Policy Guided Planning in Learned Latent Space for Model-based Reinforcement Learning

Mohammad Amini

Computer Science
McGill University, Montreal

February 2, 2021

Dedication

This thesis is dedicated to my beautiful mom, who is mad at me for not visiting her during this time.

Acknowledgements

I am so grateful for having my family being there any time. Every time my mom and I talk together, I immediately realize nothing is ever more important than family. My dad would remind me of my inner strength and how I should keep going and trust myself in the face of adversity and it matters so much more to keep going when nothing seems to be working. My mom would be the support that no matter what I go through calms me and reminds me of what really matters.

I am extremely thankful for my supervisor Doina Precup for her amazing patience and optimism during all this time. Doina has pushed me so subtly that I usually only realize it after some time. I am extremely thankful to her that despite her busy schedule she would always make some time and listen to me and make sure I am set properly. She gave me an endless stream of opportunities. I hope, she is proud of me.

I would like to thank Sarath Chandar who has helped throughout this thesis and many personal life challenges in the last two years, without whom this thesis would not be possible. I have always wanted an elder brother and the universe has given me the best brother I could ever hope for. He has held me accountable to my ideal self and that is a gift I expect to keep receiving for the rest of my life.

I would like to extend my appreciation to Hannah Alsdurf for her amazing support. She has reviewed this thesis thoroughly and has provided me her valuable feedback. Every time I meet Hannah she reminds me of the very spirit that never gives up. She is the living embodiment of living life on her own terms while knocking out one obstacle after another and yet looking as if she is just having another chill day. I am extremely lucky to have met her.

My elder sister has been my rock, who has also reminded me that the beauty of life is this journey of going through one challenge at a time. My younger brother, Erfan, who has inspired me and made sure I am held accountable to growth. Him alone, has made sure I maintain being his role model and since he regularly levels up, I have to do too. Erfan makes sure I workout regardless of any excuse. I very much thank Sepehr who has reminded me numerous times how lucky I am to be where I am and how important it is to focus on the present moment and appreciate what I already have.

I am extremely thankful to Maude Godard, Who has been an enormous source of optimism in the toughest part of writing this thesis. She has helped me realize the importance of focusing on the goal and the positive side of the goal. She has brought forward the big picture in mind, that in return removed all stress and only made the journey full of enthusiasm and joy.

I really would like to thank Nithin Vasisth for his presence in my life. He is one of the most courageous men I have ever met and he has an immense resilience and persistence throughout any of his journeys. Also, I have become a better listener because of Nithin. Ardalan is a loyal friend who has stuck by me in thick and thin. No matter what we went through we could always just sit down and enjoy that moment we spend with each other. We could have the most laughable moments ever. Another really influential person in my life has been James Ough who has ideal combination of winner mindset and go-getter attitude. James is, without a doubt, a bundle of abundance of confidence and morale.

I would also love to thank all my friends and colleagues who helped and contributed in different forms: Chegini, R., Dutta, D., Ericson, E.,Faramarzi, M., Hamouni, P., Konar, A., Teru, K.

Abstract

Model-based Reinforcement Learning (MBRL) is one of the longstanding methodologies of Reinforcement Learning (RL). However, model-based RL algorithms have empirically been lagging behind model-free RL algorithms in asymptotic performance, particularly for complex tasks in which learning a perfect model of the task dynamics is hard. PlaNet, a recent model-based algorithm, combined decision-time planning in latent space with a learned dynamics model of the environment and achieved superior performance when compared to state-of-the-art model-free deep RL methods. Even though PlaNet is orders of magnitude more sample efficient, the decision time planning module performs a naive Cross Entropy Method (CEM) based search on the action space. In this thesis, we propose PG+, which improves upon PlaNet by learning a policy in the latent space and then using the policy while doing decision-time planning. We also propose an efficient way to add noise to the policy parameters that speeds up the CEM search in policy space. Our experimental results show that PG+ achieves better performance than PlaNet in all five continuous control tasks that we considered.

Résumé

L'apprentissage par renforcement basé sur un modèle (MBRL) est l'une des méthodologies importantes de l'apprentissage par renforcement (RL). Cependant, les algorithmes RL basés sur des modèles n'ont pas été empiriquement si performants que les algorithmes RL sans modèle, en particulier pour les tâches complexes dans lesquelles l'apprentissage d'un modèle parfait de la dynamique est difficile. PlaNet (Hafner et al., 2018), un algorithme récent basé sur un modèle, combine la planification du temps de décision dans l'espace latent avec un modèle de la dynamique appris par interaction avec l'environnement et a obtenu des performances supérieures par rapport aux méthodes de RL profonds sans modèle de pointe. Même si PlaNet est d'un ordre de magnitude plus efficace par rapport au montant de données, le module de planification du temps de décision effectue une recherche naïve basée sur la méthode d'entropie croisée (CEM) (Botev et al., 2013) sur l'espace d'action. Dans cette thèse, nous proposons PG+, qui améliore PlaNet en apprenant une politique dans l'espace latent et ensuite enutilisant la politique, tout en faisant la planification du temps de décision. Nous proposons également un moyen efficace d'ajouter du bruit aux paramètres de politique qui accélère la recherche de CEM dans l'espace des politiques. Nos résultats expérimentaux montrent que PG+ obtient de meilleures performances que PlaNet dans toutes les cinq tâches de contrôle continu que nous avons considerées.

Contents

1 Introduction

Designing autonomous decision making systems is one of the longstanding goals of Artificial Intelligence (AI). Reinforcement Learning (RL) is a sub-field of AI that attempts to design agents that can learn to make decisions. A typical RL agent interacts with the world by taking actions and receives feedback in the form of rewards. The goal of the agent is to maximize its expected reward over its lifetime. Traditional RL methods, while theoretically grounded, can exhibit poor scalability for complex tasks with high-dimensional observation space and/or complex environment dynamics. The pioneering work of Mnih et al. (2013) introduced Deep Reinforcement Learning (DRL), which uses powerful Deep Neural Networks (DNNs) as function approximators for RL. Deep reinforcement learning made it possible to design a superhuman agent that beat the human world champion in the game of Go (Silver et al., 2017).

RL algorithms can be broadly classified into model-free RL and model-based RL. Model-free RL algorithms aim to directly learn the policy to take actions given the current state of the world; most algorithms also learn the value function, which estimates the expected future return, in order to guide the search for a good policy. On the other hand, model-based RL algorithms learn a model of the world and use the learnt model to plan actions, as well as to update thee value function. Learning the model of the world is desirable for several reasons:

1. The model of the dynamics of the world is task-independent and hence can be used to solve future tasks in a sample efficient way.

2. Model based RL algorithms can quickly adapt to changes in the reward function, unlike model-free algorithms, since the world dynamics and the reward functions are learnt separately.

However, the asymptotic performance of model-based RL algorithms has traditionally been inferior to model-free RL algorithms in the high data regime, when using function approximation. Recently, Chua et al. (2018) proposed a model-based Deep RL algorithm called Probabilistic Ensembles with Trajectory Sampling (PETS) that achieved superior asymptotic performance when compared to existing state-of-the-art model-free RL methods. PETS uses the learnt model to perform decision-time planning using the Cross Entropy Method (CEM) (Botev et al., 2013). While achieving impressive performance, PETS has two major limitations:

- it assumes access to true reward functions which might not be available for all the tasks.

- it uses the low-dimensional latent state representation instead of high-dimensional observations; hence, is not clear if PETS can learn sufficiently good models.

Hafner et al. (2018) proposed PlaNet which alleviates both these limitations of PETS by learning the reward function and a stochastic variational RNN model that can deal with high-dimensional observations. PlaNet does CEM-based decision-time planning in the learned latent space; hence the planning still happens in an effectively low-dimensional space. This thesis builds upon PETS and PlaNet and proposes a new model-based RL algorithm called PG+.

The contributions of this thesis are as follows:

1. PETS has several arbitrary design choices and a lot of hyper-parameters. In this thesis, we provide a systematic empirical analysis of PETS to understand

the importance of several design choices. Our analysis helped us to improve PETS both in terms of final performance and sample-efficiency.

2. We propose PG+, a policy-guided model-based reinforcement learning which plans in a learned latent space. PG+ combines PlaNet, a decision-time planning algorithm, with a learned policy network in the learned latent space.

3. We also propose a scalable method to add noise to the policy parameters for efficient CEM search in policy space.

4. We empirically demonstrate that PG+ achieves better performance and stability as well as faster convergence than PlaNet in a suite of five DM-Control tasks.

The thesis is organized as follows. Chapter 2 reviews the basics of Reinforcement Learning (RL) and introduces the standard RL algorithms. Chapter 2 also discusses the differences between model-free RL and model-based RL. Chapter 3 discusses several recent model-based RL algorithms. Chapter 4 reports our empirical analysis of Probabilistic Ensembles with Trajectory Sampling (PETS) (Chua et al., 2018) and improvements to PETS in terms of performance and sample-efficiency. In Chapter 5, we introduce PG+, a model-based RL algorithm that combines the decision-time planning algorithm PlaNet (Hafner et al., 2018) with learned policy networks. Chapter 5 also contains an empirical evaluation of PG+, which demonstrates that PG+ performs better than PlaNet both in terms of final performance and sample-efficiency. Chapter 6 concludes the thesis and discusses avenues for future work.

Statement of contributions: PG+ is also presented in a recent paper that we published at the NeurIPS'2020 Deep Reinforcement Learning Workshop (Amini et al., 2020). I have carried out all the design, implementation, and empirical evaluation of this algorithm, with input from the co-authors. I have also written the bulk of the paper.

2

Reinforcement Learning

Reinforcement Learning (RL) studies the interaction of an agent with an environment. In this interaction, the goal of the agent is to maximize its expected reward by learning to map states to actions, but unlike in supervised learning, the agent is not told what actions to take at each state. The actions that the agent takes affect not only the immediate reward, but also all subsequent rewards, because they modify the state of the agent. Delayed reward signals and trial-and-error search are the two most interesting aspects of RL. Usually, the problem of RL is formalized through ideas borrowed from dynamical systems and control theory, especially from the optimal control of Markov Decision Processes (MDPs). We will now review the most important RL concepts that are necessary to put in context the work that we will present.

2.1 Markov Decision Process

Figure 2.1 shows an example of an agent interacting with the environment. The agent is the learner and everything outside the agent is known as the environment. The agent and the environment continually interact; the agent takes actions and the environment presents the agent with new situations and/or rewards. Concretely, the agent interacts with the environment at discrete time steps $t = 1, 2, 3, ...$ At each time step t, the environment outputs an observation which represents the state

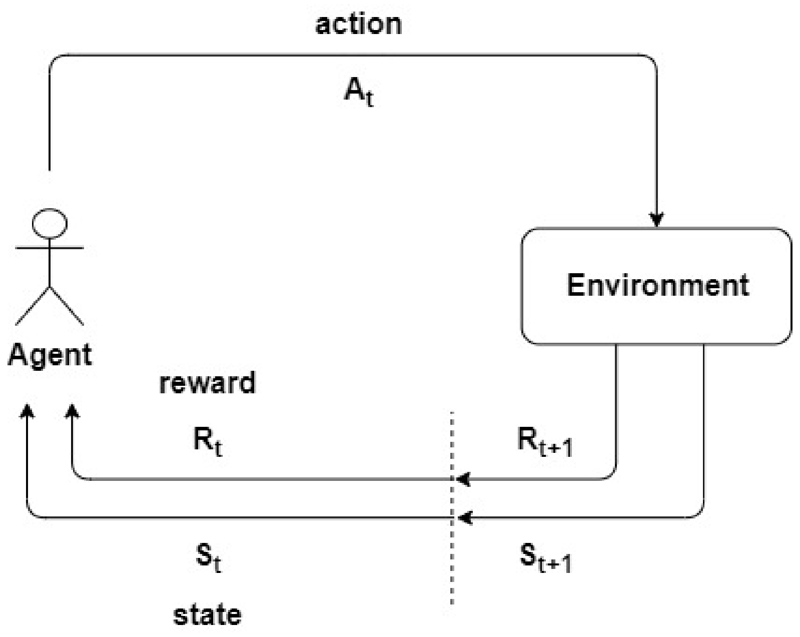


Figure 2.1: Interaction of an agent.

$S_t \in S$, where S is the set of all possible states. The agent takes an action $A_t \in A$, where A is the set of all possible actions, based on S_t. As a consequence, the environment returns a numerical reward $R_{t+1} \in \mathbb{R}$ and transitions the agent to the next state S_{t+1} . This sequence produces a trajectory τ:

$$\tau = S_0 A_0 R_1 S_1 A_1 R_2 S_3 ... \tag{2.1}$$

An observation O_t is a noisy version of the state. For example, a camera mounted on a mobile robot only collects the image of what is in front of it (the observation), but the full state would also include the robot's real position in the world, its velocity, as well as information about objects or people that might be behind it. A fully observable environment is one in which the agent is able to observe the full state of the world. In a partially observable environment, on the other hand, the agent only observes $O_t \neq S_t$.

A Markov Decision Process (MDP) is the mathematical formulation of choice for the RL problem, with which theoretical statements and guarantees are made. In a finite MDP, the sets of actions and states are finite; R_t and S_t are defined as random variables drawn from a probability distribution conditioned on the preceding state and action, $P\left(S_{t+1}, R_{t+1} \mid S_t = s, A_t = a\right)$. The Markovian assumption means that the preceding state captures the information from the entire history which is relevant to predict the next state and reward distribution, and hence we have:

$$P\left(S_{t+1}, R_{t+1} \mid S_t, A_t, S_{t-1}, A_{t-1}, ..., S_1, A_1\right) = P\left(S_{t+1}, R_{t+1} \mid S_t, A_t\right) \tag{2.2}$$

The first state of the world, S_0, is sampled from start-state distribution ρ_0 as:

$$S_0 \sim \rho_0(.) \tag{2.3}$$

where the dot (.) indicates a random variable. Every time an interaction between the agent and the environment takes place, a transition is made to a new state. The state transition function can be either stochastic:

$$S_{t+1} \sim P\left(. \mid S_t, A_t\right) \tag{2.4}$$

or deterministic:

$$S_{t+1} = f(S_t, A_t) \tag{2.5}$$

Note that the latter is a special case.

Environments with a discrete action space A provide a finite number of actions to the agent. On the other hand, in environments with continuous action space, actions are represented as real valued vectors.

