Partially Unsupervised Deep Meta-Reinforcement Learning

Teilweise unüberwachtes tiefes Meta-Reinforcement Learning
Master thesis by Jonas Eschmann
Date of submission: March 29, 2021

1. Review: M.Sc. Fabio Muratore
2. Review: Dr. Claudio Zito
3. Review: Prof. Jan Peters

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 und §23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Jonas Eschmann, die vorliegende Masterarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Fall eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß §23 Abs. 7 APB überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 29. März 2021

J. Eschmann

Abstract

Reinforcement Learning (RL) recently has been applied successfully to many domains. One particular example is the continuous control domain, where RL has been shown to solve complex tasks in simulation. However, it turns out that agents trained in simulation can usually not directly be transferred to real systems, and training directly using a real system is considered infeasible due to the large number of samples required. This work focuses on the simulation-to-reality (sim-to-real) transfer technique called domain randomization and shows that it closely relates to Meta-Reinforcement Learning (meta-RL). Based on this assessment, we propose a novel algorithm that is based on recent advancements in deep meta-RL to expand on the capabilities of domain randomization. In particular, we show that domain randomization faces issues with wide domain parameter distributions and demonstrate that our Partially Unsupervised Deep Meta-Reinforcement Learning (PUDM-RL) approach can handle these scenarios well. Furthermore, we improve the training stability by splitting the training into two separate parts. In one case, we train a dynamics encoder using unsupervised learning, while in the other case, we use the dynamics embedding from the learned dynamics encoder to aid the training of the agent. Experimentally we show that the learned dynamics encoder exhibits an informative latent space by compressing an interaction history from a particular environment into an embedding characterizing its dynamics. We find that agents aided by the dynamics embeddings perform much better than alternative approaches in the case of domain randomization with broad distributions over the dynamics parameters.

Zusammenfassung

Reinforcement Learning (RL) wurde zuletzt in vielen verschieden Bereichen erfolgreich angewandt. Einer dieser Bereiche ist die kontinuierliche Regelung, in der gezeigt wurde, dass RL komplexe Aufgaben in Simulationen lösen kann. Allerdings stellt sich heraus, dass in einer Simulation trainierte Agenten normalerweise nicht direkt auf echte Systeme übertragen werden können. Des Weiteren wird es aufgrund der Menge an benötigten Interaktionen als sehr schwer angesehen, Agenten direkt an einem echten System zu trainieren. In dieser Arbeit fokussieren wir uns auf Domain Randomization als Methode um den Transfer von der Simulation in die Realität (simulation-to-reality (sim-to-real)) zu ermöglichen und zeigen, dass die Methode eng mit Meta-Reinforcement Learning (meta-RL) verwandt ist. Basierend auf dieser Einsicht entwickeln wir einen neuartigen Algorithmus, welcher auf aktuellen Fortschritten im Bereich des meta-RL basiert, um die Möglichkeiten von Domain Randomization zu erweitern. Wir zeigen, dass Domain Randomization in Verbindung mit weiten Wahrscheinlichkeitsverteilungen Probleme bereitet, und dass unser Partially Unsupervised Deep Meta-Reinforcement Learning (PUDM-RL) Ansatz in diesen Szenarien gute Resultate liefert. Des Weiteren verbessern wir die Stabilität des Trainingsprozesses, indem wir diesen in zwei Teile aufspalten. In einem Teil trainieren wir einen Enkodierer für die dynamischen Eigenschaften von Systemen und nutzen dafür unüberwachtes Lernen. In dem anderen Teil nutzen wir die, in einen latenten Raum eingebetteten, Informationen des Enkodierers um das Training des Agenten zu unterstützen. Wir zeigen experimentell, dass der Enkodierer einen informativen latenten Raum aufweist, indem er die Interaktionshistorie in einen die dynamischen Eigenschaften des Systems beinhaltenden Vektor komprimiert. Dabei finden wir heraus, dass Agenten, welche die Informationen des Enkodierers zur Verfügung haben, im Falle von weiten Wahrscheinlichkeitsverteilungen über die Systemparameter, wesentlich bessere Resultate liefern als alternative Ansätze.

Contents

Figures, Tables and Algorithms

List of Figures

List of Tables

List of Algorithms

Abbreviations

List of Abbreviations

Notation	Description
ADR	Automatic Domain Randomization
AGI	Artificial General Intelligence
CNN	Convolutional Neural Network
DDPG	Deep Deterministic Policy Gradient
DNN	Dense Neural Network
DoF	Degree of Freedom
DPG	Deterministic Policy Gradient
DQN	Deep Q-Networks
DR	Domain Randomization
GAE	Generalized Advantage Estimation
GAN	Generative Adversarial Network
GPU	Graphics Processing Unit
GRU	Gated Recurrent Unit
i.i.d.	independent and identically distributed
LSTM	Long Short-Term Memory
MAML	Model Agnostic Meta-Learning
MCTS	Monte Carlo Tree Search
MDP	Markov Decision Process
meta-RL	Meta-Reinforcement Learning
MPC	Model Predictive Control

MSE	Mean Squared Error
POMDP	Partially Observable Markov Decision Process
PPO	Proximal Policy Optimization
PUDM-RL	Partially Unsupervised Deep Meta-Reinforcement Learning
ReLU	Rectified Linear Unit
RL	Reinforcement Learning
RNN	Recurrent Neural Network
SAC	Soft Actor-Critic
SGD	Stochastic Gradient Descent
sim-to-real	simulation-to-reality
sim-to-sim	simulation-to-simulation
TD	Temporal Difference
TD3	Twin Delayed Deep Deterministic Policy Gradient
TRPO	Trust Region Policy Optimization
VAE	Variational Autoencoder

1. Introduction

In recent years there has been rapid progress in machine learning, with deep learning enabling supervised and unsupervised models to scale with the vast (and rising) amount of data available due to cheap sensors (e.g. cameras) but also large text corpora crawled from the internet. Deep learning-based models have reached and increased state-of-the-art performance in many fields like, e.g., image classification [1]. With a short delay, deep learning methods have also made their way into Reinforcement Learning (RL), where they lead to big leaps like reaching human-level scores in Atari games [2] and beating the world champion in the game of Go [3]. In the continuous control domain, deep RL was shown to be capable of learning complex locomotion behavior in simulation [4] [5].

With the advances of RL for continuous control in simulation, the desire to apply policies to real robots arose. Taking into account the sample inefficiency of modern RL algorithms, it is generally considered infeasible to directly train a control policy on a robot due to wear and tear but also supervision requirements like resetting the environment after an episode has been executed. Facing this issue, two different strands of work have emerged. One is concerned with improving the sample efficiency of RL algorithms while the other is working on transferring learned behavior from simulation to reality (simulation-to-reality (sim-to-real)). In this work, we are focusing on the latter approach.

More particularly, we are focusing on the Domain Randomization (DR) approach to facilitate sim-to-real transfer. DR is based on the premise to train a single agent across a distribution of environments with varying dynamics parameters. After training the agent, the policy is transferred to the real system, which is assumed to be just another sample from the distribution over environments the agent has seen during training. This approach has been successfully applied to transfer, e.g., a policy for dexterous manipulation from simulation to reality [6].

Parallel to work on sim-to-real transfer using DR, there has also been increasingly successful work on deep Meta-Reinforcement Learning (meta-RL), which is concerned with training a single agent that can act well across different tasks. We draw the connection between meta-RL and DR and show the necessity of context-awareness under broad environment distributions. Our premise is to enable the training over vast distributions in the dynamics parameters, to avoid costly and task-specific modeling and system identification. We argue that it is beneficial to devise general algorithms that can be transferred to the real world after training in simple but widely varying simulations.

To facilitate training over broad distributions, we propose a novel training paradigm that separates the training of the inference of the environment from the training of the behavior. We refer to this approach as Partially Unsupervised Deep Meta-Reinforcement Learning (PUDM-RL) because it uses unsupervised learning of a dynamics encoder which compresses the characteristics of an environment's dynamics into a dynamics embedding vector. This vector can then be used by the RL agent to take more appropriate actions in each particular environment.

Aspirationally our algorithm also paves a way to Artificial General Intelligence (AGI), where the agent is able to quickly solve new tasks by using general world knowledge to reason about them. While practically tackling the AGI problem is very far from the current state-of-the-art, structurally, PUDM-RL could be a way to approach it. Additionally, our approach might also benefit from the effects (mainly the exponential increase in the available amount of computing power) described in Richard Sutton's *The Bitter Lesson* should they continue to hold in the future.

In Chapter 2, we will provide a general picture of work that our approach builds upon or is related to. After drawing the bigger picture, in Chapter 3, we provide the major foundations that our work is based on, namely RL, multi-task RL, meta-RL, and DR. In Chapter 4, we first theoretically analyze the latter two approaches and combine them into a joint framework. Based on viewing DR as meta-RL, we devise our PUDM-RL approach. We apply PUDM-RL in two continuous control environments (Pendulum and Acrobot) described in Chapter 5 and report the results in Chapter 6. In the final Chapter 7, based on the results, we discuss why we view our approach as a success and as a good foundation to move forward in future work.

2. Related Work

In recent years there has not only been vast progress in the field of Reinforcement Learning (RL) but also particularly in simulation-to-reality (sim-to-real) transfer and Meta-Reinforcement Learning (meta-RL), which are related to our approach.

In comparison to meta-RL, sim-to-real transfer is not only relevant to RL but also computer vision and other fields. Hence a broad variety of different approaches have been proposed to bridge the *reality gap*. Early mentions of the *reality gap* and the sim-to-real problem have been brought up in the context of evolutionary robotics [7], which is tightly related to RL for robotics. The major difference between evolutionary robotics and RL for robotics is the optimization scheme used and that in evolutionary robotics, sometimes not only the parameters of a controller are optimized but also the robot's morphology, which also spurs the need for simulation as building hundreds or thousands of different robots is prohibitively costly. In [7], the authors already suggest that the simulator to reality transfer should be a major direction for future research and propose the usage of a range of different environments in simulation. The authors also cast strong doubts about the long-term viability of simulators for evolutionary robotics, especially when it comes to the simulation of visual sensors. According to [8], this view was wide-spread, and it was believed that realistic simulators are first "so computationally expensive that all speed advantages over real-world evolution are lost" and second "so hard to design that the time taken in development outweighs time saved by fast evolution". To move forward, the author proposes the usage of fast, minimal simulations which capture all the important aspects of the real system but obfuscate less important aspects and implementation details by injecting noise sampled at the beginning of each trial/episode. As described in the following, the notion of a crude but fast simulator with widely varying distributions of dynamics is appealing again, as advancements in meta-RL allow us to train adaptive agents.

While there have been early reports of sim-to-real transfer failing with RL derived policies [9] [10], the sim-to-real problem for RL based agents only gained in relevance after new continuous control algorithms have been developed and demonstrated to excel at complex robotics tasks like walking [11] [12] [4] [13] [5] [14]. Prior to these demonstrations of policies learning complex behavior in simulation, the research in sim-to-real transfer for robots has been mainly focused on visual perception. Given its versatility and effectiveness, the most popular approach for the sim-to-real transfer of visual perception modules has been Domain Randomization (DR) over textures [15] [16] [17] [18]. While in these works the focus is on transferring a perception module from simulated scenes to reality, control policies are either not included in the evaluation [15] or trained by supervision of a privileged expert [16] [17] [18]. In [19] [20] [21], advanced RL algorithms are combined with visual DR, achieving successful sim-to-real transfer. A different approach to visual sim-to-real transfer problems is domain adaptation, where samples are adapted to be more likely in a desired domain. In [22], a Generative Adversarial Network (GAN) was used to make simulated scenes appear more real, while in [23], a combination of two Variational Autoencoders (VAEs) is used to transfer images from a source and a target domain into a new shared domain, which is in turn used to train a position detector.

While often conflated, DR over dynamics parameters poses a different problem than visual DR (more details in Section 3.4.1). With the emergence of powerful RL algorithms, the desire to transfer policies capable of learning complex behavior from simulation to real robots increased. Successful sim-to-real transfer using DR over dynamics parameters has been reported in [24] [25] [26] [27]. In [28], DR of dynamics parameters is formalized in terms of the simulation optimization bias, deriving a tractable upper bound for the optimality gap based on the uncertainty in the true dynamics parameters of the real system.

By directly training a convolutional neural network-based grasp quality predictor for two months on up to 16 similar real robots in parallel, an orthogonal approach has been demonstrated in [29]. The result of this effort is a grasping policy that is able to pick up (and place) objects with a comparatively good failure rate of $10 - 17.5\%$ and a huge dataset of images and grasp success labels. Though it is invariant wrt. the camera position to some degree, the dataset is not transferable to other setups because the labels are highly dependent on the type of gripper used.

In addition to the desire for sim-to-real transfer of trained policies, there is also an interest in modifying the modern RL algorithms to generate more versatile agents that can act well in multiple tasks. Multi-task RL is often applied to increase sample efficiency in the individual tasks through positive transfer (cf. Section 3.2). The improved sample complexity of multi-task RL is theoretically confirmed in [30]. In this work, the authors already use the meta-RL paradigm, referring to it as multi-task RL, which is distinguished more clearly in recent literature [31] (refer to Section 3.3 for a distinction). The notion of meta-RL is appealing because it allows to devise a fast learning algorithm which is using an inductive bias based on a particular set of tasks that the system is trained on. This appeal has been recognized in early work on meta-RL [32], where the *incremental self-improvement paradigm* has been introduced. Many of the foundational ideas presented in this work have been carried into recent research, while implementation details like dismissing the concept of episodes/trials as unrealistic have not proven practical.

More recently, the term deep meta-RL has been coined in [33], and an almost identical algorithm has been proposed in [34]. The usage of Recurrent Neural Networks (RNNs) for the actor and critic in these approaches allows the agent to use past interactions to reason about the current task at hand (i.e. the transition and/or reward function of the current Markov Decision Process (MDP)). This setup poses a special case of the more general setup proposed earlier for general Partially Observable Markov Decision Processes (POMDPs) in [35] or [36] (cf. Section 3.2 for more details about the relation of multi-task RL and meta-RL to general Partially Observable Markov Decision Processes (POMDPs)). The usage of RNNs for meta-RL algorithms is inspired by [37], where a Long Short-Term Memory (LSTM) network [38] is successfully trained for few-shot meta-learning.

In addition to these inference-based meta-RL approaches, there is another recent, optimization-based contender called Model Agnostic Meta-Learning (MAML) [39]. MAML is not restricted to RL because it uses a two-level gradient-based optimization scheme, where the outer training process optimizes the outcome of an inner optimization process (consisting of a few gradient steps) that is based on data for a particular, current task. In [31], a benchmark suite for multi-task RL and meta-RL is introduced, where the mentioned meta-RL algorithms are compared using 50 qualitatively different manipulation tasks. The evaluations show that meta-RL and even multi-task RL are far from being solved and also yield no clear winner under the examined algorithms.

With the emergence of meta-RL methods, the connection between DR, multi-task RL and meta-RL has recently started to be recognized. Works in this realm are most related to our approach. In [6] and [40], the deep meta-RL algorithm proposed in [33] and [34] has been very successfully applied in combination with DR (visual & dynamics) to solve highly complex robotic manipulation tasks using a high Degree of Freedom (DoF) robotic hand. In the latter work, the authors also include the idea of asymmetric actor-critic learning from

[21] and propose a simple but effective DR scheduling mechanism (called Automatic Domain Randomization (ADR)). The goal of ADR is to train the agent on an ever broader distribution over environments and draws ideas from minimal simulations which have been already proposed for evolutionary robotics in [8] (described at the beginning of this chapter). The authors observe and examine emergent meta-learning behavior and attribute this to the broadness of the task distribution their system is able to handle. In comparison to our work, the authors evaluate the meta-RL behavior only empirically and justify their setup post-hoc. Additionally, they do not decouple the training setup and use on-policy RL.

Off-policy RL (described in Section 3.1.4) with RNNs has been described as challenging in [41] and is examined thoroughly in [42]. Despite this challenge, in [43], the authors are able to transfer a trained policy from simulation to reality for a comparatively simple task using DR and a recurrent Deep Deterministic Policy Gradient (DDPG) (cf. Section 3.1.4) architecture.

In [41], off-policy RL is used in combination with an amortized inference network which gives a distribution over context variables to condition the policy. The authors focus on qualitatively different tasks and do not split the training procedure. While the authors report good results in simulation, we would not expect this approach to work well for sim-to-real transfer because the structure of the amortized inference network is not suited to cope with observation noise (more details in Section 4.3).

A different approach has been taken in [44], where a dedicated *environment probing policy* is used to generate system response data at the beginning of an episode, which is then embedded to condition the behavioral policy. We think that this approach goes in the right direction by embedding an interaction history but believe that the structure to optimize the information in the gathered data is unnecessarily complex. At the same time, the structure (a fully connected neural network) of the embedding network is too simple and not exploiting the structure of the data. Furthermore, using a fully connected neural network for their embedding network, the authors necessitate a fixed split between the collection policy and the acting policy, which is arbitrary and should rather be determined by the agent itself. In our approach, a single policy learns the trade-off between environment probing and acting to receive a high reward by implicitly encoding uncertainty into the embeddings. Additionally, our approach can be directly applied to adaptive control, where the dynamics change during the episode. If at some point, the uncertainty about the dynamics grows due to a change in the dynamics (leading to the past interactions being inconsistent), this can be expressed through uncertainty in the embeddings, and the policy can take actions to probe the new dynamics. The policy may, depending on the uncertainty, also combine the probing with acting in a useful way (e.g., when stabilized in some point, to wiggle around this point to increase the certainty about the dynamics while still staying in the range where it can quickly stabilize again). The approach with a dedicated probing policy proposed in [44] has been studied for sim-to-real transfer in [45] but found to be outperformed by simpler means like DR and random force injection.

3. Foundations

In this chapter, we describe the foundations that our Partially Unsupervised Deep Meta-Reinforcement Learning (PUDM-RL) approach is based on. First, we lay out the foundations of Reinforcement Learning (RL). Based on these foundations, we extend the scope to multi-task and Meta-Reinforcement Learning (meta-RL). Additionally, we describe the challenges with transferring RL-based agents that have been trained in simulation to real systems and simulation-to-reality (sim-to-real) approaches that aim to tackle them. In particular we focus on sim-to-real approaches using Domain Randomization (DR).

3.1. Reinforcement Learning

This work is based heavily on Reinforcement Learning (RL) based approaches as means to find solutions to Markov Decision Processes (MDPs). MDPs form a very general framework that can model a wide variety of real-world tasks. They recognize that many systems are based on a state that evolves over time, forming a trajectory and that this trajectory of subsequent states can be influenced by taking actions along the way. The evolvement of the system over time is governed by a transition model which resembles the insight from control theory that all real-time systems are causal. The real-time aspect is important as we are interested in making decisions, i.e., taking actions, without knowledge of future states. This insight is implemented by defining a forward model for the change-rate in the case of time-continuous systems or the next state in the case of time-discrete systems, given the current state and action. Importantly this forward model fulfills the Markov property and is hence conditionally independent of previous states and actions, given a current state and action.

This thesis only deals with the case of time-discrete systems as that is a precondition for RL to be applicable. Applying RL means using RL approaches to derive behavior for a given MDP. The behavior is characterized by a policy (synonymous with controller) that governs the selection of an action given only the current state. The current state is sufficient for action selection because, in combination with the action taken, it is sufficient to describe the probability distributions of the next state and reward due to the Markov assumption described earlier.

Because for each MDP, there are many possible different behaviors, there needs to be a way to characterize the value of a particular behavior. This is done by means of a reward model providing a scalar reward at each interaction, based on the state and the action taken by the policy.

With MDPs, the described transition and reward model are probabilistic, i.e., they are modeled by conditional probability distributions. With this, a policy (in combination with a distribution over the initial conditions) gives rise to a distribution over trajectories. The distribution over trajectories, in combination with the distribution over actions, in turn, gives rise to a distribution over rewards. The rewards for a trajectory can be aggregated into a scalar return, which in the case of a distribution over rewards leads to a distribution over scalar returns.

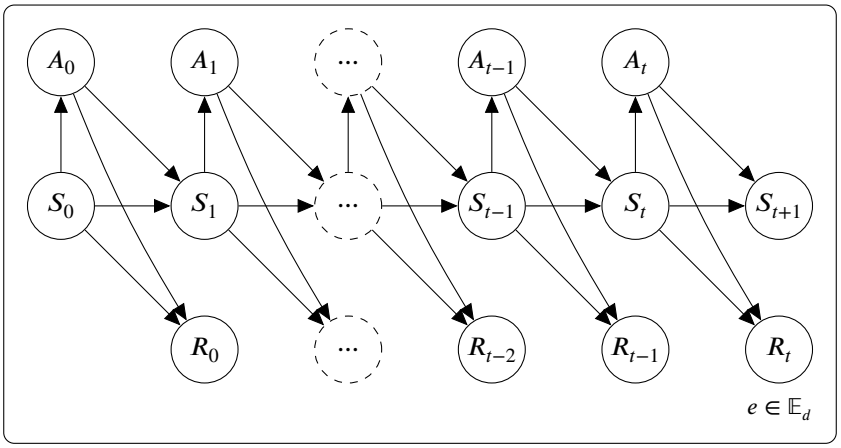


Figure 3.1.: Bayesian network depicting an MDP as it is used in RL. The action distribution at each timestep is governed by a policy that is conditionally independent of past states and actions given the current state.

The aggregation is usually done by summation and optionally using a weighting scheme, where often an exponential decay in time is used.

Sim-to-real (described in detail in Section 3.4) is mainly applicable and challenging in the case of continuous control, where states and actions are (multidimensional) continuous variables. Because of this and the focus on RL as the class of approaches to derive behavior, in the following, only the case of discrete-time and continuous states and actions is considered.

In the following, the notational standard for MDPs MDPNv1 defined in [46] is used. For an introductory treatment of RL, please refer to the book [47]. The structure of an MDP, including a Markovian policy governing the action distribution depending only on the current state, can be represented by a directed graphical model (Bayesian network), as shown in Figure 3.1.

Using the d-separation criterion, we can observe that this Bayesian network indeed represents the conditional independence of a next state S_{t+1} from all past states S_x and A_x with $0 <= x < t$ for each episode e from the set of all episodes $\mathbb{E}$. Per the definition of the platter notation, the states and actions of different episodes e are unconditionally independent. While the distribution of the start state S_0 is given by the initial state distribution d_0, the subsequent states are distributed according to the transition function, depending on the previous state and action. The representation of MDPs as Bayesian networks helps to understand and prove (conditional) independencies.

To keep the graphical networks concise, from now on, we will disregard the rewards because all the approaches presented in this work are based on modifications to the transition function while the reward function is kept unmodified. Especially in the case of multi-task RL, it can be of interest to work with varying reward functions, and it is possible to straightforwardly extend the presented approaches to the general case of varying dynamics and reward functions. Without the rewards, the single task MDP can be represented by the graphical model in Figure 3.2. In the following sections, we will use this graphical model as a base to show how, e.g., DR or multi-task RL influences the dependency graph.

The following foundational sections are intended to describe the major building blocks of the proposed PUDM-RL approach while highlighting certain aspects in which this thesis deviates or extends from the common

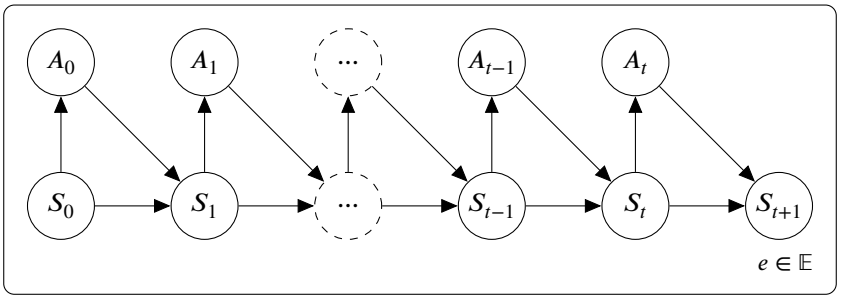


Figure 3.2.: Bayesian network hiding the random variables of the rewards to maintain conciseness when extending this network to the domain randomized case.

literature.

3.1.1. Finite vs. Infinite Horizon Reinforcement Learning in Common, Time-Limited Environments

When we take the definition of MPDs from either MDPNv1 [46] or the seminal RL book [47] and apply it to common time-limited environments like the Pendulum of the popular OpenAI gym collection ([48]), it can be argued that the environments do not adhere to the conditional independence property required to model them as MDPs. In both formulations, termination is defined as ending up in a terminal absorbing state. Following the MDPNv1 notational standard, this terminal state is denoted as $\overset{\infty}{s}$. For the transition probability into the final terminal state, the Markov assumption does not hold in general:

$$P\left(\overset{\infty}{s} | s, a, s_{t-1}, a_{t-1}, \cdots, s_0, a_0\right) \neq P\left(\overset{\infty}{s} | s, a\right)$$

This is due to the fact that if the number of previous states is known, the probability of ending up in the terminal state $\overset{\infty}{s}$ at the final step of a time-limited environment ($t = T$) is one and zero in all prior steps:

$$P\left(\overset{\infty}{s} | s, a, s_{T-1}, a_{T-1}, \cdots, s_0, a_0\right) = 1 \neq P\left(\overset{\infty}{s} | s, a\right)$$

In our experiments, we found that treating these time-limited environments as MDPs and assuming conditional independence, the uncertainty of termination can be detrimental to the learning process (cf. Section 6.1.1). This could be explained by the value function estimation receiving very noisy signals. Depending on wether a state is visited at the beginning of the trajectory or at a later part, the value of a particular state varies, but the value function estimator can not account for that because it only receives the current state (and action) due to the Markov assumption. A solution could be to augment the state with an additional step variable that gives information about the current timestep in the course of an episode. This would restore the Markov property and would make the modeling of the environment as an MDP sound.

But this also emphasizes that optimizing this augmented problem might not be the intended goal of the designers of the environment because they probably intentionally left out the time from the state representation to formulate a stationary problem.

The solution we chose to tackle this problem is a bit different in that we disregard the time-limit and model the problem as an infinite horizon MDP but train it in an episodic way. As usually with infinite horizon RL, there is a need for a discount factor $\gamma < 1$ to not run into the problem Marcus Hutter described as "Immortal agents are lazy" [49]. This approach is also used in the SpinningUP implementation by OpenAI (the creators of the gym package) in their Soft Actor-Critic (SAC) implementation[1] [50]:

```
# Ignore the "done" signal if it comes from hitting the time
# horizon (that is, when it's an artificial terminal signal
# that isn't based on the agent's state)
d = False if ep_len==max_ep_len else d
```

In Section 6.1.1, we show that the issue described in this section has practical implications and can cause bad performance, even when using popular, stable RL implementations.

3.1.2. Reinforcement Learning Algorithms

This section provides insights into why certain RL algorithms have been chosen in this work.

As stated earlier, the focus of this thesis is on applying RL methods to solve continuous control tasks. A major distinction of structurally different RL methods is if they are model-based or model-free. In the model-based case, there can be a further distinction between algorithms that learn a model from interactions and algorithms that need a prior model. In this context model usually refers to both the transition and reward model. On the other side, there are model-free approaches that have been identified to be preferable for robotics applications in previous publications. In their survey on policy search for robotics [51], the authors state that despite their huge number of required robot interactions, learning the policy is often easier than learning an accurate forward model. This statement has stood the test of time very well in that model-free methods are still considered sample inefficient but have shown the most impressive results in continuous control up to this point.

Opposing views to this can be found when looking beyond robotics and at RL in general. For example, Richard Sutton proposes his latest revision, *SuperDyna* of the model-based *Dyna* algorithm [47], as the way to expand the capabilities of RL. The argument for model-free RL algorithms is also challenged by MuZero [52], which is a model-based evolution of AlphaZero [53]. In comparison to AlphaZero, it learns a transition model to execute the Monte Carlo Tree Search (MCTS) in a latent representation of the state. Because of this, it no longer requires prior knowledge of the transition rules, which expands its scope from discrete environments with known transition dynamics (like chess and go) to the more general class of environments with (possibly) unknown transition and reward models. This enabled the application of MuZero to the Atari benchmark suite, which yielded state-of-the-art scores. Technically there is nothing limiting one from using the Atari simulator as a given model and using it in the MCTS directly, were it not for the design of the Atari environments. So for future studies, it would be interesting to apply this promising approach to cases where the model is either truly not available (e.g. real-world applications), or evaluation of the model is so computationally intensive that the learned model provides benefit in wall-clock training and execution time. In addition to that, more work on extending MuZero to continuous actions could bring the class of model-based RL approaches back into consideration for robotic applications.

[1] Link to the code snippet: `https://github.com/openai/spinningup/blob/20921137141b154454c0a2698709d9f9a0302101/spinup/algos/pytorch/sac/sac.py#L301`

Based on these considerations, this thesis will focus on established, model-free algorithms for continuous control with continuous states and actions. The continuous control requirement rules out a broad class of algorithms that rely on either discrete states and actions or continuous states and discrete actions. The case of discrete states and continuous actions is usually not considered in RL. The reason is that this case is not applicable in the case of deterministic transition dynamics and lacks real-world applications in the case of stochastic transition dynamics.

The continuity of states rules out tabular methods like Q-learning described in [54]. The continuity in the actions also rules out methods based (purely) on value function approximation which could work with continuous states. These very capable methods are usually based on Deep Q-Networks (DQN) (proposed in [2]) that arguably sparked the recent interest in deep RL by achieving (super-)human performance on games from the Atari benchmark suite. But these are not suited for continuous actions as they rely on being able to take the *argmax* of the approximated action-value function $Q\left(s_t, a\right)$ for action selection

$$a_t = \begin{cases} a \sim \text{Uniform}\{\mathcal{A}\}, & \text{if } x < \epsilon \\ \underset{a}{\text{argmax}}\, Q\left(s_t, a\right), & else \end{cases}, \quad x \sim \text{Uniform}\,(0, 1)$$

which is only feasible in the case of discrete actions.

