer is not $None$, in the worst-case scenario, there can be at most $N - 1$ $None$ rewards after such samples in the trajectory-buffer. Hence, by counting them as well, each sequence in the trajectory-buffer starting from a state in the state-buffer can be at most of the length N. Therefore, the total number of samples stored in the trajectory-buffer would be at most $B \times N$.

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
Training steps	Phase 1 steps	PlaNet: 3×10^5, DreamerV2: 10^6
	Phase 2 steps	PlaNet: 5×10^5, DreamerV2: 1.5×10^6
Other details	Number of steps before an episode terminates	1000
	Training steps between two evaluations	PlaNet: 1.5×10^4, DreamerV2: 10^4
	Number of runs Number of evaluation episodes	10 PlaNet: 5, DreamerV2: 8

Table 6.11. Experiment setup for testing the PlaNet and the DreamerV2 method on the Reacher-LoCA domain.

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
Training steps	Phase 1 steps Phase 2 steps	1.5×10^6 3.5×10^6
Other details	Number of steps before an episode terminates Training steps between two evaluations Number of runs Number of evaluation episodes	1000 10000 10 8

Table 6.12. Experiment setup for testing the DreamerV2 method on the RandomizedReacher-LoCA domain.

Embedding network architecture	CNN: (Channels:$[32 \times 64 \times 128 \times 256]$ Kernel Sizes:$[4 \times 4 \times 4 \times 4]$ Strides:$[2 \times 2 \times 2 \times 2]$), Followed by MLP:$[512 \times 64, 32]$, Activation Function: *relu*
Optimizer	Adam, learning rate: 10^{-4}
β	50
Number of negative samples	128
Mini-batch size	32
Total number of random steps for creating dataset $\mathbb{D}$	10^5
Number of training epochs	5
D_{local}	0.05
N_{local}	ReacherLoCA: 10, RandomizedReacherLoCA: 2

Table 6.13. Hyperparameters used for the LoFo replay buffer on the ReacherLoCA and RandomizedReacherLoCA domains.

Chapter 7

Conclusion

In this thesis, we introduced an improved version of the LoCA setup, which is simpler, less sensitive to its hyperparameters, and can be more easily applied to stochastic environments. We then studied the adaptivity of two deep MBRL methods, PlaNet (Hafner et al., 2019b) and DreamerV2 (Hafner et al., 2019a, 2020), 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 model-based learning (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 the LoCA setup. The two first of these failure modes can be overcome by using appropriate environment models and planning techniques. However, we showed there exists an interplay between the last two failure modes, namely catastrophic forgetting and interference, for deep model-based methods, which is challenging to resolve.

To address the challenges with deep model-based methods, we proposed the LoFo replay buffer, whose samples are approximately spread equally across the state-space. Furthermore, we conducted various experiments to show that utilizing the LoFo replay buffer with deep MBRL methods can make them adapt effectively to local changes in the reward function. This is the first time–to the best of our knowledge–that this type of adaptivity has been shown for deep MBRL methods. This is an important step towards more practical real-world application of RL, since the stationary assumption does not always apply there.

7.1. Limitations and Future Directions

For adaptive deep MBRL, the general strategy behind the replay buffer variation we presented boils down to forgetting samples that are *spatially close, but temporally far* from currently observed samples. We believe this to be a good general strategy to achieve effective adaptation in a non-stationary world. However, the specific implementation of this principle will differ depending on the problem type and the non-stationarity considered. In this thesis, we considered only reward non-stationarity and domains where exploration is easy. In this scenario, learning a locality function during the initial learning phase and keeping it fixed thereafter is sufficient. However, when the transition dynamics is non-stationary, the locality function needs to be maintained and updated across time as well, as the distances between individual states can change over time. Finding solutions for dynamically learning and updating the locality function is one of the future next steps.

Now that we have shown that adaptive deep MBRL is possible in principle, a logical next step for future work is scaling up these methods to larger and harder domains. Because while DreamerV2 has shown to be able to achieve good single-task performance on such domains, it is not a given that our replay buffer variation is sufficient for making it adaptive for such domains as well. For example, the ReacherLoCA we considered has a fairly small decision horizon (i.e., how far an agent needs to plan ahead to construct a good policy—for ReacherLoCA it takes on average about 25 actions to reach a goal state). Achieving adaptivity for longer decision horizons puts higher demands on the planning routine as well (see Section 4.1 for planning-related pitfalls that impede adaptivity), so it might be needed to make changes to DreamerV2's planning routine as well.

