Distributed Reinforcement Learning with Neural Networks for Robotics

Verteiltes bestärkendes Lernen mit neuronalen Netzen für Robotik
Master-Thesis von Denny Dittmar
Tag der Einreichung:

1. Gutachten: Dr. Elmar Rueckert
2. Gutachten: Prof. Dr. Jan Peters

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Distributed Reinforcement Learning with Neural Networks for Robotics
Verteiltes bestärkendes Lernen mit neuronalen Netzen für Robotik

Vorgelegte Master-Thesis von Denny Dittmar

1. Gutachten: Dr. Elmar Rueckert
2. Gutachten: Prof. Dr. Jan Peters

Tag der Einreichung:

Erklärung zur Master-Thesis

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

Darmstadt, den 2. Januar 2018

(Denny Dittmar)

Abstract

We propose and investigate a novel type of parameterized policy based on a two-layered stochastic spiking neural network consisting of multiple populations of stochastic spiking neurons. We show that the proposed policy type is a spatially distributed generalization of a discrete basic policy with lookup table parameterization. Our policy reveals remarkable capabilities but also crucial limitations. In particular, our policy is able to deal with high-dimensional state and action spaces but loses expressive power such that it cannot represent certain functions like XOR.
Furthermore, we propose corresponding reinforcement learning methods to train the policy. These methods are based on value function methods and generalize these to train our distributed policy type. We compare these methods to state-of-the-art methods including black-box approaches and likelihood ratio approaches. It turns out that our proposed methods outperform these methods significantly. In our experiments we demonstrate that our policy can be trained effectively from rewards to guide a 10-link robot-arm in a toy task through a grid world to reach a specified target without hitting obstacles.

Zusammenfassung

Wir schlagen eine neuartigen parametrisierten Policytyp vor, der auf einem 2-lagigem stochastischem gepulsten neuronalem Netz bestehend aus mehreren Populationen von stochastischen gepulsten Neuronen basiert und untersuchen diesen. Wir zeigen, dass der vorgeschlagene Policytyp eine räumlich verteilte Verallgemeinerung einer Basispolicy mit Lookup-Tabellen-Parametrisierung darstellt. Unser Policytyp offenbart bemerkenswerte Fähigkeiten, aber auch bedeutende Limitierungen. Unsere Policy kann insbesondere mit hochdimensionalen Zustands- und Aktionsräumen umgehen, verliert aber auch Ausdrucksmächtigkeit und kann daher bestimmte Funktionen wie XOR nicht repräsentieren.
Desweiteren schlagen wir Methoden vor, um diesen Policytyp zu trainieren. Diese Methoden basieren auf Wertefunktionsmethoden und verallgemeinern diese, um unseren verteilten Policytyp zu trainieren. Wir vergleichen diese Methoden mit State-of-the-art-Methoden, die Black-box-Ansätze and Likelihood-ratio-Ansätze umfassen. Es stellt sich heraus, dass unsere vorgeschlagenen Ansätze diese State-of-the-art-Methoden deutlich übertreffen. In unseren Experimenten demonstrieren wir, dass unsere Policy effektiv mit Belohnungen trainiert werden kann, um einen 10-gelenkigen Roboterarm in einem Toytask durch eine Gitterwelt zu bewegen, um ein vorgegebenes Ziel zu erreichen, wobei keine Hindernisse berührt werden dürfen.

Contents

Figures and Tables

List of Figures

List of Tables

1 Introduction

Intelligent autonomous agents are supposed to improve our lives by solving tasks for us that rely on a certain degree of intelligence. For example, this can comprise applications with an entertaining, a medical, or an industrial background. The most relevant aspect of an intelligent autonomous agent is its policy, i.e., the program that defines how the agent acts in a certain situation. More formally, a policy $\pi : S \rightarrow A$ is mapping from the current state $s \in S$ to an action $a \in A$ where the definitions of the state space S, the action space A or the policy π do not have significant restrictions. However, the realization of these policies is also the most difficult aspect.
We can distinguish between at least two different ways to realize such a policy. One way is to hard-code these policies where analytic expert knowledge is translated into a code representation. Unfortunately, for many tasks such an approach can be extraordinary time consuming. Furthermore, many relevant aspects of systems with a high degree of complexity cannot be captured completely by expert knowledge. Consequently, such an approach often turns out to be infeasible. A promising alternative to hard-coding policies is to learn them. The ability to learn behavior from experiences and to generalize it to unforeseen situations can be seen as a crucial property of intelligence as it is also employed by humans as we know. Without the ability to learn humans could not produce a behavior that could be interpreted as intelligent. In the context of machine learning a policy is learned by representing it in a parameterized way and by optimizing the parameters according to some scalar performance measure that corresponds to a goal-based policy definition. Goal-based policy definitions are more abstract and easier to produce than concrete hard-coded policies.
However, learning a policy still relies on a proper choice of its parameterized representation and the learning method. There are several popular learning frameworks and corresponding learning methods to address the problem of learning policies. The most general and powerful learning framework in which probably each learning problem can be formulated is reinforcement learning. In reinforcement learning the performance of a policy is judged by scalar rewards and the learning is done by exploratory interaction of the agent with its environment. However, a general approach for representing parameterized policies and training them efficiently from rewards in a reinforcement learning setting even for high-dimensional state and action spaces is still very difficult and most state-of-the-art methods are limited to low-dimensional state and action spaces.

1.1 Related work

The foundation of the work presented in this thesis is provided by the work of [1, 2] where populations of stochastic spiking neurons are used to solve planning problems for robots. In these planning problems the goal was to find a proper trajectory distribution of a robot that starts in some initial location and ends in a desired target location while fulfilling certain constraints such as avoiding to hit obstacles. In [1] different populations of stochastic spiking neurons were used to represent a distribution of trajectories in a two-dimensional task space. The distribution was initially trained from demonstration data gathered from kinesthetic teaching and refined with reinforcement learning afterwards. For both training phases specific Hebbian learning rules were developed. For the training from demonstrations a learning approach based on contrastive divergence [3] was used. For the refinement with reinforcement learning a reward modulated Hebbian learning rule related to REINFORCE [4] was used. However, the approach in [1] was not able to represent trajectory distributions for a high-dimensional joint space of a robot arm. To scale to high-dimensional spaces the approach was in [2] extended by employing multiple neuron populations. Here, each joint was assigned a single population that depended on a single task population of neurons. By doing so it enabled the representation of trajectories in a high-dimensional joint space without relying on an inverse kinematics model. In our experiments we make use of a stochastic spiking network with a network topology and a population encoding similar to the ones used in [2] to learn to move a robot arm with multiple joints through a grid-world without hitting obstacles.

Other inspiring work which is closely related to the approach presented in this thesis was found in [5, 6, 7, 8], where semi-stochastic feed-forward networks with stochastic winner-take-all dynamics in the output-layer and corresponding reinforcement learning rules to train these were proposed and investigated. The basis of this work is provided by the framework in [5] with the name "attention-gated reinforcment learning". The used network is similar to a standard multilayer pereceptron with the difference that the dynamics in the output layer employ stochastic winner-take-all dynamics. Furthermore, reinforcement learning rules were derived to train such a network from rewards. For classification tasks the reinforcement learning rules were compared to a supervised learning approach based on backpropagation and it turned out that the reinforcement learning approach was able to compete with the supervised learning approach. It is also argued that supervised learning and particularly the backpropagation algorithm is not biologically plausible. However,

the way to update the non-stochastic hidden units in this framework is still very similar to backpropagation. This is also underlined in [8] where it was shown that the update rules in a certain sense are even equivalent to backpropagation. The framework proposed in [5] uses only a single winner-take-all circuit in the output layer and was in [6] extended to a framework that is closely related to the framework in this thesis. Here, a semi-stochastic multi-layer feed-forward network with multiple winner-take-all circuits in the output layer was employed and a value function related reinforcement learning method was used to train this network. The network topology is similar to the network topology we used in our framework and the used reinforcement learning method corresponds to the distributed global value function method we used to train our policy. As in [5] the network topology in [6] contains non-stochastic continuous hidden units that are trained with a backpropagation-like approach, which differs from our approach as we only use populations of stochastic spiking neurons as elementary processing units.

1.2 Overview

In the following we give an overview over the structure of this thesis. In chapter 2 we provide foundations of reinforcement learning. In chapter 3 we provide foundations of artificial neural networks and distributed models of computation. In chapter 4 we introduce our proposed distributed policy based on a stochastic spiking neural network and corresponding learning methods to train this network. In chapter 5 we evaluate the proposed policy for different learning methods on different toy tasks based on a 10-link robot arm. In particular, this includes a sequential task where the goal is to navigate a robot arm through a grid-world such that it hits a target while avoiding obstacles. In chapter 6 we give our conclusion and in chapter 7 we discuss possible future work.

2 Foundations of reinforcement learning

Reinforcement learning is a general learning framework for the goal-based specification of desired behaviors of agents acting in an environment. Probably, each machine learning problem can be formulated in terms of reinforcement learning. In particular, this includes supervised learning problems, imitation learning problems and bandit problems. However, it is still an unanswered question whether supervised learning problems and imitation learning problems can in general be solved effectively in the context of reinforcement learning.
In reinforcement learning, the goal is to achieve a behavior of an agent such that sequences of actions executed by the agent maximize a scalar performance measure, the return. The behavior of the agent is defined by a parameterized policy, i.e., the program of the agent, that maps the received signals from the environment to an action. In realistic scenarios these state signals are only partially observable. At each time step a reward is received that depends on the state of the environment and the executed action at this time step. The agent has to execute a proper sequence actions to maximize the return defined as the discounted sum of future rewards. For finding an optimal policy the agent has to interact with its environment in an exploratory manner. Formally a reinforcement learning setting can be described by a Markov decision process (MDP).

2.1 Formalizing reinforcement learning problems

A reinforcement learning problem can be formalized by a Markov decision process (MDP). A Markov decision process MDP=(S, A, T, r, γ) consists of a state space S, an action space A, transition probabilities $T(s, a, s') = p(s'|s, a)$, a reward function $r(s, a)$ and a discount factor $\gamma \in [0, 1[$ where $s \in S$ is the current state, $a \in A$ the action executed in the current state and $s' \in S$ is the successor state. The transition $T(s, a, s')$ represents the dynamics of the environment, i.e., the probability for changing from state s to state s' after executing action a and satisfies the Markov property. The user defined reward function $r(s, a)$ maps the state s and the action a executed by the agent to a reward r received by the agent. The discount factor γ specifies the weights of future rewards.
The policy of the agent maps the current state s to an action a, i.e., $\pi(s) = a$. Policies can also be realized in a stochastic way where we have $\pi(s, a) = p(a|s)$. The objective in reinforcement is to find an optimal policy such that the expected return given by $J(\pi) = E[R_1|\pi] = E[\sum_{t=1}^{T} \gamma^t r_t|\pi]$ resulting from following policy π under the transitions and rewards defined by the MDP is maximized. T corresponds to the number of time steps until the episode terminates. In infinite horizon settings with $T = \infty$, due to $\gamma \in [0, 1[$ this return is guaranteed to converge if the rewards r are bounded. Bandit problems can be modeled by one-step MDPs with $T = 1$.

2.2 Challenges in reinforcement learning

We can identify different challenges when learning a policy that maximizes the expected return in a reinforcement learning setting. A part of these is the ability to deal with high-dimensional state and action spaces, delayed rewards in tasks with sequential decision-making and partially observable states.
Complex action and state spaces make it hard to identify the relationship between states and corresponding actions that are advantageous for these. Furthermore, the policy must be represented in a way such that the mapping from states and actions with a high complexity can be realized in a computationally efficient way. Unfortunately, the complexity of state and action spaces, i.e., their volume, increases rapidly with their dimensionality. This problem is also know as the curse of dimensionality [9].
Another challenge are delayed rewards where $\gamma > 0$ holds, i.e., the problem that the return for an action is not instantly available after its execution. This is a problem, since the correct knowledge of the true returns is essential to realize sequential decision-making properly.
Another challenge is the problem of partially observable states which means that the agent cannot observe the complete state of the environment. In this case the agent may not be able to infer the best action from its current observation and relies also on observations made in the past to realize proper decision-making. In such a case the agent has to maintain an internal state that relies on several past observations. A reinforcement learning problem with partially observable states can be formalized by a partially observable MDP (POMDP).

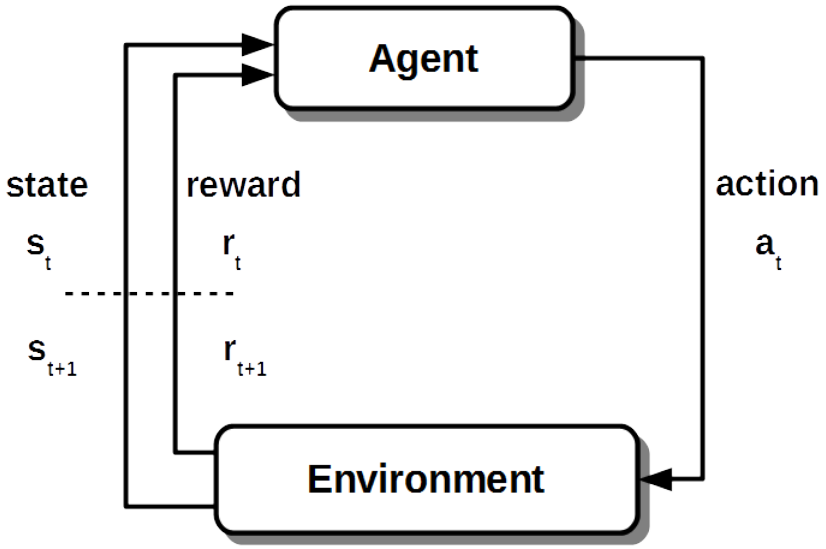

Figure 2.1: Interaction between an agent and the environment in a reinforcement learning setting.

2.3 Reinforcement learning methods

In the following we will show different methods[10, 11, 12, 13] to find a policy that achieves a return expectation as high as possible for a given MDP. This is done by optimizing the parameters θ of a parameterized policy $\pi(s, a; \theta)$.

2.3.1 Model-free vs model-based methods

Reinforcement learning methods can be categorized in model-based and model-free methods. Model-based approaches make use of a transition model to realize a simulator of the MDP that can be used to optimize the policy without the necessity of evaluating the policy on a real system. In a robotics context, this reduces the amount of time that has to be invested in experiments on a real system as well as wear and tear caused to the robot. Model-free methods do not employ transition models or rely on them but can be extended straightforward to a model-based approach by applying them to a simulated MDP. A big challenge of model-based methods is to find a model with high accuracy as model errors lead to degraded policies. This is a reason why model-free approaches are often preferred.
A remarkable example for a model-based methods is PILCO[14]. PILCO realizes the transition model with Gaussian processes and optimizes the policy with gradient descent based on analytic gradients. PILCO was able to learn policies for cart-pole, double cart-pole and unicycle successfully from scratch with only a small number of interactions with the real system that were used to learn the transition model.

2.3.2 Black-box vs white-box methods

Reinforcement learning methods can be categorized in black-box and white-box methods. Black-box methods perturb the parameters at the start of an episode and use only the sampled returns from the episodes to learn the policy. That is, they treat the MDP as a black-box and do not make any use of the inner structure of the RL problem. In particular, the state-action pairs that occurred during an episode are not taken into account. The advantage of black-box methods is that they can be applied to each type of parameterized policy, including non-differentiable or non-stochastic policies. This makes black-box methods flexible and convenient to apply. The perturbation of the policy parameters is often realized by stochastic Gaussian noise, i.e., $\Delta\theta \sim \mathcal{N}(0, \epsilon)$. Examples for black-box methods are finite different gradient methods belonging to policy gradient methods[11] and evolutionary algorithms like CMA-ES[15]. CMA-ES is currently considered as state-of-the-art in black-box optimization.
In contrast to black-box methods, white-box DPS methods make use of the internal structure of the RL problem, i.e., state-action pairs occurring during an episode are taken into account. From a theoretical point of view, this makes them more powerful than black-box methods. White-box methods are often only applicable to specific types of policies. Despite

the better performance of white-box methods in theory, black-box methods are often preferred over white-box methods. Examples for white-box methods are likelihood-ratio methods[11, 4] and value function methods[10].

2.3.3 Likelihood ratio methods

Likelihood ration methods[11, 4] are model-free white-box methods that rely on stochastic differentiable policies. They belong to the class of policy gradient methods. Therefore, they optimize a policy $\pi(s,a;\theta)$ by gradient descent based on an estimate of the gradient of the corresponding parameterized return $\nabla_\theta J(\theta)$, i.e., $\Delta\theta = \alpha\nabla_\theta J(\theta)$. This is done by by making use of the likelihood ratio or REINFORCE trick given by $\nabla_\theta p(\tau|\theta) = p(\tau|\theta)\nabla_\theta \log p(\tau|\theta)$ with $\tau = (s_1, a_1, ..., s_T, a_T)$ as path. Using this trick, we can write the policy gradient as $\nabla_\theta J(\theta) = \int_\tau \nabla_\theta p(\tau|\theta) R^\tau = \int_\tau p(\tau|\theta)\nabla_\theta \log p(\tau|\theta) R^\tau = E\{\nabla_\theta \log p(\tau|\theta) R^\tau\} = E\{\sum_{t=1}^{T} \nabla_\theta \log \pi(s_t, a_t; \theta) R^\tau\}$ where we made use of $\log \nabla_\theta \log p(\tau|\theta) = \sum_{t=1}^{T} \nabla_\theta \log \pi(s_t, a_t; \theta)$ with R^τ as the return received for path τ. Since the resulting gradient estimate does not depend on the transitions likelihood ration methods can be applied in a model-free fashion.

2.3.4 Value function methods

Value function (VF) methods [10] are traditional reinforcement learning methods for learning optimal policies. As likelihood ratio methods they are white-box methods. For simple multi-armed bandit problems, i.e., one-step MDPs with only a single state, where the action spaces are low-dimensional value function related approaches are still the most effective methods[16, 17, 18]. However, due to their bad scaling to high-dimensional state and action spaces they lost in popularity and became more and more replaced by other types of methods such as black-box methods or likelihood ratio methods.
In VF methods value functions are estimated and employed that map a state or a state-action pair to the expected return if starting and following from this state or state-action pair under some policy. Thus, value functions make a prediction and have a model property. Despite this fact, value functions are usually are not considered as models.
We can discriminate between two types of value functions. These are state value functions and action value functions. A state-value function $V^\pi : S \to \mathscr{R}$ is a mapping from a state s to the expected return received if starting in state s and following policy π, i.e., $V^\pi(s) = E[R_t|s_t = s; \pi] = E[\sum_{k=1}^{\infty} r_{t+k}|s_t = s; \pi]$. Similarly an action-value function $Q^\pi : S \times A \to \mathscr{R}$ is a mapping from a state-action pair (s,a) to the corresponding expected return under policy π, i.e., $Q^\pi(s,a) = E[R_t|s_t = s, a_t = a; \pi] = E[\sum_{k=1}^{\infty} r_{t+k}|s_t = s, a_t = a; \pi]$. The estimates of these a priori unknown value functions are denoted by $V(s)$ and $Q(s,a)$ in the following.

