Bachelor Thesis

Expanding Action Space in Reinforcement Learning through Latent Models: A 2D Inverse Kinematics Benchmark

Robin Uhrich

Examiner: Prof. Dr. Joschka Boedecker
Advisers: Jasper Hoffman

University of Freiburg
Faculty of Engineering
Department of Computer Science
Neurobotics Lab

September 25, 2023

Writing Period

11. 07. 2023 – 25. 09. 2023

Examiner

Prof. Dr. Joschka Boedecker

Advisers

Jasper Hoffman

Bachelor Thesis

Expanding Action Space in Reinforcement Learning through Latent Models: A 2D Inverse Kinematics Benchmark

Robin Uhrich

Gutachter: Prof. Dr. Joschka Boedecker
Betreuer: Jasper Hoffman

Albert-Ludwigs-Universität Freiburg

Technische Fakultät

Institut für Informatik

Lehrstuhl für Neurorobotik

September 25, 2023

Bearbeitungszeit

11. 07. 2023 – 25. 09. 2023

Gutachter

Prof. Dr. Joschka Boedecker

Betreuer

Jasper Hoffman

Declaration

I hereby declare that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.
I hereby also declare that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

_____ _____

Place, Date Signature

Abstract

Reinforcement Learning (RL) is a powerful paradigm for training agents to make sequential decisions in dynamic environments. This thesis explores innovative approaches to address the challenge of expanding the action space of RL agents. This research leverages latent models, specifically Variational Autoencoders (VAEs) and artificial neural networks trained in a supervised fashion, with the goal to transform the action space, enabling RL agents to handle a bigger action space.

The investigation centers on the application of these latent models to a 2D Inverse Kinematics problem in a robotic arm with numerous joints. Through a careful analysis of experimental results, the thesis assesses the impact of action space enlargement on RL agent performance and scalability.

This thesis discusses tey findings and contributions to the advantages and limitations of latent model integration and the challenges of high-dimensional action spaces for optimizing RL agents' decision-making capabilities.

By bridging the gap between latent models and RL, this research provides a foundation for future investigations into enhancing the capabilities of autonomous agents.

Zusammenfassung

Selbstbestimmtes Lernen (Reinforcement Learning, RL) ist ein leistungsstarkes Paradigma zur Schulung von Agenten, um sequenzielle Entscheidungen in dynamischen Umgebungen zu treffen. Diese Arbeit erforscht Ansätze zur Bewältigung der Herausforderung, den Handlungsraum von RL-Agenten zu erweitern. Diese Thesis nutzt latente Modelle, insbesondere Variational Autoencoders (VAEs) und künstliche neuronale Netze mit dem Ziel, den Handlungsraum zu transformieren und zu vergrößern, um RL-Agenten in die Lage zu bringen, einen größeren Handlungsraum zu bewältigen.

Die Untersuchung konzentriert sich auf die Anwendung dieser latenten Modelle auf ein 2D-Inverse-Kinematikproblem in einem Roboterarm mit zahlreichen Gelenken. Durch eine Analyse der experimentellen Ergebnisse bewertet die Arbeit die Auswirkungen der Vergrößerung des Handlungsraums auf die Leistung und die Skalierbarkeit von RL-Agenten.

Diese Arbeit diskutiert hierbei die Hauptergebnisse und Beiträge zu den Vor- und Nachteile der Integration latenter Modelle und die Herausforderungen von hochdimensionalen Handlungsräumen zur Optimierung der Entscheidungsfähigkeiten von RL-Agenten.

Indem sie die Kluft zwischen latenten Modellen und RL überbrückt, legt diese Forschung die Grundlage für zukünftige Untersuchungen zur Verbesserung der Fähigkeiten autonomer Agenten.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

In the ever-evolving landscape of artificial intelligence and machine learning, Reinforcement Learning (RL) stands as a powerful paradigm for enabling autonomous agents to learn and adapt to their environments. RL has demonstrated remarkable success in a variety of applications, from game-playing agents [1] to robotics control systems [2] [3]. However, one of the enduring challenges in RL lies in the limitation of the action space—an agent's ability to act upon its environment. The dimensionality and complexity of the action space profoundly impact an agent's capacity to solve increasingly intricate tasks [4].

This thesis explores a novel approach to address the action space limitations in RL by leveraging latent models. Specifically, it investigates the integration of pre-trained latent models, including Variational Autoencoder (VAE) decoders and supervised models, into RL frameworks. The primary aim is to empower RL agents to handle significantly more extensive and complex action spaces effectively. To evaluate the effectiveness of this approach, we developed a new scalable benchmark environment: the 2D Inverse Kinematics problem.

This thesis is motivated by two central questions:

1. **How does an increase in action space dimensionality affect the performance of a reinforcement learning agent?** In RL, an agent's success at completing tasks of varying complexity is greatly influenced by the dimension

and complexity of the action space. An exploration of how changes in action space dimensionality impact an agent's capabilities is vital.

2. **Does the employment of latent models increase the action space complexity an agent can handle?** Leveraging pre-trained latent models, including Variational Autoencoder (VAE) decoders and supervised models, into RL frameworks offers a promising avenue to expand the boundaries of RL action spaces, but does it effectively enhance an agent's capability to navigate complex action spaces?

These questions lie at the core of our research endeavor and guide our exploration into the integration of latent models into RL frameworks to empower agents to handle significantly more extensive and complex action spaces effectively.

The Challenge of Action Space in Reinforcement Learning

The action space of an RL agent comprises the set of actions it can take to interact with its environment. In many real-world scenarios, especially those involving robotic systems and control tasks, the action space can be high-dimensional and continuous. Traditional RL algorithms often struggle to operate effectively in such settings due to the curse of dimensionality, making the exploration of all possible actions computationally infeasible.

This challenge has motivated researchers to seek innovative solutions that expand the boundaries of RL action spaces. The integration of latent models offers a promising avenue. These models can encode complex actions into lower-dimensional representations, enabling RL agents to navigate large action spaces more efficiently and effectively.

The Role of Latent Models in Expanding Action Spaces

Latent models, such as VAEs and supervised models, have demonstrated their prowess in capturing essential patterns and representations within data [5]. By pre-training these models on relevant tasks, we can harness their latent spaces to transform high-dimensional action spaces into more compact and manageable forms. This, in turn, equips RL agents with the ability to explore and exploit action spaces that were previously deemed insurmountable.

Benchmarking Progress: The 2D Inverse Kinematics Problem

To assess the efficacy of integrating latent models into RL, we turn to the 2D Inverse Kinematics problem. This problem simulates the control of a multi-jointed robotic arm, where each joint represents a dimension in the action space. The challenge lies in determining the joint angles required to position the end effector at a specified target location. By scaling the complexity of this benchmark through varying the number of joints N, we can rigorously evaluate the impact of latent models on action space expansion.

Structure of the Thesis

This thesis is structured as follows: in Chapter 3, we provide an overview of the background for this research. Chapter 2 delves into a comprehensive literature review, examining related work in RL, latent models, and their integration. Chapter 4 outlines the methodology used, including details on the latent models employed and their integration into RL. Chapter 5 presents the experimental results and analysis from the 2D Inverse Kinematics benchmark. Chapter 6 discusses experimental results and finally, Chapter 7 offers conclusions and outlines potential future directions for research in this domain.

The integration of latent models with RL to expand action space capabilities holds great promise for advancing the capabilities of autonomous agents. By addressing one of the core challenges in RL, this research aims to contribute to the broader field

of artificial intelligence and robotics, opening doors to new possibilities in complex task execution and problem-solving.

2 Related Work

In the following section we will briefly explain the already existing research regarding hwo previous research tackled the problem of inverse kinematics and how current research combines Latent models with reinforcement learning.

2.1 Learning Operational Space Control

The authors form the paper [6], Jan Peters and Stefan Schaal, present a novel approach for controlling the end-effector of a rigid robot arm. Using operational space control in robots meant to work safely in human environments is challenging due to unmodeled nonlinearities, leading to accuracy reduction and unpredictable behavior.
To address this challenge, learning control methods are explored. Traditional learning methods struggle to capture the structured knowledge required for operational space control, such as Jacobians and inertia matrices, which may not always be observable. This paper introduces novel approaches to learning operational space control, focusing on learning the operational space control law directly, similar to an inverse model learning problem.
Key insights for this project include the realization that a physically correct solution to the inverse problem with redundant degrees-of-freedom is attainable through piecewise linear learning. Additionally, many operational space controllers can be viewed as constrained optimal control problems. A learning algorithm is formulated to synthesize a globally consistent desired resolution of redundancy while learning the

operational space controller, treated as a reinforcement learning problem maximizing an immediate reward.

2.2 Motor synergy development in high-performing deep reinforcement learning algorithms

In their paper [7] the researchers Jiazheng Chai and Mitsuhiro Hayashibe investigate if the motor synergy concept could also be observed in deep reinforcement learning for robotics. The motor synergy concept states that a sets of actuators e.g. muscles in animals or motors in robots, are controlled by a reduced set of commands with respect to the degrees of freedom resembled by the actuators. They study this concept by training an agent either with SAC or TD3 on several benchmark environments like Half-Cheetah [8], and further investigate if it is possible ot reduce the dimensionality of the collected control signal via principal component analysis [9] and reconstruct it as close as possible to the original signal.

2.3 LASER: Learning Latent Action Space for Efficient Reinforcement Learning

In the paper from Arthur Allshire, Roberto Martín-Martín, Charles Lin, Shawn Manuel, Silvio Savarese and Animesh Gartg from 2021 [10] the authors trained jointly a Soft Actor-Critic algorithm and a conditional Variational Autoencoder for robot manipulation. The authors split the problem into two sub-problems. First they lear a mapping $g(o) : \mathcal{O} \rightarrow \bar{\mathcal{A}}$ from observation space into a latent space and second using a robot controller $f(\bar{a}) : \bar{\mathcal{A}} \rightarrow \mathcal{A}$ to map their latent signals into actuation commands. Their algorithm LASER is trained as an encoder-decoder model which learns to map manifold of low-level commands to a latent action space.

2.4 CALM: Conditional Adversarial Latent Models for Directable Virtual Characters

Researchers from NVIDIA Chen Tessler et al. have developed Conditional Adversarial Latent Models (CALM) in the paper "CALM: Conditional Adversarial Latent Models for Directable Virtual Characters"[11] to enable users to direct the behavior of virtual characters in interactive simulations. CALM combines imitation learning with a control policy and motion encoder to create a representation of human motion that is diverse and controllable. The key phases of CALM include:

1. low-level training, where it learns a motion encoder and decoder. The encoder converts a motion from a reference motion dataset into a low-dimensional latent representation using a time series of joint locations. A low-level policy called the decoder interacts with the simulator and produces movements that are similar to those in the reference dataset.

2. directionality control, where a high-level policy guides motion direction and style in latent space which are than passed to the low level policy.

3. and inference, where without additional training, complex movements are composed via a finite-state machine by combining the previously trained models (low-level policy and directional controller).

This approach allows users to intuitively control virtual characters, making it applicable for interactive applications like video games.

3 Background

The following part is to provide a short introduction into the theory of reinforcement learning, the Soft Actor Critic Algorithm (SAC) as well to Variational Autoencoders (VAE) and Conditional Variational Autoencoders (CVAE). In the end we will explain the concepts behind forward and inverse kinematics and go deeper into the algorithm representing a expert for inverse kinematics, Cyclic Coordinate Descent (CCD).

The following sections are a summary of the present work from several different groups and researchers. For more details please refer to their original publications or to more sophisticated literature like the textbook "Reinforcement Learning: An Introduction" from Richard Sutton and Andrew Barto [12], "Deep Learning" from Ian Goodfellow, Yoshua Bengio and Aaron Courville [13] and "Probabilistic Machine Learning: Advanced Topics" from Kevin Murphy [5].

3.1 RL Framework

Learning is a topic that is present in all our lives since their beginning. We all learn to move our arms, like an infant that is wiggling around, learn to walk, to speak and more or less any other skill that we find in our personal repository until now.

Reinforcement Learning (RL) is a computational approach based on learning by doing, where an agent interacts with an environment to maximize a reward. RL problems, like riding a bicycle, present challenges without explicit instructions for achieving

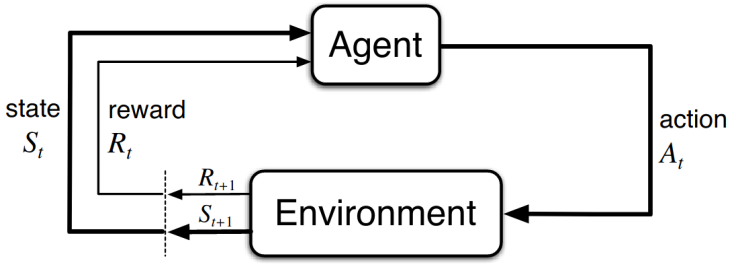


Figure 3.1: interaction between agent and the environment is living in to pursue his goal. Figure is kindly provided by R. Sutton and A. Barto in "Reinforcement Learning: An Introduction" [12]

the goal. They involve a closed-loop system where actions yield feedback, whether immediate (e.g., not crashing a bike) or delayed (e.g., earning a bonus in a board game). RL problems can be characterized by three key features:

- Closed-loop interaction between actions and environment feedback.
- Lack of explicit problem-solving instructions for the agent.
- Actions having consequences over time for the agent or the environment.

Before we can have an exemplary look on the closed loop mechanics, we would like to introduce a couple of key components of RL including the Markov Decision Process.

Every agent exists in a specified environment and also interacts within a sequence of discrete time steps $t \in \mathbb{N}$. In order to do so the agent has to perceive the environment through a **state** representation $S_t \in \mathcal{S}$ which is an element of all possible states, the state space $\mathcal{S}$. To interact with the environment the agent has to perform an **action** $A_t \in \mathcal{A}(S_t)$ where $\mathcal{A}(S_t)$ is the set of all possible actions in given state S_t. To chose an action the agent calculates for each possible action A_t a probability $\pi_{\theta,t}(A_t|S_t)$ for a given state S_t and samples from an action from this distribution. To achieve a more compact notation we will refer to the action $A_t = a$ and the state $S_t = s$. Further we

refer to the policy always as a parameterized policy $\pi_\theta = \pi_{\theta,t}(a|s)$. This probability distribution is called the **policy**. After the agent has chosen and executed an action, which also means the environment will move one time step further $t \to t+1$, the environment will return the next state S_{t+1} and the agent will perceive a **reward** $R_{t+1} \in \mathbb{R}$ with $\mathbb{R}$ as the set of all possible rewards.
The long term goal of the agent is to adapt its policy to maximize the total amount reward the agent receives from the environment. These key terminology can also be found in the notation of a Markov Decision Process which is classified as a 4-Tuple of: $(\mathcal{S}, \mathcal{A}, P_a(s,s'), R_a(s,s'))$ with $P_a(s,s')$ as a probability distribution that action a in state state s will lead to the next state $s' = s_{t+1}$ and $R_a(s,s')$ as reward function for transitioning from s to s' with a [12].

The overall goal of an agent is to maximize its received cumulative return G_t for one run until time step T [12]. This is metric is denoted in Equation (3.1). In Equation (3.1) γ is the discount rate, $0 \leq \gamma \leq 1$, a parameter to regularize the *importance* of individual received rewards over time.

$$G_t \doteq \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \tag{3.1}$$

$$= R_{t+1} + \gamma G_{t+1} \tag{3.2}$$

Due to the dependence of individual rewards R_t on state-action pairs from the same time step t, we can not compute the received cumulative return until the episodes end. To solve this problem reinforcement learning algorithms try to maximize the expected return $v_{\pi_\theta}(s)$ as in Equation (3.3) [12], or value function, for a given state. For Markov decision process we can define the value function as in Equation (3.4) [12].

$$v_{\pi_\theta}(s) \doteq \mathbb{E}_{\pi_\theta}[G_t|S_t = s] \tag{3.3}$$

$$\begin{aligned}
&\overset{(3.2)}{=} \mathbb{E}_{\pi_\theta}[R_{t+1} + \gamma G_{t+1}|S_t = s] \\
&= \sum_a \pi_\theta(a|s) \sum_{s'} \sum_r P(s', r|s, a) \left[r + \gamma \mathbb{E}_{\pi_\theta}[G_{t+1}|S_{t+1} = s']\right] \\
&= \sum_a \pi_\theta(a|s) \sum_{s',r} P(s', r|s, a) \left[r + \gamma v_{\pi_\theta}(s'm)\right] \qquad (3.4)
\end{aligned}$$

For a finite Markov decision process we can also define a the optimal value function $v_*(s)$ with respect to the optimal policy in Equation (3.7) adapted from Sutton and Barto [12].

$$q_{\pi_\theta}(s, a) = \sum_{s',r} P(s', r|s, a) \left[r + \gamma \sum_{a'} \pi_\theta(a'|s') q_{\pi_\theta}(s', a')\right] \tag{3.5}$$

$$v_*(s) \doteq \max_{\pi_\theta} v_{\pi_\theta}(s) \tag{3.6}$$

$$= \max_a \sum_{s',r} P(s', r|s, a)[r + \gamma v_*(s')] \tag{3.7}$$

Because Equation (3.6) holds we can formulatethe reinforcement learning problem as in Equation (3.8) for the optimal policy π^*_θ and with respect to a performance measure $J(\pi_\theta) = \mathbb{E}_{\pi_\theta}[G_t|S_t = s]$ as:

$$\pi^*_\theta = \arg\max_{\pi_\theta} J(\pi_\theta) \tag{3.8}$$

$$= \arg\max_{\pi_\theta} \mathbb{E}_{\pi_\theta}[G_t|S_t = s] \tag{3.9}$$

$$\overset{(3.3)}{=} \arg\max_{\pi_\theta} v_{\pi_\theta}(s) \tag{3.10}$$

$$q_*(s,a) = \sum_{s',r} P(s',r|s,a)\left[r + \gamma \max_{a'} q_*(s',a')\right] \tag{3.11}$$

3.2 Generalized Policy Iteration

This section is about to provide an introduction to the idea of policy iteration. For a more detailed look into the topic it is recommended refer to the Reinforcement Learning Textbook from Richard Sutton and Andrew Barto [12].

One fundamental concept for reinforcement learning algorithms is policy iteration. At Its core it is described as the alternating interaction between a policy improvement step and a policy evaluation step as in Figure 3.2. This interaction goes back and forth during the execution of most reinforcement learning algorithms until a state of convergence in the value function and policy has arrived. That means both value function and policy are optimal for the given problem. This state can be reached if the "value function is consistent with the current policy, and the policy stabilizes only when it is greedy with respect to the current value function". This can be also seen in the Bellman optimally equation as in Equation (3.7).

We can also think about this process as a tradeoff between improving the policy towards an optimal greedy behavior which makes the value function incorrect (blue

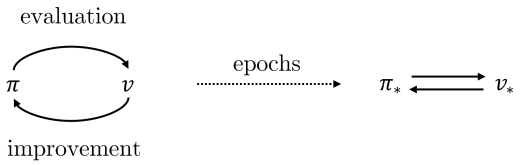


Figure 3.2: Policy iteration concept. We can see the alternating uptate between policy and value-function. This process continues until we reached either the optimal policy and optimal value-function or is limited by the number of epochs. Adapted graphic from [12]

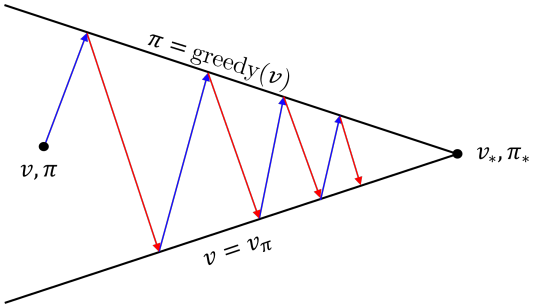


Figure 3.3: Policy iteration funnel. The red arrows are a symbolic policy improvement step which is moving away from a precise value function. The blue arrows are a symbolic policy evaluation step which is moving the current policy further away from a greedy and thus stable policy. Both criterions are moving over time closer together until convergence with the optimal policy π_* and the optimal value function v_*. Figure was adapted from [12]

arrows indicate a policy improvement step) and adapting the value function with respect to the current policy which makes the policy automatically less optimal (red arrows indicate a policy evaluation step). Despite this tradeoff the algorithm tends to find an optimal solution for the policy and the value function in the long run [12].

3.3 Soft Actor Critic

Soft Actor-Critic was introduced by Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel and Sergey Levine in 2018 in their paper: "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor" [14]. It is an algorithm belonging to the family of model free, off-policy, actor-critic reinforcement learning algorithms. The family of actor-critic algorithms addresses challenges in exploration and sample efficiency. Similar to other actor-critic algorithms it also employs the alternating concept of a policy improvement step and a policy evaluation step. In contrast to other actor-critic algorithms like deep deterministic policy gradient (DDPG) from [15] [16] it uses the concept of entropy regularization and uses a

stochastic policy for promoting exploration in the reinforcement learning environment. By jointly optimizing the policy and value functions using the maximum entropy objective enabled by entropy regularized, SAC effectively explores the environment while seeking to maximize rewards. This approach results in robust and adaptive policies that can efficiently handle various RL tasks also shown by Tuomas Haarnoja et al. [2]

In this section will introduce the concept of actor-critic algorithms, provide a detailed description of the SAC algorithm architecture, including entropy regularization and explain how to train the algorithm under the maximum entropy objective.

