Technische Universität Hamburg

Data Science Foundations

Morphological Computation as Intrinsic Reward for Reinforcement Learning

Master's Thesis
am Institut Data Science Foundations
Technische Universität Hamburg

vorgelegt von
Omar Baiazid
am
15.09.2024

Erstprüfer:	Prof. Dr. N. Ay
Zweitprüfer	Dr. Manfred Eppe
Betreuung:	Dr. Manfred Eppe

Omar Baiazid
geboren am: 01.06.1998
Matrikelnummer: 21760121
Studiengang: Computer Science

Abstract

This thesis explores the integration of Morphological Computation (MC) as an intrinsic reward to Reinforcement Learning (RL). Integrating MC to RL is a novel method designed to accelerate and enhance the learning process. By incorporating the agent's physical properties into the learning algorithm, MC helps optimize the actions taken by the agent to leverage the dynamics in the environment, ultimately enhancing its performance.

Our approach focuses on enhancing the Soft Actor-Critic (SAC) algorithm by modifying its reward structure to include contributions from MC. We examine different rates of MC and compare them to the baseline performance of the SAC algorithm without MC.

Experimental results demonstrate that integrating MC significantly accelerates the learning process and improves the performance of the SAC algorithm. These findings are discussed in the context of intrinsic reward mechanisms, positioning MC as a promising approach for advancing RL algorithms.

Zusammenfassung

Diese Arbeit untersucht die Integration von Morphological Computation (MC) als intrinsische Belohnung in Reinforcement Learning (RL). Diese Integration ist eine neuartige Methode zur Beschleunigung und Verbesserung des Lernprozesses. Durch die Einbeziehung der physikalischen Eigenschaften des Agenten in den Lernalgorithmus hilft MC, die vom Agenten ausgeführten Aktionen zu optimieren und letztlich seine Leistung zu verbessern. Unser Ansatz konzentriert sich darauf, den Soft Actor-Critic (SAC) Algorithmus zu verbessern, indem wir seine Belohnungsstruktur modifizieren, um Beiträge von MC einzubeziehen. Wir untersuchen verschiedene MC-Raten und vergleichen diese mit der Basisleistung des SAC-Algorithmus. Experimentelle Ergebnisse zeigen, dass die Integration von MC den Lernprozess erheblich beschleunigt und die Leistung des SAC-Algorithmus. Diese Ergebnisse werden im Kontext intrinsischer Belohnungsmechanismen diskutiert und positionieren MC als vielversprechenden Ansatz zur Weiterentwicklung von RL-Algorithmen.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Reinforcement Learning (RL) is a type of Machine Learning where an agent learns to make decisions by interacting with an environment. The agent performs actions and receives feedback in the form of rewards, learning to maximize cumulative rewards over time. This process is analogous to training a pet, where the pet learns behaviors through a system of rewards like being fed after performing a desired series of actions.

In RL, rewards can be classified into two categories: Intrinsic and extrinsic Rewards. Intrinsic rewards come from within, such as curiosity or joy and satisfaction of mastering a new skill or solving a challenging problem. One can imagine a child who is curious and enjoys exploring their environment for the sheer fun of it. For instance, a child might be intrinsically motivated to explore a playground, discovering new ways to climb and jump, simply because they find it enjoyable and interesting. Extrinsic rewards, on the other hand, are external incentives like prizes, medals, or grades. Another child might be driven to perform tasks primarily to earn these rewards, such as winning a race to get a gold medal or completing homework to receive praise from parents or teachers.

While extrinsic rewards are important, more advanced RL algorithms also incorporate intrinsic rewards. This combination is crucial for effective exploration, especially when dealing with complex tasks and reward systems. Intrinsic rewards can guide the agent to explore more efficiently, discovering strategies and behaviors that may not be immediately associated with extrinsic rewards but are beneficial in the long run. This blend of reward types helps in balancing exploration and exploitation, leading to more robust learning outcomes.

Despite significant advancements in RL, the field faces challenges related to the efficiency and stability of the learning process. Among various RL algorithms, the Soft Actor-Critic (SAC) algorithm stands out due to its robust performance and ability to balance exploration and exploitation through entropy maximization. However, even state-of-the-art algorithms like SAC can benefit from further optimization to enhance learning speed and efficacy.

This thesis investigates a novel approach to optimizing the RL algorithms by in-

tegrating Morphological Computation (MC) as an intrinsic reward. MC uses the physical properties and dynamics of an agent's body and the environment to enhance controlling tasks by introducing a new dimension of information for the agent. This allows the agent to utilize information provided by the environment, such as the dynamics of the environment and its own body, to make more informed decisions.

Incorporating MC into our RL framework is like the intrinsically motivated child exploring and utilizing the dynamics of its environment. For example, they could learn to swing higher by timing their leg movements perfectly with the swing's motion or use the momentum of a slide to propel themselves farther.

The primary goal of this research is to demonstrate that incorporating MC into the RL framework can lead to faster learning and improved performance. We propose a novel modification to RL algorithms that incorporates MC into its reward structure. By doing so, we create a hybrid reward system that combines the intrinsic benefits of MC with the robust exploration-exploitation balance of SAC.

To validate our approach, we conduct a series of experiments comparing different rates of MC integration with the baseline performance of SAC without MC. Finally showing the results of integration of the MC to the baseline SAC algorithm across given benchmark tasks.

1.1 Research Question

The central research question guiding this thesis is: " *Will the integration of Morphological Computation (MC) into Reinforcement Learning (RL) lead to better performance of RL agents*?" Specifically, we seek to explore whether MC can enhance the learning efficiency and overall effectiveness of the Soft Actor-Critic (SAC) algorithm, ultimately contributing to more robust and adaptive RL systems.

1.2 Outline

This thesis is structured as follows: we begin with an in-depth background section that introduces the fundamentals of RL, details the SAC algorithm, and explains the principles of our MC approach. Following this, we present a review of state-of-the-art intrinsic reward mechanisms and position our approach within this context. The approach section details the integration of MC with SAC, including theoretical justifications and implementation details. The experiments section outlines our methodology, experimental setup, and results. Finally, we discuss the implications of our findings, compare them with existing approaches, and conclude with potential directions for future research.

Chapter 2

Background

This chapter offers a thorough overview of the essential concepts required for a deeper grasp of the thesis topic. Section 2.1 lays out the fundamental principles of probabilistic neural networks, which serve as the foundation for implementing and assessing Morphological Computation and thus calculating the intrinsic reward. Section 2.2 explores the core concepts of traditional Reinforcement Learning (RL), leading to a discussion on Deep Reinforcement Learning (DRL). We will focus on Actor-Critic methods, with particular emphasis on Soft Actor-Critic (SAC). Finally, the chapter concludes by introducing Morphological Computation, following a review of several state-of-the-art intrinsic rewards.

2.1 Probabilistic Neural Networks

For the implementation of Morphological Computation, we require specialized neural networks. These networks operate slightly differently from standard ones. Standard artificial neural networks (ANNs) are designed to generate deterministic outputs for tasks like classification or regression. These outputs are compared to the ground truth, and the difference between them is measured using loss functions such as Mean Squared Error (MSE). The goal is to minimize this difference, which is done iteratively using the backpropagation algorithm. On the other hand Neural networks trained using Maximum Likelihood Estimation (MLE) approach the problem from a probabilistic perspective. Instead of providing a single deterministic output, these networks output a probability distribution over possible outcomes. For instance, in the case of regression, instead of predicting a single value, the network might predict the parameters of a distribution (e.g., mean and variance of a Gaussian distribution). The training objective is to maximize the likelihood of observing the true data given the predicted distribution, which involves minimizing a function like the negative log-likelihood for example. To understand how these networks function, we first need to explore Maximum Likelihood Estimation, which serves as the foundation for this approach.

In the context of statistical modeling and machine learning, MLE is a powerful method for estimating the parameters of a given model. MLE is grounded in the

concept of likelihood, which fundamentally differs from probability. Let x represent the observed data, and θ the set of model parameters. A *probability density function* (pdf), denoted as $p(x|\theta)$, defines the likelihood of observing data x given parameters θ. The pdf provides the probability density for different values of x. Crucially, in MLE, the likelihood function is not viewed as a function of x, but rather as a function of θ, with x held constant. Therefore, the likelihood function is given by:

$$\mathcal{L}(\theta|x) = p(x|\theta)$$

For a dataset $X = \{x^{(1)}, x^{(2)}, \ldots, x^{(m)}\}$ of m independent and identically distributed (IID) samples, the joint likelihood of the observed data is:

$$\mathcal{L}(\theta|X) = \prod_{i=1}^{m} p(x^{(i)}|\theta)$$

In practice, maximizing the log-likelihood simplifies computations due to the logarithmic properties of products, converting the product into a summation:

$$\log \mathcal{L}(\theta|X) = \sum_{i=1}^{m} \log p(x^{(i)}|\theta)$$

This is the function that MLE optimizes to find the parameter values θ that best explain the observed data.

2.1.1 Independence and Identically Distributed (IID) Assumption

The IID assumption plays a critical role in the formulation of MLE. Statistical independence means that for two random variables A and B, the joint distribution $P(A, B)$ can be factorized as the product of their marginal distributions:

$$P(A, B) = P(A)P(B)$$

This property extends to datasets, where the joint distribution of the entire dataset X can be written as the product of the individual likelihoods of each sample under the assumption that the samples are independent:

$$P(X|\theta) = \prod_{i=1}^{m} p(x^{(i)}|\theta)$$

2.1.2 Modeling and Optimization in MLE

In machine learning, the parameters θ are optimized to ensure that the model distribution $p_{\text{model}}(x|\theta)$ closely approximates the true data distribution $p_{\text{data}}(x)$. This is the essence of MLE: finding the parameter set θ_{MLE} that maximizes the likelihood of the observed data:

$$\theta_{\text{MLE}} = \arg\max_{\theta} \mathcal{L}(\theta|X)$$

or equivalently, by maximizing the log-likelihood:

$$\theta_{\text{MLE}} = \arg\max_{\theta} \sum_{i=1}^{m} \log p(x^{(i)}|\theta)$$

In supervised learning tasks, where the model predicts labels y based on inputs x, the likelihood function is conditioned on the input-output pairs, and the MLE objective becomes:

$$\theta_{\text{MLE}} = \arg\max_{\theta} \sum_{i=1}^{m} \log p(y^{(i)}|x^{(i)}, \theta)$$

2.1.3 Cross-Entropy and Negative Log-Likelihood

The cross-entropy loss function and negative log-likelihood (NLL) function are used in classification tasks as loss functions and provide the same thing. The cross-entropy function is defined as:

$$L_{\text{CE}}(p_{\text{data}}, p_{\text{model}}) = -\sum_{x} p_{\text{data}}(x) \log p_{\text{model}}(x \mid \theta)$$

Minimizing the cross-entropy loss is mathematically equivalent to maximizing the log-likelihood in Maximum Likelihood Estimation (MLE). In the context of classification. The same works for the Negative Log-Likelihood, which can be expressed as:

$$L_{\text{NLL}} = -\log P(y \mid x)$$

where:

- y is the true class label.
- $P(y \mid x)$ is the predicted probability assigned to the true class y, given the input x.

This formulation reflects the log of the predicted probability for the true class, with a negative sign. During training, our objective is to minimize this value. If the true labels y are one-hot encoded (i.e., where only the correct class has a value of 1, and all other classes are 0), the Cross-Entropy Loss and the Negative Log-Likelihood Loss are equivalent. For a single data point, the relationship can be expressed as:

$$L_{\text{CE}}(y, \hat{y}) = -\log(\hat{y}) = L_{\text{NLL}}$$

Thus, in both cases, we are minimizing the negative log of the probability assigned to the true class. This minimization process is equivalent to maximizing the likelihood of the correct prediction.

Neural networks trained using Maximum Likelihood Estimation (MLE) output parameters such as the mean μ and standard deviation σ, which define a Gaussian distribution. The goal is to model this distribution in such a way that it closely approximates the true distribution of the observed data. To achieve this, the model seeks to minimize the difference between the predicted distribution and the actual data distribution. This is typically done by minimizing the Negative Log-Likelihood (NLL), which is as mentioned above a measure of how well the predicted probability distribution explains the observed data. A more detailed discussion of this process will be covered in Chapter 3.

2.2 Reinforcement Learning (RL)

RL has seen numerous applications and successes across various domains. Notable achievements include training computers to control robots in both simulated and real-world environments. RL has also been instrumental in developing advanced AI for complex strategy games such as Go, Dota, and Chess, enabling computers to play Atari games using raw pixel data, and training simulated robots to follow human instructions. At the core of RL are two main components: the agent and the environment. The environment represents the world in which the agent operates and with which it interacts. During each interaction step, the agent receives an observation of the state of the environment (which may be partial) and selects an action. The action affects the environment, which in turn provides the agent with a new observation and a reward signal indicating the quality of the current state. The agent's objective is to maximize its cumulative reward, also known as the return. RL methodologies provide ways for the agent to learn behaviors that help achieve this goal. To delve deeper into RL, we need to introduce several key terms:

- States S and observations O.
- Action space A.
- Reward r.
- Policy $\pi(\cdot \mid s_t)$.
- Return $G(\tau)$.

A state s provides a complete description of the environment at a given time, with no hidden information, while an observation o is a partial description of the state, potentially omitting some details. In deep RL, states and observations are typically represented as vectors, matrices, or higher-order tensors. For example, a visual observation could be represented by a matrix of pixel values, while a state

might be described by more information where the exact position of the agent its velocity, and other properties are given. An environment is considered fully observed when the agent has access to the complete state and partially observed when it only has access to observations. In RL the agent mostly has only access to the observations after taking an action.

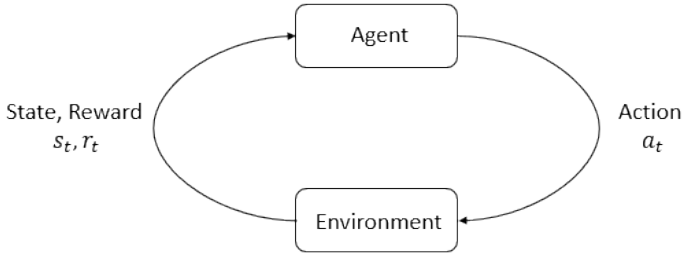


Figure 2.1: Agent-environment interaction loop.

The action space of an environment is the set of all possible actions the agent can take. Action spaces can be discrete, with a finite number of actions (e.g., Chess or Go), or continuous, where actions are represented by real-valued vectors (e.g., controlling a robot). It is important to know, with which action space we are dealing in advance to be able to choose the right RL algorithm [1]

In reinforcement learning, the reward function R plays a crucial role in guiding the learning agent. It is typically defined as a function of the current state s_t, the action taken a_t, and the resulting next state s_{t+1}:

$$r_t = R(s_t, a_t, s_{t+1})$$

The reward r_t can take on positive, negative, or neutral values, depending on the desired behavior and the environment's dynamics.

Markov Decision Process (MDP)

Now that we know what states, actions and rewards are we can introduce the Markov Decision Process (MDP), which provides a mathematical framework for modeling decision-making in situations where outcomes are partly random and partly under the control of a decision-maker. The Markov Assumption, a key property of MDPs, posits that the future state of the process depends only on the present state and the action taken, not on the sequence of events that preceded it. This property implies that the system is "memoryless," and the state at time step t is independent of past agent-environment interactions.

An MDP is defined by a five-tuple (S, A, P, R, γ), where:

[1] Some RL algorithms are designed specifically for discrete or continuous spaces.

- S, A and R are states, actions and rewards respectively
- P: The state transition probability function, $P(s_t, a_t, s_{t+1})$, which represents the probability of transitioning to state s_{t+1} from state s_t after taking action a_t. Formally, $P : S \times A \times S \to [0, 1]$, where $\sum_{s' \in S} P(s_t, a_t, s_{t+1}) = 1$.
- γ: The discount factor, $\gamma \in [0, 1]$, which is used to discount future rewards to their present value. It captures the notion that immediate rewards are generally more valuable than future rewards. A discount factor close to 1 indicates that future rewards are almost as important as immediate ones, while a factor close to 0 emphasizes short-term rewards.

An MDP is considered finite if the sets of states S and actions A are finite. This property simplifies the computational aspects of solving the MDP, making it feasible to compute optimal policies using various algorithms like Dynamic Programming, Monte Carlo methods, or Temporal Difference Learning. The Markov Assumption and the concept of MDPs form the foundational basis for various algorithms in reinforcement learning, enabling the systematic study and solution of sequential decision-making problems under uncertainty. The goal in an MDP is typically to find a policy that maximizes the expected cumulative reward.

A policy π is a mapping function that, for a given state, returns an action. In general, it is a strategy used by the agent to select actions based on its observations. Policies can be either deterministic, where a specific action is chosen for a given observation, or stochastic, where actions are selected according to a probability distribution. In deep reinforcement learning (RL), policies are often parameterized by a set of parameters (e.g., weights and biases of a neural network). Since we are dealing with stochastic policies in this work, we will denote stochastic policies as π. The return of a policy is thus a distribution over actions, and the action with the highest probability is often chosen, i.e., $a_t \sim \pi(\cdot|s_t)$. The parameters of the policy are usually denoted by θ, leading to the notation π_θ.

Lastly, the return G is the cumulative reward over a trajectory $G(\tau)$. There are two common return formulations which are discounted and undiscounted. Yet since it is obvious what an undiscounted return is, we will only present the discounted return:

$$G(\tau) = \sum_{t=0}^{T} \gamma^t r_t. \tag{2.1}$$

The goal of reinforcement learning is to find an optimal policy π^* that maximizes the expected return from each state s. Formally, the objective is to maximize the expected cumulative reward, or the expected return, over all possible states and actions. This can be expressed as:

$$\pi^* = \arg\max_{\pi} \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid \pi\right]$$

Where:

- $\mathbb{E}$ denotes the expected value.
- $\sum_{t=0}^{\infty} \gamma^t r_t$ is the cumulative discounted reward.

To achieve this, we often utilize the value function $V^\pi(s)$ and the action-value function $Q^\pi(s, a)$:

Value Function ($V^\pi(s)$): The value of a state s under a policy π is the expected return starting from s and following π:

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

Action-Value Function ($Q^\pi(s, a)$): The value of taking action a in state s under a policy π is the expected return starting from s, taking action a, and thereafter following policy π:

$$Q^\pi(s, a) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s, a_0 = a \right]$$

The optimal value function $V^*(s)$ and optimal action-value function $Q^*(s, a)$ are defined as:

$$V^*(s) = \max_\pi V^\pi(s)$$

$$Q^*(s, a) = \max_\pi Q^\pi(s, a)$$

The optimal policy π^* can then be derived from the optimal action-value function $Q^*(s, a)$:

$$\pi^*(a \mid s) = \begin{cases} 1 & \text{if } a = \arg\max_{a'} Q^*(s, a') \\ 0 & \text{otherwise} \end{cases}$$

Note that the value function and the action-value function are the basis on which reinforcement learning builds.

2.2.1 Temporal Difference Methods

Temporal Difference (TD) methods are a class of model-free reinforcement learning algorithms that are particularly useful for online learning and continuous RL tasks. Unlike Monte Carlo methods, which require complete episodes to update the policy, TD methods update the policy at each time step. This characteristic allows TD methods to be applied to ongoing tasks where the episode may not have a clear end or is very long. Moreover, TD methods do not require a model of the environment, making them broadly applicable and less computationally demanding. A significant advantage of TD methods is their faster convergence compared to Monte Carlo methods. This efficiency arises from TD methods' ability to bootstrap their estimates, meaning they update the value function using other learned

estimates instead of waiting for the final outcome of the episode.

