Graz University of Technology

Thomas Bohnstingl, BSc BSc

# Development of an agent for solving Markov Decision Processes embedded in Spiking Neural Networks

**MASTER'S THESIS**

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Information and Computer Engineering

submitted to

**Graz University of Technology**

Supervisor:
Em.Univ.-Prof. Dipl.-Ing. Dr.rer.nat. Wolfgang Maass

Institute for Theoretical Computer Science
Graz University of Technology, Austria

Kirchhoff Institute
Heidelberg University, Germany

Graz, February 22, 2018

## Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

| Place, date | (signature) |
|---|---|

## Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

| Ort, Datum | (Unterschrift) |
|---|---|

## Abstract

The present master thesis aims to endow a network of spiking neurons with the capability to solve Markov Decision Processes. In recent years, this technology has emerged as a potential alternative to modern computing and can circumvents energy problems or the von Neumann bottleneck.
To provide the desired learning capability to a network of spiking neurons, a light-weight network structure, which allows for an implementation on a resource limited neuromorphic hardware device, is proposed. This approach is then combined with well-studied algorithms from reinforcement learning, to guarantee a good performance.
The thesis starts with a literature research about related work and the learning algorithms used. Based on this background, the network structure is introduced, implemented as a pure software model and finally ported to the dedicated hardware. In addition, the concept of learning to learn is applied to enable learning an entire family of tasks, rather than only a single one. The last part is an extensive evaluation on both developed variants in combination with a comparison.
The results of this work indicate that the performance of the software as well as of the hardware model can compete with known reference algorithms and that the network is able to solve Markov Decision Processes.

## Kurzfassung

Die vorliegende Masterarbeit setzt sich zum Ziel, ein Netzwerk aus spikenden Neuronen mit der Fähigkeit auszustatten, Markov Decision Processes zu lösen. Die genannte Technologie hat sich in den letzten Jahren, aufgrund der hohen Energieeffizienz und der Vermeidung des von Neumann Bottlenecks, verstärkt als Alternative zu modernen Computersystemen erwiesen.
Um ein Netzwerk von spikenden Neuronen mit der erforderlichen Lernfähigkeit auszustatten, wird eine schlanke Netzwerkstruktur, welche auch eine Implementierung auf einer ressourcenlimitierten Hardware ermöglicht, vorgeschlagen. Dieser Ansatz wird mit bereits bekannten und bewährten Algorithmen aus dem Bereich des Reinforcement Learnings kombiniert, um beste Ergebnisse zu erreichen.
Die Arbeit eröffnet mit einer Literaturrecherche über die Behandlung ähnlicher Probleme und der verwendeten Lernalgorithmen. Aufbauend auf diesen Grundlagen wird die entworfene Netzwerkstruktur vorgestellt, in einem Software-Modell umgesetzt und schließlich auf die neuromorphe Hardware portiert. Zudem wird das Konzept von Learning to learn verwendet, um ein schnelleres Lernen von verschiedenen, ähnlichen Problemen zu ermöglichen. Den Abschluss der Arbeit bildet eine umfangreiche Auswertung, in Kombination mit einem Vergleich der Ergebnisse beider vorgestellter Varianten.
Das Ergebnis der Arbeit umfasst, dass die Performance sowohl vom Software- als auch vom Hardware-Modell mit den verwendeten Referenzalgorithmen vergleichbar ist und dass das behandelte Netzwerk in der Lage ist, Markov Decision Processes zu lösen.

## Danksagung

Diese Arbeit widme ich meiner Freundin Christina, die mich mit viel Kraft und Einsatz durch schwere Zeiten während der Anfertigung getragen hat.

Ein besonderer Dank gilt Herrn Professor Maass, für die richtungsweisenden Gespräche und konstruktiven Vorschläge. Des Weiteren gebührt ein großer Zuspruch auch meinem Kollegen Franz, der durch viele Diskussionen und Anregungen zum Gelingen der vorliegenden Arbeit beigetragen hat.

Einen großen Dank möchte ich auch meinen Kollegen aus Heidelberg, Christian, David und Benjamin aussprechen. Ohne ihre Unterstützung bei Anliegen aller Art bezüglich der Hardware und Kommentaren zur Arbeit, wäre ein Abschluss nahezu unmöglich gewesen. Nicht zu Letzt möchte ich meiner Familie und allen Menschen in meinem Umfeld für die Entgegenbringung von viel Verständnis und Hilfe über die gesamte Dauer dieser Arbeit danken.

# Contents

**Abbreviations** **13**

**1 Introduction** **15**
1.1 Overview . . . . . . . . . . 15
1.2 Motivation . . . . . . . . . . 15
1.3 Markov Decision Processes . . . . . . . . . . 16
1.3.1 Definition and Notation . . . . . . . . . . 16
1.4 Related work . . . . . . . . . . 17
1.5 Hardware introduction . . . . . . . . . . 18
1.5.1 Neuron model . . . . . . . . . . 19
1.5.2 Synapse model . . . . . . . . . . 20

**2 Methods** **21**
2.1 Bellman equations . . . . . . . . . . 21
2.2 Temporal Difference Learning . . . . . . . . . . 22
2.3 Temporal Difference Learning with eligibility traces . . . . . . . . . . 23
2.4 Learning to Learn . . . . . . . . . . 25
2.5 Reference Algorithms . . . . . . . . . . 26

**3 Implementation** **27**
3.1 High-level network structure . . . . . . . . . . 27
3.1.1 Details of update rules . . . . . . . . . . 28
3.1.2 Action selection process . . . . . . . . . . 29
3.2 Software implementation . . . . . . . . . . 30
3.3 Hardware implementation . . . . . . . . . . 31
3.3.1 Framework on host computer . . . . . . . . . . 31
3.3.2 Plasticity Processor . . . . . . . . . . 34
3.3.3 Mailbox . . . . . . . . . . 37
3.3.4 Chip calibration . . . . . . . . . . 37
3.3.5 Limitations . . . . . . . . . . 37
3.4 Learning to Learn framework . . . . . . . . . . 38
3.4.1 Crossentropy . . . . . . . . . . 38
3.4.2 Evolution Strategies . . . . . . . . . . 39
3.4.3 Simulated Annealing . . . . . . . . . . 40
3.4.4 Classic gradient descent . . . . . . . . . . 42
3.4.5 Hyperhyperparameters for Learning to Learn framework . . . . . . . . . . 44

**4 Experiments** **45**
4.1 Tasks . . . . . . . . . . 45
4.1.1 MDP . . . . . . . . . . 45
4.1.2 2D Maze . . . . . . . . . . 46

**5 Results** **48**
5.1 Learning to Learn result . . . . . . . . . . 48
5.2 Hardware variability . . . . . . . . . . 49
5.3 Software simulation . . . . . . . . . . 50
5.3.1 Fixed MDP and maze tasks . . . . . . . . . . 50
5.3.2 Random MDPs . . . . . . . . . . 51
5.3.3 Random Maze tasks . . . . . . . . . . 57

5.4 Hardware simulation . . . . . . . . . . 58
5.4.1 Fixed MDP task . . . . . . . . . . 58
5.4.2 Random MDPs . . . . . . . . . . 61
5.4.3 Random Maze tasks . . . . . . . . . . 66
5.5 Comparison . . . . . . . . . . 67

**6 Discussion and Outlook 69**
6.1 Discussion . . . . . . . . . . 69

**A Training time 70**

# Abbreviations

ANN ........ Artificial neural network
CE ........ Crossentroy
CPU ........ Central processing unit
DLS ........ Digital Learning System
EPSP ........ Excitatory postsynaptic potential
ES ........ Evolution Strategies
GD ........ Gradient descent
HBP ........ Human Brain Project
HP ........ Hyperparameters
HHP ........ Hyperhyperparameters
HICANN ........ High Input Count Analog Neural Network
LIF ........ Leaky integrate and fire
LTL ........ Learning to learn
MDP ........ Markov Decision Process
QL ........ Q-Learning with adaptive epsilon
QLF ........ Q-Learning with fixed epsilon
Rnd ........ Random policy
SA ........ Simulated Annealing
SNN ........ Spiking neural network
VI ........ Value Iteration
WTA ........ Winner-take-all

# 1 Introduction

## 1.1 Overview

The aim of this master thesis is to endow a neural network consisting of spiking neurons, with the capability of solving Markov Decision Processes (MDPs). To address this problem, a properly designed feedforward network is combined with well-studied plasticity rules, providing convergence to an optimal solution, in such a way that efficient learning is achieved. To demonstrate the progress of current neuromorphic hardware and to speedup training, an implementation on such a hardware device, developed under the Human Brain Project (HBP), is a sub goal of this thesis. Consequently, the proposed network structure is designed to run as a pure software simulation as well as on the hardware, taking present constraints and limitations into account. Finally, a new aspect of learning is addressed by exploiting the capabilities of Learning to Learn (LTL). This mechanism enables the network to learn not only a single task, but rather from an entire family of tasks, where each new task is learned faster than the previous one.

This thesis is divided into several parts. The first introductory chapter as well as the second chapter provide background information about the problem. After the problem is introduced, the implementation chapter gives detailed insights into the implementation of the hardware and software model and discusses present limitations. Tasks used for evaluation are introduced in the experiment chapter, followed by the result chapter, where results are shown and compared to state-of-the-art reference algorithms used in the literature of MDPs. Finally, the last chapter discusses the results, outcomes and problems which occurred throughout this work.

## 1.2 Motivation

Today, one major limiting part in digital computing is the power consumption and the problem, that Moore's law might not be further fulfilled for the next generations of integrated circuits [Simonite Tom, 2016]. The power dissipation density in a modern processor can already be compared to an electric stove and when shrinking it further, it gets harder to guide it off the chip. Another problem is the so-called von Neumann bottleneck. There the memory is separated from the central processing unit (CPU), which limits the performance, since most of the energy and computing time is spent on transferring data over this bottleneck [Schuman et al.], [Monroe, 2014].
Non von Neumann computing approaches provide a possible alternative to circumvent the problems encountered by modern processors. An instance for this technique are spiking neural networks (SNN). They are fundamentally different from artificial neural network (ANN), because spiking neurons only occasionally emit a spike, whereas in the latter case, the activity of single neurons is continuous. A network constructed with spiking neurons is much more energy efficient, produces less amount of data and removes the barrier between data and computation [Calimera et al., 2013].
The approach of SNNs is inspired by the human brain and uses many neurons interconnected with synapses to mimic the brain. There is a large literature on spiking neural networks and a lot of fundamental research has been done [Ahmed et al., 2014].

However, fundamental questions about how the brain really works are still open and compared to artificial neural networks (ANN). the number of applications for SNNs is very limited. It is harder to design a network of spiking neurons which can be used to solve certain problems compared to a network design of artificial neurons. This is mainly due to the spiking character, which results in a non-differentiable signal. In addition, it is harder to maintain a proper activation within the network and to prevent the network activity from either exploding or dying out. Like the network design it is also more challenging to come up with training algorithms or learning rules since a gradient information is not present. This leads to the fact that the prominent gradient descent learning algorithms, which are the driving forces when dealing with ANNs, cannot be used with spiking neurons. There has been work done, trying to port the gradient descent approach to a network of spiking neurons with so-called random backpropagation [Neftci et al., 2017] or even with a backpropagation variant [Lee et al.]. Unfortunately, in the case of random backpropagation, there is still a rigorous proof missing how this procedure is working and therefore possible corner cases where the algorithm does not work, may not be discovered yet.
The present thesis uses the approach of SNNs and aims at implementing an agent capable of solving MDPs. Already studied learning algorithms for this problem class are ported to SNNs and combined with a properly designed network structure. Equipped with the new LTL approach, finally an implementation on a neuromorphic hardware to address the problems of modern computing is done.

## 1.3 Markov Decision Processes

MDPs are considered as a model for general decision-making processes. In this thesis, discrete MDPs, with a discrete state and action space as well as with discrete time are considered. This means that decisions can only be taken at certain time steps, the agent can only choose an action from a finite set and the environment is also in a particular state from a finite set. With the notation of such a MDP the outcome of a decision, even a random one, can be modelled well.
One crucial property of MDPs is that in a sequence of decisions and the corresponding states $\{s_1, s_2, ..., s_t\}$, the state at timestep $t, s_t$ only depends on the state at the former time step $t-1, s_{t-1}$. This property is called Markov Property and allows a properly designed feedforward network of spiking neurons to solve such decision problems.

In addition, the state might not be fully available to the agent or attached with noise. These scenarios are an extension of the MDP which can be grouped under the term Partially Observable Markov Decision Process (POMDP). However, such problems will not be discussed here in more detail (see [Kaelbling et al., 1998]).

### 1.3.1 Definition and Notation

This section is intended to introduce the notation for MDPs used in this thesis. In general, a MDP can be described by a tuple $(\mathbb{S}, \mathbb{A}, P, R, \gamma)$.

- State space $\mathbb{S}$: At any point in time, the environment is in a particular state $s \in \mathbb{S}$. In general, the state space can either be continuous or discrete, whereas here only discrete state spaces are considered.

- Action space $\mathbb{A}$: The action space contains all possible actions for all states. In a state $s \in \mathbb{S}$ an action $a \in \mathbb{A}$ can be performed. In this thesis the action space $\mathbb{A}$ is the same for each state from the state space. If actions are not applicable in a particular state, the transition probabilities for those actions can be set to zero which effectively leads to the fact that they will not be performed.

- Discount factor $\gamma$: This discount factor is used to describe how important future rewards are in contrast to current ones. This value is in the interval $[0, 1]$, where the value 0 means that only the current reward is relevant and 1 means that each future reward has the same importance.

- Transition matrix $P$: This matrix determines the transition given a state-action pair $(s, a)$. Using the notation that $s'$ is the next state, the elements of this matrix can be interpreted as probabilities:

  $$P(a, s, s') = Pr(s'|a, s) = P^a_{s,s'}$$

  Note that the matrix $P$ has the shape of $(||\mathbb{A}|| \times ||\mathbb{S}|| \times ||\mathbb{S}||)$ and $Pr()$ from the above equation denotes a probability. $P(a, s, s')$ gives the probability for getting into state $s'$ when taking action $a$ in state $s$.
  With this transition matrix, uncertain outcomes of a decision can be modelled.

- Reward matrix $R$: For each transition from one state $s$ to a new state $s'$ involving an action $a$, a reward $R(a, s, s')$ is given. The reward matrix is also of the same shape $(||\mathbb{A}|| \times ||\mathbb{S}|| \times ||\mathbb{S}||)$, where each element represents the reward for the transition from $s$ to $s'$ using action $a$. For consistency reasons, a slightly different notation is used in the following:

  $$R(a, s, s') = R^a_{s,s'}$$

  Throught this thesis, the reward matrix contains only elements in the interval $[0, 1]$.

One example of a MDP with one action and three states is shown in figure 1.1.

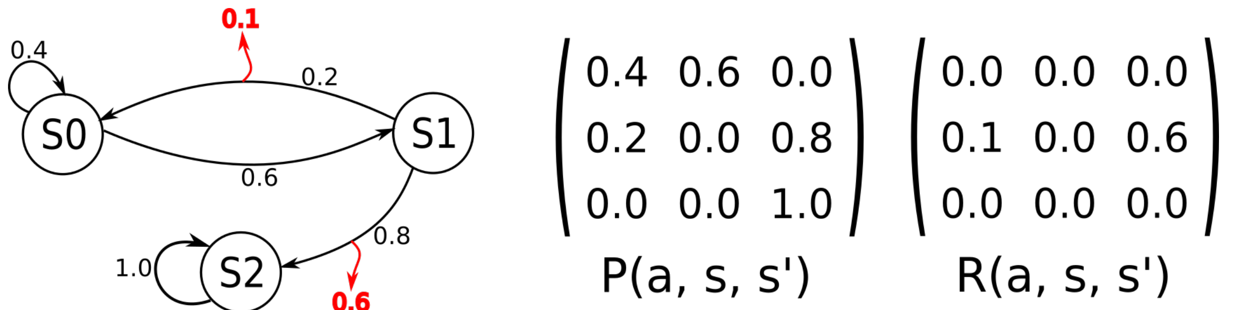


*Figure 1.1: Example MDP with $||\mathbb{A}|| = 1$ and $||\mathbb{S}|| = 3$. On the left-hand side, the graphical representation of the MPD is shown. The transitions are drawn as black arrows, where the transition probability is written in black above the transition arrows. The rewards of a transition are shown as red arrows with the reward value above. Transitions with a probability of 0.0 or rewards with a value of 0.0 are not shown. On the right-hand side, the corresponding transition and reward matrices are shown. Both have a shape of $(1 \times 3 \times 3)$ and since there is only one action, the matrices are represented as $(3 \times 3)$ matrices.*

For the rest of this thesis the term "state" is used to denote a particular state of the MDP environment which is then represented to the network. An action-state pair is a tuple, consisting of a particular state $s \in \mathbb{S}$ and the selected action $a \in \mathbb{A}$ for this state. Consistently the term "state neuron" denotes a particular neuron which is used to express a state of the MDP. The same notation holds true for the term "action" and the term "action neuron".

## 1.4 Related work

There exists a broad literature on applications of SNNs, corresponding learning algorithms and also a lot of biological measured data can be reproduced using certain neuron, synapse and

network structures [Ahmed et al., 2014], [Brea and Gerstner, 2016].
The main focus in this thesis is on one concrete applications where a network of spiking neurons is used to implement an agent, capable of learning a reinforcement problem, namely a MDP. There are few papers in the literature related to a similar problem. One of them was written by Friedrich et. al. [Friedrich and Lengyel, 2016]. In this paper they tackle different MDPs with discrete state and action spaces and one task with a continuous state space and a discrete action space. For a MDP with $|\mathbb{S}|$ states and $|\mathbb{A}|$ actions they use a SNN which can be split into $|\mathbb{S}|$ different clusters consisting of $|\mathbb{A}|$ neurons each. Every cluster represents one particular state and the neuron within such a cluster represent the actions available for this state. In a more compact notation, the $j$-th neuron in the $i$-th cluster of the network represents the state action pair $(s_i, a_j)$. The model uses rate coding, where the rates of the neurons within a cluster represent an approximation of the value function in that state. The neurons coding for the same state are inhibitory connected to each other to be able to solve the Bellman equation and are excitatory to the neurons of the other states. Additionally, it also receives an external input representing the reward for that state-action pair.

A different approach to solve MDP tasks was done by Nakano et. al. [Nakano et al., 2015]. They focused especially on noisy observations and used similar maze tasks for evaluation. To tackle those problems, they mimicked a Restricted Boltzmann Machine (RBM) with a SNN where the binary neurons from the RBM were replaced by Leaky-Integrate and Fire (LIF) neurons. The overall network was then constructed using state neurons and action neurons in the visible layer, hidden neurons and additionally "memory neurons". To endow the network with the ability of using past transitions to improve future ones, the newly introduced "memory neurons" were used. Those neurons extended the network as an additional layer between the state and the hidden neurons, whereas the action neurons in contrast were directly connected to the hidden layer. The memory neurons were recurrently connected to each other with a special weight distribution to maintain activity patterns depending on the past observation sequences.

In parallel to the development of this work, a master thesis related to similar problems was composed by Shein [Shein, 2017]. This work also addressed similar approaches to solve MDPs with spiking neural networks consisting of LIF neurons.
The state-action transition probabilities were encoded into the weights of one population of neurons, the "State and Action" population. To update the Q values according to those weights, a second so-called "Product" population was used which "multiplies" the transition probabilities with the current best action in each state. In the first part of the thesis, the transition probabilities as well as the weights of the "State and Action" and the "Product" population were fixed and statically encoded. The second part deals with essentially learning the transition probabilities and the weights of the two population. For this procedure an error signal between the predicted and the actual state was used. A two-step task was used in order to test the model and compare it to human behavior.

## 1.5 Hardware introduction

The problems of modern computing mentioned in section 1.2 can be avoided by using dedicated hardware for efficiently implementing SNNs. Such a neuromorphic hardware system was developed under the HBP as a co-operation with many different universities [Calimera et al., 2013], [HBP], [Heidelberg].
The so-called "High Input Count Analog Neural Network - Digital Learning System" (HICANN-DLS) is a mixed signal hardware system, where the neurons are implemented as analog circuits. Each neuron circuit can generate a digital signal, a spike, which is then routed over synapse

drivers to the target synapse. At the target synapse this digital signal is converted into an analog signal again and fed to the target neuron.
To enable more complex and custom plasticity rules, the present hardware incorporates a separate dedicated digital processor including a vector unit. This processor has access to the synaptic efficacies as well as to other different pieces of information about the network and the chip (see 1.5.2 for more details).
When writing this thesis, HICANN-DLS version 2 was available which holds 32 neurons, 32x32 synapses and is designed to operate 1000 times faster compared to the biological equivalent.
This introductory chapter about the hardware is intended to give a brief high level overview of the mathematical description of the neuron and synapse model, because an entire description of the hardware alone can fill many papers e.g.: [Friedmann et al., 2017], [Aamir et al., 2016] or [Aamir et al., 2017]. However, the steps necessary to implement an actual experiment on the chip are detailed in chapter 3.

### 1.5.1 Neuron model

The neurons of the hardware are analog circuits and implement the Leaky Integrate and Fire (LIF) neuron model. From a high-level perspective, the digital spikes from other neurons are transformed into currents, through the synapse and then charge a capacitor, representing the current membrane potential. The formulas for the neuron and synapse model are taken from [Stöckel, 2017]. There are also publications about the verification of the theoretical models and the differences to the actual hardware [Stradmann, 2016] [Aamir et al., 2016].
Figure 1.2 shows the internal circuit of the synapse on the neuromorphic chip.