3.3.1 Actor-Critic Algorithms

The basic actor-critic architecture is a type of reinforcement learning algorithm that consists of two components: an actor and a critic [17]. The actor also referred to as the policy π_θ is responsible for selecting actions based on the current state, while the critic also referred to as a value function v_{π_θ} or a state value function q_{π_θ}, evaluates the quality of the actors actions by estimating the expected return. Like as introduced in Section 3.2 the actor uses the feedback from the critic to adjust its policy and improve its performance.

One additional factor to distinguish between reinforcement learning is by either on-policy learning or off-policy learning as in [12].
On-policy learning or off-policy learning refer to different methods for updating the policy in reinforcement learning. In on-policy learning, the agent learns from the data generated by its current policy, while in off-policy learning, the agent learns from data generated by a different policy. On-policy learning can be more stable, but it may require more data to converge. Off-policy learning can be more efficient, but it can be more sensitive to the quality of the data.

The advantage of using a critic in the actor-critic algorithm is: it provides a more stable feedback signal than using only rewards [18]. The critic estimates the expected return from a state, for the value function, or a state action pair for the action-value function, which takes into account the long-term consequences of actions. This allows the actor to learn from the critics feedback and improve its performance more efficiently than if it only received reward signals.

One limitation of the basic actor-critic architecture is that it can suffer from high variance and slow convergence due to the interaction between the actor and critic. This can be addressed through the use of techniques such as baseline subtraction and eligibility traces [18] [12].

Popular examples for actor critic algorithms are:

- Deep Deterministic Policy Gradient presented by Timothy P. Lillicrap et al. [16]
- Soft Actor-Critic presented by Tuomas Haarnoja [14]
- Asynchronous advantage actor critic (A3C) presented by Volodymyr Mnih et al. [19]

3.3.2 Entropy Regularization

Entropy regularization is a policy regularization technique by incorporating the policy entropy, used to encourage the policy to explore a diverse range of actions during training. In SAC, entropy regularization is achieved by adding the entropy $H(\pi_\theta)$ as in Equation (3.12) to the policy objective function in Equation (3.27) from [12].

$$\begin{aligned} \mathrm{H}(\pi_\theta) &:= \mathop{\mathbb{E}}_{s\sim\mathcal{D}}\left[\mathrm{H}(\pi_\theta(\cdot|s))\right] \\ &= \mathop{\mathbb{E}}_{s\sim\mathcal{D},a\sim\pi_\theta(\cdot|s)}\left[-\log(\pi_\theta(a|s))\right] \end{aligned} \tag{3.12}$$

Including entropy into the objective function encourages the policy to generate actions with higher entropy (higher randomness), leading to more exploration, possibly accelerate training and preventing converging to a poor local optimum [18].

The randomness or uncertainty of a policy for a given state action pair $\pi_\theta(a|s)$ can be computed by seeing $\pi_\theta(a|s)$ as a density function and taking $\pi_\theta(\cdot|s) = \mathcal{N}\left(\mu(\pi_\theta(\cdot|s)), \sigma(\pi_\theta(\cdot|s))\right)$ as a parameterized normal distribution from which the action $a \sim \pi_\theta(\dot{|}s)$ is sampled. Therefor it possible to acquire the probability $\pi_\theta(a|s)$ of an action a with as in Equation (3.13).

$$\pi_\theta(a|s) = \frac{1}{\sigma(\pi_\theta(\cdot|s))\sqrt{2\pi}} \exp\left(-\frac{1}{2}\frac{(a-\mu(\pi_\theta(\cdot|s)))^2}{\sigma(\pi_\theta(\cdot|s))^2}\right) \tag{3.13}$$

As a result of entropy regularization in each time step both value function v_{π_θ} and action-value function q_{π_θ} become affected and turn into their counterpart $v_{\pi_\theta,H}$ and $q_{\pi_\theta,H}$ in Equation (3.14) and Equation (3.15) below. Note that Equation (3.16) is the recursive version of Equation (3.15).

$$v_{\pi_\theta,H}(s) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T}\gamma^t\left(R_t + \alpha H(\pi_\theta(\cdot|s_t))\right) \mid s_0 = s\right] \tag{3.14}$$

$$q_{\pi_\theta,H}(s,a) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T}\gamma^t R_t + \alpha\sum_{t=1}^{T}\gamma^t H(\pi_\theta(\cdot|s_t)) \mid s_0 = s, a_0 = a\right] \tag{3.15}$$

$$= \mathop{\mathbb{E}}_{s'\sim P, a'\sim\pi_\theta}\left[R_t + \gamma(q_{\pi_\theta,H}(s',a') + \alpha H(\pi_\theta(\cdot|s')))\right] \tag{3.16}$$

This changes subsequently also the original relation between v_{π_θ} and q_{π_θ}. $v_{\pi_\theta,H}$ and $q_{\pi_\theta,H}$ are connected via Equation (3.17).

$$v_{\pi_\theta,H}(s) = \mathbb{E}_{\pi_\theta}\left[q_{\pi_\theta}(s,a)\right] + \alpha H(\pi_\theta) \tag{3.17}$$

Note, that we introduce a new parameter α into $v_{\pi_\theta,H}$ and $q_{\pi_\theta,H}$. This parameter controls the tradeoff between exploration and exploitation. Higher values of α promote more exploration, whereas lower values of α encourage more exploitation in the Soft Actor-Critic algorithm [18]. For more details how to this parameter is used in Soft Actor-Critic please have look into Section 3.3.3.

For simplification we will now refer to the entropy regularized value function $v_{\pi_\theta,H}$ as v_{π_θ} and action-value-function $q_{\pi_\theta,H}$ as q_{π_θ}.

This change also influences the original reinforcement learning problem as stated in Equation (3.10) and converts it into Equation (3.18). Due to the stated relation between regularized value-function and regularized action-value-function as in Equation (3.17) we can make the optimization problem independent from the value-function and get Equation (3.19).

$$\pi_\theta^* = \arg\max_{\pi_\theta} v_{\pi_\theta,H}$$

$$\overset{(3.10)}{=} \arg\max_{\pi_\theta} \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} \gamma^t \left(R_t + \alpha H(\pi_\theta)\right) \mid S_t = s\right] \tag{3.18}$$

$$\overset{(3.17)}{=} \arg\max_{\pi_\theta} \mathbb{E}_{\pi_\theta}\left[q_{\pi_\theta}(s,a)\right] + \alpha H(\pi_\theta)$$

$$\overset{(3.12)}{=} \arg\max_{\pi_\theta} \mathbb{E}_{\pi_\theta}\left[q_{\pi_\theta}(s,a) - \alpha \log \pi_\theta(a|s)\right] \tag{3.19}$$

3.3.3 Algorithm Architecture

In contrast to other actor-critic algorithms like: DDPG [16] or A3C [19] SAC learns additional to a parameterized policy π_θ, two critics in the form of two parametrized q functions Q_{ϕ_0} and Q_{ϕ_1} as well as their corresponding target functions $Q_{\phi_{\text{target},1}}$ and $Q_{\phi_{\text{target},1}}$. The usage of target action-value functions enhances the accuracy of the estimates and contributes to a more stable learning process [2].

Further SAC utilizes a stochastic policy, which means that the policy outputs a parameterized probability distribution over actions instead of directly selecting deterministic actions and optionally adding noise on top. This stochastic nature allows SAC to capture the exploration-exploitation trade-off directly within the policy. The training process in SAC is based on the maximum entropy objective [18]. Its goal is to find a policy that maximizes the cumulative reward Equation (3.1) and the maximum entropy Equation (3.12). Fortunately, we have derived an entropy regularized value-function and action-value-function in Section 3.3.2 and are able to use those definition to derive loss functions for actor and critic based on the maximum entropy objective.

Because we will have a closer look into actual implementation, we now refer to the action-value-function as their parameterized approximations Q_{π_i}.

To **train the parameterized action-value-functions** the objective function (3.20) is defined as the expected squared difference between the action-state-value $Q_{\phi_i}(s, a)$ and its temporal difference target $y(r, s', d)$. Equation (3.20) leads directly to the update equation (3.26) as its derived empirical counterpart.

$$\mathcal{L}_Q(\phi_i, \mathcal{D}) = \mathbb{E}_{\mathcal{D}} \left[\left(Q_{\phi_i}(s, a) - y(r, s', d) \right)^2 \right] \tag{3.20}$$

The temporal difference target target y is defined as in Equation (3.21) with $\hat{a}' \sim \pi\theta(\cdot|s')$. To stabilize the target signal SAC applies the clipped double-Q trick, where it selects the minimum q-value between the two target q-functions $Q_{\phi_{\text{target},i}}$.

$$y(r, s', d) = r + \gamma(1 - d) \left(\min_{j=0,1} Q_{\phi_{\text{target},i}}(s', \hat{a}') - \alpha \log \pi_\theta(\hat{a}'|s') \right) \tag{3.21}$$

This function is employed in the SAC algorithm in Equation (3.25) but with the small difference that all arguments are sampled from a minibatch $\mathcal{B}$.

To **train the policy** we are required to use the reparameterization trick to be able to differenciate the parameterized policy π_θ [20]. This process transforms the action $\hat{a}$ sampled from the policy into the function as specified in Equation (3.22).

$$\hat{a}_\theta(s, \epsilon) = \tanh(\mu(\pi_\theta(\cdot|s)) + \sigma(\pi_\theta(\cdot|s)) \bigodot \epsilon), \ \epsilon \sim \mathcal{N}(0, I) \tag{3.22}$$

Within the reparameterization of an action we also bound it into a finite range of $(-1, 1)$. This has the advantage bounding actions to standardized limits reducing the risk of undesirable or catastrophic action outcomes. Since we squash the actions with tanh we also have to adapt the log-likelihood $\log \pi_\theta(a|s)$ of an action into:

$$\log \pi_\theta(a|s) = \log \pi_\theta(a|s) - \sum_{i=1}^{|\mathcal{A}|} \log(1 - \tanh(\hat{a}_\theta(s, \epsilon))),\ \epsilon \sim \mathcal{N}(0, I)$$

For more details please refer to [2].

To compute the policy loss, as in training the action-value-function approximation, the crucial step involves replacing q_{π_θ} with one of our function approximators Q_{ϕ_i}. SAC utilizes the minimum of the two q_{π_θ} approximators $\min_{i=0,1} Q_{\phi_i}$. Consequently, the policy is optimized based on this minimum action-state-value approximation and therefor make only more conservative estimates.

$$\mathcal{L}_\pi(\theta, \mathcal{D}) = - \mathop{\mathbb{E}}_{s\sim\mathcal{D}, \epsilon\sim\mathcal{N}(0,I)} \left[\min_{i=0,1} Q_{\phi_i}(s, \hat{a}_\theta(s, \epsilon) - \alpha \log \pi_\theta(\hat{a}_\theta(s, \epsilon)|s)) \right] \tag{3.23}$$

Note that we are taking the negative expectation because we want to maximize the expected return and the entropy by using SGD as in Equation (3.27) in Algorithm 2. The different notation can be explained a $\{s, a, r, s', d\}_{\mathcal{B},k}$ stand for the kth element from the minibatch $\mathcal{B}$..

Now lets turn to the entropy regularization parameter α. In general there are two types of Soft-Actor-Critic implementations. One proposed by [14] and implemented in [21] treads α as constant parameter. The optimal α parameter, leading to the most stable and rewarding learning, may vary across different environments, necessitating thoughtful tuning for optimal performance. The second approach how to treat α, proposed by [2], adjusts α constantly during the training process. The optimization criterion is stated as:

$$\mathcal{L}_\alpha(\pi_\theta, \bar{H}) = \mathop{\mathbb{E}}_{a_t\sim\pi_\theta} \left[-\alpha \log \pi_\theta(a_t|s_t) - \alpha\bar{H} \right] \tag{3.24}$$

Equation (3.24) resembles an objective for dual gradient descent because we are trying to minimize α but also the expected difference between the policy entropy and a target entropy $\bar{H}$ as a hyperparameter. In practice this can be translated into a positive gradient if the expected difference is positive so the agent is less exploring as should be and into a negative gradient if the difference is negative so the agent is to focused on exploring. Selecting a target entropy is not as delicate as selecting a fixed α because you are able to read from training results if your environment requires a more or less greedy policy [18].

3.3.4 Advantages

SAC offers several advantages over other actor-critic algorithms. Firstly, the entropy regularization leads to improved exploration, enabling the agent to efficiently explore its environment and discover optimal or near-optimal solutions [14]. Secondly, by encouraging stochastic policies, SAC provides more robust and adaptive policies that can handle uncertainties and variations in the environment.

Moreover, the SAC algorithm exhibits enhanced sample efficiency, meaning that it requires fewer interactions with the environment to learn effective policies [2]. The utilization of two critics Q_{ϕ_0} and Q_{ϕ_1} further contributes to a more stable learning process and can mitigate the issues of overestimation bias, leading to more accurate value estimates [14].

Applications

In robotics, the SAC algorithm has been used to control robots in tasks such as grasping objects, locomotion, and manipulation. The SAC algorithms ability to handle continuous action spaces makes it a suitable choice for robotic control, where fine-grained control is often required [2].

Algorithm 1 Soft Actor Critic

Input: initial policy parameters θ, Q-function parameters ϕ_0, ϕ_1, empty replay buffer $\mathcal{D}$

Set target parameters equal to main parameters $\phi_{\text{target},0} \leftarrow \phi_0$, $\phi_{\text{target},1} \leftarrow \phi_1$

for i in number of epochs **do**

$s \leftarrow$ reset environment

for t number of timesteps **do**

$a \sim \pi_\theta(\cdot|s)$

s', r, $d \leftarrow$ execute a in environment

Store (s, a, r, s', d) in replay buffer $\mathcal{D}$

if d is true **then**

break and reset environment

end if

$s \leftarrow s'$

end for

if $|\mathcal{D}| >$ minimal buffer size **then**

for number of train iterations **do**

sample minibatch: $(s_\mathcal{B}, a_\mathcal{B}, r_\mathcal{B}, s'_\mathcal{B}, d_\mathcal{B}) = \mathcal{B} \leftarrow \mathcal{D}$

compute td target y with $\tilde{a}'_\mathcal{B} \sim \pi_\theta(\cdot|s'_\mathcal{B})$

$$y(r_\mathcal{B}, s'_\mathcal{B}, d_\mathcal{B}) = r_\mathcal{B} + \gamma(1 - d_\mathcal{B}) \left(\min_{j=0,1} Q_{\phi_{\text{target},i}}(s'_\mathcal{B}, \hat{a}'_\mathcal{B}) - \alpha \log \pi\theta(\hat{a}'_\mathcal{B}|s'_\mathcal{B}) \right) \quad (3.25)$$

Update Q-functions for parameters ϕ_i $i \in \{0, 1\}$ using:

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{k \in |\mathcal{B}|} \mathcal{L}_Q(\phi_i, \mathcal{D}) \quad (3.26)$$

Update policy using:

$$\nabla_\theta \frac{1}{|\mathcal{B}|} \sum_{k \in |\mathcal{B}|} \mathcal{L}_\pi(\theta, \mathcal{D}) \quad (3.27)$$

Update α with target entropy H_{target} and $\tilde{a}_\mathcal{B} \sim \pi_\theta(\cdot|s_\mathcal{B})$ using:

$$\nabla_\alpha - \frac{\alpha}{|\mathcal{B}|} \sum_{k \in |\mathcal{B}|} \mathcal{L}_\alpha(\pi_\theta, \bar{H})$$

Update target networks $\phi_{\text{target},i}$ $i \in \{0, 1\}$ with ϕ_i $i \in \{0, 1\}$ using:

$$\phi_{\text{target},i} \leftarrow \rho\phi_{\text{target},i} + (1 - \rho)\phi_i$$

end for

end if

end for

In game playing, the SAC algorithm has been used to train agents to play video games such as Atari [22]. The SAC algorithms ability to balance exploration and exploitation makes it effective in game playing, where agents must learn to navigate complex environments and respond to changing conditions [1].

3.4 Neural Networks

Neural networks are a class of artificial intelligence algorithms inspired by the structure and functioning of the human brain [13].
They consist of interconnected layers of artificial neurons that process and transform data. Neural networks are crucial in modern machine learning and deep learning applications due to their ability to learn complex patterns and representations from data, enabling them to solve a wide range of problems effectively [13].

In this chapter, we will delve into neural networks, looking at their evolution, architecture, training process and diverse applications.

3.4.1 The Rise of Deep Learning

Neural networks were first invented in the 1940s and 1950s as a computational model inspired by the interconnected structure and functioning of the human brain, but their practical development was hindered by limitations in computing power and the lack of large datasets. The resurgence of interest in neural networks in the 2000s can be attributed to two major factors: the development of new algorithms and the availability of large datasets. Before the 2000s, neural networks faced limitations in training and optimization, hindering their effectiveness. However, new algorithms, such as the backpropagation algorithm and variants like stochastic gradient descent, emerged, making it feasible to train deeper networks efficiently. Additionally, advancements in computing power and the availability of vast datasets

facilitated by the internet like image net [23] allowed neural networks to leverage big data for improved learning and performance [24] [25].

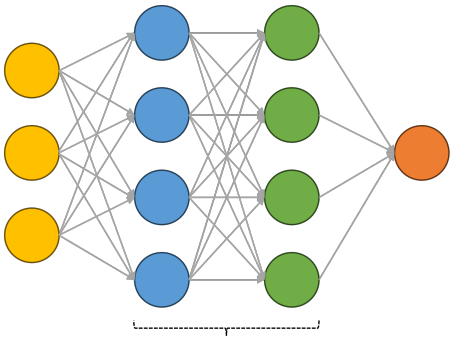
Nowadays neural networks are employed over a vast variety of tasks like image recognition [24], speech recognition [26], natural language processing [27] or robotics [28].

3.4.2 Neural Network Architecture

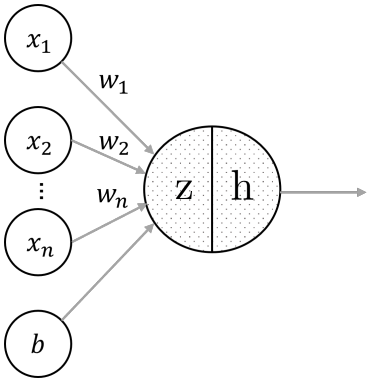
Neural networks are a type of machine learning model inspired by the structure and function by the cell type of neurons. It consists like their biological model, of interconnected layers of individual units, called neurons [13].

The basic architecture of a neural network as in Figure 3.4a, includes an input layer (yellow), one or more hidden layers (blue and green), and an output layer (red). The input layer receives the input data, which is then passed through the hidden layers before producing the output. Because the flow of information is only in the forward direction we call this basic architecture a feed forward neural network. The number of nodes in each layer and the connections between them are configured by the architecture of a network.

If we look at each individual neuron as in Figure 3.4b, we can observe that each neuron is designed in the same way. First it calculates a weighted sum z of its inputs x, the corresponding weights w and bias b, before passing it into an activation function h and finally passing the information to the next neuron. The activation function is designed to introduce nonlinearity into the model, allowing it to capture complex relationships between variables. Without a nonlinear activation function the whole network would be a single linear combination of its inputs and weights. Common activation functions include the sigmoid function [29], the hyperbolic tangent function [29], and the rectified linear unit (ReLU) function [30] as in Figure 3.5.

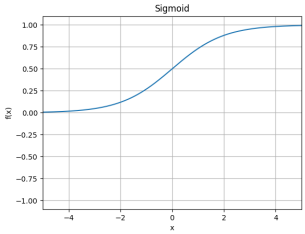


(a) Schematic drawing of a feed forward neural network.

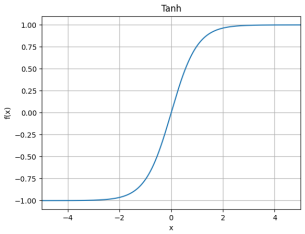


(b) Schematic drawing of a single neuron.

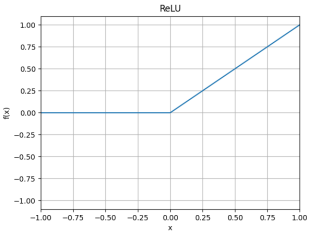
Figure 3.4: The figure shows the schematic drawing of a feed forward neural network and a neuron



(a) Sigmoid

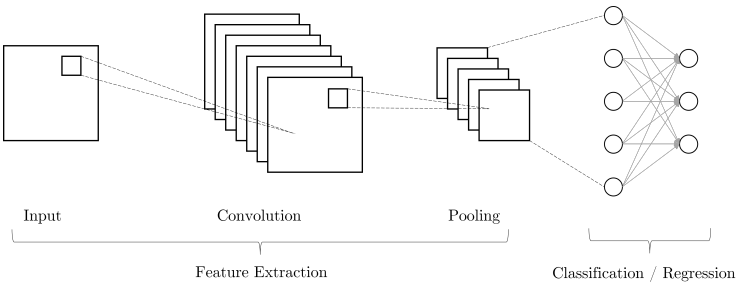


(b) hyperbolic tangent function

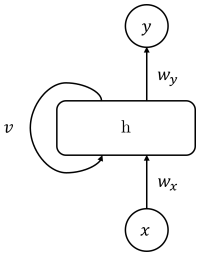


(c) rectified linear unit

Figure 3.5: common activation function for neural networks.



