TUM School of Computation, Information and Technology
Technical University of Munich

Master's Thesis in Robotics, Cognition, Intelligence

Meta Reinforcement Learning through Task Inference Reutilization

Supervisor Prof. Dr.-Ing. habil. Alois C. Knoll

Advisor Dr. Zhenshan Bing

Author Juan de los Rios Ruiz

Date September 30, 2024 in Munich

Disclaimer

I confirm that this Master's Thesis is my own work and I have documented all sources and material used.

Munich, September 30, 2024

(Juan de los Rios Ruiz)

Abstract

Reinforcement Learning (RL) is a promising field that has seen a rapid development over the past years. This comes as the result of more sophisticated algorithms and improvements in simulation software as well as hardware. RL approaches the learning process in a manner similar to human learning. However, humans clearly outperform machines when learning new tasks that share similarities to already learned tasks. Machines lack the capabilities that human posses of leveraging existing knowledge about different processes to learn new tasks. In robotics, the ideal agent can quickly adapt to new scenarios and execute tasks for which it hasn't been explicitly trained for. Unfortunately, this is far from reality.

Meta Reinforcement Learning (MRL) combines insights from of Meta Learning (MetaL) and RL to address this challenge. In inference-based MRL this problem is tackled by separating task recognition and policy training, a combination which enables the agent to perform tasks on which it has not been trained before. However, this approach introduces a dependency between the two components making the learning more difficult, especially in agents with a higher movement complexity.

We will introduce a method that will solve this problem by learning actions in different hierarchies. Additionally, we will show that the task inference module can be robot agnostic by proving that it can be reused in different types of robots without the necessity of retraining.

Contents

Acronyms

AC Actor-Critic. 10

AE Autoencoder. 16, 17

CL Curriculum Learning. 37, 63

DA Domain Adaptation. 21, 23, 24

DDPMM Deep Dirichlet Process Mixture Model. 18

DL Deep Learning. 10, 14

DP Dirichlet Process. 18, 23, 32

DPMM Dirichlet Process Mixture Model. 17–19, 33, 49, 54, 64

DRL Deep Reinforcement Learning. 1

ELBO Evidence Lower BOund. 30

GMM Gaussian Mixture Model. 23

GRUs Gated Recurrent Units. 15, 30, 34, 43

HRL Hierachical Reinforcement Learning. 3, 23

MAML Model-Agnostic Meta-learning. 21, 22, 24

MDP Markov Decission Process. 3–5, 12, 22, 23, 29, 30, 44

MetaL Meta Learning. iv, 21, 24

ML Machine Learning. 21, 23, 41

MLP Multilayer Perceptron. 30, 38

MRL Meta Reinforcement Learning. iv, 1–3, 5, 13, 14, 16, 17, 21–25, 27, 32, 38, 43, 46, 49, 63, 64

MSE Mean Squared Error. 31

MTL Multi-task Learning. 1

multi-task RL Multi-task Reinforcement Learning. 1, 12, 13

NN Neural Network. 9, 10, 12, 14–16, 18, 23, 30, 37, 39, 46, 49, 50

POMDP Partially Observable Markov Decision Process. 5, 13, 16, 25, 30

RL Reinforcement Learning. iv, 1, 3, 4, 7, 8, 10–12, 14, 20–23, 28, 35, 41

RNNs Recurrent Neural Networks. 14, 15

SAC Soft Actor-Critic. 10–12, 23, 28, 35, 36, 55, 63

SARSA State–action–reward–state–action. 9

TD Temporal Difference. 8–11

VAE Variational Autoencoder. 16, 17, 19, 25, 30, 33, 54

Chapter 1

Introduction

Robots are used in a wide range of fields. The applications of robots were limited for many years to repetitive tasks in static environments. But with the introduction of Deep Reinforcement Learning (DRL), a whole new range of applications emerged. Reinforcement Learning (RL) allows robots to learn from experience similar to the way humans learn. In combination with improvements in the area of computing allowing for better simulations, robots can be trained to perform tasks in dynamic environments. Some examples for new fields range from automating factories due to the advances in dexterous manipulation [Ope+19] and multi-robot collaboration [OD23] to household cleaning enabled by advances in robot navigation [Che+23].

Unlike humans, robots still exhibit significant challenges at learning to perform new, unseen tasks. Humans leverage knowledge obtained from other tasks to effectively combine skills and learn to perform new tasks. In traditional RL algorithms, in order to learn how to perform a new task, the training generally has to be redone from scratch. This requires a lot of time and resources.

To tackle this inefficiency two subfields of RL emerged, Multi-task Reinforcement Learning (multi-task RL) and Meta Reinforcement Learning (MRL). The former trains a policy to perform multiple tasks from a predefined set of tasks. MRL trains a model to learn how to learn, thereby enabling the agent to quickly adapt to new conditions and objectives. Compared to Multi-task Learning (MTL) it has shown to better adapt to new, unseen tasks.

Existing research can be subdivided into three main subfields of MRL. An initial approach was introduced in [FAL17]. This method can be categorized under gradient-based method as it focuses on creating a model which can quickly adapt to new tasks with few gradient steps. A different idea was followed by algorithms which can be classified under recurrent-based methods. These make use of recurrent networks that maintain a hidden state which is continuously adapted. Similarly, inference-based MRL focus on the idea that a task can only be identified from various transitions of states. The focus lies in learning a network that can extract important information about the task. This information then serves as input to the policy module. [Rak+19] and [Bin+15] showed that this approach offers a faster adaptation to new tasks making the inference part of the model an important factor to consider in MRL.

This thesis will contribute to the field of inference-based MRL in the following ways:

1. It continues on the approach of [Dur23] to test the hypothesis that a robot-agnostic inference module can be learned. By doing so, the inference module can be reused on different agents with distinct dynamical models.

2. It explores the idea of hierarchical policy structures to allow agents which are harder to train due to the high degrees of freedom and complex interactions with the environment to facilitate exploration.
3. Proposing a novel inference-based approach to transfer learned knowledge across agents with different levels of complexity.
4. Opens a new direction of research by proposing a method that can be reused for different applications.

Following the introduction of the method, a set of experiments are presented which show the applicability of the method. Since the goal of this thesis was to create a method that is applicable to a wide range of robots, the experiments will focus on validating the proposed approach on robots with different configurations. The experiments will be performed in the mujoco environment. This allows to simulate different robots and validate whether these perform the required tasks. Additionally, we will inspect two different training methods for the proposed algorithm and compare both, outlining the respective benefits and disadvantages.

The rest of this thesis is structured as follows: First, some background on important topics for the algorithms used are presented; followed by a mention to related work done in the field. In chapter 5, a detailed description of the proposed inference-based MRL is given. This is then tested and the results are displayed in chapter 6. Lastly, open topics and future work are discussed in chapter 7.

Chapter 2

Background

MRL is a field which combines various aspects of machine learning. In order to grasp the concept of MRL it is important to understand a few fundamental concepts which will be covered in the following sections. We begin with an overview of RL which builds upon the fouondational concept of Markov Decission Process (MDP)s. In Section 2.3 the extension for multitask and meta reinforcement learning will be explained, highlighting the differences between both terminologies. This thesis aims to expand on the concept of inference-based reinforcement learning. One essential block of this approach is the inference module. Sections 2.4 and 2.5 will explain the necessary concepts to understand how the inference module is constructed and how it can retrieve information from sequential data. Additionally, an extension to the inference module will be discussed in Section 2.6.2 which enables the representation of non-parametric task variations. Lastly, a brief introduction into Hierachical Reinforcement Learning (HRL) will be given as this is a fundamental concept for the method proposed in this thesis.

2.1 Markov Decision Processes

It is essential to model the processes in a structured manner. Only then can algorithms be created. MDPs offer a mathematical framework for modeling sequential decision making in a stochastic scenario where an agent can influence the outcome. These form the basics for all RL algorithms.

A MDP consists of following tuple: (S, A, P, R, γ), where each element is a crucial aspect of the decision making process. These are defined in the following way:

- **State Space** S**:** Defines the set of all possible states the agent can end up in. For the example of a legged robot this includes all combinations of positions and velocities of each joint. Depending on the definition of the MDP, also the overall position and velocity of the robot must be taken into account.
- **Action Space** A**:** Defines the set of all possible actions the agent can undertake. For the same example as above, actions can vary depending on the level of description. In a more detailed model of the robot, the actions can be defined as the torque applied to each joint. Conversely, in a higher level of abstraction, the actions can be modeled as the degrees of rotation of each joint.
- **Transition Probability** P**:** This probability distribution describes the dynamics of the system, i.e. it defines the likelihood of ending up in state s' when taking action a in state s.

- **Reward Function** R**:** A reward function is part of every MDP which inherently defines the goal of it. It describes the reward received after a transition from one state to another. A higher reward generally indicates the agent has performed an action that moves it closer to the goal.

- **Discount Factor** γ**:** balances the importance of future versus current rewards. A higher value signifies more importance for future rewards.

MDPs are used to describe sequential processes such that at every time step, the current state s_t, action a_t, reward r_t and transition probability $p(s_{t+1} \mid s_t, a_t)$ describe the dynamics of the system. Moreover, all MDPs must satisfy the Markov Property, which states that the transition probability solely depends on the current state and the action performed in that state.

In a typical MDP problem, the primary objective is to guide the agent such that it obtains the maximum accumulated reward over time. Given that the rewards depend on the states visited, the goal is to visit the states that deliver the highest reward. Since the transition probability depends on the current state and the action taken, the accumulated reward is a result of the actions taken. This is where the importance of the policy π becomes evident as it is defined as the strategy for choosing the action given the state: $\pi(a_t \mid s_t)$. Mathematically, the objective can be quantified by following equation:

$$G_t = \sum_{t=0}^{\infty} \gamma^t r_{t+k} \tag{2.1}$$

The discount factor $\gamma \in [0, 1)$ ensures that $G_t < \infty$, which is important for the practical implementation of RL algorithms. Using this equation, one can calculate the expected accumulated return for the current state s, given a policy π. In the literature, this quantity is denoted as the value function $V^{\pi}(s)$:

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s\right] \tag{2.2}$$

The value function helps evaluate the consequences of following a certain policy. Understanding how the actions affect the outcome is essential for selecting the right actions and thus constructing a better policy. To achieve the objective of maximum returns, the policy that maximizes the value function has to be found. Such a policy receives the name of optimal policy and is denoted as π^*. However, finding the optimal policy can be challenging depending on the settings of the MDP. Different algorithms can be employed depending on how much information is known beforehand.

The structure of the reward function can vary significantly depending on the objective of the MDP.

- **Dense reward functions:** Provides feedback after every step. One example is a reward function which goal is to guide an agent to reach a specific goal. After each transition, the reward can be calculated as the negative distance between the agent and the goal position.

- **Sparse reward functions:** Provides feedback after longer trajectories. One example is the completion of a maze where the agent only receives a reward once the maze has been successfully exited. Generally, sparse reward functions make the learning process more challenging.

2.1.1 Partially Observable Markov Decision Process

The term Partially Observable Markov Decision Process (POMDP) will be used in Section 2.3.2 with the introduction of MRL. This framework extends traditional MDPs for scenarios where the full state cannot be observed. The tuple describing the process becomes $(S, A, P, R, O, Z, \gamma)$. The newly added terms are introduced to aid in handling the partial observability:

- **Observable State O:** Represents the part of the state space that can be observed.
- **Observation Function Z:** Probability of making observation o given state s and action a.

In some settings, the agent is not able to fully observe the scene. Therefore, it must act based on its beliefs about the current state. This implies the need for the introduction of a belief state which makes an informed estimate to approximate the part of the state space that cannot be observed. This belief state gets updated at every step based on past and current transitions. Mathematically, the update equation can be formulated as shown in equation 2.3.

$$b_{t+1}(s') = \frac{O(o_{t+1}|s', a_t) \sum_s P(s'|s, a_t) b_t(s)}{P(o_{t+1}|b_t, a_t)} \tag{2.3}$$

Where:

- $b_{t+1}(s')$ is the updated belief state for state s' at time $t+1$.
- $Z(o_{t+1}|s', a_t)$ is the observation function, representing the probability of observing o_{t+1} given state s' and action a_t.
- $P(s'|s, a_t)$ is the transition probability from state s to state s' given action a_t.
- $b_t(s)$ is the belief state for state s at time t.
- $P(o_{t+1}|b_t, a_t)$ is the normalization factor.

In summary, a POMDP allows an agent to perform actions that, if done correctly, guides the agent closer to the goal. This is achieved by updating a belief state since the entire state is not observable.

2.1.2 Dynamic Programming

With the goal in mind of finding the best policy, one can differentiate between different MDPs. In the most simple case, one of deterministic nature with finite horizon, simple dynamic programming methods can be used to find the solution of the value function from which the optimal policy can be derived.
The procedure to find the solution can be summarized as starting from the end state and iterating through preceding states until the initial state is reached. Along this trajectories, the rewards are summed up thus obtaining the accumulated reward.

However, in practical settings, one often encounters more complex scenarios that may be of infinite horizon and non-deterministic. These type of problems can be tackled with the Bellman equation. Equations 2.4 and 2.5 represent the Bellman equations for the value function. It is a recursive function to calculate the value function.

- **Bellman equation for policy π:**

$$\begin{aligned} V^{\pi}(s) &= \mathbb{E}[R(s') + \gamma V^{\pi}(s')] \\ V^{\pi}(s) &= \sum_{a} \pi(a|s) \sum_{s'} P(s'|s,a)\big[R(s,a,s') + \gamma V^{\pi}(s')\big] \end{aligned} \tag{2.4}$$

  This equation represents the value function for the policy π.

- **Bellman optimality equation:**

$$\begin{aligned} V^{*}(s) &= \max_{a} \mathbb{E}[R(s') + \gamma \max_{a'} V^{(}s')] \\ V^{\pi^{*}}(s) &= \max_{a} \sum_{s'} P(s'|s,a)\big[R(s,a,s') + \gamma V^{\pi^{*}}(s')\big] \end{aligned} \tag{2.5}$$

  This equation represents the optimal value function, i.e. the value function obtained if the optimal policy is followed at every time step.

These equations form the foundation for various dynamic programming algorithms by decomposing a complex problem of potentially infinite horizon into one of manageable size. This is made possible by the discount factor as future rewards will tend to zero when $t \to \infty$ making the sum converge to a finite number.

Two common algorithms that make use of this equation to find the optimal policy will be discussed:

Value Iteration Algorithm: This algorithm iterates over Equation 2.6 until it finds the optimal policy π^{*}.

$$V(s) \leftarrow \max_{a} \left\{ \sum_{s'} P(s,a,s') \big(R(s,a,s') + \gamma V\big(s'\big)\big) \right\} \tag{2.6}$$

In every iteration, the optimal action is chosen for every state s. Hence, the algorithm will converge to the value function of the optimal policy. The optimal policy is then given by the action taken at each state s which will maximize the term $\sum_{s'} P(s,a,s') \big(R(s,a,s') + \gamma V\big(s'\big)\big)$.

Iterating infinitely many times over Equation 2.6 ensures finding the true value function. However, in practice, a close approximation is sufficient since it is infeasible to iterate endlessly, and the policy will converge for a sufficiently close approximation of the true value function.

Policy iteration algorithm: This algorithm divides the process into updating the policy and updating the value function. The benefit of this is that it does not require to calculate the optimal policy at every time step. It works as follows:

1. Calculate an approximation of the value function for the current policy π, i.e. iterate Equation 2.7 until $V' - V < \theta$ for a given θ.

$$V'(s) \leftarrow \sum_{s'} P(s,a,s') \big(R(s,a,s') + \gamma V\big(s'\big)\big) \tag{2.7}$$

2. Compute the optimal policy.

$$\pi(s) \leftarrow \arg\max_a \sum_{s'} P(s,a,s')\big(R(s,a,s') + \gamma V\big(s'\big)\big) \tag{2.8}$$

These functions however still don't apply to many real world problems due to the assumption that the transition probability distribution is known. In many applications, the dynamics are too complex to make models about the transitions.

2.1.3 Q-function

The previously discussed algorithms don't have the capability to be applied in many real world scenarios. Reason is the curse of dimensionality. This issue is caused by the need to evaluate the value function for every state in each iteration. When the state space grows large, the number of computations grow exponentially. This problem intensifies when the problem is of continuous nature.

In [WD92] Watkins introduced the concept of the Q-function. The Q-function is different to the value function in that it does not represent the accumulated reward for a given state, but it approximates the future rewards for an action-state pair. This difference can be explained mathematically by following equation:

$$Q^{\pi}(s,a) = \mathbb{E}\left[\sum_{t=1}^{\infty} \gamma^t r_t | s_0 = s, a_0 = a\right] \tag{2.9}$$

This representation provides a computational advantage compared to the value function: finding the optimal action does not require to iterate over all actions. Instead, it can be deduced by selecting the highest Q-value. This reduces the computational complexity especially for large action spaces.

However, this improvement comes at a cost. Since the Q-function stores the values for all combinations of states and actions, the set of state-action pairs is larger than the set of states alone. Therefore, the improvement in computational cost comes tied to an increase in memory capacity.

The Bellman equations (Equations 2.4,2.5) for the Q-function can be applied analogously to the value function and thus also the algorithms mentioned above.

2.2 Reinforcement Learning

This section will build upon the discussed theory in the previous sections and explain how it can be used in real world scenarios, especially in robotics. In the robotic scene, RL is used to enable robots to act autonomously by learning from interactions with the environment. This happens in a cycle over many iterations of learning and action-taking, illustrated in Figure 2.1. This section will clarify the mathematical background that enables robots to learn from experiences similar to the way humans learn.

Until now, the assumption of a given dynamical model was needed, i.e. the transition probability of ending up in state s' from state s given action a had to be known. This assumption,

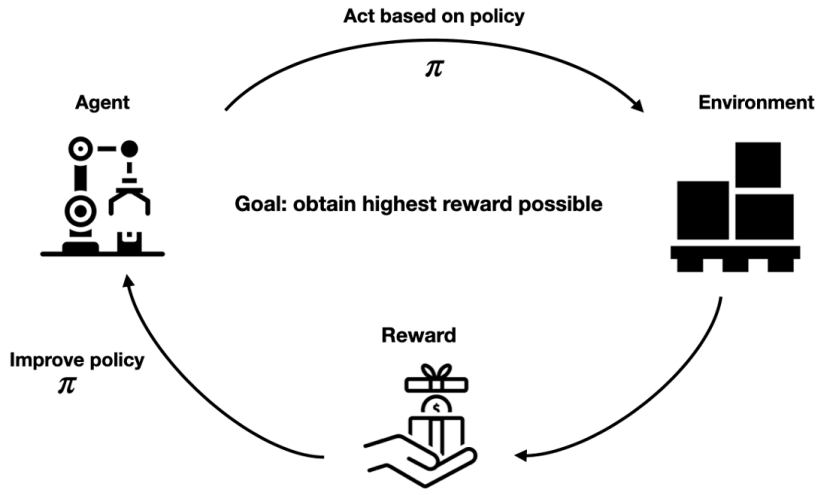


Figure 2.1: RL cycle: How robots learn from experience

however, does not hold in most robotic applications. The dynamic models are not precise enough to predict future states with enough precision. Temporal Difference (TD) learning offers a solution to this issue by enabling learning from the outcomes of actions in the environment, eliminating the need for a predefined model. This new method employs a technique called bootstrapping which updates the Value/Q-function based on the current estimate. Although TD learning algorithms like the ones described in Section 2.2.1 and 2.2.2 are model-free, they converge to the true value function given that the policy does not change and the Robbins-Monro conditions are met:

- $\sum_{t=1}^{\infty} \alpha_t = \infty$
- $\sum_{t=1}^{\infty} \alpha_t^2 < \infty$

From here on, the value function and the Q-function will be used interchangeably unless stated otherwise as they serve the same purpose.

Every update step in TD learning is based on the TD-error:

$$\gamma V\left(s'\right) + r - V(s) \tag{2.10}$$

It measures the difference between the current estimate of the next state plus the reward and the current estimate of the current state. For the case that the true value function is known, this error is zero as the current value is the value of the next state plus the reward.

The term bootstrapping was introduced to describe updating an estimate based on other estimated values. This incremental improvement is described by Equation 2.11:

$$V(s) \leftarrow V(s) + \alpha\left(\gamma V\left(s'\right) + r - V(s)\right) \tag{2.11}$$

where α is a scalar that represents the learning rate hyperparameter.

Note that the Q-function can be used instead of the value function.

2.2.1 On-policy

From TD learning, two types of algorithms emerge, off- and on-policy algorithms. Off-policy algorithms make use of the TD error to update the Q-function based on the current policy π.

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha\big(r + \gamma Q(s',a') - Q(s,a)\big) \tag{2.12}$$

This algorithm in particular, called State–action–reward–state–action (SARSA), is part of the on-policy family because the updates are based on the actions selected from the current policy. In SARSA, the action a' is chosen to be consistent with the current policy, ensuring that the learning process is directly related to the policy the agent is following.

2.2.2 Off-policy

On the other hand, the off-policy algorithms update the Q-function based on actions that may differ from the ones dictated by the current policy. A widely used off-policy algorithm is called Q-Learning. It updates the Q-function based on the maximum expected future reward, regardless of the action chosen under the current policy with the aim of finding the Q-function that corresponds to the optimal policy.

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha\Big(r + \gamma \max_{a'} Q\big(s',a'\big) - Q(s,a)\Big) \tag{2.13}$$

The aforementioned methods treat the Q-function as a tabular. Every state-action pair has a Q-value assigned which gets updated in every iteration. With increasing state and action spaces, the problem known as curse of dimensionality arises: The number of possible states grows exponentially with the dimensions of the state space. Even more problematic becomes continuous state spaces.

