Jens Erik Kveen

Sample Efficient Deep Reinforcement Learning via Model-Ensemble-Based Exploration

Master's thesis in Cybernetics and Robotics
Supervisor: Prof. Jan Tommy Gravdahl
Co-supervisor: Akhil S. Anand
June 2022

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Kunnskap for en bedre verden

Jens Erik Kveen

Sample Efficient Deep Reinforcement Learning via Model-Ensemble-Based Exploration

Master's thesis in Cybernetics and Robotics
Supervisor: Prof. Jan Tommy Gravdahl
Co-supervisor: Akhil S. Anand
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

Preface

This master's thesis is submitted as a part of the requirements for a master's degree at the Department of Engineering Cybernetics at the Norwegian University of Science and Technology. The work presented in this thesis has been carried out under the supervision of Prof. Jan Tommy Gravdahl and PhD Candidate Akhil S. Anand at the Department of Engineering Cybernetics, NTNU.

This master's thesis is a continuation of a specialization project [1] conducted during the autumn of 2021. As is customary, the specialization project is not published. Therefore, some important background theory from the project is restated fully in this report to provide the best reading experience. The following sections are adapted from [1]:

- Part of chapter 2 (Specifically section 2.1)
- The subsection *Learning Dynamics* in 2.4.1

All implementations done during the project are based on open source libraries or my own work. More specifically, the implementation of the methods developed in this thesis is based on the implementation of similar methods found in the *mbrl-lib* framework developed by Pineda et al. [2] at Meta Research. During the master's project, I have further extended this framework with the methods developed in this thesis as well as the model implemented in the preliminary project.

Acknowledgements

Firstly, I would like to thank my co-supervisor, PhD Candidate Akhil S. Anand, for a thorough follow-up, fruitful discussions and valuable feedback throughout the whole project. I would also like to thank my supervisor, Professor Jan Tommy Gravdahl, for valuable insight and support.

Secondly, I would like to thank my fellow graduates. Without our numerous lunch gatherings, coffee breaks and quizzes, this semester would not have been as joyful as it has been.

Lastly, I would like to thank my family for their encouragement and support from start to finish. Without you, none of this would have been possible.

Jens Erik Kveen

Trondheim, June 2022

Abstract

Model-based reinforcement learning is able to achieve much higher sample efficiency than model-free methods, making them more suitable for practical applications. However, model-based methods lack the performance of their model-free counterparts. This thesis contributes to filling this performance gap with a model-based reinforcement learning method based on targeted exploration by estimating model uncertainty. The performance of model-based reinforcement learning methods relies on accurately learning a model of the environment, which can be exploited during planning. However, current methods focus more on exploiting the model than focusing on improving it. State of the art model-based reinforcement learning methods tries to maximize the reward, focusing on exploiting what is already learnt. However, exploration is a critical part of reinforcement learning, but most model-based methods lack an explicit way of exploring the state space. This thesis presents a model-based reinforcement learning method which utilizes estimated model uncertainty with a deep network ensemble to target exploration towards states where the model uncertainty is high. By balancing exploration and exploitation, the algorithm aims to exploit the model when uncertainty is low and explore when uncertainty is high. The method is tested on several control benchmark tasks in the OpenAI gym framework. Results show that through targeted exploration the algorithm achieves lower model uncertainty on all benchmark tasks while maintaining or improving the performance in terms of sample efficiency compared to current model-based methods.

Sammendrag

Modellbasert forsterkende læring er i stand til å oppnå mye høyere prøve-effektivitet enn modellfrie metoder, noe som gjør dem mer egnet for praktiske anvendelser. Imidlertid mangler modellbaserte metoder ytelsen til sine modellfrie motparter. Denne oppgaven bidrar til å fylle dette ytelses-gapet med en modellbasert forsterkende læringsmetode basert på målrettet utforskning ved å estimere modellusikkerhet. Ytelsen til modellbaserte forsterkende læringsmetoder er avhengig av å lære en nøyaktig modell av miljøets dynamikk, som kan utnyttes under planlegging. Nåværende metoder fokuserer imidlertid mer på å utnytte modellen enn på å forbedre den. Toppmoderne modellbaserte forsterkende læringsmetoder prøver å maksimere belønningen, med fokus på å utnytte det som allerede er lært. Utforskning er imidlertid en kritisk del av forsterkende læring, men de fleste modellbaserte metoder mangler en eksplisitt måte å utforske tilstandsrommet på. Denne oppgaven presenterer en modellbasert forsterkningslæringsmetode som bruker estimert modellusikkerhet med et dypt nettverksensemble for å målrette utforskning mot tilstnader hvor modellusikkerheten er høy. Ved å balansere utforsking og utnyttelse har algoritmen som mål å utnytte modellen når usikkerheten er lav og utforske når usikkerheten er høy. Metoden er testet på flere vanskelige oppgave i OpenAI gym-rammeverket. Resultatene viser at gjennom målrettet utforskning oppnår algoritmen lavere modellusikkerhet på alle benchmarkoppgaver samtidig som ytelsen opprettholdes eller forbedres når det gjelder prøveeffektivitet sammenlignet med gjeldende modellbaserte metoder.

Contents

List of Figures **xi**

List of Tables **xv**

List of Algorithms **xvi**

Acronyms **xvii**

1 Introduction **1**

1.1 Contributions 3

1.2 Outline 4

2 Background **5**

2.1 Neural Network Models 6

2.1.1 Fully Connected Neural Networks 6

2.1.2 Probabilistic Neural Networks 8

2.1.3 Ensembles 10

2.2 Cross Entropy Method 12
2.2.1 Standard CEM 12
2.2.2 CEM for Trajectory Optimization 13
2.3 Reinforcement Learning 15
2.3.1 Markov Decision Processes 16
2.3.2 Policies and Value Functions 19
2.3.3 Optimal Policies 22
2.3.4 The Reinforcement Learning Problem 23
2.4 Model-Based Reinforcement Learning 25
2.4.1 Model Learning 26
2.4.2 Model Utilization 29
2.4.3 PETS - Probabilistic Ensembles with Trajectory Sampling 33
2.5 Open AI gym 37
2.5.1 Gym Environments 37
2.5.2 MuJoCo 38
2.6 MBRL-Lib 39
2.6.1 Replay Buffer 39
2.6.2 Dynamics Models 39
2.6.3 Configuration 40

3 Developing an RL algorithm **43**
3.1 Previous Work and Inspirations 44
3.2 The Exploration Algorithm 45
3.2.1 Uncertainty-Based Exploration 45
3.2.2 Balancing Exploration and Exploitation 47
3.2.3 Evaluating Model Uncertainty 49
3.2.4 Algorithm Summary 50

4 Implementation Details **53**
4.1 Model Exploration Wrapper 54
4.2 Sampling Data for Uncertainty Estimation 56
4.3 Schematic Implementation Overview 58
4.4 Configurations 59
4.5 Task Environments 63
4.5.1 Cartpole 63
4.5.2 Halfcheetah 64
4.5.3 2D Walker 65
4.5.4 7-DOF Reacher 66
4.5.5 Summary 68

5 Experimental Results **69**
5.1 2D Walker 70
5.2 Halfcheetah 74
5.3 Reacher 78
5.4 Cartpole 82

5.5 Model Transferability 85

6 Discussion 89

6.1 Performance Effects of Exploration 90

6.1.1 Low Exploration Utilization 90

6.1.2 Higher Exploration Utilization 92

6.1.3 Model Uncertainty 95

6.2 Uncertainty Estimation and Exploration Weighting 98

6.2.1 Limitations of the Exploration Weighting 98

6.3 Model Generalizability 100

7 Conclusion and Future Work 103

7.1 Conclusion 104

7.2 Future Work 106

Bibliography 107

List of Figures

2.1 Structure of a single node in an FCNN [23] 7
2.2 A random state of a chess game, green arrows highlights some of the available actions 16
2.3 The interaction between an agent and the environment as depicted in [5] 24
2.4 Hybrid model-based / model-free learning, as depicted in [11] 30
2.5 Explicit model planning, as depicted in [11] 31

4.1 A high level schematic overview of the algorithm implementation. The stapled line of training indicates that it is not happening at every timestep. The two-way arrows indicate communications between MPC controller and model wrapper environment when optimizing action sequences. . 59
4.2 Screenshot of the cartpole environment 63
4.3 Screenshot of the halfcheetah environment 64
4.4 Screenshot of the 2D walker environment. 65

4.5 Screenshot of the reacher environment. The red ball corresponds to the desired target position of the end effector . . 67

5.1 Algorithm performance on the 2D walker environment. The plot shows mean of three different test with different random seeds. Shaded area corresponds to the standard deviation between each seed. 70
5.2 Development of the weighting parameter ε through training on the walker environment. Random and policy refers π_u being either random or following the exploitation policy of the current model. 71
5.3 Development of model uncertainty during training. Uncertainty is evaluated on random state data using model checkpoints every 10 000 training steps. 72
5.4 Mean agent performance after 50, 100, 150 and 200k training steps on the 2D walker task. The performance is evaluated without utilizing exploration, similar to policy used in PETS 73
5.5 Algorithm performance on the halfcheetah environment. The plot shows mean of three different test with different random seeds. Shaded area corresponds to the standard deviation between each seed. 74
5.6 Development of the weighting parameter ε through training on the halfcheetah environment. Random and policy refers π_u being either random or following the exploitation policy of the current model. 75

5.7 Development of model uncertainty during training. Uncertainty is evaluated on random state data using model checkpoints every 10 000 training steps. The first uncertainty estimate is removed to better visualize the later development. 76
5.8 Mean agent performance after 50, 100, 150 and 200k training steps on the halfcheetah task. The performance is evaluated without utilizing exploration, similar to policy used in PETS 77
5.9 Algorithm performance on the 3D reacher environment. The plot shows the mean of three different tests with different random seeds. The shaded area corresponds to the standard deviation between each seed. The initial reward for the two exploration agents is about -900 and -600 for random and policy, respectively. 78
5.10 Development of the weighting parameter ε through training on the 3D reacher environment. Random and policy refers π_u being either random or following the exploitation policy of the current model. 79
5.11 Development of model uncertainty during training on the reacher environment. Uncertainty is evaluated on random state data using model checkpoints every 600 training steps. 80

5.12 Mean agent performance after 1800, 4200, 9000 and 15000 training steps on the 7-DOF reacher task. The performance is evaluated without utilizing exploration, similar to policy used in PETS 81
5.13 Algorithm performance on the cartpole environment. The plot shows mean of three different test with different random seeds. Shaded area corresponds to the standard deviation between each seed. 82
5.14 Development of the weighting parameter ε through training on the cartpole environment. Random and policy refers π_u being either random or following the exploitation policy of the current model. 83
5.15 Development of model uncertainty throughout training on cartpole environment. 84
5.16 Mean agent performance after 3000 training steps. The performance is evaluated without exploration, similar to the policy used in PETS 84
5.17 Mean performance of three trials on the halfcheetah environment, where agents aim to track a desired velocity opposed to running as fast as possible. 86
5.18 Mean performance of 10 trials in the walker environment, where agents aim to walk backwards instead of forwards. . 87

List of Tables

4.1 Space dimensions and task horizon for each gym environment 68

4.2 Reward and termination functions for each gym environment. x_t and z_t corresponds to horizontal and vertical Cartesian position respectively. For the reacher; g and ee_t are the goal and end effector positions. 68

List of Algorithms

2.1 Cross Entropy Method [3] 13

2.2 CEM for Trajectory Optimization 14

2.3 PETS [4] 36

3.1 The Exploration Algorithm 51

Acronyms

CEM Cross-Entropy Method. viii, xvi, 12–14, 32, 33, 35, 46, 50, 51, 54, 58

DOF Degrees of Freedom. 63, 66

FCNN Fully Connected Neural Network. xi, 6, 7

GP Gaussian Processes. 27, 31

MBRL Model-Based Reinforcement Learning. 25, 27, 29, 32, 33, 39–41, 43, 106

MDP Markov Decision Process. 15, 16, 18–20, 23, 25, 37

MPC Model Predictive Control. xi, 13, 31–33, 35, 45, 50, 56, 59, 69

MSE Mean Square Error. 8, 9

NN Neural Network. 6, 8, 10, 11

PETS Probabilistic Ensembles with Trajectory Sampling. xii–xiv, xvi, 2, 3, 25, 31–36, 40, 44–46, 50, 54, 61, 69–82, 84–86, 90–96, 99–102, 104, 105

PILCO Probabilistic Inference for Learning COntrol. 27, 31

PNN Probabilistic Neural Network. 8, 9, 27, 34

RL Reinforcement Learning. 15, 19, 21, 23–26, 29, 30, 32, 33, 37, 38, 44, 47, 106

TS Trajectory Sampling. 34–36, 40, 50, 51

Chapter 1

Introduction

Learning from interacting with our surroundings is arguably the most natural and intuitive way of learning. The way humans and animals learn from experience is the fundamental idea behind reinforcement learning, which is the branch of machine learning focusing on learning behaviour through interactions with the environment[5]. Reinforcement learning has shown promising results in solving a number of tasks, including beating the best human players in a variety of games[6, 7, 8] as well as learning automatic control of physical systems[9, 10]. However, reinforcement learning methods often require an impractically large amount of interactions with the environment in order to perform well, often limiting their applicability outside of simulated systems. Therefore, recent research has focused on sample efficiency and model-based reinforcement learning[11].

Model-based reinforcement learning methods build a model of the environment dynamics parallel to learning behaviour. By utilizing this learnt information in combination with policy updates, the number of environment interactions can be reduced substantially[12]. However, the performance of model-based methods is highly dependent on the accuracy of the learnt model. Therefore they often tend to converge to a lower performance than their model-free counterparts. With the advancements in deep supervised learning, model-based methods have recently adapted the use of deep neural networks to model dynamics [13]. Replacing the previous standard of Gaussian processes[14], the expressiveness of neural networks has advanced the performance of model-based methods to be comparable to model-free methods with substantially higher sample efficiency [4, 15].

Current research in model-based methods focuses on both learning the model [16] and utilizing the learnt model. Learned dynamics models can be used to improve existing model-free methods by speeding up the learning of value functions[17] or combining model-free and model-based data to perform policy updates[15, 18]. However, the currently considered state-of-the-art method PETS utilizes the learnt dynamics for planning by incorporating uncertainty aware neural network ensembles and state propagation [4]. Moreover, by utilizing stochastic network models PETS achieves performance comparable to the model-free methods like Soft Actor Critic[19] in orders of magnitude fewer environment samples[4].

When simultaneously learning both a dynamics model and a policy, the importance of exploration only increases, as performance is heavily reliant

on the model's accuracy. Exploration has seen much research in model-free methods[20]; however, current model-based methods lack an explicit way of exploration. Although Chua et al. [4] achieves high performance with PETS, this can be further improved with the use of exploration. This thesis presents a model-based algorithm based on PETS, which utilizes the probabilistic ensemble model's ability to estimate uncertainty to target exploration. Resulting in reduced model uncertainty and increased sample efficiency in several challenging benchmark tasks.

1.1 Contributions

This thesis aims to explore further the potential of uncertainty aware models in model-based reinforcement learning. By using the PETS algorithm as a baseline, an algorithm utilizing the model ensemble's ability to estimate epistemic uncertainty for exploration is developed and assessed in several benchmark tasks. The thesis' main contributions to the field of model-based reinforcement learning are as follows:

- Literature review
- The developed algorithm implemented in the existing mbrl framework
- Analysis of the algorithm's performance on several control benchmarks

1.2 Outline

The report is organized into seven chapters, with this being the first one serving as an introduction to the thesis. The following chapter 2 provides the thesis' theoretical background in deep model-based reinforcement learning. Then the developed algorithm is presented in detail in chapter 3, followed by details regarding the implementation of the algorithm and test environments in chapter 4. The experimental results of the implemented algorithm are presented in chapter 5. Lastly, the results are discussed in chapter 6, ending with a conclusion and future directions in chapter 7.

Chapter 2

Background

In this chapter, the theoretical background for the thesis is presented. The chapter is divided into six sections. The first two sections present some preliminary theory about neural networks and the cross entropy method for optimization. Then the focus shifts to reinforcement learning. Firstly, reinforcement learning is introduced and formalized as Markov decision processes, and the concept of policies is introduced, followed by a section presenting concepts and methods in model-based reinforcement learning. Finally, the last two sections present important software frameworks in reinforcement learning that are used later in the thesis.

2.1 Neural Network Models

Neural Networks (NN) are efficient and universal function approximators[21]. There exist several different structures of NNs that are used for different purposes, for instance, convolutional networks[22], which are used in image classification and similar tasks. However, this section presents neural network structures often used in modelling dynamics.

2.1.1 Fully Connected Neural Networks

An FCNN consists of layers of nodes or neurons where each node is connected to every node in the following layer. When an input is passed to a node, the signal is fed through a set of weights and a bias. In order for the network to approximate non-linear functions the output of a node is passed through a nonlinear activation function. This whole process is depicted in figure 2.1.