(a) schematic drawing of convolutional neural network



(b) schematic drawing of a recurrent neural network

Figure 3.6: architecture of more advanced neural networks like the convolutional and the recurrent neural network

There are several types of neural network architectures but most of them are based on following types:

- **Feedforward networks** [31] as in Figure 3.4a are the simplest type of neural network, consisting of a series of layers that process information in a single direction.

- **Convolutional networks** [32] as in Figure 3.6a are designed for image processing tasks and use convolutional layers to identify patterns and features within images.

- **Recurrent networks** [33] as in Figure 3.6ballow information to be passed between nodes in a cyclical manner, making them suitable for processing sequential data. The development of Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) architectures further improved RNNs ability to model long-range dependencies.

3.4.3 Train Neural Networks

Neural networks are trained using techniques such as backpropagation [34] and stochastic gradient descent [35] [36] as presented in Algorithm 2. Backpropagation is an algorithm for calculating the gradient $\hat{g}$ of an optimization criterion with respect to the weights of the network. The gradient can then be used to update the weights and therefor improve the performance of the model with respect to the optimization function. A we can see in Algorithm 2 stochastic gradient descent minimizes the error presented by the optimization criterion by iteratively adjusting the weights based on randomly selected subsets of the training data.

Selecting a random subset of training data has a couple of advantages [37].

Algorithm 2 Stochastic gradient descent

Input: Learning rate schedule: $\epsilon_1, \epsilon_2, \ldots$. Initial parameter θ
$k \leftarrow 1$
while stopping criterion not met **do**
Sample a minibatch of m examples from training set $\{x^{(1)}, \ldots, x^{(m)}\}$ with corresponding targets $y^{(i)}$.
Compute gradient estimate:
$\hat{g} \leftarrow \frac{1}{m} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$
Apply update:
$\theta \leftarrow \theta - \epsilon_k \hat{g}$
$k \leftarrow k + 1$
end while

- **Computational Efficiency**: In one epoch we are computing the gradient only on a small subset. We do not have to pass the complete dataset through the network to get a sufficient gradient.

- **Improved Generalization**: Because each minibatch contains a different composition of data points from the dataset the gradient signal is also varying. These variations are improving the generalization of the model with respect to unseen data.

- **Avoiding Local Minima**: One of the risks during the training-process of a neural network are local minima. By introducing randomness through mini-batch sampling, SGD can escape local minima more easily and continue to explore the parameter space, increasing the chances of finding a global minimum.

- **Parallel Processing**: Using mini-batches allows parallel processing during training.

3.4.4 Applications of Neural Networks

Neural networks are widely used in real-world applications like in computer vision [24] [25], marketing [38] or medical applications [39]. Particularly in combination

with reinforcement learning they are successfully used in robotics [40], for object recognition [41] and manipulation [42], in gaming for complex decision-making [1], in biochemistry for predicting protein foldings [43] or even in computer science to write even more performant algorithms [44].

3.5 Variational Autoencoder

Variational autoencoders (VAEs) and conditional variational autoencoders (CVAEs) are types of deep generative models that are used for unsupervised learning [5]. Unsupervised learning is a type of machine learning where the model is not given labeled data for training [45]. Instead, the model is tasked with finding patterns or structure in the data on its own. The goal of unsupervised learning is often to find hidden relationships or groupings within the data that can be used for further analysis or decision-making VAEs and CVAEs are important because they can learn to generate realistic and diverse samples from complex high-dimensional data distributions, such as images or audio [46].

3.5.1 Autoencoders

Lets start with a Autoencoder. An Autoencoder $f = q \circ p$ consists of two parameterized function approximators in series, one encoder q_ϕ and one decoder p_ψ. The encoder maps the high dimensional input $x \in \mathbb{X}$ into a lower dimensional latent space $z \in \mathbb{Z}$, $z = q(x)$. The latent vector z is typically a lower dimensional representation of the input x. In the second stage, we map the latent vector z back into the high dimensional features space $\mathbb{X}$, $d : \mathbb{Z} \to \mathbb{X}$. This should optimally reconstruct the given input $\hat{x} = p(z) = q(p(x))$.

Because Autoenders are designed to learn a compact determine representation in the latent space it follows that similar input values x should correspond to similar latent vectors z and similar latent vectors z correspond to similar outputs $\hat{x}$.

Traditional Autoenders simple and effective design comes along with some limitations [5]:

- Overfitting: Because of the deterministic mapping into the latent space and from the latent space back to the feature space traditional Autoencoders are prone to suffer from overfitting and can lack of regularization which could lead to poor performance on unseen data.

- Incomplete or noisy data: Due to the focus on only reconstructing the input, Autoencoders are not well suited for incomplete or noisy data. One reason could be the lack of disentanglement in the latent space, which means that different dimensions correspond to different underlying features in the input data distribution.

- Lack of Probabilistic Interpretation: One limitation of deterministic autoencoders is that they cannot naturally capture uncertainty or variations in the data. In real-world data, there is often inherent uncertainty or variability, and deterministic autoencoders struggle to represent this effectively. To address this limitation, probabilistic autoencoders like Variational Autoencoders (VAEs) are introduced. VAEs model uncertainty by assuming that the latent space (the encoded representation) follows a probabilistic distribution, typically a Gaussian distribution. This allows VAEs to provide a probabilistic interpretation of the data, meaning that for a given input, the model can generate multiple plausible representations or outputs along with associated probabilities.

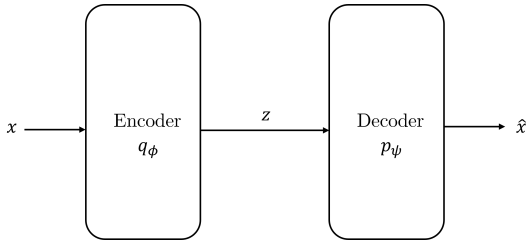


Figure 3.7: schematic drawing of a Autoencoder

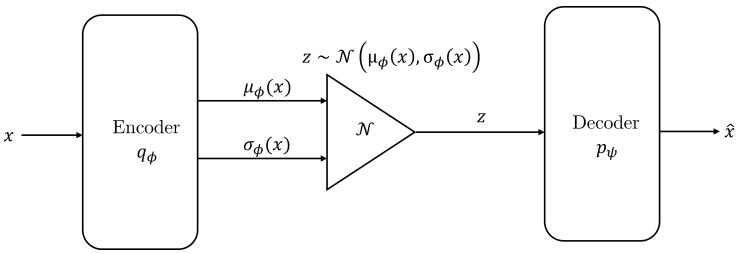


Figure 3.8: schematic drawing of a Variational Autoencoder

3.5.2 Variational Autoencoders

A Variational Autoencoder is similar to an Autoencoder but also with some key changes. Similar are the basic setup as a generative unsupervised learning model and the underlying encoder decoder structure. The key difference lays in the in the latent space between the encoder and decoder. Instead of having a fixed deterministic latent space VAEs operate on a probabilistic latent space [5].

The encoder network denoted as a approximated parameterized posterior distribution $q_\phi(z|x)$, is a conditional probability distribution in Bayesian inference with the probability of the latent variable $p(z)$ (the prior) and the evidence $p(x)$:

$$q(z|x) = \frac{p(x|z)p(z)}{p(x)} \tag{3.28}$$

$$p(x) = \int p(x|z)p(z)dz \tag{3.29}$$

In the context of a Variational Autoencoder, calculating $q(z|x)$ is not possible. As we can see in Equation (3.29) accessing the evidence as the probability of the observed data is not possible because we have to integrate over all possible values of the latent variable z. Therefor we approximate the posterior distribution with a parameterized conditional distribution $q_\phi(z|x)$. $q_\phi(z|x)$ as known as the encoder network, parameterized with ϕ, hereby maps input data x to latent parameters like mean and variance for a gaussian distribution, of the latent distribution with random variable z [5].

On the other hand, the decoder network takes samples from the latent space with latent variables z and reconstructs the data from these samples, generating a parameterized probabilistic distribution over the data given the latent variables $p_\psi(x|z)$ [5].

As mentioned before the objective in training a Variational Autoencoder is to find parameters ϕ and ψ and matches best the true posterior distribution. In order to do so we can use the Evidence Lower Bound as the objective function. We go a bit deeper into the fundamentals of the Evidence Lower bound in Section 3.5.4.
During training, the VAE aims to maximize the ELBO by iteratively adjusting its parameters ϕ and ψ using backpropagation. The encoder network maps the input data to a distribution over latent variables, while the decoder network reconstructs the data from the sampled latent variables. The reparameterization trick similar to SAC in Section 3.3 is employed to ensure differentiability during backpropagation through the stochastic sampling process [5].

3.5.3 Conditional Variational Autoencoders

A Conditional Variational Autoencoder is an extension of the VAE architecture that takes into account conditional information c. As we can see in Figure 3.9 in a Conditional Variational Autoencoder, both the encoder and decoder are modified to accept an additional input that represents the conditional information [47].

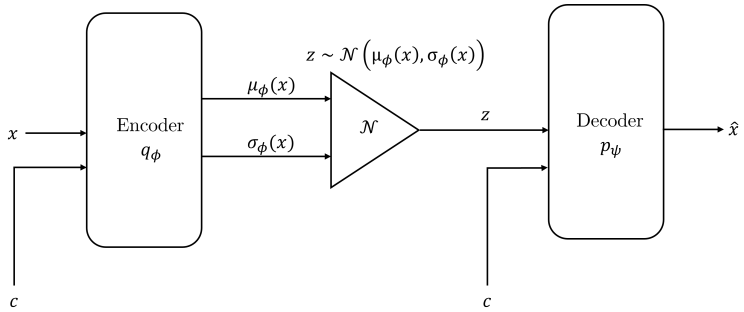


Figure 3.9: schematic drawing of a conditional Variational Autoencoder

The role of the conditional input in the encoder is to help the model capture the conditional dependencies between the input data and the class label. So the conditional information could be for instance a label encoding or a set of labels that describe the class or category to which the input belongs. The encoder must learn to map both the input data and the conditional input to a meaningful latent space representation that captures the relevant information for the generation process. In the decoder, the conditional input is used to guide the generation process and ensure that the generated samples are representative of the specified class [47].

Similar to Variational Autoencoders the objective function to train the encoder and decoder is the Evidence Lower Bound. But there have to be a couple of changes to make the individual ELBO components suitable to take the conditional information c. The final objective can be observed in Equation (3.33).

3.5.4 Evidence Lower Bound

The Evidence Lower Bound [5] (ELBO) is a fundamental concept in the training of Variational Autoencoders (VAEs) It is derived from the principle of variational inference and represents a lower bound on the log-likelihood of the data given the models parameters. Maximizing the ELBO is equivalent to minimizing the Kullback-Leibler (KL) divergence [5] between the true posterior distribution $q(z|x)$ over latent

variables z and the approximated posterior distribution $q_\phi(z|x)$ used in the VAE.

$$\mathrm{KL}(q_\phi(z)||p_\psi(z|x)) \geq \mathcal{L}_{\mathrm{ELBO}}(x,z) = \mathbb{E}_{z\sim q_\phi(\cdot|x)}\left[\log p_\psi(x|z)\right] - \mathrm{KL}\left(q_\phi(z|x)\|p(z)\right) \tag{3.30}$$

Further we will discuss the individual components of the Evidence lower bound

- **Posterior Distribution**: In general the posterior distribution $q(z|x)$ is considered as the probability of the latent variable z given the observed data x. Specific for Variational Autoencoder it can be computed from Bayesian Inference as in Equation (3.28) which is as mentioned in Section 3.5.2 not feasible is therefor approximated.

- **Likelihood**: The likelihood function $p(x|z)$ or conditional likelihood in the context of probabilistic models is the probability distribution of the observed data x given the latent variables z. Dependent on the observed data it is possible to chose from a set of different probability distributions. Examples are a multivariate gaussian distribution for continuous data, a Bernoulli distribution for binary data or categorical distribution for discrete data with multiple categories.

- **Prior Distribution**: The prior distribution $p(z)$ is a probabilistic distribution that represents the initial belief or assumption about the latent variables z in Bayesian inference. Additional $p(z)$ serves in Variational Autoencoder as a regularizer during training as it encourages the VAE to learn meaningful and smooth latent representations in the latent space. Most popular choice for the prior distribution is the standard normal distribution $\mathcal{N}(0, I)$.

- **Evidence**: p(x) represents the evidence, also known as the marginal likelihood or model evidence, in the context of Bayesian statistics. It is the probability of

the observed data x given a particular statistical model. The evidence serves in Equation (3.29) as a normalization constant, ensuring that the posterior distribution $q(z|x)$ over model parameters integrates to 1. It quantifies how well the model, with its specific set of parameters, explains or fits the observed data.

- **Reconstruction Loss**:

$$\mathbb{E}_{z\sim q(\cdot|x)}\left[\log p(x|z)\right] \approx \mathbb{E}_{z\sim q_\phi(\cdot|x)}\left[\log p_\psi(x|z)\right] \tag{3.31}$$

Therm (3.31) measures the similarity between the reconstructed data and the original input. Described in words it is the expected log-likelihood of the data given the latent variables, which is computed by sampling from the approximating distribution $q_\phi(z|x)$ and evaluating the likelihood $p_\psi(x|z)$ using the decoder network.

- **KL Divergence Loss**:

$$\mathrm{KL}\left(q(z|x)\|p(z)\right) \approx \mathrm{KL}\left(q_\phi(z|x)\|p(z)\right) \tag{3.32}$$

This term quantifies the difference between the approximated parameterized posterior distribution $q_\psi(z|x)$ over latent variables z and a chosen prior distribution $p(z)$. As mentioned before it is a popular choice to use a standard normal distribution $p(z) = (N)(0, I)$ as the prior distribution. The KL divergence encourages the latent variables to be close to the prior distribution, promoting regularization and preventing overfitting.

The basic notation of the Evidence Lower Bound for Variational Autoencoders as in Equation (3.30) can be easily extented for an Conditional Variational Autoen-

coder:

$$\mathcal{L}_{\text{C, ELBO}}(x, z, c) = \mathbb{E}_{z \sim q_\phi(\cdot|x,c)} \left[\log p_\psi(x|z, c)\right] - \text{KL}\left(q_\phi(z|x, c) \| p(z|c)\right) \tag{3.33}$$

By optimizing the ELBO, VAEs effectively learn to approximate the true posterior distribution and generate meaningful latent representations of the data, enabling various tasks such as data generation, interpolation, and denoising within a Bayesian framework [5].

3.5.5 VAEs and CVAEs in Practice

VAEs and CVAEs have been successfully applied to a wide range of real-world applications. In image generation [5], VAEs have been used to generate novel images of faces, objects, and scenes [5]. Similarly, CVAEs have been used for conditional image generation, allowing for the generation of images based on specific attributes or classes. In text generation, VAEs have been used to generate natural language sentences and paragraphs.

VAEs and CVAEs can also be used for data compression [48] and denoising [49]. By learning a compressed representation of the input data, VAEs and CVAEs can reduce the dimensionality of the input space while preserving important features. Similarly, by learning to reconstruct the original input from noisy or corrupted data, VAEs and CVAEs can be used for denoising and data restoration.

One of the advantages of VAEs and CVAEs is their ability to learn a continuous latent representation of the input data. This allows for easy manipulation and exploration of the latent space, enabling applications such as image editing and style transfer [46].

However, VAEs and CVAEs have some limitations. The generated samples may not be as sharp or detailed as those produced by other generative models such as GANs [50].

Additionally, the trade-off between the reconstruction loss and the KL divergence term can be difficult to balance, potentially leading to overfitting or underfitting [51]. Nonetheless, VAEs and CVAEs remain a popular and powerful tool for generative modeling and data compression.

3.6 Kinematics

Kinematics is the study of motion, specifically the description and analysis of the position, velocity, and acceleration of objects or systems [52] [53] without considering the forces causing the motion. It focuses on understanding the spatial relationships and geometrical aspects of moving objects. The following sections provide an insight into forward and inverse kinematics for kinematic chains. Those two concepts are often used in robotics, animation, virtual reality or even protein folding.

3.6.1 Forward kinematics

Forward kinematics is a concept from robotics that involves determining the position and orientation of an end-effector, like a gripper of a robot arm, based on the joint angles and geometric parameters, like segment length, of the system. It provides a mathematical model for mapping the joint angles to the end-effector position in order to understand the overall configuration and motion of the robot. Equation (3) describes a forward kinematics implementation.

Algorithm 3 Forward Kinematics

Input: Current joint angles q, Segment Length l.
Define origin position in 2D space: $p \leftarrow (0, 0)$
for each i in $[0, \ldots, N-1]$ **do**
    Update position
    $p_0 \leftarrow p_0 + \cos(q_i) * l_i$
    $p_1 \leftarrow p_1 + \sin(q_i) * l_i$
end for

Throughout this thesis this function is used with a constant segment length $l = \{1\}^N$ therefor it is referred to as in Equation (3.34) which maps an angle configuration q into the end-effector position in 2D space.

$$FK : \mathbb{R}^N \rightarrow \mathbb{R}^2 \tag{3.34}$$

3.6.2 Inverse kinematics

Inverse kinematics (IK) is a fundamental problem in robotics, animation or virtual reality that involves finding a required joint angle configuration or positions to reach a desired end-effector position and optionally a desired orientation [54]. It plays a crucial role in controlling the motion and manipulation of robotic systems, enabling them to interact with the environment and perform complex tasks. In this section you will find a brief summary of existing approaches, a deeper explanation of the Cyclic Coordinate Descent algorithm and the description of the used inverse kinematics RL environment.

Existing Approaches to Solve Inverse Kinematics

Numerous approaches have been proposed to solve the inverse kinematics problem. These approaches can be broadly categorized into:

- analytical methods [55]: rely on geometric methods to derive closed form solutions.

- numerical methods [56]: iteratively approximate the joint angles that satisfy the desired end-effector position and orientation.

- heuristic methods [56]: iteratively propagate positions along the kinematic chain to converge on the desired end-effector position.

- sampling-based methods [56]: randomized search strategy to explore the joint space and find feasible solutions to the inverse kinematics problem

Cyclic Coordinate Descent

Cyclic Coordinate Descent (CCD) [56] is a popular numerical method for solving inverse kinematics. It is an iterative algorithm that adjusts the joint angles of a robotic system one at a time, from a base joint to the end-effector, in order to align the end-effector with the desired target position.

Algorithm 4 Cyclic Coordinate Descent Pseudo Code

Input: Current joint angles q, Desired end-effector position p_{target}.
while unitl convergence **do**
  for each i in $[N-1, \ldots, 0]$ **do**
    Calculate vector from current joint position to end-effector position:
    $v_{\text{current}} \leftarrow p_{N-1} - p_i$
    Calculate vector from current joint position to target position:
    $v_{\text{target}} \leftarrow p_{\text{target}} - p_i$
    Calculate the necessary rotation to align v_{current} with v_{target}:
    $\delta q_i \leftarrow \text{angle\_between}(v_{\text{current}}, v_{\text{target}})$
    Update joint angle:
    $q_i \leftarrow q_i + \delta q_i$
  end for
end while

The algorithm works by iteratively updating the joint angles based on the discrepancy between the current and desired end-effector positions (p_{target}). At each iteration, CCD focuses on a single joint and adjusts its angle to minimize the positional error. By sequentially updating the joint angles in a cyclic manner, CCD aims to converge towards a solution that satisfies the desired end-effector position.

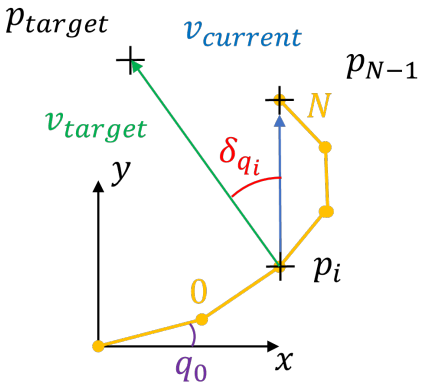
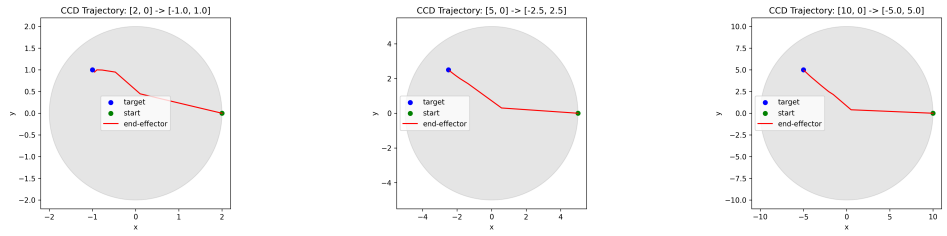


Figure 3.10: CCD geometry: Shown is the geometry of one step in Algorithm 4. p_{target} is the target position. p_i is the 2D position of each joint. Not that we start with the first joint outside the origin since the base position is fixed to the origin.

As illustrated in Algorithm 4 and Figure 3.10 in each iteration, CCD calculates the vector from the current joint position p_i with p_i as the position of the ith joint with $i \in \{0, \ldots, N-1\}$, to the end-effector position (v_{current}) and the vector from the current joint position to the target position (v_{target}). By finding the rotation necessary to align v_{current} with v_{target}, represented as δq_i, the algorithm updates the joint angle accordingly. This process is repeated for each joint in the kinematic chain until convergence is achieved.