The general working principle of VF based reinforcement learning can be expressed by generalized policy iteration (GPI). GPI consists of two key parts that are repeated alternately such that both the estimated value function and policy converge to optimality. One key part is to estimate the value function for a given policy and the other key part is to improve this policy by selecting actions that lead to promising outcomes according to the estimated value function. The actions are selected in a greedy manner, i.e. the best action is chosen. Estimating value functions for a given policy is called policy evaluation and improving policies according to learned value functions is called policy improvement. The principle of GPI is visualized in figure 2.2.

There are different VF methods which differ mainly in how the policy evaluation is done, i.e. the estimation of the value function and can be divided into the classes dynamic programming (DP) methods, Monte Carlo (MC) methods and temporal difference (TD) methods. DP methods are model-based methods whereas MC and TD methods can be used model-free if an action-value function is employed. The second key element of GPI is the policy improvement step, which improves a policy by selecting proper actions for a given state according to the estimated value function. In the simplest case this can be realized by just selecting actions in a greedy manner. For an estimated state-value or action-value function this gives

$$\pi(s) = \arg\max_a \sum_{s'} T(s,a,s')(r(s,a) + \gamma V(s')) = \arg\max_a Q(s,a). \tag{2.1}$$

The state-value function based action selection without direct knowledge of the action-value function relies on a transition model and the consideration of the whole distribution of successor states. This can be avoided if the action-value function is known and used instead. The crucial problem of value function based methods is the reliance on a search over the whole action space for the action selection, which can be assumed to be very expensive especially for high-dimensional action spaces. This is one of the main reasons why VF methods are difficult for applications where high-dimensional action spaces are involved.

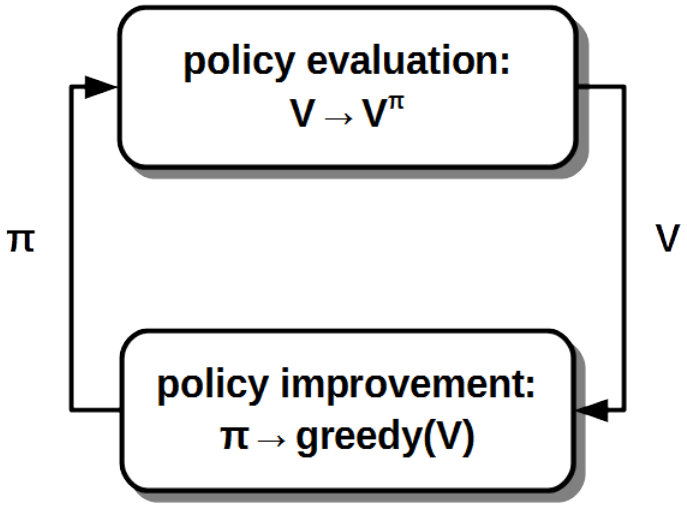

Figure 2.2: Generalized policy iteration: The policy evaluation step and the policy improvement step are repeated until convergence of both the value function and the policy.

Dynamic programming

Dynamic programing (DP)[9] methods are a class of model-based VF methods. They rely on a perfect model of the MDP and involve high computational effort, which limits their practical utility. A notable property of DP methods is that they perform bootstrapping, i.e., value functions estimates are updated on the basis of value function estimates.
Two classical DP methods are policy iteration and value iteration. Policy iteration performs alternately policy evaluation up to convergence of the value function for the current policy and a greedy policy improvement afterwards. The update rule for the value function in value iteration works similar as in policy iteration but truncates the policy evaluation to a single update step. In value iteration both the policy evaluation step and policy improvement step can be summarized to a single update of the value function that is equivalent to the Bellman optimality equation. After the (near-)optimal value function has been determined in value iteration the (near-)optimal policy can be inferred from it by a single greedy action selection for all states.

Input: arbitrary initialized value function V and policy π
Output: (near-)optimal value function V and policy π
repeat
  // Policy Evaluation
  repeat
    forall $s \in S$ **do**
      $a = \pi(s)$
      // Bellman equation
      $V(s) := \sum_{s'} T(s,a,s')(r(s,a) + \gamma V(s'))$
  until *V converged*
  // Policy Improvement
  forall $s \in S$ **do**
    $\pi(s) := \arg\max_a \sum_{s'} T(s,a,s')(r(s,a) + \gamma V(s'))$
until π *is stable*

Algorithm 1: Policy iteration

Input: arbitrary initialized value function $V(s)$ and policy π
Output: (near-)optimal value function $V(s)$ and policy π
repeat
  forall $s \in S$ **do**
    // Bellman optimality equation
    $V(s) := \max_a \sum_{s'} T(s,a,s')(r(s,a) + \gamma V(s'))$
until *V converged*
$\pi(s) := \arg\max_a \sum_{s'} T(s,a,s')(r(s,a) + \gamma V(s'))$

Algorithm 2: Value iteration

Monte-Carlo methods

Monte Carlo (MC) methods estimate value functions purely by returns produced by direct interaction with a (simulated) environment. Unlike DP methods, for a single value function update only one transition sample is generated instead of the whole transition distribution. In comparison to DP methods this makes MC methods computationally much more efficient and allows for a natural combination of interacting with the environment and learning. MC methods can be treated as model-free RL methods if action-value functions are used for the policy improvement. MC methods estimate value functions by averaging the returns of the corresponding long-term samples. An incremental Monte update rule with non-stationarity assumption for updating a state-value function is given by $V(s) := (1-\alpha)V(s) + \alpha R$, where R is the latest return sample received for state s. For a constant and sufficiently small learning rate, this update rule implies convergence to the true value function for some fixed policy. MC methods do not bootstrap as DP methods, since the updates of value function estimates are exclusively based on return samples and not on value function estimates. A problem is the fact that the return is not known before the episode ends. In case of infinite horizons the exact returns are never known. Thus, usual MC approaches are based on episodes with a finite horizon and perform the value function update at the end of an episode. As stated MC methods do not consider the whole transition distribution for a single value function update but only a single transition sample, which implies the problem that certain states or states-action pairs may never be visited if greedy action is used exclusively. This makes it necessary to select actions in a non-greedy manner, e.g., by using ϵ-greedy or softmax for action selection.

Input: arbitrary initialized action-value function Q and exploratory action selection strategy (e.g., ϵ-greedy)
Output: (near-)optimal action-value function Q
repeat
  1. generate an episode using π specified by Q and the action selection strategy
  2. let Z be the set of all state-action pairs (s,a) appearing in the episode
  forall $(s,a) \in Z$ **do**
    $R :=$ return following the first occurrence of (s,a)
    $Q(s,a) := (1-\alpha)Q(s,a) + \alpha R$
until *Q converged*

Algorithm 3: An episodic Monte-Carlo control algorithm for learning a (near-)optimal action-value function Q based on a lookup table.

Temporal difference learning

Temporal difference (TD) learning methods combine ideas from both DP and MC methods. As MC methods, value functions are learned by experience gathered from direct interaction with the (simulated) environment where for each value update of a state (or state-action pair) only a single transition sample is considered instead of the whole distribution of successor states for a certain state as in DP methods. Thus, as MC methods, TD methods rely on non-greedy action selection to produce reliable value function estimates. TD learning can be applied model-free without knowledge of the dynamics of the environment if the action-value function is known and employed. The main difference between MC and TD methods lies in how the value function is updated for given transition samples. TD methods update a value function estimate on the basis of an existing value function estimate, i.e., in contrast to MC they bootstrap like DP. The simplest TD method, TD(0), updates the state-value function according to

$$
\begin{aligned}
V(s) &:= (1-\alpha)V(s) + \alpha(r(s,a) + \gamma V(s')) \\
&= V(s) + \alpha(r(s,a) + \gamma V(s') - V(s)) = V(s) + \alpha\delta
\end{aligned} \tag{2.2}
$$

The increment $r(s,a) + \gamma V(s') - V(s) = \delta$ is called temporal difference. The value of δ can be computed in the next time step. Thus and in contrast to MC methods, TD(0) can update value functions easily in an online fashion. For the discrete lookup table case, i.e., where have discrete state and action spaces and where we store values for the states or state-action pairs in a lookup table, it can be mathematically shown that TD methods in the mean are sound for a sufficiently small learning rate α and converge to the true value function of a corresponding fixed policy. TD methods are usually assumed to converge faster than MC methods due to bootstrapping. However, bootstrapping may also cause instability problems in the learning process if function approximation instead of lookup tables is used for representing the value function.
As in MC methods, TD methods can be used model-free if an action-value function is used. The two most popular TD methods for learning action-value functions are SARSA[19] and Q-learning[20]. SARSA learns the action-value functions according to

$$
\begin{aligned}
Q(s,a) &:= Q(s,a) + \alpha(r(s,a) + \gamma Q(s',a') - Q(s,a)) \\
&= Q(s,a) + \alpha\delta
\end{aligned} \tag{2.3}
$$

and for Q-learning the update is given by

$$
\begin{aligned}
Q(s,a) &:= Q(s,a) + \alpha(r(s,a) + \gamma \max_{a'} Q(s',a') - Q(s,a)) \\
&= Q(s,a) + \alpha\delta.
\end{aligned} \tag{2.4}
$$

with $a' \in A$ as the action in the next step and δ as the temporal difference error. SARSA learns the value function on-policy, i.e., it uses only return estimates for the actions that were selected by the policy. In contrast, Q-learning learns off-policy and performs updates on the base of return estimates of actions resulting from greedy action selection. In practice the advantages of off-policy learning are limited. Off-policy algorithms are often much more complicated and do not give significant performance gains in comparison to on-policy learning. Usually off-policy methods get outperformed by on-policy methods. However, off-policy methods like Q-learning are still interesting from a theoretical perspective as they enable convergence proofs.

A problem with the described TD methods is that the value for only one state or state-action pair is affected by an update which implies a slow convergence. In order to improve convergence and to affect more states and state-action pairs by updates TD methods can be generalized by using eligibility traces. Eligibility traces mark recently visited states and decay exponentially over time with a certain rate. The greater the eligibility trace of a state the more this state is assigned credit for an TD error. Eligibility traces are a basic mechanism to solve the temporal credit assignment problem emerging from the backward view. Moreover, they can be easily implemented for the lookup table case. This leads to the TD(λ) algorithm method which generalizes TD(0) for state-value functions. A popular application of this algorithm was the learning of a strong backgammon player based on an artificial neural network that was trained with TD(λ) without having any sort of prior knowledge about the game[21, 22]. For the action-value function based TD methods given by SARSA(0) and Q-learning(0) the corresponding generalizations are SARSA(λ) and Q-learning(λ). In

the lookup table case these methods maintain an additional eligibility trace variable for each state or state-action pair.

Input: arbitrary initialized action-value function Q, exploratory action selection strategy (e.g., ϵ-greedy), discount factorγ, decay factor λ
Output: (near-)optimal action-value function Q
repeat
1. initialize e-traces: $e(s,a) := 0$
2. start an episode in some initial state $s := s_0$ and action $a := a_0$
3.**repeat**
1. execute action a and observe the received reward r and the next state s'
2. select an action a' in the current state according to the current action-value function and the action selection strategy
3. compute the temporal difference: $\delta := r(s,a) + \gamma Q(s',a') - Q(s,a)$
4. $e(s,a) := e(s,a) + 1$
5. **forall** $(s,a) \in S \times A$ **do**
1.$Q(s,a) := Q(s,a) + \alpha\delta e(s,a)$
2. $e(s,a) := \gamma\lambda(s,a)$
6. $s := s'$
7. $a := a'$

until *episode terminated*
until Q *converged*

Algorithm 4: Online SARSA(λ) control algorithm for learning a (near-)optimal action-value function Q based on a lookup table.

Action selection strategies for MC and TD methods

MC and TD consider consider only single transition samples for policy evaluation instead of the whole distribution of successor states as DP methods do. This implies that the efficiency of policy evaluation, i.e., the production of reliable value function estimates, is affected by the action selection. In fact greedy action selection as in equation 2.1 can prevent the encountering of states and state-action pairs and therefore destroy the policy evaluation process.
A solution to this problem is the usage of action selection strategies which employ stochastic exploration rather than pure exploitation as a greedy action selection does. By using such exploratory strategies even actions that are suboptimal according to the current value function estimates may be selected with a certain probability such that the value function estimates can become more reliable since more states and state-action pairs are encountered and more knowledge is generated. In general it is not clear how a trade-off between exploration and greedy exploitation can be realized in an optimal way. Two popular and simple strategies, namely ϵ-greedy and softmax are an answer to the exploration-exploitation dilemma and are explained in the following. These strategies are also heavily used for basic multi-armed bandit problems[16, 17, 18].

ϵ-greedy: ϵ-greedy selects with probability ϵ a random action according to the uniform distribution which corresponds to exploration and otherwise the action with the highest Q-value in the current state is taken, i.e.,

$$\pi(a,s) = \begin{cases} \text{select random action } a \in A \text{ according to uniform distribution, if } \xi < \epsilon \\ \arg\max_a Q(s,a) \text{ otherwise} \end{cases}$$

where $0 \leq \xi \leq 1$ is a uniform random number. A property which can be seen as disadvantage is the fact that all actions are selected with the same probability in case of exploratory selection, regardless of their Q-values. Thus, even very bad actions can be selected in exploration phases. For $\epsilon = 1$ all actions are always selected according ro uniform distribution whereas for $\epsilon = 0$ action selection is equivalent to greedy action selection.

Softmax: If using softmax actions are selected according to ,i.e.,

$$\pi(s,a) = \frac{exp(\beta Q(s,a))}{\sum_{a' \in A} exp(\beta Q(s,a'))}$$

with $\beta > 0$ as the inverse temperature. Similarly to ϵ-greedy for $\beta \to 0$ all actions have the same probability whereas for $\beta \to \infty$ softmax is equivalent to greedy action selection. In contrast to ϵ-greedy an action with a higher Q-value is always more likely to be selected than an action with a smaller Q-value.

2.3.5 Actor-critic methods

Actor-critic methods employ an actor and a critic. The actor represents the actual policy that corresponds to the skill of the agent. The critic represents a value function, typically a state-value function, that corresponds to the knowledge of the agent. The critic uses its gathered knowledge to judge the actions executed by the actor on the base of TD errors δ. The actor tries to optimize its policy based on these TD errors. A picture of this principle is shown in figure 2.3.
An advantage of actor-critic methods is that the selection of an action can be done efficiently by the actor. Thus, no computationally expensive consideration of the whole action space is needed to select an action as in pure VF methods, also called critic-only methods. On the other side, the critic contains knowledge to train the actor efficiently with TD-errors. TD errors have in general a much lower variance than the rewards or returns provided directly by the MDP and are at least as meaningful as these to train the actor properly. High variances of rewards and returns that are used to train the actor make the training of the actor extremely difficult and lead to a very poor convergence in general. In particular, this turns out to be a huge problem of methods that do not employ a value function, also called actor-only methods, such as likelihood ratio methods. Thus, actor-critic methods can be seen as reinforcement learning methods that combine the advantages of both actor-only methods and critic-only methods.

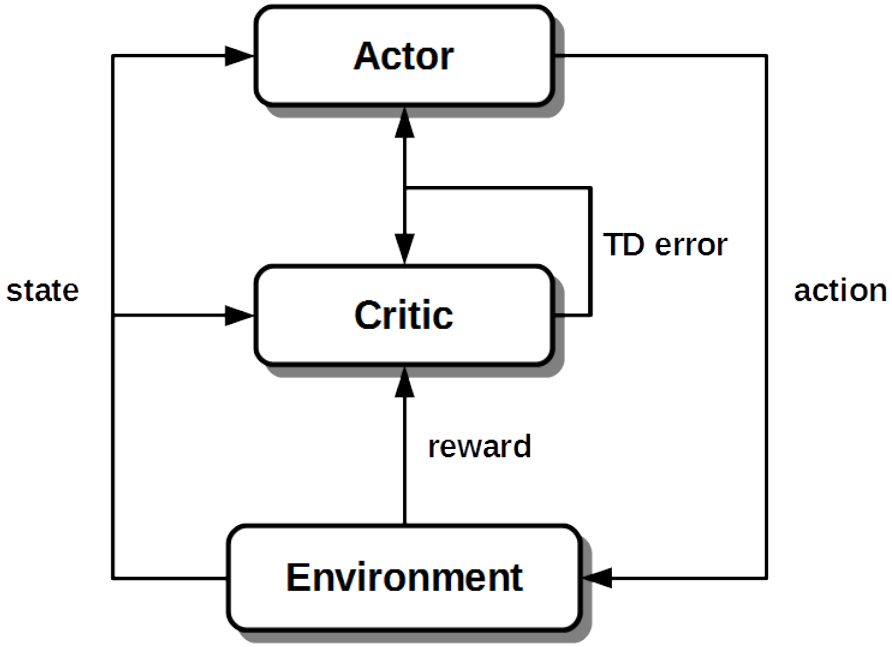

Figure 2.3: Actor-critic methods use a separate actor and critic. The critic computes TD errors and uses these to update the value function. The actor uses the TD errors to train the policy.

3 Foundations of artificial neural networks

In this chapter artifical neural networks(ANNs)[23, 24, 25, 26, 27, 28] are introduced. ANNs are a biological inspired models of computation and an abstraction of the working principles of the nervous system of animals, i.e., how information processing is done. ANNs provide a very powerful framework for function approximation that is heavily used in machine learning. An ANN is a composition of many interacting simple processing units, called neurons. The global information processing in neural networks is the result of the local information processing of all neurons which is done simultaneously and independently. Therefore neural networks are systems working in a parallel and distributed manner.

