TU Graz
Graz University of Technology

Franz Scherr, BSc BSc

Spike-based Agents for Multi-armed Bandits

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 of Theoretical Computer Science
Graz University of Technology, Austria

In corporation with Heidelberg University, Germany

Graz, February 23, 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.

date (signature)

Abstract

A key challenge for mobile devices is energy consumption. Despite the successes of digital integrated circuitry, devices of the like contribute significantly to the exhaustion of batteries. The flagship of a low power, high-end computer is the human brain. In this work the properties of the key components, neurons and synapses, are exploited in order to compose a network of biologically inspired neurons that can handle the multi-armed bandit problem as a unit. The synapses in this designed network are altered by the means of specifically developed plasticity rules that allow the network to interact with the bandit task and learn from reward observations. The obtained network model is simulated on novel neuromorphic prototype hardware located at Heidelberg University. Furthermore, the plasticity rules are characterized by a number of free hyperparameters, each of which adjusts certain aspects. Stochastic optimization methods are applied to find values for the hyperparameters that work well not only with a single multi-armed bandit task but instead, work well on an entire family of such tasks. By the process of hyperparameter optimization, experience at previous bandit tasks is transferred to new instances. The results of this thesis include a spiking neural network scaffold for handling the multi-armed bandit problem, if equipped with appropriate plasticity rules that are also presented. Furthermore, the model is developed on neuromorphic hardware and the capability of the resulting spike-based agent to handle the multi-armed bandit problem is demonstrated. It is shown that applying stochastic optimization methods to the hyperparameters, characterizing aspects of plasticity rules and network properties, can significantly improve the agents ability to solve the multi-armed bandit task. In particular, if structure on the considered task family is introduced, the hyperparameters absorb the knowledge and equip the spiking agent with priors about the underlying task family. Another aspect that was considered during this work is to create a freely parametrized synaptic plasticity rule, optimized to work well with multi-armed bandit tasks. It is demonstrated that this approach endows the spiking neural network with the ability to outperform common algorithms on certain families of multi-armed bandit problems.

Kurzfassung

Eine wichtige Herausforderung für mobile Geräte ist der Energieverbrauch. Trotz der Erfolge digitaler Computerchips tragen Schaltungen dieser Art wesentlich zur Erschöpfung von Batterien bei. Das Paradebeispiel eines energieeffizienten, hoch performanten Computers ist das menschliche Gehirn. In dieser Arbeit werden die Eigenschaften der Schlüsselkomponenten, dazu zählen die Neuronen und Synapsen, ausgenützt, um ein biologisch inspiriertes Netzwerk von Neuronen zu bilden, welches das Multi-Armed Bandit Problem behandeln kann. Die Synapsen in diesem Netzwerk werden durch speziell entwickelte Plastizitätsregeln verändert, die es dem Netzwerk ermöglichen mit dem Multi-Armed Bandit zu interagieren und aus Rewards zu lernen. Das so entworfene Netzwerkmodell wird auf neuartiger neuromorpher Prototyphardware an der Universität Heidelberg ausgeführt. Die Plastizitätsregeln sind im weiteren durch eine Vielzahl an Hyperparametern charakterisiert. Stochastische Optimierungsmethoden werden angewendet, um Werte für diese Hyperparameter zu finden, die nicht nur an einem einzelnen Multi-Armed Bandit gut funktionieren sollen, sondern gut für eine ganze Familie solcher Aufgaben. Durch den Prozess der Hyperparameter Optimierung wird die Erfahrung von vorherigen Bandit Aufgaben auf neue Instanzen übertragen. Die Ergebnisse dieser Arbeit beinhalten das Design eines spikenden neuronalen Netzes, dass zur Behandlung des Multi-Armed Bandit Problems herangezogen werden kann. Dies wird durch entsprechende Plastizitätsregeln erreicht, welche ebenfalls als Teil dieser Arbeit vorgestellt werden. Darüber hinaus wird das Modell auf neuromorpher Hardware entwickelt und es wird die Fähigkeit des resultierenden spike-basierten Agenten, das Multi-Armed Bandit Problem zu lösen, demonstriert. Im Weiteren wird gezeigt, dass die Anwendung von stochastischen Optimierungsmethoden auf Hyperparameter, die die Plastizitätsregeln charakterisieren und Netzwerkeigenschaften festlegen, die Performance signifikant verbessern kann. Wenn den Multi-Armed Bandits eine gemeinsame Struktur zu Grunde liegt, kann durch die Optimierung der Hyperparameter die Performance für Probleme selber Struktur gesteigert werden. Eine weitere Einsicht dieser Arbeit ist die Erstellung einer frei parametrisierbaren synaptischen Plastizitätsregel, die auf maximale Performance bei Multi-Armed Bandits optimiert werden kann. Es wird gezeigt, dass dieser Ansatz dem spike-basierten Agenten die Fähigkeit verleiht, herkömmliche Algorithmen bei bestimmten Strukturen von Multi-Armed Bandit Problemen zu übertreffen.

Acknoledgements

At first I would like to thank Em.Univ.-Prof. Dipl.-Ing. Dr.rer.nat. Wolfgang Maass for his support and professional advice. It was also him, who helped me to get in touch with the fields of neuroscience and machine learning. In addition, he also assisted me with this master thesis by words and deeds.

I thank also my friend and colleague Thomas Bohnstingl for providing me with support throughout my entire studies.

Gratitude is devoted to the people of the Institute of Theoretical Computer Science at Graz University of Technology, as they supported me and provided me with advice as well as inspiration.

I thank the members of the Electronic Vision(s) group at Heidelberg University for giving me the opportunity to write this thesis and for the great support throughout. In particular I would like to mention David Stöckel and Christian Pehle.

I would also like to thank all people who supported me in any kind when writing this thesis, especially my relatives and friends. They backed me in different stages of my life and laid the foundation for my education and my further life.

Contents

1 Introduction **11**
1.1 Objective and Delimitation . . . 11
1.2 Background . . . 12
1.3 New Computing Paradigms . . . 12
1.3.1 Networks of Spiking Neurons . . . 13
1.3.2 Neuromorphic Hardware . . . 14
1.4 Multi-armed Bandit Problems . . . 14
1.4.1 Dependent Multi-armed Bandits . . . 15
1.4.2 Policy Performance Measurements . . . 15
1.5 MAB Policies in Literature . . . 16
1.5.1 ϵ-Greedy . . . 16
1.5.2 Bayes Upper Confidence Bound . . . 16
1.5.3 Gittins Indices . . . 17
1.5.4 Optimistic Gittins Indices . . . 18
1.6 Demixed Principal Component Analysis - dPCA . . . 18

2 Learning to Learn **19**
2.1 Optimizers . . . 20
2.1.1 Cross Entropy Method . . . 21
2.1.2 Parallel Simulated Annealing . . . 22
2.1.3 Finite Difference Gradient Method . . . 23
2.1.4 Evolution Strategies . . . 24

3 Spiking Network Architecture **25**
3.1 Network Design . . . 25
3.2 Update Rules . . . 27
3.2.1 Greedy Update . . . 27
3.2.2 Incremental Update . . . 28
3.2.3 Artificial Neural Network Update . . . 28

4 Digital Learning System - HICANN-DLS **32**
4.1 Overview . . . 32
4.2 Hardware Neurons . . . 33
4.3 Hardware Synapses . . . 34
4.3.1 Synapse Array . . . 34
4.4 Analog Parameters - Calibration . . . 35
4.5 Plasticity Processing Unit - PPU . . . 36
4.6 Infrastructure . . . 36
4.7 Limitations and Constraints . . . 38

5 Implementation **39**
5.1 Hardware Implementation . . . 39
5.1.1 Network Mapping . . . 39
5.1.2 Used Hardware Calibration . . . 40
5.1.3 Infrastructure Setup . . . 44

6 Approach Validation **46**
6.1 Greedy Update Rule . . . 47
6.2 Incremental Update Rule . . . 47

7 Learning to Learn Results 48
7.1 Incremental Update Rule . . . 49
7.1.1 Independent Bandits . . . 50
7.1.2 Dependent Bandits . . . 52
7.2 Greedy Update Rule . . . 54
7.2.1 Independent Bandits . . . 55
7.2.2 Dependent Bandits . . . 57
7.3 ANN Update Rule . . . 59
7.4 Comparison . . . 61

8 Regret Evaluation 63
8.1 Greedy Update Rule . . . 64
8.2 Incremental Update Rule . . . 65

9 Conclusion 66

1 Introduction

The human brain is one of the most intriguing systems of our known world. Developed by evolution, it endowed the species of humankind with cognitive abilities, such as to understand problems by thought. In fact, the brain is a sophisticated learning machine. Acquisition of knowledge happens in a continuous process at an early age and, in the ideal case, only stops at the eventual death.

In order to gain deeper insights into the brains function, tremendous research efforts are being made. A multitude of scientists in diverse fields, such as computational neuroscience, medicine, neuromorphic engineering and similar join forces to unravel the mysteries of the human brain. Novel discoveries in this matter are vital for biological purposes and medicine, for instance to cure neurological diseases and similar. In addition to the many advantages thereof, uncovering principles of brain computation paves the way for new computing paradigms in a world eager for ever faster and more powerful computing machines. More importantly however, one key motivation of the study of the human brain is to reveal what aspects make it behave intelligently, in order to build and construct intelligent agents, capable of solving complex problems. For instance, consider autonomously driving cars. It is a must that such vehicles perceive possible hazards along the road and act in an appropriate way, which is to be found out by the car itself. Interest in intelligent systems also comes from the realm of robotics. Industrial robots often require long teach-in periods, programming the trajectory of the robotic arm into the system, before any manufacturing can happen. Interestingly, even the slightest savings in production time can imply significant improvements in profit. Having industrial robots that learn autonomously how to do a task in the best way and at the same time learn from experience of tasks previously done, would have major impacts in industry. Besides medical and economical interests, the advantages of understanding the brains function also reaches to energy consumption. By a matter of fact, the human brain consumes just a fraction of what a standard home office computer requires. Computing paradigms based on the human brain therefore, could mitigate, to some extent, the waste of energy caused by computers. Mobile devices probably suffer the most under unnecessary dissipation, as energy coming from a battery is precious. All of the mentioned use cases justifies investigating principles of brain computation for its own.

1.1 Objective and Delimitation

The intent of this thesis is to clarify the possibility and viability regarding networks of spiking neurons to deal with the **multi-armed bandit** problem (described in Section 1.4). One central point is to find and investigate appropriate update rules of synaptic weights that establish in such neural networks a capability to handle such a task. In addition, it will be a main part of this work to realize the designed neural circuit on prototype neuromorphic hardware (Section 1.3.2) to demonstrate feasibility of this certain application on novel devices.

Another key aspect of this work is then given by exploring possible application of the paradigm of **learning to learn**, also known under the names **meta learning** or **transfer learning** con-

cerning multi-armed bandits. Although precisely described in Chapter 2, the concept is to endow intelligent systems with a more abstract understanding of the structure of the underlying task family and thus, leading to a better learning performance by experience in similar tasks. In the case of multi-armed bandits, task structure can be introduced by making reward probabilities dependent on each other, as will be explained in Section 1.4.1.

1.2 Background

One of the most accessible approaches to intelligent systems is based on artificial neural networks. Well-founded with the theory of backpropagation and backed up by the advent of mighty parallel processors (e.g.: graphics processing units - **GPUs**), the field of machine learning has grown continuously by ever increasing demands from business and industry. Successes of artificial intelligence involve reinforcement learning by function approximation [Mnih et al., 2015], asynchronous policy gradient methods [Mnih et al., 2016], image recognition tasks [He et al., 2016], machine translation [Bahdanau et al., 2014], speech recogition and countless other tasks. Despite its successes however, machine learning techniques have diverged from its biological inspiration and it still remains a largely controversial debate if such artificial models work in a similar way as the human brain does. Recent developments show that innovative approaches are often inspired by certain aspects of the human mind [Hassabis et al., 2017], which again brings in the necessity of understanding the brain.

Unlike artificial models, there is also work devoted to harness computational power from biological inspired neural networks, based on **spiking neurons**. As opposed to artificial ones, the celebrated backpropagation rule that led to so many successes is not out of the box applicable to networks of such spiking neurons. Although modifications thereof have been developed and found to work in some scenarios [Lillicrap et al., 2016], it is general belief that the human brain is not solely based on error backpropagation and includes many other mechanisms to guide learning. In particular, biological plausible learning models emphasize on local learning rules, operating and influencing components mainly in immediate vicinity of neurons, with some global signals such as reward, attention or similar. Neurophysiological experiments revealed some key components of the dynamics happening in the brain. Among other many discoveries, one has to mention **Spike-timing dependent plasticity (STDP)**, a dynamic model of the weight shift occuring in neuron-to-neuron connections, **synapses**. The Hebbian theory already predicted such weight shifts previously [Hebb, 1949]. STDP can be regarded as a key principle in neural circuits and is being used to explain learning in many different scenarios.

1.3 New Computing Paradigms

Digital integrated circuits are one foundation of the modern world and are by now ubiquitious. Chip manufacturers find themselves in a heated competition to fit increasingly complex circuitry onto the same area of silicon. The number of transistors on a chip, making up the basic switches in this digital world, approximately doubled every two years since beginning. This observed phenomenon of exponential increase in transistor count is famously known as **Moore's law** [E. Moore, 2006]. Yet this statement hold true for a long time, it is unclear how long it remains valid. At a certain point, the structure sizes on the chip approach the scale of single atoms, effectively putting a limit to miniaturization of transistors as we know it. Even more worrisome are some problems that come along with smaller and smaller switches. For instance, reliability or heat dissipation. One proposed solution that seemingly obviates those problems is given by **Neuromorphic Computing**, see 1.3.2. Inspired by the brain, networks of connected neurons

implemented in hardware, perform a highly parallel an asynchronous computation. Incarnations of this concept typically use a symbiosis of **analog and digital** components, combining the best of both worlds.

1.3.1 Networks of Spiking Neurons

Spiking neurons are the key component of computation for every intelligent, concious biological organism. Critically for most living organisms is the ability to learn. This is particularly true for humans. Even the memory capacity within the DNA, storage for hereditary information encoded in genetic sequences, is not large enough to pinpoint the whole network of the spiking neurons found in a human brain. Thus, basic abilities, for instance communication by language, have to be aquired by the means of learning. This process of knowledge acquisition happens continuously and is crucial for individuals to adapt to new situations.

In general, the concept of spiking neurons is characterized by their main communication method: **Spikes**, which happen to be sudden events, large voltage peaks that transmit information to connected neurons. The connections between neurons are called **synapses**, in which such voltage peaks get converted to current pulses, induced by the release of neurotransmitters. Subsequent neurons accumulate spikes, causing an increase in their local cell membrane potential. If a certain threshold thereof is surpassed, a spike (action potential) is evoked. Many mathematical models were developed to model the complicated dynamics of the neurons membrane potentials. One of the biologically most accurate models is the Hodgkin-Huxley model [Hodgkin and Huxley, 1952]. Despite its accurateness, often simpler models are used for the reason of being analytically more accessible. A simplyfied model would be given by the Leaky integrate-and-fire model (LIF), which also suffices to describe the essential features of spiking neurons. Consider that each of the neurons is equipped with a quantity v_{mem} being the membrane potential. Then the dynamics of v_{mem}, in the LIF model, are governed by a differential equation, following [Gerstner, 2008]:

$$\frac{\mathrm{d}v_{\mathrm{mem}}(t)}{\mathrm{d}t} = -\frac{v_{\mathrm{mem}}(t) - v_{\mathrm{leak}}}{\tau} + \frac{I(t)}{C_{\mathrm{mem}}} \tag{1.1}$$

with t the time, v_{leak} the resting membrane potential, τ the membrane time constant, C_{mem} the membrane capacity and $I(t)$ summarizes all input current. If it happens that v_{mem} crosses a certain threshold θ from below, a spike is emitted and the membrane potential v_{mem} is resetted to the value of v_{reset}. By integration, and matching the initial conditions, another form is obtained:

$$v_{\mathrm{mem}}(t) = (v_{\mathrm{reset}} - v_{\mathrm{leak}})\mathrm{e}^{-(t-\hat{t})/\tau} + \int_0^\infty \Theta(t - \hat{t} - s)\mathrm{e}^{-s/\tau}\frac{I(t-s)}{C_{\mathrm{mem}}}\, ds \tag{1.2}$$

$\hat{t}$ denotes the time of the last spike and v_{reset} denotes the membrane potential after a spike was evoked. $\Theta(t) = \begin{cases} 1 & t \geq 0 \\ 0 & \text{else} \end{cases}$ is the Heaviside function. The input current may further be decomposed into a part that originates from synaptic input and some external input current. To model synaptic input in terms of the LIF model, there exist many alternatives. However, one commonly uses an exponentially decaying current after occurance of spikes, weighted by the respective synaptic efficacy:

$$I(t) = \sum_j \sum_{t'_j} w_j \mathrm{e}^{-(t-t'_j)/\tau_{\mathrm{syn}}}\Theta(t - t'_j) + I_{\mathrm{ext}}(t) \tag{1.3}$$

with j summing over all presynaptic neurons, t'_j summing over all spike times of presynaptic neuron j, w_j the synaptic efficacy of presynaptic neuron j to the considered one (not indexed here), τ_{syn} the synaptic time constant and I_{ext} an external current.

The equations are reminiscent of the ones describing a capacitor. In fact, in the LIF model, the membrane is regarded as a simple capacitor which is charged by incoming (synaptic) current. Thus, the LIF model is convenient for concrete hardware implementations, as the neuron itself can be instantiated by a capacitor.

1.3.2 Neuromorphic Hardware

Neuromorphic engineering is the concept of implementing hardware that tries to resemble and mimick key components of the brain in analog hardware. Originally, neuromorphic computing was coined by scientist and engineer Carver Mead [Mead, 1990]. Since then, an increasing amount of work was devoted to embrace this novel concept, with resulting devices being referred to as neuromorphic hardware.

The reasons to be interested in the massive endeavour of hardware implementations of neurological models are numerous, and a few were already mentioned in the beginning. A survey of neuromorphic systems, conducted by [Schuman et al., 2017], not only lists the main motivations for developing such systems but also takes the reader on a journey through the history in this field, models, algorithms and concrete hardware implementations. One of the most important points that drive publications in this field, is founded by the low power consumption of neuromorphic devices as opposed to classical computersimulations of equivalent neuronal network models. Also, if one attempts to simulate a large portion of spiking neurons, organised in a network, one quickly gets to the verge of what is doable in an economic way. Conventional hardware faces problems in power consumption and in speed of simulation (required time). Therefore, to conduct experiments on a large scale, such as in the Human Brain Project [Markram et al., 2011], neuromorphic hardware is, and continues to become an essential cornerstone [Schemmel et al., 2010].

1.4 Multi-armed Bandit Problems

Multi-armed bandits (MABs) are a classic example in the realm of reinforcement learning, operations research and computer science. A multi-armed bandit consists of several one-armed bandits, more commonly referred to as slot machines. The casino player, which will be denoted as the agent, acting upon those slot machines, is posed with the problem to figure out which slot machine to play or which arm to pull in order to maximize the cash outcome, his reward. To do so the agent has to conduct experiments by deciding on which arm to pull and observing the reward he is given. Using those observations and an appropriate time horizon in which he intends to play, the decisions of which action to commit to (which arm to pull) will trade off between exploration and exploitation, the key dilemma in reinforcement learning.

To formalize the notation used throughout this thesis, some key terms are introduced. Let the multi-armed bandit consist of N arms to pull, then the set of possible actions, which refer to an arm pull, will be denoted by $\mathcal{A} = (a_i)_{i \in \mathcal{I}}$, with $\mathcal{I} = \{1, \ldots, N\}$ representing an index set enumerating the arms. At a designated time step n, the agent commits to take the action denoted by $a^{(n)}$ and observes a stochastic reward $r^{(n)}$. Therefore, the agent generates a sequence of taken actions up to time step n: $A^{(n)} = (a^{(1)}, \ldots, a^{(n)})$ with the corresponding observed rewards $R^{(n)} = (r^{(1)}, \ldots, r^{(n)})$. Furthermore, let $\tilde{p}_i^{(n)} = \frac{1}{N_i} \sum_{j=1}^{n} r^{(j)} \cdot \mathbb{1}_{a^{(j)}=a_i}$ be the sample average

of observed rewards from arm i, N_i is the number of times the respective arm was pulled and $\mathbb{1}_{a^{(j)}=a_i}$ is the indicator function that the arm pulled at timestep j was the one with index i.

This thesis will particularly target Bernoulli bandits as the underlying problem to solve. This type of multi-armed bandit is characterized by binary rewards, either win or loss, with a certain probability $p_i(n) = P(r^{(n)} = 1|a^{(n)} = a_i)$. In addition, in this thesis, this probability of reward will be independent of the time step n such that $p_i(n) = p_i$.

1.4.1 Dependent Multi-armed Bandits

Another interesting type of bandit problem is given, when there exists an underlying structure between the reward probabilities in the multi-armed bandit, i.e. they are dependent. For instance, consider a two-armed bandit in which the reward probabilities sum to one. Thus, if one bandit arm has a probability of reward of p, then the others is given by $1 - p$. Such a dependent structure was already considered in [Wang et al., 2016]. If this dependence is known beforehand, more efficient policies for achieving maximum performance can be constructed. This particular dependency between two bandit arms will be considered as part of learning to learn later.

1.4.2 Policy Performance Measurements

A stochastic policy $pi(a|s)$ specifies the probability of taking action a if all observations regarding the multi-armed bandit (rewards) are summarized in s.

In order to be able to determine how *good* a certain multi-armed bandit policy really is, one needs to define a performance measurement. In the case of multi-armed bandit tasks, an established performance measurement is given by the expected cumulative regret, which gives the expected loss in reward by taking suboptimal actions. The expected cumulative regret of a multi-armed bandit policy π, with regard to a certain instance B of a multi-armed bandit, is defined by:

$$\text{ExpCumReg}(\pi, B) = \sum_{n=1}^{N_P} \mathbb{E}_{a^{(n)} \sim \pi(\,\cdot\,|r^{(n-1)},a^{(n-1)},\ldots,r^{(1)},a^{(1)}),r^{(n)} \sim B} \left[\max_m P\left(r^{(n)} = 1 \middle| a_m\right) - P\left(r^{(n)} = 1 \middle| a^{(n)}\right) \right] \tag{1.4}$$

where the actions are sampled according to the policy π, the rewards are sampled according to the reward probabilities given in B and N_P denotes the number of bandit arm pulls, the agent has to conduct, the **task length**.

Thus, the lower the regret, the better the policy is. In later scenarios the negative of the expected cumulative regret will be regarded as a **fitness**, desired to be maximized. The expected cumulative regret is for a certain reason a suitable performance measure. Consider for example independent bandits. Not every single instance of such a bandit task exhibits the same chance for expected cumulative reward, when the maximum reward probabilities are different. Thus, in order compare the performances across a whole family of bandit tasks, the cumulative regret is viewed to be a more appropriate choice as a performance measure than the cumulative reward.

1.5 MAB Policies in Literature

Multi-armed bandits are an often observed problem in general and not only computer scientists but also mathematicians dedicated a lot of effort to investigate optimal solutions. Indeed, the simple structure of such a problem exhibits an excellent surface for theoretical considerations. In this section common heuristic algorithms, which proved good performance on this type of problem, are studied. In addition, the optimal strategy in terms of expected cumulative regret, i.e. the solution for multi-armed bandit problems, Gittins indices are presented.

The fundamental challenge critical for the performance of algorithms in the reinforcement learning domain in general is efficient exploration. Only having available action choices sufficiently explored, the acting agent can switch to exploiting the best actions. Multi-armed bandits are well-suited to discover and determine proper strategies for trading off exploration and exploitation.

