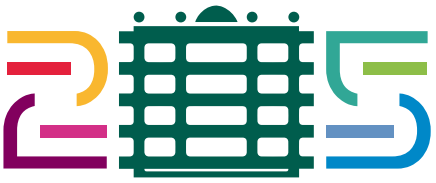


TECHNISCHE UNIVERSITÄT
IN DER KULTURHAUPTSTADT EUROPAS
CHEMNITZ

Faculty of Computer Science
Professorship of Neurorobotics

Master thesis for the award of the academic degree
Master of Science (M.Sc.)

Joint Space Control via Deep Reinforcement Learning with a Population-Coded Spiking Actor Network

by

Lukas Dammer, B.Sc.

First supervisor:	Prof. Dr. Florian Röhrbein
Second supervisor:	Saranraj Nambusubramaniyan, M.Sc.
Submission date:	August 8, 2023

Contents

List of Figures

List of Tables

List of Abbreviations

ANN	Artificial Neural Network
DNN	Deep Neural Network
DRL	Deep Reinforcement Learning
LIF	Leaky Integrate-and-Fire
PopSAN	Population-coded Spiking Actor Network
PPO	Proximal Policy Optimization
ROS	Robot Operating System
SGD	Stochastic Gradient Descent
SNN	Spiking Neural Network
STBP	Spatiotemporal Backpropagation
ac	accuracy
sr	success rate
TCP	Tool-Center-Point
DoF	Degrees of freedom

1 Introduction

In recent years, the field of robotics has witnessed significant advancements, ranging from autonomous vehicles to industrial automation systems. With the rapid development of robotics, we are also witnessing a gradual integration of robots into our everyday lives, including our homes. As robots become more common in our homes, they bring about a new set of challenges and opportunities. One of those challenges is to operate reliably and safely around people. Therefore, robots must be able to adapt to unforeseen situations and interact autonomously with their environment.

Figure 1.1: An example of a human-robot interaction

Central to these developments is the ability to design intelligent control systems that enable robots to interact seamlessly with their environment. One approach that has been very successful in addressing these challenges is the combination of Deep Neural Networks (DNNs) with reinforcement learning, called Deep Reinforcement Learning (DRL). This enables robots to learn complex behaviors from high dimensional sensory input and optimize their actions based on previous experiences, ultimately allowing them to handle dynamic and uncertain environments more effectively.

Even though DNNs, have demonstrated remarkable success recently, in various domains, including computer vision, natural language processing, and speech recognition, they are not without disadvantages. DNNs often struggle to effectively handle temporal information and deal with the real-time demands of robot control. This limitation arises from the inherent nature of DNNs, which rely on continuous-valued activations and lack explicit time representation. As a result, they may not fully capture the temporal dynamics and spike-based computations observed in biological neural networks. Considering that the human brain does not possess these limitations, it is particularly compelling to explore biologically plausible alternatives that can overcome the shortcomings of DNNs.

1.1 Motivation

One of those biologically plausible alternatives to DNNs are Spiking Neural Networks (SNNs). SNNs draw inspiration from the functioning of the brain, where neurons communicate via discrete electrical pulses or spikes. Thereby, SNNs offer several advantages over traditional DNNs in the context of robot control. Firstly, they inherently incorporate the concept of time, enabling them to represent and process temporal information accurately. This feature is crucial for tasks such as real-time perception, where the robot must continuously process sensor inputs to react promptly to changes in the environment. Secondly, SNNs exhibit event-driven computation, meaning they only need to compute and transmit spikes when there are significant changes in the input, leading to energy-efficient processing. And SNNs facilitate the incorporation of biological plausibility into artificial neural networks. This is essential, because it allows robots to perceive the world in a manner that's closer to how humans do, which can lead to a more intuitive form of interaction between humans and robots.

For these reasons, SNNs appear to be a promising alternative in addressing the limitations of DNNs and expanding their capabilities in the domain of robot control. However, it is important to note that SNNs currently face numerous challenges, and extensive research is required to fully unlock their potential. As of today, SNNs often exhibit lower performance compared to DNNs, particularly in high-dimensional problems such as robot control. The complexities introduced by temporal dynamics and spike-based computations pose additional difficulties in training and optimization.

However, the advantages outweigh the disadvantages. It is therefore interesting to explore the potential of SNNs to enhance robot control. Therefore, the main objective of this thesis is to investigate the feasibility and performance of training a DRL agent

utilizing SNNs for the task of controlling a robot with seven Degrees of freedom (DoF). Hence this study aims to address two interesting questions:

- What approach would be most suitable for employing DRL in conjunction with SNNs to learn continuous robot control in a high-dimensional state-action space?
- How does the performance of a SNN agent compare to an equivalent DNN agent in the context of continuous robot control?

If this work could demonstrate that an SNN agent can achieve comparable performance as a traditional DNN agent, while retaining the advantages of SNNs, it would provide strong justification for the application of SNNs in other complex tasks, particularly in the domain of mobile robotics. Since the application of SNNs could significantly increase the operating duration of robots due to the lower energy consumption. This would pave the way for leveraging the unique capabilities of SNNs in diverse real-world applications.

1.2 Overview

This section provides an overview of the objectives and specific milestones of this work. The research question described consists of several sub-disciplines: reinforcement learning, deep learning, and spiking neural networks. Therefore, a basic knowledge of these areas should be acquired, before starting this work. Hence, chapter 2 provides a basic, but only selective, overview of these areas relevant to this work.

After gaining a firm understanding of the fundamentals, a promising approach for combining DRL with SNNs should be selected, which is the first milestone of this thesis. For this reason, chapter 3 is reviewing the current state of the art, selecting a promising method, and explaining it in detail.

The second milestone of this thesis is described in chapter 4. Here the method and the implementation details of the SNN agent are provided. After implementing the agent, it needs to be trained on a predefined task within a simulation environment, which is also explained in chapter 4, together with other details regarding the training procedure. In order to provide a comprehensive comparison, not only a SNN agent is trained in this research, but also a traditional DNN agent. Thus, the performance of both agents can be compared later on.

In chapter 5, a set of criteria is defined against which the performance of the agents is compared. Followed, by a series of experiments, which are designed and carried out to

evaluate the established criteria. Further, the results of the experiments are presented. Finally, the obtained results are discussed in chapter 6 to address the main research question of this thesis: whether the SNN agent is capable of matching the performance of a DNN agent in the context of continuous robot control.

2 Fundamentals

The first section of this chapter provides an overview of the basics and principles that are relevant to understanding Reinforcement Learning and policy based algorithms. The second section of this chapter focuses on two main topics: SNNs and their underlying Leaky Integrate-and-Fire (LIF) model, as well as the STBP algorithm and population coding. Understanding these concepts is essential for understanding the neural network architecture used in this study.

2.1 Reinforcement Learning

The idea behind Reinforcement Learning can be observed in everyday life and is very similar to the way a child learns. Let us consider a child playing chess, as a simple and familiar scenario to explain the concept of Reinforcement Learning. So the child is the agent in this scenario, interacting with its environment, which in this case is the chessboard with all the chess pieces. In general, an agent can be a human, a program, a robot, or any other system, which interacts with its environment. But the agent can't change the rules or dynamics of the environment e.g., the game of chess. Regarding chess, this means that the agent can move the game pieces, which will change the state of the environment, but not the rules of the game. At each new time step, the agent must select an action to be executed from a set of possible actions, moving one of the chess pieces. Each new move is an interaction with the environment and results in a change of state. A state in the game of chess is the current configuration of chess pieces on the board. But in general, states can be actual states of a physical machine (e.g. on/off), an internal representation of a virtual environment, or abstract concepts. These states can be observed by the agent, either completely or partially. In chess, the rules and the position of all pieces are known and are everything that is relevant to make a decision. Therefore, chess is fully observable. This is different, with robot applications in the real world, for example grasping objects. Many of the parameters such as mass, centre of gravity, coefficient of friction, etc. are not observable. Therefore, the agent can only partially observe its environment, for example through visual sensors.

After each action, the agent receives a reward, which is only a numeric feedback signal that rewards desired behavior with a positive reward or punishes misbehavior with a negative reward. The reward in this context is the feedback or outcome the child receives after each move. For example, if the child successfully captures an opponent's piece, they may receive a positive reward. If the child makes a move that puts their king in danger, they might receive a negative reward. The overall goal of the agent is to maximize its reward. By doing so, the agent will learn a desired behavior autonomously over time or in the context of chess will learn its own strategies to win a game. In summary, the idea behind Reinforcement Learning is to learn which action to take in the current state, in order to maximize the reward in the long run. It is therefore a decision making problem.

2.1.1 Markov Decision Process

Markov Decision Processes are a standard way of formulating sequential decision making problems and represent a mathematically idealized form of the Reinforcement Learning problem. At each time step t the agent receives the current state $S_t \in \mathcal{S}$ of the environment, upon which the agent must decide on an action A_t from the set of possible actions $A(s)$. This action will lead to a transition into a successor state S_{t+1}. At the end of this transition, the agent will receive a numerical reward $R_{t+1} \in \mathcal{R} \in \mathbb{R}$. This feedback loop of selecting an action, transitioning into another state, and receiving a reward is called the Agent–Environment Interface, which is shown in Figure 2.1.

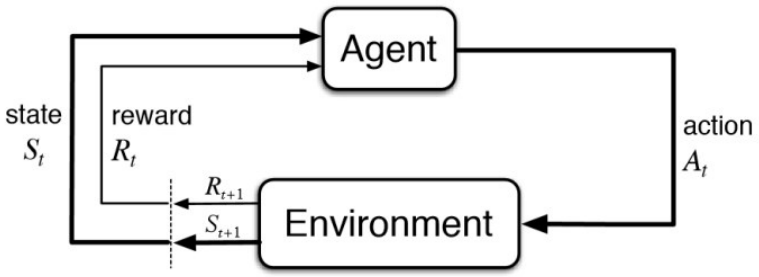


Figure 2.1: The Agent–Environment Interface. Every new time step t, the agent has to take an action A_t, which leads to a new state S_{t+1} and a reward R_{t+1} that the agent receives. [SB18]

Moving from the current state $S_t = s$ to the next state $S_{t+1} = s'$ is called a transition. A sequence of all transitions from start to end is called an episode, in chess, a single game is referred to as a single episode. A single transition $< s_0; a_0; r_1; s^{'} >$ consists of the state, the selected action, the received reward, and the successor state.

The dynamics of the environment are described by the probability distribution p. This probability distribution, formulated in Equation 2.1, indicates the probability of transitioning into the successor state s' and receiving reward r under the condition of starting from state s and selecting action a.

$$p(s', r|s, a) = Pr(S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a) \tag{2.1}$$

That feature is called the Markov property. That is, that the next state $s^{'}$ depends only on the current state s and the action a taken, but not on all other previous states and actions before. This indicates that s contains all the relevant information about the previous interactions with the environment. Considering chess, the position of all pieces on the board would serve as a Markov state. Because all relevant information for the further progress of the game can be extracted from the current configuration of the pieces. [SB18]

The goal of the agent is formalized as a reward function, rewarding desired behavior and punishing misbehavior. Therefore, the reward R can be interpreted as a feedback signal. By maximizing its cumulative reward, the agent will learn a behavior with which the task can be mastered. The cumulative reward is called the expected return G_t and is formulated in Equation 2.2.

$$G_t = \sum_{i=0}^{n} R_{i+1} \tag{2.2}$$

To distinguish between rewards, which can be obtained immediately, and rewards which can only be reached in the far future, a discount factor γ is introduced, see Equation 2.3. This enables the agent to prioritize rewards according to the time steps it will take the agent to reach them. This finite definition is also used for infinite episodes since this prioritization of rewards is immensely helpful during training.

$$G_t = \sum_{k=0}^{T} \gamma^k R_{t+k+1} \tag{2.3}$$

2.1.2 Value Functions and Policies

In order to maximize the expected Return G_t, the agent needs to estimate how useful it is to be in a certain state s. This is achieved by value functions $v(s)_\pi$. Value functions

try to estimate the usefulness of a state s. The usefulness is defined by how much future reward can be expected from the current state moving onward. Further, the value function $v(s)_\pi$ also depends on the policy π of the agent. A policy $\pi(a|s)$ assigns a probability to each possible action a in a state s. Thus, the policy describes the behavior of the agent. For instance, always choosing actions with the highest value are considered greedy policies. Because the probabilities of all other actions, than the one with the highest value, are 0. Such a behavior will lead to a different expected return G_t, then a behavior in which all possible actions are executed with the same probability. Therefore, value functions are defined with respect to a certain policy π. The expected value of state is described in the state-value function as in Equation 2.4. [SB18]

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s], \forall s \in S \tag{2.4}$$

Another possibility is to assign a value to a state-action pair (s, a). The value of a state-action pair is called a q-value $q_\pi(s, a)$. The q-value indicates the expected reward, starting in state s taking action a and thereafter following policy π. This is called the action-value function or Q-function, which is defined in Equation 2.5. [SB18]

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a] \tag{2.5}$$

In most cases, the true value of a state is unknown and only estimated. To distinguish those cases the true value function will be noted as $v_\pi(s), q_\pi(s, a)$ and the estimated value functions are written in upper cases $V_\pi(s), Q_\pi(s, a)$. Another value function that is used often, is the Advantage function $A(s, a)$, see Equation 2.6. The Advantage function tells how good it is to take a certain action a in a certain state s compared to the other available actions at that state s. Or in other words, it estimates the advantage of an action a over all other possible actions in that state s [SB18].

$$A(S_t, A_t) = Q(s, a) - V(s) \tag{2.6}$$

2.1.3 Policy Gradient

Policy gradient methods are a subset of policy search methods that directly optimize a parameterized policy $\pi(a|s, \boldsymbol{\theta})$, where $\boldsymbol{\theta}$ is the parameter vector on which the policy

depends. The objective is to improve the policy's performance by optimizing $\boldsymbol{\theta}$. The performance of the policy can be defined in two cases: in the episodic case, an episode ends after a certain number of steps or when a terminal state is reached, while in the continuous case, the agent can take an infinite number of steps. Following, only the episodic case is considered.

In order to improve the policy, the performance needs to be measured, with respect to the policy parameter $\boldsymbol{\theta}$. In Reinforcement Learning the performance can be measured as the cumulative reward. Therefore, the objective function for an episodic task can be formulated as:

$$\begin{aligned} J_G(\boldsymbol{\theta}) &= \mathbb{E}_{S_0 \sim d_0, \pi_{\boldsymbol{\theta}}} \left[\sum_{t=0}^{T} \gamma^t R_{t+1} \right] \\ &= \mathbb{E}_{S_0 \sim d_0} [\mathbb{E}_{\pi_{\boldsymbol{\theta}}} [G_t | S_t = S_0]] \\ &= \mathbb{E}_{S_0 \sim d_0} [\upsilon_{\pi_{\boldsymbol{\theta}}}(S_0)] \end{aligned} \quad , \tag{2.7}$$

where S_0 is the starting state, d_0 the start-state distribution and $\upsilon_{\pi_{\boldsymbol{\theta}}}$ is the true value function under $\pi_{\boldsymbol{\theta}}$. Now gradient ascent is used to update $\boldsymbol{\theta}$, to maximize the objective function $J(\boldsymbol{\theta})$, see Equation 2.8. [vH21]

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \tag{2.8}$$

To update $\boldsymbol{\theta}$ the gradient of the objective function $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ is required, which is called the policy gradient, defined in Equation 2.9. For an episodic task, the policy gradient is formed using the score function trick, where $\tau = S_0, A_0, R_1, S_1, A_1, R_1, S_2, ...$ is a trajectory.

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} J_{\boldsymbol{\theta}}(\pi) &= \mathbb{E}[G(\tau) \nabla_{\boldsymbol{\theta}} \log p(\tau)] \\ &= \mathbb{E}_{\pi}[G(\tau) \nabla_{\boldsymbol{\theta}} \sum_{t=0}^{T} \log \pi(A_t | S_t)] \\ &= \mathbb{E}_{\pi}[\sum_{t=0}^{T} (\gamma^t G_t) \nabla_{\boldsymbol{\theta}} \log \pi(A_t | S_t)] \end{aligned} \tag{2.9}$$

This basic definition of the policy gradient struggles with slow convergence or unstable learning, caused by a high variance of the gradient. The reason for the high variance of the gradient is that the Return G_t can vary a lot from one state to another state. To avoid this, an arbitrary baseline $b(s)$ can be included, to reduce variance, demonstrated

in Equation 2.10. The most common choice of a baseline is the state-value function $V_\pi(S_t)$, which can be interpreted as the average reward an agent receives if the agent starts in state s and acts according to policy π.

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} J_{\boldsymbol{\theta}}(\pi) &= \mathbb{E}_\pi[\sum_{t=0}^{T}(\gamma^t G_t - b(S_t))\nabla_{\boldsymbol{\theta}} \log \pi(A_t|S_t)] \\ &= \mathbb{E}_\pi[\sum_{t=0}^{T}(\gamma^t G_t - V(S_t))\nabla_{\boldsymbol{\theta}} \log \pi(A_t|S_t)] \\ &= \mathbb{E}_\pi[\sum_{t=0}^{T}(\gamma^t A(S_t, A_t))\nabla_{\boldsymbol{\theta}} \log \pi(A_t|S_t)]\end{aligned} \tag{2.10}$$

G_t is finally replaced by the Q-Function $Q(S_t, A_t)$, which leads to $A(S_t, A_t) = Q(S_t, A_t) - V(S_t))$. Based on this policy gradient the update rule can be rewritten as:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha[\gamma^t A(S_t, A_t)\nabla_{\boldsymbol{\theta}} \log \pi(A_t|S_t)]] \tag{2.11}$$

and the loss function for updating the policy can finally be defined as Equation 2.12.

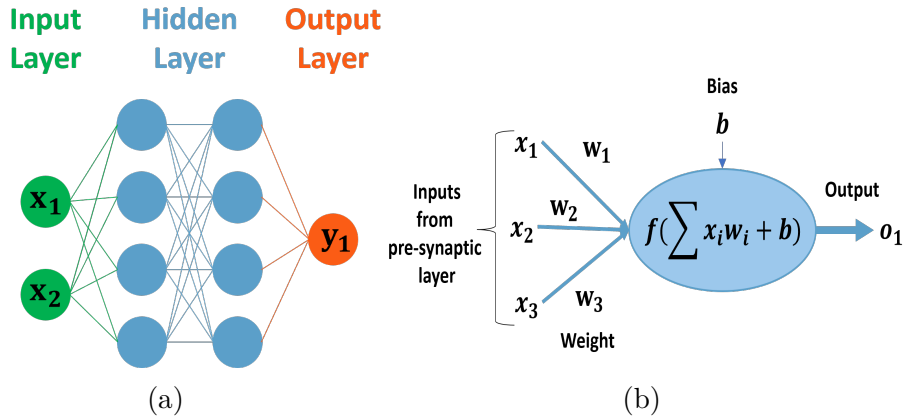
$$L^{PG} = \mathbb{E}_\pi \left[A_t \log \pi_{\boldsymbol{\theta}}(A_t, S_t)\right] \tag{2.12}$$

2.2 Neural Networks

Neural networks are computational models inspired by the structure of the biological nervous system. These networks are designed to mimic the behavior of interconnected neurons and have emerged as a powerful tool for solving complex problems in various domains, including image recognition, natural language processing, and reinforcement learning. This section gives a selective overview of the most important details, which are required for understanding this work. At first, it is explained in general what an artificial neuron is and how networks are built from neurons. Then it is explained how Spiking Neural Networks (SNNs) differ from traditional Deep Neural Networks (DNNs) and which specific neuron model is used in this work. For training SNNs the stochastic gradient descent algorithm can be used, in combination with an adapted version of the backpropagation algorithm, which is explained as well. Lastly, the framework of population coding is introduced.

2.2.1 From Neurons To Networks

Artificial neurons are the basic unit of every Artificial Neural Network (ANN). Because an ANN is only a collection of artificial neurons, which are connected with each other. Neurons in a neural network are typically organized into layers, forming a layered architecture. The most common types of layers are the input layer, hidden layers, and output layer, shown in Figure 2.2a. The input layer receives the initial input data, which is then propagated through the hidden layers, with each layer applying its transformation to the input. Finally, the output layer produces the network's final output, which could be a classification label, a predicted value, or an action to be taken.



(a) (b)

Figure 2.2: (a) Structure of a neural network, highlighting the three different types of layers. (b) Each neuron processes its input as a weighted sum, which is passed into an arbitrary activation function $f()$.

Each neuron in the neural network represents a mathematical function. The standard model for an artificial neuron is the M-P model from McCulloch and Pitts. The role of the neuron is to receive input signals x_i from the units in the previous layer, also called pre-synapitc layer, or from external sources and process them in the form of a weighted sum, described in Equation 2.13 and illustrated in Figure 2.2b.

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right) \tag{2.13}$$

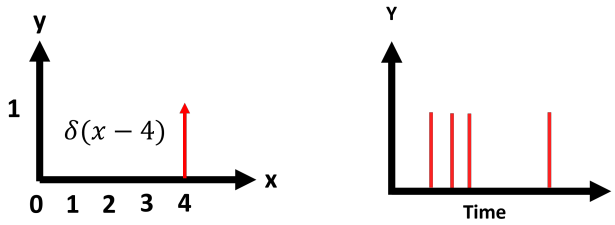
Each input x_i is multiplied by its associated weight w_i and n is the number of connections to neurons in the pre-synaptic layer. Then the individual products are summed together and a bias b is added. The result is then put into an activation function $f()$, which determines the output of the neuron. Standard activation functions for DNNs

are e.g. the Rectified Linear Unit (ReLU) function or the Sigmoid function. Both functions return continuous-valued signals. These activation functions introduce non-linearities into the network, enabling it to learn complex patterns and relationships in the data.

To make the neural network learn and adapt to specific tasks, a process known as training is performed. During training, the network is presented with a set of input-output pairs, often referred to as training examples or training datasets. The network adjusts its internal parameters, known as weights and biases, based on a learning algorithm and an objective or loss function that quantifies the network's performance. The goal is to minimize the discrepancy between the network's predicted outputs and the desired outputs for the given inputs. One of the most common learning algorithms is the Stochastic Gradient Descent (SGD) algorithm, which is covered in section 2.2.4.