3.1 Distributed models of computation

A model of computation is defined by a set of rules that specify how a single computation step takes place, i.e., how an output or successor state is produced for a given input or current state. Examples for models of computation are ANNs, Turing machines (TMs) and cellular automata (CAs)[29, 30, 31].
ANNs and cellular automata are both models of computation provided by nature. ANNs are inspired from the nervous system of animals. According to [29] all processes occurring in the universe are the result of the computation of a single discrete cellular automaton. This implies that each program that runs on a computer also runs on a discrete cellular automaton, i.e., our universe. In contrast to TMs, natural models of computation given by ANNs an CAs have in common that they work highly efficiently. The powerful capabilities of animal brains, in particular the human brain, ANNs were inspired from are well-known. If a CA would not work efficiently no program would, since each executed program always runs indirectly on a cellular automaton given by our universe.
A way to explain why ANNs and CAs work efficiently is their spatially distributed and parallel working principle, i.e., they are a composition of multiple parallel working processing units interacting with each other. At each time step each processing units receives an input from the environment or other processing units and produces an output which is done independently for each processing unit. In case of ANNs these processing units are neurons and in case of CAs these processing units are cells. A difference between both computation models is how the processing units interact with each other. In ANNs the input of a neuron receives at some certain point can depend on the output of all neurons, but has a compressed content of information in the sense that the output of these neurons cannot be inferred from the input. In CAs the input of a cell does only depend on the output of the neighbored cells but is not compressed as in ANNs. In result, the input and output spaces of single processing units in ANNs and CAs are simple and atomic as these processing units itself.

3.2 Formalizing ANNs

An artificial neural network can be formalized by a set of neurons N, the number of synapses $c_{ij} \in \{0,1\}$ from neuron $i \in N$ to neuron $j \in N$ and corresponding weights $w_{ij} \in \mathbf{R}$ of these synapses. Each neuron $i \in N$ has an activation state $\dot{s}_i$. If a neuron is an input neuron its activation state is induced by the environment. A neuron j that is no input neuron receives input signals from other presynaptic neurons $i \in N_j$ with $I_j = \{i \in N : c_{ij} = 1\}$ and produces an output signal $\dot{s}_j$ depending on these and some activation function f. In most neural network models this function primitive is defined by $\dot{s}_j = f(u_j) = f(\sum_{i \in I_j} w_{ij}\dot{s}_i)$, with $u_j = \sum_{i \in I_j} w_{ij}\dot{s}_i$ as membrane potential. f is some nonlinear activation function used by all neurons in the network and $\dot{s}_j$ is the resulting activation state of neuron j which is equivalent to the signal sent from this neuron. The activation state $\dot{s}_j$ of a neuron can be continuous or binary. Continuous activation states $\dot{s}_j \in \mathbf{R}$ are used in classical MLPs and model the spiking frequency of a neuron. Binary activation states $\dot{s}_j \in \{0,1\}$ of neurons are used in spiking neural networks[32, 33] and model whether the neuron produced a spike or not. Different types of ANNs differ mainly in the choice of the activation function and the network topology.

3.3 Activation functions

In this section important and popular types of activation functions for ANNs are presented.

3.3.1 Heaviside step function

The Heaviside step function is given by $\dot{s}_j = f(u_j) = \begin{cases} 0, & u_j < 0 \\ \frac{1}{2}, & u_j = 0 \\ 1, & u_j > 0 \end{cases}$. It is a binary threshold function and is for instance used in single-layer perceptrons.

3.3.2 Sigmoid functions

Sigmoid functions refers to a class of "S"-shaped functions. Two popular sigmoid functions used in the field of ANNs are the logistic function given by $\dot{s}_j = f(u_j) = \frac{1}{1+exp(-\beta u_j)}$ and the hyperbolic tangent given by $\dot{s}_j = f(u_j) = \tanh(\beta u_j) = \frac{exp(\beta u_j)-exp(-\beta u_j)}{exp(\beta u_j)+exp(-\beta u_j)}$, where $\beta > 0$ is the inverse temperature. The Heaviside step function is a special case of the logistic function for $\beta = \infty$.

3.3.3 Population based activation functions

Population based activation functions are generalizations of standard activation functions. They are applied to populations of neurons and compute the activation state of each neuron of a population in a dependent way, that is the state of each neuron in the population depends on the membrane potentials of all neurons in the population, i.e.,$\underline{\dot{s}}_p = f(\underline{u}_p)$ with $\underline{\dot{s}}_p$ as the activity of the neurons in population p and $\underline{u}_p$ as their corresponding potentials. Thus, the neuron populations form atomic computation primitives.

Softmax

Softmax is a population based activation function and a generalization of the logistic function. It is given by $\dot{s}_{p,j} = f_j(\underline{u}_p) = \frac{exp(\beta u_{p,j})}{\sum_{l=1}^{N_p} exp(\beta u_{p,l})}$ where N_p denotes the number of neurons in population p, $\underline{u}_p$ are the neuron potentials of the neurons in population p, $\dot{s}_{p,j}$ denotes the activation state and $u_{p,j}$ denotes the membrane potential of neuron j, $1 \leq j \leq N_p$, in population p and $\beta > 0$ is the inverse temperature. The activation states of all neurons in p depend on all membran potentials $\underline{u}_p$ of the neurons in that population.
Softmax activation functions are often used for the output layer in classification tasks, where the resulting neuron states can be interpreted as class probabilities. A stochastic version of this function is often used in stochastic winner-take-all circuits. The softmax function is also used in reinforcement learning for exploratory action selection. Thus, the softmax function can be seen as a natural link between the theories of neural networks and reinforcement learning.

Winner-take-all dynamics

Winner-take-all(WTA) circuits[34, 35] are neuron populations that use an activation function such that exactly one neuron j in population p, $1 \leq j \leq N_p$, has the activation state $\dot{s}_{p,j} = 1$ whereas all other neurons l of that population, $j \neq l \wedge 1 \leq j \leq N_p$, have the activation state $\dot{s}_{p,l} = 0$. Thus, the neurons compete for activity and the winning neuron inhibits all other neurons. The activation function of WTA circuits can be deterministic or stochastic. A common activation function for stochastic WTA circuits is the stochastic softmax function with $p(\dot{s}_{p,j} = 1|\underline{u}_p) = f_j(\underline{u}_p) = \frac{exp(\beta u_{p,j})}{\sum_{l=1}^{N_p} exp(\beta u_{p,l})}$. If $\beta = \infty$ holds, the stochastic softmax functions becomes deterministic and chooses the neuron with the highest membrane potential as winner.

3.4 Network topologies

Topologies of ANNs define how the neurons are connected and how the flow of information processing takes place. We can distinguish between feed-forward and recurrent networks. Feed-forward networks consist of multiple layers of neurons which includes a layer of input units, a layer of output units and possibly additional layers of hidden units between them. All neurons of a layer can only have outgoing synapses to neurons of the next layer. In a single processing cycle input signals are given to the input layer and are processed from there layer wise to the output layer. Examples

for feed-forward networks are single-layer perceptrons and multi-layer perceptrons. Recurrent ANNs allow backward connections such that the state of a neuron can serve as input for previous layers in the next processing cycle. Popular examples for recurrent networks are Boltzmann machines[36] and long short-term memories[37].

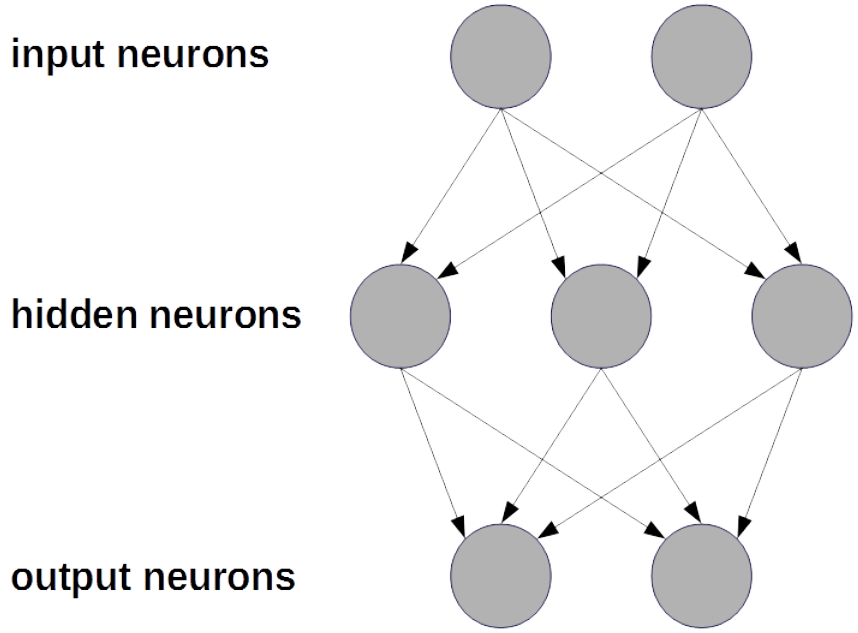

Figure 3.1: A multi-layer feed-forward network with three layers. The neurons are depicted as circles and the synaptic connections are depicted as directed arrows. The processing flow corresponds to the direction of the arrows.

3.5 Expressive power of ANNs

Given an ANN with fixed topology and fixed activation functions, the expressive power of the ANN is defined by the set of all functions it can compute for a proper choice of synaptic weights. A very famous example that shows up the limitations of the expressive power of ANNs is the incapability of perceptrons with two inputs and one output neuron to compute the XOR function. A perceptron is a linear classifier and the simplest feed-forward network and consist only of an input layer and an output layer and uses the Heaviside step function as activation function.
A solution to the XOR problem can be achieved by adding one or more layers of hidden units to the perceptron. This leads to multi-layer perceptrons (MLP). An MLP is a feed-forward network and a generalisation of the perceptron with additional hidden layers and arbitrary activation function. In particularly, MLPs with at least one additional hidden layer are able to compute the XOR function. According to the universal approximation theorem a feed-forward network with a single hidden layer containing a sufficient number of hidden neurons can approximate each function with finitely many discontinuities with arbitrary accuracy if the hidden neurons use a non-linear activation function[38].

3.6 Training of ANNs

The objective of an ANN is the production of desired outputs for given inputs. There are different factors that influence this behavior, i.e., the function realized by the ANN. These factors include the network topology, the used activation function and the synaptic weights. All of these can be set a priori by making use of expert knowledge. However, setting the weights properly in manual manner by the making use of expert knowledge can be seen as intractable in general. Instead, a learning method needs to be employed to adjust the synaptic weights. Trivially, an ANN can only learn functions it can compute, i.e., that are covered by its expressive power.
The training of ANNs can be done in the context of supervised learning and reinforcement learning. In supervised learning the neural network is provided a desired output (teaching signal) for a given input. The goal is to adjust the weights such that the distance (e.g., squared euclidean distances) between the produced and desired outputs of the network for given inputs is minimized. In reinforcement learning the neural network is given a scalar feedback, i.e., a reward, after it

produced an output for a corresponding input. This feedback expresses how advantageous the behavior of the ANN was in this certain situation. In reinforcement learning the weights need to be adjusted such that the expected return, i.e., the long-term reward, is maximized. In contrast to supervised learning no teaching signal is given which gives information about how a desirable output looks like, i.e., an output with a high return expectation.

3.6.1 Hebbian learning

Hebbian learning methods refer to an important class of white-box methods to train the synaptic weights of an ANN. This class is based on the original Hebbian learning rule [39] given by $\Delta w_{ij} = \alpha \dot{s}_i \dot{s}_j$. This learning rule qualitatively means the corresponding synaptic weight w_{ij} increases if the pre- and postsynaptic neurons are active at the same time. Furthermore, if $\dot{s}_i = 0$ or $\dot{s}_j = 0$ holds the synaptic weight cannot change, i.e., $\Delta w_{ij} = 0$. A more general framework of Hebbian learning rules is given by $\Delta w_{ij} = \alpha \dot{s}_i g(\dot{s}_j, \Theta)$, where Θ denotes additional specific parameters which may include all synaptic weights of the ANN, a provided reward in a reinforcement learning setting or a teaching signal that corresponds to the desired output in a supervised learning setting. In these cases the weight w_{ij} cannot change if $\dot{s}_i = 0$ holds for the presynaptic state, which is plausible since this weight has no impact on the postsynaptic membrane potential or neuron state if $\dot{s}_i = 0$ holds. This property of almost all Hebbian learning rules can explain why in general Hebbian learning methods like backpropagation outperform black-box optimization methods like evolutionary algorithms that would optimize the weights without taking care of the neuron states.

3.6.2 Backpropagation

Backpropagation is a very popular algorithm for training MLPs which uses a deterministic activation function for all neurons and is mainly used in a supervised learning setting. Backpropagation is based on gradient descent with analytic gradients and results in a Hebbian learning rule. It is a generalization of the delta rule for single-layer networks and can train multi-layer feed-forward networks with additional hidden layers.
In the following the update rule for backpropagation is derived where we assume a network state resulting from a given input. We make use of an error measure E expressing how well the output state of the network matches the provided teaching signal. The goal is to minimize this error. According to gradient descent backpropagation updates the weights by following the negative gradient of the error E with respect to the weights, i.e.,

$$\Delta w_{ij} = -\alpha \frac{\partial E}{\partial w_{ij}} = -\alpha \frac{\partial E}{\partial \dot{s}_j} \frac{\partial \dot{s}_j}{\partial u_j} \frac{\partial u_j}{\partial w_{ij}} \tag{3.1}$$

For $\frac{\partial u_j}{\partial w_{ij}}$ in equation 3.1 we have

$$\frac{\partial u_j}{\partial w_{ij}} = \frac{\partial \sum\limits_{i \in I_j} w_{ij} \dot{s}_i}{\partial w_{ij}} = \dot{s}_i. \tag{3.2}$$

In order to apply backpropagation the activation function needs to be differentiable such that $\partial \dot{s}_j / \partial u_j$ in equation 3.1 can be calculated(e.g., by using the logistic function or softmax). If j is an output neuron the evaluation of $\partial E / \partial \dot{s}_j$ in equation 3.1 can be done immediately depending on the used error function applied to the produced outputs and the teaching signals. Assuming that neuron i is not an output neuron, i.e., an inner neuron which has outgoing synapses and denoting the postsynaptic neurons of these synapses with $O_i = \{j \in N : c_{ij} = 1\}$ we can write $\partial E / \partial \dot{s}_j$ as

$$\frac{\partial E}{\partial \dot{s}_j} = \sum_{k \in O_j} \frac{\partial E}{\partial u_k} \frac{\partial u_k}{\partial \dot{s}_j} = \sum_{k \in O_j} \frac{\partial E}{\partial u_k} \frac{\partial \sum_{i \in I_k} w_{ik} \dot{s}_i}{\partial \dot{s}_j} = \sum_{k \in O_j} \frac{\partial E}{\partial u_k} w_{jk}.$$

Thus, we have

$$\frac{\partial E}{\partial \dot{s}_j} = \begin{cases} \frac{\partial E}{\partial \dot{s}_j}, & \text{if neuron j is an output neuron} \\ \sum\limits_{k \in O_j} \frac{\partial E}{\partial u_k} w_{jk}, & \text{if neuron j is an inner neuron.} \end{cases} \tag{3.3}$$

By making use of the abbreviation $\delta_j = \frac{\partial E}{\partial u_j} = \frac{\partial \dot{s}_j}{\partial u_j}\frac{\partial E}{\partial \dot{s}_j}$ and equation 3.4 we have

$$\delta_j = \begin{cases} \frac{\partial \dot{s}_j}{\partial u_j}\frac{\partial E}{\partial \dot{s}_j}, & \text{if neuron j is an output neuron} \\ \frac{\partial \dot{s}_j}{\partial u_j}\sum_{k \in O_j} \delta_k w_{jk}, & \text{if neuron j is an inner neuron.} \end{cases} \tag{3.4}$$

Plugging equations 3.2 and 3.4 into equation 3.1 with case distinction leads to

$$\Delta w_{ij} = -\alpha \frac{\partial E}{\partial w_{ij}} = \begin{cases} -\alpha \dot{s}_i \frac{\partial \dot{s}_j}{\partial u_j}\frac{\partial E}{\partial \dot{s}_j}, & \text{if neuron j is an output neuron} \\ -\alpha \dot{s}_i \frac{\partial \dot{s}_j}{\partial u_j}\sum_{k \in O_j} \delta_k w_{jk}, & \text{if neuron j is an inner neuron} \end{cases} \tag{3.5}$$

. In order to compute $\partial E / \partial w_{ij}$ for all weights of the neural network we have to determine δ_j for all neurons j of the network. Since for inner neurons δ_j depends on δ_k of neurons in a lower layer, the δ_j values must be determined recursively and backwards beginning from the output layer. A common choice for the the error function is given by $E = \frac{1}{2}\sum_{j \in O}(t_j - \dot{s}_j)^2$, where O denotes the output neurons of the neural network and t_j is the teaching signal for output neuron j. This error function leads to $\partial E / \partial \dot{s}_j = -(t_j - \dot{s}_j)$.
For hidden neurons, if they are used, sigmoid functions are often used as activation functions. For these activation functions it usually holds that $\partial \dot{s}_j / \partial u_j = \partial f(u_j)/\partial u_j \in [0, 1]$ which means that the absolute value of δ_j and thus of Δw_{ij} decreases exponentially with a layer's height such that there is no effective weight change in higher layers. This problem is also known as vanishing gradient problem [40] and explains why training deep ANNs with backpropagation has crucial limitations.
Furthermore, as a gradient descent based method, backpropagation is prone to get stuck in local optima if the error function is non-convex which is usually the case for multi-layer perceptrons with at least one hidden layer.

4 A distributed policy based on spiking neural networks

In the following we will motivate and explain a distributed type of policy $\pi: S \to A$ based on stochastic spiking neural networks (SNN) that can be trained in a reinforcement learning setting. Here we will always assume discrete state and action spaces. This is not necessarily a disadvantage, since we can always discretize continuous state and action spaces with arbitrary accuracy. Furthermore, the most advanced processors are working binary without exception. This includes artificial processors as well as the human brain, where neurons are communicating via binary spikes and not continuous signals.
Furthermore, we assume that states are completely observable, that is at each time step the current observation contains all information of the current global state. Thus, we will not discriminate between states and observations.

To motivate our SNN based policy we will start with a discrete basic type of parameterized stochastic policy inspired by VF methods, where we have a discrete state-action space and exactly one real-valued parameter $\theta(s,a)$ for each state-action (s,a) pair and vice versa, which corresponds to a lookup table parameterization. Assuming that we would apply value function based learning methods to train these parameters they would form a Q-table with $Q(s,a;\theta) = \theta(s,a)$. Such a parameterization is sufficient to encode an arbitrary action selection distribution for each state. However, this type of policy suffers strongly from the curse of dimensionality, since high-dimensional state and action spaces will lead to an explosion of the number of states and actions and consequently to an explosion of the number of parameters given by $|S||A|$. Maintaining the number of parameters as well as realizing the dynamics, i.e., the choosing of an action, rapidly becomes intractable for any computation scheme with an increasing number of dimensions of the state and action spaces. Furthermore, since the frequency of encountering the same state-action pair decreases we can also expect a poor learning speed in higher dimensions if using the described type of policy.