1.5.1 ϵ-Greedy

A common choice is the ϵ-Greedy strategy. The key advantage responsible for its popularity is simplicity as well as its applicability to all different sorts of problems. Yet simple, in many cases also sufficient. The outline of an ϵ-Greedy strategy is as follows: With probability $1-\epsilon$ take the action that is believed to be the best, and in the other case take a random action.

The notion of believed to be the best refers to be a maximum *a posteriori* value of taking a certain action. In the case of Bernoulli multi-armed bandits, the value of a specific action would simply be its associated reward probability. If there exists no *a priori* information, the maximum *a posteriori* reward probabilities at time step n of bandit-arm i, are given by the sample averages of the corresponding arm: $\tilde{p}_i^{(n)}$, as defined in Section 1.4. Thus, the next action $a^{(n+1)}$ is determined by:

$$i^{(n)} = \begin{cases} \arg\max_{i'} \tilde{p}_{i'}^{(n-1)} & \text{with probability } (1-\epsilon) \\ \text{randomchoice}(\{1, ..., N\}) & \text{else} \end{cases} \tag{1.5}$$

$$a^{(n)} = a_{i^{(n)}} \tag{1.6}$$

1.5.2 Bayes Upper Confidence Bound

For completeness, and by the importance in history, a very prominent strategy named `UCB1` [Auer et al., 2002] is also presented as part of this introduction. The strategy is based on calculating priority values of each bandit arm and play at each timestep n the one that maximizes this priority value. In addition, it is also proven that the policy given by `UCB1` obeys an upper regret bound [Auer et al., 2002]. Recall the reward sample average until timestep n: $\tilde{p}_i^{(n)}$. Then, the priority value according to `UCB1` to play arm i at timestep n, denoted by $\texttt{UCB1}_i^{(n)}$, is given by:

$$\texttt{UCB1}_i^{(n)} = \tilde{p}_i^{(n-1)} + \sqrt{\frac{2\log(n)}{N_i}} \tag{1.7}$$

with N_i counting the number of times arm i has been pulled. In the case of $N_i = 0$ it is set to one instead (Division). And therefore, the arm pulled according to the UCB1 policy is given by:

$$i^{(n)} = \arg\max_{i'} \texttt{UCB1}_{i'}^{(n)} \tag{1.8}$$

$$a^{(n)} = a_{i^{(n)}} \tag{1.9}$$

1.5.3 Gittins Indices

Perhaps one of the most celebrated results in theoretical computer science is the discovery of an optimal index policy regarding multi-armed bandit problems by [Gittins and Gittins, 1979]. In particular, index policy refers to the fact that for each bandit arm, present in the multi-armed bandit, one can compute an index value, representing the priority to play the corresponding arm. The resulting optimal policy, based on such indices, thus chooses to play at each timestep the bandit arm with the highest priority (index value). In fact, it is this very property that such a priority value can be computed for each arm seperately that enables an efficient solution in terms of maximum expected cumulative reward:

$$V_\pi = \mathbb{E}_\pi \left[\sum_{n=1}^{\infty} \gamma^n r^{(n)} \right] \tag{1.10}$$

where $r^{(n)}$ is the reward obtained at timestep n, by choosing actions (arm pulls) according to the policy π. $\gamma \in [0, 1)$ represents a discounting factor.

Consider a single arm of the multi-armed bandit with index i. At every timestep n, a certain amount of observations regarding arm pulls and obtained rewards are available for respective bandit arm, which are summarized in a state variable $s_i^{(n)}$. Then, the priority to play this arm is characterized by a function called Gittins index $G(s_i^{(n)})$. Hence, to execute the policy proposed by Gittins, one is required to compute such a value for every arm $i \in \mathcal{I}$ and choose the bandit arm with the maximum index value. So, the computational demands to compute such a policy scale linearly with N, the number of bandit arms. If instead, the priority to play a specific arm cannot be expressed by its connected state alone, but would require also the states of other bandit arms in the problem, computational demands would scale by $N \cdot |S|^N$, as such a priority value would have a bigger domain.

To express the Gittins index $G(s_i^{(n)})$, following [Mansour and M. Schain, 2011] (economic interpretation), one considers a single bandit arm in state $s_i^{(n)}$. Suppose that at each time the bandit arm i is played, a fixed charge λ has to be paid, basically a negative reward. Then, one can formulate an expression for a "fair" charge λ^* that results in zero expected cumulative reward, if the bandit arm is pulled for a maximal number of times. Thus (taken from [Mansour and M. Schain, 2011]):

$$\lambda^*(s_i^{(n)}) = \sup \left\{ \lambda \middle| \sup_{\tau>0} \mathbb{E} \left[\sum_{n'=n}^{n+\tau-1} \gamma^{n'-n} \left[r(s_i^{(n')}) - \lambda \right] \middle| s_i^{(n)} \right] \right\} \tag{1.11}$$

with $r(s_i^{(n')})$ representing the random reward of the bandit arm with the state $s_i^{(n')}$.

Then, the Gittins index is defined by this fair charge $G(s_i^{(n)}) = \lambda^*(s_i^{(n)})$. Also presented is

the more common, but equivalent, formulation:

$$G(s_i^{(n)}) = \lambda^*(s_i^{(n)}) = \sup_{\tau>0} \frac{\mathbb{E}\left[\sum_{n'=n}^{n+\tau-1} \gamma^{n'-n} r(s^{(n')}) \big| s_i^{(n)}\right]}{\mathbb{E}\left[\sum_{n'=n}^{n+\tau-1} \gamma^{n'-n} \big| s_i^{(n)}\right]} \tag{1.12}$$

Various proofs, regarding the theorem that the Gittins index policy is optimal, were developed since the publication. The interesting reader hence may be referenced to [Frostig and Weiss, 1999], where four different proofs are elucidated. A sketch of a proof in [Mansour and M. Schain, 2011] 7.5 is explained in the following. Reconsider Equation 1.11, representing both the fair charge and the Gittins index. Now assume that the paid charges for each bandit arm are set to the fair charges and thus, no reward can be made from pulling any of the bandit arms. However, set aside the charges that are paid, the rewards obtained (in expectation) by pulling bandit arms are maximized if the one with highest fair charge, or Gittins index, is played.

The Gittins index policy is thus given by:

$$i^{(n)} = \arg\max_{i'} G(s_i^{(n)}) \tag{1.13}$$

$$a^{(n)} = a_{i^{(n)}} \tag{1.14}$$

1.5.4 Optimistic Gittins Indices

Another interesting development in the area of multi-armed bandits is given in [Gutin and Farias, 2016], where an optimistic variant of Gittins indices is developed. The key principle is that, as opposed to ordinary Gittins Indices, as in Section 1.5.3, the optimistic variant only looks ahead one time step. Although this policy will not be used in the later analysis on the actual objective, it is to be pointed out that an important advantage of this approach is the speed of computation compared to Gittins indices.

1.6 Demixed Principal Component Analysis - dPCA

As part of analyzing results obtained in later Sections, the recent method of demixed principal component analysis, as developed in [Brendel et al., 2011], is used. Plain principal component analysis (PCA) seeks to find orthogonal projection vectors that maximize variance captured along those directions. However, PCA does not try to explain which additional labels or parameters caused the variance. Demixed principal component analysis (dPCA) on the other hand tries to maximize the variance captured along projections, while at the same time, finding directions that depend on the smallest number of parameters or labels.

In the later analysis of the results obtained in this thesis, dPCA will be used. In particular, the interesting label or parameter will be given by a performance measurement that is demixed from other variations in parameter space, summarized as noise.

2 Learning to Learn

The goal of learning to learn is, in general, to enhance a learning algorithms performance to learn. In the same way as humans will get better at driving vehicles as experience at similar tasks is collected, the concept of learning to learn tries to improve learning algorithms as tasks of a similar kind are encountered. Hence, one considers a probability distribution over an entire task family $\mathcal{T}$, on which learning performance is to be maximized. More specifically, one defines a **fitness** function $\mathcal{F}(T, \boldsymbol{\theta})$ that expresses how good a model, whose learning algorithm depends on hyperparameters $\boldsymbol{\theta}$, can learn a task T. So, one can formulate the best set of hyperparameters, performing well on an entire distribution of tasks $\mathcal{T}$, by:

$$\boldsymbol{\theta} = \arg\max_{\boldsymbol{\theta}'} \left\{ \mathbb{E}_{T \sim p_{\mathcal{T}}(\,\cdot\,)} \left[\mathcal{F}(T, \boldsymbol{\theta}') \right] \right\} \tag{2.1}$$

where $p_{\mathcal{T}}(T)$ is the probability density of task T in family $\mathcal{T}$. The overall setup of this concept is depicted in Figure 2.1. An optimizer suggests to evaluate the fitness of new hyperparameters based on the fitness of previously evaluated hyperparameter settings.

The origins of learning to learn, also known as meta-learning, dates back to [Schmidhuber, 1987]. Later on, in [Hochreiter et al., 2001] supervised meta-learning was performed, using LSTMs on classification and regression tasks. As the idea grew, also reinforcement learning tasks were considered, capable of learning to navigate in mazes of a similar type efficiently [Wang et al., 2016] and [Duan et al., 2016]. Additionally, also optimization of optimization algorithms was considered by [Andrychowicz et al., 2016].

As this concept will also be applied as a core part of this thesis, an appropriate software framework, carrying out the necessary optimization algorithms, will be used. The **Learning to Learn** Framework developed locally at the **Institute of Theoretical Computer Science - Graz University of Technology** matches the proposed needs. It is implemented in a modular way, allowing for a fast integration of learning models. Also, a number of stochastic optimization algorithms are included, the ones being used in this thesis are presented in Section 2.1. It is implemented with `Python 3` as the programming language and follows the principles as given in Figure 2.1.

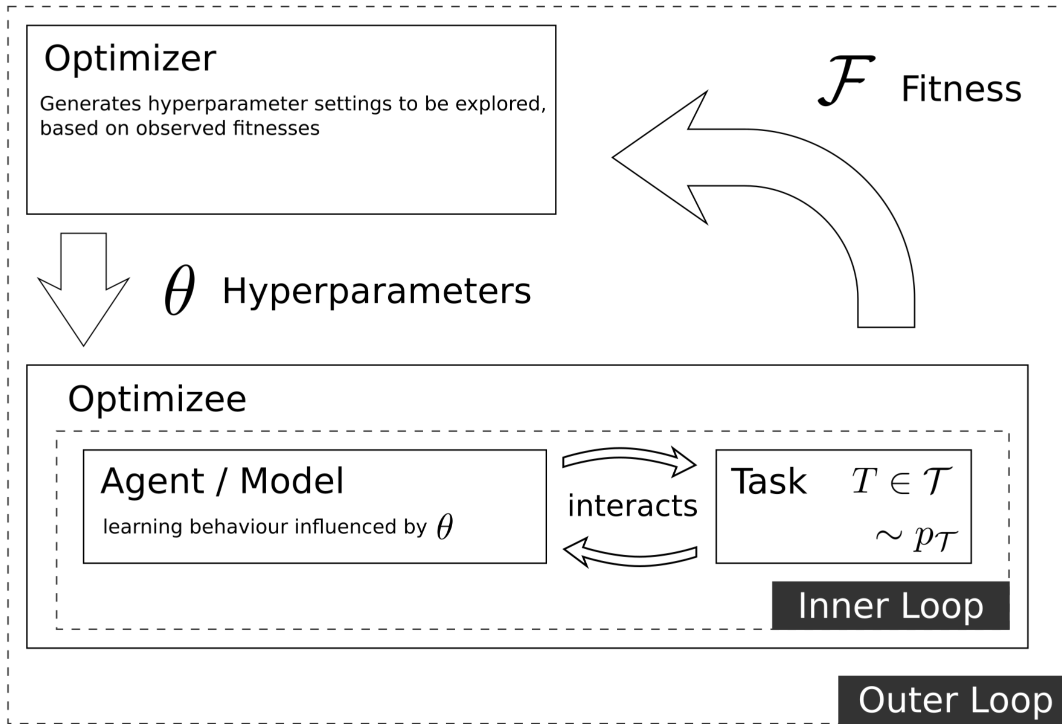

Figure 2.1: Outline of learning to learn optimization. Instances of hyperparameter settings $\boldsymbol{\theta}$ *are evaluated in the optimizee. Evaluation is performed with regard to some performance measure, for instance cumulative reward, on sampled task instances. The result of which is fed back to the optimizer that again tries to come up with new, better hyperparameters.* ***Inner Loop:*** *Learning of a concrete task instance.* ***Outer Loop:*** *Hyperparameter optimization on the entire task family.*

2.1 Optimizers

As optimization algorithms are required to carry out this kind of hyperparameter optimization, this section will introduce the algorithms that are used in Chapter 7. Fitness values, coming along with certain hyperparameter settings, have to be considered stochastic because both learning algorithm and/or task can introduce randomness. Therefore, an optimizer has to be chosen that can handle the amount of noise present in fitness evaluation. Population-based, stochastic optimization algorithms are of advantage because evaluating whole populations can average-out such effects. The negative side of population-based techniques on the other hand is speed. The more evaluations required, the more time will pass.

Also, each optimizer is again characterized by certain values, for instance learning rates or similar. Those values will be denoted as **hyperhyperparameters** and remain to be chosen manually. The use of unreasonable hyperhyperparameters can result in the failure of the optimization procedure.

Optimization algorithms that were used throughout this thesis are presented in the remainder of this Section.

2.1.1 Cross Entropy Method

The cross entropy method [Rubinstein and Kroese, 2004] uses at its core a parametric probability distribution over well performing points, in the present case these are hyperparameter settings. In one iteration, a certain amount of samples are taken from this distribution, yielding a population of points. After evaluation of the fitnesses of those points, the parameters of respective probability distribution are modified towards well-performing points. Specifically a maximum likelihood fit of a certain number of best performing points, the elite, is performed. A common choice for such a probability distribution is a Gaussian, which is also used here. In addition, noise can be introduced to avoid premature convergence. A problem which is characterized by a sharp sampling distribution in a suboptimal region. Additionally, one may further add a smooth parameter update of the underlying probability distributions parameters over the iterations. The procedure is described in Algorithm 1

Algorithm 1 Cross entropy method

Require: $p(\,\cdot\,;\boldsymbol{\psi})$: Initial population distribution, parametrized by $\boldsymbol{\psi}$
Require: n: Population size
Require: ρ: Fraction of population considered as elite
Require: s: Parameter smoothing factor
  while stopping criterion not met **and** maximum iterations not reached **do**
    for $i \in \{1, 2, ..., n\}$ **do**
      sample $\boldsymbol{\theta}_i \sim p(\,\cdot\,;\boldsymbol{\psi})$
      $F_i = \mathcal{F}(\boldsymbol{\theta}_i)$
    end for
    pick the $\lfloor \rho \cdot n \rfloor$ best performing samples, giving the set M (elite)
    $\psi' = \arg\max_{\boldsymbol{\psi}} \sum_{\boldsymbol{\theta} \in M} \log(p(\boldsymbol{\theta};\boldsymbol{\psi}))$ (maximum likelihood)
    (optional) widen distribution $p(\,\cdot\,;\boldsymbol{\psi}')$ to prevent early convergence
    $\psi \leftarrow s\psi + (1-s)\psi'$ (smoothing)
  end while
  return $\theta \sim p(\,\cdot\,;\boldsymbol{\psi})$

The cross entropy method thus requires the applyer of LTL to chose the hyperhyperparameters of

- $p(\,\cdot\,;\psi)$, the particular population distribution, which is a parametric distribution and parametrized by ψ,
- n, the population size,
- ρ, the fraction of the population which is considered as elite at each iteration and
- s, the smoothing factor, which describes how much of the old distribution parametrization goes into the new one.

2.1.2 Parallel Simulated Annealing

Simulated annealing describes an optimization procedure conceptually similar to heating and cooling a metal in such a way that the number of defects in the resulting crystal becomes less in order to reduce energy.

As a standard stochastic optimization procedure, its main purpose here is to compare to other methods. Essentially serving as a baseline. In simulated annealing a point in hyperparameterspace is deviated and its fitness is evaluated. If the fitness outcome is better than the previous, the deviation is kept. Otherwise, depending on a temperature and the fitness difference, the deviation may be discarded. In the context of simulated annealing, the probability of keeping a worse performing hyperparameter vector scales as a boltzmann factor, given by $e^{x/T}$, when x is the fitness difference and T the temperature, which changed over iterations according to a schedule. In the parallel version of simulated annealing, there are simply many simulated annealing procedures operating concurrently and independent of each other. In Algorithm 2 simulated annealing, parallelized as used throughout this thesis, is explained in terms of pseudocode. The temperature schedule used, can be as simple as a multiplicative decay: $T^{(n)} = T_0 \cdot c^n$, with $c \in (0, 1)$ a decay factor, n the current iteration number and T_0 the initial temperature.

Algorithm 2 Parallel Simulated Annealing as it is used in this work

Require: $\theta_i, i \in \{1, 2, ..., n\}$: Hyperparameter starting points
Require: σ: Exploration step size standard deviation
Require: n: Population size
Require: $T^{(i)}, i \in \{1, 2, ...\}$: Temperature schedule
  $F_i \leftarrow \mathcal{F}(\theta_i), \quad \forall i \in \{1, 2, ..., n\}$
  counter $\leftarrow 0$
  while stopping criterion not met **and** maximum iterations not reached **do**
    for $i \in \{1, 2, ..., n\}$ **do**
      $\boldsymbol{\epsilon}_i \sim \mathcal{N}(\mathbf{0}, \sigma \cdot \mathbb{1})$
      $F'_i = \mathcal{F}(\boldsymbol{\theta}_i + \boldsymbol{\epsilon}_i)$
      if $F'_i \geq F_i$ **then** (keep solution)
        $\boldsymbol{\theta}_i \leftarrow \boldsymbol{\theta}_i + \boldsymbol{\epsilon}_i$
        $F_i \leftarrow F'_i$
      else (keep solution with a certain probability)
        $p = \exp\left((F_i - F'_i)/T^{(\text{counter})}\right)$
        if Uniform$(0, 1) < p$ **then**
          $\boldsymbol{\theta}_i \leftarrow \boldsymbol{\theta}_i + \boldsymbol{\epsilon}_i$
          $F_i \leftarrow F'_i$
        end if
      end if
    end for
    counter $\leftarrow$ counter $+ 1$
  end while
  $i = \arg\max_{i'} F_{i'}$
  return $\boldsymbol{\theta}_i$

The parallelized version of simulated annealing requires to chose the hyperhyperparameters population size n, σ, which is the standard deviation of the exploration steps and the temperature schedule $T^{(i)}$.

2.1.3 Finite Difference Gradient Method

Gradient-based methods work well in many cases, including deep learning. Since the objective considered here is not to be assumed differentiable, such a method cannot be implemented. However, an approximated gradient can be found by exploring the fitnesses around a basepoint. This method is commonly known as taking the finite difference gradient. More precisely, the strategy is collect samples with a random distance and direction from the basepoint and then solve a overdetermined equation system in terms of least squares, yielding an approximated gradient. A problem that occurs in practice is that the objective is noisy and thus, the gradient is too. The detailed description of the optimizer used is to be found in Algorithm 3.

Algorithm 3 Finite Difference Gradient Ascent

Require: $\boldsymbol{\theta} \in \mathbb{R}^d$: Hyperparameter starting point
Require: σ: Exploration step size standard deviation for finite difference gradient
Require: n: Number of exploration steps
Require: α: Learning rate
  while stopping criterion not met **and** maximum iterations not reached **do**
    for $i \in \{1, 2, ..., n\}$ **do**
      $\boldsymbol{\epsilon}_i \sim \mathcal{N}(\mathbf{0}, \sigma \cdot \mathbb{1})$
      $F_i = \mathcal{F}(\boldsymbol{\theta} + \boldsymbol{\epsilon}_i)$
    end for
    $F' = \mathcal{F}(\boldsymbol{\theta})$
    $\boldsymbol{D} = (F_1 - F', F_2 - F', ..., F_n - F')^T$
    $\boldsymbol{M} = [\boldsymbol{\epsilon}_1 | \boldsymbol{\epsilon}_2 | ... | \boldsymbol{\epsilon}_n]^T \in \mathbb{R}^{n \times d}$
    $\boldsymbol{g} = \arg\min_{\boldsymbol{g}'} (\boldsymbol{M}\boldsymbol{g}' - \boldsymbol{D})^2$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{g}$
  end while
  return $\boldsymbol{\theta}$

This finite difference gradient method requires the hyperhyperparameters of

- n, the number of random exploration steps,
- σ, the standard deviation of the exploration step and
- α, the learning rate.

2.1.4 Evolution Strategies

Evolution Strategies propose a similar approach to optimization as established, well-known evolutionary methods. However, due to a more sophisticated underlying theory, evolution strategies lead to a more profound optimization. Originally proposed by [Rechenberg and Eigen, 1971], many forms of similar ideas were found to work well in optimization. The subclass given by natural evolution strategies [Wierstra et al., 2008], which is being used in this thesis, already proved to work well on reinforcement learning tasks [Salimans et al., 2017]. The outline of optimization given by the proposed method is sketched in Algorithm 4.

Algorithm 4 Evolution strategies, as used in this thesis.

Require: $\boldsymbol{\theta} \in \mathbb{R}^d$: Hyperparameter starting point
Require: n: Population size
Require: σ: Exploration standard deviation
Require: α: Learning rate
  while stopping criterion not met **and** maximum iterations not reached **do**
    for $i \in \{1, 2, ..., n\}$ **do**
      if mirror sampling enabled **and** $i \geq \lceil \frac{n}{2} \rceil$ **then**
        $\boldsymbol{\epsilon}_i = -\boldsymbol{\epsilon}_{n-i}$
      else
        $\boldsymbol{\epsilon}_i \sim \mathcal{N}(\mathbf{0}, \sigma \cdot \mathbb{1})$
      end if
      $F_i = \mathcal{F}(\boldsymbol{\theta} + \boldsymbol{\epsilon}_i)$
    end for
    if fitness shaping enabled **then**
      $k(i)$: Index of i-th greatest fitness
      $u_i = \frac{\max\left(0, \log\left(\frac{n}{2}+1\right) - \log(k(i))\right)}{\sum_{j=1}^{n} \max\left(0, \log\left(\frac{n}{2}+1\right) - \log(j)\right)} - \frac{1}{n}, \quad \forall i \in \{1, 2, ..., n\}$
    else
      $u_i = F_i$
    end if
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \frac{\alpha}{\sigma n} \sum_{i=1}^{n} u_i \boldsymbol{\epsilon}_i$
  end while
  return $\boldsymbol{\theta}$

Evolution strategies requires that the hyperhyperparameters of

- n, the number of random exploration steps,
- σ, the standard deviation of the exploration steps and
- α, the learning rate

are chosen.

3 Spiking Network Architecture

Particular types of spiking neurons exhibit different responses if stimulated. The behaviour of a network of spiking neurons will in turn also depend on the particular type of neurons by which the network is composed. The dynamic behaviour of spiking neurons are commonly characterized by underlying mathematical models. It follows that a neuron description with a deterministic model will lead to deterministic behaviour of the overall network itself and can be used to design a deterministic agent for multi-armed bandits.

Addressing such a task requires the agent to decide on a profitable bandit arm when the opportunity for executing an action, which is pulling such a bandit arm, is given. To signal that an action choice should be made to a spking neural network, a cue in form of stimulating spikes is given as input. On the other hand, the agent is designed that its spiking neural network communicates the decision on which arm to play by single spikes of different neurons. If one of those neurons, which will be referred by **action neurons** in the further, emits a spike, the corresponding bandit arm will be pulled.