Policy Gradient Methods

Ruling out these value estimation-centered approaches leads to the class of policy search methods. Instead of first estimating a value function and then deriving the policy from the state-(action) value function, policy search methods directly optimize a parametric policy. Apart from the requirement of continuous actions, there are also other arguments in favor of directly optimizing the policy. For example, in [55], the authors state that optimal policies often have fewer parameters than optimal value functions.

Policy search methods aim at directly optimizing a parametric policy by estimating the gradient of the expected return with respect to the policy's parameters. This gradient can then be used in gradient-based optimization algorithms like Stochastic Gradient Descent (SGD) to optimize the expected return. In the undiscounted, episodic case (can be extended to the discounted episodic case w.l.o.g.), the return G_k following time step k of an episode is the sum of the received rewards:

$$G_k = \sum_{t=k}^{\infty} R_t$$

The sum runs from $t{=}k$ to $t{=}\infty$ to generalize the case of episodes of different lengths. This exploits the definition of a terminal, absorbing state $\overset{\infty}{s}$ in the MDPNv1. The goal is to maximize the expected total return:

$$J(\boldsymbol{\theta}) = \mathbb{E}\left[G_0\right] \tag{3.1}$$

Where $\boldsymbol{\theta}$ are the parameters of a parametric policy (as defined in MDPNv1) and the expectation of the return G_0 is taken over the distribution $Pr(G_0 = g_0)$ with g_0 being a particular return starting from timestep 0.

REINFORCE We will start deriving the REINFORCE algorithm and then extend it to the policy gradient theorem. Historically policy search methods were united under the REINFORCE gradient estimator described in [56]. REINFORCE employs the log-likelihood ratio trick, which allows to express the gradient of the expected return as an expectation over trajectory samples.

Starting from Equation (3.1), the expectation can be expanded to an integral, where we can make the dependence of the return G_0 on the trajectory T explicit:

$$\begin{aligned} P(g_0) &:= Pr(G_0 = g_0) \\ J(\boldsymbol{\theta}) = \mathbb{E}\left[G_0\right] &= \int_{g_0} P(g_0) g_0 \, \mathrm{d}g_0 \\ &= \int_{g_0} \int_{\tau} P(g_0 \mid \tau) P(\tau) g_0 \, \mathrm{d}\tau \, \mathrm{d}g_0 \\ &= \int_{\tau} P(\tau) \int_{g_0} P(g_0 \mid \tau) g_0 \, \mathrm{d}g_0 \, \mathrm{d}\tau \\ &= \int_{\tau} P(\tau) \, \mathbb{E}\left[G_0 \mid T = \tau\right] \, \mathrm{d}\tau \end{aligned}$$

We focus on the distribution over trajectories because the (conditional) distribution over returns is governed by the given environment, and $\mathbb{E}\left[G_0 \mid T = \tau\right]$ is approximated by sampling as part of the trial and error learning innate to RL.

The random variable T is distributed according to the trajectory distribution $Pr(T = \tau)$, where τ represents a trajectory consisting of particular states and actions over time $\{s_0, s_1, \ldots, s_\infty; a_0, a_1, \ldots, a_\infty\}$ (the trajectory view is inspired by [57]). The probability of a trajectory is defined by the initial state distribution $d_0(s)$ and subsequent states that are governed by the transition dynamics while following the policy:

$$P(\tau) := Pr(T = \tau) = d_0(s_0) \cdot \prod_{t=0}^{\infty} \left[P(s_{t+1} \mid s_t, a_t) \pi_{\boldsymbol{\theta}}(a_t \mid s_t)\right]$$

Taking the log of the trajectory distribution simplifies it to a sum of log probabilities:

$$\log P(\tau) = \log d_0(s_0) + \sum_{t=0}^{\infty} \left[\log P(s_{t+1} \mid s_t, a_t) + \log \pi_{\boldsymbol{\theta}}(a_t \mid s_t)\right]$$

With this sum of terms taking the gradient with respect to the policy parameters $\boldsymbol{\theta}$ is straightforward:

$$\nabla_{\boldsymbol{\theta}} \log P(\tau) = \sum_{t=0}^{\infty} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid s_t) \tag{3.2}$$

These definitions can now help to find an estimator for the gradient of the expected total return $J(\boldsymbol{\theta})$ with respect to the policy parameters $\boldsymbol{\theta}$:

$$\begin{aligned} J(\boldsymbol{\theta}) &= \int_{\tau} P(\tau) \, \mathbb{E}\left[G_0 \mid T = \tau\right] \, \mathrm{d}\tau \\ \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \nabla_{\boldsymbol{\theta}} \int_{\tau} P(\tau) \, \mathbb{E}\left[G_0 \mid T = \tau\right] \, \mathrm{d}\tau \\ &= \int_{\tau} \nabla_{\boldsymbol{\theta}} \left[P(\tau)\right] \, \mathbb{E}\left[G_0 \mid T = \tau\right] \, \mathrm{d}\tau \end{aligned} \tag{3.3}$$

Under mild conditions (the conditions of the Leibniz rule [58]), we can pull the gradient operator into the integral over trajectories. The expected total return is independent of the policy parameters given a particular trajectory. In other words, the policy parameters $\boldsymbol{\theta}$ only influence the probability of a trajectory and not the states and actions themselves. Hence the expected gradient of the expected total return with respect to the policy parameters is zero, and the product rule leaves us with the gradient of the trajectory distribution.

At this point, the aforementioned log-likelihood ratio trick comes into play. From calculus, we know:

$$\begin{aligned} \nabla_x \log(x) &= \frac{1}{x} \\ \nabla_x \log(f(x)) &= \frac{\nabla_x f(x)}{f(x)} \\ \nabla_x f(x) &= \nabla_x \left[\log(f(x))\right] f(x) \end{aligned} \tag{3.4}$$

So we can replace the gradient of the trajectory probability distribution with the product of the probability distribution and the gradient of the log probability distribution:

$$\nabla_{\boldsymbol{\theta}} P(\tau) = P(\tau) \nabla_{\boldsymbol{\theta}} \log P(\tau)$$

The gradient of the log trajectory distribution $\nabla_{\boldsymbol{\theta}} \log P(\tau)$ is already stated in Equation (3.2) and only consists of a sum of gradients of the log policy distribution:

$$\nabla_{\boldsymbol{\theta}} P(\tau) = P(\tau) \sum_{t=0}^{\infty} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid s_t)$$

This form of the gradient of the trajectory distribution can now be inserted back into Equation (3.3):

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \int_{\tau} \nabla_{\boldsymbol{\theta}} \left[P(\tau)\right] \, \mathbb{E}\left[G_0 \mid T = \tau\right] \, \mathrm{d}\tau \\ &= \int_{\tau} P(\tau) \sum_{t=0}^{\infty} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid s_t) \, \mathbb{E}\left[G_0 \mid T = \tau\right] \, \mathrm{d}\tau \end{aligned}$$

This now has the form of an expectation over trajectories:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathop{\mathbb{E}}_{\tau}\left[\sum_{t=0}^{\infty} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid s_t) \, \mathbb{E}\left[G_0 \mid T = \tau\right]\right]$$

This is the general form of the REINFORCE policy gradient estimator as described in [56].

Policy Gradient Theorem The policy gradient theorem described in [59] builds on the same roots as the REINFORCE policy gradient estimator but incorporates enhancements that improve its practicality. One main insight is that past rewards can not be influenced by the current action. This builds on what is referred to as

the "expected grad log lemma" in [50]:

$$\begin{aligned}
\int_x P_\theta(x)\mathrm{d}x &= 1 \\
\nabla_\theta \int_x P_\theta(x)\mathrm{d}x &= \nabla_\theta 1 = 0 \\
\int_x \nabla_\theta P_\theta(x)\mathrm{d}x &= 0 \\
\int_x P_\theta(x)\nabla_\theta \log P_\theta(x)\mathrm{d}x &= 0 \\
\mathbb{E}\left[\nabla_\theta \log P_\theta(x)\right] &= 0
\end{aligned} \tag{3.5}$$

The expected grad log lemma uses the log probability gradient trick from Equation (3.4). Next we can split the total return G_0 at timestep $t-1$ to get:

$$\begin{aligned}
G_0 &= G_{0:t-1} + G_t \\
G_{0:t-1} &= \sum_{k=0}^{t-1} R_k \\
\nabla_\theta J(\theta) &= \mathop{\mathbb{E}}_{\tau}\left[\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, \mathbb{E}\left[G_{0:t-1} + G_t \mid T = \tau\right]\right] \\
&= \mathop{\mathbb{E}}_{\tau}\left[\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, \left(\mathbb{E}\left[G_{0:t-1} \mid T = \tau\right] + \mathbb{E}\left[G_t \mid T = \tau\right]\right)\right]
\end{aligned}$$

With partial trajectories $\tau_{k:l} = \{s_k, a_k, s_{k+1}, a_{k+1}, \ldots, s_l, a_l\}$ (with $k, l \in \mathbb{N}_0, k <= l$ and the random variable T defined accordingly), we can express that $\left(G_{0:t-1} \perp\!\!\!\perp T_{t:\infty}\right) \mid T_{0:t-1}$ and $\left(G_t \perp\!\!\!\perp T_{0:t-1}\right) \mid T_{t:\infty}$ (past rewards are independent of future states and actions given the past states and actions, and vice versa for future rewards):

$$\begin{aligned}
\nabla_\theta J(\theta) &= \mathop{\mathbb{E}}_{\tau}\left[\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, \left(\mathbb{E}\left[G_{0:t-1} \mid T_{0:t-1} = \tau_{0:t-1}\right] + \mathbb{E}\left[G_t \mid T_{t:\infty} = \tau_{t:\infty}\right]\right)\right] \\
&= \underbrace{\mathop{\mathbb{E}}_{\tau}\left[\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, \mathbb{E}\left[G_{0:t-1} \mid T_{0:t-1} = \tau_{0:t-1}\right]\right]}_{= 0 \text{ (due to Equation (3.5))}} \\
&\quad + \mathop{\mathbb{E}}_{\tau}\left[\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, \mathbb{E}\left[G_t \mid T_{t:\infty} = \tau_{t:\infty}\right]\right] \\
&= \mathop{\mathbb{E}}_{\tau}\left[\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, \mathbb{E}\left[G_t \mid T_{t:\infty} = \tau_{t:\infty}\right]\right]
\end{aligned} \tag{3.6}$$

This form still takes into account the trajectory to go $\tau_{t:\infty}$ but to be able to train the parametric policy using gradient descent with mini-batches of independent and identically distributed (i.i.d.) samples, the goal is to express the gradient estimator in terms of single timestep samples. To accomplish this, we first show that the expected return-to-go given the trajectory actually takes the role of the state-action value function $Q(s, a)$ in

Equation (3.6):

$$
\begin{aligned}
\nabla_\theta J(\theta) &= \int_\tau P(\tau) \sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, \mathbb{E}\left[G_t \mid T_{t:\infty} = \tau_{t:\infty}\right] \mathrm{d}\tau \\
&= \sum_{t=0}^{\infty} \int_\tau P(s_0) \prod_{k=0}^{\infty} P(s_{k+1} \mid s_k, a_k) \pi(a_k \mid s_k) \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, \mathbb{E}\left[G_t \mid T_{t:\infty} = \tau_{t:\infty}\right] \mathrm{d}\tau \\
&= \sum_{t=0}^{\infty} \int_\tau P(s_0) \prod_{k=0}^{t-1} P(s_{k+1} \mid s_k, a_k) \pi(a_k \mid s_k) \prod_{k=t}^{\infty} P(s_{k+1} \mid s_k, a_k) \pi(a_k \mid s_k) \\
&\quad \cdot \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, \mathbb{E}\left[G_t \mid T_{t:\infty} = \tau_{t:\infty}\right] \mathrm{d}\tau \\
&= \sum_{t=0}^{\infty} \int_{\tau_{0:t}} \int_{\tau_{t+1:\infty}} P(s_0) \prod_{k=0}^{t-1} P(s_{k+1} \mid s_k, a_k) \pi(a_k \mid s_k) \prod_{k=t}^{\infty} P(s_{k+1} \mid s_k, a_k) \pi(a_k \mid s_k) \\
&\quad \cdot \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, \mathbb{E}\left[G_t \mid T_{t:\infty} = \tau_{t:\infty}\right] \mathrm{d}\tau_{t+1:\infty}\, \mathrm{d}\tau_{0:t} \\
&= \sum_{t=0}^{\infty} \int_{\tau_{0:t}} P(s_0) \prod_{k=0}^{t-1} P(s_{k+1} \mid s_k, a_k) \pi(a_k \mid s_k) \nabla_\theta \log \pi_\theta(a_t \mid s_t) \\
&\quad \cdot \underbrace{\int_{\tau_{t+1:\infty}} \prod_{k=t}^{\infty} P(s_{k+1} \mid s_k, a_k) \pi(a_k \mid s_k)\, \mathbb{E}\left[G_t \mid T_{t:\infty} = \tau_{t:\infty}\right] \mathrm{d}\tau_{t+1:\infty}}_{= \pi(a_t \mid s_t) Q(s_t, a_t)} \mathrm{d}\tau_{0:t}
\end{aligned} \tag{3.7}
$$

In the following, we will show how the part emphasized by the brace ends up being the compact term $\pi(a_t \mid s_t) Q(s_t, a_t)$:

$$
\begin{aligned}
&\int_{\tau_{t+1:\infty}} \prod_{k=t}^{\infty} P(s_{k+1} \mid s_k, a_k) \pi(a_k \mid s_k)\, \mathbb{E}\left[G_t \mid T_{t:\infty} = \tau_{t:\infty}\right] \mathrm{d}\tau_{t+1:\infty} \\
&= \int_{\tau_{t+1:\infty}} P(s_{t+1} \mid s_t, a_t) \pi(a_t \mid s_t) \prod_{k=t+1}^{\infty} P(s_{k+1} \mid s_k, a_k) \pi(a_k \mid s_k)\, \mathbb{E}\left[G_t \mid T_{t:\infty} = \tau_{t:\infty}\right] \mathrm{d}\tau_{t+1:\infty} \\
&= \pi(a_t \mid s_t) \int_{\tau_{t+1:\infty}} P(s_{t+1} \mid s_t, a_t) \prod_{k=t+1}^{\infty} P(s_{k+1} \mid s_k, a_k) \pi(a_k \mid s_k)\, \mathbb{E}\left[G_t \mid T_{t:\infty} = \tau_{t:\infty}\right] \mathrm{d}\tau_{t+1:\infty} \\
&= \pi(a_t \mid s_t) \int_{\tau_{t+1:\infty}} P\left(\tau_{t+1:\infty} \mid s_t, a_t\right) \mathbb{E}\left[G_t \mid T_{t:\infty} = \tau_{t:\infty}\right] \mathrm{d}\tau_{t+1:\infty}
\end{aligned} \tag{3.8}
$$

The expected return-to-go is still dependent on the current state s_t and a_t which are not covered by the integration over the trajectory $\tau_{t+1:\infty}$. To marginalize over the trajectory $\tau_{t+1:\infty}$, we can split up the trajectory

in the condition of the expected return:

$$\begin{aligned}
&\mathbb{E}\left[G_t \mid T_{t:\infty} = \tau_{t:\infty}\right] \\
&= \int_{g_t} P\left(G_t = g_t \mid T_{t:\infty} = \tau_{t:\infty}\right) g_t \, \mathrm{d}g_t \\
&= \int_{g_t} P\left(G_t = g_t \mid a_t, s_t, T_{t+1:\infty} = \tau_{t+1:\infty}\right) g_t \, \mathrm{d}g_t
\end{aligned}$$

We can now substitute this form of the expected return-to-go back into Equation (3.8):

$$\begin{aligned}
&\pi\left(a_t \mid s_t\right) \int_{\tau_{t+1:\infty}} P\left(\tau_{t+1:\infty} \mid s_t, a_t\right) \int_{g_t} P\left(G_t = g_t \mid a_t, s_t, T_{t+1:\infty} = \tau_{t+1:\infty}\right) g_t \, \mathrm{d}g_t \, \mathrm{d}\tau_{t+1:\infty} \\
&= \pi\left(a_t \mid s_t\right) \int_{g_t} \int_{\tau_{t+1:\infty}} P\left(\tau_{t+1:\infty} \mid s_t, a_t\right) P\left(G_t = g_t \mid a_t, s_t, T_{t+1:\infty} = \tau_{t+1:\infty}\right) \, \mathrm{d}\tau_{t+1:\infty} \, g_t \, \mathrm{d}g_t \\
&= \pi\left(a_t \mid s_t\right) \int_{g_t} P\left(G_t = g_t \mid a_t, s_t\right) \, g_t \, \mathrm{d}g_t \\
&= \pi\left(a_t \mid s_t\right) \underbrace{\mathbb{E}\left[G_t \mid a_t, s_t\right]}_{:= Q(s,a)} \\
&= \pi\left(a_t \mid s_t\right) Q\left(s_t, a_t\right)
\end{aligned}$$

This term can now be substituted back into the policy gradient estimator in Equation (3.7):

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \sum_{t=0}^{\infty} \int_{\tau_{0:t}} P(s_0) \prod_{k=0}^{t-1} P(s_{k+1} \mid s_k, a_k) \pi(a_k \mid s_k) \pi\left(a_t \mid s_t\right) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid s_t) Q(s_t, a_t) \mathrm{d}\tau_{0:t} \\
&= \sum_{t=0}^{\infty} \int_{\tau_{0:t-1}} \int_{a_t, s_t} P(s_0) \prod_{k=0}^{t-1} P(s_{k+1} \mid s_k, a_k) \pi(a_k \mid s_k) \pi\left(a_t \mid s_t\right) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid s_t) Q(s_t, a_t) \mathrm{d}a_t \, \mathrm{d}s_t \, \mathrm{d}\tau_{0:t-1} \\
&= \sum_{t=0}^{\infty} \int_{a_t, s_t} \int_{\tau_{0:t-1}} P(s_0) \prod_{k=0}^{t-1} P(s_{k+1} \mid s_k, a_k) \pi(a_k \mid s_k) \pi\left(a_t \mid s_t\right) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid s_t) Q(s_t, a_t) d\tau_{0:t-1} \, \mathrm{d}a_t \, \mathrm{d}s_t \\
&= \sum_{t=0}^{\infty} \int_{a_t, s_t} \underbrace{\int_{\tau_{0:t-1}} P(s_0) \prod_{k=0}^{t-1} P(s_{k+1} \mid s_k, a_k) \pi(a_k \mid s_k) d\tau_{0:t-1}}_{P(S_t = s_t)} \; \pi\left(a_t \mid s_t\right) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid s_t) Q(s_t, a_t) \mathrm{d}a_t \, \mathrm{d}s_t \\
&= \sum_{t=0}^{\infty} \int_{a_t, s_t} P(S_t = s_t) \, \pi\left(a_t \mid s_t\right) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid s_t) Q(s_t, a_t) \mathrm{d}a_t \, \mathrm{d}s_t
\end{aligned}$$

Based on this we can rename s_t and a_t to s and a

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \sum_{t=0}^{\infty} \int_{a,s} P(S_t = s) \, \pi\left(a \mid s\right) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid s) Q(s, a) \mathrm{d}a \, \mathrm{d}s$$

In this formulation, the integral integrates over all states s including the terminal, absorbing state $\overset{\infty}{s}$. By definition, we know that the action-value function is zero in the terminal state for all actions:

$$Q\left(\overset{\infty}{s}, a\right) = 0 \quad \forall a \in \mathcal{A}$$

Because the value of the terminal state is always zero, we can exclude it from the integral:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \sum_{t=0}^{\infty} \int_{a,s \in S_{nt}} P(S_t = s)\, \pi(a \mid s)\, \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid s) Q(s,a) \mathrm{d}a\, \mathrm{d}s \tag{3.9}$$

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \int_{a,s \in S_{nt}} \underbrace{\sum_{t=0}^{\infty} \left[P(S_t = s)\right]}_{:= \eta(s) \;\; \forall s \in S_{nt}} \pi(a \mid s)\, \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid s) Q(s,a) \mathrm{d}a\, \mathrm{d}s \tag{3.10}$$

Where $\eta(s)$ is the visitation density. We would like to write the integral as an expectation over trajectory samples from the current policy again. The integral already has the form of a conditional expectation over the actions a (given a state s); to complete it, we would also like to write the state visitation density as a probability distribution. This can be accomplished by normalizing $\eta(s)$ to sum to one (renaming the integration variable $s^+ := s$ in the normalization term to prevent confusion with the parameter s):

$$d(s) := \frac{\eta(s)}{\int_{s^+ \in S_{nt}} \eta(s^+) \mathrm{d}s^+} \quad \forall s \in S_{nt} \tag{3.11}$$

Where $d(s)$ is the probability density of a state occurring in a trajectory under a particular policy. This form is common in the literature (e.g. [47]). It can be justified that $d(s)$ represents the actual probability density for a state to occur in a trajectory by viewing $P(S_t = s)$ as a conditional distribution on the timestep $P(s \mid t)$. Viewing it as a conditional probability distribution (on the timestep t) is the right way to look at it because integrating over the states yields one:

$$\int_s P(S_t = s) \mathrm{d}s = \int_s P(s \mid t) \mathrm{d}s = 1$$

Using the conditional probability distribution, we can write the marginal distribution over states as follows:

$$P(s) = \sum_{t=0}^{\infty} P(s \mid t) \underbrace{P(t)}_{c}$$

To be able to write this in terms of Equation (3.11), $P(t)$ needs to be constant. This is a reasonable assumption because we defined the episodic case to have an infinite number of steps which in theory are all visited each episode. Practically the probability of ending up in the absorbing, terminal state $\overset{\infty}{s}$ goes to one in the limit of t to ∞ (can be verified using a geometric series).

With $P(s) = c$:

$$\begin{aligned} P(s) &= \sum_{t=0}^{\infty} P(s \mid t) c \\ &= c \underbrace{\sum_{t=0}^{\infty} P(s \mid t)}_{\eta(s) \text{ (according to the definition in Equation (3.10))}} \\ &= c\, \eta(s) \end{aligned} \tag{3.12}$$

Integrating the state distribution over the states has to result in a probability mass of one. In the estimation of the policy gradient in Equation (3.9), we are only integrating over non-terminal states. To be able to rewrite this as an expectation over non-terminal states, we also want the state distribution $P(s)$ to be defined over non-terminal states $s \in \mathcal{S}_{nt}$:

$$\begin{aligned} \mathcal{S}_{nt} &:= \mathcal{S} \setminus \{ \overset{\infty}{s} \} \\ \int_{s \in \mathcal{S}_{nt}} P(s)\, \mathrm{d}s &= 1 \\ \int_{s \in \mathcal{S}_{nt}} c\ \ \eta(s)\, \mathrm{d}s &= 1 \\ \int_{s \in \mathcal{S}_{nt}} \eta(s)\, \mathrm{d}s &= \frac{1}{c} \\ c &= \frac{1}{\int_{s \in \mathcal{S}_{nt}} \eta(s)\, \mathrm{d}s} \end{aligned} \tag{3.13}$$

Substituting Equation (3.13) back into Equation (3.12) while renaming the integration variable $s^+ := s$ to prevent confusion (like in Equation (3.11)):

$$P(s) = c\, \eta(s) = \frac{\eta(s)}{\int_{s^+ \in \mathcal{S}_{nt}} \eta(s^+)\, \mathrm{d}s^+} = d(s)$$

With this, we showed that the common form used as the state distribution $d(s)$ as defined in Equation (3.11) is actually the marginal state distribution $P(s)$. So using this normalization, $d(s)$ is the visitation probability density of a state s (under a particular policy).

To substitute the state visitation density distribution back into the gradient estimator, we can start from Equation (3.10) and multiply the right side by one:

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \int_{a,s \in \mathcal{S}_{nt}} \eta(s)\ \ \pi(a \mid s)\, \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid s) Q(s,a) \mathrm{d}a\, \mathrm{d}s \\ &= \frac{\int_{s^+ \in \mathcal{S}_{nt}} \eta\left(s^+\right) \mathrm{d}s^+}{\int_{s^+ \in \mathcal{S}_{nt}} \eta(s^+)\, \mathrm{d}s^+} \int_{a,s \in \mathcal{S}_{nt}} \eta(s)\ \ \pi(a \mid s)\, \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid s) Q(s,a) \mathrm{d}a\, \mathrm{d}s \end{aligned}$$

Because the normalization integral is independent of the integration variables of the policy gradient estimator, we can pull the denominator into the integral, resulting in the state visitation probability density $d(s)$:

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \int\limits_{s^+ \in S_{nt}} \eta\left(s^+\right) \mathrm{d}s^+ \int\limits_{a,s \in S_{nt}} \frac{\eta(s)}{\int\limits_{s^+ \in S_{nt}} \eta\left(s^+\right) \mathrm{d}s^+} \pi(a \mid s) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid s) Q(s,a) \mathrm{d}a \, \mathrm{d}s \\ &= \int\limits_{s^+ \in S_{nt}} \eta\left(s^+\right) \mathrm{d}s^+ \int\limits_{a,s \in S_{nt}} d(s) \; \pi(a \mid s) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid s) Q(s,a) \mathrm{d}a \, \mathrm{d}s\end{aligned}$$

For a particular policy, the normalization integral is a constant that embodies the expected episode length c_l:

$$\int\limits_{s^+ \in S_{nt}} \eta\left(s^+\right) \mathrm{d}s^+ = c_l$$

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = c_l \int\limits_{a,s \in S_{nt}} d(s) \; \pi(a \mid s) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid s) Q(s,a) \mathrm{d}a \, \mathrm{d}s$$

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \propto \int\limits_{a,s \in S_{nt}} d(s) \; \pi(a \mid s) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid s) Q(s,a) \mathrm{d}a \, \mathrm{d}s$$

The normalization constant c_l can be absorbed into the learning rate of the particular gradient ascent algorithm that uses this policy gradient estimator.

We can now finally rewrite the integral as an expectation which can be approximated by samples of single timesteps:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \propto \mathop{\mathbb{E}}_{a,s \in S_{nt}} \left[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid s) Q(s,a)\right] \tag{3.14}$$

It is important to express the gradient estimator in terms of single timesteps to be able to apply mini-batch gradient ascent, where a set of N i.i.d. samples is used to estimate the (stochastic) gradient:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \propto \frac{1}{N} \sum_{n=1}^{N} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_n \mid s_n) Q(s_n, a_n) \tag{3.15}$$

Where the index n of the state-action tuples $\left(s_n, a_n\right)$ is not characterizing the timestep but just distinct samples that might also be part of different rollouts as long as the rollouts have been done under the same, current policy. This is not possible using the REINFORCE estimator, which is based on whole episodes/trajectories and hence a big advantage of the policy gradient theorem.

Actor-Critic Algorithms When using the gradient estimator as shown in Equation (3.15), in practice, we use two approximations. First, we take a state-action sample from some trajectory generated by following the current policy. From this, we can determine the gradient of the action probability given the state (policy $\pi_{\boldsymbol{\theta}}(a \mid s)$). This gradient of the policy is a vector that points into the direction in which the parameters can be changed to increase the probability of the sampled action. This proposition only holds strongly for either a linear policy or infinitesimal small increments. In practice, we use, e.g., a neural network (which outputs sufficient statistics for some family of distributions) which allows taking bigger than infinitesimal steps due to the assumption of a certain degree of smoothness.

The second approximation is concerning the expected return-to-go $Q(s, a)$. For this quantity, we usually only get a single sample which is the sum of future rewards starting from the timestep that the previously mentioned state-action sample has been taken from. This estimate consisting of a single sample is referred to as a Monte-Carlo estimate. Using Monte-Carlo estimates is one way of approaching the (temporal) credit assignment problem in RL. The downside of using Monte-Carlo samples of the return in the gradient estimator is that they introduce a high variance. A simple example to demonstrate the high variance is an agent that executes two independent rollouts and at some point experiences the same state and action in both trajectories. Due to stochasticity in the transition dynamics, the policy, and, to a lesser degree, also in the reward model, the states and actions following this common state might drastically differ.

An illustrative example would be that the agent is very close to, e.g., a cliff in this common state, and in one rollout, he falls off this cliff. The Monte-Carlo estimates of the expected return of these two rollouts drastically differ, albeit the true expected return being defined to be consistent across the two rollouts. The variance of the Monte-Carlo estimate for the expected return also grows with the length of an episode because the cause and effect between the first actions and the final return become increasingly disconnected ([57] shows quadratic growth of the variance with respect to episode-length under certain assumptions).

In the case that we can reset the environment to arbitrary states, we could obtain additional samples for the expected return-to-go, starting from a certain state-action sample $Q(s, a)$ by following the current policy. Using more samples would decrease the variance according to the law of large numbers. However, resetting the environment to a desired state neither fits into the general RL framework nor is very popular in practice.

The variance over the state-action samples can likewise be reduced by using more samples which in the case of training neural networks is usually done by using mini-batch SGD, where a mini-batch consists of a certain number of samples to reduce the variance to some degree. A certain level of remaining variance has been found to foster learning which is why usually, the batches do not contain all available samples at once.

Because we can tune the variance of the estimation of the outer expectation by modifying the batch size, for the policy gradient theorem-based gradient estimator in Equation (3.14), we are left with the undesired high variance of the state-action value estimate when approximating it by Monte-Carlo sampling. Two common approaches to decrease this variance are the subtraction of a baseline from the state-action value estimate and learning an estimator for the state-action value function. This estimator of the state-action value function is commonly referred to as the critic, while the policy is also referred to as the actor.