A way out of this dilemma caused by the curse of dimensionality can be derived by taking inspirations from natural computation models, i.e., neural networks and cellular automata. A crucial basic property of these computation models is their spatially distributed working principle, i.e., they can be decomposed into multiple parallel working processing units interacting with each other, where each processing units computes its output for a given input independently from all other processing units. A key intuition is to make the policy consistent with this distributed working principle of natural computation models that can be assumed to work efficiently.
For doing so we decompose the action space A into N_A partial action spaces $A_1, ..., A_{N_A}$ with $A = A_1 \times ... \times A_{N_A}$ and define the policy π to select an action $a = (a_1, ..., a_{N_A}) \in A = A_1 \times ... \times A_{N_A}$ by selecting partial actions $a_p \in A_p$ independently from each other according to partial policies $\pi_p: S \to A_p$, $1 \leq p \leq N_A$ such that $\pi(s, a = (a_1, ..., a_{N_A}; \theta)) = \prod_{p=1}^{N_A} \pi_p(s, a_p; \theta_p))$. In the following we write $\pi(s, a_p; \theta)$ for $\pi_p(s, a_p; \theta_p)$ as short-hand notation. The size $|A_p|$ of the partial action spaces is significantly smaller than the size the whole action space given by $|A| = \prod_{p=1}^{N_A} |A_p|$. The selection of the partial actions can be done in parallel, which means that it can be realized highly efficiently assuming that in general the computational effort to sample from a space according to some underlying distribution scales linearly with its size. Since the selection of the partial actions is not correlated, parameters that would encode such a correlation can be dropped. The parameterization is generalized such that we have one parameter for each pair of a state $s \in S$ and a partial action $a_p \in A_p$, $1 \leq p \leq N_A$. The number of parameters becomes $N_\theta = |S| \sum_{p=1}^{N_A} |A_p|$, which is a significant reduction in comparison to the case of non-distributed action selection where the number of parameters is given by $N_\theta = |S||A| = |S| \prod_{p=1}^{N_A} |A_p|$. Such a parameter reduction correlates with an improved learning behavior due to faster generalizing.

Another result of such distributed action selection is that the rewards received for selecting a certain partial action for a fixed state according to a partial policy becomes more noisy due to the missing correlation with the selection of the remaining partial actions and furthermore parameter changes due to learning dynamics also cause non-stationarity of the rewards in this case. There are no guarantees to converge to a global optimum even if the rewards are constant for all state-action pairs. Furthermore, in the context of VF methods it is not possible to encode exact Q-values $Q(s,a)$ for a

global joint action $a = (a_1, ..., a_{N_A})$ in a certain state s, if partial policies with distributed parameterizations and dynamics are used.
With certain advantages and disadvantages the distributed approach described so far is able to deal with high-dimensional actions. However, high-dimensional states are still a problem, as number the of parameters given by $N_\theta = |S| \prod_{p=1}^{N_A} |A_p|$ reveals. In order to deal with high-dimensional state spaces we will again take inspirations from natural distributed computation models. For all of these the space of inputs provided to each atomic processing unit has a limited complexity that is significantly smaller than the complexity of the whole system. In case of cellular automata, the input space of a cell is only specified by the joint complexity of state spaces of the cells neighbored to this cell and in case of neural networks the input space of a neuron is a real-valued scalar resulting from the linear combinations of signals sent from other neurons.

4.1 Realizing a distributed policy with an SNN

For our approach we will focus on neural networks as they appear to be a quite practical computation model as it has been shown in numerous practical applications and furthermore they are a parameterized. In particularly we generalize our approach by decomposing the state space in partial state spaces, i.e., $S = S_1 \times ... \times S_{N_S}$ and parameterize the policy such that we have now exactly one parameter for each pair of a partial state and a partial action (s_o, a_p) with $s_o \in S_o$, $1 \leq o \leq N_S$, $a_p \in A_p$, $1 \leq p \leq N_A$. For a given global state $s = (s_1, ..., s_{N_S})$ each partial action $a_p \in A_p$, $1 \leq p \leq N_A$, is now assigned several parameters given by $\theta(s, a_p) = \{\theta(s_o, a_p) : 1 \leq o \leq N_S\}$. According to inspirations provided by neural networks the selection distribution over all partial actions $a_p \in A_p$ in some partial action space A_p, $1 \leq p \leq N_A$, according to the partial policy π_p for a given state s is now specified by potentials given by $u(s, a_p) = \sum_{\theta \in \theta(s,a_p)} \theta$, which is an admissible distributed generalization for distributed states, since $u(s, a_p) = \theta(s, a_p)$ if s is non-distributed, i.e., $N_S = 1$.
Following the formalism of neural networks, the described distributed policy can be expressed equivalently by populations of spiking neurons[1, 2] contained in two different layers, a state and an action layer, that are fully connected as shown in figure 4.1, i.e., there is exactly one synapse with the weight θ between two neurons i and j, if i is a state neuron and j is an action neuron, otherwise there is no synapse between these neurons. Each partial state space S_o, $1 \leq o \leq N_S$, is represented by a neuron population in the state layer and each partial action space A_p, $1 \leq p \leq N_A$, is represented by a neuron population in the action layer.
Furthermore, each partial state $s_{o,k} \in S_o$, $1 \leq k \leq |S_o|$, in some partial state space S_o, $1 \leq o \leq N_S$, is represented by a neuron i, $1 \leq i \leq \bar{N}_S = \sum_{o=1}^{N_S} |S_o|$, in the state layer with activation state $\dot{s}_i$ and each partial action $a_{p,l} \in A_p$,$1 \leq l \leq |A_p|$, in some partial action space A_p, $1 \leq p \leq N_A$, is represented by a neuron j, $1 \leq j \leq \bar{N}_A = \sum_{p=1}^{N_A} |A_p|$, in the action layer with activation state $\dot{a}_j$. To encode a certain partial state or partial action its corresponding neuron has an activity of 1 and the activity of all other neurons in the population is set to 0, i.e., $\dot{s}_i, \dot{a}_j \in \{0, 1\}$, since partial states or partial actions in the same partial state space or partial action space are mutually exclusive.
In the following $\underline{\dot{s}}$ denotes the the activities of all state neurons encoding the global state s, $\underline{\dot{s}}_o$ denotes the activities of all neurons in the state population o, $1 \leq o \leq N_S$, encoding the partial state s_o, $\underline{\dot{a}}$ denotes the activities of all action neurons encoding the global action a and $\underline{\dot{a}}_p$ denotes the activities of all neurons in the action population p, $1 \leq p \leq N_A$, encoding the partial action a_p. Since exactly one neuron spikes in each action population they form winner-take-all(WTA) circuits where the activation function realizes the stochastic dynamics of the corresponding partial policy, that is the choosing of an partial action by activating the corresponding neuron and inhibiting the remaining ones of the population. The activation function $f_l(\underline{u}_p = (u_{p,1}, ..., u_{p,|A_p|})) = \pi(\underline{\dot{s}}, \dot{a}_{p,l} = 1; \theta) = \pi(s, a_p = a_{p,l}; \theta) = \rho_{p,l}$ activates neuron $\dot{a}_{p,l}$ in the action population p with probability $\rho_{p,l}$ that depends on the potentials of the neurons in this population defining a WTA circuit where the neuron potential of some neuron j is given by $u_j = \sum_{i=1}^{\bar{N}_S} \dot{s}_i \theta_{ij}$ with θ_{ij} as the synaptic weight between state neuron i and action neuron j.
The number of parameters is now given by $N_\theta = \sum_{o=1}^{N_S} |S_o| \sum_{p=1}^{N_A} |A_p|$ in comparison to the non-distributed case of parameterization given by $N_\theta = |S||A| = \prod_{o=1}^{N_S} |S_o| \prod_{p=1}^{N_A} |A_p|$. Thus, similar to distributed actions, distributed state parameterizations enable to deal with high-dimensional states and improve the generalization due to a further reduction of the number of parameters. However, since parameters are removed that express how the partial policies react to states, it may also result in a significant loss of expressive power similar to the incapability of single-layer perceptrons to learn the XOR function.

4.2 Training the SNN in an one-step MDP setting

In the following we will show different learning methods to train our SNN in a one-step MDP setting where the episode ends after one step. Thus, only a single state-action pair (s, a) is encountered and only one reward r is received during the episode which equals the return, i.e., $R = r$. The resulting learning rules can also be applied straight-forward in a multi-step MDP setting with $\gamma = 0$.

4.2.1 Black-box methods

Due to the flexibility of black-box methods they can be applied directly to the parameters θ of the SNN without making any further assumptions about aspects like the used activation function or the activity of the neurons that occurred during the episode.

4.2.2 Likelihood ratio methods

The SNN can be trained with likelihood ratio approaches, if the used activation function is differentiable. For likelihood ratio methods we assume that softmax is always used as activation function. REINFORCE leads to a simple update rule for a one-step MDP given by:

$$\Delta\theta_{ij} = \alpha \frac{\partial \ln \pi(s,a;\theta)}{\partial \theta_{ij}} R \tag{4.1}$$

In the following we write the potential of the active neuron that won the WTA competition in action population p as $\sum_{l=1}^{|A_p|} \dot{a}_{p,l} u_{p,l}$ and the population containing neuron j is denoted by $p(j)$. By making use of softmax as activation function $\frac{\partial \ln \pi(s,a;\theta)}{\partial \theta_{ij}}$ results in

$$\begin{aligned}
&\frac{\partial \ln \pi(s,a;\theta)}{\partial \theta_{ij}} = \frac{\partial \ln \pi(\underline{\dot{s}},\underline{\dot{a}};\theta)}{\partial \theta_{ij}} = \frac{\partial \ln \prod_{p=1}^{N_A} \pi(\underline{\dot{s}},\underline{\dot{a}}_p;\theta)}{\partial \theta_{ij}} = \frac{\partial}{\partial \theta_{ij}} \ln \prod_{p=1}^{N_A} \frac{exp(\beta \sum_{l=1}^{|A_p|} \dot{a}_{p,l} u_{p,l})}{\sum_{l=1}^{|A_p|} exp(\beta u_{p,l})} \\
&= \frac{\partial}{\partial \theta_{ij}} \ln \frac{exp(\beta \sum_{p=1}^{N_A} \sum_{l=1}^{|A_p|} \dot{a}_{p,l} u_{p,l})}{\prod_{p=1}^{N_A} \sum_{l=1}^{|A_p|} exp(\beta u_{p,l})} = \frac{\partial}{\partial \theta_{ij}} \ln \frac{exp(\beta \sum_{l=1}^{\tilde{N}_A} \dot{a}_l u_l)}{\prod_{p=1}^{N_A} \sum_{l=1}^{|A_p|} exp(\beta u_{p,l})} = \frac{\partial}{\partial \theta_{ij}} (\beta \sum_{p=1}^{\tilde{N}_A} \dot{a}_l u_l - \sum_{p=1}^{N_A} \ln \sum_{l=1}^{|A_p|} exp(\beta u_{p,l})) \\
&= \frac{\partial}{\partial \theta_{ij}} \beta \sum_{p=1}^{\tilde{N}_A} \dot{a}_l u_l - \sum_{p=1}^{N_A} \frac{\partial}{\partial \theta_{ij}} \ln \sum_{l=1}^{|A_p|} exp(\beta u_{p,l}) = \frac{\partial}{\partial \theta_{ij}} \beta \sum_{p=1}^{\tilde{N}_A} \dot{a}_l \sum_{k=1}^{\tilde{N}_S} \dot{s}_k \theta_{kl} - \sum_{p=1}^{N_A} \frac{\partial}{\partial \theta_{ij}} \ln \sum_{l=1}^{|A_p|} exp(\beta u_{p,l}) \\
&= \beta \dot{s}_i \dot{a}_j - \sum_{p=1}^{N_A} \frac{\partial}{\partial \theta_{ij}} \ln \sum_{l=1}^{|A_p|} exp(\beta u_{p,l}) = \beta \dot{s}_i \dot{a}_j - \sum_{p=1}^{N_A} \frac{1}{\sum_{l=1}^{|A_p|} exp(\beta u_{p,l})} \frac{\partial}{\partial \theta_{ij}} \sum_{l=1}^{|A_p|} exp(\beta u_{p,l}) \\
&= \beta \dot{s}_i \dot{a}_j - \sum_{p=1}^{N_A} \frac{1}{\sum_{l=1}^{|A_p|} exp(u_{p,l})} \sum_{l=1}^{|A_p|} exp(\beta u_{p,l}) \frac{\partial}{\partial \theta_{ij}} \beta u_{p,l} = \beta \dot{s}_i \dot{a}_j - \sum_{p=1}^{N_A} \frac{1}{\sum_{l=1}^{|A_p|} exp(\beta u_{p,l})} \sum_{l=1}^{|A_p|} exp(\beta u_{p,l}) \frac{\partial}{\partial \theta_{ij}} \sum_{k=1}^{\tilde{N}_S} \beta \dot{s}_k \theta_{k(p,l)} \\
&= \beta \dot{s}_i \dot{a}_j - \frac{1}{\sum_{l=1}^{|A_{p(j)}|} exp(\beta u_{p(j),l})} exp(\beta u_j) \beta \dot{s}_i = \beta \dot{s}_i \dot{a}_j - \frac{exp(\beta u_j)}{\sum_{l=1}^{|A_{p(j)}|} exp(\beta u_{p(j),l})} \beta \dot{s}_i = \beta \dot{s}_i \dot{a}_j - \rho_j \beta \dot{s}_i = \beta \dot{s}_i (\dot{a}_j - \rho_j)
\end{aligned} \tag{4.2}$$

Using equations 4.1 and 4.2 with $\alpha := \alpha\beta$ results in

$$\Delta\theta_{ij} = \alpha \dot{s}_i (\dot{a}_j - \rho_j) R \tag{4.3}$$

for the REINFORCE update rule.

4.2.3 Distributed value function methods

If states and actions are non-distributed, i.e., if the state and action layers contain only one population each, the SNN is equivalent to a discrete policy with lookup table parameterization. Such a policy is natural to be applied to a VF based approach where each parameter is used to store a Q-value. However, we cannot apply VF methods if there are multiple populations in the state or action layer. In order to make the SNN applicable to VF based ideas in case of distributed states or actions, we will generalize VF methods. In the following we propose two different generalizations.

Distributed local value function method

In the first generalization each partial action a_p is assigned a local Q-value $Q(s,a_p;\theta)$. These are used by the partial policies to choose a partial action from the partial action spaces according to some action selection strategy. In terms of our SNN we interpret the WTA based activation function used by the action populations as action selection strategy and the neuron potential of the active neuron in action population p as an approximation of the local Q-value of the corresponding partial action a_p, i.e., $Q(s,a_p;\theta) = Q(\underline{\dot{s}},\underline{\dot{a}}_p;\theta) = \sum_{l=1}^{|A_p|} \dot{a}_{p,l}u_{p,l}$, where $\sum_{l=1}^{|A_p|} \dot{a}_{p,l}u_{p,l}$ corresponds to the neuron potential of the active neuron in action population p.
After receiving the return $R = r$, we try to update the local Q-value of the selected partial action a_p in population p according to $Q(s,a_p;\theta) := Q(s,a_p;\theta) + \Delta Q(s,a_p;\theta)$ with $\Delta Q(s,a_p;\theta) = \alpha(R - Q(s,a_p;\theta))$. To realize this update for all selected partial actions we minimize an error function given by $E = \frac{1}{2}\sum_{p=1}^{N_A} \Delta Q(s,a_p;\theta)^2$ by performing one step of gradient descent w.r.t. θ, i.e.,

$$\begin{aligned}
\Delta\theta_{ij} &= -\bar{\alpha}\frac{\partial E}{\partial \theta_{ij}} = -\bar{\alpha}\frac{\partial \frac{1}{2}\sum_{p=1}^{N_A}\Delta Q(s,a_p;\theta)^2}{\partial\theta_{ij}} = -\bar{\alpha}\frac{\partial \frac{1}{2}\sum_{p=1}^{N_A}(\alpha(R-Q(s,a_p;\theta)))^2}{\partial\theta_{ij}} = -\bar{\alpha}\frac{\partial \frac{1}{2}\sum_{p=1}^{N_A}(\alpha(R-\sum_{l=1}^{|A_p|}\dot{a}_{p,l}u_{p,l}))^2}{\partial\theta_{ij}} \\
&= -\bar{\alpha}\frac{\partial \frac{1}{2}\sum_{l=1}^{\bar{N}_A}\dot{a}_l(\alpha(R-u_l))^2}{\partial\theta_{ij}} = -\bar{\alpha}\sum_{l=1}^{\bar{N}_A}\dot{a}_l\alpha(R-u_l)\frac{\partial\alpha(R-u_l)}{\partial\theta_{ij}} = -\bar{\alpha}\sum_{l=1}^{\bar{N}_A}\alpha\dot{a}_l(R-u_l)\frac{\partial\alpha(R-\sum_{k=1}^{\bar{N}_S}\dot{s}_k\theta_{kl}))}{\partial\theta_{ij}} \\
&= -\bar{\alpha}\alpha\dot{a}_j(R-u_j)(-\alpha\dot{s}_i) = \bar{\alpha}\alpha^2\dot{s}_i\dot{a}_j(R-u_j). \qquad (4.4)
\end{aligned}$$

With $\alpha := \bar{\alpha}\alpha^2$ equation 4.4 results in

$$\Delta\theta_{ij} = \alpha\dot{s}_i\dot{a}_j(R-u_j) \qquad (4.5)$$

as update rule for our distributed local value function method.