Although it is now specified how the spiking neural network interfaces the multi-armed bandit, it is still open to resolve how the input cue is transformed to the particular action output. The specification of this transform is a design choice that is mostly concerned with network topology. The essential idea is to enhance signals coming from the stimulating cue towards the action neurons that represent rewarding bandit arms. And on the other hand, attenuate cue signals projecting to action neurons representing unprofitable bandit arms. Assuming identical properties among the action neuron population, it then follows that the action neuron, representing the most profitable arm, will emit a spike first. Furthermore, if a spike was elicited by an action neuron, it shall prevent all other action neurons from spiking. This mutual exclusion of actions requires to add inhibitory neurons to the circuit.

3.1 Network Design

To enable such attenuation or enhancement of signals the weights of synapses connecting the source of the cue, which will be denoted as stimulating cue neuron, and an action neuron are used.

The agent also has to adapt its action choices according to observed rewards from pulling arms of the multi-armed bandit. Since the next action is effectively determined by the synaptic weights in the network, it is required that those undergo some dynamic changes: Synaptic plasticity is therefore a necessary component to endow a network of spiking neurons with the ability to solve multi-armed bandit problems in this design. In particular, Section 3.2 is dedicated to introduce specific update rules of the synaptic weights that ultimately lead to a certain algorithm for multi-armed bandits.

Consider Figure 3.1. The ideas are summarized to the following conceptual network circuit: Each possible action $a_i \in \mathcal{A}$ will be represented by a certain action neuron, denoted by $\mathcal{C}^{(i)}$. If a

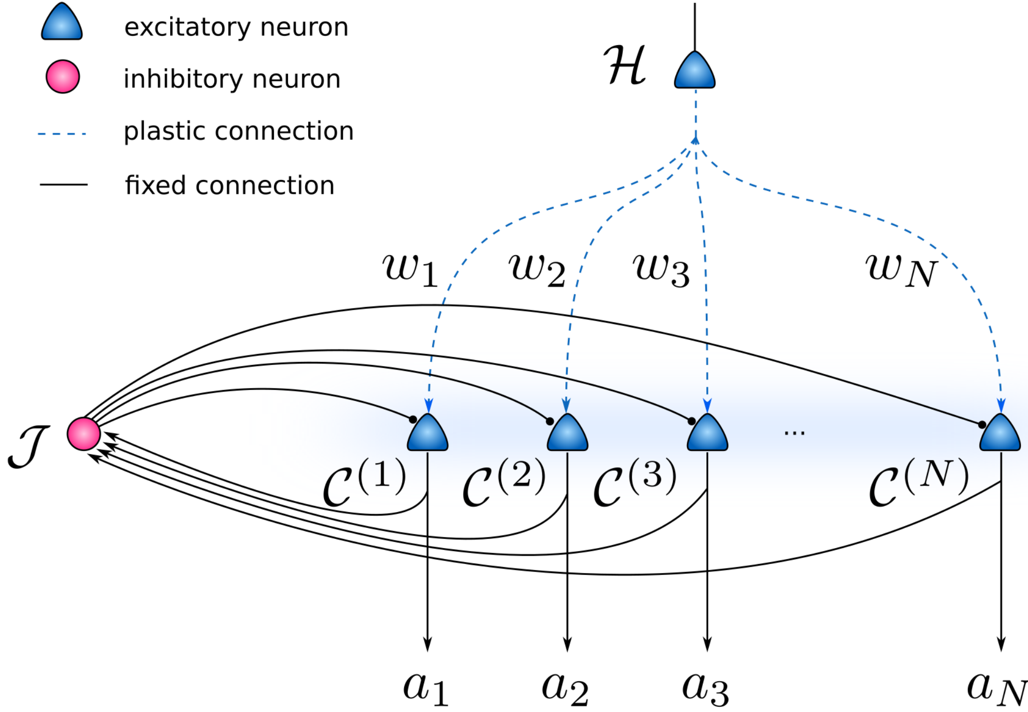

Figure 3.1: The perceived network motif consists of a stimulating cue neuron $\mathcal{H}$ that projects to the action neurons $\mathcal{C}^{(i)}$ with weights w_i. To achieve competition between action choices, an inhibition among the action neurons is introduced with the population $\mathcal{J}$.

spike is emitted from such a neuron, the corresponding action will be triggered. For stimulation, those neurons receive synaptic input by the means of a stimulating neuron $\mathcal{H}$, representing the cue that triggers action selection. As described, stimulating cue spikes are connected to action neurons by synapses. In particular, the weight of the synapse that connects $\mathcal{H}$ to $\mathcal{C}^{(i)}$ will be denoted by w_i. By the previous assumption that each action neuron has identical behaviour, the projection with having the greatest weight will cause the postsynaptic neuron to elicit a spike first.

In order to have a single action neuron emitting a spike, an inhibitory population $\mathcal{J}$ is introduced to enable a competition among the actions. If synaptic delay is nonzero, multiple action neurons can still emit a spike if they spike shortly after each other until inhibition takes place. In such a case, a random action from the respective set of spiking action neurons is taken. The resulting delay, until inhibition can become effective among action neurons, will be denoted by τ_S. An exemplary process of action selection from the viewpoint of spikes in the network is shown in Figure 3.2.

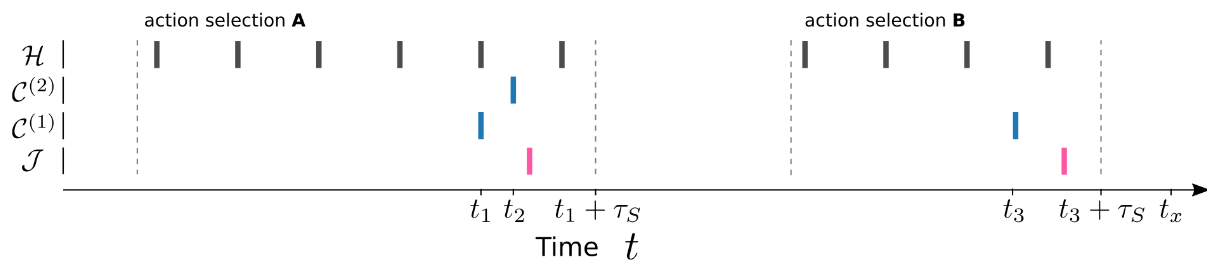

*Figure 3.2: Network activity in terms of spikes during two action selection processes **A** and **B**. Both action selections are triggered by spikes from the cue neuron $\mathcal{H}$.*
***A**: Cue neuron $\mathcal{H}$ elicits action neuron $\mathcal{C}^{(1)}$ to emit a spike at time t_1. Likewise, $\mathcal{C}^{(2)}$ fires a spike later at t_2. The inhibition ($\mathcal{J}$) that follows the spike of $\mathcal{C}^{(1)}$ did not affect $\mathcal{C}^{(2)}$ because it gets effective only after some delay τ_S. Spikes of other neurons are thus possible before $t_1 + \tau_S$ ($t_2 < t_1 + \tau_S$).*
***B**: A modified weight situation causes a different scenario. Action neuron $\mathcal{C}^{(1)}$ emits a spike at time t_3. By continous stimulation $\mathcal{C}^{(2)}$ **would** spike at t_x. This is prevented by inhibition ($\mathcal{J}$) caused by the spike of $\mathcal{C}^{(1)}$ ($t_x > t_3 + \tau_S$).*

3.2 Update Rules

So far, the designed network supports action selection based on weights associated to each bandit arm. This section discusses necessary schedules to modify the weights within the network according to observations of the environment. Mechanisms of such a kind will simply be referred to as **update rules**.

Since the weights within the network of spiking neurons will eventually be responsible which arm of the bandit will be pulled, the weights need to change based on observed rewards, in order to reinforce viable actions. The extent and the particular schedule of weight changes, i.e. the update rule, will ultimately be accountable for the performance on such a bandit task. Thus, in order for the conceived spiking-agent to handle multi-armed bandits well, sophisticated update rules need to be considered. However, the computations performed per synapse will be expensive in later considerations, so that we restrict the weight update rules to be computational simple ones, for the sake of concrete implementations (Chapter 5).

Possible update rules, suitable for Bernoulli bandits, will be explored throughout the remainder of this section. In principle, arbitray known multi-armed bandit algorithms can be modeled by a suitable weight update rule.

To aid subsequent implementations of such update rules and to be able to compare the different dynamics, it was decided to consider **normalized weights** in the interval $[0, 1]$. This is justified by the assumption that only relative differences in the weights are relevant for the first action spike, leading to a specific action choice. In a concrete implementation scenario, one would map a weight value of 1 to a suitable, realizable high weight and 0 respectively to an appropriate low weight.

3.2.1 Greedy Update

As the counterpart to the ϵ-Greedy algorithm presented in Section 1.5.1, a Greedy update rule for the synaptic weights in the neural network is introduced.

The underlying idea is to take the action that is, by the observations available until the considered time step, believed to be the most rewarding one. In terms of Bayesian probability, the

best believed action is the one having the maximum *a posteriori* reward probability among all actions. Since the focus in this case are independent multi-armed bandits with fixed reward probabilities, single rewards for each arm are Bernoulli distributed. In order to make use of Bayesian inference, while keeping the computational demands small, it is of great advantage to use the **conjugate prior** to the Bernoulli distribution, being the Beta distribution. Equipped with these statistical tools, updating the *a posteriori* reward probability, as new reward observations are being made, reduces to a simple parameter update. In fact, the Beta distribution is fully characterized by 2 parameters, denoted by $\alpha^{(n)}$ and $\beta^{(n)}$ at a considered timestep n. By counting the number of rewards and plays of each arm i, the parameter update is obtained by:

$$\alpha_i^{(n+1)} = \alpha_i^{(n)} + \begin{cases} 1 & r^{(n)} = 1 \wedge a^{(n)} = a_i \\ 0 & \text{else} \end{cases} \tag{3.1}$$

$$\beta_i^{(n+1)} = \beta_i^{(n)} + \begin{cases} 1 & r^{(n)} = 0 \wedge a^{(n)} = a_i \\ 0 & \text{else} \end{cases} \tag{3.2}$$

$$w_i^{(n+1)} = \frac{\alpha_i^{(n+1)}}{\alpha_i^{(n+1)} + \beta_i^{(n+1)}} \tag{3.3}$$

Thus, the weights, corresponding to arm i, are set the mean reward that was observed by playing bandit arm i until the current time step. The initialisations of α and β are hyperparameters, reflecting prior knowledge of the underlying bandit tasks, which can be optimized by the means of learning to learn 2. If nothing is known about the reward probabilities of the bandit arms, the initialization to unform reward probability distribution is given by $\alpha_i^{(0)} = 1$ and $\beta_i^{(0)} = 1$.

Altough this update rule appears to be completely greedy, it is not if one considers a nonzero synaptic delay in the inhibition of the spiking neural network.

3.2.2 Incremental Update

Instead of having additional memory to keep track of every bandit arms reward statistics, a different approach is to use the current weight value at time step n as a proxy and to modify this value appropriately.

The presented update rule is, in fact, an exponential filter of the obtained rewards of the considered bandit arm:

$$w_i^{(n+1)} = \begin{cases} (1 - \epsilon_i^{(n)}) \cdot w_i^{(n)} + \epsilon_i^{(n)} r^{(n)} & a^{(n)} = a_i \\ w_i^{(n)} & \text{else} \end{cases} \tag{3.4}$$

$$\epsilon_i^{(n+1)} = \epsilon_i^{(n)} \cdot \begin{cases} d & a^{(n)} = a_i \\ 1 & \text{else} \end{cases} \tag{3.5}$$

with ϵ representing a learning rate that may decay by a factor of $d \leq 1$ per synapse, to allow larger updates in the beginning of a bandit task.

3.2.3 Artificial Neural Network Update

In order to be fully flexible at specifying an update rule, one can make use of a general function approximator such as a multilayer perceptron. The specific input to output mapping of this approximator is specified by a number of free parameters, which are to be chosen or optimized

in such a way that a good update rule is obtained. Since the multilayer perceptron is a type of an artificial neural network (ANN), this update rule will be referred to as **ANN update rule**. The design of such a multilayer perceptron is outlined in Figure 3.3. There exists an input

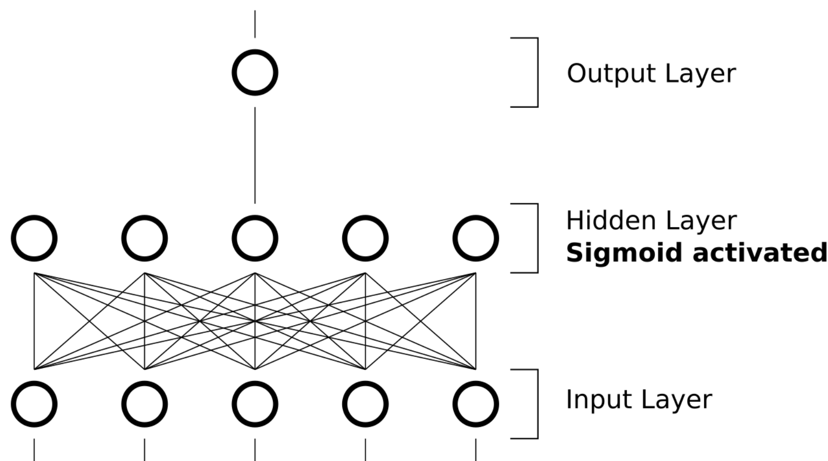

Figure 3.3: Multilayer perceptron as used as an update rule. The single hidden layer has an activation function given by the sigmoid $\sigma(x) = \frac{1}{1+e^{-x}}$. All lines between the artificial neurons, drawn as circles, represent weights that are free parameters and thus, subject to an optimization procedure.

layer, a single hidden layer that is equipped with the sigmoid $\sigma(x) = \frac{1}{1+e^{-x}}$ as an activation function and finally an output layer. The single neuron of the ANN output layer is used to modify the weights, $\mathcal{H} \to \mathcal{C}^{(i)}$, of the spiking neural network in an additive manner and each weight is updated by the ANN in every timestep. Specifically, the parametrization of the ANN is the same throughout all connections, i.e. the update function is independent of the particular connection. Hence, the inputs to the ANN, denoted by $\boldsymbol{E}_i^{(n)}$ for the connection $\mathcal{H} \to \mathcal{C}^{(i)}$, are composed of quantities that are necessary or helpful for determining the viability of a bandit arm. So, the update rule for the weight associated to bandit arm i at timestep n can be specified by:

$$w_i^{(n+1)} = w_i^{(n)} + f_{\text{ANN}}(\boldsymbol{E}_i^{(n)}; \boldsymbol{\theta}) \tag{3.6}$$

if the output of the ANN is denoted by f_{ANN} and the parametrization of the ANN is given by the vector $\boldsymbol{\theta}$.

As inputs $\boldsymbol{E}_i^{(n)}$ to the ANN, a set of variables is chosen that would in the further sense also be required by algorithms in Section 1.5. Thus, the following set of quantities is assumed to be vital for a well-performing update rule:

- Current timestep: n
- Reward at timestep n: $r^{(n)}$
- Flag (1 or 0) if assoctiated bandit arm was pulled: $\mathbb{1}_{a_i = a^{(n)}}$
- Current weight of the connection $\mathcal{H} \to \mathcal{C}^{(i)}$: $w_i^{(n)}$
- **Dependent bandits:** - Current weight of the other connection: $w_{3-i}^{(n)}$

A visual depiction of this setup is given in Figure 3.4. An additional observations is that the given setup implements a simple artificial recurrent architecture, with the weight $w_i^{(n)}$ representing the recurrent cell.

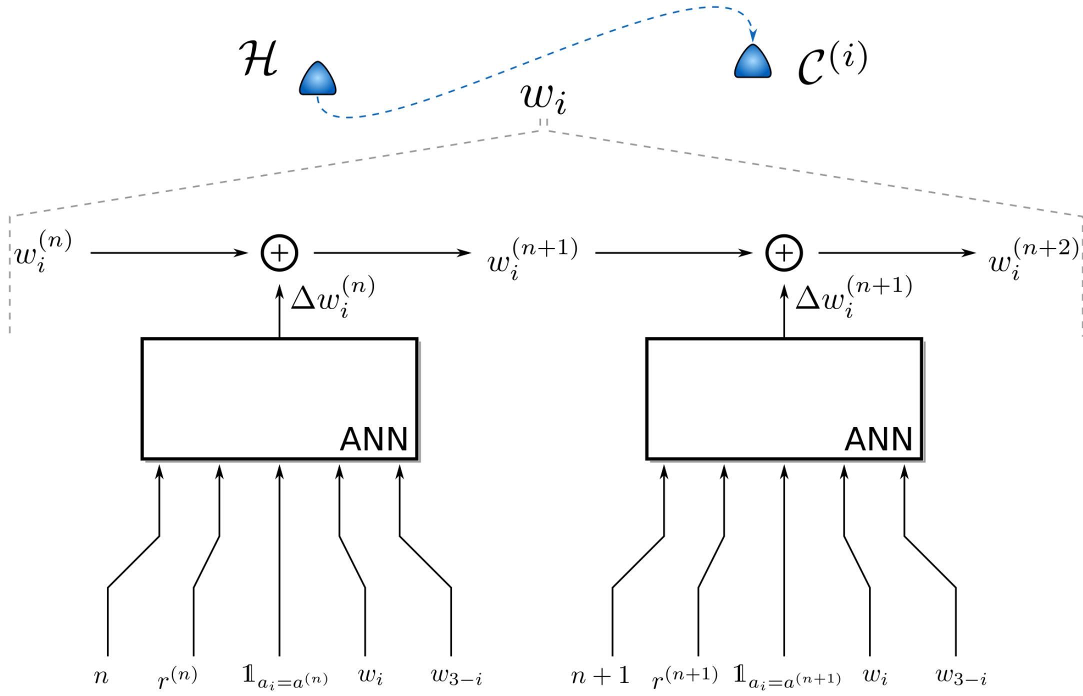

Figure 3.4: Proposed setup of using a parametrized artificial neural network to encode an update rule. At each timestep the ANN outputs additive changes for a weight in the spiking neural network. The setup is reminiscent of a recurrent artificial neural network architecture

Optimization/Simulation

To determine the viability of this approach, a software simulation is conducted. As the weights and biases of the ANN are free parameters, an optimization procedure is required to find an appropriate setting. The objective chosen to be minimized, in order to obtain the correct parametrization $\boldsymbol{\theta}$, is the cumulative regret at the end of a particular bandit task instance. The length of the bandit task, i.e. the number of arm pulls, was decided to be $N_P = 100$. In addition the size of the multi-armed bandit was set to two bandit arms. One can view the resulting multi-armed bandit policy, based on this artificial neural network weight update, to be parametrized by this parameter vector $\boldsymbol{\theta}$: $\pi_{\text{ANN}}(a_i|r^{(n-1)}, a^{(n-1)}, ..., r^{(1)}, a^{(1)}; \boldsymbol{\theta})$. Hence, to find the best parameter vector, one can formulate an optimization with regard to a whole family of bandit tasks $\mathcal{B}$, being dependent or independent multi-armed bandits:

$$\boldsymbol{\theta} = \arg\min_{\boldsymbol{\theta}'} \mathbb{E}_{B\sim\mathcal{B}} \left[\text{ExpCumReg}(\pi_{\text{ANN}}(\,\cdot\,|\boldsymbol{\theta}'), B)\right] \tag{3.7}$$

To solve the optimization problem, the learning to learn framework was utilized. An optimization with cross entropy revealed that the ANN is capable of performing comparable to existing algorithms, as in Section 1.5 in the case of **dependent bandits**. It is therefore the case that the update rule can transfer knowledge about the concrete bandit task family and thus, learning to learn is achieved. The performance over iterations during this run is depicted in Figure 3.5 for independent bandits and in Figure 3.6 for dependent bandits as described in Section 1.4.

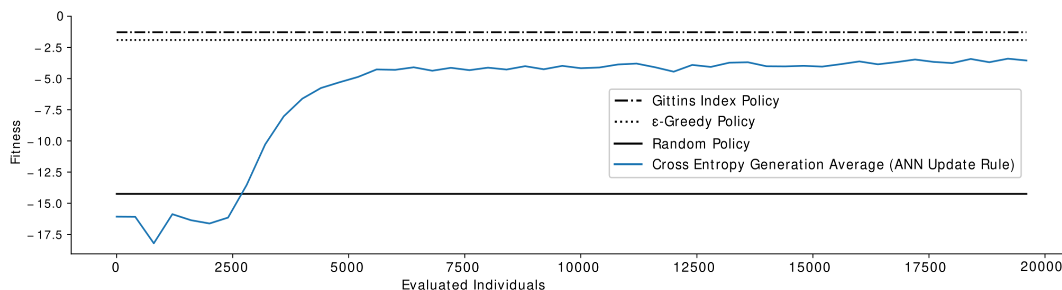

Figure 3.5: Test optimization with cross entropy and ***independent*** *bandits. The curve indicates the negative cumulative regret (fitness objective) obtained after 100 pulls depending on the number of evaluated individual hyperparameter settings, averaged over the population in the optimization algorithm. Horizontal lines represent baselines of well-known multi-armed bandit algorithms.*

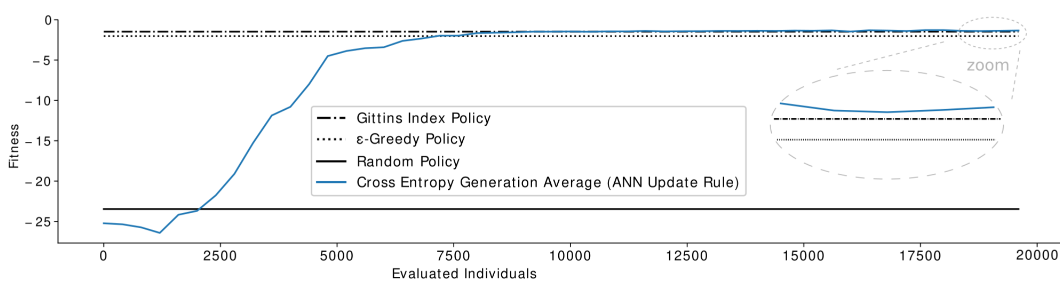

Figure 3.6: Test optimization with cross entropy and ***dependent*** *bandits. The curve indicates the negative cumulative regret (fitness objective) obtained after 100 pulls depending on the number of evaluated individual hyperparameter settings, averaged over the population in the optimization algorithm. Horizontal lines represent baselines of well-known multi-armed bandit algorithms.*

4 Digital Learning System - HICANN-DLS

The abbreviation HICANN-DLS stands for **High Input Count Analog Neural Network with Digital Learning System** and is an incarnation of neuromorphic hardware 1.3.2. It is being developed as part of the **Human Brain Project** [Markram et al., 2011] at **Heidelberg University** and introduced in [Friedmann et al., 2017]. Currently, the second prototype version of the chip is available. In the remainder of this chapter an overview of the prototype chip in its second version, its capabilities and limitations, characteristics as well as the infrastructure implemented around the chip will be given. Furthermore, for the sake of verbosity, the second prototype version of the HICANN-DLS chip will simply be referred to as DLSv2.

4.1 Overview

The DLSv2 implements a spiking neurons in analog hardware, as well as hardware synapses that enable synaptic connections between them. Specifically, 32 neurons are arranged in an line with a synapse array above. The layout is depicted in Figure 4.1. A particular feature of this novel design is that there exists a seperate digital processor on the chip, whose computational capabilities are devoted to compute and execute plasticity rules for the neural network. The characteristics of this digital coprocessor are described in Section 4.5.

The focus parts of such a neuromorphic chip are the neurons and synapses. Those components are implemented in analog circuits and enable the simulation of a spiking neural network. The neuron model is presented in Section 4.2, whereas the synapse model and the workings of the whole synapse array are described in Section 4.3. A notable characteristic of the DLSv2 is that the hardware elements enable dynamics of the membrane voltage and spike times that are **1000 fold** faster than their biological counterparts.