2.2.2 Spiking Neural Networks

Spiking Neural Networks (SNNs) represent a specialized class of Artificial Neural Networks (ANNs) that aim to capture the temporal dynamics of the biological nervous system more accurately. In contrast to traditional neural networks, which perform computations using continuous functions, SNNs introduce a more biologically plausible approach by incorporating the concept of discrete-time events called spikes, illustrated in Figure 2.3b. Thus the neurons don't output continuous signals, but instead only a spike or no spike. These spikes are represented by using the Dirac function $\delta(t)$, shown Figure 2.3a. [GKNP14]



(a) (b)

Figure 2.3: (a) Visualization of the Dirac function $\delta(t)$. (b) Visualization of multiple spikes, called spike train.

This event driven nature enables SNNs to better model phenomena such as neural oscillations and spike-timing-dependent plasticity. Furthermore, SNNs can be implemented on specialized neuromorphic hardware, taking advantage of the event-driven

nature of spiking neurons and achieving high energy efficiency. In order to determine when a spike is generated, the internal state of the neuron needs to be modeled. The neuron model that is used in this work is explained in the following section.

2.2.3 Leaky Integrate-and-Fire Neuron Model

In order to capture the dynamics of biological neurons a model is required. One of these models used in computational neuroscience is the Leaky Integrate-and-Fire (LIF) neuron model. Unlike other models, which are capable of very accurately representing the dynamics of a biological neuron, the LIF neuron model is a simplistic model, which makes it computationally efficient.

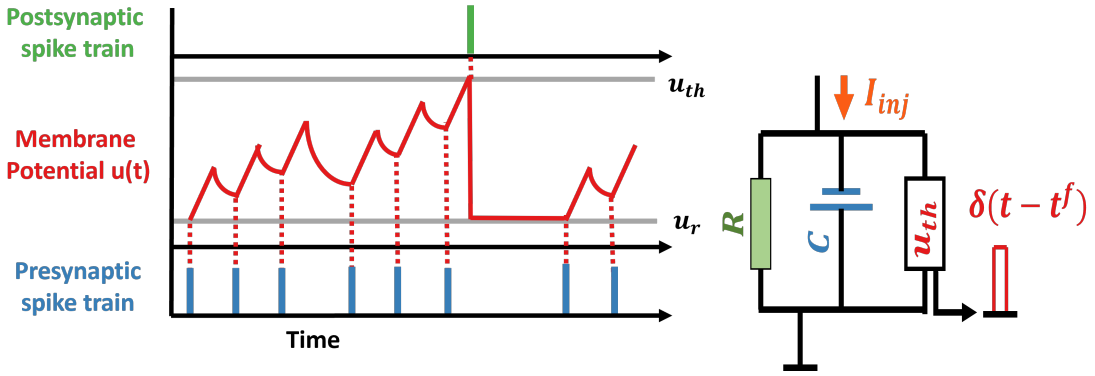
The LIF neuron model is based on the idea that a neuron integrates incoming electrical signals, called spike trains (see Figure 2.3b) from other neurons. Each spike train comes from a different neuron from the pre-synaptic layer and can consist of an arbitrary number of spikes $\delta(t - t_j)$ at time step t_j. Since every connection in the network is assigned to a weight w_i, each spike from the spike train is multiplied by the weight of the connection w_i. That way a weighted sum is calculated, called injection current $I_{inj}(t)$, which is taken as the input of the neuron. Equation 2.14 describes how $I_{inj}(t)$ is obtained, where n^l indicates the number of connections from the current neuron to the pre-synaptic neurons, n is the number of spikes inside the pre-synaptic spike train and w_i is the weight of the connection between the pre-synaptic neuron and the current neuron. [LX22]

$$I_{inj}(t) = \sum_{i=1}^{n^l} w_i \sum_{j=1}^{n} \delta(t - t_j) \tag{2.14}$$

In the LIF neuron model, the membrane of the neuron is modeled as a capacitor with a capacity C, which is charged by the injected current $I_{inj}(t)$. Since the cell membrane is not perfect, the charge will decay over time. This leakage is modeled by a finite leak resistance R. Therefore this leaky-integrator can be represented as a circuit consisting of a resistor R and capacitor C in parallel, driven by the input current $I_{inj}(t)$, illustrated in Figure 2.4b. The standard form of describing this circuit mathematically is formulated in Equation 2.15, where τ_m is the membrane time constant of the neuron. [GKNP14]

$$\tau_m \frac{\partial u}{\partial t} = -[u(t) - u_r] + RI_{inj}(t) \quad with\ \tau_m = RC \tag{2.15}$$

If the neuron receives an input, the membrane potential $u(t)$ is charged. If $u(t)$ exceeds a certain threshold u_{th}, the neuron spikes. After the spike, the membrane potential is reset to its resting potential u_r. If the neuron receives no input the charge decays over time exponentially until the membrane potential $u(t)$ reaches its resting potential u_r. This behavior is illustrated in Figure 2.4a.



(a) (b)

Figure 2.4: (a) Course of the membrane potential $u(t)$ according to the LIF Model. Once the membrane potential u reaches a certain threshold u_{th}, the neuron spikes and the potential is reset to u_r. (b) The LIF model as an electrical circuit.

2.2.4 Stochastic Gradient Decent

Stochastic Gradient Descent (SGD) is one of the most common learning algorithms used in machine learning to find the minimum of a loss function L. The loss function measures the difference between the predicted output x and the actual output y. The overall goal in deep learning is to optimize the parameters $\boldsymbol{\theta}$ of neural network $f(\boldsymbol{\theta})$ in a way that the behavior of the neural network is approximate to a target function. For ANNs the parameters optimized are the weights w_i and biases b_i of the neural network. To approximate the behavior of the neural network to a desired target function, SGD is used to find the minimum of the Loss function. Because a loss of zero states that the approximated function is equivalent to the target function. It is to mention that SGD can only guarantee to find a local minimum, even though the desired goal is to find the global minimum. In order to find the minimum of a loss function L, an arbitrary start point is chosen, shown in Figure 2.5. Then the gradient of the loss function L is formed and the parameters $\boldsymbol{\theta}$ are updated in the direction of the negative gradient, according to Equation 2.16 where α is the learning rate [SB18]. This reduces the loss

and thus the behavior of the neural network approaches the target function. Now the procedure is repeated until a minimum is reached.

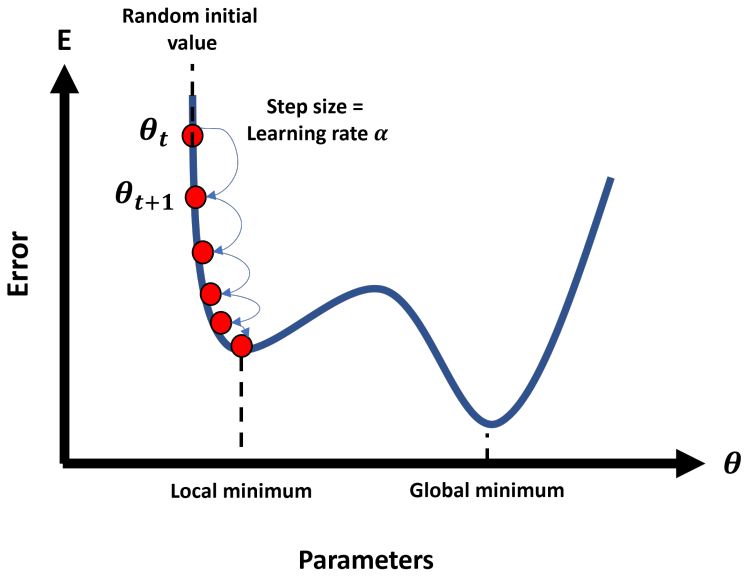


Figure 2.5: The visualized process of the Stochastic Gradient Descent algorithm, for finding the local minimum of a function. At the beginning at time step t an arbitrary start point is chosen on the loss function. From there the parameters $\boldsymbol{\theta}$ are updated in the direction of the negative gradient, until a minimum is found.

$$\boldsymbol{\theta}_{new} = \boldsymbol{\theta}_{old} + \alpha \nabla L \qquad \text{with} \quad \nabla L(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial L}{\partial w_1} \\ \frac{\partial L}{\partial b_1} \\ \vdots \\ \frac{\partial L}{\partial w_n} \\ \frac{\partial L}{\partial b_n} \end{pmatrix} \tag{2.16}$$

2.2.5 Spatiotemporal Backpropagation

As shown in Equation 2.16 the SGD algorithm requires the partial derivative of the Loss function for each weight and bias. For standard DNNs the Backpropagation algorithm [RHW86] provides an efficient way of calculating all partial derivatives for the gradient. But for SNNs temporal dynamics of the information should be considered, as well. Therefore the standard Backpropagation algorithm is extended to the Spatiotemporal Backpropagation (STBP) algorithm [WDL+18]. Before explaining the STBP algorithm, a notation is defined:

- t: time step
- n: nth layer
- $l(n)$: number of neurons in the nth layer
- w_{ij}: synaptic weight from the jth neuron in the pre-synaptic layer to the ith neuron in the post-synaptic layer
- o_j: output of the jth neuron where $o_j = 1$ denotes a spike activity and $o_j = 0$ denotes nothing occurs.
- $u_j^{t,n}$: membrane potential of neuron j in layer n at time step t
- b_i^n: bias of ith neuron in layer n

The output o_i of a neuron is defined as:

$$o_i^{t+1,n} = g(u_i^{t+1,n}) \tag{2.17}$$

with the membrane potential $u_i^{t+1,n}$, which is defined as:

$$u_i^{t+1,n} = u_i^{t,n} f(o_i^{t,n}) + x_i^{t+1,n} + b_i^n \tag{2.18}$$

and the input $x_i^{t+1,n}$:

$$x_i^{t+1,n} = \sum_{j=1}^{l(n-1)} w_{ij}^n o_j^{t+1,n-1} \tag{2.19}$$

where $g(x)$ is the output gate and generates a spike activity, when it is activated, illustrated in Equation 2.20. And $f(x)$ in $u_i^{t+1,n}$ controls the exponential decay of the membrane potential, see Equation 2.21.

$$g(x) = \begin{cases} 1, u \geq u_{th} \\ 0, u \leq u_{th} \end{cases} \tag{2.20}$$

$$f(x) = \tau e^{-\frac{x}{\tau}} \tag{2.21}$$

With these notations and equations 2.17 - 2.21 everything is setup for obtaining the partial derivatives of the loss function with respect to the individual weights and the biases. These can be obtained according to Equation 2.22 and Equation 2.23.

$$\frac{\partial L}{\partial \boldsymbol{W}^{\boldsymbol{n}}} = \sum_{t=1}^{T} \frac{\partial L}{\partial \boldsymbol{u}^{t,n}} \frac{\partial \boldsymbol{u}^{t,n}}{\partial \boldsymbol{x}^{t,n}} \frac{\partial \boldsymbol{x}^{t,n}}{\partial \boldsymbol{W}^{\boldsymbol{n}}} = \sum_{t=1}^{T} \frac{\partial L}{\partial \boldsymbol{u}^{t,n}} \boldsymbol{o}^{t,n-1^T} \tag{2.22}$$

$$\frac{\partial L}{\partial \boldsymbol{b}^{\boldsymbol{n}}} = \sum_{t=1}^{T} \frac{\partial L}{\partial \boldsymbol{u}^{t,n}} \frac{\partial \boldsymbol{u}^{t,n}}{\partial L \boldsymbol{b}^{\boldsymbol{n}}} \tag{2.23}$$

According to Equation 2.22 and Equation 2.23 $\frac{\partial L}{\partial \boldsymbol{u}^{t,n}}$ is required to calculate the partial derivatives of $\frac{\partial L}{\partial \boldsymbol{W}^{\boldsymbol{n}}}$ and $\frac{\partial L}{\partial \boldsymbol{b}^{\boldsymbol{n}}}$. The solution of $\frac{\partial L}{\partial \boldsymbol{u}^{t,n}}$ depends on the the Loss function L. In order to obtain $\frac{\partial L}{\partial \boldsymbol{u}^{t,n}}$ four different cases need to be considered. Because with the STBP algorithm, the error is not only propagated backwards through the network in the spatial domain but also sideways in the temporal domain, as illustrated in Figure 2.6.

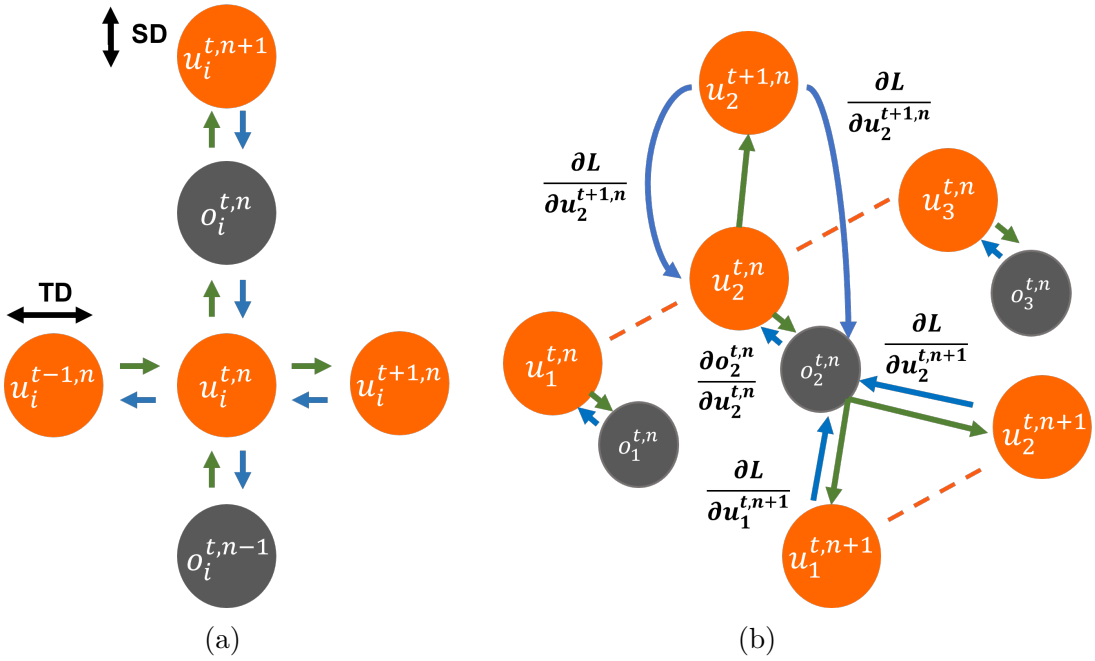


Figure 2.6: Error propagation in the STBP algorithm. (a) At the single-neuron level, the vertical path and horizontal path represent the error propagation in the spatial domain and temporal domain, respectively. (b) Similar propagation occurs at the network level, where the error in the spatial domain requires the multiply-accumulate operation like the feedforward computation. [WDL+18]

For the purpose of demonstration the following Loss function L is used:

$$L = \frac{1}{2S} \sum_{s=1}^{S} \left\| \boldsymbol{y_s} - \frac{1}{T} \sum_{t=1}^{T} \boldsymbol{o_s^{t,N}} \right\|_2^2 \tag{2.24}$$

where y_s and $\boldsymbol{o_s}$ denote the label vector of the sth training sample and the neuronal output vector of the last layer N. At first, we consider the two cases of backpropagating the error through the spatial domain, which are similar to the standard Backpropagation algorithm. As the name suggests the STBP algorithm starts at the end of the neural network and iterates backwards from there. Therefore, the first case starts at the output layer N. [WDL+18]

Case 1: $t = T$ at the output layer $n = N$

Starting from the output layer N the partial derivative of the Loss function can obtained directly as follows:

$$\frac{\partial L}{\partial u_i^{T,N}} = \frac{\partial L}{\partial o_i^{T,N}} \frac{\partial o_i^{T,N}}{\partial u_i^{T,N}} \tag{2.25}$$

where $\frac{\partial L}{\partial o_i^{T,N}}$ is defined as:

$$\frac{\partial L}{\partial o_i^{T,N}} = -\frac{1}{TS}(y_i - \frac{1}{T} \sum_{k=1}^{T} o_i^{k,N}) \tag{2.26}$$

Case 2: $t = T$ at a hidden layer $n < N$

Now that the partial derivative for the output layer in the spatial domain is computed, the partial derivatives for the hidden layers in the spatial domain have to be computed as well. First $\frac{\partial L}{\partial o_i^{T,n}}$ is computed as:

$$\frac{\partial L}{\partial o_i^{T,n}} = \sum_{j=1}^{l(n+1)} \frac{\partial L}{\partial o_j^{T,n+1}} \frac{\partial o_j^{T,n+1}}{\partial o_i^{T,n}} \tag{2.27}$$

It can be seen that the obtained partial derivatives $\frac{\partial L}{\partial o_j^{T,n+1}}$ from the layer before (e.g. the output layer) are required to calculate the partial derivatives for the current layer. With $\frac{\partial L}{\partial o_i^{T,n}}$ being calculated, $\frac{\partial L}{\partial u_i^{T,n}}$ can be obtained as:

$$\frac{\partial L}{\partial u_i^{T,n}} = \frac{\partial L}{\partial o_i^{T,n}} \frac{\partial o_i^{T,n}}{\partial u_i^{T,n}} \tag{2.28}$$

Case 3: $t < T$ at the output layer $n = N$

The third case starts again at the output layer N. But now the error is not propagated in the spatial domain, but in the temporal domain, see Figure 2.6. Now the derivative $\frac{\partial L}{\partial o_i^{t,N}}$ depends on the error propagation in the temporal domain. Based on the chain rule the following derivatives can be obtained for $\frac{\partial L}{\partial o_i^{t,N}}$ and $\frac{\partial L}{\partial u_i^{t,N}}$:

$$\frac{\partial L}{\partial o_i^{t,N}} = \frac{\partial L}{\partial o_i^{t+1,N}} \frac{\partial o_i^{t+1,N}}{\partial o_i^{t,N}} + \frac{\partial L}{\partial o_i^{T,N}} \tag{2.29}$$

$$\frac{\partial L}{\partial u_i^{t,N}} = \frac{\partial L}{\partial u_i^{t+1,N}} \frac{\partial u_i^{t+1,N}}{\partial u_i^{t,N}} \tag{2.30}$$

Case 4: $t < T$ at the hidden layers $n < N$

For the last case, the spatial domain and the temporal domain have to be considered for obtaining the derivative $\frac{\partial L}{\partial o_i^{t,n}}$ in the hidden layers. Just like in the second case, the weighted error signals from the upper layer are accumulated by the neuron. Additionally, the state space is iteratively unfolded based on the chain rule. Thus, each neuron receives the propagated error from self-feedback dynamics in the temporal domain. This results in the following equations for $\frac{\partial L}{\partial o_i^{t,n}}$ and $\frac{\partial L}{\partial u_i^{t,n}}$:

$$\frac{\partial L}{\partial o_i^{t,n}} = \sum_{j=1}^{l(n+1)} \frac{\partial L}{\partial o_j^{t,n+1}} \frac{\partial o_j^{t,n+1}}{\partial o_i^{t,n}} + \frac{\partial L}{\partial o_j^{t+1,n}} \frac{\partial o_i^{t+1,n}}{\partial o_i^{t,n}} \tag{2.31}$$

$$\frac{\partial L}{\partial u_i^{t,n}} = \frac{\partial L}{\partial o_i^{t,n}} \frac{\partial o_i^{t,n}}{\partial u_i^{t,n}} + \frac{\partial L}{\partial o_i^{t+1,n}} \frac{\partial o_i^{t+1,n}}{\partial u_i^{t,n}} \tag{2.32}$$

Based on those four cases the STBP algorithm can be used to apply SGD to a SNN, which is based on the LIF model.

2.2.6 Population Coding

Population coding is a theoretical framework used in neuroscience to describe how populations of neurons can represent information in the brain. It is used to encode or decode information by the collective activity of a group of neurons. The basic idea is that each neuron in a population encodes a small amount of information about the stimulus, and the overall pattern of activity across the population represents the

complete information about the stimulus. Since population coding relies on the firing pattern of a group of neurons rather than on the activity of individual neurons it is more robust against noise and reduces uncertainty. [GSK86]

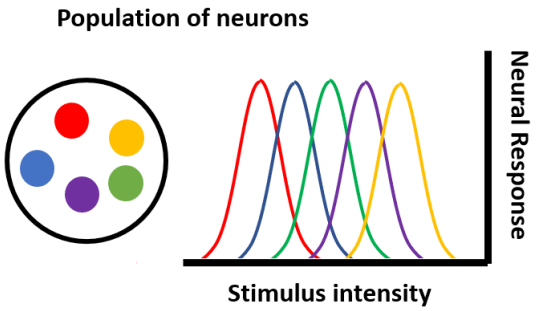


Figure 2.7: Population of neurons, the strength of the neuronal response depends on the intensity of the stimulus. Each neuron responds at its maximum when stimulated at a different intensity.

One specific model of population coding for encoding spatial information is position coding. In position coding, each neuron in a population has a preferred location in space that it is most sensitive to, which is represented with a Gaussian tuning curve, shown in Figure 2.7. Meaning that the fire intensity of the neuron depends on the location of the stimulus. The closer the stimulus is to the mean of the Gaussian tuning curve of a neuron, the more often the individual neuron will fire per second. With this method, continuous variables such as joint positions, joint velocities, or distances can be encoded. [WAN02]

3 State of the Art

In recent years, DRL has gained a lot of attention as a powerful tool for solving complex control tasks in robotics. The combination of deep neural networks with reinforcement learning algorithms, enables robots to learn control policies directly from raw sensor data, without relying on explicit programming or handcrafted features. For this reason DRL has been successfully applied to a wide range of challenging control tasks, such as robotic manipulation, locomotion, and navigation.

The aim of this work is to investigate the feasibility of training a policy with DRL using SNNs for controlling a high-DoF robot in joint space. Furthermore, the performance of the SNN agent will be compared to a policy trained with a DNN to assess its performance. Therefore, this chapter will cover the following topics. First, the current state of the art in the application of DRL with DNNs to the task of joint space control for high-DoF manipulators. Second, this chapter will explore different approaches for learning continuous control tasks using DRL with SNNs, as to the best of my knowledge, the control of a high-DoF robot with SNNs in the context of reinforcement learning has not yet been solved. Afterwards, the approach, which seems best suited for this task, is used for this work and introduced in detail. Finally, the Proximal Policy Optimization (PPO) algorithm is discussed, covering its advantages and disadvantages, and providing reasons for selecting this algorithm as the training method for this specific task.

3.1 Joint Space Control via Deep Reinforcement Learning with Deep Neural Networks

To reach a specific target position within the robot's workspace, the robot's joint angles must be calculated, which requires solving the inverse kinematics problem. The idea of using neural networks to learn the inverse kinematics of a robot is not new. E.g. D'Souza et al. [DVS01] approximate the nonlinear mapping of the inverse kinematics with local piecewise linear models. Other most recent research, e.g. [KIP+18]

and [STJL17], focuses on controlling the robot in end-effector space to execute manipulation tasks, such as grasping. However, a more biologically plausible approach would be direct control in the joint space.

