Fortgeschrittenenpraktikum

Release 2025-12-18

Electronic Vision(s) Group

Dec 18, 2025

CONTENTS

Welcome to this lab course! You will learn about BrainScaleS-2, a neuromorphic platform developed in Heidelberg, and gain hands-on experience with the tools required to conduct experiments on this platform. For that, we will familiarize you with concepts in neuromorphic computing and how they originated from neuroscience. We start with an overview of the biological behavior of neurons and synapses and present models from literature that are useful in neuromorphic computing. We then introduce the BrainScaleS-2 system and the software module used to interface with it. In the experiment part, you will first investigate the behavior of silicon neurons using Lu.i, an analog electronic implementation of a simple neuron model. Following this, you will learn how to set up and run experiments on BrainScaleS-2 and apply your knowledge in configuring a feedforward network and solving sudoku using neural networks.

Throughout this course, we will use the Python programming language to interact with the neuromorphic hardware and analyze the resulting data. To guide your experiments, we provide Jupyter notebooks containing pre-written scripts. These scripts serve as starting points and will require small modifications or extensions to fulfill the tasks at hand. We will demonstrate the fundamental operations of the hardware and the necessary Python packages using clear examples, making it easy to follow along. However, a basic understanding of Python programming is highly beneficial for a smooth experience. If you're unfamiliar with Python, we recommend starting with the introductory tutorial provided here: http://www.physi.uni-heidelberg.de/Einrichtungen/AP/Python.php.

In addition to the technical aspects, this course aims to provide the necessary neuroscientific background for the lab. Nonetheless, consulting some of the referenced literature is highly recommended. For example, Gerstner (2014) offers a comprehensive overview of various neuron models, including the leaky integrate-and-fire model that you will use in this experiment. Additionally, we encourage you to read the following material before you start the lab course to ensure you are well-prepared.

CHAPTER

ONE

BIOLOGICAL BACKGROUND

Before digging into the field of neuromorphic computing, it is essential to understand concepts from the central nervous system, the source of inspiration behind this field. The central nervous system is a highly complex system responsible for many functions in diverse life forms. These functions include thinking, processing sensory data, and coordinating movements. The field of neuroscience has explored the nervous system from different perspectives, including anatomy, behavior, and modeling. In this section, we highlight the major components in the nervous system, and we mathematically describe their behavior. Specifically, we deal with the concept of an ideal spiking neuron (Gerstner (2014)); the name will become clear after the introduction of its basic behavior. Then, we present models from literature that mimic the neural behavior of interest.

1.1 Neurons in Biological Systems

The major building blocks of the nervous system are neurons. Neurons come in a variety of morphologies or anatomical structures, but the basic structure can be described as consisting of three major parts which serve different functions. The cell body of the neuron is called the soma, which is the central processing unit of the neuron. The dendrites are extensions from the soma that carry input signals to the neuron. The axon is also an extension from the soma that carries output signals to other neurons. Neurons are connected through synapses, where transmission of signals takes place from a presynaptic neuron to a postsynaptic neuron. Most synapses are chemical, in which signalling molecules called neurotransmitters are released by the presynaptic neuron.

The electrical signal of interest in the nervous system is the difference in the electrical potential between the interior of the neuron and the exterior of the neuron, referred to as the membrane potential. This physical quantity is used to describe the state of a single neuron and the communication between neurons. Originally, a presynaptic neuron receives inputs from synapses through the neurotransmitters. These neurotransmitters can alter the membrane potential of the neuron either by increasing the potential (excitatory) or decreasing the potential (inhibitory). Depending on the type and strength of the inputs, the membrane potential exceeds a threshold, and the neuron generates a signal. This signal is referred to as an “action potential” or a “spike”. A spike travels along the axon of the presynaptic neuron until it reaches the axon terminals where neurotransmitters are released to different synapses. These neurotransmitters initiate specific biophysical mechanisms that induce postsynaptic currents in the postsynaptic neurons. The postsynaptic currents can thus alter the membrane potential of the postsynaptic neurons. The resulting signals influencing the membrane potential of the postsynaptic neurons are referred to as postsynaptic potentials.

1.2 Mathematical Formulation of Neuronal Behavior

We are interested in mathematically formulating the state of a single neuron and the communication between neurons. For that, let's have a closer look at the physical behavior of a neuron, especially at the membrane potential of the soma. In the absence of any stimulation, we observe a negative resting membrane potential of approximately -65 mV, a strongly negative polarized state. A stimulation happening through the dendrites changes the membrane potential. This can be formalized as follows: At a given time t the membrane potential of neuron i is $u_i(t)$. When the neuron is not stimulated, for all times smaller than 0, the potential is $u_i(t < 0) = u_{\text{rest}}$. At $t = 0$, a stimulation is triggered by a presynaptic neuron j. We define the postsynaptic potential (PSP) as

$$\epsilon_{ij}(t) := u_i(t) - u_{\text{rest}}.$$

To indicate the direction of change we define further

$$\epsilon_{ij} = u_i(t) - u_{\text{rest}} \begin{cases} > 0 & \text{excitatory PSP} \\ < 0 & \text{inhibitory PSP.} \end{cases}$$

Typically, the membrane potential $u_i(t)$ does not stay at the same level. In the absence of a stimulation, it decays to the resting potential u_{rest}.

Now let's assume there are 2 presynaptic neurons $j = 1, 2$ to the postsynaptic neuron i. The presynaptic neurons stimulate the postsynaptic neuron, and PSPs are induced in the postsynaptic neuron at respective times $t_j^{(m)}$, where m states the m-th signal peak coming from the j-th neuron. At each incoming signal peak, charges are deposited at the membrane and the potential rises. If many excitatory inputs arrive at neuron i, the membrane potential u_i can reach a critical value, triggering the neuron to fire. A sharp rise of the membrane potential (reaching a voltage of about $+100mV$) is observed, and the potential rise propagates down the axon. The pulse is referred to as a "spike" and the neuron is said to have "spiked".

After such a pulse, the membrane potential drops below the resting potential, and later it returns to the resting state. This final phase is called hyperpolarization or spike-afterpotential. During the hyperpolarization, the neuron can hardly be stimulated.

The trajectory of such an event can be observed in Fig. 1.1.

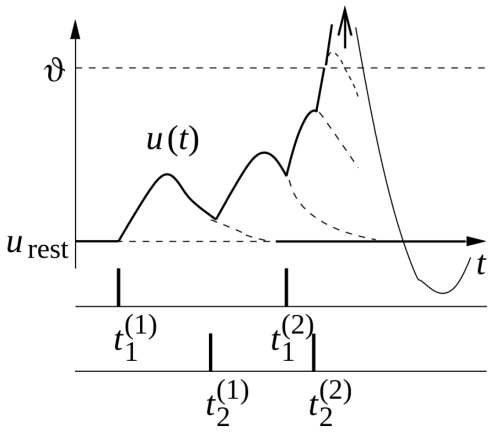

Fig. 1.1: Membrane potential u of a postsynaptic neuron receiving inputs from two presynaptic neurons ($j = 1, 2$). Each presynaptic spike generates an excitatory postsynaptic potential, which contributes to the cumulative depolarization of the membrane potential. Dotted lines indicate the hypothetical trajectory of the membrane potential in the absence of additional stimulation. When u reaches the threshold ϑ, an action potential is triggered, indicated by a large positive pulse-like excursion (arrow). This pulse exceeds the graph's bounds and is followed by a reset, where the membrane potential returns to a value below the resting potential u_{rest}. Taken from Gerstner et al. 2014, Chapter 1.2.

In case the threshold is not reached, i.e., when only a few presynaptic spikes occur, the membrane potential behaves as the sum of the individual PSPs:

$$u_i(t) \approx \left[\sum_j \sum_f \epsilon_{ij} \left(t - t_j^{(f)} \right) \right] + u_{\text{rest}}. \tag{1.1}$$

This is also called PSP-Stacking.