To simulate a rich set of neural responses, the behaviour of the single neurons may be altered by an analog parameter storage. In addition, due to the fact that analog components exhibit device mismatch even on the same wafer, this parametrization of the single neurons behaviour enables also the possibility of calibration. The analog parameter storage is described together with the topic of calibration in Section 4.4.

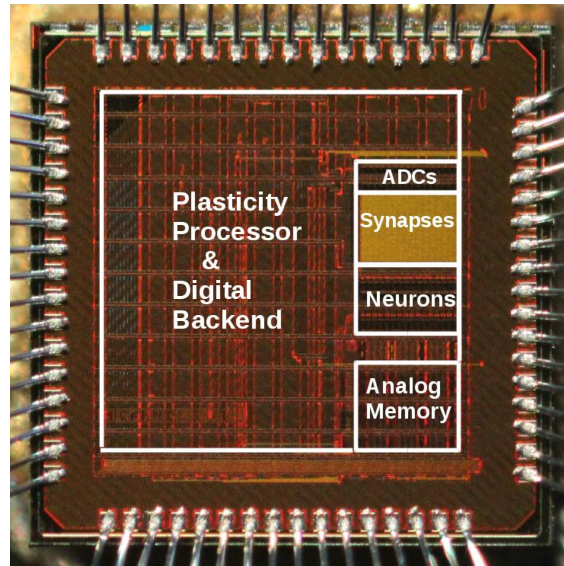

Figure 4.1: The layout of the chip, taken from [Aamir et al., 2016]. All major components to this thesis are observable. The synapse array is described in Section 4.3, the neurons in Section 4.2, the analog parameter storage in Section 4.4 and especially important to this thesis is the plasticity processing unit (PPU), described in Section 4.5.

4.2 Hardware Neurons

The neuron model that is implemented in the DLSv2 is Leaky integrate-and-fire (LIF), as described in Section 1.3.1. In terms of the hardware, it is practical to write the differential equation with the leak conductance $g_{\text{leak}} = \frac{C_{\text{mem}}}{\tau}$, identical to [Aamir et al., 2016]:

$$C_{\text{mem}} \frac{\mathrm{d}v_{\text{mem}}(t)}{\mathrm{d}t} = -g_{\text{leak}}(v_{\text{mem}}(t) - v_{\text{leak}}) + I(t) \tag{4.1}$$

In addition, the DLSv2 implements a common extension to the standard LIF model, which is the introduction of a refractory period of length τ_{ref}, during which the membrane voltage is clamped to the reset potential v_{reset}:

$$\forall t \in [\hat{t}, \hat{t} + \tau_{\text{ref}}] : v_{\text{mem}}(t) = v_{\text{reset}} \tag{4.2}$$

when $\hat{t}$ denotes the time of last spike of the considered neuron.

The circuit as it is implemented in hardware consists of several components and an overview is depicted in Figure 4.2. The membrane potential is modelled by a capacitor C_{mem} that in turn receives synaptic input current stemming from the synapses in the synapse array (Section 4.3.1). The capacitor also leaks charge by a connected leakage circuit. If the voltage of capacitor rises above a value of v_{thresh}, then the “SpikeGen” circuit triggers a spike that is digitally transmitted to the infrastructure around and can also be fed back into the synapse array. At the same time, the membrane capacitor is clamped to the reset voltage v_{reset} for a duration τ_{ref}, representing the refractory period.

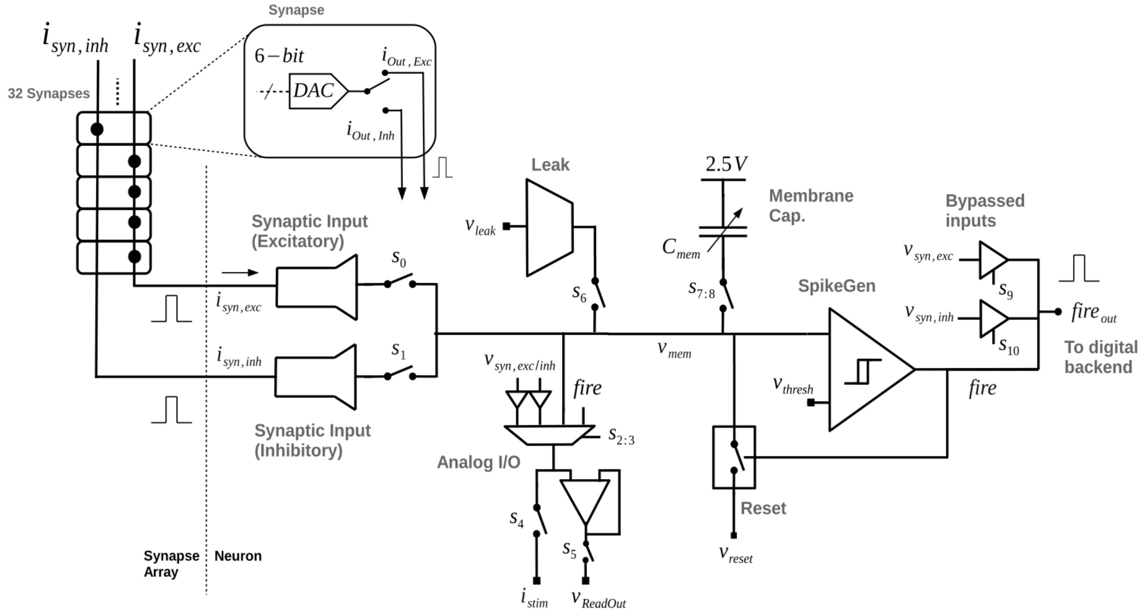

Figure 4.2: Neuron circuit overview, taken from [Aamir et al., 2016]. The main components are the capacitor C_{mem} that models the membrane voltage, the leakage circuit responsible for the continuous discharge of the capacitor, the "SpikeGen" circuit that will trigger a spike if the voltage in the capacitor rises above the threshold voltage v_{thresh}. Simultaneously to emitting a spike the voltage of the capacitor is clamped to the reset voltage v_{reset} for the duration of the refractory period τ_{ref}. Input current is received by the circuits in the synapse array, as described in Section 4.3.1.

4.3 Hardware Synapses

The synapse model chosen for the DLSv2 is of a current-based kind. Each synapse transforms spikes from neuron j according to the following equation, taken from [Stöckel, 2017], into current I_{ij} that is in turn input to the postsynaptic neuron i:

$$\tau_{\text{syn}} \frac{dI_{ij}(t)}{dt} = -I_{ij}(t) + \sum_{\hat{t}_j} w_{ij} \cdot \delta(t - \hat{t}_j) \tag{4.3}$$

with τ_{syn} the synaptic time constant, $\hat{t}_j$ sums over all spikes of neuron j and w_{ij} representing the weight of the projection $j \to i$. Thus, the entire synaptic input to neuron i is given by:

$$I_i(t) = \sum_j I_{ij}(t) \tag{4.4}$$

4.3.1 Synapse Array

All synapses are arranged in a 32×32 array, so that a full connection matrix between all the neurons, self-connections included, is possible. Each column in the matrix is connected to a single neuron. Hence, input can arrive at a neuron only via 32 different synapses. Each row is driven by a synapse driver that can be configured to be either excitatory or inhibitory.

A particular feature of the spikes in hardware is that each spike is equipped with an address. This address can be regarded as an additional piece of information required when such a spike enters the synapse array. Each synapse in the array is not only characterized by a weight, as required by Equation 4.3, but also an address can be set for each synapse. Only if the address

of an incoming spike matches the address of the considered synapse, the synapse will transform the spike into current input for the connected neuron. Otherwise, if the address of the incoming spike and the address set in the synapse do not agree, the spike will not cause any input.

Spikes are always injected by activating a certain synapse driver (row in the matrix picture). Then, all synapses in this row compare their address with the address of the incoming spike. Subsequently, all synapses that match the address of the spike convert the spike to current input for the connected neuron. To illustrate the process further, Figure 4.3 visualizes the circumstances in terms of synapse drivers and connected neurons.

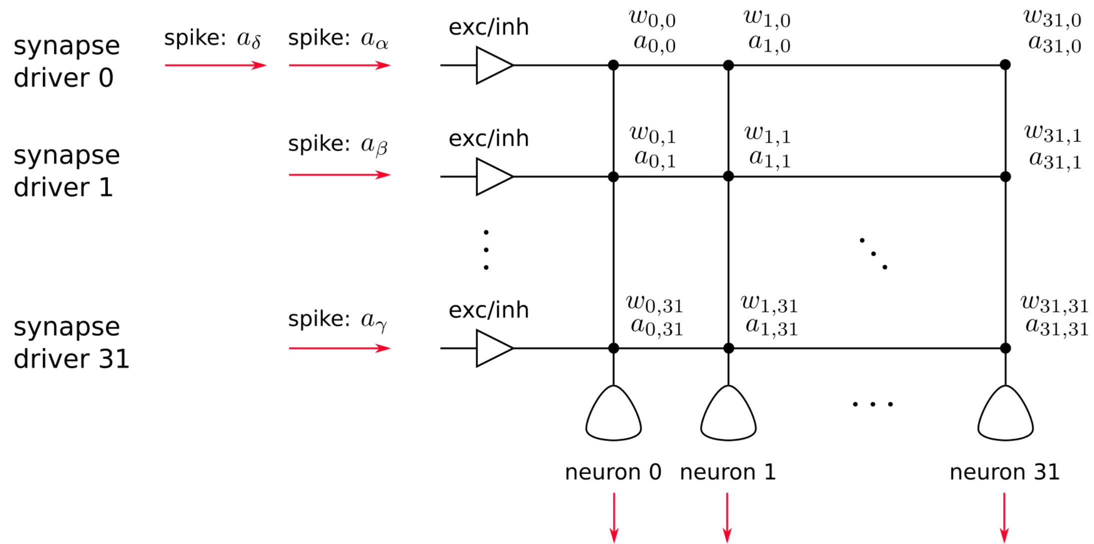

Figure 4.3: Visualization of spikes that are introduced to the synapse array. A spike is sent to the synapse array by activating a certain synapse driver that can be configured either excitatory (exc) or inhibitory (inh). Each spike is equipped with a certain address that is compared to the addresses $a_{i,j}$ stored in each synapse of the row corresponding to the activated synapse driver. If the address matches, the synapse scales the spike by the stored weight $w_{i,j}$, if the addresses do not match, the respective synapse disregards the spike.
For instance a spike with address a_α is introduced at synapse driver 0. Neuron 0 will receive synaptic input, scaled by $w_{0,0}$, if $a_\alpha = a_{0,0}$. Likewise, neuron 31 will receive synaptic input, scaled by $w_{31,0}$, if $a_\alpha = a_{31,0}$.

Important: The weights stored in the synapses, as well as the addresses are **6-bit discrete values**. It must be beared in mind that there exists a quite limited number of weight settings (64 different values), when it comes to practical implementations.

4.4 Analog Parameters - Calibration

Neuron models in general are characterized by a number of parameters. For example membrane time constant, threshold potential and leak potential to name a few. The hardware neurons on the chip are itself parametrized by values stored in the analog memory (Figure 4.1), giving the associated neuron desired behaviours. However, the neurons are implemented by analog circuitry, whose components always have diverse properties, even on a single chip. Therefore, the same set of analog values will result in a different behaviours among the hardware neurons. To counteract the negative effects caused by this diversity, each neuron can be configured with its own set of analog values. In particular, there are **18 individual parameters** for each neuron

and in addition, there is **1 global parameter** specifying the reset voltage v_{reset}, see [Aamir et al., 2016]. Each analog value is transferred to the DLSv2 as **10-bit** values (the highest of which, 1023, is reserved) and then, on chip digital-to-analog converted and loaded into the analog memory storage. A special circuit takes care of refreshing the values in this memory.

Finding the correct values for the desired dynamical properties is a non-trivial task and referred to as calibration. Thorough characterizations of these analog values, as well as methods to calibrate them, are to be found in [Stradmann, 2016] and [Wunderlich, 2016]. Indeed, the work of Stradmann is actively used in this thesis to achieve a proper and similar behaviour among all neurons. The analysis of the effects of calibration are found in Section 5.1.2, as the requirements are specific to the hardware experiments in this thesis.

4.5 Plasticity Processing Unit - PPU

A novel addition to the predecessors of the DLS is the **Plasticity Processing Unit** (PPU). This general purpose digital coprocessor is assigned with the task of computing updates to the weights of each synapse in the neural network. In the DLSv2 the PPU operates at 100 MHz. To obtain detailed information to the PPU one is referenced to [Friedmann, 2013] while the main features that were necessary to this thesis are presented in the remainder of this Section.

The PPU is equipped with 16 kB main memory of which 4 kB are considered as shared memory between the PPU and external host computers that control the DLS. In particular, this shared memory is given the name **mailbox** and its use is further described in Section 4.6. By its very purpose, the PPU is connected to the so-called synapse ram, which encompasses all weights and addresses that define the synapse array. Furthermore, the PPU is also equipped with an additional vector processing unit that can carry out multiple calculations (SIMD) at once, so as to compute many synaptic updates at the same time.

To create programs for the PPU a C-compiler and other necessary binary tools are provided.

4.6 Infrastructure

To operate the DLSv2 prototype an infrastructure around the silicon chip is required, an overview of the involved components is given in Figure 4.5. The chip is mounted on a test board, pictured in Figure 4.4, which itself is equipped with a FPGA, operating at 96 MHz, that serves several purposes, including

- enabling an USB communication link between the DLS and a host computer,
- controlling and playing back predefined external events for experiments, for instance the injection of spikes at certain times, or start and stop of the PPU,
- spike-routing, which will also be explained in this Section.

To execute experiments, there exists a `Python 2` API that implements the interfacing to the DLSv2. Via this API, one can, for example, set the weights and addresses in the synapse array, trigger spike injection at certain times, load a program onto the PPU, start and stop the PPU, write to or read from the mailbox and similar.

An important task of the FPGA is given by spike routing. Previously, it was mentioned that

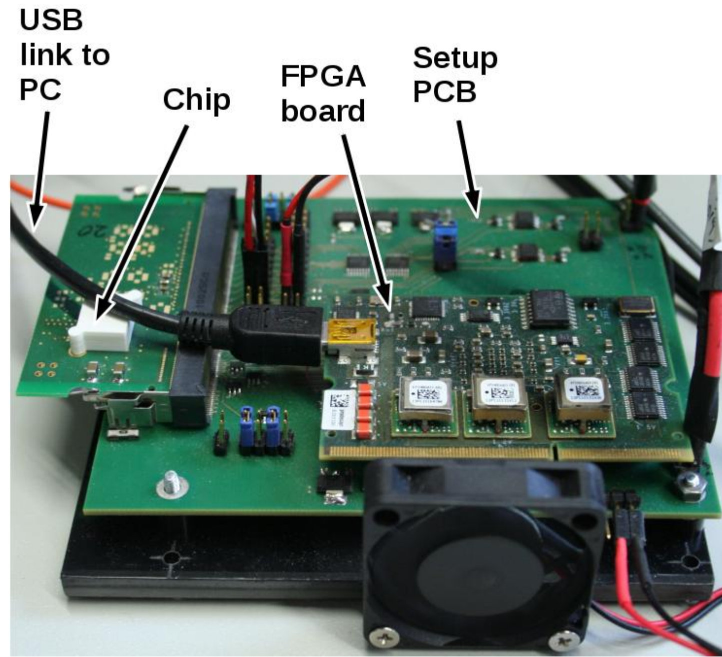

Figure 4.4: The test system, image taken from [Aamir et al., 2016]. It consists of the DLSv2 chip mounted on the Setup PCB, providing the necessary infrastructure for the chip. A FPGA is also connected to this board that handles the communication to a host computer (PC) over an USB communication link.

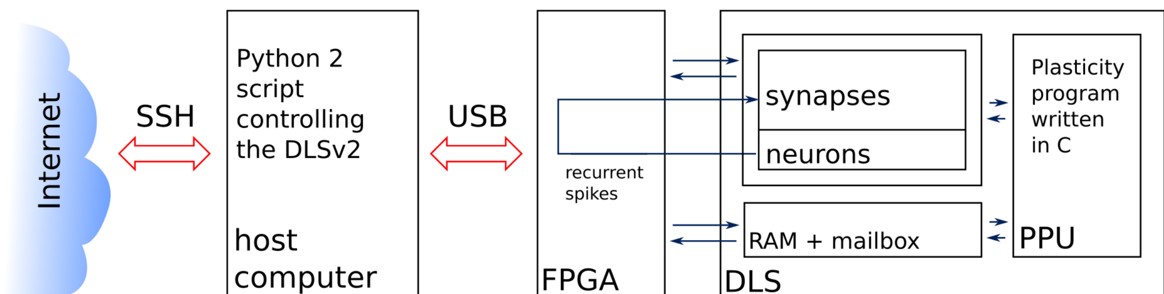

Figure 4.5: The DLSv2 is controlled through several layers. As the typical experimentator can be located anywhere, a SSH link over the internet is used to connect to a host computer. This machine executes a `Python 2` *script that interfaces via USB with the FPGA mounted on the test board. The FPGA in turn is not only responsible for communication but also controls the time flow of a certain experiment. Also, the FPGA receives the spikes emitted by hardware neurons and re-introduces them to the associated synapse driver in the synapse array.*

via the FPGA one can introduce spikes at certain times to the synapse array. Such spikes are declared as **input spikes**. However, the FPGA is also required when a hardware neuron emits a spike and this spike is supposed to be re-injected to the synapse array, which is required if one wants to build a network of spiking neurons. In the context of the hardware, such spikes that are emitted by a neuron on the chip and re-introduced to the synapse array, are called **recurrent spikes**, despite the actual meaning of recurrent spikes in biology.

There exist several modes for the FPGA to handle recurrent spikes. In the simplest version, which is used in this work, if neuron i emits a spike it will activate only the i-th synapse driver and thus only synapses in this row are able to handle spikes of this neuron. Combined with the

fact that each synapse driver is either excitatory or inhibitory, this leads to the fact that each neuron can have either excitatory or inhibitory effects on other neurons. Operating the FPGA in this mode, it is important for the practical use of the DLSv2 that all recurrent spikes (emitted from hardware neurons and sent to hardware neurons) **share the same address**. This must be considered when configuring the synapse array.

It is also possible to let the PPU program send spikes to the synapse array, which are denoted as **PPU spikes** throughout the remainder of this thesis. Such spikes elicited by the PPU and input spikes can be specified to have any addresses.

4.7 Limitations and Constraints

Compared to a computer simulation, when designing experiments on the HICANN-DLSv2 chip hardware constraints and problems need to be considered. This Section is intended to highlight and recall important constraints and characteristics

- The number of hardware neurons on the DLSv2 is 32.
- The synapse array supports 32 synapses for each neuron but each synapse driver (row in the matrix view) can be configured to be either excitatory or inhibitory
- Recurrent spikes have the same address
- Spike packets can only be introduced to the DLSv2 in distances of 48 FPGA cycles, as discussed with David Stöckel (FPGA clock: 96 MHz)
  If spike packets arrive faster, they get queued and congestion happens. Effectively, this results in a lower bound of synaptic delay
- The host computer API is `Python 2`, `Python 3` is not supported
- The PPU only supports a fixed point number representation, floating point operations are unavailable
- It is unadvisable to abort an experiment during execution, as it may leave the test system in an unusable state, thus rendering a power cycle of the FPGA necessary
- Device mismatch of neurons and synapses must be considered when designing neural circuits

5 Implementation

A central point of this thesis is also to implement the designed neural circuit, given in Chapter 3.1, in concrete hardware. More specifically, by hardware is meant the novel prototype neuromorphic chip HICANN-DLSv2, being developed in Heidelberg, see Chapter 4.

5.1 Hardware Implementation

In order to port the conceived network structure of 3.1 to the hardware system, one needs to bear in mind all constraints, limitations and special properties that come along with such a system. For example, the restriction to 32 neurons limits the size of multi-armed bandits that can be considered. A more important limitation of the current prototype system are the time distances in which the synapse array can process incoming spikes, either external or reccurent spikes. In fact, the hard limitation currently present is that each 0.5 ms in biological time a spike packet, which can target more than one neuron, can be processed by the synapse array (see Section 4.6). Within this thesis, the setup is chosen in such a way that recurrent spikes take 1 ms in biological time to serve as input again. Thus, the synapse delay can be specified by 1 ms.

In contrast to software simulations in general, for example carried out by the `NEST` simulator [Peyser et al., 2017], the analog neurons implemented on chip behave nonhomogeneous and are subject to noise. Thus, in order for approaches that rely on the absolute comparability of synapse weights to work, methods to calibrate the single neurons are necessary. Empirically it was found that the nonhomogeneous behaviour is much more problematic than the noise present in the system.

5.1.1 Network Mapping

Recalling the proposed architecture of Chapter 3.1, the network model itself is deterministic and thus, agrees with the hardware neuron model, which is leaky integrate-and-fire (LIF). The infrastructure of the network of spiking neurons will itself be composed of a stimulating cue neuron and several action neurons, whose spikes ultimately trigger bandit arm pulls. Part of the designed structure was also an inhibitory population that introduces competition between action neurons. However, in order to have as little delay between action spike and mutual inhibition as possible, action neurons are set themselves to be inhibitory to other neurons. So, one action neuron has inhibitory connections to every other action neuron and hence, there is no need for an extra inhibitory population. To visualize the configuration on the neuromorphic chip, the circuit is shown in Figure 5.1. It demonstrates how the neurons and synapses from the model, given in 3.1, are mapped to the hardware.

For action selection the stimulation neuron is excited by an external spike, being sent by the

PPU. This input is set to be so strong that the stimulating neuron itself spikes subsequently. To have an ongoing stimulation, the stimulating neuron is recurrently connected to itself by a strong connection. Therefore, permanent spiking of this stimulating neuron is generated, with a rate determined by the refractory period and the delay of recurrent spikes. Since the stimulating cue neuron projects to the action neurons, the ongoing stimulation will eventually trigger a spike of one action neuron. If an action neuron elicits a spike, inhibitory synaptic connections to the stimulating cue neuron stop the stimulation. However, since a spike of an action neuron can not be guaranteed if the weights are too small, there is a mechanism implemented in the PPU that sets the recurrent self-connection of the cue neuron to zero after a certain time. Given this scenario that no action neuron spiked in a given time, a random action is chosen.