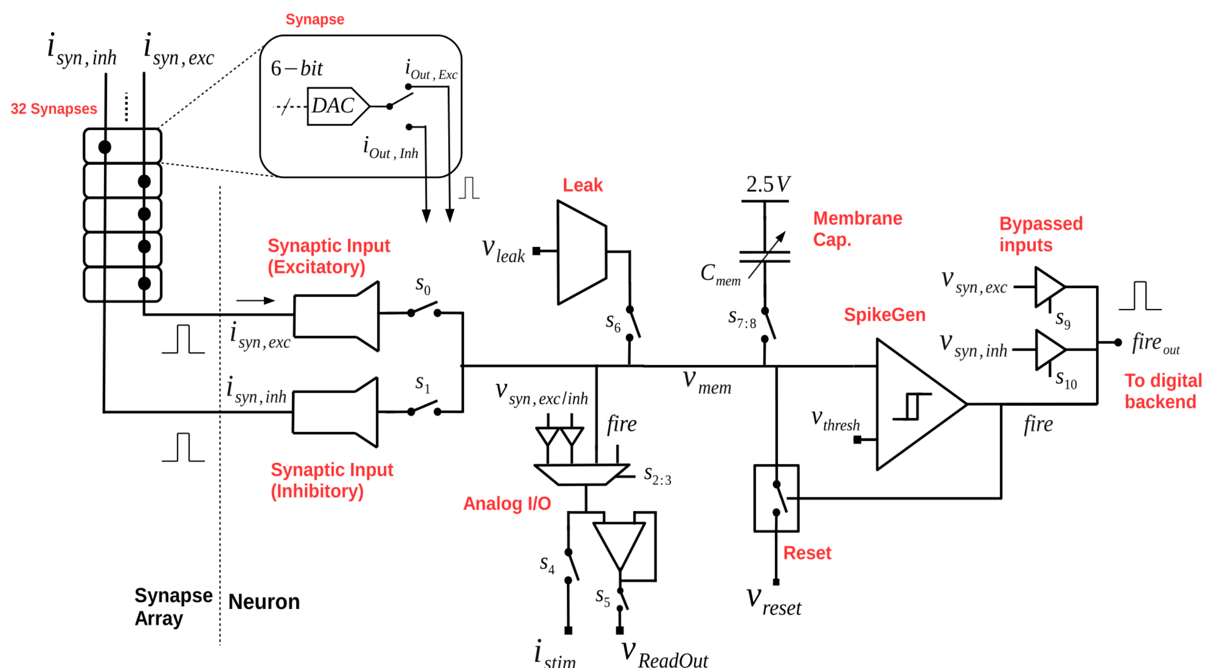


*Figure 1.2: This figure, taken from [Aamir et al., 2016], shows a gross version of the schematics of the HICANN-DLS neuron. On the left hand side, a part of the synapse array is shown, where a black dot represents a created synapse. The neuron has two separate inputs for excitatory and inhibitory currents. If a spike arrives at a neuron, the digital spike is transformed, with the corresponding weight, into an electric current which then charges the membrane capacitor. If the voltage at the capacitor exceeds a preset value, the threshold voltage, a spike is emitted via a dedicated module (In this figure depicted with "Spike Gen"). In addition to that, common mechanisms such as the leak current or the reset voltage are also in place.*

The time evolution of the membrane potential follows the equation:

$$\tau_{mem}\frac{du(t)}{dt} = -[u(t) - u_{leak}] + \frac{I(t)}{g} \tag{1.1}$$

Here $u(t)$ denotes the current membrane potential, $\tau_{mem}$ is the membrane time constant, $u_{leak}$ is the leak potential, $I(t)$ is the total input current to the neuron and $g$ the conductance of the membrane.
When a spike is emitted, the neuron remains at the reset potential $u_{reset}$ for the refractory period $\tau_{ref}$.

$$u(t) = \begin{cases} u_{reset} & t \in [t_{spike}, t_{spike} + \tau_{ref}] \\ u(t) & else \end{cases} \tag{1.2}$$

### 1.5.2 Synapse model

The HICANN-DLS chip implements a current-based synapse model. The digital spike and the weight are combined to the synaptic current:

$$\tau_{syn}\frac{dI_{ij}(t)}{dt} = -I_{ij}(t) + \sum_{t_j} w_{ij}\delta(t - t_j) \tag{1.3}$$

Here $I_{ij}(t)$ denotes the current to neuron $i$ if neuron $j$ fires, $\tau_{syn}$ is the synaptic time constant, $w_{ij}$ is the synaptic weight from neuron $j$ to neuron $i$ and $t_j$ is the spike time of neuron $j$.
Since there can be multiple neurons connected to the target neuron $i$, the currents of them are added up to form the total current.

$$I_i(t) = \sum_j I_{ij}(t) \tag{1.4}$$

$I_i(t)$ denotes the total current to neuron $i$

# 2 Methods

## 2.1 Bellman equations

MDPs have been studied in literature and many different ways about how to tackle those problems emerged. Dynamic programming and reinforcement learning algorithms yield best performance in solving such problems [Bellman, 2010], [Sutton and Barto, 1998]. Behind those algorithmic approaches, a mathematical framework to threat MDPs was developed by Bellman. The following derivations are similar to the one presented by [Richard S. Sutton and Andrew G. Barto] or [Singh and Sutton, 1996].
One can define a policy $\pi(a|s)$ as a probability distribution over actions $a$ in the state $s$. In this work, only deterministic policies $\pi(s)$ are considered. The policy $\pi(s)$ does not yield a probability distribution over actions, but rather a single action when being in the state $s$. This policy is learned to maximize the expected reward over an episode where an episode in this sense denotes a sequence of states, actions and rewards, until a final state or a given number of steps is reached.
Special focus will be set on reinforcement approaches where two key concepts are important, namely the Bellman equation for the State-Value function or simply Value-Function

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.1}$$
$$V^\pi(s) = E\{R_t | s_t = s\}$$
$$V^\pi(s) = E\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\} \tag{2.2}$$
$$V^\pi(s) = \sum_a \pi(a|s) \sum_{s'} P^a_{s,s'}[R^a_{s,s'} + \gamma V^\pi(s')] \tag{2.3}$$

$R_t$ is the total expected reward at time step $t$, $\gamma$ is the discount factor
and the second most important concept, the Action-Value function or the so-called Q-Function:

$$Q^\pi(s,a) = E\{R_t | s_t = s, a_t = a\} \tag{2.4}$$
$$Q^\pi(s,a) = E\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\} \tag{2.5}$$
$$Q^\pi(s,a) = \sum_{s'} P^a_{s,s'}[R^a_{s,s'} + \gamma V^\pi(s')] \tag{2.6}$$

Those equations are following the notation of [Sutton and Barto, 1998]. There is a set of possible policies $\pi(s)$ that may result in different values for the Q-Function or the Value-Function. To determine if a policy $\pi'(s)$ results in a higher expected reward over an episode compared to $\pi(s)$, the Value-Function can be used. If for the policy $\pi'(s)$ the equation $V^{\pi'}(s) \geq V^\pi(s), \forall s \in \mathbb{S}$ holds,

it yields a higher expected reward. This can also be written in a shorter form as $\pi'(s) \geq \pi(s)$. An optimal policy can then be found by searching for a policy $\pi'(s)$ for which $\pi'(s) \geq \pi(s), \forall \pi(s)$ [Richard S. Sutton and Andrew G. Barto], [Szepes ari Richard Sutton, 2010].

The solution of the Bellman equations is known to result in an optimal policy for any given MDP and can be used as a reference method for decision making processes [Bellman, 2010]. However, the solution of those equations requires to solve a system of equations simultaneously of the size of the state space $||\mathbb{S}||$. This process can be done iteratively as it is done by a reference algorithm discussed later 2.5, but requires many steps and a lot of computing effort. Due to the limited computing power of the hardware system used, this approach cannot be implemented.

## 2.2 Temporal Difference Learning

To avoid solving the Bellman equations, Temporal Difference Learning (TD-Learning) can be used. The advantage of TD-Learning comes from the fact that for learning it is not required to play episodes fully to the end, but it rather learn from every single step. This algorithm emerged from Monte Carlo Methods which yield an iterative procedure to update the Value-Function or the Q-Function:

$$V^{\pi}(s) = V^{\pi}(s) + \alpha[R_t - V^{\pi}(s)] \tag{2.7}$$

$$Q^{\pi}(s,a) = Q^{\pi}(s,a) + \alpha[R_t - Q^{\pi}(s,a)] \tag{2.8}$$

where $\alpha$ denotes learning rate.
TD-Learning improves the Monte Carlo Method in terms of computational effort by evaluating the total expected reward $R_t$, from equation 2.7 or 2.8, with only a single step. Although the same calculations can be applied to the Value-Function as well, in this thesis TD-Learning in combination with the Q-Function 2.6 is used to endow the network to learn an optimal policy for any given MDP.
The total expected reward can be formulated as:

$$\begin{aligned} R_t &= \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \\ &= \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \\ &= r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \\ R_t &= r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) \end{aligned} \tag{2.9}$$

This results in a modified update rule from the Monte Carlo Method:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)] \tag{2.10}$$

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_{t+1} + \gamma \arg\max_a Q(s_{t+1},a) - Q(s_t,a_t)] \tag{2.11}$$

Since the Q-Function is estimated for the current policy $\pi(s)$, this super script is omitted. In addition, the time index is reintroduced to emphasize the time relation of the individual components in equation 2.11. From equation 2.10 it can be observed, that the update procedure

requires the next action to be selected. To do this, we can remind ourselves that the Q-Function represents the total expected reward from state $s$ and action $a$ (see equation 2.4). The selection process can then be done by choosing the action with the highest Q-Value in state $s$ which effectively leads to 2.11. This update rule is also known under the term Q-Learning in literature [Watkins and Dayan, 1992].
The pseudocode for TD-Learning can be formulated as:

**Algorithm 1** TD-Learning

1: **procedure** TD($\gamma$, $\alpha$)
2: Choose $a_t$ using current $\epsilon$-greedy policy
3: Take action $a_t$, observe $r_{t+1}$ and $s_{t+1}$
4: Choose $a_{t+1}$ using $\arg\max_a Q(s_{t+1}, a)$
5: $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$
6: $s_t \leftarrow s_{t+1}$
7: **end procedure**

An $\epsilon$-greedy policy $\pi'$ does not only take the action proposed by $\pi'(s)$, but also picks a random action with a probability of $\epsilon \in [0, 1]$.
This pseudocode shows only the weight update rule. The initialization of the Q-Values, the initialization of the first state and the loop until the maximum number of steps is reached are not shown.

## 2.3 Temporal Difference Learning with eligibility traces

The trade-off between Monte Carlo Method using a complete episode for learning and TD-Learning, using only a single step, is Temporal Difference Learning with Eligibility traces (TD($\lambda$)-Learning). Convergence speed of TD-Learning can be improved when using more than just a single step for learning. The update formula can be derived in a similar manner as before. The sum to compute the total expected reward given in equation 2.1, is expanded for more than two terms and again the Q-Function of the state $s_{t+\lambda}$ is used to estimate the remaining reward. How many steps are considered for learning can be specified with a parameter $\lambda \in [0, 1]$, which also gives rise to the name of the learning algorithm and obeys two limits:

$$\lambda = \begin{cases} 1 & \text{A complete episode is considered, resulting in the Monte Carlo approach} \\ 0 & \text{Only a single step is considered, resulting in the TD-Learning approach} \end{cases}$$

According to the described procedure the total reward expands to:

$$\begin{aligned} R_t &= \sum_{k=0}^{\lambda} \gamma^k r_{t+k+1} \\ &= r_{t+1} + \gamma r_{t+2} + ... + \gamma^{\lambda-1} r_{t+\lambda} + \gamma^{\lambda} \sum_{k=0}^{\infty} \gamma^k r_{t+(1+\lambda)+k} \\ R_t &= r_{t+1} + \gamma r_{t+2} + ... + \gamma^{\lambda-1} r_{t+\lambda} + \gamma^{\lambda} Q(s_{t+\lambda}, a) \end{aligned} \tag{2.12}$$

The resulting expression in equation 2.12, can again be inserted into the Monte Carlo formula 2.8 to obtain the TD($\lambda$)-Learning rule (similar [Szepes ari Richard Sutton, 2010]). However, this forward view would again require knowledge about future rewards ($\{r_{t+2}, ..., r_{t+\lambda}\}$) collected by playing a certain amount of steps like in the case of the Monte Carlo algorithm. To overcome

this, a backward view is introduced by so-called eligibility traces $e_t(s,a)$. There is a separate eligibility trace per individual state-action pair $(s,a)$ which indicates how important this state and action is for the previous ones. Using those eligibility traces, the resulting update rule can be formulated as:

$$e_t(s,a) = \begin{cases} \gamma\lambda e_{t-1}(s,a) + 1 & \text{if } s = s_t \text{ and } a = a_t \\ \gamma\lambda e_{t-1}(s,a) & \text{otherwise} \end{cases} \tag{2.13}$$

$$\delta_t = r_{t+1} + \gamma \arg\max_a Q(s_{t+1},a) - Q(s_t,a_t) \tag{2.14}$$

$$Q(s_t,a_t) = Q(s_t,a_t) + \alpha\delta_t e_t(s,a) \tag{2.15}$$

One can check the above-mentioned limits for those formulas. If $\lambda = 0$, the eligibility trace is 0 for all state-action pairs, except for the one currently considered for updating, where it is 1. In case of $\lambda = 1$, the trace accumulates the visits for the state-action pair $(s,a)$ effectively resulting in the Monte Carlo method.

The pseudocode for TD($\lambda$)-Learning can be formulated as:

**Algorithm 2** TD($\lambda$)-Learning

```
1: procedure TDLambda(γ, λ, α)
2:     Choose a_t using current ε-greedy policy
3:     Take action a_t, observe r_{t+1} and s_{t+1}
4:     Choose a_{t+1} using arg max_a Q(s_{t+1}, a)
5:     δ ← r_{t+1} + γQ(s_{t+1}, a_{t+1}) − Q(s_t, a_t)
6:     e_t(s_t, a_t) ← e_t(s_t, a_t) + 1
7:     for all pairs (s_t, a_t) do
8:         Q(s_t, a_t) ← Q(s_t, a_t) + αδe_t(s_t, a_t)
9:         e_t(s_t, a_t) ← γλe_t(s_t, a_t)
10:    end for
11:    s_t ← s_{t+1}
12: end procedure
```

Again this pseudocode only shows the weight update rule, where the initialization and the loop over all steps is done outside.

Both Q-Learning and Q-Learning with eligibility traces are chosen as learning algorithms, because they are proven to converge to the optimal solution of any given MDP [Dayan and Sejnowski, 1994], [Dayan, 1992]. The derived learning rules, in formulas 2.11 and 2.15, provide different parameters, marked in red, which need to be chosen properly. They can take arbitrary values between $[0,1]$, but influence convergence speed or stability behavior of practical implementations.
For the rest of this thesis, the term TD-Learning or TD(1)-Learning corresponds to Q-Learning without eligibility traces and TD($\lambda$)-Learning to the version of the Q-Learning with eligibility traces.

## 2.4 Learning to Learn

Learning to Learn is a computing paradigm that is inspired by the human brain and was recently used to yield promising results [Andrychowicz et al., 2016], [Chen et al.], [Duan et al.]. The human brain is able to learn different tasks efficiently and is able to overtake experiences and knowledge from previously learned tasks, to new similar ones, which improves learning speed.
The general idea of LTL is to

- define a family of learning tasks which share some common concept.
- define a fitness, indicating how well the agent is able to perform this task.

Humans also face such situations throughout their lives, for example a family of tasks is riding two-wheeled vehicles, like bicycles and motorcycles. The fitness in this case is how well the human being is able to ride the vehicle. The essence of LTL is the transfer of knowledge from one task to another similar one from the same family. In the given example the human is able to learn how to ride a motorcycle faster, once he has already learned how to ride a bicycle.

From an algorithmically perspective, the agent which should learn a given task is parameterized with so-called hyperparameters $\boldsymbol{\theta}$. Here, the parameters of the agent can range from learning rates to more complex parameters of a neural network or of an underlying hardware.
The agent itself is not limited to any particular model and can therefore be implemented as a software neural network, a hardware chip or any arbitrary model capable of learning a task. In the case of this thesis, the agent is a neural network in combination with a learning rule, targeting to solve MDPs. It learns a given task $\mathcal{T}$ from the family $\mathcal{F}$ based on the hyperparameters $\boldsymbol{\theta}$ and produces a fitness accordingly. If a new, similar task $\mathcal{T}' \in \mathcal{F}$ is given to the agent, the same hyperparameters might not yield a good performance. The hyperparameters $\boldsymbol{\theta}$ must then be optimized to yield an appropriate performance for both tasks, $\mathcal{T}, \mathcal{T}'$.
Figure 2.1 depicts the described structure of LTL. From a high-level perspective the structure can be split into two separate parts. The first part is the optimizee which is the agent learning a given task with the learning rule parameterized by $\boldsymbol{\theta}$. The second part involved is the optimizer which is in charge of optimizing the hyperparameters of the optimizee, such that learning a new task from the same family $\mathcal{F}$ is improved in terms of learning speed.

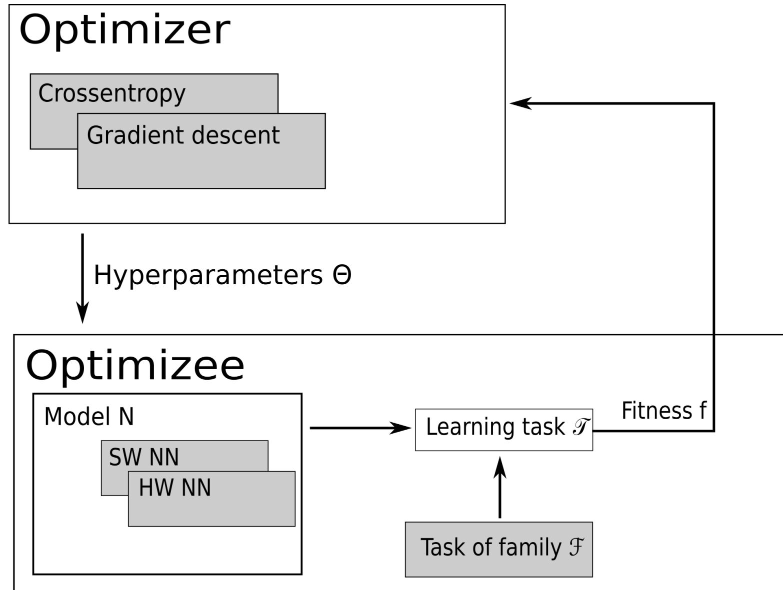


*Figure 2.1: Structure of the LTL paradigm. The approach consists of two components interacting with each other. The optimizee (in the lower part of the figure) represents the agent, learning a given task $\mathcal{T}$. The agent is allowed to be any model with learning capabilities, whereas the applied learning rule is parameterized by hyperparameters $\boldsymbol{\theta}$. The optimizee receives $\boldsymbol{\theta}$ as an input, draws a new task $\mathcal{T} \in \mathcal{F}$ and produces a fitness value $f$ as an output. This fitness is then fed into the second part, the optimizer (shown in the upper half of the figure). In this thesis the optimizer is an optimization algorithm that receives the fitness $f$ as an input and produces a proper new hyperparameter set $\boldsymbol{\theta}$ for the optimizee. This new hyperparameter set should then improve learning speed for new tasks $\mathcal{T}'$ from the family $\mathcal{F}$*

An often encountered application in machine learning is to optimize parameters $\boldsymbol{\theta}$ for a particular task, which is usually done by hand. Using the introduced notation, this can be achieved by defining a family $\mathcal{F}$ which holds only a single task. The optimizer then tries to optimize the fitness value of the single task with respect to the parameters $\boldsymbol{\theta}$. However, it has to be mentioned that this approach allows the agent to learn a whole family of tasks rather than particular single task.

## 2.5 Reference Algorithms

The performance of the implemented network structure needs to be evaluated. In order to do this, the results are compared to three different reference algorithms from literature:

- **Value Iteration (VI):** Iteratively solves equation 2.6 and produces Q-Values for each state in each iteration. This algorithm essentially solves the Bellman equation iteratively therefore converges to the optimal policy and can be seen as the optimal algorithm.
- **Q-Learning with fixed epsilon (QLF):** Implements Q-Learning as given in equation 2.11. The learning rate stays constant during training and is chosen by hand upfront.
- **Q-Learning with adaptive epsilon (QL):** Implements Q-Learning in a slightly modified way. The learning rate is adaptive and changes during training. This mechanism is thought to provide better performance as the training time evolves.
- **Random policy (Rnd):** A Random policy algorithm is also implemented in order to show the performance when only guessing actions. This algorithm chooses a random policy and does not apply any further learning or update to it.

# 3 Implementation

The theoretical model of the learning algorithm presented in chapter 2 will be implemented on two different platforms. The first platform is a pure software simulation and the second aims at taking the advantages of a neuromorphic hardware into account. To have a unified framework, the structure of the network is designed under the given requirements and restrictions of the hardware:

- Operate mainly on the level of single or few spikes
- Operate on deterministic neurons, since the hardware per se has deterministic neurons
- Scale down to the limit of 32 neurons, while still provide the possibility to learn non trivial tasks

## 3.1 High-level network structure

From an abstract point of view, the proposed network uses the learning algorithms derived in section 2.2 or 2.3, because they are known in literature and there exists a proof that those algorithms reach an optimal solution for a given MDP (see [Dayan and Sejnowski, 1994] or [Dayan, 1992] for details).
The network consists of two fully connected layers of spiking neurons. The first layer consists of $n = ||\mathbb{S}||$ neurons, also called state population, where $\mathbb{S}$ denotes the state space. Each neuron of this layer represents a single state $s_i \in \mathbb{S}$.
The second layer consists of $m = ||\mathbb{A}||$ neurons, also referred to as action population, representing all possible actions $a_i \in \mathbb{A}$, where $\mathbb{A}$ denotes the action space.
The synapses for each neuron in the first layer can be interpreted as a connection from state $s_i$ to all action neurons $a_j$. With this structure, only a total amount of $||\mathbb{S}|| + ||\mathbb{A}||$ neurons are needed which fulfills the scalability constraint.

When comparing the structure to the Q-Function presented in 2.2 or in 2.3, one can find that the efficacies of the different synapses could represent the Q-Values $Q(s_i, a_j)$ or a scaled version of it. To have a consistent notation with literature, the weight of the synapse connecting the j-th action neuron with the i-th state neuron is denoted by $w_{a_j,s_i}$. For the rest of this thesis, this is equivalent to $w_{ji}$, where the first index indicates the action neuron and the second index the state neuron.

### 3.1.1 Details of update rules

The two derived update rules are stated again for convenience:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \textcolor{red}{\alpha}[r_{t+1} + \textcolor{red}{\gamma} \arg\max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{3.1}$$

$$e_t(s,a) = \begin{cases} \gamma \textcolor{red}{\lambda} e_{t-1}(s,a) + 1 & \text{if } s = s_t \text{ and } a = a_t \\ \gamma \textcolor{red}{\lambda} e_{t-1}(s,a) & \text{otherwise} \end{cases} \tag{3.2}$$

$$\delta_t = r_{t+1} + \textcolor{red}{\gamma} \arg\max_a Q(s_{t+1}, a) - Q(s_t, a_t) \tag{3.3}$$

$$Q(s_t, a_t) = Q(s_t, a_t) + \textcolor{red}{\alpha} \delta_t e_t(s,a) \tag{3.4}$$

Where the red marked variables indicate the hyperparameters considered for Learning to Learn.