In the next step, we want to use the equations to derive a concrete model of the neuron, so that it is possible to implement on a neuromorphic substrate.

1.3 Modeling Neuronal Behavior

The neuron is driven by biochemical and bioelectrical principles, involving various interactions (for a short overview see Purves (2009)). We aim to find a suitable model that describes the basic behavior of neurons without necessarily incorporating all biological aspects. The goal is to obtain a somewhat similar behavior as described in the previous section.

1.3.1 Leaky integrate-and-fire (LIF) model

In this model, the following constraints are applied. First, the spikes always have the same shape, i.e., the shape does not contain any information. Instead, all information will be coded in the timings of the spikes. Second, the membrane potential processes the information. Another feature that needs to be modeled is when $u_i(t)$ reaches a critical voltage threshold ϑ, a spike has to be initiated, causing this neuron "to fire" at this time $t_i^{(m)}$. Such a model was proposed by Lapique (1907) and is called leaky integrate-and-fire (LIF).

Essentially, the cell membrane acts as a good insulator. When a current $I(t)$ arrives at the membrane, additional charge is deposited. This behavior is similar to that of a capacitor, so we abstract a cell membrane by a capacitor with capacitance C. As previously discussed, the membrane potential decays over time; therefore, the charge leaks. This can be modeled by a resistance R. In addition, we require a source to define the resting potential.

This completes the basic circuitry for a neuron model, as depicted in Fig. 1.2.

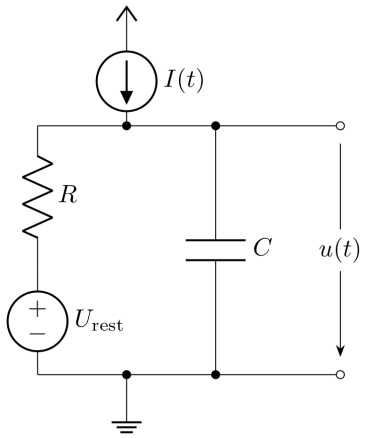

Fig. 1.2: Circuit diagram of the LIF neuron. The neuron's membrane potential is modeled by the voltage u across a capacitor C, which is connected via the resistor R to a constant voltage source U_{rest}, representing the resting potential. Stimulation is provided by a time-dependent current source $I(t)$.

If we analyze the electrical circuit, we can find a differential equation describing the behavior of the capacitor voltage:

$$\tau_m \frac{\mathrm{d}u_i(t)}{\mathrm{d}t} = -\left[u_i(t) - u_{\text{rest}}\right] + R \cdot I(t) \tag{1.2}$$

Here, $\tau_m = R \cdot C$ is also called the *membrane time constant*, and the index i refers to the i-th neuron. $I(t)$ in this equation represents a time dependent current flow onto (excitatory) or away from (inhibitory) the membrane. In neuroscience, this equation, which describes a leaky integrator, is the equation of a passive neuron. Currently, it fulfills the requirement of integrating incoming spikes (see equation (1.1)), but it lacks an active part in the form of a spiking mechanism. For the basic model, we define a threshold value ϑ. When this value is crossed from below, an additional circuit emits a voltage spike that propagates to all connected neurons. At the same time, the potential of the capacitance is clamped to a defined value u_{reset} and kept at this level for the *refractory period* τ_{r}.

1.3.2 Adaptive exponential (AdEx) model

The LIF-model captures only some basic dynamics of a real neuron. Therefore, various models were proposed to include more interesting dynamics. Brette (2005) presented an improved LIF-model. The main additions are an exponential and an adaptation term. These terms are required to process high frequent synaptic input and model the spike initiation accurately. Further, a recovery variable is introduced to capture adaptation and resonance properties.

$$\tau_m \frac{\mathrm{d}u_i}{\mathrm{d}t} = -\left(u_i - u_{\text{rest}}\right) + R\left(I_{\text{stim}} - I_{\text{ad}}\right) + \Delta_{\text{T}} \exp\left(\frac{u_i - u_T}{\Delta_{\text{T}}}\right) \tag{1.3}$$

$$\tau_w \frac{\mathrm{d}I_{\text{ad}}}{\mathrm{d}t} = a\left(u_i - u_{\text{rest}}\right) - I_{\text{ad}} \tag{1.4}$$

The equation above is arranged in such a way that the extension to the LIF-equation (1.2) is easily visible. As new terms are introduced: I_{ad} which is the adaptation current, Δ_{T} is the threshold slope factor, u_{T} is the effective threshold potential. Further, a second equation is introduced to describe the dynamics of the adaptation. For the adaptation term, a conductance a and a time constant for the adaptation current τ_w are required. Another modification is the reset condition. While previously only the membrane potential was set to a reset potential, now the adaptation current has to be modified as well. The action of a spike is now described as

$$\text{if } u_i > \vartheta \text{ then } \begin{cases} u_i \to u_{\text{reset}} \\ I_{\text{ad}} \to I_{\text{ad}} + b. \end{cases}$$

Here, an additional variable is used, the spike-triggered adaptation b.

This model is called AdEx due to the adaptation and exponential terms. Depending on the parametrization, it can describe different neuron types and model more sophisticated behaviors observed in biological neurons.

1.4 Synaptic Plasticity

The modification in the strength and structure of the synapses is referred to as synaptic plasticity. This modification affects the synaptic signal that is received by a postsynaptic neuron. Neuroscientists have long thought of synaptic plasticity as the basic mechanism underlying learning and memory, which explains the interest behind studying and modeling this mechanism.

In computational neuroscience, the strength and effect of a synapse is modeled by a synaptic weight w. Positive synaptic weights generally indicate excitatory synapses while negative synaptic weights indicate inhibitory synapses. The higher the absolute value of the synaptic weight, the stronger the synaptic current affecting the postsynaptic neuron. A plasticity rule is a set of mathematical expressions that govern the relationship between neuronal activity and synaptic plasticity. There are two types of plasticity rules. Phenomenological models are simple expressions that do not explicitly model the physiology of the synapse. Rather, they are based on an immediate input-output relationship between neuronal activity and synaptic plasticity. On the other hand, biophysical models incorporate further details on the complex biophysical phenomena governing synaptic plasticity. In this lab course, we focus on phenomenological models.

CHAPTER

TWO

NEUROMORPHIC COMPUTING

The energy-efficiency of the computation taking place in the nervous system has long inspired researchers and engineers in building similarly efficient machines simply by adopting architectural as well as dynamical properties of the biological counterpart. There exists a whole spectrum of neuromorphic systems with some relying on fully digital logic – then mostly focusing on asynchronous and parallel communication – and others utilizing analog circuits to further efficiency gains. These different systems all focus on different aspects and target a variety of applications. The following sections introduce some of the computing paradigms and platforms as well as a range of use cases ranging from neuroscience research to machine intelligence.

2.1 Analog computation

Neuronal computation is fundamentally based on the conservation of charge: Neurons accumulate and integrate stimuli over space (i.e., their dendrites) and time to perform, e.g., spatio-temporal summation and coincidence detection. This is a stark contrast to today's processors, which operate on mostly binary digital representations. Here, any state change or computation involves charging or discharging of multiple digital states and logic circuits are typically composed of a plethora of transistors. This observation led Carver Mead (1990) to envision *neuromorphic electronic systems* based on simple analog circuits performing neuron-like analog computation.

2.2 Computation through structure

The specific computation implemented by a neural circuit is inherently defined by the individual cell properties and – more centrally - the network structure and synaptic strengths. This effectively results in a *collocation of processing and memory resources* and, again, represents a contrast to current processors. The latter almost all implement the von Neumann architecture, where program instructions as well as the data to be processed are stored in memory separate from the processing unit, inherently resulting in a frequent expensive transfer of data through the infamous *von Neumann bottleneck*.