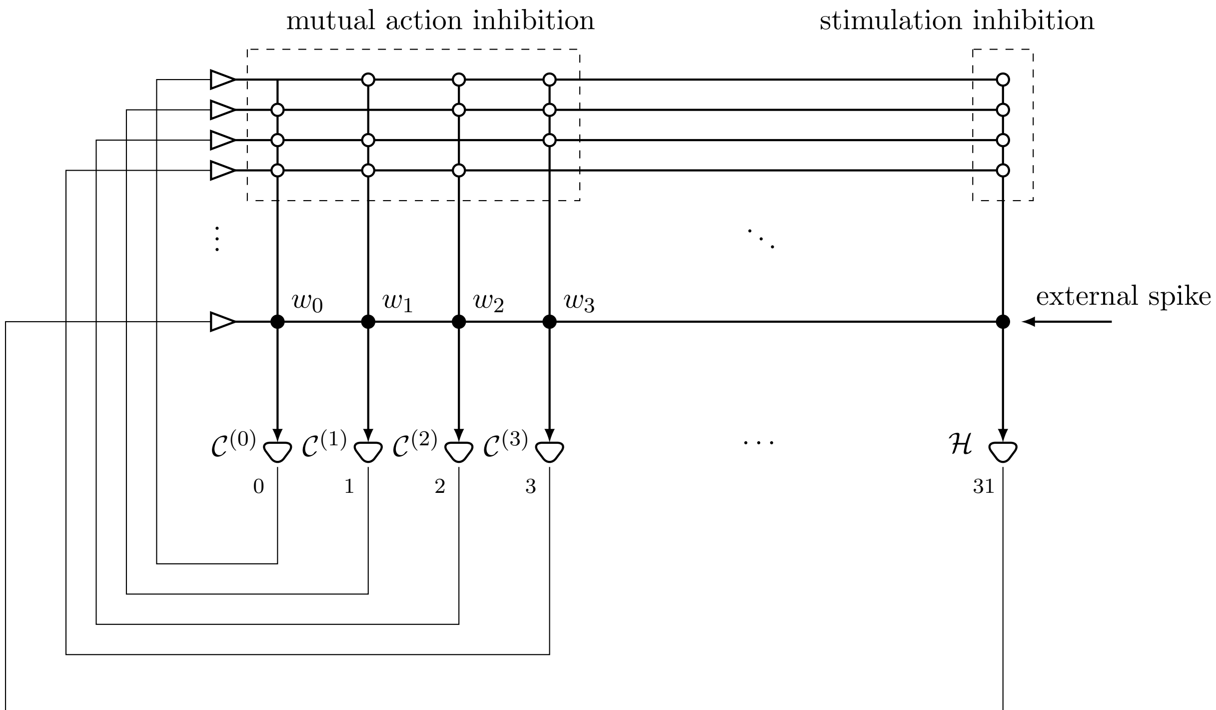

Figure 5.1: Shown in the bottom are 32 hardware neurons on the chip. Each action is mapped to a hardware neuron beginning from index zero. The last neuron is set to be the stimulating cue neuron. The recurrent connection from the cue neuron causes a permanent spiking of itself while also stimulating the action neurons. The stimulation is triggered by external spikes and inhibited if an action was selected. Filled black circles represent excitatory connections as opposed to the white circles that resemble inhibitory connections.

5.1.2 Used Hardware Calibration

The network design as it is, requires that stimulation of action neurons with similar weights from the stimulating cue neuron result in similar spike timings of those action neurons. It is therefore crucial for the ability of handling multi-armed bandit tasks that the considered neurons have identical behaviour with respect to the first spike time after stimulation. Two main constraints can therefore be formulated:

1. Stimulation through similar synaptic weights need to result in similar spike times among action neurons

2. Stimulation under different weights need to result in a difference in spike times that can be detected by the means of the PPU

In order to conduct experiments, the chip has to be provided with a file containing the values for the analog neuron parameters, representing a calibration. By default, the analog values are set to values that result in a reasonable, but diverse, spiking behaviour among all the neurons. However, this default configuration is not sufficient for the purposes of action selection by the spike times because both of the above mentioned constraints are violated when stimulating the neurons under a realistic setting. Stimulation takes place by a spike train with about 3 ms interspike distance (fastest possible recurrent spiking), biological time. To enhance the time difference between spike times of different weights, in order to improve the number of distinguishable weights, a certain property of the synapse array was used. Every spike input is transformed to a voltage pulse which gets converted to a postsynaptic current in a synapse. The length of this voltage pulse is configurable by the so-called spike pulse length and can be set to a lower value than the default one. This in turn results in less synaptic input and thus, because more stimulation is required, more weights can be distinguished. By stimulation with the default configuration, the times of first spikes are depicted in Figure 5.2 A, as opposed to the desired homogenious behaviour given in Figure 5.2 B. Two spikes are distinguishable by the PPU if they have a time difference of approximately 5 ms biological time or 5 µs chip time, which is clearly not met. More devastating even with this default configuration, a low weight to a particular action neuron achieves an earlier spike time than a high weight to some other action neuron.

A calibration or optimization of the analog values is therefore required. For this purpose an optimization objective was constructed, a fitness function, optimized by an appropriate algorithm. Summing up the requirements, homogenetiy across the neurons in terms of spike times is the most important. Thus, experiments are constructed in the following way: All neurons are stimulated by identical spike trains and identical weights, resulting spikes of the neurons are recorded and mapped to a fitness value for maximization. A variety of fitness functions was tried in order to come up with a good configuration file, fulfilling the required constraints. Several optimization algorithms, being evolution strategies, crossentropy, gradient descent and variants were tested. An example result of such a optimization with evolution strategies is depicted in Figure 5.3. Unfortunately, the optimization of those parameters seems to be difficult and finding a fitness function that describes best the required homogenity is nontrivial, with only having access to spike times after stimulation. Since the calibration of the neurons is not the main task but merely serves as a starting point, the optimization trials, which approximately lasted for approximately 1-2 hours each, were decided to be stopped, in order to dedicate more time to other topics.

Luckily, the bachelor thesis of Yannik Stradmann [Stradmann, 2016] was themed by exactly such a calibration task. Unlike an optimization that solely considers spike times, he used physical models of the underlying implementation and voltage measurements, to achieve the goal of a homogenious calibration. Unfortunately, by the time being, only a few chips with such an existing calibration is online. Figure 5.4 displays the time of the first spikes of all neurons using the calibration carried out by Stradmann, which proves to be the most promising among the others.

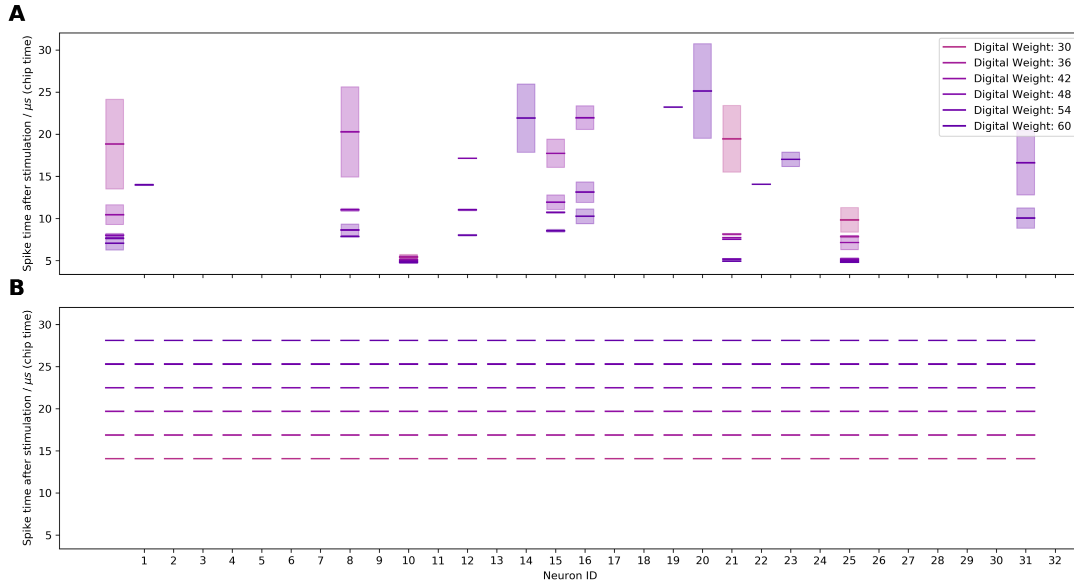

Figure 5.2: Stimulation of all neurons by a spike train with interspike distance of 3 ms *biological time and mimimal spike pulse length. Each figure consists of several different color codings representing a digital weight value. The x-axis refers to the different neurons on the chip while the y-axis represents the time of the first spike after stimulation is applied. Time is given in microseconds of chip time, equivalently to milliseconds biological. The shaded are represents the standard deviation in a batch of size 10.* ***A****: Default analog parameter configuration* ***B****: Desired homogenity across different neurons*

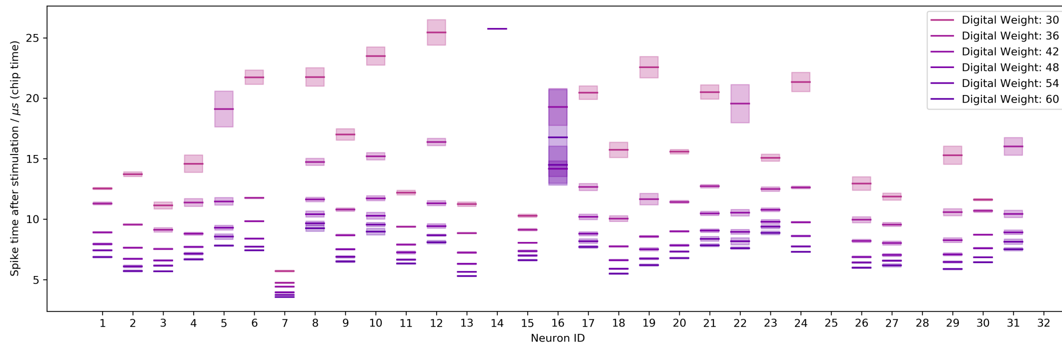

Figure 5.3: Spike timings of different weight settings using the calibration obtained by optimization of single spike times by the means of cross entropy. Stimulation of all neurons by a spike train with interspike distance of 1 ms *biological time and minimal spike pulse length. Each figure consists of several different color codings representing a digital weight value. The x-axis refers to the different neurons on the chip while the y-axis represents the time of the first spike after stimulation is applied. Time is given in microseconds of chip time, equivalently to milliseconds biological. The shaded are represents the standard deviation in a batch of size 10.* ***A****: Default analog parameter configuration* ***B****: Desired homogenity across different neurons*

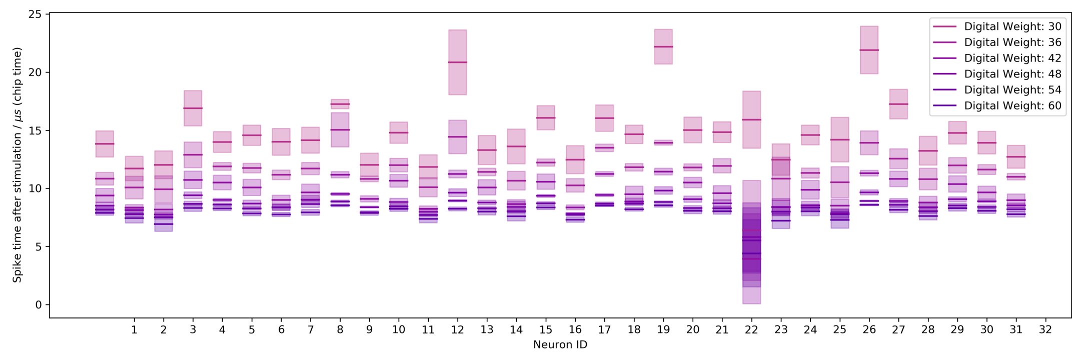

Figure 5.4: Spike timings of different weight settings using the calibration carried out by Yannik Stradmann [Stradmann, 2016]. Stimulation of all neurons by a spike train with interspike distance of 3 ms *biological time and minimal spike pulse length. Each figure consists of several different color codings representing a digital weight value. The x-axis refers to the different neurons on the chip while the y-axis represents the time of the first spike after stimulation is applied. Time is given in microseconds of chip time, equivalently to milliseconds biological. The shaded are represents the standard deviation in a batch of size 10.*

5.1.3 Infrastructure Setup

This Section describes the software setup and clarifies which component will execute which code.

As the main point is to come up with an agent for multi-armed bandits, it is necessary to emulate such a bandit task in the first place. Since, the spiking neural network on the DLSv2 plays the role of the agent, it is advisable to emulate the reinforcement learning environment, the multi-armed bandit, in a close neighboorhood in terms of communication. Therefore, it was decided that the **PPU will compute the environment**. So, the actual purpose of computing synaptic weight updates is, in this case, extended to also simulating the multi-armed bandit. This is a feasible design decision because a multi-armed bandit is a computational very undemanding reinforcement learning environment. It is only required to sample binary rewards (Bernoulli bandits) according to fixed reward probabilities.

Hence, a `C`-program, executed by the PPU, was developed that encompasses not only the proposed update rules, as in Section 3.2, but also includes procedures to trigger action selection processes and the computation of binary rewards according to which action was selected by the spiking neural network. In addition, this program also needs to implement routines to exchange task information, e.g. reward probabilities, and results of a roll out via the mailbox (shared memory).

On the host computer, a `Python 2` script controls the execution of such experiments and takes care of proper configuration and calibration. This script passes the specific multi-armed bandit tasks via the mailbox to the DLSv2 and receives the results of the experiment likewise.

Approximately, the following tasks are executed when conducting a single multi-armed bandit experiment:

1. Host computer: Establish connection to a DLSv2 chip
2. Host computer: Configure synapse array to a valid configuration, network structure is loaded, initial weights are set
3. Host computer: Spike pulse length is set to the minimal value, the effect of which is explained in Section 5.1.2
4. Host computer: Calibration file is loaded and the DLSv2 analog parameter storage is set to the calibrated values
5. Host computer: PPU program is loaded, contains plasticity rules and the environment
6. Host computer: FPGA spike routing for recurrent spikes is enabled
7. Host computer: Mailbox is filled with necessary parameters, such as number of arms, reward probabilities, hyperparameters
8. Host computer: Trigger the start of the PPU
9. DLSv2: Execute multi-armed bandit environment, run the network of spiking neurons, apply plasticity rules
10. DLSv2: Write results to mailbox
11. Host computer: Read back results from mailbox for evaluation
12. Host computer: Close connection

For further experiments regarding learning to learn, an extension to the previous configuration is made. In the LTL case, the host computer executes the learning to learn framework as described in Chapter 2. In particular an optimizer executes multi-armed bandit tasks in a loop, trying to find the set of hyperparameters with the best performance on the considered task family. For a visual representation of the circumstances one is referred to Figure 5.5.

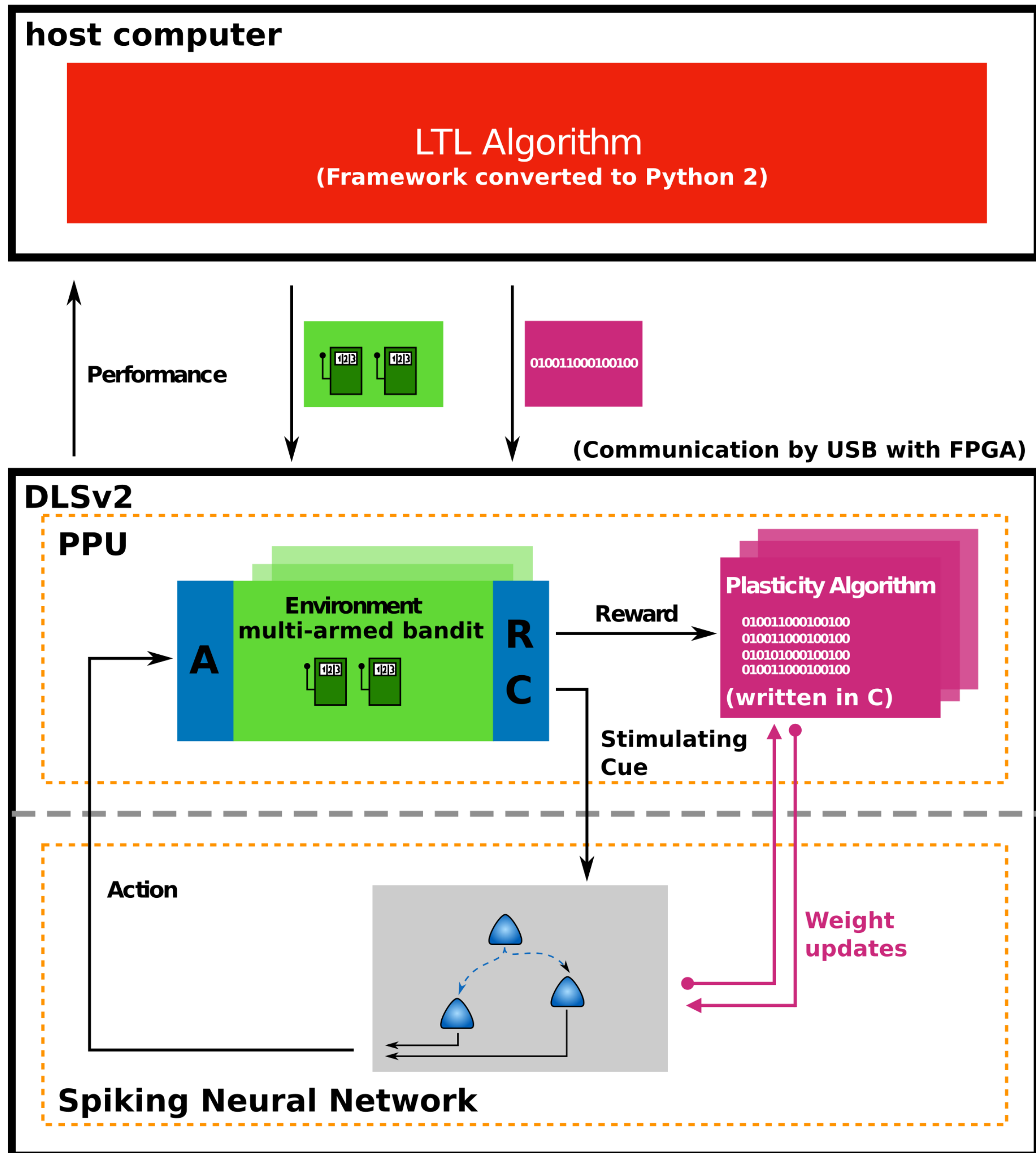

Figure 5.5: Visual representation of LTL applied to multi-armed bandits on the DLSv2. The figure was firstly made by Christian Pehle and was adapted to match the situation when executing LTL on multi-armed bandit tasks.
The PPU performs weight updates via the previously loaded plasticity algorithm and additionally calculates rewards based on the actions that the spiking neural network in hardware proposed. Also, to trigger an action selection of the SNN, the PPU needs to stimulate the SNN via spikes. The LTL algorithm suggests a certain set of hyperparameters, which the DLSv2 computes the performance for. This value is given by the negative expected cumulative regret, for reasons to be found in Section 1.4.2.

6 Approach Validation

This section presents a validation of the proposed approach, as given in Chapter 3, by conducting experiments on the hardware. Experiments are given by reinforcement learning with independent multi-armed bandits. An instance of such a task is sampled and passed to the DLSv2. In particular, multi-armed bandits with 2 arms were considered. The spiking agent was then asked to interact with the bandit task for $N_P = 100$ arm pulls (task length). To see wether the approach is valid, the spikes produced in the network are examined and it is evaluated if the activity is similar to what is anticipated as in Chapter 3.

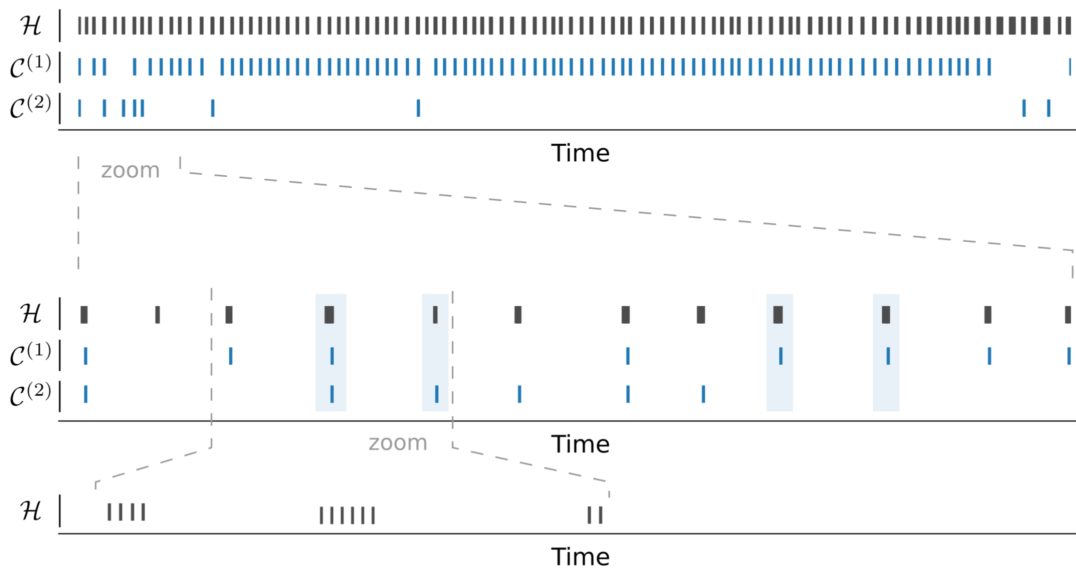

Figure 6.1: Spike raster of the spiking neural networks activity during a picked-out two-armed bandit task sample with reward probabilities $p_1 = 0.413, p_2 = 0.09$. The spiking neural network is equipped with the greedy update rule. ***Top****: Spikes within the whole duration of the bandit task. Unsureness about the probabilities in the beginning and later exploiting the better action choice.* ***Middle****: Zoomed-in view of the learning phase. Blue shaded rectangles indicate that a reward was obtained in this particular arm pull.* ***Bottom****: Stimulation neuron spikes. If no reward was obtained, the corresponding synapse weight gets lower and as a result, longer stimulation (number of stimulation spikes) is necessary.*

6.1 Greedy Update Rule

It is investigated on the level of network activity, given by the occured spikes, if the proposed greedy update rule, see Section 3.2.1 will exhibit the anticipated behaviour. In Figure 6.1 the network activity, in form of spikes, during a picked-out independent multi-armed bandit task is displayed. The reward probabilities of this instance are given by: $p_1 = 0.413$ and $p_2 = 0.09$. It is observable that the agent tries out the different possible actions and ultimately learns, by observing the given rewards, that bandit arm 1 should be pulled, in order to maximize the reward (minimize regret).

6.2 Incremental Update Rule

In order to investigate the incremental update rule, as presented in Section 3.2.2, Figure 6.2 shows the network activity, in form of spikes for a two-armed bandit task with reward probabilities of $p_1 = 0.621$ and $p_2 = 0.07$. It visualizes the learning and exploiting phase. As is observable in the Figure, if no reward is obtained, the weights get decreased which in turn makes more stimulation necessary to elicit a spike of an action neuron. Thus, the network activity matches the expectations of what was anticipated when designing the approach in Chapter 3.

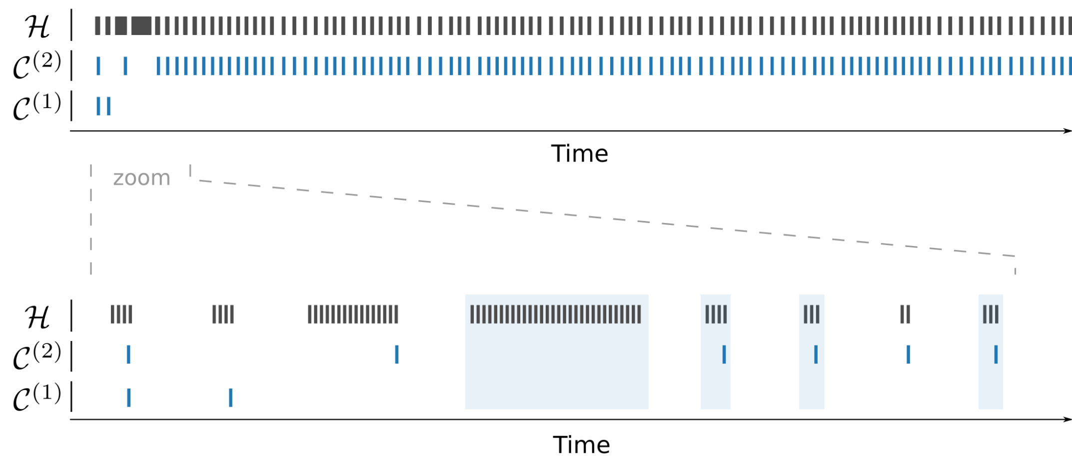

*Figure 6.2: Spike raster of the spiking neural networks activity. Update rule is the incremental update rule and activity is recorded during a picked-out two-armed bandit task sample with reward probabilities $p_1 = 0.621$ and $p_2 = 0.07$. **Top**: Spikes within the whole duration of the bandit task. Unsureness about the probabilities in the beginning and later exploiting the better action choice. **Bottom**: Zoomed-in view of the learning phase. Blue shaded rectangles indicate that a reward was obtained in this particular arm pull. Observation: The stimulating neuron has to produce more and more spikes if no rewards are observed, since the weights to action neurons get lowered. If at a certain point, no action neuron produces a spike anymore, random actions are committed as can be seen in the fourth action selection process.*

7 Learning to Learn Results

The presented weight update rules in Section 3.2 all incorporate hyperparameters required to be set to some value. Also, network specific hyperparameters arise, such as the strength of inhibition between action neurons. Despite educated guesses can be made, the best possible hyperparameter setting remains unknown beforehand. Therefore, in order to endow the spike-based agent with the best possible capababilities to handle a certain task family, task dependent hyperparameter optimization in the fashion of learning to learn, introduced in Chapter 2, is conducted. The **fitness** which is being maximized by learning to learn, is given by the negative expected cumulative regret at the end of the bandit task, averaged over 40 multi-armed bandit task samples of the considered family. This averaging elimitates some of the noise and is essential because single sampled bandit tasks are not representative to evaluate a fitness value. The length of a bandit task is set again to be $N_P = 100$ arm pulls. Indeed, the desired result is that a certain setting of hyperparameters allows the agent to perform well on a particular family of tasks, effectively enabling transfer learning.