2.2 Return

The reward is the scalar value received at every time step, dependent on the current state, current action, and sometimes the next state. Most often, we consider that $R_t = R(S_t, A_t)$. The return is defined as a cumulative function of the rewards obtained over a trajectory (i.e. a sequence of agent-environment interactions) and is denoted $G(\tau)$. The horizon T is the time duration of a trajectory. The return for a trajectory is defined as follows, for a finite and an infinite horizon, respectively:

$$G(\tau) = \sum_{t=0}^{T-1} R_t \tag{2.6}$$

and

$$G(\tau) = \sum_{t=0}^{\infty} \gamma^t R_t \tag{2.7}$$

The discount factor $\gamma \in (0, 1)$ is used to emphasize more recent rewards; γ close to 0 indicates a myopic and shortsighted agent that cares only about the immediate reward, and $\gamma = 1$ indicates an agent that assigns equal importance to rewards achieved now and later in time. In an infinite horizon setting, the return may not have a finite value if $\gamma = 1$, hence discounting (i.e. $\gamma < 1$) is required.

2.3 Policies

A policy is similar to a rule that tells the agent what actions to take in any given state. The terms "policy" and "agent" are sometimes used interchangeably. In a deterministic policy, the action A_t at any time step t is computed as:

$$A_t = \mu_\theta(S_t) \tag{2.8}$$

where μ is a deterministic function and θ is a set of parameters of this function. For stochastic policies, the action A_t at time step t is sampled from a distribution conditioned on the state:

$$A_t \sim \pi_\theta\left(. \mid S_t\right) \tag{2.9}$$

where π is a conditional probability distribution parameterized by θ.

Without any function approximation, policies would be represented as a table with one row for every state, containing the probability with which each action should be chosen in that state. However, for large and/or continuous state and action spaces, it is not possible to maintain this explicit table, so function approximators are used. In such cases, the policy function can be modeled using a simple logistic function or a deep neural network. The behaviour of the agent can be altered by changing the parameters of this policy function. A policy π is better than policy π' if its expected return is greater than that of π' for all states. When using function approximation, stochastic policies are often modelled as categorical distributions, for discrete action spaces, and diagonal Gaussian distributions, for continuous action spaces. In order to apply these policies, we need to be able to sample actions from the associated probability distribution.

2.4 Reinforcement Learning

The objective of reinforcement learning (RL) agents is to learn a policy that acts to maximize its expected return. If both the transition function and policy are stochastic, then the probability of a T-horizon trajectory is given by:

$$P\left(\tau \mid \pi\right) = \rho_0(S_0) \prod_{t=0}^{T-1} P\left(S_{t+1} \mid S_t, A_t\right) \pi\left(A_t \mid S_t\right) \tag{2.10}$$

And this leads to an expected return of:

$$J(\pi) = \int_{\tau} P\left(\tau \mid \pi\right) G(\tau) = \mathbb{E}_{\tau \sim \pi}[G(\tau)] \tag{2.11}$$

The optimal policy is given by the following equation:

$$\pi^* = \arg\max_{\pi} J(\pi). \tag{2.12}$$

Next, it is necessary to define value functions, which can be viewed as helper quantities that can direct the agent's search for an optimal policy.

2.5 Value Functions

The value of a state is defined as the expected return given that the agent starts in that state and takes actions from a particular policy. It is defined as:

$$V^{\pi}(s) = \mathbb{E}_{\tau \sim \pi}[G\left(\tau\right) \mid S_0 = s] \tag{2.13}$$

The value of a terminal state, if it exists, is always zero. V^{π} is called the state-value function for any policy π.

The value of a state-action pair is defined as:

$$Q^{\pi}(s,a) = \mathbb{E}_{\tau\sim\pi}[G(\tau) \mid S_0 = s, A_0 = a] \tag{2.14}$$

which is the expected return starting at state s and taking an action a, followed by taking actions from policy π. Q^{π} is the action-value function for policy π.

Solving an RL problem is the same as learning the policy that gets the agent the optimal expected return. To that end, it is useful to determine the value of each state under the optimal policy, known as the optimal value function.

$$V^{*}(s) = \max_{\pi} \; \mathbb{E}_{\tau\sim\pi}[G(\tau) \mid S_0 = s] \tag{2.15}$$

which is the expected value of starting at state s and always acting according to the optimal policy. The optimal action-value function is defined as:

$$Q^{*}(s,a) = \max_{\pi} \; \mathbb{E}_{\tau\sim\pi}[G(\tau) \mid S_0 = s, A_0 = a] \tag{2.16}$$

which is the expected value of starting at state s and take an action a, and forever after taking actions from the optimal policy.

These value functions are related through the following relationships

$$V^{\pi}(s) = \mathbb{E}_{a\sim\pi}[Q^{\pi}(s,a)] \tag{2.17}$$

$$V^{*}(s) = \max_{a} Q^{*}(s,a) \tag{2.18}$$

The optimal action-value function and optimal policy are also related as follows:

$$\pi^{*}(s) = \arg\max_{a} Q^{*}(s,a) \tag{2.19}$$

One last point to note is that there may be more than one action that maximizes $Q^{*}(s,a)$, in which case any of those actions is optimal. Yet, there will always be at least one optimal policy which selects an action deterministically .

Sometimes we want to determine how valuable one action is relative to other actions, on average, or the advantage of one action compared to other actions. The advantage function is defined as:

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s) \tag{2.20}$$

It is often used in policy gradient methods (see Section 2.7).

2.6 Value Function Estimation

As explained above, value estimation is one of the main methods for solving an RL problem. Since the future returns can depend on many factors other than the current state, value estimation can be very noisy, which in turn makes this problem hard. We will use past experience to estimate the value functions. Naturally, we can estimate the value of a state by averaging the returns we get from that state, a method known as Monte Carlo estimation. This is equivalent to using the sample average to estimate the expectation in Eq. (2.13). The Monte Carlo value estimates will converge to the true value of that state as the number of visits to that state tends to infinity. However, if that state is visited by only one trajectory, Monte Carlo estimates the value of that state to be the return from that single trajectory. Hence, with little data, the Monte Carlo estimate can have high variance, especially if the policy or environment are stochastic.

Lower variance estimates of value functions can be obtained by using the Markov property. For this, we can expand the value function definition as follows:

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi[G_t(\tau) \mid S_t = s] && (2.21)\\ &= \mathbb{E}_\pi[R(S_t) + \gamma G_{t+1}(\tau) \mid S_t = s] && (2.22)\\ &= \mathbb{E}_\pi[R(S_t) + \gamma \mathbb{E}_\pi[V^\pi(S_{t+1})] \mid S_t = s, S_{t+1} = s'] && (2.23)\\ &= \sum_a \pi(a|s) \sum_{s',r} P(s', r|s, a)(r + \gamma V^\pi(s')) && (2.24) \end{aligned}$$

where $R(S_t)$ is the expected reward from state S_t (slightly abusing notation) and $G_{t+1}(\tau)$ is the return of the trajectory τ starting from state $S_{t+1} = s'$. The expression above is the Bellman equation for V^π (Bellman, 1956). The Bellman equation shows the relationship between the value of a state and the value of its successor states. $Q^\pi(s, a)$ obeys a similar equation, conditioning on (s, a).

The optimal state value function and optimal action-value function also obey a set of recursive relationships, called Bellman optimality equations:

$$V^*(s) = \max_a \sum_{s',r} P(s', r|s, a)[r + \gamma V^*(s')] \tag{2.25}$$

and similarly for Q^*.

As shown above, the main difference between the Bellman equations for the fixed-policy value functions and the optimal value functions is the max operation over actions. This indicates that whenever the agent is given the choice among different actions, it has to choose the one that yields the highest return.

2.6.1 Dynamic Programming

Dynamic programming (DP) offers a set of methods to find the optimal policy, if the agent has access to the model of the environment. These are not the most efficient algorithms, since they require enormous computational resources, but they provide the necessary theoretical foundation for subsequent algorithms.

Policy iteration (Howard, 1960) constantly improves the policy to obtain the optimal state value function. In this algorithm, $V(s)$ is initialized to 0 and $\pi(s)$ is initialized to random for all states. Then, $V(s)$ is updated iteratively for all states using the following update rule, derived from the Bellman equation, until convergence:

$$V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} P(s', r \mid s, a)[r + \gamma V(s')] \tag{2.26}$$

This is known as the policy evaluation stage. Next, the policy $\pi(s)$ is updated for all the states using the following equation:

$$\pi(s) \leftarrow \arg\max_a \sum_{s',r} P(s', r \mid s, a)[r + \gamma V(s')] \tag{2.27}$$

This is known as policy improvement. If policy improvement results in any change in policy, then the policy evaluation and policy improvement steps are repeated until there is no change in the policy.

Policy iteration alternates these two main steps: 1) policy evaluation, and 2) policy improvement. Policy evaluation works to find the true values of all the states given the current policy. Policy improvement is greedifying the policy by choosing the action that maximizes the value of a given state.

Value Iteration (Howard, 1960) tries to perform both the above steps at once. Specifically, $V(s)$ is initialized to random values for all states, except the terminal states, whose values are set to 0. Then $V(s)$ for all states are updated iteratively, using the following update rule until convergence:

$$V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)[r + \gamma V(s')] \tag{2.28}$$

The final policy can be computed by using argmax:

$$\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r \mid s, a)[r + \gamma V(s')] \tag{2.29}$$

2.6.2 Temporal Difference Learning

Temporal Difference (TD) learning (Sutton, 1988) is an algorithms that combines Monte Carlo and dynamic programming. TD is similar to Monte Carlo in the sense that it does not require the model of the world, and similar to dynamic programming in the ability to bootstrap, which means using the current estimate of the value function as an approximation of future return. TD learning and Monte Carlo are usually referred to as sample updates, since they both require seeing the sampled trajectory of the agent, instead of knowing the model.

In short, Monte Carlo waits for the episode to finish and then uses that return to update the value estimates as follows:

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t(\tau) - V(S_t)] \tag{2.30}$$

where $G_t(\tau)$ is the return on trajectory τ from time t onward and $\alpha \in (0, 1)$ is the step size. The main difference between Monte Carlo and TD learning is that in TD learning, we do not have to wait until the end of an episode to have a target.

Rather, the target is already set after only one time step. The TD learning update is given by the following equation:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{2.31}$$

The difference between $V(S_t)$ and $(R_{t+1} + \gamma V(S_{t+1}))$ is called the TD-error.

As mentioned above, TD methods are preferred to DP methods since the requirement of having access to the model of the environment both for transitions and rewards is relaxed. TD methods are preferred to Monte Carlo methods since they can learn in a fully online and incremental fashion and they do not need to wait until the end of the episode, when the return is known.

2.6.3 Q-Learning

Q-Learning (Watkins and Dayan, 1992) is an off-policy TD control method, which aims to learn the optimal action value function from sampled transitions. The update equation for Q-Learning is:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{2.32}$$

where $\alpha \in (0, 1)$ is the step size. The only difference with the previous TD definition is that the state-value functions are replaced by action-value functions, and we assume that from state S_{t+1} onward, the agent will follow the policy which maximizes its current value estimates. The advantage of this method is that we can approximate the optimal action-value function without having access to transitions coming from a single policy. To reach convergence, the only requirement is that all state-action pairs keep getting updated.

2.6.4 Deep Q-Networks (DQN)

The Deep Q-Network (DQN) algorithm (Mnih et al., 2013) is an example of successfully deploying the Q-Learning algorithm to achieve impressive results, defeating human experts in Atari Games (Bellemare et al., 2013). DQN uses deep neural networks to approximate $Q(s, a)$ because storing value functions explicitly in large state spaces would not be feasible. Since DQN uses a form of function approximation typically used in supervised learning, in order to solve this RL problem, it needs to ensure that the data is decorrelated, as it would be if it obeyed the i.i.d (independently and identically distributed) assumption. The mechanism which ensures this decorrelation is called Experience Replay. Experience Replay refers to storing all transitions observed by the agent in a buffer and then taking batches at random among these samples during the training process, in order to break the correlation between the transitions within one trajectory. This also has been shown to be an efficient method to reuse the samples.

Using the same deep neural network to learn and compute target action-values makes the problem highly non-stationary. DQN solves this issue by using a separate target network to compute the target values, which makes the learning stable. The weights of the action-value network are copied onto the target network after every k updates, where k is a hyper-parameter.

2.7 Policy Optimization

Policy optimization is a family of RL algorithms that try to learn $\pi_\theta(a|s)$, via one of two main methods. The first is to directly optimize for $J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[G(\tau)]$ using gradient ascent. The second method is to indirectly maximize local approximations of $J(\pi_\theta)$. Normally this optimization is done in an on-policy fashion. On-policy indicates that the rollouts are generated by acting in the world using the latest version of the policy. On the other hand, off-policy indicates that the data used to learn comes from a different policy than the one we are trying to optimize. The most famous algorithms of the policy optimization family are Policy Gradient (Williams, 1992), Actor Critic methods (Sutton and Barto, 2018), Trust Region

Policy Optimization (Schulman et al., 2015), and Proximal Policy Optimization (Schulman et al., 2017).

2.7.1 Policy Gradient

To derive the vanilla policy gradient algorithm, we assume a stochastic parameterized policy π_θ and a finite action space. Intuitively, the goal is to increase the probability of the actions with a higher expected return. The aim is to use gradient ascent to optimize the parameters θ of the policy, as follows:

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\pi_\theta)|_{\theta_k} \tag{2.33}$$

The derivative term $\nabla_\theta J(\pi_\theta)$ is a stochastic approximation whose expectation estimates the gradient of the performance measure with respect to the parameters of the policy, referred to as the *policy gradient*. Algorithms that optimize the policy using this term are called policy gradient algorithms. The main benefit of this family of methods is that as long as the policy gradient term exists and is finite, the policy can be improved. Using a stochastic policy also helps in exploration.