The collocation of memory and computing resources observed in neural tissue sparked interest in *in-memory computation*, where simple computational units perform calculations on locally stored operands/parameters as well as a stream of input data. In-memory computation typically emerges in the context of machine learning accelerators where vector-matrix multiplications dominate the overall workload and at the same time can be straight-forwardly rendered to an array of synapse-like multiply-accumulate units. Importantly, in-memory computation differs from traditional sequential processing in that it feeds off its high intrinsic parallelism.

2.3 Event-based communication

The analog, time-continuous computation in a neuron is contrasted by a rather binary, event-based communication: Upon threshold crossing, neurons generate a spike-like action potential, which propagates along the axon and is finally relayed to other neurons. These spikes mark singular points in time and the neuron remains silent otherwise, not unlike an intrinsic zero suppression.

Spikes can encode information in their sole presence, their absolute or relative timing, or their rates averaged over time or multiple neurons. Especially the limit of only few spikes can realize an energy-efficient communication scheme.

With its event-based communication the nervous system, furthermore, realizes an asynchronous operation. Avoiding the need for global synchronization can optimize the energy footprint and speed of processing elements but introduces challenges with respect to reliability. Here, the nervous system appears to have found a solution resilient to temporal jitter or even spike loss and can thus inspire further research.

CHAPTER

THREE

THE BRAINSCALES-2 SYSTEM

Note: This chapter uses Billaudelle, S. (2022). From Transistors to Learning Systems: Circuits and Algorithms for Brain-inspired Computing (Doctoral dissertation, Heidelberg University). It provides an overview of the BrainScaleS-2 system required to successfully complete this lab course.

BrainScaleS-2 is a spiking, accelerated mixed-signal neuromorphic platform: It captures the dynamics of neurons and synapses in mostly analog integrated circuits and augments this physical emulation of the underlying differential equations by digital periphery. The analog circuits exploit the native scale of capacitances and conductances in microelectronic circuits to accelerate the neural dynamics by a factor of 1000. Typical time constants of multiple milliseconds are therefore rendered at the scale of microseconds. The speed-up is relevant for applications in the fields of machine intelligence, brain modelling, and accelerated robotics, which BrainScaleS-2 aims to cover. The emulated dynamics are flexibly configurable to target vastly different neuron models and modes of operation. They support simple, non-time-continuous artificial neurons as well as biologically realistic dynamics in structured multi-compartmental cells.

BrainScaleS-2 expands on the static emulation of the neural dynamics and provides the means for implementing on-chip plasticity rules and advanced learning experiments. For that purpose, it features custom embedded processor cores tailored for an efficient interaction with accelerated network dynamics. These on-chip controllers can be freely programmed to dynamically reconfigure all aspects of the on-chip circuits, facilitating the implementation of learning rules and closed-loop cybernetic experiments directly on the chip itself.

A photograph of the chip is shown in Fig. 3.1.

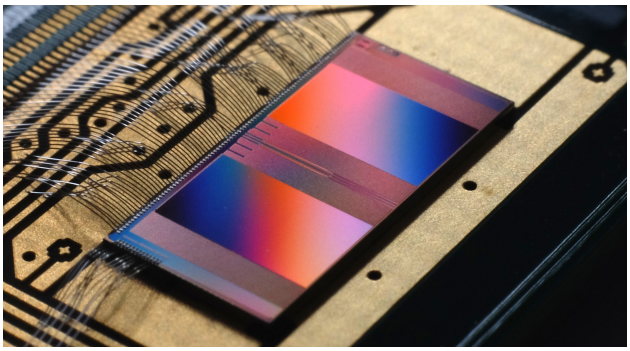

Fig. 3.1: Photograph of a BrainScaleS-2 chip mounted on its carrier board. The chip is bonded to the board for electrical connectivity.

3.1 Overview

The BrainScaleS-2 *ASIC* is centered around an analog neuromorphic core housing the circuits emulating the neuronal and synaptic dynamics. This conglomerate of mostly analog full-custom circuits operates in continuous time, asynchronously to its periphery. These dynamics can be configured and tuned through local digital memory *SRAM*. The synapse matrices, basically representing large in-memory compute arrays, are a prime example of this architecture.

The *ASIC* is optimized for event-based communication and to that end provides an on-chip spike router. Spikes can be directly injected into the *ASIC* from the "outside", e.g. a host computer or – in a multi-chip system – other *ASICs* or potentially event-based sensors. In these cases, the events are relayed by an external controller chip (i.e., an *FPGA*) acting as a real-time communication interface. Output spikes can be also recorded to host or injected into external actuators in robotics applications. In addition, background spike sources are directly provided on the *ASIC* itself. They can be configured to emit regular or Poisson spike trains at a total bandwidth of 8 × 125 MEvent/s.

A block-level diagram of the BrainScaleS-2 system is shown in Fig. 3.2.

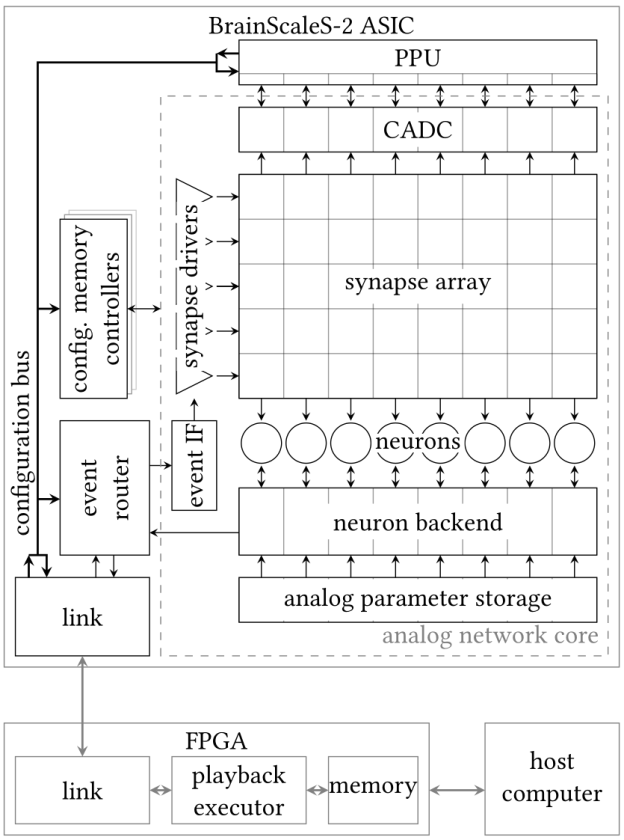

Fig. 3.2: Block-level diagram of a BrainScaleS-2 system, including the *ASIC* itself as well as an *FPGA* managing the communication to the host system using standard Ethernet networking.

3.2 The analog neuromorphic core

The analog core is segmented into multiple specialized subsystems for both the neuromorphic emulation itself and supporting circuitry. It is primarily composed of neurons and synapses circuits. The following paragraphs discuss these individual circuits and in that process roughly follow the path of an event from the synapse circuits to a neuron's membrane.

3.2.1 Synapses

The synapse circuits take on one of the most central roles in the analog neuromorphic core. They perform the actual in-memory computation, primarily define the network topology, and feature sensor circuits facilitating the on-chip implementation of (temporal) correlation-based, biology-inspired learning rules. The synaptic efficacy controls the conversion of events or spikes into analog current pulses using a 6 bit *DAC* in the following way: The magnitude of the synaptic current is scaled by the respective weight value, w, which is locally stored in 6 bit of full-custom *SRAM*. Thus, the amplitude of the resulting current is the product of the synapse circuit's current generated by a *DAC*, and a globally set bias current of 0 to 1 µA. The sign of a synapse, i.e. its either excitatory or inhibitory nature, is set as a shared property between a group of synapses.