The term $Q(s_t, a_t)$ in both equations is represented by the current synaptic weight $w_{ji}$ connecting the state neuron $s_i$ to the action neuron $a_j$ at time step $t$. The same also holds for the term $Q(s_{t+1}, a_{t+1})$ at timestep $t+1$, whereas this may address a different synaptic weight, since the tuple $(s_{t+1}, a_{t+1})$ can be different from the tuple $(s_t, a_t)$.
The reward $r_{t+1}$ is given as a global signal from the environment and is accessible for all synapses when needed.
The term $\arg\max_a Q(s_{t+1}, a)$ is nonlinear and responsible for performing the "off-policy" action selection by choosing the highest Q-Value for the state $s_{t+1}$ depending on the action $a$. In the proposed setup, this operation is carried out by the network itself. This is because the action neuron $a_j$, whose synaptic weight is the highest, compared to all other action neurons connected to this state, fires a spike before the other action neurons. Since the action neurons are also inhibitory connected to each other, a spike from neuron $a_j$ reduces the spiking probability for all other action neurons $a_i \in A, i \neq j$ effectively resulting in the nonlinear $\arg\max_a$ operation.
Note that the eligibility traces required for TD($\lambda$)-Learning are not problematic and stored in memory. It has also to be mentioned that on hardware only 8-bit values for the parameters of the learning rule can be used. Although the Plasticity Processing Unit (PPU), see chapter 3.3.2, itself is capable of computing with higher precision, the main bottle neck was the available memory which only allowed for 8-bit parameter values.

The state of the MDP is indicated by a spiking state neuron $s_i$ which is self-excitedly connected to itself as well as connected to the action neurons $a_j \in \mathbb{A}$. Depending on the weight values, one or multiple action neurons might eventually emit a spike which can be interpreted as the action for this particular state. In general, such a model would already produce actions depending on the current state but at arbitrary time steps. To be able to use Q-Learning as introduced, actions are only read within a defined action selection window to get discrete update times. Figure 3.1 shows a simple version of the described network with 3 state neurons and 4 action neurons.

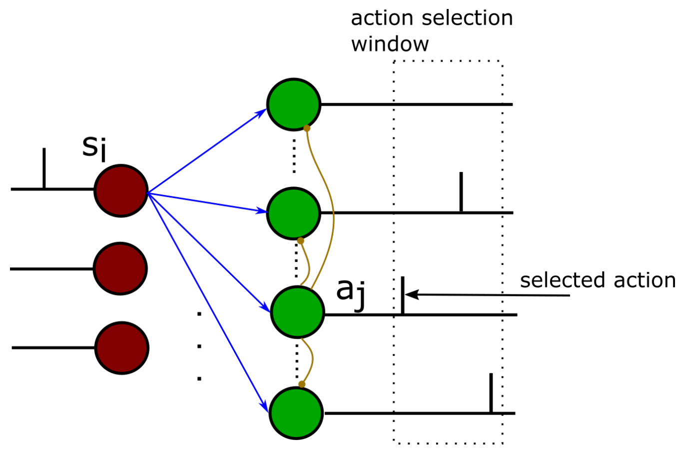


*Figure 3.1: High-level network structure which operates on the level of few spikes and where Q-Values are represented as the synaptic efficacies. The red neurons on the left-hand side build up the state population to represent the current state. The states are one-hot encoded, meaning that for each state $s \in \mathbb{S}$ exactly one of the state neurons is active, i.e. if state $s_i$ is represented, the i-th neuron from the left population is active, while all others remain silent. The green neurons form the action population and represent the possible actions $a \in \mathbb{A}$. A single or few spikes are emitted by the current state neuron $s_i$ to stimulate the action neurons. Depending on the weights, one the j-th action neuron eventually emits an output spike which is then considered as the taken action $a_j$ in state $s_i$.*
*The action for the state $s_i$ is selected as the first spike of an action neuron $a_j$ within the action selection time window. The reward from the environment is integrated into the according synaptic weight update. The action neurons are inhibitory connected to each other to suppress the other actions. If there is no spike within this time window, a random action is selected.*

### 3.1.2 Action selection process

To better understand the action selection procedure of the network, one considers a general MDP with a set of states $\mathbb{S}$, whereas the agent is in an arbitrary state $s_i \in \mathbb{S}$ and a set of actions $\mathbb{A}$. To perform an interaction with the environment, the agent must be able to come up with an action given the state $s_i$, which is done in the following way:

1. The current state is presented to the network as a one-hot vector, a zero-vector of length $||\mathbb{S}||$ with excatly one 1 entry. This means that for each state $s \in \mathbb{S}$ there is exactly one neuron from the state population active. The one-hot vector for $s_1$ looks like $\begin{pmatrix} 0 & 1 & 0 & \dots & 0 \end{pmatrix}^T$
2. The action of the network is selected as the first output spike produced by the action neurons within a predefined time window (action selection window). This action selection window starts after the state neuron spiked and ends at a defined time (maximum wait time).
3. If there was no spike within this time window, or all action neurons spiked exactly at the same time, a random action is selected.
4. The update of the corresponding weight using the reward from the environment is performed.
5. The next state is presented to the network and the action selection process starts again from point 1.

## 3.2 Software implementation

To model the SNN, the neural simulator NEST with a Python binding is used. The network models the explained structure in a straight forward way and uses LIF neurons for this purpose. The programming language Python was selected for various reasons. First of all, it allows for fast prototyping with convenient debugging capabilities. Furthermore, it is independent of the underlying operating system and the platform, which makes it perfectly suitable to create a portable implementation for the hardware from Heidelberg. Finally the Learning to Learn framework, explained in a later section 3.4, was also developed in Python and can therefore be smoothly integrated.

Available libraries for the maze and MDP environments are modified to allow for an easy adaption of the task difficulty. For example, it is possible to generate new random MDPs with different sizes of state and action spaces, or to generate new mazes with various sizes.

It has to be mentioned that the Q-Learning algorithm cannot be implemented in an online manner using only the NEST Python binding, without modifying the NEST kernel. To address this issue a batch mode is used, where the weight update is performed after each action selection process done by the network. For example, the network is in a particular state $s_i$, selects action $a_j$ and gets into the next state $s_k$ while receiving the reward $r$. Before continuing with the next state, the update rule is applied and the synaptic weights are updated.

## 3.3 Hardware implementation

When discussing the hardware, additional components beside the HICANN-DLS itself are needed. In figure 3.2 one can see the overall system architecture and the main components involved in a hardware experiment.

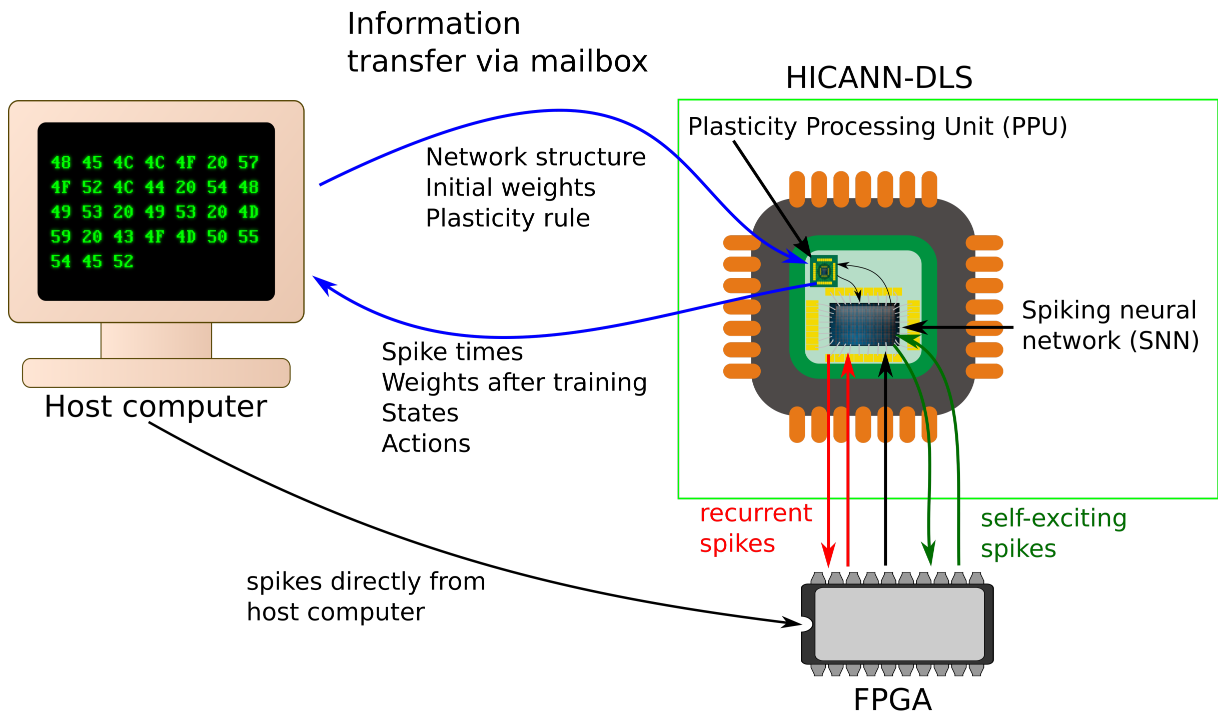


*Figure 3.2: General overview about involved components when performing experiments with the HICANN-DLS. There are mainly three components involved: A conventional host computer, a FPGA circuit as well as the HICANN-DLS chip itself. The communication between the different components are performed over different channels. The host computer can be programmed in standard Python 2 and is responsible for the setup of the network structure, the initial synaptic weight setup as well as the preparation of the mailbox (see 3.3.1). The mailbox is a memory area within the HICANN-DLS accessible from the host computer and can be used to exchange information as well as results (blue arrows). The real hardware, the HICANN-DLS, holds basically two components. One of them is the digital processor (PPU) used for synaptic plasticity and the other one is spiking neural network (SNN) part. The PPU (see 3.3.2) implements the environment for the current task, as well as the plasticity rule and is also able to send spikes to the SNN. In addition to the host computer and the HICANN-DLS chip there is also a FPGA involved. This module allows sending spikes to the SNN directly from the host computer (black arrows) and is also responsible for routing self-exciting spikes of the neurons within the SNN (green arrows) and recurrent spikes (red arrows). For example, with this module it is possible that neuron $i$ sends a spike to any other neuron, including itself. In the present hardware version, also spikes from one neuron $i$ to another neuron $j \neq i$are routed via the FPGA circuit (red arrows).*

### 3.3.1 Framework on host computer

The implementation on the host computer is needed to prepare the network structure, to set the initial conditions and to evaluate the results after the experiment on the chip is done. In contrast to the software model, the network structure is implemented by setting entries in a so-called synapse array. This is a 32 x 32 matrix, whereas each element $(i, j)$ indicates a synapse from neuron i to neuron j. In the hardware implementation, a synapse holds a synaptic efficacy as well as a synaptic address:

- Synaptic efficacy: The weight of the synapse $w_{ji}$ is used to represent a scaled version of the Q-Value $Q(s_i, a_j)$ as in the software model.

- Synaptic address: The addresses of the individual synapses are used to build network structures.

The address of the synapse is important when implementing network structures and is new compared to the NEST software model. Another interpretation of the synapse array is that the 32 rows represent synapse drivers and the 32 columns represent individual neurons. Spikes are injected by the 32 synapse drivers and might come from the PPU, from the network itself or from the host computer via the external FPGA circuit.
Figure 3.3 depicts a simple example to illustrate the concept of synapse drivers and synapse addresses. Every spike transmitted to the network has an address assigned. If this spike address matches the address of a synapse connected to the same synapse driver, the spike gets translated into a current, according to the weight of the synapse and finally transmitted to the target neuron. If the address between the spike and the synapse does not match, or the synapse is not connected to the same synapse driver, the spike is not transmitted to the neuron.

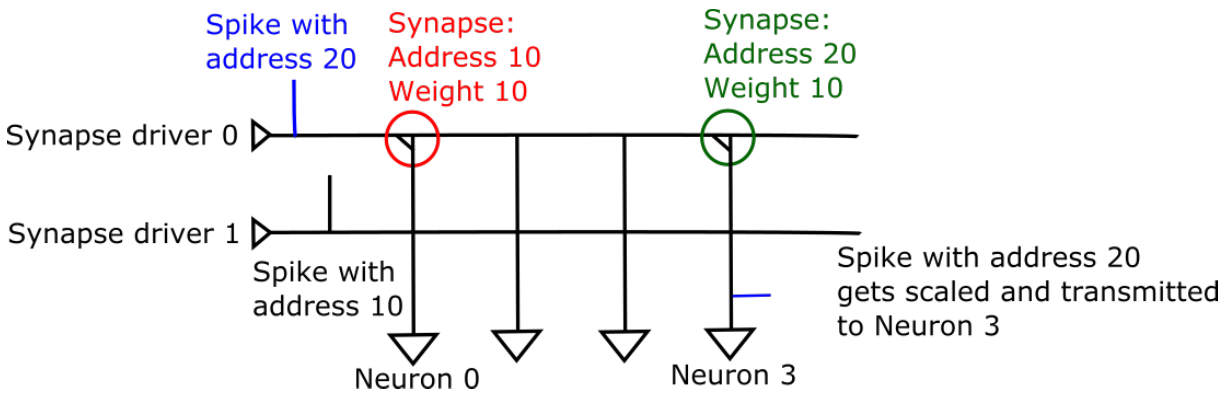


*Figure 3.3: This figure shows two synapse drivers drawn as horizontal lines, four neurons drawn as vertical lines and two synapses connecting the neurons with the synapse drivers. The one incoming spike is injected by the synapse driver 0 and has address 20 assigned (blue spike). The two synapses, connected to the same synapse driver also have a particular weight and an address. Only if the address of the synapse matches the address of the spike (green synapse), the spike gets scaled and transmitted to the neuron. The spike injected by the driver 1 is not propagated further to any neuron, because there is no synapse with the correct address connected to this synapse driver.*

Note that the synapse driver additionally controls whether inhibitory or excitatory synapses are formed. Per default all synapse drivers are set to be excitatory. To use inhibitory connections with the chosen routing configuration, a synapse driver, corresponding to an entire row in the weight matrix, must be reconfigured. Effectively this means that a particular neuron can either have all excitatory or inhibitory outgoing connections.

To enable the full connection matrix, each neuron is per default connected to its dedicated synapse drivers. This means that neuron $i$ is connected to synapse driver $i$ via an external FPGA circuit. A spike from neuron $i$ to itself is in this context denoted as a self-exciting spike. In biology this self-excited synapse is referred to as an autapse [Wang et al., 2017].
The term recurrent spikes is also somewhat differently used in the context of the hardware than one might expect. The spike from neuron $i$ to any neuron $j$ is already denoted as a recurrent spike. The reason for this is that both, self-exciting and recurrent spikes are not transmitted within the chip itself, but are instead routed using the external FPGA. Therefore, the spike leaves the network and is recurrently fed back by the FPGA circuit. This mechanism introduces some limitations in terms of the possible connection matrix and the usability of networks which will be further discussed in section 3.3.5.

In figure 3.4 the network structure network for a MDP problem using 3 states and 2 actions is

demonstrated. One can see the self-exciting connections from the state neurons to themselves and the connections from the state neurons to the action neurons. The weight of the synapses used to connect the state and the action neurons are again interpreted as scaled versions of the Q-Values.

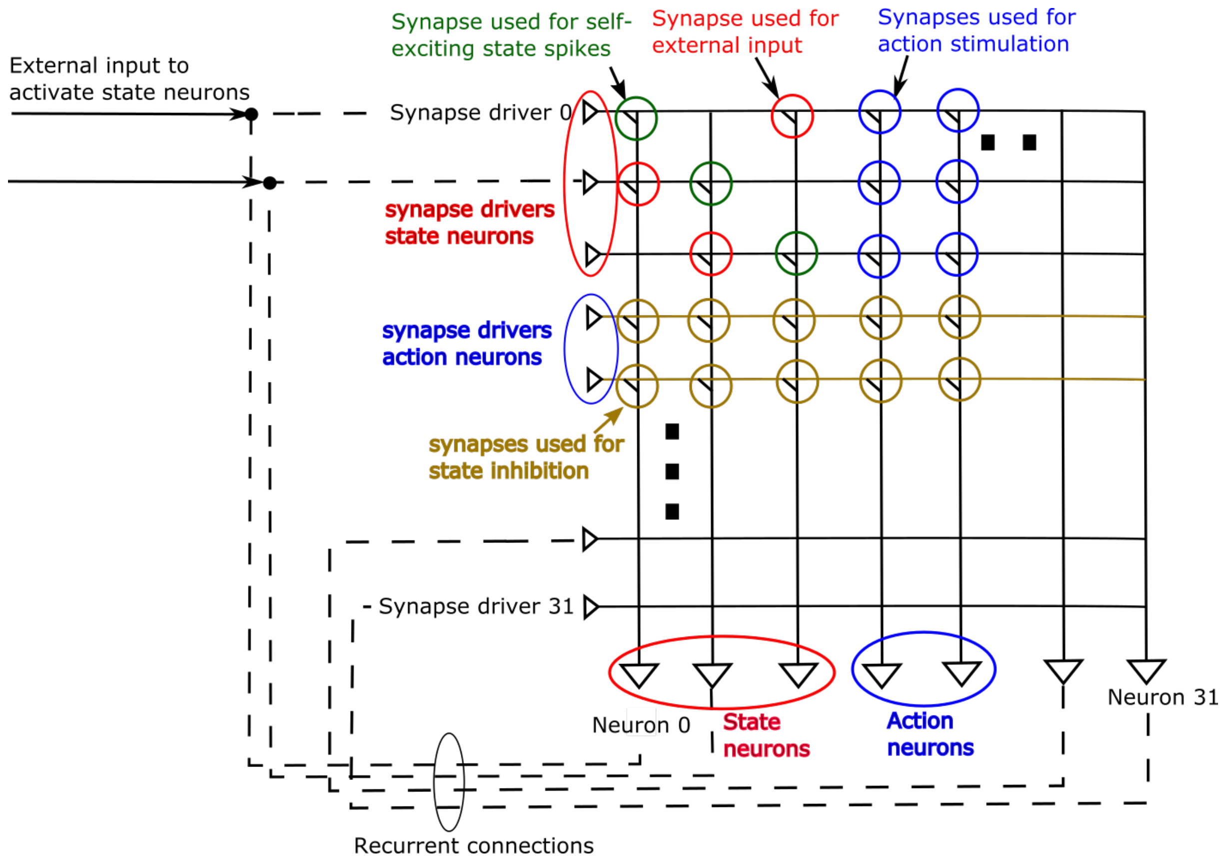


*Figure 3.4: Network structure for a simple MDP with $||\mathbb{S}|| = 3$ states and $||\mathbb{A}|| = 2$ actions. The network is arranged as a 32 x 32 matrix whereas each horizontal row represents a synapse driver and each vertical column represents an actual neuron. The synapse driver, here on the left-hand side, are responsible for injecting spikes, sent from the PPU or the host computer (indicated by black arrow on the left-hand side) as well as for injecting self-exciting spikes from the neurons itself (indicated by dashed lines).*
*The black bars at the intersections between the synapse driver and the neuron line correspond to single synapses. Each synapse holds a 6-bit weight value between 0 and 63 as well as an address. The address can be used to distinguish if the spike approaching from synapse driver $i$ is relevant for neuron $j$. If there is no synapse connecting synapse driver $i$ and neuron $j$, this means that the weight is zero and no spike will be transmitted.*
*The PPU uses the synapse driver of the state neurons to indicate the current state of the environment, in particular this is done via the red synapse drivers and the red synapses. For example, if the environment is in state zero, the PPU will send a single spike to the synapse driver 1 with the address of the red synapse. The green synapses are used for the self-exciting connections of the state neurons. The weight of both, the red and the green synapse, are fixed throughout the entire run.*
*The blue marked synapses are used to activate the action neurons when the corresponding state neuron spikes. These weights are learned during training. Depending on the weights, one action neuron might fire before the other ones which is then the selected action for the current state. Finally, the action neurons inhibit both, the state and the other action neurons which is done via the brown synapses. Note that the synapse drivers of the action neurons are set to inhibitory, indicated by the brown color.*

A detailed step-by-step description of the ongoing processes when selecting an action in a given state, can be found in section 3.3.2.
In addition to the configuration and the setup of the experiment, the evaluation of the learned

Q-Values and the achieved results needs to be done on the host computer. Therefore, the spike times, the weights, the states and the chosen actions are transferred from the chip to the host part. Based on the initial state, the transition and the reward matrices, the reference algorithms are trained and compared to the network, based on the cumulative reward.

To give a better overview, the actual hardware and the board are depicted in figure 3.5.

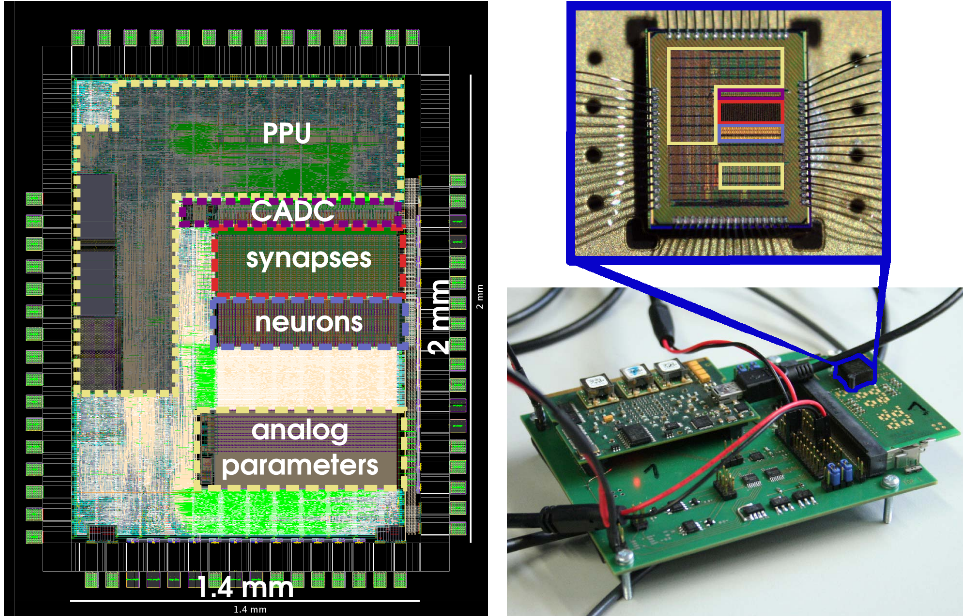


*Figure 3.5: Taken from [Hartel, 2016] and [Friedmann et al., 2017]. The left-hand side shows the floor plan of the HICANN-DLS and the separate modules on the chip. The overall dimensions are approximately* 2 mm *by* 1.5 mm*. The synapse array is marked in red followed by the neuron array in violet. The analog synapse parameters are stored in the yellow marked analog parameter area, whereas the weights are stored inside a digital part in the PPU.*

### 3.3.2 Plasticity Processor

The plasticity processing unit (PPU) is mounted on the HICANN-DLS beside the neural network and can be used to implement more complex synaptic update rules. It is an open-source digital processor with an additional vector processing unit capable of processing up to 16 synapses at once to increase performance [Heidelberg], [Stöckel, 2017]. Since the chip has to interact with the environments and no dedicated hardware to emulate these environments is available, they are also emulated by the PPU. A high-level overview of the software structure and the information flow inside the HICANN-DLS can be found in figure 3.6.