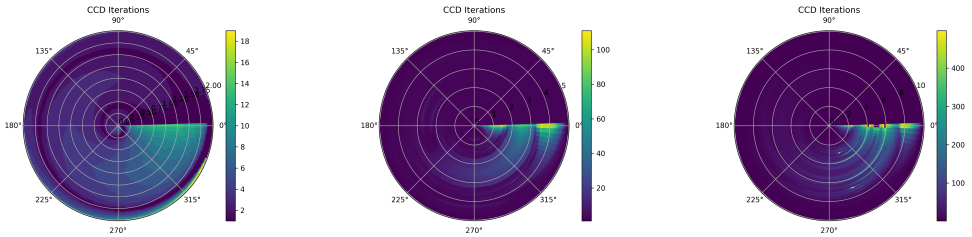
This afore mentioned strategy can be reviewed in Figure 3.11. Here we can see that the strategy appears to move closer and closer to the origin before heading towards the target position in blue.

That CCD needs different amounts of while iterations depending on start and end position can be observed in Figure 3.12. The heatmap plots the different amounts of while iterations needed to get from a constant start position at $p = [N, 0]$ with all angles equal 0, to a desired target position. Interesting to see are the patterns emerging in those heatmaps where target positions counter clockwise of the start positions are faster to solve than target positions clockwise to the start positions.



(a) $N = 2$ with 5 iterations **(b)** $N = 5$ with 11 iterations **(c)** $N = 10$ with 64 iterations

Figure 3.11: Trajectory to reach from start position at $[N, 0]$ to target position at $[-\frac{1}{2}N, \frac{1}{2}N]$. The two scattered dots resemble the start end effector position (green) and the target position (blue). The red line is the trajectory of the end effector after each while iterations as in Algorithm 4.



(a) $N = 2$ **(b)** $N = 5$ **(c)** $N = 10$

Figure 3.12: Heatmaps for iterations needed to reach from a start position at $[N, 0]$ to target positions at $p_{(i,j)} = [\cos(\omega_i), \sin(\omega_i)] \cdot \Re_j, \omega = [0, \ldots, 2\pi), \Re = [0, \ldots, N]$

4 Methodology

In this section we are going to explain the research idea and introduce the inverse kinematics benchmark environment. Further we will dive a bit deeper into the techniques how to create a dataset and explain the used criterions for the latent models. We are going to conclude this chapter by providing a short introduction into the software written and used tools.

4.1 Research Idea

The primary research idea of this thesis is to pursue a transformation from the RL environment action space $\mathcal{A}$ into a lower-dimensional latent action space $\mathcal{A}_L$. This latent action space will align with the latent space between the encoder and decoder of a VAE model or the feature space of a feed-forward-neural network. By doing so, we aim to enable RL agents to effectively explore and learn within a more compact and smoothened action representation. An schematic drawing can be revisited at Figure 4.1

We propose two potential approaches for achieving this reduction in dimensionality: VAEs and supervised models.

In the VAE approach, a conditional or unconditional generative model is employed to learn a latent representation of the action space. By training the VAE on actions from an expert or on state target combinations to emphasis a solution which is independent

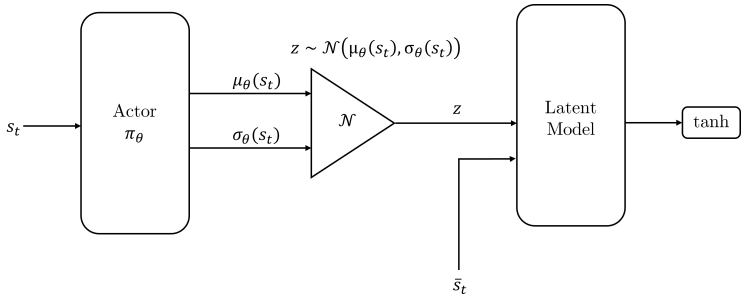


Figure 4.1: Schematic drawing of how the models would be chained together. The laten model will be replaced by either the decoder of a VAE, a CVAE or a supervised feed forward network. Note that we adapt the additional information $\bar{s_t}$ by the needs of the individual models.

from an expert, we aim to capture the underlying structure and patterns within the action space from the RL environment. This latent representation can potentially offer a more concise and informative representation of the actions, encourage more efficient learning and exploration for RL agents.

Alternatively, the supervised model approach involves training a supervised learning model, such as a neural network, to directly transform the a defined lower dimensional latent action into the high-dimensional action space. This transformation is learned based on labeled examples of actions and their corresponding latent representations. In the conducted experiments the lower dimensional-action space is just the desired action outcome. By leveraging supervised learning techniques, we aim to find a mapping between state information and desired action outcome to the RL environment action space, allowing RL agent to operate effectively on a higher level within the reduced-dimensional latent action space.

4.2 RL Environment

To apply RL in general you need an environment the agent can send actions to and receives feedback as discussed in Section 3.1. As previously introduced the key

problem we are targeting is inverse kinematics of a robot arm. This problem turn out to be suitable because:

- the action space is scalable by simply adding additional joints to the robot arm
- it can be simplified into a 2D space with a fast and reliable implementation
- it is expandable. You are always able to extend the environment with additional constrains like objects the robot has to navigate around or joint angle constrains.

In this section we are going to present the a novel RL-Environment for bench-marking the performance of different algorithms to solve inverse kinematics for a robot arm with N many joints and subsequently N many segments with lengths $l \in \mathbb{R}^N_{>0}$ in 2D space. For simplicity reasons l is constant with $l = \{1\}^N$ as a vector with length N and filled with ones. The environment is embedded into the Farama Gymnasium framework [57].

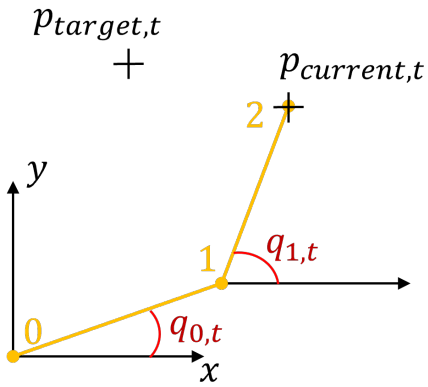


Figure 4.2: Schematic drawing of the individual state space components.

4.2.1 State Space

The state space $\mathcal{S} \subset \mathbb{R}^{4+N}$ for this environment consists of three main building blocks.

- **goal information** $p_{\text{target},t} \in \mathbb{R}^2$: This vector provides information where to move the end-effector. This position is lies always in for the robot arm, reachable distance. Mathematical speaking: $||p_{\text{target}}||_2 \leq \sum_{i=1}^{N} l_i$

- **state position** $p_{\text{current},t} \in \mathbb{R}^2$: This vector contains the current end-effector position in 2D space and should help the agent to understand where the end-effector is placed and how an action has influenced the current end-effector position. Because the current position is attached to the robot arm: $||p_{\text{current}}||_2 \leq \sum l$.

- **joint angles** $q_t \in [0, 2\pi^N)$: This vector contains information about the joint angle configuration at time t. Note that each joint angles describes the delta angle between the joint direction and a horizontal line from origin in positive x direction.

A state $s_t \in \mathcal{S}$ at time t has for all t the same composition

$$s_t = (p_{\text{target},t}, p_{\text{current},t}, q_t) \tag{4.1}$$

To refer to individual parts from index i to j of a state with: $s_{t,(i,j)}$. Therefor we can extract the individual parts with:

- $p_{\text{target},t} = s_{t,(0,1)}$

- $p_{\text{current},t} = s_{t,(2,3)}$

- $q_t = s_{t,(4,N+4)}$

4.2.2 Action Space

The action space $\mathcal{A} \subseteq \mathbb{R}^N$ for this particular environment is continuous and contains all possible joint angle configurations for a robot arm with N joints.

A generated action $\hat{a}$ from the agent is sent to the environment. Inside the environment the incoming action is added on top of the current state angles q_t. To ensure the constrains of $q_{t+1} \in [0, 2\pi)^N$ we take the signed remainder of a division by 2π to write into q_{t+1}:

$$q_{t+1} = (q_t + \hat{a}) \ \% \ 2\pi$$

Subsequently after updating the state angles the current end-effector position get updated by a forward kinematics call on q_{t+1}:

$$p_{\text{current},t} = \text{FK}(q_{t+1})$$

4.2.3 Reward Function

The reward function $R : \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ as in Equation (4.2) for the conducted experiments and this environment aims to minimize the distance between the current end-effector position $p_{\text{current},t}$ and the current target position $p_{\text{target},t}$. The current end-effector position is calculated by the forward-kinematics function on state-angles q_t plus action a_t. The target position is sampled at the beginning of an episode and stays constant throughout the episode until completion or time limit is reached.

$$R(s_t, a_t) = -||FK(s_{t,(4,\ldots,N+4)} + a_t) - s_{t,(0,1)}||_2 \tag{4.2}$$

If you want to normalize the reward function you have to divide R by N.

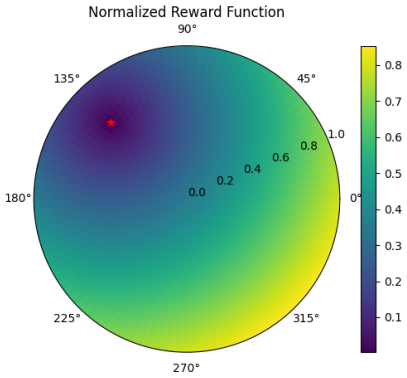


Figure 4.3: Normalized Reward function with $q \in [0, 2\pi]^N$ as element of the whole space and $p_{\text{target}} = [-0.5, 0.5]$. The target position p_{target} is marked with a red start.

In Figure 4.2 we can see the linear gradient for any current position towards the target position at $[-0.5, 0.5]$.

4.3 Dataset Creation

To realize the idea of joining a latent model to the actor network of SAC we do need a dataset to train such latent model. A dataset should contain environment state information as well as possible actions. Consistent over all experiments with the VAE and Supervised Model sizes of the datasets are consistent:

- train: 10.000 data points
- validation: 2000 data points
- test: 1000 data points

4.3.1 Uniform Sampling

The vanilla sampling algorithm for sampling a dataset is to sample state angles q_t and actions a_t from a uniform distribution

$$q_t, a_t \sim \mathcal{U}^N_{[0,2\pi)} \tag{4.3}$$

and the target position with

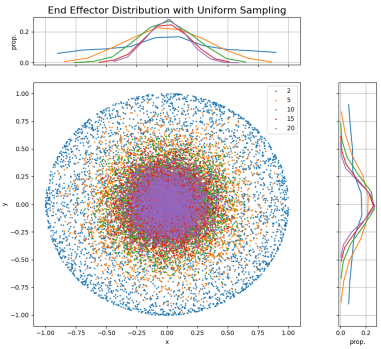
$$\begin{aligned} p &= [\cos(\beta), \sin(\beta)] \cdot r \\ \beta &\sim \mathcal{U}_{[0,2\pi)} \quad r \sim \mathcal{U}_{[0,N]} \end{aligned} \tag{4.4}$$

While observing $p_{\text{current},t}$ in Figure 4.4a we can observe that with an increasing number of joints the scattered dots are forming a two dimensional gaussian distribution around the origin. As we can see in Figure 4.4b the standard deviation is exponentially decreasing.

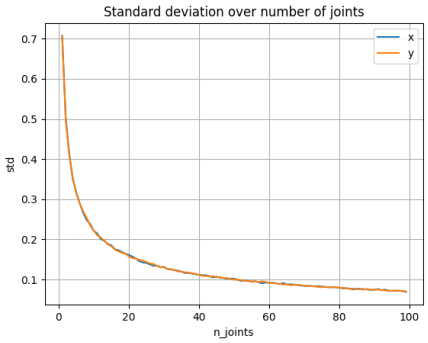
In our application this makes the vanilla algorithm not suitable because we are not covering the observation space close to the maximum reach of the robot arm with more than 2 joints.

4.3.2 Expert Guidance

The problems encountered with the vanilla sampling algorithm in Section 4.3.1 lead to the development of separate sampling algorithm which involve guidance by an expert.



(a) Scatter plot of different end effector positions generated by applying forward kinematics on uniformly sampled angles. The end effector positions are normalized by their corresponding number of joints. This leads to the constrain that all positions have a maximum distance to the origin of 1. For each number of joints we sample 5e3 angles. The upper and right plot are describing the desnsity function of an end-effector position.



(b) Standard deviation of end effector positions in x and y direction with repsect to the number of joints for a robot arm. For each number of joints we sample 1e4 angles.

Figure 4.4: Properties of dataset with actions sampled from a uniform distribution.

Algorithm 5 Expert Guided Dataset Creation

Input: number of joints: N, number of samples: K.
$\mathcal{D} \leftarrow \{\}$ Dataset
for $i \in [0, \dots, N-1]$ **do**
  $p_{\text{current}} \leftarrow$ (4.4) sample a position in 2D space with Equation (4.4)
  $q_t \sim \mathcal{U}$ sample initial arm angles from a uniform distribution as in Equation (4.3)
  $q_t \leftarrow IK(q_t, p_{\text{current}})$ solve IK for p_{current} and with q_t as start
  $p_{\text{target}} \leftarrow$ (4.4) sample a position in 2D space with Equation (4.4)
  $a_t \leftarrow IK(q_t, p_{\text{target}}) - q_t$ IK for the target position
  $\mathcal{D}_i \leftarrow ((p_{\text{target}}, p_{\text{current}}, q_t), a_t)$
end for

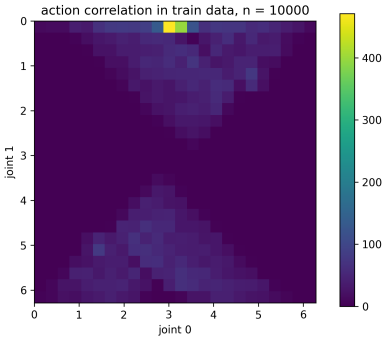
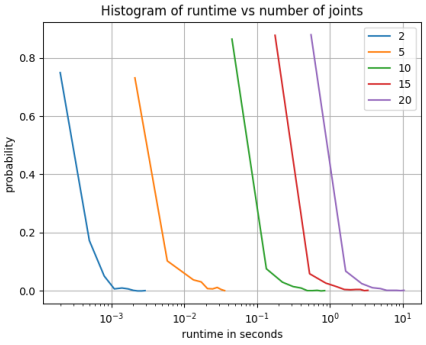


Figure 4.5: Plotted is the state angle correlation of two joints as a 2D histogram. Each bucket can be translated to the amount of angles in one 3D bucket b with limits $b_{0,0}, b_{0,1}, b_{1,0}, b_{1,1}$. Mathematical speaking: $|b| = |\{q_t | q_t \in [0, 2\pi)^2 \wedge q_{0,t} \in [b_{0,0}, b_{0,1}] \wedge q_{1,t} \in [b_{1,0}, b_{1,1}]\}|$

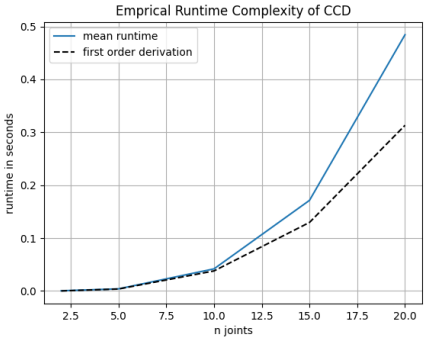
The clear advantage from this algorithm is that we ensure now that p_{target} and p_{current} are uniform with respect to direction and distance to origin.

On the other hand we are now bound to the actions the IK-solver, in our application CCD as in Algorithm 4, is providing. This could lead to unpredicted behavior of the RL agent if we leaf the covered part of q_t as shown in Figure 4.5. Another disadvantage is that this algorithm is now much slower compared to the vanilla sampling algorithm.

To have a more detailed look into the runtime complexity we focus in the runtime complexity of CCD as this is the slowest part of Algorithm 5. We can observe that the runtime of each CCD run is different since CCD belongs to the family of iterative inverse kinematics solver and the runtime is either limited by a given maximum number of iterations or a break condition depending on how close we are to the goal. In Figure 4.6a we can see that all probability density functions have a similar shape but with different means. Since we are interested in the runtime complexity of CCD we plotted the mean runtime of each density function in Figure 4.6b. Here we can observe that the runtime complexity of CCD is at least $\mathcal{O}(N^k)$ with $k \geq 3$ since the first order empirical derivation is still at least quadratic. Further derivations



(a) continuous desnsity function of collected runtimes of CCD over different amount of joints. For each joint we sampled 1000 differnt start conditions, (target, angles) for CCD to solve inverse kinematics. Note that the x axis is in log scale.



(b) plotted is the mean runtime for 1000 runs of CCD over the number of joints

Figure 4.6: Runtime complexity CCD

would be not plausible because with each derivation we would decrease the number of datapoints by one and therefor having no points left anymore.

Another interesting point of view is the correlation between resulting angles from CCD. Since we are limited to maximal two dimensional plot we can only plot a heatmap to show the correlations between state joint actions as in Figure 4.5. Important to note is that the actions are already in environment format and therefor are absolute angles with respect to a horizontal line in positive x direction. We can clearly see that CCD does not use the whole space but only focuses on a rhombus shaped area.

4.4 Latent Criterion

In this section we will cover the employed loss functions to train the latent models. As discussed in Section 3.5.2 we employ the Evidence Lower Bound as the objective function to train Variational Autoencoder. To fit our problem of inverse kinematics we tried out 3 different reconstruction loss function.

In this section we will explain the tangible KL divergence as well as the individual reconstruction loss functions.

4.4.1 Kulback Leiber Divergence

The Kulllback Leiber divergence first published by Solomon Kullback and Richard Leiber in 1951 [5] is mathematical measure of how a probability distribution P differs from a second distribution P' and is denoted as $D_{\mathrm{KL}}(P||P')$. Inside the ELBO it functions as a regularization element between the latent distribution $p_\phi(z|x)$ and a target distribution $p(z)$. Throughout this thesis we take the standard normal distribution as the target distribution, $p(z) = \mathcal{N}(0, I)$. Since we implement a parameterized gaussian as the latent distribution due to the nature of continuous input data, $p_\phi(z|x) = \mathcal{N}_\phi(\mu_\phi(x)|\ \Sigma_\phi(x))$. Therefor we can calculate the Kullback Leiber Divergence as:

$$\begin{aligned} D_{\mathrm{KL}}(\mathcal{N}_\phi(\mu_\phi(x)|\Sigma_\phi(x))||\mathcal{N}(0,I)) &= \frac{1}{2}\left(\mu_\phi(x)^T\mu_\phi(x) + tr\left(\Sigma_\phi(x)\right) - K - \log|\Sigma_\phi(x)|\right) \\ &= \frac{1}{2}\left(\mu_\phi(x)^T\mu_\phi(x) + \sum_{i=1}^{K}\Sigma_\phi(x)_{ii} - K - \log\left(\sum_{i=1}^{K}\Sigma_\phi(x)_{ii}\right)\right) \end{aligned}$$

4.4.2 Reconstruction Loss

Inside the Evidence Lower Bound objective, the reconstruction loss minimizes the distance between the input data x and the model output $\hat{x}$. In our experiments we employed three different approaches for the reconstruction loss:

Imitation Loss. The imitation loss is a criterion to minimize the mean squared error between a given label y, in this case an action from an expert which results in $y \in \mathbb{R}^N$ and the predicted action $\hat{x} \in \mathbb{R}^N$. It is defined as in Equation (4.5).

$$\begin{aligned} \mathcal{L}_{\text{Imi}} : \mathbb{R}^N \times \mathbb{R}^N &\to \mathbb{R} \\ y, \hat{x} &\mapsto \frac{1}{N} \sum_{i=0}^{N-1} (y_i - \hat{x}_i)^2 \end{aligned} \tag{4.5}$$

Distance Loss. This loss function minimizes the distance between a desired position in 2D space as label p_{target} and the action outcome of a state action combination. The state information needed for this criterion are only the current robot arm angles q. Like before the action is referred to as $\hat{x}$.

$$\begin{aligned} \mathcal{L}_{\text{Dist}} : \mathbb{R}^2 \times \mathbb{R}^2 \times \mathbb{R}^N &\to \mathbb{R} \\ p_{\text{target}}, \hat{x}, q &\mapsto ||p_{\text{target}} - \text{FK}(q + \hat{x})||_2 \end{aligned} \tag{4.6}$$

This criterion is also used in a standard supervised learning approach where the model predicts actions based on state information.