Distributed global value function method

Another value function approach can be motivated by assuming that we make use of softmax as activation function, that is as action selection strategy. In this case the probability of a global action a in some state s is given by

$$\pi(s,a;\theta) = \pi(\underline{\dot{s}},\underline{\dot{a}};\theta) = \prod_{p=1}^{N_A}\pi(\underline{\dot{s}},\underline{\dot{a}}_p;\theta) = \prod_{p=1}^{N_A}\frac{exp(\beta\sum_{l=1}^{|A_p|}\dot{a}_{p,l}u_{p,l})}{\sum_{l=1}^{|A_p|}exp(\beta u_{p,l})} = \prod_{p=1}^{N_A}\frac{exp(\beta\sum_{l=1}^{|A_p|}\dot{a}_{p,l}u_{p,l})}{Z_p} = \frac{exp(\beta\sum_{p=1}^{N_A}\sum_{l=1}^{|A_p|}\dot{a}_{p,l}u_{p,l})}{Z}. \qquad (4.6)$$

The shape of the probability a global action in equation 4.6 gives rise to the idea to use global Q-values for global actions with $Q(s,a;\theta) = \sum_{p=1}^{N_A}\sum_{l=1}^{|A_p|}\dot{a}_{p,l}u_{p,l}$, i.e.,

$$\pi(s,a;\theta) = \frac{exp(\beta Q(s,a;\theta))}{Z} = \frac{exp(\beta\sum_{p=1}^{N_A}\sum_{l=1}^{|A_p|}\dot{a}_{p,l}u_{p,l})}{Z}.$$

After receiving return $R = r$, we try to update the global Q-value of the selected global action according to $Q(s,a;\theta) := Q(s,a;\theta) + \Delta Q(s,a;\theta)$ with $\Delta Q(s,a;\theta) = \alpha(R - Q(s,a;\theta))$. To realize this update we minimize an error function given by $E = \frac{1}{2}\Delta Q(s,a;\theta)^2$ by performing one step of gradient descent w.r.t. θ, i.e.,

$$\begin{aligned}
\Delta\theta_{ij} &= -\bar{\alpha}\frac{\partial E}{\partial \theta_{ij}} = -\bar{\alpha}\frac{\partial \frac{1}{2}\Delta Q(s,a;\theta)^2}{\partial \theta_{ij}} = -\bar{\alpha}\frac{\partial \frac{1}{2}(\alpha(R - Q(s,a;\theta)))^2}{\partial \theta_{ij}} = -\bar{\alpha}\frac{\partial \frac{1}{2}(\alpha(R - \sum_{p=1}^{N_A}\sum_{l=1}^{|A_p|}\dot{a}_{p,l}u_{p,l}))^2}{\partial \theta_{ij}} \\
&= -\bar{\alpha}\frac{\partial \frac{1}{2}(\alpha(R - \sum_{l=1}^{\bar{N}_A}\dot{a}_l u_l))^2}{\partial \theta_{ij}} = -\bar{\alpha}\alpha(R - \sum_{l=1}^{\bar{N}_A}\dot{a}_l u_l)\frac{\partial \alpha(R - \sum_{l=1}^{\bar{N}_A}\dot{a}_l u_l)}{\partial \theta_{ij}} = -\bar{\alpha}\alpha(R - \sum_{l=1}^{\bar{N}_A}\dot{a}_l u_l)\frac{\partial \alpha(R - \sum_{l=1}^{\bar{N}_A}\dot{a}_l \sum_{k=1}^{\bar{N}_S}\dot{s}_k\theta_{kl})}{\partial \theta_{ij}} \\
&= -\bar{\alpha}\alpha(R - \sum_{l=1}^{\bar{N}_A}\dot{a}_l u_l)(-\alpha\dot{a}_j\dot{s}_i) = \bar{\alpha}\alpha^2\dot{s}_i\dot{a}_j(R - \sum_{l=1}^{\bar{N}_A}\dot{a}_l u_l). \qquad (4.7)
\end{aligned}$$

With $\alpha := \bar{\alpha}\alpha^2$ equation 4.7 this results in

$$\Delta\theta_{ij} = \alpha\dot{s}_i\dot{a}_j(R - \sum_{l=1}^{\bar{N}_A}\dot{a}_l u_l) \qquad (4.8)$$

as update rule for our distributed global value function method.

4.2.4 Theoretical comparison of the methods for training the SNN

The update rules of REINFORCE in equation 4.3 and the update rules of the proposed distributed VF methods in equations 4.3 and 4.8 are learning rules of Hebbian type.
For the REINFORCE update rule the synaptic weight θ_{ij} can only change if $\dot{s}_i = 1$. However, in general the inactivity of action neuron j does not prevent REINFORCE from modifying weight θ_{ij}[6]. Thus, REINFORCE cannot modify parameters if they do not belong to the occurred state but it can modify parameters if they belong to the occurred state but not to the occurred action.
According to the equations 4.3 and 4.8 the proposed distributed VF methods can only modify parameters if they belong to the occurred state-action pair, i.e., if $\dot{s}_i = 1$ and $\dot{a}_j = 1$ holds.
In contrast to these Hebbian learning rules, black-box methods do not take the neuron activities into account. Thus, they may modify each synaptic weight at each update step regardless of the neural states.

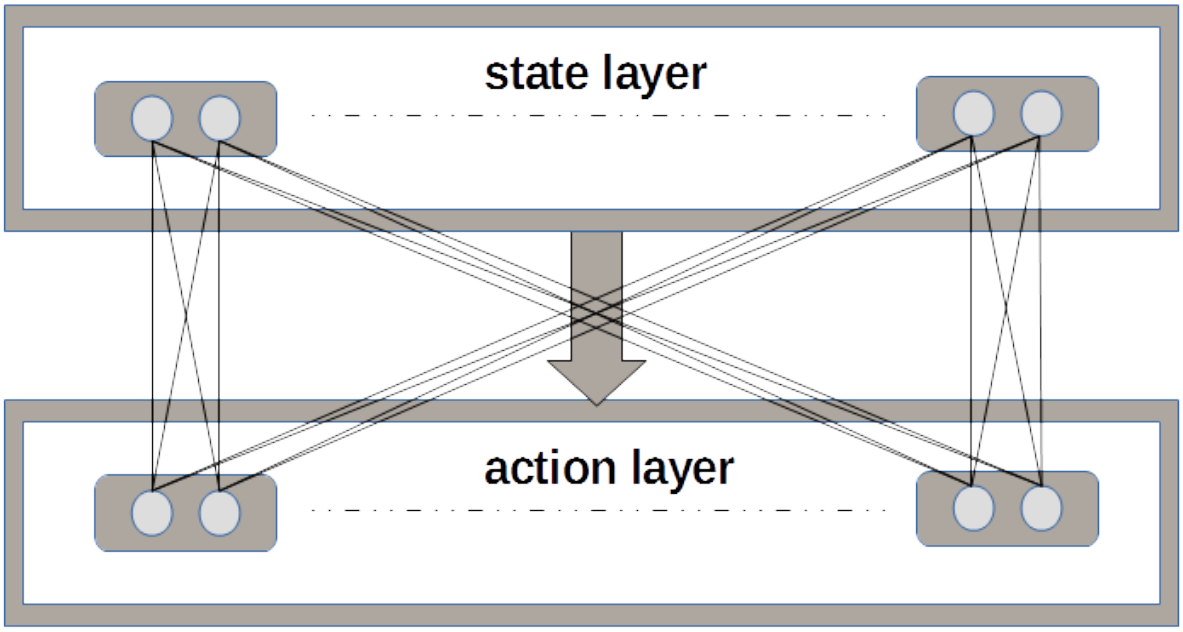

Figure 4.1: The proposed distributed policy based on a network of populations of stochastic spiking neurons (SNN). The SNN consists of two layers, a state layer and an action layer. Both layers contain populations of neurons representing partial states and partial actions each. For each pair of a state neuron i, $1 \leq i \leq \bar{N}_S$, and an action neuron j, $1 \leq j \leq \bar{N}_A$, we have exactly one synaptic connection with weight θ_{ij}. There are no synaptic connections between neurons in the same layer. In the above example each population contains exactly two neurons. In each population exactly one neuron is active at one moment. This holds for all populations in the state and layer action. To ensure this for the populations in the action layer winner-take-all dynamics are employed. If the state and layer contain exactly one population each, the SNN is equivalent to a discrete basic policy with lookup table parameterization. Thus, the proposed SNN is a spatially distributed generalization of such a basic policy type.

5 Experiments

In this section we evaluate the proposed spiking network on different learning methods and reinforcement learning tasks. As tasks we chose two robot-arm tasks where the goal is reach a target with the end effector of the robot arm. The first task is a one-step MDP, where no sequential decision-making is required and the second task is a multi-step MDP, where the task can only be solved by executing a proper sequence of actions. The tasks are toy tasks and are supposed to give us a first proper insight into the quality of our SNN and the corresponding learning methods.

5.1 Non-sequential 10-link robot arm task

In our first toy task we have a robot arm with 10 links connected by 10 rotational joints and the goal is to adjust the joints such that the end effector hits a specified target. This task is modeled as one-step MDP where the episode ends after one action was executed. In this task an action defines the absolute values q of the joints. After executing the action a reward is received. The base of the robot arm is located at $(0, 0)$ and the target to be reached is located at $(10, 0)$. Each joint can take four different values $q_i \in \theta \in \{\frac{\pi}{2}, \pi, \frac{3\pi}{2}, 2\pi\}$, $1 \leq i \leq 10$. As topology of the network we have an action layer with 20 binary populations. Each joint value is encoded by the neural states of two binary populations.
The difficulty of this task lies in adjusting the joints to reach the target. In later tasks the difficulty of the tasks will be increased as we change the complexity of the state space and the structure of the state layer for these tasks. The reward is defined as the negative euclidean distance between the end effector and the target. This task is a basic task to get first insights into the power of the SNN and the corresponding different learning methods but is still a well-defined reinforcement learning problem. A visualization of the task is shown in figure 5.1.

5.1.1 Constant state

We start with a one-step MDP that has only one state which corresponds to a multi-armed bandit problem. Consequently, we use a state layer that contains only one population with one neuron that is always active. As learning methods we evaluate the CMA-ES black-box optimizer, REINFORCE and the two proposed distributed value function approaches. As activation function we used softmax. For all experiments the synaptic weights are set to zero at the start of each run. As hyperparameters we use the default setting for the CMA-ES method. The hyperparameters of the remaining methods are given in table 5.1. These hyperparameter settings appeared to work best among the tested ones. For each method we performed 10 runs with 25000 episodes each.
The results are shown in figure 5.2 in the top picture. The results show that the distributed global VF method outperformed all other methods significantly and was even able to find the global optimum, i.e., hitting the target, constantly. The local VF method performed worse than the global VF method, but did also outperform the CMA-ES and the REINFORCE method significantly. REINFORCE got stuck in a local optimum which can be accounted to the fact that it is based on gradient descent. The CMA-ES method and REINFORCE performed equally. We also tried CMA-ES for a setting where the incoming synapses of one neuron in each action population was removed and the corresponding potential of neuron set to zero which allows for a reduction of the number of parameters by half without changing the expressive power of the SNN. However, despite a small speed up of the optimization process the results were qualitatively the same.

	α	β
CMA-ES	-	1.0
REINFORCE	0.005	1.0
VF, local	0.02	1.5
VF, global	0.02	2.5

Table 5.1: Hyperparameters of the different learning methods for the non-sequential robot-arm task with a constant state.

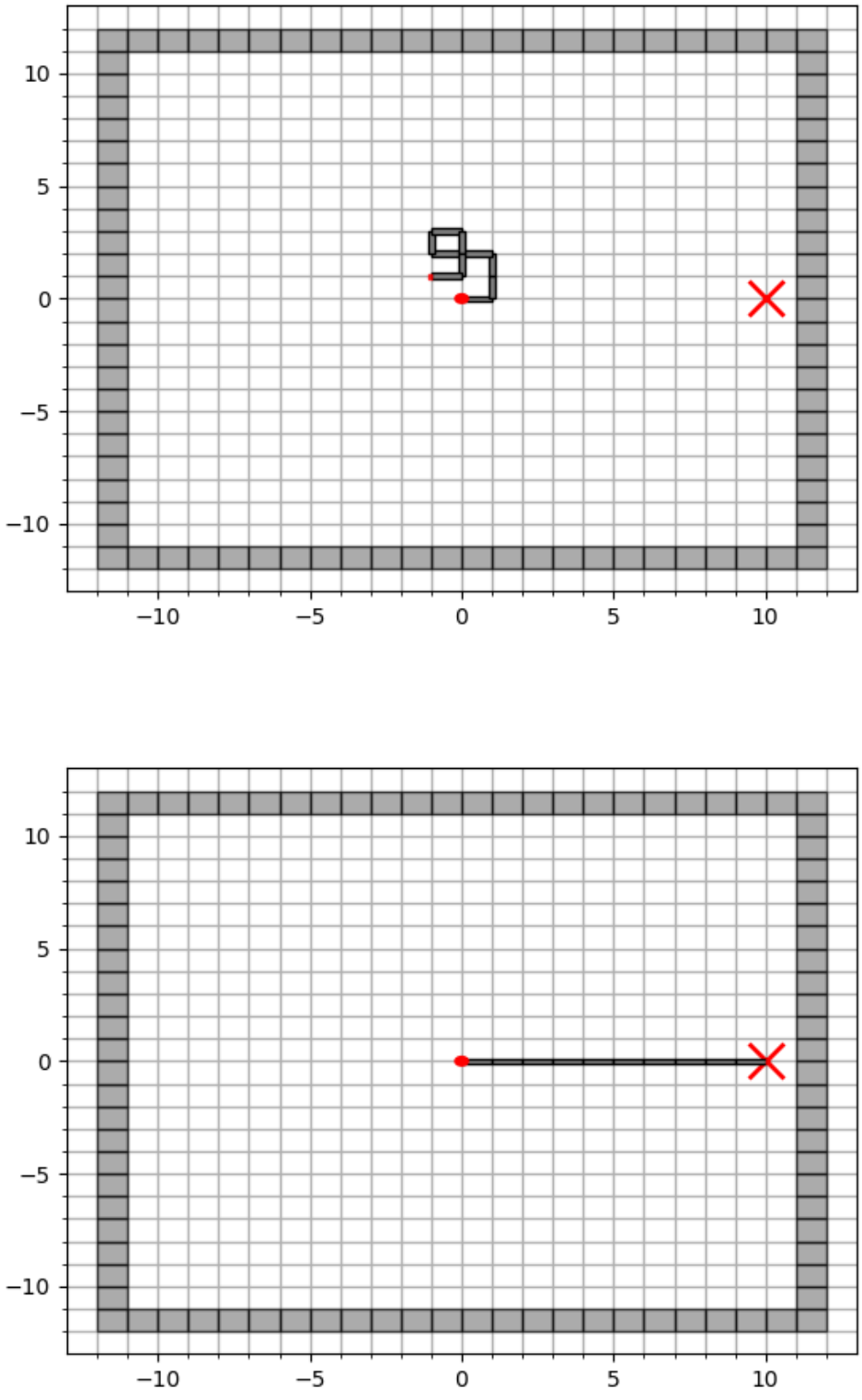

Figure 5.1: Visualization of the non-sequential 10-link robot arm task where the episode terminates after one step. The goal is to adjust the joints such that the end effector reaches the target. In the top picture the joints are adjusted poorly, since there is a huge distance between the end effector and the target. In the bottom picture the end effector hits the target.

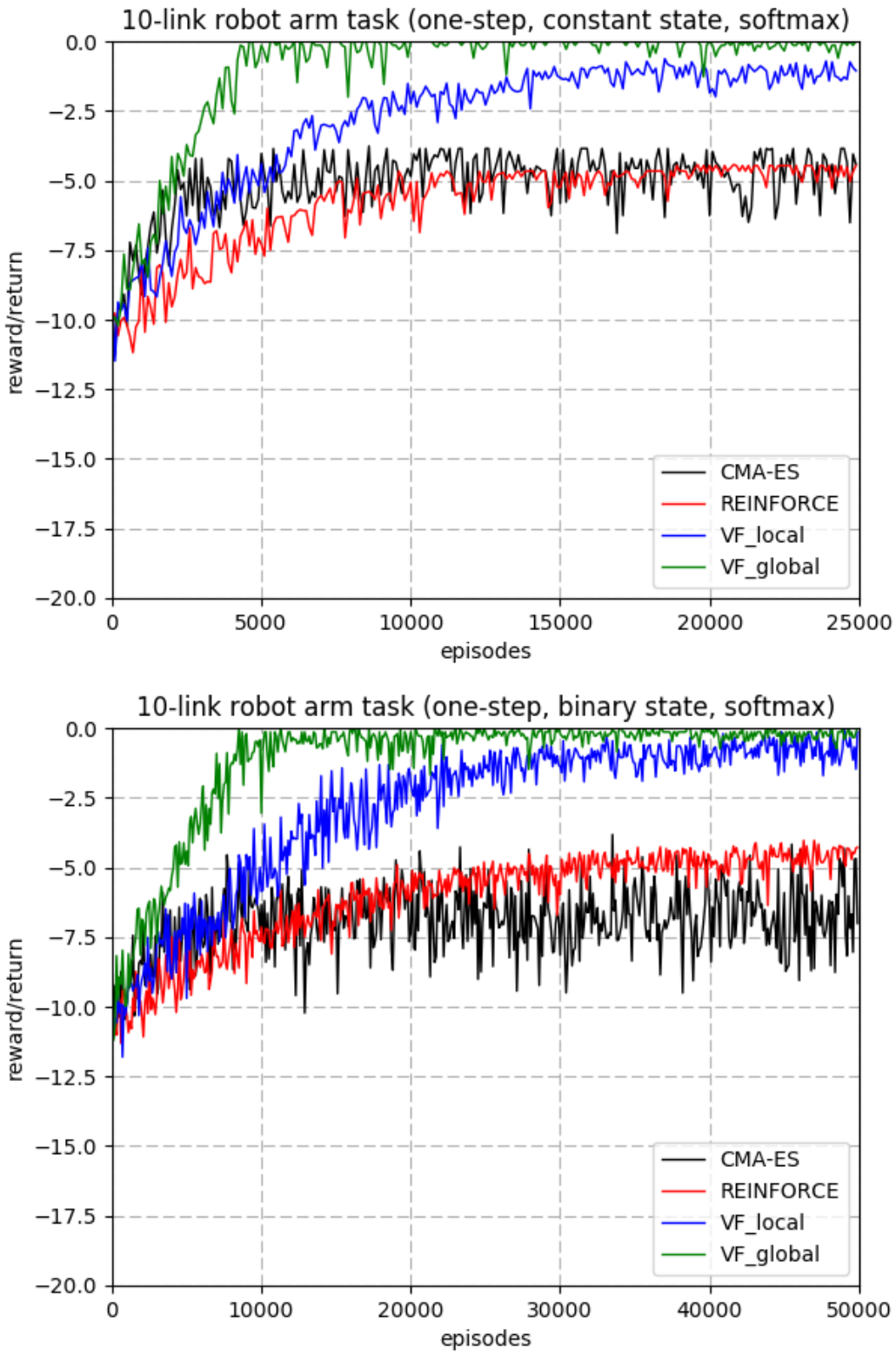

Figure 5.2: Comparison of different learning methods for the non-sequential robot arm task with complex actions. In the first case (top picture) we have a constant state and in the second case (bottom picture) we have a binary state. In order to make the comparison easier we doubled the number of episodes for the task with a binary state, since we have to learn a proper action for 2 different states in this case. Shown is the average of 10 runs for each method. As we can see only the performance of the CMA-ES method decreases whereas the Hebbian learning methods do not lose in performance.