Using elementary calculus and rearranging terms, we can derive the policy gradient from first principles, for the undiscounted, finite horizon return, as follows:

1. The probability of an episode $\tau = (s_0, a_0, ..., s_T)$ where actions are sampled from policy π_θ is:

$$P(\tau|\theta) = \rho_0(S_0) \prod_{t=0}^{T} P(S_{t+1}|S_t, A_t)\pi_\theta(A_t|S_t) \tag{2.34}$$

2. Since we know the derivative of $\log x$ with respect to x is $\frac{1}{x}$, when it is applied through the chain rule we get:

$$\nabla_\theta P(\tau|\theta) = P(\tau|\theta)\nabla_\theta \log P(\tau|\theta) \tag{2.35}$$

3. The log probability of a trajectory is:

$$\log P(\tau|\theta) = \log \rho_0(S_0) + \sum_{t=0}^{T}(\log P(S_{t+1}|S_t, A_t) + \log \pi_\theta(A_t|S_t)) \tag{2.36}$$

Since the environment and parameters θ have no relationship, the gradients of $\rho_0(S_0)$, $P(S_{t+1}|S_t, A_t)$ are zero, so the gradient of the log probability of a trajectory is:

$$\nabla_\theta \log P(\tau|\theta) = \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(A_t|S_t) \tag{2.37}$$

The derivation of the policy gradient can be written as:

$$\begin{aligned}
\nabla_\theta J(\pi_\theta) &= \nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}[G(\tau)] && (2.38)\\
&= \nabla_\theta \int_\tau P(\tau|\theta)G(\tau) && (2.39)\\
&= \int_\tau \nabla_\theta P(\tau|\theta)G(\tau) && (2.40)\\
&= \int_\tau P(\tau|\theta)\nabla_\theta \log P(\tau|\theta)G(\tau) && (2.41)\\
&= \mathbb{E}_{\tau\sim\pi_\theta}[\nabla_\theta \log P(\tau|\theta)G(\tau)] && (2.42)\\
\nabla_\theta J(\pi_\theta) &= \mathbb{E}_{\tau\sim\pi_\theta}[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(A_t|S_t)G(\tau)] && (2.43)
\end{aligned}$$

Now we can approximate the policy gradient term using sample episodes. As we already discussed, this is an on-policy method, which means the trajectories will come from the same policy as the one we are trying to improve. To calculate the gradient, we can compute the gradient for every trajectory using the above term (Eq.(2.43)), find the average and then use it to perform an update for the gradient ascent algorithm. This algorithm is called REINFORCE (Williams, 1992).

2.7.2 Policy Gradient with Baseline

In Equation (2.43), the term $G(\tau)$ is used. However, it can also be implemented by comparing the action value with an arbitrary baseline $b(s)$ that is independent of action, as follows:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(A_t|S_t)(G(\tau) - b(S_t))] \tag{2.44}$$

The baseline b can be chosen to reduce variance. The only constraint is that it should not depend on actions. In many scenarios, the baseline is simply the moving average of the rewards observed up to the current time point. This encourages the agent to take actions that are at least as good as the average value of the actions taken thus far.

2.7.3 Actor-Critic Methods

A very useful baseline is the on-policy state value function $V^{\pi}(S_t)$, which needs to be approximated using a function approximator, such as a neural network. This value function network gets updated along with the policy network, so that it uses samples of the most recent policy. The goal is improve on REINFORCE with baseline. In this case, the policy is updated essentially based on the advantage function A^{π}, which is also called the critic. This approach is called actor-critic (Sutton and Barto, 2018), and many versions exist, depending on how the actor and the critic are implemented and updated. For example, TD methods could be used to update the critic.

2.8 Model-based Reinforcement Learning

A good way to distinguish among different RL algorithms is to see if the agent needs to have access to an estimated transition probability model, sometimes called model of the world, in order to make decisions. This model is defined as the probability distribution of the next state, given the current state and action, as per Equations (2.4) and (2.5). It can either be given to the agent or learned.

The main component differentiating model-based RL from model-free RL is the fact that model-based RL algorithms perform planning using the model. Despite their differences, both approaches rely heavily on calculating value functions. Both methods first predict future events and then approximate the value function using this prediction.

The goal of model-based RL is to learn the transition probability function

$P(S_{t+1}|S_t, A_t)$. Then, we can explicitly use the model to unroll various sequences of states and actions and pick the action which leads to the highest return.

For the deterministic case, there is always one next state, $S_{t+1} = f(S_t, A_t)$, and the model can be trained to minimize mean squared error loss as follows:

$$\sum_i ||f(s_i, a_i) - s_{i+1}||^2 \tag{2.45}$$

where s_{i+1} is the observed next state. But often this approach does not work well because a deterministic model is not expressive enough. To get around this problem, we can re-plan at every step and add the action taken from the current policy to the experience buffer, which helps learn the model. Both re-planning and executing the actions are done using Model Predictive Control (MPC) which will be explained in the next section.

The most common approaches to model the environment are:

1. **Gaussian Processes (GPs):** Here the input is (S_t, A_t) and the output is S_{t+1}. GPs are efficient in a low-data regime. However, they can be extremely slow when the dataset (the buffer in which we store the transitions) is large. GPs also do not do a great job when the dynamics are not smooth.

2. **Neural Networks (NNs):** Like GPs, NNs also learn to map the inputs (S_t, A_t) to S_{t+1}. An advantage of using NNs is that we can use a lot of data and they can be very expressive. However, if we have very few data points, then NNs perform poorly.

3. **Gaussian Mixture Models (GMMs):** These approximate $P(S_{t+1}|S_t, A_t)$ using the tuples (S_t, A_t, S_{t+1}) and maximizing data log-likelihood, usually through Expectation Maximization (EM).

2.8.1 Model Predictive Control

Model Predictive Control (MPC) (Camacho and Alba, 2013), is a simple method for control problems, best suited when the state space and the action space are both large. MPC works very well when joined with a learned model of the world. MPC works by planning a sequence of actions for the current state for a finite horizon, using the current model. However, only the first action of the best trajectory is then taken, the data obtained is used to update the model, and then a new plan (a sequence of actions) is recomputed and the cycle repeats. Hence, even if the model is inaccurate, since only the first action of the plan is taken, the impact of model errors on the action choice is not catastrophic. Using a short time horizon for the planning also means that the computation will be more feasible.

2.8.2 Pitfalls in Model-based RL

One important point to always keep in mind is that the planner could look for regions where the model is wrongly optimistic. Some planners, in order to generate good solutions, require very accurate models. On the other hand, some tasks can be solved by a very simple policy even if the environment dynamics are very complex. In such cases, policy gradient and other model-free methods can be more effective than model-based RL.

3 Learning the model of the world

While model-based RL has long been viewed as a very promising approach in the RL field, it has always been very challenging empirically when compared to model-free methods. However, the recent success of AlphaZero (Silver et al., 2017) and MuZero (Schrittwieser et al., 2019) has renewed the interest in model-based RL, by demonstrating the power of planning based methods, provided we have access to an accurate model of the world. In this chapter, we will survey some of the recent successful RL methods that aim to learn the model of the world in order to solve the given task efficiently.

3.1 World Models

Ha and Schmidhuber (2018) proposed an agent design for solving control tasks from pixel-level observations, called World Models. Specifically, they learn the following three components separately to solve a task:

1. A vision module, which maps the pixel observations to a latent vector space.
2. A memory module, which integrates the information in the latent vector space learnt by the vision module and can be used to make future predictions.
3. A controller, which uses the output of the vision module and the memory module to predict what action to take next.

The vision module of the world model is a variational auto-encoder (VAE) which takes a 2D image at any time-step t as input and learns to compress it into a low-dimensional latent vector z_t, such that the image can be reconstructed from the latent vector.

While the vision module computes a low-dimensional representation of the information from the current time step, the memory module integrates this information over time. This is done by using a recurrent neural network (RNN) which takes a sequence of z vectors as inputs and predicts the next z vector that the VAE will produce after seeing the next image. Since the environment could be stochastic, the RNN outputs a probability distribution over z, $p(z)$. Specifically, the RNN will model the following conditional distribution: $P(z_{t+1}, r_{t+1}, a_{t+1}, d_{t+1} | a_t, z_t, h_t)$ where a_t is the action taken at time t, d_{t+1} is an indicator function of whether the episode ends at time step $t + 1$, and h_t is the hidden state of the RNN at time t. This probability distribution is modeled as a mixture of Gaussians, with a temperature parameter which can be tuned to increase the stochasticity of the prediction. This memory model is depicted in Figure 3.1.

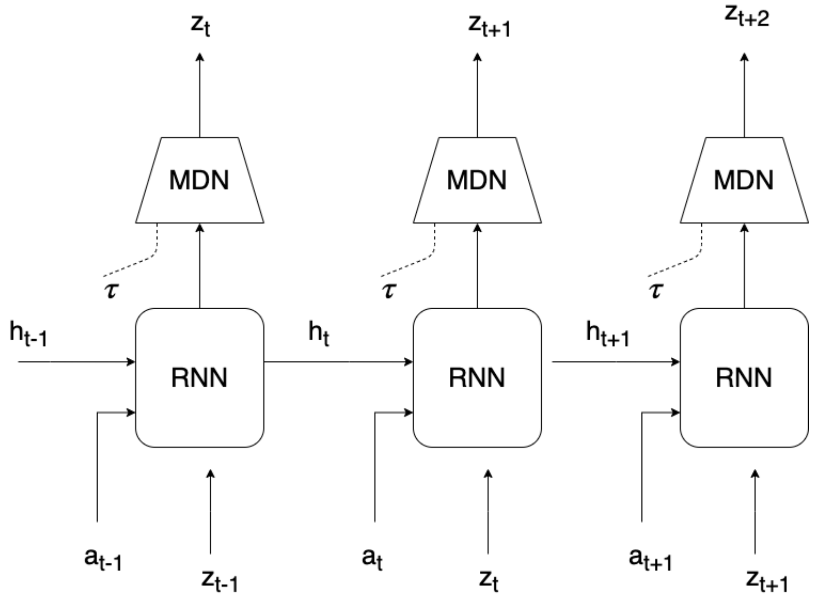


Figure 3.1: The memory module of the World Model architecture.

The controller is a linear model which takes h_t and z_t as inputs to predict the action a_t. The authors argue that by transferring most of the capacity to the representation learner and the model, we can afford to train a simple linear controller in more unconventional ways, such as evolutionary methods. Specifically, they use Covariance-Matrix Adaptation Evolution Strategy (CMA-ES) (Hansen, 2016).

In the World Model architecture, all three components are trained in a sequential way. First, a set of rollout trajectories are collected by using a randomly initialized controller (which could be updated based on the interactions during this data collection process). Next, the images in the rollouts are used to train the VAE model. Then, the rollout trajectories are encoded in z space to train the RNN model, using the encoded trajectories. The main novelty is in using this trained model to create an environment for training the controller "in the dream". Specifically, the authors wrap the model in an RL environment (using the standard Gym interface) and use it to train the controller. The temperature parameter of the MDN-RNN model is increased, to increase the stochasticity of the dream world, which helps the agent to avoid exploiting the imperfections of the learnt model, and which leads to robustness while performing in a real environment. The experimental results show that an agent trained purely in the dream can achieve state-of-the-art performance in the real environment. The entire world model architecture is summarized in Figure 3.2.

3.2 Gradient-based Planning

Henaff et al. (2017) introduced a model-based RL algorithm which uses gradient descent based planning for taking actions. The authors collect a set of trajectories by executing a random policy and use them to learn the reward function $f_R(S_t, A_t)$ and the transition function $f_S(S_t, A_t)$, which predicts the next state S_{t+1}, These functions together are known as the forward model, and are represented by a neural

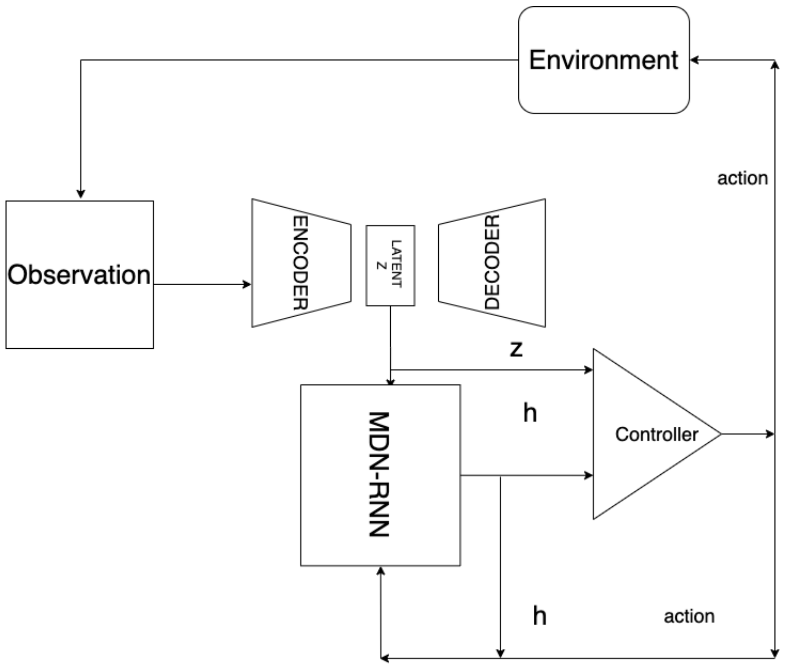


Figure 3.2: The World Model architecture

network. The loss function per time step is defined as follows:

$$l(S_t, A_t, R_t, S_{t+1}) = L_S(S_{t+1}, f_S(S_t, A_t)) + L_R(R_t, f_R(S_t, A_t)) \tag{3.1}$$

Both L_S and L_R are mean-squared error loss in their experiments, but in principle, they could be different. The functions f_S and f_R can be completely independent or they could be a single neural network which takes S_t and A_t as inputs and predicts both S_{t+1} and R_t with multiple heads. The next state predicted by the model can be fed back as the current state during the next prediction, instead of teacher-forcing the current state, which helps the model to learn how to be robust to its own imperfections.