2.2.3 Deep Reinforcement Learning

The solution to this problem are function approximations. Due to their capability of approximating non-linear functions and learnable weights, Neural Network (NN)s have been succesfully implemented to approximate the Q-function and are commonly used. As a result a new family of algorithms appear: Deep Reinforcement Learning algorithms. These algorithms take advantage of the ability to handle high-dimensional input spaces to extract information from large action and state spaces. Due to the nature of these approximators, the guarantee of convergence to the true Q-function is lost. However, in practice, these type of algorithms have proven to deliver good results. One example is Deep Q-Learning. This algorithms follows the update step of the Q-Learning algorithm with the difference that the Q-function is not represented as a tabular, but approximated by a NN. Based on the TD error, the gradient for the Q-function approximation can be computed and hence updated.

Like in many other Deep Learning (DL) solutions, these algorithms require a vast amount of data. However, letting robots perform actions and learn from them is generally unfeasible due to two reasons:

- **Scalability:** In order to collect the data needed for the training, many robots would need to run simultaneously. This requires resources as robotic hardware tends to be expensive. Obtaining the necessary resources for training would limit the research to a few resourceful institutions.

- **Safety:** The robots need to explore many different states to learn efficient policies. Some states are, however, not safe. Entering these states could harm the robot itself or its surroundings, potentially leading to costly damages or dangerous situations.

The solution to this problem lies in the use of simulations. Digital twins offer a way to replicate the robot and its environment which are simulated with physics engines. This solution is not flawless as simulations don't represent the real world perfectly. Applying the networks to the real world, sometimes lead to degradation in performance due to the simulation-to-reality gap. The higher the precision of the simulation, the less the agent suffers this problem. Different publications have proposed ways to reduce this effect ([ZQW20]).

2.2.4 (Soft) Actor-Critic

A different way of improving the performance of robots trained with RL is to create more suitable algorithms. Deep Q-Learning meant a notable step in the direction of autonomous robots. Learning the Q-function, however, is not sufficient. The need to iterate over all actions to find the highest Q-value makes it problematic for continuous action spaces; and since many control problems require the action spaces to be continuous, other algorithms are better suited. This section will focus on the algorithm family called Actor-Critic (AC) used for finding the optimal policy later in the experiments (Chapter 6).

AC consists of two components, as the name suggests, the actor and the critic. The actor defines a policy $\pi_\theta(a|s)$ parameterized by the parameters θ. The critic serves as an estimator of the value function with parameters ϕ. Both components are typically modeled by NNs. The main difference among AC variations is how the critic is used in the actor update step. The basic AC employs the TD-error to update the policy. Another example is the Advantage Actor-Critic which utilizes the advantage function for the update: $A(s,a) = Q(s,a) - V(s)$. This measures the advantage of taking action a compared to the value function, which encompasses all actions.

These algorithms offer a valuable extension to Deep Q-Learning by combining policy-based and value-based algorithms. However, they lack the flexibility to balance exploration and exploitation. For agent with many variables and degrees of freedom, exploration is essential as there are many states that need to be visited. Too little exploration can lead to convergence in suboptimal local minima as a large amount of states have not been visited. Soft Actor-Critic (SAC) expands this idea by providing the possibility to adjust the exploration-exploitation ratio. It does so by finding a balance between maximizing future rewards and maximizing entropy in the actions taken. The weight of the entropy is controlled by the temperature parameter α. The greater the entropy, the greater the exploration.

The goal of finding the policy that maximizes the expected accumulated reward while maintaining a certain level of exploration can be described by following equation:

$$J(\pi) = \mathbb{E}_{s\sim p, a\sim\pi}\left[\sum_t \gamma^t \left(r(s_t, a_t) + \alpha\mathcal{H}(\pi(\cdot|s_t))\right)\right] \tag{2.14}$$

where $\mathcal{H}(\pi(\cdot|s_t))$ is defined as the entropy of policy π:

$$\mathcal{H}(\pi(\cdot|s)) = -\sum_a \pi(a|s)\log\pi(a|s) \tag{2.15}$$

Aside from combining the strengths of policy-based methods (actor) and value-based methods (critic), the SAC algorithm enables backpropagation through the gradient of the Q-function for the policy update step. Without it, a very accurate dynamics model would be necessary to backpropagate through the simulation. This is visualized in Figure 2.2.

Due to the separation of actor and critic, the update step in every iteration consists of two parts. The first one adjusts the Q-function through minimizing following loss function:

$$\begin{aligned} y &= r + \gamma\left(\min_{i=1,2} Q_{\theta_i'}(s', a') - \alpha\log\pi_\phi(a'|s')\right) \\ \mathcal{L}(\theta_i) &= \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[\left(Q_{\theta_i}(s,a) - y\right)^2\right] \end{aligned} \tag{2.16}$$

By minimizing this term, the model learns the dynamics of the agent interacting with the environment. Higher exploration offers a significant advantage since the agent encounters more states and thus derives a more general model of the dynamics.

This equation is similar to the TD error with two additions. One is the usage of multiple Q-functions and choosing the minimum for the update. This has shown to stabilize training. The second addition is the entropy adjustment term. This accounts for consistency with the actor's goal which not only aims to find the most optimal policy but also one with certain entropy:

$$J(\phi) = \mathbb{E}_{s\sim\mathcal{D}, a\sim\pi_\phi}\left[\min_{i=1,2} Q_{\theta_i}(s,a) - \alpha\log\pi_\phi(a|s)\right] \tag{2.17}$$

The policy is then updated with gradient ascent without the need to backpropagate through the model of the environment.

2.3 Meta Reinforcement Learning

Recent advances in RL have enabled autonomous control of complex robots. Nevertheless, RL still has some hurdles to overcome. The difficulty to generalize to different tasks and the sample inefficiency of retraining RL models, make it challenging to create robots that perform multiple tasks in dynamic environments. Two fields have emerged that tackle this problem that will be discussed in this section.

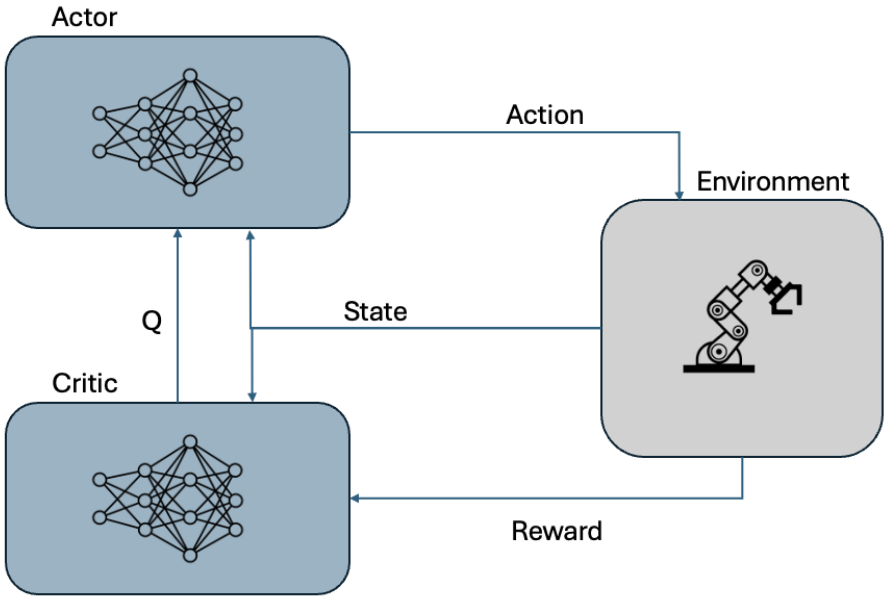


Figure 2.2: Process of SAC

2.3.1 Multi-task Learning

Multi-task RL expands the concept of RL to solve different types of tasks. Every task is part of a set $\mathcal{C} = \{\mathcal{T}_i\}_{i=1}^{N}$. In the general formulation, every task $\mathcal{T}_i$ has its own MDP model $(S_i, A_i, P_i, R_i, \gamma_i)$. However, we will only focus on the subset of scenarios where the MDP only differs in the reward function. This focus can be reasoned by the use case we are observing. Focus lies on scenarios that change the task but not the agent. Thus, the dynamics stay the consistent, and the only part that differs is the reward at each state.

The objective of the problem remains the same as in RL: train a policy that maximizes the accumulated reward:

$$\theta^* = \arg\max_{\theta} \mathbb{E}_{\mathcal{T}\sim P(\mathcal{T})}\left[J_\pi^{\mathcal{T}}(\theta)\right] \tag{2.18}$$

where $J_\pi^{\mathcal{T}}$ is task specific and depends on R_i. During training and testing, the task information is given as an input to the model. The goal is to find a model that best generalizes to all tasks. Due to their ability to handle inputs with a high degree of dimensions, NNs are well suited for this type of training.

Multi-task RL aims to take advantage of similarities between tasks to perform a more efficient training. By training on multiple tasks, the agent can learn basic behaviors that can be used in different tasks. This reduces the amount of training steps and may lead to better generalization. Hence, multi-task RL not only improves efficiency but also improves robustness of robotic systems.

2.3.2 Meta Reinforcement Learning

The aim of MRL is to train an agent that acquires the skill to adapt to new tasks, not just solve a single task. It can be described as learning how to learn. In essence, it shares the same objective as in multi-task RL; maximizing the accumulated reward for a combination of different tasks. The main difference in the scenarios that we will be exploring is that no task information is given to the model. Instead, the task has to be inferred from the rewards received, it needs to explore the space and learn how to solve this new task. This can be seen as a POMDP where the model has to maintain a belief state to infer the task it is meant to solve based on past transitions. Equation 2.18 can be extended by taking into account the belief state h_t. It can then be rewritten as follows:

$$\theta^* = \arg\max_{\theta} \mathbb{E}_{\tau_i \sim P(\mathcal{T}), h_i \sim H} \left[\sum_{t=0}^{T} \gamma^t r(s_t, a_t) \right] \tag{2.19}$$

Like in POMDPs, the belief state can be update based on a function f which depends on the previous belief state h_t, the current state s_t, the action taken a_t, and the current observation o_s:

$$h_{t+1} = f(h_t, s_t, a_t, o_t) \tag{2.20}$$

The function f is not known and in many applications non-linear. Since training a policy module already requires a lot of data, a general approach is to make use of this data to learn the function f and thus obtain a method to infer the belief state.

Additionally, the problem requires to handle sequential data as we are looking at transitions that get updated over time. Section 2.4 will cover the theory behind extracting data from sequences of information and present an approach applicable to MRL.

2.3.3 Task variations

As mentioned above, the goal of MRL is to find a model that performs well for a set of different tasks. The question arises is: What type of task variations are considered?

Since the robots are limited by the laws of physics, the tasks need to be consistent with the robot dynamics. With this in mind, an agent may be able to perform different types of tasks. There are multiple ways of subdiving tasks in different groups. In this work, the division is done into parametric tasks and non-parametric tasks. An illustration of this is shown in Figure 2.3

Parametric tasks:
Parametric task variations are defined as the set of tasks where the difference can be described by a set of parameters and the basic structure of the reward function remains the same. One example is an agent moving to a specific position. The parameter here is the position to where the robot is commanded to move; different tasks are differentiated by different positions.

Non-parametric tasks:
On the other hand, non-parametric tasks variations don't share the same reward structure

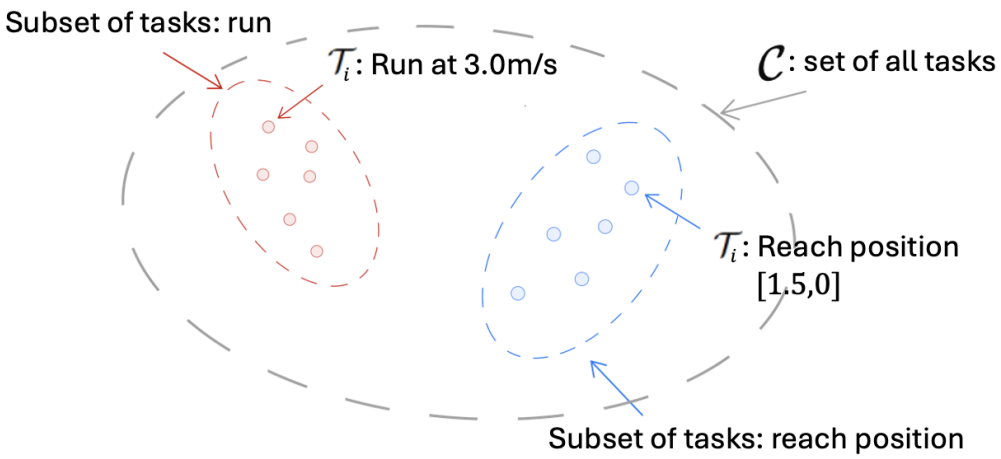


Figure 2.3: Example of task distribution for a robot able to perform locomotion tasks.

and are fundamentally different. Two different non-parametric task variations for a robot may be to move to a certain position or run at a certain velocity. The second reward, unlike the first one, cannot be calculated only by the current position; it needs a different calculation and is therefore fundamentally different.

It is also possible to combine parametric and non-parametric task variations to obtain the set of all tasks depicted by $\mathcal{C}$ in Figure 2.3.

2.4 Sequential Learning

As mentioned above, the inference module needs to be have the ability to handle sequential data, i.e. the belief state which influences the action needs to be updated sequentially. The reason for this requirements is that in most scenarios it is not possible to infer the task with a single transition. Updating the belief state sequentially helps reduce the uncertainty by accumulating evidence. However, often the model of the transition, denoted by $P(s'|s, a_t)$ and $P(o_{t+1}|b_t, a_t)$, is not known and thus Equation 2.3 cannot be used for updating the belief state. Like in RL when the model is not known, research has shifted towards learning it and draw on the potential of NNs.

DL models dealing with sequential data are called recurrent models. Recent advances in Large Language Models have made the research of these of models very popular [Bro+20]. The focus has been set on improving the ability to extract relevant information from sequential data at the same time as maintaining important long-term dependencies. Transformers have emerged as a powerful tool. In this work, however, MRL scenarios are observed which are not affected by the problem of long-term dependencies. Tasks can be inferred from few transitions. For this reason, simpler networks can be used.

Recurrent Neural Networks (RNNs) can be seen as the basis of sequential DL models. The main idea behind RNNs is to maintain a hidden state (Equation 2.21) similar to the belief

state described above. This hidden state is updated every time a new observation is received. Updating the belief state is not straightforward due to non-linear relation between sequential input and output. However, these non-linearities can be handled by NNs. The most simple representation of RNNs can be described by following equations:

$$h_t = \sigma(W_{hh}h_{t-1} + W_{xh}x_t + b_h) \tag{2.21}$$
$$y_t = W_{hy}h_t + b_y \tag{2.22}$$

where:

- x_t is the input at time step t,
- h_t is the hidden state at time step t,
- y_t is the output at time step t,
- W_{hh}, W_{xh}, W_{hy} are the weights of the network,
- b_h and b_y are the biases,
- σ is the activation function

These networks have an issue called vanishing gradient problem which stems from back-propagating through time. During training, backpropagation happens at every time step. Normally, the activation function is chosen to be the sigmoid function. This constraints the outputs to be in the range $[0, 1]$. For longer time steps, terms < 1 get multiplied during the gradient backpropagation and the gradient vanishes, i.e. moves closer to zero.

To solve this problem, Gated Recurrent Units (GRUs) extended RNNs with gating mechanisms. Each unit is complemented by two gates: a reset and an update gate. The reset gate determines how much past information to forget. It is important since forwarding every bit of information is not feasible. This gate learns to filter information that is important for future states from irrelevant information hidden in the observations. From this the candidate hidden state $\tilde{h}_t$ is derived:

$$r_t = \sigma(W_r[h_{t-1}, x_t] + b_r) \tag{2.23}$$
$$\tilde{h}_t = \tanh(W[r_t * h_{t-1}, x_t] + b_h) \tag{2.24}$$

The update gate determines the amount of past information that is important to forward to the current hidden state. It serves as a weight to update the current hidden state based on the current belief and the past belief. At the same time, it decides how much of the past information will continue through the network.

$$z_t = \sigma(W_z[h_{t-1}, x_t] + b_z) \tag{2.25}$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t \tag{2.26}$$

2.5 Representation Learning

The topic of representation learning focuses on compressing data and representing it in a different dimensional space such that the minimum amount of data is lost. In the general

procedure, the original data is mapped to a lower dimensional space which can be used for further calculations. This helps retain the most important information which makes subsequent tasks easier then if the original, high dimensional data were to be used.

There are multiple methods that deal with dimensionality reduction. Traditional techniques like PCA and singular value decomposition are commonly used. These methods take the data and compress it to a lower dimensional space with the goal of finding the projection that minimizes the data loss. The basis of these methoda are matrix projections and matrix decompositions, thus, the data can be reconstructed from linear transformations of the lower dimensional features. If the data is formed by a non-linear transformation of the extracted features, this methods are less effective. For the complex dynamics of robots, this approach is usually not optimal since the relations are often non-linear.

Autoencoder (AE) are a different, more advanced type of dimensionality reductors. These can be constructed with NNs making them able to handle non-linear relations. AE learn features that best describe the data in a given lower dimensional space than that of the original data. The architecture of this method is shaped like an hourglass. A simplified representation of this can be seen in Figure 2.4 which shows the split of the data flow in two symmetrical parts: encoder and decoder.

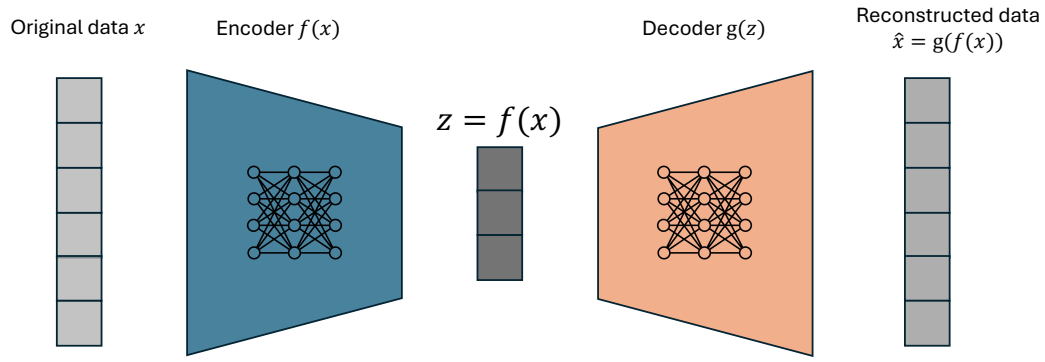


Figure 2.4: Graph showing the symmetric architecture of the AE. The encoder and decoder are both depicted by a block containing a NN insinuating that the computation is done by NNs.

The encoder maps the data $x \in \mathbb{R}^d$ to a latent representation $z \in \mathbb{R}^h$. Typically, as stated above, $h < d$. This mapping is done by the function $f(x)$.

The decoder then attempts to reconstruct the underlying data x from the latent representation z. The result is a reconstructed representation $\hat{x} \in \mathbb{R}^d$ obtained by the reconstruction function $g(z)$. This is done to reduce the amount of information that is lost during the dimensionality reduction. If the original data x can be reconstructed completely, no relevant information is lost.

AEs serve as a powerful tool to learn latent representations of high dimensional inputs. However, it is not well suited to handle uncertainty, which is very present in MRL due to the similarities to POMDPs. Variational Autoencoder (VAE) is an extension of original AEs that maintains the ability to represent non-linear relations, but has a more stochastic nature. Being able to combine stochasticity with the latent space within the task inference part of MRL enables it to have the capability of measuring uncertainty in the process. This results in more robust decision making. Making use of this, [Rak+19] showed good results for the problem

of MRL. As in AEs, the goal is to reconstruct the original data as good as possible from a lower dimensional representation:

$$L(x, \hat{x}) = ||x - \hat{x}||^2 \tag{2.27}$$

where $\hat{x} = g(f(x))$ and $g(x)$ is the function learned by the decoder that reconstructs the input from the encoded data. Since both networks are trained with the same loss function, the decoder moves the encoder to learn information that is important to reconstruct the input as close as possible with the specified latent dimensions.

The addition that differentiates VAEs from traditional AEs is that the latent variable doesn't get represented as a single point in the latent space but as a distribution, providing a probabilistic representation. Usually, Gaussian distributions are used for the latent representation. The training objective is the same as in classic AEs; recover the original data, while reducing the dimensions in the latent space. The loss for the VAE is composed of two parts, the reconstruction loss and the KL divergence:

$$\begin{aligned} \text{Reconstruction Loss} &= -\mathbb{E}_{q(z|x)}[\log p(x|z)] \\ \text{KL Divergence} &= D_{KL}[q(z|x)\|p(z)] \\ \text{VAE Loss} &= \text{Reconstruction Loss} + \text{KL Divergence} \end{aligned} \tag{2.28}$$

The reconstruction loss is the expected log likelihood of the data given the latent variables. This encourages the model to reconstruct the data from the given latent representation. The second term, serves as a regularization term and measures the distance from the distribution of the latent representation to the prior distribution. Intuitively, it can be thought of as a term that enforces a smooth and continuous latent space where close points in the input space are mapped closely in the latent space. Appendix A shows a step by step derivation of this loss.

2.6 Meta Reinforcement Learning for Non-parametric Task Variations

VAEs make it possible to represent parametric task variations in a compact manner. However, they don't suffice to cluster non-parametric task variations, since it is not possible to represent multiple clusters with a single Gaussian. Previous work [Bin+15] has combined the stochastic representation of the VAE with mixture models, allowing the clustering of latent variables. The goal is to cluster similar task variations that only vary in parameters together and have different clusters for fundamentally different tasks separated. The limitation of this approach is that the amount of clusters need to be determined beforehand which limits its applicability to non-continual learning.

To solve this problem Bing et al. made use of the Dirichlet Process Mixture Model (DPMM) introduced by Antoniak in 1974 [Ant74] which allows to simultaneously infer the amount of clusters and assign the data to those clusters. In some dynamic learning problems this is beneficial because the number of clusters cannot be determined beforehand. This section will introduce the theory behind DPMM which enables the clustering of latent variables and thus facilitates the training of the policy module.