In another experiment for the same problem we also compared the activation functions softmax and ϵ-greedy for the value function based approaches. The value function based approaches were chosen due to their dominant performance in the previous experiment. The hyperparameter settings are given in table 5.2. As hyperparameter settings we used the best ones we were able to find for the corresponding methods and activation functions. For each method we performed 10 runes with 25000 episodes each.

	α	β	ϵ
VF, local, softmax	0.02	1.5	-
VF, global, softmax	0.02	2.5	-
VF, local, ϵ-greedy	0.001	-	0.03
VF, global, ϵ-greedy	0.001	-	0.03

Table 5.2: Hyperparameters of the distributed VF methods for the activation functions softmax and ϵ-greedy for the non-sequential robot-arm task with a constant state.

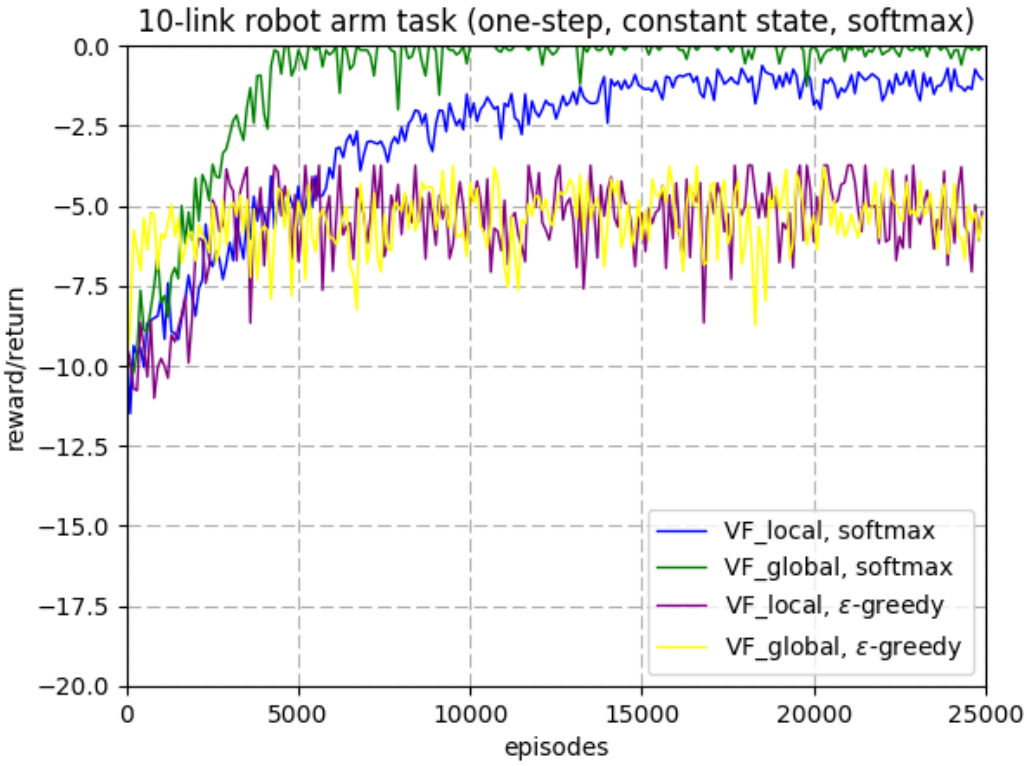

Figure 5.3: Comparison of different activation functions for the two different distributed VF methods in the one-step MDP setting. The softmax activation function outperforms ϵ-greedy significantly.

The results are shown in figure 5.3. We can observe that ϵ-greedy gets significantly outperformed by softmax as it gets stuck at a bad solution and is not able to escape from there for both distributed VF methods. Even a low learning rate used for ϵ-greedy was not able to circumvent this issue.

5.1.2 Binary state

In a more difficult setting we use an MDP with two states. The states are uniformly distributed. Depending on the state the goal to be hit is either at position $(-10, 0)$ or at $(10, 0)$, that is the learned policy from the last experiment will not work here, since hitting the location at $(10, 0)$ will lead to a very bad reward in half of all cases. To deal with such conditions the state layer of our SNN is extended to a single population with two neurons, where the activity of the two neurons depends on the MDP state. The used hyperparameters are the same as used in table 5.1. Since have to learn a

joint configuration for twice the number of states we performed for each method 10 runs with 50000 episodes each.
The results are shown in figure 5.1 together with the results for the constant state and reveal that the performance in comparison to the case with a constant state is basically the same for all methods except CMA-ES. This can be accounted to the fact that, in contrast to all other methods based on Hebbian learning, CMA-ES modifies parameters even for states that did not occur. This reveals a crucial problem of black-box optimization. We can assume that the performance gap between CMA-ES and the Hebbian learning methods would increase even further with an increasing number of states. We also tested CMA-ES for a setting with a reduced number of parameters, but as in the last experiment it had no significant impact on the results.

5.1.3 Complex states

In this experiment we will investigate how well we can deal with states of high complexity which differs significantly from the experiments done so far. For doing so we will only focus on the distributed global VF method with softmax as activation function as it appears to work best for our SNN according to the results of the last experiments. This time we have 2^{20} states with a uniform distribution. To encode these states we make use of a distributed state representation with 20 binary populations. Obviously, a single state population for encoding these states is not possible due to the high number of states.
We will evaluate for four different cases where target to be hit is determined by specific state patterns. In the first case the target location is constant and in the other three cases one out of two target locations is determined by a binary value that depends on the current state. We will investigate how well we can deal these cases and show up crucial limitations for our SNN. As hyperparameters we used $\alpha = 0.002$ and $\beta = 2.5$. In order to avoid an unstable learning process with multiple state populations we had to reduce the learning rate for these experiments. For each experiment we performed 10 runs with 50000 episodes each.

Constant target

In the first experiment for complex states the target is independent of the state and has a fixed location at $(10, 0)$. We want to investigate how fast we can learn a proper joint configuration that hits the target if complex random state patterns are fed into our SNN that have no relation to the reward received for executing an action.
The resulting weight matrix of a single run is shown in figure 5.4. We can see that the second action neuron of each population receives a higher weight than the first one for each (partial) state. For our used joint encoding this corresponds to a qualitatively correct solution for this problem.

Binary target

In this experiment we have a binary target location either at $(10, 0)$ or at $(-10, 0)$. The target location is determined by a single bit specified by the first binary state population. The difficulty of this task is two-folded. On the one hand we have to identify the bit, i.e., the state population that defines the target location and separate it from the remaining random bits and on the other hand we still have to adjust the joints such that the endeffector reaches the target specified by this bit. The resulting matrix of one run is shown in figure 5.5 and corresponds to a correct solution.
The resulting matrix is almost the same the matrix as in the last last experiment with the difference that the synaptic weights between the two neurons of the first action population whose activity contributes to the adjustment of the base joint and the neurons of the first state population that determines the target have very specific and dominant weights. In fact the joint configuration for reaching the two different targets differs only in the adjustment of the first joint at the base. The learning method is able to identify and to exploit this fact.

Binary target specified by OR function

This experiment is similar to the last one with the difference that the binary target location is specified by the result of the boolean OR function applied to the bits given by the first two state populations.
The resulting matrix is shown in figure 5.6 and corresponds to a correct solution. Similar to the last experiment the weights between the neurons of the first action population and the first two state populations have very specific and dominant weights expressing that the first two state populations have a high impact on the base joint.

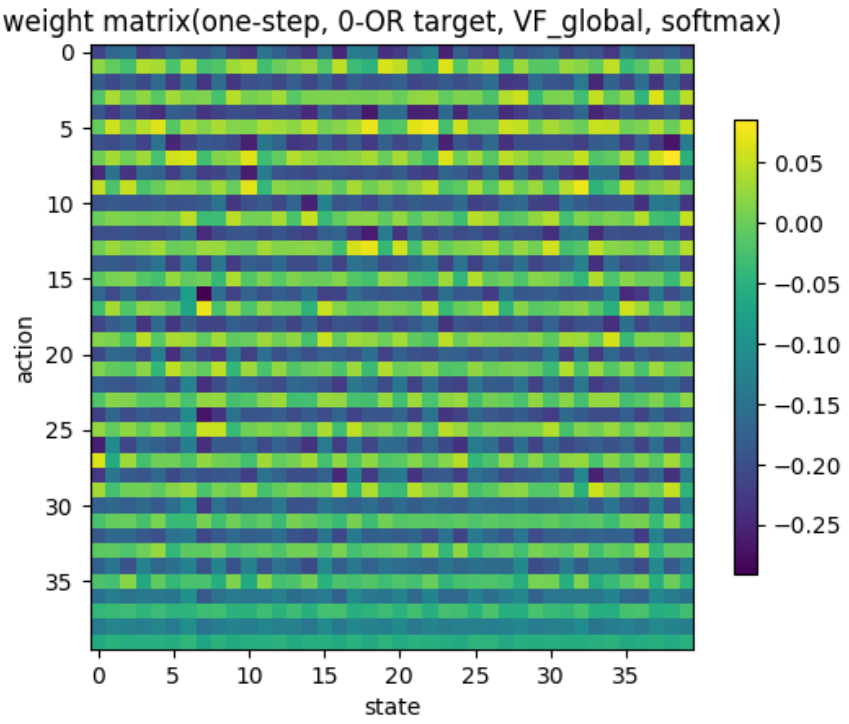

Figure 5.4: Resulting weight matrix for complex states with a constant target location. The weights between the neurons of each second action population and the neurons all state populations are higher then the remaining ones, which corresponds to a correct solution according to our used encoding.

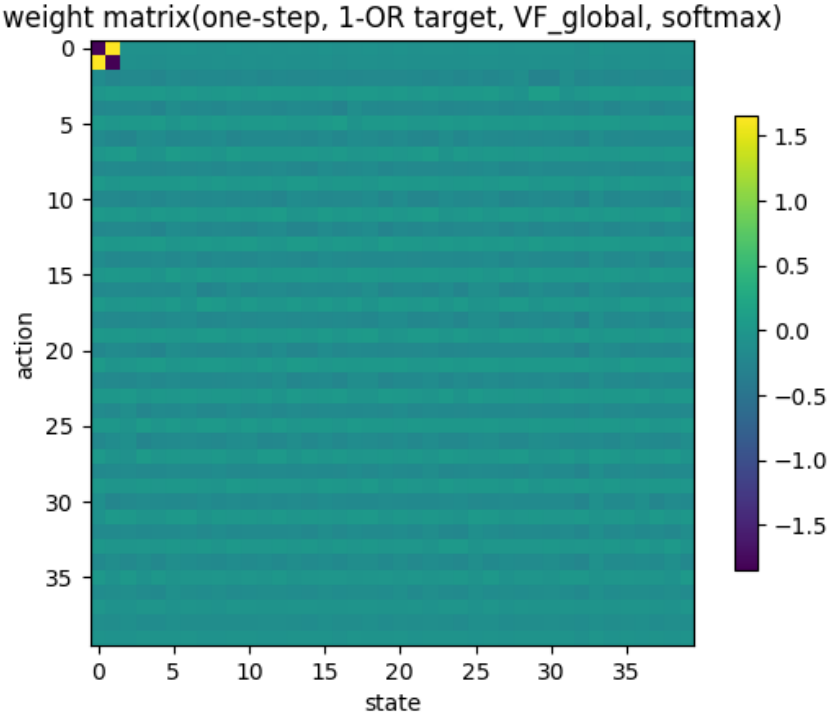

Figure 5.5: Resulting weight matrix for complex states with a binary target location specified by a single state population. The weights between the neurons of the first action population contributing to the angle of the base joint and the neurons of the the first state population specifying the target location have significant and dominant weights.

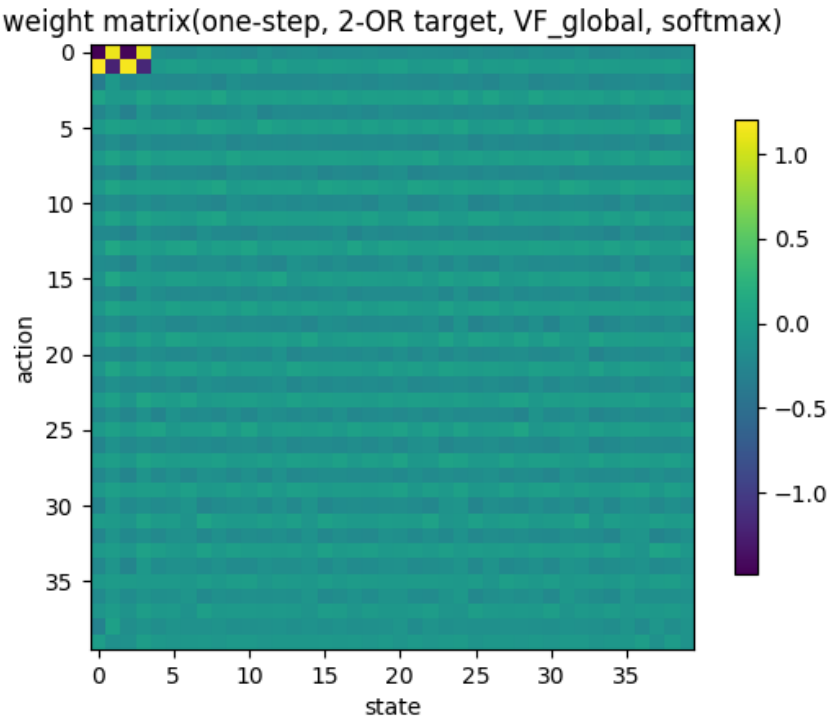

Figure 5.6: Resulting weight matrix for complex states with a binary target location specified by the boolean OR function applied to the bits specified by the first two state populations. The weights between the neurons of the first action population contributing to the angle of the base joint and the neurons of the first and second state population specifying the target location have significant and dominant weights.

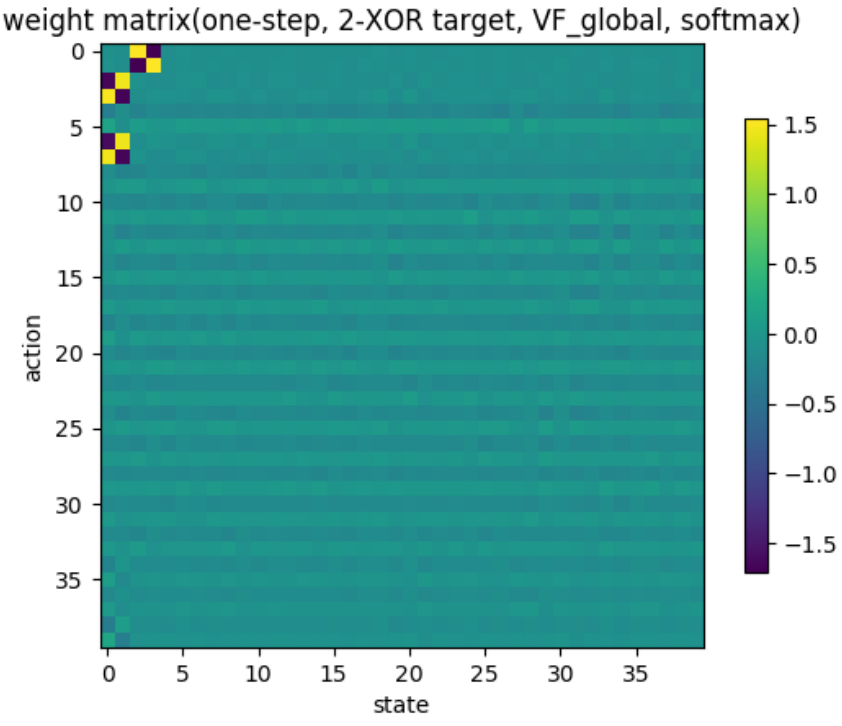

Figure 5.7: Resulting weight matrix for complex states with a binary target location specified by the boolean XOR function applied to the bits specified by the first two state populations. The weights between the neurons of the first four action populations contributing to the angle of the first two joints and the neurons of first two state populations specifying the target location have significant and dominant weights. In contrast to the OR specification of the target, in this case the state populations also have an impact on the second joint action population due to the fact that the SNN cannot solve the XOR task properly.

Binary target specified by XOR function

In this experiment the target is specified by the boolean XOR function applied to the bits given by the first two state populations. The resulting matrix is shown in figure 5.7. In contrast to the last experiments this task cannot be learned correctly, since the recognition of XOR patterns goes beyond the expressive power of our perceptron like SNN. As the weight matrix in figure 5.7 shows there are not only significant weights for the neurons of the first action population but also for the second and fourth population contributing to the first and second joint angles respectively. In fact the best policy we can learn with our SNN for an XOR pattern can only hit one of the two target locations correctly. The other target can only be reached closely but not exactly as shown in figure 5.9.

Comparison of the learning speed for the tasks with complex states

The results of the described tasks with complex states are shown in figure 5.8. As we can see the number of episodes until convergence increases with the difficulty of the task. For all tasks except for the XOR task the learning process is converging fast to a correct solution. For the XOR task the convergence gets significantly worse in comparison to the other tasks and no correct solution can be found, since the SNN cannot identify XOR patterns correctly.