Policy Gradient Estimation Using a Baseline Both REINFORCE, and the policy gradient theorem-based gradient estimator share the feature that it is possible to subtract a baseline $b(s)$ which is dependent on the state, from the action-value function. In the following, we will only show for the policy gradient theorem-based

gradient estimator that it remains unbiased even when subtracting a baseline dependent on the state:

$$\begin{aligned}
\nabla_\theta J(\theta) &\propto \mathop{\mathbb{E}}_{a,s\in S_{nt}}\left[\nabla_\theta \log \pi_\theta(a \mid s)\left(Q(s,a) - b(s)\right)\right] \\
&\propto \mathop{\mathbb{E}}_{s\in S_{nt}}\left[\mathop{\mathbb{E}}_{a}\left[\nabla_\theta \log \pi_\theta(a \mid s)\left(Q(s,a) - b(s)\right)\right]\right] \\
&\propto \mathop{\mathbb{E}}_{s\in S_{nt}}\left[\mathop{\mathbb{E}}_{a}\left[\nabla_\theta \log \pi_\theta(a \mid s) Q(s,a)\right] - \mathop{\mathbb{E}}_{a}\left[\nabla_\theta \log \pi_\theta(a \mid s) b(s)\right]\right] \\
&\propto \mathop{\mathbb{E}}_{s\in S_{nt}}\left[\mathop{\mathbb{E}}_{a}\left[\nabla_\theta \log \pi_\theta(a \mid s) Q(s,a)\right] - b(s) \underbrace{\mathop{\mathbb{E}}_{a}\left[\nabla_\theta \log \pi_\theta(a \mid s)\right]}_{=0}\right] \\
&\propto \mathop{\mathbb{E}}_{s\in S_{nt}}\left[\mathop{\mathbb{E}}_{a}\left[\nabla_\theta \log \pi_\theta(a \mid s) Q(s,a)\right]\right] \\
&\propto \mathop{\mathbb{E}}_{a,s\in S_{nt}}\left[\nabla_\theta \log \pi_\theta(a \mid s) Q(s,a)\right]
\end{aligned}$$

The latter expectation is equal to zero according to the "expected grad log lemma" in Equation (3.5). Using this insight, the baseline is multiplied by zero, and we have established that subtracting a baseline does not introduce bias to the policy gradient theorem-based gradient estimator:

$$\mathop{\mathbb{E}}_{a,s\in S_{nt}}\left[\nabla_\theta \log \pi_\theta(a \mid s)\left(Q(s,a) - b(s)\right)\right] = \mathop{\mathbb{E}}_{a,s\in S_{nt}}\left[\nabla_\theta \log \pi_\theta(a \mid s) Q(s,a)\right]$$

A common choice for the baseline is the state value function, representing the expected return-to-go from state s_t:

$$\begin{aligned}
b\left(s_t\right) &:= V\left(s_t\right) \\
&= E\left[G_t \mid s_t\right] = E\left[R_t \mid s_t\right] + V\left(s_{t+1}\right) \\
&= E\left[G_t \mid s_t\right] = \mathop{E}_{a_t\sim\pi}\left[Q\left(s_t, a_t\right) \mid s_t\right]
\end{aligned}$$

The state value function $V(s)$ can be written either in a recursive form in terms of the next state or as an expectation over the state-action values weighted by the action probabilities in the given state s_t governed by the policy. Using the state value function as a baseline leads to the difference $A(s,a)$ better known as the advantage function:

$$\begin{aligned}
A(s,a) &= Q(s,a) - V(s) \\
\nabla_\theta J(\theta) &\propto \mathop{\mathbb{E}}_{a,s\in S_{nt}}\left[\nabla_\theta \log \pi_\theta(a \mid s)\left(Q(s,a) - V(s)\right)\right] \\
&\propto \mathop{\mathbb{E}}_{a,s\in S_{nt}}\left[\nabla_\theta \log \pi_\theta(a \mid s) A(s,a)\right]
\end{aligned}$$

The usage of the advantage function leads to the intuitive behavior of the gradient ascent increasing the probabilities of actions that have a higher expected return than the expected return of the current policy from a particular state. Moreover, in the opposite manner, it allows decreasing the probabilities of actions that lead to a lower expected return than the return expected from the current policy.

Implementation of the Advantage Estimation Usually, the advantage function $A(s,a)$ is not estimated directly but implemented using a state value estimator :

$$\begin{aligned}
A(s,a) &= Q(s,a) - V(s) \\
&= \mathbb{E}\left[V(S') + R \mid s, a\right] - V(s) \\
&= \underbrace{\mathbb{E}\left[V(S') \mid s, a\right]}_{\sim V(s')} + \underbrace{\mathbb{E}\left[R \mid s, a\right]}_{\sim r} - V(s)
\end{aligned}$$

The expectations of the state-value of the next state and the reward for the current transition can be approximated without bias by using samples. Depending on the entropy of the distribution over the next states and rewards given a state and action, the variance of this estimate can be high when few samples are used. In practice (especially in the case of continuous control) for a state-action tuple (s, a), only one observed next state s' and reward r are available. This may lead to a high variance. As previously described for the Monte-Carlo estimate of the return-to-go, for the estimation of the advantage, the variance could be reduced by resetting the environment to the particular state s and taking action a multiple times to observe different next states s' and rewards r. However, resetting the environment to a particular state is uncommon and assumed to not be possible in general, as stated earlier. Because of this restriction, most RL algorithms that build upon the policy gradient theorem estimate this by single samples of state, action, reward, next state tuples. Also, in practice, especially in the case of continuous control problems, the reward function is often deterministic, and the duration of a timestep dt is short. Using short timesteps leads to a high correlation between the current and next state and also results in a low variance.

To estimate the advantage in the described way, we still need to be able to calculate the expected state values $V(s)$. In the case of actor-critic algorithms, the state value function is referred to as the critic. In the case of continuous control problems, it is infeasible to use tabular methods to learn a state value function due to the infinite number of states. Discretization of the state space can work in a limited set of simple problems but ultimately faces the curse of dimensionality when applied to higher dimensional state spaces. Because of this, for continuous control problems, the critic is implemented using function approximation. The term deep RL refers to the case where the actor, as well as the critic, is implemented by neural networks, which are known to be universal function approximators in the limit of infinite width (first shown in [60]). A simple intuition for why infinitely wide neural networks are universal function approximators is that one (or a set of) neurons in a layer define a piece of a piecewise-defined function. In the limit of infinite pieces, the function approximator in the form of the piecewise-defined function can become arbitrarily close to the original function.

In the case of the state value function $V(s)$, we want the neural network-based approximator $\hat{V}_{\theta_V}(s)$, parameterized by θ_V, to be as close as possible to the actual expected value, which can be achieved by, e.g., minimizing the expected squared error:

$$J_c\left(\theta_V\right) = \mathbb{E}\left[\left(V(s) - \hat{V}_{\theta_V}(s)\right)^2\right] \tag{3.16}$$

Where $V(s)$ is subject to the (undiscounted) Bellman equation based on [61]:

$$\begin{aligned} V(s) &= \mathbb{E}\left[V(S') + R \mid s\right] \\ &= \mathbb{E}\left[V(S') \mid s\right] + \mathbb{E}\left[R \mid s\right] \\ &= \int_{a,s'} P\left(s' \mid s, a\right) \pi(a \mid s)\, V(s')\mathrm{d}a\mathrm{d}s + \int_{a,r} \pi(a \mid s)\, R(s, a, r)\mathrm{d}a\mathrm{d}r \end{aligned}$$

Like in the case of the policy gradient estimator, we can also estimate both expectations by using samples for action a, reward r and next state s' for a given state s:

$$V(s) \approx V\left(s'\right) + r$$

Replacing this back into Equation (3.16) yields the following, temporally unrolled, unbiased estimator for the critic loss:

$$J_c\left(\theta_V\right) = \mathbb{E}\left[\left(V\left(s'\right) + r - \hat{V}_{\theta_V}(s)\right)^2\right]$$

Because the critic loss is still dependent on the unknown, true expected return-to-go $V\left(s'\right)$, a trick called bootstrapping is used. With bootstrapping, the approximated function itself is used as the training target:

$$J_c\left(\theta_V\right) = \mathbb{E}\left[\left(\underbrace{\hat{V}_{\theta_V}\left(s'\right)}_{\text{const wrt } \theta_V} + r - \hat{V}_{\theta_V}\left(s\right)\right)^2\right] \tag{3.17}$$

This form only contains tractable terms, and the loss can be minimized using gradient descent with respect to the parameters of the critic θ_V. The approximated state value for the next state is often treated to be constant with respect to the parameters θ_V, as it is a replacement for the true state value function, which is also independent of the parameters. In this way, the gradient is blocked in the case of the value of the next state, and the gradient descent only happens in terms of the current state value prediction. Blocking the gradient is known to improve the stability of the training procedure and is often combined with other measures. One common way to further stabilize training is the usage of a separate target value function for the estimation of the next state's value. This target network is usually a delayed version of the current critic and either updated in a certain interval (DQN [2]) or by using Polyak averaging (SAC [62]).

This update rule is generally referred to as Temporal Difference (TD) learning, proposed by Sutton in [63]. As stated earlier, it is based on unrolling the Bellman equation for one step, and it can be generalized by further unrolling the estimation, incorporating more future steps, which increases the variance and reduces the bias until it is equivalent to the Monte-Carlo estimate in the limit (with no bias but usually high variance).

A related approach is to use a discounting scheme to determine the so-called eligibility of an action to future rewards. This is referred to as the TD(λ) algorithm and is computationally preferable to the algorithm with a fixed horizon because the eligibility can be transferred backwards in time by discounting and does not have to be recomputed for each step. Closely related to TD(λ) is the Generalized Advantage Estimation (GAE) algorithm proposed in [13]. The GAE takes the main idea of TD(λ) and directly applies it to estimate the advantage instead of the state value function. In combination with Proximal Policy Optimization (PPO), it is one of the most commonly used RL algorithms.

Policy Gradient With Function Approximation Because we want to use a universal function approximator in the approximation of the critic (Equation (3.17)) to be able to represent complex, non-linear concepts emerging from acting in the given environment, the best result we can usually expect from the regression using gradient descent is convergence to a local optimum of the loss function. Considering this inaccuracy, it is not directly apparent that it is possible to replace the theoretical true advantage $A\left(s, a\right)$ with an estimate of the advantage based on the approximated state value function:

$$\hat{A}\left(s, a\right) = \underbrace{\mathbb{E}\left[\hat{V}(S') \mid s, a\right]}_{\sim \hat{V}(s')} + \underbrace{\mathbb{E}\left[R \mid s, a\right]}_{\sim r} - \hat{V}\left(s\right)$$

In the original paper introducing the policy gradient theorem [59], the authors also establish conditions under which the usage of a function approximation-based critic leads to correct policy gradient estimates. Their proof is made in terms of the state-action value function, and their central condition for the state-action value function approximation $\hat{Q}_{\theta_Q}\left(s, a\right)$ is:

$$\nabla_{\theta_Q} \hat{Q}_{\theta_Q}\left(s, a\right) = \frac{\nabla_\theta \pi_\theta\left(a \mid s\right)}{\pi_\theta\left(a \mid s\right)} = \nabla_\theta \log \pi_\theta\left(a \mid s\right)$$

This condition gives rise to a form where both the state-action value function and the policy are an inner product with a common feature vector $\phi(s,a)$:

$$\begin{aligned} \hat{Q}_{\theta_Q}(s,a) = \phi(s,a)\,\theta_Q &\rightarrow \nabla_{\theta_Q} \hat{Q}_{\theta_Q}(s,a) \quad = \phi(s,a) \\ \log \pi_\theta(a \mid s) = \phi(s,a)\,\theta &\rightarrow \nabla_\theta \log \pi_\theta(a \mid s) = \phi(s,a) \end{aligned}$$

In [57], the authors note that the linear form of the state-action value approximation has a zero mean w.r.t. the action distribution. In general, the action-value function can not be assumed to be mean zero w.r.t. to the actions, which leads to the interpretation of the approximate state-action value function actually resembling an advantage function [57].

In practice, the compatibility requirements between the actor and critic are widely disregarded. While there are compatible approaches like GProp proposed in [64], the best results in common RL benchmarks (and also the biggest popularity in downstream usage), have been attained by algorithms disregarding these compatibility requirements. Two particular, incompatible algorithms will be described in the following.

3.1.3. Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) is a widely used policy gradient algorithm proposed in [5]. It can be seen as the successor to Trust Region Policy Optimization (TRPO) [4]. TRPO is motivated by the insight that in deep learning in general and in deep RL in particular, too big gradient steps can destabilize the training process. This is true for deep learning in particular, as a gradient step can be seen as the optimization of a locally linear approximation, which is usually justified by smoothness assumptions of the class of functions. In the case of on-policy RL algorithms, this poses a particular issue because not only might the policy change more than would be justified by the smoothness of the function but changing the policy also changes the data distribution. This can lead to a reinforcing feedback loop of bad gradient updates leading to worse behavior and worse behavior leading to the data distribution used for training to only contain samples of inferior trajectories. TRPO, as its name suggests, uses an update rule that bounds the divergence of the old and new policy distribution. This turned out to be very beneficial for the robustness of the training process but also comes with higher computational costs (as it resembles second-order optimization methods) and a complex implementation.

PPO builds on the same intuition but simply clips the probability ratio between the old and new policy during the update to restrict the shift in the policy distribution. We usually observed single-digit percentages of samples leading to clipping and, therefore, deactivation of the gradient. PPO is usually considered to be an on-policy RL algorithm, but because of the restriction of the distribution shift in the policy, it is possible to execute rollouts on possibly many parallel workers and then do many gradient steps using the collected data. After the first gradient step/batch, the collected data would theoretically be invalid to estimate the policy gradient because the policy used to collect it deviates from the updated one. In the case of PPO (and TRPO), it turns out that even after many gradient steps and even multiple epochs of training on the old data, the training process remains stable.

Despite the possibility of reusing data by training across multiple epochs, PPO is still considered a very sample inefficient algorithm. We consider PPO to be a good choice for our approach because it is widely regarded as a robust and comparatively simple algorithm. We do not consider the sample inefficiency of PPO an issue because we are solely training in simulation and because we assume that we are able to build a very fast simulator for our applications.

3.1.4. Soft Actor-Critic (SAC)

In addition to the policy gradient algorithms based on REINFORCE and the policy gradient theorem, recently also a different strand of algorithms based on deterministic policy gradients gained in popularity. These algorithms are based on the Deterministic Policy Gradient (DPG) proposed in [65], where the authors derive a policy gradient algorithm using a deterministic policy and claim that prior to their work, it was believed that the deterministic policy gradient did not exist. The fundamental difference between DPG and policy gradient theorem-based algorithms is that DPG algorithms exploit that the critic is usually implemented using a neural network. In the case of implementing the critic using a neural network, gradients w.r.t. to the inputs (the actions in particular) can be calculated easily using backpropagation. Using a deterministic, parametric policy $\mu_\theta(s)$, the RL objective (expected return) can be written as follows:

$$
\begin{aligned}
J(\theta) &= \mathop{\mathbb{E}}_{s \in S_{nt}} \left[Q\left(s, \mu_\theta(s)\right) \right] \\
\nabla_\theta J(\theta) &= \mathop{\mathbb{E}}_{s \in S_{nt}} \left[\nabla_\theta Q\left(s, \mu_\theta(s)\right) \right] \\
\nabla_\theta J(\theta) &= \mathop{\mathbb{E}}_{s \in S_{nt}} \left[\nabla_\theta \mu_\theta(s) \, \nabla_a Q(s,a) \big|_{a=\mu_\theta(s)} \right]
\end{aligned}
$$

The deterministic policy gradient is the result of pulling the gradient into the expectation and applying the chain-rule.

To be able to use a function approximator $\hat{Q}_\psi(s,a)$ for the state-action value function $Q(s,a)$, Silver et al. make extensive use of the compatible function approximation theorem, that was first introduced by Sutton alongside the policy gradient theorem in [59]. As stated earlier, the compatible function approximation theorem establishes compatibility conditions between the actor and critic under which the true policy gradient can be estimated without bias when replacing the action-value function with an approximation (which is required to have converged to a local optimum). Silver et al. adapt the compatibility conditions to the DPG and propose a compatible algorithm based on a deterministic policy.

In [65], the authors were able to show promising results using deterministic policy gradients, but it was only until the authors of [11] resolved stability issues by using additional, delayed target networks for the actor and critic that deterministic policy gradients were shown to be able to achieve state-of-the-art results in continuous control tasks. The algorithm proposed in [11] is called Deep DPG (Deep Deterministic Policy Gradient (DDPG)), and in comparison to DPG, like most popular policy gradient algorithms, it does not guarantee compatibility between actor and critic. DDPG is also the basis for more recent algorithms like Twin Delayed Deep Deterministic Policy Gradient (TD3) and SAC. A significant advantage this family of algorithms has over conventional policy gradient algorithms is that they can use off-policy data (data not sampled from the current policy) to train both actor and critic.

Soft Actor-Critic (SAC) is a policy gradient algorithm that technically builds on the idea of DDPG but uses a soft (stochastic) policy again. SAC has initially been proposed in [14], followed by an improved version of the algorithm in [62], and results of applying SAC to quadrupedal locomotion by directly using a real robot in [66]. In comparison to other policy gradient methods, SAC optimizes a slightly different objective using an additional entropy bonus term and thus also fits into the maximum entropy RL framework. Instead of using a REINFORCE or policy gradient theorem type estimator to optimize the Kulback-Leibler (KL) divergence between the policy and the implied optimal policy derived from the action-value function, the authors opt to exploit the easy differentiability of the neural network-based critic wrt. the action. To enable taking the gradient through the stochastic actions, they limit the policy network to be a deterministic function that outputs sufficient statistics to parameterize a family of distributions over actions. In the case of the family of

Gaussian distributions, the reparameterization trick known from Variational Autoencoders (VAEs) [67] can be used to enable backpropagation through the sampling process. The reparameterization trick replaces the sampling from a Gaussian distribution with biasing and rescaling standard normal noise. In this way, the noise can be treated as an input to the policy and does not obstruct the flow of the gradient.

In the subsequent paper [62], the authors improve the initial SAC algorithm in multiple ways. Most notable is an automatic tuning procedure for the weight of the entropy bonus (also called temperature) in the reward function. This simplifies the hyperparameter tuning as only an initial temperature and a step size for the automatic tuning need to be chosen. According to the authors, these hyperparameters of the tuning algorithm for the temperature parameter generalize better than a single temperature value, in which case the optimal setting is often task-dependent. In [66], the authors show that SAC can be used to train locomotion policies directly on a quadrupedal robot. Because of the good sample efficiency of the SAC algorithm (due to the possibility to learn from off-policy samples), the authors are able to observe locomotion policies after two hours of training on the real robot without pretraining.

3.2. Multi-Task Reinforcement Learning

Multi-task RL extends the scope of RL from a single task/environment to multiple tasks. Instead of training a particular agent for each environment, a single agent is trained to act in different environments. As each environment is fully defined by its own MDP tuple $(S_z, \mathcal{A}_z, P_z, \mathcal{R}_z, R_z, d_{0z}, \gamma_z)$ where $z \in \mathcal{Z}$ is a context variable characterizing the particular environment and $\mathcal{Z}$ is the set of context variables corresponding to environments in a particular multi-task problem.

The context variable z can be implemented in different forms, e.g., as a one-hot vector, as an integral identifier, or a (potentially real-valued) feature vector. The real-valued feature vector can, for example, contain information about the modes of variation in the task distribution. If the distribution over tasks is created by, e.g., perturbing a set of parameters (masses, friction terms, available torques, etc.), then the values of the varying parameters can be collected into a feature vector z that can be used to distinguish a particular environment from the others. Usage of a task distribution that is created from perturbing real-valued dynamics parameters also gives rise to multi-task learning with infinitely many tasks and can also be straightforwardly combined with variations in categorical parameters.

Independent of the representation of z, the collection of environments gives rise to a probability distribution over MDPs, which can be rewritten as a probability distribution over context variables $P(Z = z) = P(z)$. Where each context variable z represents a particular MDP and with Z being the random (context) variable representing the distribution over MDPs.

Figure 3.3 shows how the distribution over MDPs is governed by the context variable Z. The Bayesian network is based on the Bayesian network for a single MDP but now extruded into a new dimension consisting of the different domain parameters. The outer plate represents the domain d and only contains the context variable Z. For a single domain d, Z is sampled only once and can be valid for multiple episodes e. The context variable can be interpreted as a selector for the MDP $(S_z, \mathcal{A}_z, P_z, \mathcal{R}_z, R_z, d_{0z}, \gamma_z)$ and hence can influence all the distributions (including the policy). The dependence of all (conditional) distributions in the graph on the context variable Z is chosen so we maintain the theoretical optimality guarantees that have been established for algorithms like Q-Learning [54]. The idea is that using the dependency of the action distribution on the context variable Z, an optimal policy for the multi-task RL case can be simply defined as a multiplexed ensemble of optimal policies for the individual tasks, where the selector is the current context variable z.

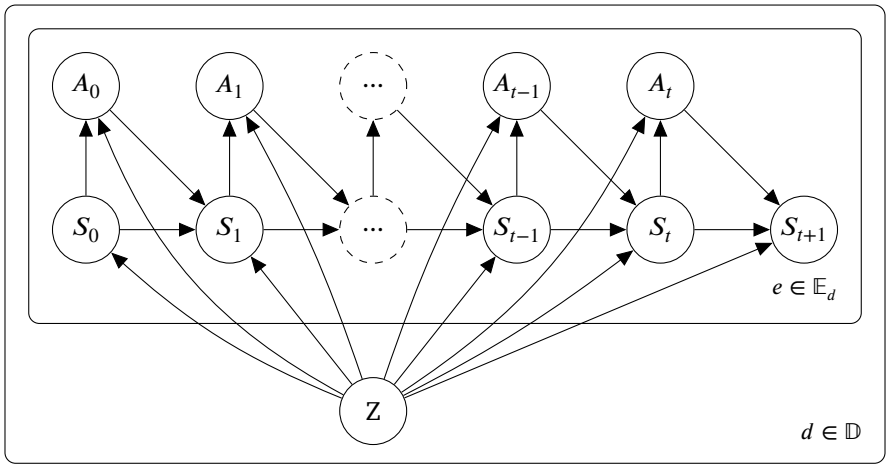


Figure 3.3.: The multi-task RL setup as a graphical model, based on the single MDP case depicted in Figure 3.2. Using d-separation, we find that next states are not conditionally independent of previous states and actions, given only the current state, anymore. The conditional independence only holds when the context variable Z is also given. Using the augmented multi-task MDP formulation, we can show that not observing Z would lead to a partially observable case. Hence the action distribution (policy) is also dependent on Z.

To be able to apply the broad variety of existing methods for single MDPs, it is handy to unify the distribution over MDPs from the multi-task RL setup into a single MDP. This can be accomplished by augmenting the state s with the context variable z to get the augmented state $s^a = (s, z)$. Depending on the representation of z, the augmented state-space $\mathcal{S}^a$ is the union of all augmented state-spaces:

$$\mathcal{S}^a = \cup\{\{(s, z) \mid s \in \mathcal{S}_z\} \mid z \in \mathcal{Z}\}$$

Where $\cup\mathcal{A}$ is the union of all sets in a set $\mathcal{A}$. Calculating the augmented transition probabilities P^a and reward function R^z is simple because they can just be multiplexed using the context variable:

$$\begin{aligned} P^a(s^{a\prime} \mid s^a, a) &= P_z(s' \mid s, a) \quad \text{where} \quad s^{a\prime} = (s', z) \\ R^a(s^a, a) &= R_z(r \mid s, a) \end{aligned}$$

The augmented transition function ensures that the next augmented state will always contain the same context variable as the current state. Furthermore, in this framework for augmenting a multitude of MDPs, the distribution over tasks $P(z)$ can be embedded into the augmented initial state distribution d_0^a:

$$d_0^a(s^a) = d_{0_z}(s)P(z) \tag{3.18}$$

Using this framework, multi-task problems, consisting of many, possibly related MDPs, can be reformulated in terms of a single MDP. This allows the application of a wide variety of RL algorithms to derive a single policy that can act in different environments. Figure 3.4 depicts a Bayesian network of the augmented multi-task RL MDP, which is structurally identical to the MDP of a single task, depicted in Figure 3.2. Going from Figure 3.3 to Figure 3.4, augmenting the context variable z into the state is possible because the distributions for the transition function, policy, and also reward function are conditional on the state and, after augmentation, are hence conditional on the context variable as well. The only simplification that takes place is with the initial state distribution $d_0(s)$, which is not conditional on the state. For d_0 we directly construct a joint distribution

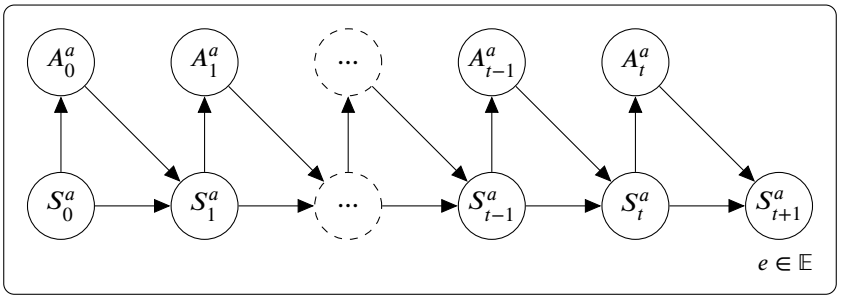


Figure 3.4.: Bayesian network of the augmented multi-task MDP. The context variable Z has been absorbed into the augmented states S_t^a. Note that the augmented multi-task MDP has the same structure as a normal MDP (Figure 3.1 and Figure 3.2) and hence can be approached with normal RL methods.

according to Equation (3.18), which leaves us without the domain handle d. However, using the augmented initial state distribution still results in an equivalent joint distribution over the whole model, and the domain handle d can be seen as an implementation detail signaling episodes with identical domain parameters.

Possible benefits of training a single agent to act in different environments over training a separate agent for each distinct task can be achieved when the tasks share a similar structure. In that case, the goal can be to achieve *positive transfer* using multi-task RL. The term *positive transfer* stems from the general meta-learning community and, based on the definition in [68], refers to the case where training a single model across multiple tasks to reach a certain performance threshold takes fewer data than training individual models for each task and accounting for their combined data requirements. Leveraging *positive transfer* can also not only improve the data efficiency but also enable reaching a certain performance threshold at all. For example, in the case where a problem consists of multiple tasks with equal, small amounts of data (not sufficient to train a distinct model), multi-task learning, by sharing its internal representations across tasks, can lead to a desired performance.

This notion of the term *positive transfer* can also be analogously applied to multi-task RL problems. Concerning the inherent shortage of data from real robots, an interesting application of multi-task RL with the goal of leveraging *positive transfer* could be to train across a distribution of n simulated environments with varying dynamics parameters and to include the real system as task $n + 1$. In this case, the number of simulated environments n could be matched to fill the gap where the training algorithm would wait for the execution on the real robot. If *positive transfer* can be achieved in this setup, it could help bridging the *reality gap* occurring during the sim-to-real transfer (more details in Section 3.4).

One major limiting factor to this approach would be the representation of the context variable z. It is not possible to use the modes of deviation, aka the varying dynamics parameters, for the representation of the environments because these are usually not known for the real system. If the real system would have been identified and its dynamics parameters are known with a high degree of confidence, one could directly use a single simulation to train an agent and transfer it to the real system. With unknown dynamics (parameters), the environments have to be distinguished by, e.g., one-hot vectors or by integral identifiers. These discrete representations can be fine for qualitatively different tasks but relatively uninformative for quantitatively different tasks like with, e.g., unknown dynamics parameters in the sim-to-real case.

3.3. Meta-Reinforcement Learning

Meta-Reinforcement Learning (meta-RL) is concerned with training a single system once, which then can quickly learn to act in different environments. In general, meta-learning is based on presenting example inputs and their expected outputs to a model, so it is able to infer the task that it is asked to do. In addition to these input/output pairs, also called *support set*, the model is presented with a *query set* which consists of inputs assumed to be sampled from the same distribution (task) as the *support set*. The model should then predict the outputs for the *query set* by using the knowledge from the *support set* to discern the task it is asked to solve.

An example of meta-learning in computer vision could be the recognition of certain objects. For example, one task could be to decide if there is an animal in a picture, and another task could be to decide if there is a car in the picture. The *query set* would contain images with and without animals as the *support set* (with appropriate labels reflecting the presence of animals in each image). While presenting this *query set*, the agent is trained to classify unseen images (not contained in the *query set*) that are sampled from the same distribution as the input samples in the *support set* (same task). Using meta-learning benchmarks like the Omniglot challenge [69], it was shown that if trained over a wide enough distribution of tasks, the model is able to generalize to new, unseen tasks [70].

Common neural network-based models for meta-learning are often tailored towards a certain task, e.g., one-/few-shot image recognition. Successful models for these particular tasks are, e.g., Siamese Networks [71], Matching Networks [72], and Prototypical Networks [73]. In [37], a problem agnostic meta-learning model based on a Long Short-Term Memory (LSTM) neural network architecture was first proposed and shown to successfully exhibit meta-learning behavior. Recurrent Neural Networks (RNNs) like LSTMs are well suited for meta-learning because their structure allows to efficiently input datasets consisting of multiple objects of the same structure. This inductive bias is useful for meta-learning because we want to present the agent with the support-set and query predictions using samples from a corresponding query-set.

A different approach is taken with the Model Agnostic Meta-Learning (MAML) [39] algorithm. In contrast to most meta-learning systems, MAML is not focused on building a problem-specific or agnostic model but proposes a meta-learning algorithm that can be applied to all models supporting gradient descent. In comparison to most meta-learning algorithms, MAML uses gradient steps instead of inference to adapt to tasks. For training, MAML uses a second-order derivative of the loss with respect to the parameters to find parameter settings that show a high decrease in the loss when applying a limited number of gradient steps using task-specific data.

In general, meta-learning algorithms are not directly applicable in the meta-RL setup because the objective is not differentiable wrt. the parameters of the policy. In addition to the training procedure, RL tasks are also not specified in terms of desired input-output pairs like it is the case for supervised meta-learning but rather in terms of MDPs. Because of this, the common data structure in the form of support-set and query-set of supervised meta-learning methods can not be used in meta-RL.

The goal of meta-RL, training agents that quickly learn new tasks in terms of either number of interactions with a new environment or computation time, is similar to supervised meta-learning. In [74], the authors note that it is possible to unify the meta-RL system into the policy of a single agent and that there is no pre-wired difference between "learning", "metalearning" and "metametalearning" (in the case of Partially Observable Markov Decision Processes (POMDPs)). In [33], the authors build on the LSTM architecture presented in [37] for supervised meta-learning and coin the term Deep Meta-Reinforcement Learning (deep meta-RL). The

authors propose a simple Recurrent Neural Network (RNN) based agent that exhibits meta-RL capabilities and is able to adapt by using experience gathered from interaction with new tasks.