In TD methods, the true expected return G_t is not known and is approximated using the TD target. The TD target is a combination of the immediate reward R_{t+1} and the discounted value of the next state $\gamma V(S_{t+1})$. This can be seen as an approximation of the Bellman equation:

$$\text{TD}_{\text{target}} = R_{t+1} + \gamma V(S_{t+1}) \tag{2.2}$$

The update rule for TD methods involves adjusting the value of the current state S_t based on the difference between the TD target and the current value estimate $V(S_t)$. This difference is known as the TD error:

$$\text{TD}_{\text{error}} = R_{t+1} + \gamma V(S_{t+1}) - V(S_t). \tag{2.3}$$

The general update rule for the value function V is:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)], \tag{2.4}$$

where α is the learning rate, controlling the step size during the update. The pseudo-code for a general TD method is outlined in Algorithm 1:

Algorithm 1 General TD Algorithm

Input: Policy π, learning rate $\alpha \in (0, 1]$
Initialize $V(s)$ for all $s \in S$, arbitrarily, except $V(\text{terminal}) = 0$
for each episode **do**
  Initialize S_t
  while S_t is not terminal **do**
    Choose action A_t given by policy π for state S_t
    Take action A_t, observe reward R_{t+1} and next state S_{t+1}
    Compute TD target: $R_{t+1} + \gamma V(S_{t+1})$
    Update $V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$
    $S_t \leftarrow S_{t+1}$
  end while
end for

In summary, TD methods provide a flexible and efficient approach to policy evaluation and improvement in reinforcement learning. By using the TD target and TD error, these methods iteratively refine the value function estimates, enabling the learning of optimal policies without the need for a model of the environment or complete episodes.

2.2.2 Deep Reinforcement Learning (DRL)

In classical Reinforcement Learning (RL), as discussed before, algorithms typically employ a tabular setting to represent value or action-value functions. However, this

approach has significant limitations. One major issue is the scalability; the use of tables constrains classical RL methods to tasks with relatively small state and action spaces. In practical applications, the state space can become extraordinarily large, making it impractical to visit and evaluate every possible state to retrieve the corresponding action-state pair values. This problem is further compounded by hardware memory constraints, which restrict the feasible size of the value table. Additionally, tabular methods do not support the sharing of knowledge across similar states. This lack of generalization requires that each state be learned individually, which can lead to inefficient learning processes and prolonged training durations. To address these challenges, Deep Reinforcement Learning (DRL) employs Artificial Neural Networks (ANNs) as function approximators in place of tabular representations. ANNs are capable of approximating complex, non-linear functions and can extract relevant features from raw input data. This capability allows DRL methods to generalize across states, significantly improving learning efficiency and reducing training times. Furthermore, the use of ANNs enables DRL to scale to problems with extensive state spaces, where maintaining and updating a complete table of values is impractical. The shift from tabular methods to DRL represents a substantial advancement in the field, allowing for the application of RL techniques to more complex and realistic problems.

Value-Based Methods

Value-based methods in Deep Reinforcement Learning (DRL) extend the concepts from classical Reinforcement Learning, specifically those from Temporal Difference (TD) Methods discussed in Section 2.2.1. In this approach, the discrete value table used in traditional RL is replaced with a Deep Neural Network (DNN), which serves as a function approximator. This network takes the state as input and outputs a set of values corresponding to the possible actions. The output of the network is interpreted as the estimated value of each action given the current state, essentially approximating the action-value function $Q(s, a)$. A policy is then applied to this output to select the final action. For instance, an ϵ-greedy policy might be used, where the agent usually chooses the action with the highest estimated value but occasionally selects a random action to ensure exploration. This approach allows the agent to learn more efficiently from high-dimensional state spaces and to generalize across similar states, overcoming the limitations of the tabular methods used in traditional RL.

Policy-Based Methods:

Policy-based methods directly parameterize the stochastic policy[2] $\pi_\theta(a|s)$ and update its parameters θ to maximize the expected return. The objective function for

[2]Here, another advantage of the stochastic policy arises: by using a stochastic policy, we obtain a smooth gradient and smooth optimization Problem. This is quite beneficial since we apply it repeatedly. We wish for a result that does not diverge much from the previous one, which could happen if we use a deterministic policy.

policy optimization is given by:

$$J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right] \tag{2.5}$$

In many cases, the policy can be simpler to optimize than the actual value function or action-value function. Optimizing a policy is often more explicit, whereas optimizing an action-value function can be more complex and implicit. To illustrate this, consider a task where the goal is for an agent to grab an object. Conceptually, guiding the agent to the object and having it grasp the object is simpler than learning the corresponding value function. Learning the value function requires precisely calculating the reward associated with each movement. Furthermore, the value function depends on the dynamics of the environment, which are often unknown or difficult to model. This is one reason why the action-value function is often preferred, as it bypasses the need to explicitly model the environment's dynamics. However, optimizing the action-value function in higher-dimensional and continuous environments presents its own challenges. High-dimensional spaces and continuous action spaces increase the complexity of the optimization process, making it computationally intensive and difficult to achieve convergence. Therefore, while the policy-based approach can be more straightforward, it requires careful consideration of the trade-offs involved in the complexity of the environment and the specific requirements of the task at hand. These considerations are crucial for designing efficient algorithms that can effectively learn and perform in diverse and dynamic environments.[3]

The policy gradient theorem provides the gradient of the objective function, which is used to update the policy parameters:[4]

$$\nabla_\theta J(\theta) = \sum_{\tau} P(\tau;\theta)\nabla_\theta \log P(\tau;\theta) R(\tau) \tag{2.6}$$

2.2.3 Actor-Critic Methods

Actor-Critic methods are a class of algorithms in reinforcement learning (RL) that combine the advantages of both policy-based and value-based methods. These methods are designed to overcome the limitations of each approach when used independently.

Origins and Motivation

The Actor-Critic framework arises from the need to optimize the policy directly (as in policy-based methods) while still leveraging the value function (as in value-

[3] A trajectory τ, which is a sequence of states and actions experienced by the agent: $\tau = (s_0, a_0, s_1, a_1, \dots)$, where the initial state s_0 is sampled from a start-state distribution. State transitions depend on the agent's actions and can be deterministic or stochastic. Its policy dictates the agent's actions.

[4] Note that this formula is valid, even if the reward is discontinuous or unknown. The derivative does not rely on the reward.

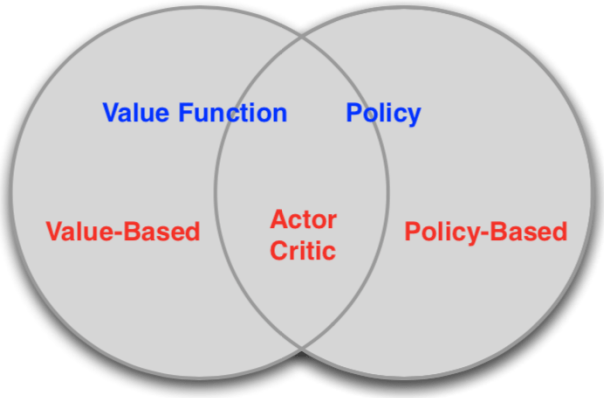


Figure 2.2: Actor-Critic.

based methods) to reduce variance and improve stability. The term "Actor-Critic" comes from the two main components of the algorithm:

- **Actor:** The component that updates the policy i.e. decides which action to take.
- **Critic:** The component that evaluates the action the actor takes, i.e., it estimates the value function.

Actor-Critic Framework

In the Actor-Critic framework, the actor updates the policy parameters θ, and the critic updates the value function parameters w. The actor uses the policy gradient theorem to update the policy, while the critic minimizes the temporal difference (TD) error to update the value function.

Actor Update

The actor updates the policy parameters θ using the policy gradient:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s)\hat{Q}(s,a)\right] \tag{2.7}$$

where $\hat{Q}(s,a)$ is the estimate of the action-value function provided by the critic.

Critic Update

The critic updates the value function parameters w by minimizing the TD error:

$$L(w) = \mathbb{E}_{(s,a,r,s')}\left[(r + \gamma V_w(s') - V_w(s))^2\right] \tag{2.8}$$

where $V_w(s)$ is the value function parameterized by w.

Advantages of Actor-Critic Methods

Actor-Critic methods benefit from both the stability of value-based methods and the direct policy optimization of policy-based methods. By using the critic's value function to reduce variance in policy updates, Actor-Critic methods achieve more stable and efficient learning. Actor-Critic methods represent a powerful approach to reinforcement learning, combining the strengths of policy-based and value-based methods. By leveraging both the policy and value function, these methods achieve efficient and stable learning, making them a cornerstone of modern RL algorithms.

2.2.4 Soft Actor Critic

The Soft Actor-Critic or short SAC is one of the most recent and powerful RL algorithms. In short SAC algorithm is an off-policy actor-critic method that maximizes the entropy. This algorithm has been developed by researchers at UC Berke-

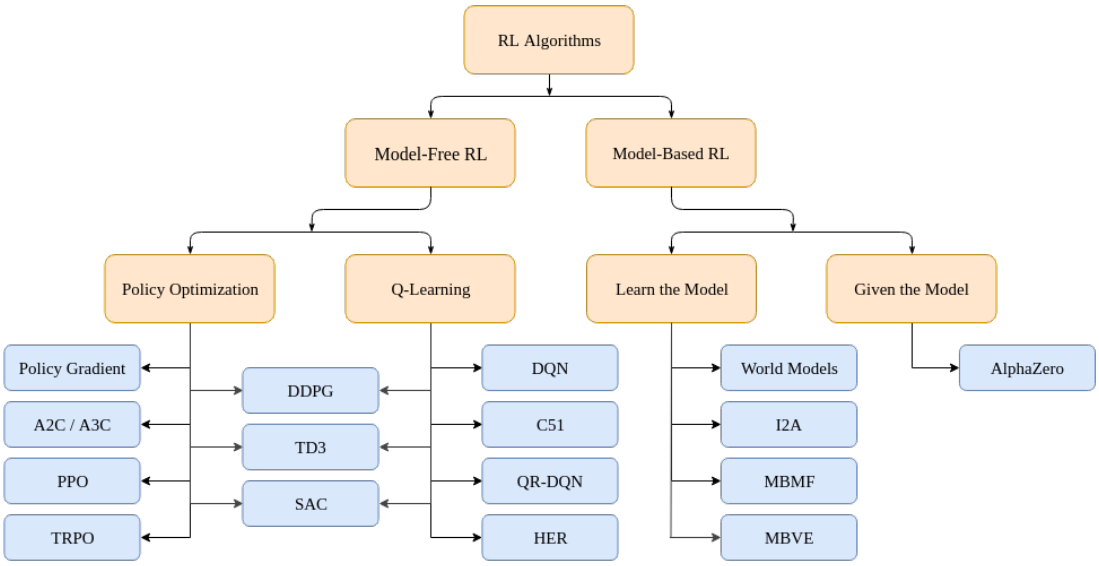


Figure 2.3: A Taxonomy of RL Algorithms

ley and has garnered significant attention. This algorithm not only distinguishes itself by demonstrating superior sample efficiency when compared to conventional Reinforcement Learning algorithms but also holds the promise of robustness in addressing issues related to convergence fragility. Its emergence has notably sparked interest within the academic and research communities, marking a significant stride in the field of RL methodologies. Normally there are a lot of challenges faced by model-free deep reinforcement learning algorithms, which often suffer from high sample complexity and fragile convergence properties. These issues necessitate meticulous tuning of hyperparameters, limiting the applicability of these methods in complex real-world scenarios. This SAC's approach involves maximizing expected reward while also maximizing entropy, encouraging randomness in actions. Unlike other methods based on this framework, which were formulated as

Q-learning methods, the proposed approach combines off-policy updates with a stable stochastic actor-critic formulation. To be able to understand Soft Actor-Critic, it's essential to first delve into entropy-regularized reinforcement learning. This approach introduces specific equations for value functions, setting it apart from traditional reinforcement learning settings. Entropy is a quantity that, roughly speaking, says how random a random variable is. If a coin is weighted so that it almost always comes up heads, it has low entropy; if it's evenly weighted and has a half chance of either outcome, it has high entropy. Let x be a random variable with probability mass or density function P. The entropy H of x is computed from its distribution P according to

$$H(P) = \mathbb{E}_{x\sim P}[-\log P(x)]. \tag{2.9}$$

In entropy-regularized reinforcement learning, the agent gets a bonus reward at each time step proportional to the entropy of the policy at that timestep. This changes the RL problem to:

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty} \gamma^t \left(R(s_t, a_t, s_{t+1}) + \alpha H\left(\pi(\cdot|s_t)\right)\right)\right]. \tag{2.10}$$

where $\alpha > 0$ is the trade-off coefficient. We can now define the slightly different value functions in this setting. V^π is changed to include the entropy bonuses from every timestep:

$$V^\pi(s) = \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty} \gamma^t \left(R(s_t, a_t, s_{t+1}) + \alpha H\left(\pi(\cdot|s_t)\right)\right) \middle| s_0 = s\right]. \tag{2.11}$$

Q^π is changed to include the entropy bonuses from every timestep except the first:

$$Q^\pi(s,a) = \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) + \alpha \sum_{t=1}^{\infty} \gamma^t H\left(\pi(\cdot|s_t)\right) \middle| s_0 = s, a_0 = a\right]. \tag{2.12}$$

With these definitions, V^π and Q^π are connected by:

$$V^\pi(s) = \mathbb{E}_{a\sim\pi} Q^\pi(s,a) + \alpha H\left(\pi(\cdot|s)\right). \tag{2.13}$$

and the Bellman equation for Q^π is:

$$\begin{aligned} Q^\pi(s,a) &= \mathbb{E}_{s'\sim P a'\sim\pi}[R(s,a,s') + \gamma\left(Q^\pi(s',a') + \alpha H\left(\pi(\cdot|s')\right)\right)] \\ &= \mathbb{E}_{s'\sim P}[R(s,a,s') + \gamma V^\pi(s')]. \end{aligned} \tag{2.14}$$

Up until now, we have discussed what entropy-regularized methods, in general, look like. Next, the details of SAC will be demonstrated. SAC concurrently learns a policy π_θ and two Q-functions Q_{ϕ_1}, Q_{ϕ_2}. Two variants of SAC are currently standard: one that uses a fixed entropy regularization coefficient α, and another that enforces an entropy constraint by varying α over the course of training. For

simplicity let's take the version with a fixed entropy regularization coefficient, but the entropy-constrained variant is generally preferred by practitioners. The SAC algorithm has changed a little bit over time. An older version of SAC also learns a value function V_ψ in addition to the Q-functions; this thesis will focus on the modern version that omits the extra value function since this is the version we used for training.

The SAC algorithm learns both Q-functions with MSBE minimization, by regressing to a single shared target. The shared target is computed using target Q-networks, and the target Q-networks are obtained by polyak averaging the Q-network parameters over the course of training, where the shared target makes use of the clipped double-Q trick. The target obviously also includes a term that comes from SAC's use of entropy regularization. Two important things to mention are that the next-state actions used in the target come from the current policy instead of a target policy and there is no explicit target policy smoothing.

Before we give the final form of the Q-loss, let's take a moment to discuss how the contribution from entropy regularization comes in. We'll start by taking our recursive Bellman equation for the entropy-regularized Q^π from earlier, and rewriting it a little bit by using the definition of entropy:

$$\begin{aligned} Q^\pi(s,a) &= \mathop{\mathbb{E}}_{s' \sim P a' \sim \pi} \left[R(s,a,s') + \gamma \left(Q^\pi(s',a') + \alpha H\left(\pi(\cdot|s')\right)\right)\right] \\ &= \mathop{\mathbb{E}}_{s' \sim P a' \sim \pi} \left[R(s,a,s') + \gamma \left(Q^\pi(s',a') - \alpha \log \pi(a'|s')\right)\right] \end{aligned} \tag{2.15}$$

and this results in approximately:

$$Q^\pi(s,a) \approx r + \gamma \left(Q^\pi(s', \tilde{a}') - \alpha \log \pi(\tilde{a}'|s')\right), \qquad \tilde{a}' \sim \pi(\cdot|s'). \tag{2.16}$$

We switch the next action notation to $\tilde{a}'$, instead of a', to highlight that the next actions have to be sampled fresh from the policy. SAC sets up the MSBE loss for each Q-function using this kind of sample approximation for the target. The only thing still undetermined here is which Q-function gets used to compute the sample backup. SAC uses the clipped double-Q trick and takes the minimum Q-value between the two Q approximations. Putting it all together, the loss functions for the Q-networks in SAC are:

$$L(\phi_i, \mathcal{D}) = \mathop{\mathrm{E}}_{(s,a,r,s',d) \sim \mathcal{D}} \left[\left(Q_{\phi_i}(s,a) - y(r,s',d) \right)^2 \right], \tag{2.17}$$

where the target is given by

$$y(r,s',d) = r + \gamma(1-d) \left(\min_{j=1,2} Q_{\phi_{\mathrm{targ},j}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s') \right), \qquad \tilde{a}' \sim \pi_\theta(\cdot|s'). \tag{2.18}$$

The policy should, in each state, act to maximize the expected future return plus expected future entropy. That is, it should maximize $V^\pi(s)$, which we expand out into

$$V^\pi(s) = \mathop{\mathbb{E}}_{a\sim\pi}\left[Q^\pi(s,a)\right] + \alpha H\left(\pi(\cdot|s)\right) = \mathop{\mathbb{E}}_{a\sim\pi}\left[Q^\pi(s,a) - \alpha \log \pi(a|s)\right]. \tag{2.19}$$

[5]The way we optimize the policy makes use of the reparameterization trick, in which a sample from $\pi_\theta(\cdot|s)$ is drawn by computing a deterministic function of state, policy parameters, and independent noise. To illustrate: following the authors of the SAC paper, a squashed Gaussian policy is used, which means that samples are obtained according to

$$\tilde{a}_\theta(s,\xi) = \tanh\left(\mu_\theta(s) + \sigma_\theta(s) \odot \xi\right), \qquad \xi \sim \mathcal{N}(0, I). \tag{2.20}$$

The reparameterization trick allows us to rewrite the expectation over actions (which contains a pain point: the distribution depends on the policy parameters) into an expectation over noise (which removes the pain point: the distribution now has no dependence on parameters):

$$\mathbb{E}_{a\sim\pi_\theta}\left[Q^{\pi_\theta}(s,a) - \alpha \log \pi_\theta(a|s)\right] = \mathbb{E}_{\xi\sim\mathcal{N}}\left[Q^{\pi_\theta}(s,\tilde{a}_\theta(s,\xi)) - \alpha \log \pi_\theta(\tilde{a}_\theta(s,\xi)|s)\right] \tag{2.21}$$

To get the policy loss, the final step is that we need to substitute Q^{π_θ} with one of our function approximators. SAC uses $\min_{j=1,2} Q_{\phi_j}$ (the minimum of the two Q approximators). The policy is thus optimized according to

$$\max_\theta \mathop{\mathbb{E}}_{s\sim\mathcal{D},\xi\sim\mathcal{N}}\left[\min_{j=1,2} Q_{\phi_j}(s,\tilde{a}_\theta(s,\xi)) - \alpha \log \pi_\theta(\tilde{a}_\theta(s,\xi)|s)\right] \tag{2.22}$$

SAC trains a stochastic policy with entropy regularization and explores it in an on-policy way. The entropy regularization coefficient α explicitly controls the explore-exploit trade-off, with higher α corresponding to more exploration, and lower α corresponding to more exploitation. The right coefficient (the one that leads to the stablest / highest-reward learning) may vary from environment to environment and could require careful tuning. At test time, to see how well the policy exploits what it has learned, we remove stochasticity and use the mean action instead of a sample from the distribution. This tends to improve performance over the original stochastic policy.