Inverse kinematics Loss. This criterion as in Equation (4.7) is the result of merging distance loss and imitation loss in one loss function as a weighted sum of those two components with weights $w_{\text{Imi}}, w_{\text{Dist}} \in \mathbb{R}$. Inside the code, the loss function can be configured to work also a pure distance loss function without providing any expert action and make it therefor also applicable to the afore mentioned supervised learning approach.

$$\begin{aligned} \mathcal{L}_{\text{IK}} : \mathbb{R}^2 \times \mathbb{R}^2 \times \mathbb{R}^N, \mathbb{R}^N &\to \mathbb{R} \\ p_{\text{target}}, \hat{x}, q, y &\mapsto w_{\text{Imi}} \cdot \mathcal{L}_{\text{Imi}}(y, \hat{x}) + w_{\text{Dist}} \cdot \mathcal{L}_{\text{Dist}}(p_{\text{target}}, \hat{x}, q) \end{aligned} \tag{4.7}$$

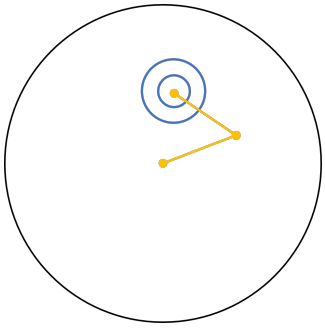


Figure 4.7: Schematic drawing how the target gaussian dataset works. First we are going to take the state as is comes form Algorithm 5 (in yellow). Then we are going to sample a target position from inside a the two dimensional truncated or non truncated gaussian distribution (in blue) and define it as new target position. Optionally we can also compute the corresponding CCD action.

Name	Value
VAE x	$p_{\text{target},t}$
Supervised x	s_t
c	$(p_{\text{current},t}, q_t)$
$\hat{x}$	q_{t+1}

Table 4.1: Latent workflow parameter with their correlation

4.5 Learning the Latent Model

For the upcoming experiments we settled one setting with one kind of dataset, the TargetGaussian.

The idea behind the gaussian target dataset is to learn an action which moves the end-effector towards a target sampled from a two dimensional gaussian distribution around the end-effector as demonstrated in Figure 4.7.

While training the Variational Autoencoder we settled on hyperparameter as described in Chapter 9 and input vectors as in Table 4.1. It is important to note that x for VAE, concatenated with c equal to s_t is which is the same input as for the Supervised model.

While training VAE and Supervised model you have to account the additional hyperbolic tangent function in the back of Figure 4.1. This additional stage in introduced with an instance of `PosProcessor`. The `PosProcessor` is enabled during training the latent model but deactivated during training SAC, because SAC applies a hyperbolic tangent function in the `PolicyNet`.

4.6 Software

In this section, we present the software stack developed to train the latent models and the reinforcement learning agent for our research. The software tools are available on the following GitHub repositories:

- Latent Models and RL Agent: `https://github.com/RobinU434/Bachelorthesis`
- Inverse Kinematics Environment: `https://github.com/RobinU434/IK-RL`

4.6.1 Inverse Kinematics Environment

As described earlier in Section 4.2, our research focuses on solving the inverse kinematics problem for a robot arm with N joints in a two-dimensional setting. To facilitate this, we have implemented an environment, which is available at `https://github.com/RobinU434/IK-RL`. This environment is developed in Python and is integrated into the Farama Gymnasium framework [57].

4.6.2 Machine Learning Software

The machine learning software used in our experiments, including Variational Autoencoders (VAEs), supervised models, and the Soft Actor-Critic (SAC) algorithm, is hosted at `https://github.com/RobinU434/Bachelorthesis`. All the implemented

artificial neural networks are based on the PyTorch framework. The RL algorithms are built upon the implementations provided by `https://github.com/seungeunrho/minimalRL` and have been further customized to suit our experimental requirements. For more detailed information, please refer to the respective GitHub repository.

5 Experiments

In this chapter we are going to present the experimental results. We focus hereby on two main sections: “Training Latent Models” and “Reinforcement Learning”. In the first section we will cover all results on training Latent Models including experiments with the Variational Autoencoder and Supervised model trained with different loss functions as introduced in Chapter 4.
The second section “Reinforcement Learning” is going to present the results on how the Soft Actor-Critic handles the introduced 2D inverse kinematics environment and how the fusion with the pre-trained latent models affects the training

5.1 Training of Latent Models

In this section we will present the results on the conducted experiments how to train a Variational Autoencoder and supervised model. We train both models with two distinct loss functions: The distance loss and a weighted sum of the distance loss and imitation loss.

5.1.1 Variational Autoencoder

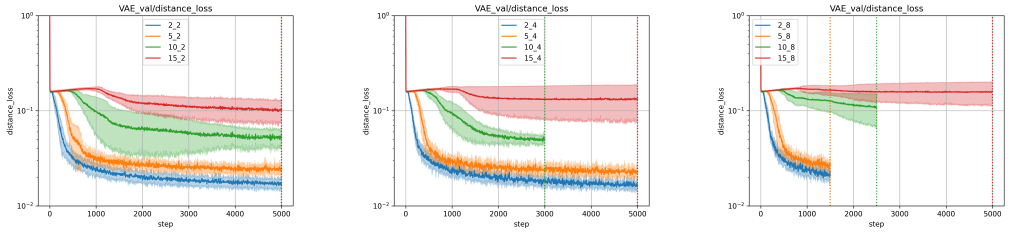
The outcomes of the experiments with the Variational-Autoencoder to encode and decode actions from the environment but also learn to solve inverse kinematics by minimizing the presented distance loss form Equation (4.6) that were conducted will

be presented in this section. Within the section we are covering two loss function setups. The first setup we are going to concentrate on only uses the distance loss while the second setup incorporates also an imitation loss.

Distance Loss

Coming up are the training results for different latent dimensions but only with the distance loss enabled which means `dataset target mode = POSITION`.
Because we only leverage the usage of the distance loss we do not need to encode and decode an action from an inverse kinematics solver, we just need to encode target information where we want to go plus state information as conditional information.

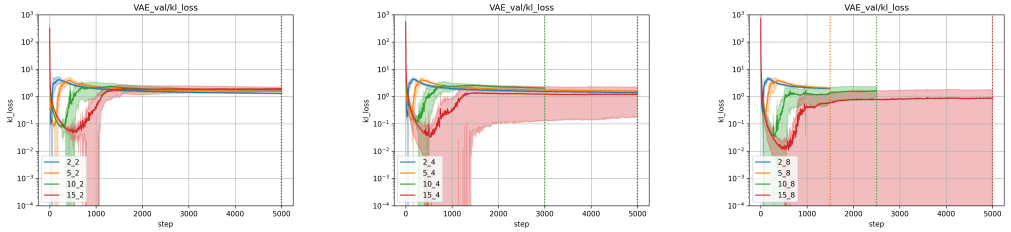
In Figure 5.1 we have a closer look into the reconstruction loss and kl loss. While we increase the number of joints $N \in [2, 5, 10, 15]$ we also increase the level of complexity the network has to master to come up with suitable action which leads to a low distance loss. The increasing level of complexity is quite noticeable in both the kl loss and reconstruction loss. Turning to Figure 5.1 we can make following observations:
All KL-Loss curves in Figure 5.1 (d) to (e) are shaped in the same way. First a drop before rising with a little overshoot and finally approaching the terminal value asymptotically. While increasing N we can see that the first local minimum shifts to the right, the upwards slope becomes slacker and the overshoot diminishes. This behavior confirms the the theory of increasing complexity while increasing the number of joints.
We can find another indicator by analyzing the reconstruction loss in Figure 5.1 (a) to (c). For all joints it is quite clear that the learning curves are shaped similar but approaching different final performances. For those experiments of $N \in [2, 5]$ we get a reconstruction loss < 0.02 independent of the latent dimension. But if we turn towards $N = 10$ we can clearly see that the latent dimension does make a significant difference between finding a solution, with a latent dimension of four, or



(a) mean reconstruction loss over the las 20 epochs for **latent dimension = 2**

(b) mean reconstruction loss over the las 20 epochs for **latent dimension = 4**

(c) mean reconstruction loss over the las 20 epochs for **latent dimension = 8**



(d) mean KL-loss over the las 20 epochs for **latent dimension = 2**

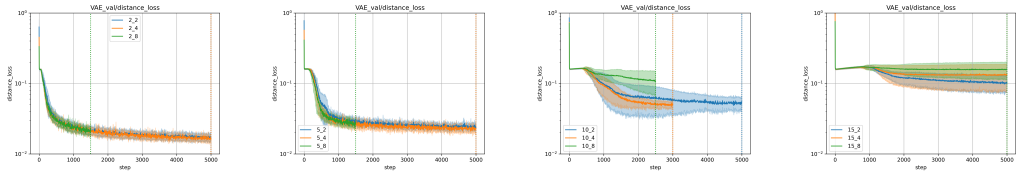
(e) mean KL-loss over the las 20 epochs for **latent dimension = 4**

(f) mean KL-loss over the las 20 epochs for **latent dimension = 8**

Figure 5.1: VAE validation results over different amounts of joints and with a latent dimension of 4. Each experiment was conducted 10 times with different random seeds. The solid curve is the average over those 10 experiments and the color shaded area resembles the standard deviation. Notice that the y axis in bot plots is in log scale.

failing with a latent dimension of eight. A further increase of N does not lead to better reconstruction losses across all latent space dimensions.

By observing the individual number of joints in each plot of Figure 5.2 form (a) to (d) we can observe that the latent dimension does make a difference in the final reconstruction loss performances. Starting with two joints there is no significant difference between the latent dimensions of two to eight. Continuing with 5 joints we can observe no significant difference in the finial performance but a slight difference with respect of how fast the models are converging. The experiments on a latent dimension of four and eight are very similar while the standard deviation for a latent dimension equal to two increases around 600 epochs indicating that some experiments



(a) $N = 2$ **(b)** $N = 5$ **(c)** $N = 10$ **(d)** $N = 15$

Figure 5.2: Comparison on the VAE latent dimension based on validation results over different amounts of joints. Each experiment was conducted 10 times with different random seeds. The solid curve is the average over those 10 experiments and the color shaded area resembles the standard deviation. Notice that the y axis in bot plots is in log scale. In the legend each label is structured as `<`N`>_<latent dimension>`

have slight troubles to converge.

A significant difference in final performance can be observed with $N = 10$. Here the best performance is returned by experiments with a latent dimension of four closely followed by two. Comparing two and four in Figure 5.2c shows that four is much more stable and faster in learning the problem as two. Last ranked are the experiments for a latent dimension of eight. Those experiments are by far not on the same level as with a latent dimension of two and four. A deeper look into the inference shows that the model fails to come up with an action to reach the given target position.

In Figure 5.2d a performance difference is noticeable with a latent dimension of two as best and eight as worst. But since the best performance of the experiments with a latent dimension of two is on the same level as eight in Figure 5.2c we can conclude that the the VAE fails at $N = 15$ with the present hyperparameter as in Chapter 9.

Distance and Imitation Loss

In this section, we present the outcomes of training the Variational Autoencoder (VAE) with the imitation loss using pre-calculated solutions from Algorithm 4 (CCD). An important configuration change involved setting `dataset target mode = ACTION`

in the configuration file, instructing the dataset to calculate actions from CCD as well and setting the imitation loss weight to 0.01.

Due to time constraints, we applied this loss function only in a setting with a latent dimension of four. This choice was based on the fact that a latent dimension of four had previously proven to be the best fit when using only the distance loss.

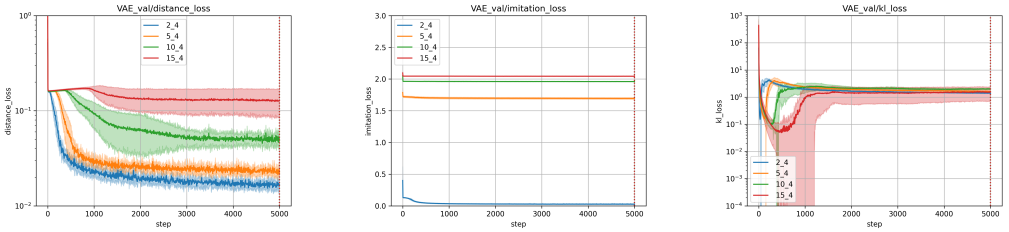
Overall, the distance loss functions depicted in Figure 5.3a yield almost identical results to those obtained using the distance loss functions shown in Figure 5.1b. This similarity is evident in the rapid improvement of experiments with two and five joints but also in the same level increase in experiments with $N = 10$ and $N = 15$.

In addition to the distance loss, we also introduced an imitation loss between the computed action from CCD and the action generated by the neural network Figure 5.3b. Notably, the imitation loss increases with the parameter N, although not at the same scale. As N grows, the differences between the imitation losses become smaller, but the convergence behavior where the curves experience a significant drop at the beginning of training and then stabilize, seemingly reaching their optimal values stays constant over all different N.

Finally, when examining the KL-divergence in Figure 5.3c, we observe an almost indistinguishable pattern compared to Figure 5.1e.

5.1.2 Supervised

For actions computed by the supervised model we only relied on state information as the input to predict an action to reach from the current state the given target position. Since this approach is very similar to the Variational Autoencoder setting we are also able to apply the afore mentioned Inverse Kinematics Loss with distance and imitation loss as in Equation (4.7).



(a) Distance loss

(b) Imitation loss

(c) KL diveregence from standard normal distribution

Figure 5.3: Results on the validation dataset while training a VAE with a latent dimension of 4 but incorporating an imitation loss weighted with 0.01. Results are collected from ten different runs and with a solid mean curve and a shaded standard deviation area. $N \in [2, 5, 10, 15]$

The results of the supervised experiments are divided into two parts. The first is only applying the distance loss while the second one is incorporating also an imitation loss with respect ot pre computed actions from the CCD solver.

Distance Loss

The distance loss results over different amount of joints in Figure 5.4 are very similar compared to Figure 5.1 (a) to (c). Parallel to Figure 5.1 we can also observe an overall increase in distance loss and finishing at different convergence levels while increasing N. Different to the experiments on the VAE is the standard deviation between the experiments. The supervised experiments show a significant smaller standard deviation across the whole training period compared to the VAE experiments.

Distance and Imitation Loss

For this series of experiments we also want to emphasis solutions close to solutions computed by the CCD by setting the dataset mode to `ACTION` and setting the imitation loss weight to 0.01.

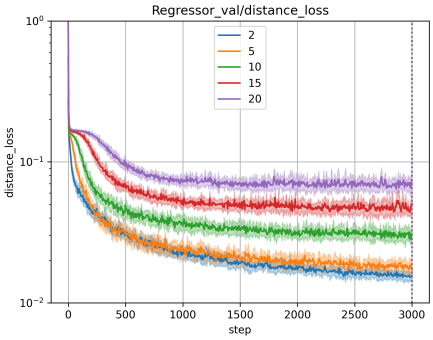


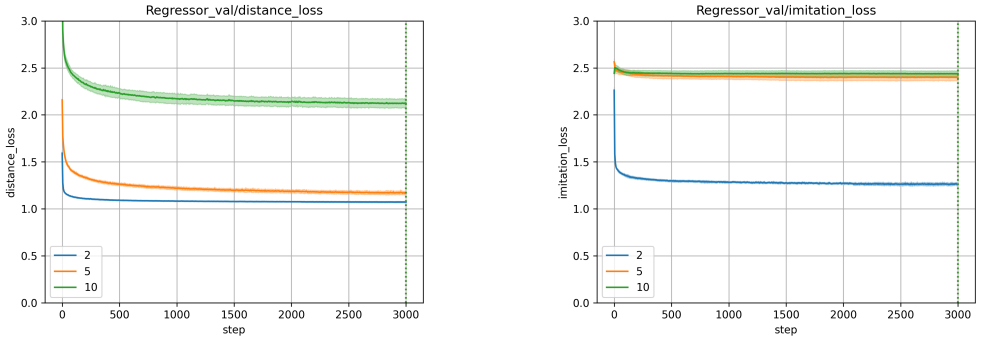
Figure 5.4: Distance loss for supervised experiments over different $N \in [2, 5, 10, 15, 20]$. For each N we conducted 10 experiments.

Unlike as in the previous section the distance loss differs a lot from Figure 5.4. In Figure 5.4 we can all curves start at almost the same loss value but the metric curves in Figure 5.3a are verticaly shifted and approaching their final value very fast and with a lot less variance between the experiements with the same N.
A bit more interesting are the results on the imitation loss. Similar to Figure 5.3b all curves have a drop in the first couple of epochs but are constant for the rest of the training. Comparing those curves with respect to the number of joints we observed that all mean curves have vastly different levels of convergence which seems not to be linear correlated with N.

In this series of experiments, our primary focus is on generating solutions that closely align with those computed by the Cyclical Coordinate Descent (CCD) method. To achieve this, we configured the dataset mode as `ACTION` and set a low imitation loss weight of 0.01.

In contrast to the previous section, the distance loss exhibits substantial differences from what we observed in Figure 5.4. Unlike the curves in Figure 5.4, where they all start with nearly identical loss values and need many epochs to converge, the metric curves in Figure 5.3a exhibit vertical shifts and converge to their final values rapidly, with considerably less variability among experiments sharing the same N.

More intriguing are the outcomes related to the imitation loss. Much like what is depicted in Figure 5.3b, all curves experience a drop in the initial epochs and subsequently maintain a relatively stable value throughout the rest of the training. Upon comparing these curves in relation to the number of joints, we have observed that the mean curves exhibit notably distinct convergence levels, and these do not appear to follow a linear correlation with N.



(a) Distance Loss **(b)** Imitation Loss

Figure 5.5: Validation results for supervised experiments on distance and imitation loss.

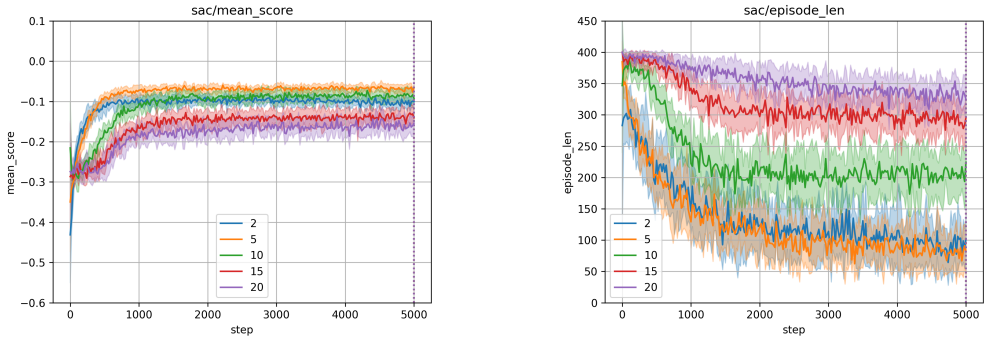
5.2 Reinforcement Learning

In this section we will present results on the reinforcement learning experiments on the 2D inverse kinematics environment.

First we will set a benchmark with the plane Soft Actor-Critic. Further we will combine Soft Actor-Critic with pre-trained latent models and have a look into their performance statistics but also comparing it to the plane Soft Actor-Critic results.

5.2.1 Baseline Soft Actor-Critic

First we would like to have a look how a the plain SAC algorithm, where the actor network directly calculates actions for the environment, performs on the inverse kinematics environment as described in Section 4.2. Further we will describe this kind of experiments as SAC baseline experiments. All Hyperparameters used for the SAC baseline experiments can be found in Chapter 9.



(a) Mean reward per step over the last 20 episodes.

(b) Mean episode length over the last 20 episodes

Figure 5.6: SAC baseline experiment results with an increasing number of joints. We can clearly see that the algorithm does not scale well with an increasing number of joints and therefor drops in performance. Each experiment was conducted 10 times and the shaded areas resemble the standard deviation around the mean.

In Figure 5.6 we plotted the mean reward per step and the average episode length over the last 20 epochs for different $N \in [2, 5, 10, 15, 20]$. Overall we can observe a decrease of collected reward per step in Figure 5.6a and an increasing amount of steps in one episode in Figure 5.6b while increasing the number of joints N. In Figure 5.6b the training curves are approaching the maximum step limit of one episode while increasing the number of joints. Additionally we can observe that the standard deviation over multiple experiments is decreasing while increasing the number of joints.

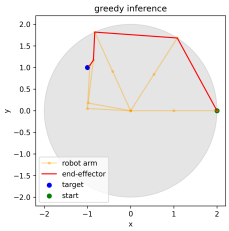
To show the actual behavior of an agent in a standardized setting we also plot an inference episode in Figure 5.7 and the distance remaining to target in Figure 5.8. The standardization in Figure 5.7 for one selected experiment for a relative consistent target position at $[-0.5, 0.5] \cdot N$.
As expected from the performance statistics in Figure 5.6 agents with two and five joints perform much better than agents with ten or more joints.
Interesting to observe is that all agents are minimizing the distance to the target quite fast in their first couple of steps. But after missing the target closely the agents behavior starts to oscillate. For closer inspection please refer to Figure 5.7d and Figure 5.7e.

Another noteworthy observation is the agent's approach in reaching the target position. Rather than taking a direct path towards the target, the agent tends to move its end-effector to the outer boundary of its reach before descending towards the target. This behavior warrants further discussion.

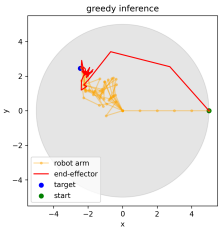
5.2.2 Soft Actor-Critic and Variational Autoencoder

In this section we are going to show the experimental results of combining a Variational Autoencoder with Soft Actor-Critic. As we know from Chapter 4 we only use the decoder of the trained Variational Autoencoder with input from the corresponding state and a latent action directly sampled from the parameterized distribution from the actor network.

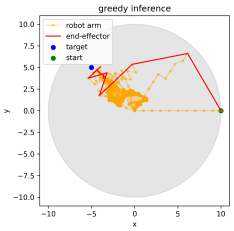
As we have seen in the previous section we carried out VAE experiments with three different latent space dimensions. Accordingly we trained ten SAC runs for each latent space size and number of joints $N \in [2, 5, 10, 15]$. We used the best VAE checkpoint available according to the overall validation loss. A detailed listing which checkpoints where actually used can be found in Table 5.1.