The output of each node is then passed as input to every node in the following layer. Mathematically, the process from input $\boldsymbol{x}$ to output $\boldsymbol{y}$ in a complete network with n layers can be summarized by

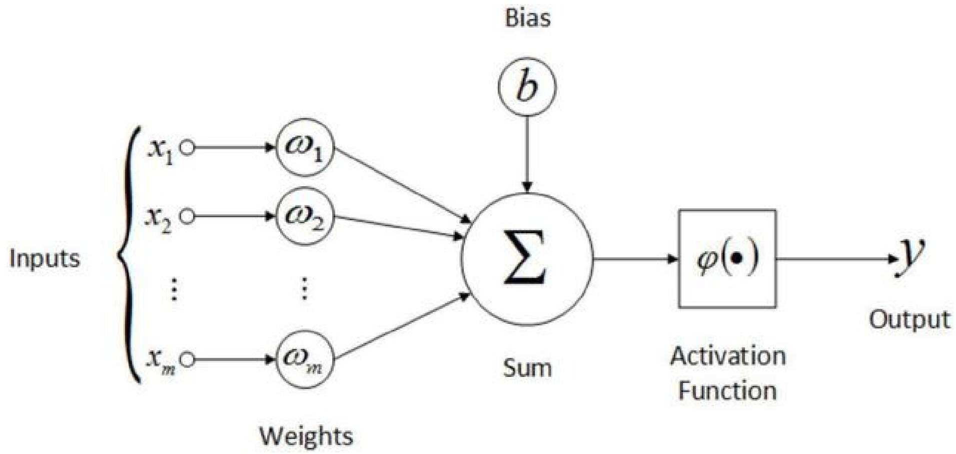


Figure 2.1: Structure of a single node in an FCNN [23]

$$\boldsymbol{l}_0 = \boldsymbol{x} \tag{2.1a}$$

$$\boldsymbol{l}_i = \varphi_i \left(\mathbf{W}_i \boldsymbol{l}_{i-1} + \boldsymbol{b}_i \right), \quad i \in \{1, \ldots, n\} \tag{2.1b}$$

$$\boldsymbol{y} = \boldsymbol{l}_n \tag{2.1c}$$

Where $\boldsymbol{W}_i$, $\boldsymbol{b}_i$ and φ_i corresponds to the weights, bias' and activation function of the i-th layer, $\boldsymbol{l}_i$, respectively. For simplicity, the learnable parameters of a given NN will be denoted by θ; in this case, θ corresponds to $\boldsymbol{W}$ and $\boldsymbol{b}$.

The network learns by minimizing a cost function, often referred to as the objective function or loss function. This cost function is based on the

error of the model's predictions. Therefore, a common choice is the mean square error (MSE)

$$\text{MSE} = \frac{1}{N}\sum_{n=1}^{N}(\boldsymbol{y}_i - \hat{\boldsymbol{y}}_i)^2 = \frac{1}{N}\sum_{n=1}^{N}(\boldsymbol{y}_i - \text{NN}_\theta(\boldsymbol{x}_i))^2 \tag{2.2}$$

To train the network, the learnable parameters θ are updated through backpropagation[24]. Backpropagation takes advantage of the differentiability of the loss function to compute the gradients with respect to network parameters efficiently. These gradients are then used to optimize the network parameters through some optimization method, like stochastic gradient descent[24].

2.1.2 Probabilistic Neural Networks

A probabilistic neural network (PNN) is a particular case of the deterministic NN presented in section 2.1.1. The difference lies in the network output. Where deterministic NNs output the prediction directly, PNNs instead output a parametrized probability distribution of the prediction. For example, consider a Gaussian distribution; the PNN model would, in this case, output a vector of means and corresponding variances. The final prediction is then drawn from the corresponding Gaussian distribution. Following the same structure as equation 2.1, the PNN forward pass can be summarized by

$$[\boldsymbol{\mu}, \Sigma] = \text{PNN}_\theta(\boldsymbol{x}) \tag{2.3a}$$

$$\boldsymbol{y} \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma) \tag{2.3b}$$

Since the output of a PNN is not a direct prediction, a loss function such as MSE is not applicable, as there is not necessarily a direct comparison between the distribution parameters and the final prediction. Instead, as discussed in [4], one can use the negative log prediction as a loss function:

$$\text{loss} = -\sum_{n=1}^{N} \log \text{PNN}_\theta(\boldsymbol{x}_n) = -\sum_{n=1}^{N} \log \Pr(\boldsymbol{y}_n | \boldsymbol{x}_n) \tag{2.4}$$

Continuing with the Gaussian example from before, this equates to:

$$\text{loss} = -\sum_{n=1}^{N} (\boldsymbol{y}_n - \boldsymbol{\mu}_n)\Sigma_n^{-1}(\boldsymbol{y}_n - \boldsymbol{\mu}_n) + \log \det(\Sigma_n) \tag{2.5}$$

By outputting a probability distribution rather than direct predictions, PNNs are able to capture aleatoric uncertainty in the data. Knowing the uncertainty of a model prediction can be beneficial as it an estimate the model's precision. In addition, estimating of the aleatoric uncertainty can help uncover unknown uncertainties in the training data.

2.1.3 Ensembles

Model ensembling is a method used to reduce epistemic uncertainty in model predictions. A model ensemble is a collection of different NN models which are trained in parallel with one another on the same training data. Due to variations in parameter initialization or stochastic elements of the training, each model in the ensemble learns to adapt to the given data in slightly different ways. This results in a set of similar yet different models that fit the same data. By combining predictions from multiple models, an ensemble is able to reduce model uncertainty compared to a single network.

As an example, consider an ensemble of models $\{f_i(x)\}_{i=1}^{n}$, that outputs a prediction as the mean of all member model outputs. Assume model f_i has a prediction error given by e_i on some data sample, and let the variance and covariance of these errors be given by $\mathbb{E}[e_i^2] = v$ and $\mathbb{E}[e_i e_j] = c$. The total expected square error of the ensemble is then given by:

$$\mathbb{E}\left[\left(\frac{1}{n}\sum_{i=1}^{n} e_i\right)^2\right] = \frac{1}{n}v + \frac{n-1}{n}c = \frac{1}{n}(v-c) + c \leq v \tag{2.6}$$

In the worst case, all member models are perfectly correlated ($c = v$). Thus the ensemble is no better than just a single model. However, when this is not the case, the resulting uncertainty of the ensemble prediction is lower than each member model's uncertainty.

Utilizing ensembles of models instead of single NN models can therefore help to reduce model uncertainty. Furthermore, it can also be an excellent way to quantify model uncertainty, providing reasonable uncertainty estimates through model variance [25].

2.2 Cross Entropy Method

The cross-entropy method (CEM) is a derivative-free, sampling-based optimization method [26]. Due to its versatility and ability to solve difficult optimization problems, it offers a more sophisticated way of planning over an uncertain dynamics model compared to random shooting methods [4] [27].

2.2.1 Standard CEM

The goal of CEM, as all optimization methods, is to minimize a given objective function $f(x)$ by finding the optimal value of $x = x^*$. In CEM, a number of individual points are sampled from an initial distribution or population and evaluated on f. The samples are then sorted based on their cost, and a set number of the best samples called an elite-set is selected. The population parameters are then updated based on the elite set, and a new set of samples is drawn. After a some number of iterations, an estimate of the optimal point x^* is found as the best sample of the current elite set. Pseudocode for the plain CEM optimization, using a Gaussian distribution as population, is described in algorithm 2.1.

Algorithm 2.1: Cross Entropy Method [3]

input : N: Number of Samples, K: Size of elite-set, CEM_{iters}: Number of iterations, f: cost function

output : x^*: Optimal solution

1 Initialize μ, σ;
2 **for** $i \leftarrow 0$ **to** CEM_{iters} **do**
3   samples $\leftarrow$ N samples from $\mathcal{N}(\mu, \sigma)$;
4   cost $\leftarrow$ $f(x)$ for x in particles;
5   elite-set $\leftarrow$ The K best particles based on cost;
6   μ, σ $\leftarrow$ fit Gaussian distribution to elite-set;
7 **end**
8 return the best sample from the elite set

2.2.2 CEM for Trajectory Optimization

Utilizing CEM for planning is often done when other derivative-based optimization methods are unavailable due to a non-differentiable objective function. This is done by incorporating the CEM in algorithm 2.1 in an MPC framework. Instead of finding an optimal single value x^*, CEM is used to find an optimal sequence of actions a^*, which minimizes the cost of the resulting trajectory.

A common modification to the CEM in algorithm 2.1, when used in planning, is to use mean and standard deviations from the last timestep as initial distribution parameters for optimization in the next timestep[4, 28]. This modification is based on the intuition that the optimal action sequence is somewhat similar to the previous actions, thus reducing the number of

iterations by searching close to the optimum. Pseudocode for the modified CEM for trajectory optimization is given in algorithm 2.2

Algorithm 2.2: CEM for Trajectory Optimization

```
    input: N: Number of Samples, K: Size of elite-set, h: planning
           horizon, CEM_iters: Number of iterations, f: cost function
 1  Initialize μ_0, σ_0;
 2  for t ← 1 to T do
 3      μ_t ← μ_{t−1};
 4      σ_t ← σ_{t−1};
 5      for i ← 0 to CEM_iters do
 6          particles ← N samples from N(μ_t, σ_t);
 7          cost ← f(x) for x in particles;
 8          elite-set ← The K best particle based on cost;
 9          μ_t, σ_t ← fit Gaussian distribution to elite-set;
10      end
11      Execute the first action of the mean elite sequence;
12  end
```

2.3 Reinforcement Learning

Reinforcement Learning (RL) is one of the three main branches of machine learning, the other two being supervised and unsupervised learning. Where the two latter often handle tasks like classification or regression, RL is used to learn behaviour. An RL algorithm learns how to behave in an environment based on the received reward. Simply put, if the agent receives a positive or high reward, it is more likely to perform similar actions in the future and less prone to perform actions that yield negative or low reward.

RL algorithms try to mimic how a human or animal would learn. Through trial and error, it explores the state space of an environment and acts based on experience. Very similar to how one would train a dog, for example. Give it a treat when it behaves nicely, and it will learn to act similarly because it knows that the consequence is a rewarded treat. The dog in this example can be considered an RL agent. A trained agent is the result of an RL algorithm, comparable to a trained neural network in a classification task, for instance.

This section presents the mathematical preliminaries of RL and finally combines them to define the problem RL aims to solve. These preliminaries include defining Markov Decision Processes (MDP), policies and value functions, including what it means for a policy to be optimal.

2.3.1 Markov Decision Processes

A Markov decision process (MDP) is a mathematical structure or framework to describe interaction with an environment or sequential decision making. An MDP consists of states, actions, a transition function and a reward function. A formal definition of an MDP is given at the end of this subsection.

States

The possible states of an MDP is denoted by $\mathcal{S} = \{s_0, s_{,} \ldots, s_n\}$, where each $s \in \mathcal{S}$ uniquely describes a given instance of the environment. As an example, consider a game of chess. Each state must in this case describe the position of all the pieces on the board and which pieces are left on the board. In total, $\mathcal{S}$ then describes all possible configurations of a chessboard.

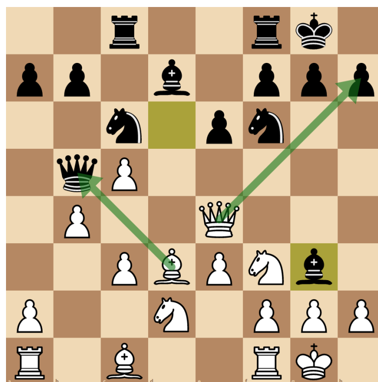


Figure 2.2: A random state of a chess game, green arrows highlights some of the available actions

Actions

The set $\mathcal{A} = \{a_0, a_1, \ldots, a_m\}$ describes all the possible actions that can be applied to the environment to change its state. In general not every action is possible at every state, therefore all the possible actions in a given state s is denoted by $\mathcal{A}(s) \subseteq \mathcal{A}$. In the chess example, $\mathcal{A}(s)$ corresponds to every legal move at a given board state; an example of some of these actions is shown in figure 2.2.

Transition Function

When applying an action $a \in \mathcal{A}$ in a given state $s \in \mathcal{S}$, the resulting state $s' \in \mathcal{S}$ is in generally given by a probability distribution. This probability distribution is called the transition function or transition operator and is defined as

$$\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \longrightarrow [0, 1] \tag{2.7}$$

$$(s, a, s') \longmapsto P(s' \mid s, a) \tag{2.8}$$

In a deterministic case, like the chess example, $\mathcal{T}$ simply defines what the resulting state is by a probability of 1 for the resulting state and 0 for all other states. However, not all systems are deterministic, thus in general $\mathcal{T}$ must fulfill certain criteria to be a properly defined distribution over the

possible states:

1. $0 \leq \mathcal{T}(s, a, s') \leq 1, \quad \forall\, s, s' \in \mathcal{S},\ a \in \mathcal{A}$
2. $\mathcal{T}(s, a, s') = 0, \quad \forall\, s, s' \in \mathcal{S},\ a \in \mathcal{A} \setminus \mathcal{A}(s)$
3. $\sum_{s' \in \mathcal{S}} \mathcal{T}(s, a, s') = 1, \quad \forall\, s \in \mathcal{S},\ a \in \mathcal{A}$

In practice, one often says that $\mathcal{T}$ defines the dynamics of an MDP system and is therefore sometimes referred to as a dynamics function.

Reward Function

Each action applied in an MDP system results in a state s' given by $\mathcal{T}$, but also a scalar reward r. This reward describes the quality of a given transistion, and the reward function is defined as:

$$\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \longrightarrow \mathbb{R} \tag{2.9}$$

$$(s, a, s') \longmapsto \mathcal{R}(s, a, s') \tag{2.10}$$

This reward function is what defines good and bad actions in an MDP. In some cases, as the chess example, the reward can be as simple as +1 for a winning move, -1 for a losing move, and 0 for every other transition. However, complexity varies greatly depending on the problem.

MDP Definition

Combining these four components discussed so far gives the formal definition of a Markov decision process:

Definition: A Markov decision process (MDP) is a tuple $\{\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}\}$. Where $\mathcal{S}$ and $\mathcal{A}$ are the sets of states and actions, called state space and action space, $\mathcal{T}$ is the transition operator and $\mathcal{R}$ is the reward function.

In addition, the transition operator must be *Markovian*, which means that for each state-action pair (s_t, a_t) the resulting transition is independent of all previous transitions:

$$P(s_{t+1} \mid s_t, a_t, s_{t-1}, a_{t-1}, \dots) = P(s_{t+1} \mid s_t, a_t) = \mathcal{T}(s_t, a_t, s_{t+1}) \tag{2.11}$$

2.3.2 Policies and Value Functions

Two key concepts of RL are policies and policy evaluation. When an agent learns from interacting with an environment to maximize reward, it is said to learn a policy. The quality of a given policy can be determined through policy evaluation with a value function.

Policy

A policy is a function that determines which action to execute in a given state. For explanation purposes consider a deterministic policy, which is a function $\pi : \mathcal{S} \longrightarrow \mathcal{A}$, such that $\pi(s) = a \in \mathcal{A}(s)$.

Through the use of a policy π an MDP can be controlled by cyclically applying the following steps from a given initial state s_0:

1. Pick an action based on the policy; $a_t = \pi(s_t)$
2. Apply action in MDP system. Through the underlying transition operator and reward function, the system will give a next state s_{t+1} and a reward r_t
3. Repeat with state $s_t = s_{t+1}$

Given an initial state s_0, this results in a series of triples (s_t, a_t, r_t) which describes a trajectory in the MDP system.

A policy is not necessarily deterministic and instead describes a probability distribution over possible actions given the current state. To distinguish between the deterministic and stochastic case, a stochastic policy is often denoted as $\pi(a \mid s)$ and equals the probability of choosing action a given the state s.

Value Function

In RL, the goal of an agent is not necessarily the reward gained only from a single step but rather the cumulative reward gained over a specific time horizon or period. In order to easily discuss this, the expected return from time t over a horizon T is defined as

$$G_t = \sum_{k=0}^{T} \gamma^k r_{t+k+1} \tag{2.12}$$

where $\gamma \in [0, 1]$ is known as the discount factor. A discount factor $\gamma < 1$ implies that immediate reward is more favorable than reward received further into the future. In the extreme case of $\gamma = 0$, the agent only cares about the immediate reward and dismiss any potential future reward.

Using equation 2.12, the *value function* of a state s under a policy π, denoted by $V_\pi(s)$, is defined as the expected return starting in state s and following action decided by π as described in 2.3.2. Formally, $V_\pi(s)$ is defined as:

$$V_\pi(s) = \mathbb{E}\left[G_t \mid S_t = s\right] = \mathbb{E}\left[\sum_{k=0}^{T} \gamma^k r_{t+k+1} \mid S_t = s\right] \tag{2.13}$$

Similarly, the Q-value defines a value to a state action pair under a given policy:

$$Q_\pi(s,a) = \mathbb{E}\left[G_t \mid S_t = s, A_t = a\right] = \mathbb{E}\left[\sum_{k=0}^{T} \gamma^k r_{t+k+1} \mid S_t = s, A_t = a\right] \tag{2.14}$$

These value functions present a way of comparing policies to one another and consequently can be used to define an optimal policy.

2.3.3 Optimal Policies

With the definition of policies and value functions, the meaning of an optimal policy can be defined. One says that a policy π is optimal if every other policy π' has a lower value in any given state. The optimal policy is often denoted by π^*. Formally this is summarized by

$$V_{\pi^*}(s) > V_\pi(s), \quad \forall s \in \mathcal{S} \tag{2.15}$$

Similarly, one can use an optimal policy to define the corresponding optimal Q-value and value function:

$$V^* = \max_{\pi} V_{\pi}(s) \tag{2.16}$$

$$Q^* = \max_{\pi} Q_{\pi}(s, a) \tag{2.17}$$

Knowing either the optimal policy or the optimal value function is equivalent. For example, with an optimal value function, one can act greedily with respect to the values to achieve the highest possible reward, resulting in an optimal policy. Similarly, the optimal value of a state s can be found by following an optimal policy, starting in s.

2.3.4 The Reinforcement Learning Problem

The different concepts described previously in this section define the problem RL is aiming to solve. In RL, the problem is defined as an MDP, where the goal is to find a solution in the form of an optimal policy π^*. Usually, this is done either by learning an optimal value or Q-function or directly by learning a parameterized policy.