Lastly, we think the LoFo replay buffer could also be used beyond the scope of model-based learning and adaptation capabilities. One possible future direction is to consider the replay-based RL methods and replace their replay buffer, commonly a FIFO, with a LoFo buffer. That being said, we can explore if using the LoFo buffer leads to improved performance or a much smaller bound in terms of the size of the saved transitions. However, a crucial point for using the LoFo buffer on a larger scale is to increase the speed of finding the local neighbourhood of a given transition in the buffer. Currently, our strategy is the simplest form of nearest-neighbour methods which is of quadratic complexity. Therefore, we might need to revisit how the local neighbourhoods are determined in the LoFo buffer.

References

Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017.

Richard Bellman. Dynamic programming, system identification, and suboptimization. *SIAM Journal on Control*, 4(1):1–5, 1966.

Dimitri Bertsekas. *Dynamic programming and optimal control: Volume I*, volume 1. Athena scientific, 2012.

Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

Maxime Chevalier-Boisvert, Lucas Willems, and Suman Pal. Minimalistic gridworld environment for gymnasium. `https://github.com/Farama-Foundation/MiniGrid`, 2018.

Kurtland Chua, Roberto Calandra, Rowan McAllister, and Sergey Levine. Deep reinforcement learning in a handful of trials using probabilistic dynamics models. *Advances in neural information processing systems*, 31, 2018.

Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

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.

Alexey Dosovitskiy, Jost Tobias Springenberg, Martin Riedmiller, and Thomas Brox. Discriminative unsupervised feature learning with convolutional neural networks. *Advances in neural information processing systems*, 27, 2014.

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

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

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

Ian J Goodfellow, Mehdi Mirza, Da Xiao, Aaron Courville, and Yoshua Bengio. An empirical investigation of catastrophic forgetting in gradient-based neural networks. *arXiv preprint arXiv:1312.6211*, 2013.

David Ha and Jürgen Schmidhuber. World models. *arXiv preprint arXiv:1803.10122*, 2018.

Raia Hadsell, Sumit Chopra, and Yann LeCun. Dimensionality reduction by learning an invariant mapping. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 2, pages 1735–1742. IEEE, 2006.

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*, pages 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.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, et al. Model-based reinforcement learning for atari. *arXiv preprint arXiv:1903.00374*, 2019.

Ronald Kemker, Marc McClure, Angelina Abitino, Tyler Hayes, and Christopher Kanan. Measuring catastrophic forgetting in neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526, 2017.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8:293–321, 1992.

Fan-Ming Luo, Tian Xu, Hang Lai, Xiong-Hui Chen, Weinan Zhang, and Yang Yu. A survey on model-based reinforcement learning. *arXiv preprint arXiv:2206.09328*, 2022.

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

Tom M Mitchell and Tom M Mitchell. *Machine learning*, volume 1. McGraw-hill New York, 1997.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

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

Andrew William Moore. Efficient memory-based learning for robot control. Technical report, University of Cambridge, Computer Laboratory, 1990.

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.

Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

Ali Rahimi-Kalahroudi, Janarthanan Rajendran, Ida Momennejad, Harm van Seijen, and Sarath Chandar. Replay buffer with local forgetting for adaptive deep model-based reinforcement learning. *arXiv preprint arXiv:2303.08690*, 2023.

Arthur George Richards. *Robust constrained model predictive control*. PhD thesis, Massachusetts Institute of Technology, 2005.

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

Reuven Y Rubinstein. Optimization of computer simulation models with rare events. *European Journal of Operational Research*, 99(1):89–112, 1997.

Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

Stuart J Russell. *Artificial intelligence a modern approach*. Pearson Education, Inc., 2010.

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. *Nature*, 588(7839):604–609, 2020.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362 (6419):1140–1144, 2018.

Richard S Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Machine learning proceedings 1990*, pages 216–224. Elsevier, 1990.

Richard S Sutton. Planning by incremental dynamic programming. In *Machine learning proceedings 1991*, pages 353–357. Elsevier, 1991.

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 and Rich Sutton. A deeper look at planning as learning from replay. In *International conference on machine learning*, pages 2314–2322, 2015.

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 *Advances in Neural Information Processing Systems*, volume 33, pages 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.

Yi Wan, Ali Rahimi-Kalahroudi, Janarthanan Rajendran, Ida Momennejad, Sarath Chandar, and Harm H Van Seijen. Towards evaluating adaptivity of model-based reinforcement learning methods. In *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 22536–22561. PMLR, 17–23 Jul 2022. URL `https://proceedings.mlr.press/v162/wan22d.html`.

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.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256, 1992.

Zhirong Wu, Yuanjun Xiong, Stella X Yu, and Dahua Lin. Unsupervised feature learning via non-parametric instance discrimination. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3733–3742, 2018.