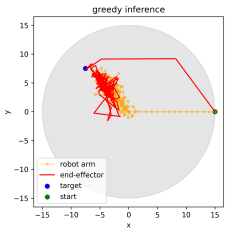
(a) From experiment 2_1691621262.



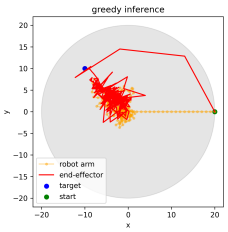
(b) From experiment 5_1691624939. To make things more clear only every 5th robot arm is drawn



(c) From experiment 10_1691622498. To make things more clear only every 5th robot arm is drawn



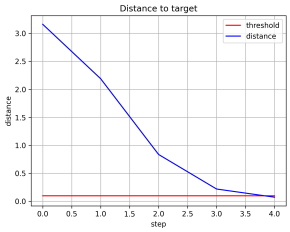
(d) From experiment 15_1691619106. To make things more clear only every 40th robot arm is drawn



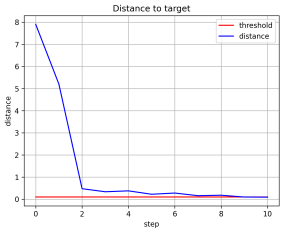
(e) From experiment 20_1691619159. To make things more clear only every 40th robot arm is drawn

Figure 5.7: SAC inference. The trajectory is plotted in red. robot arms are drawn in yellow with the little dots as positions of each joint. Target position and start position are scattered in blue and green. The space an end-effector is able to reach is plotted in grey.

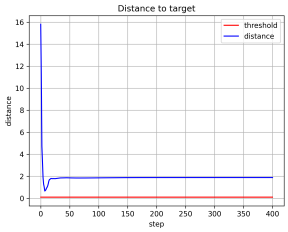
In Figure 5.10 we can observe what difference the choice of latent dimension has on the performance of the Soft Actor Critic algorithm. Quite obviously to see is that the choice of latent dimension really starts to matter after five joints. Where at $N = 10$ a latent dimension of two and four succeed but eight doesn't. This is clearly related to the performance of the VAE during its training. There we were also able to see that the latent dimension has an impact of the reconstruction loss performance of the VAE. Therefor it is no surprise to see that the best VAE checkpoints on 15 joints and the best VAE checkpoint on ten joints with a latent dimension of eight fails for the SAC + VAE setting.



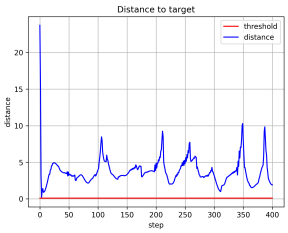
(a) Experiment 2_1691621262.



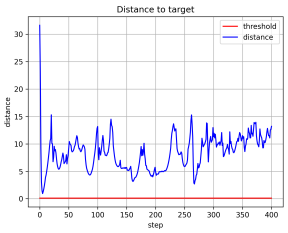
(b) Experiment 5_1691624939



(c) Experiment 10_1691622498



(d) Experiment 15_1691619106



(e) Experiment 20_1691619159

Figure 5.8: Distance to target for each action outcome. The target is consistent at $[-0.5, 0.5] \cdot N$. The threshold of 0.1 at which an episode is concluded successfully is drawn in red.

By having a closer look into the individual plots of Figure 5.10 we can difference between the collected mean score. Starting at two joints all mean curves are packet closely together where at five joints we can start to see a slight decrease in performance at 1000 epochs continuing up to 5000 epochs, between the experiments done with a latent dimension of two and the others.

This decrease is even more noticeable at ten joints where is now a gap between the experiments done with a latent dimension of four and the baseline experiments too.

In Figure 5.11 we can observe the mean episode length for different $N \in [2, 5, 10, 15]$. Starting at two joints we can see not much of a difference but at five joints we can observe slight differences between different latent dimensions. First up is the latent dimension of two At this value the experiments perform as in Figure 5.10b worst. Moving on to a latent dimension of four, those experiments perform best in terms of

N	latent dimension					
	2		4		8	
	random seed	epoch	random seed	epoch	random seed	epoch
2	1693105015	4950	1692461245	4890	1691608881	1480
5	1693096269	4495	1692476369	4795	1691606505	1290
10	1693098415	3615	1691627455	2535	1691608513	2485
15	1693103204	4815	1691628295	3845	1691618374	4760

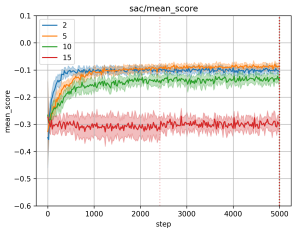
Table 5.1: Used checkpoints to train SAC with decoder from VAE. All experiments can be found in `results/vae/<`N`>_<latent dimension>_<random seed>/VAE_<epoch>*.pt`

mean episode length and even undercut the baseline experiments although they have not performed as good in terms of the mean score per step in Figure 5.10b. Finally those experiments with a latent dimension of eight are performing almost on the same level as the baseline experiments but not as good as the experiments with a latent dimension of four.

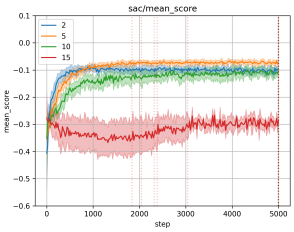
In Figure 5.10c we can see wide variety of best performances. The best performing setting are the baseline experiments followed by the experiments with a latent dimension of four. The next best performance is coming from a latent dimension equal to two and last ranked are the experiments with a latent dimension of eight. As we have mentioned earlier we suspect this drop in performance caused by the quality the decoder gets hand over to the Soft Actor Critic.

Note that although the maximum number of steps per episode was set to 400 the standard deviation of the experiment with a latent dimension of eight seam to surpass this threshold which is with a closer look into the actual curves clearly not the case.

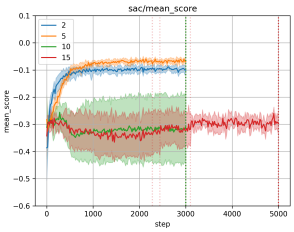
On the performed experiments with 15 joints we can see that none of the latent experiments is able to match the performance of the baseline experiments with even a couple of experiments with a latent dimension of four or eight aborting early. As it turns out this abort is caused by abnormally high values for alpha loss as shown in Figure 9.1 resulting in `nan` values in update steps. A positive alpha loss translates to a an increase of alpha translates to an increase in exploration for the policy, by



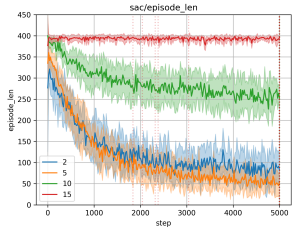
(a) mean episode reward per step averaged over the last 20 episodes. With a **latent dimension = 2**



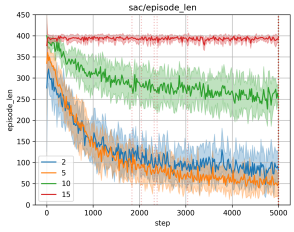
(b) mean episode reward per step averaged over the last 20 episodes. With a **latent dimension = 4**



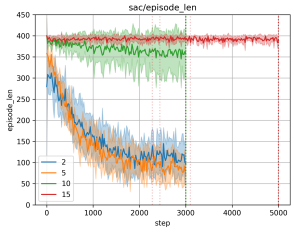
(c) mean episode reward per step averaged over the last 20 episodes. With a **latent dimension = 8**



(d) mean episode length averaged over the last 20 episodes. With a **latent dimension = 2**



(e) mean episode length averaged over the last 20 episodes. With a **latent dimension = 4**

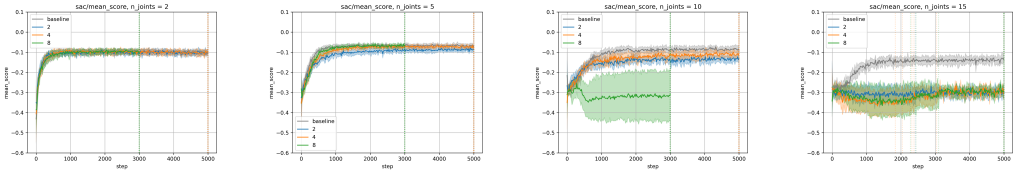


(f) mean episode length avergaed over the last 20 episodes. With a **latent dimension = 8**

Figure 5.9: SAC + decoder form VAE with different latent dimensions. Each experiment was conducted 10 times with different random seeds. The solid strong line is the mean over those 10 experiments. The color shaded area covering the mean is the standard deviation around the mean.

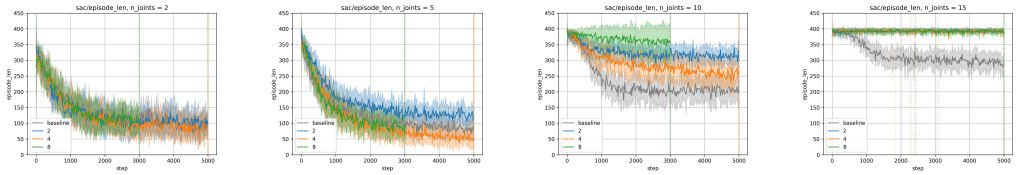
encouraging a higher standard deviation for the parameterized distribution returned by the actor network.

Because we have seen a lack in performance between the baseline SAC experiments we additional trained VAEs with the imitation loss enabled to minimize the distance between the state angle distribution the decoder is receiving as in Figure 5.13 versus the state angle distribution the decoder is trained as in Figure 4.5. But if we have a look into Figure 5.13c and compare to the plots on the left we do not see a significant change. This could be extrapolated to higher N and explain the missing improvement



(a) $N = 2$ **(b)** $N = 5$ **(c)** $N = 10$ **(d)** $N = 15$

Figure 5.10: Shown are the collected mean scores over the the last 20 episodes per step, compared different latent dimensions and the baseline. Solid line is the mean over 10 experiments and the color shaded area is the corresponding standard deviation. Higher means better with 0 as the maximum. The colored dotted lines mark an end of an experiment.



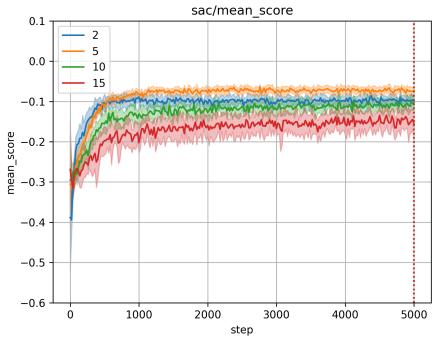
(a) $N = 2$ **(b)** $N = 5$ **(c)** $N = 10$ **(d)** $N = 15$

Figure 5.11: Shown are the mean episode length over the the last 20 episodes, compared different latent dimensions and the baseline. Solid line is the mean over 10 experiments and the color shaded area is the corresponding standard deviation. Lower means better with 1 as the minimum and 400 as the maximum.

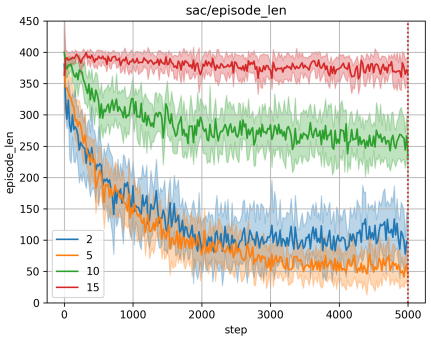
we would expect. But since this is not the main focus of this thesis we leaf these steps for future research.

5.2.3 Soft Actor-Critic and Supervised

Merging the supervised model with SAC yields performances as in Figure 5.14. We do can see very similar performances as with merging SAC with VAE. Starting with the similarities the achieved mean scores achieved from the SAC + VAE model in Figure 5.10 are very close to those achieved by SAC + supervised in Figure 5.14a.



(a) mean episode reward per step averaged over the last 20 episodes.



(b) mean episode length avergaed over the last 20 episodes

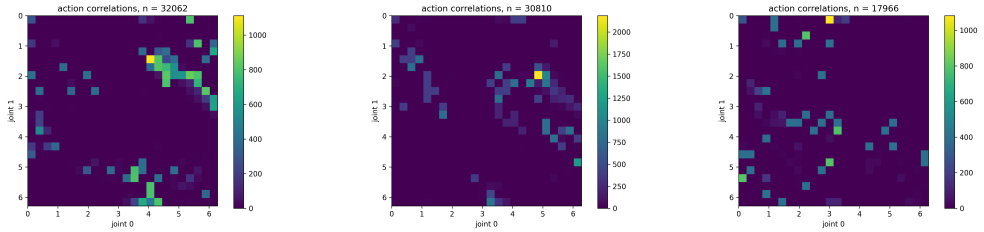
Figure 5.12: SAC + decoder form VAE with **latent dimension = 4** and trained with imitation loss enabled. Each experiment was conducted 10 times with different random seeds. The solid strong line is the mean over those 10 experiments. The color shaded area covering the mean is the standard deviation around the mean.

Also the curve shapes are similar but the achieved episode lenghts are significant higher for SAC + supervised (factor ≈ 2).

Combining the supervised model with SAC produces performance results similar to those depicted in Figure 5.14. Notably, the mean scores achieved by the SAC + VAE model in Figure 5.10 closely resemble those attained by SAC + supervised in Figure 5.14a. Furthermore, the overall curve shapes exhibit similarities, but it's worth highlighting that the episode lengths achieved by SAC + supervised are notably higher, approximately a factor of 2.

Another distinguishing factor when comparing the fusion of SAC with either the VAE or the supervised regressor approach can be observed in the distribution of log probabilities returned by the actor networks for sampling from the Gaussian distribution, see Table 5.2 .

For the *baseline* experiments, the mean log probability decreases as the number of joints increases, indicating an increase in policy stochasticity. In contrast, experiments



(a) SAC baseline experiments

(b) SAC + VAE with latent dimension = 4 and only distance loss enabled

(c) SAC + VAE with latent dimension = 4, distance and imitation loss enabled

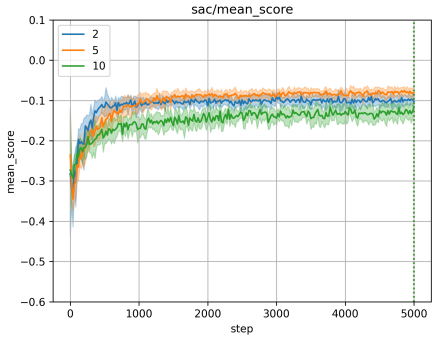
Figure 5.13: Action correlation for baseline experiments, latent dimension only trained on distance loss and SAC + VAE trained on distance and imitation loss.

experiment series	N							
	2		5		10		15	
	μ	σ	μ	σ	μ	σ	μ	σ
baseline	-5.2	1.0	-15.2	1.6	-32.5	2.6	-47.8	3.6
latent actor 4	-13.5	2.9	-15.6	2.5	-15.5	4.4	32.8	102.4
latent imitation 4	-12.6	4.0	-12.4	1.5	-13.2	0.8	-23.2	3.3
supervised distance loss	-9.0	1.0	-10.5	0.4	-10.9	1.0		
supervised imitation loss	11.0	2.1	41.8	2.8	39.7	2.9		

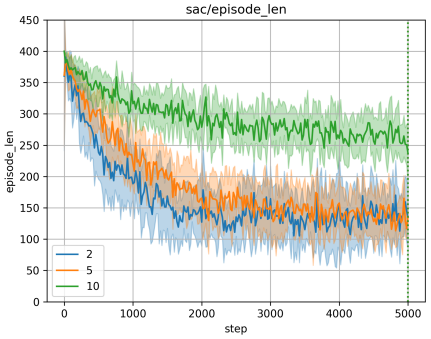
Table 5.2: Log probabilities of each policy series during training form the last 50 episodes each over 10 experiments.

with latent models *latent actor 4* and *latent imitation 4* maintain almost consistent log probabilities across different values of N. However, experiments involving SAC + supervised models exhibit substantially higher log probabilities. Further discussion on this observation will be provided in Chapter 6.

Lastly, experiments conducted with a supervised model trained using imitation loss are depicted in Figure 5.15. It's evident that all experiments in this category do not reach the same level of performance as the baseline experiments or other combinations of SAC with latent models.

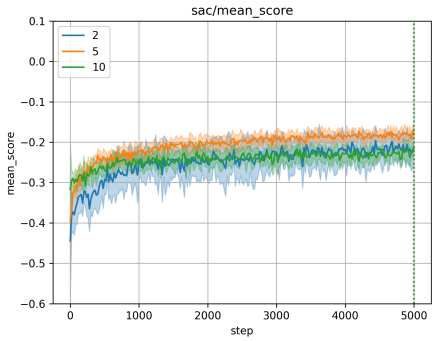


(a) Mean score per step

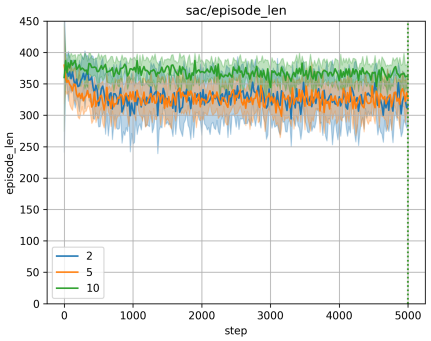


(b) episode length

Figure 5.14: Validation results for supervised experiments only on distance loss



(a) Mean score per step



(b) episode length

Figure 5.15: Validation results for supervised experiments on distance and imitation loss.

6 Discussion

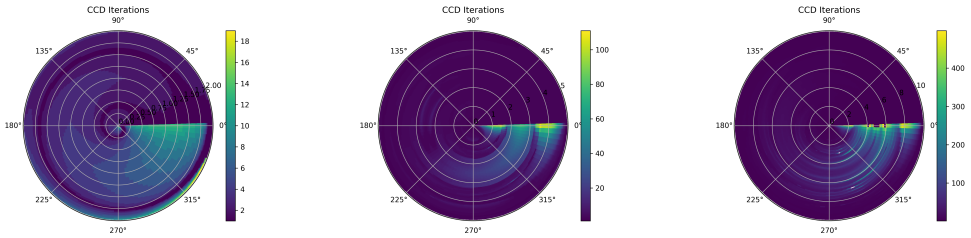
In this section we are going to discuss the before presented results in the experiments. We focus hereby only on the results from the Reinforcement Learning section in Chapter 5.

6.1 Baseline SAC

Let's delve into the foundational results of the baseline SAC experiments. We immediately noticed a pronounced correlation between the performance of the SAC algorithm and the number of joints it was trained on. This connection is quite logical, as the effectiveness and computational demands of other inverse kinematics solvers, such as CCD, are intricately linked to the number of joints in the robotic system. This connection beautifully aligns with the runtime complexity analysis portrayed in Figure 4.6b.

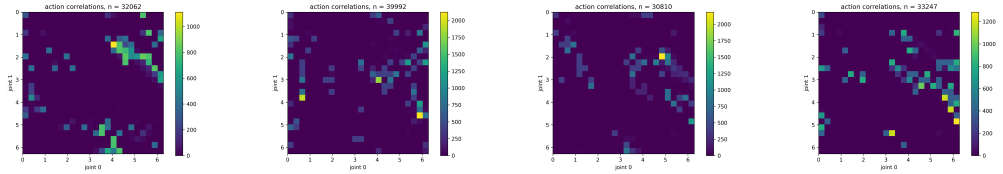
A comparison between CCD and baseline SAC revealed several intriguing distinctions:

- **Target Dependence**: Upon closer examination of CCD, it became evident that its performance in a single run was significantly influenced by the initial configuration and the specific target location. This was well-illustrated in Figure 6.1, which depicted CCD's performance in terms of the number of iterations required to reach the desired target position. Interestingly, when we



(a) $N = 2$ **(b)** $N = 5$ **(c)** $N = 10$

Figure 6.1: Heatmap of how many iterations CCD has needed to solve inverse kinematics starting from $[N, 0]$ targeting the middle of each polygon. The maximum budged are 20 times N with a target precision of 0.1.