3.2.2 Neurons

A BrainScaleS-2 *ASIC* houses 512 neuron circuits, distributed over two horizontal rows of 256 neurons each. These rows are also divided into two quadrants with 128 neurons each. The neurons implement the *AdEx*. model, but can be reduced to the *LIF*. model for the realization of *SNNs*. Their parameters can be also modified to realize *ANNs*. These dynamics are combined primarily with synaptic currents having exponentially decaying kernels. Multiple instances can be merged to form larger logical neurons to, i.e., bundle the synaptic resources of the individual units and thus trade the total number of neurons for an increased fan-in. This process also allows to connect compartments with finite and configurable conductances to form multi-compartment emulations of morphologically more realistic neurons as well as dendritic computations.

The neuron circuit and its backend can be digitally configured via a total of 64 bit of local *SRAM*. Additionally, the analog components can be tuned through 8 reference voltages and 16 current parameters per neuron. They allow to set the circuits up for a wide range of target dynamics and at the same time allow the equalization of production-induced variations (i.e. fixed-pattern noise) between individual instances. Each analog parameter can be freely programmed as a digital 10-bit value and converted to analog signals using *DACs*.

3.2.3 Analog I/O

BrainScaleS-2 allows to route many of these potentials across the *ASIC* and to apply them to one of two analog *IO* pads, making them available to external measurement equipment or reference potentials. Furthermore, different *ADCs* can digitize these signals directly on the chip itself; two of these *ADCs* are fast, used for single neuron investigation, while 1024 are slow, referred to as *CADCs*, and used for sampling in parallel across a group of neurons. These capabilities do not only facilitate lower level measurements and the commissioning of the neuromorphic circuits but are also crucial to more directly interact with the system and bridge the gap between the analog and digital domains to, e.g., implement advanced plasticity rules.

Parallel analog-to-digital conversion

The analog emulation of neural states evolves asynchronously and fully parallel. While quite apparent for the 512 neuronal membrane potentials, this high-degree of parallelism becomes even more precarious for reading out the analog correlation sensors located in each synapse circuit which is required to execute plasticity calculations in the PPU (see next paragraph), using the digitized correlation values. BrainScaleS-2, hence, features massively parallel *CADCs* to still be able to incorporate these states into plasticity calculations.

The two *CADCs* – one per vertical half of the *ASIC* – each feature 512 channels, two per column of synapses. The *CADCs* digitizes the parallel channels at a resolution of 8 bit and a maximum sampling rate of 1.85 MSample/s per channel.

3.3 On-chip plasticity

A significant body of work goes beyond static neural networks and focus on different learning paradigms or closed-loop setups where a (simulated) actor interacts with an artificial environment. To facilitate this, BrainScaleS-2 augments the accelerated dynamics of the analog neuromorphic core with custom embedded processors, one per vertical half of the *ASIC*. The *PPUs* have full read and write access to all on-chip components and can thus read out and operate on, e.g., the neuronal firing rates and, in addition, dynamically reparameterize all of the *ASICs* subsystems.

The general purpose parts are accompanied by custom vector extensions. They are capable of calculating with 128 8-bit or 64 16-bit integers in parallel. They allow to efficiently perform calculations based on vectorized data acquired from the analog neuromorphic core by the column *ADCs*, e.g., for weight update calculations. Additionally, the vector units directly attach to the synapse arrays' memory interfaces and can thus access the synaptic weights and address labels in a row-wise fashion to read current weight values and write the new weight values.

3.4 System integration and experiment flow

The BrainScaleS-2 *ASIC* relies on a stack of peripheral circuitry and software for its functionality and user interface. Hardware setups are available in different form factors, from laboratory setups to smaller and portable setups as well as multi-chip systems. All of them, however, share a very similar overall architecture: The neuromorphic *ASIC* is supported by a set of *PCBs* providing necessary power supplies, analog references, and *IO* circuits.

For running experiments on the chip, the chip itself is interfaced in real time via an *FPGA*. The user code executed on the experiment host compiles programs which are interpreted and executed by a state machine on the *FPGA*. These programs support simple instructions to read from and write to memory locations on both the BrainScaleS-2 chip and its periphery, and provide dedicated commands for the injection of spike events.

This program flow and hardware abstraction is encapsulated by the BrainScaleS-2 software stack. The software stack consists of multiple layers from communication protocols to high-level experiment descriptions. A hardware abstraction layer represents each of the system's components and subsystems as a configuration container. However, users typically interface with BrainScaleS-2 using high-level modules written in Python, namely PyNN and Pytorch.

CHAPTER

FOUR

INTRODUCTION TO PYNN

The neuron circuits have a lot of tunable parameters. But how do we set their values? For this the module PyNN can be used. It is a language that allows you to form groups of neurons and to connect them in different ways to each other. Also, the parameters of individual neurons can be varied and the resulting dynamic can be observed.

In the following, we want to build a simple network in which a neuron is stimulated by a group of five neurons, as depicted in Fig. 4.1.

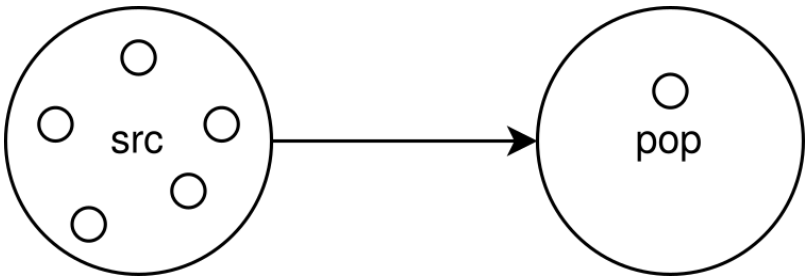

Fig. 4.1: Illustration of a source population (src) comprising five neurons connected to a target population (pop) consisting of a single neuron.

Note: This print-friendly version of the instructions only lists the most central functions and code blocks. The interactive variant provides additional helpers.

Groups of neurons are called populations. Such a population is instantiated by setting the number of neurons it should contain, the cell type of these neurons and the values of the cell parameters. The cell type of our artificial neurons is called `HXNeuron`. Its parameters are not expressed in the same units as for biological neurons, but in hardware units. These two systems of units are not directly related. Also within the hardware units there is no general translation to physical voltages and currents. Furthermore, these values can have different meanings for the different parameters on a chip, for example, a threshold voltage of 300 may be higher than a leakage voltage of 400. Especially in the comparison between chips, even though they are designed completely identical, the actual measured values can vary slightly.

```
# define the neuron parameters of the population
n_neurons = 1
neuron_parameters = {                           # range
    "leak_v_leak": 400,                         # (300-1000)
    "leak_i_bias": 200,                         # (0-1022)
    "threshold_v_threshold": 400,               # (0-600)
    "threshold_enable": True,                   #
    "refractory_period_refractory_time": 100,   # (0-255)
    "reset_v_reset": 300,                       # (300-1000)
    "reset_i_bias": 1000,                       # (0-1022)
```

(continues on next page)

(continued from previous page)

```
    "membrane_capacitance_capacitance": 63      # (0-63)
}

neuron_type = pynn.cells.HXNeuron(**neuron_parameters)

# save the configured neuron in the population 'pop'
pop = pynn.Population(n_neurons, neuron_type)
```

The spikes of all neurons that have been stored in populations can be recorded. Furthermore, it is also possible to record the membrane potential of a single neuron. Consequently, for this purpose the population must have a size of one.