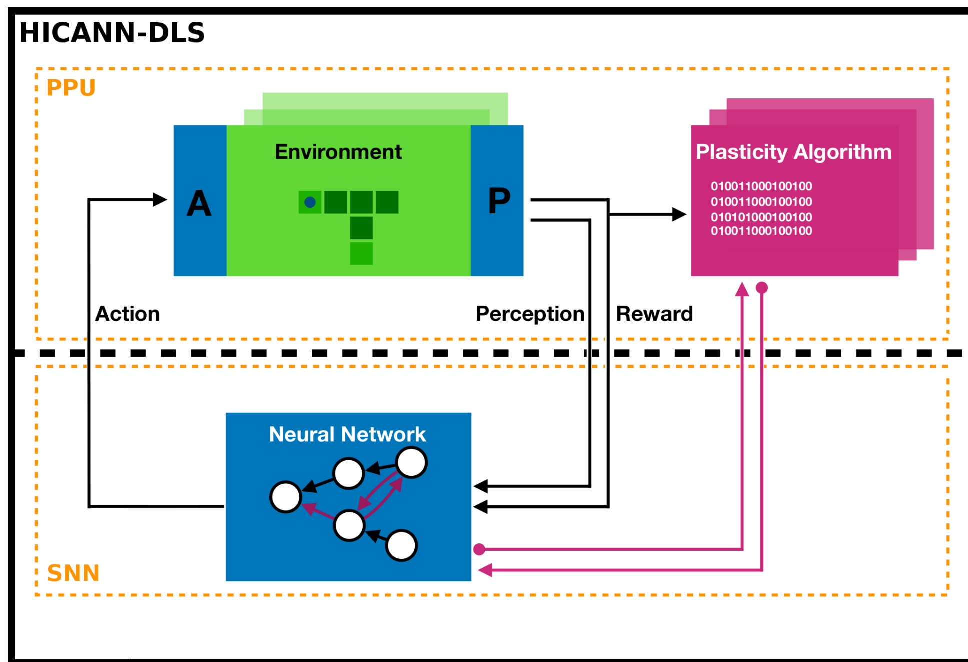


*Figure 3.6: An abstract view of the software structure and the information flow inside the HICANN-DLS. The three main components implemented on the neuromorphic are the neural network structure, the environment of the different tasks and the learning algorithms. The environment for the task as well as the learning algorithm are implemented on the PPU. The neural network is constructed by the host computer and runs in parallel to the PPU. In the beginning the environment is initialized to an initial state which is presented as an observation to the network. The neural network then produces an action according to this state. This action is then selected and performed from the environment. The third step involved is the weight update of the neural network according to a plasticity algorithm. The base version of this figure is provided by Christian Pehle, which was then modified to adapt to the current situation.*

As already mentioned, the PPU implements the MDP and maze environment, the action selection procedure and the weight update rule. For the action selection process, the following steps are performed:

1. The environment is in a state $s \in \mathbb{S}$.
2. The PPU sends a single spike to the state neuron $s_i$ (see figure 3.4).
3. The activated state neuron is then self-excitedly connected to itself, causing it to periodically spike.
4. Eventually, depending on the weights, one or multiple of the action neurons, which are also connected to the state neuron, will emit a spike as well.
5. The action neurons are inhibitory connected among each other. This inhibition prevents multiple action neurons from spiking. In addition, the action neurons are also inhibitory connected to the state neuron and cause the state neuron to stop spiking.
6. Finally the PPU selects the action for the given state as the action neuron which spiked. If multiple action neurons spiked, a random one among them is chosen. If no action neuron spiked, a random action is taken as well.

One of the hardware limitations is that the inhibition is not arbitrary fast and it might happen that multiple action neurons emit a spike, before the inhibition present between the action neurons becomes active (see 3.3.5).

In contrast to the software implementation, the hardware faces also the limitation that the exact spike times are not accessible within the PPU (see 3.3.5). To illustrate the ongoing process from the network point of view, an exemplar spike raster is shown in figure 3.7.

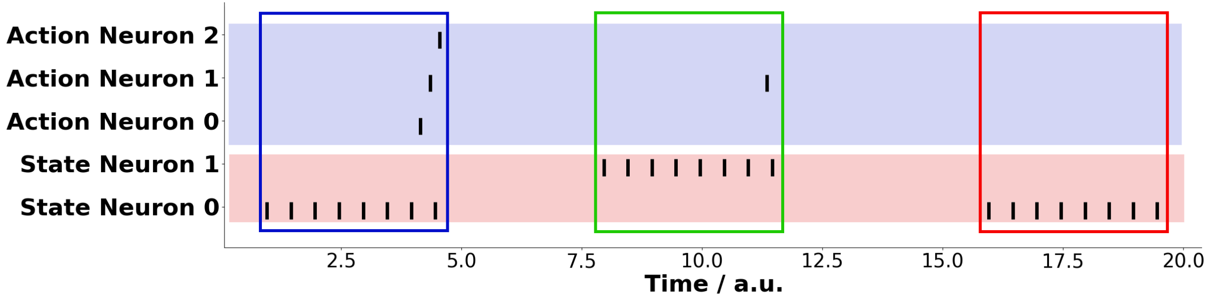


*Figure 3.7: Example sketch to demonstrate the action selection process. Three different situations with two state neurons and three actions are shown. The rectangles indicate the action selection window which is considered for selecting an action for a specific state. In the first case, the blue rectangle, the environment is in state 1 and the state neuron 1 spikes. When this state is presented to the network, the state neuron 1 receives a single spike from the PPU and emits a spike shortly afterwards. Since all state neurons are self-excitedly connected, it starts to spike periodically. In this case three action neurons also emit a spike within the action selection window. The action neurons are inhibitory connected to each other, but this inhibition has a delay to become active. This is the reason why it is possible for multiple action neurons to spike. After a defined delay, the next state is presented to the network. In the second case, marked with the green rectangle, the environment is in state 0 with state neuron 0 spiking periodically. Only a single action neuron fires within this window, which is also the selected action. Again, after a pause, the next state is presented, red rectangle. In this case no action neurons spikes and after a delay time, the periodic spiking of the state neuron is aborted to allow for the next state to be presented to the network*

To overcome the problem that no action neuron spikes for a particular state, a maximum wait time is implemented. If no action is chosen within this time window, also a random action is selected.
An additional limitation of the hardware implementation is that it only uses 8-bit fixed-point resolution for the parameters of the learning rule. This reduces the granularity of update steps applicable to the synaptic weights. The memory limitation in combination with the implemented framework, restricts the parameter values to eight bits. When using more bits for the parameters, also the granularity of the weight update will increase and result in a better overall performance.

The proposed learning algorithm in this thesis is based on the fact that a higher synaptic weight results in an earlier spike of the postsynaptic neuron (action neuron) after a spike from the presynaptic neuron (state neuron) occurred. Since the PPU does not have access to the precise spike times, a large time difference between spikes of the postsynaptic neurons using different weights is required for this method. The mentioned inhibition between the action neurons can improve this time difference, because the first spiking action neurons inhibits all others. However, if the spike time difference is too small for the inhibition to become active, another mechanism must be in place to increase the spike time difference.
One possibility is to keep the synaptic weights are within a certain range. On the one hand if the weights are at the upper limit around 63 the spike time difference for different weights is too small to perform the action selection based on them. On the other hand, if the synaptic weights are too low, the postsynaptic neuron does not emit a spike at all if the presynaptic neuron spikes. To keep the weights in a proper regime, a weight rescaling with three parameters is implemented:

- Rescale periodicity: The weights are rescaled periodically, which is determined by this parameter. At fixed times, the weights of each state neuron to all action neurons are rescaled

separately. At the beginning of this process, the current weight interval $[\min_i(w_{i,s_j}), \max_i(w_{i,s_j})]$ for each state $s_j$ is linearly transformed to the rescale interval $[w_{lower}, w_{upper}]$. For a better understanding this process can be illustrated with the following examples:
Consider for example the following old weights $\{23, 27, 35, 50\}$ resulting in the interval $[23, 50]$. This should be rescaled to the interval new $[30, 40]$. After rescaling, the new weights look as follows: $\{30, 31, 34, 40\}$.

- Lower weight limit $w_{lower}$: This is the lower limit of the new rescaled interval.
- Upper weight limit $w_{upper}$: This is the upper limit of the new rescaled interval.

### 3.3.3 Mailbox

The mailbox is the interface between the program running on the PPU and the Python code on the host computer and has a limited storage capacity of $4\,kB$. The implemented framework uses this mailbox to exchange configuration information about the MDP and maze problem as well as the learning rates for the update rule from the host computer to the DLS chip. Information about the performed actions and occupied states are exchanged in the opposite direction from the chip to the host computer.

### 3.3.4 Chip calibration

Since the neuromorphic hardware used contains analog components, there is chip to chip and also trial to trial variability present. To be able to use similar neuron configurations as in the software model and to also exchange the hardware, a unified calibration is needed. A student from the University of Heidelberg investigated the influence of various analog and digital hardware parameters and developed a database, which can be used for calibration, for his bachelor thesis. When using this database, one can collect values for desired neuron parameters and for different hardware chips. To mention one example, it is possible to find calibration values for a desired membrane time constant in biological time for two different hardware chips [Stradmann, 2016]. This is essential, since the hardware resources in Heidelberg are limited and with this calibration, the implementation is not tailored to one specific chip.

### 3.3.5 Limitations

The neuromorphic hardware used was in a prototype status when this thesis was carried out. Therefore, there are some limitations present which limit the proposed algorithm.

- The current version of the HICANN-DLS only holds 32 neurons. However, this neuron limitation will be improved in future generations.
- Only a Python 2 interface is available per default. This caused issues with the LTL framework from the Institute for Theoretical Computer Science which was developed in Python3. To use this framework in combination with the hardware, it must be translated into Python2 first. However, the software framework is compatible with Python 3, but was not recommended to use at the moment of conducting the experiments.
- The exact times of the spikes are not accessible within the PPU. This limits the proposed approach and requires using only spike occurrences to determine which action should be selected. If more than one action neuron fired within the selection window, a random action gets selected. However, this mechanism did not destroy the performance of the algorithm and the results show that the learning rule and the inhibition between the action neurons can compensate for this limitation (see hardware performance in section 5.4).

- In the current version of the hardware, all recurrent spikes from neurons to other neurons are accumulated by the external FPGA circuit. They are sent back in on a periodic basis, but all with the same address. This means that all spikes from neuron $i$ injected by synapse driver $i$ have the same address. This has to be taken into account when designing a network structure.
  Future hardware releases will in cooperate on chip routing which should resolve this issue.

- DLS is unavailable if an experiment crashes when operating the PPU. An exception with the message "Query does not read or write" occurs if the chip is not available anymore and no further experiment can be performed until the chip is properly restarted. This situation can easily occur if the running job is aborted for some reason when executing on the PPU. At the moment when this thesis was carried out, no proper solution for this problem was present.

## 3.4 Learning to Learn framework

The Learning to Learn framework used, was developed at the Institute for Theoretical Computer Science at Technical University of Graz. This framework is built-up in an abstract and modular way according to the Learning to Learn methodology introduced in section 2.4. In this thesis, the optimizee is a neural network used to learn given MDPs, and the optimizer is used to tune the hyperparameters of this network. Due to the modular structure, the two components are separated and can be exchanged easily.
Right from the beginning, the implementation of both, the software and the hardware model was an integral part of this framework to reduce the later effort when applying this concept. The framework holds a various number of optimization algorithms as optimizers, where for this thesis, mainly the following four were used.

### 3.4.1 Crossentropy

A tutorial on Crossentroy (CE) was proposed by [de Boer et al., 2005]. Although there are modifications available (Fully Adaptive Cross Entropy, FACE), the preset framework uses the standard version. The following pseudocode sketches the main concepts of the algorithm. However, for further details, the reader is referred to the tutorial in [de Boer et al., 2005].:

**Algorithm 3** Crossentropy

1: **procedure** CROSSENTROPY($n_{max}$, $n$, $\rho$, $sc$, $sm$, $\mathcal{P}$)
2: Initialize probability distribution $p(\,\cdot\,;\boldsymbol{\phi})$ from the family $\mathcal{P}$
3: **while** maximum iterations $n_{max}$ not reached and $sc$ not met **do**
4: **for** $i \in \{1, 2, ..., n\}$ **do**
5: Sample $\boldsymbol{\theta}_i$ from $p(\,\cdot\,;\boldsymbol{\phi})$
6: $F_i$ = Fitness of $(\boldsymbol{\theta}_i)$
7: **end for**
8: Sort fitnesses $F_i$ in descending order
9: Select the $\lfloor \rho \cdot n \rfloor$ best performing individuals, resulting in the elite set $E$
10: $\phi' = \arg\max_{\boldsymbol{\phi}} \sum_{\boldsymbol{\theta}\in E} \log(p(\boldsymbol{\theta};\boldsymbol{\phi}))$ (Minimize crossentropy)
11: Optionally perform smoothing $\phi \leftarrow s\phi + (1-s)\phi'$
12: **end while**
13: **return** $\boldsymbol{\theta} sampledfromp(\,\cdot\,;\boldsymbol{\phi})$
14: **end procedure**

This algorithm tries to fit a parametric distribution $p(\,\cdot\,;\boldsymbol{\phi})$ on the current best individuals. It starts out with randomly generated individuals and an initial parametric probability dis-

tribution. In subsequent steps, the algorithm samples new individuals from this distribution, evaluates them and keeps the best performing ones. The new probability distribution is fitted onto these elite individuals by minimizing the cross-entropy. In the next iteration, individuals are then sampled from this new distribution.
With this approach, the probability distribution evolves towards a region of individuals with high fitness values, because with each iteration it becomes more likely to sample better individuals than in the previous iteration.
The algorithm can be improved with a smoothing parameter, enabling a smoother parameter update of the distribution. In addition, the probability distribution used can be customized according to the present problem and fitness landscape. In this thesis, a Noisy Gaussian distribution family is used, which can be seen as a Gaussian distribution attached with noise components.
The parameters of this optimization algorithms, also called hyperhyperparameters (HHP), are:

- Maximum iteration $n_{max}$: This parameter denotes the maximum number of iterations performed. If no other stopping criterion is met, the algorithm will stop if the number of iterations exceeds this limit.
- Population size $n$: This denotes the amount of individuals sampled in each iteration.
- Elite factor $\rho$: This factor, in combination with the population size, determines how many individuals are considered during the fitting process of the probability distribution.
- Stopping criterion $sc$: This is an optional criterion, denoting which fitness value stops the execution of the algorithm.
- Smoothing parameter $sm$: This denotes the optional smoothing value. In this case, the update of the parameters of the probability distribution, is a linear combination of the old parameters and the newly fitted ones weighted with this smoothing parameter.
- Distribution family $\mathcal{P}$: This parameter denotes the used probability distribution family used for fitting the elite individuals. As already mentioned a Noisy Gaussian distribution is used in this thesis.

### 3.4.2 Evolution Strategies

Evolution Strategies (ES) presented by [Wierstra et al., 2014] is a modification of the well-know evolutionary strategies and was already successfully used in a recent work by [Salimans et al., 2017]. Again, the basis outline of the algorithm is sketched in the pseudocode 4 and detailed information can be obtained from the stated papers.
The algorithm starts out with a randomly generated individual. In each following iteration, this individual is modified with random perturbations until a population size of $n$ is reached. In case the extension of "mirrored sampling" is used, the perturbations are mirrored, meaning that half of the individuals are modified in one direction and the other half with the same absolute value in the opposite direction.
Once this perturbation is done, the new individuals are evaluated and linearly combined to yield the new base parameter set $\boldsymbol{\theta}$ which is again perturbated in the next iteration.
The hyperhyperparameters of this optimization algorithms are:

- Maximum iteration $n_{max}$: This parameter denotes the maximum number of iterations performed. If no other stopping criterion is met, the algorithm will stop if the number of iterations exceeds this limit.
- Learning rate $\alpha$: This denotes the learning rate of the algorithm when updating the base parameter set $\boldsymbol{\theta}$, which should be perturbated in the next iteration.

**Algorithm 4** Evolution Strategies

1: **procedure** EVOLUTIONSTRATEGIES($n_{max}$, $\alpha$, $\sigma$, $ms$, $n$, $sc$, $fs$)
2: Randomly generate one individual $\boldsymbol{\theta}$
3: **while** maximum iterations $n_{max}$ not reached and $sc$ not met **do**
4: **for** $i \in \{1, 2, ..., n\}$ **do**
5: **if** $ms$ **and** $i \geq \lceil \frac{n}{2} \rceil$ **then**
6: $\boldsymbol{\epsilon}_i = -\boldsymbol{\epsilon}_{n-i}$
7: **else**
8: Sample $\boldsymbol{\epsilon}_i$ from $\mathcal{N}(\mathbf{0}, \sigma)$
9: **end if**
10: $F_i =$ Fitness of $(\boldsymbol{\theta} + \boldsymbol{\epsilon}_i)$
11: **end for**
12: **if** $fs$ **then**
13: $k(i) \leftarrow$ Index of $i$-th greatest fitness
14: $u_i = \frac{\max\left(0, \log\left(\frac{n}{2}+1\right) - \log(k(i))\right)}{\sum_{j=1}^{n} \max\left(0, \log\left(\frac{n}{2}+1\right) - \log(j)\right)} - \frac{1}{n}, \quad \forall i \in \{1, 2, ..., n\}$
15: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \frac{\alpha}{\sigma n} \sum_{i=1}^{n} u_i \boldsymbol{\epsilon}_i$
16: **else**
17: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \frac{\alpha}{\sigma n} \sum_{i=1}^{n} F_i \boldsymbol{\epsilon}_i$
18: **end if**
19: **end while**
20: **return** $\boldsymbol{\theta}$
21: **end procedure**

- Population size $n$: This parameter denotes the size of the population which emerges after the perturbation process

- Noise standard deviation $\sigma$: This denotes the standard deviation of the random perturbation applied to the individual.

- Stopping criterion $sc$: This is an optional criterion, denoting which fitness value stops the execution of the algorithm.

- Mirror sampling $ms$: This flag indicates if the perturbations should be mirrored or if the individuals should be perturbated in only one direction.

- Fitness shaping $fs$: This flag indicates if the fitness shaping method should be considered or not. Using this flag the update of the individual to perturbate is not done with the fitness directly, but with a so-called utility function $u_i$ which might yield a performance improvement as shown in [Salimans et al., 2017].

### 3.4.3 Simulated Annealing

Simulated Annealing (SA) as presented in [Kirkpatrick et al.] provides a reasonable baseline for more advanced optimization algorithms. The concept underlying this technique is the annealing process of metals, where the temperature is precisely controlled to reach a state with low energy (e.g.: low number of defects inside the metal).
The framework implements a special version of this method, in which several annealing processes are performed in parallel to increase the chance to find a good fitness maximum. The main idea of the algorithm is shown in algorithm 5:
Like many algorithms, this one also starts out with randomly generating $n$ individuals. As the algorithm mimics an annealing process, a temperature exists, which decreases from an initially high value to a lower value throughout iterations. In each iteration of the algorithm,

**Algorithm 5** SimulatedAnnealing

1: **procedure** SIMULATEDANNEALING($n_{max}$, $n$, $\sigma$, $sc$, $\mathcal{T}$)
2: Randomly generate individuals $\boldsymbol{\theta}_i$, $\forall i \in \{1, 2, ..., n\}$
3: Initialize temperature $T$
4: $cnt \leftarrow 0$
5: $F_i \leftarrow$ Fitness of $(\theta_i)$, $\forall i \in \{1, 2, ..., n\}$
6: **while** maximum iterations $n_{max}$ not reached and $sc$ not met **do**
7: **for** $i \in \{1, 2, ..., n\}$ **do**
8: Sample $\boldsymbol{\epsilon}_i$ from $\mathcal{N}(\mathbf{0}, \sigma)$
9: $F_i' =$ Fitness of $(\boldsymbol{\theta}_i + \boldsymbol{\epsilon}_i)$
10: **if** $F_i' \geq F_i$ **then**
11: $\boldsymbol{\theta}_i \leftarrow \boldsymbol{\theta}_i + \boldsymbol{\epsilon}_i$
12: $F_i \leftarrow F_i'$
13: **else**
14: $p = \frac{exp(-(F_i' - F_i))}{T}$
15: **if** $U(0, 1) < p$ **then**
16: $\boldsymbol{\theta}_i \leftarrow \boldsymbol{\theta}_i + \boldsymbol{\epsilon}_i$
17: $F_i \leftarrow F_i'$
18: **end if**
19: **end if**
20: **end for**
21: $cnt \leftarrow cnt + 1$
22: $T \leftarrow \mathcal{T}(T, cnt)$
23: **end while**
24: $i = \arg\max_{i'} F_{i'}$
25: **return** $\boldsymbol{\theta}_i$
26: **end procedure**

the $n$ individuals are treated separately in the annealing process. For simplicity, $n = 1$ in this description.
The individual is perturbated with a random step and evaluated and this new fitness value is compared with the previous fitness value. If the new fitness value is higher than the previous one, the perturbated individual is kept and used in the next iteration. On the other hand, if the new fitness value is lower than the previous one, the perturbated individual is only kept with a certain probability, which is decreasing when the temperature is decreasing. This approach allows for exploration at high temperatures and reduces to a greedy approach at low temperatures. Before the next iteration starts over again, the temperature is decreased according to the given schedule. The HHPs of this algorithm are:

- Maximum iteration $n_{max}$: This parameter denotes the maximum number of iterations performed. If no other stopping criterion is met, the algorithm will stop if the number of iterations exceeds this limit.
- Population size $n$: This denotes the amount of parallel simulated annealing processes. Those individuals are treated separately during the optimization process and can therefore not be directly compared to a population size as in case of Crossentropy 3.4.1.
- Stopping criterion $sc$: This is an optional criterion, denoting which fitness value stops the execution of the algorithm.
- Noise standard deviation $\sigma$: This denotes the standard deviation of the random perturbation applied to the individual.

- Temperature schedule $\mathcal{T}$: The way how the temperature is decreased can be specified with this temperature schedule. The present framework supports various schedules which provide different advantages in certain situations. In this thesis the "Quadratic adaptive" cooling schedule is used. The behavior of the other schedules was not investigated to a greater detail, but were already investigated in the literature, e.g.: [Fernando Díaz Martín and Riaño Sierra, 1].