2.6.1 Dirichlet Process

Dirichlet Process (DP) is a tool used in probability theory that enables the creation of models with infinite number of parameters. This is done by creating a distribution over distributions. The concept was introduced by Ferguson in [Fer73] as follows: With H as the base distribution and α denoted as the concentration parameter, G is said to be drawn from a Dirichlet process $G \sim DP(H, \alpha)$, if for any partition of the sample space into measurable disjoint sets $A_1, ..., A_r$, the random vector $(G(A_1), ..., G(A_r))$ has a Dirichlet distribution with parameters $(\alpha H(A_1), ..., \alpha H(A_r))$. The vector $G(A_r)$ represents the probability of the subset A_r.
A common algorithm that offers an intuitive understanding of the DP as well as an algorithmical procedure is the Stick-Breaking Process.

Stick-Breaking Process

This process is used for sampling weights in a mixture model. The example used for this algorithm is the process of iteratively breaking a stick. The length of the broken part represents the weight for the k-th component. With the other part, the process is repeated recursively, leaving a shorter stick after every step. Mathematically, the sampling of weights can be described as follows:

$$w_k = \beta_k \Pi_{j=1}^{j=k} (1 - \beta_j) \tag{2.29}$$

β_k are sampled from a beta distribution with concentration parameter α, $Beta(1, \alpha)$.

In practice, this procedure is terminated after some iterations since the weights get increasingly smaller and thus insignificant after a few steps.

2.6.2 Dirichlet Process Mixture Model

The DPMM is an extension of the DP that is used to described clusters of data for which the number of clusters is not known beforehand. Given a dataset $X = \{x_1, \ldots, x_n\}$, the DPMM models each data point as being modeled from a mixture model. After each data point is assigned to the mixture model, the model is adapted such that it fits the model best. In practice methods such as Markov Chain Monte Carlo, Variational Inference or Gibbs sampling is used. This is further discussed in Section 5.3.2

2.6.3 Deep Dirichlet Process Mixture Model

The term Deep Dirichlet Process Mixture Model (DDPMM) was introduced by Li et al in [Li+22]. It combines the aforementioned DPMM with the power of feature extraction obtained from NNs. First, the raw data is passed through the NN and a latent or feature representation is obtained. On this data, DPMM is performed in order to cluster the latent variables. This allows to cluster complex, high-dimensional data based on important which have been extracted to satisfy an objective function. During learning, both the DPMM model and the NN are trained. This creates a NN that outputs a latent space which can be clustered and a DPMM model that clusters based on informative features.

In summary, the DPMM is there to guide the neural network to encode the information in such a way that the resulting latent representation is clustered into different tasks. This is

achieved by calculating a loss from the DPMM which is added to the VAE loss. In an ideal scenario, the both losses are evenly balanced such that the information is properly clustered, while maintaining the most amount of information possible.

2.7 Hierarchical Reinforcement Learning

Hierarchical Reinforcement Learning has turned out to be helpful for complex Reinforcement Learning tasks. The idea behind it is to decompose a complex task into smaller tasks which are easier to solve. The results can then be combined to solve the complex task. This approach utilizes a hierarchy of policies.

Consider as example a two level hierarchy, where π^{hi} and π_i^{lo} are respectively the high-level policy and a set of low-level policies. The high-level policy acts as the orchestrator of the low-level policies.
Analog to standard Reinforcement Learning algorithms, the goal is to maximize the accumulated reward function. Here, each policy is trained to maximize its corresponding expected return:

Expected return for high level policy:

$$\mathbb{E}_{\pi^{hi}}\left[\sum_{k=0}^{\infty}\gamma^k R(s_{t+k}, o_{t+k}, s_{t+k+1})\right] \tag{2.30}$$

Expected return for lower level policy:

$$\mathbb{E}_{\pi_i^{lo}}\left[\sum_{k=0}^{\tau-1}\gamma^k R(s_{t+k}, a_{t+k}, s_{t+k+1}) \mid o_i = o\right] \tag{2.31}$$

where the additional term o is introduced as the option or the subgoal.

The higher level policy selects the sub-policy with the highest expected return. The sub policies are commonly denoted as options o_i. This then performs a lower level control to achieve the specified sub-goals.

Intuitively, this can be explained with a simple example:
Almost all human interactions are based on many small tasks. An example would be the action of cooking. Cooking can be decomposed into many smaller subtasks:

- **Ignite the fire:** Depending on the available tools this may include turning on the stove or striking a match.
- **Place the pan on the fire:** This action includes picking up the pan, placing it on the stove and releasing it.
- **Adding ingredients:** Depending on the recipe all ingredients need to be placed simultaneously or in a successive manner.

At the same time, all these processes can further be decomposed into simpler actions. In order to place the pan on the fire, the pan needs to be reached and grasped. Then the pan needs to be moved to the heat source and released. All these actions are also a combination of simpler

actions. This decomposition can be done until the most basic movements are reached which for the mechanical movement of the body would be the movement of the joints. The process, however, does not end there. A further breakdown of the joint movements can be seen at muscular level which in turn can be segmented further.
As illustrated here, the level of hierarchies can be decomposed indefinitely and the harder the task, the more levels of hierarchies form the combined policy.

This hierarchical structure can be seen everywhere in the real world. From animal movements to economics. This strategy not only makes problem-solving more efficient by optimizing every hierarchy of the process, but it can also strongly improve the learning speed and adaptability. For this reason it is also beneficial to take this method into consideration when training RL agents.

Chapter 3

Related Work

This section will cover recent work done on related topics to this thesis. First of all, the research area of MRL will be analyzed. In this subsection we will present three different focus directions of MRL, namely recurrent-based, gradient-based and inference-based MRL. Additionally, this section will also present work done in the fields of Domain Adaptation (DA) in (Meta) RL and Hierarchical (Meta) RL.

3.1 Meta Reinforcement Learning

The term MRL encapsulates two different areas of Machine Learning (ML). It is a composition of Meta Learning (MetaL) and RL. MetaL is often described as "learning to learn". Combined with the premise of RL which is to learn from interaction with the environment, it results in an agent that learns to learn from exploring the environment; i.e. we look for a model that can learn how to adapt quickly to new tasks by interacting with the environment.

MRL algorithms can be classified in different ways. There is the distinction between few-shot and zero-shot adaptation, where it classifies algorithms based on the amount of iterations it takes to find a reasonable result. Algorithms can also be model-based or model-agnostic. The third categorization is the one we will be using to categorize different approaches. This categorization has been prior used in related research [Bin+15], distinguishing between gradient-based, recurrence-based and inference-based MRL.

3.1.1 Gradient-based Meta Reinforcement Learning

The first of the three categories is known as gradient-based MRL algorithms. The foundation for this type of algorithms was laid out in [FAL17]. The main idea of the algorithm called Model-Agnostic Meta-learning (MAML) is to have a model that quickly adapts to new tasks by doing few gradient updates. In order to achieve this, Finn et al. designed an algorithm that finds the initialization parameters θ from which it is easiest to find the optimal parameters for a variety of tasks. Their method is comprised by following update steps:

From the task distribution $p(T)$ a prespecified number of tasks are sampled. For each task, new parameter sets θ_i are obtained independently through backpropagation. The new parameters each converge to carry out their respective tasks τ_i. In the next step, the method tries to find a balance such that all θ_i can be found with the least amount of effort possible. This is achieved through Eq.3.1.

Here, L_{τ_i} denotes the loss function for each task τ_i, and the model parameters are updated by moving in the direction that reduces this cumulative loss, guided by a learning rate β. This process effectively tunes the model to be well-suited for a rapid and efficient adaptation to a variety of new tasks as long as the tasks are obtained from the same distribution.

$$\Theta \leftarrow \Theta - \beta \nabla_{\Theta} \sum_{\tau_i \sim p(\tau)} L_{\tau_i}(f_{\Theta_i}) \tag{3.1}$$

Based on this work, numerous studies have been conducted and some advancements have been published. These include [LSX19] where the authors use control variates to tackle the problem of high variance occurring from the estimation of the gradients. In [Son+20], Song et al. focus on the problem of having to estimate Hessians during back propagation by replacing the gradient descent algorithm with Evolution Strategies. [Li+17b] extends the algorithm MAML by learning not just the initialization of parameters, but also learning the learning rate of said parameters. Experiments showed a faster convergence of this algorithm compared to MAML. [Al-+18] takes also the work from Finn et al. as inspiration and extends its approach to make it work in non-stationary environments by focusing on the MDP assumption that the data is dependent on time. [Gup+18] argue that MAML is not good at exploration since it cannot introduce history-dependent stochasticity. Gupta et al. incorporate random noise based on a latent space which can be learned.

3.1.2 Recurrence-based Meta Reinforcement Learning

Another approach that has been followed by numerous researchers is that of recurrence-based MRL. The core concept is to provide the agent with a memory mechanism, allowing it to make optimal actions based on both current and past observations.

This concept has been applied before in the supervised learning domain [HYC01; San+16]. A more recent work [Wan+17] used this in the setting of RL with variable tasks. Since the policy model takes the input of the recurrent network, two agents in the same state, but with different rewards in the past due to a distinct tasks, can have different outputs. This makes this approach suitable for Meta Reinforcement Learning as an extension to classic RL. Analog to [Wan+17] a paper which was published concurrently [Dua+16], also follows the same idea. The main difference is that Dean et al. put the focus on combining the memory aspect and the decision taking in one recurrent network. In [Mis+18] the authors replace classic recurrent networks with a combination of dilated 1D convolutions and self attention layers. This enables it to access information from a longer time range (dilated convolutions) while being able to focus on important data (attention layers).

3.1.3 Inference-based Meta Reinforcement Learning

Rakelly et al. [Rak+19] argue that the previously described methods are sample-inefficient and that it can be overcome by decoupling the actor and the task inference. Their method creates the basis for inference-based MRL algorithms. The inference module consists of a variational inference network that creates a bottleneck only containing important information for solving the task. The data contained in the bottleneck is also referred to as latent space and serves as additional input to the action module. Multiple extensions have been made since [Rak+19] was published.

The method presented by Lerch in [Ler20] offers a valuable extension by incorporating Gaussian Mixture Model (GMM) into the inference module. This incorporates the ability to encode and later distinguish non-parametric task variations. In an ideal scenario, each different non-parametric variation is parameterized by a unique cluster. Inside this cluster the parametric variations have different representations. Additionally, the method presented in [Ler20] decouples the inference training from the SAC training. This is done by training the decoder through predicting future steps of the MDP and comparing it to the actual steps. This is a key feature needed to achieve the goal presented in this thesis, which includes decoupling inference module and policy module.

Bing et al. [Bin+15] took the inference module and extended it with the DP. By doing so, it opened the possibility to continual learning thanks to the potential of representing infinite clusters gained by introducing Bayesian non-parametrics into MRL.

Durmann presented an idea that serves as a basis for a new research topic [Dur23], reutilizing the inference module for different agents. In [Dur23] Durmann transfered the inference module on a simple toy environment with different characteristics effectively. This thesis will focus on expanding on this research and creating a general solution which can be applied to a wider range of agents including ones with more complex dynamics.

3.2 Hierarchical (Meta) Reinforcement Learning

As presented above, the theory behind HRL is to decompose a complex problem into smaller, easier subtasks. This can also be seen as reducing the horizon in a Markov decision process. RL has seen great advances, but there are some limitations that still need to be solved. One of this limitations is to handle complex state spaces. Reason for that is the exponential growth of possible states with growing state space dimensions. Since exploration is fundamental in many RL environments, it becomes challenging to explore a representative amount of all possible state spaces when these increase in size and complexity. Hierarchically ordered policies have gained in interest among researchers [GM01; Geh+21] as it offers a way to more effectively explore these state spaces.
Applications in multi task setting can also be found in the literature [Nac+18].

More recent work has also leveraged the advantages of HRL for the MRL problem[Fra+17]. The work by Frans et al. decomposes the problem like in HRL. The algorithm works by learning a set of sub-policies that are useful for a variety of tasks and a master policy that serves as the orchestrator by choosing which sub policy to activate at each time step.

3.3 Domain Adapation in Meta Reinforcement Learning

The term DA is a concept in ML that refers to the ability of a model to perform in a different environment than the one it was trained on. Extensive of research has focused on improving the DA of deep NNs [GL15; Tob+17]. A widely researched application is the training of a network in simulation and transferring the obtained knowledge to the real world [Tob+17] due to the many advantages of training in simulation.

In [Li+17a] Li et al. train a meta learner to learn a model that quickly adapts to new environments. The authors employ a similar training procedure to MAML, but rather than creating a model that adjusts to new tasks, the task remains constant. The goal is to adapt to new environments in a few gradient update steps. A similar goal is pursued in [KSS20]. The procedure, however, is different. Instead of learning a good initialization for a model from which to learn fast, it creates a model that learns the loss function. When applied to an unseen environment, the model can then be trained with that loss function. This is advantageous because finding an objective function is not always trivial and requires expert knowledge.

As mentioned above, the terms MRL and DA have been used together before. But the idea in previous research is to learn a model that leverages MetaL to adapt to new environments. The focus of this work is to train a meta learner capable of solving different tasks in a different environment than it is validated on, so the mentioned work does not apply for this use case.

Chapter 4

Problem Statement

Inference-based MRL has shown promising results when training simple models to perform different tasks. However, previous approaches still have some shortcomings that hinder their implementation in more complex agents. Durmann [Dur23] identified two problems that inference-based MRL algorithms face.

Encoder shortcut learning. This problem arises due to the design of the encoder training objective. During training, the goal of the VAE is to extract features with which it can later reconstruct future rewards and states. With this objective, it is possible to extract simple features that serve this purpose but do not carry enough information to infer the task at hand. Hence, the policy module is unable to learn a policy for different tasks. The opposite of a model that suffers from encoder shortcut learning is one that can be employed on different agents as it learns to focus only on the important features.

Chicken-and-egg problem. Durmann defined this as the problem that arises due to the dependency between inference module and policy module [Dur23]. In order to learn a robust inference method, the agent has to explore the environment. However, this is not possible if the agent is unable to perform basic motions. On the other hand, without an informative task embedding, the agent is not able to learn the proper Q function. This problem arises because the task encoding is the only information the Q function receives about the task. If the embedding is arbitrary, the best the Q function can learn is an average, which hinders learning for varying tasks. As this serves the purpose of guiding the policy, the agent will not be able to differentiate between good and bad actions. This is a cycle that is complicated to avoid in a system with complex dynamics.

Summarized, it can be attributed to challenges in state exploration. This problem arises due to challenges in navigating complex dynamics. For simplification, we categorize the exploration into two different types, low-level and high-level. in this categorization, we separate the observation space in two groups: global variables (position, velocity, orientation, etc.) and robot-specific variables (joint positions, torques, etc.). For robots with many degrees of freedom, exploring the global variables generally poses the harder problem as the actions are not directly reflected in the high-level observations. However, the global variables are directly coupled with the rewards. Therefore, exploration at this level is essential for the learning process. In scenarios where the agent can be commanded with high level instructions, this chicken-egg problem is not present.

In addition to addressing these issues, there is potential to improve the training of different agents in the MRL scenario. Based on the assumption made in Section 2.1.1, which states that for different agents, the POMDP only differs in the reward function, the encoded fea-

tures used for task inference should not be dynamic-dependent but robot-agnostic. For the examples chosen in this thesis, reward functions are only dependent on global variables (position, orientation and velocity of the robot) not on joint coordinates specific to the agent. Therefore, the task inference module should be reusable for different agents. Following this reasoning, learning the inference module in one agent and reusing the learned model with a different agent should be possible. Such an approach can be highly beneficial if the agent on which the inference module is trained simplifies the exploration process.

In the following section, a methodology will be presented that decouples both parts of the algorithm-inference and policy-and solves the problem of exploration for agents with higher degrees of freedom, thereby addressing the above mentioned issues. The method presented will showcase a strategy to reuse the inference module for different agents while using an agent that simplifies the exploration of the global variables during training.

Chapter 5

Methodology

5.1 Overview

This section provides an overview of the method in order to facilitate the understanding of its procedure. The different components as well as the training process will be introduced and in later sections explained in more detail.

This method will work in the following way: An agent with a highly simplified dynamical model will be trained to perform different tasks, without prior knowledge of the tasks themselves. In order to perform the tasks, the agent will subjected to inference-based MRL training. The simplification of the dynamics is accomplished by reducing the agent to a simple sphere with mass 1, thereby removing any dynamic complexities. Further details regarding this implementation are given in 5.4. From the training process, both a policy and an inference module will be obtained. The task inference will be reused, without retraining, on a different agent.

The training process is divided in two phases:
MRL training on a simplified agent: First, the inference module will be trained on a highly simplified agent. This means that the relation between the action performed and the reward is clearly defined and known. Also, the degrees of freedom of the agent are reduced to the minimum that are required to perform the task, thereby, greatly simplifying the exploration and thus the training overall.
Following the algorithm introduced in [Bin+15], we begin with a data collection step. During data collection, the agent performs actions based on the current state and the latent parameter z. z is generated by the encoder from a fixed number of transitions containing information about the current state s, the reward r and the next state s'. The values are stored in a buffer such that they can be accessed for training.
Training alternates between encoder and decoder updates and policy updates with data sampled randomly from the aforementioned buffer. These steps are repeated to sample new data with the updated models and trained thereafter creating a cycle which is repeated until certain convergence criterion is achieved.
The architecture of this method is depicted in Figure 5.2.

Training new agent: To transfer the learned knowledge, the strategy is to train the new agent to mimic the movements of the simplified agent. The goal is to create a bridge between the inference module and the policy as described above and the new agent. This new agent will have a different dynamical model. To achieve this, a model is trained that converts high-level actions or subgoals to low-level actions that can be processed by the agent. These

high-level actions are the actions with which the simple agent is guided. During training, this actions are sampled from a distribution that lie in the same space as the movement of the toy. The training follows a similar structure as the one used for the policy update in the first step, with the distinction that it receives high-level actions as input instead of latent embeddings. This distinction makes the second part of the training a multi-task RL problem.

For both training steps, the SAC framework is used.

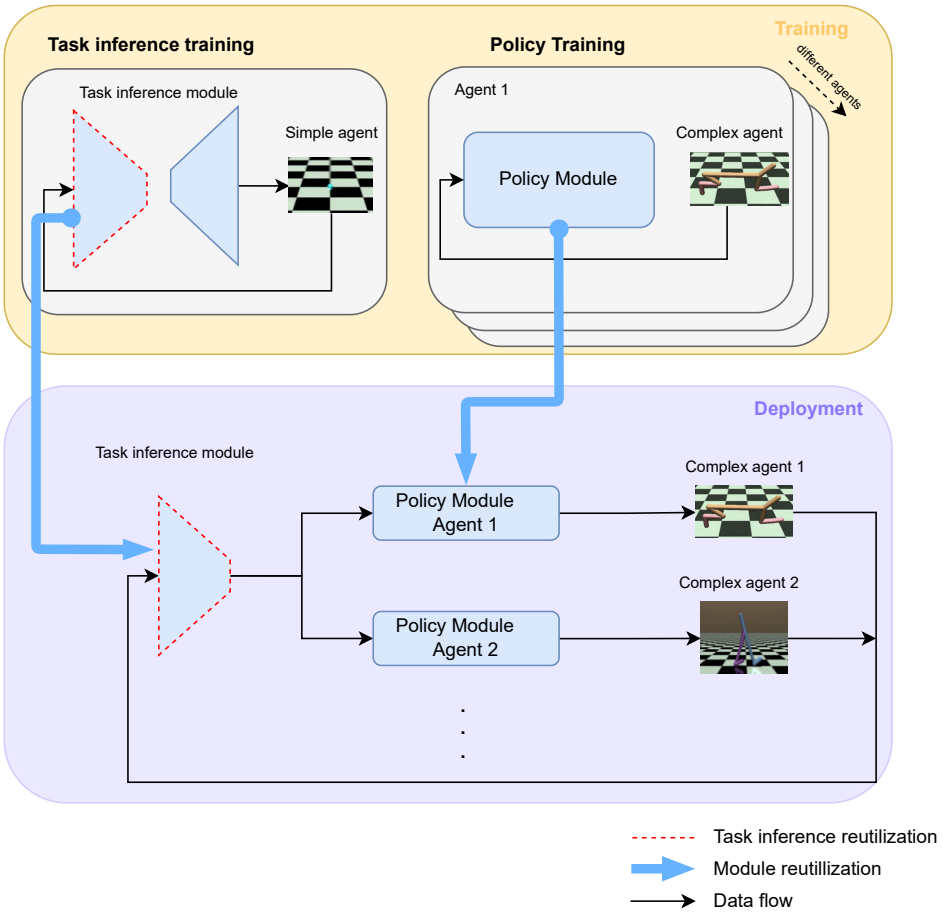


Figure 5.1: Overview of the method employed. The inference module is trained on an agent for which exploration is easy and then reutilized without retraining on different agents.

Once the training is completed, the different components are integrated and their functionality tested. The encoder generates latent embeddings which guide the high-level policy to create subgoals that move the agent closer to the goal. These are then processed by the low-level policy to create actions that can be applied to the new agent. This method can be used for different agents by only training the low-level policy on each agent. With this procedure we address the main goals of this thesis: solve the exploration problem and create a method to reuse the inference module across different robot designs. An illustration of the process is depicted in Figure 5.1.

5.2 Preliminary assumptions and prerequisites

For this methodology to be successful, several assumptions must considered. These assumptions originate from the goal of using one inference module for different agents. To achieve this, the agents must share certain common characteristics.

Tasks need to be robot-agnostic. The first and most important prerequisite is that the goals need to be robot-agnostic. The tasks need to be solvable by all robots considered, regardless of their design. This means that the goal cannot depend on the structure of a specific robot. A valid example would be the navigation through a series of checkpoints as this can be performed by a legged, a wheeled robot or an aerial robot like a drone. While the dynamics are important for the execution of the task, they are not critical for predicting the task.