The proposed deep meta-RL agent shows efficient exploration and exploitation in a maze navigation problem, where each task differs by the goal position, using just inference with the trained recurrent policy. In addition to multi-armed bandits and maze-navigation, the authors also conduct an experiment inspired by a classic study of animal behavior conducted by Harlow in [75]. Under Harlows paradigm, a monkey was trained to exhibit meta-learning behavior. The tasks consisted of choosing between two distinct objects at each step, where one object was chosen to be the rewarded one at the beginning of an episode. The objects could permute positions, but for a particular task, the rewarded object was kept constant. With significant training, the monkey learned to explore in the first step by randomly picking one of the two new objects. Because of the determinism of the environment, the task at hand can be fully known after receiving the first reward. The monkey learned this structure and was able to pick the right object with high accuracy in subsequent steps. For examination of the deep meta-RL agent, in [33] the task is adapted by replacing two objects with two visually distinct images. In this setup, the deep meta-RL agent is able to exhibit close to optimal behavior.

At the same time, a closely related work has been published in [34] where the authors use a Gated Recurrent Unit (GRU) based RNN instead of an LSTM and evaluate their approach called RL^2 (Fast Reinforcement Learning via Slow Reinforcement Learning) on very similar tasks (multi-armed bandits, maze navigation). One interesting finding in [34] is that the meta-learning is not limited by the representational capacity of the RNN based model but by the RL training algorithm it is trained with. The authors come to this conclusion based on a multi-armed bandit experiment, where the policy, trained using TRPO (cf. Section 3.1.3), is compared to a policy trained in a supervised way. The training targets for the supervised learning are provided by an approximately optimal algorithm that has access to privileged information (in this case, the true conditional distribution over rewards given an action). This setup is similar to the one proposed in [18], where a policy is trained to perform acrobatic maneuvers with a quadrotor drone through supervision by a *privileged expert* which uses privileged ground truth information from a simulator to find actions leading to optimal trajectories. In the case of the multi-armed bandit experiment with the RL^2 algorithm, this setup shows that the RL^2 model is able to represent the optimal behavior, cloned from the *privileged expert*, while the policy trained using TRPO is suboptimal. From this result, the authors conclude that there is room for improvement of the RL algorithms. This argument also works in favor of our PUDM-RL setup because by separating the challenging part of the training (training the RNN in the dynamics encoder) we can use unsupervised learning instead of relying on an RL algorithm to train it using trial and error learning (where all the information needs to be fed through a very noisy gradient estimate).

Interestingly, the authors do not mention that their algorithm is a meta-RL algorithm but describe it as the slow learning of a fast learner. However, considering RL^2 has the same structure as the deep meta-reinforcement agent proposed in [33], it can also be considered a meta-RL algorithm. Additionally, in [31], RL^2 is referred to as an "on-policy meta-RL algorithm".

A very successful application of the RL^2 model has been shown in [40], where an RNN based policy is trained in simulation over an increasingly broad task distribution, created using an elaborate DR schedule. The goal of the agent is to solve a Rubik's cube using a complex robotic hand, and the reason for the usage of a meta-RL algorithm is to facilitate sim-to-real transfer, which will be introduced in the next Section 3.4.

3.4. Simulation-to-Reality Transfer

Simulation-to-reality (sim-to-real) describes the problem of applying a model learned using simulated data to real-world data (where the model can also refer to a control policy). This problem comes up frequently in the area of robotics and especially RL for robotics because training a policy directly, using a real robot, is not feasible due to the combination of high costs for executing trajectories and low sample efficiency of contemporary RL algorithms. The high costs for executing trajectories are constituting of wall-clock time, wear/breaking of robot parts, and also safety concerns. In addition to that, early-stage, randomly initialized policies can exhibit correlated actions in a way that breaks the robot or surrounding gear (similar to a *resonance disaster*). In practice, these destructive behaviors rarely produce a good return because the reward function is usually designed in a way to prevent, e.g., high torques or continuous fast movement (refer to [76] for more details on reward function design). This is another reason why training in simulation, in this case, especially for the initial exploration phase, can be beneficial.

Another reason for training a control policy in simulation is the possibility to use the asymmetric actor-critic setup, which has been first proposed for robot learning in [21]. In the asymmetric actor-critic setup, the critic has access to more or better information about the state of the robot. In [21], the critic receives the ground truth state of the system, while the actor only receives an image of the scene. The authors set up the problem in a way such that the systems are fully observable by only using an RGB image to not break the Markov property as seen from the actor. So the usage of an asymmetric actor-critic setup is motivated mainly by the signal-to-noise ratio of the available inputs at test time (on the real system) versus the dimensionality of the ground truth state vector. This disparity becomes relevant when only visual input is available at test time because images of robots with a finite number of links (opposed to, e.g., soft-robots) are extremely redundant and contain a lot of useless data with respect to the robot's state. In these cases, training in simulation can be beneficial because the ground truth state is usually readily available. While directly using the real system, the task setup would need to be modified to, e.g., contain an optical tracking system. Using the ground truth state information for the critic can help the training because in actor-critic methods (due to the policy gradient theorem described in Section 3.1.2), the training of the actor is guided by the critic because all of the information in the reward signal needs to be modeled by the critic first (using some proxy for the mean squared value error, e.g., TD learning, GAE or Monte-Carlo estimates) before it can benefit the actor through training using the policy gradient.

In contrast to the problem of learning behavior using RL, many problems which can be approached using supervised or unsupervised learning can usually use real-world data. For supervised learning, apart from, e.g., time-series forecasting, often manual annotation is needed, which can also imply high costs. A major example of that is the ImageNet dataset [77], where in the original version, 3.2 million real-world pictures were annotated by humans. This huge annotation effort lead to numerous breakthroughs and turned out to be highly reusable for downstream tasks using, e.g., transfer-learning. This poses the question if a similar success could be achieved with shared datasets for RL in robotics. In contrast to data involved in most computer vision problems, data from rollouts during the training of an RL agent is rarely reusable. Even assuming a robot-setup can be replicated very closely (which would already pose a big challenge in reality), the data could only be reused using off-policy RL algorithms. But even off-policy RL algorithms will probably lead to suboptimal solutions because of the sparsity of the data under the trajectory distribution of the actual policy.

One notable exception where the computer vision community is also using simulation/synthetic images is optical flow estimation. The optical flow problem is concerned with estimating the movement of each pixel given two subsequent frames of a scene with moving objects (and/or a moving camera). The ground-truth optical flow map is very hard to measure or annotate. To still facilitate the training of an optical flow model

using supervised learning, datasets like *Sintel* [78] and *Flying Chairs* [79] have been created. While the Sintel dataset is based on a 3D animation movie where the ground-truth optical flow can be calculated from the structure and movement of objects in the 3D scene and the camera movement, the *Flying Chairs* dataset places 3D renderings of chairs on top of random pictures of cities, landscapes and mountains and applies affine transformations to generate displaced frames with known pixel correspondences. Prior to the creation of these synthetic datasets, it was not feasible to train deep learning-based models on the existing small and handcrafted datasets. In [79], the authors observe that their optical flow model generalizes very well to realistic images despite being only trained using images of flying chairs. The idea of using synthetic data to overcome the high cost of annotation has also been applied to other problems like semantic segmentation, but the most prominent datasets are still based on human labeling.

With robotics being an interdisciplinary field and perception being one foundational pillar for robotic control, the transfer of (camera-based, learned) perception systems from simulation to reality has been an early focus of the community. A diverse set of approaches has been proposed to bridge the visual *reality gap* between rendered 3D scenes and images of real scenes. While DR (described in detail in the next Section 3.4.1) has been shown to be effective for zero-shot transfer and subsequently been applied and studied under many different circumstances, there are also notable alternative approaches based on, e.g., transfer learning.

In [19], the authors achieve sim-to-real transfer of end-to-end visuomotor policies using progressive nets for a reaching task. Progressive nets are a transfer learning technique where the pre-trained network is extended by a new neural network for each task. The new neural network is connected to the hidden representations of the pre-trained network through lateral connections. By deactivating the gradient with respect to the weights of the old network, it is possible to mitigate the catastrophic forgetting effect, which makes normal fine-tuning of pre-trained networks challenging. The authors pre-train an expressive policy using 50M steps ($\approx$ 53 days using the real robot) in simulation and extend it by a less complex policy for transfer to the real system. The less complex, progressive policy can use the representations from the pre-trained policy to adapt it to the real system in 60000 steps. 60000 steps would equal $(53 * 24h)/(50 * 10^6) * 60000 \approx 1.5h$ taking their earlier estimate for the training-time of the pre-trained policy, but they report the 60000 steps of transfer learning to have taken approximately four hours. More than a single-digit number of hours for training on the real system is often undesirable due to wear, breaking parts, and also due to turnaround times for evaluation of hyperparameters.

In addition to transfer learning and DR techniques (described in the next Section 3.4.1), there are also domain adaptation approaches which, according to [22], can be separated into feature-level and pixel-level adaptation techniques in the case of visual recognition problems. Feature-level adaptation builds on domain invariant feature extraction, while pixel-level adaptation mutates data from the distribution of a source domain to be more likely under a target domain. To facilitate learning a mapping between source and target domain, according to [22] usually Generative Adversarial Networks (GANs) are used. Generative Adversarial Networks (GANs) are practical for this because supervised learning is not possible due to the lack of training targets, as there are usually no direct correspondences between samples from the source and target dataset. The adversarial loss used in GANs mitigates this by simultaneously training a generator, which in the case of domain adaptation is conditioned on an image from the source domain and should output a corresponding image in the target domain, and a discriminator telling apart synthetic and true samples. Using the usual adversarial loss in this setup can lead to the generator outputting images that are realistic but do, e.g., depict a different position of the robotic manipulator. To maintain semantic correspondence between the source image and the generated target image from the generator, the authors of [22] add additional loss terms based on a mean squared reconstruction error and a reconstruction of a ground truth semantic segmentation mask from the synthetic source image.

3.4.1. Domain Randomization

Domain Randomization (DR) is a sim-to-real technique with the goal of learning transferable models and policies by training over a distribution of datasets. The main idea is that it is impossible to truly simulate the real physics either due to unknown parameters or complex physical effects, and hence a model or policy should be trained using a distribution over datasets that are designed in a way such that data from the actual, physical world has a decent probability under that distribution. To generate a distribution over datasets, the datasets should be parametric and hence governed by domain parameters.

Visual Domain Randomization In the case of computer vision models, the domain parameters are often chosen to be the textures because it is well known that Convolutional Neural Networks (CNNs) are mainly sensitive to textures rather than shapes. In [80], the authors show that CNNs are indeed mainly relying on textures to classify images, and they observe that to CNNs a cat with an elephant texture is an elephant, while to humans, it is still a cat. In the case of natural images (like with the ImageNet dataset), overly relying on textures might not pose a direct problem (when not considering, e.g., adversarial examples), but in the case of the sim-to-real transfer of vision models, it can hamper the performance of the transferred model. The degradation in performance can be attributed to the complexity of simulating real cameras, including effects like, e.g., rolling-shutter distortion, motion-blur, lighting/flares, and complex, nonlinear correlated noise. Except for rolling shutter distortion, all the named effects directly influence the perception of textures. While humans are still able to understand degraded pictures from real cameras using mainly shape-based cues, vision models trained on synthesized images usually fail due to the large *reality gap* in the textures.

The problem of over-reliance on textures in modern perception models has been recognized in [15], where the authors coin the term DR, which has been used before but gained a lot of attention after their publication. To mitigate the reliance on textures and emphasize the usage of shape-based cues for object recognition (for a robotic manipulation task), the authors randomize the textures of their synthetic scenes, which leads to visually very unreal images. Because after randomization of the textures, the generated images are inconsistent wrt. their texture, the vision model is forced to rely on shapes to detect objects accurately. Prior to [15], in [20], the authors already used randomized textures (inter alia) to train vision-based transferable navigation policies using RL but did not motivate why they chose this approach. In the generation of the training data, this approach of visual DR is closely related to the approaches for optical flow estimation in computer vision. DR of textures has since been applied on many occasions, like, e.g., for drone racing [18] or robotic grasping [21].

Domain Randomization of Dynamics Parameters In robotics, in addition to the sim-to-real transfer of perception modules, also the transfer of control policies usually poses a challenge due to inaccurate dynamics models used in the simulation. In contrast to the training of perception models, where, as stated earlier, the creation of accurate simulators is very challenging and hence uncommon, in the case of the development of controllers, coming up with ever more accurate models and identifying precise parameters for them has been the most common approach in the past decades. State-of-the-art methods in optimal control and Model Predictive Control (MPC) rely heavily on accurate dynamics models to find good control policies, exhibiting desired behavior. In contrast to these common methods from control engineering, many RL algorithms are model-free but nevertheless are able to attain state-of-the-art performance in complex control environments. Despite being model-free, these RL algorithms still rely on interaction with an environment whose dynamics should be the same at training and test-time. Hence when training in simulation, model-free RL algorithms in

theory also require the simulator to be an accurate model of the real system to facilitate transfer to the real system without loss of performance.

An alternative approach to the extensive tuning of the simulator is training a single agent across multiple versions of the simulator with perturbations in the parameters of the dynamics model. Examples for dynamics parameters are friction coefficients, inertia tensors, masses, lengths, and even the dt of an interaction step. The distributions over the different parameters are usually independent and should ideally give a high probability (-density) for the actual parameters of the real system. By training across a distribution of environments, the goal is that the trained agent will be robust against discrepancies between the domain parameters of the simulator and the real system. Successful sim-to-real transfer using DR of dynamics parameters has been reported for a robotic pivoting task in [24] and locomotion in [25]. Policies trained across a distribution of different dynamics are also often referred to as *conservative policies* because they are incentivized to pick actions that result in good rewards in all of the environments (weighted by the probability of their domain parameters). Conservative policies are generally not able to pick the optimal action for a particular environment because they can not distinguish its dynamics from the dynamics of the other environments.

Conservative policies are not able to distinguish between the environments because DR is a special case of multi-task learning, with partial observability (POMDP). The domain parameters are collected in a random variable Ξ with particular realizations ξ. The domain parameters govern the dynamics of the system and hence give rise to distinct MDPs, each with distinct transition probabilities $P_\xi\left(s' \mid s, a\right)$ depending on realizations ξ of Ξ. So, in terms of the transition probabilities, the distribution over MDPs arising from DR has the same structure as the multi-task setup described in Section 3.2, with ξ taking the role of the context variable z.

Because with DR, we want to train a single agent to act in diverse environments, we can take advantage of the reformulation of multi-task learning into a single, augmented MDP (described in Section 3.2). This results in a new MDP based on augmented states $s^a = (s, z) = (s, \xi)$. In simulation, we could directly use this MDP to train an agent using, e.g., one of the RL algorithms described in Section 3.1.2. The RL algorithm does not need to be modified because the DR is transparent by using the initial state distribution (Equation (3.18)) to sample the domain parameters and using the augmented transition probabilities to generate a full trajectory in a particular environment. In the results section, we will refer to this setup as a *multi-task learning* setup and show that it yields agents with very good behavior. The big issue in using this setup for a sim-to-real transfer, which is the ultimate goal of DR, is that the augmented state $s^a = (s, \xi)$ requires the domain parameters ξ to be known. Identifying accurate domain parameters ξ would again require extensive task-specific measurements on the real system, which might be prohibitively expensive or even infeasible for certain parameters. In addition to the cost of system identification, once the parameters are estimated with good accuracy, the multi-task learning-based DR setup should give a low expected benefit over just using the identified parameters directly in a single simulator.

Using the augmented state $s^a = (s, \xi)$ is not only troublesome for the sim-to-real transfer, but it is also not compatible with the notion of a conservative policy, which, as described before, is not aware of the current environment. In contrast, policies having access to the augmented state are also sometimes referred to as *oracle policies*. Applying a conservative policy to the augmented MDP leads to partial observability of the augmented state with a deterministic observation function $O(s^a) = s$ that is disregarding the augmented domain parameters. This yields a special case of a POMDP where the unobserved part of the state is constant throughout an episode.

This partial observability is usually not assumed in the case of visual DR. Firstly, in, e.g., [19] [21] and [15], the cameras are positioned in a way that allows to infer the state with a minimum level of ambiguities, and through their usage of conservative policies, the full observability of the state is assumed. Secondly, the DR of, e.g., textures does not impact the environment's dynamics, and it also maintains the structural information

content of the positions, shapes, etc., which constitute the state of the scene. This fundamental difference between visual DR and DR of dynamics parameters is usually not highlighted but separates them into two almost disjunct problems. In the following with DR, we will refer to domain randomization of dynamics parameters.

The special case of a POMDP arising from DR gives rise to approaches with adaptive policies. The idea of adaptive policies mainly revolves around the finding that policies with recurrent neural networks outperform conservative policies under DR. In Section 4.1, we will show that the usage of adaptive policies with DR of dynamics parameters is meta-RL. In the following, we give an overview of works that successfully apply adaptive policies to achieve sim-to-real transfer using DR.

Examples of Successful Simulation-to-Reality Transfer Using Domain Randomization As described at the end of Section 3.3, a very successful application of DR as a sim-to-real technique has been demonstrated in [40]. In this work, the goal is to train a policy that is able to manipulate a Rubik's cube using a complex robotic hand consisting of 20 joints. The focus lies on learning the physical manipulation of the cube and not on developing sequences of moves to restore the uniform colors on all faces. The sequence of moves is generated by an external, high-level planner. The authors note that they put much effort into the calibration of their simulation to find appropriate parameters for the dynamics of the system, but due to the high dimensionality and due to the involvement of contacts between objects (which are inherently hard to simulate), a straightforward transfer of a policy trained in simulation to the real system is not feasible. With this problem, the sim-to-real challenge can be divided into a perception and a control part. The control part can be examined in isolation under the assumption that the perception system provides a good estimate of the state of the Rubik's cube and the robotic hand. By using an adaptive policy, the authors are able to solve a Rubik's cube with the real robotic hand after only training in simulation. They demonstrate that their policy is even robust against adverse perturbations of the system, like tying together two fingers of the hand or adding a sticky rubber glove.

The groundwork for moving a solid cube into a desired position in a single hand has been laid in the preceding work [6]. In this work, the authors also show successful sim-to-real transfer after training an adaptive policy in simulation using DR. They also show that using an adaptive policy (consisting of an LSTM with 512 units compared to 1024 units in [40]) is central to allow domain transfer by studying the effect of memory (aka hidden LSTM state) in policies. In their observations, policies without recurrence severely underperform adaptive ones in training speed (wall-clock time) and final performance. In both [6] and [40], the asymmetric actor-critic approach is used, where the critic has access to privileged information about the state to better guide the training of the actor. Using privileged information for the critic is possible as the critic is only applied in the training phase using the simulator, where ground truth information about, e.g., object positions is available. At test time on the real system, only the trained policy is executed.

Prior to these impressive successes concerning in-hand manipulation reported in [6] and [40], the usage of adaptive policies in combination with DR, facilitating sim-to-real transfer, has been proposed for a pushing task in [43].

4. Approach

In this chapter, we will introduce our proposed Partially Unsupervised Deep Meta-Reinforcement Learning (PUDM-RL) approach. First, we motivate the combination of Domain Randomization (DR) and Meta-Reinforcement Learning (meta-RL). Based on this motivation, we derive our PUDM-RL approach, which allows for separation of the training of the dynamics encoder from the Reinforcement Learning (RL) loop. During the derivation, we leave the implementation details of the dynamics encoder open and treat it as a (differentiable) black box. Following the derivation, we present different architectures for dynamics encoders including their advantages and disadvantages. Finally, we give an overview of the whole training process, spanning data collection, dynamics encoder training, and training of the final agent using RL.

4.1. Combining Domain Randomization and Meta-Reinforcement Learning

In this section, we will show how using DR of dynamics parameters for simulation-to-reality (sim-to-real) transfer lends itself naturally to the application of meta-RL. Analogous to the view of DR as multi-task RL with partial observability, we show that using *conservative* policies in the unaugmented case leads to a violation of the Markov property, which is usually a requirement for the applicability of RL algorithms.

As stated in Section 3.4.1, we can view DR as a multi-task RL problem where the context variable z, distinguishing the different environments, consists of the domain parameters $z = \xi_d$. Starting with the directed graphical model in Figure 3.3, DR leads to the graphical model depicted in Figure 4.1. Note that we only randomize parameters of the dynamics model, which leaves the initial state distribution independent of the domain parameters Ξ_d. The structure of the MDP using a conservative policy is shown in Figure 4.2 and, as shown in the following, breaks the Markov assumption. An overview of the possible RL setups involving DR is provided in Table 4.1.

Policies Requiring Observation of Domain Parameters Because the oracle/multi-task approach assumes full observability of the domain parameters Ξ_d, this policy structure can be used as a benchmark in simulation or on the real system when there are only a few domain parameters, and the effort necessary for measuring them is not too high. In the following, we assume that the domain parameters are not directly observable. Adaptive policies, on the other hand, are dependent on all prior interactions from episodes in an environment with particular domain parameters.

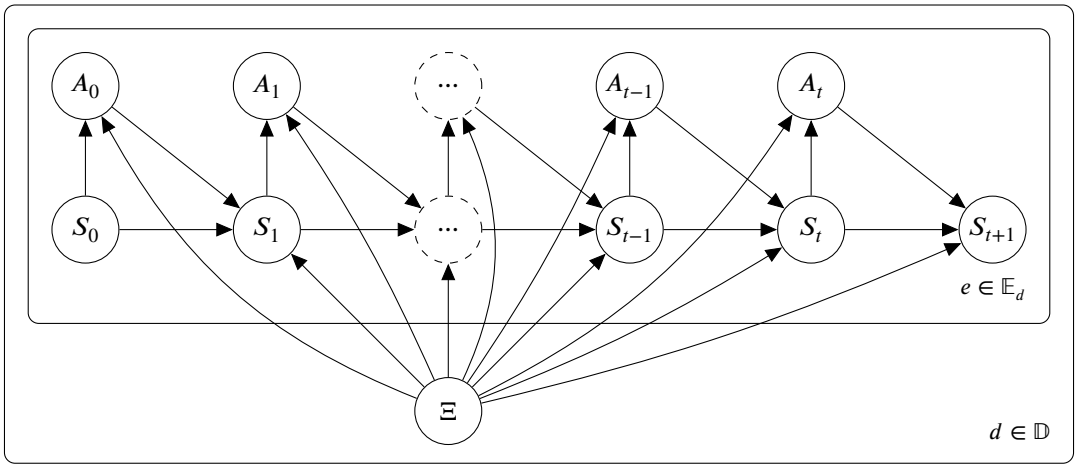


Figure 4.1.: DR creates a distribution over Markov Decision Processes (MDPs) and hence can be cast as a multi-task RL problem. Comparing the resulting Bayesian network to the one in Figure 3.3, we can see that it is a special case of the multi-task RL case, where the initial state distribution is independent of the context variable.

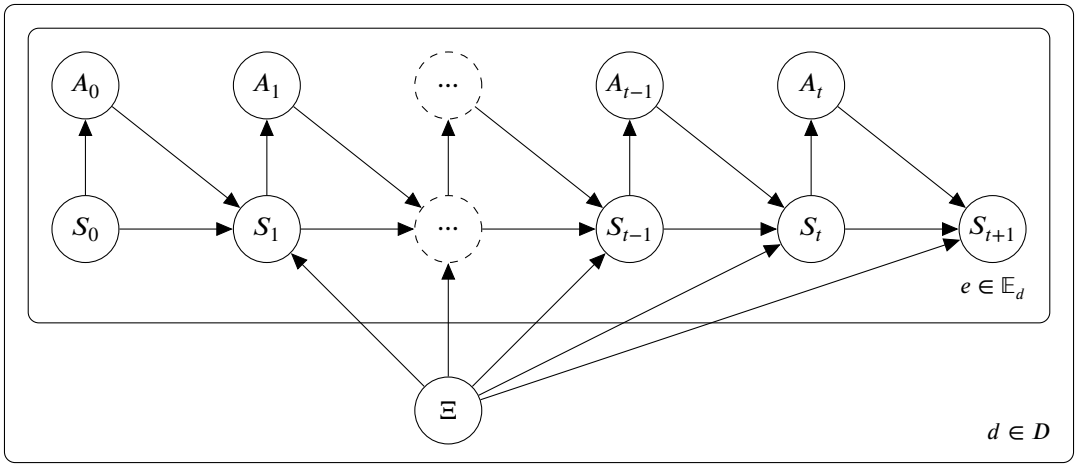


Figure 4.2.: Same Bayesian network as in Figure 4.1 but now assuming a conservative policy, only depending on the state. By augmenting the MDP (cf. Section 3.2), we can see that usage of a conservative policy leads to partial observability.

Policy Type	Figure	Signature	Remarks
oracle/multi-task	4.1	$\pi\left(a \mid s, \xi_d\right) = \pi\left(a \mid s^a\right)$	Ξ_d is observed → independence on past interactions
conservative/plain DR	4.2	$\pi\left(a \mid s\right)$	Ξ_d is unobserved → breach of the Markov property (disregarded)
adaptive/meta-RL	4.1	$\pi\left(a \mid \tau_{d,e,t}\right)$	Ξ_d is unobserved → dependence on past interactions

Table 4.1.: Overview over different policy structures.

Policies Not Observing Domain Parameters In the following, we assume that the domain parameters Ξ_d are not observed and study the implications of this assumption on the dependency structure of policies. These policies (conservative/adaptive) are generally viable for sim-to-real transfer because they are only dependent on observable variables. Applying the d-separation criterion to the Bayesian network in Figure 4.1, we notice that the usual conditional independence does not hold anymore. An example of an active path proving conditional dependence is, e.g., the path from S_{t+1} to S_{t-1} given S_t and A_t. The path along the chain of states is blocked by observing S_t, but as long as the domain parameters Ξ_d are not observed, the dependence can flow from S_{t-1} through Ξ_d to S_{t+1}. The same is holding for all other paths to states and actions prior to timestep t except for S_0 and A_0, which are only involved when S_1 is also observed because they are engaged in a collider formation. The same conditional dependence can be observed when a conservative policy is used (depicted in Figure 4.2).

In addition to graphical models, we can also express the conditional independence in equations using the conditional independence notation $(\bullet \perp\!\!\!\perp \bullet) \mid \bullet$:

$$(A \perp\!\!\!\perp B) \mid C \;\Leftrightarrow\; P(A \mid B, C) = P(A \mid C)$$

For a normal MDP (depicted in Figure 3.2 and 3.1), the Markov property entails the following conditional independence (for a single episode):

$$\left(S_{t+1} \perp\!\!\!\perp \left\{S_{t-1}, A_{t-1}, S_{t-2}, A_{t-2} \ldots, S_0, A_0\right\}\right) \mid S_t$$

To maintain a concise notation, we denote $T_{d,e,t}$ as the collection of state and action random variables in a certain episode e that is generated under the domain parameterization d, until timestep t (inclusive last state, exclusive last action):

$$T_{d,e,t} = \left\{S_{d,e,0}, A_{d,e,0}, S_{d,e,1}, A_{d,e,1}, \ldots, A_{d,e,t-1}, S_{d,e,t}\right\}$$

Note that we do not include $A_{d,e,t}$ to avoid the usage of a set difference every time we condition, e.g., the policy on the experience. Considering DR, using $T_{d,e,t}$, we can now express the conditional dependence of a

next state $S_{d,e,t+1}$ on previous states and actions from the same episode and even from other episodes within the same domain:

$$\left(S_{d,e,t+1} \not\perp T_{d,e,t-1} \cup \left\{A_{d,e,t-1}\right\} \cup \left\{T_{d,e^+,N} \mid e^+ \neq e\right\}\right) \mid S_{d,e,t}$$

By marginalization, we can find the transition function given past interactions in that particular domain d:

$$\begin{aligned} P\left(s_{t+1} \mid s_t, a_t, D_{d,e,t}\right) &= P\left(s_{t+1} \mid a_t, D_{d,e,t}\right) \\ &= \int_{\xi} P\left(s_{t+1} \mid s_t, a_t, \xi_d\right) P\left(\xi_d \mid D_{d,e,t}\right) \mathrm{d}\xi \end{aligned} \tag{4.1}$$

Where $s_t \sim S_{d,e,t}$, $a_t \sim A_{d,e,t}$, $\xi_d \sim \Xi_d$ and $D_{d,e,t}$ is the set of observed states and actions in environment d up until timestep t in episode e:

$$D_{d,e,t} = \left\{ \begin{array}{c} s_{d,0,0}, a_{d,0,0}, s_{d,0,1}, a_{d,0,1}, \ldots, a_{d,0,N_{d,e}-1}, s_{d,0,N_{d,e}}, \\ s_{d,1,0}, a_{d,1,0}, s_{d,1,1}, a_{d,1,1}, \ldots, a_{d,1,N_{d,e}-1}, s_{d,1,N_{d,e}}, \\ \vdots \\ s_{d,e,0}, a_{d,e,0}, s_{d,e,1}, a_{d,e,1}, \ldots, a_{d,e,t-1}, s_{d,e,t} \end{array} \right\}$$

$$s_{d,e,t} \sim S_{d,e,t} \ , \ a_{d,e,t} \sim A_{d,e,t}$$

Where $N_{d,e}$ is the number of steps in episode e of domain d. This notation is sustained from hereon. Equation (4.1) shows the dependency of the transition function on past observations in the case of unobserved domain parameters Ξ_d. This dependence also propagates into optimal policies. According to Bellman's principle of optimality, the optimal policy is greedy with respect to the optimal value function:

$$\begin{aligned} Q^*(s,a) &= \underset{s'}{E}\left[R + \gamma \max_{a'} Q^*\left(s', a'\right) \mid D_{d,e,t}\right] \\ &= \int_{s'} P\left(s' \mid a, D_{d,e,t}\right)\left(R + \max_{a'} \gamma Q^*\left(s', a'\right)\right) \mathrm{d}s' \end{aligned}$$

Where the expectation is over next-state transition probabilities according to Equation (4.1). We can see that in the case when the domain parameters are not observed, the optimal action-value function is dependent on the past interactions $D_{d,e,t}$ and hence the optimal (greedy) policy is dependent on $D_{d,e,t}$. This shows that under DR, a conservative policy can only be approximately optimal (in the best case).