### 3.4.4 Classic gradient descent

The classic gradient descent optimization algorithm (GD) is also a baseline algorithm used as a reference. In the implemented version, finite differences are used to estimate the gradient around the current base point. Also, the pseudocode of the last algorithm is depicted here:

**Algorithm 6** GradientDescent

1: **procedure** GRADIENTDESCENT($n_{max}$, $\alpha$, $n$, $sc$, $\sigma$)
2: Randomly generate initial base point individual $\theta$
3: **while** maximum iterations $n_{max}$ not reached and $sc$ not met **do**
4: **for** $i \in \{1, 2, ..., n\}$ **do**
5: Sample $\boldsymbol{\epsilon}_i$ from $\mathcal{N}(\mathbf{0}, \sigma)$
6: $F_i = \mathcal{F}(\boldsymbol{\theta} + \boldsymbol{\epsilon}_i)$
7: **end for**
8: $F' =$ Fitness of $(\boldsymbol{\theta})$
9: $\boldsymbol{\Delta}F = (F_1 - F', F_2 - F', ..., F_n - F')^T$
10: $\boldsymbol{E} = [\boldsymbol{\epsilon}_1|\boldsymbol{\epsilon}_2|...|\boldsymbol{\epsilon}_n]^T \in R^{n \times d}$
11: $\boldsymbol{g} = \arg\min_{\boldsymbol{g}'} (\boldsymbol{E}\boldsymbol{g}' - \boldsymbol{\Delta}F)^2$
12: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\boldsymbol{g}$
13: **end while**
14: **return** $\boldsymbol{\theta}$
15: **end procedure**

The algorithm starts out with a random individual as the base point. In each following iteration, the neighborhood of this base point is explored with small perturbations and evaluated. The gradient is then computed based on the finite fitness differences between the base point and the neighboring points. Finally, the new base point is computed by taking a step into the direction of the gradient and the next iteration starts with its exploration. One problem with this approach is that the base point might not yield a good reference value in noisy fitness landscapes. To reduce this noise, the base point is evaluated ten times, before the neighborhood is explored.

The HHPs of this algorithm are:

- Maximum iteration $n_{max}$: This parameter denotes the maximum number of iterations performed. If no other stopping criterion is met, the algorithm will stop if the number of iterations exceeds this limit.

- Number of random steps $n$: This parameter indicates how many points in the vicinity of the base point should be evaluated and considered for the gradient computation.

- Stopping criterion $sc$: This is an optional criterion, denoting which fitness value stops the execution of the algorithm.

- Noise standard deviation $\sigma$: This denotes the standard deviation of the random perturbation applied to the individual.

- Learning rate $\alpha$: This parameters denotes the learning rate and control how large the step into the direction of the gradient will be during the update of the base point.

For the rest of this thesis, the term "LTL algorithm" refers to the optimization algorithm used in the optimizer to tune the hyperparameters of the optimizee.

The hyperparameters of the optimizee consist of learning rule and implementation parameters specific for hardware and software. The two learning rules used are parameterized by the red marked and circled parameters stated in equations 3.1, 3.2, 3.3 and 3.4 regardless of the underlying platform.

For the software implementation the additional parameters are:

- Weight bias: This represents the initial weight between the state and action population.
- Scaling factor: This multiplicative factor is used to scale the Q-Values stored in the synaptic weights.
- Learning rate decay: The learning rate is multiplied with this factor in each iteration in order to reduce the learning rate over time.

For the hardware implementation following parameters are used in addition:

- Rescale period: Since the Q-Values modify the weights of the synapses and the spiking behavior of the network depends on them, the current weights are rescaled with this periodicity. This allows for a tradeoff between too high weights, where almost all action neurons spike within the action selection window and too low weights, were no action neuron spikes.
- Lower weight boundary: This represents the lower weight boundary used for rescaling.
- Upper weight boundary: This represents the upper weight boundary used for rescaling.
- Weight bias: Similar to the software implementation, the initial weight for the network is parameterized.
- Inhibition strength: In contrast to the software implementation, the hardware does not explicitly implement an $\epsilon$-greedy policy, where a random action is selected with a probability of $\epsilon$. Instead, this explorative behavior is modified by the strength of the inhibition. A stronger inhibition allows only one action neuron to spike within the action selection window, while a weaker inhibition may allow multiple action neurons to spike, effectively increasing the chance of taking an alternative action.

One important aspect when applying different optimization algorithms in the optimizer is that all hyperparameters are scaled to the same interval, here for simplicity $[0, 1]$. This is necessary since optimization algorithms might use the same update step for different hyperparameters and if they are not scaled properly, this might result in too high or too low updates.

As explained in the background section on LTL 2.4, the optimizee learns a new task from the family $\mathcal{F}$ based on the given hyperparameters, where the fitness landscape might be noisy. To reduce this noise, the optimizee learns 20 new tasks with the same hyperparameters and the final fitness is averaged over those tasks.

### 3.4.5 Hyperhyperparameters for Learning to Learn framework

This section is intended to state a table (3.1) with the HHPs used for the different LTL algorithms and for the software and hardware model:

*Table 3.1: Hyperhyperparameters used for various Learning to Learn algorithms*

| LTL algorithm | HHP | MDP | | Maze | |
|---|---|---|---|---|---|
| | | SW | HW | SW | HW |
| CE | $n_{max}$ | 75 | 30 | 50 | 30 |
| | $n$ | 75 | 50 | 48 | 50 |
| | $\rho$ | 0.2 | 0.2 | 0.2 | 0.2 |
| | $sm$ | 0.0 | 0.0 | 0.0 | 0.0 |
| SA | $n_{max}$ | 75 | 30 | 50 | 30 |
| | $n$ | 75 | 50 | 48 | 50 |
| | $\sigma$ | 0.03 | 0.03 | 0.03 | 0.03 |
| | $\mathcal{T}$ | Quadratic Adaptive | | | |
| ES | $n_{max}$ | 75 | 30 | 30 | 30 |
| | $n$ | 38 | 25 | 24 | 25 |
| | $\alpha$ | 0.1 | 0.5 | 0.1 | 0.3 |
| | $\sigma$ | 0.1 | 0.1 | 0.1 | 0.05 |
| | $ms$ | True | | | |
| | $fs$ | True | | | |
| GD | $n_{max}$ | 75 | 30 | 50 | 30 |
| | $n$ | 75 | 50 | 48 | 50 |
| | $\alpha$ | 0.0001 | 0.001 | 0.0001 | 0.001 |
| | $\sigma$ | 0.0001 | 0.001 | 0.0001 | 0.001 |

# 4 Experiments

## 4.1 Tasks

To evaluate the designed network and implementation, two types of MDPs are considered. Since the network structure is the same for the hardware and the software model, the tasks are also similar. To be able to compare the results of the software and the hardware model, the tasks are mainly limited by the capabilities of the neuromorphic hardware. Both types of tasks are executed using TD-Learning as well as TD($\lambda$)-Learning as the weight update rule.

### 4.1.1 MDP

The main tasks are MDPs, with randomly generated probability and transition matrices. In this context the difficulty of the tasks depends on the size ($\mathbb{S}$ x $\mathbb{A}$) of the problem.

For Learning to Learn, the family of tasks needs to be defined. In this case the family is defined by random MDPs with the same size. The transition and reward probabilities are different for each individual of the family. Different sizes of MDPs are considered for evaluation, which can be grouped into the following classes:

*Table 4.1: MDP classes considered for evaluation. The class "Fixed" has a fixed transition and reward matrix and is only used as a proof-of-principle task for the hardware. The classes "Small" and "Large" have random reward and transition matrices and present different difficulty levels, as the state and action space size increase.*

| Class | $\|\|\mathbb{S}\|\|$ | $\|\|\mathbb{A}\|\|$ |
|---|---|---|
| Fixed | 2 | 4 |
| Small | 2 | 4 |
| Large | 6 | 8 |

The class "Fixed" contains only a single individual with a fixed transition and reward matrix and a size of 2 x 4. This class is used as a proof-of-principle tasks to demonstrate that the agent is in principle capable of learning. It is designed such that action 2 in state 0 and action 3 in state 1 yields a reward. The other actions in the different states do not yield a reward at all. Figure 4.1 shows an array representation of the reward and transition matrix of this problem.

```
P=[[[1., 0.], [0., 1.]], R=[[[-1., -1.], [-1., -1.]],
   [[1., 0.], [0., 1.]],    [[-1., -1.], [-1., -1.]],
   [[0., 1.], [0., 1.]],    [[-1.,  1.], [-1., -1.]],
   [[1., 0.], [1., 0.]]]    [[-1., -1.], [1. , -1.]]]
```

*Figure 4.1: Transition and reward matrix for "Fixed" MDP. The left-hand side shows the transition matrix $P$ in an array notation and the right-hand side shows the corresponding reward matrix $R$. For this special problem only action $a = 2$ in state $s = 0$ and action $a = 3$ in state $s = 1$ yield a positive reward.*

A more difficult task is the "Small" class, where the same size of MDPs are used, but a random reward and transition probabilities are drawn at every execution. LTL can be applied here to learn parameters for the entire family rather than for a single task.
The last class is the largest one, which is usable with the PPU and the implemented framework. Although the number of neurons would allow for an even larger problem class, the memory of the PPU is limited and already fully allocated at the size of 6 x 8. Again, a random MDP is drawn at every execution, allowing for the use of the LTL framework.

### 4.1.2 2D Maze

The second family of tasks used to evaluate the agent, are slight derivations from normal MDPs, namely 2D mazes. To generate random mazes, a library is used. In figure 4.2 an example of a maze setting and the resulting maze is depicted.

*Figure 4.2: This figure shows an example maze setting of size 7 x 7 on the left-hand side and the resulting maze on the right-hand side. The entry "1" in the maze array represents a wall and a "0" a free element. The goal state G, the agent should ultimately reach, is configured separately and not shown on the left-hand configuration. Note: The represented maze is flipped around the horizontal axis to be compatible with the used reference library.*

The matrix generated by the library contains wall elements, denoted with a "1" and free elements, denoted with a "0". Each free element is a potential starting point for the agent. At the beginning of a new maze task, the agent is initialized to a random position among the potential starting points. Once the goal state is reached, the agent is reset to another random starting point.
In addition the library creates only connected mazes, where it possible to reach the goal state from every potential starting point. Mazes which are non-connected, meaning that the goal state is not reachable from every starting point, are not considered in this thesis.
In this case the size of the action space is fixed to $||\mathbb{A}|| = 4$ (for moving north, south, west and east) and the transition probability matrix is designed in a way that a 2D maze environment is emulated. The size of the state space $||\mathbb{S}||$ is determined by the size of the maze without the outermost walls. It is considered for the difficulty level of the problem and is also limited due to the hardware and computation time (see chapter 5 and A). Similar to the MDP tasks, the family for applying LTL is defined as the set random mazes with a fixed size where the goal state can also vary. Two different sizes are used to evaluate the agent:
Similar to the MDP case, the class "Fixed" contains only a single individual with a fixed transition and reward matrix. This class is the proof-of-principle tasks to demonstrate that the agent is capable of learning.

*Table 4.2: Maze classes considered for evaluation. The class "Fixed" has a fixed goal state and is only used as a proof-of-principle task for the hardware. For evaluating the hardware implementation, a maze with the size of 3 x 3 and for the software case a maze of size 5 x 5 is used.*

| Class | Maze height | Maze width |
|---|---|---|
| Fixed | 3 | 3 |
| Hardware | 3 | 3 |
| Software | 5 | 5 |

Figure 4.3 shows the maze setting and the resulting maze for the "Fixed" problem class.

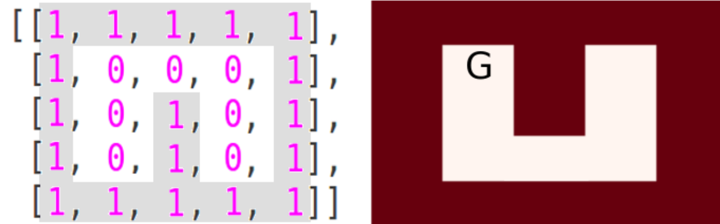


*Figure 4.3: Maze configuration for the class "Fixed". The maze setting with 3 x 3 is shown on the left-hand side and the resulting maze on the right-hand side. The entry "1" in the maze array represents a wall and a "0" a free element. The goal state, the agent should ultimately reach, is configured separately and not shown on the left-hand configuration. In the maze representation, the goal state is denoted by a capital G. Note: The represented maze is flipped around the horizontal axis to be compatible with the used reference library.*

After the learning capability was shown on MDPs, the last class considered for evaluation is a random maze of such a size that fit onto the neuromorphic hardware, where 13 neurons (9 state neurons and 4 action neurons) are used in total. Similar to the MDP case, the mazes are randomly generated with a fixed size, but varying goal positions.

# 5 Results

This section gives an overview about the collected results. Starting with background information about the LTL plot generation in section 5.1 and the hardware investigation in section 5.2, the software results 5.3 are then followed by the hardware results in 5.4.

## 5.1 Learning to Learn result

This section is intended to explain the resulting plots of the Learning to Learn experiments.
The common part of all optimization algorithms used is that they produce hyperparameter settings, so called individuals, which should increase the overall performance of the agent. The individuals are clustered, depending on the actual algorithm used.
To explain this more clearly, consider the example of Crossentropy. There, the optimizer creates a certain number of individuals per generation, according to a population size, evaluates those individuals and keeps the best of them. Then the next generation of individuals is generated by the optimizer. The evaluation of this algorithm is then done based on the average fitness value of individuals within a generation.
For evaluating algorithms which do not use a population size during operation, a similar average measure is considered:

- Crossentroy: The algorithm in cooperates a population size, which is used to sample individuals from the current probability distribution. For evaluation, the average fitness over those individuals is computed.
- Simulated Annealing: The algorithm produces a certain number of new individuals per iteration, according to the amount of parallel runs (see 3.4). For evaluation, the mean value is computed as the average fitness over the $n$ individuals for each iteration of SA.
- Evolution Strategies: This algorithm produces a preset amount of individuals in each iteration by modifying the components of the old individuals. For evaluation, the average fitness over those generated individuals in each iteration is used.
- Gradient descent: The algorithm samples new individuals around a base point to estimate the gradient and then moves into the direction of the steepest ascent of the fitness. In this case, the average fitness value over the sampled individuals including the basepoint fitness is considered.

Figure 5.1 shows the individuals of five generations and the corresponding average values.

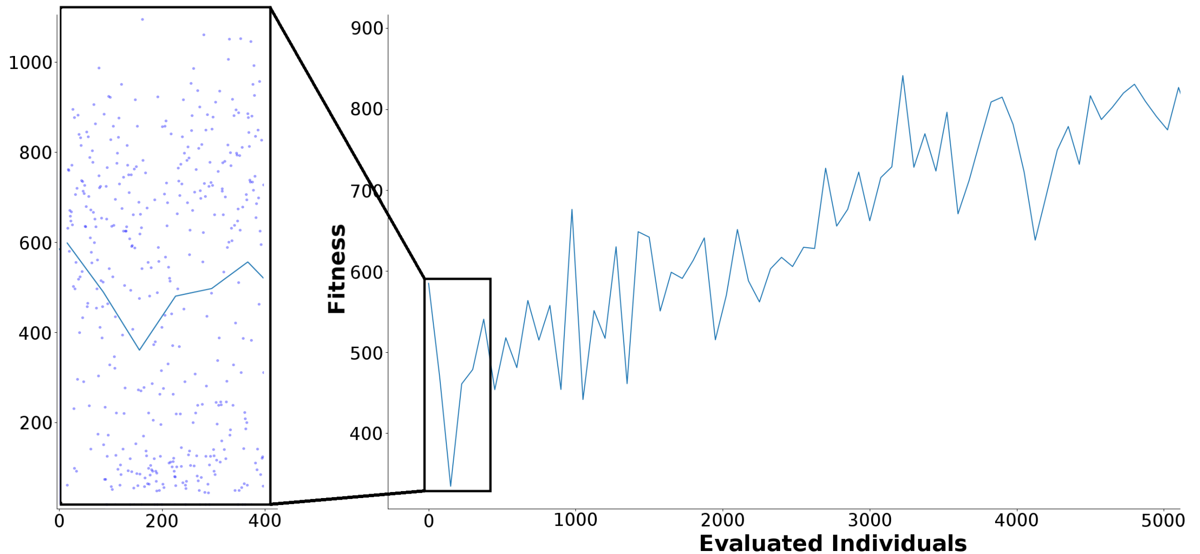


*Figure 5.1: Example result of a software simulation using Crossentropy (population size $n = 75$ and max iterations $n_{max} = 66$) on a random MDP from the class "Small". The right-hand side shows the averaged fitness evolution over the evaluated individuals, where the single individuals are not shown. The averaging is done over the entire population of each iteration of CE, which is the reason why a new value is present at every $n$ evaluated individuals. The left-hand side shows a zoom onto the first 5 iterations, where the evaluated individuals are shown as dots in the background. The average values of the iterations are connected by the thicker line in the foreground. As one can see, the average value can be interpreted as a kind of low-pass filtered version of the single individuals and therefore have a reduced amount of data points. In the following sections, only the average values are compared and the single individuals are not shown anymore.*

The HHPs used for the different LTL algorithms and for both, for the hardware as well as for the software model, can be found in table 3.1.

## 5.2 Hardware variability

To investigate the variability of the underlying neuromorphic hardware, a single random MDP problem with $||\mathbb{S}|| = 6$ and $||\mathbb{A}|| = 8$ was executed 20 times consecutively. The network setup was fixed among those runs to only investigate the same neurons and to avoid an additional influence from other neurons. Since the framework was developed in such a way that the underlying hardware can be exchanged (see 3.3.4), the exactly same experiment was done on two different chips. Figure 5.2 shows a comparison of the outcome of the two experiments.

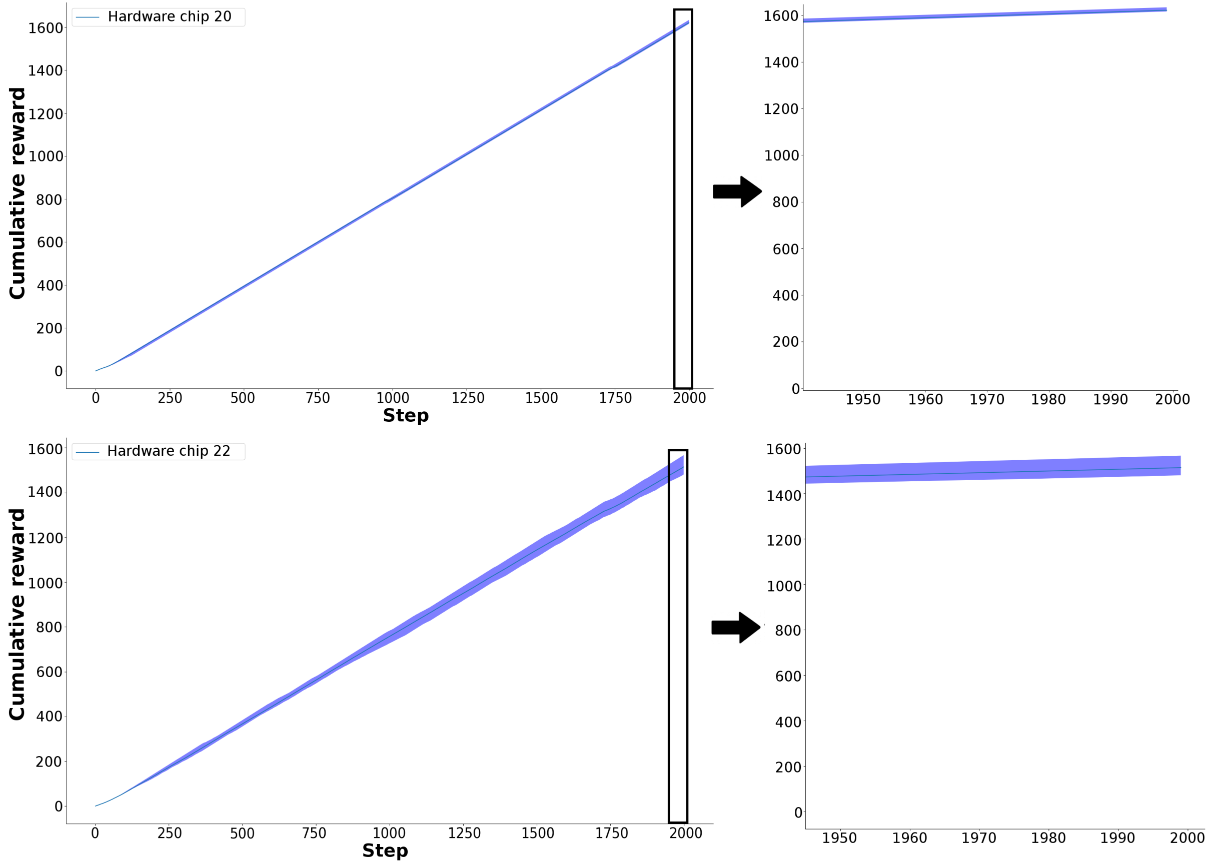


*Figure 5.2: Investigation of hardware variability for random MDP with $||\mathbb{S}|| = 6$ and $||\mathbb{A}|| = 8$ using two different hardware chips. The upper left image shows the cumulative reward for a random MDP with 20 executions using chip 20. The line represents the mean value and the shaded area represents the variability of the experiments. The upper right image shows only the last steps of the experiment since the variability has its maximum value at the end. The lower two images show the same information for the chip 22. Although the calibration explained in 3.3.4 theoretically work for both hardware chips, the mean performance as well as the variance differ.*

As one can see, the variability is larger on chip 22 and also the average value is lower (chip 20 $(1622 \pm 18)$ and chip 22 $(1514 \pm 86)$). This leads to the fact that results might not be fully comparable across different chips. To avoid this issue during the evaluation of the experiment, all LTL algorithms for each separate problem class are performed on the same chip. However, due to limited resources and the time-consuming experiments, different classes may be executed on different chips.

## 5.3 Software simulation

The results of the software simulation were carried out with NEST on small clusters using up to 84 processor cores. Since the learning rule cannot be implemented in an online fashion, the execution time ranged from ten hours for small MDPs up to two to three days for the maze tasks.

### 5.3.1 Fixed MDP and maze tasks

Since the software model has access to the precise spike times of the action neurons, even random hyperparameters for the class of "Fixed" MDP and maze tasks are sufficient to solve it. Although there might be improvement potential for this classes, they are of no deeper interest

for this model. This class was mainly designed for the hardware implementation to show a proof-of-concept.

### 5.3.2 Random MDPs

In contrast to the class of artificial MDPs, randomly chosen hyperparameters are not sufficient to solve these problems. Therefore, the LTL framework is applied to navigate through the hyperparameter space and to find right hyperparameters which optimize the performance on the task family. The network performs 2000 steps to learn the given MDP, where the fitness of the optimizee is defined as the accumulated reward. Figure 5.3 shows the evolution of the cumulative reward over evaluated individuals for the different LTL algorithms for TD-Learning, while figure 5.4 shows the LTL results for TD($\lambda$)-Learning.