A common method, yet quite inefficient, is to perform a grid search over an appropriate region of hyperparameter settings. In a very low dimensional setting of up to 3 dimensions, perhaps also with a coarse discretization, there are no objections to do so. However, by the curse of dimensionality, this method becomes intractable by the fact of exponential growth of possible parameter combinations with an increasing number of hyperparameters. In fact, the problem turns even more severe if resources are limited, i.e. prototype hardware as in this case. To circumvent mentioned difficulties, learning to learn is executed to reduce the number of evaluations.

All plots in this Chapter share the same structure: On the left hand side the evolution of fitness is shown, depending on the number of evaluated hyperparameter settings. In particular, each of the used optimization algorithms uses in each iteration a number of hyperparameter settings. Thus, the sample mean of the fitnesses among those is shown as a graph. It must be kept in mind that the sample average of the fitnesses is not equal to fitness of the sample averaged parameters.

$$\mathbb{E}[f(x)] \neq f(\mathbb{E}[x]) \tag{7.1}$$

And thus, the actual performance of the hyperparameter settings after optimization is reported seperately in Section 7.4. On the right hand side is a plot showing the hyperparameter dynamics as projected to two dimensional space by demixed principal component analysis, see Section 1.6.

Another peculiarity regarding the LTL optimizations is that each hyperparameter was scaled to live in the range of $[0, 1]$. The reason for that is that some optimizer do random exploration steps. However, the size of which is the same in each direction, hence to achieve a uniform exploration rescaling had to be applied.

It is also important to mention that in the case of dependent bandits, the baseline of Gittins indices are computed as if they were facing an independent bandit.

7.1 Incremental Update Rule

This Section investigates hyperparameter optimization of spiking agents based on the incremental update rule, given in Section 3.2.2. Several hyperparameters control the properties of this update rule. In addition, there are hyperparameters concerning the spiking neural network itself, for instance the strength of inhibition. Both kinds of hyperparameters are optimized together via learning to learn, in order to find the best performing values for the considered multi-armed bandit family. A summary of the hyperparameters that are subject to optimization is given in Table 7.1.

Table 7.1: Hyperparameters of the incremental update rule, as in 3.2.2, together with hyperparameters concerning the spiking neural network. All action specific hyperparameters, i.e. if dependent on the bandit arm i, are set identical among all different arms.

Hyperparameter	Description
Initial Learning Rate $\epsilon_i^{(0)}$	Controls to which extent existing weights are overwritten by new information
Learning Rate Decay d	Multiplicative factor decreasing the learning rate over time
Stimulation Inhibition Strength	Spikes from action neurons inhibit the stimulating neuron. Hyperparameter controls the strength
Mutual Action Inhibition	Strength of mutual inhibition among action neurons
Initial Weight $w_i^{(0)}$	Synaptic weight of the connection from stimulating neurons to action neurons at start

7.1.1 Independent Bandits

In the Figures 7.1, 7.2, 7.3, 7.4 the results for LTL optimization of hyperparameters is shown for the optimizers parallel simulated annealing, cross entropy, finite difference gradient and evolution strategies respectively. The considered task family is given by **independent** multi-armed bandits with the incremental update rule 3.2.2.

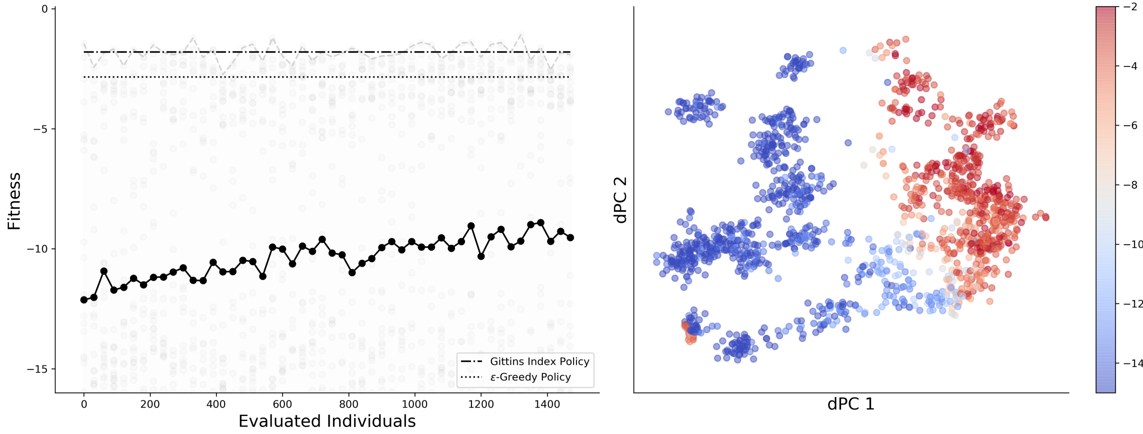

*Figure 7.1: Hyperparameter optimization by **simulated annealing**. Left: Mean fitness among the current population. Horizontal lines show the performance of well-known strategies as baselines. Gray, dashed lines represent the fitness boundaries of the current population. Gray dots signify the fitness of single individuals in the population. Right: Demixed principal component analysis of the hyperparameter setting of every population member evolving the generations.*

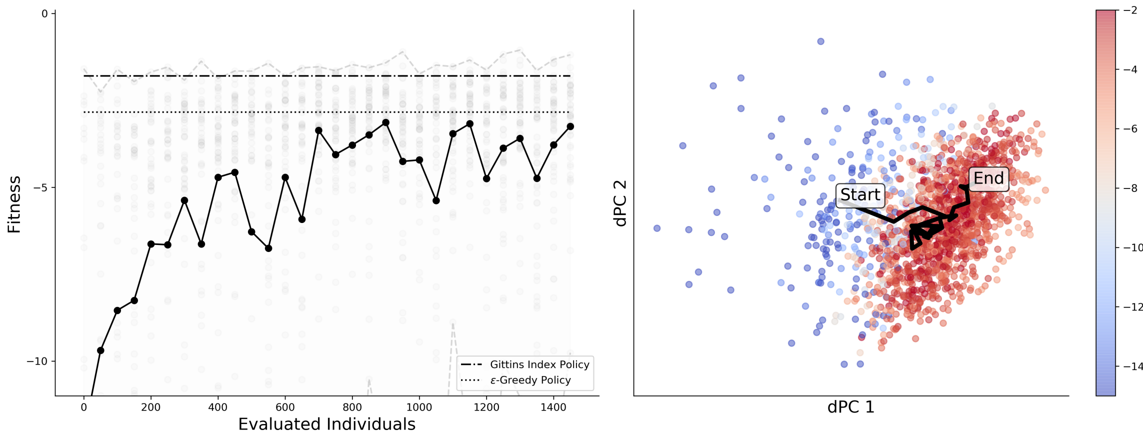

*Figure 7.2: Hyperparameter optimization by the **cross entropy** method. Left: Mean fitness among the current population. Horizontal lines show the performance of well-known strategies as baselines. Gray, dashed lines represent the fitness boundaries of the current population. Gray dots signify the fitness of single individuals in the population. Right: Demixed principal component analysis of the hyperparameter setting of every population member evolving the generations. Black thick line indicates the movement of the population mean parametrization over generations*

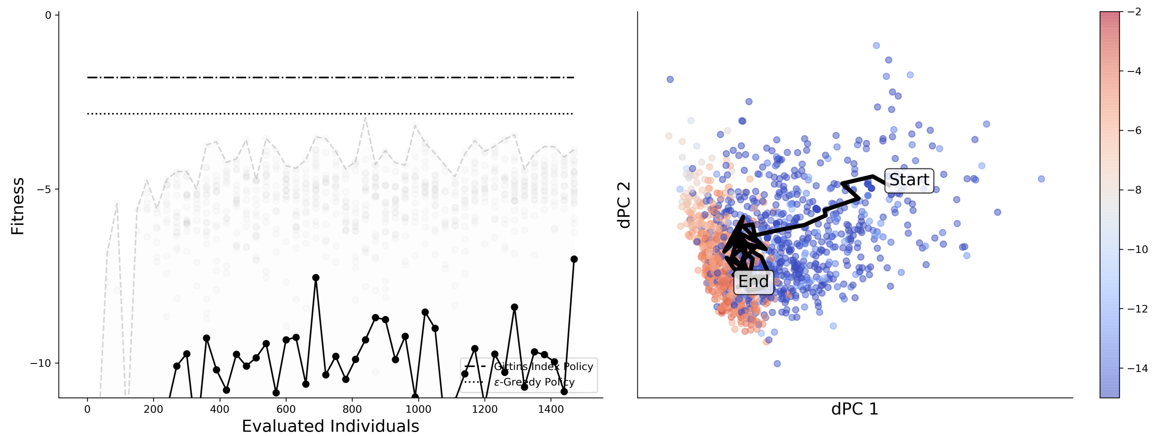

Figure 7.3: Hyperparameter optimization by the ***finite difference gradient*** *method. Left: Mean fitness among the current population. Horizontal lines show the performance of well-known strategies as baselines. Gray, dashed lines represent the fitness boundaries of the current population. Gray dots signify the fitness of single individuals in the population. Right: Demixed principal component analysis of the hyperparameter setting of every population member evolving the generations. Black thick line indicates the movement of the population mean parametrization over generations*

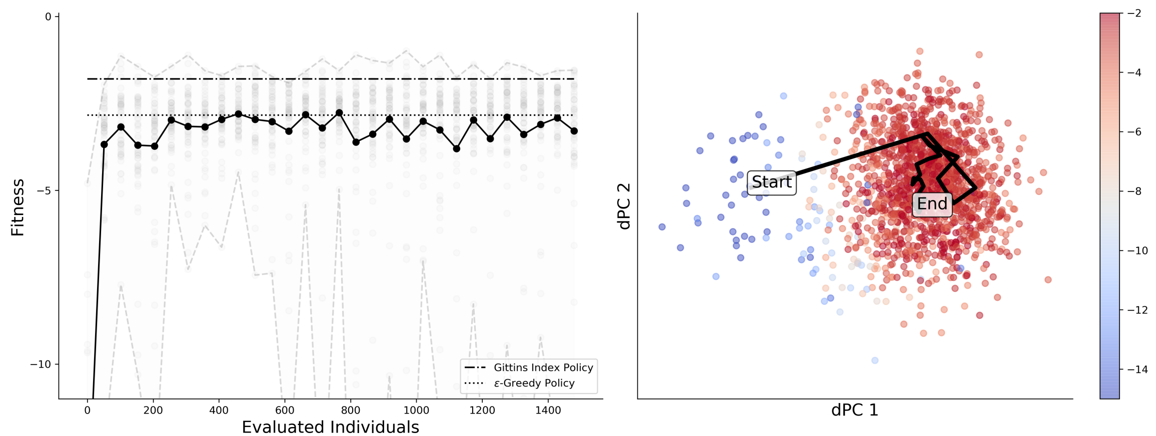

Figure 7.4: Hyperparameter optimization by the ***evolution strategies****. Left: Mean fitness among the current population. Horizontal lines show the performance of well-known strategies as baselines. Gray, dashed lines represent the fitness boundaries of the current population. Gray dots signify the fitness of single individuals in the population. Right: Demixed principal component analysis of the hyperparameter setting of every population member evolving the generations. Black thick line indicates the movement of the population mean parametrization over generations*

7.1.2 Dependent Bandits

In the Figures 7.5, 7.6, 7.7, 7.8 the results for LTL optimization of hyperparameters is shown for the optimizers parallel simulated annealing, cross entropy, finite difference gradient and evolution strategies respectively. The considered task family is given by **dependent** multi-armed bandits with the incremental update rule 3.2.2.

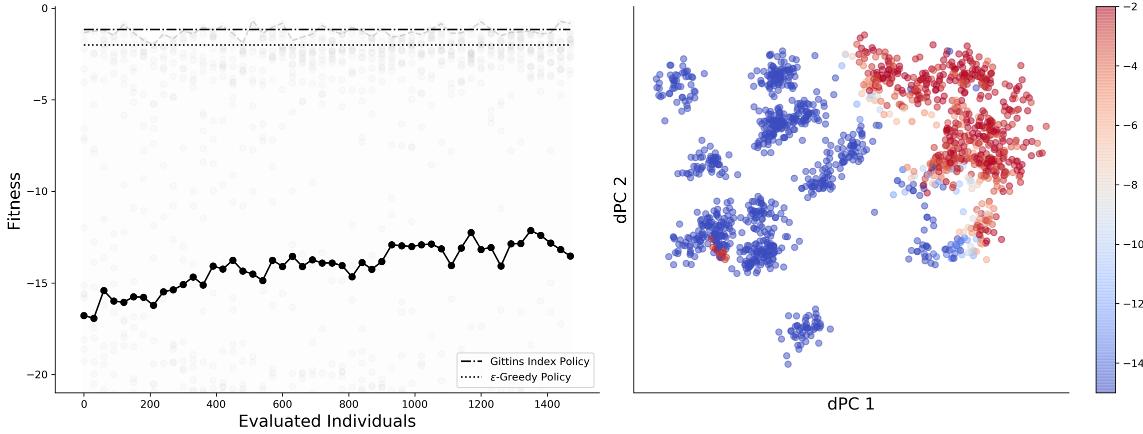

*Figure 7.5: Hyperparameter optimization by **simulated annealing**. Left: Mean fitness among the current population. Horizontal lines show the performance of well-known strategies as baselines. Gray, dashed lines represent the fitness boundaries of the current population. Gray dots signify the fitness of single individuals in the population. Right: Demixed principal component analysis of the hyperparameter setting of every population member evolving the generations.*

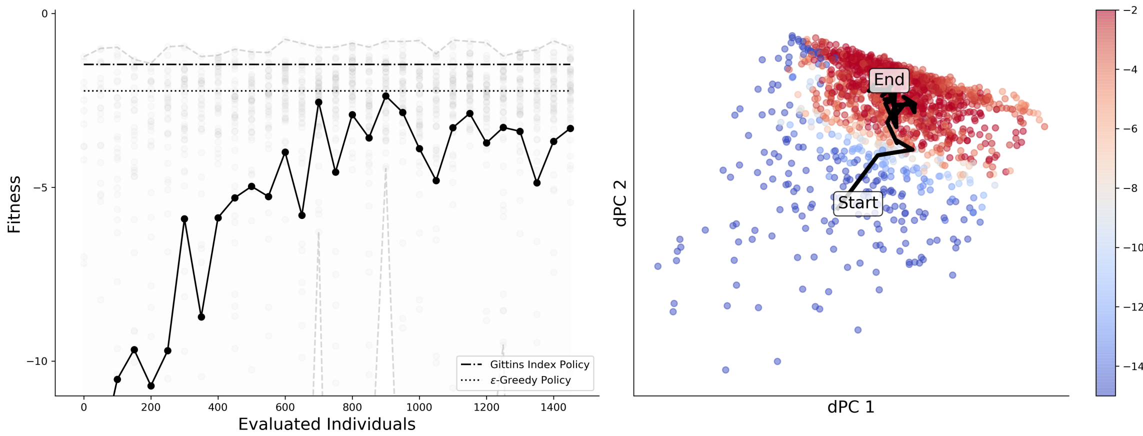

*Figure 7.6: Hyperparameter optimization by the **cross entropy** method. Left: Mean fitness among the current population. Horizontal lines show the performance of well-known strategies as baselines. Gray, dashed lines represent the fitness boundaries of the current population. Gray dots signify the fitness of single individuals in the population. Right: Demixed principal component analysis of the hyperparameter setting of every population member evolving the generations. Black thick line indicates the movement of the population mean parametrization over generations*

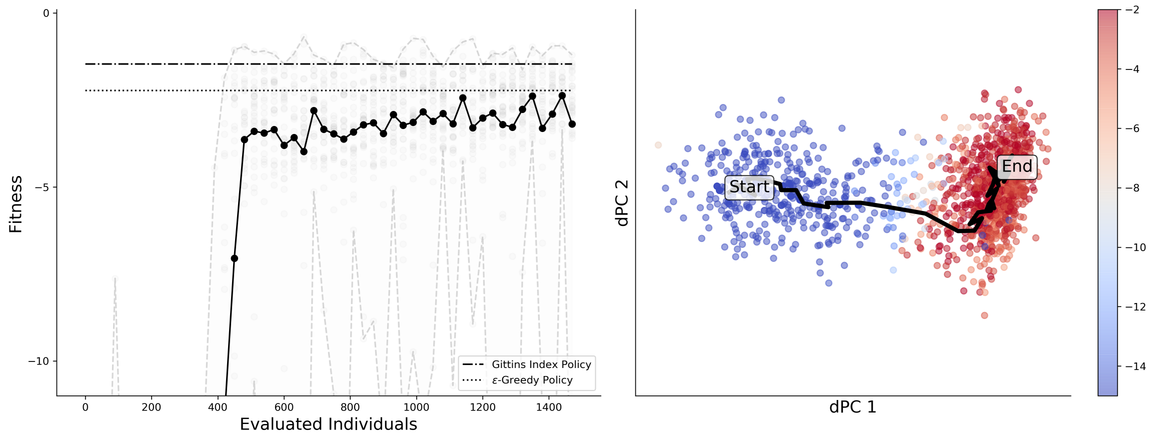

Figure 7.7: Hyperparameter optimization by the ***finite difference gradient*** *method. Left: Mean fitness among the current population. Horizontal lines show the performance of well-known strategies as baselines. Gray, dashed lines represent the fitness boundaries of the current population. Gray dots signify the fitness of single individuals in the population. Right: Demixed principal component analysis of the hyperparameter setting of every population member evolving the generations. Black thick line indicates the movement of the population mean parametrization over generations*

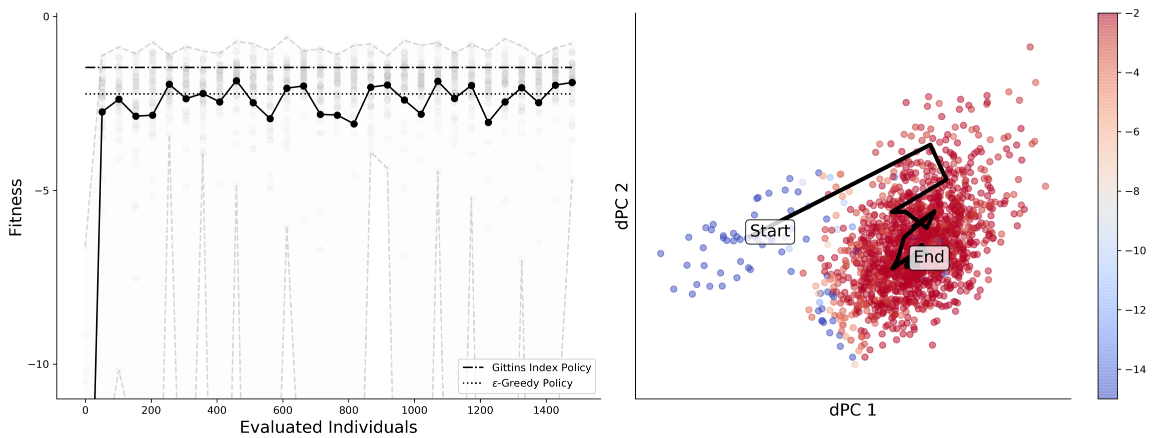

Figure 7.8: Hyperparameter optimization by the ***evolution strategies****. Left: Mean fitness among the current population. Horizontal lines show the performance of well-known strategies as baselines. Gray, dashed lines represent the fitness boundaries of the current population. Gray dots signify the fitness of single individuals in the population. Right: Demixed principal component analysis of the hyperparameter setting of every population member evolving the generations. Black thick line indicates the movement of the population mean parametrization over generations*

7.2 Greedy Update Rule

This Section in turn lists the results on learning to learn optimizations concerning the greedy update rule, given in 3.2.1. For this update rule there exist hyperparameters that characterize the prior belief over the reward probabilities. In the case of independent bandits, when the reward probabilities are sampled from $[0, 1]$, the prior should be uniform. However, the optimization comes up with different settings for those parameters. The cause for this to happen is that the weights in the spiking neural network are responsible for the action decision. Thus, if the resulting weight updates allow the spiking agent, as a whole, to perform better, the LTL algorithm will prefer this setting. In addition, there are again the hyperparameters concerning the spiking neural network itself, for instance the strength of inhibition. Both kinds of hyperparameters are optimized together via learning to learn, in order to find the best performing values for the considered multi-armed bandit family. A summary of the hyperparameters that are subject to optimization is given in Table 7.2.

Table 7.2: Hyperparameters of the greedy update rule, as in 3.2.1, together with hyperparameters concerning the spiking neural network. All action specific hyperparameters, i.e. if dependent on the bandit arm i, are set identical among all different arms.

Hyperparameter	Description
$\alpha_i^{(0)}$	Parameter of the Beta prior distribution, corresponds to the number of successes observed prior to the experiment start
$\beta_i^{(0)}$	Parameter of the Beta prior distribution, corresponds to the number of losses observed prior to the experiment start
Stimulation Inhibition Strength	Spikes from action neurons inhibit the stimulating neuron. Hyperparameter controls the strength
Mutual Action Inhibition	Strength of mutual inhibition among action neurons

7.2.1 Independent Bandits

In the Figures 7.9, 7.10, 7.11, 7.12 the results for LTL optimization of hyperparameters is shown for the optimizers parallel simulated annealing, cross entropy, finite difference gradient and evolution strategies respectively. The considered task family is given by **independent** multi-armed bandits with the greedy update rule 3.2.1.