Warning: We can only record one neuron at the time. Make sure to use populations of size one or use views to select a single neuron, see. PyNN documentation.

```
# record spikes and membrane potential 'v' of the neuron in the
# population 'pop'
pop.record(["spikes", "v"])
```

Different populations can be connected by so-called projections. For this, firstly it must be specified which is the pre-synaptic (source) and which the post-synaptic (receiver) population. Furthermore, the way in which the neurons within the populations are exactly connected to each other is specified, e.g., all neurons are connected, or only a certain percentage of the neurons are connected to each other. In addition, the synaptic weight which describes the strength of the connection and the synapse type are specified. This can either be excitatory, meaning that the membrane voltage increases in case of stimulation, or it is inhibitory, which causes the membrane voltage to decrease.

```
# create a source population that generates spikes at given times
spike_times = [0.01, 0.03, 0.05, 0.07, 0.09]
src = pynn.Population(5, pynn.cells.SpikeSourceArray(spike_times=spike_times))

# define a synapse and its weight
synapse_weight = 63
synapse = pynn.synapses.StaticSynapse(weight=synapse_weight)

# connect the pre-synaptic population 'src' to the post-synaptic
# neuron in 'pop'
pynn.Projection(src, pop, pynn.AllToAllConnector(),
                synapse_type=synapse, receptor_type="excitatory")
```

The created network of populations and projections can now be emulated for a selected time.

```
# the duration is given in ms,
# this is in the hardware domain, not in the biological
# (the hardware is faster by a factor of approx. 1000
# compared to biology)
duration = 0.1
pynn.run(duration)
```

Thereafter, the recorded behavior of the neurons can be read out.

```
# read out the spikes of the neuron in 'pop'
spiketrain = pop.get_data("spikes").segments[0].spiketrains[0]
print(f"The neuron spiked {len(spiketrain)} times.")
print(f"The spike times were: {spiketrain}")
```

(continues on next page)

(continued from previous page)

```
# plot its membrane potential
mem_v = pop.get_data("v").segments[0].irregularlysampledsignals[0]

import matplotlib.pyplot as plt
%matplotlib inline
plt.figure()
plt.plot(mem_v.times, mem_v)
plt.xlabel("time [ms]")
plt.ylabel("membrane potential [LSB]")
plt.show()
```

CHAPTER

FIVE

LU.I EXPERIMENT

5.1 Overview

To get accustomed to the concept of neuromorphic computation, you will work with *Lu.i*. Lu.i is an electronic neuron circuit mimicking and illustrating the basic dynamics of real, biological neurons. The printed circuit board (PCB) features a configurable, fully analog implementation of the leaky integrate-and-fire model and visualizes the internal state, the membrane potential, through a VU-meter-style chain of LEDs. The neuron emits a short pulse whenever the membrane potential crosses a predefined threshold voltage. Neurons communicate by exchanging these spikes because multiple boards can be connected via jumper wires to form networks.

A picture of a Lu.i is shown in Fig. 5.1.

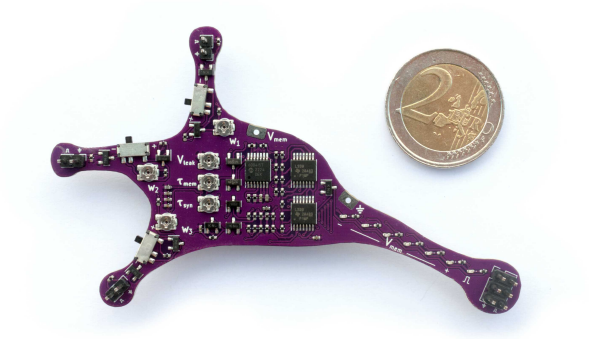

Fig. 5.1: Photograph of a Lu.i electronic neuron circuit, with a 2-euro coin included for scale.

You may find additional information including schematics in the main paper and on the project's homepage, giantaxon.org.

5.2 Exercises

5.2.1 Getting familiar with Lu.i's parameters

Before you begin, connect Lu.i's membrane to the oscilloscope.

- Familiarize yourself with the parameters of the neuron emulation. What happens if you increase the leak potential? Can you observe the membrane time constant? Note the impact of the individual potentiometers.
- Connect two or multiple neurons to form a simple network and note how signals propagate across neurons, e.g., while playing with the synaptic weights and the synaptic time constant.

- Pick a configuration where you can approximately measure τ_{ref} and the threshold ϑ. Take notes of your results and your methods of determining the individual parameters.

Note: Keep the setting for the membrane time constant τ_{mem} fixed for the following task.

5.2.2 Measuring of an f-I curve

Next, you will record your first f-I curve, i.e., the dependence of a neuron's firing rate on its stimulating current. For this purpose, you will rely on a voltage-controlled current source, depicted in Fig. 5.2.

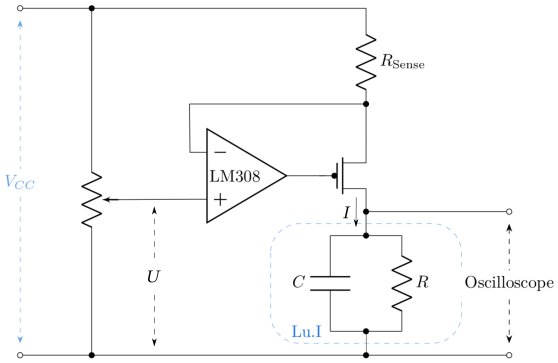

Fig. 5.2: Circuit diagram of the measurement setup for the Lu.i. board. A voltage-controlled current source injects current into the Lu.i., while the membrane potential is monitored using an oscilloscope.

The output current of this circuit is governed by the following equation:

$$I(U) = \frac{V_{\text{CC}} - U}{R_{\text{sense}}}$$

In the present case, R_{sense} is 10 kΩ and the supply voltage of the voltage-controlled current source V_{CC} is 5V. Note that this supply voltage is different from that of the Lu.i. board, which is 3.3V, as indicated on its battery.

Make sure that Lu.i's membrane is connected to both the oscilloscope and the current source. A second set of cables allows you to read out the voltage at the lower node of R_{sense} at the same time such that you can calculate the voltage drop across the resistor and thus the current.

- Set the leak potential at its minimum. Now measure about 20 firing frequencies as a function of the input current to Lu.i.'s membrane. Make sure that the controlling voltage drop U measured on the oscilloscope is in a range of $1.75V$ to $5V$.
- Plot your results.
- Derive an equation for $f_{\text{theo}}(I)$. Your function should depend on the chosen parametrization of the *LIF* equation, namely ϑ, C_m, R, and τ_{ref}. Here R is the membrane resistance of the Lu.i. board, the inverse of the membrane conductance, and C_m is the membrane capacitance. You might need to define your integration limits. (Hint 1: Consult equation (1.2) as a starting point; hint 2: You'll find $R \cdot I > \vartheta$ as a condition)
- Fit the function to your measurements. You might find it helpful to extract the membrane capacitance value from the circuit diagram.

CHAPTER

SIX

INVESTIGATING A SINGLE NEURON

In this task you can familiarise yourself with the way you will interact with the BrainScaleS-2 chip during the lab exercises. For all tasks there exist *jupyter* notebooks that are intended to be used as a basis for the results. If you are not familiar with *python* or *jupyter* notebooks before beginning this lab course, we recommend that you spend some time beforehand to familiarise yourself.

During the execution of some of the notebook cells, you will cause an experiment to be run on a BrainScaleS-2 system remotely. You will find that due to the analog nature of some of the system's implementation the execution is not deterministic, i.e., not each hardware execution will result in the same outcome.

Generally speaking the simplest hardware experiment can be divided in three steps

- hardware configuration
- hardware execution
- analysis of hardware observables.

More sophisticated experiments can involve multiple iterations of these basic three steps or a subset of them. At the level of abstraction you will be working on for these lab exercises, most of the intricacies of the hardware configuration and execution will be hidden behind a common high-level *API*, called *PyNN*.

In the next section we will set up a connection to an RPC server that multiplexes connections to the hardware, which will allow us to work with the system interactively. This process is handled by a custom microscheduler (*quiggeldy*), a conceptual view of which you can see in Fig. 6.1. The actual hardware execution time has been colored in blue.