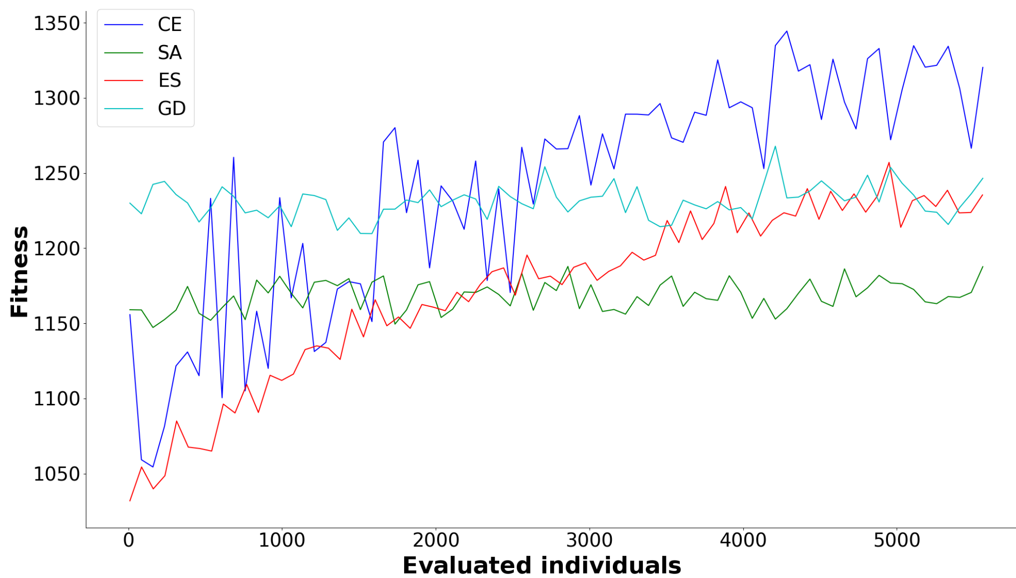


*Figure 5.3: Fitness evaluation over evaluated individuals for TD-Learning with different LTL algorithms and for $||\mathbb{S}|| = 2$ and $||\mathbb{A}|| = 4$. CE and ES show a good performance improvement, while GD and SA stay constant in terms the fitness values.*

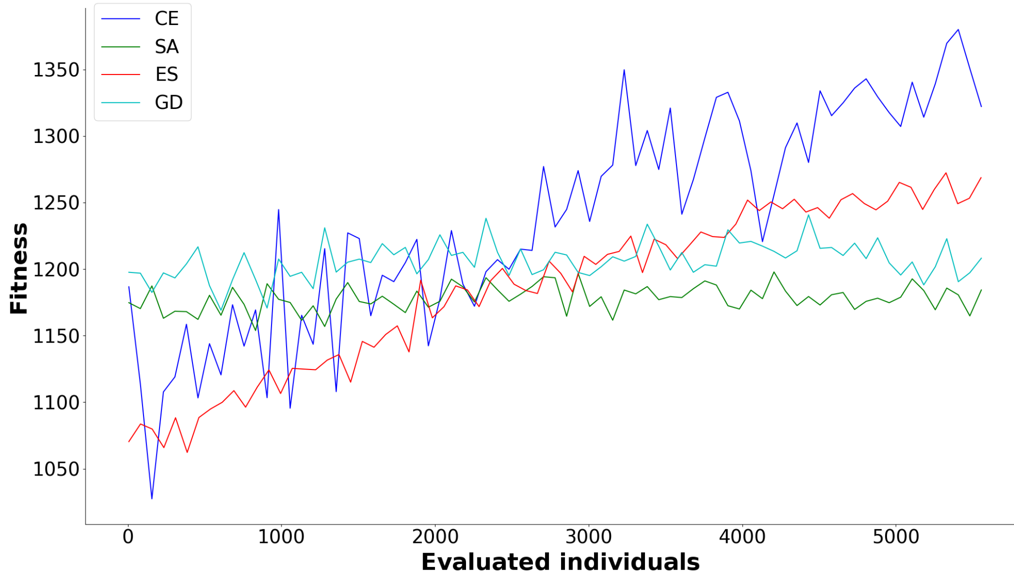


*Figure 5.4: Fitness evaluation over evaluated individuals for TD($\lambda$)-Learning with different LTL algorithms and for $||\mathbb{S}|| = 2$ and $||\mathbb{A}|| = 4$. This figure shows a similar behavior to the case of TD-Learning and the overall performance is almost the same.*

Both figures show a similar behavior, whereas traditional algorithms like GD and SA are not able to make a big performance improvement, newer algorithms like CE and ES are able to improve the performance. The performance after applying the LTL algorithms are similar for TD-Learning and TD($\lambda$)-Learning.

Similar results can also be observed for the larger MDP class with $||\mathbb{S}|| = 6$ and $||\mathbb{A}|| = 8$. Again, figure 5.5 and 5.6 show the performance improvement when applying LTL to TD-Learning and TD($\lambda$)-Learning.

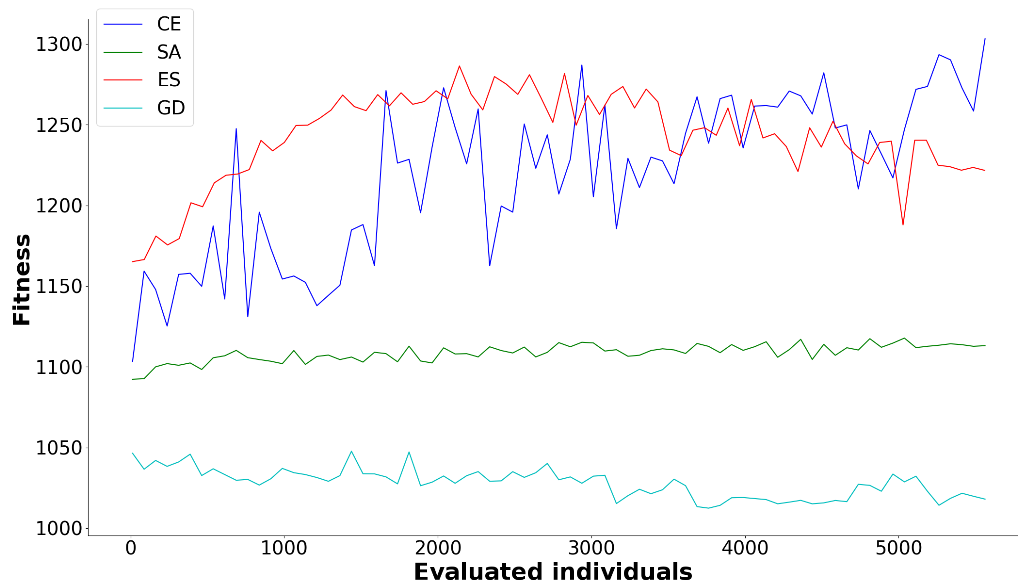


*Figure 5.5: Fitness evaluation over evaluated individuals for TD-Learning with different LTL algorithms and for $||\mathbb{S}|| = 6$ and $||\mathbb{A}|| = 8$. As the difficulty of the task increase, CE and ES clearly outperform SA and GD in terms of the average fitness value.*

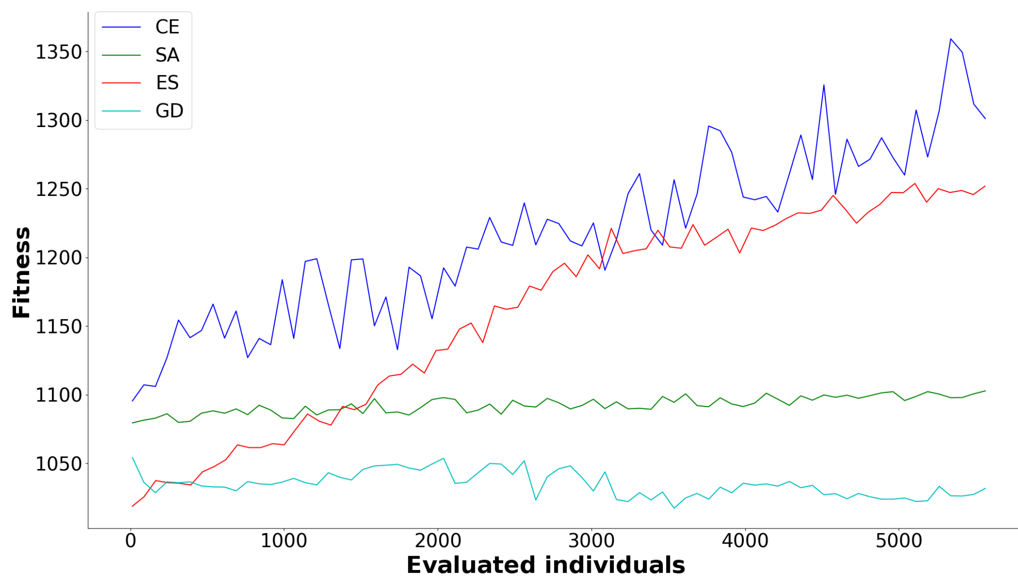


*Figure 5.6: Fitness evaluation over evaluated individuals for TD($\lambda$)-Learning with different LTL algorithms and for $||\mathbb{S}|| = 6$ and $||\mathbb{A}|| = 8$. Similar to the case of TD-Learning, CE and ES outperform the other two LTL algorithms. Again, the overall fitness reached is comparable to TD-Learning.*

The difference between the two learning algorithms used, is that TD($\lambda$)-Learning is able to learn the problem faster, i.e. it can learn a good policy for new tasks with fewer steps compared to TD-Learning. The following table 5.1 lists the number of steps needed to converge to a fixed policy for 50 random MDPs of size $||\mathbb{S}|| = 2$ and $||\mathbb{A}|| = 4$.

*Table 5.1: Comparison of steps needed to reach fixed policy.*

CE ... Crossentropy, SA ... Simulated Annealing, ES ... Evolution Strategies, GD ... Gradient Descent

| LTL-Algorithm | TD-Learning | TD($\lambda$)-Learning |
|---|---|---|
| CE | $287 \pm 511$ | $159 \pm 274$ |
| SA | $269 \pm 514$ | $346 \pm 588$ |
| ES | $402 \pm 627$ | $209 \pm 282$ |
| GD | $362 \pm 631$ | $301 \pm 771$ |

The average values show that TD($\lambda$)-Learning is able to learn the problem with fewer steps compared to TD-Learning. The high variance values come from the issue that for some problems no fixed policy was learned within the maximum number of steps.

To show that the LTL approach actually is effective, the agent is trained on 50 random MDPs for each class, for each learning algorithm and for each LTL algorithm. Those performance values are then used to fill the table 5.2.
A single problem, out of those 50 randomly generated tasks, is shown in figure 5.7 for $||\mathbb{S}|| = 2$ and $||\mathbb{A}|| = 4$ and in figure 5.8 for $||\mathbb{S}|| = 6$ and $||\mathbb{A}|| = 8$. In both cases CE was used as the LTL algorithm to tune the hyperparameters.

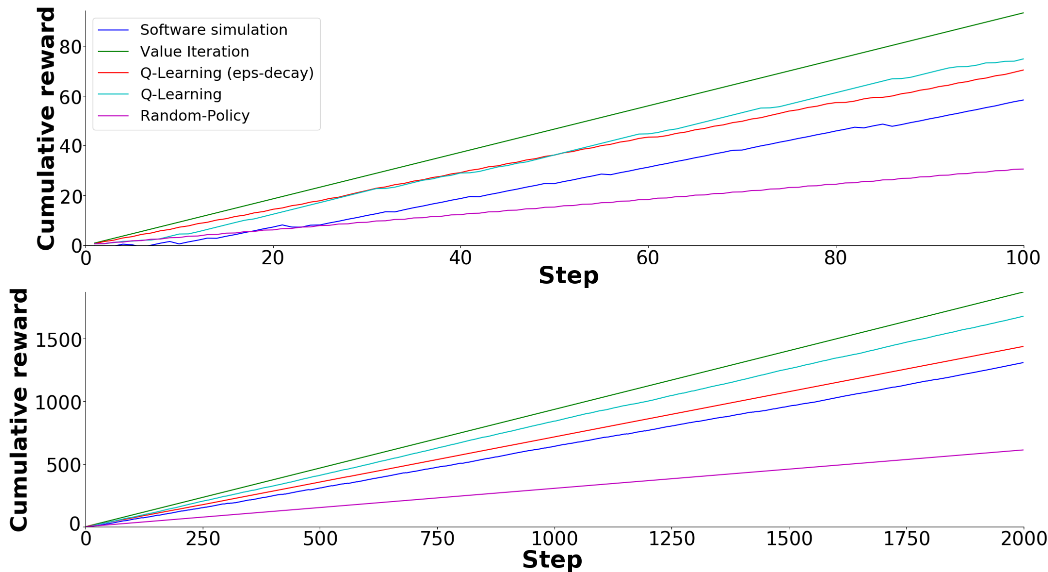


*Figure 5.7: Performance comparison of network with tuned hyperparameters (CE) using TD($\lambda$)-Learning for an MDP of size $||\mathbb{S}|| = 2$ and $||\mathbb{A}|| = 4$. The upper part shows a zoom on the first 100 steps while the lower part shows the full 2000 steps. The network can compete with the reference algorithms and clearly performs better than random.*

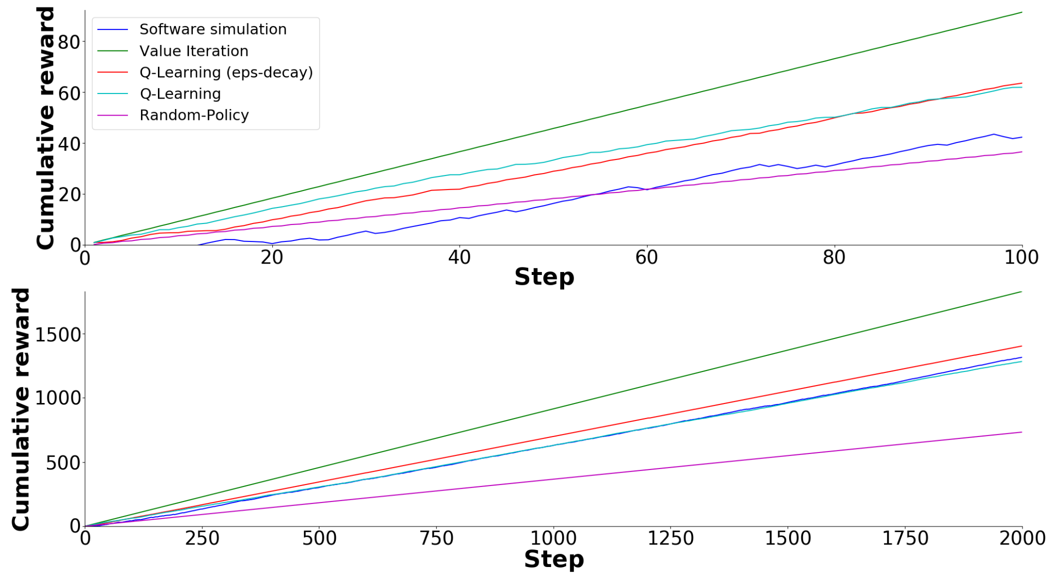


*Figure 5.8: Performance comparison of network with tuned hyperparameters (CE) using TD($\lambda$)-Learning for an MDP of size $||\mathbb{S}|| = 6$ and $||\mathbb{A}|| = 8$. The upper part shows a zoom on the first 100 steps while the lower part shows the full 2000 steps. The network can compete with the reference algorithms and clearly performs better than random.*

Table 5.2 shows the complete comparison of the performances:

*Table 5.2: Comparison of performances for different problem classes, reference algorithms, learning algorithms and LTL algorithms*

VI ... Value Iteration, QL ... Q-Learning with adaptive epsilon, QLF ... Q-Learning with fixed epsilon, Rnd ... Random policy

| Class | LTL-Algorithm | SNN | VI | QL | QLF | Rnd |
|---|---|---|---|---|---|---|
| Small (TD1) | CE | **1489 ± 196** | 1695 ± 205 | 1350 ± 342 | 1452 ± 206 | 947 ± 552 |
| | SA | **1517 ± 199** | 1678 ± 209 | 1253 ± 281 | 1442 ± 210 | 908 ± 511 |
| | ES | **1424 ± 232** | 1676 ± 195 | 1143 ± 366 | 1221 ± 282 | 1013 ± 545 |
| | GD | **1384 ± 276** | 1668 ± 207 | 1266 ± 345 | 1463 ± 208 | 1027 ± 533 |
| Small (TD(λ)) | CE | **1529 ± 202** | 1690 ± 199 | 1420 ± 304 | 1438 ± 197 | 959 ± 581 |
| | SA | **1418 ± 211** | 1661 ± 241 | 1425 ± 323 | 1494 ± 198 | 934 ± 586 |
| | ES | **1489 ± 220** | 1682 ± 193 | 1277 ± 323 | 1419 ± 218 | 1048 ± 512 |
| | GD | **1311 ± 248** | 1643 ± 294 | 1442 ± 295 | 1486 ± 210 | 1064 ± 511 |
| Large (TD1) | CE | **1476 ± 125** | 1694 ± 173 | 1151 ± 243 | 1161 ± 176 | 1004 ± 423 |
| | SA | **1390 ± 107** | 1729 ± 197 | 1117 ± 263 | 1114 ± 185 | 1026 ± 426 |
| | ES | **1425 ± 95** | 1704 ± 171 | 1158 ± 269 | 1100 ± 146 | 925 ± 427 |
| | GD | **1042 ± 102** | 1697 ± 236 | 1134 ± 251 | 1101 ± 163 | 1004 ± 369 |
| Large (TD(λ)) | CE | **1473 ± 128** | 1692 ± 165 | 1080 ± 240 | 1184 ± 175 | 1089 ± 407 |
| | SA | **1480 ± 126** | 1705 ± 165 | 1121 ± 250 | 1187 ± 145 | 1010 ± 434 |
| | ES | **1346 ± 117** | 1708 ± 188 | 1159 ± 263 | 1152 ± 149 | 933 ± 460 |
| | GD | **1093 ± 130** | 1729 ± 192 | 1056 ± 255 | 1129 ± 185 | 1018 ± 441 |

Although ES and CE show a performance increase over evaluated individuals, the finally best-found parameters by SA and GD also yield a high cumulative reward on the final evaluation. This is because SA does not in cooperate a population size in the sense as CE does. In the case of SA these are individual annealing processes, where some manage to find a good region with the parameter space and some not. The final parameter set produced by SA is the best individual among those parallel runs, which can lead to a high fitness as well.
All in all, it can be observed that the proposed network, with the chosen learning rules, is able to learn the given MDPs and can compete with the reference algorithms in a software simulation.

### 5.3.3 Random Maze tasks

The non-trivial maze class which is considered in this thesis is the class of 5x5 mazes. 10000 steps are used for training and the amount of goal visits within those steps is used as the fitness measure of the optimizee. Similar to the MDP case, random parameters are not able to solve the problem within this given step limit. Figure 5.9 shows the comparison of the different LTL algorithms using TD($\lambda$)-Learning. This learning rule was used because it shows faster learning and a slightly better overall performance.

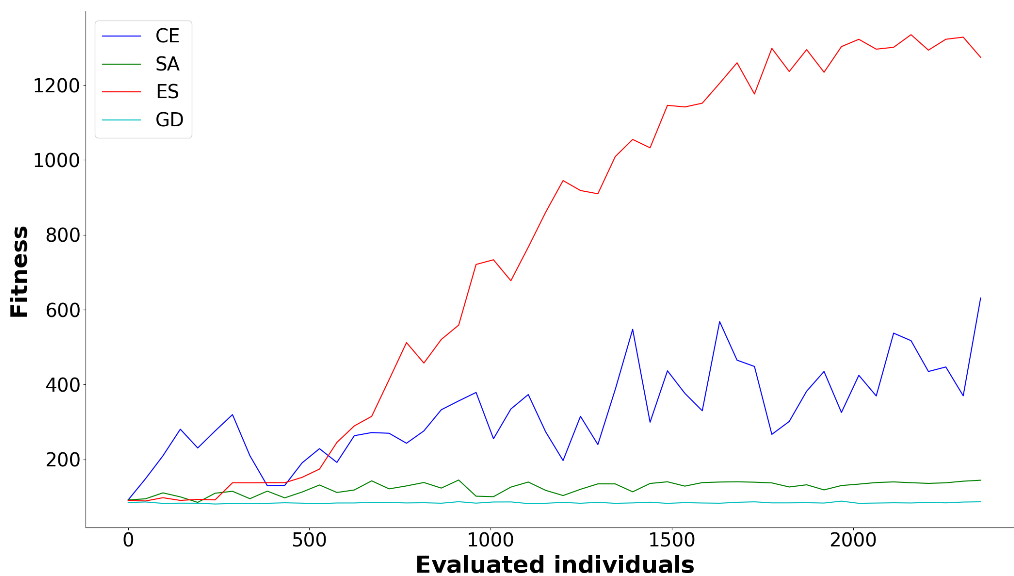


*Figure 5.9: Fitness over evaluated individuals for TD($\lambda$)-Learning with different LTL algorithms and for mazes of size 5 x 5.Similar to the case of MDPs, CE and ES achieve a high performance improvement, while GD and SA cannot compete.*

Again, to evaluate the performance after and before training, 50 random mazes are generated and solved by the agent. Figure 5.10 shows one example out of these were the network with random hyperparameters is compared to the network with optimized hyperparameters.

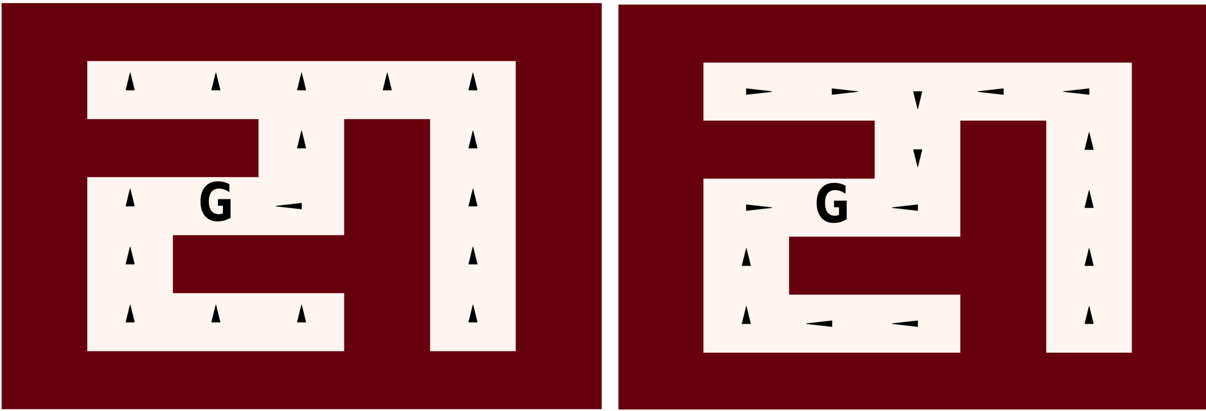


*Figure 5.10: Comparison of network with random and with optimized hyperparameters. On the left-hand side, the policy after training with random hyperparameters is shown, while on the right-hand side optimized hyperparameters were used where CE was used as the LTL algorithm. Without tuned hyperparameters the network is not able to learn the task within the given step limit. In contrast to this, the network learns the optimal policy within the same limit when using optimized hyperparameters.*

One can see that the agent is not able to learn the task within the given number of steps using randomly chosen hyperparameters. Learning to Learn brings an improvement here in the sense that the agent is able to improve learning speed. With this approach, the agent manages to visit the goal more often and finds a better policy.