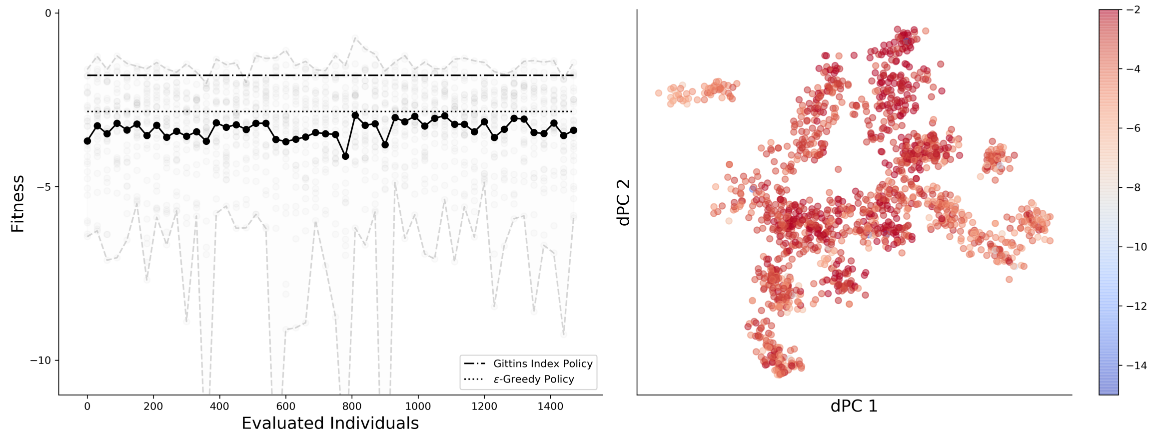

*Figure 7.9: Hyperparameter optimization by **simulated annealing**. Left: Mean fitness among the current population. Horizontal lines show the performance of well-known strategies as baselines. Gray, dashed lines represent the fitness boundaries of the current population. Gray dots signify the fitness of single individuals in the population. Right: Demixed principal component analysis of the hyperparameter setting of every population member evolving the generations.*

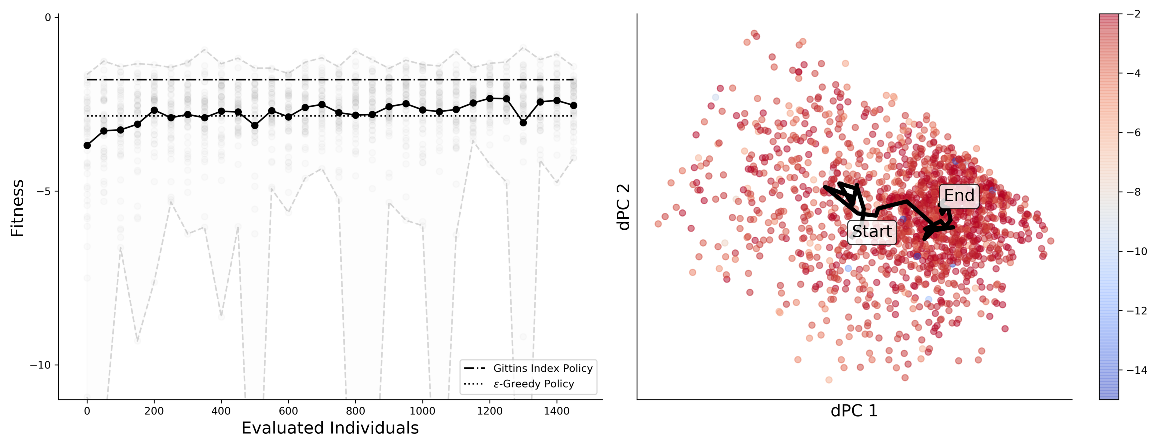

*Figure 7.10: Hyperparameter optimization by the **cross entropy** method. Left: Mean fitness among the current population. Horizontal lines show the performance of well-known strategies as baselines. Gray, dashed lines represent the fitness boundaries of the current population. Gray dots signify the fitness of single individuals in the population. Right: Demixed principal component analysis of the hyperparameter setting of every population member evolving the generations. Black thick line indicates the movement of the population mean parametrization over generations*

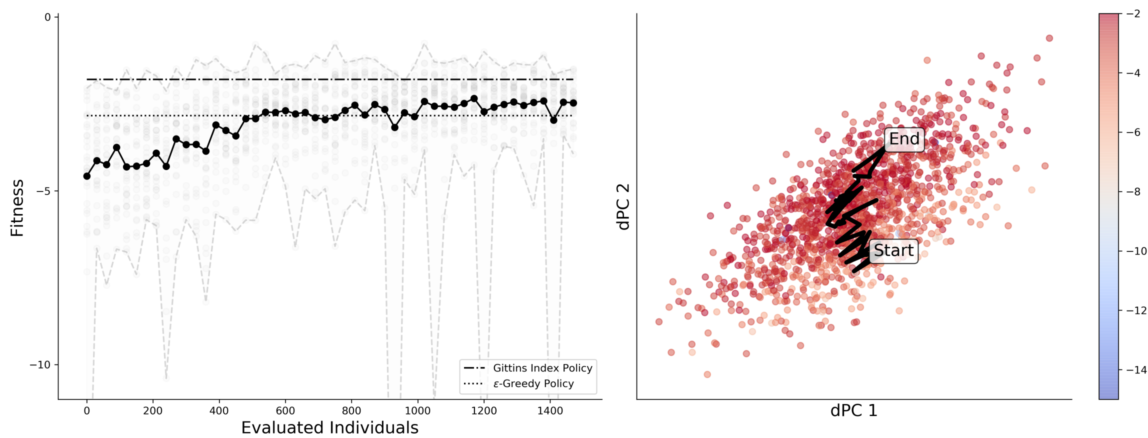

Figure 7.11: Hyperparameter optimization by the ***finite difference gradient*** *method. Left: Mean fitness among the current population. Horizontal lines show the performance of well-known strategies as baselines. Gray, dashed lines represent the fitness boundaries of the current population. Gray dots signify the fitness of single individuals in the population. Right: Demixed principal component analysis of the hyperparameter setting of every population member evolving the generations. Black thick line indicates the movement of the population mean parametrization over generations*

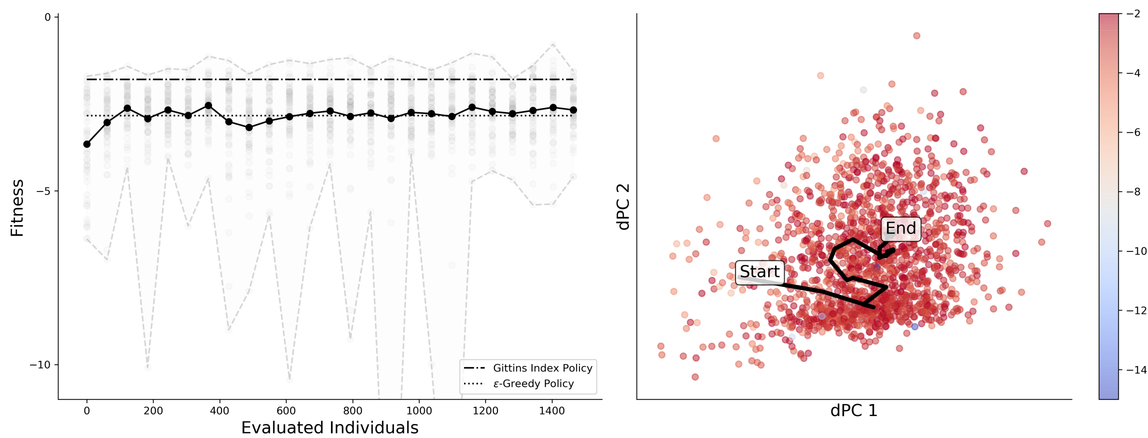

Figure 7.12: Hyperparameter optimization by the ***evolution strategies****. Left: Mean fitness among the current population. Horizontal lines show the performance of well-known strategies as baselines. Gray, dashed lines represent the fitness boundaries of the current population. Gray dots signify the fitness of single individuals in the population. Right: Demixed principal component analysis of the hyperparameter setting of every population member evolving the generations. Black thick line indicates the movement of the population mean parametrization over generations*

7.2.2 Dependent Bandits

In the Figures 7.13, 7.14, 7.15, 7.16 the results for LTL optimization of hyperparameters is shown for the optimizers parallel simulated annealing, cross entropy, finite difference gradient and evolution strategies respectively. The considered task family is given by **dependent** multi-armed bandits with the greedy update rule 3.2.1.

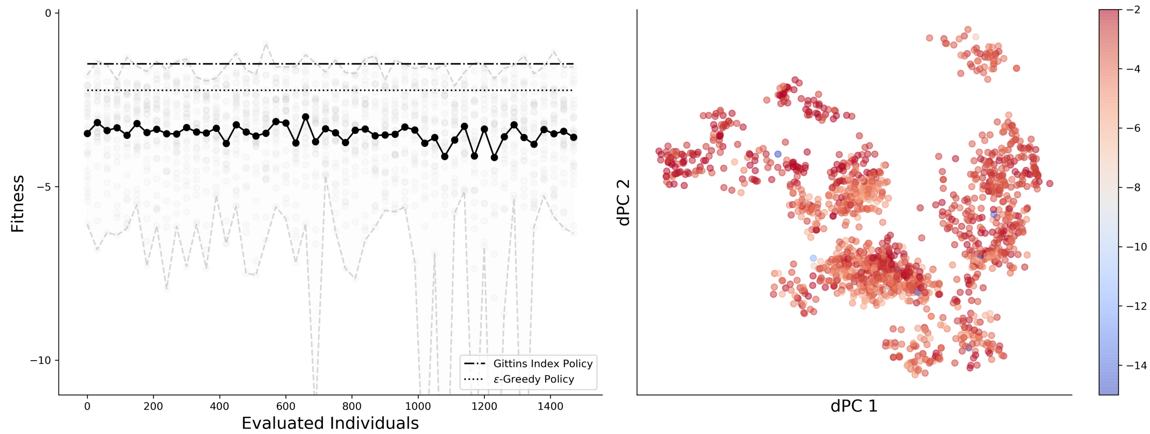

Figure 7.13: Hyperparameter optimization by ***simulated annealing****. Left: Mean fitness among the current population. Horizontal lines show the performance of well-known strategies as baselines. Gray, dashed lines represent the fitness boundaries of the current population. Gray dots signify the fitness of single individuals in the population. Right: Demixed principal component analysis of the hyperparameter setting of every population member evolving the generations.*

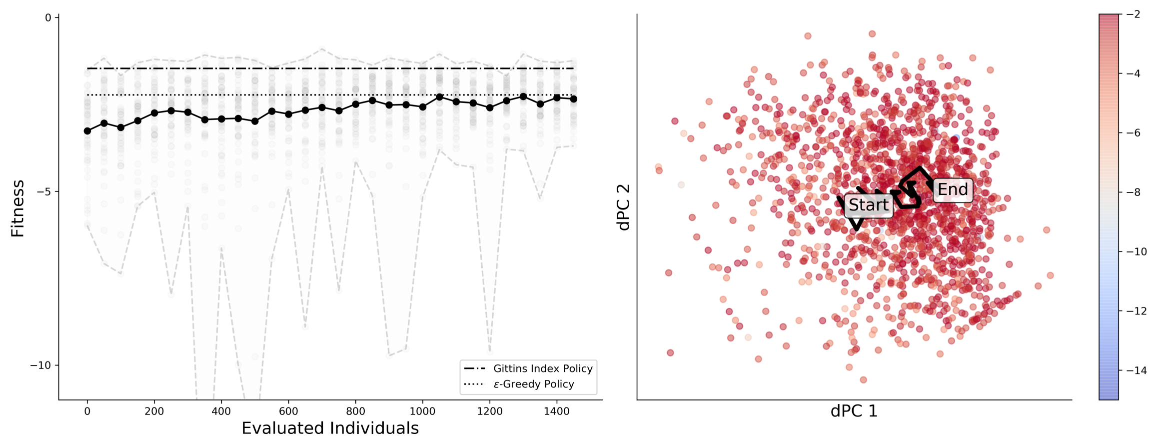

Figure 7.14: Hyperparameter optimization by the ***cross entropy*** *method. Left: Mean fitness among the current population. Horizontal lines show the performance of well-known strategies as baselines. Gray, dashed lines represent the fitness boundaries of the current population. Gray dots signify the fitness of single individuals in the population. Right: Demixed principal component analysis of the hyperparameter setting of every population member evolving the generations. Black thick line indicates the movement of the population mean parametrization over generations*

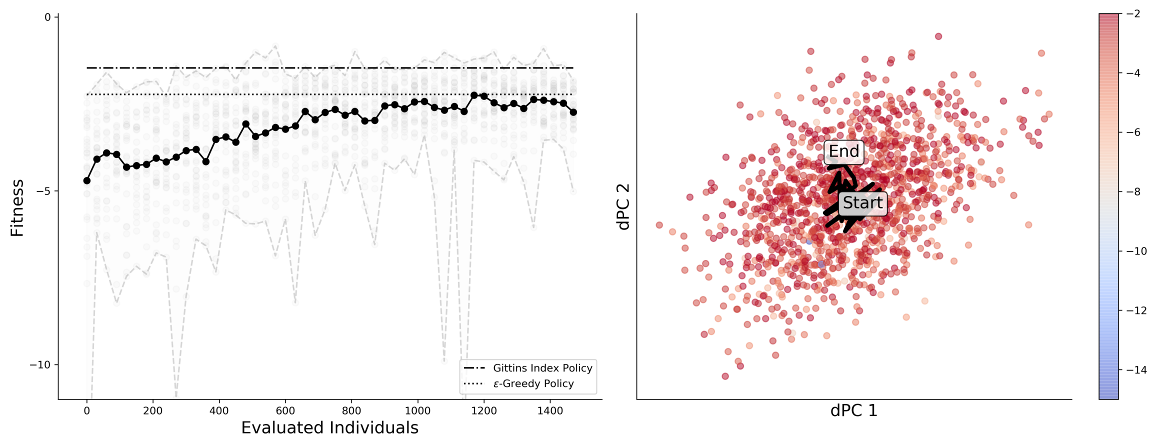

*Figure 7.15: Hyperparameter optimization by the **finite difference gradient** method. Left: Mean fitness among the current population. Horizontal lines show the performance of well-known strategies as baselines. Gray, dashed lines represent the fitness boundaries of the current population. Gray dots signify the fitness of single individuals in the population. Right: Demixed principal component analysis of the hyperparameter setting of every population member evolving the generations. Black thick line indicates the movement of the population mean parametrization over generations*

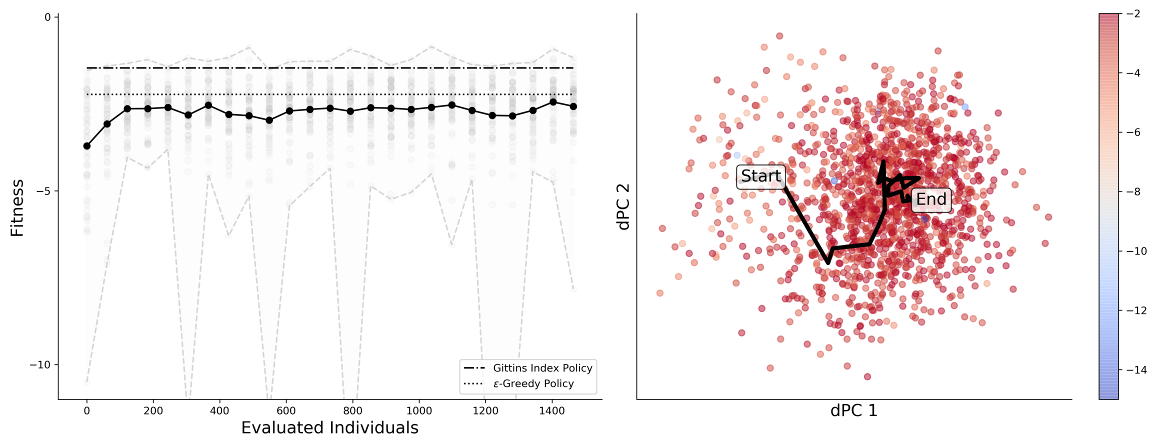

*Figure 7.16: Hyperparameter optimization by the **evolution strategies**. Left: Mean fitness among the current population. Horizontal lines show the performance of well-known strategies as baselines. Gray, dashed lines represent the fitness boundaries of the current population. Gray dots signify the fitness of single individuals in the population. Right: Demixed principal component analysis of the hyperparameter setting of every population member evolving the generations. Black thick line indicates the movement of the population mean parametrization over generations*

7.3 ANN Update Rule

This section displays the results that were obtained with the ANN update rule, as described in Section 3.2.3. In contrast to the other update rules, the main task of learning to learn is not only to find the best set of hyperparameters, but to find a hyperparameter setting that works at all. In contrast to that, the greedy update rule was not influenced by modified hyperparameters to a great extent because it is already designed to represent a proper update rule. With the ANN, the situation is quite different because the hyperparameters really define how the agent learns and updates the weights of the spiking neural network.

In Figure 7.17, a learning to learn optimization is reported, with the cross entropy method used as the optimizer and independent multi-armed bandits as the task family. As one can extract from the fitness evolution on the left panel, the weights of the ANN are changed in such a way that the resulting update rule performs better than random. However, it is not able to approach the performance of well known bandit algorithms.

Figure 7.18 reports on the learning to learn results regarding dependent bandits. Cross entropy was used as the LTL optimizer.

Another learning to learn result concerning a modified version of dependent bandits is given in Figure 7.19. Again cross entropy was used as the underlying LTL optimizer. The considered task family of dependent bandits is modified compared to Section 1.4.1. One important difference was made to the task family to make learning easier for the agent: The reward probability of the first arm p_1 is not sampled uniformly from $[0, 1]$, but instead from the interval $[\mathbf{0}, \mathbf{0.3}] \cup [\mathbf{0.7}, \mathbf{1}]$. As can be seen from the Figure, the spiking neural network equipped with an artificial neural network as a synaptic update rule, is able to handle this special case of multi-armed bandits comparable to existing algorithms, even **outperforms** them. This is only made possible by the use of an LTL optimizer, which requires in this case approximately 2000 individual hyperparameter evaluations to get to a stable solution.

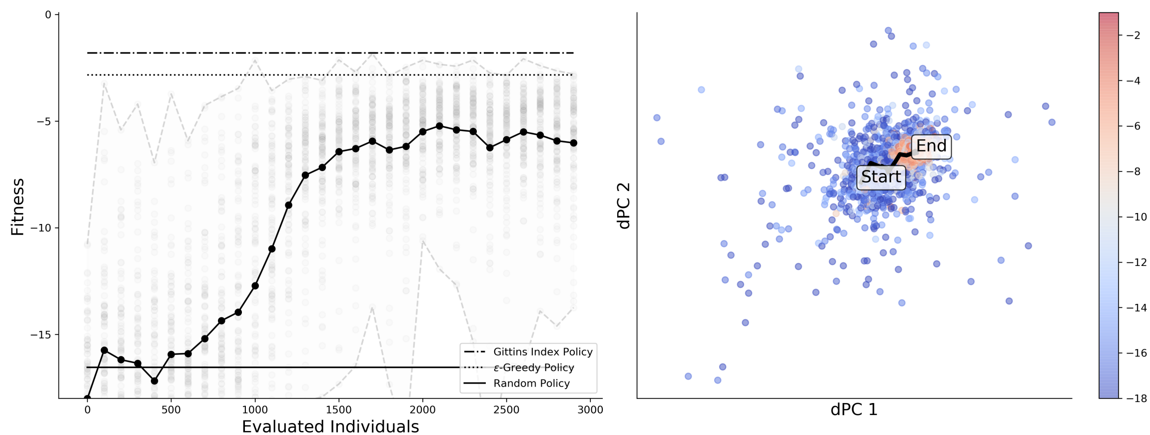

*Figure 7.17: Hyperparameter optimization by the **cross entropy** method. Task family are **independent** bandits. Left: Mean fitness among the current population. Horizontal lines show the performance of well-known strategies as baselines. Gray, dashed lines represent the fitness boundaries of the current population. Gray dots signify the fitness of single individuals in the population. Right: Demixed principal component analysis of the hyperparameter setting of every population member evolving the generations. Black thick line indicates the movement of the population mean parametrization over generations*

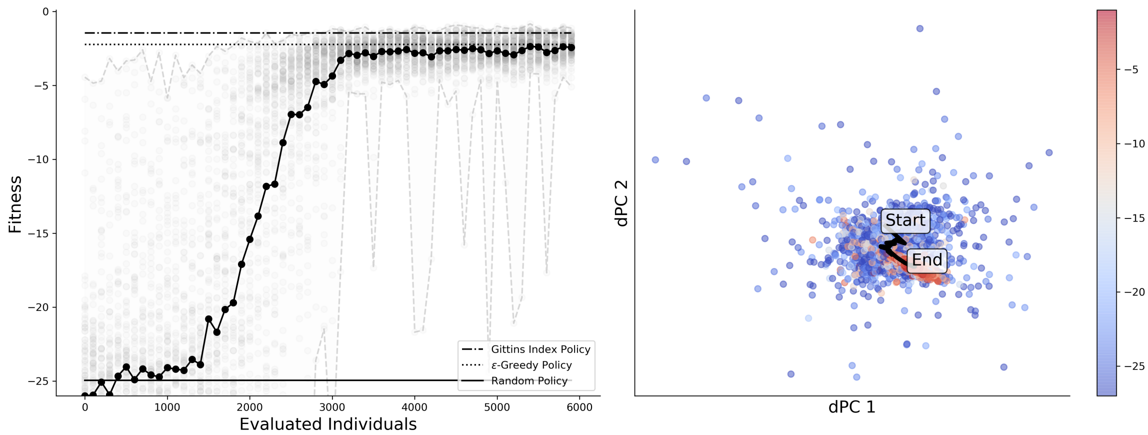

Figure 7.18: Hyperparameter optimization by the ***cross entropy*** *method. Task family are* ***dependent*** *bandits. Left: Mean fitness among the current population. Horizontal lines show the performance of well-known strategies as baselines. Gray, dashed lines represent the fitness boundaries of the current population. Gray dots signify the fitness of single individuals in the population. Right: Demixed principal component analysis of the hyperparameter setting of every population member evolving the generations. Black thick line indicates the movement of the population mean parametrization over generations*

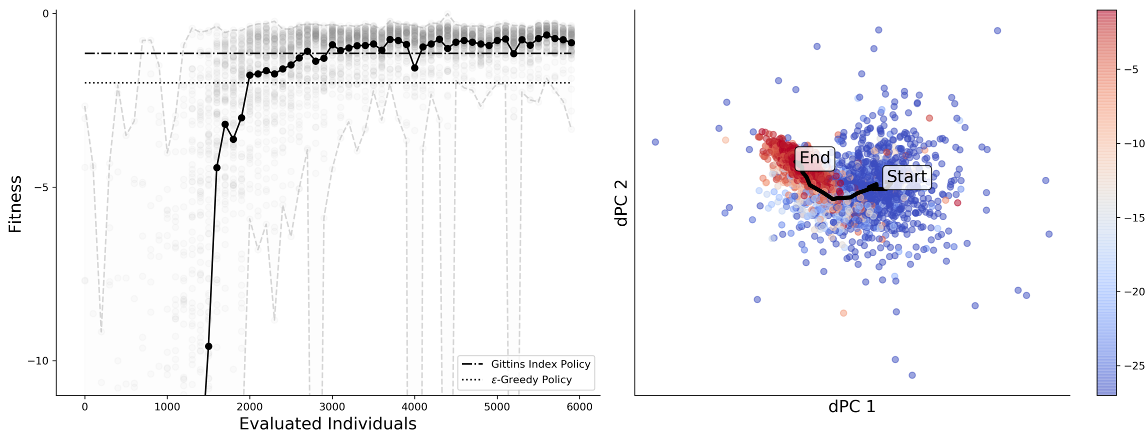

Figure 7.19: Hyperparameter optimization by the ***cross entropy*** *method. Task family are* ***dependent*** *bandits, in the simplified version. Left: Mean fitness among the current population. Horizontal lines show the performance of well-known strategies as baselines. Gray, dashed lines represent the fitness boundaries of the current population. Gray dots signify the fitness of single individuals in the population. Right: Demixed principal component analysis of the hyperparameter setting of every population member evolving the generations. Black thick line indicates the movement of the population mean parametrization over generations*

7.4 Comparison

The result of the learning to learn optimizations are values for the considered hyperparameters. In this section, the hyperparameters were obtained by the various runs are now evaluated. Evaluation is done by examining a sample mean and standard deviation of the expected cumulative regret on multi-armed bandit tasks, either dependent or independent. The sample size is given with 400 instances of multi-armed bandits. The expected cumulative regret is computed after the agent has conducted 100 arm pulls, hence at the end of a task.