This observation renders adaptive policies theoretically sounder than conservative policies. For a simple example, emphasizing the dependence of adaptive policies on past interactions, consider the one-dimensional force control of a point-mass towards a setpoint with some given reward function. In this case, we could define the mass to be the single domain parameter Ξ_d. If a policy has interacted with the system for a couple of steps, by observing the cause and effect of its actions on the state of the system, it can infer the mass and chose appropriate actions to maximize the reward for that particular configuration of Ξ_d. Alternatively, if the mass Ξ_d was provided to the agent (as it is the case for oracle policies), its (optimal) actions are not dependent on past interactions.

This formalization shows that when combining DR with a conservative policy, the RL is taking place over a decision process that is not Markovian, which invalidates the optimality guarantees of the RL algorithms (e.g. Q-Learning). Nevertheless, in practice, this approach has been shown to be effective in [24] and [25], where the effectiveness can be attributed to narrow distributions over the domain parameters.

Theoretically, we can construct a simple thought experiment as a counter-example, where taking the optimal action is impossible for the conservative policy, using action inversion. Revisiting the point-mass example, if the distribution of masses is very narrow, the learned conservative policy can be close to optimal, as the optimal actions for similar masses are very similar (considering a simple reward function like, e.g., minus squared distance). For the counter-example, let us now assume the single domain parameter Ξ_d is now a Boolean variable determining if the actions are multiplied by -1 or 1. In this example, the conservative policy can not consistently take the optimal action as it is unaware if it is acting in the environment where actions are multiplied by -1 or not. In contrast, an adaptive policy could use past interactions to reason about the domain parameter of the current environment it is acting in and chose more appropriate (theoretically even optimal) actions.

In practice, adaptive policies are usually implemented using Recurrent Neural Networks (RNNs), as described at the end of Section 3.4.1. Recurrent Neural Networks (RNNs) like Long Short-Term Memorys (LSTMs) allow for the efficient implementation of the policy's dependence on previous interactions. The major benefits of LSTMs lie in the sharing of weights enabled by the sequential ingestion of identically structured inputs and a hidden state implementing memory to reduce sequences of possibly variable length to a fixed dimensional representation. In general, the sequential nature of past interactions does not need to be accounted for in the policy. All the information lies in state, action, next-state tuples (s, a, s'). However, in the case of stochastic observations (e.g. due to observation noise occurring in a real system), the exact sequence of (s, a, s') tuples can be crucial for the adaptive policy to simultaneously learn to filter the data into a belief-state and deduce information about the environment's dynamics from it (more details in Section 4.3).

Alternatively, adaptive policies can also be implemented using gradient-based meta-learners that use the available experience in the form of past interactions to execute further gradient steps on the trained policy. An example of this is Model Agnostic Meta-Learning (MAML), which is described in Section 3.3 and is not limited to applications in RL problems. The applicability of MAML is already a hint that the DR setup (concerning the interaction of an agent with its environment) is equivalent to the meta-RL setup. Furthermore, the usage of adaptive policies using RNNs to learn a behavior in a distribution over MDPs is equivalent to (deep) meta-RL as defined in [33], [31], and [40]. In [40], the authors extensively study what they call "emergent meta-learning" and come to the conclusion that their agent exhibits meta-(reinforcement) learning behavior after it was trained in environments with a wide variety of dynamics parameters. To train their agent, the authors propose a scheduling algorithm for the variance in the environment parameters. Their Automatic Domain Randomization (ADR) scheme starts off with a narrow distribution over the dynamics parameters, and each time the agent is able to accomplish a certain performance threshold, it increases the variance in the environments the agent is exposed to, which degrades its performance again. In the end, the agent is able to act in a wide variety of environments and is able to adapt to each particular one during the execution of episodes, which is meta-RL behavior.

In general meta-RL tasks may not only differ in their transition dynamics but also in their reward function. The LSTM-based meta-RL approaches can straightforwardly be extended to the case of varying reward functions by incorporating the rewards into the past experience $D_{d,e,t}$.

4.2. Partially Unsupervised Deep Meta-Reinforcement Learning

With Partially Unsupervised Deep Meta-Reinforcement Learning (PUDM-RL), the idea is to use the fact that, given the domain parameters, the Markov property is holding, to break the dependence of the agent on the history of interactions. From the perspective of the agent, this setup is similar to oracle policies (cf. Table 4.1)

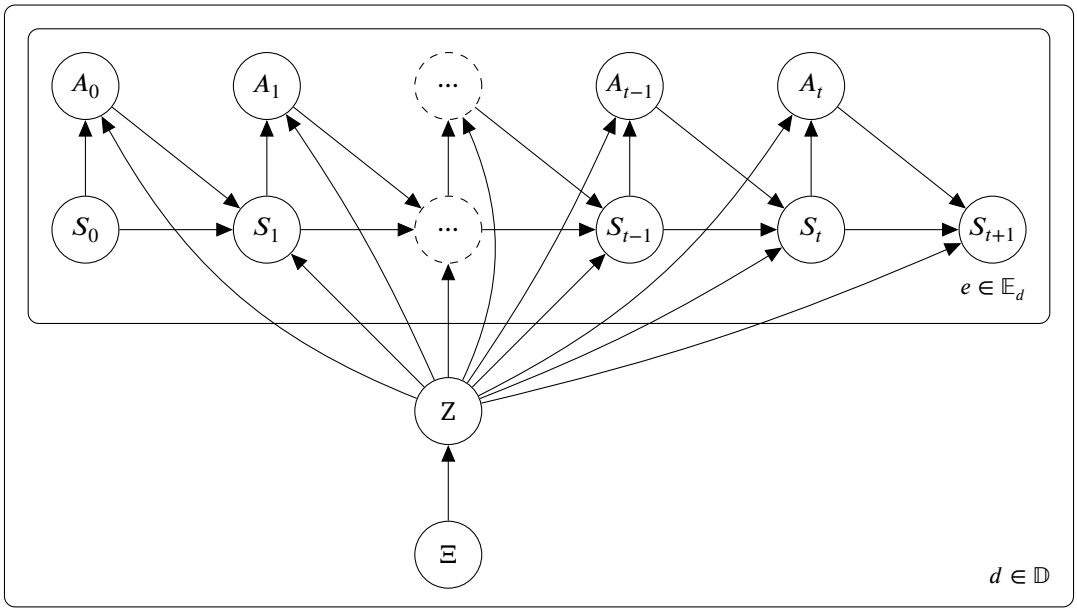


Figure 4.3.: Based on the Bayesian network in Figure 4.1, we introduce a random variable Z_d which is an embedding of the domain parameters and hence captures all the relevant information in Ξ_d. When Z_d is observed, the usual conditional independence assumption holds, even if Ξ_d is unobserved.

because it receives information about the environment as part of an augmented state. In the case of PUDM-RL, the additional information that the state is augmented with has been compressed from past interactions into a low dimensional vector (compared to the dimensionality of the experience), characterizing a particular environment from the distribution over environments.

To break the dependency on the unknown/unobserved domain parameters, we introduce a new random variable Z_d characterizing an embedding vector for the domain parameters Ξ_d. The idea behind introducing a new latent variable Z_d to capture the relevant information from Ξ_d is that the actual representation of the differences between the environments is not important as long as the information suffices to discriminate between them. Alternatively, the dynamics parameters could be estimated directly like it has been proposed in [81], which allows for direct regression using the privileged information during training in the simulation. On the other hand, directly predicting the dynamics parameters can lead to a large chunk of the representational capacity (and training effort) of the system identification module being spent on parameters that are less relevant to the behavioral response of the system because Mean Squared Error (MSE) regression will weight them equally (assuming standardized data). Additionally, the dimensionality of the representation is fixed to the number of dynamics parameters which limits its applicability to environments with large numbers of parameters like, e.g., environments consisting of soft objects of different shapes and with different properties where a parameter vector could have thousands or more dimensions. In this case, the characteristics of the environment's coarse behavior (the part relevant for a particular task) might be reduced to a much lower-dimensional representation. Finally, directly estimating the dynamics parameters also limits the practical representational capacity of uncertainty to unimodal distributions, whereas an embedding vector can, in general, represent arbitrary distributions (as shown later). The modified Bayesian network introducing the random variable Z_d representing the embedding vector is depicted in Figure 4.3 (cf. the Bayesian network prior to the introduction of Z_d in Figure 4.1). Using the conditional independencies, we can calculate the

joint distribution modeled by the Bayesian network in Figure 4.3 in a factorized way:

$$P\left(D_{d,e',N_{d,e'}},\xi_d\right) = P\left(\xi_d\right)\prod_{e<=e'}P\left(s_0\right)\prod_{t=0}^{N_{d,e}}P\left(s_{t+1}\mid s_t,a_t,\xi_d\right)\pi\left(a_t\mid s_t,\xi_d\right)$$

Where e' is the last episode in the particular domain d. As the domain parameters Ξ are unobserved, the actual data distribution that we are able to observe from interaction with the environments is the following marginal distribution:

$$\begin{aligned}P\left(D_{d,e',N_{d,e'}}\right) &= \int_{\xi_d} P\left(\xi_d\right)\prod_{e<=e'}P\left(s_0\right)\prod_{t=0}^{N_{d,e}}P\left(s_{t+1}\mid s_t,a_t,\xi_d\right)\pi\left(a_t\mid s_t,\xi_d\right)\mathrm{d}\xi_d\\ &\overset{!}{=} \int_{z_d} P\left(z_d\right)\prod_{e<=e'}P\left(s_0\right)\prod_{t=0}^{N_{d,e}}P\left(s_{t+1}\mid s_t,a_t,z_d\right)\pi\left(a_t\mid s_t,z_d\right)\mathrm{d}z_d\end{aligned}$$

Where the second equality shows that after introducing the dynamics embedding Z_d, we are still modeling the identical data distribution. Using the conditional independencies in our model (shown in Figure 4.3), we can factorize the data distribution $P\left(D_{d,e',N_{d,e'}}\right)$ backwards in time, leading to causal posterior distributions over the dynamics embedding (with $T := N_{d,e'}$):

$$P\left(D_{d,e,T}\right) = \int_{z_d} P\left(s_T\mid a_{T-1},z_d,D_{d,e,T-1}\right)\pi\left(a_{T-1}\mid z_d,D_{d,e,T-1}\right)P\left(z_d\mid D_{d,e,T-1}\right)P\left(D_{d,e,T-1}\right)\,\mathrm{d}z_d$$

We can apply the conditional independencies $\left(S_t \perp\!\!\!\perp D_{d,e,t-1}\setminus\{S_{t-1}\}\right)\mid S_{t-1},A_{t-1},Z_d$ and $\left(A_t \perp\!\!\!\perp D_{d,e,t}\setminus\{S_t\}\right)\mid S_t,Z_d$:

$$\begin{aligned}P\left(D_{d,e,T}\right) &= \int_{z_d} P\left(s_T\mid s_{T-1},a_{T-1},z_d\right)\pi\left(a_{T-1}\mid s_{T-1},z_d\right)P\left(z_d\mid D_{d,e,T-1}\right)P\left(D_{d,e,T-1}\right)\mathrm{d}z_d\\ &= \left[\int_{z_d} P\left(s_T\mid s_{T-1},a_{T-1},z_d\right)\pi\left(a_{T-1}\mid s_{T-1},z_d\right)P\left(z_d\mid D_{d,e,T-1}\right)\,\mathrm{d}z_d\right]\,P\left(D_{d,e,T-1}\right)\end{aligned}$$

This gives us a recursive formulation that is anchored at timestep 0 by the initial state distribution and recurses through the episodes:

$$\begin{aligned}P\left(D_{d,e,0}\right) &:= P\left(s_0\right)P\left(D_{d,e-1,T}\right)\\ P\left(D_{d,0,0}\right) &:= P\left(s_0\right)\end{aligned}$$

Using these anchors and unrolling the recursion, we can express the data distribution using the following product:

$$P\left(D_{d,e',N_{d,e}}\right) = \prod_{e<e'}P\left(s_0\right)\prod_{t=0}^{N_{d,e}}\int_{z_d} P\left(s_{t+1}\mid s_t,a_t,z_d\right)\pi\left(a_t\mid s_t,z_d\right)P\left(z_d\mid D_{d,e,t}\right)\,\mathrm{d}z_d$$

This expression highlights a chicken and egg problem, where the policy generating the data is dependent on the dynamics embedding, while we want to use this expression to train the *dynamics encoder*. We can circumvent this problem by using, e.g., agents with privileged information (ground truth domain parameters) or individual agents per environment to bootstrap the data-generation. Assuming that there is no sparsity of data when training these bootstrapping agents to collect interaction data of the different environments, we

can rule out the possibility of positive transfer (cf. Section 3.2), and the data represents an upper-bound in the action-selection performance. In practice, the performance of the *dynamics encoder* is not very dependent on the action-selection behavior used to collect its training data, as long as the data covers the (relevant) state-space sufficiently. Using the proclaimed bootstrapping approach, the policy is not dependent on the dynamics embedding anymore, and we can model the training distribution as follows:

$$\begin{aligned} P\left(D_{d,e',N_{d,e}}\right) &= \prod_{e<e'} P\left(s_0\right) \prod_{t=0}^{N_{d,e}} \int_{z_d} P\left(s_{t+1} \mid s_t, a_t, z_d\right) \pi\left(a_t \mid s_t\right) P\left(z_d \mid D_{d,e,t}\right) \, \mathrm{d}z_d \\ &= \prod_{e<e'} P\left(s_0\right) \prod_{t=0}^{N_{d,e}} \int_{z_d} P\left(s_{t+1} \mid s_t, a_t, z_d\right) P\left(z_d \mid D_{d,e,t}\right) \, \mathrm{d}z_d \, \pi\left(a_t \mid s_t\right) \end{aligned} \tag{4.2}$$

Now we can take a closer look at the marginal transition probability distribution:

$$\int_{z_d} P\left(s_{t+1} \mid s_t, a_t, z_d\right) P\left(z_d \mid D_{d,e,t}\right) \, \mathrm{d}z_d = P\left(s_{t+1} \mid a_t, D_{d,e,t}\right) \tag{4.3}$$

The integral is generally not tractable, and we also do not necessarily want to restrict the family of distributions for the posterior distribution $P\left(z_d \mid D_{d,e,t}\right)$. To overcome this, we introduce a deterministic function $f\left(D_{d,e,t}\right) = z'_d$ that takes interaction data as the input and outputs a vector z'_d. z'_d is a vector containing the sufficient statistics for the posterior distribution $P\left(z_d \mid D_{d,e,t}\right)$ to carry the information in the experienced data $D_{d,e,t}$ that is relevant to the environment's dynamics. The deterministic function f can, e.g., be implemented using an RNN and represents the *dynamics encoder*. Using z'_d, we can introduce a new form for the transition probability, which includes the marginalization over the dynamics embedding implicitly:

$$P\left(s_{t+1} \mid a_t, D_{d,e,t}\right) = P\left(s_{t+1} \mid s_t, a_t, z'_d\right) = P\left(s_{t+1} \mid s_t, a_t, f\left(D_{d,e,t}\right)\right) \tag{4.4}$$

We can implement the transition probability distribution using an isotropic Gaussian observer model, where the mean is given by another deterministic function $g\left(s_t, a_t, f\left(D_{d,e,t}\right)\right)$:

$$P\left(s_{t+1} \mid s_t, a_t, f\left(D_{d,e,t}\right)\right) = \mathcal{N}\left(s_{t+1}; g\left(s_t, a_t, f\left(D_{d,e,t}\right)\right), I\right) \tag{4.5}$$

Where we refer to g as the *transition model*, and it can, e.g., be implemented using a Dense Neural Network (DNN). Plugging this form of the transition probability back into the data distribution (Equation (4.2)), we get:

$$P\left(D_{d,e',N_{d,e}}\right) = \prod_{e<e'} P\left(s_0\right) \prod_{t=0}^{N_{d,e}} \mathcal{N}\left(s_{t+1}; g\left(s_t, a_t, f\left(D_{d,e,t}\right)\right), I\right) \pi\left(a_t \mid s_t\right)$$

For the training of the function approximators f and g, we can use maximum likelihood estimation using the log trick:

$$\begin{aligned} \log P\left(D_{d,e',N_{d,e}}\right) &= \log\left(\prod_{e<e'} P\left(s_0\right) \prod_{t=0}^{N_{d,e}} \mathcal{N}\left(s_{t+1}; g\left(s_t, a_t, f\left(D_{d,e,t}\right)\right), I\right) \pi\left(a_t \mid s_t\right)\right) \\ &= \sum_{e<e'} \log P\left(s_0\right) + \sum_{t=0}^{N_{d,e}} \log \mathcal{N}\left(s_{t+1}; g\left(s_t, a_t, f\left(D_{d,e,t}\right)\right), I\right) + \log \pi\left(a_t \mid s_t\right) \end{aligned}$$

For the training of g and f, we are taking the gradient of the data likelihood wrt. their parameters; hence in the cost function, we can disregard the initial state distribution and the policy, which are independent of the trained functions:

$$\begin{aligned}\log P\left(D_{d,e',N_{d,e}}\right) &\sim \sum_{e<e'}\sum_{t=0}^{N_{d,e}} \log \mathcal{N}\left(s_{t+1}; g\left(s_t, a_t, f\left(D_{d,e,t}\right)\right), I\right) \\ &\sim -\sum_{e<e'}\sum_{t=0}^{N_{d,e}} \left(s_{t+1} - g\left(s_t, a_t, f\left(D_{d,e,t}\right)\right)\right)^2 \end{aligned} \tag{4.6}$$

While the log trick maintains the same local and global optima, leaving out the terms from the log-likelihood of the Gaussian leaves the direction of the gradient wrt. g intact and gives rise to standard MSE regression.

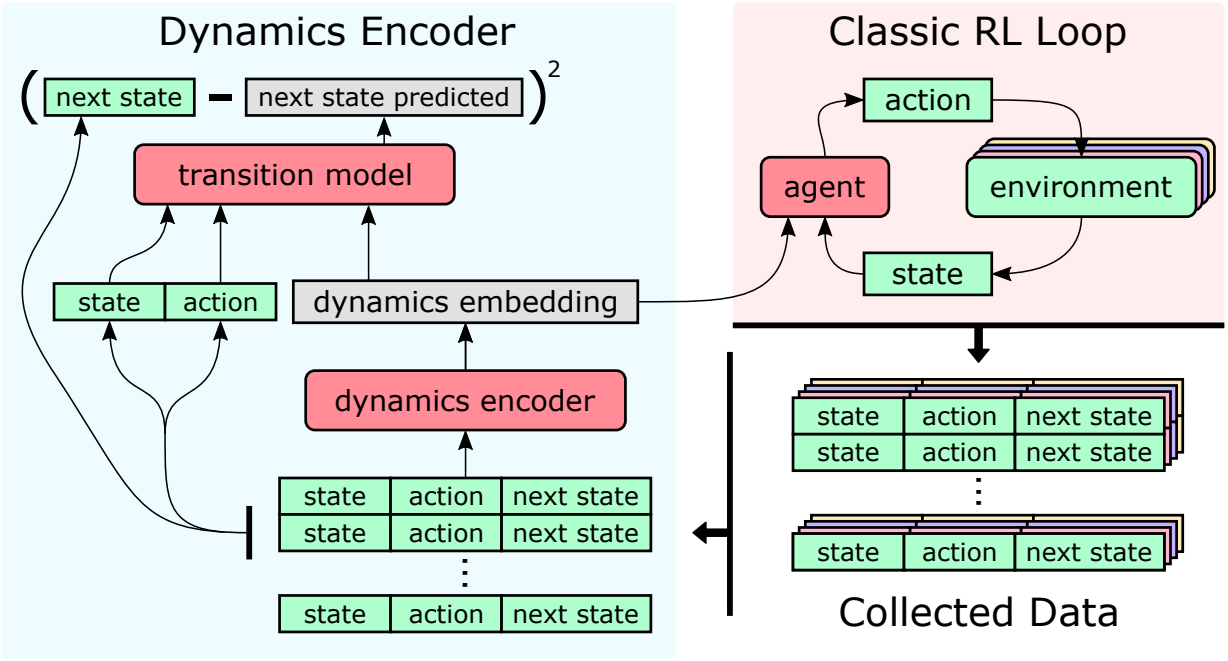


Figure 4.4.: Structure of the dynamics encoder (corresponding to Equation (4.6)) and its relation to the classic RL loop.

Figure 4.4 depicts the described structure for the dynamics encoder in a block diagram. On the right-hand side, the classic RL loop is shown, where the agent now acts in a distribution of different environments due to the DR. When interacting with the different environments, the agent collects information about how a particular environment works by observing cause and effect in the form of state, action, next state tuples. Observing these tuples, the belief of the current environment's dynamics should change, which is expressed through the posterior over dynamics embeddings in Equation (4.3) and through the deterministic function f in Equation (4.4). The latter is depicted as the *dynamics encoder* in Figure 4.4 and compresses a history of interactions with a particular environment into a *dynamics embedding* characterizing the behavior of that environment. The *dynamics encoder* can, e.g., be implemented using an RNN.

Given that the latent space of the dynamics embeddings has an informative structure, the agent can exploit it to act better in each individual environment, accessing the compressed knowledge from its past interactions.

This cycle again highlights the chicken and egg problem mentioned before, and there are straightforward options to tackle it, exploiting the availability of a simulator as stated earlier.

Looking at the assumption that the latent space containing the dynamics embeddings is informative, the question arises of how we are able to enforce this. At this point, the *transition model* represented through a Gaussian observer based on the function approximator g in Equation (4.5) can be applied to provide a training target in the form of a simple MSE cost function. A normal transition model is supposed to predict the next state given a current state and action from a particular environment. In our case, the transition model shall be able to predict the next state in a distribution over environments, which raises the requirement for additional information as there is ambiguity about the dynamics that govern a particular state, action, next state tuple.

The required information can be provided by a dynamics embedding, given that its structure is informative about distinguishing the dynamics of the different environments. Hence assuming we could find optimal parameters for the transition model and the dynamics encoder wrt. the prediction loss, we know that the dynamics embeddings provide the appropriate information to distinguish between the environments. Using this insight and the fact that we can easily compute gradients of the transition model's loss not only wrt. the its parameters but also wrt. the dynamics encoder's parameters, we can use standard deep learning optimizers to find good parameters for the models. Although we are usually not able to find global optima in practice, the combined optimization of transition model and dynamics encoder is robust and leads to informative end even understandable latent space layouts, as reported in Chapter 6.

4.3. Architectures for Dynamics Encoders

In Section 4.2, describing our PUDM-RL approach, the architecture choice for the transition model, and, more specifically, the dynamics encoder has been left open. For the transition model, we use a DNN (hyper-parameters given in Appendix B) because the structure of the inputs is fixed. We concatenate the state, action, and dynamics embedding to form an input vector. The dynamics encoder, on the other hand, needs to cope with sequences/sets of past interactions and hence requires a more flexible structure. We take a middle ground and choose to view the past interaction data as a set of sequences. In comparison, the authors of [44] take a completely rigid view on past interaction data, assuming a fixed size and to some degree unrelatedness between individual interactions. Similarly, in [41], the authors view the interaction data as a set and hence assume temporal unrelatedness. While the latter assumption is fine for the fully observable case (which is the case for the evaluations the authors conduct in simulation), it is not suited to cope with even slight amounts of partial observability introduced through, e.g., observation noise. Observation noise is problematic for the fully set-based view because it renders temporally proximate states (observations) dependent.

Accounting for the possibility of observation noise, we believe there should at least be a sequential (and hence RNN-based) part in the dynamics encoder. Figure 4.5 depicts a simple recurrent dynamics encoder that takes one big sequence of interactions as the input. This fully sequential dynamics encoder is inferior to the set of sequences based dynamics encoder depicted in Figure 4.6 in three major ways:

- VANISHING GRADIENTS Despite LSTMs being developed to tackle the vanishing gradient problem, it is well known that long sequences (in the order of hundreds of steps) still pose a problem
- PARALLELISM RNNs are hardly parallelizable because each step depends on the previous one. Hence computation time scales badly with sequence length. In the case of a set of sequences, we can trivially parallelize the computations and exploit the massive parallelism available in modern Graphics Processing Units (GPUs)

- VERSATILITY With a single input sequence, ingesting data from multiple disjunct trajectory parts (e.g. different episodes) complicates the problem as the network needs to learn to understand when a boundary is signaled to it (e.g. by a binary *done* flag). With multiple sequences, we are free to input parts of trajectories as we see fits, as long as they are sampled from the same environment. This flexibility can, e.g., be helpful when applying our approach to a real system, where only parts of a trajectory might be usable but where the experiment can be repeated multiple times. For example, imagine a drone that is held, shortly released, and then caught again to prevent it from crashing. In this case, we can identify the parts where the dynamics are not influenced and aggregate them through the set of sequence-based dynamics encoder.

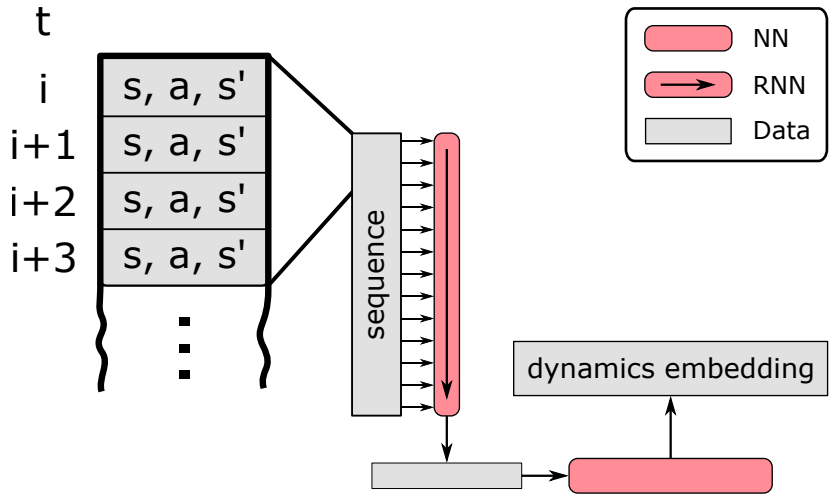


Figure 4.5.: Dynamics encoder implemented by a single RNN in combination with a fully connected neural network for dimensionality reduction towards a desired shape for the dynamics embedding.

Both dynamics encoder structures have in common that the hidden state of the RNN is fed into a fully connected neural network to produce a dynamics embedding that can have a different dimensionality than the hidden state. In practice, we use a single fully-connected layer with a linear activation function to reduce the dimensionality of the hidden state. The set of sequence-based *multi-sequential* dynamics encoder depicted in Figure 4.6 takes the hidden state of the sequence encoders as an input to the backbone aggregator. The aggregator can, in theory, be any network structure that can take a set as an input (like, e.g., the approach used in [41] based on the aggregation of independent factors). However, in practice, we found that using a second RNN, which is trained to be permutation invariant by randomizing the order of the hidden states it receives as an input during training, works well (cf. Section 6.3).

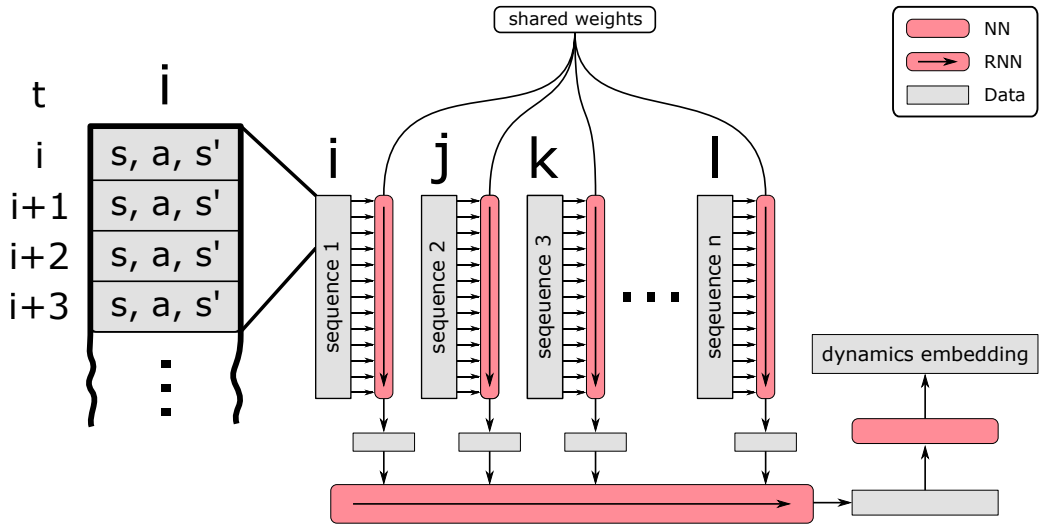


Figure 4.6.: Dynamics encoder implemented by a hierarchy of RNNs, where the upstream RNNs receive (possibly) independent sequences to output a single hidden state, which is in turn aggregated by the backbone RNN. The hidden state of the backbone RNN is reduced in the same way as in the case of the single sequential dynamics encoder depicted in Figure 4.5.

4.4. Algorithm

As described earlier, one goal of our approach is to decouple the training of the inference of the environment from the training of the agent's behavior. To accomplish this decoupling, we need to solve the chicken and egg problem described in Section 4.2. We can tackle this problem by exploiting the availability of a simulator where we have control over the environment parameters.

While in Section 6.2, we show that a single agent can not be used to collect rich data, when the distribution over dynamics parameters is too wide, we can use dedicated agents for particular environments (or alternatively for a set of environments from a narrow dynamics distribution). Another alternative is to use a single multi-task/oracle policy-based agent, which receives the ground truth dynamics parameters. As can be seen from the results of our experiments in Chapter 6, the latter approach turns out to be much more sample efficient and is hence the strategy that we use. With rich data (where, by rich, we mean data that contains our desired behavior and covers the distribution over environments), we can train the dynamics encoder in an unsupervised way. The trained dynamics encoder can then be used to infer a current environment's dynamics during the training of the agent.