Having learned these two functions to predict the next state and reward, one can use gradient descent to find a sequence of actions that maximizes the expected return of a specific trajectory, provided the actions are also continuous. This is done by solving the following optimization problem:

$$\arg\max_{A_0,\ldots,A_T} \sum_{t=0}^{T-1} f_R(f_S(\tilde{S}_t, A_t), A_{t+1}) \tag{3.2}$$

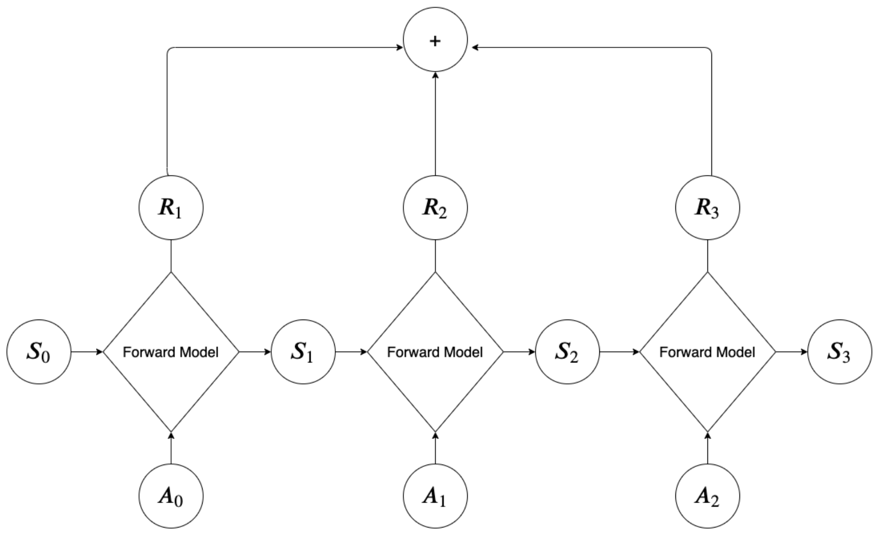


Figure 3.3: Gradient based planning model. Only the actions are updated using gradient descent while the forward model is fixed.

where $\tilde{S}_0 = s_0$ and $\tilde{S}_{t+1} = f_S(\tilde{S}_t, A_t)$. The entire architecture for gradient based planning is depicted in Figure 3.3.

While this method works for continuous action spaces, it cannot handle discrete actions spaces. If we initialize the action sequence with one-hot vectors, there is no guarantee that gradient descent will find the right one-hot action vector. Hence, Henaff et al. (2017) propose to re-parameterize the discrete action space to be a continuous action space, by restricting the actions to a simplex. If $e_1, e_2, ...e_d$ are the set of one-hot action vectors, then the optimization problem after the re-parameterization is as follows:

$$\arg \max_{A_0,\ldots,A_T} \sum_{t=0}^{T-1} f_R(f_S(\tilde{S}_t, \sigma(X_t)), \sigma(X_t)) \tag{3.3}$$

where σ is the softmax function. A sequence of actions can then be chosen by solving this optimization problem to quantize each $\sigma(X_t)$ to the closest one-hot vector e.

Even though this reformulation forces the action space to be in the simplex, optimizing using gradient descent may leave the actions in the interior of the simplex rather than at the vertices, which are the one-hot vectors. The next modification is to add Gaussian noise ϵ to the action vectors at the time of training the forward model, which will make the loss surface more convex around the action vectors. This results in the following modified loss function:

$$\tilde{l}(S_t, A_t, R_t, S_{t+1}) = \mathbb{E}_{\epsilon \sim \mathcal{N}(0,\sigma^2)}[l(S_t, A_t + \epsilon, R_t, S_{t+1})] \tag{3.4}$$

which can be re-written as:

$$\int \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{\epsilon^2}{2\sigma^2}} l(S_t, A_t + \epsilon, R_t, S_{t+1}) d\epsilon \tag{3.5}$$

The probability mass is highest at $\epsilon = 0$, which corresponds to the one-hot action vector A_t. This increases the number of points sampled closer to the one-hot vector at the time of training, causing it to lower the loss surface around those points at the time of the gradient descent update. The result is a smoother loss surface around the one-hot vector encoding each of the actions, making it easier to optimize.

Doing gradient descent based planning to take every action is time consuming. Hence, the authors proposed learning a policy network, by generating numerous trajectories using the learnt forward models and training the policy network in a supervised way. The authors demonstrated that this policy network with knowledge distilled from the gradient based planner achieves better results than state-of-the-art RL methods like Trust Region Policy Optimization (TRPO) (Schulman et al., 2015).

3.3 Universal Planning Network

Universal Planning Network (UPN) (Srinivas et al., 2018) is an end-to-end model-based learning algorithm for MPC. Like World Models (Ha and Schmidhuber, 2018), UPN learns an encoder representation for the states and a forward transition model in the latent space. Like Henaff et al. (2017), UPN has a gradient-based planning module. All these components are put together and everything is trained with a single high-level imitation learning objective.

Specifically, UPN is a goal-conditioned policy architecture which receives the current observation and a goal observation as inputs. These two observations are encoded by a neural network f_ϕ into a latent vector representation. UPN also learns a forward prediction model g_θ which takes a latent state x_t and action a_t as inputs and predicts the next latent state x_{t+1}. Now, given x_t, x_g, and a randomly initialized action sequence of length T (the horizon), the planner module first unrolls the future states from x_t to x_{t+T+1} by using g_θ. The planning loss is computed as the difference between x_{t+T+1} and x_g. The Huber loss is computed between the two entities. The action sequence is updated based on the gradient with respect to the planning loss. Now, this gradient based planning is repeater for k iterations, with the updated action sequence from the previous iteration. Note that only the action sequence $a_t, ..., a_{t+T}$ is updated during gradient based planning. The parameters of f_ϕ and g_θ are not updated.

UPN also has an outer loss, which takes the action sequence returned by the gradient based planner and computes a behaviour cloning loss with respect to the action sequence from the demonstration data. This loss is back-propagated through the entire gradient descent planning module and is used to update f_ϕ and g_θ. The goal of UPN is to learn a representation and a forward model which is directly useful for planning. Once trained, UPN can be used to plan a sequence of actions and after taking the first step in the sequence, one can re-plan using UPN. The authors have shown that MPC using UPN achieves state-of-the-art results when compared to other model-free methods and simple imitation learning methods. The gradient descent planner module of UPN is depicted in Figure 3.4.

3.4 PETS

Probabilistic Ensembles with Trajectory Sampling (PETS) (Chua et al., 2018) is the first known method to accomplish the same asymptotic performance as model-free

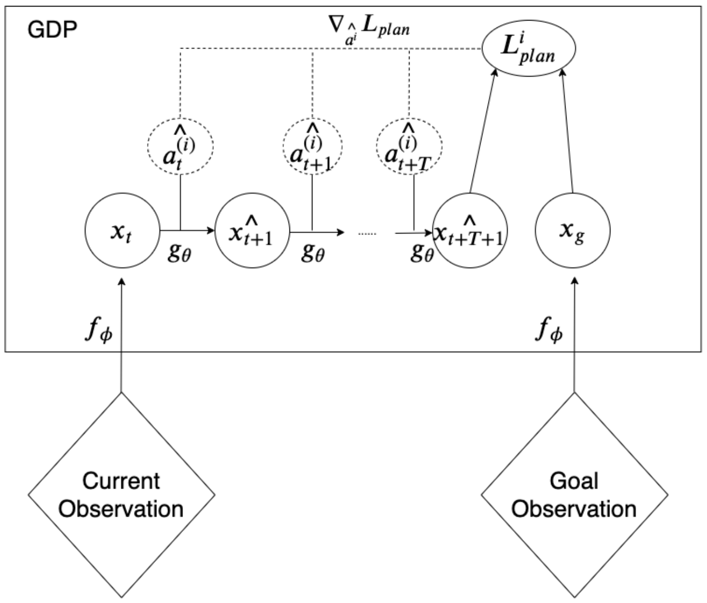


Figure 3.4: Gradient descent planner module in Universal Planning Networks.

algorithms while simultaneously being orders of magnitude more sample efficient. Model capacity and expressivity are the most important aspects of choosing a model to learn the dynamics. While Gaussian Processes (GPs) are capable of learning very fast, they are a good choice of model to represent the dynamics only if the amount of available data is small. However, as mentioned by Calandra et al. (2016), in the case of very complex and discontinuous environments, GPs should not be chosen, since they are not scalable.

To incorporate neural networks into model-based RL, one needs to handle the different types of uncertainty that exist in modelling. According to Chua et al. (2018), there are two main types of uncertainty. One is called *aleatoric uncertainty*, and refers to inherent stochasticites and non-stationarity of the system. To take care of aleatoric uncertainty, one can deploy probabilistic neural networks that predict a probability distribution from which to sample, instead of a mean point prediction. Another type of uncertainty is the *epistemic uncertainty*, which is caused by our function approximator and its limits in capturing the entirety of the dynamics, typically caused by lack of data. The usual assumption is that if we have infinite data and a model with high capacity to represent the data, epistemic uncertainty should disappear. An ensemble of probabilistic neural networks can be used to tackle epistemic uncertainty. The goal is to capture both types of uncertainty, which is done well with PETS.

A probabilistic neural network outputs a probability distribution over the next state S_{t+1} given the current state S_t and the current action A_t. It is trained by maximizing the log-likelihood of the sampled next state, which is equivalent to minimizing the following loss function:

$$\sum_{t=0}^{T-1} - \log f_\theta(S_{t+1}|S_t, A_t) \tag{3.6}$$

where T is the length of the trajectory and f_θ is a neural network with parameter θ. One caveat when using probabilistic neural networks is that their estimate of variance may have arbitrary values for out-of-distribution inputs. These variance estimates may collapse to zero or even explode towards infinity, whereas this issue does not exist in GPs, because the variance is well behaved, bounded and Lipschitz smooth. Chua et al. (2018) bounded the output using a maximum and minimum value to avoid this issue.

While one can use Bayesian Neural Networks (BNNs) to handle epistemic uncertainty, they require heavy computation. Ensembles of models trained with bootstrapped datasets are simpler than BNNs. B-many boostrap models are used in PETS, with θ_b denoting the parameters of the b^{th} model. To train the b^{th} model, a dataset D_b is generated by sampling with replacement from the original dataset D.

The actions can be generated from a policy $\pi(a|s)$ or from MPC (Camacho and Alba, 2013). The advantage of MPC is that it is simple to use and we do not have to define the entire task horizon in advance. The only two requirements for MPC are knowing the current state S_t and a prediction horizon T.

The most straightforward way of generating action sequences is using the random shooting method, which was originally proposed by Nagabandi et al. (2018) using

Gaussian Processes. Through this method we randomly generate action trajectories that are sampled from a Gaussian distribution, apply the optimization and then select the best trajectory. Instead, Chua et al. (2018) use the Cross Entropy Method (CEM) method (Botev et al., 2013), in which action trajectories are sampled from the distribution that is closer to the action trajectories that yielded the highest return last time.

Since we are learning uncertainty aware models, we should take care of approximating uncertainty propagation. The three most common uncertainty propagation methods are:

1. **Deterministic:** in which the mean prediction will be used and the variance (uncertainty) ignored.

2. **Particle:** in which a set of Monte Carlo samples is propagated.

3. **Parametric:** in which Gaussian or Gaussian mixture models are used.

As shown by Kupcsik et al. (2013), particle methods are highly accurate and computationally effective, while simultaneously making no strong assumptions on the prior distributions. Thus, Chua et al. (2018) use particle based propagation in PETS, specifically, Trajectory Sampling (TS) state propagation. In trajectory sampling, given a state S_0 and P particles, each particle is propagated using:

$$S_{t+1}^{p} = \hat{f}_{\theta_{b(p,t)}}(S_t^p, A_t) \tag{3.7}$$

where $b(p,t)$ is one of the bootstrap models that may include time index as an impacting parameter.

Two types of TS propagation are used:

1. ***TS1:*** In TS1, particles re-sample a bootstrap model at every time step. If we used a Bayesian model instead of an ensemble of models, then particles would be consistently re-sampled from the approximate marginal distribution

of dynamics. This type of state propagation imposes a soft limit on trajectory multi-modality.

2. ***TS*∞**: In $TS\infty$ the same particle bootstrap is used throughout the entire trial. $TS\infty$ is able to learn the time invariance due to the fact that each bootstrap index is made consistent over time. In $TS\infty$, aleatoric and epistemic uncertainties are distinct (Depeweg et al., 2017). In this type of propagation technique, aleatoric state variance is the average variance of particles of the same bootstrap. However, epistemic state variance is the variance of the average of the particles of the same bootstrap indices.

The PETS training procedure is summarized in Algorithm-1.

Algorithm 1 PETS

```
 1: Initialize dynamics network parameters $\phi$, data-set $\mathcal{D}$
 2: while Training iterations not Finished do
 3:     for $i^{th}$ time-step of the agent do                          ▷ Sampling Data
 4:         Initialize action-sequence distribution. $\mu = \mu_0$, $\Sigma = \sigma_0^2 \mathbf{I}$
 5:         for $j^{th}$ CEM Update do                                  ▷ CEM Planning
 6:             Sample action sequences $\{\hat{a}_i\}$ from $\mathcal{N}(\mu, \Sigma)$.
 7:             for Every candidate $\delta_i$ do                       ▷ Trajectory Predicting
 8:                 for $t = i$ to $i + \tau$, $S_{t+1} = f_\phi(S_{t+1}|S_t, A_t = \hat{A}_t)$▷ $f$ is a probabilistic
    ensemble.
 9:                 Evaluate expected reward of this candidate.
10:             end for
11:             Fit distribution of the elite candidates as $\mu', \Sigma'$.
12:             Update noise distribution $\mu = (1 - \alpha)\mu + \alpha\mu'$, $\Sigma = (1 - \alpha)\Sigma + \alpha\Sigma'$
13:         end for
14:         Execute the first action from the optimal candidate action sequence.
15:     end for
16:     Update $\phi$ using data-set $\mathcal{D}$                        ▷ Dynamics Update
17: end while
```

4

Empirical Analysis of PETS

PETS (Chua et al., 2018) was the first model-based deep RL algorithm to match the asymptotic performance of model-free RL algorithms while requiring fewer samples. In this chapter, we perform a systematic empirical analysis of the PETS algorithm in order to study the effect of various design choices in the algorithm. Our motivation comes from the work of Henderson et al. (2018) who showed that it is very hard to make a meaningful comparison of deep RL algorithms due to non-determinism of the benchmarks, intrinsic variance in the algorithms, and poor standardization of the experimental reporting.