What is shared between all RL methods, however, is the interaction loop between the agent and the environment depicted in figure 2.3. Most RL methods assume to know nothing about the environment and must therefore interact with it to learn. Often starting with random actions, the agent receives information about the resulting state of its actions and

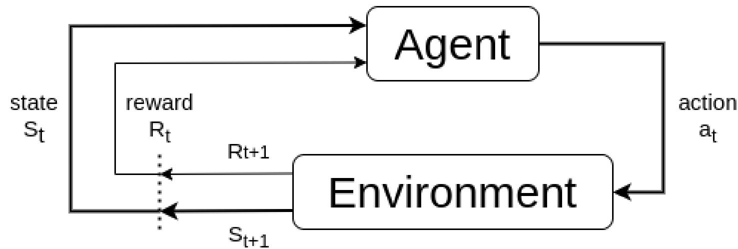


Figure 2.3: The interaction between an agent and the environment as depicted in [5]

receives a reward for said action. By learning from experience, the agent slowly understands how to act to get the highest reward possible.

While the dynamics of an RL problem, meaning the transition function, is usually given by the environment, as well as all possible states and actions. The reward function, however, must often be defined. Generally, an RL agent is trained to solve a specific task in a given environment. Therefore, the reward function must reflect the goal of the given task, and is in many ways what explicitly defines the agent's goal.

2.4 Model-Based Reinforcement Learning

Reinforcement learning methods can be roughly divided into two main categories; model-based RL (MBRL) and model-free RL.

Model-based methods either use a known model, ie. the transition and reward function of the MDP system, or they learn a model by sampling transitions in the environment. By utilizing this learnt model, model-based methods learn an optimal policy through some iterative process, often alternating between model and policy updates.

On the other hand, model-free methods ignore the model and aim to learn a policy directly, and the MDP system is viewed as a black box responding to actions. Model-free methods gather state action pairs by exploring the environment and use the gathered information to learn an optimal policy, either directly or through learning Q-values or a value function.

Since the focus of this thesis is on model-based RL, this section focuses on concepts and methods within MBRL. The first two parts present methods for model learning and utilization, and the last part presents details about the state of the art MBRL algorithm PETS.

2.4.1 Model Learning

The foundation of any model-based RL method includes some way of representing the model of the environment. In most cases, the model is unknown and must be learned from collected data. There are several ways of representing the dynamics model, and the choice of representation often depends on the problem's complexity.

Model Representation

If the model is known, the representation is often intuitive. For example, this can be in the form of a differential equation describing the system dynamics or a set of rules defining a board game. Therefore the focus is on learnable model representations.

In the simplest case, the environment consists of a finite set of states, making it possible to iteratively visit every state and try every action to see what happens. Furthermore, due to the finite state and action space, one can learn a tabular representation of the model and simply look up the resulting state of an action. However, this approach is not practical when the state space becomes too large. It would take an unreasonably long time to visit every state and action, leading to the necessity of more efficient model representations.

In general, it is necessary to learn a model representation able to infer

from data it has not seen before, especially in the continuous case where it is impossible to visit every state. The two most common options are Gaussian Processes (GP) and Neural Network models.

Gaussian processes are highly sample efficient and can learn accurate models with very few data points [29]. In addition, GPs can accurately predict model uncertainty which can be incorporated into planning. By utilizing the analytic nature of GPs methods like PILCO [14] achieves state of the art performance on low dimensional problems. Due to their high sample efficiency GPs has typically been the model of choice in many MBRL methods [14][30]. However, due to the so-called curse of dimensionality [31], both the sample efficiency and accuracy of GP models suffer in high dimensional and more complex problems. To overcome this dimensionality issue, more recent methods replace GP with expressive neural networks in order to solve more complex tasks.

Neural networks are, as discussed in section 2.1, in theory, universal function approximators which can efficiently handle the high number of dimensions where GP models fall short. Additionally, they are highly expressive and can therefore fit more complex dynamics than a GP model. However, they can easily suffer from overfitting, especially in a low data regime. Modelling methods like uncertainty aware ensembles are therefore often used as a method to maintain and reduce uncertainty and stabilize training [18]. Furthermore, uncertainty aware network models, such as PNNs, are also able to capture model uncertainty, which can be incorporated into planning to help learn faster [4].

Learning Dynamics

With a way of representing the dynamics model, there is still the choice of what this model represents. In some way, it must describe transitions by predicting the resulting state (and potentially reward) given a state-action pair as input.

The simplest and most intuitive way is for the model to output a state prediction directly, ie. $\hat{s}_{t+1} = f(s_t, a_t)$, where f is some learnt dynamics function. This function can be hard to learn when s_{t+1} and s_t are too similar, and the action taken seems to have little effect on the state, as discussed by Nagabandi et al. [32]. Although direct state predictions are sufficient in discrete cases like board games, they might be hard to learn when dealing with continuous state dynamics.

To solve this issue, the standard way of learning continuous state dynamics is instead to predict the change in state, ie. $\hat{s}_{t+1} = s_t + f(s_t, a_t)$. This method also makes the most intuitive sense in a continuous setting, as the underlying dynamics are, although unknown, usually described by some differential equation $\dot{s} = g(s, a)$. When learning to predict state change, the model resembles this type of differential equation more closely compared to direct state predictions.

2.4.2 Model Utilization

The utilization of a dynamics model is the key to MBRLs high sample efficiency compared to model-free methods. Whilst learning the model itself, up to model representation, is usually quite similar between methods, it is how one utilizes the model that separates MBRL methods. There are many methods out there utilizing the model in different ways. However, this section will focus on two separate groups; ones that use the model to learn a separate policy and methods that explicitly plans over the model to choose actions.

Learning Explicit Policies

Inspired by model-free methods, one can learn a parameterized policy or value function to decide actions. In model-free RL this is done by sampling the environment. However, a benefit of having a dynamics model available is that one can use the learnt dynamics to update the policy instead of sampling from the environment, thus increasing sample efficiency over model-free methods.

If the dynamics are known, one can learn a policy completely without sampling the environment, which is often the case in games like Go or chess. Methods like AlphaZero, for example, managed to beat the best human players in various games only through self-play, without any external interaction [8]. Unfortunately, although recent studies have been able to

apply the concept of self-play to practical tasks like video compression[33], it is not as directly applicable to control problems with uncertain dynamics.

In cases where one has to learn the dynamics as well as a policy, certain methods seek inspiration from model-free RL. One can utilize the learnt dynamics to sample trajectories to optimize the policy. These methods are often referred to as hybrid model-based/model-free [11] or Dyna-style algorithms[27]. These types of algorithms are able to improve upon the sample efficiency of state of the art model-free methods by combining model-based and model-free policy updates, as depicted in figure 2.4. This idea allows them to learn a policy by generating transition data using the learnt model, allowing the agent to train on "environment data" without sampling the environment.

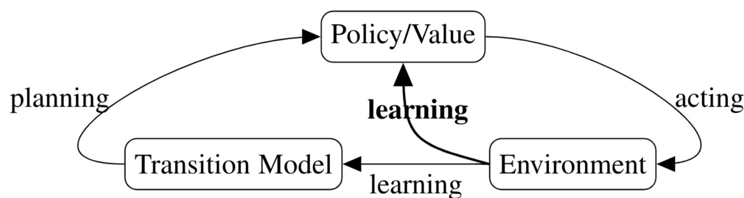


Figure 2.4: Hybrid model-based / model-free learning, as depicted in [11]

The original Dyna algorithm was introduced back in 1991 by Sutton [12]. However, with the increased use of accurate deep network models, methods like model-based policy optimization (MBPO)[15] achieves both high performance by utilizing the benefits of model-free policy updates and high sample efficiency by utilizing a learnt model.

Explicit Planning Over Dynamics Model

In contrast to methods like MBPO, which samples the model to train a policy, algorithms like PILCO [14], and PETS[4] focus on learning the model and explicitly using this to plan actions. Figure 2.5 shows the interaction between the model and environment in this type of algorithm; notice the missing arrow compared to figure 2.4. In methods like PILCO, one utilizes the analytical gradients of the GP model to optimize actions. However, with more complex tasks where GPs are not sufficient as a model, the most common method is using some version of MPC, which is discussed in detail in section 2.4.2. These MPC-based methods have become more popular and better performing with the use of expressive neural networks as dynamics models.

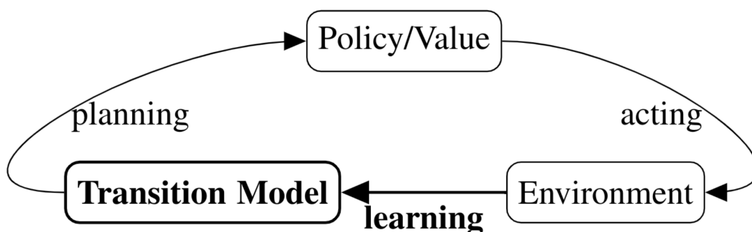


Figure 2.5: Explicit model planning, as depicted in [11]

Methods that utilize MPC are often referred to as shooting methods [27]. They solve the optimization problem by generating a set of action sequences which are then evaluated using the dynamics model and a reward function. Random shooting methods are the simplest form of this; however,

more sophisticated ways to optimize action sequences like CEM (discussed in section 2.2) often result in more optimal solutions and consequently better performance.

The state-of-the-art method PETS utilizes MPC with CEM in order to achieve performance comparable to the best model-free methods. However, methods using random shooting, like Model-Based Model-Free [32], have also shown good results in certain benchmark tasks[27].

MPC in RL

Model-predictive control (MPC) is a method for controlling dynamical systems through planning and optimization and is the go-to planning framework for the best planning based MBRL algoirthms[27]. By predicting system behaviour through a dynamics model, MPC solves an online optimization problem at every timestep in order to estimate the optimal control inputs. In order to do this, MPC needs an objective function to evaluate the quality of its inputs. In the context of classical control, the optimal actions are typically the ones that minimize the state deviation from a reference signal; however, when utilizing MPC in RL the objective is instead to maximize the expected return (equation 2.12).

Standard methods for MPC plans actions for a given horizon, which is the length of the planned trajectory in terms of time, and then execute the first input in the action sequence before re-planning again. This re-planning is crucial for MPCs performance, as the estimated dynamics model is usually

not perfectly accurate, especially in a MBRL setting where the dynamics are learnt through training.

In a classical control setting, the optimization problem is often solved with classic optimization methods involving the derivative of the objective function. However, in RL the cost function is represented by expected return (2.12) over a finite horizon, which does not necessarily have readily available derivatives. Therefore, most methods often rely on derivative-free optimization methods like CEM to solve the planning problem.

2.4.3 PETS - Probabilistic Ensembles with Trajectory Sampling

Chua et. al significantly reduced the performance gap between model-free and model-based methods with PETS [4]. Through ensembles of probabilistic neural networks and MPC, PETS is able to achieve performance comparable to the best model-free methods at convergence with significantly fewer samples. As a result, PETS is currently considered the state of the art method in MBRL, achieving the highest or close to highest performance on several benchmarking tasks [27].

The algorithm consists of two main parts; the ensemble model and trajectory sampling state propagation.

Probabilistic ensemble model

The model used in the PETS algorithm is an ensemble of PNNs parametrizing a Gaussian distribution, as described in section 2.1.2. By inducing a distribution over the state transitions rather than using deterministic prediction models, the planning agent is able to account for model uncertainty in the low data regime, improving the data efficiency of the learning [4].

Utilizing an ensemble of bootstrap models provides a simple and reasonable way to capture epistemic uncertainty. Furthermore, through the use of their proposed propagation method, PETS incorporates this model uncertainty into the planning, achieving high performance despite present modelling errors.

Trajectory Sampling

In addition to uncertainty aware dynamics models, Chua et al. [4] introduced a state propagation method called Trajectory Sampling (TS).

TS is a particle based sampling method. Initially it starts by creating P number of particles from a given initial state s_0. The p-th particle at time t is denoted by s_t^p, thus initially $s_0^p = s_0 \forall p$. Given some way of choosing action a_t, each particle is propagated through the dynamics model $\tilde{f}$, by sampling $s_{t+1}^p \sim \tilde{f}_b(s_t^p, a_t)$. Where $b \in \{1, \ldots, B\}$ denotes which model in the ensemble to use for propagation. Chua et al. [4] considered two

versions of TS, one where the the index b remains constant during a trial, essentially considering B different dynamics models as possible options. The other version uniformly re-samples the model index b for each time step, which effectively mimics a Bayesian model where dynamics are re-sampled every timestep.

Algorithm Overview

The PETS algorithm sequentially samples trajectories in the task environment, and trains the dynamics model on collected data. During planning PETS uses the CEM optimization presented in algorithm 2.2 to optimize actions in order to maximize the earned reward. The full planning scheme is an MPC framework incorporating the proposed propagation method TS to plan over the uncertainty aware dynamics model. The PETS algorithm is summarized in algorithm 2.3, as presented by Chua et al. [4].

Algorithm 2.3: PETS [4]

```
1  Initialize data D with a random controller for one trial;
2  for Trial k = 1 to K do
3      Train dynamics model f~ on given data D;
4      for Time t = 0 to TaskHorizon do
5          for Actions sampled a_{t:t+T} ~ CEM(·), 1 to N_samples do
6              Propagate state particles s_τ^p using TS and f~;
7              Evaluate actions as Σ_{τ=t}^{t+τ} (1/P) Σ_{p=1}^{P} r(s_τ^p, a_τ);
8              Update CEM(·) distribution;
9          end
10         Execute first action a_t^* from optimal actions a_{t:t+T}^*;
11         record outcome: D ← D ∪ {s_t, a_t^*, s_{t+1}};
12     end
13 end
```

2.5 Open AI gym

Open AI gym is a toolkit for developing and comparing reinforcement learning algorithms [34]. The tool kit consists of several tasks wrapped in an MDP like environment that gives rewards based on performed actions and states. These environments range from classic control tasks like balancing a pole on a cart to classic atari games and more complex physics-based tasks. Open AI gym is considered the standard framework for benchmarking RL algorithms and implementing task environments.

2.5.1 Gym Environments

Intending to standardize test environments for RL, gym environments all follow a specific structure. In short, each environment needs a method for stepping and resetting. The stepping method is what executes a given action in the MDP environment, resulting in the next state and potential reward received. The reset method is responsible for resetting the environment between trials. These two methods, apart from initialization, make up the basis of any gym environment.

Some of the environments supplied by OpenAI are simple tasks with simple dynamics. For example, these are tasks like cartpole or inverse pendulum. However, many tasks include complicated physics interactions and involve robots with multiple joints, simulating animals or humanoids. Therefore, many of the control environments rely on the physics engine MuJoCo[35].

2.5.2 MuJoCo

MuJoCo (Multi-Joint dynamics with Contact) is an advanced physics engine explicitly designed for model-based and automatic control[35]. Due to its optimized forward and backwards kinematics and easy-to-use model design, it is the physics backbone in most control-based gym environments.

MuJoCo utilizes its own modelling language, MJCF, which aims to be as accessible and intuitive as possible while simultaneously providing access to all of MuJoCo's computing abilities. This makes it possible to implement models fast and easy for testing, which is practical when testing RL methods in different scenarios before testing on physical robots, for example. As a consequence, many of the standard benchmarking tasks used in RL are MuJoCo-based physics and control tasks[27]. The default MuJoCo environments in openAI gym range from simple 2D pendulums to complex humanoid and animal-like robots.

2.6 MBRL-Lib

The mbrl framework is a toolbox for Model-Based Reinforcement Learning algorithms, based on PyTorch[36]. The library is highly modularized, allowing for interchangeability of the various subcomponents in a MBRL algorithm[2]. This section presents some of the key components found in the framework.

2.6.1 Replay Buffer

A practical utility found in the library is the replay buffer. An essential part of MBRL is to train a dynamics model on collected data. The replay buffer handles this data collection. Data collected in a MBRL method is structured as a tuple of state transitions, including current state, action, resulting state, reward, and whether the episode terminated or not. This replay buffer provides a way to handle and store all trajectory information in a structured manner to facilitate an easy-to-use data loader to train the dynamics model in a supervised way.

2.6.2 Dynamics Models

Mbrl-lib provides a base class for all models in the framework. However, of interest to this thesis is the probabilistic network ensemble.

The *GaussianMLPEnsemble* provides a PyTorch implementation of the probabilistic network ensembles presented by Chua et al. [4]. In addition to the model itself, the implementation also includes the ability to forward particles through the model by sampling according to the TS propagation used in PETS.

To further increase the ease-of-use principle, the library provides a set of wrappers around the model itself to help with training. To utlize the training data stored as a replay buffer, the *TransitionModel* wrapper provides functionality to convert transition data to input and output of the network model. This includes methods for combining state and actions into a single model input, as well as methods for generating target values for training the model in a supervised manor.

To complete the dynamics modelling, the *ModelTrainer* class provides a way to train a PyTorch model in a supervised way. The trainers *train* method implements a classical supervised training loop that updates the model for a set number of epochs with the provided dataset. In combination with the TransitionModel wrapper, it provides a way to train a dynamics model using transition data with only a few lines of code.

2.6.3 Configuration

When everything is implemented correctly, all parameters necessary to run an MBRL session are specified in a configuration file utilizing Hydra

configuration[37], an example of one such configuration file is found in listing 4.1. This type of configuration is made to quickly run simulations directly from a terminal without altering any code. Furthermore, the library is made to facilitate MBRL research, and the configuration system is therefore designed to facilitate easy testing of different parameters and tasks without the need to re-code the algorithm itself.

Chapter 3

Developing an RL algorithm

This chapter presents the development of the exploration-based MBRL method. It is divided into two sections. The first one discusses previous work and inspiration behind the algorithm developed, while the last section presents the algorithm itself. The pseudocode of the final algorithm is given at the end of the chapter in algorithm 3.1.