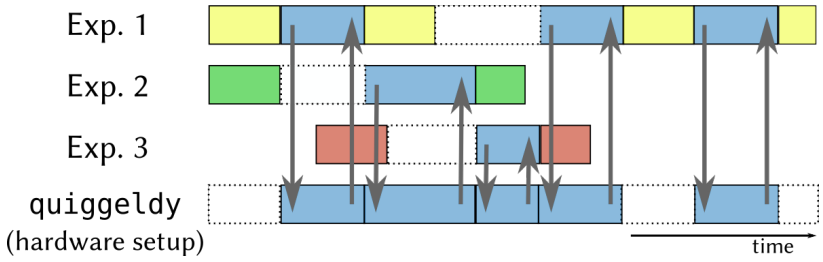

Fig. 6.1: Conceptual view of the custom microscheduler quiggeldy. The scheduler decouples hardware execution of experiments (blue) from required pre- and postprocessing time (yellow, green, red), thereby maximizing hardware utilization with minimal overhead for the individual users.

At the end of this notebook you will find a number of exercises to complete.

6.1 Experiment setup

Note: This print-friendly version of the instructions only lists the most central functions and code blocks. The interactive variant provides additional helpers.

For our first experiment, we create a single neuron and record its spikes as well as its membrane potential.

```
def experiment(title, v_leak, v_threshold, v_reset, i_bias_leak, savefig = False):
    """
    Set up a leak over threshold neuron.

    :param v_leak: Leak potential.
    :param v_threshold: Spike threshold potential.
    :param v_reset: Reset potential.
    :param i_bias_leak: Controls the leak conductance (membrane time constant).
    :param savefig: Save the experiment figure.
    """

    plt.figure()
    plt.title(title)

    # everything between pynn.setup() and pynn.end()
    # below is part of one hardware run.
    pynn.setup()

    # a pynn.Population corresponds to a certain number of
    # neuron circuits on the chip
    pop = pynn.Population(1, pynn.cells.HXNeuron(
        # Leak potential, range: 400-1000
        leak_v_leak=v_leak,
        # Leak conductance, range: 0-1022
        leak_i_bias=i_bias_leak,
        # Threshold potential, range: 0-500
        threshold_v_threshold=v_threshold,
        # Reset potential, range: 300-1000
        reset_v_reset=v_reset,
        # Membrane capacitance, range: 0-63
        membrane_capacitance_capacitance=63,
        # Refractory time (counter), range: 0-255
        refractory_period_refractory_time=255,
        # Enable reset on threshold crossing
        threshold_enable=True,
        # Reset conductance, range: 0-1022
        reset_i_bias=1022,
        # Increase reset conductance
        reset_enable_multiplication=True))

    pop.record(["v", "spikes"])

    # this triggers a hardware execution
    # with a duration of 0.2 ms
    pynn.run(0.2)
    plot_membrane_dynamics(pop, ylim=(100, 800))

    if savefig:
```

(continues on next page)

(continued from previous page)

```
        plt.savefig(results_folder.joinpath(f'fp_task1_{time.strftime("%Y%m%d-%H%M%S
↪")}.png'))

    plt.show()

    mem = pop.get_data("v").segments[-1].irregularlysampledsignals[0]
    spikes = pop.get_data("spikes").segments[-1].spiketrains[0]
    pynn.end()

    return mem, spikes
```

6.2 Exercises

6.2.1 A continuously spiking neuron

As a first exercise, set the neuron up such that it spikes continuously without any external input. For that purpose, play with the function's parameters and note down a suitable configuration in the following cell. Make sure to also save the resulting plot.

```
mem, spikes = experiment(
    title="A continuously spiking neuron",
    v_leak = 1000,
    v_threshold = 500,
    v_reset = 400,
    i_bias_leak = 150,
    savefig = True
)
```

6.2.2 Simple spike train statistics

- Calculate the neuron's average firing rate.
- Derive the inter-spike intervals (ISIs) of the spike train recorded in the previous exercise. The inter-spike interval denotes the time between two consecutive spikes of a neuron. Plot a histogram of the ISIs.
- Identify a method to calculate the mean and standard deviation of the neuron's instantaneous firing rate. The instantaneous firing rate provides a moment-by-moment measure of the neuron's activity.

Hint: You may use, e.g., np.diff, np.mean, and np.std to complete these tasks.

6.2.3 Recording the f-I curve of a silicon neuron

Next, we want to study the behavior of a single neuron stimulated with a current source. For this, we introduce a new parameter for our neuron model. This will allow to enable/disable a constant current source and observe the impact.

- Add *constant_current_enable* (True/False) and *constant_current_i_offset* (range=0-1022) as parameters into your experiment
- Configure a non firing neuron
- Vary the current and observe the impact

```
interact(
    experiment,
    title="Task 2: Current introduced firing",
    v_leak=IntSlider(min=400, max=1022, step=1, value=1000),
    v_threshold=IntSlider(min=0, max=500, step=1, value=500),
    v_reset=IntSlider(min=300, max=1022, step=1, value=400),
    i_bias_leak=IntSlider(min=0, max=1022, step=1, value=150),
    current=IntSlider(min=0, max=1022, step=1, value=300),
)
```

- To quantify your observations write a program that sweeps over a current range from 0 up to 800.
- Plot the neuron's mean instantaneous firing rate including its error over current.

6.2.4 Synaptic stimuli and PSP stacking

In the previous tasks, we analyzed the spiking dynamics of an isolated neuron. In this task, we want to look at the membrane response of the neuron to stimulations of incoming spikes. Your goal is to reproduce the image of PSP-Stacking (summation) from the introduction. For this, you might find it helpful to look into pynn_introduction again.

A remark about the calibration loaded above: While this calibration is not needed for the simple experiments above, it is essential when looking at synaptic inputs. Therefore, you should load it when setting up this experiment using: `pynn.setup(initial_config=calib)`.

Steps:

- Set up a population of one neuron with the default neuron parameters to record its membrane potential.
- Create a population of one *SpikeSourceArray* as a stimulating population.
- Connect the stimulating population to the neuron population.
- Find a suitable spike pattern to recreate the plot (you might require multiple spikes).
- Can the neuron spike with the current neuron configuration? Explain.
- Suggest and implement changes in the configuration so that the neuron spikes. Try to apply one change at a time. Explain your choices.
- Can you tell which changes can be theoretically equivalent?

Hint: Changes can be related to neuron parameters, projection, or population.

CHAPTER

SEVEN

A SIMPLE FEEDFORWARD NETWORK

In this task we consider a so-called synfire chain, an illustration of which can be seen in Fig. 7.1: A synfire chain is a chain of `numb_pops` "excitatory" populations (red), each consisting of n_{exc} neurons, which are connected via excitatory connections, such that events in a prior population excite the neurons in the following population. So, if the first population is activated, the activity begins traveling through the chain. In order to stop a population from firing as soon as it has excited the next population, it is also connected to an "inhibitory" population (blue) of n_{inh} neurons via inhibitory synapses. This inhibitory population is excited by the spike events from the previous excitatory population.

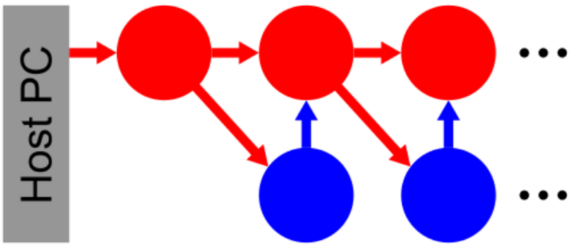

Fig. 7.1: Structure of the synfire chain presented in this section. The synfire chain is made up of several groups of excitatory (red) and inhibitory (blue) populations. The inhibitory population connects to the excitatory population within the same group. Each excitatory population is connected to the excitatory and inhibitory population of the next group. By repeating this construction schema chains of arbitrary length can be realized. The network is initially excited by an externally injected stimulus.

The number of neurons in the "excitatory" and "inhibitory" populations, as well as the strength of the connections in between needs to be chosen carefully in order for the activity to travel neatly. The chain can also be closed by connecting the last population to the first one, making the activity travel through the chain repetitively.

Note: This print-friendly version of the instructions only lists the most central functions and code blocks. The interactive variant provides additional helpers.