4.1 Tasks and Experimental Setting

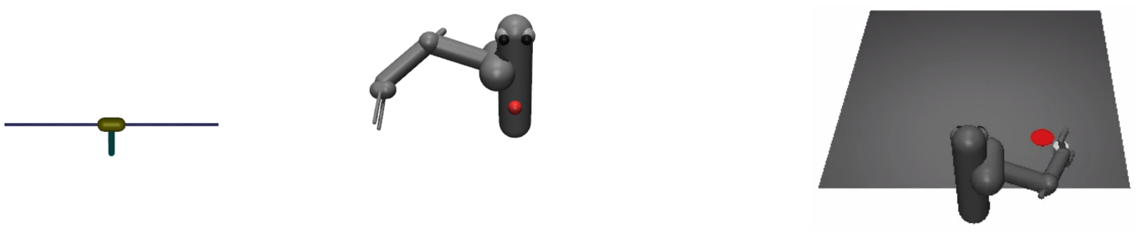
Chua et al. (2018) used the four continuous control benchmark tasks built on top of the Mujoco physics simulator (Todorov et al., 2012) in their experiments: cartpole, reacher, pusher, and half-cheetah. For our analysis, we used the first three tasks which are explained below:

1. Cartpole is a classic benchmark where the goal is to balance a pole as long as possible. The pole is attached to a cart, and the agent has to push the cart in order to avoid the pole falling below a certain angle. The state space has 4 features and there are 2 possible actions.

2. Reacher is a simulated 7-DOF PR2 robot. The goal for the robot is to reach a target. The state space has 14 features and there are 7 possible actions.

3. Pusher uses the same simulated 7-DOF PR2 robot. The goal for the robot in this task is to push an object to a target location. The state space has 14 features and there are 7 possible actions.

The three tasks are depicted in Figure 4.1. While these images are used for visualization purposes, the agent takes only the low-dimensional, raw state features as input. The code to reproduce all the experiments reported in this chapter is available at https://github.com/amini2nt/pets.



((a)) Cartpole ((b)) Reacher ((c)) Pusher

Figure 4.1: Cartpole, reacher, and pusher tasks.

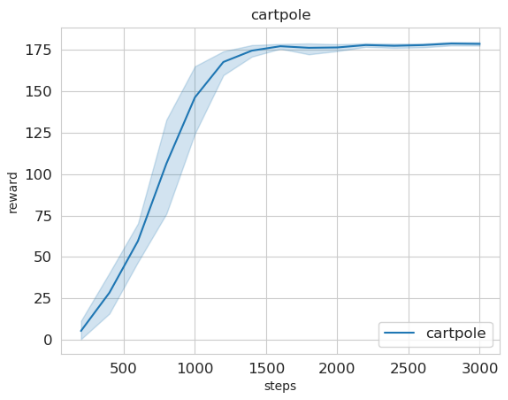
4.2 Baseline Performance

Chua et al. (2018) ran 100 experiments for each task and reported the average performance for the top 10 best performing experiments. This reporting has the following issues:

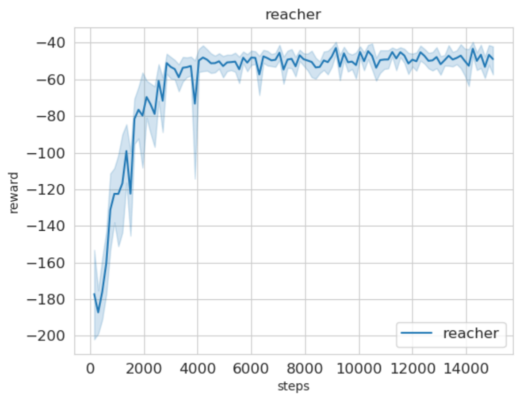
1. Choosing the top 10 experiments skews the mean performance towards the best performance and hence is not the true average performance of PETS.

2. There is no standard deviation information in the reports, meaning no available data on the variance of PETS performance.

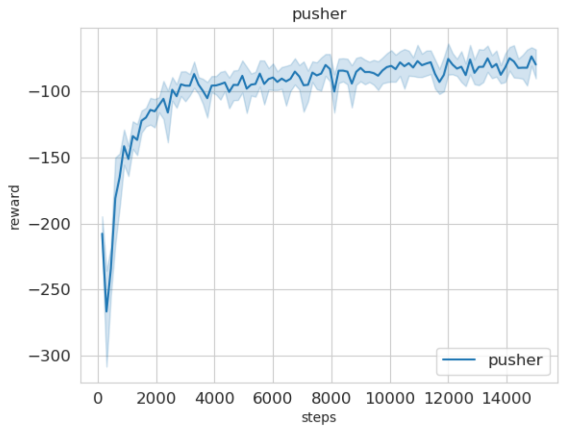
Deep RL experiments are costly and hence running a lot of experiments to obtain statistically significant average performance of an algorithm is a computationally-heavy and time-consuming process. However, discarding 90 completed experiments and only reporting the average performance of the top 10 experiments is not appropriate. In order to get meaningful performance results, we ran PETS with the same configuration for 10 random seeds and reported the average performance, along with the confidence interval of one standard deviation in Figure 4.2.



((a)) Cartpole



((b)) Reacher



((c)) Pusher

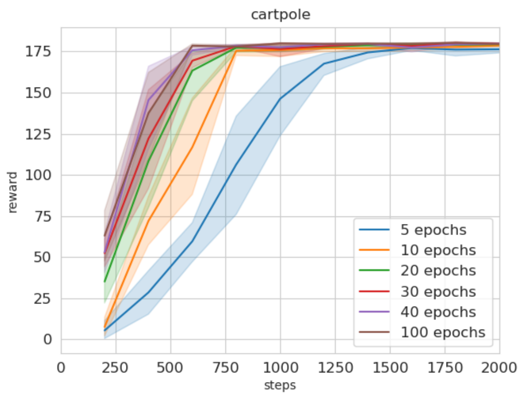
Figure 4.2: Performance of PETS in cartpole, reacher, and pusher tasks. Results are averaged over 10 random seeds.

Comparing these performance curves with the ones reported by Chua et al. (2018), we observe that while the performance in Cartpole is similar, the reported performance in Pusher and Reacher in their paper is higher than the average performance we report in Figure 4.2. This is due to the fact that Cartpole is a relatively simple environment and hence it is easy to learn an accurate model with less variance in performance. On the other hand, Pusher and Reacher are complex environments where modeling errors result in variance in the performance.

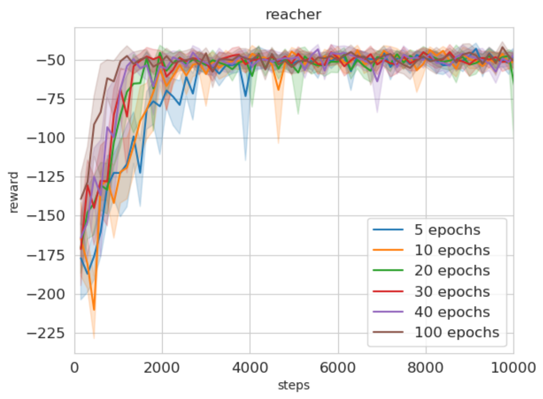
4.3 Effect of training models longer

In this section, we study how training models longer affects performance. In every iteration of PETS training, we used the current model with MPC to collect training episodes and then applied the collected data set of transitions to train the model of the environment. This trained model is then used in the next iteration for more data collection. Chua et al. (2018) train their models for 5 epochs in every iteration. While this will help us obtain a better model over several iterations, there is an advantage to improving the accuracy of the model earlier. If we can obtain a better model in one iteration, that would help in collecting high performing trajectories in the next iteration to better train the model. Overall, this process should increase the sample efficiency of the algorithm.

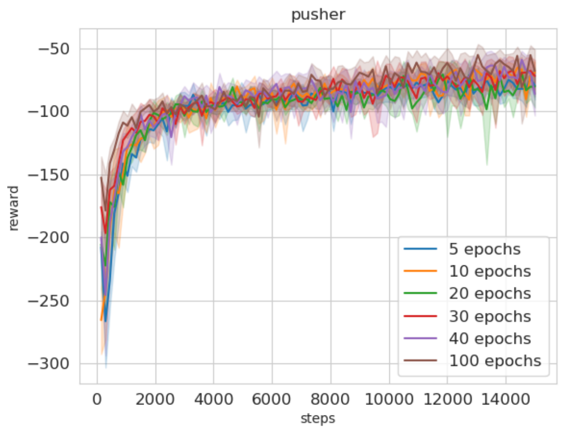
This is verified in our experiments reported in Figure 4.3. As we can see, increasing the number of epochs in model training resulted in faster convergence in all three tasks. The maximum improvement is in Cartpole, where it is easy to learn an accurate model of the environment with very little data. The improvements saturate after 40 epochs. However, in the case of Reacher, which is more complex, there is clear improvement in training for 100 epochs when compared to training for 40 epochs. Pusher is the most complex task, thus the improvements are only minor, which suggests that more training is needed. Pusher results also highlight that the reported results were limited by the inaccuracy of the model since we can see performance improvement when the models are more accurate.



((a)) Cartpole



((b)) Reacher



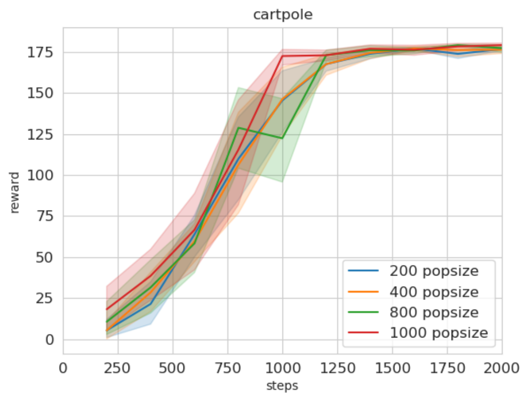
((c)) Pusher

Figure 4.3: Effect of training the model longer. Chua et al. (2018) reported results with models trained for 5 epochs.

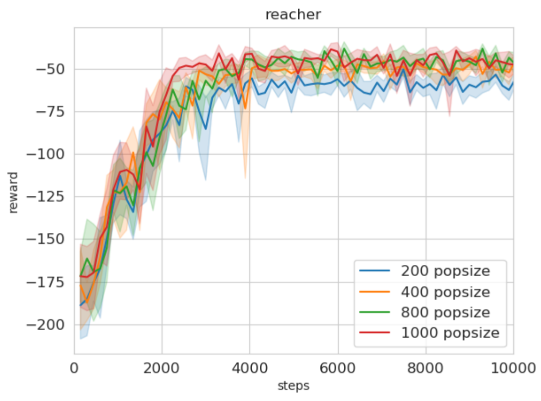
We would like to highlight that data collection is the bottleneck in PETS training, not the model training time. This is because data collection requires population based search (as implemented in CEM) and is often done in CPU. On the other hand, model training is simple supervised learning with batch gradient descent on GPUs and does not require interaction with the environment.

4.4 Effect of population size

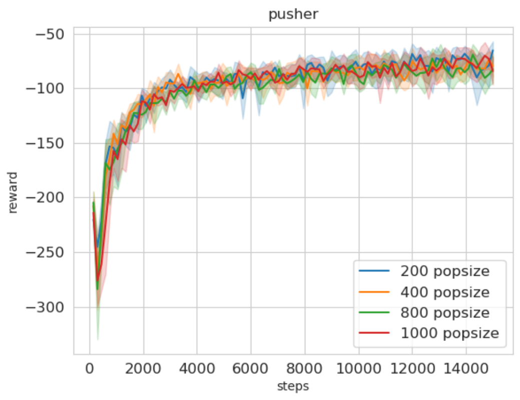
In this section, we study the effect of the population size of the CEM algorithm on the final performance. CEM uses particle based trajectory sampling in which



((a)) Cartpole



((b)) Reacher



((c)) Pusher

Figure 4.4: Effect of increasing the population size. Chua et al. (2018) used a population size of 400.

p different trajectories are sampled from the current state, where p is the number of particles. Then CEM chooses the top-k particles (also known as elite particles) based on the returns from the trajectories and uses these elite particles to update the action distribution from which future action sequences are sampled.

The population size of CEM refers to the number of particles used in the trajectory sampling. Intuitively, the higher the population size, the better the expected performance. In Figure 4.4, we compare the performance of PETS for various population sizes. While Chua et al. (2018) used a population size of 400, we analyzed the performance by varying the population size from 200 to 1000. As we can see from the Cartpole and Reacher experiments, increasing the population size can help by providing slightly faster convergence and better performance. However, the return is negligible after a population size of 800. Pusher is a complex task, and thus it requires an even greater population size to improve the performance.

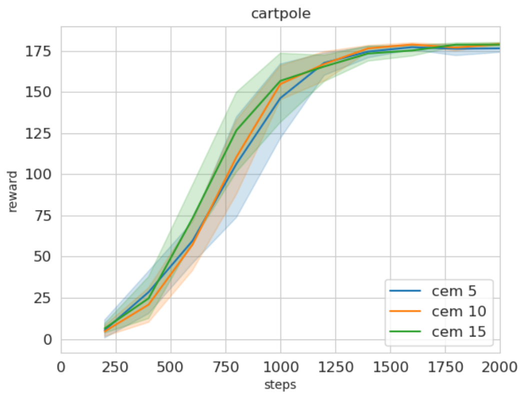
4.5 Effect of CEM iterations

CEM particles are action sequences sampled from a Gaussian distribution, where the mean and standard deviation are updated iteratively based on the population statistics from the elite particles. As we increase the number of CEM iterations, the mean of this action distribution should approach the optimal action and the variance of the action distribution should reduce significantly. Hence, longer CEM iterations should help us to improve the performance.

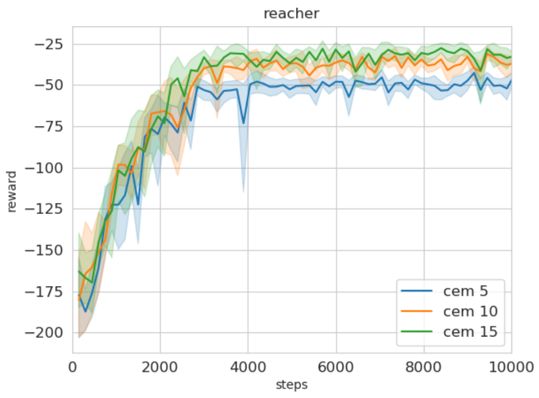
Figure 4.5 reports the performance of PETS when varying the number of CEM iterations. Chua et al. (2018) used 5 CEM iterations. As we can see, increasing the number of CEM iterations improves the performance in all three tasks. In the simple Cartpole task, we do not see a big difference since PETS can learn a highly accurate model even with fewer samples. However, the performance improvement is clear in Reacher and even Pusher shows some improvement.