3.1 Previous Work and Inspirations

The intuition behind the algorithm developed in section 3.2 is that exploring uncertainty areas will lead to higher information gain and thus learning faster. This concept is known as targeted exploration and is a well-known idea in RL. Although the term uncertainty is not used, the concept of using uncertainty to direct exploration was presented in the early 1990's[38].

Although much research exists on exploration in model-free RL, model-based exploration has only started seeing research in recent years. Previous methods like "curiosity" [39] achieve exploration by rewarding the agent based on prediction errors compared to the real dynamics. Furthermore, counting based methods "count" the number of visits to each state, aiming to explore as many of the unseen states as possible[40]. However, all these methods have in common that they reward the agent for visiting states about which it has little information or is uncertain, aiming to maximize the information gained from each step. This is also the idea behind targeting exploration with uncertainty estimation.

The algorithm presented in section 3.2 is an extension of PETS [4] which incorporates uncertainty guided exploration into planning. Inspired by previous work in model-based exploration by Pathak et al. [41] and the field of active learning [42], the algorithm utilizes the probabilistic ensemble model proposed by Chua et al. [4] to estimate uncertainty in order to guide and balance exploration.

3.2 The Exploration Algorithm

PETS outperforms most other model-based methods with its probabilistic ensembles and MPC scheme. However, the full potential of the probabilistic ensemble model is not utilized. Inspired by the ideas mentioned in section 3.1, this section proposes a method for incorporating the estimated model uncertainty into planning, aiming to improve the agents' ability to explore the state space efficiently. The first few subsections explain details about each component of the algorithm before a summary, and the full algorithm is given in section 3.2.4.

3.2.1 Uncertainty-Based Exploration

The probabilistic ensemble dynamics model captures both aleatoric and epistemic uncertainty. We propose using the epistemic uncertainty measurement directly during planning to direct exploration towards parts of the state space where the model is uncertain.

Uncertainty Estimation

Given an ensemble dynamics model $\tilde{f}$ of B bootstrap models, and a model input consisting of a state action pair (s, a). The epistemic uncertainty of the model on the given data point can be estimated by the empirical variance between the prediction of each member model in the ensemble:

$$\sigma_s^2 = \frac{1}{B-1} \sum_{b=1}^{B} \left(\tilde{f}_b(s,a) - \overline{\tilde{f}(s,a)} \right)^2 \tag{3.1}$$

This estimate measures the disagreement between each model in the ensemble. If it is low, it indicates that the model is quite confident in its prediction, whilst if high, the prediction is uncertain. This estimate, therefore, gives a natural indication of whether to explore in the given state or not.

Planning

During planning, PETS evaluates action sequences by using the reward from the environment as an optimization function. The result is an agent actively exploiting the current model and potentially visiting the same states repeatedly to solve the task. However, shifting the focus toward better learning outcomes rather than earned rewards should result in a more accurate model.

During exploration, trajectories can be evaluated based on the uncertainty of the model's one step predictions. When exploring, the agent, therefore, receives a reward equal to the variance between each ensemble model (equation 3.4) instead of a reward from the task environment. In practice, this means changing the objective function used by the CEM optimizer from the expected return to the cumulative uncertainty estimate. Fur-

thermore, by visiting states with the highest model uncertainty, the agent aims to learn as much new information as possible from each step, thus reducing training time and overall model uncertainty.

3.2.2 Balancing Exploration and Exploitation

The exploration vs exploitation dilemma is always present in every RL problem. Typically model-free methods resort to some form of ε-greedy approach, where $\varepsilon \in [0, 1]$ corresponds to the probability of exploring instead of exploiting. In the model-based setting, there is more information available. Using this information, we propose a continuous and adaptive method inspired by the ε-greedy approach in model-free RL.

The principle behind ε-greedy methods is that an agent does less exploration and more exploitation as it learns. This reduced exploitation is usually achieved by reducing ε during training, either by an exponential approach or similar [5]. However, this approach does not consider that the agent might benefit from exploring states it does not reach until its performance is good enough. Instead, updating the ε parameter based on the model's uncertainty can achieve a more adaptive weighting between exploration and exploration. Since the model's uncertainty remains the same between training sessions, the uncertainty is estimated each time the model is updated, and ε is adjusted accordingly. To achieve this adaptivety, ε is updated as the normalized model uncertainty based on the highest uncertainty estimate seen during training:

$$\varepsilon = \sigma_m^2 / \sigma_{max}^2 \tag{3.2}$$

Here, σ_m represent the overall model uncertainty, which will be defined later in equation 3.4. Utilizing this update-law both ensures that ε follows the change in model uncertainty and is contained in the interval $[0, 1]$.

In addition to updating ε adaptively, the algorithm uses ε as a scaling factor between exploration and exploitation during action evaluation, rather than a probability of choosing one or the other. During training, the agent receives a reward both for exploration, given by equation 3.1, and for exploitation, which is given by the environment reward. These two rewards are then scaled by ε to give a single scalar reward:

$$r_{\text{agent}} = \sum_{t=1}^{T} \varepsilon \times \sigma_s^2(\boldsymbol{s}_t, \boldsymbol{a}_t) + (1 - \varepsilon) \times r(\boldsymbol{s}_t, \boldsymbol{a}_t) \tag{3.3}$$

Here σ_s is the one-step uncertainty from the current state (equation 3.1), and r is the actual reward from the environment's reward function. Providing the agent reward in this manner achieves two things. Firstly, for some given value of ε this reward enables the agent to exploit areas where its uncertainty is low rather than forcing the agent to explore areas where it is already quite confident. If $\sigma_s \simeq 0$, the agent will focus on exploiting the environment reward even though ε is high. Secondly, this approach still retains the idea of exploring less as the agent improves. The model's un-

certainty should naturally decrease as the agent trains. Therefore it should also tend towards preferring exploitation rewards instead of exploration as its performance increases.

3.2.3 Evaluating Model Uncertainty

In order to update ε as discussed in section 3.2.2, the need for an accurate uncertainty estimate of the model is necessary. The uncertainty estimation of the model described in equation 3.1 only considers one single data point. Thus, in order to get a more general uncertainty estimate, the model's uncertainty is instead evaluated as the mean over a whole set of data points denoted by $\mathcal{D}$. The adapted version of equation 3.1 then becomes:

$$\sigma_m^2 = \frac{1}{|\mathcal{D}|} \sum_{(s,a)\in\mathcal{D}} \frac{1}{B-1} \sum_{b=1}^{B} \left(\tilde{f}_b(s,a) - \overline{\tilde{f}(s,a)} \right)^2 \tag{3.4}$$

Using the same data as the agent collects during training would not necessarily yield an accurate estimate of the model uncertainty, as the model is trained to fit this data. Therefore, a separate dataset $\mathcal{D}_{unc}$ is sampled in order to be used for uncertainty evaluation.

In order to sample this external dataset, a separate policy π_u is used instead of the policy used for training. This gives the ability to sample data independently of the current model and the current policy, which results in

a more general estimate of the model uncertainty. However, the uncertainty estimate will always be biased towards the data used for evaluation. Thus, this dataset should not be static throughout the whole training. However, updating $\mathcal{D}_{unc}$ for every uncertainty estimate would result in multiple times the number of interactions with the environment, which reduces the sample efficiency drastically. So instead, $\mathcal{D}_{unc}$ is updated every few model updates with some given frequency of training episodes.

3.2.4 Algorithm Summary

Combining all the ideas discussed in this chapter with the ideas from PETS, the result is an exploration based PETS algorithm summarized as pseudocode in algorithm 3.1. Between each trial the dynamics model is trained on the collected dataset, similar to PETS. After each model update, a new uncertainty estimate of the model, σ_m^2 is evaluated on the $\mathcal{D}_{unc}$ dataset (equation 3.4), which itself is re-sampled at set frequency of trials. Based on this uncertainty a new value for the weighting ε is calculated using the update rule in equation 3.2. The planning part itself follows the idea from PETS; an MPC control scheme utilizing CEM optimization and the TS state propagation. However, trajectories are evaluated based on the weighting between exploration and exploitation in equation 3.3.

Notice that the trajectory evaluation in algorithm 3.1 contains a mapping $\phi : \mathbb{R} \mapsto \mathbb{R}$, which is not present in equation 3.3. This mapping is used to map the one step uncertainty σ_s^2 to a value comparable to the environment

reward and is potentially experiment specific. Specific details regarding this are discussed in the implementation chapter, section 4.1.

Algorithm 3.1: The Exploration Algorithm

---

1 Initialize dynamics model $\tilde{f}$, ε, π_u;
2 Populate dataset $\mathcal{D}$ using random controller for n initial trials.
3 **for** $k \leftarrow 1$ **to** *K Trials* **do**
4 Train dynamics model $\tilde{f}$ on $\mathcal{D}$;
5 Populate uncertainty data $\mathcal{D}_{unc}$ with π_u with some frequency;
6 $\sigma_m \leftarrow$ Evaluate model uncertainty on $\mathcal{D}_{unc}$ according to 3.4;
7 $\varepsilon \leftarrow$ Calculate new ε from σ_m according to 3.2;
8 **for** $t \leftarrow 1$ **to** *TaskHorizon* **do**
9 **for** *Actions sampled* $\boldsymbol{a}_{t:t+T} \sim CEM(\cdot)$*, 1* **to** *NIters* **do**
10 Propagate state particles $\boldsymbol{s}^p$ using TS and $\tilde{f}$;
11 Evaluate actions as $\sum_{\tau=t}^{t+T} \frac{1}{P} \sum_{p=1}^{P} (1-\varepsilon) r(\boldsymbol{s}_\tau^p, \boldsymbol{a}_\tau) + \varepsilon\phi(\sigma_s^2(s_\tau^p, \boldsymbol{a}_\tau))$;
12 Update CEM distribution;
13 **end**
14 Execute first action $\boldsymbol{a}_t^*$ from optimal action sequence $\boldsymbol{a}_{t:t+T}^*$;
15 Record outcome: $\mathcal{D} \leftarrow \mathcal{D} \cup (\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{s}_{t+1})$
16 **end**
17 **end**

Chapter 4

Implementation Details

This chapter presents the details regarding implementing algorithm 3.1 in the *mbrl-lib* framework by Pineda et al. [2]. The first section describes how the trajectory evaluation and exploration scaling are implemented. The following section discusses details regarding data sampling for model uncertainty estimation, followed by a schematic overview of the various algorithm components. The last section presents details of the task environment used to evaluate the algorithm.

4.1 Model Exploration Wrapper

A vital part of the algorithm proposed in section 3.2 is how trajectories are evaluated by the CEM optimizer during the planning. The *mbrl-lib* framework provides an excellent class for evaluating trajectories based on received rewards by wrapping the model in a gym-like interface. In order to prevent re-implementing a lot of the already existing functionality, the implementation inherits this evaluation by expanding the model wrapper to facilitate the exploration-based trajectory evaluation.

Stepping and Reward Calculation

Similar to gym environments, the wrapper contains a function called *step*, which is responsible for propagating a state and action using the dynamics model and returning the resulting state, reward and termination details. In the base wrapper in *mbrl-lib*, this reward is based on an external reward calculated using a provided reward function typically representing the reward function of the task environment, which is what is done in the PETS algorithm. However, in the exploration case, it is also necessary to incorporate uncertainty (equation 3.3). To facilitate this extra exploration reward, the *step* method was overwritten. The parent base wrapper is still responsible for calculating the environmental reward; however, in addition, the new *step* method calculates the disagreement between member models in the ensemble and scales the rewards according to equation 3.3. One

interesting consequence of this implementation is the ability to turn off exploration, which can be helpful during testing. This feature also proved to be useful in further implementations discussed in section 4.2.

In order to properly implement the reward in equation 3.3, the need for a way to handle ε arises. The extended model wrapper, therefore, includes this parameter as an internal variable in order to properly evaluate rewards from the *step* method. Consequently, this wrapper also needs methods for setting and updating this ε parameter.

Updating ε

In order to facilitate the adaptive ε update in equation 3.2, this wrapper implements a method to update ε based on given model input. This update is performed by evaluating the model uncertainty on the given state and action data using equation 3.4. The wrapper class itself then keeps track of the highest uncertainty seen so far, and evaluates the new ε as the normalized uncertainty estimate given in equation 3.2: $\varepsilon = \sigma_m^2/\sigma_{max}^2$.

Uncertainty and Reward Mapping

The exploration reward, or uncertainty estimate, used to evaluate trajectories grows exponentially with each propagation loop due to compiling model uncertainty with each step. Therefore, the final reward calculation

utilizes a mapping of this uncertainty to make it more comparable to the environmental rewards. This is the map denoted as ϕ in algorithm 3.1, and the default is implemented as the mapping $\phi(x) = \log(1 + x)$. However, this can be changed using the configurations discussed in section 4.4. A similar map is also implemented for the environment reward, which defaults to the identity map: $x \mapsto x$. However, this map was not used in any experiments and is therefore not included in algorithm 3.1, although the implementation remains for potential use in future work.

4.2 Sampling Data for Uncertainty Estimation

Two versions of the exploration algorithm were considered and implemented. The difference is the policy used to sample data for model uncertainty estimation. One version uses a random policy to sample data, while the other utilizes the learnt MPC policy without exploration.

Two sampling agents

The random sampling agent is straightforward to implement; in fact, *mbrl-lib* has a class implementing a random agent which samples the action space of the environment to choose actions. However, the implementation of the policy agent is a bit trickier. Although, the way the model wrapper

performs trajectory evaluations was implemented made it much more manageable. Since the policy is essentially the same as the one used for training, except it only utilizes the expected environment reward, it was possible to use the same agent as done during training. All that is necessary is to disable exploration in the model wrapper and enable it after sampling the uncertainty data.

Sampling Frequency

Re-sampling the uncertainty dataset every time the model uncertainty is evaluated would drastically reduce the sample efficiency. While not updating the dataset would make the estimate biased towards the static data. The uncertainty of the model is more biased toward data used for evaluation at the start of training when it can be very inaccurate, as opposed to later when it is more generalized. Therefore, the frequency with which this dataset is updated is reduced throughout training. To achieve this, the sampling frequency is reduced according to the model uncertainty by utilizing the model wrappers ε parameter as an estimate of uncertainty decrease. Practically, this frequency is given as the minimum number of training steps between each update and is updated by multiplying the current frequency with $\frac{1}{\varepsilon}$. However, in some cases, ε converges towards zero. Therefore, the frequency is bound from below at 10% of the total training steps to prevent it from becoming too low.

4.3 Schematic Implementation Overview

The complete algorithm was implemented as an algorithm in the *mbrl-lib* framework, utilizing various utilities presented in 2.6. The existing replay buffer and model trainer was used to help train the dynamics model, and their implementation of a CEM optimizer was used to optimize action selection through the evaluation described in section 4.1.

A simplified schematic of the interaction between the various components in the algorithm implementation is depicted in figure 4.1. For simplicity, the uncertainty data sampling is left out of this schematic.

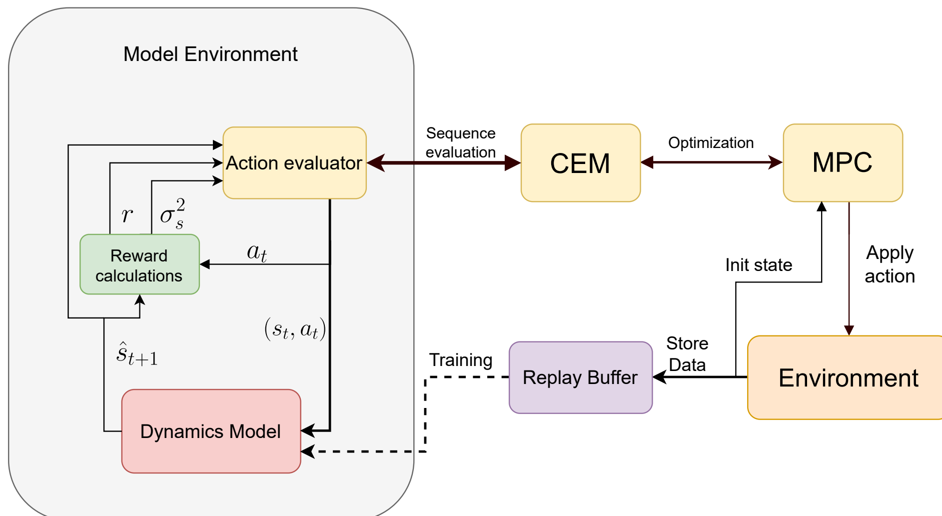


Figure 4.1: A high level schematic overview of the algorithm implementation. The stapled line of training indicates that it is not happening at every timestep. The two-way arrows indicate communications between MPC controller and model wrapper environment when optimizing action sequences.

4.4 Configurations

Similar to the other algorithms already present in the *mbrl-lib* framework, the implementation is compatible with the configuration options in the framework. This allows the algorithm to be tested in various environments with different configurations without needing to alter the code itself. All configurations needed to run an experiment are given in the configuration files. These files are written in YAML format and loaded using hydra [37]. An example of one such configuration file is shown in listing 4.1.

```yaml
algorithm:
  initial_exploration_steps: 1000
  target_is_delta: true
  normalize: true
  num_particles: 20

overrides:
  env: pets_halfcheetah
  num_steps: 200000
  trial_length: 1000
  agent_type: "pets"
  num_uncertainty_trajectories: 3
  random_uncertainty: true
  uncertainty_sample_freq: 1000
  adaptive_uncertainty_freq: true
  reward_map: None
  uncertainty_map: None

action_optimizer:
    #Omitted to save space
dynamics_model:
    #Omitted to save space
```

Listing 4.1: Example configuration file, highlighting some of the important configuration options in the algorithm implementation.