7.1 Experiment setup

You are given a scaffold to run the experiment where populations are created and connected, and finally also stimulated. This scaffold can be used to investigate the behaviour of a synfire chain and answer the questions below.

For this task, you will use the oscilloscope to visualize the membrane potential trace of the first neuron from the first excitatory population of your defined network on the BrainScaleS-2 chip. This is achieved by connecting its voltage to an external pad and connecting the oscilloscope to the pad. Our system offers two pads for the readout. We already set the experiment up such that the voltage is available at pad 0 (*pop1exc[[0]].record('v', device='pad_0_buffered')* from above), and connected this pad (named `ANA0` on the printed circuit board) to the oscilloscope for you.

7.2 Exercises

7.2.1 Adjusting weights

- Tune the weights to obtain a synfire chain behavior. Explain your strategy.
- Explain how the visualized plot relates to the synfire chain behavior.
- Which connection affects the synfire chain behavior the most?
- What happens if you disable inhibition?
- Visualize the membrane potential of the neuron using the oscilloscope. Include a picture of the obtained pulse in your protocol. *Hint*: You might find it useful to use the `normal` mode of the oscilloscope and assign a trigger to detect your pulse.

Make comments in your lab book.

After executing the cells above, you can execute this cell as oft as you want.

```
synapse_weights = dict(
    stim_exc=...,  # int in range 0 - 63
    stim_inh=...,  # int in range 0 - 63
    exc_exc=...,   # int in range 0 - 63
    exc_inh=...,   # int in range 0 - 63
    inh_exc=...    # int in range -63 - 0
)
set_network_weights(weights=synapse_weights, projections=projs)

results = run(pops, 0.2 * pq.ms)

plot_data(pops, *results)
```

7.2.2 Adjusting the number of neurons per population

Reduce the number of neurons in each population and use the free neurons to increase the chain size.

- Which hardware feature limits the minimal number of neurons in each population?
- What is the maximal chain length that you can produce?

Make comments in your lab book.

```
projs, pops = setup_network(
    numb_pops=...,                      # chain length
    pop_sizes={'exc': ..., 'inh': ...})  # size of each chain link

# we reuse the weights from the exercise above
set_network_weights(weights=synapse_weights, projections=projs)

results = run(pops, 0.2 * pq.ms)

plot_data(pops, *results)
```

7.2.3 Closing the loop

Close the loop from the last to the first population.

- Visualize the membrane potential on the oscilloscope. Do the neurons still fire after the software has completed?
- Note the change over time in the membrane potential trace, and include pictures of these changes.

Make comments in your lab book.

Hint: For this part it might be easier to switch to a smaller chain with larger populations.

Hint: Take a look into the code above. The closed loop is already implemented and just needs to be activated.

```
projs, pops = setup_network(...)

set_network_weights(weights=..., projections=projs)

results = run(pops, 0.2 * pq.ms)

plot_data(pops, *results)
```

CHAPTER

EIGHT

SOLVING SUDOKUS WITH BRAINSCALES-2

In this task we want to apply our newly gained knowledge to a well-known problem: We will deal with solving a sudoku on spiking hardware.

If you never before encountered this logic puzzle from 1984, let's define the rules. Usually you have a 9x9 square grid and the goal is to fill in numbers from 1 to 9, given crucial constraints:

1. In each row, each number is only allowed exactly once.
2. In each column, each number is only allowed exactly once.
3. In each 3x3 sub grid, each number is only allowed exactly once.

To find a valid solution, hints are given in form of given, fixed numbers. An obvious strategy for solving a sudoku is to, sequentially, find a square where only one number is allowed. This seems trivial but due to the high dimensionality finding a solution is a non-trivial task. However, checking the correctness of a potential solution is a simple task. To circumvent the slowness of a brute-force solution one can resort to a stochastic strategy. In fact, a spiking neuronal network (SNN) can accomplish this task in this manner quickly and with high precision.

The basic idea in creating a sudoku solver with an SNN is to assign to each possible number in each field one neuron, called one-hot encoding. In our case, each field would have four neurons representing the numbers 1 to 4, respectively. A neuron being active is interpreted as that field being filled with the respective number, consequently a given, hinted number is represented by a continuously spiking neuron. An excluding constraint will be realised with an inhibitory synapse due to the suppressing effect on the activity. To allow the network to explore different states, a random Poisson-distributed background noise is applied to all neurons. In addition, each neuron is excitatorily connected to itself to maintain possible activity.

Let's start with a simplified version of a sudoku with a 4x4 grid, as shown in Fig. 8.1.

Assume the grey three is given, applying the previously mentioned constraints we obtain:

- Due to 1, the purple threes are not allowed.
- Due to 2, the green threes are not allowed.
- Due to 3, the blue threes are not allowed.
- Due to the choice of our encoding, the orange numbers are not allowed (in each field only one number is allowed to be active)

For further information read into Solving the Constraint Satisfaction Problem Sudoku on Neuromorphic Hardware (MA Thesis).

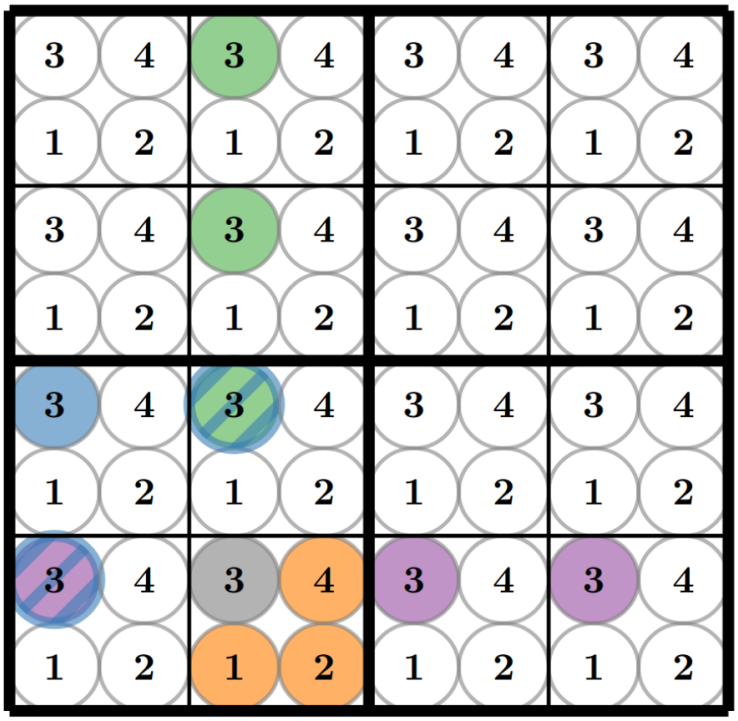

Fig. 8.1: Representation of a 4x4 sudoku with one-hot encoding: In each small square, each number from 1 to 4 has one respective neuron; four small squares, framed by the thick line, make one block in the original sudoku; there are four such blocks to complete the 4x4 sudoku.