4.6 Complementary effects

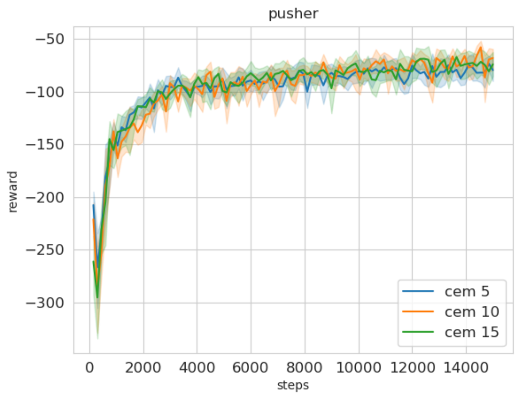
In this section, we try to understand if the benefits gained in the last three experiments are complementary to each other. First, we study the complementary effects of training the model and increasing the CEM iterations. The results are reported in Figure 4.6. In Cartpole, both longer model training and increased CEM iterations helped to achieve better performance. We can see that models that are trained longer have faster convergence (1750 steps vs. 600 steps). The configuration with longer model training and more CEM iterations achieves the best performance.



((a)) Cartpole



((b)) Reacher

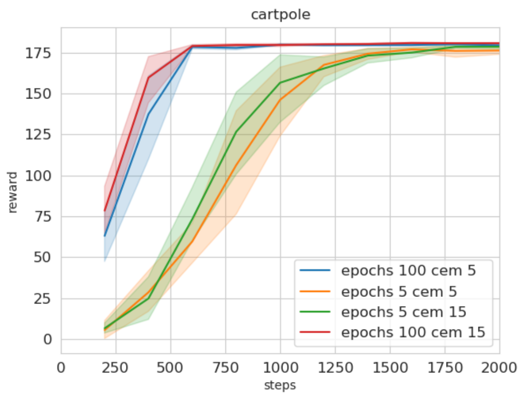


((c)) Pusher

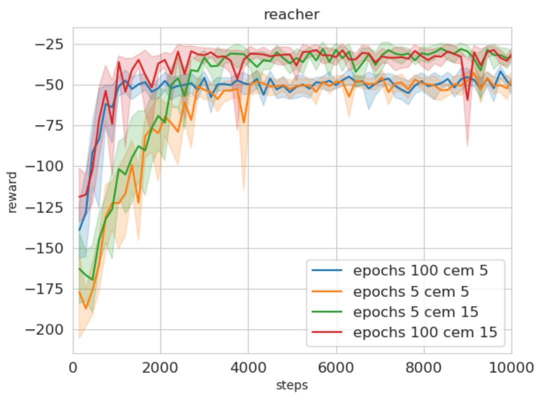
Figure 4.5: Effect of increasing the CEM iterations.

The importance of more CEM iterations is clear from the Reacher experiments. Even though the model trained for 100 epochs converged three times faster than the model trained with 5 epochs (1000 steps vs 3000 steps), it achieved the same final performance. However, the model trained for just 5 epochs, when more CEM iterations were added, converged to a significantly better solution. Increasing the amount of model training helps to obtain faster convergence to a superior solution.

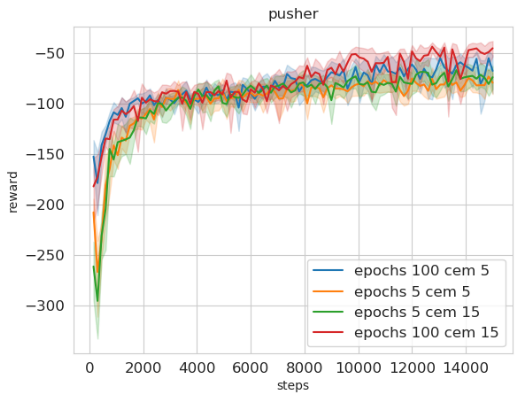
Results in Pusher show that as the task gets more complex, both better model and longer CEM iterations are needed to improve the performance. This is evident from the faster convergence and significantly superior performance of the model trained for 100 epochs with 15 CEM iterations.



((a)) Cartpole



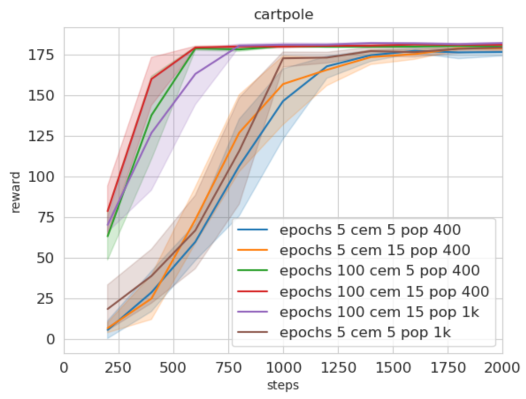
((b)) Reacher



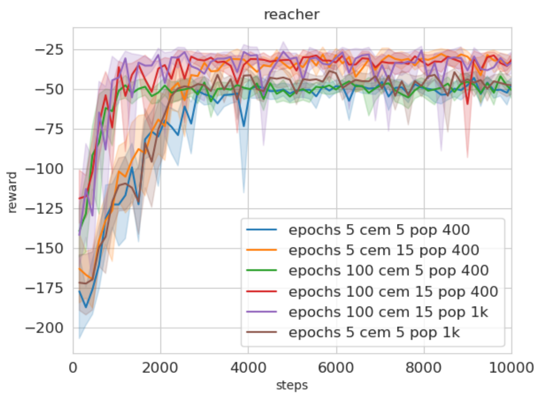
((c)) Pusher

Figure 4.6: Complementary effects of training the model longer and increased CEM iterations.

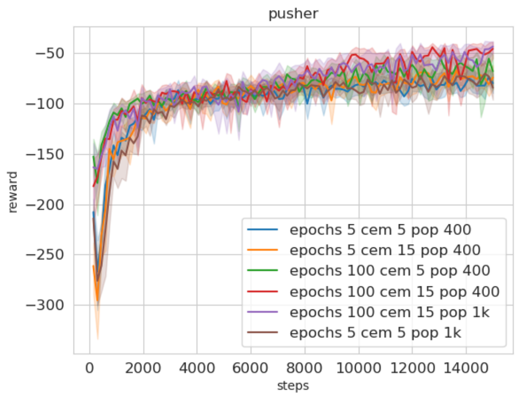
Lastly, we study how the population size of CEM affects the performance when combined with the other two improvements. The results are reported in Figure 4.7. From the figure, it is clear that increasing the population size helps when we have a less perfect model. However, as we increase the model training (from 5 epochs to 100 epochs), we can see that increasing the population size either hurts (as in the Cartpole experiments) or has no significant effect (as in the Reacher and Pusher experiments).



((a)) Cartpole



((b)) Reacher



((c)) Pusher

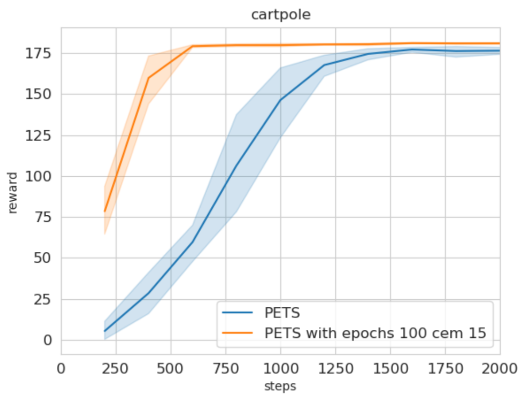
Figure 4.7: Complementary effects of population size.

4.7 Final performance

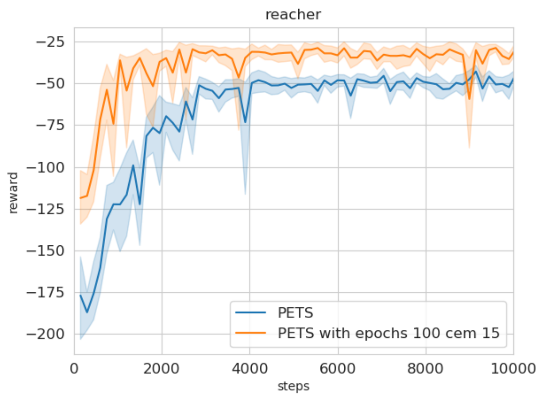
Based on our analysis, we have the following recommendations for improving the performance of PETS:

1. Train the model longer in every iteration. This does not require extra data, since we are training the model in batch mode by using the collected data. This helps data efficiency by improving the MPC. We would like to highlight that training the model longer does not have huge computational overhead since the training time is dominated by the CEM steps.

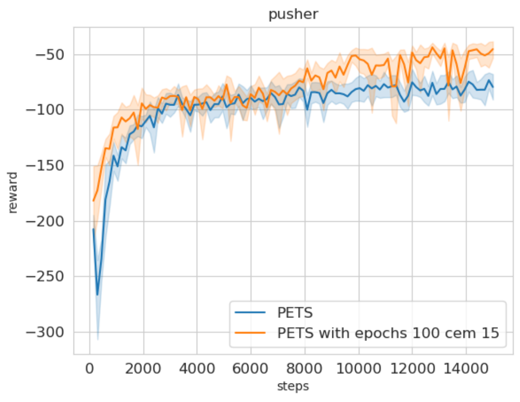
2. Increase the CEM iterations. While this makes the training slower (in terms



((a)) Cartpole



((b)) Reacher



((c)) Pusher

Figure 4.8: Performance of PETS with and without the suggested improvements.

of CPU time), this is necessary in more complex tasks to achieve better performance.

Figure 4.8 summarizes the performance of baseline PETS model and the improved PETS model. We achieve faster convergence and significantly better performance in all the three tasks.

4.8 Discussion

Recently Wang and Ba (2020) proposed a new algorithm called POPLIN which extended PETS by adding a policy network. The authors demonstrated that POPLIN can achieve superior performance to PETS. Our improvements to PETS close the gap between these two algorithms without adding new components to the algorithm. For example, using the suggested setting for PETS, we achieved approximately the same performance as POPLIN with fewer time steps (10k vs 50k) in a complex simulated robot reaching task (Reacher). In Pusher, we achieved better performance than POPLIN with fewer steps (15k vs 50k). We would like to highlight that POPLIN is a compute-intensive approach when compared to PETS since it does CEM search in the high-dimensional policy parameter space instead of the low-dimensional action space and hence cannot be efficiently mini-batched like PETS. This limitation of POPLIN is further discussed in the next chapter where we propose an efficient solution to do CEM in policy parameter space. Our PETS results highlight the importance of proper analysis of new algorithms.

5

Policy-guided Latent Space Planning

In this chapter, we propose a new model-based reinforcement learning algorithm called PG+ which is built on top of PETS (Chua et al., 2018) and PlaNet (Hafner et al., 2018). The results in this chapter are also presented in (Amini et al., 2020).

5.1 Background

Deep Planning Network (PlaNet) (Hafner et al., 2018) is a model-based RL algorithm which learns a latent dynamics model that can predict both the transition dynamics and the reward function in latent space.

Unlike PETS, which uses an ensemble of feed-forward networks with probabilistic outputs, PlaNet uses a single variational recurrent neural network (RNN) with probabilistic output. Specifically, given the previous latent state s_{t-1} and action a_{t-1}, PlaNet first computes a deterministic state model as follows:

$$h_t = f(h_{t-1}, s_{t-1}, a_{t-1}) \tag{5.1}$$

where f is a recurrent neural network. Given this deterministic hidden state, then a stochastic state model is constructed: $s_t \sim p(s_t|h_t)$. PlaNet learns an observation model and reward model conditioned on the state models: $p(o_t|h_t, s_t)$ and $p(r_t|h_t, s_t)$. Together, these models form the Recurrent State Space Model (RSSM).

The complete architecture of an RSSM model unrolled over three time steps is shown in Figure 5.1. Unlike PETS, which takes a low-dimensional state as input, PlaNet takes a high-dimensional observation space (in the form of images) as input. However, predicting forward in high-dimensional observation space is not desirable due to the following reasons:

1. We need to evaluate thousands of action sequences at every time step of the agent. Doing this with a forward model that predicts high-dimensional observations is a computationally intensive process to perform during decision time.

2. Image observations often contain a lot of irrelevant information. Hence, trying to predict the entire observation can be error-prone, since these errors then get propagated as we predict far in the future. It is also not necessary to capture the irrelevant details (like image background) in the transition dynamics.

To avoid these issues, RSSM learns to predict forward purely in the low-dimensional latent space s_t. Hafner et al. (2018) emphasize the importance of conditioning both on the deterministic path h_t and on the stochastic path s_t for better modeling of the transition dynamics.

RSSM is trained by maximizing the variational lower bound on the observation log-likelihood, defined as follows:

$$\log\ p(o_{1:T}|a_{1:T}) = \log \textstyle\int \prod_t\ p(s_t|s_{t-1}, a_{t-1})\ p(o_t|s_t)\, ds_{1:T}$$
$$\geq \textstyle\sum_{t=1}^{T} \left(\mathbb{E}_{q(s_t|o_{\leq t}, a_{<t})}[\ln\ p(o_t|s_t)] - \mathbb{E}_{q(s_{t-1}|o_{\leq t-1}, a_{<t-1})}[\mathrm{KL}[q(s_t|o_{\leq t}, a_{<t}) p(s_t|s_{t-1}, a_{t-1})]] \right)$$

Similarly, we can also compute the variational lower bound for the reward log-likelihood.

Since the RSSM is trained to predict forward in the latent space, the agent can perform imaginary rollouts in the learned latent space and choose the action which leads to highest return among the imagined rollouts. Similar to PETS, Hafner et al. (2018) use a CEM based planner to choose the next action. However, PETS does

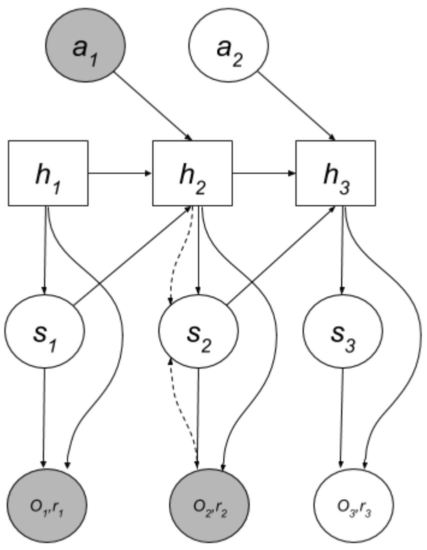


Figure 5.1: Recurrent State Space Model. The model observes the first two time steps and predicts the third. Circles represent stochastic variables and squares deterministic variables. Solid lines denote the generative process and dashed lines the inference model.