All of the experiment specific parameters are specified in the overrides section of the configuration. This includes which environment to use, trial length, the total number of training steps and similar configurations. Compared to the already present PETS configurations, the exploration algorithm requires a few more configuration inputs:

- **Agent type**: This specifies the type of agent to be used for training. The current implementation's only options are 'pets' and 'Exploration', which decide whether to utilize the targeted exploration (Exploration) in planning or not (pets).
- **Number of uncertainty trajectories**: This refers to the size of the dataset used to estimate model uncertainty and specifies how many trajectories to sample in order to populate the dataset. In our experiments, 3 was used for all tests.
- **Random uncertainty**: This specifies whether to sample the uncertainty data using a random agent or not. If false, it will use the exploiting policy agent; otherwise, a random agent will be used.
- **Adaptive uncertainty frequency** specifies whether to change the frequency of how often the uncertainty estimation data is re-sampled based on model uncertainty. If not specified, this defaults to True. If False, the frequency remains constant throughout training.
- **Uncertainty sample frequency** This specifies the frequency at which the uncertainty data is updated; if adaptive uncertainty is True, this sets the initial frequency. If unspecified, it defaults to the trajectory length.

Additionally, there is the configuration option of the reward and uncertainty maps discussed in section 4.1. These specify which functions to use for mapping exploration and exploitation rewards during planning. As mentioned in 4.1, the default is the identity map and $\log(1 + x)$, which is what will be used if these are not specified. To specify any other maps, the corresponding functions must be implemented in the math utility module of the *mbrl-lib* framework.

Any additional configurations regarding the action optimizer and dynamics model is already present in the framework. The full implementation is available at `https://github.com/JayKays/Robotic-mbrl`.

4.5 Task Environments

The algorithm was tested on four different gym environments; cartpole, half-cheetah,7-DOF reacher and 2D-walker. Each having different dimensions, rewards and complexity. This section present details about complexity and reward for each task environment.

4.5.1 Cartpole

The cartpole environment is the simplest of the four environments. The goal in cartpole is to balance a pole on a cart by moving the cart back and forth (see figure 4.2). The state space consists of position and velocity of the cart as well as the angle and angular velocity of the pole joint. The action space is a continuous scalar value between -1 and 1, representing the control input to the carts velocity.

An episode in the cartpole environments is terminated if the pole angle or cart position becomes too large, or the time horizon is reached. And the reward is simply +1 for every time step that does not end in termination.

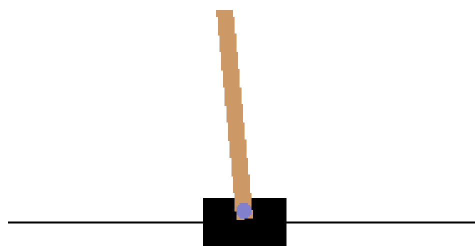


Figure 4.2: Screenshot of the cartpole environment

4.5.2 Halfcheetah

In the half-cheetah environment the goal is to learn how to control a cheetah-like robot (figure 4.3) to run as fast as possible. The cheetah has 6 controllable rotational joints, whose applied torques make up the action space. The state space contains positions and angular velocities of the joints as well as Cartesian position and velocity of the cheetah, adding up to a total of 18 states.

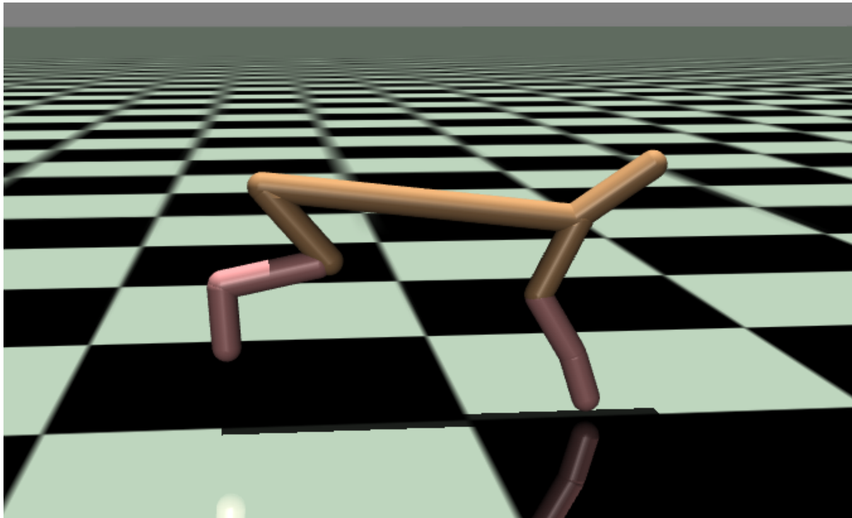


Figure 4.3: Screenshot of the halfcheetah environment

In contrast to the cartpole environment, the halfcheetah environment has no termination other than the episode horizon. The reward is based on the cartesian velocity of the cheetah, in addition to a penalty for using to large actions.

4.5.3 2D Walker

The walker environment is similar to the halfcheetah in that the goal is to run (or walk) as fast as possible. However, the controllable robot here represents a two-dimentional humanoid. The walker has 6 controllable joints which makes up the action space. The same joints' position and velocity is also part of the state space. In addition, the state space contains the vertical position and angle of the walkers head, as well as vertical, horizontal and angular velocities of the head. Adding up to 17 total states.

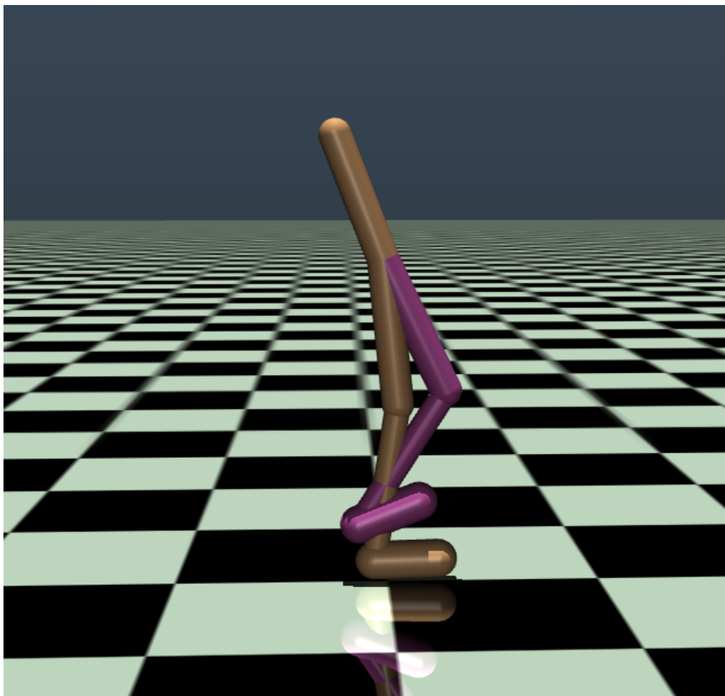


Figure 4.4: Screenshot of the 2D walker environment.

In contrast to the cheetah, the goal of the walker is to walk both fast and upright, therefore there is a termination based on how high above the ground the head is and the magnitude of the head angle. The walker gets a rewards based on horizontal velocity and a penalty for large actions. In addition it gets an "is alive" reward of +1 for every timestep that does not end in termination, similar to cartpole.

4.5.4 7-DOF Reacher

The reacher evironment is the same as the one used in the original pets paper [4]. It consists of a 3D 7-DOF robot arm, where the goal is to reach a desired end effector position. Similar to the other evironments, the actions space consists of torques and forces applied to the robots 7 joints, and the state space contains the position and velocity of these joints. In this task the state space also contains the Cartesian position of the target position of the end effector, resulting in 17 total states.

The reacher task has no termination, but the reward given here is purely negative. Similar to the walker and cheetah, the reacher gets a penalty for large actions. In addition it also gets a penalty equal to the distance between the end effector and the target for each time step.

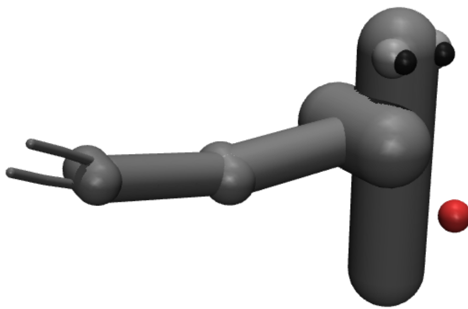


Figure 4.5: Screenshot of the reacher environment. The red ball corresponds to the desired target position of the end effector

4.5.5 Summary

A summary of the environment details is given in table 4.1 and 4.2. Firstly, table 4.1 describes state and action dimensions of each environment as well as task horizon, while table 4.2 gives mathematical descriptions of each environment's reward and potential termination function.

Table 4.1: Space dimensions and task horizon for each gym environment

Environment	State Dim	Action Dim	Task Horizon
Cartpole	4	1	200
Half-cheetah	18	6	1000
Walker	17	6	1000
Reacher	17	7	150

Table 4.2: Reward and termination functions for each gym environment. x_t and z_t corresponds to horizontal and vertical Cartesian position respectively. For the reacher; g and ee_t are the goal and end effector positions.

Environment	Reward	Termination when
Cartpole	1	$\lvert x_t\rvert > 2.4 \vee \lvert\theta\rvert > 12°$
Halfcheetah	$\dot{x}_t - 0.01\lvert\lvert \boldsymbol{a}_t \rvert\rvert^2$	None
Walker	$\dot{x}_t - 0.01\lvert\lvert \boldsymbol{a}_t \rvert\rvert^2 + 1$	$z_t \notin [0.2, 0.8] \vee \lvert\theta\rvert > 1$
Reacher	$-\lvert\lvert \mathrm{g} - \mathrm{ee}_t \rvert\rvert^2 - 0.01\lvert\lvert \boldsymbol{a}_t \rvert\rvert^2$	None

Chapter 5

Experimental Results

This chapter presents the experimental results from testing the algorithm on the four different benchmark tasks described in section 4.5. Performance is evaluated by using the PETS algorithm as a baseline. Two different versions of the exploration algorithm are considered, one uses a random controller to sample data for uncertainty measurements, while the other exploits the current model using an MPC policy similar to PETS.

The chapter is divided into five sections. The first four sections present test results from each of the four test environments described in section 4.5. The last section presents test results regarding the transferability of the trained agents to solve different tasks with the same dynamics. For visual clarity, the training curves in this chapter are smoothed using a SciPy [43] Gaussian filter.

5.1 2D Walker

This section presents the results of the algorithm evaluated on the 2D walker environment. Figure 5.1 shows training curves of PETS and the two versions of the exploration algorithm, and figure 5.2 shows the value of ε for the two versions of the exploration algorithm.

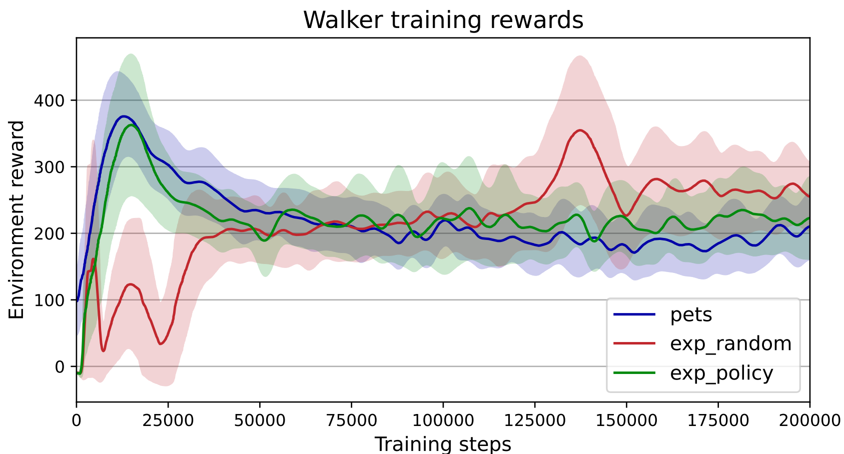


Figure 5.1: Algorithm performance on the 2D walker environment. The plot shows mean of three different test with different random seeds. Shaded area corresponds to the standard deviation between each seed.

Initially, PETS achieves higher rewards than both versions of the exploration algorithm until about 75'000 training steps, where both exploration agents achieve higher rewards than PETS. Figure 5.2 shows how the parameter ε changes through training for the two exploration agents. Using the learnt policy to sample data for uncertainty estimation results in ε

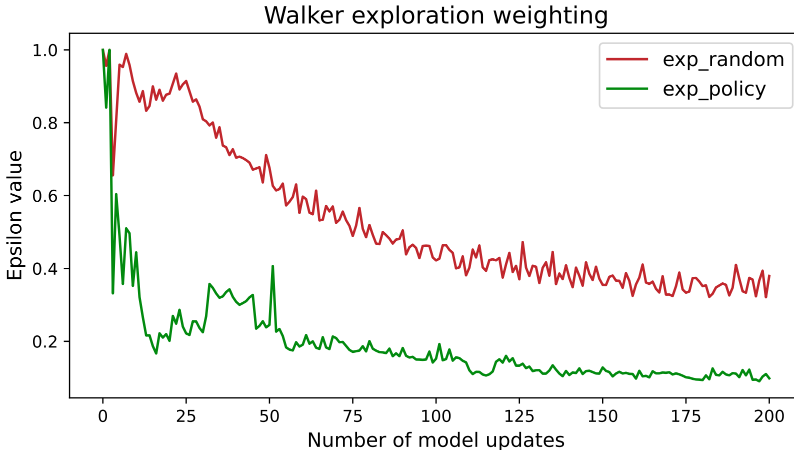


Figure 5.2: Development of the weighting parameter ε through training on the walker environment. Random and policy refers π_u being either random or following the exploitation policy of the current model.

dropping to about 0.2 early in training, while the random sampling agent still has an ε around 0.9. This difference in exploration weighting is reflected in figure 5.1, where the random uncertainty agent receives a much lower reward than the other two during the first 50'000 steps. This significant difference in exploration weighting is reduced towards the end of the training. However, the random sampling agent still has a significantly higher ε than the policy sampling agent while performing better in terms of received reward. Comparing the two exploration agents to the PETS agent, the policy sampling exploration agent closely follows the curve of PETS. The random sampling agent utilizes more exploration and does not experience the same reward peak at the begging as the two other methods, maintaining increasing performance until the end of training.

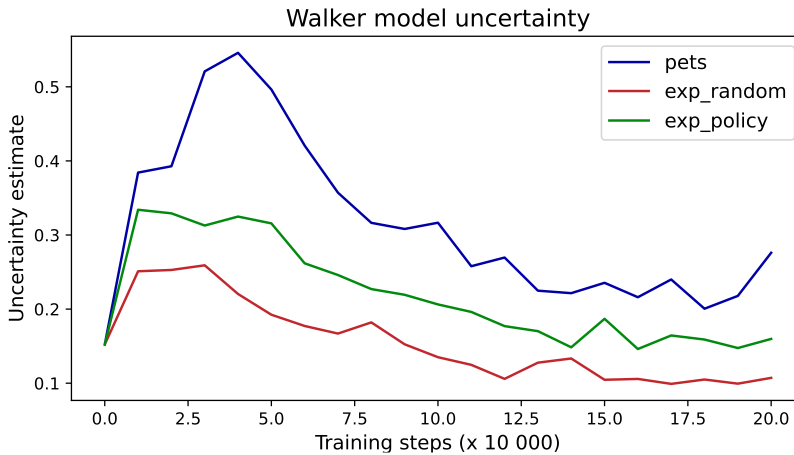


Figure 5.3: Development of model uncertainty during training. Uncertainty is evaluated on random state data using model checkpoints every 10 000 training steps.

Figure 5.3 shows the estimated uncertainty of the models during training for the three different algorithms. Similar to the performance seen in the training curve (figure 5.1), the model uncertainty of the two exploration agents is noticeably lower than that of PETS, with the random sampling agent achieving the lowest uncertainty of the two.

Figure 5.4 shows the average performance of each agent after 50, 100, 150 and 200k training steps without exploring. The two exploration agents achieve generally higher rewards than PETS in all checkpoints, with the exploration agent achieving the highest performance after the training. Unlike figure 5.1, PETS shows lower performance early in training compared to the two exploration agents.

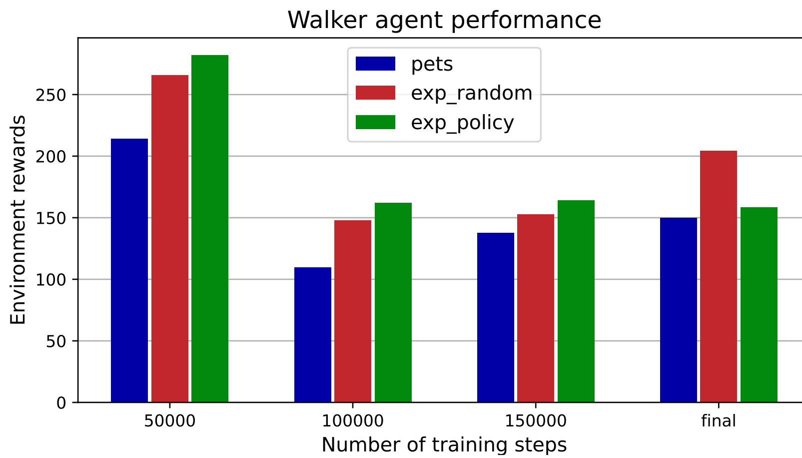


Figure 5.4: Mean agent performance after 50, 100, 150 and 200k training steps on the 2D walker task. The performance is evaluated without utilizing exploration, similar to policy used in PETS

5.2 Halfcheetah

The performance of the three agents during training in the halfcheetah environment is shown in figure 5.5. Contrary to the walker results, the three agents show much similar training curves when solving this task.