Similar goals and reward structure Different agents must have a similar reward calculation structure. Since the inference module takes the reward as input to predict the task, it is necessary to employ the same reward calculation for different agents. It is not necessary to choose exactly the same reward function but the structure must be similar. Consider the reward calculated by the distance to a certain position. One agent may calculate the distance to the torso and another to the center of mass. This will not affect the performance as long as the same criteria are used consistently.

It is also worth mentioning that the goals should similar, and also the dimensions of the states used for the reward calculation need to be in a reasonable range. One example where this may not be the case is when trying to train an inference module that identifies the goal position in a 2-dimensional environment. If the goal is to reuse this for both a robot arm and a quadruped robot this will likely not work. Reason is that the quadruped robot unless reduced in size will move in a significantly larger range than the robotic arm. The robotic arm will have a workspace of around $1m^2$. In contrast, the workspace of the quadruped robot could reach dimensions 10 times higher. Thus, the inference module cannot be reused for both agents. If, however, the dimensions relate closely, the inference module may be reusable despite the apparent disparity between the two types of robots and its different applications.

Expert knowledge Another prerequisite is the availability of expert knowledge. This is needed to create an abstract representation of the more complex dynamics depending on the task. For the tasks of locomotion this problem simply reduces to taking the positions, velocities and orientations.

Locomotion tasks are well suited for this method since every robot performing locomotion tasks has fundamental variables such as position, velocity and orientation of its center of mass which can be easily obtained. Thus the body of the robot can be simplified to a point mass for the calculation of the rewards. This can be done for a wide range of robots. In the experiments section, three different variations of robot models are shown.

5.3 Task Inference Module

The goal of the task inference module is to learn the mapping between trajectories of MDPs to a lower dimensional space. This lower dimensional space, also called the latent space, needs to contain sufficient information about the task from interactions with the environment, such that the policy module is capable of distinguishing between tasks and executing the desired

task.

Training the inference module to produce an embedding from which the MDP can be predicted, generally offers a good characterization of the task [Ler20]. A task is characterized by the rewards obtained from a set of different states. For a single transition, the reward value can be ambiguous for different tasks. For trajectories of sufficient length, however, multiple reward-state combinations can only be assigned to a specific task. This argumentation makes the objective of predicting the future state and reward using a decoder a reasonable choice. As shown in [Bin+15], it also offers a good representation from which the policy module can retrieve the necessary information to form a good action.

The Markov property states that the next state only depends on the current state. Therefore, when dealing with a MDP, only one transition should be needed to perform an action. This does not match with the need for longer trajectories for the task inference as mentioned previously. This discrepancy can be explained with the argumentation that the hidden state is an extension of the actual state as discussed for POMDPs problem. Thus, the current state is described by the state of the robot in addition to the belief state. From this observation, the requirement arises of a module that not only infers the task but maintains a belief state needed for future estimations.

5.3.1 Encoder Loss

The requirements imposed on the inference module can be met by an architecture consisting of a VAE where the encoder is composed by GRUs to handle sequences of transitions. For the decoder, a simple Multilayer Perceptron (MLP) with two output heads is used as a linear regression problem as this has shown to deliver good results before [Bin+15]. One head is used for the reward prediction and one for the state prediction. In the following it will be shown how to derive a training objective from the encoder input.

The training objective of the VAE is to maximize the Evidence Lower BOund (ELBO) (derived in Appendix A). By taking the negative of the ELBO, we derive the loss function used for updating the weights of the NNs:

$$\mathcal{L}(\phi,\theta;x) = -\text{ELBO}(\phi,\theta;x) = -\Big(\mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x \mid z)] - \text{KL}(q_\phi(z \mid x)\|p(z))\Big) \tag{5.1}$$

The loss function consists of two parts:

1. **Reconstruction Loss:** Reducing this loss is equivalent to maximizing $\log p_\theta(x|z)$, i.e. maximizing the probability of reconstructing the original data from the latent distribution. This trains the encoder to produce the latent variables that retain the relevant information for this purpose.

   Since the decoder aims at reconstructing two quantities, the reconstruction loss consists of two terms. Through the assumption that the conditional probability $p_\theta(x \mid z)$ is a Gaussian parameterized by the latent variable z, one can derive an easy computable reconstruction loss as the log-likelihood becomes the mean squared error:

$$\log p_\theta(x \mid z) = -\frac{1}{2\pi^2} \mid\mid x - f(z) \mid\mid^2 + \text{const} \tag{5.2}$$

- State loss: The first loss is the deviation from the predicted state to the actual state. This is represented as the Mean Squared Error (MSE) between predicted state $\hat{s}$ and current state s:

$$\text{MSE}(s, \hat{s}) = \frac{1}{N} \sum_{i=1}^{N} (\hat{s_i} - s_i)^2 \tag{5.3}$$

- Reward loss: Analogously, the same loss is applied for the reward prediction.

$$\text{MSE}(r, \hat{r}) = \frac{1}{N} \sum_{i=1}^{N} (\hat{r_i} - r_i)^2 \tag{5.4}$$

  Both losses are added and weighed by a respective weight α_i to form the reconstruction loss:

$$\text{Reconstruction Loss} = \alpha_1 \text{MSE}(r, \hat{r}) + \alpha_2 \text{MSE}(s, \hat{s}) \tag{5.5}$$

2. **KL Divergence:** Each latent variable is assigned to a cluster as explained in Section 5.3.2. To better fit the data, each cluster is compared to every latent variable weighted by the probability of it belonging to the cluster.The mean is then taken and serves as the KL divergence loss:

$$\text{KL divergence} = \sum_{i=1}^{N} \sum_{k=1}^{K} KL\left(q_\phi(z_i|x_i)||p(z_k)\right) \tag{5.6}$$

Together, both losses compute the overall inference loss:

$$\mathcal{L}(\phi, \theta; x) = \text{Reconstruction Loss} + \beta * \text{KL Divergence} \tag{5.7}$$

where the parameter β serves as the regularization coefficient. During training, this loss backpropagates through the network and updates the encoder and decoder.

5.3.2 Cluster Model

The derivation of the loss described above assumes the existence of clusters. These clusters, however, first have to be created and refined. [HS13] introduces the term of memoized variation inference which serves as a method to update the mixture model. After initialization it consists of two steps: parameter update and birth and merge moves.

Initialization: This can be done via the stick-breaking process as described above. Each cluster then gets assigned a weight by iteratively breaking a stick of length one.

Parameter update: In order for this method to be valid, the assumption is made that the data is generated from sampling from the base distributions H and F with parameter λ_0 which belong to the exponential family of distributions:

$$H(\phi_k|\lambda_0): \; p(\phi_k|\lambda_0) = \exp\left(\lambda_0^T t_0(\phi_k) - a_0(\lambda_0)\right)$$
$$F(\phi_k): \; p(x_n|\phi_k) = \exp\left(\phi_k^T t(x_n) - a(\phi_k)\right)$$

From the stick breaking process, every cluster receives a weight w and the parameters sample from the base distribution H. Thus the joint distribution can be described as follows:

$$p(\mathbf{x}, \mathbf{z}, \phi, \nu) = \prod_{n=1}^{N} F(x_n|\phi_{z_n}) \,\mathrm{Cat}(z_n|w(\nu)) \prod_{k=1}^{\infty} \mathrm{Beta}(\nu_k|1, \alpha_0) H(\phi_k|\lambda_0) \tag{5.8}$$

This is the true, but intractable posterior of the data x. In order to find the parameters that best approximate the data, a variational distribution q is introduced. As described in [HS13] q is parameterized by following parameters:

- **Cluster assignment z_n:** This is a categorical distribution which determines the probability of a data point to belong to the cluster n. It is parameterized by the parameters r_{n_i}:
$$q(z_n) = Cat(z_n|r_{n_1}, ..., r_{n_K}) \tag{5.9}$$

- **Cluster weights ν_k:** These are the weights assigned to the cluster by the DP stick-breaking process. Every cluster is assigned a weight by a Beta distribution with parameters α_{k_0}, α_{k_1}. The parameters vary between clusters for increased flexibility in inferring the true posterior:
$$q(\nu_k) = \mathrm{Beta}(\nu_k|\alpha_{k_0}, \alpha_{k_1}) \tag{5.10}$$

- **Cluster parameters θ_k:** The shape of the each cluster k is defined by the parameters θ_k which are assumed to come from the base distribution H generally chosen to be a Gaussian distribution. Prior knowledge of the data distribution is helpful in this step since the selection of the prior distribution H will affect the shape of each cluster:
$$q(\theta_k) = \mathcal{N}(\theta_k|\lambda_k) \tag{5.11}$$

The true posterior is then approximated by the joint distribution of the subdistributions mentioned above. By maximizing q, the ELBO gets minimized:

$$\mathrm{ELBO} = \mathbf{E}[\log p(x, \nu, z, \phi|\alpha_0, \lambda_0) - \log q(\nu, z, \phi)] \tag{5.12}$$

Due to the interdependency between parameters, the update has to be performed sequentially, i.e. when one parameter is updated, the rest is frozen. This is done iteratively until a certain convergence criterium is reached.

Birth and merge moves: Hughes and Saderth [HS13] also employ birth and merge moves in their algorithm. This makes it possible to capture new and dynamic data in the model. In MRL this is essential since the embeddings change after every iteration.

- **Birth moves:** The pourpose of birth moves is to create new clusters to represent clusters of data that are not well represented by the existing separations. The initialization of new clusters can be done in different ways but are only accepted if the ELBO increases.

- **Merge moves:** It can also be the case that too many clusters exist making some of them redundant. This move aims to combine pairs of clusters. The decission is taken based on the impact on the ELBO. The parameters of the combined cluster is calculated as a weighted average.

The DPMM serves as a method to cluster the embeddings obtained from the encoder. With this output, the VAE loss is computed closing the cycle between the VAE and DPMM. This dependency calls again for an iterative update where first the mixture model is computed and then the VAE is updated.

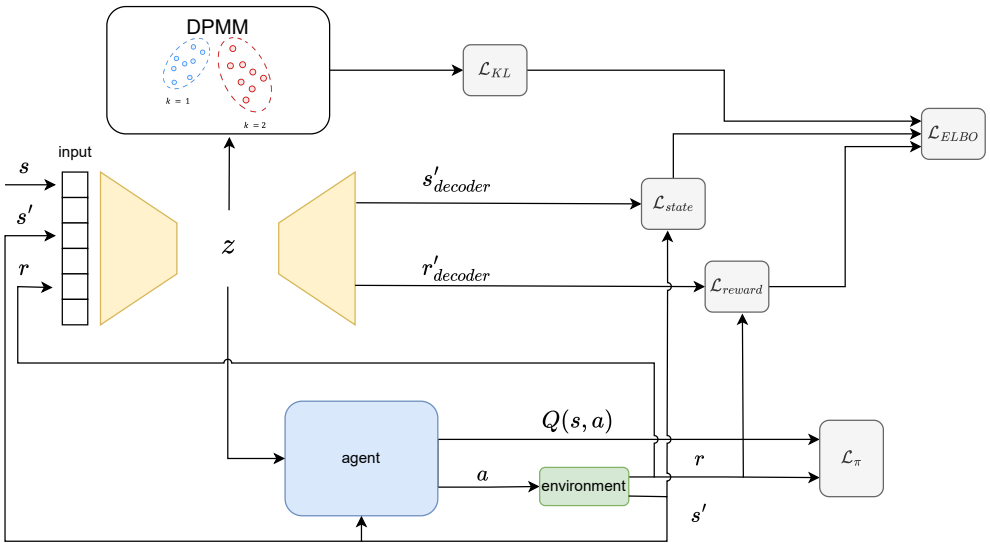


Figure 5.2: Training architecture of the task inference

5.3.3 Encoder Input

Another important aspect is the selection of data for the input of the encoder. Based on the input, the encoder needs to extract the relevant information to classify the task and guide the policy. Too much data could lead to overfitting and too little could cause underfitting. [Bin+15] showed good results with an input consisting of transitions constituted by previous observations, actions, rewards and next observations. This data is enough to infer any type of tasks. The goal of this thesis is to create an inference mechanism that is independent of the agent dynamics. Therefore, it is counterproductive to include the agent-specific actions in the input vector. Thus, the inclusion of the action to the input vector will be omitted and the remaining input consists solely of previous observations, rewards and next observations.

Additionally, this will help improve the performance of the model. When training the toy agent with the action as part of the input, the encoder will most likely learn a relation between action and next state. During the transfer, however, the agent will not be able to replicate the action as precisely as the toy agent. For example, it is very complicated to control a cheetah to follow a trajectory with millimetric precision. This will introduce randomness into the model which may cause for issues when testing with the more complex agents. Thus, removing the action from the input will eliminate this source of error.

The intuition behind why this approach should be effective is that the processes observed here only depend on global variables like the position and velocity of the robot. This means

that the action is not directly needed for the reconstruction of the task.
On the other hand, the decoder needs to reconstruct the next step. Since the transition is dependent on the action, the decoder requires the action performed in the current state. This is, however, not a problem since the decoder will not be reused in the testing phase where the task inference module will be reused for a different agent and the action may belong to a different vector space.

As mentioned above, inferring the task does not require a large amount of transitions. This helps reduce the complexity of the feature embedding generator, namely the GRUs, as no complex structures like Transformers or attention mechanisms will be needed. Additionally, it enables the fixation of input length to the encoder. Having a fix input length facilitates the adaptability to zero-shot learning.

Figure 5.2 shows the training procedure of the task inference module. To ensure that the space is explored, the policy guides the toy agent. The inference module works best if it is applied with a similar exploration strategy as it is trained on. For this reason, training the agent to maximize the return in the toy environment will ensure a similar exploration as during testing. Reaching exactly the same behavior between the toy agent and the target agent is not feasible due to errors and small deviations. Therefore, small randomizations are used during training. More details on the type randomization used and the reason for it are presented in section 6.4.3

5.4 Simplified Agent

Finding a model that serves as an abstraction of other agents is where the expert knowledge finds its use. The model needs to fulfill certain criteria:

- **Simplification**: The dynamics of the model need to be simplified in order to improve exploration. Also, simple models can be seen as a generalization of the more complex counterparts. These include details which are not present in the simple models. Thus a simple model can be applied to more complex models but not the other way around. From the separation at the beginning of this chapter into high- and low-level exploration, it's made clear that finding an agent which actions directly correlate to the variables with which the rewards are calculated, is the more reasonable choice.

- **Relatedness:** The abstracted model must behave similar to the models where the inference module is going to be reused with. The performance of the inference module is heavily dependent on the exploration of the observation space. If one agent moves significantly different then the complex counterpart, the inference module might struggle to understand and predict the task. That is, in the scenario where locomotion tasks are observed, all agents for which the inference module should be reused should have similar speeds and movements.

This thesis focuses on locomotion tasks and how an inference module can be reused in different agents. For this purpose, the most simple abstraction is to reduce the agent to a sphere. The actions directly influence the movement in the form of an external force. To emulate the movement of the robots, the sphere was simulated as a simple mass-damper system as depicted in 6.1 which follows the dynamic equation 5.13. All forces except the damping

forces and external forces are neglected. Exploring the environment with this agent becomes a trivial task.

$$m\frac{d^2(x)}{dt^2} = F_{ext} - c \cdot \frac{d(x)}{dt} - mg \tag{5.13}$$

with $(x) = (x, y, z)^T$ and $g = (0, 0, -9.81)$
For the first experiment, this agent will be simplified even further as described by Equation 6.1.
For future references, the agent represented by Equation 6.1 as well as Equation 5.13 will be denoted as the toy agent.

5.5 Policy Module

The goal of reusing the inference module for different tasks imposes certain requirements on the policy module. Firstly, the training must be decoupled from the inference module which is beneficial to the chicken-egg problem described in Section 4. At the same time, the policy must learn actions that performs tasks based on the information provided by the task embeddings obtained from the inference module.
Second, the agent must follow a similar exploration strategy as the one employed to learn the inference mechanism. Thus, it is crucial to find a model that allows the agent to move similar to the toy environment during the inference training.

A hierarchical-like structure is chosen to fulfill theses requirements. The result is a policy that consists of two levels or hierarchies, the lower and higher level controller. In the lower level controller, a policy is learned that guides the agent to move like the toy. The toy performs simple movements like move 1m to the front or run at speed 2m/s. The lower level policy learns to perform robot-specific actions that translate to these types of movements. It can be done in one of two ways: by learning one network that performs all tasks or learning different networks for different non-parametric task variations. The resulting policy will be referenced as the transfer function since it is the policy that transfers the actions from the domain of the complex dynamics to the toy domain, i.e. it is a bridge between the high-level actions and low-level control.

The high-level policy is trained to guide the lower level policy. Its output is the higher level actions like moving a certain amount or reaching a certain speed. The training offers two options: taking advantage of the policy trained on the toy environment during the inference training or retraining a higher policy from scratch directly on the complex agent. Both methods will be compared in Section 6.

All agents are trained with the SAC framework since it offers a good basis for training agents in continual RL scenarios. Figure 5.3 depicts a simplified version of the SAC method where the policy receives the task as input. This figure shows the dependencies of the different parts as well as the gradient flow necessary for the training.

5.6 Multi-Task Policy Learning

During the training of the low-level policy, the policy receives a task as input. Therefore, this part of the training falls under the category of multi-task RL. This section will explore how it

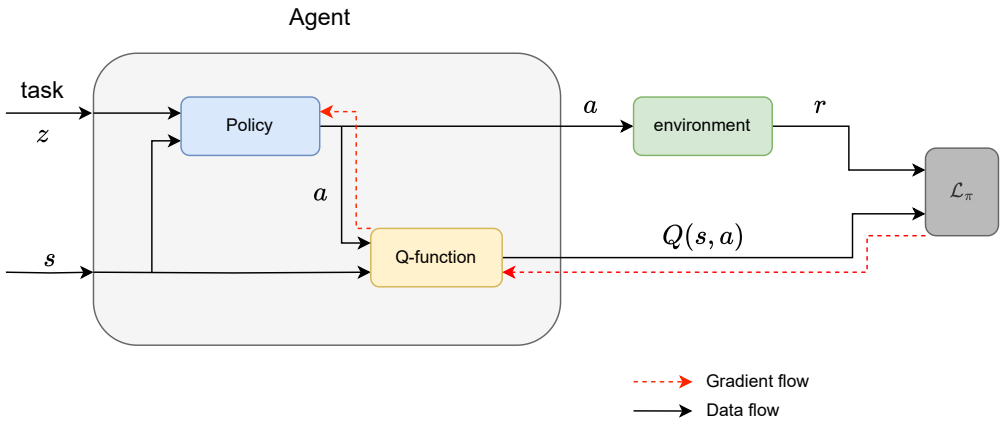


Figure 5.3: Training architecture of the SAC in the multi-task setting.

can be incorporated into the proposed method.

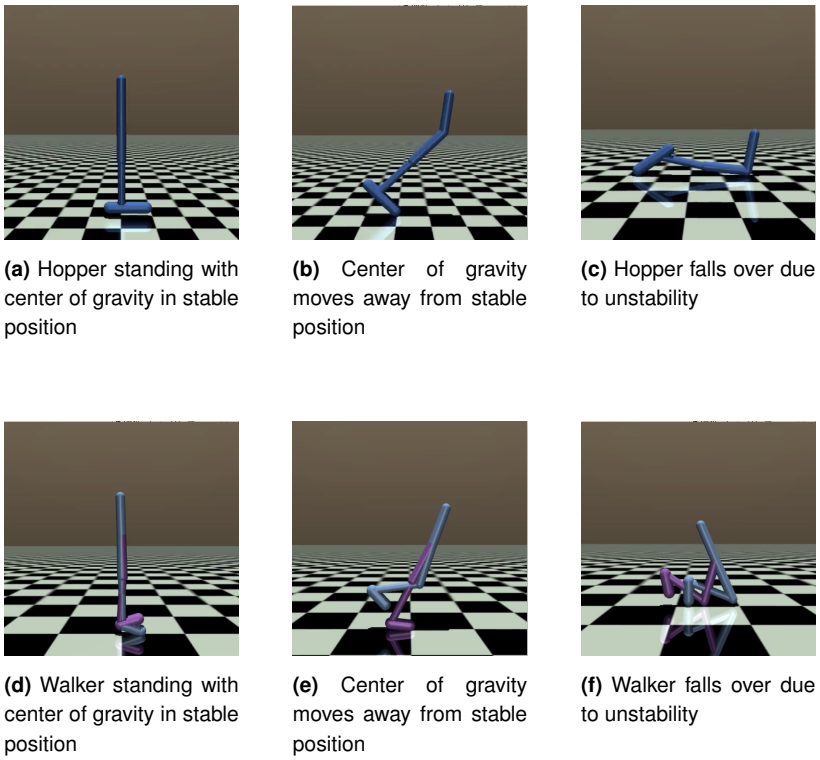
In multi-task learning the objective is to find a policy that generalizes to different tasks τ_i given by the task distribution $p(\tau)$:

$$\max_{\pi} \mathbf{E}_{\tau \sim p(\tau)} \left[\sum_{t=1}^{T} \gamma^t r_t^\tau \right] \tag{5.14}$$

where r_t^τ represents the reward at time step t for task τ.

The complication with multitask learning arises due to the potentially high variability in tasks. This can slow or even impede the robot from learning. The cause is a set of states with potentially conflicting goals. Each goal requires a different behavior that may oppose the behavior of a different task making the learning process unstable and unable to converge.