This procedure is described in Algorithm 1 on a high level. The algorithm allows for different data collection strategies, two of which are described in Algorithm 2 and 3.

Algorithm 1 Full Training Process

procedure FULLTRAININGPROCESS
  Input: Distribution over environments P
  Output: Dynamics encoder g, policy π
    $data \leftarrow$ DATACOLLECTION(P)
    $g \leftarrow$ UNSUPERVISEDTRAINING($data$)
    $P_{embedded} \leftarrow$ EMBEDENVIRONMENTDISTRIBUTION(P, g) ▷ Modify environments to incorporate g
    $\pi \leftarrow$ REINFORCEMENTLEARNING$\left(P_{embedded}\right)$
    return g, π

Algorithm 2 Data Collection: Individual Agents

procedure DATACOLLECTION
  Input: Distribution over environments P
  Output: Data $data$
    $data \leftarrow \{\}$
    while $|data| < threshold$
      $env \leftarrow$ SAMPLE(P)
      $data_{env} \leftarrow$ REINFORCEMENTLEARNING(env)
      $data \leftarrow data \cup data_{env}$
    return $data$

Algorithm 3 Data Collection: Multi-Task/Oracle

procedure DATACOLLECTION
  Input: Distribution over environments P
  Output: Data $data$
    $envs \leftarrow \{\text{SAMPLE}(P) \mid i \in 1 \ldots threshold\}$
    $envs \leftarrow$ AUGMENTENVIRONMENTS($envs$) ▷ Augmentation as described in Section 3.2
    $data \leftarrow$ REINFORCEMENTLEARNING($envs$)
    return $data$

5. Experiments

In this chapter, we will describe the setup for the experiments we are conducting to validate the claims and theoretical results stated in the previous chapters. To be able to conduct experiments with Domain Randomization (DR), we modify the common Pendulum and Acrobot environments contained in the OpenAI gym package [48] and replicated as part of the *ReinforcementLearning.jl* package for usage with the Julia programming language [82].

5.1. Pendulum

The Pendulum is the simplest common continuous control environment, having only a single Degree of Freedom (DoF). The goal of this environment is to swing up and stabilize the pole in the unstable top position after it has been initialized with a random position and speed. The Pendulum is not underactuated in the sense that the number of actuated joints is lower than its number of DoFs, but using the standard mass setting $m = 1\text{kg}$, the torque available at the center rotary joint is not high enough to directly drive the Pendulum upwards, when it is hanging downwards with zero energy.

We modify the environment to enable DR over the mass of the pole, and we introduce an action inversion parameter that toggles the multiplication of the actions given by the agent by -1 or 1. We include action inversion to examine how our agents can cope with drastic changes to the environment's dynamics. In Figure 5.1, the different environments are depicted.

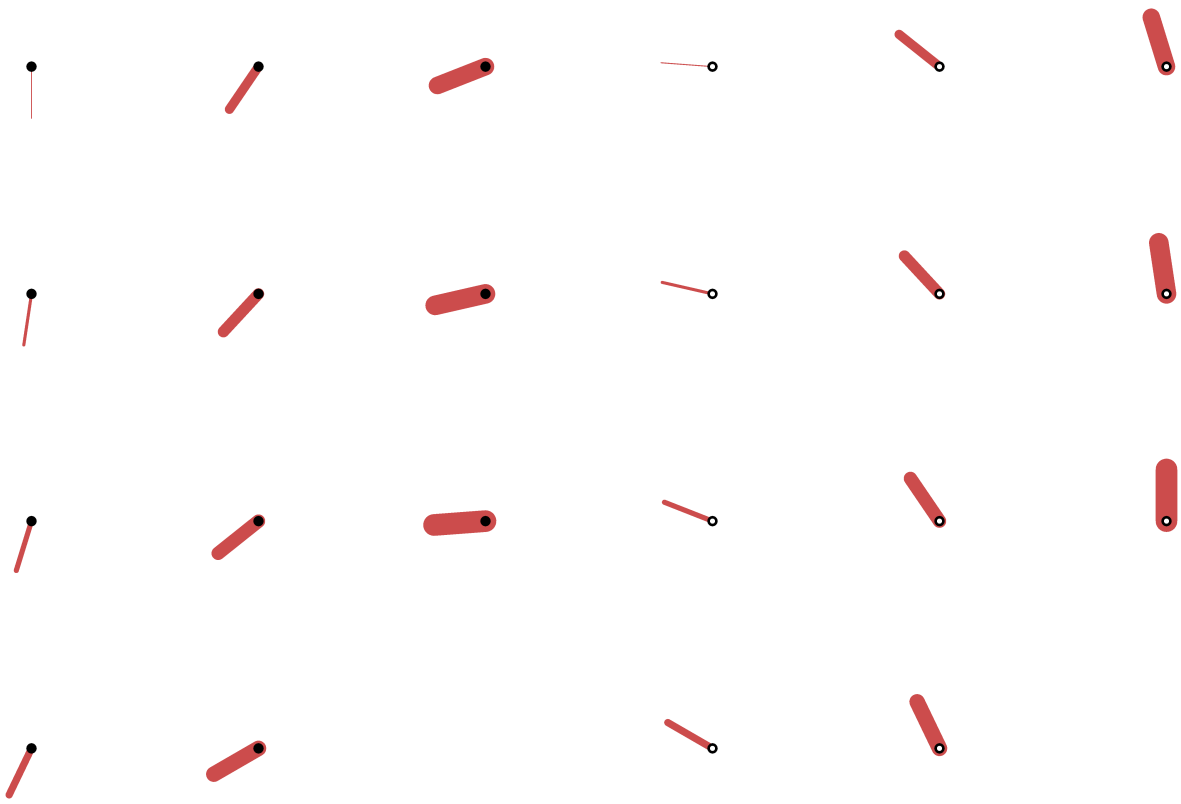


Figure 5.1.: Modified Pendulum environment with masses $m \in \{0.1, 0.3, 0.5, 0.7, 0.9, 1.1, 1.3, 1.5, 1.7, 1.9, 2.1\}$kg and action inversion $a_{inv} \in \{\text{true}, \text{false}\}$ (action inversion is indicated by the presence of a white dot in the actuated link).

5.2. Acrobot

Compared to the Pendulum, the Acrobot is a more complex environment consisting of two links and hence exposing two DoF. By default, the second joint (the one connecting the links) is controlled by discrete actions $a \in \{-1, 0, +1\}$Nm, and the goal is to accumulate enough energy to swing the tip of the second link above a certain height. We adapt the environment to continuous actions $a \in [-10, 10]$ Nm and change the goal to swing-up and stabilization in the top position. Experimentally, we found the following deterministic reward function to capture this goal well:

$$r(s, a) = \exp(-(\theta_1 - \pi)^2) + \exp(-(\theta_2)^2) + 0.2 \exp(-\dot{\theta}_1{}^2) + 0.2 \exp(-\dot{\theta}_1{}^2) + 0.3 \exp(-a^2)$$
$$s = [\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2]$$

Where θ_1 and θ_2 are the joint angles and $\dot{\theta}_1$ and $\dot{\theta}_2$ are velocities respectively (joint angles are normalized to $[-\pi, \pi]$). We also adapt the reset behavior to set the initial state of the Acrobot to states sampled uniformly from the state space.

For DR, we vary the masses of the two links and also introduce an action inversion parameter in the same way we did for the Pendulum environment (cf. Section 5.1). The different environment variations are depicted in Figure 5.2.

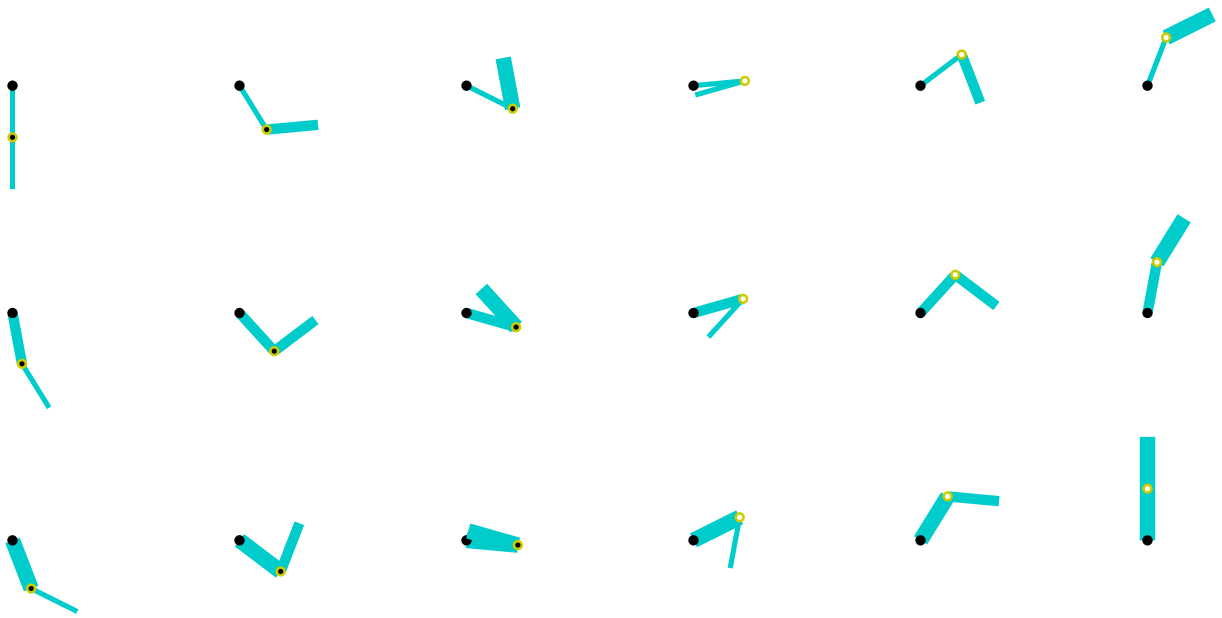


Figure 5.2.: Modified Acrobot environment with link masses $m_1, m_2 \in \{0.5, 1.0, 1.5\}$kg and action inversion $a_{inv} \in \{\text{true}, \text{false}\}$ (action inversion is indicated by the presence of a white dot in the actuated link).

6. Results

In this chapter, we will present results to empirically motivate and evaluate the ideas presented in Section 4.2. First, we will show how our initial Reinforcement Learning (RL) algorithm choice Proximal Policy Optimization (PPO) (cf. Section 3.1.3) fails in the simplest multi-task RL setups. We view successful multi-task RL as a necessary condition to move to the more challenging Meta-Reinforcement Learning (meta-RL) case. After examining more literature, we settle for Soft Actor-Critic (SAC) (cf. Section 3.1.4) and find that it works well in the multi-task setup. In comparison to PPO, SAC is off-policy and hence more sample efficient but also more compute-intensive.

As a motivating example for our Partially Unsupervised Deep Meta-Reinforcement Learning (PUDM-RL) approach, we show that when the variance in the distribution over environment dynamics is high, plain Domain Randomization (DR) (with a conservative policy) leads to suboptimal agents. For this example, we use the Acrobot environment with DR over both link masses.

To improve the turnaround times, we started the evaluations of our PUDM-RL approach using the simple Pendulum environment. Using data gathered from training multiple agents (or alternatively a single or a few multi-task agents) over a wide distribution of environment dynamics, we train a dynamics encoder according to the architectures described in Section 4.3. We find that despite that we do not enforce any latent-space structure, the latent spaces learned using unsupervised training are disentangled representations of the dynamics parameters. In low dimensions, we find that the dynamics parameters are also human-interpretable.

Using the learned embeddings, we show that we can train agents that exhibit almost identical performance to agents trained with ground truth domain parameters available (oracle/multi-task setup in Table 4.1). We show this comparison for the Pendulum and Acrobot environments described in the previous Chapter 5. Furthermore, we investigate the impact of observation noise on the dynamics encoder and find that it encodes not only a point estimate of a representation for the environment's parameters but also uncertainty and hence a full distribution.

Note that it is relatively simple to show that a machine mearning/RL-based approach works, given successful experiments, but that it is notoriously hard to show that a particular approach does not work because there is always the possibility that by changing some implementation detail or hyperparameters it could be made to work successfully. We use our custom implementations (in Julia for the RL part and in Python/Tensorflow for the dynamics encoder part), which is beneficial for performance and by having a structure that is tailored for our PUDM-RL approach but we also acknowledge our fallibility. When we observe unexpected underperformance of a certain approach, we assume it is our failure and fall back to stable third-party implementations to validate our results.

The hyperparameters used for the experiments can be found in Appendix B.

6.1. Multi-Task Reinforcement Learning: PPO vs. SAC

When first implementing PPO and applying it to the Pendulum and a quadrotor drone stabilization task, we found that it produces great stable policies. Based on these results, we moved to a multi-task setup (cf. Section 3.2) where the agent receives the ground truth domain parameters. To our surprise, PPO did struggle to learn good policies in even the simplest multi-task cases (like the Pendulum with only action inversion). To validate that this unexpected bad performance is not due to some mistake in our implementation, we verified our results using the PPO implementation of the popular *stable-baselines3* collection of RL algorithm implementations [83]. To our surprise, their PPO implementation was not even able to learn the ordinary Pendulum task (without any DR). We modified their implementation to incorporate the insights from Section 3.1.1. In the following section, we show that the theoretical issue described in Section 3.1.1 actually leads to a practical problem. After that, we use the modified PPO algorithm to show that in the multi-task setup, PPO still underperforms SAC.

6.1.1. Time Limited Gym Environments are Not Markovian

In Figure 6.1 we compare the learning curves of the modified and unmodified PPO from the *stable-baselines3* collection [83]. We only incorporated a minimal modification, similar to the code snippet from Section 3.2. We can see that it has a drastic effect on the agent's performance and hence validates our claim that the issues with time-limited RL environments should get more attention.

After roughly 1M training steps, the performance of the agent drops again, which is a common feature of PPO that does not pose a big challenge in practice because we can just restore a good checkpoint.

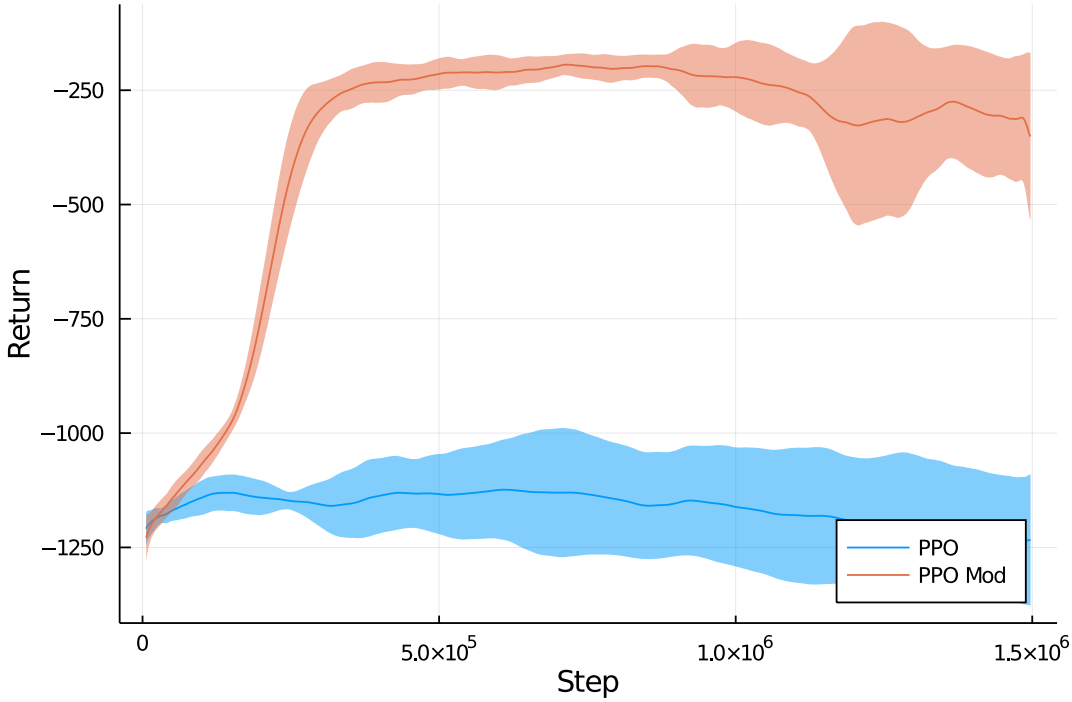


Figure 6.1.: Comparison of the standard PPO algorithm from the *stable-baselines3* collection [83] with our modified version, incorporating the insights from Section 3.1.1. The training is done for 1.5M steps in an unmodified Pendulum environment (no DR etc.). The plot shows mean and standard deviation over 10 runs with different random seeds for each of the algorithms. After converging to a good policy, the training starts to destabilize, which is a well-known issue of PPO. In practice, the destabilization does not pose a big problem because we can revert to a good checkpoint of the policy.

6.1.2. PPO vs. SAC in Multi-Task Reinforcement Learning

Using the modified PPO implementation we validated to be working in the normal Pendulum case, we can move to the multi-task RL case. For this demonstration, we use the modified Pendulum described in Section 5.1 but use the action inversion parameter as the only mode of variation. We use a multi-task setup, which means that the agent receives ground truth information about the environment (in this case, a Boolean, signaling if the actions are inverted or not).

In Figure 6.2, we show the results of training the modified PPO and a modified (same modification) SAC from the same collection [83] of RL algorithms. The negative reward scale makes it a bit hard to see the huge difference in performance between the two algorithms. When visually inspecting the policies, we can see that the agent trained using SAC is able to fulfill the task, while the one trained using PPO is not.

We initially chose PPO as the RL algorithm to be used for our work, particularly because we are only training in simulation, and hence the sample efficiency is not a big issue. The reason to accept the sample inefficiency of PPO was that PPO is otherwise considered a very stable algorithm with a rather simple implementation and hyperparameter settings (like the clip range) which work across many different tasks. After observing the unexpected underperformance of PPO in the multi-task RL case and finding further hints in the literature for PPO performing badly in multi-task RL setups [31], we switched to SAC for all following evaluations.

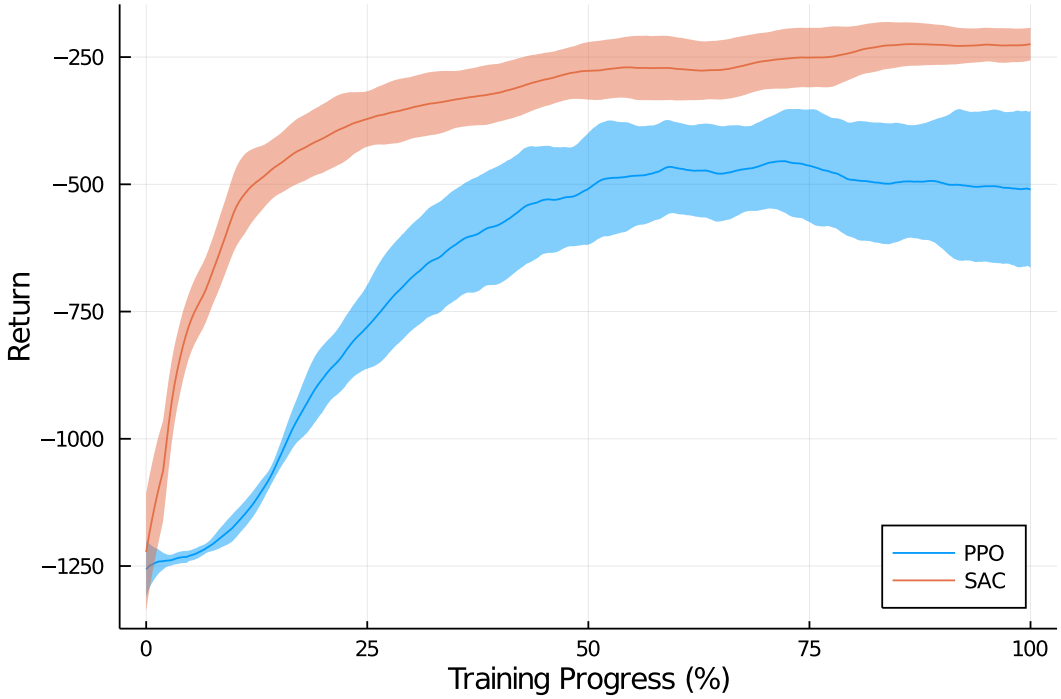


Figure 6.2.: Comparison of SAC and PPO on a multi-task Pendulum (varying in the action inversion parameter). The agents receive the ground truth action inversion signal and hence should be able to act well in both environments. The comparison is over the relative training progress because to have a fair comparison, we train PPO using 1.5M training steps (because it is an on-policy, sample inefficient algorithm), while in the case of SAC 250k steps suffice. The plot shows mean and standard deviation over 10 runs with different random seeds for each of the algorithms.

6.2. Motivating Example: Domain Randomization Fails for Broad Distributions

In the previous sections, we claimed that plain DR (conservative policy) violates the Markov assumption (Section 4.1) and that hence broad distributions over environments lead to a degradation in training performance. In the following, we demonstrate this effect using the Acrobot environment (cf. Section 5.2).

We narrow down the distribution over environments by excluding the action inversion parameter and hence only randomize across the two link masses. As a baseline, we train individual agents for all combinatorial configurations. These individual agents each just act in one particular environment and hence form a baseline for the maximum return. For comparison, we train 5 agents (different random seeds) in a plain DR setup (using a conservative policy). During the training of these agents after each episode, a new environment is sampled from the distribution without informing the agent about the current environment at any point. Figure 6.3 shows the aggregated learning curves for the two different cases. We can observe a big gap between the performance of individual agents and the DR-based ones.

Visually examining the behavior, we find that the individual agents manage to swing up and stabilize the Acrobot in all cases, whereas the DR-based agent only manages to swing up and stabilize in some of the environments while just rotating in the other ones. When we also include the action inversion parameter, the difference becomes even bigger, as we show in the following.

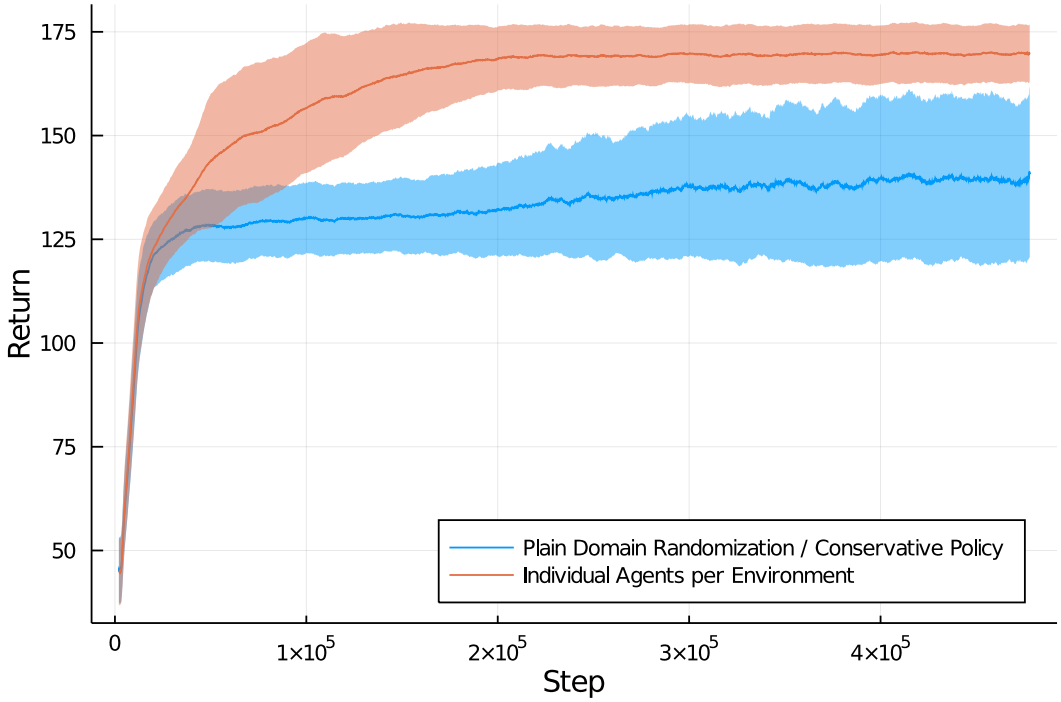


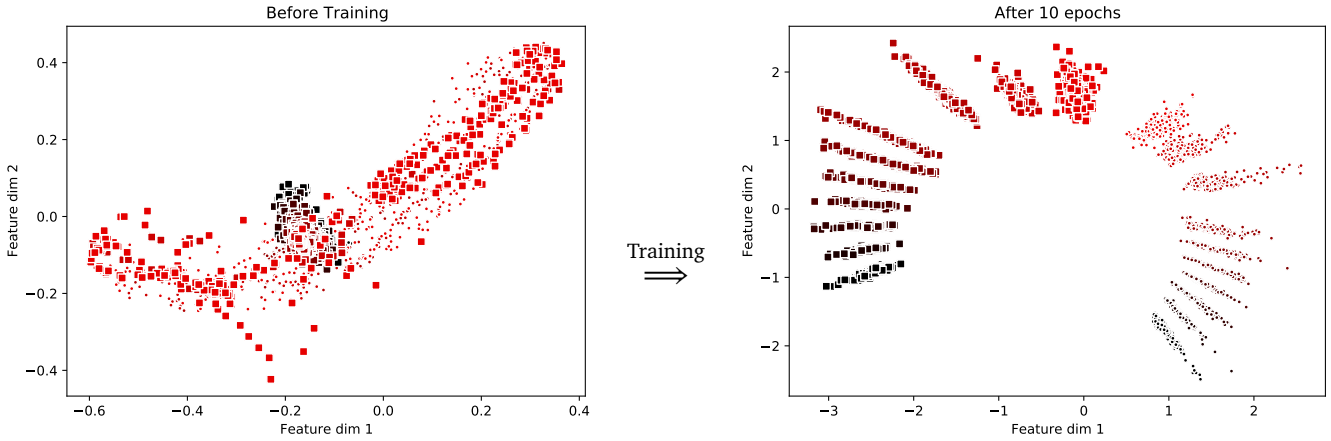
Figure 6.3.: Comparison of the performance of individual agents per environment vs. a single agent based on a conservative policy with DR in the Acrobot environment. The horizontal axis shows the number of interaction steps with the environments. The plot shows mean and standard deviation over 5 runs with different random seeds in the plain DR case, while the performance of the individual agents is aggregated over 9 runs with one individual agent for each of the 9 combinatorial environments.

6.3. Dynamics Encoder Training

We could use the data gathered during the training of the individual agents in Figure 6.3 to train the dynamics encoder following the maximum likelihood scheme described in Section 4.2. In practice, this takes unnecessarily much training time, and hence we use a multi-task setup, where a single agent receives the ground truth domain parameters. It turns out that this approach is much more sample-efficient as we only need to train a single agent to have a good performance which in our observations (for the used environments) exhibits a large amount of positive transfer. This can be seen by comparing Figure 6.3 with Figure 6.11, where the multi-task agents train almost as fast as individual agents, and hence it takes roughly 20 (number of distinct environments) times more samples to train individual agents.

Figure 6.4a shows the latent space before the training (after random initialization) of a sequential dynamics encoder (structure according to Figure 4.5), while Figure 6.4b shows the result after ten epochs of training. We changed the mass distribution to a reciprocal sequence because we observed that the dynamics encoder lays out the latent space in a semantically meaningful way. By semantically meaningful, we mean that distances in the latent space correspond to the magnitude of the environment's response to a certain torque. Using the reciprocal mass distribution consisting of masses $m_i = 2/(2i)\text{kg}$ with $i \in 1 \dots 10$, we get a sequence of accelerations that is linear in i for a constant torque a: $a \propto m_i \ddot{x} \rightarrow \ddot{x} \propto a/m_i \propto ai$. In addition to that, the

latent space is also incorporating the action inversion in a semantically meaningful way, by mirroring the embeddings of environments with the same mass along an axis through the origin.



(a) (b)

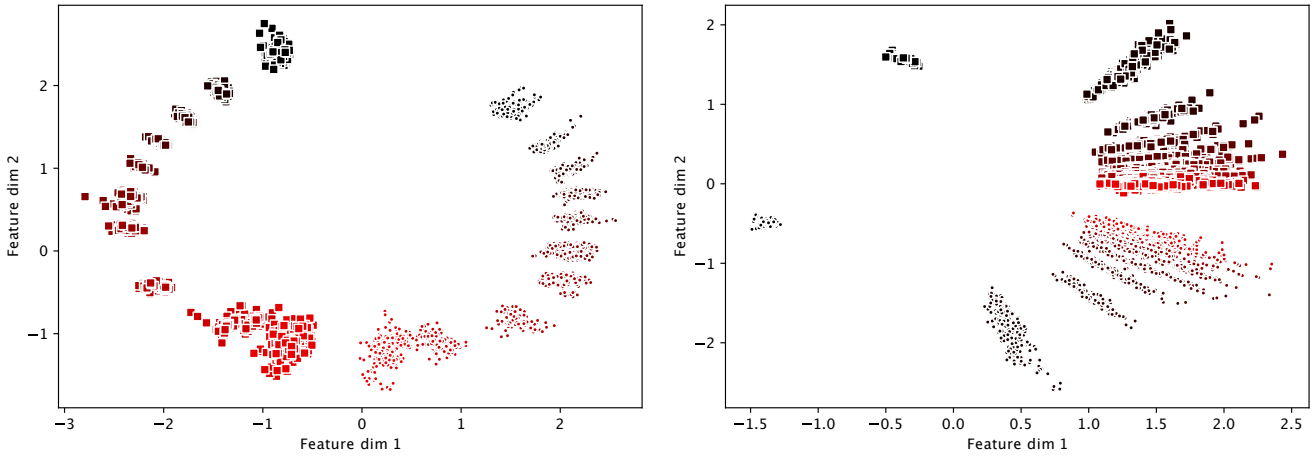
Figure 6.4.: Comparison of trajectories embedded into the latent space before and after training of a sequential encoder (structure like depicted in Figure 4.5). The environment is a Pendulum as described in Section 5.1 but with the mass distribution changed to: $m \in \{2/(2 * i) \mid i \in 1 \dots 10\}$kg. Each point characterizes a trajectory in a particular environment, with a certain mass and action inversion setting. The mass of the Pendulum is characterized by the redness (reciprocal, for better usage of the dynamic range), and the action inversion is characterized by the shape of the dots (small marker: actions inverted, big marker: actions not inverted). Each dot is the result of passing trajectories of length 80 steps without observation noise into the dynamics encoder.