(a) baseline. number of samples = 32062 **(b)** laten dimension is 2. number of samples = 39992 **(c)** laten dimension is 4. number of samples = 30810 **(d)** laten dimension is 8. number of samples = 33247

Figure 6.2: Plotted is the state angle correlation of two joints as a 2D histogram. The state angles are sampled from multiple episodes to uniformly sampled target positions. Each bucket can be translated to the amount of angles in one 3D bucket b with limits $b_{0,0}, b_{0,1}, b_{1,0}, b_{1,1}$. Mathematical speaking: $|b| = |\{q_t | q_t \in [0, 2\pi)^2 \wedge q_{0,t} \in [b_{0,0}, b_{0,1}] \wedge q_{1,t} \in [b_{1,0}, b_{1,1}]\}|$

experiment series	N	2	5	10	15	20
baseline	$\mathfrak{S}$	0.73	0.83	0.57	0.42	0.30
	max eval $\bar{R}_t$	0.09	0.10	0.32	0.74	0.82
latent 2	$\mathfrak{S}$	0.66	0.78	0.33	0.04	
	max eval $\bar{R}_t$	0.11	0.15	0.83	3.62	
latent 4	$\mathfrak{S}$	0.76	0.84	0.54	0.04	
	max eval $\bar{R}_t$	0.09	0.09	0.50	3.89	
latent 8	$\mathfrak{S}$	0.72	0.88	0.22	0.06	
	max eval $\bar{R}_t$	0.09	0.08	2.97	4.22	
latent imitation	$\mathfrak{S}$	0.85	0.85	0.45	0.11	
	max eval $\bar{R}_t$	0.08	0.10	0.44	2.01	
super distance	$\mathfrak{S}$	0.62	0.60	0.53		
	max eval $\bar{R}_t$	0.11	0.20	0.64		
super imitation	$\mathfrak{S}$	0.19	0.25	0.21		
	max eval $\bar{R}_t$	0.37	0.34	0.71		

Table 6.1: Solved ratio $\mathfrak{S}$ and closest distance to target during one episode as known as the maximum reward during one episode for one target position, for all SAC experiments starting at $[N, 0]$ and targeting the same targets as in Figure 6.3 with a maximal budged of 400 steps. Those are examples from individual experiments all from epoch 3000. For more details which experiments please have a look into the figures directory. The suffix of each png describes where to find the experiment. For a closer inspection of how the individual models perform over the evaluation please refer to Chapter 9

analyzed the steps needed to reach the target position using SAC, a noteworthy trend emerged. As the number of joints increased, we found that SAC more frequently reached the maximum iteration limit rather than successfully reaching the target as illustrated in Chapter 9 and Table 6.1. One potential explanation, alluded to in Figure 5.7, was that the agent's behavior sometimes declined as it neared the target. After coming tantalizingly close, the end-effector would rapidly retreat towards the arms origin, failing to get as close to the desired target as before. This observation was backed up by Figure 6.4, which showed that the minimum distance to the target was typically achieved in around 50 steps or fewer during an episode. It's worth noting that this difficulty in reaching the target could be attributed to SAC's fixed threshold for ending an episode, which required a 0.1 Euclidean distance between the end-effector and the target by definition. As the number of joints increased, the area within which the end-effector's position could end an episode diminished quadratically, making it progressively harder to reach the target with more joints. However, this argument was counterbalanced by the fact that the robot arm could execute more precise movements or potentially decompose the problem into smaller parts. For instance, joint 1 to $N-2$ could be utilized to roughly position the end-effector near the target, while joint $N-1$ to N could perform the precise adjustments. This raises a crucial question: why did the agent consistently fail, often by a substantial margin, after coming so close to the target? One plausible explanation is that the agent frequently encountered scenarios where it almost reached the target but had no opportunity to learn from these experiences. On the contrary, the extended episode length and increasing failure rate while increasing N, as seen in Table 6.1, should work against this trend.

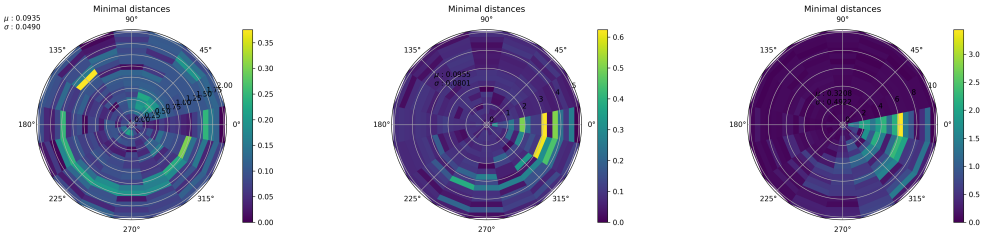
- **Solutions**: A deeper exploration of how SAC and CCD tackled inverse kinematics problems unveiled a strong contrast in the distribution of state angles. Comparing Figure 4.5, which depicted the state angle distribution required

to reach the target in the dataset, with Figure 6.2a, which illustrated the distribution resulting from baseline SAC, highlighted this discrepancy. While the former exhibited a rhombus-shaped distribution, the latter showcased a more scattered pattern.

It is important to recognize that this qualitative comparison between CCD and SAC is inherently limited. CCD enjoys unrestricted access to the entire action space, whereas SAC's action range is constrained due to action normalization. It's also noteworthy that scaling the hyperbolic tangent output to a range of 2π or higher, while a potential solution, is not without its challenges. Doing so could lead to SAC finding multiple actions that yield the same action outcome or potentially complicating the calculation of log probabilities for the policy.

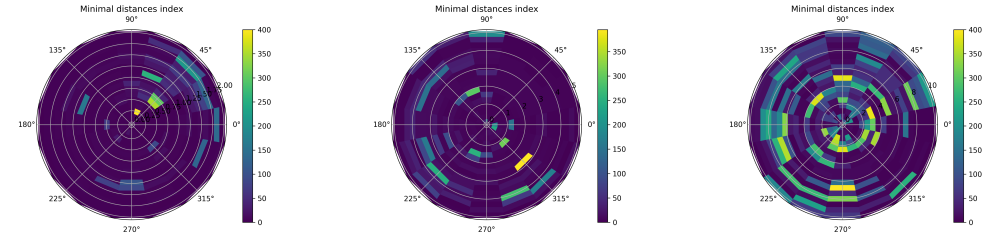
At this juncture, we can draw connections between our baseline experiments and the concepts presented in the Motor Synergy paper [7]. An interesting proposition arises when we closely examine the inverse kinematics problem as defined in the environment description. One conceivable strategy involves dividing the robot arm into two sections, each comprising $N/2$ segments, and constraining the relative angles within each segment. These two segments would effectively resemble a robot arm with just two extended joints. As previously demonstrated, environments with two joints are quite tractable. Yet, why doesn't the policy opt for such a strategy? Several arguments come to the fore. It's plausible that this strategy results in a longer trajectory to reach the target, which could affect the accumulated reward. Additionally, the choice could be influenced by the exploration strategy, as increasing entropy to encourage exploration does not seem effective due to the collapsing distribution, as seen in Figure 4.4.

The decrease in SAC's performance, as evidenced in Figure 5.6, can also be attributed to the linear growth in the complexity of the observation space, given that S is a subset of $\mathbb{R}^{4+N}$. To address this escalating complexity, researchers might consider increasing the number of hidden layers or neurons per layer while delving into neural



(a) $N = 2$ **(b)** $N = 5$ **(c)** $N = 10$

Figure 6.3: Minimal distance to a target during inference starting from $[N, 0]$ and targeting the middle of each polygon.



(a) $N = 2$ **(b)** $N = 5$ **(c)** $N = 10$

Figure 6.4: Heatmap of how many iterations baseline SAC has needed to solve inverse kinematics starting from $[N, 0]$ targeting the middle of each polygon. The maximum budged are 20 times N with a target precision of 0.1.

architecture search. An alternative approach for investigating this theory could involve the application of Neural-Evolution-of-Augmented-Topologies (NEAT). This algorithm seeks to identify the smallest neural architecture capable of solving a given task. Given the context provided, it's reasonable to assume that the discovered neural architecture becomes increasingly complex as N increases.

6.2 SAC with Latent Model

The integration of the Latent Model into the SAC framework yielded the results showcased in Chapter 5. While comprehending the striking disparities in problem-solving strategies between the trained latent models, as illustrated in Figure 6.2, and the Inverse Kinematics solver CCD, as depicted in Figure 4.5, it is rather astonishing that this approach was effective. Typically, such a substantial shift in the distribution of the observation space would result in unpredictable and unworkable behavior in the predicted actions within the Action space. We attempted to address this challenge by introducing the imitation loss, emphasizing predicted actions that led to state angles much closer to the original distribution.

As seen earlier, we obtained varying performances from the VAEs with different latent dimensions. But why? Let's first consider the experiments with a latent dimension of two. Given that we are endeavoring to encode two-dimensional target information, which is sampled in accordance with the concept of the `TargetGaussianDataset`, into a two-dimensional Gaussian manifold, the most efficient strategy for retaining information would involve channeling the mean values while returning a standard deviation close to zero. Since we employ the Evidence Lower Bound (ELBO) with a Gaussian distribution for a Variational Autoencoder (VAE), we must select a parameterized latent distribution that closely approximates the standard normal distribution. An additional challenge lies in encoding the target position as independent random variables. Given these properties and constraints, compressing this information can indeed be quite challenging. On the other hand, selecting a latent dimension that is too high can introduce excessive computational complexity, sampling difficulties, and a reduction in robustness.

Based on these observations, we propose setting the latent dimension of a VAE as an additional hyperparameter to optimize. Setting it too high can result in the extremely

high variance observed in the Soft Actor Critic task, while setting it too low may lead to a loss of encoded information.

Furthermore, let's revisit the action correlation plots. As previously mentioned, these plots across all SAC experiments deviate significantly from the initial distribution, as depicted in Figure 4.5. However, they exhibit remarkable similarity among themselves, as illustrated in Figure 6.2. This is an unexpected outcome, as each model could adopt a distinct problem-solving strategy and start anew in each iteration. Conversely, activating the imitation loss in the VAE experiments and integrating this approach with SAC should ideally result in a substantial shift towards the rhombus-shaped distribution observed in Figure 5.4. However, the familiar hotspots seen in Figure 4.5 vanish, and a new distribution emerges, characterized by the same spike at $(\pi, 0)$, as depicted in Figure 5.13c.

Finally we want to discuss which kind of latent model someone should prefer. The VAE or the supervised learning model?
As we have seen in the experimental part but also in Table 6.1, the SAC + VAE returns much more stable results and also better results compared to the supervised model. One reason might be the generalization within the training of a VAE. This results in a much smoother latent space than we could achieve with the supervised model. Another advantage from the VAE is the implied dimensionality invariance we do achieve with a constant latent dimension but different prediction dimension. An indication for this theory can seen by the almost constant mean log probability.

Lastly, let's address the choice between the VAE and the supervised learning model. As demonstrated in the experimental section and summarized in Table 6.1, SAC combined with the VAE returns significantly more stable results and outperforms the supervised model. One possible reason for this is the generalization achieved within the training of a VAE. This leads to a much smoother latent space compared to what can be achieved with the supervised model. Another advantage of the VAE lies in the implied dimensionality invariance achieved with a constant latent dimension but

varying prediction dimensions. This notion is supported by the nearly constant mean log probability, indicating the model's ability to capture the underlying patterns across various prediction dimensions.

Choosing Between Latent Models

The discussion culminates in a comparison between VAEs and supervised learning models, highlighting the advantages of each. Notably, SAC combined with VAEs returned more stable and superior results compared to the supervised model. This was attributed to the generalization capabilities inherent in VAE training, resulting in a smoother latent space compared to the supervised model. Additionally, VAEs offered dimensionality invariance with a constant latent dimension and different prediction dimensions.

Overall, the results and insights presented in this discussion section offer a valuable foundation for understanding the performance and potential applications of VAEs and supervised models in solving complex tasks like inverse kinematics in robotic systems. The choice between these models depends on specific use cases and optimization objectives.

7 Conclusion

In this thesis, we have explored the integration of latent models, specifically pre-trained Variational Autoencoder (VAE) decoders and supervised models, into Reinforcement Learning (RL) frameworks with the aim of expanding the action space capabilities of RL agents. Our research was evaluated using a benchmark environment: solving a 2D Inverse Kinematics problem on a robot arm with a variable number of joints, denoted as N. The study investigated how the incorporation of latent models enhances the RL agent's performance in handling complex action spaces.

Key Findings

The key findings of this research can be summarized as follows:

1. **Latent Model Integration**: The integration of a pre-trained decoder from a VAE with the Soft Actor-Critic (SAC) algorithm shows promise in addressing high-dimensional action spaces. However, it is not yet on par with standard approaches and requires further improvement.

2. **RL Benchmark Environment**: The 2D inverse kinematics benchmark environment has proven to be a suitable and flexible candidate for benchmarking different RL algorithms, offering a standardized platform for assessing agent performance.

Theoretical Contributions

Our research contributes to RL methodology and theory in several ways:

- **Advancements in RL**: This research advances RL methodology by introducing a novel approach for handling high-dimensional action spaces and expanding the scope of RL applications.

- **Latent Model Integration**: We contribute to the theoretical understanding of integrating latent models into RL, addressing aspects in design, training, and utilization.

- **Action Space Abstraction**: Our work advances the theory of action space abstraction, demonstrating the effectiveness of latent models in simplifying complex action spaces.

Limitations

While our research provides valuable insights, it is essential to acknowledge its limitations. We observed a performance gap between expected and actual results, particularly in scenarios involving a high number of joints. Additionally, our approach fell short when compared to baseline experiments, raising questions about its effectiveness. Challenges related to scalability, the choice of latent models, and the complexity of the RL environment may have influenced the outcomes. Further considerations include data efficiency, generalizability to other RL tasks, and the specific complexities of the 2D Inverse Kinematics problem.

Future Directions

Future research directions encompass refining latent models for improved action space expansion. This may involve exploring advanced generative models such as Generative Adversarial Networks (GANs) or incorporating additional latent model refinement

during the RL training process. Extensive hyperparameter tuning should be explored to optimize model and algorithm parameters, enhancing overall performance and efficiency. Investigating data augmentation techniques and designing benchmark environments that mirror real-world complexities will improve the practicality of the proposed approach. Additionally, future avenues include transfer learning, multi-agent RL applications, and interdisciplinary collaborations to expand the applicability of latent model-enhanced RL.

Key Takeaways

Readers of this thesis will gain insights into the importance of expanding action spaces in Reinforcement Learning (RL) through the integration of latent models, such as Variational Autoencoders (VAEs) and supervised models. They will also understand the challenges and limitations faced in achieving performance improvements, particularly in scenarios with a high number of joints. The thesis highlights the significance of benchmarking RL agents in various environments and provides directions for future research, including advanced latent models, hyperparameter optimization, and ethical considerations. Ultimately, readers will appreciate the real-world relevance of this research, particularly in fields like robotics and automation, and recognize the iterative nature of scientific inquiry in advancing RL capabilities.

8 Acknowledgments

I would like to take this opportunity to express my sincere gratitude to all those who have contributed to the completion of this thesis. This journey has been both challenging and rewarding, and I am deeply appreciative of the support and guidance I have received along the way.

First and foremost, I extend my heartfelt thanks to my supervisor, Jasper Hoffman, for his unwavering support, mentorship, and invaluable insights throughout the research process. His dedication to pushing the boundaries of knowledge and commitment to excellence have been truly inspiring.

I would also like to acknowledge the significant contributions of my co-workers, Jan Ole von Hartz and Jens Rahnfeld, who helped also a lot in the development and debugging of the Variational Autoencoder (VAE) models used in this research. Their expertise, tireless efforts, and collaborative spirit greatly enriched the quality of this work.

I am grateful to my colleagues, friends, and family for their encouragement, patience, and understanding during this demanding endeavor. Your unwavering support provided the motivation needed to overcome challenges and persist in pursuit of our research goals.

Additionally, I extend my appreciation to the academic and research communities whose work has paved the way for this research. The wealth of knowledge and

resources available in these communities has been instrumental in shaping the ideas presented in this thesis.

Lastly, I would like to acknowledge the resources provided by Neurobotics Lab lead by Professor Boedecker, which facilitated the successful execution of this research.

In closing, this research represents the collective efforts of many individuals who have contributed in various ways. Your support, expertise, and encouragement have been invaluable, and for that, I am truly grateful.

Bibliography

[1] O. Delalleau, M. Peter, E. Alonso, and A. Logut, "Discrete and continuous action representation for practical RL in video games," *CoRR*, vol. abs/1912.11077, 2019.

[2] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and S. Levine, "Soft actor-critic algorithms and applications," *CoRR*, vol. abs/1812.05905, 2018.

[3] N. Rudin, D. Hoeller, P. Reist, and M. Hutter, "Learning to walk in minutes using massively parallel deep reinforcement learning," 2022.

[4] T. Hubert, J. Schrittwieser, I. Antonoglou, M. Barekatain, S. Schmitt, and D. Silver, "Learning and planning in complex action spaces," 2021.

[5] K. P. Murphy, *Probabilistic Machine Learning: Advanced Topics*. MIT Press, 2023.

[6] J. Peters and S. Schaal, "Learning to control in operational space," pp. 197–212, 2008.

[7] J. Chai and M. Hayashibe, "Motor synergy development in high-performing deep reinforcement learning algorithms," *IEEE Robotics and Automation Letters*, vol. 5, no. 2, pp. 1271–1278, 2020.

[8] P. Wawrzynski, "Learning to control a 6-degree-of-freedom walking robot," pp. 698 – 705, 10 2007.

[9] A. Maćkiewicz and W. Ratajczak, "Principal components analysis (pca)," *Computers and Geosciences*, 1993.

[10] A. Allshire, R. Martín-Martín, C. Lin, S. Manuel, S. Savarese, and A. Garg, "LASER: learning a latent action space for efficient reinforcement learning," *CoRR*, vol. abs/2103.15793, 2021.

[11] C. Tessler, Y. Kasten, Y. Guo, S. Mannor, G. Chechik, and X. B. Peng, "CALM: Conditional adversarial latent models for directable virtual characters," in *Special Interest Group on Computer Graphics and Interactive Techniques Conference Conference Proceedings*, ACM, jul 2023.

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

[13] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[14] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," *CoRR*, vol. abs/1801.01290, 2018.

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

[16] 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.," in *ICLR* (Y. Bengio and Y. LeCun, eds.), 2016.

[17] V. Konda and J. Tsitsiklis, "Actor-critic algorithms," in *Advances in Neural Information Processing Systems* (S. Solla, T. Leen, and K. Müller, eds.), vol. 12, MIT Press, 1999.

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

[19] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," *CoRR*, vol. abs/1602.01783, 2016.

[20] M. C. Fu, "Chapter 19 gradient estimation," in *Simulation* (S. G. Henderson and B. L. Nelson, eds.), vol. 13 of *Handbooks in Operations Research and Management Science*, pp. 575–616, Elsevier, 2006.

[21] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations," *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021.

[22] P. Christodoulou, "Soft actor-critic for discrete action settings," *CoRR*, vol. abs/1910.07207, 2019.

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

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

[25] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.

[26] I. Papastratis, "Speech recognition: a review of the different deep learning approaches," *https://theaisummer.com/*, 2021.

[27] D. W. Otter, J. R. Medina, and J. K. Kalita, "A survey of the usages of deep learning in natural language processing," *CoRR*, vol. abs/1807.10854, 2018.

[28] H. A. Pierson and M. S. Gashler, "Deep learning in robotics: A review of recent research," *CoRR*, vol. abs/1707.07217, 2017.

[29] S. R. Dubey, S. K. Singh, and B. B. Chaudhuri, "A comprehensive survey and performance analysis of activation functions in deep learning," *CoRR*, vol. abs/2109.14545, 2021.

[30] A. F. Agarap, "Deep learning using rectified linear units (relu)," *CoRR*, vol. abs/1803.08375, 2018.

[31] G. Bebis and M. Georgiopoulos, "Feed-forward neural networks," *IEEE Potentials*, vol. 13, no. 4, pp. 27–31, 1994.

[32] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," *CoRR*, vol. abs/1511.08458, 2015.

[33] A. Sherstinsky, "Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network," *CoRR*, vol. abs/1808.03314, 2018.

[34] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, 1986.

[35] H. E. Robbins, "A stochastic approximation method," *Annals of Mathematical Statistics*, vol. 22, pp. 400–407, 1951.

[36] J. Kiefer and J. Wolfowitz, "Stochastic estimation of the maximum of a regression function," *The Annals of Mathematical Statistics*, vol. 23, no. 3, pp. 462–466, 1952.

[37] S. Ruder, "An overview of gradient descent optimization algorithms," *CoRR*, vol. abs/1609.04747, 2016.

[38] E. Häglund and J. Björklund, "Ai-driven contextual advertising: A technology report and implication analysis," 2022.

[39] S. S .S, S. v, and B. Krishnasamy, "A review on deep learning in medical image analysis," *International Journal of Multimedia Information Retrieval*, vol. 11, 03 2022.

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

[41] Z. Zhao, P. Zheng, S. Xu, and X. Wu, "Object detection with deep learning: A review," *CoRR*, vol. abs/1807.05511, 2018.

[42] R. Li, A. Jabri, T. Darrell, and P. Agrawal, "Towards practical multi-object manipulation using relational reinforcement learning," *CoRR*, vol. abs/1912.11032, 2019.

[43] J. M. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Zídek, A. Potapenko, A. Bridgland, C. Meyer, S. A. A. Kohl, A. Ballard, A. Cowie, B. Romera-Paredes, S. Nikolov, R. Jain, J. Adler, T. Back, S. Petersen, D. A. Reiman, E. Clancy, M. Zielinski, M. Steinegger, M. Pacholska, T. Berghammer, S. Bodenstein, D. Silver, O. Vinyals, A. W. Senior, K. Kavukcuoglu, P. Kohli, and D. Hassabis, "Highly accurate protein structure prediction with alphafold," *Nature*, vol. 596, pp. 583 – 589, 2021.

[44] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, T. Hubert, P. Choy, C. de Masson d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. S. Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals, "Competition-level code generation with AlphaCode," *Science*, vol. 378, pp. 1092–1097, dec 2022.

[45] K. P. Murphy, *Probabilistic Machine Learning: An introduction*. MIT Press, 2022.

[46] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," 2022.