2.3 Intrinsic Rewards in RL

Intrinsic rewards are internal signals used to motivate agents in reinforcement learning, beyond the external rewards provided by the environment. These rewards are designed to encourage exploration, curiosity, or other behaviors that

[5]The squashing function. The tanh in the SAC policy ensures that actions are bounded to a finite range. It also changes the distribution: before the tanh the SAC policy is a factored Gaussian-like the other algorithms' policies, but after the tanh it is not. (You can still compute the log-probabilities of actions in closed form)

may improve learning efficiency and performance. Below are some of the state-of-the-art intrinsic reward mechanisms:

Curiosity

Curiosity-driven rewards are based on the idea that agents are motivated to explore and learn about their environment. The intuition is that by rewarding the agent for encountering novel or unexpected outcomes, it will be encouraged to explore more thoroughly. This approach leverages the agent's inherent desire to reduce uncertainty and gain knowledge about its surroundings. Curiosity rewards are typically generated by measuring the prediction error of the agent's internal model. When the agent encounters a state or outcome that it cannot predict accurately, it receives a higher reward. This prediction error serves as an intrinsic motivation for the agent to seek out and learn from new and unfamiliar experiences.

One common approach to implementing curiosity-driven exploration is using an Intrinsic Curiosity Module (ICM). This module consists of two primary components:
1. **Forward Model:** A neural network that predicts the next state S_{t+1} given the current state S_t and action A_t.
2. **Inverse Model:** A neural network that predicts the action A_t taken to transition from state S_t to S_{t+1}. The forward model is used to calculate the prediction error, which forms the basis of the curiosity reward. The process can be summarized as follows:

- **Prediction:** The forward model predicts the next state $\hat{S}_{t+1}$ based on the current state S_t and action A_t.
- **Error Calculation:** The curiosity reward is computed as the difference (prediction error) between the predicted next state $\hat{S}_{t+1}$ and the actual next state S_{t+1}:

$$\text{Curiosity Reward} = \|S_{t+1} - \hat{S}_{t+1}\|^2$$

- **Reward Assignment:** This prediction error is used as an intrinsic reward signal, which is added to the extrinsic rewards provided by the environment.

By continually seeking out states with higher prediction errors, the agent effectively explores the environment, learns about its structure, and reduces uncertainty in its internal model. Over time, this exploration helps the agent discover the optimal approach to the goal, guided by both intrinsic (curiosity) and extrinsic (goal-oriented) rewards. This approach tries to balance between exploration and exploitation, improving the agent's overall learning efficiency and performance.

Empowerment

Empowerment-based rewards are based on the idea that agents are motivated to increase their control over the environment. The intuition is that by rewarding the agent for actions that increase its ability to influence future states, it will be motivated to seek out and exploit opportunities for control. Empowerment rewards are given for actions that increase the agent's ability to affect its environment. This can help the agent develop strategies that maximize its influence over future states. Empowerment can be measured using information-theoretic approaches, such as calculating the mutual information between the agent's actions and the resulting states. The agent is rewarded for actions that increase this mutual information.

Competence

Competence-based rewards are inspired by the idea that agents are motivated to improve their skills and achieve goals. The intuition is that by rewarding the agent for achieving goals or improving its performance, it will be motivated to learn and master new tasks. Competence rewards are given when the agent successfully completes a task or improves its performance on a task. This can help the agent focus on learning and mastering specific skills. Competence can be measured by setting specific goals for the agent and rewarding it when it achieves these goals. For example, in a robotic manipulation task, the agent might be rewarded for successfully throwing objects upwards.

Playfulness

Playfulness-based rewards are inspired by the idea that agents are motivated to engage in activities that are inherently enjoyable or interesting. The intuition is that by rewarding the agent for engaging in playful behavior, it will be motivated to explore and learn in a more relaxed and creative manner. Playfulness rewards are given for actions that are enjoyable or interesting, even if they don't directly contribute to achieving external goals. This can help the agent develop a more flexible and creative approach to learning. Playfulness can be encouraged by designing environments that are rich and varied, and by providing rewards for actions that are novel or interesting. For example, in a game environment, the agent might be rewarded for discovering hidden features or trying out new strategies.

There are a lot of approaches and ways to develop and implement intrinsic rewards. We covered some of the most common intrinsic rewards. In the following, we will present morphological computation, from which we will later derive the intrinsic reward for our approach.

2.4 Morphological Computation (MC)

Morphological computation refers to the exploitation of the information gained from physical form and dynamics. This concept emphasizes the role of the body's physical characteristics in optimizing control and enhancing the efficiency of performing tasks.

Measuring Morphological Computation

There are various ways and formulas to measure morphological computation (MC), yet we used one of the most common MC formulations. This formula will be the baseline for the MC model that we will present in later sections:

$$D_{KL}\left(p(s' \mid s, a) \parallel p(s' \mid a)\right) \tag{2.23}$$

In the following, we will explain this formula in detail:

- s' represents the next state.
- s represents the current state.
- a represents the action taken.
- $p(s' \mid s, a)$ is the probability of reaching the next state s' given the current state s and the action a.
- $p(s' \mid a)$ is the probability of reaching the next state s' given only the action a.

Assume that the world state at a given time step does not influence its state at the next time step when conditioned only on the action taken. This assumption is reflected in the second half of the formula where we condition the next state s' only on the current action a. The first part of the formula, $p(s' \mid s, a)$, represents the probability of landing in the next state s' given the current state s and the action a that was taken. This probability distribution accounts for the complete information available about the current state and the action. Again the second part of the formula, $p(s' \mid a)$, represents the probability of landing in the next state s' given only the action a, ignoring the current state. This distribution abstracts away the specific current state, focusing solely on the action.

We then apply the Kullback-Leibler Divergence (KLD), also known as relative entropy, which is a measure of how one probability distribution diverges from a second, expected probability distribution. KLD quantifies the amount of information lost when approximating one distribution with another. Specifically, in our context, it measures how much additional information the current state s provides about the next state s' given the action a. The use of KLD in this context allows us to quantify the morphological computation by comparing the information content in $p(s' \mid s, a)$ and $p(s' \mid a)$. A lower KLD value indicates that the action a alone is

sufficient to predict the next state s' effectively, implying low morphological computation where the body's dynamics do not play a significant role in determining the next state.

For two probability distributions P and Q defined on the same probability space, the KLD from the two is given by:[6]

$$D_{KL}(P \parallel Q) = \sum_{x \in \mathcal{X}} P(x) \log \left(\frac{P(x)}{Q(x)} \right)$$

- **Positive Definiteness:** $D_{KL}(P \parallel Q) \geq 0$, with equality if and only if $P = Q$ almost everywhere. This means KLD is always non-negative and zero only when the two distributions are identical.
- **Asymmetry:** $D_{KL}(P \parallel Q) \neq D_{KL}(Q \parallel P)$. This property indicates that the divergence from P to Q is not the same as from Q to P.

KLD measures the difference between two distributions. The larger the difference, the greater the KLD. This KLD quantifies how much information about the next state is lost when considering only the action and ignoring the current state. A high KLD value indicates a significant difference between the two predicted distributions. This suggests that the current state provides crucial information for predicting the next state, beyond what can be inferred from the action alone. In other words, the agent's morphology plays a significant role in determining the next state. A high KLD means the dynamics are highly dependent on the state, making the state information critical for accurate predictions. Conversely, a low KLD value implies that the next state can be predicted accurately based solely on the action, with little additional information provided by the current state. This indicates that the agent's actions are the primary determinants of state transitions, and the morphology has a lesser impact. A low KLD suggests that the system's behavior is more influenced by the actions than the specific state, indicating a form of morphological simplicity. We will see in later sections how we will make use of the value received from the KLD and how to transform it into a meaningful reward the agent can exploit.

[6]KLD can also be continuous: $D_{KL}(P \parallel Q) = \int_{-\infty}^{\infty} p(x) \log \left(\frac{p(x)}{q(x)} \right) dx$, where $p(x)$ and $q(x)$ are the probability density functions of P and Q respectively.

Chapter 3

Training RL with MC

Our approach integrates morphological computation into the Soft Actor-Critic (SAC) algorithm. We augment the SAC algorithm with two additional neural networks (which, as illustrated in Section 2.1, outputs a mean and standard deviation that we aim to train so that the resulting distribution approximates the actual data distribution as closely as possible). For the implementation of the MC part, we define the Transition Model and the Action-Only Model. These models predict the next state of the environment, enabling us to capture the influence of the agent's morphology.

First, let's start with the transition model. This model predicts the next state given the current state and action, capturing the combined effect of the agent's morphology and the action taken. Formally, let: $s_t \in S$ be the current state, $a_t \in A$ be the action taken, and $\hat{s}_{t+1}$ be the predicted next state. The Transition Model can be described as:

$$\hat{s}_{t+1} = f_T(s_t, a_t)$$

where f_T is the neural network representing the transition model. The architecture comprises multiple layers of fully connected neural networks. The input to the model is a concatenation of the current state s_t and the action a_t. The output is a distribution over the next state, represented by a mean and standard deviation.

- **Input:** Concatenation of s_t and a_t
- **Hidden Layers:** Fully connected layers with ReLU activations
- **Output:** Mean and standard deviation of the predicted next state distribution

$$\text{TransitionModel}(s_t, a_t) \rightarrow \text{Normal}(\mu_{t+1}, \sigma_{t+1})$$

where μ_{t+1} and σ_{t+1} are the mean and standard deviation of the predicted next state distribution.

The Action-Only Model predicts the next state based solely on the action taken, ignoring the current state. This model captures the influence of actions in isolation. Formally, let:

- $a_t \in A$ be the action taken,
- $\hat{s}_{t+1}(a)$ be the predicted next state based on the action only.

The Action-Only Model can be described as:

$$\hat{s}_{t+1}(a) = f_A(a_t)$$

where f_A is the neural network representing the Action-Only Model. The architecture of the Action-Only Model is similar to that of the Transition Model but takes only the action a_t as input.

- **Input:** Action a_t
- **Hidden Layers:** Fully connected layers with ReLU activations
- **Output:** Mean and standard deviation of the predicted next state distribution

The model is defined as follows:

$$\text{ActionOnlyModel}(a_t) \rightarrow \text{Normal}(\mu^a_{t+1}, \sigma^a_{t+1})$$

where $\mu^a_{t+1}, \sigma^a_{t+1}$ are the mean and standard deviation of the predicted next state distribution based on the action.

3.1 Updating MC Networks

Once these models have made their predictions, our goal is to align these predictions as closely as possible with the actual observed next states. Since both models are neural networks, we achieve this alignment by computing a loss function, which measures the difference between the predicted outputs and the actual outcomes. The loss function provides a quantitative assessment of how far off the predictions are from the true states. A lower loss indicates that the model's predictions are closer to the actual outcomes, suggesting a better understanding of the underlying state transition dynamics. To minimize this loss, we use the process of backpropagation. This involves calculating the gradient of the loss concerning the model's weights and adjusting these weights to reduce the loss. By iteratively updating the weights through backpropagation, the models progressively learn to make more accurate predictions. This training process continues until the models effectively approximate the true probability distributions of the next states based on the given inputs. The result is a pair of trained neural networks capable of making reliable state predictions, which are crucial for the MC.

The loss for the Transition Model and Action-Only Model is based on the negative log-likelihood of the true next state s_{t+1} given the predicted next state distribution. This is equivalent to taking the log of the predicted probability for the true class, with a negative sign, indicating that we want to minimize this value during training. Translating it to both models results in the following equations:

$$L_T = -\log p(s_{t+1} \mid f_t(s_t, a_t)) \tag{3.1}$$

$$L_A = -\log p(s_{t+1} \mid f_a(a_t)) \tag{3.2}$$

In the following, we are presenting how the training procedure works:

1. **Sample Transitions:** Collect samples of state transitions (s_t, a_t, s_{t+1}) from the environment using the current policy.
2. **Compute Predictions:** Use the transition model and the action model to predict the next state separately
3. **Calculate Losses:** Compute the losses L_T and L_A using the true next state s_{t+1} and the predicted next states.
4. **Update Networks:** Perform gradient descent to update the parameters of the Transition Model and Action-Only Model using their respective losses.
5. **Policy Update:** Update the SAC-MC[1] policy network using the combined reward r_{total}.

3.2 Normalizing the Intrinsic Reward

Intrinsic rewards as mentioned before play a crucial role in reinforcing specific behaviors in reinforcement learning algorithms. However, intrinsic rewards can sometimes become excessively large, leading to instability in training. This instability arises because the KL divergence values can vary significantly and occasionally include extreme outliers, which can skew the reward distribution. (See Figure 3.1) Therefore, it is essential to implement a normalization strategy to manage the scale of these rewards effectively. In our approach, we focus on normalizing the intrinsic reward derived from the KL divergence to ensure it is consistently scaled between a fixed range. This normalization process is vital for maintaining a stable and efficient learning process within the Soft Actor-Critic (SAC) framework. By doing so, we can prevent the occasional spikes in intrinsic reward from dominating the training process and causing erratic learning behavior.

To achieve this, we use a more robust normalization technique that mitigates the impact of outliers. Specifically, we calculate the xth percentile of the intrinsic reward history ($x \in \{90, 91, ..., 99\}$), using this value as the upper bound for normalization. This method helps to provide a stable reward scale, which is less sensitive

[1]We will call the augmented SAC with Morphological Computation SAC-MC.

to outliers compared to using the mean or the absolute maximum. Additionally, we consider using the median for normalization as an alternative approach, further enhancing robustness.

By scaling the intrinsic rewards within a given range and clamping extreme values, our normalization strategy ensures that the rewards are kept within a manageable range. This consistent scaling improves the overall stability and efficiency of the learning process, facilitating a more reliable integration of intrinsic rewards into the SAC algorithm.

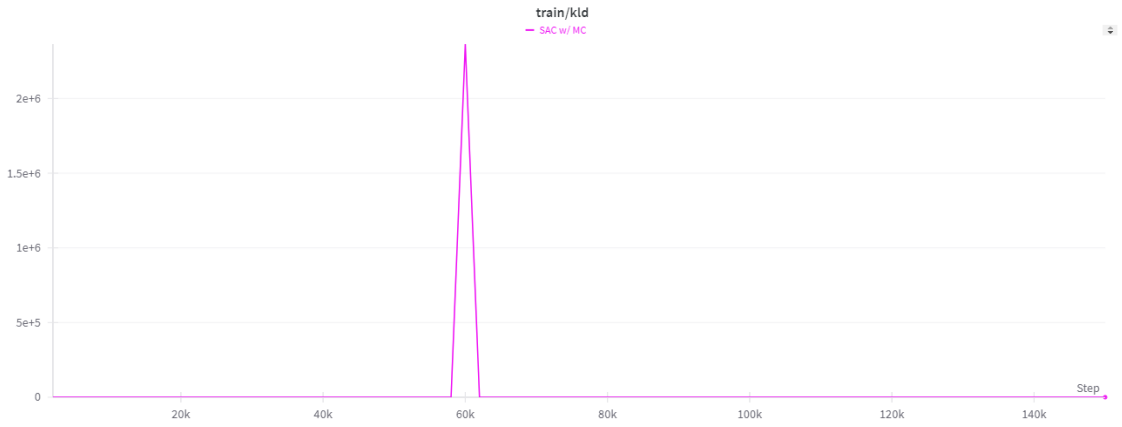


Figure 3.1: The occurrence of a high Morphological Computation value, as there is no upper bound for the KL Divergence.

We maintain a history of the intrinsic rewards up to a specified length. This history is used to compute the normalization parameters. We calculate the xth percentile as mentioned before as the maximum value[2] ($\max x_p$) and the minimum value of the reward history. This percentile is used as the upper bound, respectively, for normalization.

$$max = \text{percentile}(\text{rewards}, x)$$

We will see in Chapter 4 that the 94th and 98th percentiles are the best for normalizing the intrinsic rewards of the two tasks studied, respectively. The normalized reward is scaled and shifted to fit within the desired range. If the normalized reward exceeds max value or is below min value, it is clamped to again fit into the range, respectively. Otherwise, it is linearly scaled:

[2]We conducted several experiments with different percentile values to determine the optimal one. Two key challenges were considered:

- Selecting a sufficiently high value to retain essential information about the MC.
- Selecting a sufficiently low value to avoid including outliers that could undesirably skew the resulting reward.

$$r_{scaled} = \left(\frac{r - \min}{\max - \min} \right)$$

By implementing this robust normalization technique, we ensure that the intrinsic reward r_{mc} remains within a consistent and manageable range, improving the stability and efficiency of the learning process. This approach effectively mitigates the impact of outliers and provides a reliable method for integrating the intrinsic reward (MC) with the SAC algorithm. Note that the reward can be shifted to match the reward range of the studied task.

3.3 Using MC as Intrinsic Reward

To incorporate morphological computation into the SAC framework, we introduce an intrinsic reward based on the Kullback-Leibler (KL) divergence between the outputs of the Transition Model and the Action-Only Model. The KL divergence quantifies the difference between the predicted next state distributions of the two models. Formally, the KL divergence is given by:

$$KL(\mathcal{N}(\mu_{t+1}, \sigma_{t+1}) \parallel \mathcal{N}(\mu^a_{t+1}, \sigma^a_{t+1}))$$

This KL divergence is transformed into an intrinsic reward r_{mc}, which is as mentioned previously scaled and shifted into a given range to match the extrinsic reward shape and to ensure stability during training. The intrinsic reward is combined with the extrinsic reward from the SAC algorithm.[3]

The combined reward used for training the agent is a weighted sum of the extrinsic reward r_{ext} from the SAC algorithm and the intrinsic reward r_{mc} from the morphological computation:

$$r_{total} = (1 - \alpha) r_{ext} + \alpha r_{mc}. \tag{3.3}$$

[3]In the context of SAC (Soft Actor-Critic), we describe the reward as extrinsic, which is notionally accurate. Although the SAC paper itself does not classify the entropy term explicitly as an intrinsic reward, it does play a role in balancing exploration and exploitation. The entropy term is incorporated into SAC's objective function to encourage exploration, which aligns with the idea of intrinsic reward. Thus, while the authors did not frame it as such, the entropy term can indeed be interpreted as an intrinsic reward.

Chapter 4

Experiments, Results and Discussion

In this chapter, we present the results of our experiments conducted to evaluate the performance of the Soft Actor-Critic (SAC) algorithm and our modified SAC algorithm incorporating Morphological Computation (SAC-MC) with varying percentages of MC reward (Controlled by the factorα presented in the section 3.3). The experiments were conducted on the Fetch Push and the Ant Maze environments, these benchmark tasks are commonly used to assess the capabilities of reinforcement learning algorithms. We will start with the Fetch Push task, present our results, and analyze these, after that, we will do the same for the Ant Maze Task.