## 5.4 Hardware simulation

As already mentioned, the hardware simulation was carried out on a neuromorphic hardware. All evaluation and plot scripts were developed in such a way, that results from both platforms can be handled in a straight forward manner.

### 5.4.1 Fixed MDP task

In contrast to the software model, the hardware implementation does not have access to the exact spike times and therefore it is also not trivial to show that the network is able to learn the artificial MDP class.
However, the network is also able to learn the artificial task with randomly chosen hyperparameters. Similar to the software simulation, 2000 steps are used for learning and the agent should maximize the cumulative reward for the given problem.
Figure 5.11 shows a raster plot before and after training. As described in 4.1.1, only action 3 and 2 yield a positive reward for this class. One can see that the network learns to use the correct actions after training because only the corresponding two action neurons spike.

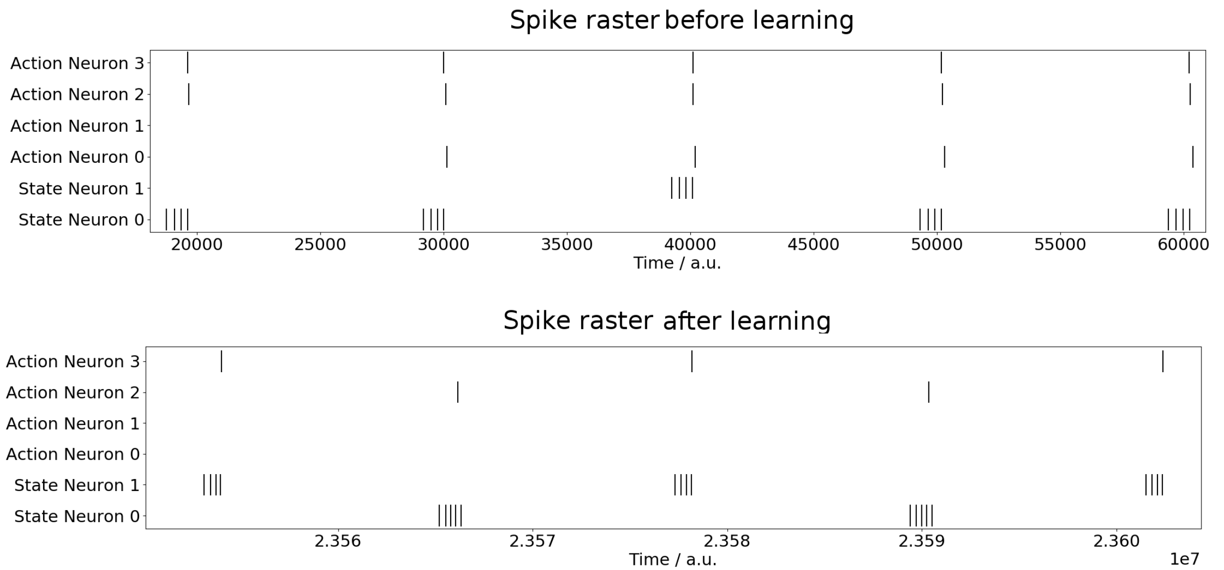


*Figure 5.11: Comparison of spike rasters before and after learning for the "Fixed" MDP with random hyperparameters. The upper spike raster shows the situation before learning. The periodic spiking behavior of the state neurons, due to the self-exciting connections, is clearly visible. Before learning, the wrong, or even multiple action neurons spike within one state. In contrast to that the lower part shows the situation after learning where only one action neuron per state emits a spike. This demonstrates that the network is able to solve the class of "Fixed" MDP with randomly selected parameters, similar to the software model.*

Although the network is already able to learn the task, the time needed to learn the optimal policy is larger compared to the reference algorithms (see figure 5.12). To address this issue, CE as a LTL algorithm was used to tune the hyperparameters which improve the learning speed of the network. With those hyperparameters, it is able to compete with the reference algorithms as shown in figure 5.13.

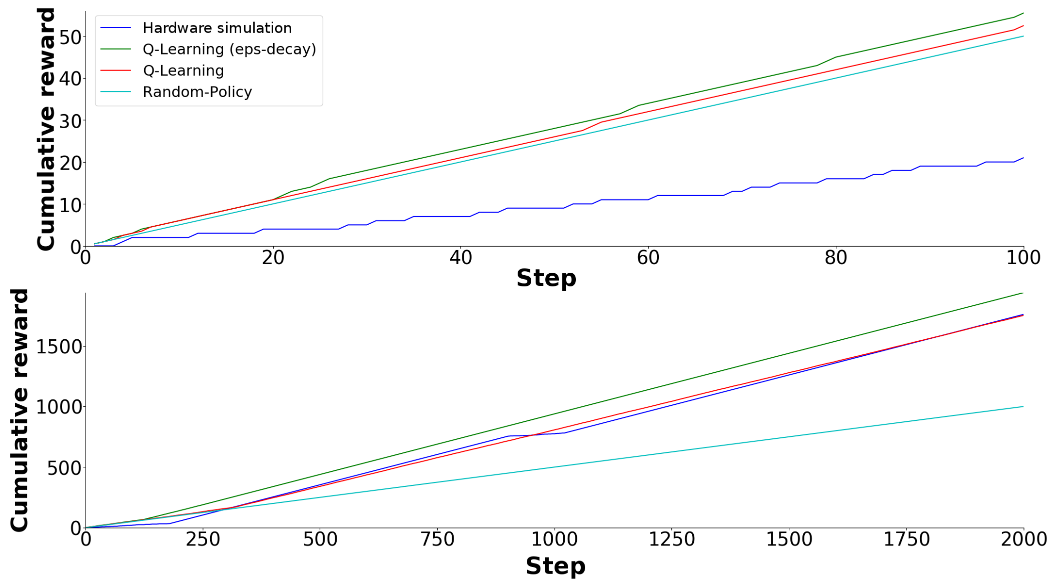


*Figure 5.12: Performance comparison of network with randomly chosen hyperparameters for the "Fixed" MDP class. The upper part shows a zoom on the first 100 steps while the lower part shows the full 2000 steps. The network is able to learn the problem, but slower than the reference algorithms, as one can see from the zoomed part.*

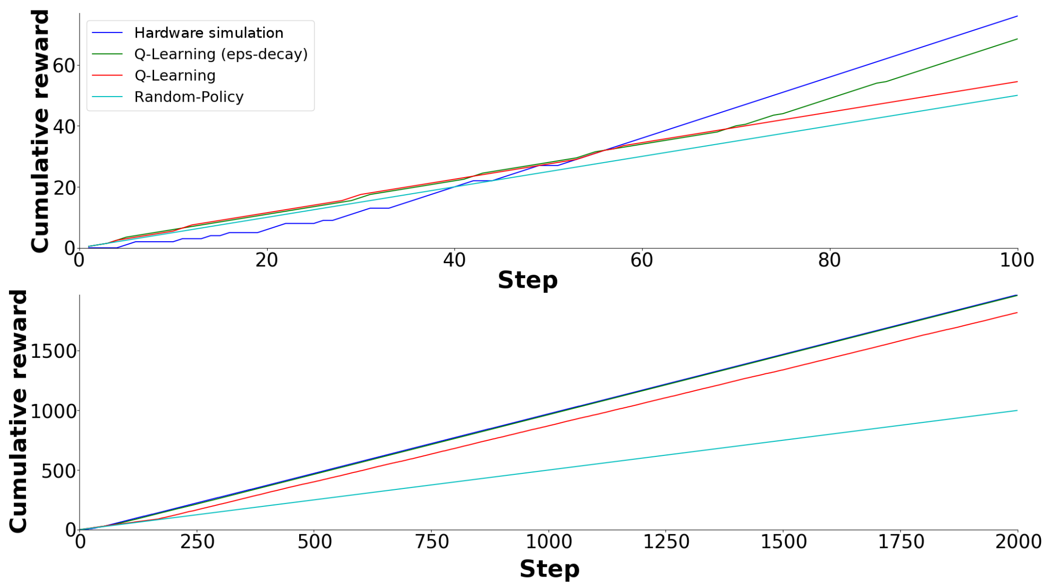


*Figure 5.13: Performance comparison of network with optimized hyperparameters for the "Fixed" MDP class. The upper part shows a zoom on the first 100 steps while the lower part shows the full 2000 steps. The network is able to learn the problem faster and can compete with the reference algorithms.*

### 5.4.2 Random MDPs

The class of random MDPs provides again a non-trivial problem class, where randomly selected hyperparameters are not sufficient anymore. The experimental setup used in this case is the same as for the software simulation. Besides the same fitness measure, the same number of steps, the same random MDPs and the same learning algorithms, also the same LTL algorithms are used.
The fitness evolution over evaluated individuals for different problem classes and learning algorithms can be found in figure 5.14, 5.15, 5.16 and 5.17.

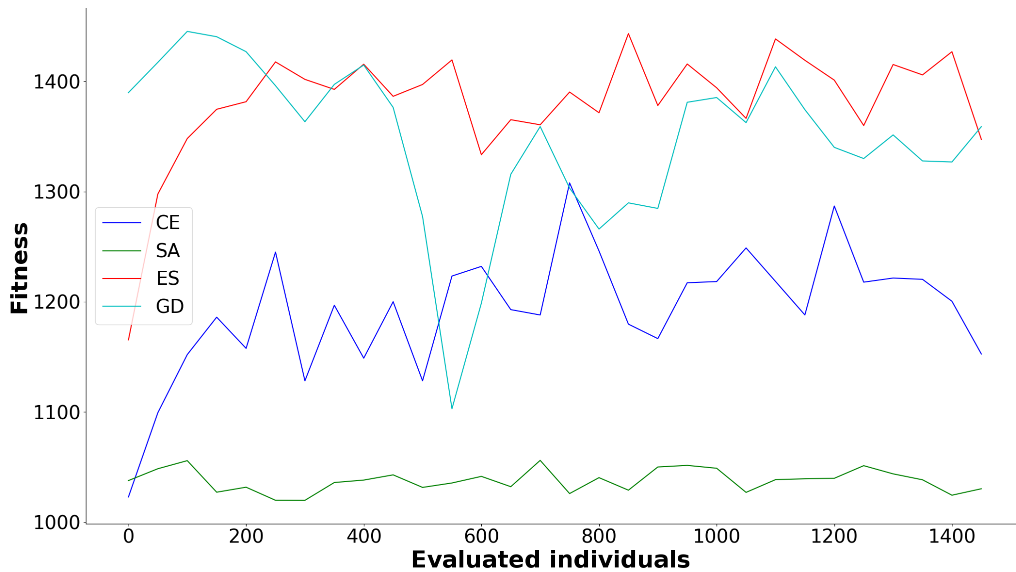


*Figure 5.14: Fitness over evaluated individuals for TD-Learning with different LTL algorithms and for* $||\mathbb{S}|| = 2$ *and* $||\mathbb{A}|| = 4$*. In contrast to the software model GD also achieves a good performance on this task. CE and ES both also work well on the hardware.*

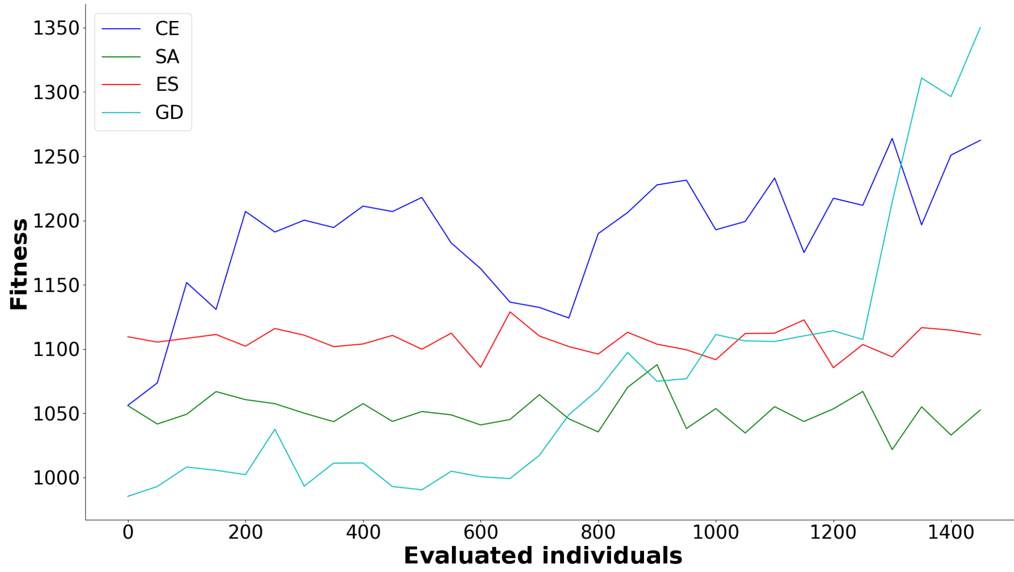


*Figure 5.15: Fitness over evaluated individuals for TD(λ)-Learning with different LTL algorithms and for $||\mathbb{S}|| = 2$ and $||\mathbb{A}|| = 4$*

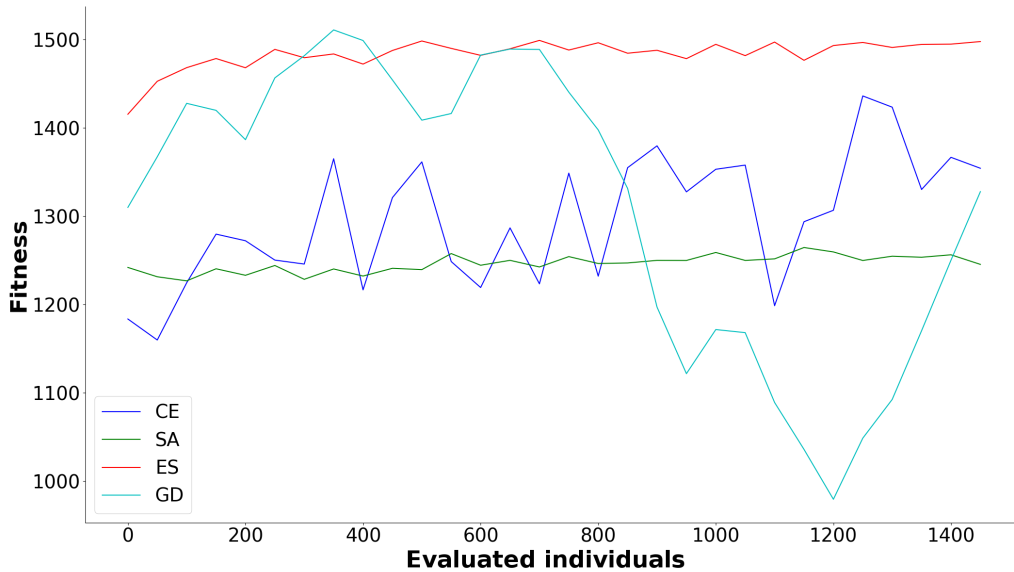


*Figure 5.16: Fitness over evaluated individuals for TD-Learning with different LTL algorithms and for $||\mathbb{S}|| = 6$ and $||\mathbb{A}|| = 8$. This figure shows the fitness increase over evaluated individuals of the LTL algorithms.*

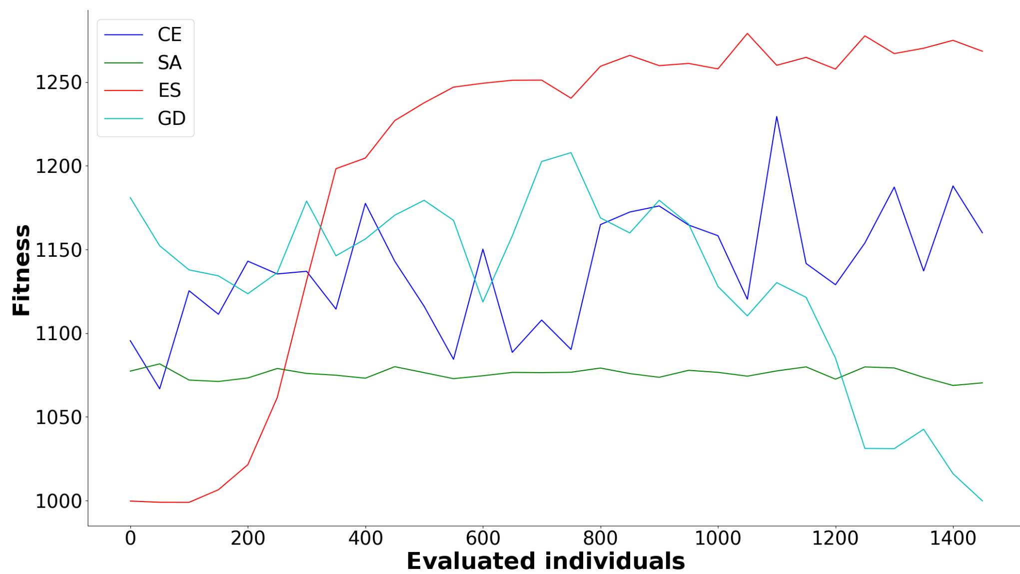


*Figure 5.17: Fitness over evaluated individuals for TD(λ)-Learning with different LTL algorithms and for $||\mathbb{S}|| = 6$ and $||\mathbb{A}|| = 8$. This figure shows the fitness increase over evaluated individuals of the LTL algorithms.*

As one can see, SA is on average not able to bring an improvement on any of the stated problem classes. Nevertheless, the best-found parameters of SA result in a high fitness for certain problem classes (see 5.3), similar to the software model.
GD on the other hand is in some problem classes able to bring an improvement which is not observed in the software simulations. One possible reason for this behavior is the different fitness landscape of the hardware model and the software model.
CE as well as ES show a performance improvement in all problem classes like in the software model. Especially ES is able to cope with the noisy fitness landscape and yields a good performance improvement.

Again, the agent is trained on 50 random MDPs for each combination of each class, of each learning algorithms and of each LTL algorithm. These values are collected and shown in the performance table 5.3:

*Table 5.3: Comparison of performances for different problem classes, reference algorithms, learning and LTL algorithms*

VI ... Value Iteration, QL ... Q-Learning with adaptive epsilon, QLF ... Q-Learning with fixed epsilon, Rnd ... Random policy

| Class | LTL-Algorithm | SNN | VI | QL | QLF | Rnd |
|---|---|---|---|---|---|---|
| Small (TD1) | CE | **1503 ± 286** | 1645 ± 254 | 1432 ± 272 | 1485 ± 204 | 1022 ± 517 |
| | SA | **1481 ± 300** | 1645 ± 254 | 1429 ± 274 | 1488 ± 204 | 1004 ± 542 |
| | ES | **1515 ± 257** | 1665 ± 200 | 1357 ± 320 | 1443 ± 183 | 955 ± 520 |
| | GD | **1361 ± 295** | 1665 ± 200 | 1291 ± 293 | 1449 ± 206 | 1055 ± 586 |
| Small (TD($\lambda$)) | CE | **1477 ± 291** | 1645 ± 254 | 1359 ± 314 | 1473 ± 195 | 888 ± 522 |
| | SA | **1091 ± 397** | 1672 ± 200 | 1100 ± 398 | 1348 ± 274 | 1034 ± 527 |
| | ES | **1444 ± 228** | 1674 ± 197 | 1316 ± 380 | 1436 ± 213 | 920 ± 491 |
| | GD | **1035 ± 297** | 1645 ± 254 | 1448 ± 265 | 1458 ± 211 | 1089 ± 507 |
| Large (TD1) | CE | **1518 ± 171** | 1669 ± 280 | 1175 ± 232 | 1212 ± 111 | 991 ± 442 |
| | SA | **1440 ± 185** | 1670 ± 275 | 1195 ± 261 | 1199 ± 159 | 989 ± 482 |
| | ES | **1482 ± 156** | 1669 ± 280 | 1164 ± 225 | 1200 ± 155 | 1013 ± 516 |
| | GD | **1316 ± 144** | 1712 ± 248 | 1124 ± 292 | 1162 ± 164 | 1061 ± 484 |
| Large (TD($\lambda$)) | CE | **1273 ± 166** | 1675 ± 191 | 1207 ± 273 | 1194 ± 152 | 981 ± 402 |
| | SA | **1272 ± 158** | 1675 ± 191 | 1206 ± 235 | 1169 ± 145 | 894 ± 461 |
| | ES | **1326 ± 148** | 1673 ± 191 | 1181 ± 214 | 1180 ± 165 | 896 ± 465 |
| | GD | **995 ± 127** | 1713 ± 205 | 1134 ± 278 | 1101 ± 140 | 1034 ± 493 |

### 5.4.3 Random Maze tasks

The experimental setup for the maze class differs from the software simulation in the sense that only mazes with the size of 3x3 are considered. This is because larger mazes result in problems with the underlying hardware.
Despite the fact that only 4000 steps were used for learning, similar to the software model, the amount of goal visits within these steps is used as the fitness measure and TD($\lambda$)-Learning was used for learning. Figure 5.18 shows the fitness evolutions over evaluated individuals for the LTL algorithms.