As mentioned in Section 5.5, multiple tasks can be achieved by either training different networks for different tasks with non-parametric variations or training a single model for all task variations. The first one eliminates the problem of opposing behavior between tasks, but requires more storage capacity.
Another significant downside of training multiple networks is that one network may not encounter states that are relevant for other tasks. For example, let's consider the cheetah agent with the goal of learning how to jump and run forward. If each task is learned in a separate model, the network learning the forward movement might not explore states where the agent is up in the air. This could lead to unexpected behavior if, during testing, the agent is instructed to jump and then run forward as the running model has not encountered a similar state during training. One potential solution is to randomize the starting state of the agent, although, this makes the training more complicated.
The option of training only one network has the benefit of generalization. Since it has to learn more than one behavior, the model needs to be more abstract and it may learn skills that can benefit different tasks. Therefore, the approach followed here is to train a single model.



(a) Hopper standing with center of gravity in stable position

(b) Center of gravity moves away from stable position

(c) Hopper falls over due to unstability

(d) Walker standing with center of gravity in stable position

(e) Center of gravity moves away from stable position

(f) Walker falls over due to unstability

Figure 5.4: These images show the hopper and walker in a mjuoco simulation. The instability arises from the fact that the the center of gravity rapidly moves away from the area enclosed by the points of contact with the floor. This makes it prone to falling over. Once the agent falls over, it is hard to recover.

5.6.1 Curriculum Learning for Multi Task Reinforcement Learning

Learning a policy to perform different tasks on a complex robot is not a straightforward task. Especially, when the robot has certain instabilities like in the case of the hopper and walker depicted in Figure 5.4. One solution to this problem is to start with simple training goals and, systematically, increase the difficulty of those as the robot's abilities improve. This method is known as Curriculum Learning (CL). It is also the way humans learn complex behaviors to solve difficult tasks.
By implementing CL, the NN starts by learning a simple behavior which it can tackle. Next, the difficulty of the tasks increases. With the understanding of the basic dynamics acquired from the previous learning step, the agent can better handle more complex goals. This process can be represented by the sequence of task sets $\tau_1, \tau_2, ..., \tau_n$, where τ_{i+1} has a higher degree of difficulty than τ_i. Equation 5.14 can be extended to describe the objective of multitask learning making use of CL:

$$\max_{\pi} \mathbf{E}_{\tau_i \sim p_i(\tau)} \left[\sum_{t=1}^{T} \gamma^t r_t^{\tau_i} \right] \tag{5.15}$$

$p_i(\tau)$ is the distribution of the task set τ_i. It must hold:

$$\bigcup_{i=1}^{n} p_i(\tau) = p(\tau) \tag{5.16}$$

This equation ensures that all tasks are addressed during the training procedure.

5.6.2 High- and Low-Level Interface

In this section we will inspect how to design the interface between the low-level policy and the high-level policy. To do so, two possibilities have to be taken into consideration given that the method will be tested for two separate training approaches for the high-level policy. First, the policy will be reused from the training on the toy agent. Secondly, only the inference-module will be reused and the high-level policy will be trained on the new agent.
Before making the distinction, it is necessary to clarify the process of training the low-level policy. Apart from the current state of the agent, the input to the policy module consists of a scaled one-hot vector that serves as the task indicator. As mentioned above, the low-level policy receives information about the task. However, the vector that carries the information has a single non-zero value. This needs to be taken considered when integrating it with the high-level policy.

Reusing the High-Level Policy

- **Training:** The toy agent will be trained to perform tasks using MRL, and the resulting policy will be reused and will serve as the high-level policy for the new agent.
- **Integration:** The hypothesis is that if the new agent can closely mimic the movement of the toy, the inference module will be reused. The high-level policy trained on the toy agent will produce actions. These actions can be applied to the motion equation (refer to Equation 5.13). The resulting state can then be taken as the input to the low-level policy, enabling the more complex agent to imitate the movements of the toy and therefore perform the different tasks effectively.
- **Interface between high- and low-level controller:** In the example of controlling the position and velocity, the state of the toy is two dimensional. However, as specified above, the low-level policy requires a scaled one-hot encoding, specifying the base task and the parametric variability. Thus, a simple classifier is trained based on the latent embeddings from the task inference encoder. These embeddings should contain enough information for the agent to distinguish which base task they represent. The classifier converts the state vector into a vector with a single non-zero value which can be used to guide the low-level actions.

Retraining the High-Level Policy on the Complex Agent

- **Setup and training:** For the high-level controller, a network is trained to produce an output that is consistent with the input dimensions of the low-level network. The input to the high-level network consists of the agent's state and the latent embeddings obtained from the inference module encoder. During training, the inference module is frozen, thus not updated. Figure 5.5 shows the model used during testing with the inference module trained on the toy. It also represents this approach of retraining the high-level policy when the network is updated after every iteration of trajectory sampling.
- **Network design:** The network is designed as a simple MLP which outputs a vector of the same size as the input vector for the low-level policy. To generate the necessary scaled one-hot vector, a max pooling layer is applied at the end of the network.

5.7 Complete Model with Inference Reutilization

After training, the different modules are tested together. The inference module is reused on a different agent on which it was trained on. If it is the case that the agent can perform the tasks effectively, the hypothesis of a robot-agnostic inference module is confirmed. The method works with the combination of five components:

- **Task inference encoder:** The encoder has been trained on the toy agent and will be applied to a new agent. The idea is to reutilize this component for different robots that fulfill the prerequisites listed above. The encoder requires, however, the input in the same observation space as the one it is trained on. This leads to the second component.
- **Observation mapping module:** NNs don't perform well when dealing with varying input sizes. Therefore, mapping the current state of the agent to the same observation space as the one used in training is essential. For the use case observed in this thesis, this is the equivalent to extracting the global parameters of the robot.
- **High-level policy:** As explained above, the high level policy can be trained in different ways. The high-level policy takes the latent embedding z and outputs an action a_{high} that serves as the input to the low level policy. A valid denomination for this action is also subgoal as it is the goal which the transfer function is trying to achieve.
- **Transfer function:** This is the module that enables the transfer of the inference module by bridging the action spaces of the toy agent and the new agent. Another expression used throughout the thesis is low-level policy module as it constructs an action that can be processed by the robot.
- **New environment:** Lastly, an environment with the new agent is needed. The agent takes the output of the transfer function and applies the low level action to the robot. After applying the action, the agent ends up in a new state s' and receives a reward r; information from which the encoder can infer the task after processing it with the observation mapping module. This step is the last of the cycle which begins with a new encoder input.

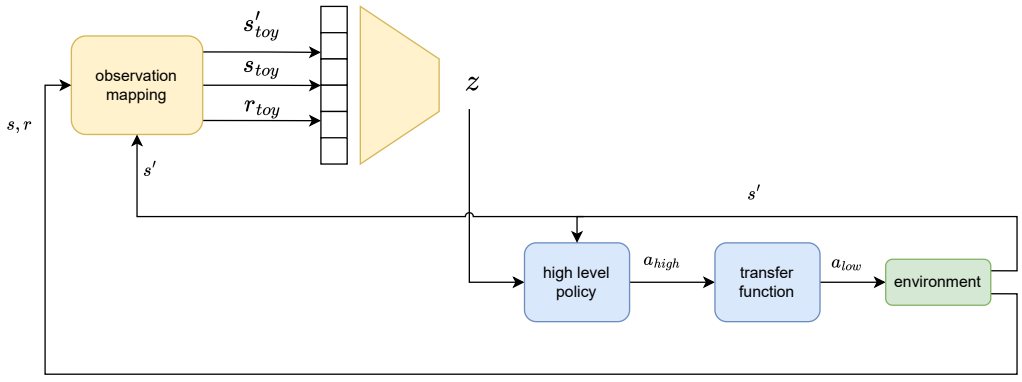


Figure 5.5: Architecture of the complete model during test time

When transferring the inference module to a different agent, some discrepancies have to be considered. One of those is the reaction times between input and output. For the toy agent, the actuation of the joints leads directly to a change in position and velocity. Contrary to

this, other agents where the joints are not directly coupled with the environment, it may take more than one input to obsserve the desired output. How to cope with this issue will be discussed in Section 6.3.3.

Chapter 6

Experiments

To proof the applicability of this method, different experiments will be showcased in this section. Starting off, the environments on which the method was tested will be presented, mentioning the tasks that the agents need to perform. Next, we will show the reutilization of the inference module for the toy environment in settings with different limitations. This will be followed by the reutilization of a simple inference module for parametric task variations. Lastly, a more complex inference module will be used to transfer it to different agents for non-parametric task variations. We will also compare different policy training methods.

6.1 Success Criteria

This section will clarify the criteria used for assessing whether the method was successful in transferring the knowledge from one agent to another. A valid criterion used in research concerning RL scenarios is the accumulated reward over time. In many ML applications, the loss over time serves as a measure to assess the performance of the agent as these two measures usually correlate. In RL scenarios, because of the use of the actor critic during training, the loss may not accurately reflect the performance of the agent. An increasing reward over time is more likely correlated with the agent learning to perform the desired actions. The reward function, however, may not always represent the whole truth. This is because the reward function is designed based on specific criteria and might be incomplete omitting important aspects of the training. Consider, for example, a scenario where the goal is to reach a specific position. One agent stands still. Another agent reaches the task, but overshoots and keeps moving in the same direction. On average the first agent might receive a higher reward , despite a clear lack of learning. The second behavior is more likely to be learning. However, this is not shown in the overall rewards. Thus, for the assessment of the performance, the reward history will be used in combination with the trajectories of the robot for a qualitative analysis when suitable.

Also, the goal of this thesis is not to reach a high degree of precision, but to show that the tasks can be recovered from a different dynamical configuration and that the new agent can perform the desired task in a zero-shot manner. This requires a high degree of generalizability which is what this chapter will focus on validating.

6.2 Training on simple agent

The goal of this thesis is to create a method that enables the transfer of an inference module between different agents. This can be done for two agents that share a common goal and have similar ways of computing the reward. In order to facilitate exploration, we make use of an agent that meets the requirements but also has very simple dynamics. This reduces the state space significantly, thereby reducing the amount of states that must be explored. The result is a toy environment which was introduced in [Dur23] and further adapted to tackle a wider variety of tasks including non-parametric task variations. The toy agent is an adaptation for different robots, which have been reduced to a sphere removing all dynamic complexities. We will be using different versions of the toy environment based on the tasks that need to be solved. These will be explained in the following.

For the first experiments discussed in Section 6.3.3, a very simple toy agent will be used. The focus lies on solving the task of reaching positions with parametric variability. Therefore, it suffices to simulate the agent by following this simple update step:

$$s' = s + a \quad \text{with} \quad a \in [-a_{max}, a_{max}] \tag{6.1}$$

where s and s' are the current and next positions and a is the action taken. The action a is bounded such that the other agents can mimic the movement of it as they cannot move arbitrary long steps in a single time step due to physical limitations.
Let's consider, for instance, that the toy is trained to reach different positions in the range $[-15, 15]$. Without a bound on a, the toy can reach the goal in very few steps. If we try to transfer the knowledge to a robot agent like the cheetah, it will struggle since it would take the new agent many steps to reach the desired position. This behavior is different from the training conditions for the inference module and will therefore encounter difficulties to infer the task in the new setting. Hence, it is important to bound the actions in Equation 6.1.

In Section 6.4, the task variation will expand to also validate the method for tasks moving at a certain velocity. Since position and velocity are correlated, it is not enough to simply set the next position and velocity as this would conflict with the laws of physics which other agents like the cheetah need to adhere to. Instead, it is necessary to employ motion equations that follow Newton's principles of motion. To take this into account, the dynamic equations were update respectively (Equation 5.13). The simulation now adheres to Newton's laws of motion with the incorporation of a friction term such that the velocity cannot increase arbitrarily. This is depicted in Figure 6.1. Again in this scenario, the toy should not be able to move at arbitrary long steps. Analog to the previous example, the action is bounded. However, the action now consists of an external force acting on the center of mass of the toy agent, not a movement.

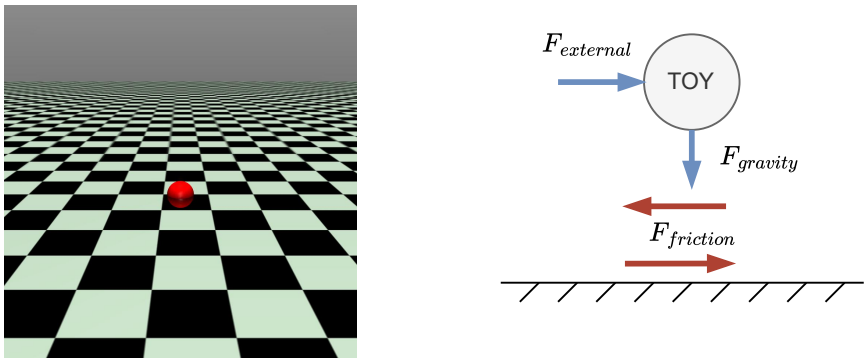
This model is extended when broader tasks are considered, like rotation around its own axis where the external force can act as a torque in y-direction.

The tasks are sampled from a uniform distribution in the ranges described in the Table B.2 unless stated otherwise. For training unless stated otherwise, tasks are sample randomly from the following distribution:

$$i \sim \text{Cat}(\pi_1, \pi_2, ..., \pi_n), \quad \pi_1 = \pi_2 = ... = \pi_n \tag{6.2}$$

$$\tau^i_{spec} = \mathcal{U}^i_{[s\_min, s\_max]} \tag{6.3}$$

- π_i: probability of choosing base task i



(a) Toy agent in mujoco environment **(b)** Abstracted figure with interaction forces

Figure 6.1: Depicted is the toy environment in the mujoco simulation (left) and an abstracted model of it with acting forces (right).

- $\mathcal{U}^{i}_{[s\_min,s\_max]_i}$: uniform distribution of base task i
- $[s\_min,s\_max]_i$: range of parametric variation of base task i.

For the agents with more complex dynamics, we chose already existing models from mujoco [TET12]. These are models consisting of links (body parts) linked by joints. They interact with the environment through contact forces with the floor. Figure B.1 serves as a visual overview of the different models, namely the cheetah, the hopper and the walker.

The configurations are also represented in Table B.2 in more detail. These different configurations are used due to their different dynamics and environment interactions. This helps show the applicability of the inference reusability method to a range of varying models.

6.3 Task Inference Reutilization for Parametric Task Variations

6.3.1 Inference Module Training in Toy Environment

The first experiments are meant to validate the methodology on a simple task with parametric variations. This does not require a complex inference module. Thus, in this section the inference module will consist of simple GRUs that extract information from trajectories. The tasks will vary in paramters, but the base task will remain the same throughout the experiments. The goal will be to reach a certain position τ^{i}_{spec} in the range $[0,10]$. The inference module and higher level policy will be trained on the toy agent mentioned above.

Figure 6.2 shows the average rewards over the training steps for the toy agent. As the high-level policy as well as the inference module will be reused from this training, it is important to see an improvement of performance over time as shown.

Figure 6.3 shows sample trajectories sampled with the trained networks in a MRL setting where the agent doesn't receive any information about the task besides the reward. This results will serve as the baseline for the rest of the experiments in this section. In the top left, the figure plots the trajectories over time against the desired position. The other two graphs show the latent variables that serve as input to the policy module. In this example,

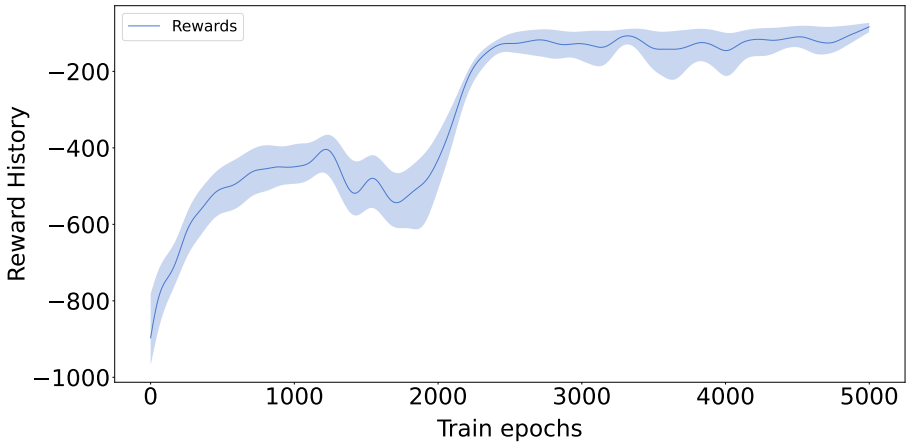


Figure 6.2: Reward history of toy training.

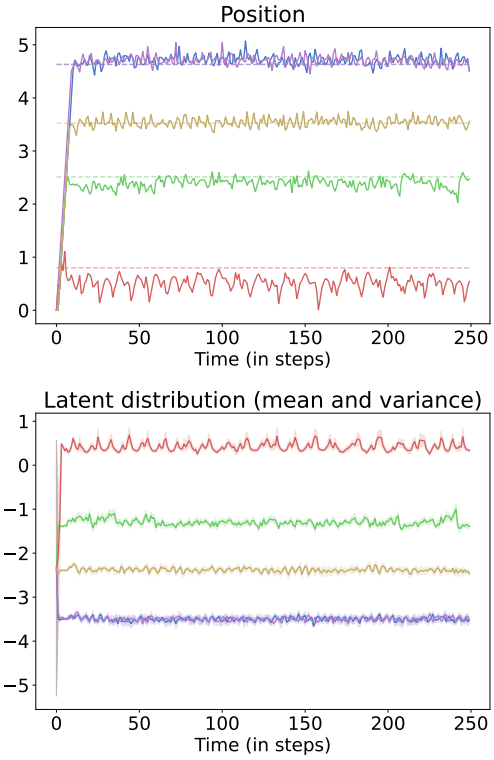
the encoder reduces the dimension of the MDP to a one-dimensional vector. Not only makes it easier to visualize, but it is also sufficient for task deduction. This hypothesis is backed by the right plot of Figure 6.3. The graph shows the relation between the desired position and the encoded latent variable. After the training is completed, the relation between encodings and true task is observed to be linear, indicating that the true task can be inferred from the agent's interactions with the environment.

6.3.2 Reutilization of Inference Modules Without Hierarchical Structure: A Non-Functional Approach

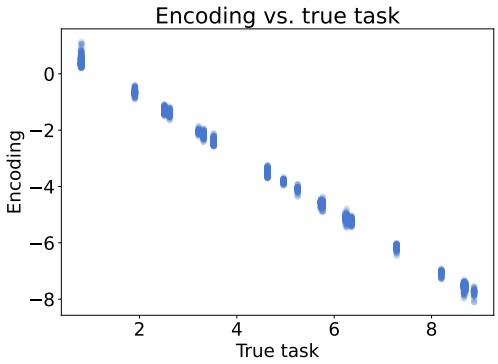
We begin by presenting an example that serves as the motivation for employing a hierarchical approach for the objective of reusing the inference module with a different agent. Lerch showed that inference reutilization works for a simple environment where the agent's dynamics are simply scaled, not completely replaced [Ler20]. In his thesis, Lerch trained an inference module on a simple toy agent and reused it on a similar toy with a scaled action by only retraining the policy module. The next logic step is to validate this approach for a completely different agent by retraining the policy module as illustrated in Fig 6.4. This, however, is not possible if the agent's dynamics surpass a certain complexity. Results of such an attempt are depicted in Figures 6.5 and 6.6.

The reason for this inability to transfer the inference module is due to the chicken-and-egg problem discussed in Section 4. This issue creates a cycle which can be summarized in three steps:

1. The agent cannot perform basic movements and therefore is unable to perform any type of significant exploration.
2. Without exploration, the inference module is unable to encode meaningful latent variables.
3. The latent encoding serves as input to the policy. Without any information about the task, the policy cannot guide the agent to a higher reward. If the agent has no meaningful information about the task, the reward is perceived as arbitrary by the policy and thus cannot make relevant policy updates.



(a) Trajectories of the agent (top) and latent embeddings over time (bottom).



(b) Latent embeddings over time.

Figure 6.3: Output of training for a simple toy environment on reaching positions. (Left) Trajectories of the agent (top) and latent embeddings (bottom). (Right) Latent embeddings over true tasks.

This is a vicious cycle from which the agent cannot learn.

6.3.3 Inference Module Reutilization Using a Hierarchical Policy

The idea behind employing a hierarchical structure is to enable the agent to explore the low-level state space effectively and thus break the aforementioned cycle. For the remaining of this thesis, we will be using a two-layer hierarchical policy consisting of a low- and a high-level policy. The low-level policy will command the robot at joint-level and the high-level policy will operate at the level of global variables.

Multi-Task Reinforcement Learning for the Low-Level Policy

In the first experiments, we will reuse the inference module as well as the policy trained on the toy agent. Hence, for every new agent tested, only the low level policies are needed additionally. The training of these follows the procedure described in Section 5.5 and 6.3.3. The reused policy will serve as the master policy which will command the low-level policy in the form of subgoals. This section offers an overview of the training of the low-level policy. As explained in more detail in Section 5.6, training the low-level control of the agents, for which the inference module is to be reused, is performed separately. During this training phase, the agent learns to reach a position sampled randomly from a distribution similar to the one used to train the inference module. However, the agent must not only be able to reach a desired position from the zero point.

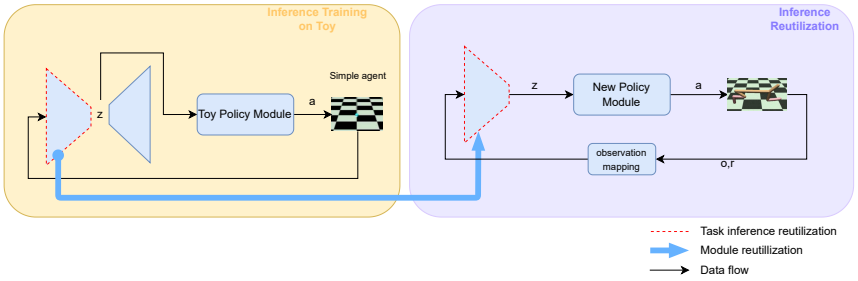


Figure 6.4: Aproach to inference reutilization by training the policy module on the new agent with the reused inference module. No hierarchical structure for the policy was used here.