Kumar et al. [KHS+21], have successfully trained a DRL agent with the PPO algorithm on the task of joint space control for the *Kinova Jaco* robot (6-DoF). The task of the agent is to drive the Tool-Center-Point (TCP) of the robot to a random target position inside a given workspace, by controlling the joint velocities of the robot actuators. The observation vector of the agent therefore includes its current joint angles, joint velocities, and the distance between the current TCP position and the target TCP position. The authors call this setup the Joint Action-space Learned Reacher (JAiLeR), which is illustrated in Figure 3.1.

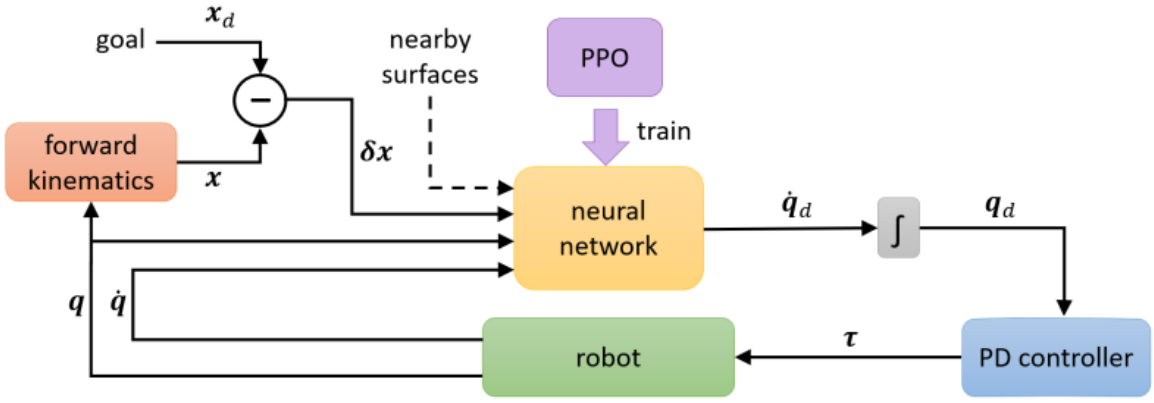


Figure 3.1: Overview of the proposed approach. Where δx is the distance between the end-effector and the target position, q ist the vector containing the joint angles and $\dot{q}$ contains the joint velocities. [KHS+21]

The workspace of the robot is defined as a torus with a major radius of 45 cm and a minor radius of 30 cm, centered at the base of the robot, with a rotational angle of 180°, as shown in Figure 3.2. To facilitate learning for the agent, curriculum learning is applied. For this purpose, the workspace is divided into four regions (R_1, R_2, R_3, R_4). Each Region is a partial torus and contains the previous region, to prevent catastrophic forgetting. How crucial curriculum learning is to learn this task successfully, becomes clear by looking at the results.

The results from [KHS+21] presented in Table 3.1 show that without curriculum learning the agent was not able to learn this task successfully and only achieved a success rate of 6.6 %. With curriculum learning instead the agent (JAiLeR) was able to successfully reach 96.5 % of the targets within 1 cm with an average error of 0.4 cm. The JAiLeR agent is able to outperform the classic operational space velocity controller

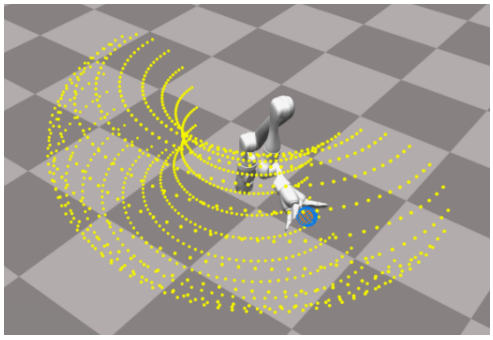


Figure 3.2: The workspace in this experiment for the Jaco robot is defined as torus with a rotation angle of 180°, visualized here with the small yellow dots. [KHS+21]

(OSC-V) and matches the performance of the simplified acceleration-based controller (OSC-A).

Table 3.1: Results of the trained PPO agent for the Jaco robot, tested on the full workspace (R_4)

	success in range of 1 cm (%)	average error (cm)	completion time (s)
without curriculum	6.6	2.9	-
JAiLeR	96.5	0.4	14.3 ± 2.2
OSC-V	53.1	0.8	20.8 ± 3.6
OSC-A	97.9	0.4	13.8 ± 2.3

This paper serves as a state-of-the-art reference for joint space control using DRL with DNNs. Since this master thesis aims also to learn joint space control using DRL, but with SNNs, this paper becomes a crucial benchmark for later comparison of the results.

3.2 Deep Reinforcement Learning with Spiking Neural Networks for continuous control tasks

Until now there is no unified approach for training DRL agents on SNNs. Recently, the literature has grown up around introducing SNNs into Reinforcement Learning algorithms. Typically these approaches ([RV.07], [FSG13a],[YWYT19],[OS13]) are based on reward-modulated local plasticity rules, such as Spike-Timing-Dependent Plasticity (STDP) [CD08]. These approaches demonstrated to perform well in low-dimensional

control problems, e.g. for lane keeping [BMH+18] or learning a simple navigation [FSG13b]. But the results of these approaches show, that they only perform well in low-dimensional tasks. For high-dimensional problems, these approaches do not provide a satisfactory result.

In contrast, an approach that has been shown to work even for high-dimensional problems is the conversion of a trained DNN into an SNN ([PHS+19a],[TPK20]). The problem with this approach is that the performance of the transformed SNN is often inferior to the performance of the DNN [RSPR20]. Additionally, the transformed SNN requires large time steps for inference that dramatically increases the energy cost, which compromises one of the biggest advantages of SNNs over DNNs.

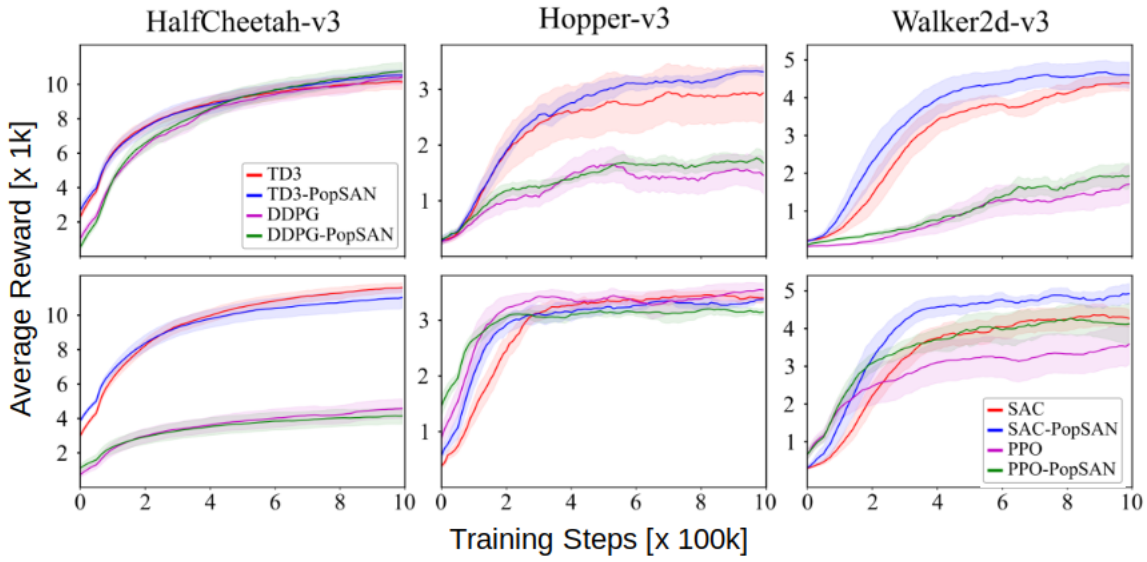


Figure 3.3: The average reward obtained by different off-policy algorithms and the Population-coded Spiking Actor Network (PopSAN) version of these off-policy algorithms in the classic OpenAi-Gym environments HalfCheetah-v3, Hopper-v3, and Walker2d-v3. [TKYM20]

Another approach to successfully solving high-dimensional state-action spaces is a hybrid approach. Tang et al. [TKYM20] used an Actor-Critic framework to train different DRL agents with different DRL algorithms, such as PPO, SAC, DDPG, and TD3. The authors called their approach Population-coded Spiking Actor Network (PopSAN). As the name suggests only the Actor is run on a SNN, while the Critic remains on a DNN. In order to convert the observations into activation signals for the SNN, the technique of population coding is used. The output of the SNN is then again decoded from spikes into continuous action signals. After training, the critic is not used anymore and can be discarded, allowing the trained agent to run solely on the SNN, which can reduce

energy consumption drastically. These hybrid agents were trained and run on the classic OpenAi-Gym control environments (Walker2d, Ant, Hopper, HalfCheetah). After training the performance of these hybrid agents was compared against the performance of the standard agents, trained on a DNN. The results are shown in Figure 3.3.

As it can be seen in Figure 3.3, these hybrid agents are able to match the performance of the standard agents, and in some cases slightly outperform them. Therefore, this approach seems very promising when it comes to solving high-dimensional continuous control problems. But the main advantage of the PopSAN is its energy efficiency. Because the actor is a SNN it can be run on neuromorphic Hardware. In this case, the agent was executed on the Loihi Chip [DSL+18], achieving an energy consumption of 11.98 µJ/Inf. In comparison to that the DNN agent required 1670.94 µJ/Inf, which made the SNN agent 140 times more energy-efficient compared to the deep actor network, while achieving the same performance.

When it comes to the specific task of controlling a high-DoF robot, using DRL with SNNs, very little work has been done yet. In [OKG23] the authors trained a six-DoF robotic arm on the task of target-detection, target-reaching, and collision avoidance, using a hybrid approach with the Deep Deterministic Policy Gradient algorithm, where only the actor is run on a SNN. But this work focused less on the control and more on the other named aspects. E.g. They evaluated the robot starting only in five different positions and the robot always had to reach the same target position. Further, they counted the target as achieved if the agent could approach the target to seven centimeters or less. Therefore, it still needs to be shown, that the task of target reaching for random target positions and starting positions can be learned via DRL using SNNs.

Since the PopSAN approach from Tang et al. [TKYM20] seemed to work best for high dimensional state-action-spaces, it is used as the main approach of this work. Therefore, a detailed description of how the PopSAN works is given in the next section.

3.3 Population-Coded Spiking Actor Network

This section provides a detailed explanation of the PopSAN, describing its architecture and operation in detail. The PopSAN consists of three modules and the Deep Critic Network, shown in Figure 3.4.

The first module is the encoder module. The encoder is responsible for transforming the input observation into activation signals for the SNN. The encoding process is performed using population coding. For each dimension of the observation i a population

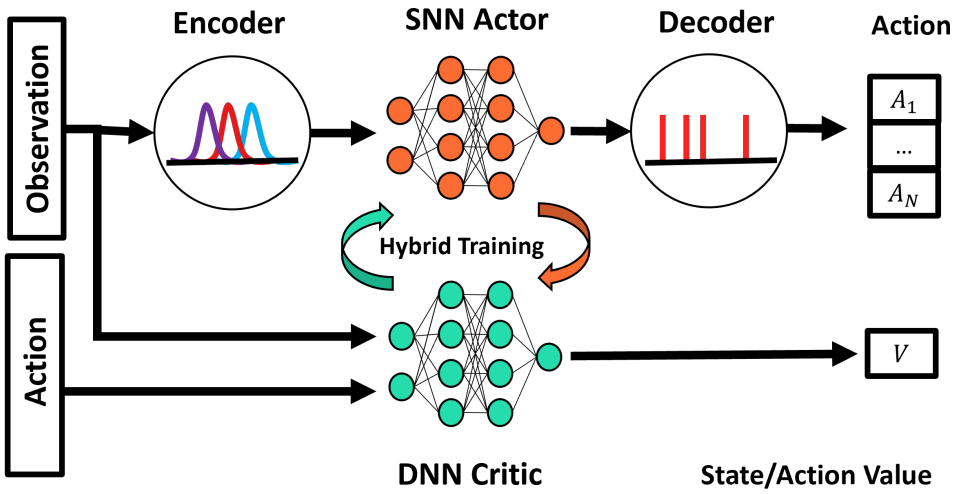


Figure 3.4: Overview of the PopSAN setup. At first the observation is encoded into spikes by the encoder module, then these spikes are input into the SNN and at the end these spikes are decoded into action values. Then the action with the current state are input into the DNN critic.

of neurons E_i is created to encode it. The neurons in E_i have Gaussian receptive fields $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$. $\boldsymbol{\mu}$ is uniformly distributed over the observation space s and $\boldsymbol{\sigma}$ is set to be large enough to ensure that the activity of the population is not zero in the whole space of s. [TKYM20]

At first, the observation values are transformed into a stimulation strength $\boldsymbol{A_E}$ for each neuron, according to Equation 3.1.

$$\boldsymbol{A_E} = \exp(-\frac{1}{2}((\frac{s-\boldsymbol{\mu}}{\boldsymbol{\sigma}\dot{\mathrm{t}}})^2) \tag{3.1}$$

$\boldsymbol{A_E}$ is then used for the generation of spikes for each neuron. This spike generation is based on deterministic encoding. Here $\boldsymbol{A_E}$ is seen as the presynaptic input to the neurons, which are simulated as one-step soft reset IF (Integrate and Fire) neurons. The behavior of the neurons is described by Equation 3.2, where k denotes the index of a neuron in E, ϵ is a constant, and o_k is the output of the neuron, spike or no spike.

$$\begin{aligned} \boldsymbol{v}(t) &= \boldsymbol{v}(t-1) + \boldsymbol{A_E} \\ o_k(t) = 1 \;\&\; v_k(t) &= v_k(t) - (1-\epsilon), \quad if\ v_k(t) > 1-\epsilon \end{aligned} \tag{3.2}$$

The parameters $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ of the encoder module are also trainable and updated during training. The parameters for each input population $i, i \in 1, ..., N$ are updated inde-

pendently according to Equation 3.3. Where L is the loss function and depends on the selected algorithm.

$$\begin{aligned} \nabla_{\boldsymbol{\mu}^{(i)}} L &= \sum_{t=1}^{T} \nabla_{\boldsymbol{o}_i^{(t)(0)}} L \cdot \boldsymbol{A}_{\boldsymbol{E}}^{(i)} \cdot \frac{s_i - \boldsymbol{\mu}^{(i)}}{\boldsymbol{\sigma}^{(i)^2}} \\ \nabla_{\boldsymbol{\sigma}^{(i)}} L &= \sum_{t=1}^{T} \nabla_{\boldsymbol{o}_i^{(t)(0)}} L \cdot \boldsymbol{A}_{\boldsymbol{E}}^{(i)} \cdot \frac{(s_i - \boldsymbol{\mu}^{(i)})^2}{\boldsymbol{\sigma}^{(i)^3}} \end{aligned} \tag{3.3}$$

Now that the observation is encoded into spikes, these spikes are fed into the SNN, which is the second module. The behavior of the individual neuron in the SNN is based on the LIF neuron model. In order to train the network, the stochastic gradient descent algorithm is used, where the gradient of a spike is approximated as a rectangular function $z(v)$, defined in Equation 3.4. With a being the threshold window for passing the gradient, v is the membrane potential, and v_{th} is the threshold of the membrane potential.

$$z(v) = \begin{cases} 1 & \text{if} \quad |v - v_{th}| < a \\ 0 & \text{otherwise} \end{cases} \tag{3.4}$$

To obtain the gradient of the loss for each layer k with respect to the individual weights $\boldsymbol{w}$ and biases $\boldsymbol{b}$, the STBP algorithm is used to collect all backpropagated gradients from all time steps. Thus the gradient can be calculated as follows:

$$\begin{aligned} \nabla_{\boldsymbol{w}^{(k)}} L &= \sum_{t=1}^{T} \boldsymbol{o}^{(t)(k-1)} L \cdot \nabla_{\boldsymbol{c}^{(t)(k)}} L \\ \nabla_{\boldsymbol{b}^{(k)}} L &= \nabla_{\boldsymbol{c}^{(t)(k)}} L \end{aligned} \tag{3.5}$$

where $\boldsymbol{c}^{(t)(k)}$ is the input current depending on the presynaptic spikes $\boldsymbol{o}$. Finally, the output spikes. of the SNN must be decoded into continuous actions. Therefore a decoder module, based on a population of neurons, is used again. This time each population of neurons D_i represents a certain dimension of the action-space A. The decoding process is done in two phases. First, the firing rate $\boldsymbol{fr}$ is obtained, by summing up the spikes of each neuron for T time steps. Next the action a is returned as the weighted sum of the firing rate $\boldsymbol{fr}$. Just as the encoder, so are the parameters of the decoder trainable. The parameters of the decoder population $i, i \in 1, ..., M$ are updated independently as follows:

$$\begin{aligned} \nabla_{\boldsymbol{w}_d^{(i)}} L &= \nabla_{a_i} L \cdot \boldsymbol{W}_d^{(i)} \cdot \boldsymbol{fr}^{(i)} \\ \nabla_{b_d^{(i)}} L &= \nabla_{a_i} L \cdot \boldsymbol{W}_d^{(i)} \end{aligned} \tag{3.6}$$

The algorithm which describes a complete forward pass through the PopSAN is written in pseudo code in Algorithm 1. The use of a hybrid approach such as the PopSAN brings with it a number of advantages and disadvantages. The greatest advantage of such systems is its energy efficiency, when executed on neuromorphic hardware. Such systems are also more biological plausible and they can potentially encode and transmit more information per spike, leading to higher information capacity in the network. However, such systems come at the cost of increased complexity, which makes training more difficult.

Algorithm 1 Forward propagation through PopSAN

Randomly initialize weight matrices $\boldsymbol{W}$ and biases $\boldsymbol{b}$ for each SNN layer
Initialize encoding means $\boldsymbol{\mu}$ and standard deviations $\boldsymbol{\sigma}$ for all input populations
Randomly initialize decoding weight vectors $\boldsymbol{W_d}$ and bias $\boldsymbol{b_d}$ for each action dimension
N-dimensional observation, s
Spikes from input populations generated by the encoder module: $\boldsymbol{X} = Encoder(\boldsymbol{s}, \boldsymbol{\mu}, \boldsymbol{\sigma})$
for $t = 1, ..., T$ **do**
    Spikes from input populations at timestep t: $\boldsymbol{o}^{(t)(0)} = \boldsymbol{X}^{(t)}$
    for $k = 1, ..., K$ **do**
        Update LIF neurons in layer k at timestep t based on spikes from layer $k-1$
        $\boldsymbol{c}^{(t)(k)} = d_c \cdot \boldsymbol{c}^{(t-1)(k)} + \boldsymbol{W}^{(k)} \boldsymbol{o}^{(t)(k-1)} + \boldsymbol{b}^{(k)}$
        $\boldsymbol{v}^{(t)(k)} = d_v \cdot \boldsymbol{v}^{(t-1)(k)} \cdot (1 - \boldsymbol{o}^{(t-1)(k)}) + \boldsymbol{c}^{(t)(k)}$
        $\boldsymbol{o}^{(t)(k)} = Threshold(\boldsymbol{v}^{(t)(k)})$
    end for
end for
M-dimensional action a generated by the decoder module:
Sum up the spikes of output populations: $\boldsymbol{sc} = \sum_{t=1}^{T} \boldsymbol{o}^{(t)(K)}$
for $i = 1, ..., M$ **do**
    Compute firing rates of the ith output population: $\boldsymbol{fr}^{(i)} = \frac{\boldsymbol{sc}^{(i)}}{T}$
    Compute ith dimension of action: $a^i = \boldsymbol{W}_d^{(i)} \cdot \boldsymbol{fr}^{(i)} + b_d^{(i)}$
end for

3.4 Proximal Policy Optimization

The algorithm used for training the PopSAN is the Proximal Policy Optimization (PPO) [SWD+17] algorithm, which belongs to the policy gradient methods and is an on-policy algorithm. Off-policy methods are using a so-called behavior policy in order

to generate samples, but are optimizing a different policy. On-policy algorithms on the other hand improve the same policy of the agent directly that is used to generate samples. The challenge thereby is to improve the current policy as much as possible, without stepping too far. If the new policy π_θ is too far away from the old policy $\pi_{\boldsymbol{\theta}_{old}}$, then this could cause a drop in performance. Two primary variants of the PPO algorithm exist, PPO-Penalty and PPO-Clip. In the following, only PPO-Clip will be discussed. To update its current policy parameters, PPO-Clip uses the following update rule:

$$\boldsymbol{\theta}_{new} = \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{s,a\ \boldsymbol{\theta}_{old}} \left[L^{clip}(s, a, \boldsymbol{\theta}_{old}, \boldsymbol{\theta}) \right] \tag{3.7}$$

The objective function L^{clip} is defined as:

$$L^{clip}(s, a, \boldsymbol{\theta}_{old}, \boldsymbol{\theta}) = \min \left(\frac{\pi(a|s, \boldsymbol{\theta})}{\pi(a|s, \boldsymbol{\theta}_{old})} A^{\pi_{\boldsymbol{\theta}_{old}}}(s, a), g(\epsilon, A^{\pi_{\boldsymbol{\theta}_{old}}}(s, a)) \right) \tag{3.8}$$

where

$$g(\epsilon, A(s, a)) = \begin{cases} (1+\epsilon)A(s, a) & \text{for} \quad A(s, a) \geq 0 \\ (1-\epsilon)A(s, a) & \text{for} \quad A(s, a) < 0 \end{cases} \tag{3.9}$$

The main idea of Equation 3.8 is to increase the likelihood of actions with a high advantage $A(s, a)$ and reduce the probability of actions with a low advantage. This idea originates from the Loss function of the Policy Gradient L^{PG} (see Equation 2.12). But the original Loss function L^{PG} struggles with stepping too far away, when the old policy $\pi_{\boldsymbol{\theta}_{old}}$ is updated, which results in a performance collapse. Therefore, PPO-Clip introduces the clipping term $g(\epsilon, A(s, a))$, see Equation 3.9. The term changes in dependence on A. If the Advantage $A(s, a)$ is positive L^{clip} reduces to:

$$L^{clip}(s, a, \boldsymbol{\theta}_{old}, \theta) = \min \left(\frac{\pi(a|s, \theta)}{\pi(a|s, \boldsymbol{\theta}_{old})}, (1+\epsilon) \right) A^{\pi_{\boldsymbol{\theta}_{old}}}(s, a) \tag{3.10}$$

Remember if an action a has a high Advantage $A(s, a)$, the likelihood of this action being chosen should be increased. Therefore, the new policy $\pi(a|s, \boldsymbol{\theta})$ is divided by the old policy $\pi(a|s, \boldsymbol{\theta}_{old})$. If the new policy $\pi(a|s, \boldsymbol{\theta})$ increases the likelihood of an action, the result will be greater one. Meaning, that if the advantage is positive, the objective function L^{clip} will be increased. The *min*-term limits the update to $(1+\epsilon)$, where ϵ is

a hyper parameter which defines the maximum of how far the new policy can be away from the old policy. In the other case of the Advantage being negative, the likeliness of that action should be decreased and L^{clip} reduces to:

$$L^{clip}(s, a, \boldsymbol{\theta}_{old}, \boldsymbol{\theta}) = \max\left(\frac{\pi(a|s, \boldsymbol{\theta})}{\pi(a|s, \boldsymbol{\theta}_{old})}, (1-\epsilon)\right) A^{\pi_{\boldsymbol{\theta}_{old}}}(s, a) \tag{3.11}$$

The mechanics of this equation remain the same, but now, because the Advantage value is negative, the objective function will be increased, if the new policy $\pi(a|s, \boldsymbol{\theta})$ decreases the likeliness of an action a with a negative Advantage. Instead of a min-term, a max-term is used to limit the update of L^{clip} to $(1+\epsilon)$.