[47] K. Sohn, H. Lee, and X. Yan, "Learning structured output representation using deep conditional generative models," in *Advances in Neural Information Processing Systems* (C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, eds.), vol. 28, Curran Associates, Inc., 2015.

[48] R. Yu, "A tutorial on vaes: From bayes' rule to lossless compression," 2020.

[49] Y. Bengio, L. Yao, G. Alain, and P. Vincent, "Generalized denoising auto-encoders as generative models," 2013.

[50] N. C. Cueto Ceilis and H. Peters, *An empirical comparison of generative capabilities of GAN vs VAE*. PhD thesis, 2022.

[51] A. Asperti and M. Trentin, "Balancing reconstruction error and kullback-leibler divergence in variational autoencoders," 2020.

[52] D. Constantin, M. Lupoae, C. Baciu, and B. Ilie, "Forward kinematic analysis of an industrial robot," 03 2015.

[53] R. Sun and Y. Ye, "Worst-case complexity of cyclic coordinate descent: $o(n^2)$ gap with randomized version," 2018.

[54] A. Aristidou and J. Lasenby, "Inverse kinematics: a review of existing techniques and introduction of a new fast iterative solver," 09 2009.

[55] P. Chang, "A closed-form solution for inverse kinematics of robot manipulators with redundancy," *IEEE Journal on Robotics and Automation*, vol. 3, no. 5, pp. 393–403, 1987.

[56] K. Hauser, "Robotic systems (draft)," 10 2018.

[57] M. Towers, J. K. Terry, A. Kwiatkowski, J. U. Balis, G. d. Cola, T. Deleu, M. Goulão, A. Kallinteris, A. KG, M. Krimmel, R. Perez-Vicente, A. Pierré, S. Schulhoff, J. J. Tai, A. T. J. Shen, and O. G. Younis, "Gymnasium," Mar. 2023.

9 Appendix

9.1 Hyperparameters

Name	Value
num joints	-
segment length	1
n time steps	400
action mode	"strategic"
discrete mode	false
rescale actions enabled	False
rescale actions min	-1
rescale actions max	1
task type	"ReachGoalTask"
task epsilon	0.1
task bonus	0
task normalize	true
task order	2

Table 9.1: Hyperparameter for the inverse kinematics environment. Not that num joints is different from experiment to experiment. You can adjust those values by altering *config/base_ vae.yaml*

9.2 Additional Experiment Plots

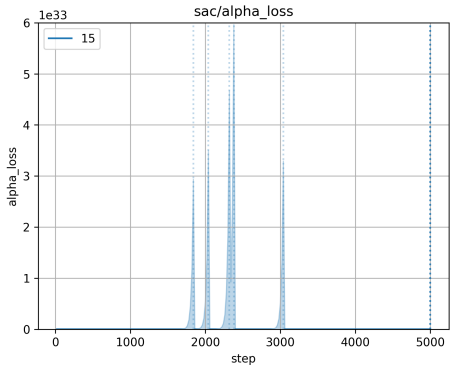


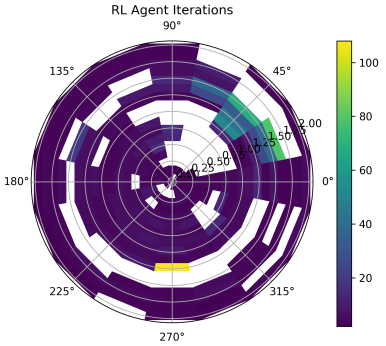
Figure 9.1: Plot to show the correlation between training abortion and spike in alpha loss.

Name	Value
num joints	-
num epochs	5001
learning rate	0.003
val interval	5
latent dim	-
encoder architecture	[128, 128]
encoder activation function	"ReLU"
decoder architecture	[256, 256]
decoder activation function	"ReLU"
kl loss weight	0.001
reconstruction loss weight	1
distance loss weight	1
imitation loss weight	0.001
dataset type	VAETargetGaussianDataset
dataset mode	"IK random start"
dataset batch size	2096
dataset shuffle	False
dataset action constrain radius	0.2
dataset std	0.2
dataset target mode	"POSITION"
dataset truncation	0
post processor enabled	True
post processor min action	-1
post processor max action	1

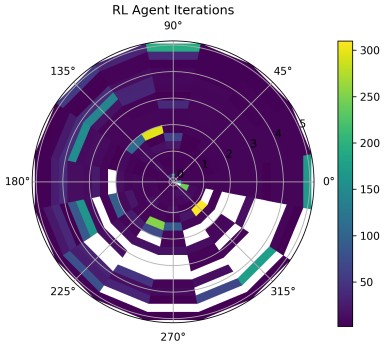
Table 9.2: Hyperparameter for Variational Autoencoder. Note that num joints and latent dim varies from between different types of experiments and is therefor not fixed. You can adjust those values by altering *config/base_ vae.yaml*. There are multiple dataset types available. But for the majority of our experiments we used the VAETargetGaussianDataset

Name	Value
lr q	0.001
init alpha	0.01
gamma	0.98
batch size	32
buffer limit	50000
start buffer size	1000
train iterations	20
tau	0.01
target entropy	-40.0
lr alpha	0.001
n epochs	5000
actor type	-
actor learning rate	0.0005
actor learning mode	0
actor latent dim	4
actor latent checkpoint dir	"results/vae/baseline"
actor architecture	[128, 128]
actor activation function	ReLU

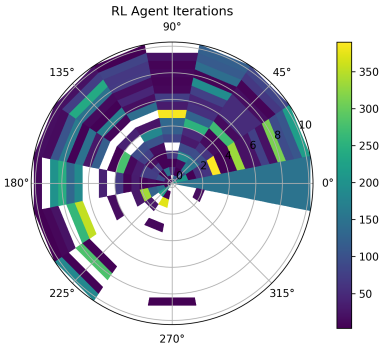
Table 9.3: Hyperparameter for SAC. Note that the actor type was left empty. For using a standard feed forward actor use "Actor", for using a VAE Decoder as latent model use: "LatentActor" and for using a separated pretrained supervised training model use "SuperActor".



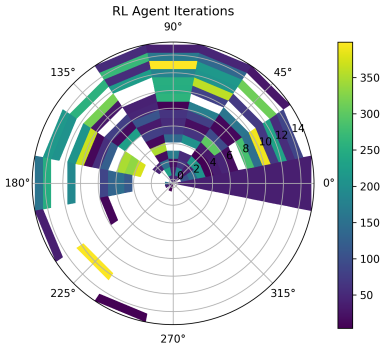
(a) $N = 2$



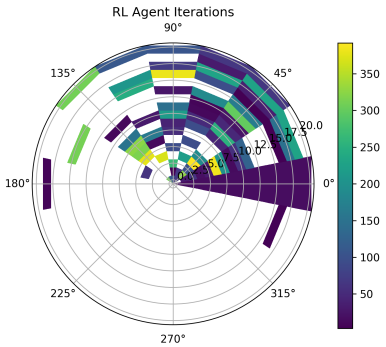
(b) $N = 5$



(c) $N = 10$

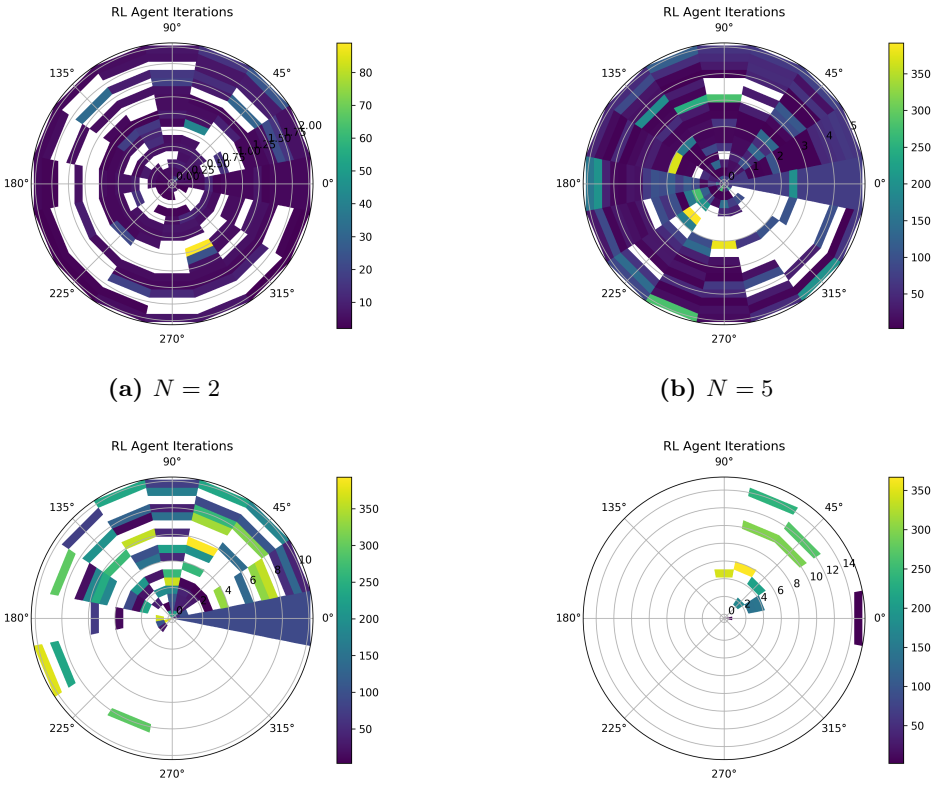


(d) $N = 15$



(e) $N = 20$

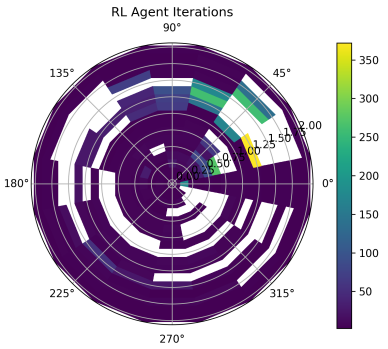
Figure 9.2



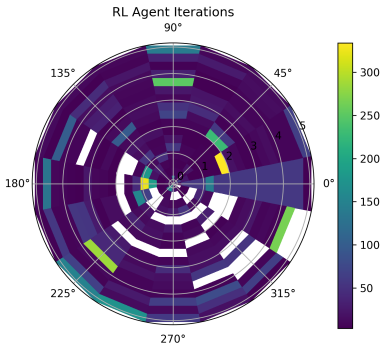
(a) $N = 2$ **(b)** $N = 5$

(c) $N = 10$ **(d)** $N = 15$

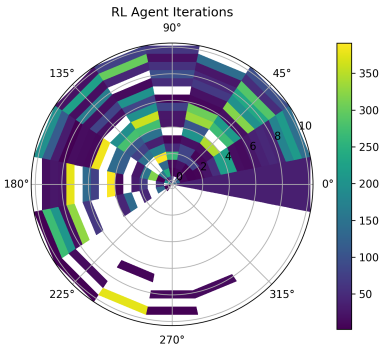
Figure 9.3: RL Agent iteration evaluation on VAE with latent $= 2$



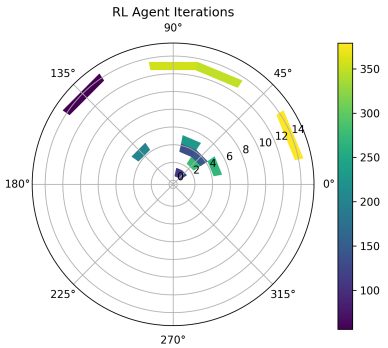
(a) $N = 2$



(b) $N = 5$

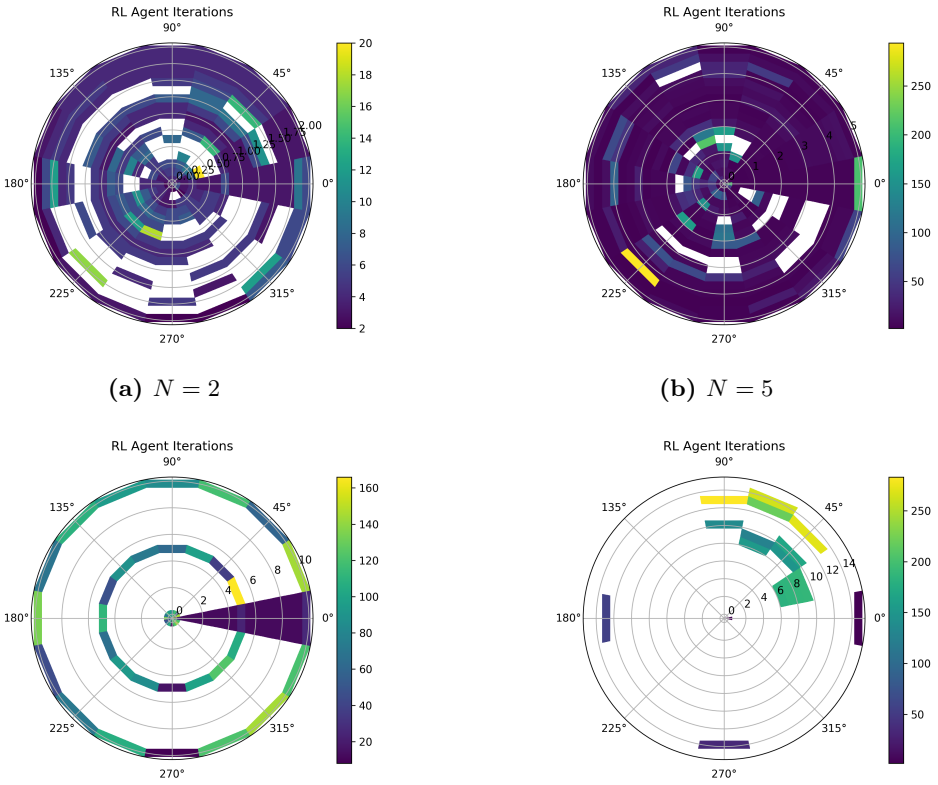


(c) $N = 10$



(d) $N = 15$

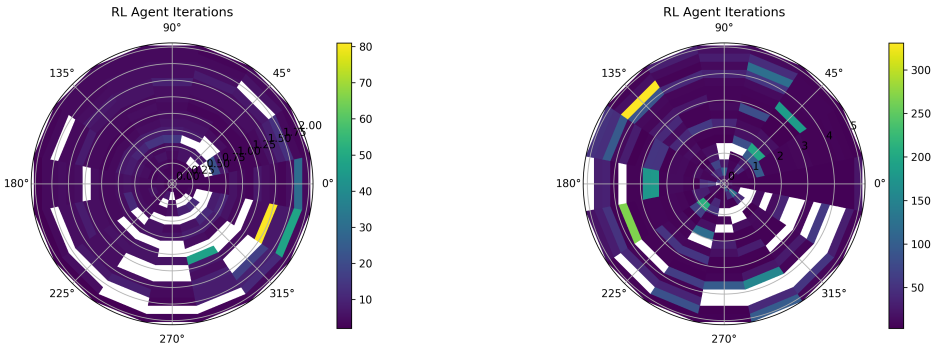
Figure 9.4: RL Agent iteration evaluation on VAE with latent = 4



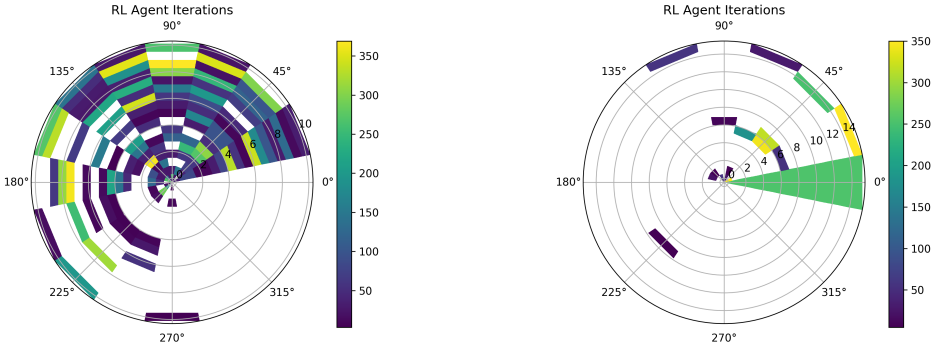
(a) $N = 2$ **(b)** $N = 5$

(c) $N = 10$ **(d)** $N = 15$

Figure 9.5: RL Agent iteration evaluation on VAE with latent = 8

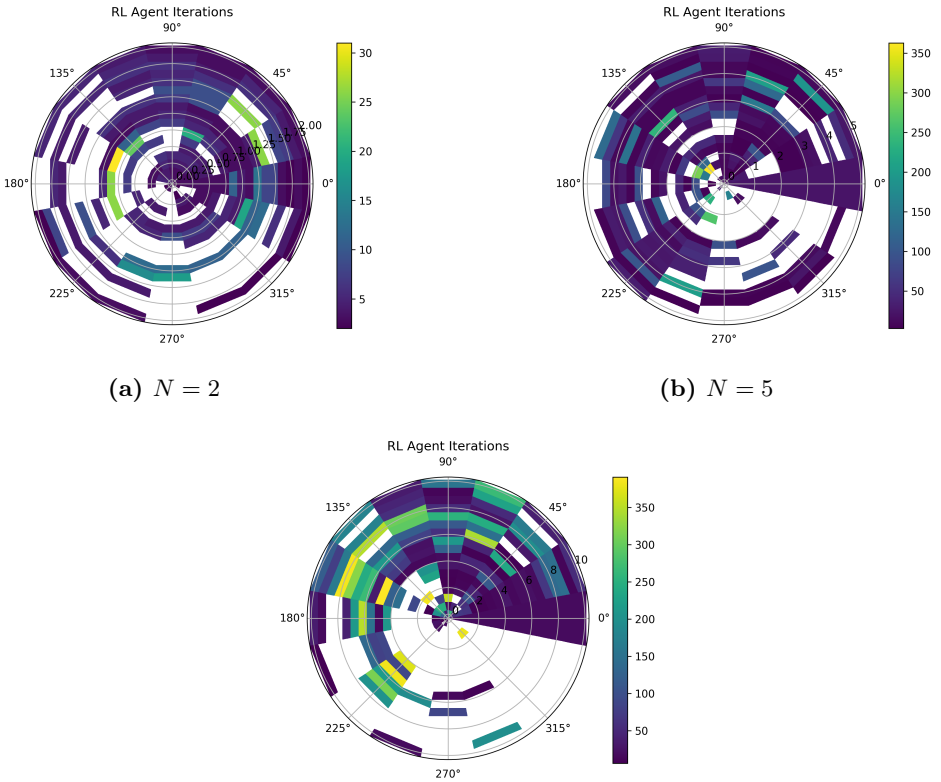


(a) $N = 2$ **(b)** $N = 5$



(c) $N = 10$ **(d)** $N = 15$

Figure 9.6: RL Agent iteration evaluation on imitation VAE with latent = 8

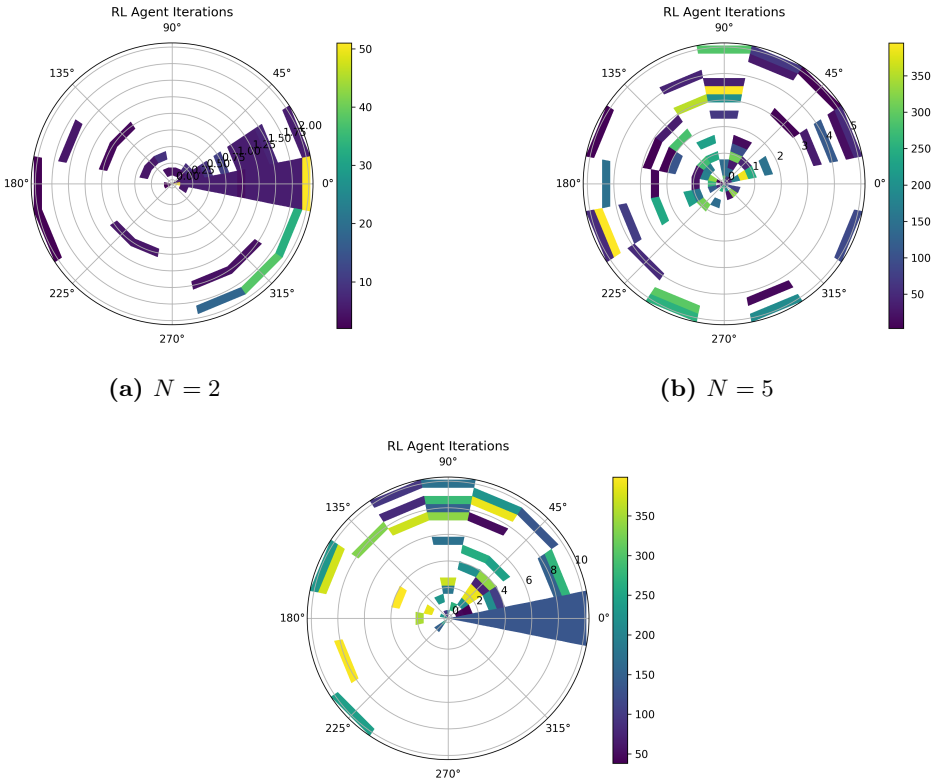


(a) $N = 2$

(b) $N = 5$

(c) $N = 10$

Figure 9.7: RL Agent iteration evaluation on supervised model



(a) $N = 2$

(b) $N = 5$

(c) $N = 10$

Figure 9.8: RL Agent iteration evaluation on supervised imitation model