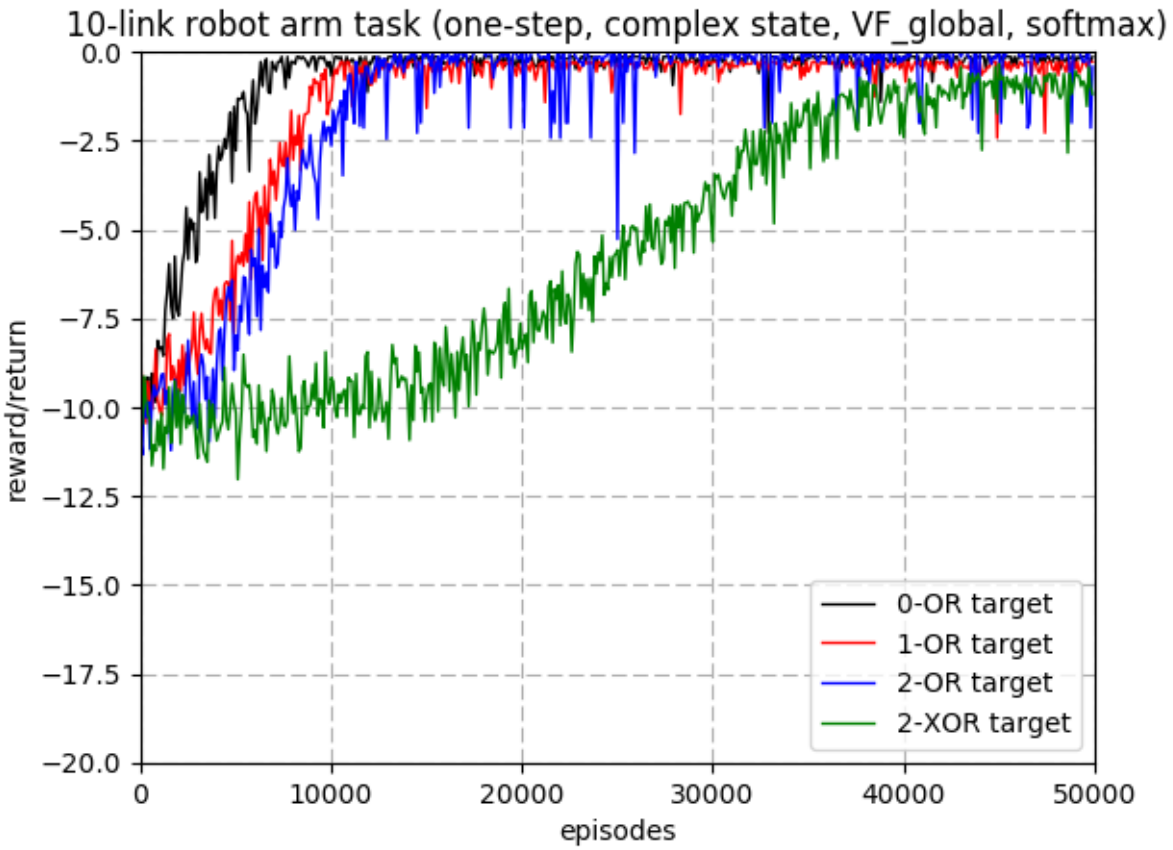

Figure 5.8: Comparison of the convergence for the tasks with complex states. The convergence for the XOR task is significantly worse than for the other ones and no solution that hits the target in each case can be found due to the limited expressive power of our SNN.

5.2 Sequential 10-link robot arm task

In this experiment we want to make use of the knowledge gained so far from the experiments with a one-step MDP setting and use it for a more interesting sequential 10-link robot-arm task formulated as a multi-step MDP. In this task we have a robot arm starting in some initial configuration and the goal is to produce a proper sequence of actions such that the end effector moves through a two-dimensional grid world to hit a target without hitting any obstacle. A visualization of this problem is shown in figure 5.10.

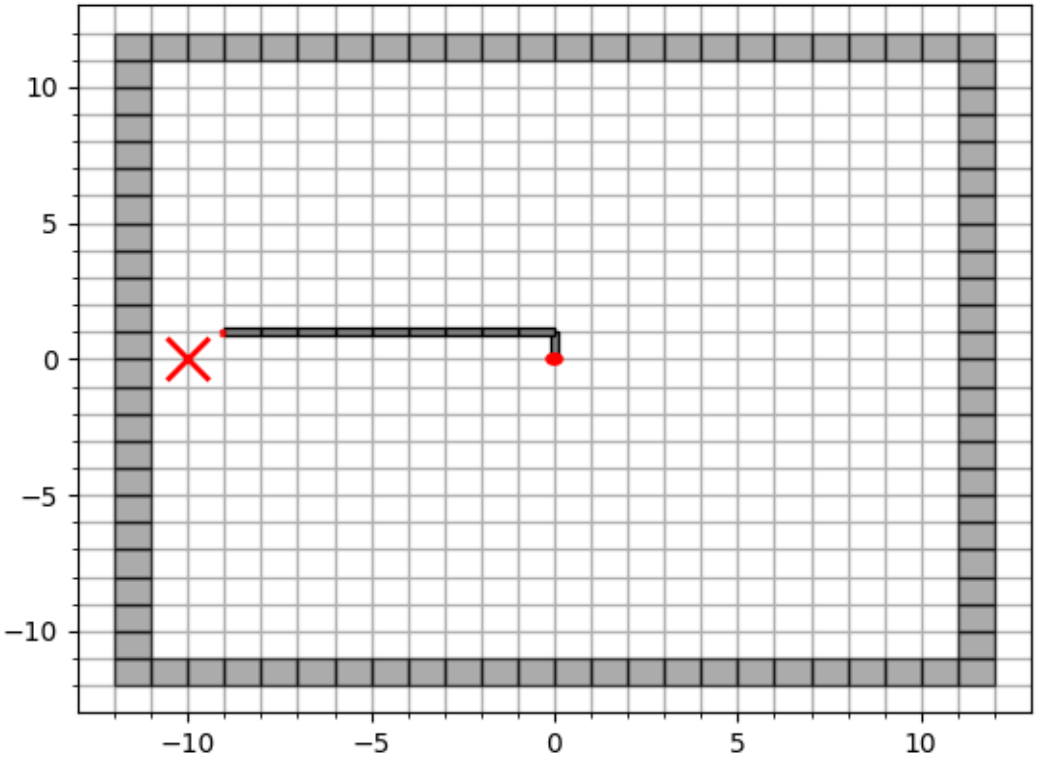

Figure 5.9: Non-sequential robot arm task where the target to hit is specified by an XOR-pattern in the state string. It is not possible to solve this task perfectly such that the robot arm hits the correct target depending on whether the two bits determining the target are different or not. However, the learning method is still able to find an almost optimal solution for the used policy, which is in fact the best solution that can be found with our perceptron-like SNN. In this solution one target is hit correctly and the other one is almost hit as shown in the picture above.

As before the robot arm has 10 links. In this toy task the actions are values $\Delta q_{i,t}$ added to the current joint angles $q_{i,t}$. For each joint we have three possible values $\Delta q_{i,t} \in \{-\frac{\pi}{50}, 0, \frac{\pi}{50}\}$, $1 \leq i \leq 10$ that can be added to $q_{i,t}$, which encoded by two bits each. Thus, we can encode Δq_t with 20 binary action populations where each joint is assigned two binary action populations as in the non-sequential robot arm task. Furthermore, for all joints except the basis joint we restrict the corresponding joint angles to be in the interval given by $q_{i,t} \in [0, \pi]$, $2 \leq i \leq 10$. The initial angle of each joint is $q_{i,t} = 0$. As states we use the grid cells as locations of the end effector and encode these by a single neuron population in the state layer where each single neuron corresponds to a single grid cell. This is possible, since the number of grid cells is not too high and furthermore we do not lose expressive power by using non-distributed state representations as this can lead to serious problems as demonstrated in the XOR task.
For the rewards r_t we use a distance measure d_t to express how close the end effector is to the target and define the rewards as the difference between the distance in the next step and the current distance, i.e., $r_t = d_{t-1} - d_t$. As distance we use the minimum number of steps required to reach the target in the grid world assuming that we can only move up, down, left or right. To compute the distance map we used a dynamic programming related approach and applied it until convergence of the distances. A visualization of the resulting distance map is shown in figure 5.11. If the end effector hits an obstacle a reward of $r_t = -10$ is provided and the episode ends. Furthermore we do not use delayed rewards with $\gamma > 0$, i.e., we use $\gamma = 0$. This reduces the variance of the returns, since $R_t = r_t$ holds and it makes our learning methods that are based on the assumption of a one-step MDP easily applicable to a multi-step MDP. In fact our learning methods cannot deal with delayed rewards at the current moment. An important property of the reward definition based on relative distances is the resulting advantageous return distribution. Assuming that the end effector does not hit an obstacle the return has approximately a mean of zero and a low variance, which is ideal to train our SNN that serves as an actor. We still assume that the used reward function is powerful enough to guide the robot arm through the task.

We performed 10 runs with 1000 episodes each and restarted the episode if the end effector hit an obstacle or the target or if the number of episode steps reached 1000. Based on the results of the experiments for the one-step MDP we used the distributed global VF method as learning method and softmax as activation function. For the hyperparameters we chose a learning rate of $\alpha = 0.001$ and an inverse temperature of $\beta = 50$ which appeared to work best. As before the synaptic weights were initially set to 0.0. Assuming that no obstacle was hit this weight initialization now corresponds to the average return received and not to the optimal return as before in the one-step MDP experiments.
The results are shown in figure 5.12 where the final negative distance to the target at the end of the episode is shown depending on the number of episodes. As we can see there, the SNN is able to produce trajectories over time with increasing reliability and the end effector gets in average very close to the target location without hitting obstacles and in some episodes the end effector even hits the target for all runs. These results reveal a high performance for this non-trivial task.

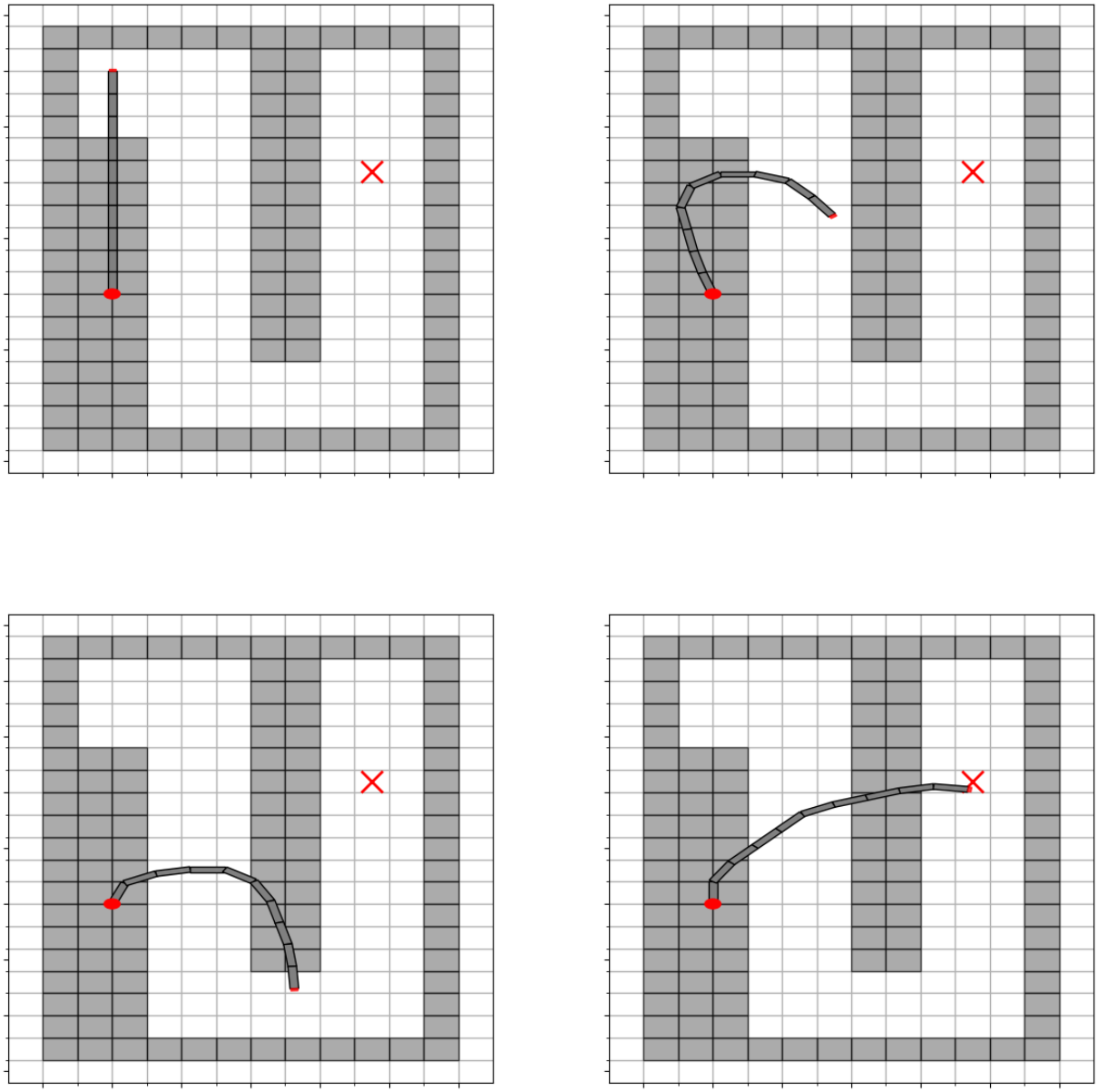

Figure 5.10: Visualization of the sequential 10-link robot arm task where the goal is to produce a proper sequence of actions such that the end effector reaches the target(red cross) without hitting an obstacle(grey cells). The initial joint configuration is shown in the top left picture.

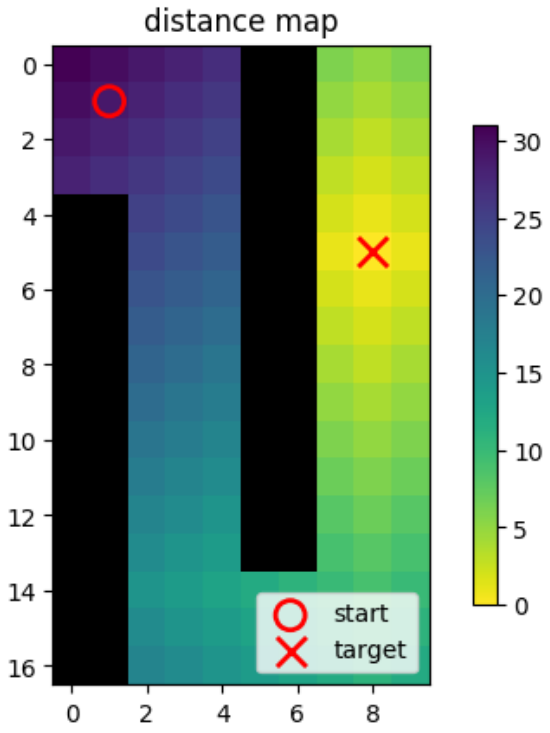

Figure 5.11: Visualization of the distance map for the sequential 10-link robot arm task. For each grid cell the distance to the target is visualized. The distance is defined as the the minimum number of steps an agent has to move up, down, left or right to reach the target without hitting an obstacle. Such a distance map can be computed efficiently with dynamic programming related techniques.

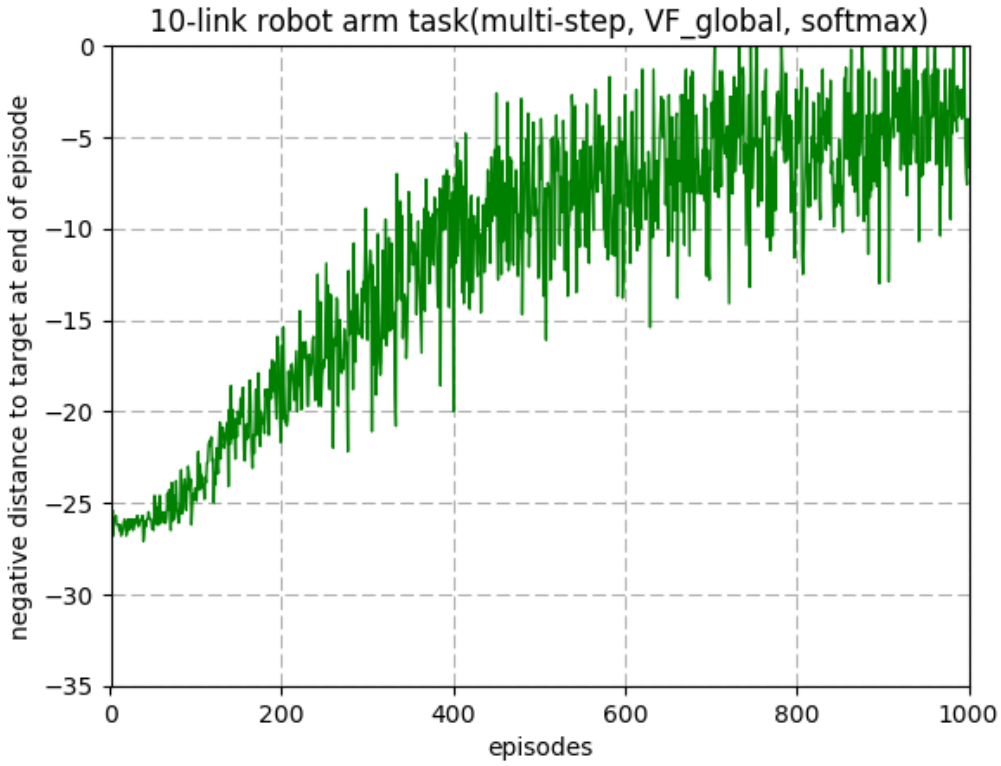

Figure 5.12: Results of the sequential 10-link robot arm task where the negative distance to the target at the end of the episode is shown depending on the number episodes. We performed 10 runs for this task and shown is the average of all runs.