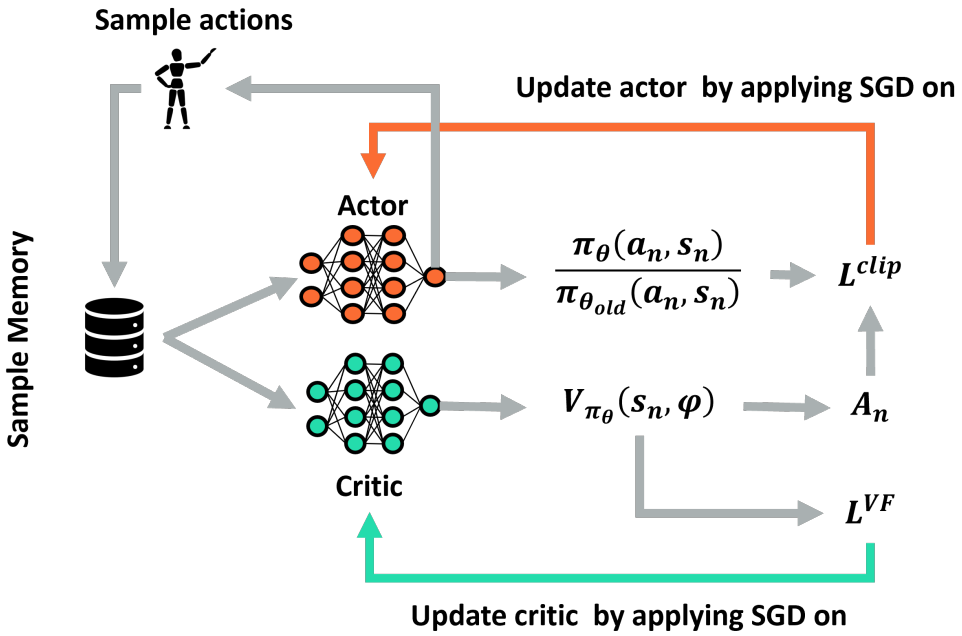


Figure 3.5: Flow chart of the PPO algorithm. At first the actor receives the current observation and outputs an action, which are stored in the sample memory. The critic receives then the state at time step t and the action that was chosen by the actor at time step t as an input. Based on this input the value $V_{\boldsymbol{\phi}_{\boldsymbol{\theta}}}$ of this state-action pair is estimated by the critic. This value is then used to calculate the loss for the critic L^{VF} and the actor L^{clip}.

The PPO algorithm in this work is based on the Actor-Critic framework [SB98] (see Figure 3.5), by which the policy and value function is simultaneously approximated using two separate neural networks named actor and critic. The actor network denoted as $\boldsymbol{\theta}$ represents the policy $\pi(s, a, \boldsymbol{\theta})$ and receives the current state s as an input. Based

on the current state the actor must choose an action a. Using this actor network n-steps are sampled in the environment. Each transition s, a, r_t, s' of these n-steps is stored.

The task of the Critic Network, denoted as $\boldsymbol{\phi}$, is to evaluate action a, executed by the actor. Hence the Critic network represents the value function $V_{\boldsymbol{\phi}}(s)$. Based on the estimated value function the advantage function is estimated as follows:

$$A = r + \gamma V_{\boldsymbol{\phi\theta}}(s'; \boldsymbol{\phi}) - V_{\boldsymbol{\phi\theta}}(s; \boldsymbol{\phi}) \tag{3.12}$$

A_t is then used to compute L^{clip}, which is required to update the actor network $\boldsymbol{\theta}$ Equation 3.7. In order to update the critic network $\boldsymbol{\phi}$ another loss function is required:

$$L^{VF} = (V_{\boldsymbol{\phi}}(s) - G)^2 \tag{3.13}$$

Based on Equation 3.13 the critic is updated over time, which leads to a better estimation of the advantage and finally to an improved actor. To obtain the estimated return G, the transitions of the current run are stored in memory. But after computing the Loss these transitions are deleted. Therefore, it is not comparable to an experience replay buffer. Finally, both Loss functions are combined and an entropy bonus S is added, which encourages exploration, see Equation 3.14. A final description of the algorithm is provided in Algorithm 2. The main disadvantage of the PPO algorithm is its sample inefficiency. Otherwise, the PPO tends to be a top-performing algorithm and performs well in continuous high-dimensional state-action spaces and it is easy to find working hyperparameters.

$$L_{Clip+VF+S} = \hat{\mathbb{E}}[L_{Clip}(\boldsymbol{\theta}) - c_1 L_{VF}(\boldsymbol{\theta}) + c_2 S[\pi_{\boldsymbol{\theta}}](s)] \tag{3.14}$$

Algorithm 2 PPO-Clip, Actor-Critic Style

1: **for** iteration= $1, 2, ...$ **do**
2: **for** actor= $1, 2, ..., N$ **do**
3: Run policy $\pi_{\boldsymbol{\theta}_{old}}$ in environment for T timesteps
4: Compute advantage estimate $A_1, ..., A_T$
5: **end for**
6: Optimize surrogate L wrt $\boldsymbol{\theta}$, with K epochs and minibatch size $M \leq NT$
7: Update actor $\boldsymbol{\theta_{old}} \leftarrow \boldsymbol{\theta}$
8: **end for**

4 Method and Implementation

This chapter is divided into two parts. The first part explains the method used in this thesis. The second part focuses on the implementation. Starting with the implementation of PopSAN agent, followed by a detailed explanation of the simulation environment and how the control task is formulated within the reinforcement learning framework. Next, the hardware used in training is described and finally, the implementation for transferring the agent from simulation onto a real robot is described.

4.1 Method

The thesis aims to investigate two main questions. First, what approach can be used to learn control of a high-dimensional robot in joint space using DRL with SNNs? Second, how does the performance of this approach compare with that of a state-of-the-art DNN agent?

In order to answer the first question, chapter 3 reviews different state of the art approaches for using DRL with SNNs on complex continuous control problems. By reviewing different approaches, it becomes clear that the PopSAN approach, might be best suited for the task of controlling a high-DoF robot, since it reached state-of-the-art performance for other complex control problems in simulation. Therefore, this work will utilize a PopSAN architecture with the PPO algorithm for training a DRL agent with a SNNs. The PPO algorithm is used since it is a robust state-of-the-art algorithm, which performs very well in high-dimensional continuous state-action spaces.

In order to answer the second research question and have a fair comparison between the performance of the PopSAN agent and a standard agent, a DNN-based agent is also trained with the PPO algorithm. That way the results of both agents can be compared later on. Therefore the first step of the method shown in Figure 4.1 is to implement both agents. Training such a complex task on a real robot could take weeks and includes the risk of causing damage to the robot. Therefore, creating a controlled simulation

environment where the agent can safely explore and learn without the risk of causing damage is essential to the success of this training. Furthermore, simulations allow for fast and parallelized training, enabling agents to undergo thousands of episodes in a shorter time span. Because of that the second step shown in Figure 4.1, is the creation of a simulation environment and the training of the agents.

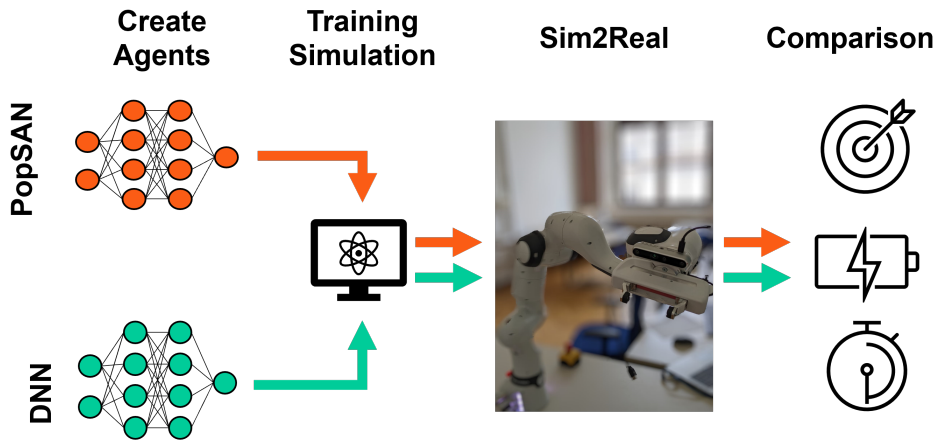


Figure 4.1: Visualization of the method. At first the PopSAN agent is implemented, as well as the DNN agent for reference. Then a simulation environment is created, where both agents are trained. Next the communication between the real robot and the trained agents is implemented. At last the performance of the agents is evaluated under different criteria.

Another critical aspect is to ensure that the SNN agent is able to transfer its in simulation learned ability onto a real robot. Hence, both agents have to be applied to a real robot. For this, the communication and control must be implemented for a real robot. After these preparations have been completed, the performance of the agents can be evaluated. Once both agents have been trained in a simulated environment and all the necessary preparations have been made to perform the agents' actions on a real robot, the evaluation will begin. Therefore in the last step, the performance of the agents is evaluated in different experiments and compared under a certain set of criteria, which are defined in chapter 5.

4.2 Implementation

This section covers the specific details of the implementation. Starting with the implementation of the PopSAN agent, followed by the implementation of the simulation

environment and the specific structure of a single episode. It then goes on to explain how the agents will be trained and the hardware settings on which they will be trained. Finally, an overview is given of the architecture used to communicate and control the real robot. The complete implementation of the work can be found on the cd attached to the end of the work.

4.2.1 PopSAN

The implementation of the PopSAN agent with the PPO algorithm was built upon an existing GitHub repository (`https://github.com/combra-lab/pop-spiking-deep-rl`) provided by the original authors of the paper [TKYM20]. This repository served as the foundation for the implementation, which was adapted slightly to meet the specific research requirements and relies completely on PyTorch. As illustrated in Figure 3.4, the PopSAN architecture consists of four components. First, the Critic, which is a standard DNN implemented as a fully connected network with two hidden layers:

- layer 1: hidden-layer-size: 256,
- layer 2: hidden-layer-size: 256.

Next is the encoder module, which uses population coding, to encode the observation into activation signals for the SNN. The encoder creates for each of the fifteen observation dimensions a population of ten neurons, which are then used to generate the spike input for the SNN. The SNN receives then 150 spikes to its input layer. The SNN is implemented as a fully connected network with two hidden layers:

- layer 1: hidden-layer-size: 256,
- layer 2: hidden-layer-size: 256.

The output layer of the SNN outputs 10 spikes for each of the six dimensions of the action space. These spike trains are then fed into the decoder. Which decodes these spikes into joint velocities, by the formula mentioned in the Algorithm 1. The complete structure of the PopSAN architecture remains equivalent to the structure of the original authors, as shown in Figure 3.4. The hyperparameters used for training are shown in the appendix in Table A.1a.

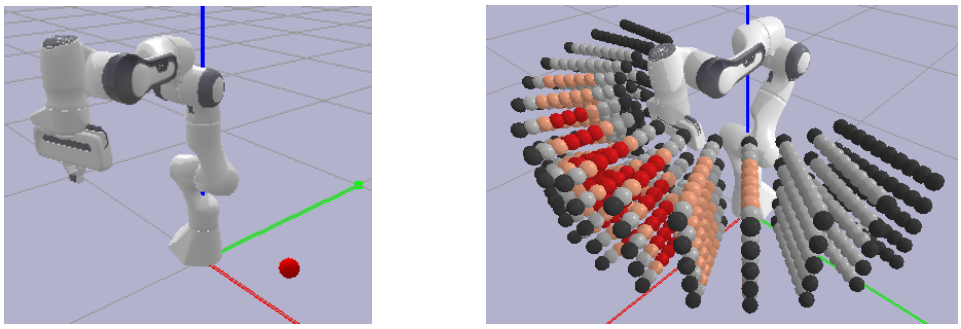
Additionally to the PopSAN agent, another agent was trained on a standard DNN. This is done for comparing the results of both agents later on. The DNN agent is created with *RlLib*, which is an open-source library for reinforcement learning. *RlLib* uses also an actor-critic implementation of the PPO algorithm. For this work, both,

the actor and the critic are implemented as fully connected networks with two hidden layers. Where each hidden layer has a size of 256 neurons. The hyperparameters that are used for training are listed in the appendix in Table A.1b.

4.2.2 Simulation Environment

The next step after implementing the agents is to create a simulation environment. This is done using *PyBullet*. *PyBullet* is a Python module for physics simulation and is based on the Bullet Physics SDK [CB21]. It supports various sensors, actuators, and terrains. It is therefore often used by roboticists for the application of reinforcement learning to robots. Besides *PyBullet* there are various other simulation tools available such as *MuJuCo* or *Gazeboo*. The reasons for choosing *PyBullet* for this project are that it is very beginner friendly, well-documented, and sufficient for this task.

The main objective of this work is to investigate the ability of the PopSAN agent to learn to control a high-DoF robot. For this purpose, the following task is constructed. The PopSAN agent must move the TCP of the robot to a random target position inside its workspace, by controlling the joint velocities of the robot. Therefore, the simulation environment contains only a Panda robot, which is placed on the ground, as shown in Figure 4.2a. The model for the robot is provided by *PyBullet* itself. Around the robot, at an angle of 180°, the workspace is defined as a torus with a major radius of 70 cm and a minor radius of 40 cm, centered on the base of the robot, see Figure 4.2b.



(a) (b)

Figure 4.2: (a) Setup of the task in the simulation environment. The agent has to reach the random target position (here marked with a red ball), (b) Visualization of the workspace, which is divided into four regions, for the purpose of curriculum learning (R_1 = red, R_2 = skin color, R_3 = gray, R_4 = black).

At the beginning of each new episode, a random target position inside the given workspace is generated and the robot is placed at a random starting position inside

the workspace. Now the agent runs for 500 time steps, which gives the agent enough time to achieve its goal. The goal for the agent is, to minimize the distance between the end-effector and the target position. Each episode is terminated exactly after 500 time steps, besides that there are no other termination conditions.

Each time step the agent receives at first its current state S, which is a multi-dimensional vector $S = (\delta x, q, \dot{q})$. The state contains the current distance δx (3) in x, y, and z between the TCP position and target position, the current joint angles q (6) and the current joint velocities $\dot{q}$ (6). Based on the current state, the agent has to output the action vector $A = \dot{q}$ (6), which contains the desired joint velocities for each joint of the robot for the next time step. The seventh joint of the robot only affects the orientation of the robot, but not its position. But in this setup, the orientation is excluded. Because of that, the agent's target position can be achieved regardless of its orientation, making the seventh joint irrelevant to the learning process. The design choice of using only six joints and limiting the workspace to 180° is made for the sake of comparability. By adopting a similar setup to [KHS+21], the results of the PopSAN agent can be effectively compared to their findings at the end of this work.

After the agent executed an action, a reward $r(\boldsymbol{s}, \boldsymbol{a})$ is given to the agent. The reward function designed for this experiment consists of three terms, shown in Equation 4.1.

$$r(\boldsymbol{s}, \boldsymbol{a}) = r_{dist} - r_{acel} + r_{penalty} \tag{4.1}$$

The first term r_{dist} encourages the agent to minimize its distance δx between the TCP and the target position. The term distinguishes three cases, depending on the current Euclidean distance $||\delta x||$, shown in Equation 4.2. The closer the TCP is to the target position the higher the reward, with an exponential increase of the reward as soon as the distance is less than 10 cm.

$$r_{dist} = \begin{cases} 0 & \text{if} \quad ||\delta x|| > 0.6\,\text{m} \\ -0.5 \cdot \delta x + 0.3 & \text{if} \quad 0.6\,\text{m} < ||\delta x|| \leq 0.1\,\text{m} \\ \exp(-520||\delta x||^2) + 0.245 & \text{if} \quad ||\delta x|| < 0.1\,\text{m} \end{cases} \tag{4.2}$$

The r_{acel} term in Equation 4.3 encourages the agent to minimize joint acceleration, as the highest acceleration value is subtracted from the reward. This promotes smooth movements of the robot. Moreover, the agent learns to approach the target position with small joint velocities and stay stationary at the target, as any unnecessary acceleration leads to a decrease in the reward.

$$r_{acel} = \max(|\ddot{q}|) \cdot 0.005 \tag{4.3}$$

And lastly, the agent is penalized with the $r_{penalty}$ term, defined in Equation 4.4. The agent will receive a negative reward of -1 if the height of the TCP is below or above a certain threshold. In order to prevent the robot from crashing into the table or something else. If the agent exceeds one of the joint limits of the robot, it will also receive a penalty.

$$r_{penalty} = \begin{cases} -1 & \text{if} \quad 0.7\,\text{m} < z_{tcp} < 0.2\,\text{m} \\ -1 & \text{if} \quad 0.2\,\text{m} > z_{tcp} \\ -1 & \text{if} \quad |q_i| > q_{thresh} \\ 0 & \text{else} \end{cases} \tag{4.4}$$

To enhance training effectiveness, curriculum learning is implemented into the simulation environment. This involves dividing the workspace, which is defined as a torus, into four regions, denoted as $R_1 \subset R_2 \subset R_3 \subset R_4$. Each region contains the previous one, as shown in Figure 4.2b. The centre of the torus workspace is defined at a distance of 55 cm from the robot base. Starting from this centre line, the radius and rotation angle of the torus are given in table Table 4.1 for the different regions. Thus, at R_4 the workspace begins at 40 cm and ends at 70 cm, and surrounds the robot by 180°.

Table 4.1: Defined parameters for the four different workspace regions. The center of the torus is defined at a distance of 55 cm from the robot base. The center line is drawn around the robot with the current active rotational angle. And from this center line, the torus is extended in both directions with the current radius.

	rotation-angle (°)	radius of torus (cm)
R_1	±22.5	3.75
R_2	±45	7.5
R_3	±67.5	11.25
R_4	±90	15

At the end of each episode, the Euclidean distance between the TCP and the target position is measured and stored. If the average error of the last ten episodes falls below the threshold of 2 cm the next region is unlocked. That way the workspace expands in relation to the learning progress made by the agent.

4.2.3 Training procedure

Effective training of a DRL agent requires seamless communication between the neural network and the simulation environment. On the one side, the agent relies on the current state of the environment, and on the other side the action of the neural network needs to be executed in the simulation environment. To overcome this communication gap, the simulation environment is registered as a custom *Gym* environment.

The *OpenAi Gym* framework is an open-source interface for reinforcement learning, provided by *OpenAi* [BCP+16]. With the gym environment serving as an interface seamless and standardized communication between the simulation environment and the agent can be achieved. Enabling the agent to carry out its actions and receive its current state, as illustrated in Figure 4.3.

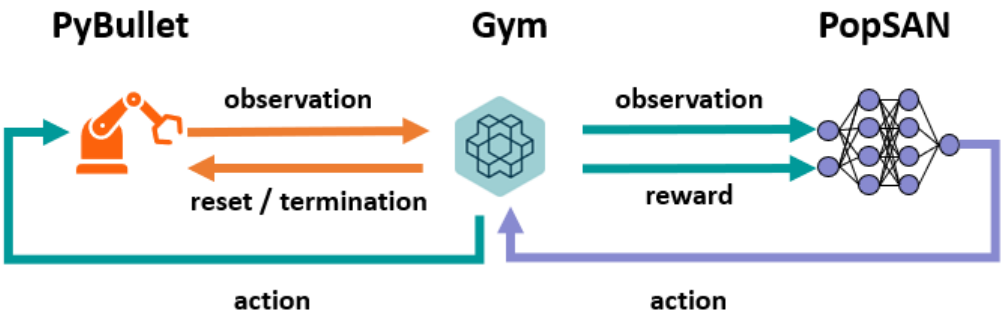


Figure 4.3: Communication between the simulation environment, the gym interface and the agent

Another challenge during training is hyperparameter optimization, which is crucial for the success of this work. While tools like *Hyperopt* [Ber] are commonly used for hyperparameter optimization, they often require numerous training runs to optimize each set of hyperparameters. Given that training the PopSAN agent takes days, a more promising approach is to utilize hand-picked hyperparameters based on experience, which can save significant computation time, although it may not yield optimal results. To tackle this challenge, the training process is organized into iterative cycles, wherein recorded hyperparameters, configurations, and training results are saved and analyzed. Metrics such as reward progress, trajectory smoothness, and agent precision are evaluated to make necessary adjustments to hyperparameters and other configurations. This iterative approach allows for fine-tuning and optimization of the training process, ultimately enhancing the performance of the trained agent.

4.2.4 Hardware specifications

In this section, an overview is given of the hardware utilized during the training of the DRL agents for this master thesis. The training is conducted on a specialized workstation equipped with the following components:

- CPU: Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30 GHz (48 cores),
- GPU: Nvidia Quadro RTX 8000 (48 GB) and
- RAM: 62 GB

The Nvidia Quadro RTX 8000 graphic card offered accelerated performance specifically tailored for deep learning tasks, contributing to faster training times and improved model convergence. But given the relatively small size of the neural networks employed, a GPU with less than 2 GB of memory would have been sufficient for training. The actual bottleneck that influenced the training speed was the number of available CPU cores and the RAM. Because the training time can be drastically reduced by running multiple simulation environments in parallel. But running these multiple simulation environments in parallel requires a significant amount of RAM and CPU power. Therefore, the availability of such a large amount of resources leads to a great acceleration of the training process.