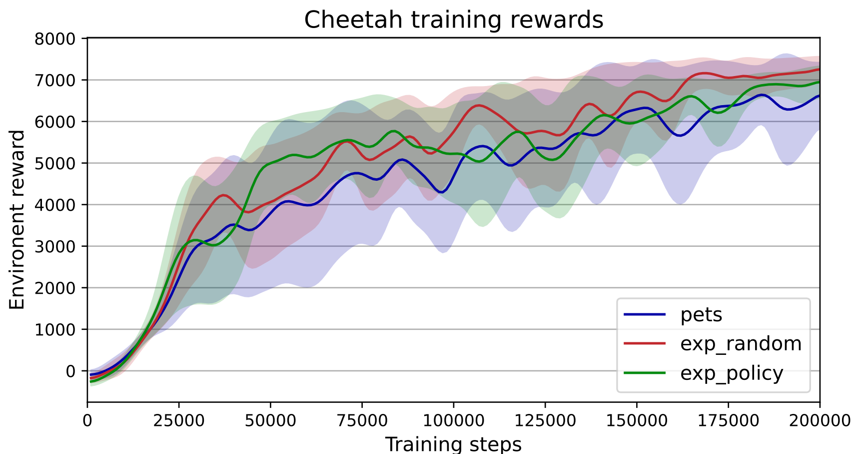


Figure 5.5: Algorithm performance on the halfcheetah environment. The plot shows mean of three different test with different random seeds. Shaded area corresponds to the standard deviation between each seed.

The two exploration agents achieve higher rewards during the beginning of training compared to PETS. While PETS catches up to the policy sampling agent, the random sampling exploration agent generally achieves better performance than PETS during the whole training session. In general, the two exploration agents show more consistent performance between each separate seed than PETS which shows an overall larger standard deviation

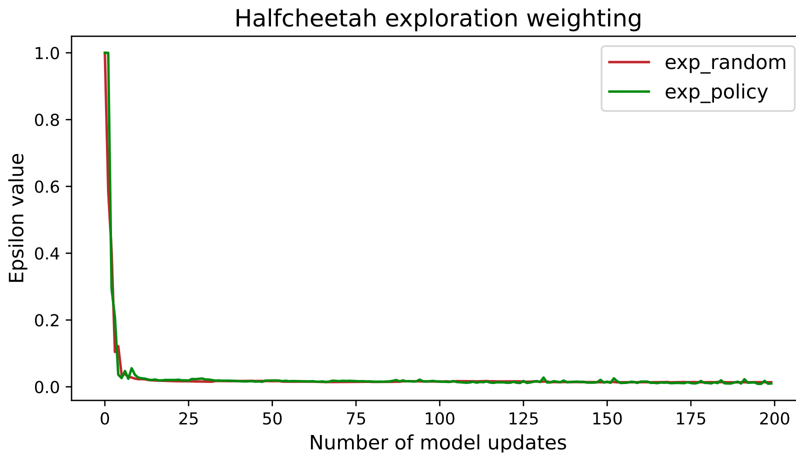


Figure 5.6: Development of the weighting parameter ε through training on the halfcheetah environment. Random and policy refers π_u being either random or following the exploitation policy of the current model.

between trials. Figure 5.6 shows mean value of ε for each of the exploration agents. The large difference in the walker task (figure 5.2) is not present here, and the value of ε converges toward zero rather early in training. Similar results are seen in the uncertainty of the models in figure 5.5. The uncertainty converges rather fast with all three methods. However, the model uncertainty with both exploration agents is consistently lower than that of PETS, with the random sampling exploration being slightly lower than the one using policy sampling.

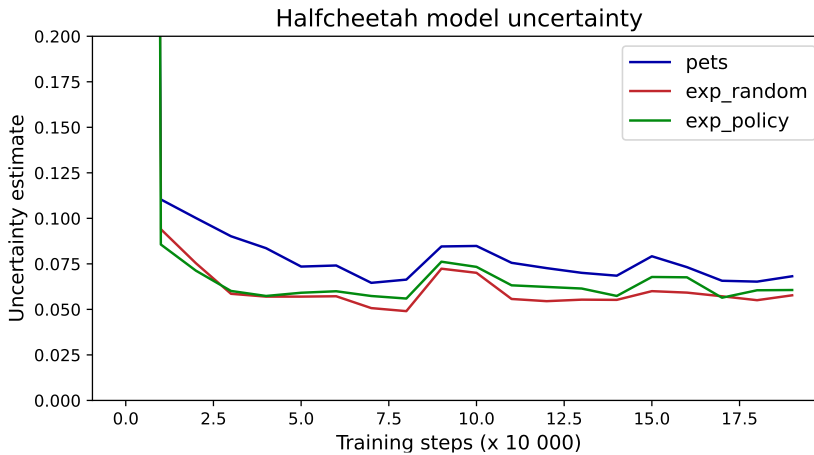


Figure 5.7: Development of model uncertainty during training. Uncertainty is evaluated on random state data using model checkpoints every 10 000 training steps. The first uncertainty estimate is removed to better visualize the later development.

The performance of the three agents after 50, 100, 150 and 200 trials is shown in figure 5.8. This evaluation was performed without the exploration enabled, only focusing on maximizing environment reward. After 200k training steps, the performance of the three agents is almost identical, while the two exploration agents are able to achieve higher rewards earlier in training than PETS.

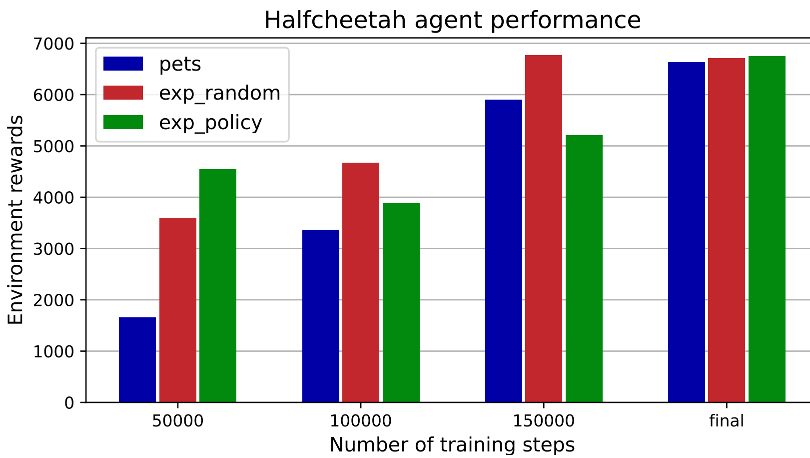


Figure 5.8: Mean agent performance after 50, 100, 150 and 200k training steps on the halfcheetah task. The performance is evaluated without utilizing exploration, similar to policy used in PETS

5.3 Reacher

The training results on the reacher environment are shown in figure 5.9. Initially, both version of the exploration agent achieves much lower rewards than PETS. However, after about 2000-3000 steps, both exploration agents achieve similar rewards as PETS, and all three agents converge to the same reward after about 4000 training steps. For visual clarity, the exact initial rewards of the exploration agents are not shown in the plot.

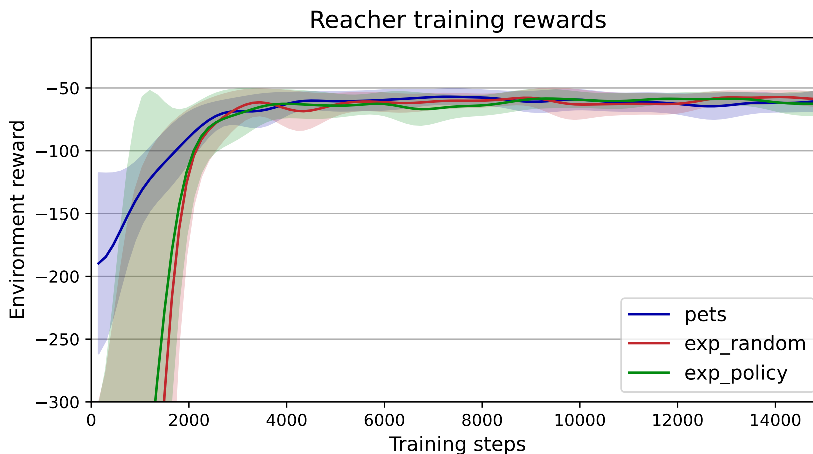


Figure 5.9: Algorithm performance on the 3D reacher environment. The plot shows the mean of three different tests with different random seeds. The shaded area corresponds to the standard deviation between each seed. The initial reward for the two exploration agents is about -900 and -600 for random and policy, respectively.

Figure 5.10 shows the development of the ε parameter throughout the training of the two versions of the exploration algorithm. Similar to the

walker results (figure 5.2), there is a significant gap between the agent evaluating uncertainty on random data compared to the one sampling using the exploitation policy. The agent sampling with the exploitation policy shows similar development as in other environments, where ε converges towards zero. However, the version using random data reaches its minimum value of $\varepsilon = 0.2$ after about 20 trials before slowly increasing to around 0.4-0.5 throughout the rest of the training.

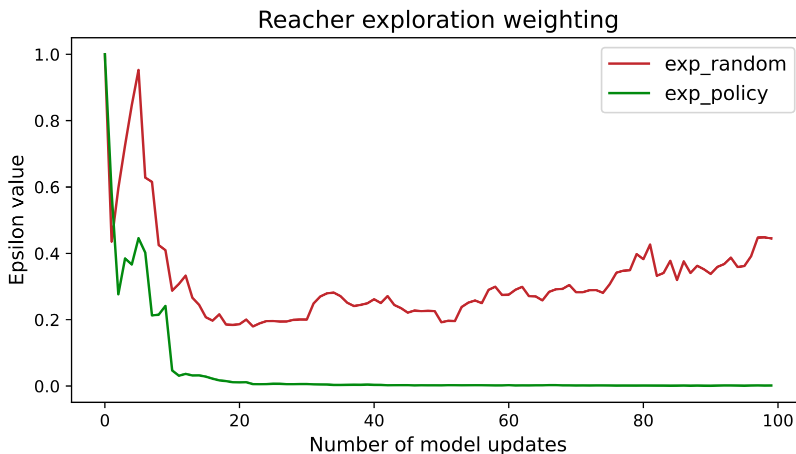


Figure 5.10: Development of the weighting parameter ε through training on the 3D reacher environment. Random and policy refers π_u being either random or following the exploitation policy of the current model.

The model uncertainty of the three agents is shown in figure 5.11. The uncertainty of all three models is quite similar, following the same trend throughout training. However, the model uncertainty of the PETS agent shows large oscillations toward the end of the training. In contrast, the two exploration agents show much smaller and more stable oscillations.

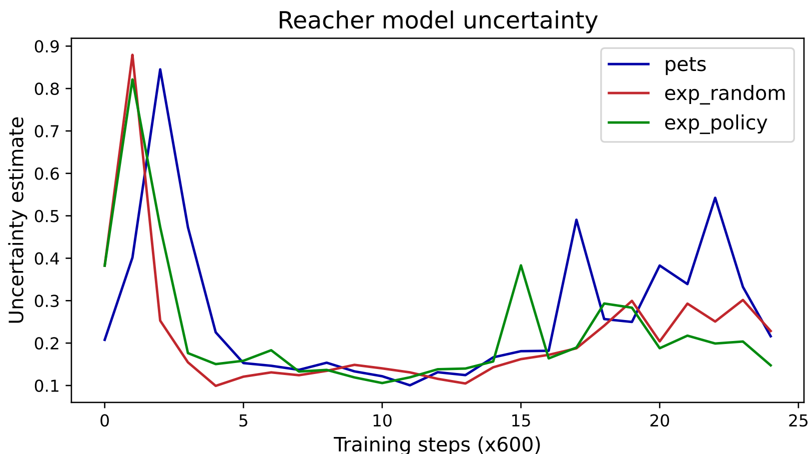


Figure 5.11: Development of model uncertainty during training on the reacher environment. Uncertainty is evaluated on random state data using model checkpoints every 600 training steps.

However, all three agents achieve the lowest model uncertainty about halfway through training, which increases towards the end of the training.

The performance of each agent was evaluated after 1800, 4200, 9000, and 15000 training steps without exploration. The average rewards are shown in figure 5.12. The two exploration agents perform better than PETS in the first checkpoint at 1800 steps. However, after 4200 training steps and later, performance is not changing much, with PETS showing generally better performance than the two exploration agents.

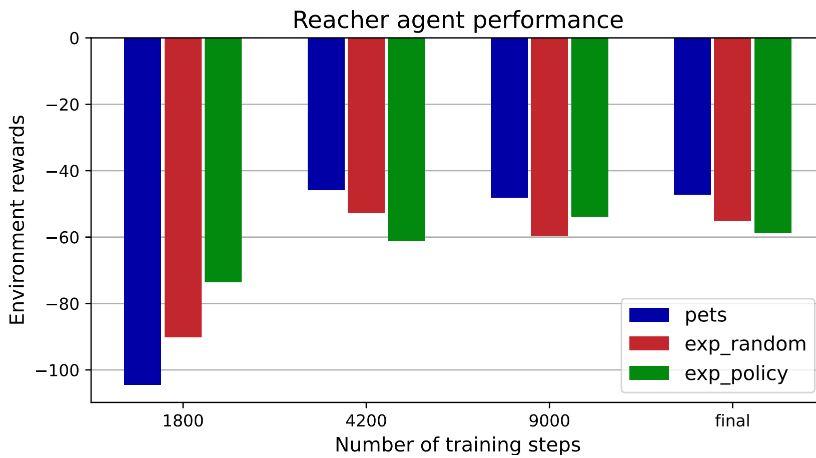


Figure 5.12: Mean agent performance after 1800, 4200, 9000 and 15000 training steps on the 7-DOF reacher task. The performance is evaluated without utilizing exploration, similar to policy used in PETS

5.4 Cartpole

Cartpole is the least complex environment used for testing. The training curves in figure 5.13 show that both exploration agents and PETS can completely solve the tasks of balancing the pole for the entire task horizon in under 1000 steps earning the maximum reward of 200.

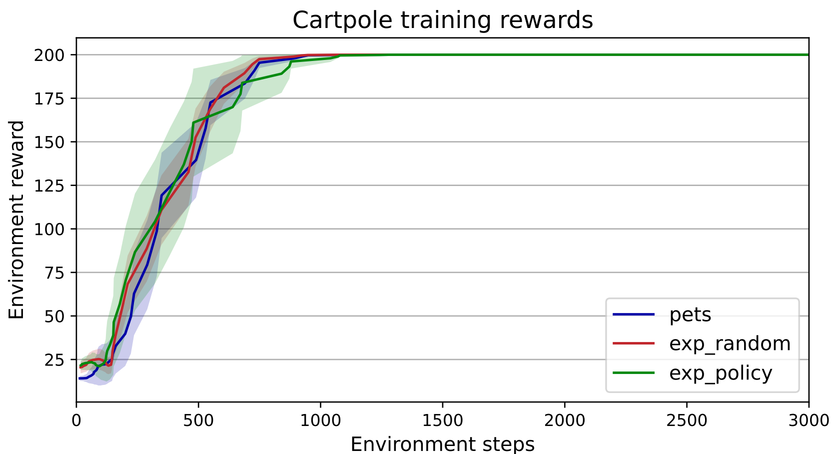


Figure 5.13: Algorithm performance on the cartpole environment. The plot shows mean of three different test with different random seeds. Shaded area corresponds to the standard deviation between each seed.

Similar to halfcheetah (figure 5.6) the value of ε for the two exploration agents on cartpole converges towards zero after only the first few trials as seen in figure 5.14. The model uncertainty shown in figure 5.15 shows a similar fast converging development for all three agents.

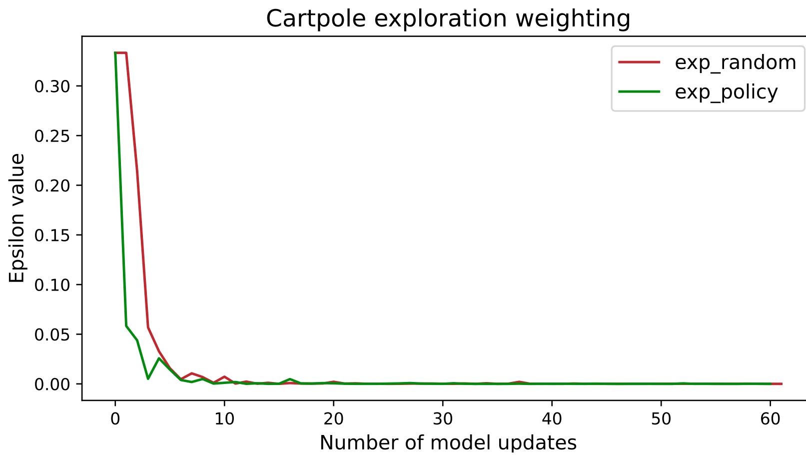


Figure 5.14: Development of the weighting parameter ε through training on the cartpole environment. Random and policy refers π_u being either random or following the exploitation policy of the current model.

Figure 5.16 shows the agents' performance after training, which reflects the results seen in figure 5.13. Each of the three agents can consistently achieve the maximum reward of 200.

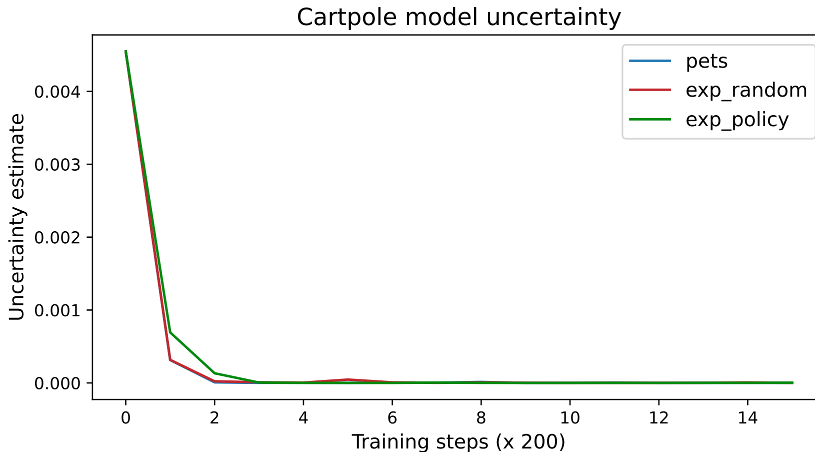


Figure 5.15: Development of model uncertainty throughout training on cartpole environment.