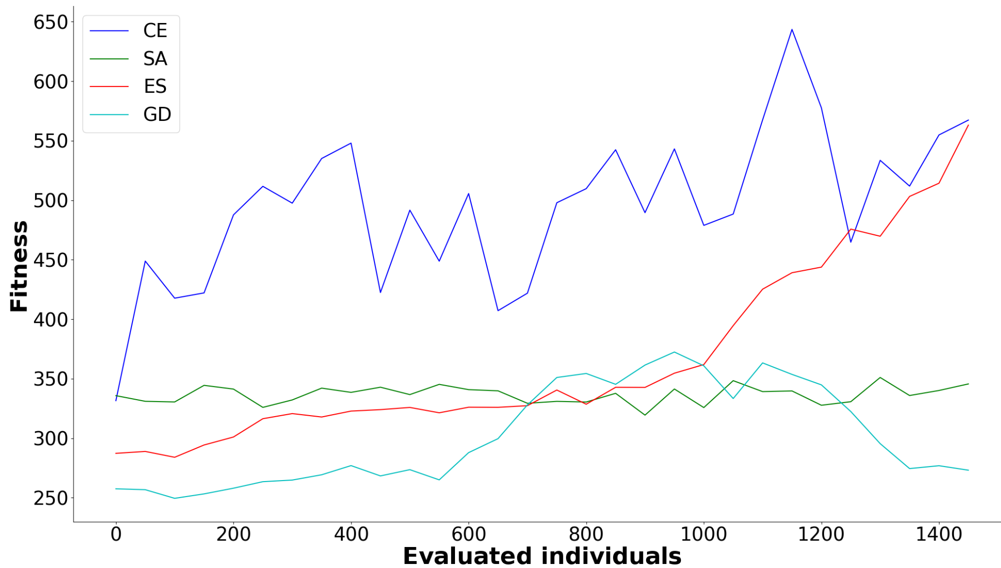


*Figure 5.18: Fitness over evaluated individuals for TD($\lambda$)-Learning with different LTL algorithms for a maze of size 3 x 3. Also, this figure shows similar behavior to the software equivalent. CE and ES yields a high performance increase and lead to parameters, which can be used to find the optimal policy for the given mazes.*

Again, 50 randomly generated mazes are used to evaluate the performance of the agent before and after applying LTL. Figure 5.19 shows one example out of these, were the network with random hyperparameters is compared to the network with optimized hyperparameters.

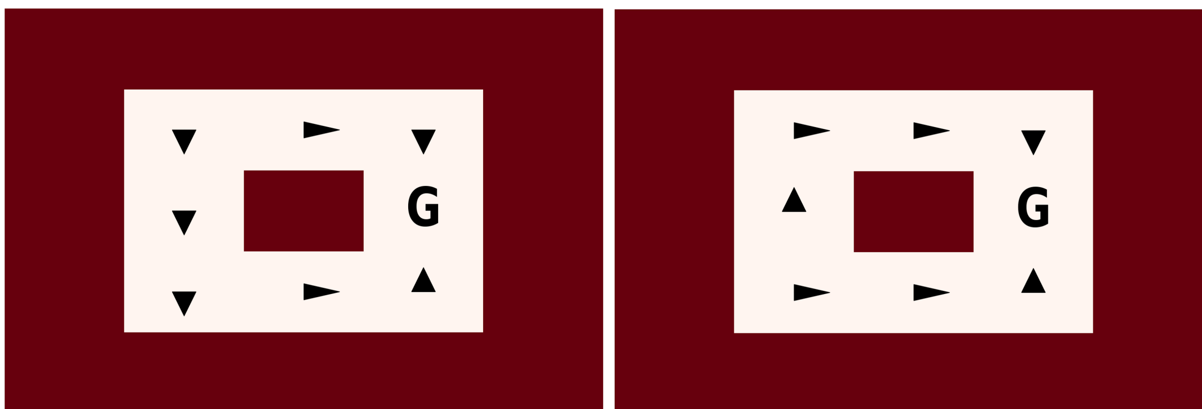


*Figure 5.19: Comparison of network with random and with optimized hyperparameters. On the left-hand side, the policy after training with random hyperparameters is shown, while on the right-hand side optimized hyperparameters were used with CE as the LTL algorithm. The network with randomly chosen hyperparameters is not able to learn the optimal policy within the given step limit, but the network using tuned hyperparameter is able.*

## 5.5 Comparison

The results for the MDP classes are comparable between the software and the hardware implementation. The cumulative reward for the different learning algorithms shows similar behavior. Figure 5.20 and 5.21 show direct comparisons between the software and the hardware model for 50 random MDPs, with different sizes, after applying ES as the LTL algorithm.

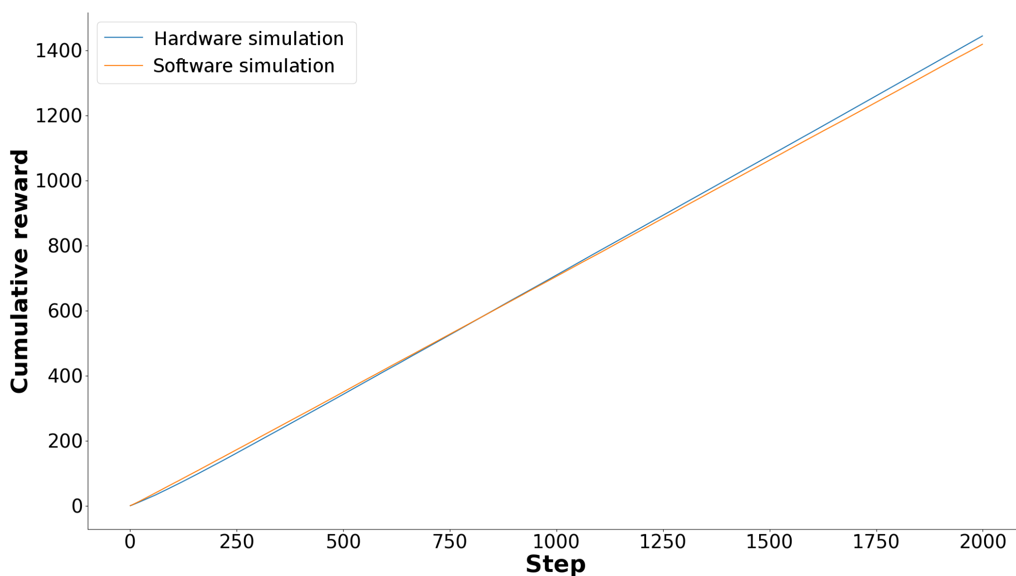


*Figure 5.20: Comparison of average performance of hardware and software model for MDPs of size $||\mathbb{S}|| = 2$ and $||\mathbb{A}|| = 4$. In both cases TD($\lambda$)-Learning and the same 50 random MDPs were used after ES was applied to tune the hyperparameters. The lines represent the average cumulative reward over those 50 random problems and only a minor difference can be observed. This supports the fact that the hardware implementation provides a similar performance than the software model.*

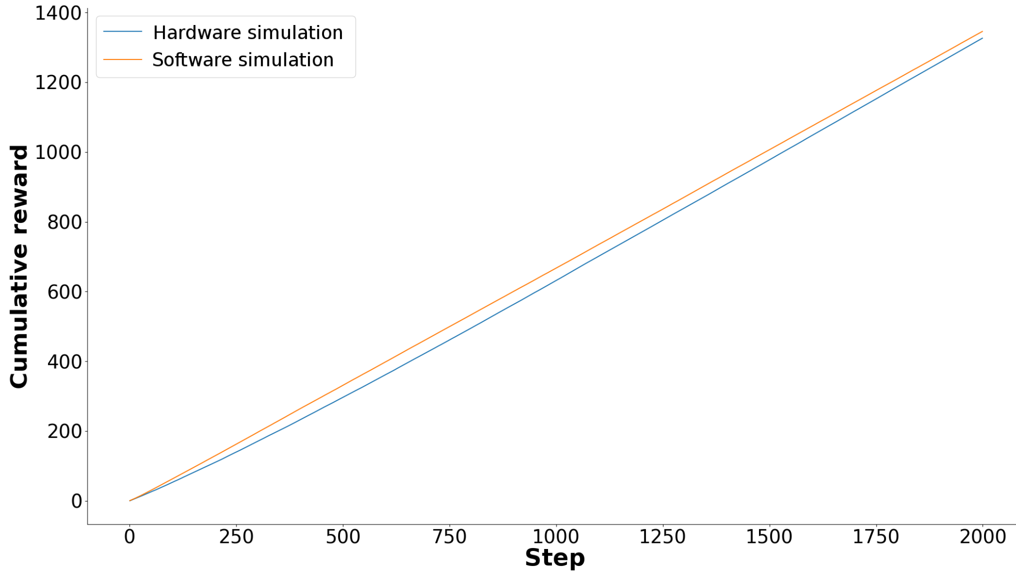


*Figure 5.21: Comparison of average performance of hardware and software model for MDPs of size $||\mathbb{S}|| = 6$ and $||\mathbb{A}|| = 8$. In both cases TD($\lambda$)-Learning and the same 50 random MDPs were used after ES was applied to tune the hyperparameters. The lines represent the average cumulative reward over those 50 random problems. For the larger problem class, there is a slight derivation in performance between software and hardware.*

As one can see from the figures, the mean cumulative reward for the software and the hardware model are in good agreement. However, when the problem difficulty increases, the software benefits from the access to the precise spike times and the hardware implementation loses performance.

For the maze tasks a comparison is not directly possible, because the considered problem size are different. Nevertheless, the evaluation of both individually showed that solving random mazes is also possible in software as well as on hardware.

# 6 Discussion and Outlook

## 6.1 Discussion

The results show that the presented approach can be used to endow a network of spiking neurons to solve MDPs. As one can see from the performance tables 5.2 and 5.3, the implemented method is able to compete with the chosen reference algorithms, clearly outperforms a random policy and also works on the neuromorphic hardware. Although the hardware limits the possible problem sizes, the provided experiments show a proof-of-concept for non-trivial problems.

One key concept of this approach, provided on hardware, is an implicit implementation of working memory. Since the weights are values stored in the digital part of the chip, they are not subject to any volatility. Working memory is important in many applications like image recognition or complex planning processes [Baddeley, 2012], [Burgess and Hitch, 2005].
One construction of working memories with SNNs is done in [Giulioni et al., 2012] by using a population of self-excitatory connected LIF neurons. The memory is represented by bistable attractor states within such a population. The activity of the network is either high or low, corrects itself through the self-excitatory connections and the value stored can be manipulated from outside.
Such approaches have the problem of keeping the correct information present inside the "memory" cell. A proper network activity is needed, while in the present case, the desired values can be directly written to the "memory" and read without further interference.

Despite the resource limitation, Learning to Learn can improve the performance compared to handcrafted or even random hyperparameters. This becomes significant when the hyperparameter space becomes larger or the fitness landscape hardly allows for manual hyperparameter finding. Especially on the neuromorphic hardware LTL is relevant, because it was hard to manually find parameters that worked even for a single maze or a larger random MDP.

An additional point regarding the LTL algorithms used is that they are also parameterized. Those parameters, here called hyperhyperparameters, can be essential for the algorithm to produce good results and were set by hand. It is possible that algorithms like SA or GD would achieve better results if the hyperhyperparameters are also tuned by another optimization loop. However, this scenario would require even more evaluations of individuals and was therefore not considered.

# Training time

The computing times required to carry out all the present experiments are listed in this section. For example, the execution of one random MDP experiment with $||\mathbb{S}|| = 6$ and $||\mathbb{A}|| = 8$ on hardware only lasts for a few seconds. However, for the learning to learn framework the evaluation of thousands of individuals was necessary resulting in a runtime of approximately nine hours per LTL algorithm. Together with the variety of problem classes and LTL algorithm, the collection of the results was a time-consuming process. This situation is even worse for the software simulation, since the execution of one single experiment takes longer than compared to the hardware. In the following table following measures are shown:

- Class: indicates the problem class
- Learning algorithm: indicates the learning rule used for the weight updates
- Runtime: indicates the pure runtime for learning a single problem instance without any over runtime overhead from initializations and cleanup. The value is computed over 50 different runs
- LTL Runtime: indicates the overall runtime when applying LTL algorithms on one single CPU core. This value is computed over the four used LTL algorithms

*Table A.1: Training time*

| Class | Learning algorithm | Runtime | | LTL Runtime | |
|---|---|---|---|---|---|
| | | SW / s | HW / s | SW / h | HW / h |
| Small MDP | TD1 | $3.17 \pm 0.37$ | $1.15 \pm 0.02$ | $105.86 \pm 4.91$ | $10.66 \pm 0.54$ |
| | TD($\lambda$) | $4.87 \pm 0.05$ | $1.20 \pm 0.02$ | $180.40 \pm 10.89$ | $11.17 \pm 0.86$ |
| Large MDP | TD1 | $5.31 \pm 0.04$ | $1.20 \pm 0.01$ | $188.22 \pm 9.12$ | $14.70 \pm 4.00$ |
| | TD($\lambda$) | $18.74 \pm 0.19$ | $1.21 \pm 0.01$ | $655.41 \pm 32.56$ | $15.97 \pm 2.74$ |
| Maze 3x3 | TD($\lambda$) | - | $3.60 \pm 0.02$ | - | $27.41 \pm 2.29$ |
| Maze 5x5 | TD($\lambda$) | $1833.84 \pm 7.54$ | - | $3217.72 \pm 117.49$ | - |

The hardware framework has a runtime which is not dependent on the size of the MDP, which is different for the software simulation. Therefore, the hardware is beneficial for the application of LTL as it provides a more constant and overall shorter runtime for the LTL algorithms. Note also that the overall LTL runtime of the software simulations is computed using a single core only. This runtime can be reduced when using more cores and a parallel execution.

# Bibliography

Simonite Tom, “Intel Puts the Brakes on Moore’s Law - MIT Technology Review,” 2016. [Online]. Available: https://www.technologyreview.com/s/601102/intel-puts-the-brakes-on-moores-law/

C. D. Schuman, T. E. Potok, R. M. Patton, J. D. Birdwell, M. E. Dean, G. S. Rose, and J. S. Plank, “A Survey of Neuromorphic Computing and Neural Networks in Hardware.” [Online]. Available: https://arxiv.org/pdf/1705.06963.pdf

D. Monroe, “Neuromorphic computing gets ready for the (really) big time,” *Communications of the ACM*, vol. 57, no. 6, pp. 13–15, jun 2014. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2602695.2601069

A. Calimera, E. Macii, and M. Poncino, “The Human Brain Project and neuromorphic computing.” *Functional neurology*, vol. 28, no. 3, pp. 191–6, 2013. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/24139655http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC3812737

F. Y. H. Ahmed, B. Yusob, H. Nuzly, and A. Hamed, “Computing with Spiking Neuron Networks A Review,” *Int. J. Advance. Soft Comput. Appl*, vol. 6, no. 1, 2014. [Online]. Available: https://pdfs.semanticscholar.org/ecba/7b5a713cf3b175b78e59f0789f851245d01b.pdf

E. O. Neftci, C. Augustine, S. Paul, and G. Detorakis, “Event-Driven Random Back-Propagation: Enabling Neuromorphic Deep Learning Machines.” *Frontiers in neuroscience*, vol. 11, p. 324, 2017. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/28680387http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC5478701

J. H. Lee, T. Delbruck, and M. Pfeiffer, “Training Deep Spiking Neural Networks using Backpropagation.” [Online]. Available: https://arxiv.org/pdf/1608.08782.pdf

L. P. Kaelbling, M. L. Littman, and A. R. Cassandra, “Planning and acting in partially observable stochastic domains,” *Artificial Intelligence*, vol. 101, no. 1-2, pp. 99–134, may 1998. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S000437029800023X

J. Brea and W. Gerstner, “Does computational neuroscience need new synaptic learning paradigms?” *Current Opinion in Behavioral Sciences*, vol. 11, pp. 61–66, oct 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2352154616301048

J. Friedrich and M. Lengyel, “Goal-Directed Decision Making with Spiking Neurons.” *The Journal of neuroscience : the official journal of the Society for Neuroscience*, vol. 36, no. 5, pp. 1529–46, feb 2016. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/26843636http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC4737768

T. Nakano, M. Otsuka, J. Yoshimoto, and K. Doya, “A Spiking Neural Network Model of Model-Free Reinforcement Learning with High-Dimensional Sensory Input and Perceptual Ambiguity,” *PLOS ONE*, vol. 10, no. 3, p. e0115620, mar 2015. [Online]. Available: http://dx.plos.org/10.1371/journal.pone.0115620

M. Shein, “A spiking neural network of state transition probabilities in model-based reinforcement learning,” p. 69, oct 2017. [Online]. Available: https://uwspace.uwaterloo.ca/handle/10012/12574

HBP, “Human Brain Project Home.” [Online]. Available: https://www.humanbrainproject.eu/en/

K. Heidelberg, “Digital Learning System.” [Online]. Available: https://www.kip.uni-heidelberg.de/vision/research/dls/

S. Friedmann, J. Schemmel, A. Grubl, A. Hartel, M. Hock, and K. Meier, “Demonstrating Hybrid Learning in a Flexible Neuromorphic Hardware System,” *IEEE Transactions on Biomedical Circuits and Systems*, vol. 11, no. 1, pp. 128–142, feb 2017. [Online]. Available: http://ieeexplore.ieee.org/document/7563782/

S. A. Aamir, P. Muller, A. Hartel, J. Schemmel, and K. Meier, “A highly tunable 65-nm CMOS LIF neuron for a large scale neuromorphic system,” in *ESSCIRC Conference 2016: 42nd European Solid-State Circuits Conference*. IEEE, sep 2016, pp. 71–74. [Online]. Available: http://ieeexplore.ieee.org/document/7598245/

S. A. Aamir, P. M\"uller, L. Kriener, G. Kiene, J. Schemmel, K. Meier, . A. Aamir, P. M\"uller, and and L. Kriener and G. Kiene and J. Schemmel and K. Meier, “From LIF to AdEx Neuron Models: Accelerated Analog 65 nm CMOS Implementation,” in *IEEE Biomedical Circuits and Systems Conference (BioCAS)*, 2017. [Online]. Available: http://www.kip.uni-heidelberg.de/Veroeffentlichungen/details.php?id=3518

D. Stöckel, “Exploring Collective Neural Dynamics under Synaptic Plasticity,” Masterarbeit, Universität Heidelberg, 2017.

Y. Stradmann, “Characterization and Calibration of a Mixed-Signal Leaky Integrate and Fire Neuron on HICANN-DLS,” Bachelorarbeit, Universität Heidelberg, 2016.

R. Bellman, *Dynamic programming*. Princeton University Press, 2010. [Online]. Available: https://press.princeton.edu/titles/9234.html

R. S. Sutton and A. G. Barto, *Reinforcement learning : an introduction*. MIT Press, 1998. [Online]. Available: https://mitpress.mit.edu/books/reinforcement-learning

Richard S. Sutton and Andrew G. Barto, “Chapter 7: Eligibility Traces,” p. 38. [Online]. Available: http://www-anw.cs.umass.edu/{~}barto/courses/cs687/Chapter7.pdf

S. P. Singh and R. S. Sutton, “Reinforcement learning with replacing eligibility traces,” *Machine Learning*, vol. 22, no. 1-3, pp. 123–158, 1996. [Online]. Available: http://link.springer.com/10.1007/BF00114726

C. S. Szepes ari Richard Sutton, “Reinforcement Learning Algorithms in Markov Decision Processes AAAI-10 Tutorial Part II: Learning to predict values,” 2010. [Online]. Available: http://old.sztaki.hu/{~}szcsaba/research/AAAI10{_}Tutorial/tutorial02-slidesonly.pdf

C. J. C. H. Watkins and P. Dayan, “Q-learning,” *Machine Learning*, vol. 8, no. 3-4, pp. 279–292, may 1992. [Online]. Available: http://link.springer.com/10.1007/BF00992698

P. Dayan and T. J. Sejnowski, “TD($\lambda$) Converges with Probability 1,” *Machine Learning*, vol. 14, no. 3, pp. 295–301, 1994. [Online]. Available: http://link.springer.com/10.1023/A:1022657612745

P. Dayan, “The Convergence of TD(X) for General X,” *Machine Learning*, vol. 8, pp. 341–362, 1992. [Online]. Available: https://link.springer.com/content/pdf/10.1023/A:1022632907294.pdf

M. Andrychowicz, M. Denil, S. Gomez, M. W. Hoffman, D. Pfau, T. Schaul, B. Shillingford, and N. de Freitas, “Learning to learn by gradient descent by gradient descent,” jun 2016. [Online]. Available: http://arxiv.org/abs/1606.04474

Y. Chen, M. W. Hoffman, S. Gómez Colmenarejo, M. Denil, T. P. Lillicrap, M. Botvinick, and N. De Freitas, “Learning to Learn without Gradient Descent by Gradient Descent.” [Online]. Available: http://proceedings.mlr.press/v70/chen17e/chen17e.pdf

Y. Duan, J. Schulman, X. Chen, P. L. Bartlett, I. Sutskever, and P. Abbeel, “FAST REINFORCEMENT LEARNING VIA SLOW REINFORCEMENT LEARNING.” [Online]. Available: https://arxiv.org/pdf/1611.02779.pdf

C. Wang, S. Guo, Y. Xu, J. Ma, J. Tang, F. Alzahrani, and A. Hobiny, “Formation of Autapse Connected to Neuron and Its Biological Function,” *Complexity*, vol. 2017, pp. 1–9, feb 2017. [Online]. Available: https://www.hindawi.com/journals/complexity/2017/5436737/

A. Hartel, “Implementation and Characterization of Mixed-Signal Neuromorphic ASICs,” Ph.D. dissertation, 2016. [Online]. Available: http://www.ub.uni-heidelberg.de/archiv/20179

P.-T. de Boer, D. P. Kroese, S. Mannor, and R. Y. Rubinstein, “A Tutorial on the Cross-Entropy Method,” *Annals of Operations Research*, vol. 134, no. 1, pp. 19–67, feb 2005. [Online]. Available: http://link.springer.com/10.1007/s10479-005-5724-z

D. Wierstra, T. Schaul, T. Glasmachers, J. Peters, and J. Schmidhuber, “Natural Evolution Strategies,” *Journal of Machine Learning Research*, vol. 15, pp. 949–980, 2014. [Online]. Available: http://www.jmlr.org/papers/volume15/wierstra14a/wierstra14a.pdf

T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, “Evolution Strategies as a Scalable Alternative to Reinforcement Learning,” mar 2017. [Online]. Available: http://arxiv.org/abs/1703.03864

S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, “Optimization by Simulated Annealing,” pp. 671–680. [Online]. Available: http://www.jstor.org/stable/1690046

J. Fernando Díaz Martín and J. M. Riaño Sierra, “A Comparison of Cooling Schedules for Simulated Annealing,” in *Encyclopedia of Artificial Intelligence*. IGI Global, jan 1, pp. 344–352. [Online]. Available: http://services.igi-global.com/resolvedoi/resolve.aspx?doi=10.4018/978-1-59904-849-9.ch053

A. Baddeley, “Working Memory: Theories, Models, and Controversies,” *Annual Review of Psychology*, vol. 63, no. 1, pp. 1–29, jan 2012. [Online]. Available: http://www.annualreviews.org/doi/10.1146/annurev-psych-120710-100422

N. Burgess and G. Hitch, “Computational models of working memory: putting long-term memory into context,” *Trends in Cognitive Sciences*, vol. 9, no. 11, pp. 535–541, nov 2005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1364661305002731

M. Giulioni, P. Camilleri, M. Mattia, V. Dante, J. Braun, and P. Del Giudice, “Robust Working Memory in an Asynchronously Spiking Neural Network Realized with Neuromorphic VLSI,” *Frontiers in Neuroscience*, vol. 5, p. 149, feb 2012. [Online]. Available: http://journal.frontiersin.org/article/10.3389/fnins.2011.00149/abstract