4.2.5 Sim2Real transfer

At the end of this work, the performance of the trained agents should also be evaluated on a real robot. Thus a new communication framework becomes crucial as the agent needs to interact directly with the physical system and not with the simulation environment anymore. In this thesis, the trained agent is applied to the Franka Research 3 robot. To enable the communication between the agent and the robot the Robot Operating System (ROS) [Sta] is used.

ROS is an open-source framework widely used in robotics for developing and controlling robotic systems. It provides a flexible and modular framework for inter-process communication. One of the key concepts in ROS is the publisher-subscriber model, which enables different components of a robot system to communicate with each other. Publishers are responsible for broadcasting messages, containing data or commands, to a specific topic in the ROS ecosystem. Subscribers listen to those topics and receive the messages published by publishers, allowing them to react or process the

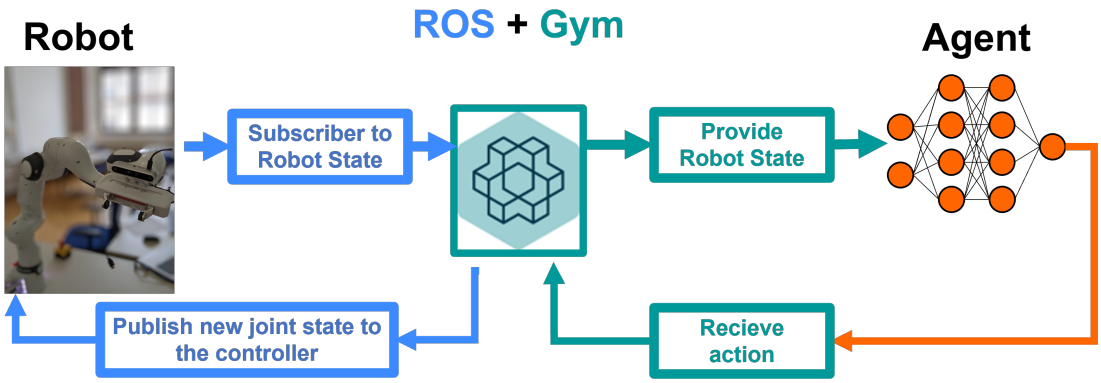


Figure 4.4: Communication between the real robot, the gym interface and the agent

information accordingly. Combining ROS with the Gym framework enables seamless communication between the real robot and the neural network.

Figure 4.4 shows the setup used in this work. The current state of the robot is published to the topic *joint-states*. This information is subscribed by a subscriber in the Gym environment, which receives the robot's current joint states and provides them as input to the agent. The agent then generates joint velocities, which are published to the *joint-states-desired* topic. Lastly, the robot's controller subscribes to this topic, adopting the newly obtained desired joint values as the target values.

For transferring the agents from simulation onto the real robot a few improvements are made to enhance the control process. At first, the conventional approach of directly setting joint velocities is replaced. Instead, the joint velocities are utilized to calculate new joint angles, according to:

$$q_{t+1} = q_t + \dot{q} \cdot \tau \tag{4.5}$$

with a time constant of $\tau = 0.05$, the computed joint velocities $\dot{q}$ and the new joint positions q_{t+1}. Kumar et al. [KHS+21] suggested that this implementation is more robust against noise and generates more stable results. Further, a moving average is applied to the generated joint velocities. Thus, the output joint velocities from the agents are not directly applied to the robot, but are first smoothed. At last, the joint velocities are input to a PID controller, before being applied to the robot. Hence, this setup is almost identical to the setup of [KHS+21] illustrated in Figure 3.1.

5 Experiments and Results

This chapter focuses on the experiments and presents their results. To fully comprehend the experimental context, it is essential to develop a clear understanding of the research objective. The research objective is to investigate, if it is possible to achieve state-of-the-art performance for training a DRL agent with a Population-coded Spiking Actor Network (PopSAN) on the task of controlling a seven-DoF robot. Therefore, a PopSAN agent and a standard DNN agent are trained for twenty million time steps on the task of reaching a random target position with the TCP, by controlling the joint velocities of the robot. The DNN agent is trained as a reference point, so that the results of the PopSAN agent can be compared with the current state-of-the-art.

In the first section of this chapter a set of criteria is defined to evaluate and assess the performance of the agents. The second section describes the experiments, which are used to evaluate the defined criteria. And lastly, the results of the experiments are presented.

5.1 Defining performance criteria

First of all, it is important to define a set of criteria by which the performance of the agents can be determined. Controlling a robot is a complex task and a controller needs to fulfill different criteria, in order to be considered as a useful controller. For this purpose, seven criteria are defined below, on the basis of which the agent's performance is assessed. The criteria are arranged in order of importance, starting with the most important criterion for a controller.

Accuracy and success rate

The main two criteria by which a controller can be judged are its accuracy and its success rate. Both criteria are related to one another and are therefore discussed together. In this work, the accuracy and the success rate are defined as follows. The accuracy is defined as the Euclidean distance between the current TCP position and the target

TCP position, at the end of an episode. The second criterion is the success rate. An episode is considered successful if the Euclidean distance at the end of the episode is less than two centimeters. The higher the accuracy and the higher the success rate, the better the performance of the agent.

Trajectory quality

The third criterion is the generated trajectory. The trajectory should fulfill two requirements. First, an efficient controller should keep the movement effort as low as possible and move along the fastest path to the target in the Cartesian space. Secondly, the trajectory should be smooth, characterized by the absence of abrupt changes in velocity or acceleration. This indicates that the agent maintains a steady and controlled motion throughout its path.

These two requirements can be validated as follows. In order to check if the agent takes an efficient path towards the goal, the course of the Euclidean distance between the target TCP position and the current TCP position, can be recorded for multiple runs, normalized and visualized. If the graph shows a continuous curve that approaches the target without detours, it can be assumed that the generated trajectories are efficient. Furthermore, the course of the generated joint angles and velocities can be plotted. They also should be smooth and without big jumps. In order to evaluate the smoothness of a trajectory in qualitative terms, the total variation is computed, according to Equation A.1.

Robustness against noise

The fourth important criterion which is taken into account is the robustness of the controller against noise. Because real-world environments are often characterized by various sources of noise, including sensor noise, environmental variations, or uncertainties in the system dynamics. A robust controller can handle these noise sources and still generate reliable control commands. Therefore, it is investigated how the performance of the agent changes when different levels of noise are applied to the sensory input data.

Behavior in extreme situations

The fifth criterion selected is the behavior of the agents in extreme situations. It is important that a controller is as reliable at the edges of the workspace as it is in the rest of it. Therefore, it is investigated whether the performance of the agent is affected when the target positions are at the edge of the workspace or slightly above it.

Inference speed

The sixth criterion is the inference speed of the network. That is the duration it takes for the neural network to process an input and produce an output. The more inferences per second can be achieved by the neural network the faster the controller can react to changing conditions. This is crucial for robot control. Because robots often operate in dynamic environments. Meaning the higher the inference speed the quicker the robot can respond to changing conditions, avoiding collisions or completing tasks efficiently.

Transferability

The seven and last criterion considered is the transferability of the agents. Here it is studied how the achieved performance of the agents in simulation transfers to a robot in the real world. The better the results of an agent transfer to the real world, the less the agents need to be refined, making the agent more practical and cost-effective.

Summary

With these seven criteria, the performance of the agents can be effectively evaluated. Thereby the success rate and accuracy are the most important criteria, followed by the quality of generated trajectories, the robustness against noise, and the behavior in extreme situations. Any deficiency in one of those criteria necessitates changes in the configuration, retraining, or both. The inference speed is also important, but as long as the network can reach a control rate between 100 Hz and 500 Hz it should be sufficient for achieving real time control. In the context of this work, the least important criterion is transferability, because of its relatively minor influence on the overall performance of a controller. The performance in the real world can mostly be improved with some additional effort.

5.2 Experiments

Now that a set of criteria has been established, it is time to design different experiments to evaluate these criteria. The majority of the criteria are evaluated in simulation. Because it is safe and time efficient. The only criterion that needs to be evaluated in the real world, is the transferability of the agents. To evaluate the defined criteria four different scenarios are constructed, which are introduced in the following sections.

5.2.1 Basic scenario definition

The first scenario is named the basic scenario. This scenario is identical to the training scenario described in section 4.2.2. For every new episode the agent is initialized in a random position inside its workspace without obstacles. Then the agent has to move the TCP to a random target position, by controlling the joint velocities. This setup is illustrated in Figure 5.2. The workspace of the robot is defined as a partial torus with 180° around the robot and a diameter of 30 cm, illustrated in Figure 4.2a.

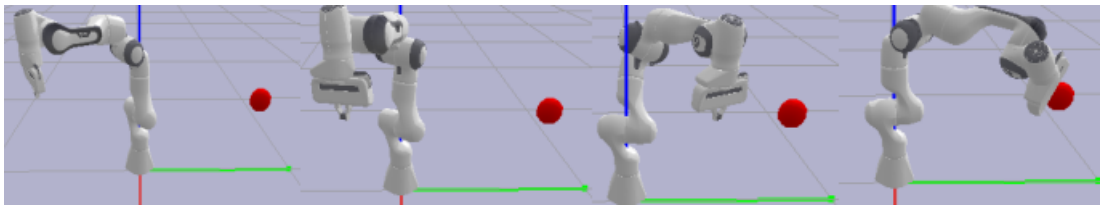


Figure 5.1: Example of the executed basic scenario. The robot starts in a random position and has to reach a random generated target position (red sphere).

This scenario serves as the baseline, where the accuracy, the success rate, the quality of the generated trajectories, and the inference speed are evaluated. Therefore, each agent is run five times for 100 episodes to obtain a statistically significant sample size of data. From this, the average accuracy, success, and inference speed are calculated.

5.2.2 Noise scenario definition

The second scenario is called the noise scenario. This scenario, is almost identical to the basic scenario, with one exception. In this scenario three different levels of Gaussian noise:

- light: $\sigma = 0.01$,
- medium: $\sigma = 0.025$
- and heavy: $\sigma = 0.05$.

are added to the current joint velocities and joint angles of the agent's observation, in order to evaluate the agent's robustness against noise. σ is the standard deviation. In this scenario only one criterion is evaluated, the robustness against noise. For each level of noise, each agent is run five times for 100 episodes. To assess the robustness against noise, the average accuracy, success rate, and generated trajectory are used as metrics and they are compared to the achieved results in the basic scenario, to understand how the performance of the agents is affected by the noise.

5.2.3 Extreme scenario definition

In the third scenario, called the extreme scenario, the target positions are always generated at the edges of the workspace, or slightly above. This allows investigating how the performance of the agents is affected by extreme or novel situations. During training the workspace of the robot is limited to a maximum and minimum Euclidean distance between the TCP and the robot base. In training the agent is punished if it is exceeding a maximum distance of 70 cm or if it falls below a minimum distance of 40 cm. Therefore, in this scenario, the target position are generated in an area of 75 cm to 65 cm and 45 cm to 35 cm. Thus, the agents are forced to operate at the edges of the workspace. Both agents are run again five times for 100 episodes, to obtain a statistically significant sample size. To assess the behavior of the agents in extreme situations, the average accuracy, success rate, and generated trajectory are used as metrics and they are compared to the basic scenario to understand how the performance of the agents is affected by operating in extreme situations.

5.2.4 Real world scenario definition

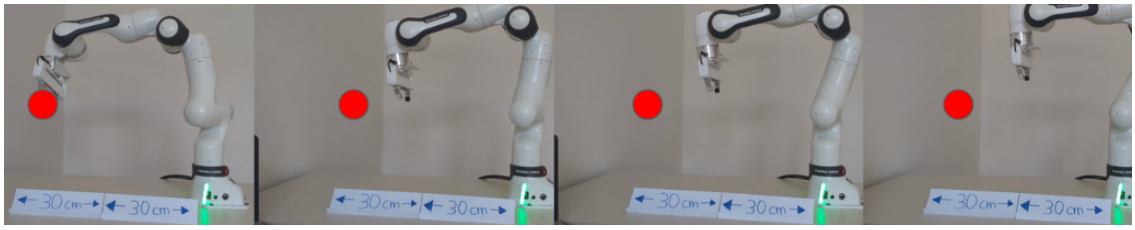


Figure 5.2: Example of the executed real world scenario. Just as in the basic scenario the real robot starts in a random position and has to reach a random generated target position (red sphere).

In the fourth and final experiment, the agents are transferred onto a real robot, the *Franka Research 3*. The setup of this experiment is almost identical to the basic scenario, with the exception that this scenario is run in the real world. But since running the agents on a real robot takes a lot longer, the agents are only run five times for 25 episodes, instead of 100 episodes. By comparing the achieved accuracy, success rate, and generated trajectories to the results achieved in the basic scenario, a statement can be made about how well the in simulation learned capabilities can be transferred to the real world, without any additional retraining or fine tuning.

5.3 Results

This section presents the results of the various experiments described in the previous section. Firstly, the acquired rewards during training are presented, followed by the outcomes of the basic, noise, and extreme experiments. Finally, the section concludes with the results of the transfer from simulation to the real world.

5.3.1 Training

Before executing new experiments, it can be helpful to analyze the achieved reward during the training phase of both agents. The analysis of the achieved reward can serve as an initial indicator of the agent's potential performance and give interesting insights into the agents learning dynamics and convergence behavior. That way, this analysis sets the foundation for interpreting the results achieved later on.

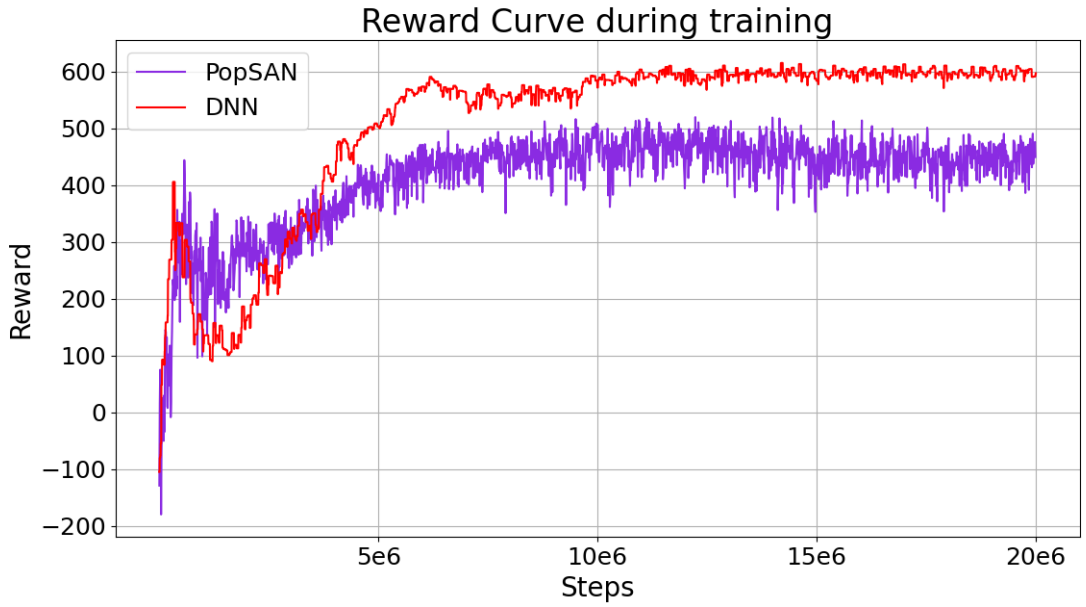


Figure 5.3: Course of reward during training of the PopSAN agent (violet) and the DNN agent (red).

Figure 5.3 shows the course of the reward obtained during training. At first, it can be seen that the DNN converges to a reward value of around 600. The PopSAN agent converges to a lower reward value at around 500. Furthermore, it can be observed that the variance of the PopSAN agent during training is significantly higher than the

variance of the DNN agent. And lastly, it can be observed that neither of these agents converges significantly earlier or later than the other.

5.3.2 Basic scenario

At first the results of the basic scenario, which is described in section 5.2.1, are presented. Starting with the accuracy (ac) and the success rate (sr). Table 5.1 shows the results of the individual runs and the average accuracy and success rate of all five runs together.

Accuracy and success rate

Table 5.1: Achieved accuracy (ac) in cm and the achieved success rate (sr) in % of the PopSAN agent and the DNN agent in the basic scenario.

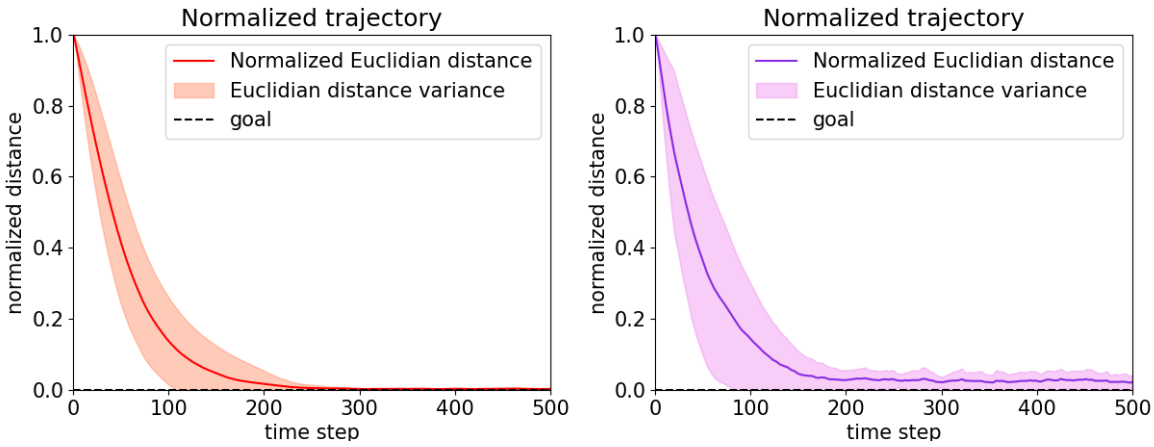
	run 1		run 2		run 3		run 4		run 5		average	
	ac	sr	ac	sr	ac	sr	ac	sr	ac	sr	ac	sr
PopSAN	1.39	90	1.32	91	1.30	91	1.37	93	1.30	90	1.34	91
DNN	0.48	98	0.59	99	0.56	99	0.48	98	0.57	100	0.58	99

Table 5.1 shows that the the DNN agent achieves an average success rate of 99 % and an average accuracy of 0.58 cm over those five runs. In comparison to that, the PopSAN agent achieves a lower success rate of 91 %, with an average accuracy of 1.34 cm. Therefore, it can be stated that the PopSAN agent fails to match the accuracy and the success rate of the DNN agent.

Trajectory quality

The next criterion assessed is the quality of the trajectory generated by the agents, defined in section 5.1. At first, it is examined if the agents take a direct path towards the goal. This is done by plotting the average normalized Euclidean distance of 100 episodes and the standard deviation. Based on Figure 5.4 it can be observed, that both agents take a direct path towards the goal in the Cartesian space. Further, it is visible that the DNN achieves a higher accuracy and a smaller standard deviation than the PopSAN agent.

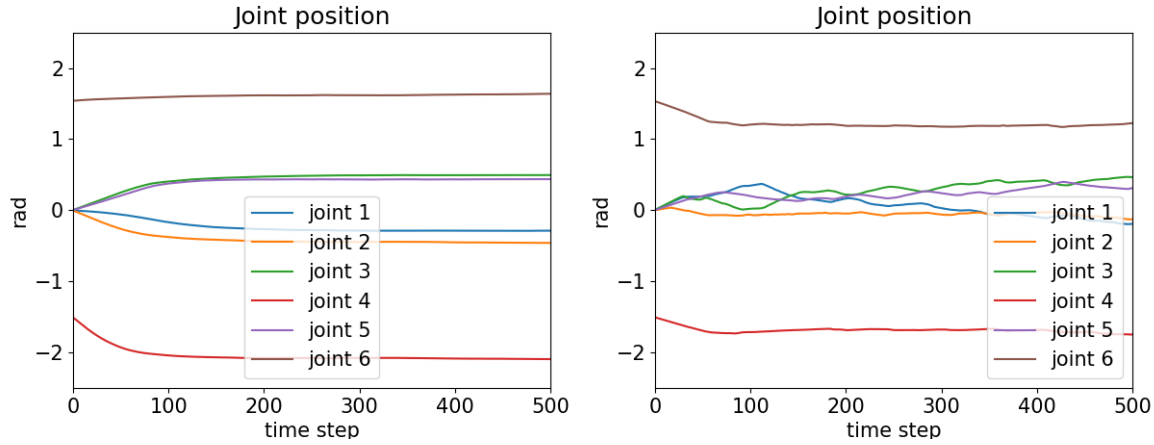
Next, the course of the joint angles is investigated. Therefore, 5.5 shows the generated joint angles by the agents for approaching the same target position, starting from the same initial position. 5.5 demonstrates visible differences in the generated joint angles. The joint angles of the DNN agent are smooth and continuous. For the PopSAN agent only three generated joint angles are smooth and continuous. The course of the joint



(a) (b)

Figure 5.4: Normalized Euclidean distance of 100 episodes for both agents, where the transparent area represents the standard deviation. (a) Result of the DNN agent, (b) Result of the PopSAN agent.

angles one, three, and five are jagged. This visual expression can be quantified by calculating the average total variation of 100 trajectories (Equation A.1). The average total variation for the joint angles of the DNN agent is 0.53 rad and the total variation of the joint angles generated by the thePopSAN is 0.99 rad.

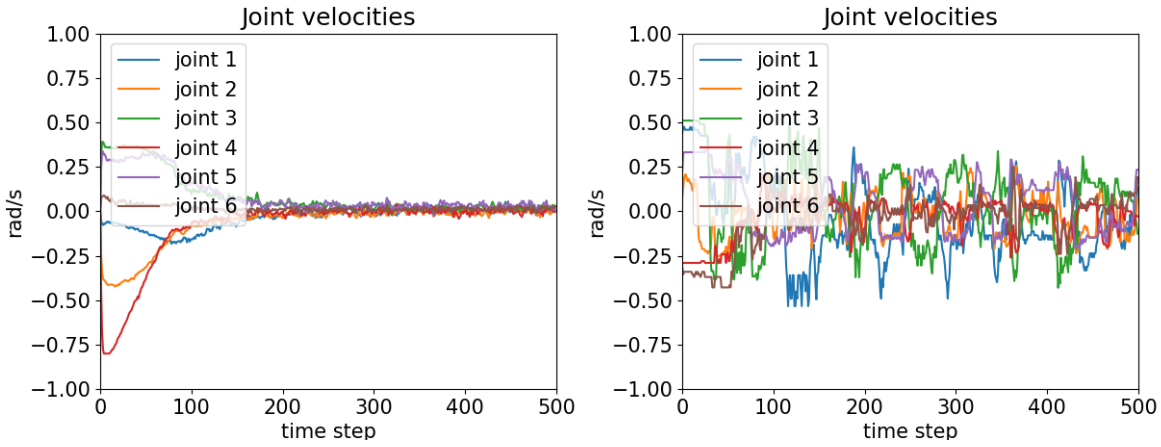


(a) (b)

Figure 5.5: Course of joint angles generated by (a) the DNN agent and (b) by the PopSAN agent. Starting and ending at identical positions.