When adding observation noise, we observe that the dynamics encoder better incorporates uncertainty and concentrates the embeddings more in the fully informed case. With fully informed, we refer to the case where the dynamics encoder receives sequences of the maximum length it has witnessed during training. During training, the dynamics encoder is faced with sequences of varying length to prepare it for application in the RL loop, where at the beginning of an episode, it starts with zero steps. Figure 6.5a shows a latent space layout after training the same setup depicted in Figure 6.4 but with additional observation noise using a standard deviation of $\sigma = 0.05$ (data is standardized). Additionally, in Figure 6.5b, the latent space of a dynamics encoder for the original Pendulum setup (with uniform, non-reciprocal masses) is shown. In comparison to the setup with reciprocal masses shown in figure Figure 6.5a and Figure 6.4, the uniform masses are unevenly distributed due to the difference in effect they have on the dynamics of the system. For example, switching from $m = 0.1$kg to $m = 0.3$kg is a threefold increase in acceleration, assuming a constant torque, while moving from $m = 0.3$kg to $m = 0.5$kg is only a $1.\overline{6}$ fold increase in acceleration.

To us, it is striking that (especially in the deterministic case) the gradient descent-based training procedure, in combination with neural networks as capable function approximators, leads to such simple and semantically plausible solutions without enforcing any structure a priori. We tried enforcing normal distributions in the latent space, using an additional Kullback-Leibler divergence based term and the reparameterization trick (similar setup to Variational Autoencoders (VAEs) [67]) but did not obtain better results (neither in the prediction loss of the transition model nor when visually inspecting the resulting latent space).

Instead of explicitly inducing structure, we observe that the combination of a general model structure and the optimization process seems to yield results that align with Occam's razor. We believe that there is a connection

to the double descent and benign overfitting phenomenons and are hopeful that future research will shed light on the reason why this approach leads to such good results in so many different fields.



(a) (b)

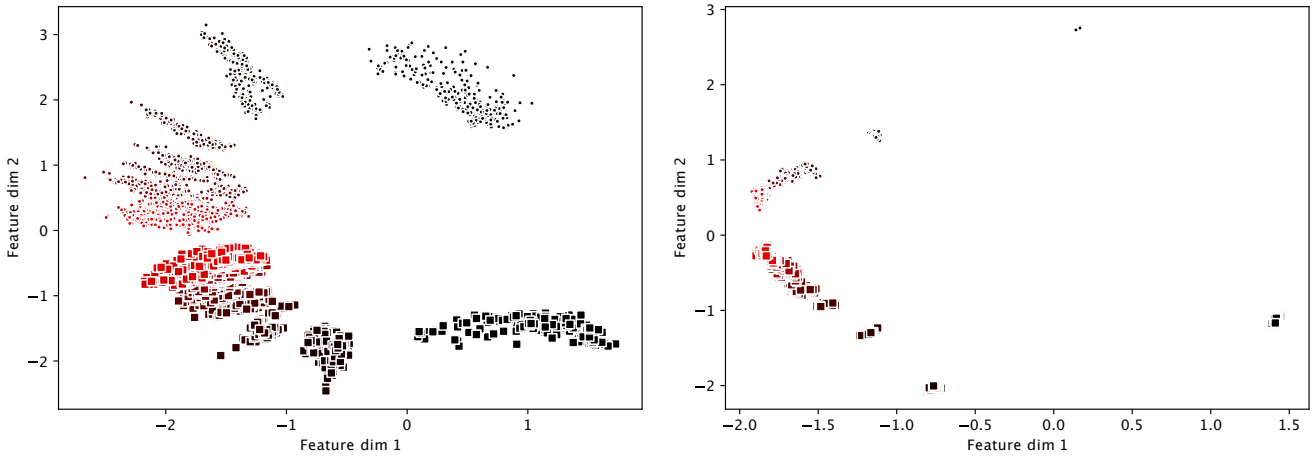
Figure 6.5.: The two plots show latent spaces resulting from a setup based on the setup shown in Figure 6.4. In Figure 6.5a, artificial Gaussian observation noise of standard deviation $\sigma = 0.05$ has been added during the training process. This additional noise leads to better handling of uncertainty, where in the fully informed case, the variance is lower and used to encode uncertainty. Figure 6.5b shows the case of the original (non-reciprocal) mass distribution, described in Section 5.1. We can observe that the learned latent space layout reflects the semantic analysis, that the difference in dynamics between the lower masses are higher than between the higher masses.

The two modifications in Figure 6.5 are combined in Figure 6.6a. The resulting latent space combines the features of more noisy clusters and a non-uniform distribution of the clusters due to the difference in the effect of the masses.

In Figure 6.6b, the same Pendulum setup (masses as described in Section 5.1 and Gaussian observation noise with $\sigma = 0.05$) is shown for the multi-sequential dynamics encoder (cf. Section 4.3 and Figure 4.6). Compared to the purely sequential dynamics encoder, the multi-sequential encoder produces a visually similar latent space structure with a similar separability while concentrating the clusters a bit more. Given the similarity and the benefits highlighted in Section 4.3, we chose to conduct further experiments using the multi-sequential dynamics encoder.

In Appendix A, we visualize the discovery process of the different environments during an episode. We can observe that the dynamics encoder starts by emitting a prior embedding that is identical for all environments when it has no experience available at timestep 0. With subsequent interactions, the uncertainty in the environments quickly decreases, which is expressed by the dynamics encoder through the concentration of the environment embeddings to a particular region for each environment.

In general, when training the varying structures and configurations of dynamics encoders, we observe that the training process is very stable, in the sense that the resulting latent space layouts are very repeatable for different random seeds (discounting for rotations, which tend to be arbitrary).



(a) (b)

Figure 6.6.: The scatter plot in Figure 6.6a shows a latent space resulting from training the purely sequential dynamics encoder (block diagram in Figure 4.5) using a Pendulum environment with the original mass distribution, described in Section 5.1, with additive Gaussian observation noise of $\sigma = 0.05$. This setup is transferred to the multi-sequential dynamics encoder (structure depicted in Figure 4.6), and the resulting latent space is depicted in Figure 6.6b.

6.4. Full Training: Pendulum

Using the trained dynamics encoder described in the previous section, we can train an agent using RL (cf. Line 7 of Algorithm 1). The agent receives information about the current environment through the dynamics embeddings (arrow from the dynamics embeddings to the agent in Figure 4.4). Figure 6.7 shows a plot comparing the learning curves for three different approaches. We can observe that the plain DR approach (using a conservative policy, as described in Table 4.1) performs very badly and is not able to learn a good behavior because it is not aware of the current environment. Particularly the variation between environments multiplying the actions taken by the agent by -1 and 1 makes it impossible for a context unaware, conservative policy to exhibit good behavior. In comparison to the plain DR approach, our PUDM-RL approach learns a good behavior using the information from the dynamics encoder very quickly and attains similar returns to the multi-task learning setup, where the agent receives ground truth information about the environments domain parameters.

Inspecting the behavior of resulting policies visually, both our approach and the multi-task RL-based approach, learn to swing up and stabilize the Pendulum in the upward position across all the environments variants.

In addition to the previously mentioned approaches, we also compare PUDM-RL against using a recurrent policy (and critic). We expected at least a noticeable benefit in using a recurrent setup over plain DR, but we were not able to observe that in the first 500 episodes (equating to $500 \cdot 200 = 100$k steps). Because we were not able to achieve a noticeable benefit with our own recurrent SAC implementation, we conducted additional evaluations using the *SAC-v2-LSTM* implementation from [84], which is one of the few collections of RL algorithm implementations that features a recurrent version of SAC.

Integrating recurrence into off-policy actor-critic algorithms is not as straightforward as in the case of on-policy

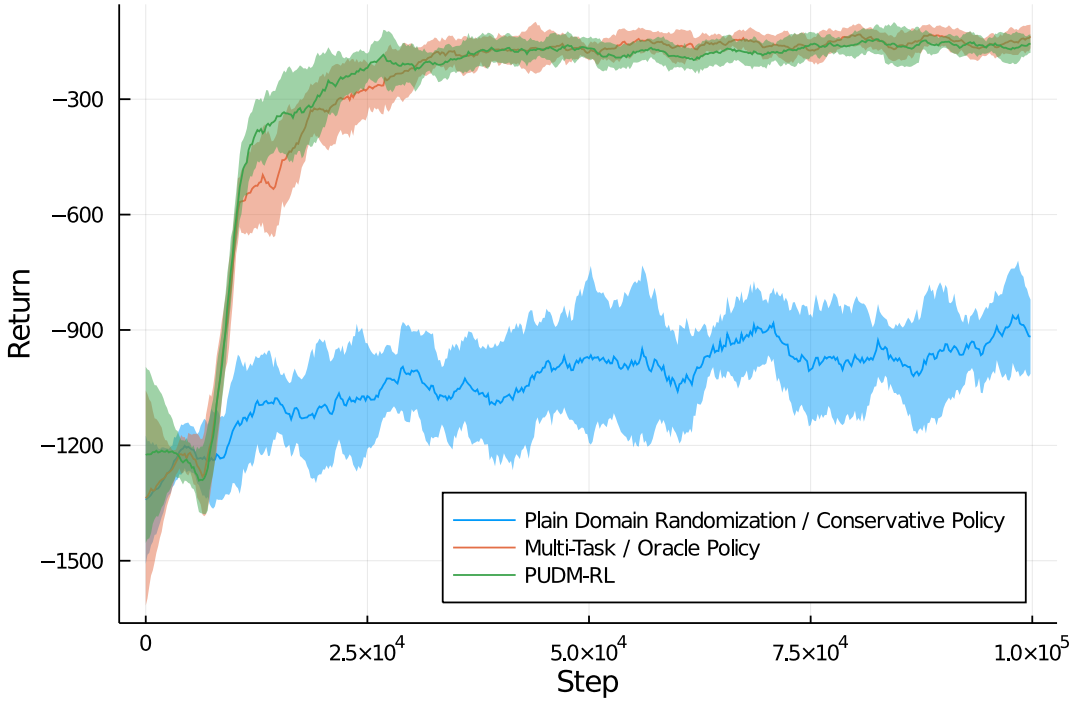


Figure 6.7.: Comparison of different approaches to the Pendulum task defined in Section 5.1. In the case of the Pendulum, each episode is 200 steps long, so each training run of 500 episodes consists of $500 \cdot 200 = 100$k interaction steps in total. Each training approach has been run ten times with different random seeds.

algorithms because there are different ways to modify the replay buffer to support sequences of transitions. According to the taxonomy in [85], our approach uses the *Bootstrapped Random Update*, where the hidden state is reset for each sequence, and the Recurrent Neural Network (RNN) is then trained by unrolling a fixed number of steps. The major advantage of this approach over the alternative *Bootstrapped Sequential Update* is the ability to sample decorrelated (sub-)sequences.

In [42], two more elaborate schemes to cope with sequences in the replay memory are proposed, a *stored-state* and a *burn-in* strategy. While these strategies can be mixed and matched, in the *SAC-v2-LSTM* implementation of [84], a notion of the *stored-state* strategy without *burn-in* is used.

The results using *SAC-v2-LSTM* are depicted in Figure 6.8 and are equally discouraging as the results from our implementation. We ran the training of the recurrent model for longer than the multi-task and PUDM-RL models because after visually inspecting the behavior of the latter ones, we were certain that they converged to a good behavior and that it is very unlikely for the policy to improve performance with more training. Likewise, for the plain DR case, where the policy is conservative and assumes the Markov property, while it is actually not holding, we do not expect improvements with further training. In the case of the recurrent policy, we were not certain that 100k training steps are sufficient and hence trained the model for longer. However, even with more training steps, the recurrent policy did not yield a noticeable performance advantage. The recurrent policy performing equally bad as the plain DR based one is also in line with the findings in [44], where the authors refer to the plain DR based policy as an *invariant policy* and find that it performs on the same level as an Long Short-Term Memory (LSTM)-based policy.

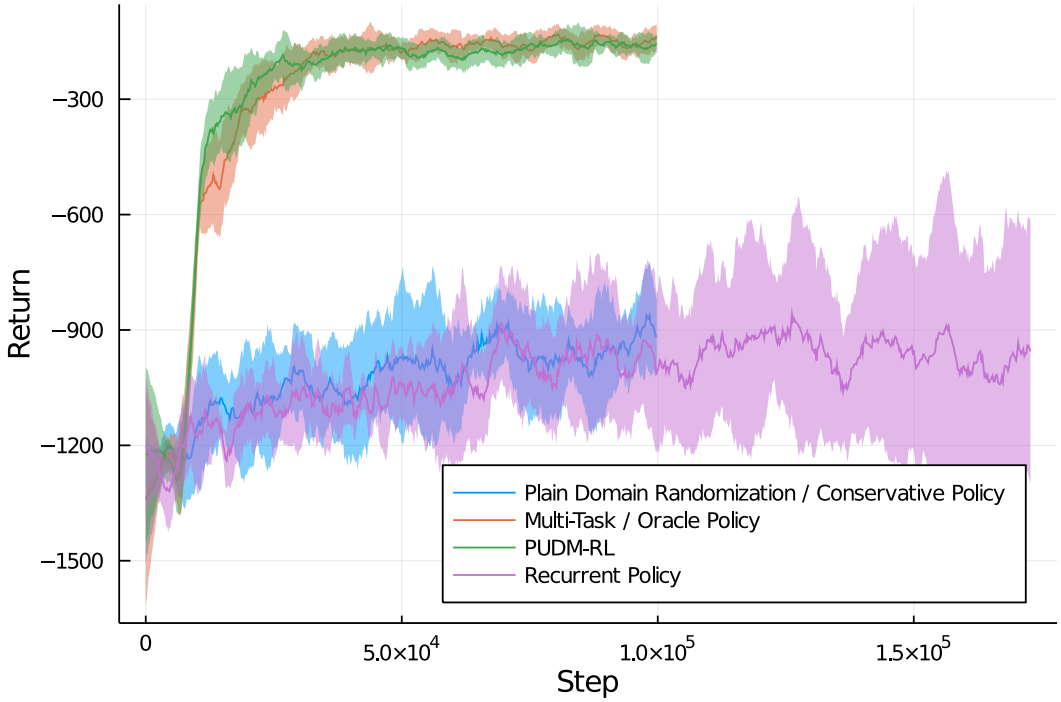


Figure 6.8.: Additional comparison of using a recurrent policy in the Pendulum environment. Training the recurrent policy is very computationally intensive and hence only carried out for six random seeds (instead of ten random seeds for the other approaches).

After more than 150k steps of training, the number of training steps is high enough to comfortably fit the whole training process (cf. Algorithm 1) of our PUDM-RL approach into it. This comparison of the sample complexity is emphasized in Figure 6.9, where the multi-task learning part is used for data collection. Even well before reaching 150k interaction steps, our PUDM-RL approach attains the desired behavior with very high confidence. In practice, the number of samples collected during the multi-task training vastly exceeds the number required to train a well-performing dynamics encoder. We observe that especially the data where the learning curve is rising the steepest is most valuable for the training of the dynamics encoder because, during that period, the learning agent has not yet settled for a single behavior and spans the state-space much more than in the late stage, where the change in behavior (and hence the returns) plateaus.

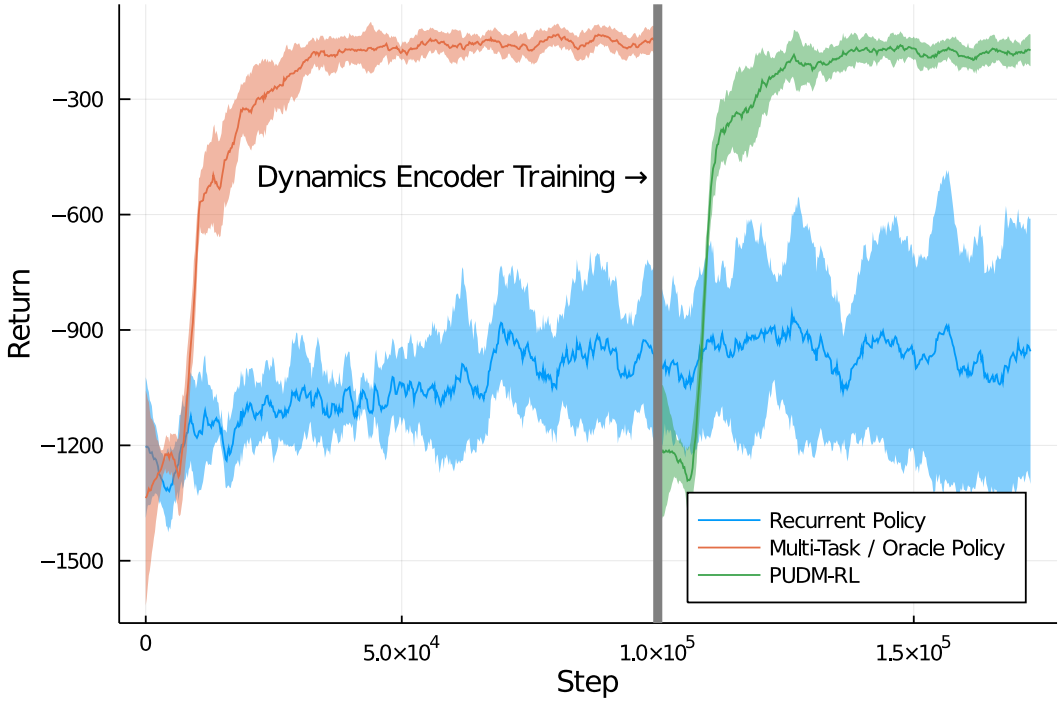


Figure 6.9.: Visualization of the sample efficiency of a full PUDM-RL process. First, samples are collected using multi-task RL (alternatively, e.g., individual agents as described in Section 4.4 can be used). Second, the collected data is used to train a dynamics encoder (unsupervised training, no additional environment interactions needed). Third and finally, an agent, conditioned on the dynamics embeddings from the dynamics encoder, is trained.

6.4.1. Generalization and Extrapolation

In addition to expected returns and visual examination of the resulting behaviors, we also study the generalization and extrapolation capabilities. During training, the agent experiences only a small set of different environments (cf. Section 5.1 for a detailed description), which span a broad range of values. To facilitate simulation-to-reality (sim-to-real) transfer, generalization within the training range is essential because it is unlikely that the real system directly coincides with one of the environments used during training. In Figure 6.10, we compare the resulting performance in a genaralization/test scenario (unseen environments within the range of training environments). We find that the performance does not degrade in the test environments and even exhibits fewer outliers than in the training environments. We hypothesize that the latter effect is caused by the domain parameter values of the training environments, which are on the fringe of the range, and hence are unlikely to be sampled using a uniform distribution.

Additionally, we study the extrapolation behavior, where we test the performance in environments that lie outside of the training distribution. Note that the extrapolation performance is not central to the argument for our PUDM-RL approach because we aim at enabling the training over very broad environment distributions rather than training on narrow distributions and optimizing for the extrapolation towards out-of-distribution samples. For pendula with lower weights than seen during training, the performance is still very good, which we attribute to the effect that for low masses, the Pendulum is fully actuated (in a sense that the available torque is large enough to directly drive the Pendulum upwards). In these cases, the policy can learn to exhibit

the behavior of a PD-controller and extrapolate well to smaller masses. Conversely, for higher masses, we observe a massive drop in performance.

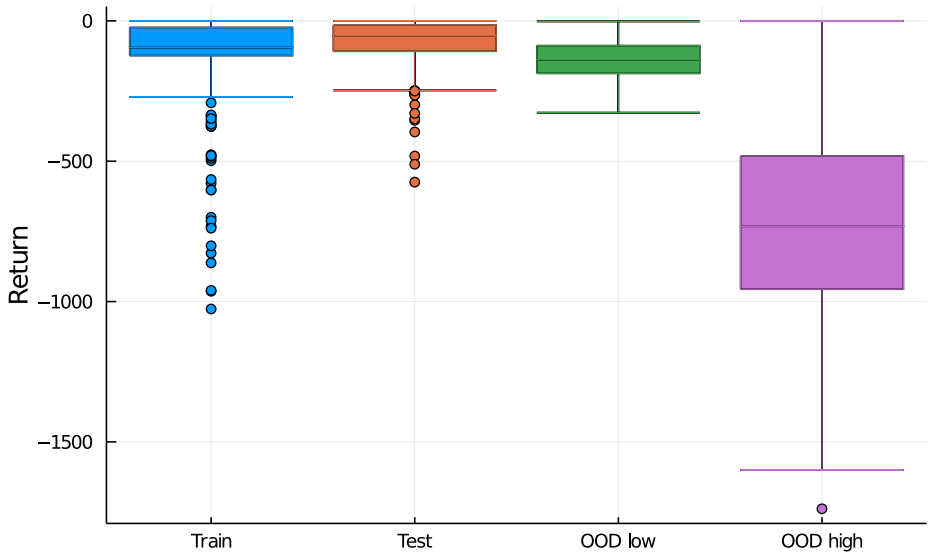


Figure 6.10.: Comparison of the agent's performance in environments with unseen dynamics parameters. The *Train* column shows the distribution of the returns in the 20 environments known from training (20 repetitions each). The other columns are based on 400 randomly sampled, unseen environments from the training distribution (*Test*) and from distributions over masses that are out of the original training distribution - *OOD low* is a uniform distribution over lower masses than the training distribution ($m \in [0.05, 0.1]$kg) and *OOD high* is a uniform distribution over higher masses ($m \in [1, 2]$kg).

6.5. Full Training: Acrobot

With the promising results in the Pendulum environment, we can now move to the more complex Acrobot environment. To make the task even more challenging, we use the setup described in Section 5.2, which in addition to the randomization over link masses for both links, used in the setup described at the beginning of this chapter (in Figure 6.3), includes the additional action inversion parameter that is uniformly randomized to be true or false (50% probability each). This additional mode of variation poses a drastic change to the dynamics (from the perspective of the agent) and deteriorates the performance of plain DR-based agents even more. In Figure 6.11, the wide gap in performance between plain DR and our approach (PUDM-RL) can be observed. We also notice that in contrast to the experiments using the Pendulum (Figure 6.7), our agent performs slightly worse than the multi-task setup, where the agent receives ground truth information about the current environment's dynamics parameters. We hypothesize that at least a share of the difference in performance can be attributed to the dynamics discovery process, which, during training, takes place in each episode because the dynamics encoder starts with zero knowledge after resetting the environment.

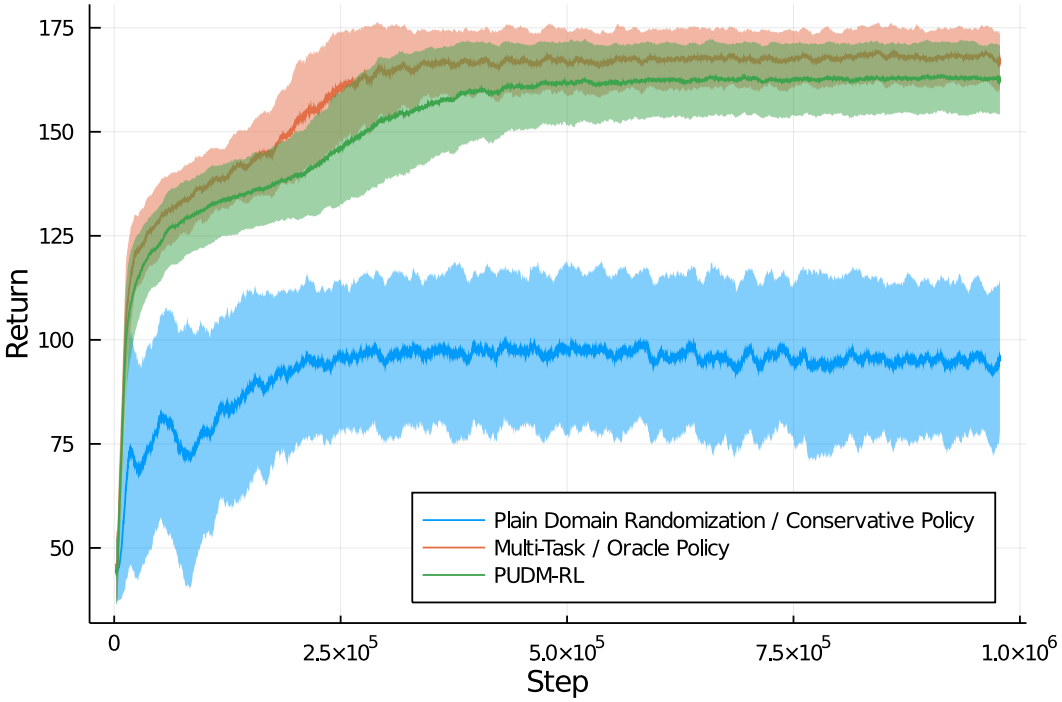


Figure 6.11.: Comparison of different approaches to the Acrobot task defined in Section 5.2. Each approach has been trained with five different seeds over 1M interaction steps.

6.5.1. Generalization and Extrapolation

We also test the agents resulting from the training process depicted in Figure 6.11 in unseen environments, especially because with $m \in \{0.5, 1.0, 1.5\}$kg we use a very sparse sampling of the parameter range for training. For evaluation of the agents, we use deterministic rollouts, where the mean/mode given by the policy is directly taken as the action instead of sampling from the distribution like it is done during training to foster exploration. The mean and mode are identical in the case of the family of distributions used in SAC (squashed Gaussian distributions). Note that the outcome (i.e. the return) is still stochastic due to random initial conditions.

Figure 6.12 shows a comparison of the return distribution of the different approaches under deterministic evaluation. We can observe that all approaches reach higher returns than in the final return in the stochastic case (Figure 6.11). Like before, our PUDM-RL approach underperforms the multi-task RL-based agents slightly while outperforming the plain DR-based agents by a wide margin. More importantly, we can also witness that both the multi-task RL and also the PUDM-RL based agents generalize very well to unseen environments, despite being only trained on a sparse set of samples from the parameter space.

In addition to the generalization capabilities, we also study the performance under extrapolation to out-of-distribution samples, depicted in Figure 6.13. We observe that the multi-task agents once again perform better than our approach, which in turn performs better than the plain DR setup. We view the extrapolation capabilities as less essential for evaluating our approach compared to the generalization performance. We concentrate on the generalization performance to environments that lie within the support of our domain distribution, as this is aligned with our initial premise of creating an algorithm to excel when trained in a broad distribution over environments.

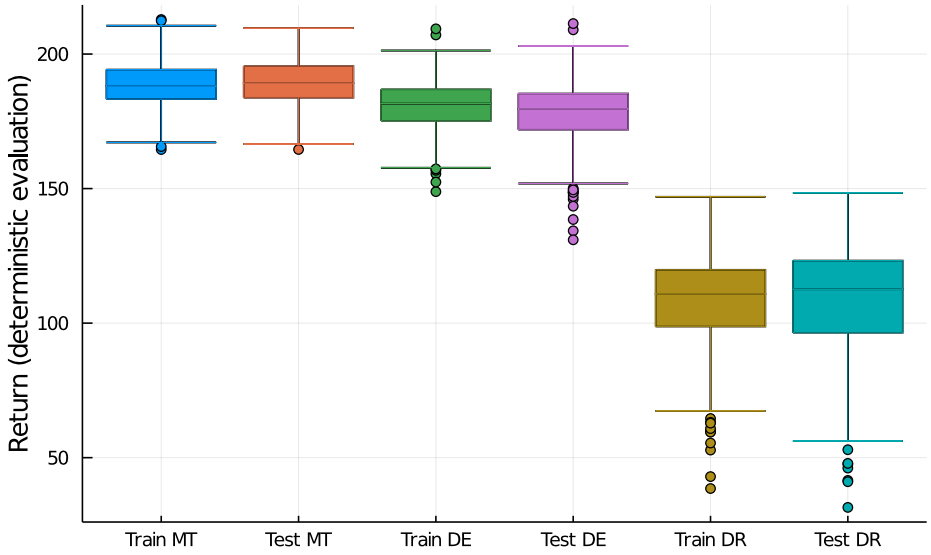


Figure 6.12.: Comparison of the generalization performance of different approaches in the Acrobot environment. We refer to generalization as the capability of performing a learned behavior in unseen environments whose dynamics parameters lie within the range of the training parameters. Actions are chosen deterministically across $18 \cdot 20 = 360$ runs with different random starting positions (and different random environments in the *Test* case).

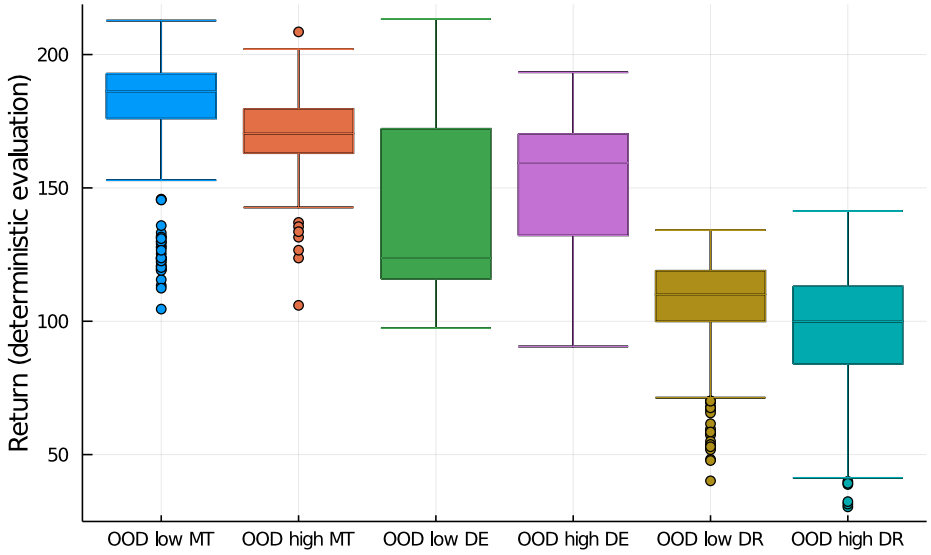


Figure 6.13.: Same evaluation setup as in Figure 6.12 but with out-of-distribution domain parameters. Masses are randomly sampled from a uniform distribution in $[0.25, 0.5]$kg and $[1.5, 2.0]$kg for the *low* and *high* cases respectively.