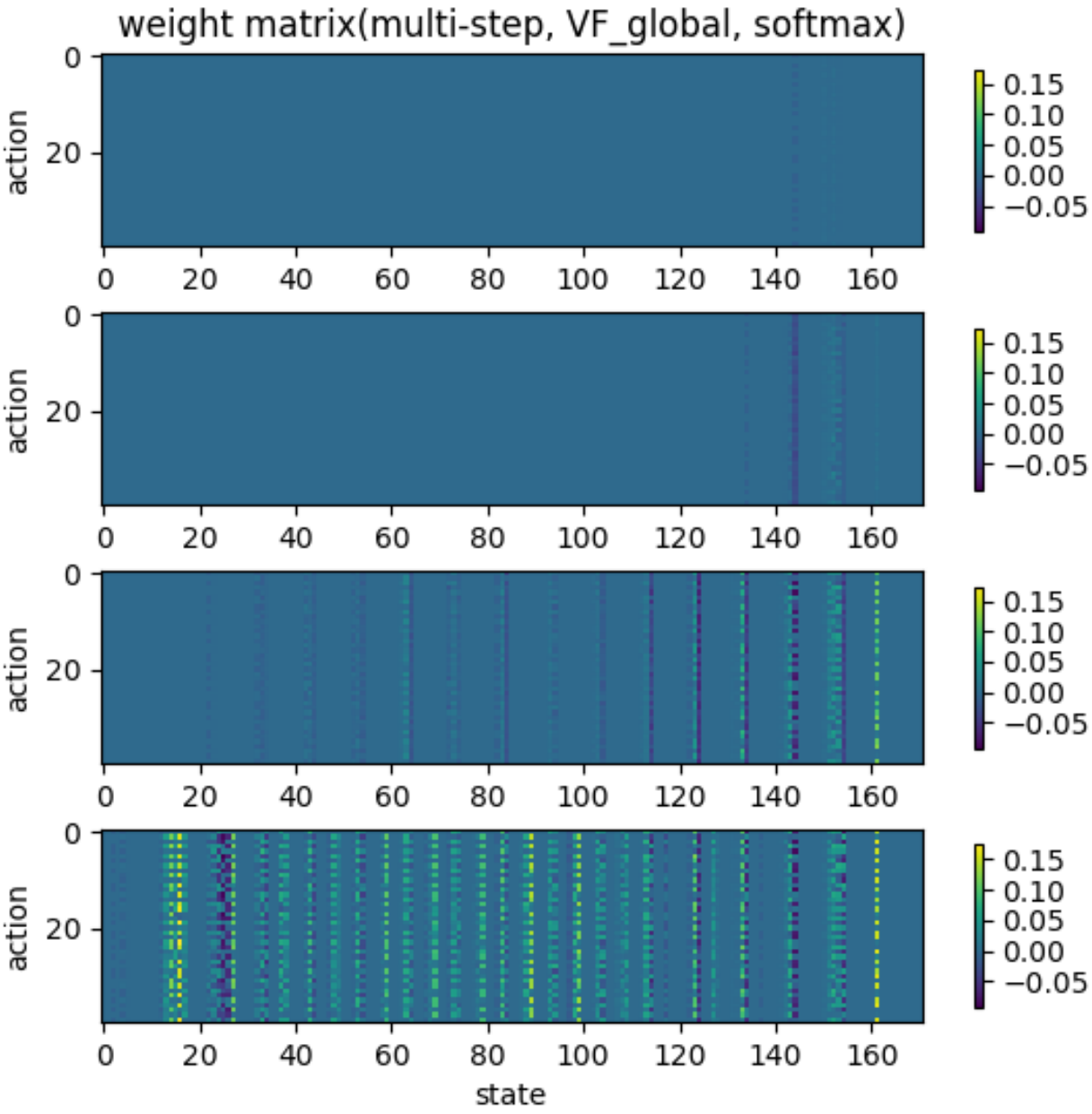

Figure 5.13: Visualization of the weight matrix for a sample run of the sequential 10-link robot arm task. The actions correspond to Δq_t added to the current joint angles q_t and the states corresponds to the grid cell containing the end effector. The resulting matrices are shown for episodes 1, 10, 100 and 1000 beginning from top.

6 Conclusion

In this thesis we proposed, motivated and evaluated a distributed type of policy based on a two-layered stochastic spiking neural network (SNN). The motivation and derivation was done by generalizing a discrete basic policy with lookup table parameterization to make its representation and dynamics consistent with the distributed and efficient working principle of natural computation models given by neural networks and cellular automata. We showed that by doing so we are able to overcome the curse of dimensionality such that we can deal with high-dimensional state and action spaces. Furthermore, we proposed two distributed value function based Hebbian learning methods to train the SNN from rewards in a reinforcement learning setting.

We evaluated the SNN and the proposed learning methods together with state-of-the-art methods on a simple non-sequential robot arm task with a non-complex state space and a complex action space. The used state-of-the-art methods were CMA-ES as black-box optimizer and REINFORCE. We used a single state population and multiple action populations. It turned out that the distributed global VF method in combination with the softmax activation function outperformed all other approaches significantly and achieved a high performance. Plausible explanations why the state-of-the-art methods were outperformed is the fact that CMA-ES as black-box method does not take the activity of neurons into account as Hebbian learning methods do and modifies all parameters and REINFORCE is based on gradient descent and thus prone to get stuck in local optima. Furthermore the most effective algorithms for simple bandit problems are value function based[16, 17, 18] which translated to our generalizations of VF methods used to train our distributed policy based on a SNN for more complicated MDPs.
Based on these observations we evaluated the SNN in combination with the distributed global VF method and the softmax activation function on other non-sequential tasks. One of these was a task with complex states defining one out of two possible target locations. Here the SNN had to use multiple state populations to deal with this problem. We investigated how well we can deal with this task for different certain input patterns defining the target location. It turned out that we were able to solve almost all of these tasks with a satisfying performance except for the XOR task where the SNN had to react properly to XOR patterns. In the XOR-task we had a significantly worse convergence and we were not able to solve this task correctly. Unfortunately, this reveals an important limitation of our SNN, i.e., the incapability of the SNN to realize XOR mappings if multiple state populations are used due to the loss of expressive power. Since the usage of multiple state populations is absolutely necessary for dealing with high-dimensional state spaces, our perceptron-like two-layered SNN is in general not able to solve tasks like the XOR task. Consequently, the proposed SNN cannot deal with high-dimensional state spaces in a reliable and satisfying way.
Based on the observations gathered from the one-step MDP experiments we also evaluated the SNN with the same method and activation function on a sequential task where the goal was to navigate the end effector of a 10-link robot arm through a two-dimensional grid world. We used a single state population that encoded the grid cell containing the end effector, multiple binary action populations for the values Δq_t added to the joints q_t and a simple reward function based on the distance between the end effector and the target where we assumed instantaneous returns without delayed rewards, i.e., $\gamma = 0$ and $R_t = r_t$. It turned out that this was sufficient to solve this task with a high performance. In summary we showed that our SNN and our learning methods can deal with complex actions, non-complex states and non-delayed rewards. This was sufficient to solve a non-trivial sequential robot arm task with multiple degrees of freedom. However, at the current moment our learning methods cannot deal with delayed rewards which corresponds to $\gamma > 0$ and our SNN can in general not deal with complex state spaces containing certain patterns in reliable way, as we have shown for the XOR task.

7 Future work

For future work we plan to extend the capabilities of our SNN and our learning methods to make it applicable to more challenging tasks. This includes sequential tasks where it is not straightforward to define a simple but still admissible reward function for the case of delayed rewards, i.e., $\gamma > 0$. Simple examples are the mountain-car problem, underactuated pendulum swing-up and cart-pole. In these tasks we rely most certainly on delayed rewards where the discount factor has to be greater than zero such that the reward definition can be kept easy, but still corresponds to a valid solution of the task. Thus, we cannot solve these tasks at the current moment. To deal with delayed rewards we plan to extend our learning dynamics with eligibility traces[41] as done in [1] where they were used to deal with delayed rewards in an actor-only fashion. Additionally to this approach, we are also investigating actor-critic approaches. By making use of a reliable critic, temporal difference errors can be used as simple and admissible rewards for the actor SNN without relying on $\gamma > 0$ on the actor side.

Another class of problems we cannot solve at the current moment are tasks with complex state spaces. An example for such a task is cart-pole where the state space comprises four continuous variables. Such a state space is already too complex to be represented by a single state population. Thus, we rely on multiple state populations. However, this may lead to a loss of expressive power in the sense that it would make us unable to solve the task, as the results for the XOR task in the experiments demonstrated.
Two possible solutions to deal with the problem of losing expressive power for multiple state populations are currently under investigation. The first one is the usage of alternative action encodings. The other possible solution closely related to this idea is the extension of our SNN topology to a multi-layered topology similar to multi-layer perceptrons. Here we would have additional layers of hidden units. These hidden units are binary populations of stochastic spiking neurons that serve as action in one computation step and as state in the next computation step according to the processing flow of a feed-forward network. A picture is shown in figure 7.1. In contrast to classic multi-layer perceptrons we would use populations of stochastic spiking neurons for the hidden units instead of deterministic continuous neurons and train them with our proposed generalized distributed VF methods resulting in reward-modulated Hebbian learning rules. This would differ from typical approaches to train multi-layer perceptrons. Such approaches include black-box optimizers and backpropagation. With our experiments we already found evidence that our learning rules outperform black-box methods and REINFORCE, since black-box optimizers do not exploit the neuron activities as Hebbian learning rules do and REINFORCE as a gradient descent method is prone to get stuck in local optima. The problem of getting stuck in local optima is also an issue for backpropagation which is also based on gradient descent but uses analytic gradients instead which makes it even more unlikely to escape from local optima, since no stochastic dynamics are involved. This is a main reason why backpropagation relies on huge amounts of data to train networks effectively. Furthermore, backpropagation suffers from the vanishing gradient problem in deep multi-layer networks. An extended multi-layered SNN trained with the proposed distributed VF methods resulting in Hebbian learning rules would not suffer from these problems described for the other methods. In particular, the problem of local optima can be tackled easily by adjusting the hyperparameters to give more weight to an exploratory behavior rather than an exploitory. In result such a multi-layered SNN could serve as a universal function approximator that could be trained with powerful reinforcement learning methods and would be an alternative to MLPs trained with backpropagation. Additionally, it would also be an alternative approach to the related framework in [5, 6, 7, 8] where semi-stochastic multi-layer networks are used that employ non-stochastic continuous hidden units trained with backpropagation-like approaches. Thus, we could not only learn mappings from perceptions to motor commands effectively with reinforcement learning but also other mappings like transitions, reward functions or value functions.

In future we also plan to apply our methods to real robot systems. The sequential robot-arm task can also be done with a real robot arm where the actions are real motor commands. In [2] it was already shown that a network consisting of populations of stochastic spiking neurons can be trained from demonstration data to produce desired robot arm trajectories while fulfilling certain constraints without relying on inverse kinematics. However, the approach still relied on inverse dynamics. With our approach we would neither rely on inverse kinematics nor inverse dynamics. However, the main problem would still be to find a proper reward function. This could be done by producing a three-dimensional grid representation of the environment with an RGB-D camera and compute the distance to the target for each grid cell to use these distances for a proper reward definition similar to the approach we used in our experiments. Another possibility would be to make use of an actor-critic approach where the critic could be trained from demonstration data. For the rewards we would use the temporal differences provided by the critic and train the actual actor with them. It is not

necessary that the actor and the critic share the same state spaces. The state space for the actor or the critic can be based on grid cells as we used it in our experiments for our actor SNN or it can be based on visual information. The actor and the critic can both be realized with the proposed SNN and be trained from proper rewards. For a critic negative squared temporal differences could be used as rewards, for instance. If the state space for the actor or the critic is based on visual information it might also be possible to incorporate ideas of shared weights as used in convolutional neural networks to speed up learning.

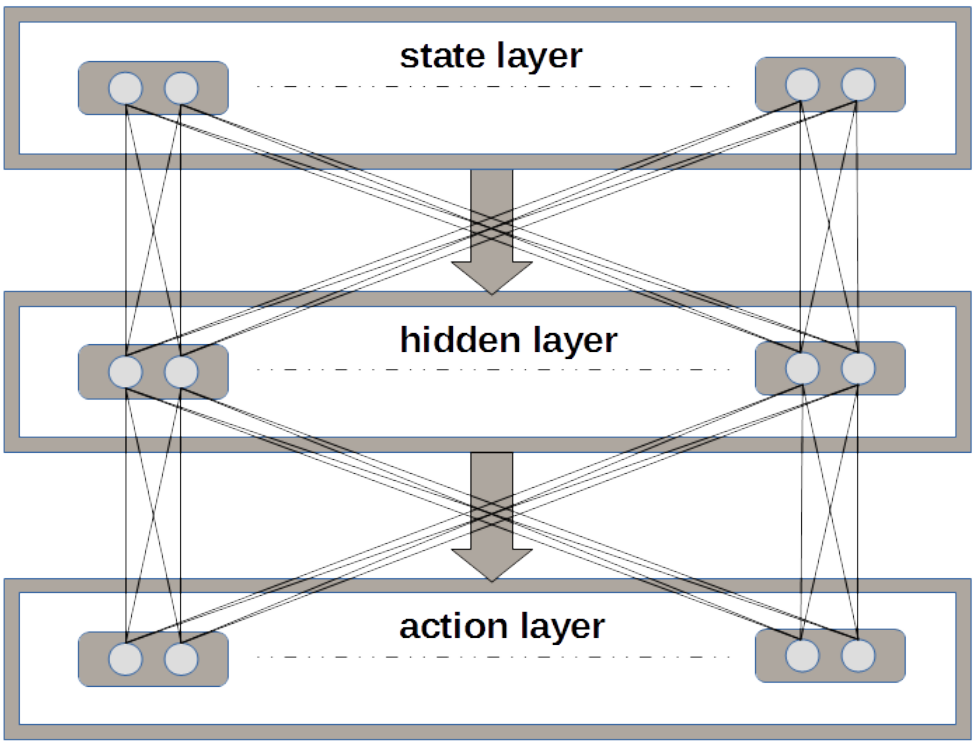

Figure 7.1: For future work we plan to extend the capabilities of our SNN such that it can learn complicated mappings like the XOR function. The picture above shows a possible solution for this problem. The shown network is a multi-layered feed-forward network with a single hidden layer consisting of binary populations of stochastic spiking neurons. According to the processing flow, two computation steps are needed to compute the resulting action for a given state. In the first computation step the hidden layer serves as action layer for the actual state layer and in the second computation step the hidden layer serves as state layer for the actual action layer. This network could be trained with our distributed VF methods in a general reinforcement learning setting based on the activities of all neurons in the network. In particular, we would not rely on black-box methods or backpropagation to train this network.

Bibliography

[1] E. Rueckert, D. Kappel, D. Tanneberg, D. Pecevski, and J. Peters, "Recurrent spiking networks solve planning tasks," *Scientific reports*, vol. 6, p. 21142, 2016.

[2] D. Tanneberg, A. Paraschos, J. Peters, and E. Rueckert, "Deep spiking networks for model-based planning in humanoids," in *Humanoid Robots (Humanoids), 2016 IEEE-RAS 16th International Conference on*, pp. 656–661, IEEE, 2016.

[3] G. E. Hinton, "Training products of experts by minimizing contrastive divergence," *Neural computation*, vol. 14, no. 8, pp. 1771–1800, 2002.

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

[5] P. R. Roelfsema and A. van Ooyen, "Attention-gated reinforcement learning of internal representations for classification," *Neural computation*, vol. 17, no. 10, pp. 2176–2214, 2005.

[6] J. O. Rombouts, A. Van Ooyen, P. R. Roelfsema, and S. M. Bohte, "Biologically plausible multi-dimensional reinforcement learning in neural networks," in *International Conference on Artificial Neural Networks*, pp. 443–450, Springer, 2012.

[7] J. Rombouts, P. Roelfsema, and S. M. Bohte, "Neurally plausible reinforcement learning of working memory tasks," in *Advances in Neural Information Processing Systems*, pp. 1871–1879, 2012.

[8] T. Brosch, F. Schwenker, and H. Neumann, "Attention-gated reinforcement learning in neural networks—a unified view," in *International Conference on Artificial Neural Networks*, pp. 272–279, Springer, 2013.

[9] R. Bellman and R. Corporation, *Dynamic Programming*. Rand Corporation research study, Princeton University Press, 1957.

[10] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*, vol. 1. MIT press Cambridge, 1998.

[11] J. Peters and S. Schaal, "Policy gradient methods for robotics," in *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, pp. 2219–2225, IEEE, 2006.

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

[13] M. P. Deisenroth, G. Neumann, J. Peters, *et al.*, "A survey on policy search for robotics," *Foundations and Trends® in Robotics*, vol. 2, no. 1–2, pp. 1–142, 2013.

[14] M. Deisenroth and C. E. Rasmussen, "Pilco: A model-based and data-efficient approach to policy search," in *Proceedings of the 28th International Conference on machine learning (ICML-11)*, pp. 465–472, 2011.

[15] N. Hansen, S. D. Müller, and P. Koumoutsakos, "Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (cma-es)," *Evolutionary computation*, vol. 11, no. 1, pp. 1–18, 2003.

[16] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, no. 2-3, pp. 235–256, 2002.

[17] J. Vermorel and M. Mohri, "Multi-armed bandit algorithms and empirical evaluation," in *ECML*, vol. 3720, pp. 437–448, Springer, 2005.

[18] V. Kuleshov and D. Precup, "Algorithms for multi-armed bandit problems," *arXiv preprint arXiv:1402.6028*, 2014.

[19] G. A. Rummery and M. Niranjan, *On-line Q-learning using connectionist systems*, vol. 37. University of Cambridge, Department of Engineering, 1994.

[20] C. J. C. H. Watkins, *Learning from delayed rewards*. PhD thesis, King's College, Cambridge, 1989.

[21] G. Tesauro, "Temporal difference learning and td-gammon," *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, 1995.

[22] G. Tesauro, "Practical issues in temporal difference learning," in *Advances in neural information processing systems*, pp. 259–266, 1992.

[23] R. Rojas, *Neural networks: a systematic introduction*. Springer Science & Business Media, 2013.

[24] M. A. Nielsen, "Neural networks and deep learning," 2015.

[25] D. Kriesel, "A brief introduction on neural networks," 2007.

[26] S. S. Cross, R. F. Harrison, and R. L. Kennedy, "Introduction to neural networks," *The Lancet*, vol. 346, no. 8982, pp. 1075–1079, 1995.

[27] C. M. Bishop, *Neural networks for pattern recognition*. Oxford university press, 1995.

[28] D. E. Rumelhart, J. L. McClelland, P. R. Group, *et al.*, *Parallel distributed processing*, vol. 1. MIT press Cambridge, MA, USA:, 1987.

[29] K. Zuse, *Rechnender raum*. Springer-Verlag, 2013.

[30] S. Wolfram, *A new kind of science*, vol. 5. Wolfram media Champaign, 2002.

[31] J. Schiff, "Introduction to cellular automata," 2005.

[32] W. Maass, "Networks of spiking neurons: the third generation of neural network models," *Neural networks*, vol. 10, no. 9, pp. 1659–1671, 1997.

[33] W. Gerstner and W. Kistler, "Spiking neuron models cambridge university press," 2002.

[34] W. Maass, "On the computational power of winner-take-all," *Neural computation*, vol. 12, no. 11, pp. 2519–2535, 2000.

[35] D. Kappel, S. Habenschuss, R. Legenstein, and W. Maass, "Synaptic sampling: A bayesian approach to neural network plasticity and rewiring," in *Advances in Neural Information Processing Systems*, pp. 370–378, 2015.

[36] D. H. Ackley, G. E. Hinton, and T. J. Sejnowski, "A learning algorithm for boltzmann machines," *Cognitive science*, vol. 9, no. 1, pp. 147–169, 1985.

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

[38] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of Control, Signals, and Systems (MCSS)*, vol. 2, no. 4, pp. 303–314, 1989.

[39] D. O. Hebb, *The organization of behavior: A neuropsychological theory*. Psychology Press, 2005.

[40] S. Hochreiter, "Untersuchungen zu dynamischen neuronalen netzen," *Diploma, Technische Universität München*, vol. 91, 1991.

[41] E. M. Izhikevich, "Solving the distal reward problem through linkage of stdp and dopamine signaling," *Cerebral cortex*, vol. 17, no. 10, pp. 2443–2452, 2007.