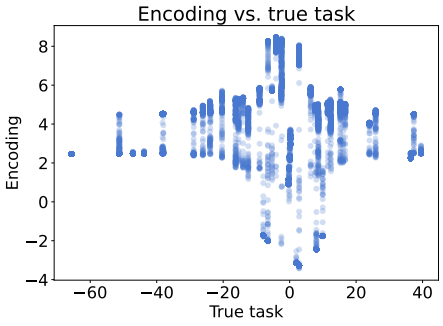
The higher level control, which is reused from training the toy example, updates its output after every step. Given that exploration is an important factor in MRL for task deduction, it may at first command the agent to move in the wrong direction followed by the opposite one. Therefore, the low level control must be able to recover and reach targets from different states.
To achieve this, a straightforward, yet effective training strategy is employed. Instead of randomizing the initial configuration, the agent is allowed to perform longer trajectories. During these trajectories, the task is altered after a certain amount of steps. This method simulates random initialization of the agent ensuring an agent capable of following the directions of the high level controller. This NN is also referred to as the transfer function since it acts as the intermediary function that bridges the high-level controller and the actions of the agent, considering the system's dynamics.

Naive Approach to the Inference Module Reutilization

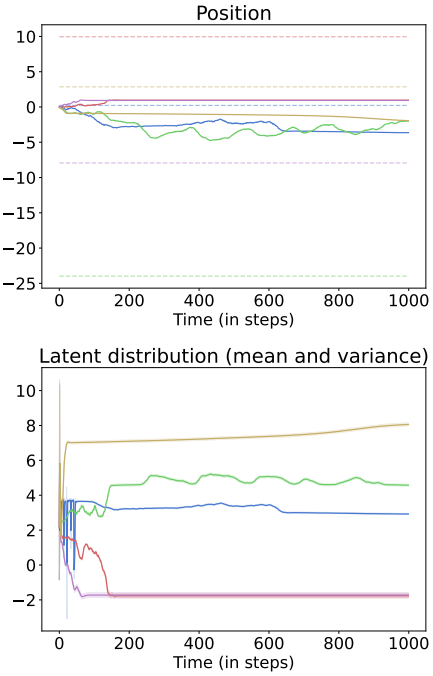
After all components are trained, they have to be combined and it needs to be tested whether the inference module trained on a highly simplified agent can be reused. The goal of this section is, therefore, to show that an inference reutilization is possible and how it behaves for different agents in a simple scenario. These results show that when the more complex agent can move similar to the toy agent, the task inference as well as the policy module can be transferred successfully.

Figures 6.2 and 6.7 show the learning process of the high and low-level policies separately. It can be said that both modules show a good performance in separate. The question that is left to answer is whether the proposed hypothesis is valid and thus the method works. The hypothesis: The inference module is only dependent on high-level features and can thus be reutilized for different agents. This can be achieved by employing a hierarchical structure on the policies to bridge exploration gap between different agents.
For the case that this hypothesis holds, the inference module should be reusable for different agents with the aid of different low-level policies.

Figure 6.8 shows the results obtained from reusing the inference module for the half cheetah. The plots show a noisy linear relation between the encoded latent variable and the target position. Although the cheetah seems to reach the desired position with reasonable accuracy, the plots showcasing the latent embeddings, show a high amount of noise compared to the baseline (Figure 6.3). This results in an oscillatory behavior of the agent around the desired



(a) Latent variables for different trajectories during an attempt to transfer the inference module without using a hierarchical approach.



(b) Position of the agent over time for different goals during an attempt to transfer the inference module without using a hierarchical approach.

Figure 6.5: Results from attempting to train the cheetah while reusing the inference module without a hierarchical approach. The cheetah is not able to perform the desired tasks.

position. The reason for this issue is the discrepancy between the movement of the toy and the cheetah. One step of the toy can vary in the range of [-0.1,0.1]. The cheetah, however, moves by actuating the joints which move the links of the body that interact with the ground of the simulated world. This not only makes the movement more complex but also introduces certain latency between the desired input (subgoal from the high-level policy) and the actual output (movement).

Inference Module Reutilization With Striding

This problem can be resolved by letting the agent move for a certain number of steps before updating the high level action and the encoder. This technique will be referred to as striding. Figure 6.9 show the scenario where the high-level action is only updated every 10 (Figure 6.9a), 50 (Figure 6.9b) and 100 (Figure 6.9c) steps respectively.

The process is as follows: The agent receives an action from the high-level controller, which serves as input to the low-level controller. The low-level controller then attempts to reach the frozen subgoal within 10, 50, or 100 steps, respectively. Following this, the encoder is updated and returns a latent representation that serves as input for a new high-level action.

All plots show an notable improvement in performance compared to the scenario shown in Figure 6.8. Increasing the intermediate steps from 1 to 10 considerably improves the transfer. However, this improvement plateaus when increasing the steps to 50 and 100. This indicates

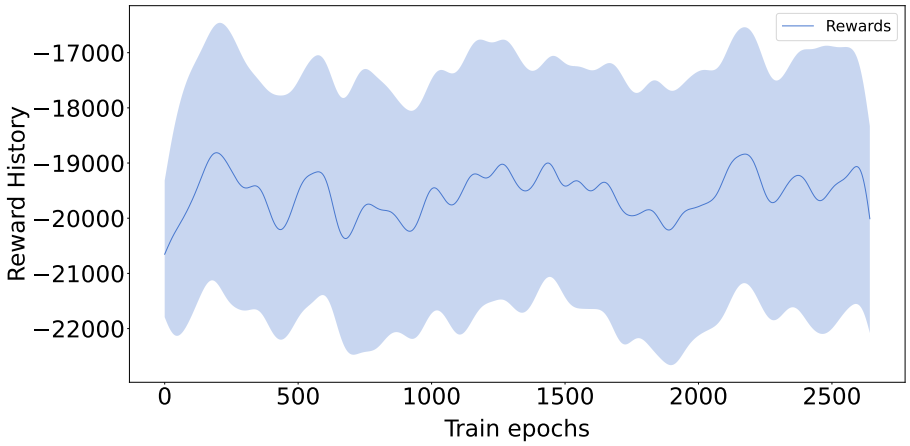


Figure 6.6: Rewards during policy training in an attempt to transfer the inference module to the cheetah agent without using a hierarchical approach. This approach is unable to learn.

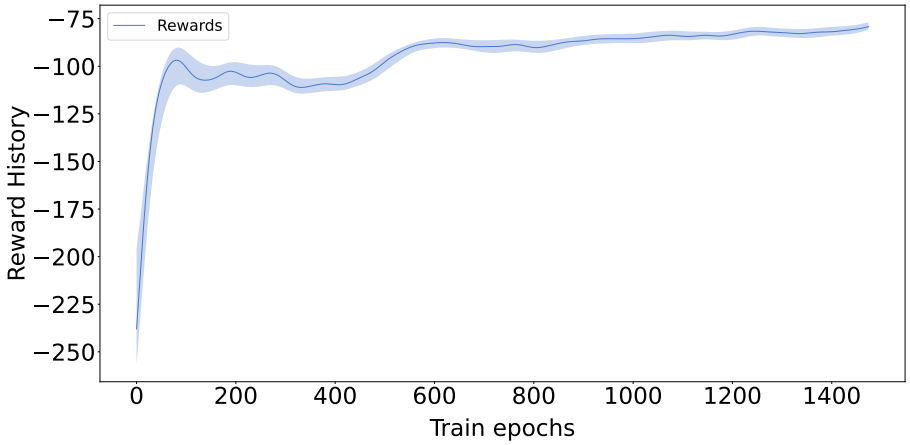
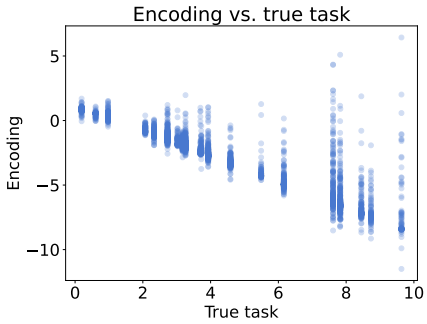


Figure 6.7: Rewards during training of the low-level policy for the cheetah.

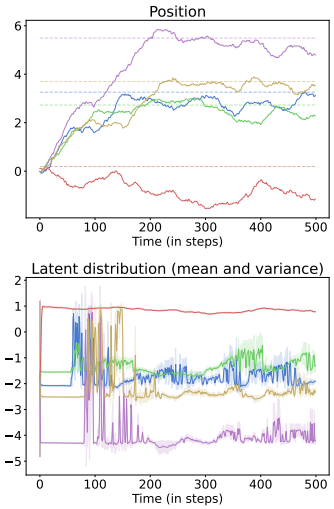
that once the latency problem gets minimized, the performance is limited by the low-level controller. The agent can only reach a precision as good as the precision of the low-level controller.

In order to show the validity of the model for different agents and that the training of the toy is not overfitted for the cheetah agent, the transfer is also performed for the hopper and the walker. A different policy that serves as the low-level controller is used for each agent. On the high-level controller and the inference module, which are solely trained on the toy environment, there are no adjustments made.

Figures 6.10 and 6.11 show the validity of the method for two additional agents, namely the hopper and walker. The striding used is 5 but every number higher than this would also be valid. A higher striding means generally a slower convergence to the desired goal. As shown in these figures, the method reaches a good precision with all agents.



(a) Latent variables for different trajectories



(b) Position of the agent over time for different goals (top) and latent variables over time (bottom)

Figure 6.8: Results of naive attempt to reuse the inference module on the cheetah.

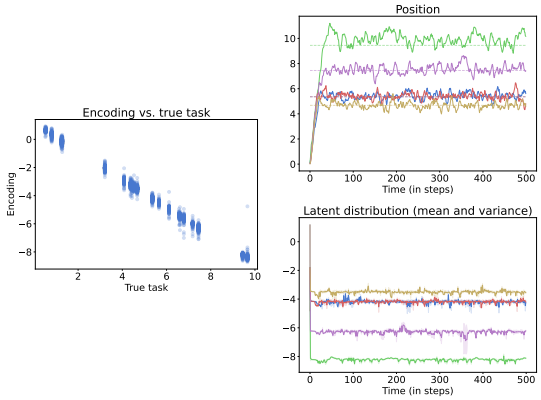
6.4 Task Inference Reutilization for Non-Parametric Task Variations

The next step is to verify the validity of this method for non-parametric task variations. The inference module used in the previous step is not able to capture non-parametric task variations due to the simple nature of it. Thus, in this section, a more refined version of the inference module will be employed. The inference module will consist of a similar encoder as the above. Additionally, the addition of a DPMM module will guide the encoder to cluster different tasks based on the trajectories. In this section, the tasks observed will be to reach a goal in the back, in the front as well as run at a certain speed backwards and forwards. The toy agent will be trained to perform the mentioned tasks in a MRL setting. Again here, a different low-level controller will be trained for each agent which will serve as the bridge between the toy dynamics and the new agent's dynamics. Unless stated otherwise, both the inference module and the high-level policy will be reused.

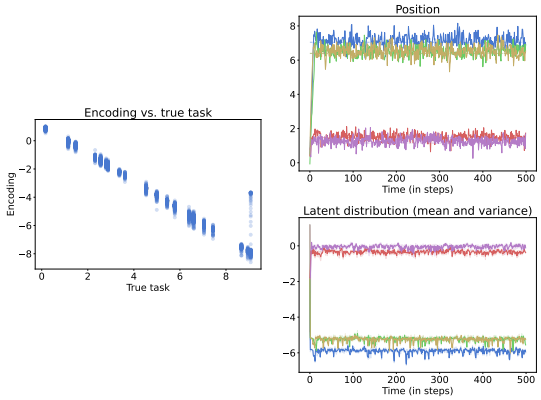
As mentioned in Section 5.6.2, the interface between high- and low-level policy is an important factor to consider. Contrary to the first experiments discussed, the interface is not one dimensional. Here, the agent must be able to perform fundamentally different tasks. Hence, the low-level policy must receive higher-level inputs. The method now works as follows:

- The high-level policy produces an action
- The action is processed by equation 5.13. This is the same as simulating the toy which ends up in a new state.
- The low-level policy receives the state of the toy as input to mimic its movement. Since the toy has been trained to perform the tasks, if the movement of the toy is mimicked, also the new agent will consequently complete the task.

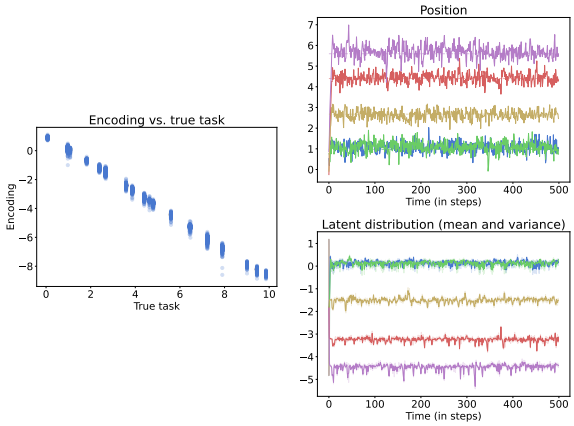
However, since the algorithm requires striding, following the positional trajectory does not imply reaching the same velocity. Therefore, the agent has to be able to chose whether to follow the positional trajectory or velocity. For this purpose, a different NN is trained. This



(a) Latent embeddings and position of cheetah for 10 intermediary steps



(b) Latent embeddings and position of cheetah for 50 intermediary steps



(c) Latent embeddings and position of cheetah for 100 intermediary steps

Figure 6.9: Results for the inference module reutilization for different striding factors.

is treated as a classification problem. The input is the encoded trajectory and the output is a one hot encoding that decides between position or velocity. It is trained with the binary cross entropy loss between the output of the NN and the real base task. When the one-hot encoding is combined with the output of Equation 5.13, the result is a subgoal which serves as input to the low-level controller. Together, they form the hierarchically-structured policy that guides the agent towards the goal.
This hierarchical approach has one shortcoming. Since there is one high-level policy that guides the low-level policy, the subgoals obtained from the high-level policy change every few steps. This poses a challenge when training the low-level policy since the agents must be able to change goals frequently while maintaining the desired stability. The frequent changes intensify the problem of conflicting goals explained in section 5.6.

6.4.1 Curriculum Learning for Challenging Multi-Task Settings

To solve this problem we will explore a simple curriculum learning as an addition to the procedure described in Section 6.3.3. The agents that suffer from said instabilities will be taken as examples. As far as the curriculum is concerned, a simple recipe will be used. The

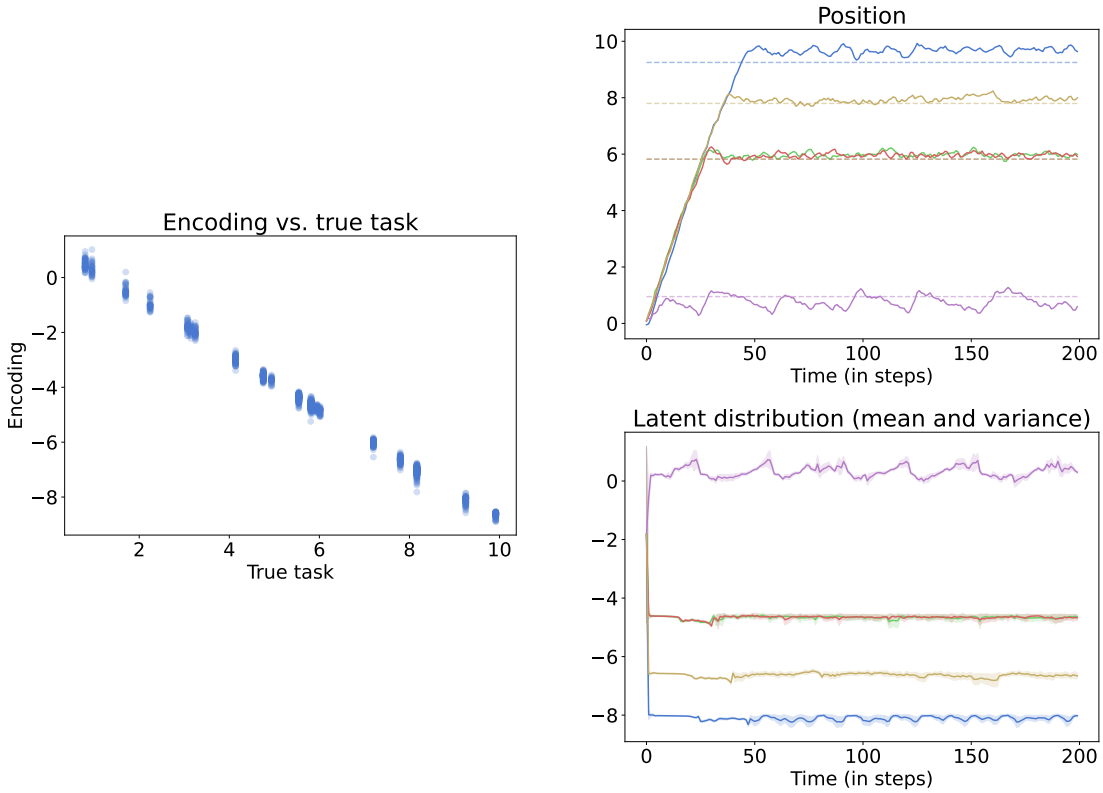


Figure 6.10: Results of reusing the inference module trained on the toy agent with the hopper. Latent embeddings and position for hopper with striding 5.

idea is to start simple and gradually increase the complexity of the desired behavior. The parameters that will be altered during the training are how often the task change and the trajectory length. At the beginning, the maximum trajectory length is 300 steps and no task changes are undertaken during a trajectories. In the end, the maximum trajectory length is 1000 steps and the agent changes the task every 125 steps. Additionally, if the agent reaches the goal of reaching a certain position or moving at a certain speed with a deviation of $0.1m$ or $0.1m/s$ the task changes with a probability of 0.2. Thus, the agent doesn't remain stationary for too long and can explore more states.

Additionally, for agents suffering the aforementioned instabilities, the reward function must be modified to include a term that rewards stable states. Thus, the reward is similar to the cheetah agent with the addition of a scalar every step it maintains balance. As a result the reward histories result in higher values than the one from the cheetah training.

The results are shown in Figure 6.12. The average rewards are displayed over time, along the steps performed without entering an unstable state as this is an important measure to consider since the method cannot work in unstable states.

Since both agents were trained with the same reward and curriculum it is possible to compare the reward histories. This comparison reveals two interesting observations.
First, the walker has more difficulties in the beginning of the training. It is not until the epoch around 1100 when the policy starts to show significant improvements as shown by the steep increase in reward. For the hopper, however, the increase can be seen around the epoch 500,

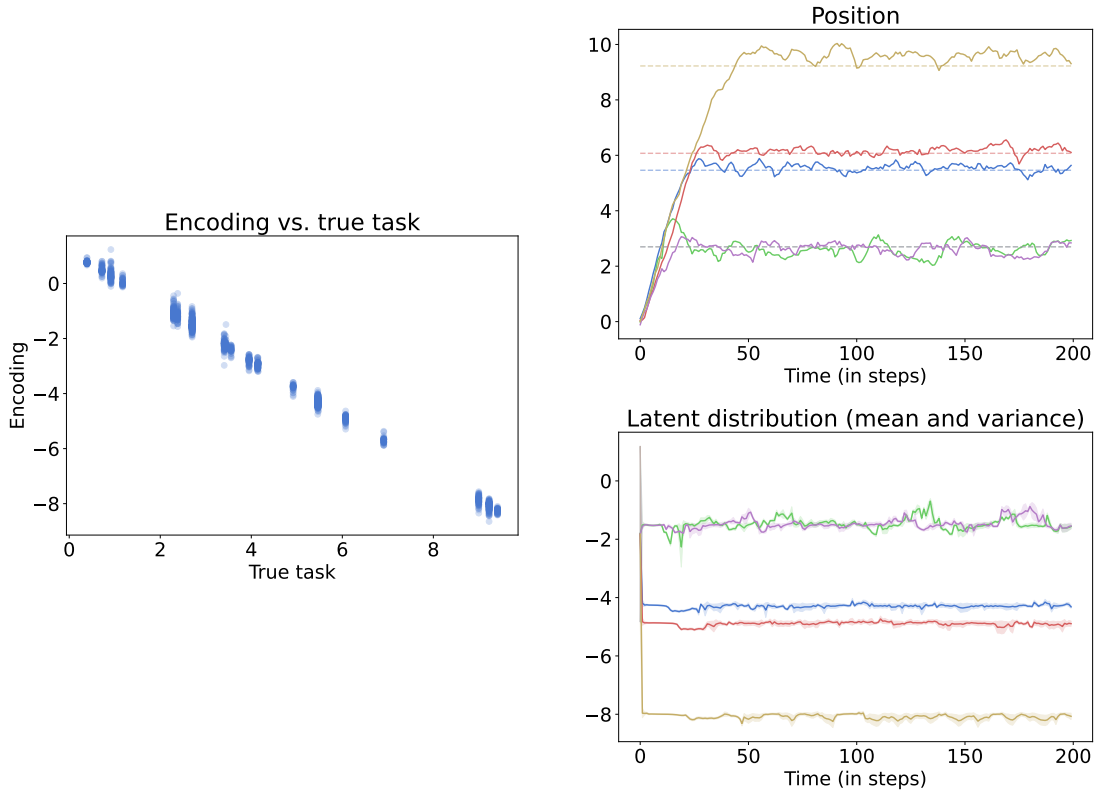


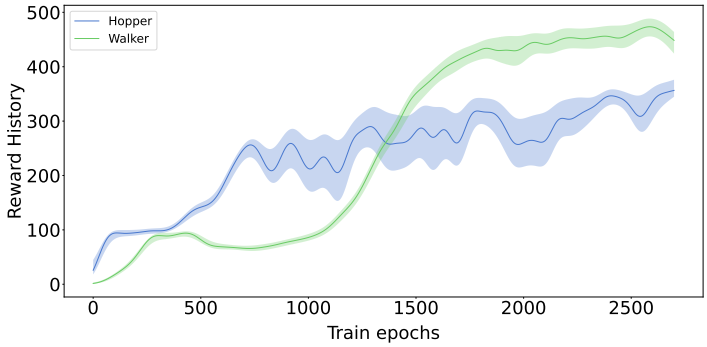
Figure 6.11: Results of reusing the inference module trained on the toy agent with the hopper. Latent embeddings and position for walker with striding 5

a significant difference. A possible explanation for this phenomenon is the higher degrees of freedom from the walker which makes it more challenging to control. This requires more exploration and therefore, more training steps.
Secondly, it can be observed that asymptotically, the walker offers a better performance as seen by the average rewards as well as the average trajectory lengths. Once the walker overcomes the challenging exploration phase, it reaches rewards that on average are around 100 points higher. The same can be observed for the average trajectory lengths. A potential interpretation is that the walker has two independent contact points with the environment; the hopper only possesses one. This makes the walker more flexible and allows a better control.