7. Conclusion & Outlook

In this chapter, we discuss our results and draw a conclusion before we give an outlook for interesting future work that can be based on our approach.

7.1. Conclusion

In this work, we proposed and evaluated a novel approach to deep Meta-Reinforcement Learning (meta-RL), particularly suited to facilitate simulation-to-reality (sim-to-real) transfer using Domain Randomization (DR). We show theoretically why DR (with broad distributions) should be viewed as a meta-RL problem and conduct experiments that show the benefit of doing so. We find that our Partially Unsupervised Deep Meta-Reinforcement Learning (PUDM-RL) approach vastly outperforms plain DR and the usage of a recurrent policy. While plain DR is arguably the simplest setup, we view our PUDM-RL approach as much simpler than the recurrent policy-based one. Looking superficially at the recurrent policy setup, this might look counter-intuitive because one might think that we just modify the model architecture to be recurrent and keep the Reinforcement Learning (RL) algorithm as it is. Particularly in the case of off-policy RL algorithms, a recurrent policy comes with a myriad of issues like *recurrent staleness* and *representational drift/initial recurrent state mismatch* described in [42] and [85] that need to be accounted for. Measures like *burn-in* that aim to tackle the mentioned effects introduce additional hyperparameters and further complexity.

Our approach, on the other hand, comes down to unsupervised training of an Recurrent Neural Network (RNN) with a transition model on top. Unsupervised (sometimes also called self-supervised) training is known to be more stable than RL, mainly because it is possible to calculate end-to-end gradients using backpropagation, whereas, in RL, all the training information has to go through a very noisy gradient estimate based on trial and error. We exploit this property and build our approach on the premise to train as much as possible of our model in an unsupervised way. From our experiments, we can conclude that the training of the dynamics encoder is indeed very stable and leads to informative latent space structures. Furthermore, our experiments show that the compressed information in the dynamics embedding, generated by the dynamics encoder from earlier interactions with an environment, is informative to the RL agent and helps it to achieve good returns across broad distributions over environments, where other (non-oracle) approaches fail.

Additionally, we show that our approach generalizes well to unseen environments that lie within the range of training environments. Even when the range of parameters is only sparsely sampled (like with $m \in \{0.5, 1.0, 1.5\}$kg in the case of the Acrobot), our agent generalizes well to unseen environments. The successful generalization under a low number of environment samples is also an indication that our approach may scale to more modes of variation and we expect it to be less affected by the curse of dimensionality that usually arises with the addition of more randomized parameters. We view the successful generalization as a necessary condition to facilitate actual sim-to-real transfer.

7.2. Outlook

This work presents a foundation for future work, like further studying the PUDM-RL approach, applying it to an actual sim-to-real transfer, extending it to adaptive control, or applying it to meta-RL in qualitatively differing tasks.

Further Studies (in Simulation) Based on the promising results we describe in Chapter 6, we can think of many different avenues to modify the experiments, benchmarking the PUDM-RL approach in different scenarios. In our experiments, we focused on swing-up and stabilization tasks, but it would be very interesting to apply our approach in environments with contact dynamics, e.g., for manipulation tasks. It is well known that contact dynamics are notoriously hard to simulate realistically, and hence we believe this is an area where the *reality gap* might be widest. Applying RL in such tasks is particularly tempting because alternative approaches like dynamic trajectory optimization and model predictive control heavily rely on good models/simulators. In addition to contact dynamics, also soft bodies and (passively) compliant robots exhibit physical properties that are hard to model. For example, in the case of an environment containing soft bodies, the problem of system identification might be ill-posed. In this case, our PUDM-RL approach might help because we can restrict the latent space to a lower dimensionality and hence force the dynamics encoder to encode the more coarse-grained behavior of the complex system.

Simulation-to-Reality Transfer: Drone To validate our approach, we are planning to conduct a sim-to-real transfer to a real quadrotor drone. To facilitate this transfer, we will build on preliminary work on applying RL to a drone simulator. The drone simulator is designed to be crude but very flexible in the number of ways to randomize its dynamics and to be very fast. The first step of the plan is to use this crude simulator to train a single agent over a broad range of different drones (different masses, arm length, motor torque curves, delays) and then attempt a simulation-to-simulation (sim-to-sim) transfer to the *GymFC* simulator [86]. The GymFC simulator is a drone simulator that has been highly tuned to match the dynamics of a particular real drone. This simulator (in connection with relaxations on the task) enabled direct sim-to-real transfer of a Proximal Policy Optimization (PPO)-based RL agent. Because of this, a successful sim-to-sim transfer of a PUDM-RL agent to the GymFC simulator can be seen as a necessary condition for a later sim-to-real transfer to be successful.

Adaptive Control In Section 4.1, we describe why DR is a special case of a Partially Observable Markov Decision Process (POMDP). In this light, adaptive control can be seen as a relaxation of the DR special case, which allows the partially observable variables to change during an episode. On a high level, an important feature of adaptive control over a general POMDP is the notion of temporal correlation of the unobserved part of the state. From that viewpoint, we can view DR as a special case of adaptive control, where the temporal correlation in the unobserved parameters is perfect because they are constant within an episode. An example for adaptive control, relating to the previous paragraph concerning drone stabilization, is the case where a drone picks up a payload and hence its dynamics change (center of mass, moment of inertia, aerodynamics). In this case, an adaptive controller is supposed to detect the change in dynamics and adapt its control policy accordingly, to recover the desired behavior in the new scenario.

Meta-Reinforcement Learning: Qualitatively Differing Tasks A task in the form of a Markov Decision Process (MDP) is mainly defined by its transition and reward probabilities. In this work, we focused on variations in the transition dynamics because we rephrased the DR approach, which is only concerned with variation in the dynamics, as a meta-RL problem. However, the general meta-RL problem may also include variations in the reward function. In Section 4.1, we briefly mentioned how variations in the reward function could be integrated into our PUDM-RL approach. There are multiple ways to extend the dynamics embedding to also contain information about the reward function of a particular environment. In all cases, we include the reward in each step of the experience for a particular environment and hence make it available for the dynamics encoder. To enforce the dynamics embeddings (or more fitting in this case: *task embeddings*) to contain information about the reward function, we also need to include a reward prediction model which receives a state and action as inputs and predicts the reward. If we use a multi-headed approach with a shared dynamics encoder for the transition and reward model, we face the problem of trading off the losses of both models. If coming up with a reasonable trade-off turns out to be challenging, training two separate dynamics encoders altogether might be a variation that is worthwhile to be tried.

Going forward, we will focus on future work for the sim-to-real transfer of drone stabilization policies and perspectively foray into adaptive control for dynamic drone tasks.

Bibliography

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[3] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–503, 2016.

[4] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *Proceedings of the 32nd International Conference on Machine Learning* (F. Bach and D. Blei, eds.), vol. 37 of *Proceedings of Machine Learning Research*, (Lille, France), pp. 1889–1897, PMLR, 07–09 Jul 2015.

[5] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017.

[6] O. M. Andrychowicz, B. Baker, M. Chociej, R. Józefowicz, B. McGrew, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray, J. Schneider, S. Sidor, J. Tobin, P. Welinder, L. Weng, and W. Zaremba, "Learning dexterous in-hand manipulation," *The International Journal of Robotics Research*, vol. 39, no. 1, pp. 3–20, 2020.

[7] D. Cliff, P. Husbands, and I. Harvey, "Explorations in evolutionary robotics," *Adaptive Behavior*, vol. 2, no. 1, pp. 73–110, 1993.

[8] N. Jakobi, *Minimal simulations for evolutionary robotics*. PhD thesis, University of Sussex, 1998.

[9] F. Zhang, J. Leitner, M. Milford, B. Upcroft, and P. I. Corke, "Towards vision-based deep reinforcement learning for robotic motion control," *Australasian Conference on Robotics and Automation (ACRA) 2015*, vol. abs/1511.03791, 2015.

[10] S. James and E. Johns, "3d simulation for robot arm control with deep q-learning," in *Workshop on Deep Learning for Action and Interaction*, 2016.

[11] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," 2019.

[12] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *Proceedings of The 33rd International Conference on Machine Learning* (M. F. Balcan and K. Q. Weinberger, eds.), vol. 48 of *Proceedings of Machine Learning Research*, (New York, New York, USA), pp. 1928–1937, PMLR, 20–22 Jun 2016.

[13] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," 2018.

[14] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *Proceedings of the 35th International Conference on Machine Learning* (J. Dy and A. Krause, eds.), vol. 80 of *Proceedings of Machine Learning Research*, (Stockholmsmässan, Stockholm Sweden), pp. 1861–1870, PMLR, 10–15 Jul 2018.

[15] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain randomization for transferring deep neural networks from simulation to the real world," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 23–30, 2017.

[16] S. James, A. J. Davison, and E. Johns, "Transferring end-to-end visuomotor control from simulation to real world for a multi-stage task," in *Proceedings of the 1st Annual Conference on Robot Learning* (S. Levine, V. Vanhoucke, and K. Goldberg, eds.), vol. 78 of *Proceedings of Machine Learning Research*, pp. 334–343, PMLR, 13–15 Nov 2017.

[17] F. Zhang, J. Leitner, Z. Ge, M. Milford, and P. Corke, "Adversarial discriminative sim-to-real transfer of visuo-motor policies," *The International Journal of Robotics Research*, vol. 38, no. 10-11, pp. 1229–1245, 2019.

[18] E. Kaufmann, A. Loquercio, R. Ranftl, A. Dosovitskiy, V. Koltun, and D. Scaramuzza, "Deep drone racing: Learning agile flight in dynamic environments," in *Proceedings of The 2nd Conference on Robot Learning* (A. Billard, A. Dragan, J. Peters, and J. Morimoto, eds.), vol. 87 of *Proceedings of Machine Learning Research*, pp. 133–145, PMLR, 29–31 Oct 2018.

[19] A. A. Rusu, M. Večerík, T. Rothörl, N. Heess, R. Pascanu, and R. Hadsell, "Sim-to-real robot learning from pixels with progressive nets," in *Proceedings of the 1st Annual Conference on Robot Learning* (S. Levine, V. Vanhoucke, and K. Goldberg, eds.), vol. 78 of *Proceedings of Machine Learning Research*, pp. 262–270, PMLR, 13–15 Nov 2017.

[20] F. Sadeghi and S. Levine, "Cad2rl: Real single-image flight without a single real image," in *Proceedings of Robotics: Science and Systems*, (Cambridge, Massachusetts), July 2017.

[21] L. Pinto, M. Andrychowicz, P. Welinder, W. Zaremba, and P. Abbeel, "Asymmetric actor critic for image-based robot learning," in *Proceedings of Robotics: Science and Systems*, (Pittsburgh, Pennsylvania), June 2018.

[22] K. Bousmalis, A. Irpan, P. Wohlhart, Y. Bai, M. Kelcey, M. Kalakrishnan, L. Downs, J. Ibarz, P. Pastor, K. Konolige, S. Levine, and V. Vanhoucke, "Using simulation and domain adaptation to improve efficiency of deep robotic grasping," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 4243–4250, 2018.

[23] T. Inoue, S. Choudhury, G. De Magistris, and S. Dasgupta, "Transfer learning from synthetic to real images using variational autoencoders for precise position detection," in *2018 25th IEEE International Conference on Image Processing (ICIP)*, pp. 2725–2729, 2018.

[24] R. Antonova, S. Cruciani, C. Smith, and D. Kragic, "Reinforcement learning for pivoting task," *CoRR*, vol. abs/1703.00472, 2017.

[25] J. Tan, T. Zhang, E. Coumans, A. Iscen, Y. Bai, D. Hafner, S. Bohez, and V. Vanhoucke, "Sim-to-real: Learning agile locomotion for quadruped robots," in *Proceedings of Robotics: Science and Systems*, (Pittsburgh, Pennsylvania), June 2018.

[26] Y. Chebotar, A. Handa, V. Makoviychuk, M. Macklin, J. Issac, N. Ratliff, and D. Fox, "Closing the sim-to-real loop: Adapting simulation randomization with real world experience," in *2019 International Conference on Robotics and Automation (ICRA)*, pp. 8973–8979, 2019.

[27] J. Hwangbo, J. Lee, A. Dosovitskiy, D. Bellicoso, V. Tsounis, V. Koltun, and M. Hutter, "Learning agile and dynamic motor skills for legged robots," *Science Robotics*, vol. 4, no. 26, 2019.

[28] F. Muratore, M. Gienger, and J. Peters, "Assessing transferability from simulation to reality for reinforcement learning," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 43, no. 4, pp. 1172–1183, 2021.

[29] S. Levine, P. Pastor, A. Krizhevsky, J. Ibarz, and D. Quillen, "Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection," *The International Journal of Robotics Research*, vol. 37, no. 4-5, pp. 421–436, 2018.

[30] E. Brunskill and L. Li, "Sample complexity of multi-task reinforcement learning," in *Proceedings of the Twenty-Ninth Conference on Uncertainty in Artificial Intelligence*, UAI'13, (Arlington, Virginia, USA), p. 122–131, AUAI Press, 2013.

[31] T. Yu, D. Quillen, Z. He, R. Julian, K. Hausman, C. Finn, and S. Levine, "Meta-world: A benchmark and evaluation for multi-task and meta reinforcement learning," in *Proceedings of the Conference on Robot Learning* (L. P. Kaelbling, D. Kragic, and K. Sugiura, eds.), vol. 100 of *Proceedings of Machine Learning Research*, pp. 1094–1100, PMLR, 30 Oct–01 Nov 2020.

[32] J. Schmidhuber, "On learning how to learn learning strategies," tech. rep., 1995.

[33] J. X. Wang, Z. Kurth-Nelson, D. Tirumala, H. Soyer, J. Z. Leibo, R. Munos, C. Blundell, D. Kumaran, and M. Botvinick, "Learning to reinforcement learn," 2017.

[34] Y. Duan, J. Schulman, X. Chen, P. L. Bartlett, I. Sutskever, and P. Abbeel, "Rl^2: Fast reinforcement learning via slow reinforcement learning," 2016.

[35] D. Wierstra, A. Förster, J. Peters, and J. Schmidhuber, "Recurrent policy gradients," *Logic Journal of the IGPL*, vol. 18, pp. 620–634, 09 2009.

[36] N. Heess, J. J. Hunt, T. P. Lillicrap, and D. Silver, "Memory-based control with recurrent neural networks," in *Deep Reinforcement Learning Workshop, NIPS 2015*, 2015.

[37] S. Hochreiter, A. Younger, and P. Conwell, "Learning to learn using gradient descent," pp. 87–94, 09 2001.

[38] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[39] C. Finn, P. Abbeel, and S. Levine, "Model-agnostic meta-learning for fast adaptation of deep networks," in *Proceedings of the 34th International Conference on Machine Learning* (D. Precup and Y. W. Teh, eds.), vol. 70 of *Proceedings of Machine Learning Research*, (International Convention Centre, Sydney, Australia), pp. 1126–1135, PMLR, 06–11 Aug 2017.

[40] OpenAI, I. Akkaya, M. Andrychowicz, M. Chociej, M. Litwin, B. McGrew, A. Petron, A. Paino, M. Plappert, G. Powell, R. Ribas, J. Schneider, N. Tezak, J. Tworek, P. Welinder, L. Weng, Q. Yuan, W. Zaremba, and L. Zhang, "Solving rubik's cube with a robot hand," *arXiv preprint*, 2019.

[41] K. Rakelly, A. Zhou, C. Finn, S. Levine, and D. Quillen, "Efficient off-policy meta-reinforcement learning via probabilistic context variables," in *Proceedings of the 36th International Conference on Machine Learning* (K. Chaudhuri and R. Salakhutdinov, eds.), vol. 97 of *Proceedings of Machine Learning Research*, pp. 5331–5340, PMLR, 09–15 Jun 2019.

[42] J. Q. R. M. W. D. Steven Kapturowski, Georg Ostrovski, "Recurrent experience replay in distributed reinforcement learning," in *International Conference on Learning Representations*, 2019.

[43] X. B. Peng, M. Andrychowicz, W. Zaremba, and P. Abbeel, "Sim-to-real transfer of robotic control with dynamics randomization," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 3803–3810, 2018.

[44] W. Zhou, L. Pinto, and A. Gupta, "Environment probing interaction policies," in *International Conference on Learning Representations*, vol. abs/1907.11740, 2019.

[45] E. Valassakis, Z. Ding, and E. Johns, "Crossing the gap: A deep dive into zero-shot sim-to-real transfer for dynamics," in *International Conference on Intelligent Robots and Systems (IROS)*, 2020.

[46] P. S. Thomas, "A notation for markov decision processes," *CoRR*, vol. abs/1512.09075, 2015.

[47] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. The MIT Press, second ed., 2018.

[48] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[49] M. Hutter, *Universal Algorithmic Intelligence: A Mathematical TopDown Approach*, pp. 227–290. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007.

[50] J. Achiam, "Spinning Up in Deep Reinforcement Learning," 2018.

[51] M. P. Deisenroth, G. Neumann, and J. Peters, *A survey on policy search for robotics*. now publishers, 2013.

[52] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, *et al.*, "Mastering atari, go, chess and shogi by planning with a learned model," *Nature*, vol. 588, no. 7839, pp. 604–609, 2020.

[53] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[54] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[55] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1238–1274, 2013.

[56] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, no. 3-4, pp. 229–256, 1992.

[57] J. Peters and S. Schaal, "Reinforcement learning of motor skills with policy gradients," *Neural Networks*, vol. 21, no. 4, pp. 682–697, 2008. Robotics and Neuroscience.

[58] H. Flanders, "Differentiation under the integral sign," *The American Mathematical Monthly*, vol. 80, no. 6, pp. 615–627, 1973.

[59] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in neural information processing systems*, pp. 1057–1063, 2000.

[60] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314, 1989.

[61] R. E. Bellman, *The Theory of Dynamic Programming*. Santa Monica, CA: RAND Corporation, 1954.

[62] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and S. Levine, "Soft actor-critic algorithms and applications," 2019.

[63] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine learning*, vol. 3, no. 1, pp. 9–44, 1988.

[64] D. Balduzzi and M. Ghifary, "Compatible value gradients for reinforcement learning of continuous deep policies," 2015.

[65] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *Proceedings of the 31st International Conference on Machine Learning* (E. P. Xing and T. Jebara, eds.), vol. 32 of *Proceedings of Machine Learning Research*, (Bejing, China), pp. 387–395, PMLR, 22–24 Jun 2014.

[66] T. Haarnoja, S. Ha, A. Zhou, J. Tan, G. Tucker, and S. Levine, "Learning to walk via deep reinforcement learning," in *Proceedings of Robotics: Science and Systems*, (FreiburgimBreisgau, Germany), June 2019.

[67] D. P. Kingma and M. Welling, "Auto-Encoding Variational Bayes," in *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014.

[68] C. Finn, A. Rajeswaran, S. Kakade, and S. Levine, "Online meta-learning," in *Proceedings of the 36th International Conference on Machine Learning* (K. Chaudhuri and R. Salakhutdinov, eds.), vol. 97 of *Proceedings of Machine Learning Research*, pp. 1920–1930, PMLR, 09–15 Jun 2019.

[69] B. M. Lake, R. Salakhutdinov, and J. B. Tenenbaum, "Human-level concept learning through probabilistic program induction," *Science*, vol. 350, no. 6266, pp. 1332–1338, 2015.

[70] B. M. Lake, R. Salakhutdinov, and J. B. Tenenbaum, "The omniglot challenge: a 3-year progress report," *Current Opinion in Behavioral Sciences*, vol. 29, pp. 97–104, 2019. Artificial Intelligence.

[71] G. Koch, R. Zemel, and R. Salakhutdinov, "Siamese neural networks for one-shot image recognition," in *ICML deep learning workshop*, vol. 2, Lille, 2015.

[72] O. Vinyals, C. Blundell, T. Lillicrap, k. kavukcuoglu, and D. Wierstra, "Matching networks for one shot learning," in *Advances in Neural Information Processing Systems* (D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, eds.), vol. 29, Curran Associates, Inc., 2016.

[73] J. Snell, K. Swersky, and R. Zemel, "Prototypical networks for few-shot learning," in *Advances in Neural Information Processing Systems* (I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, eds.), vol. 30, Curran Associates, Inc., 2017.

[74] J. Schmidhuber, J. Zhao, and M. Wiering, "Simple principles of metalearning," *Technical report IDSIA*, vol. 69, pp. 1–23, 1996.

[75] H. F. Harlow, "The formation of learning sets.," *Psychological review*, vol. 56, no. 1, p. 51, 1949.

[76] J. Eschmann, *Reward Function Design in Reinforcement Learning*, pp. 25–33. Cham: Springer International Publishing, 2021.

[77] J. Deng, W. Dong, R. Socher, L. Li, Kai Li, and Li Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 248–255, 2009.

[78] D. J. Butler, J. Wulff, G. B. Stanley, and M. J. Black, "A naturalistic open source movie for optical flow evaluation," in *European Conf. on Computer Vision (ECCV)* (A. Fitzgibbon et al. (Eds.), ed.), Part IV, LNCS 7577, pp. 611–625, Springer-Verlag, Oct. 2012.

[79] A. Dosovitskiy, P. Fischer, E. Ilg, P. Häusser, C. Hazırbaş, V. Golkov, P. v.d. Smagt, D. Cremers, and T. Brox, "Flownet: Learning optical flow with convolutional networks," in *IEEE International Conference on Computer Vision (ICCV)*, 2015.

[80] R. Geirhos, P. Rubisch, C. Michaelis, M. Bethge, F. A. Wichmann, and W. Brendel, "Imagenet-trained CNNs are biased towards texture; increasing shape bias improves accuracy and robustness.," in *International Conference on Learning Representations*, 2019.

[81] W. Yu, J. Tan, C. K. Liu, and G. Turk, "Preparing for the unknown: Learning a universal policy with online system identification," in *Proceedings of Robotics: Science and Systems*, (Cambridge, Massachusetts), July 2017.

[82] J. Tian and other contributors, "Reinforcementlearning.jl: A reinforcement learning package for the julia language," 2020.

[83] A. Raffin, A. Hill, M. Ernestus, A. Gleave, A. Kanervisto, and N. Dormann, "Stable baselines3." `https://github.com/DLR-RM/stable-baselines3`, 2019.

[84] Z. Ding, "Popular-rl-algorithms." `https://github.com/quantumiracle/Popular-RL-Algorithms`, 2019.

[85] M. Hausknecht and P. Stone, "Deep recurrent q-learning for partially observable mdps," in *AAAI Fall Symposium on Sequential Decision Making for Intelligent Agents (AAAI-SDMIA15)*, November 2015.

[86] W. Koch, R. Mancuso, R. West, and A. Bestavros, "Reinforcement learning for uav attitude control," *ACM Transactions on Cyber-Physical Systems*, vol. 3, no. 2, p. 22, 2019.

A. Latent Space Encoding of Uncertainty

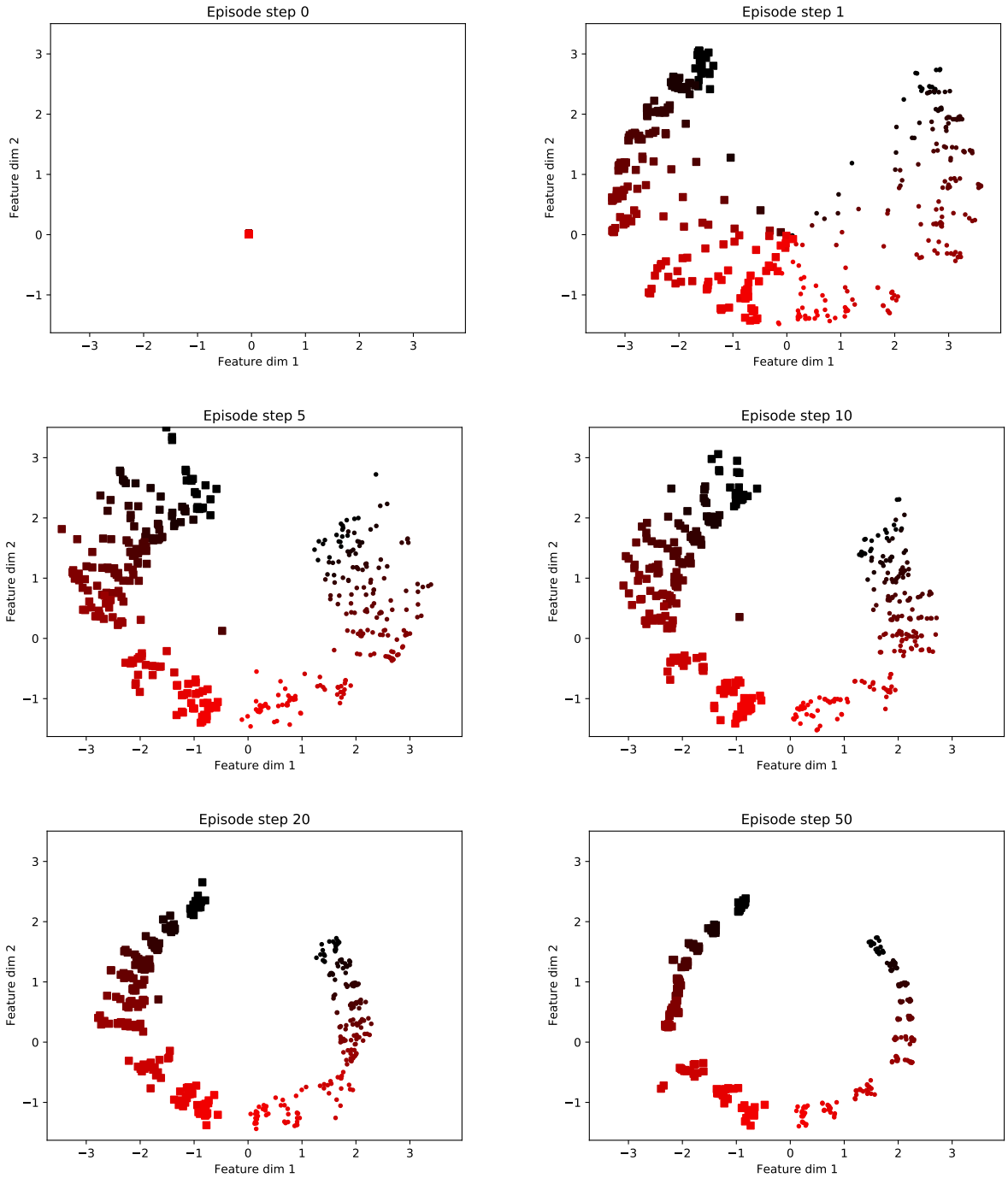


Figure A.1.: Discovery process of the environment dynamics during an episode. At step 0, the dynamics encoder has no experience in the environment and hence gives a single prior dynamics embedding vector, characterizing the uncertainty about the current environment. After only one step, the dynamics encoder already has a good guess about the environments but still expresses uncertainty. With more interaction data, the certainty in the environment's dynamics rises until they are easily distinguishable after around 50 steps.

B. Hyperparameters

B.1. Dynamics Encoder

Part	Parameter	Value
Dynamics Encoder	Max Sequence Length (Training)	80
	Masking	true
	RNN-Type	LSTM
	Hidden State Dimensionality	64
	Embedding Vector Dimensionality	2
Transition Model	Type	DNN
	Activation Function	ReLU
	Layer Dimensionalities	[256, 256, 256]
Combined Model	Learning Rate	0.001
	Epochs	10
	Batch Size	32

Table B.1.: Hyperparameters: Sequential dynamics encoder (structure in Figure 4.5).

Part	Parameter	Value
Sequence Encoder	RNN-Type	Multi-Layer LSTM
	Max Sequence Length (Training)	16
	Masking	true
	Hidden State Dimensionalities	[64, 64]
Backbone Network	RNN-Type	LSTM
	Number of Sequences	8
	Masking	false
	Hidden State Dimensionality	32
	Embedding Vector Dimensionality	2/3 (Pendulum/Acrobot)
Transition Model	Type	DNN
	Activation Function	ReLU
	Layer Dimensionalities	[256, 256, 256]
Combined Model	Learning Rate	0.001
	Epochs	10
	Batch Size	32

Table B.2.: Hyperparameters: Multi-sequential dynamics encoder (structure in Figure 4.6).

B.2. Reinforcement Learning

Part	Parameter	Value
General	Warmstart Interactions	5000
	Replay Buffer Capacity	1000000
	Initial Entropy Bonus Trade-off α	0.2
	Discount Factor γ	0.99
	Polyak Averaging Constant ρ	0.995
	Batch Size	256
	Entropy Target	-
	Learning Rate (Actor, Critic and α Optimizer)	0.0003
Actor & Critic	Type	DNN
	Activation Function	ReLU
	Layer Dimensionalities	[64, 64]

Table B.3.: Hyperparameters for the SAC-based RL agent in the Pendulum environment. Modifications to the hyperparameters for the Acrobot are given in Table B.4.

Part	Parameter	Value
General	Initial Entropy Bonus Trade-Off α	2
	Entropy Target	-0.5
Actor & Critic	Layer Dimensionalities	[512, 512]

Table B.4.: Hyperparameter modifications for the Acrobot environment (starting from the hyperparameters in Table B.3).