4.1 Fetch Push Environment

To evaluate the performance of our approach, we conducted a series of experiments using the FetchPush environment. We tested the baseline SAC algorithm and compared it with our modified SAC-MC algorithm at different levels of morphological computation. In this experiment, we utilize the environment introduced in "Multi-Goal Reinforcement Learning: Challenging Robotics Environments and Request for Research". The task involves a 7-DoF Fetch Mobile Manipulator equipped with a two-fingered parallel gripper. The goal is for the manipulator to move a block to a target position on a table by pushing it with its gripper. The robot operates with its gripper locked in a closed configuration, and it must maintain the block in the target position indefinitely. An illustration is shown in 4.1

Control and Action Space

The robot is controlled with a frequency of $f = 25$ Hz, applying the same action for 20 simulator steps (with a time step of $dt = 0.002$ s) before returning control. The action space is defined as a Box(-1.0, 1.0, (4,), float32), representing the Cartesian displacements dx, dy, dz of the end effector and a control for the gripper's opening

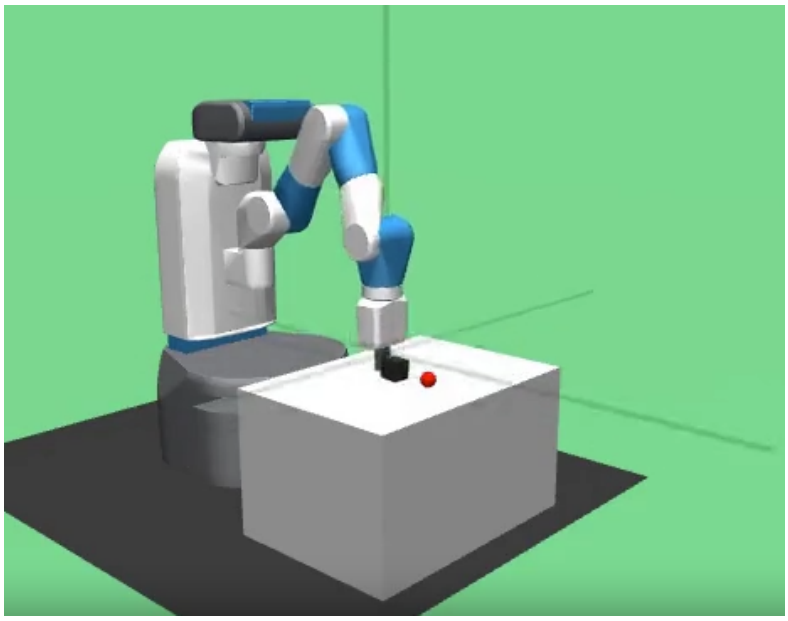


Figure 4.1: Fetch Push Experimentation Environment.

and closing.

Num	Action	Control Min	Control Max
1	Displacement in x direction dx	-1	1
2	Displacement in y direction dy	-1	1
3	Displacement in z direction dz	-1	1
4	Gripper control	-1	1

Table 4.1: Action space details

Observation Space

The observation space consists of a dictionary with three keys: `observation`, `desired_goal`, and `achieved_goal`.

- **observation**: An array of shapes (25,) containing kinematic information about the block and gripper.
- **desired_goal**: A 3-dimensional array representing the target block position $[x, y, z]$.
- **achieved_goal**: A 3-dimensional array representing the current block position $[x, y, z]$.

The `observation` includes details like the end effector's position, the block's position and rotation, and velocities relative to the gripper.

Rewards

The environment supports both sparse and dense reward functions, yet our tests only implied sparse rewards:

- **Sparse Reward**: Returns -1 if the block hasn't reached the target and 0 if it has (within a Euclidean distance of 0.05 m).
- **Dense Reward**: Returns the negative Euclidean distance between the achieved goal and the desired goal.

The episode is truncated after a default of 50 timesteps, although this can be modified, we stayed by the default settings. So the reward switches to 0 when the object reaches the red dot and returns -1 otherwise. This is shown in 4.2.

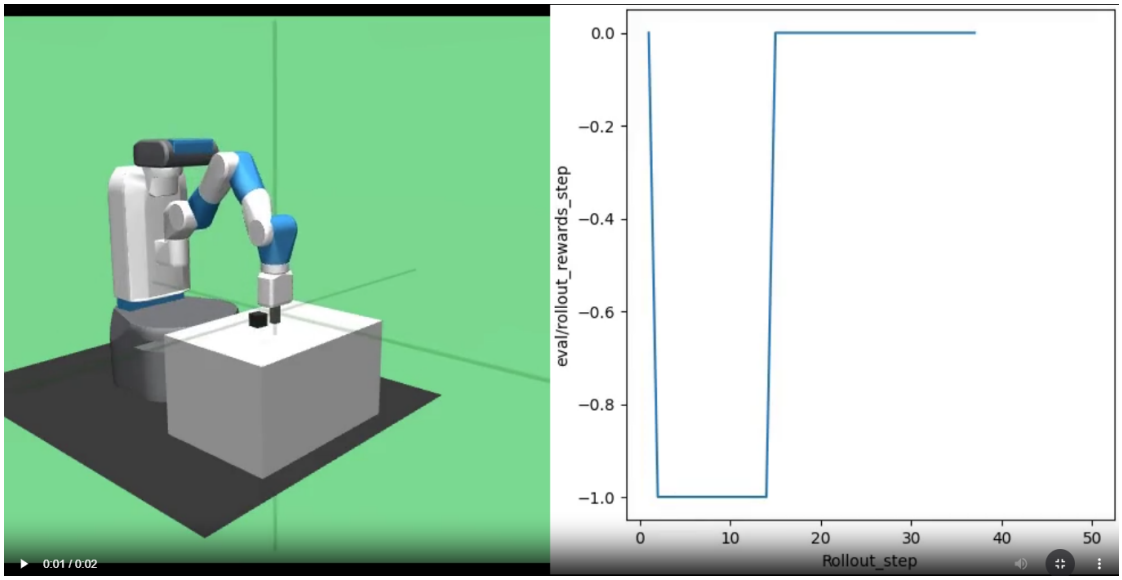


Figure 4.2: FetchPush Sparse Reward

4.1.1 Evaluation Metrics

In our experiments, all variants were trained for 150,000 steps. The primary metric used to evaluate the performance of the agent was the success rate. The success rate is determined after training, by evaluating the agent's ability to achieve the desired goals.

- **Success Rate**: This metric measures the percentage of trials in which the agent successfully completed the task. It provides a direct indication of the agent's performance after the learning process.

Besides the success rate, we also considered additional metrics to provide a more comprehensive evaluation of the agent's performance:

- **Mean Reward**: This metric represents the average reward obtained by the agent over a set of episodes. It helps in understanding the overall performance and the learning progress of the agent.
- **Losses for Actor-Critic**: These losses indicate how well the agent's policy (actor) and value function (critic) are being optimized during training. Monitoring these losses helps in diagnosing any issues in the training process.
- **Transition and Action Models Losses**: These losses measure how accurately the transition and action models predict the next state given the current state and action. They are crucial for understanding the dynamics of the environment and the agent's interaction with it.
- **(Extrinsic) SAC Reward**: This metric pertains to the Soft Actor-Critic (SAC) algorithm and measures the reward associated with the SAC's policies.
- **(Intrinsic) MC Reward**: The Morphological Computation (MC) reward represents the reward calculated using KLD between the two MC models.

There are more metrics that we considered when training the RL agent. By utilizing these metrics, we ensured a comprehensive evaluation of the agent's performance and learning progress throughout the training process.

4.1.2 Experimental Conditions

We tested the SAC algorithm and SAC-MC algorithm under the following conditions:

- **Baseline SAC**: The standard SAC algorithm without morphological computation.
- **SAC-MC**: The SAC algorithm with morphological computation integrated at various levels. These levels were implemented as demonstrated in the last Chapter and controlled by the factor α, for which we tested the following values: (0.2, 0.4, 0.6, 0.8, and 1.0). In other words, these values represent the percentage of the reward that is assigned to the morphological computation (MC) reward. For example:
  - **0.2**: 20% of the combined reward is assigned to the MC reward and the remaining 80% is assigned to the SAC reward.
  - **1.0**: 100% the whole reward is assigned to the MC reward.

4.1.3 Quantitative Evaluation

For each variant, including the baseline SAC, we conducted six cycles of training, averaging the results to ensure consistency and reliability.[1] Each training cycle consisted of 150,000 steps, during which we measured key performance metrics such as the success rate, mean reward, and variance of success rate. These metrics provide a comprehensive overview of the algorithm's performance. The baseline Soft Actor-Critic (SAC) algorithm served as a benchmark for our study, providing a point of comparison against our proposed SAC-MC approach.

Metric	Value
Mean of Success Rate	0.52
Variance of Success Rate	0.2
Mean Reward	-31.3

Table 4.2: Baseline SAC Performance

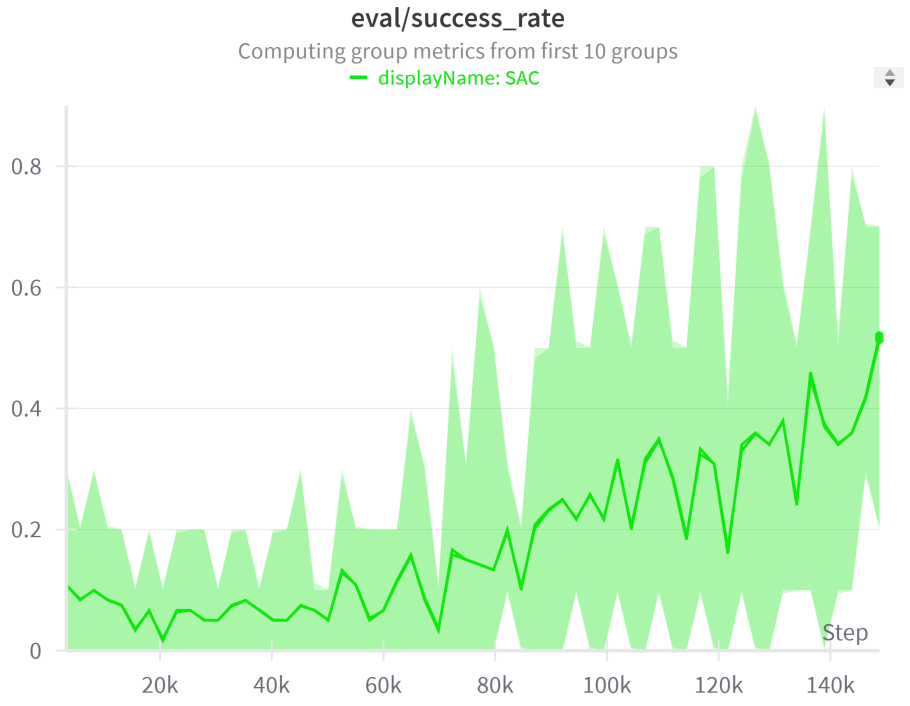


Figure 4.3: SAC Success Rate

[1]The training cycles were performed on a system featuring an Intel Core i9-13900K processor with 36 MB cache, an Nvidia GeForce RTX 3060 GPU with 12 GB of GDDR6 memory, and 64 GB of RAM. This setup provides the necessary computational power and memory capacity for handling intensive tasks and large datasets efficiently. Each cycle needed about 35 minutes

SAC-MC Performance

The best-performing model in our study was SAC-MC-0.2, which demonstrated a significant improvement with an MC reward part of 20% or $\alpha = 0.2$. This model outperformed the baseline SAC, indicating the effectiveness of the proposed modifications. The details of the SAC-MC-0.2 model's performance, alongside the baseline SAC results, are presented in Figure 4.4.

Metric	Value
Mean of Success Rate	0.75
Variance of Success Rate	0.2
Mean Reward	-23

Table 4.3: SAC-MC with $\alpha = 0.2$

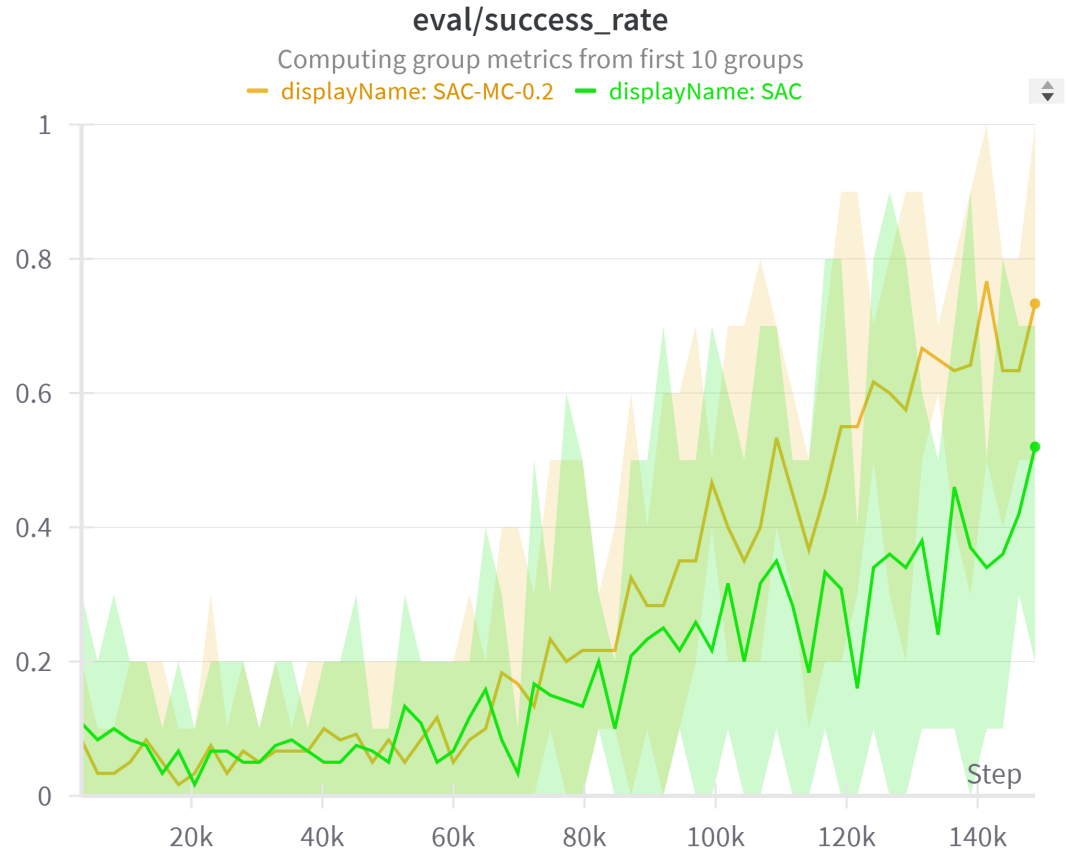


Figure 4.4: SAC-MC Success Rate with $\alpha = 0.2$

Empirical evidence demonstrates a clear advantage of the SAC-MC algorithm over the baseline SAC algorithm in terms of success rate. Specifically, the SAC-MC

algorithm shows a notable performance improvement when morphological computation is applied, particularly at the 20% level of the total reward. At this level, the SAC-MC algorithm achieves a significantly higher success rate compared to the baseline SAC. However, as the level of morphological computation increases to 40% and 60%, the success rate of the SAC-MC algorithm becomes comparable to that of the baseline SAC algorithm Figure4.5. This suggests that while a moderate application of morphological computation can enhance learning, excessive levels may not provide additional benefits and might even lead to performance stagnation. The trend observed in average reward metrics aligns with the success rate findings. The SAC-MC algorithm consistently yields higher average rewards when

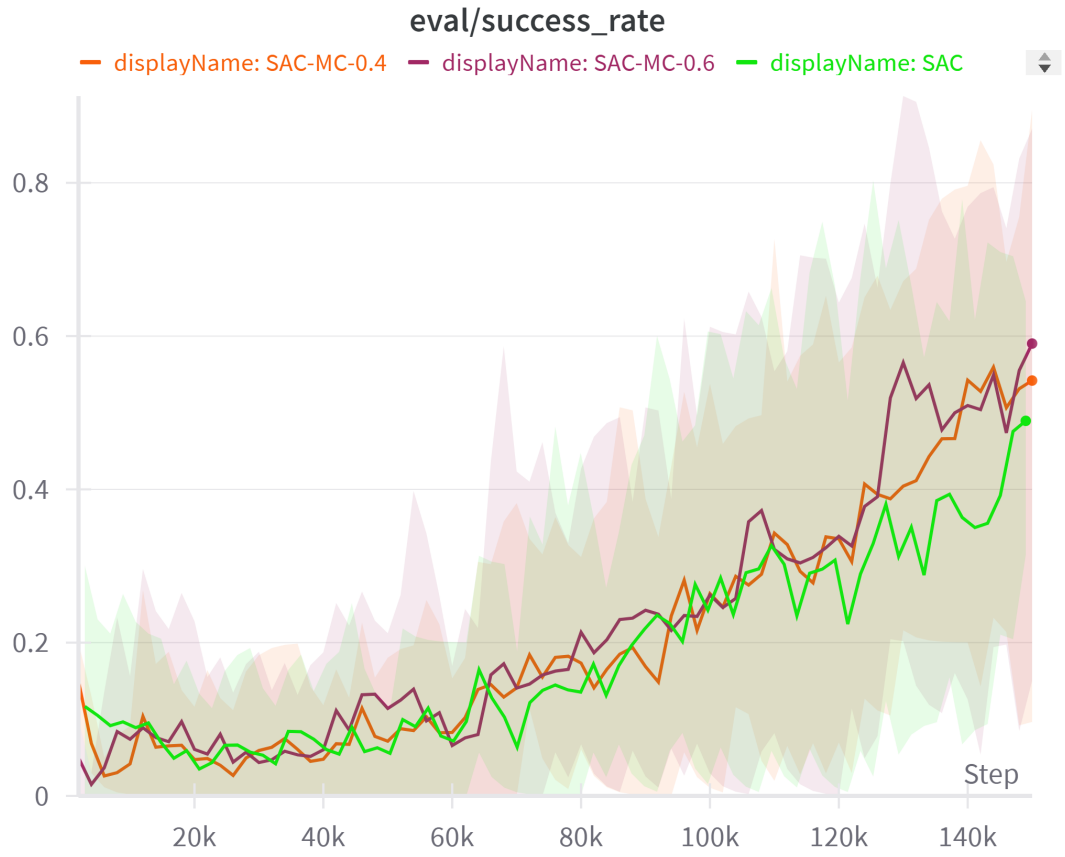


Figure 4.5: Success Rate of SAC-MC with $\alpha = 0.4$ *and* 0.6

using 20% morphological computation, outperforming the baseline SAC algorithm. This indicates that the agent is able to derive more effective policies and achieve superior results when a moderate amount of morphological computation is utilized. Conversely, at 40% and 60% morphological computation, the performance in terms of average reward converges with that of the baseline SAC with a mean reward of 28.81 and 28.96 respectively. This suggests a diminishing return on reward improvement with increasing levels of morphological computation beyond a certain

point. Furthermore, at 80% and 100% morphological computation, the Success rate and the average rewards decrease, indicating that excessive levels of morphological computation hinder the learning process rather than enhance it. In Figure 4.6. The success rates observed for both SAC-MC 0.8 & 1.0 models are relatively low. This outcome is not unexpected, as the intrinsic rewards (MC) have a significant influence on the agents' behavior. In these models, the intrinsic rewards are designed to prioritize actions that maximize MC, rather than directly optimizing for extrinsic environmental rewards. Intrinsic rewards, such as those based on MC, are conceptually focused on encouraging exploration and self-improvement rather than exploiting the specific rewards provided by the environment. As a result, the models may not fully align with the environmental reward structure, leading to lower overall success rates in tasks where the optimal strategy is tightly coupled with the extrinsic rewards. This inherent divergence between intrinsic motivation and task-specific performance metrics can explain the observed performance in both models.