CEM planning in the observation space while PlaNet does CEM planning in the learned latent space.

5.2 PG+: An algorithm for policy-guided planning in learned latent space

While PlaNet achieves state-of-the-art performance when compared to other model-based methods, it is still doing simple model predictive control on top of the learned model using CEM. While CEM is easy to perform in low-dimensional action spaces, it does not scale up well for high-dimensional action spaces. Also, CEM learning is only episodic. Hence, the CEM module is reinitialized to a standard Gaussian in every action step.

Recently, Wang and Ba (2020) proposed POPLIN, which combined PETS with a learned policy network. In POPLIN, along with the model of the world, authors also learn a policy network which takes the current state and predicts the action.

This policy network is learnt by behaviour cloning using the data collected by the agent up to that point. During the CEM planning stage, POPLIN uses the policy network in either of the following ways:

1. In POPLIN-A, the policy network and the learned model of the world are used to propose an action trajectory. A CEM module is then used to learn a noise sequence to be added to this action trajectory, such that the resulting action trajectory leads to the highest return.

2. In POPLIN-P, a CEM module is used to learn a noise distribution sequence to be added to the parameters of the policy network, such that the policy network can produce an action trajectory that leads to the highest return.

Wang and Ba (2020) reported that POPLIN-P achieves superior performance when compared to PETS and POPLIN-A in most of the tasks. However, extending POPLIN-P to PlaNet has the following challenges:

1. In the Mujoco domains under consideration, PETS had access to the true low-dimensional state space, which made the tasks easier when compared to PlaNet's raw image-based observation space. Hence, bootstrapping from the collected data with behaviour cloning was sufficient to learn a good policy for PETS. However, in the case of PlaNet, the policy needs to be learnt on the latent space, which is more challenging for the policy network.

2. The latent space is also learnt simultaneously and hence there is continuous drift in the state distribution seen by the policy network. This further increases the difficulty of the task.

3. Adding noise to the parameter space is a computationally intensive process. We need to compute a separate noise for every time step and also for every example in the mini-batch. Hence, we cannot mini-batch the policy network predictions, so the training is very slow when compared to POPLIN-A.

With this motivation in mind, we now describe our proposed algorithm, Policy Guided Planning in Latent Space (PGPLaS or PG+), which aims to improve the above mentioned issues in combining a policy network with PlaNet.

5.2.1 Learning a Policy Network

Given a collected dataset of state-action sequences, behaviour cloning (BC) aims to maximize the log-likelohood of the action given the state:

$$\mathbb{E}_{(s,a)\in\mathcal{D}} \log \pi_\theta(a|s) \tag{5.2}$$

where π is the policy network with parameters θ and $\mathcal{D}$ is the dataset collected so far. BC assumes that all actions collected in the dataset are "good" actions. However, learning a policy network in the latent space of PlaNet introduces additional challenges. In the initial stages of the learning, the latent state space evolves faster since it is also simultaneously learnt. So it is difficult to learn a good policy and this in turn affects the quality of the data collected. Fitting the low quality data leads to further degradation in the quality of the policy. To avoid this, we use the Monotonic Advantage Re-Weighted Imitation Learning (MARWIL) loss proposed by Wang et al. (2018). Since we also have access to the reward obtained by performing the action, MARWIL exploits this additional information to weigh the "good" actions more than the "bad" actions. Specifically, MARWIL maximizes the following advantage reweighted log-likelihood:

$$\mathbb{E}_{(s,a)\in\mathcal{D}} \exp(\beta \hat{A}^\pi(s,a)) \log\pi_\theta(a|s) \tag{5.3}$$

where β is a hyper-parameter. The advantage function $\hat{A}$ is defined as

$$\hat{A}(s,a) = (r - V_\theta(s))/c \tag{5.4}$$

where r is the reward observed after (s,a) and c is a normalization term for the advantage value that helps to maintain the scale of β across different environments, computed by using the moving average of the norm of the advantage:

$$c^2 = c^2 + 10^{-8}((r - V_\theta(s))^2 - c^2) \tag{5.5}$$

Note that when $\beta = 0$, the MARWIL loss is reduced to BC loss. Wang et al. (2018) used the MARWIL loss with batch data only to fit a policy. Our setting has two major differences:

1. We are updating the policy by using the data collected by the same policy.
2. We use MARWIL in latent space instead of observation space and this latent space is also simultaneously learnt.

The policy network is detached from the dynamics network and hence there is no gradient flow between the policy network and the dynamics network. In our ablation study, we show that it is necessary to detach the policy network for efficient learning.

Our experimental results show that one can indeed use a batch-RL algorithm like MARWIL on the top of a learnt latent space and achieve state-of-the-art performance. This opens up avenues for transferring advances in the batch RL or the offline RL setting to the model-based RL setting.

5.2.2 Computing Policy Noise Efficiently

POPLIN-P adds noise to the parameters of the policy network and uses CEM to find the best noise distribution, that results in a policy maximizing the return. Consider the following simple single hidden layer policy network:

$$h = f(Ws + b) \tag{5.6}$$

$$\pi = Vh + c \tag{5.7}$$

POPLIN-P adds noise to both the W matrix and the V matrix. Specifically, it samples W_n and V_n from the noise distribution and performs the following operation:

$$h = f((W + W_n)s + b) \tag{5.8}$$

$$\pi = (V + V_n)h + c \tag{5.9}$$

This type of policy noise has two limitations:

1. Since the noise is added to the parameters of the policy, we cannot benefit from mini-batching. Hence, training is extremely slow.

2. The dimensionality of the noise distribution is equal to the number of parameters in the policy network. This is much larger than the action space itself and CEM suffers from this increased dimensionality.

To avoid these limitations, we propose to add noise vectors to the input and hidden layers of the policy network as follows:

$$h = f(W(s + s_n) + b) \tag{5.10}$$

$$\pi = V(h + h_n) + c \tag{5.11}$$

The dimensioanlity of the noise distribution that models s_n and h_n is much smaller than that of POPLIN-P. Also, the matrix operations can be efficiently batched with this approach and hence we incur no extra computational overhead. One can see that the proposed state noise can be re-cast as parameter noise. However, this state noise does not cover the entire space of parameter noise. Empirically, this restricted but scalable noise itself is sufficient to achieve good performance. Our experimental results show that PG+ with state noise achieves superior performance when compared to PlaNet.

The complete method is presented in Algorithm-2. The high-level algorithm is similar to POPLIN-P, but we use the proposed state noise for scalable policy noise, the RSSM and PlaNet loss for learning the dynamics, and the MARWIL loss for policy updates.

Algorithm 2 PG+

```
1: Initialize policy network parameters $\theta$, dynamics network parameters $\phi$, data-set $\mathcal{D}$
2: while Training iterations not Finished do
3:     for $i^{th}$ time-step of the agent do                                ▷ Sampling Data
4:         Initialize policy state noise distribution. $\mu = \mu_0$, $\Sigma = \sigma_0^2 \mathbf{I}$
5:         for $j^{th}$ CEM Update do                                        ▷ CEM Planning
6:             Sample policy state noise sequences $\{\omega_i\}$ from $\mathcal{N}(\mu, \Sigma)$.
7:             for Every candidate $\omega_i$ do                             ▷ Trajectory Predicting
8:                 for $t = i$ to $i + \tau$, $s_{t+1} = f_\phi(s_{t+1}|s_t, a_t = \pi_\theta(s_t, \omega_t))$    ▷ $s_t$ is the latent state from RSSM
9:                 Evaluate expected reward of this candidate.
10:            end for
11:            Fit distribution of the elite candidates as $\mu', \Sigma'$.
12:            Update noise distribution $\mu = (1-\alpha)\mu + \alpha\mu'$, $\Sigma = (1-\alpha)\Sigma + \alpha\Sigma'$
13:        end for
14:        Select the first action from the optimal candidate action sequence.
15:        Add exploration noise to the selected action.
16:        Execute the action for *action_repeat* times.
17:    end for
18:    Update $\phi$ using data-set $\mathcal{D}$ and PlaNet loss.                ▷ Dynamics Update
19:    Detach the policy network and update $\theta$ using dataset $\mathcal{D}$ and MARWIL loss.    ▷ Policy Update
20: end while
```

5.3 Relation to other algorithms

PETS learns to predict forward in the observed state space and combines the learned model with a CEM search for decision-time planning. PlaNet improved PETS by learning to predict forward in the learned latent state space and doing a CEM search in the latent space, which helps PlaNet deal with high-dimensional observational better than PETS. PG+ improves PlaNet further by learning a policy network in the learned latent space and combining the policy network with the dynamics model during decision-time planning. Searching in the policy space is much more efficient than searching in the action space, since smaller changes to the policy network can easily explore significantly different policies in fewer CEM iterations.

POPLIN (Wang and Ba, 2020) is similar to PG+ since both algorithms combine policy networks with dynamics based decision-time planning. However, PG+ is different from POPLIN in the following aspects:

- POPLIN learns a policy in the original state space while PG+ learns the policy in the learned latent space. Simple behaviour cloning as used in POPLIN cannot be used in the learned latent space and hence PG+ uses MARWIL loss to learn the policy.
- POPLIN assumes access to true reward function while PG+ learns a reward function by itself.
- POPLIN proposes adding noise in either the action space (POPLIN-A) or in the policy parameter space (POPLIN-P). While the action space is low dimensional when compared to the parameter space, POPLIN-P is superior in performance when compared to POPLIN-A. PG+, on the other hand, proposes adding noise in the input and hidden states of the policy network. This combines the benefits of both POPLIN-A and POPLIN-P since state noise works in the low dimensional state space (input state and hidden state) and is still as effective as POPLIN-P.

PG+ is also related to Dreamer (Hafner et al., 2020) since Dreamer learns a policy on top of the learnt latent space of the dynamics model. However, Dreamer is not a decision-time planning algorithm. Dreamer directly takes actions based on the predictions by the policy network. The two approaches are useful in different scenarios, and our focus is on decision-time planning.

5.4 Experiments

In this section, we present our empirical evaluation of the PG+ algorithm. Code to reproduce the results is available at `https://github.com/amini2nt/mb_rainbow`.

5.4.1 Tasks

We evaluate PG+ on the following five visual control tasks of the DeepMind Control Suite (Tassa et al., 2020) illustrated in Figure-5.2: Cheetah Run, Reacher Easy, Walker Walk, Finger Spin, and Quadruple Run. All the DeepMind Control Suite tasks have an *action-repeat* hyper-parameter which controls the number of times each action is repeated. This hyper-parameter controls the difficulty of the task. Setting the action-repeat to 4 divides the horizon of the problem by 4 and hence the task becomes easier. Hafner et al. (2018) chose different action-repeat values for different tasks to make the tasks easy for PlaNet. To avoid these discrepancies, we decided to keep a uniform action-repeat of 2 for all the tasks. This approach is also followed by Dreamer (Hafner et al., 2020). In effect, we removed a few easy-to-solve tasks that PlaNet experiments considered, and added a few more challenging tasks.

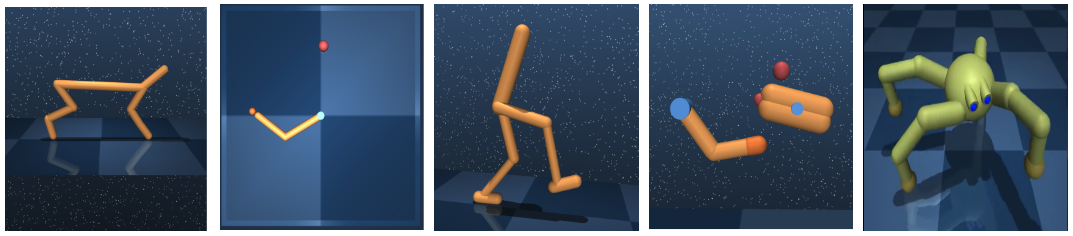


Figure 5.2: The five DeepMind Control tasks considered in our experiments (from left): Cheetah Run, Reacher Easy, Walker Walk, Finger Spin, Quadruple Run.

Each experiment is averaged over five random runs, and the mean and standard deviations of the results are presented. Following PlaNet, we trained all the agents for 1000 episodes.

5.4.2 Results

We use the PyTorch implementation of PlaNet provided by Arulkumaran (2019). First, to reproduce the PlaNet results, we ran experiments on the setting for cheetah-run recommended by Hafner et al. (2018) which is action-repeat = 4. The results are provided in Figure-5.3. After reproducing PlaNet results for cheetah-run with action-repeat = 4, we also benchmarked PG+ in the same easy setting. From Figure-5.3, we can see that PG+ achieves significantly better results when compared to PlaNet.

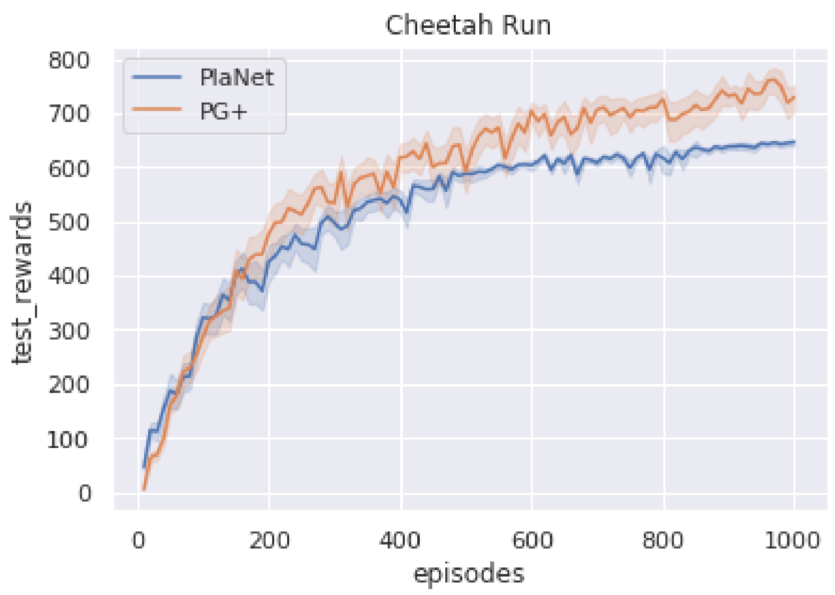


Figure 5.3: Results on Cheetah Run with action-repeat=4.