Consider the independent multi-armed bandit family. The results of learning to learn optimizations, using various optimizers and update rules for the spiking neural network, are listed in Table 7.3. As one can observe, the spike-based agent is not able to surpass the performance of the ϵ-Greedy policy with $\epsilon = 1\%$. It was found that the hyperparameter settings found by evolution strategies work the best among others. The artificial neural network update rule, as proposed in Section 3.2.3, performs comparably to the software simulation but does not come close to other performances.

Table 7.3: Expected cumulative regret on the indepenedent multi-armed bandit problem. Bold values mark the best performances across well known algorithms or update rules for the spiking agent.

Update Rule	Optimizer (LTL)	Expected Cumulative Regret on **Independent** Bandits
Incremental	Cross Entropy	2.390 ± 3.545
	Parallel Simulated Annealing	2.706 ± 4.070
	Finite Difference Gradient	5.124 ± 4.322
	Evolution Strategies	$\mathbf{2.146 \pm 3.836}$
Greedy	Cross Entropy	2.376 ± 4.226
	Parallel Simulated Annealing	2.837 ± 4.878
	Finite Difference Gradient	2.861 ± 4.677
	Evolution Strategies	$\mathbf{2.187 \pm 3.725}$
ANN	Cross Entropy	$\mathbf{6.021 \pm 3.395}$
Gittins Indices	-	$\mathbf{1.799 \pm 2.449}$
ϵ-Greedy	-	2.838 ± 6.148
Random	-	16.545 ± 11.882

Likewise, Table 7.4 summarizes the same results obtained for dependent bandits. Interestingly, the cross entropy optimizer is now responsible for providing the best performing hyperparameters. Again, the hardware implemented spiking agent fails to reach a better performance level other than the random strategy, comes close however. The expected cumulative regret is in the same range of well performing algorithms in the literature. However, it is observable that the spiking agent exhibits a lower variability than the ϵ-Greedy strategy for instance.

In addition to the previously handled tasks, a modified version of the dependent bandit problem was introduced. One reward probability is in this new version sampled from the interval $p \in [0, 0.3] \cup [0.7, 1]$, the other is then given by $1 - p$ and therefore, the task is easier. The results for optimizing the ANN update rule compared to reference policies are depicted in Table 7.5. In this special case, the spiking agent **outperforms** the Gittins index policy. Note: The Gittins index policy is optimal in the sense of expected cumulative regret but in this evaluation the Gittins indices are computed as if the reward probabilities of the multi-armed bandits were independent.

Table 7.4: Expected cumulative regret on the depenedent multi-armed bandit problem. Bold values mark the best performances across well known algorithms or update rules for the spiking agent.

Update Rule	Optimizer (LTL)	Expected Cumulative Regret on **Dependent** Bandits
Incremental	Cross Entropy	$\mathbf{2.342 \pm 3.472}$
	Parallel Simulated Annealing	2.630 ± 4.044
	Finite Difference Gradient	4.514 ± 4.343
	Evolution Strategies	2.707 ± 3.229
Greedy	Cross Entropy	$\mathbf{2.340 \pm 4.181}$
	Parallel Simulated Annealing	2.448 ± 4.330
	Finite Difference Gradient	2.452 ± 3.708
	Evolution Strategies	2.523 ± 4.330
ANN	Cross Entropy	$\mathbf{1.760 \pm 2.981}$
Gittins Indices	-	$\mathbf{1.469 \pm 2.454}$
ϵ-Greedy	-	2.232 ± 5.131
Random	-	25.159 ± 14.596

Table 7.5: Expected cumulative regret on the dependent multi-armed bandit problem with the independent reward probability sampled from the interval $[0, 0.3] \cup [0.7, 1]$. Bold values mark the best performances across well known algorithms or update rules, composed to handle multi-armed bandits.

Update Rule	Optimizer (LTL)	Expected Cumulative Regret on **Dependent** Bandits with $p_i \in [0, 0.3] \cup [0.7, 1]$
ANN	Cross Entropy	$\mathbf{0.842 \pm 1.441}$
Gittins Indices	-	$\mathbf{1.155 \pm 0.861}$
ϵ-Greedy	-	2.002 ± 4.075
Random	-	35.001 ± 9.208

8 Regret Evaluation

This Chapter visualizes the expected cumulative regret **during** multi-armed bandit tasks. The evolution of expected cumulative regret, depending on the conducted arm pull, allows to see in which parts of the experiment exploration happens and when the agent, or bandit algorithm, focuses on exploitation. In addition, the effects of applying learning to learn, or hyperparameter optimization in this case, become visible at the level of task execution, i.e. one is able to observe how the expected cumulative regrets of the optimized and non-optimized agent diverge as the agent continues to interact with the multi-armed bandit.

For evaluation, a total of 100 bandit tasks is sampled to investigate the agents performance averaged over different tasks of the same family. Task length was set to be $N_P = 100$ and the size of the multi-armed bandit is again given by two arms. Expected cumulative regret, as explained and reasoned by in Section 1.4.2, characterizes how much the agent could have done better up until the considered time step and is the performance measure used here.

Because the effects of hyperparameter optimization is to be demonstrated, the best hyperparameters obtained in Chapter 7 are used here for the optimized agent. For the non-optimized agent, reasonable hyperparameters, were guessed. For comparison, baseline multi-armed bandit algorithms are included in the evaluation.

All figures share the same structure: The expected cumulative regret, averaged over 100 task instances of the same bandit family, is plotted as the multi-armed bandit task proceeds. When the number of arm pulls has reached 100, the task is stopped. Section 8.1 investigates the greedy update rule and Section 8.2 evaluates the incremental update rule.

8.1 Greedy Update Rule

Figure 8.1 shows the expected cumulative regret of the greedy update rule, both optimized and non-optimized, together with baseline algorithms as the multi-armed bandit task evolves. The task family is given with independent multi-armed bandits. Figure 8.2 shows the same with the difference that the task family is given with dependent multi-armed bandits. It is observable that the optimized agent performs by far better than the non-optimized agent.

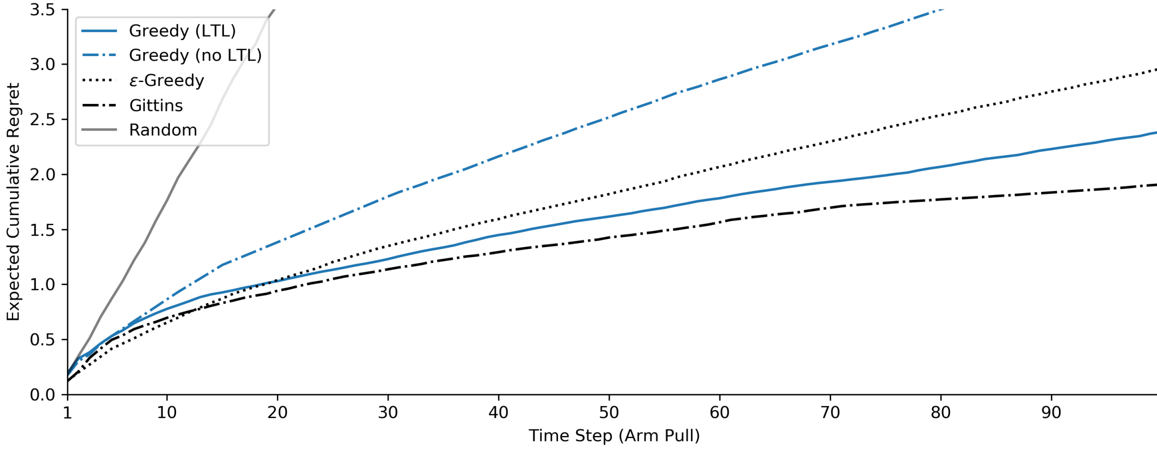

Figure 8.1: Expected cumulative regret depending on bandit task time steps. The task is given by ***independent*** *multi-armed bandits with 2 arms. Averaging happens over 100 different instances. The considered update rule is the greedy update rule, given in Section 3.2.1. Shown is the performance of an update rule* ***with*** *LTL hyperparameter optimization and also* ***without*** *hyperparameter optimization. For comparison, also the expected cumulative regret of baseline policies, performing on the same instances of bandit tasks, are shown.*

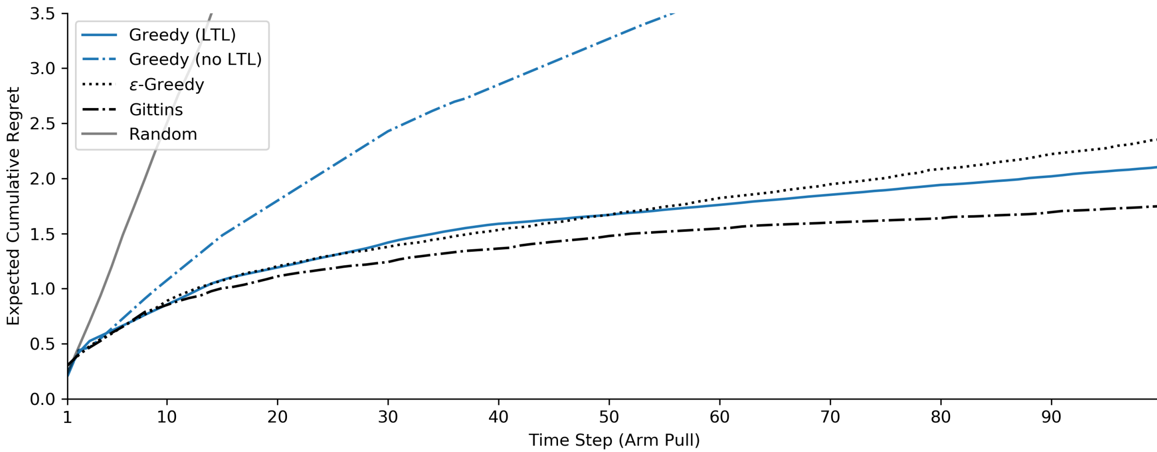

Figure 8.2: Expected cumulative regret depending on bandit task time steps. The task is given by ***dependent*** *multi-armed bandits with 2 arms. Averaging happens over 100 different instances. The considered update rule is the greedy update rule, given in Section 3.2.1. Shown is the performance of an update rule* ***with*** *LTL hyperparameter optimization and also* ***without*** *hyperparameter optimization. For comparison, also the expected cumulative regret of baseline policies, performing on the same instances of bandit tasks, are shown.*

8.2 Incremental Update Rule

Figure 8.3 shows the expected cumulative regret of the greedy update rule, both optimized and non-optimized, together with baseline algorithms as the multi-armed bandit task evolves. The task family is given with independent multi-armed bandits. Figure 8.4 shows the same with the difference that the task family is given with dependent multi-armed bandits. It is observable that the optimized agent performs better than the non-optimized agent, particularly in the independent bandit case.

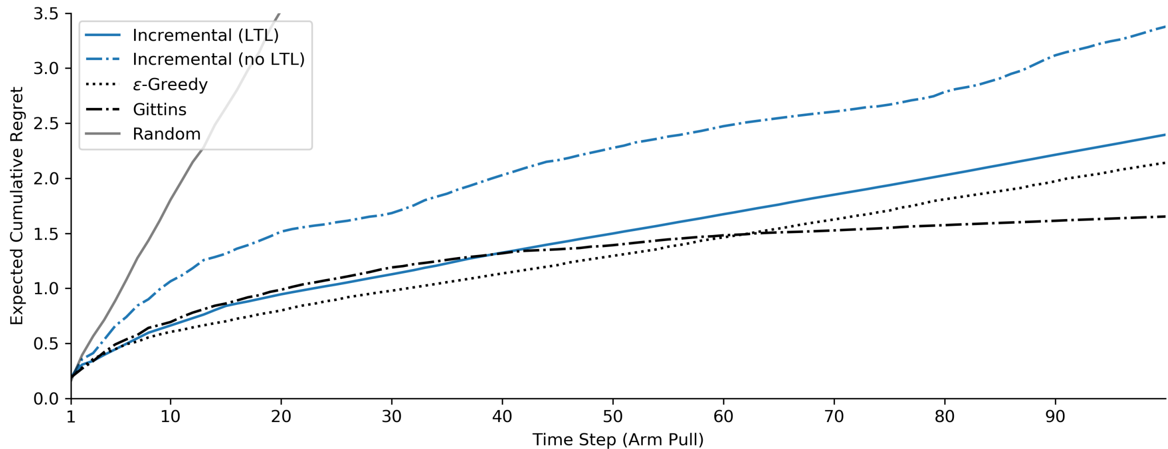

Figure 8.3: Expected cumulative regret depending on bandit task time steps. The task is given by ***independent*** *multi-armed bandits with 2 arms. Averaging happens over 100 different instances. The considered update rule is the incremental update rule, given in Section 3.2.2. Shown is the performance of an update rule* ***with*** *LTL hyperparameter optimization and also* ***without*** *hyperparameter optimization. For comparison, also the expected cumulative regret of baseline policies, performing on the same instances of bandit tasks, are shown.*

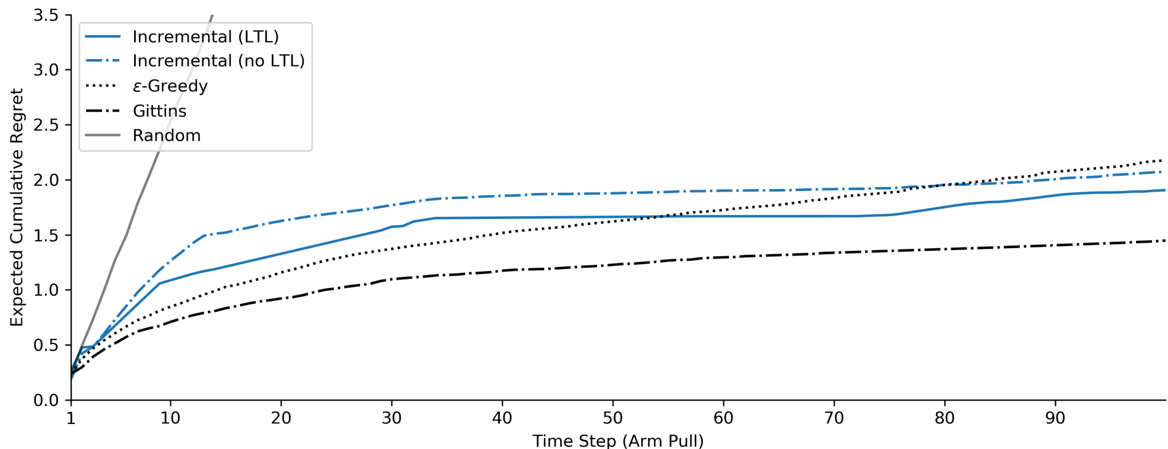

Figure 8.4: Expected cumulative regret depending on bandit task time steps. The task is given by ***dependent*** *multi-armed bandits with 2 arms. Averaging happens over 100 different instances. The considered update rule is the incremental update rule, given in Section 3.2.2. Shown is the performance of an update rule* ***with*** *LTL hyperparameter optimization and also* ***without*** *hyperparameter optimization. For comparison, also the expected cumulative regret of baseline policies, performing on the same instances of bandit tasks, are shown.*

9
Conclusion

In this work it was shown that networks of spiking neurons can be utilized to handle the famous **multi-armed bandit** problem. Proper update rules of the weights in connecting synapses were developed and investigated. In particular, it is to emphasize that a freely parametrizable update rule, in the form of an artificial neural network, was also used.

An essential component was given by the concept of **learning to learn**, in order to find the best set of hyperparameters that work well on the considered task family. Especially in the case of the parametrized update rule, an optimization of the hyperparameters, given by the weights in the artificial neural network, was absolutely necessary in order to obtain a working update rule.

It was also demonstrated that this approach can be implemented on novel neuromorphic prototype hardware, the HICANN-DLS in the second version. As the update rules are customized to the needs of solving such a problem, it was a requirement for the hardware to be flexible enough for programming specific update rules into the hardware. Thus, the plasticity processor of the HICANN-DLS was actively used, also to simulate the multi-armed bandit environment.

Future work may be devoted to develop parametrized update rules that establish learning of more complicated reinforcement environments in spiking neural networks. Indeed, it is conceivable that learning rules arise that work independent of the hardware variabilities, or even more desirable, make use of the diversity of available neurons to accomplish a task faster.

Bibliography

[Aamir et al., 2016] Aamir, S. A., Müller, P., Hartel, A., Schemmel, J., and Meier, K. (2016). A highly tunable 65-nm cmos lif neuron for a large scale neuromorphic system. In *ESSCIRC Conference 2016: 42nd European Solid-State Circuits Conference*, pages 71–74.

[Andrychowicz et al., 2016] Andrychowicz, M., Denil, M., Colmenarejo, S. G., Hoffman, M. W., Pfau, D., Schaul, T., and de Freitas, N. (2016). Learning to learn by gradient descent by gradient descent. *CoRR*, abs/1606.04474.

[Auer et al., 2002] Auer, P., Cesa-Bianchi, N., and Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2):235–256.

[Bahdanau et al., 2014] Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473.

[Brendel et al., 2011] Brendel, W., Romo, R., and Machens, C. K. (2011). Demixed principal component analysis. In *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011. Proceedings of a meeting held 12-14 December 2011, Granada, Spain.*, pages 2654–2662.

[Duan et al., 2016] Duan, Y., Schulman, J., Chen, X., Bartlett, P. L., Sutskever, I., and Abbeel, P. (2016). Rl$ˆ2$: Fast reinforcement learning via slow reinforcement learning. *CoRR*, abs/1611.02779.

[E. Moore, 2006] E. Moore, G. (2006). Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. 11:33 – 35.

[Friedmann, 2013] Friedmann, S. (2013). *A new approach to learning in neuromorphic hardware.* PhD thesis, Heidelberg, Univ., Diss., 2013.

[Friedmann et al., 2017] Friedmann, S., Schemmel, J., Grübl, A., Hartel, A., Hock, M., and Meier, K. (2017). Demonstrating hybrid learning in a flexible neuromorphic hardware system. *IEEE Trans. Biomed. Circuits and Systems*, 11(1):128–142.

[Frostig and Weiss, 1999] Frostig, E. and Weiss, G. (1999). Four proofs of gittins' multiarmed bandit theorem.

[Gerstner, 2008] Gerstner, W. (2008). Spike-response model. *Scholarpedia*, 3(12):1343. revision #91800.

[Gittins and Gittins, 1979] Gittins, A. J. C. and Gittins, J. C. (1979). Bandit processes and dynamic allocation indices. *Journal of the Royal Statistical Society, Series B*, pages 148–177.

[Gutin and Farias, 2016] Gutin, E. and Farias, V. F. (2016). Optimistic gittins indices. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 3153–3161.

[Hassabis et al., 2017] Hassabis, D., Kumaran, D., Summerfield, C., and Botvinick, M. (2017). Neuroscience-inspired artificial intelligence. *Neuron*, 95(2):245–258.

[He et al., 2016] He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778.

[Hebb, 1949] Hebb, D. O. (1949). *The organization of behavior: A neuropsychological theory.* Wiley, New York.

[Hochreiter et al., 2001] Hochreiter, S., Younger, A. S., and Conwell, P. R. (2001). Learning to learn using gradient descent. In Dorffner, G., Bischof, H., and Hornik, K., editors, *Artificial Neural Networks — ICANN 2001*, pages 87–94, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Hodgkin and Huxley, 1952] Hodgkin, A. L. and Huxley, A. F. (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. *J Physiol*, 117(4):500–544. 12991237[pmid].

[Lillicrap et al., 2016] Lillicrap, T. P., Cownden, D., Tweed, D. B., and Akerman, C. J. (2016). Random synaptic feedback weights support error backpropagation for deep learning. *Nature Communications*, 7:13276 EP –. Article.

[Mansour and M. Schain, 2011] Mansour, Y. and M. Schain, M. (2011). Lecture 7: Bayesian approach to mab - gittins index.

[Markram et al., 2011] Markram, H., Meier, K., Lippert, T., Grillner, S., Frackowiak, R., Dehaene, S., Knoll, A., Sompolinsky, H., Verstreken, K., Defelipe, J., Grant, S., Changeux, J.-P., and Saria, A. (2011). Introducing the human brain project. 7:39–42.

[Mead, 1990] Mead, C. (1990). Neuromorphic electronic systems. *Proceedings of the IEEE*, 78(10):1629–1636.

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

[Mnih et al., 2015] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518:529 EP –.

[Peyser et al., 2017] Peyser, A., Sinha, A., Vennemo, S. B., Ippen, T., Jordan, J., Graber, S., Morrison, A., Trensch, G., Fardet, T., Mørk, H., Hahne, J., Schuecker, J., Schmidt, M., Kunkel, S., Dahmen, D., Eppler, J. M., Diaz, S., Terhorst, D., Deepu, R., Weidel, P., Kitayama, I., Mahmoudian, S., Kappel, D., Schulze, M., Appukuttan, S., Schumann, T., Tunç, H. C., Mitchell, J., Hoff, M., Müller, E., Carvalho, M. M., Zajzon, B., and Plesser, H. E. (2017). Nest 2.14.0.

[Rechenberg and Eigen, 1971] Rechenberg, I. and Eigen, M. (1971). Evolutionsstrategie - Optimierung technischer Systeme nach Prinzipien der biologischen Evolution. Dissertation. Published 1973 by Fromman-Holzboog.

[Rubinstein and Kroese, 2004] Rubinstein, R. Y. and Kroese, D. P. (2004). *The Cross Entropy Method: A Unified Approach To Combinatorial Optimization, Monte-carlo Simulation (Information Science and Statistics).* Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[Salimans et al., 2017] Salimans, T., Ho, J., Chen, X., Sidor, S., and Sutskever, I. (2017). Evolution Strategies as a Scalable Alternative to Reinforcement Learning. *ArXiv e-prints.*

[Schemmel et al., 2010] Schemmel, J., Briiderle, D., Griibl, A., Hock, M., Meier, K., and Millner, S. (2010). A wafer-scale neuromorphic hardware system for large-scale neural modeling. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pages 1947–1950.

[Schmidhuber, 1987] Schmidhuber, J. (1987). Evolutionary principles in self-referential learning, or on learning how to learn: the meta-meta-... hook. Diploma thesis, Inst. f. Inf., Tech. Univ. Munich. http://www.idsia.ch/˜juergen/diploma.html.

[Schuman et al., 2017] Schuman, C. D., Potok, T. E., Patton, R. M., Birdwell, J. D., Dean, M. E., Rose, G. S., and Plank, J. S. (2017). A survey of neuromorphic computing and neural networks in hardware. *CoRR*, abs/1705.06963.

[Stöckel, 2017] Stöckel, D. (2017). Exploring collective neural dynamics under synaptic plasticity. Masterarbeit, Universität Heidelberg.

[Stradmann, 2016] Stradmann, Y. (2016). Characterization and calibration of a mixed-signal leaky integrate and fire neuron on hicann-dls. Bachelorarbeit, Universität Heidelberg.

[Wang et al., 2016] Wang, J. X., Kurth-Nelson, Z., Tirumala, D., Soyer, H., Leibo, J. Z., Munos, R., Blundell, C., Kumaran, D., and Botvinick, M. (2016). Learning to reinforcement learn. *CoRR*, abs/1611.05763.

[Wierstra et al., 2008] Wierstra, D., Schaul, T., Peters, J., and Schmidhuber, J. (2008). Natural evolution strategies. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, pages 3381–3387.

[Wunderlich, 2016] Wunderlich, T. (2016). Synaptic calibration on the hicann-dls neuromorphic chip. Bachelor, Heidelberg University.