8.1 Experiment setup

We start by creating a population with the required number of neurons. Additionally, we create a view for each field to facilitate easier access later `pops_collector[row][field][neuron]`

```
# Defining total runtime and dimensionality of sudoku
runtime = 0.5
dimension = 4

# -> we need 4 (rows) * 4 (columns) * 4 (numbers) = 4^3 neurons
pop = pynn.Population(dimension**3, HXNeuron())
pop.record("spikes")

# to define the connections easier, we save a "view" of each neuron in a list
pops_collector = []
for row in range(dimension):
    pops_row = []
    for field_in_row in range(dimension):
        pops_field = []
        for number_in_field in range(dimension):
            neuron = pynn.PopulationView(
                pop,
                [row * dimension**2 + field_in_row * dimension
                + number_in_field])
            pops_field.append(neuron)
        pops_row.append(pops_field)
```

(continues on next page)

(continued from previous page)

```
    pops_collector.append(pops_row)
```

```
# Create background noise
poisson_source = pynn.Population(dimension**3,
    SpikeSourcePoisson(duration=runtime - 0.01, rate=5e5, start=0.01))

# connect random sources with neurons
# additionally each neuron is connected to itself excitatorily to
# sustain possible activity

pynn.Projection(pop,
                pop,
                pynn.OneToOneConnector(),
                synapse_type=StaticSynapse(weight=20),
                receptor_type='excitatory')

pynn.Projection(poisson_source,
                pop,
                pynn.OneToOneConnector(),
                synapse_type=StaticSynapse(weight=30),
                receptor_type='excitatory')
```

```
# create stimulation for clues and connect to according neurons
stim_given_numbers = pynn.Population(
    1, SpikeSourceArray(spike_times=np.linspace(0.0, runtime, 500)))

clue_projections = []

for row in range(4):
    clues_row = []
    for column in range(4):
        clues_field = []
        for number in range(4):
            clues_field.append(pynn.Projection(
                stim_given_numbers,
                pops_collector[row][column][number],
                pynn.AllToAllConnector(),
                synapse_type=StaticSynapse(weight=0),
                receptor_type='excitatory'))
        clues_row.append(clues_field)
    clue_projections.append(clues_row)
```

```
# functions to solve the sudoku:

def set_clues(clues=None):
    """Sets the clues in the network."""
    if clues is None:
        clues = np.zeros((4, 4), dtype=int)
    for row, row_clues in enumerate(clue_projections):
        for col, field_clues in enumerate(row_clues):
            for number, clue_projection in enumerate(field_clues, start=1):
                for connection in clue_projection:
                    connection.weight = 63. if clues[row,col] == number else 0.

def hide_solution(grid, num_clues, seed=None):
```

(continues on next page)

(continued from previous page)

```
    """Hides the solution and only leaves `num_clues` hints."""
    indices = np.argwhere(np.logical_and(grid > 0, grid <= 4))
    if len(indices) < num_clues:
        raise RuntimeError(
            f"The sudoku has less than {num_clues} clues, which is the number of␣
↪required clues :(")
    np.random.seed(seed)
    indices = indices[np.random.choice(len(indices), num_clues, replace=False)]
    clues = np.zeros_like(grid)
    clues[(indices.T[0], indices.T[1])] = grid[(indices.T[0], indices.T[1])]
    return clues

def get_solution(runtime, clues):
    """Executes the network and returns the current solution."""
    set_clues(clues)
    grid = np.zeros((4, 4), dtype=int)

    # Define duration of poisson spikes
    poisson_source.set(duration=runtime - 0.01)

    # emulate the network
    pynn.reset()
    pynn.run(runtime)
    # read back solution
    for row, row_populations in enumerate(pops_collector):
        for col, field_populations in enumerate(row_populations):
            num_spikes = [
                len(num_population.get_data("spikes").segments[-1].spiketrains[0])
                for num_population in field_populations
            ]
            grid[row, col] = np.argmax(num_spikes) + 1
    return grid
```

```
# Constraints

# create inhibitory connections to neurons in the same field
# representing different numbers



# create inhibitory connections to neurons in the same row
# representing the same number



# create inhibitory connections to neurons in the same column
# representing the same number



# create inhibitory connections to neurons in the same block
# representing the same number
```

```
SP: SudokuPlotter = SudokuPlotter(dimension, pop, set_clues, hide_solution, get_
↪solution)
```

(continues on next page)

(continued from previous page)

```
# this sudoku shall be solved
global sudoku
sudoku = np.array([
    [3, 2, 4, 1],
    [1, 4, 3, 2],
    [2, 3, 1, 4],
    [4, 1, 2, 3]
])
```

```
# The cell below will only work, of course, if you implemented the correct␣
↪constraints above.
# Red/green frame show (in)correctness of the proposed solution (consistency with the␣
↪given sudoku).

def experiment(**kwargs):
    SP.solve_sudoku(sudoku, kwargs["runtime"], kwargs["num_clues"], kwargs["random_
↪seed"])
    SP.plot()
    # plt.savefig("solved_sudoku.png", backend="png", bbox_inches="tight")

build_gui(experiment, ["num_clues", "random_seed", "runtime"])
```

The hardware will try to solve the given sudoku. If you want to implement your own sudoku with unknown numbers you can enter *0* as an empty field. (*Hint*: When you change the sudoku, rerun the cell)

The hints are chosen randomly. By varying the seed you vary the position of the clue and by changing the number the sudoku changes the difficulty.

8.2 Exercises

- **Task 1:** Run the network without any constrains, observe the spike pattern and the solution of the sudoku. Try to explain.
- **Task 2:** Now implement the four constraint rules discussed above. If they are correctly implemented you should get a solved sudoku (keep number of hints and seed).

Now you have a working sudoku solver. Let's test it:

- **Task 3:** Vary the number of clues. If the solver fails can you find an explanation for that behavior? Can you think of possible strategies to reduce such errors? (*It might help to inspect the code and numpy documentation*)
- **Task 4:** What do you expect to happen, if you set the number of clues to zero? Check your hypothesis. Can you explain your observation?
- **Task 5:** Now, investigate how the success rate is related to the number of clues. For this, vary the number of clues from four to ten. Repeat each configuration ten times, while keeping the sudoku fixed. Visualize your result.
- **Task 6:** Is there a constraint that is not necessarily required? Why?
- **Task 7:** Crunching some numbers: How many neurons and how many synaptic connections were required in this task? How many would be required for a 9x9 sudoku?

CHAPTER

NINE

OPTIONAL EXPERIMENTS

There are further experiments that are not part of the usual course, but can be both very instructive and helpful in case you want to do a long/short report. These experiments include working on the calibration, studying the synaptic input, dealing with more complex neuron models, and training neural networks. If you are interested, you can have a look online at the Electronic Visions github page.

CHAPTER

TEN

REFERENCES

If you are curious and want to learn more about the kind of work that has been done with the system, here are a few references that you can check out

- The BrainScaleS-2 Accelerated Neuromorphic System With Hybrid Plasticity
- Surrogate gradients for analog neuromorphic computing
- Versatile emulation of spiking neural networks on an accelerated neuromorphic substrate
- hxtorch: PyTorch for BrainScaleS-2 – Perceptrons on Analog Neuromorphic Hardware
- Control of criticality and computation in spiking neuromorphic networks with plasticity
- Demonstrating Advantages of Neuromorphic Computation: A Pilot Study
- Fast and energy-efficient neuromorphic deep learning with first-spike times
- Inference with Artificial Neural Networks on Analog Neuromorphic Hardware
- Spiking neuromorphic chip learns entangled quantum states
- Structural plasticity on an accelerated analog neuromorphic hardware system
- Emulating dendritic computing paradigms on analog neuromorphic hardware
- Accelerated analog neuromorphic computing
- Demonstrating hybrid learning in a flexible neuromorphic hardware system
- A mixed-signal structured AdEx neuron for accelerated neuromorphic cores
- Demonstrating analog inference on the brainscales-2 mobile system
- Demonstrating brainscales-2 inter-chip pulse-communication using extoll
- Extending brainscales OS for BrainScaleS-2
- PyNN: a common interface for neuronal network simulators
- The BrainScaleS-2 multi-chip system: Interconnecting continuous-time neuromorphic compute substrates

CHAPTER

ELEVEN

GLOSSARY

LIF Leaky integrate-and-fire.

AdEx Adaptive exponential integrate-and-fire.

ADC Analog-to-digital converter.

DAC Digital-to-analog converter.

CADC Column-wise analog-to-digital converter.

ASIC Application-specific integrated circuit.

PPU Plasticity processing unit.

FPGA Field-programmable gate array.

SRAM Static random-access memory.

MOSFET Metal–oxide–semiconductor field-effect transistor.

PCB Printed circuit board.

IO Input/output.

SIMD Single instruction, multiple data.

API Application programming interface.

ANN Artificial neural network.

SNN Spiking neural network.