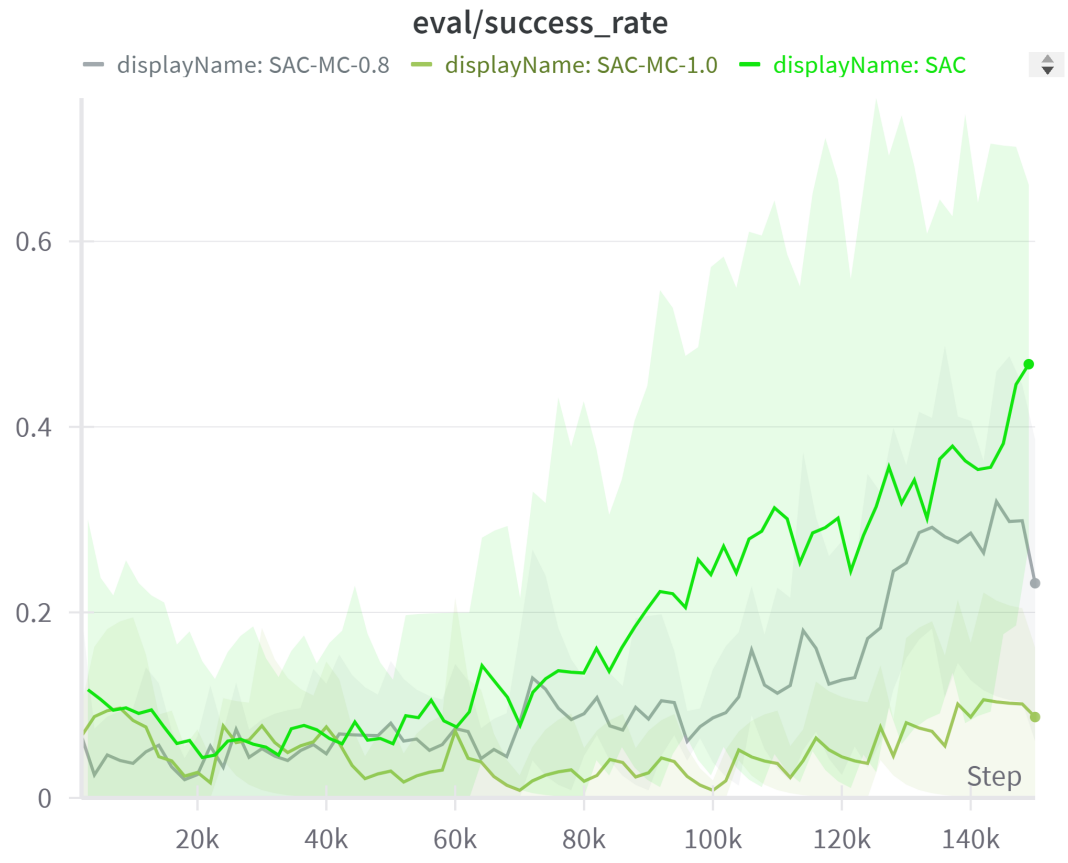


Figure 4.6: Success Rate of SAC-MC with $\alpha = 0.8$ *and* 1.0

4.1.4 Qualitative Evaluation

To comprehensively evaluate the agents' performance, it is essential to go beyond just the quantitative metrics. Analyzing the learned policies, particularly the driving behaviors exhibited by the agents, provides valuable insights. Specifically, we aim to examine the policy derived from our best-performing model after substantial training, as well as the policies of models with high Morphological Computation (MC) values. This analysis helps us understand how MC influences the actions taken by the agent and shapes the overall policy. To understand the effects of the intrinsic reward (morphological computation) on the agent's actions, we need to revisit the fundamental role of morphological computation. High morphological computation values indicate that the agent is incentivized to optimize its morphology, which refers to leveraging the information contained in its observations. As illustrated in Figure 4.7.[2] The agent consistently selects actions that maximize the intrinsic rewards across all models incorporating morphological computation (MC). This behavior highlights the influence of intrinsic rewards on the agent's decision-making process, driving it to prioritize actions that enhance its morphology and leverage the environment's physical dynamics. This intrinsic reward encourages the agent to possibly exploit the physical properties of the environment, such as momentum, transferable force, and speed, to enhance its performance.

Agent with high MC

Agents with high morphological computation (MC) or driven solely by MC tend to exhibit unusual behaviors. These actions, though seemingly bizarre, can provide valuable insights. For instance, the agent often exploits the environment's physics by using momentum or moving at full speed to interact with objects like a cube. The agent sometimes tries to push the cube into the ground, attempting to control or manipulate it in unconventional ways. In some cases, the agent even pushes the limits of the simulated environment's physics, trying to force the cube under the ground. These behaviors highlight the agent's exploration of the environment's dynamics and its attempt to maximize intrinsic rewards through the creative use of physical interactions. An example of the effect of high MC of the policy is presented in Figure 4.8.

[2]Note that the measure of the intrinsic reward is relative, a high intrinsic reward means, its related MC is larger than most of the intrinsic rewards that have been recorded but it does not state the magnitude of the MC itself.

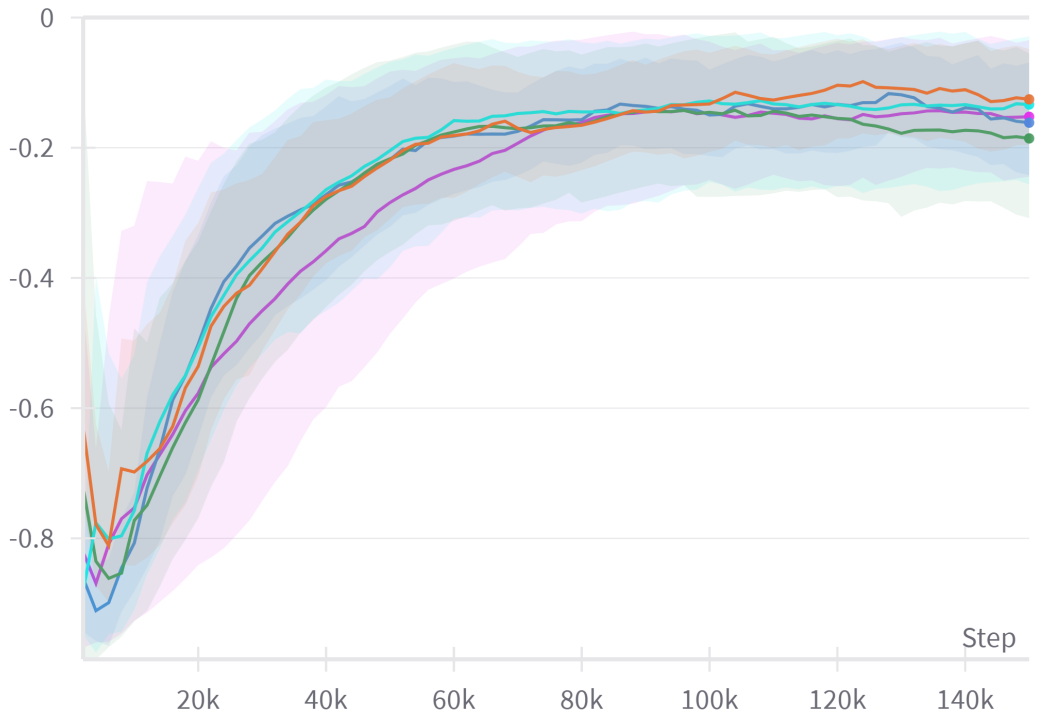
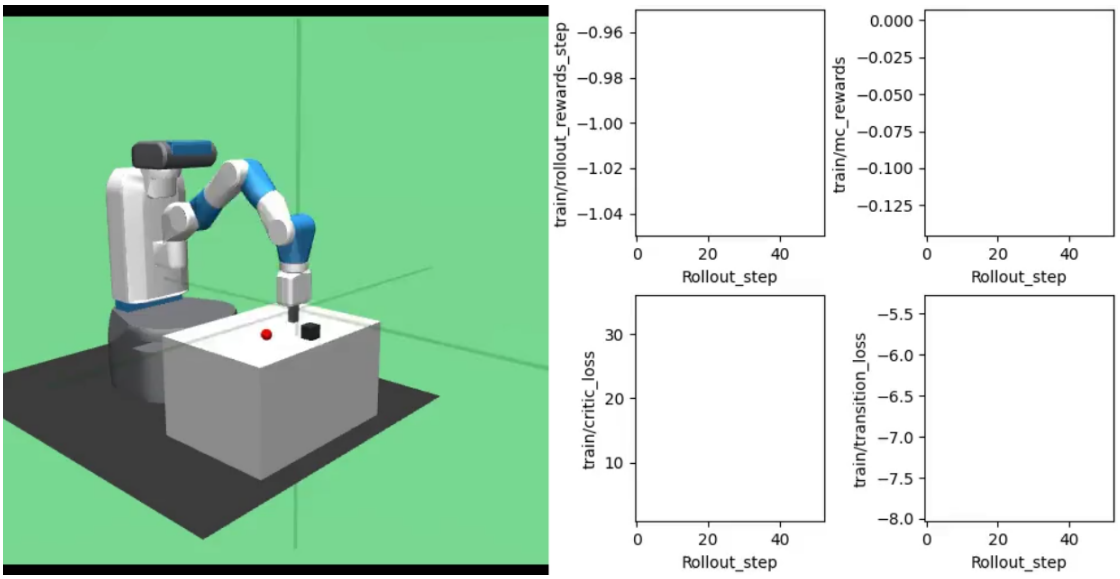
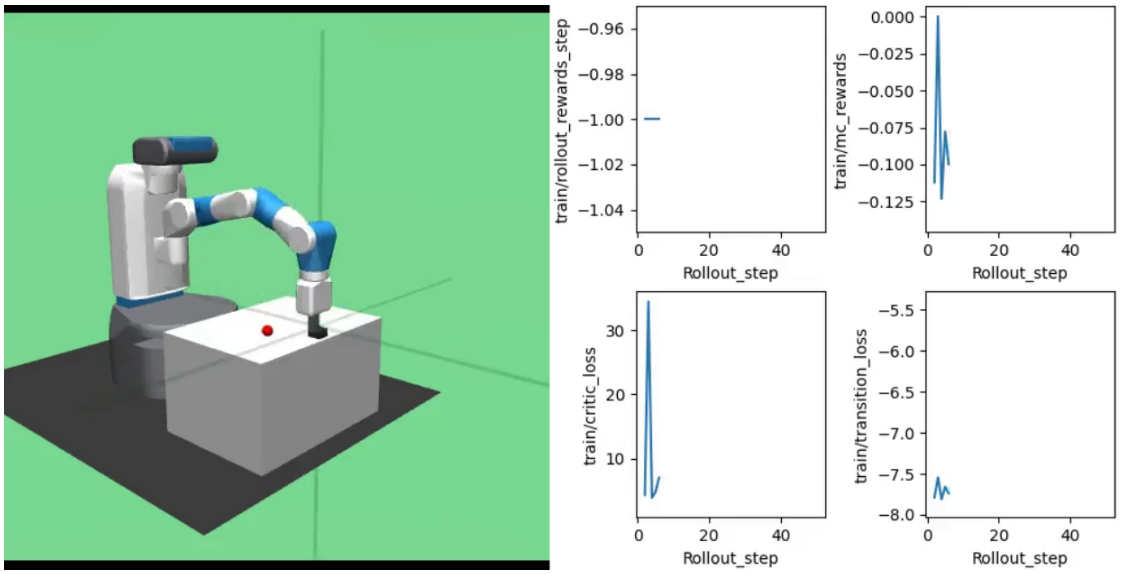


Figure 4.7: Development of Intrinsic Reward of the SAC-MC Algorithm during Training with Different Percentages of MC, $\alpha = \{0.2, 0.4, 0.6, 0.8, 1.0\}$

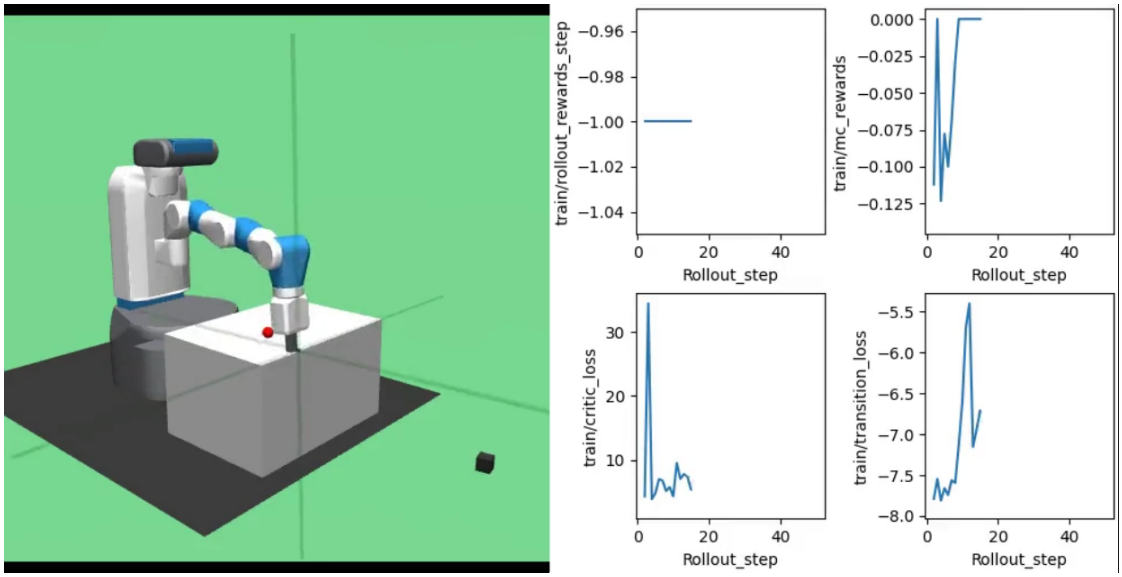


(a) Initial Position of the robotic arm and the cube

Figure 4.8: (Continued in the next page)



(b) The Arm accelerating towards the cube

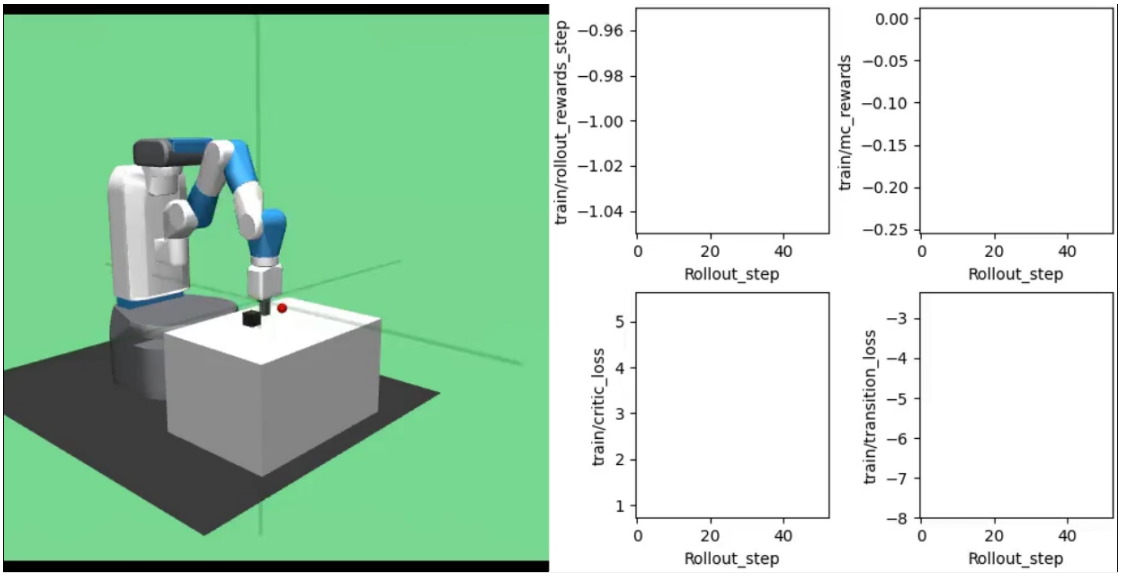


(c) Pushing the cube off the edge

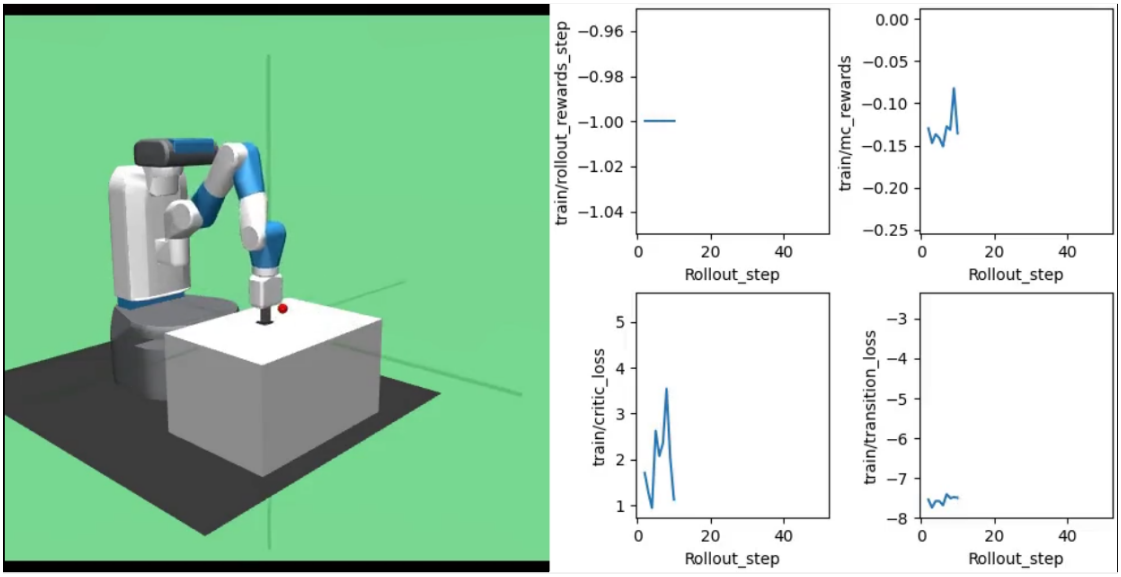
Figure 4.8: An example of high morphological computation influencing the development of the RL agent's policy, here the agent rapidly uses its arm to throw a cube over long distances. During this process, the intrinsic reward, which reflects the level of morphological computation, reaches its peak.

In the next Figure 4.9, the agent approaches the cube and pushes it into the table, eventually causing the cube to fly away. This sequence of actions corresponds with the moments when the intrinsic reward, specifically the Morphological Computation (MC) reward, is at its peak. This behavior frequently occurred, with the agent consistently pushing the cube into the table, causing it to fly away due to the applied force. Another example is shown in the subsequent figure, where the agent pushes the cube toward the table while attempting to control its position.

The agent exerts force to push the cube down into the table, resulting in the cube ultimately being displaced and flying away.