At last, the generated joint velocities of the agents are compared in Figure 5.6. The generated joint velocities differ visibly. The velocities generated by the DNN agent are relatively smooth and continuous. In contrast to that, the joint velocities of the

PopSAN agent are very irregular and bumpy. This can be quantified by obtaining the average total variation of the generated joint velocities. Therefore, the total variation of 100 episodes is computed for the joint angles and joint velocities. Based on this the average total variation is obtained. The average total variation of the joint velocities generated by the DNN agent is $4.57\,\mathrm{rad\,s^{-1}}$. For the joint velocities of the PopSAN agent, the average total variation is $18.51\,\mathrm{rad\,s^{-1}}$.



(a) (b)

Figure 5.6: Course of joint velocities generated by (a) the DNN agent and (b) the PopSAN agent. Starting and ending at identical positions.

Inference speed

The third criterion evaluated is the inference time of the agent, described in section 5.1. In a single episode, the neural network receives a new observation as input for each of the 500 time steps. Consequently, a new inference time can be measured at each time step. At the end of a single episode, a total of 500 inference times are obtained. With each run comprising 100 episodes, this leads to a collection of 50,000 inference times in total, by the end of a single run. By calculating the average of these 50.000 inferences times the inference time for a single run is obtained. Based on this average inference time, it is then possible to determine the number of inferences per second (Inf/s) that the neural network is capable of performing.

The results in Table 5.2 show that the PopSAN has a higher inference speed than the DNN. In average the PopSAN agent achieves in average 536.40 Inf/s, while the DNN only achieves 380.96 Inf/s in average. Over those five runs the PopSAN has a standard deviation of 3.26 Inf/s. The DNN agent has a standard deviation of 12.55 Inf/s.

Table 5.2: Comparison of the inference speed of both agents achieved during the evaluation

	Inf/s					
	run 1	run 2	run 3	run 4	run 5	average
PopSAN	537.53	538.33	529.94	537.46	538.76	536.40
DNN	385.49	366.29	367.03	398.86	387.15	380.96

5.3.3 Noise scenario

In this section, the results of the noise scenario are presented, which is defined in section 5.2.2. In this scenario, the criterion of robustness against noise is evaluated. Therefore, the metrics of accuracy, success rate, and trajectory are assessed and compared with the results from the basic scenario.

Accuracy and success rate

The main criteria for assessing performance degradation due to noise are the accuracy (ac) and the success rate (sr). For reasons of space, the results of the individual runs are summarised and only the average accuracies and success rates achieved under the individual noise levels are presented in table Table 5.3. The results of the individual runs are shown in the appendix in Table A.2, Table A.3, and Table A.4.

Table 5.3: Average accuracy (ac) and success rate (sr) of the agents, under the influence of light, medium, and heavy Gaussian noise in simulation.

	light noise		medium noise		heavy noise	
	ac	sr	ac	sr	ac	sr
PopSAN	1.04	88.6	1.24	68.8	1.33	37
DNN	0.39	100	0.79	99.2	1.25	76.6

For light noise, the PopSAN agent achieves an average accuracy of 1.04 cm and an average success rate of 88.6 %. The average accuracy improved a little bit and the average success rate decreased slightly. Overall these results are similar to the results obtained by the PopSAN agent in the basic scenario, see Table 5.1. For the DNN agent, the average accuracy improved slightly from 0.58 cm to 0.39 cm and the average success rate increased by one percent to 100 %. Thus, the accuracy and success rate of the DNN agent also remains stable.

For medium noise, the performance of the PopSAN agent drops significantly compared to the basic scenario from an average success rate of 88.6 % down to 68.8 % and the accuracy decreases from 1.04 cm down to 1.24 cm. In contrast to that, the average

success rate of the DNN agent remains as before at around 99 %. Only the average accuracy decreases notably from 0.39 cm to 0.79 cm.

In the final iteration, heavy Gaussian noise is applied to the agent's sensor input. The results in Table A.4 demonstrate that the performance of both agents is strongly affected by the heavy noise. The success rate of the DNN agent decreases from almost 100 % to 76 %. And the success rate of the PopSAN agent drops even further to 37 %.

Trajectory quality

Noise can also affect the quality of the robot's trajectory, by disrupting the agent from the optimal path, leading to suboptimal trajectories or unstable behavior. Therefore, Figure 5.7 illustrates the normalized Euclidean distance of 100 episodes for both agents under the influence of all three levels of noise and Table 5.4 shows the average total variation of 100 trajectories for the course of the joint angles and joint velocities under the different strengths of noise. Exemplary joint angles and joint velocities under the different levels of noise are illustrated the appendix in section A.3.1.

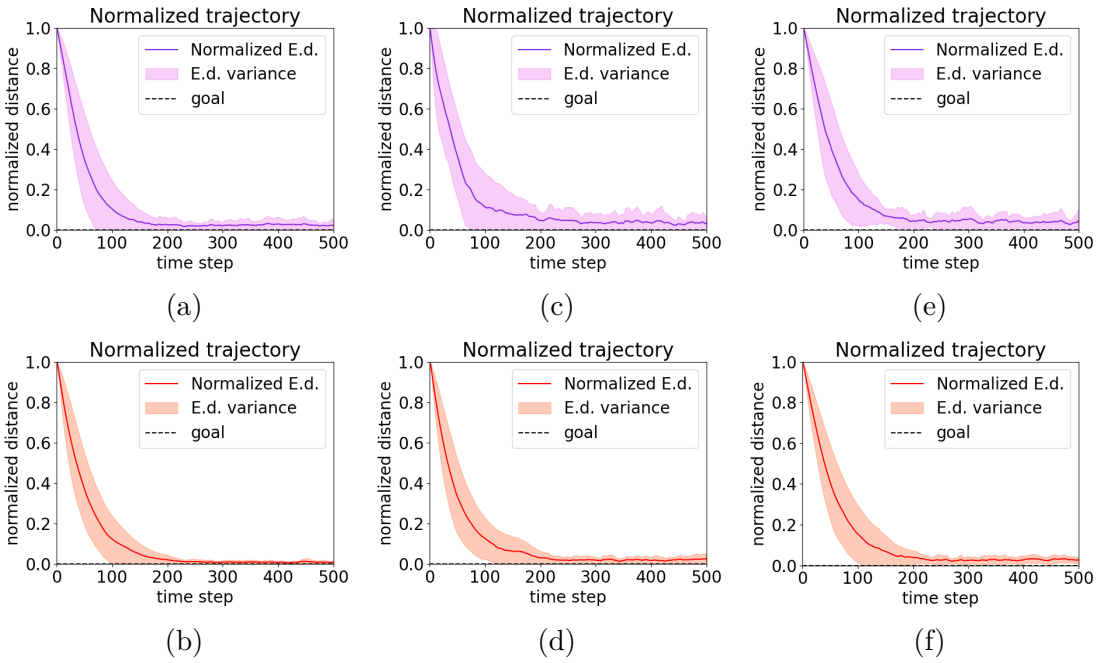


Figure 5.7: Normalized Euclidean distance (E.d.) under the influence of noise, where the transparent areas represent the standard deviation. The upper images show the normalized Euclidean distance for the PopSAN agent under the influence of (a) light noise, (c) medium noise, and (e) heavy noise. The lower images show the normalized Euclidean distance for the DNN agent under the influence of (b) light noise, (d) medium noise, and (f) heavy noise.

By comparing the normalized Euclidean distance under the influence of light noise to the obtained Euclidean distance in the basic scenario, no visible difference stands out. Only an increase in the average total variation for the joint velocities can be noted. The total variation for joint velocities of the PopSAN agent increases from $18.51\,\mathrm{rad\,s^{-1}}$ to $24.61\,\mathrm{rad\,s^{-1}}$ and the total variation of the joint velocities of the DNN agent increases from $4.57\,\mathrm{rad\,s^{-1}}$ to $9.39\,\mathrm{rad\,s^{-1}}$.

Table 5.4: Calculated average total variation for the different noise levels of the joint angles (q) and joint velocities ($\dot{q}$)

	Average Total Variation for					
	light noise		medium noise		heavy noise	
	q	$\dot{q}$	q	$\dot{q}$	q	$\dot{q}$
PopSAN	0.99	24.61	1.03	40.94	1.13	61.59
DNN	0.55	9.39	0.75	19.63	0.99	35.25

For medium noise, the normalized Euclidean distance of both agents shows greater variance and a more disturbed course, which indicates less effective trajectories. It is also visible that the trajectories of the PopSAN agent are more affected than the trajectories of the DNN agent. This is supported by the average total variation. The total variation of the DNN agent for joint velocities increases from $9.39\,\mathrm{rad\,s^{-1}}$ to $19.63\,\mathrm{rad\,s^{-1}}$. In contrast, the total variation of the PopSAN agent for joint velocities increases from $24.61\,\mathrm{rad\,s^{-1}}$ to $40.94\,\mathrm{rad\,s^{-1}}$.

For heavy noise, a further increase in variance and disturbance can be noticed in Figure 5.7, for both agents. Again this is quantified by the increase in the average total variation. The total variation of the DNN agent for the joint velocities increases by $15\,\mathrm{rad\,s^{-1}}$ and the total variation of the PopSAN agent for the joint velocities increases by $20\,\mathrm{rad\,s^{-1}}$.

5.3.4 Extreme scenario

The third scenario examines the agent's ability to operate at the edges of the workspace. Therefore, the target positions are intentionally generated only at the edges of the workspace or slightly above. This allows for investigating how the performance of the agents is affected by extreme or novel situations. The setup of the experiment is described in section 5.2.3. The first metrics used to detect a change in performance are the success rate (sr) and the accuracy (ac) of the agents see Table 5.5.

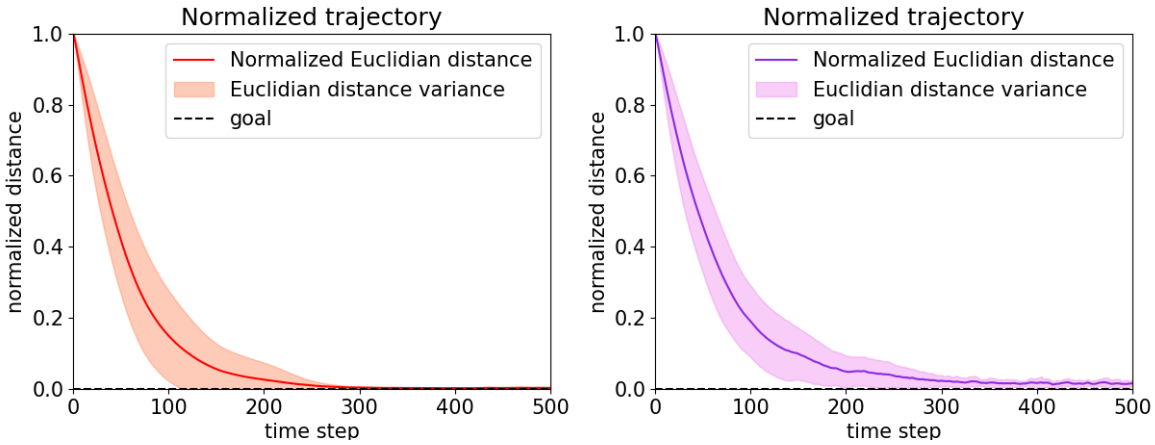
Table 5.5: Achieved accuracy (ac) and success rate (sr) at the edges of the workspace.

	run 1		run 2		run 3		run 4		run 5		average	
	ac	sr	ac	sr	ac	sr	ac	sr	ac	sr	ac	sr
PopSAN	1.53	68	1.58	59	1.46	63	1.53	61	1.53	77	1.53	68.8
DNN	0.88	95	0.89	98	0.93	97	0.93	99	0.91	94	0.91	95.4

Accuracy and success rate

For the DNN agent, the average success rate drops from 99 % to 95 % and the average accuracy increases from 0.58 cm to 0.91 cm, compared to the performance in the basic scenario. For the PopSAN agent, the average success rate decreases significantly from 91 % down to 68.8 % and the average accuracy reduces as well from 1.34 cm to 1.53 cm. This is a drastic decline in performance, compared to the basic scenario. Further, the quality of the trajectories is examined.

Trajectory quality



(a) (b)

Figure 5.8: Normalized euclidean distance of 100 episodes for both agents in extreme situations, where the transparent area represents the standard deviation. (a) Result of the DNN agent, (b) Result of the SNN agent.

The generated trajectories in Figure 5.8 do not differ visibly from the generated trajectories in the basic scenario, illustrated in Figure 5.4. This is also true for the generated joint angles and joint velocities, see Figure A.7. The average total variation for the joint velocities and joint angles also remains stable for both agents. For the DNN agent, the joint angles have an average total variation of 0.56 rad and for the joint velocities the total variation is 4.73 rad s^{-1}. For the PopSAN agent, the joint angles have an average total variation of 0.96 rad and for the joint velocities total variation is 18.32 rad s^{-1}.

5.3.5 Real world scenario

The last criterion evaluated is the transferability of the agents from the simulation onto a real robot. This scenario is described in section 5.2.4. Therefore, the accuracy, success rate, and the quality of trajectories are used as metrics to investigate if the same performance can be achieved as in the simulation. At first, the accuracy and success rate obtained in the real world are presented and compared to the results achieved in the basic scenario.

Accuracy and success rate

Table 5.6: Achieved accuracy (ac) and success rate (sr) of the PopSAN agent and the DNN agent on the real robot

	run 1		run 2		run 3		run 4		run 5		average	
	ac	sr	ac	sr	ac	sr	ac	sr	ac	sr	ac	sr
PopSAN	1.34	68	1.15	73	1.38	66	1.24	75	1.29	72	1.28	70.8
DNN	0.58	100	0.74	99	0.69	100	0.83	99	0.62	98	0.69	99.2

The results in Table 5.6 demonstrate differences in the accuracy and success rate of the agents. The DNN agent achieves an average accuracy of 0.69 cm and an average success rate of 99.2 %. This corresponds approximately to the performance achieved in the basic scenario in simulation, see Table 5.1. In contrast to that the PopSAN agent achieves an average accuracy of 1.28 cm and an average success rate of 70.8 %. This is a significant decrease in the success rate of the agent, compared to the success rate of 91 % achieved in the basic scenario.

Trajectory quality

Additionally to the accuracy of the agents, it is important to analyze the trajectories of the agents. The agents must demonstrate the capability to transfer smooth and effective trajectories from the simulation environment onto the real robot. Figure 5.9 illustrates the normalized Euclidean distance over 25 episodes.

Figure 5.9 shows that the trajectories of the DNN agent are smooth and effective, just as in simulation. The trajectories of the PopSAN agent show a much greater variance and small oscillations over the complete trajectory. Demonstrating that the trajectories of the PopSAN agent are affected by the changing conditions in the real world, while the trajectory of the DNN remains unaffected.

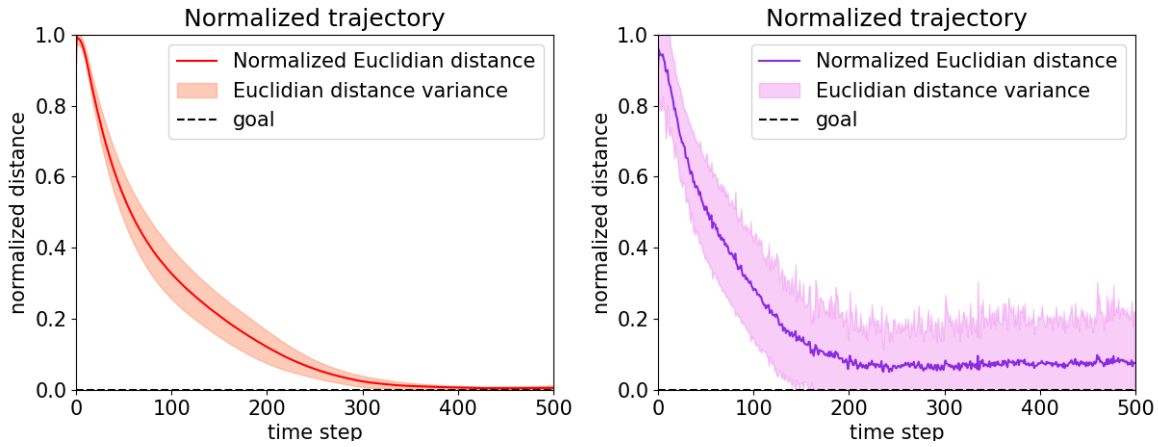


Figure 5.9: Normalized euclidean distance of 25 episodes on the real robot for each agent, where the transparent area represents the standard deviation. (a) Result of the DNN agent, (b) result of the SNN agent.

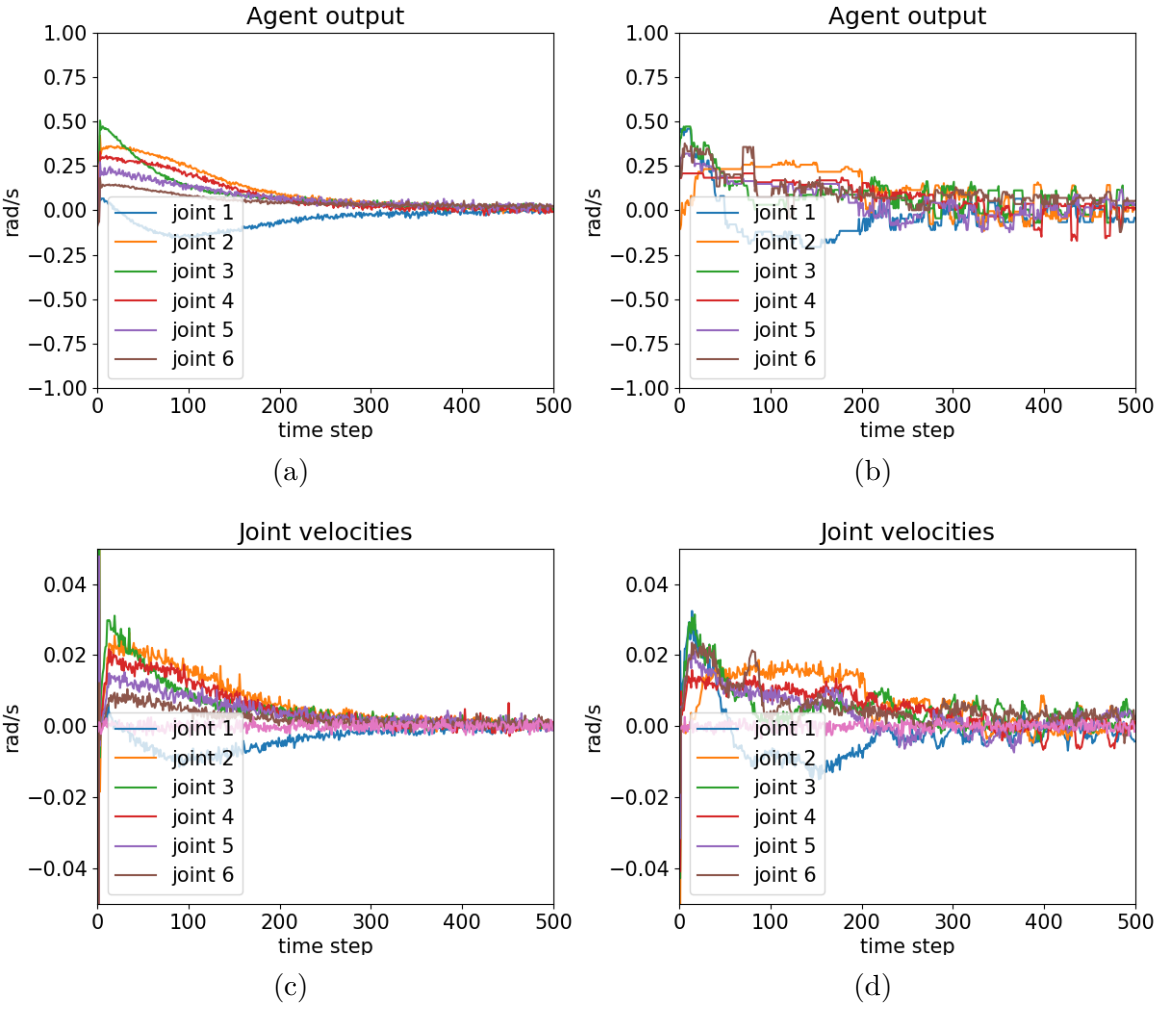


Figure 5.10: The upper figures show the joint velocities for the real robot generated by (a) the DNN agent and (b) the PopSAN agent. Starting and ending at identical positions. The figures below show the smoothed joint velocities, which are applied to the robot for (c) the DNN agent and (d) the PopSAN agent.

This statement is confirmed by analyzing the joint velocities of the agents. Figure 5.10 shows two different joint velocities for both agents. The upper images are the joint velocities directly generated by the agents, while the lower images show the joint velocities, which are actually applied to the real robot. Because the outputted joint velocities of the agents are smoothed by a moving average and a PID controller before they are applied to the robot. For more details about the implementation see section 4.2.5.

The generated joint velocities of the DNN agent remain smooth as they are in simulation, see Figure 5.6 for comparison. The moving average and PID controller, which are applied to the input, only decrease the joint velocities for safety reasons. For the PopSAN agent it shows, that the jerky joint velocities from simulation, see Figure 5.6, are also transferred onto the real robot. Therefore, the PopSAN agent requires the moving average and PID controller in order to smooth the generated joint velocities. The average total variation remains about the same, compared to the basic scenario, except for the joint velocities of the PopSAN agent. The average total variation for the joint velocities is $7.07\,\mathrm{rad\,s^{-1}}$, which is significantly lower, compared to the basic scenario.

6 Discussion

This chapter consists of two parts. The first part discusses the findings of the previous chapter, summarises the discussion, and draws a final conclusion from the discussion in relation to the research question. In the second part, the obtained results are compared and discussed in the context of other research.