As a next step, we benchmarked both the algorithms in all the 5 control tasks with action-repeat = 2. The results are shown in Figure-5.4. From the figure, we can see that PG+ achieves superior asymptotic performance in three out of five tasks (cheetah-run, reacher-easy and finger-spin) and matches the final performance of PlaNet but with faster convergence in the remaining two tasks (Walker-walk and quadruple-run). For example, in the reacher-easy task, PlaNet with action-repeat = 4 (as reported in Hafner et al. (2018)) can solve the task and achieve the maximum of 1000 reward points. However, when we switch to action-repeat = 2, we can see that PlaNet cannot solve the task anymore, while PG+ solves the task within 500 episodes. This demonstrates that PG+ can learn more successfully in longer horizon tasks when compared to PlaNet. This is intuitive, since PG+ starts the decision-time planning with a much better initialization provided by the policy network and also employs a search process in a higher level policy space. PG+ is also more stable when compared to PlaNet. As we can see in reacher-easy, finger-spin, and quadruple-run, PlaNet has large variance in performance throughout training, while PG+ is more stable. The stability of the PG+ performance is intuitively due to the learnt policy network, which gets better over time.

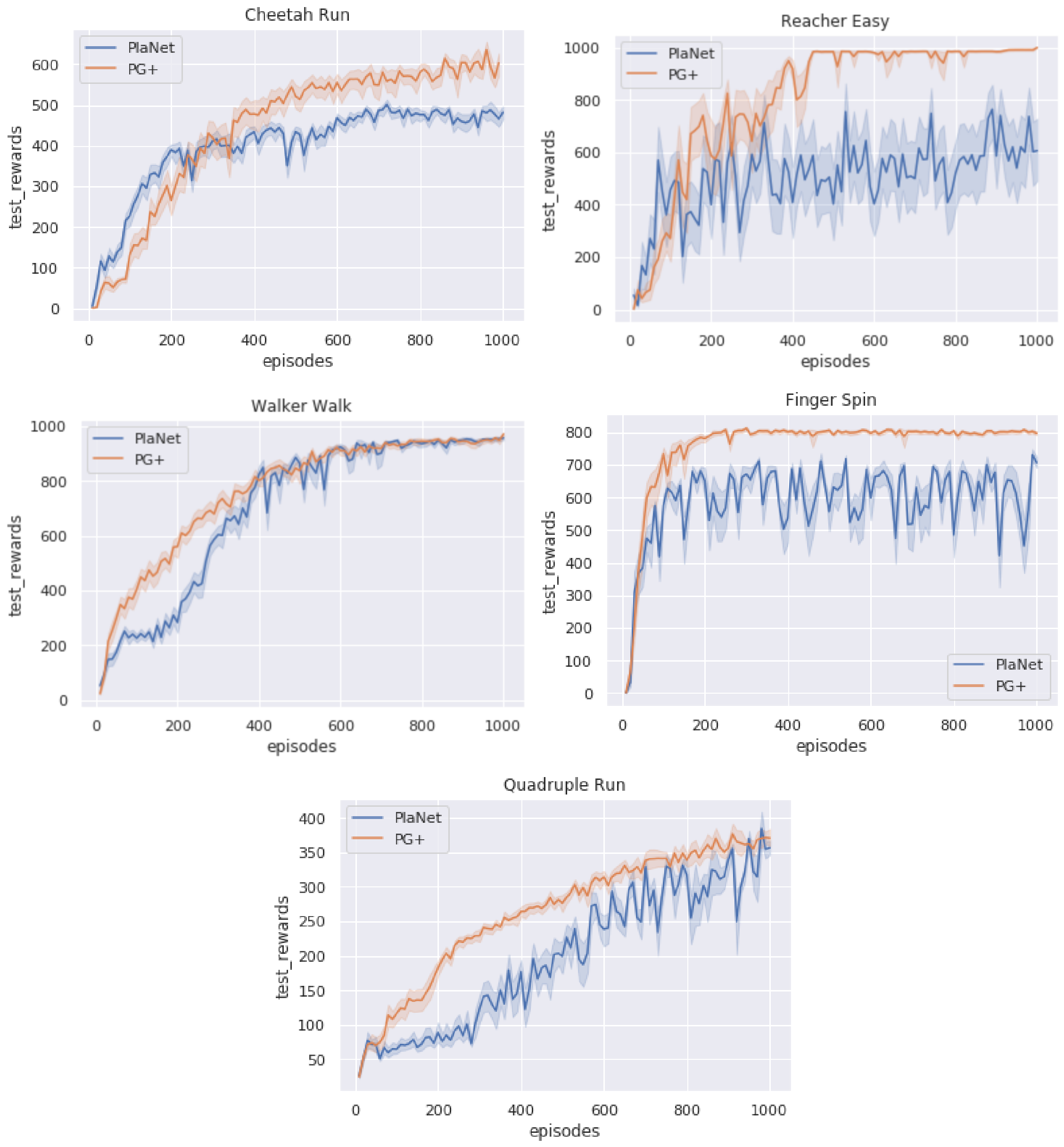


Figure 5.4: Comparison of PlaNet and PG+ in the following five DeepMind Control tasks: Cheetah Run, Reacher Easy, Walker Walk, Finger Spin, Quadruple Run. All experiments are run for 5 random seeds and the average performance and standard deviation are plotted. We used action-repeat=2 for all the tasks. The results demonstrate that PG+ performs better than PlaNet, both in terms of final performance and in terms of learning speed.

5.4.3 Ablation Study

PG+ has three major design choices: detaching the policy network from the dynamics network, using probabilistic output for the policy network, and using the MARWIL loss for training the policy network. To understand the relative importance of these three design choices, we conducted an ablation study of PG+ for the cheetah-run task. The results are reported in Figure-5.5. From the figure, we can see that not detaching the policy network from the dynamics network (PG+ no-detach) performs the worst. We hypothesize that in the beginning of the training, the policy network will see a huge shift in the state distributions due to the fact that the latent state space is also simultaneously learnt. Passing gradients from the policy network to the dynamics networks slows down the dynamics learning and hence makes the entire learning process hard.

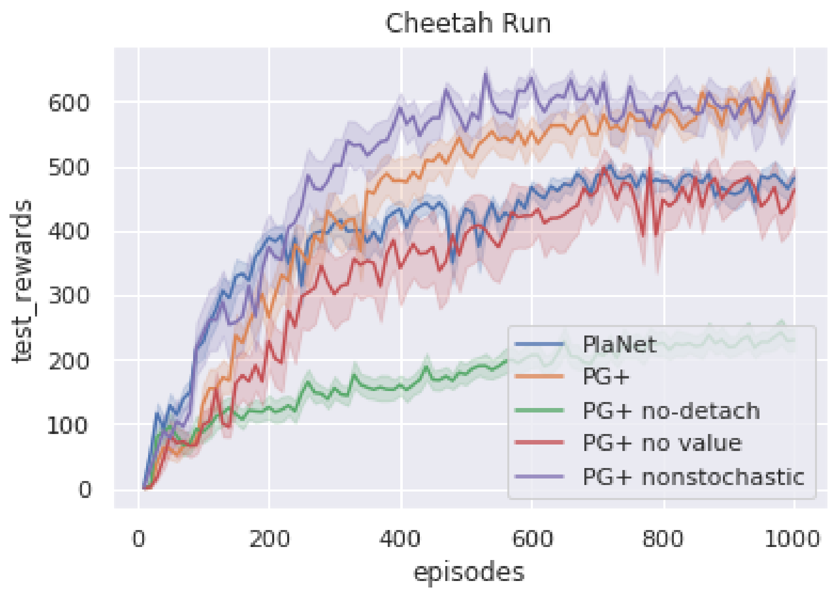


Figure 5.5: Ablation of different components in PG+ for Cheetah Run task with action-repeat=2.

Using a standard BC loss instead of MARWIL loss (PG+ no value) results in a drop in PG+ performance below the performance of PlaNet. This clearly demonstrates the importance of using a better imitation learning algorithm to tackle the challenges in training a policy network that has to work on top of an evolving latent state distribution. Finally, there is no clear benefit in using a policy network with probabilistic output, since PG+ with a deterministic output policy network (PG+ nonstochastic) achieves similar results.

5.5 Discussion

In this chapter, we introduced PG+, a model-based RL algorithm that combines a dynamics prediction network, policy network, and search based action selection module. PG+ strictly improves the performance of the previously proposed PlaNet algorithm. The improvements come from the fact that PG+ uses a learned policy network to guide the search process. We also proposed a scalable way to add noise to the policy network parameters, which could be applied in other settings, including for better exploration algorithms. PG+ demonstrates that any recent advance in batch RL can be extended to thee model-based RL setting.

6

Conclusion and Future Work

This thesis has focused on model-based deep reinforcement learning. As a first contribution, we provided a detailed empirical analysis of PETS (Chua et al., 2018), a recently proposed model-based deep RL algorithm. PETS was the first model-based RL algorithm that achieved superior asymptotic performance when compared to state-of-the-art model-free RL algorithms, while requiring orders of magnitude fewer samples. Our analysis highlights that by improving the accuracy of both the model and the search process, through longer training and increased number of CEM iterations respectively, the sample efficiency of PETS can be significantly improved. While improving the accuracy of the model does not add any significant computational overhead, increasing the accuracy of search process requires more computation. We would like to highlight that PETS with our suggested improvements achieved the same or better performance as POPLIN (Wang and Ba, 2020), an algorithm which was presented as superior in the the literature, while still taking less training time.

As a second contribution, we introduced PG+, a new model-based deep RL algorithm. PG+ improves upon PlaNet (Hafner et al., 2018), the previous state-of-the-art algorithm for decision-time planning. PG+ combines PlaNet with a policy network in the simultaneously learnt latent space, which results in superior performance, faster convergence and more stable training process on a suite of challenging visual control tasks. PG+ highlights the fact that recent advances in batch RL algorithms can be used to improve the performance of model-based RL algorithms. In the future, we would like to explore this direction further, by investigating other ideas from batch RL that can be used for model-based RL.

While all the experiments in this thesis focused on continuous control problems, exploring empirically the use of model-based RL algorithms for decision-time planning in other types of problems, such as discrete control, is an interesting future direction. Finally, we would like to mention that an interesting future research direction would be to further explore ways of stabilizing these kinds of algorithms when the state distribution changes, which would be very useful in non-stationary, continual learning RL problems.

Bibliography

Mohammad Amini, Doina Precup, and Sarath Chandar. Policy guided planning in learned latent space. In *NeurIPS Deep Reinforcement Learning Workshop*, 2020.

Kai Arulkumaran, 2019. URL https://github.com/Kaixhin/PlaNet.

Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

Richard Bellman. Dynamic programming and lagrange multipliers. *Proceedings of the National Academy of Sciences of the United States of America*, 42(10):767, 1956.

Zdravko I Botev, Dirk P Kroese, Reuven Y Rubinstein, and Pierre L'Ecuyer. The cross-entropy method for optimization. In *Handbook of statistics*, volume 31, pages 35–59. Elsevier, 2013.

Roberto Calandra, Jan Peters, Carl Edward Rasmussen, and Marc Peter Deisenroth. Manifold gaussian processes for regression. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pages 3338–3345. IEEE, 2016.

Eduardo F Camacho and Carlos Bordons Alba. *Model predictive control*. Springer Science & Business Media, 2013.

Kurtland Chua, Roberto Calandra, Rowan McAllister, and Sergey Levine. Deep reinforcement learning in a handful of trials using probabilistic dynamics models. In *Advances in Neural Information Processing Systems*, pages 4754–4765, 2018.

Stefan Depeweg, José Miguel Hernández-Lobato, Finale Doshi-Velez, and Steffen Udluft. Decomposition of uncertainty in bayesian deep learning for efficient and risk-sensitive learning. *arXiv preprint arXiv:1710.07283*, 2017.

David Ha and Jürgen Schmidhuber. Recurrent world models facilitate policy evolution. In *Advances in Neural Information Processing Systems*, pages 2450–2462, 2018.

Danijar Hafner, Timothy P. Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. Learning latent dynamics for planning from pixels. *CoRR*, abs/1811.04551, 2018. URL `http://arxiv.org/abs/1811.04551`.

Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to control: Learning behaviors by latent imagination. In *International Conference on Learning Representations*, 2020. URL `https://openreview.net/forum?id=S1lOTC4tDS`.

Nikolaus Hansen. The cma evolution strategy: A tutorial. *arXiv preprint arXiv:1604.00772*, 2016.

Mikael Henaff, William F Whitney, and Yann LeCun. Model-based planning with discrete and continuous actions. *arXiv preprint arXiv:1705.07177*, 2017.

Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

Ronald A Howard. Dynamic programming and markov processes. 1960.

Andras Gabor Kupcsik, Marc Peter Deisenroth, Jan Peters, and Gerhard Neumann. Data-efficient generalization of robot skills with contextual policy search. In *Twenty-Seventh AAAI Conference on Artificial Intelligence*, 2013.

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.

Anusha Nagabandi, Gregory Kahn, Ronald S Fearing, and Sergey Levine. Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7559–7566. IEEE, 2018.

Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy P. Lillicrap, and David Silver. Mastering atari, go, chess and shogi by planning with a learned model. *CoRR*, abs/1911.08265, 2019.

John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897, 2015.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676): 354–359, 2017.

Aravind Srinivas, Allan Jabri, Pieter Abbeel, Sergey Levine, and Chelsea Finn. Universal planning networks: Learning generalizable representations for visuomotor control. In *International Conference on Machine Learning*, pages 4732–4741, 2018.

Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

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

Yuval Tassa, Saran Tunyasuvunakool, Alistair Muldal, Yotam Doron, Siqi Liu, Steven Bohez, Josh Merel, Tom Erez, Timothy Lillicrap, and Nicolas Heess. dm$_c ontrol : Software and tasks for continuous control$, 2020.

Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

Qing Wang, Jiechao Xiong, Lei Han, peng sun, Han Liu, and Tong Zhang. Exponentially weighted imitation learning for batched historical data. In *Advances in Neural Information Processing Systems 31*, pages 6288–6297. 2018. URL http://papers.nips.cc/paper/7866-exponentially-weighted-imitation-learning-for-batched-historical-data.pdf.

Tingwu Wang and Jimmy Ba. Exploring model-based planning with policy networks. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=H1exf64KwH.

Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4): 279–292, 1992.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.