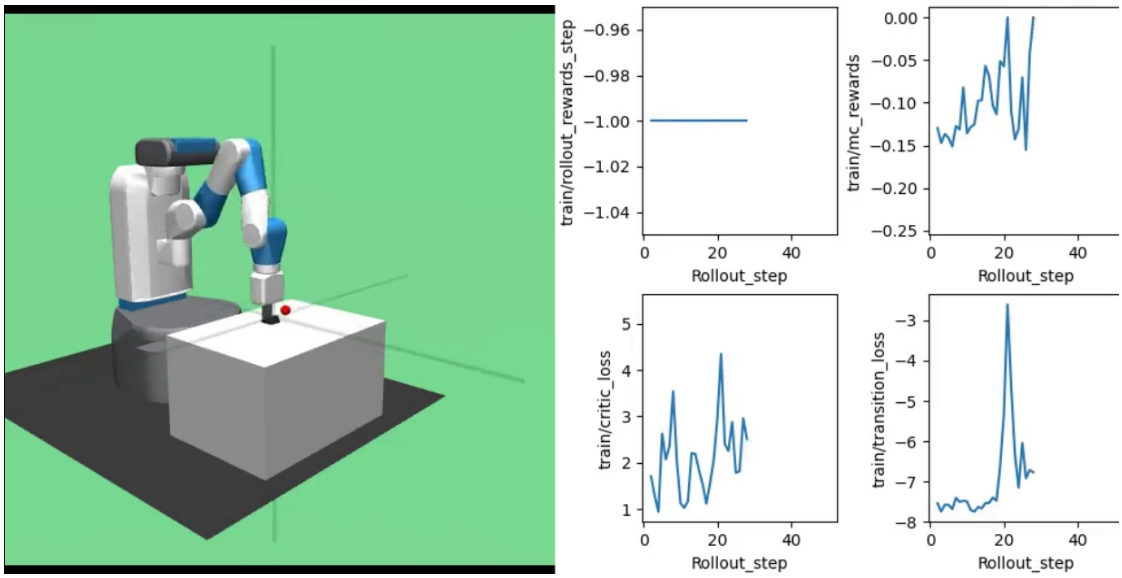


(a) Initial Position of the robotic arm and the cube

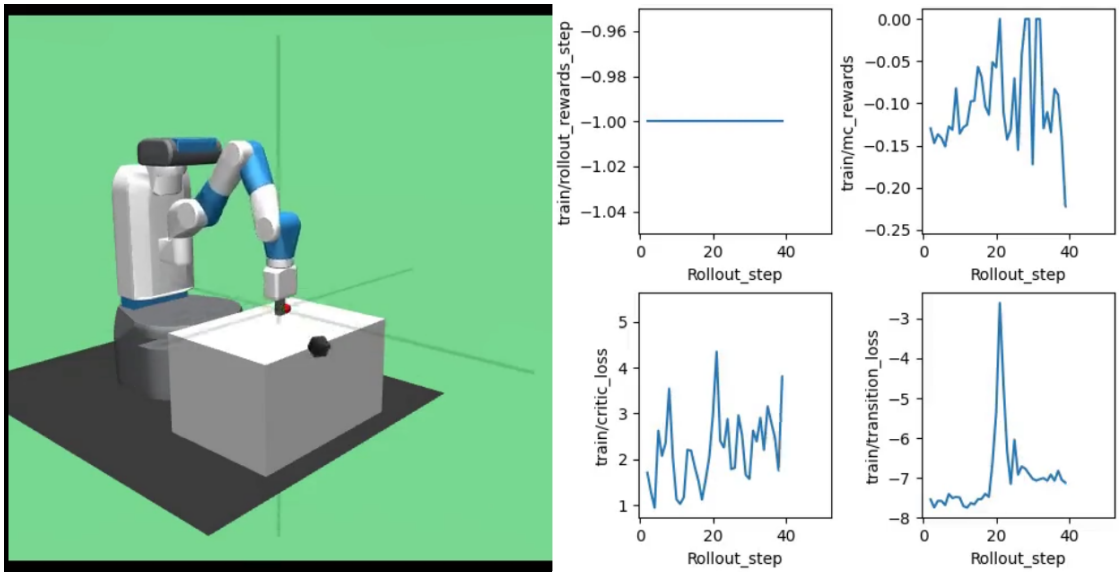


(b) The Arm pushing the cube into the table

Figure 4.9: (Continued in nest Page)



(c) Manipulating the cube under the table



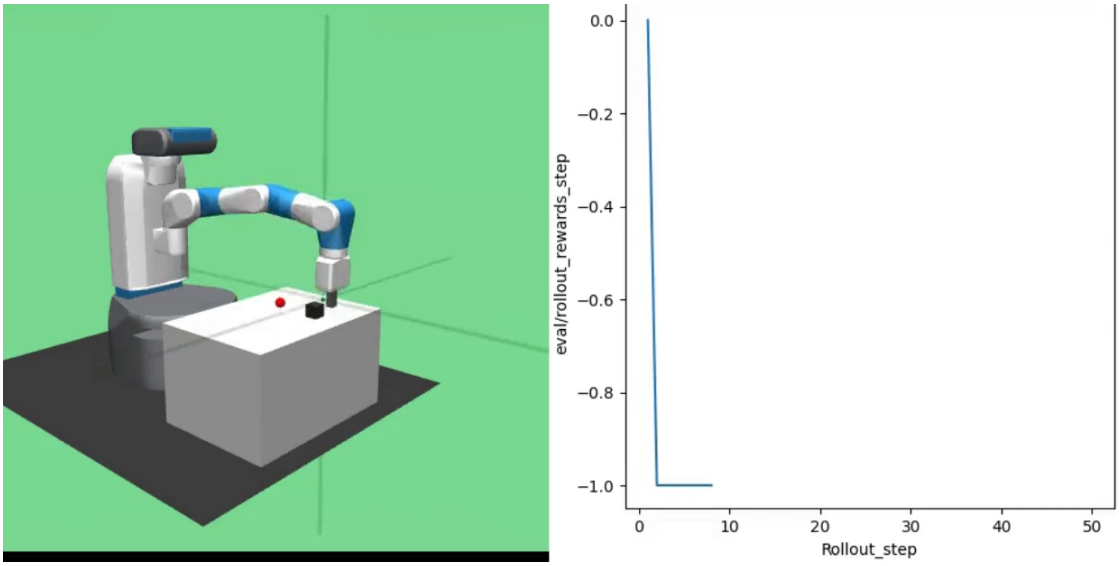
(d) Cube at the end flying away

Figure 4.9: Here, we observe another effect of morphological computation on the RL agent, as the robotic arm increases the applied force while attempting to push the cube into the table, ultimately causing the cube to fly off.

High Performance with MC

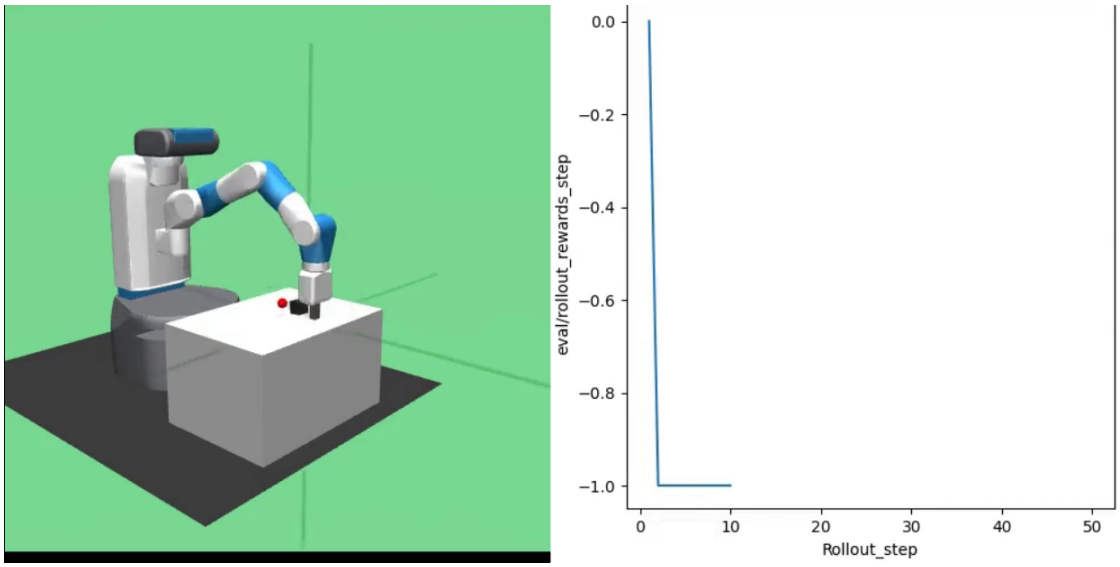
The agent demonstrates an ability to execute sophisticated actions beyond merely exploiting environmental rewards. Our best model learns to perform these actions rapidly, not only improving the learning efficiency but also leveraging the physics of the environment effectively. Specifically, the agent learns to apply calculated forces to push the cube accurately to a target location, such as a red dot, without the need to stay behind the cube while pushing it slowly, showcasing its proficiency

in utilizing physical principles to achieve desired outcomes. Although the agent does not possess an explicit understanding of concepts like physics, force, or pressure, it effectively harnesses these principles through the intrinsic Morphological Computation (MC) reward. This can be likened to a squirrel adjusting its body position while jumping to optimize aerodynamic properties; although the squirrel does not consciously understand aerodynamics, it intuitively exploits these principles to enhance its performance. The following figure illustrates the discussed above.

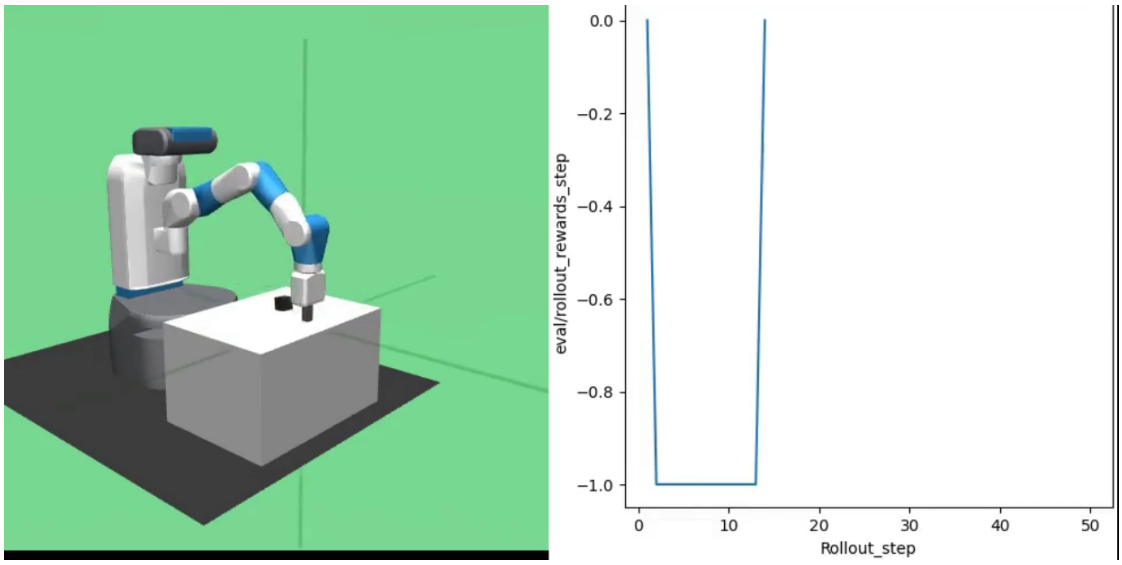


(a) Navigating to a good spot for pushing

Figure 4.10: (Continued on next page)



(b) Pushing the cube with high speed without guiding it



(c) Cube landing exactly at the desired position

Figure 4.10: The agent learned to leverage the observation data, utilizing both velocity and positional information to develop a policy for pushing the cube without needing to remain behind it continuously. As a result, the robotic arm applies a single touch, causing the cube to slide toward the goal.

We consider these results to be highly effective, as the agent achieves the goal within just over ten steps out of the fifty available. This performance is close to the theoretical limit of optimality for this task. Given the constraints and the current setup, it is challenging to conceive a scenario in which the agent could achieve the goal more efficiently than demonstrated in the discussed scenario.

4.2 Ant Maze Environment

The Ant Maze environment serves as a benchmark for evaluating reinforcement learning algorithms, particularly in tasks involving navigation and control. Here the goal is for the ant to reach the red ball. In this environment, the agent is represented by the Ant quadruped. The control frequency for the Ant is set at 20 Hz, with each simulation timestep lasting 0.01 seconds. The ant robot repeats the same action for five consecutive simulation steps, providing a realistic and challenging control scenario.

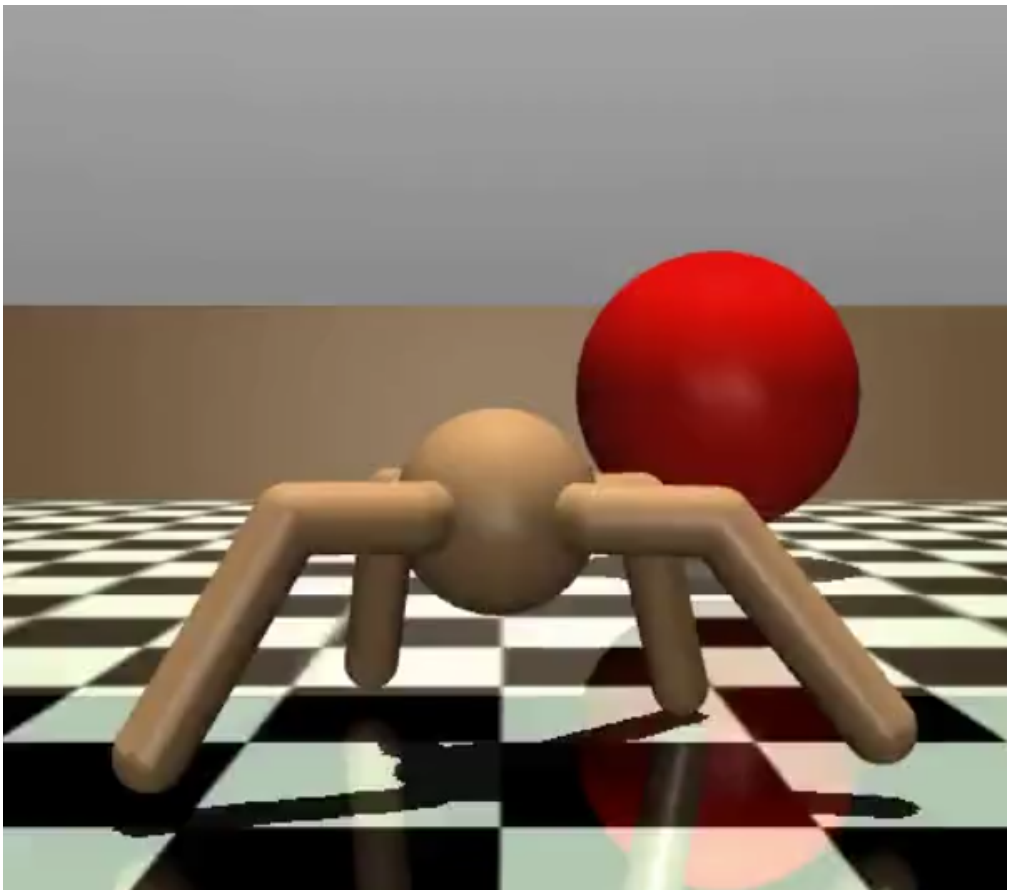


Figure 4.11: Ant Maze Experimentation Environment

Action Space

The action space in the Ant Maze environment is defined as a continuous range of torque values applied to the hinge joints of the ant robot. Each action corresponds to the torque applied to one of the eight joints, with values ranging from -1 to 1. This action space challenges the agent to control the complex dynamics of the quadruped robot effectively.

Observation Space

The observation space is goal-aware and provides a comprehensive representation of the robot's state. It includes positional values of the ant's body parts, their corresponding velocities, and information about the goal. The observation is structured as a dictionary with three keys:

- **observation:** Contains positional and velocity information of the ant's joints.
- **desired_goal:** Represents the final goal coordinates that the ant must reach.
- **achieved_goal:** Represents the current position of the ant relative to the goal, useful for goal-oriented learning algorithms.

The detailed structure of the observation space allows the agent to receive rich sensory feedback, facilitating the learning process.

Rewards

The environment supports both sparse and dense reward structures, in our setup we decided on the dense reward structure:

- **Sparse Reward:** The agent receives a reward of 1 when it reaches the final target position, defined as being within 0.5 meters of the goal. Otherwise, the reward is 0.
- **Dense Reward:** The reward is defined as an exponential function with a negative exponent of the distance.

Starting State and Episode Termination

The initial position of the ant and the goal location are selected within the maze, with some noise added to create variability in the starting conditions. Episodes can be truncated based on the maximum number of allowed timesteps or terminated when the ant reaches the goal, depending on the environment settings.

4.2.1 Quantitative Evaluation

For all variants, including the baseline Soft Actor-Critic (SAC) algorithm, we conducted four independent training cycles, each comprising 400,000 steps, to ensure consistency and reliability in our results. The outcomes from these cycles were averaged to provide a robust performance metric. The baseline SAC algorithm functioned as a benchmark in our study, facilitating a comparative analysis against our proposed SAC-MC approach.

Metric	Value
Mean of Success Rate	0.38
Variance of Success Rate	0.14
Mean Reward	155.3

Table 4.4: Baseline SAC Performance

SAC-MC Performance

We selected the best-performing model in the FetchPush environment as our starting point. The results were comparable to those of the baseline SAC. However, we observed an unusually high intrinsic reward, which was attributed to a maximum percentile value of 94. To address this, we conducted further experiments with the max percentile and determined that a value of 98 was optimal. In the following, we present the results from the last model:

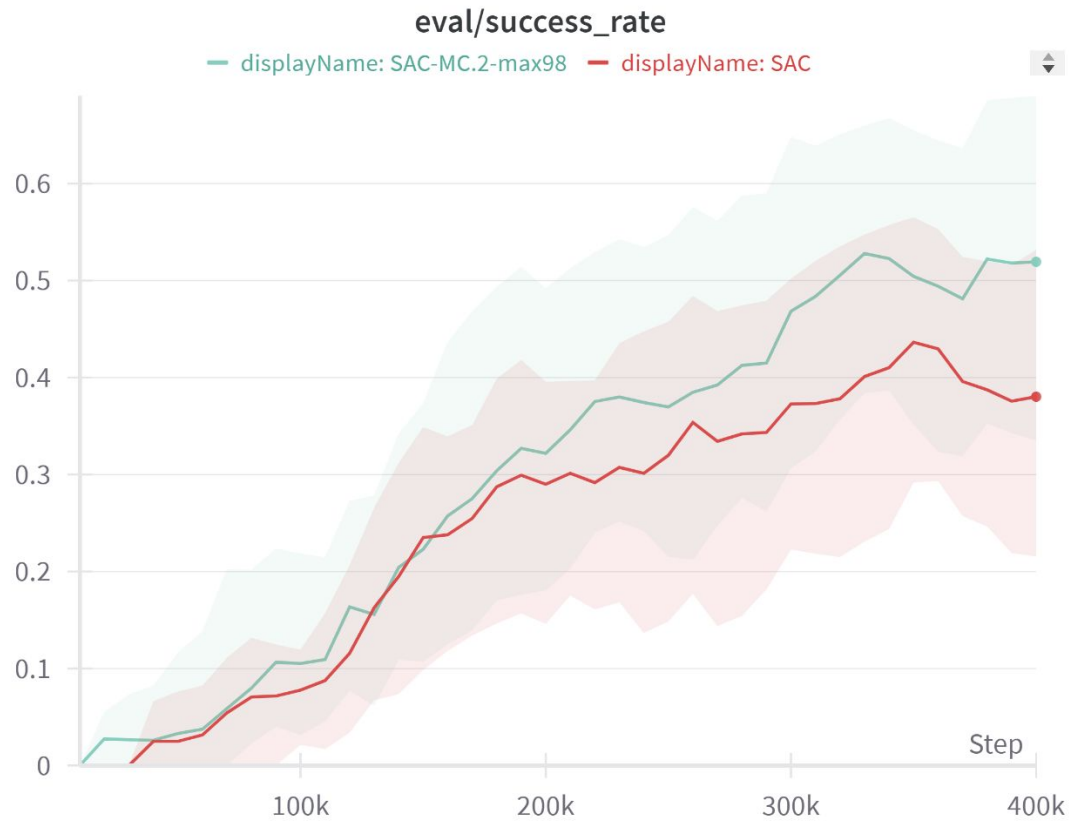


Figure 4.12: SAC-MC Success Rate with $\alpha = 0.2$ and Max Percentile = 98

It is worth mentioning that both the baseline SAC and SAC-MC algorithms successfully develop effective policies, despite their relatively average success rates.