6.1 Discussion of Results

This section discusses the experimental results presented in chapter 5. The aim of this section is to explain the findings and draw conclusions from these findings in relation to the research objective. The research objective of this thesis is to investigate if it is possible to achieve state-of-the-art performance for controlling a robot with 7-DoF using DRL with a PopSAN. Therefore, a PopSAN agent and a DNN agent are trained with DRL and their performance is compared according to seven criteria: accuracy, success rate, robustness against noise, robustness in extreme/new situations, generated trajectory, inferences per second, and transferability. In order to answer the formulated research question, each criterion will be assessed separately and the results will be summarized at the end, where a final conclusion will be drawn.

6.1.1 Accuracy and success rate

The accuracy and the success rate are the most important metrics for a controller and the success rate is defined as a part of the accuracy. Therefore, both criteria are discussed together. The accuracy and the success rate of the agents were evaluated in the basic scenario, in Table 5.1. In the other scenarios, the accuracy and success rate were only used as a metric to evaluate changes in performance. For this reason, this discussion refers exclusively to the results achieved in the basic scenario. The results obtained in the basic scenario confirmed what had already been suggested by the performance during the training. The PopSAN agent achieves an average success rate

of 91 % with an accuracy of 1.34 cm. Hence, it can be stated that the PopSAN agent successfully learned this task. But the DNN agent achieved an average success rate of 99 % and an accuracy of 0.54 cm. Therefore it can be concluded that the PopSAN agent is not able to match the accuracy and success rate of the DNN agent in this scenario and therefore is not able to reach state-of-the-art performance.

The inferior accuracy and success rate of the PopSAN agent may be attributed to the selected hyperparameters, which might not be optimal. In general SNNs are more sensitive to hyperparameter tuning due to their unique temporal processing nature. The timing of spikes plays a crucial role in information representation and processing within SNNs. As a result, setting hyperparameters such as spike thresholds, encoding rates and many more significantly impacts the network's performance. Therefore, SNNs require meticulous tuning for optimal functionality. But, since a complete training run for 20 million time steps takes about five days, it was not possible to conduct an extensive hyperparameter search. Consequently, it is very likely that better hyperparameters exist, which will increase the performance. However, there are also other possible reasons for the inferior performance such as the inherent stochasticity or the training algorithm, which are discussed in detail in the following sections.

6.1.2 Trajectory quality

The third criterion discussed is the quality of the generated trajectories. The agent should always move very directly towards the target in the Cartesian space, without any unnecessary movements and it should generate smooth trajectories without large jumps in velocity. The results in the basic scenario, in Figure 5.3, show that both agents manage to reach the goal relatively efficiently. However, two things stand out. First, in Figure 5.4 it can be seen that the normalized Euclidean distance of the PopSAN has a larger variance than that of the DNN agent. Second, the course of the joint angles and joint velocities are visible more ragged than the generated trajectories of the DNN agent. This is also confirmed by the total variation, where the joint velocities of the PopSAN agent have a total variation of $18.51\,\mathrm{rad\,s^{-1}}$ and the joint velocities of the DNN agent have a total variation of $4.57\,\mathrm{rad\,s^{-1}}$. These results demonstrate that the generated trajectories of the DNN agent are more effective and smoother, compared to the PopSAN agent.

A possible reason for this might be the inherent stochastic nature of SNNs. The discrete and probabilistic nature of spikes introduces randomness into the network's behavior. Because the spikes are generated based on the timing of events rather than continuous

activation values and the timing of the events can be different every time. This results in inconsistent responses to the same input across different trials. This variability can hinder the network's ability to converge to stable solutions, leading to less smooth and effective trajectories. Furthermore, the LIF neuron model, introduces a specific characteristic known as quantization. The LIF neuron accumulates incoming signals over time and releases an output spike once a predetermined threshold is reached, which leads to a step-like change in output. In contrast to the continuous and smoothly varying activations in conventional neural networks. This inherent quantization can lead to abrupt changes and disrupt the continuity of trajectory paths.

6.1.3 Robustness against noise

When it comes to robustness against noise the analyzed results of the noise scenario (section 5.3.3), prove that the DNN agent is more robust against noise than the PopSAN agent, see Table 5.3. But it can be stated that both agents demonstrated robustness against light noise since the average success rate obtained under light Gaussian noise is similar to the average success rate obtained in the basic scenario. This changes for medium Gaussian noise. While the success rate of the DNN agent remains steady, the success rate of the PopSAN agent drops from 88.6 % down to 68.8 %. This indicates that the DNN agent is more robust against noise than the PopSAN agent. Under the influence of heavy noise, the performance dropped even further, but this time the performance of the DNN agent was also affected significantly. Further, the results illustrate the influence of noise on the trajectories of the agents. This becomes particularly evident in the total variation calculated for the joint angles and joint velocities. Table 5.4 shows that with every noise level, the total variation of the trajectories increases. This proves that noise affects the success rate and generated trajectories of the agents directly, which is not surprising since the agents aren't used to noise. In summary, the results from the noise scenario prove that the performance of the PopSAN agent is more affected by the noise. Thus, it is stated that the DNN agent is more robust against noise than the PopSAN agent.

This result is surprising, since SNNs are generally considered to be more robust against noise ([AEWK23], [PHS+19b]). Because SNNs exhibit sparse activation patterns driven by events, enabling them to disregard random noise and prioritize significant signal spikes. Additionally, SNNs can leverage temporal signal averaging, potentially diminishing sensitivity to minor fluctuations by integrating noise over time. One reason for this surprising result might be sup-optimal hyperparameters. Another reason might be

the encoding and decoding process. Encoding and decoding mechanisms play a critical role in population coded SNNs in translating inputs into spike trains. Inadequate handling of noise characteristics during this process may lead to reduced robustness.

6.1.4 Robustness in extreme situations

The goal of the extreme scenario was to investigate, how the performance of the agents is affected if the target positions are at the edge of the workspace or slightly above. In this scenario the DNN agent achieved an average success rate of 95 %. Compared to the basic scenario this is a slight decline by 4 %. In contrast to that, the success rate of the PopSAN agent declines from 91 % to 68 %. However, the extreme situations had no influence on the generated trajectories, see Figure 5.8. Nevertheless, the results prove a superior performance of the DNN agent in extreme situations.

It's challenging to pinpoint a definitive explanation for this particular result within the existing literature. My personal suggestion, which would require further experiments, is that the PopSAN agent didn't generalize as well as the DNN agent. This might be attributed to different hyperparameters, e.g. a different minibatch-size, which could lead to a better generalization for the DNN agent. Another reason could be the different learning algorithms. The backpropagation algorithm is specifically designed for DNNs, while the STBP algorithm is not able to fully exploit the capabilities of SNNs.

6.1.5 Inferences per second

The next criterion evaluated is the inference speed of the networks. Here the PopSAN agent achieved in average 536 Inf/s. In comparison to that the DNN agent achieved in average 380 Inf/s. The *Franka Reasearch 3* robot runs with a control frequency of 1 kHz. Therefore, no agent is capable of utilizing the full potential. But it should be noted that the inference speed is heavily influenced by the hardware on which the agents are executed. Consequently, this value can vary with different hardware configurations, and for the PopSAN agent, faster inference speeds are possible, particularly when leveraging neuromorphic hardware. However for this experiment with that specific setup, it can be stated, that the inference speed of both agents is suited for standard control tasks and the PopSAN agent achieved more inferences per second than the DNN agent.

In SNNs, information is communicated through discrete spikes, which results in a more efficient computation process. Unlike DNNs, which operate on continuous activations and require computations at every step, SNNs only perform computations when there are spike events, effectively reducing the overall computation workload. This event-driven processing allows Spiking Neural Network (SNN) to handle sparse data more efficiently, leading to faster inference times in certain scenarios.

6.1.6 Transferability

In order to check the transferability of the agents, both agents were applied to a real robot and the results were compared to the results achieved in simulation, in the basic scenario. The DNN agent achieved an average success rate of 99.2 % on the real robot an an average accuracy of 0.69 cm. These results are very similar to the results obtained in simulation. This shows that the DNN agent is able to transfer its learned policy to the real world without additional training. The PopSAN agent achieved in the real world an average success rate of of 70.8 % and an average accuracy of 1.28 cm. This is a decrease by 21 % in the success rate and demonstrates that the PopSAN agent is not able to transfer its learned policy onto a real robot, without retraining.

The reason for that decrease in performance might be found in previous results. In the noise scenario PopSAN agent proved to be less robust against noise. When medium noise was applied to the PopSAN agent, its success rate also decreased to 68 %, which is similar to the success rate achieved in the real world. Therefore, it is concluded that the decrease in performance is attributed to the noise introduced by the real world.

6.1.7 Summary and Conclusion of Discussion

In the basic scenario, the DNN agent achieved an average success rate of 99 %, which is 8 % higher than the PopSAN agent. In addition, the DNN agent also achieved a higher accuracy, with 0.58 cm, compared to the PopSAN agent, which achieved an average accuracy of 1.34 cm. Furthermore, the DNN agent proved to generate effective and smooth trajectories. The generated trajectories of the PopSAN agent are also effective, but visibly more ragged, which is confirmed by the higher total variation of the trajectories. The DNN agent also proved to be more robust against noise. E.g., under the influence of medium noise the success rate of the PopSAN agent decreased by 20 %, while the success rate of the DNN agent remained steady. Furthermore, the DNN agent demonstrated its ability to operate reliably at the edge of the workspace,

achieving an average success rate of 95 % in this scenario. In contrast, the PopSAN agent struggled in the extreme scenario, as evidenced by the 23 % drop in the average success rate. For the criterion of the inference speed, the PopSAN achieved 536 Inf/s, which are 156 Inf/s more than the DNN agent. This is the only criterion where the PopSAN agent achieved superior performance. For the last criterion, transferability, the results proved that the average success rate of the PopSAN agent decreased by 21 % when the agent was transferred to the real world. In contrast the DNN agent matched its success rate from simulation.

Based on these results, it can be said that the DNN agent outperformed the PopSAN agent in all criteria, except for higher inferences speed. In terms of the research question, this means that the PopSAN agent is not able to match the state-of-the-art performance of the DNN agent for this scenario with the hyperparameters chosen in this work. Possible reasons for this inferior performance might include inadequate handling of noise during the encoding and decoding process, the discrepancy in learning algorithms, or the quantization effect introduced by the LIF neuron model. However, this does not mean that the PopSAN architecture cannot achieve state-of-the-art performance for this task in general. Other authors have demonstrated the ability to match the state-of-the-art performance of DNNs using similar approaches for complex control tasks ([TKYM20], [OKG23], [ZZJX22]). But the distinction lies in the fact that these authors possessed greater experience in the field of hyperparameter tuning, regarding SNNs, and they conducted comprehensive hyperparameter searches. Due to the constraints of limited time, this work conducted only minimal hyperparameter tuning through manual adjustments. As a result, it is very likely that the performance of the PopSAN agent can be enhanced by careful and systematic hyperparameter fine-tuning.

6.2 Comparative Discussion

This section discusses how the findings of this master thesis fit into the context of previous research. To the best of my knowledge, this work represents the first successful attempt to apply the PopSAN architecture ([TKYM20]) on the task of learning contentious control to a robot with 7-DoF. Tang et al. [TKYM20] tested their PopSAN model only in simulation, whereas our study goes a step further by successfully transferring the agent from simulation to the real world. With that, we demonstrated that the PopSAN architecture can be successfully applied to high-dimensional real-world robotic problems. In contrast, typical reward-modulated approaches ([BMH+18],

[FSG13b]), failed to provide satisfactory results for high dimensional problems.

The work of Oikonomou et al. [OKG23] is the closest related to this research. In their study, the authors trained a hybrid actor-critic network with DRL on the task of target-detecting, target reaching, and collision avoidance for a 6-DoF robot. But the target that always remained at the same position. The main focus of their work was on detecting and reaching the fixed target, rather than learning the inverse kinematics of the robot. In contrast, our study demonstrates the ability of the agent to reach random target positions within its workspace, starting from random initial positions.

Kumar et al. [KHS+21] trained an agent with DRL using DNNs on the same task as this work, setting the current benchmark for continuous robot control with DNNs. Their DNN agent reached an accuracy of 0.4 cm with a success rate of 96.5 %, defining success as the agent reducing the Euclidean distance below one centimeter by episode end. In contrast, our success criterion considers a two-centimeter threshold, making direct success rate comparison useless. However, the trained PopSAN agent in this work falls short of the accuracy and success rate attained by the agent of Kumar. et. al, despite the greater threshold of two centimeters, the PopSAN agent only achieved a success rate of 91 %.

For future work, it's noteworthy that enhanced iterations of the standard PopSAN architecture have emerged. Zhang et al. [ZZJX22] refined the encoding process by introducing spatial population coding at the input layer and temporal dynamic neuron coding at hidden layers, resulting in heightened performance compared to the standard PopSAN model. Another approach for achieving better continuous control with the PopSAN architecture is proposed by Yan et al. [ZJXY22]. They identify that the conventional LIF neuron model neglects the resistance flexibility in the neuron. This limitation leads to a decreased capacity for representing continuous information, which is crucial for continuous control tasks. To overcome this, Yan et al. propose a new dynamic resistance leaky-integrate-and-fire (R-LIF) model. These two ideas come at the cost of higher training complexity, requiring even more hyperparamter tuning. However, integrating these innovations into this work has the potential to enhance the performance of the PopSAN agent.

7 Conclusion

This chapter provides a conclusion of the work conducted and the results derived from the evaluation. In addition, possible extensions and limitations of the system are discussed.

7.1 Summary

This work investigated the question of whether it is possible to achieve state-of-the-art performance for the control of a high-dimensional manipulator with a DRL agent using a SNN. Therefore, a literature review was conducted to compare the results of previous approaches. Based on this comparison, the PopSAN architecture was selected as the most promising approach for solving this task. The next step was to devise a method for evaluating the performance of the PopSAN agent. The first step of the method was to implement the PopSAN agent. However, in order to test whether the PopSAN agent could achieve state-of-the-art performance, a conventional DNN agent was also implemented. After implementing both agents, a task was designed in which the agents had to move the TCP to a random target position within their workspace, by controlling the joint velocities of the robot. In this process reward shaping played a crucial role, because the robot should not only reach the target TCP position, but also reduce its speed to zero at the goal, and the course of joint velocities should be smooth. This task was then implemented in a simulation environment. After a reasonable hyperparameter search was conducted, the best working hyperparameters were used to train the agents for 20 million time steps in the designed simulation environment.

After training the agents, a set of the following seven criteria was defined: accuracy, success rate, trajectory quality, robustness against noise, behavior in extreme situations, inference speed, and transferability. In order to evaluate the performance according to these seven criteria four different experiments were conducted. The first experiment

involved a basic scenario that mirrored the training conditions. In this scenario, the accuracy, success rate, trajectory quality, and inference speed were evaluated. The results of this scenario showed that the PopSAN agent reached 91 % of the target positions with an error of less than two centimeters. Thus, the PopSAN agent learned this task successfully. But in comparison the DNN agent achieved an average success rate of 99 % and generated smoother trajectories. Only in inference speed did the PopSAN agent outperform the DNN agent, with 536 inf /s to 380 inf /s.

Next, the robustness against noise was evaluated by running the agents again in the same scenario, but this time introducing three different levels of noise to the observations. The results indicated that the accuracy, success rate, and quality of generated trajectories of the PopSAN agent were more adversely affected by the noise compared to the DNN agent. Consequently, the DNN agent demonstrated greater robustness against noise in this evaluation. In the third experiment, the target positions were intentionally sampled at the edges of the workspace to investigate the behavior of the agents in extreme situations. In this scenario, the success rate of the PopSAN agent dropped by 23 % compared to the basic scenario, while the performance of the DNN agent only decreased by 4 %. This observation clearly demonstrates the superior performance of the DNN agent in handling extreme situations. Lastly, the transferability of the agents was evaluated by transferring the agents from simulation onto a real robot. The DNN agent demonstrated the ability to maintain its achieved performance from simulation, even in the real-world setting. However, in contrast, the success rate of the PopSAN agent decreased by 21 % compared to the basic scenario, primarily due to its lower robustness against noise.

In summary, the DNN agent demonstrated superior performance over the PopSAN agent in all evaluation criteria, except for inference speed. Hence, this research successfully established that the complex task of robot control for a high-DoF robot can indeed be learned through the utilization of the PopSAN architecture. However, to achieve state-of-the-art performance, further exploration and future work are required.

7.2 Future work

The achieved results show promise, but they also reveal areas for improvement. As this work paves the way for future research in this domain, several directions for future work can be explored to enhance the performance of the PopSAN agent and bridge the gap with the current state-of-the-art DNN agents.

Hyperparameter Optimization: Since SNNs are sensitive to the additional hyperparameters introduced by spiking neuron models, like current, voltage, decay factors, and firing thresholds, extensive hyperparameter tuning is inevitable [AEWK23]. As the agent is in general capable of learning this task and only very little hyperparameter tuning was conducted in this work, it can be assumed that the performance of the PopSAN agent can be further improved by conducting extensive hyperparameter optimization.

Different neuron models: The generated trajectories of the PopSAN agent have a higher total variance, which indicates a less efficient use of the continuous actions space. Yan et al. [ZJXY22] state that the LIF neuron model is not optimal for representing continuous information. Therefore, they propose an alternative R-LIF model, which could potentially enhance the agent's performance. Thus, future work should explore different neuron models to improve the agent's capabilities in continuous control tasks.

Encoding: The authors of [ZZJX22] have shown that integrating dynamic neurons coding with population coding can lead to improved decision-making capabilities in the PopSAN architecture. Therefore, building upon this insight, a promising direction for future work is to explore the enhancement of the encoding mechanism for population coding.

Bibliography

[AEWK23] Mahmoud Akl, Deniz Ergene, Florian Walter, and Alois Knoll. Toward robust and scalable deep spiking reinforcement learning. *Frontiers in Neurorobotics*, 16, 2023.

[BCP+16] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[Ber] Yamins D. Cox D. D. Bergstra, J. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures.

[BMH+18] Zhenshan Bing, Claus Meschede, Kai Huang, Guang Chen, Florian Rohrbein, Mahmoud Akl, and Alois Knoll. End to end learning of spiking neural network based on r-stdp for a lane keeping vehicle. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4725–4732, 2018.

[CB21] Erwin Coumans and Yunfei Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning. `http://pybullet.org`, 2016–2021.

[CD08] Natalia Caporale and Yang Dan. Spike timing-dependent plasticity: A hebbian learning rule. *Annual Review of Neuroscience*, 31:25–46, 2008.

[DSL+18] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, Yuyun Liao, Chit-Kwan Lin, Andrew Lines, Ruokun Liu, Deepak Mathaikutty, Steven McCoy, Arnab Paul, Jonathan Tse, Guruguhanathan Venkataramanan, Yi-Hsin Weng, Andreas Wild, Yoonseok Yang, and Hong Wang. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38(1):82–99, 2018.

[DVS01] A. D'Souza, Sethu Vijayakumar, and Stefan Schaal. Learning inverse kinematics. volume 1, pages 298 – 303 vol.1, 02 2001.

[FSG13a] Nicolas Frémaux, Henning Sprekeler, and Wulfram Gerstner. Reinforcement learning using a continuous time actor-critic framework with spiking neurons. *PLOS Computational Biology*, 9(4):1–21, 04 2013.

[FSG13b] Nicolas Frémaux, Henning Sprekeler, and Wulfram Gerstner. Reinforcement learning using a continuous time actor-critic framework with spiking neurons. *PLOS Computational Biology*, 9(4):1–21, 04 2013.

[GKNP14] Wulfram Gerstner, Werner M. Kistler, Richard Naud, and Liam Paninski. *Neuronal Dynamics: From Single Neurons to Networks and Models of Cognition*. Cambridge University Press, 2014.

[GSK86] Apostolos P. Georgopoulos, Andrew B. Schwartz, and Ronald E. Kettner. Neuronal population coding of movement direction. *Science*, 233(4771):1416–1419, 1986.

[KHS+21] Visak Kumar, David Hoeller, Balakumar Sundaralingam, Jonathan Tremblay, and Stan Birchfield. Joint space control via deep reinforcement learning, 2021.

[KIP+18] Dmitry Kalashnikov, Alex Irpan, Peter Pastor, Julian Ibarz, Alexander Herzog, Eric Jang, Deirdre Quillen, Ethan Holly, Mrinal Kalakrishnan, Vincent Vanhoucke, and Sergey Levine. Qt-opt: Scalable deep reinforcement learning for vision-based robotic manipulation, 2018.

[LX22] Sijia Lu and Feng Xu. Linear leaky-integrate-and-fire neuron model based spiking neural networks and its mapping relationship to deep neural networks. *Frontiers in Neuroscience*, 16, 2022.

[OKG23] Katerina Maria Oikonomou, Ioannis Kansizoglou, and Antonios Gasteratos. A hybrid reinforcement learning approach with a spiking actor network for efficient robotic arm target reaching. *IEEE Robotics and Automation Letters*, 8(5):3007–3014, 2023.

[OS13] Michael J. O'Brien and Narayan Srinivasa. A spiking neural model for stable reinforcement of synapses based on multiple distal rewards. *Neural Computation*, 25(1):123–156, 2013. Epub 2012 Sep 28.

[PHS+19a] Devdhar Patel, Hananel Hazan, Daniel J. Saunders, Hava Siegelmann, and Robert Kozma. Improved robustness of reinforcement learning policies upon conversion to spiking neuronal network platforms applied to atari games, 2019.

[PHS+19b] Devdhar Patel, Hananel Hazan, Daniel J. Saunders, Hava T. Siegelmann, and Robert Kozma. Improved robustness of reinforcement learning policies upon conversion to spiking neuronal network platforms applied to atari breakout game. *Neural Networks*, 120:108–115, 2019. special Issue in Honor of the 80th Birthday of Stephen Grossberg.

[RHW86] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning Representations by Back-propagating Errors. *Nature*, 323(6088):533–536, 1986.

[RSPR20] Nitin Rathi, Gopalakrishnan Srinivasan, Priyadarshini Panda, and Kaushik Roy. Enabling deep spiking neural networks with hybrid conversion and spike timing dependent backpropagation, 2020.

[RV.07] Florian RV. Reinforcement learning through modulation of spike-timing-dependent synaptic plasticity. jun 2007.

[SB98] Richard S. Sutton and Andrew G. Barto. Simple reinforcement learning with a critic. *Adaptive Behavior*, 5(2):219–246, 1998.