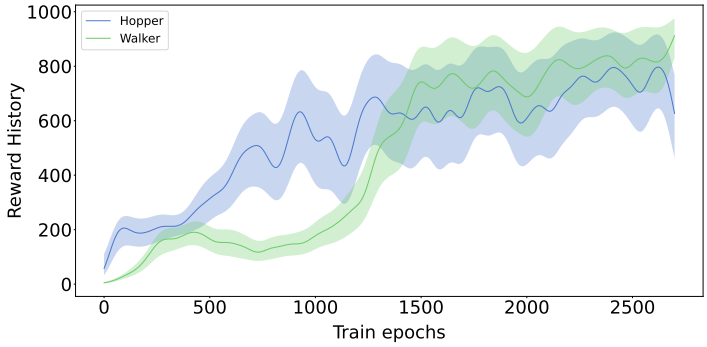
6.4.2 Training the Inference Module on the Toy Agent for Parametric Task Variations

With the policies obtained from training the different agents with the above mentioned algorithms to perform what during test time will be the subgoals, an attempt on reutilizing the inference module was done for non-parametric task variations. The inference module was again trained on the simple agent environment. This time, the toy was trained on the tasks of moving at different speeds and reaching different positions.

Again, for this new inference module, the rewards over time are an important factor to consider as the high-level controller will be reused. Therefore, if the high-level controller cannot guide the agent towards high rewards, the complete model will fail. As shown in Figure 6.13 the agent learns over time to perform the different tasks (Figure 6.14). Apart



(a) Reward history for the training of thelow-level policy of the hopper in the multi-task setting with curriculum learning.



(b) Trajectory length history for the training of the low-level policy of the hopper and walker in the multi-task setting with curriculum learning.

Figure 6.12: Plots showing the evolution of the hopper's and walker's performance during training of the low-level policy.

from that, Figure 6.14b shows the projection of the latent embeddings of multiple trajectories onto two dimensions. From the different clusters, it can be deducted that the VAE along with the DPMM successfully manages to cluster the different tasks which makes it possible for the agent to learn to perform those tasks.

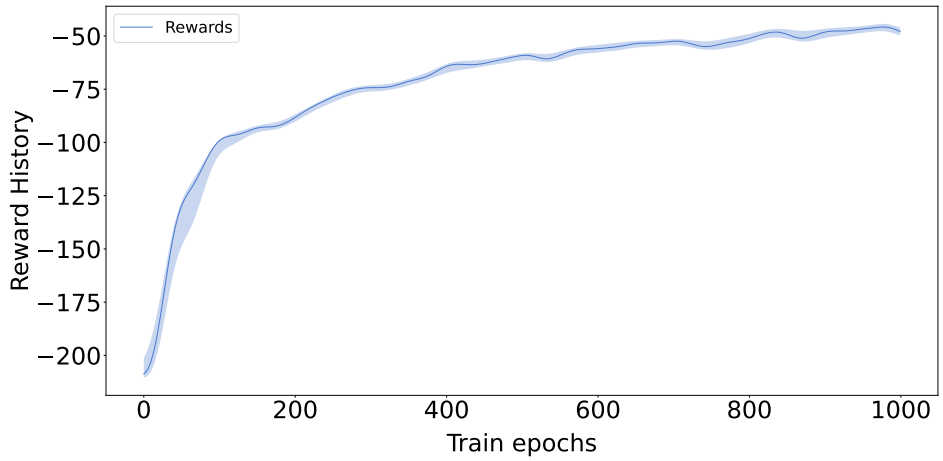
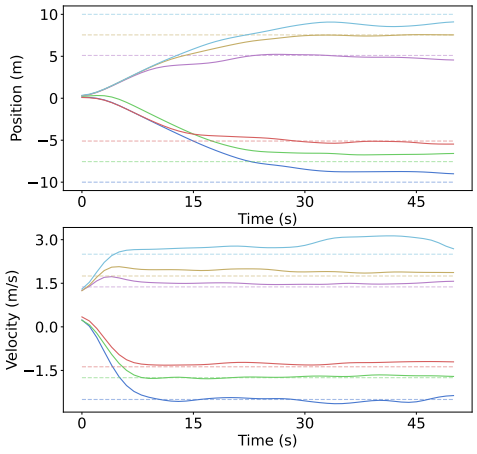
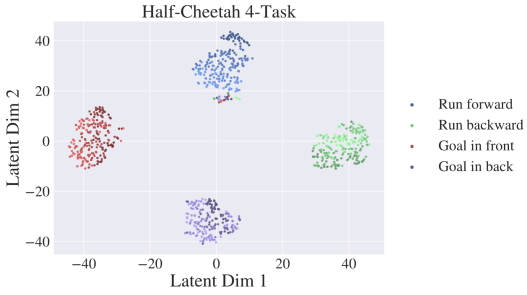


Figure 6.13: Reward history for the training of the toy agent on 4 different tasks: reach goal front/back, move at a certain speed forward/backward.



(a) Multiple position and velocity trajectories of the toy agent plotted against the desired position.



(b) t-SNE plot: Latent embeddings projected onto two dimensions with the t-SNE technique for multiple runs.

Figure 6.14: Multiple trajectories for the toy agent doing one of the following tasks: run forward, run backward, reach goal in the front, reach goal in the back.

6.4.3 Inference Module Reutilization

A similar procedure as in 6.3.3 will be followed with three distinctions. First, as introduced above, a classifier will be trained to distinguish between base tasks. Based on the latent embeddings, it predicts the base task at hand and outputs a one-hot vector. In combination with the output of the high-level policy, it will form the subgoal that serves as the input to the low-level policy.

The second difference comes from the updated toy agent. To consider velocity tasks, the agent is slightly modified and is described by Equation 5.13.
Additionally, a model is trained to tackle the problem of striding. In the examples discussed before, this issue can be easily solved by introducing a striding factor. However, the latency varies for different base tasks and different states. This makes it very difficult to fine-tune the striding parameter. When introducing new, non-parametric task variations, this leads to problems and the agent does not reach the desired performance. Thus, in order to solve different tasks that vary non-parametrically, we introduce the step predictor model. For each subgoal imposed by the high-level policy, it predicts how many steps it will take for the complex agent to reach the subgoal from the current state.
This new model introduced as the step predictor model, will be trained similar to the policy modules with a SAC architecture where the reward is the negative distance to the subgoal.

The results presented in Figure 6.15b show the performance of the proposed approach. In comparison to the toy example, the performance is clearly degraded. The hypothesis to explain this behavior is the substantial difference in movement between the cheetah and the toy. In order to validate this hypothesis and improve the performance of the model, more experiments are conducted by introducing randomness during the inference module training with the toy. The resulting equation for the toy simulation becomes:

$$\mathbf{x}' = \mathbf{x} + \mathbf{v} \cdot \Delta t + \mathcal{N}(0, \sigma^2) \tag{6.4}$$

$$\mathbf{v}' = \mathbf{v} + \mathbf{a} \cdot \Delta t + \mathcal{N}(0, \sigma^2) \tag{6.5}$$

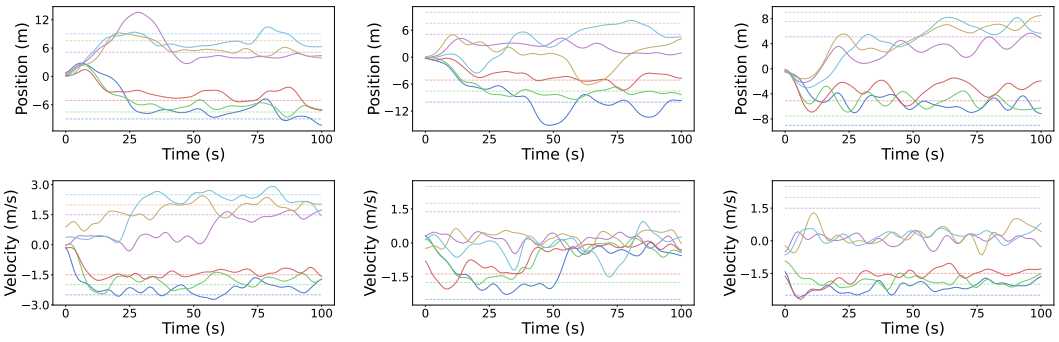
$$a_x = \frac{f_x - c_x \cdot v_x}{m}, \quad a_y = \frac{f_y - c_y \cdot v_y}{m}, \quad a_z = \frac{f_z - 9.81}{m} \tag{6.6}$$

where the action is represented by $\mathbf{f} = \left(f_x, f_y, f_z\right)^T$ and $\mathbf{x}$, $\mathbf{v}$ and $\mathbf{a}$ represent vectors containing all three dimensions. The randomization terms are only added during the training of the inference module on the toy.

Applying this randomization strategy, the performance improved significantly as shown in Figure 6.16. A comparison for different randomization strategies is shown in Figure 6.15. The results show that randomizing the simulation can be very beneficial for learning a more general inference module. However, the variance used is a parameter that has to be carefully chosen as too much variance can also hinder learning.

This procedure is again tested on the same three different agents. The results clearly show that the inference module can be reused for different agents as the desired trajectories are closely followed by the agents.

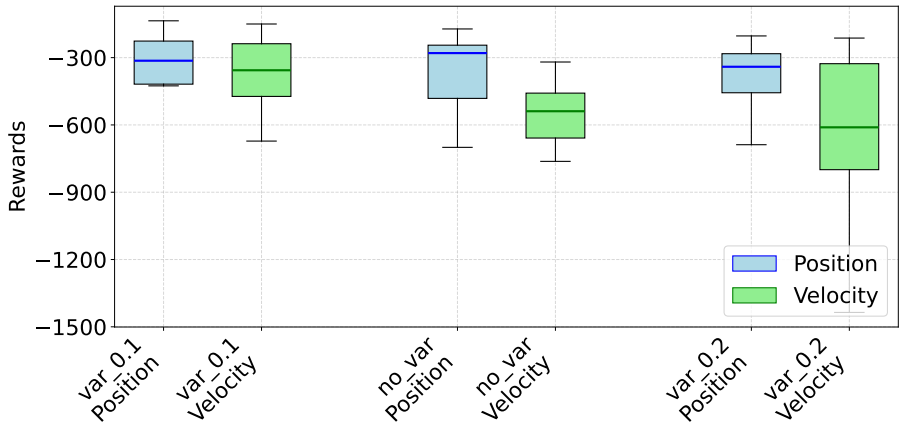
As mentioned above, the accuracy of the method is bounded by the accuracy of the low-level controller. In the case of the hopper, the complexity of the configuration makes it complicated to precisely move the robot like the toy agent. This makes the accuracy decrease. In Figures 6.17a and 6.18a, a horizontal line in the velocity plot likely indicates a fall from the agent from which it can not recover.
Additionally, the hopper doesn't seem to reach a velocity outside of the range of $[-1.3\text{m/s}, 2.3\text{m/s}]$ consistently. The asymmetric construction of the robot can be observed in the asymmetric boundaries of the maximum velocities. Nevertheless, the agent correctly recognizes the base task in most cases.



(a) The observations were randomized with a variance of 0.1.

(b) No randomization was used

(c) The observations were randomized with a variance of 0.2.



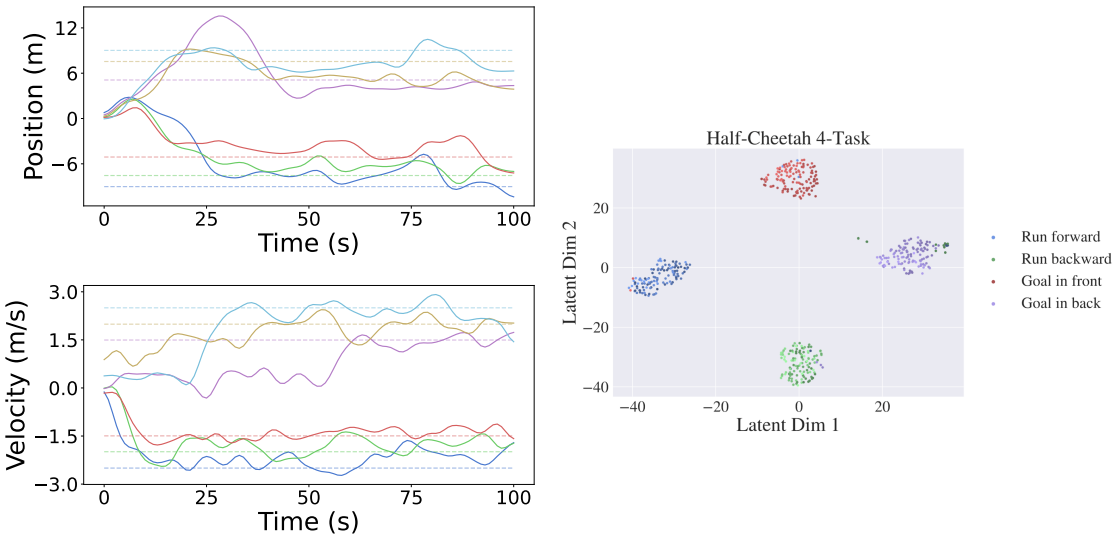
(d) Box plot showing the performance comparing different randomization strategies during training.

Figure 6.15: Multiple trajectories for the cheetah agent doing one of the following tasks: run forward, run backward, reach goal in the front, reach goal in the back. The inference module was trained on the toy agent and reused for the cheetah. The high-level policy was reused from the toy training as well.

6.5 Alternative Approach: Learning the High-Level Policy During Knowledge Transfer

Until now, the policy consisted of two hierarchies for which one was trained on the toy and the other on the agent to be used during testing. This section will explore the possibility of training the high-level policy as well on the complex agent. In comparison to the method used until now, this approach offers some benefits as well as drawbacks which will be discussed after presenting the results.

After the training on the toy has finished, the inference module is reused and an additional training step is performed during which two models will be trained. The high-level policy that guides the low-level policy and the step predictor, as this has shown good results for solving the striding problem. After every epoch, both models will be updated based on the trajectories stored in the buffer.



(a) Multiple position and velocity trajectories of the cheetah agent plotted against the desired position utilizing the inference module trained on the toy agent.

(b) t-SNE plot: Latent embeddings projected onto two dimensions with the t-SNE technique for multiple runs.

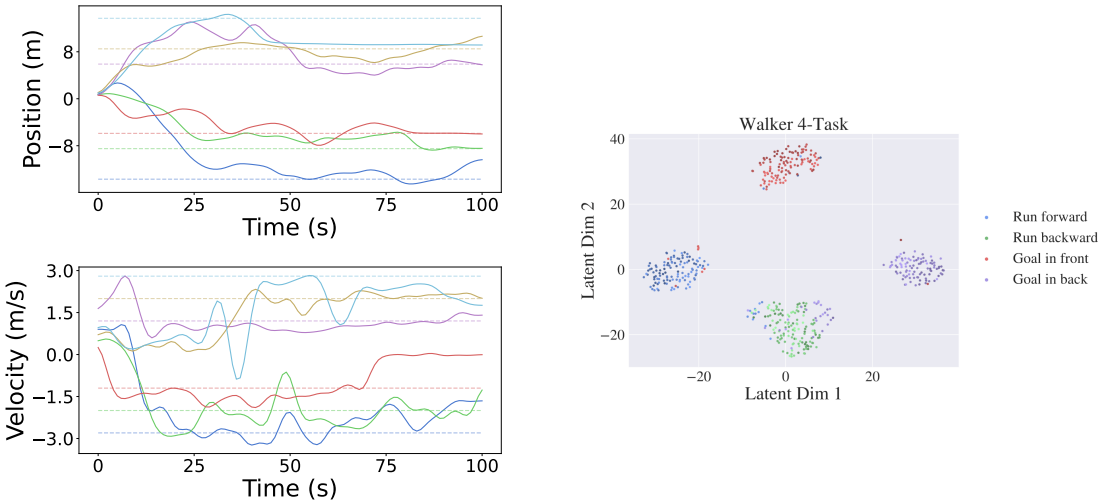
Figure 6.16: Multiple trajectories for the cheetah agent doing one of the following tasks: run forward, run backward, reach goal in the front, reach goal in the back. The inference module was trained on the toy agent and reused for the cheetah.

6.5.1 Relearning the High-Level Policy for Position and Velocity Goals

In order to compare this approach to the one employed in Section 6.4 the tasks remain the same: reaching a goal in the front/back and running at a certain speed forward/backwards. Additionally, the same inference modules as used in the previous section will be reused. From training of the two models (high-level policy and step predictor), two different reward histories emerge (Figure 6.19).
Figure 6.19a presents the reward histories for three distinct inference modules, each trained with different randomization strategies. Unlike with the approach described in section 6.4.3, there is no significant difference in performance between different randomization strategies which is also shown in the results depicted in Figure 6.20c. This indicates that the inference module can generalize without the need of randomization during training. On the contrary, the policy module trained on the toy agent exhibits more difficulties with domain shifts.

The resulting trajectories are shown in Figure 6.20c. These show the capability of the agent to distinguish between tasks and identify the parametric variability with an encoder trained in a very simplified setting (toy agent).

The same procedure was tested again with the walker and hopper to prove that also this approach is valid for different configurations. Again, the complex control of these agents degrades the overall performance during these tasks. However, it is visible that the agents can distinguish between tasks and closely follow the desired trajectories. The results are shown in Figure 6.21. Both plots show the agents facing problems with instabilities during some trajectories.



(a) Multiple trajectories of the walker agent plotted against the goal utilizing the inference module trained on the toy agent.

(b) t-SNE plot: Latent embeddings projected onto two dimensions with the t-SNE technique for multiple runs.

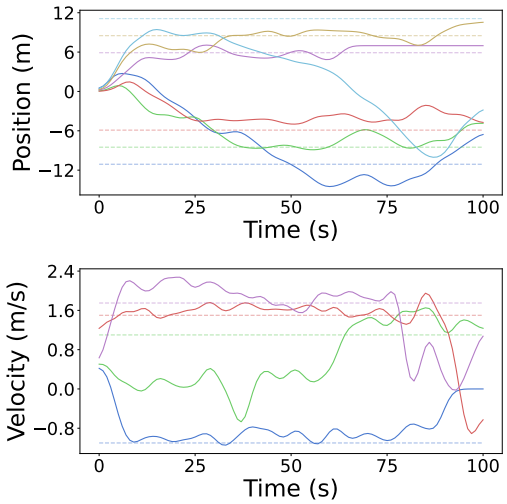
Figure 6.17: Multiple trajectories for the walker agent doing one of the following tasks: run forward, run backward, reach goal in the front, reach goal in the back. The inference module and the high-level policy were trained on the toy agent and reused for the walker.

6.5.2 Additional non-parametric tasks

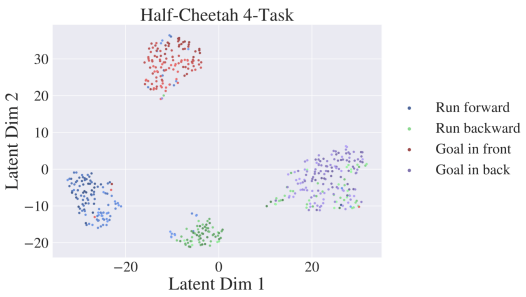
Finding a structure for the high level policy that can be trained on the toy and reused on another agent is straightforward for the case of reaching a goal. It becomes slightly more complicated when also the velocity tasks is taken into account. Adding more tasks requires increasingly more expert knowledge to find a representation that can be used for the toy and the complex agent. However, retraining the high-level controller, removes this step as it can be constructed according to the new agent. The only prerequisite is that it takes as input the current state and outputs a subgoal according to the input of the low-level controller.
Finding a representation of the toy agent on which to train the inference module with additional tasks is easier in comparison. That is why the approach of retraining the high-level controller is more flexible than reusing the policy trained on the simple agent.
For the following experiment, the toy was also able tested on tasks that required it to move its orientation around the y-axis. In other words, stand on the front or back legs. This shows that the approach is also valid for tasks that completely differ in the reward structure.

To test this approach, the cheetah agent was trained to perform the tasks from previous sections in addition to standing at an angle. The results are shown in Figure 6.22. It is clearly visible that the agent can also learn a high-level policy when the inference module is reused from a different agent as the agent is able to approximately follow the trajectories.

More tasks can be added to the training. However, the toy training has to be updated in accordance.



(a) Multiple trajectories of the hopper agent plotted against the goal utilizing the inference module trained on the toy agent.



(b) t-SNE plot: Latent embeddings projected onto two dimensions with the t-SNE technique for multiple runs.

Figure 6.18: Multiple trajectories for the hopper agent doing one of the following tasks: run forward, run backward, reach goal in the front, reach goal in the back. The inference module and the high-level policy were trained on the toy agent and reused for the hopper.