Metric	Value
Mean of Success Rate	0.52
Variance of Success Rate	0.16
Mean Reward	206.5

Table 4.5: SAC-MC $\alpha = 0.2$

These policies exhibit sharp behavior, consistently directing the agent toward the goal in the most efficient manner. The relatively low success rates can be attributed to the nature of the dense reward structure, which requires a high degree of precision for the agent to achieve the full reward. In the following, we demonstrate an illustration of it:

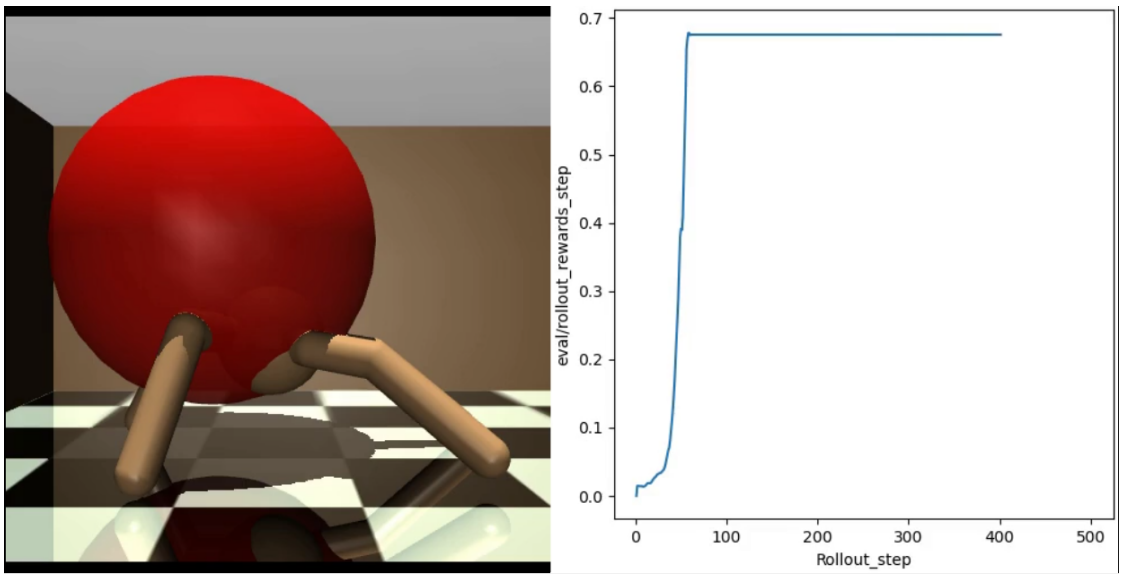


Figure 4.13: Dense Reward of Ant Maze Environment

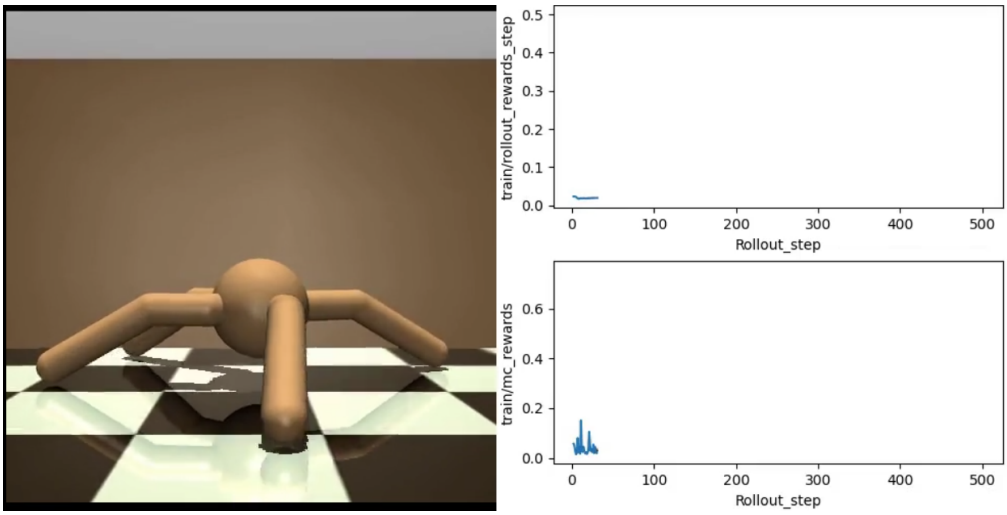
Even though the ant reached the goal, it achieved a success rate of only below 70%. Nonetheless, we considered this to be a good policy and evaluated the results accordingly.

Empirical evidence as for the FetchPush environment demonstrates a clear advantage of the SAC-MC algorithm over the baseline SAC algorithm in terms of success rate. Given that SAC-MC with 20% morphological computation outperformed the baseline SAC, and due to the limited time remaining for the experiment, we focused our testing exclusively on the best-performing SAC-MC variant with $\alpha = 0.2$.

4.2.2 Qualitative Evaluation

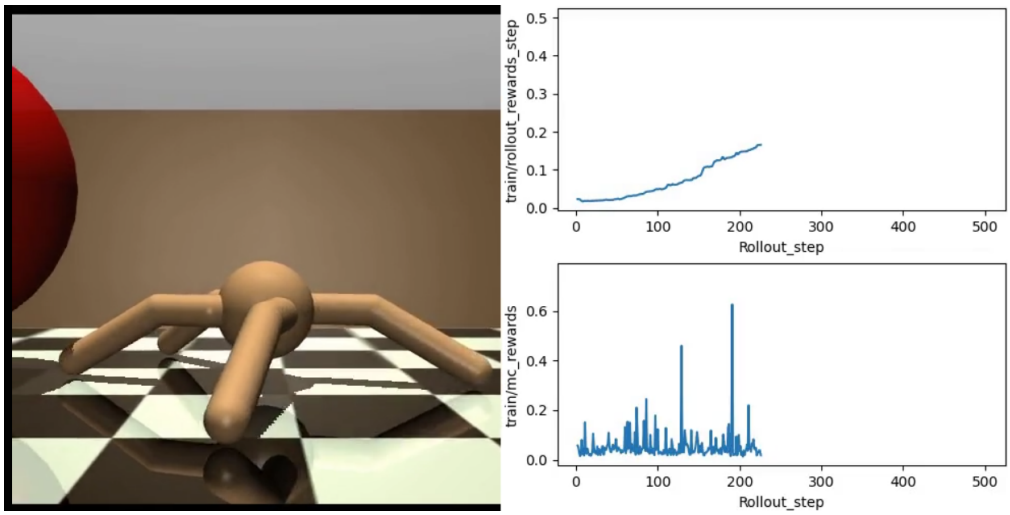
In the following, we examine the behavioral adaptations of the Ant agent when influenced solely by high levels of Morphological Computation (MC). Without the

influence of any goal-directed reward, the agent developed unique and intriguing strategies to navigate its environment. Notably, the agent learned to control its body in an unconventional manner, often freezing certain joints and propelling itself by sliding using only one leg. This behavior indicates an exploration of bodily control, where the agent minimizes energy expenditure in some joints while maximizing it in others. A possible explanation for why the agent tends to slide its legs instead of moving them when high morphological computation is applied could be that the agent is relying heavily on the positional and velocity information of its joints. This detailed feedback, combined with the dynamics of the environment, might lead the agent to adopt a strategy that minimizes complex leg movements in favor of sliding, as this could be perceived as a more efficient way to maximize MC. By leveraging the rich sensory input, the agent might be optimizing for a smoother movement, even if it involves unconventional movement patterns like sliding. In the following figure, we demonstrate the sliding phenomena that happen while training the agent with only an MC reward. Figure 4.14 shows the discussed above.

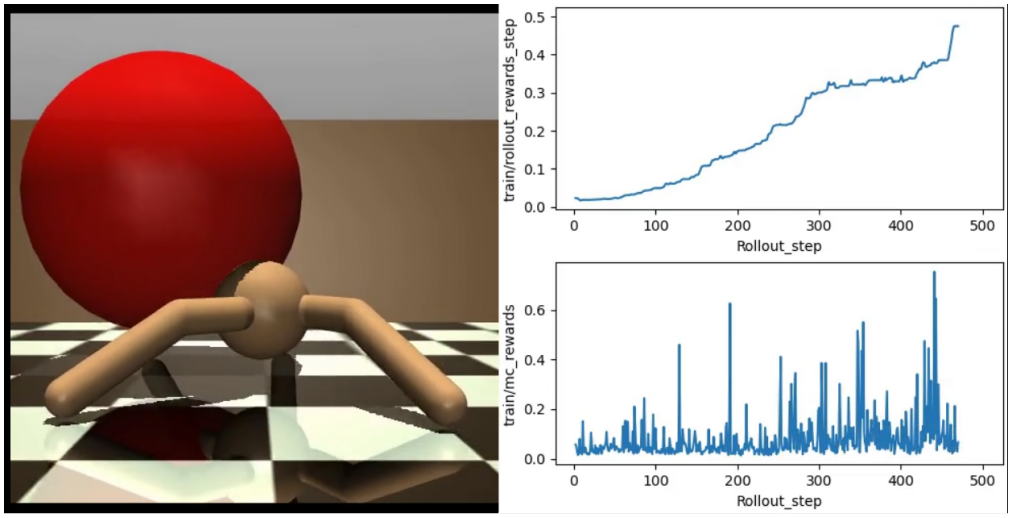


(a) Initial Position of the ant

Figure 4.14: (Continued on next page)



(b) Moving with one leg only while freezing other joints

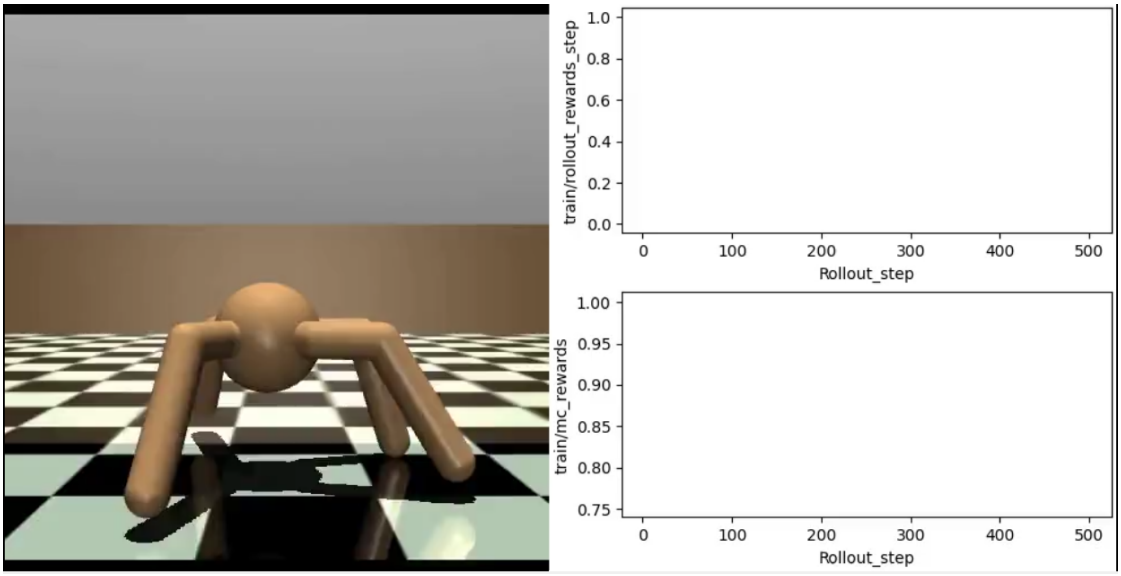


(c) Maintaining the same policy over a long-distance traversal

Figure 4.14: The ant agent developed an unusual phenomenon of sliding while minimizing joint movement, yet still being able to move effortlessly in any direction and cover long distances.

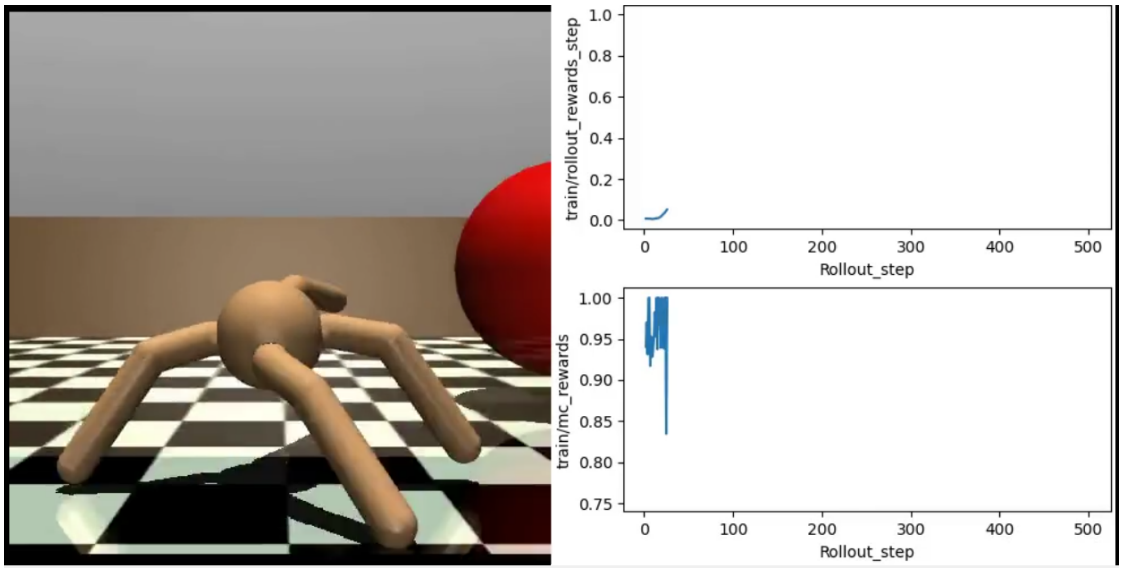
Effect of MC as Part of the Reward

When Morphological Computation (MC) was integrated as part of the reward system, combined with the Soft Actor-Critic (SAC) algorithm, the agent's behavior exhibited significant improvements in terms of efficiency and goal-directed actions. The inclusion of MC in the reward structure resulted in a quicker, more agile agent whose movements were optimized to reach the goal as swiftly as possible. A possible explanation is that the agent utilizes the velocities obtained from its observations, which MC reward aims to maximize, to capitalize on high speed, which results in quicker achievement of the goal. The agent displayed an enhanced ability to pick up speed towards the goal, incorporating small jumps or transitioning between walking on three or even two legs to accelerate and turn smoothly. This behavior suggests that the integration of MC with SAC not only preserved the agent's ability to utilize its body in creative ways but also enhanced its ability to adapt those movements toward achieving the set objectives more effectively. In Figure 4.15 we demonstrate the discussed above.

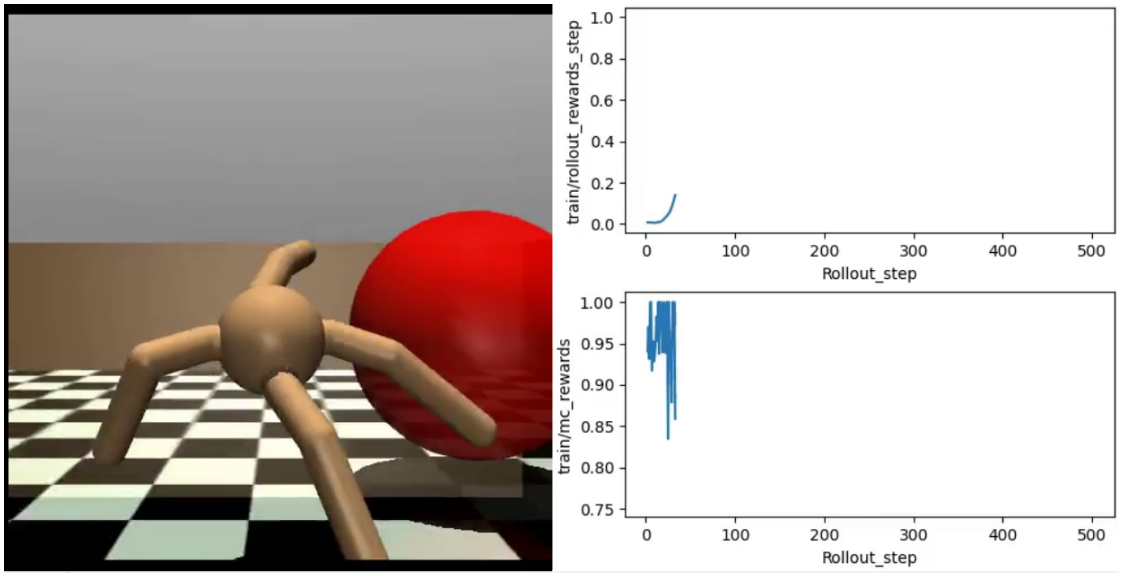


(a) Initial Position of the ant

Figure 4.15: (Continued in next Pages)

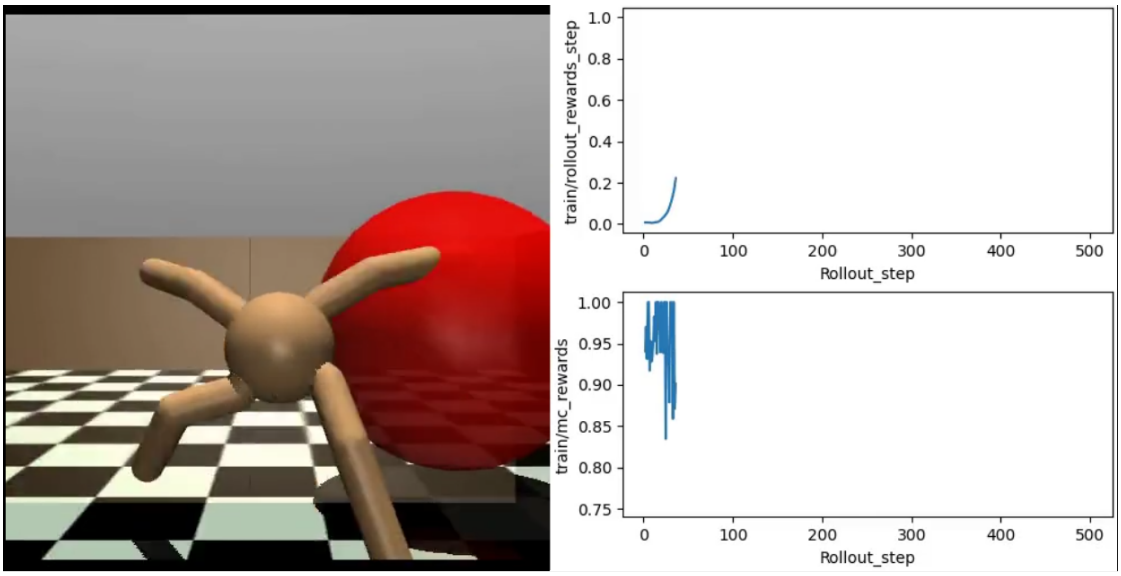


(b) Picking up speed

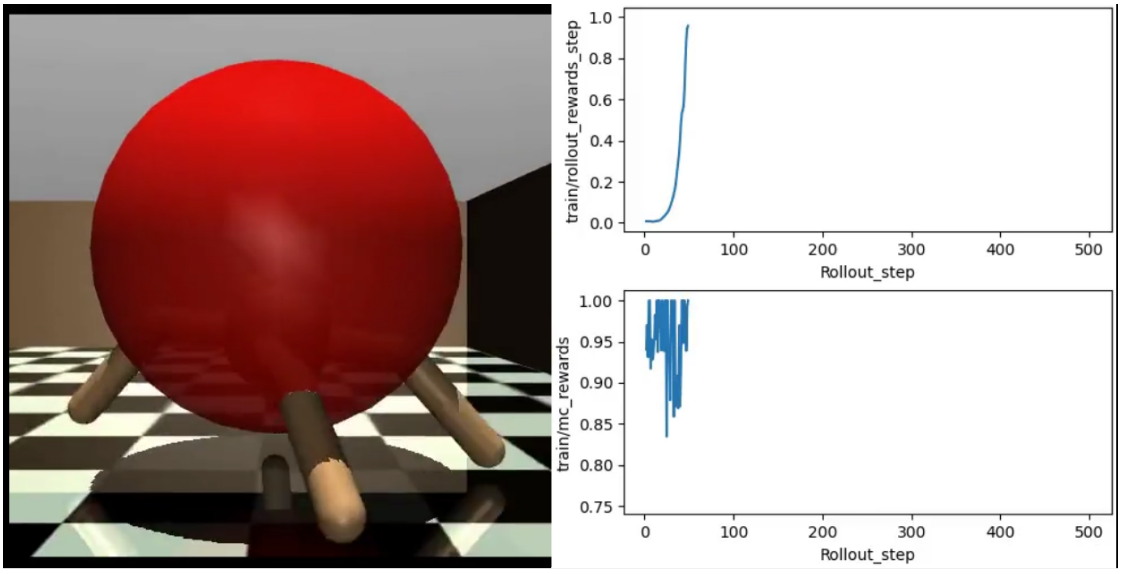


(c) Jumping towards the goal

Figure 4.15: (Continued in next Page)



(d) Landing and proceeding



(e) Achieving the goal

Figure 4.15: The agent moves at full speed toward the goal, sometimes resulting in it standing on two legs or even jumping. Despite these actions, it remains stable and tries to utilize every joint movement to its full potential.