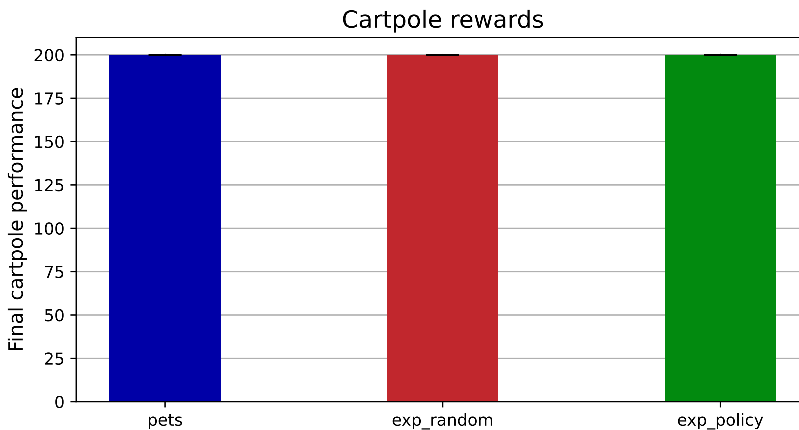


Figure 5.16: Mean agent performance after 3000 training steps. The performance is evaluated without exploration, similar to the policy used in PETS

5.5 Model Transferability

One major benefit to model-based methods over model-free methods is the ability to transfer the learnt dynamics to different tasks. In order to test this, the trained agents were tested in the same environment it was trained in by using a different reward function, thus altering the task to solve. Two environments were tested; halfcheetah and walker. The comparison is done between PETS and the random sampling exploration agent, as these generally showed the largest performance difference in results from section 5.1 - 5.4.

For the halfcheetah environment, the agents was set to follow a desired velocity v_d opposed to running as fast as possible, which is the task they were trained to do. The reward used to evaluate the performance was formulated as:

$$r(s,a) = -0.01||\boldsymbol{a}_t||^2 - 10\exp(-0.3||\dot{x}_t - v_d||_2^2) \tag{5.1}$$

The maximum velocity achieved by all agents was around $9 - 10m/s$ when running as fast as possible. Therefore, the agents were tested by tracking velocities of 2, 4 and 8 m/s. The resulting rewards are shown in figure 5.17.

When tracking a desired velocity of 2 and 4, both agents achieve similar performance, with the exploration agent performing slightly better than PETS. However, when tracking a desired velocity of 8, the exploration

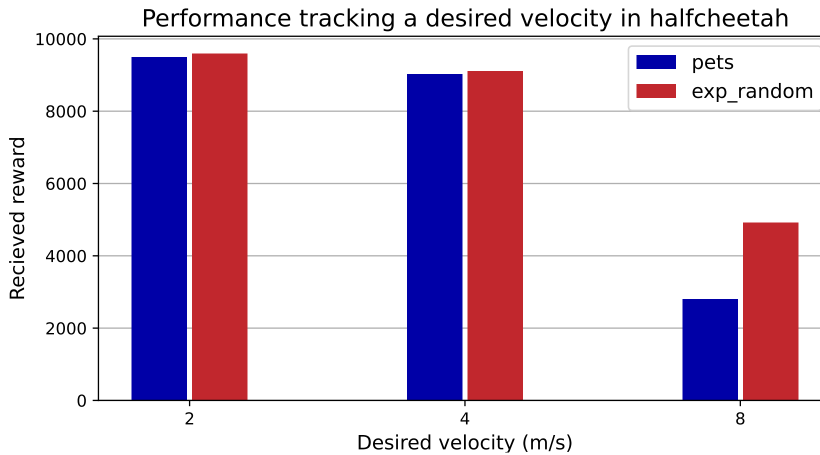


Figure 5.17: Mean performance of three trials on the halfcheetah environment, where agents aim to track a desired velocity opposed to running as fast as possible.

agent achieves significantly higher reward than PETS.

A similar test was done in the walker environment. However, instead of tracking a desired velocity, the task was set to walking backwards. To achieve this, the reward function in table 4.2 was altered by changing the sign of $\dot{x}_t$. The results in figure 5.18 show that the exploration agent is able to achieve noticeably higher reward than the PETS agent when solving this new task.

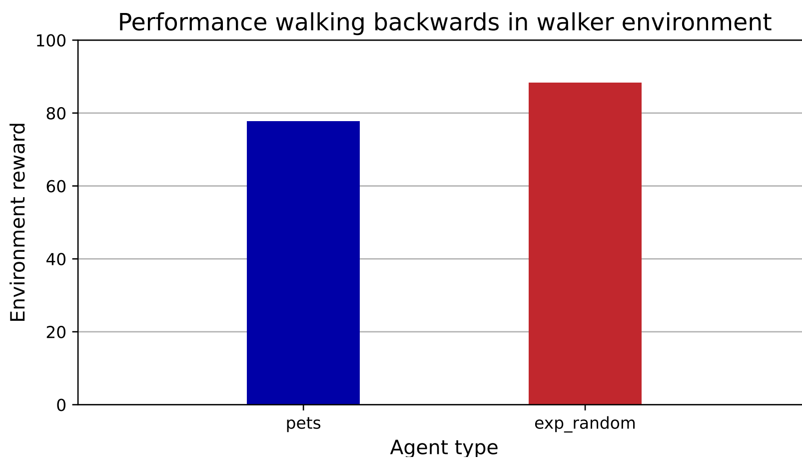


Figure 5.18: Mean performance of 10 trials in the walker environment, where agents aim to walk backwards instead of forwards.

Chapter 6

Discussion

In this chapter, the findings from chapter 5 are discussed. Firstly, the effects of exploration in the different task environments is discussed with respect to performance and model quality. Subsequently, the method of evaluating uncertainty and scaling exploration during training is reviewed. Lastly, the trained agents' transferability to other tasks within the same environment is discussed.

6.1 Performance Effects of Exploration

Overall the exploration algorithm shows promising results, improving the sample efficiency of PETS in most benchmark tasks. Additionally, the incorporation of exploration show reduced model uncertainty across most tasks.

6.1.1 Low Exploration Utilization

The results suggest that the halfcheetah and cartpole environments are the two benchmark tasks with the easiest dynamics to learn. Development of the exploration weighting (figure 5.6 and 5.14) shows that in both these cases, the amount of exploration compared to exploitation drastically favours exploitation only after a few trials of training as ε tend towards 0.

In cartpole, both exploration agents, as well as PETS, manage to solve the task after about 1000 steps, consistently earning the maximum reward of 200. Although there are slight variations in model uncertainty between each method, results suggest that the cartpole task is solved too quickly to see any significant effects of exploration.

The halfcheetah task shows similar development to cartpole in terms of ε (figure 5.6); the amount of exploration is significantly reduced after only the first few trials. This similarity alone might suggest reduced effects of the exploration, similar to cartpole. However, the training curves in figure

5.5 and midway evaluations in figure 5.8 suggest otherwise. Training curves show that the random sampling exploration agent, in general, performs better than PETS during the whole training, except at the very beginning. Apart from a few slight dips in reward, the policy sampling exploration agent performs similarly. Evaluations of the performance without exploration tell a similar story. Both exploration agents shows higher performance than PETS in almost all checkpoint evaluations in figure 5.8. However, this performance gap decreases throughout training ending in almost identical performance of the final agents. The large performance increase early in training suggests that even though ε heavily favours exploitation after the first few trials, the exploration in the first few trials and the small amount of exploration reward received throughout training is enough to increase sample efficiency. However, given enough training time PETS will achieve similar performance when $\varepsilon << 1$, as seen in figure 5.8. These results imply that the initial exploration is effective in increasing sample efficiency, but without continued use of exploration, PETS will eventually catch up.

In summary, the development of ε shows similar behaviour in both halfcheetah and cartpole; however, performance in terms of rewards differs. The simplicity and reward cap of cartpole allows all agents to solve the task rather quickly, resulting in little effect from the exploration. While in halfcheetah, the exploration shows clear improvements in terms of sample efficiency despite the fast convergence of exploration weighting. However, without continued exploration, the performance improvement dissipates given enough training.

6.1.2 Higher Exploration Utilization

Compared to the cartpole and halfcheetah environments, the exploration weighting in figure 5.2 and 5.10 suggests that these dynamics are harder to learn, utilizing much more exploration than seen in cartpole and halfcheetah. Although the amount of exploration used in walker and reacher environments show similarities, the performance in these two environments shows opposite effects of exploration.

The walker task shows a clear performance difference between the exploration algorithm and PETS. Focusing on the version using random data to estimate uncertainty, figure 5.1 shows that the exploration agent achieves a significantly lower reward than PETS early in training. The training curve correlates well with the plot showing the weighting factor ε in figure 5.2, which shows high values in early training that slowly decrease towards the end. Seeing as the goal of exploration is not to achieve high rewards but to learn more about the state space, this behaviour is expected. However, later parts of training where the amount of exploration is reduced show a clear benefit of the exploration, where the exploring agent achieves noticeably better performance towards the end than PETS.

Interestingly, the reward peak in the early parts of training is not present in the training curve of the exploration agent but is present in the first checkpoint evaluation in figure 5.4. Although receiving lower external rewards during training, the exploration agent can both learn and outperform PETS in a similar amount of training steps, suggesting a clear benefit

in terms of sample efficiency by introducing exploration. The exploration agent estimating uncertainty using policy data shows similar results. The training curve more closely follows the curve of PETS, likely due to the lower amount of exploration. However, there is a similar trend in both exploration agents in the checkpoint evaluations without exploration in figure 5.4. Both agents outperform PETS in all model checkpoints, although the random sampling agent shows a significantly higher reward in the final checkpoint. These results further strengthen what is seen in the halfcheetah task, that without continued exploration PETS will catch up in performance given enough time. The reduced exploration with the policy sampling agent results in a similar performance to PETS with $\varepsilon \simeq 0$, while the more exploring random sampling agent achieves significantly higher rewards in the end.

However, in the reacher task, this correlation between exploration and increased performance is not present. Training rewards in figure 5.9 show low reward during the early trials as a consequence of exploration, which is similar to results seen on the walker task. However, all three methods converge to the same performance after the first 20% of total training steps, with almost no distinction between them for the rest of the training. Comparing this to the exploration weighting in figure 5.10 does not show the same correlation as with the walker results. Although the weighting on the random sampling exploration is much higher and increases toward the end of the training, the performance in terms of reward is almost identical compared to the policy sampling agent. The midway evaluations of each agent show that after the first checkpoint PETS achieves better

performance than the two exploration agents. Suggesting that in this task, the introduction of exploration reduces the agent's performance rather than improves it. However, the performance during training in figure 5.9 is very similar between all three agents. Thus, using exploration seems to help maintain a similar performance to PETS than without exploration.

One thing to note about the reacher environment compared to the three others is that the goal of the task changes. For instance, in the walker task, the goal is constant; walk as far right as possible. However, in the reacher task, the goal position of the end effector is random for every trial. This type of randomness might naturally induce a form of exploration by simply trying to solve the task, which is not the case in other environments. This natural exploration in the reacher task indicates a potential reason for the contradicting results, where exploration improves performance in both walker and halfcheetah, but is reduced in reacher. Another cause of the performance difference might be a result of the model overfitting on task data and thus underfitting on general state data. However, this is further discussed in section 6.1.3.

Lastly, the walker results show a spike in performance early in training from all three agents (figure 5.4). This spike represents the highest reward achieved by all agents during training and is not reached again during the 200k training steps. This type of behaviour indicates that all agents might be stuck in some form of local minimum. Although the exploring agents achieve better performance than PETS, the exploration itself is not enough to push the agent out of this minimum in the given training time.

Previous work shows that PETS is able to achieve higher rewards in a similar environment by increasing the training time to 1 million steps[27]. Thus, more testing might be needed to see the full effects of introducing exploration in this particular task.

In summary, although the development of ε shows similar results in both walker and reacher, the performance in the walker environment is increased with exploration, while it decreases in the reacher environment. This behaviour suggests that the amount of exploration beneficial to performance is highly dependent on the task to solve, and more exploration is not necessarily always beneficial. However, in environments where exploration is beneficial, the walker results further strengthen the importance of continued exploration, as suggested by the halfcheetah results.

6.1.3 Model Uncertainty

Introducing targeted exploration in PETS shows varying effects in terms of reward performance. However, in all tasks, excluding cartpole where the task seems to be solved entirely, the use of exploration helps reduce model uncertainty compared to PETS, generally with the random sampling exploration agent resulting in the lowest model uncertainty.

Focusing on the walker and halfcheetah environments in figure 5.3 and 5.7 shows a clear distinction between all three methods. The uncertainty in the walker models shows a much more significant distinction between

the three methods, which correlates well with the amount of exploration used in this task; more exploration seems to result in a more certain model. Although smaller, the same distinction is seen in the halfcheetah model uncertainty, where the amount of exploration, in general, is much lower. These results back up the hypothesis that exploring uncertainty areas results in higher information gain and, thus, lower model uncertainty.

The model uncertainty in the reacher task, seen in figure 5.11, also suggests that exploration can help reduce uncertainty. However, the model uncertainty increases slightly towards the end of the training, indicating that the model might be overfitting on the data from the task subspace. Thus, a larger model might be beneficial in this environment compared to the rest. Moreover, the uncertainty is estimated on random data, and the states used in this estimation might not be part of the task space on which the model is trained, which can explain the development in model uncertainty seen in figure 5.11.

However, despite the increase in uncertainty, both exploration agents show reduced uncertainty compared to PETS, especially toward the end of the training, when uncertainty starts to increase. This reduction indicates that the exploration agents learn a more general dynamics model compared to PETS. However, if model complexity is not high enough to adequately capture the robot dynamics, this can also explain the reduced performance in terms of reward.

The PETS agent is less likely to visit states outside of the task space than

the two exploring agents. Thus, resulting in lower uncertainty on task data and higher uncertainty on general data, which is what the results in figure 5.11 represent. However, despite any potential overfitting, similar to walker and halfcheetah, these results also suggest a general model improvement with the use of exploration.

In summary, the results suggest that the introduction of exploration yields overall lower model uncertainty. Which fits well with both the hypothesis and previous work on information gain [42], suggesting that information gain is increased by utilizing uncertainty to target exploration.

6.2 Uncertainty Estimation and Exploration Weighting

The two versions of the exploration algorithm evaluate model uncertainty, σ_m^2, in different ways, either using data from a random controller or using data collected with an exploitation based policy. Results from all environments, excluding cartpole, show that the agent that utilizes policy data results in faster convergence of ε. Consequently, the amount of exploration is also reduced significantly in this exploration agent compared to the random sampling version. In all tasks except cartpole, results show that the reduced amount of exploration results in higher overall model uncertainty. In the walker and halfcheetah tasks where exploration is beneficial for performance, the reduced exploration also results in lower performance. This difference implies that using a random controller to sample data to evaluate uncertainty is the more prominent approach. At least with the current update law for ε.

6.2.1 Limitations of the Exploration Weighting

If the algorithm is to be used in a physical system and not only in simulations, the approach of sampling random data might be undesirable. In a simulation, applying random actions works perfectly well. However, applying random input to a physical system might cause unsafe or damaging behaviour to the robot or environment. Thus, following a safer

policy might be a better option. Results show that the lower amount of exploration done by the policy sampling agent still improves model uncertainty as well as performance compared to PETS which does not have include any explicit exploration. However, if the algorithm is to allow for more exploration while still maintaining the safety of following a planning policy compared to random input, a different update law for ε might be needed.

The current update law for ε is a linear scaling of the current model uncertainty with the maximum uncertainty. Although easy to implement, it might not offer the smoothest or optimal adaptation. In all of the tasks used to test the algorithm, the value of ε experiences significant "jumps" in value. This behaviour is evident in reacher and walker environments (figure 5.10 and 5.2) where ε oscillates between the first few iterations. In the halfcheetah environment (figure 5.6), ε rather quickly converges to 0, discarding the potential benefit of further exploration. These effects might be reduced by a less "aggressive" adaptation law. Smoothing out the curve of ε might allow the algorithm to utilize the safety of a policy sampling agent and still achieve the benefits from more exploration, as seen with the random sampling. However, this is left for future work.

6.3 Model Generalizability

The results in section 5.5 show the trained model's ability to be used for planning in different tasks than they were originally trained. The overall results show that the trained exploration agent achieves higher performance in all tasks tested compared to the PETS agent, indicating a clear correlation between the reduced model uncertainty and ability to learn the general dynamics of the environment.

When testing the trained agents on the halfcheetah environment by aiming to run at a constant speed, the performance of both the PETS agent and the exploration agent is very similar when running at the low speeds of 2 and 4 m/s, with only a slight performance improvement from the exploration agent. However, running at a faster velocity of 8 m/s shows a lower reward for both agents, and there is a much more significant performance difference between the two, as seen in figure 5.17. Since the agents are trained to solve the task of running as fast as possible, they naturally will run faster and faster with each training episode, and as mentioned in section 5.5 the final agents achieve velocities of up to 10 m/s. Thus, the lower velocities of 2 and 4 will naturally contribute to a a larger portion of the sampled training data than the higher velocity of 8. This, in addition to the time it takes to accelerate, can explain the better performance on the lower desired velocities from both agents. Furthermore, as seen in figure 5.8, the final performance of both agents on the default halfcheetah task are almost identical, which should suggest that the performance on higher

velocities would be pretty similar as well. However, this is not the case as the exploration agent achieves achieves significantly higher reward than the PETS agent when tracking 8 m/s.