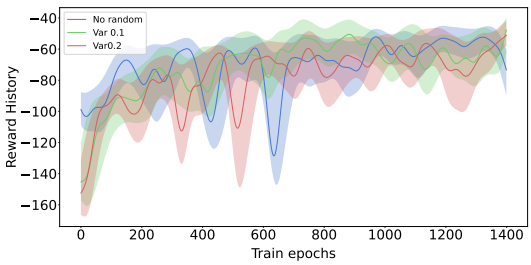
6.5.3 Final Thoughts on the Experiments

Summarized, the experiments have shown that training the high-level policy in both described manners deliver the wanted results. Each method have its own benefits and drawbacks.

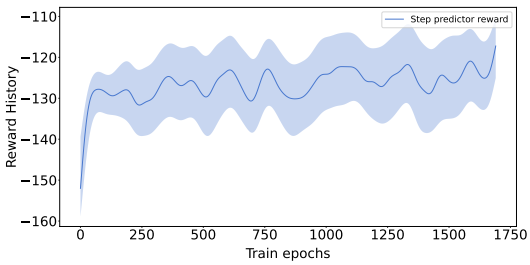
Reusing high-level policy: This approach is more data-efficient as it reutilizes the high-level policy trained on the toy agent.

Retraining high-level policy: As seen in 6.20c, the approach performs well even without randomization during training. Therefore, the additional data makes the training more flexible.

Also, this approach does not require the training of a base task classifier making it more suitable for continual learning.

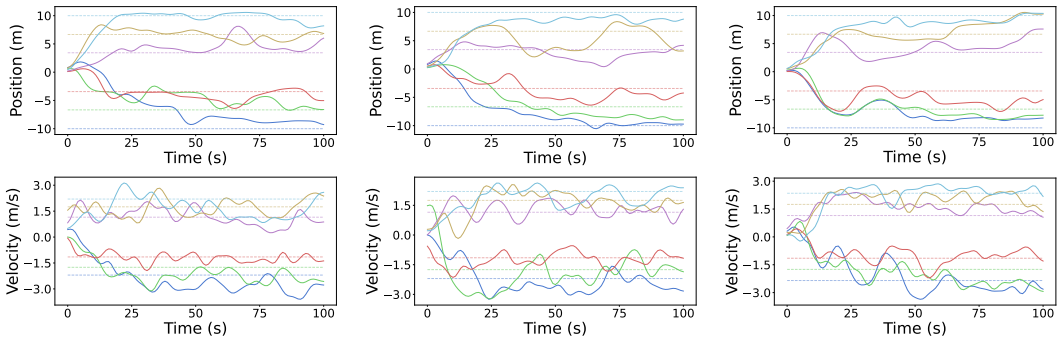


(a) Comparison of reward histories during training of the high-level policy for different randomization strategies during the inference module training.



(b) Reward history for the network used for predicting the striding.

Figure 6.19: Reward histories for the method of retraining the high-level policy while learning the striding parameter.

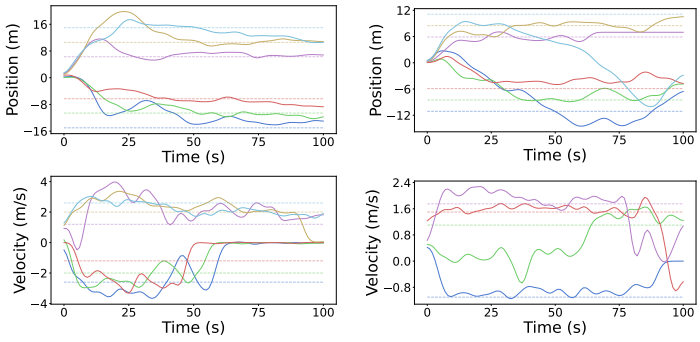


(a) The observations were randomized with a variance of 0.1.

(b) No randomization was used

(c) The observations were randomized with a variance of 0.2.

Figure 6.20: Multiple trajectories for the cheetah agent doing one of the following tasks: run forward, run backward, reach goal in the front, reach goal in the back. The inference module was trained on the toy agent and reused for the cheetah. The high-level policy was trained from scratch using the static inference module. Different randomization strategies were used during the inference module training.



(a) Walker agent

(b) Hopper agent

Figure 6.21: Multiple trajectories for the walker and hopper agents doing one of the following tasks: run forward, run backward, reach goal in the front, reach goal in the back. The inference module was trained on the toy agent and reused for the cheetah. The high-level policy was trained from scratch using the static inference module. Different randomization strategies were used during the inference module training.

Both methods can be used depending on the application. But most importantly, the experiments have shown that the inference module is not dependent on the robot and the same can be used on a variety of robots. We have introduced a method to reuse this inference module for different agents which can pave the way for future research and further the development of meta-reinforcement learning for complex agents.

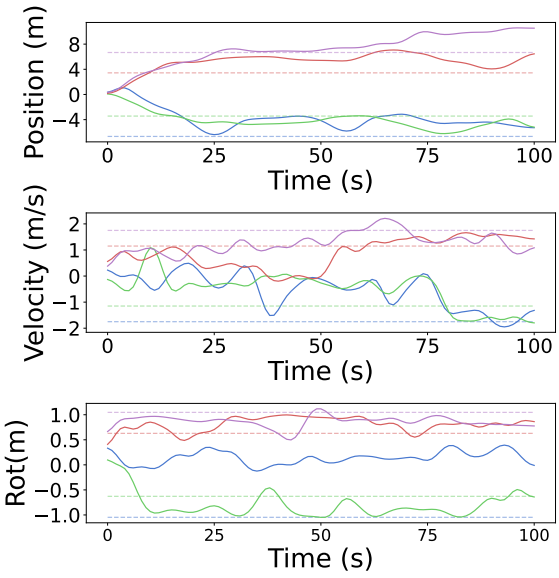


Figure 6.22: Trajectories of the cheetah plotted over time representing following variables: position, velocity in x, orientation in y.

Chapter 7

Discussion and Future Work

The goal of this work was to verify the hypothesis that the inference module can be reused by different agents and create a methodology which accomplishes this. This had been tried before in [Dur23]. The approach shown there, was successfully applied to transfer between agents with similar dynamics. Testing it on agents with notable differences in dynamics showed no learning or any signs of improving performance with training. Some experiments revealed that the problem was the chicken-egg problem mentioned in Section 4.

By employing a hierarchical structure, we managed to bridge the differences between the dynamics of distinct agents and thus solve the problem. In Section 6 two methods were described to reuse the inference module trained on a different agent. Each method offers its benefits and disadvantages. Although both methods showed promising results, this is not a claim that these methods are perfected. They serve as a possible solution but most importantly as a basis for future research in this field which offers significant benefits like the possibility to train inference modules in simple agents that offer a higher exploration with a highly reduced state space. It can also be viewed as transferring the inference module to a different domain thus opening the possibility to train MRL models in simulation and transferring the knowledge to the real world.

7.1 Future Work

As this is not a claim of a perfect algorithm, improvements can be made. Various aspects can be enhanced to improve the overall performance like improving the multi-task training by incorporating other algorithms better suited for this task than the one employed, namely SAC. However, this are small details that will not greatly affect the overall behavior. Consequently, this section focus on two different directions which can have a greater impact.

1. **Improve Hierarchical Structure:**
   For the methodology described in this thesis, a simple hierarchical structure was used to facilitate exploration. The high-level policy was commanding to perform specific tasks such as moving to a specific position. This served as the input to the low-level policy which commanded in this case the joints of the robots to achieve said goal. However, this hierarchy can be further stretched out and expanded into more depths [Lev+19]. Apart from having the benefit of possibly simpler training, also for continual learning.

   - **Facilitates Learning:** For the case of the hopper and the walker, it was necessary to use a CL strategy as the instabilities pose a challenge. Especially, if the agent has to perform tasks that change every few steps. If, however, the high-level tasks

(move to a specific position, run at a certain speed ...) were to be broken down into more primitive movements, the agent would have more flexibility. With this approach we propose the hypothesis, that the hopper would be easier to train and the instabilities would result in a less complicated challenge due to the added flexibility.

- **Continual Learning:** In Section 6.4 it was shown that the proposed method can be successfully applied to an DPMM inference module. The DPMM architecture was used in part due to the ability to represent models of potentially infinite parameters, thus making it suitable for continual learning. However, the method employed here restricts the application to the tasks learned during the multi-task learning stage. If instead of learning these tasks, we expand the hierarchical structure to learn more basic movements that combined solve these high-level tasks, it would be possible to apply this method to continual learning.

2. **Reduce Sim-to-Real Gap:**
There is research that has tackled the problem of the sim-to-real gap with MRL techniques [Arn+19]. Instead of training a model to solve different tasks, the authors trained a model to solve the same task in different scenarios. To the best of our knowledge, there hasn't been any research that has tried to tackle the sim-to-real gap for MRL algorithms. The method described in this thesis can also be applied for this purpose. An approach would be to train the low-level policy in the real world with for example imitation learning techniques. Thus making the training safer. Once the lower-level policy has been trained in the real world, the task inference could be reused from a training in simulation.

Appendix A

ELBO derivation

The VAE is of stochastic nature. Therefore the objective can be defined as the maximum of the underlying data which in turn is the marginal likelihood of the joint distribution of the observed data and the latent variable $p_\theta(X,Z)$: $p_\theta(X) = \int p_\theta(X,Z)$. Due to the integral, this optimization is usually intractable.

The computation of $p_\theta(z|x)$ is intractable due to the appearance of $p_\theta(X)$ when applying Bayes' theorem. To address this challenge, VAEs introduce a variational distribution $q_\phi(z|x)$ to approximate the true posterior which represents the encoder. This allows to represent the log likelihood as an expectation:

$$\begin{aligned} \log p_\theta(x) = \log \int_z q_\phi(z|x) \frac{p_\theta(x,z)}{q_\phi(z|x)}\, dz = \log \mathbb{E}_{q\phi(z|x)} \left[\frac{p_\theta(x,z)}{q_\phi(z|x)} \right] \\ \log p_\theta(x) \geq \mathbb{E}_{q\phi(z|x)} \left[\log \frac{p_\theta(x,z)}{q_\phi(z|x)} \right] \end{aligned} \tag{A.1}$$

From conditional probability we know: $P(X,Z) = P(X|Z) \cdot P(Z)$. This results in the following equation which simplifies to the ELBO (Evidence Lower Bound) equation:

$$\begin{aligned} \mathbb{E}_{q_\phi(z|x)} \left[\log \frac{p_\theta(x,z)}{q_\phi(z|x)} \right] &= \mathbb{E}_{q_\phi(z|x)} [\log p_\theta(x|z)] - \mathbb{E}_{q_\phi(z|x)} \left[\log \frac{q_\phi(z|x)}{p_\theta(z)} \right] \\ &= \mathbb{E}_{q_\phi(z|x)} [\log p_\theta(x|z)] - D_{KL}(q_\phi(z|x)||p_\theta(z)) = ELBO \end{aligned} \tag{A.2}$$

From above we know:

$$\log p_\theta(x) \geq \mathbb{E}_{q\phi(z|x)} \left[\log \frac{p\theta(x,z)}{q_\phi(z|x)} \right] = ELBO \tag{A.3}$$

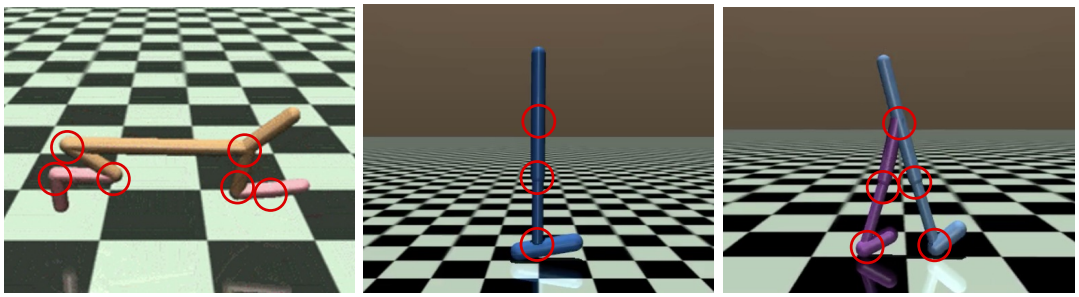
Therefore, the ELBO equation serves as a lower bound to the likelihood of the original data. This allows for an efficient optimization of our objective function.

Appendix B

Environment/Task Specifications

MuJoCo Agent	Links	Joints/Degrees of freedom
Toy (position)	1	1
Toy (position & velocity)	1	1
Toy (position, velocity, rotation, jump)	3	3
Cheetah	9	6
Hopper	4	3
Walker2d	7	6

Table B.1: Number of links and controllable joints for MuJoCo agents



(a) Image of the cheetah agent. **(b)** Image of the hopper agent. **(c)** Image of the walker agent.

Figure B.1: Images of the cheetah, hopper and walker agents. The controllable joints are marked with a red circle.

Base Task	Task Specification
Goal front/back	$[\pm 2, \pm 15]$ in m
Run forward/backward	$[\pm 1.0, \pm 3.0]$ in m/s
Jump	$[\pm 1.5, \pm 3.0]$ in m/s
Stand back/front	$[\pm 1.0, \pm 3.0]$ in rad

Table B.2: Tabular showing task specifications. These specifications are valid for the respective tasks unless states otherwise

Bibliography

[Ant74] Antoniak, C. E. "Mixtures of Dirichlet Processes with Applications to Bayesian Nonparametric Problems". In: *The Annals of Statistics* 2.6 (1974), pp. 1152–1174. DOI: 10.1214/aos/1176342871. URL: https://doi.org/10.1214/aos/1176342871.

[Arn+19] Arndt, K., Hazara, M., Ghadirzadeh, A., and Kyrki, V. *Meta Reinforcement Learning for Sim-to-real Domain Adaptation*. 2019. arXiv: 1909.12906 [cs.CV]. URL: https://arxiv.org/abs/1909.12906.

[Bin+15] Bing, Z., Lerch1, D., Huang, K., and Knoll, A. "Meta-Reinforcement Learning in Non-Stationary and Dynamic Environments". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2015).

[Bro+20] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. *Language Models are Few-Shot Learners*. 2020. arXiv: 2005.14165 [cs.CL].

[Che+23] Chen, Y., Ye, R., Tao, Z., Liu, H., Chen, G., Peng, J., Ma, J., Zhang, Y., Ji, J., and Zhang, Y. *Reinforcement Learning for Robot Navigation with Adaptive Forward Simulation Time (AFST) in a Semi-Markov Model*. 2023. arXiv: 2108.06161 [cs.RO]. URL: https://arxiv.org/abs/2108.06161.

[Dua+16] Duan, Y., Schulman, J., Chen, X., Bartlett, P. L., Sutskever, I., and Abbeel, P. *RL^2: Fast Reinforcement Learning via Slow Reinforcement Learning*. 2016. arXiv: 1611.02779 [cs.AI].

[Dur23] Durmann, J. *Meta-Reinforcement Learning*. 2023.

[Fer73] Ferguson, T. S. "A Bayesian Analysis of Some Nonparametric Problems". In: *The Annals of Statistics* 1.2 (1973), pp. 209–230. DOI: 10.1214/aos/1176342360. URL: https://doi.org/10.1214/aos/1176342360.

[FAL17] Finn, C., Abbeel, P., and Levine, S. *Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks*. 2017. arXiv: 1703.03400 [cs.LG].

[Fra+17] Frans, K., Ho, J., Chen, X., Abbeel, P., and Schulman, J. *Meta Learning Shared Hierarchies*. 2017. arXiv: 1710.09767 [cs.LG].

[GL15] Ganin, Y. and Lempitsky, V. *Unsupervised Domain Adaptation by Backpropagation*. 2015. arXiv: 1409.7495 [stat.ML].

[Geh+21] Gehring, J., Synnaeve, G., Krause, A., and Usunier, N. *Hierarchical Skills for Efficient Exploration*. 2021. arXiv: 2110.10809 [cs.LG].

[GM01] Ghavamzadeh, M. and Mahadevan, S. "Continuous-Time Hierarchical Reinforcement Learning". In: (May 2001).

[Gup+18] Gupta, A., Mendonca, R., Liu, Y., Abbeel, P., and Levine, S. *Meta-Reinforcement Learning of Structured Exploration Strategies*. 2018. arXiv: 1802.07245 [cs.LG].

[HYC01] Hochreiter, S., Younger, A. S., and Conwell, P. R. "Learning to Learn Using Gradient Descent". In: *Artificial Neural Networks — ICANN 2001*. Ed. by Dorffner, G., Bischof, H., and Hornik, K. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 87–94.

[HS13] Hughes, M. and Sudderth, E. "Memoized online variational inference for Dirichlet process mixture models". In: *Advances in Neural Information Processing Systems* (Jan. 2013).

[KSS20] Kirsch, L., Steenkiste, S. van, and Schmidhuber, J. *Improving Generalization in Meta Reinforcement Learning using Learned Objectives*. 2020. arXiv: 1910.04098 [cs.LG].

[Ler20] Lerch, D. *Meta Reinforcement Learning in Non-Stationary and Dynamic Environments*. 2020.

[Lev+19] Levy, A., Konidaris, G., Platt, R., and Saenko, K. *Learning Multi-Level Hierarchies with Hindsight*. 2019. arXiv: 1712.00948 [cs.AI]. URL: https://arxiv.org/abs/1712.00948.

[Li+17a] Li, D., Yang, Y., Song, Y.-Z., and Hospedales, T. M. *Learning to Generalize: Meta-Learning for Domain Generalization*. 2017. arXiv: 1710.03463 [cs.LG].

[Li+22] Li, N., Li, W., Jiang, Y., and Xia, S.-T. "Deep Dirichlet process mixture models". In: *Proceedings of the Thirty-Eighth Conference on Uncertainty in Artificial Intelligence*. Ed. by Cussens, J. and Zhang, K. Vol. 180. Proceedings of Machine Learning Research. PMLR, Aug. 2022, pp. 1138–1147. URL: https://proceedings.mlr.press/v180/li22c.html.

[Li+17b] Li, Z., Zhou, F., Chen, F., and Li, H. *Meta-SGD: Learning to Learn Quickly for Few-Shot Learning*. 2017. arXiv: 1707.09835 [cs.LG].

[LSX19] Liu, H., Socher, R., and Xiong, C. "Taming MAML: Efficient unbiased meta-reinforcement learning". In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Chaudhuri, K. and Salakhutdinov, R. Vol. 97. Proceedings of Machine Learning Research. PMLR, June 2019, pp. 4061–4071. URL: https://proceedings.mlr.press/v97/liu19g.html.

[Mis+18] Mishra, N., Rohaninejad, M., Chen, X., and Abbeel, P. *A Simple Neural Attentive Meta-Learner*. 2018. arXiv: 1707.03141 [cs.AI].

[Nac+18] Nachum, O., Gu, S., Lee, H., and Levine, S. *Data-Efficient Hierarchical Reinforcement Learning*. 2018. arXiv: 1805.08296 [cs.LG].

[Ope+19] OpenAI, Andrychowicz, M., Baker, B., Chociej, M., Jozefowicz, R., McGrew, B., Pachocki, J., Petron, A., Plappert, M., Powell, G., Ray, A., Schneider, J., Sidor, S., Tobin, J., Welinder, P., Weng, L., and Zaremba, W. *Learning Dexterous In-Hand Manipulation*. 2019. arXiv: 1808.00177 [cs.LG]. URL: https://arxiv.org/abs/1808.00177.

[OD23] Orr, J. and Dutta, A. "Multi-Agent Deep Reinforcement Learning for Multi-Robot Applications: A Survey". In: *Sensors* 23.7 (2023). ISSN: 1424-8220. DOI: 10.3390/s23073625. URL: https://www.mdpi.com/1424-8220/23/7/3625.

[Rak+19] Rakelly, K., Zhou, A., Quillen, D., Finn, C., and Levine, S. *Efficient Off-Policy Meta-Reinforcement Learning via Probabilistic Context Variables*. 2019. arXiv: 1903.08254 [cs.LG].

[San+16] Santoro, A., Bartunov, S., Botvinick, M., Wierstra, D., and Lillicrap, T. "Meta-learning with memory-augmented neural networks". In: *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*. ICML'16. New York, NY, USA: JMLR.org, 2016, pp. 1842–1850.

[Al-+18] Al-Shedivat, M., Bansal, T., Burda, Y., Sutskever, I., Mordatch, I., and Abbeel, P. *Continuous Adaptation via Meta-Learning in Nonstationary and Competitive Environments*. 2018. arXiv: 1710.03641 [cs.LG].

[Son+20] Song, X., Gao, W., Yang, Y., Choromanski, K., Pacchiano, A., and Tang, Y. *ES-MAML: Simple Hessian-Free Meta Learning*. 2020. arXiv: 1910.01215 [cs.LG].

[Tob+17] Tobin, J., Fong, R., Ray, A., Schneider, J., Zaremba, W., and Abbeel, P. *Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World*. 2017. arXiv: 1703.06907 [cs.RO].

[TET12] Todorov, E., Erez, T., and Tassa, Y. "MuJoCo: A physics engine for model-based control". In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2012, pp. 5026–5033.

[Wan+17] Wang, J. X., Kurth-Nelson, Z., Tirumala, D., Soyer, H., Leibo, J. Z., Munos, R., Blundell, C., Kumaran, D., and Botvinick, M. *Learning to reinforcement learn*. 2017. arXiv: 1611.05763 [cs.LG].

[WD92] Watkins, C. and Dayan, P. "Technical Note: Q-Learning". In: *Machine Learning* 8 (May 1992), pp. 279–292. DOI: 10.1007/BF00992698.

[ZQW20] Zhao, W., Queralta, J. P., and Westerlund, T. "Sim-to-Real Transfer in Deep Reinforcement Learning for Robotics: a Survey". In: *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*. 2020, pp. 737–744. DOI: 10.1109/SSCI47803.2020.9308468.