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[Sta] Stanford Artificial Intelligence Laboratory et al. Robotic operating system.

[STJL17] Fereshteh Sadeghi, Alexander Toshev, Eric Jang, and Sergey Levine. Sim2real view invariant visual servoing by recurrent control, 2017.

[SWD+17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[TKYM20] Guangzhi Tang, Neelesh Kumar, Raymond Yoo, and Konstantinos P. Michmizos. Deep reinforcement learning with population-coded spiking neural network for continuous control, 2020.

[TPK20] Weihao Tan, Devdhar Patel, and Robert Kozma. Strategy and benchmark for converting deep q-networks to event-driven spiking neural networks, 2020.

[vH21] Hado van Hasselt. Lecture 9: Policy gradients and actor critics, 2021.

[WAN02] Si Wu, Shun-ichi Amari, and Hiroyuki Nakahara. Population coding and decoding in a neural field: A computational study. *Neural computation*, 14:999–1026, 06 2002.

[WDL+18] Yujie Wu, Lei Deng, Guoqi Li, Jun Zhu, and Luping Shi. Spatio-temporal backpropagation for training high-performance spiking neural networks. *Frontiers in Neuroscience*, 12, 2018.

[YWYT19] Mengwen Yuan, Xi Wu, Rui Yan, and Huajin Tang. Reinforcement Learning in Spiking Neural Networks with Stochastic and Deterministic Synapses. *Neural Computation*, 31(12):2368–2389, 12 2019.

[ZJXY22] Jie Zhang, Runhao Jiang, Rong Xiao, and Rui Yan. Dynamic resistance based spiking actor network for improving reinforcement learning. In *Proceedings of the 8th International Conference on Computing and Artificial Intelligence*, ICCAI '22, page 18–23, New York, NY, USA, 2022. Association for Computing Machinery.

[ZZJX22] Duzhen Zhang, Tielin Zhang, Shuncheng Jia, and Bo Xu. Multi-sacle dynamic coding improved spiking actor network for reinforcement learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36(1):59–67, Jun. 2022.

A Appendix

A.1 Hyperparameters

Table A.1a and Table A.1b list the hyperparameters used to train both agents with the PPO algorithm.

(a) Hyperparameters that are used for training the PopSAN agent

PopSAN	
ppo_epochs	25
batch_size	512
step_per_epoch	1000
gamma	0.99
clip_ratio	0.2
critic_lr	1×10^{-5}
actor_lr	1×10^{-5}
lam	0.95
beta	0.001
hidden_size	(256, 256)
encoder_pop_dim	10
decoder_pop_dim	10
mean_range	(-3, 3)
std	$\sqrt{0.15}$
spike_ts	5

(b) Hyperparameters that are used for training the DNN agent

DNN	
entropy-coeff	0.2
num-sgd-iter	30
sgd-minibatch-size	128
clip-param	0.5
lr	1×10^{-5}
horizon	500
train-batch-size	1024
lambda	0.95

A.2 Total variation

Total variation is a mathematical concept used to quantify the amount of variation or "jaggedness" in a signal, function, or data. In the context of joint angle trajectories, total variation measures how much the joint angles q change from one time step to the next, indicating the overall smoothness or roughness of the trajectory, see Equation A.1.

$$TV(x) = |q_{t+1} - q_t| + |q_{t+2} - q_{t+1}| + ... + |q_{t+n} - q_{t+(n-1)}| \tag{A.1}$$

A.3 Additional Results

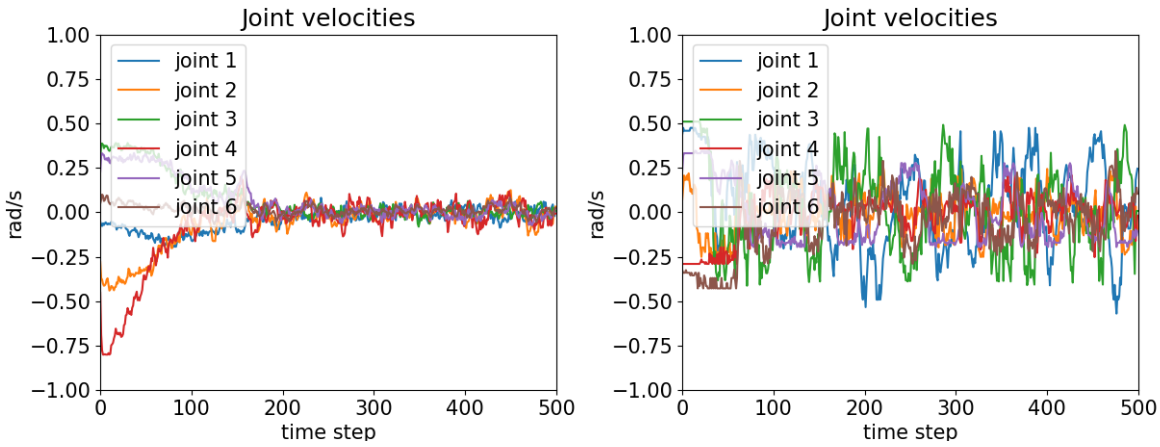
This section provides additional results generated by the two agents in the different scenarios.

A.3.1 Noise scenario

Table A.2 shows the success rates and accuracies obtained under the influence of light Gaussian noise. Figure A.1 and Figure A.2 illustrate the generated joint velocities under the influence of light noise.

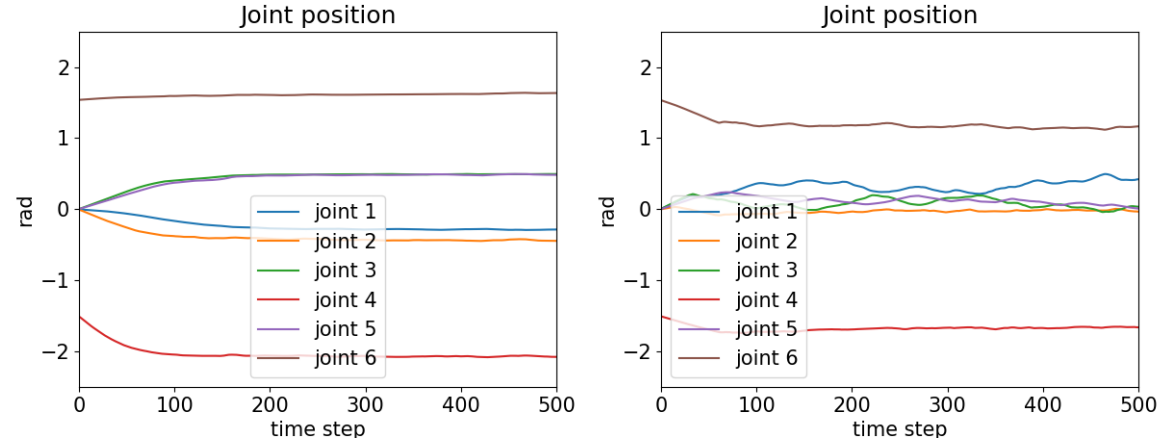
Table A.2: Achieved accuracy (ac) and success rate (sr) of the agents, when light Gaussian noise is applied in simulation.

with light noise												
	run 1		run 2		run 3		run 4		run 5		average	
	ac	sr	ac	sr	ac	sr	ac	sr	ac	sr	ac	sr
PopSAN	1.10	89	1.04	92	1.05	88	1.02	85	1.09	89	1.04	88.6
DNN	0.42	100	0.39	100	0.39	100	0.37	100	0.37	100	0.39	100



(a) (b)

Figure A.1: Generated joint velocities under the influence of light noise: (a) DNN agent, (b) PopSAN agent. Starting and ending at identical positions.



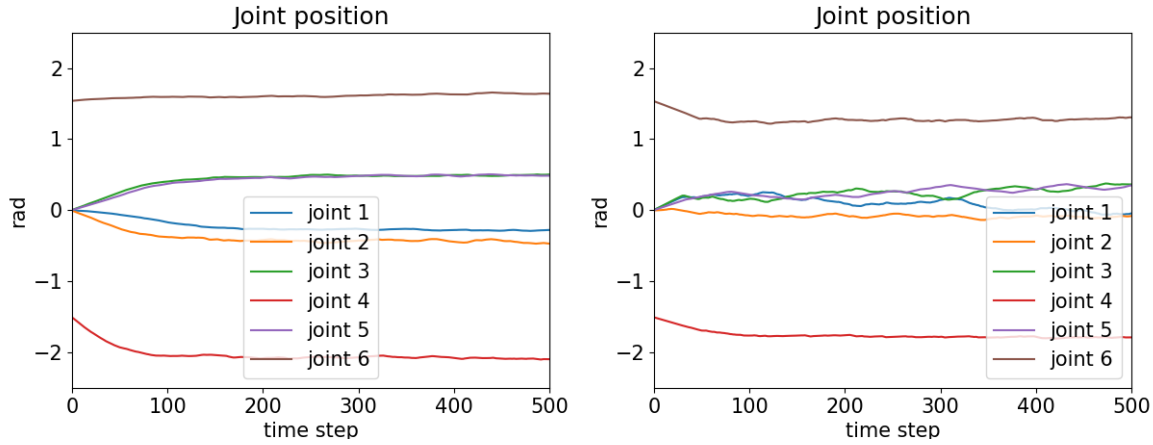
(a) (b)

Figure A.2: Joint angles under the influence of light noise. (a) DNN agent, (b) PopSAN agent. Starting and ending at identical positions.

Table A.3 shows the success rate and accuracy obtained under the influence of medium Gaussian noise. Figure A.3 And Figure A.4 illustrates the generated joint angles and joint velocities under the influence of medium noise.

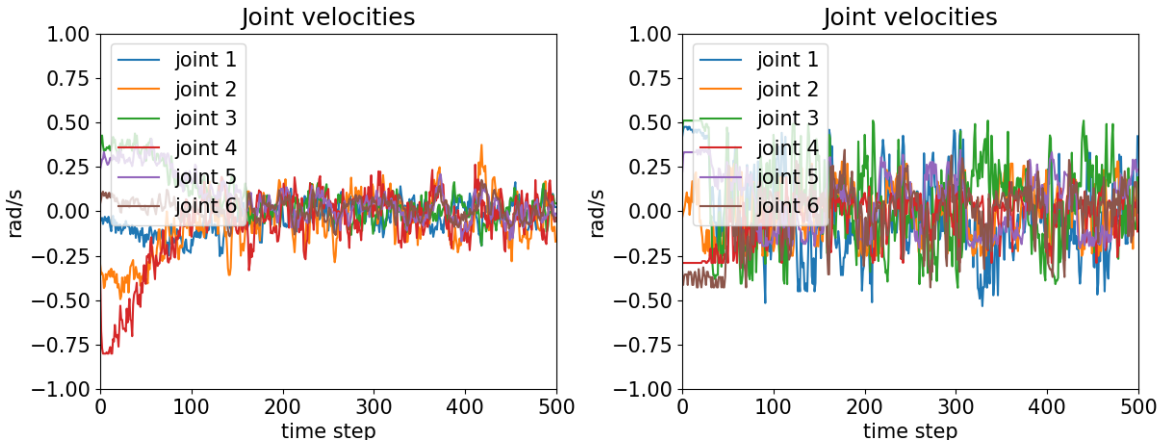
Table A.3: Achieved accuracy (ac) and success rate (sr) of the agents, when medium Gaussian noise is applied in simulation.

with medium noise												
	run 1		run 2		run 3		run 4		run 5		average	
	ac	sr	ac	sr	ac	sr	ac	sr	ac	sr	ac	sr
PopSAN	1.19	75	1.27	65	1.29	65	1.25	73	1.21	66	1.24	68.8
DNN	0.75	100	0.80	100	0.82	100	0.82	99	0.82	97	0.79	99.2



(a) (b)

Figure A.3: Joint angles under the influence of medium noise of the (a) DNN agent, (b) PopSAN agent. Starting and ending at identical positions.



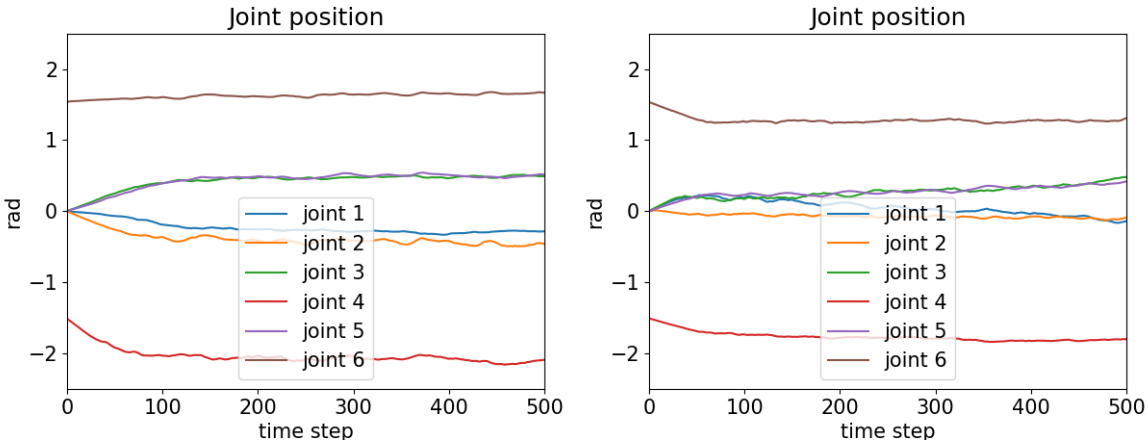
(a) (b)

Figure A.4: Joint velocities under the influence of medium noise of the (a) DNN agent, (b) PopSAN agent. Starting and ending at identical positions.

Table A.4 shows the success rates and accuracies obtained under the influence of heavy Gaussian noise. And Figure A.5 and Figure A.6 illustrates the generated joint angles and joint velocities under the influence of heavy Gaussian noise.

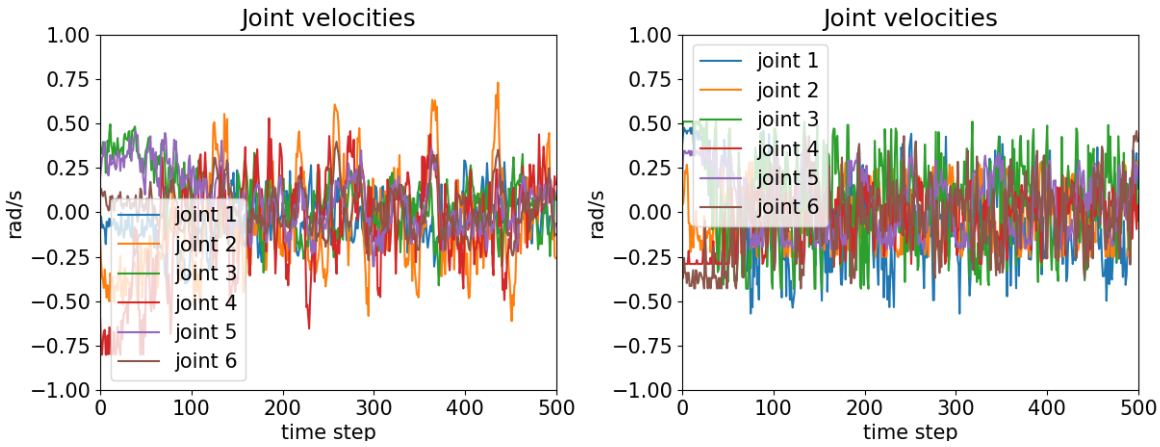
Table A.4: Achieved accuracy (ac) and success rate (sr) of the agents, when heavy Gaussian noise is applied in simulation.

with light noise												
	run 1		run 2		run 3		run 4		run 5		average	
	ac	sr	ac	sr	ac	sr	ac	sr	ac	sr	ac	sr
PopSAN	1.36	32	1.31	40	1.44	45	1.21	34	1.35	34	1.33	37
DNN	1.33	71	1.21	75	1.25	79	1.21	81	1.33	79	1.25	76.6



(a) (b)

Figure A.5: Joint angles under the influence of heavy noise of the (a) DNN agent, (b) PopSAN agent. Starting and ending at identical positions.



(a) (b)

Figure A.6: Joint velocities under the influence of heavy noise of the (a) DNN agent, (b) PopSAN agent. Starting and ending at identical positions.

A.3.2 Extreme scenario

Figure A.7 shows the generated joint trajectories in the extreme scenario.

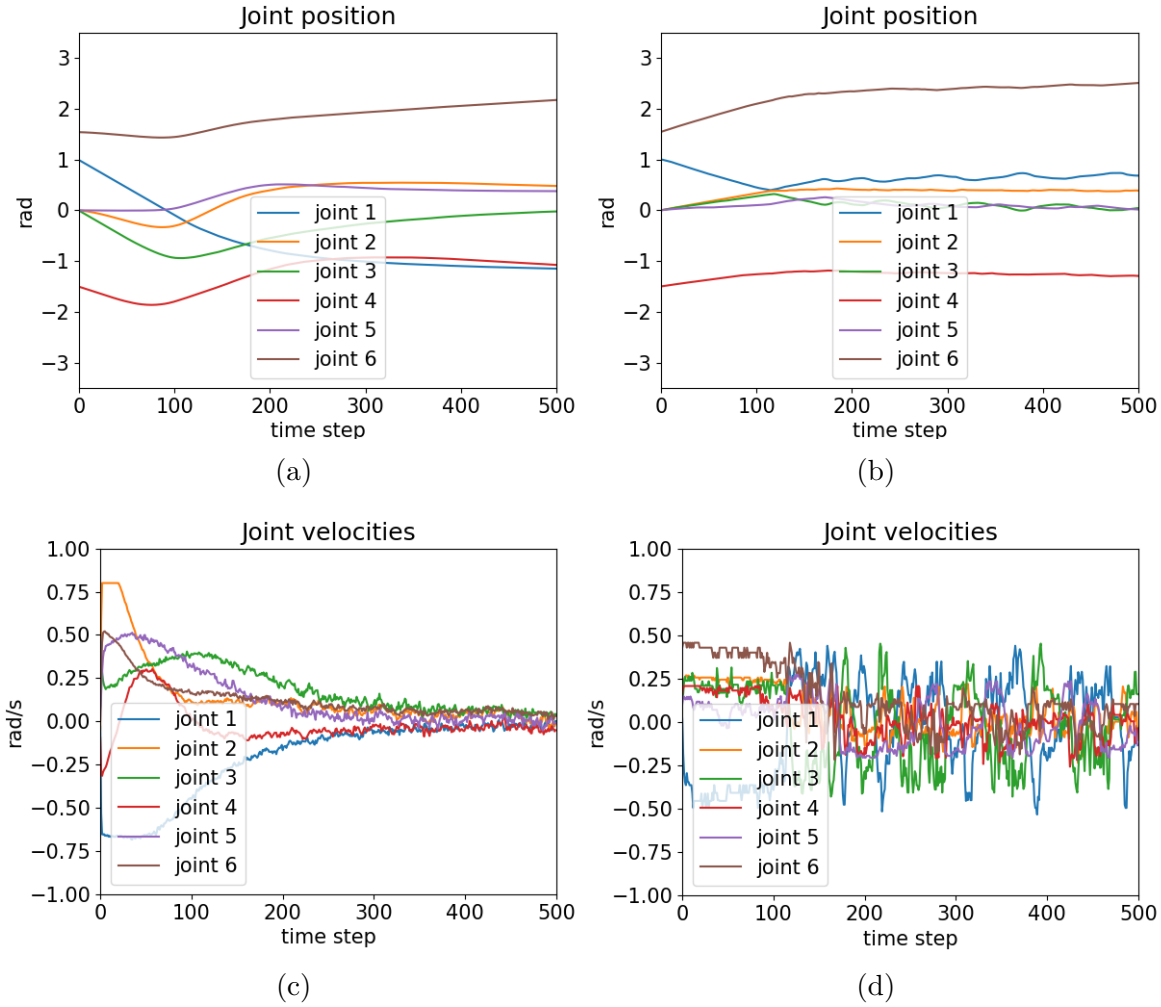
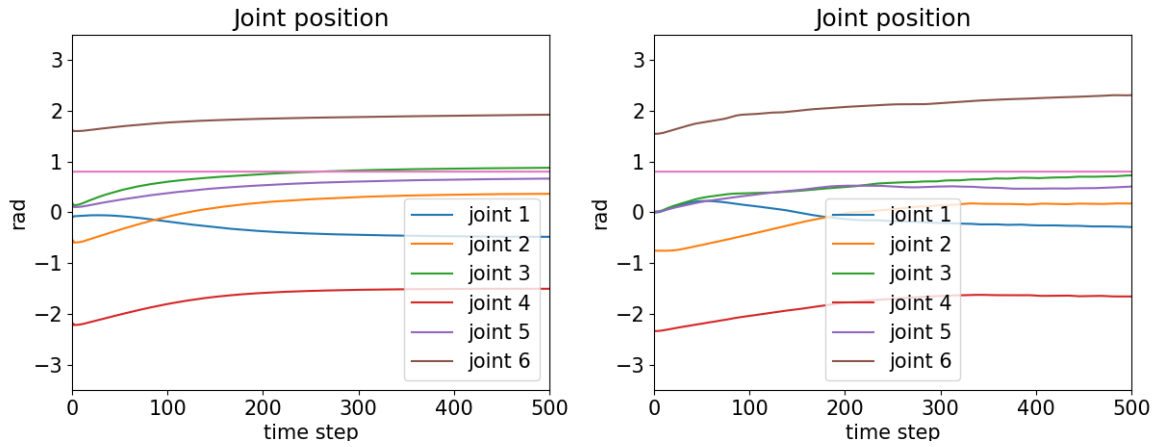


Figure A.7: Course of the joint angles and joint velocities during the extreme scenario: (a) DNN agent joint angles, (b) PopSAN agent joint angles, (c) DNN agent joint velocities and (d) PopSAN agent joint velocities. Starting and ending at identical positions.

A.3.3 Real World scenario

Figure A.8 shows the course of the joint angles of the real robot.



(a) (b)

Figure A.8: Joint angles of the real robot generated by (a) the DNN agent, (b) the PopSAN agent. Starting and ending at identical positions.

Statement of Authorship

I hereby confirm to the Chemnitz University of Technology that this thesis is exclusively my own work and does not use any external materials other than those stated in the text.

This work contains no plagiarism and all sentences or passages directly quoted from other people's work or including content derived from such work have been specifically credited to the authors and sources.

This paper has neither been submitted in the same or a similar form to any other examiner nor for the award of any other degree, nor has it previously been published.

Chemnitz, August 8, 2023

Place and date | Signature