Chapter 5

Conclusion

In this work, we explored the integration of Morphological Computation (MC) into the RL to enhance the learning performance of reinforcement learning agents. Our approach, termed SAC-MC, incorporates varying levels of morphological computation to assess its impact on the agent's success rate and training time. The results of our extensive experiments reveal several critical insights:

- Firstly, the novel integration of morphological computation significantly enhances the performance of the SAC algorithm. The empirical evidence indicates that SAC-MC consistently outperforms the baseline SAC algorithm in terms of success rate and average reward, particularly when a moderate level of morphological computation is applied. Specifically, the SAC-MC algorithm with 20% morphological computation demonstrates the highest success rate and average reward, indicating that this level of MC is optimal for balancing the benefits of enhanced learning without introducing excessive complexity.

- Secondly, our findings suggest a diminishing return on performance improvements as the level of morphological computation increases beyond 20%. While moderate levels of MC (e.g., 20%) enhance learning, higher levels (e.g., 80%, and 100%) do not provide additional benefits and may even hinder the learning process. This is evidenced by the lower success rates and average rewards at these higher levels of MC. The extended training times and potential lack of convergence at these levels further support this observation.

- Finally, we observed several intriguing phenomena where the agent effectively leveraged the dynamics of the environment to its advantage. This included utilizing principles such as applying pressure to objects and harnessing momentum, among other dynamic interactions.

These conclusions highlight the potential of morphological computation as a valuable tool for enhancing reinforcement learning algorithms. However, they also emphasize the need for careful parameter tuning to achieve optimal results.

5.1 Summary of Contributions

This research makes the following contributions to the field of reinforcement learning and the development of more efficient learning algorithms through the integration of morphological computation:

1. **Novel Integration of MC with RL:** We introduced a novel approach to integrate morphological computation with the Soft Actor-Critic (SAC) algorithm, resulting in the SAC-MC framework. This integration allows for the systematic evaluation of the impact of varying levels of MC on learning performance.

2. **Assumption over the best MC Level:** Our results for the Fetch Push task show that a 20% level of morphological computation or $\alpha = 0.2$ performed best for enhancing the success rate and average reward of the SAC algorithm. This finding provides a valuable guideline for applying MC in reinforcement learning tasks.

These contributions collectively advance the state-of-the-art in reinforcement learning by demonstrating the potential of morphological computation to enhance learning algorithms, providing empirical evidence for its benefits, and offering practical guidelines for its application.

5.2 Future Research

While this study has provided valuable insights into the integration of morphological computation with reinforcement learning algorithms, several avenues for future research remain. These include:

1. **Exploration of Other RL Algorithms:** Future research can explore the integration of morphological computation with other reinforcement learning algorithms beyond SAC. Investigating its impact on algorithms such as Proximal Policy Optimization (PPO), Deep Q-Networks (DQN), and A3C (Asynchronous Advantage Actor-Critic) could provide broader insights into the generalizability of our findings.

2. **Adaptive MC Levels:** Developing adaptive mechanisms to dynamically adjust the level of morphological computation during training could further optimize learning performance. Future studies could investigate methods for real-time calibration of MC levels based on the agent's performance and learning progress.

3. **Complex Environments:** Testing the SAC-MC framework in more complex and diverse environments could provide a deeper understanding of its robustness and scalability. Evaluating its performance in high-dimensional, continuous action spaces, and multi-agent scenarios would be particularly valuable.

4. **Long-term Performance:** Analyzing the long-term performance and stability of SAC-MC over extended training periods is crucial for understanding its potential in real-world applications. Future research could focus on evaluating the algorithm's performance in long-duration tasks and its ability to retain learned behaviors.

By addressing these research directions, future studies can build on the findings of this work, further advancing the field of reinforcement learning and the effective integration of morphological computation for improved learning algorithms.

Appendix A

Nomenclature

Parameters of the used models

Name	Value
Learning Rate	0.0003
Buffer Size	1,000,000
Batch Size	256
Tau	0.005
Gamma	0.99
Entropy Coefficient (ent_coef)	auto
Use Hindsight Experience Replay (HER)	True
Number of Critics (n_critics)	2

Table A.1: Parameters used in training for the SAC baseline model on Fetch Push Environment.

Parameter	**Value**
Learning Rate	0.0003
Buffer Size	1,000,000
Batch Size	256
Tau	0.005
Gamma	0.99
Entropy Coefficient (ent_coef)	auto
Use Hindsight Experience Replay (HER)	True
Number of Critics (n_critics)	2
Morphological Computation Reward (mc_alpha)	x
Max Percentile (Upper Bound for KLD normalization)	94
Min Percentile (Lower Bound for KLD normalization)	0
Morphological Computation Learning Rate (mc_lr)	0.0003

Table A.2: Hyperparameters used for training the SAC-MC on the Fetch Push Environment with the following variants, where $x \in \{0.2, 0.4, 0.6, 0.8, 1.0\}$.

Parameter	**Value**
Learning Rate	0.0004
Buffer Size	1,000,000
Batch Size	2048
Tau	0.05
Gamma	0.95
Entropy Coefficient (ent_coef)	auto
Use Hindsight Experience Replay (HER)	True
Number of Critics (n_critics)	2

Table A.3: Hyperparameters used for training the baseline SAC on Ant Maze Environment.

Parameter	**Value**
Learning Rate	0.0004
Buffer Size	1,000,000
Batch Size	2048
Tau	0.05
Gamma	0.95
Entropy Coefficient (ent_coef)	auto
Use Hindsight Experience Replay (HER)	True
Number of Critics (n_critics)	2
Morphological Computation Reward (mc_alpha)	0.2
Max Percentile (Upper Bound for KLD normalization)	98
Min Percentile (Lower Bound for KLD normalization)	0
Morphological Computation Learning Rate (mc_lr)	0.0004

Table A.4: Hyperparameters used for training the SAC-MC on Ant Maze Environment.

Appendix B

Additional Proofs

B.1 Policy and Value Functions

In reinforcement learning, a policy π defines the agent's behavior by specifying a probability distribution over actions for each state. Formally, a policy $\pi(a|s)$ represents the probability of taking action a given the state s, where $\pi(a|s) \geq 0$ and $\sum_{a \in A(s)} \pi(a|s) = 1$. The agent samples its next action from this distribution.

The state-value function $V^\pi(s)$ measures the expected return when starting in state s and following policy π thereafter. It provides a measure of the long-term desirability of states under a specific policy. The state-value function is formally defined as:

$$V^\pi(s) = \mathbb{E}^\pi[G_t \mid S_t = s] = \mathbb{E}^\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s \right], \tag{B.1}$$

where G_t is the return and γ is the discount factor. This can be expressed recursively using the Bellman equation:

$$V^\pi(s) = \mathbb{E}^\pi[R_{t+1} + \gamma V^\pi(S_{t+1}) \mid S_t = s]. \tag{B.2}$$

By breaking down the expectation, we have:

$$V^\pi(s) = \sum_{a \in A(s)} \pi(a|s) \sum_{s' \in S} P(s, a, s') \left[R(s, a, s') + \gamma V^\pi(s') \right]. \tag{B.3}$$

The action-value function $Q^\pi(s, a)$ provides the expected return of taking action a in state s and thereafter following policy π. It is defined as:

$$Q^\pi(s, a) = \mathbb{E}^\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}^\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a \right]. \tag{B.4}$$

Similarly, $Q^\pi(s, a)$ can be expressed recursively:

$$Q^\pi(s, a) = \mathbb{E}^\pi[R_{t+1} + \gamma V^\pi(S_{t+1}) \mid S_t = s, A_t = a]. \tag{B.5}$$

Expanding this, we get:

$$Q^{\pi}(s,a) = \sum_{s' \in S} P(s,a,s') \left[R(s,a,s') + \gamma \sum_{a' \in A(s')} \pi(a'|s') Q^{\pi}(s',a') \right]. \tag{B.6}$$

The goal of reinforcement learning is to find an *optimal policy* π^* that maximizes the expected return for all states. An optimal policy satisfies:

$$\pi^* = \arg\max_{\pi} V^{\pi}(s) \quad \text{for all } s \in S. \tag{B.7}$$

The optimal state-value function $V^*(s)$ and *the optimal action-value function* $Q^*(s,a)$ under the optimal policy π^* are defined as:

$$V^*(s) = \max_{\pi} V^{\pi}(s) = \max_{a \in A(s)} Q^*(s,a), \tag{B.8}$$

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a). \tag{B.9}$$

The Bellman optimality equation for $V^*(s)$ is:

$$V^*(s) = \max_{a \in A(s)} \sum_{s' \in S} P(s,a,s') \left[R(s,a,s') + \gamma V^*(s') \right]. \tag{B.10}$$

Similarly, *the Bellman optimality equation* for $Q^*(s,a)$ is:

$$Q^*(s,a) = \sum_{s' \in S} P(s,a,s') \left[R(s,a,s') + \gamma \max_{a' \in A(s')} Q^*(s',a') \right]. \tag{B.11}$$

These equations provide a foundation for various algorithms in reinforcement learning, which aim to approximate or directly compute $V^*(s)$ and $Q^*(s,a)$ to derive the optimal policy π^*.

B.1.1 Q-Learning

Q-learning is a widely used off-policy Temporal Difference (TD) control algorithm. In this approach, the policy π is not directly used for updating the Q-values, which makes it distinct from on-policy methods like Sarsa. The key difference between Q-Learning and Sarsa is that Q-Learning uses the maximum Q-value of the next state S_{t+1}, rather than the Q-value of the actual next state-action pair (S_{t+1}, A_{t+1}) chosen by the policy. The action-value update rule for Q-Learning is given by:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_{a} Q(S_{t+1}, a) - Q(S_t, A_t) \right] \tag{B.12}$$

Q-Learning iteratively updates the Q-values using the reward received and the best possible future rewards, aiming to find the optimal action-value function that maximizes the cumulative reward.

Convergence of Q-Learning

Q-Learning is guaranteed to converge to the optimal action-value function $Q^*(s, a)$ under certain conditions, including that all state-action pairs continue to be updated, and the learning rate decreases appropriately. This convergence means that the agent can learn the best possible policy for the given environment, which can then be used to determine the optimal sequence of actions to maximize cumulative rewards. Thus, Q-Learning provides a robust framework for learning optimal policies in unknown environments, balancing exploration and exploitation through its update rule and the choice of policy during learning

Appendix C

Complete Simulation Results

We changed the maximum percentile for the normalization of the intrinsic reward to 98, as mentioned in Chapter 4 for the AntMaze task. Here are the results with the 94th percentile, which worked best for the FetchPush task. We argued that 94 is too low a value for normalization, as the intrinsic reward during training was frequently reaching the maximum value or even exceeding it. The results showed comparable performance when considering the success rate, this is demonstrated in Figure C.1

Parameter	**Value**
Learning Rate	0.0004
Buffer Size	1,000,000
Batch Size	2048
Tau	0.05
Gamma	0.95
Entropy Coefficient (ent_coef)	auto
Use Hindsight Experience Replay (HER)	True
Number of Critics (n_critics)	2
Morphological Computation Reward (mc_alpha)	0.2
Max Percentile (Upper Bound for KLD normalization)	94
Min Percentile (Lower Bound for KLD normalization)	0
Morphological Computation Learning Rate (mc_lr)	0.0004

Table C.1: Hyperparameters used for training the SAC-MC on Ant Maze Environment with max percentile = 94.

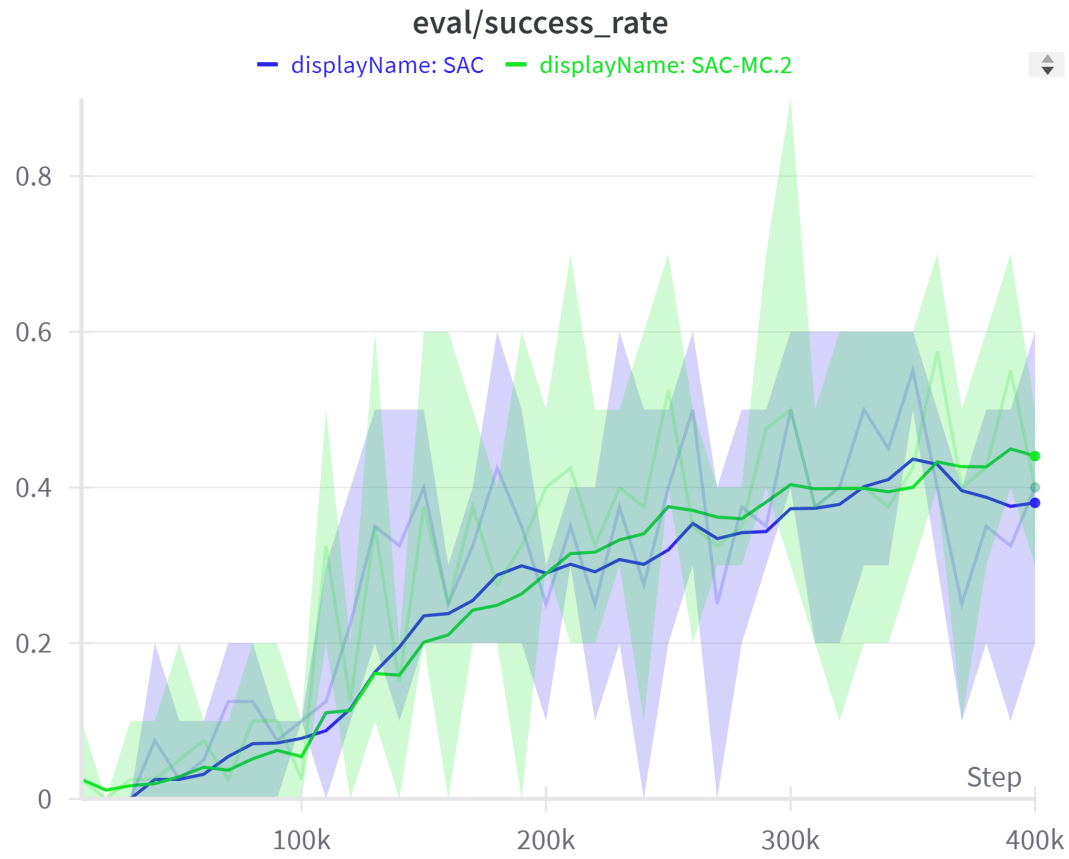


Figure C.1: SAC-MC with $\alpha = 0.2$ and max percentile $= 94$

Bibliography

[1] Manfred Eppe, Adwait Datar, Carlotta Langer, Danny Cam Hoa Tien, Fin Armbrecht, Frank Röder, Jan Dohmen, Leon Kruse, Leon Sierau, Maik Marius Rebaum, and Omar Baiazid. Scilab-rl, 2024. Created: November 02, 2021.

[2] Keyan Ghazi-Zahedi, Raphael Deimel, Guido Montúfar, Vincent Wall, and Oliver Brock. Morphological computation: The good, the bad, and the ugly. *IEEE*, September 2017.

[3] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, and Sergey Levine. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905*, 2018. Available online at `https://arxiv.org/abs/1812.05905`.

[4] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *Computing Research Repository (CoRR)*, abs/1509.02971, 2016. Available at: `https://arxiv.org/abs/1509.02971`.

[5] Gaia Molinaro and Anne G. E. Collins. Intrinsic rewards explain context-sensitive valuation in reinforcement learning. *PLOS Biology*, 21(7):e3002201, 2023.

[6] Vincent C. Müller and Matej Hoffmann. What is morphological computation? on how the body contributes to cognition and control. *Frontiers in Robotics and AI*, 2018.

[7] Author names. Social navigation with human empowerment driven deep reinforcement learning. In *Proceedings of the International Conference on Social Robotics*, pages 395–407, 2020. First Online: October 14, 2020.

[8] David A. Nix and Andreas S. Weigend. Estimating the mean and variance of the target probability distribution. *Department of Computer Science and Institute of Cognitive Science, University of Colorado at Boulder*, 1994. dnix@cs.Colorado.edu.

[9] Jaak Panksepp. *Affective Neuroscience: The Foundations of Human and Animal Emotions*. Oxford University Press, New York, 1998.

[10] Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[11] Christoph Salge and Daniel Polani. Empowerment as replacement for the three laws of robotics. *Frontiers in Robotics and AI*, 2014.

[12] Iqbal H. Sarker. Deep learning: A comprehensive overview on techniques, taxonomy, applications and research directions. *Journal of Computer Science and Technology*, 2021.

[13] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 2nd edition, 2018.

[14] Christopher J.C.H. Watkins and Peter Dayan. Q-learning. Technical report, University of Edinburgh, Edinburgh EH8 9EH, Scotland, 1992. Technical Note.

[15] Mingqi Yuan, Bo Li, Xin Jin, and Wenjun Zeng. Automatic intrinsic reward shaping for exploration in deep reinforcement learning. *The Fortieth International Conference on Machine Learning (ICML 2023)*, page 24, 2023. Submitted on 26 Jan 2023 (v1), last revised 12 Oct 2023 (v5).

Erklärung der Urheberschaft

Ich versichere an Eides statt, dass ich die Master's Thesis im Studiengang Computer Science selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ort, Datum Unterschrift

Erklärung zur Veröffentlichung

Ich erkläre mein Einverständnis mit der Einstellung dieser Master's Thesis in den Bestand der Bibliothek.

Ort, Datum Unterschrift