There are a few potential reasons for this behaviour. Firstly, the reduced model uncertainty of the exploration agent seen in figure5.7 implies that the model is generally more accurate. Thus, it should be able to more precisely predict the dynamics of unknown states than the model trained with PETS. Consequently, this should give better control of the cheetah in lesser-known states, which suggests a reason for the exploration agents' performance improvement over PETS when tracking a higher velocity. Secondly, the exploration agent achieves higher sample efficiency than PETS, as it is able to reach higher velocity states earlier in training, as suggested by the checkpoint performance in figure 5.8. Thus the exploration agent might sample more data in the higher velocity states than the PETS agent. Since the trained model is more accurate in states similar to the data it is trained on, the exploration agent then results in a model more accurate on higher velocities. This, in turn, results in better performance when tracking the higher velocity of 8 m/s.

Interestingly the performance of the exploration is slightly better on the lower velocities as well, even though the PETS model is potentially trained on higher amounts of low-velocity data. This slight performance difference further strengthens the results that the overall model using exploration is more accurate on a broader range of the state space with overall lower uncertainty than the model from the PETS agent. Thus, the exploration

algorithm learns a more generalized dynamics model.

The test performed on the walker environment where the new task was to walk backwards instead of forwards suggests similar model improvements. As seen in figure 5.18, the trained exploration agent achieves noticeably better performance on this new task than the PETS agent. As discussed in section 6.1.2, all agents seem to be stuck in some local minimum when training ends after 200k steps. Thus, the performance difference seen in figure 5.18 is likely a similar performance difference to the one seen at the end of training and a potential consequence of the additional "is alive" reward the walker environment gives. However, the performance is clearly improved with the use of exploration, and the exploration agent is able to keep the walker "alive" longer than the PETS agent when aiming to solve the task of walking backwards. This result further suggests a general improvement in terms of model accuracy when utilizing exploration compared to PETS.

In summary, the trained exploration agent outperforms the PETS agent when set to solve different tasks with the same dynamics. This result suggests a more general and accurate model resulting from the incorporation of exploration in training, which can be very useful in a transfer learning setting. The model from one task forms a baseline for other tasks, with a more accurate model the exploration can allow for faster training of different agents in the same environment.

Chapter 7

Conclusion and Future Work

This chapter concludes the thesis. Firstly the conclusion summarises the report and results discussed in chapter 6. Followed by a section on different directions for future research.

7.1 Conclusion

This thesis presented a model-based reinforcement learning algorithm incorporating explicit exploration in the state of the art algorithm PETS. The algorithm was implemented as an extension to the existing framework *mbrl-lib* [2]. Furthermore, relevant literature within the field of deep reinforcement learning was reviewed, and the algorithm's performance was analyzed in terms of its sample efficiency and the accuracy and uncertainty of the resulting dynamics model.

The proposed algorithm utilizes uncertainty aware network ensembles to model the environment's dynamics and estimate model uncertainty. By targeting exploration towards states with high model uncertainty, the algorithm aims to increase information gain in each training step, resulting in improved sample efficiency with lower model uncertainty in comparison with the baseline PETS algorithm. Two versions of the algorithm were assessed. One evaluates the general model uncertainty with random state data to guide exploration over the entire state space. The other evaluates model uncertainty on the task space by exploiting the current model when sampling data for estimation. The version that utilized random data generally utilized more exploration in all tasks than the one using task space data, resulting in a more general and less uncertain dynamics model.

Results indicate that the introduction of explicit exploration is beneficial in increasing sample efficiency and reducing model uncertainty by increasing information gain and speeding up learning. Both versions of the explo-

ration algorithm improved the sample efficiency of PETS and resulted in a more accurate dynamics model. However, results on the reacher environment also suggest that the amount of beneficial exploration is highly dependent on the task and can, if the task itself induces natural exploration, reduce overall performance rather than improve it. Where exploration is beneficial, the incorporation of exploration also results in a more generalized dynamics model able to capture a broader range of the state space dynamics compared to the dynamics model learned with PETS.

7.2 Future Work

The benchmark performance is highly dependent on the amount of exploration done by the agent. Therefore, it would be interesting to further investigate other options for updating the exploration weighting. By incorporating adaptive methods from model-free RL or adaptive control, it is likely that the developed algorithm can achieve further improved performance on tasks where exploration is beneficial.

Lastly, the algorithm developed in this thesis is based off of uncertainty estimation and planning. However, the concepts regarding efficient model learning through maximizing information gain applies to most MBRL algorithms. Therefore, it would be interesting to investigate the potential for incorporating explicit exploration in algorithms that train a policy. Dyna-style methods that train a policy by sampling the trained model should also benefit from a more certain and accurate model. Thus, incorporating explicit exploration should also help increase performance of these types of algorithms.

Bibliography

[1] Jens Erik Kveen. *Dynamics Modelling with Probabilistic and Bayesian Network Ensembles*. Dec. 2021.

[2] Luis Pineda et al. "MBRL-Lib: A Modular Library for Model-based Reinforcement Learning". In: *Arxiv* (2021). URL: https://arxiv.org/abs/2104.10159.

[3] Cristina Pinneri et al. *Sample-efficient Cross-Entropy Method for Real-time Planning*. 2020. DOI: 10.48550/ARXIV.2008.06389. URL: https://arxiv.org/abs/2008.06389.

[4] Kurtland Chua et al. "Deep Reinforcement Learning in a Handful of Trials using Probabilistic Dynamics Models". In: *Advances in Neural Information Processing Systems* 2018-Decem.Nips (2018), pp. 4754–4765. ISSN: 10495258. arXiv: 1805.12114. URL: https://arxiv.org/pdf/1805.12114.pdf.

[5] Richard S Sutton and Andrew G Barto. *Reinforcement Learning, An Introduction, Second Edition*. MIT Press, 2018.

[6] David Silver et al. "Mastering the Game of Go with Deep Neural Networks and Tree Search". In: *Nature* 529.7587 (Jan. 2016), pp. 484–489. ISSN: 0028-0836. DOI: 10.1038/nature16961.

[7] Gerald Tesauro. "Temporal Difference Learning and TD-Gammon". In: *Commun. ACM* 38.3 (Mar. 1995), pp. 58–68. ISSN: 0001-0782. DOI: 10.1145/203330.203343. URL: https://doi.org/10.1145/203330.203343.

[8] David Silver et al. "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play". In: *Science* 362.6419 (2018), pp. 1140–1144. DOI: 10.1126/science.aar6404. URL: https://www.science.org/doi/abs/10.1126/science.aar6404.

[9] Pieter Abbeel et al. "An Application of Reinforcement Learning to Aerobatic Helicopter Flight". In: *Advances in Neural Information Processing Systems*. Ed. by B Schölkopf, J Platt, and T Hoffman. Vol. 19. MIT Press, 2006. URL: https://proceedings.neurips.cc/paper/2006/file/98c39996bf1543e974747a2549b3107c-Paper.pdf.

[10] Timothy P. Lillicrap et al. *Continuous control with deep reinforcement learning*. 2015. DOI: 10.48550/ARXIV.1509.02971. URL: https://arxiv.org/abs/1509.02971.

[11] Aske Plaat, Walter Kosters, and Mike Preuss. *Deep Model-Based Reinforcement Learning for High-Dimensional Problems, a Survey.* 2020. arXiv: 2008.05598 [cs.LG].

[12] Richard S. Sutton. “Dyna, an Integrated Architecture for Learning, Planning, and Reacting”. In: *SIGART Bull.* 2.4 (July 1991), pp. 160–163. ISSN: 0163-5719. DOI: 10.1145/122344.122377. URL: https://doi.org/10.1145/122344.122377.

[13] Yarin Gal, Rowan McAllister, and Carl Edward Rasmussen. “Improving PILCO with Bayesian neural network dynamics models”. In: *Data-Efficient Machine Learning workshop, International Conference on Machine Learning*. 2016.

[14] Marc Deisenroth and Carl Rasmussen. “PILCO: A Model-Based and Data-Efficient Approach to Policy Search.” In: Jan. 2011, pp. 465–472.

[15] Michael Janner et al. *When to Trust Your Model: Model-Based Policy Optimization.* 2019. DOI: 10.48550/ARXIV.1906.08253. URL: https://arxiv.org/abs/1906.08253.

[16] Michael Lutter et al. “Learning Dynamics Models for Model Predictive Agents”. In: (2021), pp. 1–16. arXiv: 2109.14311. URL: http://arxiv.org/abs/2109.14311.

[17] Shixiang Gu et al. “Continuous Deep Q-Learning with Model-based Acceleration”. In: *Proceedings of The 33rd International Conference on Machine Learning*. Ed. by Maria Florina Balcan and Kilian Q.

Weinberger. Vol. 48. Proceedings of Machine Learning Research. New York, New York, USA: PMLR, June 2016, pp. 2829–2838. URL: https://proceedings.mlr.press/v48/gu16.html.

[18] Thanard Kurutach et al. *Model-Ensemble Trust-Region Policy Optimization*. 2018. DOI: 10.48550/ARXIV.1802.10592. URL: https://arxiv.org/abs/1802.10592.

[19] Tuomas Haarnoja et al. *Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor*. 2018. DOI: 10.48550/ARXIV.1801.01290. URL: https://arxiv.org/abs/1801.01290.

[20] Susan Amin et al. *A Survey of Exploration Methods in Reinforcement Learning*. 2021. DOI: 10.48550/ARXIV.2109.00157. URL: https://arxiv.org/abs/2109.00157.

[21] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural networks* 2.5 (1989), pp. 359–366.

[22] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. "Understanding of a convolutional neural network". In: *2017 International Conference on Engineering and Technology (ICET)*. 2017, pp. 1–6. DOI: 10.1109/ICEngTechnol.2017.8308186.

[23] Amit Borundiya. *Activation Function for Multi-Layer Nerual Networks*. June 2019. URL: https://medium.com/@aborundiya/

activation-function-for-multi-layer-neural-networks-a07ac473f69e.

[24] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016. Chap. 6.

[25] Glenn Palmer et al. *Calibrated bootstrap for uncertainty quantification in regression models*. 2021. DOI: 10.48550/ARXIV.2105.13303. URL: https://arxiv.org/abs/2105.13303.

[26] Zdravko I Botev et al. "Chapter 3 - The Cross-Entropy Method for Optimization". In: *Handbook of Statistics*. Ed. by C R Rao and Venu Govindaraju. Vol. 31. Handbook of Statistics. Elsevier, 2013, pp. 35–59. DOI: https://doi.org/10.1016/B978-0-444-53859-8.00003-5. URL: https://www.sciencedirect.com/science/article/pii/B9780444538598000035.

[27] Tingwu Wang et al. "Benchmarking Model-Based Reinforcement Learning". In: (), pp. 1–25. URL: https://www.cs.toronto.edu/$%5Csim$tingwuwang/mbrl.pdf.

[28] Tingwu Wang and Jimmy Ba. *Exploring Model-based Planning with Policy Networks*. 2019. DOI: 10.48550/ARXIV.1906.08649. URL: https://arxiv.org/abs/1906.08649.

[29] Christopher M. Bishop. *Pattern recognition and machine learning*. Springer Verlag, 2006.

[30] Sanket Kamthe and Marc Peter Deisenroth. *Data-Efficient Reinforcement Learning with Probabilistic Model Predictive Control.* 2017. DOI: 10.48550/ARXIV.1706.06491. URL: https://arxiv.org/abs/1706.06491.

[31] Richard Bellman. "Dynamic programming". In: *Science* 153.3731 (1966), pp. 34–37.

[32] Anusha Nagabandi et al. *Neural Network Dynamics for Model-Based Deep Reinforcement Learning with Model-Free Fine-Tuning.* 2017. arXiv: 1708.02596 [cs.LG].

[33] Amol Mandhane et al. *MuZero with Self-competition for Rate Control in VP9 Video Compression.* 2022. DOI: 10.48550/ARXIV.2202.06626. URL: https://arxiv.org/abs/2202.06626.

[34] Open AI. *Gym.* URL: https://gym.openai.com.

[35] 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.* 2012, pp. 5026–5033. DOI: 10.1109/IROS.2012.6386109.

[36] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32.* Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-

pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[37] Omry Yadan. *Hydra - A framework for elegantly configuring complex applications*. Github. 2019. URL: https://github.com/facebookresearch/hydra.

[38] Sebastian Thrun. *Efficient Exploration In Reinforcement Learning*. Tech. rep. CMU-CS-92-102. Pittsburgh, PA: Carnegie Mellon University, Jan. 1992.

[39] Deepak Pathak et al. *Curiosity-driven Exploration by Self-supervised Prediction*. 2017. DOI: 10.48550/ARXIV.1705.05363. URL: https://arxiv.org/abs/1705.05363.

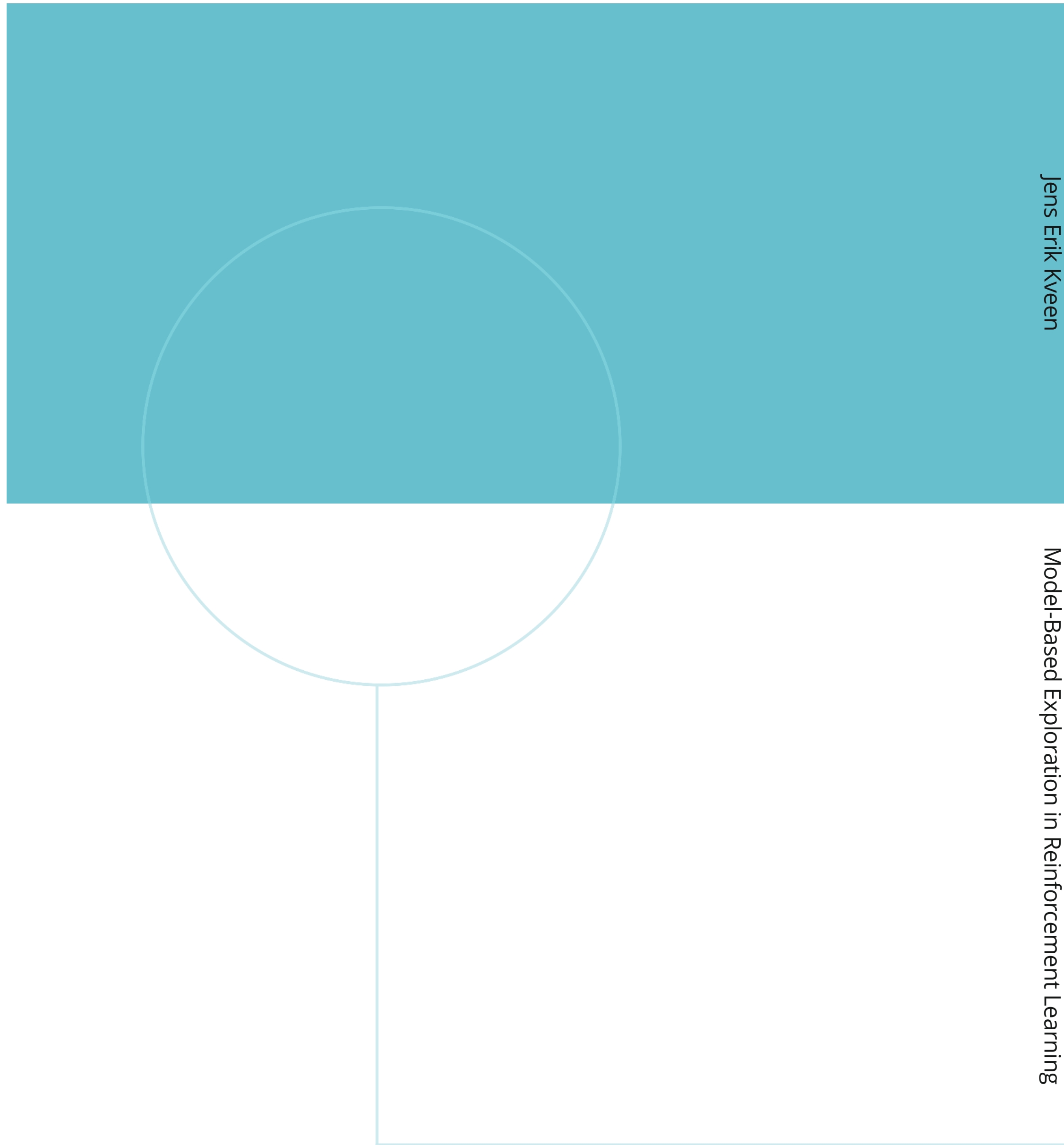
[40] Marc G. Bellemare et al. *Unifying Count-Based Exploration and Intrinsic Motivation*. 2016. DOI: 10.48550/ARXIV.1606.01868. URL: https://arxiv.org/abs/1606.01868.

[41] Deepak Pathak, Dhiraj Gandhi, and Abhinav Gupta. *Self-Supervised Exploration via Disagreement*. 2019. DOI: 10.48550/ARXIV.1906.04161. URL: https://arxiv.org/abs/1906.04161.

[42] H. S. Seung, M. Opper, and H. Sompolinsky. “Query by Committee”. In: *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*. COLT ’92. Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 1992, pp. 287–294. ISBN: 089791497X.

DOI: 10.1145/130385.130417. URL: https://doi.org/10.1145/130385.130417.

[43] Pauli Virtanen et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.



Jens Erik Kveen

Model-Based Exploration in Reinforcement Learning

NTNU

Kunnskap for en bedre verden